Django 数据库访问优化

by: flora5 2017-01-23 Monday 晴天 ☀

性能分析

优化之前先要找出-优化目标，哪些查询耗费资源。 用类似 django-debug-toolbar 的工具，或直接监控数据库的工具。根据需要，优化速度，或内存，或两者。

connection.queries 包含了所有 SQL 语句,每次程序访问数据库，查询语句都会被记录下来，包括执行的时间。

优化项之间可能互相抵消或互相促进。在数据库中与在 python 中完成相同的工作，开销不同，这取决于你的优先级，平衡点。根据需要进行配置，因为这依赖于你的程序和服务器。

每次修改后记得性能分析 ，考虑到代码可读性的降低，保证修改能带来足够大的好处。

标准数据库优化技术

1， 索引

用 Field.db_index 或 Meta.index_together 在 django 中添加索引

注意：确定最佳索引是一个依赖于特定应用程序的复杂数据库相关主题，维护索引的开销可能超过查询速度带来的好处。

2， 合理的字段类型

在以上优化完成的基础上：

目录：

1. 理解 QuerySets
2. 不要在 python 中执行应该在数据库中执行的工作
3. 用唯一的，索引列来检索单个对象
4. 将需要的数据一次请求过来
5. 不要取不需要的数据
6. 不要做不必要的排序

7. 批量插入

文档没有涉及适用于所有大型开销的其他优化技术，如：通用缓存

执行上述优化：

1，理解 QuerySets

理解 QuerySet 求值

避免性能问题，了解以下几点：（每个条目有单独的详细文档）

- QuerySets 是延迟求值的
- 什么时候求值
- 数据在内存中是如何存储的

理解 cached attributes

跟缓存整个 QuerySet 一样，除此之外，还有对 ORM 对象属性结果的缓存。 一般来说，不可调用的属性将被缓存。 例如，假设例子（官网文档中的例子）Weblog models：

```
>>> entry = Entry.objects.get(id=1)
>>> entry.blog   # Blog object is retrieved at this point
>>> entry.blog   # cached version, no DB access
```

Callable 的属性每次都会导致数据库查询：

```
>>> entry = Entry.objects.get(id=1)
>>> entry.authors.all()   # query performed
>>> entry.authors.all()   # query performed again
```

在阅读模板代码时要注意 - 模板系统不允许使用括号：entry.authors.all( )

自定义属性是手动实现缓存的，例如使用 cached_property 装饰器

使用 with 模板标签

使用 QuerySet 的缓存行为，要用 with 模板标签

使用 iterator()

如果有很多对象，QuerySet 的缓存行为会占用大量内存。 这时，可以用 iterator()

2，不要在 python 中执行应该在数据库中执行的工作

例如：

一般，用 filter 和 exclude 在数据库中进行过滤。

使用 F 表达式对基于同一模型的其他字段进行过滤。 （详细见 F 表达式文档）

使用 annotate 在数据库中进行聚合

如果这些不足以生成需要的 SQL：

兼容性低，但是更强大的方法是 RawSQL 表达式，它允许一些 SQL 被显式的添加到查询中。如果这还不够强大：

用原生 SQL

原生 SQL 查询数据或填充模型（populate models）。用 django.db.connection.queries 找出 Django 中已经有的方法，从那里开始。

3，用 unique，indexed 列检查询个对象

当使用 get（）查询单个对象时，有两个原因要使用具有 unique 或 db_index 的列：

首先，因为索引，查询将更快。其次， 若有多个对象匹配查找条件，查询会更慢；列上有唯一约束 (unique constraint)能保证这不会发生。

所以，使用例子中的:Weblog models

```
>>> entry = Entry.objects.get(id=10)
>>> entry = Entry.objects.get(headline="News Item Title") #更慢
```

因为 id 是被数据库索引的，保证唯一的.

以下操作可能相当慢：

```
>>> entry = Entry.objects.get(headline__startswith="News")
```

首先，headline 没有被索引，数据库查询更慢；

其次，查询并不保证只返回一个对象。 若匹配多个，会获取和传输所有这些对象。 返回数百或数千条记录，速度更慢。 数据库是独立服务器，还会有网络开销和延迟。

4，将需要的数据一次请求过来

多次查询数据库获取单个集合数据的不同部分，显然不如一次查询全部获取；

在循环语句中执行的查询，可能只需要执行一次数据库查询就够了，却执行了很多次。

所以：

使用 QuerySet.select_related() 和 prefetch_related()

全面理解 select_related() 和 **prefetch_related()** ， 在以下场景中使用它们：

- view 代码中
- managers anddefault managers 中的适当情况下。要注意什么时候使用和不使用 manager

有时候这有点复杂，不要做假设。

Manager 是 向 Django models 提供数据库查询操作的接口。 Django 应用程序中的每个 model 至少有一个 Manager。

5，不要取不需要的数据

使用 QuerySet.values() and values_list()

若只需要值列表(list of values)dict，或者值字典列表(dict of values)，不需要 ORM model 对象，可以适当使用 values（）。 这对替换 template 中的 model 对象很有用 - 只要提供的字典(dicts)与模板中使用的具有相同的属性既可；

使用 QuerySet.defer() ， only()

如果知道有数据库列不需要（或在大多数情况下不需要），则使用 defer（）和 only（），以避免加载它们。注意，如果确实使用它们，只能在 ORM 中通过单独的查询中获取，如果使用不当，就成了’最差化’（pessimization – the opposite of an optimist.）

此外，在构建具有延迟字段（deferred fields）的 model 时，Django 内部会产生一些（小的额外）开销。 不要在没有分析的情况使用武断使用延迟字段，因为为了结果集中的单独一行，即使只使用几个列 ，数据库都必须从磁盘读取大部分的 non-text, non-varchar 数据 。 defer（）和 only（）在避免加载大量文本数据，或可能需要大量处理以转换回 Python 的字段时最为有用。 仍然是：先分析，后优化。

如果只想要计数，而不是执行 len（queryset），用 QuerySet.count()

如果只想找出是否至少有一个结果存在，而不是 if queryset 用 QuerySet.exists()

但：

不要过度使用 count() and exists()

如果您将需要 QuerySet 中的其他数据，只需计算

例如，假设 Email model 具有 body 属性和与 User 多对多的关系，以下模板代码是最佳的：

```
{% 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 %}
```

1，由于 QuerySets 是延迟（lazy）的，如果’display_inbox’为 False，则不会执行数据库查询

2, 使用 with 意味着将 user.emails.all 存储在变量中供以后使用

3，{%if emails%}行 会调用 QuerySet .__ bool __（），这导致 user.emails.all（）查询数据库，并且至少第一行被转换为 ORM 对象 。 如果没有结果，返回 False，否则返回 True

4，用{{emails | length}}调用 QuerySet .__ len __（），填充其余的缓存，而不进行新的查询

5， for 循环遍历已经填充的缓存

总的来说，此代码执行一个或零个数据库查询。 唯一刻意的优化是使用 with 标记。 在任何时候使用 QuerySet.exists（）或 QuerySet.count（）将导致额外的查询。

直接的使用 foreign key values

如果只需要一个 foreign key 的值，使用已经在对象上的 foreign key 值，而不是获取整个对象，并获取其主键。 即：

```
entry.blog_id # 这样做

entry.blog.id #不要这样做
```

6，不要做不必要的排序

排序是有开销的，order by 每个字段，数据库都必须执行相应的操作。 如果 Model 有默认排序（Meta.ordering），若不需要，调用没有参数的 order_by（） 能在 QuerySet 中将它删除。

7，批量插入

创建对象时，尽量用 bulk_create() 减少 SQL 查询。 例如：

```
Entry.objects.bulk_create([
    Entry(headline='This is a test'),
    Entry(headline='This is only a test'),
])
```

优于

```
Entry.objects.create(headline='This is a test')
Entry.objects.create(headline='This is only a test')
```

这也适用于 ManyToManyFields：

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

优于：

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

Bands 和 Artists 是 many-to-many 关系

- The end

Reference： https://docs.djangoproject.com/en/1.10/topics/db/optimization/