开源项目阅读

发布 *v1.0*

2020 年 12 月 12 日

Contents:

开源代码阅读，学习笔记，目前主要是 python

CHAPTER 1

开源项目阅读系列

1.1 现状与不足

可完成公司任务，即使对于小众或冷门的用法也看得懂，凡是想清楚的业务逻辑，都可以用程序表达出来。

但是，总觉得依然不够，只能算及格，60 分。

信手拈来的研发思路依然只是，父子类，接口，工厂模式，单例模式，代理模式等。不够开阔，对于复杂的烂代码阅读能力较差，难以较快理解其代码调用逻辑。

1.2 进身之阶

如何进步？阅读他人代码，个人认为是一个捷径.

计划:request,tablib,gunicore，timeit

10 个不到 500 行代码的超牛 Python 练手项目（技术清单系列）：https://zhuanlan.zhihu.com/p/52881791

CHAPTER 2

开源项目阅读 01cpythonDemo

2.1 beer

地址:https://github.com/python/cpython/blob/master/Tools/demo/beer.py

```
n = 100
if sys.argv[1:]:
    n = int(sys.argv[1])
```

更佳写法:

```
n = int(sys.argv[1]) if sys.argv[1:] else 100) # 推荐, 易懂
n = sys.argv[1:] and int(sys.argv[1]) or 100)
```

2.2 hanoi

地址:https://github.com/python/cpython/blob/master/Tools/demo/hanoi.py

```
    if sys.argv[2:]:
        bitmap = sys.argv[2]
        # Reverse meaning of leading '@' compared to Tk
        if bitmap[0] == '@': bitmap = bitmap[1:]
```

(下页继续)

(续上页)

```
        else: bitmap = '@' + bitmap
    else:
        bitmap = None
```

更佳写法:(一定程度上牺牲了可读性)

```
bitmap=None if not sys.argv[2:] else sys.argv[2][1] if sys.argv[2][0]=='@' else '@' +␣
↪sys.argv[2]
```

2.3 rpythond

地址:https://github.com/python/cpython/blob/master/Tools/demo/rpythond.py

```
def execute(request):
    stdout = sys.stdout
    stderr = sys.stderr
    sys.stdout = sys.stderr = fakefile = io.StringIO() # 链式赋值
    try:
        try:
            exec(request, {}, {})
        except:
            print()
            traceback.print_exc(100) # 代替 print e 来输出详细的异常信息
    finally:
        sys.stderr = stderr # 替换回来
        sys.stdout = stdout
    return fakefile.getvalue()
```

try except finally 如果使用 with 的上下文管理逻辑更好

2.4 vector

地址:https://github.com/python/cpython/blob/master/Tools/demo/vector.py

```


class Vec:
    """A simple vector class.
    Instances of the Vec class can be constructed from numbers
    >>> a = Vec(1, 2, 3)
```

(下页继续)

(续上页)

```
    >>> b = Vec(3, 2, 1)
    added
    >>> a + b
    Vec(4, 4, 4)
    subtracted
    >>> a - b
    Vec(-2, 0, 2)
    and multiplied by a scalar on the left
    >>> 3.0 * a
    Vec(3.0, 6.0, 9.0)
    or on the right
    >>> a * 3.0
    Vec(3.0, 6.0, 9.0)
    """
    def __init__(self, *v):
        self.v = list(v)

    @classmethod
    def fromlist(cls, v):
        if not isinstance(v, list):
            raise TypeError
        inst = cls() # 创建实例？
        inst.v = v
        return inst

    def __repr__(self):
        args = ', '.join(repr(x) for x in self.v)
        return 'Vec({})'.format(args)

    def __len__(self):
        return len(self.v)

    def __getitem__(self, i):
        return self.v[i]

    def __add__(self, other):
        # Element-wise addition
        v = [x + y for x, y in zip(self.v, other.v)]
        return Vec.fromlist(v)
```

(下页继续)

(续上页)

```
    def __sub__(self, other):
        # Element-wise subtraction
        v = [x - y for x, y in zip(self.v, other.v)]# 列表同位置元素减法运算
        return Vec.fromlist(v)

    def __mul__(self, scalar):
        # Multiply by scalar
        v = [x * scalar for x in self.v]
        return Vec.fromlist(v)

    __rmul__ = __mul__ # 函数赋值，(int* 自己) 和 (自己 *int) 同方法


def test():
    import doctest
    doctest.testmod() # 文档测试方法

test()
```

2.5 要点

```
sys.stdout = sys.stderr = fakefile = io.StringIO() # 链式赋值
traceback.print_exc(100) # 代替 print e 来输出详细的异常信息
inst = cls() # 创建实例？
v = [x - y for x, y in zip(self.v, other.v)]# 列表同位置元素减法运算
__rmul__ = __mul__ # 函数赋值，(int* 自己) 和 (自己 *int) 同方法
doctest.testmod() # 文档测试方法
```

CHAPTER 3

开源项目阅读 02requests

最大收获在于代码可读性切分，以及变量命名的合理。

需求驱动的模块划分：一般是有公用方法才提出独立 func，没有的话就大段代码堆积，除非非常长的代码，影响阅读效果，才会考虑切分。

可读性驱动的模块划分：request 更多偏向于“注释型切分”，按照功能角色进行切分，哪怕只有几行代码，如果是独立小 block，也会抽取出独立函数，通过函数名标识代码块功能，所以代码即使不看注释也很容易读懂 (当然，request 模块本身代码注释也很完善)。

3.1 基础

3.1.1 功能

request 可以看做**基于 urllib3 的二次封装**，使得其更易用，所以本身逻辑性代码并不多，很多代码是异常处理或者兼容性处理和注释等。

urllib、urllib2、urllib3 的关系

```
urllib 和 urllib2 是独立的模块，并没有直接的关系，两者相互结合实现复杂的功能
urllib 和 urllib2 在 python2 中才可以使用
requests 库中使用了 urllib3(多次请求重复使用一个 socket)
```

支持功能

打开 README.md 文件：

```
Feature Support
---------------
Requests is ready for today's web.

-   International Domains and URLs # 国际化域名和 URLS
-   Keep-Alive & Connection Pooling #keep—Alive& 连接池
-   Sessions with Cookie Persistence # 持久性 cookie 的会话
-   Browser-style SSL Verification # 浏览器式 SSL 认证
-   Basic/Digest Authentication # 基本/摘要认证
-   Elegant Key/Value Cookies # 简明的 key/value cookies
-   Automatic Decompression # 自动解压缩
-   Automatic Content Decoding # 自动内容解码
-   Unicode Response Bodies #Unicode 响应体
-   Multipart File Uploads # 文件分块上传
-   HTTP(S) Proxy Support #HTTP(S) 代理支持
-   Connection Timeouts # 连接超时
-   Streaming Downloads # 数据流下载
-   `.netrc` Support #'.netrc' 支持
-   Chunked Requests #Chunked 请求
```

3.1.2 代码量

Language	files	blank	comment	code
Python	34	1936	1990	5852
Markdown	10	536	0	1508

可见 comment 和代码比例为 1936:5852=1:3 左右, 开源代码大多数注释比较完备。

3.1.3 模块

网络请求参数类，包括 Request 和 PrepareRequest

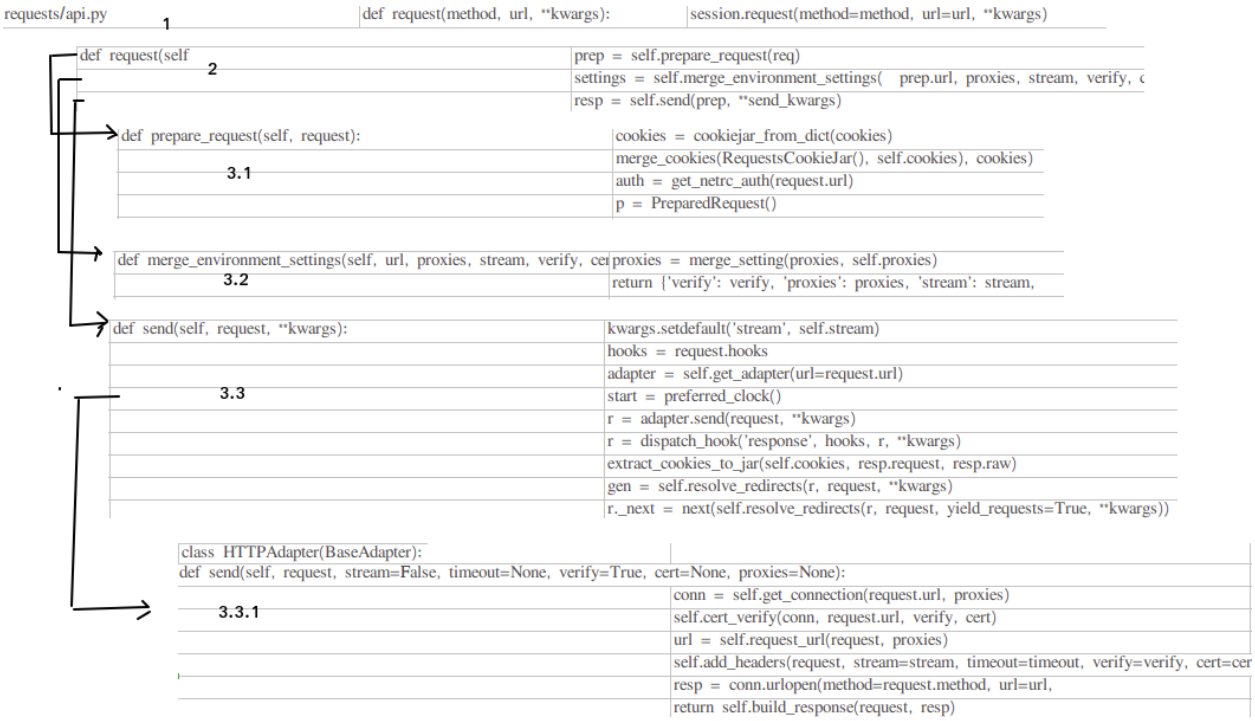
网络请求返回类，Response

会话管理类，Session

实际网络请求处理类，包括 BaseAdapter 和 HTTPAdapter

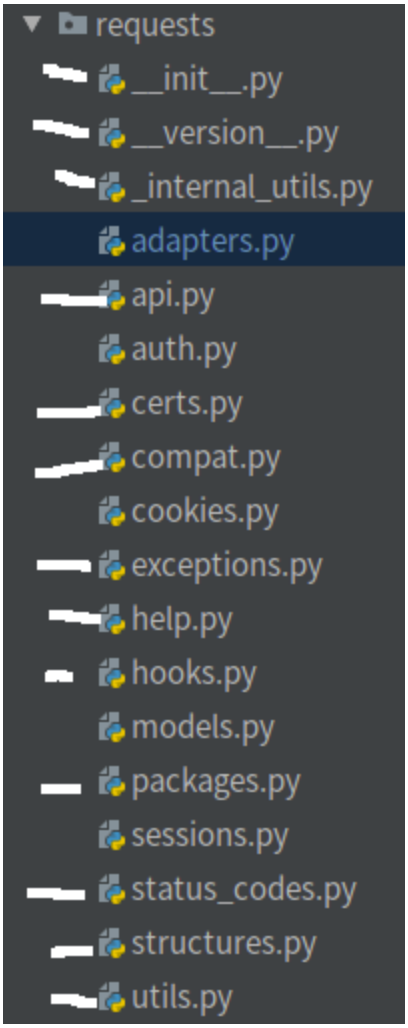
以及 Cookie 管理，Auth 身份验证。

3.2 深度阅读，调用链



随便找一个 requests.get() 阅读，依次跟进各函数，就可以得到核心函数表，借此表，可得知各模块在 request 请求里的调用次序和功能角色。

3.3 广度阅读，模块简析



边缘模块简析:

api: 接口的 get,post 封装, 本质调用了:session.request(method=method, url=url, **kwargs)

compat: 对 py2 和 py3 中不同的数据格式做了兼容处理，比较好

exceptions: 一些异常

help: 写的比较少，也不打算研究

hooks: 只有一个干货方法

```
def dispatch_hook(key, hooks, hook_data, **kwargs):
    """Dispatches a hook dictionary on a given piece of data."""
    hooks = hooks or {}
    hooks = hooks.get(key)
```

(下页继续)

(续上页)

```
    if hooks:
        if hasattr(hooks, '__call__'):
            hooks = [hooks]
        for hook in hooks:
            _hook_data = hook(hook_data, **kwargs)
            if _hook_data is not None:
                hook_data = _hook_data
    return hook_data
```

钩子函数，依次调用 list() 里各个函数，如果有返回值，把返回值单做下一阶段的入参

packages:

status_codes: 状态码映射

structures:2 个自定义数据结构 CaseInsensitiveDict,LookupDict

utils: 工具函数

需要重点学习的有:

adapters:

auth:

cookies:

models:

sessions:

3.4 模块解释 (简单到复杂)

3.4.1 httpbin

httpbin: A simple HTTP Request & Response Service.

简单来说：告诉你 (resp) 请求他时你带的参数，自测神器

关于 httpbin 细节，可阅读参考文献，learn_python/read_requests_v2.22.0/read_requests_v2.22.0.md。

3.4.2 hooks(钩子)

简单来说就是一个接一个的函数调用，有点像 django 的中间件，层层调用。

代码:

```
def dispatch_hook(key, hooks, hook_data, **kwargs):
    """Dispatches a hook dictionary on a given piece of data."""
    hooks = hooks or {}
    hooks = hooks.get(key)
    if hooks:
        if hasattr(hooks, '__call__'):
            hooks = [hooks]
        for hook in hooks:
            _hook_data = hook(hook_data, **kwargs)
            if _hook_data is not None:
                hook_data = _hook_data
    return hook_data
```

说回之前的流程，实例化属性 self.hooks 是 {'response' : [ ] } 这样的数据结构，类属性 hooks 是一个字典 (可能为空，可能不为空)。然后通过对类属性 hooks 的主键和值进行迭代，并将键值对作为参数传入继承而来的 self.register_hook 方法。

self.register_hook 方法首先判断主键是否存在于实例化属性 self.hooks 之中，如果不存在则抛出错误，很明显，类属性 hooks 这个字典的主键值也只允许是'respones'，这也很好理解，因为我们回调函数也只有一个 respones 对象可以处理而已。

self.register_hook 方法其次判断 value 值是否是可调用对象。

如果可调用就直接将其添加到实例化属性 self.hooks 数据 {'response' : [ ] } 的值的列表中。

如果不可调用，则判断该 value 是否可被迭代。

如果可以被迭代再判断每个迭代的元素是否可以被调用，只将可以被调用的元素添加到实例化属性 self.hooks 数据 {'response' : [ ] } 的值的列表中。

自此，实例属性 self.hooks 创建完成。

3.4.3 structures

CaseInsensitiveDict

其中 MutableMapping 为抽象基类，类似 Mapping。它们在 collections 模块中，供我们实现自定义的 map 类。Mapping 包含 dict 中的所有不变方法，MutableMapping 扩展包含了所有可变方法，但它们两个都不包含那五大核心特殊方法：getitem、setitem、delitem、len、iter。也就是说我们的目标就是实现这五大核心方法使该数据结构能够使用。

核心代码如下

```
class CaseInsensitiveDict(MutableMapping):
    def __init__(self, data=None, **kwargs):
```

(下页继续)

(续上页)

```
        self._store = OrderedDict()# 有序 key 的 map
        if data is None:
            data = {}
        self.update(data, **kwargs)

    def __setitem__(self, key, value):
        # Use the lowercased key for lookups, but store the actual
        # key alongside the value.
        self._store[key.lower()] = (key, value)# 转小写

    def __getitem__(self, key):
        return self._store[key.lower()][1]#? 为何 [1]

    def __iter__(self):
        return (casedkey for casedkey, mappedvalue in self._store.values())# 生成器

    def lower_items(self):
        """Like iteritems(), but with all lowercase keys."""
        return (
            (lowerkey, keyval[1]) #? 为何只取了 [1]，既是第一个也应该是 [0]，为何在 eq 中需
要使用
            for (lowerkey, keyval)
            in self._store.items()
        )

    def __eq__(self, other):
        if isinstance(other, Mapping):
            other = CaseInsensitiveDict(other) # 兼容普通 map
        else:
            return NotImplemented
        # Compare insensitively
        return dict(self.lower_items()) == dict(other.lower_items())   # 间接调用普通 dict
的__eq__

    # Copy is required
    def copy(self):
        return CaseInsensitiveDict(self._store.values())
```

LookupDict

本身只是个有名字 (name) 的 map

```
class LookupDict(dict):
    """Dictionary lookup object."""

    def __init__(self, name=None):
        self.name = name
        super(LookupDict, self).__init__()
```

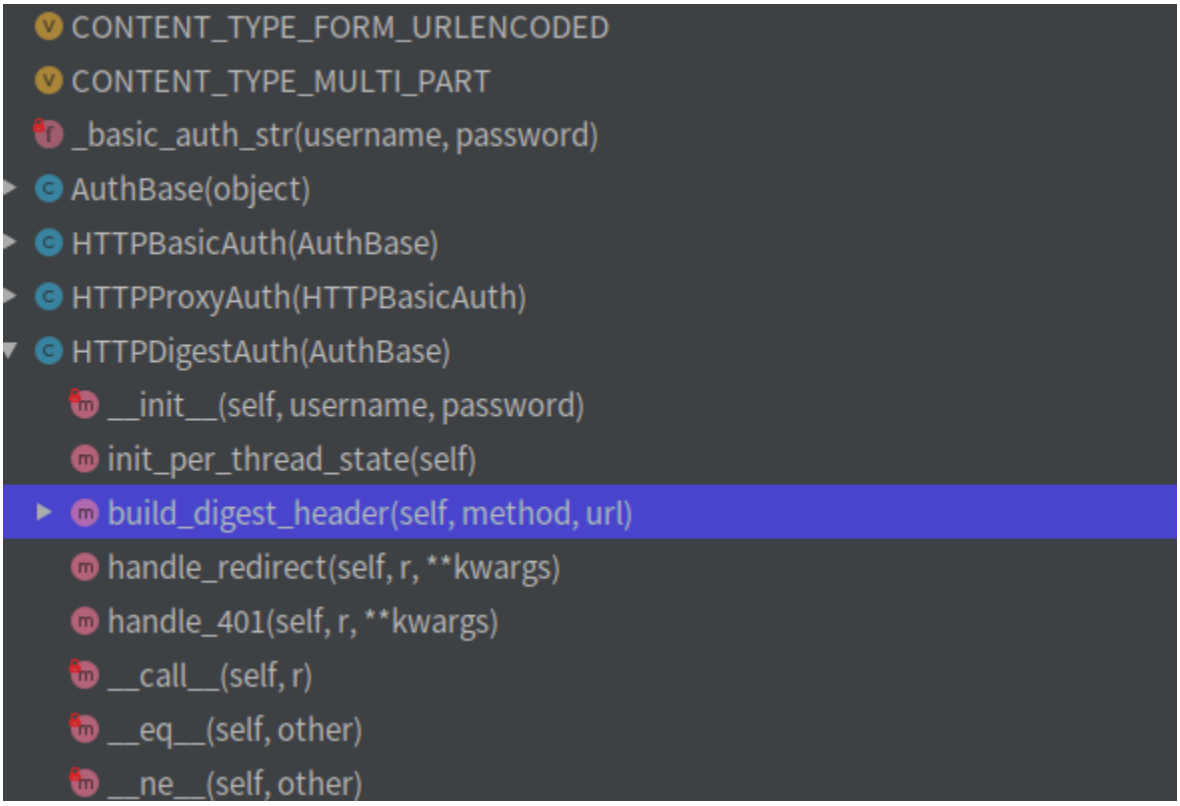
特殊的在于其用法

```
_codes = {

    # Informational.
    100: ('continue',),
    101: ('switching_protocols',),
    102: ('processing',),
    103: ('checkpoint',),
    122: ('uri_too_long', 'request_uri_too_long'),
    200: ('ok', 'okay', 'all_ok', 'all_okay', 'all_good', '\\o/', '✓'),
    }
=>键值对互换
codes.continue=100
codes.uri_too_long=122
codes.request_uri_too_long=122
```

键值对互换且多个键对应一个取值

3.4.4 auth

代码结构：



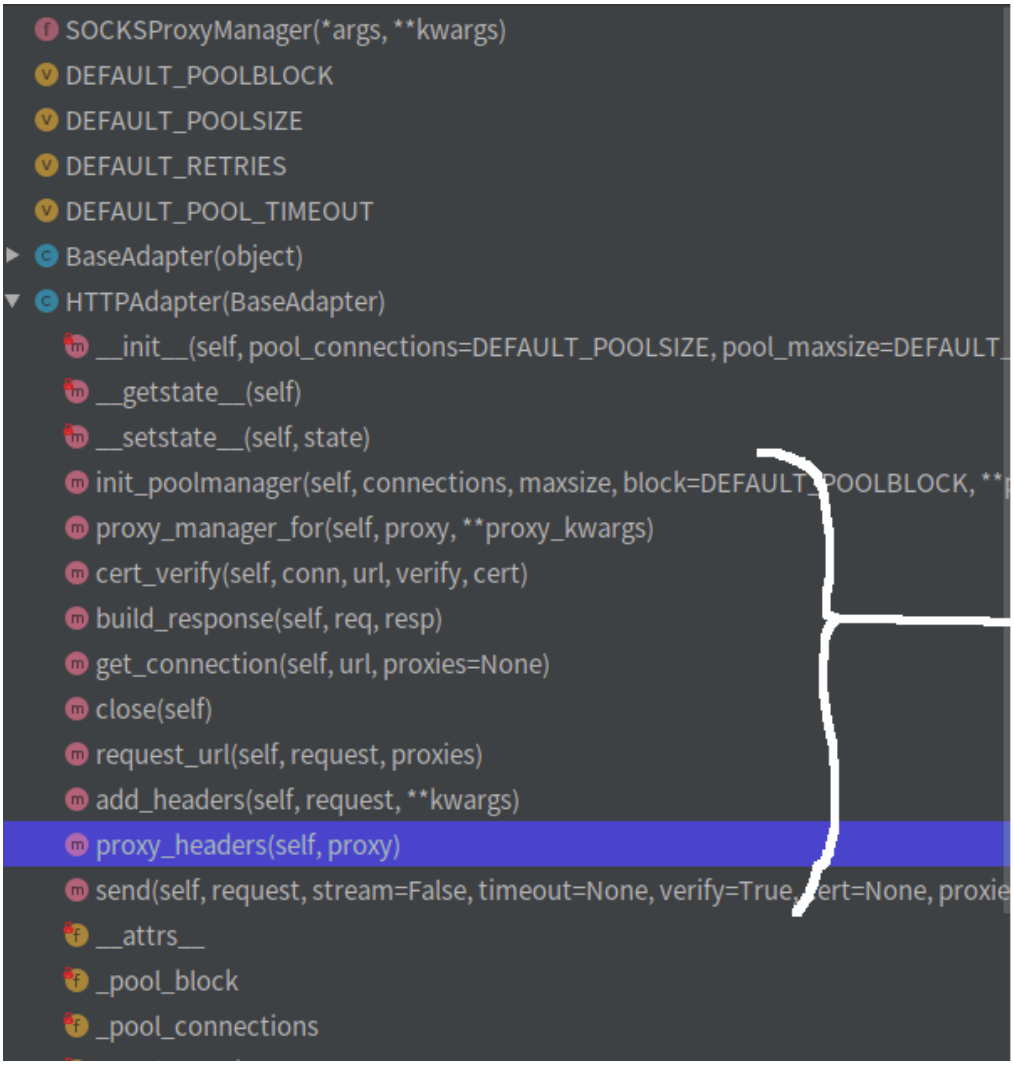
其中核心方法就是图中标出的 build_digest_header()(摘要认证)

auth 代码本身就是实现了 http 的基本认证和摘要认证 (可以看做一种约定或协议)，所以阅读代码前先了解下 http 的基本认证和摘要认证。

```
基本认证
Base64(user:pwd) 后，放在 Http 头的 Authorization 中发送给服务端来作认证.
用 Base64 纯只是防君子不防小人的做法。所以只适合用在一些不那么要求安全性的场合。

摘要认证
digest authentication (HTTP1.1 提出的基本认证的替代方法)
这个认证可以看做是基本认证的增强版本，不包含密码的明文传递。
```

详情自行百度。明白了基本认证和摘要认证，这段代码基本也就看明白了，就是实现了一些**约定的规范**。

3.4.5 adapters



实际的网络请求类包括 BaseAdapter 和 HTTPAdapter 。其中 BaseAdapter 只定义了基础的接口，而 HTTPAdapter 是对 BaseAdapter 的实现。

从类名，可看出这应该是适配器，结合前文中提过，requests 本身是基于 urllib3 开发的，**而 adapters 就是对 urllib3 进行的二次封装**。

从函数名称也能大概看出，这里才是**真正执行上层 (api,session) 操作的模块**，扮演“执行者”或者“基层公务员”的角色。

关于适配器设计模式，参考博文:08_ 适配器

核心代码为 send(), 简化后代码

```
def send(self, request, stream=False, timeout=None, verify=True, cert=None,␣
↪proxies=None):
```

(下页继续)

(续上页)

```
    conn = self.get_connection(request.url, proxies)

    self.cert_verify(conn, request.url, verify, cert)
    url = self.request_url(request, proxies)
    self.add_headers(request, stream=stream, timeout=timeout, verify=verify, cert=cert,␣
↪proxies=proxies)

        resp = conn.urlopen(
                method=request.method,
                url=url,
                body=request.body,
                headers=request.headers,
                redirect=False,
                assert_same_host=False,
                preload_content=False,
                decode_content=False,
                retries=self.max_retries,
                timeout=timeout
        )
    return self.build_response(request, resp)
```

略过连接建立，超时机制，异常处理的部分，只看实际发送请求的部分：

从 urllib3 维护的 Connection Pool 中获取连接

进行 SSL 验证

url 做转化 (考虑使用代理的情况)

添加 request 头部 putheader

发送 request

接受 response

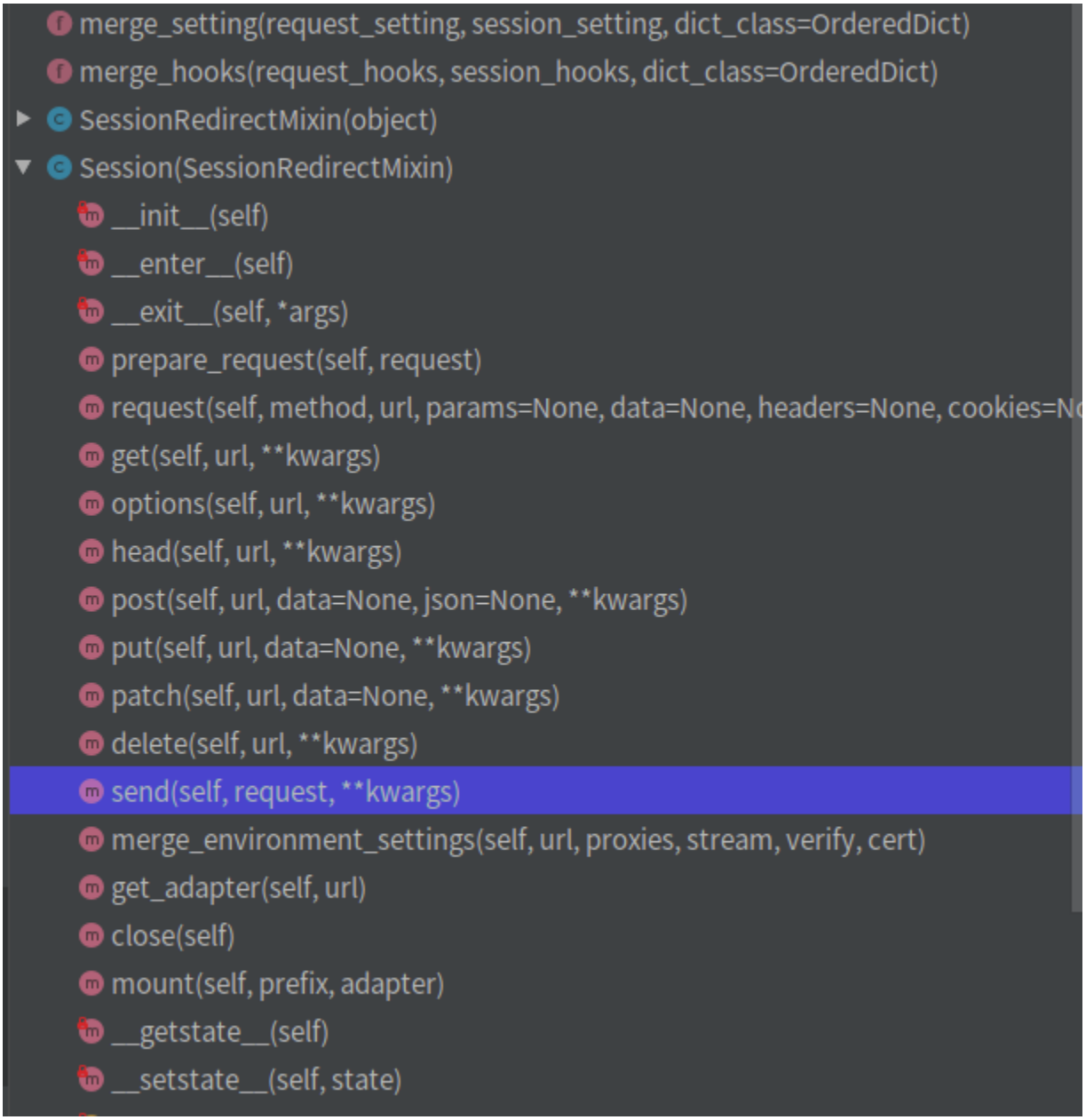
最后通过调用 build_response 来基于 urllib3 response 构建 request.Respnse 对象返回给用户，到此为止一次 requests.get() 动作便结束。

3.4.6 sessions

上层入口 api.py 的调用都会到 sessions 这里，所以**这里才是 requests 里比较核心**的东西。

使用 Session 对象可以让你跨请求保持参数，在同一个 Session 实例中发出的网络请求可以保持 cookie，同一主机的 TCP 请求会被重用，从而带来性能提升。

保留参数信息 和 cookie
利用 urllib3 的连接池
可以为 request 对象提供默认数据



入口方法 request:

```
def request(self, method, url,
    # Create the Request.
    req = Request(
        method=method.upper(),
```

(下页继续)

(续上页)

```
        url=url,
        headers=headers,
        files=files,
        data=data or {},
        json=json,
        params=params or {},
        auth=auth,
        cookies=cookies,
        hooks=hooks,
    )
    prep = self.prepare_request(req)
    settings = self.merge_environment_settings(
        prep.url, proxies, stream, verify, cert
    )

    # Send the request.
    send_kwargs = {
        'timeout': timeout,
        'allow_redirects': allow_redirects,
    }
    send_kwargs.update(settings)
    resp = self.send(prep, **send_kwargs)

    return resp
```

可以拆分为以下四个步骤:

创建 Request 对象 request: 根据用户传入的一系列传输构建的 request，用于准备真正传送出去的 PreparedRequest

创建 prepare request 对象 prep

发送 request send

send 返回值 response，返回给用户

prepare_request: 相比原始 request 做了一定的二次加工, 主要体现在几个 merge_xx 方法上

```
p = PreparedRequest()
        p.prepare(
            method=request.method.upper(),
            url=request.url,
            files=request.files,
```

(下页继续)

(续上页)

```
            data=request.data,
            json=request.json,
            headers=merge_setting(request.headers, self.headers, dict_
↪class=CaseInsensitiveDict),# 改进
            params=merge_setting(request.params, self.params),# 改进
            auth=merge_setting(auth, self.auth),# 改进
            cookies=merged_cookies,# 改进
            hooks=merge_hooks(request.hooks, self.hooks),# 改进
        )
```

在构建网络请求参数时，调用了 merge_cookies() 方法将 Session 中的 cookies 与本次请求的 cookies 合并了，因此使用同一个 Session 对象发起网络请求时才能实现跨请求保持 cookie。至于其他的参数，可以看到调用了 merge_setting() 方法进行了合并。而在网络请求返回时，会将请求的必要信息，比如 cookie 保存在 Session 中。

send: 发送逻辑

```
def send(self, request, **kwargs):
    kwargs.setdefault('stream', self.stream)
    kwargs.setdefault('verify', self.verify)
    kwargs.setdefault('cert', self.cert)
    kwargs.setdefault('proxies', self.proxies)

    hooks = request.hooks
    adapter = self.get_adapter(url=request.url)
    r = adapter.send(request, **kwargs)
    r = dispatch_hook('response', hooks, r, **kwargs)
    extract_cookies_to_jar(self.cookies, request, r.raw)

    return r
```

只是对 hooks 和 cookie 做了部分处理，主要逻辑 send 由 adapter 执行 (adapter 前文解释过，不在赘述).

3.5 架构

3.5.1 短方法 (函数)

```
def get_adapter(self, url):
    """
    Returns the appropriate connection adapter for the given URL.
```

(下页继续)

(续上页)

```
    :rtype: requests.adapters.BaseAdapter
    """
    for (prefix, adapter) in self.adapters.items():

        if url.lower().startswith(prefix.lower()):
            return adapter
```

其中:self.adapters

```
self.adapters = OrderedDict()
self.mount('https://', HTTPAdapter())
self.mount('http://', HTTPAdapter())
```

可以发现，很多短方法，如果不考虑 _Init_ 部分，基本 50 行以上的都非常少，大多数 30 行以下。

而就本人开发经验而言，很多情况下，开发都是长方法，50+ 是比较常见的。

短方法最大好处是节约注释，但劣势就是对**命名和参数提炼**比较讲究，既可以标识出“**代码块功能**”，又能区分出和其他**代码块差异**（拆为 step01(),step02(),step03() 也是代码拆分，但是代码块差异则无法体现）。

3.5.2 阅读 v0.10.0

核心调用链:api=>session(py).request()=>models(py).Request.send()

3.6 编码

3.6.1 长文本折行

```
warnings.warn(
            "Non-string usernames will no longer be supported in Requests "
            "3.0.0. Please convert the object you've passed in ({!r}) to "
            "a string or bytes object in the near future to avoid "
            "problems.".format(username),
            category=DeprecationWarning,
        )
```

3.6.2 all() 函数

```
def __eq__(self, other):
    return all([
        self.username == getattr(other, 'username', None),
        self.password == getattr(other, 'password', None)
    ])
```

以下是 all() 方法的语法:

```
all(iterable)
```

参数:iterable – 元组或列表。

返回值: 如果 iterable 的所有元素不为 0、”、False 或者 iterable 为空，all(iterable) 返回 True，否则返回 False；

注意：空元组、空列表返回值为 True，这里要特别注意。

3.6.3 连续比较

```
if value[:1] == value[-1:] == '"':
```

3.6.4 变量函数

函数当做变量使用，避免 if-else 切入代码过多

```
get_proxy = lambda k: os.environ.get(k) or os.environ.get(k.upper())
if no_proxy is None:
    no_proxy = get_proxy('no_proxy')
```

3.6.5 上下文

先修改环境变量, 使用后再恢复回来。

```
@contextlib.contextmanager
def set_environ(env_name, value):
    """Set the environment variable 'env_name' to 'value'

    Save previous value, yield, and then restore the previous value stored in
    the environment variable 'env_name'.
```

(下页继续)

(续上页)

```
    If 'value' is None, do nothing"""
```

3.7 其他问题

python 的动态类型的确很影响对代码的阅读和理解.

改进：通过类型注解 (Type Annotations)（增加文档 Decumentation String，函数注解）.

3.8 参考

Requests 库请求过程简析：https://blog.csdn.net/weixin_41677555/article/details/85246464

python Requests 源码阅读：https://hustyichi.github.io/2019/08/10/requests-codes/

拆轮子系列：requests：https://www.jianshu.com/p/83ffcbe99bb2

python requests v2.22.0 源码阅读:https://zhuanlan.zhihu.com/p/82694710

从 Request 库理解 HTTP 消息：https://www.cnblogs.com/yc913344706/p/7995225.html

用 Python 实现数据结构之映射：https://www.cnblogs.com/sfencs-hcy/p/10350475.html

python 找到 dict 值对应的键, 读 requests 有感：https://www.jianshu.com/p/55f17668f156

如何阅读源代码（以 Python requests 库为例）:https://www.jianshu.com/p/145959b76ff3

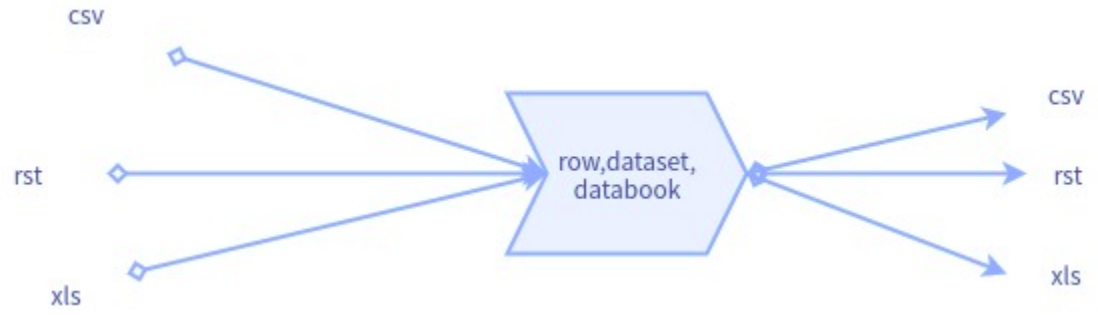
learn_python/read_requests_v2.22.0/read_requests_v2.22.0.md:https://github.com/BigFlower666/learn_python/blob/ma

CHAPTER 4

开源项目阅读 03tablib

代码结构合理, 支持多种数据格式，通过定义” 内部格式”(row,dataset,databook) 实现了多种格式的导入和导出。

功能角度类似 pandas, 也是管理多维表的，相对轻量级一些。这个真的很简单，大概看一下吧



4.1 基础

4.1.1 功能

Tablib 是一个**表格操作库**，使用 python 编写，目前 (2020-07-11) 支持如下格式：cli,csv,dbf,df (DataFrame),html,jira,json,latex,ods,rst,tsv,xls,xlsx,yaml 的导入/导出，及修改操作。实现方法是使用各种数据格式的 python 支持库 (大多是各种格式的有明支持库) 导入数据成 list（列表，python 内置数据结构），**每个 list 的成员就是数据表的一行 (建立一个类 Row)。对数据的操作就转化成对 list 和 Row 的操作。导出时，导出时又使用各库封闭的导出接口**就可以了。

使用样例代码 (from:test 测试案例):

```
def setUp(self):
    """Create simple data set with headers."""

    global data, book

    data = tablib.Dataset()
    book = tablib.Databook()

    self.headers = ('first_name', 'last_name', 'gpa')
    self.john = ('John', 'Adams', 90)
    self.george = ('George', 'Washington', 67)
    self.tom = ('Thomas', 'Jefferson', 50)

    self.founders = tablib.Dataset(headers=self.headers, title='Founders')
    self.founders.append(self.john)
    self.founders.append(self.george)


data.append(self.john)
data.append(self.george)
data.headers = self.headers

_csv = data.csv
```

4.1.2 代码量

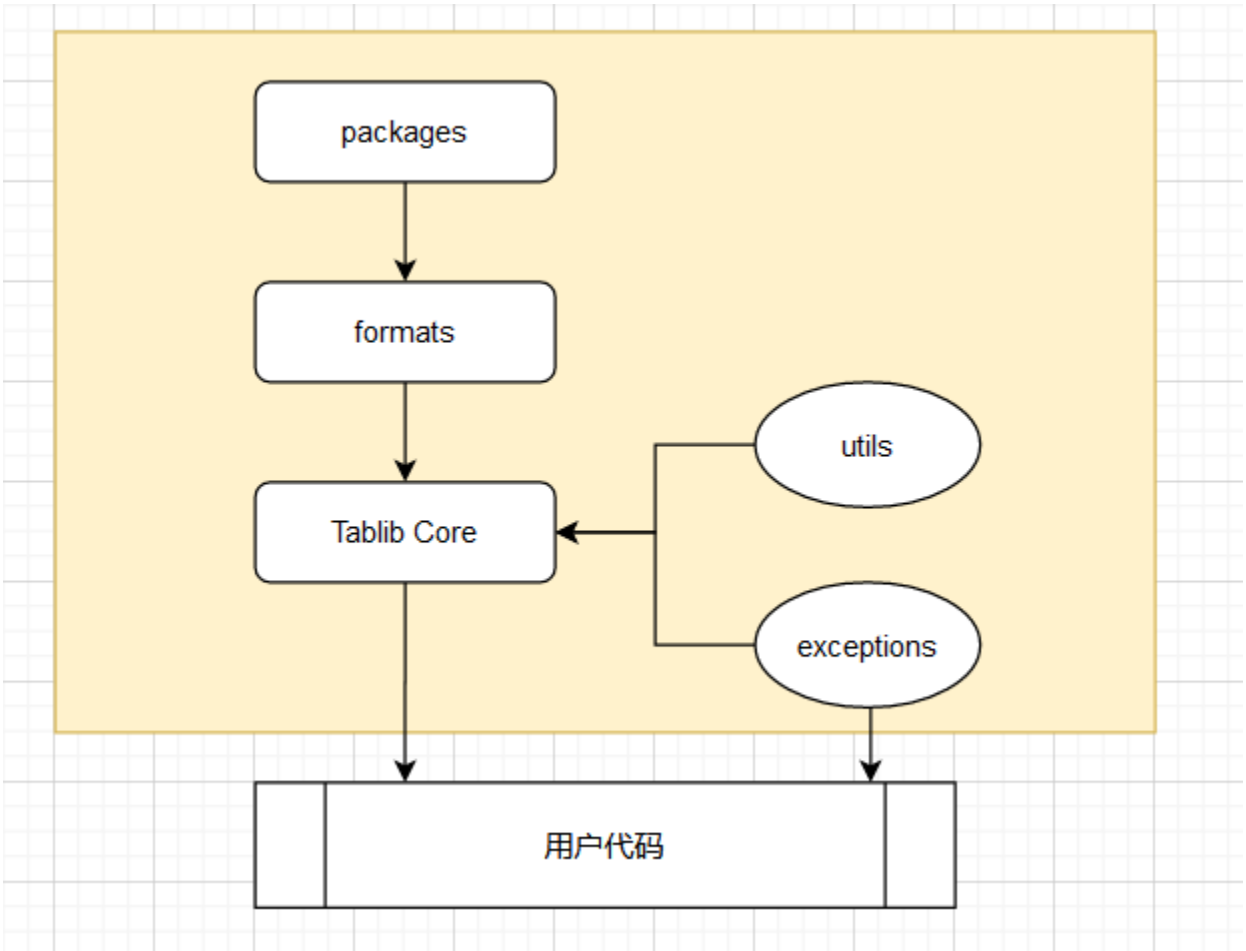
```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          30           1308           1456           3059
XML                              5              0              0            375
Markdown                         4            156              0            260
```

comments 和代码比例大约 2:1，可见开源代码之“友好”。本人认为开源代码由于没有 kpi 压力，可以较好的代表研发人员的架构和编码水平，这样的**大牛的高质量代码都尚且需要如此高比例的注释**。那么国内大多研发人员或公司是不是应该汗颜。

4.1.3 模块


src
tablib
formats
packages
__init__.py
core.py
exceptions.py
utils.py
对各种格式的支持
内部核心类
异常
工具类

模块关系

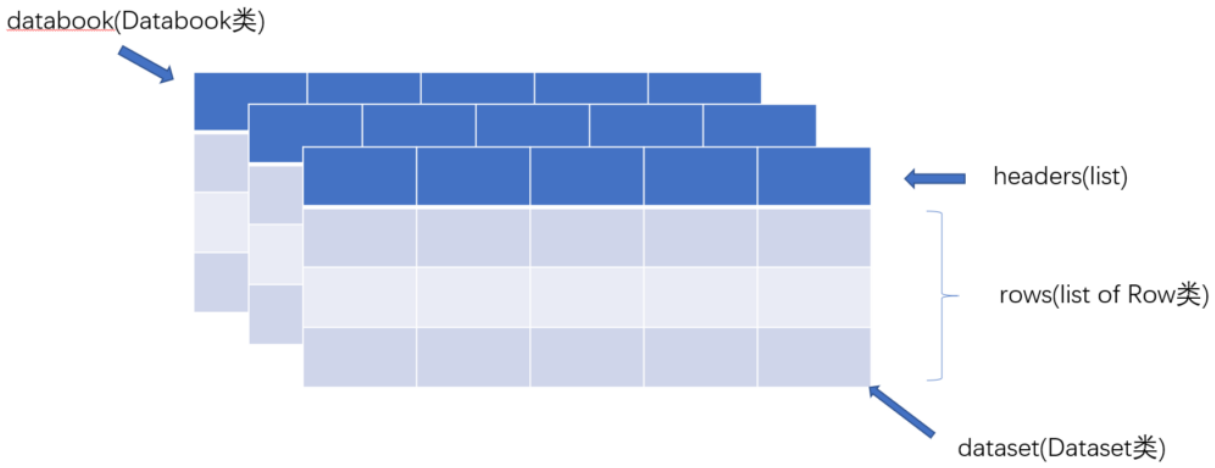

packages
formats
utils
Tablib Core
exceptions
用户代码

4.2 深度阅读，调用链 (略)

4.3 广度阅读，模块简析

core：Row，Dataset 和 Databook 三个类

Tablib 对表格的实现跟 Excel 中的表格概念是一致的，**一个 Databook 可以类比成一个 Excel 文件**，里边可以包含多个 **Dataset（可以类比于 Excel 中的 sheet**，代码中是 self._datasets），而每一个 Dataset 中有可能会有表头 headers（可选）是通过 Python 的 List 存储的（代码中是 self.___headers），每一个 Dataset 中会有很多行的数据（代码中是 self._data），**每一行都是一个 Row 的实例**，而每一行中的元素则在 Row 的成员变量中用一个 List 来存储 (代码中是 self._row)，该 List 中的每一个元素对应于一个单元里边的值。



4.4 模块解释

4.4.1 Row

初始化方法:

```
def __init__(self, row=list(), tags=list()):
    self._row = list(row)
    self.tags = list(tags)
```

通过 self._row 这一个成员变量来存储一行里边的所有元素，通过一些双下划线方法，**借助 Python 的语法糖对 Row 中的单元格数据的存取**。

4.4.2 Dataset

和 row 类似，也是通过实现 python 的双下划线方法的语法糖，支持 python 的常见操作符。

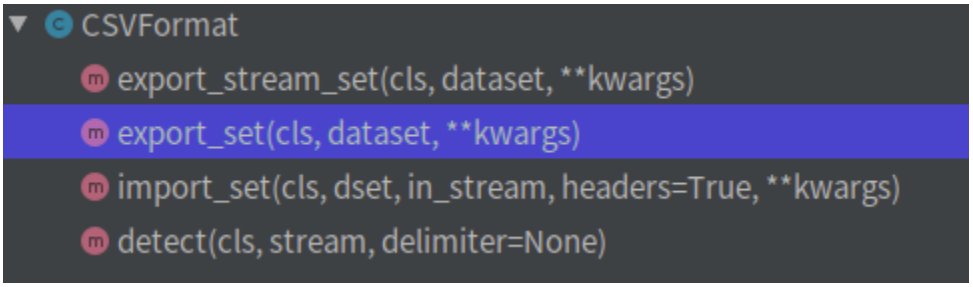
另外为了提供所有的针对 Dataset 的操作，还实现了**表格的格式化，表格数据的验证**（用于当有新数据存进来的时候，确保新数据的格式跟表格是匹配的），以及不同格式的数据的导入和导出（主要依赖于 formats 模块）

初始化方法:

```
def __init__(self, *args, **kwargs):
    self._data = list(Row(arg) for arg in args)
    self.__headers = None

    # ('title', index) tuples
    self._separators = []

    # (column, callback) tuples
    self._formatters = []

    self.headers = kwargs.get('headers')

    self.title = kwargs.get('title')
```

4.4.3 formats

以 csv 为例：实现了 4 个方法，从名字也可以看出功能。不再赘述。



4.4.4 registry

tablib 最有意思的就是这个模块了。整体上实现了按需求安装和动态 import 的能力 (为了截屏能一次截完，对代码做了简化)

4.5 缺点

啥？大牛代码也有缺点？

任何事情有好处必然有坏处，事物自带属性，**享受一方面优惠的同时就要付出另一方成本**，哪怕大牛也无法避免。

这里更多是见仁见智的问题，可能很多人不认同。**代码采用了“动态编码导入”，享受的好处是避免了安装一堆自己不需要的软件包，带来的缺点丢失代码的依赖关系**，导致编译器无法识别（只有运行时才会真实的关联），**所以在 formats 文件夹下的类，通过编译器的 find usages 无法定位到哪里被使用了**，在 debug 时，这样的类导入方式带来难以定位，调试和追踪的问题。

4.6 参考

跟我一起读源码 – Tablib 源码阅读：https://zhuanlan.zhihu.com/p/147786720

tablib 源代码学习：https://www.cnblogs.com/hustlijian/p/3782525.html

CHAPTER 5

开源项目阅读 04records

5.1 基础

只需要输入 sql 语句就可以把结果包成对象返回，极大的方便了用户（这个应该是早期结论，目前大多数开发框架都会集成 ORM 框，一样很方便，所以实用角度看，意义不大，抱着学习的态度看一看即可）

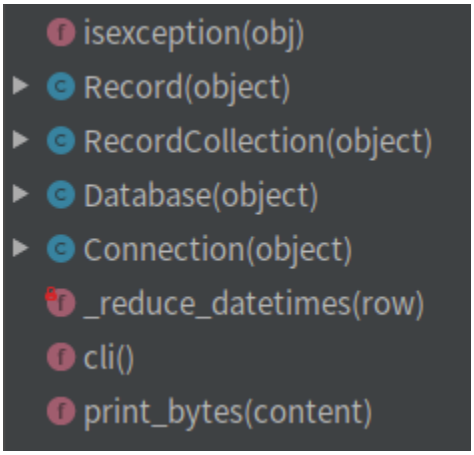
借助项目（基于以下项目封装）：

```
sqlalchemy：python 的 ORM 框架
tablib：kennethreitz 的另一个项目，主要是把数据处理为 XLS，CSV，JSON，YAML 格式返回。
```

5.2 用法

```
    db.query("SELECT * FROM users WHERE id = :user", user="Te'ArnaLambert")
```

5.3 广度阅读



Record 储存每条数据的详情；RecordCollection 储存 query 的查找结果，也就是 Record 的集合；Database 数据库的操作集合。

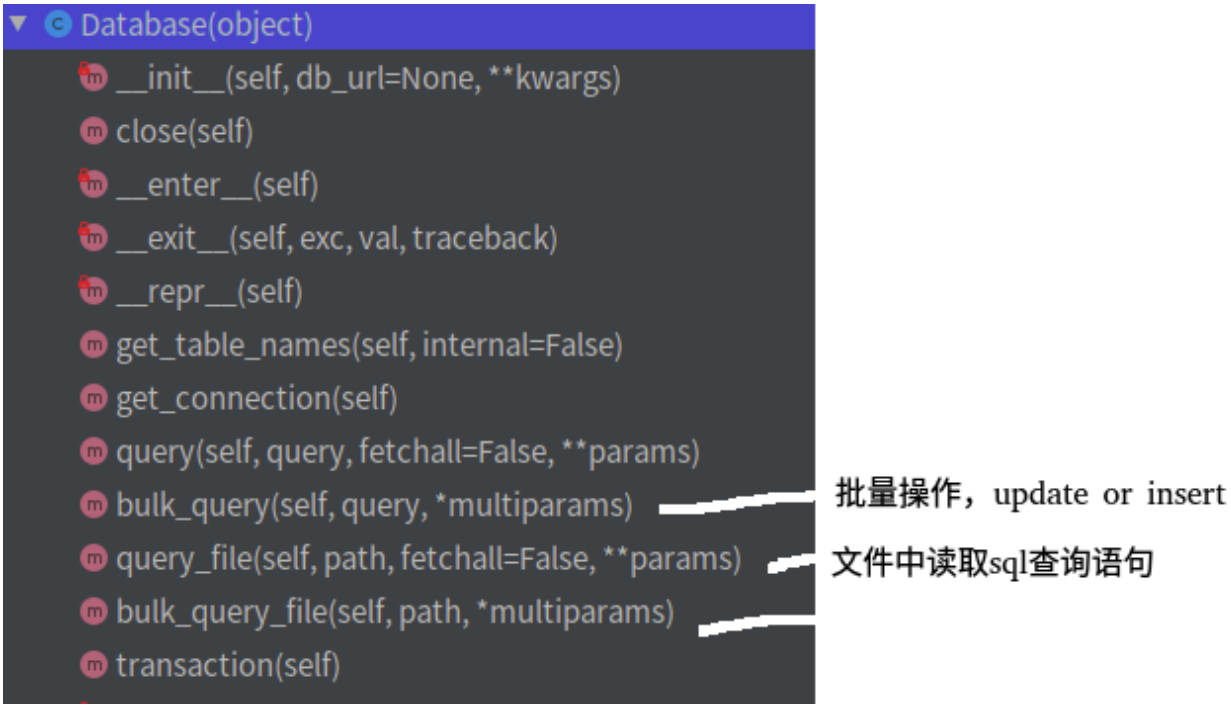
形象的来说：RecordCollection 就是拉皮条的，手里有很多的 Record，来源自 Database，客户就是我们。。。。

```
class Record(object):
    """A row, from a query, from a database."""
    __slots__ = ('_keys', '_values')# 不使用 dict, 可减少内存占用

    def __init__(self, keys, values):
        self._keys = keys
        self._values = values

        # Ensure that lengths match properly.
        assert len(self._keys) == len(self._values)

class RecordCollection(object):
    """A set of excellent Records from a query."""
    def __init__(self, rows):#rows Record 组成的列表
        self._rows = rows
        self._all_rows = []
        self.pending = True

def all(self, as_dict=False, as_ordereddict=False):
    """Returns a list of all rows for the RecordCollection. If they haven't
    been fetched yet, consume the iterator and cache the results."""
```

(下页继续)

(续上页)

```
    # By calling list it calls the __iter__ method
    rows = list(self)# 这个写法有点意思，本质是调用 self.__iter__ 不断产生对象
```



5.4 评价

较好的利用了 python 的魔法方法特性，这也是 python 明显不同 java 等语言的地方，等价于对运算符的重载。

5.5 参考

python 库学习：records 库源码分析：https://zhuanlan.zhihu.com/p/52605543

CHAPTER 6

开源项目阅读 05imutils

6.1 基础

6.1.1 功能

imutils 是一个图像处理工具包，它对 opencv 的一些方法进行了二次加工，使其更加简单易用。相比较于 opencv 的学习难度，导致很多方法使用起来需要一定的基础，新手可能会起步的较慢，而 imutils 使用起来比较便利，能够辅助我们理解 opencv

6.1.2 代码量

Language	files	blank	comment	code
Python	44	421	686	1178
Markdown	1	60	0	131
XML	5	0	0	41

6.2 用法

6.2.1 查询 opencv 中的函数

可以使用关键词搜索 opencv 中的相应函数

```
import imutils
imutils.find_function("area")
#output:
1. CC_STAT_AREA
2. INTER_AREA
3. contourArea
4. minAreaRect
```

6.2.2 图像平移 Translation

图像在 x 轴方向左右平移，y 轴方向上下平移，

```
# 向右平移 25 像素，向上平移 75 像素
translated = imutils.translate(image,25,-75)
```

6.2.3 图像旋转 Rotation

```
rotated = imutils.rotate(image,90)
```

6.2.4 图像大小 Resizing

改变图像大小，但保持原来图像的长宽比不变。

可以只单独设置 width 或者 height；

```
resized = imutils.resize(image,width=300)
resized = imutils.resize(image,height=300)
```

6.2.5 骨架化 Skeletonization

```
gray = cv2.cvtColor(logo, cv2.COLOR_BGR2GRAY)
skeleton = imutils.skeletonize(gray, size=(3, 3))
cv2.imshow("Skeleton", skeleton)
```

6.2.6 URL 转换为 Image

```
image = imutils.url_to_image(url)
自动边缘检测 Automatic Canny Edge Detection
gray = cv2.cvtColor(logo, cv2.COLOR_BGR2GRAY)
edgeMap = imutils.auto_canny(gray)
# 源码
def auto_canny(image, sigma=0.33):
    # compute the median of the single channel pixel intensities
    v = np.median(image)
    # apply automatic Canny edge detection using the computed median
    lower = int(max(0, (1.0 - sigma) * v))
    upper = int(min(255, (1.0 + sigma) * v))
    edged = cv2.Canny(image, lower, upper)
    # return the edged image
    return edged
```

6.3 深度阅读 (略)

6.4 广度阅读

6.4.1 dense

```
class DENSE:
    def detect(self, img):
        kps = []
        for x in range(0, img.shape[1], self.step):
            for y in range(0, img.shape[0], self.step):
                # create a keypoint and add it to the keypoints list
                kps.append(cv2.KeyPoint(x, y, self.radius))
```

cv2.KeyPoint 是 opencv 中关键点检测函数 detectAndCompute() 返回的关键点的类，他包含关键点的位置，方向等属性

6.4.2 gftt

```
class GFTT:
    def detect(self, img):
        cnrs = cv2.goodFeaturesToTrack(img, self.maxCorners, self.qualityLevel, self.
↪minDistance,
                                       mask=self.mask, blockSize=self.blockSize,
                                       useHarrisDetector=self.useHarrisDetector, k=self.
↪k)

        return corners_to_keypoints(cnrs)


def corners_to_keypoints(corners):
    """function to take the corners from cv2.GoodFeaturesToTrack and return cv2.KeyPoints
↪"""
    if corners is None:
        keypoints = []
    else:
        keypoints = [cv2.KeyPoint(kp[0][0], kp[0][1], 1) for kp in corners]

    return keypoints
```

cv::goodFeaturesToTrack()，它不仅支持 Harris 角点检测，也支持 Shi Tomasi 算法的角点检测。但是，该函数检测到的角点依然是像素级别的，若想获取更为精细的角点坐标，则需要调用 cv::cornerSubPix() 函数进一步细化处理，即亚像素。

6.4.3 harris

```
class HARRIS:
    def detect(self, img):
        gray = np.float32(img)
        H = cv2.cornerHarris(gray, self.blockSize, self.apertureSize, self.k)
        kps = np.argwhere(H > self.T * H.max())
        kps = [cv2.KeyPoint(pt[1], pt[0], 3) for pt in kps]
        return kps
```

函数主要用于检测图像的哈里斯（Harris）角点检测，

6.4.4 factories

```
if is_cv2():
    def FeatureDetector_create(method):
        method = method.upper()
        if method == "DENSE":
            return DENSE()
        elif method == "GFTT":
            return GFTT()
        elif method == "HARRIS":
            return HARRIS()
        return cv2.FeatureDetector_create(method)

    def DescriptorExtractor_create(method):
        method = method.upper()
        if method == "ROOTSIFT":
            return RootSIFT()
        return cv2.DescriptorExtractor_create(method)

    def DescriptorMatcher_create(method):
        return cv2.DescriptorMatcher_create(method)
```

6.5 评价

整体比较简单，目前**估计已经不在维护了**（有 40+ 个 pull request 了，猜测属于无人管理状态），封装后的接口更易用，但也带来一定学习成本。建议当做 opencv 的学习材料阅读下即可，熟悉 opencv 里的方法函数（比如 **fps 计算和不同输入流的整合**）。实际项目中建议将代码**复制出来二次开发较好**，不建议在项目中使用不维护的项目。

在**代码方面可以优化的地方还是较多**的.

比如 dense 的 detect 双重循环，用推导表达式更好

```
for x in range(0, img.shape[1], self.step):
    for y in range(0, img.shape[0], self.step):
        # create a keypoint and add it to the keypoints list
        kps.append(cv2.KeyPoint(x, y, self.radius))
=>
[cv2.KeyPoint(x,y) for x in range(0, img.shape[1], self.step) for y in range(0, img.
↪shape[0], self.step)]
```

再比如:

```
    def FeatureDetector_create(method):
        method = method.upper()
        if method == "DENSE":
            return DENSE()
        elif method == "GFTT":
            return GFTT()
        elif method == "HARRIS":
            return HARRIS()
        return cv2.FeatureDetector_create(method)
```

由于算法本身是无状态的. 所以采用提前建立 map("HARRIS":HARRIS) 的方式可能更好。

并且其类名命名也不规范，有全部大写的类名, 全部大写一般出现在枚举类中等类静态配置型变量中。

6.6 参考

imutils：https://blog.csdn.net/zimiao552147572/article/details/105919440

imutils–图像处理工具包：https://www.lizenghai.com/archives/25637.html

imutils 库源码解析，看它如何调用 opencv（一）- 基本函数：www.xiaoheidiannao.com/71631.html

CHAPTER 7

开源项目阅读 06flask

7.1 基础

7.1.1 功能

Flask 是一个使用 Python 编写的轻量级 Web 应用框架, 让我们可以使用 Python 语言快速搭建 Web 服务, Flask 也被称为"microframework"，因为它使用简单的核心, 用 extension 增加其他功能。

flask 可以看做对 **jinja2 和 Werkzeug 的二次封装**。

Jinja2 是一个功能齐全的模板引擎。它有完整的 unicode 支持，所谓模板引擎，可简单理解为" **变量替换器**", 将**网页的变量填充起来**。Werkzeug 是一个 WSGI 工具包。WSGI 是一个 web 应用和服务器通信的协议，web 应用可以通过 WSGI 一起工作。

由于 Jinja2 模板引擎工作简单且界限明确，可看做黑盒，所以将 flask 看做对 Werkzeug 的二次封装也 ok。

7.1.2 代码量 (v0.1)

Language	files	blank	comment	code
Python	9	340	618	852
XML	5	0	0	312

总代码才 800+，可见整体代码并不多。

7.1.3 模块

flask v0.1 采用了单文件的方式，所有核心代码都在 flask.py 中

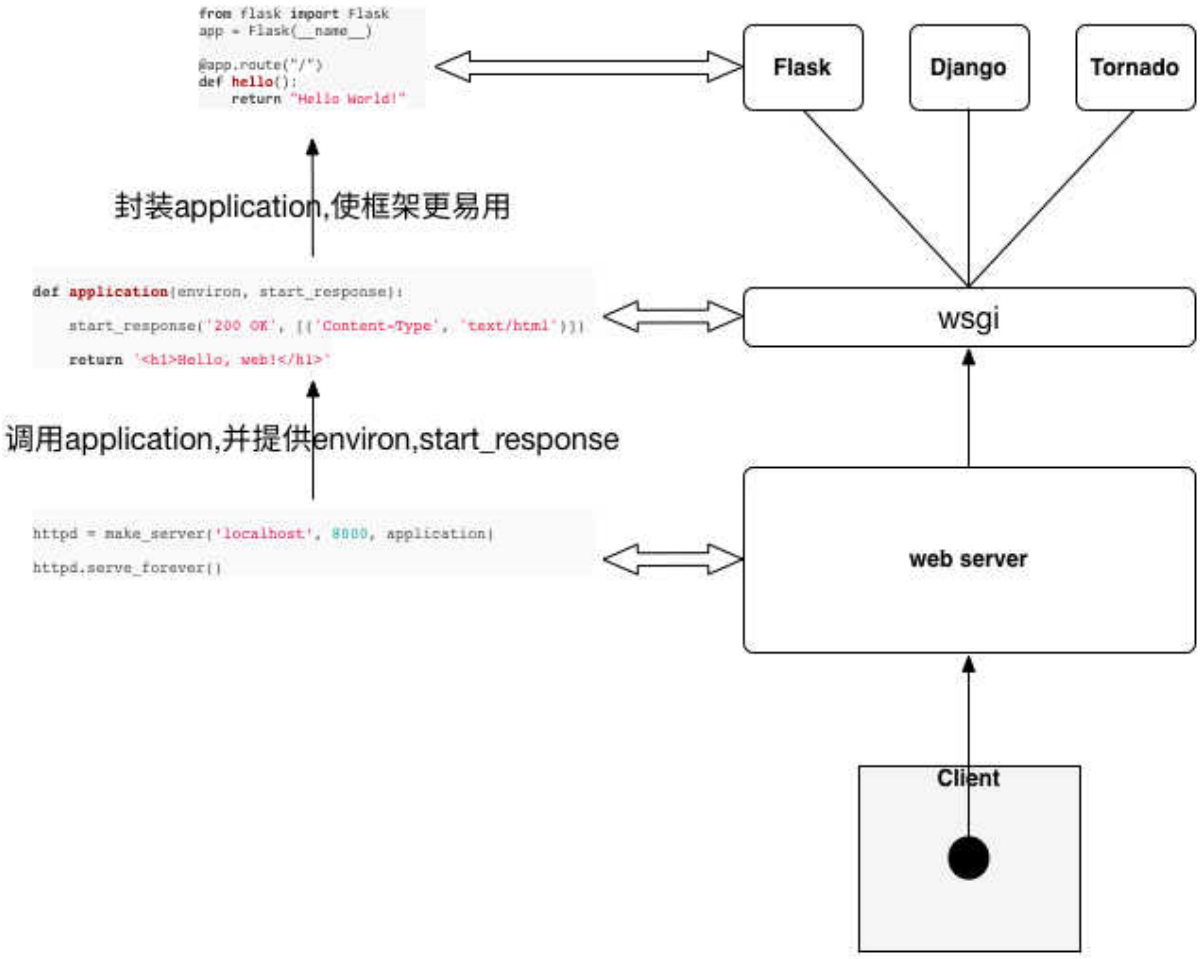
一共定义了五个类：

```
Request 和 Response，它们分别是 Flask 的请求和响应对象，分别继承自 Werkzeug 中的请求和响应类
_RequestContext，请求上下文类。它包含了所有请求的相关信息。包括程序实例 app，url 匹配器，请求对象，session 对象，g 对象以及用于记录闪现的消息的 flashes
_RequestGlobals，使用该类创建 g 对象，这个对象内没有任何的属性，你可以给该类的实例（即 g）绑定任何的全局属性。
Flask，它是整个 Flask 框架的中心类，它实现了 WSGI 程序用于处理请求和响应，并且，它是整个所有视图函数、模板配置、URL 规则的中心注册处。
```

另外，Flask 中还定义了一些函数：如 render_template、url_for、flash、get_flashed_messages 等

7.2 前置知识

下面这张图来自这里，通过这张图，读者对 web 框架所处的位置和 WSGI 协议能够有一个感性的认识。



7.2.1 WSGI

wikipedia 上对 WSGI 的解释就比较通俗易懂。为了更好的理解 WSGI，我们来看一个例子:

```
from eventlet import wsgi
import eventlet

def hello_world(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello, World!\r\n']

wsgi.server(eventlet.listen(('', 8090)), hello_world)
```

我们定义了一个 hello_world 函数，这个函数接受两个参数。分别是 environ 和 start_response，我们将这个 hello_world 传递给 eventlet.wsgi.server 以后，eventlet.wsgi.server 在调用 hello_world 时，会自动传入 environ 和 start_response 这两个参数，并接受 hello_world 的返回值。而这，就是 WSGI 的作用。

也就是说，在 python 的世界里，通过 WSGI 约定了 web 服务器怎么调用 web 应用程序的代码，web 应用程序需要符合什么样的规范，只要 web 应用程序和 web 服务器都遵守 WSGI 协议，那么，web 应用程序和 web 服务器就可以随意的组合。这也就是 WSGI 存在的原因。

WSGI 是一种协议，这里，需要注意两个相近的概念：

```
uwsgi 同 WSGI 一样是一种协议
而 uWSGI 是实现了 uwsgi 和 WSGI 两种协议的 web 服务器
```

7.2.2 jinja2

Jinja2 是一个功能齐全的模板引擎。它有完整的 unicode 支持，一个可选的集成沙箱执行环境，被广泛使用。

jinja2 的一个简单示例如下：

```
>>> from jinja2 import Template
>>> template = Template('Hello !')
>>> template.render(name='John Doe')
u'Hello John Doe!'
```

7.2.3 Werkzeug

Werkzeug 是一个 WSGI 工具包，它可以作为 web 框架的底层库。

Werkzeug 是一个 WSGI 工具包。WSGI 是一个 web 应用和服务器通信的协议，web 应用可以通过 WSGI 一起工作。一个基本的”Hello World” WSGI 应用看起来是这样的：

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello World!']
```

上面这小段代码就是 WSGI 协议的约定，它有一个可调用的 start_response 。environ 包含了所有进来的信息。start_response 用来表明已经收到一个响应。通过 Werkzeug，我们可以不必直接处理请求或者响应这些底层的东西，它已经为我们封装好了这些。

请求数据需要 environ 对象，Werkzeug 允许我们以一个轻松的方式访问数据。响应对象是一个 WSGI 应用，提供了更好的方法来创建响应。如下所示：

```
from werkzeug.wrappers import Response

```

(下页继续)

(续上页)

```
def application(environ, start_response):
    response = Response('Hello World!', mimetype='text/plain')
    return response(environ, start_response)
```

7.2.4 wsgi, Werkzeug, flask 之间的关系

Flask 是一个基于 Python 开发并且**依赖 jinja2 模板和 Werkzeug WSGI 服务的一个微型框架**，对于 Werkzeug，它只是工具包，其用于接收 http 请求并对请求进行预处理，然后触发 Flask 框架，开发人员基于 Flask 框架提供的功能对请求进行相应的处理，并返回给用户，如果要返回给用户复杂的内容时，需要借助 jinja2 模板来实现对模板的处理。将模板和数据进行渲染，将渲染后的字符串返回给用户浏览器。

7.2.5 Flask 是什么，不是什么

Flask 永远不会包含数据库层，也不会有表单库或是这个方面的其它东西。**Flask 本身只是 Werkzeug 和 Jinja2 的之间的桥梁**，前者实现一个合适的 WSGI 应用，后者处理模板。当然，Flask 也绑定了一些通用的标准库包，比如 logging。除此之外其它所有一切都交给扩展来实现。

7.3 深度阅读，调用链

7.3.1 入口

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

```
class Flask(object):
    request_class = Request
    response_class = Response
    static_path = '/static'
    secret_key = None
    session_cookie_name = 'session'

    def __init__(self, package_name):
        self.debug = False
        self.package_name = package_name
        self.root_path = _get_package_path(self.package_name)
        self.view_functions = {}
        self.error_handlers = {}
        self.before_request_funcs = []
        self.after_request_funcs = []
        self.template_context_processors = [_default_template_ctx_processor]

        self.url_map = Map()
```

```
def run(self, host='localhost', port=5000, **options):
    options.setdefault('use_reloader', self.debug)
    options.setdefault('use_debugger', self.debug)
    return run_simple(host, port, self, **options)
```

执行 app.run() 时，调用堆栈如下:

```
app.run()
    run_simple(host, port, self, **options)
        __call__(self, environ, start_response)
            wsgi_app(self, environ, start_response)
```

大多调用栈比较深，但代码都比较少，如果想深入学习，需要结合 Werkzeug 来看。

7.3.2 flask 请求响应流程 (参考文献 4)

Flask 中定义了 wsgi_app(self, environ, start_response) 方法作为 WSGI 的程序，它并没有写死在 ___call___ 方法中，因此可以为其添加中间件。当请求到来时，WSGI 服务器会调用此方法，并将请求的参数和用于发起响应的函数作为参数传递给它。

```
def wsgi_app(self, environ, start_response):
    with self.request_context(environ):
        rv = self.preprocess_request()  # 预处理请求
        if rv is None:
            rv = self.dispatch_request()  # 请求分发
        response = self.make_response(rv)  # 生成响应
        response = self.process_response(response)  # 响应处理
        return response(environ, start_response)
```

Flask 在 with 语句下执行相关操作，这会触发 _RequestContext 中的 ___enter___ 方法，从而推送请求上下文到堆栈中。在 with 中，Flask 先通过 **preprocess_request() 预处理请求，在 preprocess_request() 中调用所有在 beforce_request() 装饰器中注册的请求前要调用的函数**。随后，**Flask 使用 dispatch_request() 来进行请求分发，获得视图函数的返回值或是错误处理器的返回值**。然后 Falsk 将请求分发时获得的返回值传给 **make_response() 方法来生成一个响应对象**，接下来，Flask 在 process_response() 方法中**调用所有在 after_request() 装饰器中注册的请求完成后要调用的函数**。最后，通过 response 来发起一个响应，这会自动调用 start_response 方法来发起响应并将响应的值返回给 WSGI 服务器。

7.3.3 预处理 (参考文献 4)

```
def preprocess_request(self):
        for func in self.before_request_funcs:
            rv = func()
            if rv is not None:
                return rv
```

上面的函数会在实际的请求分发之前调用，而且将会调用每一个使用 before_request() 装饰的函数。如果其中某一个函数返回一个值，这个值将会作为视图返回值处理并停止进一步的请求处理。

7.3.4 请求分发 (参考文献 4)

```
def dispatch_request(self):
    try:
        endpoint, values = self.match_request()
        return self.view_functions[endpoint](**values)
    except HTTPException, e:
        handler = self.error_handlers.get(e.code)
        if handler is None:
            return e
        return handler(e)
    except Exception, e:
        handler = self.error_handlers.get(500)
        if self.debug or handler is None:
            raise
        return handler(e)
```

在上面的方法中，Flask 对 URL 进行匹配获取端点值和参数，然后调用相应的视图函数并将视图函数的返回值返回，或者返回相应的错误处理器的返回值。这里的返回值不一定是响应对象，比如我们可以在视图函数中返回一个字符串或者是使用 render_template() 渲染好的模板，所以，为了能够将返回值转换成合适的对象，我们需要 make_response() 方法来生成响应

7.3.5 生成响应 (参考文献 4)

```
def make_response(self, rv):
    """
    rv 允许的类型如下所示:
    ====================== ================================================
    response_class          这个对象将被直接返回
    str                     使用这个字符串作为主体创建一个请求对象
    unicode                 将这个字符串进行 utf-8 编码后作为主体创建一个请求对象
    tuple                   使用这个元组的内容作为参数创建一个请求对象
    a WSGI function         这个函数将作为 WSGI 程序调用并缓存为响应对象
    ====================== ================================================
    :param rv: 视图函数返回值
    """
    if isinstance(rv, self.response_class):
        return rv
    if isinstance(rv, basestring):
        return self.response_class(rv)
```

(下页继续)

(续上页)

```
    if isinstance(rv, tuple):
        return self.response_class(*rv)
    return self.response_class.force_type(rv, request.environ)
```

在上面的方法中，也印证了我们上面所说的请求分发中视图函数的返回值不一定是请求对象这一点。所以，我们在 make_response 方法中对请求分发中获取的返回值的类型进行判断，通过不同的方式来创建真正的响应对象并返回。

7.3.6 响应处理 (参考文献 4)

```
def process_response(self, response):
    session = _request_ctx_stack.top.session
    if session is not None:
        self.save_session(session, response)
    for handler in self.after_request_funcs:
        response = handler(response)
    return response
```

响应处理和预处理请求类似，都会循环调用所有注册的请求后调用的函数来对响应对象 response 进行处理，不过在此之前会先将 session 添加到响应对象中。

7.3.7 返回响应 (参考文献 4)

我们 Flask 中的响应对象会继承自 Werkzeug 中的 Response 对象。Response 的实例可以根据传入的参数，来发起一个特定的响应。你可以认为 Response 是你可以创建的另一个标准的 WSGI 应用，这个应用可以根据你传入的参数，来帮你做发起响应这件事。例如下面一个简易的 WSGI 程序：

```
def application(environ, start_response):
    request = Request(environ)
    response = Response("Hello %s!" % request.args.get('name', 'World!'))
    return response(environ, start_response)
```

好了，到此，Flask 的一次请求就处理完了。不难发现，在 Flask 中，对 Werzeug 这个工具库是很依赖的，从请求处理，路由匹配，到发起请求，都可见到 Werkzeug 的身影。

7.4 广度阅读，模块简析

7.4.1 2 个装饰器

self.url_map 用以保存 URI 到视图函数的映射，即保存 app.route() 这个装饰器的信息，如下所示：

```
def route(...):
    def decorator(f):
        self.add_url_rule(rule, f.__name__, **options)
        self.view_functions[f.__name__] = f
        return f
    return decorator

def errorhandler(self, code):
    def decorator(f):
        self.error_handlers[code] = f
        return f
    return decorator
```

这里被装饰器修饰过的基本上不在程序中直接调用 (find usage 找不到)，而是” 注册” 到了

```
self.add_url_rule()
self.view_functions()
self.error_handlers
```

如果想知道代码真正执行点就需要从上面入手了.

7.4.2 Local、LocalStack

这是 flask 中比较难理解的部分，位于 flask.py 尾部。

```
_request_ctx_stack = LocalStack()
current_app = LocalProxy(lambda: _request_ctx_stack.top.app)
request = LocalProxy(lambda: _request_ctx_stack.top.request)
session = LocalProxy(lambda: _request_ctx_stack.top.session)
g = LocalProxy(lambda: _request_ctx_stack.top.g)
```

向下追溯 LocalStack

```
class LocalStack(object):
    def __init__(self):
        self._local = Local()
```

(下页继续)

(续上页)

```
    def push(self,value):
        rv = getattr(self._local, 'stack', None) # self._local.stack =>local.getattr
        if rv is None:
            self._local.stack = rv = [] #  self._local.stack =>local.setattr
        rv.append(value) # self._local.stack.append(666)
        return rv


    def pop(self):
        """Removes the topmost item from the stack, will return the
        old value or `None` if the stack was already empty.
        """
        stack = getattr(self._local, 'stack', None)
        if stack is None:
            return None
        elif len(stack) == 1:
            return stack[-1]
        else:
            return stack.pop()

    def top(self):
        try:
            return self._local.stack[-1]
        except (AttributeError, IndexError):
            return None
```

再向下追溯 Local

```
class Local(object):

    def __init__(self):
        object.__setattr__(self, '__storage__', {})
        object.__setattr__(self, '__ident_func__', get_ident)

    def __getattr__(self, name):
        try:
            return self.__storage__[self.__ident_func__()][name]
        except KeyError:
            raise AttributeError(name)
```

(下页继续)

(续上页)

```
    def __setattr__(self, name, value):
        ident = self.__ident_func__()
        storage = self.__storage__
        try:
            storage[ident][name] = value # !!! 划重点，storage 第一级主键是 ident，类似线
程 id
        except KeyError:
            storage[ident] = {name: value}
```

既然是 storage[线程 id]，那么猜测 storage 应该是全局的吧 (全局含义是是进程内全局)

```
def __init__(self):
    object.__setattr__(self, '__storage__', {})
```

显然不是，storage 是 Local 实例的属性。既然是实例属性，为何还需要做线程屏蔽呢?

别忘了，最外层的 LocalStack() 其实是一种单例 (位于.py 中最外层的只会被执行一次)，LocalStack() 只会执行一次 (也就是一个 _request_ctx_stack)

再参考

```
class LocalStack(object):
    def __init__(self):
        self._local = Local()
```

也就是说只有一个 Local(), 所以其实也就只有个 storage 了，这也是为何 storage 需要做线程屏蔽的原因。

其实这里本人**有个疑问，为何采用这么复杂的方式，而不是直接用 threadlocal** 呢?

7.4.3 为何使用 LocalProxy(部分摘录自参考文献 3)

为什么一定要用 proxy，而不能直接调用 Local 或 LocalStack 对象呢? 这主要是在有多个可供调用的对象的时候会出现问题，如下图：

我们再通过下面的代码也许可以看出一二:

```
# use Local object directly
from werkzeug.local import LocalStack
user_stack = LocalStack()
user_stack.push({'name': 'Bob'})
user_stack.push({'name': 'John'})
```

(下页继续)

(续上页)

```
def get_user():
    # do something to get User object and return it
    return user_stack.pop()


# 直接调用函数获取 user 对象
user = get_user()
print user['name']
print user['name']
```

打印结果是：

```
John
John
```

再看下使用 LocalProxy

```
# use LocalProxy
from werkzeug.local import LocalStack, LocalProxy
user_stack = LocalStack()
user_stack.push({'name': 'Bob'})
user_stack.push({'name': 'John'})

def get_user():
    # do something to get User object and return it
    return user_stack.pop()

# 通过 LocalProxy 使用 user 对象
user = LocalProxy(get_user)
print user['name']
print user['name']
```

打印结果是：

```
John
Bob
```

怎么样，看出区别了吧，直接使用 LocalStack 对象，user 一旦赋值就无法再动态更新了，而使用 Proxy，每次调用操作符 (这里 [] 操作符用于获取属性)，都会重新获取 user，从而实现了动态更新 user 的效果。见下图：

Flask 以及 Flask 的插件很多时候都需要这种动态更新的效果，因此 LocalProxy 就会非常有用了。

那么这个解释有问题么?

想想这里的 request,session,g 是让谁调用的？当然是 flask 的使用方，就是 web 应用开发者。

如果**开发者调用一次 session['xx'] 后，session 对象就变成其他线程的 session 对象，那就麻烦了**.

继续深究

```
session = LocalProxy(lambda: _request_ctx_stack.top.session)

session['xx']=>等价于=>LocalProxy(lambda: _request_"ctx_stack.top.session)['xx']
```

结合

LocalPorxy 部分源代码:

```
# LocalProxy 部分代码

@implements_bool
class LocalProxy(object):
    __slots__ = ('__local', '__dict__', '__name__', '__wrapped__')

    def __init__(self, local, name=None):#!!! 划重点  这里的 local 就是 lambda: _request_
↪"ctx_stack.top.session
        object.__setattr__(self, '_LocalProxy__local', local)
        object.__setattr__(self, '__name__', name)
        if callable(local) and not hasattr(local, '__release_local__'):
            # "local" is a callable that is not an instance of Local or
            # LocalManager: mark it as a wrapped function.
            object.__setattr__(self, '__wrapped__', local)

    def _get_current_object(self):
        """Return the current object.  This is useful if you want the real
        object behind the proxy at a time for performance reasons or because
        you want to pass the object into a different context.
        """
        # 由于所有 Local 或 LocalStack 对象都有__release_local__ method, 所以如果没有该属性
就表明 self.__local 为 callable 对象
        if not hasattr(self.__local, '__release_local__'):
            return self.__local()
        try:
            # 此处 self.__local 为 Local 或 LocalStack 对象
            return getattr(self.__local, self.__name__)
```

(下页继续)

(续上页)

```
        except AttributeError:
            raise RuntimeError('no object bound to %s' % self.__name__)

    @property
    def __dict__(self):
        try:
            return self._get_current_object().__dict__
        except RuntimeError:
            raise AttributeError('__dict__')

    def __getattr__(self, name):
        if name == '__members__':
            return dir(self._get_current_object())
        return getattr(self._get_current_object(), name)

    def __setitem__(self, key, value):
        self._get_current_object()[key] = value

    def __delitem__(self, key):
        del self._get_current_object()[key]

    if PY2:
        __getslice__ = lambda x, i, j: x._get_current_object()[i:j]

        def __setslice__(self, i, j, seq):
            self._get_current_object()[i:j] = seq

        def __delslice__(self, i, j):
            del self._get_current_object()[i:j]

    # 截取部分操作符代码
    __setattr__ = lambda x, n, v: setattr(x._get_current_object(), n, v)
    __delattr__ = lambda x, n: delattr(x._get_current_object(), n)
    __str__ = lambda x: str(x._get_current_object())
    __lt__ = lambda x, o: x._get_current_object() < o
    __le__ = lambda x, o: x._get_current_object() <= o
    __eq__ = lambda x, o: x._get_current_object() == o
```

同时可能注意到 _get_current_object 中 getattr(self.___local,xx) 但是 self.___local 是不存在的，其实就是

```
object.__setattr__(self, '_LocalProxy__local', local)
```

里面的 _LocalProxy___local,

因为在类里面，双下划线标注的变量，会被自动转换为 _yourclass___yourvariables 的形式，你可以试下在某个类中定义一个双下划綫变量，然后实例化出来输出 ___dict___ 查看

继续前面的分析:

```
session['xx']=> 等价于 =>LocalProxy(lambda: _request_"ctx_stack.top.session)['xx']
=> LocalProxy.__getattr__('xx')
def __getattr__(self, name):
    if name == '__members__':
        return dir(self._get_current_object())
    return getattr(self._get_current_object(), name)

=> self._get_current_object().xx
def _get_current_object(self):
    return getattr(self.__local, self.__name__)
    或者
    return self.__local()
    此处认为 xx 是普通属性，不可 run() (localstack 有__release_local__ 属性) ，所以走前面
(第一个) 的分支代码

=> self.__local.xx# 这里的__local 是 localstack 的 local
=> self.__storage__[线程 id].xx
```

这才是正确解读，至于” 参考文献 3” 解释虽然是正确的，但是走的 _get_current_object 的另一个分支代码

7.5 参考

浅谈 flask 源码之请求过程:https://www.jb51.net/article/144480.htm

Flask 源码剖析详解:https://www.cnblogs.com/caoxing2017/p/8282530.html

Werkzeug(Flask) 之 Local、LocalStack 和 LocalProxy:https://www.jianshu.com/p/3f38b777a621

Flask 源码浅析:https://www.cnblogs.com/tcctw/p/10743518.html

Flask 源码剖析（二）:https://blog.csdn.net/weixin_42239402/article/details/97618844

Flask 部分源码阅读:https://my.oschina.net/jianming/blog/2208507

flask-annotated:https://github.com/hhstore/annotated-py-projects/tree/master/flask

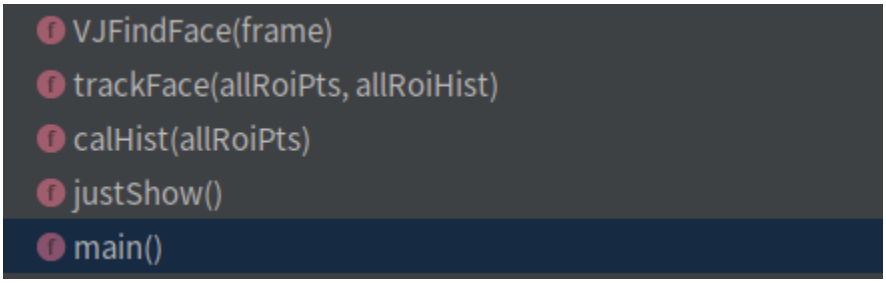
CHAPTER 8

开源项目阅读 07 人脸检测和追踪

代码来源:github,search,face Detection and track

8.1 dstoyanova/face-detection-and-tracking.git

主逻辑:VJCMS.py



8.1.1 主入口 main

```
def main():
    global cap
    i=0
    while(cap.isOpened()):
        if i % 2 == 0:
            ret, frame = cap.read()
```

(下页继续)

(续上页)

```
            allRoiPts = VJFindFace(frame)
            allRoiHist = calHist(allRoiPts)
        else:
            error = trackFace(allRoiPts, allRoiHist)
    cap.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

8.1.2 追溯 VJFindFace

功能: 绘图传入的 frame, 返回人脸坐标 boxes 的 List(allRoiPts)

```
def VJFindFace(frame):
    dim = (frame.shape[1]/RATIO, frame.shape[0]/RATIO);
    resized = cv2.resize(frame, dim, interpolation = cv2.INTER_AREA)
    gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
    fd = FaceDetector('cascades/haarcascade_frontalface_default.xml')
    faceRects = fd.detect(gray, scaleFactor = 1.1, minNeighbors = 5, minSize = (10, 10))
    for (x, y, w, h) in faceRects:
        # decrease the size of the bounding box
        x = RATIO*(x+10)
        y = RATIO*(y+10)
        w = RATIO*(w-15)
        h = RATIO*(h-15)

        # insert the coordinates of each face to the list
        allRoiPts.append((x, y, x+w, y+h))

    # show the detected faces
    cv2.imshow("Faces", frame)
    cv2.waitKey(1)
    return allRoiPts
```

8.1.3 追溯 calHist

功能: 计算传入的各个 boxes 的小图的 hsv 空间的直方图, 并对直方图归一化后返回,

```
def calHist(allRoiPts):
    global orig
    allRoiHist = []
    for roiPts in allRoiPts:
        roi = orig[roiPts[1]:roiPts[-1], roiPts[0]:roiPts[2]]
        roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
        roiHist = cv2.calcHist([roi], [0], None, [16], [0, 180])
        roiHist = cv2.normalize(roiHist, roiHist, 0, 255, cv2.NORM_MINMAX)
        allRoiHist.append(roiHist);
    return allRoiHist
```

8.1.4 追溯 trackFace

功能说明: 用直方图反投影 (calcBackProject) 和 CamShift 算法，追踪其他帧的人脸信息, 结果划线方式添加到 frame 中

```
def trackFace(allRoiPts, allRoiHist):
    for k in range(0, TRACK):
        ret, frame = cap.read()
        i=0
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        for roiHist in allRoiHist:
            backProj = cv2.calcBackProject([hsv], [0], roiHist, [0, 180], 1)
            (r, allRoiPts[i]) = cv2.CamShift(backProj, allRoiPts[i], termination)
            for j in range(0,4):
                if allRoiPts[i][j] < 0:
                    allRoiPts[i][j] = 0
            pts = np.int0(cv2.cv.BoxPoints(r))
            cv2.polylines(frame, [pts], True, (0, 255, 255), 2)
            i = i + 1
        cv2.imshow("Faces", frame)
        cv2.waitKey(1)
    return 1;
```

8.1.5 评价

程序本身还是非常简单的，**识别算法 cv2.CascadeClassifier，追踪算法 CamShift，中间用了直方图反传**(本人不才，至今没看懂直方图反传是如何辅助 camshift，从而实现跟踪的)。

代码本身质量并不高，先不说**变量命名是否易懂规范**（高阶，这个难度很高，完美的变量命名是不再需要注释进行程序逻辑解释的，**变量名除了用了“标识”数据，让程序运行正确的角色外**，同时也担负着**解释程序**

功能，运行逻辑的角色，这也就是为何很多受欢迎的开源代码中有非常多的短函数，哪怕只在一个地方被调用（言外之意，不是被强迫抽离为函数的），也要独立出来，就是为了更好的可读性，完成“注释代码”的角色），或者 global 变量的滥用（低阶），**仅仅在循环中 new VJFindFace 就是个糟糕的方法**，会导致程序效率低下，这个问题是比较严重的。

这里也并非鄙视原作者，自己也曾写过很多烂代码，个人认为**烂代码是程序员成长的必经之路**，你无法对要求 1 **岁小孩抵挡你 20 岁的小伙子一拳，哪怕这小孩是未来的拳王**。

8.2 shreyasbhatia09/Face-Detection-and-Tracking

8.2.1 主入口 main

```
if __name__ == '__main__':
    question_number = 2

    # read video file
    video = cv2.VideoCapture('../Input/02-1.avi');

    if (question_number == 1):
        CAM_shift_tracker(video, "output_camshift.txt")
    elif (question_number == 2):
        particle_filter_tracker(video, "output_particle.txt")
```

(下页继续)

(续上页)

```
    elif (question_number == 3):
        kalman_filter_tracker(video, "output_kalman.txt")
    elif question_number == 4:
        optical_flow_tracker(video, "output_of.txt")
```

8.3 Linzaer/Face-Track-Detect-Extract

8.3.1 入口 start.py

```
def main():
    tracker = Sort()
    pnet, rnet, onet = detect_face.create_mtcnn(sess, os.path.join(project_dir, "align"))
    cam = cv2.VideoCapture(video_name)
    if c % detect_interval == 0:
        faces, points = detect_face.detect_face(r_g_b_frame, minsize, pnet, rnet, onet,␣
↪threshold,factor)
        judge_side_face( np.array(facial_landmarks))
    trackers = tracker.update(final_faces, img_size, directoryname, addtional_attribute_
↪list, detect_interval)
    cv2.rectangle(frame, (d[0], d[1]), (d[2], d[3]), colours[d[4] % 32, :] * 255, 3)
```

8.3.2 追溯 detect_face.create_mtcnn(sess, model_path):

```
with tf.variable_scope('pnet'):
with tf.variable_scope('rnet'):
with tf.variable_scope('onet'):
pnet_fun = lambda img
rnet_fun = lambda img
onet_fun = lambda img
return pnet_fun, rnet_fun, onet_fun
```

猜测是创建 tf 网络结构

8.3.3 追溯 def detect_face(img, minsize, pnet, rnet, onet, threshold, factor)

```
这个有点复杂，尚未看懂
```

8.3.4 追溯 def judge_side_face(facial_landmarks):

```
也没太懂
high_ratio_variance = np.fabs(high_rate - 1.1)  # smaller is better
width_ratio_variance = np.fabs(width_rate - 1)
```

8.3.5 追溯 def update(self, dets, img_size, root_dic, addtional_attribute_list, predict_num):

```
matched, unmatched_dets, unmatched_trks = associate_detections_to_trackers(dets, trks)
for t, trk in enumerate(self.trackers):# 已有搭配的 track 对象
    if t not in unmatched_trks:
        d = matched[np.where(matched[:, 1] == t)[0], 0]
        trk.update(dets[d, :][0])
        trk.face_addtional_attribute.append(addtional_attribute_list[d[0]])
for i in unmatched_dets:# 没有搭配的 track 则创建新 tracker
    trk = KalmanBoxTracker(dets[i, :])
    trk.face_addtional_attribute.append(addtional_attribute_list[i])
```

整体代码有点复杂，挺多地方看的不是很明白，等以后经验丰富了再回来看吧

8.4 ZidanMusk/experimenting-with-sort

8.4.1 入口 main.py

```
def main():
        detector = GroundTruthDetections()
        tracker =  Sort(use_dlib= use_dlibTracker) #create instance of the SORT tracker
        with open(out_file, 'w') as f_out:
        frames = detector.get_total_frames()
                for frame in range(0, frames):  #frame numbers begin at 0!
                detections = detector.get_detected_items(frame)
                trackers = tracker.update(detections,img)
            `   for d in trackers:
                  plot(trackers)
```

8.4.2 追溯 class Sort: 创建新 tracker 或 update 已有 tracker

```
def update(self,dets,img=None):
    for t,trk in enumerate(trks):
      pos = self.trackers[t].predict(img) #for kal!
    if dets != []:
       matched, unmatched_dets, unmatched_trks = associate_detections_to_trackers(dets,
↪trks)
       #update matched trackers with assigned detections
      for t,trk in enumerate(self.trackers):
        trk.update(dets[d,:][0],img)    for dlib re-intialize the trackers ?!
       #create and initialise new trackers for unmatched detections
       for i in unmatched_dets:
        trk = KalmanBoxTracker(dets[i,:])
      self.trackers.append(trk)
```

看到这里基本清晰了。

main 里面调用的 tracker.update(detections,img)

实际内部可能 update(已有 tracker)，也可能是 create(没有匹配 tracker)

那么 tracker 和检测出的 detection 是如何匹配的呢? 答案是计算 iou，之后用匈牙利二分图匹配算法.

8.5 twairball/face_tracking

8.5.1 入口 run

核心代码

```
def run
        while True:
            boxes, detected_new = pipeline.boxes_for_frame(frame)
            draw_boxes(frame, boxes, color)
```

8.5.2 追溯 pipeline

核心代码

```
def boxes_for_frame(self, frame):
        if self.controller.trigger():
```

(下页继续)

(续上页)

```
                return self.detect_and_track(frame)
        else:
            return self.track(frame)

def __init__(self, event_interval=6):
        self.controller = Controller(event_interval=event_interval)
        self.detector = FaceDetector()
        self.trackers = []

def detect_and_track(self, frame):
        faces = self.detector.detect(frame)
        self.trackers = [FaceTracker(frame, face) for face in faces]

def track(self, frame):
        boxes = [t.update(frame) for t in self.trackers]
        return boxes, False
```

可见其机制在于

如果时间间隔足够久，则 self.detect_and_track(detect 消耗资源多，速度慢)

否则:self.track(frame)(因为 track 的消耗资源少，速度快)

8.5.3 追溯 FaceDetector 和 FaceTracker

核心代码

```
class FaceDetector():
        def __init__(self, cascPath="./haarcascade_frontalface_default.xml"):
                self.faceCascade = cv2.CascadeClassifier(cascPath)
        def detect(self, frame):
                return self.faceCascade.detectMultiScale

class FaceTracker():
        def __init__(self, frame, face):
                self.tracker = cv2.TrackerKCF_create()
                self.tracker.init(frame, self.face)
        def update(self, frame):
                return self.tracker.update(frame)
```

代码非常简单优雅, 尤其是和上一部分的

```
faces = self.detector.detect(frame)
self.trackers = [FaceTracker(frame, face) for face in faces]
```

结合起来，一段时间后自动重新检测，重新更新所有 track, 代码通俗简单

8.5.4 评价

代码简单，明了，易懂。不论是变量命名还是代码结构，都比较合理，这才是“诗一样的代码”。

8.6 dlib 目标跟踪

地址:https://blog.csdn.net/weixin_40277254/article/details/82188461

使用 dlib.correlation_tracker 实现目标跟踪基本分以下四步：

dlib.correlation_tracker() 创建一个跟踪类；

start_track() 中设置图片中的要跟踪物体的框；

update() 实时跟踪下一帧；

get_position() 得到跟踪到的目标的位置。

核心代码:

```
tracker = dlib.correlation_tracker()    # 导入 correlation_tracker() 类
while(1):
        if(第一帧)
                tracker.start_track(image, dlib.rectangle(track_window[0], track_
↪window[1], track_window[2], track_window[3]))
        else:
                tracker.update(image)      # 更新，实时跟踪
        box_predict = tracker.get_position()       # 得到目标的位置
        cv2.rectangle(image,(int(box_predict.left()),int(box_predict.top())),(int(box_
↪predict.right()),int(box_predict.bottom())),(0,255,255),1)
        cv2.imshow('image',image)
cv2.destroyAllWindows()
```

8.7 ITCoders/Human-detection-and-Tracking

入口 main.py

```
recognizer.read("model.yaml")# 读取训练好的人脸识别模型（所以后面直接 predict 了）
for video in list_of_videos:
    while True:
        temp = background_subtraction(previous_frame, frame_resized_grayscale, min_area)#
图片变化是否超过阈值
        if temp == 1:
            frame_processed = detect_people(frame_resized)# 检测人
            faces = detect_face(frame_resized_grayscale)# 检测脸
            if len(faces) > 0:
                frame_processed = draw_faces(frame_processed, faces)# 图片上画出脸
                label = recognize_face(frame_resized, faces)# 识别人
                frame_processed = put_label_on_face(frame_processed, faces, label)# 画出
的脸添加标签（姓名）

            cv2.imshow("Detected Human and face", frame_processed)
```

代码简单，不在赘述

这个项目的 scripts 里面有很多脚本，对于 opencv 的学习比较有帮助，有兴趣自行阅读

8.8 其他

nishagandhi/Face-Detection-and-Tracking(略,shr 的早期版本)

brpat07/Face-Detection-and-Tracking

8.9 参考

Python-OpenCV 创建人脸识别器 (LBPHFaceRecognizer_create 的使用方法): https://blog.csdn.net/weixin_44803791/article/details/103203143

基于 LBPH 的人脸识别（LBPHFaceRecognizer_create 的使用方法，仅预测）: https://www.cnblogs.com/monsterhy123/p/12930903.html

目标跟踪初探（DeepSORT）:https://zhuanlan.zhihu.com/p/90835266

图像人脸进行检测、识别和跟踪 (大小, 分辨率, 模糊, 光照, 角度, 遮挡): https://blog.csdn.net/zhiboxu9716/article/details/79344910

python OpenCV 图片相似度 5 种算法 (均值哈希算法相似度、差值哈希算法相似度、感知哈希算法相似度、三直方算法相似度、单通道直方图相似度): https://blog.csdn.net/enter89/article/details/90293971

CHAPTER 9

Indices and tables

- genindex
- modindex
- search