`CREATE FUNCTION ... LANGUAGE pl/<any>`

make adding new pl-languages easy and safe

2023.02.24 hannuk@google.com

HannuKrosing

Cloud SQL / PostgreSQL

hannuk@google.com

Working with PostgreSQL since it was called `Postgres95` (and also played around with `Postgres 4.2` - without the "SQL" - a little before that).

My oldest *surviving* post on `postgresql-hackers@` mailing list archives is from January 1998, proposing using index for fast ORDER BY queries with LIMIT.

The first DBA at Skype, where I wrote patches for making **VACUUM** able t**o work on more than one table in parallel** and invented the sharding and remote call language **`pl/proxy`** to make it easy to use PostgreSQL in an infinitely scalable way.

Have written books, **PostgreSQL 9 Admin Cookbook** and **PostgreSQL Server Programming**

After Skype I did 10+ years of PostgreSQL consulting all over the world as part of 2ndQuadrant.

For last four years he has been a PostgreSQL Database Engineer at Google working mostly with PostgreSQL / Cloud SQL.

Plan for this talk

- What is a Database Server and why Server-Side programming
- PostgreSQL as a mixed-language platform, many flavours of functions
- What is a "pl/" language, types and flavours of PgSQL languages
- Supporting any language in **pl/v8**, turning it into **pl/<any>**
- Fun things to do with **pl/proxy**

What is a Database Server (and why Server-Side programming)

- Database server is **not** primarily "a place to store data"
- It is about organization, correctness and and control
- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Interacting with Database Server

- You send you code (processing requests) to the data
- You get back result, usually some data, or status
- SQL is not complicated
-
-

Interacting with Database Server

- You send you code (processing requests) to the data
- You get back result, usually some data, or status
- SQL is not complicated
- … but data management is
- SQL is just a representation of this complexity

SQL is a "no-code development environment"

- You tell the server in "an English-like language" what you want to do
- And it figures out all the complexities of doing it efficiently and securely
- If you think SQL is complicated …
-
-

SQL is a "no-code development environment"

- You tell the server in "an English-like language" what you want to do
- And it figures out all the complexities of doing it efficiently and securely
- If you think SQL is complicated …
- … try AI-based code generation, the latest craze in programming
- SQL is orders of magnitudes easier to get right

When pure SQL is not enough

- SQL does a good job of manipulating your data (the persistent state)
- Some things are not easy to express as Relational Algebra
- Then a Procedural Language is is the easiest way to organize your data interaction
- And it can be run in "client", "middleware" and/or "server"
- Or it can be automatically moved around as in the following example

Automatic procedural code "remoting" (from 2010)

Writing a remoting decorator

Decorator as class

```
class run_in_database(object):
    def __init__(self, f):
        self.f = f
        self.code = f.func_code
        self.mcode = marshal.dumps(self.code)
        self.code64 = base64.encodestring(self.mcode)
        self.db_name = "%s_%d" % (f.__name__, hash(self.mcode))
    def __call__(self, *args, **kargs):
        json_args = json.dumps((args, kargs))
        usrcur.execute('select * from run_code(%s,%s)',
                       (self.code64, json_args))
        result = cjson.loads(usrcur.fetchone()[0])
        return result
```

- serialises actual code of function
- serializes arguments, passed argunments and code to pl/python
- deserialises return value

Automatic procedural code "remoting" (from 2010)

Remote executor in postgreSQL

All the following samples need these modules and connections

```
create or replace function run_code(in code64 text, in json_args text,
                                    out json_result text) as
$$
import json, marshal, base64
args, kargs = json.loads(json_args)

code = marshal.loads(base64.decodestring(code64))
def f():pass
f.func_code = code

res = f(*args, **kargs)

return json.dumps(res)
$$ language plpythonu security definer;
```

- Deserializes arguments and code
- Creates a dummy function, then attaches deserialised code to it
- Calls the function, serialises result and returns it

Automatic procedural code "remoting" (from 2010)

Defining and calling the remoted function

All the following samples need these modules and connections

```
@run_in_database
def get_order(i_order_id):
    if 'order_plan' not in SD:
        q = 'select * from orders where id = $1'
        SD['order_plan'] = plpy.prepare(q, [ 'int' ])
    if 'order_line_plan' not in SD:
        q = 'select * from order_lines where order_id = $1'
        SD['order_line_plan'] = plpy.prepare(q, [ 'int' ])
    order = plpy.execute(SD['order_plan'], [i_order_id])[0]
    order_lines = plpy.execute(SD['order_line_plan'], [i_order_id])
    return (order, [order_line for order_line in order_lines])

#call the decorated function, it is executed in database
res = get_order(1000)
```

- Main overhead returning data, not shipping the code
- Code shipping can be changed to happen only when code is changed

PostgreSQL as a multi-language development platform

- Everything in PostgreSQL is configurable
  - Even most basic things like operator '+' is defined in system tables
  - Also types, casts, index and table access methods, type conversions
  - And most of the time there are functions as part of the definition
- Functions in any language can be called from SQL
- Functions in any language can call each other
- Common data types for arguments and return values are PostgreSQL data types

Types of functions in PostgreSQL

- simplest is scalar function - **`abs(-7)` → `7`**
- Functions can also return tuples (though this is still "kind of scalar")
- And they can return tables (or SETOF record)
- They can operate on sets of records and return scalars (AGGREGATE and WINDOW)
- They can implement TRIGGERs (RETURNS TRIGGER)
- Some are "internal", some are undocumented and can change

Functions in PostgreSQL by "language type"

- "C" and "internal"
- "pl/..." - interpreted languages

```
CREATE OR REPLACE FUNCTION add_abs(a int, b int, OUT c int)
LANGUAGE plpgsql
AS 'BEGIN c := abs(a) + abs(b); END';

SELECT add_abs(3, -7);
 add_abs
---------
      10
(1 row)
```

- SQL and "ANSI/ISO standard SQL"

Functions in PostgreSQL by "language type"

- "C" and "internal"
- "pl/..." - interpreted languages

```
CREATE OR REPLACE FUNCTION add_abs(a int, b int, OUT c int)
LANGUAGE plpgsql
AS 'BEGIN c := abs(a) + abs(b); END';

SELECT add_abs(3, -7);
 add_abs
---------
      10
(1 row)
```

```
CREATE OR REPLACE FUNCTION add_abs(a int, b int, OUT c int)
LANGUAGE plpgsql
AS $$
BEGIN
    c := abs(a) + abs(b);
END';
$$;
```

- SQL and "ANSI/ISO standard SQL"

The power of "dollar quoting"

```
CREATE OR REPLACE FUNCTION make_sql_adder(n int) RETURNS void
LANGUAGE plpgsql
AS $plpgsql$
BEGIN
    EXECUTE format($sql$
    CREATE FUNCTION add_%s(INOUT n int) LANGUAGE SQL AS $$SELECT n + %L$$;
    $sql$, n, n);
END;
$plpgsql$;

hannuk=# SELECT make_sql_adder(5);
 make_sql_adder
----------------

(1 row)

Time: 2.055 ms
hannuk=# SELECT add_5(10);
 add_5
-------
    15
(1 row)

Time: 0.842 ms
```

Functions properties in PostgreSQL

volatility

- IMMUTABLE - required for functional indexes (you can lie, but at your own peril)
- STABLE - now()
- VOLATILE - clock_timestamp(), random(),

"other"

- (NOT) LEAKPROOF
- PARALLEL { UNSAFE | RESTRICTED | SAFE }

Trusted and Untrusted languages

- Trusted - i.e. nicely sandboxed - pl/perl, pl/tcl, pl/v8
- Untrusted - pl/perlu, pl/pythonu, pl/R
- C and internal are always all-powerful, that is "untrusted"
- SQL is always trusted

Available pl languages

- "Core extensions"
  - pl/pgsql - available by default, but can be dropped
  - pl/python3u, pl/tclu, pl/perl(u)
- 3rd party
  - pl/R, pl/php, pl/sh, pl/java
  - **pl/v8**

What is a "V8"

- V8 is a
  - JIT compiled Javascript interpreter
  - for Google Chrome,
  - by Google
- High performance, Stable, keeps up with latest standards
- Runs **`JavaScript`** and **`wasm`**
- Lot's of work on security and sandboxing, process isolation, etc
- Only a subset of this security work is applicable to pl/v8
- No versioned / stable libv8.so released by Google Chrome team
  - when this was announced Linuxes dropped pl/v8 packaging

What is a "pl/v8"

pl/v8 is an extension which exposes Google's V8 javascript engine as pl language in PostgreSQL

```
create or replace function addtwo(vals int[])
returns json as $$
    return vals.map(function(i) {
        return i + 2;
    });
$$ language plv8;

select addtwo(array[0, 47, 30]);
-- Returns [2, 49, 32]
```

https://pgxn.org/dist/plv8/doc/plv8.html

What is Javascript (a.k.a. JS)

- JavaScript is a high-level, often just-in-time compiled language that conforms to the ECMAScript standard.[10] It has dynamic typing, prototype-based object-orientation, and first-class functions. It is multi-paradigm, supporting event-driven, functional, and imperative programming styles.
- started as a variant of Scheme, so functional side is still strong
- very strong developer community, 98% of web pages use JS
- JS has moved into server development as **node.js** and derivates.
- Because of the "Good Parts" lot of interesting stuff being done
- Lots of "frameworks" , some include transpilers

What is WASM

- Developers started to compile "anything" into minimalistic subset of JS called asm.js
- For example running a full linux kernel + userspace in browser https://browsix.org/ , https://bellard.org/jslinux/tech.html
- As this was found to be good, a Web-Assembly language was developed
- Together with APIs to interact with JS, CSS and DOM/HTML
- Many compilers support WASM as target
- Performance much closer to native compiled code than to interpreted JS, no slow startup issues.
- V8 fully supports WASM

Where did `pl/ls` and `pl/coffee` come from ?

When you install `pl/v8` from source or `create extension plv8;` on AWS you

will get two more languages

- `pl/coffe` - for creating functions in CoffeScript
- `pl/ls` - for creating functions in lightscript

Neither of these exists as an independent interpreter in pl/v8 code

Instead there are chunks on Javascript, which transpile the function body into standard JS and then execute the transpiled code instead of the original using V8 engine

And as it is run as Javascript, all the security and sandboxing still applies

What is a "transpiler"

- Rewrites one language to another (usually JS)
- Started as a way to unify different JS dialects, now supports almost anything
- The rewrite/transpile is automatic and invisible to users
- Often resulting code as fast as hand-written JS code
- Adds to startup time, as one more parsing rewriting pass at first call in session
- There seems to be a transpiler (or three) for any popular language
- There are even transpilers for running Java .jar files in Javascript

List of languages that compile to JS

milahu edited this page 4 days ago · 636 revisions

List of languages that compile to JavaScript and many other transpilers

Table of contents

- CoffeeScript + Family & Friends
  - Family (share genes with CoffeeScript)
  - Friends (philosophically related)
- JavaScript Extensions
  - Security enforcing JavaScript
  - Static typing
  - Synchronous to Asynchronous JavaScript Compilers (CPS)
  - JavaScript Language Extensions
- Ruby
- Python
- Erlang
- Elixir
- Perl
- Java/JVM
- Scala
- C#, F#, .NET related languages
- Lisp, Scheme
  - Clojure-like
  - Scheme-like
  - Other
- OCaml
- Haskell
- Smalltalk
- C/C++
- Basic
- Pascal, Modula, Oberon
- Go
- Multitarget
- Tierless languages (produce both client & server)
- Visual programming tools
- SQL
- PHP
- Lua
- Prolog
- Others

- https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

pl/<insert any language you want here>

pl/coffe and pl/ls are by now mostly obsolete, but the underlying technology in pl/v8 is good, so we can open transpiling by providing the transpiler as a pl/v8 function

```
CREATE FUNCTION func(args)$$
<coffescript code >
$$ LANGUAGE plv8
SET plv8.transpiler='plv8.coffescript';
```

This way all the caching and optimisations of functions are automatically applied here as well.

And just by that we have hundreds of languages available, with minimal extra work.

The extra work will mostly be some glue for parameters and return value parsing which may differ between the transpiler source and Javascript.

transpiler as pl/v8 function

```
pl_any_test=# select plv8.coffeescript2js($$cubes = (math.cube num for num in list)$$);
                coffesecript2js
-----------------------------------------------------------
 (function() {                                            +
   var cubes, num;                                        +
                                                          +
   cubes = (function() {                                  +
     var _i, _len, _results;                              +
     _results = [];                                       +
     for (_i = 0, _len = list.length; _i < _len; _i++) {+
       num = list[_i];                                    +
       _results.push(math.cube(num));                     +
     }                                                    +
     return _results;                                     +
   })();                                                  +
                                                          +
 }).call(this);                                           +
```

```
157
158  CREATE OR REPLACE FUNCTION plv8.coffeescript2js(
159    IN coffeescript text,
160    OUT javascript text
161  )
162  LANGUAGE plv8
163  AS $plv8$
164  (function(root) {
165      var CoffeeScript = function() {
166          function require(a) {
167              return require[a]
168          }
169  >       require["./helpers"] = new function() {···
209  >       }, require["./rewriter"] = new function() {···
397  >       }, require["./lexer"] = new function() {···
728  >       }, require["./parser"] = new function() {···
10990 >      }, require["./scope"] = new function() {···
11056 >      }, require["./nodes"] = new function() {···
12219 >      }, require["./coffee-script"] = new function() {···
12302 >      }, require["./browser"] = new function() {···
12343        };
12344        return require["./coffee-script"]
12345    }();
12346    typeof define == "function" && define.amd ? define(function() {
12347        return CoffeeScript
12348    }) : root.CoffeeScript = CoffeeScript
12349 })(this)
12350 return this.CoffeeScript.compile(coffeescript)
12351 $plv8$
12352 ;
```

added this line

Transpiler does not need to be full language → language translation

- It can be ORM
- It can be "embedded SQL"
- It can be simple "macro processor"
- It also does not need to be pl/v8 function

pl/proxy - a pl/language for remote calls and sharding

- What if your in-database code needs to access other databases ?
  - postgres_fdw, dblink, pl/proxy
- All these access other databases via libpq
- The "other database" can be a new session in the same database :)
- For easy connection we added `cloudsql.allow_passwordless_local_connections`

pl/proxy - sample use cases

- Autonomous transactions
- Writing from a read-only replica
- Writing from a parallel worker in parallel query

  (CASE WHEN condition THEN write_a_bit(x) END)
- Massively parallel CREATE TABLE AS SELECT
- Sub-second precision statistics collection

LINKS

- JavaScript Transpilers
- https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS
- Pyodide is a Python distribution for the browser and Node.js based on WebAssembly
- R in WebAssembly | Fermyon Technologies (@FermyonTech) V8 source: R/wasm.R