PostgreSQL 数据库启动过程分析

1　启动/停止 PostgreSQL 数据库（实例）的方法

启动/停止数据库可以使用命令 pg_ctl，它位于/opt/db/pgsql/bin 目录下。执行下面的命令，可以获取命令 pg_ctl 的使用帮助：

```
[postgres@dbsvr ~]$ pg_ctl --help
pg_ctl is a utility to initialize, start, stop, or control a PostgreSQL server.

Usage:
  pg_ctl init[db]   [-D DATADIR] [-s] [-o OPTIONS]
  pg_ctl start      [-D DATADIR] [-l FILENAME] [-W] [-t SECS] [-s]
                    [-o OPTIONS] [-p PATH] [-c]
  pg_ctl stop       [-D DATADIR] [-m SHUTDOWN-MODE] [-W] [-t SECS] [-s]
  pg_ctl restart    [-D DATADIR] [-m SHUTDOWN-MODE] [-W] [-t SECS] [-s]
                    [-o OPTIONS] [-c]
  pg_ctl reload     [-D DATADIR] [-s]
  pg_ctl status     [-D DATADIR]
  pg_ctl promote    [-D DATADIR] [-W] [-t SECS] [-s]
  pg_ctl logrotate  [-D DATADIR] [-s]
  pg_ctl kill       SIGNALNAME PID

Common options:
  -D, --pgdata=DATADIR   location of the database storage area
  -s, --silent           only print errors, no informational messages
  -t, --timeout=SECS     seconds to wait when using -w option
  -V, --version          output version information, then exit
  -w, --wait             wait until operation completes (default)
  -W, --no-wait          do not wait until operation completes
  -?, --help             show this help, then exit
If the -D option is omitted, the environment variable PGDATA is used.

Options for start or restart:
  -c, --core-files       allow postgres to produce core files
  -l, --log=FILENAME     write (or append) server log to FILENAME
  -o, --options=OPTIONS  command line options to pass to postgres
                         (PostgreSQL server executable) or initdb
  -p PATH-TO-POSTGRES    normally not necessary

Options for stop or restart:
  -m, --mode=MODE        MODE can be "smart", "fast", or "immediate"

Shutdown modes are:
  smart       quit after all clients have disconnected
```

```
      fast        quit directly, with proper shutdown (default)
      immediate   quit without complete shutdown; will lead to recovery on restart

    Allowed signal names for kill:
      ABRT HUP INT KILL QUIT TERM USR1 USR2

    Report bugs to <pgsql-bugs@lists.postgresql.org>.
    PostgreSQL home page: <https://www.postgresql.org/>
    [postgres@dbsvr ~]$
```

1.1 启动 PostgreSQL 数据库（实例）

执行下面的命令，启动 PostgreSQL 数据库：

```
    [postgres@dbsvr ~]$ pg_ctl -D /opt/db/userdb/pgdata/ start
    waiting for server to start....2024-03-25 07:45:35.819 CST [122354] LOG:  redirecting log
output to logging collector process
    2024-03-25 07:45:35.819 CST [122354] HINT:  Future log output will appear in directory
"log".
     done
    server started
    [postgres@dbsvr ~]$ pgps
    postgres  122354       1  0 07:45 ?        00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
    postgres  122355  122354  0 07:45 ?        00:00:00 postgres: logger
    postgres  122357  122354  0 07:45 ?        00:00:00 postgres: checkpointer
    postgres  122358  122354  0 07:45 ?        00:00:00 postgres: background writer
    postgres  122359  122354  0 07:45 ?        00:00:00 postgres: walwriter
    postgres  122360  122354  0 07:45 ?        00:00:00 postgres: autovacuum launcher
    postgres  122361  122354  0 07:45 ?        00:00:00 postgres: archiver
    postgres  122362  122354  0 07:45 ?        00:00:00 postgres: stats collector
    postgres  122363  122354  0 07:45 ?        00:00:00 postgres: logical replication launcher
    [postgres@dbsvr ~]$
```

1.2 关闭 PostgreSQL 数据库（实例）的三种方式

一般来说，PostgreSQL 数据库用户会通过执行下面的命令，关闭 PostgreSQL 数据库：

```
    [postgres@dbsvr ~]$ pg_ctl stop
    waiting for server to shut down.... done
    server stopped
    [postgres@dbsvr ~]$ pgps
    PostgreSQL is not up!
    [postgres@dbsvr ~]$
```

本质上，关闭 PostgreSQL 数据库（实例）有如下的三种方式：

- **smart**：不再允许新的连接，但是允许所有活跃的会话正常完成他们的工作，必须等待所有客户端断开连接后才关闭 PostgreSQL 数据库（实例）。
- **fast**：默认的关闭模式，开始关闭后，不允许建立新的数据库连接，已有的连接将回滚未提交的事务，回滚结束后将关闭 PostgreSQL 数据库（实例）。
- **immediate**：不等待现有连接回滚事务，立即关闭 PostgreSQL 数据库（实例），这相当的野蛮，相当于发生了一次 PostgreSQL 数据库（实例）故障，会导致在重启动 PostgreSQL 数据库（实例）时，需要进行数据库恢复（recovery）操作。建议只有在紧急的情况下，才使用这个方法关闭数据库。

使用 pg_ctl 命令行工具的-m 选项，可以指定关闭 PostgreSQL 数据库（实例）的方式：

- **-m smart**：以 smart 模式关闭数据库实例。
- **-m fast** ：以 fast 模式关闭数据库实例。
- **-m immediate**：以 immediate 模式关闭数据库实例。

下面的示例，用不同的方式关闭 PostgreSQL 数据库（实例）：

```
[postgres@dbsvr ~]$ pg_ctl -D /opt/db/userdb/pgdata start
（省略了一些输出）
server started
[postgres@dbsvr ~]$ pg_ctl -m smart -D /opt/db/userdb/pgdata stop
waiting for server to shut down.... done
server stopped
[postgres@dbsvr ~]$ pg_ctl -D /opt/db/userdb/pgdata start
（省略了一些输出）
server started
[postgres@dbsvr ~]$ pg_ctl -m fast  -D /opt/db/userdb/pgdata stop
waiting for server to shut down.... done
server stopped
[postgres@dbsvr ~]$ pg_ctl -D /opt/db/userdb/pgdata start
（省略了一些输出）
server started
[postgres@dbsvr ~]$ pg_ctl -m immediate  -D /opt/db/userdb/pgdata stop
waiting for server to shut down.... done
server stopped
[postgres@dbsvr ~]$
```

1.3 使用 postgres 程序启动 PostgreSQL 数据库（实例）和 kill 命令

可执行程序 postgres 位于/opt/db/pgsql/bin 目录下：

```
[postgres@dbsvr ~]$ postgres --help
postgres is the PostgreSQL server.

Usage:
  postgres [OPTION]...

Options:
  -B NBUFFERS        number of shared buffers
  -c NAME=VALUE      set run-time parameter
  -C NAME            print value of run-time parameter, then exit
  -d 1-5             debugging level
  -D DATADIR         database directory
  -e                 use European date input format (DMY)
  -F                 turn fsync off
  -h HOSTNAME        host name or IP address to listen on
  -i                 enable TCP/IP connections
  -k DIRECTORY       Unix-domain socket location
  -N MAX-CONNECT     maximum number of allowed connections
  -p PORT            port number to listen on
  -s                 show statistics after each query
  -S WORK-MEM        set amount of memory for sorts (in kB)
  -V, --version      output version information, then exit
  --NAME=VALUE       set run-time parameter
  --describe-config  describe configuration parameters, then exit
  -?, --help         show this help, then exit

Developer options:
  -f s|i|o|b|t|n|m|h forbid use of some plan types
  -n                 do not reinitialize shared memory after abnormal exit
  -O                 allow system table structure changes
  -P                 disable system indexes
  -t pa|pl|ex        show timings after each query
  -T                 send SIGSTOP to all backend processes if one dies
  -W NUM             wait NUM seconds to allow attach from a debugger

Options for single-user mode:
  --single           selects single-user mode (must be first argument)
  DBNAME             database name (defaults to user name)
  -d 0-5             override debugging level
  -E                 echo statement before execution
  -j                 do not use newline as interactive query delimiter
  -r FILENAME        send stdout and stderr to given file

Options for bootstrapping mode:
  --boot             selects bootstrapping mode (must be first argument)
  DBNAME             database name (mandatory argument in bootstrapping mode)
```

```
  -r FILENAME        send stdout and stderr to given file
  -x NUM             internal use

Please read the documentation for the complete list of run-time
configuration settings and how to set them on the command line or in
the configuration file.

Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
[postgres@dbsvr ~]$
```

执行下面的命令，启动 PostgreSQL 数据库：

```
[postgres@dbsvr ~]$ postgres -D /opt/db/userdb/pgdata &
[1] 123439
[postgres@dbsvr ~]$ 2024-03-25 07:58:46.636 CST [123439] LOG:  redirecting log output to
logging collector process
2024-03-25 07:58:46.636 CST [123439] HINT:  Future log output will appear in directory
"log".
（按“回车键”）
[postgres@dbsvr ~]$ pgps
postgres  123439   35919  0 07:58 pts/2    00:00:00 postgres -D /opt/db/userdb/pgdata
postgres  123440  123439  0 07:58 ?        00:00:00 postgres: logger
postgres  123442  123439  0 07:58 ?        00:00:00 postgres: checkpointer
postgres  123443  123439  0 07:58 ?        00:00:00 postgres: background writer
postgres  123444  123439  0 07:58 ?        00:00:00 postgres: walwriter
postgres  123445  123439  0 07:58 ?        00:00:00 postgres: autovacuum launcher
postgres  123446  123439  0 07:58 ?        00:00:00 postgres: archiver
postgres  123447  123439  0 07:58 ?        00:00:00 postgres: stats collector
postgres  123448  123439  0 07:58 ?        00:00:00 postgres: logical replication launcher
[postgres@dbsvr ~]$
```

1.4 使用 kill 命令关闭 PostgreSQL 数据库（实例）

使用 Linux 命令 kill，通过给主进程 postgres 发以下的三种信号，来关闭 PostgreSQL 数据库（实例）：

- 发送信号 SIGTERM，以 smart 模式关闭数据库实例。
- 发送信号 SIGINT，以 fast 模式关闭数据库实例。
- 发送信号 SIGOUIT，以 immediate 模式关闭数据库实例。

下面的示例，用不同的方式关闭 PostgreSQL 数据库（实例）：

```
[postgres@dbsvr ~]$ postgres -D /opt/db/userdb/pgdata &
[postgres@dbsvr ~]$ # 以 smart 方式关闭正在运行的 PostgreSQL 数据库实例
[postgres@dbsvr ~]$ kill -TERM `head -1 /opt/db/userdb/pgdata/postmaster.pid`
[postgres@dbsvr ~]$
[postgres@dbsvr ~]$ postgres -D /opt/db/userdb/pgdata &
[postgres@dbsvr ~]$ # 以 fast 方式关闭正在运行的 PostgreSQL 数据库实例
[postgres@dbsvr ~]$ kill -QUIT `head -1 /opt/db/userdb/pgdata/postmaster.pid`
[postgres@dbsvr ~]$
[postgres@dbsvr ~]$ postgres -D /opt/db/userdb/pgdata &
[postgres@dbsvr ~]$ # 以 immediate 方式关闭正在运行的 PostgreSQL 数据库实例
[postgres@dbsvr ~]$ kill -INT `head -1 /opt/db/userdb/pgdata/postmaster.pid`
[postgres@dbsvr ~]$
```

2 Postgres 程序简介

可执行程序 postgres 是 PostgreSQL 数据库服务器的核心组件之一，负责管理数据库系统的各个方面，包括处理客户端请求、执行 SQL 查询、维护数据库文件、日志记录、错误处理等：

- **启动和停止数据库服务器：**通过运行 postgres 二进制程序，可以启动一个 PostgreSQL 数据库服务器实例；通过停止 postgres 进程，您可以关闭数据库服务器。
- **配置：**postgres 程序读取并解释数据库服务器的配置文件，通常是 postgresql.conf。配置文件包含有关数据库服务器行为的各种设置，例如监听地址、端口号、内存使用等。
- **管理数据库：**postgres 程序允许您连接到 PostgreSQL 数据库服务器并执行管理任务，如创建、删除和管理数据库、表和用户等。
- **与客户端通信：**客户端应用程序通过连接到数据库服务器上的 postgres 进程来与数据库交互。客户端应用程序可以使用各种编程接口和工具，如 psql、pgAdmin 等，通过 postgres 进程执行 SQL 查询和管理数据库。
- **日志和错误处理：**postgres 程序生成日志以记录数据库服务器的活动和错误信息，帮助管理员进行故障排除和性能优化。
- **扩展：**PostgreSQL 是一个高度可扩展的数据库系统，postgres 程序允许您启用和管

理各种扩展功能，如复制、分区、索引等。

3　分析 src/backend/main/main.c

3.1 调试分析前的准备工作

如果 PostgreSQL 数据库正在运行：

```
[postgres@dbsvr db]$ pgps
postgres   51320      1  1 14:41 ?        00:00:00 /opt/db/pg14/bin/postgres
postgres   51321   51320  0 14:41 ?        00:00:00 postgres: logger
postgres   51323   51320  0 14:41 ?        00:00:00 postgres: checkpointer
postgres   51324   51320  0 14:41 ?        00:00:00 postgres: background writer
postgres   51325   51320  0 14:41 ?        00:00:00 postgres: walwriter
postgres   51326   51320  0 14:41 ?        00:00:00 postgres: autovacuum launcher
postgres   51327   51320  0 14:41 ?        00:00:00 postgres: archiver
postgres   51328   51320  0 14:41 ?        00:00:00 postgres: stats collector
postgres   51329   51320  0 14:41 ?        00:00:00 postgres: logical replication launcher
[postgres@dbsvr db]$
```

执行下面的命令，关闭 PostgreSQL 数据库：

```
[postgres@dbsvr db]$ pg_ctl stop
waiting for server to shut down.... done
server stopped
[postgres@dbsvr db]$
```

使用备份恢复数据库集簇：

```
[postgres@dbsvr ~]$ cd /opt/db
[postgres@dbsvr db]$ rm -rf userdb
[postgres@dbsvr db]$ tar xf userdb.tar
[postgres@dbsvr db]$
```

3.2 开始调试分析 postgres

```
[postgres@dbsvr ~]$ cd /opt/db/pgsql/bin
[postgres@dbsvr bin]$ gdb postgres -q
Reading symbols from postgres...
(gdb) list main
55      /*
```

```
56       * Any Postgres server process begins execution here.
57       */
58      int
59      main(int argc, char *argv[])
60      {
61              bool            do_check_root = true;
62
63              /*
64               * If supported on the current platform, set up a handler to be called if
(gdb)
```

执行下面的 gdb 命令，设置断点：

```
(gdb) break main
Breakpoint 1 at 0x7b05dc: file main.c, line 61.
(gdb)
```

执行下面的 gdb 命令，运行 postgres：

```
(gdb) run -D /opt/db/userdb/pgdata
Starting program: /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib64/libthread_db.so.1".

Breakpoint 1, main (argc=3, argv=0x7fffffffeb08) at main.c:61
61              bool            do_check_root = true;
(gdb)
```

3.3 程序入口 main 函数

```
(gdb) list 55,62
50      static void init_locale(const char *categoryname, int category, const char *locale);
51      static void help(const char *progname);
52      static void check_root(const char *progname);
53
54
55      /*
56       * Any Postgres server process begins execution here.
57       */
58      int
59      main(int argc, char *argv[])
60      {
61              bool            do_check_root = true;
62
```

```
(gdb)
```

第 59 行：main 函数，是程序的入口点。它接受两个参数：argc 表示命令行参数的数量，argv 是一个指向参数字符串数组的指针。

第 61 行：定义了一个布尔变量 do_check_root，初始化为 true。

3.4 设置 Window 平台的异常处理函数

```
(gdb) list 63,70
63              /*
64               * If supported on the current platform, set up a handler to be called if
65               * the backend/postmaster crashes with a fatal signal or exception.
66               */
67      #if defined(WIN32) && defined(HAVE_MINIDUMP_TYPE)
68              pgwin32_install_crashdump_handler();
69      #endif
70
(gdb)
```

第 67-69 行：条件编译，如果是在 Windows 平台且支持 minidump，则调用 pgwin32_install_crashdump_handler() 函数。这个函数的作用是在后台进程或主进程发生致命信号或异常时设置一个处理程序。

3.5 获取运行程序的名字

```
(gdb) list 71,72
71              progname = get_progname(argv[0]);
72
(gdb)
```

第 71 行：调用 get_progname(argv[0]) 函数，获取程序的名称。progname 是一个全局变量，存储了程序的名字。

3.6 执行平台特定的启动操作

```
(gdb) list 73,77
73              /*
74               * Platform-specific startup hacks
75               */
76              startup_hacks(progname);
77
(gdb)
```

第 76 行：调用 startup_hacks(progname) 函数，执行平台特定的启动操作。

3.7 保存命令参数

```
(gdb) list 78,90
78              /*
79               * Remember the physical location of the initially given argv[] array for
80               * possible use by ps display.  On some platforms, the argv[] storage must
81               * be overwritten in order to set the process title for ps. In such cases
82               * save_ps_display_args makes and returns a new copy of the argv[] array.
83               *
84               * save_ps_display_args may also move the environment strings to make
85               * extra room. Therefore this should be done as early as possible during
86               * startup, to avoid entanglements with code that might save a getenv()
87               * result pointer.
88               */
89              argv = save_ps_display_args(argc, argv);
90
(gdb)
```

78-88 行：这段注释描述了将初始给定的 argv[]数组的物理位置保存下来的目的。这样做是为了可能会使用到的 ps 显示。在一些平台上，argv[]存储空间必须被覆盖以设置 ps 的进程标题。在这种情况下，save_ps_display_args 函数创建并返回 argv[]数组的一个新副本。此外，save_ps_display_args 函数也可能会移动环境字符串以腾出额外的空间。因此，这个操作应该尽早在启动过程中完成，以避免与可能保存 getenv()结果指针的代码纠缠在一起。

89 行：调用 save_ps_display_args 函数来保存 argv 数组。这个函数的返回值赋给了 argv 变量。

3.8 初始化内存上下文

```
(gdb) list 91,99
91              /*
92               * Fire up essential subsystems: error and memory management
93               *
94               * Code after this point is allowed to use elog/ereport, though
95               * localization of messages may not work right away, and messages won't go
96               * anywhere but stderr until GUC settings get loaded.
97               */
98              MemoryContextInit();
99
(gdb)
```

91-96 行：这段注释描述了启动必要的子系统的过程，包括错误和内存管理。在这一点之后的代码可以使用 elog/ereport，尽管消息的本地化可能不会立即生效，并且消息直到 GUC 设置加载完成之前都只会输出到 stderr。

98 行：调用 MemoryContextInit()函数，初始化内存上下文。

3.9 初始化本地信息

```
(gdb) list 100,137
100             /*
101              * Set up locale information
102              */
103             set_pglocale_pgservice(argv[0], PG_TEXTDOMAIN("postgres"));
104
105             /*
106              * In the postmaster, absorb the environment values for LC_COLLATE and
107              * LC_CTYPE.  Individual backends will change these later to settings
108              * taken from pg_database, but the postmaster cannot do that.  If we leave
109              * these set to "C" then message localization might not work well in the
110              * postmaster.
111              */
112             init_locale("LC_COLLATE", LC_COLLATE, "");
113             init_locale("LC_CTYPE", LC_CTYPE, "");
114
115             /*
116              * LC_MESSAGES will get set later during GUC option processing, but we set
117              * it here to allow startup error messages to be localized.
118              */
119     #ifdef LC_MESSAGES
```

```
120             init_locale("LC_MESSAGES", LC_MESSAGES, "");
121     #endif
122
123             /*
124              * We keep these set to "C" always, except transiently in pg_locale.c; see
125              * that file for explanations.
126              */
127             init_locale("LC_MONETARY", LC_MONETARY, "C");
128             init_locale("LC_NUMERIC", LC_NUMERIC, "C");
129             init_locale("LC_TIME", LC_TIME, "C");
130
131             /*
132              * Now that we have absorbed as much as we wish to from the locale
133              * environment, remove any LC_ALL setting, so that the environment
134              * variables installed by pg_perm_setlocale have force.
135              */
136             unsetenv("LC_ALL");
137
(gdb)
```

100-109 行：设置地区信息。

通过调用 set_pglocale_pgservice 函数初始化了 PostgreSQL 的本地化设置，这对于错误消息和系统输出的本地化至关重要。

此外，它还指出了在 postmaster 进程中，需要吸收环境变量中的 LC_COLLATE 和 LC_CTYPE 设置。这是因为虽然单个后端（backends）将根据 pg_database 的设置改变这些本地化设置，postmaster 进程本身则不能这样做。如果这些设置保留为"C"，那么在 postmaster 进程中，消息的本地化可能无法正常工作。

112-113 行：调用 init_locale 函数设置 LC_COLLATE 和 LC_CTYPE。这两个函数调用传递了要设置的本地化类别、类别值以及一个空字符串，这可能意味着使用默认的环境变量值。

115-121 行：尽管 LC_MESSAGES 将在后续的 GUC 选项处理过程中被设置，但这里也对其进行了初始化，以允许启动错误消息的本地化。这是通过条件编译指令#ifdef LC_MESSAGES 实现的，以确保只在支持 LC_MESSAGES 的系统上执行这一操作。

123-129 行：对 LC_MONETARY、LC_NUMERIC 和 LC_TIME 进行了设置，将它们固定为"C"。这样做是为了确保与金钱、数字和时间相关的格式化和解析操作的一致性，避免因地区设置差异导致的问题。

131-136 行：在吸收了所需的地区环境信息后，代码通过调用 unsetenv 函数移除了

LC_ALL 设置。这是为了确保之前通过 pg_perm_setlocale 安装的环境变量能够生效。

3.10　检查与字符串转换相关的特定 bug

```
(gdb) list 138,139
138             check_strxfrm_bug();
139
(gdb)
```

138 行：调用 check_strxfrm_bug 函数。这个函数的调用是为了检查与字符串转换相关的特定 bug，确保系统的稳定性和一致性。

3.11　处理第 1 个参数（如--help、-?、--versopm、-V、）以及 root 用户执行的命令参数

```
(gdb) list 140,172
140             /*
141              * Catch standard options before doing much else, in particular before we
142              * insist on not being root.
143              */
144             if (argc > 1)
145             {
146                     if (strcmp(argv[1], "--help") == 0 || strcmp(argv[1], "-?") == 0)
147                     {
148                             help(progname);
149                             exit(0);
150                     }
151                     if (strcmp(argv[1], "--version") == 0 || strcmp(argv[1], "-V") == 0)
152                     {
153                             fputs(PG_BACKEND_VERSIONSTR, stdout);
154                             exit(0);
155                     }
156
157                     /*
158                      * In addition to the above, we allow "--describe-config" and "-C var"
159                      * to be called by root.  This is reasonably safe since these are
160                      * read-only activities.  The -C case is important because pg_ctl may
161                      * try to invoke it while still holding administrator privileges on
162                      * Windows.  Note that while -C can normally be in any argv position,
163                      * if you want to bypass the root check you must put it first.  This
```

```
164                     * reduces the risk that we might misinterpret some other mode's -C
165                     * switch as being the postmaster/postgres one.
166                     */
167                    if (strcmp(argv[1], "--describe-config") == 0)
168                            do_check_root = false;
169                    else if (argc > 2 && strcmp(argv[1], "-C") == 0)
170                            do_check_root = false;
171            }
172
(gdb)
```

140-155 行：这部分代码处理标准选项，如--help 和--version，在执行大多数其他操作之前进行。如果检测到这些选项之一，程序将分别显示帮助信息或版本信息，并随后退出。这是常见的命令行程序行为，允许用户快速获取程序信息而不需要执行更复杂的启动逻辑。

157-170 行：除了上述标准选项，代码还特别允许以 root 用户执行--describe-config 和-C var 选项，因为这些操作被认为是只读的，相对安全。尤其是在 Windows 上，pg_ctl 可能在保持管理员权限的情况下尝试调用-C 选项。为了安全起见，如果要绕过 root 检查，则这些选项必须放在参数列表的首位。

3.12　确保不是 root 执行程序

```
(gdb) list 173,179
173             /*
174              * Make sure we are not running as root, unless it's safe for the selected
175              * option.
176              */
177             if (do_check_root)
178                     check_root(progname);
179
(gdb)
```

172-178 行：进行非 root 用户检查。出于安全考虑，PostgreSQL 不允许以 root 用户身份运行，除非是执行了上述的安全操作。如果 do_check_root 标志为 true，则调用 check_root 函数来确保当前用户不是 root。

3.13　处理第 1 个参数是--fork 的情况

```
(gdb) list 180,188
180             /*
181              * Dispatch to one of various subprograms depending on first argument.
182              */
183
184     #ifdef EXEC_BACKEND
185             if (argc > 1 && strncmp(argv[1], "--fork", 6) == 0)
186                     SubPostmasterMain(argc, argv);  /* does not return */
187     #endif
188
(gdb)
```

180-186 行：基于第一个参数，将控制权分派给不同的子程序。这里特别提到了 EXEC_BACKEND 宏定义下的情况，如果 argv[1] 以 "--fork" 开头，程序将执行 SubPostmasterMain 函数，这通常发生在特定的后台进程启动场景下。SubPostmasterMain 函数不返回，意味着它将接管进程执行。

3.14　Windows 平台初始化 Win32 信号机制

```
(gdb) list 189,199
189     #ifdef WIN32
190
191             /*
192              * Start our win32 signal implementation
193              *
194              * SubPostmasterMain() will do this for itself, but the remaining modes
195              * need it here
196              */
197             pgwin32_signal_initialize();
198     #endif
199
(gdb)
```

189-198 行：在 Windows 环境下（通过 WIN32 宏定义检测），初始化 Win32 信号机制。这是因为 Windows 平台上的信号处理与 Unix-like 系统有所不同，需要特定的初始化过程。注意，如果是通过 SubPostmasterMain 函数启动的子进程，那么它将为自己进行信号初始化，而主进程或其他模式下启动的进程则需要在这里进行信号初始化。

3.15　处理第 1 个参数参数（--boot、--describe、--single 或者不是这三者之一）

```
(gdb) list 200,214
200             if (argc > 1 && strcmp(argv[1], "--boot") == 0)
201                     AuxiliaryProcessMain(argc, argv);       /* does not return */
202             else if (argc > 1 && strcmp(argv[1], "--describe-config") == 0)
203                     GucInfoMain();                  /* does not return */
204             else if (argc > 1 && strcmp(argv[1], "--single") == 0)
205                     PostgresMain(argc, argv,
206                                  NULL,                                  /* no dbname */
207                                  strdup(get_user_name_or_exit(progname)));  /* does not return */
208             else
209                     PostmasterMain(argc, argv); /* does not return */
210             abort();                                        /* should not get here */
211     }
212
213
214
(gdb)
```

200-201 行：如果第一个命令行参数是"--boot"，则调用 AuxiliaryProcessMain 函数。这个函数是为辅助进程准备的，比如在系统初始化过程中需要的特殊后台进程。AuxiliaryProcessMain 函数不返回，表示它会接管进程的执行直到完成。

202-203 行：如果第一个命令行参数是"--describe-config"，则调用 GucInfoMain 函数。这个函数用于输出 PostgreSQL 配置参数的信息，同样不返回。这个功能允许用户查询 PostgreSQL 的配置参数而无需启动数据库。

204-207 行：如果第一个命令行参数是"--single"，则调用 PostgresMain 函数，但是这次调用以一种特殊模式执行，即单用户模式。在这种模式下，PostgreSQL 可以执行数据库维护任务或进行故障恢复。此函数调用不返回，NULL 作为数据库名（意味着在这个模式下不直接连接到特定数据库），并且使用当前用户的名称作为参数。

208-209 行：如果命令行参数不匹配任何特殊模式，则默认调用 PostmasterMain 函数。这是启动 PostgreSQL 常规操作模式的入口点，即作为一个后台守护进程运行，等待并处理客户端请求。此函数同样不返回。

210 行：abort()函数的调用表明，如果程序执行到这里，那么一定是出现了错误。理论

上，由于所有上述的函数调用都不应该返回，程序不应该执行到这一行。这是一种安全措施，确保在逻辑上可能的错误导致的异常情况下，程序能够被终止。

4　分析 MemoryContexInit()函数

MemoryContextInit()位于 src/backend/utils/mmgr/mcxt.c 文件中，作用是初始化内存上下文。

4.1 调试分析前的准备工作

```
[postgres@dbsvr bin]$ pg_ctl stop
```

4.2 启动跟踪调试函数 MemoryContextInit()

```
[postgres@dbsvr ~]$ cd /opt/db/pgsql/bin
[postgres@dbsvr bin]$ gdb postgres -q
Reading symbols from postgres...
(gdb) break MemoryContextInit
Breakpoint 1 at 0xb44029: file mcxt.c, line 101.
(gdb) run -D /opt/db/userdb/pgdata
Starting program: /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib64/libthread_db.so.1".

Breakpoint 1, MemoryContextInit () at mcxt.c:101
101             AssertState(TopMemoryContext == NULL);
(gdb)
```

4.3 函数 MemoryContextInit()的注释

```
(gdb) list 82,97
```

```
82      /*
83       * MemoryContextInit
84       *              Start up the memory-context subsystem.
85       *
86       * This must be called before creating contexts or allocating memory in
87       * contexts.  TopMemoryContext and ErrorContext are initialized here;
88       * other contexts must be created afterwards.
89       *
90       * In normal multi-backend operation, this is called once during
91       * postmaster startup, and not at all by individual backend startup
92       * (since the backends inherit an already-initialized context subsystem
93       * by virtue of being forked off the postmaster).  But in an EXEC_BACKEND
94       * build, each process must do this for itself.
95       *
96       * In a standalone backend this must be called during backend startup.
97       */
(gdb)
```

这段代码是 PostgreSQL 数据库初始化内存上下文子系统的关键部分，位于数据库启动过程中。内存上下文是 PostgreSQL 中用于管理内存分配的一种机制，允许有效地管理和隔离不同数据库操作和对象的内存使用。

82-97 行：介绍了 MemoryContextInit 函数的目的和使用场景。它必须在创建其他内存上下文或在内存上下文中分配内存之前调用。

在多后端（multi-backend）操作中，这个函数在 postmaster 启动时被调用一次，而在每个后端进程（在 EXEC_BACKEND 构建中）中则需要各自进行初始化。

对于独立后端，这个函数也必须在启动时调用。

4.4 定义了 MemoryContextInit 函数的开始部分

```
(gdb) list 98,102
98      void
99      MemoryContextInit(void)
100     {
101             AssertState(TopMemoryContext == NULL);
102
(gdb)
```

98-100 行：定义了 MemoryContextInit 函数的开始部分。

101 行：通过 AssertState 宏确认 TopMemoryContext（所有其他内存上下文的根）在这

一点上应该是 NULL，这是初始化过程的预期起始状态。

4.5 创建内存上下文 TopMemoryContext

```
(gdb) list 103,115
103             /*
104              * First, initialize TopMemoryContext, which is the parent of all others.
105              */
106             TopMemoryContext = AllocSetContextCreate((MemoryContext) NULL,
107                                                      "TopMemoryContext",
108                                                      ALLOCSET_DEFAULT_SIZES);
109
110             /*
111              * Not having any other place to point CurrentMemoryContext, make it point
112              * to TopMemoryContext.  Caller should change this soon!
113              */
114             CurrentMemoryContext = TopMemoryContext;
115
(gdb)
```

103-109 行：使用 AllocSetContextCreate 函数创建 TopMemoryContext。

110-114 行：TopMemoryContext 作为所有其他内存上下文的父上下文。

4.6 创建内存上下文 ErrorContext

```
(gdb) list 116,115
116             /*
117              * Initialize ErrorContext as an AllocSetContext with slow growth rate ---
118              * we don't really expect much to be allocated in it. More to the point,
119              * require it to contain at least 8K at all times. This is the only case
120              * where retained memory in a context is *essential* --- we want to be
121              * sure ErrorContext still has some memory even if we've run out
122              * elsewhere! Also, allow allocations in ErrorContext within a critical
123              * section. Otherwise a PANIC will cause an assertion failure in the error
124              * reporting code, before printing out the real cause of the failure.
125              *
126              * This should be the last step in this function, as elog.c assumes memory
127              * management works once ErrorContext is non-null.
128              */
129             ErrorContext = AllocSetContextCreate(TopMemoryContext,
130                                                  "ErrorContext",
131                                                  8 * 1024,
132                                                  8 * 1024,
```

```
133                                                 8 * 1024);
134             MemoryContextAllowInCriticalSection(ErrorContext, true);
135     }
136
(gdb)
```

注释部分：

116-120 行：提到 ErrorContext 被初始化为一个 AllocSetContext，这是一种内存上下文，具有慢速增长率。这是因为不预期在此上下文中会有大量的内存分配。更重要的是，这个上下文至少需要随时包含 8KB 的内存。这是唯一一个保留内存是必要的上下文，确保即使其他地方内存耗尽，ErrorContext 仍有一些内存可用。

121-124 行：在关键部分允许 ErrorContext 进行内存分配。如果不这样做，当系统 PANIC 时，错误报告代码中的断言失败会在打印失败的真正原因之前触发。

126-127 行：这应该是此函数的最后一步，因为一旦 ErrorContext 非空，elog.c 假定内存管理已经可以工作。

实现部分：

129-133 行：调用函数 AllocSetContextCreate，创建一个新的 AllocSetContext 内存上下文。这个函数接受几个参数：

- TopMemoryContext：这是父内存上下文，ErrorContext 将成为其子上下文。
- "ErrorContext"：新创建的内存上下文的名称。
- 三个 8 * 1024 参数：这些分别代表初始内存分配大小、最小内存块大小以及最大内存块大小，都设置为 8KB。这意味着，ErrorContext 将始终至少有 8KB 的内存可用，这对于保证即使在内存紧张的情况下也能处理错误是非常关键的。

134 行：MemoryContextAllowInCriticalSection 函数调用允许 ErrorContext 在关键区域（critical section）中进行内存分配。这是为了确保在系统崩溃或发生严重错误时，能够记录错误信息，即使是在最关键的代码执行期间。true 参数表明允许在关键部分进行分配。

135 行：函数 MemoryContextInit()结束。

5 分析 PostmasterMain 函数

PostmasterMain()位于 src/backend/postmaster/postmaster.c 文件中，主要负责：

（1）　初始化服务器环境

设置服务器进程的运行环境，包括信号处理、日志系统、配置参数的加载与解析等。

初始化内存管理、网络连接监听以及多种子系统，如事务日志、后台写入器（background writer）、自动清理进程（autovacuum）等。

（2）　监听数据库连接

侦听来自客户端的连接请求，这包括 TCP/IP 连接和/或 UNIX 域套接字连接。

对新的客户端连接请求，postmaster 会启动一个新的后端进程（或线程，在某些配置中）来处理该客户端的请求。

（3）　管理后台进程

管理数据库的后台工作进程，如写入器（writer）、清理器（cleaner）、自动清理（autovacuum）等。

监控子进程的健康状态，并在必要时重启这些进程。

（4）　安全性

实施安全策略，包括认证和授权，确保只有合法的用户可以连接到数据库。

监控和应对潜在的安全威胁，如拒绝服务攻击（通过限制连接数等方式）。

（5）　重新加载配置文件

监听配置变化，允许某些配置参数在不重启数据库的情况下更新，通过接收特定信号来重新加载配置文件（如 SIGHUP）。

（6）　日志管理

管理数据库日志记录的各个方面，包括错误日志、查询日志等，根据配置进行日志的旋转和归档。

（7）　关闭和清理

在接收到关闭数据库的信号后（如 SIGINT、SIGTERM），安全地关闭数据库，包括关闭所有客户端连接、将数据写回磁盘并关闭所有后台进程。

通过上述功能，PostmasterMain()确保 PostgreSQL 数据库服务器的稳定运行，处理客户端请求，并管理数据库的内部工作流程。这个函数基本上是 PostgreSQL 服务器运行的心脏，负责协调和管理数据库的主要活动。

5.1 启动调试

```
[postgres@dbsvr ~]$  cd /opt/db
[postgres@dbsvr db]$ rm -rf userdb
[postgres@dbsvr db]$ tar xf userdb.tar
[postgres@dbsvr db]$ cd /opt/db/pgsql/bin
[postgres@dbsvr bin]$ gdb postgres -q
Reading symbols from postgres...
(gdb) break PostmasterMain
Breakpoint 1 at 0x8ae04a: file postmaster.c, line 585.
(gdb) run -D /opt/db/userdb/pgdata
Starting program: /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib64/libthread_db.so.1".

Breakpoint 1, PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:585
585             char       *userDoption = NULL;
(gdb)
```

5.2 PostmasterMain 函数入口

```
(gdb) list 577,582
577     /*
578      * Postmaster main entry point
579      */
580     void
581     PostmasterMain(int argc, char *argv[])
582     {
(gdb)
```

577-579 行：注释说明了这一部分代码是 postmaster 的主要入口点。

580-581 行：定义了 PostmasterMain 函数，它接受命令行参数 argc 和 argv 作为输入。

5.3 局部变量声明

```
(gdb) list 583,589
```

```
583             int                     opt;
584             int                     status;
585             char       *userDoption = NULL;
586             bool            listen_addr_saved = false;
587             int                     i;
588             char       *output_config_variable = NULL;
589
(gdb)
```

583-588 行：声明了一些局部变量，包括：

- opt：用于解析命令行选项。
- status：用于存放各种状态或返回值。
- userDoption：用于保存-D 命令行选项（指定数据目录的路径）。
- listen_addr_saved：作为一个布尔标志指示监听地址是否已保存。
- i：用于循环。
- output_config_variable：用于保存输出配置变量的名称。

5.4 初始化进程全局变量

```
(gdb) list 590,593
590             InitProcessGlobals();
591
592             PostmasterPid = MyProcPid;
593
(gdb)
```

590 行：调用 InitProcessGlobals 函数，初始化进程全局变量。这可能包括设置进程级别的参数和状态。

592 行：将全局变量 PostmasterPid 设置为当前进程的 PID，MyProcPid 通常是用于存储当前进程 PID 的全局变量。

5.5 标识当前环境为 postmaster 环境

```
(gdb) list 594,595
594             IsPostmasterEnvironment = true;
595
(gdb)
```

594 行：设置 IsPostmasterEnvironment 为 true，标识当前环境为 postmaster 环境。这可能影响后续代码的执行路径，因为某些操作可能仅在 postmaster 进程中执行。

5.6 umask 设置

```
(gdb) list 596,605
596             /*
597              * We should not be creating any files or directories before we check the
598              * data directory (see checkDataDir()), but just in case set the umask to
599              * the most restrictive (owner-only) permissions.
600              *
601              * checkDataDir() will reset the umask based on the data directory
602              * permissions.
603              */
604             umask(PG_MODE_MASK_OWNER);
605
(gdb)
```

597-603 行：在检查数据目录（通过 checkDataDir 函数）之前，将 umask 设置为最严格的权限（只有所有者有权限）。这是为了避免在确认数据目录的权限设置之前意外创建具有过宽权限的文件或目录。checkDataDir 函数稍后将根据数据目录的实际权限重新设置 umask。

5.7 创建 PostmasterContext 内存上下文

```
(gdb) list 606,616
606             /*
607              * By default, palloc() requests in the postmaster will be allocated in
608              * the PostmasterContext, which is space that can be recycled by backends.
609              * Allocated data that needs to be available to backends should be
610              * allocated in TopMemoryContext.
```

```
611             */
612             PostmasterContext = AllocSetContextCreate(TopMemoryContext,
613                                                       "Postmaster",
614                                                       ALLOCSET_DEFAULT_SIZES);
615             MemoryContextSwitchTo(PostmasterContext);
616
(gdb)
```

606-611 行：说明了在 postmaster 中，默认的 palloc()请求（负责在 PostgreSQL 内存上下文中分配内存）将在 PostmasterContext 中进行分配，这是可以被后端回收使用的空间。需要对后端进程可见的分配数据应在 TopMemoryContext 中进行。

612-615 行：使用 AllocSetContextCreate 函数创建了 PostmasterContext 内存上下文，它是 TopMemoryContext 的一个子上下文，并使用默认的分配大小。

然后通过 MemoryContextSwitchTo 函数切换当前的内存上下文到 PostmasterContext。

5.8 获取可执行文件 postgres 的安装路径

```
(gdb) list 617,619
617             /* Initialize paths to installation files */
618             getInstallationPaths(argv[0]);
619
(gdb)
```

618 行：获取可执行文件 postgres 的安装路径。

5.9 信号处理

```
(gdb) list 620,692
620             /*
```

```
621             * Set up signal handlers for the postmaster process.
622             *
623             * In the postmaster, we use pqsignal_pm() rather than pqsignal() (which
624             * is used by all child processes and client processes).  That has a
625             * couple of special behaviors:
626             *
627             * 1. Except on Windows, we tell sigaction() to block all signals for the
628             * duration of the signal handler.  This is faster than our old approach
629             * of blocking/unblocking explicitly in the signal handler, and it should
630             * also prevent excessive stack consumption if signals arrive quickly.
631             *
632             * 2. We do not set the SA_RESTART flag.  This is because signals will be
633             * blocked at all times except when ServerLoop is waiting for something to
634             * happen, and during that window, we want signals to exit the select(2)
635             * wait so that ServerLoop can respond if anything interesting happened.
636             * On some platforms, signals marked SA_RESTART would not cause the
637             * select() wait to end.
638             *
639             * Child processes will generally want SA_RESTART, so pqsignal() sets that
640             * flag.  We expect children to set up their own handlers before
641             * unblocking signals.
642             *
643             * CAUTION: when changing this list, check for side-effects on the signal
644             * handling setup of child processes.  See tcop/postgres.c,
645             * bootstrap/bootstrap.c, postmaster/bgwriter.c, postmaster/walwriter.c,
646             * postmaster/autovacuum.c, postmaster/pgarch.c, postmaster/pgstat.c,
647             * postmaster/syslogger.c, postmaster/bgworker.c and
648             * postmaster/checkpointer.c.
649             */
650            pqinitmask();
651            PG_SETMASK(&BlockSig);
652
653            pqsignal_pm(SIGHUP, SIGHUP_handler);    /* reread config file and have
654                                                     * children do same */
655            pqsignal_pm(SIGINT, pmdie);     /* send SIGTERM and shut down */
656            pqsignal_pm(SIGQUIT, pmdie);    /* send SIGQUIT and die */
657            pqsignal_pm(SIGTERM, pmdie);    /* wait for children and shut down */
658            pqsignal_pm(SIGALRM, SIG_IGN);  /* ignored */
659            pqsignal_pm(SIGPIPE, SIG_IGN);  /* ignored */
660            pqsignal_pm(SIGUSR1, sigusr1_handler);  /* message from child process */
661            pqsignal_pm(SIGUSR2, dummy_handler);    /* unused, reserve for children */
662            pqsignal_pm(SIGCHLD, reaper);   /* handle child termination */
663
664    #ifdef SIGURG
665
666            /*
667             * Ignore SIGURG for now.  Child processes may change this (see
668             * InitializeLatchSupport), but they will not receive any such signals
669             * until they wait on a latch.
670             */
671            pqsignal_pm(SIGURG, SIG_IGN);   /* ignored */
```

```
672     #endif
673
674             /*
675              * No other place in Postgres should touch SIGTTIN/SIGTTOU handling.  We
676              * ignore those signals in a postmaster environment, so that there is no
677              * risk of a child process freezing up due to writing to stderr.  But for
678              * a standalone backend, their default handling is reasonable.  Hence, all
679              * child processes should just allow the inherited settings to stand.
680              */
681     #ifdef SIGTTIN
682             pqsignal_pm(SIGTTIN, SIG_IGN);  /* ignored */
683     #endif
684     #ifdef SIGTTOU
685             pqsignal_pm(SIGTTOU, SIG_IGN);  /* ignored */
686     #endif
687
688             /* ignore SIGXFSZ, so that ulimit violations work like disk full */
689     #ifdef SIGXFSZ
690             pqsignal_pm(SIGXFSZ, SIG_IGN);  /* ignored */
691     #endif
692
(gdb)
```

621-648 行：介绍了信号处理器设置的背景和考虑。

- 在 postmaster 进程中，使用 pqsignal_pm()而不是子进程和客户端进程中使用的 pqsignal()。
- pqsignal_pm()具有一些特殊行为：在非 Windows 系统上，通过 sigaction()在信号处理器执行期间阻塞所有信号，这比显式在信号处理器中阻塞/解除阻塞信号的旧方法更快，也能防止信号快速到达时导致过度的堆栈消耗。
- 此外，不设置 SA_RESTART 标志，因为信号始终被阻塞，除非 ServerLoop 等待某些事件发生，在此期间，希望信号能够中断 select(2)的等待，使 ServerLoop 能够响应任何有趣的事件。有些平台上，标记了 SA_RESTART 的信号不会导致 select()等待结束。注意，更改信号处理器设置时需要注意对子进程信号处理设置的副作用。

649-691 行：设置信号处理器。

- 使用 pqinitmask()和 PG_SETMASK()初始化信号掩码，阻塞某些信号。
- 为多种信号注册了不同的处理函数：
  - SIGHUP：通过 SIGHUP_handler 重新读取配置文件并通知子进程做相同操作。
  - SIGINT、SIGQUIT、SIGTERM：这些信号通过 pmdie 函数处理，它们会触发

关闭流程，SIGINT 和 SIGTERM 让系统尝试正常关闭，而 SIGQUIT 导致立即退出。

- SIGALRM 和 SIGPIPE：被忽略（SIG_IGN），不对这些信号做任何处理。
- SIGUSR1：通过 sigusr1_handler 处理，通常用于子进程间的消息传递。
- SIGUSR2：通过 dummy_handler 处理，保留给子进程使用，但在 postmaster 中不使用。
- SIGCHLD：通过 reaper 处理，管理子进程的终止。

- 对于特定的信号（如 SIGURG、SIGTTIN、SIGTTOU、SIGXFSZ），如果它们在平台上可用，也设置为被忽略。这些信号的处理在 postmaster 环境中通常不是必需的，而且忽略它们可以避免一些潜在问题（例如，SIGTTIN 和 SIGTTOU 可能导致后台进程因尝试写入标准输出而挂起）。

这一段代码的目的是确保 postmaster 进程能够正确响应或忽略各种系统信号，从而保持系统的稳定运行和正确响应外部事件或内部状态变化。

信号	信号处理函数	功能
SIGHUP	PostgresSigHupHandler	当配置文件发生变化时，产生 SIGHUP 信号。 服务进程收到此信号后，设置 ConfigReloadPending 为真，重新读取配置文件。
SIGINT	StatementCancelHandler	收到 SIGINT 信号后调用此函数，终止正在进行的查询操作。 若此时进程正在退出（proc_exit_inprogress 为真），则什么也不做; 否则，将标志位 InterruptPending 和 QueryCancelPending 设置为真，表明准备处理查询取消中断。
SIGTERM	die	用于终止当前事务。若此时进程正在退出，则什么也不做; 否则，将标志位 InterruptPending 和 ProcDiePending 设置为真，表明准备处理进程退出中断。 在单用户模式下，调用 ProcessInterrupts 退出。
SIGQUIT	quickdie / die	首先屏蔽其他信号，然后结束正在进行的工作并退出。
SIGALRM	handle_sig_alarm	处理 SIGALRM 信号，由等待锁的进程超时引发，如果存在死锁则将自己从锁等待队列中退出。
SIGPIPE	SIG_IGN	忽略对应的信号。
SIGUSR1	procsignal_sigusr1_handler	处理用户自定义信号。
SIGUSR2	SIG_IGN	忽略对应的信号。
SIGFPE	FloatExceptionHandler	调用 floating-point exception 函数报浮点数异常错误。
SIGCHLD	SIG_DFL	SIGCHLD 信号由 postmaster 进程接收，将信号重置为 0。

5.10　命令行选项解析

```
(gdb) list 693,881
693             /*
694              * Options setup
695              */
696             InitializeGUCOptions();
697
698             opterr = 1;
699
700             /*
701              * Parse command-line options.  CAUTION: keep this in sync with
702              * tcop/postgres.c (the option sets should not conflict) and with the
703              * common help() function in main/main.c.
704              */
705             while ((opt = getopt(argc, argv,
                                  "B:bc:C:D:d:EeFf:h:ijk:lN:nOPp:r:S:sTt:W:-:")) != -1)
706             {
707                     switch (opt)
708                     {
709                             case 'B':
710                                     SetConfigOption("shared_buffers", optarg,
                                                     PGC_POSTMASTER, PGC_S_ARGV);
711                                     break;
712
713                             case 'b':
714                                     /* Undocumented flag used for binary upgrades */
715                                     IsBinaryUpgrade = true;
716                                     break;
717
718                             case 'C':
719                                     output_config_variable = strdup(optarg);
720                                     break;
721
722                             case 'D':
723                                     userDoption = strdup(optarg);
724                                     break;
725
726                             case 'd':
727                                     set_debug_options(atoi(optarg), PGC_POSTMASTER,
                                                       PGC_S_ARGV);
728                                     break;
729
730                             case 'E':
731                                     SetConfigOption("log_statement", "all",
```

```
                                        PGC_POSTMASTER, PGC_S_ARGV);
732                             break;
733
734                     case 'e':
735                             SetConfigOption("datestyle", "euro",
                                        PGC_POSTMASTER, PGC_S_ARGV);
736                             break;
737
738                     case 'F':
739                             SetConfigOption("fsync", "false", PGC_POSTMASTER,
                                        PGC_S_ARGV);
740                             break;
741
742                     case 'f':
743                             if (!set_plan_disabling_options(optarg,
                                        PGC_POSTMASTER, PGC_S_ARGV))
744                             {
745                                     write_stderr("%s: invalid argument for option
                                                -f: \"%s\"\n",
746                                             progname, optarg);
747                                     ExitPostmaster(1);
748                             }
749                             break;
750
751                     case 'h':
752                             SetConfigOption("listen_addresses", optarg,
                                        PGC_POSTMASTER, PGC_S_ARGV);
753                             break;
754
755                     case 'i':
756                             SetConfigOption("listen_addresses", "*",
                                        PGC_POSTMASTER, PGC_S_ARGV);
757                             break;
758
759                     case 'j':
760                             /* only used by interactive backend */
761                             break;
762
763                     case 'k':
764                             SetConfigOption("unix_socket_directories", optarg,
                                        PGC_POSTMASTER, PGC_S_ARGV);
765                             break;
766
767                     case 'l':
768                             SetConfigOption("ssl", "true", PGC_POSTMASTER,
                                        PGC_S_ARGV);
769                             break;
770
771                     case 'N':
772                             SetConfigOption("max_connections", optarg,
                                        PGC_POSTMASTER, PGC_S_ARGV);
```

```
773                             break;
774
775                     case 'n':
776                             /* Don't reinit shared mem after abnormal exit */
777                             Reinit = false;
778                             break;
779
780                     case 'O':
781                             SetConfigOption("allow_system_table_mods", "true",
                                                PGC_POSTMASTER, PGC_S_ARGV);
782                             break;
783
784                     case 'P':
785                             SetConfigOption("ignore_system_indexes", "true",
                                                PGC_POSTMASTER, PGC_S_ARGV);
786                             break;
787
788                     case 'p':
789                             SetConfigOption("port", optarg, PGC_POSTMASTER,
                                                PGC_S_ARGV);
790                             break;
791
792                     case 'r':
793                             /* only used by single-user backend */
794                             break;
795
796                     case 'S':
797                             SetConfigOption("work_mem", optarg, PGC_POSTMASTER,
                                                PGC_S_ARGV);
798                             break;
799
800                     case 's':
801                             SetConfigOption("log_statement_stats", "true",
                                                PGC_POSTMASTER, PGC_S_ARGV);
802                             break;
803
804                     case 'T':
805
806                     /*
807                      * In the event that some backend dumps core, send SIGSTOP,
808                      * rather than SIGQUIT, to all its peers.  This lets the wily
809                      * post_hacker collect core dumps from everyone.
810                      */
811                             SendStop = true;
812                             break;
813
814                     case 't':
815                             {
816                                     const char *tmp =
                                                get_stats_option_name(optarg);
817
```

```
818                                     if (tmp)
819                                     {
820                                             SetConfigOption(tmp, "true",
                                                   PGC_POSTMASTER, PGC_S_ARGV);
821                                     }
822                                     else
823                                     {
824                                             write_stderr("%s: invalid argument for
                                                        option -t: \"%s\"\n",
825                                                        progname, optarg);
826                                             ExitPostmaster(1);
827                                     }
828                                     break;
829                             }
830
831                     case 'W':
832                             SetConfigOption("post_auth_delay", optarg,
                                                 PGC_POSTMASTER, PGC_S_ARGV);
833                             break;
834
835                     case 'c':
836                     case '-':
837                             {
838                                     char       *name,
839                                                *value;
840
841                                     ParseLongOption(optarg, &name, &value);
842                                     if (!value)
843                                     {
844                                             if (opt == '-')
845                                                 ereport(ERROR,
846                                                     (errcode(ERRCODE_SYNTAX_ERROR),
847                                                     errmsg("--%s requires a value",
848                                                               optarg)));
849                                             else
850                                                 ereport(ERROR,
851                                                   (errcode(ERRCODE_SYNTAX_ERROR),
852                                                    errmsg("-c %s requires a value",
853                                                    optarg)));
854                                     }
855
856                                     SetConfigOption(name, value, PGC_POSTMASTER,
                                                          PGC_S_ARGV);
857                                     free(name);
858                                     if (value)
859                                             free(value);
860                                     break;
861                             }
862
863                     default:
864                             write_stderr("Try \"%s --help\" for more
```

```
                                                information.\n",
865                                             progname);
866                                 ExitPostmaster(1);
867                         }
868                 }
869
870                 /*
871                  * Postmaster accepts no non-option switch arguments.
872                  */
873                 if (optind < argc)
874                 {
875                         write_stderr("%s: invalid argument: \"%s\"\n",
876                                                 progname, argv[optind]);
877                         write_stderr("Try \"%s --help\" for more information.\n",
878                                                 progname);
879                         ExitPostmaster(1);
880                 }
881
(gdb)
```

上面的代码主要处理了 PostgreSQL postmaster 进程的命令行选项解析和设置。

693-703 行：设置选项的前言注释，提醒开发者在修改时注意与 tcop/postgres.c 中的选项同步，以及与 main/main.c 中的 help()函数保持一致。

696 行：调用 InitializeGUCOptions 函数初始化 Grand Unified Configuration (GUC) 系统，这是 PostgreSQL 用于管理配置参数的系统。

698 行：设置 opterr 为 1，告诉 getopt 库在解析命令行选项时输出错误消息。

705 行：使用 getopt 函数解析命令行选项。这一行定义了接受的短选项字符串。

707-868 行：根据解析出的选项执行相应的操作。每个 case 对应一个命令行标志。

- 'B'：设置共享缓冲区的大小。
- 'b'：设置二进制升级标志。
- 'C'：准备输出一个配置变量的值。
- 'D'：指定数据库的数据目录。
- 'd'：设置调试级别。
- 'E'和'e'：分别开启所有 SQL 语句的日志和设置日期样式为欧洲样式。
- 'F'：关闭文件系统同步。

- 'f'：设置计划选项，用于调试或性能调优。
- 'h'：设置监听地址。
- 'i'：监听所有接口（较老的选项，等价于'h' '*'）。
- 'j'、'r'：特定模式下使用的选项，通常在交云模式或单用户模式下。
- 'k'：设置 UNIX 套接字目录。
- 'l'：启用 SSL。
- 'N'：设置最大连接数。
- 'n'：非正常退出时不重新初始化共享内存。
- 'O'、'P'：允许修改系统表，忽略系统索引。
- 'p'：设置服务端口。
- 'S'：设置工作内存。
- 's'：开启语句统计日志。
- 'T'：设置后端出错时发送的信号类型。
- 't'：启用特定的统计选项。
- 'W'：设置身份验证后的延迟。
- 'c'和'-'：处理长选项或带值的设置。

871-880 行：检查是否有非选项参数（optind < argc 检查）。Postmaster 不接受非选项参数，如果存在，则打印错误信息并退出。

整个循环逐步处理每个命令行参数，根据参数类型更新 PostgreSQL 的配置或执行特定操作。这是 PostgreSQL 初始化过程中关键的一步，它允许管理员在启动数据库时自定义行为和配置参数。

5.11　读取配置文件 postgresql.conf

```
(gdb) list 882,902
882             /*
883              * Locate the proper configuration files and data directory, and read
884              * postgresql.conf for the first time.
885              */
```

```
886             if (!SelectConfigFiles(userDoption, progname))
887                     ExitPostmaster(2);
888
889             if (output_config_variable != NULL)
890             {
891                     /*
892                      * "-C guc" was specified, so print GUC's value and exit.  No extra
893                      * permission check is needed because the user is reading inside the
894                      * data dir.
895                      */
896                     const char *config_val = GetConfigOption(output_config_variable,
897                                                              false, false);
898
899                     puts(config_val ? config_val : "");
900                     ExitPostmaster(0);
901             }
902
(gdb)
```

882-885 行：这部分代码注释说明了接下来的操作目的：定位正确的配置文件和数据目录，并首次读取 postgresql.conf 配置文件。

886-887 行：调用 SelectConfigFiles 函数，传入用户通过-D 选项指定的数据目录（如果有的话）和程序名称（progname）。这个函数试图确定正确的配置文件位置和数据目录。

如果失败（函数返回 false），则调用 ExitPostmaster 函数退出 postmaster 进程，退出代码为 2，表示配置文件或数据目录有问题。

889-901 行：检查是否指定了输出配置变量（通过-C 选项）。如果是，这部分代码的目的是打印一个给定的配置参数（GUC）的值并退出。

892-894 行：注释解释了当使用-C 选项指定 GUC 时，将会打印该 GUC 的值并退出。因为用户正在数据目录内部读取配置，所以不需要额外的权限检查。

896-897 行：调用 GetConfigOption 函数，传入用户想要查询的配置变量名（output_config_variable），该函数返回配置变量的当前值。false, false 参数表示不需要额外的处理，如不需要错误报告或不强制返回非 NULL 值。

899 行：使用 puts 函数打印配置变量的值。如果 config_val 为 NULL（即配置变量未设置或不存在），则打印空字符串。

900-901 行：打印完配置变量的值后，通过调用 ExitPostmaster 函数以退出代码 0 正常

退出 postmaster 进程。

这段代码允许 PostgreSQL 在启动时通过命令行选项-C 查询指定配置项的值，这可以用于脚本或命令行工具来检查 PostgreSQL 配置的当前状态，而不必启动完整的数据库实例。

5.12　检查数据目录

```
(gdb) list 903,905
903             /* Verify that DataDir looks reasonable */
904             checkDataDir();
905
(gdb)
```

904 行：checkDataDir()确保数据目录（DataDir）存在并且配置得当。

5.13　检查控制文件

```
(gdb) list 906,908
906             /* Check that pg_control exists */
907             checkControlFile();
908
(gdb)
```

907 行：checkControlFile()检查控制文件（pg_control）是否存在，这是 PostgreSQL 数据库的一个关键文件，包含数据库系统的状态信息。

5.14　将工作目录切换到数据目录

```
(gdb) list 909,911
```

```
909             /* And switch working directory into it */
910             ChangeToDataDir();
911
(gdb)
```

910 行：ChangeToDataDir()将工作目录切换到数据目录。这是重要的初始化步骤，确保接下来的操作都在正确的目录下进行。

5.15　检查 GUC 参数的组合配置是否有效

```
(gdb) list 912,911
912             /*
913              * Check for invalid combinations of GUC settings.
914              */
915             if (ReservedBackends >= MaxConnections)
916             {
917                     write_stderr("%s: superuser_reserved_connections (%d) must be less
                                      than max_connections (%d)\n",
918                                  progname,
919                                  ReservedBackends, MaxConnections);
920                     ExitPostmaster(1);
921             }
922             if (XLogArchiveMode > ARCHIVE_MODE_OFF && wal_level == WAL_LEVEL_MINIMAL)
923                     ereport(ERROR,
924                                     (errmsg("WAL archival cannot be enabled when wal_level
                                                is \"minimal\"")));
925             if (max_wal_senders > 0 && wal_level == WAL_LEVEL_MINIMAL)
926                     ereport(ERROR,
927                                     (errmsg("WAL streaming (max_wal_senders > 0) requires
                                                wal_level \"replica\" or \"logical\"")));
928
(gdb)
```

913-921 行：检查保留后端连接数（ReservedBackends）是否小于最大连接数（MaxConnections）。这是一个合理性检查，因为保留连接是为了确保超级用户（如数据库维护人员）即使在数据库达到最大连接数时也能连接到数据库。如果配置不当，会通过 write_stderr 输出错误信息并退出。

922-927 行：检查 WAL 归档和流复制的配置是否合理。

- 如果开启了 WAL 归档（XLogArchiveMode > ARCHIVE_MODE_OFF），则 wal_level

不能设置为 minimal，因为归档需要更完整的 WAL 记录。

- 如果设置了 max_wal_senders 大于 0（开启了 WAL 流复制），wal_level 也需要设置为 replica 或 logical 以支持复制所需的 WAL 信息级别。这些检查使用 ereport 报告错误，这可能会导致程序终止。

5.16　其他的一次性的内部合理性检查

```
(gdb) list 929,938
929             /*
930              * Other one-time internal sanity checks can go here, if they are fast.
931              * (Put any slow processing further down, after postmaster.pid creation.)
932              */
933             if (!CheckDateTokenTables())
934             {
935                     write_stderr("%s: invalid datetoken tables, please fix\n", progname);
936                     ExitPostmaster(1);
937             }
938
(gdb)
```

929-937 行：其他一次性的内部合理性检查可以放在这里，只要它们的处理速度很快。CheckDateTokenTables()函数检查日期标记表是否有效。这些表对于日期和时间的解析非常重要。如果检查失败，将输出错误信息并退出。

5.17　重置 getopt 库的状态

```
(gdb) list 939,947
939             /*
940              * Now that we are done processing the postmaster arguments, reset
941              * getopt(3) library so that it will work correctly in subprocesses.
942              */
943             optind = 1;
944     #ifdef HAVE_INT_OPTRESET
945             optreset = 1;                           /* some systems need this too */
946     #endif
947
```

```
(gdb)
```

939-946 行：这部分代码负责重置 getopt 库的状态，以便在子进程中正确地重新使用 getopt 函数解析命令行选项。

- optind 变量被重置为 1，这是因为 getopt 函数使用 optind 来跟踪下一个要处理的命令行参数的索引。
- 如果系统支持 optreset（由 HAVE_INT_OPTRESET 宏控制），则也将 optreset 设置为 1，进一步确保 getopt 的状态被重置。这一步是必要的，因为 postmaster 在启动过程中可能会启动多个子进程，这些子进程可能需要解析它们自己的命令行参数。

5.18　在日志级别 DEBUG3 下输出 postmaster 进程的初始环境变量

```
(gdb) list 948,964
948             /* For debugging: display postmaster environment */
949             {
950                     extern char **environ;
951                     char      **p;
952
953                     ereport(DEBUG3,
954                                     (errmsg_internal("%s: PostmasterMain: initial environment dump:",
955                                                                             progname)));
956                     ereport(DEBUG3,
957                                     (errmsg_internal("-----------------------------------------")));
958                     for (p = environ; *p; ++p)
959                             ereport(DEBUG3,
960                                             (errmsg_internal("\t%s", *p)));
961                     ereport(DEBUG3,
962                                     (errmsg_internal("-----------------------------------------")));
963             }
964
(gdb)
```

948-963 行：这段代码用于调试目的，它会在日志级别 DEBUG3 下输出 postmaster 进程的初始环境变量。通过遍历 environ 全局变量（这是一个字符串数组，每个元素都是一个环境变量的"键=值"对），并使用 ereport 函数输出每个环境变量。这可以帮助开发者或管理员了解在 postmaster 启动时环境变量的状态，可能对诊断启动问题有帮助。

5.19　为数据目录创建锁

```
(gdb) list 965,980
965             /*
966              * Create lockfile for data directory.
967              *
968              * We want to do this before we try to grab the input sockets, because the
969              * data directory interlock is more reliable than the socket-file
970              * interlock (thanks to whoever decided to put socket files in /tmp :-().
971              * For the same reason, it's best to grab the TCP socket(s) before the
972              * Unix socket(s).
973              *
974              * Also note that this internally sets up the on_proc_exit function that
975              * is responsible for removing both data directory and socket lockfiles;
976              * so it must happen before opening sockets so that at exit, the socket
977              * lockfiles go away after CloseServerPorts runs.
978              */
979             CreateDataDirLockFile(true);
980
(gdb)
```

965-977 行：创建数据目录锁文件(CreateDataDirLockFile)。这是为了防止多个 Postgres 实例尝试使用同一数据目录运行，这可能导致数据损坏。选择在尝试获取输入套接字之前执行此操作是因为数据目录的锁定机制比基于套接字文件的锁定（套接字文件可能位于如/tmp 这样的目录中）更为可靠。此外，此步骤还会设置一个 on_proc_exit 函数，用于在进程退出时移除数据目录和套接字的锁文件，因此必须在打开套接字之前执行，以确保退出时正确清理资源。

5.20　读控制文件

```
(gdb) list 981,991
981             /*
982              * Read the control file (for error checking and config info).
983              *
984              * Since we verify the control file's CRC, this has a useful side effect
985              * on machines where we need a run-time test for CRC support instructions.
986              * The postmaster will do the test once at startup, and then its child
```

```
(gdb) list 987,991
987             * processes will inherit the correct function pointer and not need to
988             * repeat the test.
989             */
990            LocalProcessControlFile(false);
991
(gdb)
```

981-990 行：读取控制文件(LocalProcessControlFile)。控制文件包含了数据库的关键状态信息，读取它可以进行错误检查和获取配置信息。此操作还会验证控制文件的 CRC 校验和，这对于需要在运行时测试 CRC 支持指令的机器来说是一个有用的副作用，因为它允许 postmaster 在启动时执行一次测试，然后其子进程可以继承正确的函数指针，无需重复测试。

5.21 注册应用启动器(ApplyLauncherRegister)

```
(gdb) list 992,999
992             /*
993              * Register the apply launcher.  Since it registers a background worker,
994              * it needs to be called before InitializeMaxBackends(), and it's probably
995              * a good idea to call it before any modules had chance to take the
996              * background worker slots.
997              */
998             ApplyLauncherRegister();
999
(gdb)
```

993-998 行：注册应用启动器(ApplyLauncherRegister)。这个步骤是为了设置后台工作进程，它需要在 InitializeMaxBackends()之前调用，确保在后台工作槽被占用之前进行注册，这对于维护系统稳定性和性能是必要的。

5.22 预加载库

```
(gdb) list 1000,1004
1000            /*
1001             * process any libraries that should be preloaded at postmaster start
1002             */
1003            process_shared_preload_libraries();
1004
```

```
(gdb)
```

1001-1003 行：处理在 postmaster 启动时应该预加载的库(process_shared_preload_libraries)。这允许在数据库系统启动时加载用户指定的共享库，这些库可能包含额外的功能或对现有功能进行修改或增强。预加载库是 PostgreSQL 灵活性和可扩展性的一个重要体现。

5.23 初始化 SSL 库

```
(gdb) list 1005,1015
1005            /*
1006             * Initialize SSL library, if specified.
1007             */
1008    #ifdef USE_SSL
1009            if (EnableSSL)
1010            {
1011                    (void) secure_initialize(true);
1012                    LoadedSSL = true;
1013            }
1014    #endif
1015
(gdb)
```

1005-1014 行：如果启用了 SSL（通过编译时定义的 USE_SSL 和运行时的配置选项 EnableSSL），调用 secure_initialize(true)初始化 SSL 库。这一步骤确保了如果数据库配置为支持 SSL 加密连接，相应的库和资源会被正确初始化。LoadedSSL = true;标记 SSL 库已加载，这对于后续处理 SSL 连接请求很重要。

5.24 计算 MaxBackends 的值

```
(gdb) list 1016,1021
1016            /*
1017             * Now that loadable modules have had their chance to register background
1018             * workers, calculate MaxBackends.
1019             */
```

```
1020            InitializeMaxBackends();
1021
(gdb)
```

1016-1020 行：在加载模块有机会注册它们的后台工作进程之后，计算 MaxBackends 的值。MaxBackends 是数据库能够支持的最大后端进程数（包括用户连接和系统进程），这个计算考虑了后台工作进程对系统资源的占用。

5.25　设置共享内存和信号量

```
(gdb) list 1022,1026
1022            /*
1023             * Set up shared memory and semaphores.
1024             */
1025            reset_shared();
1026
(gdb)
```

1022-1026 行：设置共享内存和信号量。reset_shared()函数重置或初始化数据库系统的共享内存区域，这对于多个数据库进程之间的通信和同步是必要的。

5.26　设置最大打开的文件数

```
(gdb) list 1027,1032
1027            /*
1028             * Estimate number of openable files.  This must happen after setting up
1029             * semaphores, because on some platforms semaphores count as open files.
1030             */
1031            set_max_safe_fds();
1032
(gdb)
```

1027-1031 行：估算可打开的文件数目。这在设置信号量之后进行，因为在某些平台上，信号量也会被计算为打开的文件之一。这是为了确保数据库不会因为超出操作系统允许的打开文件数限制而失败。

5.27　设置堆栈深度

```
(gdb) list 1033,1037
1033            /*
1034             * Set reference point for stack-depth checking.
1035             */
1036            (void) set_stack_base();
1037
(gdb)
```

1033-1036 行：设置用于堆栈深度检查的参考点。set_stack_base()函数记录了一个基准点，以便后续检测递归调用或深层函数调用是否可能导致堆栈溢出。

5.28　初始化用于监视 postmaster 进程生命周期的机制

```
(gdb) list 1038,1043
1038            /*
1039             * Initialize pipe (or process handle on Windows) that allows children to
1040             * wake up from sleep on postmaster death.
1041             */
1042            InitPostmasterDeathWatchHandle();
1043
(gdb)
```

1038-1042 行：初始化用于监视 postmaster 进程生命周期的机制。

InitPostmasterDeathWatchHandle()设置了一个机制，允许子进程在 postmaster 进程死亡时被唤醒。在 Windows 上，这可能通过一个进程句柄实现；在 Unix-like 系统上，通常是通过一个管道实现。

5.29　Windows 平台特有：初始化一个 I/O 完成端口

```
(gdb) list 1044,1054
1044    #ifdef WIN32
1045
1046            /*
1047             * Initialize I/O completion port used to deliver list of dead children.
```

```
1048            */
1049           win32ChildQueue = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 1);
1050           if (win32ChildQueue == NULL)
1051                   ereport(FATAL,
1052                                   (errmsg("could not create I/O completion port for child queue")));
1053   #endif
1054
(gdb)
```

1044-1053 行：这部分代码专门针对 Windows 平台，初始化一个 I/O 完成端口，用于传递已终止子进程的信息。CreateIoCompletionPort 函数创建了一个 I/O 完成端口，如果创建失败，将报告一个致命错误。这种机制是 Windows 特有的，利用它可以高效地管理子进程状态变化的通知。

5.30 在没有 fork 系统调用平台：初始化非默认的 GUC 参数

```
(gdb) list 1055,1069
1055    #ifdef EXEC_BACKEND
1056            /* Write out nondefault GUC settings for child processes to use */
1057            write_nondefault_variables(PGC_POSTMASTER);
1058
1059            /*
1060             * Clean out the temp directory used to transmit parameters to child
1061             * processes (see internal_forkexec, below).  We must do this before
1062             * launching any child processes, else we have a race condition: we could
1063             * remove a parameter file before the child can read it.  It should be
1064             * safe to do so now, because we verified earlier that there are no
1065             * conflicting Postgres processes in this data directory.
1066             */
1067            RemovePgTempFilesInDir(PG_TEMP_FILES_DIR, true, false);
1068    #endif
1069
(gdb)
```

1055-1067 行：在 EXEC_BACKEND 模式下（主要用于没有 fork()的平台，比如 Windows），这段代码负责写出非默认的 GUC 设置，以便子进程能够使用这些设置。然后清理用于传递参数给子进程的临时目录。这一步骤很重要，因为它防止了在启动子进程时可能出现的竞争条件，即在子进程读取这些参数文件之前被移除。

5.31　移除可能触发备用服务器晋升为主服务器的指示文件

```
(gdb) list 1070,1088
1070            /*
1071             * Forcibly remove the files signaling a standby promotion request.
1072             * Otherwise, the existence of those files triggers a promotion too early,
1073             * whether a user wants that or not.
1074             *
1075             * This removal of files is usually unnecessary because they can exist
1076             * only during a few moments during a standby promotion. However there is
1077             * a race condition: if pg_ctl promote is executed and creates the files
1078             * during a promotion, the files can stay around even after the server is
1079             * brought up to be the primary.  Then, if a new standby starts by using
1080             * the backup taken from the new primary, the files can exist at server
1081             * startup and must be removed in order to avoid an unexpected promotion.
1082             *
1083             * Note that promotion signal files need to be removed before the startup
1084             * process is invoked. Because, after that, they can be used by
1085             * postmaster's SIGUSR1 signal handler.
1086             */
1087            RemovePromoteSignalFiles();
1088
(gdb)
```

1070-1087 行：这部分代码负责移除可能触发备用服务器晋升为主服务器的指示文件。这是为了避免不希望的自动晋升行为，特别是在某些边缘情况下，比如 pg_ctl promote 命令在晋升过程中执行并创建了这些文件，但在服务器升级为主服务器后，这些文件仍然保留。如果使用新主服务器的备份启动了一个新的备用服务器，那么这些文件可能在服务器启动时存在，必须被移除以避免意外的晋升。

5.32　移除日志轮回指示文件

```
(gdb) list 1089,1054
1089            /* Do the same for logrotate signal file */
1090            RemoveLogrotateSignalFiles();
1091
(gdb)
```

1089-1090 行：执行与移除触发备用晋升信号文件相似的操作，但这次是针对日志轮换

指示文件。这同样是为了确保不会因为过时的指示文件而触发不期望的行为。

5.33　移除包含当前日志文件名的过时文件

```
(gdb) list 1092,1098
1092            /* Remove any outdated file holding the current log filenames. */
1093            if (unlink(LOG_METAINFO_DATAFILE) < 0 && errno != ENOENT)
1094                    ereport(LOG,
1095                                    (errcode_for_file_access(),
1096                                     errmsg("could not remove file \"%s\": %m",
1097                                                    LOG_METAINFO_DATAFILE)));
1098
(gdb)
```

1092-1097 行：移除包含当前日志文件名的过时文件。如果 unlink 函数调用失败，并且错误不是因为文件不存在（ENOENT），则记录一个日志消息。这一步骤确保了关于日志文件名的信息是最新的，避免了可能的混淆或错误。

5.34　初始化输入套接字

```
(gdb) list 1099,1203
1099            /*
1100             * Initialize input sockets.
1101             *
1102             * Mark them all closed, and set up an on_proc_exit function that's
1103             * charged with closing the sockets again at postmaster shutdown.
1104             */
1105            for (i = 0; i < MAXLISTEN; i++)
1106                    ListenSocket[i] = PGINVALID_SOCKET;
1107
(gdb)
```

1099-1108 行：初始化 ListenSocket 数组，将所有监听套接字标记为未使用状态(PGINVALID_SOCKET)。这是准备监听来自客户端的连接请求的前置步骤。

5.35　注册 on_proc_exit 回调函数 CloseServerPorts

```
(gdb) list 1108,1109
1108            on_proc_exit(CloseServerPorts, 0);
1109
(gdb)
```

1108-1109 行：注册 on_proc_exit 回调函数 CloseServerPorts，它会在 postmaster 进程退出时关闭所有打开的服务器端口。这是一种资源清理机制，确保即使在异常退出的情况下，资源也能被适当释放。

5.36　启动系统日志收集器进程（启动进程 postgres: logger）

```
(gdb) list 1110,1114
1110            /*
1111             * If enabled, start up syslogger collection subprocess
1112             */
1113            SysLoggerPID = SysLogger_Start();
1114
(gdb)
```

1111-1113 行：如果启用了系统日志收集器（syslogger），则启动它。SysLogger_Start 函数负责创建 syslogger 子进程，该进程负责收集和管理日志输出。

```
(gdb) break 1113
Breakpoint 2 at 0x8aea26: file postmaster.c, line 1113.
(gdb) c
Continuing.

Breakpoint 2, PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1113
1113            SysLoggerPID = SysLogger_Start();
(gdb) shell pgps
postgres  213103  213092  0 05:44 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
(gdb) next
[Detaching after fork from child process 213121]
2024-03-26 05:44:48.191 CST [213103] LOG:  redirecting log output to logging collector process
2024-03-26 05:44:48.191 CST [213103] HINT:  Future log output will appear in directory "log".
1126            if (!(Log_destination & LOG_DESTINATION_STDERR))
(gdb) shell pgps
postgres  213103  213092  0 05:44 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
postgres  213121  213103  0 05:44 ?        00:00:00 postgres: logger
（启动了 logger）
(gdb)
```

分析一下函数 SysLogger_Start()

5.37　调整日志输出设置

```
(gdb) list 1115,1141
1115            /*
1116             * Reset whereToSendOutput from DestDebug (its starting state) to
1117             * DestNone. This stops ereport from sending log messages to stderr unless
1118             * Log_destination permits.  We don't do this until the postmaster is
1119             * fully launched, since startup failures may as well be reported to
1120             * stderr.
1121             *
1122             * If we are in fact disabling logging to stderr, first emit a log message
1123             * saying so, to provide a breadcrumb trail for users who may not remember
1124             * that their logging is configured to go somewhere else.
1125             */
1126            if (!(Log_destination & LOG_DESTINATION_STDERR))
1127                    ereport(LOG,
1128                                    (errmsg("ending log output to stderr"),
1129                                     errhint("Future log output will go to log destination
\"%s\".",
1130                                                     Log_destination_string)));
1131
1132            whereToSendOutput = DestNone;
1133
1134            /*
1135             * Report server startup in log.  While we could emit this much earlier,
1136             * it seems best to do so after starting the log collector, if we intend
1137             * to use one.
1138             */
1139            ereport(LOG,
1140                            (errmsg("starting %s", PG_VERSION_STR)));
1141
(gdb)
```

1116-1132 行：在 postmaster 完全启动后，将 whereToSendOutput 从 DestDebug（其初始状态）设置为 DestNone，这意味着 ereport 将不再将日志消息发送到 stderr，除非 Log_destination 设置允许。如果正在禁用 stderr 日志输出，首先发出一条日志消息，说明将来的日志输出将被重定向。

1134-1140 行：在日志中报告服务器启动信息。通常，这个消息在日志收集器启动之后尽可能晚地发出，以确保日志消息被适当地捕获和管理。

5.38　建立网络监听端口

```
(gdb) list 1142,1203
1142            /*
1143             * Establish input sockets.
1144             */
1145            if (ListenAddresses)
1146            {
1147                    char       *rawstring;
1148                    List       *elemlist;
1149                    ListCell   *l;
1150                    int                     success = 0;
1151
1152                    /* Need a modifiable copy of ListenAddresses */
1153                    rawstring = pstrdup(ListenAddresses);
1154
1155                    /* Parse string into list of hostnames */
1156                    if (!SplitGUCList(rawstring, ',', &elemlist))
1157                    {
1158                            /* syntax error in list */
1159                            ereport(FATAL,
1160                                    (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
1161                                     errmsg("invalid list syntax in parameter \"%s\"",
1162                                     "listen_addresses")));
1163                    }
1164
1165                    foreach(l, elemlist)
1166                    {
1167                            char       *curhost = (char *) lfirst(l);
1168
1169                            if (strcmp(curhost, "*") == 0)
1170                                    status = StreamServerPort(AF_UNSPEC, NULL,
1171                                                              (unsigned short) PostPortNumber,
1172                                                              NULL,
1173                                                              ListenSocket, MAXLISTEN);
1174                            else
1175                                    status = StreamServerPort(AF_UNSPEC, curhost,
1176                                                              (unsigned short) PostPortNumber,
1177                                                              NULL,
1178                                                              ListenSocket, MAXLISTEN);
1179
1180                            if (status == STATUS_OK)
1181                            {
1182                                    success++;
1183                                    /*record the first successful host addr in lockfile */
1184                                    if (!listen_addr_saved)
1185                                    {
```

```
1186                                    AddToDataDirLockFile(LOCK_FILE_LINE_LISTEN_ADDR,
                                                             curhost);
1187                                    listen_addr_saved = true;
1188                            }
1189                    }
1190                    else
1191                            ereport(WARNING,
1192                              (errmsg("could not create listen socket for \"%s\"",
1193                                    curhost)));
1194            }
1195
1196            if (!success && elemlist != NIL)
1197                    ereport(FATAL,
1198                            (errmsg("could not create any TCP/IP sockets")));
1199
1200            list_free(elemlist);
1201            pfree(rawstring);
1202    }
1203
(gdb)
```

1142-1202 行：根据 ListenAddresses 配置建立输入套接字。这个配置指定了 postmaster 应该监听的地址。代码首先解析 ListenAddresses 字符串，然后对每个地址调用 StreamServerPort 函数尝试建立监听套接字。如果地址是"*"，则监听所有接口。对于每个成功创建的监听端口，如果这是首个成功的地址，则将其记录到数据目录的锁文件中。如果没有成功创建任何 TCP/IP 套接字，将报告一个致命错误。成功建立的套接字将用于接收来自客户端的连接请求。

5.39　支持 BONJOUR

```
(gdb) list 1204,1242
1204    #ifdef USE_BONJOUR
1205            /* Register for Bonjour only if we opened TCP socket(s) */
1206            if (enable_bonjour && ListenSocket[0] != PGINVALID_SOCKET)
1207            {
1208                    DNSServiceErrorType err;
1209
1210                    /*
1211                     * We pass 0 for interface_index, which will result in registering on
1212                     * all "applicable" interfaces.  It's not entirely clear from the
1213                     * DNS-SD docs whether this would be appropriate if we have bound to
```

```
    1214                     * just a subset of the available network interfaces.
    1215                     */
    1216                    err = DNSServiceRegister(&bonjour_sdref,
    1217                                                                    0,
    1218                                                                    0,
    1219                                                                    bonjour_name,
    1220                                                                    "_postgresql._tcp.",
    1221                                                                    NULL,
    1222                                                                    NULL,
    1223
pg_hton16(PostPortNumber),
    1224                                                                    0,
    1225                                                                    NULL,
    1226                                                                    NULL,
    1227                                                                    NULL);
    1228                    if (err != kDNSServiceErr_NoError)
    1229                            ereport(LOG,
    1230                                                    (errmsg("DNSServiceRegister() failed: error
code %ld",
    1231                                                                            (long) err)));
    1232
    1233                    /*
    1234                     * We don't bother to read the mDNS daemon's reply, and we expect
that
    1235                     * it will automatically terminate our registration when the socket
is
    1236                     * closed at postmaster termination.  So there's nothing more to be
    1237                     * done here.  However, the bonjour_sdref is kept around so that
    1238                     * forked children can close their copies of the socket.
    1239                     */
    1240            }
    1241    #endif
    1242
    (gdb)
```

在这部分代码中，PostgreSQL 支持通过 Bonjour（也称为零配置网络）注册服务，以便在局域网中自动发现 PostgreSQL 服务器。这是对数据库服务器网络服务的一个可选增强，特别有用于开发和测试环境，或者在需要在局域网内自动发现服务的情景下。

1204-1241 行：如果编译了 Bonjour 支持（通过 USE_BONJOUR 宏定义检查），并且配置了启用 Bonjour（通过 enable_bonjour 变量检查），则这段代码会注册 PostgreSQL 服务到 Bonjour。

1206 行：检查是否成功打开了 TCP 套接字（ListenSocket[0] != PGINVALID_SOCKET）。只有在至少有一个 TCP 监听套接字打开的情况下，才会尝试注册 Bonjour 服务。

1210-1227 行：调用 DNSServiceRegister 函数注册 Bonjour 服务。这个调用指定了服务

的名称（bonjour_name）、服务类型（_postgresql._tcp.）、端口号（PostPortNumber），以及其他一些参数指定为零或 NULL，表示使用默认设置。这里，pg_hton16 函数用于确保端口号以网络字节顺序传递。

1211-1214 行：对于 interface_index 参数传递 0，意味着在所有"适用"的接口上注册服务。这可能不完全符合仅绑定到部分网络接口的情况，但在大多数情况下应该是合适的。

1228-1231 行：检查 DNSServiceRegister 的返回值，如果不等于 kDNSServiceErr_NoError，则记录一条日志消息，报告 Bonjour 服务注册失败。

1233-1238 行：注释解释了不需要读取 mDNS 守护进程的回复，因为期望在 postmaster 终止时，mDNS 守护进程会自动终止服务注册。bonjour_sdref 保留下来，以便 fork 出的子进程可以关闭它们的套接字副本。

这部分代码展示了 PostgreSQL 如何整合现代网络发现技术，如 Bonjour，来简化数据库服务的发现和连接过程，尤其是在那些支持零配置网络的环境中。

5.40　设置 Unix 域套接字监听

```
(gdb) list 1243,1294
1243    #ifdef HAVE_UNIX_SOCKETS
1244            if (Unix_socket_directories)
1245            {
1246                    char       *rawstring;
1247                    List       *elemlist;
1248                    ListCell   *l;
1249                    int                     success = 0;
1250
1251                    /* Need a modifiable copy of Unix_socket_directories */
1252                    rawstring = pstrdup(Unix_socket_directories);
1253
1254                    /* Parse string into list of directories */
1255                    if (!SplitDirectoriesString(rawstring, ',', &elemlist))
1256                    {
1257                            /* syntax error in list */
1258                            ereport(FATAL,
1259                                            (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
1260                                             errmsg("invalid list syntax in parameter
\"%s\"",
1261                                                                    "unix_socket_directories")));
1262                    }
1263
```

```
    1264                    foreach(l, elemlist)
    1265                    {
    1266                            char       *socketdir = (char *) lfirst(l);
    1267
    1268                            status = StreamServerPort(AF_UNIX, NULL,
    1269                                                                                    (unsigned
short) PostPortNumber,
    1270                                                                                    socketdir,
    1271                                                                                    ListenSocket,
MAXLISTEN);
    1272
    1273                            if (status == STATUS_OK)
    1274                            {
    1275                                    success++;
    1276                                    /* record the first successful Unix socket in lockfile
*/
    1277                                    if (success == 1)
    1278
AddToDataDirLockFile(LOCK_FILE_LINE_SOCKET_DIR, socketdir);
    1279                            }
    1280                            else
    1281                                    ereport(WARNING,
    1282                                                    (errmsg("could not create Unix-domain
socket in directory \"%s\"",
    1283                                                                                    socketdir)));
    1284                    }
    1285
    1286                    if (!success && elemlist != NIL)
    1287                            ereport(FATAL,
    1288                                            (errmsg("could not create any Unix-domain
sockets")));
    1289
    1290                    list_free_deep(elemlist);
    1291                    pfree(rawstring);
    1292            }
    1293    #endif
    1294
    (gdb)
```

这段代码专门处理了在具有 Unix 套接字支持的系统上（通过 HAVE_UNIX_SOCKETS 宏定义检查）设置 Unix 域套接字监听。

1243-1293 行：如果配置了使用 Unix 套接字（Unix_socket_directories 不为空），则执行以下操作：

1244-1252 行：复制 Unix_socket_directories 字符串以便修改。因为原始字符串可能是常量或者不应该被修改，所以通过 pstrdup 函数创建它的副本。

1254-1262 行：解析 Unix_socket_directories 字符串，将其分割成目录列表。

SplitDirectoriesString 函数基于逗号分隔符进行分割。如果字符串格式不正确，将报告一个致命错误。

1264-1284 行：遍历目录列表，尝试在每个指定的目录中创建 Unix 域套接字。StreamServerPort 函数用于尝试创建监听套接字，其中 AF_UNIX 指定了使用 Unix 域协议，socketdir 指定了套接字文件所在的目录。

1273-1278 行：如果在某个目录成功创建了套接字，递增成功计数。如果这是第一个成功创建的 Unix 域套接字，则将其目录记录到数据目录的锁文件中。这有助于其他可能需要这些信息的 PostgreSQL 进程或工具。

1280-1283 行：如果尝试创建套接字失败，则发出警告，指出在指定目录中无法创建 Unix 域套接字。

1286-1288 行：如果没有成功创建任何 Unix 域套接字（success 为 0 且 elemlist 不为空），则报告一个致命错误。这意味着尽管配置了 Unix 套接字，但系统无法在任何指定目录中创建它们。

1290-1291 行：释放解析目录列表所占用的内存，并释放复制的 Unix_socket_directories 字符串。

这段代码展示了 PostgreSQL 如何管理 Unix 域套接字的创建和配置，这是在支持 Unix 套接字的平台上进行本地客户端连接的一种高效方式。通过在指定目录下创建套接字文件，PostgreSQL 能够监听来自本地客户端的连接请求，同时也处理了可能的错误情况和配置问题。

```
(gdb) list 1295,1309
1295            /*
1296             * check that we have some socket to listen on
1297             */
1298            if (ListenSocket[0] == PGINVALID_SOCKET)
1299                    ereport(FATAL,
1300                                    (errmsg("no socket created for listening")));
1301
1302            /*
1303             * If no valid TCP ports, write an empty line for listen address,
1304             * indicating the Unix socket must be used.  Note that this line is not
1305             * added to the lock file until there is a socket backing it.
1306             */
1307            if (!listen_addr_saved)
```

```
1308                    AddToDataDirLockFile(LOCK_FILE_LINE_LISTEN_ADDR, "");
1309
(gdb)
```

1295-1300 行：这里检查是否至少创建了一个有效的监听套接字。如果没有（ListenSocket[0]等于 PGINVALID_SOCKET），则报告一个致命错误，因为这意味着没有任何方式可以接受客户端的连接请求。

1302-1308 行：如果没有为 TCP 端口保存监听地址（即 listen_addr_saved 为 false），则向数据目录的锁文件中添加一个空行。这表示 Unix 套接字必须被使用。这是为了确保在没有有效 TCP 监听端口时，仍然有明确的指示表明系统是通过 Unix 套接字接受连接。

5.41　记录 postmaster 选项

```
(gdb) list 1310,1316
1310            /*
1311             * Record postmaster options.  We delay this till now to avoid recording
1312             * bogus options (eg, unusable port number).
1313             */
1314            if (!CreateOptsFile(argc, argv, my_exec_path))
1315                    ExitPostmaster(1);
1316
```

1310-1315 行：记录 postmaster 选项到一个特定文件中。这一步被延后执行是为了避免记录错误的选项（例如，不可用的端口号）。CreateOptsFile 函数创建这个文件，如果失败，则退出 postmaster 进程。

5.42　写 pid 文件

```
(gdb) list 1317,1340
```

```
    1317            /*
    1318             * Write the external PID file if requested
    1319             */
    1320            if (external_pid_file)
    1321            {
    1322                    FILE       *fpidfile = fopen(external_pid_file, "w");
    1323
    1324                    if (fpidfile)
    1325                    {
    1326                            fprintf(fpidfile, "%d\n", MyProcPid);
    1327                            fclose(fpidfile);
    1328
    1329                            /* Make PID file world readable */
    1330                            if (chmod(external_pid_file, S_IRUSR | S_IWUSR | S_IRGRP |
S_IROTH) != 0)
    1331                                    write_stderr("%s: could not change permissions of
external PID file \"%s\": %s\n",
    1332                                                 progname, external_pid_file,
strerror(errno));
    1333                    }
    1334                    else
    1335                            write_stderr("%s: could not write external PID file
\"%s\": %s\n",
    1336                                         progname, external_pid_file,
strerror(errno));
    1337
    1338                    on_proc_exit(unlink_external_pid_file, 0);
    1339            }
    1340
    (gdb)
```

1317-1339 行：如果配置了外部 PID 文件（通过 external_pid_file 变量），则创建该文件并写入当前进程的 PID。这允许其他程序或脚本轻松地找到运行中的 PostgreSQL 主进程的 PID。文件权限被设置为全世界可读（S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH），以便不同的用户和服务可以读取 PID。如果无法创建或写入 PID 文件，将通过标准错误输出一条警告消息。同时，注册了一个 on_proc_exit 回调函数 unlink_external_pid_file，以确保在 PostgreSQL 主进程退出时自动删除外部 PID 文件，避免留下过时的信息。

5.43　清理旧的临时文件

包括清理旧的临时文件、

```
(gdb) list 1341,1346
1341            /*
1342             * Remove old temporary files.  At this point there can be no other
1343             * Postgres processes running in this directory, so this should be safe.
1344             */
1345            RemovePgTempFiles();
1346
(gdb)
```

1341-1345 行：在确认当前目录下没有其他 Postgres 进程运行的情况下，移除旧的临时文件。这一步是为了清理可能由于之前实例异常退出留下的临时文件，保证系统的干净启动。

5.44　初始化统计收集子系统（不会启动统计收集器进程）

```
(gdb) list 1347,1352
1347            /*
1348             * Initialize stats collection subsystem (this does NOT start the
1349             * collector process!)
1350             */
1351            pgstat_init();
1352
(gdb)
```

1348-1351 行：初始化统计收集子系统，但此时不启动统计收集器进程。统计收集子系统负责收集数据库操作的各种统计信息，帮助数据库管理员优化数据库性能和监控数据库健康。

5.45　自动清理（autovacuum）子系统（不会启动自动清理进程）

```
(gdb) list 1353,1357
1353            /*
1354             * Initialize the autovacuum subsystem (again, no process start yet)
1355             */
1356            autovac_init();
1357
```

```
(gdb)
```

1354-1356 行：初始化自动清理（autovacuum）子系统。同样，此步骤不会立即启动自动清理进程。自动清理是 PostgreSQL 的一个背景进程，用于自动执行清理任务，如清理过时的数据行版本，以保持数据库性能。

5.46　加载客户端认证配置文件

```
(gdb) list 1358,1379
1358            /*
1359             * Load configuration files for client authentication.
1360             */
1361            if (!load_hba())
1362            {
1363                    /*
1364                     * It makes no sense to continue if we fail to load the HBA file,
1365                     * since there is no way to connect to the database in this case.
1366                     */
1367                    ereport(FATAL,
1368                                    (errmsg("could not load pg_hba.conf")));
1369            }
1370            if (!load_ident())
1371            {
1372                    /*
1373                     * We can start up without the IDENT file, although it means that you
1374                     * cannot log in using any of the authentication methods that need a
1375                     * user name mapping. load_ident() already logged the details of
error
1376                     * to the log.
1377                     */
1378            }
1379
(gdb)
```

1359-1378 行：加载客户端认证配置文件（pg_hba.conf 和 pg_ident.conf）。这些文件对于数据库安全至关重要，因为它们定义了客户端连接到数据库时的认证方法和策略。如果 pg_hba.conf 加载失败，则报告致命错误并停止启动，因为没有它，数据库将无法安全地处理连接请求。如果 pg_ident.conf 加载失败，系统仍然可以启动，但某些需要用户名映射的认证方法将无法使用。

5.47　macOS 特殊检查

```
(gdb) list 1380,1397
1380    #ifdef HAVE_PTHREAD_IS_THREADED_NP
1381
1382            /*
1383             * On macOS, libintl replaces setlocale() with a version that calls
1384             * CFLocaleCopyCurrent() when its second argument is "" and every relevant
1385             * environment variable is unset or empty.  CFLocaleCopyCurrent() makes
1386             * the process multithreaded.  The postmaster calls sigprocmask() and
1387             * calls fork() without an immediate exec(), both of which have undefined
1388             * behavior in a multithreaded program.  A multithreaded postmaster is the
1389             * normal case on Windows, which offers neither fork() nor sigprocmask().
1390             */
1391            if (pthread_is_threaded_np() != 0)
1392                    ereport(FATAL,
1393                                    (errcode(ERRCODE_OBJECT_NOT_IN_PREREQUISITE_STATE),
1394                                     errmsg("postmaster became multithreaded during
startup"),
1395                                     errhint("Set the LC_ALL environment variable to a
valid locale.")));
1396    #endif
1397
(gdb)
```

1380-1396 行：这部分特别针对 macOS，检查 PostgreSQL 主进程（postmaster）是否意外地变成了多线程。由于 macOS 上的一个特殊情况，libintl 的行为可能导致 PostgreSQL 主进程变为多线程，这在使用 fork()和 sigprocmask()时是未定义行为。pthread_is_threaded_np()用于检测这种情况，如果检测到多线程，则报告致命错误。这是为了确保 PostgreSQL 能在一个安全和可预测的环境下启动，因为它的设计并不是为了在多线程环境下运行主进程。

5.48　记录 PostgreSQL 主进程（postmaster）的启动时间

```
(gdb) list 1398,1402
1398            /*
1399             * Remember postmaster startup time
1400             */
1401            PgStartTime = GetCurrentTimestamp();
1402
(gdb)
```

1398-1401 行：记录 PostgreSQL 主进程（postmaster）的启动时间，使用 GetCurrentTimestamp 函数获取当前时间戳。

5.49 记录 postmaster 的状态

```
(gdb) list 1403,1408
1403            /*
1404             * Report postmaster status in the postmaster.pid file, to allow pg_ctl to
1405             * see what's happening.
1406             */
1407            AddToDataDirLockFile(LOCK_FILE_LINE_PM_STATUS, PM_STATUS_STARTING);
1408
(gdb)
```

1403-1407 行：在数据目录的锁文件（通常是 postmaster.pid）中记录 postmaster 的状态，这里设置状态为"starting"。这样做允许 pg_ctl 等工具观察到数据库启动过程中的状态，以便于监控和故障排查。

5.50 调用 StartupDataBase()函数开始启动数据库（实例恢复）

```
(gdb) list 1409,1416
1409            /*
1410             * We're ready to rock and roll...
1411             */
1412            StartupPID = StartupDataBase();
1413            Assert(StartupPID != 0);
1414            StartupStatus = STARTUP_RUNNING;
1415            pmState = PM_STARTUP;
1416
(gdb)
```

1410-1415 行：调用 StartupDataBase 函数开始启动数据库，这个过程包括恢复数据库到一致状态、准备接受连接等任务。

函数返回的是启动进程的 PID，且断言这个 PID 不为 0，表明启动进程已经成功创建。此外，设置了：

StartupStatus 变量，表示启动状态。

pmState 变量，表示 postmaster 的当前状态。

实际上，

#define StartupDataBase()　　StartChildProcess(StartupProcess)

```
(gdb) break 1412
Breakpoint 4 at 0x8af13b: file postmaster.c, line 1412.
(gdb) c
Continuing.

Breakpoint 4, PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1412
1412            StartupPID = StartupDataBase();
(gdb) shell pgps
postgres  217660  217374  0 06:50 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
postgres  217677  217660  0 06:51 ?        00:00:00 postgres: logger
(gdb)
(gdb) next
[Detaching after fork from child process 218690]
1413            Assert(StartupPID != 0);
(gdb) shell pgps
postgres  217660  217374  0 06:50 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
postgres  217677  217660  0 06:51 ?        00:00:00 postgres: logger
postgres  218690  217660  0 07:01 ?        00:00:00 [postgres] <defunct>
(gdb)
```

5.51　启动配置为在启动时创建的后台工作进程（background workers）

```
(gdb) list 1417,1419
1417            /* Some workers may be scheduled to start now */
1418            maybe_start_bgworkers();
1419
(gdb)
```

1418 行：调用 maybe_start_bgworkers 尝试启动那些配置为在数据库启动时运行的后台工作进程（background workers）。

```
(gdb) break 1418
Breakpoint 4 at 0x8af182: file postmaster.c, line 1418.
(gdb) c
Continuing.
Breakpoint 4, PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1418
1418            maybe_start_bgworkers();
(gdb) shell pgps
postgres  213337  213092  0 05:47 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
postgres  217105  213337  0 06:41 ?        00:00:00 postgres: logger
postgres  217106  213337  0 06:41 ?        00:00:00 [postgres] <defunct>
(gdb) next
1420            status = ServerLoop();
(gdb) shell pgps
postgres  213337  213092  0 05:47 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
postgres  217105  213337  0 06:41 ?        00:00:00 postgres: logger
postgres  217106  213337  0 06:41 ?        00:00:00 [postgres] <defunct>
(gdb)
```

5.52　执行 Serverloop()

```
(gdb) list 1420,1429
1420            status = ServerLoop();
1421
1422            /*
1423             * ServerLoop probably shouldn't ever return, but if it does, close down.
1424             */
1425            ExitPostmaster(status != STATUS_OK);
1426
1427            abort();                                        /* not reached */
1428    }
1429
(gdb)
```

1420-1425 行：进入 ServerLoop 函数，这是 postmaster 的主循环，负责接收新的连接请求、管理子进程、处理信号等。

理论上，ServerLoop 函数不应该返回；

如果它返回了，意味着出现了异常情况，此时会调用 ExitPostmaster 函数以退出 postmaster 进程。

传递给 ExitPostmaster 的参数基于 ServerLoop 的返回状态，如果不是 STATUS_OK，则以错误状态退出。

1427 行：abort()函数调用表示这一行代码不应该被达到。如果执行到这里，表明程序流程出现了未预期的行为，abort 将生成一个核心转储文件，便于调试。

```
(gdb) frame
#0  PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1420
1420            status = ServerLoop();
(gdb) step
ServerLoop () at postmaster.c:1671
1671            last_lockfile_recheck_time = last_touch_time = time(NULL);
(gdb) bt
#0  ServerLoop () at postmaster.c:1671
#1  0x00000000008af18c in PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1420
#2  0x00000000007b088f in main (argc=3, argv=0xe9d770) at main.c:209
(gdb) shell pgps
postgres  219033  219018  0 07:09 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
postgres  219048  219033  0 07:10 ?        00:00:00 postgres: logger
postgres  219266  219033  0 07:12 ?        00:00:00 [postgres] <defunct>
(gdb)
```

可以看到，此时还未有其它的 PostgreSQL 实例进程启动！

6　分析 StartChildProcess()函数

StartChildProcess 的目的是为 postmaster 启动一个辅助进程，并且根据传入的类型参数 type 确定启动哪种类型的子进程。所有类型的子进程最初都进入 AuxiliaryProcessMain 函数，该函数处理一些通用的设置。函数返回子进程的 PID，如果启动失败则返回 0。

函数 StartChildProcess 原型：

static pid_t

StartChildProcess(AuxProcType type)

其中：

```
typedef enum
{
    NotAnAuxProcess = -1,
    CheckerProcess = 0,
    BootstrapProcess,
    StartupProcess,
    BgWriterProcess,
    ArchiverProcess,
    CheckpointerProcess,
    WalWriterProcess,
    WalReceiverProcess,
    NUM_AUXPROCTYPES                /* Must be last! */
} AuxProcType;
```

```
#define StartupDataBase()           StartChildProcess(StartupProcess)
```

实际调用的是 StartChildProcess()函数

其他

```
#define StartArchiver()             StartChildProcess(ArchiverProcess)
#define StartBackgroundWriter() StartChildProcess(BgWriterProcess)
#define StartCheckpointer()         StartChildProcess(CheckpointerProcess)
#define StartWalWriter()        StartChildProcess(WalWriterProcess)
#define StartWalReceiver()          StartChildProcess(WalReceiverProcess)
```

```
(gdb) list 5433,5486
5433    /*
```

```
5434    * StartChildProcess -- start an auxiliary process for the postmaster
5435    *
5436    * "type" determines what kind of child will be started.  All child types
5437    * initially go to AuxiliaryProcessMain, which will handle common setup.
5438    *
5439    * Return value of StartChildProcess is subprocess' PID, or 0 if failed
5440    * to start subprocess.
5441    */
5442   static pid_t
5443   StartChildProcess(AuxProcType type)
5444   {
5445           pid_t           pid;
5446           char       *av[10];
5447           int                     ac = 0;
5448           char            typebuf[32];
5449
5450           /*
5451            * Set up command-line arguments for subprocess
5452            */
5453           av[ac++] = "postgres";
5454
5455   #ifdef EXEC_BACKEND
5456           av[ac++] = "--forkboot";
5457           av[ac++] = NULL;                        /* filled in by postmaster_forkexec */
5458   #endif
5459
5460           snprintf(typebuf, sizeof(typebuf), "-x%d", type);
5461           av[ac++] = typebuf;
5462
5463           av[ac] = NULL;
5464           Assert(ac < lengthof(av));
5465
5466   #ifdef EXEC_BACKEND
5467           pid = postmaster_forkexec(ac, av);
5468   #else                                                   /* !EXEC_BACKEND */
5469           pid = fork_process();
5470
5471           if (pid == 0)                           /* child */
5472           {
5473                   InitPostmasterChild();
5474
5475                   /* Close the postmaster's sockets */
5476                   ClosePostmasterPorts(false);
5477
5478                   /* Release postmaster's working memory context */
5479                   MemoryContextSwitchTo(TopMemoryContext);
5480                   MemoryContextDelete(PostmasterContext);
5481                   PostmasterContext = NULL;
5482
5483                   AuxiliaryProcessMain(ac, av);   /* does not return */
5484           }
```

```
5485    #endif                                                  /* EXEC_BACKEND */
5486
(gdb)
```

这段代码展示了 PostgreSQL 如何启动一个辅助进程。辅助进程是指那些不直接处理客户端连接的后台进程，例如写入前的 WAL 日志写进程（WAL writer）、后台写进程（background writer）和自动清理进程（autovacuum daemon）。

5433-5440 行：函数注释说明了 StartChildProcess 的目的是为 postmaster 启动一个辅助进程，并且根据传入的类型参数 type 确定启动哪种类型的子进程。所有类型的子进程最初都进入 AuxiliaryProcessMain 函数，该函数处理一些通用的设置。函数返回子进程的 PID，如果启动失败则返回 0。

5442-5464 行：定义了 StartChildProcess 函数。函数首先声明一些局部变量，包括子进程的 PID（pid）、一个命令行参数数组（av）和其计数器（ac），以及一个类型缓冲区（typebuf）来存储辅助进程的类型信息。

5453-5463 行：设置辅助进程的命令行参数。对于 EXEC_BACKEND 模式，特别添加了 --forkboot 参数，这是因为在某些系统上，PostgreSQL 通过模拟 fork（而非直接调用）来创建子进程，需要传递额外的参数来指示启动类型。

5466-5484 行：根据编译选项 EXEC_BACKEND，有两种不同的方式启动子进程。

在 EXEC_BACKEND 模式下，使用 postmaster_forkexec 函数创建子进程，它处理 EXEC_BACKEND 模式下的进程创建和初始化。

在非 EXEC_BACKEND 模式下，使用 fork_process 直接创建一个 fork。成功 fork 后，在子进程中（pid == 0），执行一系列初始化操作，包括关闭 postmaster 的套接字、释放 postmaster 的工作内存上下文，并最终调用 AuxiliaryProcessMain 进入辅助进程的主函数。

```
(gdb) list 5487,5539
5487            if (pid < 0)
5488            {
5489                    /* in parent, fork failed */
5490                    int                     save_errno = errno;
5491
5492                    errno = save_errno;
5493                    switch (type)
5494                    {
5495                            case StartupProcess:
```

```
5496                                    ereport(LOG,
5497                                            (errmsg("could not fork startup
process: %m")));
5498                                    break;
5499                            case ArchiverProcess:
5500                                    ereport(LOG,
5501                                            (errmsg("could not fork archiver
process: %m")));
5502                                    break;
5503                            case BgWriterProcess:
5504                                    ereport(LOG,
5505                                            (errmsg("could not fork background
writer process: %m")));
5506                                    break;
5507                            case CheckpointerProcess:
5508                                    ereport(LOG,
5509                                            (errmsg("could not fork checkpointer
process: %m")));
5510                                    break;
5511                            case WalWriterProcess:
5512                                    ereport(LOG,
5513                                            (errmsg("could not fork WAL writer
process: %m")));
5514                                    break;
5515                            case WalReceiverProcess:
5516                                    ereport(LOG,
5517                                            (errmsg("could not fork WAL receiver
process: %m")));
5518                                    break;
5519                            default:
5520                                    ereport(LOG,
5521                                            (errmsg("could not fork
process: %m")));
5522                                    break;
5523                    }
5524
5525                    /*
5526                     * fork failure is fatal during startup, but there's no need to choke
5527                     * immediately if starting other child types fails.
5528                     */
5529                    if (type == StartupProcess)
5530                            ExitPostmaster(1);
5531                    return 0;
5532            }
5533
5534            /*
5535             * in parent, successful fork
5536             */
5537            return pid;
5538    }
5539
(gdb)
```

5487 行：检查 fork 操作的结果。

- 如果 fork 失败，pid 将会小于 0。
  - 5489-5523 行：在 fork 失败的情况下，根据进程的类型（如启动进程、归档进程等）输出相应的错误消息，并进行适当的处理。
  - 5489 行：保存 errno 的值。
  - 5493-5523 行：根据进程的类型输出相应的错误消息，并根据是否是启动进程决定是否退出 postmaster 进程。

- 5532-5538 行：如果 fork 成功，在父进程中返回子进程的 PID。

7 分析 maybe_start_bgworkers(void)函数

```
(gdb) list 5989,6022
5989    /*
5990     * If the time is right, start background worker(s).
5991     *
5992     * As a side effect, the bgworker control variables are set or reset
5993     * depending on whether more workers may need to be started.
5994     *
5995     * We limit the number of workers started per call, to avoid consuming the
5996     * postmaster's attention for too long when many such requests are pending.
5997     * As long as StartWorkerNeeded is true, ServerLoop will not block and will
5998     * call this function again after dealing with any other issues.
5999     */
6000    static void
6001    maybe_start_bgworkers(void)
6002    {
6003    #define MAX_BGWORKERS_TO_LAUNCH 100
6004            int                     num_launched = 0;
6005            TimestampTz now = 0;
6006            slist_mutable_iter iter;
6007
6008            /*
```

```
6009            * During crash recovery, we have no need to be called until the state
6010            * transition out of recovery.
6011            */
6012           if (FatalError)
6013           {
6014                   StartWorkerNeeded = false;
6015                   HaveCrashedWorker = false;
6016                   return;
6017           }
6018
6019           /* Don't need to be called again unless we find a reason for it below */
6020           StartWorkerNeeded = false;
6021           HaveCrashedWorker = false;
6022
(gdb)
```

这段代码是 PostgreSQL 中用于启动后台工作进程的部分，主要用于检查是否需要启动后台工作进程，并在必要时启动它们。

5989-6022 行：这段代码是一个名为 maybe_start_bgworkers 的静态函数。它用于检查是否需要启动后台工作进程，并根据需要启动它们。

5990-5998 行：这段注释说明了函数的目的和行为。它们说明了在什么情况下需要启动后台工作进程，以及如何限制每次启动的工作进程的数量。

6000-6021 行：这段代码实现了函数的主要逻辑。

6003 行：定义了一个常量 MAX_BGWORKERS_TO_LAUNCH，用于限制每次启动的后台工作进程的数量。

6004-6007 行：声明了一些变量，包括已启动的后台工作进程数量 num_launched、当前时间 now、一个迭代器 iter。

6008-6016 行：在进行一些前置检查后，如果在崩溃恢复过程中（FatalError 为真）则不需要启动后台工作进程，直接返回。否则，将 StartWorkerNeeded 和 HaveCrashedWorker 设置为假。

6019-6021 行：最后，将 StartWorkerNeeded 和 HaveCrashedWorker 设置为假，表示不需要再次调用该函数。

```
(gdb) list 6023,6113
6023           slist_foreach_modify(iter, &BackgroundWorkerList)
6024           {
```

```
6025                    RegisteredBgWorker *rw;
6026
6027                    rw = slist_container(RegisteredBgWorker, rw_lnode, iter.cur);
6028
6029                    /* ignore if already running */
6030                    if (rw->rw_pid != 0)
6031                            continue;
6032
6033                    /* if marked for death, clean up and remove from list */
6034                    if (rw->rw_terminate)
6035                    {
6036                            ForgetBackgroundWorker(&iter);
6037                            continue;
6038                    }
6039
6040                    /*
6041                     * If this worker has crashed previously, maybe it needs to be
6042                     * restarted (unless on registration it specified it doesn't want to
6043                     * be restarted at all).  Check how long ago did a crash last happen.
6044                     * If the last crash is too recent, don't start it right away; let it
6045                     * be restarted once enough time has passed.
6046                     */
6047                    if (rw->rw_crashed_at != 0)
6048                    {
6049                            if (rw->rw_worker.bgw_restart_time == BGW_NEVER_RESTART)
6050                            {
6051                                    int                     notify_pid;
6052
6053                                    notify_pid = rw->rw_worker.bgw_notify_pid;
6054
6055                                    ForgetBackgroundWorker(&iter);
6056
6057                                    /* Report worker is gone now. */
6058                                    if (notify_pid != 0)
6059                                            kill(notify_pid, SIGUSR1);
6060
6061                                    continue;
6062                            }
6063
6064                            /* read system time only when needed */
6065                            if (now == 0)
6066                                    now = GetCurrentTimestamp();
6067
6068                            if (!TimestampDifferenceExceeds(rw->rw_crashed_at, now,
6069
rw->rw_worker.bgw_restart_time * 1000))
6070                            {
6071                                    /* Set flag to remember that we have workers to start
later */
6072                                    HaveCrashedWorker = true;
6073                                    continue;
```

```
    6074                            }
    6075                    }
    6076
    6077                    if (bgworker_should_start_now(rw->rw_worker.bgw_start_time))
    6078                    {
    6079                            /* reset crash time before trying to start worker */
    6080                            rw->rw_crashed_at = 0;
    6081
    6082                            /*
    6083                             * Try to start the worker.
    6084                             *
    6085                             * On failure, give up processing workers for now, but set
    6086                             * StartWorkerNeeded so we'll come back here on the next
iteration
    6087                             * of ServerLoop to try again.  (We don't want to wait,
because
    6088                             * there might be additional ready-to-run workers.)  We could
set
    6089                             * HaveCrashedWorker as well, since this worker is now marked
    6090                             * crashed, but there's no need because the next run of this
    6091                             * function will do that.
    6092                             */
    6093                            if (!do_start_bgworker(rw))
    6094                            {
    6095                                    StartWorkerNeeded = true;
    6096                                    return;
    6097                            }
    6098
    6099                            /*
    6100                             * If we've launched as many workers as allowed, quit, but
have
    6101                             * ServerLoop call us again to look for additional ready-to-
run
    6102                             * workers.  There might not be any, but we'll find out the
next
    6103                             * time we run.
    6104                             */
    6105                            if (++num_launched >= MAX_BGWORKERS_TO_LAUNCH)
    6106                            {
    6107                                    StartWorkerNeeded = true;
    6108                                    return;
    6109                            }
    6110                    }
    6111            }
    6112    }
    6113
    (gdb)
```

这段代码位于 PostgreSQL 的后台工作进程管理逻辑中，它负责检查和启动注册的后台工作进程（background workers）。

后台工作进程是数据库系统中执行特定任务的独立进程，例如自动清理（autovacuum）、统计信息收集等。

6023 行：使用 slist_foreach_modify 宏迭代后台工作进程列表中的每个后台工作进程。

6023-6038 行：遍历已注册的后台工作进程列表。

对于每个工作进程，如果它已经在运行（rw->rw_pid != 0），则跳过当前迭代。

如果工作进程被标记为需要终止（rw->rw_terminate），则从列表中移除并继续到下一个。

6040-6075 行：处理之前崩溃的工作进程。

如果一个工作进程之前崩溃过（rw->rw_crashed_at != 0），检查是否应该重启它。

如果工作进程配置为不重启（BGW_NEVER_RESTART），或者从上次崩溃到现在的时间还没有超过设定的重启等待时间，就跳过当前工作进程。

如果工作进程配置为不重启，并且存在通知进程 ID（bgw_notify_pid），会向该进程发送 SIGUSR1 信号。

6077-6098 行：判断是否应该现在启动工作进程。

首先重置崩溃时间（rw->rw_crashed_at = 0），尝试启动工作进程。

如果启动失败（do_start_bgworker 返回 false），设置标志 StartWorkerNeeded 为 true，这样在 ServerLoop 的下一个迭代中会再次尝试启动。

6099-6110 行：如果已经启动的工作进程数量达到了限制（MAX_BGWORKERS_TO_LAUNCH），同样设置 StartWorkerNeeded 为 true 并返回。这意味着即使还有工作进程准备就绪，也会等到下一个迭代再尝试启动它们。

这个循环的主要目的是确保所有需要运行的后台工作进程都有机会被启动，同时处理崩溃和需要延迟启动的情况。

StartWorkerNeeded 标志的使用确保了即使在某次迭代中不能启动某些工作进程，系统也会在未来重新尝试启动。这个机制保证了数据库能够有效地管理和利用后台工作进程来执行各种任务。

8　分析 ServerLoop()函数

整个函数是 PostgreSQL 数据库在后台运行时的心脏，负责监听新的客户端连接请求，管理数据库后台进程，以及执行一些定期维护任务，确保数据库的正常运行和数据的安全性。

8.1.1　ServerLoop()函数注释、定义

```
(gdb) list 1658,1664
1658    /*
1659     * Main idle loop of postmaster
1660     *
1661     * NB: Needs to be called with signals blocked
1662     */
1663    static int
1664    ServerLoop(void)
(gdb)
```

1658-1662. 注释说明这是 postmaster 的主空闲循环，需要在阻塞信号的情况下调用。

1663-1664. ServerLoop 函数定义为静态整型，无参数。

8.2 定义局部变量并初始化

```
(gdb) list 1665,1670
1665    {
1666            fd_set          readmask;
1667            int                     nSockets;
1668            time_t          last_lockfile_recheck_time,
1669                                    last_touch_time;
1670
(gdb)
```

1665-1669. 定义局部变量：

- readmask 为文件描述符集合，用于 select()函数；
- nSockets 表示监听套接字数量；

- last_lockfile_recheck_time：记录上次检查锁文件的时间。
- last_touch_time：上次 touch 套接字文件的时间。

```
(gdb) list 1671,1674
1671            last_lockfile_recheck_time = last_touch_time = time(NULL);
1672
1673            nSockets = initMasks(&readmask);
1674
(gdb)
```

1670-1671. 初始化 last_lockfile_recheck_time 和 last_touch_time 为当前时间。

1672-1673. 调用 initMasks()函数初始化 readmask 并返回监听的套接字数量。

8.3 死循环开始

```
(gdb) list 1675,1676
1675            for (;;)
1676            {
(gdb)
```

8.3.1 每次循环初始化

```
(gdb) list 1677,1680
1677                    fd_set          rmask;
1678                    int                     selres;
1679                    time_t          now;
1680
(gdb)
```

1676-1679. 在每次循环开始时定义新的局部变量：

- rmask 为临时文件描述符集合，用于当前循环的 select()调用；
- selres 用于存储 select()的返回结果；
- now 记录当前时间。

8.3.2　等待连接请求，将 readmask 复制到 rmask 为 select()调用准备

```
(gdb) list 1681,1692
1681                    /*
1682                     * Wait for a connection request to arrive.
1683                     *
1684                     * We block all signals except while sleeping. That makes it safe for
1685                     * signal handlers, which again block all signals while executing, to
1686                     * do nontrivial work.
1687                     *
1688                     * If we are in PM_WAIT_DEAD_END state, then we don't want to accept
1689                     * any new connections, so we don't call select(), and just sleep.
1690                     */
1691                    memcpy((char *) &rmask, (char *) &readmask, sizeof(fd_set));
1692
(gdb)
```

1681-1690：注释解释了等待连接请求到来的过程。

如果服务器处于 PM_WAIT_DEAD_END 状态，则不会接受新的连接，只是简单地睡眠一段时间。

1690 行：使用 memcpy 函数将 readmask 复制到 rmask，为 select()调用准备。

8.3.3　根据 PM_WAIT_DEAD_END 状态进行处理

```
(gdb) list 1693,1716
1693                    if (pmState == PM_WAIT_DEAD_END)
1694                    {
```

```
1695                            PG_SETMASK(&UnBlockSig);
1696
1697                            pg_usleep(100000L); /* 100 msec seems reasonable */
1698                            selres = 0;
1699
1700                            PG_SETMASK(&BlockSig);
1701                    }
1702                    else
1703                    {
1704                            /* must set timeout each time; some OSes change it! */
1705                            struct timeval timeout;
1706
1707                            /* Needs to run with blocked signals! */
1708                            DetermineSleepTime(&timeout);
1709
1710                            PG_SETMASK(&UnBlockSig);
1711
1712                            selres = select(nSockets, &rmask, NULL, NULL, &timeout);
1713
1714                            PG_SETMASK(&BlockSig);
1715                    }
1716
(gdb)
```

1692-1701. 如果处于 PM_WAIT_DEAD_END 状态，则暂时解除信号屏蔽，睡眠 100 毫秒，再次屏蔽所有信号。否则，将根据 select()的结果来处理。

1702-1715. 如果不在 PM_WAIT_DEAD_END 状态，则设置超时，调用 select()等待连接请求或超时，之后重新屏蔽信号。

8.3.4　检查 select()结果：select()失败且错误不是由中断引起

```
(gdb) list 1717,1728
1717                    /* Now check the select() result */
1718                    if (selres < 0)
1719                    {
1720                            if (errno != EINTR && errno != EWOULDBLOCK)
1721                            {
1722                                    ereport(LOG,
1723                                                    (errcode_for_socket_access(),
1724                                                     errmsg("select() failed in postmaster: %m")));
1725                                    return STATUS_ERROR;
1726                            }
1727                    }
1728
(gdb)
```

1717-1727：检查 select()的返回结果，如果 select()调用失败且错误不是由中断引起的，则记录日志并返回错误状态。

8.3.5　检查 select()结果：select()成功且有连接请求，遍历并创建子进程（服务器进程）

```
(gdb) list 1729,1760
1729                /*
1730                 * New connection pending on any of our sockets? If so, fork a child
1731                 * process to deal with it.
1732                 */
1733                if (selres > 0)
1734                {
1735                        int                     i;
1736
1737                        for (i = 0; i < MAXLISTEN; i++)
1738                        {
1739                                if (ListenSocket[i] == PGINVALID_SOCKET)
1740                                        break;
1741                                if (FD_ISSET(ListenSocket[i], &rmask))
1742                                {
1743                                        Port       *port;
1744
1745                                        port = ConnCreate(ListenSocket[i]);
1746                                        if (port)
1747                                        {
1748                                                BackendStartup(port);
1749
1750                                                /*
1751                                                 * We no longer need the open socket or port structure
1752                                                 * in this process
1753                                                 */
1754                                                StreamClose(port->sock);
1755                                                ConnFree(port);
1756                                        }
1757                                }
1758                        }
1759                }
1760
(gdb)
```

1728-1758. 如果 select()成功且有连接请求，则遍历所有监听套接字，为每个活跃的连接创建子进程来处理。

8.3.6　日志收集器进程的检查和启动

```
(gdb) list 1761,1764
1761                    /* If we have lost the log collector, try to start a new one */
1762                    if (SysLoggerPID == 0 && Logging_collector)
```

```
1763                        SysLoggerPID = SysLogger_Start();
1764
(gdb)
```

1759-1763: 日志收集器进程的检查和启动

如果当前没有运行日志收集器进程（SysLoggerPID 为 0）且日志收集功能开启（Logging_collector 为真），则尝试启动一个新的日志收集器进程。

8.3.7 后台写入进程和检查点进程的管理

```
(gdb) list 1765,1778
1765                    /*
1766                     * If no background writer process is running, and we are not in a
1767                     * state that prevents it, start one.  It doesn't matter if this
1768                     * fails, we'll just try again later.  Likewise for the checkpointer.
1769                     */
1770                    if (pmState == PM_RUN || pmState == PM_RECOVERY ||
1771                            pmState == PM_HOT_STANDBY)
1772                    {
1773                            if (CheckpointerPID == 0)
1774                                    CheckpointerPID = StartCheckpointer();
1775                            if (BgWriterPID == 0)
1776                                    BgWriterPID = StartBackgroundWriter();
1777                    }
1778
(gdb)
```

1765-1777: 后台写入进程和检查点进程的管理

检查系统的当前状态（pmState），如果系统处于运行（PM_RUN）、恢复（PM_RECOVERY）或热备（PM_HOT_STANDBY）状态之一，则进行以下操作：

如果没有检查点进程（CheckpointerPID 为 0），则尝试启动一个检查点进程。

如果没有后台写入进程（BgWriterPID 为 0），则尝试启动一个后台写入进程。

8.3.8　WAL 日志写进程的检查和启动

```
(gdb) list 1779,1786
1779                    /*
1780                     * Likewise, if we have lost the walwriter process, try to start a new
1781                     * one.  But this is needed only in normal operation (else we cannot
1782                     * be writing any new WAL).
1783                     */
1784                    if (WalWriterPID == 0 && pmState == PM_RUN)
1785                            WalWriterPID = StartWalWriter();
1786
(gdb)
```

1779-1785: WAL 日志写进程的检查和启动

如果当前没有 WAL 写入器进程(WalWriterPID 为 0)且系统处于运行状态(PM_RUN)，则尝试启动一个新的 WAL 写入器进程。

8.3.9　自动清理进程的管理

```
(gdb) list 1787,1786
1787                    /*
1788                     * If we have lost the autovacuum launcher, try to start a new one. We
1789                     * don't want autovacuum to run in binary upgrade mode because
1790                     * autovacuum might update relfrozenxid for empty tables before the
1791                     * physical files are put in place.
1792                     */
1793                    if (!IsBinaryUpgrade && AutoVacPID == 0 &&
1794                            (AutoVacuumingActive() || start_autovac_launcher) &&
1795                            pmState == PM_RUN)
1796                    {
1797                            AutoVacPID = StartAutoVacLauncher();
1798                            if (AutoVacPID != 0)
1799                                    start_autovac_launcher = false; /* signal processed */
1800                    }
1801
(gdb)
```

1787-1799: 自动清理进程的管理

如果不在二进制升级模式（IsBinaryUpgrade 为假）、没有自动清理启动器进程（AutoVacPID 为 0）、自动清理功能激活或者需要启动自动清理启动器

（start_autovac_launcher 为真）且系统处于运行状态，则尝试启动自动清理启动器进程。成功启动后，标记 start_autovac_launcher 为假，表示已处理启动信号。

8.3.10　统计信息收集器进程的检查和启动

```
(gdb) list 1802,1806
1802                    /* If we have lost the stats collector, try to start a new one */
1803                    if (PgStatPID == 0 &&
1804                            (pmState == PM_RUN || pmState == PM_HOT_STANDBY))
1805                            PgStatPID = pgstat_start();
1806
(gdb)
```

1802-1805: 统计信息收集器进程的检查和启动

如果当前没有统计信息收集器进程（PgStatPID 为 0）且系统处于运行或热备状态，则尝试启动一个统计信息收集器进程。

8.3.11　归档进程的管理

```
(gdb) list 1807,1810
1807                    /* If we have lost the archiver, try to start a new one. */
1808                    if (PgArchPID == 0 && PgArchStartupAllowed())
1809                            PgArchPID = StartArchiver();
1810
(gdb)
```

1807-1809: 归档进程的管理

如果当前没有归档进程（PgArchPID 为 0）且允许启动归档进程（PgArchStartupAllowed() 返回真），则尝试启动一个归档进程。

8.3.12　向自动清理启动器发送信号

```
(gdb) list 1811,1810
1811                    /* If we need to signal the autovacuum launcher, do so now */
1812                    if (avlauncher_needs_signal)
1813                    {
1814                            avlauncher_needs_signal = false;
1815                            if (AutoVacPID != 0)
1816                                    kill(AutoVacPID, SIGUSR2);
1817                    }
1818
(gdb)
```

1811-1817: 向自动清理启动器发送信号

如果需要向自动清理启动器发送信号（avlauncher_needs_signal 为真），且自动清理启动器进程存在，则向其发送 SIGUSR2 信号，并重置 avlauncher_needs_signal 标志。

8.3.13 WAL 日志接收器的条件启动

```
(gdb) list 1819,1822
1819                    /* If we need to start a WAL receiver, try to do that now */
1820                    if (WalReceiverRequested)
1821                            MaybeStartWalReceiver();
1822
(gdb)
```

1819-1821: WAL 接收器的条件启动

如果请求启动 WAL 接收器（WalReceiverRequested 为真），则尝试启动 WAL 接收器。

8.3.14 启动后台工作进程

```
(gdb) list 1823,1826
1823                    /* Get other worker processes running, if needed */
1824                    if (StartWorkerNeeded || HaveCrashedWorker)
1825                            maybe_start_bgworkers();
1826
(gdb)
```

1823-1825: 启动后台工作进程

如果需要启动后台工作进程（StartWorkerNeeded 为真）或存在崩溃的工作进程（HaveCrashedWorker 为真），则尝试启动后台工作进程。

8.3.15　线程检查（特定于编译选项）

```
(gdb) list 1827,1835
1827    #ifdef HAVE_PTHREAD_IS_THREADED_NP
1828
1829                    /*
1830                     * With assertions enabled, check regularly for appearance of
1831                     * additional threads.  All builds check at start and exit.
1832                     */
1833                    Assert(pthread_is_threaded_np() == 0);
1834    #endif
1835
(gdb)
```

1827-1834: 线程检查（特定于编译选项）

如果编译时定义了 HAVE_PTHREAD_IS_THREADED_NP，使用断言检查当前是否存在多线程执行，以确保多线程环境下的安全性。

8.3.16　时间更新和维护任务

```
(gdb) list 1836,1845
1836                    /*
1837                     * Lastly, check to see if it's time to do some things that we don't
1838                     * want to do every single time through the loop, because they're a
1839                     * bit expensive.  Note that there's up to a minute of slop in when
1840                     * these tasks will be performed, since DetermineSleepTime() will let
1841                     * us sleep at most that long; except for SIGKILL timeout which has
1842                     * special-case logic there.
1843                     */
1844                    now = time(NULL);
1845
(gdb)
```

1836-1843：时间更新和维护任务的概述

这部分注释解释了接下来的代码块涉及一些不需要在每次循环中执行的操作，因为这些操作可能比较耗时。为了减少开销，这些任务可能会被推迟执行，但有特定的超时逻辑（如 SIGKILL 的超时）是个例外。

1844：更新当前时间，为后续的定期任务检查提供基准。

8.3.17　强制关闭缓慢关闭的子进程

```
(gdb) list 1836,1845
1846                    /*
1847                     * If we already sent SIGQUIT to children and they are slow to shut
1848                     * down, it's time to send them SIGKILL.  This doesn't happen
1849                     * normally, but under certain conditions backends can get stuck while
1850                     * shutting down.  This is a last measure to get them unwedged.
1851                     *
1852                     * Note we also do this during recovery from a process crash.
1853                     */
1854                    if ((Shutdown >= ImmediateShutdown || (FatalError && !SendStop)) &&
1855                            AbortStartTime != 0 &&
1856                            (now - AbortStartTime) >= SIGKILL_CHILDREN_AFTER_SECS)
1857                    {
1858                            /* We were gentle with them before. Not anymore */
1859                            ereport(LOG,
1860                                            (errmsg("issuing SIGKILL to recalcitrant children")));
1861                            TerminateChildren(SIGKILL);
1862                            /* reset flag so we don't SIGKILL again */
1863                            AbortStartTime = 0;
1864                    }
1865
(gdb)
```

1846-1864：强制关闭缓慢关闭的子进程

如果系统已经尝试通过发送 SIGQUIT 信号让子进程优雅关闭，但子进程关闭过程缓慢或卡住，且满足以下条件之一，则向子进程发送 SIGKILL 信号强制关闭它们：

系统正在进行立即关机（Shutdown >= ImmediateShutdown）或遇到致命错误且未发送停止信号（FatalError && !SendStop）。

自从上次尝试终止子进程以来已经过了一定的时间（now - AbortStartTime >= SIGKILL_CHILDREN_AFTER_SECS）。

8.3.18　检查数据目录锁文件

```
(gdb) list 1866,1886
1866                /*
1867                 * Once a minute, verify that postmaster.pid hasn't been removed or
1868                 * overwritten.  If it has, we force a shutdown.  This avoids having
1869                 * postmasters and child processes hanging around after their database
1870                 * is gone, and maybe causing problems if a new database cluster is
1871                 * created in the same place.  It also provides some protection
1872                 * against a DBA foolishly removing postmaster.pid and manually
1873                 * starting a new postmaster.  Data corruption is likely to ensue from
1874                 * that anyway, but we can minimize the damage by aborting ASAP.
1875                 */
1876                if (now - last_lockfile_recheck_time >= 1 * SECS_PER_MINUTE)
1877                {
1878                        if (!RecheckDataDirLockFile())
1879                        {
1880                                ereport(LOG,
1881                                                (errmsg("performing immediate shutdown because data
directory lock file is invalid")));
1882                                kill(MyProcPid, SIGQUIT);
1883                        }
1884                        last_lockfile_recheck_time = now;
1885                }
1886
(gdb)
```

1866-1884：检查数据目录锁文件

每分钟检查一次 postmaster.pid 文件以确保其没有被删除或覆盖。这个文件是 PostgreSQL 用来确保只有一个 postmaster 进程运行在给定数据目录上的机制。如果文件被篡改，可能会导致数据损坏或其他严重问题。如果检查失败，则向自己发送 SIGQUIT 信号，触发立即关闭，以防数据目录被错误地访问或修改。

8.3.19　更新 Unix 套接字文件和锁文件的访问和修改时间

```
(gdb) list 1887,1886
1887                /*
1888                 * Touch Unix socket and lock files every 58 minutes, to ensure that
1889                 * they are not removed by overzealous /tmp-cleaning tasks.  We assume
1890                 * no one runs cleaners with cutoff times of less than an hour ...
1891                 */
1892                if (now - last_touch_time >= 58 * SECS_PER_MINUTE)
1893                {
1894                        TouchSocketFiles();
1895                        TouchSocketLockFiles();
1896                        last_touch_time = now;
1897                }
(gdb)
```

1886-1897：更新 Unix 套接字文件和锁文件的访问和修改时间（touch）

每 58 分钟对 Unix socket 文件和锁文件进行"触摸"（更新文件的访问和修改时间），以防这些文件被系统的清理任务错误地删除。这是基于假设，没有清理任务会在一小时内删除文件，因此每 58 分钟触摸这些文件可以保证它们在清理任务运行时不会被视为旧文件而被删除。

8.4 死循环结束

```
(gdb) list 1898,1898
1898            }
(gdb)
```

8.5 ServerLoop 函数结束

```
(gdb) list 1899,1900
1899    }
1900
(gdb)
```

1899.　ServerLoop 函数结束。

9　打点调试示例（启动 PostgreSQL 实例的所有进程）

```
[postgres@dbsvr ~]$   cd /opt/db
[postgres@dbsvr db]$  rm -rf userdb
[postgres@dbsvr db]$  tar xf userdb.tar
[postgres@dbsvr db]$  cd /opt/db/pgsql/bin
[postgres@dbsvr bin]$ gdb postgres -q
Reading symbols from postgres...
(gdb) break PostmasterMain
Breakpoint 1 at 0x8ae04a: file postmaster.c, line 585.
(gdb) run -D /opt/db/userdb/pgdata
```

```
Starting program: /opt/db/pg14/bin/postgres -D /opt/db/userdb/pgdata
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib64/libthread_db.so.1".

Breakpoint 1, PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:585
585             char       *userDoption = NULL;
(gdb)  list 1110,1114
1110            /*
1111             * If enabled, start up syslogger collection subprocess
1112             */
1113            SysLoggerPID = SysLogger_Start();
1114
(gdb) break 1113
Breakpoint 2 at 0x8aea26: file postmaster.c, line 1113.
(gdb) c
Continuing.

Breakpoint 2, PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1113
1113            SysLoggerPID = SysLogger_Start();
(gdb) shell pgps
postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
(gdb)  next
[Detaching after fork from child process 13940]
2024-03-27 22:07:40.333 CST [13922] LOG:  redirecting log output to logging collector
process
2024-03-27 22:07:40.333 CST [13922] HINT:  Future log output will appear in directory "log".
1126            if (!(Log_destination & LOG_DESTINATION_STDERR))
(gdb) shell pgps
postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
(gdb)
```

可以看到此时启动了 logger 进程。

```
(gdb) list 1409,1416
1409            /*
1410             * We're ready to rock and roll...
1411             */
1412            StartupPID = StartupDataBase();
1413            Assert(StartupPID != 0);
1414            StartupStatus = STARTUP_RUNNING;
1415            pmState = PM_STARTUP;
1416
(gdb) break 1412
Breakpoint 3 at 0x8af13b: file postmaster.c, line 1412.
(gdb) c
Continuing.
```

```
Breakpoint 3, PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1412
1412            StartupPID = StartupDataBase();
(gdb) shell pgps
postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
(gdb) next
[Detaching after fork from child process 229270]
1413            Assert(StartupPID != 0);
(gdb) shell pgps
postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
postgres   13970   13922  0 22:09 ?        00:00:00 [postgres] <defunct>
(gdb)
```

可以看到，此时由 StartupDataBase()函数生成了一个子进程（处于僵尸状态）

```
(gdb) list 1417,1419
1417            /* Some workers may be scheduled to start now */
1418            maybe_start_bgworkers();
1419
(gdb) break 1418
Breakpoint 3 at 0x8af182: file postmaster.c, line 1418.
(gdb) c
Continuing.

Breakpoint 4, PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1418
1418            maybe_start_bgworkers();
(gdb) shell pgps
postgres  230684  230673  0 09:56 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
postgres  230729  230684  0 09:57 ?        00:00:00 postgres: logger
postgres  230888  230684  0 10:00 ?        00:00:00 [postgres] <defunct>
(gdb)  next
1420            status = ServerLoop();
(gdb)  shell pgps
postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
postgres   13970   13922  0 22:09 ?        00:00:00 [postgres] <defunct>
(gdb)
```

可以观察到，maybe_start_bgworkers()函数没有生成子进程。

```
(gdb) step
```

```
ServerLoop () at postmaster.c:1671
1671            last_lockfile_recheck_time = last_touch_time = time(NULL);
(gdb) bt
#0  ServerLoop () at postmaster.c:1671
#1  0x00000000008af18c in PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1420
#2  0x00000000007b088f in main (argc=3, argv=0xe9d770) at main.c:209
(gdb)
```

现在已经进入了 ServerLoop()函数执行。

```
(gdb) break StartChildProcess
Breakpoint 4 at 0x8b3f18: file postmaster.c, line 5447.
(gdb) c
Continuing.

Breakpoint 4, StartChildProcess (type=CheckpointerProcess) at postmaster.c:5447
5447            int                     ac = 0;
(gdb) shell pgps
postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
(gdb) bt
#0  StartChildProcess (type=CheckpointerProcess) at postmaster.c:5447
#1  0x00000000008b159b in reaper (postgres_signal_arg=17) at postmaster.c:3055
#2  <signal handler called>
#3  __GI___pthread_sigmask (how=2, newmask=<optimized out>, oldmask=0x0) at
pthread_sigmask.c:44
#4  0x00007ffff7d2026d in __GI___sigprocmask (how=<optimized out>, set=<optimized out>,
oset=<optimized out>) at ../sysdeps/unix/sysv/linux/sigprocmask.c:25
#5  0x00000000008af780 in ServerLoop () at postmaster.c:1710
#6  0x00000000008af18c in PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1420
#7  0x00000000007b088f in main (argc=3, argv=0xe9d770) at main.c:209
(gdb)
```

可以观察到，僵尸进程消失了！

```
(gdb) c
Continuing.
[Detaching after fork from child process 227690]

Breakpoint 4, StartChildProcess (type=BgWriterProcess) at postmaster.c:5447
5447            int                     ac = 0;
(gdb) shell pgps
postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
postgres   14191   13922  0 22:11 ?        00:00:00 postgres: checkpointer
(gdb) bt
#0  StartChildProcess (type=BgWriterProcess) at postmaster.c:5447
#1  0x00000000008b15b5 in reaper (postgres_signal_arg=17) at postmaster.c:3057
#2  <signal handler called>
```

```
    #3  __GI___pthread_sigmask (how=2, newmask=<optimized out>, oldmask=0x0) at
pthread_sigmask.c:44
    #4  0x00007ffff7d2026d in __GI___sigprocmask (how=<optimized out>, set=<optimized out>,
oset=<optimized out>) at ../sysdeps/unix/sysv/linux/sigprocmask.c:25
    #5  0x00000000008af780 in ServerLoop () at postmaster.c:1710
    #6  0x00000000008af18c in PostmasterMain (argc=3, argv=0xe9d770) at postmaster.c:1420
    #7  0x00000000007b088f in main (argc=3, argv=0xe9d770) at main.c:209
    (gdb)
```

可以观察到，新增了进程 postgres: checkpoint

```
    (gdb) c
    Continuing.
    [Detaching after fork from child process 227708]

    Breakpoint 4, StartChildProcess (type=WalWriterProcess) at postmaster.c:5447
    5447            int                     ac = 0;
    (gdb) shell pgps
    postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
    postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
    postgres   14191   13922  0 22:11 ?        00:00:00 postgres: checkpointer
    postgres   15716   13922  0 22:33 ?        00:00:00 postgres: background writer
    (gdb)
```

可以观察到，新增了进程 postgres: background writer

```
    (gdb) c
    Continuing.
    [Detaching after fork from child process 15752]
    [Detaching after fork from child process 15753]

    Breakpoint 5, StartChildProcess (type=ArchiverProcess) at postmaster.c:5447
    5447            int                     ac = 0;
    (gdb) shell pgps
    postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
    postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
    postgres   14191   13922  0 22:11 ?        00:00:00 postgres: checkpointer
    postgres   15716   13922  0 22:33 ?        00:00:00 postgres: background writer
    postgres   15752   13922  0 22:34 ?        00:00:00 postgres: walwriter
    postgres   15753   13922  0 22:34 ?        00:00:00 postgres: autovacuum launcher
    (gdb)
```

可以观察到，新增了进程 postgres: walwriter 和进程 postgres: autovacuum launcher

```
    (gdb) c
    Continuing.
```

```
[Detaching after fork from child process 15944]
[Detaching after fork from child process 15945]
[Detaching after fork from child process 15946]

Program received signal SIGUSR1, User defined signal 1.
0x00007ffff7de1989 in __GI___select (nfds=9, readfds=0x7fffffffe780, writefds=0x0,
exceptfds=0x0, timeout=0x7fffffffe800) at
../sysdeps/unix/sysv/linux/select.c:69
69      int r = SYSCALL_CANCEL (pselect6_time64, nfds, readfds, writefds, exceptfds,
(gdb) shell pgps
postgres   13922   13906  0 22:06 pts/0    00:00:00 /opt/db/pg14/bin/postgres -D
/opt/db/userdb/pgdata
postgres   13940   13922  0 22:07 ?        00:00:00 postgres: logger
postgres   14191   13922  0 22:11 ?        00:00:00 postgres: checkpointer
postgres   15716   13922  0 22:33 ?        00:00:00 postgres: background writer
postgres   15752   13922  0 22:34 ?        00:00:00 postgres: walwriter
postgres   15753   13922  0 22:34 ?        00:00:00 postgres: autovacuum launcher
postgres   15944   13922  0 22:35 ?        00:00:00 postgres: archiver last was
000000010000000000000002
postgres   15945   13922  0 22:35 ?        00:00:00 postgres: stats collector
postgres   15946   13922  0 22:35 ?        00:00:00 postgres: logical replication launcher
(gdb)
```

可以观察到，新增了：

- 进程 postgres: archiver、
- 进程 postgres: stats collector、
- 进程 logical replication launcher

```
(gdb) quit
(gdb) quit
A debugging session is active.

        Inferior 1 [process 13922] will be killed.

Quit anyway? (y or n) y
[postgres@dbsvr bin]$
```

10 打点调试示例（启动 PostgreSQL 实例，跟踪内存数据结构的初始化）

请同学们完成