新連載！

内部を知って業務に活かす

PostgreSQL研究所

石井 達夫 ISHII Tatsuo
ishii@postgresql.org

第1回 PostgreSQLの構造と機能
～ディレクトリ構成とデバッガによる解析方法

はじめに

PostgreSQLについてすでに過去2年に渡って連載を続けさせていただきました．3年目の今期は新しい試みとして，PostgreSQLの内部構造を1年かけて解説します．

1年分の連載というと，本誌の場合，だいたい1回あたり8ページ，全6回で50ページほどになります．このページ数の範囲で本当に踏み込んだ解説は無理かもしれませんが，その代わり，実際にソースコードレベルで探求したい方の指針となるポインタ情報は豊富に示していきたいと思っています．

なお，説明の前提になるPostgreSQLのバージョンは，現時点で最新安定版の8.0系とします．また，PostgreSQLの稼働するプラットフォームはUNIXまたはLinuxなどのUNIX互換OSとします．Windowsは若干動きが違いますが，データベースエンジンの内部構造はほぼUNIX版と同じです．

PostgreSQLの利用形態

PostgreSQLはいわゆるクライアント／サーバ型のアーキテクチャを採用しています．PostgreSQLを利用するアプリケーションは，まず定められたプロトコルにしたがってTCP/IPまたはUNIXドメインソケットを通じてPostgreSQLのサーバに接続しなければなりません（図1）．

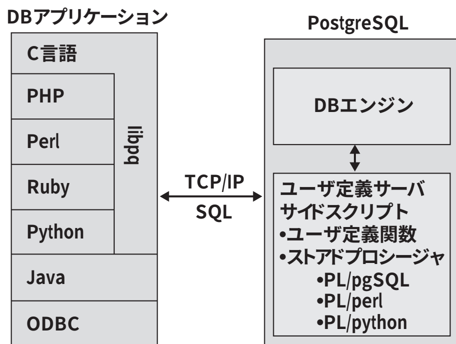
PostgreSQLでは，クライアントをフロントエンド（frontend），サーバをバックエンド（backend）と呼びます．バックエンドは複数のプロセスから構成されていますが，それについてはこの後で説明します．

フロントエンドと通信

フロントエンドとバックエンドの通信プロトコルはPostgreSQLの付属ドキュメントの「第42章フロントエンド／バックエンドプロトコル」に詳しく書かれています．詳細はそちらを見てほしいのですが，基本的には問い合わせ（SQL文）をフロントエンドからバックエンドに送信し，その結果が複数のパケットに分かれてバックエンドからフロントエンドに返ってくるような仕掛けになっています．

もちろん実際には接続の開始処理やエラー処理など，いろいろなことをやらなければならないので，結構複雑な処理をする必要があります．これらをすべてのフロントエンドで実装するのは大変なので，C言語で書かれた「libpq」という共通ライブラリが提供されていて，これを使えばより簡単にバックエンドとの通信を

▼図1 PostgreSQLの利用形態



行うことができます．PostgreSQLはPerlやPHPなど，C言語以外のプログラミング言語をサポートしていますが，これらは内部でlibpqを呼び出しています．

一方，libpqを使わずに独自にPostgreSQLとの通信を行うライブラリもあります．代表的なものがJavaで，「タイプ4」と呼ばれる全部Javaで書かれたPostgreSQL用のJDBCドライバは，Javaで独自にネットワーク通信を行うことにより，libpqに頼ることなく実装されています[注1]．

ODBCも同じ方法をとっています．

バックエンド

一方バックエンドのほうは，中心となるのがデータベース処理を実行する「データベースエンジン」です．この中身を解説するのが本連載の主要目的になります．

バックエンドにはアプリケーションを実行するインフラストラクチャがあります．すなわち，ユーザ定義のサーバサイドスクリプト（PostgreSQLでは「ユーザ定義関数」あるいは「プロシージャ言語」と呼びます）群で，言語別に表1のものが利用できます．

PostgreSQLでは，ユーザが独自にサーバサイドスクリプトを定義することができます．そこで，各種言語用のサーバサイドスクリプトがサードパーティによって提供されています．主なものとしては，Tcl，Ruby，Java，PHP，R，Shellなどがあります．

PostgreSQLの構造

それでは，もう少し詳しくPostgreSQLの構造を見てみましょう．図2をご覧ください．

表1◎サーバサイドスクリプト

●言語	●サーバサイドスクリプト
C	C関数
SQL	SQL関数
独自言語	PL/pgSQL
Perl	PL/Perl
Python	PL/Python

バックエンド側はいくつかのプロセスから構成されています．

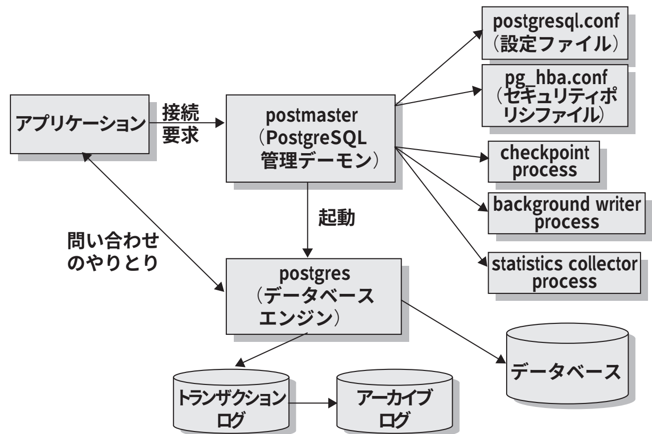
postmaster

「postmaster」は，バックエンドを管理する常駐プロセスです．通常UNIXドメインソケットまたはTCP/IPの5432番のポートをlistenしていて，フロントエンドがここに接続するのを待ち受けています．フロントエンドがpostmasterに接続すると，セキュリティポリシファイルであるpg_hba.confによって接続の可否がチェックされます．ポリシにより，特定のIPアドレスやネットワークからの接続を拒否したり，接続をUNIXドメインソケットのみに制限したりすることもできます．このチェックがOKならば，その後設定によってユーザやグループのチェックを受けます．これらをすべて通過すると，postgresプロセスが起動（fork/exec）され，以後フロントエンドはpostgresとのみ通信し，postmasterの手を離れます．

postgres

「postgres」は，データベースエンジン本体です．フロントエンドからの問い合わせを受け取ってはデータベースを検索して結果を返したり，データベースの更新を行ったりします．また，更新データは「トラザンクションログ」と呼ばれる特別なファイルに記録され，

▼図2　PostgreSQLの構造



注1）フロントエンドではありませんが，pgpool（http://www2b.biglobe.ne.jp/ caco/pgpool/）も独自にPostgreSQLとの通信処理を実装しています．

万が一停電などでデータベースの不整合が起きた際にはリカバリ処理を行うために使用されます．さらに適当なタイミングでアーカイブログ領域に移され，リカバリ処理に備えて保存が可能になっています．

その他のプロセス

これ以外にいくつか補助プロセスがあります．これらはすべてpostmasterから起動されるプロセスです．

●チェックポイントプロセス

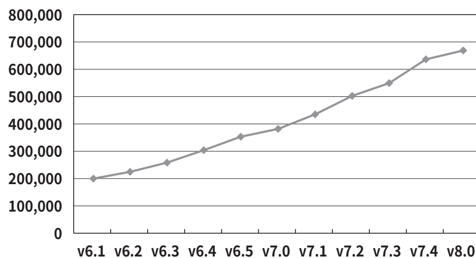
「チェックポイントプロセス」（checkpoint process）は，定期的に古いトランザクションログをアーカイブログに移動し，トランザクションログが満杯になるのを防ぎます．そのほか，テーブルやインデックスなどを「同期書き込み」と呼ばれる方法でメモリ上のバッファからハードディスク上に確実に書き込んで，停電などの際にデータが失われないようにするなどの重要な処理を行います[注2]．

チェックポイントプロセスは通常postgresql.confのcheckpoint_segments，checkpoint_timeoutで規定されるタイミングで自動的に起動されます．

●バックグラウンドライタプロセス

「バックグラウンドライタプロセス」（background writer process）は，共有メモリ上のバッファを最適なタイミングでハードディスクに書き出します．これによって，チェックポイントの際に大量のディスク書き込みが起きてパフォーマンスが劣化することを防ぎ，安定したパフォーマンスが維持できます．バックグラウンドライタプロセスは一度起動されたら以後常駐しますが，ずっと動き続けているわけではなく，postgresql.confのbgwriter_delayで規定される時間休止してはまた動くという動作を繰り返します．

▼図3　PostgreSQLのソースコード行数の変遷



●統計情報収集器プロセス

「統計情報収集器プロセス」（statistics collector process）は，テーブルへのアクセス回数やディスクへのアクセス回数などの情報を収集するプロセスです．ここで収集された情報はPostgreSQLがシステムとして利用するものではなく，データベースの管理者が参照してデータベースの管理に役立てるためのものです．したがって，このプロセスを起動しておかなくても，PostgreSQLの稼働自体には影響を与えません[注3]．

PostgreSQLのソースコード

PostgreSQLのおおまかな構造がわかったところで，今後のためにPostgreSQLのソースコードの構造を解説しておきましょう．

PostgreSQLが初めて基本的な機能を持ち合わせたときは20万行ほどだったソースコードは，今では70万行にもなっています（図3）．

これほどの量になると，闇雲にコードを読んでも理解するのは難しいでしょう．そこでまず本稿で大まかなソースコードの構造を理解していただきたいと思います．

トップレベル

PostgreSQLのソースコードを展開すると，トップレベルには表2のファイルやディレクトリがあります．PostgreSQLの本体はsrc以下にあります．本連載で解説の対象になるのもここです（表3）．

この中で中心となるのはbackend，bin，interfacesの各ディレクトリです．backendはバックエンドに対応し，binとinterfacesはフロントエンドに対応しま

注2）チェックポイント処理の詳細は本誌Vol.20に掲載された「徒然PostgreSQL散策 第6回トランザクションログ」で説明しています．バックナンバーをお持ちでない方はhttp://www2b.biglobe.ne.jp/ caco/support/webdb-pdfs/vol20_224-232.pdfで記事のPDFファイルを公開していますので，そちらをご覧ください．

注3）ただし，contribで提供されているautovacuumはこの情報を使うので影響を受けます．

表2◎トップレベルのファイルとディレクトリ

●名称	●内容
COPYRIGHT	著作権の表示
GNUMakefile	トップレベルのMakefile
GNUMakefile.in	Makefileのひな形
HISTORY	改訂履歴
INSTALL	インストール方法の概略
Makefile	ダミーのMakefile
README	概要の説明
aclocal.m4	config用ファイルの一部
config/	config用ファイルを格納するディレクトリ
configure	configureファイル
configure.in	configureファイルのひな形
contrib/	contributionプログラムのディレクトリ
doc/	ドキュメントディレクトリ
src/	ソースディレクトリ

す．

binには，psqlやinitdb，pg_dumpなどの各種ツールのソースがあります．interfacesには，PostgreSQLのC言語ライブラリであるlibpq，そしてC言語にSQLを埋め込むことができるecpgコマンドのソースがあります．

本連載で焦点を当てるのはもちろんbackendディレクトリです．その構造を表4に示します．一応コメントは付けていますが，たぶん初めてPostgreSQLのソースに触れる方はなんのことやら，だと思います．次回以降で順次解説していきますので，しばしの辛抱を．

バックエンドなどのヘッダファイルはまとめてincludeにあります（表5）．backendのディレクトリ構造を反映しているのですが，そのままではなく，基本的にサブディレクトリの下にさらにサブディレクトリを設けることはしていません．たとえば，backendディレクトリの下のutilsの下にはさらにadtなどのサブディレクトリがありますが，ここではそれは省略され，平坦な構造になっています．

表3◎src以下のファイルとディレクトリ

●名称	●内容
DEVELOPERS	開発者向けの注釈
Makefile	Makefile本体
Makefile.global	make用の設定値（configureが生成）
Makefile.global.in	configureが使用するMakefile.globalのひな型
Makefile.port	プラットフォーム依存のmake設定値．実際にはmakefile/Makefile.プラットフォームへのリンク（configureが生成）
Makefile.shlib	共有ライブラリ用のMakefile
backend/	バックエンドのソース一式
bcc32.mak	Win32ポート用のMakefile（Borland C++用）
bin/	psqlなどのUNIXコマンドのソース
corba/	CORBA対応の試み（未完成）
include/	ヘッダファイル
interfaces/	フロントエンドライブラリのソース一式
makefiles/	プラットフォーム依存のmake設定値
nls-global.mk	メッセージカタログ用Makefile用ルール
pl/	プロシージャ言語のソース
port/	プラットフォーム移植用のソース
template/	プラットフォーム依存の設定値
test/	各種テストツール
timezone/	タイムゾーンの実装
tools/	開発用の各種ツール，ドキュメント
tutorial/	チュートリアル
utils/	フロントエンド／バックエンド共通のモジュール
win32.mak	Win32ポート用のMakefile（Visual C++用）

ソースコードの歩き方

デバッガを使う

PostgreSQLのような巨大なシステムでは，目でソースコードの流れを追うのは容易ではありません．そこでお勧めなのが，gdbなどのデバッガを使って実際のコードの実行の流れを追うことです．デバッガとい

PostgreSQL研第1回　06.2.27　10:35 AM　ページ216

表4◎backend以下のディレクトリ

●名称	●内容
Makefile	Makefile
access/	各種アクセスメソッド（以下サブディレクトリ）
common/	共通関数
gist/	Generalized Search Tree（gist）という汎用的なインデックスメソッド
hash/	Hash
heap/	ヒープアクセス関数
index/	インデックスアクセス関数
nbtree/	Btree
rtree/	Rtree
transam/	トランザクション処理
bootstrap/	データベース初期化（initdbのとき）の処理
catalog/	システムカタログのハンドリング
commands/	比較的単純なSQL文を実行する処理
executor/	エグゼキュータ
lib/	共通関数
libpq/	PostgreSQLプロトコル関数
main/	postmaster, ブートストラップ，スタンドアロンpostgres用のメイン
nodes/	パースツリー操作関数
optimizer/	オプティマイザ
parser/	パーサ
port/	プラットフォーム依存コード
postmaster/	postmaster
regex/	正規表現処理
rewrite/	ルールとビュー
storage/	共有メモリ，ディスク上のストレジ，バッファなど，すべての1次/2次記憶管理（以下サブディレクトリ）
buffer/	バッファ管理
file/	ファイルアクセス
freespace/	Fee Space Map管理
ipc/	プロセス間通信
large_object/	ラージオブジェクトアクセス関数
lmgr/	ロックマネージャ
page/	ページアクセス関数
smgr/	ストレジマネージャ
tcop/	postgresプロセスの主要部分
utils/	さまざまなモジュール（以下サブディレクトリ）
adt/	各種組み込みデータ型
cache/	キャッシュ管理
error/	エラー処理関数
fmgr/	関数管理
hash/	hash関数
init/	データベースの初期化，postgresの初期処理
mb/	マルチバイト処理
misc/	その他
mmgr/	メモリ管理関数
resowner/	問い合わせの間だけ有効なデータ（バッファピンやテーブルロック）の管理
sort/	ソート処理
time/	トランザクションのタイムスタンプ管理

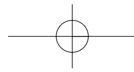
表5◎includeの下の主要ディレクトリとファイル

access/	funcapi.h	nodes/	pg_config_os.h@	postgres_fe.h	tcop/
bootstrap/	getaddrinfo.h	optimizer/	pgstat.h	postmaster/	utils/
c.h	getopt_long.h	parser/	pgtime.h	regex/	
catalog/	lib/	pg_config.h	port/	rewrite/	
commands/	libpq/	pg_config.h.in	port.h	rusagestub.h	
executor/	mb/	pg_config.h.win32	postgres.h	storage/	
fmgr.h	miscadmin.h	pg_config_manual.h	postgres_ext.h	strdup.h	

うと尻込みする方もいらっしゃるかもしれませんが，単に実行の流れを追うだけなら非常に簡単です．

以下，具体的な手順を説明します．この実行例はLinuxのものですが，UNIX系のOSならばあまり違い

PostgreSQL研第1回　06.2.27　10:35 AM　ページ217



はないと思います注4．

　ただ，そのためには多少準備が必要で，PostgreSQLをデバッグシンボル付でコンパイルしておかなければなりません．PostgreSQLを構築する際にconfigureに--enable-debugオプションを追加してください．また，できればsrc/Makefile.globalを編集し，

```
CFLAGS = -O2 -Wall -Wmissing-prototypes -
Wpointer-arith -fno-strict-aliasing -g
```
(実際は1行)

のような行から「-O2」を削除して

```
CFLAGS = -Wall -Wmissing-prototypes -Wpointer-
arith -fno-strict-aliasing -g
```
(実際は1行)

とします．-O2はコンパイラの最適化オプションです．これを有効にすると，コードの実行順序が入れ替わってしまったりしてソースとの対応を追っ掛けるのが困難になることがあるので，外してしまってください．もちろん，こうして作ったPostgreSQLの実行バイナリは大きくかつ遅いものになるので，本番環境などに適用してはなりません．あくまで勉強用あるいは解析用に使うPostgreSQLであると理解してください．

デバッガの使用例

　実例を示します．

```
SELECT 1;
```

という非常に単純なSELECT文で，SELECT文を実行するバックエンドの関数であるExecSelectという関数で停止し，そこに至るまでにどのような関数が呼ばれているか調べてみましょう．

　まずPostgreSQLのスーパユーザでログインします．なお，筆者の環境ではPostgreSQLをインストールしたユーザがt-ishiiになっていますが，通常はpostgresユーザなどを使うと思いますので，適当に読み替えてください．

　次に，psqlでデータベースに接続します．その状態でpsコマンドで見ると，図4のようなプロセスが見つかると思います．これがバックエンドプロセスです．他にもいろいろなユーザがPostgreSQLに接続しているとこのようなプロセスがたくさん表示されてわかりにくいですし，そういう意味でも実験用の環境を用意したほうが良いと思います．

　PostgreSQLをインストールしたディレクトリの，src/backendディレクトリに移ります．

```
$ cd /usr/local/src/pgsql/postgresql-8.0.1/src/backend
```

　ここでgdbを起動し，psで表示されたプロセス番号のプロセスにアタッチします（図5）．(gdb)はgdbのプロンプトで，gdbコマンドを受け付けるようになっているので，ExecSelectが呼ばれたら停止するようにbコマンドを入力します（図6）．

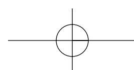
▼図4　バックエンドプロセス

```
$ ps x
 1679 ?        S      0:00 postgres: t-ishii test [local] idle
```

▼図5　gdbでプロセスにアタッチ

```
$ gdb ./postgres 1679
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
/usr/local/src/pgsql/8.0.1/postgresql-8.0.1/src/backend/1679: No such file or directory.
Attaching to program: /usr/local/src/pgsql/8.0.1/postgresql-8.0.1/src/backend/./postgres, process 1679
Reading symbols from /usr/lib/libz.so.1...done.
[中略]
Reading symbols from /lib/libnss_dns.so.2...done.
Loaded symbols for /lib/libnss_dns.so.2
0x401b2442 in __libc_recv () from /lib/libc.so.6
(gdb)
```

注4）残念ながらWindowsではこのとおりにはいかないと思いますが，筆者はWindowsには詳しくないので，説明は省略します．



PostgreSQL研第1回　06.2.27　10:35 AM　ページ218

▼図6　bコマンド

```
(gdb) b ExecSelect
Breakpoint 1 at 0x814162b: file execMain.c, line 1285.
(gdb)
```

psqlを起動した端末から「SELECT 1;」を入力し，バックエンドに実行を依頼します．しかし，このままではpostgresプロセスが停止したままなので，psqlは固まっているはずです．実行を継続するcコマンドをgdbから入力します（図7）．すると，ExecSelectで停止します．

ExecSelectまでに至る関数呼び出しの道筋は，btコマンドで表示できます（図8）．この見方ですが，下のほうが呼び出し元で，上が呼び出されるほうになっています．つまり，ExecSelectを呼び出したのはExecutePlanであり，ExecutePlanを呼び出したのはExecutorRunで…という風になっています．とくに，真ん中辺りの#5の行では，

```
exec_simple_query (query_string=0x83af680
"SELECT 1;"  （実際は1行）
```

のように表示され，いかにもSELECT文を処理しているな，という感じがします:-) gdbの出力もじっくり見てみると，このように発見があります．

gdbはソースデバッガですから，ソースコードとの対応をすぐに見られるようになっています．たとえばlistコマンドで現在実行中の行の付近を見ることができます（図9）．

上位の関数への移動は，upコマンドです．図10のように，listコマンドで実際にExecSelectを呼び出していることが確認できます．逆に下位の関数への移動はdownです．upとdownを組み合わせて，関数の呼び出し関係を調べることができます．

最後に，gdbの終了はquitです（図11）．なお，ここでgdbは終了しますが，バックエンドプロセスは終了しません．

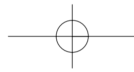
▼図7　cコマンド

```
Continuing.

Breakpoint 1, ExecSelect (slot=0x83bbfe0, dest=0x83afd50, estate=0x83bbf40) at execMain.c:1285
1285        tuple = slot->val;
(gdb)
```

▼図8　btコマンド

```
(gdb) bt
#0  ExecSelect (slot=0x83bbfe0, dest=0x83afd50, estate=0x83bbf40) at execMain.c:1285
#1  0x081414e4 in ExecutePlan (estate=0x83bbf40, planstate=0x83bc028, operation=CMD_SELECT,
    numberTuples=0, direction=ForwardScanDirection, dest=0x83afd50) at execMain.c:1200
#2  0x081402b3 in ExecutorRun (queryDesc=0x83bbb60, direction=ForwardScanDirection, count=0)
    at execMain.c:226
#3  0x081e2ce0 in PortalRunSelect (portal=0x83b9b30, forward=1 '\001', count=0, dest=0x83afd50)
    at pquery.c:746
#4  0x081e2973 in PortalRun (portal=0x83b9b30, count=2147483647, dest=0x83afd50, altdest=0x83afd50,
    completionTag=0xbfffeac0 "") at pquery.c:561
#5  0x081de6a6 in exec_simple_query (query_string=0x83af680 "SELECT 1;") at postgres.c:933
#6   0x081e1505  in  PostgresMain  (argc=4,  argv=0x836a2f0,  username=0x836a2d0  "t-ishii")  at
postgres.c:3007
#7  0x081aaab7 in BackendRun (port=0x8379ba0) at postmaster.c:2816
#8  0x081aa0d0 in BackendStartup (port=0x8379ba0) at postmaster.c:2452
#9  0x081a8043 in ServerLoop () at postmaster.c:1199
#10 0x081a790e in PostmasterMain (argc=3, argv=0x83689e8) at postmaster.c:918
#11 0x081654b7 in main (argc=3, argv=0x83689e8) at main.c:268
#12 0x400f213f in __libc_start_main (main=0x81652d0 <main>, argc=3, ubp_av=0xbffff774,
    init=0x8073e10 <_init>, fini=0x82939c0 <_fini>, rtld_fini=0x4000c770 <_dl_fini>,
    stack_end=0xbffff76c) at ../sysdeps/generic/libc-start.c:129
(gdb)
```



▼図9　listコマンド

```
(gdb) list
1280    TupleDesc     attrtype;
1281
1282    /*
1283     * get the heap tuple out of the tuple table slot
1284     */
1285    tuple = slot->val;
1286    attrtype = slot->ttc_tupleDescriptor;
1287
1288    /*
1289     * insert the tuple into the "into relation"
```

▼図10　upコマンド

```
(gdb) up
#1  0x081414e4 in ExecutePlan (estate=0x83bbf40, planstate=0x83bc028, operation=CMD_SELECT,
    numberTuples=0, direction=ForwardScanDirection, dest=0x83afd50) at execMain.c:1200
1200                ExecSelect(slot,    /* slot containing tuple */
(gdb) list
1195         * delete it from a relation, or modify some of its attributes.
1196         */
1197        switch (operation)
1198        {
1199            case CMD_SELECT:
1200                ExecSelect(slot,    /* slot containing tuple */
1201                           dest,    /* destination's tuple-receiver obj */
1202                           estate);
1203                result = slot;
1204                break;
```

▼図11　quitコマンド

```
(gdb) quit
The program is running.  Quit anyway (and detach it)? (y or n) y

Detaching from program: /usr/local/src/postgresql-8.0.1/src/backend/./postgres, process 1679
```

tagsを使って関数を定義した対応したファイルにジャンプ

さて，gdbを使ってPostgreSQLの動きを調べることはできましたが，さすがにgdbのlistコマンドだけでソースを追うのはつらいので，普通はEmacsなどのエディタも併用してソースを眺めつつgdbを使うことになります[注5]．

このときに，たとえば上に出てくる「exec_simple_query」の定義を見たければ，Emacsのtagsコマンドを使って即座にその関数を定義している個所にジャンプできます．tagsを使うためにはtagsファイルを作成する必要がありますが，PostgreSQLにはtagsファイルを作成するスクリプトが付属しています[注6]．

```
$ cd /usr/local/src/postgresql-8.0.1/src
$ tools/make_etags
```

これでOKです．後はEmacsの中で，「ESC-.」(ESCキーの後にピリオドを入力）でexec_simple_queryと叩くか，exec_simple_queryという文字列があるところにカーソルを持っていってESC-.でexec_simple_queryの定義されているソースファイルを開くことができます．

さいごに

連載1回目ではPostgreSQLの構造とソースのコードの概要と，それにソースコードを見るための手順をご紹介しました．次回はいよいよバックエンド内部の処理の流れを見ていきます．お楽しみに[注7]．

注5）gdbモードを使うともっと便利なのかもしれませんが，筆者は食わず嫌いで使っていません．
注6）vi派の方は，tools/make_tagsをお使いください．
注7）予習をしたい方は．PostgreSQL付属ドキュメントの「第40章 PostgreSQL内部の概要」に目を通すと良いでしょう．