rev 03.08.11

PostgreSQL 解析資料

～ 初期 DB の作成処理 (Bootstrap) ～

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

---

1. はじめに

本資料では、PostgreSQLのインストール後、初期データベース(テンプレートデータベース)を作成する部分の処理について紹介する。初期DB作成の処理は、主にBootstrapモジュール[1]で実装されており、この部分はSQLではない独自の言語インタフェースを使用している。Bootstrap モジュールで読み込むファイルは、ソースコードのヘッダファイルからソースコードのコンパイル時に生成されており、ソースコード中のOIDの定義とシステムテーブル内のOIDを一致させる役目がある。

1.1. 本資料について

本資料は、PostgreSQL7.3.3 を対象にソースコードの調査を行ったものである。従って、他のバージョンでは、内容が異なる場合があるので注意して頂きたい。また、あまり十分な時間をかけず、さっとソースを眺めるレベルで資料を作成しているため、内容に誤りがあった場合はご容赦頂きたい。それでも、基本的な流れを押さえることはできると思うので、ソースコード解析の参考になれば幸いである。

1.2. PostgreSQL のソースを読む方へ

Bootstrap モジュールは、独自の言語インタフェースで動くので、SQL の処理と一見無関係のようだが、構文解析後に使用される内部関数は、SQL の処理に使用するものと全く同じ関数を呼び出している。また、SQL より非常に簡単な言語になっているため、ソースコードを読みはじめる際の入門としてお勧めできる。特に、字句解析、構文解析、データ挿入の流れを理解するのに役に立つだろう。

2. 動作の概要

PostgreSQLをコンパイルしてインストールした場合、インストール後にinitdb スクリプトを実行する必要がある。initdbは、ＤＢクラスタ[2]の作成、ＤＢインスタンスの初期化といった処理が行われる。
PostgreSQLでは、initdb実行時にテンプレートとなるDBを作成しておき、通常のDB作成時(createdb実行時)には、テンプレートデータベースをディレクトリごとコピーしている。
本資料では、３章で initdb スクリプトの動作を説明し、４章でテンプレートデータベースの基本部分を作成する postgres –boot を実現している Bootstrap モジュールについて説明する。

---

[1] src/backend/bootstrap ディレクトリ以下にあるファイル
[2] PostgreSQL の場合、$PGDATA ディレクトリ以下に生成されるデータベースを構成するファイルのこと。

Copyright © 2003 NTTDATA

3. initdb スクリプト

initdb スクリプトの処理概要は次のようになっている。

1. データ用に必要なディレクトリを作成する
2. PG_VERSION ファイルにバージョンを書き込む
3. テンプレートデータベース templete1 を作成する。
   template1 の作成は、postgresコマンドを使用して、次のように実行されている。[3]

```
cat postgres.bki | postgres -boot -x1 -F -D$PGDATA template1
```

4. Conf ファイルの作成 (sample ファイルをコピーするだけ)
5. VIEW の作成 他 (CREATE VIEWS and other things)
6. loading pg_description...
7. template1 データベースから template0 を作成する。

initdb スクリプトは、次のようにして実行する。

```
postgres$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data --no-locale --encoding=EUC_JP
```

PostgreSQL 7.3 より前は、コンパイル時にロケールの機能を使わないようにしていた。7.3 から完全にソースに統合されてコンパイル時に外すことができないため、initdb 時に、明示的にロケール機能を使わないように設定しなければならない(--no-locale オプション)。ロケール機能を使わないようにするのは、日本語環境でしばしば問題[4]を引き起こすためである。

3.1. ディレクトリの構成

$PGDATA 配下のディレクトリは、全て initdb スクリプトから作成されている。
ところどころのディレクトリにあるバージョンファイル PG_VERSION も、全て initdb スクリプトから書き込まれている。
スクリプト中で作成されるディレクトリは次の 5 つである。

```
$PGDATA  $PGDATA/base  $PGDATA/global  $PGDATA/pg_xlog  $PGDATA/pg_clog
```

3.2. initdb 中の関数

exit_nicely()

exit_nicely()は、エラーなどで終了する際に、それまでに initdb で作成したディレクトリなどを削除して終了するためのエラー処理関数である。

pg_getlocale()

pg_getlocale()は、第１引数で指定した変数のロケールに使用するべき値を返す。優先順位は、PGLC_<変数名>, PGLOCALE, LC_ALL, LC_<.変数名> の順である。例えば、CTYPE の場合、PGLC_CTYEP,

---

[3] 環境変数の設定部分は省略している
[4] ソートの順序やインデックスを使う際の性能の問題が知られている。

Copyright © 2003 NTTDATA

PGLOCALE, LC_ALL, LC_CTYPE を順に参照し、最初に設定されているものが選択される。

3.3. initdb (Shell スクリプト) の詳細

実際に initdb のスクリプトで実行されるものを順に説明していく。

・パスの確認

・実行時のユーザＩＤチェック

・引数のチェックと設定

　引数の値を変数に設定する。誤った引数が渡された場合エラーを表示して終了する。

・Usage の表示

　--help オプションがあった場合に、詳しい Usage を表示して終了する。

・ENCODING の確認と ENCODINGID の設定

　引数で指定されたデータのエンコード(言語)が設定された場合、正しいものか確認し、正しいものであれば ENCORDINGID を設定する。

・PGDATA が設定されていることを確認

　環境変数 $PGDATA が設定されているか、 または、-D オプションで位置が与えられている場合以外はエラーで終了する。

・BKI ファイル[5]とコンフィグファイルの確認

　BKI ファイルおよびコンフィグファイルの初期ファイルがあるか確認する。

　また、バージョンの確認も行う

　BKI ファイルは、ソースコード中の src/backend/catalog/genbki.sh でコンパイル時に生成される。

・PGDATA ディレクトリが空であることの確認

　空でなければエラーで終了

・ディレクトリの作成

　- mkdir "$PGDATA

　- mkdir "$PGDATA"/base

　- mkdir "$PGDATA"/global

　- mkdir "$PGDATA"/pg_xlog

　- mkdir "$PGDATA"/pg_clog

・postgres コマンドを BOOTSTRAP MODE (-boot) で実行して template1 を作成する。

　- mkdir "$PGDATA"/base/1

　- PG_VERSION ファイルにバージョンを書き込む

```
echo "$short_version" > "$PGDATA/PG_VERSION"
echo "$short_version" > "$PGDATA/base/1/PG_VERSION"
```

　- postgres コマンドを実行して DB を作成する

```
cat "$POSTGRES_BKI" ¥
| sed -e "s/POSTGRES/$POSTGRES_SUPERUSERNAME/g" -e "s/ENCODING/$MULTIBYTEID/g" ¥
```

[5] postgres.bki ファイルのこと。Bootstrap 時に使用する、専用の言語で書かれている。

Copyright © 2003 NTTDATA

```
| "$PGPATH"/postgres -boot -x1 $PGSQL_OPT $BACKEND_TALK_ARG template1 ¥
|| exit_nicely
```

※ LANG 系の設定部分は省略

つまり、基本的には postgres -boot -x1 -F -D$PGDATA template1 くらいのオプションで実行されている。あとは、引数によって、若干異なる。

x1 ==> BS_XLOG_BOOTSTRAP

F ==> fsync 省略オプション

・Conf ファイルの作成

pg_hba.conf、pg_ident.conf、postgresql.conf をサンプルからコピーすることで作成する。

ここ以降、VIEW の作成 他 (CREATE VIEWS and other things)

・pg_shadow、pg_group にトリガを作成することで、pg_shadow を直接更新しても、password/group ファイルや pg_pwd、pg_group テーブルが更新されるようにする。

```
CREATE TRIGGER pg_sync_pg_pwd AFTER INSERT OR UPDATE OR DELETE ON pg_shadow ¥
      FOR EACH ROW EXECUTE PROCEDURE update_pg_pwd_and_pg_group();
CREATE TRIGGER pg_sync_pg_group AFTER INSERT OR UPDATE OR DELETE ON pg_group ¥
      FOR EACH ROW EXECUTE PROCEDURE update_pg_pwd_and_pg_group();
```

・パスワードの設定

パスワード設定のオプションがあった場合、DB 管理者ユーザのパスワードを設定する。

```
ALTER USER "$POSTGRES_SUPERUSERNAME" WITH PASSWORD '$FirstPw';
```

・次のテーブルを TOAST にすることで、行数の限界をなくす。

pg_attrdef, pg_constraint, pg_database, pg_description, pg_group, pg_proc, pg_rewrite, pg_shadow, pg_statistic

・依存関係のテーブルを構築

次のテーブルのエントリとテーブルの oid を関連付けて pg_depend テーブルに保存する。

pg_class, pg_proc, pg_type, pg_cast, pg_constraint, pg_attrdef, pg_language, pg_operator, pg_opclass, pg_rewrite, pg_trigger, pg_namespace

※ pg_namespace については、pg で始まるシステム用の名前空間に限って登録する。

・次の system view を作成する。

pg_user, pg_rules, pg_views, pg_tables,pg_indexes, pg_stats, pg_stat_all_tables, pg_stat_sys_tables, pg_stat_user_tables, pg_statio_all_tables, pg_statio_sys_tables, pg_statio_user_tables, pg_stat_all_indexes, pg_stat_sys_indexes, pg_stat_user_indexes, pg_statio_all_indexes, pg_statio_sys_indexes, pg_statio_user_indexes, pg_statio_all_sequences, pg_statio_sys_sequences, pg_statio_user_sequences, pg_stat_activity, pg_stat_database, pg_locks, pg_settings, pg_settings_u, pg_settings_n

Copyright © 2003 NTTDATA

※ PostgreSQL7.3 よりスキーマの概念が導入されたので、pg_catalog と pg_toast の２つのスキーマを sys 系のテーブル、それに属さないもの(つまり public)を user 系のテーブルとして区別している。

・pg_description テーブルの読み込み

postgres.description ファイルの内容を一時テーブル tmp_pg_description に流し込む。
pg_class と tmp_pg_description を JOIN して pg_description テーブルを生成する。
※ postgres.description ファイルは、コンパイル時に生成される。

・pg_conversion (文字コード変換)の生成とデータ変換支援関数の作成

conversion_create.sql ファイルから DROP CONVERSION の行を除いたものを実行する。ここで文字コード変換の FUNCTION の定義と、スキーマ pg_catalog に対してデフォルトの文字コード変換が定義される。
※ conversion_create.sql ファイルは、コンパイル時に次の位置に作成される。
src/backend/utils/mb/conversion_procs/conversion_create.sql

・既に出来上がっているオブジェクトのいくつかに対して、特権モードの設定を行う

```
UPDATE pg_class SET relacl = '{"=r"}' WHERE relkind IN ('r', 'v', 'S') AND relacl IS NULL;
UPDATE pg_proc SET proacl = '{"=X"}' WHERE proacl IS NULL;
UPDATE pg_language SET lanacl = '{"=U"}' WHERE lanpltrusted;
UPDATE pg_language SET lanacl = '{"="}' WHERE NOT lanpltrusted;
```

・template1 データベースに対して、VACUUM を実行する。

postgres コマンドを使用して、ANALYZE と VACUUM FULL FREEZE; を実行する。

・template0 データベースを作成する。

CREATE DATABASE template0; を実行することで、template1 データベースから template0 にコピーを行う。CREATE DATABASE を実行すると template1 データベースをコピーする機能を利用しているため、単に CREATE DATABASE を実行している。
template0 データベース構築後に、テンプレートであることを示すフラグと更新が出来ないようにするフラグを設定する。

```
UPDATE pg_database SET datistemplate = 't', datallowconn = 'f' WHERE datname = 'template0';
```

・template0 を使って、最後のシステム OID を決定する。

```
UPDATE pg_database SET datlastsysoid = ¥
            (SELECT oid - 1 FROM pg_database WHERE datname = 'template0');
```

・pg_database に再度 VACUUM をかける

前の VACUUM 後に pg_database に対して変更がかかっているので、pg_database テーブルにのみ VACUUM をかける。

Copyright © 2003 NTTDATA

3.4. template1 vs template0

データベース作成する、CREATE DATABASE コマンドは、デフォルトで template1 データベースをテンプレートに使用する。従って、そのＤＢクラスタでデフォルトとして使用したい設定を template1 データベースに反映しておくことで、データベースを作成するたびに毎回設定する必要がなくなる。

テンプレートデータベース template0 は、initdb 実行時の初期状態のテンプレートで、作成後変更を加えることは許されていない。従って、template1 に手を加えている場合に、PostgreSQL オリジナルの状態のデータベースを作成したい時は、CREATE DATABASE の実行時に template0 をテンプレートとして指定する。

Copyright © 2003 NTTDATA

4. Bootstrap モジュール

initdb スクリプトからテンプレートデータベース template1 を作成するために、postgres コマンドが –boot オプション付きで呼び出される。src/backend/bootstrap ディレクトリ以下にあるのが、Bootstrap モード(postgres -boot)で実行される際のコードである。main()から BootstrapMain() <src/backend/bootstrap.c> が実行される。

-boot 付で postgres コマンドが呼び出された場合、次のような処理を行う。

1. 各モジュールの初期化 (postgres が動くための初期化)
2. LOG 関係のファイルの初期化
3. 標準入力から入ってくる文 (postgres.bki) の実行

```
# PostgreSQL 7.3
# 1 "/tmp/genbkitmp.c"
create bootstrap pg_proc
 (
 proname = name ,
 pronamespace = oid ,
 proowner = int4 ,
 prolang = oid ,
 proisagg = bool ,
 prosecdef = bool ,
 proisstrict = bool ,
 proretset = bool ,
 provolatile = char ,
 pronargs = int2 ,
 prorettype = oid ,
 proargtypes = oidvector ,
 prosrc = text ,
 probin = bytea ,
 proacl = aclitem[]
 )
insert OID = 1242 ( boolin 11 1 12 f f t f i 1 16 "2275" boolin - _null_ )
insert OID = 1243 ( boolout 11 1 12 f f t f i 1 2275 "16" boolout - _null_ )
insert OID = 1244 ( byteain 11 1 12 f f t f i 1 17 "2275" byteain - _null_ )
insert OID = 31 ( byteaout 11 1 12 f f t f i 1 2275 "17" byteaout - _null_ )
insert OID = 1245 ( charin 11 1 12 f f t f i 1 18 "2275" charin - _null_ )
insert OID = 33 ( charout 11 1 12 f f t f i 1 2275 "18" charout - _null_ )
insert OID = 34 ( namein 11 1 12 f f t f i 1 19 "2275" namein - _null_ )
insert OID = 35 ( nameout 11 1 12 f f t f i 1 2275 "19" nameout - _null_ )
insert OID = 38 ( int2in 11 1 12 f f t f i 1 21 "2275" int2in - _null_ )
insert OID = 39 ( int2out 11 1 12 f f t f i 1 2275 "21" int2out - _null_ )
     :
     :
   <中略>
     :
     :
declare unique index pg_trigger_tgrelid_tgname_index on pg_trigger using btree(tgrelid oid_ops, tgname
name_ops)
declare unique index pg_trigger_oid_index on pg_trigger using btree(oid oid_ops)
declare unique index pg_type_oid_index on pg_type using btree(oid oid_ops)
declare unique index pg_type_typname_nsp_index on pg_type using btree(typname name_ops, typnamespace
oid_ops)
build indices
```

postgres.bki の一部

Copyright © 2003 NTTDATA

4.1. BootstrapMain() からの処理

BootstrapMain()から実行される主な処理は次のような流れである。

- ストレージやバッファのマネージャの初期化
- ログ関連ファイルの初期化を呼び出す
  (global/pg_control, pg_xlog/0000000000000000, pg_clog/0000 ファイルが生成される)
- 入力から BKI ファイルを読み込んで、処理を実行する。(Int_yyparse(); をコールする。)

Int_yyparse() を実行すると bootscanner.l で定義されたスキャナ（字句解析器）を使って BKI ファイルの文字が読み込まれ、bootparser.y で定義されたパーザ（構文解析器）が実行される。bootparser.y 中から、構文に合わせて、テーブルを作成する関数やデータ投入の関数などが呼び出される。
bootparser.y から呼び出される関数は、直接通常の SQL を処理するときに呼び出されるものと、bootstrap.c で定義されている bootparse.y を補助する関数がある。bootparser.y を補助する関数では、構文解析の補助を行いながらデータ構造を生成するものと、生成されたデータ構造を利用して通常の SQL を処理する関数を呼び出すものがある。BKI ファイルの処理も、最終的には通常の SQL を処理に使われる内部関数を呼び出している。
実際にどのような関数が呼び出されているかは、後述の「言語仕様と実装」のそれぞれの内部動作を参考にして欲しい。しかし、この資料では、bootstrap.c 以外の関数については関数のエントリポイントまでしか紹介しない。それ以上の内容は、今後の別のドキュメントの範囲とする。

4.2. 言語仕様と実装

Bootstrap 時に使用される BKI ファイルの構文は、open, close, create, insert, declare index, declare unique index, build indices の７個である。
内部的に各文の中で do_start(), do_end() が呼ばれている。do_start(), do_end()の中では、それぞれ StartTransactionCommand(true);、CommitTransactionCommand(true); が実行されているので、1 行ずつトランザクションはコミットされることになる。
データ挿入の構文シーケンスは、リレーションを open して、insert を発行する。データ投入が終わったら close を行う。テーブル作成は、create で行うが、create を行うと open されている状態になっているので、そのまま続けて insert を記述することができる。
インデックス作成のシーケンスは、declare index または declare unique index でインデックスを定義して、最後に build indices を実行することで実際のインデックスの作成が行われる。

4.2.1. open

テーブルをオープンする。

構文：　open <テーブル名>

内部動作：

　　boot_openrel(LexIDStr($2)); <bootstrap/bootstrap.c> を実行する。

4.2.2. close

テーブルをクローズする。

構文：　close [<テーブル名>]

Copyright © 2003 NTTDATA

内部動作：

引数があれば closerel(LexIDStr($2)); <bootstrap/bootstrap.c>、引数が省略されている場合は closerel(NULL); を実行する。

4.2.3. create

テーブルの作成を行う。

構文： create [bootstrap] [shared_relation] [without_oids] <テーブル名> (
<属性名>=<型の名称> [<属性名>=<型の名称> …])

内部動作：

まず、DefineAttr(LexIDStr($1),LexIDStr($3),numattr-1); で属性を登録する。これは、属性の定義の数だけ実行される。

次に、CreateTupleDesc(numattr, !($4), attrtypes) <access/common/tupdesc.c>を実行して、タプルデスクリプタを生成する。続いて、heap_create() <catalog/heap.c> または、heap_create_with_catalog () <catalog/heap.c> を実行して、実際のテーブルを作成する。create 文に bootstrap オプションが指定されていた場合に、heap_create_with_catalog () を実行する。

4.2.4. insert

データの挿入を行う。

構文： insert [oid=<OID>] (<データ> [<データ> …])

※ <データ>は、カンマ区切りでもスペース区切りでもよい

※ <データ>には、定数、文字列、_null_ で NULL 値を代入することもできる。

内部動作：

oid=<OID> を見て $2 に OID が設定される。 oid=<OID> の指定がない場合は 0 になる。

<データ> の数だけInsertOneValue(LexIDStr($1) , num_columns_read++); を実行する。それで、values[] 配列にデータを生成していく[6]。

<データ> の登録が終わったら、InsertOneTuple($2); で 1 行のインサート処理を行う。

4.2.5. declare index

インデックスの作成の定義を行う。実際に index が作成されるのは、build indices が呼ばれてからである。

構文： declare index <インデックス名> on <リレーション名> using <インデックスの種類>(<属性名> [, <属性名>])

内部動作：

属性名を読みながら、makeList1($1)、lappend($1, $3); で属性のリストを作成する。

DefineIndex() <commands/indexcmds.c> で、インデックスの定義情報を作成する。

4.2.6. declare unique index

ユニークインデックスの定義を行う。実際に index が作成されるのは、build indices が呼ばれてからである。

構文： declare unique index <インデックス名> on <リレーション名> using <インデックスの種類>(<属

[6] 具体的なデータの作り方は、５章のデータ構造の説明を参照。

Copyright © 2003 NTTDATA

性名> [, <属性名>])

内部動作：

declare index と同じ。DefineIndex() 呼び出し時に、第 5 引数が true になっているだけ。

4.2.7. build indices

構文： build indices

内部動作：

build_indices() < bootstrap/bootstrap.c> を実行するだけ。これを実行すると定義しておいた index 全部を一度に作成する。

Copyright © 2003 NTTDATA

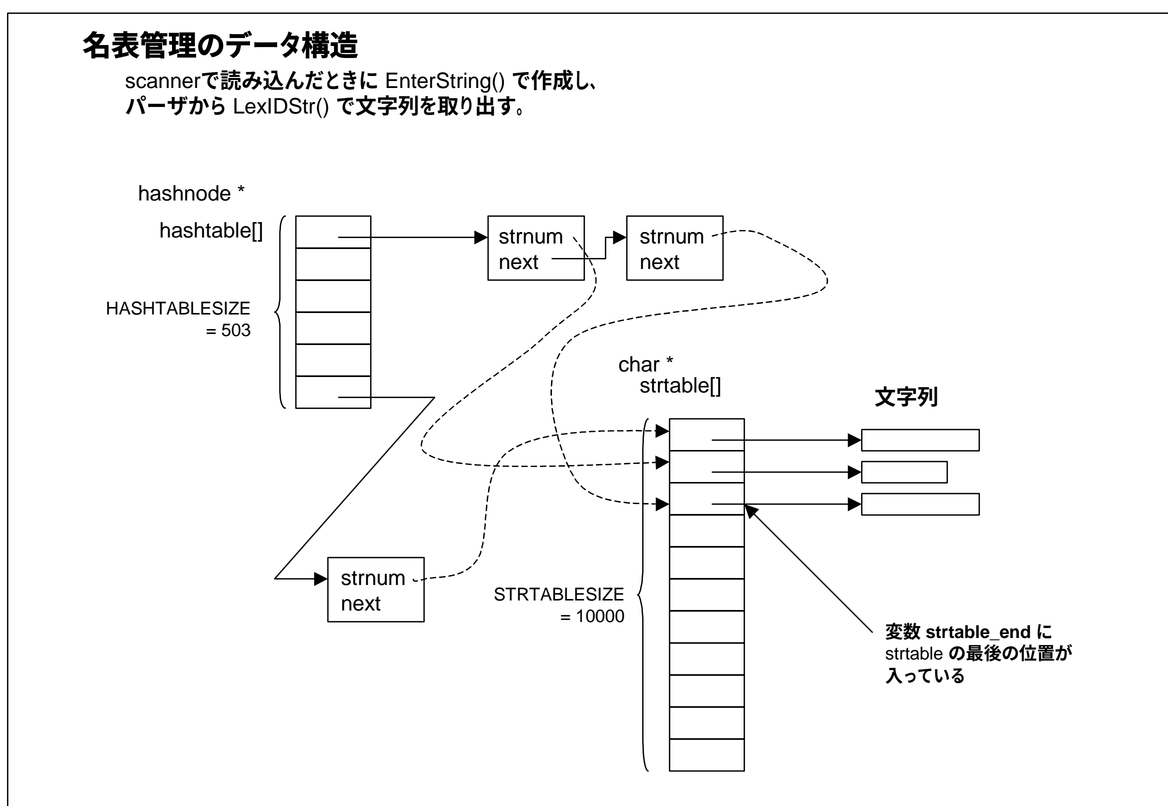
5. Bootstrap.c で使用するデータ構造

ここでは、Bootstrap.c の中のコードで使用されているデータ構造をそれぞれの利用用途ごとに説明する。

5.1. 名表管理

名表は、スキャナで予約語以外の文字列[7]を読み込んだ際に、その文字列をテーブルに登録し、IDを付与する機能である。スキャナからパーザへは、そのIDが渡される。パーザで構文解析を終えて、実際の処理を行う際には、名表から実際の文字列を取り出して、処理を行う。

名前を格納するテーブル strtable[] ポインタ配列は、文字列の検索を高速化するためにハッシュを作成して管理されている。



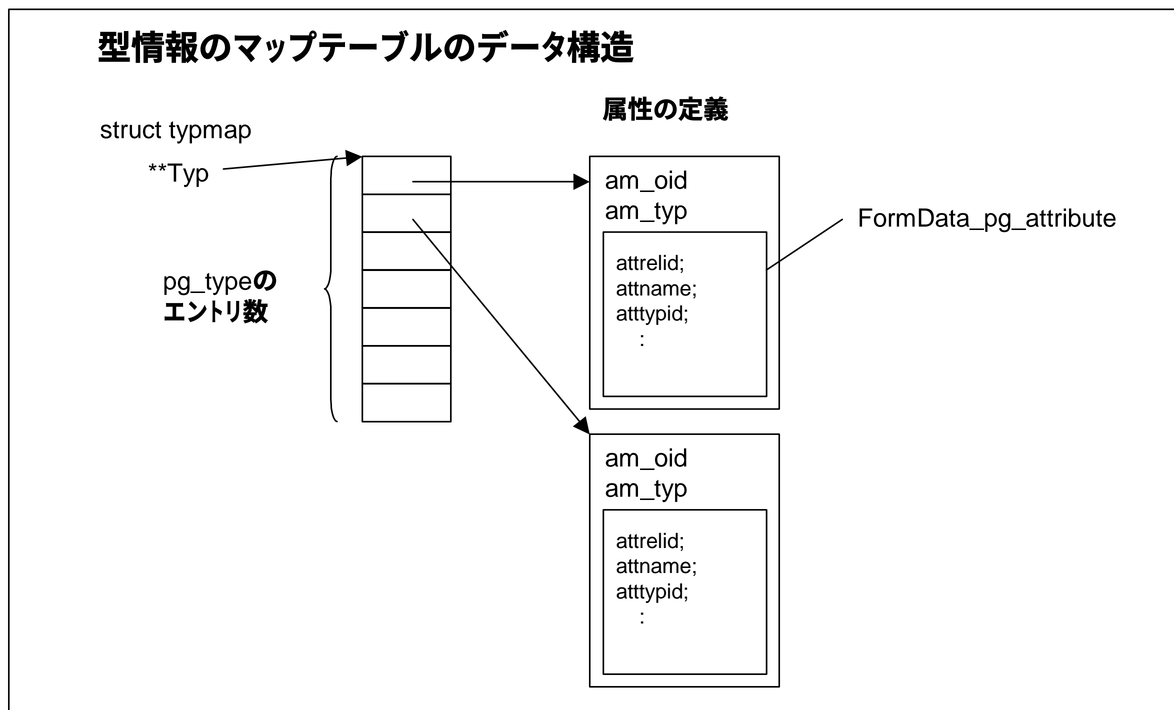
スキャナが使用する EnterString()では、まず、ハッシュテーブルを検索し、引数で与えられた文字列を探す。見つかった場合は、その ID を返す。引き数で与えられた文字列が見つからなかった場合、ハッシュ、および文字列テーブルに新しい文字列を登録し、そのときに付与したＩＤを返す。
パーザで使用される LexIDStr() は、引数に与えた ID の文字列を返す。

[7] 主にテーブル名、属性名、関数名などの名称や属性値である。PostgreSQLの場合、属性の型をユーザ定義できるので、型名も予約語になっているのではなく、名表に登録される。

Copyright © 2003 NTTDATA

5.2. 型情報のマップテーブル

PostgreSQL では、使用できるデータ型を pg_type テーブルに格納して管理している。Bootstrap モジュールでは、この型情報を Typ ポインタ配列を利用してキャッシュしている。



Typ マップテーブルは、boot_openrel() か gettype() 実行時に pg_type テーブルから値を読み込む。ただし、gettype() では、Typ マップテーブルにまだ値が読み込まれていない場合、固定配列 Procid[] を検索し、それでも見つからない場合に限り、pg_type テーブルを読みに行く。

postgres.bki ファイルを処理する際に、実際に Typ マップテーブルに pg_type を読み込むのは、pg_class の float4 の gettype() を行うタイミングくらいだと予想している(未確認)。pg_type を作成したあとであれば、どこであっても問題はない。ちなみに、pg_type が作成されるのは、pg_proc に続いて 2 番目である。

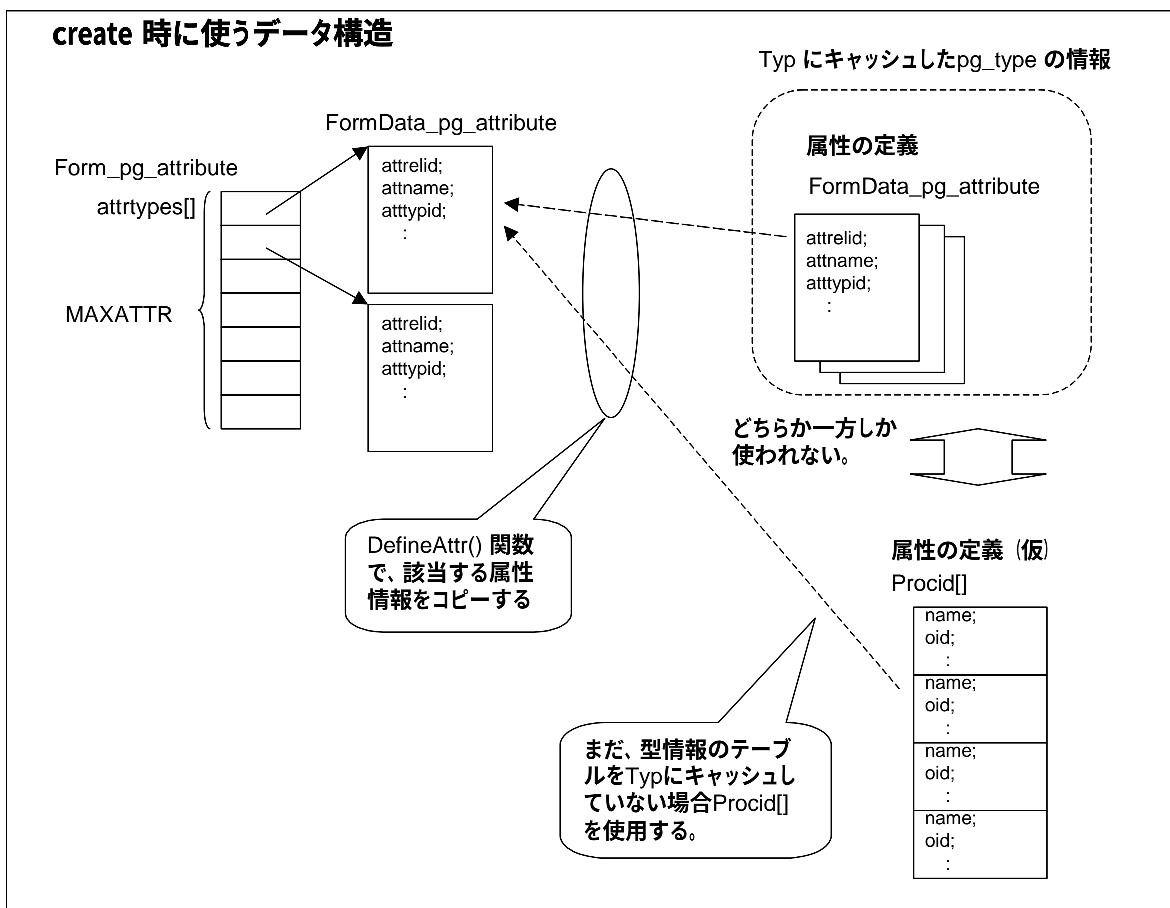
create 文を実行する際に、このFormData_pg_attribute[8] の部分を参照する。

[8] pg_attributeテーブルに対応した構造体。include/catalog/pg_attribute.h に定義されている。

Copyright © 2003 NTTDATA

5.3. create

create 文を実行するときに使用するのが次のデータ構造である。create 文のカラム名とそのデータ型の定義の際に、attrtypes[] 配列に属性の型情報を登録していく。



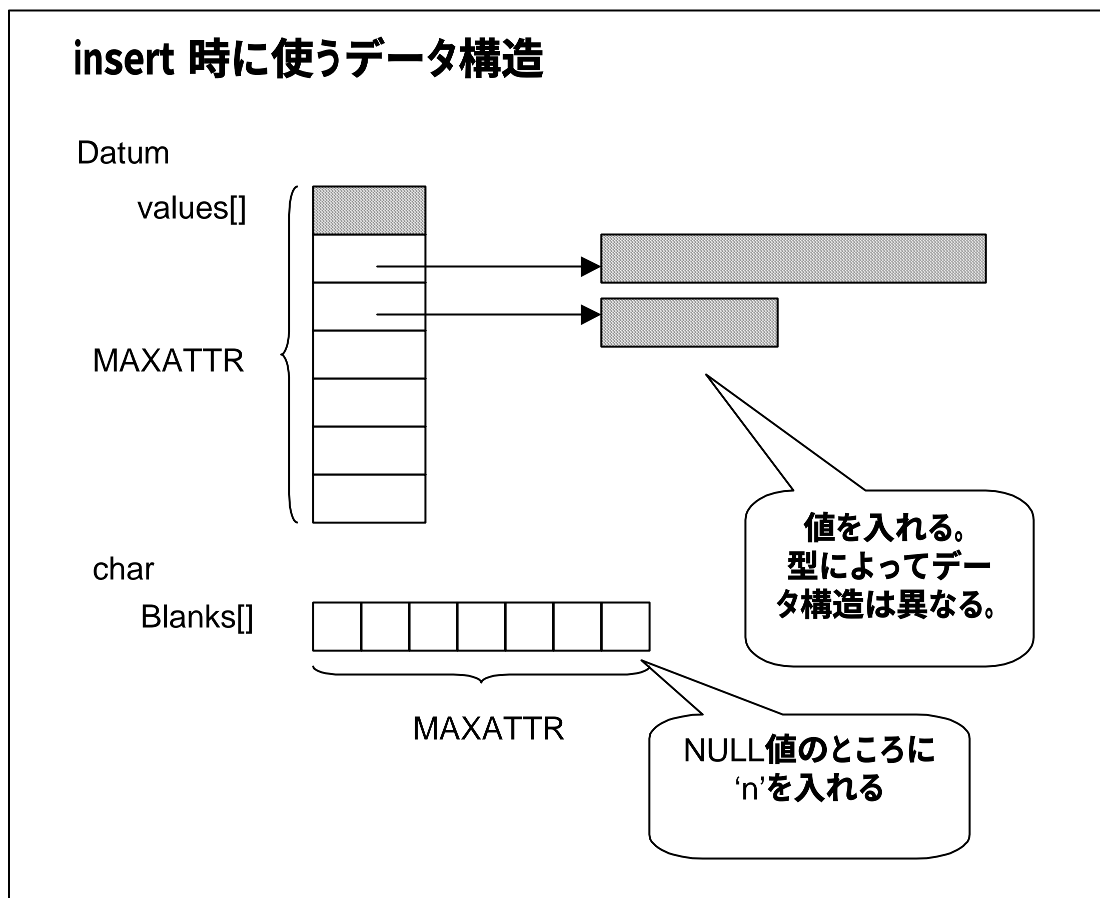
特徴があるのは、Typ マップテーブルに pg_type の情報をキャッシュしていれば、Typ マップテーブルの情報を使用し、まだ、キャッシュしていなければ、固定配列 Procid[] からデータを読み出す。

attrtypes[] 配列のエントリー数は、MAXATTR 個(つまり 40 個)で固定になっている。これは、Bootstrap 用のモジュールなので、Bootstrap 時に登録するテーブルの上限を満たしていれば十分である。従って、可変長サイズにはなっていない。SQL のインタフェースを使用する場合は、もっと多くの続整数を持つテーブルを作成できる。

Copyright © 2003 NTTDATA

5.4. insert

insert 文実行の際に使用するデータ構造が、次の構造である。



Datum 型の values[] 配列は、属性の値を保存する配列である。Datum では、属性の種類によってその場所に値が入っている場合と、ポインタとして読み替えてその先にオブジェクトがつながれている場合がある。その属性値が NULL の場合、Blanks[] 配列に 'n' を入れる。属性に値が入っている場合、Blanks[] 配列のそのエントリは、空白(ホワイトスペース)である。

5.5. Datum について

insert に使用した Datum 型の使い方について、もう少し詳しく触れておく。
データベースでは、属性の値は属性の型によってフォーマットが異なる。これをプログラム中で抽象的に扱う型(C 言語のマクロ)が Datum である。Datum は、unsigned long で定義されている。Datum は、その中に直接データを格納する場合と、Datum にポインタを保存しておいて別のメモリエリアにデータを格納する場合がある。

Datum に値を格納するためのメソッドとして inproc が定義してあり、Datum から文字列として値を取り出すメソッドとして outproc が定義される。この入出力のメソッドのうち、メソッド ID のマクロ名とメソッド ID が include/utils/fmgroids.h に定義されていて、メソッド ID と組み込み関数の定義が utils/fmgrtab.c に登録されている。
組み込み関数の実体は、utils/adt/ 以下のファイルに定義されている。

Copyright © 2003 NTTDATA

例えば、bool 型の値を Datum に格納する関数を調べてみる。

Procid[] の bool 型の定義を調べると inproc のメソッド ID のマクロ名が F_BOOLIN であることがわかる。(pg_type テーブルから調べることもできる)

```
static struct typinfo Procid[] = {
    {"bool", BOOLOID, 0, 1, F_BOOLIN, F_BOOLOUT},
    {"bytea", BYTEAOID, 0, -1, F_BYTEAIN, F_BYTEAOUT},
    {"char", CHAROID, 0, 1, F_CHARIN, F_CHAROUT},
    {"name", NAMEOID, 0, NAMEDATALEN, F_NAMEIN, F_NAMEOUT},
    {"int2", INT2OID, 0, 2, F_INT2IN, F_INT2OUT},
    {"int2vector", INT2VECTOROID, 0, INDEX_MAX_KEYS * 2, F_INT2VECTORIN, F_INT2VECTOROUT},
    {"int4", INT4OID, 0, 4, F_INT4IN, F_INT4OUT},
            :
            :
    {"text", TEXTOID, 0, -1, F_TEXTIN, F_TEXTOUT},
            :
    {"_aclitem", 1034, 1033, -1, F_ARRAY_IN, F_ARRAY_OUT}
};
```

F_BOOLIN を include/utils/fmgroids.h で調べると、次のように定義されているので、メソッドの ID が 1242 であることがわかる。

```
#define F_BOOLIN 1242
```

utils/fmgrtab.c で 1242 のメソッドを調べると、次のような行が見つかり、 boolin という関数名であることがわかる。

```
{ 1242, "boolin", 1, true, false, boolin },
```

関数 boolin() は、utils/adt/bool.c で定義されている。boolin() は、戻り値として直接 true/false を返すので、直接 Datum の中に値を格納するタイプの属性である。

次に、可変長データ型である text 型の場合を調べてみる。メソッド ID の定義は、次のようになっており、メソッドのＩＤが 46 である。

```
#define F_TEXTIN 46
```

utils/fmgrtab.c で 46 のメソッドを調べると、次のような行から関数名は textin とわかる。

```
{ 46, "textin", 1, true, false, textin },
```

textin() は、utils/adt/varlena.c に定義してある。textin()の中では、palloc()でデータ領域を確保して、その中に値をコピーする。結果として、palloc() で確保した領域のポインタを返す。

実際に、bootstrap.c で values に値を格納している部分は次のようになっている。

```
values[i] = OidFunctionCall3(ap->am_typ.typinput,
                             CStringGetDatum(value),
                             ObjectIdGetDatum(ap->am_typ.typelem),
                             Int32GetDatum(-1));
```

Copyright © 2003 NTTDATA

OidFunctionCall3() は、３つの引数をとる関数を呼び出す関数である。ここで使用する関数は、ap->am_typ.typinput である。これは、属性の型によって異なる。
例えば、前述の bool 型の場合、ap->am_typ.typinput は、boolin になっている。３つの引数を与えた状態、つまり次のような関数呼び出しを行う。

```
boolin( CStringGetDatum(value), ObjectIdGetDatum(ap->am_typ.typelem), Int32GetDatum(-1))
```

結果として、true か false が返ってくるので、それが valus[i] に入る。

同様に text の場合は textin()が呼び出され、ポインタが返ってくるので、それが valus[i] に入る。

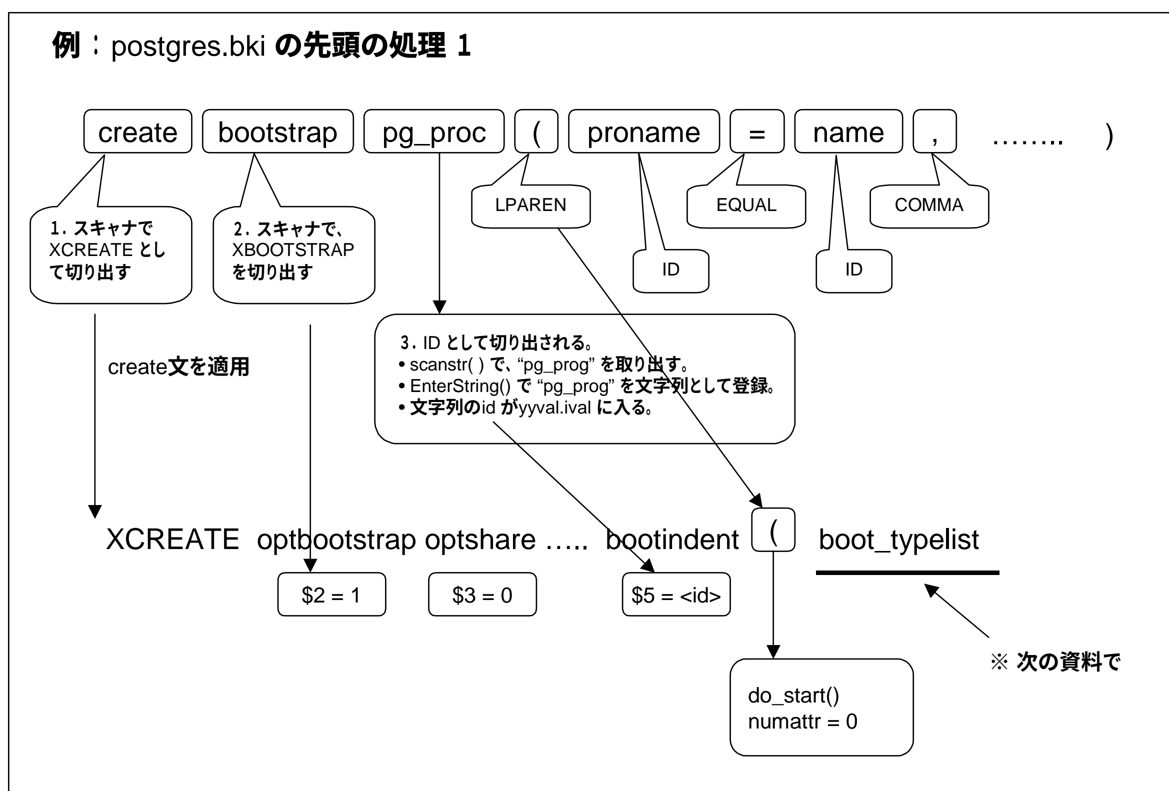
5.6. その他

インデックスの定義情報にも、リスト構造を使用しているが、普通のリスト構造なので省略する。

Copyright © 2003 NTTDATA

6. 字句解析、構文解析の動き

ここでは、Bootstrap 時の字句解析・構文解析の動きを、例を挙げて説明する。
PostgreSQL では、字句解析に flex、構文解析に bison を使用している。Bootstrap 時のスキャナ(字句解析器)部分は src/backend/bootstrap/bootscanner.l で定義してあり、パーザ(構文解析器)は src/backend/bootstrap/bootparse.y で定義してある。
以下に示した例は、postgres.bki ファイルの先頭を処理する部分である。



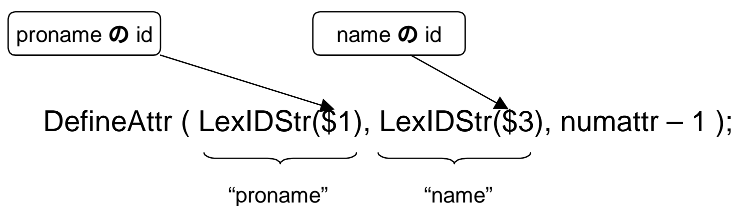
1. スキャナで create まで読み込んだ時点で XCREATE として切り出される。これを見てパーザは、XCREATE の文として判断する。
2. 次に、スキャナで bootstrap まで読んだところで、XBOOTSTRAP として切り出す。この時点で 2 番目の変数として $2 = 1 が設定される。
3. 次に、 pg_proc が ID として切り出される。scanstr() が実行され yytext から”pg_proc”という文字列をコピーする。コピーした文字列に対して EnterString() を実行すると、ハッシュを検索した時点で未登録であることがわかり、名表テーブルに”pg_proc” を登録する。登録したときの ID が $5 に入る。ちなみに$3 などには、値が設定されないので 0 が入っている。
4. ‘(‘ が読み出された時点で、do_start() が実行され、トランザクションが開始される。また、定義されている属性数 num_att などが初期化される。
5. 以降、属性名 = 型名 の定義が繰り返される。それについては次の図になる。

Copyright © 2003 NTTDATA

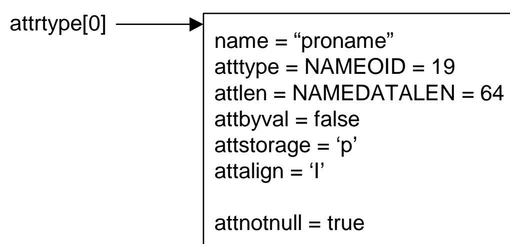
例：postgres.bki **の先頭の処理 2**（boot_typelist）

proname = name ,

構文： boot_ident EQUALS boot_ident



typeoid = gettype() ◀── Procid[] **より** 3 **が返る**



最終的に) **以降、**do_start(), **タプルデスクリプタの作成、**heap_create_with_catalog(), do_end() **が実行されて、実テーブルが作成される。**

「属性名 ＝ 型名」の処理では、1 つの属性の定義につき、DefineAttr() が 1 回実行される。proname = name という例の場合、次のようになる。

1. proname がスキャナで読み込まれ、これも名表に登録されていないので登録する。
2. また、name も登録されていないので、同様に登録する。
3. スキャナ実行時は一度名表に登録するが、パーザ処理で DefineAttr() を呼ぶ際には、LexIDStr() を使用して文字列へのポインタを渡す。ちなみに、第 3 引数で、numattr –1 しているのは、ここに来る前に、numattr をインクリメントしてしまっているので、１つ前の値を渡しているだけである。
4. DefineAttr の中では、Form_pg_attribute 構造体のデータ領域を用意し、attrtypes[0] につなぐ。まだ、型のマップテーブル Typ は NULL で、かつ、型名「name」は、Procid[] の中から見つかり、typeoid としては 3 が返ってくる。Procid[3] の情報を元に、attrtypes[0] を埋めていく。

以降、同様に属性を登録していき、最後に')' が来たところで、do_end() を一度実行する。その後、再び do_start() を実行して、タプルデスクリプタの作成CreateTupleDesc()、 実際のヒープファイルの作成 heap_create_with_catalog()[9]、do_end() が実行される。CreateTupleDesc()、heap_create_with_catalog()などは、通常のSQL文からのcreate文で使用される関数と同じである。

[9] bootstrap オプションがない場合は、heap_create() が呼ばれる。

Copyright © 2003 NTTDATA

7. postgres.bki ファイルから作られるテーブル

PostgreSQL 7.3.3 現在で、次のようなテーブルが作成される。

テーブル名	オプション	データ数	内容
pg_proc	B	1402	関数とプロシージャ
pg_type	B	117	データ型
pg_attribute	B, no_oid	146	属性の定義
pg_class	B	8	DB を構成するテーブル、属性などのオブジェクト。 pg_type, pg_attribute, pg_proc, pg_class, pg_shadow, pg_group, pg_database, pg_xactlock など
pg_attrdef		0	属性のデフォルト値
pg_constraint		0	制約条件
pg_inherits	no_oid	0	テーブルの継承関係
pg_index	no_oid	0	インデックス情報
pg_operator		643	演算子
pg_opclass		51	インデックスアクセスメソッドの演算子クラス
pg_am		4	インデックスのアクセスメソッド(rtree,btree,hash,gist)
pg_amop		180	アクセスメソッドの演算子
pg_amproc	no_oid	57	アクセスメソッドの支援のための関数
pg_language		3	関数記述言語
pg_largeobject	no_oid	0	ラージオブジェクト
pg_aggregate	no_oid	60	集約関数
pg_statistic	no_oid	0	オプティマイザ用の統計情報
pg_rewrite		0	リライターの書き換え規則
pg_trigger		0	トリガ
pg_listener	no_oid	0	非同期通知
pg_description	no_oid	0	データベースオブジェクトの説明やコメント
pg_cast		174	キャスト(型変換)のルール
pg_namespace		3	名前空間
pg_conversion		0	変換のルール
pg_database	B, S	1	データベースクラスタ(インスタンス)内のデータベース
pg_shadow	B, S, no_oid	1	データベースユーザ
pg_group	B, S, no_oid	0	データベースユーザのグループ
pg_depend	no_oid	0	オブジェクトの依存関係

※ オプションの記号（B : bootstrap, S: shared_relation, no_oid : without_oids）

7.1. postgres.bki ファイルの作成

postgres.bki ファイルは、catalog/genbki.sh により作成される。include/catalog ディレクトリ以下の次のファイル（pg_proc.h pg_type.h pg_attribute.h pg_class.h pg_attrdef.h pg_constraint.h

Copyright © 2003 NTTDATA

pg_inherits.h pg_index.h pg_operator.h pg_opclass.h pg_am.h pg_amop.h pg_amproc.h pg_language.h pg_largeobject.h pg_aggregate.h pg_statistic.h pg_rewrite.h pg_trigger.h pg_listener.h pg_description.h pg_cast.h pg_namespace.h pg_conversion.h pg_database.h pg_shadow.h pg_group.h pg_depend.h indexing.h ）と include ディレクトリ以下の postgres_ext.h、pg_config.h から生成する。
自動生成することにより、プログラム内部で使用する OID の定義とデータとして DB に登録してある OID が確実に一致するようになっている。

Copyright © 2003 NTTDATA