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。

MySQL Replica相关log问题--WAL（Write Ahead Log）技术

1. WAL: 数据库写入机制，首先事务commit之后，数据发生更新，变动先写入内存， 然后将对应变动写入log, 再异步的写入数据库本身的磁盘。因为log是对磁盘的顺序写入，所以io更快，而数据库本身磁盘写入往往是随机或者无序的，磁盘寻址磁头移动导致io更慢。
2. Mysql 有3个log比较关键：redo log（数据页变动的log，存储引擎层，大多数用于单机上crash后的恢复），undo log（与更新反向操作的log）和binlog（基于statement或者row的逻辑日志， mysqlserver层的，用于主从数据同步和恢复）
3. redo log：存储引擎层产生的log，记录具体数据库物理数据页的变动，用来在crash后恢复数据库到最新的commit提交。redo log循环写入，写满之后再从头开始重新写入。由于他是在存储引擎层记录对于数据页的变动，所以对于多个主从数据库，若是不同的数据引擎的数据库，无法通用，若是不同版本的数据库，数据页上可能也会有微小的不同，这样也不建议使用。
4. undo log：记录与当前更新相反的操作的逻辑日志，用来回滚事务中的操作，保证事务原子性。
5. binlog：在mysql server层记录当前操作（基于语句statement或行row）的逻辑日志，用来进行主从库的同步，以及基于时间点的数据还原。binlog是顺序写入，log文件满了之后，新开一个文件继续写入。基于statement是指记录的是类似sql的更新语句，但是由于一些像now（），rand（）或者一些用户定义的函数无法保证主从的数据相同，而且用于并发导致主从表的自增id也可能不同，这样where查询时会有问题。基于row的是记录每行的具体变动，这种方式有更好的一致性。

事务特性：acid, 原子性，一致性，隔离性，持久化性

隔离级别：read uncommitted, read committed, repeatable read, serializable

1. 脏写，所有级别都不会出现：t1 read x=1， t2 read x=1 write x=2 and commited, t1 rollback x=1, 一个事务的写入被同时的另一个事务回滚了。
2. 脏读（dirty reads）：t1 read x=1 and write x=2, t2 read x=2, t1 rollback x=1, t2出现脏读，读取了t1未提交的数据，出现在RU级别
3. 不可重复度（non-repeatable Reads）: t1 read x=1, t2 read x=1 and write x=2 commit, t1 read again x=2, t1同一个事务两次读取获得的值不同，出现在RC级别，针对的是update操作一行数据
4. 幻读（Phantom Reads）：t1 read x<5 得(1,2,3), t2 insert x=4 and commit, t2 read x < 5 again得(1,2,3,4)，事务读取某个范围数据时，由于其他并发事务导致的获得值不同,一般是多行数据，insert操作，non-repeatable read是一行数据，出现在RR级别。
5. mysql默认RR级别，oracle默认RC级别

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 signal机制：比如监听某个表的增添和删除 post\_save/pre\_delete, 只针对save（）和delete（）两个操作，但是调用bulk\_create()，update()时不会触发，这个要注意，所以在用到不会触发signal的函数方法时，要手动的写代码去完成listener里的功能。
11. django中查看sql query，可以对queryset.query来查看，比如xxx.objects.filter().query
12. F() expression 可以直接在数据库中操作字段的值，而不需要把字段load到内存，再从内存改完后存会数据库a) 这样无须load效率更高 b) 可以避免race condition,比如下面代码的+1就是直接在数据库的update原子操作

from django.db.models import F

reporter = Reporters.objects.get(name='Tintin')

reporter.stories\_filed = F('stories\_filed') + 1

reporter.save()

1. Q() 可以用来逻辑连接复合查询条件，比如

Poll.objects.get(

Q(question\_\_startswith='Who'),

Q(pub\_date=date(2005, 5, 2)) | Q(pub\_date=date(2005, 5, 6)))

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. 后段数据库更改的上线部署操作（比如一个表单/库拆成2个，mysql换成hbase）
   1. 写新的db的代码
   2. double wirite，两个数据库都要写入（可能一个是更新，另外一个没有这个值需要新建），保证新数据在新数据库里出现，旧库一起更新，且是source of truth
   3. migrate script 数据库迁移，把旧数据一点一点的批量迁移到新数据库
   4. 校验，以旧数据库为准进行校验修正新库的错误（比如migrate时旧值改掉了double write的新值，double write时新库写入失败等）
   5. double read，线上读数据时两边一起读，遇到不一致，以旧库为准，并记录下来后续进行修改。需要监听目前的错误率进行监控。
   6. 5步记录下来的，运行第4步进行修改，两种步骤循环进行直到不一致为0
   7. 去掉旧库

memcached:

1. 可使用telnet方式访问
2. 纯内存的缓存，重启就清空了，存的字符串
3. 可以通过telnet 127.0.0.1 11211 来连默认的端口运行memcached命令行
4. flush\_all刷新缓存
5. 获得所有的key：运行stats items获取所有items，根据items后面的id，运行stats cachedump id 0 来获取该id对应的key，get key就可以获取这个key的缓存内容了
6. 扩展阅读：scaling memcached at facebook
7. 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。

Redis：

1. redis-cli 命令查看
2. web层面的代码一般不加锁，加锁在db层
3. 五种数据类型：string，list，set，hash，zset（sorted set）
   1. string：set <name> <value> set user1 “testuser1”;

get <name> get user1 -> “testuser1”

* 1. hash: hmset <name> <key1> <value1> <key2> <value2> ...

hmset user1 username “testuser1” nickname “testnickname1”

hget <name> <key> hget user1 username -> “testuser1”

* 1. list: lpush/rpush <name> <value1> <value2>... lpush users user1 user2 user3 ...

lrange <name> <startid> <endid> 范围是开区间 lrange users 0 -1

* 1. set: sadd <name> <value> sadd users user1

smembers users

* 1. zset: zadd <name> <score> <value> 以score来排序 zadd users 0 user1

zrangebyscore users 0 -1

1. 存入redis要对orm的objects/queryset进行serialize，从redis取出后进行deserialize。
2. 测试时clear cache，但是production上不可以clear redis的cache
3. 若对于cache的数据进行update，目前不支持，但是若进行此种修改的话，因为数据存在了redis的list中，难以查找到该数据的所有cache，可以1.直接从redis删除掉update的相关list，重新从数据库读取2. redis的list只存id，每个数据（tweet）存到memcached的缓存里面，取数据时进行2次查找，先从redis拿id再去memcache中取具体数据。这种二级缓存更加复杂一些。
4. 在newsfeed/tweet增加时，我们都会用signal机制自动的去redis中lpush一条数据，但是fanout时候我们会bulkcreate，这个不用触发signal，此时需要手动写代码把新的newsfeed push到缓存中。
5. 缓存也要设置大小，我们不能无限制的缓存数据到redis中，通常我们缓存最新的前几百个比如200个到redis中，若用户访问到更旧的数据，则直接从数据库读取，此时的流量就少了很多了。所以我们在list这个方法中要对分页数据进行判断，先去cache取，redis中的都取完了再去数据库中取。
6. cache的操作：
   1. 读cache，则读取后看是否存在如cache.get(key), 查看是否存在，存在就返回，不存在就去数据库里面读取
   2. 写cache，也是先看cache中key是否存在比如conn.exists(key)，存在就更新该key的值，不存在也是直接去数据库里面取，此时取出后不需要更新值了，因为数据库里就已经是最新的了，这也是我们要确保的，更新cache前先更新数据库的。

冗余存储denormalization：

1. twitter项目，对于likes\_count，comments\_count进行了冗余存储，将他们作为字段存在tweet/comment表里，便于直接取数据，否则需要进行select count（\*）from。。。来做count方面的查询，这样比较慢，而且用户经常查看的一个列表页，这样要对列表中每一个tweet都进行select count操作，更慢了，所以我们进行了denormalization，将count冗余的存在表里。
2. 可能会有冗余存储的不一致问题，比如count字段与select count（\*）出来的数据不同，因为有可能在count+1/-1操作时失败（数据库挂了），这样我们可以有两种方式进行一致性处理：
   1. 每1000次请求，进行一次select count(\*)的操作进行校正，更新到count字段
   2. 每隔一段时间，运行异步任务进行select count（\*）的操作refill，更新到count字段
3. 因为likes\_count这种字段是经常变化的，而我们cache了一个tweet，且把likes\_count冗余存在了tweet表里，那like\_count变化tweet的属性也就变化了，此时若更新缓存或者删除对应缓存，是不适合的，毕竟likes\_count的变动频率太快了，这种情况下，我们把count的数据分开来单独做一份缓存。

消息队列异步任务(Celery: distributed task queue)：

1. webserver会把一些任务放入消息队列，作为异步任务来执行，比如发邮件，消息推送，twitter中fanout操作，因为比较耗时，会放入异步任务队列中，其中消息队列可以用RabbitMQ， Redis等，我们可以用celery作为异步任务的调度工具，也叫distributed task queue分布式任务队列，负责把任务（task，一般是一些任务的要执行的代码标识）放入broker（消息队列，RabbiitMQ， Redis），worker作为任务执行单元去broker取任务来执行，任务执行结果放入backend中。
2. worker的执行结果可以a. 直接写入数据库，b. 也可以返回webserver，通过webserver处理再存到数据库。需要看情况考虑，若一般情况下可以直接写入数据库；若有安全问题，还是返回webserver，因为若直接写入数据库，会有安全风险，用户可能会hack数据库（比如leetcode的代码评测，用户可能会写一些hack代码），所以这种情况下不允许用户直接访问数据库，而是返回webserver进行处理。
3. 若一个异步任务太大，比如fanout的follower很多有1m个，这样一个任务运行的时间空间都太大，在执行过程中失败的可能行更大，代价更大，所以可以把一个任务拆成几个来做，比如每个任务处理1000个followers，一共拆出1000个子任务，这样每个子任务失败代价更小，即使重试也更简单。具体实现上我们额外用一个async task异步任务去拆分，再分发给其他异步任务，这样可以对web和异步任务的代码进行解耦。
4. 常用用途：1. webserver的执行慢的任务进行异步执行，优化用户体验2. 削峰，当数据库的操作数量很大时，用MQ进行削峰操作，对于数据库的操作慢慢的来，缓解数据库的压力。
5. 若存在不同的异步任务，比如fanout，短信验证码，fanout若执行时间很长可能会拖延后面的验证码任务，此时我们就需要把任务设置优先级或者分成不同的队列，这样后面生成的不同队列任务不会被卡住。

限流器：

1. 项目中用到ratelimit用来保护数据库的恶意多次访问
2. 原理：
   1. 方法一：把key和对应的次数记录到cache中(因为是临时用到的数据，不需要持久化，所以没必要用db)，key一般是对应的限流的具体api + ip/user（比如create\_tweet.user\_id\_1000），value是对应的限流次数，每次访问次数减一直到0则不能继续访问了，另外还需要设置timeout，即限流器具体配置对这个key设置expire时间。缺点：过期时间总是从第一次来记录，比如1分钟5次，若其他4次访问集中在最后的时间，那么过期之后立刻来了新的3次访问，都会通过，但是最近几秒钟其实是出现了7次的访问量的，大于我们限制的1分钟5次，会有一些问题，此时的限流没有针对最近一分钟内的限流，而是针对第一次访问之后的一分钟。
   2. 方法二：在上述cache基础上，key再加上此时timestamp，value为这个timestamp的访问次数，从0开始。比如1分5次限流的话，每秒都有一个key，新的访问来了，会计算新timestamp之前的59个timestamp对应key的value （cache可以用get\_many方式一次读取获得多个key的值），逐个相加，看是否大于5次，大于就不能访问了。对于不同限流器的时间分钟/小时/天，存在不同的bucket里面。

关于api访问流量的监控

1. 可以像百度统计那样从前段埋点js代码进行统计
2. 后端监控，可以把数据存到cache中

HBase：

1. kv数据库一种，高并发，分布式，写优先
2. Thrift(protobuf in Google) :RPC and serializer
3. hbase vs redis: 都是kv存储，hbase支持对key的范围查询
4. rowkey 可以理解为mysql的pk，支持排序（若有排序需求的字段则需要设置到这里，或者想用联合索引的话，可以把多个字段组合在一起存到rowkey中，一般经常把两个字段组合，一个用来标识比如id、name等，一个用来排序比如时间等），想对不同的字段排序，则需要建立不同的冗余表，columnkey不支持排序。
5. rowkey中若想存整数，则需要把他补零，转化为所有整数都同等长度的字符串，方便字符串排序
6. 情景：若短时间内来了很多数据，生成了相似的rowkey（尤其是有时序性的，比如常见的自增id+timestamp的rowkey），存在了相同的region或者机器上，导致他比其他region或者机器更热，数据不平衡，（cassandra的rowkey不负责排序只负责标记，columnkey进行排序，这样可以把rowkey打散），hbase可以将rowkey部分反转（比如id部分），这样既可以把rowkey打散，又可以保持后半部分（用于范围查询的那部分）的排序功能，这样可以分散到其他机器上。
7. 可以参考“hbase rowkey的设计”的相关文章
8. zookeeper：分布式协调服务，会部署到每台机器上一部分数据，一般是配置数据，配置变动时产生网络通信进行每台机器的配置更新。

关于数据库上线

Linux:

1. contrl + r, 反向查询之前输入过的cmd

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
4. memcache 缓存user和tweet，都是单个粒度的缓存，但是我们经常是列出很多user和tweet，这种情况下用不上缓存吧？若用到缓存，则会有a.objects.get(id=xx)的情况，这样单个取出来后合并成一个list，是否就形成n+1？这里最好自己去实际操作一下，看看memcache的缓存情况以及sql的执行情况。答案：第一次取确实是这种情况，每个都单独的去数据库里取了。