第 3 章

公共组件源码解析

在数据库组件中，一些组件是专用的，如词法解析只用于 SQL 引擎，而另外一些组件是公共的，用于整个数据库系统。openGauss 的公共组件包括系统表、数据库初始化、多线程架构、线程池、内存管理、多维监控和模拟信号机制等，每个组件实现一个独立的功能。本章对公共组件的源代码实现进行介绍。

3.1 系统表

系统表又称为数据字典或者元数据，存储管理数据库对象的定义信息，如表、索引、触发器等。用户可通过系统表查询用户定义的具体对象信息，如表的每个字段类型。因为 openGauss 支持一个实例管理多个数据库，所以系统表分为实例级别的系统表和数据库级别的系统表。实例级别的系统表在一个实例管理的多个数据库之间共享，整个实例只有一份，这些系统表为 pg_authid、pg_auth_members、pg_database、pg_db_role_setting、pg_tablespace、pg_shdepend、pg_shdescription、pg_shseclabel。数据库级别的系统表如 pg_class、pg_depend、pg_index、pg_attribute 等，每个数据库各有一份。

3.1.1 系统表的定义

openGauss 系统表定义全部在 src/include/catalog 目录下，每个头文件就是一个系统表的定义，如 pg_database.h 就是对 pg_database 的定义。在 pg_database.h 中，主要包括 pg_database 的表 OID(Object Identifier，对象标识符)、类型 OID、结构体定义、字段个数和每个字段 ID(Identifier，标识符)枚举值、数据库初始化默认值。下面是代码及其具体解释：

```
/* pg_database 本身也是一张表，DatabaseRelationId 表示 pg_database 在系统表 pg_class 中的 OID
为 1262(pg_class 系统表中保存的是表的定义信息) */
#define DatabaseRelationId 1262
/* pg_database 本身也是一个结构类型，DatabaseRelation_Rowtype_Id 表示 pg_database 在系统表
pg_type 中的 OID 为 1248 */
#define DatabaseRelation_Rowtype_Id 1248
/* BKI_SHARED_RELATION 表示 pg_database 是实例级别的系统表 */
CATALOG(pg_database,1262) BKI_SHARED_RELATION BKI_ROWTYPE_OID(1248) BKI_SCHEMA_MACRO
{
    NameData    datname;            /* 数据库名称 */
    Oid         datdba;             /* 数据库拥有者 */
    int4        encoding;           /* 字符集编码 */
    NameData    datcollate;         /* LC_COLLATE 设置值 */
```

```
    NameData    datctype;             /* LC_CTYPE 设置值 */
    bool        datistemplate;        /* 是否允许作为模板数据库 */
    bool        datallowconn;         /* 是否允许连接 */
    int4        datconnlimit;         /* 最大连接数 */
    Oid         datlastsysoid;        /* 系统 OID 最大值 */
    ShortTransactionId datfrozenxid;      /* 冻结事务 ID,所有小于这个值的事务 ID 已经冷冻.
为了兼容原来的版本,使用 32 位事务 ID */
    Oid         dattablespace;        /* 数据库的默认表空间 */
    NameData    datcompatibility;    /* 数据库兼容模式,比如除 0 是报错还是当作正常处理 */
#ifdef CATALOG_VARLEN                 /* 下面字段是变长字段 */
    aclitem     datacl[1];            /* 访问权限 */
#endif
    TransactionId datfrozenxid64;     /* 冷冻事务 ID,64 - bit(比特)事务 ID */
} FormData_pg_database;
```

CATALOG 的宏定义代码为

```
#define CATALOG(name,oid)  typedef struct CppConcat(FormData_,name)
```

因此,CATALOG(pg_database,1262)就是对结构体 FormData_pg_database 的定义。之所以采用 CATALOG,是因为这个格式是和 BKI(Backend Interface,后端接口)脚本约定的格式,BKI 脚本根据这个格式生成数据库的建表脚本。

接下来是数据库对象字段总数和每个字段 ID 的值的定义代码,定义这些值的目的主要是代码访问数据库对象时清晰,方便维护,避免魔鬼数字(魔鬼数字指在代码中没有具体含义的数字、字符串。魔鬼数字会影响代码可读性,使读者无法理解看到的数字对应的含义,从而难以理解程序的意图)。

```
#define Natts_pg_database                14
#define Anum_pg_database_datname         1
#define Anum_pg_database_datdba          2
#define Anum_pg_database_encoding        3
#define Anum_pg_database_datcollate      4
#define Anum_pg_database_datctype        5
#define Anum_pg_database_datistemplate   6
#define Anum_pg_database_datallowconn    7
#define Anum_pg_database_datconnlimit    8
#define Anum_pg_database_datlastsysoid   9
#define Anum_pg_database_datfrozenxid    10
#define Anum_pg_database_dattablespace   11
#define Anum_pg_database_compatibility   12
#define Anum_pg_database_datacl          13
#define Anum_pg_database_datfrozenxid64  14
```

最后是创建数据库时的默认数值。代码中的值表示默认创建 template1 数据库,DATA 的字段值和数据库结构体的值一一对应,对应代码如下:

```
DATA(insert OID = 1 ( template1 PGUID ENCODING "LC_COLLATE" "LC_CTYPE" t t -1 0 0 1663 "DB_
COMPATIBILITY" _null_ 3));
SHDESCR("default template for new databases");
```

```
#define TemplateDbOid          1
#define DEFAULT_DATABASE "postgres"
```

系统表头文件的内容和格式基本类似。

3.1.2　系统表的访问

定义系统表后，接下来介绍数据库在运行过程中对系统表的访问。openGauss 对系统表的访问主要通过 syscache 机制。syscache 机制是一个通用的机制，主要对系统表的数据进行缓存，提升系统表数据的访问速度，详细细节参照具体章节描述。这里主要介绍与 pg_database 相关的部分。pg_database 在枚举类型 enum SysCacheIdentifier 中定义的枚举值有一个：DATABASEOID，表示根据数据库 OID 访问 pg_database 系统表，同时需要把 pg_database 系统表访问模式添加到“struct cachedesc cacheinfo”中。与 pg_database 相关的代码如下：

```
{DatabaseRelationId,            /* DATABASEOID */
    DatabaseOidIndexId,
    1,
    {ObjectIdAttributeNumber,0,0,0},
    4},
```

这几个值与 cachedesc 结构体的字段对应，表示 pg_database 表的 OID 值、索引的 OID 值、搜索时有 1 个 key 字段、搜索 key 字段 ID 为 ObjectIdAttributeNumber、初始化为 4 个哈希桶。相关的代码如下：

```
struct cachedesc {
    Oid reloid;             /* 缓存的表的 OID */
    Oid indoid;             /* 缓存数据的索引 OID */
    int nkeys;              /* 缓存搜索的 key 的个数 */
    int key[4];             /* key 属性的编号 */
    int nbuckets;           /* 缓存哈希桶的个数 */
};
```

系统表的定义和访问主要逻辑如上所述。与 pg_database 相关的 SQL 命令是 ALTER DATABASE、CREATE DATABASE、DROP DATABASE，这些命令执行的结果是把数据库相关的信息存储到 pg_database 系统表中。

其他系统表的逻辑与 pg_database 相似，不再重复。

3.2　数据库初始化

数据库正常启动时需要指定数据目录，数据目录中包含了系统表的初始化数据。数据库初始化的过程会生成这些初始系统表数据文件，该过程由 initdb 和 openGauss 进程配合生成。initdb 控制执行过程，创建目录和基本的配置文件；openGauss 进程负责系统表的初

始化。initdb 通过 PG_CMD_OPEN 宏启动 openGauss 进程，同时打开一个管道流，然后通过解析系统表文件中的 SQL 命令，并把命令通过 PG_CMD_PUTS 宏的管道流发给 openGauss 进程，最后通过 PG_CMD_CLOSE 宏关闭管道流。PG_CMD_OPEN 宏是系统函数 popen 的封装宏，PG_CMD_PUTS 宏是系统函数 fputs 的封装宏，PG_CMD_CLOSE 宏是系统函数 pclose 的封装宏。初始化交互过程如图 3-1 所示。

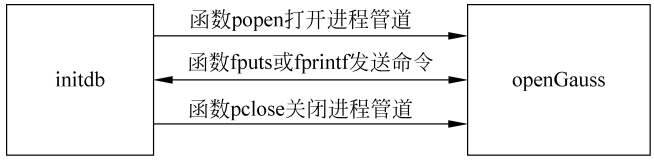

图 3-1　初始化交互过程

initdb 在创建 template1 模板数据库时，命令参数指定了“snprintf_s(cmd, sizeof(cmd), sizeof(cmd)-1,"\"%s\" --boot -x1 %s %s", backend_exec, boot_options, talkargs);”，其中“--boot”表示 openGauss 进程以一个特殊的 bootstrap 模式运行。在其他初始化系统表时，initdb 命令参数指定了“snprintf_s(cmd, sizeof(cmd), sizeof(cmd) -1,"\"%s\" %s template1 >%s", backend_exec, backend_options, DEVNULL); ”，“static const char * backend_options = "--single ""表示 openGauss 进程以单用户模式运行。

下面以 setup_schema 函数为例详细介绍这个过程，相关代码如下：

```
static void setup_schema(void)
{
    PG_CMD_DECL;
    char ** line;
    char ** lines;
    int nRet = 0;
    char * buf_features = NULL;
    fputs(_("creating information schema ... "),stdout);
    (void)fflush(stdout);
    lines = readfile(info_schema_file);
    /*
     * 使用 -j 避免在 information_schema.sql 反斜杠处理
     */
    nRet = snprintf_s(
        cmd,sizeof(cmd),sizeof(cmd) - 1,"\"%s\" %s -j template1 >%s",backend_exec,
backend_options,DEVNULL);
    securec_check_ss_c(nRet,"\0","\0");
    PG_CMD_OPEN;
    for (line = lines; *line != NULL; line++) {
        PG_CMD_PUTS(*line);
        FREE_AND_RESET(*line);
    }
    FREE_AND_RESET(lines);
    PG_CMD_CLOSE;
    nRet = snprintf_s(
```

```
        cmd,sizeof(cmd),sizeof(cmd) - 1,"\"%s\" %s template1 > %s",backend_exec,backend
_options,DEVNULL);
    securec_check_ss_c(nRet,"\0","\0");
    PG_CMD_OPEN;
    PG_CMD_PRINTF1("UPDATE information_schema.sql_implementation_info "
                   " SET character_value = '%s'"
                   " WHERE implementation_info_name = 'DBMS VERSION';\n",
        infoversion);

    buf_features = escape_quotes(features_file);
    PG_CMD_PRINTF1("COPY information_schema.sql_features "
                   " (feature_id,feature_name,sub_feature_id,"
                   " sub_feature_name,is_supported,comments) "
                   " FROM E'%s';\n",
        buf_features);
    FREE_AND_RESET(buf_features);
    PG_CMD_CLOSE;
    check_ok();
}
```

在这个函数中，PG_CMD_DECL 是一个变量定义宏，通过语句“char cmd[MAXPGPATH]”和“FILE* cmdfd = NULL”定义了两个变量。这样的作用是代码格式统一，阅读方便。

语句“readfile(info_schema_file)”表示读取 info_schema_file 文件，这个文件中存放了系统表初始化的 SQL 命令。

语句“snprintf_s(cmd,sizeof(cmd),sizeof(cmd)-1,"\"%s\" %s -j template1 >%s",backend_exec,backend_options,DEVNULL)”是格式化 openGauss 后台进程的命令。语句“PG_CMD_OPEN”是以 popen 的方式运行 cmd 命令，启动 openGauss 进程。

语句“for (line = lines; *line != NULL; line++)”表示遍历 info_schema_file 文件中的每条 SQL 命令，宏 PG_CMD_PUTS 把每个 SQL 命令发送给 openGauss 进程执行。

整个文件执行完毕，调用宏 PG_CMD_CLOSE 停止进程，关闭管道。setup_schema 函数的后面代码处理是类似的，只是 SQL 命令是函数内生成的，使用宏 PG_CMD_PRINTF1 写入管道，发给 openGauss 进程。

setup_sysviews、setup_dictionary、setup_privileges 等其他系统对象初始化函数过程都是类似的，不再重复描述。

initdb 的整个初始化过程如下。

(1) 对命令行参数进行解析。

(2) 查找 openGauss 程序，设置 $PGDATA、$PGPATH 等环境变量。设置数据库初始化原始文件，这些文件在 shell 命令 make install 执行安装后，默认都在“openGauss-server/dest/share/postgresql”目录下。

(3) 数据库本地初始化，locale 默认初始化为 en_US.UTF-8，数据库编码默认初始化为 UTF8，文本搜索默认初始化为 English。

(4) 检查数据库数据目录 pg_data 是否为空,是否需要创建,权限是否正确。

(5) 创建 subdirs 变量指定的子目录。

(6) 初始化 conf 配置文件。

(7) 创建 template1 数据库 bootstrap_template1。这一步需要启动后台 openGauss 进程执行数据库的 SQL 语句,创建系统表。bootstrap_template1 这个函数主要是读取 bki 文件中的 SQL 语句,发送到 openGauss 进程去执行,主要功能是创建系统表。语句 create pg_type 表示创建 pg_type 系统表,语句 INSERT OID 表示插入这个系统表的默认数据。这里的语法是专门为 initdb 定制的 bootstrap 解析语法,不是正式的 SQL 语法,语法文件也是单独的,可参照"openGauss-server\src\gausskernel\bootstrap"目录下的 bootscanner.l 和 bootparse.y 文件。pg_type 系统对象在 initdb 初始化中的 bootstrap 语法相关代码如下,在初始化时就是解析下面语法格式完成 pg_type 系统对象的创建。

```
create pg_type 1247 bootstrap rowtype_oid 71
 (
 typname = name ,
 typnamespace = oid ,
 typowner = oid ,
 typlen = int2 ,
 typbyval = bool ,
 typtype = char ,
 typcategory = char ,
 typispreferred = bool ,
 typisdefined = bool ,
 typdelim = char ,
 typrelid = oid ,
 typelem = oid ,
 typarray = oid ,
 typinput = regproc ,
 typoutput = regproc ,
 typreceive = regproc ,
 typsend = regproc ,
 typmodin = regproc ,
 typmodout = regproc ,
 typanalyze = regproc ,
 typalign = char ,
 typstorage = char ,
 typnotnull = bool ,
 typbasetype = oid ,
 typtypmod = int4 ,
 typndims = int4 ,
 typcollation = oid ,
 typdefaultbin = pg_node_tree ,
 typdefault = text ,
 typacl = aclitem[]
 )
INSERT OID = 16 ( bool 11 10 1 t b B t t \054 0 0 1000 boolin boolout boolrecv boolsend --- c p
f 0 -1 0 0 _null_ _null_ _null_ )
```

```
INSERT OID = 17 ( bytea 11 10 -1 f b U f t \054 0 0 1001 byteain byteaout bytearecv byteasend -
-- i x f 0 -1 0 0 _null_ _null_ _null_ )
…
close pg_type
```

(8) 使用 setup_auth 函数初始化 pg_authid 权限。该函数执行的 SQL 语句是在函数内静态定义“static const char * pg_authid_setup[]”。

(9) 使用 setup_depend 函数创建系统表依赖关系。该函数执行的 SQL 语句是在函数内静态定义“static const char * pg_depend_setup[]”。

(10) 使用 load_plpgsql 函数加载 plpgsql 扩展组件。该函数只执行一条 SQL 语句“CREATE EXTENSION plpgsql;”。

(11) 使用 setup_sysviews 函数创建系统视图。该函数会读取 system_views.sql 文件中的 SQL 语句,发送到 openGauss 去执行,主要功能是创建系统视图。

(12) 使用 setup_perfviews 函数创建性能视图。该函数会读取 performance_views.sql 文件中的 SQL 语句,发送到 openGauss 去执行,主要功能是创建性能视图。

(13) 使用 setup_conversion 函数创建编码转换。该函数会读取 conversion_create.sql 文件中的 SQL 语句,发送到 openGauss 去执行,主要功能是创建编码转换函数。

(14) 使用 setup_dictionary 函数创建词干数据字典。该函数会读取 snowball_create.sql 文件中的 SQL 语句,发送到 openGauss 去执行,主要功能是创建文本搜索函数。

(15) 使用 setup_privileges 函数设置权限。setup_privileges 函数通过 xstrdu 复制 SQL 常量字符串到一个动态数组内,然后遍历执行指定的 SQL 语句。

(16) 使用 load_supported_extension 函数加载外表。该函数执行相应扩展组件的 CREATE EXTENSION 语句。

(17) 使用 setup_update 函数更新系统表。该函数执行语句 COPY pg_cast_oid.txt 到数据库中,主要功能是创建类型强制转换处理函数。

(18) 对 template1 进行垃圾数据清理,即执行三个 SQL 语句“ANALYZE;”“VACUUM FULL;”“VACUUM FREEZE;”。

(19) 创建 template0 数据库,即复制 template1 到 template0。

(20) 创建默认数据库,即复制 template1 到默认数据库。

(21) 对 template0、template1、默认数据库进行垃圾数据清理和事务 ID 冻结。

3.3　多线程架构

openGauss 内核源自 PostgreSQL,但在架构上进行了大量改造,其中一个调整就是将多进程架构修改为多线程架构。openGauss 在启动后只有一个进程,后台任务都是以一个进程中的线程来运行的。对于客户端的新连接,在非线程池模式下也是以启动一个业务线程来处理的。在多线程架构下更容易实现多个线程资源的共享,如并行查询、线程池等。

3.3.1 openGauss 主要线程

openGauss 的后台线程是不对等的，其中 Postmaster 是主线程，其他线程都是它创建出来的。openGauss 后台线程的功能介绍如表 3-1 所示。

表 3-1 openGauss 后台线程的功能

后台线程	功能介绍
Postmaster	openGauss 数据库主线程。主要有两个功能：一是对连接进行监听，接收新的连接；二是监控所有子线程的状态，并根据子线程退出状态进行处理，如果线程是 FATAL 退出，则重新拉起子线程，如果线程是 PANIC 退出，则进行整个数据库重新初始化，保证数据库的正常运行
Startup	数据库启动线程。数据库启动时 Postmaster 主线程拉起的第一个子线程，主要完成数据库的日志 REDO(重做)操作，进行数据库的恢复。日志 REDO 操作结束，数据库完成恢复后，如果不是备机，Startup 线程就退出了。如果是备机，那么 Startup 线程一直在运行，REDO 备机接收到新的日志
Bgwriter	后台数据写线程。周期性地把数据库数据缓冲区的内容同步到磁盘上
Checkpointer	检查点线程。进行检查点操作，完成数据库的周期性检查点和执行检查点命令
Walwriter	后台 WAL 写线程。主要功能是周期性地把日志缓冲区的内容同步到磁盘上
Stat	数据库运行信息统计收集线程。主要功能是收集各个线程操作数据库的统计信息，进行汇总后写入数据库的统计文件中，供查询优化分析和垃圾清理使用
Sysloger	运行日志写线程。主要功能是把各个线程的运行日志信息写到运行日志文件中
Vacuum Launch	垃圾清理启动线程。主要有两个功能：一是通知 Postmaster 启动一个垃圾清理线程；二是平衡各个垃圾清理线程的负载
Vacuum worker	垃圾清理线程。主要功能是对 openGauss 的垃圾数据进行清理
Arch	日志归档线程。主要功能是完成归档操作，把在线日志复制到归档目录
Postgres	服务线程。在非线程池模式下，每个客户端连接对应一个服务线程，主要功能是接收客户端的操作请求，代表客户端在服务器完成数据库操作

3.3.2 线程间通信

openGauss 后台线程之间紧密配合，共同完成数据库的数据处理任务。这些后台线程之间需要交换信息来协调彼此的行为。openGauss 多线程通信使用了原来的 PostgreSQL 的多线程通信方式，具体如表 3-2 所示。

表 3-2 多线程通信方式

通信方式	说　明
共享内存	在数据库初始化时，Postmaster 线程通过 OS(操作系统)申请一块大的共享内存，并完成初始化工作。openGauss 使用到的所有共享内存都是这块内存的一部分。线程之间的一些信息交换就是通过共享内存完成的，共享内存的访问需要加锁保护

续表

通信方式	说　　明
信号	对于一些紧急任务的处理，openGauss 使用信号通知作为线程间通信的手段，因为信号可以中断处理线程当前的任务，立即响应信号对应的任务
TCP	客户端连接数据库服务器时，一般使用 TCP 进行通信
UNIX 域套接字协议	如果是本地客户端，即客户端和服务器在同一台机器上，并且是 UNIX 操作系统，可以使用 UNIX 域套接字协议建立客户端和服务器进程的通信
UDP	UDP(User Datagram Protocol，用户数据报协议)是不可靠协议，主要用于后台线程向统计线程发送统计信息时使用
管道	管道可以是双向的，也可以是单向的。在 openGauss 中，主要使用了单向管道，用在后台线程向运行日志守护线程发送运行日志信息时使用
文件	主要用于一些不太重要的场合，并且通信量比较大。在 openGauss 中，主要用在统计线程汇总统计信息，写到统计文件，供垃圾清理线程和后台服务器线程成本优化使用
全局变量	一种线程间共享信息的机制。openGauss 对原来的 PostgreSQL 中进程内的全局变量添加 THR_LOCAL 定义为线程的局部变量，避免线程之间误用

3.3.3　线程初始化流程

下面介绍线程的初始化流程。首先介绍 openGauss 进程的启动。openGauss 进程的主函数入口在“\openGauss-server\src\gausskernel\process\main\main.cpp”文件中。在 main.cpp 文件中，主要完成实例 Context(上下文)的初始化、本地化设置，根据 main.cpp 文件的入口参数调用 BootStrapProcessMain 函数、GucInfoMain 函数、PostgresMain 函数和 PostmasterMain 函数。BootStrapProcessMain 函数和 PostgresMain 函数是在 initdb 场景下初始化数据库使用的。GucInfoMain 函数的作用是显示 GUC(Grand Unified Configuration，配置参数，在数据库中指的是运行参数)参数信息。正常的数据库启动会进入 PostmasterMain 函数。下面对这几个函数进行更详细的介绍。

(1) 进行 Postmaster 的 Context 初始化，初始化 GUC 参数，解析命令行参数。

(2) 调用 StreamServerPort 函数启动服务器监听和双机监听(如果配置了双机)，调用 reset_shared 函数初始化共享内存和 LWLock 锁，调用 gs_signal_monitor_startup 函数注册信号处理线程，调用 InitPostmasterDeathWatchHandle 函数注册 Postmaster 死亡监控管道，把 openGauss 进程信息写入 pid_file 文件中，调用 gspqsignal 函数注册 Postmaster 的信号处理函数。

(3) 根据配置初始化黑匣子，调用 pgstat_init 函数初始化统计信息传递使用的 UDP 套接字通信，调用 InitializeWorkloadManager 函数初始化负载管理器，调用 InitUniqueSQL 函数初始化 UniqueSQL，调用 SysLogger_Start 函数初始化运行日志的通信管道和 SYSLOGGER 线程，调用 load_hba 函数加载 hba 鉴权文件。

(4) 调用 initialize_util_thread 函数启动 STARTUP 线程，调用 ServerLoop 函数进入一个周期循环。在 ServerLoop 函数的周期循环中，进行客户端请求监听，如果有客户端连

接请求，在非线程池模式下，则调用 BackendStartup 函数创建一个后台线程 worker 处理客户请求。在线程池模式下，把新的链接加入一个线程池组中。在 ServerLoop 函数的周期循环中，检查其他线程的运行状态。如果数据库是第一次启动，则调用 initialize_util_thread 函数启动其他后台线程。如果有后台线程 FATAL 级别错误退出，则调用 initialize_util_thread 函数重新启动该线程，如果是 PANIC 级别错误退出，则整个实例进行重新初始化。

PostmasterMain 完成了线程之间的通信初始化和线程的启动，无论是后台线程的启动函数 initialize_util_thread 还是工作线程的启动函数 initialize_worker_thread，最后都是调用 initialize_thread 函数完成的线程启动。下面对 initialize_thread 函数进行介绍。

initialize_thread 函数调用 gs_thread_create 函数创建线程，调用 InternalThreadFunc 函数处理线程，它的相关代码如下所示：

```
ThreadId initialize_thread(ThreadArg * thr_argv)
{
    gs_thread_t thread;
    if (0 != gs_thread_create(&thread,InternalThreadFunc,1,(void * )thr_argv)) {
        gs_thread_release_args_slot(thr_argv);
        return InvalidTid;
    }
    return gs_thread_id(thread);
}
```

InternalThreadFunc 函数的代码如下。该函数根据角色调用 GetThreadEntry 函数，GetThreadEntry 函数直接以角色为下标，返回对应 GaussdbThreadEntryGate 数组对应的元素。数组的元素是处理具体任务的回调函数指针，指针指向的函数为 GaussDbThreadMain。相关代码如下：

```
static void * InternalThreadFunc(void * args)
{
    knl_thread_arg * thr_argv = (knl_thread_arg * )args;
    gs_thread_exit((GetThreadEntry(thr_argv->role))(thr_argv));
    return (void * )NULL;
}
GaussdbThreadEntry GetThreadEntry(knl_thread_role role)
{
    Assert(role > MASTER && role < THREAD_ENTRY_BOUND);
    return GaussdbThreadEntryGate[role];
}
static GaussdbThreadEntry GaussdbThreadEntryGate[ ] = {GaussDbThreadMain<MASTER>,
    GaussDbThreadMain<WORKER>,
    GaussDbThreadMain<THREADPOOL_WORKER>,
    GaussDbThreadMain<THREADPOOL_LISTENER>,
  …};
```

在 GaussDbThreadMain 函数中，首先初始化线程基本信息、Context 和信号处理函数，然后根据 thread_role 角色的不同调用不同角色的处理函数，进入各个线程的 MAIN 函数，

如 GaussDbAuxiliaryThreadMain 函数、AutoVacLauncherMain 函数、WLMProcessThreadMain 函数等。其中，GaussDbAuxiliaryThreadMain 函数是后台辅助线程处理函数。该函数的处理也类似 GaussDbThreadMain 函数，根据 thread_role 角色的不同调用不同角色的处理函数，进入各个线程的 MAIN 函数，如 StartupProcessMain 函数、CheckpointerMain 函数、WalWriterMain 函数、walrcvWriterMain 函数等。

总结上面整个过程，openGauss 多线程架构主要包括 3 个方面：

（1）多线程之间的通信，由主线程在初始化阶段完成；

（2）多线程的启动，由主线程创建各个角色线程，调用不同角色的处理函数完成；

（3）主线程负责监控各个线程的运行、异常退出和重新拉起。

3.4　线程池技术

openGauss 在多线程架构的基础上实现了线程池。线程池机制实现了会话和处理线程分离，在大并发连接的情况下仍然能够保证系统有很好的 SLA 响应。另外，不同的线程组可绑到不同的 NUMA（Non-Uniform Memory Access，非一致性内存访问）核上，天然匹配 NUMA 化的 CPU 架构，从而提升 openGauss 的整体性能。

3.4.1　线程池原理

openGauss 线程池原理如图 3-2 所示，图 3-2 中的主要对象如表 3-3 所示。

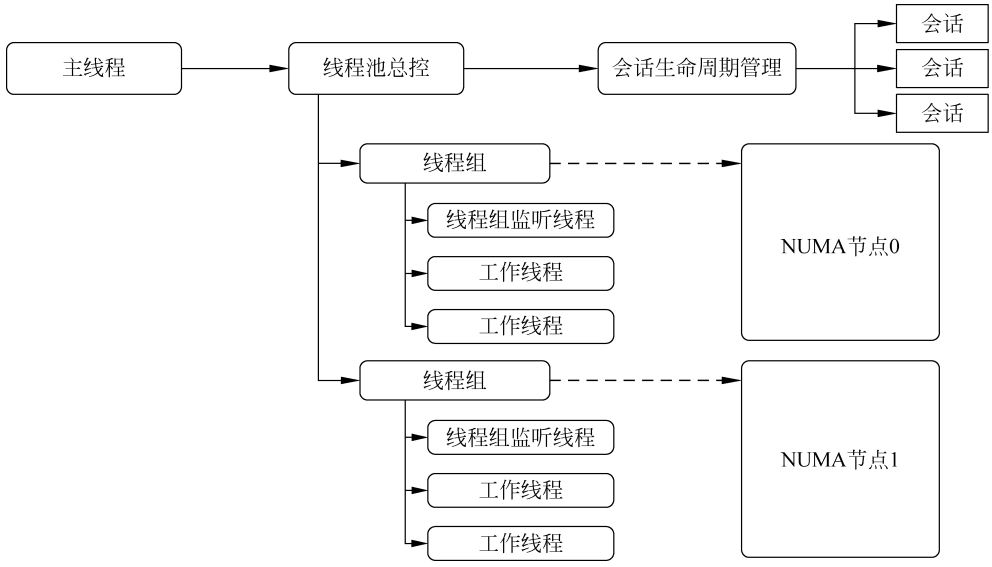

图 3-2　线程池原理

表 3-3 线程池对象

对 象	说 明
Postmaster	主线程。负责监听客户端发出的请求
ThreadPoolControler	线程池总控。负责线程池的初始化和资源管理
ThreadSessionControler	会话生命周期管理
ThreadPoolGroup	线程组。可以定义灵活的线程数量和绑核策略
ThreadPoolListener	线程组监听线程。负责事件的分发和管理
ThreadPoolWorker	工作线程
session	客户端连接的一个会话
NUMA NODE	NUMA 节点。表示一个线程组在 NUMA 结构下可以映射到一个 NUMA 节点上

这些对象相互配合实现了线程池机制,它们的主要交互过程如下。

(1) 客户端向数据库发起连接请求,Postmaster 线程接收到连接请求并被唤醒。Postmaster 线程创建该连接对应的 socket(套接字,用于描述 IP 地址和端口,是一个通信链的句柄),调用 ThreadPoolControler 函数创建会话(session)数据结构。ThreadPoolControler 函数遍历当前所有的 Thread Group(线程组),找到当前活跃会话数量最少的 Thread Group,并把最新的会话分发给该 Thread Group,加入该 Thread Group 的 epoll(Linux 内核为处理大批量句柄而改进的 poll(轮询),能显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率)列表中。

(2) Thread Group 的 listener 线程负责监听 epoll 列表中所有的客户连接。

(3) 客户端发起任务请求,listener 线程被唤醒。listener 线程检查当前的 Thread Group 是否有空闲 worker 线程;如果有,则把当前会话分配给该 worker 线程,并唤醒该 worker 线程;如果没有,则把该会话放在等待队列中。

(4) worker 线程被唤醒后,读取客户端连接上的请求,执行相应请求,并返回请求结果。在一次事务结束(提交、回滚)或者事务超时退出时,worker 线程的一次任务完成。worker 线程将会话返回 listener 线程,listener 线程继续等待该会话的下一次请求。worker 线程返还会话后,检查会话等待队列;如果存在等待响应请求的会话,则直接从该队列中取出新的会话并继续工作;如果没有等待响应的会话,则将自身标记为 free(空闲)状态,等待 listener 线程唤醒。

(5) 客户端断开连接时,worker 线程被唤醒,关闭连接,同时清理会话相关结构,释放内存和 fd(文件句柄)等资源。

(6) 如果 worker 线程 FATAL 级别错误退出,退出时 worker 线程会从 worker 队列中注销掉。此时 listener 线程会重新启动一个新的 worker 线程,直到达到指定数量的 worker 线程。

3.4.2 线程池实现

线程池功能由 GUC 参数 enable_thread_pool 控制,该变量设置为 true 时才能使用线

程池功能。代码主要在“openGauss-server/src/gausskernel/process/threadpool”目录中，下面介绍主要代码实现流程。

Postmaster 线程在 ServerLoop 中判断如果启用了线程池功能，则会调用“ThreadPoolControler::Init”函数进行线程池的初始化。在线程池初始化时，会判断 NUMA 节点的个数进行 NUMA 结构处理，相关代码如下：

```
    if (threadPoolActivated) {
        bool enableNumaDistribute = (g_instance.shmem_cxt.numaNodeNum > 1);
        g_threadPoolControler->Init(enableNumaDistribute);
}
```

“ThreadPoolControler::Init”函数的主要作用是创建 m_sessCtrl 成员和 m_groups 成员对象，根据绑核策略分配线程个数，调用“ThreadPoolGroup::init”函数进行线程组的初始化，调用“ThreadPoolGroup::WaitReady”函数等待各个线程组初始化结束。创建 m_scheduler 成员对象，并且调用“ThreadPoolScheduler::StartUp”函数启动线程池调度线程。在“ThreadPoolGroup::init”函数中，创建 m_listener 对象，启动 listener 线程。为 ThreadWorkerSentry 函数分配内存，初始化每个 worker 的互斥量和条件变量。调用“ThreadPoolGroup::AddWorker”函数创建 worker 对象，启动 worker 线程。

Postmaster 线程在 ServerLoop 中如果监听到有客户端链接请求，判断启用了线程池功能，则会调用“ThreadPoolControler::DispatchSession”函数进行会话分发，相关代码如下：

```
if (threadPoolActivated &&!(i < MAXLISTEN && t_thrd.postmaster_cxt.listen_sock_type[i] == HA
_LISTEN_SOCKET))
result = g_threadPoolControler->DispatchSession(port);
/* ThreadPoolControler::DispatchSession 的代码实现如下，找到一个会话数最少的线程组，创建会
话，把会话添加到线程组的监听线程中 */
int ThreadPoolControler::DispatchSession(Port* port)
{
    ThreadPoolGroup* grp = NULL;
    knl_session_context* sc = NULL;
    grp = FindThreadGroupWithLeastSession();
    if (grp == NULL) {
        Assert(false);
        return STATUS_ERROR;
    }
    sc = m_sessCtrl->CreateSession(port);
    if (sc == NULL)
        return STATUS_ERROR;
    grp->GetListener()->AddNewSession(sc);
    return STATUS_OK;
}
```

listener 线程的主函数为“TpoolListenerMain(ThreadPoolListener* listener)”。在该函数中设置线程的名字和信号处理函数，创建 epoll 等待事件，通知 Postmaster 线程已经准备好，调用 t_pool_listener_loop 函数（其实是调用“ThreadPoolListener::WaitTask”函数进

入等待事件状态）。如果有事件到来，调用“ThreadPoolListener::HandleConnEvent”函数找到事件对应的会话。调用“ThreadPoolListener::DispatchSession”函数，如果有空闲的 worker 线程，通知 worker 线程进行处理；如果没有空闲的 worker 线程，则把会话挂到等待队列中。

worker 线程的主函数就是正常的 SQL 处理函数 PostgresMain，与非线程模式相比，主要多了 3 处处理：

（1）worker 线程准备就绪的通知；

（2）等待会话通知；

（3）连接退出处理；

worker 线程的相关代码如下：

```
if (IS_THREAD_POOL_WORKER) {
       u_sess->proc_cxt.MyProcPort->sock = PGINVALID_SOCKET;
       t_thrd.threadpool_cxt.worker->NotifyReady();
      }
      if (IS_THREAD_POOL_WORKER) {
           t_thrd.threadpool_cxt.worker->WaitMission();
           Assert(u_sess->status != KNL_SESS_FAKE);
       }
  case 'X':
  case EOF:
              RemoveTempNamespace();
              InitThreadLocalWhenSessionExit();
              if (IS_THREAD_POOL_WORKER) {
                  t_thrd.threadpool_cxt.worker->CleanUpSession(false);
                  break;
              }
```

“ThreadPoolWorker::WaitMission”函数的主要作用是阻塞所有系统信号，避免系统信号如 SIGHUP 等中断当前的处理。清除线程上的会话信息，保证没有上一个会话的内容，等待会话上新的请求，把会话给线程进行处理，允许系统信号中断。

“ThreadPoolWorker::CleanUpSession”函数的主要作用是清除会话，从 Listener 中去除会话，释放会话资源。

上面介绍了线程池的主要机制，综上所述，线程池主要是解决大并发的用户连接，在一定程度上可以起到流量控制的作用，即使用户的连接数很多，后端也不需要分配太多的线程。线程是 OS 的一种资源，如果线程太多，OS 资源占用很多，并且大量线程的调度和切换会带来昂贵的开销。如果没有线程池，随着连接数的增多，系统的吞吐量会逐渐降低。另外一方面，把线程池划分为线程组，可以很好地匹配 NUMA CPU 架构的节点，提升多核情况下的访问性能。每个线程组一个监听者，避免了线程池的“惊群效应”。

3.5　内存管理

数据库在运行过程中涉及许多对象，这些对象具有不同的生命周期，有些处理需要频繁分配内存。如一个 SQL 语句，在解析时需要对词法单元和语法单元分配内存，在执行过程中需要对执行状态分配内存。在事务结束时，如果不是 PREPARE 语句，那么 SQL 语句的执行计划内存和执行过程的状态内存都需要释放。如果是 PREPARE 语句，那么执行计划需要保存到缓冲池中，执行过程的状态内存释放即可。为了保证内存分配的高效和避免内存泄漏，openGauss 设计开发了自己的内存管理，代码实现在“openGauss-server\src\common\backend\utils\mmgr”目录。

openGauss 在内存管理上采用了上下文的概念，即具有同样生命周期或者属于同一个上下文语义的内存放到一个 MemoryContext 管理，MemoryContext 的结构代码如下（结构成员参照注释）：

```
typedef struct MemoryContextData * MemoryContext;
typedef struct MemoryContextData {
    NodeTag type;                /* 上下文类别 */
    MemoryContextMethods * methods; /* 虚函数表 */
    MemoryContext parent;        /* 父上下文.顶级上下文为 NULL */
    MemoryContext firstchild;    /* 子上下文的链表头 */
    MemoryContext prevchild;     /* 前向子上下文 */
    MemoryContext nextchild;     /* 后向子上下文 */
    char * name;                 /* 上下文名称,方便调试 */
    pthread_rwlock_t lock;       /* 上下文共享时的并发控制锁 */
    bool is_shared;        /* 上下文是否在多个线程共享 */
    bool isReset;          /* isReset 为 true 时,表示复位后没有内存空间用于分配 */
    int level;             /* 上下文层次级别 */
    uint64 session_id;     /* 上下文属于的会话 ID */
    ThreadId thread_id;    /* 上下文属于的线程 ID */
} MemoryContextData;
```

虚函数表就是具体的内存管理操作函数指针，具体定义代码如下（函数功能参照注释）：

```
typedef struct MemoryContextMethods {
/* 在上下文中分配内存 */
    void * ( * alloc)(MemoryContext context,Size align,Size size,const char * file,int line);
    /* 释放 pointer 内存到上下文中 */
void ( * free_p)(MemoryContext context,void * pointer);
/* 在上下文中重新分配内存 */
void * ( * realloc)(MemoryContext context,void * pointer,Size align,Size size,const char *
file,int line);
    void ( * init)(MemoryContext context);              /* 上下文初始化 */
    void ( * reset)(MemoryContext context);             /* 上下文复位 */
    void ( * delete_context)(MemoryContext context);    /* 删除上下文 */
```

```
    Size ( * get_chunk_space)(MemoryContext context,void * pointer);   /* 获取上下文块大小 */
    bool ( * is_empty)(MemoryContext context);          /* 上下文是否为空 */
    void ( * stats)(MemoryContext context,int level);   /* 上下文信息统计 */
#ifdef MEMORY_CONTEXT_CHECKING
    void ( * check)(MemoryContext context);             /* 上下文异常检查 */
#endif
} MemoryContextMethods;
```

这些回调函数指针初始化是在 AllocSetContextSetMethods 函数中调用 AllocSetMethodDefinition 函数完成的。AllocSetMethodDefinition 函数的实现代码如下：

```
template < bool enable_memoryprotect,bool is_shared,bool is_tracked >
void AlignMemoryAllocator::AllocSetMethodDefinition(MemoryContextMethods * method)
{
    method - > alloc = &AlignMemoryAllocator:: AllocSetAlloc < enable _ memoryprotect, is _
shared,is_tracked >;
    method - > free _ p = &AlignMemoryAllocator:: AllocSetFree < enable _ memoryprotect, is _
shared,is_tracked >;
    method - > realloc = &AlignMemoryAllocator::AllocSetRealloc < enable_memoryprotect,is_
shared,is_tracked >;
    method - > init = &AlignMemoryAllocator::AllocSetInit;
    method - > reset = &AlignMemoryAllocator:: AllocSetReset < enable _ memoryprotect, is _
shared,is_tracked >;
    method - > delete _ context = &AlignMemoryAllocator:: AllocSetDelete < enable _
memoryprotect,is_shared,is_tracked >;
    method - > get_chunk_space = &AlignMemoryAllocator::AllocSetGetChunkSpace;
    method - > is_empty = &AlignMemoryAllocator::AllocSetIsEmpty;
    method - > stats = &AlignMemoryAllocator::AllocSetStats;
#ifdef MEMORY_CONTEXT_CHECKING
    method - > check = &AlignMemoryAllocator::AllocSetCheck;
#endif
}
```

可以看到，这些实际操作内存管理的函数为 AlignMemoryAllocator 类中的 AllocSetAlloc 函数、AllocSetFree 函数、AllocSetRealloc 函数、AllocSetInit 函数、AllocSetReset 函数、AllocSetDelete 函数、AllocSetGetChunkSpace 函数、AllocSetIsEmpty 函数、AllocSetStats 函数和 AllocSetCheck 函数。在这些处理函数中，涉及的结构体代码如下：

```
typedef AllocSetContext * AllocSet;
typedef struct AllocSetContext {
    MemoryContextData header;  /* 内存上下文,存储空间是在这个内存上下文中分配的 */
     AllocBlock blocks;        /* AllocSetContext 所管理内存块的块链表头 */
    AllocChunk freelist[ALLOCSET_NUM_FREELISTS];     /* 空闲块链表 */
    /* 这个上下文的分配参数 */
    Size initBlockSize;       /* 初始块大小 */
    Size maxBlockSize;        /* 最大块大小 */
    Size nextBlockSize;       /* 下一个分配的块大小 */
    Size allocChunkLimit;     /* 块大小上限 */
```

```
    AllocBlock keeper;          /* 在复位时保存的块 */
    Size totalSpace;            /* 这个上下文分配的总空间 */
    Size freeSpace;             /* 这个上下文总的空闲空间 */
    Size maxSpaceSize;          /* 最大内存空间 */
    MemoryTrack track;          /* 跟踪内存分配信息 */
} AllocSetContext;
/* AllocBlock 定义如下: */
typedef struct AllocBlockData* AllocBlock;
typedef struct AllocBlockData {
    AllocSet aset;   /* 哪个 AllocSetContext 拥有此块,AllocBlockData 归属 AllocSetContext 管
理 */
    AllocBlock prev;      /* 在块链表中的前向指针 */
    AllocBlock next;      /* 在块链表中的后向指针 */
    char* freeptr;        /* 这个块空闲空间的起始地址 */
    char* endptr;         /* 这个块空间的结束地址 */
    Size allocSize;       /* 分配的大小 */
#ifdef MEMORY_CONTEXT_CHECKING
    uint64 magicNum;                /* 魔鬼数字值,用于内存校验.当前代码固定填写为 DADA */
#endif
} AllocBlockData;
typedef struct AllocChunkData* AllocChunk; /* AllocChunk 内存前面部分是一个 AllocBlock 结构
*/
typedef struct AllocChunkData {
       void* aset;    /* 拥有这个 chunk 的 AllocSetContext,如果空闲,则为空闲列表链接 */
       Size size;          /* chunk 中的使用空间 */
#ifdef MEMORY_CONTEXT_CHECKING
       Size requested_size;    /* 实际请求大小,在空闲块中时为 0 */
       const char* file;       /* palloc/palloc0 调用时的文件名称 */
       int line;               /* palloc/palloc0 调用时的行号 */
       uint32 prenum;          /* 前向魔鬼数字 */
#endif
} AllocChunkData;
```

从前面的数据结构可以看出,核心数据结构为 AllocSetContext,这个数据结构有 3 个成员“MemoryContextData header;”“AllocBlock blocks;”和“AllocChunk freelist[ALLOCSET_NUM_FREELISTS];”。这 3 个成员把内存管理分为 3 个层次。

(1) MemoryContext 管理上下文之间的父子关系,设置 MemoryContext 的内存管理函数。

(2) AllocBlock blocks 把所有内存块通过双链表链接起来。

(3) 内存单元 chunk 是从内存块 AllocBlock 内部分配的,内存块和内存单元 chunk 的转换关系为:“AllocChunk chunk = (AllocChunk)(((char *) block) + ALLOC_BLOCKHDRSZ);”和“AllocBlock block = (AllocBlock)(((char *) chunk) - ALLOC_BLOCKHDRSZ);”。

内存单元 chunk 经过转换得到最终的用户指针,内存单元 chunk 和用户指针 AllocPointer 的转换关系为:((AllocPointer)(((char *)(chk)) + ALLOC_CHUNKHDRSZ))和((AllocChunk)(((char *)(ptr)) - ALLOC_CHUNKHDRSZ))。数据结构的基本关系如图 3-3 所示。

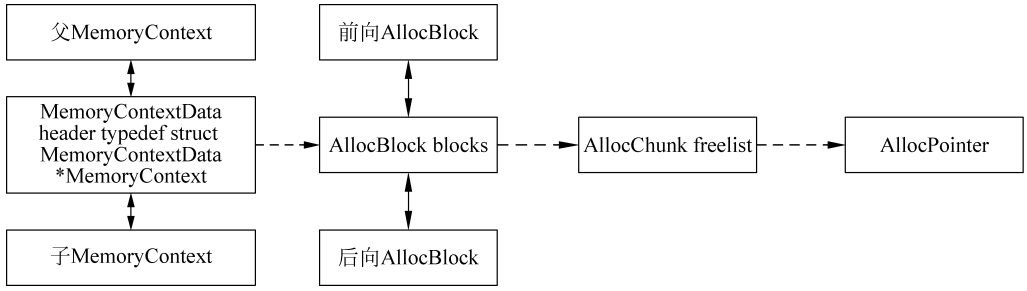

图 3-3　数据结构的基本关系

下面先看第 1 层 MemoryContext(内存上下文)的实现,主要实现在 mcxt.cpp 文件中,如表 3-4 所示。

表 3-4　MemoryContext 的实现函数

函　　数	功能介绍
ChooseRootContext	在线程池机制下,上下文有 3 个类别,即实例级别、会话级别、线程级别。这个函数根据 tag(标签)类型和 parent(上一级)返回相应类别的根上下文(最顶层的上下文)
MemoryContextCreate	首先根据 root 是否为空确定是从父 MemoryContext 分配内存还是从操作系统调用 malloc 分配内存,然后对分配的 MemoryContext 进行初始化,如果存在父 MemoryContext,则挂到父 MemoryContext 上
MemoryContextDelete	先删除这个 MemoryContext 的子节点,把这个 MemoryContext 的父节点置为空,回调 AllocSetDelete 方法释放分配的对象,最后释放上下文本身
MemoryContextIsEmpty	先看当前上下文是否有子节点,然后回调 is_empty 检查当前上下文是否为空
MemoryContextReset	先看当前上下文是否有子节点,如果有子节点,则遍历子节点,递归调用 MemoryContextReset 进行复位,最后回调 reset 复位当前上下文
MemoryContextSetParent	如果上下文有父节点,则从父上下文解除当前上下文,然后把上下文挂到新的父上下文
GetMemoryChunkSpace	当前指针 pointer 偏移 STANDARDCHUNKHEADERSIZE 找到标准 StandardChunkHeader 位置,然后根据块属于上下文的回调 AllocSetGetChunkSpace 获取块空间大小
MemoryContextStats	先回调 AllocSetStats 统计当前上下文信息,然后遍历子节点,递归调用 MemoryContextStatsInternal 统计上下文信息
MemoryContextAllocDebug	检查分配内存大小是否小于 MaxAllocSize,回调 AllocSetAlloc 分配内存
pfree	根据当前指针 pointer 偏移 STANDARDCHUNKHEADERSIZE 找到标准头,根据头部 StandardChunkHeader 找到归属的上下文,回调 AllocSetFree 释放内存

再看第 2 层 AllocSet 的实现,主要实现在 aset.cpp 文件中,如表 3-5 所示。

表 3-5　AllocSet 的实现函数

函　　数	功能介绍
AllocSetFreeIndex	根据请求的 size(内存大小)计算应该在哪个空闲块链表 freechunk 中分配内存
set_sentinel	设置哨兵 0x7E,用于内存越界写操作(修改了不应该修改的内存地址)检查
sentinel_ok	内存检查是否正常
MemoryContextControlSet	根据白名单设置上下文的 maxSpaceSize 限制大小
AllocSetContextCreate	根据 contextType 类型,调用不同的 AllocSetContextCreate 分配器分配 MemoryContext
AllocSetMethodDefinition	设置 MemoryContext 回调处理方法
AllocSetContextSetMethods	设置不同上下文类型的分配器
AllocSetContextCreate	创建一个具体的 MemoryContext。根据类型,确定内存保护函数。调用 MemoryContextCreate 函数创建一个 AllocSetContext,设置 maxSpaceSize 大小,设置回调方法,设置初始块大小、下一个块大小、最大块大小。根据分配的最大块大小设置 allocChunkLimit。如果上下文最值超过了“ALLOC_BLOCKHDRSZ + ALLOC_CHUNKHDRSZ”,则分配一个 AllocBlock;设置 AllocBlock 的上下文,空闲起始地址(freeptr)为跳过头部管理占用空间后剩余的空间,末尾地址(endptr)为块结束地址,分配大小(allocSize)为分配的块大小,魔鬼数字(magicNum)为 0xDADADADADADADADA,context 的总空间(totalSpace)加上这次分配的块大小,上下文的空闲空间(freeSpace)加上这个块的空闲空间,块的前向和后向指针为空,AllocSetContext 的第一个块(blocks)指向这个块,保留块(keeper)指向这个块。返回 AllocSetContext
AllocSetInit	特定 AllocSetContext 初始化函数,当前为空,没有使用
AllocSetReset	根据上下文类型选择保护函数,把空闲链表(freelist)置空,遍历内存块,如果是保留块,则对这块内存重新初始化,如果不是,则释放掉。根据是否有保留块重新初始化上下文
AllocSetDelete	如果上下文没有内存块,则直接返回。根据上下文类型选择保护函数,获取内存块首地址,把保留块和内存块头置空。遍历所有内存块,释放掉。把上下文的空闲空间(freeSpace)和总空间(totalSpace)置为 0
AllocSetAlloc	根据上下文类型选择保护函数,如果申请内存大小超过了内存块大小上限,则直接调用 OS(操作系统)接口分配一个内存块,初始化这个内存块。把内存块转换为一个内存单元(chunk),对这个内存单元进行初始化。把这个内存块挂到上下文上,返回内存单元 chunk 指针。 如果申请内存大小没有超过内存块大小上限,根据内存大小映射的空闲链表(freelist),看相应空闲链表中是否有相应大小的内存单元,如果有空闲的内存单元,则分配一个内存单元返回。 如果空闲链表(freelist)没有空闲的内存单元,看当前的块(blocks)是否有足够的内存,如果没有足够的内存,则根据块剩余的内存大小放到相应的空闲链表中。分配一个新的块,对这个新的块进行初始化,把块挂到上下文上,从这个块上分配一个内存单元返回。 如果当前的块有足够的内存,则从当前块上分配一个内存单元返回

续表

函　　数	功能介绍
AllocSetFree	根据类型，确定内存保护函数。如果释放内存单元大小超过了内存块大小上限，把这个内存单元转换为 AllocBlock，把内存块从上下文中解除，把内存块变量置空，释放内存块。 如果内存单元大小小于内存块上限，则根据内存单元大小映射的空闲链表，把释放的内存单元挂到上下文的空闲链表上
AllocSetRealloc	根据类型，确定内存保护函数。如果原来的内存单元大小能够满足请求的大小，则重新赋值新的内存检查信息后直接返回当前的内存单元。 如果旧的内存单元大小超过了内存块的上限，则调用 OS 重分配内存接口重新分配一个内存块，对新的内存块初始化，把新的内存块转换为内存单元 AllocChunk 返回。 如果旧的内存单元大小没有超过内存块的上限，则根据新的大小调用 AllocSetAlloc 分配一个新的内存单元，把旧的内存单元值复制到新的内存单元，调用 AllocSetFree 释放原来的内存单元
AllocSetGetChunkSpace	返回内存单元的大小，包括头部占用的空间
AllocSetIsEmpty	检查是否复位(isReset)，如果是，则返回 true；否则返回 false
AllocSetStats	显示上下文的内存消耗信息，打印到标准 stderr(标准错误)输出上

3.6　多维监控

数据库是企业的关键组件，数据库的性能直接决定了很多业务的吞吐量。为了简化数据库维护人员的调优，openGauss 对数据库运行进行了多维度的监控，并且开发了一些维护特性，如 WDR(Wordload Diagnostic Report，工作负荷诊断报告)性能诊断报告、慢 SQL 诊断、会话性能诊断、系统 KPI(Key Performance Indicator，关键性能指标)辅助诊断等，帮助维护人员对数据库的性能进行诊断。这些监控项都以视图的方式对外呈现，集中在 DBE_PERF 模式下。WDR Snapshot 除了自身快照的元数据，其他数据表来源也是 DBE_PERF schema 下的视图。WDR Snapshot 数据表命名原则：snap_{源数据表}，根据这个关系可以找到 snap 表所对应的原表。对这些视图的解释参照 openGauss 的官网(https://opengauss.org)中《开发者指南》手册的“DBE_PERF schema”章节。

性能视图的源代码在“openGauss-server\src\common\backend\catalog\performance_views.sql”文件中(网址为 https://gitee.com/opengauss/openGauss-server/blob/master/src/common/backend/catalog/performance_ views.sql，安装后会复制到安装路径的“performance_views.sql”下)。在数据库初始化阶段由 initdb 读取这个文件在数据库系统中创建相应的视图。这些视图遵循了 openGauss 通用视图的实现逻辑，即视图来自函数的封装，这些函数可能是内置函数，也可能是存储函数。OS 运行的性能视图“dbe_perf.get_global_os_runtime”的相关代码如下：

```
CREATE OR REPLACE FUNCTION dbe_perf.get_global_os_runtime
  (OUT node_name name,OUT id integer,OUT name text,OUT value numeric,OUT comments text,OUT
cumulative boolean)
RETURNS setof record
AS $ $
DECLARE
  row_data dbe_perf.os_runtime%rowtype;
query_str := 'SELECT * FROM dbe_perf.os_runtime';
      FOR row_data IN EXECUTE(query_str) LOOP
   …
    END LOOP;
    return;
  END; $$
LANGUAGE 'plpgsql' NOT FENCED;
CREATE VIEW dbe_perf.global_os_runtime AS
  SELECT DISTINCT * FROM dbe_perf.get_global_os_runtime();
```

global_os_runtime 视图来自存储函数 get_global_os_runtime 的封装，在存储函数内访问 dbe_perf.os_runtime 视图、os_runtime 视图的 SQL 语句为“CREATE VIEW dbe_perf.os_runtime AS SELECT * FROM pv_os_run_info();”。pv_os_run_info 是内置函数，而内置函数负责读取数据库系统的监控指标，pv_os_run_info 函数的相关代码如下：

```
Datum pv_os_run_info(PG_FUNCTION_ARGS)
{
    FuncCallContext* func_ctx = NULL;
    /*判断是不是第一次调用*/
    if (SRF_IS_FIRSTCALL()) {
        MemoryContext old_context;
        TupleDesc tup_desc;
        /*创建函数上下文*/
        func_ctx = SRF_FIRSTCALL_INIT();
        /*
         * 切换内存上下文到多次调用上下文
         */
        old_context = MemoryContextSwitchTo(func_ctx->multi_call_memory_ctx);
        /*创建一个包含 5 列的元组描述模板*/
        tup_desc = CreateTemplateTupleDesc(5,false);
        TupleDescInitEntry(tup_desc,(AttrNumber)1,"id",INT4OID,-1,0);
        TupleDescInitEntry(tup_desc,(AttrNumber)2,"name",TEXTOID,-1,0);
        TupleDescInitEntry(tup_desc,(AttrNumber)3,"value",NUMERICOID,-1,0);
        TupleDescInitEntry(tup_desc,(AttrNumber)4,"comments",TEXTOID,-1,0);
        TupleDescInitEntry(tup_desc,(AttrNumber)5,"cumulative",BOOLOID,-1,0);

        /*填充元组描述模板*/
        func_ctx->tuple_desc = BlessTupleDesc(tup_desc);
        /*收集系统信息*/
        getCpuNums();
        getCpuTimes();
        getVmStat();
```

```
            getTotalMem();
            getOSRunLoad();
            (void)MemoryContextSwitchTo(old_context);
        }

        /* 设置函数的上下文,每次函数调用都需要 */
        func_ctx = SRF_PERCALL_SETUP();
        while (func_ctx->call_cntr < TOTAL_OS_RUN_INFO_TYPES) {
            /* 填充所有元组每个字段的值 */
            Datum values[5];
            bool nulls[5] = {false};
            HeapTuple tuple = NULL;
            errno_t rc = 0;
            rc = memset_s(values,sizeof(values),0,sizeof(values));
            securec_check(rc,"\0","\0");
            rc = memset_s(nulls,sizeof(nulls),0,sizeof(nulls));
            securec_check(rc,"\0","\0");
            if (!u_sess->stat_cxt.osStatDescArray[func_ctx->call_cntr].got) {
                ereport(DEBUG3,
                    (errmsg("the %s stat has not got on this plate.",
                        u_sess->stat_cxt.osStatDescArray[func_ctx->call_cntr].name)));
                func_ctx->call_cntr++;
                continue;
            }
            values[0] = Int32GetDatum(func_ctx->call_cntr);
            values[1] = CStringGetTextDatum(u_sess->stat_cxt.osStatDescArray[func_ctx->
    call_cntr].name);
           values[2] = u_sess->stat_cxt.osStatDescArray[func_ctx->call_cntr].getDatum(
                u_sess->stat_cxt.osStatDataArray[func_ctx->call_cntr]);
        values[3] = CStringGetTextDatum(u_sess->stat_cxt.osStatDescArray[func_ctx->call_
    cntr].comments);
            values[4] = BoolGetDatum(u_sess->stat_cxt.osStatDescArray[func_ctx->call_
    cntr].cumulative);

            tuple = heap_form_tuple(func_ctx->tuple_desc,values,nulls);
            SRF_RETURN_NEXT(func_ctx,HeapTupleGetDatum(tuple));
        }
        /* 填充结束,返回结果 */
        SRF_RETURN_DONE(func_ctx);
    }
```

pv_os_run_info 函数可以分为三段：

(1) 调用 CreateTemplateTupleDesc 函数和 TupleDescInitEntry 函数定义元组描述信息。

(2) 调用 getCpuNums 函数、getCpuTimes 函数、getVmStat 函数、getTotalMem 函数、getOSRunLoad 函数收集系统信息。

(3) 把收集的 u_sess 信息填充到元组数据中,最后返回给调用者。openGauss 提供了实现返回结果的通用 SQL 函数的实现步骤和方法,它们是 SRF_IS_FIRSTCALL、SRF_PERCALL_SETUP、SRF_RETURN_NEXT 和 SRF_RETURN_DONE。从代码可以看

出，pv_os_run_info 的实现流程也遵循 openGauss 通用的 SQL 函数实现方法。

系统指标的收集来自读取系统信息，对数据库系统中一些模块进行打点（打点就是按照规格采集指定数据，用以记录系统运行的一些关键点）。很多打点集中在两个方面：事务执行次数和执行时间，从而推断最大时间、最小时间、平均时间。这些比较分散，代码逻辑相对简单，这里不再进行介绍，只需要根据内置函数读取的变量查看这些变量赋值的地方就可以追踪具体的实现位置。

openGauss 主要维护特性的实现代码在“openGauss-server\src\gausskernel\cbb\instruments”目录中，如 WDR、SQL 百分位计算，这里不再进行介绍。

性能统计对 openGauss 的正常运行也会带来一定的性能损耗，所以这些特性都有开关控制，具体说明如下。

（1）等待事件信息实时收集功能的开关为 enable_instr_track_wait。

（2）Unique SQL 信息实时收集功能的开关为 enable_instr_unique_sql、enable_instr_rt_percentile。

（3）数据库监控快照功能的开关为 enable_wdr_snapshot。

其他功能也都有相应的 GUC 参数进行调节，根据平常使用的需要，可以打开具体维护项查看系统的运行情况。

3.7　模拟信号机制

信号是 Linux 进程/线程之间的一种通信机制，向一个进程发送信号的系统函数是 kill，向一个线程发送信号的系统函数是 pthread_kill。在 openGauss 中既有 gs_ctl 向 openGauss 进程发送的进程间信号，也有 openGauss 进程中线程间的信号。

信号是一种有限的资源，OS 提供的信号有 SIGINT、SIGQUIT、SIGTERM、SIGALRM、SIGPIPE、SIGFPE、SIGUSR1、SIGUSR2、SIGCHLD、SIGTTIN、SIGTTOU、SIGXFSZ 等。这些信号一般都是系统专用的，每个信号都有专门的用途，比如 SIGALRM 是系统定时器的通知信号。留给应用的信号主要是 SIGUSR1、SIGUSR2。

在系统信号有限的情况下，为了在 openGauss 中表达不同的丰富的通信语义，openGauss 额外增加了新的变量表示具体的语义。openGauss 是多线程架构，在同一个进程内如果不同的线程注册了不同的处理函数，则后者会覆盖前者的信号处理。为了不同线程能够注册不同的处理函数，需要自己管理信号对应的注册函数。为了解决这些问题，openGauss 实现了信号的模拟机制。信号模拟的基本原理是每个线程注册管理自己的信号处理函数，信号枚举值仍然使用系统的信号值，线程使用自己的变量记录信号和回调函数对应关系。线程之间发送信号时，先设置变量为具体的信号值，然后使用系统调用 pthread_kill 发送信号，线程收到通知后再根据额外的变量表示的具体信号值，回调对应的信号处理函数。

信号处理涉及的数据结构代码如下。每个线程有一个 GsSignalSlot 结构，保存了线程 ID、线程名称和 GsSignal 结构，而 GsSignal 结构保存了每个信号对应的处理函数数组和每

个线程相关的信号池。信号池 struct SignalPool 包含了使用的信号列表和空闲的信号列表，当一个模拟信号到达时，找一个空闲信号 GsNode，然后放到使用的列表中。GsNode 中存放了信号值结构 GsSndSignal sig_data。GsSndSignal 结构中保存了发送的信号具体值和发送的线程 ID。当需要设置一些额外检查信息时，设置 GsSignalCheck 内容。相关代码如下：

```
typedef struct GsSignalSlot {
    ThreadId thread_id;
    char * thread_name;
    GsSignal * gssignal;
} GsSignalSlot;
typedef struct GsSignal {
    gs_sigfunc handlerList[GS_SIGNAL_COUNT];
    sigset_t masksignal;
    SignalPool sig_pool;
    volatile unsigned int bitmapSigProtectFun;
} GsSignal;
typedef struct SignalPool {
    GsNode * free_head;          /* 空闲信号列表头部 */
    GsNode * free_tail;          /* 空闲信号列表尾部 */
    GsNode * used_head;          /* 使用信号列表头部 */
    GsNode * used_tail;          /* 使用信号列表尾部 */
    int pool_size;               /* 数组列表大小     */
    pthread_mutex_t sigpool_lock;
} SignalPool;
typedef struct GsNode {
    GsSndSignal sig_data;
    struct GsNode * next;
} GsNode;
typedef struct GsSndSignal {
    unsigned int signo;          /* 需要处理的信号 */
    gs_thread_t thread;          /* 发送信号的线程 ID */
    GsSignalCheck check;         /* 信号发送线程需要检查的信息 */
} GsSndSignal;
typedef struct GsSignalCheck {
    GsSignalCheckType check_type;
    uint64 debug_query_id;
    uint64 session_id;
} GsSignalCheck;
```

信号处理的几个主要流程为初始化模拟信号机制、注册信号处理函数、发送信号和处理信号。具体的处理逻辑如下。

(1) 初始化模拟信号机制函数 gs_signal_slots_init。在 gs_signal_slots_init 处理函数中完成以下功能。

① 根据传入的槽位个数，分配内存。遍历每个槽位，进行初始化，初始化时调用 gs_signal_init 函数对每个槽位的 GsSignal(GsSignal 是 openGauss 封装的模拟信号结构体，里面包含了信号掩码和信号处理函数等成员)进行初始化。

② 在 gs_signal_init 函数中，对 GsSignal 分配内存和初始化，初始化时调用 gs_signal_

sigpool_init 函数对信号池初始化。

③ 在 gs_signal_sigpool_init 函数中，对信号池进行分配内存和初始化。

(2) 注册信号处理函数 gspqsignal。在 gspqsignal 处理函数中完成以下功能。

① 调用“gs_signal_register_handler(t_thrd. signal_slot-> gssignal，signo，func)；”函数把信号对应的处理函数注册到 GsSignal 中。在注册之前，需要为线程分配一个 signal_slot，这个是在 gs_signal_startup_siginfo 函数中完成的。

② 在 gs_signal_startup_siginfo 函数中，调用 gs_signal_alloc_slot_for_new_thread 函数为线程分配一个 signal_slot。该函数的功能是遍历“g_instance. signal_base-> slots”，找到一个未使用的 slot(thread_id 为 0 表示未使用)，然后设置本线程 ID 和线程名称。

(3) 发送信号函数 gs_signal_send。在 gs_signal_send 处理函数中完成以下功能。

① 调用函数 gs_signal_find_slot 找到要发送线程所在的 GsSignalSlot。

② 调用函数 gs_signal_set_signal_by_threadid 设置模拟信号。该函数首先检查信号在使用列表中是否已经存在，如果已经存在，则直接返回；如果不存在，则在空闲列表中找到一个空闲 GsNode，设置信号值，发送线程 ID、check_type 到 sig_data 中，最后把空闲 GsNode 移到使用列表中。

③ 调用函数 gs_signal_thread_kill 发送信号通知。该函数遍历 GsSignalSlot，找到匹配的线程 ID，然后调用“gs_signal_thread_kill(thread_id，RES_SIGNAL)；”函数给具体线程发送信号通知。语句“# define RES_SIGNAL SIGUSR2”表示内部统一都使用 SIGUSR2 发送通知。

(4) 处理信号函数 gs_signal_handle。在函数 gs_signal_handle 中完成以下功能。

① 遍历信号池使用列表，找到一个需要处理的信号。

② 找到这个信号对应的信号处理函数，把 GsNode 移到空闲列表中。

③ 调用 gs_signal_handle_check 函数检查当前的条件是否仍然满足，如果仍然有效，回调处理函数。

3.8　本章小结

本章主要介绍了 openGauss 的一些公共组件机制，每个内容都比较独立。系统表是 openGauss 的元数据，本章主要介绍了系统表的定义和 syscache 访问机制；数据库初始化是数据库安装后的第一步，它负责创建数据库的模板数据库和数据目录；多线程架构是 openGauss 启动后的运行机制，本章介绍了主线程的初始化流程、后台线程的启动、各个线程的功能和线程之间的通信机制；线程池技术是解决大并发链接的有效方法，本章介绍了线程池机制的原理，各个类之间的关系和设计原因；内存管理是 openGauss 的内存资源管理组件，本章介绍了 openGauss 的三级内存管理机制；多维监控是 openGauss 性能调优手段的基础，本章介绍了性能视图的基本实现原理；模拟信号机制是 openGauss 多线程处理紧急事件的机制，本章介绍了这个机制的实现原理。