翻译了Django1.4数据库访问优化部分 5536°C

作者：the5fire | 标签： django (/tag/django) 数据库优化 (/tag/数据库优化) 文档翻译 (/tag/文档翻译) | 发布：2013-05-08 10:21 p.m.

Django数据库访问优化

- by huyang
- @手机搜狐
- date 2013-05-08

*rst*生成的*html5*在线*ppt*下载:*http://www.kuaipan.cn/file/id_12834302878348970.htm*

先做性能分析 - 两个工具

- `django.db.connection`

```
from django.db import connection

# context
print connection.queries
# content

''' result is:
[{
'time': '0.002',
'sql': u'SELECT `django_session`.`session_key`, `django_session`.`session_dat
a`, `django_session`.`expire_date` FROM `django_session` WHERE (`django_sessio
n`.`session_key` = 5584f8d708ddc2d5e32831885fc36084  AND `django_session`.`expi
re_date` > 2013-05-07 10:39:36 )'}]
'''
```

- `django_debug_toolbar` link (https://github.com/django-debug-toolbar/django-debug-toolbar/)

标准的数据库优化技巧

- Indexes, 分析应该添加什么样的索引,使用 `django.db.models.Field.db_index`
- 使用对应的字段类型

```
title = models.CharField(max_length=100, blank=True, db_index=True, verbose_nam
e=u'标题')
```

理解QuerySets

理解QuerySet的求值过程

- QuerySets是惰性的

```
news_list = News.object.all()
# 此时并未执行数据库查询
print news_list   # 用时方执行查询操作
```

- 何时它们被执行.

```
# 用时方执行查询操作
print news_list
```

- 数据如何被缓存

```
#  这样的QuerySet没有被缓存
print([e.headline for e in Entry.objects.all()])
print([e.pub_date for e in Entry.objects.all()])

# 这么做
entries = Entry.objects.all()
print([e.headline for e in entries])
```

理解被缓存的属性

- QuerySet 会被缓存

- 不可被调用的属性会被缓存

```
>>> news = News.objects.get(id=1)
>>> news.channel   # 此时的channel对象会从数据库取出
>>> news.channel   # 这时的channel是缓存的版本，不会造成数据库访问
```

- 方法的调用每次都会触发数据库查询

```
>>> news = News.objects.get(id=1)
>>> news.authors.all()   # 执行查询
>>> news.authors.all()   # 再次执行查询
```

注意

- 模板系统不允许使用括号，但它会自动调用可被调用的属性
- 自定义的属性需要由你来实现缓存。

使用with模板标签

在模板中使用QuerySet缓存，需要使用with标签

使用iterator()

获取大量数据时

```
news_list = News.objects.filter(title__contains=u'违法')
for news in news_list.iterator():
    print news
```

让数据库做它自己的工作

基本概念

- 使用 filter and exclude 在数据库层面执行过滤操作

```
news_list = News.objects.filter(title__contains=u'和谐').exclude(status=1)
```

- 使用 F() object query expressions 在同一模型中使用不同字段进行对比过滤

```
# 查询所有title和sub_title相同的数据
queryset = News.objects.filter(title=F('sub_title'))
```

- 使用 注解

```
# 给每个对象添加一个news_count的属性
cl = Channel.objects.filter(parent__id=1).annotate(news_count=Count('new
s'))
print cl[0].news_count
```

如果这些还不足以生成你需要的SQL的话，继续往下看：

使用 QuerySet.extra()

显式的执行SQL语句

```
cl = Channel.objects.filter(parent__id=1).extra(
    select={
        'another_news_count': 'SELECT COUNT(*) FROM web_news WHERE web_news.cha
nnel_id = web_channel.id'
    }
)

print cl[0].another_news_count
```

使用原生的SQL

```
cl = Channel.objects.raw('SELECT * FROM web_channel WHERE parent_id = 1')
print cl
# <RawQuerySet: 'SELECT * FROM web_channel WHERE parent_id = 1'>
for c in cl:
    print c
```

预加载数据

尽量一次加载你需要的数据

- QuerySet.select_related() , 针对foreign key 和 one-to-one

```
news = News.objects.select_related().get(id=372924135)
print news.channel # 不会访问数据库
```

- QuerySet.prefetch_related() , 1.4中存在, 和select_related()类似，针对many-to-many

不要获取你不需要的数据

使用 QuerySet.values() 和 values_list()

当只需要一个字段的值，返回list或者dict时，使用

- values

```
news_list = News.objects.values('title').filter(channel__id=1)
print news_list
# [{'title': ''}, ...]
```

- values_list

```
news_list = News.objects.values_list('title').filter(channel__id=1)
print news_list
# [('新闻标题',),('新闻标题', ) ...]
```

使用 QuerySet.defer() 和 only()

- QuerySet.defer() 来延迟加载某字段，加载时会产生额外查询

```
news_list = News.object.defer('title').all()
n = news_list[0]
print n.title   # 会产生额外的查询语句
```

- QuerySet.only() 只加载某字段，之后读取任何属性都会产生查询

使用 QuerySet.count()

如果你只是想要获取有多少数据，不要使用 len(queryset) 。

```
nl = News.objects.filter(channel__id=2)
nl.count()
# SELECT COUNT(*) FROM `web_news` WHERE `web_news`.`channel_id` = 2 ; 'time':
'0.014'

len(nl)
# 'time': '0.422'
```

使用 QuerySet.exists()

如果你只是想要知道是否至少存在一个结果，不要使用 if querysets 。

不要过度使用 count() 和 exists()

比如，假设有一个Email的model，有一个 body 的属性和一个多对多关系的User 属性，下面的模板代码是最优的：

```
{% if display_inbox %}
  {% with emails=user.emails.all %}
    {% if emails %}
      <p>You have {{ emails|length }} email(s)</p>
      {% for email in emails %}
        <p>{{ email.body }}</p>
      {% endfor %}
    {% else %}
      <p>No messages today.</p>
    {% endif %}
  {% endwith %}
{% endif %}
```

它是最优的是因为：

1. 因为QuerySet是惰性的，如果 'display_inbox' 是False的话，这不会产生数据库 查询。
2. 使用 with 意味着我们会存储 user.emails.all 在一个变量中供后面使用，这允许被缓存以便重用。
3. {% if emails %} 其实是调用 QuerySet.__nonzero__() ，在数据库层面执行 user.emails.all() ，然后返回结果，放入缓存。
4. {{ emails|length }} 的使用将调用 QuerySet.__len__(),数据已在缓存
5. for 循环的email数据已经在缓存中了。

> - with的使用是关键
> - 每次的QuerySet.count()调用都会产生查询

使用 QuerySet.update() 和 delete()

- 批量更新使用 QuerySet.update()
- 批量删除使用 QuerySet.delete()

> 批量操作不会调用类中定义的 save() 或 delete() 方法

直接使用外键的值

获取频道ID:

```
news.channel_id
```

而不是:

```
news.channel.id
```

批量插入

- 用 django.db.models.query.QuerySet.bulk_create() 批量创建对象,减少SQL查询的 数量。比如

```
Entry.objects.bulk_create([
    Entry(headline="Python 3.0 Released"),
    Entry(headline="Python 3.1 Planned")
])
```

...而不是

```
Entry.objects.create(headline="Python 3.0 Released")
Entry.objects.create(headline="Python 3.1 Planned")
```

这同样适用于 ManyToManyFields,因此，这么做

```
team.members.add(me, my_friend)
```

...而不是这么做

```
team.members.add(me)
team.members.add(my_friend)
```

...这里 team 和 members 是多对多的关系。

参考资源：

https://docs.djangoproject.com/en/1.4/topics/db/optimization/
(https://docs.djangoproject.com/en/1.4/topics/db/optimization/)