用Python实现设计模式——工厂模式

segmentfault.com/a/1190000013053013

python 设计模式 工厂模式 更新于 2018-01-30 约 9 分钟

前言

工厂模式，顾名思义就是我们可以通过一个指定的“工厂”获得需要的“产品”，在设计模式中主要用于抽象对象的创建过程，让用户可以指定自己想要的对象而不必关心对象的实例化过程。这样做的好处是用户只需通过固定的接口而不是直接去调用类的实例化方法来获得一个对象的实例，隐藏了实例创建过程的复杂度，解耦了生产实例和使用实例的代码，降低了维护的复杂性。
本文会用Python实现三种工厂模式的简单例子，所有代码都托管在Github上。

简单工厂

首先，我们先看一个简单工厂的例子：

```
class Mercedes(object):
    """梅赛德斯
    """
    def __repr__(self):
        return "Mercedes-Benz"

class BMW(object):
    """宝马
    """
    def __repr__(self):
        return "BMW"
```

假设我们有两个“产品”分别是**Mercedes**和**BMW**的汽车，如果没有“工厂”来生产它们，我们就要在代码中自己进行实例化，如：

```
mercedes = Mercedes()
bmw = BMW()
```

但现实中，你可能会面对很多汽车产品，而且每个产品的构造参数还不一样，这样在创建实例时会遇到麻烦。这时就可以构造一个“简单工厂”把所有汽车实例化的过程封装在里面。

```
class SimpleCarFactory(object):
    """简单工厂
    """
    @staticmethod
    def product_car(name):
        if name == 'mb':
            return Mercedes()
        elif name == 'bmw':
            return BMW()
```

有了**SimpleCarFactory**类后，就可以通过向固定的接口传入参数获得想要的对象实例，如下：

```
c1 = SimpleCarFactory.product_car('mb')
c2 = SimpleCarFactory.product_car('bmw')
```

工厂方法

虽然有了一个简单的工厂，但在实际使用工厂的过程中，我们会发现新问题：如果我们要新增一个“产品”，例如Audi的汽车，我们除了新增一个Audi类外还要修改SimpleCarFactory内的product_car方法。这样就违背了软件设计中的开闭原则[1]，即在扩展新的类时，尽量不要修改原有代码。所以我们在简单工厂的基础上把SimpleCarFactory抽象成不同的工厂，每个工厂对应生成自己的产品，这就是工厂方法。

```
import abc

class AbstractFactory(object):
    """抽象工厂
    """
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def product_car(self):
        pass

class MercedesFactory(AbstractFactory):
    """梅赛德斯工厂
    """
    def product_car(self):
        return Mercedes()

class BMWFactory(AbstractFactory):
    """宝马工厂
    """
    def product_car(self):
        return BMW()
```

我们把工厂抽象出来用abc模块[2]实现了一个抽象的基类AbstractFactory，这样就可以通过特定的工厂来获得特定的产品实例了：

```
c1 = MercedesFactory().product_car()
c2 = BMWFactory().product_car()
```

每个工厂负责生产自己的产品也避免了我们在新增产品时需要修改工厂的代码，而只要增加相应的工厂即可。如新增一个Audi产品，只需新增一个Audi类和AudiFactory类。

抽象工厂

工厂方法虽然解决了我们“修改代码”的问题，但如果我们要生产很多产品，就会发现我们同样需要写很多对应的工厂类。比如如果MercedesFactory和BMWFactory不仅生产小汽车，还要生产SUV，那我们用工厂方法就要再多构造两个生产SUV的工厂类。所以为了解决这个问题，我们就要再更进一步的抽象工厂类，让一个工厂可以生产同一类的多个产品，这就是抽象工厂。具体实现如下：

```
import abc


class Mercedes_C63(object):
    """梅赛德斯 C63
    """
    def __repr__(self):
        return "Mercedes-Benz: C63"

class BMW_M3(object):
    """宝马 M3
    """
    def __repr__(self):
        return "BMW: M3"


class Mercedes_G63(object):
    """梅赛德斯 G63
    """
    def __repr__(self):
        return "Mercedes-Benz: G63"

class BMW_X5(object):
    """宝马 X5
    """
    def __repr__(self):
        return "BMW: X5"

class AbstractFactory(object):
    """抽象工厂
    可以生产小汽车外，还可以生产SUV
    """
    __metaclass__ = abc.ABCMeta

    @abc.abstractmethod
    def product_car(self):
        pass

    @abc.abstractmethod
    def product_suv(self):
        pass

class MercedesFactory(AbstractFactory):
    """梅赛德斯工厂
    """
    def product_car(self):
        return Mercedes_C63()

    def product_suv(self):
        return Mercedes_G63()

class BMWFactory(AbstractFactory):
    """宝马工厂
    """
    def product_car(self):
        return BMW_M3()

    def product_suv(self):
        return BMW_X5()
```

我们让基类**AbstractFactory**同时可以生产汽车和SUV，然后令**MercedesFactory**和**BMWFactory**继承**AbstractFactory**并重写product_car和product_suv方法即可。

```
c1 = MercedesFactory().product_car()
s1 = MercedesFactory().product_suv()
print(c1, s1)
s2 = BMWFactory().product_suv()
c2 = BMWFactory().product_car()
print(c2, s2)
```

抽象工厂模式与工厂方法模式最大的区别在于，抽象工厂中的一个工厂对象可以负责多个不同产品对象的创建 ，这样比工厂方法模式更为简单、有效率。

结论

初学设计模式时会对三种工厂模式的实际应用比较困惑，其实三种模式各有优缺点，应用的场景也不尽相同：

- 简单工厂模式适用于需创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂的情况下，而且用户只关心那种类型的实例被创建，并不关心其初始化过程时，比如多种数据库(MySQL/MongoDB)的实例，多种格式文件的解析器(XML/JSON)等。
- 工厂方法模式继承了简单工厂模式的优点又有所改进，其不再通过一个工厂类来负责所有产品的创建，而是将具体创建工作交给相应的子类去做，这使得工厂方法模式可以允许系统能够更高效的扩展。实际应用中可以用来实现系统的日志系统等，比如具体的程序运行日志，网络日志，数据库日志等都可以用具体的工厂类来创建。
- 抽象工厂模式在工厂方法基础上扩展了工厂对多个产品创建的支持，更适合一些大型系统，比如系统中有多于一个的产品族，且这些产品族类的产品需实现同样的接口，像很多软件系统界面中不同主题下不同的按钮、文本框、字体等等。

参考

[1]维基百科
[2]Python官方文档

2018/1/30更新：修改工厂方法的代码示例，新增结论一节。