目錄

postgresql教程

postgresql 最全面，最细致的特性介绍与应用教程

原文发布于我的个人博客：https://www.rails365.net

源码位于：https://github.com/yinsigan/postgresql_tutorial

电子版: PDF Mobi ePbu

联系我:

email: hfpp2012@gmail.com

qq: 903279182

1. Full Text Search介绍

如果要实现一个站内搜索的功能，可能可以这么办，假如你的网站是个放博客的，那么可能有博客这张表，要搜索的时候，用SQL语句就好了，用where like，但是这样不够好，因为你只能搜索包含或未包含的，不能像百度那样，根据关键词来搜索，也就是分词系统。我们来介绍一下，全文检索是搜索引擎的一部分，它首先得有数据，有了数据，要根据数据来分词，例如"this is a cat"就可以分成四个词，分别是"this"，"is", "a", "cat"，或许像这些不太重要的"is"，"a", "this"的停止词(stop word)可能会被去掉，那就剩下一个词，这只是规则而已，不管怎样，不管是中文，英文，都是会切成一个个词。根据词来建立索引。索引就先理解为书中的目录，建立索引是要消耗磁盘空间的，想下就清楚了，不然索引存哪啊。索引建立好了，用户一搜索关键词，假如用户搜索了"cat"，刚好命中了那个建立过的关键词，那就会通过索引把相关的记录取出来。这就是一个全文检索系统啦。只是要实现一个较完整的全文检索系统，那是需要好多功能的，例如实时搜索，关键词提示，错误提示，还有排名等。PostgreSQL作为一个关系型数据库系统，它本身就支持全文检索，它比其他数据库支持得更好。通过简单的扩展，还能实现中文检索。这是后话。

2. PostgreSQL的文本匹配

接下来，我们用PostgreSQL的tsvector等命令处理器来测试分词和文本匹配。先来看一个例子

```
postgres=# SELECT 'hello world hfpp2012'::tsvector @@ 'hello'::t
squery;
```

输出的结果是这样的。

```
 ?column?
----------
 t
(1 row)
```

t就是true，说明是匹配成功的。如果是f，那就是false，表示匹配不成功。

上面语句的意思是总共有三个词，"hello world hfpp2012"，然后用"hello"来匹配，相当于数据库存了三个词，在搜索引擎输入框输入了"hello"，因为数据库是有"hello"这个词的，所以是能匹配到的。

再试试下面的例子。

```
postgres=# SELECT 'hello world hfpp2012'::tsvector @@ 'hello & w
orld'::tsquery;
 ?column?
----------
 t
(1 row)
```

"&"符号是停止符，是不被索引的，因为没有意义啊。也就不存储在数据中了。PostgreSQL有一条规则就是

"Define stop words that should not be indexed."

还有另一种写法是这样的。

```
postgres=# SELECT 'hello & world'::tsquery @@ 'hello world hfpp2
012'::tsvector;
 ?column?
----------
 t
(1 row)
```

来搜索中文的试下

```
rails365_pro=# SELECT '号'::tsquery @@ '2015 - Rails365 Inc. All
 rights reserved. | 粤ICP备15004902号-2'::tsvector;
 ?column?
----------
 f
(1 row)

rails365_pro=# SELECT 'Rails365'::tsquery @@ '2015 - Rails365 In
c. All rights reserved. | 粤ICP备15004902号-2'::tsvector;
 ?column?
----------
 t
(1 row)
```

显然，默认情况下，对中文是不支持的。

3. PostgreSQL的数据库全文检索

刚才测试的只是分词，我们用实际的数据库来测试一下。

```
# 列出所有数据库
\l
# 选择数据库
\c rails365_pro;
select title from articles where to_tsvector('english', body) @@
 to_tsquery('english', 'ruby')
```

输出的结果是这样的。

```
                          title
-----------------------------------------------------
 最简单的用户登录注册系统
 登录认证系统的进阶使用
 用OneAPM作为你的监控平台
 使用backup来备份数据库
 使用mina来部署ruby on rails应用
 Mina的进阶使用
 用logrotate切割Ruby on rails日志
 用exception_notification结合Slack或数据库来捕获异常
 devise简单入门教程
(9 rows)
```

上面是搜索了articles表中body字段，只要包含ruby的都找出来。这不是where like，而是先将body分成一个个词，之后再来找的。

再看一个例子。

```
rails365_pro=# select title from articles where to_tsvector('eng
lish', body) @@ to_tsquery('english', 'Mina')
;
              title
---------------------------------
 使用mina来部署ruby on rails应用
 Mina的进阶使用
(2 rows)
```

上面是搜索Mina这个关键词，两个例子都是用english作为语法的，如果搜索中文是搜索不到的。

例如，搜索title为devise简单入门教程的这篇文章。

```
rails365_pro=# select title from articles where to_tsvector('eng
lish', title) @@ to_tsquery('english', '教程')
;
 title
-------
(0 rows)
```

我们把"english"去掉。

```
rails365_pro=# select title from articles where to_tsvector(titl
e) @@ to_tsquery('教程')
;
 title
-------
(0 rows)
```

还是不行。因为没有中文分词器。这个后续再说。

3. 创建索引

上面的是没有建立索引的情况下操作的，那样肯定不行的，如果数据量大，会很慢。

```
CREATE INDEX articles_idx ON articles USING gin(to_tsvector('eng
lish', body));
```

具体的操作可以看官方的这篇文章textsearch-tables

完结。

下一篇：PostgreSQL 的全文检索系统之进阶 (二)

1. 前言

这一篇文章简要介绍了PostgreSQL的全文检索系统之基本介绍(一)，这一节来介绍一些额外的功能，比如排名，比如高亮等。

2. 解析文档(Parsing Documents)

要查看一段文本是怎么被PostgreSQL分词的，可以用to_tsvector这个指令，是这样使用的。

```
postgres=# SELECT to_tsvector('english', 'a fat  cat sat on a ma
t - it ate a fat rats');
                   to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
(1 row)
```

3. 搜索结果排名(Ranking Search Results)

就是使用ts_rank或ts_rank_cd按照匹配词的出现次数做个排名。

例如:

```
rails365_pro=# select title, ts_rank(to_tsvector('english', body
), to_tsquery('english', 'ruby')) AS rank from articles where to
_tsvector('english', body) @@ to_tsquery('english', 'ruby')
order by rank desc;
                         title                          |   rank

--------------------------------------------------------+----------
-
 使用mina来部署ruby on rails应用                          | 0.0928561
 登录认证系统的进阶使用                                   | 0.0906656
 devise简单入门教程                                       | 0.0889769
 用exception_notification结合Slack或数据库来捕获异常 | 0.0889769
 Mina的进阶使用                                           | 0.0865452
 使用backup来备份数据库                                   | 0.0827456
 用logrotate切割Ruby on rails日志                         | 0.0759909
 用OneAPM作为你的监控平台                                 | 0.0759909
 ruby                                                   | 0.0607927
(9 rows)
```

4. 结果的高亮(Highlighting Results)

有时候你需要把搜索的关键词高亮起来，就像谷歌，百度那样，PostgreSQL默认就支持的。

PostgreSQL有一个指令**ts_headline**就是来做这个事情的。

ts_headline使用起来也简单，看下面的例子:

```
rails365_pro=# select title,ts_headline('testzhcfg', title, to_t
squery('testzhcfg', 'mina')), ts_rank(to_tsvector('testzhcfg', t
itle), to_tsquery('testzhcfg', 'mina')) AS rank from articles wh
ere to_tsvector('testzhcfg', body) @@ to_tsquery('testzhcfg', 'm
ina')
order by rank;
              title              |             ts_headline
        |   rank
---------------------------------+-----------------------------
--------+-----------
 使用mina来部署ruby on rails应用 | 使用<b>mina</b>来部署rubyonrails
应用 | 0.0607927
 Mina的进阶使用                  | <b>Mina</b>的进阶使用
     | 0.0607927
(2 rows)
```

高亮的地方就用 `<b></b>` 包住了。

具体的内容可阅读官方文档textsearch-controls

完结。

下一篇PostgreSQL 的全文检索系统之中文支持 (三)

1. 前言

在这篇文章中，介绍了PostgreSQL的全文检索系统，里面有提到，**PostgreSQL**默认是不支持中文的。看下面的例子。

```
rails365_pro=# SELECT '号'::tsquery @@ '2015 - Rails365 Inc. All
 rights reserved. | 粤ICP备15004902号-2'::tsvector;
 ?column?
----------
 f
(1 row)

rails365_pro=# SELECT '粤ICP备15004902号'::tsquery @@ '2015 - Rai
ls365 Inc. All rights reserved. | 粤ICP备15004902号-2'::tsvector;
 ?column?
----------
 t
(1 row)
```

说明没有按照我们的意愿分词，我们可以自己来看看PostgreSQL是怎么分词的。只要用**to_tsvector**这个指令就好了。

```
rails365_pro=# SELECT to_tsvector('english', '2015 - Rails365 In
c. All rights reserved. | 粤ICP备15004902号-2');
                                    to_tsvector

-------------------------------------------------------------------
-----------------
 '-2':8 '2015':1 'inc':3 'rails365':2 'reserv':6 'right':5 '粤ic
p备15004902号':7
(1 row)
```

明显不符合我们的意愿。"粤icp备15004902号"应该被更详细的切分的。至少把"icp"，"号“等分开。

我们用一个中文切词的网站来演示一下。网址是 http://www.xunsearch.com/scws/demo/v48.php。

它切好的词大概是这样的。

```
2015 - Rails 365 Inc . All rights reserved . | 粤 ICP 备 15004902
 号 - 2
```

每个词都是以空格分开的。这样才是比较符合的。所以我们需要一款中文分词的PostgreSQL插件。

2. 安装

zhparser是一款中文分词的**PostgreSQL**插件。我使用过，效果不错，故推荐。

zhparser只是一个**PostgreSQL**扩展插件，它是基于**SCWS**的(一个简易中文分词系统，Simple Chinese Word Segmentation)。

2.1 第一步，安装SCWS

```
# 下载并解压
wget -q -O - http://www.xunsearch.com/scws/down/scws-1.2.2.tar.b
z2 | tar xvjf -
# 编译安装
cd scws-1.2.2 ; ./configure ; sudo make install
```

2.2 第二步，编译和安装zhparser

```
# 先安装PostgreSQL的扩展包
sudo apt-get install postgresql-server-dev-9.3
git clone https://github.com/amutu/zhparser.git
cd zhparser
SCWS_HOME=/usr/local make && sudo make install
```

2.3 第三步，进入数据库安装扩展

```
# 进入数据库
sudo -u postgres psql
# 连接数据库
\c rails365_pro
# 安装扩展
CREATE EXTENSION zhparser;
CREATE TEXT SEARCH CONFIGURATION testzhcfg (PARSER = zhparser);
ALTER TEXT SEARCH CONFIGURATION testzhcfg ADD MAPPING FOR n,v,a,
i,e,l WITH simple;
```

3. 使用

接下来我们来测试一下，是不是按照我们的意愿来分词。

```
postgres=# SELECT to_tsvector('testzhcfg','2015 - Rails365 Inc.
All rights reserved. | 粤ICP备15004902号-2');
                                          to_tsvector

-----------------------------------------------------------------
-------------------------------------------------
 '15004902':10 '2':12 '2015':1 '365':3 'all':5 'icp':8 'inc':4 '
rails':2 'reserved':7 'rights':6 '号':11 '备':9
(1 row)
```

果然，切好词了。

还可以这样使用。

```
postgres=# SELECT * FROM ts_parse('zhparser', '2015 - Rails365 I
nc. All rights reserved. | 粤ICP备15004902号-2');
 tokid |  token
-------+----------
   101 | 2015
   117 | -
   101 | Rails
   101 | 365
   101 | Inc
   117 | .
   101 | All
   101 | rights
   101 | reserved
   117 | .
   117 | |
   106 | 粤
   101 | ICP
   118 | 备
   101 | 15004902
   110 | 号
   117 | -
   101 | 2
(18 rows)
```

既然能够中文切词，我们就可以方便地结合其他技术来实现一个带中文支持的检索系统的。

完结。

下一篇：PostgreSQL 的全文检索系统之 pg_search 实现 (四)

1. 前言1

pg_search是一个用于**PostgreSQL**全文检索的gem，它使用起来简单，功能也很强大。

以本站为例，文章都是放在articles这张表中，文章有标题和内容，即title和body，现在要对这两个做全文检索。

2. 安装和使用

安装gem

```
gem 'pg_search'
```

查看**pg_search**的readme文档就可以知道。它主要有两种使用方式，分别是Multi-search和search scopes。Multi-search是适合于网站比较复杂，例如多张表，要把多张表揉在一起，放到一张表来做查找。现在我们的网站简单，不需要这个功能，所以我们来看看search scopes的用法。

```
class Article < ActiveRecord::Base
  include PgSearch
  pg_search_scope :search_by_title_or_body, :against => [:title,
:body]
end
```

用 `rails console` 创建一些文档，之后就能
用 `Article.search_by_title_or_body` 来搜索了。

这样再结合表单就能实现一个搜索系统的。

3. 其他功能

pg_search还有其他强大的功能。我们来介绍一下。

3.1 关联(Searching through associations)

我们现在是在articles这张上做查询，那是因为我们的网站简单，但有时候是要跨表的，那也很简单。比如,article是has_many :tags的，就可以这样。

```
class Article < ActiveRecord::Base
  include PgSearch
  pg_search_scope :search_by_title_or_body,
                  :against => [:title, :body],
                  :associated_against => {
                    :tags => [:name],
                  }
end
```

可以查看log看具体做了什么操作。其实就是joins之类。

3.2 字典和中文支持(dictionary)

这篇文章PostgreSQL的全文检索系统之中文支持(三)有介绍**PostgreSQL**中文支持。

安装好那个中文插件，和pg_search结合那太简单了，指定**dictionary**就好了。

```
class Article < ActiveRecord::Base
  include PgSearch
  pg_search_scope :search_by_title_or_body,
                  :against => [:title, :body],
                  :associated_against => {
                    :tags => [:name],
                  },
                  :using => {
                    :tsearch => {:dictionary => "testzhcfg"}
                  }
end
```

3.3 权重(Weighting)

可以给需要搜索的项加上优先级，比如，标题要优先于内容。

```
class Article < ActiveRecord::Base
  include PgSearch
  pg_search_scope :search_by_title_or_body,
                  :against => {
                    :title => 'A',
                    :body => 'B'
                  },
                  :associated_against => {
                    :tags => [:name],
                  },
                  :using => {
                    :tsearch => {:dictionary => "testzhcfg"}
                  }
end
```

可以看看日志，如果有类似于"setweight"的输出，说明成功了。

3.4 前缀(prefix)

假如要搜索一个词，例如rails，但是忘了怎么拼写，只记得前两个单词，那就是ra，但输入ra时也能找到关于rails的文章，这就是前缀的作用。使用也很简单。

```
class Article < ActiveRecord::Base
  include PgSearch
  pg_search_scope :search_by_title_or_body,
                  :against => {
                    :title => 'A',
                    :body => 'B'
                  },
                  :associated_against => {
                    :tags => [:name],
                  },
                  :using => {
                    :tsearch => {:dictionary => "testzhcfg", :pr
efix => true}
                  }
end
```

3.5 否定(negation)

否定就是可以搜索不包含的内容，比如!ruby，就是不搜索ruby，其他的都搜索，一个相反过程啦。

```
class Article < ActiveRecord::Base
  include PgSearch
  pg_search_scope :search_by_title_or_body,
                  :against => {
                    :title => 'A',
                    :body => 'B'
                  },
                  :associated_against => {
                    :tags => [:name],
                  },
                  :using => {
                    :tsearch => {:dictionary => "testzhcfg", :pr
efix => true, :negation => true}
                  }
```

还有其他各种用法，normalization用于排序，any_word是否匹配任何一个，dmetaphone模糊匹配等。

完结。

基于**web**的客户端程序**pgweb**

pgweb是用go语言写的，基于web的PostgreSQL客户端程序。用它可以管理PostgreSQL，比如增删改查等。它的界面清爽，安装简单，使用也简单，它支持mac，linux，windows等平台，所以推荐这个工具。

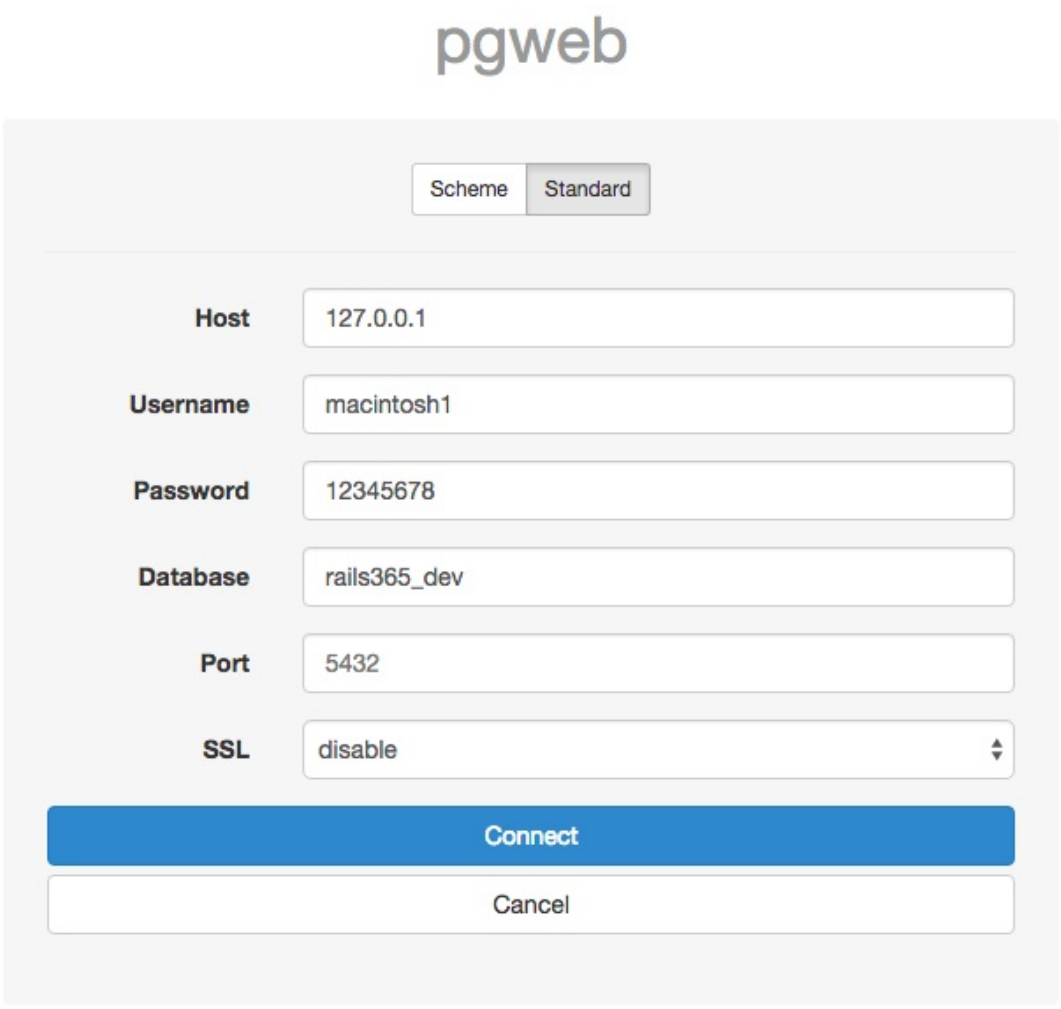
在mac下安装很简单，用brew就可以。

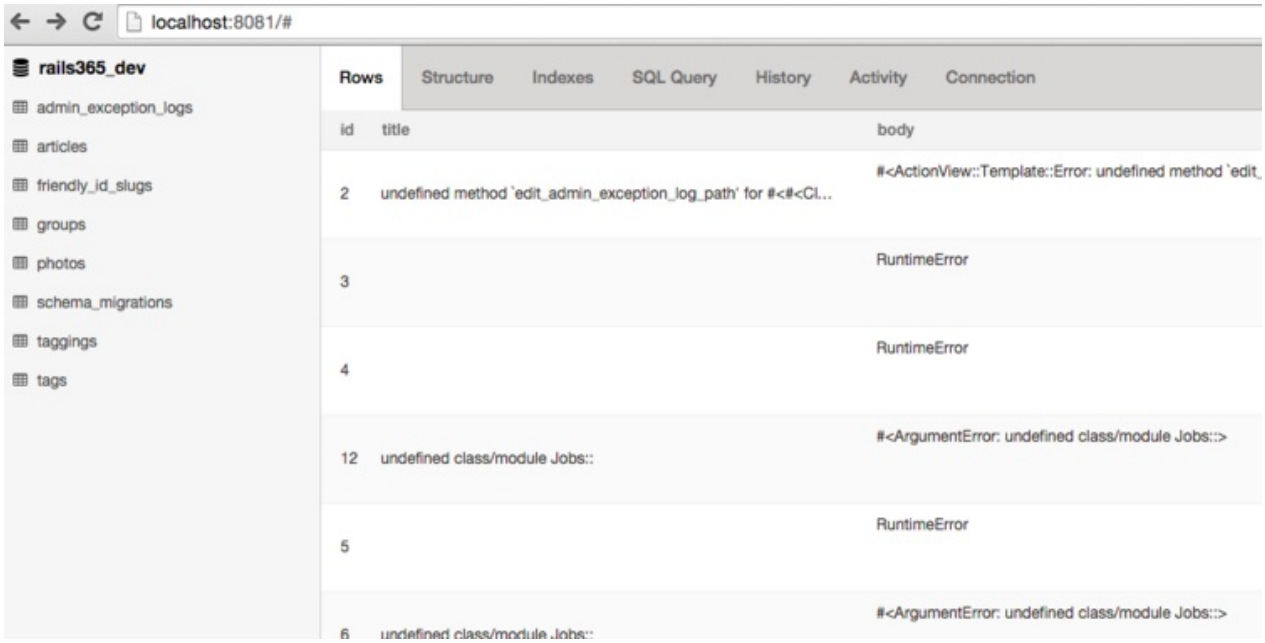
```
brew install caskroom/cask/brew-cask
brew cask install pgweb
```

要使用也很简单。

```
pgweb
```

它会自动打开浏览器。输入主机，用户名，密码，数据库就可以登录操作了。


pgweb
Scheme
Standard
Host
127.0.0.1
Username
macintosh1
Password
12345678
Database
rails365_dev
Port
5432
SSL
disable
Connect
Cancel


localhost:8081/#
rails365_dev
admin_exception_logs
articles
friendly_id_slugs
groups
photos
schema_migrations
taggings
tags
Rows
Structure
Indexes
SQL Query
History
Activity
Connection
id
title
body
2
undefined method `edit_admin_exception_log_path' for #<#<Cl...
#<ActionView::Template::Error: undefined method `edit_
3
RuntimeError
4
RuntimeError
12
undefined class/module Jobs::
#<ArgumentError: undefined class/module Jobs::>
5
RuntimeError
6
undefined class/module Jobs::
#<ArgumentError: undefined class/module Jobs::>

完结。

1. 传统方式

有很多数据或资源是这样，具有一个类型或状态属性，比如，订单有pending，approve状态，博文有草稿(draft)，出版(published)的状态，而一般来存这种数据可以选择存成字符串(string)，或整型(integer)。建议如果是中文的字符串就不要存进数据库了，不存可以避免很多问题。而大多数人是存整形，就是数字1、2、3之类，比如，1代表draft，2代表published，这样可以节约空间啊，整型肯定比字符串占用的空间小些，如果要读出1或2代表的数据，用一个常量hash来匹配就好了，比如 `STATUS_TEXT = { 1: '待处理', 2: '操盘中', 3: '已完结' }` 。

而Rails的activerecord也支持enum方法，来支持更多的判断等操作。比如

```
class Conversation < ActiveRecord::Base
  enum status: [ :active, :archived ]
end

# conversation.update! status: 0
conversation.active!
conversation.active? # => true
conversation.status  # => "active"

# 返回所有类型
Conversation.statuses # => { "active" => 0, "archived" => 1 }
```

2. PostgreSQL的枚举类型

PostgreSQL官方文档enum介绍了枚举类型和它的操作。

创建枚举类型。

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

使用只要指定TYPE的名称即可。

```
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

functions-enum这里有enum所有支持的函数。

2. 在Rails中的使用

添加枚举的列。

```
# 20151009022320_add_status_to_articles.rb
class AddStatusToArticles < ActiveRecord::Migration
  def up
    execute <<-SQL
      CREATE TYPE article_status AS ENUM ('draft', 'published');
    SQL
    add_column :articles, :status, index: true
  end

  def down
   execute <<-SQL
     DROP TYPE article_status;
   SQL
   remove_column :articles, :status
  end
end
```

在article.rb中定义enum。

```
# article.rb
class Article < ActiveRecord::Base
  enum status: {
    draft:                  'draft',
    published:                'published'
  }
end
```

假如之后有另外的值要添加的话，那也简单。用 `ALTER TYPE` 命令即可。

```
ALTER TYPE enum_type ADD VALUE 'new_value'; -- appends to list
ALTER TYPE enum_type ADD VALUE 'new_value' BEFORE 'old_value';
ALTER TYPE enum_type ADD VALUE 'new_value' AFTER 'old_value';
```

用Rails可以这样做。

```
disable_ddl_transaction!

def up
  execute <<-SQL
    ALTER TYPE article_status ADD VALUE IF NOT EXISTS 'archived'
 AFTER 'published';
  SQL
end
```

查看数据库的所有枚举类型可以这样。

```
SELECT n.nspname AS enum_schema,
       t.typname AS enum_name,
       e.enumlabel AS enum_value
  FROM pg_type t
      JOIN pg_enum e ON t.oid = e.enumtypid
      JOIN pg_catalog.pg_namespace n ON n.oid = t.typnamespace
```

完结。

ltree介绍

ltree是PostgreSQL的一个扩展插件，即extension，使用它可以实现树型结构，而且还支持索引和丰富的查询。

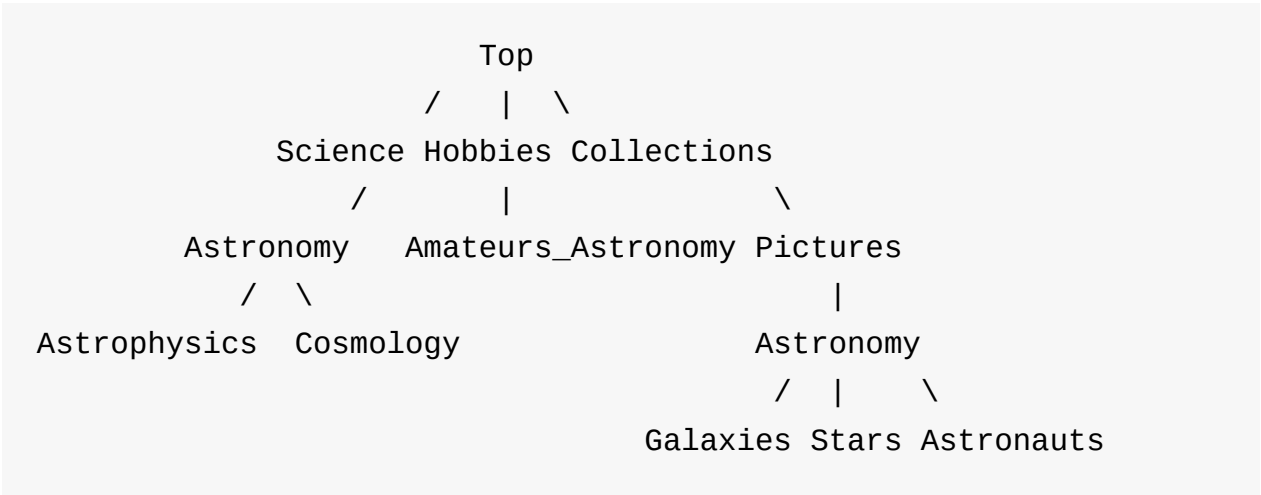
ltree官方文档给出了详细的解释。

它的概念很简单，打个比方，比如，我们要存一个树型菜单，不限定级数，有祖先(根)，根子节有子节点，子节点又有子节点，以此类推，形成一颗树。假如我们存公司的数据，它的表名叫companies，有个字段叫name，存的是分公司的名称。有一个很简单的方法，来实现这种树型结构，只要在companies表中增加一个字段parent_id即可，它存的是父节点的id，以此来找到父节点，根节点的parent_id为null，其他节点都为父节点的id。这种方式是可以的，它也能查询到所有的节点，由于存有parent_id，它找子节点，父节点，根节点都很简单，若要找其他节点，只能通过遍历了，效率较低。而且，每次添加节点，都要查找父节点的id，即parent_id，这样也不够直观和灵活。

而ltree是在数据库级别支持的树型结构。它支持丰富的查询。

ltree使用

我们来演示一下搭建这样的树型结构。



也会演示它强大的查询方法。

首先开启ltree扩展。

```
sudo -u postgres psql
CREATE EXTENSION IF NOT EXISTS ltree;
```

创建数据库表。表名为test，字段名为path，类型指定为ltree。

```
CREATE TABLE test (path ltree);
```

插入数据。

```
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Sta
rs');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Gal
axies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Ast
ronauts');
```

因为要查询，给数据表加上索引，索引有两种，分别是btree和gist，GiST支持的操作符更为丰富些。具体可看官方文档。

```
CREATE INDEX path_gist_idx ON test USING gist(path);
CREATE INDEX path_idx ON test USING btree(path);
```

现在整张表的结果是这样。

```
rails365_pro=# select * from test;
                     path
------------------------------------------------
 Top
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies
 Top.Hobbies.Amateurs_Astronomy
 Top.Collections
 Top.Collections.Pictures
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(13 rows)
```

接下来演示查询方法。

先来演示第一个，再来介绍语法。

```
rails365_pro=# SELECT path FROM test WHERE path ~ '*.Astronomy.*
';
                     path
------------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)
```

这样会查找所有包含Astronomy的项。

根据官方的解释。语法大约是这样的。

```
SELECT path FROM test WHERE ltree 操作符 lquery;
```

`ltree` 就是要查找的字段名。 `~` 就是操作符，官方列出了所有支持的操作符，也给了解释。 `lquery` 是表示被匹配的正则表达式的字符串。

```
rails365_pro=# SELECT path FROM test WHERE path <@ 'Top.Science'
;
                path
-------------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

`<@` 的意思是"is left argument a descendant of right (or equal)?"，就是返回指定元素的后代啦，返回的结果正是我们期待的。

只要懂得了语法，ltree支持的所有操作符只要看官方文档的解释就可以使用了。

ltree还支持函数。用于选择和组合返回我们想要的结果。比如:

```
rails365_pro=# select * from test WHERE path <@ 'Top.Science.Ast
ronomy';
                path
-------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)

rails365_pro=# SELECT subpath(path,0,2)||'Space'||subpath(path,2
) FROM test WHERE path <@ 'Top.Science.Astronomy';
                ?column?
-----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

subpath的语法是这样的。`subltree(ltree, int start, int end)` 中 `start` 是起始位置(从0开始算)，`end` 是结束位置，但不包含结束位置。`subpath(path,0,2)` 返回的就是第一个元素和第二个,即 `Top.Science.` 。

还有其他函数，看官方文档的解释就好了。

ltree介绍完了。

ltree_hierarchy的使用

ltree_hierarchy是一个ruby的gem，它实现了PostgreSQL的ltree的功能。提供了简单的方法来实现树型结构的功能。

先安装**ltree_hierarchy**这个gem。

```
gem 'ltree_hierarchy'
```

然后执行 `bundle` 。

我们用已存在的articles这张表来演示。

```
# 20151010060005_add_ltree_to_articles.rb
class AddLtreeToArticles < ActiveRecord::Migration
  def change
    enable_extension "ltree"
    add_column :articles, :parent_id, :integer, index: true
    add_column :articles, :path, :ltree
  end
end
```

执行 `rake db:migrate` 。

在app/models/article.rb文件中添加下面那行。

```
class Article < ActiveRecord::Base
  has_ltree_hierarchy
end
```

```
  root     = Article.create!(name: 'UK')
  child    = Article.create!(name: 'London', parent: root)
  subchild = Article.create!(name: 'Hackney', parent: child)

  root.parent   # => nil
  child.parent # => root
  root.children # => [child]
  root.children.first.children.first # => subchild
  subchild.root # => root
```

`parent_id` 存的是交节点的id，像上述所说的， `path` 存的是以"."分隔的id。

除了 `root` ， `children` ， `parent` ，ltree_hierarchy还实现了其他查询方法。比如查叶子节点，查后代所有节点之类的。具体的查看官方的readme文件就好了。

完结。

这节来介绍**PostgreSQL**的一个特性，叫"Window Functions"，这个功能有点类似于"group by"，它很强大，能够实现意想不到的功能。而且这个功能不是所有数据库系统都有的，例如**MySQL**就没有。它结合统计图来用更为强大。

它的官方定义是这样的："A window function performs a calculation across a set of table rows that are somehow related to the current row. "。

说那么多没什么用，直接看例子。

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY de
pname) FROM empsalary;
```

```
depname   | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

假如把上面的sql语句中的"OVER (PARTITION BY depname)"改成"GROUP BY depname"的话，结果就是只有三条记录，它会根据depname(develop、personnel、sales)合并成三条的。

```
depname   | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
(10 rows)
```

在某些场合下，这种肯定没有**"Window Functions"**，因为salary和empno只有一个了。假如我们需要输出salary和empno的话，只能再查一次，然后用程序循环出来，只能这样组合了。在实际的开发中，是遇到过这种问题的。而**PostgreSQL**默认就提供了**"Window Function"**机制来解决这一问题，很方便。

还支持排序。

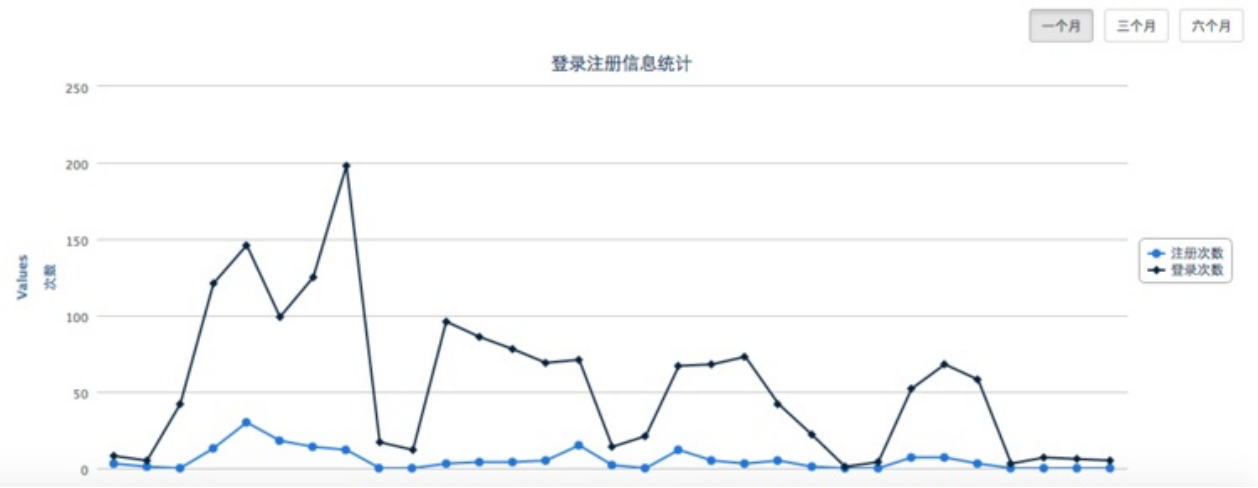
```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

```
 depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

在**Rails**中可以这样用

```
Article.find_by_sql("SELECT *, rank() OVER (ORDER BY group_id DE
SC) FROM articles")
```

结合一些图表统计的库，比如hightcharts，可以实现类似这样的效果。


一个月
三个月
六个月
登录注册信息统计
250
200
150
100
50
0
Values
次数
注册次数
登录次数

PostgreSQL支持最丰富的数据类型，更是最具有Nosql的特性。本节的内容会基于官方的active_record_postgresql，进行扩展和完善。

1. 二进制类型

PostgreSQL可以直接存储二进制的文件。例如图片、文档，视频等。

```
rails g model document payload:binary

# db/migrate/20140207133952_create_documents.rb
class CreateDocuments < ActiveRecord::Migration
  def change
    create_table :documents do |t|
      t.binary :payload

      t.timestamps null: false
    end
  end
end

# Usage
data = File.read(Rails.root + "tmp/output.pdf")
Document.create payload: data
```

2. 数组

其他数据库系统也是可以存数组的，不过还是最终以字符串的形式存的，取出和读取都是用程序来序列化。假如不用字符串存，那就得多准备一张表，例如，一篇文章要记录什么人收藏过。就得多一张表，每次判断用户是否收藏过，就得查那张表，而数据以冗余的方式存在数据中，就是把user_id存进去一个字段，这样就大大方便了。**PostgreSQL**默认就支持数据的存取，还支持对数据的各种操作，比如查找等。

```
# db/migrate/20140207133952_create_books.rb
create_table :books do |t|
  t.string 'title'
  t.string 'tags', array: true
  t.integer 'ratings', array: true
end
add_index :books, :tags, using: 'gin'
add_index :books, :ratings, using: 'gin'

# app/models/book.rb
class Book < ActiveRecord::Base
end

# Usage
Book.create title: "Brave New World",
            tags: ["fantasy", "fiction"],
            ratings: [4, 5]

## Books for a single tag
Book.where("'fantasy' = ANY (tags)")

## Books for multiple tags
Book.where("tags @> ARRAY[?]::varchar[]", ["fantasy", "fiction"]
)

## Books with 3 or more ratings
Book.where("array_length(ratings, 1) >= 3")
```

PostgreSQL还支持对array的各种操作，官方文档给了详细的解释。

```
# 返回数组第一个元素和第二个元素不相同的记录
Book.where("ratings[0] <> ratings[1]")

# 查找第一个tag
Book.select("title, tags[0] as tag")

# 返回数组的维数
Book.select("title, array_dims(tags)")
```

像类似array_dims的操作符，官方这篇文章functions-array有详细的记录。

比如，把数组进行类似join的操作。

```
Book.select("title, array_to_string(tags, '_')")

SELECT title, array_to_string(tags, '_') FROM "books";
      title      | array_to_string
-----------------+-----------------
 Brave New World | fantasy_fiction
(1 row)
```

3. Hstore

Hstore是PostgreSQL的一个扩展，它能够存放键值对，比如，json，hash等半结构化数据。一般的数据库系统是没有这种功能，而这种需求是很常见的，所以说，PostgreSQL是最具Nosql特性的。只要前端通过js提交一些hash或json，或者通过form提交一些数据，就能直接以json等形式存到数据库中。例如，一个用户有1个，0个，或多个联系人，如果以关系型数据库来存的话，只能多建立一张表来存，然后用has_many，belongs_to来处理。而Hstore就是以字段的形式来存，这就很方便了。

```
# 开启扩展
rails365_dev=# CREATE EXTENSION hstore;

# 或者

class AddHstore < ActiveRecord::Migration
  def up
    execute 'CREATE EXTENSION IF NOT EXISTS hstore'
  end

  def down
    execute 'DROP EXTENSION hstore'
  end
end

# 或者
class AddHstore < ActiveRecord::Migration
  def change
    enable_extension 'hstore'
  end
end

rails g model profile settings:hstore

# Usage
Profile.create(settings: { "color" => "blue", "resolution" => "8
00x600" })

profile = Profile.first
profile.settings # => {"color"=>"blue", "resolution"=>"800x600"}

profile.settings = {"color" => "yellow", "resolution" => "1280x1
024"}
profile.save!
```

像array一样，Hstore也是支持很多操作的，官方文档hstore给了详细的描述。

比如：

```
rails365_dev=# SELECT  "profiles".settings -> 'color' FROM "prof
iles"
;
 ?column?
----------
 yellow
 blue
(2 rows)

rails365_dev=# SELECT  "profiles".settings ? 'color' FROM "profi
les"
;
 ?column?
----------
 t
 t
(2 rows)

rails365_dev=# SELECT  hstore_to_json("profiles".settings) FROM
"profiles"
;
                        hstore_to_json
-------------------------------------------------------------
 {"color": "yellow", "resolution": "1280x1024"}
 {"color": "blue", "resolution": "[\"800x600\", \"750x670\"]"}
(2 rows)

rails365_dev=# SELECT  "profiles".settings -> 'color' FROM "prof
iles"
 where settings->'color' = 'yellow';
 ?column?
----------
 yellow
(1 row)
```

更多详细只要查看官文文档就好了。

关于Hstore有一个gem是activerecord-postgres-hstore，这个gem提供了很多关于Hstore的查询方法。

using-postgres-hstore-rails4这篇文章介绍了Hstore的用法。

其他的特性，“JSON”、"Range Types"、“Enumerated Types”、“UUID”等就不再赘述，要使用时，结合官方文档查看即可。

完结。

1. Schema

PostgreSQL-schemas这里介绍了Schema的概念和用法。

简而言之，Schema是一种命名空间，它可以用来隔离表，隔离数据，又避免了连接多个数据库。在同一个数据库下，不同的Schema可以有相同名字的表(table)，每个数据库默认都有public这个Schema，还可以对Schema进行权限的限制等。对Schema的操作也很简单，比如，创建 `CREATE SCHEMA myschema;` 、 `DROP SCHEMA myschema;` ，要对Schema下的表进行操作只需要加上前缀就好了。比如， `CREATE TABLE public.products ( ... );` 。

还有个比较重要的东西要说，那就是search_path。它是表的搜索路径，相当于linux系统的$PATH变量，找可执行程序的，不过它是找表的，一般来说，找表可以加Schema，如果不加就找search_path指定的Schema，查看search_path可以用 `SHOW search_path;` ，而使用 `SET search_path TO myschema,public;` 可以更改搜索路径。

Schema特别适合于以下几种场合。

- 管理员管理自己所属分公司的数据。
- 隔离不同幼儿园的数据。

其实acts_as_tenant就可以实现类似上面的效果，不过acts_as_tenant相对简单，只是代码级加上少量数据级的控制，而Schema就是数据库级别的真正数据隔离，也是原生支持，所以更好，不过只支持PostgreSQL数据库。

2. multi-tenancy

我们使用apartment这个gem来实现多Schema的系统，也叫做multi-tenancy。

2.1 安装

添加下面这一行到Gemfile文件。

```
gem 'apartment'
```

生成配置文件config/initializers/apartment.rb。

```
bundle exec rails generate apartment:install
```

2.2 创建新的Tenants

使用ruby代码来创建PostgreSQL Schema是这样的。在 `rails console` 中执行下面这行。

```
Apartment::Tenant.create('tenant_name')
```

执行这行命令会有很多输出，其实它首先会创建一个PostgreSQL Schema叫tenant_name，然后会把以前的表也在这个叫tenant_name的Schema下生成一遍。

我们来验证一下。用 `rails db` 进入数据库。

执行 `\dn` 来查看所有的Schema。

```
rails365_pro=# \dn
    List of schemas
    Name     |  Owner
-------------+----------
 public      | postgres
 tenant_name | postgres
(2 rows)
```

使用 `\dt` 来查看所有tenant_name下的表(table)。

```
rails365_pro=# \dt tenant_name.*;
                   List of relations
   Schema    |          Name          | Type  |  Owner
-------------+------------------------+-------+----------
 tenant_name | admin_exception_logs | table | postgres
 tenant_name | articles             | table | postgres
 tenant_name | friendly_id_slugs    | table | postgres
 tenant_name | groups               | table | postgres
 tenant_name | photos               | table | postgres
 tenant_name | schema_migrations    | table | postgres
 tenant_name | taggings             | table | postgres
 tenant_name | tags                 | table | postgres
(9 rows)
```

有Apartment::Tenant.create这个命令，结合数据库维护起整个Schema就很灵活了，比如，`School.create`的时候也顺便`Apartment::Tenant.create :school`。

2.3 切换Tenants

Schema避免了连接不同的数据库，但也是要切换默认的Tenants。使用`Apartment::Tenant.switch!`。

```
# 先切换到public下查看数据
Apartment::Tenant.switch!('public')
Article.all

# 切换到tenant_name下验证数据
Apartment::Tenant.switch!('tenant_name')
Article.all
```

有`create`命令和`switch!`命令，结合起来再灵活地配合application_controller.rb等文件就可以很好地实现multi-tenancy系统了。原理就是先用`create`创建好Schema，然后到查数据或资源地方`switch!`，切换到正确的Schema来查就好，而这个可以用controller中的before_action之类的方法搞定，设定一个当前的tenant即可。怎么来设定当前的tenant，那就可以结合传过来的参数或子域名等来处理了。

2.4 删除Tenants

有创建就有删除的，那就是drop，用这个命令可以来维护Schema。

```
Apartment::Tenant.drop('tenant_name')
```

2.5 通过子域名来切换Tenants

默认情况下，是通过子域名来切换Tenants的，这个可以通过配置文件 `config/initializers/apartment.rb` 查看到。

```
require 'apartment/elevators/subdomain'
Rails.application.config.middleware.use 'Apartment::Elevators::Subdomain'
```

意思就是，假如是foo.example.com，就会切换到foo这个Schema，如果是bar.example.com，就会切换到bar这个Schema，是这个gem提供的功能，是自动切换的，如果不需要这个功能也可以注释掉上面两行代码即可。

更加详细的功能可以看apartment的github官方readme文档或查看其源码。

完结。

1. 介绍

PostgreSQL的分区是建立在继承的基础上的，所以先来讲讲继承。

2. 继承

继承指的是表的继承，就是一个表继承自另一个表，字段也继承自父表，跟面向对象的概念差不多。因为有时候几张表就是具有差不多的属性或字段，唯一有区别的就是其中一两个字段，这个时候可以用继承来简化操作和管理。

比如，如果不用继承，会像下面这样处理的。

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,     -- (in ft)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int      -- (in ft)
);

CREATE VIEW cities AS
  SELECT name, population, altitude FROM capitals
    UNION
  SELECT name, population, altitude FROM non_capitals;
```

要查找那两张表就得使用union语句。

而使用继承就是这样处理的。

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int      -- (in ft)
);

CREATE TABLE capitals (
  state      char(2)
) INHERITS (cities);
```

这样就创建了两张表，插入(insert)数据之后就可以用select来查询的。

3. 分区

PostgreSQL-partitioning对分区作了完整的描述。

分区是数据库的一种设计实现方法。我们知道，当一张表的数据越来越多时，假如到了上亿条或几十亿条记录，对这张表的操作都会比较慢，比如，查询，更改等。而分区技术就是把这一张大表分成几个逻辑分片。分区之后有很多好处：

- 单个分区表的索引和表都变小了，可以保持在内存里面，适合把热数据从大表拆分出来的场景；
- 对于大范围的查询，大表可以通过索引来避免全表扫描，但是如果分区的话，可以使用分区的全表扫描；
- 大批量的数据导入或删除，删除大量的数据使用DELETE会很慢，可是如果使用分区表，直接drop或truncate整个分区表即可；

而分区技术就是基于上面所提的继承技术来实现的。

PostgreSQL实现了两种分区。

- `Range Partitioning`：比如数值范围，时间范围等。
- `List Partitioning`：按照固定的值。

4. 实战分区

其中一种实现分区的方法是基于继承并配合触发器来实现。

先创建母表，它其实是一张只有数据结构的表。

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

创建分区表，用时间范围来分区。

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '200
6-03-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '200
6-04-01' )
) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '200
7-12-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '200
8-01-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '200
8-02-01' )
) INHERITS (measurement);
```

check指定的是约束条件，按照时间来规定范围。

按照需要可以添加索引。

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m0
2 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m0
3 (logdate);
...
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m1
1 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m1
2 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m0
1 (logdate);
```

当执行 `INSERT INTO measurement ...` 时，为了让数据插入到正确的分区表上，我们需要创建触发器来实现这个逻辑。

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
            NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the measurement
_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE PROCEDURE measurement_insert_trigger();
```

这样就OK了。

另外来实现同样插入逻辑的方式是用rule(规则)。

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-0
1' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-0
1' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

具体的详细可以阅读官方文档。

下一篇: PostgreSQL的表的继承和分区之pg_partman(二)

完结。

1. 介绍

在这一篇文章PostgreSQL的表的继承和分区之介绍(一)介绍了表的继承和分区的概念以及如何使用的方法。

首先是分区建立在继承的基础上，先创建母表，通过约束条件创建子表，再通过创建触发器保证数据能插入到相应的子表中。这一切都是需要我们手动来创建的。

而pg_partman把这一些手动的过程全封装到函数中，通过函数的调用即可方便创建与维护，并且避免了手工创建引入错误。

2. 安装

下载源代码安装。

```
git clone git@github.com:keithf4/pg_partman.git
cd pg_partman
make install
```

进入psql安装pg_partman扩展。

```
CREATE SCHEMA partman;
CREATE EXTENSION pg_partman SCHEMA partman;
```

3. 使用

设置partman为当前的表搜索路径。

```
set search_path to partman;
```

先创建一张母表。

```
CREATE schema test;
CREATE TABLE test.part_test (col1 serial, col2 text, col3 timest
amptz NOT NULL DEFAULT now());
```

这张表很简单，只有三列，最后一列是时间。

```
SELECT partman.create_parent('test.part_test', 'col3', 'time', '
daily');
```

我们先来查看创建成功后结果，使用下面的命令。

```
partman=# \d+ test.part_test
```

结果是：

```
                                                                Table
"test.part_test"
 Column |           Type           |                           M
odifiers                           | Storage  | Stats target | D
escription
--------+--------------------------+-----------------------------
-----------------------------------+----------+--------------+--
-----------
 col1   | integer                  | not null default nextval('t
est.part_test_col1_seq'::regclass) | plain    |              |
 col2   | text                     |
                                   | extended |              |
 col3   | timestamp with time zone | not null default now()
                                   | plain    |              |
Triggers:
    part_test_part_trig BEFORE INSERT ON test.part_test FOR EACH
 ROW EXECUTE PROCEDURE test.part_test_part_trig_func()
Child tables: test.part_test_p2015_10_10,
              test.part_test_p2015_10_11,
              test.part_test_p2015_10_12,
              test.part_test_p2015_10_13,
              test.part_test_p2015_10_14,
              test.part_test_p2015_10_15,
              test.part_test_p2015_10_16,
              test.part_test_p2015_10_17,
              test.part_test_p2015_10_18
```

由上可知，总共创建了一个叫part_test_part_trig_func的触发器和九张子表(part_test_p2015_10_11到part_test_p2015_10_18)。

可以用下面的命令来查看触发器part_test_part_trig_func的内容。

```
partman=# select prosrc from pg_proc where proname='part_test_pa
rt_trig_func';
```

今天的时间是：

```
➜  rails365 git:(master) date
Wed Oct 14 21:46:30 HKT 2015
```

也就是14号之前有四张表，14号之后有四张表。

当然这些规则可以由我们自己自定义的，只要我们熟悉了create_parent的用法就可以。

在上面的例子中。

```
SELECT partman.create_parent('test.part_test', 'col3', 'time', '
daily');
```

create_parent的第一个参数接的是母表的名称，第二个接的是要分区的列的名称，第三个表示按照时间来分区，第四表示是按照时间上的每天来分区。这也解释了为什么会出现上面的按时间顺序命名的分区表的情况。

关于pg_partman的更多内容可以查看官方的这篇文档pg_partman.md。

关于pg_partman的更多示例可以查看官方的这篇文档pg_partman_howto.md。

另外，ruby也有相关的分区工具，就是用ruby来生成分区的指令。https://github.com/ankane/pgslice

完结。

1. 消息队列的简介

什么是消息队列呢？队列就是排队，就像在银行办理业务排队一样，排在最前面的先处理，后面的后处理，按照顺序来，先进先出。这个队列可以是程序，可以是数据，也可以是任务，是任何你可以存储的东西。消息队列就是给队列传递消息。这么来说，打个比方，我们在一个网站上注册了账号，系统可能会给你发送一封注册邮件，同时在页面上提示你"稍等几分钟后会收到一封邮件"，发邮件这个事是通过操作系统的调用，例如linux的sendmail，或者接口来发送的，发邮件是通过排队来发的，先到的先发，假如很多邮件等着发，那就得像银行那样排队了，所以未必就能实时，总有延迟。总结来说，发邮件这个事是有延迟的，是需要等待之后用户才能收到邮件的。然而，这种延迟对用户的体验还有操作并不影响啊。在网站上的其他应用他还是照样用，没有任何影响。对这种对用户没直接影响或者有延迟的任务就可以扔到消息队列处理。所以，发邮件，发短信，捕获异常等任务都可以扔到消息队列。也就是消息队列是独立于web进程的另一个进程，因为它有可能耗时很长的，所以要另开一个进程来处理，对web进程没有任务影响，用户还是照常访问网站。这么来说，假如网站有一个需要扔的消息队列叫A，但用户触发了A，就把A扔到消息队列，这时给用户感觉是这个A任务是一瞬间完成，其实它是给消息队列那个进程发送了消息，可能跟它说，我要发短信，就把发短信这个指令，加上短信的内容一并传给消息队列的进程，消息队列收到消息后，就把这个任务放到队列中进行排队，因为前面还有一堆任务没处理，所以要慢慢处理，轮到A的时候才处理A，由于A是耗时的动作所以就慢慢处理就好，反正对用户不太影响。

前面说到，消息队列的进程要把任务放入队列中，由于有很多任务，需要排队，所以这些任务是需要存储起来的。在ruby中，有很多gem可以实现后台的消息队列，但它们的存储方式有区别，比如delayed_job就是用数据库(MySQL，Sqlite，PostgreSQL等)来存的，它会先让你创建表，如果有任务进来，就会插入到表中作为一条记录，要处理的时候就会取出这条记录。像resque和sidekiq就是用redis来存储数据的，redis是存储在计算机的内存中的。比较一下，就知道resque和sidekiq在存储方式上比delayed_job有优势，而delayed_job的好处是能直接利用数据库，不用额外安装redis。

消息队列是另外的一个进程，任务进入消息队列中，一个接一个的处理，也就是说，A进程在被处理时，必须等前面的任务被处理完才能轮到它。这种方式体现在delayed_job和resque中。sidekiq的处理方式是多线程的，它是基于celluloid的，用Actor作为并发模型，它能同时处理多个任务。

值得一提的是Ruby on Rails从4.2开始加上了active_job。因为有各种各样的消息队列的解决方案，active_job就是提供了统一的接口和调用，要用到消息队列还是会用到上面提到的几个。这个东西就像activerecord一样，要指定数据库那也是很简单的，只要换相应的gem和改配置文件就行了，而active_job也正是这样。

不过，这一篇文章不会详说上面的三种消息队列的实现，只会说到特用于PostgreSQL的消息队列queue_classic

2. PostgreSQL的listen/notify

queue_classic是基于PostgreSQL的listen/notify来实现的，列队在等任务进来就是用的listen，把任务放入队列就是notify。

PostgreSQL的listen/notify，也就是一种消息的订阅/发布模式，也就是类似那种生产者/消费者模式。这种模式很常见，例如redis的pub/sub模式、rails的Notifications组件。

懂了PostgreSQL的listen/notify，也就等于懂了其他的订阅/发布模式。

它很简单，就相当于一种广播机制，比如，你定阅杂志，还有其他人也订阅了，这个过程就叫listen，也就是监听，等杂志有更新了，或者有新的杂志出来，它就会广播，就会送一份给订阅杂志的人，这个过程就是notify，也就是通知。

listen/notify的使用很简单。

首先是listen(监听)，只接监听的通道的名称，这个名称自己定义。

```
rails365_pro=# LISTEN virtual;
LISTEN
```

这个时候可以直接执行notify。

```
rails365_pro=# NOTIFY virtual;
NOTIFY
Asynchronous notification "virtual" received from server process
 with PID 4996.
```

表示监听的通道已知收到消息了。

还可以传参数。

```
rails365_pro=# NOTIFY virtual, 'This is the payload';
NOTIFY
Asynchronous notification "virtual" with payload "This is the pa
yload" received from server process with PID 4996.
```

也可以结合sql语句来使用。

```
LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
```

只要连接到同一个数据库的所有session都会接到监听通道传过来的信息。

可以尝试另开一个psql进程。然后notify，再回到之前的psql执行listen就可以测试的，如果显示正确的pid就成功的。

下一章: PostgreSQL的listen/notify之queue_classic(二)

完结。

1. 介绍

queue_classic是一个ruby的gem，用来实现PostgreSQL的消息队列。它基于PostgreSQL的listen/notify，有很多接口，使用起来比较简单。

如果你有使用过sidekiq，resque，delayed_job，就会发现基本每个这种消息队列的gem的使用方法都是类似的，里面的概念也是差不多，也就是说，学会了queue_classic就等于会其中三种，只要掌握思想就好了。

2. 安装

假如我们已经有一个rails项目了。

在Gemfile添加下面这行。

```
gem "queue_classic", "~> 3.0.0"
```

执行 `bundle install`

正如我们上面所说的，任务是需要存储的。所以要创建相应的表来存。

```
# 创建queue_classic_jobs表
rails generate queue_classic:install
# username替换为你自己的数据库的用户名，password是数据库的密码，rails365_dev是数据库名
export QC_DATABASE_URL="postgres://username:password@localhost/rails365_dev"
bundle exec rake db:migrate
```

3. 测试

我们先在 `rails console` 里测试。

前面说过，消息队列是跑在一个进程里的。所以要启动那个进程。

```
bundle exec rake qc:work
```

你会发现delayed_job，sidekiq也是差不多的启动方法。

还可以指定队列来启动。

```
QUEUES="priority_queue,secondary_queue" bundle exec rake qc:work
```

启动好后，我们进入 `rails console` 中，执行下面这行语句。

```
➜  rails365 git:(master) ✗ rails c
Loading development environment (Rails 4.2.3)
2.2.2 :001 > QC.enqueue("Kernel.puts", "hello world")
nil
```

你会在进程里看到类似这样的输出，就说明成功了。

```
➜  rails365 git:(master) ✗ bundle exec rake qc:work
hello world
```

QC.enqueue就是后面的命令加上参数作为任务push到队列中。

上面只是个最简单的例子，还有可以在指定时间，指定队列来执行，具体更为详细的命令要看官方的readme文档。

如果需要调试或查看日志，可以开启调试功能，有两种不同的日志，分别是：

```
export QC_MEASURE="true"

# or

export DEBUG="true"
```

在运行 `rake qc:work` 之前运行，具体的效果，尝试下就知道的。

4. 在rails中使用

以本站为例，是一个放博客的网站，文章是存放在articles这张表，当时为了查看哪篇文章最受欢迎，就在articles存了一个字段叫visit_count，但每次用户查看文章时，就会往这个字段加1。这个动作是用rails的ActiveSupport::Notifications配合sidekiq的消息队列来做的，现在要改成用queue_classic来做。

我们来看下相关的代码。

```
# app/workers/update_article_visit_count_worker.rb
class UpdateArticleVisitCountWorker
  include Sidekiq::Worker
  def perform(article_id)
    logger.info 'update article visit count begin'
    @article = Article.find(article_id)
    @article.visit_count += 1
    @article.save!(validate: false)
    logger.info 'update article visit count end'
  end
end
```

在哪里调用呢，我们是结合ActiveSupport::Notifications来做的，这个先不管，你也可以在articles_controller的show action直接调用。

```
# config/initializers/notification.rb
ActiveSupport::Notifications.subscribe "process_action.action_co
ntroller" do |name, started, finished, unique_id, payload|
  Rails.logger.info payload
  if payload[:controller] == "ArticlesController" && payload[:ac
tion] == "show"
    UpdateArticleVisitCountWorker.perform_async(payload[:params][
"id"]) if payload[:params]["id"].present?
  end
end
```

上文提过，queue_classic主要是利用 `QC.enqueue` 这条命令把任务push到队列中的。只需要把这行 `UpdateArticleVisitCountWorker.perform_async(payload[:params]["id"]) if payload[:params]["id"].present?` 改成我们需要的就可以了。

把增加visit_count的值的逻辑移动model中去，然后在 `ActiveSupport::Notifications.subscribe` 用 `QC.enqueue` 中调用就好了。

改造之后是这样的。

```
# app/models/article.rb
class Article < ActiveRecord::Base
  def self.update_article_visit_count(article_id)
    article = Article.find(article_id)
    article.visit_count += 1
    article.save!(validate: false)
  end
end
```

```
# config/initializers/notification.rb
ActiveSupport::Notifications.subscribe "process_action.action_co
ntroller" do |name, started, finished, unique_id, payload|
  Rails.logger.info payload
  if payload[:controller] == "ArticlesController" && payload[:ac
tion] == "show"
    QC.enqueue "Article.update_article_visit_count",payload[:par
ams]["id"] if payload[:params]["id"].present?
  end
end
```

现在需要重启下 `rails server` 和 `bundle exec rake qc:work` 。

重启 `rails server` 运行好 `export QC_DATABASE_URL="postgres://username:password@localhost/rails365_dev"` 这个命令。

为了让 `rake qc:work` 更能明显地看到日志信息，在运行 `rake qc:work` 前先执行 `export QC_MEASURE="true"` 。

现在可以去页面上测试的。

5. 注意事项

第一点是关于环境变量，也就是 `QC_DATABASE_URL` 、 `QC_MEASURE` 、 `DEBUG` 这三个，当部署到线上环境时，就要把这三个变量写进shell的配置文件，如比ubuntu系统，就写进~/.bashrc_profile就好了。

第二点是关于错误的任务，任务也是有可能会报错的，但是我们不知道哪个任务报错了，所以很不方便，其实官方提供了接口的，你自己可以捕获那个错误信息，捕获后就可以进行自己想要的处理了。其实错误的任务都会一直存在表queue_classic_jobs中，这样查看就好，至于那个接口，就是worker.rb中的handle_failure方法，官方readme文档也有示例，这里不再深究。

完结。

1. pgcenter

pgcenter是一个postgresql的扩展，是用c语言写的，类似top命令的监控工具。

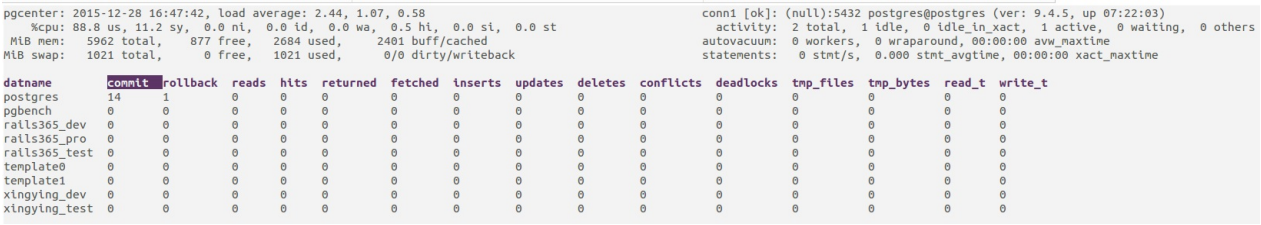
官方的readme文档有相关的安装方法。

```
$ git clone https://github.com/lesovsky/pgcenter
$ cd pgcenter
$ make
$ sudo make install
$ pgcenter
```

运行。

```
$ sudo -u postgres pgcenter
```

效果图如下：



2. pg_activity

pg_activity是一个用Python语言写的监控工具，用于监控postgresql的运行情况，和sql语句的性能。

安装。

```
$ git clone https://github.com/julmon/pg_activity
$ sudo python setup.py install
```

运行。

```
$ sudo -u postgres pg_activity -U postgres
```

效果图如下：

```
PostgreSQL 9.4.5 - yinsigan-virtual-machine - postgres@localhost:5432 - Ref.: 2s
  Size:    77.92M -     35.47K/s         | TPS:          806
  Mem.:    49.40% -      2.88G/5.82G     | IO Max:      2182/s
  Swap:     0.00% -      0.00B/1022.00M  | Read :       0.00B/s -      0/s
  Load:     1.18 0.78 0.64               | Write:       5.67M/s -   1451/s
                                                                                      RUNNING QUERIES
PID    DATABASE                  USER            CLIENT   CPU% MEM%   READ/s  WRITE/s     TIME+  W  IOW  Query
17271  pgbench               postgres              None    7.4  0.3    0.00B  630.23K  0.023618  N    N  UPDATE pgbench_branches SET bbalance = bbalance + 2143 WHERE bid = 1;
17272  pgbench               postgres              None    6.9  0.3    0.00B  610.53K  0.012576  Y    N  UPDATE pgbench_branches SET bbalance = bbalance + 603 WHERE bid = 1;
17277  pgbench               postgres              None    8.9  0.3    0.00B  594.77K  0.012516  Y    N  UPDATE pgbench_tellers SET tbalance = tbalance + -3403 WHERE tid = 1;
17279  pgbench               postgres              None    6.5  0.3    0.00B  638.40K  0.008133  Y    N  UPDATE pgbench_branches SET bbalance = bbalance + -2074 WHERE bid = 1;
17269  pgbench               postgres              None    7.4  0.3    0.00B  638.98K  0.006385  Y    N  UPDATE pgbench_tellers SET tbalance = tbalance + 804 WHERE tid = 10;
17275  pgbench               postgres              None    6.9  0.3    0.00B  677.49K  0.006302  Y    N  UPDATE pgbench_tellers SET tbalance = tbalance + -3371 WHERE tid = 10;
17278  pgbench               postgres              None   11.4  0.3    0.00B  732.62K  0.002917  Y    N  UPDATE pgbench_branches SET bbalance = bbalance + 4795 WHERE bid = 1;
17270  pgbench               postgres              None    7.4  0.3    0.00B  654.62K  0.000376  N    N  UPDATE pgbench_accounts SET abalance = abalance + -2456 WHERE aid = 28487;
17276  pgbench               postgres              None    8.9  0.3    0.00B  630.22K  0.000015  Y    N  UPDATE pgbench_branches SET bbalance = bbalance + 1789 WHERE bid = 1;
```

完结。