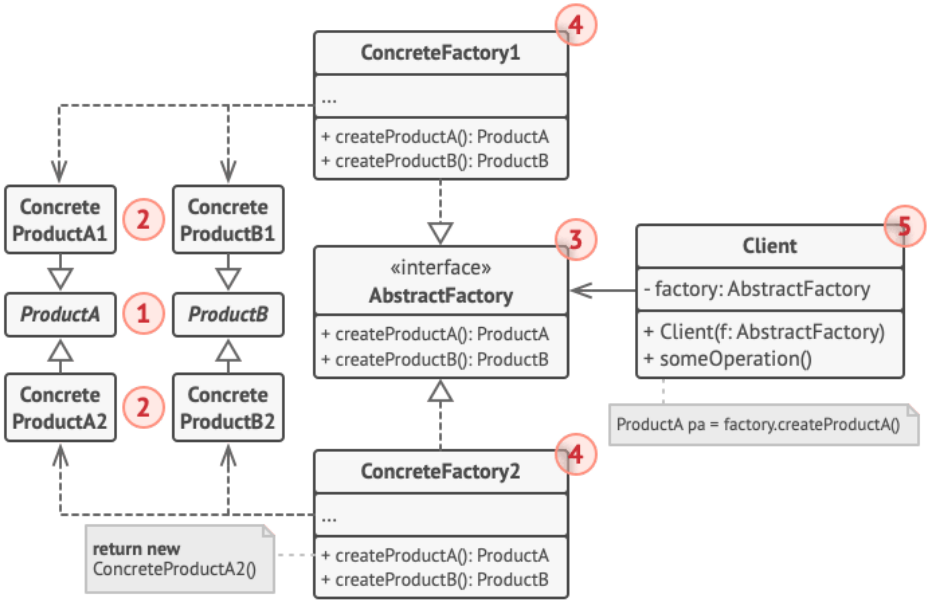
创建型模式 Creation Pattern

Abstract Factory 抽象工厂

抽象工厂模式是一种创建型设计模式， 它能创建一系列相关的对象， 而无需指定其具体类。

The Abstract factory pattern is a creative design pattern that creates a series of related objects without specifying their concrete classes.



1. **抽象产品** （Abstract Product） 为构成系列产品的一组不同但相关的产品声明接口。
2. **具体产品** （Concrete Product） 是抽象产品的多种不同类型实现。 所有变体 （维多利亚/现代） 都必须实现相应的抽象产品 （椅子/沙发）。
3. **抽象工厂** （Abstract Factory） 接口声明了一组创建各种抽象产品的方法。
4. **具体工厂** （Concrete Factory） 实现抽象工厂的构建方法。 每个具体工厂都对应特定产品变体， 且仅创建此种产品变体。
5. 尽管具体工厂会对具体产品进行初始化， 其构建方法签名必须返回相应的抽象产品。 这样， 使用工厂类的客户端代码就不会与工厂创建的特定产品变体耦合。 **客户端** （Client） 只需通过抽象接口调用工厂和产品对象， 就能与任何具体工厂/产品变体交互。

场景

- 如果代码需要与多个不同系列的相关产品交互， 但是由于无法提前获取相关信息， 或者出于对未来扩展性的考虑， 你不希望代码基于产品的具体类进行构建， 在这种情况下， 你可以使用抽象工厂。
- 如果你有一个基于一组抽象方法的类， 且其主要功能因此变得不明确， 那么在这种情况下可以考虑使用抽象工厂模式。

优缺点

Advantage

- 你可以确保同一工厂生成的产品相互匹配。
- 你可以避免客户端和具体产品代码的耦合coupling。
- 单一职责原则。 你可以将产品生成代码抽取到同一位置， 使得代码易于维护。
- 开闭原则。 向应用程序中引入新产品变体时， 你无需修改客户端代码。

Disadvantage

- 由于采用该模式需要向应用中引入众多接口和类， 代码可能会比之前更加复杂。

关系

- 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
- 生成器重点关注如何分步生成复杂对象。 抽象工厂专门用于生产一系列相关对象。 抽象工厂会马上返回产品， 生成器则允许你在获取产品前执行一些额外构造步骤。
- 抽象工厂模式通常基于一组工厂方法， 但你也可以使用原型模式来生成这些类的方法。
- 当只需对客户端代码隐藏子系统创建对象的方式时， 你可以使用抽象工厂来代替外观模式。
- 你可以将抽象工厂和桥接模式搭配使用。 如果由桥接定义的抽象只能与特定实现合作， 这一模式搭配就非常有用。 在这种情况下， 抽象工厂可以对这些关系进行封装， 并且对客户端代码隐藏其复杂性。
- 抽象工厂、 生成器和原型都可以用单例模式来实现。

Code

```python
from __future__ import annotations
from abc import ABC, abstractmethod


class AbstractFactory(ABC):

    @abstractmethod
    def create_product_a(self) -> AbstractProductA:
        pass

    @abstractmethod
    def create_product_b(self) -> AbstractProductB:
        pass


class ConcreteFactory1(AbstractFactory):

    def create_product_a(self) -> AbstractProductA:
        return ConcreteProductA1()

    def create_product_b(self) -> AbstractProductB:
        return ConcreteProductB1()


class ConcreteFactory2(AbstractFactory):

    def create_product_a(self) -> AbstractProductA:
        return ConcreteProductA2()

    def create_product_b(self) -> AbstractProductB:
        return ConcreteProductB2()


class AbstractProductA(ABC):

    @abstractmethod
    def useful_function_a(self) -> str:
```

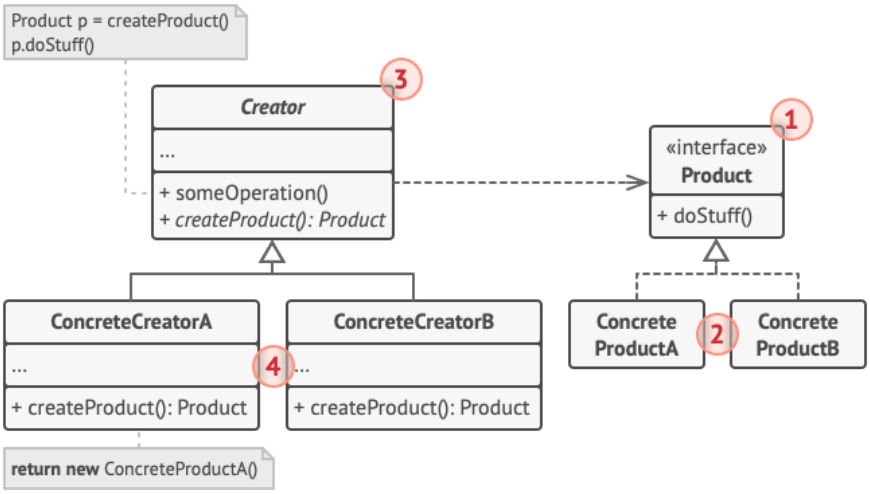
```
        pass


class ConcreteProductA1(AbstractProductA):
    def useful_function_a(self) -> str:
        return "The result of the product A1."


class ConcreteProductA2(AbstractProductA):
    def useful_function_a(self) -> str:
        return "The result of the product A2."


class AbstractProductB(ABC):

    @abstractmethod
    def useful_function_b(self) -> None:
        """
        Product B is able to do its own thing...
        """
        pass

    @abstractmethod
    def another_useful_function_b(self, collaborator: AbstractProductA) -> None:
        pass


class ConcreteProductB1(AbstractProductB):
    def useful_function_b(self) -> str:
        return "The result of the product B1."

    def another_useful_function_b(self, collaborator: AbstractProductA) -> str:
        result = collaborator.useful_function_a()
        return f"The result of the B1 collaborating with the ({result})"


class ConcreteProductB2(AbstractProductB):
    def useful_function_b(self) -> str:
        return "The result of the product B2."

    def another_useful_function_b(self, collaborator: AbstractProductA):
        result = collaborator.useful_function_a()
        return f"The result of the B2 collaborating with the ({result})"


def client_code(factory: AbstractFactory) -> None:
    product_a = factory.create_product_a()
    product_b = factory.create_product_b()

    print(f"{product_b.useful_function_b()}")
    print(f"{product_b.another_useful_function_b(product_a)}", end="")


if __name__ == "__main__":
    print("Client: Testing client code with the first factory type:")
    client_code(ConcreteFactory1())

    print("\n")
```

```
    print("Client: Testing the same client code with the second factory type:")
    client_code(ConcreteFactory2())
```

Factory Method 工厂方法

工厂方法模式是一种创建型设计模式， 其在父类中提供一个创建对象的方法， 允许子类决定实例化对象的类型。

The factory method pattern is a creative design pattern that provides a method to create an object in a parent class, allowing subclasses to determine the type of object to instantiate.



1. **产品** （Product） 将会对接口进行声明。 对于所有由创建者及其子类构建的对象， 这些接口都是通用的。
2. **具体产品** （Concrete Products） 是产品接口的不同实现。
3. **创建者** （Creator） 类声明返回产品对象的工厂方法。 该方法的返回对象类型必须与产品接口相匹配。

   你可以将工厂方法声明为抽象方法， 强制要求每个子类以不同方式实现该方法。 或者， 你也可以在基础工厂方法中返回默认产品类型。

   注意， 尽管它的名字是创建者， 但它最主要的职责并**不是**创建产品。 一般来说， 创建者类包含一些与产品相关的核心业务逻辑。 工厂方法将这些逻辑处理从具体产品类中分离出来。 打个比方， 大型软件开发公司拥有程序员培训部门。 但是， 这些公司的主要工作还是编写代码， 而非生产程序员。
4. **具体创建者** （Concrete Creators） 将会重写基础工厂方法， 使其返回不同类型的产品。

   注意， 并不一定每次调用工厂方法都会**创建**新的实例。 工厂方法也可以返回缓存、 对象池或其他来源的已有对象。

场景

- 当你在编写代码的过程中， 如果无法预知对象确切类别及其依赖关系时， 可使用工厂方法。
- 如果你希望用户能扩展你软件库或框架的内部组件， 可使用工厂方法。
- 如果你希望复用现有对象来节省系统资源， 而不是每次都重新创建对象， 可使用工厂方法。

优缺点

Advantage

- 你可以避免创建者和具体产品之间的紧密耦合。
- 单一职责原则。 你可以将产品创建代码放在程序的单一位置， 从而使得代码更容易维护。
- 开闭原则。 无需更改现有客户端代码， 你就可以在程序中引入新的产品类型。

Disadvantage

- 应用工厂方法模式需要引入许多新的子类， 代码可能会因此变得更复杂。 最好的情况是将该模式引入创建者类的现有层次结构中。

关系

- 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
- 抽象工厂模式通常基于一组工厂方法， 但你也可以使用原型模式来生成这些类的方法。
- 你可以同时使用工厂方法和迭代器模式来让子类集合返回不同类型的迭代器， 并使得迭代器与集合相匹配。
- 原型并不基于继承， 因此没有继承的缺点。 另一方面， 原型需要对被复制对象进行复杂的初始化。 工厂方法基于继承， 但是它不需要初始化步骤。
- 工厂方法是模板方法模式的一种特殊形式。 同时， 工厂方法可以作为一个大型模板方法中的一个步骤。

Code

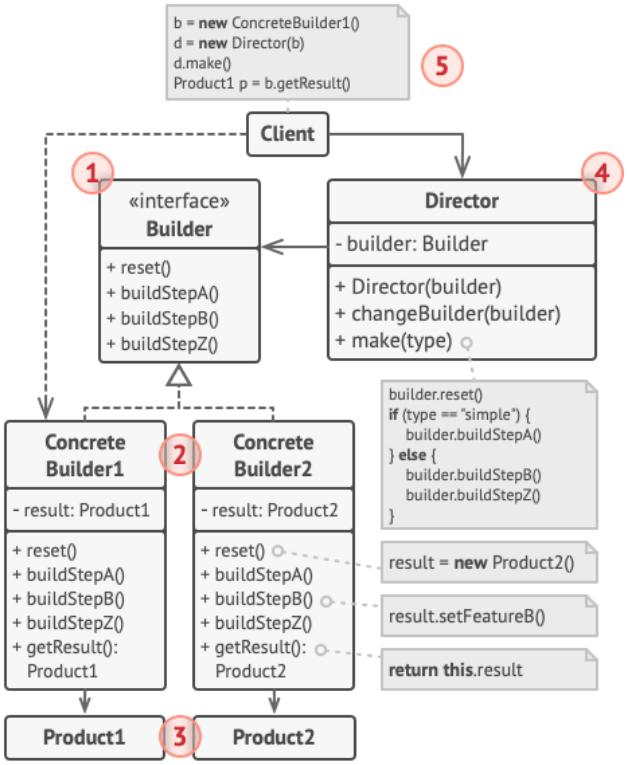
```
from abc import ABC, abstractmethod


class Creator(ABC):
    @abstractmethod
    def factory_method(self):
        pass

    def some_operation(self) -> str:
        product = self.factory_method()
        result = f"Creator: The same creator's code has just worked with
{product.operation()}"
        return result

class ConcreteCreator1(Creator):
    def factory_method(self) -> Product:
        return ConcreteProduct1()


class ConcreteCreator2(Creator):
    def factory_method(self) -> Product:
        return ConcreteProduct2()


class Product(ABC):
    @abstractmethod
    def operation(self) -> str:
        pass
```

```
class ConcreteProduct1(Product):
    def operation(self) -> str:
        return "{Result of the ConcreteProduct1}"


class ConcreteProduct2(Product):
    def operation(self) -> str:
        return "{Result of the ConcreteProduct2}"


def client_code(creator: Creator) -> None:
    print(f"Client: I'm not aware of the creator's class, but it still works.\n"
          f"{creator.some_operation()}", end="")


if __name__ == "__main__":
    print("App: Launched with the ConcreteCreator1.")
    client_code(ConcreteCreator1())
    print("\n")

    print("App: Launched with the ConcreteCreator2.")
    client_code(ConcreteCreator2())
```

Builder 生成器

生成器模式是一种创建型设计模式， 使你能够分步骤创建复杂对象。 该模式允许你使用相同的创建代码生成不同类型和形式的对象。

Builder pattern is a creative design pattern that allows you to create complex objects in steps. This pattern allows you to generate objects of different types and forms using the same creation code.



1. **生成器** （Builder） 接口声明在所有类型生成器中通用的产品构造步骤。

2. **具体生成器** （Concrete Builders） 提供构造过程的不同实现。 具体生成器也可以构造不遵循通用接口的产品。
3. **产品** （Products） 是最终生成的对象。 由不同生成器构造的产品无需属于同一类层次结构或接口。
4. **主管** （Director） 类定义调用构造步骤的顺序， 这样你就可以创建和复用特定的产品配置。
5. **客户端** （Client） 必须将某个生成器对象与主管类关联。 一般情况下， 你只需通过主管类构造函数的参数进行一次性关联即可。 此后主管类就能使用生成器对象完成后续所有的构造任务。 但在客户端将生成器对象传递给主管类制造方法时还有另一种方式。 在这种情况下， 你在使用主管类生产产品时每次都可以使用不同的生成器。

场景

- 使用生成器模式可避免 “重叠构造函数 （telescopic constructor）” 的出现。
- 当你希望使用代码创建不同形式的产品 （例如石头或木头房屋） 时， 可使用生成器模式。
- 使用生成器构造组合树或其他复杂对象。

优缺点

- 你可以分步创建对象， 暂缓创建步骤或递归运行创建步骤。
- 生成不同形式的产品时， 你可以复用相同的制造代码。
- 单一职责原则（Single responsibility principle）。 你可以将复杂构造代码从产品的业务逻辑中分离出来。

Disadvantage:

- 由于该模式需要新增多个类， 因此代码整体复杂程度会有所增加。

关系

- 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
- 生成器重点关注如何分步生成复杂对象。 抽象工厂专门用于生产一系列相关对象。 抽象工厂会马上返回产品， 生成器则允许你在获取产品前执行一些额外构造步骤。
- 你可以在创建复杂组合模式树时使用生成器， 因为这可使其构造步骤以递归的方式运行。
- 你可以结合使用生成器和桥接模式： 主管类负责抽象工作， 各种不同的生成器负责实现工作。
- 抽象工厂、 生成器和原型都可以用单例模式来实现。

Code ★

> Use the Builder Pattern to generate a Mac computer product and an Lenovo computer. (The computer product has CPU, ram, display and keyboard properties). Write a test class to test the properties of the generated product.

```
from __future__ import annotations
from abc import ABC, abstractmethod
from typing import Any


class Builder(ABC):

    @property
    @abstractmethod
    def product(self) -> None:
        pass
```

```
    @abstractmethod
    def produce_cpu(self) -> None:
        pass

    @abstractmethod
    def produce_ram(self) -> None:
        pass

    @abstractmethod
    def produce_display(self) -> None:
        pass

    @abstractmethod
    def produce_keyboard(self) -> None:
        pass


class MacComputerBuilder(Builder):
    def __init__(self) -> None:
        self._product = MacComputer()
        self.reset()

    def reset(self) -> None:
        pass

    @property
    def product(self) -> MacComputer:
        product = self._product
        self.reset()
        return product

    def produce_cpu(self) -> None:
        self._product.set_cpu('A13')

    def produce_display(self) -> None:
        self._product.set_display('Dell')

    def produce_keyboard(self) -> None:
        self._product.set_keyboard('MA887')

    def produce_ram(self) -> None:
        self._product.set_ram('9987')


class LenovoComputerBuilder:
    def __init__(self) -> None:
        self._product = LenovoComputer()
        self.reset()

    def reset(self) -> None:
        pass

    @property
    def product(self) -> LenovoComputer:
        product = self._product
        self.reset()
        return product
```

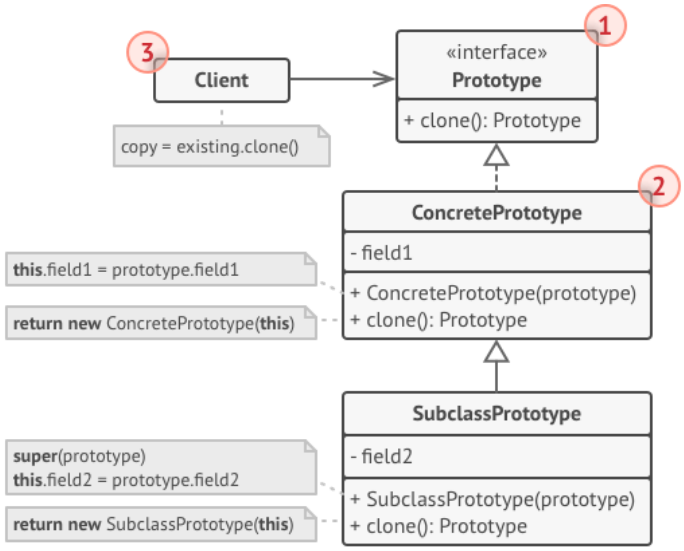
```
    def produce_cpu(self) -> None:
        self._product.set_cpu('IntelI8')

    def produce_display(self) -> None:
        self._product.set_display('DellAAA')

    def produce_keyboard(self) -> None:
        self._product.set_keyboard('asd')

    def produce_ram(self) -> None:
        self._product.set_ram('asd')


class Computer:
    def __init__(self):
        self.cpu = 'A13'
        self.ram = ''
        self.display = ''
        self.keyboard = ''

    def set_ram(self, ram):
        self.ram = ram

    def set_cpu(self, cpu):
        self.cpu = cpu

    def set_display(self, display):
        self.display = display

    def set_keyboard(self, keyboard):
        self.keyboard = keyboard

    def list_parts(self) -> None:
        print(f"Product
parts:\n{self.cpu}\n{self.ram}\n{self.display}\n{self.keyboard}")


class MacComputer(Computer):
    def __init__(self) -> None:
        super().__init__()
        self.cpu = 'A13'


class LenovoComputer(Computer):
    def __init__(self) -> None:
        super().__init__()
        self.cpu = 'Intel_i8'


class Director:

    def __init__(self) -> None:
        self._builder = None

    @property
    def builder(self) -> Builder:
        return self._builder
```

```
    @builder.setter
    def builder(self, builder: Builder) -> None:
        self._builder = builder

    def build_full_computer(self) -> None:
        self.builder.produce_keyboard()
        self.builder.produce_ram()
        self.builder.produce_cpu()
        self.builder.produce_display()


if __name__ == "__main__":
    director = Director()

    print("Basic product MacComputer")
    builder = MacComputerBuilder()
    director.builder = builder
    director.build_full_computer()
    builder.product.list_parts()

    print("Basic product LenovoComputer")
    builder2 = LenovoComputerBuilder()
    director.builder = builder2
    director.build_full_computer()
    builder2.product.list_parts()
```

原型模式 Prototype (Clone)

原型模式是一种创建型设计模式， 使你能够复制已有对象， 而又无需使代码依赖它们所属的类。

The prototype pattern is a creative design pattern that allows you to copy existing objects without having to make your code dependent on the class to which they belong.



1. **原型** （Prototype） 接口将对克隆方法进行声明。 在绝大多数情况下， 其中只会有一个名为 `clone` 克隆的方法。
2. **具体原型** （Concrete Prototype） 类将实现克隆方法。 除了将原始对象的数据复制到克隆体中之外， 该方法有时还需处理克隆过程中的极端情况， 例如克隆关联对象和梳理递归依赖等等。
3. **客户端** （Client） 可以复制实现了原型接口的任何对象。

应用场景

- 如果你需要复制一些对象， 同时又希望代码独立于这些对象所属的具体类， 可以使用原型模式。
- 如果子类的区别仅在于其对象的初始化方式， 那么你可以使用该模式来减少子类的数量。 别人创建这些子类的目的可能是为了创建特定类型的对象。

优缺点

- 你可以克隆对象， 而无需与它们所属的具体类相耦合。
- 你可以克隆预生成原型， 避免反复运行初始化代码。
- 你可以更方便地生成复杂对象。
- 你可以用继承以外的方式来处理复杂对象的不同配置。

Dis

- 克隆包含循环引用的复杂对象可能会非常麻烦。

关系

- 在许多设计工作的初期都会使用工厂方法模式 （较为简单， 而且可以更方便地通过子类进行定制）， 随后演化为使用抽象工厂模式、 原型模式或生成器模式 （更灵活但更加复杂）。
- 原型并不基于继承， 因此没有继承的缺点。 另一方面， 原型需要对被复制对象进行复杂的初始化。 工厂方法基于继承， 但是它不需要初始化步骤。

Code ★

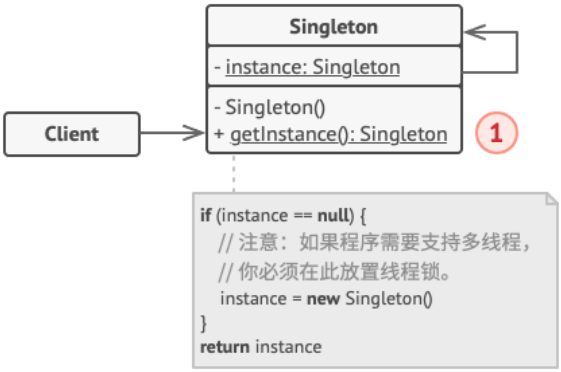
```
import copy


class Log:
    def clone(self):
        return copy.deepcopy(self)


class WeeklyLog(Log):
    __name = ""
    __date = "2000-01-02"
    __content = ""

    def __init__(self, name, date, content):
        self.__name = name
        self.__date = date
        self.__content = content

    def display(self):
        print(f"{self.__name} {self.__date} {self.__content}")

    def set_date(self, date):
        self.__date = date


class Client:
    def operation():
        a = WeeklyLog('log1', '2000-02-19', 'fail!')

        b = a.clone()
```

```
        b.set_date('2012-12-21')

        a.display()
        b.display()


def main():
    clint = Client
    clint.operation()


if __name__ == '__main__':
    main()
```

Singleton 单例模式！

单例模式是一种创建型设计模式， 让你能够保证一个类只有一个实例， 并提供一个访问该实例的全局节点。

The singleton pattern is a creative design pattern that allows you to ensure that there is only one instance of a class and provide a global node to access that instance.



1. **单例** （Singleton） 类声明了一个名为 `getInstance` 获取实例的静态方法来返回其所属类的一个相同实例。

   单例的构造函数必须对客户端 （Client） 代码隐藏。 调用 `获取实例` 方法必须是获取单例对象的唯一方式。

场景

- 如果程序中的某个类对于所有客户端只有一个可用的实例， 可以使用单例模式。
- 如果你需要更加严格地控制全局变量， 可以使用单例模式。

优缺点

- 你可以保证一个类只有一个实例。
- 你获得了一个指向该实例的全局访问节点。
- 仅在首次请求单例对象时对其进行初始化。

Disadvantage

- 违反了单一职责原则。 该模式同时解决了两个问题。
- 单例模式可能掩盖不良设计， 比如程序各组件之间相互了解过多等。
- 该模式在多线程环境下需要进行特殊处理， 避免多个线程多次创建单例对象。
- 单例的客户端代码单元测试可能会比较困难， 因为许多测试框架以基于继承的方式创建模拟对象。由于单例类的构造函数是私有的， 而且绝大部分语言无法重写静态方法， 所以你需要想出仔细考

虑模拟单例的方法。 要么干脆不编写测试代码， 或者不使用单例模式。

关系

- 外观模式类通常可以转换为单例模式类， 因为在大部分情况下一个外观对象就足够了。
- 如果你能将对象的所有共享状态简化为一个享元对象， 那么享元模式就和单例类似了。 但这两个模式有两个根本性的不同。
    1. 只会有一个单例实体， 但是享元类可以有多个实体， 各实体的内在状态也可以不同。
    2. 单例对象可以是可变的。 享元对象是不可变的。

Code ★

```
class Singleton {
    private static instance: Singleton;

    private constructor() { }

    public static getInstance(): Singleton {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }

        return Singleton.instance;
    }

    public someBusinessLogic() {
    }
}

function clientCode() {
    const s1 = Singleton.getInstance();
    const s2 = Singleton.getInstance();

    if (s1 === s2) {
        console.log('Singleton works, both variables contain the same
instance.');
    } else {
        console.log('Singleton failed, variables contain different instances.');
    }
}

clientCode();
```