翻页Pagination操作只针对list方法，其它的方法不需要翻页。

传统的翻页：可以指定page，page\_size等参数



django中PageNumberPagination用来实现传统翻页。一定要设置一个max\_page\_size，否则无限制的话，取数据会太慢，而且有安全问题，因为我们不太希望数据被轻易拿走，需要限制api的访问频率，所以不能让一次request就拿走所有数据。

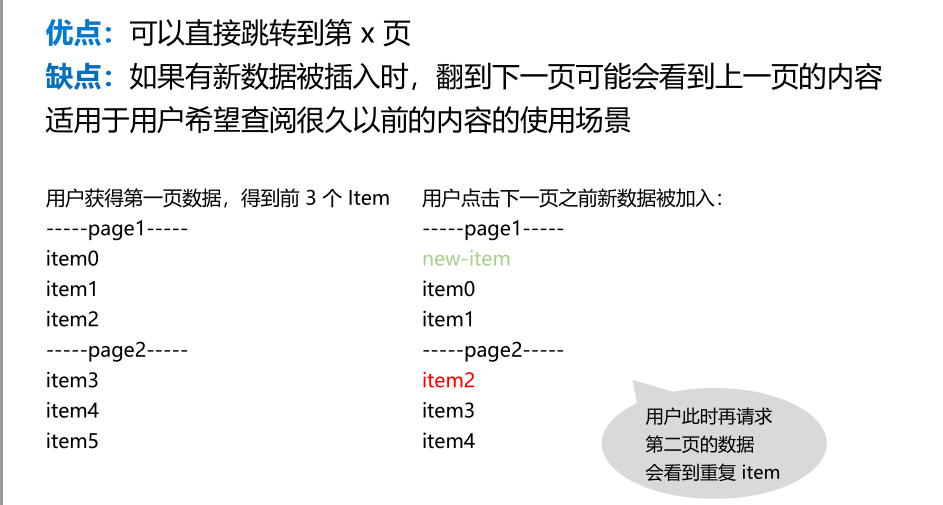


Tweet.objects.all()[50:100]，

这句话执行时是直接翻译成上面的sql语句，而不是先全部取出来，然后再切片。

mysql中的索引index中不仅存储了结点地址信息，也会存储每个结点下存储了多少条数据，所以OFFSET可以利用索引提高查询速度。

传统翻页的优劣：适合更新频率很慢的系统。



社交类平台不适合用传统的翻页方式，要使用无限翻页，Endless Pagination。

上滑(手机屏幕向上滚动)是向下翻页，翻出来的是更旧的数据



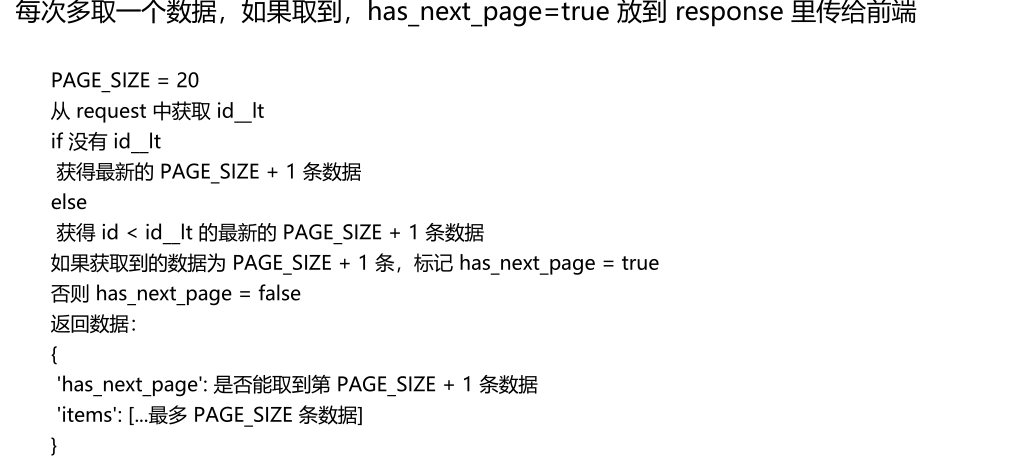
这里需要降序排列，这样取出的是小于指定id的最大的几条数据。

无限翻页其实没有第几页的概念，只有相邻页的概念。

判断是否有下一页：

不能先查找下一页的内容，如果发现返回为空内容，才判断是否为空。因为不仅浪费了一次数据库查询，更重要的是浪费了用户的一次滑动，用户体验很不好，而且前端在滚到底部时显示的提示也无法及时显示出来。

正确的做法每次翻页查询时多取1条数据，比如一次翻页20条，但request请求是多取1条数据，如果能多取一条，说明有下一页，否则就是没有，但显示时只显示20条。



下滑是向上翻页，查找最新的数据。



这里的排序规则和上滑的不一致，原因是要找的是比指定id大并且靠近它的数据，需要是升序排列，即id从小到大排列，否则取出来的数据和指定id并不挨着。

memory是基于进程隔离，每个进程都有自己的一片内存空间。

为什么要使用cache，因为cache的访问速度快，Memcached所有数据都存放在内存中，显然访问内存的速度比访问硬盘的速度要快很多。

数据库本身会自带一些cache，但是无法控制那些数据cache，那些不cache，另外数据库为了保证数据一致性，会导致cache hit rate即命中率(就是访问某个数据中从缓存中访问到的概率)不一定很高，而且数据库cache会有容量限制，这样就不可控。

User这个表单的cache hit rate大概能达到98%。

cache访问速度和数据库访问速度基本是百倍到千倍的差距。

Facebook曾对Memcached进行了优化，其中有一个机制是防止“缓存穿透”，所谓缓存穿透指的是如果有大量的query都同时去访问cache中的某个key，但该key不存在或失效了，这样大量的query都会去访问DB，这就是缓存穿透，会导致DB挂掉，为此Facebook设计了一个机制，只让第一个query去访问DB并回填到memcached中，其它query等待一个很短的时间后，再去访问cache，这样通常就有了key。

TTL(Time to Live)，存活时间，配置中是TIMEOUT，单位是秒，也就是缓存中某个key自动的存活时间，超过这个时间后，缓存会自动失效。

删除key同样会带来数据不一致的问题，删除key后，会重新从数据库读取数据库并回填到cache中，如果这时有request更新了数据库，同样会造成数据不一致的问题。

当某个key对应的内容发生变化，删除key带来的数据不一致的问题远低于修改key。

对于社交系统而言，数据不一致性通常是通过TIMEOUT来解决。

Memcached通常适用于社交Web系统，不适用于金融系统，金融系统对数据一致性要求很高。

如果希望某个key常驻在Memcached中，应该怎么处理？

Memcached和web服务器通常不在一台服务器上，如果加锁需要加分布式锁，不建议使用分布式锁，效率非常低，原本使用Memcached就是为了提高效率。

Memcached更适合缓存一个一个的对象，User信息不会经常改变，例如你的用户名和头像不会经常改变，在真实系统中，cache hit rate往往高达98%以上，对User表单的请求，98%都可以使用cache，不需要去访问数据库。

Django的代码中，models.py一般是底层的代码，它不要再去依赖其他的代码，像services.py, serializers.py都是去依赖models.py。

Model中的方法一是不需要其他参数，只和它自身有关，二是代码量不要太长，如果太长就放到Service中。

在单元测试Newsfeed缓存tweet和user时，发现通过get方法去获取newsfeeds时，从缓存中读取tweet时，打印的日志如下：



可以看出先从cache中get到Tweet，然后Tweet从cache中获取到User，乍一看逻辑不是很通顺，毕竟User是Tweet中字段，应该是先User后Tweet，这比较符合包含关系，毕竟C++中的继承和包含都是类似的。产生这样效果的原因是：读取缓存cache是在serializer中完成的。NewsFeed执行get方法时，数据库查找的结果会通过NewsFeedSerializer进行序列化，此时会从缓存中读取tweet，读取后的tweet对象会通过TweetSerializer再进行序列化，而TweetSerializer会从缓存中读取User对象，这样就产生了上述的输出效果。

