一、限流Throttling

限流类似于权限机制，因为它也决定是否接受当前请求。限流可以形象地比喻为节流阀，指示一种临时状态，用于控制客户端在某段时间内允许向API发出请求的次数，也就是频率。

你的API可能对未经身份验证的请求具有较强的访问次数限制，而对经身份验证的请求一般不怎么限制访问次数。

另一个你可能希望使用多个节流阀的场景是，如果您需要对API的不同部分施加不同的约束，因为某些服务需要特别的资源，防止服务器负荷过重，或者IO读取等待时间过长等等。

也可以同时使用多种限流措施，以同时应用突发节流率和持续节流率。例如，你可能希望将用户限制为每分钟最多60个请求，每天1000个请求。

限流不一定单指访问次数的限制，也可以是别的形式。例如，存储服务可能还需要限制带宽，而付费数据服务可能需要限制正在访问的特定记录的数量。

与权限和认证机制一样，可以同时使用多个节流阀，也是以类的列表的方式。在运行视图的主体代码之前，会逐个检查列表中的限流措施。如果任何限流检查失败，将引发`Exceptions.Throttled`异常，并且视图主体将不运行。

DRF默认的限流是使用Django的缓存机制，它的数据结构如下：

```
1  cache =  {
2
3  '注册用户的id或者匿名用户的ip地址': [timestamp,timestamp,timestamp,timestamp],
4  '192.168.1.112':
   [...,2019.10.1.12:00:11,2019.10.1.12:00:08,2019.10.1.12:00:03,],
5  'jack':[...,2019.10.1.12:00:10,2019.10.1.12:00:02,2019.10.1.12:00:01,],
6  ...
7
8  }
```

注意，时间戳在列表里是有顺序的，由最近到最早。

阅读该模块的源码会对你理解限流非常有帮助。

配置限流机制

在settings中，通过`DEFAULT_THROTTLE_CLASSES`和`DEFAULT_THROTTLE_RATES`配置项，为限流机制设置全局性的配置，例如：

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ),
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',  # 未认证用户一天只许访问100次
        'user': '1000/day'  # 认证用户一天可以访问1000次
    }
}
```

`DEFAULT_THROTTLE_RATES` 配置项中的时间周期单位可以使用 `second`，`minute`，`hour` 或者 `day` 。

应用限流机制

也可以为使用APIView，基于类的视图添加视图级别的限流措施：

```
from rest_framework.response import Response
from rest_framework.throttling import UserRateThrottle  # 导入
from rest_framework.views import APIView

class ExampleView(APIView):
    throttle_classes = (UserRateThrottle,)  # 看这里！注意参数形式

    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)
```

对于 `@api_view` 装饰器装饰的基于函数的视图，也是一样的：

```
@api_view(['GET'])
@throttle_classes([UserRateThrottle])  # 看这里，注意参数形式
def example_view(request, format=None):
    content = {
        'status': 'request was permitted'
    }
    return Response(content)
```

如何识别客户端?

既然要限流，那么必须识别客户端！那么DRF是如何判断和区分当前客户端的身份的呢?

DRF利用HTTP报头的`'x-forwarded-for'`或WSGI中的`'remote-addr'`变量来唯一标识客户端的IP地址。如果存在`'x-forwarded-for'`头部属性，则使用它，否则将使用WSGI中`'remote-addr'`变量的值。

在代理的情况下，如果想严格标识唯一的客户端IP地址，需要首先设置`NUM_PROXIES`来配置API后面运行的应用程序代理的数量。此设置应为大于等于0的整数。如果设置为非零，则一旦排除了任何应用程序代理IP地址，客户端IP将被标识为`'x-forwarded-for'`头中的最后一个IP地址。如果设置为零，则`'remote-addr'`的值将始终用作标识IP地址。重要的是要清楚，如果配置了`NUM_PROXIES`，那么NAT（网络地址转换）网关后面的所有客户机都将被视为单个客户机。

设置缓存

我们还可以在DRF中为限流措施设置缓存功能。

DRF框架提供的限流类需要使用Django的缓存后端。应该确保已经设置了适当的缓存设置。默认的`LocMemCache`缓存后端就比较合适了。

如果需要使用`default`以外的缓存，可以通过自定义Throttle类并设置`cache`属性来实现。例如:

```
from django.core.cache import caches

class CustomAnonRateThrottle(AnonRateThrottle):
    cache = caches['alternate']
```

自定义限流类后，不要忘了在settings中进行配置，或者在视图中添加类属性。

二、API 参考

DRF的限流类都定义在throttling模块中，这个模块很简单，一共250多行代码而已，有兴趣可以读读源代码。

主要包括下面几个类:

- BaseThrottle：限流类的基类，用于占位，提供三个方法`allow_request`、`get_ident`和`wait`
- SimpleRateThrottle：继承了BaseThrottle，添加和重写了一些方法，重点是添加了`get_cache_key`方法，但你必须自己实现该方法。

- AnonRateThrottle：继承了SimpleRateThrottle，仅仅是重写了 `get_cache_key` 方法
- UserRateThrottle：继承了SimpleRateThrottle，仅仅是重写了 `get_cache_key` 方法
- ScopedRateThrottle：继承了SimpleRateThrottle，重写了 `get_cache_key` 和 `allow_request` 方法

我们一般使用后三个类。

AnonRateThrottle

`AnonRateThrottle` 只会限制未经身份验证的用户。传入的请求的IP地址用于生成一个唯一的密钥。

允许的请求频率由以下各项之一确定（按优先顺序）：

- 类的 `rate` 属性，可以通过继承 `AnonRateThrottle` 并设置该属性来修改这个值。优先级高。
- settings配置文件中 `DEFAULT_THROTTLE_RATES['anon']` 配置项的值。优先级低。

`anonratetrottle` 适用于想限制来自未知用户的请求频率的情况。

UserRateThrottle

`UserRateThrottle` 用于限制已认证的用户在整个API中的请求频率。用户ID用于生成唯一的密钥。未经身份验证的请求将使用传入的请求的IP地址生成一个唯一的密钥。

允许的请求频率由以下各项之一确定（按优先顺序）：

- 类的 `rate` 属性，可以通过继承 `UserRateThrottle` 并设置该属性来修改这个值。优先级高。
- settings配置文件中 `DEFAULT_THROTTLE_RATES['user']` 配置项的值。优先级低。

一个API可能同时会进行多个 `UserRateThrottles` 类型的限流措施，比如每小时限制100次的同时每天限制1000次。那么在这种情况下，就需要自定义限流类。继承 `UserRateThrottle` 类，然后为每个子类添加一个 `scope` 类属性，如下所示：

```
class BurstRateThrottle(UserRateThrottle): # 第一个限流子类
    scope = 'burst'      # 主要用于控制爆发期的访问频率

class SustainedRateThrottle(UserRateThrottle):# 第二个限流子类
    scope = 'sustained' # 主要用于控制持续时期的访问频率
```

定义后，还需要在settings中配置：

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'example.throttles.BurstRateThrottle',    # 配置我们自定义的限流类
        'example.throttles.SustainedRateThrottle'
    ),
    'DEFAULT_THROTTLE_RATES': {
        'burst': '60/min',  # 每分钟最多60次，控制爆发期
        'sustained': '1000/day'  # 每天最多100次，控制持续访问
    }
}
```

`UserRateThrottle` 适用于限制每个用户的全局访问频率。

ScopedRateThrottle

`ScopedRateThrottle` 类用于限制对APIs特定部分的访问，也就是视图级别的限流，不是全局性的。只有当正在访问的视图包含 `throttle_scope` 属性时，才会应用此限制。然后，通过将视图的“scope”属性值与唯一的用户ID或IP地址连接，生成唯一的密钥。

允许的请求频率由 `scope` 属性的值在 `DEFAULT_THROTTLE_RATES` 中的设置确定。

看下面的例子：

```
class ContactListView(APIView):
    throttle_scope = 'contacts'  # 定义scope属性，并提供一个字符串，用于去settings中查找设置的值
    ...

class ContactDetailView(APIView):
    throttle_scope = 'contacts' # 同上
    ...

class UploadView(APIView):
    throttle_scope = 'uploads' # 同上
    ...
```

不要忘了添加下面的配置：

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.ScopedRateThrottle',  # 注册限流类
    ),
    'DEFAULT_THROTTLE_RATES': {
        'contacts': '1000/day',  # 字典的键，对应视图中的scope属性
        'uploads': '20/day'
    }
}
```

三、自定义限流类

要自定义限流类，请参照下面的步骤：

1. 继承 `BaseThrottle` 类
2. 实现 `allow_request(self, request, view)` 方法。如果请求应该被允许，那么方法应该返回 `True` ，否则返回 `False` 。
3. 可额外实现 `wait()` 方法。该方法应该返回一个建议的等待秒数（只有等待相应的秒数后，用户才可以尝试下一个请求），或返回None。只有当 `allow_Request ()` 方法返回 `False` 时，才会调用 `wait ()` 方法。

如果实现了 `wait()` 方法，并且当前请求被限流了，那么在响应头部将包含一个 `Retry-After` 属性。

下面是一个自定义限流类的例子，它将随机限制每10个请求中的一个：

```
import random

class RandomRateThrottle(throttling.BaseThrottle):
    def allow_request(self, request, view):
        return random.randint(1, 10) != 1
```