Herança no PostgreSQL

Pelo visto o uso de herança no PostgreSQL além de ser um recurso que não está redondo não acrescenta nada ao modelo relacional, devendo ser evitado.

Veja abaixo alguns comentários e o tutorial de uso da documentação oficial.

```
Em primeiro lugar, o uso de herança não é uma boa ainda, eu não
aconselharia, mas dê uma olhada na documentação oficial:
http://pgdocptbr.sourceforge.net/pg80/tutorial-inheritance.html
```

Coutinho

Herança

Vamos criar duas tabelas. A tabela capitais contém as capitais dos estados, que também são cidades. Por conseqüência, a tabela capitais deve herdar da tabela cidades.

```
CREATE TABLE cidades (
    nome        text,
    populacao   float,
    altitude    int     -- (em pés)
);

CREATE TABLE capitais (
    estado      char(2)
) INHERITS (cidades);
```

Neste caso, as linhas da tabela capitais *herdam* todos os atributos (nome, população e altitude) de sua tabela ancestral, cidades. As capitais dos estados possuem um atributo adicional chamado estado, contendo seu estado. No PostgreSQL uma tabela pode herdar de zero ou mais tabelas, e uma consulta pode referenciar tanto todas as linhas de uma tabela, quanto todas as linhas de uma tabela mais todas as linhas de suas descendentes.

> **Nota:** A hierarquia de herança é, na verdade, um grafo acíclico dirigido. [1]

Por exemplo, a consulta abaixo retorna os nomes de todas as cidades, incluindo as capitais dos estados, localizadas a uma altitude superior a 500 pés:

```
SELECT nome, altitude
    FROM cidades
    WHERE altitude > 500;

   nome    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

Por outro lado, a consulta abaixo retorna todas as cidades situadas a uma altitude superior a 500 pés, que não são capitais de estados:

```
SELECT nome, altitude
    FROM ONLY cidades
    WHERE altitude > 500;
```

```
   nome    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

O termo "ONLY" antes de cidades indica que a consulta deve ser executada apenas na tabela cidades, sem incluir as tabelas descendentes de cidades na hierarquia de herança. Muitos comandos mostrados até agora — SELECT, UPDATE e DELETE — suportam esta notação de "ONLY".

> **Em obsolescência:** Nas versões anteriores do PostgreSQL, o comportamento padrão era não incluir as tabelas descendentes nos comandos. Descobriu-se que isso ocasionava muitos erros, e que também violava o padrão SQL:1999. Na sintaxe antiga, para incluir as tabelas descendentes era necessário anexar um * ao nome da tabela. Por exemplo:
>
> ```
> SELECT * FROM cidades*;
> ```
>
> Ainda é possível especificar explicitamente a varredura das tabelas descendentes anexando o *, assim como especificar explicitamente para não varrer as tabelas descendentes escrevendo "ONLY". A partir da versão 7.1 o comportamento padrão para nomes de tabelas sem adornos passou a ser varrer as tabelas descendentes também, enquanto antes desta versão o comportamento padrão era não varrer as tabelas descendentes. Para ativar o comportamento padrão antigo, deve ser definida a opção de configuração SQL_Inheritance como desativada como, por exemplo,
>
> ```
> SET SQL_Inheritance TO OFF;
> ```
>
> ou definir o parâmetro de configuração sql_inheritance.

Em alguns casos pode-se desejar saber de qual tabela uma determinada linha se origina. Em cada tabela existe uma coluna do sistema chamada tableoid que pode informar a tabela de origem:

```
SELECT c.tableoid, c.nome, c.altitude
FROM cidades c
WHERE c.altitude > 500;

 tableoid |   nome    | altitude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

Se for tentada a reprodução deste exemplo, os valores numéricos dos OIDs provavelmente serão diferentes. Fazendo uma junção com a tabela "pg_class" é possível mostrar o nome da tabela:

```
SELECT p.relname, c.nome, c.altitude
FROM   cidades c, pg_class p
WHERE  c.altitude > 500 AND c.tableoid = p.oid;

 relname  |   nome    | altitude
----------+-----------+----------
 cidades  | Las Vegas |     2174
 cidades  | Mariposa  |     1953
 capitais | Madison   |      845
```

Uma tabela pode herdar de mais de uma tabela ancestral e, neste caso, possuirá a união das colunas definidas nas tabelas ancestrais (além de todas as colunas declaradas especificamente para a tabela filha).

Uma limitação séria da funcionalidade da herança é que os índices (incluindo as restrições de

unicidade) e as chaves estrangeiras somente se aplicam a uma única tabela, e não às suas descendentes. Isto é verdade tanto do lado que faz referência quanto do lado que é referenciado na chave estrangeira. Portanto, em termos do exemplo acima:

- Se for declarado cidades.nome como sendo UNIQUE ou PRIMARY KEY, isto não impedirá que a tabela capitais tenha linhas com nomes idênticos aos da tabela cidades e, por padrão, estas linhas duplicadas aparecem nas consultas à tabela cidades. Na verdade, por padrão, a tabela capitais não teria nenhuma restrição de unicidade e, portanto, poderia conter várias linhas com nomes idênticos. Poderia ser adicionada uma restrição de unicidade à tabela capitais, mas isto não impediria um nome idêntico na tabela cidades.
- De forma análoga, se for especificado cidades.nome REFERENCES alguma outra tabela, esta restrição não se propagará automaticamente para a tabela capitais. Neste caso, o problema poderia ser contornado adicionando manualmente a restrição REFERENCES para a tabela capitais.
- Especificar para uma coluna de outra tabela REFERENCES cidades(nome) permite à outra tabela conter os nomes das cidades, mas não os nomes das capitais. Não existe um maneira boa para contornar este problema.

Estas deficiências deverão, provavelmente, serem corrigidas em alguma versão futura, mas enquanto isso deve haver um cuidado considerável ao decidir se a herança é útil para resolver o problema em questão.

Notas

[1] *Grafo*: uma coleção de vértices e arestas; *Grafo dirigido*: um grafo com arestas unidirecionais; *Grafo acíclico dirigido*: um grafo dirigido que não contém ciclos. FOLDOC - Free On-Line Dictionary of Computing (N. do T.)

CREATE TABLE -- cria uma tabela

Sintaxe

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE nome_da_tabela (
  { nome_da_coluna tipo_de_dado [ DEFAULT expressão_padrão ]
[ restrição_de_coluna [ ... ] ]
    | restrição_de_tabela
    | LIKE tabela_ancestral [ { INCLUDING | EXCLUDING } DEFAULTS ] }  [, ... ]
)
[ INHERITS ( tabela_ancestral [, ... ] ) ]
```

...
INHERITS (tabela_ancestral [, ...])

A cláusula opcional INHERITS (herda) especifica uma lista de tabelas das quais a nova tabela herda, automaticamente, todas as colunas.

O uso de INHERITS cria um relacionamento persistente entre a nova tabela descendente e suas tabelas ancestrais. As modificações de esquema nas tabelas ancestrais normalmente se propagam para as tabelas descendentes também e, por padrão, os dados das tabelas descendentes são incluídos na varredura das tabelas ancestrais.

Se existir o mesmo nome de coluna em mais de uma tabela ancestral é relatado um erro, a menos que o tipo de dado das colunas seja o mesmo em todas as tabelas ancestrais. Não havendo conflito,

então as colunas duplicadas são combinadas para formar uma única coluna da nova tabela. Se a lista de nomes de colunas da nova tabela contiver um nome de coluna que também é herdado, da mesma forma o tipo de dado deverá ser o mesmo das colunas herdadas, e a definição das colunas será combinada em uma única coluna. Entretanto, declarações de colunas novas e herdadas com o mesmo nome não precisam especificar restrições idênticas: todas as restrições fornecidas em qualquer uma das declarações são combinadas, sendo todas aplicadas à nova tabela. Se a nova tabela especificar explicitamente um valor padrão para a coluna, este valor padrão substituirá o valor padrão das declarações herdadas da coluna. Se não especificar, toda tabela ancestral que especificar um valor padrão para a coluna deverá especificar o mesmo valor, ou será relatado um erro.

Herança

Heranças múltiplas por meio da cláusula INHERITS é uma extensão do PostgreSQL à linguagem. O SQL:1999 (mas não o SQL-92) define herança única utilizando uma sintaxe diferente e semânticas diferentes. O estilo de herança do SQL:1999 ainda não é suportado pelo PostgreSQL.

pg_inherits

O catálogo pg_inherits registra informações sobre hierarquias de herança de tabelas. Existe uma entrada para cada tabela diretamente descendente no banco de dados (As descendências indiretas podem ser determinadas seguindo a cadeia de entradas).

Tabela 42-17. Colunas de *pg_inherits*

Nome	Tipo	Referencia	Descrição
inhrelid	oid	pg_class.oid	OID da tabela descendente.
inhparent	oid	pg_class.oid	OID da tabela ancestral.
inhseqno	int4		Se houver mais de um ancestral direto para a tabela descendente (herança múltipla), este número informa a ordem pela qual as colunas herdadas devem ser dispostas. O contador começa por 1.

Herança no PostgreSQL

Olá amigos,

nesse artigo vamos tratar um conceito que tem, a cada dia, sido mais discutido no âmbito dos bancos de dados relacionais, em especial o PostgreSQL, o conceito de herança.

Inicialmente, herança é um conceito da orientação à objetos onde, uma determinada classe herda características (no caso das classes - propriedades ou atributos e métodos ou funções) de uma outra "super" classe. A relação de herança pode ser lida da seguinte forma: "a classe que herda É UM TIPO DA super classe (classe herdada)".

Assim teríamos, por exemplo, uma classe funcionário que contém os dados comuns dos funcionários (matricula, nome, dataNascimento, ...) e uma classe gerente que É UM TIPO DE funcionário, mas com algumas outras características exclusivas como por exemplo (percentualParticipacaoLucro, telCeluar, ...).

Então, no exemplo acima temos uma classe gerente que herda da classe funcionário. Isso significa que não precisamos replicar todas as características de um funcionário para um gerente. A herança se encarrega de fazer isso e o gerente passa a ter todas as características do funcionário + as suas características próprias.

Agora falando do PostgreSQL. Ao contrário do que lemos muitas vezes por ai, o PostgreSQL não é um SGBD (Sistema de Gerência de Banco de Dados) Orientado a Objetos. Ele é o que chamamos de Sistema de Gerência de Banco de Dados Relacional Estendido ou Objeto-Relacional. Essa sua Extensão nos permite utilizar alguns conceitos próprios de um Banco Orientado a Objetos, como por exemplo tipos de dados definidos pelo usuário e herança. No caso da herança no PostgreSQL, uma tabela pode herdar de nenhuma, uma, ou várias tabelas.

Implementando o exemplo acima no PostgreSQL teríamos:

```
create table funcionario(
	matricula int,
	nome varchar,
	dataNascimento Date,
	primary key(matricula)
)
```

Ao executar o comando acima, o PostgreSQL automaticamente cria a tabela funcionário (com a estrutura informada) no esquema public.

```
create table gerente(
	percentParticipacaLucro int,
	telCel varchar
) inherits (funcionario);
```

Ao executar o comando acima, o PostgreSQL cria uma tabela gerente com os atributos de funcionário + os atributos de gerente.

Então agora temos as seguintes tabelas em nossa base:

Funcionário
 matricula int,
 nome varchar,

datanascimento date,
CONSTRAINT funcionario_pkey PRIMARY KEY (matricula)

Gerente
matricula int4 NOT NULL,
nome varchar,
datanascimento date,
percentparticipacalucro int4,
telcel varchar

Notem que ao contrário do que dissemos acima com relação às classes, na herança de tabelas no PostgreSQL os campos herdados se repetem "fisicamente" nas duas tabelas.

Para inserir um gerente usamos:
insert into gerente values (1000, 'Hesley', '01/01/1975', 10, '99999999'); -- Ao inserir um gerente, automaticamente os atributos herdados (matricula, nome, dataNascimento) são inseridos também na tabela de funcionários.

Para inserir um funcionário usamos:
insert into funcionario values (2000, 'Maria', '02/02/1980'); -- Os dados são inseridos somente na tabela funcionários e não na tabela gerente.

Para visualizarmos os dados das tabelas.

select * from gerente;
Retorna
1000;"Hesley";"1975-01-01";10;"99999999"

select * from funcionario;
Retorna
2000;"Maria";"1980-02-02"
1000;"Hesley";"1975-01-01"

Bom, agora eu quero saber quais os funcionários que não são gerentes. Em uma situação normal (sem herança) podia usar o conceito de subquery para selecionar os funcionários que não estão na tabela gerente. Uma vez implementada a herança, basta eu usar a palavrinha ONLY como na query abaixo.

select * from only funcionario; -- Retorna somente os funcionários que não são gerentes.
Retorna
2000;"Maria";"1980-02-02"

Como no SELECT, os comando UPDATE e DELETE também suportam o uso do "ONLY".

Existem algumas deficiências com relação à restrição de integridade no uso das heranças. Por exemplo, a tabela funcionário foi criada com uma chave primária matricula, o que impede que eu possua dois funcionários com uma mesma matrícula, entretanto essa restrição não é válida para a tabela herdada. Dessa forma poderiam existir dois gerentes com a mesma matrícula.

Com relação à utilidade da herança no PostgreSQl, ainda não consegui ver muita utilidade (posso estar enganado) e a maioria dos textos que tenho lido sobre o tema, desencoraja seu uso. Particularmente, acredito que os recursos relacionais existentes no PostgreSQL e nos outros bancos relacionais são suficientes e adequados para resolver os problemas que seriam tratados usando a herança (posso estar enganado).
Hesley Py
http://www.devmedia.com.br/articles/viewcomp.asp?comp=9182

Ele é considerado objeto-relacional por implementar, além das características de um SGBD relacional, algumas características de orientação a objetos, como herança e tipos personalizados. A equipe de desenvolvimento do PostgreSQL sempre teve uma grande preocupação em manter a compatibilidade com os padrões SQL92/SQL99.

Boa tarde pessoal,

Vi **alguns** posts passados questionando **a herança** na nova versão 8.3 e pelo que entendi não melhorou e nem vai melhorar.

Bem... há **algum** tempo eu fui forç**ado** (monografia) **a** modelar um banco que ficasse implementado o conceito de **herança**. No PG estava indo legal **até** que vieram os problemas de integridade referencial, procurei **ajuda** e percebi que **herança** no PG era uma "bomba". Coloquei os miolos para funcionar e o Tico-e-Teco me deram uma idéia de implementação, daí estou passando em **anexo** um script de como eu consegui implementar o conceito de **herança**.

Para o que eu precisava serviu legal e **até agora** não deu problema e talvez, quem sabe, essa idéia consiga **ajudar alguém**.
Celso Henrique Mendes Ferreira
Analista de Requisitos
www.itecgyn.com.br

Abaixo a solução do Celso:

```
---------------------------------------------------------------------------
-- LANGUAGE
---------------------------------------------------------------------------

CREATE PROCEDURAL LANGUAGE plpgsql;

---------------------------------------------------------------------------


---------------------------------------------------------------------------
-- TRIGGERS
---------------------------------------------------------------------------

CREATE FUNCTION adicionarpessoa() RETURNS "trigger"
   AS $$
begin
        insert into tbPessoa (cdPessoa, tpPessoa, nmPessoa, cpf, dtNascimento)
                values (new.cdPessoa, new.tpPessoa, new.nmPessoa, new.cpf, new.dtNascimento);
        return null;
end;
$$
   LANGUAGE plpgsql;

---------------------------------------------------------------------------

CREATE FUNCTION atualizarpessoa() RETURNS "trigger"
   AS $$
begin
```

```
        update tbPessoa set
                tpPessoa     = new.tpPessoa,
                nmPessoa     = new.nmPessoa,
                cpf          = new.cpf,
                dtNascimento = new.dtNascimento
        where cdPessoa = old.cdPessoa;
        return null;
end;
$$
   LANGUAGE plpgsql;

-----------------------------------------------------------------------------

CREATE FUNCTION removerpessoa() RETURNS "trigger"
   AS $$
begin
        delete from tbPessoa where cdPessoa = old.cdPessoa;
        return null;
end;
$$
   LANGUAGE plpgsql;

-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- SEQUENCE
-----------------------------------------------------------------------------

CREATE SEQUENCE tbpessoa_cdpessoa_seq
   INCREMENT BY 1
   NO MAXVALUE
   NO MINVALUE
   CACHE 1;

-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- TABLES
-----------------------------------------------------------------------------

CREATE TABLE tbpessoa (
   cdpessoa integer NOT NULL,
   nmpessoa character varying(50),
   tppessoa integer,
   cpf character varying(20),
   dtnascimento date
);

ALTER TABLE ONLY tbpessoa
   ADD CONSTRAINT pk_tbpessoa PRIMARY KEY (cdpessoa);

-----------------------------------------------------------------------------
```

```
CREATE TABLE tbaluno (
    cdpessoa integer DEFAULT nextval('tbpessoa_cdpessoa_seq'::regclass) NOT NULL,
    nmpessoa character varying(50),
    tppessoa integer DEFAULT 0,
    matricula character varying(20),
    cpf character varying(20),
    dtnascimento date
);

ALTER TABLE ONLY tbaluno
    ADD CONSTRAINT pk_tbaluno PRIMARY KEY (cdpessoa);

CREATE TRIGGER tgadicionarpessoa
    AFTER INSERT ON tbaluno
    FOR EACH ROW
    EXECUTE PROCEDURE adicionarpessoa();

CREATE TRIGGER tgatualizarpessoa
    AFTER UPDATE ON tbaluno
    FOR EACH ROW
    EXECUTE PROCEDURE atualizarpessoa();

CREATE TRIGGER tgremoverpessoa
    AFTER DELETE ON tbaluno
    FOR EACH ROW
    EXECUTE PROCEDURE removerpessoa();

--------------------------------------------------------------------------------

CREATE TABLE tbfuncionario (
    cdpessoa integer DEFAULT nextval('tbpessoa_cdpessoa_seq'::regclass) NOT NULL,
    nmpessoa character varying(50),
    tppessoa integer DEFAULT 1,
    cargo character varying(100),
    cdfuncionario character varying(20),
    cpf character varying(20),
    dtnascimento date
);

ALTER TABLE ONLY tbfuncionario
    ADD CONSTRAINT pk_tbfuncionario PRIMARY KEY (cdpessoa);

CREATE TRIGGER tgadicionarpessoa
    AFTER INSERT ON tbfuncionario
    FOR EACH ROW
    EXECUTE PROCEDURE adicionarpessoa();

CREATE TRIGGER tgatualizarpessoa
    AFTER UPDATE ON tbfuncionario
    FOR EACH ROW
    EXECUTE PROCEDURE atualizarpessoa();
```

```
CREATE TRIGGER tgremoverpessoa
   AFTER DELETE ON tbfuncionario
   FOR EACH ROW
   EXECUTE PROCEDURE removerpessoa();
```

Em 26/02/2008, às 00:37, Euler Taveira de Oliveira escreveu:

> Celso Henrique Mendes Ferreira wrote:
>
>> Para o que eu precisava serviu legal e até agora não deu problema e
>> talvez, quem sabe, essa idéia consiga ajudar alguém.
>>
> Mas isso *não* é herança. É de certo modo uma "replicação parcial" das
> tabelas do banco de dados.

De acordo com o Euler, porque replicar os atributos de pessoa (cpf e dtnascimento) em aluno por exemplo?
Acho que se mapearmos os conceitos OO da seguinte forma:

Classe -> Variável de relação
Atributo de classe -> Atributo de relação
Objeto -> Tupla
Composição de objetos -> Relacionamentos

O conceito mais próximo da herança no banco relacional será especialização/generalização.

PS.: Se alguém me acusar de estar cometendo um grande erro de mapeamento (como no livro do date) vou responder que não proponho um banco relacional/oo mas um mapeamento mais "solto" entre um SGBDR puro e modelagem OO (e uma definição bem geral de classes).

Pelo contrário, Diogo, você está seguindo o Date direitinho, Terceiro Manifesto e tudo.

A única nota que eu faria — que de maneira alguma é questionamento do que você escreveu — é que 'objeto' é um conceito ambígüo entre livros e implementações várias, de modo que um objeto pode ser tanto uma tupla quanto uma relação. Mas concordo contigo em que o mais são, em princípio, me parece objeto como tupla.
Leandro

Diogo,

Eu concordo que não atentei para o detalhe da Especialização/Generalização. Assumindo a OO eu não precisaria dos campos nome, cpf e dtnascimento nas tabelas filhas, apenas uma chave estrangeira de relacionamento. Assim, qualquer alteração na estrutura desses campos poderia ser feita direto na tabela mãe, o que não poderia ser feito com o esquema que criei, pois nele eu teria que alterar não só a mãe, mas em todas as filhas.

Um detalhe que não comentei é que essa implementação foi feita baseada no funcionamento de **herança** dentro do próprio PG, onde um registro numa tabela filha é **automaticamente** inserido na tabela mãe. Vendo pela Especialização/Generalização percebo que **a** estrutura fica mais simples e flexível, tornando **as** triggers completamente desnecessárias. Só que isso não é nada **al**ém de normalização de banco de dados, daí penso o seguinte: qual **a** dificuldade que os desenvolvedores têm em corrigir o problema de **herança** no PG?

--
Celso Henrique Mendes Ferreira

2008/2/26, Celso Henrique Mendes Ferreira <celsohenrique@...>:
> Vendo pela
> Especialização/Generalização percebo que **a** estrutura fica mais simples e
> flexível, tornando **as** triggers completamente desnecessárias. Só que isso não
> é nada **al**ém de normalização de banco de dados, daí penso o seguinte: qual **a**
> dificuldade que os desenvolvedores têm em corrigir o problema de **herança** no
> PG?

Exatamente — é que não tem ganho **algum**.

Atualmente, o único uso legítimo que conheço de **herança** no **PostgreSQL**
é particionamento de tabelas, que todo mundo **admite** que é **apenas** uma
gambiarra, merecendo uma implementação melhor.

Deve haver outros usos legítimos, mas todos gambiarras como o particionamento.

OO em bases de dados foi uma idéia infeliz.
Leandro