1. Vagrant:

<https://www.virtualbox.org/wiki/Downloads>下载并安装Oracle-Vm-VirtualBox

https://www.vagrantup.com/downloads安装vagrant

https://app.vagrantup.com/hashicorp/boxes/bionic64下载ubuntu镜像

将镜像改名为bionic64.box 即虚拟机

命令行运行vagrant box add hashicorp/bionic64 bionic64.box 添加虚拟机

hashicorp/bionic64是虚拟机名称，bionic64.box即镜像文件

vagrant box list 查看是否安装成功

虚拟机配置：

1. 将vagrantfile.temp（改名为vagranfile作为虚拟机配置文件）,

requirements.txt （项目依赖的三方的python包）,

provision.sh, （初始化开发环境的脚本）

mysql-apt-config\_0.8.15-1\_all.deb （mysql配置文件）

复制到我们的工程目录，比如weitter主目录下面

1. 进入工程目录 vagrant up 启动虚拟机
2. vagrant ssh 登录虚拟机
3. cd /vagrant 进入我们的工程目录

以后若改了vagrant的一些配置，比如vagrantfile的端口映射，需要运行vagrant halt再vagrant up进行重启后才可以应用。

Django：

django-admin startproject weitter

setting.py中可以配置数据库，以及允许访问的host

python manage.py migrate把配置中的数据进行数据迁移，根据setting.py在mysql中生成对应表

运行 python manage.py runserver 0.0.0.0:8000

Pycharm配置：

配置成vagrant虚拟机上的python环境: Settings -> project:weitter ->python interpreter -> 配置按钮 add -> 选择vagrant

**Django每次新加一个app，比如requirement.txt里，或者通过python manage.py startapp appname时，都要在settings.py中的INSTALLED\_APPS中加入这个新的app**。

python manage.py test运行单测， 加上-v2 可以看具体步骤

数据迁移：

Python manage.py makemigrations 准备迁移，会在migrations目录下生成一个py文件做操作

Python manage.py migrate

TIPS:

1. 后端设置数据库表单时间字段，比如create\_time/update\_time，都要用UTC时间
2. Model代码新增一个字段，即数据库表单新增一个字段时，都要在代码的model中设置该字段null=True而不是用default设置，因为migrate时default会把for循环所有旧数据都加入default值，会把数据库卡死，导致正常无法访问。另外新增后更新服务器端代码时，要先migrate再去deploy new code
3. Model代码删除一个字段，数据库表单删除一个字段，需要先重启加载新代码（即deploy new code），再通过migration去删除表单 （若先把数据库里字段删除的话， 会导致代码中字段比数据库中的多，使得服务器端失败）
4. 这样的话，新增一个字段和删除一个字段，不可以同时上线，否则不论是先migrate数据还是先deploy新代码，都会出错

数据库索引：

https://www.bilibili.com/video/BV1aE41117sk?p=1&vd\_source=bb0f99a516f816dfe3381cef39cc31f8

1. B+tree存储：k是索引值，v是数据地址或者具体数据，且(k,v)对只存在于叶节点，中间节点只有k值。每个节点中是按索引大小存索引，叶子结点像链表一样连起来，也是根据索引大小连起来的（方便做range query）。

* 聚集索引：所有数据存在b+树的叶节点的k，v中的v中。（innodb表）
* 非聚集索引：b+树叶节点k，v中v存的是数据的地址，要想访问数据，需要再去这个地址上去查找，这样需要查找两次。（myisam表，不支持事务操作）
* Innodb表必有主键，且推荐是整形自增的，若建表时没有声明或者不存在这种唯一标识主键，innodb会自己创建一栏作为主键，作为聚集索引的k。
* Innodb表若设置了uuid作为主键，这样比整型存储更多，做比较时更耗时。
* 自增是因为叶子结点按照索引值顺序链起来，用于范围查找，uuid不是自增的维护困难，每次都要插入到节点中，因为每个节点大小是固定的（模式16kb），需要破坏后面的每一个叶子节点做分裂，性能很差。
* 非主键索（也叫二级索引）引叶节点v存主键，查到主键后再去主键索引二次查找（也叫回表查询）

1. 联合索引：最左前缀，（a,b,c）索引，以a排序a相同的以b排序。。。，a=1可以用索引（因为a有序）,b=1无法用索引（索引失效，因为b不是有序的）
2. 索引失效：
   1. 联合索引违反最左前缀法则（如上）
   2. Like xxx%可以用索引，%xxx不能用索引 （与最左前缀有点相似）
   3. 对索引列函数运算无法用索引
   4. 表数据量比较小，可能不会走索引，而直接走全表扫描
   5. In会用索引，not in不会用索引
   6. 。。。。等等，比如能用二级索引就用二级，而不用一级（比如查询主键， 二级索引叶节点就记录了主键值了，可以不回表直接返回数据，二级叶节点存储的数据更少，更方便查询）

MYSQL工程生产环境中的注意点:

1. 在实时相应的环境中，JOIN很少用，因为更耗时，O(n方)复杂度，（两个表的笛卡尔乘积），离线环境可以用。
2. 不要用cascade级联，因为delete很容易造成多米多效应导致很多数据被删除
3. DROP ForeignKey constraint，不用外键的约束
   1. 上面级联删除的问题
   2. 每次update表都要去查外键关联的表，更加耗时，而且数据库性能开销更大，容易成为瓶颈
   3. 多并发时会加锁，很容易导致死锁
   4. 扩展性问题，数据迁移更麻烦。分库分表更麻烦
   5. 代码中一个数据实体a的外键往往是一个model类的实体b，我们想得到b.id然而却通过a.b访问时很容易造成N+1query再次去查询，造成性能问题
4. Unique\_together，django对数据库的唯一性约束，比如对于用户的一些连续点击post的处理（接口幂等性问题），这是在数据库层面进行的，我们很难在webapp代码中进行这种唯一性的检测，因为并发问题，很可能2个并发线程都通过了唯一性条件后再进行了一些创建，这样会破坏唯一性了，所以需要在数据库层面进行约束。

Django/web：

1. 对于serializer，调用create前都需要调用is\_valid（），在is\_valid中会对data进行validate成validated\_data, 来作为后面create()的参数，当然我们也可以实现validation这个函数，在is\_valid（）时也会顺带调用自己重写的validation。
2. 可以在创建serializer实例时传入context=a这个参数， 以便在自己的serializer中重写的各个函数中使用a
3. Restapi的方法名若想与具体url中不同，比如方法名为get\_follower，url中为get-follower，可以在@action中添加url\_path这个参数来设置。
4. N+1问题：有外键的表，若取到外键后，很容易去for循环中query查询具体值，会造成for循环中有n次query，即n+1问题，一般解决是使用预加载，查询时用 in的方式解决。
5. N+1 query 问题，除了常见的循环中query，还有一点要注意的是用到serializer时，因为会有对于model的映射，也是有可能出现n+1的要注意下。
6. 403错误：forbidden，403突然增多很可能是有黑客、恶意用户
7. 400错误：bad request，400突然增多可能是接口变了，但是后面数据处理没有做相应改变，比如版本升级、api升级，但是没有兼容老版本的api，导致老版本用户依然用旧接口数据类型访问，尤其是app的client端，经常有老版本的请求。
8. 外键中related\_name值可用作主表对从表的反向查询，没有设置的话也可用小写的从表model名\_set来做。
9. django的通知-监听机制：在各处代码中有listener注册到signal中，当某件事发生时，signal向各个listener发信号send，各个listener接收到信号后进行处理。但是这种机制上实践并不好，因为listener的代码分布在各个地方，对于修改代码/维护有很大成本。
10. django中查看sql query，可以对queryset.query来查看，比如xxx.objects.filter().query

Git：

1. Git commit、push之后代码有问题，需要重新修改rollback到之前的commit
   1. Git log 取到之前的commit的标识串abcdf
   2. Git reset abcdf 撤回到上次改动前
   3. Git add.
   4. Git stash 这样把上次带动的先存起来以防万一的备用（git stash pop来取出备用）
   5. 此时再次git log会发现之前的那次commit已经没有了
   6. Git push origin –f branchname， 用本地branchname强制覆盖掉服务器对应的branch
2. 但是一般不这么rollback，因为多人协同情况下，别人pull下来会造成严重冲突，更好的方式还是重新提一个commit把之前的修改删除掉，继续增量更新而不是回滚。
   1. 用git revert操作
   2. 取到要撤销的commit标识符fjklmn
   3. Git revert fjklmn 算是对这个commit的反向commit
   4. Git push origin –u branch 继续做增量更新即可
3. Git pull与git fetch区别：
   1. Git pull基本等于 git fetch + git merge
   2. Git pull更加暴力而且隐藏了一些merge信息，更好的用法还是git fetch后再去git merge，这样让我们对本地库的commit和远程库的commit有更清晰的认识
   3. 比如本地的commitID=1，远程有commitID=1，后面又有人更新到了commitID=2,
   4. 我们git fetch的话，本地会有commitID=1的版本和commitId=2的版本，共两个版本，我们再去git merge程新的commit，比如commitID更新成了3
   5. 但是git pull的话会强制将本地的commitID=1的版本更新成commitID=2
   6. 故从结果上来看git pull与 git fetch+git merge没有区别，但是从commitID来看两者还是有区别的

Python：

1. Decorator：@functools.wraps(func)将被装饰的函数的一些属性比如\_\_doc\_\_(注释信息)\_\_name\_\_(函数名) \_\_annotation\_\_(参数、返回值等) 替换成func的上述属性，经常用于装饰器函数内的inner()用来保存原函数的信息。
2. Decorator写法：被装饰函数func（\*args, \*\*kwargs）， 第一层def decorator(func) 返回inner，第二层def inner（\*args, \*\*kwargs）做具体的装饰处理，func的参数即为inner参数。若func有返回值，则inner也要返回同样的。若装饰器有参数比如@decorator(param=p)

则再多包一层用来传参数

1. 循环依赖，比如

a.py: from b import bb aa=1 def f(): x=bb

b.py: from a import aa bb= 2 def f(): y=aa

这样会造成循环依赖，我们python会逐行解释执行，并将变量等内容存入内存，比如 执行a时，

* 1. 把a放入sys.modules中，
  2. 执行到from b import bb时，会把b放入sys.modules中
  3. b逐行执行, from a import aa 发现a已经放啊入sys.modules但是却找不到a.aa，因为a还未执行到aa=1时就跳到b.py去import了，这就是循环依赖问题

解决方式：可以把from a import aa放到函数f()中，因为函数b.py加载时不会运行函数内部代码，只是把函数名加载到b的命名空间去，而不会执行函数，而当我们调用b.py中的函数f（）时，a.py已经完成了加载

小知识点：

1. 推送：push notification （GCM-google cloud message， APNS-apple push notification service）
   1. 手机app客户端向GCM注册token
   2. App将token发送给app服务器比如twitter服务器
   3. Twitter服务器device表单记录token、user信息（比如手机版本等）
   4. 推送信息产生时，拿device表中的对应账户的token发送到GCM，GCM根据token推送到具体的client。
   5. 关闭某个功能的推送（比如关闭twitter评论时的推送，或者发推时的推送等，可以将这种功能的switch存在preference里， 这样发推时去preference查一下就知道要不要推送了）
2. User与userprofile一对一关联存储在两个表单，而不是一个表单
   1. 虽然他们是一对一的可以存一个表单，但是不合适，因为profile是经常会被用户改动的，若存在同一个表单，一旦改动，那么一些仅涉及user的功能（比如登录）也需要因为user被改动了而造成登录失效，和cache miss，从而要重新登录以及reload到cache，对于性能不好。
3. cache aside vs cache through:
   1. cache aside 旁路缓存，cache与db是分开的，web端与他们分别交互，偶尔在cache和db会存在数据不一致，需要cache设置timeout来刷新cache来进行一致性化，cache更新使用delete来操作，这种架构最简单最广泛应用。
   2. cache through 穿透式缓存，cache与db在一起，他俩相当于被包在一起了，web与包在一起的进行交互，这种结构可以避免数据不一致，目前redis实现这种架构。
4. 扩展阅读：scaling memcached at facebook
5. db相关数据更新时，我们用delete cache而不是更新cache，原因是delete能尽量确保cache与db一致，避免因为update造成的race condition。

TODO/问题：

1. Views中的queryset的意义是什么，重载get\_queryset（）？
2. 18课的likes等与之前模块不同的难点，以及后面一些答疑可以重看(已经重看过了)。
   1. Likes的content\_type实现时关注下mysql中django\_content\_type表里的内容是否有增加。
3. 返回某个weit前几个/3个评论，以及评论一个评论的功能