

装饰器的应用

一、实现一个cache装饰器，实现可过期被清除的功能

简化设计，函数的形参定义不包含可变位置参数、可变关键词参数和keyword-only参数
可以不考虑缓存满了之后的换出问题

数据类型的选择

缓存的应用场景，是有数据需要频繁查询，且每次查询都需要大量计算或者等待时间之后才能返回结果的情况，使用缓存来提高查询速度，用空间换取时间。

cache应该选用什么数据结构？

便于查询的，且能快速好到数据的数据结构。

每次查询的时候，只要输入一致，就应该得到同样的结果（顺序也一致，例如减法函数，参数顺序不一致，结果不一样）

基于上面的分析，此数据结构应该是字典。

通过一个key，对应一个value。

key是参数列表组成的结构，value是函数返回值。难点在于key如何处理。

key的存储

key必须是hashable。

key能接受到位置参数和关键字参数传参。

位置参数是被收集在一个tuple中的，本身就有顺序。

关键字参数被收集在一个字典中，本身无序，这会带来一个问题，传参的顺序未必是字典中保存的顺序。如何解决？

OrderedDict行吗？可以，它可以记录顺序。

不用OrderedDict行吗？可以，用一个tuple保存排过序的字典的item的kv对。

key的异同

什么才算是相同的key呢？

```
import time
import functools
```

```
@functools.lru_cache()
def add(x,y):
    time.sleep(3)
    return x+y
```

定义一个加法函数，那么传参方式就应该有以下4种：

1. add(4, 5)
2. add(4, y=5)
3. add(y=4, x=5)
4. add(x=5, y=4)

上面4种，可以有下面两种理解：

第一种：3和4相同，1、2和3都不同。

第二种：1、2、3、4全部相同。

lru_cache实现了第一种，可以看出单独的处理了位置参数和关键字参数。

但是函数定义为def add(4, y=5)，使用了默认值，如何理解add(4, 5)和add(4)是否一样呢？

如果认为一样，那么lru_cache无能为力。

就需要使用inspect来自自己实现算法

key的要求

key必须是hashable。

由于key是所有实参组合而成，而且最好要作为key的，key一定要可以hash，但是如果key有不可hash类型数据，就无法完成。

lru_cache就不可以。

```
def add1(x,y):
    return y

>>>add1([],5)
5

@functools.lru_cache()
def add(x,y=5):
    time.sleep(3)
    return y

>>>add(4)
5
```

```
>>>add([],5)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-46-a6b43c6e78a2> in <module>()  
----> 1 add([],5)
```

```
TypeError: unhashable type: 'list'
```

缓存必须使用key，但是key必须可hash，所以只能适用实参是不可变类型的函数调用。

key算法设计

inspect模块获取函数签名后，取parameters，这是一个有序字典，会保存所有参数的信息。

构建一个字典params_dict，按照位置顺序从args中依次对应参数名和传入的实参，组成kv对，存入params_dict中。

kwargs所有值update到params_dict中。

如果使用了缺省值的参数，不会出现在实参params_dict中，会出现在签名的取parameters中，缺省值也在定义中。

调用的方式

普通的函数调用可以，但是过于明显，最好类似lru_cache的方式，让调用者无察觉的使用缓存。构建装饰器函数。

代码模板如下：

```
from functools import wraps  
import inspect  
  
def mag_cache(fn):  
    local_cache = {} # 对不同函数名是不同的cache  
    @wraps(fn)  
    def wrapper(*args, **kwargs): #接收各种参数  
        # 参数处理，构建key  
        ret = fn(*args, **kwargs)  
        return ret  
    return wrapper  
  
@mag_cache  
def add(x,y,z=6):  
    return x + y + z
```

目标

```
def add(x, z, y=6):  
    return x + y + z
```

```
add(4, 5)  
add(4, z=5)  
add(4, y=6, z=5)  
add(y=6, z=5, x=4)  
add(4, 5, 6)
```

上面几种都等价，也就是key一样，这样都可以缓存。

代码实现

完成了key的生成。

注意，这里使用了普通的字典params_dict，先把位置参数的对应好，再填充关键字参数，最后补充缺省值，然后再排序生成key。

```
from functools import wraps  
import inspect  
  
def mag_cache(fn):  
    local_cache = {} # 对不同函数名是不同的cache  
  
    @wraps(fn)  
    def wrapper(*args, **kwargs): # 接收各种参数,kwarg普通字典参数无序  
        # 参数处理，构建key  
        sig = inspect.signature(fn)  
        params = sig.parameters # 只读有序字典  
  
        param_names = [key for key in params.keys()] # list(params.keys())  
        params_dict = {}  
  
        for i, v in enumerate(args):  
            k = param_names[i]  
            params_dict[k] = v  
  
        # for k, v in kwargs.items():
```

```

#     params_dict[k] = v
params_dict.update(kwargs)

# 缺省值处理
for k,v in params.items():
    if k not in params_dict.keys():
        params_dict[k] = v.default

key = tuple(sorted(params_dict.items()))
# 判断是否需要缓存

ret = fn(*args, **kwargs)
return key, ret

return wrapper

```

```

@mag_cache
def add(x, z, y=6):
    return x + y + z

```

```

result = []
result.append(add(4, 5))
result.append(add(4, z=5))
result.append(add(4, y=6, z=5))
result.append(add(y=6, z=5, x=4))
result.append(add(4, 5, 6))

```

使用缓存

```

import time
import inspect
from functools import wraps

def mag_cache(fn):
    local_cache = {} # 对不同函数名是不同的cache

    @wraps(fn)
    def wrapper(*args, **kwargs): # 接收各种参数,kwags普通字典参数无序

```

```

# 参数处理, 构建key
sig = inspect.signature(fn)
params = sig.parameters # 只读有序字典

param_names = [key for key in params.keys()]
params_dict = {}

for i, v in enumerate(args):
    k = param_names[i]
    params_dict[k] = v

# for k, v in kwargs.items():
#     params_dict[k] = v
params_dict.update(kwargs)

# 缺省值处理
for k,v in params.items():
    if k not in params_dict.keys():
        params_dict[k] = v.default

key = tuple(sorted(params_dict.items()))
# 判断是否需要缓存
if key not in local_cache.keys():
    local_cache[key] = fn(*args, **kwargs)

return key, local_cache[key]

```

```

return wrapper

```

```

@mag_cache

```

```

def add(x, z, y=6):
    time.sleep(3)
    return x + y + z

```

```

result = []
result.append(add(4, 5))
result.append(add(4, z=5))
result.append(add(4, y=6, z=5))

```

```
result.append(add(y=6, z=5, x=4))
result.append(add(4, 5, 6))

for x in result:
    print(x)
```

增加logger装饰器查看执行时间

```
def logger(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs)
        delta = (datetime.datetime.now() - start).total_seconds()
        print(fn.__name__, delta)
        return ret
    return wrapper

@logger
@mag_cache
def add(x, z, y=6):
    time.sleep(3)
    return x + y + z
```

过期功能

一般缓存系统都有过期功能。

过期什么？

它是某一个key过期。可以对每一个key单独设置过期时间，也可以对这些key统一设定过期时间。

本次的实现就简单点，统一设定key的过期时间，当key生存超过了这个时间，就自动被清除。

注意：这里并没有考虑多线程等问题。而且这种过期机制，每一次都有遍历所有数据，大量数据的时候，遍历可能有效率问题。

在上面的装饰器中增加一个参数，需要用到了带参装饰器了。

@mag_cache(5)代表key生存5秒钟后过期。

带参装饰等于在原来的装饰器外面在嵌套一层。

清除的时机，何时清除过期key？

- 1、用到某个key之前，先判断是否过期，如果过期重新调用函数生成新的key对应value值。
- 2、一个线程负责清除过期的key，这个以后实现。本次在创建key之前，清除所有过期的key。

value的设计

1、key => (v, createtimestamp)

适合key过期时间都是统一的设定。

2、key => (v, createtimestamp, duration)

duration是过期时间，这样每一个key就可以单独控制过期时间。在这种设计中，-1可以表示永不过期，0可以表示立即过期，正整数表示持续一段时间过期。

本次采用第一种实现。

```
import time
import inspect
from functools import wraps
import datetime

def mag_cache(duration):
    def _cache(fn):
        local_cache = {} # 对不同函数名是不同的cache

        @wraps(fn)
        def wrapper(*args, **kwargs): # 接收各种参数,kwarg普通字典参数无序
            # 清除过期的key
            expire_keys = []
            for k, (_, stamp) in local_cache.items():
                now = datetime.datetime.now().timestamp()
                if now - stamp > duration:
                    expire_keys.append(k)
            for k in expire_keys:
                local_cache.pop(k)

            # 参数处理，构建key
            sig = inspect.signature(fn)
            params = sig.parameters # 只读有序字典

            param_names = [key for key in params.keys()]
            params_dict = {}

            for i, v in enumerate(args):
                k = param_names[i]
                params_dict[k] = v
```



```

# for k, v in kwargs.items():
#     params_dict[k] = v
params_dict.update(kwargs)

# 缺省值处理
for k,v in params.items():
    if k not in params_dict.keys():
        params_dict[k] = v.default

key = tuple(sorted(params_dict.items()))

# 判断是否需要缓存
if key not in local_cache.keys():
    local_cache[key] = (fn(*args, **kwargs),
                        datetime.datetime.now().timestamp()) # 时间戳

    return key, local_cache[key]

return wrapper

return _cache

def logger(fn):
    @wraps(fn)
    def wrapper(*args,**kwargs):
        start = datetime.datetime.now()
        ret = fn(*args, **kwargs)
        delta = (datetime.datetime.now() - start).total_seconds()
        print(fn.__name__, delta)
        return ret
    return wrapper

@logger
@mag_cache(10)
def add(x, z, y=6):
    time.sleep(3)
    return x + y + z

```

```
result = []
result.append(add(4, 5))
result.append(add(4, z=5))
result.append(add(4, y=6, z=5))
result.append(add(y=6, z=5, x=4))
result.append(add(4, 5, 6))
result.append(add(4,6))
```

```
for x in result:
    print(x)
```

```
time.sleep(10)
result = []
result.append(add(4, 5))
result.append(add(4, z=5))
result.append(add(4, y=6, z=5))
result.append(add(4,6))
```

抽象函数

```
import time
import inspect
from functools import wraps
import datetime

def mag_cache(duration):
    def _cache(fn):
        local_cache = {} # 对不同函数名是不同的cache

        @wraps(fn)
        def wrapper(*args, **kwargs): # 接收各种参数,kwags普通字典参数无序
            def clear_expire(cache):
                # 清除过期的key
                expire_keys = []
                for k, (_, stamp) in cache.items():
                    now = datetime.datetime.now().timestamp()
                    if now - stamp > duration:
                        expire_keys.append(k)
                for k in expire_keys:
```

```
cache.pop(k)
```

```
clear_expire(local_cache)
```

```
def make_key():
```

```
    # 参数处理, 构建key
```

```
    sig = inspect.signature(fn)
```

```
    params = sig.parameters # 只读有序字典
```

```
    param_names = [key for key in params.keys()]
```

```
    params_dict = {}
```

```
    for i, v in enumerate(args):
```

```
        k = param_names[i]
```

```
        params_dict[k] = v
```

```
    # for k, v in kwargs.items():
```

```
    #     params_dict[k] = v
```

```
    params_dict.update(kwargs)
```

```
    # 缺省值处理
```

```
    for k,v in params.items():
```

```
        if k not in params_dict.keys():
```

```
            params_dict[k] = v.default
```

```
    return tuple(sorted(params_dict.items()))
```

```
key = make_key()
```

```
# 判断是否需要缓存
```

```
if key not in local_cache.keys():
```

```
    local_cache[key] = (fn(*args, **kwargs),
```

```
                        datetime.datetime.now().timestamp()) # 时间戳
```

```
    return key, local_cache[key]
```

```
return wrapper
```

```
return _cache
```

```
def logger(fn):
```

```
    @wraps(fn)
```

```
    def wrapper(*args, **kwargs):
```

```
start = datetime.datetime.now()
ret = fn(*args, **kwargs)
delta = (datetime.datetime.now() - start).total_seconds()
print(fn.__name__, delta)
return ret
return wrapper
```

```
@logger
@mag_cache(10)
def add(x, z, y=6):
    time.sleep(3)
    return x + y + z
```

```
result = []
result.append(add(4, 5))
result.append(add(4, z=5))
result.append(add(4, y=6, z=5))
result.append(add(y=6, z=5, x=4))
result.append(add(4, 5, 6))
result.append(add(4,6))
```

```
for x in result:
    print(x)
```

```
time.sleep(10)
result = []
result.append(add(4, 5))
result.append(add(4, z=5))
result.append(add(4, y=6, z=5))
result.append(add(4,6))
```

如果使用OrderedDict，要注意，顺序要以签名声明的顺序为准。

```
def make_key():
    # 参数处理，构建key
    sig = inspect.signature(fn)
    params = sig.parameters # 只读有序字典
```

```
param_names = [key for key in params.keys()]
```

```
params_dict = OrderedDict() #{} 
```

```
for i, v in enumerate(args):
```

```
    k = param_names[i]
```

```
    params_dict[k] = v
```

```
# for k, v in kwargs.items():
```

```
#     params_dict[k] = v
```

```
# params_dict.update(kwargs)
```

```
# 缺省值和关键字参数处理
```

```
# 如果在params_dict中，说明是位置参数
```

```
# 如果不在params_dict中，如果在kwargs中，使用kwargs的值，如果也不在kwargs中，就使用缺省值
```

```
for k,v in params.items(): # 顺序由签名的顺序定
```

```
    if k not in params_dict.keys():
```

```
        if k in kwargs.keys():
```

```
            params_dict[k] = kwargs[k]
```

```
        else:
```

```
            params_dict[k] = v.default
```

```
return tuple(params_dict.items())
```

二、写一个命令分发器

程序员可以方便的注册函数到某一个命令，用户输入命令时，路由到注册的函数

如果此命令没有对应的注册函数，执行默认函数

用户输入用input(">>")

分析

输入命令映射到一个函数，并执行这个函数。应该是cmd_tbl[cmd] = fn的形式，字典正好合适。

如果输入了某一个cmd命令后，没有找到函数，就要调用缺省的函数执行，这正好是字典缺省参数。

cmd是字符串。

基础框架

```
# 构建全局字典
cmd_tbl = {}

# 注册函数
def reg(cmd, fn):
    cmd_tbl[cmd] = fn

# 缺省函数
def default_func():
    print('Unknown command')

# 调度器
def dispatcher():
    while True:
        cmd = input('>>>')
        # 退出条件
        if cmd.strip() == '':
            return
        cmd_tbl.get(cmd, default_func)()

# 自定义函数
def foo1():
    print('magedu')

def foo2():
    print('python')

# 注册函数
reg('mag', foo1)
reg('py', foo2)

# 调度循环
dispatcher()
```

这个代码有些弊端：

- 1、函数的注册太丑
- 2、所有的函数和字典都在全局中定义，不好如何改进？

封装

将reg函数封装成装饰器，并用它来注册函数

```
# 注册函数
def reg(cmd):
    def _reg(fn):
        cmd_tbl[cmd] = fn
        return fn
    return _reg
```

```
# 自定义函数
@reg('mag')
def foo1():
    print('magedu')
```

```
@reg('py')
def foo2():
    print('python')
```

是否能把字典、reg、dispatcher等封装起来，因为外面只要使用调度和注册就可以了。

```
def command_dispatcher():
    # 构建全局字典
    cmd_tbl = {}

    # 注册函数
    def reg(cmd):
        def _reg(fn):
            cmd_tbl[cmd] = fn
            return fn
        return _reg

    # 缺省函数
    def default_func():
        print('Unknown command')

    # 调度器
```

```
def dispatcher():
    while True:
        cmd = input('>>')
        # 退出条件
        if cmd.strip() == '':
            return
        cmd_tbl.get(cmd, default_func)()

    return reg, dispatcher

reg, dispatcher = command_dispatcher()

# 自定义函数
@reg('mag')
def foo1():
    print('magedu')

@reg('py')
def foo2():
    print('python')

# 调度循环
dispatcher()
```

问题

重复注册的问题

如果一个函数使用同样的cmd名称注册，就等于覆盖了原来的cmd到fn的关系，这样的逻辑也是合理的。

也可以加一个判断，如果key已经存在，重复注册，抛出异常。看业务要求。

注销

有注册就应该有注销。

一般来说注销是要有权限的，但是什么样的人拥有注销的权限，看业务要求。

装饰器的用途

装饰器是**AOP面向切面编程 Aspect Oriented Programming**的思想的体现。

面向对象往往需要通过继承或者组合依赖等方式调用一些功能，这些功能的代码往往可能在多个类中出现，例如logger。这样造成代码的重复，增加了耦合。logger的改变影响所有使用它的类或方法。

而AOP在需要的类或方法上切下，前后的切入点可以加入增强的功能。让调用者和被调用者解耦。

这是一种不修改原来的业务代码，给程序动态添加功能的技术。例如logger函数功能就是对业务函数增加日志的，而业务函数中应该把与业务无关的日志功能剥离干净。

装饰器应用场景

日志、监控、权限、设计、参数检查、路由等处理。

这些功能与业务功能无关，很多业务都需要的公共功能，所以适合独立出来，需要的时候，对目标对象增强。

