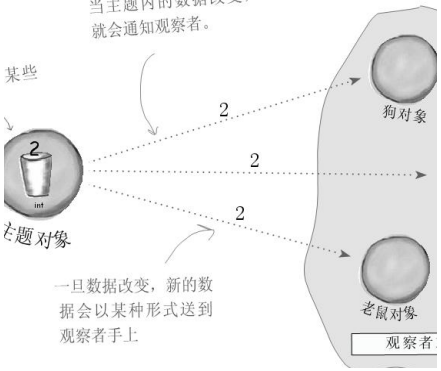


论 Python 与设计模式

赖勇浩（http://laiyonghao.com）

2013-12-8

珠海

设计模式？

自我介绍

- 赖勇浩
- 从业 8 年多，主要编程语言是 Python
- game -> webgame -> web
- 常在珠三角技术沙龙出没
- http://laiyonghao.com

PyCon 的老朋友


CHINA
PyCon 2011
Python 于 web-game 的应用
赖勇浩（http://laiyonghao.com）
2011.12.04
1


PyCon 2012
CHINA
页游开发中的 Python 组件与模式
赖勇浩（http://laiyonghao.com）
2012-10-21
上海

+

好，正式开始吧！

先热热场子……

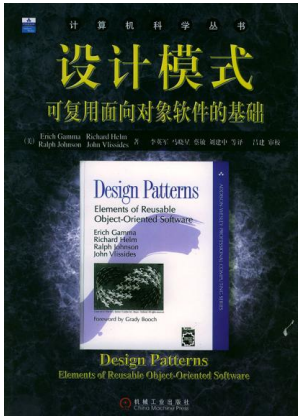
- 写 Python 代码赚钱的，有没有？

先热热场子……

- 写 Python 代码赚钱的，有没有？
- 写 Python 超过 1 年的，有没有？

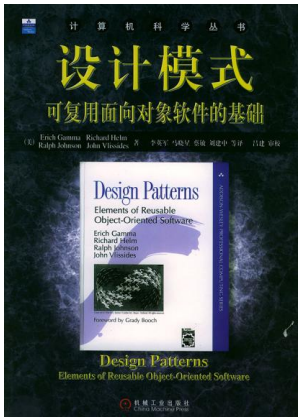
再热热场子……

- 读过《设计模式—可复用面向对象软件的基础》这本书的有没有？

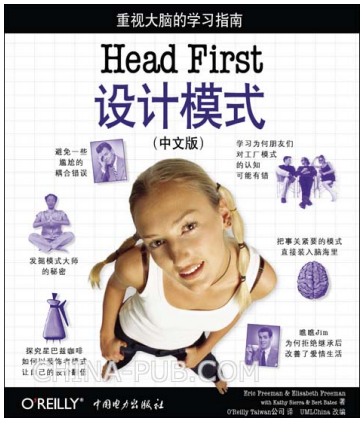


再热热场子……

- 读过《设计模式—可复用面向对象软件的基础》这本书的有没有？



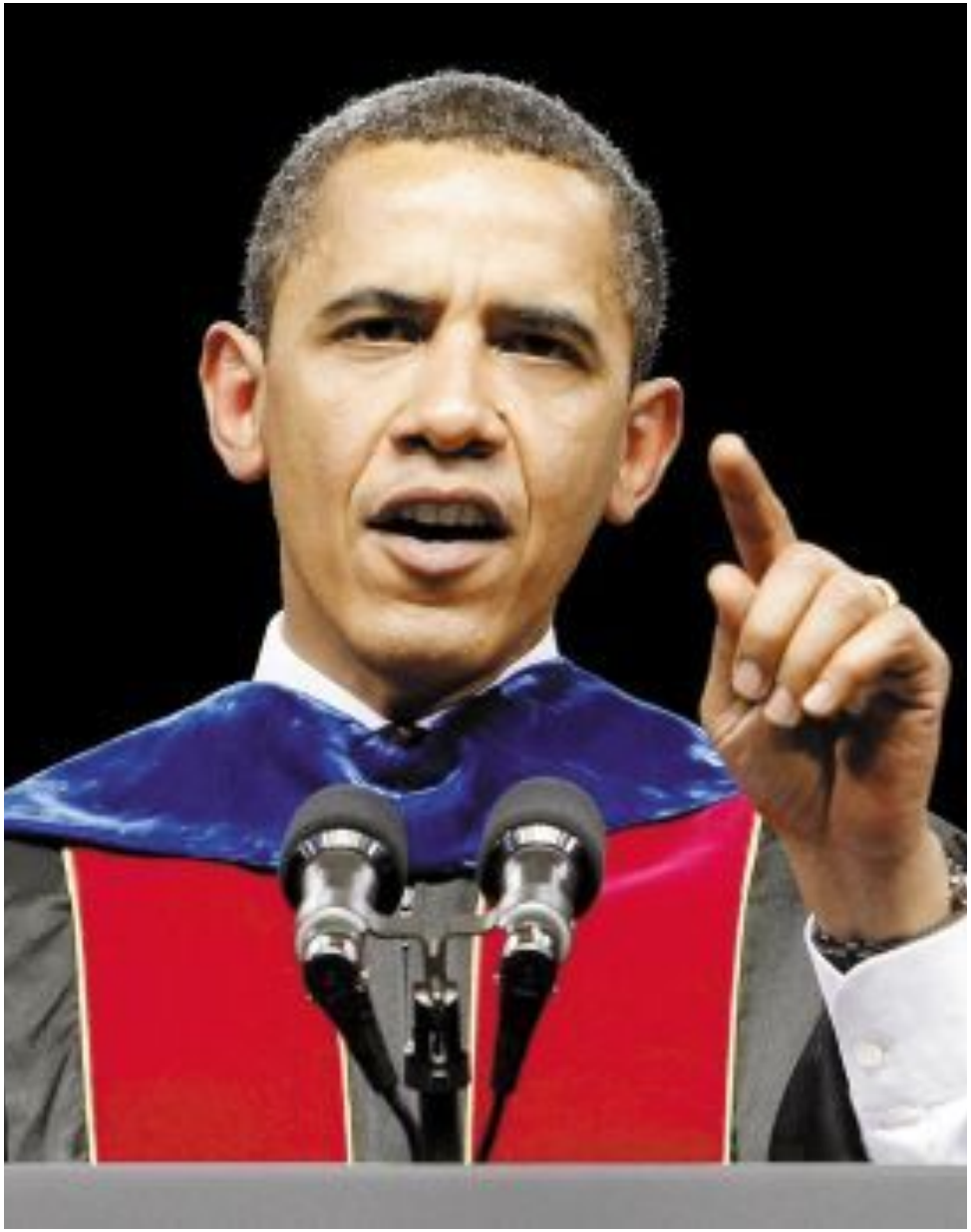
- 读过《Head First 设计模式》的有没有？



一个观点

我也希望是这样……

但事实是……



先看事实：工厂函数（方法）

```
int(…)

float(…)

type(name, bases, dict)
  >>> class X(object):
  ...     a = 1
  ...
  >>> X = type('X',
  (object,), dict(a=1))

collections.namedtuple()
```

```
datatime.fromtimestamp(…)

Decimal.from_float(…)

Fraction.from_float(…)

Fraction.from_decimal(…)
```

先看事实：享元（FlyWeight）

```
>>> i = 10
>>> j = 5 + 5
>>> id(i)
140479275503872
>>> id(j)
140479275503872
>>> a = 'ab'
>>> b = 'a' + 'b'
>>> id(a) == id(b)
True
```

先看事实：享元（FlyWeight）

```
>>> i = 10

>>> j = 5 + 5

>>> id(i)

140479275503872

>>> id(j)

140479275503872

>>> a = 'ab'

>>> b = 'a' + 'b'

>>> id(a) == id(b)

True
```

```
>>> a = a * 10

>>> intern(a)

'ababababababababab'

>>> b = 'ababababababababab’

>>> c = 'abababababababababa' + 'b’

>>> d = 'ab' * 10

>>> id(a) == id(b) == id(c) ==
id(d)

True
```

先看事实：适配器

```
import SocketServer

class ThreadedTCPServer(SocketServer.ThreadingMixIn,
SocketServer.TCPServer):

    pass
```

先看事实：代理模式

```
>>> import weakref
>>> class A(object):pass
...
>>> a = A()
>>> a.attr = 1
>>> a.attr
1
```

```
>>> a1 = weakref.proxy(a)
>>> a1.attr
1
>>> a1
<weakproxy at 0x10dc375d0
to A at 0x10dc3a410>
```

先看事实：模板方法

```
import SocketServer

class MyTCPHandler(SocketServer.BaseRequestHandler):

    def handle(self):

        self.data = self.request.recv(1024).strip()

        print "{}
wrote:".format(self.client_address[0])

        print self.data

        self.request.sendall(self.data.upper())
```

先看事实：模板方法

```
From abc import ABCMeta
class C:
    __metaclass__ = ABCMeta
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
```

所以，真相是……

- 标准库都在用……
- 标准库都推荐用……
- 怎么可以说不需要？！

所以我们不是不需要模式……

而是要 Pythonic 的模式实现……

不 Pythonic 的设计模式：单例

```
class Singleton(object):
    def __new__(cls, *args, **kw):
        if not hasattr(cls, '_instance'):
            orig = super(Singleton, cls)
            cls._instance = orig.__new__(cls, *args, **kw)
        return cls._instance
```

单例的三个需求：

- 只能有一个实例；
- 它必须自行创建这个实例；
- 它必须自行向整个系统提供这个实例。

单例遇上并行

```
class Singleton(object):
    objs  = {}
    objs_locker =  threading.Lock()
    def __new__(cls, *args, **kv):
        if cls in cls.objs:
            return cls.objs[cls]
```

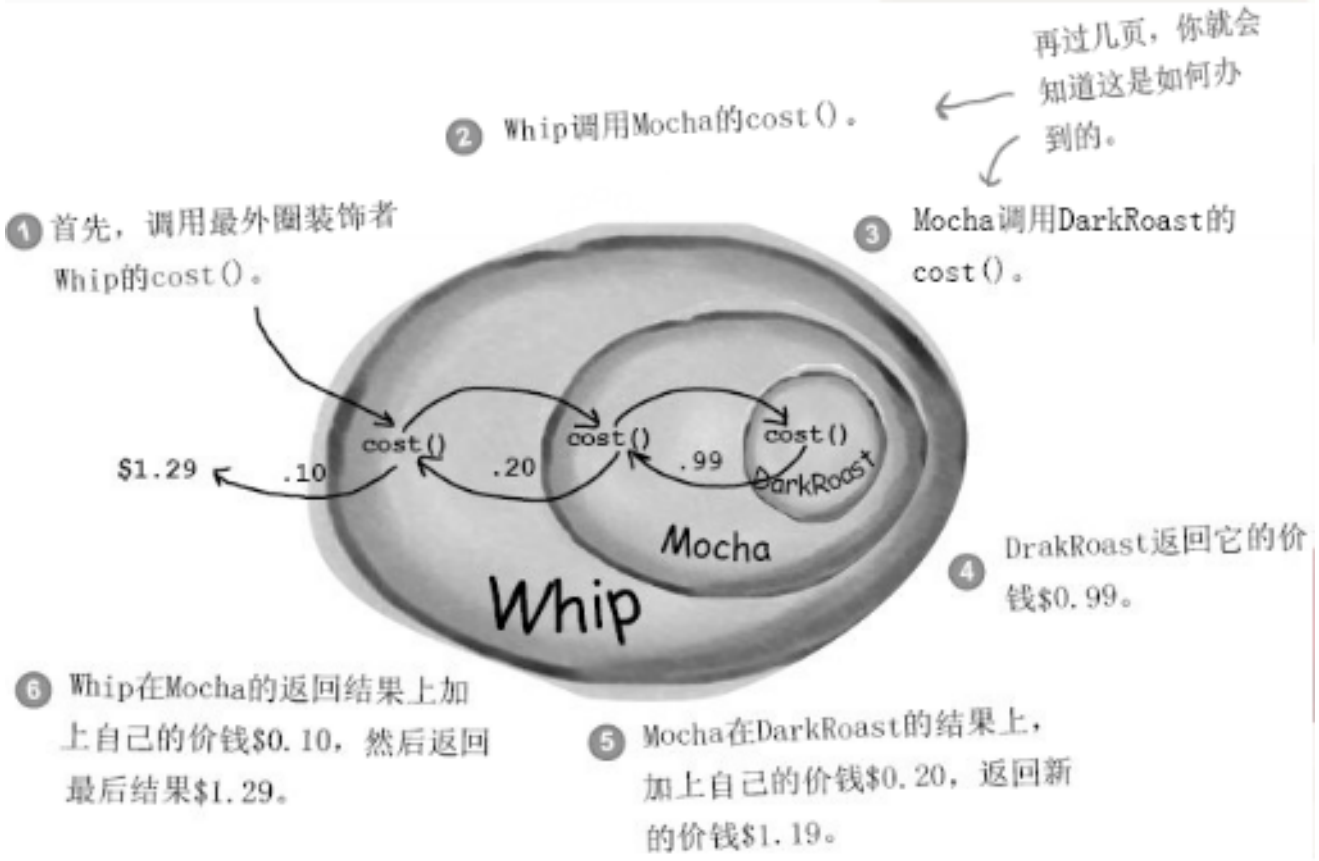
单例遇上并行

```
cls.objs_locker.acquire()
try:
    if cls in cls.objs:
        return cls.objs[cls]
    cls.objs[cls] = object.__new__(cls)
finally:
    cls.objs_locker.release()
```

Pythonic 的设计模式：单例

- 重新审视 Python 的语法元素，尽量利用已有基础设施。
- 模块
  - 所有的变量都会绑定到模块；
  - 模块只初始化一次；
  - import 机制是线程安全的（保证了在并发状态下模块也只有一个实例）；
- 惯用法：
  - 文件名首字母大写，如 Moon.py
  - __all__

不 Pythonic 的设计模式：装饰器



代码大概是这样的

```
class darkroast(Beverage):

  def cost(self):return 0.99

…

class Whip(Beverage):
  def __init__(self, beverage):
    self._beverage = beverage

  def cost(self):
    return self._beverage.cost() + 0.1

print Whip(Mocha(darkroast())).cost()
```

Pythonic 的设计模式：装饰器

```
def beverage(cost):
  def _(orig = 0.0):
    return orig + cost
  return _
darkroast = beverage(0.99)
whip = beverage(0.1)
mocha = beverage(0.2)
print whip(mocha(darkroast()))
```

其它设计模式：Borg(MonoState)

- 保持对象的唯一性并不重要，只要共享状态就行
- Alex Martelli
- http://code.activestate.com/recipes/66531/

其它设计模式：Borg(MonoState)

```
class Borg(object):
    _state = {}
    def __new__(cls, *args, **kw):
        ob = super(Borg, cls).__new__(cls, *args, **kw)
        ob.__dict__ = cls._state
        return ob
```

动态语言特有的设计模式：mixin

- 动态地改变实例的类型的基类，在不修改生成实例过程的情况下，给实例增加（改变）功能。可用以实现插件框架。

```
class Bird(object):pass

bird = Bird()

class FlyMixin:

  def fly(self):print 'I can fly.'

bird.__class__.__bases__ += (FlyMixin, )

bird.fly()
```

动态语言特有的模式：duck typing

- 一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。
- 当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。（James Whitcomb Riley）
- 干掉模板方法？
  - 不，模板方法是想要保证子类实现接口

利用设计模式提供更好的接口

- getopt
- optparse
- argparse
- docopt

- Command-line interface description language
- define interface for your command-line app, and
- automatically generate parser for it.
- 解释器模式

docopt

```
Naval Fate.

Usage:
  naval_fate ship new <name>...
  naval_fate ship <name> move <x> <y> [--speed=<kn>]
  naval_fate ship shoot <x> <y>
  naval_fate mine (set|remove) <x> <y> [--moored|--drifting]
  naval_fate -h | --help
  naval_fate --version

Options:
  -h --help     Show this screen.
  --version     Show version.
  --speed=<kn>  Speed in knots [default: 10].
  --moored      Moored (anchored) mine.
  --drifting    Drifting mine.
```

解释器模式的应用

- GM 指令

```
player 0 money 10000
player 0 attack 10000
monster 0 die
scene monsters die
```

- 过场剧情脚本

```
monster 0 spawn 0,0
monster 0 moveto 0,-10
monster 0 attack
monster 0 moveto 0,0
```

Q&A

mail@laiyonghao.com