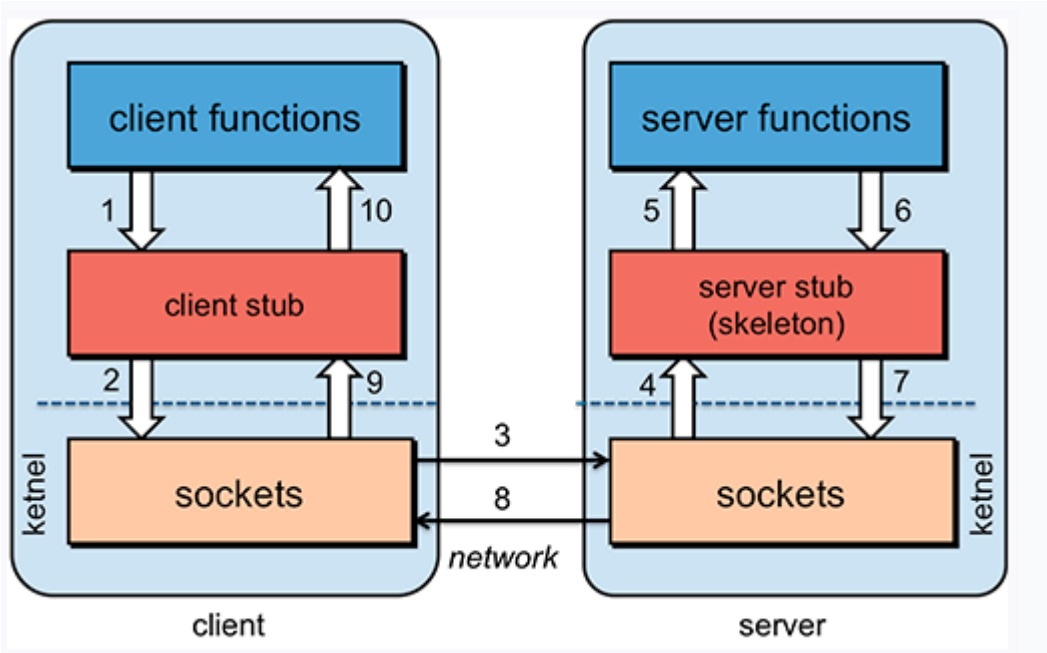
API

1.什么是webservice

> WebService就是一个应用程序向外界暴露出一个能通过Web进行调用的API，也就是说能用编程的方法通过 Web来调用这个应用程序。我们把调用这个WebService的应用程序叫做客户端，而把提供这个WebService的应用程序叫做服务端

2.什么是RPC

RPC 全称 Remote Procedure Call——远程过程调用。在学校学编程，我们写一个函数都是在本地调用就行了。但是在互联网公司，服务都是部署在不同服务器上的分布式系统，如何调用呢？ RPC技术简单说就是为了解决远程调用服务的一种技术，使得调用者像调用本地服务一样方便透明。 下图是客户端调用远端服务的过程：



1、客户端client发起服务调用请求。 2、client stub 可以理解成一个代理，会将调用方法、参数按照一定格式进行封装，通过服务提供的地址，发起网络请求。 3、消息通过网络传输到服务端。 4、server stub接受来自socket的消息 5、server stub将消息进行解包、告诉服务端调用的哪个服务，参数是什么 6、结果返回给server stub。 7、sever stub把结果进行打包交给socket 8、socket通过网络传输消息 9、client slub 从socket拿到消息。 10、client stub解包消息将结果返回给client。

一个RPC框架就是把步骤2到9都封装起来。

3.谈谈你对restfull 规范的认识?

> 一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件,规定如何编写以及如何设置返回值、状态码等信息

```
# ----API与用户的通信协议：  https ------

# ---------- 域名 --------------------
    #应该尽量将API部署在专用域名之下。
        https://api.example.com
```

```
    #如果确定API很简单，不会有进一步扩展，可以考虑放在主域名下
        https://example.org/api/

#----------- 版 本 -------------------
    #应该将API的版本号放入URL。
        https://api.example.com/v1/
    #另一种做法是，将版本号放在HTTP头信息中，但不如放入URL方便和直观

# ------------ 路径 ------------------
    #视网络上任何东西都是资源，均使用名词表示（可复数）
        https://api.example.com/v1/zoos
        https://api.example.com/v1/animals
        https://api.example.com/v1/employees
#  --------- method -------------------
    GET ：从服务器取出资源（一项或多项）
    POST：在服务器新建一个资源
    PUT：在服务器更新资源（客户端提供改变后的完整资源）
    PATCH：在服务器更新资源（客户端提供改变的属性）
    DELETE ：从服务器删除资源
# ---------- 过滤 ----------------------
    #通过在url上传参的形式传递搜索条件
        https://api.example.com/v1/zoos?limit=10：指定返回记录的数量
        https://api.example.com/v1/zoos?offset=10：指定返回记录的开始位置
        https://api.example.com/v1/zoos?page=2&per_page=100：指定第几页，以及每页的记录数
        https://api.example.com/v1/zoos?sortby=name&order=asc：指定返回结果按照哪个属性排序，以
及排序顺序
        https://api.example.com/v1/zoos?animal_type_id=1：指定筛选条
#---------- 状态码 ------------
    200 OK：客户端请求成功,一般用于GET和POST请求
    400 Bad Request：客户端请求有语法错误，不能被服务器所理解。
    301 Moved Permanently:永久移动,请求的资源已被永久移动到新url,返回信息会包含新的url,浏览器会自动
定向到新url
    401 Unauthorized：请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用。
    403 Forbidden：服务器收到请求，但是拒绝提供服务。
    404 Not Found：请求资源不存在，举个例子：输入了错误的URL。
    500 Internal Server Error：服务器发生不可预期的错误。
    502 Bad Gateway：充当网关或代理的服务器,从远端接收到一个无效的请求
    503 Server Unavailable：服务器当前不能处理客户端的请求，一段时间后可能恢复正常
# ------错误处理 ---------------
    状态码是4xx时，应返回错误信息，error当做key。
        { error: "Invalid API key" }
#------ 返回结果 --------------
    GET /collection：返回资源对象的列表（数组）
    GET /collection/resource：返回单个资源对象
    POST /collection：返回新生成的资源对象
    PUT /collection/resource：返回完整的资源对象
    PATCH /collection/resource：返回完整的资源对象
    DELETE /collection/resource：返回一个空文档
```

4.接口的幂等性是什么意思?

```
一个接口通过1次相同的访问，再对该接口进行N次相同的访问时，对资源不造影响就认为接口具有幂等性
    GET,    #第一次获取结果、第二次也是获取结果对资源都不会造成影响，幂等。
    POST,   #第一次新增数据，第二次也会再次新增，非幂等。
    PUT,    #第一次更新数据，第二次不会再次更新，幂等。
    PATCH，#第一次更新数据，第二次不会再次更新，非幂等。
    DELTE，#第一次删除数据，第二次不在再删除，幂等。
```

5.为什么要使用django rest framework框架?

```
# 在编写接口时可以不使用django rest framework框架，
# 不使用：也可以做，可以用django的CBV来实现，开发者编写的代码会更多一些。
# 使用：内部帮助我们提供了很多方便的组件，我们通过配置就可以完成相应操作，如：
'序列化'可以做用户请求数据校验+queryset对象的序列化称为json
'解析器'获取用户请求数据request.data，会自动根据content-type请求头的不能对数据进行解析
'分页'将从数据库获取到的数据在页面进行分页显示。
# 还有其他组件：
'认证'、'权限'、'访问频率控制
```

6.django rest framework框架中都有那些组件?

```
序列化、视图、认证、权限、限制
分页、版本控制、过滤器、解析器、渲染器
```

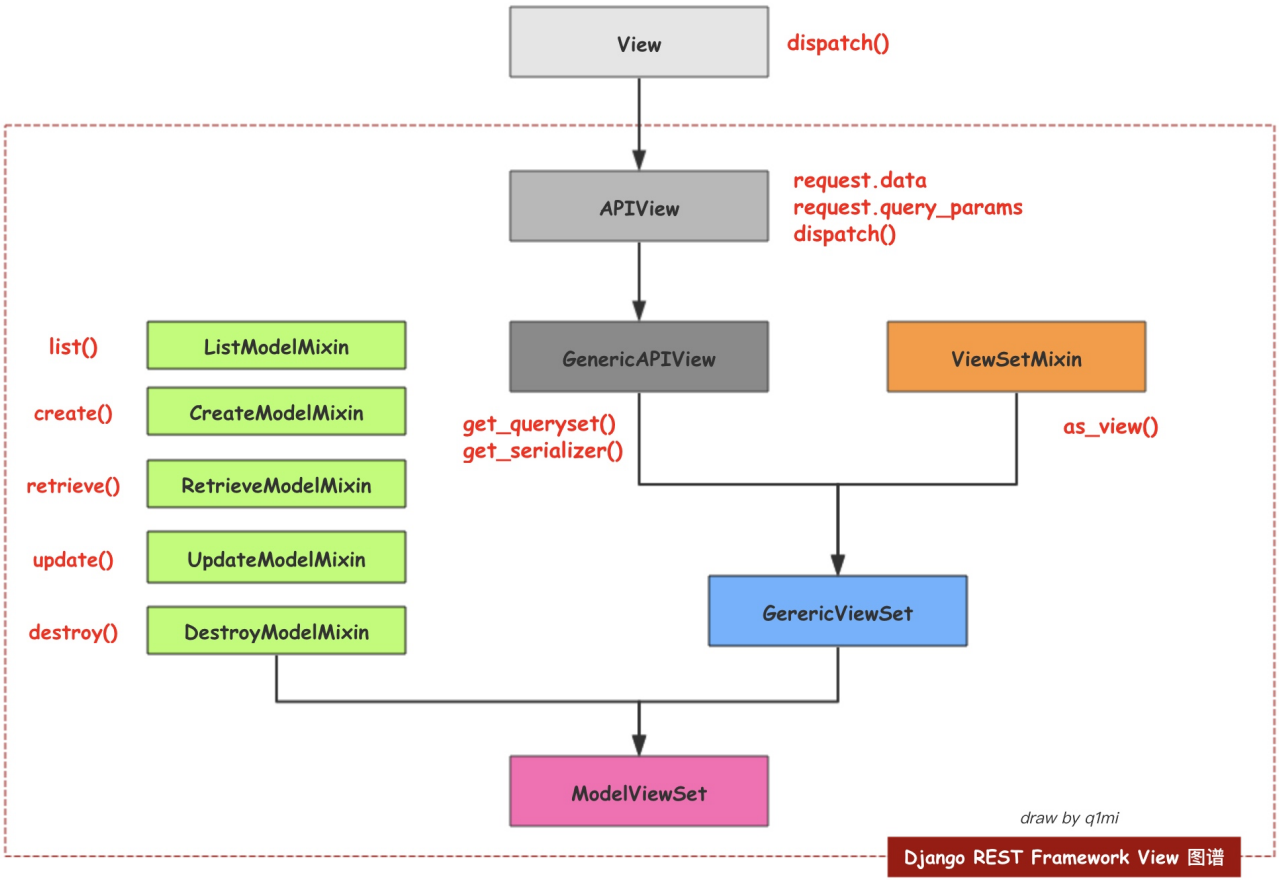
```
# --------版本控制 ----------
    1. 域名:   luffycity.com/api/v1
    2. URL参数luffycity.com/version=v1
    3. 在请求头中添加 version信息
    #配置:
        REST_FRAMEWORK = {
        ...
        'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.URLPathVersioning',
        'DEFAULT_VERSION': 'v1',  # 默认的版本
        'ALLOWED_VERSIONS': ['v1', 'v2'],  # 有效的版本
        'VERSION_PARAM': 'version',  # 版本的参数名与URL conf中一致
        }
        在视图中拿到版本信息:  request.version拿到当前这次请求的版本号

# ----------分页 ------------
#分页模式
    #PageNumberPagination   -->  按页码数分页，第n页，每页显示m条数据
        http://127.0.0.1:8000/publisher/?page=2&size=1
    #LimitOffsetPagination  --> 分页，在n位置，向后查看m条数据 例如:
        http://127.0.0.1:8000/publisher/?offset=2&llmit=1
    #CursorPagination  --  > 加密分页，把上一页和下一页的id值记住，页码都是随机字符串
        http://127.0.0.1:8000/publisher/?cursor=cj0xJnA9NQ%3D%3D

# ----------解析器 ------------
    #解析器的作用: 服务端接收客户端传过来的数据，把数据解析成自己可以处理的数据。本质就是对请求体中的数据进行解析。
   # Accept & ContentType请求头
        Accept是告诉对方我能解析什么样的数据，通常也可以表示我想要什么样的数据。
```

```
        ContentType是告诉对方我给你的是什么样的数据类型。
    #解析器工作原理的就是拿到请求的ContentType来判断前端给我的数据类型是什么，然后我们在后端使用相应的解析器去解析数据
# ---------渲染器 -------------
渲染器同解析器相反，它定义了框架按照content_type来返回不同的响应
```

7.django rest framework框架中的视图都可以继承哪些类




```
# ---- 继承APIView --------
# APIView继承于View ，但是重写了父类View中的dispatch()，将get、post、put的数据放入request。data中，将请求的参数放入request.query_params
```

```
#----继承GenericAPIView----
#每一个接口都是生成一个序列化对象，实例化，调用data方法，对其进行封装
class GenericAPIView(views.APIView):.
    queryset = None
    serializer_class = None
# python mixin (混合类) :不能单独使用，和其它类搭配起来使用（利用了Python支持多继承）
class PublisherList(GenericView,ListMixin,CreateMixin):
    queryset = models.Publisher.objects.all()
    serializer_class = PublisherSerializer
```

```
# -----继承GenericViewSet ----------
# GenericViewSet(ViewSetMixin, generics.GenericAPIView):
# ViewSetMixin重写了as.view()方法，实现了根据请求的方法执行具体的类方法
#路由注册的时候，利用actions参数，实现路由的定向分发 而不是简单的 反射
url(r'authors/$', views.AuthorViewSet.as_view(actions={'get': 'list', 'post': 'create'})),
# 作者列表
url(r'authors/(?P<pk>\d+)/$', views.AuthorViewSet.as_view(
        actions={'get': 'retrieve', 'put': 'update', 'delete': 'destroy'})
        ),
```

```
# -----继承ModelViewSet ------
#class ModelViewSet(mixins.CreateModelMixin,
                   mixins.RetrieveModelMixin,
                   mixins.UpdateModelMixin,
                   mixins.DestroyModelMixin,
                   mixins.ListModelMixin,
                   GenericViewSet)
#将不同的请求连接到不同的方法
```

8.django rest framwork 框架如何对QuerySet进行序列化?

序列化过程：ORM对象-->JSON格式数据
反序列化：JSON格式数据-->ORM对象

主要内容	小标题	Serializer	ModelSerializer
field	常用的field	CharField、BooleanField、IntegerField、DateTimeField	
	Core arguments	read_only、write_only、required、allow_null / allow_blank、label、help_text、style	
	HiddenField	HiddenField的值不需要用户自己post数据过来，也不会显式返回给用户。配合CurrentUserDefault()可以实现获取到请求的用户	
save instance		serializer.save()，它会调用了serializer的create或update方法	
		当有post请求，需要重写serializer.create方法	ModelSerializer已经封装了这两个方法，如果额外自定义，也可以进行重载
		当有patch请求，需要重写serializer.update方法	
Validation自定义验证逻辑	单独的validate	对某个字段进行自定义验证逻辑，重载validate_ + 字段名	
	联合validate	重写validate()方法，对多个字段进行验证或者对read_only的字段进行操作	
	Validators	1.单独作用于某个field，作用于重载validate_ + 字段名相似。 2. UniqueValidator: 指定某一个对象是唯一的，直接作用于某个field。 3. UniqueTogetherValidator: 联合唯一，需要在Meta属性中设置	
ModelSerializer专属	validate		重载validate可以实现删除用户提交的字段，但该字段不存在指定model，避免save()出错
	SerializerMethodField		SerializerMethodField实现将model不存在字段或者获取不到的数据序列化返回给用户
外键的serializers	正向	PrimaryKeyRelatedField，不关心外键具体内容	通过field已经映射，不关心外键具体内容
		嵌套serializer，获取外键具体内容	
	反向	Model设置related_name，通过related_name嵌套serializer	

serializers.serializer

序列化过程：（get请求）
定义一个serializer类，继承serializers.Serializer，定义字段告诉REST框架，哪些字段(field)，需要被序列化/反序列化
使用serializer类，将查询结果集QuerySet传入，并标明 many=True，表示序列化多个

```
    得到序列化的结果对象ser_obj，ser_obj.data即为得到的json格式的数据
反序列化过程：
    1.若是post请求提交数据 request.data即为提交的json格式的数据
      使用serializer类对数据进行反序列化 -->ser_obj对象
      对ser_obj对象进行is_vaild()校验，此处可以自定义校验规则，具体参照上述表格
      ser_obj.save() 需要重写serializer.create方法
    2.若是put请求：
        根据pk去查询具体的那本书籍对象obj
        获取用户发送过来的数据并且更新对象,赋值给instance的obj对象 ,partial=True的意识允许做局部更新
        ser_obj = Serializer(instance=obj, data=request.data, partial=True)
        对ser_obj对象进行is_vaild()校验，此处可以自定义校验规则，具体参照上述表格
        ser_obj.save() 需要重写serializer.update方法
```

serializers.ModelSerializer

```
#序列化过程  -  不用定义字段
    class Meta:
        model = models.Book
        fields = "__all__"
        # depth = 1  # 所有有关系的字段都变成 read_only
        # exclude = []  # 排除某个字段
        extra_kwargs = {  # 每个字段的一些额外参数
            'publisher': {'write_only': True},
            'authors': {'write_only': True},
            'category': {'write_only': True},
        }
#反序列化的过程：
#.save() 直接一键更新或创建，已经封装了这两个方法
#另外：SerializerMethodField 会自动去找 get_字段名 的方法执行
class BookModelSerializer(serializers.ModelSerializer):
    # SerializerMethodField 会自动去找 get_字段名 的方法执行
    category_info = serializers.SerializerMethodField(read_only=True)
    publisher_info = serializers.SerializerMethodField(read_only=True)
    authors_info = serializers.SerializerMethodField(read_only=True)

    def get_category_info(self, book_obj):
        return book_obj.get_category_display()
    def get_publisher_info(self, book_obj):
        return PublisherSerializer(book_obj.publisher).data
    def get_authors_info(self, book_obj):
        return AuthorSerializer(book_obj.authors.all(), many=True).data

    class Meta:
        model = models.Book
        fields = "__all__"
        # depth = 1  # 所有有关系的字段都变成 read_only
        # exclude = []  # 排除某个字段
        extra_kwargs = {  # 每个字段的一些额外参数
            'publisher': {'write_only': True},
            'authors': {'write_only': True},
            'category': {'write_only': True},
        }
```

9.简述django rest framwork 框架的认证流程?

```
#1.认证、权限和限制是在执行请求之前做的，路由-->as.view()-->APIView中的dispatch()方法
   中的 initial方法
   self.initial(request, *args, **kwargs)
#2.在initial函数中
   # Ensure that the incoming request is permitted
    self.perform_authentication(request)  #认证
    self.check_permissions(request)       #权限
    self.check_throttles(request)         #限制
#3.执行perform_authentication方法，其中request.user是一个@property的函数
#4.在user方法中通过 self._authenticate()函数，去执行_authenticate()方法，将当前请求的用户交给定义的
在 authentication_classes=[]中的类的 authenticate进行身份验证
#5.实现 BaseAuthentication中的authenticate方法，返回元组，元组的第一个元素赋值给
request.user 第二个元素复制给了request.auth  (token)
#6.若是authenticate方法 抛错后，捕获到报错(raise),此时执行的  _not_authenticated 方法的，return
结果 : user &auth 都赋值为None
```

10.Token：jwt 替代session存储的方案

11.简述django rest framwork 框架的权限实现

```
#1.认证、权限和限制是在执行请求之前做的，路由-->as.view()-->APIView中的dispatch()方法
   中的 initial方法
   self.initial(request, *args, **kwargs)
#2.在initial函数中
   # Ensure that the incoming request is permitted
    self.perform_authentication(request)  #认证
    self.check_permissions(request)       #权限
    self.check_throttles(request)         #限制
#3.执行 check_permissions方法，从当前 permission_classes列表中，执行has_permission()方法，判断有
没有权限
#4.实现BasePermission中的has_permission()方法
class MyPermission(BasePermission):
    message = '只有VIP才能访问'
    def has_permission(self, request, view):
        #通过上面的认证源码得知:当不输入token参数或者未登录,则 user ,auth 均为None,当auth存在则此时
的user不为None
        if not request.auth:
            return False
        #当有Vip才有权限访问
        #if request.user 当前经过认证的用户对象
        if request.user.vip:
            return True
        else:
            #如果不是Vip就拒绝的范围
            return False
#5.如果存在验证不通过，那么就执行self.permission_denied，然后这个异常在dispatch函数中被捕捉，当做结果传
递给response
```

12. DRF如何实现用户访问频率控制(匿名用户, 注册用户)

```
#1.若是未注册用户，所以不可能经过认证，则此时user,auth 均为None
#2.自定义allow_request方法
    #拿到当前的请求的ip作为访问记录的key
    #把当前的请求的访问记录拿出来保存到一个变量中
    #循环访问历史,把超过10 秒钟的请求事件去掉
#在视图或者全局中进行配置
```

throttle.py

```
#使用内置限制类
from rest_framework.throttling import SimpleRateThrottle
class VisitThrottle(SimpleRateThrottle):
    scope = "xxx"
    def get_cache_key(self, request, view):
        return self.get_ident(request)
```

```
   #在settings文件中进行配置
   "DEFAULT_THROTTLE_CLASSES": ["BAR.XXX.VisitThrottle", ],
    "DEFAULT_THROTTLE_RATES": {
        "xxx": "1/s",
```