

# Desenvolvendo sua extensão

Euler Taveira de Oliveira

Timbira  
PostgreSQL Brasil

23 de outubro de 2009

timbira

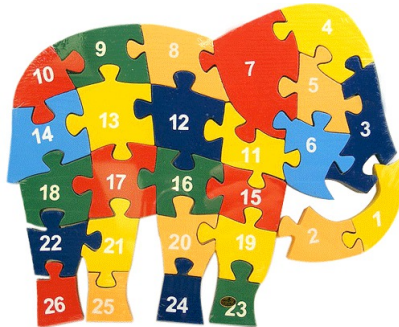
# Agenda

- 1 Introdução
- 2 Funções em C
- 3 Tipos e Operadores
- 4 Hooks
- 5 Minha Extensão

timbira

# Extensibilidade

- capacidade de adicionar novas funcionalidades a um sistema



timbira

## Extensibilidade (2)

- objetos definidos pelo usuário
  - funções
  - tipos de dados
  - conversões de tipos
  - operadores, classes de operadores e família de operadores
  - linguagens procedurais
  - busca textual
  - ...
- catálogo armazena objetos definidos pelo usuário
- PostgreSQL pode incorporar código do usuário através de carga dinâmica!

timbira

## Extensibilidade (3)

- tipos de dados
  - citext
  - hstore
  - ...
- operadores
- *hooks* (ganchos)
  - índices hipotéticos
  - GUCs
  - EXPLAIN automático
  - planos com muitas junções
  - uso de memória compartilhada
  - estatísticas hipotéticas
- API

timbira

# Agenda

- 1 Introdução
- 2 Funções em C
- 3 Tipos e Operadores
- 4 Hooks
- 5 Minha Extensão

timbira

## raiz quadrada

```
1 /* mysqrt.c */
2 #include "postgres.h"
3
4 #include <math.h>
5
6 float
7 mysqrt(float a)
8 {
9     return sqrt(a);
10 }
```

```
1 -- mysqrt.sql
2 CREATE FUNCTION mysqrt(float) RETURNS float
3 AS 'mysqrt'
4 LANGUAGE C STRICT;
```

# Funções em C

- funções compiladas em bibliotecas compartilhadas
- carregadas sob demanda pelo servidor
- carga dinâmica – diferença entre funções em C e funções internas
- convenções de funções em C e funções internas são as mesmas!
- duas versões
  - antiga (*version 0*): obsoleta mas ainda suportada
  - nova (*version 1*): portabilidade e funcionalidades novas

timbira

## Carga Dinâmica

- primeira chamada da função na sessão, biblioteca compartilhada é carregada na memória
- bibliotecas compartilhadas especificadas por:  
`dynamic_library_path = '$libdir'`
- código carregado dinamicamente é retido na memória durante aquela sessão
- bloco mágico ( $\geq 8.2$ ) garante que biblioteca não seja carregada em servidor incompatível
- `void _PG_init(void)`: chamada ao carregar
- `void _PG_fini(void)`: chamada ao descarregar (nunca é chamada)

timbira

## Funções em C: versão 0

```
1 #include "postgres.h"  
2 #include "fmgr.h"  
3  
4 PG_MODULE_MAGIC;  
5  
6 int add_one(int arg);  
7 int add_one(int arg)  
8 {  
9     return arg + 1;  
10 }
```

timbira

# Funções em C: versão 1

```
1 #include "postgres.h"
2 #include "fmgr.h"
3
4 PG_MODULE_MAGIC;
5
6 Datum add_one(PG_FUNCTION_ARGS);
7 PG_FUNCTION_INFO_V1(add_one);
8 Datum add_one(PG_FUNCTION_ARGS)
9 {
10     int32 arg = PG_GETARG_INT32(0);
11     PG_RETURN_INT32(arg + 1);
12 }
```

timbira

## Funções em C: criando função

```
1 CREATE FUNCTION add_one(integer) RETURNS integer
2 AS '/my/dir/v1', 'add_one'
3 LANGUAGE C STRICT;
4
5 CREATE FUNCTION foo(integer, integer) RETURNS integer
6 AS 'v1'
7 LANGUAGE C STRICT;
```

timbira

## Escrevendo Código

- devem ser escritas em C
- `pg_config --includedir-server` (encontrar arquivos de cabeçalho)
- compilar e ligar o seu código (produzir biblioteca compartilhada)
- lembrar de definir **PG\_MODULE\_MAGIC**
- ao alocar memória utilize **palloc** e **pfree** ao invés de **malloc** e **free**
- sempre utilize **memset** nas estruturas (garantir que "lixo" não seja escrito)

timbira

## Escrevendo Código (2)

- tipos internos são declarados em *postgres.h*
- interfaces de funções (PG\_FUNCTION\_ARGS, etc) são declarados em *fmgr.h*
- por razões de portabilidade inclua *postgres.h* antes de qualquer arquivo de cabeçalho do sistema ou do usuário
- *postgres.h* inclui *palloc.h* e *elog.h*
- nomes de símbolos não podem ser os mesmos que símbolos no PostgreSQL

## Compilação e Ligação

```
1 # Linux / FreeBSD
2 gcc -fpic -c foo.c bar.c
3 gcc -shared -o foobar.so foo.o bar.o
4
5 # Solaris
6 cc -KPIC -c foo.c bar.c
7 cc -G -o foobar.so foo.o bar.o
8
9 # Windows? WTF? ;)
```

timbira

# Agenda

- 1 Introdução
- 2 Funções em C
- 3 Tipos e Operadores**
- 4 Hooks
- 5 Minha Extensão

timbira

# Tipos Básicos

- tipo básico é um conjunto de endereços de memória
- funções que você define sobre o tipo definem como o PostgreSQL utilizará ele
  - `foo_in`
  - `foo_out`
  - `foo_recv`
  - `foo_send`
  - `foo_typmod_in`
  - `foo_typmod_out`

timbira

# Formatos Internos

- formatos internos
  - passado por valor, tamanho fixo (mesmo tamanho em todas arquiteturas)
  - passado por referência, tamanho fixo
  - passado por referência, tamanho variável

timbira

## Formatos Internos (2)

```
1 /* passed by value, fixed length */
2 typedef int int4;
3
4 /* passed by reference, fixed length */
5 typedef struct
6 {
7     double x, y;
8 } Point;
9
10 /* passed by reference, variable length */
11 typedef struct
12 {
13     int4 length;
14     char data[1];
15 } mytext;
```

## Formatos Internos: alocação de memória

```
1 #include "postgres.h"
2
3 ...
4 char buf[100];
5 ...
6 text *foo = (text *) palloc(100 + VARHDRSZ);
7 SET_VARSIZE(foo, 100 + VARHDRSZ);
8 memcpy(VARDATA(foo), buf, 100);
9 ...
```

timbira

## Equivalência de tipos C e SQL

SQL	C
boolean	bool
bytea	bytea*
character	BpChar*
integer	int4 / int32
real	float4
oid	Oid
text	text*
tid	ItemPointer
timestamp	Timestamp*
varchar	Varchar*

timbira

## Exemplo: Tipo Complexo

Mão na massa...

timbira

## Exemplo: pg\_similarity

Mão na massa...

timbira

# Agenda

- 1 Introdução
- 2 Funções em C
- 3 Tipos e Operadores
- 4 Hooks**
- 5 Minha Extensão

timbira

## Hook: o que é?

```
1 ...
2 /* Hooks for plugins to get control in ExecutorStart/
   Run/End() */
3 ExecutorStart_hook_type ExecutorStart_hook = NULL;
4 ExecutorRun_hook_type  ExecutorRun_hook   = NULL;
5 ExecutorEnd_hook_type  ExecutorEnd_hook   = NULL;
6 ...
7 void
8 ExecutorStart(QueryDesc *queryDesc, int eflags)
9 {
10     if (ExecutorStart_hook)
11         (*ExecutorStart_hook) (queryDesc, eflags);
12     else
13         standard_ExecutorStart(queryDesc, eflags);
14 }
```

## Hook: o que é? (2)

```
1 ...
2 /* Saved hook values in case of unload */
3 static ExecutorStart_hook_type prev_ExecutorStart =
4     NULL;
5 static ExecutorRun_hook_type prev_ExecutorRun = NULL;
6 static ExecutorEnd_hook_type prev_ExecutorEnd = NULL;
7 ...
8 void
9 _PG_init(void)
10 {
11     ...
12     /* Install hooks. */
13     prev_ExecutorStart = ExecutorStart_hook;
14     ExecutorStart_hook = explain_ExecutorStart;
15     ...
16 }
17
```

## Hook: onde?

- EXPLAIN (índices hipotéticos)
- GUCs
- Executor (EXPLAIN automático)
- Planejador (planos com muitas junções)
- IPC (uso de memória compartilhada)
- Planejador (estatísticas hipotéticas)

timbira

## Exemplo: auto\_explain

Mão na massa...

timbira

# Agenda

- 1 Introdução
- 2 Funções em C
- 3 Tipos e Operadores
- 4 Hooks
- 5 Minha Extensão

timbira

## Desenvolvimento

- procurar projeto similar (contrib, PgFoundry, SourceForge, ...)
- PostgreSQL dispõe de uma interface para sua extensão?
  - não: apresentar na -hackers a necessidade de uma interface naquela parte do *core*
- enviar protótipo para -hackers para comentários
- documentar a extensão (pelo menos com exemplos)
- publicar código-fonte
  - contrib
  - PgFoundry
  - *Seu Repositório Preferido Aqui*

timbira

## Módulos Adicionais (contrib)

- vários projetos fora do *core* do PostgreSQL
- Vantagens
  - desenvolvimento e lançamento separados
  - flexibilidade na escolha de uma extensão
  - personalizar uma extensão não exige reconstruir o PostgreSQL
- Desvantagens
  - **não** é possível estender o *parser* (linguagem SQL)

timbira

# Contrib x PgFoundry

- soluções experimentais
- público alvo restrito

timbira

## Contrib x PgFoundry (2)

- *contrib*
  - módulos adicionais (*contrib*): 38 (8.4) / 39? (8.5)
  - período de desenvolvimento e lançamento é o mesmo do *core*
  - maior visibilidade
  - documentação
  - facilidade em detectar que mudanças no *core* afetam uma extensão
  - extensões mais próximas ao *core*
  - "lenta" adição de funcionalidades (passar pelo crivo de um *committer*?)

## Contrib x PgFoundry (2)

- *PgFoundry*
  - projetos hospedados: 338
  - período de desenvolvimento e lançamento flexível
  - "rápida" adição de funcionalidades

timbira

## Conclusão

- **extensibilidade:** desenvolvedores com conhecimento em problemas específicos podem produzir funcionalidades sem ter que conhecer o funcionamento interno de um SGBD
- **praticidade:** resolver problemas eficientemente sem precisar aplicar patches no *core*
- **independência:** pode ser um projeto separado
- **24 x 7:** instalação ou atualização de uma extensão não necessita de parada

timbira

# Perguntas

?

Euler Taveira de Oliveira  
euler@timbira.com  
<http://www.timbira.com/>

timbira