老男孩教育
oldboyedu.com

# 封装，多态

讲师：尚玉杰

老男孩教育
oldboyedu.com

# 本章目录

- 封装性
- 多态性
- 常见设计模式

# 封装

- 封装”就是将抽象得到的数据和行为相结合，形成一个有机的整体
  - 元组，列表，字典等等：数据的封装，通过引用去使用数据
  - 函数：算法的封装
    - 没有函数，功能靠每一行代码去直接执行
    - 耦合度太高，复用性太差，开发效率太低

# 封装

- 封装的目的是简化编程和增强安全性
  - 使用者不必关心该类具体的实现细节
  - 通过接口（万能的点）
  - 还可以给予的特定的访问权限来使用类的成员

  - 明确区分内外：
    - 类的实现者可以修改内部封装的东西而不影响外部调用者
    - 外部调用者只需要知道自己可以使用该类对象的哪些功能

# 封装

- 私有属性，私有方法
  - “ 双下划线 ” 开始的是私有成员，在类外部不可以直接用属性或方法名调用，子类中也不能访问到这个数据
  - 可以提供外界访问的接口
    - 将不需要对外提供的内容都隐藏起来
    - 把属性都隐藏，提供公共方法对其访问
  - 双下滑线开头的属性在继承给子类时，子类是无法覆盖的

# 封装

- 破解私有属性和私有方法：
  - 在名称前加上 _类名，即 _类名__名称（例如a._A__N）
  - 其实加双下划线仅仅是一种变形操作
    - 类中所有双下划线开头的名称如__x都会自动变形成：_类名__x的形式

# 多态/多态性

- 多态体现1：python是一种多态语言，不关心对象的类型
  - 对于弱类型的语言来说，变量并没有声明类型，因此同一个变量完全可以在不同的时间引用不同的对象
  - 毫无疑问，在python中对象是一块内存，内存中除了包含属性、方法之外，还包含了对象类型，我们通过引用来访问对象，比如a＝A()，首先python创建一个对象A，然后声明一个变量a，再将变量a与对象A联系起来。变量a是没有类型的，它的类型取决于其关联的对象

# 多态/多态性

```
class Bird:
    def move(self, field):
        print('鸟在%s上自由地飞翔' % field)
class Dog:
    def move(self, field):
        print('狗在%s里飞快的奔跑' % field)
x = Bird()
x.move('天空')
x = Dog()
x.move('草地')
#同一个变量 x 在执行同一个 move() 方法时，由于 x 指向的对象不同，因此呈现出不同的行为特征，这就是多态
```

# 多态

- 多态体现2：一类事物有多种形态（polymorphic）
  - 一个抽象类有多个子类，但方法实现不同
  - 例如：动物类有多个子类，每个子类都重新实现了父类的某个方法，但方法的实现不同（休息的方法）
  - 此时需要有继承，需要有方法重写

# 多态

- 多态体现2：一类事物有多种形态（polymorphic）

```
class Girl(object):
    def __init__(self, name):
        self.name = name
    def game(self):
        print("%s 蹦蹦跳跳的玩耍..." % self.name)
class Nurse(Girl):
    def game(self):
        print("%s 脱裤子打针" % self.name)

class Boy(object):
    def __init__(self, name):
        self.name = name
    def game_with_girl(self, girl):
        print("%s 和 %s 快乐的玩耍..." % (self.name, girl.name))
        girl.game()
```

# 多态/多态性

- 多态性polymorphism：在多态基础上，定义统一的接口（类外定义单独的函数）
  - 不同类的对象作为函数的参数时，得到的结果不同

# 鸭子类型

- python崇尚鸭子类型
  - 不关心类型，不需要继承，只关注方法实现，这种情况被称为鸭子类型
  - “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。
  - 在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的

```
class Person:
    def swim(self):
        pass
class Duck:
    def swim(self):
        pass
def swimming(arg):  #在我眼里会游泳的都一样
    arg.swim()
```

# 多态/多态性

- 总结：Python本身就是支持多态性的
  - 增加了程序的灵活性（通用性），以不变应万变，不论对象千变万化，使用者都是同一种形式去调用
  - 增加了程序的可扩展性，通过继承某个类创建了一个新的类，接口使用者无需更改自己的代码，还是用原方法调用

# 多态/多态性

- 对比：
  - 多态强调的是：一类事物 不同的形态
  - 多态性强调的是：同一操作，作用对象不同，表现出不同实现方式（只关心行为结果）
- 练习：
  - 创建汽车类（Car）含实例属性颜色red,对象方法run，功能是打印XX颜色小汽车在跑。
  - 创建猫类（Cat）含实例属性名字name,对象方法run，功能是打印猫咪XX在跑。
  - 实例化汽车类颜色为红色，实例化猫类，使用公共函数调用对象方法

# 设计模式

Alice

麦辣鸡腿堡
薯条
可乐

Jack

A套餐
麦辣鸡腿堡
薯条
可乐


创建套餐是提高点餐效率的可重用解决方案。

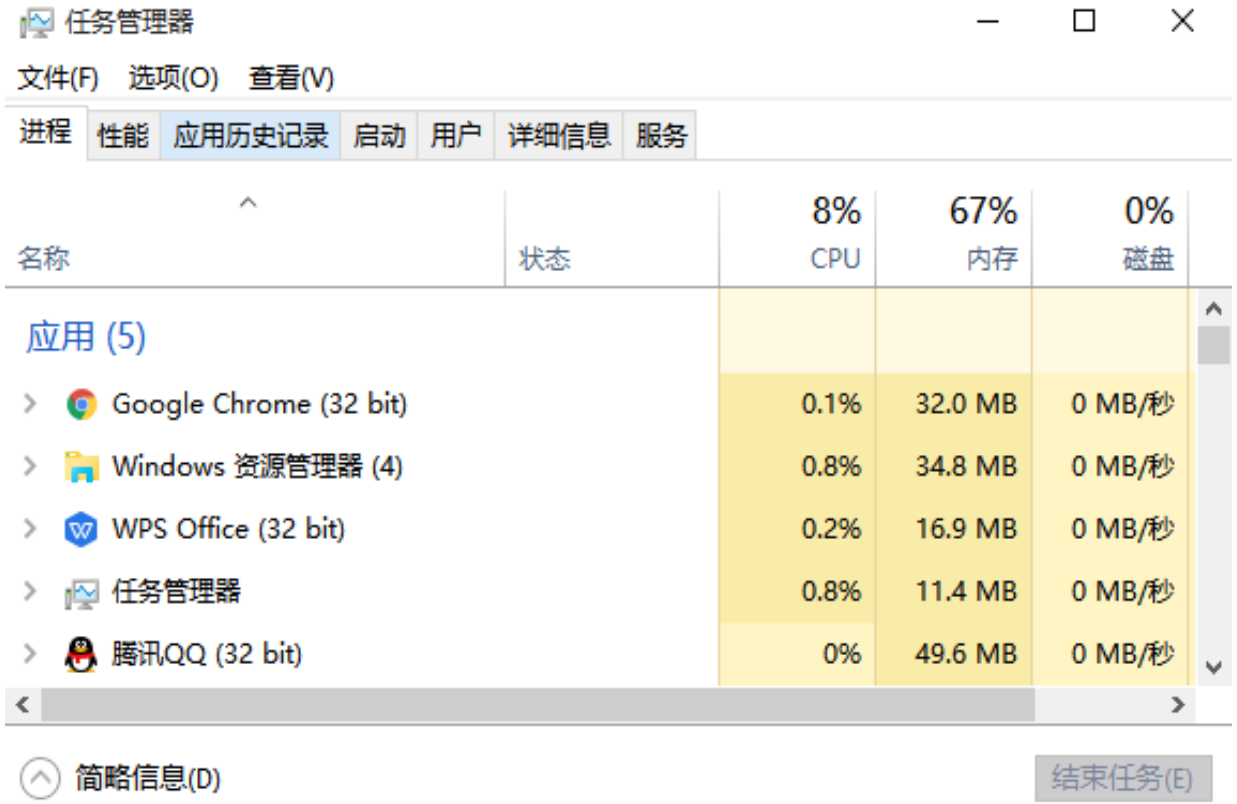
- 套餐可重复被点
- 提高了顾客和服务员之间的沟通效率

# 设计模式

- 软件世界本没有设计模式，用的人多了也便总结出了设计模式
  - 对于同一情境，众多软件开发人员经过长时间总结出的最佳可重用解决方案

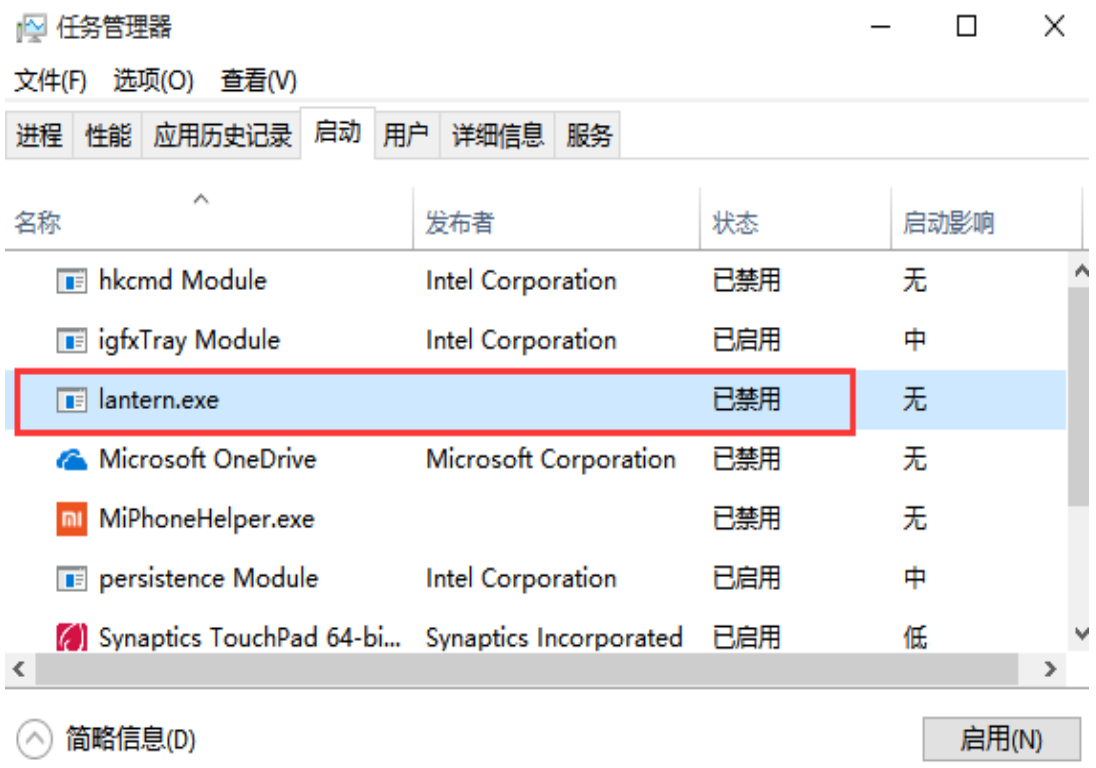
# 单例模式

- 计算机中的单例模式



# 单例模式

- 非单例的情况


任务管理器
文件(F) 选项(O) 查看(V)
进程 性能 应用历史记录 启动 用户 详细信息 服务
名称 发布者 状态 启动影响
hkcmd Module Intel Corporation 已禁用 无
igfxTray Module Intel Corporation 已启用 中
lantern.exe 已禁用 无
Microsoft OneDrive Microsoft Corporation 已禁用 无
MiPhoneHelper.exe 已禁用 无
persistence Module Intel Corporation 已启用 中
Synaptics TouchPad 64-bi... Synaptics Incorporated 已启用 低
简略信息(D)
启用(N)


任务管理器
文件(F) 选项(O) 查看(V)
进程 性能 应用历史记录 启动 用户 详细信息 服务
名称 发布者 状态 启动影响
hkcmd Module Intel Corporation 已禁用 无
igfxTray Module Intel Corporation 已启用 中
lantern.exe 已启用 无
Microsoft OneDrive Microsoft Corporation 已禁用 无
MiPhoneHelper.exe 已禁用 无
persistence Module Intel Corporation 已启用 中
Synaptics TouchPad 64-bi... Synaptics Incorporated 已启用 低
简略信息(D)
禁用(A)

# 单例模式

- 有时通过单一对象方便集中管理资源
  - 单例模式是保证一个类仅有一个实例的设计模式
  - 保证了在程序的不同位置都可以且仅可以取到同一个对象实例：如果实例不存在，会创建一个实例；如果已存在就会返回这个实例。

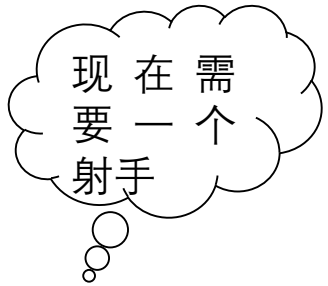
# 单例模式

```
class Singleton(object):
    instance= None
    def __new__(cls, *args, **kwargs):
        if not cls.instance:
            cls.instance = object.__new__(cls)
        return cls.instance

s1 = Singleton()
s2 = Singleton()

print("s1的地址：{}, s2的地址：{}".format(id(s1), id(s2)))
```

# 单例模式

```
class Mother(Singleton):
    def __init__(self, msg = ""):
        self.msg = msg
    def get_food(self, new_food):
        self.msg += new_food
    def food(self):
        print('做菜: ', self.msg)
mother1 = Mother()
mother2 = Mother()
mother1.get_food('西红柿')
mother2.get_food('鸡蛋')
print('儿子的妈妈id: ', id(mother1))
mother1.food()
print('女儿的妈妈id: ', id(mother2))
mother2.food()
```

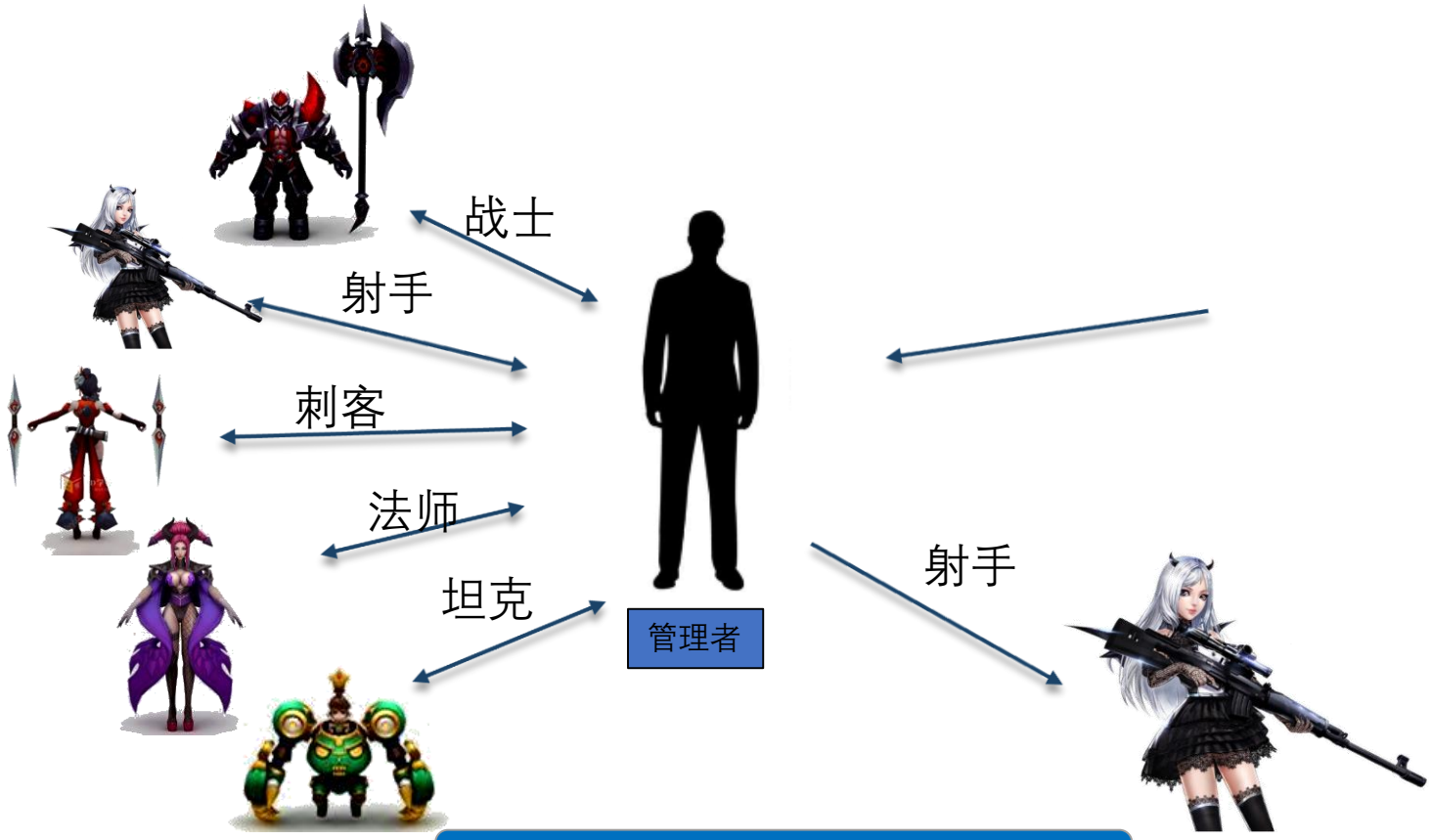
# 工厂模式


需要遍历整个人物类后，确定找到射手类后再创建人物

# 工厂模式

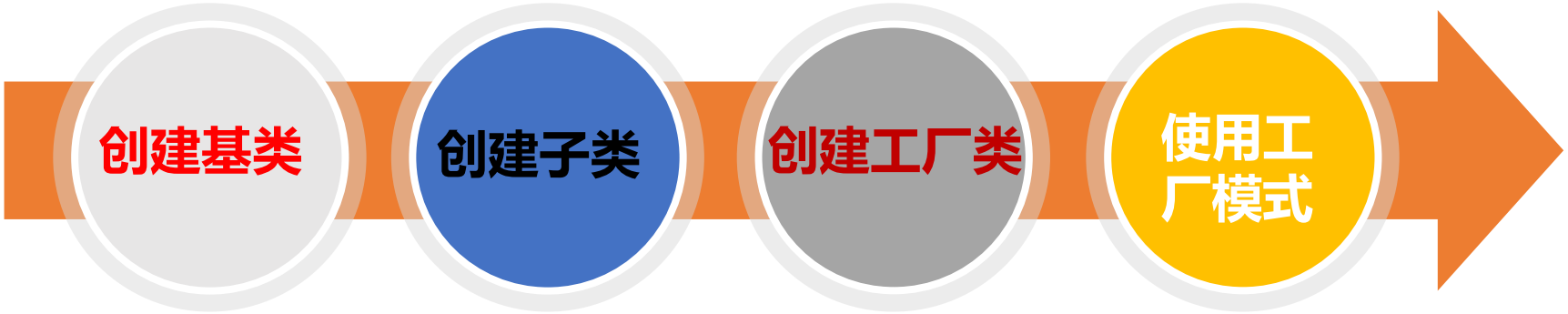
## 为什么使用工厂模式




通过一个工厂类方便快速创建所需人物

# 工厂模式

- 工厂模式是不直接暴露对象创建细节，而是通过一个共用类创建对象的设计模式，需要4步



# 工厂模式

- 创建基类

```
class Person(object):
    def __init__(self, name):
        self.name = name
    def get_name(self):
        return self.name
```

# 工厂模式

- 创建子类

```
class Male(Person):
    def __str__(self):
        return "Hello Mr." + self.get_name()

class Female(Person):
    def __str__(self):
        return "Hello Miss." + self.get_name()
```

# 工厂模式

- 创建工厂类

```
class Factory(object):
    def get_person(self, name, gender='M'):
        if gender == 'M':
            return Male(name)
        if gender == 'F':
            return Female(name)
```

# 工厂模式

- 使用工厂模式

```
factory = Factory()
person = factory.get_person("Bob", "M")
print(person)
```