Дизайн шаблони

д-р Филип Андонов

12 юли 2022 г.

Съдържание

Singleton

Шаблонът Singleton в python служи за създаването на една единствена инстанция на клас по време на изпълнение на програмата. Използването на singleton носи някои предимства:

- Ограничава конкуртния достъп до споделен ресурс
- Създава глобална точка за достъп до ресурс
- Създава само една инстанция на клас за цялото изпълнение на програмата

Singleton

Има различни начини да се реализира този шаблон:

- Singleton на ниво модул
- Класически singleton
- Borg Singleton

Singleton

```
1  ## singleton.py
2  shared_variable = "Shared Variable"
3   singleton.py

5  ## samplemodule1.py
6  import singleton
7  print(singleton.shared_variable)
8  singleton.shared_variable += "(modified by
       samplemodule1)"
9  samplemodule1.py

11 ##samplemodule2.py
12 import singleton
13 print(singleton.shared_variable)
```

Singleton на ниво модул

Всички модули са singleton по дефиниция. Нека създадем singleton на ниво модул, който споделя данните си с други модули. Създаваме три файла - singleton.py, sample_module1.py и sample_module2.py.

Singleton

```
1  class SingletonClass(object):
2    def __new__(cls):
3      if not hasattr(cls, 'instance'):
4        cls.instance = super(SingletonClass, cls).
             __new__(cls)
5      return cls.instance

7  singleton = SingletonClass()
8  new_singleton = SingletonClass()

10 print(singleton is new_singleton)

12 singleton.singl_variable = "Singleton Variable"
13 print(new_singleton.singl_variable)
```

Singleton

Класическата имплементация на Singleton създава инстанция само ако до момента няма създадена такава, в противен случай връща референция към вече създадената.

```
1  class Singleton:
2      instance = None
3
4      def getInstance():
5          if Singleton.instance == None:
6              Singleton.instance = Singleton()
7          return Singleton.instance
8
9      def __init__(self):
10         pass
11
12
13 s1 = Singleton.getInstance()
14 s2 = Singleton.getInstance()
15
16 print(s1==s2)
```

В този примитивен вариант не използваме нито статични нито клас методи, нито имаме нещо в конструктора. Това работи, стига да не ползваме конструктора директно, което ще доведе до създаването на повече от една инстанция. Тоест тази имплементация не предотвратява заобикаляне на механизъма, реализиран в getInstance за създаване само на единична инстанция.

```
1  class Singleton:
2      instance = None
3
4      @staticmethod
5      def getInstance():
6          """ Static access method. """
7          if Singleton.instance == None:
8              Singleton()
9          return Singleton.instance
10     def __init__(self):
11         """ Virtually private constructor. """
12         if Singleton.instance != None:
13             raise Exception("This class is a
                   singleton!")
14         else:
15             Singleton.instance = self
```

Тук използваме декоратор за статичен метод. Независимо дали създаваме инстанция през конструктора или през метода getInstance, резултатът е винаги един и същи обект.

```
1 s1 = Singleton()
2 s2 = Singleton.getInstance()
3 print(s==s2)
```

Нека видим още една имплементация. В специалния метод __new__ проверяваме дали има вече създадена инстанция. Ако да - ще я върнем, в противен случай ще я създадем. Тук освен всичко останало се уверяваме, че ако наследим от класа SingletonClass неговите наследници ще сочат към същата инстанция и споделят едно и също състояние.

```
1  class SingletonClass(object):
2    def __new__(cls):
3      if not hasattr(cls, 'instance'):
4        cls.instance = super(SingletonClass, cls).
             __new__(cls)
5      return cls.instance
6
7  class SingletonChild(SingletonClass):
8      pass
9
10 singleton = SingletonClass()
11 child = SingletonChild()
12 print(child is singleton)
13
14 singleton.singl_variable = "Singleton Variable"
15 print(child.singl_variable)
```

Борг сингълтън

Борг сингълтън е дизайн шаблон който позволява споделяне на състояние измежду различни инстанции. Методът връща нова инстанция всеки път когато бъде извикан, но всички инстанции споделят състояние дефакто това отново изпълнява условията за сингълтън. Решението е в използването на специална променлива на класа dict. Всички свойства на обекта се съхраняват в тази специална променлива на класа.

```
1 class Borg:
2     __shared_state = {}
3     def __init__(self):
4         self.__dict__ = self.__shared_state

6 a=Borg()
7 a.toto = 12
```

Борг сингълтън

```
1  class BorgSingleton(object):
2      _shared_borg_state = {}
3
4      def __new__(cls, *args, **kwargs):
5          obj = super(BorgSingleton, cls).__new__(cls,
               *args, **kwargs)
6          obj.__dict__ = cls._shared_borg_state
7          return obj
8
9  borg = BorgSingleton()
10 borg.shared_variable = "Shared Variable"
```

Борг сингълтън

```
11  class ChildBorg(BorgSingleton):
12    pass

14  childBorg = ChildBorg()
15  print(childBorg is borg)
16  print(childBorg.shared_variable)
```

Борг сингълтън

В процеса на създаване на нова инстанция се дефинира и споделено състояние в метода __new__. То се записва в атрибута shared_borg_state и се съхранява в речника __dict__ на всяка инстанция. Ако искаме различно състояние, можем да го занулим с атрибута shared_borg_state.

Борг сингълтън

```
1  class BorgSingleton(object):
2      _shared_borg_state = {}
3
4      def __new__(cls, *args, **kwargs):
5          obj = super().__new__(cls, *args, **kwargs)
6          obj.__dict__ = cls._shared_borg_state
7          return obj
8
9  borg = BorgSingleton()
10 borg.shared_variable = "Shared Variable"
11
12 class NewChildBorg(BorgSingleton):
13     _shared_borg_state = {}
14
15 newChildBorg = NewChildBorg()
16 print(newChildBorg.shared_variable)
```

Сингълтън

Нека разгледаме някои употреби на сингълтън класове.

- Управление на връзки към бази данни
- Глобално място за достъп до записите на дневника (log)
- Файлов мениджър
- Опашка за принтер

Factory

Шаблонът Factory се използва за скриване на процеса на създаване на обекти.
Предимствата му са:

- Създаването на обекти може да е независимо от реализацията на класа
- Добавянето на поддръжка на нов тип обекти е лесно
- Логиката по създаване на обекти е скрита

Factory

Съществуват няколко вида шаблона:

- Factory метод шаблон - позволява създаването на обекти, но оставя решението на производните класове за това от кой клас да създадат обект.
- Абстрактен Factory шаблон - Представлява интерфейс за създаване на свързани обекти без да се определя/излага принадлежността им към даден клас. Шаблонът предоставя обекти към друго factory, което вътрешно създава други обекти.

Factory

Общият проблем е следният:

- По време на изпълнение на задълженията си, функция трябва да създаде обекти от името на извикващия.
- Но класът, който функцията ще инстанциира по подразбиране не покрива всички възможни случаи.
- Вместо да пишем явно класът по подразбиране, функцията ще иска от извикващия да определи какви класове да инстанциира.

Factory

За Python имплементацията ще се възползваме от факта, че всеки извикваем обект е first class. Това означава, че извикваемите обекти могат да се подават като параметри, да се връщат като стойности и да се съхраняват в структури. Това ни дава мощен механизъм за реализиране на factory, което просто означава "функция, която създава и връща нови обекти".

Factory

Тук Factory предоставя възможност за създаване на всичко свързано със задачата - както структурата, в която да се съхраняват обектите, така и отделните обекти.

```
1 class Factory(object):
2     def build_sequence(self):
3         return []
4
5     def build_number(self, string):
6         return Decimal(string)
```

Factory

А това е класът, който използва това factory.

```
1  class Loader(object):
2      def load(string, factory):
3          sequence = factory.build_sequence()
4          for substring in string.split(','):
5              item = factory.build_number(substring
                   )
6              sequence.append(item)
7          return sequence
8
9  f = Factory()
10 result = Loader.load('1.23, 4.56', f)
11 print(result)
```

Factory

Всеки избор, който трябва да направи относно създаването на инстанции е оставен за factory-то, вместо да се намира в самия парсер.
В строго типизираните езици такава имплементация ще бъде проблемна, защото ще ограничава какъв тип factory можем да подадем. Вместо това, за да разделим спецификацията от имплементацията, ще трябва да създадем абстрактен клас. Затова и шаблонът се нарича така. Абстрактният клас ще "обещава че factory-то което зарежда ще бъде клас, който се придържа към интерфейса.

Factory

```
1  from abc import ABCMeta, abstractmethod

3  class AbstractFactory(metaclass=ABCMeta):

5      @abstractmethod
6      def build_sequence(self):
7          pass

9      @abstractmethod
10     def build_number(self, string):
11         pass
```

Factory

Когато Factory наследи абстрактния клас, всичко, което се случва по време на изпълнение е същото. Методите на factory се извикват с различни аргументи, което създава различни видове обекти. Те биват върнати без извикващата програма да знае детайлите. Това е прекалено усложнение, затова използвайте описаното в предходния пример.

Factory метод

Преди да използваме този шаблон трябва да проверим дали наистина класът, който създаваме, трябва да създава други класове. Ако предварително знаем всички обекти, от които класът се нуждае, можем да използваме Вмъкване на Зависимости (Dependancy Injection). Това по-лесен шаблон се използва в библиотеките на Python - например където подавате вече отворен файлов обект вместо да предоставяте пътя и да очаквате метода да създаде файла. Ето пример от стандартния JSON парсер. Неговият метод load() очаква отворен файл.

```
1 import json
2 with open('input_data.json') as f:
3     data = json.load(f)
```

Factory метод

Като оставя създаването на инстанцията на файловия обект извън метода, това постига няколко неща:
Decoupling: библиотеката няма нужда да знае всички параметри на метода open() и да ги приема като свои параметри. Ако open() бъде променен в бъдеще, json.load() няма да има нужда от промяна.
Ефективност: Ако по някаква причина вече имате файла отворен, можете директно да го предоставите файловия обект, вместо библиотеката да го отваря повторно.
Гъвкавост: Можете да подадете какъвто си искате файлоподобен обект. Може да е наследник на стандартния файлов обект, а може и да нещо съвсем различно.

Factory метод

В такива прости случаи вмъкването на зависимости напълно елиминира нуждата от обекти като jSON парсера да бъдат усложнявани с детайли за това как се създават други обекти. Следователно ако А. класът ви винаги се нуждае от съществуването на конкретен обект и Б. ако потребителите искат поне да могат да настройват параметрите, с които се създават инстанциите, трябва да обмислим дали да не ползваме вмъкване на зависимости.

Factory метод

В Python няма особен смисъл от този шаблон, но ако използвате език, в който: Класовете не са first-class обекти. Не се разрешава подаването на клас като атрибут на друга инстанция на клас или на друг клас. Функциите не са first-class обекти. Не се разрешава записването на функции като атрибут на клас или инстанция на клас. Не съществуват друг тип извикваеми обекти, които да могат динамично да определят и се закачат за обекти по време на изпълнение.

Factory метод

При такива обстоятелства можем да използваме наследяемостта като естествен начин да закачаме действия към съществуващи класове, и да използваме пренаписването на методи за промяна на поведението. И ако един от класовете има специален метод, чиято единствено предназначение е да изолира процеса на създаване на нови обекти, тогава използваме шаблона Factory метод.

Factory метод

Този шаблон често се наблюдава в код, взаимстван от обектно-ориентирани, но ограничени, езици. Например модулът за дневници в стандартната библиотека:

Listing 1: Наличие на ключ

```
class Handler(Filterer):
    ...
    def __init__(self, level=NOTSET):
        ...
        self.createLock()
    ...
    def createLock(self):
        """
        Acquire a thread lock for serializing
           access to the underlying I/O.
        """
        self.lock = threading.RLock()
```

Factory метод

Ако искаме да създадем Handler който използва специален вид lock трябва да наследим класа и да пренапишем метода createLock() така, че да връща желания lock. Този процес е тромав и ако искаме да настройваме обектите Handler ще трябва да правим много класове.

Factory метод

Дизайн шаблоните Factory и Factory метод се считат за неподходящи за Python. Те са измислени за програмни езици, в които класовете и функциите не могат да бъдат подавани като параметри или съхранявани като атрибути. В тези езици Factory метода е необходим, макар и странен.
Абстрактно Factory се обезмисля ако можем да подадем директно клас или функция когато библиотека иска да създава обекти от наше име.

Декоратор

Този шаблон се отнася за добавянето на допълнителна функционалност без да се използва наследяемост и като се запазва интерфейса. В библията на GangOfFour името му е "Wrapper". Този шаблон може също да се използва за да се преработи сложен клас така, че да се разбие на по-малки парчета. Дори да не е необходимо да се закачат отговорности динамично, по-чисто е всяко действие да е в отделен клас.

Декоратор

Проблемът, който декораторът решава, е експлозията на подкласове. Нека имаме Logger клас, чието стандартно поведение е да записва дневника на съобщенията във файл. Ако искаме да имаме подобно поведение, но записът да става в сокет или в системния дневник, класическото решение е да наследим базовия клас Logger и да направим производни класове SocketLogger и SyslogLogger, които да пренаписват метода log().

Декоратор

```
1  class Logger(object):
2      def __init__(self, file):
3          self.file = file
4
5      def log(self, message):
6          self.file.write(message + '\n')
7          self.file.flush()
8
9  class SocketLogger(Logger):
10     def __init__(self, sock):
11         self.sock = sock
12
13     def log(self, message):
14         self.sock.sendall((message + '\n').encode
               ('ascii'))
```

Декоратор

```
15 class SyslogLogger(Logger):
16     def __init__(self, priority):
17         self.priority = priority
18
19     def log(self, message):
20         syslog.syslog(self.priority, message)
```

Декоратор

Ако в тази ситуация искаме да имаме Logger, който филтрира съобщенията, ние можем лесно да го направим за базовия клас.

```
1  class FilteredLogger(Logger):
2      def __init__(self, pattern, file):
3          self.pattern = pattern
4          super().__init__(file)
5
6      def log(self, message):
7          if self.pattern in message:
8              super().log(message)
9
10 f = FilteredLogger('Error', sys.stdout)
11 f.log('Ignored: this is not important')
12 f.log('Error: but you want to see this')
```

Декоратор

Проблемът е, че ако искаме да имаме филтриращ Logger, който работи със сокети или със системния дневник, трябва да направим още два класа. Така общият брой стана 6.
Решението са осъзнаем, че класът е отговорен за прекалено много неща. Това нарушава ООП принципа, че един клас трябва да отговаря само за едно нещо (Single Responsibility).

Декоратор

Решението на този проблем може да се търси и в шаблоните адаптер и мост. Но с декоратор постигаме нещо повече - можем да наслагваме няколко филтъра един върху друг.

Декоратор

```
1  class FileLogger:
2      def __init__(self, file):
3          self.file = file
4
5      def log(self, message):
6          self.file.write(message + '\n')
7          self.file.flush()
8
9  class SocketLogger:
10     def __init__(self, sock):
11         self.sock = sock
12
13     def log(self, message):
14         self.sock.sendall((message + '\n').encode
               ('ascii'))
```

Декоратор

```
15 class SyslogLogger:
16     def __init__(self, priority):
17         self.priority = priority
18
19     def log(self, message):
20         syslog.syslog(self.priority, message)
21
22 class LogFilter: # The filter calls the same
      method it offers.
23     def __init__(self, pattern, logger):
24         self.pattern = pattern
25         self.logger = logger
26
27     def log(self, message):
28         if self.pattern in message:
29             self.logger.log(message)
```

Филтриращият код е извън който и да е logger клас. Той е самостоятелна функционалност, която може да се обвива около който logger пожелаем.
Филтрирането може да се комбинира с изхода на съобщенията без да имаме специлни комбинирани класове за това.
Но най-важното е, че понеже декораторът е симетричен - той предлага точно същия интерфейс, който обвива, ние можем да слагаме като матрьошки един филтър върху друг.

Декоратор

```
30 log1 = FileLogger(sys.stdout)
31 log2 = LogFilter('Error', log1)

33 log1.log('Noisy: this logger always produces
       output')

35 log2.log('Ignored: this will be filtered out')
36 log2.log('Error: this is important and gets
       printed')

38 log3 = LogFilter('severe', log2)

40 log3.log('Error: this is bad, but not that bad')
41 log3.log('Error: this is pretty severe')
```

Задача

Напишете класове за няколко животни. Нека те имат абстрактен базов клас, който дефинира, че всеки производен трябва да реализира методите make_sound() и move(). Създайте абстрактно factory и factory метод, които да създават необходимото животно, в зависимост от текстов файл, който съдържа списък с имена на животни.

Задача

Използвайте декоратор, за да се калкулира цената и разширява описанието на продукти, продавани в гурме кафене - например продукти кафе и decaf и добавки сметана, веган сметана, краве мляко, соево, кокосово, бадемово, канела и т.н.

Благодаря за вниманието!

Въпроси?

References

https://python-patterns.guide/gang-of-four/factory-method/