4 python元编程

- Everything is an Object in Python
- 元编程定义
- metaclass
- descriptor
- inspector

4.1 Everything is an Object in Python

python里常用的类型：

- module
- class
- type
- object
- function

在python里，将上述所有类型看成对象，因为它们都和普通对象一样，含有属性(attributes)和方法(methods)。

4.1 Everything is an Object in Python

从python源码角度来看:

methods

```c
typedef struct {
    PyObject_HEAD
    PyObject *im_func;   /* The callable object implementing the method */
    PyObject *im_self;   /* The instance it is bound to */
    PyObject *im_weakreflist; /* List of weak references */
    vectorcallfunc vectorcall;
} PyMethodObject;
```

modules

```c
typedef struct {
    PyObject_HEAD
    PyObject *md_dict;
    struct PyModuleDef *md_def;
    void *md_state;
    PyObject *md_weaklist;
    // for logging purposes after md_dict is cleared
    PyObject *md_name;
} PyModuleObject;
```

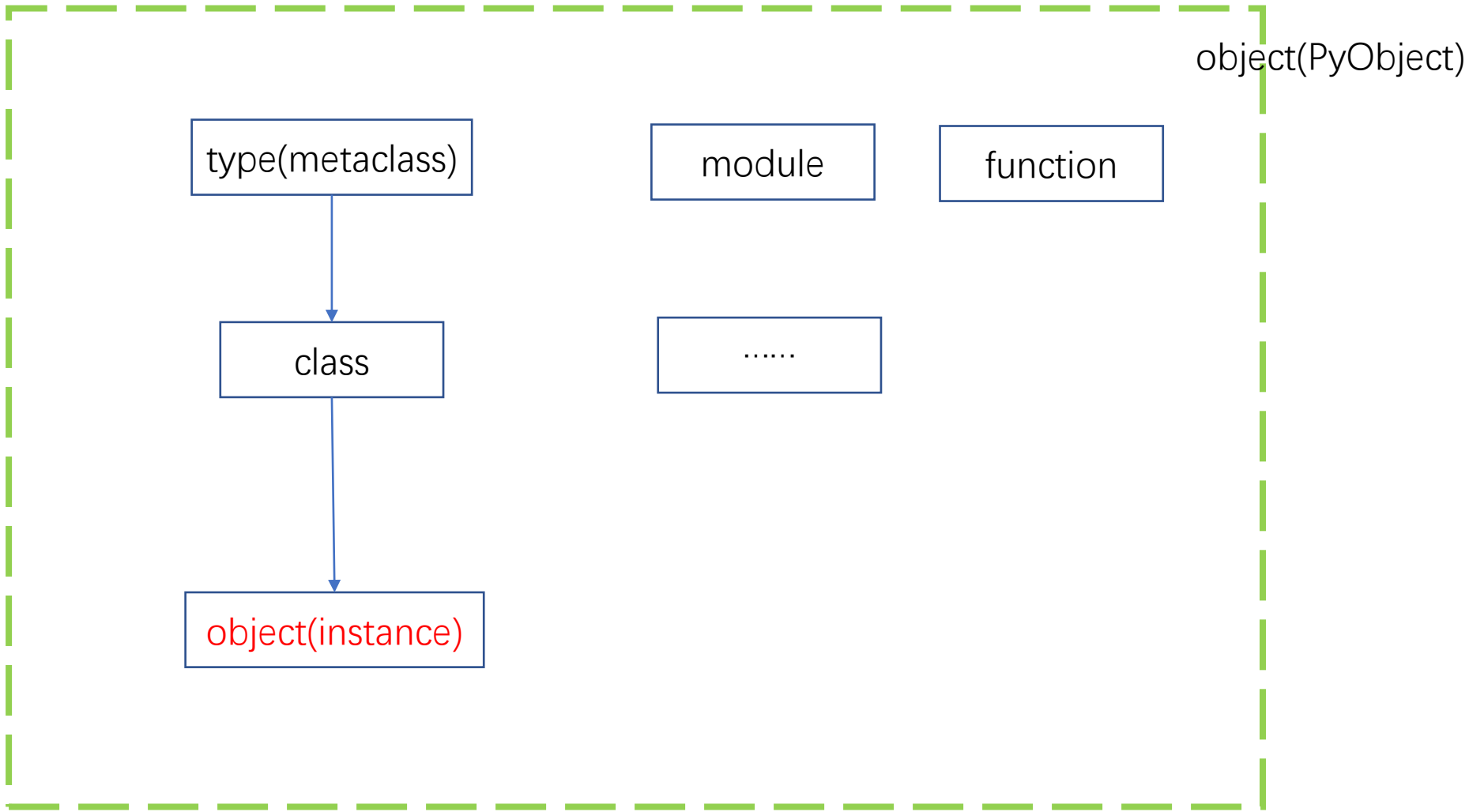
4.1 Everything is an Object in Python

从python源码角度来看:

```
static int
module_init_dict(PyModuleObject *mod, PyObject *md_dict,
                 PyObject *name, PyObject *doc)
{
    _Py_IDENTIFIER(__package__);
    _Py_IDENTIFIER(__loader__);

    if (md_dict == NULL)
```

```
}

static PyObject *
method_richcompare(PyObject *self, PyObject *other, int op)
{
    PyMethodObject *a, *b;
    PyObject *res;
    int eq;

    if ((op != Py_EQ && op != Py_NE) ||
        !PyMethod_Check(self) ||
        !PyMethod_Check(other))
    {
        Py_RETURN_NOTIMPLEMENTED;
    }
    a = (PyMethodObject *)self;
    b = (PyMethodObject *)other;
```

4.1 Everything is an Object in Python



4.2 元编程定义

元数据(meta data):用于描述数据的数据

元编程(meta programming):用于操作代码的程序。例如:

- 用于生成代码的程序
- 读取代码
- 分析代码
- 转换代码
- 修改代码
- ……

4.3 metaclass

既然class在python里是一个object,那么也可以像创建普通的object一样动态创建class。

```
class NormalClass:
    FLAG = 1

    def __init__(self, name):
        self.name = name

    def hello(self):
        print(f"NormalClass, name={self.name}, flag={self.FLAG}")
```

4.3 metaclass

NEW_CLASS = **type**(*class_name, bases, dict, **kwds*)

class_name : 类名
bases : 继承的父类
dict : 设置到创建的新类__dict__的内容，可以是属性或者方法

meta_1.py

4.3 metaclass

metaclass :

1. 继承type
2. 重载__new__函数
3. 在类定义时传入metaclass=参数

meta_2.py

问题：如何使用metaclass实现单例模式(singleton)?

单例模式：一个类型(class)的实例最多只能存在一个。

问题：如何使用metaclass实现单例模式(singleton)?

单例模式：一个类型(class)的示例最多只能存在一个。

第一版实现:

```python
class A:
    _ins = None

    def __init__(self):
        print('create new instance')


    @classmethod
    def get(cls):
        if cls._ins is None:
            cls._ins = cls()

        return cls._ins
```

引入了额外的接口
没有屏蔽掉构造功能，还是可以进行实例化

singleton.py

4.3 metaclass

第二版实现: 虚拟机中创建一个instance的流程

__new__

__init__

```
    if (type->tp_new == NULL) {
        _PyErr_Format(tstate, PyExc_TypeError,
                      "cannot create '%s' instances", type->tp_name);
        return NULL;
    }

    obj = type->tp_new(type, args, kwds);
    obj = _Py_CheckFunctionResult(tstate, (PyObject*)type, obj, NULL);
    if (obj == NULL)
        return NULL;

    /* If the returned object is not an instance of type,
       it won't be initialized. */
    if (!PyType_IsSubtype(Py_TYPE(obj), type))
        return obj;

    type = Py_TYPE(obj);
    if (type->tp_init != NULL) {
        int res = type->tp_init(obj, args, kwds);
        if (res < 0) {
            assert(_PyErr_Occurred(tstate));
            Py_DECREF(obj);
            obj = NULL;
        }
        else {
            assert(!_PyErr_Occurred(tstate));
        }
    }
    return obj;
}
```

4.3 metaclass

第二版实现: 使用metaclass的方式，简单粗暴的屏蔽掉__new__, __init__

singleton_2.py

对原逻辑有一定的破坏

4 python元编程

- Everything is Object in Python
- 元编程定义
- metaclass
- inspector
- descriptor

4.4 inspect

- **inspect模块**提供了多个有用的函数，来帮助我们获取运行时中模块、类、methods、traceback等各种对象的信息，包括函数参数信息、文件注释、源码、对应行号等信息。

- **inspect.getmembers**(*object, [filter]*) 获取目标对象的成员

- **inspect.ismethod**(*o*)、**inspect.isclass**(*o*), **inspect.ismodule**(*o*) ……各类方法用于判断对象是否属于某个类型，也可以用于getmembers的filter

inspect_1.py

4.4 inspect

问题：如何使用metaclass、inspect等工具实现组件模式来代替继承，避免mro带来的性能损失？

mro的规则确定的，我们是否能够按照一定的规则，使这个动态计算的过程变成一个静态查询的过程？

component.py

4 python元编程

- Everything is Object in Python
- 元编程定义
- metaclass
- inspector
- 特殊方法名

4.5 特殊方法名

在python中，我们可以通过定义带有特殊名字的方法来重载操作符。与c++中operator类似。

1. 自定义属性访问
2. 自定义模块的属性访问
3. 描述符Descriptors
4. 自定义类创建
5. 模拟容器对象
6. 模拟数值对象
7. 实现with 上下文管理器
8. 模拟可调用对象
9. ……

更多详细内容可以阅读python手册 Special method names相关内容

自定义属性访问

- **object.__getattribute__**(*self, name*)当访问名为name的属性，会首先调用该方法
- **object.__getattr__**(*self, name*) 当访问名为name的属性，但是属性不存在时候，会调用该方法
- **object.__setattr__**(*self, name, value*) 当给名为name的属性赋值时，会调用该方法
- **object.__delattr__**(*self, name*) 当删除名为name的属性时，会调用该方法

需要注意无穷递归的问题

smn_1.py

描述符Descriptors

如果一个对象实现了__get__ , __set__, __del__这些方法，那么就称为描述符。它定义（描述）了属性的读取、设置、删除等操作。

- **object.__set_name__**(*self, owner, name*): 定义该类所定义的属性时，调用的方法
- **object.__get__**(*self, obj, objtype*): 读取该类所定义的属性时，调用的方法
- **object.__set__**(*self, obj, value*): 设置该类所定义的属性时，调用的方法
- **object.__delete__**(*self, instance, value*): 删除该类所定义的属性时，调用的方法

描述符Descriptors

```python
class ThisIsADescriptor:
    def __set_name__(self, owner, name):
        print(f'descriptor.__set_name__ is called. owner={owner}, name={name}')
        self.private_name = '_' + name

    # 为什么使用的是private_name
    def __get__(self, obj, objtype=None):
        print(f'descriptor.__get__ is called. self={self}, obj={obj}, objtype={objtype}')
        return getattr(obj, self.private_name)

    def __set__(self, obj, value):
        print(f'descriptor.__set__ is called. self={self}, obj={obj}, value={value}')
        setattr(obj, self.private_name, value)
```

smn_descriptor.py

描述符Descriptors常用情况：

- 只读属性
- 属性值验证validator
- ORM
- ……

smn_descriptor_2.py && smn_descriptor_orm.py

模拟可调用对象：

- **object.__call__**(*self, *args, **kwargs*): 可以使该对象如函数一般可调用(callable)

  Everything is an Object in Python

  singleton_3.py