译 | hon Django性能测试与优化指南

2017年 2日 15:43:35

> 原 Guide to Performance Testing and Optimization With Python and Django
> 作 LIAN GULEA
> 翻 惊寒

摘要：通过一个简单的实例一步一步引导读者对其进行全方位的性能优化。以下是译文。

唐纳德·克努特（Donald Knuth）曾经说过：“不成熟的优化方案是万恶之源。”然而，任何一个承受高负载的成熟项目都不可避免地需要进行优化。在本文中，我们将谈谈**优化Web项目代码的五种常用方法**。虽然本文是以Django为例，但其他框架和语言的优化原则也是类似的。通过使用这些优化方法，文中例程的执行时间从原来的77秒减少到了3.7秒。

本文用到的例程是从一个我曾经使用过的真实项目改编而来的，是性能优化技巧的典范。如果你想自己尝试着进行优化，可以在GitHub上获取优化前
并跟着下文做相应的修改。我使用的是Python 2，因为一些第三方软件包还不支持Python 3。

示例代码介绍

这个Web项目只是简单地跟踪每个地区的房产价格。因此，只有两种模型：

```
# houses/models.py
from utils.hash import Hasher


class HashableModel(models.Model):
    """Provide a hash property for models."""
    class Meta:
```

```
        abstract = True

    @property
    def hash(self):
        return Hasher.from_model(self)


ss Country(HashableModel):
    """Represent a country in which the house is positioned."""
    name = models.CharField(max_length=30)

    def __unicode__(self):
        return self.name


ss House(HashableModel):
    """Represent a house with its characteristics."""
    # Relations
    country = models.ForeignKey(Country, related_name='houses')

    # Attributes
    address = models.CharField(max_length=255)
    sq_meters = models.PositiveIntegerField()
    kitchen_sq_meters = models.PositiveSmallIntegerField()
    nr_bedrooms = models.PositiveSmallIntegerField()
    nr_bathrooms = models.PositiveSmallIntegerField()
    nr_floors = models.PositiveSmallIntegerField(default=1)
    year_built = models.PositiveIntegerField(null=True, blank=True)
    house_color_outside = models.CharField(max_length=20)
    distance_to_nearest_kindergarten = models.PositiveIntegerField(null=True, blank=True)
    distance_to_nearest_school = models.PositiveIntegerField(null=True, blank=True)
    distance_to_nearest_hospital = models.PositiveIntegerField(null=True, blank=True)
    has_cellar = models.BooleanField(default=False)
    has_pool = models.BooleanField(default=False)
    has_garage = models.BooleanField(default=False)
    price = models.PositiveIntegerField()

    def __unicode__(self):
        return '{} {}'.format(self.country, self.address)
```

抽象类`HashableModel`提供了一个继承自模型并包含`hash`属性的模型，这个属性包含了实例的主键和模型的内容类型。 这能够隐藏像实例ID这样的敏感
列进行代替。如果项目中有多个模型，而且需要在一个集中的地方对模型进行解码并要对不同类的不同模型实例进行处理时，这可能会非常有用。 请
文的这个小项目，即使不用散列也照样可以处理，但使用散列有助于展示一些优化技巧。

这是`Hasher`类：

```
# utils/hash.py
import basehash


class Hasher(object):
    @classmethod
    def from_model(cls, obj, klass=None):
        if obj.pk is None:
            return None
        return cls.make_hash(obj.pk, klass if klass is not None else obj)

    @classmethod
    def make_hash(cls, object_pk, klass):
        base36 = basehash.base36()
        content_type = ContentType.objects.get_for_model(klass, for_concrete_model=False)
        return base36.hash('%(contenttype_pk)03d%(object_pk)06d' % {
            'contenttype_pk': content_type.pk,
            'object_pk': object_pk
        })

    @classmethod
    def parse_hash(cls, obj_hash):
        base36 = basehash.base36()
        unhashed = '%09d' % base36.unhash(obj_hash)
```

```
        contenttype_pk = int(unhashed[:-6])
        object_pk = int(unhashed[-6:])
        return contenttype_pk, object_pk

    @classmethod
    def to_object_pk(cls, obj_hash):
        return cls.parse_hash(obj_hash)[1]
```

由于我们通过API来提供这些数据，所以我们安装了Django REST框架并定义以下序列化器和视图：

```
# houses/serializers.py
class HouseSerializer(serializers.ModelSerializer):
    """Serialize a `houses.House` instance."""

    id = serializers.ReadOnlyField(source="hash")
    country = serializers.ReadOnlyField(source="country.hash")

    class Meta:
        model = House
        fields = (
            'id',
            'address',
            'country',
            'sq_meters',
            'price'
        )
```

```
# houses/views.py
class HouseListAPIView(ListAPIView):
    model = House
    serializer_class = HouseSerializer
    country = None

    def get_queryset(self):
        country = get_object_or_404(Country, pk=self.country)
        queryset = self.model.objects.filter(country=country)
        return queryset

    def list(self, request, *args, **kwargs):
        # Skipping validation code for brevity
        country = self.request.GET.get("country")
        self.country = Hasher.to_object_pk(country)
        queryset = self.get_queryset()

        serializer = self.serializer_class(queryset, many=True)

        return Response(serializer.data)
```

现在，我们将用一些数据来填充数据库（使用`factory-boy`生成10万个房屋的实例：一个地区5万个，另一个4万个，第三个1万个），并准备测试应用程

性能优化其实就是测量

在一个项目中我们需要测量下面这几个方面：

- 执行时间
- 代码的行数
- 函数调用次数
- 分配的内存
- 其他

但是，并不是所有这些都要用来度量项目的执行情况。一般来说，有两个指标比较重要：执行多长时间、需要多少内存。

在Web项目中，*响应时间*（服务器接收由某个用户的操作产生的请求，处理该请求并返回结果所需的总的时间）通常是最重要的指标，因为过长的响应户厌倦等待，并切换到浏览器中的另一个选项卡页面。

在编程中，分析项目的性能被称为profiling。为了分析API的性能，我们将使用Silk包。在安装完这个包，并调用`/api/v1/houses/?country=5T22RI`后，的结果：

```
200 GET
/api/v1/houses/

92ms overall
54ms on queries
4
04 queries
```

整体响应时间为77秒，其中16秒用于查询数据库，总共有5万次查询。这几个数字很大，提升空间也有很大，所以，我们开始吧。

1. 优化数据库查询

性能优化常见的技巧之一是对数据库查询进行优化，本案例也不例外。同时，还可以对查询做多次优化来减小响应时间。

1.1 只提供所有数据

仔细观察这5万次查询查的是什么：都是对`houses_country`表的查询：

```
200 GET
/api/v1/houses/

77292ms overall
15854ms on queries
50004 queries
```

时间戳	表名	联合	执行时间（毫秒）
+0:01 :15.874374	“houses_country”	0	0.176
+0:01 :15.873304	“houses_country”	0	0.218
+0:01 :15.872225	“houses_country”	0	0.218
+0:01 :15.871155	“houses_country”	0	0.198
+0:01 :15.870099	“houses_country”	0	0.173
+0:01 :15.869050	“houses_country”	0	0.197
+0:01 :15.867877	“houses_country”	0	0.221
+0:01 :15.866807	“houses_country”	0	0.203
+0:01 :15.865646	“houses_country”	0	0.211
+0:01 :15.864562	“houses_country”	0	0.209
+0:01 :15.863511	“houses_country”	0	0.181
+0:01 :15.862435	“houses_country”	0	0.228
+0:01 :15.861413	“houses_country”	0	0.174

这个问题的根源是，Django中的查询是*惰性的*。这意味着在你真正需要获取数据之前它不会访问数据库。同时，它只获取你指定的数据，如果需要其他则要另外发出请求。

这正是本例程所遇到的情况。当通过`House.objects.filter(country=country)`来获得查询集时，Django将获取特定地区的所有房屋。但是，在序列化一时，`HouseSerializer`需要房子的`country`实例来计算序列化器的`country`字段。由于地区数据不在查询集中，所以django需要提出额外的请求来获取这查询集中的每一个房子都是如此，因此，总共是五万次。

当然，解决方案非常简单。为了提取所有需要的序列化数据，你可以在查询集上使用`select_related()`。因此，`get_queryset`函数将如下所示：

```
def get_queryset(self):
    country = get_object_or_404(Country, pk=self.country)
    queryset = self.model.objects.filter(country=country).select_related('country')
    return queryset
```

我们来看看这对性能有何影响：

```
200 GET
/api/v1/houses/
```

```
35979ms overall
102ms on queries
    ueries
```

总体时间降至36秒，在数据库中花费的时间约为100ms，只有4个查询！这是个好消息，但我们可以做得更多。

1.2 仅提供相关的数据

默认情况下，Django会从数据库中提取所有字段。但是，当表有很多列很多行的时候，告诉Django提取哪些特定的字段就非常有意义了，这样就不会根本用不到的信息。在本案例中，我们只需要5个字段来进行序列化，虽然表中有17个字段。明确指定从数据库中提取哪些字段是很有意义的，可以进时间。

Django使用`defer()`和`only()`这两个查询方法来实现这一点。第一个用于指定哪些字段*不要加载*，第二个用于指定*只加载*哪些字段。

```
get_queryset(self):
country = get_object_or_404(Country, pk=self.country)
queryset = self.model.objects.filter(country=country)\
    .select_related('country')\
    .only('id', 'address', 'country', 'sq_meters', 'price')
return queryset
```

这减少了一半的查询时间，非常不错。总体时间也略有下降，但还有更多提升空间。

```
200 GET
/api/v1/houses/

33111ms overall
52ms on queries
4 queries
```

2. 代码优化

你不能无限制地优化数据库查询，并且上面的结果也证明了这一点。即使把查询时间减少到0，我们仍然会面对需要等待半分钟才能得到应答这个现实转移到另一个优化级别上来了，那就是：*业务逻辑*。

2.1 简化代码

有时，第三方软件包对于简单的任务来说有着太大的开销。本文例程中返回的序列化的房子实例正说明了这一点。

Django REST框架非常棒，包含了很多有用的功能。但是，现在的主要目标是缩短响应时间，所以该框架是优化的候选对象，尤其是我们要使用的序功能非常的简单。

为此，我们来编写一个自定义的序列化器。为了方便起见，我们将用一个静态方法来完成这项工作。

```
# houses/serializers.py
class HousePlainSerializer(object):
    """
    Serializes a House queryset consisting of dicts with
    the following keys: 'id', 'address', 'country',
    'sq_meters', 'price'.
    """

    @staticmethod
    def serialize_data(queryset):
        """
        Return a list of hashed objects from the given queryset.
        """
        return [
            {
                'id': Hasher.from_pk_and_class(entry['id'], House),
                'address': entry['address'],
                'country': Hasher.from_pk_and_class(entry['country'], Country),
                'sq_meters': entry['sq_meters'],
                'price': entry['price']
            } for entry in queryset
        ]


# houses/views.py
```

```
class HouseListAPIView(ListAPIView):
    model = House
    serializer_class = HouseSerializer
    plain_serializer_class = HousePlainSerializer  # <-- added custom serializer
    country = None

    def get_queryset(self):
        country = get_object_or_404(Country, pk=self.country)
        queryset = self.model.objects.filter(country=country)
        return queryset

    def list(self, request, *args, **kwargs):
        # Skipping validation code for brevity
        country = self.request.GET.get("country")
        self.country = Hasher.to_object_pk(country)
        queryset = self.get_queryset()

        data = self.plain_serializer_class.serialize_data(queryset)  # <-- serialize

        return Response(data)
```

```
200 GET
/api/v1/houses/

17312ms overall
38ms on queries
4 queries
```

现在看起来好多了，由于没有使用DRF序列化代码，所以响应时间几乎减少了一半。

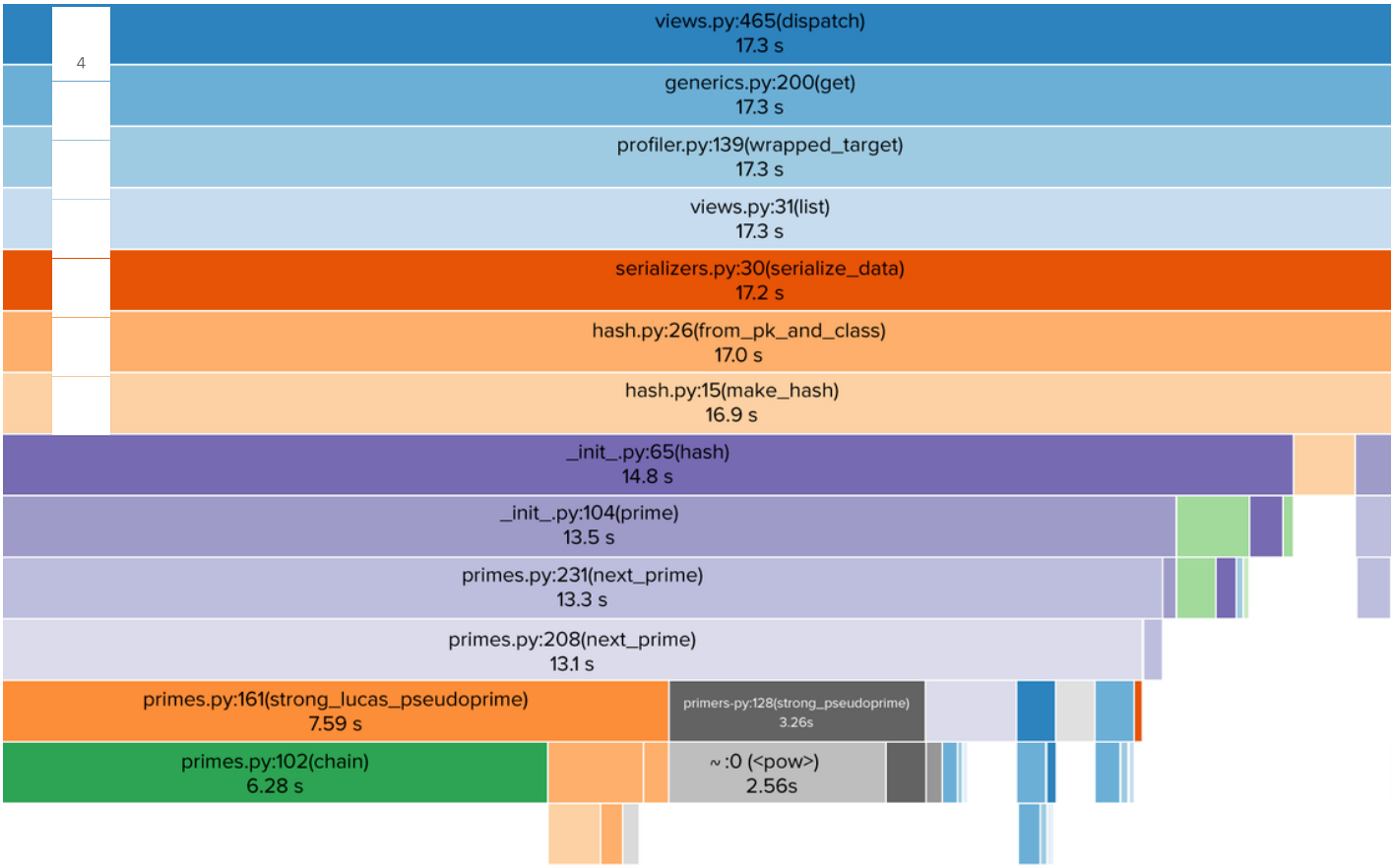
另外还有一个结果：在请求/响应周期内完成的总的函数调用次数从15,859,427次（上面1.2节的请求次数）减少到了9,257,469次。这意味着大约有三
调用都是由Django REST Framework产生的。

2.2 更新或替代第三方软件包

上述几个优化技巧是最常见的，无需深入地分析和思考就可以做到。然而，17秒的响应时间仍然感觉很长。要减少这个时间，需要更深入地了解代码
生了什么。换句话说，需要分析一下代码。

你可以自己使用Python内置的分析器来进行分析，也可以使用一些第三方软件包。由于我们已经使用了 `silk`，它可以分析代码并生成一个二进制的
此，我们可以做进一步的可视化分析。有好几个可视化软件包可以将二进制文件转换为一些友好的可视化视图。本文将使用 `snakeviz` 。

这是上文一个请求的二进制分析文件的可视化图表：



从上到下是调用堆栈，显示了文件名、函数名及其行号，以及该方法花费的时间。可以很容易地看出，时间大部分都用在计算散列上（紫罗兰色的`__i imes.py`矩形）。

目前，这是代码的主要性能瓶颈，但同时，这不是我们自己写的代码，而是用的第三方包。

在这种情况下，我们可以做的事情将非常有限：

- 检查包的最新版本（希望能有更好的性能）。
- 寻找另一个能够满足我们需求的软件包。
- 我们自己写代码，并且性能优于目前使用的软件包。

幸运的是，我们找到了一个更新版本的`basehash`包。原代码使用的是v.2.1.0，而新的是v.3.0.4。

当查看v.3的发行说明时，这一句话看起来令人充满希望：

> *“使用素数算法进行大规模的优化。”*

让我们来看一下！

```
pip install -U basehash gmpy2
```

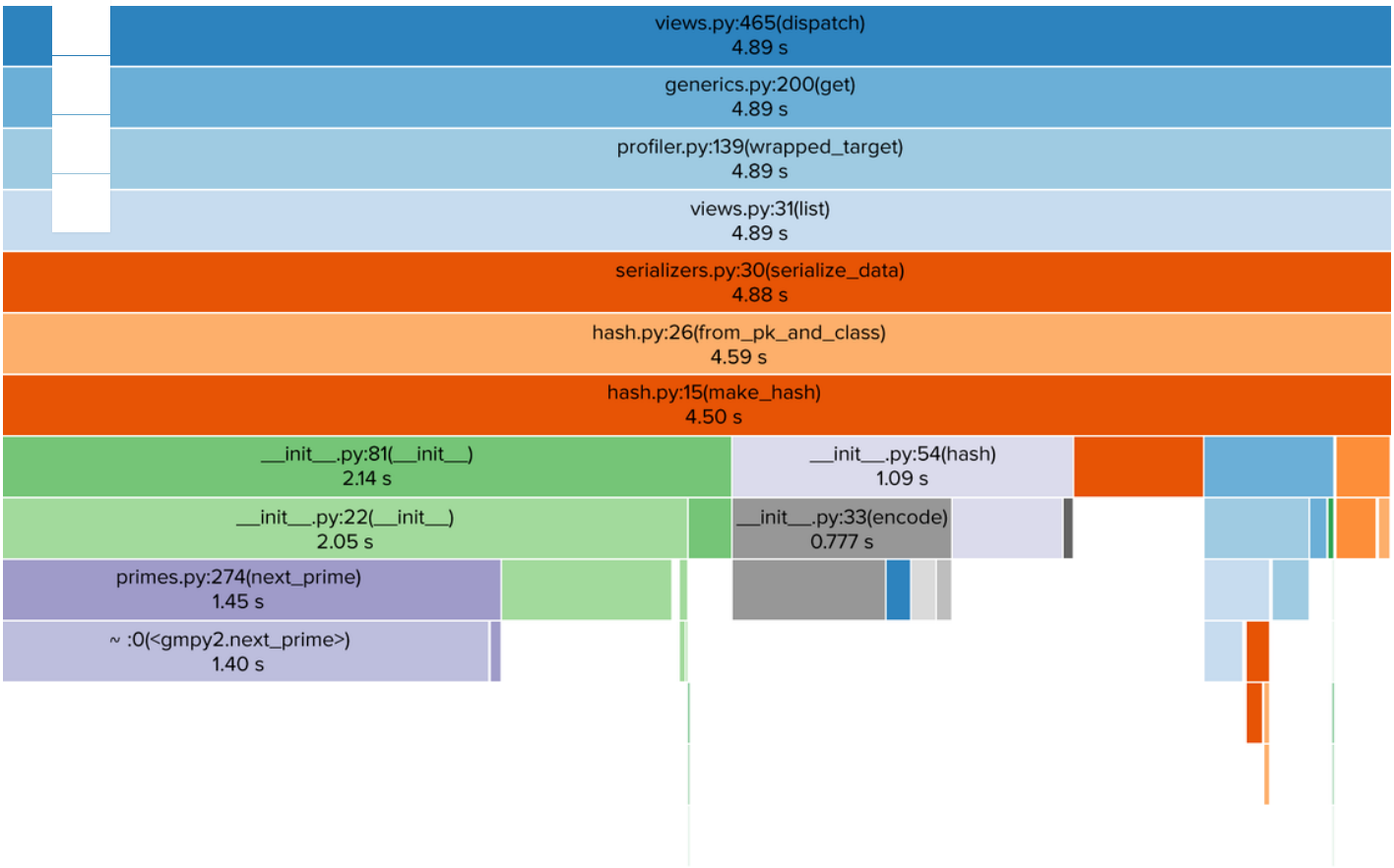
```
200 GET
/api/v1/houses/

7738ms overall
59ms on queries
4 queries
```

响应时间从17秒缩短到了8秒以内。太棒了！但还有一件事我们应该来看看。

2.3 [illegible]代码

到目[illegible]，我们已经改进了查询、用自己特定的函数取代了第三方复杂而又泛型的代码、更新了第三方包，但是我们还是保留了原有的代码。但有时进行[illegible]的重构可能会带来意想不到的结果。但是，为此我们需要再次分析运行结果。



仔细看一下，你可以看到散列仍然是一个问题（毫不奇怪，这是我们对数据做的唯一的事情），虽然我们确实朝这个方向改进了，但这个绿色的矩形表
py花了2.14秒的时间，同时伴随着灰色的`__init__.py:54(hash)`。这意味着初始化工作需要很长的时间。

我们来看看`basehash`包的源代码。

```
# basehash/__init__.py

# Initialization of `base36` class initializes the parent, `base` class.
class base36(base):
    def __init__(self, length=HASH_LENGTH, generator=GENERATOR):
        super(base36, self).__init__(BASE36, length, generator)


class base(object):
    def __init__(self, alphabet, length=HASH_LENGTH, generator=GENERATOR):
        if len(set(alphabet)) != len(alphabet):
            raise ValueError('Supplied alphabet cannot contain duplicates.')

        self.alphabet = tuple(alphabet)
        self.base = len(alphabet)
        self.length = length
        self.generator = generator
        self.maximum = self.base ** self.length - 1
        self.prime = next_prime(int((self.maximum + 1) * self.generator))  # `next_prime` call on each initialized instance
```

正如你所看到的，一个base实例的初始化需要调用next_prime函数，这是太重了，我们可以在上面的可视化图表中看到左下角的矩形。

我们再一看Hash类：

```
class Hasher(object):
    @classmethod
    def from_model(cls, obj, klass=None):
        if obj.pk is None:
            return None
        return cls.make_hash(obj.pk, klass if klass is not None else obj)

    @classmethod
    def make_hash(cls, object_pk, klass):
        base36 = basehash.base36()  # <-- initializing on each method call
        content_type = ContentType.objects.get_for_model(klass, for_concrete_model=False)
        return base36.hash('%(contenttype_pk)03d%(object_pk)06d' % {
            'contenttype_pk': content_type.pk,
            'object_pk': object_pk
        })

    @classmethod
    def parse_hash(cls, obj_hash):
        base36 = basehash.base36()  # <-- initializing on each method call
        unhashed = '%09d' % base36.unhash(obj_hash)
        contenttype_pk = int(unhashed[:-6])
        object_pk = int(unhashed[-6:])
        return contenttype_pk, object_pk

    @classmethod
    def to_object_pk(cls, obj_hash):
        return cls.parse_hash(obj_hash)[1]
```

正如你所看到的，我已经标记了这两个方法初始化base36实例的方法，这并不是真正需要的。

由于散列是一个确定性的过程，这意味着对于一个给定的输入值，它必须始终生成相同的散列值，因此，我们可以把它作为类的一个属性。让我们来看行：

```
class Hasher(object):
    base36 = basehash.base36()  # <-- initialize hasher only once

    @classmethod
    def from_model(cls, obj, klass=None):
        if obj.pk is None:
            return None
        return cls.make_hash(obj.pk, klass if klass is not None else obj)

    @classmethod
    def make_hash(cls, object_pk, klass):
        content_type = ContentType.objects.get_for_model(klass, for_concrete_model=False)
        return cls.base36.hash('%(contenttype_pk)03d%(object_pk)06d' % {
            'contenttype_pk': content_type.pk,
            'object_pk': object_pk
        })

    @classmethod
    def parse_hash(cls, obj_hash):
        unhashed = '%09d' % cls.base36.unhash(obj_hash)
        contenttype_pk = int(unhashed[:-6])
        object_pk = int(unhashed[-6:])
        return contenttype_pk, object_pk

    @classmethod
    def to_object_pk(cls, obj_hash):
        return cls.parse_hash(obj_hash)[1]
```

```
**200 GET**

  /api/v1/houses/

3766ms overall
```

```
6  38ms on queries
7  4 queries
```

最后的 4 是在4秒钟之内，比我们一开始的时间要小得多。对响应时间的进一步优化可以通过使用缓存来实现，但是我不会在这篇文章中介绍这个。

结论

性能优化是一个分析和发现的过程。 没有哪个硬性规定能适用于所有情况，因为每个项目都有自己的流程和瓶颈。 然而，你应该做的第一件事是分析你的查询。 这样一个短的例子中，我可以将响应时间从77秒缩短到3.7秒，那么对于一个庞大的项目来说，就会有更大的优化潜力。

文章标签： python django 性能测试 ▼查看关于本篇文章更多信息

想对作者说点什么