Redis和Memcached的共性和区别：

1. 二者都是key-value结构，也就是哈希表，不同的是Memcached是纯粹的key-value，类似java中的HashMap<String, String>，django会使用pickle把对象序列化为string
2. Memcached所有数据都在内存中，访问速度快，但断电或重启后数据就没了，需要从DB中回填到cache，一旦不能及时回填，request就会访问DB，这样会极大增加DB的压力，这就是缓存击穿。Memcached存储数据灵活性不够高，它只具备简单的哈希表get和set，例如之前讨论过的如果缓存了一个id列表，当两个request同时往缓存中增加一个新的id时，需要先把之前的列表拿出来，加上新id，然后set回去，这样有可能导致结果不正确，需要先删除之前的缓存，然后重新set回去，即使这样也有可能出问题。而Redis则支持多种数据结构，存储的value可以是string，set，list，hashmap，zset(有序集合)。像之前的存储list，然后添加新id的操作，Redis就支持添加和删除的原子操作。

backup指备份，replica指副本，二者最大区别是backup一般是离线的，每隔一定时间进行备份，而replica需要实时修改，你对正本数据进行的修改，都要实时在replica上进行，这样当正本数据不可用时，才能无缝切换到replica上。

Redis的数据也存储在内存中，但支持持久化到硬盘上，可以用作数据库、缓存和消息代理。

在一台机器上，如果Memcached能支持1M的qps，即每秒100万次查询，那Redis就可以支持100k的qps，HBase能支持10k的qps，MySQL只能支持1k的qps。

对于普通配置的笔记本电脑，MySQL可能就是几十的qps，可以对应算出Memcached，Redis的qps。

缓存雪崩：cache avalanche

当某一个时刻出现大规模的缓存失效的情况，那么就会导致大量的请求直接打在数据库上面，导致数据库压力巨大，如果在高并发的情况下，可能瞬间就会导致数据库宕机。这时候如果运维马上又重启数据库，马上又会有新的流量把数据库打死。这就是缓存雪崩

产生的原因一是缓存挂了，二是大量的缓存同时都到了失效时间。

解决办法：

1. 在原来的失效时间上加一个随机值，避免大量缓存同时到期。
2. 可以设置缓存服务器集群，避免某台服务器宕机，导致的雪崩
3. 使用熔断机制，防止过多的数据请求同时打到数据库上
4. 通过限流器限制访问DB的频率。

缓存击穿：cache breakdown

类似于缓存雪崩，缓存雪崩是大规模的key失效，而缓存击穿是一个热点的key，有大并发集中对其进行访问(例如吴亦凡跪了，吴亦凡对应的key)，而这个key失效了，导致大并发全部打在数据库上。

解决办法：

1. 如果业务允许，可以设置热点key不失效。
2. 使用互斥锁，缓存失效时，只有拿到锁才可以查询数据库，当然这样会阻塞其它线程，导致系统的性能变差
3. 如果是Memcached的缓存，可以像之前提到的Facebook那样把第一个访问数据库的请求通过，其它请求等待一定的时间，然后把数据回填到缓存中。
4. 对于Redis而言，可以设置集群，将热key分布到不同的服务器上，均衡读流量

缓存穿透：cache penetration

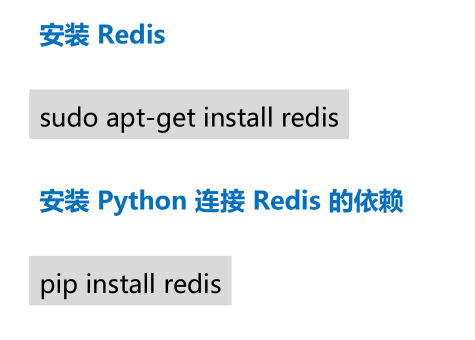
无论是Memcached还是Redis的缓存，都是通过key来查询的，如果发送的请求的key不存在，则就不会在缓存中查询到，就会直接打到数据库上。如果有大量的这样请求，很可能是非法或恶意的请求。

解决办法：

1. 做好参数校验，如用户权限校验，id基础校验等，非法的参数直接返回错误
2. 缓存空对，把无效的key存进Redis，如果Redis和数据库都查不到数据，就设置value=null
3. 使用布隆过滤器来过滤掉无效的key。

布隆过滤器是将所有的key都映射(hash映射)到一个很长二进制向量上，它可以判断某个key“一定不存在和可能存在”，这样就可以过滤掉无效的key，缺点是存在误判的可能性，删除起来比较困难。

安装Redis：



sudo apt-get install redis安装的是一个redis的server，会一直运行，而pip install redis本质上安装的是一个client，可以通过用它的接口访问Redis sever。

可以像使用Memcached那样在settings.py中设置Redis后端处理，但那样的话就没法使用Redis丰富的数据结构了，Django没有兼容Redis的数据结构，只能像Memcached那样简单存储string，所以需要自己对数据进行序列化和反序列化。本项目中实现了JSONEncoder，它继承自django的DjangoJSONEncoder，可以把object序列化为json(还可以考虑序列化为xml，但也有datetime时间精度丢失的问题)，只是在处理datetime时，没有像默认的那样删除掉小数点后的部分位，而是都保留下来。mysql数据库中对datetime保留了小数点后6位精度，在程序中要全部留下来。

尝试在redis中只缓存tweet\_id。

尝试在redis中缓存comment

打log，把每个request的运行时间打印下来，使用一个中间件。

pagination翻页时，以时间戳作为标准进行排序，这也是之前序列化时一定要保留至少小数点后6位的精度的原因。通常用timestamp+random(6)，即在时间戳后加6位随机码。这种加随机码的方式在web系统中很常见，以新浪微博为例，短网址url中后面的一串无意义的字符，就是随机码。如果有重复的，就再随机一次。

不使用PRI KEY的原因，一是mysql数据库的自增id在不同的机器上会产生重复的id，而是有些数据库有可能就没有自增id，例如HBase。这样就没法进行分布式。

mysql同一个table不能存放在多台机器上，但不同的机器上的mysql可以有相同名称的table。

缓存tweets列表时，方法A：可以将tweet整个都缓存到redis中，方法B：可以只缓存tweet\_id到redis。

方法A的好处是读取速度快，使用方便，直接从缓存中取出tweet列表进行渲染，占用内存空间比较大，而方法B则需要拿到id后还要再去cache或数据库进行取数据，占用内存空间比较小。

方法A缓存tweets可能还没有问题，但缓存newsfeed可能会有问题。

方法A和方法B的写入操作其实差不多，没有太大影响。

方法B的好处是支持修改和删除，方法A很难支持修改和删除，如果要修改和删除，只能清空cache，尤其是修改tweet，无法知道缓存列表中的tweets是那个tweet被修改了。而且如果自己发的tweet修改了， newsfeed也采用方法A进行缓存，这个tweet可能被分发给了100个好友的newsfeed中，那就需要去newsfeed中把对应的tweet也给修改掉，但这个好友关系是有可能发生变化的。找到之前的好友关系是很困难的。

方法B在redis中tweet缓存id，如果修改了tweet，可以在其它cache中存储修改后的tweet，然后通过id去读取cache或数据库就得到最小的tweet。newsfeed中缓存的也是tweet\_id，读取时可以查看是否和tweet对应的user还是好友关系，如果是就继续读，如果不是就不读了，并且读取也是从cache和数据库中读取，而这时读到的也是最小的。

cache newsfeeds时，可以考虑把明星用户单独拿出来，采用pull模式来处理，这需要使用k路归并排序。

csrf攻击：

context：共享的文本信息，给NewsFeedSerializer传递了context参数后，drf会把它传递给NewsFeedSerializer中的包含的其它serializer中。

自己写的代码最好不要用Signal机制，只在操作Model相关的保存和删除时，才使用Signal。

各个基于Signal的listener之间的执行顺序无法控制，所以必须保证它们之间相互无关，如果有关就要把有关的部分保证起来。

Signal的connect最好在Model定义的后面，因为这样在py文件被加载时，就会执行connect，保证之后Model保存和删除时，已经关联上了。

在项目中的Signal包括post\_save,pre\_delete等方法中，需要添加sender，因为有多个py文件中包含了这些方法，只有指定了sender，例如sender=Like，那么只有Like对象保存或删除时才会触发指定的方法。