PostgreSQL

Chapter 3

Learning PostgreSQL

PostgreSQLをより深く知る

PostgreSQL

3-1 PostgreSQLのプロセス&モジュール構造

ここまでお読みいただいた方は，すでにPostgreSQLをインストールして簡単なデータベース操作ができるようになっていることと思います．本章では，より本格的にPostgreSQLを使いこなすために必要な事柄を説明します．

◆ ◆ ◆ ◆ ◆

本節では，PostgreSQLを構成するプロセスの成り立ち（生成／消滅），そしてモジュール構造について説明します．やや専門的な話になりますが，実装面について理解することがPostgreSQLをより深く知るための助けになるので，お付き合いください．

3.1.1 PostgreSQLを構成するプロセス

PostgreSQLは，いわゆるクライアント／サーバ構成を取っています．すなわち，クライアントであるアプリケーションプログラムはデータベースサーバに接続し，ネットワークを通じて問い合わせを発行し，その結果を受け取ります．PostgreSQLでは，クライアント側を「フロントエンド」，サーバ側を「バックエンド」と呼びます．

バックエンド側は実際には2つのプロセス[注1]に分かれています．ひとつはpostmasterで，一度起動されたらずっと動き続けるデーモンプログラムです．postmasterの役目は主に以下の3つです．

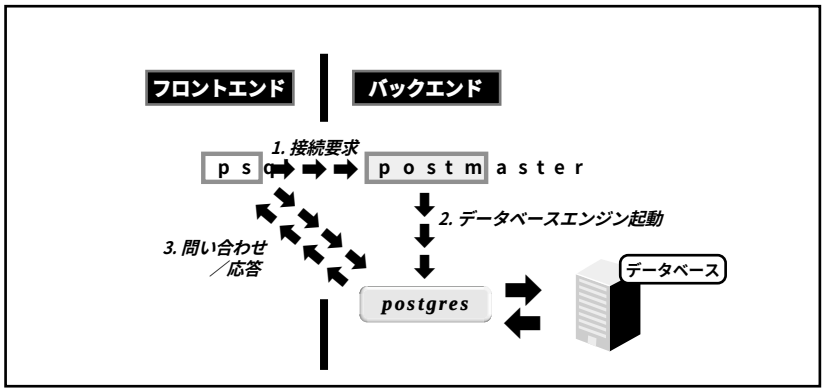
- フロントエンドからの接続要求を受け付ける
- 要求を受け付けたらデータベースエンジンプロセスを起動する
- データベースエンジンの間で共有されるメモリ資源を管理する

たとえばpsqlを使うときのことを考えてみましょう（図3.1.1）．psqlは起動されると，ホスト名（省略時は自ホスト），ポート番号（省略時は5432）の組で指定されるpostmasterプロセスに接続します（1）．postmasterはpsqlの指定したデータベースを引数にしてデータベースエンジン（postgres）のプロセス[注2]を生成します（2）．

●注1
プロセスとはOSがプログラムを実行する際の単位のひとつで，関数やサブルーチンとは違って個々のプロセスは自分用のメモリ空間などの資源を持ち，独立性が高くなっています．アプリケーションプロセスからデータベースプロセスを分離することにより，バグなどのアプリケーションプログラムの悪影響を受けなくなる利点があります．

●注2
実際には，postmasterはpostgresへのシンボリックリンクになっており，両者は同じコードを共有します．にもかかわらず，postgresの起動は6.3.2まではプロセスを生成するfork()と，プログラムコードをロードするexec()で行っていましたが，exec()は無駄だということで，6.4からはfork()のみとなり，オーバヘッドが減少しました．

●図 3.1.1　PostgreSQLを構成するプロセス



postgresが正常に起動されると，以後psqlは直接postgresと通信します（3）．この時点でpostmasterは再び他のフロントエンドからのリクエストを待つ状態に戻ります．

PostgreSQLでは，共有メモリを使って複数のpostgresプロセスが共有するバッファ領域を効率的に管理していますが，postgresプロセスが異常終了すると，このバッファ領域の状態も異常になる可能性があります．postmasterはpostgresが異常終了したことを検知すると，バッファ領域をチェックし，必要ならば初期化を行います．

6.4では時間のかかる問い合わせを途中でキャンセルできるようになりましたが，これにもpostmasterが絡んでいます．フロントエンドの発行したキャンセルリクエストをpostmasterが受信し，シグナル[注3]をpostgresに送って処理を中断させるのです．

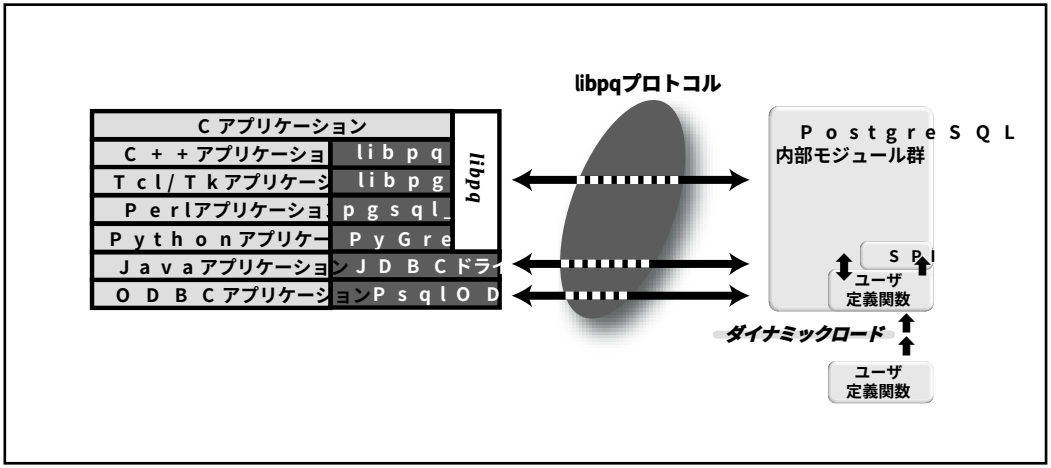
●注3
OSの持つ機能のひとつで，任意の時点でプロセスの実行に割り込みをかけ，情報を伝達することができます．

3.1.2　PostgreSQLのモジュール構造

フロントエンドのモジュール構造

さて，ここまで述べたように，フロントエンドとバックエンドの間では，ネットワークを介してさまざまな通信が行われますが，これを「libpqプロトコル」と呼ぶことにしましょう．PostgreSQLでは，libpqプロトコルを意識せずに容易にアプリケーションを実装できるように，libpqというC言語用のライブラリが提供されています．実は，psqlもそうしたアプリケーションのひとつなのです．また，C言語で書かれたソースプログラムにSQL文を埋め込むためのプリプロセッサであるecpgも，SQL文をlibpqの関数の呼び出しに変換します．

●図3.1.2　PostgreSQLのモジュール構造



C以外の言語用のライブラリも用意されています．これらはほとんどが内部でlibpqの関数を呼び出す形で実装されています．例外はJavaインターフェース（JDBCドライバ）とODBCインターフェースです．これらは直接libpqプロトコルを扱います（図3.1.2）．

バックエンドのモジュール構造

バックエンドのデータベースエンジンであるpostgresには，データベースエンジンとしての機能を実現するために，さまざまなモジュールが組み込まれています．主要なモジュールは以下です．

- パーサ…SQL文を解析し，パースツリーを生成する
- プランナ／オプティマイザ…パースツリーから最適な問い合わせの実行プランを作成しする
- エグゼキュータ…実行プランを実際に実行する

これらの機能の一部は，ユーザ定義関数から呼び出すことができます．また，ユーザ定義関数の中でSQLを実行するために，SPI（Server Programming Interface）が用意されています．SPIを使うと，postgresのパーサ，プランナ，オプティマイザ，エグゼキュータにアクセスすることができます．

3-2 PostgreSQLのソースツリー

PostgreSQLをより深く理解したい方，そして自分でPostgreSQLを改良してみたいという方への最高のドキュメントはソースコードでしょう[注1]．

PostgreSQLはかなり大がかりなシステムです．ソースコードはコメントも含めて数えると，全部で30万行ほどもあります．バックエンドを構成するすべてのモジュールについて詳細な説明をしたいところですが，筆者自身全部を理解しているわけではありませんし[注2]，紙面の都合もあるので，ここではPostgreSQLのソースツリーについて説明するに留めます．

トップレベルのディレクトリ構成については，第2章の表2.3.2（p.26）で説明したので省略します．実際にソースが格納されているsrc/以下（表3.2.1）をまず見てみましょう[注3]．bin/の下はほぼコマンド名がそのままディレクトリ名になっています（表3.2.2）．interfacesの下は，libpqをはじめ，各種プログラミング言語インターフェースです（表3.2.3）．backendの下はバックエンド用のソースです（表3.2.4）．

●注1
6.4からはdoc/FAQ_DEVという開発者向けのFAQが追加されています．

●注2
というか，理解している部分の方が少ないくらいです．PostgreSQLのすべてを把握するにはまだまだ精進が必要なようです．

●注3
toolsにあるmake_ctagsとmake_etagsを実行すると，viやEmacsで使うtagsファイルが生成されるので，関数や構造体の定義を参照する際に便利です．

●表3.2.1 src/以下のファイル／ディレクトリ

DEVELOPERS	開発者向けの注釈
GNUmakefile	GNU make用のトップレベルのMakefile（configureが生成）
GNUmakefile.in	configureが使用するGNUmakefileの雛型
Makefile	ダミーのMakefile
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）
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
config.cache	configureのキャッシュファイル（configureが生成）
config.guess*	configureのサブプログラム
config.log	configureのログファイル（configureが生成）

config.status*	configure実行時のオプションを記憶する．このファイルを実行すると，もう一度同じ設定でconfigureを実行したのと同じ効果がある（configureが生成）
config.sub*	configureのサブプログラム
configure*	configure本体
configure.in	configureの雛型
data/	キリル文字用のデータ
include/	ヘッダファイル
install-sh*	インストールスクリプト
interfaces/	フロントエンドのソース一式
lextest/	lexのテストプログラム
makefiles/	プラットフォーム依存のmake設定値
man/	オンラインマニュアルソース
pl/	プロシージャ言語（C，SQL以外の言語で関数を定義する）6.3.2ではTclのみ．6.4ではPL/pgSQLという独自のSQL言語プロシージャも追加された
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

●表3.2.2　bin/以下のファイル／ディレクトリ

Makefile	makefile
cleardbdir/	cleardbdir（現在使われていません）
createdb/	createdb
createuser/	createuser
destroydb/	destroydb
destroyuser/	destroyuser
initdb/	initdb
initlocation/	initlocation
ipcclean/	ipcclean
pg_dump/	pg_dump
pg_encoding/	pg_encoding（6.4から追加）
pg_id/	pg_id
pg_passwd/	pg_passwd
pg_version/	pg_version
pgaccess/	PostgreSQLへのGUIインターフェースpgaccessのTclソースとドキュメント
pgtclsh/	PostgreSQLへのインターフェース組み込みのtclsh（pgtclsh）とwish（pgtksh）
psql/	psql

●表3.2.3 interfaces以下のファイル／ディレクトリ

Makefile	makefile
ecpg/	C言語用埋め込みSQLプリプロセッサecpg（embedded SQL preprocessor for C）
jdbc/	JDBCドライバ
libpgtcl/	Tclインターフェース
libpq/	libpq
libpq++/	C++用インターフェース
odbc/	ODBCドライバ
perl5/	perl5用インターフェース
python/	Python用インターフェース

●表3.2.4 backend以下のファイル／ディレクトリ

Makefile	makefile
access/	各種アクセスメソッド（以下サブディレクトリ）
nbtree/	Btree
hash/	Hash
rtree/	Rtree
index/	上記アクセスメソッドを使ったインデックスアクセス関数
heap/	テーブル本体のアクセス関数
gist/	Generalized Search Tree（gist）という汎用的なインデックスメソッド*
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/	executor
fmgr.h	関数管理（function manager）用のヘッダファイル．自動生成
global1.bki.source	システムカタログ（pg_databaseなどの全データベース共通）生成用のテンプレート
global1.description	システムカタログ（pg_databaseなどの全データベース共通）用の注釈生成用テンプレート
lib/	共通関数
libpq/	PostgreSQLプロトコル関数
local1_template1.bki.source	template1システムカタログ生成用
local1_template1.description	template1システムカタログ注釈生成用
main/	mainプログラム
nodes/	パースツリー操作関数
optimizer/	オプティマイザ
parse.h	パーサ用ヘッダファイル
parser/	パーサ
port/	プラットフォーム依存コード

＊）ただし現在は使われていないようです．
参考URL●http://s2k-ftp.cs.berkeley.edu/gist/

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ージ，バッファなど，すべての1次/2次記憶管理
tcop/	postgresのメインループ
tioga/	使われていない
utils/	“ utils ”という名前にだまされてはいけない．非常に重要なモジュールがある
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/	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/	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

3-3 PostgreSQLの問い合わせ言語 ●その１：データ型

データベースシステムに対する指令は，問い合わせ言語というものを通じて行います．PostgreSQLの問い合わせ言語は，標準的なSQLに独自の拡張を加えたものです．本節では，PostgreSQLのSQLがサポートする問い合わせ言語のうち，データ型について説明します．

なお本書では，SQL自体については解説しません．必要ならば，巻末の参考文献などに挙げたSQLの解説書をご覧ください．

3.3.1 概要

PostgreSQLは，SQL92で定義されたほとんどのデータ型と，SQL3のデータ型の一部をサポートします．表3.3.1にPostgreSQLでのデータ型と，対応するSQL92／SQL3におけるデータ型を示します．SQL文中では，PostgreSQLでのデータ型名も，SQL92／SQL3におけるデータ型名も，どちらも同じように使えます．

PostgreSQLでの制限事項およびSQLとの違いを以下に示します．

- numeric, およびdecimalは内部的にint4で実装されているため，スケールs（小数部分の精度）は0以外指定できません．また，numericとdecimalは内部的にはまったく同じものなので，精度pも同じです．
- SQL92では，realは単精度浮動小数点と規定されていますが，なぜかPostgreSQLではfloat8として実装されています．
- time with time zoneはサポートされていません．
- timestamp with time zoneはサポートされていません．
- bitおよびbit varyingはサポートされていません．
- national character（nchar）はサポートされていません．

PostgreSQLには，SQL92では定義されていない独自のデータ型もあります．これらのうち，アプリケーションを構築する上で有用と思われるデータ型を表3.3.2に示します．PostgreSQLに定義されているすべてのデータ型を参照するには，psqlのバックスラッシュコマンドである\dTをお使いください．

次に，これらのデータ型を使う上での留意事項を述べます．

●表3.3.1　PostgreSQLとSQL92／SQL3との違い

PostgreSQLにおけるデータ型	SQL92におけるデータ型	コメント
char	characterまたはchar	文字
char(n)	character(n)またはchar(n)	固定長文字列
varchar(n)	character varying(n)またはchar varying(n)またはvarchar(n)	可変長文字列
float4/8	float(p)	精度pの浮動小数点
float8	double precision	倍精度浮動小数点
float8	real	単精度浮動小数点
int4	integerまたはint	符号つき整数
int2	smallint	小桁符号つき整数
int4	numeric(p,s)	任意精度の10進数数値
int4	decimal(p,s)	任意精度の10進数数値
date	date	日付（年月日）
time	time (with timezone)	時刻（時分秒）
timestamp	timestamp (with time zone)	日付と時刻
timespan	interval	時間間隔

PostgreSQLにおけるデータ型	SQL3におけるデータ型	コメント
bool	boolean	true/falseの論理値

●表3.3.2　PostgreSQL独自のデータ型

データ型	コメント	使用方法／説明
box	矩形	1.0,2.0のように，左下と右上の点を指定する
circle	円	中心(x,y)，半径rの円は，(x,y),<r>またはx,y,rのように指定する
lseg	直線	始点と終点を指定する：[(x1,y1),(x2,y2)]／(x1,y1),(x2,y2)／x1,y1,x2,y2／[x1,y1,x2,y2]／(x1,y1,x2,y2)のいずれかのフォーマット
path	経路	経路上の点のリストを指定する：[(x1,y1),(x2,y2),...]／(x1,y1),(x2,y2),...／x1,y1,x2,y2,...／[x1,y1,x2,y2,...]／(x1,y1,x2,y2,...)のいずれかのフォーマット
point	点	(x, y)およびx,yのいずれかのフォーマット
polygon	多角形	頂点のリストを指定する：(x1,y1),(x2,y2),...／x1,y1,x2,y2,...／(x1,y1,x2,y2,...)のいずれかのフォーマット
text	可変長テキスト	
int8	8バイト整数	contrib/に添付．サポートされないプラットフォームもある

データ長の制限

PostgeSQLでは，1レコードの大きさは，8192バイトを超えることはできません[注1]．このため，配列や可変長文字列の大きさもたかだか8192バイト以内でなければなりません．1レコードが複数のカラムからなる場合には，その合計が8192バイト以内でなければなりません．

8192バイトよりも大きなデータを扱うためには，後述するlarge objectという特殊なデータ型を使います．

●注1
実際には，システムが利用するオーバーヘッドがあるため，8192バイトよりは少なくなります．

マルチバイト文字

日本語などのマルチバイト文字を使用する場合，文字コードによって1文字を表現するのに必要なバイト数が異なります．char/varchar/textなどの文字列型のデータ型でマルチバイト文字を使用する場合は，文字数ではなく，マルチバイト文字の内部表現に使用されるバイト数で桁数を指定してください．表3.3.3に日本語の場合の各種

●表3.3.3　日本語文字コードと必要なバイト数

文字コード	文字種	バイト数
日本語EUC	ASCII	1
	JISX0208	2
	JISX0201（カタカナ）	2
	JISX0212	3
UNICODE(UTF-8)	ASCII	1
	JISX0208	3
	JISX0201（カタカナ）	3
	JISX0212	3
MULE_INTERNAL	ASCII	1
	JISX0208	3
	JISX0201（カタカナ）	2
	JISX0212	3

必要なバイト数の調べ方

octet_length()という関数を使えば，ある文字を表現するのに何バイト必要か調べることができます．「日本と」は日本語EUCなので，1文字あたり2バイトで2×3＝6バイト，「America」はASCII文字ですから1×7＝7バイト，合計13バイトでたしかに計算が合っています．

```
test=> select octet_length(text '日本とAmerica');
octet_length
--------------
          13
(1 row)
```

文字コードと必要なバイト数を示します．なお，文字列を扱う関数では，逆にバイト数でなく文字数を指定します．詳細は3.3.3をご覧ください．

では，個々のデータ型を詳しく見ていくことにします．

3.3.2　数値データ型

PostgreSQLの数値データ型を表3.3.4に示します．必要な値の範囲と，データベース上のバイト数の兼ね合いで最適なデータ型を決定してください．

演算子

数値型のデータでは，表3.3.5のような演算子が使えます．

●表 3.3.4　PostgreSQLの数値データ型

データ型	データベース上のバイト数	値の範囲
int2	2バイト	-32768～+32767
int4	4バイト	-2147483648～+2147483647
int8	8バイト	-9223372036854775808～+9223372036854775807
float4	4バイト	—
float8	8バイト	—

●表 3.3.5　数値型で使える演算子

演算子	使用例	意味
!	3 !	3!
!!	!! 3	3!
%	5 % 4	5 mod 4（剰余）
%	% 4.5	小数点以下の切り捨て
*	2 * 3	2×3
+	2 + 3	2＋3
-	2 - 3	2－3
/	4 / 2	4÷2
:	: 3.0	e^3
;	(; 5.0)	log5
@	@ -5.0	|-5.0|
^	2.0 ^ 3.0	2^3
|/	|/ 25.0	$\sqrt{25}$
||/	||/ 27.0	$\sqrt[3]{27}$

関数

数値データの関数は表3.3.6のものが用意されています．ここで，たとえばdexp(float8)は，引数の型としてfloat8を取ることを意味します．また，演算子：はdexp( )の呼び出しに，^はdpow( )の呼び出しに変換されます．つまり，これらは実際には同じものなのです．

3.3.3　文字データ型

表3.3.7に文字データ型を示します．textとvarcharは共に可変長文字列ですが，最大文字数を指定する必要がないこと，また効率がよいことから，PostgreSQLではtextを使うことが推奨されています．SQL92との互換性を重視する場合は，varcharを使えばよいでしょう．SQL92にはnational characterというデータ型がありますが，PostgreSQLではサポートされていません．ただし，普通の文字列型で英語以外の文字を扱うことができるので，実際には不自由しません．

6.3.2ではひとつだけ注意する点があります．char(n)やvarchar(n)で，nよりも大きな文字列が入力された場合，PostgreSQLは勝手にnで文字列を切ってしまいます．英語のように1文字が1バイトの文字コードの場合はよいのですが，日本語のようなマ

●表 3.3.6　数値データの関数

関数名	戻り値の型	使用例	意味
dexp(float8)	float8	dexp(2.0)	e^2
dpow(float8,float8)	float8	dpow(2.0, 16.0)	2^{16}
float(int)	float8	float(2)	intからfloat8への変換
float4(int)	float4	float(2)	intからfloat4 への変換
integer(float)	int	integer(2.0)	floatからintへの変換

●表 3.3.7　文字データ型

データ型	データベース上のバイト数	説明
char	1バイト	1バイト文字．マルチバイト文字は格納できない
char(n)	（4+n）バイト	固定長文字列．入力された文字が指定桁数に満たない場合は空白文字が詰められる．nは4096以下でなければならない
text	（4+x）バイト	可変長文字列．PostgreSQL独自のデータ型
varchar(n)	（4+n）バイト	制限付きの可変長文字列．nは4096以下でなければならない

ルチバイト文字では，場合によっては都合の悪いところで文字列が切断され，マルチバイトの各バイトが「生き別れ」になってしまうことがあります．こうなってしまうと，もはや文字コードとして正しいものではなくなるので，表示などの処理に不都合が生じます．この問題は，6.4以降では修正されています．

このほかにnameという文字列型がありますが，システムカタログのためのデータ型であり，アプリケーションでの利用は推奨されていません．

演算子

SQL92で規定されている||（文字の連結）やLIKE述語は，もちろんPostgreSQLでも使えます[注2]．そのほかにPostgreSQL独自のものとして，正規表現があります．

正規表現はUNIXのシェルやgrepなどでおなじみの機能で，特殊な文字を使って文字パターンを指定するものです．LIKEをもっとずっと汎用的にしたもの，と言えるかもしれません．PostgreSQLで使える正規表現は，POSIX 1003.2の“basic”正規表現で，大ざっぱに言って，UNIXのedエディタで使える正規表現と大体同じです[注3]．主な正規表現を表3.3.8に示します．

SQL文の中では，正規表現の演算子は~（チルダ）です．その否定は!~です．英文の大文字と小文字を区別しない正規表現演算子もあります．~*と!~*（否定）です．たとえば，abcで始まる文字列を探すSQL文はLIKEを使うと，

```
select * from mytext like 'abc%';
```

ですが，正規表現を使うと，

●注2
ただし，ESCAPEではエスケープ文字を指定できません．エスケープ文字は\（バックスラッシュ）に固定されています．

●注3
GNUのedを除きます．GNUのedは大幅に強化されています．

●表3.3.8　主な正規表現

正規表現	意味	例
.	任意の1文字	“あ.う”は“あいう”“あううなどにマッチ
a*	0個以上の文字	“あ*う”は“あう”“ああう”“うなどにマッチ
.*	任意の個数の任意の文字	
[abc]	どれかの文字	“[あいう]”は“あ”“い”“うのどれかにマッチ
[^abc]	abc以外の文字	“[^あいう]”は“あ”“い”“う”以外にマッチ
^abc	abcで始まる文字	“^あいう”は“あいう”“あいうえなどにマッチ
abc$	abcで終わる文字	“あいう$”は“あいう”“ああいうなどにマッチ
\a	その文字そのもの	
\\	バックスラッシュそのもの	
上記以外	その文字そのもの	

```
select * from mytext ~ '^abc';
```

となります．応用例としてabcに対して，大文字，小文字，JIS0212アルファベット（いわゆる「全角」アルファベット）を区別せずに検索する正規表現を考えてみましょう．

```
select * from mytext ~ '[aAａＡ][bBｂＢ][cCｃＣ]';
```

となります．~*を使えばもう少し節約できて，

```
select * from mytext ~ '[aａＡ][bｂＢ][cｃＣ]';
```

で同じ結果が得られます．

ところでlikeや正規表現を使う場合，「前方一致」検索以外では，たとえインデックスが定義されていてもインデックスが使われず，その結果検索速度が遅くなることに注意してください．ちなみに前方一致とは，「ある文字列で始まるもの」という意味でabc%や^abcはその一例です．

SQL92関数

PostgreSQLでは，SQL92で定義されている文字列関係の関数のうち，以下のものをサポートしています．translateとconvertはサポートされていません．

■position(*string* in *source*)

positionは，sourceの中のstringの開始位置を返します．返り値の型はint4（int）で，位置は1から始まります．文字列が見つからない場合は0を返します．たとえば

```
position('English' in '日本語とEnglish')
```

は5を返します（日本語も1文字につき1と数えます）．

■substring(string from *from_position* [for *for_position*])

substringは，stringのfrom_position文字目からfor_position文字を取り出して返します．for_positionを省略すると，文字列の最後までを指定したものとみなします．たとえば

```
substring('日本語とEnglish' from 5)
```

は，Englishを返します（日本語も1文字につき1つと数えます）．

■trim([leading|trailing|both] [*omitt_text*] from *string*)

trimは，stringからomitt_textを取り除いたものを返します．leadingはstringの前，trailingはstringの後，bothは前後から取り除く指定です．omitt_textを省略すると，空白文字を指定したものとみなします．たとえば

```
trim(trailing 'とEnglish' from '日本語とEnglish')
```

は日本語を返します．

■upper

upperは大文字に変換した文字列を返します．日本語は影響を受けません．たとえば

```
upper('日本語とEnglish')
```

は日本語とENGLISHを返します．

■lower

lowerは小文字に変換した文字列を返します．日本語は影響を受けません．たとえば

```
lower('日本語とEnglish')
```

は日本語とenglishを返します．

■character_length（char_length）

character_length（char_lengthは短縮形）は文字列の長さを返します．

```
character_length('日本語とEnglish')
```

は11を返します．日本語も1文字につき1つと数えます．

■octet_length

octet_lengthは文字列の長さをバイト数で返します．

```
octet_length('日本語とEnglish')
```

は文字コードがEUC_JPの場合，15を返します．

その他の関数

■initcap

単語の先頭を大文字にします．

```
initcap('abc def')
```

はAbc Defが返ります．

■lpad/rpad

第1引数の左側に第3引数の文字列を，全体が第2引数の長さになるまで繰り返し追加します．

```
lpad('abc',6,'de')
```

はdedabcが返ります．rpadはlpadと反対に右に文字列を追加します．なお，lpad/rpadはマルチバイト対応していないので，第2引数はバイト数で指定してください．

■translate

第1引数の文字列に出現するすべての第2引数の文字列を，第3引数の文字列で置き換えます．

```
translate('12345', '1', 'a')
```

はa2345を返します．なお，この関数とSQL92のtranslateとはまったく別物です．

■その他の関数

SQL92関数と同じ機能を持った関数がいくつかあります．ltrim/rtrim（＝trim），position（＝position），substr（＝substring）です．これらはSQL92関数が導入される前に，ある商用データベースの互換関数として実装されたものです．SQL92関数がサポートされた今となっては存在価値がなくなったと言えるので，ここでは説明を省略します．

3.3.4　日付データ型

PostgreSQLの日付データの扱いは，SQL92に比べるとかなり強力です．たとえば，SQL92では紀元前の日付を扱えませんが，PostgreSQLのdatetime型は紀元前4713年からほぼ無限の未来まで扱うことができますし，タイムゾーンの扱いもより実用的になっています．

また，SQL92で定義されているデータ型はほとんどサポートしていますが，若干構文が異なる部分もあります．

問題点としては，歴史的な理由からデータ型の種類が多く，やや混沌とした印象を受けることでです．将来的にはSQL92のデータ型に統一される予定のようです．

SQL92にはdate（年月日），time（時分秒），timestamp（年月日時分秒），およびinterval（時間差）の日付関係のデータ型があります．では，これらのデータ型についてSQL92で定義とPostgreSQLの実装を比較しながら見ていきましょう．

date

このデータ型は年月日を表現します．年の範囲は，SQL92では1～9999ですが，PostgreSQLではBC4713（紀元前4713年）11月13日からAC32767（紀元後32767年）12月31日までを扱うことができます．date型は，PostgreSQLではDateADTという型で実装されていますが，実際にはDateADTは4バイトの整数です．

■入力形式

SQL92の規定ではdate型のデータは以下の形式で入力します．

```
'年（４桁）-月（２桁）-日（２桁）'
```

PostgreSQLでは，これ以外にもかなり自由な形式でデータ入力できます．

```
'1998-09-23'
'19980923'
'09 23 1998'
'09-23-1998'
'1998/09/23'
'09/23/1993'
```

```
'Wed Sep 23 12:09:18 JST 1998'
'Wed Sep 23 1998'
'Sep 23 1998'
'1998 Sep 23'
```

これらはいずれも1998年9月23日として扱われます．

■出力形式

SQL92ではデータ出力する場合も入力と同じ'年（4桁）−月（2桁）−日（2桁）'ですが，PostgreSQLではいくつかの出力形式が選択できます．出力形式を選択するには，以下の3通りの方法があります．

- postmasterを起動する際に，環境変数PGDATESTYLEに出力形式名を設定する：この場合，以後すべてのフロントエンドに対して同じ出力形式が適用されます．
- フロントエンド（libpqを使用しているもの）を起動する際に環境変数PGDATESTYLEに出力形式名を設定する：この場合，このフロントエンドに対してのみ出力形式が適用されます．
- SQL文「set datestyle to '出力形式名';」を実行する：この場合，セッションの途中で自由に出力形式を切り替えることができます．

表3.3.9に指定可能な出力形式名と対応する出力例を示します．なお，出力形式名は大文字／小文字が区別されません．これらの出力形式のうち，SQL，Postgres，Germanでは，月と日の順序を変更することができます．最初に上記の出力形式を設定した後，続いて

```
set datestyle to 'European'
```

とすると，日の次に月が来ます．逆に

```
set datestyle to 'NonEuropean'
```

●表3.3.9　指定可能な出力形式と出力例

出力形式名	出力例	備考
ISO	1998-09-23	
SQL	09/23/1998	
Postgres	09-23-1998	デフォルト
German	23.09.1998	

もしくは'US'とすると，月の次に日が来ます（こちらがデフォルトです）．
European，NonEuropean，USの単独の指定では，

```
set datestyle to 'European'
```

で23/09/1998の形式に，

```
set datestyle to 'NonEuropean'（または'US'）
```

で09/23/1998の出力形式になります．

■特殊な定数

PostgreSQLでは，date型に対して表3.3.10の特殊な定数を使うことができます．たとえば，本日の日付が「1998年9月23日」だとすると，次のように出力されます．

```
select 'tomorrow'::date;
  ?column?
----------
09-24-1998
(1 row)
```

ところで“current”と“now”はどちらも「現在」を表しますが，微妙な違いがあります．次のようなテーブルを考えてみましょう．

```
create table t1 (d date);
```

このテーブルに

```
insert into t1 values('now');
```

でデータを挿入すると，それはまさに現在の日付が登録されます．したがって，次に

●表3.3.10　特殊な定数

定数名	意味
current	現在
now	現在
today	本日
yesterday	昨日
tomorrow	明日
epoch	1970年1月1日

そのカラムをselect文で取り出すと，登録した時点の日付が出力されます．ところがcurrentでは，

```
insert into t1 values('current');
```

とすると「現在」というマークが登録されるだけなのです．したがってそのカラムをselect文で取り出すと，selectした時点の日付が出力されます（実際には'current'と出力されます）．

time

このデータ型は時分秒を表現します．時は24時間制なので，0～23の値になります．分は0～59です．秒はSQL92では0～61の値を取ります[注4]．PostgreSQLでは秒は0～59までしか許されていません．またSQL92では，time(9)のようにしてコンマ以下の秒の桁数を指定できますが，PostgreSQLではこのような指定できません．したがって，秒以下の桁を表示させることはできません．ただし，内部的にはマイクロセカンド（＝1/1000000秒）の精度で処理を行っているので，

●注4
秒が61まで許されているのは「うるう秒」のためです．

```
12:04:05.1234
```

のように.で区切って秒以下の桁を入力することができます．実際に秒以下のデータが処理されていることは，以下のようにして確かめることができます．

```
select date_part('millisecond','1998/9/23
12:04:05.1234'::datetime);
date_part
---------
    123.4
(1 row)
```

ここで，date_part()は時刻データの指定部分を取り出す関数です，date_part()を使って秒以下の部分をミリセカンド（=1/1000秒）単位で取り出しているわけです．

また，SQL92では

```
time '12:04:05.1234' at local
time '12:04:05.1234' at time zone interval '+09:00' hour to minute
```

のようにタイムゾーンを指定することができますが，PostgreSQLではできません．こ

れは「タイムゾーンは日付，時刻と関連させなければ無意味であるから」と説明されています．

time型は，PostgreSQLではTimeADTという型で実装されていますが，実際にはTimeADTは8バイトの浮動小数点です．

timestamp

このデータ型は年月日と時分秒を表現します．

年の範囲は，SQL92ではdate型と同じく1～9999ですが，PostgreSQLでは1901年12月14日から2038年1月19日までしか扱うことができません．

時分秒の方は，SQL92ではtime型とほとんど同じ扱いですが，PostgreSQLでは，1秒以下のデータは表現できません．

PostgreSQLのtimestamp型は実装時期が古いこともあって，制約が多く，あまりお勧めできるものではありません．同じ機能でより広い範囲を表現でき，精度も高い後述のdatetime型を使う方がよいでしょう．

datetime

SQL92では定義されていないPostgreSQL特有のデータ型ですが，機能はほぼSQL92のtimestamp型と同じで，より広い範囲をより高い精度で表現できます．

扱える年の範囲は，BC4713からかなり未来までを扱うことができます．いったいどこまで扱えるのかよくわかりませんが，筆者の環境では少なくとも999999年までは大丈夫でした．時分秒の方は，マイクロセカンドの精度まで表現できます．datetime型は，PostgreSQLではDateTimeという型で実装されていますが，実際にはDateTimeは8バイトの浮動小数点です．

■入力形式

年月日はdate型，時分秒の部分はtime型と同じ入力形式で，これ以外に紀元前／紀元後（BC/AD）の指定，タイムゾーンの指定がオプションで可能です．以下，入力例を示します．

```
'1998-09-23 12:05:10'
'19980923 12:05:10'
'9 23 12:05:10 1998'
```

```
'1998-09-23 12:05:10 AD'
'1998-09-23 12:05:10 BC'
'1998-09-23 12:05:10 JST'
'Wed Sep 23 12:09:18 JST 1998'
'Wed Sep 23 12:09:18 JST BC 1998'
'1998-09-23 12:05:10 HST'
```

興味深いのは最後の例で，ハワイ時間（HST）を日本時間に変換することができます．

```
select '1998-09-23 12:05:10 HST'::datetime;
?column?
----------------------------
Thu Sep 24 07:05:10 1998 JST
(1 row)
```

日本時間（JST）ではなくて，ニューヨーク時間（EST）にしたいときは，

```
set timezone to 'EST';
```

を実行すると[注5]，そのセッションのタイムゾーンが以後ニューヨーク時間になるので，

```
test=> select '1998-09-23 12:05:10 HST'::datetime;
?column?
----------------------------
Wed Sep 23 17:05:10 1998 EST
(1 row)
```

という結果が得られます．

●注5
タイムゾーンの名前は大文字／小文字が区別されるので，“est”ではだめです．また不正なタイムゾーン名がセットされると，エラーにはならずにタイムゾーンがUTCにリセットされます．

■出力形式

date型と同様，datestyleで出力形式を変更することができます．基本的にはdate型と同様ですが，タイムゾーンの形式も影響を受けます．datestyleにisoを指定すると，

```
1998-09-23 12:05:10+09
```

のようにタイムゾーンがUTCからのオフセットで表示されます．iso以外のdatestyleでは，タイムゾーンはJSTのようなアルファベットになります．

特殊な定数

date型の定数に加え，以下の定数が使えます．

- infinity：datetime型で扱えるどんな日付よりも大きい日付（無限大）
- -infinity：datetime型で扱えるどんな日付よりも小さい日付（無限小）

interval

interval型は，日付，時刻の差を扱うデータ型です．SQL92のinterval型とPostgreSQLのintervalでは，入出力形式など，かなり違う部分があります．

まず，SQL92のintervalは，日付インターバルと日時インターバルの2つに厳密に分かれ，両者を混合することはできません．それに対してPostrgeSQLのintervalは1つしかありません．また，SQL92のintervalは定数を掛けたり割ったりすることができますが，PostgreSQLではできません．

■入力形式

直接intervalのデータ値を指定するには以下のように年月日などを入力します（@はあってもなくてもかまいません）．

```
'@ 1 day'
'@ 10 years 3 months 1 day'
'@ 10 years 3 months 1 day 2 hours ago'
```

また，intervalは次のように，日時データの差から得ることもできます．

```
timestamp '1998/2/1' - timestamp '1998/3/1'
```

■出力形式

```
select timestamp '1998/2/1 11:23:45.123' - timestamp '1998/3/1
18:04:22.122';
?column?
------------------------------------
@ 28 days 6 hours 40 mins 37 secs ago
```

```
(1 row)
```

のように表示されます．

■実装

interval型は内部的にはtimespanというデータ型と同じものです．12バイトの構造体として実現されています．

その他のデータ型

このほかにもabstime，reltimeという日時に関するデータ型がありますが，扱える範囲が狭く，精度も低いので利用はお勧めしません．

関数

PostgreSQLには年齢を数える関数（age）や，日付データから年や月など特定の部分を取り出す関数（date_part）などがあります．ここではいくつかの例を示します．

■生年月日から現在の年齢を秒単位で得る

```
select age('now','1980/1/1');
```

■生年月日から現在の年齢を年月まで得る

```
select date_trunc('month',age('now','1980/1/1'));
```

■生年月日から現在の年齢を得る

```
select date_part('year',age('now','1980/1/1'));
```

3.3.5　地理データ型

地理データ型はSQL92にはない，PostgreSQL独特のものです．各種2次元データを扱うことができるpoint，circleなどの各種データ型があります．利用可能な地理データ型の一覧は3.3.1ですでに述べたので，ここではオペレータと関数を紹介します．

オペレータ

表3.3.11に示します．

●表 3.3.11　地理データ型のオペレータと使用例

オペレータ	使用例	意味
+	'((0,0),(1,1))'::box + '(2.0,0)'::point	point(x,y)分だけ平行移動する
-	'((0,0),(1,1))'::box - '(2.0,0)'::point	point(x,y)分だけ平行移動する（マイナス方向への移動）
*	'((0,0),(1,1))'::box * '(2.0,0)'::point	point(x,y)分拡大し回転する
/	'((0,0),(1,1))'::box / '(2.0,0)'::point	point(x,y)分縮小し回転する
#	'((1,-1),(-1,1))'::box # '((1,1),(-1,-1))'::box	図形の重なった部分を返す
#	# '((1,0),(0,1),(-1,0))'::polygon	polygonの頂点の数を返す
##	'(0,0)'::point ## '((2,0),(0,2))'::lseg	2つのオブジェクトの最も近接した点を返す．このようにpointとlsegの場合，pointからlsegに引いた垂線がlsegと交わる点を返す
&&	'((0,0),(1,1))'::box && '((0,0),(2,2))'::box	2つのオブジェクトが重なり合っているかどうかをtrueまたはfalseで返す
&<	'((0,0),(1,1))'::box &< '((0,0),(2,2))'::box	2つのオブジェクトが重なり合っているか，第1オブジェクトが第2オブジェクトの左にあるならtrueを返す
&>	'((0,0),(3,3))'::box &> '((0,0),(2,2))'::box	&<とは反対に，2つのオブジェクトが重なりあっているかもしくは第1オブジェクトが第2オブジェクトの右にあるならtrueを返す
<->	'((0,0),1)'::circle <-> '((5,0),1)'::circle	2つのオブジェクトの距離
<<	'((0,0),1)'::circle << '((5,0),1)'::circle	第1オブジェクトが第2オブジェクトの左にあるならtrueを返す
<^	'((0,0),1)'::circle <^ '((0,5),1)'::circle	第1オブジェクトが第2オブジェクトの上にあるならtrueを返す
>>	'((5,0),1)'::circle >> '((0,0),1)'::circle	第1オブジェクトが第2オブジェクトの右にあるならtrueを返す
>^	'((0,5),1)'::circle >^ '((0,0),1)'::circle	第1オブジェクトが第2オブジェクトの下にあるならtrueを返す
?#	'((-1,0),(1,0))'::lseg ?# '((-2,-2),(2,2))'::box;	2つのオブジェクトが重なり合っているかあるいは交わっているならtrueを返す
?-	'(1,0)'::point ?- '(0,0)'::point	2つの点を通る直線が水平ならtrueを返す
?-¦	'((0,0),(0,1))'::lseg ?-¦ '((0,0),(1,0))'::lseg	2つの線分が垂直に交わるならtrueを返す
@-@	@-@ '((0,0),(1,0))'::path	lsegまたはpathの長さ
?¦	'(0,1)'::point ?¦ '(0,0)'::point	2つの点を通る直線が垂直ならtrueを返す
?¦¦	'((-1,0),(1,0))'::lseg ?¦¦ '((-1,2),(1,2))'::lseg	2つのlsegが平行ならtrueを返す
@	'(1,1)'::point @ '((0,0),2)'::circle	第1オブジェクトが第2オブジェクトに含まれるならtrueを返す
@@	@@ '((0,0),10)'::circle	オブジェクトの中心を返す
~=	'((0,0),(1,1))'::polygon ~= '((1,1),(0,0))'::polygon	2つのオブジェクトが同じならtrueを返す

関数

表3.3.12に示します．なお，自明でない変換関数は表3.3.13に列挙します．

●表 3.3.12　地理データ型の関数

関数名	戻り値の型	使用例	意味
area(box)	float8	area('((0,0),(1,1))'::box)	面積
area(circle)	float8	area('((0,0),2.0)'::circle)	面積
box(box,box)	box	box('((0,0),(1,1))','((0.5,0.5),(2,2))')	2つのboxの重なり
center(box)	point	center('((0,0),(1,2))'::box)	中心
center(circle)	point	center('((0,0),2.0)'::circle)	中心
diameter(circle)	float8	diameter('((0,0),2.0)'::circle)	直径
height(box)	float8	height('((0,0),(1,1))'::box)	高さ
isclosed(path)	bool	isclosed('((0,0),(1,1),(2,0))'::path)	pathが閉じているか
isopen(path)	bool	isopen('[(0,0),(1,1),(2,0)]'::path)	pathが開いているか
length(lseg)	float8	length('((-1,0),(1,0))'::lseg)	長さ
length(path)	float8	length('((0,0),(1,1),(2,0))'::path)	長さ
pclose(path)	path	pclose('[(0,0),(1,1),(2,0)]'::path)	閉じたpathに変換
point(lseg,lseg)	point	point('((-1,0),(1,0))'::lseg,'((-2,-2),(2,2))'::lseg)	交点
points(path)	int4	points('[(0,0),(1,1),(2,0)]'::path)	頂点の数
popen(path)	path	popen('((0,0),(1,1),(2,0))'::path)	開いたpathに変換
radius(circle)	float8	radius('((0,0),2.0)'::circle)	半径
width(box)	float8	width('((0,0),(1,1))'::box)	幅

●表 3.3.13

関数名	戻り値の型	使用例	意味
box(circle)	box	box('((0,0),2.0)'::circle)	円に内接するbox
box(polygon)	box	box('((0,0),(1,1),(2,0))'::polygon)	polygonを含むbox
circle(box)	circle	circle('((0,0),(1,1))'::box)	boxに外接する円
circle(point,float8)	circle	circle('(0,0)'::point,2.0)	pointで中心／半径を指定して円に変換
lseg(box)	lseg	lseg('((-1,0),(1,0))'::box)	boxの対角線をlsegに変換
lseg(point,point) lseg	lseg	('(-1,0)'::point,'(1,0)'::point)	2点を結ぶlseg
path(polygon)	path	path('((0,0),(1,1),(2,0))'::polygon)	polygonをpathに変換
point(circle)	point	point('((0,0),2.0)'::circle)	円の中心
point(lseg,lseg)	point	point('((-1,0),(1,0))'::lseg,'((-2,-2),(2,2))'::lseg)	lsegの交点
point(polygon)	point	point('((0,0),(1,1),(2,0))'::polygon)	polygonの中心
polygon(box)	polygon	polygon('((0,0),(1,1))'::box)	boxをpolygonに変換
polygon(circle)	polygon	polygon('((0,0),2.0)'::circle)	円を12頂点のpolygonに変換
polygon(npts,circle)	polygon	polygon(12,'((0,0),2.0)'::circle)	円を任意の頂点のpolygonに変換
polygon(path)	polygon	polygon('((0,0),(1,1),(2,0))'::path)	pathをpolygonに変換

3.3.6　その他の関数

今までの分類に属さないものとしては以下の関数があります．

- CURRENT_USER：現在のPostgreSQLのユーザ名を返す．SQL92関数
- USER：現在のPostgreSQLのユーザ名を返す．PostgreSQLではCURRENT_USERとまったく同じ．6.4からサポート．SQL92関数
- getdatabaseencoding：現在のデータベースの文字コード名を返す．6.4からサポート

3.3.7　配列

PostgreSQLでは，後述のユーザ定義データ型も含め，どのようなデータ型も配列にすることができます．配列の定義は簡単で，

```
create table myarray (ar int4[10]);
```

のようにデータ型の後に[ ]を付けるだけです．[ ]の中の数字は要素の数です．数字を省略すると，可変長の配列になります．

[ ]を追加すれば，2次元，3次元の配列も作れます．

```
create table myarray2 (ar int4[][][]);
```

データ入力には{ }を使います．

```
insert into myarray values('{1,2,3}');
```

データの取り出しは普通にselectでできます．

```
test=> select * from myarray;
ar
-------
{1,2,3}
(1 row)
```

配列の一部を取り出すには，配列要素を指定します．配列の添字は1から始まります．

```
test=>  select myarray.ar[2] from myarray;
ar
--
 2
(1 row)
```

ただし，

```
select ar[2] from myarray;
```

ではエラーになってしまいます．バグでしょうか．

また，配列の一部を取り出すこともできます．

```
test=> select myarray.ar[2:3] from myarray;
ar
-----
{2,3}
(1 row)
```

できそうでできないのが，配列の中身を横断的に検索することです．たとえば「ar[1]...ar[n]の中から1であるものを探す」というようなことは残念ながらできません[注6]．

配列のデータを更新するのは普通にupdateでできます．

```
update myarray set ar = '{10,11,12}' where myarray.ar[1] = 1;
```

配列の途中から値を更新できることにもなっているのですが，バグがあるようでうまく動きません．

●注6
contrib/array/を使えば，ある程度対応できます．

3-4 PostgreSQLの問い合わせ言語 ●その2：ユーザ定義関数

3.4.1 はじめに

今まで紹介してきた関数は，すべてPostgreSQLにあらかじめ組み込まれた（builtin）関数です．PostgreSQLでは，このほかユーザが自由に関数を定義することができます．ユーザが定義した関数は，builtin関数と同様に問い合わせの中などで使うことができます．

ユーザ定義関数は，プログラミング言語で記述します．6.3.2ではプログラミング言語としてSQL，C，Tclを使うことができます．6.4では，PL/pgSQLという制御構造を記述できるSQL言語も使用できるようになりました．本節では，このうちSQL関数とC関数について説明します．

3.4.2 create function

ユーザ定義関数はcreate function文で定義します．PostgreSQLでは，関数はすべて型を持ち，0～8個の引数を持ちます[注1]．関数の名前が同じでも，引数の型や数が異なれば別の関数として扱われます[注2]．関数名は大文字／小文字が区別されません．

●注1
引数にも型があります．

●注2
引数が同じで関数の型が違う関数は同一のものとして扱われてしまいます．

3.4.3 SQL関数

SQL関数を定義するには，以下のcreate function文を用います．

```
CREATE FUNCTION function_name ([type1, ...typeN])
RETURNS [setof] return_type
AS 'sql-queries'
LANGUAGE 'sql';
```

各キーワードの意味は表3.4.1のようになります．簡単な例として，1を返す関数を作ってみましょう．

```
create function simple() returns int as '
       select 1
' language 'sql';
CREATE
select simple();
simple
------
     1
(1 row)
```

次は複数のタプルを返す関数の例です．このuser_relns()という関数は，システムカタログ以外のテーブル，つまりユーザが定義したテーブル名だけを返します[注3]．

```
CREATE FUNCTION user_relns()
   RETURNS setof name
   AS 'select relname
       from pg_class
```

●注3
この例はPostgreSQL付属のregression testから取りました．

●表3.4.1 create function文のキーワード

function_name	関数名
type1...typeN	引数の型名
[set of]	返り値が複数のタプルを返す場合に指定
return_type	返り値の型名
sql-quries	SQL文．複数のSQL文を書けるが，最後は必ずselect文で終わっていなければならない

```
        where relname !~ ''pg_.*'' and
              relkind <> ''i'' '
    LANGUAGE 'sql';
CREATE

select user_relns();
?column?
--------
t2
t1
(2 rows)
```

このように，複数のタプルが関数から返る場合は，キーワード“setof”を使います．また，

```
relkind <> ''i'' '
```

のように'が連続して2つ使われているのは，create function文の中でSQL問い合わせの区切りを示す'と区別するためです．\を使って，

```
relkind <> \'i\'
```

としても同じです．

PostgreSQLでは，テーブルを作るとそれが型名として自動的に登録されます．このような型が使える場面は限られていますが，そのひとつがcreate function文です．

リスト3.4.1を見てください．まず，addressとpersonという2つのテーブルを作り，データを入れておきます．2つのテーブルはidフィールドで関連付けられています．

これで，personがデータ型としても登録されたことになります．そこで，personを引数として受け取り，そこからidを取り出して同じidを持つaddressテーブルのadrカラムを検索する関数name2addrを定義します．

●リスト3.4.1

```
create table address (id int,zip char(7), adr text);
insert into address values (1, '2240037','横浜市都築区茅ヶ崎南');
insert into address values (2, '1038537','東京都中央区日本橋');
create table person (id int, name text);
insert into person values (1,'石井達夫');
insert into person values (2,'東京太郎');
```

```
create function name2addr(person)
        returns setof text as '
        select adr from address
        where id = $1.id '
        language 'sql';
```

ここで，$1は1番目の引数を示します．$1.idにより，idカラムの値を取り出すことができます．この関数を利用してたとえば次のような検索ができます．

```
select name2addr(p) from person p where p.name = '東京太郎';
?column?
------------------
東京都中央区日本橋
(1 row)
```

3.4.4 C関数

SQL関数は手軽ですが，できることはSQL文で表現できることに限られます．制御構造を記述することもできませんし，後述のfunctional indexも使えません[注4]．C関数を使えばより高度な関数を作ることができます．

C関数を定義するにも，やはりcreate function文を使います．

```
CREATE FUNCTION function_name ([type1, ...typeN]) RETURNS return_type
AS 'object_filename'
LANGUAGE 'c';
```

SQL関数との違いは，ASの後にSQLではなくC関数を定義したオブジェクトファイルの名前を書くことと，LANGUAGEとして'c'とすることです．オブジェクトファイルを作成するためには，まずC言語でC関数を記述しなければなりません．C関数の場合にとくに注意することは，関数名を小文字のアルファベットにすることです．いくつかの記号（アンダースコア，ハイフン）や数字も使えます．

PostgreSQLにおいてC関数を書くためにはいろいろな約束事があります．ここでは，“bigtext”というC関数の作成を例に取ってC関数の具体的な書き方を説明します．

●注4
C関数以外に，PL/TclとPL/pgSQLで定義した関数でもfunctional indexを作成することができます．

bigtextとは…

■large object

PostgreSQLは，8Kバイトを超えるデータを扱うことができません．そのため，text型などを使っても，メールなどちょっと大きなテキストファイルをそのままデータベースに格納することができません．このような場合，PostgreSQLではlarge objectという特殊なデータ格納形式を使います．large objectではデータの大きさに制限がなく[注5]，テキストデータに限らず，画像ファイルなどの大きくなりがちなファイルもデータベースで扱うことができます．

large objectの作成はlo_import( )という組み込み関数を使います．lo_import( )は引数としてファイル名を受け取り，ファイルの内容をデータベースに登録してOID（object id）を返します．OIDはデータベースの中でユニークな数字で，個々のlarge objectを識別することができます．以後，large objectのアクセスはこのOIDを使って行います．たとえば，large objectを読み出すには，lo_export( )という組み込み関数にoidを渡すことによって行います．

●注5
もちろんOSの制限はあります．large objectは1個のファイルになるので，通常2Gバイト（＝2000Mバイト）より大きなデータは扱えません．

■bigtextの使用例

bigtextは，large objectに格納したテキストファイルに対して正規表現検索を行う関数です．引数としてlarge objectのOIDと検索文字列を受け取り，検索結果をtrueまたはfalseで返します．

bigtextは実際にはいくつかのC関数の集まりで，これらがbigtext.cという1個のファイルに書かれています．関数には表3.4.2のものがあります．

bigtextの使用例として，まずテキストの表題とテキスト本体をlarge objectで管理するテーブルを考えましょう．

```
create table big (
        name text,      ←テキストのタイトル
        id      oid     ←テキストを格納するlarge objectのOID
);
```

●表3.4.2　bigtextの関数

bigtextregexeq	検索文字列が見つかればtrueを返す
bigtextregexne	検索文字列が見つからなければtrueを返す
bigtexticregexeq	検索文字列が見つかればtrueを返す（大文字／小文字を区別しない）
bigtexticregexne	検索文字列が見つからなければtrueを返す（大文字／小文字を区別しない）

データはあらかじめUNIXのファイルに書いておき，

```
insert into big (name,id) values ('a big text',lo_import('/tmp/
mytext.txt'));
```

のようにして登録します．このようにして登録したlarge objectとしてのテキストは，bigtextの関数群を使って検索することができます．

たとえば，「データモデル」という文字列を含むテキストのタイトルを検索するには，

```
select name from big where bigtextregexeq(big.id,'データモデル');
```

となります．

bigtextの構造

それではさっそく実際のプログラムを見ていきましょう．ここでご紹介するbigtext関連のファイルは，本書付属CD-ROMのexamples/bigtext/に収録してあります．まず，この下のファイルを適当な場所にコピーします．

```
% cp -r /mnt/cdrom/examples/bigtext .
```

これでbigtextというディレクトリができるはずです．その下には表3.4.3のファイルがあります．

リスト3.4.2は関数のソースbigtext.cです．

●表3.4.3　bigtext/以下のファイル

Makefile	makefile（GNU make用）
bigtext.c	Cソースファイル
bigtext.source	関数の登録，デモ用テーブル定義sql文テンプレート
insert.sh	テスト用テキストデータの登録シェルスクリプト
check.sh	検索デモ用シェルスクリプト
testdata	テスト用データのテキストファイル

●リスト3.4.2　bigtext.c

```
 1: #include "postgres.h"
 2:
 3: #include "utils/elog.h"        /* for elog() */
 4:
 5: #include "storage/fd.h"                /* for O_ */
 6: #include "storage/large_object.h"
```

```
 7:
 8: #include "libpq/libpq-fs.h"    /* for INV_READ */
 9:
10: #include "utils/builtins.h"    /* for text* functions */
11:
12: /* public function prototypes */
13: bool bigtextregexeq(Oid obj, struct varlena *p);
14: bool bigtextregexne(Oid obj, struct varlena *p);
15: bool bigtexticregexeq(Oid obj, struct varlena *p);
16: bool bigtexticregexne(Oid obj, struct varlena *p);
17:
18: static bool check(Oid obj, struct varlena *pattern, bool (*func)())
19: {
20:   LargeObjectDesc *desc;
21:   struct varlena *retval;
22:   int totalread;
23:   int len;
24:   bool t;
25:
26:   desc = inv_open(obj,INV_READ);
27:   if (desc == (LargeObjectDesc *)NULL) {
28:     elog(ERROR, "bigtexteq: couldn't open oid %d", obj);
29:     return(FALSE);
30:   }
31:   len = inv_seek(desc, 0, SEEK_END);
32:   inv_seek(desc, 0, SEEK_SET);
33:   retval = (struct varlena *)palloc(len + VARHDRSZ);
34:   totalread = inv_read(desc, VARDATA(retval), len);
35:   (void)inv_close(desc);
36:
37:   if (totalread != len) {
38:         elog(ERROR, "bigtexteq: read request is %d but returns %d
(oid:%d)",
39:       len,totalread,obj);
40:     return(FALSE);
41:   }
42:
43:   VARSIZE(retval) = totalread + VARHDRSZ;
44:   t = (*func)(retval,pattern);
45:   pfree((char *)retval);
46:   return(t);
47: }
48:
49: bool bigtextregexeq(Oid obj, struct varlena *p)
50: {
51:   return(check(obj,p,textregexeq));
52: }
53:
54: bool bigtextregexne(Oid obj, struct varlena *p)
55: {
56:   return(check(obj,p,textregexne));
57: }
58:
59: bool bigtexticregexeq(Oid obj, struct varlena *p)
60: {
61:   return(check(obj,p,texticregexeq));
62: }
63:
64: bool bigtexticregexne(Oid obj, struct varlena *p)
65: {
66:   return(check(obj,p,texticregexne));
67: }
68:
```

■ヘッダファイル

まず必要なヘッダファイルは“postgres.h”です（1行目）．postgres.hは，PostgreSQLのソースディレクトリの下src/includeにあります．C関数のソースファイルに#include文を書くときは，

```
#include "/usr/local/src/postgresql-6.3.2/src/include/postgres.h"
```

などとするのではなく，Cコンパイラの-Iオプションを

```
-I /usr/local/src/postgresql-6.3.2/src/include
```

とし，ソースファイル中には，

```
#include "postgres.h"
```

と書くことをお勧めします．さらによい方法は，Makefileを書き，その中でCFLAGSを定義することです．たとえば，

```
SRCDIR= /usr/local/src/postgresql-6.3.2/src
include $(SRCDIR)/Makefile.global
CFLAGS+= $(CFLAGS_SL) -I$(SRCDIR)/include
```

とすることにより，PostgreSQLを構築する際に定義したmakeの各種変数がそのまま利用でき，移植性が高まります．実際，この方法だとLinuxやFreeBSDをはじめ，ほとんどのプラットフォームで修正なしにコンパイルが可能です．

postgres.hをincludeすることにより，さまざまなPostgreSQLのデータ型がCで使えるようになります．SQLのデータ型は，多くの場合同じ名前のCデータ型に対応します．また，4バイト以下の大きさのデータ型の場合にはその値が直接引数として渡されるのに対し，5バイト以上のデータ型はポインタが渡されます．

そのほか，データ型によってはpostgres.h以外にもヘッダファイルをincludeする必要がある場合もあります．表3.4.4にSQLのデータ型とCデータ型の対応を示します．

3行目はelog( )関係のヘッダファイルです．C関数の中ではprintf( )ではなくelog( )を使います．elog( )はメッセージなどを印字するだけでなく，エラー発生時にトランザクションをアボートするなどの重要な役割があります．elog( )の第1引数はelog( )の振る舞いを指定するコードです（表3.4.5）．

第2引数はメッセージです．printf( )と同様のフォーマッティングが可能です．

5行目から8行目はlarge object関係のヘッダファイルです．

●表3.4.4　SQLデータ型とCデータ型

SQLデータ型	Cデータ型	postgres.h以外にincludeする必要のあるヘッダファイル
bool	bool	
box	BOX *	utils/geo_decls.h
bytea	bytea *	
date	DateADT	
datetime	DateTime *	
int2	int2	
int4	int4	
integer	int4	
float	float64 *	
float4	float32 *	
float8	float64 *	
lseg	LSEG *	utils/geo_decls.h
oid	Oid	
path	PATH *	utils/geo_decls.h
point	Point *	utils/geo_decls.h
text	text *	
time	TimeADT *	
timespan	TimeSpan *	

●表3.4.5　elog()の引数

NOTICE	メッセージを表示するだけで，エラー扱いにならない
ERROR	エラー．トランザクションはアボートされる
FATAL	致命的なエラー．バックエンドプロセスを終了させられる
DEBUG	デバッグメッセージの表示．postmasterで-dを指定したときにのみメッセージを表示
NOIND	同じくデバッグメッセージの表示．ただしDEBUGと違ってインデントしない

10行目のbuiltins.hには，今回利用したtextregexeqなどの組み込み関数のプロトタイプ宣言が格納されています．

13～16行目は，今回作成する4つのC関数のプロトタイプ宣言です．これらの関数は，内部関数のcheck（18行目）を呼び出し，check( )が実際の処理を行います．

check( )の第1引数はlarge objectのoidです．第2引数は検索文字列を指定します．第2引数の型struct varlenaはPostgreSQLで可変長のオブジェクトを表現するのに使われている構造体です．ほかにtext，byteaというデータ型もありますが，実際にはどちらもvarlenaと同じものです．

さて，varlenaは以下のような構造を持ちます．

```
struct varlena
{
        int32           vl_len;
        char            vl_dat[1];
};
```

vl_lenにはデータを含むvarlena全体のバイトサイズが登録されます．vl_datは実際のデータが格納されるエリアですが，vl_dat[1]のようにサイズ1の配列になっています．奇妙に見えますが，[1]はダミーなので，実際には必要なメモリをvl_datに確保しなければなりません．これらの構造体メンバは直接扱うことはありません．varlenaを扱うためのマクロが定義されています．ここでpはvarlenaへのポインタです．

- **VARSIZE(p)：vl_lenをアクセス**
- **VARDATA(p)：vl_datをアクセス**
- **VARHDRSZ：varlenaからデータ部を除いた部分の大きさ**

典型的な使い方は以下のようになります．ユーザデータの大きさをlen（バイト）とすると，

```
struct varlena *p;
p = palloc(len + VARHDRSZ);
VARSIZE(p) = len + VARHDRSZ;
strcpy(VARDATA(p),"abc");
```

ここで，palloc( )はPostgreSQLの組み込み関数で，malloc( )と同じようにメモリを獲得する関数です．malloc( )と違い，トランザクションが完了または終了したときにメモリが自動的に解放されます．PostgreSQLでは，さまざまな原因でトランザクションがアボートし，関数の実行が中断されることがあります．このような場合，メモリをfree( )で解放することが非常に難しくなります．palloc( )を使ってメモリを獲得するようにすれば，このような問題を回避することができます．とは言っても，不要になったメモリは可能ならば解放した方がよいのは言うまでもありません．その場合，free( )ではなくてpfree( )という関数を使ってメモリを解放することができます．

26行目のinv_open( )は，large objectを開き，その記述子を返す組み込み関数です．large objectを扱うためには，まずinv_open( )を使って記述子を得る必要があります．inv_open( )の第1引数はlarge objectのオブジェクトIDで，第2引数はlarge objectをどのように扱うかのフラグです．

- INV_READ：読み込みのためにopenする
- INV_WRITE：書き込みのためにopenする

指定したlarge objectが開けなかった場合にはNULLが返ります．

31行目のinv_seek( )は，large objectを“seek”する関数で，UNIXのfseek( )やlseek( )と同様の機能を持つ関数です．large objectは直接その大きさを知ることができないので，inv_seek( )でlarge objectの最後までオフセットを移動し，返り値で返ってくる現在オフセットからlarge objectの大きさを知るわけです．

32行目は同じinv_seek( )を使ってオフセットを先頭に戻しています．

33行目でpalloc( )を使ってlarge objectと同じ大きさのメモリを獲得し，34行目のinv_read( )でlarge objectを読み込み，35行目のinv_close( )で後処理をします．

43行目でvarlena構造体にメモリサイズをセットし，44行目で正規表現を処理する関数を呼び出します．結果が求まればlarge objectを格納するのに使用したメモリは不要ですから，45行目のpfree( )でメモリを解放します．

49行目以降が実際に外部から呼び出される関数群です．

■関数のコンパイルとリンク

次はコンパイルとリンクです．ここは単にmakeまたはgmakeとするだけです．筆者の環境では，図3.4.1のようになります．

makeの結果できたbigtext.soがcreate functionで指定するオブジェクトファイルです．

■create functionの実行

今回のbigtextは，オブジェクトファイルは1個ですが，関数は4個あるので，create functionを4回実行します．添付のファイルbigtext.sqlを実行してください．

筆者の環境での実行例を図3.4.2に示します．

●図3.4.1　筆者の環境にて試したbigtext.cのコンパイル

```
gcc     -I/home/t-ishii/src/6.4/pgsql/src/include     -I/home/t-
ishii/src/6.4/pgsql/src/backend   -O2 -g -Wall -Wmissing-prototypes -fpic
-I/home/t-ishii/src/6.4/pgsql/src/include   -c bigtext.c -o bigtext.o
gcc -shared -o bigtext.so bigtext.o
rm -f bigtext.sql; \
C=`pwd`; \
sed -e "s:_OBJWD_:$C:g" \
    -e "s:_DLSUFFIX_:.so:g" < bigtext.source > bigtext.sql
rm bigtext.o
```

●図3.4.2　create function実行例

```
% psql -f bigtext.sql test
-- regular expression equal
drop function bigtextregexeq(oid,text);
DROP
create function bigtextregexeq(oid,text)
returns bool
as '/home/t-ishii/doc/book/bigtext/bigtext.so'
language 'c';
CREATE

-- regular expression not equal
drop function bigtextregexne(oid,text);
DROP
create function bigtextregexne(oid,text)
returns bool
as '/home/t-ishii/doc/book/bigtext/bigtext.so'
language 'c';
CREATE

-- regular expression equal (case ignore)
drop function bigtexticregexeq(oid,text);
DROP
create function bigtexticregexeq(oid,text)
returns bool
as '/home/t-ishii/doc/book/bigtext/bigtext.so'
language 'c';
CREATE

-- regular expression not equal (case ignore)
drop function bigtexticregexne(oid,text);
DROP
create function bigtexticregexne(oid,text)
returns bool
as '/home/t-ishii/doc/book/bigtext/bigtext.so'
language 'c';
CREATE
```

■bigtextを使ってみる

insert.sh（シェルスクリプト）を実行すると，サンプルのテーブルとテストデータが登録されます．テストデータはtestdataディレクトリにあり，insert.shはそこにある数字の名前のファイルを登録します．

登録されたデータをbigtextを使って検索してみましょう．check.shというというスクリプトを起動してみてください．図3.4.3のように，登録中のデータを表示した後，検索のデモを行います．

このデモでは，データの名前（name）に元のテキストのファイル名，データを識別するidとしてlarge objectのオブジェクトIDを使っています．

デモの表示が終わると，テキストの中身をインタラクティブに表示するフェーズに移ります．

●図 3.4.3　bigtextを使って検索する

```
=== 登録中の全データ ===
name  |   id
------+-----
01.txt|18817
04.txt|18849
05.txt|18881
06.txt|18913
07.txt|18945
08.txt|18977
(6 rows)

=== 「データモデル」を含むデータを検索 ===
01.txt
=== 「入力」を含むデータを検索 ===
07.txt
08.txt
=== 「データモデル」を含まないデータを検索 ===
04.txt
05.txt
06.txt
07.txt
08.txt
=== 「入力」を含まないデータを検索 ===
01.txt
04.txt
05.txt
06.txt
```

「表示したいデータのタイトルを入力してください.」

に対して，タイトル（01.txtなど）を入力すると，large objetを読み込んでその中身を表示します．間違ったタイトルを入力すると，次のように表示されます．

該当データがありません．次のうちから選んでください．

01.txt

04.txt

05.txt

06.txt

07.txt

08.txt

何も入力せずにリターンキーを押すと，このデモを終了します．

■タプルを引数として受け取る関数

3.4.3では“ person ”というテーブル，すなわちpersonというデータ型を引数とし

て受け取るSQL関数の例が出てきましたが，C関数でも同じようにタプルを引数として受け取る関数を作ることができます．

たとえば，personテーブル

```
create table person (id int, name text);
```

を引数として受け取とり，idカラムの値を返す関数はリスト3.4.3のようになります（あまり意味のある例ではありませんが）．

GetAttributeByName( )は，第1引数としてメモリ上の1行文のタプルデータを受け取り，第2引数で与えられたカラム名のカラムのデータを返します．返ってくるデータは大きさが4バイト以内の型の場合はデータの値そのものですが，4バイトよりも大きいデータ型ではデータへのポインタが返ります．第3引数はカラムの値がNULLかどうかを識別するフラグへのポインタです．カラムがNULLなら，0以外の値が設定されます．

create functionでこの関数を登録するときには，以下のような構文になります[注6]．

```
create function get_id_from_person(person)
returns int
as '/foo/bar/get_id_from_person.so'
language 'c';
```

●注6
もちろん/foo/bar/は適当なパスに置き換える必要があります．

このように，引数の型として“person”が使われ，Cでの定義と異なることに注意してください．

ここで取り上げた例題のソースは，examples/get_id_from_person/にあります．

●リスト3.4.3

```
#include "postgres.h"

#include "utils/builtins.h"        /* for builtin types */

#include "executor/executor.h"     /* for GetAttributeByName() */

int4 get_id_from_person(void *);

int4 get_id_from_person(void *t)
{
        int4 id;
        bool idisnull = 0;

        id = (int4)GetAttributeByName(t, "id", &idisnull);
        if (idisnull) {
                return(0);
        }
        return(id);
}
```

3.4.5 functional index

多くのデータベースと同じように，PostgreSQLにも索引（インデックス）があります．索引は，特定のカラムのデータを特殊な形式のファイルにあらかじめ格納しておき，検索を高速化するものです．実用的なデータベースを構築するためは索引はなくてはならないもので，索引があるのとないのとでは数百倍も性能が違うことも珍しくありません．

ところで，前節までで紹介したC関数は強力ですが，普通にSELECT文の中で使うと索引が働かない，という問題があります．たとえば，小文字から大文字に変換する組み込み関数upper( )を使った次のような検索を考えてみましょう．

```
create table paths (path text);
select * from paths where upper(path) = 'FOO';
```

この場合，テーブルのタプルを1つずつ読み出してはupper( )に渡し，大文字に変換した結果を比較するということが行われます．これではテーブルが大きくなると，どんどん処理が遅くなります．この問題は，PostgreSQLの“functional index”という機能を使えば解決できます．

まず，create index文で索引を作ります．

```
create index paths_index on paths using btree
(upper(path) text_ops);
```

“path_index”は索引の名前です．どんな名前でもよいのですが，データベースの中でユニークでなければなりません．“text_ops”は，内部的に索引を扱うタイプ（タイプクラス）を指定しています．関数の型により選択します．“btree”は，索引を管理する手法（アクセスメソッド）にbtreeを指定しています．タイプクラスとアクセスメソッドおよびwhere節などで使われる比較演算子の間には密接な関係があり，どれでもいいわけではありません．たとえばtext型の場合，タイプクラスはtext_opsを選択し，アクセスメソッドについては，

- 比較演算子が=の場合はbtreeまたはhashが選択可能
- 比較演算子が=のほか，>など大小比較も含まれる場合はbtreeのみ選択可能

という制約があります．詳しくはオンラインマニュアル（create_index）をご覧ください．

索引を作成したら，データベースの管理する統計情報を更新するために“vacuum”を実行します．

```
vacuum paths;
```

これで検索はかなり高速になるはずです．速くなっていることを確かめるために，ストップウオッチを使うのも手ですが，PostgreSQLの場合，explainというSQLコマンドでこのことを確かめることができます．

CD-ROMのexamples/upper/にサンプルが用意してありますので，それを使ってみましょう．このサンプルでは，UNIXの/以下のディレクトリを100個，pathsテーブルに格納します．まず，テーブルを作成します．

```
% psql -f upper.sql test
```

次に，findコマンドを使ってディレクトリ名を取得し，テーブルに登録するシェルスクリプトを実行します．

```
% sh insert.sh
```

次に統計情報を更新するため，vacummを実行します．

```
% psql -c "vacuum paths" test
```

まず索引を作成しない状態で，explainコマンドを使ってみます．explainには引数として問い合わせ文字列を渡します（図3.4.4）．

“Seq Scan on paths”は，pathsテーブル全体を読み出しながら検索することを表し

●図 3.4.4　explainコマンド実行例（索引を作成しない場合）

```
% psql test
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: test

test=> explain select path from paths where upper(path) = 'aaa';
NOTICE:  QUERY PLAN:

Seq Scan on paths  (cost=4.30 size=50 width=12)

EXPLAIN
test=>
```

●図3.4.5　functional indexの作成

```
% psql -f create_index.sql test

test=> explain select path from paths where upper(path) = 'aaa';
NOTICE:  QUERY PLAN:

Index Scan using paths_index on paths  (cost=3.67 size=34 width=12)

EXPLAIN
```

ています．すなわち，索引を使わないわけです[注7]．

次にfunctional indexを作成しましょう（図3.4.5）．explainを実行してみると，今度は“Index Scan using paths_index on paths”と表示され，索引が使われることが確認できました．

■functional indexが使えないケース

すべての場合にfunctional indexが使えるわけではありません[注8]．関数の引数にカラム以外のデータが含まれる場合にはfunctional indexを作成することができません．たとえば，foo(int, int)という関数があり，

```
create table bar (i val);
```

というテーブルがあったとします．この場合，foo(5,i)の結果を索引にすることはできませんが，foo(5,i)を呼び出すfoo2(i)を定義すればこの問題を回避することができます．すなわち，

```
select * from bar where foo(5,i) = 10;
```

のような問い合わせを

```
select * from bar where foo2(i) = 10;
```

に置き換えるわけです．

また，

```
select b1.* from bar b1, bar b2 where foo(b1.i,b2.i) = 100;
```

のような形の問い合わせに使われる関数もfunctional indexにできません．

●注7
(cost=4.30　size=50 width=12)のcost = 4.30は，検索コストの評価を数値化したものです．size=50は結果を得るためにスキャンするタプル数の期待値です．今回は全部で100個のタプルがあるので，確率的にその半分の50個のスキャンで探しているタプルが見つかると見ているわけです．width=12は，検索対象のカラムのデータサイズをバイト数で表しています．今回検索対象のpathタプルは可変長なので，12というのは参考程度の数値と思います．

●注8
バグのため，functional indexは本書に付属のパッチが当たってない6.3.2およびそれ以前のバージョンでは使用できません．

3-5 PostgreSQLの問い合わせ言語 ●その3：ユーザ定義オペレータ

3.5.1 オペレータ

オペレータ（演算子）とは，

```
select * from foo where bar = 100;
```

のような問い合わせにおいて＝の部分を指します．このように左右に変数または定数があり，その結果の真偽を返すタイプのオペレータは，バイナリオペレータ（binary operator）と呼ばれます．このほか，1つだけの変数または定数を扱うオペレータがあり，こちらはunary operatorと呼ばれます．

PostgreSQLでは，真偽（bool）を返す1または2引数の関数さえ用意すれば，ユーザ定義のオペレータを自由に作ることができます．ここでは前節で作成したbigtextの関数を使って，バイナリオペレータを定義してみましょう．

bigtextの関数はOIDとtextを引数に取り，

```
select name from big where bigtextregexeq(big.id,'データモデル');
```

のようにして検索に使えるものでした．しかし，関数名が長い[注1]，記法が直観的でないなどから，あまり使いやすいものものではありませんでした．

そこで，オペレータを定義してこれを改善してみましょう．たとえば，

```
select name from big where id ~ 'データモデル';
```

と書けるようにするわけです．以降，その方法を説明していきます．

●注1
センスの問題もありますが，C関数の名前は衝突を避けるために長めになりがちです．

3.5.2　オペレータに割り当てる記号

オペレータとして使う記号は，以下の中から選びます．

```
~ ! @ # % ^ & ` ?
```

オペレータは31文字以内の文字列ですが，2文字以上のオペレータの場合，上記に加えて下記の文字も使えます．

```
| $ : + - * / < > =
```

バイナリオペレータの場合，左右の変数／定数の型とオペレータとして使う文字列の組み合わせがユニークなら，既存のオペレータと同じ文字列も使えます．今回は，同じ正規表現を扱うことでもあり，text用の既存のオペレータと同じ~などを使うことにします（表3.5.1）．

●表 3.5.1　今回定義するオペレータ

オペレータ	関数
~	bigtextregexeq
!~	bigtextregexne
~*	bigtexticregexeq
!~*	bigtexticregexne

3.5.3　create operator

オペレータを定義するには，まずcreate functionを使ってC関数を定義しますが，これはすでに前節で説明しました．ここでは，直ちにcreate operatorを実行します．

create operatorの構文はリスト3.5.1です．引数がたくさんありますが，バイナリオペレータの場合，必須なのはleftarg，rightarg，procedureで，後はオプションです．今回はnegatorも使います．例を挙げて説明しましょう．

オペレータ~にbigtextregexeqを割り当てるSQL文は，以下のようになります．

```
create operator ~ (
       leftarg = oid,
```

●リスト3.5.1　create operatorの構文

```
create operator operator name
     ([ leftarg = type-1 ]
      [ , rightarg = type-2 ]
      , procedure = func name
      [, commutator = com op ]
      [, negator = neg op ]
      [, restrict = res proc ]
      [, hashes]
      [, join = join proc ]
      [, sort = sor op1 {, sor op2 } ]
     )
```

```
        rightarg = text,
        procedure = bigtextregexeq
        );
```

leftargはオペレータの左側のデータ型，rightargは右側の型です．procedureは関数名です．これで

```
select name from big where id ~ 'データモデル';
```

が実行できるようになります．次に，~の否定形である!~ですが，こちらの登録は以下のようになります．

```
create operator !~ (
        leftarg = oid,
        rightarg = text,
        procedure = bigtextregexne,
        negator = ~
        );
```

こちらにはnegatorが指定されています．これは，

```
select name from big where not id !~ 'データモデル';
```

の形の問い合わせを，

```
select name from big where id ~ 'データモデル';
```

とオプティマイザが書き換えられるようにするための指定です[注2]．

　すなわち，negatorには否定の演算子を指定します．とすると，最初の~の定義でもnegatorの指定があってもよさそうなものですが，最初のオペレータを定義するとき

●注2
一般にnot演算は非常に高価なので，このように書き換えができれば効率的です．

にはnegatorを指定せず，否定演算子を定義するときにnegatorを指定することになっています[注3]．

~＊および!~＊に対しても同じ要領でcreate operatorできます．

これらのcreate operator文は付属CD-ROMのexamples/bigtext/operator.sqlにありますので，

```
% psql -f operator.sql test
```

として実行してみてください．

●注3
実際には~に対してそのnegatorが!~であることは，!~の登録時に自動的に登録されるようになっています．

3-6 PostgreSQLの問い合わせ言語 ●その４：ユーザ定義データ型

PostgreSQLの大きな特長として，ユーザが自由にデータ型を定義できることが挙げられます．ここでは，PostreSQLの配列のような機能を持つデータ型を作ってみましょう．

簡単のために，扱えるデータ型はint（int4）に限定し，

```
create table foo (i int4array);
insert into foo values('{1,2,3}');
select * from foo i = '{1,2,3}';
```

のようなことができるような“int4array”という型を作ってみます．

ユーザ定義データ型を作るためには，

①データ型の内部表現をCの構造体を使って表現する
②データを内部表現から外部表現（文字列）に変換する関数を作る
③データを外部表現（文字列）から内部表現に変換する関数を作る
④create typeでデータ型を登録する
⑤オペレータを定義する

という作業が必要です．

3.6.1　C構造体の設計

今回作成するデータ型を次のようにC構造体で表現します．

```
typedef struct {
        int32   sz;     /* この構造体の大きさ */
        int4    n;      /* 配列内の要素数 */
        int4    dt[1];  /* データを格納するエリア．
                           簡単のために，データ型はint4に限る */
} int4array;
```

ユーザ定義データ型は固定長サイズのものと可変長サイズのものに分けられます．今回作成するデータ型は配列ですから後者です．この場合，構造体の先頭にint32の変数を置き，そこにサイズをセットすることになっています（sz）．nには，配列内の要素数をセットします．dtはデータを格納するエリアです．そこに動的にメモリを確保し，データを格納するものとします[注1]．この構造体はint4array.hというファイルに入れておきます．

●注1
今回は簡単のために，データ型はint4に限るものとします．

3.6.2　データを内部表現から外部表現に変換する

内部表現（int4array）から外部表現（文字列）に変換する関数を作ります（リスト3.6.1）．すなわち，int4arrayを受け取ってそれを`char *`で返します．外部表現はPostgreSQLの配列同様，`{1,2,3}`のような表現にします．

8行目は，文字列の長さを計算しています．`20`というのは，数字1個あたりの桁数と数字を区切るカンマを合わせ，余裕をみた値です．`2`は前後の中括弧の分です．

9行目でメモリを獲得した後10行目で開始の中括弧をセットします．11行目からのforループは配列の要素数分だけ回ります．12～13行目で数字を文字列に変換してセットし，14行目でカンマを添付します．

18行目で終了の中括弧をセットします．

作った関数はcreate function文で登録します．

```
create function int4array_out(opaque)
returns int4array
```

●リスト3.6.1　内部表現→外部表現に変換する関数

```
 1: char *int4array_out(int4array *ar)
 2: {
 3: int len;
 4:   char *s;
 5:   int i;
 6:   char buf[128];
 7:
 8:   len = ar->n*20+2;   /* 必要な文字列の長さを計算 */
 9:   s = (char *)palloc(len+1);
10:   sprintf(s,"{");      /* 開始中括弧 */
11:   for (i=0;i<ar->n;i++) {
12:     sprintf(buf,"%d",ar->dt[i]);         /* 数字を文字列に変換 */
13:     strcat(s,buf);
14:     if (ar->n != (i+1)) {
15:       strcat(s,",");  /* 区切りカンマ */
16:     }
17:   }
18:   strcat(s,"}");      /* 閉じ中括弧 */
19:   return(s);
20: }
```

```
as '/home/t-ishii/doc/book/int4array/int4array.so'
language 'c';
```

3.6.3　データを外部表現から内部表現に変換する

これは3.6.2の逆を行うだけです．なお，簡単のため，数字の間などにはスペースを許さず，また入力データのフォーマットチェックはほとんど行っていません（リスト3.6.2）．

これもcreate functionで登録します．

```
create function int4array_in(opaque)
returns opaque
as '/home/t-ishii/doc/book/int4array/int4array.so'
language 'c';
```

●リスト 3.6.2　外部表現→内部表現に変換する関数

```
 1: int4array *int4array_in(char *ar)
 2: {
 3:   char *p;
 4:   char *ar2;
 5:   int cnt = 0;         /* 配列要素数のカウンタ */
 6:   int len;
 7:   int4array *arp;
 8:   char buf[128];
 9:   int i;
10:
11:   if (strcmp(ar,"{}")) {
12:     ar2 = ar;
13:     ar2++;   /* "{" をスキップ */
14:
15:     /* 配列要素数を数える */
16:     while(*ar2) {
17:       if (*ar2++ == ',') cnt++;
18:     }
19:     cnt++;
20:   }
21:
22:   len = (int)&((int4array *)0)->dt + cnt * sizeof(int4);
23:   arp = (int4array *)palloc(len);
24:   arp->sz = len;
25:   arp->n = cnt;
26:
27:   ar++;       /* "{" をスキップ */
28:   i = 0;
29:   while(*ar && *ar != '}') {
30:     p = buf;
31:     while(isdigit(*ar)) {       /* 数字1個分をコピー */
32:       *p++ = *ar++;
33:     }
34:     *p = '\0';
35:     arp->dt[i++] = atoi(buf);  /* 数字を格納 */
36:     ar++;
37:   }
38:   return(arp);
39: }
```

3.6.4　create type でデータ型を登録する

create type 文の構文はリスト 3.6.3 です.
今回の場合，このようになります.

```
create type int4array (
   internallength = variable,
   input = int4array_in,
```

●リスト3.6.3　create type文の構文

```
CREATE TYPE typename (
INTERNALLENGTH = (number|VARIABLE),
[EXTERNALLENGTH = (number|VARIABLE),]
INPUT = input_function, OUTPUT = output_function
[,ELEMENT = typename][,DELIMITER = character][,DEFAULT='<string>']
[,SEND = send_function][,RECEIVE = receive_function][,PASSEDBYVALUE]
[,ALIGNMENT = ]);
```

```
    output = int4array_out,
    alignment = double
);
```

internal lengthは固定長の場合はその長さ，可変長の場合は“variable”です．input/outputはcreate functionで登録した関数名，最後のalignmentはメモリ上でこのデータ型の先頭をどのようなバウンダリに合わせるかの指定です．doubleは8バイトバウンダリになります．デフォルトはintで4バイトバウンダリです．今回の場合，4バイトでもいいと思いますが，ためしにdoubleを使ってみました．なお，float8のようなデータを使っている場合には，必ずdoubleにする必要があります．

このほか，今回は使ってませんが，4バイト以下のデータ型の場合は，passedbyvalueを指定します．

これ以外のオプションについては，オンラインマニュアルのcreate_type(l)をご覧ください．

3.6.5　オペレータを定義する

ここまででデータ型の登録が完了し，テーブル定義，データの入力，表示ができるようになりました．

```
create table t1 (i int4array);
insert into t1 values('{1,2,3}');
INSERT 144816 1
select * from t1;
i
-------
{1,2,3}
```

ただしこのままでは，

```
select * from t1 where i = '{1,2,3}'::int4array;
```

のような問い合わせができません．int4array型で使える＝オペレータが存在していないからです．そこで，最低限のものとして，＝とその否定の！＝を定義してみましょう．

オペレータの定義の仕方については，3.5節で説明したので，さっそく関数を作ることにします．

まず＝オペレータ用として，int4arayeqという関数を用意します．

```
bool int4arrayeq(int4array *a1, int4array *a2) /* = オペレータ用 */
{
  int i;
  if (a1->sz != a2->sz) return(false);
  if (a1->n != a2->n) return(false);
  for (i=0;i<a1->n;i++) {
    if (a1->dt[i] != a2->dt[i]) return(false);
  }
  return(true);
}
```

ここでは，配列のサイズ，要素数，配列要素のすべてが一致しているときに等しいと見なすようにしています．もう少し制限を緩めて，要素の並びが異っていても対応する要素があればOKということもあり得ます．

```
{1,2,3} = {1,3,2}
```

あるいは，これを別なオペレータにすることもできるでしょう．このあたりは作り方しだいでどのようにでもできるのがPostgreSQLのよいところです．

関数ができたら，create functionとcreate operatorを使って登録します．

```
create function int4arrayeq(int4array,int4array)
returns bool
as '/home/t-ishii/doc/book/int4array/int4array.so'
language 'c';

create operator = (
       leftarg = int4array,
```

```
        rightarg = int4array,
        commutator = =,
        procedure = int4arrayeq
        );
```

なお，as '/home/t-ishii/…以降は実際にはint4array.soのあるパス名によって異なります（以下同様）．

次に!=を作ります．こちらは非常に簡単で，int4arrayeqの戻りを逆にしているだけです．

```
bool int4arrayne(int4array *a1, int4array *a2) /* != オペレータ用 */
{
  if (int4arrayeq(a1, a2) == true)
    return(false);
  return(true);
}
```

create functionとcreate operatorで登録します．

```
create function int4arrayne(int4array,int4array)
returns bool
as '/home/t-ishii/doc/book/int4array/int4array.so'
language 'c';

create operator != (
       leftarg = int4array,
       rightarg = int4array,
       procedure = int4arrayne,
       commutator = != ,
       negator = =
       );
```

テストしてみましょう（図3.6.1）．ちゃんと動いているようですね．なお，6.4では型の自動変換が行われるので，

```
select * from t1 where i != '{1,2,3}';
```

でもOKです．

●図3.6.1　作成した関数のテスト

```
select * from t1;
i
-------
{1,2,3}
{100}
{}
(3 rows)

select * from t1 where i = '{1,2,3}'::int4array;
i
-------
{1,2,3}
(1 row)

select * from t1 where i != '{1,2,3}'::int4array;
i
-----
{100}
{}
(2 rows)
```

今回紹介したint4arrayのソースは，添付のCD-ROMのexamples/int4array/にあります．Makefileの1行目を適当に変更し，

```
make（またはgmake）
```

でint4array.soとint4array.sqlができるので，

```
psql -f int4array.sql test
```

としてみてください．

Cのデバッグを簡単にする？

Ｃ関数のデバッグはなかなかやっかいなものです．このようにダイナミックにロードされる関数ではデバッガの利用が困難なので，なおさらです．

これを解決する方法のひとつは，palloc/pfreeなどのPostgreSQLの組み込み関数をダミーで置き換え，単体でデバッグできるようにすることです．

int4arrayの場合，リストＡのようなメインを作りました．ご覧のように，palloc/pfreeをmalloc/freeに置き換えているだけですが，これだけで単体で動かせるようになります．

●リストＡ

```
int main()
{
  int4array *ar;

  ar = int4array_in("{1,2,3}");
  printf("%s\n",int4array_out(ar));
  return(0);
}

void *palloc(Size size)
{
  return(malloc(size));
}

void pfree(void *pointer)
{
  free(pointer);
}
```