使用 Redis 对 Tweets, NewsFeed 进行缓存

授课人 令狐东邪

版权声明

九章的所有课程均受法律保护，不允许录像与传播录像
一经发现，将被追究法律责任和赔偿经济损失

课程大纲

今天需要做什么

01 增加推特缓存

02 安装 Redis

03 缓存 NewsFeed列表

04 缓存和尺寸限制

05 点赞数和评论数去标准化

05 缓存点赞数和评论数

00 前情回顾

前情回顾

在上周的课程中，我们对项目进行了如下**优化**：

- 增加翻页 (Pagination) 功能
- 使用缓存 (Cache) 优化数据库查询

接下来我们要学习**新的内容**:

增加 Cache 来加速 NewsFeed 的访问效率

- 什么是缓存
- Redis vs Memcached
  - Redis 作为 Cache 的优势和劣势
- 数据库与缓存应该如何配合?
- 什么是 Cache Aside 什么是 Cache Through

01 安装 Redis 并将其作为缓存

安装 Redis 并将其作为缓存

在之前的内容中，我们使用了 Memcached 作为缓存，接下来我们还会使用 Redis 作为缓存。在使用之前，让我们先来对比一下两者有什么区别吧。

Memcached 是什么

什么是内存缓存?

免费和开源、高性能、分布式内存对象缓存系统，本质上是通用的，但旨在通过减轻数据库负载来加速动态 Web 应用程序。

Memcached 是一种内存键值存储，用于存储来自数据库调用、API 调用或页面渲染结果的任意数据（字符串、对象）的小块。

Memcached 简单而强大。 其简单的设计促进了快速部署、易于开发，并解决了大数据缓存面临的许多问题。 它的 API 可用于大多数流行语言。

- 内存型存储
- 键值（KV）对
- 不支持持久化

Redis 是什么

Redis 是一种开源（BSD 许可）、内存中数据结构存储，用作数据库、缓存和消息代理。

Redis 提供了诸如字符串、散列、列表、集合、带范围查询的排序集合、位图、超级日志、地理空间索引和流等数据结构。 Redis 内置复制、Lua 脚本、LRU 驱逐、事务和不同级别的磁盘持久化，并通过 Redis Sentinel 和 Redis Cluster 自动分区提供高可用性。

- 内存型存储
- 键值（KV）对
- 支持持久化
- 多种数据类型支持
  String、Hash、List、Set 和 Sorted Set
- 功能丰富
  数据库、缓存、消息队列

Redis 对比Memcached

相同点:

- 数据存储在内存中
- 键值（KV）储存
- 可以作为缓存使用

Redis 对比Memcached

不同点：

- Redis 是一个可做缓存的数据库，还可以作为消息队列、数据堆栈，Memcached 是一个纯粹的缓存
- Redis 支持服务器端的数据操作，在 Memcached 里需要将数据拿到客户端来进行类似的修改再set回去
- Redis 支持 String、Hash、List、Set 和 Sorted Set 五种数据类型，Memcached 只操作字符型数据
- Redis 支持持久化（RDB快照和AOF日志），Memcached不支持数据持久化操作
- Redis 只使用单核，而Memcached可以使用多核
- Memcached 性能略高于 Redis

安装 Redis

安装 Redis

```
sudo apt-get install redis
```

安装 Python 连接 Redis 的依赖

```
pip install redis
```

安装 Redis 并将其作为缓存

配置 Redis

```
196  + REDIS_HOST = '127.0.0.1'
197  + REDIS_PORT = 6379
198  + REDIS_DB = 0 if TESTING else 1
199  + REDIS_KEY_EXPIRE_TIME = 7 * 86400  # in seconds
```

twitter/settings.py

安装 Redis 并将其作为缓存

JSON 编码器

utils/json_encoder.py

```
+ from django.core.serializers.json import DjangoJSONEncoder
+ from django.utils.duration import duration_iso_string
+ from django.utils.functional import Promise
+ from django.utils.timezone import is_aware
+
+ import datetime
+ import decimal
+ import uuid
+
+
+ class JSONEncoder(DjangoJSONEncoder):
+     """
+     JSONEncoder subclass that knows how to encode date/time, decimal types, and
+     UUIDs.
+     """
+     def default(self, o):
+         # See "Date Time String Format" in the ECMA-262 specification.
+         if isinstance(o, datetime.datetime):
+             r = o.isoformat()
+             # 唯一修改的地方，保留 mirco second 增加时间精度
+             # if o.microsecond:
+             #     r = r[:23] + r[26:]
+             if r.endswith('+00:00'):
+                 r = r[:-6] + 'Z'
+             return r
+         elif isinstance(o, datetime.date):
+             return o.isoformat()
+         elif isinstance(o, datetime.time):
+             if is_aware(o):
+                 raise ValueError("JSON can't represent timezone-aware times.")
+             r = o.isoformat()
+             if o.microsecond:
+                 r = r[:12]
+             return r
+         elif isinstance(o, datetime.timedelta):
+             return duration_iso_string(o)
+         elif isinstance(o, (decimal.Decimal, uuid.UUID, Promise)):
+             return str(o)
+         else:
+             return super().default(o)
```

安装 Redis 并将其作为缓存

Redis Client

utils/redis_client.py

```
+ class RedisClient:
+     conn = None
+
+     @classmethod
+     def get_connection(cls):
+         # 使用 singleton 模式，全局只创建一个 connection
+         if cls.conn:
+             return cls.conn
+         cls.conn = redis.Redis(
+             host=settings.REDIS_HOST,
+             port=settings.REDIS_PORT,
+             db=settings.REDIS_DB,
+         )
+         return cls.conn
+
+     @classmethod
+     def clear(cls):
+         # clear all keys in redis, for testing purpose
+         if not settings.TESTING:
+             raise Exception("You can not flush redis in production environment")
+         conn = cls.get_connection()
+         conn.flushdb()
```

安装 Redis 并将其作为缓存

Redis Serializer

utils/redis_serializers.py

```
+ class DjangoModelSerializer:
+
+     @classmethod
+     def serialize(cls, instance):
+         # Django 的 serializers 默认需要一个 QuerySet 或者 list 类型的数据来进行序列化
+         # 因此需要给 instance 加一个 [] 变成 list
+         return serializers.serialize('json', [instance], cls=JSONEncoder)
+
+     @classmethod
+     def deserialize(cls, serialized_data):
+         # 需要加 .object 来得到原始的 model 类型的 object 数据，要不然得到的数据并不是一个
+         # ORM 的 object，而是一个 DeserializedObject 的类型
+         return list(serializers.deserialize('json', serialized_data))[0].object
```

02 将 Redis 作为推文列表的缓存

将 Redis 作为推文列表的缓存

Redis Helper

utils/redis_helper.py

```
+ class RedisHelper:
+
+     @classmethod
+     def _load_objects_to_cache(cls, key, objects):
+         conn = RedisClient.get_connection()
+
+         serialized_list = []
+         for obj in objects:
+             serialized_data = DjangoModelSerializer.serialize(obj)
+             serialized_list.append(serialized_data)
+
+         if serialized_list:
+             conn.rpush(key, *serialized_list)
+             conn.expire(key, settings.REDIS_KEY_EXPIRE_TIME)
+
+     @classmethod
+     def load_objects(cls, key, queryset):
+         conn = RedisClient.get_connection()
+
+         # 如果在 cache 里存在，则直接拿出来，然后返回
+         if conn.exists(key):
+             serialized_list = conn.lrange(key, 0, -1)
+             objects = []
+             for serialized_data in serialized_list:
+                 deserialized_obj = DjangoModelSerializer.deserialize(serialized_data)
+                 objects.append(deserialized_obj)
+             return objects
+
+         cls._load_objects_to_cache(key, queryset)
+
+         # 转换为 list 的原因是保持返回类型的统一，因为存在 redis 里的数据是 list 的形式
+         return list(queryset)
+
+     @classmethod
+     def push_object(cls, key, obj, queryset):
+         conn = RedisClient.get_connection()
+         if not conn.exists(key):
+             # 如果 key 不存在，直接从数据库里 load
+             # 就不走单个 push 的方式加到 cache 里了
+             cls._load_objects_to_cache(key, queryset)
+             return
+         serialized_data = DjangoModelSerializer.serialize(obj)
+         conn.lpush(key, serialized_data)
```

将 Redis 作为推文列表的缓存

Paginations

```
16  +     def paginate_ordered_list(self, reverse_ordered_list, request):
17  +         if 'created_at__gt' in request.query_params:
18  +             created_at__gt = parser.isoparse(request.query_params['created_at__gt'])
19  +             objects = []
20  +             for obj in reverse_ordered_list:
21  +                 if obj.created_at > created_at__gt:
22  +                     objects.append(obj)
23  +                 else:
24  +                     break
25  +             self.has_next_page = False
26  +             return objects
27  +
28  +         index = 0
29  +         if 'created_at__lt' in request.query_params:
30  +             created_at__lt = parser.isoparse(request.query_params['created_at__lt'])
31  +             for index, obj in enumerate(reverse_ordered_list):
32  +                 if obj.created_at < created_at__lt:
33  +                     break
34  +             else:
35  +                 # 没找到任何满足条件的 objects, 返回空数组
36  +                 # 注意这个 else 对应的是 for，参见 python 的 for else 语法
37  +                 reverse_ordered_list = []
38  +         self.has_next_page = len(reverse_ordered_list) > index + self.page_size
39  +         return reverse_ordered_list[index: index + self.page_size]
40  +
41      def paginate_queryset(self, queryset, request, view=None):
42  +         if type(queryset) == list:
43  +             return self.paginate_ordered_list(queryset, request)
```

utils/paginations.py

将 Redis 作为推文列表的缓存

Redis 中用于存储用户推文的 Key

```
+ # memcached
    FOLLOWINGS_PATTERN = 'followings:{user_id}'
    USER_PROFILE_PATTERN = 'userprofile:{user_id}'
+
+ # redis
+ USER_TWEETS_PATTERN = 'user_tweets:{user_id}'
```

twitter/cache.py

将 Redis 作为推文列表的缓存

推文缓存服务

```
7    class TweetService(object):
     @@ -15,3 +18,15 @@ def create_photos_from_files(cls, tweet, files):
18                   )
19                   photos.append(photo)
20               TweetPhoto.objects.bulk_create(photos)
21   +
22   +       @classmethod
23   +       def get_cached_tweets(cls, user_id):
24   +           queryset = Tweet.objects.filter(user_id=user_id).order_by('-created_at')
25   +           key = USER_TWEETS_PATTERN.format(user_id=user_id)
26   +           return RedisHelper.load_objects(key, queryset)
27   +
28   +       @classmethod
29   +       def push_tweet_to_cache(cls, tweet):
30   +           queryset = Tweet.objects.filter(user_id=tweet.user_id).order_by('-created_at')
31   +           key = USER_TWEETS_PATTERN.format(user_id=tweet.user_id)
32   +           RedisHelper.push_object(key, tweet, queryset)
```

tweets/services.py

将 Redis 作为推文列表的缓存

推文模型修改

```
82  82      post_save.connect(invalidate_object_cache, sender=Tweet)
    83  +   post_save.connect(push_tweet_to_cache, sender=Tweet)
```

tweets/models.py

将 Redis 作为推文列表的缓存

推文视图修改

```
45  -       tweets = Tweet.objects.filter(
46  -           user_id=request.query_params['user_id']
47  -       ).order_by('-created_at')
    46  +   tweets = TweetService.get_cached_tweets(user_id=request.query_params['user_id'])
```

tweets/api/views.py

将 Redis 作为推文列表的缓存

Listener

```
+ def push_tweet_to_cache(sender, instance, created, **kwargs):
+     if not created:
+         return
+
+     from tweets.services import TweetService
+     TweetService.push_tweet_to_cache(instance)
```

tweets/listeners.py

03 缓存NewsFeed列表

安装 Redis 并将其作为缓存

Redis Client

twitter/cache.py

```
@@ -4,3 +4,4 @@
4  4
5  5    # redis
6  6    USER_TWEETS_PATTERN = 'user_tweets:{user_id}'
   7  + USER_NEWSFEEDS_PATTERN = 'user_newsfeeds:{user_id}'
```

安装 Redis 并将其作为缓存

Redis Client

newsfeeds/listeners.py

```
... ...  @@ -0,0 +1,6 @@
1        + def push_newsfeed_to_cache(sender, instance, created, **kwargs):
2        +     if not created:
3        +         return
4        +
5        +     from newsfeeds.services import NewsFeedService
6        +     NewsFeedService.push_newsfeed_to_cache(instance)
```

安装 Redis 并将其作为缓存

Redis Client

newsfeeds/models.py

```
...  ...  @@ -1,5 +1,7 @@
1    1      from django.contrib.auth.models import User
2    2      from django.db import models
     3    + from django.db.models.signals import post_save
     4    + from newsfeeds.listeners import push_newsfeed_to_cache
3    5      from tweets.models import Tweet
4    6      from utils.memcached_helper import MemcachedHelper
5    7
     @@ -21,3 +23,6 @@ def __str__(self):
21   23         @property
22   24         def cached_tweet(self):
23   25             return MemcachedHelper.get_object_through_cache(Tweet, self.tweet_id)
     26   +
     27   +
     28   + post_save.connect(push_newsfeed_to_cache, sender=NewsFeed)
```

安装 Redis 并将其作为缓存

九章算法
www.jiuzhang.com

newsfeeds/services.py

```
- from newsfeeds.models import NewsFeed
  from friendships.services import FriendshipService
+ from newsfeeds.models import NewsFeed
+ from newsfeeds.models import NewsFeed
+ from twitter.cache import USER_NEWSFEEDS_PATTERN
+ from utils.redis_helper import RedisHelper


  class NewsFeedService(object):
@@ -21,3 +24,19 @@ def fanout_to_followers(cls, tweet):
            ]
            newsfeeds.append(NewsFeed(user=tweet.user, tweet=tweet))
            NewsFeed.objects.bulk_create(newsfeeds)
+
+           # bulk create 不会触发 post_save 的 signal，所以需要手动 push 到 cache 里
+           for newsfeed in newsfeeds:
+               cls.push_newsfeed_to_cache(newsfeed)
+
+       @classmethod
+       def get_cached_newsfeeds(cls, user_id):
+           queryset = NewsFeed.objects.filter(user_id=user_id).order_by('-created_at')
+           key = USER_NEWSFEEDS_PATTERN.format(user_id=user_id)
+           return RedisHelper.load_objects(key, queryset)
+
+       @classmethod
+       def push_newsfeed_to_cache(cls, newsfeed):
+           queryset = NewsFeed.objects.filter(user_id=newsfeed.user_id).order_by('-created_at')
+           key = USER_NEWSFEEDS_PATTERN.format(user_id=newsfeed.user_id)
+           RedisHelper.push_object(key, newsfeed, queryset)
```

安装 Redis 并将其作为缓存

单元测试

newsfeeds/tests.py

```
@@ -0,0 +1,49 @@
+ from newsfeeds.services import NewsFeedService
+ from testing.testcases import TestCase
+ from twitter.cache import USER_NEWSFEEDS_PATTERN
+ from utils.redis_client import RedisClient
+
+
+ class NewsFeedServiceTests(TestCase):
+
+     def setUp(self):
+         self.clear_cache()
+         self.linghu = self.create_user('linghu')
+         self.dongxie = self.create_user('dongxie')
```

安装 Redis 并将其作为缓存

newsfeeds/tests.py

```
+    def test_get_user_newsfeeds(self):
+        newsfeed_ids = []
+        for i in range(3):
+            tweet = self.create_tweet(self.dongxie)
+            newsfeed = self.create_newsfeed(self.linghu, tweet)
+            newsfeed_ids.append(newsfeed.id)
+        newsfeed_ids = newsfeed_ids[::-1]
+
+        # cache miss
+        newsfeeds = NewsFeedService.get_cached_newsfeeds(self.linghu.id)
+        self.assertEqual([f.id for f in newsfeeds], newsfeed_ids)
+
+        # cache hit
+        newsfeeds = NewsFeedService.get_cached_newsfeeds(self.linghu.id)
+        self.assertEqual([f.id for f in newsfeeds], newsfeed_ids)
+
+        # cache updated
+        tweet = self.create_tweet(self.linghu)
+        new_newsfeed = self.create_newsfeed(self.linghu, tweet)
+        newsfeeds = NewsFeedService.get_cached_newsfeeds(self.linghu.id)
+        newsfeed_ids.insert(0, new_newsfeed.id)
+        self.assertEqual([f.id for f in newsfeeds], newsfeed_ids)
+
```

安装 Redis 并将其作为缓存

newsfeeds/tests.py

```
37  +     def test_create_new_newsfeed_before_get_cached_newsfeeds(self):
38  +         feed1 = self.create_newsfeed(self.linghu, self.create_tweet(self.linghu))
39  +
40  +         RedisClient.clear()
41  +         conn = RedisClient.get_connection()
42  +
43  +         key = USER_NEWSFEEDS_PATTERN.format(user_id=self.linghu.id)
44  +         self.assertEqual(conn.exists(key), False)
45  +         feed2 = self.create_newsfeed(self.linghu, self.create_tweet(self.linghu))
46  +         self.assertEqual(conn.exists(key), True)
47  +
48  +         feeds = NewsFeedService.get_cached_newsfeeds(self.linghu.id)
49  +         self.assertEqual([f.id for f in feeds], [feed2.id, feed1.id])
```

04 缓存和尺寸限制

twitter/settings.py

```
197  197   REDIS_PORT = 6379
198  198   REDIS_DB = 0 if TESTING else 1
199  199   REDIS_KEY_EXPIRE_TIME = 7 * 86400  # in seconds
200       -
     200   + REDIS_LIST_LENGTH_LIMIT = 1000 if not TESTING else 20
201  201
202  202   try:
203  203       from .local_settings import *
```

utils/paginations.py

```
1  1    from dateutil import parser
   2  + from django.conf import settings
2  3    from rest_framework.pagination import BasePagination
3  4    from rest_framework.response import Response
4  5
@@ -39,9 +40,6 @@ def paginate_ordered_list(self, reverse_ordered_list, request):
39 40           return reverse_ordered_list[index: index + self.page_size]
40 41
41 42       def paginate_queryset(self, queryset, request, view=None):
42    -          if type(queryset) == list:
43    -              return self.paginate_ordered_list(queryset, request)
44    -
45 43           if 'created_at__gt' in request.query_params:
46 44               # created_at__gt 用于下拉刷新的时候加载最新的内容进来
47 45               # 为了简便起见，下拉刷新不做翻页机制，直接加载所有更新的数据
@@ -65,6 +63,20 @@ def paginate_queryset(self, queryset, request, view=None):
```

utils/paginations.py

```
+    def paginate_cached_list(self, cached_list, request):
+        paginated_list = self.paginate_ordered_list(cached_list, request)
+        # 如果是上翻页，paginated_list 里是所有的最新的数据，直接返回
+        if 'created_at__gt' in request.query_params:
+            return paginated_list
+        # 如果还有下一页，说明 cached_list 里的数据还没有取完，也直接返回
+        if self.has_next_page:
+            return paginated_list
+        # 如果 cached_list 的长度不足最大限制，说明 cached_list 里已经是所有数据了
+        if len(cached_list) < settings.REDIS_LIST_LENGTH_LIMIT:
+            return paginated_list
+        # 如果进入这里，说明可能存在在数据库里没有 load 在 cache 里的数据，需要直接去数据库查询
+        return None
+
    def get_paginated_response(self, data):
        return Response({
            'has_next_page': self.has_next_page,
```

utils/redis_helper.py

```
@@ -10,7 +10,10 @@ def _load_objects_to_cache(cls, key, objects):
10 10          conn = RedisClient.get_connection()
11 11
12 12          serialized_list = []
13    -         for obj in objects:
   13 +         # 最多只 cache REDIS_LIST_LENGTH_LIMIT 那么多个 objects
   14 +         # 超过这个限制的 objects，就去数据库里读取。一般这个限制会比较大，比如 1000
   15 +         # 因此翻页翻到 1000 的用户访问量会比较少，从数据库读取也不是大问题
   16 +         for obj in objects[:settings.REDIS_LIST_LENGTH_LIMIT]:
14 17              serialized_data = DjangoModelSerializer.serialize(obj)
15 18              serialized_list.append(serialized_data)
16 19
@@ -46,3 +49,4 @@ def push_object(cls, key, obj, queryset):
46 49              return
47 50          serialized_data = DjangoModelSerializer.serialize(obj)
48 51          conn.lpush(key, serialized_data)
   52 +         conn.ltrim(key, 0, settings.REDIS_LIST_LENGTH_LIMIT - 1)
```

tweets/api/views.py

```
@@ -37,16 +37,20 @@ def retrieve(self, request, *args, **kwargs):
37 37
38 38         @required_params(params=['user_id'])
39 39         def list(self, request, *args, **kwargs):
40    -         # 这句查询会被翻译为
41    -         # select * from twitter_tweets
42    -         # where user_id = xxx
43    -         # order by created_at desc
44    -         # 这句 SQL 查询会用到 user 和 created_at 的联合索引
45    -         # 单纯的 user 索引是不够的
46    -         tweets = TweetService.get_cached_tweets(user_id=request.query_params['user_id'])
47    -         tweets = self.paginate_queryset(tweets)
   40 +         user_id = request.query_params['user_id']
   41 +         cached_tweets = TweetService.get_cached_tweets(user_id)
   42 +         page = self.paginator.paginate_cached_list(cached_tweets, request)
   43 +         if page is None:
   44 +             # 这句查询会被翻译为
   45 +             # select * from twitter_tweets
   46 +             # where user_id = xxx
   47 +             # order by created_at desc
   48 +             # 这句 SQL 查询会用到 user 和 created_at 的联合索引
   49 +             # 单纯的 user 索引是不够的
   50 +             queryset = Tweet.objects.filter(user_id=user_id).order_by('-created_at')
   51 +             page = self.paginate_queryset(queryset)
48 52             serializer = TweetSerializer(
49    -             tweets,
   53 +             page,
50 54                 context={'request': request},
51 55                 many=True,
52 56             )
```

newsfeeds/api/views.py

```
@@ -1,17 +1,20 @@
 from newsfeeds.api.serializers import NewsFeedSerializer
+from newsfeeds.models import NewsFeed
 from newsfeeds.services import NewsFeedService
 from rest_framework import viewsets
 from rest_framework.permissions import IsAuthenticated
 from utils.paginations import EndlessPagination

-
 class NewsFeedViewSet(viewsets.GenericViewSet):
     permission_classes = [IsAuthenticated]
     pagination_class = EndlessPagination

     def list(self, request):
-        newsfeeds = NewsFeedService.get_cached_newsfeeds(request.user.id)
-        page = self.paginate_queryset(newsfeeds)
+        cached_newsfeeds = NewsFeedService.get_cached_newsfeeds(request.user.id)
+        page = self.paginator.paginate_cached_list(cached_newsfeeds, request)
+        if page is None:
+            queryset = NewsFeed.objects.filter(user=request.user)
+            page = self.paginate_queryset(queryset)
         serializer = NewsFeedSerializer(
             page,
             context={'request': request},
```

05 点赞数和评论数去标准化

likes/models.py

```
1       - from accounts.services import UserService
2  1      from django.contrib.auth.models import User
3  2      from django.contrib.contenttypes.fields import GenericForeignKey
4  3      from django.contrib.contenttypes.models import ContentType
5  4      from django.db import models
   5    + from django.db.models.signals import pre_delete, post_save
   6    + from likes.listeners import incr_likes_count, decr_likes_count
6  7      from utils.memcached_helper import MemcachedHelper
7  8
8  9
   ↓↑   @@ -39,3 +40,7 @@ def __str__(self):
39 40         @property
40 41         def cached_user(self):
41 42             return MemcachedHelper.get_object_through_cache(User, self.user_id)
   43   +
   44   +
   45   + pre_delete.connect(decr_likes_count, sender=Like)
   46   + post_save.connect(incr_likes_count, sender=Like)
```

comments/models.py

```
...  ...  @@ -1,7 +1,8 @@
1         - from accounts.services import UserService
     1    + from comments.listeners import incr_comments_count, decr_comments_count
2    2      from django.contrib.auth.models import User
3    3      from django.contrib.contenttypes.models import ContentType
4    4      from django.db import models
     5    + from django.db.models.signals import post_save, pre_delete
5    6      from likes.models import Like
6    7      from tweets.models import Tweet
7    8      from utils.memcached_helper import MemcachedHelper
     ↓↑   @@ -40,3 +41,7 @@ def like_set(self):
40   41         @property
41   42         def cached_user(self):
42   43             return MemcachedHelper.get_object_through_cache(User, self.user_id)
     44   +
     45   +
     46   + post_save.connect(incr_comments_count, sender=Comment)
     47   + pre_delete.connect(decr_comments_count, sender=Comment)
```

tweets/models.py

```
@@ -15,6 +15,11 @@ class Tweet(models.Model):
15  15      content = models.CharField(max_length=255)
16  16      created_at = models.DateTimeField(auto_now_add=True)
17  17
    18  +   # 新增的 field 一定要设置 null=True，否则 default = 0 会遍历整个表单去设置
    19  +   # 导致 Migration 过程非常慢，从而把整张表单锁死，从而正常用户无法创建新的 tweets
    20  +   likes_count = models.IntegerField(default=0, null=True)
    21  +   comments_count = models.IntegerField(default=0, null=True)
    22  +
18  23      class Meta:
19  24          index_together = (('user', 'created_at'),)
20  25          ordering = ('user', '-created_at')
```

```python
+ def incr_likes_count(sender, instance, created, **kwargs):
+     from tweets.models import Tweet
+     from django.db.models import F
+
+     if not created:
+         return
+
+     model_class = instance.content_type.model_class()
+     if model_class != Tweet:
+         # TODO HOMEWORK 给 Comment 使用类似的方法进行 likes_count 的统计
+         return
+
+     # 不可以使用 tweet.likes_count += 1; tweet.save() 的方式
+     # 因此这个操作不是原子操作，必须使用 update 语句才是原子操作
+     tweet = instance.content_object
+     Tweet.objects.filter(id=tweet.id).update(likes_count=F('likes_count') + 1)
+
```

```python
+ def decr_likes_count(sender, instance, **kwargs):
+     from tweets.models import Tweet
+     from django.db.models import F
+
+     model_class = instance.content_type.model_class()
+     if model_class != Tweet:
+         # TODO HOMEWORK 给 Comment 使用类似的方法进行 likes_count 的统计
+         return
+
+     # handle tweet likes cancel
+     tweet = instance.content_object
+     Tweet.objects.filter(id=tweet.id).update(likes_count=F('likes_count') - 1)
```

likes/listeners.py

comments/listeners.py

```
+ from utils.listeners import invalidate_object_cache
+
+
+ def incr_comments_count(sender, instance, created, **kwargs):
+     from tweets.models import Tweet
+     from django.db.models import F
+
+     if not created:
+         return
+
+     # handle new comment
+     Tweet.objects.filter(id=instance.tweet_id)\
+         .update(comments_count=F('comments_count') + 1)
+     invalidate_object_cache(sender=Tweet, instance=instance.tweet)
+
+
+ def decr_comments_count(sender, instance, **kwargs):
+     from tweets.models import Tweet
+     from django.db.models import F
+
+     # handle comment deletion
+     Tweet.objects.filter(id=instance.tweet_id)\
+         .update(comments_count=F('comments_count') - 1)
+     invalidate_object_cache(sender=Tweet, instance=instance.tweet)
```

06 缓存点赞数和评论数

utils/redis_helper.py

```
+
+    @classmethod
+    def get_count_key(cls, obj, attr):
+        return '{}.{}:{}'.format(obj.__class__.__name__, attr, obj.id)
+
+    @classmethod
+    def incr_count(cls, obj, attr):
+        conn = RedisClient.get_connection()
+        key = cls.get_count_key(obj, attr)
+        return conn.incr(key)
+
+    @classmethod
+    def decr_count(cls, obj, attr):
+        conn = RedisClient.get_connection()
+        key = cls.get_count_key(obj, attr)
+        return conn.decr(key)
+
+    @classmethod
+    def get_count(cls, obj, attr):
+        conn = RedisClient.get_connection()
+        key = cls.get_count_key(obj, attr)
+        count = conn.get(key)
+        if count is not None:
+            return int(count)
+
+        obj.refresh_from_db()
+        count = getattr(obj, attr)
+        conn.set(key, count)
+        return count
```

tweets/api/serializers.py

```
7  7   from tweets.constants import TWEET_PHOTOS_UPLOAD_LIMIT
8  8   from tweets.models import Tweet
9  9   from tweets.services import TweetService
   10 + from utils.redis_helper import RedisHelper
10 11
11 12
12 13  class TweetSerializer(serializers.ModelSerializer):
@@ -30,10 +31,10 @@ class Meta:
30 31              )
31 32
32 33      def get_likes_count(self, obj):
33    -         return obj.like_set.count()
   34 +         return RedisHelper.get_count(obj, 'likes_count')
34 35
35 36      def get_comments_count(self, obj):
36    -         return obj.comment_set.count()
   37 +         return RedisHelper.get_count(obj, 'comments_count')
37 38
38 39      def get_has_liked(self, obj):
39 40          return LikeService.has_liked(self.context['request'].user, obj)
```

likes/listeners.py

```
...  ...  @@ -1,3 +1,6 @@
     1  + from utils.redis_helper import RedisHelper
     2  +
     3  +
1    4    def incr_likes_count(sender, instance, created, **kwargs):
2    5        from tweets.models import Tweet
3    6        from django.db.models import F
     ↕    @@ -14,6 +17,7 @@ def incr_likes_count(sender, instance, created, **kwargs):
14   17       # 因此这个操作不是原子操作，必须使用 update 语句才是原子操作
15   18       tweet = instance.content_object
16   19       Tweet.objects.filter(id=tweet.id).update(likes_count=F('likes_count') + 1)
     20 +     RedisHelper.incr_count(tweet, 'likes_count')
17   21
18   22
19   23   def decr_likes_count(sender, instance, **kwargs):
     ↕    @@ -28,3 +32,4 @@ def decr_likes_count(sender, instance, **kwargs):
28   32       # handle tweet likes cancel
29   33       tweet = instance.content_object
30   34       Tweet.objects.filter(id=tweet.id).update(likes_count=F('likes_count') - 1)
     35 +     RedisHelper.decr_count(tweet, 'likes_count')
```

comments/listeners.py

```
1  1   from utils.listeners import invalidate_object_cache
   2 + from utils.redis_helper import RedisHelper
2  3
3  4
4  5   def incr_comments_count(sender, instance, created, **kwargs):
@@ -12,6 +13,7 @@ def incr_comments_count(sender, instance, created, **kwargs):
12 13      Tweet.objects.filter(id=instance.tweet_id)\
13 14          .update(comments_count=F('comments_count') + 1)
14 15      invalidate_object_cache(sender=Tweet, instance=instance.tweet)
   16 +    RedisHelper.incr_count(instance.tweet, 'comments_count')
15 17
16 18
17 19  def decr_comments_count(sender, instance, **kwargs):
@@ -22,3 +24,4 @@ def decr_comments_count(sender, instance, **kwargs):
22 24      Tweet.objects.filter(id=instance.tweet_id)\
23 25          .update(comments_count=F('comments_count') - 1)
24 26      invalidate_object_cache(sender=Tweet, instance=instance.tweet)
   27 +    RedisHelper.decr_count(instance.tweet, 'comments_count')
```