実践テクニックをご紹介

徒然PostgreSQL散策

第2回 PostgreSQL夏休み工作教室 ～SQLをカスタマイズしよう

日本PostgreSQLユーザ会 理事長　石井 達夫
ISHII Tatsuo●ishii@postgresql.jp

はじめに

今年も暑い夏がやってきましたが，みなさんお元気におすごしでしょうか．日本の夏は執筆作業にとっては非常につらい季節で，冷房をかけすぎると健康に悪いし，かといって冷房なしというのも汗のために原稿用紙が湿ってしまうなど不都合がおきます[注1]．

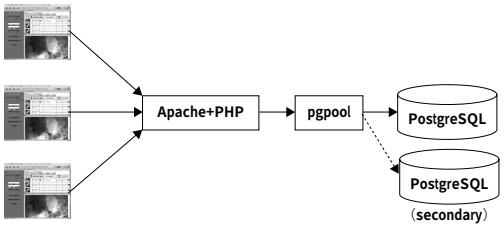
そこで最近ついに無線LANを導入し，家の中で涼しいところならどこでも執筆できる環境を整えました．これが思いのほか快適で，調子に乗ってバスパワーで動作するUSBの外付ディスクまで買ってしまいました．これで「家中どこでも書斎」環境の完成です:-)

コネクションプール再び

今回は「PostgreSQLのSQLをカスタマイズしよう」と題し，ソースコードをハックしてPostgreSQLを改良してしまおうという企画ですが，その前に前号のフォローを少し．

前回はPHPとコネクションプールの関係を取り上げ，大規模なWebシステムをPHPで構築しようとするときPHPにコネクションプール機能がないことがネックになることを説明しました．そのときに「まともなコネクションプールシステムを作ることが必要だが，すぐには実現できないのでDB処理を軽くしましょう」というまことに不完全燃焼な結論を出してしまったことを反省し，罪滅ぼしにコネクションプールシステムを作ってみたのでご紹介します．

図1◎pgpoolを使用した場合のシステム構成



コネクションプールサーバpgpool

今回開発したのは「pgpool」というシンプルなコネクションプールサーバです．pgpoolはPostgreSQLのクライアントとPostgreSQLサーバの間に割り込む形で使用します（図1）．pgpoolはクライアントから見るとPostgreSQLサーバに見え，同時にPostgreSQLサーバから見るとPostgreSQLのクライアントに見えるようになっており，いわばPostgreSQLのproxyサーバとして動作します．

PostgreSQLのアプリケーションからpgpoolを利用するためには，接続ポート番号を9999に変えるだけでよく，ポート番号の変更以外にはアプリケーションプログラムの変更は必要ありません．PostgreSQLサーバのほうは変更は一切いりません[注2]．また，図1ではPHPを使っていますが，もちろんPerlでもJavaでも使えます．

pgpoolの動作検証

pgpoolはPostgreSQLへのコネクションをキャッシ

注1）もちろん嘘です:-) でも，たまには万年筆で400字詰原稿用紙に向かうのもいいかも……．
注2）ポート番号は設定によって9999以外に変更できます．なお，PostgreSQLの認証を使用している場合はpg_hba.confの設定変更が必要になるかもしれません．

ュするので，その分効率が良くなります．また，PostgreSQLへの同時接続数を一定の数に制限し，PostgreSQLの性能を最大限に発揮できるメリットもあります．反面，通信を中継するという余計な処理が入るデメリットがあります．はたしてデメリットを上回るメリットがあるのか，検証してみることにします．

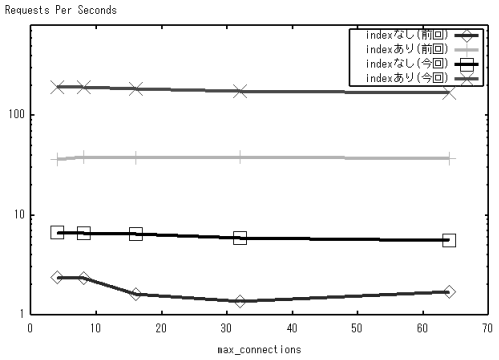
pgpoolを使って前回と同じベンチマークを行った結果が図2のグラフです（ただし，スクリプトのほうはリトライ処理が不要になるので，**リスト1**のようにシンプルになっています）．pgpoolを使用した場合の同時コネクション数は，pgpoolからPostgreSQLへの同時接続数を意味します．

ご覧のように，インデックスありの場合で約4倍から5倍，インデックスなしの場合で1.5倍から4倍の性能向上が見られました．どうやらpgpoolによって前号から持ち越した課題は解決できたようです．

フェールオーバ機能

なお，pgpoolにはフェールオーバといって，PostgreSQLサーバがダウンしたときに自動的にあらかじめ用意しておいた2台目のPostgreSQLサーバ（図1で「secondary」と表記）に切り替える機能があります．フェールオーバにより，システムのダウンタイムを最小限に留めることができる，予備機を用意してのデータベースのメインテナンスが容易になるなどの利点があります[注3]．

図2 ◎ベンチマーク結果



◇ ◇ ◇

pgpoolはftp://ftp.sra.co.jp/pub/cmd/postgres/pgpool/から入手できます．

PostgreSQL夏休み工作教室

本誌が発売される頃には，学生の読者の皆さんの夏休みも終わりに近づいていることと思います（本誌の読者に学生さんがどのくらいいらっしゃるかどうかは知らないのですが）．夏休みといえば，宿題．宿題と言えば工作．というわけで少々強引ですが，今回は「PostgreSQL夏休み工作教室」と題して，PostgreSQLのソースをハックする方法を伝授したいと思います．

PostgreSQLハッカーへの道

PostgreSQLはソースコードが公開されているので，ユーザが自由にソースコードをいじることができます．アプリケーションレベルでは散々苦労しないと実装できない機能が，PostgreSQLのソースコードにちょっと手を加えるだけで実現できることも珍しくありません．つまり，最小限の投資で大きな利益が得られる可能性があるのが，オープンソースソフトウェアの魅力の1つではないかと思います[注4]．

こういう風に言うと，気の早い読者はすぐにでもPostgreSQLのソースコードを入手して解読にかかるかもしれませんが，そこであなたの前に立ちふさがるのは膨大な量のソースコードです．Cのソースコード

▼リスト1　今回使ったPHPスクリプト

```
<?php
 ini_set("track_errors", "1");
 define_syslog_variables();

 $con = pg_connect("dbname=test user=t-ishii port=9999");
 if ($con == FALSE) {
   syslog(LOG_ERR, "could not connect $php_errormsg");
   trigger_error("Could not connect to DB", E_USER_ERROR);
   exit;
 }
 $aid = rand(1,10000);
 pg_query($con, "SELECT * FROM accounts WHERE aid = $aid");
 pg_close($con);
?>
```

注3）ただしpgpoolには2台のデータベースの内容を同期させる機能はありませんので，必要ならばdbmirrorなどのレプリケーションソフトを併用してください．

注4）もちろん単なる知的好奇心からソースコードをハックする方もいらっしゃるでしょう．こちらのほうがどちらかというと動機が純粋で「夏休み工作教室」にふさわしいかもしれませんね:-)

だけで500本以上，ステップ数にして40万行以上あります．これにいきなり立ち向かうのは，予備知識なしで富士の樹海に迷い込むようなもの．遭難する確率大です．そこで今回は，筆者が案内人を務めさせていただきます．

PostgreSQLの内部構造に関する「公式ガイドブック」

PostgreSQLの内部構造に関する「公式ガイドブック」としては，PostgreSQLに付属するドキュメントの「開発者ガイド」[注5]があります．

このドキュメントの2章「PostgreSQL内部の概要」を見ると，PostgreSQLに送られた問い合わせがどのように処理されるのかの概略がわかります．ただ，惜しいことにこの章ではすべての図が抜けてしまっています．特にパースツリー（SQL文をコンピュータ処理しやすいデータ構造に変換したもの）などの図が抜けているのは非常に残念です[注6]．

それともう1つ，このドキュメントに決定的に欠けているのは，PostgreSQLのストレジやトランザクション処理に関する説明です．また，実際にソースコードのどこに処理が書かれているのか，ということもわかりません．

実践的にソースコードを学習する

というわけで，公式ガイドブックだけではソースコードの解析は難しそうです．そこで本稿では「理論よりも実践」ということで，実際に例題に沿ってソースコードをハックしながら学んでいくことにしたいと思います．

繰り返しになりますが，なにせPostgreSQLのソースコードは巨大ですので，正攻法ではなかなか歯が立ちません．やみくもにソースコードを追うのではなく，筆者が示した道筋にしたがって必要最小限の個所に絞りこんで学習していただこうという趣向です．

お題はSQL文のカスタマイズ

今回の例題はSQL文のカスタマイズです．いかに拡張性の高いPostgreSQLといえども，ソースコードをハックせずにSQL文を変更するのは無理です．また，他の方法ではこの目標は達成できないという意味で，PostgreSQL使いの経験値を上げるよいチャンスであるとも言えるのではないでしょうか．

ただし，SQLの文法を少し変える程度では実用性に乏しいので，ついでに何かPostgreSQLにちょっとした有用な機能を追加することにしましょう．

ロックによる排他制御

PostgreSQLには，テーブルの排他制御のための「LOCK」というSQL文があります．

簡単にLOCKの使い方を説明しましょう．たとえば，売り上げを記録するためのテーブルsalesがあったとします．

```
CREATE TABLE sales (
  sales_id INTEGER PRIMARY KEY,  -- 売り上げ番号
  goods_id TEXT,                 -- 商品番号
  num INTEGER,                   -- 販売数
  sales_person TEXT,             -- 担当者名
  sales_date DATE                -- 売上日
);
```

ここで売り上げ番号は，連番でなければならないとします．

連番というと，シーケンスやSERIAL型がすぐに思い浮かびます．これらは値が重複しないことは保障しますが，トランザクションがロールバックしたときに番号が中飛びしないことは保障しないのでここでは使えません．

そこで現在テーブルの中にある最大の売り上げ番号

注5）日本PostgreSQLユーザ会のサイトに日本語訳が掲載されています．開発者ガイドのURLは，http://www.postgresql.jp/document/pg732doc/developer/index.htmlです．

注6）このあたりの背景については歴史的な事情があるようです．もともとこのドキュメントは，相当前のバージョンのPostgreSQL用に書かれたものです．ですから，現在のバージョンのPostgreSQLの説明にはどうせそんな古い図は使えない，というのも1つの理由のようです．筆者としては，古くてもよいから図を掲載してほしいものだと思うのですが……．

に1を加えたものを計算して，連番を作ることにします．つまりセールス担当者は売り上げを記録するときに，

```
SELECT max(sales_id) FROM sales;
```

を実行して現在の最大のsales_idを得て，そこに＋1して新しいsales_idを作るわけです．

さて，ここでたまたま同時に2人のセールス担当者が売り上げを記録しようとしたらどうなるでしょう？当然重複した売り上げ番号が登録されてしまいます[注7]．そこで使われるのがLOCKです．LOCKを使用することにより，2人以上のセールス担当者が同時にsalesテーブルを更新する事態を防ぐことができます．

psqlを2つ立ち上げて実際に試してみましょう．図3で，2つpsqlをそれぞれT1（セールス担当者Aが操作），T2（セールス担当者Bが操作）で表し，時系列に沿って見ていきます．なお，sales_idの現在の最大値は100であるとします．

待ち合わせを回避する方法

これで一応問題は解決ですが，LOCKの難点は，すでにテーブルがロック済の場合，そのロックが解除されるまで待たされることです．セールス担当者Aがロックしたままお昼を食べに行ってしまったら，運の悪いセールス担当者Bは彼が帰ってくるまで待ちぼうけです．

この問題に対しては，以下のような対処方法が考えられます．

① statement_timeout機能を使う．postgresql.confでstatement_timeoutを設定することにより，一定時間後ブロックされたLOCKが解除されるので，待ちぼうけになることはなくなります．

②「salesテーブル使用中」フラグを設ける．salesテーブルが使用中だったら別のテーブルなどにフラグを立てます．もしこのフラグが立っていたら，セールス担当者BはAが作業中だと判断できます．

③ とりあえず新しい売り上げデータを書き込んでみて，もしもsales_idが重複していたらもう一度やり直す．

①の方法では，すべてのSQL文にタイムアウトが設定されるので，非常に実行時間の長いSELECTまでタイムアウトしてしまうことがあります．②ではアプリケーションプログラムが複雑になりますし，もしフラグを落とすのを誰かが忘れると，そのあと誰もsalesテーブルにアクセスできなくなってしまうという問題があります．③は失敗すると操作のやり直しが必要になるので，利用者から不興を買ってしまうかもしれません．

というわけで，どの方法もパーフェクトではありません．

待ち合わせなしのロック

そこで，今回実装するのが「待ち合わせなしのロック」（と勝手に筆者が名付けました）です．普通のロックとの違いは，ロックが獲得できなかったときに待ち続けるのではなく，すぐにエラーが返ってくることです．Bさんは「ああ誰かが作業中なんだ」とわかるのでそのあいだ昼食を取るとか，時間を有効に使うことができます．先の②と似ていますが，アプリケーションの面倒がありませんし，フラグを落とし忘れる心配もないところが違います．

▼図3　2つのトランザクションとロック

```
T1: BEGIN;                                  -- トランザクションの開始
T2: BEGIN;                                  -- トランザクションの開始
T1: LOCK TABLE sales;                       -- salesテーブルをロックする
T2: LOCK TABLE sales;                       -- salesテーブルをロックしようとするが，すでにT1がロックしているので待たされる
T1: SELECT max(sales_id) FROM sales;        -- 100が返る
T1: INSERT INTO t1 VALUES(101, ....);       -- 100+1である101を登録
T1: END;                                    -- トランザクションを終了
T2: T2: SELECT max(sales_id) FROM sales; -- salesのロックが外れたので続行できる，101が返る
:
:
```

注7）実際には主キーが設定されているので値重複エラーになりますが，好ましいことでないことには変わりはありません．

アイデアは非常に簡単で，普通のLOCK文にオプションで「NO WAIT」を付けるだけです．実行例を示します．

```
test=# LOCK TABLE sales NO WAIT;
ERROR:  Cannot aquire relation lock
```

もちろん「NO WAIT」を付けなければ普通のLOCK文と同じ動作をします．

いよいよ探検開始

やりたいことが決まったので，いよいよPostgreSQLのソースコードの探検に出かけましょう．

まずhttp://www.postgresql.jp/などからPostgreSQL 7.3.3のソースコードを入手し，適当なディレクトリに展開してください．ここではpostgresユーザのホームディレクトリの下にsrcというディレクトリを作り，その下に展開するものとします．ディスクの空き容量は300Mバイト以上確保してください．

PostgreSQLのソースコードのtar ballは/tmpに置いてあるものとします．また，以下はVine Linux 2.6での実行例ですが，他のプラットフォームでもほとんどそのまま適用できると思います．

まず，まだpostgresユーザがなければそれを作ります（rootで実行します）．

```
# useradd postgres
```

postgresユーザになります．

```
# su - postgres
```

以下，postgresユーザで実行します．

```
$ cd
$ mkdir src
$ tar xfz /tmp/postgresql-7.3.3.tar.gz
```

▼図4　postgresql-7.3.3ディレクトリ

```
$ cd postgresql-7.3.3
$ ls
COPYRIGHT       INSTALL     config/         configure.in  src/
GNUmakefile     Makefile    config.log      contrib/
GNUmakefile.in  README      config.status*  doc/
HISTORY         aclocal.m4  configure*      register.txt
```

まずはトップディレクトリを探検

ここまででpostgresql-7.3.3というディレクトリができているはずなので，まずはソースコードのトップディレクトリを探検してみましょう（図4）．

COPYRIGHT，HISTORY，INSTALL，README，register.txtはドキュメントなので，直接プログラムには関係しません．docはSGMLというフォーマットで管理されているドキュメント類などが格納されています．

srcはPostgreSQL本体のソースコードが格納されているディレクトリです（あとで解説します）．

contribは標準ではインストールされない「おまけ」のプログラム類です．これらは必要性や実績が認められると，本体の一部に「昇格」することもあります．

いま説明した以外のファイルは，configureというシステムで使用されるファイル群です．今回は誌面の都合で詳しくは解説できませんが，configureはプラットフォーム間で移植性を高めるために使用されるツールです．configureを実行するとプラットフォーム固有の情報が検出され，srcなどのサブディレクトリにあるMakefileに埋め込まれます．

ソースコードディレクトリ src

では，次にソースコードディレクトリであるsrcに行ってみましょう．その前に，configureを実行し，必要なファイルを生成しておきます．configureのオプションはいろいろありますが，今回は以下のように実行します．

```
$ ./configure --enable-debug --enable-cassert
```

これらはおもに開発時に実行するオプションなので見慣れないものかもしれませんが，デバッグのときに非常に役に立ちます．--enable-debugを付けるとgdbなどのデバッガを使ってソースコードレベルのデバッグができるようになります．--enable-cassertは実行時にプログラムの整合性をチェックします．実行速度は遅くなりますが，プログラムの問題点を早期に発見できるようになります．

configureが終わったら，一応手を加える前の

PostgreSQLが正常に動作するかどうか確認します．

```
$ make check
```

これでコンパイルの後regression test（回帰テスト）が実行されます．全部OKか，randomというテストだけがfailなら問題ありません．PostgreSQLをハックする際は，節目節目でregresion testを実施してエンバグ（バグを入れ込んで品質が低下すること）がないようにチェックすると良いでしょう．

PostgreSQLが正常ならばsrcに移動します．

```
$ cd src
```

さてsrc以下を眺めてみると，表1のようなファイルとディレクトリがあります[注8]．今回はSQL文に手を加えますが，SQL文を処理する機能のためのソースはバックエンド（データベースエンジン）に含まれていますので，backendディレクトリを調べればよいことになります[注9]．

ついでにTAGSも作っておきましょう[注10]．

```
$ tools/make_etags
```

backendディレクトリの中身

backendには次ページ表2のようなファイルとディレクトリがあります[注11]．

こうしてみるとディレクトリの数だけで結構あるのでめげてしまいそうですが，幸い今回はディレクトリにして4つ触るだけですので，くじけずに先に進みましょう:-)

パーサの構造

今回はSQL文をいじりますので，まずはSQLパーサを修正します．パーサ関係のソースは，すべてsrc/backend/parserの下にあります．SQLパーサの役目は，文字列で書かれたSQL文を解析し，パースツリーというメモリ上の構造を生成することです．

PostgreSQLでは，パーサはBisonという構文解析ツールとFlexという字句解析ツールを使って書かれており，それぞれgram.yとscan.lというファイルに格納されています．gram.yからgram.cとparse.h，scan.lからはscan.cが生成されます．

PostgreSQLのソースにはこれら生成済のファイルが含まれているので，gccだけでコンパイルできますが，PostgreSQLを開発するときはBisonとFlexも必要です．今回はscan.lはいじりませんので，Bisonだ

表1◎srcディレクトリの中身

●ディレクトリ／ファイル	●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の雛型
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	共有ライブラリ生成用のmake設定値
backend/	バックエンドのソース一式
bin/	psqlなどのUNIXコマンドのソース
corba/	CORBA対応の試み（未完成）
data/	キリル文字用のデータ
include/	ヘッダファイル
interfaces/	フロントエンドのソース一式
makefiles/	プラットフォーム依存のmake設定値
nls-global.mk	各国語対応用のmake設定値
pl/	プロシージャ言語（C，SQL以外の言語で関数を定義する）
template/	プラットフォーム依存の設定値
test/	各種テストツール
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

注8）この表は拙著『改訂第3版・PostgreSQL完全攻略ガイド』（技術評論社，ISBN4-7741-1226-7）より引用，7.3用に加筆訂正しています．

注9）実際にはヘッダファイルであるincludeも関係しますが，これについてはあとで触れます．

注10）TAGSとは，Emacsなどのエディタの中から関数などの定義にジャンプできる便利な機能で使用するためのファイルのことです．

注11）この表も拙著『改訂第3版・PostgreSQL完全攻略ガイド』から引用，7.3用に加筆訂正しています．

表2◎backendディレクトリの中身

●ディレクトリ／ファイル	●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/	ハッシュインデックス
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インデックス
rtree/	Rtreeインデックス
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/	mainプログラム
nls.mk	各国語対応用のmake設定値
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
po/	メッセージカタログ
postmaster/	postmaster．main( )関数はここに定義されている
regex/	正規表現処理
rewrite/	rule/view
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
buffer	バッファ管理
file	低レベルファイルIO
freespcace	未使用領域管理
ipc	プロセス間通信
large_object	ラージオブジェクト
lmgr	ロックマネージャ
page	バッファ上のページ管理
smgr	ストレジ管理
tcop/	postgresのメインループ
utils/	さまざまなモジュール
Gen_fmgrtab.sh	関数管理用のファイル生成ツール
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（elog( )など）
fmgr/	関数管理
hash/	ハッシュ関数
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/	palloc( )などのメモリ管理関数
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

け用意してください[注12]．

gram.yの修正

gram.yは前述のようにBisonで書かれたファイルで，しかも8000行近くある大きなファイルですが，当初の方針通りポイントだけ説明します[注13]．

LOCKの構文解析

「LOCK」でgram.yをgrepすると，4053行目にリスト2のような部分が見つかります．これがLOCK文の定義です．この部分はこう読めます．第1行目は「LockStmtはLOCK_P opt_table qualified_name_list opt_lockからなる」という意味です．入力されたSQL文がこの定義に合致すればパーサはLOCK文であると認識します．

LOCK_P以降は，LOCK文の定義：

```
LOCK [ TABLE ] name [, ...] [IN lockmode MODE]
```

と比較すると表3のような対応になります．ちなみに大文字の単語は「トークン」，小文字はパースの途中結果である「シンボル」を表します．

注12）Bisonのバージョン1.875以降が必要です．プラットフォームによっては古いBisonしか付属していないこともありますので，ソースから入れるか（もしあれば）最新のパッケージを入れてください．
注13）Bisonそのものの使い方については，『lex&yaccプログラミング』（アスキー，ISBN4-7561-0297-2）などが参考になります．

▼リスト2　LOCK文の定義

```
LockStmt:    LOCK_P opt_table qualified_name_list opt_lock
               {
                   LockStmt *n = makeNode(LockStmt);   ← LOCK文に対応するノード構造体LockStmtを作成

                   n->relations = $3;                  ← テーブル名（qualified_name_list）をセット
                   n->mode = $4;                       ← ロックモード（opt_lock）をセット
                   $$ = (Node *)n;                     ← パースツリーにLockStmtノード構造体をセット
               }
       ;
```

解析の結果SQL文がLOCK文であると認識されると，中括弧の中が実行され，パースツリーが作成されます．なお，$3とか$4は，LOCK_P以下の何番目の項目であるかを表します．

LOCK構文の変更

今回は構文を以下のように変更します．

```
LOCK [ TABLE ] name [, ...] [IN lockmode MODE]
[NO WAIT]　（実際は1行）
```

パーサの定義にはNO WAITの追加に対応して新しいシンボルopt_no_waitを加えることにします．

```
LockStmt:    LOCK_P opt_table qualified_name_
list opt_lock opt_no_wait　（実際は1行）
```

中括弧の中には，NO WAITが指定されたかどうかを設定する処理を追加します．

```
LockStmt *n = makeNode(LockStmt);
n->relations = $3;
n->mode = $4;
n->no_wait = $5;   ← 今回追加
$$ = (Node *)n;
```

次にopt_no_waitを実際にどう処理するかを記述します．4063行目あたり（リスト3）のあとにリスト4を追加します．「NO WAIT」が入力されると「$$ = TRUE」によってopt_no_waitはTRUEになります．そうでなければFALSEになります．

続いて，ここで使用した「トークン」である「NO」と「WAIT」を追加します．NOのほうはすでに登録されているので，WAITのみ393行目あたり

```
VERBOSE VERSION VIEW VOLATILE
```

のあとに，

```
WAIT
```

を追加します．

最後に新しく追加したシンボルであるopt_no_waitのデータ型を指定します．データ型はgram.yの99行目あたりから書いてある

```
%union
{
    int             ival;
    char            chr;
    char            *str;
    const char      *keyword;
    bool            boolean;
    :
    :
```

表3◎LOCK文との対応

●定義	●構文	●意味
LOCK_P	LOCK	“LOCK”という文字列
opt_table	[TABLE]	“TABLE”という文字列（省略可能）
qualified_name_list	name [, ...]	テーブル名（複数可能）
opt_lock	[IN lockmode MODE]	ロックモード（省略可能）

▼リスト3　追加する場所

```
opt_lock:    IN_P lock_type MODE          { $$ = $2; }
           | /*EMPTY*/                    { $$ = AccessExclusiveLock; }
       ;
```

▼リスト4　追加分

```
opt_no_wait:    NO WAIT                   { $$ = TRUE; }
              | /*EMPTY*/                 { $$ = FALSE; }
          ;
```

の中から選択します．今回はbooleanを選び，163行目あたりの

```
%type <boolean>     opt_force opt_or_replace
```

のあとに，

```
%type <boolean>     opt_no_wait
```

を追加します．

キーワードの追加

以上でgram.yの修正は終わりですが，パーサの修正はこれで終わりではありません．同じディレクトリにkeywords.cというファイルがあり，ここに追加したトークンであるWAITを新しいキーワードとして足しこむ必要があります．

keywords.cの31行目あたりにScanKeywordsというデータ構造があります．

```
static const ScanKeyword ScanKeywords[] = {
    /* name, value */
    {"abort", ABORT_TRANS},
    {"absolute", ABSOLUTE},
    :
    :
```

ここにWAITを追加しますが，このリストはアルファベット順に並んでいなければなりません．具体的には，320行目

```
{"volatile", VOLATILE},
```

のあとに

```
{"wait", WAIT},
```

を追加します．

LockStmtの修正

以上でパーサ自体の修正は終わりですが，LockStmt構造体の修正がまだでしたね．LockStmtがどこで定義されているかは，すでに作成したTAGSファイルを使って簡単に調べることができます．Emacsでgram.yを開き，LockStmtという文字列にカーソルを合わせ，ESC-.（エスケープキーを押してからピリオド“.”を押す）を入力します．すると，src/include/nodes/parsenodes.hが開かれるはずです．

このファイルには，パースツリーで生成されるデータ構造（PostgreSQLでは「ノード」と呼びます）などが定義されています．1591行目あたりにLockStmtの定義があるので，リスト5のように変更します．

▼リスト5　LockStmtの変更部分

```
typedef struct LockStmt
{
    NodeTag     type;
    List        *relations;     /* relations to lock */
    int         mode;           /* lock mode */
    bool        no_wait;        /* no wait mode */ ⬅ これを追加
} LockStmt;
```

パーサをコンパイルしてみる

以上でパーサ関係の修正がすべて終わったので，パーサのみコンパイルしてみましょう．図5[注14]のように

▼図5　パーサのコンパイル

```
$ cd ~/src/postgresql-7.3.3/src/backend/parser
$ make
bison -y -d  gram.y
sed -e 's/"syntax error/"parse error/' < y.tab.c > ./gram.c
mv -f y.tab.h ./parse.h
rm -f y.tab.c
gcc -O2 -g -Wall -Wmissing-prototypes -Wmissing-declarations -I. -I../../../src/include   -c -o gram.o
gram.c
gcc -O2 -g -Wall -Wmissing-prototypes -Wmissing-declarations -I. -I../../../src/include    -c -o
keywords.o keywords.c
[中略]
/usr/bin/ld -r -o SUBSYS.o analyze.o gram.o keywords.o parser.o parse_agg.o parse_clause.o
parse_expr.o parse_func.o parse_node.o parse_oper.o parse_relation.o parse_type.o parse_coerce.o
parse_target.o scansup.o
```

注14）最後の行は「パーシャルリンク」と呼ばれるものです．PostgreSQLは巨大なシステムでオブジェクトファイルの数も多いため，一度にオブジェクトをリンクしてロードモジュールを生成するのが困難です．そこでサブシステムごとにオブジェクトファイルをまとめた中間的なオブジェクトファイルであるSUBSYS.oを生成しておき，リンク時にはサブシステムの数だけSUBSYS.oをリンクすればよいようにしているのです．

なれば成功です．

問い合わせ実行部の作成

当然のことながら，問い合わせをパースツリーに変換しただけでは何も起きません．パースツリーに対応した問い合わせ実行部が必要になります．この実行部はエグゼキュータ（executor）と呼ばれます．PostgreSQLではエグゼキュータは2つの部分に分かれています．

① SELECT/INSERT/UPDATE/DELETEを処理する部分
② それ以外（「ユーティリティ文（utility statement）」と呼ばれます）を処理する部分

今回説明しているLOCK文は②の分類に入ります．こちらは①に比べるとかなり処理が簡単[注15]で，パースツリーが作成された後は基本的にsrc/backend/commandsの下にある関数が呼ばれて処理されるだけです．

LOCK文の実行部

LOCK文の実行部はsrc/backend/commands/lockcmds.cにあります．これは70行ほどのシンプルなファイルで，LockTableCommand()という関数が1つ定義されているだけです．LockTableCommand()はProcessUtility()という関数から呼び出されます．

ではLockTableCommand()の中身を追ってみましょう（リスト6）．

foreachでリスト処理

LockTableCommand()全体は38行目からforeachのループで構成されています．foreachはCにはない構文のように見えますが，実は単なるマクロで，38行目は次のように展開されます．

```
for(p = lockstmt->relations; p != NIL; p = lnext(p))
```

ここではPostgreSQLで頻繁に出てくるList型（その名のとおりリスト構造を表します）の探索処理を行います．興味のある方はList型の定義ファイルであるsrc/include/nodes/pg_list.hを覗いてみるとよいでしょう．

RangeVar構造体

このループではLOCKに引数として与えられたテーブルの情報を1個ずつ処理していきます．テーブルの

▼リスト6　LOCK文の実行部LockTableCommand()のソース（一部）

```
28:  void
29:  LockTableCommand(LockStmt *lockstmt)
30:  {
31:      List       *p;
32:
33:      /*
34:       * Iterate over the list and open, lock, and close the relations one
35:       * at a time
36:       */
37:
38:      foreach(p, lockstmt->relations)
39:      {
40:          RangeVar   *relation = lfirst(p);
41:          Oid            reloid;
42:          AclResult    aclresult;
43:          Relation    rel;
44:
45:          /*
46:           * We don't want to open the relation until we've checked
47:           * privilege. So, manually get the relation OID.
48:           */
49:          reloid = RangeVarGetRelid(relation, false);
50:
51:          if (lockstmt->mode == AccessShareLock)
52:              aclresult = pg_class_aclcheck(reloid, GetUserId(),
53:                                            ACL_SELECT);
54:          else
55:              aclresult = pg_class_aclcheck(reloid, GetUserId(),
56:                                            ACL_UPDATE | ACL_DELETE);
57:
58:          if (aclresult != ACLCHECK_OK)
59:              aclcheck_error(aclresult, get_rel_name(reloid));
60:
61:          rel = relation_open(reloid, lockstmt->mode);
62:
63:          /* Currently, we only allow plain tables to be locked */
64:          if (rel->rd_rel->relkind != RELKIND_RELATION)
65:              elog(ERROR, "LOCK TABLE: %s is not a table",
66:                   relation->relname);
67:
68:          relation_close(rel, NoLock);    /* close rel, keep lock */
69:      }
70:  }
```

注15）①は問い合わせの最適化などの高度な機能を含む部分で，非常に複雑であり，PostgreSQLの頭脳とも言える部分です．機会があればいつか解説したいと思います．

情報はsrc/include/nodes/primonodes.hで定義されているRangeVarという構造体で与えられています（リスト7）.

この情報を使い，49行目でテーブルを一意に識別するオブジェクトID（OID）を得ます．51行目から59行目はテーブルへのアクセス権のチェックです.

実体はrelation_open( )

このファイルを見ても“lock”というような，いかにもそれ風の名前の付いた関数は呼ばれていませんが，実は61行目で呼ばれているrelation_open( )の中でロック処理を行っているのです．relation_open( )の本来の役目はテーブルを「開く」ことにあります[注16].

relation_open( )の引数はテーブルのOID（reloid）とロックモード（lockstmt->mode）です．つまりロックモードを指定してテーブルを開くことができれば，そのテーブルはロックされているという寸法です.

このように，テーブルを開く操作とロックする操作が不可分になっているのは，テーブルを開いてからロックするまでのわずかな時間（「ウィンドウ」と呼びます）にほかのトランザクションからロックされてしまう危険性があるからです.

relation_open( )を置き換える

現状では，ほかのトランザクションがテーブルのロックを獲得済の場合は，ロックを取得できずrelation_open( )から戻ってきません．つまり，relation_open( )をなんとかしないと「待ち合わせなしのロック」という今回の目的が達成できません.

そこでrelation_open( )のコード（access/heap/heapam.c）を見てみると，その中でLockRelation( )を呼んでロックを取得していることがわかります（リスト8）.

結局LockRelation( )の中でロック待ちが発生するわけです.

「待ち合わせなしロック」を実現

LockRelation( )はstorage/lmgr/lmgr.cに定義されています．このファイルを調べてみると，なんとConditionalLockRelation( )というロックが獲得できなかったときにロック待ちをせずにただちに戻る関数がすでに定義されています．いやーラッキーですね．今回はLockRelation( )の代わりにConditionalLockRelation( )を呼び出せば目的が達成できそうです.

ただ，relation_open( )を書き換えてしまうと，他の場所からrelation_open( )を呼び出している関数に影響を与えないとも限りません．そこで，新しくconditional_relation_open( )という関数を作り，relation_open( )の代わりに呼び出すことにしましょう．つまり，

LockTableCommand→relation_open→LockRelation

という流れを

LockTableCommand→conditional_relation_open→ConditionalLockRelation

という流れに変更するわけです.

conditional_relation_open( )

conditional_relation_open( )は，基本的にrelation_open( )をコピーして必要なところだけを修正します.

まずrelation_open( )は引数が2つだけでしたが，conditional_relation_open( )には第3の引数としてbool型のno_waitを追加します．そしてno_waitがTRUEだったらConditionalLockRelation( )を呼び，そうでなければLockRelation( )を呼ぶようにします．先ほどのロジック（リスト8）はリスト9のように変わります.

ここでelog(ERROR, ...)は，ConditionalLockRelation( )がロックを獲得できずに

▼リスト7　RangeVar構造体．コメントはかなり意訳された日本語

```
typedef struct RangeVar
{
    NodeTag     type;           /* 常にT_RangeVar */
    char        *catalogname;   /* データベース名またはNULL */
    char        *schemaname;    /* スキーマ名またはNULL */
    char        *relname;       /* テーブル名 */
    InhOption   inhOpt;         /* テーブル継承に関する処理フラグ */
    bool        istemp;         /* 一時テーブルあるいはシーケンスならtrue */
    Alias       *alias;         /* テーブルと列の別名 */
} RangeVar;
```

注16）relation_open( )が開くのはテーブルだけでなく，インデックスなども開くことができます．PostgreSQL内部ではテーブルやインデックスなどはリレーションオブジェクトとして統一的に扱われることが多いようです.

▼リスト8　relation_open( )(一部)

```
if (lockmode != NoLock)
    LockRelation(r, lockmode);
```

FALSEを返した場合にエラーメッセージを表示するとともに，トランザクションを中断しています．また，elogはこのときにlongjmp[注17]して次のトランザクションを処理するルートに飛んでしまいますので，elog以下の行は実行されません．

ヘッダファイル置場所の作法とは

新規に関数を追加しただけではプロトタイプ宣言がないため，コンパイル時にワーニングが出てしまいます．PostgreSQLでは，プロトタイプ宣言は別のヘッダファイルに定義することになっています．各サブシステムごとに専用のヘッダファイルがあり，すべてsrc/includeの下に置いてあります．

今回はconditional_relation_open( )をbackend/access/heap/heapam.cに追加しました．ヘッダファイル名はプログラムファイルの拡張子を“.c”から“.h”に変更したものになります．ただし，includeの直下ではなく，プログラムファイルの置いてあるディレクトリからサブディレクトリ以下を省略したディレクトリをincludeの下に作り，そこに置くようになっています．たとえば，access/heap/heapam.cに対応するヘッダファイルはinclude/access/heapam.hになります．

このファイルの138行目くらいにrelation_open( )のプロトタイプがありますから，この下に，

```
extern Relation conditional_relation_open(Oid
relationId, LOCKMODE lockmode, bool no_wait);
```

(実際は1行)

を追加しておきましょう．

▼リスト9　リスト8の変更

```
if (lockmode != NoLock)
{
    if (no_wait)
    {
        if (!ConditionalLockRelation(r, lockmode))
        elog(ERROR, "Cannot aquire relation lock");
    }
    else
        LockRelation(r, lockmode);
}
```

動作確認

以上で修正は終わりです．再コンパイルとインストールが終わったら，さっそく動作を確認してみましょう．確認はpsqlを2つ立ち上げるのが簡単です．

図6では，2つのpsqlをそれぞれT1，T2で表します．ここではテスト用のテーブルを“sales”としましたが，別のテーブルでもかまいません．うまく動いているようですね．

最後に

PostgreSQLのソースを修正する方法を，ごく簡単に説明しました[注18]．

今回はPostgreSQLのソースという巨大な世界の中でも，初心者向けの歩きやすいハイキングコースを取り上げたに過ぎません．あとは読者の方々自らわき道に入り込んで少しずつPostgreSQLをハックしていくとよいでしょう．

今回は「待ち合わせなしのロック」というテーマでLOCK文を改造しましたが，実はSELECT FOR UPDATEという構文でもロックで待たされるので，本当はこちらのほうも改造しないといけません．要望が多ければ，何かの機会に続きを取り上げることもできるかもしれません．興味のある方は編集部までリクエストを！WD

▼図6　動作確認

```
T1: BEGIN;                          -- トランザクションの開始
T2: BEGIN;                          -- トランザクションの開始
T1: LOCK TABLE sales;               -- salesテーブルをロックする
T2: LOCK TABLE sales NO WAIT;       -- salesテーブルをロックしようとするが，すでにT1がロックしているのでエラーになる
ERROR:  Cannot aquire relation lock
```

注17）longjmpは強制的にある個所（setjmp( )を呼び出した個所）に制御を移す特殊な関数です．

注18）今回修正した部分はパッチとして本誌のWebサイト（http://www.gihyo.co.jp/magazines/wdpress/）で公開しています（no_wait.patch）．