一、过滤Filtering

DRF的过滤器类都定义在filters模块中，这个模块也很简单，300多行代码而已，主要定义了下面四个类：

- BaseFilterBackend：基类，被继承。
- SearchFilter：继承BaseFilterBackend类
- OrderingFilter：继承BaseFilterBackend类
- DjangoObjectPermissionsFilter：继承BaseFilterBackend类。3.9版本后废除。

DRF通用列表视图的默认行为是返回一个模型的全部queryset，比如说你有一个用户的模型，当前存了1万条用户信息，那么默认会将这1万条记录全部取出。而通常情况下，我们是不想也不需要一次性全部取出来的，只需要其中的一部分就行。这就需要我们对查询的结果进行过滤。

最简单的过滤方法就是在任意继承了 `GenericAPIView` 的视图中重写 `.get_queryset()` 方法，使用这个方法来过滤查询结果。

根据用户进行过滤

你可能想要过滤queryset，只返回与发出请求的当前已验证的用户相关的结果。这可以使用 `request.user` 的值进行过滤来实现。

例如：

```
from myapp.models import Purchase
from myapp.serializers import PurchaseSerializer
from rest_framework import generics

class PurchaseList(generics.ListAPIView):
    serializer_class = PurchaseSerializer

    def get_queryset(self):  #1. 重写该方法
        """
        这个视图将返回一个列表，只显示当前用户的购物清单，而不是所有用户的购物清单
        """
        user = self.request.user # 2.获取用户
        return Purchase.objects.filter(purchaser=user)  # 3.返回用户的购物清单
```

根据URL进行过滤

还有一种过滤方式，是根据请求的URL的部分内容来过滤查询集。

例如，对于下面形式的url:

```
path('purchases/<str:username>/', PurchaseList.as_view()),
```

你就可以写一个视图，返回基于URL中的username参数进行过滤的结果。

```
class PurchaseList(generics.ListAPIView):
    serializer_class = PurchaseSerializer

    def get_queryset(self):
        """
        This view should return a list of all the purchases for
        the user as determined by the username portion of the URL.
        """
        username = self.kwargs['username'] # 与上面的例子，不同之处只是获取username的方式变了
        return Purchase.objects.filter(purchaser__username=username)
```

根据查询参数进行过滤

实际上我们还可以通过URL中携带的参数来过滤查询集。也就是上一节更广义的形式。

比如我们可以通过重写 `.get_queryset()` 方法来处理像 `http://example.com/api/purchases?username=denvercoder9` 这样的URL，并且只有在URL中包含 `username` 参数时，才过滤查询集：

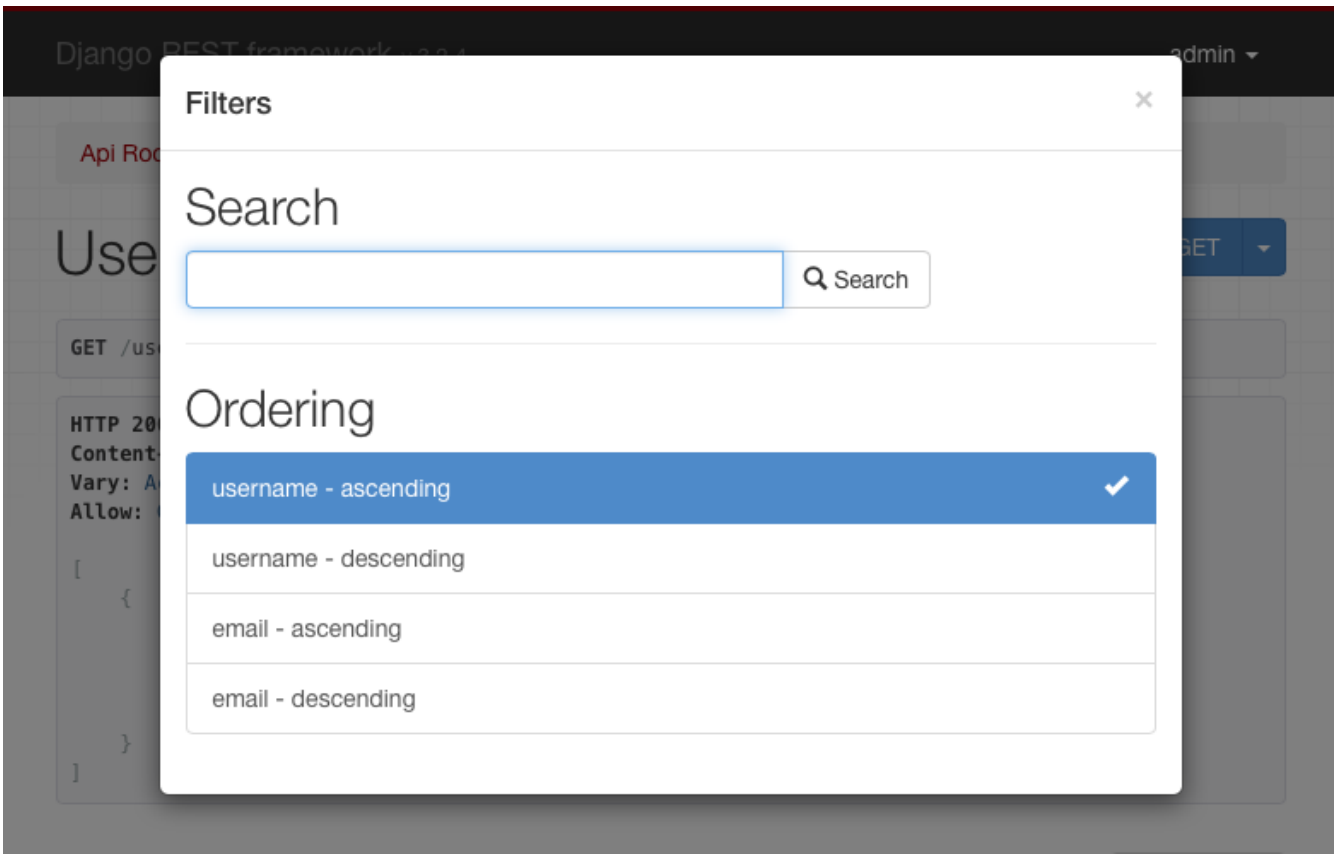
```python
class PurchaseList(generics.ListAPIView):
    serializer_class = PurchaseSerializer

    def get_queryset(self):
        """
        Optionally restricts the returned purchases to a given user,
        by filtering against a `username` query parameter in the URL.
        """
        queryset = Purchase.objects.all() # 这是保底的查询集
        username = self.request.query_params.get('username', None) # 注意获取参数的方法
        if username is not None:
            queryset = queryset.filter(purchaser__username=username) # 这是过滤后的查询集
        return queryset # 总是能返回一个查询集
```

二、通用过滤

除了能够重写默认的查询集，DRF框架还支持通用的过滤后端，让你可以轻松地构建复杂的检索器和过滤器。

通用过滤器也可以在可浏览的API和Admin后台API中显示为HTML搜索或过滤控件。



默认的过滤器后端可以在全局设置中使用 `DEFAULT_FILTER_BACKENDS` 来配置。例如。

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
('django_filters.rest_framework.DjangoFilterBackend',)
}
```

注意了，这个过滤器后端不是DRF自带的，而是django-filter模块提供的。所以需要pip安装。需要注意参数的引用方式，不要犯经验主义错误。

还可以使用基于 `GenericAPIView` 的类视图在每个view或每个viewset基础上设置过滤器后端。

```
import django_filters.rest_framework
from django.contrib.auth.models import User
from myapp.serializers import UserSerializer
from rest_framework import generics

class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    filter_backends = (django_filters.rest_framework.DjangoFilterBackend,) #
看这里
```

注意下面的例子，查找一个ID为 `4675` 的产品。以下URL将返回相应的对象或者返回404，具体取决于给定的产品实例是否满足筛选条件。也就是说，你必须同时满足id存在，过滤条件也符合，才能成功或者这个唯一对象。

```
http://example.com/api/products/4675/?category=clothing&max_price=10.00
```

我们还可以同时重写 `.get_queryset()` 方法并使用通用过滤器，并且一切都会按照预期生效。 例如，如果 `Product` 模型与 `User` 模型具有多对多关系，也就是购物清单 `purchase` ，则可能需要编写如下所示的视图：

```python
class PurchasedProductsList(generics.ListAPIView):
    """
    Return a list of all the products that the authenticated
    user has ever purchased, with optional filtering.
    """
    model = Product
    serializer_class = ProductSerializer
    filterset_class = ProductFilter

    def get_queryset(self):
        user = self.request.user
        return user.purchase_set.all()
```

三、API 参考

DjangoFilterBackend

`django-filter` 模块包含一个 `DjangoFilterBackend` 类，为DRF提供高度可定制的字段过滤功能。

要使用 `DjangoFilterBackend` ，首先需要安装 `django-filter` 模块。，然后将 `django_filters` 添加到Django的 `INSTALLED_APPS` 列表中。

```
pip install django-filter

//如果同时安装下面的库，可以让过滤的输入表单更美观
pip install django-crispy-forms
```

安装完成后，可以进行下面的配置，以进行全局性的过滤操作：

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS':
('django_filters.rest_framework.DjangoFilterBackend',)
}
```

或者将它们添加到单个视图中，进行视图级别的过滤操作：

```python
from django_filters.rest_framework import DjangoFilterBackend # 导入!

class UserListView(generics.ListAPIView):
    ...
    filter_backends = (DjangoFilterBackend,) # 看这里
```

如果你只需要一个简单的过滤功能，那么也只需要简单地在视图或视图集上添加一个 `filterset_fields` 属性，属性的值是一个元组，列出要过滤的字段，如下所示：

```python
class ProductList(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (DjangoFilterBackend,) # 先指定后端
    filterset_fields = ('category', 'in_stock') # 看这里
```

这将为指定的字段自动创建一个 `FilterSet` 类，然后你就可以发送类似下面的url请求了：

```
http://example.com/api/products?category=clothing&in_stock=True
```

Django-filter模块的默认模式是完全匹配模式，如果需要自定义匹配模式请查阅 django-filter documentation。

SearchFilter

`SearchFilter` 类是DRF自带的过滤器，支持基于简单的单个查询参数的搜索，并且基于Django admin的搜索功能。

在使用时，可浏览的API页面中将包括一个 `SearchFilter` 控件：

仅当视图中设置了 `search_fields` 属性时，才会应用 `SearchFilter` 类。 `search_fields` 只支持文本类型字段，例如 `CharField` 或 `TextField`。

```python
from rest_framework import filters

class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    filter_backends = (filters.SearchFilter,)  # DRF自带的过滤器
    search_fields = ('username', 'email')  # 看这里
```

做了上面的工作，你才可以访问下面形式的url（注意参数）：

```
http://example.com/api/users?search=russell
```

你还可以使用双下划线对ForeignKey或ManyToManyField执行关系查询：

```python
search_fields = ('username', 'email', 'profile__profession')  # 注意最后一个元素
```

默认情况下，搜索不区分大小写，并使用部分匹配的模式。实际上，可以同时有多个搜索参数，用空格和/或逗号分隔。 如果使用多个搜索参数，则仅当所有提供的模式都匹配时才在列表中返回对象。

可以通过在`search_fields`前面添加各种字符来限制搜索行为：

- `^` 以指定内容开始
- `=` 完全匹配
- `@` 全文搜索（目前只支持Django的MySQL后端）
- `$` 正则搜索

例如：

```python
search_fields = ('=username', '=email')  # 指定用户名和邮箱必须完全一致，不能局部一致
```

默认情况下，url中搜索参数的名字为`'search'`，这可以通过`SEARCH_PARAM`配置项进行自定义，比如改为`find`：

```
http://example.com/api/users?find=russell
```

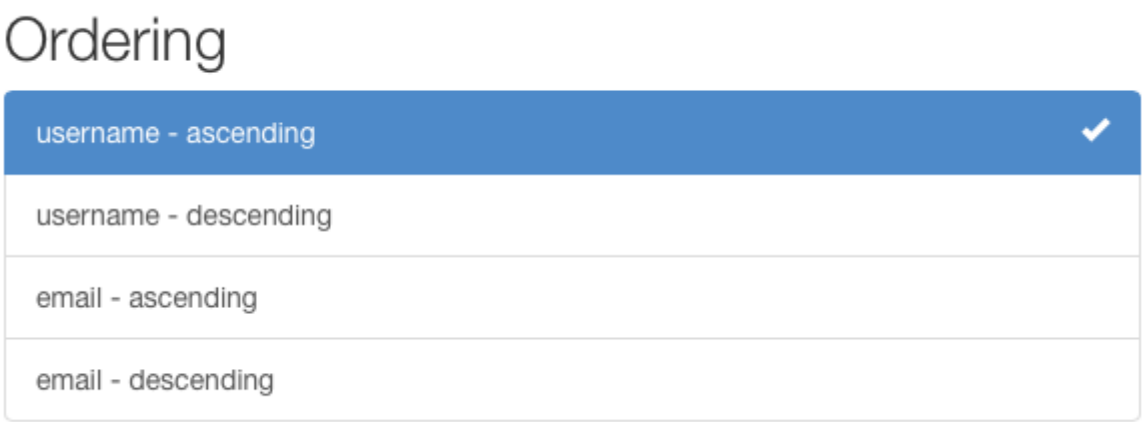
为了根据请求的内容动态地修改查找的字段，可以继承`SearchFilter`类，并重写`get_search_fields()`方法。例如，下面的搜索类只搜索title字段，如果在请求中包含了`title_only`参数。

```
from rest_framework import filters

class CustomSearchFilter(filters.SearchFilter):
    def get_search_fields(self, view, request):
        if request.query_params.get('title_only'):
            return ('title',) # 如何条件则只搜索title字段
        return super(CustomSearchFilter, self).get_search_fields(view,
request) # 保底的
```

OrderingFilter

`OrderingFilter` 类支持简单的查询参数，以控制查询集的元素顺序。



默认情况下，url中的查询参数名为 `'ordering'` ，可以通过 `ORDERING_PARAM` 配置项进行自定义，和前面的 `search` 一样。

首先看一个根据用户名进行排序的url：

```
http://example.com/api/users?ordering=username
```

客户端还可以为字段名称加上'-'来指定反向排序，如下所示：

```
http://example.com/api/users?ordering=-username
```

也可以指定多个排序：

```
http://example.com/api/users?ordering=account,username
```

指定可以排序的字段

如果不在视图上指定 `ordering_fields` 属性，过滤器默认允许用户对序列化类上的所有可读字段进行排序。建议你明确地指定API可以在哪些字段上进行排序过滤，这有助于防止意外的数据泄漏，例如不小心允许用户针对密码字段或其他敏感数据进行排序。

这一操作可以通过在视图中设置 `ordering_fields` 属性来实现，如下所示：

```
class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    filter_backends = (filters.OrderingFilter,)
    ordering_fields = ('username', 'email')  # 看这里
```

如果你确定视图正在使用的查询集中不包含任何敏感数据，还可以通过使用特殊值 `'__all__'` 来明确指定可以在所有字段上排序。

```
class BookingsListView(generics.ListAPIView):
    queryset = Booking.objects.all()
    serializer_class = BookingSerializer
    filter_backends = (filters.OrderingFilter,)
    ordering_fields = '__all__' # 看这里
```

指定默认的排序字段

如果在视图中设置了 `ordering` 属性，则将把它用作默认的排序。

通常，你可以通过在初始查询语句上设置 `order_by` 参数来控制此操作，但是使用视图中的 `ordering` 参数允许你以某种方式指定排序，然后可以将其作为上下文自动传递到渲染的模板。如果它们用于排序结果的话就能使自动渲染不同的列标题成为可能。（好麻烦....）

```
class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    filter_backends = (filters.OrderingFilter,)
    ordering_fields = ('username', 'email')
    ordering = ('username',)  # 看这里，设置了默认结果按username排序
```

`ordering` 属性可以是一个字符串，或者字符串的列表/元组。

DjangoObjectPermissionsFilter

`DjangoObjectPermissionsFilter` 需要和Django Guardian模块一起使用，用于添加自定义的 `view` 权限。过滤器将确保查询集只返回用户拥有权限的对象。这个过滤器已经从3.9版本后被废弃，并移动到了 `djangorestframework-guardian` 模块中。

四、自定义通用过滤

要自定义通用过滤后端，需要继承 `BaseFilterBackend` 类，并重写 `filter_queryset(self, request, queryset, view)` 方法。该方法应返回一个新的过滤后的查询集。

除了允许客户端执行搜索和过滤之外，通用过滤器后端还可以限制当前请求或用户能够访问的对象。

比如，你可能希望限制用户只能访问他们自己创建的对象：

```
class IsOwnerFilterBackend(filters.BaseFilterBackend):
    """
    Filter that only allows users to see their own objects.
    """
    def filter_queryset(self, request, queryset, view):
        return queryset.filter(owner=request.user)
```

实际上，我们在视图中，通过重写 `get_queryset()` 方法，也能实现上面的操作。但是，编写自定义的过滤器后端，可以让你在不同的视图或所有的API上，方便的重用这一功能。

通用过滤器有时候也需要提供一个可浏览API页面上的过滤器控件。要实现这一功能，你需要实现 `to_html()` 方法，该方法返回过滤器被渲染过的HTML形式。方法的签名如下：

```
to_html(self, request, queryset, view)
```

方法返回的是渲染过的HTML字符串形式。

五、第三方模块

下面是一些第三方模块，提供了一些额外的过滤功能：

- Django REST framework filters package
- Django REST framework full word search filter
- Django URL Filter
- drf-url-filters