删除tweet：Memcached中对应的单独的tweet可以存储tweet object，而tweet list这些只存储tweet id，渲染翻页时，for循环看看相应的id是否被删除了，就不展示了，并继续向下查找直到凑够一页的tweet。

queryset是一种懒惰加载方式，当它产生过数据库请求后，之后再访问，就不会再进行数据库请求了(如果使用iterator()来访问会重新产生数据库请求)，这也意味着如果数据库有更新，这个queryset也不会体现出来。

Memcached和Redis都缓存了tweet，其实是希望在Redis中缓存tweet id list，先从Redis中读取到tweet id，然后根据tweet id去Memcached中获取到具体的tweet对象，不过目前项目中直接在Redis中缓存了tweet list，但newsfeeds 缓存在redis中的list中存储的是tweet id，渲染newsfeeds时，需要从Memcached中获取tweet对象，还可以尝试在tweet view retrieve中进行优化，从Memcached中获取单个tweet对象。

中间件的是一种统称，辅助Web Server和数据库的其实都可以叫中间件，例如Redis，Memcached，Message Qeuue等。

Django中的MIDDELWARE，对每个request或reponse都要去做的操作，才放到中间件中。

Csrf：Cross-site request forgery，跨站请求伪造。request的发起者是前端，可以伪造一个request，进行金融诈骗等，可以在网页或APP上伪造了一个对话框和按钮，看起来是一个正常按钮，实际是一个转账操作，如果这时候你的支付宝等金融工具处于登陆状态，如果你点击了这个按钮，就会转账给骗子。

Csrf防范：但凡是post请求，还需要一个csrf token，这个token的时效性非常短，可能刷新一下，几分钟就过期了。这样就可以判断是否是在支付宝的环境下进行转账，如果不是，request就会被拒绝掉。

Django中的CsrfViewMiddleWare会把你做这些操作。

在Twitter项目中，打log可以添加到MiddleWare中，查看每个request的执行时间，这样可以针对性的进行优化，有现成的例如datalog, apm等，可以自己写一个MiddleWare。

给一条sql语句，应该怎么优化？

mysql中有个slowquerylog，可以进行配置，例如设置10秒以上的都是slow query，这样就会把所有的slow query给记录下来，然后看看怎么优化。

1. 是不是有join语句，如果有，看看能不能优化掉，例如采用prefetch\_related方式或者cache方式，尽量不使用join。
2. 是不是索引失效了？还是压根就没有索引？如果没有索引就添加索引，mysql中可以通过explain，在一条正常的sql语句前面添加explain

例：explain select \* from auth\_user where username=’xxx’，如果走了索引，会在possible\_keys记录。

点赞数和评论数的去标准化，虽然有可能导致获取的点赞数和评论数会不一致，但可以加快速度，在社交系统中这很重要，比一致性重要，而金融系统一致性最重要。

点赞数和评论数标准的取法是从数据库表中采用SELECT COUNT(\*) FROM Table，例如TweetSerializer中包含点赞数和评论数，前端渲染tweets列表时，显示时有10个tweet，那么点赞数和评论数就分别需要10次Query，这会导致N次Queries，而且sql语句中的COUNT(\*)也比较耗时，导致总的查询非常耗时。

解决办法是点赞数和评论数进行denormalization去标准化，都存储在tweet表中，相当于冗余存储。

所谓denormalization就是之前标准化获取点赞数和评论数只能通过SELECT COUNT(\*) FROM table来获取，现在可以直接从tweet的记录中直接查询出来。好处是查询tweet记录时就可以查询到点赞数和评论数，不用额外的查询，坏处是可能会产生一致性的问题，这时需要明确一个source of truth，即绝对准确的来源，点赞数和评论数的source of truth显然是SELECT查询表。

这种不一致的问题，可以不处理，例如点赞数和评论数就不需要处理，如果处理，可以采用随机的方式，在一定概率下取SELECT查询表来获取点赞数和评论数，大致代码如下：





mysql中这种操作是原子操作，它有row lock的功能，但不能先取出来，加上1，再写会去，这样无法保证是原子操作。

点赞操作时不要尝试把cache中的tweet和tweets列表失效掉，以保证likes\_count会被更新，这样做如果用户时明星用户，那么就会频繁的失效cache中的tweet，会降低访问速度。同理评论操作也不要失效cache。

点赞数和评论数会经常改变，要单独缓存到Redis。一般来说经常容易改变的对象不应该缓存在memcached或Redis中，除非像点赞数和评论数这样cache提供了相应的+1和-1操作，而且要单独缓存，不要和其他数据一起缓存。

异步任务(async task)：客户端发出的请求，有可能是短时间内做不完的，如果让用户一直等待，用户体验不好，好的做法是告诉用户需要等待或者表示服务端已经在做了，保证能完成，用户可以干其它的。例如，大规模的广告邮件，例如发10万封，点击发送时肯定没法很快完成，这时就需要异步任务。

Message Queue：用于任务分发的中间件，主要用于：

1. 处理一些耗时很长的操作
2. 处理一些需要重试机制的操作
3. 需要延迟或者定时处理的操作，例如微信红包的退回操作，每隔一定时间给用户发邮件。

常用的Message Queue有Redis，RabbitMQ。

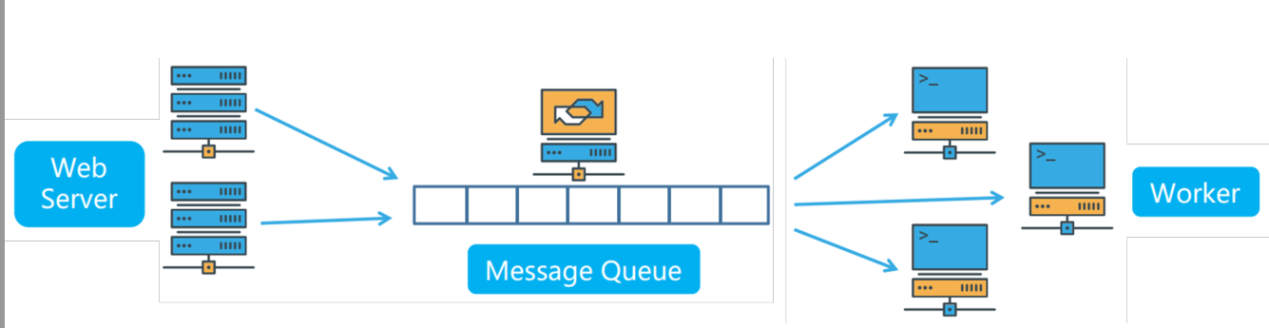
前端：手机端，PC浏览器

大前端：前端+类似django web server

后端：Message Queue。

Message Queue使用场景：

Web Server在处理一个request的时候，将一些操作放到异步任务的代码中，将任务所需的参数，丢给Message Queue，专门处理异步任务的worker机器从Message Queue中监听到新的任务后，获得任务参数并执行任务。



工作流程：

1. 用户请求request到web server。
2. web server创建异步任务并投递给Message Queue
3. web server会给用户response，但不是回复任务的结果，毕竟这个时候任务可能还没有结果，甚至还没有开始，而是告诉用户已经在处理这个任务了。
4. worker机器监控Message Queue，并从中领取任务去执行
5. 任务的结果有几种处理方式，一种是worker直接把结果写入数据库中，另一种方式通过回调的方式，把结果返回给Web Server。

这是一个典型的生产者-消费者模式，尝试用python写一个生产者-消费者模型，用python中的Queue。

Message Queue就是一个broker。

Celery：负责与Message Queue Broker沟通的python库(不依赖Django)

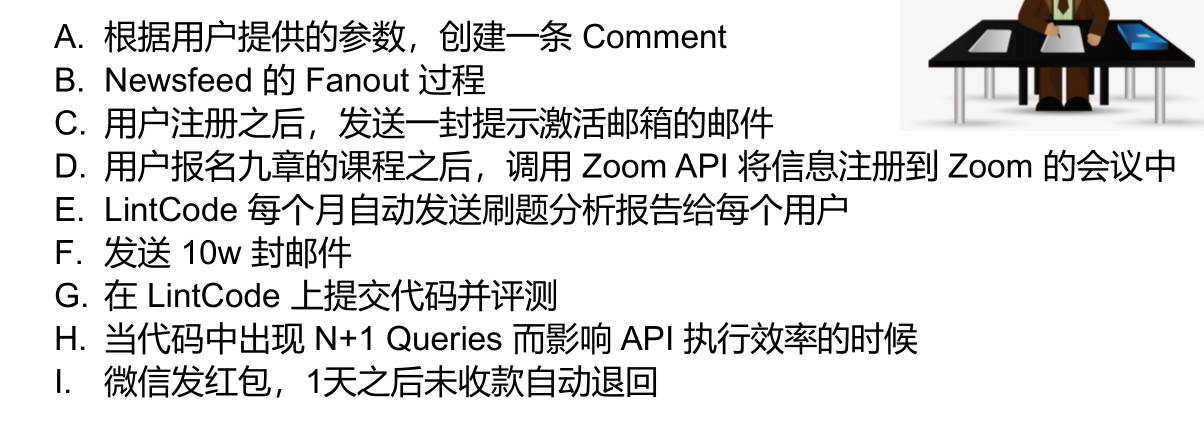
同步任务(sync task)：去餐馆吃饭，队伍很长，但你只能排着，无法去干其它的事情，直到餐馆有位置了，你进去吃饭。也就是操作没结束，需要一直等待。

异步任务(async task)：去餐馆吃饭，排队并登记，然后你就可以去逛商城，等到手机收到短信通知你可以去吃饭了。

Woker：执行异步任务的机器，和Web Server并非同一台机器。通常是单独的集群，单独管理。通常Web Server和Woker的比例在20:1左右。

Queue：任务队列，一个Broker中可以有多个队列(就像在海底捞排队，两个人的位置排队和4个人的位置排队肯定是不同的)，每个任务需要定义自己将被扔进哪个队列里，每个worker的机器也可以定制自己要监听的队列有哪些。

以下哪些场景需要使用Message Queue。



A：不需要，创建comment速度很快，而且我需要很快知道comment是否创建成功，对结果的及时性要求很高。

B：需要，fanout过程有可能很慢，而系统只需要告诉我tweet发成功了，但不需要告诉我所有的粉丝都已经收到这条消息。

C，D：需要，所有第三方都是不可信的，凡是涉及到第三方接口调用的，都应该放入到异步任务中。发送邮件会调用第三方发邮件的服务，发送邮件速度很慢，而且调用第三方的API，都需要进行重试。自身调用第三方的api，那么自身肯定是正常的，而第三方则有可能挂掉，所以需要重试。

E：需要，定时任务，需要使用Message Queue，而且也比较慢。是cron job+MessageQueue，cron job定时创建一个任务投递到Message Queue中。。定时任务在linux上有cron job，在celery中也有cron job。一般来说，需要根据各种环境进行选择。如果项目中使用的话，运行环境依赖项目中代码，那就最好别选linux上的cron job。

F：需要，10w封邮件非常慢，不仅需要Message Queue，还需要白10w邮件拆分成多组，由多台worker进行处理。

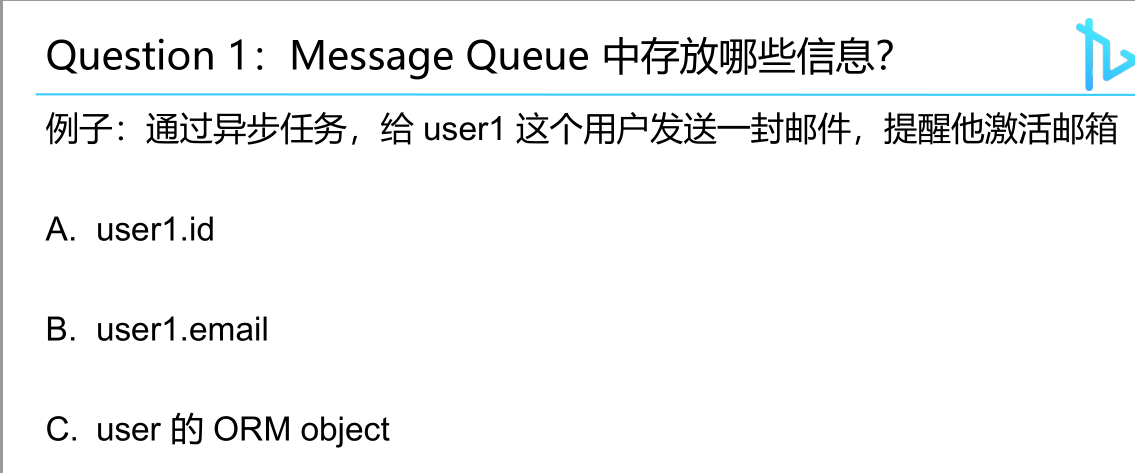
G：需要，提交代码，会出现pending->running->accepted，评测并不会很快，pending表示在排队，running表示有worker在评测了。

H：不需要，必须等待运行结果，N+1 Queries的解决办法不是Message Queue，例如优化代码，优化需求，使用cache

I：需要，延迟任务(只执行一次)，但不是定时任务(会执行N次)，将红包退回的任务投递到Message Queue中，然后等到一天之后再执行，看要不要回收。

不能使用类似cache中TTL，超时时间的方式去处理红包退回，因为TTL只是将数据失效掉，而红包退回会有后续的一系列操作，例如发送红包退回的提醒等。

Message Queue中存放的信息：

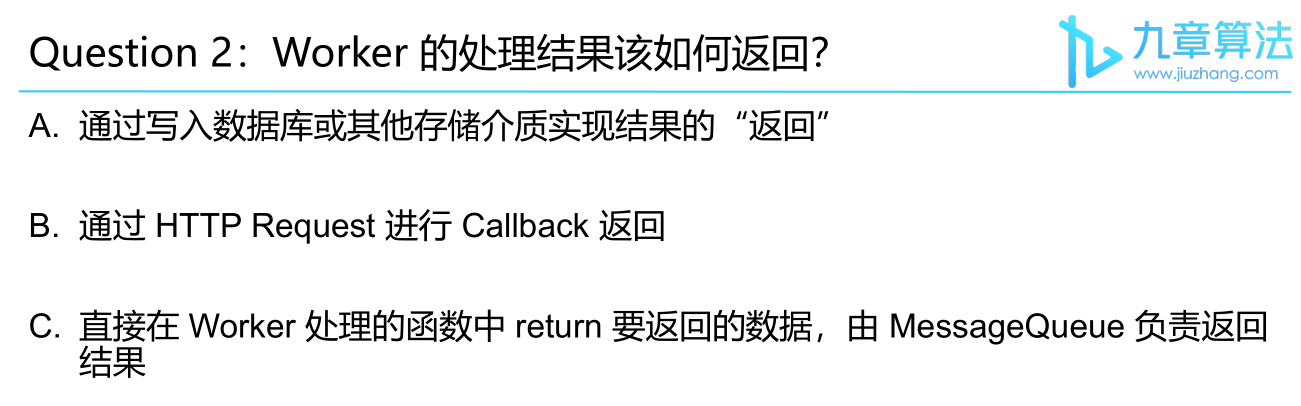


答案是A

首先排除C，python中object是一个引用，也是一个内存地址，Message Queue和Web Server不是一台机器，根本无法使用。

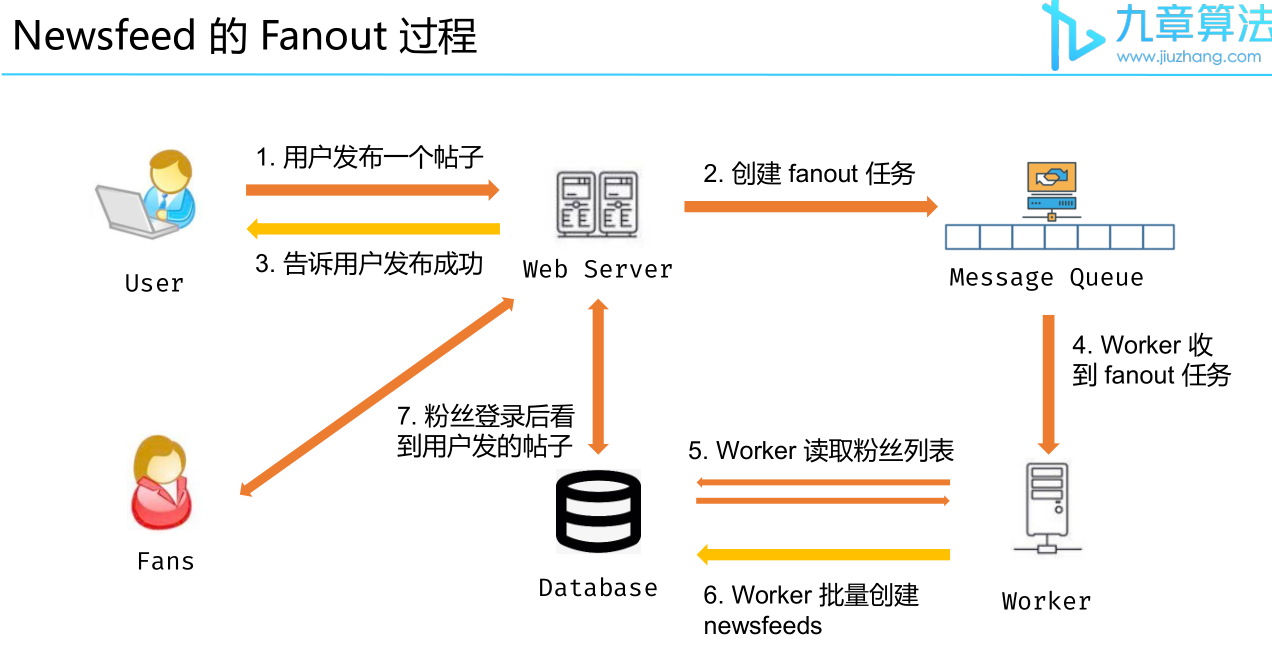
Message Queue需要存放任务的标识，通过id能找到user，email等，而email不一定是唯一标识符，而且发送邮件需要不仅仅是email。

可以把ORM object进行序列化传给Message Queue，不过这样做不好，序列化后的占用的内存更多，Message Queue中的内容太多可能会挂掉，而且序列化后的信息不一定够用。而Worker可以通过user.id能够从数据库和cache查找要用的信息。

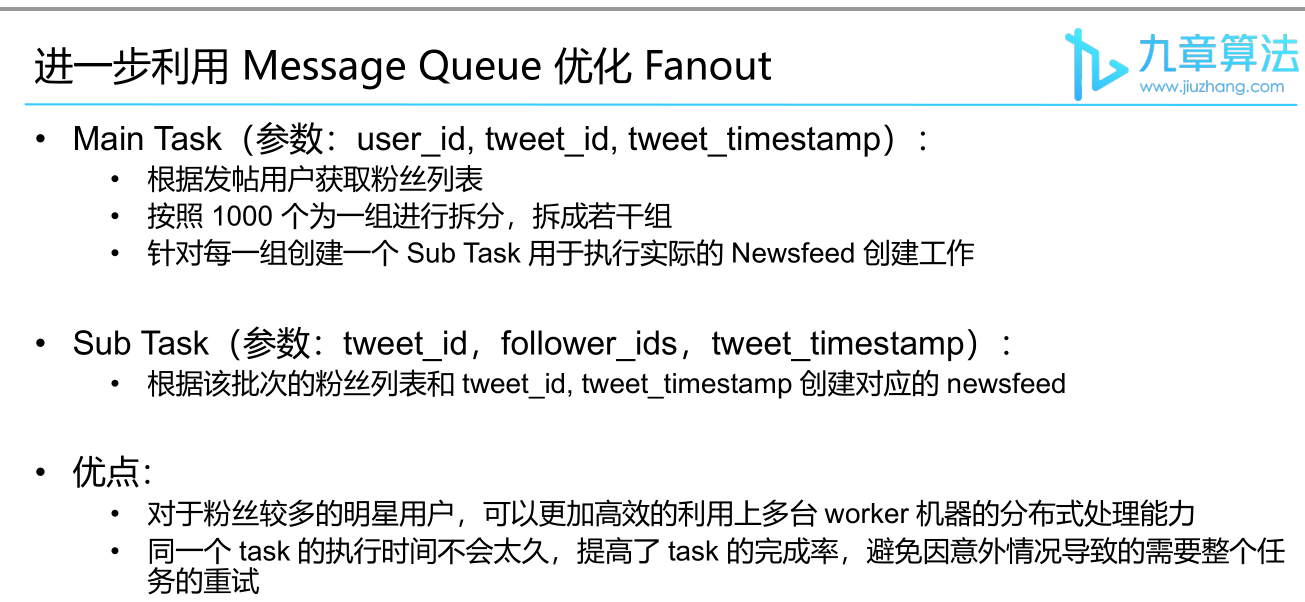


答案是：A,B

C的语法违背了Message Queue的本义了。Message Queue本来就是为了不等待函数执行完毕。

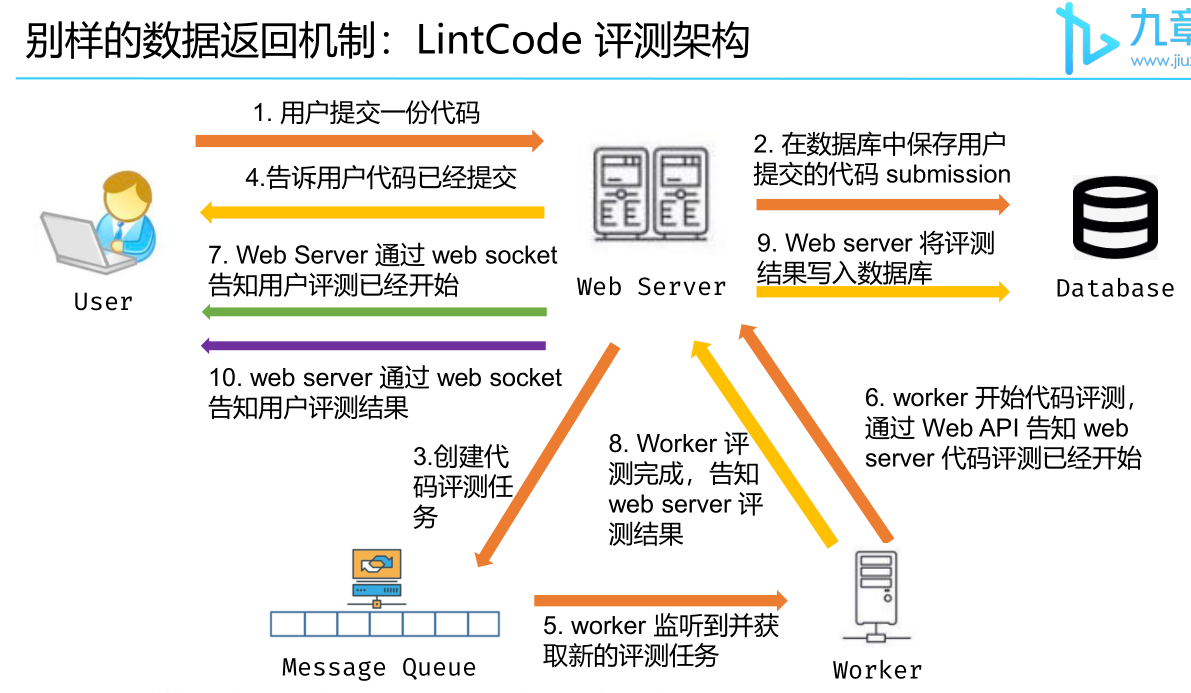


如果是一个大V发帖子，粉丝特别多，可以考虑采用pull模式。



LintCode中Web server和worker上的运行的代码就是不一样的，web server是lintcode的题目系统，包含各自各样的题目，以及提交功能，而worker则是根据测试的数据对用户的代码进行评测。

第3步，创建代码评测任务时，和前面newsfeeds fanout放入队列中的参数不太一样，它需要把各个id以及代码本身都放入message queue中，因为worker无法访问数据库，原因是出于安全的考虑，worker上跑的是用户的代码，而用户的代码是不可控的，有可能会威胁到数据库。



第6步，第8步的操作，由于worker无权访问数据库，所以通过callback(每种类型的任务对应一个接口，worker通过这个接口来通知web server，这个接口是rpc的，这个接口的地址也放在评测任务中)来通知web server。

lintcode评测有submit和runtest，这两个功能都执行异步任务，但对应的在message queue中的队列是不同的。

web server服务器主动向客户端推送消息，通过Web socket来实现。

写一个生产者-消费者的例子，采用redis作为message queue，测试重试和延迟。

测试用RabbitMQ作为message queue。

搞点简单的javascript，发起一个nginx请求，成功怎么处理，失败怎么处理，来看看回调的处理方式。

Celery是用于Web server和Message Queue打交道的工具，就类似python的包mysqlclient用于和mysql打交道。

任务队列：是一种在线程或机器间分发任务的机制。任务队列的输入是一个被称为任务的工作单元。专门的工作进程时刻监视任务队列以执行新的工作。

Celery通过消息通信，通常使用一个broker在客户端和workers之间调度。启动一个任务，客户端将一条消息到队列，broker传递这条消息给一个worker。

Celery系统可以包括多个workers和brokers，提供高可用性和水平扩展。

打开两个vagrant的命令行工具，在其中一个执行python manage.py shell，进入python的命令行，执行：

from twitter.celery import debug\_task

然后执行debug\_task()，发现在命令行中马上显示了打印信息，这样执行debug\_task是在同步执行。

如果希望异步执行，需要在另外一个命令行工具中执行：

celery -A twitter worker -l INFO

在这个命令行中执行一个worker程序，然后在之前python的命令行中执行：

debug\_task.delay()

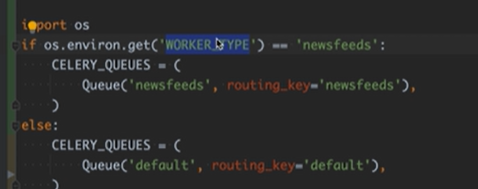
你会发现，它在worker命令行中显示打印信息，这是在异步执行。

newsfeeds fanout异步任务的拆分：如果粉丝非常多，放到一个异步任务中是很不合适的，一旦任务中某一步执行失败了，会导致整个任务失败，代价太大，需要把这个大任务拆分开，比如1000条fanout为一组，这样即使某一次fanout失败了，最多会影响1000个粉丝用户，而不会波及到所有粉丝。

任务的拆分不要放到主程序中，而应该放到aysnc task中，因为它和主程序没什么关系，而只是因为性能，执行时间等原因，才去拆分的，不应该影响到主程序。

fanout\_newsfeeds\_main\_task和fanout\_newfeeds\_batch\_task会投递到两个不同的队列中，这么做的原因是如果都放到一个队列中，如果fanout\_newfeeds\_batch\_task过多，例如粉丝有1亿个，很可能会挤占其它类型任务，比如重置密码发邮件的任务，如果等太长时间显然不合适，所以要投递到不同的队列中。

对于worker而言，如果有多个队列，那么一般情况下从队列中取任务是轮替的方式，即这个队列一个任务，做完后从另一个队列中取一个任务执行。如果希望某类任务能够优先执行，可以把worker进行分配，某一部分worker只做那些比较繁重的任务，另外的worker做高优先级的任务，毕竟高优先级的任务通常比较少。示例代码如下：



尝试进行密码重置。

1个request到达web server，一般就会有10个或几十个queries到数据库，而从数据库到disk，又可能会放大10倍，所以需要ratelimiter，限制用户的高频访问。

python之所以能用来写web server，原因是web server没有什么复杂的计算，主要是请求数据库，组织数据，返回response等，不一定需要C/C++这样的高效语言。

ratelimiter：

针对每个API进行访问次数的限制，如一分钟不超过3次。目的就是限制高频访问去打到数据库DB上。

使用django-ratelimiter。

ratelimiter使用较多的场景是抢票，医院的挂号等。

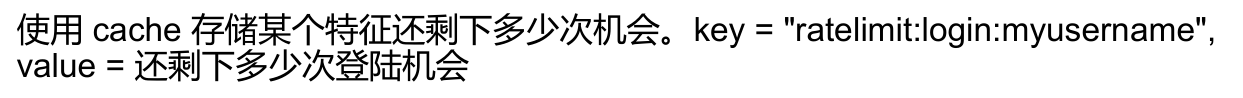
特征选取：使用事件名称 login+用户的username作为特征，不同的特征进行不同的限制。

ratelimiter的存储介质一定是cache，而且是能够全局访问的cache，这样无论哪个web server都能访问到，并且所有web server都应该访问一个统一的值。我们使用ratelimiter的目的就是为了挡住高频访问到数据库上。

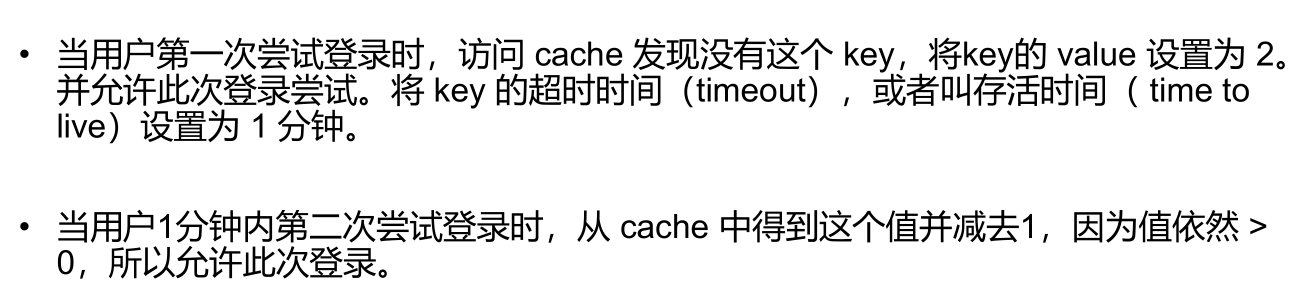
比较hack的一个做法：在web server搞一个小的cache，去过滤掉高频访问。这个两个cache进行同步的操作。

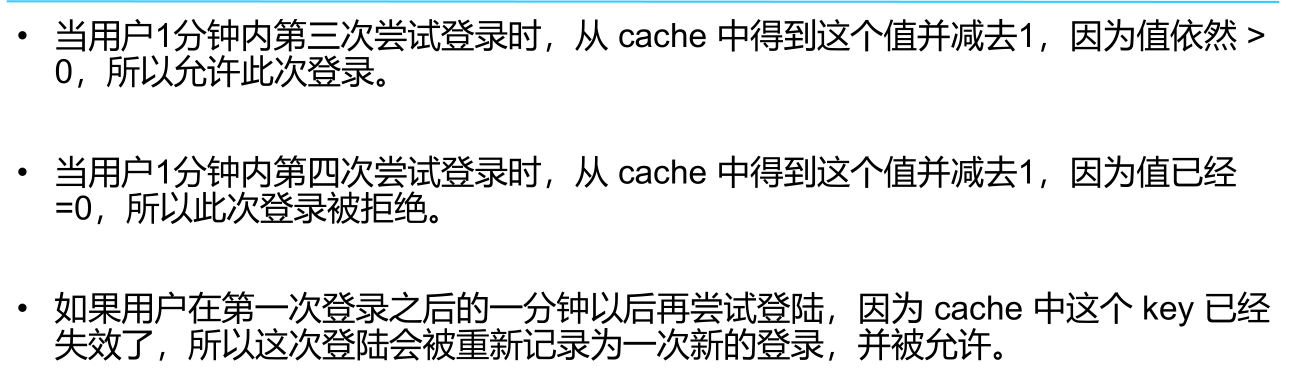
实现ratelimiter有两种方法：

1. 令牌法



具体操作：注意，value是还剩下多少次登陆机会，总共有3次机会，所以value最初存的是2.





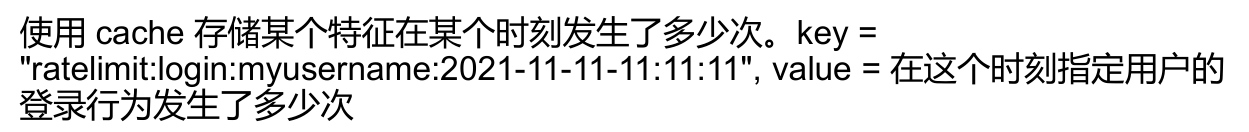
登陆被拒绝，应该返回状态码429.

缺点：

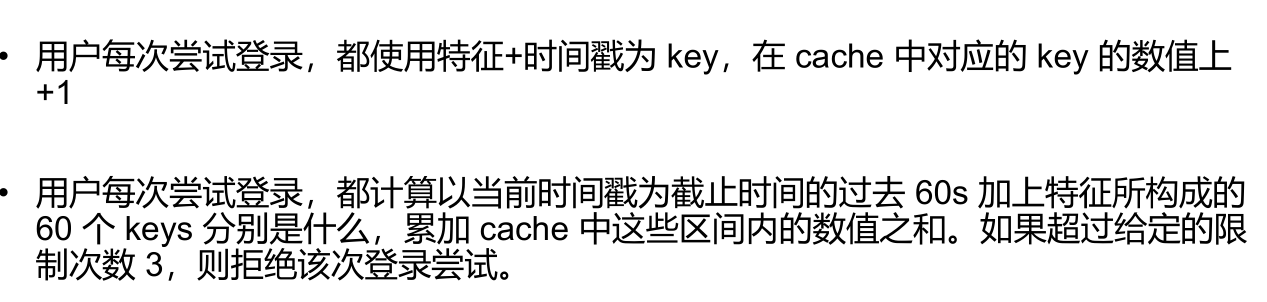
令牌法的计数不是从当前时间往前计数，而是从第1次访问的时间进行计数，这样如果第1次访问开始计数，看成是第0秒访问一次，在之后1分钟时间的末尾连续访问，第57,58分别访问，第59秒访问时就会拒绝，因为已经超过3次了，但第60秒时，又会允许，因为之前设置的key已经失效了，这样会感觉有点不合理，因为第60秒之前的几秒一直在连续访问，应该算高频访问的，按理应该拒绝，但令牌法的逻辑会同意访问。

1. 区间求和法

特征：将时间打散，如果希望精确到每分钟的访问次数，就将时间戳精确到秒。



具体操作：

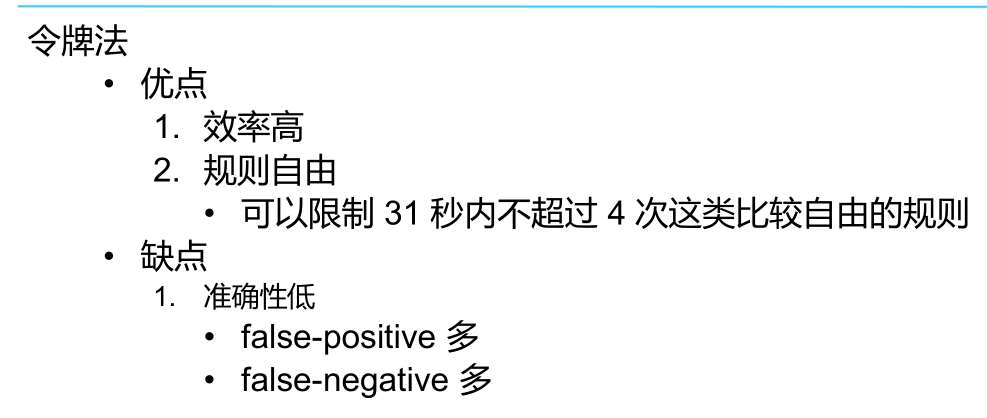


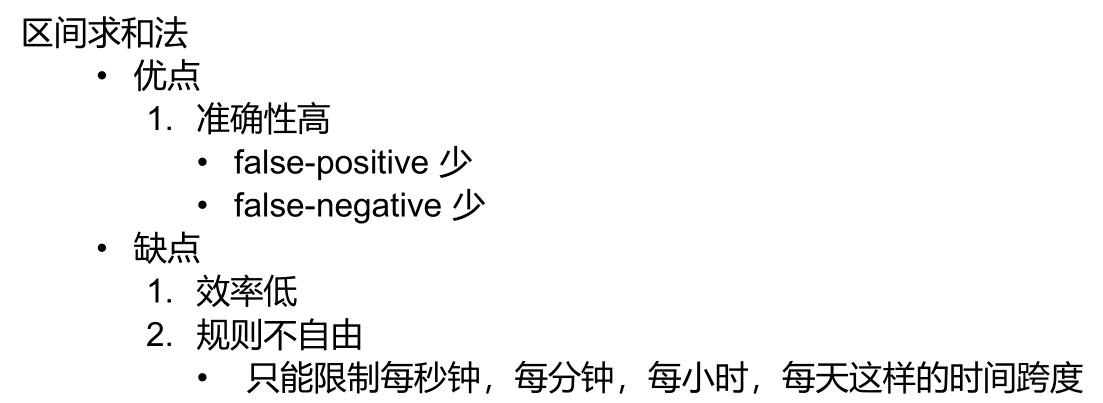
缺点：

效率低，令牌法1分钟可能只有几次cache访问，但区间求和法最坏就会有60次cache访问。

可以读读django-ratelimiter的源码。

区间求和法，如果使用Memcached来做缓存，可以采用get\_many的方法，它的好处是可以一次处理一堆的key，因为web server和cache的服务器通常不在一台机器上，如果使用for循环进行60次读取操作，web server和cache之间的通信一个来回可能需要10ms，60次就是600ms，而get\_many一次把60个key传过去，在cache上取60个key的值，每次的耗时可能就是纳秒为单位的，这样就能大大节约时间。





ratelimit在项目中的代码，有时api还有一个required\_params()的装饰器，这是ratelimit的装饰器要放在下面，让required\_params先执行，因为它访问的是本地的缓存，而ratelimit要访问远端的缓存，毕竟Memcached通常不和web server在同一台机器上，所以可以先让required\_params先过滤一部分request。

http statuscode的部分常见的含义。