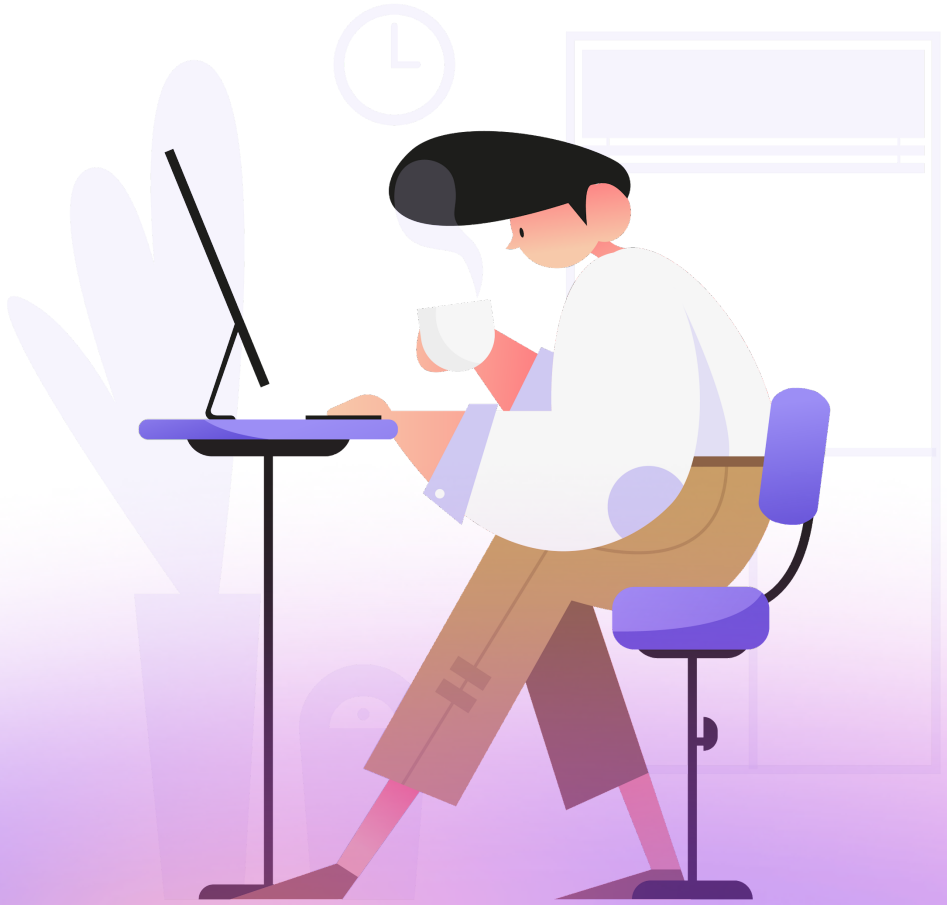
GeekBrains

Объектно-ориентированные паттерны, группа порождающие: Object Pool



Введение

Мы разберем фундаментальное понятие паттерна и рассмотрим особенности некоторых порождающих паттернов.

Что же такое паттерны проектирования?

«Это подходы к решению практических задач, выявленные при анализе полученных решений и применяемые многократно».

В ближайшие уроки мы изучим порождающие паттерны (отвечают за создание объектов) и структурные паттерны (как написаны классы и как из них формируются более сложные системы).

Порождающие абстрагируют процесс инстанцирования объектов (создания объектов классов), чтобы сделать систему более независимой от способов создания и представления объектов.

То есть, когда мы создаем объект, это вносит дополнительные зависимости, зависимости от процесса создания объекта (инстанцирования). Задача порождающих паттернов – избавиться от зависимостей так, чтобы система была более абстрактной и более гибкой.

Пул объектов

«Идея паттерна в том, что если есть необходимость повторного использования экземпляров класса, то можно работать с уже существующими объектами без создания новых».

Листинг 1. Урок 11. Коды к уроку/object_pool.py

```
class StandardReport:

    def __init__(self):
        self.is_filled = False

    def reset(self):
        self.is_filled = True
```

Мы создали простой класс с одним атрибутом и одним методом, который переустанавливает значение этого атрибута.

Мы реализовали класс для сущности «Отчет». Нам нужно создавать объекты этого класса.

Например, нам необходимо создать несколько объектов этого класса и работать только с ними. Но мы не будем каждый раз создавать объекты, а создадим их один раз и поместим в пул. Если нам нужно будет использовать объект класса, мы получим его из пула, поработаем с ним, а затем поместим обратно в пул для повторного использования.

Листинг 2. Урок 11. Коды к уроку/object_pool.py

```
class ReportsPool:
    def __init__(self, size):
```

```
        self.reports = [StandardReport() for _ in range(size)]

    def acquire(self):
        if self.reports:
            return self.reports.pop()
        else:
            self.reports.append(MyClass())
            return self.reports.pop()

    def release(self, reusable):
        reusable.reset()
        self.reports.append(reusable)
```

Теперь мы реализовали второй класс, который и обеспечивает возможности пула объектов.

Если у объекта уже есть некоторое исходное состояние, мы будем менять его после извлечения из пула.

Разберем отдельные выражения.

```
self.reports = [StandardReport() for _ in range(size)]
```

Создаем семейство объектов класса.

```
def acquire(self):
    if self.reports:
        return self.reports.pop()
    else:
        self.reports.append(MyClass())
        return self.reports.pop()
```

Если пул уже содержит объекты, мы удаляем и возвращаем крайний с правого конца объект пула. Иначе создаем и размещаем в пуле первый объект и также удаляем, и возвращаем его.

```
def release(self, reusable):
    reusable.reset()
    self.reports.append(reusable)
```

Данный метод обеспечивает изменение свойства объекта и возврат его обратно в пул для повторного использования.

Теперь реализуем клиентский код и выполним запуск.

Листинг 3. Урок 11. Коды к уроку/object_pool.py

```
pool = ReportsPool(3)
print(pool.reports)

"""
[<__main__.StandardReport object at 0x000000E23CD74748>,
<__main__.StandardReport object at 0x000000E23CD74828>,
<__main__.StandardReport object at 0x000000E23CD74710>]
"""

reusable = pool.acquire()
print(reusable.is_filled)  # -> False

print(pool.reports)
```

```
"""
[<__main__.StandardReport object at 0x000000E23CD74748>,
<__main__.StandardReport object at 0x000000E23CD74828>]
"""

pool.release(reusable)
print(reusable.is_filled)  # -> True

print(pool.reports)

"""
[<__main__.StandardReport object at 0x000000E23CD74748>,
<__main__.StandardReport object at 0x000000E23CD74828>,
 <__main__.StandardReport object at 0x000000E23CD74710>]

"""
```

У вас может возникнуть вопрос, а зачем создавать пул объектов для повторного использования вместо того, чтобы привычным образом каждый раз создавать новый объект? Дело в том, что существуют ситуации, когда создание нового объекта требует большого количества ресурсов. В этом случае пул объектов будет оптимальным решением.

Синглтон («одиночка»)

«Гарантирует, что класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему».

Довольно редко используемый паттерн.

Смысл – всегда используем один экземпляр класса. Т.е. даже когда будем создавать несколько экземпляров, всегда будем получать и работать с первым созданным объектом.

Вы можете сказать, что это противоречит «религии» Python, когда каждый новый объект класса именно новый, т.е. под каждый объект выделяется область памяти. Но когда речь идет о таких сущностях, как класс-логгер, класс-сокет, класс-база данных, нам вполне будет достаточно создать один объект класса и далее работать, и делать все операции через этот объект.

Какой смысл? Опять же, как в случае с пулом объектов, речь может идти о ресурсах на создание экземпляра класса. Паттерн «Синглтон» переплетается с паттерном «Пул объектов».

Небольшое предисловие:

Листинг 4. Урок 11. Коды к уроку/object_pool.py

```
class Origin:
    pass


o1 = Origin()
o2 = Origin()
o3 = Origin()
```

```
print(o1)
print(o2)

print(o1 is o2)

a = []
b = a
print(a is b)

b = a.copy()
print(a is b)
```

Давайте запустим приведенный код и разберем его. Этот код важен тем, что он наглядно показывает особенности религии «Python».

```
class Origin:
    pass


o1 = Origin()
o2 = Origin()
o3 = Origin()
```

Мы создаем три объекта представленного класса. Это именно три уникальных объекта. Для каждого происходит выделение памяти.

Проверим так ли это:

```
print(o1)
print(o2)
print(o3)
```

Результат:

```
<__main__.Origin object at 0x00000070029CD438>

<__main__.Origin object at 0x0000007002B47668>

<__main__.Origin object at 0x0000007002B47630>
```

Для каждого из объектов выделяется область памяти.

Уверен, вы помните, что в Python переменная – это ссылка на объект.

Рассмотрим данный блок:

```
a = []
b = a
print(a is b)
```

Результат:

```
True
```

Все верно, т.к. объект в памяти один, а ссылок на него может быть много.

Но в случае копирования происходит выделение памяти под новый объект.

```
b = a.copy()
print(a is b)
```

Результат:

```
False
```

Паттерн «Синглтон» предлагает нам нарушить «религию» Python и сделать так, что подобный код:

```
class Origin:
    pass


o1 = Origin()
o2 = Origin()
o3 = Origin()


print(o1 is o2)
```

При запуске даст нам True.

Получается, нам нужно вмешаться в фундаментальный алгоритм создания объектов. Нам нужно в него вмешаться и изменить. Но как это сделать?
В Python любой класс, по сути, тоже является объектом. Т.е. над каждым классом есть свой класс. Он называется метаклассом.
Но какой это класс? Он называется **type**. И он контролирует как при создании объектов подчиненного класса создаются его объекты. А мы хотим при создании объектов всегда получать первый созданный объект.
Значит нам нужно переопределить стандартный метакласс и вмешаться в его работу на этапе создания объекта (это метод __call__).

Листинг 5. Урок 11. Коды к уроку/singleton_2.py

```
class Singleton(type):

    def __init__(cls, name, bases, attrs):
        super().__init__(name, bases, attrs)
        cls.__instance = None

    def __call__(cls, *args, **kwargs):
        if cls.__instance is None:
            cls.__instance = super().__call__(*args, **kwargs)
        return cls.__instance


class MySqlConnection(metaclass=Singleton):
    pass
```

Теперь класс **MySqlConnection**, а также создание его объектов находятся под контролем метакласса **Singleton**.

Важно отметить, что не нужно путать понятия «Наследование» и «Метаклассы». Цель метаклассов – не передать подчиненным классам свои атрибуты и методы, а контролировать поведение подчиненных классов и их объектов. Поэтому метакласс «знает» о подчиненном классе все, в том числе его имя, список его базовых классов, атрибутов:

```
def __init__(cls, name, bases, attrs):
    cls.__instance = None
```

И может на все это влиять.

Итак, мы хотим повлиять на сам подчиненный класс (переменная **cls**).

Изначально у подчиненного класса объекты отсутствуют.

```
cls.__instance = None
```

Мы хотим повлиять на создание объектов класса, поэтому выполним перегрузку метода **__call__**.

```
def __call__(cls, *args, **kwargs):
    if cls.__instance is None:
        cls.__instance = super().__call__(*args, **kwargs)
    return cls.__instance
```

Идея проста. Если объект еще не создавался, мы его создадим и свяжем с объектом переменную ссылку cls.__instance.

Если объект уже создавался, мы получим будет возвращен именно он. Ведь на него указывает переменная cls.__instance.

Кстати, выражение:

```
super().__call__(*args, **kwargs)
```

Указывает, что мы делегируем создание объекта класса **MySqlConnection** метаклассу **Singleton**.

Напишем клиентский код и проверим как все отрабатывает:

```
sql_connection_1 = MySqlConnection()
sql_connection_2 = MySqlConnection()
sql_connection_3 = MySqlConnection()

print(sql_connection_1 is sql_connection_3)
```

Результат:

```
True
```

Метакласс выполнил свою задачу.