PostgreSQL 解析資料

～ postmaster ～

(株) NTT データ
ビジネス開発事業本部 システム方式技術 BU
井久保 寛明

1. はじめに

本資料では、postmaster の実装について紹介する。主に postmaster の起動時の処理や、起動後に他のプロセスからリクエストを受けてそれらの処理を開始する方法について説明する。postmaster から振り分けられた後の処理については、本資料では概要にとどめる。振り分けられた後の処理の詳細については、他の資料でまとめることとする。

1.1. 対象バージョン

本資料は、PostgreSQL7.4.2 を対象にソースコードの調査を行ったものである。従って、他のバージョンでは、内容が異なる場合があるので注意して頂きたい。

1.2. 用語の説明

一部、本資料で使う用語について説明しておく。

ＤＢのスタートアップ	ディスク上のデータベースファイルをオープンして、バックエンドプロセス postgres からそれらを利用できるようにする処理。データベースのオープンの際に、必要に応じてリカバリ処理が実行される。postmaster の起動処理とは使い分けている。
ＤＢのシャットダウン	ディスク上のデータベースファイルをクローズする処理。postmaster の終了処理全体のことではない。
チェックポイント	データバッファ上のダーティページをフラッシュすることで、リカバリの際に古いログを使わなくていいようにするための処理。

2. postmaster の基礎

postmaster は、次のような仕事を行う。

- 初期化処理
  - ✧ 各種マネージャの初期化（メモリマネージャ、バッファマネージャ、etc...）
  - ✧ ＤＢのスタートアップ処理（リカバリ処理を含む）を行うプロセスの起動
- フロントエンドに対する仕事
  - ✧ クライアントからの新規接続要求の受付と、そのクライアントの処理を担当するバック

エンドプロセスの起動
- ✧ クライアントからのクエリのキャンセル処理の仲介
- ✧ pg_ctl によるシャットダウン要求の受付

・ プロセス管理
- ✧ バックエンドプロセスの管理
- ✧ チェックポイント処理プロセスの起動
- ✧ 統計情報収集プロセス（statistics collector）の起動と監視

・ 終了処理
- ✧ ＤＢのシャットダウン処理プロセスの起動

・ その他
- ✧ バックエンドがクラッシュした際のシステム全体のクリーンナップ処理[1]

これらの処理のうち、通常時のサービスのイメージを表したのが図１である。

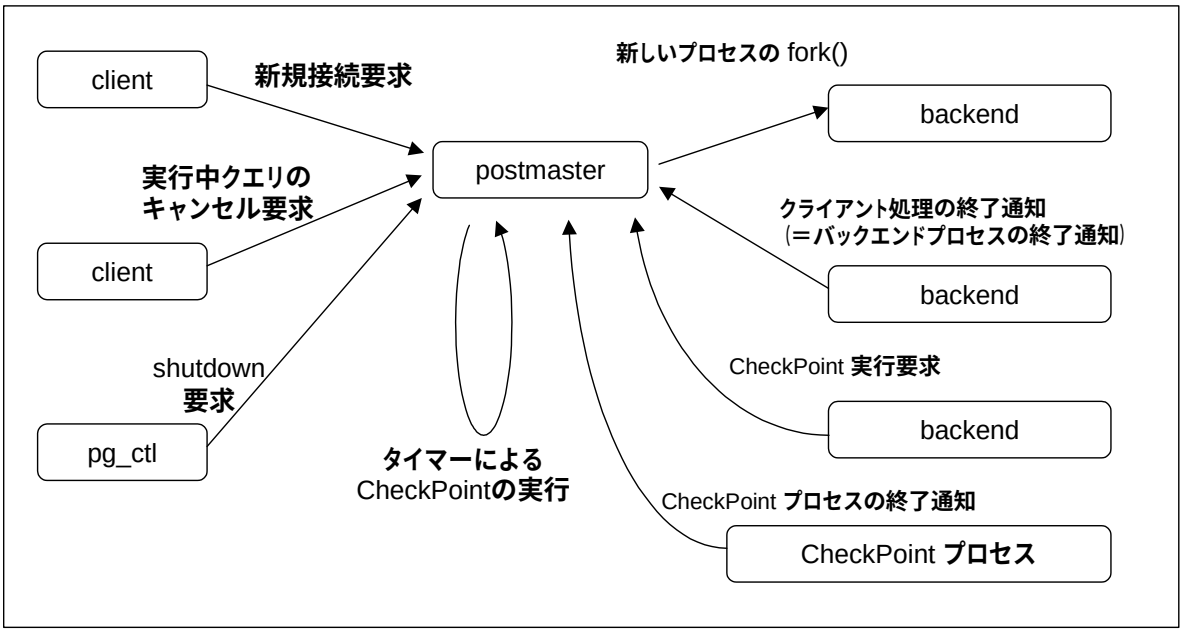

図 2.1 通常時の postmaster の主な処理

2.1. postmaster の設計思想 （耐障害性の設計）

postmaster は様々な処理を行うが、基本的に postmaster 自身はなるべくシステムのリソースとは切り離された作りになっている。クライアントの要求に対する処理は、プロセスを fork() して子プロセス（postgres）で行う。また、ＤＢのスタートアップ、ＤＢのシャットダウン、チェックポイントなどの基本的な処理でさえ、子プロセスを起動して別プロセスで行う。共有メモリやセマフォについては、postmaster の起動時に作成と初期化の処理は行うが、postmaster 自身が共有メモリやセマフォを操作することはない。

これらの工夫によって、postmaster が特定の処理に占有されることがないようになっていると同時に、

[1] PostgreSQLでは共有メモリを使っているため、１つのバックエンドが異常終了した場合、共有メモリの情報を破壊している可能性があるのでリカバリ処理を行う必要がある。

postmaster で障害が発生する機会が少なくなるようになっている。

2.2. postmaster の処理の流れの概要

postmaster の処理の流れを大まかに説明すると次のようになる。

1. 初期化
   - 各マネージャの初期化
   - ＤＢのスタートアップ処理（リカバリ処理を含む）を行う子プロセスの起動
2. サーバループで、各処理の振り分けを行う
   - クライアントの受付（受付けたらすぐに子プロセスを起動）
   - 各割り込み処理の実行
   - チェックポイントやＤＢのシャットダウン処理を実行する子プロセスの起動

初期化にあたる部分が、PostmasterMain() 関数の中で行われ、サーバループの部分は ServerLoop() 関数として実装されている。
サーバループの中では、ソケットを使ってクライアントの接続要求を待ち、接続要求があるとバックエンドプロセスを起動する。そして、クライアントの要求は直接バックエンドプロセスで処理する。

ソースコードで見ていくと、主な関数のシーケンスは次のようになっている。

```
main()
  ├ PostmasterMain()
  │   ├ MemoryContextInit();
  │   ├ InitializeGUCOptions();
  │   ├ [GUCと引数の処理]
  │   ├ ProcessConfigFile(PGC_POSTMASTER);
  │   ├ pmdaemonize() : -S が指定された時のみ
  │   ├ CreateDataDirLockFile(DataDir, true);
  │   ├ reset_shared() --> CreateSharedMemoryAndSemaphores()
  │   ├ [ソケットの初期化]
  │   ├ [シグナルの処理]
  │   ├ [統計情報収集プロセスの初期化と起動] pgstat_init(); pgstat_start(); <postmaster/pgstat.c>
  │   ├ StartupDataBase() --> SSDataBase(BS_XLOG_STARTUP)
  │   │                       * fork()して、子プロセスで --> BootstrapMain(ac, av);
  │   ├ ServerLoop()
  │   │   ├ initMasks()
  │   │   ├ select()
  │   │   ├ ConnCreate()
  │   │   ├ BackendStartup()
  │   │   │   ├ PostmasterRandom()
  │   │   │   ├ fork()
  │   │   │   │   ├ BackendFork()                    子プロセスで実行
  │   │   │   │   │   ├ ClosePostmasterPorts(true);
  │   │   │   │   │   ├ ProcessStartupPacket()
  │   │   │   │   │   ├ [クライアントの認証]
  │   │   │   │   │   ├ PostgresMain()
  │   │   │   │   ├ proc_exit(status);
  │   ├ ConnFree()
```

図 2.2 postmaster の主な処理の流れ

ＤＢのスタートアップ処理（StartupDataBase()）やリカバリ処理は子プロセスで行うため、postmaster 側の処理は ServerLoop()に進んでしまい、クライアントの受付を開始してしまう。そこで、ＤＢのスタートアップ処理が完了するまでの間に接続されたクライアントはどうなるのかという疑問が出てくる。実は、ＤＢのスタートアップ処理が完了するまでは、クライアントからの接続は、認証フェーズで全てエラーとなる。つまり、図 2.2 の[クライアントの認証]のところで、子プロセスはエラーになって exit（proc_exit(status)）するのである。
ＤＢのスタートアップ処理が完了するとＤＢのスタートアッププロセスは終了する。その時点で、親プロセスである postmaster にシグナルが上がり、それ以降のクライアントからの要求に対応するように設定される。

3. postmaster の起動シーケンス

ここでは、postmaster の起動シーケンスをもう少し細かく見ていく。main() 関数から呼び出された PostmasterMain() 関数の処理の流れは次のようになっている。ここでは、データやログのディレクトリの決定と検査、SSL の初期化など、一部の処理は省略して書いてある。

1. --help, --version などの処理
2. メモリコンテキストの作成とコンテキストスイッチ
3. GUC と引数の処理
4. ライブラリのプリローディング
5. デーモン化
6. ＤＢのロック
7. ソケットの初期化
8. 共有メモリとセマフォの初期化
9. シグナルハンドラの設定
10. 統計情報収集プロセスの起動
11. クライアント認証情報の読み込み
12. ＤＢのスタートアップ
13. サーバループ

以降で、各処理について説明していく。

3.1. --help, --version などの処理

postmaster が、コマンドラインからの第１引数に、--help や --version のオプションを渡された場合、main() 関数では処理せずに、PostmasterMain() まで渡される。これは、main() 関数が いくつかの目的で呼び出される[2]ので、処理をそれぞれの目的のusage や version を表示できるようにするためである。

[2] 単独でのpostgres プロセスとして呼び出すのに使われたり、DBの初期化時のBootstrap処理、また、7.4 からはパラメータの表示のためなどに使われたりする。

従って、postmaster 本来の処理は必要ないため、PostmasterMain() の冒頭で処理して、プログラムは終了する。
ここでの処理内容は、次のとおりである。

i. --help または -? が第１引数だった場合、usage() を呼び出して、すぐに ExitPostmaster(0) を呼び出してプログラムを終了する
ii. --version または -V が第１引数だった場合、ヘッダファイル pg_config.h で定義された PG_VERSION の値を出力し、すぐに ExitPostmaster(0) を呼び出してプログラムを終了する

3.2. メモリコンテキストの作成とコンテキストスイッチ

まずは、MemoryContextInit() を実行して、メモリマネージャ (utils/mmgr) の初期化を行う。mmgr については、別資料[3]にまとめてあるので、詳しくはそちらを参照して欲しい。
メモリマネージャを初期化すると、最上位のメモリコンテキスト TopMemoryContext が作成されるので、postmaster 用のメモリコンテキストとして、PostmasterContext を TopMemoryContext の下に作成する。PostmasterContext を作成したら、さっそく MemoryContextSwitchTo(PostmasterContext) を実行して、メモリコンテキストを postmaster 用のメモリコンテキストに切り替える。

3.3. GUC[4]と引数の処理

はじめに postmaster に与えられた引数をチェックして、必要があれば GUC の初期値を上書きする。次に postgres.conf ファイルを読み込んで、GUC の値を設定する。

3.4. ライブラリのプリローディング

プリローディングしておくライブラリがある場合、ここで process_preload_libraries() 関数を呼び出して、ライブラリのローディングを行う。特にライブラリが指定されていない場合は、何もしない。

3.5. デーモン化

-S オプション付で起動された場合、postmaster プロセスをデーモンとして実行する。ここで、pmdaemonize() 関数を実行することで、これを実現している。
postmaster をデーモン化する方法は、この関数内でプロセスを fork() して、子プロセスの方を postmaster デーモンとして処理を継続させ、親プロセスは即座に終了して処理を呼び出し元に返すという方法で実現している。
デーモン化された postmaster プロセスは、stdin, stdout, stderr を全て /dev/null にする。

3.6. データベースクラスタ（$PGDATA）のロック

データベースクラスタ（$PGDATA）のディレクトリにロックファイル（postmaster.pid）を作成して、１つのデータベースクラスタに対して、複数の postmaster プロセスが起動しないようにする。
postmaster.pid ロックファイルには、postmaster のプロセス ID やデータベースクラスタのパス、共

[3] PostgreSQL解析資料「メモリ管理」
[4] GUC(Grand Unified Configuration) : コンフィグファイルやコマンドラインで設定できるパラメータを管理しているモジュール。

有メモリのキーと識別子が入っている。
また、スタンドアローンで postgres プロセスを起動したときも postmaster.pid ロックファイルを作成するので、postmaster とスタンドアローンの postgres プロセスは、同時にはどちらか一方しか起動できないようになっている。

3.7. ソケットの初期化

TCP/IP ソケットと UNIX ソケットの初期化を行う。
TCP/IP ソケットの初期化は、TCP/IP の設定が有効になっている場合のみ行われる。デフォルトでは、TCP/IP は無効になっているので、postgresql.conf ファイルで設定を変更して、TCP/IP を有効にする必要がある。TCP/IP が有効な場合でかつ VirtualHost が設定してある場合、その数だけ listen するソケットを生成する。多くの場合は、TCP/IP を有効にしても VirtualHost は設定されないことが多いので、通常１つの TCP/IP のポートを listen している。
UNIX ソケットは、基本的に同一ホスト内の通信に使われる。UNIX ソケットは、/tmp に名前付きソケットが作られる。多くの場合、同一ホスト内の通信には UNIX ソケットが使われるのだが、JDBC では同一ホスト内であっても TCP/IP を使用する。

3.8. 共有メモリとセマフォの初期化

各モジュールの共有メモリの使用量を見積もる関数にアクセスして、使用する共有メモリの量を計算し、共有メモリをまとめて確保する。必要な量の共有メモリを確保したら、各モジュールの共有メモリ初期化関数を呼び出して初期化を行わせる。

3.9. シグナルハンドラの設定

postmaster の使用するシグナルハンドラを設定する。まず、この時点でシグナルをブロックしてしまう。そして、各シグナルにシグナルハンドラを設定していく。詳細は、５章のシグナル処理で説明する。

3.10. 統計情報収集プロセスの初期化と起動

統計情報収集プロセス（statistic collector）は、postmaster/pg_stats.c で実装されている。この段階で、統計情報収集プロセスを初期化して起動する。
統計情報収集のためのプロセスは、stats buffer process と stats collector process の２種類が起動される。

3.11. クライアント認証情報の読み込み

ここで、クライアントを認証するための情報を、ファイルから読み込んでキャッシュしておく。対象となるのは、次の４つの情報である。

- $PGDATA/pg_hba.conf ファイル
- $PGDATA/pg_ident.conf ファイル
- $PGDATA/global/pg_pwd ファイル
- $PGDATA/global/pg_group ファイル

3.12. DB のスタートアップ

ここで、ＤＢのスタートアップ処理を行う。ＤＢのスタートアップ処理とは、ディスク上のデータベースをオープンして、必要ならリカバリ処理を行い、postgres プロセスから利用できるようにすることである。

先に述べたように、postmaster はこれらの処理を直接行うことはせず、子プロセスを起動して行う。詳しくは、6.2 節の「ＤＢのスタートアッププロセス」で説明する。

3.13. サーバループ

ServerLoop()は、postmaster のメインループである。ServerLoop()では、最初に select() システムコール用のマスクを設定し、その後は、ループしながらクライアントや他のバックエンドから来る要求を処理する。

ループ内での処理の詳細は、次章以降で説明していく。

正常終了の場合、postmaster はこの関数から return してくることはない。proc_exit() が呼ばれて、そこから exit()する。この関数から return されるのは、select()システムコールのエラーの場合である。

4. ソケットとサーバループ

サーバループ中での、ソケットでクライアントの要求を待つ部分は、図 4.1 のようになっている。

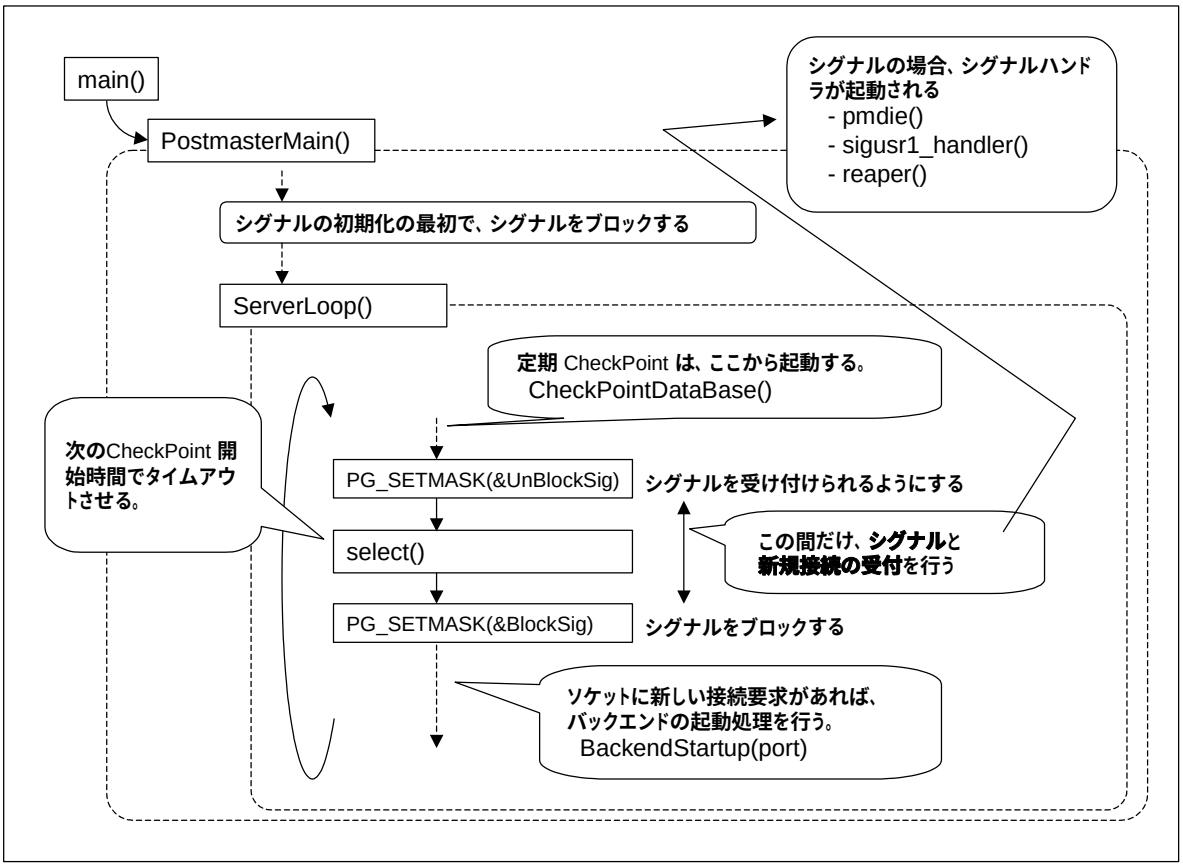

図 4.1 ソケットでの受付とシグナル処理

図中には、ソケットでの要求待ちだけでなく、シグナルの処理も書いてある。シグナルとソケットの処理は、全て、PG_SETMASK(&UnBlockSig) でシグナルのブロックを解除してから、再び、PG_SETMASK(&BlockSig) でシグナルをブロックするまでの間に受付けられる。この間では、select() システムコールとシグナルによる割込み処理が実行される。select() システムコールは、定期チェックポイントの残り時間を計算した時間でタイムアウトするように設定されている。
定期チェックポイントは、select()システムコールから抜けてループで先頭に戻った場合、タイムアウト時間が過ぎていれば実行される。チェックポイント起動のタイミングの詳細については 6.4.2 で説明する。
また、シグナル処理については、５章で述べるので、ここではソケット処理について話を進める。

4.1. TCP/IP ソケット

PostgreSQL も、他の多くのソフトウェア同様に、TCP/IP のソケットを使って、クライアントサーバ間の通信を行う。
デフォルトでは、次のように定義してあり、最大 10 個のポートを作成できるようになっている。

```
#define MAXLISTEN 10
```

これは、virtual host などを設定した場合に使用され、標準では１つのポートだけを使うことになる。

この MAXLISTEN は、同時に listen するために起動するポートの数の上限であり、listen() システムコールの引数に与える listen のためのキューの数ではない。listen() では、バックエンドの最大数の 2 倍か PG_SOMAXCONN（ = 10000 ）の小さいほうの値が与えられる。

ソケットの初期化における、ソケットの生成から listen 開始までの処理は、StreamServerPort() 関数の中で行われている。ソケット作成時には、ソケットのオプション設定 setsockopt() として、SO_REUSEADDR が与えられて、postmaster がダウンしても長時間そのポートを占有しないようになっている。

ちなみに、TCP_NODELAY や SO_KEEPALIVE の接続オプションは、ConnCreate() 関数から呼ばれている StreamConnection() 関数の中で accept() を行った直後に行っている。

4.2. UNIX ソケット

PostgreSQL は、同一ホスト中でのクライアントサーバ間の通信には、基本的に UNIX ソケットを利用する。例外は、JDBC のインタフェースで、これは TCP/IP のソケットを使用する。

ソケットの初期化などの処理は、TCP/IP の場合と同じ StreamServerPort() 関数を、引数だけ替えて呼び出している。

4.3. クライアントの新規接続要求の受付

クライアントから新規接続要求の流れは、図 4.2 のようになる。

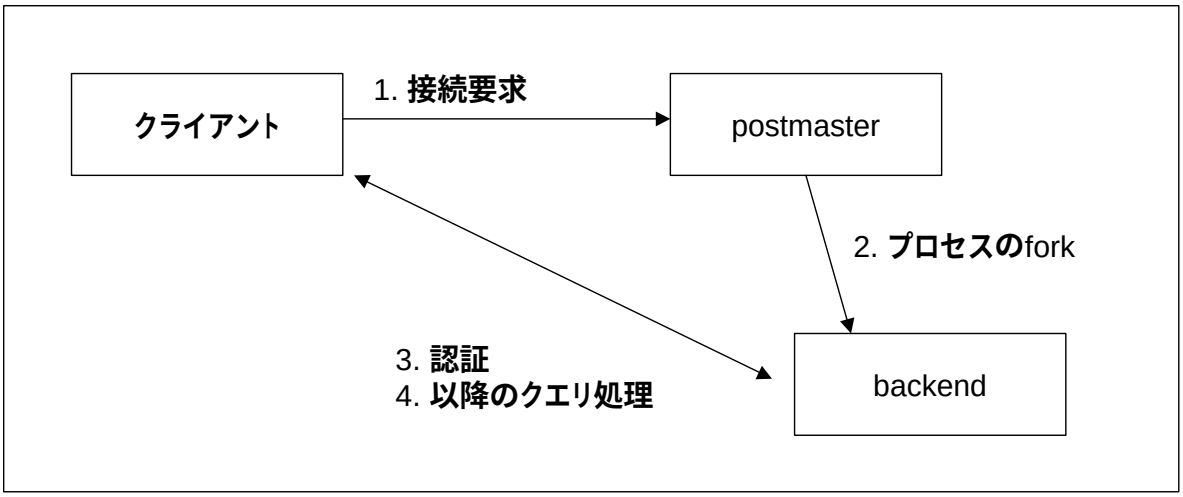

図 4.2 クライアントの新規接続処理

まず、クライアントは postmaster に接続要求を行う。
postmaster は、select() システムコールで接続要求を受付けると、どのソケットに対して接続があったかをチェックして、該当ソケットに対して ConnCreate(ListernSocket[i]) を実行してコネクションを準備する。ConnCreate() の中で、StreamConnection() 関数が呼ばれ、accept() システムコールが実行される。ConnCreate() で接続を受け付けたら、続いてバックエンドの起動処理 BackendStartup() を呼び出す。BackendStartup() では、まず、キャンセルキーを PostmasterRandom() で生成して、その後 fork() を実行する。fork() の前にキャンセルキーを生成しているのは、postmaster 側でもキャンセルキーを保持しておく必要があるからである。キャンセルキーは、後述するクライアントからのクエリの中止処理で、クライアントとそれに対応するバックエンドの組み合わせが正しいかどうかを判別するために使用される。

fork() 後の親プロセス（postmaster）側では ConnFree()を呼び出し、親プロセス側では接続情報を解放し、クライアントとの接続を閉じて、そのクライアントの処理を終了する。

fork() 後の子プロセス側では、BackendFork() を呼び出して処理を継続する。BackendFork() では、クライアント認証や postmaster として使用していたリソースの開放を行った後、PostgresMain() 関数を呼ぶことで、バックエンドプロセス postgres としての動作を開始する。

ServerLoop() 以降の主な関数シーケンスと、fork 後の PostgresMain()までの主な関数シーケンスは、図 4.3 のようになっている。

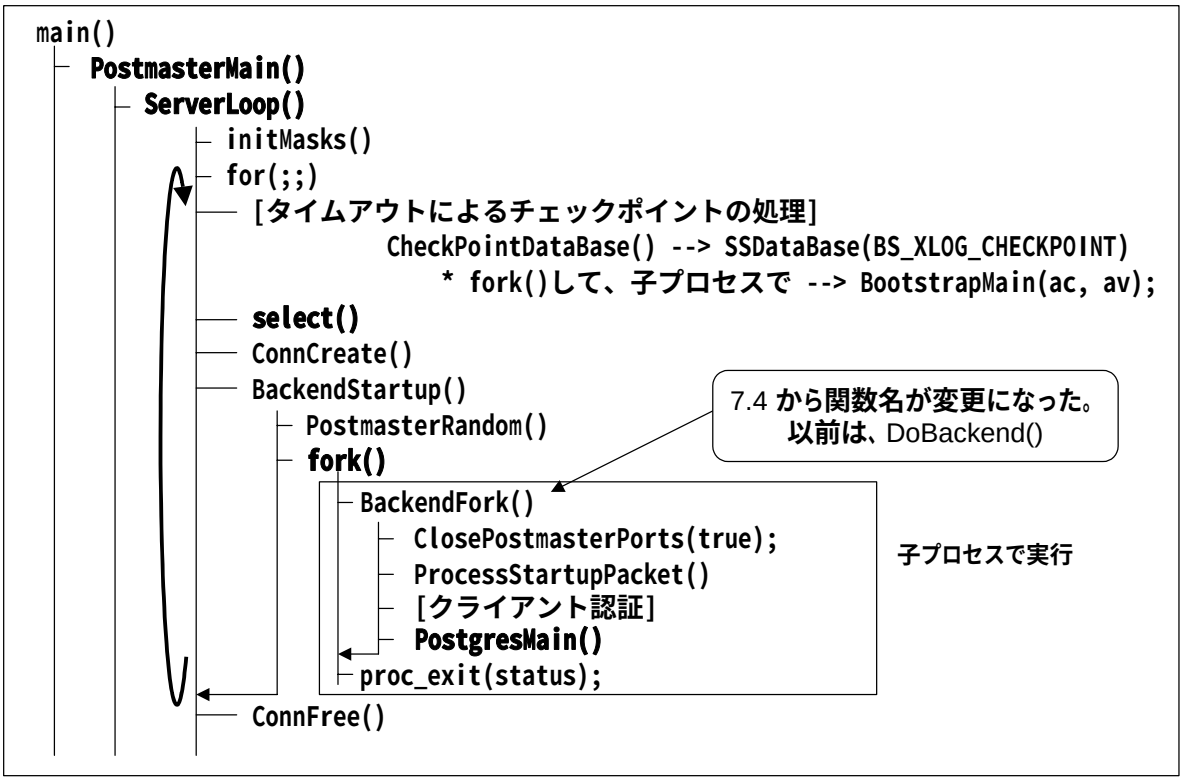

図 4.3 ServerLoop()と PostgresMain()までの流れ

4.4. fork 後の動作とクライアント認証

ここでは、fork() で子プロセスが生成されて BackendFork() 関数が呼ばれてから、PostgresMain() を実行するまでの処理について見ていく。

PostgreSQL では、子プロセスを fork()するが、exec()システムコールによるプロセスの書き換えは行わない。postmaster として使用していたリソースを開放し、必要な部分だけ再初期化して、バックエンドプロセス postgres として機能する。

4.4.1. postmaster としてのリソース開放と再初期化

まずは、postmaster として使っていた、クライアント受付用のソケットを閉じる。続いて、自分のプロセス ID とポート番号を取得して、グローバル変数に設定する。

4.4.2. クライアントとの最初の通信とクライアント認証

クライアント認証は、認証を行っている途中でクライアントが応答しなくなって、サーバ側でのリソースが解放されなくなるのを防ぐため、一定時間以内で認証に成功しなかった場合、子プロセスを exit するように作られている。

まず、シグナル SIGTERM と SIGQUIT のシグナルハンドラを、authdie() という単純に exit するだけの関数に変更する。

次に、制限時間が過ぎたときに子プロセスを exit するように、SIGALRM も authdie() に設定する。続いて、ブロックするシグナルを変更する。postmaster と違うのは、SIGALRM, SIGTERM, SIGQUIT, SIGHUP をブロックしないという点である。

ここで、enable_sig_alarm() で時間制限用のアラームを設定する。これで認証が完了するより先にアラームが上がった場合、authdie() が呼ばれて exit するようになる。

ここまでの準備が終わったら、クライアントとの通信を開始する。まず、スタートアップパケットの受信のため、ProcessStartupPacket() を呼び出す。ProcessStartupPacket() では、キャンセルパケットでないかどうかのチェック、SSL の通信の処理、プロトコル番号のチェックなどを行い、その後、クライアントが接続しようとしているデータベース名、接続ユーザ名、パスワードなどを取り出す。もしここで受け取ったパケットがキャンセルパケットであった場合、キャンセルパケット用の処理を呼び出し、それが終了し次第 return してその先で exit される。キャンセルパケットについては、4.5 節で説明する。
通常の接続要求であった場合、続いて、ClientAuthentication() を実行して認証を行う。
ClientAuthentication() では、認証に失敗すると戻ってこないで関数内で exit する。

ClientAuthentication() から戻ってきたということは認証が成功しているので、アラームの設定をオフにする。

ちなみに、クライアントの認証関数 ClientAuthentication() の中では、ＤＢのスタートアップ、リカバリ、ＤＢのシャットダウンなどの状態の場合の接続拒否や、同時接続数の第１次チェックによる接続拒否が行われる。同時接続数については、この時点では最大同時接続数（MaxBackends）の２倍を許可する。理由は、このプロセスの認証や初期化のフェーズが終わるまでに処理の終わるバックエンドがあるかもしれないし、また、他のいくつかのプロセスはまだ認証フェーズにあるかもしれず、その中のいくつかが認証に失敗してＤＢに接続する前に終了するかもしれないからである。そのため、厳密な最大同時接続数より多くのプロセスが存在することを許している。厳密な最大同時接続数による制限は、バックエンドが共有の inval バックエンド配列 (shared-inval backend array) に接続するときに調整される。

4.4.3. コンテキストスイッチ

クライアントの認証が終わった時点で、メモリコンテキストを TopMemoryContex に変更し、PostmasterContext を削除して、postmaster として確保したメモリを一気に開放する。

ここまでの処理が終わったところで、PostgresMain() が呼び出され、バックエンドプロセスとしての postgres プロセスの処理に移っていく。PostgresMain()以降の処理の概要については、図 4.4 のようになっている。

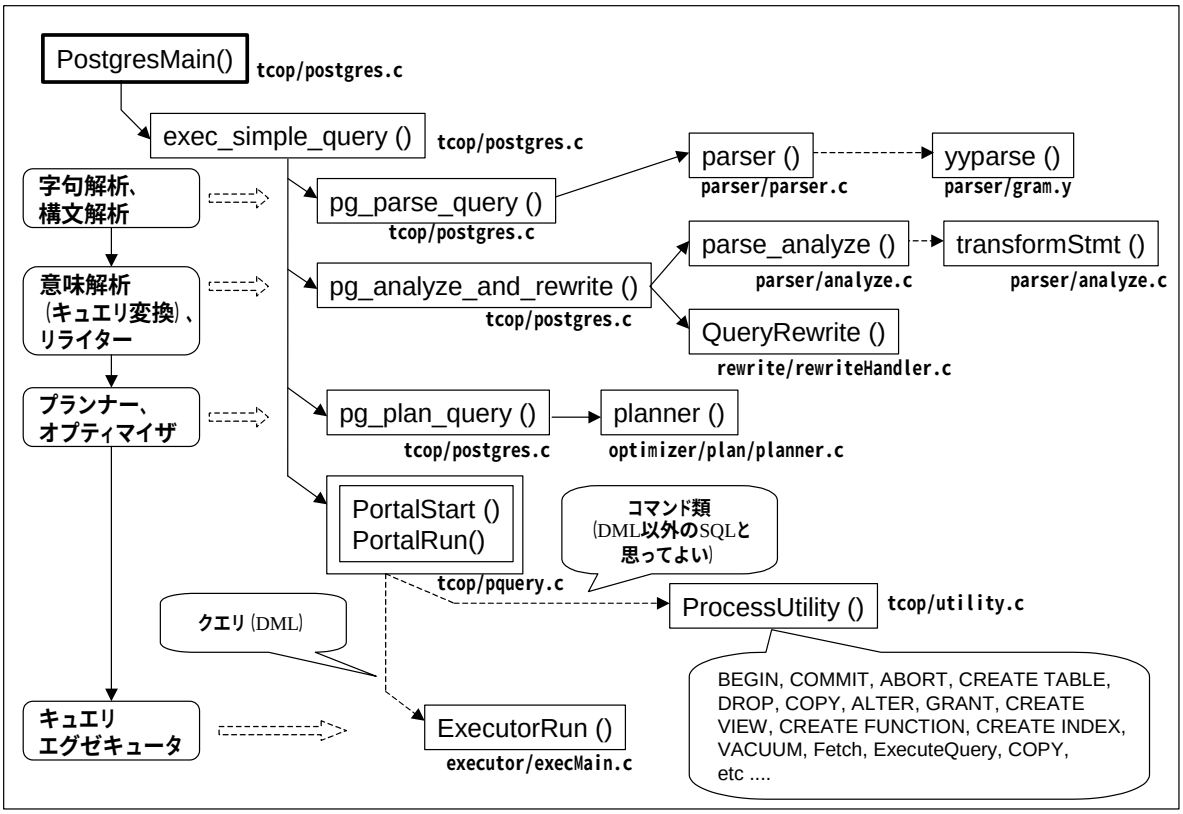

図 4.4 PostgresMain()以降の処理

4.5. クライアントからのクエリのキャンセル処理の仲介

クライアントがクエリ処理を中止する場合、直接バックエンドプロセスにクエリ処理の中止を通知するのではなく、キャンセルパケットと呼ばれる特殊なパケットを postmaster に向けて送る。例えば、psql で SQL 文の実行中に Ctrl + C を入力するとキャンセルパケットが送信される。
そして、それを受け取った postmaster が、バックエンドプロセスにクエリの中止を伝える。

クライアントからのキャンセル処理の流れは、図 4.5 のようになる。

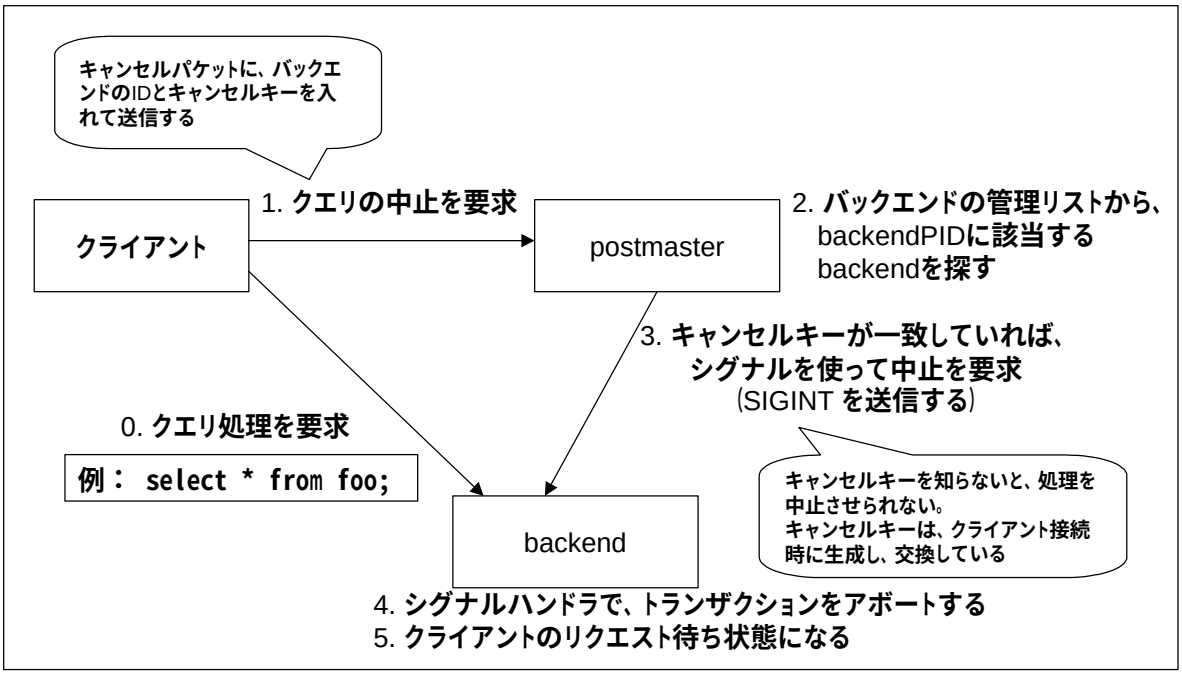

図 4.5 クエリのキャンセル処理

クライアントが実行中のクエリを中止したい場合、まず、クライアントでキャンセルパケットを生成して、それを postmaster に投げる。
postmaster は、クライアントから何らかの接続を受け取った時点で子プロセスを fork するので、キャンセルパケットを受け取った場合も子プロセスが起動される。起動された子プロセス側で、postmaster として管理しているバックエンドプロセスのリストを探索し、該当するバックエンドプロセスに対して SIGINT シグナルを送る。これら処理は、バックエンド開始時点の ProcessStartupPacket() の中で判別され、processCancelRequest() の中で処理される。バックエンドプロセスに SIGINT シグナルを送る際には、バックエンドの PID とキャンセルキーを用いて、本当にそのバックエンドプロセスに対応しているのクライアントからのキャンセル要求であるかを判別している。

キャンセルパケットを受け取った場合、起動された子プロセスは、通常の PostgresMain() などのバックエンドの処理などは一切行わず、キャンセル処理だけを実行して終了する。

起動された子プロセスから SIGINT を受け取ったバックエンドプロセスは、シグナルハンドラが呼び出されてトランザクションをアボートし、クライアントからのリクエスト待ち状態になる。ここで、バックエンドプロセスもシグナルをブロックしている場合があるので、なかなかバックエンドの処理がアボートされないこともある。

キャンセルパケットの構造は、次のようになっており、cancelRequestCode が定数 CANCEL_REQUEST_CODE に設定されている。

```
typedef struct CancelRequestPacket
{
    MsgType     cancelRequestCode;
    uint32      backendPID;
    uint32      cancelAuthCode;
```

```
} CancelRequestPacket;
```

cancelAuthCode には、キャンセルキーが入ってる。キャンセルキーとは、クライアントの新規接続時に、バックエンドを生成する際に乱数で発行したキーで、postmaster と接続要求をしてきたクライアントと、それに対応するバックエンドしか知らないキーである。

5. シグナル処理

シグナル処理は、postmaster の動作の理解を難しくしている要因の１つだと言える。シグナルによって、突然ある関数(シグナルハンドラ)に処理が移るため、ソースコードの遷移が分かりにくくなっている。例えば、postmaster の終了処理はシグナルハンドラから行われるので、シグナルを意識しないでソースコードを追いかけていると、どこで終了するのか検討がつかない。
シグナル処理の基本的な流れは、前章の図 4.1 を参照して欲しい。PostmasterMain() 関数の中で、シグナル関連の初期化処理に入ったら、すぐにシグナルは全てブロックされる。その後、ServerLoop() 関数内の select() システムコールでクライアントの新規接続を受付けている間だけ、シグナル処理も受付ける。図 4.1 のように、select() システムコールを呼び出す直前に、シグナルのブロックを開放してシグナルを受け取れるようにする。そして、select() システムコールが終わったら、すぐにシグナルをブロックする。
シグナルハンドラが起動された場合、各シグナルハンドラでは、すぐにシグナルをブロックする。そして、自分の処理が終わったところで、再びシグナルのブロックを開放してシグナルを受け取れるようにしている。

postmaster で定義しているシグナルハンドラは、次のようになっている。

SIGHUP	SIGHUP_handler	設定ファイルの読み直しを行う。
SIGINT	pmdie	実行中の全てのトランザクションをアボートして終了する。
SIGQUIT	pmdie	即時終了を行う。他のバックエンドには、一方的に非同期の即時終了シグナルを投げる。
SIGTERM	pmdie	実行中の全てのクライアント処理を待って postmaster を終了する。
SIGUSR1	sigusr1_handler	バックエンドからの通知を受け取るために使用する。
SIGUSR2	dummy_handler	バックエンド起動時のシグナルの取りこぼし予防に使う。
SIGCHLD	reaper	各種の子プロセスが終了したときの処理を行う。

この表以外のシグナルには、SIG_IGN を設定して、シグナルを無視する。

それでは、これらのシグナルハンドラの動作を１つずつ取り上げて説明していく。

5.1. SIGHUP_handler

コンフィグファイル postgresql.conf を読み直し、全てのバックエンドプロセスに SIGHUP を送り、設定ファイルの再読込みを伝える。
その後、pg_hba.conf ファイルと pg_ident.conf ファイルを読み直す。

5.2. pmdie

postmaster が SIGINT, SIGTERM, SIGQUIT のいずれかのシグナルを受け取った場合に起動されるシグナルハンドラである。基本的に postmaster の終了処理である。シグナルの種類によって動作が異なる。

5.2.1. SIGTERM の場合

スマートシャットダウンの処理であり、全てのバックエンドプロセスの処理が終了を待って、postmaster を終了する。
まず、シャットダウンの状況を示すグローバル変数 Shutdown を調べ、まだいずれかのシャットダウンの値が設定されていなければ、SmartShutdown という値を設定する。
バックエンドプロセスが１つもない場合、ＤＢのシャットダウン処理 ShutdownDataBase() をここから起動する。バックエンドが１つでも実行中の場合は、シグナルハンドラ pmdie の処理はここで終了する。グローバル変数 Shutdown に SmartShutdown を設定したことにより、バックエンドが終了した際に送られてくる SIGCHLD シグナルによってシグナルハンドラ reaper が起動された時に、バックエンドの数をチェックして全てのバックエンドがいなくなったタイミングでＤＢのシャットダウン処理 ShutdownDataBase() が実行されるようになる。また、グローバル変数 Shutdown が SmartShutdown になったことで、新規のクライアントの接続は、認証フェーズで拒否するようになる。

5.2.2. SIGINT の場合

ファストシャットダウンの処理であり、実行中のバックエンドを全てロールバックさせて、その後 postmaster を終了する。
基本的な処理の流れは、SIGTERM の処理と同じである。グローバル変数 Shutdown に設定する値が、FastShutdown になる。また、グローバル変数 Shutdown が SmartShutdown の場合は上書きする。
実行中のバックエンドが１つでもある場合、全てのバックエンドプロセスに SIGTERM を送る。シグナルハンドラ pmdie 自体はここで抜けて、後の処理は SIGTERM 同様にシグナルハンドラ reaper に任せる。その他は、SIGTERM と同じである。

5.2.3. SIGQUIT の場合

即時シャットダウンの処理であり、ＤＢのシャットダウン処理を実行せずに postmaster を終了する。ＤＢのスタートアッププロセス、または、ＤＢのシャットダウンプロセスが実行中であった場合、そのプロセスに SIGQUIT を送りつけて強制的に終了させる。実行中のバックエンドが１つでもあった場合、全てのバックエンドに SIGQUIT を送って強制的に終了させる。
最後に ExitPostmaster(0) を呼ぶことで、共有メモリのリソース開放などを行って postmaster は終了する。exit の際に、他のプロセスの終了を待つようなことはしない。

SIGKILL を postmaster に送った場合との違いは、PostgreSQL に関係する全てのプロセスに終了のための SIGQUIT を送ることと、共有メモリの開放などが行われるところである。

5.3. sigusr1_handler

PostgreSQL では、子プロセスが postmaster に何らかの要求を行う際にシグナルで通知する。しかし、UNIX のシグナルは既に 32bit 分ぎりぎりまで定義されていて、ユーザ定義用シグナルは２つしかない。そこで、子プロセスから postmaster へ要求を送る場合は、要求の内容を共有メモリの規定の場所に設定して、それから SIGUSR1 を送る。すると、この sigusr1_handler が起動して、このシグナルハンドラの中で要求を判断する。

sigusr1_handler では、まず、CheckPostmasterSignal() を使って、子プロセスの要求内容を判断する。子プロセスの要求内容の処理として次のようなことを行う

5.3.1. チェックポイント実行要求（ PMSIGNAL_DO_CHECKPOINT ）だった場合

これは、WAL セグメントを一定量使い切った場合のチェックポイント要求である。これからチェックポイントを実行するので、タイムアウトによるチェックポイントを起動時間を遅らせるために、タイムアウト用のチェックポイント実行時刻を現在時刻に設定する。つまり、経過時間を 0 にすることと同等の処理をしている。
続いて、既にチェックポイント実行中でなく、かつ、チェックポイントが disable でないなら、チェックポイント用のプロセスを起動して、シグナルハンドラを終了する。

5.3.2. パスワードファイルの変更（ PMSIGNAL_PASSWORD_CHANGE ）だった場合

パスワードファイルが変更されたことの通知なので、$PGDATA/global 以下にある pg_user ファイルと pg_group ファイルの読み直しを行う。
これだけ行ったら、シグナルハンドラは終了する。

5.3.3. 他の子プロセスへの SIGUSR2 の送信（ PMSIGNAL_WAKEN_CHILDREN ）だった場合

あるバックエンドが他の(全ての？)バックエンドになんからの要求を通知する手段である。今回は、これがどのような場合に使われるのかは調査していない。
postmaster での処理自体は、シャットダウン要求がかかっていない通常の状態であれば、全ての子プロセスに SIGUSR2 を送信するというものである。
これを受けたら、各バックエンドプロセスは、シグナルハンドラ Async_NotifyHandler が起動され、その中で要求内容を判定して何らかの処理を行う。postmaster への通知同様、共有メモリになんからの共有内容を事前に設定してあると考えられる。

5.4. dummy_handler

実際には、postmaster で使われないので、シグナルハンドラの関数自体は空になっている。
これは、postmaster でシグナルハンドラを SIG_IGN にしておくと、新しいバックエンドを起動したときに、バックエンド側で新しいシグナルハンドラが有効になるまでの間、シグナルを取りこぼしてしまう。それを予防するためのものである。つまり、SIG_IGN にしておくとシグナルをブロックできないことへの対処である。

5.5. reaper

reaper() は、子プロセスが終了したときに起動されるシグナルハンドラである。終了した子プロセスの種類によって異なる動作をする。
UNIX のシグナルは、複数のプロセスが同時に終了した場合も、１つしかシグナルが上がらないので、

wait キューに入っている全ての終了済み子プロセスの処理を行ってから、このシグナルハンドラを抜けなければならない。
waitキューからの子プロセスの取り出しは、waitpid(-1, &status, WNOHANG) のようなシステムコールを使う[5]。WNOHANGは、待つべき子プロセスが存在しない場合には すぐに返ってくることを意味する。ちなみに、WNOHANGを設定しない場合、waitpid() でずっと待ってしまうことになる。

5.5.1. 統計情報収集プロセス が終了した場合

終了した子プロセスが統計情報収集プロセスであった場合、単純に、統計情報収集プロセスの再度起動を行う。

5.5.2. ＤＢのシャットダウンプロセスが終了した場合

終了した子プロセスがＤＢのシャットダウンプロセスであった場合は、ExitPostmaster()を実行して postmaster を終了する。引数には、ＤＢのシャットダウンプロセスが正常終了している場合は 0 、異常終了だった場合は 1 を渡し、ExitPostmaster() の中で exit される。通常、postmaster は、ここから終了する。

5.5.3. ＤＢのスタートアッププロセスが終了した場合

ＤＢのスタートアッププロセスが異常終了していた場合は、ExitPostmaster(1) で終了する。
ＤＢのスタートアッププロセスが正常に終了していれば、グローバル変数 StartupPID を 0 に戻す。これによって、クライアントの接続が可能になる。
起動回数の連番（ThisStartUpId）とリカバリ時のリカバリ処理の開始位置のログレコードへのポインタ（RedoRecPtr）を記録しておく。また、リカバリ処理が終わっているはずなので、内部でエラーの処理中であることを示すグローバル変数 FatalError を false にする。
続いて、チェックポイントのタイムアウトのための情報をリセットする。シャットダウン時、または、リカバリ処理の最後でチェックポイントが実行済みのはずなので、この時点でチェックポイントが終了したことにして、チェックポイントのタイムアウトの値を設定する。
この時点で、ＤＢのシャットダウン処理が保留されていたのを検出したら、ＤＢのシャットダウン処理を行う。

5.5.4. その他の場合

その他の場合というのは、通常のバックエンドプロセス postgres か、チェックポイントプロセスが終了した場合である。
正常終了の場合、まず、バックエンドプロセスのリストから、今終了したプロセス番号の情報を外す。その後、終了したのがチェックポイントの場合、グローバル変数 CheckPointPID を 0 にして、チェックポイントの終了時間を設定する。終了したのが通常のバックエンドの場合、統計情報収集プロセスにバックエンドの終了を伝える。正常終了の場合は、ここで return する。

異常終了の場合、まだ FatalError が true でなければ、全てのバックエンドプロセスに即時終了するように伝える。このとき、SIGQUIT シグナルを送ることで、バックエンドでは proc_exit を呼ばずに終了させ、ユーザには何が起こったか知らせないようにしている。しかし、SendStop が設定されている場合(つまり -s がコマンドラインで指定されていた場合)、SIGQUIT の代わりに SIGSTOP を送ることで、各バックエンドをコアダンプさせる。

[5] waitpid がないOSでは、wait3() を使うように #ifdef が切ってある。

異常終了したのが、チェックポイントの場合、チェックポイントの情報を初期化する。
最後に FatalError を true にする。これによって、postmaster でのクライアントの受付が中止される。また、シグナルハンドラ reaper の後処理のフェーズで、全てのバックエンドが終了した場合に、共有メモリの再初期化や、データベースのリカバリが実行される。

5.5.5. 後処理

この前までが、終了した子プロセスの情報が wait キューに入っていた場合の、それぞれの子プロセスに対する終了処理である。その後、シグナルハンドラ reaper の後処理として次の２つを行う。
１つ目は、FatalError が true の場合、全ての子プロセスが終わるのを待って、共有メモリを初期化し、StartupDataBase() を実行し、ＤＢのスタートアップ処理を行う。ＤＢのスタートアップ処理の中でＤＢのリカバリも行われる。
２つ目は、シャットダウンの状態を示すグローバル変数 Shutdown が NoShutdown でなかった場合、つまりバックエンドを待つタイプのシャットダウン(スマートシャットダウン、または、ファストシャットダウン)の最中である場合は、バックエンドがもう残っていないかチェックする。バックエンドがもう残っていなくて、かつ、まだＤＢのシャットダウン処理プロセスが起動されていない場合、ＤＢのシャットダウン処理 ShutdownDataBase() を実行する。

5.6. シグナルハンドラの動作例

ここでは、postmaster のファストシャットダウン (コマンドでは"pg_ctl –m fast stop") を例に挙げて、シグナルハンドラとシャットダウン 処理の流れを説明する。図 5.1 が、ファストシャットダウンのときの処理の流れである。

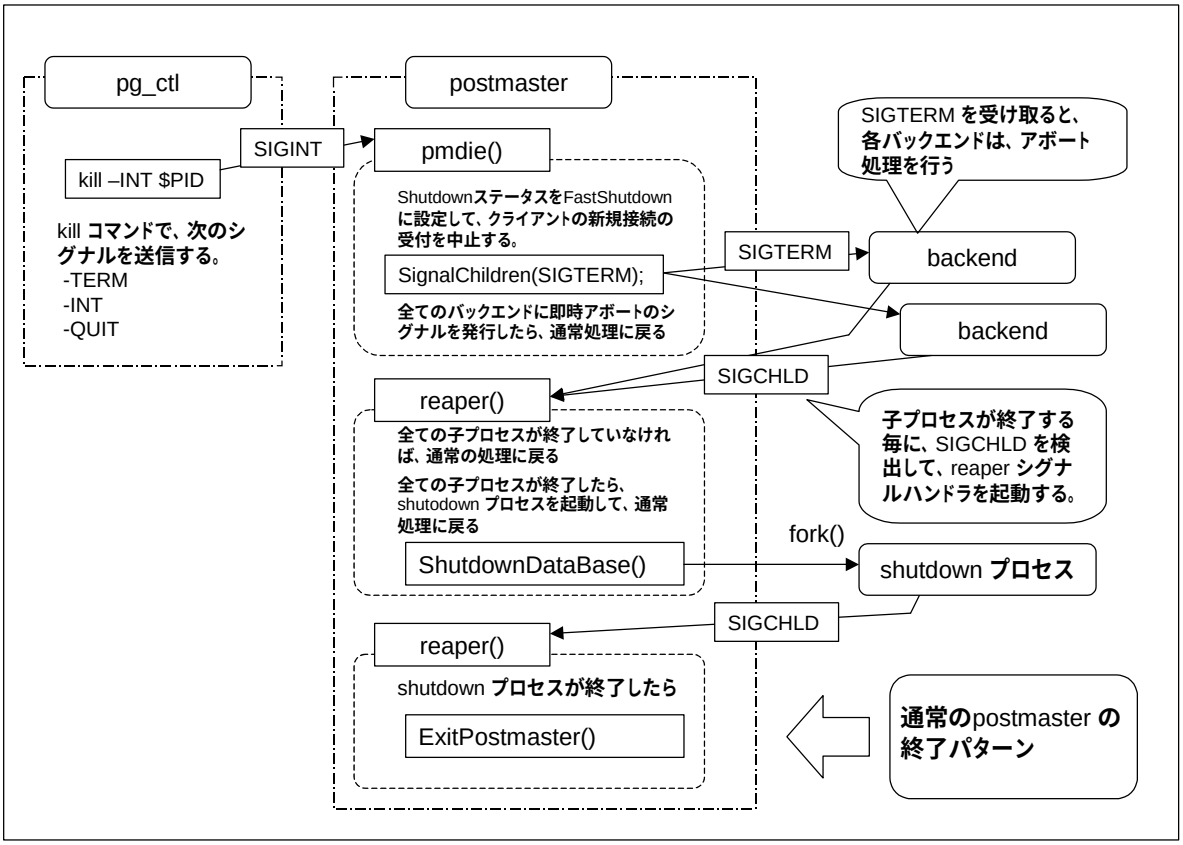

図 5.1 ファストシャットダウンの処理フロー

まず、図の左側 pg_ctl から postmaster へ SIGINT が送られる。SIGINT を受け取った postmaster では、シグナルハンドラ pmdie が起動される。postmaster でシグナルハンドラが起動されるタイミングは、ServerLoop() 中の select() システムコール直前で PG_SETMASK(&UnBlockSig) でシグナルのブロックを解除してから、再び、PG_SETMASK(&BlockSig) でシグナルをブロックするまでの間である。
シグナルハンドラ pmdie では、グローバル変数 Shutdown の値を FastShutdown に変更して、postmaster がこれ以降クライアントの接続を受付けないようにする。厳密には、postmaster は接続を受付けるが、fork 後の認証フェーズでエラーになるのである。
ここで、バックエンドプロセスが１つもなければ、すぐにＤＢのシャットダウンプロセスを起動するのだが、この例では、バックエンドがいくつかあったと仮定する。
さて、バックエンドがいくつか残っているので、各バックエンドプロセスに SIGTERM シグナルを送って、アボート処理を促す。この時点で、シグナルハンドラ pmdie の処理は終了して、postmaster は通常処理に戻る。
SIGTERM を受け取ったバックエンドプロセスは、シグナルがアンブロックされて、シグナルハンドラを起動できた時点で、現在実行中のトランザクションをロールバックしてプロセスを終了する。
バックエンドプロセスが終了するたびに、postmaster に SIGCHLDシグナルが発生し、シグナルハンドラreaperが起動される。シグナルハンドラreaperの中では、終了したバックエンドをバックエンドの管理リストから外す。ここで、１つしかシグナルが上がっていてなくても、複数のバックエンドが終了している可能性がある[6]ので、シグナルハンドラreaperでは waitキューがなくなるまでこの処理を繰り返す。
wait キューが空になった時点で、ループから抜けて残りのバックエンドの数をチェックする。まだ、バックエンドが残っている場合、そのままシグナルハンドラ reaper を抜け、通常の postmaster の処理に戻る。もし、バックエンドの数をチェックしたときに、もうバックエンドが１つもいなくなっていた場合、ＤＢのシャットダウンプロセスが実行中かどうかチェックして、まだ、ＤＢのシャットダウンプロセスが起動されていなければ、ＤＢのシャットダウンプロセスを起動する。ＤＢのシャットダウンプロセスを起動したら、シグナルハンドラ reaper を抜けて、通常の postmaster の処理に戻る。
postmaster は、この時点では、すでにクライアントの受付も中止しており、ＤＢのシャットダウンプロセスの終了を待つだけである。
シャットダウンプロセスが終了したら、バックエンドプロセスが終了したとき同様に SIGCHLD が送られてくるので、シグナルハンドラ reaper が起動される。シグナルハンドラ reaper の中で、シャットダウンプロセスが終了したことを判別し、最後に ExitPostmaster() を呼び出して、この中で exit して postmaster が終了する。
以上が、ファストシャットダウンの流れである。

先に説明したように、スマートシャットダウンの場合は、バックエンドに SIGTERM を送る部分がない。従って、各バックエンドプロセスの終了は、クライアントが終了するまで待つことになる。その他の流れは、この例と同じである。

[6] これはUNIXのシグナルの仕様であり、必ずwaitキューが空になるまでチェックする必要がある。

6. 子プロセス

ここでは、postmaster での子プロセスの管理方法と、バックエンドプロセス（postgres）以外の子プロセスの起動され方を見ていくことにする。バックエンドプロセス（postgres）の起動され方は、すでに４章「ソケットとサーバループ」で説明している。

6.1. 子プロセス管理

postmaster では、起動した子プロセスのプロセス ID を管理している。今回調査した範囲での postmaster での子プロセスの管理方法は、大きく２つの種類に分けられる。
１つは、ＤＢのスタートアッププロセス、ＤＢのシャットダウンプロセス、チェックポイントプロセスのように、起動したプロセスのプロセス ID をグローバル変数に直接保存する方法をとっているものである。ＤＢのスタートアッププロセス、ＤＢのシャットダウンプロセス、チェックポイントプロセスは、それぞれ、StartupPID, ShutdownPID, CheckpointPID というグローバル変数に子プロセスの ID を入れて管理している。
もう１つは、クライアントの要求を処理するバックエンドプロセスであり、これらは、グローバル変数 BackendList につながれたリスト構造で管理されている。例外なのは、チェックポイントプロセスで、これだけはこの BackendList のリストにもつながれている。
また、統計情報収集プロセスの管理方法は、今回は調査していないので、また別の方法で管理しているかもしれない。

6.2. ＤＢのスタートアッププロセス

ＤＢのスタートアッププロセスの起動は、ソースコード上はマクロ StartupDataBase() を呼び出すところから始まる。マクロ StartupDataBase() は次のように定義されている。

```
#define StartupDataBase()       SSDataBase(BS_XLOG_STARTUP)
```

SSDatabase() は、ＤＢのスタートアッププロセスを起動する時以外にも、ＤＢのシャットダウンプロセスやチェックポイントプロセスを起動する時もそれぞれ違う引数で呼ばれている。SSDatabase() の役割は、子プロセスを起動して BootstrapMain() 関数を呼び出させることである。

6.2.1. SSDatabase()

ＤＢのスタートアッププロセス、ＤＢのシャットダウンプロセス、チェックポイントプロセスは、引数が異なるだけで、いずれも SSDatabase() を呼び出すので、ここではまとめて SSDatabase() での処理内容を説明しておく。

SSDatabase() の中では、まず標準出力、標準エラー出力をフラッシュして、その後 fork() を実行する。
fork() された子プロセス側では、まず、postmaster から起動されたプロセスであることを示すために、グローバル変数 IsUnderPostmaster を true にする。postmaster から起動されているが、既に別プロセスであり、postmaster としての exit 処理は必要ないため、exit ルーチン（proc_exit()で実行するも

の）を初期化する。また、postmaster として使用していた listen 用のソケットも全てクローズする（ClosePostmasterPorts(true)）。そして、ps コマンドで表示する際のプロセス名の変更も行っておく。

プロセス	表示名
ＤＢのスタートアッププロセス	startup subprocess
ＤＢのシャットダウンプロセス	shutdown subprocess
チェックポイントプロセス	checkpoint subprocess

続いて、BootstrapMain() の引数を準備する。BootstrapMain() は、C 言語の main() 関数のように、引数の個数と引数の配列を渡す。そして、BootstrapMain() の中で、コマンドラインから引数をもらったときと同様の処理を行って、引数の値を取り出す。BootstrapMain() に渡す引数は、次のようになっている。

av[0]	postgres	Ｃ言語同様、プロセス名を入れる
av[1]	-Bnnn	nnn は、バッファ数 NBuffers を入れる
av[2]	-xNNN	NNN は、BootstrapMain での処理の種類が入る。 STARTUP = 2, CHECKPOINT = 3, SHUTDOWN = 4
av[3]	-p	av[4] で渡す引数の意味を与えている
av[4]	SegID,SegAddr,"template1"	使用してる共有メモリのセグメントＩＤ、セグメントアドレス、"template1"という文字列が、カンマ区切りで文字列として渡される
av[5]	NULL	終端用

引数の準備ができたら、BootstrapMain()を呼び出す。BootstrapMain()での処理は、-xNNN で指定された処理の種類で、処理が分岐する。呼び出しを BootstrapMain() に統一しているのは、子プロセスでの初期化処理をまとめて書くためである。
ちなみに BootstrapMain()は、initdb 時の初期データベースを作成する際にも呼び出される。そのときは、postmaster から fork()される訳ではないので、様々な初期化処理を必要とする。この初期化部分は、グローバル変数 IsUnderPostmaster をみて必要かどうか判断する。

postmaster から呼び出されたときの BootstrapMain()では、プロセス ID の設定、引数の取り出し、シグナルの設定変更などのいくつかの初期化処理を経た後、switch 文を使って-xNNN で指定された処理の種類で分岐する。分岐した部分のそれぞれの処理については、後述する。postmaster から呼び出された場合は、switch 文で分岐してそれぞれの処理を行った後は、switch 文の最後で proc_exit(0) を呼び出してプロセスを終了する。従って、BootstrapMain()から返ってくることはない。
switch 文以降に書かれている処理は、initdb などから初期データベースの初期化のために呼び出された場合の処理である。

さて、SSDatabase() の親プロセス側に話を戻す。まず、fork() に失敗したときであるが、チェックポイントの場合は、そのままreturnしてpostmaster の処理を継続する。これは、チェックポイントは、

無理にここで実行しなくても支障が出ないからである[7]。しかし、fork()に失敗したのが、ＤＢのスタートアッププロセスやＤＢのシャットダウンの場合、これ以上処理が続けられなくなるので、ExitPostmaster(1)を呼び出して postmaster を終了する。
ＤＢのスタートアッププロセスとＤＢのシャットダウンプロセスは、通常のバックエンドプロセスと考えないので、バックエンドプロセスのリストには入れない。チェックポイントプロセスは、バックエンドプロセスのリストに追加する処理をここで行う。
あとは、fork()で生成したプロセスの ID を return して SSDatabase()関数を終了する。

6.2.2. BootstarpMain()でのスタートアップ処理

BootstarpMain() の中の switch 文による分岐で、ＤＢのスタートアップ処理としては、StartupXLOG() と LoadFreeSpaceMap() というの関数が呼び出される。前者の StartupXLOG() では、前回ＤＢが正常にシャットダウンされているかどうかをチェックして、正常にシャットダウンされていない場合、リカバリ処理を開始するというものである。後者の LoadFreeSpaceMap() は、postmaster が前回実行されていたときに管理していた FSM（Free Space Map：空き領域情報）を読み出して引き継ぐ処理である。
これらの処理を終えた後、proc_exit() を呼び出してプロセスを終了するわけだが、スタートアッププロセスが終了した時点で、シグナル SIGCHLD が postmaster に通知され、シグナルハンドラ reaper が起動されて、グローバル変数 StartupID を 0 にすることで、postmaster でクライアントからの接続要求を受け付けるようになる。

6.3. ＤＢのシャットダウンプロセス

ＤＢのシャットダウンプロセスの起動は、ソースコード上ではマクロ ShutdownDataBase() を呼び出すところから始まる。マクロ ShutdownDataBase() は次のように定義されている。

```
#define ShutdownDataBase()      SSDataBase(BS_XLOG_SHUTDOWN)
```

ＤＢのスタートアッププロセス同様に、SSDataBase() で子プロセスを起動して、BootstrapMain() 関数が呼び出される。

6.3.1. BootstarpMain()でのシャットダウン処理

BootstarpMain() の中の switch 文による分岐で、ＤＢのシャットダウン処理としては、ShutdownXLOG() と DumpFreeSpaceMap() という２つの関数が呼び出される。
DumpFreeSpaceMap() では、メモリ中で管理している FSM の保存を行う。どのように保存するかは、今回は調査していない。
ShutdownXLOG() では、チェックポイントの実行とコミットログ（CLOG）のバッファのフラッシュを行う。チェックポイントの実行といっても、既に postmaster の子プロセスになっていることから、これ以上プロセスを生成したりはしないで、このシャットダウンプロセスでチェックポイント処理を実行する。チェックポイントの処理としては、データバッファのフラッシュ、コミットログのフラッ

[7] ＤＢのシャットダウン処理で行われるチェックポイント処理は、ＤＢのシャットダウンプロセスで行うため、この部分のfork() のエラーとは無関係である。

シュ、チェックポイントログの作成、ログバッファのフラッシュ、コントロールファイル pg_control の更新を行う。ここで、コントロールファイル pg_control に、ＤＢが正常にシャットダウンされたというステータス DB_SHUTDOWNED を記入する。

6.4. チェックポイントプロセス

チェックポイントプロセスの起動は、ソースコード上ではマクロ CheckPointDataBase() を呼び出すところから始まる。マクロ CheckPointDataBase() は次のように定義されている。

```
#define CheckPointDataBase()    SSDataBase(BS_XLOG_CHECKPOINT)
```

ＤＢのスタートアッププロセス同様に、SSDataBase() で子プロセスを起動して、BootstrapMain() 関数が呼び出される。

6.4.1. BootstarpMain()でのチェックポイント処理

BootstarpMain()の中の switch 文による分岐で、チェックポイント処理としては、CreateDummyCaches()、CreateCheckPoint()、SetSavedRedoRecPtr() という３つの関数が呼び出される。

CreateDummyCaches()、CreateCheckPoint() というのは、チェックポイント処理の一連の流れである。チェックポイントプロセスのチェックポイント処理では、ＤＢのシャットダウンプロセスから呼ばれた場合と渡す引数が少し異なり、チェックポイントの処理内容が若干異なる。例えば、シャットダウン処理の場合、コントロールファイルの中に、正常終了したステータスを書き込むのに対して、こちらの通常処理中のシャットダウンでは、ＤＢが実行中を示す DB_IN_PRODUCTION になっている。

SetSavedRedoRecPtr() は、リカバリに必要な REDO ログの開始場所を postmaster に返すための処理である。

基本的なチェックポイントの処理内容は、シャットダウン時のチェックポイント処理と同じで、データバッファのフラッシュ、コミットログのフラッシュ、チェックポイントログの作成、ログバッファのフラッシュ、コントロールファイル pg_control の更新などを行う。

6.4.2. チェックポイントプロセス起動のタイミング

チェックポイントプロセスの起動されるタイミングは２つある。１つは、select() システムコールがタイムアウトして、ServerLoop()関数中の for ループで次の周になった場所で、checkpoint_timeout の時間が経過したことを検出した場合である。もう１つは、バックエンドプロセス（postgres）で更新処理を行った際などにログを記入し、ログセグメントが一定量更新された場合である。この場合は、バックエンドプロセス（postgres）から SIGUSR1 が発行され、postmaster のシグナルハンドラ sigusr1_handler が起動され、要求が PMSIGNAL_DO_CHECKPOINT であることを検出して、マクロ CheckPointDataBase() が呼び出される。後者で起動された場合も、checkpoint_timeout の残り時間はリセットしている。

そのほかに、チェックポイントの実行されるタイミングとして、クライアントからクエリとして

CHECKPOINT文が実行された場合である。クライアントからCHECKPOINT文が実行された場合は、そのクライアントを担当しているバックエンドプロセスでそのままチェックポイント処理が実行され、postmaster経由でチェックポイントプロセスが起動されることはない[8]。チェックポイント処理を実行する場合は、内部的に排他ロックを取得するので、同時に複数のチェックポイント処理が実行されることはない。

6.5. 統計情報収集プロセス

今回、統計情報収集プロセスは、ほとんど調べていない。統計情報収集プロセスのソースコードは、src/backend/postmaster/pgstat.c のコードになる。現在分かっている情報として、統計情報収集プロセスの起動のタイミングは、postmaster の起動時と、あと統計情報収集プロセスがいないことを検出した時点で、このプロセスの起動を試みるということである。

＜ＥＯＦ＞

[8] チェックポイントプロセスを起動している理由は、postmaster プロセスでチェックポイント処理を実行しないためである。また、ログセグメントが一定量に達した場合は、チェックポイント処理自体はそのバックエンドが行うべきものでないために postmaster に処理を依頼している。