PostgreSQL 解析資料

～ postgres プロセスの概要 ～

(株) NTT データ
オープンソース開発センタ
井久保 寛明

1. はじめに

本ドキュメントでは、postgres プロセスの処理概要について説明している。PostgreSQL では、postgres プロセスで DBMS としてのほとんどの処理を行う。１つの資料では、クエリ処理の詳細までは説明しきれないため､本ドキュメントでは postgres プロセスの初期化部分とその後の制御部分を中心に説明する。パーザ、アナライザ、リライタ、プランナ、エグゼキュータなどのクエリ処理は、別のドキュメントでまとめるものとする。

1.1. 対象バージョン

本ドキュメントは、PostgreSQL7.4.3 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. 用語の定義

スタンドアロン	postgres プロセスをコマンドラインから単独で起動する方法。 postmaster と同時に起動することはできない
libpq プロトコル	PostgreSQL のクライアントとサーバ間の通信プロトコル

1.2.1. libpq プロトコルについての補足

PostgreSQL は、クライアントとサーバ間での通信に libpq プロトコルと呼ばれる独自のプロトコルを使っている。PostgreSQL 7.3.x までは、このプロトコルバージョンが 2.0 であったが、PostgreSQL 7.4.x ではプロトコルバージョンが 3.0 に上がった。プロトコルのバージョンによってデータの処理法方が異なるのだが、PostgreSQL 7.4.3 でも両方のプロトコルをサポートできるように、処理の分岐が入っている。例えば、サーバ側の PostgreSQL が 7.4.x でクライアントが 7.3.x の場合は、libpq プロトコル 2.0 を使って通信が行える。

2. postgres プロセスの概要

ここでは、postgres プロセスについて簡単に説明する。postmaster や postmaster から fork された直後の動作についての詳細は、postmaster の資料にまとめてあるので、そちらを参照して欲しい。

2.1. プロセス概要

postgres プロセスは、２つの方法で起動される可能性がある。１つは、PostgreSQL がオンラインで通常のクエリ処理を行っている状態のときに、クライアントからの接続要求を受けて、postmaster から fork されて起動される。もう１つは、メンテナンスなどのときに、コマンドラインから postgres コマンドを直接実行して、スタンドアロンの postgres として動作する場合である。

postmaster が起動されているときの PostgreSQL のプロセス構成は、次の図のようになっている。

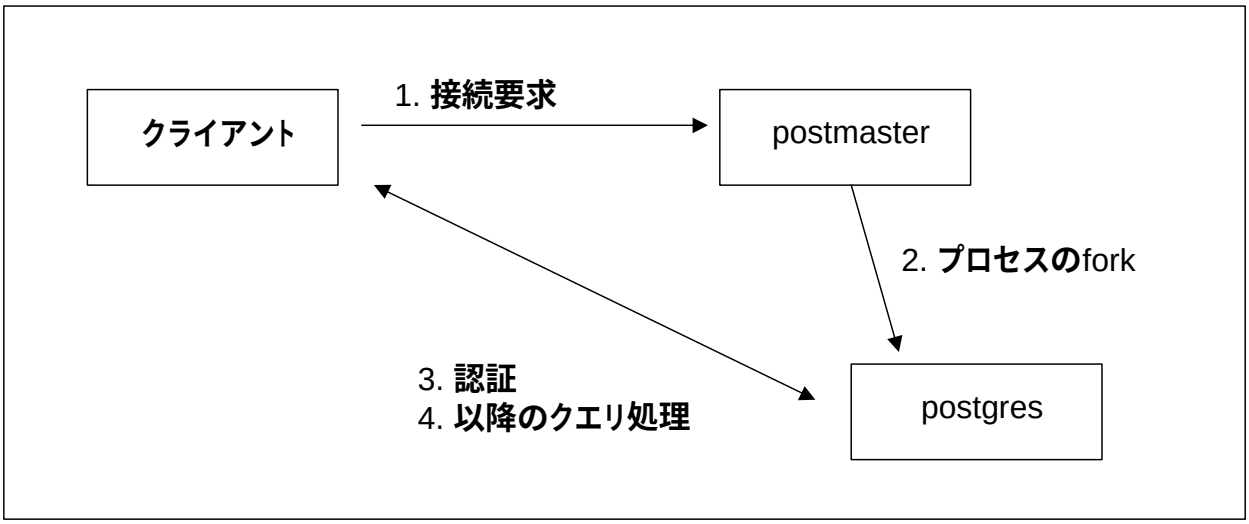

図 2-1 postgres プロセスの起動

まず、クライアントが postmaster プロセスに接続要求を出す。postmaster は、クライアントの接続要求を受け付けると、プロセスを fork してバックエンドプロセスを生成する。生成された直後のバックエンドプロセスは、厳密には postgres プロセスとは言えない。新しいバックエンドプロセスが生成されると、最初にクライアント認証を行う。クライアント認証を終えて、postgres プロセスの処理を開始した時点で postgres プロセスといえる状態になる。ソースコード中で見ると PostgresMain() に処理が移るところで、postgres プロセスになったと言ってもいいだろう。

スタンドアロンの postgres プロセスは、コマンドラインから postgres を直接起動する。特に入力ファイルを指定しなければ、対話型のインタフェースが起動される。

2.2. ソースコード

postgres プロセスの主なソースコードは、src/backend/tcop ディレクトリに入っている。postgres プロセスとしての開始場所となるのは、postgres.c の中の PostgresMain() 関数である。スタンドアロンで実行された場合も、postmaster から起動された場合も、postgres プロセスは PostgresMain() に入ってくる。main() 関数から PostgresMain() 関数を起動するシーケンスは次のようになっている。

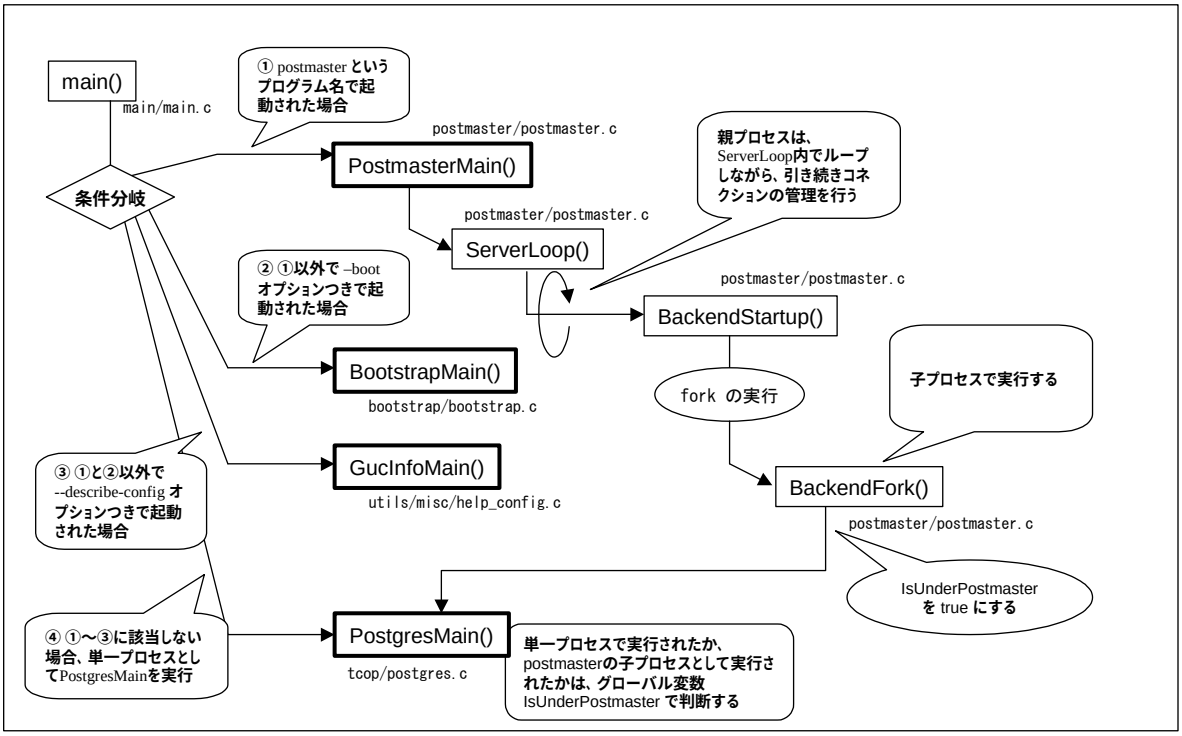

図 2-2 PostgresMain()が呼ばれるまでの関数フロー

PostgresMain() 関数をはじめ、stc/backend/tcop/postgres.c ほとんどのコードは、スタンドアロンで実行された場合と postmaster から fork された場合の両方のコードが一緒に書いてある。これらを分岐させるのが IsUnderPostmaster というグローバル変数であり、処理の分岐は次のように書かれている。

```
if( IsUnderPostmaster ) {
        /* postmaster から fork されてきた場合の処理 */
} else {
        /* スタンドアロンで postgres が起動された場合の処理 */
}
```

主な処理の分岐理由は、postgres プロセスの初期化の段階で、postmaster から fork された場合は各モジュールの初期化が終わっているのに対して、スタンドアロンの場合はそれらの初期化が必要なためである。例えば、共有メモリなどの共有モジュールの初期化やトランザクションログファイルをオープンしてリカバリ処理を行ったりするような処理である。

src/backend/tcop 以下には、次のようなファイルがある。

dest.c	クエリの実行結果の送り先(クライアントまたは上位のクエリノードのタプルの保存場所）を支援するモジュール。
fastpath.c	フロントエンドからの function の呼び出しを支援する関数群。
postgres.c	PosgresMain() を含む postgres プロセスの制御部分を書いたファイル。クエリの処理の各フェーズでモジュールを呼び出すところまで記述されていて、その後の処理は、それぞれの処理を担当するモジュールに移る。
pquery.c	ポータル（後述）を支援するモジュール。
utility.c	DDL などのユーティリティ系の SQL コマンドの処理の振り分けなどを支援する関数群。ユーティリティ系の SQL は、一度、この中で定義されている ProcessUtility() 関数に入った後、src/backend/commands ディレクトリなどのソースコードに処理が振り分けられる。

3. PostgresMain() 関数からの処理フロー

この章では、PostgresMain() から始まる処理を見ていく。はじめに概要を説明して、それから個別の処理の詳細を紹介していく。

3.1. 概要

PostgresMain() の処理フローの概要は次の図のようになっている。

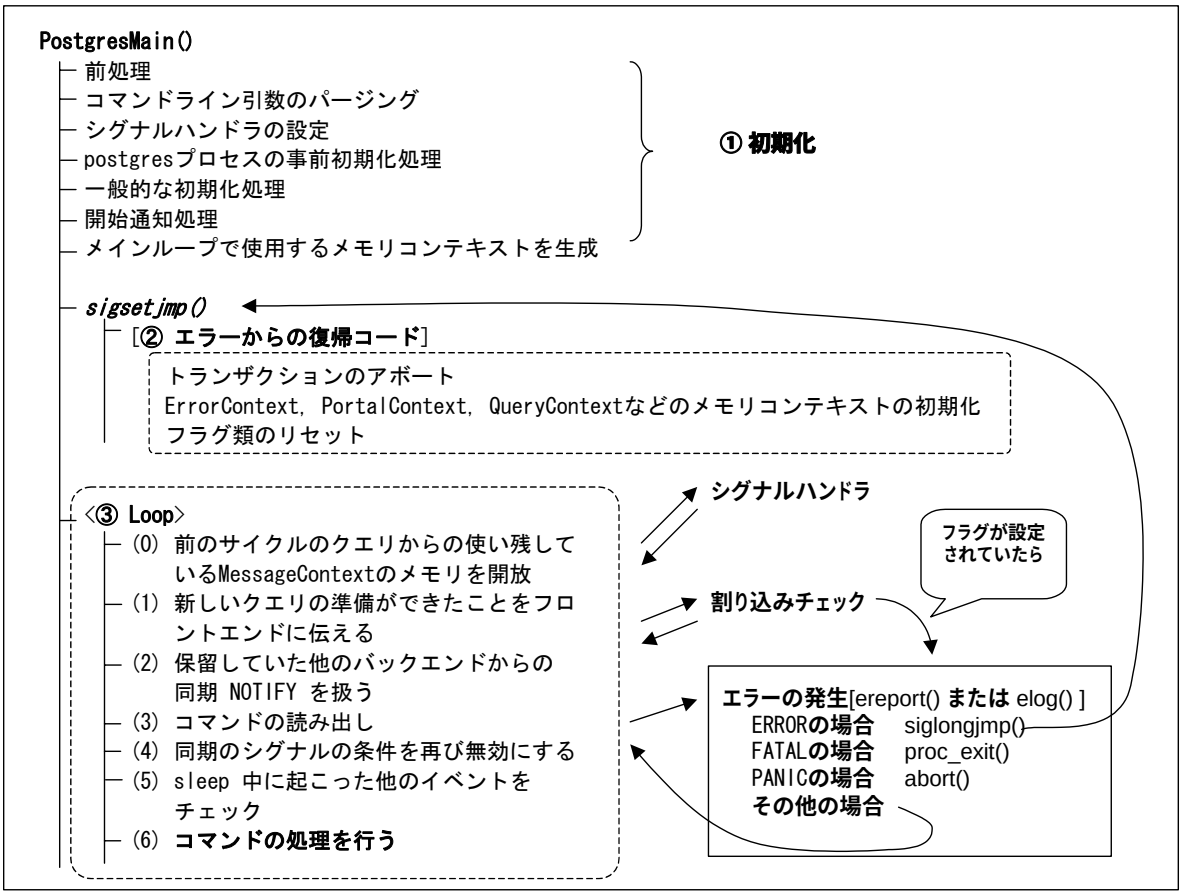

図 3-1 PostgresMain() の処理フロー

PostgresMain() の処理は「①初期化」、「②エラーからの復帰コード」、「③Loop」に分けられる。
①の初期化処理では、コマンドライン引数のパージングや各種モジュールの初期化などを行っている。
②のエラーからの復帰コードでは、PostgreSQL で ereport(ERROR, ...) のようなコードでエラーを発生させた場合に、バックエンドで次のトランザクションを受け付けられるように、バックエンドをエラーから復帰させる処理である。
③の Loop の部分は、PostgreSQL のバックエンドのメインのループである。クライアントからのコマンドの読み出しを行い、SQL コマンドを処理していく。
そのほかに、シグナルハンドラによるシグナル処理がある。

3.2. 初期化

PostgresMain() での初期化処理を見ていく。postmaster から fork された場合と、スタンドアロンで動作が大きく異なるのは、この初期化フェーズと SQL 文を受け取るところくらいである。

初期化での主な処理は次のようなものである。これらを順に説明していく。

1. 前処理
2. コマンドライン引数のパージング
3. シグナルハンドラの設定
4. postgres プロセスの事前初期化処理
5. 一般的な初期化処理
6. 開始通知処理
7. メインループで使用するメモリコンテキストの生成

3.2.1. 前処理

ソースコード中で、コマンドライン引数のパージング前で行われる処理と初期化を便宜上、前処理と呼ぶことにする。ここでは、usage の表示や簡単な初期化処理を行う。

まず、第１引数が --help, -?, --version, -V の場合の処理がある。これは、PostgreSQL の main() 関数が１つだけ定義されていて、そこから処理が分岐することに由来する。従って、処理が分岐されてからその処理向けの usage を表示するようになっている。第１引数が、--help, -? の場合はヘルプを表示して終了する。--version, -V の場合は、バージョン情報の表示だけを行って終了する。
実際にこれらの処理に入るのは、スタンドアロンで起動された場合である。

第１引数の処理で終了しなかった場合、通常の postgres の処理に入っていく。ここで行っているのは次のようなものである。

- グローバル変数 MyProc の初期化
- スタンドアロンの場合、メモリマネージャの初期化 [MemoryContextInit()]
- ps コマンドの表示が「postgres: startup」となるように設定
- 処理モード（グローバル変数 ProcessingMode）を InitProcessing にする
- コマンドラインオプションの Noversion, EchoQuery のデフォルト値を設定する
- スタンドアロンの場合、GUC の初期化と PGDATA のディレクトリを設定する
  [InitializeGUCOptions()]

処理モードは、BootstrapProcessing、InitProcessing、NormalProcessing の３種類がある。postgres プロセスの初期化処理が終わるまで、InitProcessing にしておく。これを設定することによって、一部 index やカタログ周りの動作が異なるのと、エラー処理が若干異なる。index やカタログ周りは詳しく調べていない。エラーについては、elog(ERROR, ...) や ereport(ERROR, ...) のようにエラーのレベルが ERROR であっても、処理モードが InitProcessing の場合は深刻なエラーとみなして全て FATAL エラー扱いとなる。後述するが、クエリ処理中の ERROR はエラーの復帰処理を得て、再びクエリ処理を行えるようになるのに対して、FATAL エラーではプロセスを終了するという違いになる。

この前処理の段階で、スタンドアロンの場合に、メモリマネージャの初期化や GUC の初期化など、postmaster から起動された場合には postmaster ですでに終わっている処理のいくつかを実行している。

3.2.2. コマンドライン引数のパージング

postgres プロセスに渡すコマンドライン引数には２つの形式がある。これは、スタンドアロン用と postmaster から fork されたとき用である。postmaster から fork された場合も、コマンドライン引数の形式で引数が渡される。

スタンドアロンの場合の形式は、次のようになる。

```
postgres [switches] [databasename]
```

もし、データベース名が省略されていたら、OS にログインしている OS のユーザ名を使用する。

postmaster から開始されていた場合の形式は、次のようになる。

```
postgres [secure switches] -p databasename [insecure switches]
```

-p が渡されるまでは、全てのオプションが有効である。-p 以降に現れる引数は、クライアントの接続要求の options フィールドから渡されたものである。-p 以降の引数は、セキュリティ上の理由から、どのスイッチが実行できるか制限されている。

オプションの内容は、src/backend/tcop/postgres.c の usage() を見るか、postgres コマンドを --help オプション付で実行すると簡単な説明を見ることができるので、ここでは特に紹介しない。

コマンドライン引数のパージング処理の流れとしては、次のようになっている。

1. ループの中の switch 文で、各オプションを処理していく。-p を見つけたら secure = false を設定して、これ以降のいくつかのオプションは設定できなくする。
2. デバッグレベルに合わせて、表示項目追加のための GUC オプションを追加する。
3. postmaster から起動されている場合、クライアントから受け取ったスタートアップパケットから取り出した GUC オプションを追加する。
4. オプションを設定した結果に対して、いくつかのチェックを行う。ここで行うチェックは、例えばスタンドアロンのときに、データベースクラスタのディレクトリが指定されているかどうか、ログへの表示項目のオプションに矛盾がないかなどの非常に簡単なものである。

これらの処理が終わった時点で、スタンドアロンの場合は postgresql.conf ファイルの読み込みを行う。

3.2.3. シグナルハンドラの設定

postmaster から開始された場合、postmaster で子プロセスを fork する前にシグナルをブロックしているので、シグナルハンドラを設定する前にシグナルを受け取ったかどうかという微妙な状態は存在しない。

postgres プロセス用のシグナルハンドラを設定した後、シグナルのマスクを変更する。postmaster と

異なり、SIGQUIT はすぐに受け付けられるようにしておく。
シグナル処理の詳細は後述する。

3.2.4. postgres プロセスの事前初期化処理[BaseInit()]

postgres プロセスの初期化は２つに分かれている。最初の処理が BaseInit()という関数で実行され、２番目の初期化処理が InitPostgres() という関数で実行される。これらが分かれている理由は、前者は xlog 関連の初期化が終わる前に実行しなければならないもの（つまり xlog 関連の初期化処理に必要なもの）であり、後者は xlog 関連の初期化が終わった後でなければ実行できないものである。

ただし、postmaster から起動されている場合は、xlog 関連の初期化は済んでいるので BaseInit()と InitPostgres()は連続して実行される。

まず、postmaster から起動された場合だが、単に BaseInit() を実行したら終わりである。この場合、BaseInit() では、次のことが行われる。

1. デバッグファイルの初期化（DebugFileOpen()）
2. ストレージマネジャの初期化（smgrinit()）
3. バッファマネージャの初期化（共有バッファの初期化関数 InitBufferPoolAccess()とローカルバッファの初期化関数 InitLocalBuffer()を呼び出す）

ソースコードとしては、これらの処理の前に共有メモリの初期化関数 InitCommunication() を呼び出しているが、postmaster からこの関数を呼び出しても何もしないで返ってくる。
共有バッファの全体の初期化は postmaster 起動時に終わっているので、ここではプロセスごとの共有バッファ管理領域の初期化が行われる。

次に、スタンドアロンの場合を説明する。スタンドアロンの場合は、まず、BaseInit() を呼び出す前に、postmaster の初期化フェーズで行っていることのいくつかを行う。内容は次のようなものである。

- ダイナミックローディングのためのパスの取得
- $PGDATA 以下のロックファイルの確認と作成
- xlog のパスの取得

BaseInit() 実行時は、postmaster から起動されたときとほぼ同様で、次の処理が実行される。

1. 共有メモリの初期化 （InitCommunication()）
2. デバッグファイルの初期化（DebugFileOpen()）
3. ストレージマネジャの初期化（smgrinit()）
4. バッファマネージャの初期化（共有バッファの初期化関数 InitBufferPoolAccess()とローカルバッファの初期化関数 InitLocalBuffer()を呼び出す）

共有メモリの初期化と書いているが、スタンドアロンの場合は、共有メモリではなくプロセスのローカルメモリに postmaster が共有メモリに作成するのと同じデータ構造を作成する。

BaseInit()の実行が完了したら、xlogの初期化のためにStartupXLOG()を実行する。続いて、FSM[1]の情報を読み出すためにLoadFreeSpaceMap()を実行する
ここまでで、postgres の初期化処理の前半が終了である。

3.2.5. 一般的な初期化処理 [InitPostgres()]

こちらの初期化処理は、スタンドアロンの場合も postmaster から起動された場合も共通に行う初期化処理である。
共通の初期化処理は、全て InitPostgres() 関数の中で定義されている。一部のモジュールが使えなかったりするので、スタートアップの順番に非常に気を使っている。例えば、データベースの OID を取得するときに、最初はロックが取れないので仮の状態として取得したり、リレーションキャッシュも３段階に分けて初期化したりしている。

InitPostgres()で実行されるのは、次のようなものである。

1. データベースの OID とデータベースクラスタのパスを取得する（ここでは、ロックが使えないため、ロックを取得せずにデータを読み出す。そのため、削除中のデータベースであっても OID が読めてしまう。）
2. データベースのパスを設定し、そのディレクトリに移動する
3. 共有メモリのプロセス管理構造体（PGPROC 構造体）のエントリを作成する
4. 共有 inval マネージャのエントリを取得して登録する（ここで、バックエンドの ID が決まる）
5. リレーションキャッシュの第１次初期化を行う（ここでは、仮の情報として必要最低限のデフォルトのエントリを設定する）
6. カタログキャッシュを初期化する（ここでは、キャッシュのための構造を初期化するだけである。実際にカタログ情報がキャッシュされるのは、はじめてそのカタログにアクセスしたときである。）
7. ポータルマネージャの初期化を行う（ポータルを管理するメモリコンテキストとポータルのハッシュを生成する）
8. データベースにアクセスする前に、トランザクションを開始する（この状態で、ようやくデータベースに正しくアクセスできるようになった）
9. リレーションキャッシュの第２次初期化（ここでは、本当のシステムカタログにアクセスできるようになったので、今まで使っていた仮のエントリを捨てて、データベースからリレーション情報を読み込む）
10. postgres ユーザの ID を算出する
11. １で取得したデータベースの OID が有効なものであるかチェックを行うと同時に、データベースのエンコードやその他のデータベースに関連して設定してある情報を読み込み設定する。（１ではロックなどが取得できなかったので、削除中のデータベースにアクセスしている可能性があった。ここで、pg_database にアクセスすることで、データベースを確認するので、何度もこのテーブルにアクセスしなくていいように、ついでに関連する情報を取得する）
12. リレーションキャッシュの第３次初期化

[1] Free Space Map。リレーションのファイル中の空き領域を管理するしくみ。詳細は、ストレージの資料を参照。

13. スーパーユーザ用のデータベース接続エントリが残っているか、データベースの同時接続数のチェックを行う
14. デフォルトの名前空間のサーチパスを設定する
15. クライアントのエンコードを初期化する
16. postgres プロセスを終了前にクリーンナップを行うためのコールバックを登録する
17. 8 で開始したトランザクションを終了する

InitPostgres() の処理が終了した時点で、前処理で設定した処理モードを（グローバル変数 ProcessingMode）を NormalProcessing にして、これ以降は通常の処理として実行される。

3.2.6. 開始通知処理

ここまでで、初期化フェーズが完了である。初期化処理が完了した旨を伝える処理を行う。
postmasterから起動されている場合は、クライアントにキャンセルキー[2]を送信する。スタンドアロンの場合は、バナーを表示することで、初期化が完了する。

3.2.7. メッセージコンテキストの作成

このあとメインの処理に入るのだが、ここで、メインループで使用するメモリコンテキスト MessageContext を作成する。

3.3. エラー処理とエラーからの復帰処理

PostgreSQL のソースコードでは、デバッグやワーニングなどのログの取得とエラーメッセージの出力を、まとめて１つのインタフェースで処理する。
古いインタフェースは elog() であり、7.4 から ereport() が使われるようになった。ただし、一部 elog() も残っているようである。

ereport() は次のようにマクロで定義されている。

```
#define ereport(elevel, rest)  ¥
    (errstart(elevel, __FILE__, __LINE__, PG_FUNCNAME_MACRO) ? ¥
     (errfinish rest) : (void) 0)
```

第１引数の elevel によって、どのレベルのエラーを出力するかを制御できると同時に、エラーのレベルによって必要に応じてエラー処理を実行する。

エラーレベルは次のようなものがある。

DEBUG5	debug_print_rewritten が true になる。
DEBUG4	debug_print_plan が true になる。
DEBUG3	debug_print_parse が true になる。
DEBUG2	log_statement が true になる。

[2] キャンセルパケットを送信した際に、正しいクライアントからの要求であることを識別するためのキー。

DEBUG1	log_connections が true になる。
LOG	サーバ側だけに出力するメッセージ。
COMMERROR	クライアントの通信エラーによるメッセージの出力。
INFO	クライアントに情報を送るためメッセージ。
NOTICE	クエリ操作に関するユーザ向けのメッセージ。クライアントとサーバに出力。
WARNING	警告。
ERROR	エラー。トランザクションをアボートする。
FATAL	エラー。プロセスをアボートする。
PANIC	エラー。自分のプロセス終了後、postmaster によって他のバックエンドを強制終了させる。その後、postmaster により共有メモリの再初期化とリカバリ処理が行われる。

DEBUG5 から WARNING までは、単なるログ出力機能として働く。ERROR, FATAL, PANIC では、エラーメッセージを出力後、エラー処理に入る。
エラー処理では、PANIC の場合、メッセージを出力した後、abort() によりプロセスが終了される。postgres プロセスの処理はこれだけで、後の処理は postmaster に引き継がれる。abort() では、終了ステータスが 0 ではないため、postmaster で FatalError が true になる。同時に他のバックエンドに SIGQUIT を送り、exit()による即時終了を行う。全てのバックエンドが終了したら、共有メモリの再初期化処理が行って、データベースのスタートアップ処理を起動してリカバリを行う（図 3-2)。
FATAL の場合は、メッセージを出力した後、proc_exit()を呼び出し、共有メモリ関連の終了処理を行ってプロセスを終了する。この場合は、バックエンドが正常に終了した場合とほとんど違いはない。

ERROR の場合は、メッセージを出力した後、siglongjmp() を実行して PostgresMain() 関数の sigsetjmp の位置まで戻る。そして、メインループへの復帰のための処理を行う（図 3-1)。

ERROR の際のメインループへの主な復帰処理は、次のようなものである。

1. 割り込み関連を無効にする
2. リカバリのためメモリコンテキストを ErrorContext に変更する
3. トランザクションをアボートさせる
4. メモリコンテキストを TopMemoryContext に戻し、次回のエラー処理のために ErrorContext を初期化しておく
5. PortalContext、QueryContext を NULL に設定する。
6. エラーの処理中であることを示すフラグをリセットする

1 の割り込み関連の処理は、次のことを行う

- 即時割込みを禁止
- キャンセルパケット保留フラグのリセット
- 割込みを制御する InterruptHoldoffCount と CritSectionCount を初期化
- アラームを無効にする処理
- Notify の割込みを無効にする処理

３のトランザクションのアボートは、AbortCurrentTransaction() を呼び出して、処理をトランザクションマネージャに任せる。

PortalContext と QueryContext の２つのメモリコンテキストは、別のメモリコンテキストの一部であるため、単に NULL にするだけでよい。これらは、CreatePortal() で割り当てられ、DropPortal() の時に破棄される。CreatePortal()、DropPortal() の呼び出しの流れは、概要であるが「クエリ実行の概要」で紹介する。

６では、エラー関連のフラグとして、InError を false に設定する。また、トランザクションをアボートさせているので、xact_started も false に設定しておく。

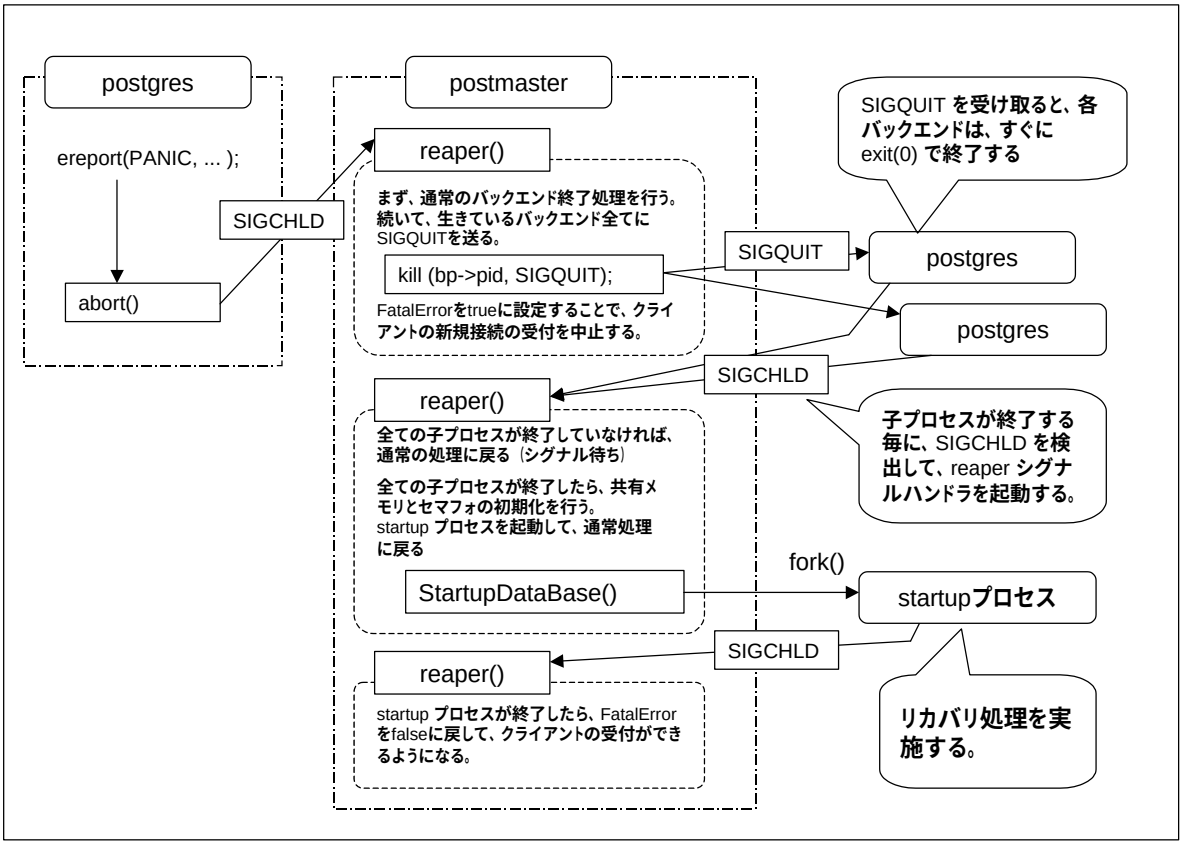

図 3-2 PANIC からの再開処理

3.4. メインの処理

図 3-1 の最後の部分でループがあるが、これがエラーが発生しない状態での正常処理を受け持つ部分である。ひとたびエラーが発生すると、前述のように ereport()を使ってエラー処理ルーチンを呼び出す。復帰可能なエラーであれば、sigsetjmp() のところまでジャンプしてきて、復帰処理を行った後、またこのメインループに戻ってくる。

必ずエラーからの復帰処理が行われるので、このループに入ってきている時点で、エラーは起こっていない状態であるということを仮定してよい。

ループの中では、次のような処理が行われる。

(0). ループ内の初期化
(1). 新しいクエリの対応の準備ができたことをフロントエンドに伝える
(2). 保留していた他のバックエンドからの同期 NOTIFY を扱う
(3). SQL コマンドの読み取り
(4). 割込み処理を保留するようにする
(5). SIGHUP を受け取っていたら、設定ファイルの読み直しを行う
(6). SQL コマンドの実行を行う

処理の最初に付けた番号は、(0)を除いて、ソースコード中のコメントの番号に対応している。

クライアントと通信を行っている場合は、(3)の SQL コマンドの読み込みで、クライアントからの要求を待っていることになる。
(6)では、SQL コマンドの実行は、クライアントから受け取ったパケットによって処理が分かれる。

'Q'	simple query。従来からのオーソドックスなクエリ処理要求。
'P'	parse。クエリのパージングのみを実行する。
'B'	bind。パージングの終わっているステートメントに対してポータルを作成する。
'E'	execute。ポータル名を指定して、そこから指定した行数の結果を取り出す。必要があれば、クエリの実行を行う。
'F'	fast function call。クライアントからの function の実行。
'C'	クローズ処理。次のバッファに、クローズするもののタイプが入っており、現在は２つのクローズが対象が存在する。'S'：prepare ステートメントのクローズ。'P'： ポータルのクローズ
'D'	２つのタイプの describe 処理がある。'S'：prepare ステートメントの describe。'P'： ポータルの describe。
'H'	flush 処理。クエリ結果のクライアントへのフラッシュを行う。
'S'	sync 処理。クライアントとサーバ間のメッセージの同期を取る。auto commit で実行されていた場合、auto commit 終了のタイミングで実行され、トランザクションを終了させる。begin で開始したトランザクションブロック内では、'S'を実行してもトランザクションを終了することはない。
'X', EOF	'X'はフロントエンドからの終了要求。EOF は、予期しない通信切断。
'd', 'c', 'f'	COPY が途中で失敗してもフロントエンドがデータを送り続けている場合に、これらのパケットを受け取ってしまう。これらは無視する。

'P','B','E','C','D','H','S' は、PostgreSQL 7.4 以降で追加された機能である。7.3.x までは、クエリ処理は全て'Q'で実行されていた。これに対して、クライアント側からクエリ処理を分割して呼び出せるよ

うにしたり、通信のタイミングを制御したりするためのプロトコルが追加された。

ソースコードを読んでいく上では、まずは'Q'の simple query を押さえておけばいいだろう。
simple quey の場合のクエリの処理は、exec_simple_query() 関数の中で全て実行される。
exec_simple_query()の中については、「4 クエリ実行の概要」で後述する。

フロントエンドから終了が通知されると'X'が送られてくる。EOF はフロントエンドとの接続が切れてしまった場合だが、この部分ではどちらにしても通常の終了処理を行うため、この２つの区別はされていない。途中になっている処理があればアボートさせて、共有メモリまわりの後始末をしたらプロセスを終了するだけである。

3.5. シグナル処理と割り込み処理

OS などの話では、シグナル処理は割り込み処理の１つであり、UNIX プログラミングにおいては、同じ意味で使われることもある。ここでは、割り込み処理という言葉は、postgres プロセスにおける割込み処理という意味で使っていく。
postgres プロセスでは、いくつかのシグナルは、シグナルが発生した時点ではシグナル発生フラグを設定するだけしか行わない。そして、コード中の都合のいい場所で、割り込みがあったかどうかをフラグの確認しに行って、割り込みがあったことを確認したら、その処理を実行する。

PostgreSQL のシグナルブロック関数を呼び出すと、次のシグナルをブロックする。

```
SIGHUP, SIGQUIT, SIGTERM, SIGALRM, SIGINT, SIGUSR1, SIGUSR2, SIGCHLD, SIGWINCH, SIGFPE
```

postmaster ではこのまま使用するのだが、postgres プロセスでは SIGQUIT はブロックしないで、いつでも受け付けられるようにしている。また、SIGUSR1 は、シグナルハンドラを SIG_IGN にするため、実質的にはブロックされない。

シグナルハンドラは、次のように定義されている。

シグナル	シグナルハンドラ	処理概要
SIGHUP	SigHupHandler	次のタイミングで、設定ファイルの読み直しをするように got_SIGHUP フラグを設定する。
SIGINT	StatementCancelHandler	実行中のクエリをアボートするためのフラグ QueryCancelPending と割込みが発生したことを判別するフラグ InterruptPending を設定する。
SIGTERM	die	実行中のクエリをアボートして、プロセスを終了するためのフラグ ProcDiePending と割込みが発生したことを判別するフラグ InterruptPending を設定する。postmaster のファストシャットダ

		ウンに使用される。
SIGQUIT	quickdie	postmaster の即時終了の時に実行される。共有メモリが壊れていることを想定しているので、proc_exit()による共有メモリ関連のコールバック関数を呼ばずに、exit(1)で即時終了する。
SIGALRM	handle_sig_alarm	デッドロック検出ロジックの開始と実行時間によるSQL処理のタイムアウトのために使用する。内部的には、どちらか一方しか有効にならないようになっている[3]。
SIGUSR2	Async_NotifyHandler	カタログキャッシュの失効通知などに使用される。
SIGFPE	FloatExceptionHandler	浮動小数点のエラーも PostgreSQL でハンドリングする。

シグナルハンドラの処理のところに、フラグを立てるものがいくつか存在する。postgres プロセスでは、シグナルを受け取ったときはフラグを設定するだけで、実際の割り込み処理を別のタイミングで行われるものがある。それらを次の２つの「設定ファイルの読み直し」と「割り込み処理」で説明する。

3.5.1. 設定ファイルの読み直し

postmaster が設定ファイルの読み直しのシグナル SIGHUP を受け取ったら、それらは実行中の postgres プロセスにもシグナル SIGHUP で伝播される。しかし、postgres プロセスでは、クエリの実行中などの場合には、SIGHUP を受け取ったからといって、すぐに設定ファイルを読み直すわけにはいかない。そのため、フラグ got_SIGHUP を設定しておいて、都合のいいタイミングで、ファイルの読み直しを行う。

実際に設定ファイルを読み直せるタイミングは、クライアントから送られてきた SQL の処理が一段落したところ、つまり、SQL 文の切れ目（ひとまとめに送られたものの間では切れない）である。「図 3-1 PostgresMain() の処理フロー」でいうと、「③Loop」内の(5)の処理である。
got_SIGHUP フラグをチェックして、true であれば ProcessConfigFile(PGC_SIGHUP)を実行して競ってファイルの読み直しを行う。

3.5.2. 割り込み処理

postgres プロセスでは、クエリキャンセルやプロセスアボートのシグナルも、設定ファイルの読み直し同様、いつでも処理できるわけではない。例えば、共有メモリのデータ構造の操作中などのクリティカルセクションで急に処理を止めると、共有メモリを破壊してしまう。ローカルな処理中であっても、エラー処理後は、リソースを再初期化して新しい SQL の処理に復帰しなければならないので、いつでも割込みでクエリ処理を中止するという訳にはいかない。そのため、クエリキャンセルやプロセ

[3] SQLの処理時間にタイムアウトがある場合は、デッドロック検出は行わない。これは、デッドロックが発生してもタイムアウトにより自動的に解決されるからである。

スアボートのシグナルを受け取ったときも、フラグを設定するだけで、割り込み処理自体はあとから都合のいい場所で実行する。
割り込み処理は、クエリ処理の中止に都合に良いところで、CHECK_FOR_INTERRUPTS() というマクロを使ってチェックを行う。保留中の割り込み処理があれば、割り込み処理関数 ProcessInterrupts()を呼び出して割り込みの処理を実施する。
CHECK_FOR_INTERRUPTS() マクロは次のように定義されている。

```
#define CHECK_FOR_INTERRUPTS() ¥
    do { ¥
        if (InterruptPending) ¥
            ProcessInterrupts(); ¥
    } while(0)
```

つまり、InterruptPending のフラグが設定されていれば、割り込み処理関数に入るわけである。
ちなみに InterruptPending が true になるのは、postmaster から shutdown かクエリキャンセルのシグナルが届いたときだけである。

ProcessInterrupts() に入ると、もう少し厳密なチェックを行う。割り込み阻止カウンタ InterruptHoldoffCount とクリティカルセクション変数 CritSectionCount をチェックして、これらが設定されている場合も割り込みは行わない。

ProcessInterrupts()では、プロセス終了待ちフラグ ProcDiePending が true なら ereport(FATAL, ...) を利用してプロセスを終了する。また、クエリキャンセル待ちフラグ QueryCancelPending が true の場合、ereport(ERROR, ...) を利用して、setlongjmp() を使ってエラーの復帰処理のところに移動して、クエリをアボートする。

grepコマンドなどを利用してCHECK_FOR_INTERRUPTS() の場所を調べてみると、vacuumなどの処理の中には比較的たくさん埋め込まれているが、エグゼキュータの処理には数箇所しか埋め込まれていない。エグゼキュータの処理におけるソースコード上のCHECK_FOR_INTERRUPTS()の数は少ないが、プラン木の各ノードの処理開始時[4]や１行のスキャンを完了したタイミング[5]など、実行頻度の高いところに入っている。
postmaster を終了するときに、ファストシャットダウンを実行してもなかなか終了しない場合があるのは、長い時間、CHECK_FOR_INTERRUPTS() が実行されない処理に入っていると考えられる。
CHECK_FOR_INTERRUPTS() を増やすか、シグナルを受け取ったら、とりあえず CHECK_FOR_INTERRUPTS() を実行してみてもいいように思う。

[4] プラン木のノードを１つ手繰るたびに実行される。
[5] バッファから１レコード読み込んだタイミングである。例えば、長くVACUUMが実行されていないと、プラン木の末端のノードで有効なレコードを１つ取り出すまでに、たくさんのレコードを読み込むことがある。こういう状況を回避するために、１レコード読み込んだタイミングで、CHECK_FOR_INTERRUPTS() が実行されている。

4. クエリ実行の概要

３章では、postgres プロセスの処理の流れを説明したが、ここではクエリ処理のところに焦点を当てて、流れを追っていく。

このドキュメントでは、src/backend/tcop モジュールを中心に説明しているので、各フェーズの処理の詳細については、別のドキュメントに個別にまとめることにする。ここでは、それぞれどのようなデータが渡され、どのような処理をして、どのようなデータを生成するかの説明と、それぞれ処理の開始場所が src/backend/tcop/postgres.c の中のどこであり、どのモジュールのどの関数へ入っていくのかを紹介するにとどめる。

PostgreSQL でのクエリ処理の概要は次のようになる。

1. SQL の読み込み
2. パーズ木の生成 (字句解析、構文解析)
3. クエリ木の生成 (意味解析)
4. リライト
5. 実行プランの生成
6. エグゼキュータでの実行

これは主に DML と呼ばれるデータ操作用の SQL が実行されたときの処理内容である。DDL やユーティリティ系のコマンドを実行したときも、同じような順に関数呼び出しが行われるのだが、関数内で何もしない処理もある。

関数の呼び出しの概要をクエリ処理に対応させると次の図のようになる。

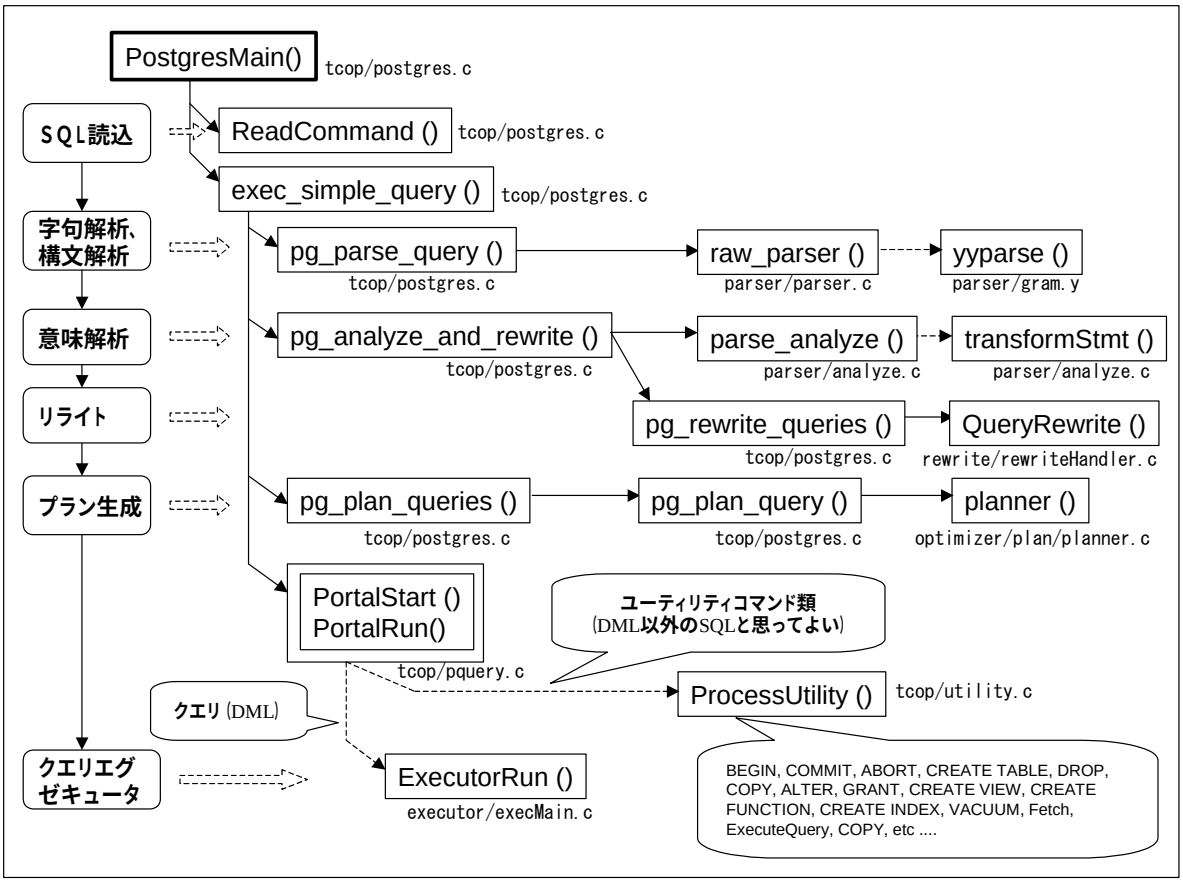

図 4-1 クエリの処理手順とそれに対応する関数フロー

4.1. SQL の読み込み

SQL の読み込みは、ReadCommand() から始まる。ReadCommand() の中で処理が分岐して、postmaster から開始されている場合は、SocketBackend() という関数に処理が移り、フロントエンドから送られてきたパケットから SQL 文を読み取る。libpq プロトコルのバージョン 3.0 未満（PostgreSQL7.3.x 以前）では、ここでクエリ文字列を読み込む。しかし、libpq プロトコルのバージョン 3.0 からは、ここに来るよりも前のクライアントからの要求を読み取った時点で、クエリ文字列の読み出しも完了している。
一方、スタンドアロンの場合は、InteractiveBackend() という関数に処理が移り、プロンプトを表示して標準入力から SQL 文を読み込む。

どちらにしても、ReadCommand() が完了するとクエリ文字列が文字列バッファに格納されている。

4.2. パーズ木の生成（字句解析、構文解析）

テキストの文字列として読み込んだ SQL 文を解析して、パーズ木を生成する。パーズ木は、SQL 文を文法どおり木構造に変換しただけの非常に単純な変換である。
パーズ木の生成の際に、テキストの文字列からトークンを切り出す字句解析と、切り出したトークンを構文規則に合わせて解釈して木構造を生成する構文解析を行う。字句解析は flex を使用し、構文解析には bison を使用している。

パーズ木の生成は、与えられた SQL 文全部に対して一度に行われる。構文解析の結果、パーズ木のリストが生成される。

字句解析、構文解析の間は、DB に一切アクセスを行わないように実装されている。

パーズ木生成のエントリポイントは、pg_parse_query()である。この中からパーザモジュールの raw_parser() を呼び出すことで、パージングを行う。字句解析用の字句解析ルールが定義されているのが scan.l で、構文解析用の構文規則が定義されているのが gram.y である。

4.3. クエリ木の生成（意味解析）

パーズ木からクエリ木を生成する処理を意味解析と呼ぶ。意味解析では、与えられたパーズ木に含まれるリレーションが実際にデータベースに存在しているどうかのチェック[6]、アクセス権のチェック、型のチェックなどが行われる。また、SQLの書き方で省略されているような項目を全て補う。クエリ木は、パーズ木の不足を補ったようなデータである。
状況によって１つのパーズ木が複数のクエリ木に分割される可能性があるため、意味解析の結果は、クエリ木のリストとして返される。

意味解析のエントリポイントは、parse_analyze()である。この関数はパーザ内の意味解析モジュールの関数である。

4.4. リライト

リライトとは、PostgreSQL の VIEW やルールの実装である。リライトでは、クエリ木のリストを受け取って、VIEW やルールを適用して、クエリ木の書換えを行う。リライトの結果、複数のクエリ木が１つにまとめられたり、１つのクエリ木が複数に分割されたりすることがある。

リライタのエントリポイントは、pg_rewrite_queries() であり、クエリ木のリストをもらってクエリ木のリストを返す。pg_rewrite_queries() の中からリライトモジュールの QueryRewrite()を呼び出し、１つずつクエリ木を処理する。

4.5. 実行プランの生成

実行プランの生成では、クエリ木からプラン木の生成が行われる。実行プランの生成は、まず、ルールベースで定型の書換えを行い、その後、考えられる多数の実行プランを生成する。ここで生成した多数の実行プランに対して統計情報を用いたコスト計算を行い、最適の実行プランを選択する。

クエリ木は、SQL 文をそのまま表したような木構造であるのに対して、プラン木は、エグゼキュータの実行指示になっているため、関係代数の演算を表すような構造である。

[6] CREATE TABLEなどの一部のDDLでは、リレーションのチェックはここでは行われず、実行時にチェックされる。

プランナのエントリポイントは、pg_plan_query() である。ここからプランナモジュールの planner() を呼び出す。

4.6. エグゼキュータでの実行

PostgreSQL7.3.xまでは、クエリの実行は、PostgresMain()からpg_exec_query_string()[7] が呼ばれ、その中でDML系のSQLならProcessQuery() を経由してExecutorRun()が実行されていた。ユーティリティ系のSQLならProcessQuery() の代わりにProcessUtility() が呼ばれるという非常に単純な呼び出しであった。

PostgreSQL 7.4 以降では、PostgresMain()からexec_simple_query() が呼ばれた後、ポータルという構造を経由してエグゼキュータであるExecutorRun() やユーティリティ系の開始ポイントである ProcessUtility() が呼ばれるようになった[8]。

4.6.1. ポータル

ポータルは、もともとカーソルのように SQL を部分的に逐次実行するのを支援するための機能である。しかし、PostgreSQL 7.4 からは、通常のクエリ処理でもポータルを使うようになった。ポータルでは、実行結果の取得データ件数を指示できるので、実行結果の全件取得（FETCH_ALL）というオプションを使って実行するのである。

ポータルを使った呼び出し手順でも、プラン木を生成するまでは同じである。クエリ処理の流れに従ってプラン木まで生成したら、クエリ文字列、クエリ木、プラン木を全てポータルの管理構造に入れた後、ポータルに実行を任せる。

ポータル関連の関数の呼び出し順は、次のようになる。

```
1.  CreatePortal("", true, true);
2.  PortalDefineQuery(portal, query_string, commandTag, querytree_list, plantree_list, MessageContext);
3.  PortalStart(portal, NULL);
4.  PortalSetResultFormat(portal, 1, &format);
5.  CreateDestReceiver(dest, portal);
6.  PortalRun(portal, FETCH_ALL, receiver, receiver, completionTag);
7.  (*receiver->rDestroy) (receiver);
8.  PortalDrop(portal, false);
```

1. まず、ポータルの管理構造を生成するために CreatePotal() を実行する。
2. そして、PortalDefineQuery()にクエリ文字列、クエリ木、プラン木などのデータを渡し、ポータルの設定を行う。
3. ここで、PortalStart() でポータルの開始処理を行う。この中では、ポータルの戦略（後述）の選択が行われ、その戦略に合わせていくつかの処理を行う。PORTAL_ONE_SELECT（select 文が１つの）の場合、ExecutorStart()でエグゼキュータの開始処理が行われる。最も多く選択される PORTAL_MULTI_QUERY では、ここでは何も行わず、PortalRun()が呼ばれてから、

[7] PostgreSQL 7.3.xまでのsimple queryのプロトコルを処理する関数。PostgreSQL 7.4 のexec_simple_query() 相当の関数。
[8] このドキュメントでは、exec_simple_query() を利用する方法に着目しているが、7.4 からはparse, bind, execute と分割して実行することもできる。この場合も、基本的な流れは、exec_simple_query() とあまり変わらない。

ExecutorStart() は呼び出される。

4. 続いて、パーズ木をみて結果データの送信形式を決定する。そして、その結果で送信形式の初期化 PortalSetResultFormat()を行う。
5. データの受取側の準備として、データの受取側の初期化処理 CreateDestReceiver()を行う。
6. ここまででポータルの実行準備が完了しているので、PortalRun() を実行して SQL の処理を行う。PortalStart()で選択したポータルの戦略によって処理が分かれる。途中の過程が若干異なるが、ユーティリティ系のコマンドの場合 ProcessUtility() が実行され、DML 系の SQL の場合 ExecutorRun()が実行される。ここで、PortalRun()には、FETCH_ALL という引数が渡され、全ての処理を終了させる。また、EXPLAIN ANALYZE などの一部のユーティリティ系のコマンドでは、ProcessUtiliy()の後に SQL の実行が必要なので ExecutorRun()を実行するものもある。
7. ポータルの実行が完了したら、結果の受取モジュールを完了させるため、(*receiver->rDestroy)() という関数ポインタで、受取モジュールを完了させる。これは、受け取り手が SQL の種類によって異なるために関数ポインタで処理している。
8. 最後にポータルの削除処理を行って、１つのパーズ木の処理が完了である。

ここまでが、ポータルを使った基本的な処理の流れである。ポータルの実行戦略には、次の３種類がある。

PORTAL_ONE_SELECT	クエリ木[9]の段階で、１つのSELECT文しか含んでいない場合に選択される。結果が要求されるごとに、エグゼキュータを実行する。この戦略は、保持可能な(holdable)カーソルをサポートする。
PORTAL_UTIL_SELECT	表示するような結果を返すユーティリティ文の場合に選択される（例えば、EXPLAIN や SHOW などである）。最初の実行でステートメントを実行し、結果をポータルのタプルストアに保存する。それから、タプルストア[10]に保存した結果をクライアントに返す。
PORTAL_MULTI_QUERY	その他の全ての場合である。ループを利用して、PORTAL_ONE_SELECT、PORTAL_UTIL_SELECT 相当の処理も実行する。この戦略では、部分実行をサポートしていない。ポータルのクエリ処理は、最初の実行時に完了する。

ほとんどのユーティリティ系コマンドと insert, update, delete のような DML は、PORTAL_MULTI_QUERY を使って実行される。select 文でも select into の場合も PORTAL_MULTI_QUERY になる。また、複数の select 文をセミコロンで区切っても PORTAL_MULTI_QUERY になる。

それでは、ポータルの実行戦略毎に、PortalStart(), PortalRun() で行われる処理フローをもう少し具体的に見ていくことにする。

---

[9] ソースコード中のコメントではpasetree と書いてあったりするが、ポータルに渡しているのはクエリ木だけで、パーズ木は渡していないので、ポータルのコード中でパーズ木と書いてあるのはクエリ木のことである。
[10] タプルを中間結果として格納するためのデータ構造。

まず、PORTAL_ONE_SELECT であるが、図 4-2 のようになる。クエリとしては、次のようなものを実行した場合だと考えればよい。

```
SELECT * FROM foo;
```

```
├ CreatePortal()
├ PortalDefineQuery()
├ PortalStart()
│   ├ ChoosePortalStrategy()
│   ├ CreateQueryDesc()
│   ├ ExecutorStart()
│   ├ [ポータルのカーソル位置の初期化]
├ PortalSetResultFormat()
├ CreateDestReceiver()
├ PortalRun()
│   ├ [メモリコンテキストの設定]
│   ├ PortalRunSelect()
│   │   ├ PortalGetQueryDesc()
│   │   └ ExecutorRun()
├ (*receiver->rDestroy) ()
├ PortalDrop()
```

図 4-2 PORTAL_ONE_SELECT の場合の関数呼び出し

PortalStart() の中で、ChooseStrategy() を使ってポータルの実行戦略を選択し、PORTAL_ONE_SELECT が選ばれる。その後、PortalStart() 内で、クエリデスクリプタの生成と ExecutorStart() によるエグゼキュータの開始が行われる。
PortalRun() では、ポータルの実行戦略によって条件分岐し、PortalRunSelect()が実行される。その中からエグゼキュータの実行関数 ExecutorRun() が呼び出されている。

次に、PORTAL_UTIL_SELECT の場合である。いくつかの場合が考えられるので、次の EXPLAIN を実行した場合を例に説明する。

```
EXPLAIN SELECT * FROM foo;
```

PORTAL_UTIL_SELECT の場合の処理の流れは、図 4-3 のようになる。

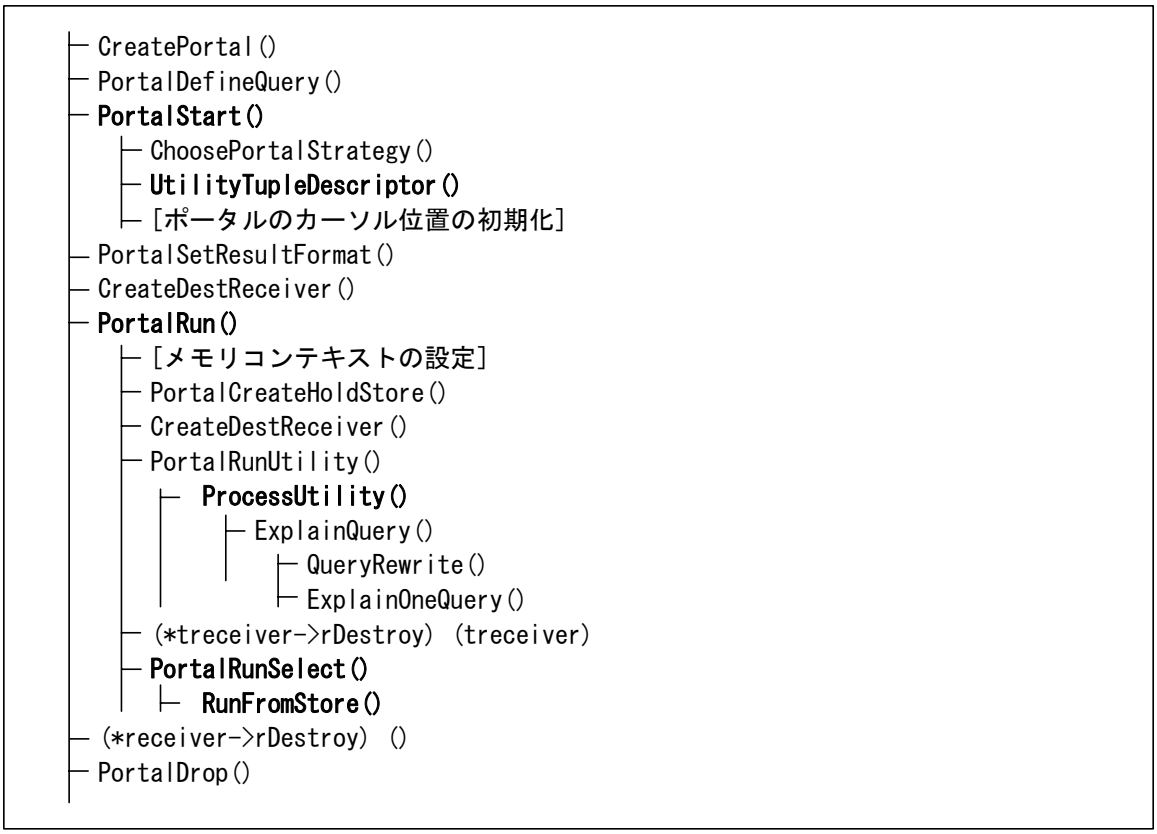

図 4-3 PORTAL_UTIL_SELECT の場合の関数呼び出し

まず、PortalStart() の中でポータルの実行戦略を選択したときに PORTAL_UTIL_SELECT が選ばれる。その後、PortalStart() の中でユーティティ用のタプルデスクリプタ生成関数 UtilityTupleDescriptor() が呼び出される。
PortalRun() に入ると、ポータルの実行戦略によって条件分岐する。ここでは、まず、ポータルでの実行結果を保存するために PortalCreateHoldStore() が呼び出されてタプルストアが作成される。続いて、実行結果の受取が直接クライアントに送られるのではなく、一時的にタプルストアに保存されるため、CreateDestReciever()を実行して、タプルストアを結果の受取先として指定する。 その後、ProcessUtility() が呼ばれ、その中で EXPLAIN が処理され、EXPLAIN の結果がタプルストアに格納される。
ProcessUtility() が実行されたあと、PortalRunSelect() を実行して ProcessUtility()で処理した結果を取り出してクライアントに送る。このとき、PORTAL_ONE_SELECT ではここでエグゼキュータを実行していたので、ExecutorRun() を呼び出していたが、この場合は、すでにタプルストアに結果が格納されているので、RunFromStore() を使って結果を取り出す。

最後に、ポータルの実行戦略が PORTAL_MULTI_QUERY の場合である。ユーティリティ系コマンドの場合と DML 系 SQL の場合がある。
ここでは、それぞれ、次の２つの SQL を想定して話を進める。

```
CREATE TABLE bar ( id INT );
INSERT INTO bar values ( 1 );
```

PORTAL_MULTI_QUERY の場合の処理の流れは、図 4-4 のようになっている。

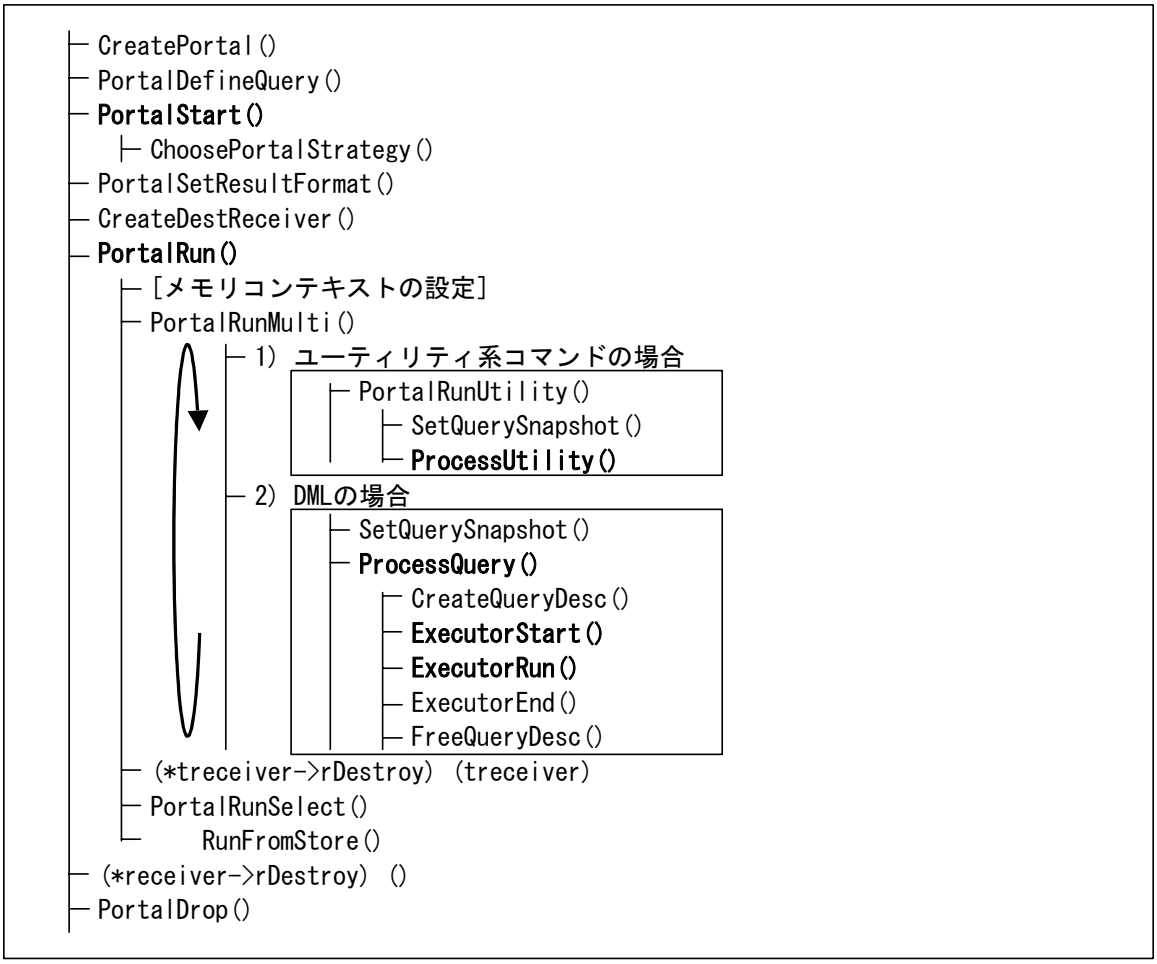

図 4-4 PORTAL_MULTI_QUERY の場合の関数呼び出し

まず、PortalStart() であるが、他と同様、ポータルの実行戦略を選択する。ここで、実行戦略 PORTAL_MULTI_QUERY が選ばれると、PortalStart() ではこれ以上何もしない。
PortalRun() に入るとメモリコンテキストの設定を行ったら、PortalRunMulti()という関数に入り、実行部分はこの関数の中で制御する。
PortalRunMulti() では、ループを使って複数の SQL を処理していく。渡されているクエリ木のリストを手繰って処理していく。ユーティリティ系のコマンドが来た場合、図のように ProcessUtility() が呼ばれ、ユーティリティ系コマンドの処理に入っていく。このとき必要に応じて、SetQuerySnapshot() が呼ばれる。（今回は CREATE TABLE を実行したので呼ばれた。）また、ここで、DML 系の SQL が呼ばれた場合は、ProcessQuery() が呼び出され、その中で ExecutorStart() や ExecutorRun()などが実行される。

4.7. ユーティリティ系コマンドの処理

ここまでは、関数呼び出しに従ってクエリ処理の流れを順番に説明してきた。DML では、これらのほとんど全ての処理を行うのだが、DDL などのユーティティ系のコマンドの場合、いくつかの実行しない処理があるので、ここで説明しておく。

DDL などのユーティティ系のコマンドの処理でも、関数呼び出しの流れ自体は DML 系の SQL と同様に行われる。しかし、リライト処理と実行プランの生成のところでは、関数に入った後何も行わないですぐに抜けてくる。つまり、ユーティティ系のコマンドではクエリ木までしか生成しない。実際、ユーティリティ系のコマンドの処理系が使うのは、プラン木ではなくクエリ木である。

エグゼキュータのところでは、DML 系の SQL は src/backend/executor 以下に定義されているモジュールで処理がされるが、ユーティリティ系コマンドは、src/backend/command 以下に定義されているモジュールを中心に処理される。これ以外に、一部トランザクションマネージャなどの関数を直接呼び出すものもある。

ユーティリティ系コマンドでもポータルを使うため、PortalStart()を実行して、PortalRun()を実行するところは同じである。その後、src/backend/tcop/utility.c に定義されている ProcessUtility()を呼び出して処理を分岐する。全てのユーティリティ系コマンドは、必ず ProcessUtility() から処理を開始する。

4.8. 関数の呼び出しとデータ形式

ここでは、それぞれの関数呼び出しに対して、どのようなデータが渡されていくかを見ていく。
関数呼び出しと、それらに渡されるデータを図にしたのが次の図である。これは、DML 系の SQL の処理とデータの流れになる。

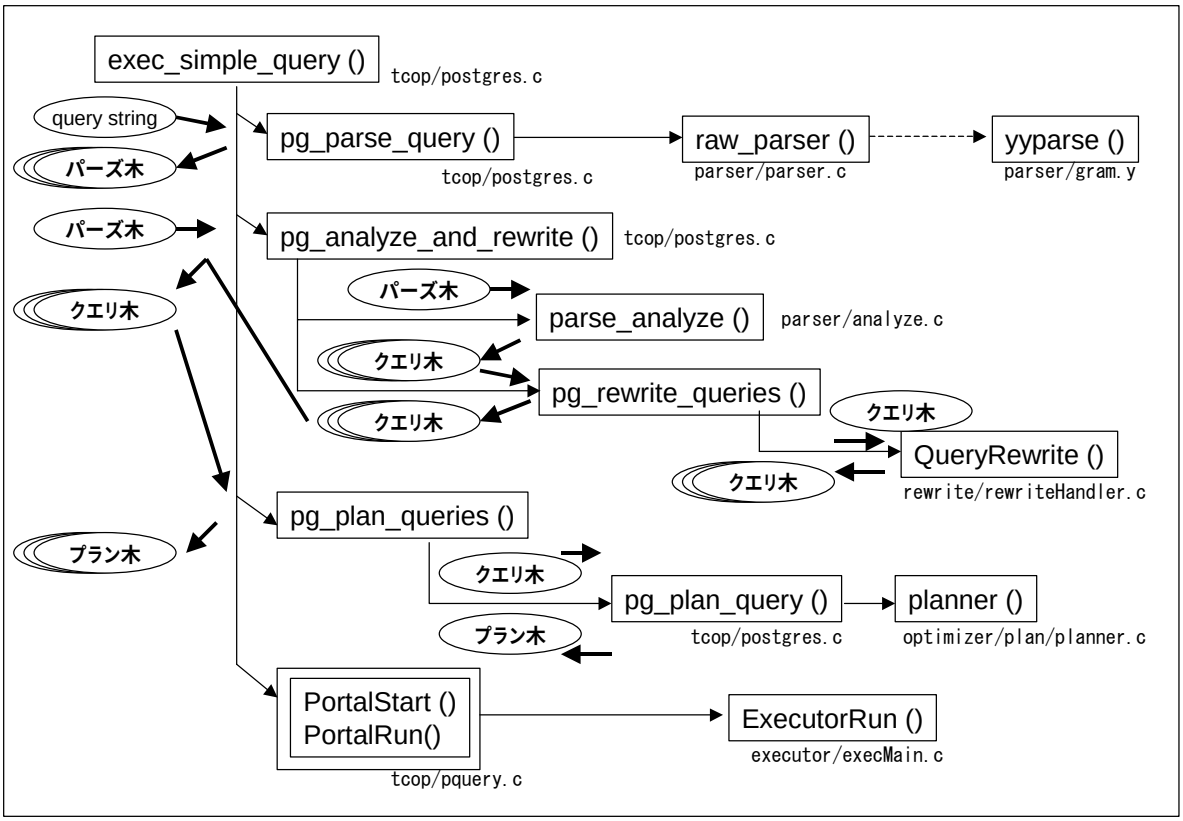

図 4-5 関数フローとデータ形式

まず、クライアントから渡されるのが、テキストの SQL 文 query_string である。これを pg_parse_query() によってパーザにかけると、パーズ木のリストが生成される。パーズ木は、渡された SQL 文のテキストを、単純に木構造に変換した程度のものである。
続いて、パーズ木のリストから１つのパーズ木を取り出して、意味解析器とリライタにかける。アナライザにかけると、１つのパーズ木が複数のクエリ木に変換される可能性があるので、parse_analyze() の戻り値は、クエリ木のリストである。pg_rewrite_queries() で、このリストから１つずつクエリ木を取り出し、リライタにかける。リライタも１つのクエリ木から複数のクエリ木を生成する可能性があるので、クエリ木のリストを返す。pg_analyze_and_rewrite()による、意味解析器とリライタの処理結果は、クエリ木である。
さらに、この処理済のクエリ木のリストを pg_plan_queries() に渡し、そこで、１つずつクエリ木を取り出してプランナにかけ、実行プランであるプラン木を生成する。実行プランは１つのクエリ木に対して１つ生成される。ただし、pg_plan_queries() では、クエリ木のリストをもらっているので、結果もプラン木のリストになる。
ところどころで、１つの木構造が複数に分割する可能性があるためにリストになっているところがあるが、単純な SQL 文の場合は１つの木構造が引き渡されていく。

プラン木のリストが生成されたところで、ポータルのところで説明したように PortalDefineQuery() を使って、クエリ文字列、クエリ木、プラン木などをひとまとめにしてポータルのデータ構造に保存する。そして、そのポータルのデータ構造を使って、PortalRun()の中からエグゼキュータなどが実行される。

ユーティティ系コマンドでも、関数の呼び出しはほとんど同じである。ただし、pg_plan_queris() でプラン木は生成されず、その前のクエリ木が処理系に渡される。また、PortalRun() から実行されるのも ExecutorRun() ではなく、ProcessUtility()となる。

＜ＥＯＦ＞