Python|实用（常用）技术

> 分享一些在 Python 脚本开发中常用的技术和代码实现

知识点

方法解析顺序（MRO）

> 2.3 版本开始采用 C3 算法，并一直沿用
>
> 用于解决类继承体系下的方法搜索解析顺序

多继承体系下，Python 使用 MRO 来确定基类的搜索顺序，解决的是基类的搜索顺序问题，例如，如下代码：

```
class A():
    def func(self): pass

class B():
    def func(self): pass

class C():
    def func(self): pass

class M(A, B):
    def func(self): pass

class N(B, C):
    def func(self): pass

class X(M, N): pass

X().func() # 调用的是 ?
print(X.mro()) # print: [<class '__main__.X'>, <class '__main__.M'>, <class '__main__.A'>, <class '__main__.N'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
```

线性化：将树状的继承关系转变为线性关系，这就是 MRO 的目标

为了继续讲实现算法，先做一些约定：

C1 C2 ... Cn 表示这些类的列表

head 表示第一个类 C1

tail 表示后面的类 C2 C3 ... Cn

据此，算法可以描述为：**C 的线性化就是 C 加上父类的线性化和父类列表的合并的结果**，*即*:

`L[C(B1 ... BN)] = C + merge(L[B1], ..., L[BN], B1 ... BN)`

当 C 为 object 时，结果为自身：`L[object] = object`

合并算法 merge 的过程是：

1. 取出第一个列表的 head
2. 如果其未出现在后续列表的 tail 内，则将其加入 C 的线性化中，并从所有所有的列表中删除；否则，取出下一个列表，重复此步骤

注意，如果找不到一个合适的 head，Python 会引发异常拒绝创建这个类

对于上述的例子，计算为：

```
 1 L(M) = M + merge(L(A), L(B), AB)
 2       = M + merge(A, B, AB)
 3       = M + A + merge(B, B)
 4       = M + A + B = MAB
 5 L(N) = N + merge(L(B), L(C), BC)
 6       = N + merge(B, C, BC)
 7       = N + C + merge(C, C)
 8       = N + B + C = NBC
 9 L(X) = X + merge(L(M), L(N), MN)
10       = X + merge(MAB, NBC, MN)
11       = X + M + merge(AB, NBC, N)
12       = X + M + A + merge(B, NBC, N)
13       = X + M + A + N + merge(B, BC)
14       = X + M + A + N + B + C = XMANBC
```

调用过程

函数执行的流程大致如下：

1. 定义函数，创建一个函数对象并绑定到对应的名字上
2. 调用函数，首先找到名字所绑定的函数对象
3. 对于，成员方法而言，方法名会执行一个特殊的属性对象，这个属性对象拥有 `__self__` 和 `__func__` 两个只读属性，分别执行示例对象和函数对象
4. 创建一个函数栈帧对象，并压入参数执行
5. 执行完毕取出返回值

函数对象中包含了这个函数被调用时的信息，如以下几个可以修改的属性

`__code__`	已编译函数体的代码对象，当一个函数执行时如何在栈帧上展开就是通过它来实现
`__defaults__`	包含具有默认值的形参默认值所组成的 tuple
`__kwdefaults__`	包含仅限关键字形参默认值的 dict

如此，我们只需要修改函数对象的这几个属性，就能达到修改函数逻辑的目的：

1. 如果只需要修改代码，则修改 `code` 属性
2. 如果需要修改位置参数默认值，则修改 `defaults` 属性
3. 如果需要修改仅限关键字参数的默认值，则修改 `kwdefaults` 属性

下面是一些例子：

```python
def func_1():
    return 'Call func_1.'

def func_2():
    return 'Call func_2.'

def func_3(arg1, arg2):
    return f'Call func_3: {arg1}, {arg2}'

def func_4(arg1=1, arg2=2, *, kwarg1='kw1', kwarg2='kw2'):
    return f'Call func_4: {arg1}, {arg2}, {kwarg1}, {kwarg2}'

class Class():
    def func(self):
        return 'Call Class::func.'

def funccc(self):
    return 'Call funccc.'

if __name__ == '__main__':
    fn = func_1
    print(fn())  # print: Call func_1.

    # 修改代码对象
    func_1.__code__ = func_2.__code__
    print(fn())  # print: Call func_2.

    # 修改代码对象并加上形参默认值
    func_1.__code__ = func_3.__code__
```

```
    func_1.__defaults__ = (1, 2)
    print(fn())  # print: Call func_3: 1, 2

    # 修改代码对象并修改仅限位置参数形参默认值
    func_1.__code__ = func_4.__code__
    func_1.__kwdefaults__ = {'kwarg1': 'new_kw1', 'kwarg2': 'new_kw2'}
    print(fn())  # print: Call func_4: 1, 2, new_kw1, new_kw2

    c = Class()
    print(c.func())  # print: Call Class::func.
    c.func.__func__.__code__ = funccc.__code__
    print(c.func())  # print: Call funccc.
```

导入过程

先梳理几个概念

模块

- 模块是包含 Python 定义和语句的文件，其文件名为模块名加上 `.py` 后缀名，其中定义的名字可以被其他模块导入使用
- 在模块内部，通过全局变量 `__name__` 可以获取模块的名字，特别地，在命令行直接执行脚本时 `__name__` 属性会被赋值为字符串 `__main__`
- 模块中的可执行语句和各类定义语句会在第一次被导入时，或者直接运行时执行

包

- 包是模块的集合（也可以拥有子包是包的集合），分为常规包和命名空间包
- 常规包是一个带有 `__init__.py` 文件的目录，导入时，该文件会被执行
- 命名空间包不带 `__init__.py` 文件的目录，并且可以由多个同名的目录共同组成一个包

命名空间

- 命名空间是名字到对象的映射集合
- 作用域是命名空间中的名字所能生效的区域：
  - 最内层作用域，包含局部名称，通常是函数
  - 外层作用域，包括外层函数等 “非全局、非局部” 的名字
  - 全局作用域，包含当前模块的全局名称
  - 最外层的作用域，是内置名称的命名空间
- 关键字 `global` 重新绑定到全局名字，关键字 `nonlocal` 重新绑定到非局部的外层作用域中的名字（排除当前最内层的作用域逐步向外搜，搜到谁是谁）

在 Python 中导入模块（import module）时，会执行以下操作：

1. 先搜索缓存，如果没有缓存，就先去搜索目录（`sys.path`）下搜索对应文件
    a. 被命令行执行运行的文件所在的目录，或者没有指定文件时的当前目录
    b. PYTHONPATH 环境变量指定的目录列表
    c. 标准库模块及相关依赖模块所在目录
    d. 站点配置目录
        i. 由 `site` 模块管理，初始化 `sys.path` 时自动导入
        ii. 通常位于安装目录的 `lib` 目录下
        iii. 可以在该目录下配置 `name.pth` 文件，其中的每一行将视为其子目录被导入
2. 搜索到文件后执行，在执行之前会现在 `sys.modules` 中创建一个空项（避免循环导入），执行完后再赋值
3. 最后将导入的模块绑定到当前作用域中

虽然 Python 导入通常情况下不会产生循环导入的问题，但还是无法完全避免：

```
''' 循环导入引发的问题
.
├── bar.py
├── baz.py
└── test.py
'''
# ----------------------------
# file: bar.py
# ----------------------------
from baz import baz_arg
bar_arg = 'this is bar arg.'

# ----------------------------
# file: baz.py
# ----------------------------
from bar import bar_arg
baz_arg = 'this is baz arg.'

# ----------------------------
# file: test.py
# ----------------------------
import bar      # 报错: ImportError: cannot import name 'bar_arg'
                # 通常循环导入并不会有问题，因为上面说过，Python 在执行模块代码前会先在 sys.modules 中创建一个空项
```

```
                # 然而，像这个例子中的情况还是会报错，过程是：
                ## 先尝试导入 bar，创建了 bar 的模块缓存
                ## 然后执行 bar 的模块代码，尝试导入 baz
                ## 尝试导入 baz 时，也是先创建了 baz 的模块缓存，再执行模块代码
                ## 在执行 baz 的模块代码时，又尝试导入 bar，而此时缓存中已经有 bar 了则认为导入成功
                ## 然后回去 bar 的模块中导入名字 bar_arg，因为此时 bar 模块缓存还仅仅是个空项没有任务内容，所以抛错
```

便利函数

下面是一些常用的便利函数

```python

'''
迭代条件关系判断 all/any
'''
it = [True, False, True, False]
# 参数为一个可迭代对象
# all() 要求所有迭代中的条件都为真，any() 要求所有迭代中的条件至少有一个为真
if all(it): print('all true.')
if any(it): print('any true.')


'''
迭代和枚举值 enumerate
'''
it = ['a', 'b', 'c', 'd']
for _i,_v in enumerate(it): print(_i,_v)  # 0 a, 1 b, 2 c, 3 d


'''
产生一个（不可变的）序列 range
'''
for i in range(0, 8, 2): print(i)  # 0 2 4 6


'''
倒转一个序列 reversed
'''
for i in reversed(range(4)): print(i)  # 3 2 1 0


'''
过滤器 filter
```

```
'''
# 遍历传入的迭代器，调用传入的函数，使用返回真值的元素所组成的一个迭代器
for i in filter(lambda x: True, ['a', 'b', 'c']): print(i)  # a b c


'''
并行迭代 zip
'''
for i in zip([1,2,3], ['A', 'B', 'C'], ['a', 'b', 'c']): print(i)  # (1, 'A', 'a'), (1, 'B', 'b'), (1, 'C', 'c')


'''
高效迭代器 itertools
标准库 itertools 提供了非常多实用的迭代工具，这里举两个例子
'''
import itertools
# 迭代器链
for i in itertools.chain('ABC', 'DEF'): print(i)  # A B C D E F
# 累加
for i in itertools.accumulate([1, 2, 3, 4, 5]): print(i)  # 1 3 6 10 15
```

装饰器

装饰器用来装饰函数（类），一般有两个目的

一是修改函数行为，改变其表现

二是在定义过程中拿到函数对象，进行注册等

函数（类）定义的本质是创建一个函数（类）对象，让后绑定到当前作用域中，装饰器改变的是创建对象的方式：

```
def f(): pass
# 相当于 locals()['f'] = createObj(f)

@wrap
def g(): pass
# 相当于 locals()['g'] = wrap(createObj(g))

@wrap(param)
def h(): pass
# 相当于 locals()['h'] = wrap(param)(createObj(h))
```

下面是两个典型的例子：

```python
'''
修改行为表现
开发过程中，经常会有需要判断功能开关，以控制是否需要执行的逻辑（尤其是暴露给客户端的接口），此时可以通过装饰器来封装此类判断逻辑
'''
def checkGameconfigEnable(name):
    def f(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            enableCall = getattr(gameconfig, name, None)
            if not enableCall or not enableCall():
                WARNING_MSG('gameconfig not enable:', name)
                return
            return func(*args, **kwargs)
        return wrapper
    return f

# 开关 xxx 控制函数是否执行
@checkGameconfigEnable('xxx')
def xxxx(): pass

'''
修改函数注册
当有很多策略需要区别处理的时候，可以通过装饰器来注册各个策略的处理逻辑，避免写很多if...else...代码
'''
BATTLE_PASS_EVENT_MAP = {}
def battlePassEvent(eventName):
    def fwrap(f):
        BATTLE_PASS_EVENT_MAP[eventName] = f

        @functools.wraps(f)
        def _func(self, *args, **kwargs):
            return f(self, *args, **kwargs)
        return _func

    return fwrap

# 当有对应类型的事件，做出对应的处理
def onBattlePassEvent(self, eventName, args):
    # ...
    BATTLE_PASS_EVENT_MAP[eventName](self, *args)

# 注册对应的处理方法
@battlePassEvent(gameconst.BattlePassEventName.BP_LOGIN)
def _onBPEventLogin(self): pass
```

迭代器和生成器

迭代器？实现了 `__next__` 方法和 `__iter__` 方法的是对象

生成器？使用 `yield` 代替 `return` 返回结果的方法

迭代器主要用于灵活地对数据结构进行遍历，而生成器用于自动产生迭代器，下面是一个典型的例子：

```
# 分批执行
def batchlyCall(self, iterableCall, batchNum, interval=0.5, callback=None):
    it = iter(iterableCall)
    for _ in range(batchNum):
        callObj = next(it, None)
        if callObj is None:
            callback and callback()
            return

        callObj()

    self._callback(interval, 'batchlyCall', (it, batchNum, interval, callback), gametimer.TIMER_TAG_BATCHLY_CALL)

# 生成器，调用生成器函数会自动生成一个迭代器
def _iterAvatar():
    for gbId, entId in gameglobal.roleGBIDToEntId.items():
        a = KBEngine.entities.get(entId)
        if a and not a.isDestroyed:
            yield lambda :func(a)

self.batchlyCall(_iterAvatar(), 30)
```

上下文管理器

使用上下文管理器更优雅地进行资源管理

什么是资源管理器？实现了上下文管理协议 `__enter__()` 和 `__exit__()` 的对象

下面是几个例子：

```
'''
使用 with 管理文件资源
'''
```

```python
with open('example.txt', 'rb') as inFile:
    pass

'''
自定义上下文管理器
'''
class TestManager():
    def __repr__(self):
        return 'This is TestManager obj.'
    # 进入上下文时执行，返回值会被绑定到 as 指定的名字上
    # 通常的做法是返回自身，如打开文件的 open() 函数
    def __enter__(self):
        print('Enter manager.')
        return self
    # 退出上下文时执行，如果是语句体内的异常引发的退出则会传入异常信息，否则传入 None 值
    # 对于异常引发的退出，如果此函数返回真值，则不会继续传播异常并继续执行 with 语句体后面的代码
    # 相对地，如果返回假值，则会继续向外传播异常
    def __exit__(self, exc_type, exc_val, exc_tb):
        print('Deal error in manager:', exc_type, exc_val, exc_tb)
        print('Exit manager.')
        return True
with TestManager() as tm:
    print(tm)       # print:
                    # Enter manager.
                    # This is TestManager obj.
                    # Deal error in manager: None None None
                    # Exit manager.

'''
使用 contextlib.contextmanager 创建上下文管理器
'''
import contextlib
def open_resource(*args, **kwargs):
    print('Open.')
    res = (args, kwargs)
    return res

def close_resource(resource):
    print('Close.')
    del resource
@contextlib.contextmanager
def managed(*args, **kwargs):
    # 先执行 yield 前的代码
    resource = open_resource(*args, **kwargs)
    try:
        # yield 出去的值相当于 __enter__ 的返回值
```

```
        yield resource
        # 执行完 with 语句体后会继续执行后面的代码，即使在语句体中触发异常
    finally:
        close_resource(resource)
with managed(1, 2, 3, a='1', b='b') as t:    # print:
    print(t)                                 # Open.
    print('Do something.')                   # ((1, 2, 3), {'a': '1', 'b': 'b'})
                                             # Do something.
                                             # Close.
```