PostgreSQL を拡張せよ！

2014-06-18 db tech showcase B13 (14:00-14:50)

SRA OSS, Inc. 日本支社 高塚 遙

PostgreSQL もう一つの特長

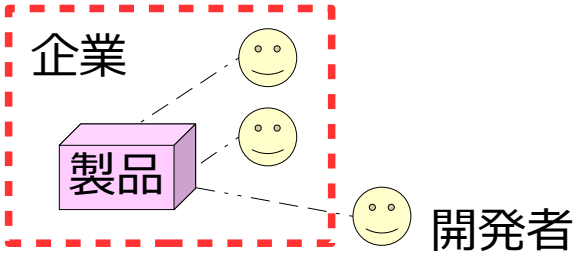
- 歴史あるリレーショナルデータベースソフトウェア
- 多機能・高性能
- 高水準のSQL標準準拠
- オーナー企業を持たないオープンソースソフトウェア開発体制
- **高い拡張性**

本日の
テーマ

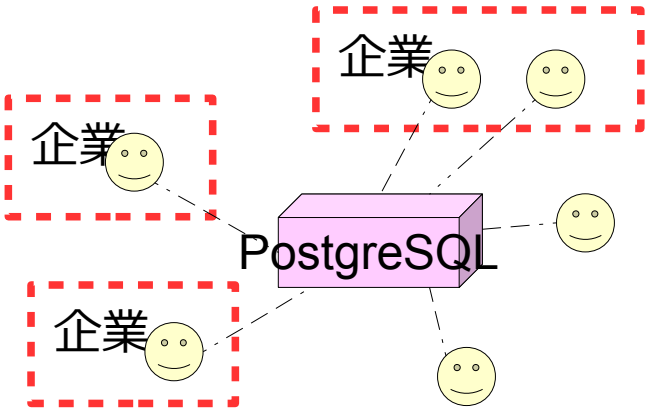
PostgreSQLの歴史

INGRES (1973)
|
POSTGRES (1985)
|
PostgreSQL (1996)
|

よくあるOSS開発体制



PostgreSQL開発体制



本セッションの目的

皆さまに
「PostgreSQL はこんな風に拡張できるんだ」
「PostgreSQL拡張モジュールを書いてみようかな」
と思っていただくことを目標にしています。

一般的な話でなく、「作る人」にむけた、
それなりに技術寄りな話になります。

PostgreSQL で拡張できるもの

- ネイティブ関数（C関数）
  - 集約関数、トリガ関数
- データ型
  - 型キャスト
- 演算子
  - 演算子クラス、演算子族
- インデックスメソッド
  - インデックスアクセスメソッド
- 手続き言語（新しいストアド関数実装用の言語）
- 外部データラッパ（FDW）
- バックグラウンドワーカプロセス
- 文字コード変換規則
- hook
  - プランナフック、Executorフック、クライアント認証フック、パーサフック、ログ出力フック、GUC設定フック、オブジェクトアクセスフック、パスワードチェックフック

拡張モジュールの使い方

- （もしバイナリ提供されていなければ）コンパイル、インストール

```
$ cd new_module
$ USE_PGXS=1 make
$ su
# USE_PGXS=1 make install
```

PostgreSQLソースツリー内で
ビルドしない場合には、
USE_PGXS=1 指定が必要
というケースがよくある。

- 必要に応じて postgresql.conf 設定を書いたり、PostgreSQL再起動をする
  - それを必要とするモジュールもあれば、必要としないモジュールもある

- データベース上にモジュールを読み込み

```
$ psql -U postgres mydb
mydb=# CREATE EXTENSION new_module;
CREATE EXTENSION
```

  - 「DROP EXTENSION」で除去できます

ネイティブ関数（C関数）

- 全ての基本
- API が用意されている。しかし、何でも書けてしまう
  - 「内部構造に触れないように作る」「管理者ユーザしか使えない関数として定義する」等は、その拡張モジュールを作った人、あるいは使う人の責任

- お手本は contrib ディレクトリの追加モジュール
  - SQL定義部分

```
CREATE FUNCTION pg_catalog.pg_file_unlink(text)
 RETURNS bool
 AS 'MODULE_PATHNAME', 'pg_file_unlink'
 LANGUAGE C VOLATILE STRICT;
```

本例は、contirb/adminpackから

  - C言語定義のモジュール部分

```
PG_MODULE_MAGIC;
PG_FUNCTION_INFO_V1(pg_file_unlink);
```

冒頭に必要なマクロ

ネイティブ関数（C関数）／ サンプルコード

```
Datum
pg_file_unlink(PG_FUNCTION_ARGS)
{
    char  *filename;
    requireSuperuser();
    filename = convert_and_check_filename(PG_GETARG_TEXT_P(0), false);
    if (access(filename, W_OK) < 0)
    {
        if (errno == ENOENT)
            PG_RETURN_BOOL(false);
        else
            ereport(ERROR,
                    (errcode_for_file_access(),
                     errmsg("file \"%s\" is not accessible: %m", filename)));
    }
    if (unlink(filename) < 0)
    {
        ereport(WARNING,
                (errcode_for_file_access(),
                 errmsg("could not unlink file \"%s\": %m", filename)));
        PG_RETURN_BOOL(false);
    }
    PG_RETURN_BOOL(true);
}
```

引数マクロ

管理者ユーザチェックや、ディレクトリ制限をしているが、それが必須という訳ではない。

エラー報告マクロの使い方は文書化済。例外スローを兼ねる。

PG_TRY()
PG_CATCH()
PG_END_TRY()

返し値マクロ

ネイティブ関数（C関数）／ ビルドと EXTENSION対応

- Makefile
  - module_name.so ファイルがビルドされる

```
MODULE = module_name
EXTENTION = module_name
DATA = module_name--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

CREATE EXTENSION
コマンドで管理するなら、
これらも加える。

これで
必要な定義が自動追加。

- CREATE EXTENSION コマンドに対応する
  - module_name.control ファイルを記述する

```
comment = 'module_name is ...'
default_version = '1.0'
module_pathname = '$libdir/moduke_name'
relocatable = false
schema = pg_catalog
```

ネイティブ関数（C関数）／ SPI・共有メモリ

- SQL実行のAPI

グローバル変数に結果が格納される

2行目第3カラム値のC文字列表現を取得

```
SPI_connect();
r = SPI_execute("SELEC * FROM t1", true, 0);
if (r == SPI_OK_SELECT && SPI_tuptable) {
    dat = SPI_getvalue( SPI_tuptable->vals[1],
                        SPI_tuptable->tupdesc,
                        3);
}
SPI_finish();
```

- 共有メモリ

ロード時のコールバック関数で予約しておいて

```
void _PG_init(void) {
 RequestAddinShmemSpace(MAXALIGN(sizeof(myShmStruct)))
 RequestAddinLWLocks(1);
}
```

PostgreSQL 9.4以降では新APIで動的確保可能

```
LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
my_shm = ShmemInitStruct("my shm", sizeof(myShmStruct), &found);
if (!found)
{
    memset(my_shm, 0, sizeof(myShmStruct));
    my_shm->lock = LWLockAssign();
}
LWLockRelease(AddinShmemInitLock);
```

名前を付けて共有メモリを新規割り当て、または、アタッチできる

ネイティブ関数（C関数）／ まとめ

- 「データベース上で使える関数を定義できる」という理解では、少々もったいない

- メモリ管理
- 共有メモリ管理
- 排他制御機能
- エラーメッセージと例外スロー処理
- 認証処理
- RDB機能全般との連携

これらが利用可能であり、サーバを記述するリッチなフレームワークといえる。

- 様々な拡張枠組みは「所定の規約にのっとった C関数を書く」という形を取るものが多く、それら関数におけるルールは、ユーザ定義C関数を書くときと、ほぼ共通
- どのヘッダファイルが必要かという情報が少ない／include ディレクトリをgrep したり、お手本の contrib モジュールを真似せよ

データ型（1）

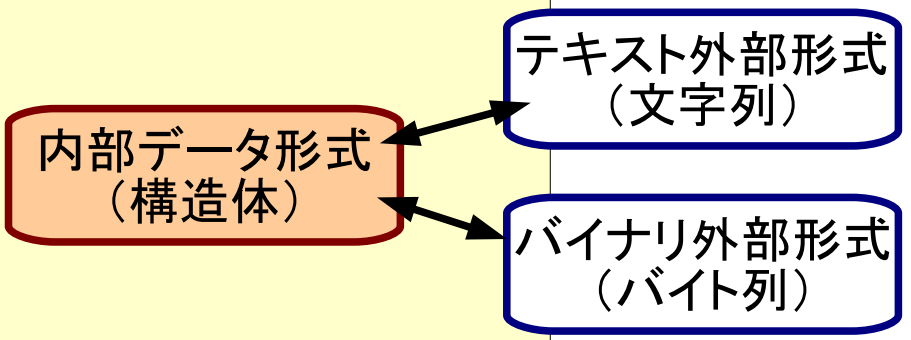
- CREATE TYPE コマンドで定義できる
  - 本質的にネイティブのデータ型と変らないものを定義できる

```
CREATE TYPE walseg; -- 仮定義

CREATE FUNCTION walseg_in(cstring) RETURNS walseg
 AS 'MODULE_PATHNAME','walseg_in' LANGUAGE C
 IMMUTABLE STRICT;

CREATE FUNCTION walseg_out(walseg) RETURNS cstring
 AS 'MODULE_PATHNAME','walseg_out' LANGUAGE
 C IMMUTABLE STRICT;

CREATE TYPE walseg (
    internallength = 16, -- データ長
    input = complex_in,  -- 入力関数
    output = complex_out -- 出力関数
);
```



  - アラインメント、ANALYZE関数、バイナリ出入力関数、等も指定可能
  - 可変長型は所定の様式に従えば自動圧縮機能も適用

型キャスト

- データ型定義のオプションで型キャスト関連の指定ができる
  - カテゴリ
  - preferred か否か？

```
=# SELECT typname, typcategory category,
    typispreferred preffer FROM pg_type;
   typname     | category | prefer
---------------+----------+--------
 char          | S        | f
 text          | S        | t
 varchar       | S        | f
   :
```

- CREATE CASTコマンドでキャスト定義できる

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION func_name (arg_type [, ...])
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

  - 関数定義せずに、テキスト出入力を通したキャストも定義可能
  - 暗黙の型変換規則について詳細に文書化されている

演算子

- CREATE OPERATOR コマンドで定義可能
  - データベース上の関数を指定
    - 先に CREATE FUNCTION が必要ということ
  - + - * / < > = ~ ! @ # % ^ & | ` ? を63文字まで組み合わせた任意の演算子が指定可能 （パーサの都合で若干の制約あり）
  - 結合時の選択率を返す関数を指定できる／プランナ向けに
  - ハッシュ結合、マージ結合に利用可能かを指定できる

```
CREATE OPERATOR = (
  LEFTARG = walseg,     RIGHTARG = walseg,
  COMMUTATOR = =,       NEGATOR  = <>,
  PROCEDURE  = walseg_eq,
  RESTRICT   = eqsel,   JOIN     = eqjoinsel,
  HASHES, MERGES
);
```

- 演算子にインデックスを適用させるには、このほかに演算子クラス（および演算子クラスを束ねる演算子族）を定義する

新しいデータ型にインデックスを適用する（1）

- 一般に言う「インデックスの種類」＝インデックスアクセスメソッド
  - これの新規追加はあまりない（大仕事になる）

```
=# SELECT amname FROM pg_am;
 amname
--------
 btree
 hash
 gist
 gin
 spgist
```

- アクセスメソッド→演算子族→演算子クラス
  - いくつかのデータ型を扱える演算子にあるデータ型のインデックスを利用させるには演算子族を定義／プランナに対するヒントになる

```
 amname |       opfname        |       opcname
--------+----------------------+----------------------
 btree  | abstime_ops          | abstime_ops
                :
 btree  | datetime_ops         | date_ops
 btree  | datetime_ops         | timestamp_ops
 btree  | datetime_ops         | timestamptz_ops
                :
```

新しいデータ型にインデックスを適用する(2)

- 演算子クラス定義の例:

```
CREATE OPERATOR CLASS waseg_ops
DEFAULT FOR TYPE walseg USING btree AS
  OPERATOR  1  <  (walseg, walseg), // 1:小なり
  OPERATOR  2  <= (walseg, walseg), // 2:以下
  OPERATOR  3  =  (walseg, walseg), // 3:等しい
  OPERATOR  4  >= (walseg, walseg), // 4:以上
  OPERATOR  5  >  (walseg, walseg), // 5:大なり
  FUNCTION  1  walseg_cmp(walseg, walseg); // 1:比較


CREATE OPERATOR CLASS walseg_ops
DEFAULT FOR TYPE walseg USING hash AS
  OPERATOR  1   =  (walseg, walseg), // 1:等しい
  FUNCTION  1   walseg_hash(walseg); // 1:ハッシュ値計算
```

btree の1～5番の「ストラテジ」に対応した演算子指定

btree のサポート番号に対応したサポート関数を指定

hash の「ストラテジ」「サポート番号」に対応した演算子と関数を指定

- ハッシュ関数には既存の汎用ハッシュ関数「hash_any(unsigned char* k, int keylen)」が流用できる

手続き言語

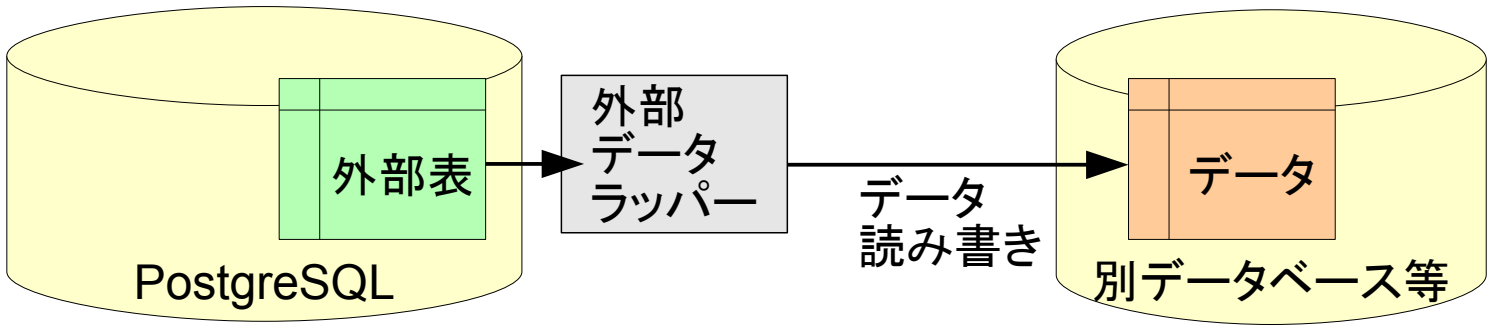
- ユーザ定義関数の記述言語は一つではない
  - PL/SQL、PL/pgSQL、PL/Python、PL/Perl、PL/Tcl、（C言語）
- 以下を定義して、手続き言語を追加できる
  - ハンドラ関数
  - バリデータ（有効性検証）関数［必須ではない］ 構文チェックに使用
  - インラインハンドラ関数　　［必須ではない］ DO コマンドで使用

```
CREATE LANGUAGE name HANDLER call_handler
 [ INLINE inline_handler ] [ VALIDATOR valfunction ]
```

- ハンドラの中で・・・
  - `fcinfo->flinfo->fn_oid` で実行された（新たな言語で定義された関数の）OID 番号 を取り出す
    - fcinfo は PG_FUNCTION_ARGS マクロの実体名 （マクロにした意味がない）
    - 引数も fcinfo 構造体変数から取り出せる
  - pg_proc テーブルを WHERE oid = 1234 として検索し、（新たな言語で書かれた）関数定義本体を読み出し、その内容を実行する

外部データラッパー（1）

- 外部のデータストアにあるデータを「外部表」として取り込み、透過的に読み書きする機能
- 新たなデータストアに対応したドライバモジュールを追加できる



- ハンドラ関数と検証関数を指定する
  - ハンドラ関数は多数のコールバック関数のポインタを格納した構造体を返すことになっている
    - スキャン用、更新用、EXPLAIN用、ANALYZE用
    - 結局、規定に従った多数の関数を実装する必要がある
  - 「PostgreSQLにおいてテーブルデータをスキャンする仕組み」を理解しないと記述がむずかしい

外部データラッパー（2）

- contrib/postgres_fdw の例

```
Datum
postgres_fdw_handler(PG_FUNCTION_ARGS)
{
  FdwRoutine *routine = makeNode(FdwRoutine);

  /* Functions for scanning foreign tables */
  routine->GetForeignRelSize = postgresGetForeignRelSize;
  routine->GetForeignPaths = postgresGetForeignPaths;
  routine->GetForeignPlan = postgresGetForeignPlan;
  routine->BeginForeignScan = postgresBeginForeignScan;
  routine->IterateForeignScan = postgresIterateForeignScan;
  routine->ReScanForeignScan = postgresReScanForeignScan;
  routine->EndForeignScan = postgresEndForeignScan;

  /* Functions for updating foreign tables */
  routine->AddForeignUpdateTargets = postgresAddForeignUpdateTargets;
  routine->PlanForeignModify = postgresPlanForeignModify;
  routine->BeginForeignModify = postgresBeginForeignModify;
  routine->ExecForeignInsert = postgresExecForeignInsert;
  routine->ExecForeignUpdate = postgresExecForeignUpdate;
  routine->ExecForeignDelete = postgresExecForeignDelete;
                  :
```

参照用

バックグラウンドワーカ

- ユーザ定義の PostgreSQL 子プロセスを作る枠組み
  - 9.3 までは PostgreSQL 起動時に起動する
    - 起動直後／接続可能後／リカバリ完了後
  - 9.4 からは動的に起動可能
    - ユーザ定義関数から RegisterBackgroundWorkerを呼べる
  - shared_preload_libraries 設定に記述してモジュールをロード

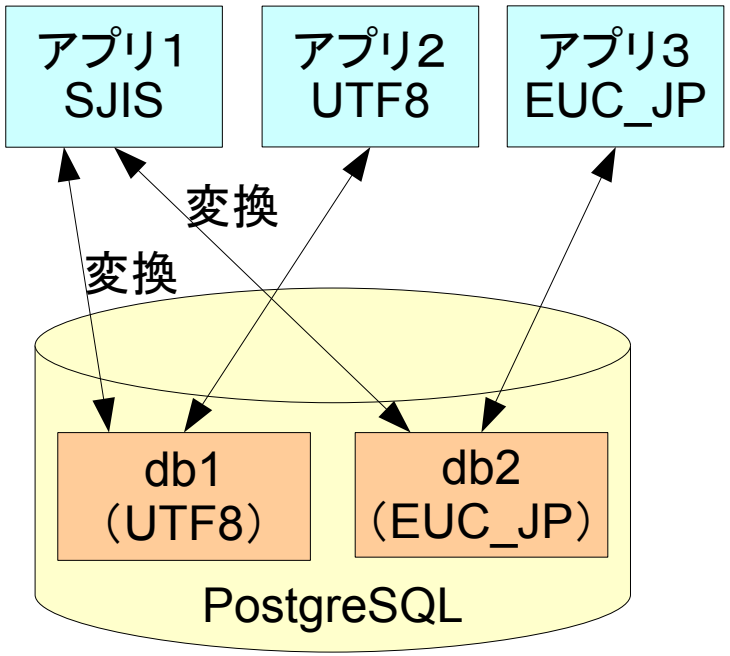
```
void
_PG_init(void)
{
  BackgroundWorker worker;
  worker.bgw_flags = BGWORKER_SHMEM_ACCESS |
                     BGWORKER_BACKEND_DATABASE_CONNECTION;
  worker.bgw_start_time = BgWorkerStart_RecoveryFinished;
  worker.bgw_restart_time = BGW_NEVER_RESTART;
  worker.bgw_main = my_worker_main;
  worker.bgw_main_arg = Int32GetDatum(0);

  RegisterBackgroundWorker(&worker);
}
```

起動タイミング、再起動の設定

文字コード変換

- PostgreSQLは文字エンコーディング変換エンジンを自身で持っている
- 新たな『変換』を定義して、指定の文字エンコーディング間のデフォルトの変換と差し替えることができる




```
CREATE [ DEFAULT ] CONVERSION name
 FOR src_encoding TO dst_encoding
 FROM func_name;
```

- UTF8 と各種文字エンコーディングとのマッピングを替える用途など
- 応用例:「eudc 拡張モジュール」
  - UTF8 ⇔ SJIS、EUC_JP で外字領域もマッピング対象に含める

```
conv_func(
 integer,  -- 変換元OID
 integer,  -- 変換先OID
 cstring,  -- 変換元文字列
 internal,-- 変換先文字列
 integer   -- 変換元文字列長
) RETURNS void ...
```

フック利用

- PostgreSQLソースコード上の hook
  - 関数ポインタのグローバル変数が定義されていて、NULL でないなら、そこで実行という記述が散在している
  - 関数定義して、LOAD 命令や shared_preload_libraries 設定等でモジュールをロードして適用する

```
/* if an advisor plugin is present, let it manage things */
if (ExplainOneQuery_hook)
  (*ExplainOneQuery_hook) (query, into, es, queryString, params);
```

- プランナフック
- Executorフック
- クライアント認証フック
- パーサフック
- ログ出力フック
- GUC設定フック
- オブジェクトアクセスフック
- パスワードチェックフック

既存のフックは、概ね決まった目的のために用意されている。

とはいえ、想定外の使い方ができないわけではない。

その他の重要な技法／カスタム設定パラメータ

- カスタム設定パラメータ(GUC)を使用する
  - パラメータ値のデータ型ごとに DefineCustome***Variable というAPIが用意されている

```
DefineCustomIntVariable(
   "my_module.my_parameter", // パラメータ
   "My parameter.",          // 説明文
   NULL,
   &my_parameter_global_var, // 値を格納する変数
   10,  // 初期値
   1,   // 最小値
   100, // 最大値
   PGC_SIGHUP, // 変更コンテクスト
   0,   // フラグ(値の単位、「ALL」に含むか、等各種特性を指定)
   NULL // チェックフック関数,
   NULL // アサインフック関数,
   NULL // 表示フック関数
);
```

グローバルスコープで定義しておく

guc.h に説明がある

新たな拡張枠組み

- Logical Decoding (PostgreSQL 9.4)
  - 行レベルの変更情報を WALファイル（PostgreSQLのトランザクションログファイル）に出力
  - レプリケーションツールや監査ツールに応用

- Custom Scan APIs / Custom Plan node (for v9.5)
  - 実行プラン要素を拡張モジュールで追加できるようにする
    - 「CREATE CUSTOM PLAN」コマンド
  - 当面の応用ターゲットは PG-Strom
    - 現状 FDW として実装されているが、適用時の透過性に劣る

- 最近は話題に上らないもの
  - ストレージマネージャ
    - 磁気ディスク用（実際にはVFS用）モジュール一つだけが存在する

PostgreSQL拡張モジュールソフトウェア名鑑（1）

- PostGIS
  - 地理情報用のデータ型、演算子（gist や spgist インデックスむけ演算子クラス）、関数を提供
  - 「キラー」拡張モジュールであり、PostGIS を使うために PostgreSQL 採用という例は少なくない

- pg_bigm（ピージーバイグラム）
  - 2-gram 全文検索用の text型に対する演算子（＋ginインデックスむけ演算子クラス）、関数を提供／日本語むけN-gram型全文検索の有力な選択肢

- PL/proxy (SkyTools)
  - 分散クエリツール／他のPostgreSQLサーバにクライアント接続を行ってリモート処理をして結果を返す／「手続き言語」として実装されている点がユニーク

PostgreSQL拡張モジュールソフトウェア名鑑（2）

- PL/v8
  - JavaScript 手続き言語／V8 JavaScriptエンジンを組み込み／高速であることが知られる
    ※ 手続き言語は、このほか多種多様に開発・公開されています
- oracle-fdw
  - Oracle database むけの外部データラッパー／同名ソフトウェアが独立して2系統存在する（NTT OSSセンタ版、Laurenz Albe氏版）
    ※ FDW はこのほか多種多様に開発・公開されています
- pgstatsinfo
  - PostgreSQL監視ツール／PostgreSQL本体と一緒に起動停止するエージェントプロセスが稼動情報を収取する／未だバックグラウンドワーカーが無かったころに作られたので独自に子プロセス起動する
- pg_hint_plan
  - hook によりヒント句機能を実装している

参考資料ならびに素晴らしい先人の皆様

- PostgreSQLマニュアル
  - 35章「SQLの拡張」
  - 44章「サーバプログラミングインタフェース」
  - 45章「バックグラウンドワーカプロセス」
  - 51章「手続き言語ハンドラの作成」
  - 52章「外部データラッパの作成」
  - 54～57章（インデックスに関する諸章）

※章番号はPostgreSQL 9.3を基準に記載しています。

- 花田茂 様 の各種発表資料
  - Postgres_fdw の開発者／FDW の第一人者
  - FDW を開発するための手引き資料など各種

その他、国内外多数のPostgreSQL開発者の皆様の成果物を参考にしています。

- ぬこ@横浜 様 の各種発表資料
  - たくさんの（特にサンプルとして）有益なPostgreSQL拡張を作成
  - PostgreSQLの各種カンファレンスで多数発表

まとめ

やってみれば簡単です。是非試してください。
PostgreSQLフレームワーク下の 快適な Cプログラムを
楽しみましょう。

これは PostgreSQL拡張してできるのでは？
と思ったら、作る（作らせる）ことを検討してみてください。

ドキュメントは充実していますが書いていないことも多いです。
しかし、オープンソースの「お手本」が沢山あります。

オープンソースとともに

URL：http://www.sraoss.co.jp/
E-mail：sales@sraoss.co.jp
Tel：03-5979-2701