GBASE

GBase 8c 数据库运维指南

www.gbase.cn

GBase 8c数据库运维指南，南大通用数据技术股份有限公司

通讯方式

南大通用数据技术股份有限公司

天津市高新区华苑产业园区工华道2号天百中心3层(300384)

电话：400-013-9696　　邮箱：info@gbase.cn

目　录

1 数据库启停

启动 GBase 8c

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用以下命令启动 GBase 8c。

```
gs_om -t start
```

说明

- 双机启动必须以双机模式启动，若中间过程以单机模式启动，则必须修复才能恢复双机关系，用 gs_ctl build 进行修复，gs_ctl 的使用方法请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“系统内部命令 > gs_ctl”章节。

停止 GBase 8c

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用以下命令停止 GBase 8c。

```
gs_om -t stop
```

说明

- 启停节点及 AZ 的操作请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_om”章节。

示例

启动 GBase 8c：

```
gs_om -t start
Starting cluster.
=========================================
=========================================
Successfully started.
```

停止 GBase 8c：

```
gs_om -t stop
Stopping cluster.
=========================================
Successfully stopped cluster.
=========================================
End stop cluster.
```

错误排查

如果启动或者停止 GBase 8c 服务失败，请根据日志文件中的日志信息排查错误，参见日志参考。

如果是超时导致启动失败，可以设置延长启动超时时间，默认超时时间为 300s。例如，设置超时时间为 600s：

```
gs_om -t start --time-out=600
```

2 查看状态

背景信息

GBase 8c 支持查看整个集群状态，通过查询结果确认集群或单主机的运行状态是否正常。

前提条件

GBase 8c 已经正常安装并启动。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令查询 GBase 8c 状态：

```
gs_om -t status --detail
```

状态显示的参数说明请参见表 2-1。

若要查询某主机上的实例状态，请增加-h 参数，指定待查询主机名。例如，查询 plat2 主机上实例状态，执行命令：

```
gs_om -t status -h plat2
```

参数说明

表 2-1 节点角色说明

字段	字段含义	字段值
cluster_state	GBase 8c 状态。显示整个 GBase 8c 是否运行正常。	Normal：表示 GBase 8c 可用，且数据有冗余备份。所有进程都在运行，主备关系正常。 Unavailable：表示 GBase 8c 不可用。 Degraded：表示 GBase 8c 可用，但存在故障的数据库节点、数据库主节点实例。
node	主机名称	表示该实例所在的主机名称。多 AZ 时会显示 AZ 编号。
node_ip	主机 IP	表示该实例所在的主机 IP。
instance	实例 ID	表示该实例的 ID。

字段	字段含义	字段值
state	实例角色	Normal：表示单主机实例。 Primary：表示实例为主实例。 Standby：表示实例为备实例。 Cascade Standby：表示实例为级联备实例。 Pending：表示该实例在仲裁阶段。 Unknown：表示实例状态未知。 Down：表示实例处于宕机状态。 Abnormal：表示节点处于异常状态。 Manually stopped：表示节点已经被手动停止。

每个角色也存在不同的状态，例如启动、连接等，其各个状态说明如下：

表 2-2 节点状态说明

状态	字段含义
Normal	表示节点启动正常
Need repair	当前节点需要修复
Starting	节点正在启动中
Wait promoting	节点正等待升级中，例如备机向主机发送升级请求后，正在等待主机回应时的状态
Promoting	备节点正在升级为主节点的状态
Demoting	节点正在降级中，如主机正在降为备机中
Building	备机启动失败，需要重建
Catchup	备节点正在追赶主节点
Coredump	节点程序崩溃
Unknown	节点状态未知

当节点出现 Need repair 状态时，可能需要对该节点进行重建使其恢复正常。通常情况

下，节点重建原因说明如下：

表 2-3 节点重建原因说明

状态	字段含义
Normal	表示节点启动正常
WAL segment removed	主机日志/WAL 日志不存在，或者备机日志比主机日志新
Disconnect	备机不能连接主机
Version not matched	主备二进制版本不一致
Mode not matched	主备角色不匹配，例如两个备机互联
System id not matched	主备数据库系统 id 不一致，主备双机要求 System ID 必须一致
Timeline not matched	日志时间线不一致
Unknown	其他原因

示例

查看 GBase 8c 集群详细状态信息，含实例状态信息。

```
$ gs_om -t status --detail
[   Cluster State   ]

cluster_state   : Normal
redistributing  : No
current_az      : AZ_ALL

[  Datanode State   ]

    node         node_ip         port      instance
state
-----------------------------------------------------------------------------
-------------------------
1  gbase8c_5_124 172.16.5.124    15400      6001 /opt/database/install/data/dn
P Primary Normal
```

3 例行维护

3.1 日维护检查项

3.1.1 检查数据库状态

通过 GBase 8c 提供的工具查询数据库和实例状态，确认数据库和实例都处于正常的运行状态，可以对外提供数据服务。

- 检查实例状态

```
$ gs_check -U gbase -i CheckClusterState
```

- 检查参数

```
SHOW parameter_name;
```

上述命令中，parameter_name 需替换成具体的参数名称。

- 修改参数

```
$ gs_guc reload  -D datadir -c "paraname=value"
```

3.1.2 检查锁信息

锁机制是数据库保证数据一致性的重要手段，检查相关信息可以检查数据库的事务和运行状况。

- 查询数据库中的锁信息

```
postgres=# SELECT * FROM pg_locks;
```

- 查询等待锁的线程状态信息

```
postgres=# SELECT * FROM pg_thread_wait_status WHERE wait_status = 'acquire lock';
```

- 结束系统进程

查找正在运行的系统进程，然后使用 kill 命令结束此进程。

```
$ ps ux
$ kill -9 pid
```

3.1.3 统计事件数据

SQL 语句长时间运行会占用大量系统资源，用户可以通过查看事件发生的时间，占用

内存大小来了解现在数据库运行状态。

- 查询事件的时间

  查询事件的线程启动时间、事务启动时间、SQL 启动时间以及状态变更时间。

```
postgres=# SELECT backend_start,xact_start,query_start,state_change FROM
pg_stat_activity;
```

- 查询当前服务器的会话计数信息

```
postgres=# SELECT count(*) FROM pg_stat_activity;
```

- 查询系统级统计信息

  查询当前使用内存最多的会话信息。

```
postgres=# SELECT * FROM pv_session_memory_detail() ORDER BY usedsize desc limit
10;
```

3.1.4 对象检查

表、索引、分区、约束等是数据库的核心存储对象，其核心信息和对象维护是 DBA 重要的日常工作。

- 查看表的详细信息

```
postgres=# \d+ table_name
```

- 查询表统计信息

```
postgres=# SELECT * FROM pg_statistic;
```

- 查看索引的详细信息

```
postgres=# \d+ index_name
```

- 查询分区表信息

```
postgres=# SELECT * FROM pg_partition;
```

- 收集统计信息

  使用 ANALYZE 语句收集数据库相关的统计信息。

  使用 VACUUM 语句可以回收空间并更新统计信息。

- 查询约束信息

```
postgres=# SELECT * FROM pg_constraint;
```

3.1.5 SQL 报告检查

使用 EXPLAIN 语句查看执行计划。

3.1.6 备份

数据备份重于一切，日常应检查备份执行情况，并检查备份有效性，确保备份能够保障数据安全，备份安全加密也应兼顾。

- 指定用户导出数据库

```
gs_dump dbname -p port -f out.sql -U user_name -W password
```

- 导出 schema

```
gs_dump dbname -p port -n schema_name -f out.sql
```

- 导出 table

```
gs_dump dbname -p port -t table_name -f out.sql
```

3.1.7 基本信息检查

基本信息包括版本、组件、补丁集等信息，定期检查数据库信息并登记在案是数据库生命周期管理的重要内容之一。

- 版本信息

```
postgres=# SELECT version();
```

- 容量检查

```
postgres=# SELECT pg_table_size('table_name');
postgres=# SELECT pg_database_size('database_name');
```

3.2 检查操作系统参数

3.2.1 检查办法

通过 GBase 8c 提供的 gs_checkos 工具可以完成操作系统状态检查。

前提条件

- 当前的硬件和网络环境正常。
- 各主机间 root 互信状态正常。

- 只能使用 root 用户执行 gs_checkos 命令。

操作步骤

(1) 以 root 用户身份登录任意一台服务器。

(2) 执行如下命令对 GBase 8c 节点服务器的 OS 参数进行检查。

```
gs_checkos -i A
```

检查节点服务器的 OS 参数的目的是保证 GBase 8c 正常通过预安装，并且在安装成功后可以安全高效的运行。详细的检查项目请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_checkos”章节。

示例

执行 gs_checkos 前需要先使用 gs_preinstall 工具执行前置脚本，准备环境。

- 以参数"A"为例。

```
[root@gbase8c ~]$ gs_checkos -i A
```

执行返回如下信息：

```
Checking items:
    A1. [ OS version status ]                                   : Normal
    A2. [ Kernel version status ]                               : Normal
    A3. [ Unicode status ]                                      : Normal
    A4. [ Time zone status ]                                    : Normal
    A5. [ Swap memory status ]                                  : Normal
    A6. [ System control parameters status ]                    : Normal
    A7. [ File system configuration status ]                    : Normal
    A8. [ Disk configuration status ]                           : Normal
    A9. [ Pre-read block size status ]                          : Normal
    A10.[ IO scheduler status ]                                 : Normal
    A11.[ Network card configuration status ]                   : Normal
    A12.[ Time consistency status ]                             : Warning
    A13.[ Firewall service status ]                             : Normal
    A14.[ THP service status ]                                  : Normal
Total numbers:14. Abnormal numbers:0. Warning number:1.
```

- 以参数"B"为例。

```
[root@gbase8c ~]$ gs_checkos -i B
```

执行返回如下信息：

```
Setting items:
    B1. [ Set system control parameters ]                    : Normal
    B2. [ Set file system configuration value ]              : Normal
    B3. [ Set pre-read block size value ]                    : Normal
    B4. [ Set IO scheduler value ]                           : Normal
    B5. [ Set network card configuration value ]             : Normal
    B6. [ Set THP service ]                                  : Normal
    B7. [ Set RemoveIPC value ]                              : Normal
    B8. [ Set Session Process ]                              : Normal
Total numbers:6. Abnormal numbers:0. Warning number:0.
```

3.2.2 异常处理

使用 gs_checkos 检查 GBase 8c 状态时，可以使用--detail 参数，来查看详细的错误信息。

返回检查状态：

- Abnormal 为必须处理项，影响 GBase 8c 的安装。
- Warning 可以不处理，不会影响 GBase 8c 的安装。
- Normal 表示正常，支持安装。

返回检查项：

- 如果操作系统版本（A1）检查项检查结果为 Abnormal，需要将不属于混编范围的操作系统版本替换为混编范围内的操作系统版本。
- 如果内核版本（A2）检查项检查结果为 Warning，则表示 GBase 8c 内平台的内核版本不一致。
- 如果 Unicode 状态（A3）检查项检查结果为 Abnormal，需要将各主机的字符集设置为相同的字符集。编辑/etc/profile 文件实现。

```
vim /etc/profile
```

在打开的配置文件中，添加如下内容：

```
export LANG=unicode
```

其中 unicode 为 Unicode 编码方式，根据实际情况指定。输入“:wq”保存并退出。

- 如果时区状态（A4）检查项检查结果为 Abnormal，需要将各主机的时区设置为相同时

区。可以将/usr/share/zoneinfo/目录下的时区文件拷贝为/etc/localtime 文件。

```
cp /usr/share/zoneinfo/zonefile /etc/localtime
```

其中 zonefile 为需要使用的时区文件名，根据实际情况指定。

- 如果交换内存状态（A5）检查项检查结果为 Abnormal，可能是因为 swap 空间大于 mem 空间。可减小 Swap 解决或者增大 Mem 空间解决。
- 如果系统控制参数（A6）检查项检查结果为 Abnormal，可使用以下两种方法进行设置。
  - 第一种：使用如下命令进行设置。

    ```
    gs_checkos -i B1
    ```

  - 第二种：根据错误提示信息，在/etc/sysctl.conf 文件中进行设置。

    ```
    vim /etc/sysctl.conf
    ```

    修改配置文件，输入“:wq”保存并退出。

    并执行如下命令，使配置修改后生效。

    ```
    sysctl -p
    ```

- 如果文件系统配置状态（A7）检查项检查结果为 Abnormal，可以使用如下命令进行设置。

```
gs_checkos -i B2
```

- 如果磁盘配置状态（A8）检查项检查结果为 Abnormal，需修改磁盘挂载格式为：“rw,noatime,inode64,allocsize=16m”。

  使用 linux 的 man mount 命令挂载 XFS 选项：

```
rw,noatime,inode64,allocsize=16m
```

也可以在/etc/fstab 文件中设定 XFS 选项。如下示例：

```
/dev/data /data xfs rw,noatime,inode64,allocsize=16m 0 0
```

- 如果预读块大小（A9）检查项检查结果为 Abnormal，可使用如下命令进行设置。

```
gs_checkos -i B3
```

- 如果 IO 调度状态（A10）检查项检查结果为 Abnormal，可使用如下命令进行设置。

```
gs_checkos -i B4
```

- 如果网卡配置状态（A11）检查项检查结果为 Warning，可使用如下命令进行设置。

```
gs_checkos -i B5
```

- 如果时间一致性（A12）检查项检查结果为 Abnormal，需检查是否安装 ntp 服务，以及 ntp 服务是否启动；并与 ntp 时钟源同步。
- 如果防火墙状态（A13）检查项检查结果为 Abnormal，需关闭防火墙服务。使用如下命令进行设置。以 CentOS7、openEuler 操作系统为例：

```
systemctl stop firewalld
systemctl disable firewalld
```

- 如果 THP 服务（A14）检查项检查结果为 Abnormal，可使用如下命令进行设置。

```
gs_checkos -i B6
```

3.3 检查 GBase 8c 健康状态

3.3.1 检查办法

通过 GBase 8c 提供的 gs_check 工具可以开展 GBase 8c 健康状态检查。

注意事项

- 扩容新节点检查只能在 root 用户下执行，其他场景都必须在 gbase 用户下执行。
- 必须指定-i 或-e 参数，-i 会检查指定的单项，-e 会检查对应场景配置中的多项。
- 如果-i 参数中不包含 root 类检查项或-e 场景配置列表中没有 root 类检查项，则不需要交互输入 root 权限的用户及其密码。
- 可使用 -skip-root-items 跳过检查项中包含的 root 类检查，以免需要输入 root 权限用户及密码。
- 检查扩容新节点与现有节点之间的一致性，在现有节点执行 gs_check 命令指定 -hosts 参数进行检查，其中 hosts 文件中需要写入新节点 ip。

操作步骤

方式 1：

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 执行如下命令对 GBase 8c 数据库状态进行检查。

```
gs_check -i CheckClusterState
```

其中，-i 指定检查项，注意区分大小写。格式：

```
-i CheckClusterState、-i CheckCPU 或-i CheckClusterState，CheckCPU。
```

取值范围为所有支持的检查项名称，详细列表请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_checkos > GBase 8c 状态检查表”，用户可以根据需求自己编写新检查项。

方式 2：

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 执行如下命令对 GBase 8c 数据库进行健康检查。

```
gs_check -e inspect
```

其中，-e 指定场景名，注意区分大小写。格式：-e inspect 或-e upgrade。

取值范围为所有支持的巡检场景名称，默认列表包括：inspect（例行巡检）、upgrade（升级前巡检）、install（安装）、binary_upgrade（就地升级前巡检）、health（健康检查巡检）、slow_node（节点）、longtime（耗时长巡检），用户可以根据需求自己编写场景。

GBase 8c 巡检的主要作用是在 GBase 8c 运行过程中，检查整个 GBase 8c 状态是否正常，或者重大操作前（升级、扩容），确保 GBase 8c 满足操作所需的环境条件和状态条件。详细的巡检项目和场景请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_checkos > GBase 8c 状态检查表”。

示例

- 执行单项检查结果：

```
$ gs_check -i CheckCPU
```

返回如下信息：

```
Parsing the check items config file successfully
Distribute the context file to remote hosts successfully
Start to health check for the cluster. Total Items:1 Nodes:1

Checking...                [=========================] 1/1
Start to analysis the check result
CheckCPU.....................................OK
The item run on 1 nodes.  success: 1
```

```
Analysis the check result successfully
Success.        All check items run completed. Total:1   Success:1
For more information please refer to
/opt/database/install/om/script/gspylib/inspection/output/CheckReport_2024030
15575810033.tar.gz
```

- 本地执行结果：

```
$ gs_check -i CheckCPU -L
```

返回如下信息：

```
2024-03-01 15:30:13 [NAM] CheckCPU
2024-03-01 15:30:13 [STD] 检查主机 CPU 占用率，如果 idle 大于 30%,或者 iowait 小
于 30%.则检查项通过，否则检查项不通过
2024-03-01 15:30:13 [RST] OK

2024-03-01 15:30:13 [RAW]
Linux 3.10.0-1127.el7.x86_64 (gbase8c_5_124)     03/01/24        _x86_64_
(2 CPU)

15:30:08
CPU     %user     %nice   %system   %iowait    %steal     %idle
15:30:09        all      2.53      0.00      0.00      0.00      0.00
97.47
15:30:10        all      4.98      0.00      0.50      0.00      0.00
94.53
15:30:11        all      3.02      0.00      1.01      0.00      0.00
95.98
15:30:12        all      2.51      0.00      0.00      0.00      0.00
97.49
15:30:13        all      4.00      0.00      0.50      0.00      0.00
95.50
Average:        all      3.41      0.00      0.40      0.00      0.00
96.19
```

- 执行场景检查，期间需键入具有 root 权限的用户及登录密码：

```
$ gs_check -e inspect
```

返回如下信息：

```
Parsing the check items config file successfully
```

```
The below items require root privileges to execute:[CheckBlockdev
CheckIOrequestqueue CheckIOConfigure CheckMTU CheckRXTX CheckMultiQueue
CheckFirewall CheckSshdService CheckSshdConfig CheckCrondService
CheckMaxProcMemory CheckBootItems CheckFilehandle CheckNICModel CheckDropCache]
Please enter root privileges user[root]:root
Please enter password for user[root]:
Check root password connection successfully
Distribute the context file to remote hosts successfully
Start to health check for the cluster. Total Items:57 Nodes:1

Checking...               [========================] 57/57
Start to analysis the check result
CheckClusterState.......................OK
The item run on 1 nodes.  success: 1

CheckDBParams...........................OK
The item run on 1 nodes.  success: 1

CheckDebugSwitch........................OK
The item run on 1 nodes.  success: 1

CheckDirPermissions.....................OK
The item run on 1 nodes.  success: 1

CheckReadonlyMode.......................OK
The item run on 1 nodes.  success: 1

CheckEnvProfile.........................NG
The item run on 1 nodes.  ng: 1
The ng[gbase8c_5_124] value:
GAUSSHOME        /opt/database/install/app
GBase8cV5 lib path does not exist in LD_LIBRARY_PATH.
PATH             /opt/database/install/app/bin


CheckBlockdev...........................OK
The item run on 1 nodes.  success: 1

CheckCurConnCount.......................OK
The item run on 1 nodes.  success: 1
```

```
CheckCursorNum.............................OK
The item run on 1 nodes.  success: 1

CheckPgxcgroup.............................OK
The item run on 1 nodes.  success: 1

CheckDiskFormat............................OK
The item run on 1 nodes.  success: 1

CheckSpaceUsage............................OK
The item run on 1 nodes.  success: 1

CheckInodeUsage............................OK
The item run on 1 nodes.  success: 1

CheckSwapMemory.........................WARNING
The item run on 1 nodes.  warning: 1
The warning[gbase8c_5_124] value:
SwapMemory(4160745472) must be 0.
MemTotal: 8201502720.

CheckLogicalBlock..........................OK
The item run on 1 nodes.  success: 1

CheckIOrequestqueue........................OK
The item run on 1 nodes.  success: 1

CheckMaxAsyIOrequests...................WARNING
The item run on 1 nodes.  warning: 1
The warning[gbase8c_5_124] value:
Asy IO requests 65536  expectedScheduler 104857600.


CheckIOConfigure...........................OK
The item run on 1 nodes.  success: 1

CheckMTU...................................OK
The item run on 1 nodes.  success: 1  (consistent)
The success on all nodes value:
1500
```

```
......

Analysis the check result successfully
Failed. All check items run completed. Total:57   Success:44   Warning:4   NG:8
Error:1
For more information please refer to
/opt/database/install/om/script/gspylib/inspection/output/CheckReport_inspect
_20240301559791868 2.tar.gz
```

3.3.2 异常处理

如果发现检查结果异常，可以根据以下内容进行修复。

表 3-1 检查 GBase 8c 运行状态

检查项	异常状态	处理方法
CheckClusterState（检查 GBase 8c 状态）	GBase 8c 未启动或 GBase 8c 实例未启动	使用以下命令启动 GBase 8c 及实例。 gs_om -t start
	GBase 8c 状态异常或 GBase 8c 实例异常	检查各主机、实例状态，根据状态信息进行排查。 gs_check -i CheckClusterState
CheckDBParams（检查数据库参数）	数据库参数错误	通过 gs_guc 工具修改数据库参数为指定值。
CheckDebugSwitch（检查调试日志）	日志级别不正确	使用 gs_guc 工具将 log_min_messages 改为指定内容。
CheckDirPermissions（检查目录权限）	路径权限错误	修改对应目录权限为指定数值（750/700）。 chmod 700 DIR
CheckReadonlyMode（检查只读模式）	只读模式被打开	确认数据库节点所在磁盘使用率未超阈值（默认 85%）且未在执行其他运维操作。 gs_check -i CheckDataDiskUsageps ux 使用 gs_guc 工具关闭 GBase 8c 只读模式。

检查项	异常状态	处理方法
		gs_guc reload -N all -I all -c 'default_transaction_read_only = off'
CheckEnvProfile（检查环境变量）	环境变量不一致	重新执行前置更新环境变量信息。
CheckBlockdev（检查磁盘预读块）	磁盘预读块大小不为16384	使用 gs_checkos 设置预读块大小为 16384KB，并写入自启动文件。 gs_checkos -i B3
CheckCursorNum（检查游标数）	检查游标数失败	检查数据库能否正常连接，GBase 8c 状态是否正常。
CheckPgxcgroup（检查重分布状态）	有未完成重分布的pgxc_group 表	继续完成扩容或缩容的数据重分布操作。 gs_expand、gs_shrink
CheckDiskFormat（检查磁盘配置）	各节点磁盘配置不一致	将各节点的磁盘规格改为相同。
CheckSpaceUsage（检查磁盘空间使用率）	磁盘可用空间不足	清理或扩展对应目录所在的磁盘。
CheckInodeUsage（检查磁盘索引使用率）	磁盘可用索引不足	清理或扩展对应目录所在的磁盘。
CheckSwapMemory（检查交换内存）	交换内存大于物理内存	将交换内存调小或关闭。
CheckLogicalBlock（检查磁盘逻辑块）	磁盘逻辑块大小不为512	使用 gs_checkos 修改磁盘逻辑块大小为512KB，并写入开机自启动文件。 gs_checkos -i B4
CheckIOrequestqu	IO 请求值不为 32768	使用 gs_checkos 设置 IO 请求值为 32768，并写

检查项	异常状态	处理方法
eue(检查 IO 请求)		入开机自启动文件。 gs_checkos -i B4
CheckCurConnCount（检查当前连接数）111	当前连接数超过最大连接数的 90%	断开未使用的数据库主节点连接。
CheckMaxAsyIORequests (检查最大异步请求)	最大异步请求值小于 104857600 或当前节点数据库实例数乘以 1048576	使用 gs_checkos 设置最大异步请求值为 104857600 和当前节点数据库实例数乘以 1048576 中的最大值。 gs_checkos -i B4
CheckMTU（检查 MTU 值）	MTU 值不一致	设置各节点的 MTU 一致为 1500 或 8192。 ifconfig eth* MTU 1500
CheckIOConfigure（检查 IO 配置）	IO 配置不是 deadline	使用 gs_checkos 设置 IO 配置为 deadline，并写入开机自启动文件。 gs_checkos -i B4
CheckRXTX （检查 RXTX 值）	网卡 RX/TX 值不是 4096	使用 checkos 设置 GBase 8c 使用的物理网卡 RX/TX 值为 4096。 gs_checkos -i B5
CheckPing （检查网络通畅）	存在 GBase 8c IP 无法 ping 通	检查异常 ip 间网络设置和状态、防火墙状态。
CheckNetWorkDrop (检查网络丢包率)	网络通信丢包率高于 1%	检查对应 IP 间网络负载、状态。
CheckMultiQueue（检查网卡多队列）	未开启网卡多队列并未将网卡中断绑定到不同 CPU 核心	开启网卡多队列并将网卡队列中断绑定到不同的 CPU 核心。
CheckEncoding（检查编码格式）	各节点编码格式不一致	在/etc/profile 中写入一致的编码信息。

检查项	异常状态	处理方法
		echo "export LANG=XXX" >> /etc/profile
CheckFirewall（检查防火墙）	防火墙未关闭	关闭防火墙服务。 systemctl disable firewalld.servicesystemctl stop firewalld.service
CheckMaxHandle（检查最大文件句柄数）	最大文件句柄数小于1000000	设置 91-nofile.conf/90-nofile.conf 最大文件句柄数软硬限制为 1000000。 gs_checkos -i B2
CheckNTPD(检查时间同步服务)	NTPD 服务未开启或时间误差超过一分钟	开启 NTPD 服务并设置时钟一致。
CheckSysParams（检查操作系统参数）	操作系统参数设置不满足要求	使用 gs_checkos 进行参数设置或手动设置。 gs_checkos -i B1vim /etc/sysctl.conf
CheckTHP（检查 THP 服务）	THP 服务未开启	使用 gs_checkos 设置 THP 服务。 gs_checkos -i B6
CheckTimeZone（检查时区）	时区不一致	设置各节点为同一时区。 cp /usr/share/zoneinfo/$ 主时区/$ 次时区 /etc/localtime
CheckCPU（检查 CPU）	CPU 占用过高或 IO 等待过高	进行 CPU 配置升级或磁盘性能升级。
CheckSshdService（检查 SSHD 服务）	未开启 SSHD 服务	启动 SSHD 服务并写入开机自启动文件。 service sshd startecho "server sshd start" >> initFile
CheckSshdConfig（检查 SSHD 配置）	SSHD 服务配置错误	设置 SSHD 服务， PasswordAuthentication=no;MaxStartups=1000;UseDNS=yes;ClientAliveInterval=10800/ClientAliveInterval=0

检查项	异常状态	处理方法
		并重启服务： server sshd start
CheckCrondService（检查 Crond 服务）	Crond 服务未启动	安装 Crond 服务并启用。
CheckStack（检查堆栈大小）	堆栈大小小于 3072	使用 gs_checkos 设置为 3072 并重启堆栈值过小进程。 gs_checkos -i B2
CheckSysPortRange（检查系统端口设置）	系统 ip 端口不在预期范围内或 GBase 8c 端口在系统 ip 端口内	设置系统 ip 端口范围参数到 26000-65535 之中；设置 GBase 8c 端口在系统 ip 端口范围外。 vim /etc/sysctl.conf
CheckMemInfo（检查内存信息）	各节点内存大小不一致	使用相同规格的物理内存。
CheckHyperThread（检查超线程）	未开启 CPU 超线程	开启 CPU 超线程。
CheckTableSpace（检查表空间）	表空间路径和 GBase 8c 路径存在嵌套或表空间路径相互存在嵌套	将表空间数据迁移到路径合法的表空间中。

3.4 检查数据库性能

3.4.1 检查办法

通过 GBase 8c 提供的性能统计工具 gs_checkperf 可以对硬件性能进行检查。

前提条件

- GBase 8c 运行状态正常。
- 运行在数据库之上的业务运行正常。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 执行如下命令对 GBase 8c 数据库进行性能检查。

```
gs_checkperf
```

具体的性能统计项目请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_checkperf > 性能检查项”。

示例

以简要格式在屏幕上显示性能统计结果。

```
gs_checkperf -i pmk -U gbase
Cluster statistics information:
    Host CPU busy time ratio                    :    1.43        %
    MPPDB CPU time % in busy time               :    1.88        %
    Shared Buffer Hit ratio                     :    99.96       %
    In-memory sort ratio                        :    100.00      %
    Physical Reads                              :    4
    Physical Writes                             :    25
    DB size                                     :    70          MB
    Total Physical writes                       :    25
    Active SQL count                            :    2
    Session count                               :    3
```

3.4.2 异常处理

使用 gs_checkperf 工具检查 GBase 8c 性能状态后，如果发现检查结果发现异常，可以根据以下内容进行修复。

表 3-2 检查 GBase 8c 级别性能状态

异常状态	处理方法
主机 CPU 占有率高	1、更换和增加高性能的 CPU。 2、使用 top 命令查看系统哪些进程的 CPU 占有率高，然后使用 kill 命令关闭没有使用的进程。 top

异常状态	处理方法
GBase 8c CPU 占有率高	1、更换和增加高性能的 CPU。 2、使用 top 命令查看数据库哪些进程的 CPU 占有率高，然后使用 kill 命令关闭没有使用的进程。 top 3、使用 gs_expand 工具扩容，增加新的主机均衡 CPU 占有率。
共享内存命中率低	1、扩大内存。 2、使用如下命令查看操作系统配置文件/etc/sysctl.conf，调大共享内存 kernel.shmmax 值。 vim /etc/sysctl.conf
内存中排序比率低	扩大内存。
I/O、磁盘使用率高	1、更换高性能的磁盘。 2、调整数据布局，尽量将 I/O 请求较合理的分配到所有物理磁盘中。 3、全库进行 VACUUM FULL 操作。 vacuum full; 4、进行磁盘整理，参考上面执行全库 vacuum full 或针对性做单表 vacuum/vacuum full 操作。 5、降低并发数。
事务统计	查询 pg_stat_activity 系统表，将不必要的连接断开。（登录数据库后查询：postgres=# \d+ pg_stat_activity;）

表 3-3 检查节点级别性能状态

异常状态	处理方法
CPU 占有率高	1、更换和增加高性能的 CPU。 2、使用 top 命令查看系统哪些进程的 CPU 占有率高，然后使用 kill 命令关闭没有使用的进程。 top

异常状态	处理方法
内存使用率过高情况	扩大或清理内存。
I/O 使用率过高情况	1、更换高性能的磁盘。 2、进行磁盘清理。 3、尽可能用内存的读写代替直接磁盘 I/O，使频繁访问的文件或数据放入内存中进行操作处理。

表 3-4 会话/进程级别性能状态

异常状态	处理方法
CPU、内存、I/O 使用率过高情况	查看哪个进程占用 CPU/内存高或 I/O 使用率高，若是无用的进程，则 kill 掉，否则排查具体原因。例如 SQL 执行占用内存大，查看是否 SQL 语句需要优化。

表 3-5SSD 性能状态

异常状态	处理方法
SSD 读写性能故障	使用以下命令查看 SSD 是否有故障，排查具体故障原因。 gs_checkperf -i SSD -U gbase

3.5 检查和清理日志

日志是检查系统运行及故障定位的关键手段。建议按月度例行查看操作系统日志及数据库的运行日志。同时，随着时间的推移，日志的增加会占用较多的磁盘空间。建议按月度清理数据库的运行日志。

3.5.1 检查操作系统日志

建议按月检查操作系统日志，排除操作系统运行异常隐患。

执行如下命令查看操作系统日志文件。

```
vim /var/log/messages
```

关注其中近一个月出现的 kernel、error、fatal 等字样，根据系统报警信息进行处理。

3.5.2 检查数据库运行日志

数据库运行时，某些操作在执行过程中可能会出现错误，数据库依然能够运行。但是此时数据库中的数据可能已经发生不一致的情况。建议按月检查 GBase 8c 运行日志，及时发现隐患。

前提条件

收集日志的主机网络通畅且未宕机，数据库安装用户互信正常。

日志收集工具依赖操作系统工具如 gstack，如果未安装该工具，则提示错误后，跳过该收集项。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 执行如下命令收集数据库日志。

```
$ gs_collector --begin-time="20240301 01:01" --end-time="20240301 23:59"
```

20240301 01:01 为日志的开始时间，20240301 23:59 为日志的结束时间。

(3) 根据界面输出提示，进入相应的日志收集目录，解压收集的日志，并检查数据库日志。

以日志收集路径“/var/log/gbase/gbase/collector_20240301_154748.tar.gz”为例进行操作。

```
$ tar -xvzf /var/log/gbase/gbase/collector_20240301_154748.tar.gz
$ cd /var/log/gbase/gbase/collector_20240301_154748
```

示例

- 以 -begin-time 与 -end-time 为参数执行 gs_collector 命令。

```
$ gs_collector --begin-time="20240301 01:01" --end-time="20240301 23:59"
```

当显示类似如下信息表示日志已经归档。

```
Successfully collected files.
All results are stored in
/var/log/gbase/gbase/collector_20240301_154748.tar.gz.
```

- 以 -begin-time、 -end-time 与-h 为参数执行 gs_collector 命令。

```
$ gs_collector --begin-time="20240301 01:01" --end-time="20240301 23:59" -h
plat2
```

当显示类似如下信息表示日志已经归档。

```
Successfully collected  files
All results are stored in
/var/log/gbase/gbase/collector_20240301_154748.tar.gz.
```

- 以 -begin-time、 -end-time 与-f 为参数执行 gs_collector 命令。

```
$ gs_collector --begin-time="20240301 01:01" --end-time="20240301 23:59" -f
/opt/software/database/output
```

当显示类似如下信息表示日志已经归档。

```
Successfully collected  files
All results are stored in
/opt/software/database/output/collector_20240301_190511.tar.gz.
```

- 以 -begin-time、 -end-time 与 -keyword 为参数执行 gs_collector 命令。

```
$ gs_collector --begin-time="20240301 01:01" --end-time="20240301 23:59"
--keyword="os"
```

当显示类似如下信息表示日志已经归档。

```
Successfully collected files.
All results are stored in
/var/log/gbase/gbase/collector_20240301_190836.tar.gz.
```

- 以 -begin-time、 -end-time 与-o 为参数执行 gs_collector 命令。

```
$ gs_collector --begin-time="20240301 01:01" --end-time="20240301 23:59" -o
/opt/software/database/output
```

当显示类似如下信息表示日志已经归档。

```
Successfully collected files.
All results are stored in
/opt/software/database/output/collector_20240301_113711.tar.gz.
```

- 以 -begin-time、 -end-time 与-l 为参数（文件名必须以.log 为后缀）执行 gs_collector 命令。

```
$ gs_collector --begin-time="20240301 01:01" --end-time="20240301 23:59" -l
/opt/software/database/logfile.log
```

当显示类似如下信息表示日志已经归档。

```
Successfully collected files.
All results are stored in
/opt/software/database/output/collector_20240301_113711.tar.gz.
```

3.5.3 清理运行日志

数据库运行过程中会产生大量运行日志，占用大量的磁盘空间，建议清理过期日志文件，只保留一个月的日志。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 清理日志。

(3) 将超过 1 个月的日志备份到其他磁盘。

(4) 进入日志存放目录。

```
cd $GAUSSLOG
```

(5) 进入相应的子目录，使用如下方式删除 1 个月之前产生的日志。

```
rm 日志名称
```

日志文件的命名格式为“postgresql-年-月-日_HHMMSS”。

3.6 检查时间一致性

数据库事务一致性通过逻辑时钟保证，与操作系统时间无关，但是系统时间不一致会导致诸多潜在问题，主要是后台运维和监控功能异常，因此在月度检查时建议检查各个节点的时间一致性。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 创建记录 GBase 8c 各节点的配置文件（_mpphosts 文件目录_用户可随意指定，建议放在/tmp 下）。

```
vim /tmp/mpphosts
```

增加各节点的主机名称。

```
plat1
plat2
plat3
```

保存配置文件。

:wq!

(3) 执行如下命令，输出各节点上的时间到“/tmp/sys_ctl-os1.log”文件中。

```
for ihost in `cat /tmp/mpphosts`; do ssh -n -q $ihost "hostname;date"; done >
/tmp/sys_ctl-os1.log
```

(4) 根据输出确认各个节点的时间一致性，节点之间时间差异不能超过 30 秒。

```
cat /tmp/sys_ctl-os1.log
plat1
Thu Feb  9 16:46:38 CST 2017
plat2
Thu Feb  9 16:46:49 CST 2017
plat3
Thu Feb  9 16:46:14 CST 2017
```

3.7 检查应用连接数

如果应用程序与数据库的连接数超过最大值，则新的连接无法建立。建议每天检查连接数，及时释放空闲的连接或者增加最大连接数。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

postgres 为需要连接的数据库名称，15400 为数据库主节点的端口号。

连接成功后，系统显示类似如下信息：

```
gsql ((GBase 8c 1.0 build 290d125f) compiled at 2020-05-08 02:59:43 commit 2143
last mr 131
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

postgres=#
```

(3) 执行如下 SQL 语句查看连接数。

```
postgres=# SELECT count(*) FROM (SELECT pg_stat_get_backend_idset() AS backendid)
AS s;
```

显示类似如下的信息，其中 2 表示当前有两个应用连接到数据库。

```
count
-------
```

```
    2
(1 row)
```

(4) 查看现有最大连接数。

```
postgres=# SHOW max_connections;
```

显示信息如下，其中 200 为现在的最大连接数。

```
 max_connections
-----------------
 200
(1 row)
```

异常处理

如果显示的连接数接近数据库的最大连接数 max_connections，则需要考虑清理现有连接数或者增加新的连接数。

执行如下 SQL 语句，查看 state 字段等于 idle，且 state_change 字段长时间没有更新过的连接信息。

```
postgres=# SELECT * FROM pg_stat_activity where state='idle' order by
state_change;
```

显示类似如下的信息：

```
 datid | datname  |       pid       |    sessionid     | usesysid | usename |
application_name    | client_
addr | client_hostname | client_port |         backend_start         |
xact_start |          query_start
     |         state_change          | waiting | enqueue | state | resource_pool
| query_id |
        query                        | connection_info | unique_sql_id |
trace_id
-------+----------+-----------------+-----------------+----------+---------+-
----------------------+--------
------+----------------+------------+------------------------------+-------
------+-------------------------
------+------------------------------+---------+---------+-------+-----------
----+----------+---------------
-----------------------------------+---------------+--------------+---
-------
 16898 | postgres | 140710206154496 | 140710206154496 |       10 | gbase   |
statement flush thread |
```

```
     |                 |              | 2023-03-22 15:04:37.313691+08 |
|
     | 2023-03-22 15:04:37.313727+08 | f       |          | idle   | default_pool
|        0 |
                                       |                 |             0 |
 16898 | postgres | 140710136903424 | 140710136903424 |        10 | gbase    | dn1_1
|
     |                 |           -1 | 2023-03-22 15:04:39.211933+08 |
| 2023-04-15 16:25:00.12081
1+08 | 2023-04-15 16:25:00.121314+08 | f       |          | idle   | default_pool
|        0 | select term, ls
n from pg_last_xlog_replay_location(); |                 |             0 |
(2 rows)
```

- 释放空闲的连接数。

查看每个连接，并与此连接的使用者确认是否可以断开连接，或执行如下 SQL 语句释放连接。其中，pid 为上一步查询中空闲连接所对应的 pid 字段值。

```
postgres=# SELECT pg_terminate_backend(140390132872976);
```

显示类似如下的信息：

```
postgres=# SELECT pg_terminate_backend(140390132872976);
 pg_terminate_backend
----------------------
 t
(1 row)
```

如果没有可释放的连接，请执行下一步。

- 增加最大连接数。

```
gs_guc set -D /gaussdb/data/dbnode -c "max_connections= 800"
```

其中 800 为新修改的连接数。

重启数据库服务使新的设置生效。

说明

- 重启 GBase 8c 操作会导致用户执行操作中断，请在操作之前规划好合适的执行窗口。

```
gs_om -t stop && gs_om -t start
```

3.8 例行维护表

为了保证数据库的有效运行，数据库必须在插入/删除操作后，基于客户场景，定期做 VACUUM FULL 和 ANALYZE，更新统计信息，以便获得更优的性能。

3.8.1 相关概念

使用 VACUUM、VACUUM FULL 和 ANALYZE 命令定期对每个表进行维护，主要有以下原因：

- VACUUM FULL 可回收已更新或已删除的数据所占据的磁盘空间，同时将小数据文件合并。
- VACUUM 对每个表维护了一个可视化映射来跟踪包含对别的活动事务可见的数组的页。一个普通的索引扫描首先通过可视化映射来获取对应的数组，来检查是否对当前事务可见。若无法获取，再通过堆数组抓取的方式来检查。因此更新表的可视化映射，可加速唯一索引扫描。
- VACUUM 可避免执行的事务数超过数据库阈值时，事务 ID 重叠造成的原有数据丢失。
- ANALYZE 可收集与数据库中表内容相关的统计信息。统计结果存储在系统表 PG_STATISTIC 中。查询优化器会使用这些统计数据，生成最有效的执行计划。

3.8.2 操作步骤

使用 VACUUM 或 VACUUM FULL 命令，进行磁盘空间回收。

- VACUUM：
  - 对表执行 VACUUM 操作

```
postgres=# VACUUM customer;
VACUUM
```

可以与数据库操作命令并行运行。（执行期间，可正常使用的语句：SELECT、INSERT、UPDATE 和 DELETE。不可正常使用的语句：ALTER TABLE）。

  - 对表分区执行 VACUUM 操作

```
postgres=# VACUUM customer_par PARTITION ( P1 );
VACUUM
```

- VACUUM FULL：

```
postgres=# VACUUM FULL customer;
VACUUM
```

需要向正在执行的表增加排他锁，且需要停止其他所有数据库操作。

使用 ANALYZE 语句更新统计信息。

```
postgres=# ANALYZE customer;
ANALYZE
```

使用 ANALYZE VERBOSE 语句更新统计信息，并输出表的相关信息。

```
postgres=# ANALYZE VERBOSE customer;
ANALYZE
```

也可以同时执行 VACUUM ANALYZE 命令进行查询优化。

```
postgres=# VACUUM ANALYZE customer;
VACUUM
```

说明

- VACUUM 和 ANALYZE 会导致 I/O 流量的大幅增加，这可能会影响其他活动会话的性能。因此，建议通过“vacuum_cost_delay”参数设置《数据库参考》中“GUC 参数说明 > 资源消耗 > 基于开销的清理延迟”。

删除表。

```
postgres=# DROP TABLE customer;
postgres=# DROP TABLE customer_par;
postgres=# DROP TABLE part;
```

当结果显示为如下信息，则表示删除成功。

```
DROP TABLE
```

3.8.3 维护建议

定期对部分大表做 VACUUM FULL，在性能下降后为全库做 VACUUM FULL，目前暂定每月做一次 VACUUM FULL。

定期对系统表做 VACUUM FULL，主要是 PG_ATTRIBUTE。

启用系统自动清理线程（AUTOVACUUM）自动执行 VACUUM 和 ANALYZE，回收被标识为删除状态的记录空间，并更新表的统计数据。

3.9 例行重建索引

3.9.1 背景信息

数据库经过多次删除操作后，索引页面上的索引键将被删除，造成索引膨胀。例行重建索引，可有效的提高查询效率。

数据库支持的索引类型为 B-tree 索引，例行重建索引可有效的提高查询效率。

如果数据发生大量删除后，索引页面上的索引键将被删除，导致索引页面数量的减少，造成索引膨胀。重建索引可回收浪费的空间。

新建的索引中逻辑结构相邻的页面，通常在物理结构中也是相邻的，所以一个新建的索引比更新了多次的索引访问速度要快。

3.9.2 重建索引

重建索引有以下两种方式：

- 先运行 DROP INDEX 语句删除索引，再运行 CREATE INDEX 语句创建索引。

在删除索引过程中，会在父表上增加一个短暂的排他锁，阻止相关读写操作。在创建索引过程中，会锁住写操作但是不会锁住读操作，此时读操作只能使用顺序扫描。

- 使用 REINDEX 语句重建索引。

使用 REINDEX TABLE 语句重建索引，会在重建过程中增加排他锁，阻止相关读写操作。

使用 REINDEX INTERNAL TABLE 语句重建 desc 表(包括列存表的 cudesc 表)的索引，会在重建过程中增加排他锁，阻止相关读写操作。

3.9.3 操作步骤

假定在导入表“areaS”上的“area_id”字段上存在普通索引“areaS_idx”。重建索引有以下两种方式：

(1) 先删除索引（DROP INDEX），再创建索引（CREATE INDEX）。

① 删除索引。

```
postgres=# DROP INDEX areaS_idx;
```

当结果显示如下信息，则表示删除成功。

```
DROP INDEX
```

② 创建索引。

```
postgres=# CREATE INDEX areaS_idx ON areaS (area_id);
```

当结果显示如下信息，则表示创建成功。

```
CREATE INDEX
```

(2) 使用 REINDEX 重建索引。

- 使用 REINDEX TABLE 语句重建索引。

```
postgres=# REINDEX TABLE areaS;
```

当结果显示如下信息，则表示重建成功。

```
REINDEX
```

- 使用 REINDEX INTERNAL TABLE 重建 desc 表（包括列存表的 cudesc 表）的索引。

```
postgres=# REINDEX INTERNAL TABLE areaS;
```

当结果显示如下信息，则表示重建成功。

```
REINDEX
```

说明

- 在重建索引前，用户可以通过临时增大 maintenance_work_mem 和 psort_work_mem 的取值来加快索引的重建。

3.10 导出并查看 wdr 诊断报告

生成快照数据需将 enable_wdr_snapshot 参数设置为 on。需要拥有 monadmin 权限的用户访问并生成 WDR 诊断报告。

(1) 新建报告文件。例如，在/opt/database/wdr 路径下创建报告名为 wdrTestNode，请根据实际情况修改：

```
$ touch /opt/database/wdr/wdrTestNode.html
```

(2) 检查 enable_wdr_snapshot 参数是否已开启：

```
$ gs_guc check -N all -I all -c 'enable_wdr_snapshot'
```

若不为 on 则需要修改参数值，并重启生效：

```
$ gs_guc reload -N all -I all -c 'enable_wdr_snapshot=on'
$ gs_om -t restart
```

(3) 创建具有 monadmin 权限的用户，并以此用户身份连接默认库 postgres。例如：

```
$ gsql -d postgres -p 15400
postgres=# CREATE USER test_wdr monadmin password 'Gbase,123';
postgres=# \q
$ gsql -d postgres -p 15400 -U test_wdr -W 'Gbase,123'
```

(4) 选择 snapshot.snapshot 表中两个不同的 snapshot，当这两个 snapshot 之间未发生服务重启，便可以使用这两个 snapshot 生成报告。

```
postgres=# select * from snapshot.snapshot order by start_ts desc limit 10;
```

(5) 在本地生成 HTML 格式的 WDR 报告。

- 执行如下命令，设置报告格式。例如：

```
postgres=# \a \t \o /opt/database/wdr/wdrTestNode.html
```

其中\a 表示不显示表行列符号，\t 表示不显示列名，\o 表示指定输出文件，/opt/database/wdr/wdrTestNode.html 为步骤(1)新建的报告文件。

- 执行如下命令，生成 HTML 格式的 WDR 报告。

```
select generate_wdr_report(begin_snap_id oid, end_snap_id oid, int report_type,
int report_scope, int node_name );
```

例如，生成集群级别的报告：

```
postgres=# select generate_wdr_report(1, 2, 'all', 'cluster',null);
```

例如，生成某个节点的报告：

```
postgres=# select generate_wdr_report(1, 2, 'all', 'node',
pgxc_node_str()::cstring);
```

说明

➢ 当前 GBase 8c 的节点名固定是“dn_6001”，也可直接代入。

表 3-6 参数说明

参数	说明	取值范围
begin_snap_id	要查看的某段时间性能的开始的 snapshot 的 id(表 snapshot.snaoshot 中的 snapshot_id)	-
end_snap_id	结束 snapshot 的 id，默认 end_snap_id 大于 begin_snap_id （表 snapshot.snaoshot 中的	-

	snapshot_id)	
report_type	指定生成 report 的类型。	● summary ● detail ● all，即同时包含 summary 和 detail。
report_scope	指定生成 report 的范围。	● cluster：集群 ● node：集群中某个节点。
node_name	在 report_scope 指定为 single node 时，需要把该参数指定为对应节点的名称。 在 report_scope 为 cluster 时，该值可以指定为省略或者为 NULL。	-

(6) 执行如下命令关闭输出选项及格式化输出命令。

```
postgres=# \o \a \t
```

(7) 进入报告目录下，查看 WDR 报告内容。

表 3-7 WDR 报告主要内容

report_type 类型	报告项	report_scope 类型	描述
Summary	Database Stat	cluster（集群范围）	数据库维度的性能统计信息：事务，读写，行活动，写冲突，死锁等。
	Load Profile	cluster（集群范围）	集群维度的性能统计信息：CPU 时间，DB 时间，逻辑读/物理读，IO 性能，登入登出，负载强度，负载性能表现等。
	Instance Efficiency Percentages	cluster（集群范围） node（节点范围）	集群级或者节点缓冲命中率。

report_type 类型	报告项	report_scope 类型	描述
	IO Profile	cluster（集群范围） node（节点范围）	集群或者节点维度的 IO 的使用情况。
Detail	Top 10 Events by Total Wait Time	node（节点范围）	最消耗时间的事件。
	Wait Classes by Total Wait Time	node（节点范围）	最消耗时间的等待时间分类。
	Host CPU	node（节点范围）	主机 CPU 消耗。
	Memory Statistics	node（节点范围）	内核内存使用分布。
	Time Model	node（节点范围）	节点范围的语句的时间分布信息。
	Wait Events	node（节点范围）	节点级别的等待事件的统计信息。
	Utility status	node（节点范围）	复制槽和后台 checkpoint 的状态信息。
	Configuration settings	node（节点范围）	节点配置。
	SQL Statistics	cluster（集群范围） node（节点范围	SQL 语句各个维度性能统计：端到端时间，行活动，缓存命中，CPU 消耗，时间消耗细分。
	SQL Detail	cluster（集群范	SQL 语句文本详情。

report_type 类型	报告项	report_scope 类型	描述
		围） node（节点范围	
	Object stats	cluster（集群范围） node（节点范围	表、索引维度的性能统计信息。
	Cache IO Stats	cluster（集群范围） node（节点范围	用户的表、索引的 IO 的统计信息。

3.11 数据安全维护建议

为保证 GBase 8c 数据库中的数据安全，避免丢失数据、非法访问数据等事故发生，请仔细阅读以下内容。

3.11.1 避免数据被丢失

建议用户规划周期性的物理备份，且对备份文件进行可靠的保存。在系统发生严重错误的情况下，可以利用备份文件，将系统恢复到备份前的状态。

3.11.2 避免数据被非法访问

建议对数据库用户进行权限分级管理。数据库管理员根据业务需要，建立用户并赋予权限，保证各用户对数据库的合理访问。

对于 GBase 8c 的服务端和客户端（或基于客户端库开发的应用程序），最好也部署在可信任的内网中。如果服务端和客户端一定要部署在非信任的网络中，需要在服务启动前，打开 SSL 加密，保证数据在非信任网络上的传输安全。需要注意的是，打开 SSL 加密会降低数据库的性能。

3.11.3 避免系统日志泄露个人数据

将调试日志发给他人进行分析前，请删除个人数据。

说明

- 因为日志级别（log_min_messages）设置为 DEBUGx（x 为 DEBUG 级别，取值范围为 1~5）时，调试日志中记录的信息可能包含用户的个人数据。

将系统日志发给其他人进行分析前，请删除个人数据。因为在默认配置下，当 SQL 语句执行错误时，日志中会记录出错的 SQL 语句，而这些 SQL 语句中可能包含用户个人数据。

将 log_min_error_statement 参数的值设置为 PANIC，可以避免将出错的 SQL 语句记录在系统日志中。若禁用该功能，当出现故障时，很难定位故障原因。

3.12 慢 sql 诊断

背景信息

在 SQL 语句执行性能不符合预期时，可以查看 SQL 语句执行信息，便于事后分析 SQL 语句执行时的行为，从而诊断 SQL 语句执行出现的相关问题。

前提条件

- 数据库实例运行正常。
- 查询 SQL 语句信息，需要合理设置 GUC 参数 track_stmt_stat_level。track_stmt_stat_level 参数控制语句执行跟踪的级别，第一部分控制全量 SQL，第二部分控制慢 SQL。对于慢 SQL，当 track_stmt_stat_level 的值为非 OFF 时，且 SQL 执行时间超过 log_min_duration_statement，会记录为慢 SQL。默认值为“OFF,L0”，建议设置为“L0,L0”。

```
gs_guc reload -N all -I all -c "track_stmt_stat_level='L0,L0'"
```

- 只能用系统管理员和监控管理员权限进行操作。

操作

- 查看数据库实例中 SQL 语句执行信息

```
select * from dbe_perf.get_global_full_sql_by_timestamp(start_timestamp,
end_timestamp);
```

例如：

```
postgres=# select * from DBE_PERF.get_global_full_sql_by_timestamp('2024-3-01
09:25:22', '2024-3-1 17:54:41');
```

返回如下信息：

```
 node_name | db_name  |  schema_name   | origin_node | user_name | application_n
ame | client_addr | client_port | unique_query_id | debug_query_id  |
                                                                          query
```

```
|          start_time           |          finish_time          |
slow_sql_thres
hold | transaction_id |    thread_id     |    session_id     | n_soft_parse | n_har
d_parse | query_plan | n_returned_rows | n_tuples_fetched | n_tuples_returned |
n_tuples_inserted | n_tuples_updated | n_tuples_deleted | n_blocks_fetched | n_b
locks_hit | db_time | cpu_time | execution_time | parse_time | plan_time | rewri
te_time | pl_execution_time | pl_compilation_time | data_io_time |             n
et_send_info               |               net_recv_info               |
n
et_stream_send_info        |       net_stream_recv_info        | lock_count |
lo
ck_time | lock_wait_count | lock_wait_time | lock_max_count | lwlock_count | lwl
ock_wait_count | lwlock_time | lwlock_wait_time |
                                 details
                          | is_slow_sql | trace_id | advise
----------+---------+---------------+------------+----------+----------
---
----+------------+------------+---------------+---------------+--------
---
--------------------------------------------------------------------------
---
--------------------------------------------------------------------------
---
+---------------------------+---------------------------+-----------
---
-----+---------------+---------------+---------------+-------------+---
---
--------+-----------+---------------+----------------+-----------------
-+-
-----------------+-----------------+----------------+-----------------+-
---
---------+---------+---------+---------------+-----------+----------+---
---
--------+------------------+--------------------+-------------+----------
---
-----------------------+-----------------------------------------+-----
---
-----------------------+---------------------------------+-----------+
---
```

```
-------+---------------+--------------+---------------+------------+-
---
--------------+------------+-----------------+--------------------------
---
-------------------------------------------------------------------------
---
----------------------+------------+---------+--------
 dn_6001    | postgres | "$user",public |           0 | gbase      | gs_clean
    | ::1          |       34048 |      2555941858 | 562949953425248 | SET connec
tion_info = '{"driver_name":"libpq","driver_version":"(single_node GBase8cV5
S5.
0.0B19 build b0e57b26) compiled at 2024-01-14 16:38:29 commit 0 last mr 443 "}'
| 2024-03-01 15:17:29.603725+08 | 2024-03-01 15:17:29.60394+08  |
300
0000 |             0 | 140705231529728 | 140705231529728 |            0 |
      0 |             |               0 |                 0 |                0 |
              0 |               0 |               0 |                1 |
        1 |     232 |      228 |             0 |         18 |         0 |
      1 |                 0 |                   0 |            0 | {"time":26,
"
n_calls":1, "size":211}    | {"time":107, "n_calls":2, "size":204}       |
{"time"
:0, "n_calls":0, "size":0} | {"time":0, "n_calls":0, "size":0} |          2 |
      0 |               0 |              0 |              2 |            0 |
            0 |           0 |               0 |
\x420000000241380000002000000
011300427566486173685461626c655365617263680001000000000000000040b00666c7573682
064
617461002700000000000000 | f           |           |
......
```

- 执行命令查看数据库实例中慢 SQL 语句执行信息

```
select * from dbe_perf.get_global_slow_sql_by_timestamp(start_timestamp,
end_timestamp);
```

例如：

```
postgres=# select * from DBE_PERF.get_global_slow_sql_by_timestamp('2024-3-01
09:25:22', '2024-3-1 17:54:41');
```

返回如下信息：

```
 node_name | db_name | schema_name | origin_node | user_name | application_name
```

```
| client_addr | client_port | unique_query_id | debug_query_id | query | start_t
ime | finish_time | slow_sql_threshold | transaction_id | thread_id | session_id
 | n_soft_parse | n_hard_parse | query_plan | n_returned_rows | n_tuples_fetched
 | n_tuples_returned | n_tuples_inserted | n_tuples_updated | n_tuples_deleted
|
 n_blocks_fetched | n_blocks_hit | db_time | cpu_time | execution_time | parse_t
ime | plan_time | rewrite_time | pl_execution_time | pl_compilation_time | data_
io_time | net_send_info | net_recv_info | net_stream_send_info | net_stream_recv
_info | lock_count | lock_time | lock_wait_count | lock_wait_time | lock_max_cou
nt | lwlock_count | lwlock_wait_count | lwlock_time | lwlock_wait_time | details
 | is_slow_sql | trace_id | advise
----------+---------+------------+------------+----------+--------------
---
+------------+------------+----------------+---------------+-------+-----
---
----+------------+------------------+---------------+----------+--------
---
-+-------------+-------------+-----------+----------------+--------------
---
-+------------------+------------------+-----------------+----------------
--+
-----------------+-------------+---------+---------+---------------+-----
---
----+----------+-------------+------------------+--------------------+---
---
--------+--------------+--------------+---------------------+-------------
---
------+-----------+----------+----------------+--------------+----------
---
---+-------------+-----------------+------------+-----------------+-----
---
-+------------+---------+--------
(0 rows)
```

- 查看当前主节点 SQL 语句执行信息

```
select * from statement_history;
```

例如：

```
postgres=# select * from statement_history;
```

返回如下信息：

```
 db_name  |  schema_name   | origin_node | user_name | application_name |
client_addr | client_port | unique_query_i
d | debug_query_id  |                                query
|          start_time
    |          finish_time          | slow_sql_threshold | transaction_id |
thread_id    |   session_id    | n_so
ft_parse | n_hard_parse | query_plan | n_returned_rows | n_tuples_fetched |
n_tuples_returned | n_tuples_inserted |
n_tuples_updated | n_tuples_deleted | n_blocks_fetched | n_blocks_hit | db_time
| cpu_time | execution_time | parse_
time | plan_time | rewrite_time | pl_execution_time | pl_compilation_time |
data_io_time |             net_send_info
             |               net_recv_info               |
net_stream_send_info        |       net_stream_recv
_info        | lock_count | lock_time | lock_wait_count | lock_wait_time |
lock_max_count | lwlock_count | lwlock_wa
it_count | lwlock_time | lwlock_wait_time |
details
                                                                  | is_slow_sql |
trace_id | advise
---------+---------------+-------------+-----------+------------------+----
--------+-------------+--------------
--+-----------------+-------------------------------------------------------------
----------+--------------------------
----+------------------------------+-------------------+---------------+--
--------------+-----------------+-----
--------+-------------+-----------+----------------+-----------------+--
----------------+------------------+-
----------------+-----------------+----------------+-------------+------
---+----------+---------------+-------
-----+----------+-------------+------------------+--------------------+--
-----------+--------------------------
------------+---------------------------------------------+------------------
---------------+----------------------
------------+-----------+----------+----------------+---------------+---
------------+-------------+----------
--------+------------+-----------------+-------------------------------
---------------------------------------
-------------------------------------------------------------+------------
+----------+--------
```

```
 postgres | "$user",public |           0 | gbase      | gsql                |
|          -1 |      138560285
7 | 562949953425306 | select * from statement_history order by start_time desc
limit ?; | 2024-03-01 15:21:49.978874
+08 | 2024-03-01 15:21:49.979871+08 |            3000000 |               0 |
140704817960704 | 140704817960704 |
       0 |            1 |            |               1 |                2 |
1 |                 0 |
               0|                 0|                4|             4|     1008
|     1004 |              0 |
  49 |       322 |            6 |                 0 |                   0 |
0 | {"time":55, "n_calls":3,
"size":3242} | {"time":25545767, "n_calls":1, "size":71} | {"time":0,
"n_calls":0, "size":0} | {"time":0, "n_calls":
0, "size":0} |          14 |         0 |               0 |              0 |
4 |            0 |
       0 |           0 |                 0 |
\x420000000241380000000200000001130042756648617368546162 6c65536561726368
000400000000000000040b00666c7573682064617461000c00000000000000 | f           |
|
......
```

- 查看当前备节点 SQL 语句执行信息

```
select * from dbe_perf.standby_statement_history(is_only_slow, start_timestamp,
end_timestamp);
```

例如：

```
select * from dbe_perf.standby_statement_history(true, '2024-03-01 09:25:22',
'2024-03-01 23:54:41');
```

4 备份与恢复

4.1 概述

数据备份是保护数据安全的重要手段之一，为了更好的保护数据安全，GBase 8c 数据库支持三种备份恢复类型，以及多种备份恢复方案，备份和恢复过程中提供数据的可靠性保障机制。

备份与恢复类型可分为逻辑备份与恢复、物理备份与恢复、闪回恢复。

- 逻辑备份与恢复：通过逻辑导出对数据进行备份，逻辑备份只能基于备份时刻进行数据转储，所以恢复时也只能恢复到备份时保存的数据。对于故障点和备份点之间的数据，逻辑备份无能为力，逻辑备份适合备份那些很少变化的数据，当这些数据因误操作被损坏时，可以通过逻辑备份进行快速恢复。如果通过逻辑备份进行全库恢复，通常需要重建数据库，导入备份数据来完成，对于可用性要求很高的数据库，这种恢复时间太长，通常不被采用。由于逻辑备份具有平台无关性，所以更为常见的是，逻辑备份被作为一个数据迁移及移动的主要手段。
- 物理备份与恢复：通过物理文件拷贝的方式对数据库进行备份，以磁盘块为基本单位将数据从主机复制到备机。通过备份的数据文件及归档日志等文件，数据库可以进行完全恢复。物理备份速度快，一般被用作对数据进行备份和恢复，用于全量备份的场景。通过合理规划，可以低成本进行备份与恢复。
- 闪回恢复：利用回收站的闪回恢复删除的表。数据库的回收站功能类似于 windows 系统的回收站，将删除的表信息保存到回收站中。利用 MVCC 机制闪回恢复到指定时间点或者 CSN 点。

以下为 GBase 8c 支持的三类数据备份恢复方案，备份方案也决定了当异常发生时该如何恢复。

表 4-1 三种备份恢复类型对比

备份类型	应用场景	支持介质	工具名称	恢复时间	优缺点

备份类型	应用场景	支持介质	工具名称	恢复时间	优缺点
逻辑备份与恢复	适合于数据量小的场景。可以备份单表和多表，单database和所有database。 备份后的数据需要使用gsql或者gs_restore工具恢复。数据量大时，恢复需要较长时间。	磁盘SSD	gs_cump	纯文本格式数据恢复时间长。归档格式数据恢复时间中等。	导出数据库相关信息的工具，用户可以自定义导出一个数据库或其中的对象（模式、表、视图等）。 支持导出的数据库可以是默认数据库postgres，也可以是自定义数据库。 导出的格式可选择纯文本格式或者归档格式。纯文本格式的数据只能通过gsql进行恢复，恢复时间较长。归档格式的数据只能通过gs_restore进行恢复，恢复时间较纯文本格式短。
			gs_dumpall	数据恢复时间长。	导出所有数据库相关信息工具，它可以导出GBase 8c数据库的所有数据，包括默认数据库postgres的数据、自定义数据库的数据、以及GBase 8c所有数据库公共的全局对象。 只能导出纯文本格式的数据，导出的数据只能通过gsql进行恢复，恢复时间较长。

备份类型	应用场景	支持介质	工具名称	恢复时间	优缺点
物理备份与恢复	适用于数据量大的场景，主要用于全量数据备份恢复，也可对整个数据库中的WAL归档日志和运行日志进行备份。		gs_backup	数据量小数据恢复时间快。	导出数据库相关信息的 OM 工具，可以导出数据库参数文件和二进制文件。支持备份、恢复重要数据、显示帮助信息和版本号信息。 在进行备份时，可以选择备份内容的类型，在进行还原时，需要保证各节点备份目录中存在备份文件。在集群恢复时，通过静态配置文件中的集群信息进行恢复。只恢复参数文件恢复时间较短。
			gs_basebackup	恢复时可以直接拷贝替换原有的文件，或者直接在备份的库上启动数据库，恢复时间快。	对服务器数据库文件的二进制进行全量拷贝，只能对数据库某一个时间点的时间作备份。结合 PITR 恢复，可恢复全量备份时间点后的某一时间点。

备份类型	应用场景	支持介质	工具名称	恢复时间	优缺点
			gs_probackup	恢复时可以直接恢复到某个备份点，在备份的库上启动数据库，恢复时间快。	gs_probackup 用于管理 GBase 8c 数据库备份和恢复，可对实例进行定期备份。可用于备份单机数据库或者集群主节点数据库，为物理备份。 可备份外部目录的内容，如脚本文件、配置文件、日志文件、dump 文件等。 支持增量备份、定期备份和远程备份。增量备份时间相对于全量备份时间比较短，只需要备份修改的文件。当前默认备份是数据目录，如果表空间不在数据目录，需要手动指定备份的表空间目录进行备份。当前只支持在主机上执行备份。

备份类型	应用场景	支持介质	工具名称	恢复时间	优缺点
闪回恢复	适用于误删除表的场景。需要将表中的数据恢复到指定时间点或者CSN。		无	可以将表的状态恢复到指定时间点或者是表结构删除前的状态，恢复时间快。	闪回技术能够有选择性的高效撤销一个已提交事务的影响，从人为错误中恢复。在采用闪回技术之前，只能通过备份恢复、PITR 等手段找回已提交的数据库修改，恢复时长需要数分钟甚至数小时。采用闪回技术后，恢复已提交的数据库修改前的数据，只需要秒级，而且恢复时间和数据库大小无关。 闪回支持两种恢复模式： ● 基于 MVCC 多版本的数据恢复：适用于误删除、误更新、误插入数据的查询和恢复，用户通过配置旧版本保留时间，并执行相应的查询或恢复命令，查询或恢复到指定的时间点或 CSN 点。 ● 基于类似 windows 系统回收站的恢复：适用于误 DROP、误 TRUNCATE 的表的恢复。用户通过配置回收站开关，并执行相应的恢复命令，可以将误 DROP、误 TRUNCATE 的表找回。

当需要进行备份恢复操作时，主要从以下方面考虑数据备份方案。

- 备份对业务的影响在可接受范围。
- 数据库恢复效率。为尽量减小数据库故障的影响，要使恢复时间减到最少，从而使恢复

的效率达到最高。

- 数据可恢复程度。当数据库失效后，要尽量减少数据损失。
- 数据库恢复成本。在现网选择备份策略时参考的因素比较多，如备份对象、数据大小、网络配置等，下表列出了可用的备份策略和每个备份策略的适用场景。

表 4-2 备份策略典型场景

备份策略	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	备份： ● 每个主机 80 Mbit/s（NBU/EISOO+磁盘） ● 约 90%磁盘 I/O 速率（SSD/HDD）
表备份	● 表所在模式 ● 网络配置(NBU)	数据：10 TB 级	备份：基于查询性能速度+I/O 速度 说明 多表备份时，备份耗时计算方式： 总时间 = 表数量*起步时间+数据总量/数据备份速度 其中 ● 磁盘起步时间为 5s 左右，NBU 起步时间比 DISK 长（取决于 NBU 部署方案）。 ● 数据备份速度为单节点 50MB/s 左右（基于 1GB 大小的表，物理机备份到本地磁盘得出此速率）。 表越小，备份性能更低。

4.2 配置文件的备份与恢复

背景信息

在 GBase 8c 使用过程中，如果静态配置文件无意损坏后，会影响 GBase 8c 感知集群拓扑结构和主备关系。使用 gs_om 工具生成的静态配置文件，可以替换已经损坏的配置文件，

保证集群的正常运行。

前置条件

无。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 执行如下命令会在本服务器指定目录下生成配置文件，例如：

```
gs_om -t generateconf -X /home/gbase/gbase_pkg/cluster_config.xml --distribute
```

其中，/home/gbase/gbase_pkg/cluster_config.xml 是指 GBase 8c 安装时的 XML 配置文件，请根据实际情况修改。

返回的日志信息中会有新文件的存放目录。打开新文件存放目录，会出现以主机名命名的配置文件。原配置文件出现损坏时，请用该文件分别替换对应主机的配置文件。

若不使用--distribute 参数，需执行步骤 3 将静态配置文件分配到对应节点；若使用--distribute 参数，则会将生成的静态配置文件自动分配到对应节点，无需执行步骤 3。

(3) （可选）分别替换各主机安装目录下损坏的静态配置文件，以/home/gbase/gbase_pkg/为例。

```
mv
/opt/database/install/om/script/static_config_files/cluster_static_config_gba
se8c_5_124 /home/gbase/gbase_pkg/cluster_config.xml
```

示例

在 GBase 8c 中的任意主机上执行如下命令，生成配置文件：

```
$ gs_om -t generateconf -X /home/gbase/gbase_pkg/clusterconfig.xml --distribute
```

返回如下信息：

```
Generating static configuration files for all nodes.
Creating temp directory to store static configuration files.
Successfully created the temp directory.
Generating static configuration files.
Successfully generated static configuration files.
Static configuration files for all nodes are saved in
/opt/database/install/om/script/static_config_files.
Distributing static configuration files to all nodes.
Successfully distributed static configuration files.
```

打开生成的配置文件目录，会看到新生成的文件：

```
$ cd /opt/database/install/om/script/static_config_files
$ ll
```

返回如下信息：

```
total 456
-rwxr-xr-x 1 gbase gbase 155648 2024-02-19 15:51 cluster_static_config_plat1
-rwxr-xr-x 1 gbase gbase 155648 2024-02-19 15:51 cluster_static_config_plat2
-rwxr-xr-x 1 gbase gbase 155648 2024-02-19 15:51 cluster_static_config_plat3
```

4.3 逻辑备份与恢复

通过 GBase 8c 提供的逻辑备份工具 gs_dump、gs_dumpall，可以实现逻辑文件备份，支持纯文本、自定义归档格式、目录归档格式、tar 归档格式四种文件格式。使用说明详见《GBase 8c V5_5.0.0_工具与命令参考手册》中”gs_dump”、”gs_dumpall”章节。

gs_restore 是 GBase 8c 提供的针对 gs_dump 导出数据的导入工具。通过此工具可将由 gs_dump 生成的导出文件进行导入。使用说明详见《GBase 8c V5_5.0.0_工具与命令参考手册》中”gs_restore”章节。

示例

(1) 使用 gs_dump 转储数据库为 SQL 文本文件或其它格式的操作，如下所示。导出操作时，请确保指定目录存在，并且当前的操作系统用户对其具有读写权限。执行 gs_dump，导出 postgres 数据库全量信息，导出的 MPPDB_backup.sql 文件格式为纯文本格式。

```
$ gs_dump postgres -U gbase -W gbase,123 -f
/home/gbase/data/backup/MPPDB_backup.sql -p 15400 -F p
gs_dump[port='15400'][postgres][2024-02-29 16:40:16]: The total objects number
is 442.
gs_dump[port='15400'][postgres][2024-02-29 16:40:16]: [100.00%] 442 objects
have been dumped.
gs_dump: [port='15400'] [postgres] [archiver] [2024-02-29 16:40:16] WARNING:
archive items not in correct section order
gs_dump[port='15400'][postgres][2024-02-29 16:40:16]: dump database postgres
successfully
gs_dump[port='15400'][postgres][2024-02-29 16:40:16]: total time: 1274  ms
```

(2) 执行 gs_dump，导出 postgres 数据库全量信息，导出的 MPPDB_backup.tar 文件格式为 tar 格式。

```
$ gs_dump postgres -U gbase -W gbase,123 -f
/home/gbase/data/backup/MPPDB_backup.tar -p 15400 -F t
gs_dump[port='15400'][postgres][2024-02-29 10:02:24]: The total objects number
is 1369. gs_dump[port='15400'][postgres][2024-02-29 10:02:53]: [100.00%] 1369
objects have been dumped. gs_dump[port='15400'][postgres][2024-02-29 10:02:53]:
dump database postgres successfully gs_dump[port='15400'][postgres][2024-02-29
10:02:53]: total time: 50086 ms
```

(3) 执行 gs_dump，导出 postgres 数据库全量信息，导出的 MPPDB_backup.dmp 文件格式为自定义归档格式。

```
$ gs_dump postgres -U gbase -W gbase,123 -f
/home/gbase/data/backup/MPPDB_backup.dmp -p 15400 -F c
gs_dump[port='15400'][postgres][2024-02-29 10:05:40]: The total objects number
is 1369. gs_dump[port='15400'][postgres][2024-02-29 10:06:03]: [100.00%] 1369
objects have been dumped. gs_dump[port='15400'][postgres][2024-02-29 10:06:03]:
dump database postgres successfully gs_dump[port='15400'][postgres][2024-02-29
10:06:03]: total time: 36620 ms
```

(4) 执行 gs_dump，导出 postgres 数据库全量信息，导出的 MPPDB_backup 文件格式为目录格式。

```
$ gs_dump postgres -U gbase -W gbase,123 -f /home/gbase/data/backup/MPPDB_backup
-p 15400 -F d
gs_dump[port='15400'][postgres][2024-02-29 10:16:04]: The total objects number
is 1369. gs_dump[port='15400'][postgres][2024-02-29 10:16:23]: [100.00%] 1369
objects have been dumped. gs_dump[port='15400'][postgres][2024-02-29 10:16:23]:
dump database postgres successfully gs_dump[port='15400'][postgres][2024-02-29
10:16:23]: total time: 33977 ms
```

(5) 执行 gs_dump，导出 postgres 数据库信息，但不导出/home/MPPDB_temp.sql 中指定的表信息。导出的 MPPDB_backup.sql 文件格式为纯文本格式。

```
$ gs_dump postgres -U gbase -W gbase,123
--exclude-table-file=/home/gbase/data/MPPDB_temp.sql -f
/home/gbase/data/backup/MPPDB_backup.sql -p 15400
gs_dump[port='15400'][postgres][2024-02-29 10:37:01]: The total objects number
is 1367. gs_dump[port='15400'][postgres][2024-02-29 10:37:22]: [100.00%] 1367
objects have been dumped. gs_dump[port='15400'][postgres][2024-02-29 10:37:22]:
dump database postgres successfully gs_dump[port='15400'][postgres][2024-02-29
10:37:22]: total time: 37017 ms
```

(6) 执行 gs_dump，仅导出依赖于指定表 testtable 的视图信息。然后创建新的 testtable 表，

再恢复依赖其上的视图。

- 备份仅依赖于 testtable 的视图。

```
gs_dump -s -p 15400 postgres -t PUBLIC.testtable --include-depend-objs
--exclude-self -f backup/ MPPDB_backup.sql -F p
gs_dump[port='15400'][postgres][2018-06-15 14:12:54]: The total objects number
is 331. gs_dump[port='15400'][postgres][2018-06-15 14:12:54]: [100.00%] 331
objects have been dumped. gs_dump[port='15400'][postgres][2018-06-15 14:12:54]:
dump database postgres successfully gs_dump[port='15400'][postgres][2018-06-15
14:12:54]: total time: 327 ms
```

- 修改 testtable 名称。

```
gsql -p 15400 postgres -r -c "ALTER TABLE PUBLIC.testtable RENAME TO
testtable_bak;"
```

- 创建新的 testtable 表。

```
postgres=# CREATE TABLE PUBLIC.testtable(a int, b int, c int);
```

- 还原依赖于 testtable 的视图。

```
gsql -p 15400 postgres -r -f backup/MPPDB_backup.sql
```

(7) 使用 gs_dumpall 一次导出 GBase 8c 的所有数据库。gs_dumpall 仅支持纯文本格式导出。只能使用 gsql 恢复 gs_dumpall 导出的转储内容。

```
gs_dumpall -f backup/bkp2.sql -p 15400
gs_dump[port='15400'][dbname='postgres'][2024-02-29 09:55:09]: The total
objects number is 2371. gs_dump[port='15400'][dbname='postgres'][2024-02-29
09:55:35]: [100.00%] 2371 objects have been dumped.
gs_dump[port='15400'][dbname='postgres'][2024-02-29 09:55:46]: dump database
dbname='postgres' successfully
gs_dump[port='15400'][dbname='postgres'][2024-02-29 09:55:46]: total time:
55567 ms gs_dumpall[port='15400'][2024-02-29 09:55:46]: dumpall operation
successful gs_dumpall[port='15400'][2024-02-29 09:55:46]: total time: 56088 ms
```

(8) 执行 gsql 程序，使用如下选项导入由 gs_dump/gs_dumpall 生成导出文件夹（纯文本格式）的 MPPDB_backup.sql 文件到 postgres 数据库。

```
gsql -d postgres -p 15400 -W gbase,123 -f
/home/gbase/data/backup/MPPDB_backup.sql
……
```

```
total time: 30476  ms
```

gs_restore 用来导入由 gs_dump 生成的导出文件。

(9) 执行 gs_restore，将导出的 MPPDB_backup.dmp 文件（自定义归档格式）导入到 postgres 数据库。

```
gs_restore -W gbase,123 /home/gbase/data/backup/MPPDB_backup.dmp -p 15400 -d postgres
gs_restore: restore operation successful
gs_restore: total time: 13053 ms
```

(10) 执行 gs_restore，将导出的 MPPDB_backup.tar 文件（tar 格式）导入到 postgres 数据库。

```
gs_restore /home/gbase/data/backup/MPPDB_backup.tar -p 15400 -d postgres
gs_restore[2024-03-01 19:16:26]: restore operation successful
gs_restore[2024-03-01 19:16:26]: total time: 21203 ms
```

(11) 执行 gs_restore，将导出的 MPPDB_backup 文件（目录格式）导入到 postgres 数据库。

```
gs_restore /home/gbase/data/backup/MPPDB_backup -p 15400 -d postgres
gs_restore[2024-03-01 19:16:26]: restore operation successful
gs_restore[2024-03-01 19:16:26]: total time: 21003 ms
```

(12) 执行 gs_restore，使用自定义归档格式的 MPPDB_backup.dmp 文件来进行如下导入操作。

导入 PUBLIC 模式下所有对象的定义和数据。在导入时会先删除已经存在的对象，如果原对象存在跨模式的依赖则需手工强制干预。

```
gs_restore /home/gbase/data/backup/MPPDB_backup.dmp -p 15400 -d postgres -e -c -n PUBLIC
gs_restore: [archiver (db)] Error while PROCESSING TOC:
gs_restore: [archiver (db)] Error from TOC entry 313; 1259 337399 TABLE table1 gaussdba
gs_restore: [archiver (db)] could not execute query: ERROR: cannot drop table table1 because other objects depend on it
DETAIL: view t1.v1 depends on table table1
HINT: Use DROP ... CASCADE to drop the dependent objects too.
Command was: DROP TABLE public.table1;
```

手工删除依赖，导入完成后再重新创建。

```
gs_restore /home/gbase/data/backup/MPPDB_backup.dmp -p 15400 -d postgres -e -c -n PUBLIC
```

```
gs_restore[2024-03-01 19:16:26]: restore operation successful
gs_restore[2024-03-01 19:16:26]: total time: 2203 ms
```

(13) 执行 gs_restore，使用自定义归档格式的 MPPDB_backup.dmp 文件来进行如下导入操作。

只导入 PUBLIC 模式下表 table1 的定义。

```
gs_restore /home/gbase/data/backup/MPPDB_backup.dmp -p 15400 -d postgres -e -c
-s -n PUBLIC -t table1
gs_restore[2024-03-01 19:16:26]: restore operation successful
gs_restore[2024-03-01 19:16:26]: total time: 21000 ms
```

(14) 执行 gs_restore，使用自定义归档格式的 MPPDB_backup.dmp 文件来进行如下导入操作。

只导入 PUBLIC 模式下表 table1 的数据。

```
gs_restore /home/gbase/data/backup/MPPDB_backup.dmp -p 15400 -d postgres -e -a
-n PUBLIC -t table1
gs_restore[2024-03-01 19:16:26]: restore operation successful
gs_restore[2024-03-01 19:16:26]: total time: 20203 ms
```

4.4 物理备份与恢复

GBase 8c 部署成功后，在数据库运行的过程中，会遇到各种问题及异常状态。GBase 8c 提供了 gs_backup 工具帮助 GBase 8c 备份、恢复重要数据、显示帮助信息和版本号信息。使用说明详见《GBase 8c V5_5.0.0_工具与命令参考手册》中”gs_backup”章节。

GBase 8c 提供了 gs_basebackup 工具做基础的物理备份。gs_basebackup 的实现目标是对服务器数据库文件的二进制进行拷贝，其实现原理使用了复制协议。远程执行 gs_basebackup 时，需要使用系统管理员账户。gs_basebackup 当前支持热备份模式和压缩格式备份。使用说明详见《GBase 8c V5_5.0.0_工具与命令参考手册》中”gs_basebackup”章节。

gs_probackup 是用于管理 GBase 8c 数据库备份和恢复的工具。它对 GBase 8c 实例进行定期备份，以便在数据库出现故障时能够恢复服务器。使用说明详见《GBase 8c V5_5.0.0_工具与命令参考手册》中”gs_probackup”章节。

示例

(1) 使用 gs_backup 脚本备份数据库主机。

```
gs_backup -t backup --backup-dir=/opt/software/backup_dir -h plat1 --parameter
Backing up GBase 8c.
Parsing configuration files.
Successfully parsed the configuration file.
```

```
Performing remote backup.
Remote backup succeeded.
Successfully backed up GBase 8c.
```

使用 gs_backup 脚本恢复数据库主机。

```
gs_backup -t restore --backup-dir=/opt/software/backup_dir -h plat1 --parameter
Restoring GBase 8c.
Parsing the configuration file.
Successfully parsed configuration files.
Performing remote restoration.
Remote restoration succeeded.
Successfully restored GBase 8c.
```

(2) 使用 gs_basebackup 工具备份：

```
gs_basebackup -D /home/gbase/data/backup -h 10.0.7.16 -p 15400
INFO:  The starting position of the xlog copy of the full build is: 0/6000028.
The slot minimum LSN is: 0/0.
[2024-02-29 16:26:39]:begin build tablespace list
[2024-02-29 16:26:39]:finish build tablespace list
[2024-02-29 16:26:39]:begin get xlog by xlogstream
[2024-02-29 16:26:39]: check identify system success
[2024-02-29 16:26:39]: send START_REPLICATION 0/6000000 success
[2024-02-29 16:26:39]: keepalive message is received
[2024-02-29 16:26:39]: keepalive message is received
[2024-02-29 16:26:44]:gs_basebackup: base backup successfully
```

(3) gs_probackup 备份流程，以下例说明，请根据实际情况修改：

① 首先初始化备份目录，在指定的目录下创建 backups/和 wal/子目录，分别用于存放备份文件和 WAL 文件。

```
gs_probackup init -B /opt/backup/
```

② 添加一个新的备份实例。gs_probackup 可以在同一个备份目录下存放多个数据库实例的备份。

```
gs_probackup add-instance -B /opt/backup/ -D /opt/database/install/ --instance
instance_name
```

③ 创建指定实例的备份。在进行增量备份之前，必须至少创建一次全量备份。

```
gs_probackup backup -B backup_dir --instance instance_name -b backup_mode
```

④ 从指定实例的备份中恢复数据。

```
gs_probackup restore -B backup_dir --instance instance_name -D pgdata-path -i
backup_id
```

4.4.1 PITR 恢复

背景信息

当数据库崩溃或希望回退到数据库之前的某一状态时，GBase 8c 的即时恢复功能（Point-In-Time Recovery，简称 PITR）可以支持恢复到备份归档数据之后的任意时间点。

说明

- PITR 仅支持恢复到物理备份数据之后的某一时间点。
- 仅主节点可以进行 PITR 恢复，备机需要进行全量 build 达成与主机数据同步。

前提条件

- 基于经过物理备份的全量数据文件。
- 基于已归档的 WAL 日志文件。

PITR 恢复流程

(1) 将物理备份的文件替换目标数据库目录。

(2) 删除数据库目录下 pg_xlog/中的所有文件。

(3) 将归档的 WAL 日志文件复制到 pg_xlog 文件中（此步骤可以省略，通过配置 recovery.conf 恢复命令文件中的 restore_command 项替代）。

(4) 在数据库目录下创建恢复命令文件 recovery.conf，指定数据库恢复的程度。

(5) 启动数据库。

(6) 连接数据库，查看是否恢复到希望预期的状态。

(7) 若已经恢复到预期状态，通过 pg_xlog_replay_resume()指令使主节点对外提供服务。

文件配置

recovery.conf 文件配置，参考如下：

归档恢复配置

- restore_command = string

这个 SHELL 命令是获取 WAL 文件系列中已归档的 WAL 文件。字符串中的任何一个%f 是用归档检索中的文件名替换，并且%p 是用服务器上的复制目的地的路径名替换。 任意一个%r 是用包含最新可用重启点的文件名替换。

示例：

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

- archive_cleanup_command = string

这个选项参数声明一个 shell 命令。在每次重启时会执行这个 shell 命令。archive_cleanup_command 为清理备库不需要的归档 WAL 文件提供一个机制。 任何一个%r 由包含最新可用重启点的文件名代替。这是最早的文件， 因此必须保留以允许恢复能够重新启动，因此所有早于%r 的文件可以安全的移除。

示例：

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

需要注意的是，如果多个备服务器从相同的归档路径恢复时， 需要确保在任何一个备服务器在需要之前，不能删除 WAL 文件。

- recovery_end_command = string

这个参数是可选的，用于声明一个只在恢复完成时执行的 SHELL 命令。

recovery_end_command 是为以后的复制或恢复提供一个清理机制。

恢复目标设置

- recovery_target_name = string

此参数声明命名还原到一个使用 pg_create_restore_point()创建的还原点。

示例:

```
recovery_target_name = 'restore_point_1'
```

- recovery_target_time = timestamp

此参数声明命名还原到一个指定时间戳。示例：

```
recovery_target_time = '2020-01-01 12:00:00'
```

recovery_target_xid = string 这个参数声明还原到一个事务 ID。示例：

```
recovery_target_xid = '3000'
```

- recovery_target_lsn = string

  这个参数声明还原到日志的指定 LSN 点。示例：

```
recovery_target_lsn = '0/0FFFFFF'
```

- recovery_target_inclusive = boolean

  声明是否在指定恢复目标(true)之后停止，或在这(false)之前停止。该声明仅支持恢复目标为 recovery_target_time，recovery_target_xid 和 recovery_target_lsn 的配置。

  示例：

```
recovery_target_inclusive = true
```

说明

recovery_target_name，recovery_target_time，recovery_target_xid，recovery_target_lsn 这四个配置项仅同时支持一项。如果不配置任何恢复目标，或配置目标不存在，则默认恢复到最新的 WAL 日志点。

4.5 闪回恢复

闪回恢复功能是数据库恢复技术的一环，可以有选择性的撤销一个已提交事务的影响，将数据从人为不正确的操作中进行恢复。在采用闪回技术之前，只能通过备份恢复、PITR 等手段找回已提交的数据库修改，恢复时长需要数分钟甚至数小时。采用闪回技术后，恢复已提交的数据库修改前的数据，只需要秒级，而且恢复时间和数据库大小无关。

说明

ASTORE 引擎暂不支持闪回功能。

4.5.1 闪回查询

背景信息

闪回查询可以查询过去某个时间点表的某个 snapshot 数据，这一特性可用于查看和逻辑重建意外删除或更改的受损数据。闪回查询基于 MVCC 多版本机制，通过检索查询旧版本，获取指定老版本数据。

语法

```
{[ ONLY ] table_name [ * ] [ partition_clause ] [ [ AS ] alias [ ( column_alias
[, ...] ) ] ]
[ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
[TIMECAPSULE { TIMESTAMP | CSN } expression ]
|( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
|with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] |
column_definition [, ...] ) ]
|function_name ( [ argument [, ...] ] ) AS ( column definition [, ...] )
|from_item [ NATURAL ] join_type from_item [ ON join_condition | USING
( join_column [, ...] ) ]}
```

语法树中“TIMECAPSULE {TIMESTAMP | CSN} expression”为闪回功能新增表达方式，其中 TIMECAPSULE 表示使用闪回功能，TIMESTAMP 以及 CSN 表示闪回功能使用具体时间点信息或使用 CSN（commit sequence number）信息。

参数说明

- TIMESTAMP

指要查询某个表在 TIMESTAMP 这个时间点上的数据，TIMESTAMP 指一个具体的历史时间。

- CSN

指要查询整个数据库逻辑提交序下某个 CSN 点的数据，CSN 指一个具体逻辑提交时间点，数据库中的 CSN 为写一致性点，每个 CSN 代表整个数据库的一个一致性点，查询某个 CSN 下的数据代表 SQL 查询数据库在该一致性点的相关数据。

4.5.2 闪回表

背景信息

闪回表可以将表恢复至特定时间点，当逻辑损坏仅限于一个或一组表，而不是整个数据

库时，此特性可以快速恢复表的数据。闪回表基于 MVCC 多版本机制，通过删除指定时间点和该时间点之后的增量数据，并找回指定时间点和当前时间点删除的数据，实现表级数据还原。

前提条件

undo_retention_time 参数用于设置 undo 旧版本的保留时间。

语法

```
TIMECAPSULE TABLE  table_name TO { TIMESTAMP |  CSN }  expression
```

4.5.3 闪回 DROP/TRUNCATE

背景信息

闪回 DROP：可以恢复意外删除的表，从回收站（recyclebin）中恢复被删除的表及其附属结构如索引、表约束等。闪回 drop 是基于回收站机制，通过还原回收站中记录的表的物理文件，实现已 drop 表的恢复。

闪回 TRUNCATE：可以恢复误操作或意外被进行 truncate 的表，从回收站中恢复 被 truncate 的表及索引的物理数据。闪回 truncate 基于回收站机制，通过还原回收 站中记录的表的物理文件，实现已 truncate 表的恢复。

前提条件

- 开启 enable_recyclebin 参数，启用回收站。
- recyclebin_retention_time 参数用于设置回收站对象保留时间，超过该时间的回收 站对象将被自动清理。

相关语法

- 删除表

```
DROP TABLE table_name [PURGE]
```

- 清理回收站对象

```
PURGE { TABLE { table_name }
| INDEX { index_name }
| RECYCLEBIN
}
```

- 闪回被删除的表

```
TIMECAPSULE TABLE { table_name } TO BEFORE DROP [RENAME TO new_tablename]
```

- 截断表

```
TRUNCATE TABLE { table_name } [ PURGE ]
```

- 闪回截断的表

```
TIMECAPSULE TABLE { table_name } TO BEFORE TRUNCATE
```

参数说明

- DROP/TRUNCATE TABLE table_name PURGE ：默认将表数据放入回收站中，PURGE 直接清理。
- PURGE RECYCLEBIN ：表示清理回收站对象。
- TO BEFORE DROP ：使用这个子句检索回收站中已删除的表及其子对象。

  可以指定原始用户指定的表的名称，或对象删除时数据库分配的系统生成名称。

  回收站中系统生成的对象名称是唯一的。因此，如果指定系统生成名称，那么数据库检索指定的对象。使用“select * from gs_recyclebin;”语句查看回收站中的内容。

  如果指定了用户指定的名称，且如果回收站中包含多个该名称的对象，然后数据库检索回收站中最近移动的对象。如果想要检索更早版本的表，你可以这样做：

  - 指定你想要检索的表的系统生成名称。
  - 执行 TIMECAPSULE TABLE ... TO BEFORE DROP 语句，直到你要检索的表。
  - 恢复 DROP 表时，只恢复基表名，其他子对象名均保持回收站对象名。用户可根据需要，执行 DDL 命令手工调整子对象名。
  - 回收站对象不支持 DML、DCL、DDL 等写操作，不支持 DQL 查询操作（后续支持）。
  - 闪回点和当前点之间，执行过修改表结构或影响物理结构的语句，闪回失败。执行过 DDL 的表进行闪回操作报错：“ERROR： The table definition of %s has been changed.”。涉及 namespace、表名改变等操作的 DDL 执行闪回操作报错：ERROR: recycle object %s desired does not exis；

- RENAME TO ：为从回收站中检索的表指定一个新名称。
- TO BEFORE TRUNCATE ：闪回到 TRUNCATE 之前。

4.6 导出数据

4.6.1 使用 gs_dump 和 gs_dumpall 命令导出数据

4.6.1.1 概述

GBase 8c 提供的 gs_dump 和 gs_dumpall 工具，能够帮助用户导出需要的数据库对象或其相关信息。通过导入工具将导出的数据信息导入至需要的数据库，可以完成数据库信息的迁移。gs_dump 支持导出单个数据库或其内的对象，而 gs_dumpall 支持导出 GBase 8c 中所有数据库或各库的公共全局对象。详细的使用场景见表 4-5。

表 4-3 适用场景

适用场景	支持的导出粒度	支持的导出格式	配套的导入方法
导出单个数据库	**数据库级导出。** 导出全量信息。 使用导出的全量信息可以创建一个与当前库相同的数据库，且库中数据也与当前库相同。 仅导出库中所有对象的定义，包含库定义、函数定义、模式定义、表定义、索引定义和存储过程定义等。 使用导出的对象定义，可以快速创建一个相同的数据库，但是库中并无原数据库的数据。 仅导出数据。	纯文本格式 自定义归档格式 目录归档格式 tar 归档格式	纯文本格式数据文件导入使用 gsql 工具，参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“客户端工具 > gsql”章节。 自定义归档格式、目录归档格式和 tar 归档格式数据文件导入请参见使用 gs_restore 命令导入数据。
	模式级导出。 导出模式的全量信息。 仅导出模式中数据。 仅导出对象的定义，包含表定义、存		

适用场景	支持的导出粒度	支持的导出格式	配套的导入方法
	储过程定义和索引定义等。		
	表级导出。 导出表的全量信息。 仅导出表中数据。 仅导出表的定义。		
导出所有数据库	**数据库级导出。** 导出全量信息。 使用导出的全量信息可以创建与当前主机相同的一个主机环境，拥有相同数据库和公共全局对象，且库中数据也与当前各库相同。 仅导出各数据库中的对象定义，包含表空间、库定义、函数定义、模式定义、表定义、索引定义和存储过程定义等。 使用导出的对象定义，可以快速创建与当前主机相同的一个主机环境，拥有相同的数据库和表空间，但是库中并无原数据库的数据。 仅导出数据。	纯文本格式	数据文件导入请参见使用 gsql 元命令导入数据。
	各库公共全局对象导出。 仅导出表空间信息。 仅导出角色信息。 导出角色与表空间。		

gs_dump 和 gs_dumpall 通过-U 指定执行导出的用户帐户。如果当前使用的帐户不具备导出所要求的权限时，会无法导出数据。此时，需先将具有权限的角色赋权给无权限角色，然后在导出命令中设置 -role 参数来指定具备权限的角色。在执行命令后，gs_dump 和 gs_dumpall 会使用 -role 参数指定的角色，完成导出动作。可使用该功能的场景请参见表 4-5，

详细操作请参见无权限角色导出数据。

gs_dump 和 gs_dumpall 通过对导出的数据文件加密，导入时对加密的数据文件进行解密，可以防止数据信息泄露，为数据库的安全提供保证。注意，使用 gs_dump 加密的纯文本格式文件，如果导出的数据库中包含存储过程，因 gsql 不支持解密导入存储过程和函数，因此如果导出的数据库中包含存储过程/函数，请使用另外三种模式导出数据库，并使用 gs_restore 恢复。

gs_dump 和 gs_dumpall 工具在进行数据导出时，其他用户可以访问数据库（读或写）。

gs_dump 和 gs_dumpall 工具支持导出完整一致的数据。例如，T1 时刻启动 gs_dump 导出 A 数据库，或者启动 gs_dumpall 导出 GBase 8c 数据库，那么导出数据结果将会是 T1 时刻 A 数据库或者该 GBase 8c 数据库的数据状态，T1 时刻之后对 A 数据库或 GBase 8c 数据库的修改不会被导出。

注意事项

- 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，可根据需要谨慎编辑导出文件。
- 如果数据库中包含的对象数量（数据表、视图、索引）在 50 万以上，为了提高性能且避免出现内存问题，建议通过 gs_guc 工具设置数据库节点的如下参数（如果参数值大于如下建议值，则无需设置）。

```
gs_guc set -N all -I all -c 'max_prepared_transactions = 1000'
gs_guc set -N all -I all -c 'max_locks_per_transaction = 512'
```

若设置如上参数，则需重启数据库使参数生效。

```
gs_om -t stop && gs_om -t start
```

- 为了保证数据一致性和完整性，导出工具会对需要转储的表设置共享锁。如果表在别的事务中设置了共享锁，gs_dump 和 gs_dumpall 会等待锁释放后锁定表。如果无法在指定时间内锁定某个表，转储会失败。用户可以通过指定 -lock-wait-timeout 选项，自定义等待锁超时时间。
- 由于 gs_dumpall 读取所有数据库中的表，因此必须以 GBase 8c 管理员身份进行连接，才能导出完整文件。在使用 gsql 执行脚本文件导入时，同样需要管理员权限，以便添加用户和组，以及创建数据库。

4.6.1.2 导出单个数据库

导出数据库

GBase 8c 支持使用 gs_dump 工具导出某个数据库级的内容，包含数据库的数据和所有对象定义。可根据需要自定义导出如下信息：

- 导出数据库全量信息，包含数据和所有对象定义。

  使用导出的全量信息可以创建一个与当前库相同的数据库，且库中数据也与当前库相同。

- 仅导出所有对象定义，包括：库定义、函数定义、模式定义、表定义、索引定义和存储过程定义等。

  使用导出的对象定义，可以快速创建一个相同的数据库，但是库中并无原数据库的数据。

- 仅导出数据，不包含所有对象定义。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用 gs_dump 导出 postgres 数据库。

```
gs_dump -U jack -f /home/gbase/backup/userdatabase_backup.tar -p 15400 postgres
-F t
Password:
```

表 4-4 常用参数说明

参数	参数说明	举例
-U	连接数据库的用户名。 说明： 不指定连接数据库的用户名时，默认以安装时创建的初始系统管理员连接。	-U jack
-W	指定用户连接的密码。 如果主机的认证策略是 trust，则不会对数据库管理员进行密码验证，即无需输入-W 选项。 如果没有-W 选项，并且不是数据库管理员，会提示用户输入密码。	-W abcd@123

参数	参数说明	举例
-f	将导出文件发送至指定目录文件夹。如果这里省略，则使用标准输出。如果输出格式为(-F c/-F d/-F t)时，必须指定-f 参数。	-f /home/gbase/backup/postgres_backup.tar
-p	指定服务器所侦听的 TCP 端口或本地 Unix 域套接字后缀，以确保连接。	-p15400
dbname	需要导出的数据库名称。	postgres
-F	选择导出文件格式。-F 参数值如下： p：纯文本格式 c：自定义归档 d：目录归档格式 t：tar 归档格式	-F t

其他参数说明请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_dump”章节。

示例

示例一：执行 gs_dump，导出 postgres 数据库全量信息，导出文件格式为 sql 文本格式。

```
gs_dump -U jack -f /home/gbase/backup/postgres_backup.sql -p 15400 postgres -F
p
Password:
gs_dump[port='15400'][postgres][2024-03-01 15:36:13]: dump database postgres
successfully
gs_dump[port='15400'][postgres][2024-03-01 15:36:13]: total time: 3793  ms
```

示例二：执行 gs_dump，仅导出 postgres 数据库中的数据，不包含数据库对象定义，导出文件格式为自定义归档格式。

```
gs_dump -U jack -f /home/gbase/backup/postgres_data_backup.dmp -p 15400 postgres
-a -F c
Password:
gs_dump[port='15400'][postgres][2024-03-01 15:36:13]: dump database postgres
successfully
gs_dump[port='15400'][postgres][2024-03-01 15:36:13]: total time: 3793  ms
```

示例三：执行 gs_dump，仅导出 postgres 数据库所有对象的定义，导出文件格式为 sql 文本格式。

```
gs_dump -f /home/gbase/backup/postgres_def_backup.sql -p 15400 postgres -s -F p
Password:
gs_dump[port='15400'][postgres][2024-03-01  15:04:14]: dump database postgres
successfully
gs_dump[port='15400'][postgres][2024-03-01  15:04:14]: total time: 472 ms
```

示例四：执行 gs_dump，仅导出 postgres 数据库的所有对象的定义，导出文件格式为文本格式，并对导出文件进行加密。

```
gs_dump -f /home/gbase/backup/postgres_def_backup.sql -p 15400 postgres
--with-encryption AES128 --with-key 1234567812345678 -s -F p
Password:
gs_dump[port='15400'][postgres][2024-03-01  11:25:18]: dump database postgres
successfully
gs_dump[port='15400'][postgres][2024-03-01  11:25:18]: total time: 1161  ms
```

导出模式

GBase 8c 目前支持使用 gs_dump 工具导出模式级的内容，包含模式的数据和定义。用户可通过灵活的自定义方式导出模式内容，不仅支持选定一个模式或多个模式的导出，还支持排除一个模式或者多个模式的导出。可根据需要自定义导出如下信息：

- 导出模式全量信息，包含数据和对象定义。
- 仅导出数据，即模式包含表中的数据，不包含对象定义。
- 仅导出模式对象定义，包括：表定义、存储过程定义和索引定义等。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用 gs_dump 同时导出 hr 和 public 模式。

```
gs_dump -U jack -f /home/gbase/backup/MPPDB_schema_backup -p 15400
human_resource -n hr -n public -F d
Password:
```

表 4-5 常用参数说明

参数	参数说明	举例

参数	参数说明	举例
-U	连接数据库的用户名。	-U jack
-W	指定用户连接的密码。 如果主机的认证策略是 trust，则不会对数据库管理员进行密码验证，即无需输入-W 选项。 如果没有-W 选项，并且不是数据库管理员，会提示用户输入密码。	-W abcd@123
-f	将导出文件发送至指定目录文件夹。如果这里省略，则使用标准输出。	-f /home/gbase/backup/MPPDB_schema_backup
-p	指定服务器所侦听的 TCP 端口或本地 Unix 域套接字后缀，以确保连接。	-p 15400
dbname	需要导出的数据库名称。	human_resource
-n	只导出与模式名称匹配的模式，此选项包括模式本身和所有它包含的对象。 单个模式：-nschemaname 多个模式：多次输入-nschemaname	单个模式：-n hr 多个模式：-n hr -n public
-F	选择导出文件格式。-F 参数值如下： p：纯文本格式 c：自定义归档 d：目录归档格式 t：tar 归档格式	-F d

示例

示例一：执行 gs_dump，导出 hr 模式全量信息，导出文件格式为文本格式。

```
gs_dump -f /home/gbase/backup/MPPDB_schema_backup.sql -p 15400 human_resource -n hr -F p
```

```
Password:
gs_dump[port='15400'][human_resource][2024-03-01 16:05:55]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 16:05:55]: total time: 2425
ms
```

示例二：执行 gs_dump，仅导出 hr 模式的数据，导出文件格式为 tar 归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_schema_data_backup.tar -p 15400
human_resource -n hr -a -F t
Password:
gs_dump[port='15400'][human_resource][2024-03-01  15:07:16]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01  15:07:16]: total time: 1865
ms
```

示例三：执行 gs_dump，仅导出 hr 模式的定义，导出文件格式为目录归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_schema_def_backup -p 15400 human_resource -n
hr -s -F d
Password:
gs_dump[port='15400'][human_resource][2024-03-01  15:11:34]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01  15:11:34]: total time: 1652
ms
```

示例四：执行 gs_dump，导出 human_resource 数据库时，排除 hr 模式，导出文件格式为自定义归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_schema_backup.dmp -p 15400 human_resource -N
hr -F c
Password:
gs_dump[port='15400'][human_resource][2024-03-01 16:06:31]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 16:06:31]: total time: 2522
ms
```

示例五：执行 gs_dump，同时导出 hr 和 public 模式，且仅导出模式定义，导出文件格式为 tar 归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_schema_backup1.tar -p 15400 human_resource
-n hr -n public -s -F t
gs_dump[port='15400'][human_resource][2024-03-01 16:07:16]: dump database
human_resource successfully
```

```
gs_dump[port='15400'][human_resource][2024-03-01 16:07:16]: total time: 2132
ms
```

示例六：执行 gs_dump，导出 human_resource 数据库时，排除 hr 和 public 模式，导出文件格式为自定义归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_schema_backup2.dmp -p 15400 human_resource
-N hr -N public -F c
Password:
gs_dump[port='15400'][human_resource][2024-03-01 16:07:55]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 16:07:55]: total time: 2296
ms
```

示例七：执行 gs_dump，导出 public 模式下所有表（视图、序列和外表）和 hr 模式中 staffs 表，包含数据和表定义，导出文件格式为自定义归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_backup3.dmp -p 15400 human_resource -t
public.* -t hr.staffs -F c
Password:
gs_dump[port='15400'][human_resource][2018-12-13 09:40:24]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2018-12-13 09:40:24]: total time: 896  ms
```

导出表

GBase 8c 支持使用 gs_dump 工具导出表级的内容，包含表定义和表数据。视图、序列和外表属于特殊的表。用户可通过灵活的自定义方式导出表内容，不仅支持选定一个表或多个表的导出，还支持排除一个表或者多个表的导出。可根据需要自定义导出如下信息：

- 导出表全量信息，包含表数据和表定义。
- 仅导出数据，不包含表定义。
- 仅导出表定义。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用 gs_dump 同时导出指定表 hr.staffs 和 hr.employments。

```
gs_dump -U jack -f /home/gbase/backup/MPPDB_table_backup -p 15400 human_resource
-t hr.staffs -t hr.employments -F d
Password:
```

表 4-6 常用参数说明

参数	参数说明	举例
-U	连接数据库的用户名。	-U jack
-W	指定用户连接的密码。 如果主机的认证策略是 trust，则不会对数据库管理员进行密码验证，即无需输入-W 选项。 如果没有-W 选项，并且不是数据库管理员，会提示用户输入密码。	-W abcd@123
-f	将导出文件发送至指定目录文件夹。如果这里省略，则使用标准输出。	-f /home/gbase/backup/MPPDB_table_backup
-p	指定服务器所侦听的 TCP 端口或本地 Unix 域套接字后缀，以确保连接。	-p 15400
dbname	需要导出的数据库名称。	human_resource
-t	指定导出的表（或视图、序列、外表），可以使用多个-t 选项来选择多个表，也可以使用通配符指定多个表对象。当使用通配符指定多个表对象时，注意给 pattern 打引号，防止 shell 扩展通配符。 单个表：-t schema.table 多个表：多次输入-t schema.table	单个表：-t hr.staffs 多个表：-t hr.staffs -t hr.employments
-F	选择导出文件格式。-F 参数值如下： p：纯文本格式 c：自定义归档 d：目录归档格式 t：tar 归档格式	-F d
-T	不转储的表（或视图、或序列、或外表）对象列表，可以使用多个-t 选项来选择多个表，也可以使用通配符制定多个表	-T table1

参数	参数说明	举例
	对象。 当同时输入-t 和-T 时，会转储在-t 列表中，而不在-T 列表中的表对象。	

其他参数说明请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_dump”章节。

示例

下方示例导出后，在导入恢复前，需要确保存在导出表所在的 schema。

示例一：执行 gs_dump，导出表 hr.staffs 的定义和数据，导出文件格式为文本格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_backup.sql -p 15400 human_resource -t
hr.staffs  -F p
Password:
gs_dump[port='15400'][human_resource][2024-03-01 17:05:10]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 17:05:10]: total time: 3116
ms
```

示例二：执行 gs_dump，只导出表 hr.staffs 的数据，导出文件格式为 tar 归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_data_backup.tar -p 15400
human_resource -t hr.staffs -a -F t
Password:
gs_dump[port='15400'][human_resource][2024-03-01 17:04:26]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 17:04:26]: total time: 2570
ms
```

示例三：执行 gs_dump，导出表 hr.staffs 的定义，导出文件格式为目录归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_def_backup -p 15400 human_resource -t
hr.staffs -s -F d
Password:
gs_dump[port='15400'][human_resource][2024-03-01 17:03:09]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 17:03:09]: total time: 2297
ms
```

示例四：执行 gs_dump，不导出表 hr.staffs，导出文件格式为自定义归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_backup4.dmp -p 15400 human_resource -T
hr.staffs -F c
Password:
gs_dump[port='15400'][human_resource][2024-03-01 17:14:11]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 17:14:11]: total time: 2450
ms
```

示例五：执行 gs_dump，同时导出两个表 hr.staffs 和 hr.employments，导出文件格式为文本格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_backup1.sql -p 15400 human_resource -t
hr.staffs -t hr.employments -F p
Password:
gs_dump[port='15400'][human_resource][2024-03-01 17:19:42]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 17:19:42]: total time: 2414
ms
```

示例六：执行 gs_dump，导出时，排除两个表 hr.staffs 和 hr.employments，导出文件格式为文本格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_backup2.sql -p 15400 human_resource -T
hr.staffs -T hr.employments -F p
Password:
gs_dump[port='15400'][human_resource][2024-03-01 17:21:02]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 17:21:02]: total time: 3165
ms
```

示例七：执行 gs_dump，导出表 hr.staffs 的定义和数据，只导出表 hr.employments 的定义，导出文件格式为 tar 归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_backup3.tar -p 15400 human_resource -t
hr.staffs -t hr.employments --exclude-table-data hr.employments -F t
Password:
gs_dump[port='15400'][human_resource][2024-03-01  11:32:02]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01  11:32:02]: total time: 1645
ms
```

示例八：执行 gs_dump，导出表 hr.staffs 的定义和数据，并对导出文件进行加密，导出文件格式为文本格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_backup4.sql -p 15400 human_resource -t
hr.staffs --with-encryption AES128 --with-key abcdefg_?1234567 -F p
Password:
gs_dump[port='15400'][human_resource][2024-03-01  11:35:30]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01  11:35:30]: total time: 6708
ms
```

示例九：执行 gs_dump，导出 public 模式下所有表（包括视图、序列和外表）和 hr 模式中 staffs 表，包含数据和表定义，导出文件格式为自定义归档格式。

```
gs_dump -f /home/gbase/backup/MPPDB_table_backup5.dmp -p 15400 human_resource -t
public.* -t hr.staffs -F c
Password:
gs_dump[port='15400'][human_resource][2018-12-13 09:40:24]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2018-12-13 09:40:24]: total time: 896  ms
```

示例十： 执行 gs_dump，仅导出依赖于 t1 模式下的 test1 表对象的视图信息，导出文件格式为目录归档格式。

```
gs_dump -U jack -f /home/gbase/backup/MPPDB_view_backup6 -p 15400 human_resource
-t t1.test1 --include-depend-objs --exclude-self -F d
Password:
gs_dump[port='15400'][jack][2024-03-01  17:21:18]: dump database
human_resource successfully
gs_dump[port='15400'][jack][2024-03-01  17:21:23]: total time: 4239  ms
```

4.6.1.3 导出所有数据库

导出所有数据库

GBase 8c 支持使用 gs_dumpall 工具导出所有数据库的全量信息，包含 GBase 8c 中每个数据库信息和公共的全局对象信息。可根据需要自定义导出如下信息：

- 导出所有数据库全量信息，包含 GBase 8c 中每个数据库信息和公共的全局对象信息（包含角色和表空间信息）。

使用导出的全量信息可以创建与当前主机相同的一个主机环境，拥有相同数据库和公共全局对象，且库中数据也与当前各库相同。

- 仅导出数据，即导出每个数据库中的数据，且不包含所有对象定义和公共的全局对象信息。

- 仅导出所有对象定义，包括：表空间、库定义、函数定义、模式定义、表定义、索引定义和存储过程定义等。

使用导出的对象定义，可以快速创建与当前主机相同的一个主机环境，拥有相同的数据库和表空间，但是库中并无原数据库的数据。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用 gs_dumpall 一次导出所有数据库信息。

```
gs_dumpall  -U gbase -f /home/gbase/backup/MPPDB_backup.sql -p 15400
```

常用参数说明请见下表，其他参数说明请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_dumpall”章节。

表 4-7 常用参数说明

参数	参数说明	举例
-U	连接数据库的用户名，需要是 GBase 8c 管理员用户。	-U gbase
-W	指定用户连接的密码。 如果主机的认证策略是 trust，则不会对数据库管理员进行密码验证，即无需输入-W 选项； 如果没有-W 选项，并且不是数据库管理员，会提示用户输入密码。	-W abcd@123
-f	将导出文件发送至指定目录文件夹。如果这里省略，则使用标准输出。	-f /home/gbase/backup/MPPDB_backup.sql
-p	指定服务器所监听的 TCP 端口或本地 Unix 域套接字后缀，以确保连接。	-p 15400

示例

示例一：执行 gs_dumpall，导出所有数据库全量信息（gbase 用户为管理员用户），导出文件为文本格式。执行命令后，会有很长的打印信息，最终出现 total time 即代表执行成功。示例中将不体现中间的打印信息。

```
gs_dumpall -U gbase -f /home/gbase/backup/MPPDB_backup.sql -p 15400
Password:
gs_dumpall[port='15400'][2024-03-01 15:57:31]: dumpall operation successful
gs_dumpall[port='15400'][2024-03-01 15:57:31]: total time: 9627  ms
```

示例二：执行 gs_dumpall，仅导出所有数据库定义（gbase 用户为管理员用户），导出文件为文本格式。执行命令后，会有很长的打印信息，最终出现 total time 即代表执行成功。示例中将不体现中间的打印信息。

```
gs_dumpall  -U gbase -f /home/gbase/backup/MPPDB_backup.sql -p 15400 -s
Password:
gs_dumpall[port='15400'][2024-03-01  11:28:14]: dumpall operation successful
gs_dumpall[port='15400'][2024-03-01  11:28:14]: total time: 4147  ms
```

示例三：执行 gs_dumpall，仅导出所有数据库中数据，并对导出文件进行加密，导出文件为文本格式。执行命令后，会有很长的打印信息，最终出现 total time 即代表执行成功。示例中将不体现中间的打印信息。

```
gs_dumpall -f /home/gbase/backup/MPPDB_backup.sql -p 15400 -a --with-encryption
AES128 --with-key abcdefg_?1234567
gs_dumpall[port='15400'][2024-03-01  11:32:26]: dumpall operation successful
gs_dumpall[port='15400'][2024-03-01  11:23:26]: total time: 4147  ms
```

导出全局对象

GBase 8c 支持使用 gs_dumpall 工具导出所有数据库公共的全局对象，包含数据库用户和组、表空间及属性（例如：适用于数据库整体的访问权限）信息。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用 gs_dumpall 导出表空间对象信息。

```
gs_dumpall -U gbase -f /home/gbase/backup/MPPDB_tablespace.sql -p 15400 -t
Password:
```

表 4-8 常用参数说明

参数	参数说明	举例
-U	连接数据库的用户名，需要是 GBase 8c 管理员用户。	-U gbase

参数	参数说明	举例
-W	指定用户连接的密码。 如果主机的认证策略是 trust，则不会对数据库管理员进行密码验证，即无需输入-W 选项； 如果没有-W 选项，并且不是数据库管理员，会提示用户输入密码。	-W abcd@123
-f	将导出文件发送至指定目录文件夹。如果这里省略，则使用标准输出。	-f /home/gbase/backup/MPPDB_tablespace.sql
-p	指定服务器所侦听的 TCP 端口或本地 Unix 域套接字后缀，以确保连接。	-p 15400
-t	或者--tablespaces-only，只转储表空间，不转储数据库或角色。	-t

其他参数说明请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 \> gs_dumpall”章节。

示例

示例一：执行 gs_dumpall，导出所有数据库的公共全局表空间信息和用户信息（gbase 用户为管理员用户），导出文件为文本格式。

```
gs_dumpall -U gbase -f /home/gbase/backup/MPPDB_globals.sql -p 15400 -g
Password:
gs_dumpall[port='15400'][2024-03-01  19:06:24]: dumpall operation successful
gs_dumpall[port='15400'][2024-03-01  19:06:24]: total time: 1150  ms
```

示例二： 执行 gs_dumpall，导出所有数据库的公共全局表空间信息（gbase 用户为管理员用户），并对导出文件进行加密，导出文件为文本格式。

```
gs_dumpall -U gbase -f /home/gbase/backup/MPPDB_tablespace.sql -p 15400 -t
--with-encryption AES128 --with-key abcdefg_?1234567
Password:
gs_dumpall[port='15400'][2024-03-01  19:00:58]: dumpall operation successful
gs_dumpall[port='15400'][2024-03-01  19:00:58]: total time: 186  ms
```

示例三：执行 gs_dumpall，导出所有数据库的公共全局用户信息（gbase 用户为管理员

用户），导出文件为文本格式。

```
gs_dumpall -U gbase -f /home/gbase/backup/MPPDB_user.sql -p 15400 -r
Password:
gs_dumpall[port='15400'][2024-03-01  19:03:18]: dumpall operation successful
gs_dumpall[port='15400'][2024-03-01  19:03:18]: total time: 162  ms
```

4.6.1.4 无权限角色导出数据

gs_dump 和 gs_dumpall 通过-U 指定执行导出的用户帐户。如果当前使用的帐户不具备导出所要求的权限时，会无法导出数据。此时，需先将具有权限的角色赋权给无权限角色，然后在导出命令中设置–role 参数来指定具备权限的角色。在执行命令后，gs_dump 和 gs_dumpall 会使用–role 参数指定的角色，完成导出动作。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用 gs_dump 导出 human_resource 数据库数据。

用户 jack 不具备导出数据库 human_resource 的权限，而角色 role1 具备该权限，要实现导出数据库 human_resource，需要将 role1 赋权给 jack，然后可以在导出命令中设置–role 角色为 role1，使用 role1 的权限，完成导出目的。导出文件格式为 tar 归档格式。

```
gs_dump -U jack -f /home/gbase/backup/MPPDB_backup.tar -p 15400 human_resource
--role role1 --rolepassword  abc@1234 -F t
Password:
```

表 4-9 常用参数说明

参数	参数说明	举例
-U	连接数据库的用户名。	-U jack
-W	指定用户连接的密码。 如果主机的认证策略是 trust，则不会对数据库管理员进行密码验证，即无需输入-W 选项。 如果没有-W 选项，并且不是数据库管理员，会提示用户输入密码。	-W abcd@123
-f	将导出文件发送至指定目录文件夹。如果这里	-f /home/gbase/backup/MPP

参数	参数说明	举例
	省略，则使用标准输出。	DB_backup.tar
-p	指定服务器所侦听的 TCP 端口或本地 Unix 域套接字后缀，以确保连接。	-p 15400
dbname	需要导出的数据库名称。	human_resource
--role	指定导出使用的角色名。选择该选项，会使导出工具连接数据库后，发起一个 SET ROLE 角色名命令。当所授权用户（由-U 指定）没有导出工具要求的权限时，该选项会起到作用，即切换到具备相应权限的角色。	-r role1
--rolepassword	指定具体角色用户的角色密码。	--rolepassword abc@1234
-F	选择导出文件格式。-F 参数值如下： p：纯文本格式 c：自定义归档 d：目录归档格式 t：tar 归档格式	-F t

其他参数说明请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_dump”章节或“服务端工具 > gs_dumpall”章节。

示例

示例一：执行 gs_dump 导出数据，用户 jack 不具备导出数据库 human_resource 的权限，而角色 role1 具备该权限，要实现导出数据库 human_resource，可以在导出命令中设置–role 角色为 role1，使用 role1 的权限，完成导出目的。导出文件格式为 tar 归档格式。

```
human_resource=# CREATE USER jack IDENTIFIED BY "1234@abc";
CREATE ROLE
human_resource=# GRANT role1 TO jack;
GRANT ROLE

gs_dump -U jack -f /home/gbase/backup/MPPDB_backup11.tar -p 15400 human_resource
--role role1 --rolepassword abc@1234 -F t
```

```
Password:
gs_dump[port='15400'][human_resource][2024-03-01 16:21:10]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 16:21:10]: total time: 4239
ms
```

示例二：执行 gs_dump 导出数据，用户 jack 不具备导出模式 public 的权限，而角色 role1 具备该权限，要实现导出模式 public，可以在导出命令中设置–role 角色为 role1，使用 role1 的权限，完成导出目的。导出文件格式为 tar 归档格式。

```
human_resource=# CREATE USER jack IDENTIFIED BY "1234@abc";
CREATE ROLE
human_resource=# GRANT role1 TO jack;
GRANT ROLE

gs_dump -U jack -f /home/gbase/backup/MPPDB_backup12.tar -p 15400 human_resource
-n public --role role1 --rolepassword abc@1234 -F t
Password:
gs_dump[port='15400'][human_resource][2024-03-01 16:21:10]: dump database
human_resource successfully
gs_dump[port='15400'][human_resource][2024-03-01 16:21:10]: total time: 3278
ms
```

示例三：执行 gs_dumpall 导出数据，用户 jack 不具备导出所有数据库的权限，而角色 role1（管理员）具备该权限，要实现导出所有数据库，可以在导出命令中设置–role 角色为 role1，使用 role1 的权限，完成导出目的。导出文件格式为文本归档格式。

```
human_resource=# CREATE USER jack IDENTIFIED BY "1234@abc";
CREATE ROLE
human_resource=# GRANT role1 TO jack;
GRANT ROLE

gs_dumpall -U jack -f /home/gbase/backup/MPPDB_backup.sql -p 15400 --role role1
--rolepassword abc@1234
Password:
gs_dumpall[port='15400'][human_resource][2024-03-01  17:26:18]: dumpall
operation successful
gs_dumpall[port='15400'][human_resource][2024-03-01  17:26:18]: total time:
6437  ms
```

4.7 导入数据

GBase 8c 数据库提供了灵活的数据入库方式：INSERT、COPY 以及 gsql 元命令\copy。

各方式具有不同的特点，具体请参见表 4-12。

表 4-10 导入方式特点说明

方式	特点
INSERT	通过 INSERT 语句插入一行或多行数据，及从指定表插入数据。
COPY	通过 COPY FROM STDIN 语句直接向 GBase 8c 数据库写入数据。 通过 JDBC 驱动的 CopyManager 接口从其他数据库向 GBase 8c 数据库写入数据时，具有业务数据无需落地成文件的优势。
gsql 工具的元命令\copy	与直接使用 SQL 语句 COPY 不同，该命令读取/写入的文件只能是 gsql 客户端所在机器上的本地文件。 说明： \COPY 只适合小批量、格式良好的数据导入，不会对非法字符做预处理，也无容错能力，无法适用于含有异常数据的场景。导入数据应优先选择 COPY。

4.7.1 通过 INSERT 语句直接写入数据

用户可以通过以下方式执行 INSERT 语句直接向 GBase 8c 数据库写入数据：

- 使用 GBase 8c 数据库提供的客户端工具向 GBase 8c 数据库写入数据。

  请参见《GBase 8c V5_5.0.0_数据库管理指南》中“向表中插入数据”章节。

- 通过 JDBC/ODBC 驱动连接数据库执行 INSERT 语句向 GBase 8c 数据库写入数据。

  详细内容请参见《GBase 8c V5_5.0.0_数据库管理指南》中“连接数据库”章节。

GBase 8c 数据库支持完整的数据库事务级别的增删改操作。INSERT 是最简单的一种数据写入方式，这种方式适合数据写入量不大，并发度不高的场景。

4.7.2 使用 COPY FROM STDIN 导入数据

关于 COPY FROM STDIN 导入数据

用户可以使用以下方式通过 COPY FROM STDIN 语句直接向 GBase 8c 写入数据。

- 通过键盘输入向 GBase 8c 数据库写入数据。详细请参见 COPY。

- 通过 JDBC 驱动的 CopyManager 接口从文件或者数据库向 GBase 8c 写入数据。此方法支持 COPY 语法中 copy option 的所有参数。

CopyManager 类简介

CopyManager 是 GBase 8c 的 JDBC 驱动中一个 API 接口类，用于批量导入数据。

CopyManager 的继承关系

CopyManager 类位于 org.postgresql.copy Package 中，继承自 java.lang.Object 类，该类的声明如下：

```
public class CopyManager
extends Object
```

构造方法

```
public CopyManager(BaseConnection connection)
throws SQLException
```

常用方法

表 4-11 CopyManager 常用方法

返回值	方法	描述	throws
CopyIn	copyIn(String sql)	——	SQLException
long	copyIn(String sql, InputStream from)	使用 COPY FROM STDIN 从 InputStream 中快速向数据库中的表导入数据。	SQLException,IOException
long	copyIn(String sql, InputStream from, int bufferSize)	使用 COPY FROM STDIN 从 InputStream 中快速向数据库中的表导入数据。	SQLException,IOException
long	copyIn(String sql, Reader from)	使用 COPY FROM STDIN 从 Reader 中快速向数据库中的表导入数据。	SQLException,IOException

long	copyIn(String sql, Reader from, int bufferSize)	使用 COPY FROM STDIN 从 Reader 中快速向数据库中的表导入数据。	SQLException,IOException
CopyOu t	copyOut(String sql)	——	SQLException
long	copyOut(String sql, OutputStream to)	将一个 COPY TO STDOUT 的结果集从数据库发送到 OutputStream 类中。	SQLException,IOException
long	copyOut(String sql, Writer to)	将一个 COPY TO STDOUT 的结果集从数据库发送到 Writer 类中。	SQLException,IOException

4.7.3 处理错误表

操作场景

当数据导入发生错误时，请根据本文指引信息进行处理。

查询错误信息

数据导入过程中发生的错误，一般分为数据格式错误和非数据格式错误。

数据格式错误

在创建外表时，通过设置参数 LOG INTO error_table_name，将数据导入过程中出现的数据格式错误信息，写入指定的错误信息表 error_table_name 中。可以通过以下 SQL 命令，查询详细错误信息。

```
postgres=# SELECT * FROM error_table_name;
```

错误信息表结构如表 4-14 所示。

表 4-12 错误信息表

列名称	类型	描述
nodeid	integer	报错节点编号。
begintime	timestamp with time	出现数据格式错误的时间。

	zone	
filename	character varying	出现数据格式错误的数据源文件名。
rownum	numeric	在数据源文件中，出现数据格式错误的行号。
rawrecord	text	在数据源文件中，出现数据格式错误的原始记录。
detail	text	详细错误信息。

- 非数据格式错误

一旦发生非数据格式错误，将导致整个数据导入失败。用户可以根据执行数据导入过程中界面提示的错误信息，用以定位问题并处理错误表。

处理数据导入错误

根据获取的错误信息，对照表 4-15 以处理数据导入错误。

表 4-13 处理数据导入错误

错误信息	原因	解决办法
missing data for column "r_reason_desc"	数据源文件中的列数比外表定义的列数少。 对于 TEXT 格式的数据源文件，由于转义字符（\）导致 delimiter（分隔符）错位或者 quote（引号字符）错位造成的错误。 示例：目标表存在 3 列字段，导入的数据如下所示。由于存在转义字符“\”，分隔符“|”被转义为第二个字段的字段值，导致第三个字段值缺失。 BE|Belgium|1	1. 由于列数少导致的报错，选择下列办法解决： ● 在数据源文件中，增加列 r_reason_desc 的字段值。 ● 在创建外表时，将参数 fill_missing_fields 设置为 on。即当导入过程中，若数据源文件中一行数据的最后一个字段缺失，则把最后一个字段的值设置为 NULL，不报错。 2. 对由于转义字符导致的错误，需检查报错的行中是否含有转义字符（\）。若存在，建议在创建外表时，将参数“noescaping”（是否不对'\'和后面的字符进行转义）设置为 true。
extra data after last	数据源文件中的列数比外表定义	● 在数据源文件中，删除多余的

expected column	的列数多。	字段值。 ● 在创建外表时，将参数 ignore_extra_data 设置为 on。即在导入过程中，若数据源文件比外表定义的列数多，则忽略行尾多出来的列。
invalid input syntax for type numeric: "a"	数据类型错误。	在数据源文件中，修改输入字段的数据类型。根据此错误信息，需将输入的数据类型修改为 numeric。
null value in column "staff_id" violates not-null constraint	非空约束。	在数据源文件中，增加非空字段信息。根据此错误信息，需增加 staff_id 列的值。
duplicate key value violates unique constraint "reg_id_pk"	唯一约束。	删除数据源文件中重复的行。通过设置关键字 DISTINCT，从 SELECT 结果集中删除重复的行，保证导入的每一行都是唯一 的。 postgres=# INSERT INTO reasons SELECT DISTINCT * FROM foreign_tpcds_reasons;
value too long for type character varying(16)	字段值长度超过限制。	在数据源文件中，修改字段值长度。根据此错误信息，需将字段值长度限制为 VARCHAR2(16)。

4.7.3.1 示例 1：通过本地文件导入导出数据

在基于，可以使用 CopyManager 接口，通过流方式，将数据库中的数据导出到本地文件或者将本地文件导入数据库中，文件格式支持 CSV、TEXT 等格式。

样例程序如下，执行时需要加载 GBase 8c 的 JDBC 驱动。

```
import java.sql.Connection; import java.sql.DriverManager; import
java.io.IOException; import java.io.FileInputStream; import
java.io.FileOutputStream; import java.sql.SQLException;
import org.postgresql.copy.CopyManager; import
org.postgresql.core.BaseConnection;
public class Copy{
public static void main(String[] args)
{
String urls = new String("jdbc:postgresql://localhost:15400/postgres");//数据
库 URL
String username =  new String("username");//用户名
String password =  new String("passwd");   //密码
String tablename = new String("migration_table"); //定义表信息
String tablename1 = new String("migration_table_1"); //定义表信息
String driver = "org.postgresql.Driver";
Connection conn = null;
try {
Class.forName(driver);
conn = DriverManager.getConnection(urls, username, password);
} catch (ClassNotFoundException e)
{ e.printStackTrace(System.out);
} catch (SQLException e)
{ e.printStackTrace(System.out);
}
// 将表 migration_table 中数据导出到本地文件 d:/data.txt
try {
copyToFile(conn, "d:/data.txt", "(SELECT * FROM migration_table)");
} catch (SQLException e) {
// TODO Auto-generated catch block
e.printStackTrace();
} catch (IOException e) {
// TODO Auto-generated catch block e.printStackTrace();
}
//将 d:/data.txt 中的数据导入到 migration_table_1 中。
try {
copyFromFile(conn, "d:/data.txt", tablename1);
} catch (SQLException e) {
// TODO Auto-generated catch block e.printStackTrace();
} catch (IOException e) {
// TODO Auto-generated catch block e.printStackTrace();
}
```

```
// 将表 migration_table_1 中的数据导出到本地文件 d:/data1.txt
try {
copyToFile(conn, "d:/data1.txt", tablename1);
} catch (SQLException e) {
// TODO Auto-generated catch block e.printStackTrace();
} catch (IOException e) {
// TODO Auto-generated catch block e.printStackTrace();
}
}
public static void copyFromFile(Connection connection, String filePath, String
tableName)
throws SQLException, IOException {
FileInputStream fileInputStream = null; try {
CopyManager copyManager = new CopyManager((BaseConnection)connection);
fileInputStream = new FileInputStream(filePath);
copyManager.copyIn("COPY " + tableName + " FROM STDIN with (" +
"DELIMITER"+"'"+ delimiter + "'" + "ENCODING " + "'" + encoding + "')",
fileInputStream);
} finally {
if (fileInputStream != null)
{ try {
fileInputStream.close();
} catch (IOException e) { e.printStackTrace();
}
}
}
}
public static void copyToFile(Connection connection, String filePath, String
tableOrQuery) throws SQLException, IOException {
FileOutputStream fileOutputStream = null;
try {
CopyManager copyManager = new CopyManager((BaseConnection)connection);
fileOutputStream = new FileOutputStream(filePath);
copyManager.copyOut("COPY " + tableOrQuery + " TO STDOUT", fileOutputStream);
} finally {
if (fileOutputStream != null) { try {
fileOutputStream.close();
} catch (IOException e)
{ e.printStackTrace();
}
}
```

```
}
}
}
```

4.7.3.2 示例 2：从 MY 迁移数据

下面示例演示如何通过 CopyManager 从 MY 向 GBase 8c 进行数据迁移的过程。

```
import java.io.StringReader; import java.sql.Connection;
import java.sql.DriverManager; import java.sql.ResultSet;
import java.sql.SQLException; import java.sql.Statement;
import org.postgresql.copy.CopyManager;
import org.postgresql.core.BaseConnection;
public class Migration{
public static void main(String[] args) {
String url = new String("jdbc:postgresql://localhost:15400/postgres"); //数据
库 URL
String user = new String("username");  //GBase 8c 数据库用户名
String pass =  new String("passwd");   //GBase 8c 数据库密码 String
tablename = new String("migration_table_1"); //定义表信息
String delimiter = new String("|"); //定义分隔符
String encoding = new String("UTF8"); //定义字符集
String driver = "org.postgresql.Driver";
StringBuffer buffer = new StringBuffer();//定义存放格式化数据的缓存
try {
//获取源数据库查询结果集 ResultSet rs = getDataSet();
//遍历结果集，逐行获取记录
//将每条记录中各字段值，按指定分隔符分割，由换行符结束，拼成一个字符串
//把拼成的字符串，添加到缓存 buffer while (rs.next()) {
buffer.append(rs.getString(1) + delimiter
+ rs.getString(2) + delimiter
+ rs.getString(3) + delimiter
+ rs.getString(4)
+ "\n");
}
rs.close();
try {
//建立目标数据库连接 Class.forName(driver);
Connection conn = DriverManager.getConnection(url, user, pass);
BaseConnection baseConn = (BaseConnection) conn;
baseConn.setAutoCommit(false);
//初始化表信息
```

```
String sql = "Copy " + tablename + " from STDIN with (DELIMITER "
+ "'" + delimiter + "'" +",'+ " ENCODING " + "'" + encoding +
"'");
//提交缓存 buffer 中的数据
CopyManager cp = new CopyManager(baseConn);
StringReader reader = new StringReader(buffer.toString());
cp.copyIn(sql, reader);
baseConn.commit(); reader.close(); baseConn.close();
} catch (ClassNotFoundException e) {
e.printStackTrace(System.out);
} catch (SQLException e) {
e.printStackTrace(System.out);
}
} catch (Exception e) {
e.printStackTrace();
}
}
//**********************************
// 从源数据库返回查询结果集
//**********************************
private static ResultSet getDataSet() { ResultSet rs = null;
try {
Class.forName("com.MY.jdbc.Driver").newInstance();
Connection conn =
DriverManager.getConnection("jdbc:MY://10.119.179.227:3306/jack?
useSSL=false&allowPublicKeyRetrieval=true", "jack", "xxxxxxxxx");
Statement stmt = conn.createStatement();
rs = stmt.executeQuery("select * from migration_table");
} catch (SQLException e) {
e.printStackTrace();
} catch (Exception e) {
e.printStackTrace();
}
return rs;
}
}
```

4.7.4 使用 gsql 元命令导入数据

gsql 工具提供了元命令\copy 进行数据导入。

\COPY 命令

\copy 命令格式以及说明参见下表。

表 4-14 \copy 元命令说明

语法	说明
\copy { table [(column_list)] | (query) } { from | to } { filename | stdin | stdout | pstdin | pstdout }[with] [binary] [delimiter [as] 'character'] [null [as] 'string'] [csv [header] [quote [as] 'character'] [escape [as] 'character'] [force quote column_list | *] [force not null column_list]]	在 gsql 客户端登录数据库成功后，可以使用该命令进行数据的导入/导出操作。但是与 SQL 的 COPY 命令不同，该命令读取/ 写入的文件是本地文件，而非数据库服务器端文件；所以，要操作的文件的可访问性、权限等，都是受限于本地用户的权限。 说明：\COPY 只适合小批量、格式良好的数据导入场景，不会对非法字符做预处理，也无容错能力，无法适用于含有异常数据的场景。导入数据应优先选择 COPY。

参数说明

- table

表的名称（可以有模式修饰）。取值范围：已存在的表名。

- column_list

可选的待拷贝字段列表。

取值范围：任意字段。如果没有声明字段列表，将使用所有字段。

- query

其结果将被拷贝。

取值范围：一个必须用圆括弧包围的 SELECT 或 VALUES 命令。

- filename

文件的绝对路径。执行 copy 命令的用户必须有此路径的写权限。

- stdin

声明输入是来自标准输入。

- stdout

声明输出打印到标准输出。

- pstdin

声明输入是来自 gsql 的标准输入。

- pstout

声明输出打印到 gsql 的标准输出。

- binary

使用二进制格式存储和读取，而不是以文本的方式。在二进制模式下，不能声明 DELIMITER，NULL，CSV 选项。指定 binary 类型后，不能再通过 option 或 copy_option 指定 CSV、FIXED、TEXT 等类型。

- delimiter [as] 'character'

指定数据文件行数据的字段分隔符。

说明

✓ 分隔符不能是\r 和\n。

✓ 分隔符不能和 null 参数相同，CSV 格式数据的分隔符不能和 quote 参数相同。

✓ TEXT 格式数据的分隔符不能包含： \.abcdefghijklmnopqrstuvwxyz0123456789。

✓ 数据文件中单行数据长度需<1GB，如果分隔符较长且数据列较多的情况下，会影响导出有效数据的长度。

✓ 分隔符推荐使用多字符和不可见字符。多字符例如'$^&'；不可见字符例如 0x07，0x08，0x1b 等。

取值范围：支持多字符分隔符，但分隔符不能超过 10 个字节。默认值：

- TEXT 格式的默认分隔符是水平制表符（tab）。
- CSV 格式的默认分隔符为“,”。
- FIXED 格式没有分隔符。

- null [as] 'string'

用来指定数据文件中空值的表示。取值范围：

- null 值不能是\r 和\n，最大为 100 个字符。
- null 值不能和分隔符、quote 参数相同。默认值：
- CSV 格式下默认值是一个没有引号的空字符串。

- 在 TEXT 格式下默认值是\N。

- header

指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。

header 只能用于 CSV，FIXED 格式的文件中。

在导入数据时，如果 header 选项为 on，则数据文本第一行会被识别为标题行，会忽略此行。如果 header 为 off，而数据文件中第一行会被识别为数据。

在导出数据时，如果 header 选项为 on，则需要指定 fileheader。fileheader 是指定导出数据包含标题行的定义文件。如果 header 为 off，则导出数据文件不包含标题行。

取值范围：true/on，false/off。默认值：false

- quote [as] 'character'

CSV 格式文件下的引号字符。默认值：双引号。

说明

- ✓ quote 参数不能和分隔符、null 参数相同。
- ✓ quote 参数只能是单字节的字符。
- ✓ 推荐不可见字符作为 quote，例如 0x07，0x08，0x1b 等。

- escape [as] 'character'

CSV 格式下，用来指定逃逸字符，逃逸字符只能指定为单字节字符。默认值：双引号。当与 quote 值相同时，会被替换为'\0'。

- force quote column_list | *

在 CSV COPY TO 模式下，强制在每个声明的字段周围对所有非 NULL 值都使用引号包围。NULL 输出不会被引号包围。

取值范围：已存在的字段。

- force not null column_list

在 CSV COPY FROM 模式下，指定的字段输入不能为空。取值范围：已存在的字段。

任务示例

(1) 创建目标表 a。

```
postgres=# CREATE TABLE a(a int);
```

(2) 导入数据。

① 从 stdin 拷贝数据到目标表 a。

```
postgres=# \copy a from stdin;
```

出现>>符号提示时，输入数据，输入\.时结束

```
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 1
>> 2
>> \.
```

查询导入目标表 a 的数据。

```
postgres=# SELECT * FROM a;
 a
---
1
2
(2 rows)
```

② 从本地文件拷贝数据到目标表 a。假设存在本地文件/home/gbase/2.csv。

- 分隔符为‘，’。
- 在导入过程中，若数据源文件比外表定义的列数多，则忽略行尾多出来的列。

```
postgres=# \copy a FROM '/home/gbase/2.csv' WITH (delimiter',',IGNORE_EXTRA_DATA
'on');
```

4.7.5 使用 gs_restore 命令导入数据

操作场景

gs_restore 是 GBase 8c 数据库提供的与 gs_dump 配套的导入工具。通过该工具，可将 gs_dump 导出的文件导入至数据库。gs_restore 支持导入的文件格式包含自定义归档格式、目录归档格式和 tar 归档格式。

gs_restore 具备如下两种功能。

- 导入至数据库

如果指定了数据库，则数据将被导入到指定的数据库中。其中，并行导入必须指定连接数据库的密码。导入时生成列会自动更新，并像普通列一样保存。

- 导入至脚本文件

如果未指定导入数据库，则创建包含重建数据库所需的 SQL 语句脚本，并将其写入至文件或者标准输出。该脚本文件等效于 gs_dump 导出的纯文本格式文件。

gs_restore 工具在导入时，允许用户选择需要导入的内容，并支持在数据导入前对等待导入的内容进行排序。

操作步骤

说明

gs_restore 默认是以追加的方式进行数据导入。为避免多次导入造成数据异常，在进行导入时， 建议选择使用"-c" 和"-e"参数。"-c"表示在重新创建数据库对象前，清理（删除）已存在于将要还原的数据库中的数据库对象；"-e"表示当发送 SQL 语句到数据库时如果出现错误请退出，默认状态下会继续，且在导入后会显示一系列错误信息。

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用 gs_restore 命令，从 postgres 整个数据库内容的导出文件中，将数据库的所有对象的定义导入到 backupdb。

```
gs_restore -U jack /home/gbase/backup/MPPDB_backup.tar -p 15400 -d backupdb -s -e -c
Password:
```

表 4-15 常用参数说明

参数	参数说明	举例
-U	连接数据库的用户名。	-U jack
-W	指定用户连接的密码。 如果主机的认证策略是 trust，则不会对数据库管理员进行密码验证，即无需输入-W 选项； 如果没有-W 选项，并且不是数据库管理员，会提示用户输入密码。	-W abcd@123
-d	连接数据库 dbname，并直接将数据导入到该数据库中。	-d backupdb
-p	指定服务器所侦听的 TCP 端口或本地 Unix 域套接字后缀，以确保连接。	-p 15400
-e	当发送 SQL 语句到数据库时如果出现错误，则退出。默认状态下会忽略错误任务并继续执行导入，且在导入后	——

	会显示一系列错误信息。	
-c	在重新创建数据库对象前，清理（删除）已存在于将要导入的数据库中的数据库对象。	——
-s	只导入模式定义，不导入数据。当前的序列值也不会被导入。	——

其他参数说明请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中“服务端工具 > gs_restore”章节。

----结束

示例

示例一：执行 gs_restore，导入指定 MPPDB_backup.dmp 文件（自定义归档格式）中 postgres 数据库的数据和对象定义。

```
gs_restore backup/MPPDB_backup.dmp -p 15400 -d backupdb
Password:
gs_restore[2024-03-01 19:16:26]: restore operation successful
gs_restore: total time: 13053 ms
```

示例二：执行 gs_restore，导入指定 MPPDB_backup.tar 文件（tar 归档格式）中 postgres 数据库的数据和对象定义。

```
gs_restore backup/MPPDB_backup.tar -p 15400 -d backupdb
gs_restore[2024-03-01 19:21:32]: restore operation successful gs_restore[2024-03-01 19:21:32]:
total time: 21203 ms
```

示例三：执行 gs_restore，导入指定 MPPDB_backup 目录文件（目录归档格式）中 postgres 数据库的数据和对象定义。

```
gs_restore backup/MPPDB_backup -p 15400 -d backupdb
gs_restore[2024-03-01 19:26:46]: restore operation successful
gs_restore[2024-03-01 19:26:46]: total time: 21003 ms
```

示例四：执行 gs_restore，将 postgres 数据库的所有对象的定义导入至 backupdb 数据库。导入前，数据库存在完整的定义和数据，导入后，backupdb 数据库只存在所有对象定义，表没有数据。

```
gs_restore /home/gbase/backup/MPPDB_backup.tar -p 15400 -d backupdb -s -e -c
Password:
gs_restore[2024-03-01 19:46:27]: restore operation successful
gs_restore[2024-03-01 19:46:27]: total time: 32993 ms
```

示例五：执行 gs_restore，导入 MPPDB_backup.dmp 文件中 PUBLIC 模式的所有定义和数据。在导入时会先删除已经存在的对象，如果原对象存在跨模式的依赖则需手工强制干预。

```
gs_restore backup/MPPDB_backup.dmp -p 15400 -d backupdb -e -c -n PUBLIC
gs_restore: [archiver (db)] Error while PROCESSING TOC:
gs_restore: [archiver (db)] Error from TOC entry 313; 1259 337399 TABLE table1 gaussdba
gs_restore: [archiver (db)] could not execute query:
ERROR: cannot drop table table1 because other objects depend on it
DETAIL: view t1.v1 depends on table table1
HINT: Use DROP ... CASCADE to drop the dependent objects too.
Command was: DROP TABLE public.table1;
```

手工删除依赖，导入完成后再重新创建。

```
gs_restore backup/MPPDB_backup.dmp -p 15400 -d backupdb -e -c -n PUBLIC
gs_restore[2024-03-01 19:52:26]: restore operation successful
gs_restore[2024-03-01 19:52:26]: total time: 2203 ms
```

示例六：执行 gs_restore，导入 MPPDB_backup.dmp 文件中 hr 模式下表 hr.staffs 的定义。在导入之前，hr.staffs 表不存在，需要确保存在 hr 的 schema。

```
gs_restore backup/MPPDB_backup.dmp -p 15400 -d backupdb -e -c -s -n hr -t staffs
gs_restore[2024-03-01 19:56:29]: restore operation successful
gs_restore[2024-03-01 19:56:29]: total time: 21000 ms
```

示例七：执行 gs_restore，导入 MPPDB_backup.dmp 文件中 hr 模式下表 hr.staffs 的数据。在导入之前，hr.staffs 表不存在数据，需要确保存在 hr 的 schema。

```
gs_restore backup/MPPDB_backup.dmp -p 15400 -d backupdb -e -a -n hr -t staffs
gs_restore[2024-03-01 20:12:32]: restore operation successful
gs_restore[2024-03-01 20:12:32]: total time: 20203 ms
```

示例八：执行 gs_restore，导入指定表 hr.staffs 的定义。在导入之前，hr.staffs 表的数据是存在的。

```
human_resource=# select * from hr.staffs;
staff_id | first_name   |    last_name   |    email    |    phone_number    |    hire_date |
employment_id | salary | commission_pct | manager_id | section_id
----------+-------------+-------------+----------+--------------------+---------------------+---------------+-
---------+----------------+------------+----
200 | Jennifer | Whalen | JWHALEN | 515.123.4444 | 1987-09-17 00:00:00 | AD_ASST |
4400.00 |      |101 |10
201 | Michael |      | Hartstein | MHARTSTE | 515.123.5555 | 1996-02-17 00:00:00 | MK_MAN
13000.00 | |    100 |20
gsql -d human_resource -p 15400
```

```
gsql ((single_node GBase8cV5 S5.0.0B19 build b0e57b26) compiled at 2024-01-14 16:38:29
commit 0 last mr 443 )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.
human_resource=# drop table hr.staffs CASCADE; NOTICE: drop cascades to view
hr.staff_details_view DROP TABLE
gs_restore /home/gbase/backup/MPPDB_backup.tar -p 15400 -d human_resource -n hr -t staffs
-s -e Password:
restore operation successful
total time: 904 ms
```

示例九：执行 gs_restore，导入 staffs 和 areas 两个指定表的定义和数据。在导入之前，staffs 和 areas 表不存在。

```
human_resource=# \d
List of relations Schema
Name    | Type | Owner |     Storage
--------+-------------------+-------+----------+--------------------------------
hr    | employment_history | table | gbase | {orientation=row,compression=no}
hr | employments   | table | gbase | {orientation=row,compression=no}
hr | places | table | gbase | {orientation=row,compression=no}
hr | sections | table | gbase | {orientation=row,compression=no}
hr | states | table | gbase | {orientation=row,compression=no}
(5 rows)
gs_restore /home/gbase/backup/MPPDB_backup.tar -p 15400 -d human_resource -n hr -t staffs
-n hr -t areas
Password:
restore operation successful total time: 724 ms
human_resource=# \d
List of relations
Schema |       Name       | Type    | Owner    | Storage
--------+-------------------+-------+----------+--------------------------------
hr    | areas | table | gbase | {orientation=row,compression=no}
hr    | employment_history | table | gbase | {orientation=row,compression=no}
hr    | employments       | table | gbase | {orientation=row,compression=no}
hr | places | table | gbase | {orientation=row,compression=no}
hr | sections | table | gbase | {orientation=row,compression=no}
hr | staffs | table | gbase | {orientation=row,compression=no}
hr | states | table | gbase | {orientation=row,compression=no}
(7 rows)
human_resource=# select * from hr.areas; area_id |    area_name
---------+-----------------------
```

```
4 | Middle East and Africa 1 | Europe
| Americas
| Asia
(4 rows)
```

示例十：执行 gs_restore，导入 hr 的模式，包含模式下的所有对象定义和数据。

```
gs_restore /home/gbase/backup/MPPDB_backup1.dmp -p 15400 -d backupdb -n hr -e
Password:
restore operation successful total time: 702 ms
```

示例十一：执行 gs_restore，同时导入 hr 和 hr1 两个模式，仅导入模式下的所有对象定义。

```
gs_restore /home/gbase/backup/MPPDB_backup2.dmp -p 15400 -d backupdb -n hr -n hr1 -s
Password:
restore operation successful total time: 665 ms
```

示例十二：执行 gs_restore，将 human_resource 数据库导出文件导入至 backupdb 数据库中。

```
postgres=# create database backupdb;
CREATE DATABASE

gs_restore /home/gbase/backup/MPPDB_backup.tar -p 15400 -d backupdb
restore operation successful total time: 23472 ms
gsql -d backupdb -p 15400 -r
gsql ((single_node GBase8cV5 S5.0.0B19 build b0e57b26) compiled at 2024-01-14 16:38:29
commit 0 last mr 443 )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

postgres=#
postgres=# select * from hr.areas;
area_id |area_name
---------+-------------------------
4 | Middle East and Africa 1 | Europe
| Americas
| Asia
(4 rows)
```

示例十三：用户 user1 不具备将导出文件中数据导入至数据库 backupdb 的权限，而角色 role1 具备该权限，要实现将文件数据导入数据库 backupdb，可以在导出命令中设置--role 角色为 role1，使用 role1 的权限，完成导出目的。

```
human_resource=# CREATE USER user1 IDENTIFIED BY "1234@abc"; CREATE ROLE
role1 with SYSADMIN IDENTIFIED BY "abc@1234";
gs_restore -U user1 /home/gbase/backup/MPPDB_backup.tar -p 15400 -d backupdb --role role1
-- rolepassword abc@1234
Password:
restore operation successful total time: 554 ms
gsql -d backupdb -p 15400 -r
gsql ((single_node GBase8cV5 S5.0.0B19 build b0e57b26) compiled at 2024-01-14 16:38:29
commit 0 last mr 443 )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.
postgres=# select * from hr.areas;
 area_id |      area_name
---------+------------------------
4 | Middle East and Africa
1 | Europe
| Americas
| Asia
(4 rows)
```

4.7.6 更新表中数据

4.7.6.1 使用 DML 命令更新表

GBase 8c 支持标准 DML 命令（数据库操作语言），对表进行更新。

操作步骤

假设存在表 customer_t，表结构如下：

```
postgres=# CREATE TABLE customer_t
( c_customer_sk    integer,
c_customer_id      char(5),
c_first_name char(6),
c_last_name  char(8)
) ;
```

可以使用如下 DML 命令对表进行数据更新。

- 使用 INSERT 向表中插入数据。
  - 向表 customer_t 中插入一行。

```
postgres=# INSERT INTO customer_t (c_customer_sk, c_customer_id,
c_first_name,c_last_name) VALUES (3769, 5, 'Grace','White');
```

- 向表 customer_t 中插入多行数据。

```
postgres=# INSERT INTO customer_t (c_customer_sk, c_customer_id,
c_first_name,c_last_name) VALUES
(6885, 1, 'Joes', 'Hunter'),
(4321, 2, 'Lily','Carter'),
(9527, 3, 'James', 'Cook'),
(9500, 4, 'Lucy', 'Baker');
```

更多关于 INSERT 的使用方法，参见《GBase 8c V5_5.0.0_数据库管理指南》中“向表中插入数据”。

- 使用 UPDATE 更新表中数据。修改字段 c_customer_id 值为 0。

```
postgres=# UPDATE customer_t SET c_customer_id = 0;
```

更多关于 UPDATE 的使用方法，请参见《GBase 8c V5_5.0.0_SQL 参考手册》UPDATE。

- 使用 DELETE 删除表中的行。

可以使用 WHERE 子句指定需要删除的行，若不指定即删除表中所有的行，只保留数据结构。

```
postgres=# DELETE FROM customer_t WHERE c_last_name = 'Baker';
```

更多关于 DELETE 的使用方法，请参见《GBase 8c V5_5.0.0_SQL 参考手册》DELETE。

- 使用 TRUNCATE 命令快速从表中删除所有的行。

```
postgres=# TRUNCATE TABLE customer_t;
```

更多关于 TRUNCATE 的使用方法，请参见《GBase 8c V5_5.0.0_SQL 参考手册》TRUNCATE。

删除表时，使用 DELETE 语句每次删除一行数据，仅删除数据而不释放存储空间；使用 TRUNCATE 语句则是通过释放表存储的数据页来删除数据，删除数据且释放存储空间。使用 TRUNCATE 语句进行表删除速度更快。

4.7.6.2 使用合并方式更新和插入数据

在需要将数据批量添加至现有表的场景下，GBase 8c 数据库提供了 MERGE INTO 语句，通过表合并的方式，高效地将新数据添加到现有表。

MERGE INTO 语句可将目标表和源表中的数据，通过关联条件进行匹配。若关联条件

匹配，则对目标表进行更新操作（UPDATE）；若关联条件不匹配，则对目标表执行插入操作（INSERT）。这种方法能够将两个表合并执行 UPDATE 和 INSERT 操作，避免多次执行。

前提条件

用户如需进行 MERGE INTO 操作，需要同时拥有目标表的 UPDATE 和 INSERT 权限，以及源表的 SELECT 权限。

操作步骤

(1) 创建源表 products，并插入数据。

```
postgres=# CREATE TABLE products ( product_id INTEGER,product_name VARCHAR2(60),
category VARCHAR2(60));
postgres=# INSERT INTO products VALUES (1502, 'olympus camera', 'electrncs'), (1601,
'lamaze', 'toys'),(1666, 'harry potter', 'toys'), (1700, 'wait interface', 'books');
```

(2) 创建目标表 newproducts，并插入数据。

```
postgres=# CREATE TABLE newproducts ( product_id INTEGER, product_name
VARCHAR2(60), category VARCHAR2(60));
postgres=# INSERT INTO newproducts VALUES (1501, 'vivitar 35mm', 'electrncs'), (1502,
'olympus ', 'electrncs'),(1600, 'play gym', 'toys'),(1601, 'lamaze', 'toys'),(1666, 'harry potter',
'dvd');
```

(3) 使用 MERGE INTO 语句将源表 products 的数据合并至目标表 newproducts。

```
postgres=# MERGE INTO newproducts np
USING products p
ON (np.product_id = p.product_id )
WHEN MATCHED THEN
UPDATE SET np.product_name = p.product_name, np.category = p.category
WHEN NOT MATCHED THEN
INSERT VALUES (p.product_id, p.product_name, p.category) ;
```

上述语句中使用的参数说明，请见表 4-16。更多信息，请参见《GBase 8c V5_5.0.0_SQL 参考手册》 MERGE INTO。

表 4-16 MERGE INTO 语句参数说明

参数	说明	举例
INTO 子句	指定需要更新或插入数据的目标表。 目标表支持指定别名。	取值 ： newproducts np 说明：名为 newproducts，别名为 np 的目标表。

USING 子句	指定源表。源表支持指定别名。	取值：products p 说明：名为 products，别名为 p 的源表。
ON 子句	指定目标表和源表的关联条件。 关联条件中的字段不支持更新。	取值：np.product_id = p.product_id 说明：指定的关联条件为，目标表 newproducts 的 product_id 字段和源表 products 的 product_id 字段相等。
WHEN MATCHED 子句	当源表和目标表中数据针对关联条件可以匹配上时，选择 WHEN MATCHED 子句进行 UPDATE 操作。 仅支持指定一个 WHEN MATCHED 子句。 WHEN MATCHED 子句可缺省，缺省时，对于满足 ON 子句条件的 行，不进行任何操作。	取值：WHEN MATCHED THEN UPDATE SET np.product_name = p.product_name, np.category = p.category 说明：当满足 ON 子句条件时，将目标表 newproducts 的 product_name、category 字段的值替换为源表 products 相对应字段的值。
WHEN NOT MATCHED 子句	当源表和目标表中数据针对关联条件无法匹配时，选择 WHEN NOT MATCHED 子句进行 INSERT 操作。 仅支持指定一个 WHEN NOT MATCHED 子句。 WHEN NOT MATCHED 子句可缺省。 不支持 INSERT 子句中包含多个 VALUES。 WHEN MATCHED 和 WHEN NOT MATCHED 子句顺序可以交换，可以缺省其中一个，但不能同时缺省。	取值：WHEN NOT MATCHED THEN INSERT VALUES (p.product_id, p.product_name, p.category) 说明：将源表 products 中，不满足 ON 子句条件的行插入目标表 newproducts。

(4) 查询合并后的目标表 newproducts。

```
postgres=# SELECT * FROM newproducts;
```

返回信息如下：

```
product_id | product_name | category
```

```
-----------+---------------+-----------
1501 | vivitar 35mm | electrncs
1502 | olympus camera | electrncs
1666 | harry potter | toys
1600    | play gym        | toys
1601 | lamaze       | toys
1700 | wait interface | books
(6 rows)
```

4.7.7 深层复制

数据导入后，如果需要修改表的分区键、或者将行存表改列存、添加 PCK（Partial Cluster Key）约束等场景下，可以使用深层复制的方式对表进行调整。深层复制是指重新创建表，然后使用批量插入填充表的过程。

GBase 8c 数据库提供了三种深层复制的方式。

4.7.7.1 使用 CREATE TABLE 执行深层复制

该方法使用 CREATE TABLE 语句创建原始表的副本，将原始表的数据填充至副本并重命名副本，以完成原始表的复制。

在创建新表时，可以指定表以及列属性，比如主键。

操作步骤

执行以下步骤对表 customer_t 进行深层复制。

(1) 使用 CREATE TABLE 语句创建表 customer_t 的副本 customer_t_copy。

```
postgres=# CREATE TABLE customer_t_copy
( c_customer_sk    integer,
c_customer_id      char(5),
c_first_name char(6),
c_last_name  char(8)
) ;
```

(2) 使用 INSERT INTO…SELECT 语句向副本填充原始表中的数据。

```
postgres=# INSERT INTO customer_t_copy (SELECT * FROM customer_t);
```

(3) 删除原始表。

```
postgres=# DROP TABLE customer_t;
```

(4) 使用 ALTER TABLE 语句将副本重命名为原始表名称。

```
postgres=# ALTER TABLE customer_t_copy RENAME TO customer_t;
```

4.7.7.2 使用 CREATE TABLE LIKE 执行深层复制

该方法使用 CREATE TABLE LIKE 语句创建原始表的副本，将原始表的数据填充至副本并重命名副本，完成原始表的复制。该方法不继承父表的主键属性，可以使用 ALTER TABLE 语句来添加主键属性。

操作步骤

(1) 使用 CREATE TABLE LIKE 语句创建表 customer_t 的副本 customer_t_copy。

```
postgres=# CREATE TABLE customer_t_copy (LIKE customer_t);
```

(2) 使用 INSERT INTO…SELECT 语句向副本填充原始表中的数据。

```
postgres=# INSERT INTO customer_t_copy (SELECT * FROM customer_t);
```

(3) 删除原始表。

```
postgres=# DROP TABLE customer_t;
```

(4) 使用 ALTER TABLE 语句将副本重命名为原始表名称。

```
postgres=# ALTER TABLE customer_t_copy RENAME TO customer_t;
```

4.7.7.3 通过创建临时表并截断原始表来执行深层复制

该方法使用 CREATE TEMP TABLE ... AS 语句创建原始表的临时表，然后截断原始表并从临时表填充原始表，以完成原始表的深层复制。

在新建表需要保留父表的主键属性，或如果父表具有依赖项的情况下，建议使用此方法。

操作步骤

(1) 使用 CREATE TEMP TABLE AS 语句创建表 customer_t 的临时表副本 customer_t_temp。

```
postgres=# CREATE TEMP TABLE customer_t_temp AS SELECT * FROM customer_t;
```

说明

与使用永久表相比，使用临时表可以提高性能，但存在丢失数据的风险。临时表只在当前会话可见，本会话结束后将自动删除。如果数据丢失是不可接受的，请使用永久表。

(2) 截断当前表 customer_t。

```
postgres=# TRUNCATE customer_t;
```

(3) 使用 INSERT INTO…SELECT 语句从副本中向原始表中填充数据。

```
postgres=# INSERT INTO customer_t (SELECT * FROM customer_t_temp);
```

(4) 删除临时表副本 customer_t_temp。

```
postgres=# DROP TABLE customer_t_temp;
```

4.7.8 分析表

执行计划生成器需要使用表的统计信息，以生成最有效的查询执行计划，提高查询性能。因此数据导入完成后，建议执行 ANALYZE 语句生成最新的表统计信息。统计结果存储在系统表 PG_STATISTIC 中。

分析表

ANALYZE 支持的表类型有行/列存表。ANALYZE 同时也支持对本地表的指定列进行信息统计。下面以表的 ANALYZE 为例，更多关于 ANALYZE 的信息，请参见《GBase 8c V5_5.0.0_SQL 参考手册》 ANALYZE | ANALYSE。

(1) 更新表统计信息。

以表 product_info 为例，ANALYZE 命令如下：

```
postgres=# ANALYZE product_info;
ANALYZE
```

表自动分析

GBase 8c 提供 GUC 参数 autovacuum，用于控制数据库自动清理功能的启动。

autovacuum 设置为 on 时，系统定时启动 autovacuum 线程来进行表自动分析，如果表中数据量发生较大变化达到阈值时，会触发表自动分析，即 autoanalyze。

- 对于空表而言，当表中插入数据的行数大于 50 时，会触发表自动进行 ANALYZE。
- 对于表中已有数据的情况，阈值设定为 50+10%*reltuples，其中 reltuples 是表的总行数。

autovacuum 自动清理功能的生效还依赖于下面两个 GUC 参数：

- track_counts 参数需要设置为 on，表示开启收集收据库统计数据功能。
- autovacuum_max_workers 参数需要大于 0，该参数表示能同时运行的自动清理线程的最大数量。

须知

➢ autoanalyze 只支持默认采样方式，不支持百分比采样方式。

- 多列统计信息仅支持百分比采样，因此 autoanalyze 不收集多列统计信息。
- autoanalyze 支持行存表和列存表，不支持外表、临时表、unlogged 表和 toast 表。

4.7.9 对表执行 VACUUM

如果导入过程中，进行了大量的更新或删除行时，应运行 VACUUM FULL 命令，然后运行 ANALYZE 命令。大量的更新和删除操作，会产生大量的磁盘页面碎片，从而逐渐降低查询的效率。VACUUM FULL 可以将磁盘页面碎片恢复并交还操作系统。

(1) 对表执行 VACUUM FULL。

以表 product_info 为例，VACUUM FULL 命令如下：

```
postgres=# VACUUM FULL product_info
VACUUM
```

4.7.10管理并发写入操作

事务隔离说明

GBase 8c 数据库是基于 MVCC（多版本并发控制），并结合两阶段锁的方式进行事务管理的。特点是读写之间不阻塞。SELECT 是纯读操作，UPDATE 和 DELETE 是读写操作。

- 读写操作和纯读操作之间并不会发生冲突，读写操作之间也不会发生冲突。每个并发事务在事务开始时创建事务快照，并发事务之间不能检测到对方的更改。
  - 读已提交隔离级别中，如果事务 T1 提交后，事务 T2 就可以看到事务 T1 更改的结果。
  - 可重复读级别中，如果事务 T1 提交事务前事务 T2 开始执行，则事务 T1 提交后，事务 T2 依旧看不到事务 T1 更改的结果，保证了一个事务开始后，查询的结果前后一致，不受其他事务的影响。
- 读写操作，支持的是行级锁，不同的事务可以并发更新同一个表，只有更新同一行时才需等待，后发生的事务会等待先发生的事务提交后，再执行更新操作。
  - READ COMMITED：读已提交隔离级别，事务只能读到已提交的数据而不会读到未提交的数据，这是缺省值。
  - REPEATABLE READ：事务只能读到事务开始之前已提交的数据，不能读到未提交的数据以及事务执行期间其它并发事务提交的修改。

4.7.10.1 事务隔离说明

GBase 8c 基于 MVCC（多版本并发控制）并结合两阶段锁的方式进行事务管理，其特点是读写之间不阻塞。SELECT 是纯读操作，UPDATE 和 DELETE 是读写操作。

- 读写操作和纯读操作之间并不会发生冲突，读写操作之间也不会发生冲突。每个并发事务在事务开始时创建事务快照，并发事务之间不能检测到对方的更改。
  - 读已提交隔离级别中，如果事务 T1 提交后，事务 T2 就可以看到事务 T1 更改的结果。
  - 可重复读级别中，如果事务 T1 提交事务前事务 T2 开始执行，则事务 T1 提交后，事务 T2 依旧看不到事务 T1 更改的结果，保证了一个事务开始后，查询的结果前后一致，不受其他事务的影响。
- 读写操作，支持的是行级锁，不同的事务可以并发更新同一个表，只有更新同一行时才需等待，后发生的事务会等待先发生的事务提交后，再执行更新操作。
  - READ COMMITED：读已提交隔离级别，事务只能读到已提交的数据而不会读到未提交的数据，这是缺省值。
  - REPEATABLE READ：事务只能读到事务开始之前已提交的数据，不能读到未提交的数据以及事务执行期间其它并发事务提交的修改。

4.7.10.2 写入和读写操作

关于写入和读写操作的命令：

- INSERT，可向表中插入一行或多行数据。
- UPDATE，可修改表中现有数据。
- DELETE，可删除表中现有数据。
- COPY，导入数据。

INSERT 和 COPY 是纯写入的操作。并发写入操作，需要等待，对同一个表的操作，当事务 T1 的 INSERT 或 COPY 未解除锁定时，事务 T2 的 INSERT 或 COPY 需等待，事务 T1 解除锁定时，事务 T2 正常继续。

UPDATE 和 DELETE 是读写操作（先查询出要操作的行）。UPDATE 和 DELETE 执行前需要先查询数据，由于并发事务彼此不可见，所以 UPDATE 和 DELETE 操作是读取事务

发生前提交的数据的快照。写入操作，是行级锁，当事务 T1 和事务 T2 并发更新同一行 时，后发生的事务 T2 会等待，根据设置的等待时长，若超时事务 T1 未提交则事务 T2 执行失败；当事务 T1 和事务 T2 并发更新的行不同时，事务 T1 和事务 2 都会执行成功。

4.7.10.3 并发写入事务的潜在死锁情况

只要事务涉及多个表的或者同一个表相同行的更新时，同时运行的事务就可能在同时尝试写入时变为死锁状态。事务会在提交或回滚时一次性解除其所有锁定，而不会逐一放弃锁定。 例如，假设事务 T1 和 T2 在大致相同的时间开始：

- 如果 T1 开始对表 A 进行写入且 T2 开始对表 B 进行写入，则两个事务均可继续而不会发生冲突；但是，如果 T1 完成了对表 A 的写入操作并需要开始对表 B 进行写入，此时操作的行数正好与 T2 一致，它将无法继续，因为 T2 仍保持对表 B 对应行的锁定，此时 T2 开始更新表 A 中与 T1 相同的行数，此时也将无法继续，产生死锁，在锁等待超时内，前面事务提交释放锁，后面的事务可以继续执行更新，等待时间超时后，事务抛错，有一个事务退出。
- 如果 T1，T2 都对表 A 进行写入，此时 T1 更新 1-5 行的数据，T2 更新 6-10 行的数据，两个事务不会发生冲突，但是，如果 T1 完成后开始对表 A 的 6-10 行数据进行更新，T2 完成后开始更新 1-5 行的数据，此时两个事务无法继续，在锁等待超时内， 前面事务提交释放锁，后面的事务可以继续执行更新，等待时间超时后，事务抛错，有一个事务退出。

4.7.10.4 并发写入示例

本章节以表 test 为例，分别介绍相同表的 INSERT 和 DELETE 并发，相同表的并发 INSERT，相同表的并发 UPDATE，以及数据导入和查询的并发的执行详情。

```
CREATE TABLE test(id int, name char(50), address    varchar(255));
```

4.7.10.4.1 相同表的 INSERT 和 DELETE 并发

事务 T1：

```
START TRANSACTION;
INSERT INTO test VALUES(1,'test1','test123');
COMMIT;
```

事务 T2：

```
START TRANSACTION;
DELETE test WHERE NAME='test1';
```

```
COMMIT;
```

场景 1：

开启事务 T1，不提交的同时开启事务 T2，事务 T1 执行 INSERT 完成后，执行事务 T2 的 DELETE，此时显示 DELETE 0，由于事务 T1 未提交，事务 2 看不到事务插入的数据；

场景 2：

- READ COMMITED 级别

开启事务 T1，不提交的同时开启事务 T2，事务 T1 执行 INSERT 完成后，提交事务 T1，事务 T2 再执行 DELETE 语句时，此时显示 DELETE 1，事务 T1 提交完成后，事务 T2 可以看到此条数据，可以删除成功。

- REPEATABLE READ 级别

开启事务 T1，不提交的同时开启事务 T2，事务 T1 执行 INSERT 完成后，提交事务 T1，事务 T2 再执行 DELETE 语句时，此时显示 DELETE 0，事务 T1 提交完成后，事务 T2 依旧看不到事务 T1 的数据，一个事务中前后查询到的数据是一致的。

4.7.10.4.2 相同表的并发 INSERT

事务 T1：

```
START TRANSACTION;
INSERT INTO test VALUES(2,'test2','test123');
COMMIT;
```

事务 T2：

```
START TRANSACTION;
INSERT INTO test VALUES(3,'test3','test123');
COMMIT;
```

场景 1：

开启事务 T1，不提交的同时开启事务 T2，事务 T1 执行 INSERT 完成后，执行事务 T2 的 INSERT 语句，可以执行成功，读已提交和可重复读隔离级别下，此时在事务 T1 中执行 SELECT 语句，看不到事务 T2 中插入的数据，事务 T2 中执行查询语句看不到事务 T1 中插入的数据。

场景 2：

- READ COMMITED 级别

开启事务 T1，不提交的同时开启事务 T2，事务 T1 执行 INSERT 完成后直接提交，事

务 T2 中执行 INSERT 语句后执行查询语句，可以看到事务 T1 中插入的数据。

- REPEATABLE READ 级别

开启事务 T1，不提交的同时开启事务 T2，事务 T1 执行 INSERT 完成后直接提交，事务 T2 中执行 INSERT 语句后执行查询语句，看不到事务 T1 中插入的数据。

4.7.10.4.3 相同表的并发 UPDATE

事务 T1：

```
START TRANSACTION;
UPDATE test SET address='test1234' WHERE name='test1';
COMMIT;
```

事务 T2：

```
START TRANSACTION;
UPDATE test SET address='test1234' WHERE name='test2';
COMMIT;
```

事务 T3：

```
START TRANSACTION;
UPDATE test SET address='test1234' WHERE name='test1';
COMMIT;
```

场景 1：

开启事务 T1，不提交的同时开启事务 T2，事务 T1 开始执行 UPDATE，事务 T2 开始执行 UPDATE，事务 T1 和事务 T2 都执行成功。更新不同行时，更新操作拿的是行级锁，不会发生冲突，两个事务都可以执行成功。

场景 2：

开启事务 T1，不提交的同时开启事务 T3，事务 T1 开始执行 UPDATE，事务 T3 开始执行 UPDATE，事务 T1 执行成功，事务 T3 等待超时后会出错。更新相同行时，事务 T1 未提交时，未释放锁，导致事务 T3 执行不成功。

4.7.10.4.4 数据导入和查询的并发

事务 T1：

```
START TRANSACTION;
COPY test FROM '...';
COMMIT;
```

事务 T2：

```
START TRANSACTION;
SELECT * FROM test;
COMMIT;
```

场景 1：

开启事务 T1，不提交的同时开启事务 T2，事务 T1 开始执行 COPY，事务 T2 开始执行 SELECT，事务 T1 和事务 T2 都执行成功。事务 T2 中查询看不到事务 T1 新 COPY 进来的数据。

场景 2：

- READ COMMITED 级别

开启事务 T1，不提交的同时开启事务 T2，事务 T1 开始执行 COPY，然后提交，事务 T2 查询，可以看到事务 T1 中 COPY 的数据。

- REPEATABLE READ 级别

开启事务 T1，不提交的同时开启事务 T2，事务 T1 开始执行 COPY，然后提交，事务 T2 查询，看不到事务 T1 中 COPY 的数据。

5 实例主备切换

操作场景

GBase 8c 在运行过程中，数据库管理员可能需要手工对数据库节点做主备切换。例如发现数据库节点主备 failover 后需要恢复原有的主备角色，或怀疑硬件故障需要手动进行主备切换。级联备机不能直接转换为主机，只能先通过 switchover 或者 failover 成为备机，然后再切换为主机。

说明：

- 主备切换为维护操作，确保 GBase 8c 状态正常，所有业务结束后，再进行切换操作。
- 在开启极致 RTO 时，不支持级联备机。因为在极致 RTO 开启情况下，备机不支持连接，所以无法与级联备机同步数据。
- 级联备机切换后，主机的 synchronous_standby_names 参数不会自动调整，因此可能需要手动调整主机的 synchronous_standby_names 参数，否则有可能会导致主机的写业务阻塞。

操作步骤

(1) 以操作系统用户 gbase 登录数据库任意节点，执行如下命令，查看主备情况。

```
gs_om -t status --detail
```

(2) 以操作系统用户 gbase 登录准备切换为主节点的备节点，执行如下命令。

```
gs_ctl switchover -D /opt/database/install/data/dn
```

/opt/database/install/data/dn 为备数据库节点的数据目录。

须知

- 对于同一数据库，上一次主备切换未完成，不能执行下一次切换。当业务正在操作时，发起 switchover，可能主机的线程无法停止导致 switchover 显示超时，实际后台仍然在运行，等主机线程停止后，switchover 即可完成。比如在主机删除一个大的分区表时，可能无法响应 switchover 发起的信号。

主机故障时，可以在备机执行如下命令。

```
gs_ctl failover -D /opt/database/install/data/dn
```

(3) switchover 或 failover 成功后，执行如下命令记录当前主备机器信息。

```
gs_om -t refreshconf
```

示例

将数据库节点备实例切换为主实例。

(1) 查询数据库状态。

```
gs_om -t status --detail

[   Cluster State   ]

cluster_state   : Normal
redistributing  : No
current_az      : AZ_ALL

[  Datanode State   ]

    node             node_ip         port      instance
state
-----------------------------------------------------------------------------
---------------------
1  pekpopgsci00235  10.244.62.204    15400      6001
/opt/database/install/data/dn    P Primary Normal
```

```
2  pekpopgsci00238  10.244.61.81      15400       6002
/opt/database/install/data/dn    S Standby Normal
```

(2) 登录备节点，进行主备切换。另外，switchover 级联备机后，级联备机切换为备机，原备机降为级联备。

```
gs_ctl switchover -D /opt/database/install/data/dn/
```

(3) 保存数据库主备机器信息。

```
gs_om -t refreshconf
```

错误排查

如果 switchover 过程中出错，请根据日志文件中的信息排查错误，详见日志参考。

异常处理

异常判断标准如下：

- 业务压力下，主备实例切换时间长，这种情况不需要处理。
- 其他备机正在 build 的情况下，主机需要发送日志到备机后，才能降备，导致主备切换时间长。这种情况不需要处理，但应尽量避免 build 过程中进行主备切换。
- 切换过程中，因网络故障、磁盘满等原因造成主备实例连接断开，出现双主现象时，此时请参考如下步骤修复。

  警告

  - 出现双主状态后，请按如下步骤操作，恢复为正常的主备状态。否则可能会造成数据丢失。

  (1) 执行以下命令，查询数据库当前的实例状态。

```
gs_om -t status --detail
```

  若查询结果显示两个实例的状态都为 Primary，这种状态为异常状态。

  (2) 在降为备机的节点上，执行如下命令关闭服务。

```
gs_ctl stop -D /opt/database/install/data/dn
```

  (3) 执行以下命令，启动备节点。

```
gs_ctl start -D /opt/database/install/data/dn -M standby
```

  (4) 保存数据库主备机器信息。

```
gs_om -t refreshconf
```

(5) 查看数据库状态，确认实例状态恢复。

6 逻辑复制

6.1 逻辑解码

6.1.1 逻辑解码概述

功能描述

GBase 8c 对数据复制能力的支持情况为：

支持通过数据迁移工具定期向异构数据库（如 Oracle 等）进行数据同步，不具备实时数据复制能力。不足以支撑与异构数据库间并网运行实时数据同步的诉求。

GBase 8c 提供了逻辑解码功能，通过反解 xlog 的方式生成逻辑日志。目标数据库解析逻辑日志以实时进行数据复制。具体如图 6-1 所示。逻辑复制降低了对目标数据库的形态限制，支持异构数据库、同构异形数据库对数据的同步，支持目标库进行数据同步期间的数据可读写，数据同步时延低。

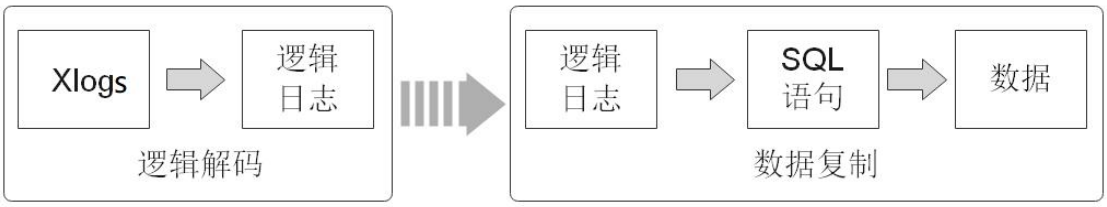

图 6-1 逻辑复制

逻辑复制由两部分组成：逻辑解码和数据复制。逻辑解码会输出以事务为单位组织的逻辑日志。业务或数据库中间件将会对逻辑日志进行解析并最终实现数据复制。GBase 8c 当前只提供逻辑解码功能，因此本章节只涉及逻辑解码的说明。

逻辑解码为逻辑复制提供事务解码的基础能力，GBase 8c 使用 SQL 函数接口进行逻辑解码。此方法调用方便，不需使用工具，对接外部工具接口也比较清晰，不需要额外适配。

由于逻辑日志是以事务为单位的，在事务提交后才能输出，且逻辑解码是由用户驱动的；因此为了防止事务开始时的 xlog 被系统回收，或所需的事务信息被 VACUUM 回收，GBase 8c 使用了逻辑复制槽，用于阻塞 xlog 的回收。

一个逻辑复制槽表示一个更改流，这些更改可以在其它数据库中以它们在原数据库上产生的顺序被重放。逻辑复制槽，由每个逻辑日志的获取者维护一个。

注意事项

- 不支持DDL语句解码，在执行特定的DDL语句(例如普通表truncate或分区表exchange)时，可能造成解码数据丢失。
- 不支持列存、数据页复制的解码。
- 不支持级联备机进行逻辑解码。
- 当执行 DDL 语句（如 alter table）后，该 DDL 语句前尚未解码的物理日志可能会丢失。
- 单条元组大小不超过 1GB，考虑解码结果可能大于插入数据，因此建议单条元组大小不超过 500MB。
- GBase 8c 支持解码的数据类型为：INTEGER、BIGINT、SMALLINT、TINYINT、SERIAL、SMALLSERIAL、BIGSERIAL、FLOAT、DOUBLE PRECISION、DATE、TIME[WITHOUT TIME ZONE]、TIMESTAMP[WITHOUT TIME ZONE]、CHAR(n)、VARCHAR(n)、TEXT。
- 如果需要 ssl 连接需要保证前置设置 GUC 参数 ssl=on。
- 逻辑复制槽名称必须小于 64 个字符，且只包含小写字母、数字或者下划线中的一种或几种。
- 当前逻辑复制不支持 MOT 特性。
- 当逻辑复制槽所在数据库被删除后，这些复制槽变为不可用状态，需要用户手动删除。
- 仅支持 utf-8 字符集。
- 对多库的解码需要分别在库内创建流复制槽并开始解码，每个库的解码都需要单独扫一遍日志。
- 不支持强起，强起后需要重新全量导出数据。
- 备机解码时，switchover 和 failover 时可能出现解码数据变多，需用户手动过滤。Qurem 协议下，switchover 和 failover 选择升主的备机，需要与当前主机日志同步。
- 不允许主备，多个备机同时使用同一个复制槽解码，否则会产生数据不一致。
- 只支持主机创建删除复制槽。
- 数据库故障重启或逻辑复制进程重启后，解码数据存在重复，用户需自己过滤。
- 计算机内核故障后，解码存在乱码，需手动或自动过滤。
- 当前备机逻辑解码，不支持开启极致 RTO。
- 请确保在创建逻辑复制槽过程中长事务未启动，启动长事务会阻塞逻辑复制槽的创建。

- 不支持 interval partition 表复制。
- 不支持全局临时表。
- 在事务中执行 DDL 语句后，该 DDL 语句与之后的语句不会被解码。
- 如需进行备机解码，需在对应主机上设置 guc 参数 enable_slot_log = on。
- 禁止在使用逻辑复制槽时在其他节点对该复制槽进行操作，删除复制槽进行的操作需在该复制槽停止解码后执行。
- 在开启逻辑复制的场景下，如需创建包含系统列的主键索引，必须将该表的 REPLICA IDENTITY 属性设置为 FULL 或是使用 USING INDEX 指定不包含系统列的、唯一的、非局部的、不可延迟的、仅包括标记为 NOT NULL 的列的索引。

性能

在 Benchmarksql-5.0 的 100warehouse 场景下，采用 pg_logical_slot_get_changes 时：

- 单次解码数据量 4K 行（对应约 5MB¯10MB 日志），解码性能 0.3MB/s¯0.5 MB/s 。
- 单次解码数据量 32K 行（对应约 40MB¯80MB 日志），解码性能 3MB/s¯5MB/s 。
- 单次解码数据量 256K 行（对应约 320MB¯640MB 日志），解码性能 3MB/s¯5MB/s 。
- 单次解码数据量再增大，解码性能无明显提升。

如果采用 pg_logical_slot_peek_changes + pg_replication_slot_advance 方式，解码性能相比采用 pg_logical_slot_get_changes 时要下降 30%¯50%。

6.1.2 使用 SQL 函数接口进行逻辑解码

GBase 8c 可以通过调用 SQL 函数，进行创建、删除、推进逻辑复制槽，获取解码后的事务日志。

前提条件

- 逻辑日志目前从主机节点中抽取，默认关闭 SSL 连接，如果进行逻辑复制，需要先配置 GUC 参数 ssl=on。

  说明：

  为避免安全风险，请保证启用 SSL 连接。

- 设置 GUC 参数 wal_level 为 logical。
- 设置 GUC 参数 max_replication_slots>每个节点所需的(物理流复制槽数+逻辑复制槽数)。

物理流复制槽提供了一种自动化的方法来确保主节点在所有备节点或从备节点收到xlog 之前，xlog 不会被移除。也就是说物理流复制槽用于支撑主备节点 HA。数据库所需要的物理流复制槽数为：备节点加从备的和与主节点之间的比例。例如，假设数据库的高可用方案为 1 主、1 备、1 从备，则所需物理流复制槽数为 2。

关于逻辑复制槽数，请按如下规则考虑。

- 一个逻辑复制槽只能解码一个 Database 的修改，如果需要解码多个 Database，则需要创建多个逻辑复制槽。
- 如果需要多路逻辑复制同步给多个目标数据库，在源端数据库需要创建多个逻辑复制槽，每个逻辑复制槽对应一条逻辑复制链路。

- 仅限初始用户和拥有 REPLICATION 权限的用户进行操作。三权分立关闭时数据库管理员可进行逻辑复制操作，三权分立开启时不允许数据库管理员进行逻辑复制操作。
- 目前默认不支持主备从部署模式。

操作步骤

(1) 以数据库安装用户登录 GBase 8c 数据库主节点。

(2) 使用如下命令通过连接默认数据库 postgres。

```
gsql -d postgres -p 15400 -r
```

其中，15400 为数据库端口号，用户可根据实际情况替换。

(3) 创建名称为 slot1 的逻辑复制槽。

```
postgres=# SELECT * FROM pg_create_logical_replication_slot('slot1',
'mppdb_decoding');
slotname | xlog_position
----------+---------------
slot1    | 0/601C150
(1 row)
```

(4) 在数据库中创建表 t，并向表 t 中插入数据。

```
postgres=# CREATE TABLE t(a int PRIMARY KEY, b int);
postgres=# INSERT INTO t VALUES(3,3);
```

(5) 读取复制槽 slot1 解码结果，解码条数为 4096。

```
postgres=# SELECT * FROM pg_logical_slot_peek_changes('slot1', NULL, 4096);
location  |  xid  | data
```

```
----------+-------+-------------------------------------------------------------
-------------------------------------------------------------------------------
-----------
-----------------------------------------
 0/601C188 | 1010023 | BEGIN 1010023
 0/601ED60 | 1010023 | COMMIT 1010023 CSN 1010022
 0/601ED60 | 1010024 | BEGIN 1010024
 0/601ED60 | 1010024 |
{"table_name":"public.t","op_type":"INSERT","columns_name":["a","b"],"columns
_type":["integer","integer"],"columns_val":["3","3"],"old_keys_name":[],"old_
keys_type":[],"old_keys_val":[]}
 0/601EED8 | 1010024 | COMMIT 1010024 CSN 1010023
(5 rows)
```

(6) 删除逻辑复制槽 slot1。

```
postgres=#  SELECT * FROM pg_drop_replication_slot('slot1');
 pg_drop_replication_slot
--------------------------


(1 row)
```

6.2 发布订阅

发布和订阅基于逻辑复制实现，其中有一个或者更多订阅者订阅一个发布者节点上的一个或者更多发布。订阅者从它们所订阅的发布拉取数据。

发布者上的更改会被实时发送给订阅者。订阅者以与发布者相同的顺序应用那些数据，这样在一个订阅中能够保证发布的事务一致性。这种数据复制的方法有时候也被称为事务性复制。

发布订阅的典型用法是：

- 在一个数据库或者一个数据库的子集中发生更改时，把增量的改变发送给订阅者。
- 在更改到达订阅者时引发触发器。
- 把多个数据库联合到单一数据库中（例如用于分析目的）。

订阅者数据库的行为与任何其他 GBase 8c 实例相同，并且可以被用作其他数据库的发布者，只需要定义它自己的发布。当订阅者被应用当作只读时，单一的订阅中不会有冲突。在另一方面，如果应用或者对相同表集合的订阅者执行了其他的写动作，冲突可能会发生。

6.2.1 发布

发布可以被定义在任何物理复制的主服务器上。定义有发布的节点被称为发布者。发布是从一个表或者一组表生成的改变的集合，也可以被描述为更改集合或者复制集合。每个发布都只存在于一个数据库中。

发布与模式不同，不会影响表的访问方式。如果需要，每个表都可以被加入到多个发布。当前，发布只能包含表。对象必须被明确地加入到发布，除非发布是用 ALL TABLES 创建的。

发布可以选择把它们产生的更改限制为 INSERT、UPDATE、DELETE 的任意组合，类似于触发器如何被特定事件类型触发的方式。默认情况下，所有操作类型都会被复制。

为了能够复制 UPDATE 和 DELETE 操作，被发布的表必须配置有一个“复制标识”，这样在订阅者那一端才能标识对于更新或删除合适的行。默认情况下，复制标识就是主键（如果有主键）。也可以在复制标识上设置另一个唯一索引（有特定的额外要求）。如果表没有合适的键，那么可以设置成复制标识“full”，它表示整个行都成为那个键。不过，这样做效率很低，只有在没有其他方案的情况下才应该使用。如果在发布者端设置了“full”之外的复制标识，在订阅者端也必须设置一个复制标识，它应该由相同的或者少一些的列组成。如何设置复制标识的细节请参考 REPLICA IDENTITY。如果在复制 UPDATE 或 DELETE 操作的发布中加入了没有复制标识的表，那么订阅者上后续的 UPDATE 或 DELETE 操作将导致错误。不管有没有复制标识，INSERT 操作都能继续下去。

每一个发布都可以有多个订阅者。

Publication 通过使用 CREATE PUBLICATION 命令创建并且可以在之后使用相应的命令进行修改或者删除。

表可以使用 ALTER PUBLICATION 动态地增加或者移除。ADD TABLE 以及 DROP TABLE 操作都是事务性的，因此一旦该事务提交，该表将以正确的快照开始或者停止复制。

6.2.2 订阅

订阅是逻辑复制的下游端。订阅被定义在其中的节点被称为订阅者。一个订阅会定义到另一个数据库的连接以及它想要订阅的发布集合（一个或者多个）。

订阅者数据库的行为与任何其他 GBase 8c 实例相同，并且可以被用作其他数据库的发布者，只需要定义它自己的发布。

如果需要，一个订阅者节点可以有多个订阅。可以在一对发布者-订阅者之间定义多个

订阅，在这种情况下要确保被订阅的发布对象不会重叠。

每一个订阅都将通过一个复制槽接收更改。预先存在的表的初始数据暂时不支持同步。

如果当前用户是一个具有 SYSADMIN 权限用户，则订阅会被 gs_dump 转储。否则订阅会被跳过并且写出一个警告，因为不具有 SYSADMIN 权限用户不能从 pg_subscription 目录中读取所有的订阅信息。

可以使用 CREATE SUBSCRIPTION 增加订阅，并且使用 ALTER SUBSCRIPTION 在任何时刻修改订阅，还可以使用 DROP SUBSCRIPTION 删除订阅。

在一个订阅被删除并且重建时，同步信息会丢失。这意味着数据必须被重新同步。

模式定义不会被复制，并且被发布的表必须在订阅者上存在。只有常规表可以成为复制的目标。例如，不能复制视图。

表在发布者和订阅者之间使用完全限定的表名进行匹配。不支持复制到订阅者上命名不同的表。

表的列也通过名称匹配。订阅表中的列顺序不需要与发布表中的顺序一样。 列的数据类型也不需要一样，只要可以将数据的文本表示形式转换为目标类型即可。 例如，您可以从 integer 类型的列复制到 bigint 类型的列。 目标表还可以具有发布表中不存在的额外列。额外列都将使用目标表的定义中指定的默认值填充。

6.2.3 冲突处理

逻辑复制的行为类似于正常的 DML 操作，即便数据在订阅者节点本地被修改，逻辑复制也会根据收到的更改来更新数据。如果流入的数据违背了任何约束，复制将停止。这种情况被称为一个冲突。在复制 UPDATE 或 DELETE 操作时，缺失的数据将不会产生冲突并且这类操作将被简单地跳过。

冲突将会产生错误并且停止复制，它必须由用户手工解决。在订阅者的服务器日志中可以找到有关冲突的详细情况。

通过更改订阅者上的数据（这样它就不会与到来的数据发生冲突）或者跳过与已有数据冲突的事务可以解决这种冲突。通过调用 pg_replication_origin_advance()函数可以跳过该事务，函数的参数是对应于该订阅名称的 node_name 以及一个位置。复制源头的当前位置可以在 pg_replication_origin_status 系统视图中看到。

6.2.4 限制

发布订阅基于逻辑复制实现，继承所有逻辑复制的限制，同时发布订阅还有下列额外的限制或者缺失的功能。

- 数据库模式和 DDL 命令不会被复制。初始模式可以手工使用 gs_dump –schema-only 进行拷贝。后续的模式改变需要手工保持同步。
- 序列数据不被复制。后台由序列支撑的 serial 或者标识列中的数据当然将被作为表的一部分复制，但是序列本身在订阅者上仍将显示开始值。如果订阅者被用作一个只读数据库，那么这通常不会是什么问题。不过，如果订阅者数据库预期有某种转换或者容错，那么序列需要被更新到最后的值，要么通过从发布者拷贝当前数据的防范（也许使用 gs_dump），要么从表本身决定一个足够高的值。
- 只有表支持复制，包括分区表。试图复制其他类型的关系，例如视图、物化视图或外部表，将会导致错误。
- 同一数据库内的多个订阅不应当订阅内容重复的发布（指发布相同的表），否则会产生数据重复或者主键冲突。
- 如果被发布的表中包含不支持 btree/hash 索引的数据类型（如地理类型等），那么该表需要有主键，才能成功的复制 UPDATE/DELETE 操作到订阅端。否则复制会失败，同时订阅端会出现“FATAL: could not identify an equality operator for type xx”的日志。
- 使用 gs_probackup 备份发布端时，由于 gs_probackup 工具本身不支持备份逻辑复制槽，所以备份恢复后，会由于逻辑复制槽不存在，而使得原有的发布订阅关系无法正常建立。建议使用 gs_basebackup 工具备份发布端。

6.2.5 架构

发布者上的更改会在它们发生时实时传送给订阅者。订阅者按照数据在发布者上被提交的顺序应用数据，这样任意单一订阅中的发布的事务一致性才能得到保证。

逻辑复制被构建在一种类似于物理流复制的架构上。它由“walsender”和“apply”进程实现。

walsender 进程开始对 WAL 的逻辑解码并且载入标准逻辑解码插件（pgoutput）。该插件把从 WAL 中读取的更改转换成逻辑复制协议并且根据发布说明过滤数据。然后数据会被连续地使用流复制协议传输到应用工作者，应用工作者会把数据映射到本地表并且以正确的事务顺序应用它们接收到的更改

订阅者数据库上的应用进程总是将 session_replication_role 设置为 replica 运行，这会产生触发器和约束上通常的效果。

逻辑复制应用进程当前仅会引发行触发器，而不会引发语句触发器。不过，初始的表同步是以类似一个 COPY 命令的方式实现的，因此会引发 INSERT 的行触发器和语句触发器。

6.2.6 监控

因为逻辑复制是基于与物理流复制相似的架构的，一个发布节点上的监控也类似于对物理复制主节点的监控。

有关订阅的监控信息在 pg_stat_subscription 中可以看到。 每一个订阅工作者在这个视图都有一行。一个订阅能有零个或者多个活跃订阅工作者取决于它的状态。

通常，对于一个已启用的订阅会有单一的应用进程运行。一个被禁用的订阅或者崩溃的订阅在这个视图中不会有行存在。如果有任何表的数据同步正在进行，对正在被同步的表会有额外的工作者。

6.2.7 安全性

用于复制连接的角色必须具有 REPLICATION 属性(或者是具有 SYSADMIN 权限用户)。如果角色缺少 SUPERUSER 和 BYPASSRLS，发布者的行安全策略可以执行。 角色的访问权限必须在 pg_hba.conf 中配置，并且必须具有 LOGIN 属性。

要创建发布，用户必须在数据库中有 CREATE 特权。

要把表加入到一个发布，用户必须在该表上有拥有权。要创建一个自动发布所有表的发布，用户必须是一个具有 SYSADMIN 权限用户。

要创建订阅，用户必须是一个具有 SYSADMIN 权限用户。

订阅的应用过程将在本地数据库上以具有 SYSADMIN 权限用户的特权运行。

特权检查仅在复制连接开始时被执行一次。在从发布者读到每一个更改记录时不会重新检查特权，在每一个更改被应用时也不会重新检查特权。

6.2.8 配置设置

发布订阅要求设置一些配置选项。

在发布者端，wal_level 必须被设置为 logical，而 max_replication_slots 中设置的值必须至少是预期要连接的订阅数。max_wal_senders 应该至少被设置为 max_replication_slots 加上同时连接的物理复制体的数量。

订阅者还要求 max_replication_slots 被设置。在这种情况下，它必须至少被设置为将被加入到该订阅者的订阅数。max_logical_replication_workers 必须至少被设置为订阅数，

6.2.9 快速设置

首先在 postgresql.conf 中设置配置选项：

```
wal_level = logical
```

对于一个基础设置来说，其他所需的设置使用默认值就足够了。

需要调整 pg_hba.conf 以允许复制（这里的值取决于实际的网络配置以及用于连接的用户）：

```
host     all     repuser     0.0.0.0/0     sha256
```

然后在发布者数据库上：

```
CREATE PUBLICATION mypub FOR TABLE users, departments;
```

并且在订阅者数据库上：

```
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=foo host=bar user=repuser'
PUBLICATION mypub;
```

上面的语句将开始复制过程，复制对那些表的增量更改。

7 升级

概述

本文档详细的描述了版本升级、回滚流程以及具体的操作指导，同时提供了常见的问题解答及故障处理方法。

读者对象

本文档主要适用于升级的操作人员。操作人员必须具备以下经验和技能：

- 熟悉当前网络的组网和相关网元的版本信息。
- 有该设备维护经验，熟悉设备的操作维护方式。

7.1 升级前必读

升级方案

本节为指导用户选择升级方式。

用户根据 GBase 8c 提供的新特性和数据库现状，确定是否对现有系统进行升级。

当前支持的升级模式为就地升级、灰度升级和滚动升级。升级方式的策略又分为大版本升级和小版本升级。版本号不变的升级方式为小版本升级，否则就是大版本升级。升级软件包的版本号在升级文件压缩包中的 version.cfg 的第 2 行查看。当前版本的版本号在 $GAUSSHOME/bin 下面 upgrade_version 文件的第 2 行查看。

用户挑选升级方式后，系统会自动判断并选择合适的升级策略。

- 就地升级：升级期间需停止业务进行，一次性升级所有节点。
- 灰度升级：灰度升级支持全业务操作，也是一次性升级所有节点。
- 滚动升级：基于灰度升级，支持升级指定节点，支持部分节点升级。

升级前的版本要求

升级前版本，可以通过执行如下工具查看。

```
gsql -V | --version
```

本文档中描述的时间仅供参考，实际操作时间以现场情况为准。

表 7-1 升级流程执行效率估计

步骤	建议起始时间	耗时（天/小时/分钟）	业务中断时长	备注
升级前准备与检查	升级操作前一天	约 2—3 小时。	对业务无影响。	升级前检查和备份数据、校验软件包等操作。
升级操作	业务空闲期	耗时主要集中在数据库的启动和停止以及每个 database 的系统表修改处。升级操作耗时一般不会超过 30 分钟。	与操作时长一致，一般不会超过 30 分钟。	依据指导书开始升级。
升级验证	业务空闲期	约 30 分钟。	与操作时长一致，约 30 分钟。	-
提交升级	业务空闲期	提交升级耗时一般不超过 10 分钟。	与操作时长一致，一般不超过 10 分钟。	-
升级版本回滚	业务空闲期	版本回滚耗时一般不会超过 30 分钟。	与操作时长一致，一般不会超过 30 分钟。	-

7.2 升级前准备与检查

升级前准备与检查清单

表 7-2 升级前准备清单

序号	升级准备项目	准备内容	建议起始时间	耗时（天/小时/分钟）
1	收集节点信息	收集到数据库涉及节点的名称、IP 地址和节点的 root、gbase 用户密码等环境信息	升级前一天	1 小时

2	设置 root 用户远程登录	设置配置文件，允许 root 用户远程登录	升级前一天	2 小时
3	备份数据	参考“备份与恢复”章节进行	升级前一天	备份数据量和方案不同，耗时也不同
4	获取并校验升级包	获取升级软件包，进行完整性校验	升级前一天	0.5 小时
5	健康检查	使用 gs_checkos 工具完成操作系统状态检查	升级前一天	0.5 小时
6	检查数据库节点磁盘使用率	使用 df 命令查看磁盘使用率	升级前一天	0.5 小时
7	检查数据库状态	使用 gs_om 工具完成数据库状态检查	升级前一天	0.5 小时

“耗时”依不同环境（包括现场数据量、服务器性能等原因）会存在一定差异。

收集节点信息

联系数据库系统管理员，获取数据库涉及节点的节点名称、节点 IP 地址。节点的 root、gbase 用户密码等环境信息。如表 7-3。

表 7-3 节点信息

序号	节点名称	节点 IP	root 用户密码	gbase 用户密码	备注
1	-	-	-	-	-

备份数据

升级一旦失败，有可能会影响到业务的正常开展。提前备份数据，就可以在风险发生后，尽快的恢复业务。

请参考备份与恢复章节，完成数据的备份。

健康检查

通过 gs_checkos 工具可以完成操作系统状态检查。

前提条件

- 当前的硬件和网络环境正常。
- 各主机间 root 互信状态正常。
- 只能使用 root 用户执行 gs_checkos 命令。

说明

➢ 该工具不支持独立调用，出于安全考虑，前置安装完成后会自动删除。

操作步骤

(1) 以 root 用户身份登录服务器。

(2) 执行如下命令对服务器的 OS 参数进行检查。

```
gs_checkos -i A
```

(3) 检查服务器的 OS 参数的目的是为了保证数据库正常通过预安装，并且在安装成功后可以安全高效的运行。详细的检查项目请参见《GBase 8c V5_5.0.0_工具与命令参考手册》中的“服务端工具 > gs_checkos”工具的“表 1 操作系统检查项”。

检查数据库节点磁盘使用率

建议数据库节点磁盘使用率低于 80%时再执行升级操作。

检查数据库状态

本节介绍数据库状态查询的具体操作。

验证步骤

(1) 以数据库用户（如 gbase）登录节点，source 环境变量。

(2) 执行如下命令查看数据库状态。

```
gs_om -t status
```

(3) 保证数据库状态正常。

7.3 升级操作

介绍就地升级、灰度升级和滚动升级的详细操作。

就地升级和灰度升级操作步骤

(1) 以 root 身份登录节点。

(2) 创建新包目录。

```
mkdir -p /opt/software/upgrade
```

(3) 将需要更新的新包上传至目录“/opt/software/upgrade”并解压。

(4) 进入安装包解压出的 script 目录下:

```
cd /opt/software/upgrade/script
```

(5) 在就地升级或灰度升级前执行前置脚本 gs_preinstall。

```
./gs_preinstall -U gbase -G gbase  -X /opt/software/Kernel/clusterconfig.xml
```

(6) 切换至 gbase 用户。

```
su - gbase
```

(7) 数据库状态正常时，使用如下命令进行就地升级或者灰度升级。

示例一：使用 gs_upgradectl 脚本执行就地升级。

```
gs_upgradectl -t auto-upgrade -X /opt/software/Kernel/clusterconfig.xml
```

示例二：使用 gs_upgradectl 脚本执行灰度升级。

```
gs_upgradectl -t auto-upgrade -X /opt/software/Kernel/clusterconfig.xml --grey
```

(8) 数据库状态正常时，使用如下命令进行滚动升级。

示例一：使用 gs_upgradectl 脚本执行指定单节点升级。

```
gs_upgradectl -t auto-upgrade -X /opt/software/Kernel/clusterconfig.xml --grey
-h hostname0
```

示例二：使用 gs_upgradectl 脚本执行指定多节点升级。

```
gs_upgradectl -t auto-upgrade -X /opt/software/Kernel/clusterconfig.xml --grey
-h hostname0,hostname1
```

(9) 数据库状态正常时，使用如下命令进行升级剩余节点。

```
gs_upgradectl -t auto-upgrade -X /opt/software/Kernel/clusterconfig.xml --grey
--continue
```

7.4 升级验证

本章介绍升级完成后的验证操作。给出验证的用例和详细操作步骤。

验证项目的检查表

表 7-4 验证项目的检查表

序号	验证项目	检查标准	检查结果
1	版本查询	查询升级后版本是否正确	-
2	健康检查	使用 gs_checkos 工具完成操作系统状态检查。	-
3	数据库状态	使用 gs_om 工具完成数据库状态检查。	-

升级版本查询

本节介绍版本查询的具体操作。

验证步骤

(1) 以数据库用户（如 gbase）登录节点，source 环境变量。

(2) 执行如下命令查看所有节点的版本信息。

```
gs_ssh -c "gsql -V"
```

检查升级数据库状态

本节介绍数据库状态查询的具体操作。

验证步骤

(1) 以数据库用户（如 gbase）登录节点。

(2) 执行如下命令查看数据库状态。

```
gs_om -t status
```

查询结果的 cluster_state 为 Normal 代表数据库正常。

7.5 提交升级

升级完成后，如果验证也没问题。接下来就可以提交升级。

说明： 一旦提交操作完成，则不能再执行回滚操作。

操作步骤

(1) 以数据库用户（如 gbase）登录节点。

(2) 执行如下命令完成升级提交。

```
gs_upgradectl -t commit-upgrade  -X /opt/software/Kernel/clusterconfig.xml
```

(3) 如果是滚动升级，需要升级完所有节点之后，才能执行提交操作。

7.6 升级版本回滚

本章介绍版本回滚方法。

操作步骤

(1) 以数据库用户（如 gbase）登录节点。

(2) 执行如下命令完成版本回滚（回滚内核代码）。回滚完成，如果需要保持内核和 om 代码的版本一致，可以执行一下旧包的前置命令（参见执行前置脚本 gs_preinstall。）。

```
gs_upgradectl -t auto-rollback  -X /opt/software/Kernel/clusterconfig.xml
```

说明

- 如果数据库异常，需要强制回滚，可以使用如下命令。

```
gs_upgradectl -t auto-rollback -X /opt/software/Kernel/clusterconfig.xml
--force
```

如果升级后再回滚，执行完回滚命令后在执行旧包的前置命令之前需要执行以下两步：

删掉集群中各个节点的/root/gauss_om/数据库用户名（比如 gbase）目录。

去掉数据库用户的互信，需要登录集群中每个节点删掉 crontab 定时任务，删除~/.ssh，杀掉互信进程，删掉 SSH_AUTH_SOCK，SSH_AGENT_PID 两个环境变量。

(3) 查看回滚之后的版本号。

```
gs_om -V | --version
```

7.7 异常处理

如果升级失败，请按照如下方式进行处理：

(1) 排查是否有环境问题。

如磁盘满、网络故障等，或者升级包、升级版本号是否正确。排除问题后，可以尝试重入升级。

(2) 如果没有发现环境问题，或者重入升级失败，需要收集相关日志，找技术支持工程师定位。

收集日志命令：

```
gs_collector --begin-time='20240301 00:00' --end-time='20240301 00:00'
```

如果条件允许，建议保留环境。

升级问题 FAQ

Q: 升级遇到如下错误，要如何处理？

```
pscp: error: no such option: --trace-id
```

A: 是由于在升级时候，该服务器上还有其他版本的 GBase 8c 数据库正在安装，导致数据库工具版本不一致。 请勿在升级过程中安装其他数据库，并回滚后重新进行升级操作。

7.8 集群管理组件增量升级

本章介绍版本集群管理组件增量升级方法。

操作步骤

- 集群管理组件升级前准备与检查
- 集群管理组件升级
- 集群管理组件升级后检查

集群管理组件增量升级注意事项:

- 集群管理组件增量升级操作不能和扩节点、缩节点同时执行。
- 集群管理组件增量升级操作不需要执行前置操作，请参考本页中升级示例六进行集群管理组件的升级。
- 建议在数据库系统业务空闲情况下进行集群管理组件的升级，尽量避开业务繁忙时段。
- 集群管理组件增量升级需要使用官方提供的组件包进行升级。
- 执行集群管理组件升级需要保障集群内节点间互信正常通信(可以通过在节点间互相执行 ssh 命令进行检查）。
- -upgrade-package 是指定集群管理升级包路径的参数，升级前请检查升级包的权限(属主、属组、读写权限）是否正常。
- 升级集群管理组件后，如果进行增加节点操作（gs_expansion）,为保障所有节点的集群管理组件的一致性，建议在增加节点操作完成后，再次执行集群管理组件的升级操作。

集群管理组件升级前准备与检查

表 7-5 集群管理组件升级前准备清单

序号	升级准备项目	准备内容	建议起始时间	耗时（天/小时/分钟）
1	检查安装版本是否支持集群管理组件升级	查看 gs_upgrade 升级工具的帮助信息是否包含 upgrade-cm 功能	升级前 1 小时	2 分钟
2	获取并校验集群管理组件升级包	获取升级软件包，进行完整性校验	升级前 1 小时	15 分钟
3	健康检查	使用 gs_checkos 工具完成操作系统状态检查	升级前 1 小时	15 分钟
4	检查数据库节点磁盘使用率	使用 df 命令查看磁盘使用率	升级前 1 小时	5 分钟
5	检查数据库状态	使用 gs_om 工具完成数据库状态检查	升级前检查	2 分钟

表中的 2-5 项详细操作请参照升级前准备与检查。

检查安装版本是否支持集群管理组件升级

登录准备执行集群管理组件升级的节点，执行 gs_upgradectl -help 命令，查看帮助信息中-t 参数值是否包含 upgrade-cm 选项。

集群管理组件升级

(1) 以集群用户 gbase 身份登录节点。

(2) 创建升级包目录。

```
mkdir -p /opt/software/cm_upgrade
```

将需要更新的新包上传至目录“/opt/software/cm_upgrade”。

(3) 执行集群管理组件升级

```
gs_upgradectl -t upgrade-cm --upgrade-package
/opt/software/cm_upgrade/GBase8cV5XXXX-cm.tar.gz
```

集群管理组件升级后检查

(1) 使用 cm_ctl 工具查看集群状态是否与升级前一致或可用状态高于升级前状态。

```
cm_ctl query -Cvd
```

(2) 查看集群临时目录下是否生成备份集群管理组件包。

```
ll $PGHOST
```

(3) 查看集群管理组件的版本。

```
cm_ctl -V
```

说明

- 执行过集群管理组件升级后，再执行增加节点操作后，新增加节点的集群管理组件是未经过升级的，需要再次执行集群管理升级以对新增加节点进行集群管理组件升级。

8 常见故障定位指南

8.1 常见故障定位手段

8.1.1 操作系统故障定位手段

查询状态时，显示一个节点上所有实例都不正常时，可能是操作系统发生了故障。

可以通过如下方法确定操作系统是否存在问题：

- 通过 SSH 或者其它远程登录工具登录该节点。如果连接失败，请尝试通过 ping 发包检查网络状态。
  - 如果 ping 操作没有回复，则表明这台机器可能存在网络连接故障、处于宕机状态或者正处于重启状态。

    如果操作系统内核发生 panic 引起系统崩溃，系统重新启动时间较慢，需经过较长时间（大约 20 分钟）才能重启。建议每 5 分钟尝试连接一次，20 分钟后不能连接成功，则表明这台机器已宕机或网络连接有问题，需要管理员到现场进行检查处理。
  - 如果网络可以 ping 通，但在 SSH 登入时卡住或登入后不能执行任何命令，通常是由系资源不足（如 CPU 或 IO 资源过载）引起的机器不响应外部连接。建议重试几次。如果 5 分钟内仍不能成功，需要管理员到现场进行检查处理。
- 可以远程登录节点，但在执行操作时，响应缓慢，需要检查系统运行情况后，进行进一步处理。例如，收集系统信息、确定系统版本、硬件、参数设置及登录用户情况。下面列出一些常用命令供参考。
  - “who”命令查看当前在线用户。

```
# who
# cat /etc/openEuler-release
# uname -a
```

  - “sysctl -a”命令（需要 root 用户执行）和“cat /etc/sysctl.conf”命令获得系统参数信息。

- “cat /proc/cpuinfo”和“cat /proc/meminfo”获得 CPU 和内存信息。

```
# cat /proc/cpuinfo
# cat /proc/meminfo
```

- “top -H”命令查看 CPU 的使用情况，确定是否因为某个进程导致 CPU 使用率过高。如果存在这种情况，通过 gdb 或 gstack 打印该程序堆栈，观察是否该程序处于死循环逻辑。
- “iostat -x 1 3”命令查看 IO 的使用情况，确定是否当前磁盘的 IO 处于饱和状态。查看当前运行的执行作业情况，决定是否对占用较多 IO 的执行作业进行处理。
- “vmstat 1 3”命令查看当前系统中内存的消耗情况，结合“top”命令获得消耗内存较多的进程，处于超出预期的状态。
- 以 root 用户查看操作系统日志信息（/var/log/messages）或 dmseg 信息，检查操作系统是否发生过异常错误。
- 操作系统的 watchdog 是为了保证 OS 系统正常运行，或者从死循环，死锁等状态退出的一种机制，如果 watchdog 超时（一般默认值为 60s），系统将会复位。

8.1.2 网络故障定位手段

在数据库正常工作的情况下，网络层对上层用户是透明的，但数据库在长期运行时，可能会由于各种原因导致出现网络异常或错误。常见的因网络故障引发的异常有：

- 数据库启动失败，报网络错误。
- 状态异常，如：节点上所有的实例都是 UnKnown 或者所有主机都切换为备机。
- 网络连接建立失败。
- 对数据库执行 SQL 操作时，报网络异常中断的错误。
- 连接数据库或执行查询时发生进程停止响应。数据库出现了网络故障后，主要通过使用 Linux 系统提供的网络相关命令工具（ping、ifconfig、netstat、lsof 等），进程堆栈查看工具（gdb、gstack），结合数据库的日志信息，进行分析定位。本节通过举例介绍常见

的网络问题，并进行基本的分析定位。

常见故障问题如下：

- **启动失败，报网络错误**

问题现象 1：日志中存在如下错误信息。可能是端口被其他进程侦听。

```
LOG: could not bind socket at the 10 time, is another postmaster already running
on port 15400?
```

处理办法：执行如下命令查看侦听该端口的进程。端口号请根据实际端口号替换。

```
# netstat -anop | grep 15400
```

根据查询结果，强行停止正在占用端口的进程或者更改数据库侦听端口。

问题现象 2：使用 gs_om -t status –detail 查询状态，如果显示主备间连接未建立。

处理办法：在 openEuler 操作系统下，使用 systemctl status firewalld.service 命令，查看节点上是否开启了防火墙。如果开启，使用 systemctl stop firewalld.service 关闭防火墙。

```
systemctl status firewalld.service
●firewalld.service - firewalld - dynamic firewall daemon
   Loaded: loaded (/usr/lib/systemd/system/firewalld.service; disabled; vendor
preset: enabled)
   Active: inactive (dead)
     Docs: man:firewalld(1)
```

操作系统不同，命令可能不同，使用对应操作系统命令查看修改。

- **数据库状态异常**

问题现象：某一节点上出现如下问题：

  - 所有实例都是 UnKnown。
  - 所有主实例都切换成了备实例。
  - 查询中出现大量 Connection reset by peer，Connection timed out 等报错信息。

处理办法

  - 如果 ssh 不能连接故障机器，在其他机器上使用 Ping 命令向该机器发数据包。如

果可以 Ping 通，说明可能是该机器上的资源（内存、CPU、磁盘）耗尽导致不能建立连接。

- 如果 ssh 可以连接该机器，尝试执行查询，并每隔 1s 执行/sbin/ifconfig eth?（?代表数字，表示第几个网卡）命令，查看如下信息中的 dropped 及 errors 值的变化情况，如果增长迅速，可能是网卡或网卡驱动故障。

```
# ifconfig enp125s0f0
enp125s0f0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.90.56.36  netmask 255.255.255.0  broadcast 10.90.56.255
        inet6 fe80::7be7:8038:f3dc:f916  prefixlen 64  scopeid 0x20<link>
        ether 44:67:47:7d:e6:84  txqueuelen 1000  (Ethernet)
        RX packets 129344246  bytes 228050833914 (212.3 GiB)
        RX errors 0  dropped 647228  overruns 0  frame 0
        TX packets 96689431  bytes 97279775245 (90.5 GiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

检查如下参数设置是否正确。

```
net.ipv4.tcp_retries1 = 3
net.ipv4.tcp_retries2 = 15
```

- **网络连接建立失败**

问题现象 1：节点连接其他节点失败，日志中报出“ Connection refused ”错误。

处理办法：

- 查看端口是否配置错误，导致连接时使用的端口并非对方侦听的端口。查看报错节点配置文件 postgresql.conf 记录端口号与对方侦听的端口就号是否一致。
- 查看对方端口侦听是否正常（可以使用“netstat –anp”命令）。
- 查看对方进程是否存在。

问题现象 2：对数据库执行 SQL 操作时，获取连接描述符失败，报错如下。

```
WARNING:  29483313: incomplete message from client:4905,9
WARNING:  29483313: failed to receive connDefs at the time:1.
ERROR:  29483313: failed to get pooled connections
```

在日志中，找到上面的错误，并向上查看一段日志内容，可以看到详细的错误信息，常

见的错误如下所示，主要是由于主备信息不正确导致。

```
FATAL:  dn_6001_6002: can not accept connection in pending mode.
FATAL:  dn_6001_6002: the database system is starting up
FATAL:  dn_6009_6010: can not accept connection in standby mode.
```

处理办法：

- 使用 gs_om -t status –detail 查询状态，确认是否发生过主备切换。重置实例状态。
- 此外，需要查看连接失败的节点是否发生了 core 或者重启。通过 om 日志可以查看到是否发生了重启。

- **对数据库执行 SQL 操作时，报网络异常中断的错误**

问题现象 1：查询执行失败，报错信息如下。

```
ERROR:  dn_6065_6066: Failed to read response from Datanodes. Detail: Connection reset by peer. Local: dn_6065_6066 Remote: dn_6023_6024
ERROR: Failed to read response from Datanodes Detail: Remote close socket unexpectedly
ERROR: dn_6155_6156: dn_6151_6152: Failed to read vector response from Datanodes
```

连接建立失败，报错信息如下。

```
ERROR:  Distribute Query unable to connect 10.145.120.79:14600 [Detail:stream connect connect() fail: Connection timed out
ERROR: Distribute Query unable to connect 10.144.192.214:12600 [Detail:receive accept response fail: Connection timed out
```

处理办法：

- 使用 gs_check 检查网络配置是否符合标准。详细参考《工具参考》中“服务端工具 > gs_check”章节中对 network 的检查。
- 查看是否有进程发生 core 或重启，以及主备切换。
- 如果不存在上述问题，可以联系网络技术人员做具体分析。

8.1.3 磁盘故障定位手段

常见的磁盘故障是磁盘空间不足、磁盘出现坏块、磁盘未挂载等。部分磁盘故障会导致文件系统损坏，例如磁盘未挂载，数据库管理自动定期执行磁盘检测时会识别故障并将实例

停止，查看数据库状态时对应实例状态异常；部分磁盘故障不会导致文件系统损坏，例如磁盘空间不足，数据库管理无法检测到，服务进程访问到故障磁盘会异常退出，例如数据库无法启动、checksum 校验不对、页面读写失败、页面校验错误等。

- 对于会导致文件系统损坏的故障，查看状态会显示对应实例状态持续为 Unknown，定位方法如下：
  - 查看日志，日志中会有类似 “data path disc writable test failed”异常，说明文件系统已损坏。
  - 文件系统损坏可能是磁盘未挂载，通过 ls –l 可以看到该磁盘对应的目录权限异常，如下。
  - 也可能是磁盘出现坏块，然后操作系统将文件系统保护起来，拒绝读写，可以使用磁盘坏块检查工具如 badblocks 检查磁盘是否有坏块，如下。

```
# badblocks /dev/sdb1 -s -v
Checking blocks 0 to 2147482623
Checking for bad blocks (read-only test): done
Pass completed, 0 bad blocks found. (0/0/0 errors)
```

- 对于不会导致文件系统损坏的故障，服务进程访问到故障磁盘会异常退出，定位方法如下。

查看日志。日志中会有文件读写错误，例如“No space left on device”、“ invalid page header n block 122838 of relation base/16385/152715”。 文件读写错误可能是磁盘空间不足，通过 df -h 可以看到磁盘空间已达 100%，如下。

```
# df -h
Filesystem                 Size  Used Avail Use% Mounted on
devtmpfs                   255G     0  255G   0% /dev
tmpfs                      255G   35M  255G   1% /dev/shm
tmpfs                      255G   57M  255G   1% /run
tmpfs                      255G     0  255G   0% /sys/fs/cgroup
/dev/mapper/openeuler-root  196G  8.8G  178G   5% /
tmpfs                      255G  1.0M  255G   1% /tmp
/dev/sda2                  9.8G  144M  9.2G   2% /boot
/dev/sda1                   10G  5.8M   10G   1% /boot/efi
```

```
/dev/mapper/openeuler-home   1.5T    69G   1.4T    5% /home
tmpfs                         51G     0    51G    0% /run/user/0
tmpfs                         51G     0    51G    0% /run/user/1004
/dev/sdb1                    2.0T   169G   1.9T    9% /data
```

8.1.4 数据库故障定位手段

- 日志。数据库日志记录了数据库服务端启动、运行或停止时出现的问题，当数据库在启动、运行或停止的过程中出现问题时，数据库用户可以通过运行日志快速分析问题的产生原因，并根据不同的原因采取相应的处理方法，尽可能地解决问题。
- 视图。数据库提供了许多视图，用于展示数据库的内部状态，在定位故障时，经常使用的视图如下：
  - pg_stat_activity，用于查询当前实例上各个 session 的状态。
  - pg_thread_wait_status，用于查询该实例上各个线程的等待事件。
  - pg_locks，用于查询当前实例上的锁状态。
- CORE 文件。数据库相关进程在运行过程中可能会因为各种意外情况导致数据库崩溃（Coredump），而崩溃时产生的 core 文件对于迅速定位程序崩溃的原因及位置非常重要。如果进程运行时出现 Coredump 现象，建议立即收集 core 文件便于分析、定位故障。
  - 对性能有一定的影响，尤其是进程频繁异常时对性能的影响更大。
  - core 文件会占用磁盘空间。因此，当检查到 core 文件产生后，应及时解决以避免对操作系统带来更严重的影响。操作系统自带 core dump 机制。开启后，系统中所有出现 Coredump 问题时都会生成 core 文件，对操作系统带来性能和磁盘占用的影响。
  - 设置 core 文件生成路径。修改/proc/sys/kernel/core_pattern 内容。

```
# cat /proc/sys/kernel/core_pattern
/data/jenkins/workspace/GBase
8cInstall/dbinstall/cluster/corefile/core-%e-%p-%t
```

8.2 常见故障定位案例

8.2.1 core 问题定位

8.2.1.1 磁盘满故障引起的 core 问题

问题现象

TPCC 运行时，注入磁盘满故障，数据库进程 gaussdb core 掉，如下图所示。

原因分析

数据库本身机制，在磁盘满时，xlog 日志无法进行写入，通过 panic 日志退出程序。

处理办法

外部监控磁盘使用状况，定时进行清理磁盘。

8.2.1.2 GUC 参数 log_directory 设置不正确引起的 core 问题

问题现象

数据库进程拉起后出现 coredump，日志无内容。

原因分析

GUC 参数 log_directory 设置的路径不可读取或无访问权限，数据库在启动过程中进行校验失败，通过 panic 日志退出程序。

处理办法

GUC 参数 log_directory 设置为合法路径，具体请参考 log_directory。

8.2.1.3 开启 RemoveIPC 引起的 core 问题

问题现象

操作系统配置中 RemoveIPC 参数设置为 yes，数据库运行过程中出现宕机，并显示如下日志消息。

```
FATAL: semctl(1463124609, 3, SETVAL, 0) failed: Invalid argument
```

原因分析

当 RemoveIPC 参数设置为 yes 时，操作系统会在对应用户退出时删除 IPC 资源（共享内存和信号量），从而使得 GBase 8c 服务器使用的 IPC 资源被清理，引发数据库宕机。

处理分析

设置 RemoveIPC 参数为 no。设置方法请参考《安装指南》中“安装准备>准备软硬件安装环境>修改操作系统配置”章节。

8.2.2 TPCC 运行时，注入磁盘满故障，TPCC 卡住的问题

问题现象

TPCC 运行时，注入磁盘满故障，TPCC 卡住，故障消除后，TPCC 自动续跑。

原因分析

数据库本身机制，在性能日志（gs_profile）所在磁盘满时，导致无法写入而陷入无限等待，表现为 TPCC 卡住。磁盘满故障消除后，性能日志能正常写入，TPCC 恢复正常。

处理分析

外部监控磁盘使用状况，定时进行清理磁盘。

8.2.3 备机处于 need repair(WAL)状态问题

问题现象

GBase 8c 备机出现 Standby Need repair(WAL)故障。

原因分析

因网络故障、磁盘满等原因造成主备实例连接断开，主备日志不同步，导致数据库在启动时异常。

处理分析

通过 gs_ctl build -D 命令对故障节点进行重建，具体的操作方法请参见 GBase 8c 工具

参考中的 build 参数。

8.2.4 内存不足问题

问题现象

客户端或日志里出现错误：memory usage reach the max_dynamic_memory。

原因分析

出现内存不足可能因 GUC 参数 max_process_memory 值设置较小相关，该参数限制一个 GBase 8c 实例可用最大内存。

处理分析

通过工具 gs_guc 适当调整 max_process_memory 参数值。注意需重启实例生效。

8.2.5 服务启动失败

问题现象

服务启动失败。

原因分析

- 配置参数不合理，数据库因系统资源不足，或者配置参数不满足内部约束，启动失败。
- 由于部分数据节点状态不正常，导致数据库启动失败。
- 目录权限不够。例如对/tmp 目录、数据库数据目录的权限不足。
- 配置的端口已经被占用。
- 开启了系统防火墙导致数据库启动失败。
- 组成数据库的各台机器之间需要正确建立互信关系，在互信关系出现异常的情况下，数据库将无法启动。
- 数据库控制文件损坏。

处理办法

- 确认是否由于参数配置不合理导致系统资源不足或不满足内部约束启动失败。
  - 登录启动失败的节点，检查运行日志确认是否因资源不足启动失败或配置参数不满足内部约束。例如出现 Out of memory 的错误或如下错误提示均为资源不足或配置参数不满足内部约束导致的启动失败。

```
FATAL: hot standby is not possible because max_connections = 10 is a lower setting than on the master server (its value was 100)
```

  - 检查 GUC 参数配置的合理性。例如，shared_buffers、effective_cache_size、bulk_write_ring_size 等消耗资源过大的参数；或 max_connections 等增加后不容易减少的参数。GUC 参数的查看及设置方法，详情请参见配置运行参数。
- 确认是否由于实例状态不正常，导致数据库启动失败。通过 gs_om -t status –detail 工具，查询当前数据库各主备机实例的状态。
  - 如果某一节点上的所有实例都异常，请进行主机替换。
  - 如果发现某一实例状态为 Unknown、Pending 和 Down 的状态，则以数据库用户登录到状态不正常的实例所在节点，查看该实例的日志检查状态异常的原因。例如：

```
……DETAIL:  The database subdirectory "base/ 13252" is missing.
```

如果日志中出现上面这种报错信息，则说明该数据节点的数据目录文件遭到破坏，该实例无法执行正常查询，需要进行替换实例操作。

- 目录权限不够处理办法。例如，对/tmp 目录、数据库数据目录的权限不足。
  - 根据错误提示，确认权限不足的目录名称。
  - 使用 chmod 命令修改目录权限使其满足要求。对于/tmp 目录，数据库用户需要具有读写权限。对于数据库数据目录，请参考权限无问题的同类目录进行设置。
- 确认是否由于配置的端口已经被占用，导致数据库启动失败。

- 登录启动失败的节点，查看实例进程是否存在。
- 如果实例进程不存在，则可以通过查看该实例的日志来检查启动异常的原因。

  例如：

```
2014-10-17 19:38:23.637 CST 139875904172320 LOG:  could not bind IPv4 socket at the 0 time: Address already in use 2014-10-17 19:38:23.637 CST 139875904172320 HINT:  Is another postmaster already running on port 40005? If not, wait a few seconds and retry.
```

如果日志中出现上面这种报错信息，则说明该数据节点的 TCP 端口已经被占用，该实例无法正常启动。

```
2015-06-10 10:01:50 CST 140329975478400 [SCTP MODE] WARNING: (sctp bind) bind(socket=9, [addr:0.0.0.0,port:1024]):Address already in use  --  attempt 10/10 2015-06-10 10:01:50 CST 140329975478400 [SCTP MODE] ERROR: (sctp bind) Maximum bind() attempts. Die now...
```

如果日志中出现上面这种报错信息，则说明该数据节点的 SCTP 端口已经被占用，该实例无法正常启动。

- 通过 sysctl -a 查看 net.ipv4.ip_local_port_range，如果该实例配置的端口在系统随机占用端口号的范围内，则可以修改系统随机占用端口号的范围，确保 xml 文件中所有实例端口号均不在这个范围内。检查某个端口是否被占用的命令如下。

```
netstat -anop | grep 端口号
```

示例如下。

```
# netstat -anop | grep 15970
tcp        0      0 127.0.0.1:15970         0.0.0.0:*               LISTEN
3920251/gaussdb      off (0.00/0/0)
tcp6       0      0 ::1:15970               :::*                    LISTEN
3920251/gaussdb      off (0.00/0/0)
unix  2      [ ACC ]     STREAM     LISTENING     197399441 3920251/gaussdb
/tmp/.s.PGSQL.15970
unix  3      [ ]         STREAM     CONNECTED     197461142 3920251/gaussdb
/tmp/.s.PGSQL.15970
```

- 确认是否是由于开启了系统防火墙导致数据库启动失败。
- 确认是否由于互信关系出现异常，导致数据库无法启动。重新配置实例中各台机器

的互信关系解决此问题。

- 确认是否由于数据库控制文件如 gaussdb.state 损坏或文件被清空，导致数据库无法启动。若主机控制文件损坏，可触发备机 failover，然后通过重建恢复原主机；若备机控制文件损坏，可直接通过重建方式恢复备机。

8.2.6 出现“Error:No space left on device”提示

问题现象

在数据库使用过程中，出现如下错误提示。

```
Error:No space left on device
```

原因分析

磁盘空间不足造成此提示信息。

处理办法

- 使用如下命令查看磁盘占用情况。显示信息如下，其中 Avail 列表示各磁盘可用的空间，Use%列表示已使用的磁盘空间百分比。

```
# df -h
Filesystem                 Size  Used Avail Use% Mounted on
devtmpfs                   255G     0  255G   0% /dev
tmpfs                      255G   35M  255G   1% /dev/shm
tmpfs                      255G   57M  255G   1% /run
tmpfs                      255G     0  255G   0% /sys/fs/cgroup
/dev/mapper/openeuler-root  196G  8.8G  178G   5% /
tmpfs                      255G  1.0M  255G   1% /tmp
/dev/sda2                  9.8G  144M  9.2G   2% /boot
/dev/sda1                   10G  5.8M   10G   1% /boot/efi
```

由于业务数据的增长情况不同，对剩余磁盘空间的要求不同。建议如下：

持续观察磁盘空间增长情况，确保剩余空间满足一年以上的增长要求。

数据目录所在磁盘已使用空间>60%则进行空间清理或者扩容。

- 使用如下命令查看数据目录大小。

```
du --max-depth=1 -h /mnt/
```

显示如下信息，其中第一列表示目录或文件的大小，第二列是“/mnt/”目录下的所有子目录或者文件。

```
du --max-depth=1 -h /mnt
83G /mnt/data3
71G /mnt/data2
365G    /mnt/data1
518G    /mnt
```

- 清理磁盘空间。建议定期将审计日志备份到其他存储设备，推荐的日志保留时长为一个月。pg_log 存放数据库各进程的运行日志，运行日志可以帮助数据库管理员定位数据库的问题。如果每日查看错误日志并及时处理错误，则可以删除这些日志。
- 清理无用的数据。通过先备份使用频率较低或者一定时间以前的数据至更低成本的存储介质中，然后清理这些已备份的数据来获取更多的磁盘空间。
- 如果以上方法无法清理出足够的空间，请对磁盘空间进行扩容。

8.2.7 在 XFS 文件系统中，使用 du 命令查询数据文件大小大于文件实际大小

问题现象

在数据库使用过程中，通过如下 du 命令查询数据文件大小，查询结果大于文件实际的大小。

```
du -sh file
```

原因分析

XFS 文件系统有预分配机制，预分配的大小由参数 allocsize 确定。du 命令显示的文件大小包括该预分配的磁盘空间。

处理办法

- 将 XFS 文件系统挂载选项 allocsize 设置为默认值（64KB）减缓该问题。
- 使用 du 命令时，增加–apparent-size 选项，查看实际文件的大小。

```
du -sh file --apparent-size
```

- XFS 文件系统有回收预分配空间的机制，文件系统可以通过回收文件预分配的空间，使 du 命令显示为实际文件的大小。

8.2.8 在 XFS 文件系统中，出现文件损坏

问题现象

在数据库使用过程中，有极小的概率出现 XFS 文件系统的报错（Input/Output error，structure needs cleaning）。

原因分析

此为 XFS 文件系统问题。

处理办法

首先尝试 umount/mount 对应文件系统，重试看是否可以规避此问题。

如果问题重现，则需要参考文件系统相应的文档请系统管理员对文件系统进行修复，例如 xfs_repair。文件系统成功修复后，请使用 gs_ctl build 命令来修复文件受损的数据节点。

8.2.9 switchover 操作时，主机降备卡住

问题现象

一主多备模式下，系统资源不足时，发生 switchover，出现主机降备时卡住。

原因分析

当系统资源不足时，无法创建第三方管理线程，导致其管理的子线程无法退出，出现主机降备时卡住。

处理办法

需要执行以下命令终止主机进程，使备机正常升主。确定为上述场景时执行以下操作，不是上述场景时请勿按照本方法执行。

```
kill -9 PID
```

8.2.10磁盘空间达到阈值，数据库只读

问题现象

执行非只读 SQL 时报错如下。

```
ERROR: cannot execute %s in a read-only transaction.
```

或者运行中部分非只读 SQL（insert、update、create table as、create index、alter table 及 copy from 等）时报错。

```
canceling statement due to default_transaction_read_only is on.
```

原因分析

磁盘空间达到阈值后，设置数据库只读，只允许只读语句执行。

处理办法

1. 使用 maintenance 模式连接数据库，以下两种方法均可。

方式一

```
gsql -d postgres -p 15400 -r -m
```

方式二

```
gsql -d postgres -p 15400 -r
```

连接成功后，执行如下命令：

```
set xc_maintenance_mode=on;
```

2. 使用 DROP/TRUNCATE 语句删除当前不再使用的用户表，直至磁盘空间使用率小于设定的阈值。

删除用户表只能暂时缓解磁盘空间不足的问题，建议尽早通过扩容解决磁盘空间不足的问题。

3. 使用系统用户 gbase 设置数据库只读模式关闭。

```
gs_guc reload -D /home/gbase/data/dn1/dn1_1/ -c
"default_transaction_read_only=off"
```

8.2.11 分析查询语句长时间运行的问题

问题现象

系统中部分查询语句运行时间过长。

原因分析

查询语句较为复杂，需要长时间运行。

查询语句阻塞。

处理办法

1. 以操作系统用户 gbase 登录主机。

2. 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

postgres 为需要连接的数据库名称，15400 为端口号。

3. 查看系统中长时间运行的查询语句。

```
SELECT timestampdiff(minutes, query_start, current_timestamp) AS runtime,
datname, usename, query FROM pg_stat_activity WHERE state != 'idle' ORDER BY 1
desc;
```

查询会返回按执行时间长短从大到小排列的查询语句列表。第一条结果就是当前系统中执行时间长的查询语句。

如果当前系统较为繁忙，可以使用 TIMESTAMPDIFF 函数通过限制 current_timestamp 和 query_start 大于某一阈值查看执行时间超过此阈值的查询语句。timestampdiff 的第一个参数为时间差单位。例如，执行超过 2 分钟的查询语句可以通过如下语句查询。

```
SELECT query FROM pg_stat_activity WHERE timestampdiff(minutes, query_start,
current_timestamp) > 2;
```

4. 分析长时间运行的查询语句状态。

- 如果查询语句处于正常状态，则等待其执行完毕。
- 如果查询语句阻塞，请参见分析查询语句是否被阻塞处理。

8.2.12分析查询语句运行状态

问题现象

系统中部分查询语句运行时间过长，需要分析查询语句的运行状态。

处理办法

1. 以操作系统用户 gbase 登录主机。

2. 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

postgres 为需要连接的数据库名称，15400 为端口号。

3. 设置参数 track_activities 为 on。

```
SET track_activities = on;
```

当此参数为 on 时，数据库系统才会收集当前活动查询的运行信息。

4. 查看正在运行的查询语句。以查看视图 pg_stat_activity 为例。

```
SELECT datname, usename, state, query FROM pg_stat_activity;
datname  | usename | state  | query
----------+---------+--------+-------
postgres | gbase     | idle   |
postgres | gbase     | active |
(2 rows)
```

如果 state 字段显示为 idle，则表明此连接处于空闲，等待用户输入命令。 如果仅需要查看非空闲的查询语句，则使用如下命令查看。

```
SELECT datname, usename, state, query FROM pg_stat_activity WHERE state !='idle';
```

5. 分析查询语句为活跃状态还是阻塞状态。通过如下命令查看当前处于阻塞状态的查询语句。

```
SELECT datname, usename, state, query FROM pg_stat_activity WHERE waiting = true;
```

查询结果中包含了当前被阻塞的查询语句，该查询语句所请求的锁资源可能被其他会话持有，正在等待持有会话释放锁资源。

8.2.13强制结束指定的问题会话

问题现象

有些情况下，为了使系统继续提供服务，管理员需要强制结束有问题的会话。

处理办法

1. 以操作系统用户 gbase 登录主机。

2. 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

postgres 为需要连接的数据库名称，15400 为端口号。

3. 从当前活动会话视图查找问题会话的线程 ID。

```
SELECT datid, pid, state, query FROM pg_stat_activity;
```

显示类似如下信息，其中 pid 的值即为该会话的线程 ID。

```
datid |       pid       | state  | query
-------+-----------------+--------+------
13205 | 139834762094352 | active |
13205 | 139834759993104 | idle   |
(2 rows)
```

4. 根据线程 ID 结束会话。

```
SELECT pg_terminate_backend(139834762094352);
```

显示类似如下信息，表示结束会话成功。

```
pg_terminate_backend
----------------------
t
(1 row)
```

显示类似如下信息，表示用户正在尝试结束当前会话，此时仅会重连会话，而不是结束会话。

```
FATAL:  terminating connection due to administrator command
FATAL:  terminating connection due to administrator command The connection to the server was lost. Attempting reset: Succeeded.
```

8.2.14分析查询语句是否被阻塞

问题现象

数据库系统运行时，在某些业务场景下，查询语句会被阻塞，导致语句运行时间过长。

原因分析

查询语句需要通过加锁来保护其要访问的数据对象。当要进行加锁时发现要访问的数据对象已经被其他会话加锁，则查询语句会被阻塞，等待其他会话完成操作并释放锁资源。这些需要加锁访问的数据对象主要包括表、元组等。

处理办法

1. 以操作系统用户 gbase 登录主机。

2. 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

postgres 为需要连接的数据库名称，15400 为端口号。

3. 从当前活动会话视图查找问题会话的线程 ID。

```
SELECT w.query AS waiting_query, w.pid AS w_pid, w.usename AS w_user, l.query AS
locking_query, l.pid AS l_pid, l.usename AS l_user, t.schemaname || '.' ||
t.relname AS tablename FROM pg_stat_activity w JOIN pg_locks l1 ON w.pid = l1.pid
AND NOT l1.granted JOIN pg_locks l2 ON l1.relation = l2.relation AND l2.granted
JOIN pg_stat_activity l ON l2.pid = l.pid JOIN pg_stat_user_tables t ON
l1.relation = t.relid WHERE w.waiting = true;
```

4. 根据线程 ID 结束会话。

```
SELECT pg_terminate_backend(139834762094352);
```

显示类似如下信息，表示结束会话成功。

```
pg_terminate_backend
----------------------
t
(1 row)
```

显示类似如下信息，表示用户正在尝试结束当前会话，此时仅会重连会话，而不是结束

会话。

```
FATAL:  terminating connection due to administrator command
FATAL:  terminating connection due to administrator command The connection to the
server was lost. Attempting reset: Succeeded.
```

8.2.15分析查询效率异常降低的问题

问题现象

通常在几十毫秒内完成的查询，有时会突然需要几秒的时间完成；而通常需要几秒完成的查询，有时需要半小时才能完成。

处理办法

通过下列的操作步骤，分析查询效率异常降低的原因。

1. 使用 analyze 命令分析数据库。

使用 analyze 命令会更新所有表中数据大小以及属性等相关统计信息，建议在作业压力较小时执行。如果此命令执行后性能恢复或者有所提升，则表明 autovacuum 未能很好的完成它的工作，有待进一步分析。

2. 检查查询语句是否返回了多余的数据信息。

例如，如果查询语句先查询一个表中所有的记录，而只用到结果中的前 10 条记录。对于包含 50 条记录的表，查询速度较快；但是，当表中包含的记录数达到 50000 条，查询效率将会有所下降。如果业务应用中存在只需要部分数据信息，但是查询语句却是返回所有信息的情况，建议修改查询语句，增加 LIMIT 子句来限制返回的记录数。这样使数据库优化器有了一定的优化空间，一定程度上会提升查询效率。

3. 检查查询语句单独运行时是否仍然较慢。

尝试在数据库没有其他查询或查询较少的时候运行查询语句，并观察运行效率。如果效率较高，则说明可能是由于之前运行数据库系统的主机负载过大导致查询低效。此外，还可能是由于执行计划比较低效，但是由于主机硬件较快使得查询效率较高。

4. 检查重复相同查询语句的执行效率。

查询效率低的一个重要原因是查询所需信息没有缓存在内存中，这可能是由于内存资源紧张，导致缓存信息被其他查询处理覆盖。重复执行相同的查询语句，如果后续执行的查询语句效率提升，则可能是由于上述原因导致。

8.2.16执行 SQL 语句时，提示 Lock wait timeout

问题现象

执行 SQL 语句时，提示“Lock wait timeout”。

```
ERROR:  Lock wait timeout: thread 140533638080272 waiting for ShareLock on relation 16409 of database 13218 after 1200000.122 ms ERROR:  Lock wait timeout: thread 140533638080272 waiting for AccessExclusiveLock on relation 16409 of database 13218 after 1200000.193 ms
```

原因分析

数据库中存在锁等待超时现象。

处理办法

- 数据库在识别此类错误后，会自动进行重跑，重跑次数可以使用 max_query_retry_times 控制。
- 需要分析锁超时的原因，查看系统表 pg_locks，pg_stat_activity 可以找出超时的 SQL 语句。

8.2.17VACUUM FULL 一张表后，表文件大小无变化

问题现象

使用 VACUUM FULL 命令对一张表进行清理，清理完成后表大小和清理前一样大。

原因分析

假定该表的名称为 table_name，对于该现象可能有以下两种原因：

- table_name 表本身没有 delete 过数据，使用 VACUUM FULL table_name 后无需清理 delete 的数据。因此表大小清理前后一样大。

- 在执行 VACUUM FULL table_name 时有并发的事务存在，可能会导致 VACUUM FULL 跳过清理最近删除的数据，导致清理不完全。

处理办法

对于第二种可能原因，有如下两种处理方法：

- 如果在 VACUUM FULL 时有并发的事务存在，此时需要等待所有事务结束，再次执行 VACUUM FULL 命令对该表进行清理。
- 如果使用上面的方法清理后，表文件大小仍然无变化，确认无业务操作后，使用以下 SQL 查询活跃事务列表状态：

```
select txid_current();
```

使用该 SQL 可以查询当前的事务 XID。再使用以下命令查看活跃事务列表：

```
select txid_current_snapshot();
```

如果发现活跃事务列表中有 XID 比当前的事务 XID 小时，停止数据库再启动数据库，再次使用 VACUUM FULL 命令对该表进行清理。

8.2.18执行修改表分区操作时报错

问题现象

执行 ALTER TABLE PARTITION 时，报错如下。

```
ERROR:start value of partition "XX" NOT EQUAL up-boundary of last partition.
```

原因分析

在同一条 ALTER TABLE PARTITION 语句中，既存在 DROP PARTITION 又存在 ADD PARTITION 时，无论它们在语句中的顺序是什么，GBase 8c 总会先执行 DROP PARTITION 再执行 ADD PARTITION。执行完 DROP PARTITION 删除末尾分区后，再执行 ADD PARTITION 操作会出现分区间隙，导致报错。

处理办法

为防止出现分区间隙，需要将 ADD PARTITION 的 START 值前移。示例如下。

```
--创建分区表 partitiontest。
postgres=#  CREATE TABLE partitiontest
(
c_int integer,
c_time TIMESTAMP WITHOUT TIME ZONE
)
PARTITION BY range (c_int)
(
partition p1 start(100)end(108),
partition p2 start(108)end(120)
);
--使用如下两种语句会发生报错:
postgres=#  ALTER TABLE partitiontest ADD PARTITION p3 start(120)end(130), DROP
PARTITION p2;
ERROR:  start value of partition "p3" NOT EQUAL up-boundary of last partition.
postgres=#  ALTER TABLE partitiontest DROP PARTITION p2,ADD PARTITION p3
start(120)end(130);
ERROR:  start value of partition "p3" NOT EQUAL up-boundary of last partition.
--可以修改语句为:
postgres=#  ALTER TABLE partitiontest ADD PARTITION p3 start(108)end(130), DROP
PARTITION p2;
postgres=#  ALTER TABLE partitiontest DROP PARTITION p2,ADD PARTITION p3
start(108)end(130);
```

8.2.19不同用户查询同表显示数据不同

问题现象

2 个用户登录相同数据库 human_resource，同样执行如下查询语句，查询同一张表 areas 时，查询结果却不一致。

```
select count(*) from areas;
```

原因分析

1. 检查同名表是否是同一张表。在关系型数据库中，确定一张表通常需要 3 个因素：database、schema、table。从问题现象描述看，database、table 已经确定，分别是 human_resource、areas。

2. 检查同名表的 schema 是否一致。使用 gbase、user01 分别登录发现，search_path 依次是 public 和“$user”。gbase 作为数据库管理员，默认不会创建 gbase 同名的 schema，即不

指定 schema 的情况下所有表都会建在 public 下。而对于普通用户如 user01，则会在创建用户时，默认创建同名的 schema，即不指定 schema 时表都会创建在 user01 的 schema 下。

3. 如果最终判断是同一张表，存在不同用户访问数据不同情况，则需要进一步判断当前该表中对象针对不同的用户是否存在不同的访问策略。

处理办法

对于不同 schema 下同名表的查询，在查询表时加上 schema 引用。格式如下。

```
schema.table
```

- 对于不同访问策略造成对同一表查询结果不同时，可以通过查询 pg_rlspolicy 系统表来确认具体的访问准则。

8.2.20修改索引时只调用索引名提示索引不存在

问题现象

修改索引时只调用索引名提示索引不存在。举例如下。

```
--创建分区表索引 HR_staffS_p1_index1，不指定索引分区的名称。
CREATE INDEX HR_staffS_p1_index1 ON HR.staffS_p1 (staff_ID) LOCAL;
--创建分区索引 HR_staffS_p1_index2，并指定索引分区的名称。
CREATE INDEX HR_staffS_p1_index2 ON HR.staffS_p1 (staff_ID) LOCAL
(
PARTITION staff_ID1_index,
PARTITION staff_ID2_index TABLESPACE example3,
PARTITION staff_ID3_index TABLESPACE example4
) TABLESPACE example;
--修改索引分区 staff_ID2_index 的表空间为 example1，提示索引不存在。
ALTER INDEX HR_staffS_p1_index2 MOVE PARTITION staff_ID2_index TABLESPACE
example1;
```

原因分析

推测是当前模式是 public 模式，而不是 hr 模式，导致检索不到该索引。

```
--执行如下命令验证推测，发现调用成功。
ALTER INDEX hr.HR_staffS_p1_index2 MOVE PARTITION staff_ID2_index TABLESPACE
example1;
```

```
--修改当前会话的 schema 为 hr。
ALTER SESSION SET CURRENT_SCHEMA TO hr;
--执行如下命令修改索引，即可执行成功。
ALTER INDEX HR_staffS_p1_index2 MOVE PARTITION staff_ID2_index TABLESPACE
example1;
```

处理办法

在操作表、索引、视图时加上 schema 引用，格式如下。

```
schema.table
```

8.2.21 重建索引失败

问题现象

当 Desc 表的索引出现损坏时，无法进行一系列操作，可能的报错信息如下。

```
index \"%s\" contains corrupted page at block
 %u" ,RelationGetRelationName(rel),BufferGetBlockNumber(buf), please reindex
it.
```

原因分析

在实际操作中，索引会由于软件或者硬件问题引起崩溃。例如，当索引分裂完发生磁盘空间不足、出现页面损坏等问题时，会导致索引损坏。

处理办法

如果此表是以 pg_cudesc_xxxxx_index 进行命名则为列存表，则说明 desc 表的索引表损坏。通过 desc 表的索引表表名，找到对应主表的 oid 和表，执行如下语句重建表的索引。

```
REINDEX INTERNAL TABLE name;
```

8.2.22 业务运行时整数转换错

问题现象

在转换整数时报错如下。

```
Invalid input syntax for integer: "13."
```

原因分析

部分数据类型不能转换成目标数据类型。

处理办法

逐步缩小 SQL 范围确定不能转换的数据类型。

8.2.23 高并发报错”too many clients already”或无法创建线程

问题现象

高并发执行 SQL，报错“sorry, too many clients already”；或报无法创建线程、无法 fork 进程等错误。

原因分析

该类报错是由于操作系统线程资源不足引起，查看操作系统 ulimit -u，如果过小（例如小于 32768），则基本可以判断是操作系统限制引起的。

处理办法

通过“ulimit -u”命令查看操作系统 max user processes 的值。

```
ulimit -u
unlimited
```

按如下简易公式计算需要设置的最小值。

```
value=max(32768，实例数目*8192)
```

其中实例数目指本节点所有实例总数。

设置最小值方法为，修改/etc/security/limits.conf，追加如下两行：

```
* hard nproc [value]
* soft nproc [value]
```

对于不同操作系统修改方式略有不同，centos6 以上版本可以修改/etc/security/limits.d/90-nofile.conf 文件，方法同上。

另外，也可以直接通过如下命令设置，但 OS 重启会失效，可以添加到全局环境变量/etc/profile 文件中使其生效。

```
ulimit -u [values]
```

在大并发模式下，建议开启线程池，使数据库内部的线程资源受控。

8.2.24 btree 索引故障情况下应对策略

问题现象

偶发索引丢失错误，报错如下。

```
ERROR: index ’xxxx_index’ contains unexpected zero page
或
ERROR: index ’pg_xxxx_index’ contains unexpected zero page
或
ERROR: compressed data is corrupt
```

原因分析

该类错误是因为索引发生故障导致的，可能引发故障的原因如下：

- 由于软件 bug 或者硬件原因导致的索引不再可用。
- 索引包含许多空的页面或者几乎为空的页面。
- 并发执行 DDL 过程中，发生了网络闪断故障。
- 创建并发索引时失败，遗留了一个失效的索引，这样的索引无法被使用。
- 执行 DDL 或者 DML 操作时，网络出现故障。

处理办法

执行 REINDEX 命令进行索引重建。

1. 以操作系统用户 gbase 登录主机。
2. 使用如下命令连接数据库。

```
gsql -d postgres -p 15400 -r
```

3. 重建索引。

- 如果进行 DDL 或 DML 操作时，因软硬件故障导致索引问题，请执行如下命令重建表索引。

```
REINDEX TABLE tablename;
```

- 如果错误中提示是 xxxx_index，其中 xxxx 代表用户表名。请执行如下命令之一重建表的索引。

```
REINDEX INDEX indexname;
```

或者

```
REINDEX TABLE tablename;
```

- 如果错误中提示 pg_xxxx_index，说明是系统表索引存在问题。请执行如下命令重建表索引。

```
REINDEX SYSTEM databasename;
```

8.2.25TPCC 高并发长稳运行因脏页刷盘效率导致性能下降

问题现象

TPCC 高并发长稳运行因脏页刷盘效率导致性能下降，具体表现为：初始性能较高，随着运行时间增加，数据库 tmpTotal 值下降，WalWriter 线程 CPU 占用 100%，其他 CPU 几乎没有负载，WDR 报告中，脏页刷盘等待时间占比最高。

原因分析

一般来说，问题原因可以通过查看进程状态、操作系统资源使用情况（CPU、IO 等）分析具体原因， 或者通过 WDR 报告对问题根因继续分析。 在该场景下，脏页刷新的效率较低。

处理方式

降低并发度或者调大 shared_buffers 参数。

调整脏页参数：在开启双写的场景下，可以调整 page_writer_sleep(下调)、max_io_capacity(上调)等参数，加快脏页淘汰效率。

更换高性能磁盘（NVME 等）。

数据库占用资源应与业务需求相吻合。对于高并发测试中，需要增加资源以保证数据库业务可用。

8.2.26共享内存泄露问题

问题现象

日志里出现如下错误：

```
This error usually means that PostgreSQL's request for a shared  memory segment
exceeded available memory or swap space,  or exceeded your kernel's SHMALL
parameter.
You can either  reduce the request size or reconfigure the kernel with larger
SHMALL.
```

原因分析

使用 free 命令查看内存使用情况，发现 shared 内存的确占用了很大一部分。

```
# free -g
              total        used        free      shared  buff/cache
available
Mem:             31           1           2          23          27           2
Swap:             3           3           0
```

使用 ipcs 命令进一步查看共享内存的使用情况，发现存在大量不再被进程使用但未回收的共享内存，即 nattch 为 0 的部分。

```
[root@pekpeuler00671 script]# ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 65536      gnome-init 777        16384      1          dest
0x00000000 131073     gnome-init 777        16384      1          dest
0x00000000 163842     gnome-init 777        3145728    2          dest
0x00000000 393219     gnome-init 600        524288     2          dest
0x00000000 425988     gnome-init 600        4194304    2          dest
0x00000000 458757     gnome-init 777        3145728    2          dest
0x00f42401 3604486    1001       600        4455342080 0
0x00f42402 14123015   1003       600        4457177088 0
0x00f42403 23592968   1005       600        4457177088 0
0x00f42404 33062921   1007       600        4457177088 0
0x00f42405 42532874   1009       600        4457177088 0
0x00f42406 52002827   1011       600        4457177088 0
0x00f42407 61472780   1013       600        4457177088 0
0x00f42408 70942733   1015       600        4457177088 0
0x00f42409 80412686   1017       600        4457177088 0
0x00f4240a 89882639   1019       600        4457177088 0
0x00f4240b 99352592   1021       600        4457177088 0
0x00f4240c 108822545  1023       600        4457177088 0
0x00f4240d 118292498  1025       600        4457177088 0
```

```
0x00f4240e 127762451  1027       600        4457177088 0
0x00f4240f 136904724  1029       600        4455342080 0
0x00f42410 146374677  1031       600        4457177088 0
0x00f42411 155844630  1033       600        4457177088 0
0x00f42412 165314583  1035       600        4457177088 0
0x00f42413 174784536  1037       600        4457177088 0
```

经过定位，这部分内存是由于使用 kill -9 命令来退出数据库进程，导致没有调用 IpcMemoryDelete 函数来清理共享内存，造成了内存泄漏。

处理方法

使用 ipcrm 释放无属主的共享内存，例如，释放 shmid 为 3604486 的共享内存，命令如下所示。

```
ipcrm -m shid3604486
```

8.2.27谓词下推引起的查询报错

问题现象

计划中出现谓词下推时，按照 SQL 标准中的查询执行顺序本不应该报错，结果执行出错。

```
postgres=# select * from tba;
 a
---
 -1
  2
(2 rows)

postgres=# select * from tbb;
 b
---
 -1
  1
(2 rows)

postgres=# select * from tba join tbb on a > b where b > 0 and sqrt(a) > 1;
ERROR: cannot table square root of a negative number
```

按照 SQL 执行标准流程: 1、执行 FROM 子句，能够保证所有数据满足 a > b。 2、执行 WHERE 子句中 b > 0，若结果为 true 则能够推导出 a > 0，并继续执行；若 false 则结束，后

面的条件被短路，不会执行。 3、执行 WHERE 子句中 sqrt(a) > 1。

但是实际却报错入参为负值。

原因分析

```
postgres=# explain (costs off) select * from tba join tbb on a > b where b > 0
and sqrt(a) > 1;
            QUERY PLAN
---------------------------------
Nest loop
  Join Filter: (a > b)
  -> Seq Scan on public.tba
       Filter: (sqrt(a) > 1)
  -> Materialize
       -> Seq Scan on public.tbb
            Filter: (b > 0)
(7 rows)
```

分析计划可知，原本 a > b, b > 0, sqrt(a) > 1 的三个条件，被拆分下推到了不同的算子之中，从而并非按顺序执行. 且当前的等价类推理仅支持等号推理，因此无法自动推理补充出 a > 0。 最终查询报错。

处理办法

谓词下推可以极大的提升查询性能，且此种短路、推导的特殊场景，在大多数数据库优化器下都没有过多考虑，因此建议修改查询语句，在相关的条件下手动添加 a > 0。

```
postgres=# select * from tba join tbb on a > b where b > 0 and a > 0 and sqrt(a) >
1;
 a | b
---+---
 2 | 1
(1 row)

postgres=# explain (costs off) select * from tba join tbb on a > b where b > 0
and a > 0 and sqrt(a) > 1;
              QUERY PLAN
-------------------------------------
Nest loop
  Join Filter: (a > b)
  -> Seq Scan on public.tba
       Filter: (a > 0 and sqrt(a) > 1)
  -> Materialize
```

```
      -> Seq Scan on public.tbb
           Filter: (b > 0)
(7 rows)
```

9 高危操作一览表

在产品的操作与维护阶段，进行日常各项操作时请严格遵守指导书。同时避免执行如下严禁、高危操作，见表 9-1、表 9-2。

表 9-1 禁用操作

操作名称	操作风险
严禁修改数据目录下文件名，权限，内容不能修改，不能删除内容。	导致数据库节点实例出现严重错误，并且无法修复。
严禁删除数据库系统表或系统表数据。	删除系统表将导致无法正常进行业务操作。

表 9-2 高危操作

操作分类	操作名称	操作风险	风险等级	规避措施	重大操作观察项目
数据库	直接在配置文件中手动修改端口号。	导致数据库启动不了或者连接不上。	▲▲ ▲▲ ▲	尽量使用工具修改，不要手动操作。	无。
	修改 pg_hba.conf 配置文件中的内容。	导致客户端连接不上。	▲▲ ▲▲ ▲	严格根据产品手册操作。	无
	手动修改 pg_xlog 的内容。	导致数据库无法启动，数据不一致。	▲▲ ▲▲ ▲	尽量使用工具修改，不要手动操作。	无
作业	使用 kill -9 终止作业进程。	导致作业占用的系统资源无法释放。	▲▲ ▲	登录数据库尽量使用 pg_terminate_backend，pg_cancel_backend	观察资源使用情况。

				操作终止作业，或使用 Ctrl+C 终止作业进程。	

10 日志参考

10.1 日志类型简介

在数据库运行过程中，会出现大量日志，既有保证数据库安全可靠的 WAL 日志（预写式日志，也称为 Xlog），也有用于数据库日常维护的运行和操作日志等。在数据库发生故障时，可以参考这些日志进行问题定位和数据库恢复的操作。

日志类型

日志类型的详细说明，请参见下表。

表 10-1 日志类型

类型	说明
系统日志	数据库系统进程运行时产生的日志，记录系统进程的异常信息。
操作日志	通过客户端工具（例如 gs_guc）操作数据库时产生的日志。
Trace 日志	通过 gstrace 工具指定启动 trace 功能的 DN 实例后，会记录大量的 Trace 日志。这些日志可以用来分析数据库的异常信息。
黑匣子日志	数据库系统崩溃的时候，通过故障现场堆、栈信息可以分析出故障发生时的进程上下文，方便故障定位。黑匣子具有在系统崩溃时，dump 出进程和线程的堆、栈、寄存器信息的功能。
审计日志	开启数据库审计功能后，将数据库用户的某些操作记录在日志中，这些日志称为审计日志。
WAL 日志	又称为 REDO 日志，在数据库异常损坏时，可以利用 WAL 日志进行恢复。由于 WAL 日志的重要性，所以需要经常备份这些日志。
性能日志	数据库系统在运行时检测物理资源的运行状态的日志，在对外部资源进行访问时的性能检测。

10.2 系统日志

GBase 8c 数据库运行时，集群和节点安装部署时产生的日志统称为系统日志。如果在数据库运行时发生故障，可以通过这些系统日志及时定位故障发生的原因，根据日志内容制定恢复 GBase 8c 的方法。

日志文件存储路径

- 数据库节点的运行日志放在“/var/log/gbase/gbase/pg_log”中各自对应的目录下。
- OM GBase 8c 安装卸载时产生的日志放在“/var/log/gbase/gbase/om”目录下。

日志文件命名格式

- 数据库节点运行日志的命名规则：

```
postgresql-创建时间.log
```

默认情况下，每日 0 点或者日志文件大于 16MB 或者数据库实例（数据库节点）重新启动后，会生成新的日志文件。

- CM 的运行日志的命名规则：
  - cm_agent 的日志：

```
cm_agent-创建时间.log/cm_agent-创建时间-current.log/system_call-创建时间.log/system_call-创建时间-current.log
```

  - cm_server 的日志：

```
cm_server-创建时间.log/cm_server-创建时间-current.log/key_event-创建时间.log/key_event-创建时间-current.log
```

  - om_monitor 的日志：

```
om_monitor-创建时间.log/om_monitor-创建时间-current.log
```

其中，不带 current 标识符的文件是历史日志文件，带 current 标识符的文件是当前日志文件。最初调用进程时，进程会先创建一个带 current 标识符的日志文件，当该日志文件的大小超过 16MB 时，会将当前日志文件重命名为历史日志文件，并以当前时间生成新的当前日志文件。

日志内容说明

- 数据库节点每一行日志内容的默认格式：

```
日期+时间+时区+用户名称+数据库名称+会话 ID+日志级别+日志内容
```

- cm_agent、cm_server、om_monitor 每一行日志内容的默认格式：

```
时间+时区+会话 ID+日志内容
```

SYSTEM_CALL 系统调用日志，记录 CM_AGENT 在运行过程中调用工具命令的情况。

- key_event 每一行日志内容的默认格式:

```
时间+线程号+线程名：关键事件类型+仲裁对象实例 ID+仲裁细节
```

10.3 操作日志

数据库管理员使用工具操作数据库时，或工具被数据库调用时，会产生操作日志。如果 GBase 8c 发生故障，可以通过这些日志信息跟踪用户对数据库进行了哪些操作，重现故障场景。

日志文件存储路径

默认在“$GAUSSLOG/bin”目录下，如果环境变量$GAUSSLOG 不存在或者变量值为空，则工具日志信息不会记录到对应的工具日志文件中，日志信息只会打印到屏幕上。

其中，环境变量$GAUSSLOG 默认为“/var/log/gbase/gbase”。

日志文件命名格式

日志文件命名格式为：

```
工具名-日志创建时间.log
工具名-日志创建时间-current.log
```

其中，“工具名-日志创建时间.log”是历史日志文件，“工具名-日志创建时间- current.log”是当前日志文件。

如果日志大小超过 16MB，在下一次调用该工具时，会重命名当前日志文件为历史日志文件，并以当前时间生成新的当前日志文件。例如，将“gs_guc-2015-01-16_183728-current.log”

重命名为“gs_guc-2015-01-16_183728.log”，然后重新生成“gs_guc-2015-01-17_142216-current.log”。

维护建议

建议定时对过期的日志文件进行转储，以避免发生大量日志占用太多的磁盘空间、重要日志丢失等问题。

10.4 审计日志

审计功能开启时会不断产生大量的审计日志，占用磁盘空间。用户可以根据磁盘空间的大小设置审计日志维护策略。

关于如何设置审计日志维护策略请参见《GBase 8c V5_5.0.0_数据库管理指南》中“管理数据库安全 > 设置数据库审计 > 维护审计日志”章节。

预写式日志 WAL（Write Ahead Log，也称为 Xlog）是指如果要修改数据文件，必须是在这些修改操作已经记录到日志文件之后才能进行修改，即在描述这些变化的日志记录刷新到永久存储器之后。在系统崩溃时，可以使用 WAL 日志对 GBase 8c 进行恢复操作。

日志文件存储路径

以一个数据库节点为例，默认在“/var/log/gbase/gbase/pg_audit”目录下。

日志文件命名格式

日志文件以段文件的形式存储的，每个段为 16MB，并分割成若干页，每页 8KB。对 WAL 日志的命名说明如下：一个段文件的名称由 24 个十六进制组成，分为三个部分，每个部分由 8 个十六进制字符组成。第一部分表示时间线，第二部分表示日志文件标号，第三部分表示日志文件的段标号。时间线由 1 开始，日志文件标号和日志文件的段标号由 0 开始。

例如，系统中的第一个事务日志文件是 000000010000000000000000。

说明

这些数字一般情况下是顺序增长使用的（要把所有可用数字都用光也需要非常长的时间），但也存在循环使用的情况。

日志内容说明

WAL 日志的内容取决于记录事务的类型，在系统崩溃时可以利用 WAL 日志进行恢复。默认配置下，GBase 8c 每次启动时会先读取 WAL 日志进行恢复。

维护建议

WAL 日志对数据库异常恢复有重要的作用，建议定期对 WAL 日志进行备份。

10.5 性能日志

性能日志主要关注外部资源的访问性能问题。性能日志指的是数据库系统在运行时检测物理资源的运行状态的日志，在对外部资源进行访问时的性能检测，包括磁盘、Hadoop GBase 8c等外部资源的访问检测信息。在出现性能问题时，可以借助性能日志及时的定位问题发生的原因，能极大地提升问题解决效率。

日志文件存储路径

数据库节点的性能日志目录在“$GAUSSLOG/gs_profile”中各自对应的目录下。日志文件命名格式：

- 数据库节点的性能日志的命名规则：

```
postgresql-创建时间.prf
```

默认情况下，每日 0 点或者日志文件大于 20MB 或者数据库实例（数据库节点）重新启动后，会生成新的日志文件。

日志内容说明

- 数据库节点每一行日志内容的默认格式：

```
主机名称+日期+时间+实例名称+线程号+日志内容
```

GBASE®

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.

微信二维码

技术支持热线：400-013-9696