目录

学习设计模式的时机与技巧

1. 建议在没有足够项目经验时可以先过一遍，了解基本的概念
2. 当积累一些工作经验后，可以挑选常见重点设计模式进行理解并可以选择历年的考题与项目加深理解

设计模式的原则

- 单一职责原则
  单一职责原则(Single Responsibility Principle, SRP)：一个类只负责一个功能领域中的相应职责，或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。
  单一职责原则告诉我们：一个类不能太“累”！在软件系统中，一个类（大到模块，小到方法）承担的职责越多，它被复用的可能性就越小，而且一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作，因此要将这些职责进行分离，将不同的职责封装在不同的类中，即将不同的变化原因封装在不同的类中，如果多个职责总是同时发生改变则可将它们封装在同一类中。
  单一职责原则是实现高内聚、低耦合的指导方针，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关实践经验

- 开闭原则(Open-Closed Principle, OCP)
  一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展。
  在开闭原则的定义中，软件实体可以指一个软件模块、一个由多个类组成的局部结构或一个独立的类。
  为了满足开闭原则，需要对系统进行抽象化设计，抽象化是开闭原则的关键。

- 里氏替换原则
  里氏代换原则(Liskov Substitution Principle, LSP)：所有引用基类（父类）的地方必须能透明地使用其子类的对象
  里氏代换原则告诉我们，在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。例如：我喜欢动物，那我一定喜欢狗，因为狗是动物的子类；但是我喜欢狗，不能据此断定我喜欢动物，因为我并不喜欢老鼠，虽然它也是动物.
  里氏代换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此**在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象**

- 依赖倒转原则(Dependency Inversion Principle, DIP)：
  抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。
  依赖倒转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。为了确保该原则的应用，一个具体类应当只实现接口或抽象类中声明过的方法，而不要给出多余的方法，否则将无法调用到在子类中增加的新方法
  在实现依赖倒转原则时，我们需要针对抽象层编程，而将具体类的对象通过依赖注入(DependencyInjection, DI)的方式注入到其他对象中，依赖注入是指当一个对象要与其他对象发生依赖关系时，通过抽象来注入所依赖的对象。常用的注入方式有三种，分别是：构造注入，设值注入（Setter注入）和接口注入。构造注入是指通过构造函数来传入具体类的对象，设值注入是指通过Setter方法来传入具体类的对象，而接口注入是指通过在接口中声明的业务方法来传入具体类的对象。这些方法在定义时使用的是抽象类型，在运行时再传入具体类型的对象，由子类对象来覆盖父类对象

- 接口隔离原则
  接口隔离原则(Interface Segregation Principle, ISP)：使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口
  接口仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。在面向对象编程语言中，实现一个接口就需要实现该接口中定义的所有方法，因此大的总接口使用起来不一定很方便，为了使接口的职责单一，需要将大接口中的方法根据其职责不同分别放在不同的小接口中，以确保每个接口使用起来都较为方便，并都承担某一单一角色。接口应该尽量细化，同时接口中的方法应该尽量少，每个接口中只包含一个客户端（如子模块或业务逻辑类）所需的方法即可，这种机制也称为“定制服务”，即为不同的客户端提供宽窄不同的接口。

- 迪米特法则
  一个软件实体应当尽可能少地与其他实体发生相互作用。
  迪米特法则要求我们在设计系统时，应该尽量减少对象之间的交互，如果两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的一个对象需要调用另一个对象的某一个方法的话，可以通过第三者转发这个调用。简言之，就是通过引入一个合理的第三者来降低现有对象之间的耦合度。
  在将迪米特法则运用到系统设计中时，要注意下面的几点：在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及；在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；在类的设计上，只要有可能，一个类型应当设计成不变类；在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

- 推设计模式原则讲解比较好的文章链接推荐

设计模式的介绍

第一章 创建型模式

- 1.1 工厂模式(工厂方法、抽象工厂、单例模式及三者的区别)(重点)
- 1.2 建造者模式
- 1.3 原型模式

第二章 结构型模式

- 2.1 适配器模式(重点)
- 2.2 修饰器模式(重点)
- 2.3 外观模式(重点)
- 2.4 享元模式(重点)
- 2.5 MVC模型(重点)
- 2.6 代理设计模式（Proxy design pattern）(重点)
  - 保护/防护代理:用于对处理敏感信息的对象进行访问控制
  - 远程代理：代表一个活跃于远程位置（例如，我们自己的远程服务器或云服务）的对象

- 虚拟代理：将一个对象的初始化延迟到真正需要使用时进行
- 智能（引用）代理：通过添加帮助信息（比如，引用计数）来扩展一个对象的行为时，可以使用智能（引用）代理

第三章 行为型模式

第一章 创建型模式

返回目录

1.1 工厂模式

- **使用场景：**
- **设计思路：**
- **示例代码：**
  - **工厂方法**

```
 import xml.etree.ElementTree as etree
 import json


class JSONConnector:
  def __init__(self, filepath):
      self.data = dict()
      with open(filepath, mode='r', encoding='utf-8') as f:
          self.data = json.load(f)

  @property
  def parsed_data(self):
      return self.data


class XMLConnector:
  def __init__(self, filepath):
      self.tree = etree.parse(filepath)

  @property
  def parsed_data(self):
      return self.tree


def connection_factory(filepath):
  if filepath.endswith('json'):
      connector = JSONConnector
  elif filepath.endswith('xml'):
      connector = XMLConnector
  else:
      raise ValueError('Cannot connect to {}'.format(filepath))
  return connector(filepath)


def connect_to(filepath):
  factory = None
  try:
      factory = connection_factory(filepath)
  except ValueError as ve:
      print(ve)
  return factory


def main():
  sqlite_factory = connect_to('data/person.sq3')
  print('====================')
  xml_factory = connect_to('data/person.xml')
  xml_data = xml_factory.parsed_data
  liars = xml_data.findall(".//{}[{}='{}']".format('person',
                                                   'lastName', 'Liar'))
  print('found: {} persons'.format(len(liars)))
  for liar in liars:
      print('first name: {}'.format(liar.find('firstName').text))
      print('last name: {}'.format(liar.find('lastName').text))
      [print('phone number ({})'.format(p.attrib['type']), p.text) for p in liar.find('phoneNumbers')]
  print('==================')
  json_factory = connect_to('data/donut.json')
  json_data = json_factory.parsed_data
  print('found: {} donuts'.format(len(json_data)))

  for donut in json_data:
      print('name: {}'.format(donut['name']))
      print('price: ${}'.format(donut['ppu']))
      [print('topping: {} {}'.format(t['id'], t['type'])) for t in donut['topping']]


if __name__ == '__main__':
  main()</code></pre>
```

  - 抽象工厂

```
class Frog:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        print('{} the Frog encounters {} and {}!'.format(self, obstacle, obstacle.action()))


class Bug:
    def __str__(self):
        return 'a bug'
```

```
        def action(self):
            return 'eats it'


    class FrogWorld:
        def __init__(self, name):
            print(self)
            self.player_name = name

        def __str__(self):
            return '\n\n\t------ Frog World -------'

        def make_character(self):
            return Frog(self.player_name)

        def make_obstacle(self):
            return Bug()


    class Wizard:
        def __init__(self, name):
            self.name = name

        def __str__(self):
            return self.name

        def interact_with(self, obstacle):
            print('{} the Wizard battles against {} and {}!'.format(self, obstacle, obstacle.action()))


    class Ork:
        def __str__(self):
            return 'an evil ork'

        def action(self):
            return 'kills it'


    class WizardWorld:
        def __init__(self, name):
            print(self)
            self.player_name = name

        def __str__(self):
            return '\n\n\t------ Wizard World -------'

        def make_character(self):
            return Wizard(self.player_name)

        def make_obstacle(self):
            return Ork()


    class GameEnvironment:
        def __init__(self, factory):
            self.hero = factory.make_character()
            self.obstacle = factory.make_obstacle()

        def play(self):
            self.hero.interact_with(self.obstacle)


    def validate_age(name):
        try:
            age = input('Welcome {}. How old are you? '.format(name))
            age = int(age)
        except ValueError as err:
            print("Age {} is invalid, please try again...".format(age))
            return (False, age)
        return (True, age)


    def main():
        name = input("Hello. What's your name? ")
        valid_input = False
        while not valid_input:
            valid_input, age = validate_age(name)
        game = FrogWorld if age < 18 else WizardWorld
        environment = GameEnvironment(game(name))
        environment.play()

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

- 单利模式

python

- **三种模式的对比总结：**

1.2 建造者模式

- **使用场景：**
- **设计思路：**
- **示例代码：**

```
from enum import Enum
import time

PizzaProgress = Enum('PizzaProgress', 'queued preparation baking ready')
PizzaDough = Enum('PizzaDough', 'thin thick')
PizzaSauce = Enum('PizzaSauce', 'tomato creme_fraiche')
PizzaTopping = Enum('PizzaTopping', 'mozzarella double_mozzarella bacon ham mushroomsred_onionoregano')
STEP_DELAY = 3  # 考虑到这是示例，单位为秒


class Pizza:
    def __init__(self, name):
        self.name = name
```

```
        self.dough = None
        self.sauce = None
        self.topping = []

    def __str__(self):
        return self.name

    def prepare_dough(self, dough):
        self.dough = dough
        print('preparing the {} dough of your {}...'.format(self.dough.name, self))
        time.sleep(STEP_DELAY)
        print('done with the {} dough'.format(self.dough.name))


class MargaritaBuilder:
    def __init__(self):
        self.pizza = Pizza('margarita')
        self.progress = PizzaProgress.queued
        self.baking_time = 5  # 考虑是示例，单位为秒

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thin)

    def add_sauce(self):
        print('adding the tomato sauce to your margarita...')
        self.pizza.sauce = PizzaSauce.tomato
        time.sleep(STEP_DELAY)
        print('done with the tomato sauce')

    def add_topping(self):
        print('adding the topping (double mozzarella, oregano) to your margarita')
        self.pizza.topping.append([i for i in (PizzaTopping.double_mozzarella, PizzaTopping.oregano)])
        time.sleep(STEP_DELAY)
        print('done with the topping (double mozzarrella, oregano)')

    def bake(self):
        self.progress = PizzaProgress.baking
        print('baking your margarita for {} seconds'.format(self.baking_time))
        time.sleep(self.baking_time)
        self.progress = PizzaProgress.ready
        print('your margarita is ready')


class CreamyBaconBuilder:
    def __init__(self):
        self.pizza = Pizza('creamy bacon')
        self.progress = PizzaProgress.queued
        self.baking_time = 7  # 考虑是示例，单位为秒

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thick)

    def add_sauce(self):
        print('adding the crème fraîche sauce to your creamy bacon')
        self.pizza.sauce = PizzaSauce.creme_fraiche
        time.sleep(STEP_DELAY)
        print('done with the crème fraîche sauce')

    def add_topping(self):
        print('adding the topping (mozzarella, bacon, ham, mushrooms, red onion,oregano) to your creamy baon')
        self.pizza.topping.append([t for t in
                                   (PizzaTopping.mozzarella, PizzaTopping.bacon,
                                    PizzaTopping.ham, PizzaTopping.mushrooms,
                                    PizzaTopping.red_onion, PizzaTopping.oregano)])
        time.sleep(STEP_DELAY)
        print('done with the topping (mozzarella, bacon, ham, mushrooms, red onion,oregano)')

    def bake(self):
        self.progress = PizzaProgress.baking
        print('baking your creamy bacon for {} seconds'.format(self.baking_time))
        time.sleep(self.baking_time)
        self.progress = PizzaProgress.ready
        print('your creamy bacon is ready')


class Waiter:
    def __init__(self):
        self.builder = None

    def construct_pizza(self, builder):
        self.builder = builder
        [step() for step in (builder.prepare_dough, builder.add_sauce, builder.add_topping, builder.bake)]

    @property
    def pizza(self):
        return self.builder.pizza


def validate_style(builders):
    try:
        pizza_style = input('What pizza would you like, [m]argarita or [c]reamy bacon?')
        builder = builders[pizza_style]()
        valid_input = True
    except KeyError as err:
        print('Sorry, only margarita (key m) and creamy bacon (key c) are available')
        return (False, None)
    return (True, builder)


def main():
    builders = dict(m=MargaritaBuilder, c=CreamyBaconBuilder)
    valid_input = False
    while not valid_input:
        valid_input, builder = validate_style(builders)
    print('==============')
    waiter = Waiter()
    waiter.construct_pizza(builder)
    pizza = waiter.pizza
    print('====================')
    print('Enjoy your {}!'.format(pizza))


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

1.3 原型模式

- 使用场景：我们已有一个对象，并希望创建该对象的一个完整副本时，原型模式就派上用场了
- 设计思路：原型设计模式（Prototype design pattern）帮助我们创建对象的克隆，其最简单的形式就是一个clone()函数，接受一个对象作为输入参数，返回输入对象的一个副本
- 示例代码：

第二章 结构型模式

返回目录

2.1 适配器模型

- 遵守的设计原则：开放/封闭原则，对扩展是开放的，对修改则是封闭的
- 不同方法实现适配器模式的示例代码：
  - 使用了类的内部字典实现适配器

```
class Synthesizer:
  def __init__(self, name):
      self.name = name

 def __str__(self):
     return 'the {} synthesizer'.format(self.name)

 def play(self):
     return 'is playing an electronic song'


class Human:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return '{} the human'.format(self.name)

    def speak(self):
        return 'says hello'


class Computer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'the {} computer'.format(self.name)

    def execute(self):
        return 'executes a program'


class Adapter:
    def __init__(self, obj, adapted_methods):
        self.obj = obj
        # print(adapted_methods)
        # print(self.__dict__)
        print(self.__dict__.update(adapted_methods))
        # print(self.__dict__)

    # 定义此方法，可以访问后续的i.name，
    # 这是__getattr__的很重要的用法
    def __getattr__(self, item):
        return getattr(self.obj, item)

    def __str__(self):
        return str(self.obj)


def main():
    objects = [Computer('Asus')]
    synth = Synthesizer('moog')
    objects.append(Adapter(synth, dict(execute=synth.play)))

    human = Human('Bob')
    objects.append(Adapter(human, dict(execute=human.speak)))
    for i in objects:
        print('{} {}'.format(str(i), i.execute()))
        print(i.name)


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

  - 使用继承实现适配器
- 不同方法的特点 优缺点
- 主要设计思路：
- 代码示例：

```
    class Synthesizer:
        def __init__(self, name):
            self.name = name

      def __str__(self):
          return 'the {} synthesizer'.format(self.name)
```

```
        def play(self):
            return 'is playing an electronic song'


    class Human:
        def __init__(self, name):
            self.name = name

        def __str__(self):
            return '{} the human'.format(self.name)

        def speak(self):
            return 'says hello'


    class Computer:
        def __init__(self, name):
            self.name = name

        def __str__(self):
            return 'the {} computer'.format(self.name)

        def execute(self):
            return 'executes a program'


    class Adapter:
        def __init__(self, obj, adapted_methods):
            self.obj = obj
            # print(adapted_methods)
            # print(self.__dict__)
            print(self.__dict__.update(adapted_methods))
            # print(self.__dict__)

        def __getattr__(self, item):
            return getattr(self.obj, item)

        def __str__(self):
            return str(self.obj)


    def main():
        objects = [Computer('Asus')]
        synth = Synthesizer('moog')
        objects.append(Adapter(synth, dict(execute=synth.play)))

        human = Human('Bob')
        objects.append(Adapter(human, dict(execute=human.speak)))
        for i in objects:
            print('{} {}'.format(str(i), i.execute()))
            print(i.name)


    if __name__ == "__main__":
        main()</code></pre></li>
```

返回目录

2.2 修饰器模型

- **遵守的设计原则：**
- **对一个对象添加额外的功能的方法有：**
  - **直接将功能添加到对象所属的类**
  - **组合:组合优于继承**
  - **继承**
  - **修饰器**
- **使用的场景：缓存，性能测试，权限认证**
- **主要设计思路：**
- **代码示例：**
  - **缓存**

```
# 斐波那契数列，以下方法计算比较耗时
def fibonacci(n):
  assert (n >= 0), 'n must be >= 0'
  return n if n in (0, 1) else fibonacci(n - 1) + fibonacci(n - 2)


if __name__ == '__main__':
    from timeit import Timer

    t = Timer('fibonacci(8)', 'from __main__ import fibonacci')
    print(t.timeit())  # 19.1498459


# 使用字典缓存计算结果进行计算提速
known = {0: 0, 1: 1}
def fibonacci(n):
    assert (n >= 0), 'n must be >= 0'
    if n in known:
        return known[n]
    res = fibonacci(n - 1) + fibonacci(n - 2)
    known[n] = res
    return res


if __name__ == '__main__':
    from timeit import Timer

    t = Timer('fibonacci(100)', 'from __main__ import fibonacci')
    print(t.timeit())  # 0.4535593


# 使用装饰器进行优化：
```

```
import functools


def memoize(fn):
  known = dict()

  @functools.wraps(fn)
  def memoizer(*args):
      if args not in known:
          known[args] = fn(*args)
      return known[args]

  return memoizer



@memoize
def fibonacci(n):
  '''返回斐波那契数列的第n个数'''
  assert (n >= 0), 'n must be >= 0'
  return n if n in (0, 1) else fibonacci(n - 1) + fibonacci(n - 2)


if __name__ == '__main__':
  from timeit import Timer

  measure = [{'exec': 'fibonacci(100)', 'import': 'fibonacci','func': fibonacci}]
  for m in measure:
      t = Timer('{}'.format(m['exec']), 'from __main__ import {}'.format(m['import']))
      print('name: {}, doc: {}, executing: {}, time:{}'.format(m['func'].__name__, m['func'].__doc__, m['exec'],
                                                               t.timeit()))
# name: fibonacci, doc: 返回斐波那契数列的第n个数, executing: fibonacci(100), time:0.4526535
```

- **性能测试**
- **权限验证**

返回目录

2.3 外观模式

- **遵守的设计原则：**
- **使用的场景：**
  i. **系统包含多层，外观模式也能派上用场。你可以为每一层引入一个外观入口点，并让所有层级通过它们的外观相互通信。这提高了层级之间的松耦合性，尽可能保持层级独立**
- **主要设计思路：外观设计模式有助于隐藏系统的内部复杂性，并通过一个简化的接口向客户端暴露必要的部分**
- **代码示例：**

```
 from enum import Enum
 from abc import ABCMeta, abstractmethod

State = Enum('State', 'new running sleeping restart zombie')


class User:
    pass


class Process:
    pass


class File:
    pass


class Server(metaclass=ABCMeta):
    @abstractmethod
    def __init__(self):
        pass

    def __str__(self):
        return self.name

    @abstractmethod
    def boot(self):
        pass

    @abstractmethod
    def kill(self, restart=True):
        pass


class FileServer(Server):
    def __init__(self):
        '''初始化文件服务进程要求的操作'''
        self.name = 'FileServer'
        self.state = State.new

    def boot(self):
        print('booting the {}'.format(self))
        '''启动文件服务进程要求的操作'''
        self.state = State.running

    def kill(self, restart=True):
        print('Killing {}'.format(self))
        '''终止文件服务进程要求的操作'''
        self.state = State.restart if restart else State.zombie

    def create_file(self, user, name, permissions):
        '''检查访问权限的有效性、用户权限等'''
        print("trying to create the file '{0}' for user '{1}' with permissions{2}".format(name, user, permissions))


class ProcessServer(Server):
    def __init__(self):
        '''初始化进程服务进程要求的操作'''
        self.name = 'ProcessServer'
        self.state = State.new
```

```
        def boot(self):
            print('booting the {}'.format(self))
            '''启动进程服务进程要求的操作'''
            self.state = State.running

        def kill(self, restart=True):
            print('Killing {}'.format(self))
            '''终止进程服务进程要求的操作'''
            self.state = State.restart if restart else State.zombie

        def create_process(self, user, name):
            '''检查用户权限和生成PID等'''
            print("trying to create the process '{}' for user '{}'".format(name, user))


    class WindowServer:
        pass


    class NetworkServer:
        pass


    class OperatingSystem:
        '''外观'''

        def __init__(self):
            self.fs = FileServer()
            self.ps = ProcessServer()

        def start(self):
            [i.boot() for i in (self.fs, self.ps)]

        def create_file(self, user, name, permissions):
            return self.fs.create_file(user, name, permissions)

        def create_process(self, user, name):
            return self.ps.create_process(user, name)


    def main():
        os = OperatingSystem()
        os.start()
        os.create_file('foo', 'hello', '-rw-r-r-')
        os.create_process('bar', 'ls /tmp')


    if __name__ == '__main__':
        main()</code></pre></li>
```

返回目录

2.4 享元模式

- **遵守的设计原则：**
- **使用的场景：享元模式是一个用于优化性能和内存使用的设计模式**
- **主要设计思路：享元设计模式通过为相似对象引入数据共享来最小化内存使用，提升性能。一个享元（Flyweight）就是一个包含状态独立的不可变（又称固有的）数据的共享对象。依赖状态的可变（又称非固有的）数据不应是享元的一部分，因为每个对象的这种信息都不同，无法共享。如果享元需要非固有的数据，应该由客户端代码显式地提供**
- **代码示例：**

```
 from enum import Enum
 import random

TreeType = Enum('TreeType', 'apple_tree cherry_tree peach_tree')


# print(TreeType.apple_tree.value)
# print(TreeType.cherry_tree.value)

class Tree:
    pool = dict()

    def __new__(cls, tree_type):
        obj = cls.pool.get(tree_type, None)
        if not obj:
            print('{0}不存在'.format(tree_type))
            obj = object.__new__(cls)
            cls.pool[tree_type] = obj
            print(cls.pool)
            obj.tree_type = tree_type
        else:
            print('{0}已存在'.format(tree_type))
        return obj

    def render(self, age, x, y):
        print('render a tree of type {} and age {} at ({}, {})'.format(self.tree_type,
                                                                       age, x, y))


def main():
    rnd = random.Random()
    age_min, age_max = 1, 30  # 单位为年
    min_point, max_point = 0, 100
    tree_counter = 0
    for _ in range(10):
        t1 = Tree(TreeType.apple_tree)
        t1.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1
    for _ in range(3):
        t2 = Tree(TreeType.cherry_tree)
        t2.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1
    for _ in range(5):
        t3 = Tree(TreeType.peach_tree)
        t3.render(rnd.randint(age_min, age_max),
                  rnd.randint(min_point, max_point),
                  rnd.randint(min_point, max_point))
        tree_counter += 1

    print('trees rendered: {}'.format(tree_counter))
    print('trees actually created: {}'.format(len(Tree.pool)))
    t4 = Tree(TreeType.cherry_tree)
    t5 = Tree(TreeType.cherry_tree)
```

```
        t6 = Tree(TreeType.apple_tree)
        print('{} == {}? {}'.format(id(t4), id(t5), id(t4) == id(t5)))
        print('{} == {}? {}'.format(id(t5), id(t6), id(t5) == id(t6)))


    if __name__ == '__main__':
        main()</code></pre></li>
```

返回目录

2.5 MVC模型

- 概念：模型-视图-控制器

返回目录

2.6 代理设计模式

- 使用的场景
- 设计思路：
- 代码示例：

```
class SensitiveInfo:
  def __init__(self):
      self.users = ['nick', 'tom', 'ben', 'mike']

  def read(self):
      print('There are {} users: {}'.format(len(self.users), ' '.join(self.users)))

  def add(self, user):
      self.users.append(user)
      print('Added user {}'.format(user))


class Info:
  def __init__(self):
      self.protected = SensitiveInfo()
      self.secret = '0xdeadbeef'

  def read(self):
      self.protected.read()

  def add(self, user):
      sec = input('what is the secret? ')
      self.protected.add(user) if sec == self.secret else print("That's wrong!")


def main():
  info = Info()
  while True:
      print('1. read list |==| 2. add user |==| 3. quit')
      key = input('choose option: ')
      if key == '1':
          info.read()
      elif key == '2':
          name = input('choose username: ')
          info.add(name)
      elif key == '3':
          exit()
      else:
          print('unknown option: {}'.format(key))


# 此模式需要优化，存在缺陷
if __name__ == '__main__':
  main()
```

行为型模式

返回目录

3.1 责任链模式

- 使用的场景：
- 主要设计思路：
- 代码示例：

```

```

返回目录

3.2 命令模式

- 使用的场景：
- 主要设计思路：
- 代码示例：

```
import os

verbose = True
```

```
class RenameFile:
    def __init__(self, path_src, path_dest):
        self.src, self.dest = path_src, path_dest

    def execute(self):
        if verbose:
            print("[renaming '{}' to '{}']".format(self.src, self.dest))
        os.rename(self.src, self.dest)

    def undo(self):
        if verbose:
            print("[renaming '{}' back to '{}']".format(self.dest, self.src))
        os.rename(self.dest, self.src)


class CreateFile:
    def __init__(self, path, txt='hello world\n'):
        self.path, self.txt = path, txt

    def execute(self):
        if verbose:
            print("[creating file '{}']".format(self.path))
        with open(self.path, mode='w') as out_file:
            out_file.write(self.txt)

    def undo(self):
        delete_file(self.path)


class ReadFile:
    def __init__(self, path):
        self.path = path

    def execute(self):
        if verbose:
            print("[reading file '{}']".format(self.path))
        with open(self.path, mode='r') as in_file:
            print(in_file.read())


def delete_file(path):
    if verbose:
        print("deleting file '{}'".format(path))
    os.remove(path)


def main():
    orig_name, new_name = 'file1', 'file2'
    commands = []
    for cmd in CreateFile(orig_name), ReadFile(orig_name), RenameFile(orig_name, new_name):
        commands.append(cmd)
        [c.execute() for c in commands]
        answer = input('reverse the executed commands? [y/n] ')
    if answer not in 'yY':
        print("the result is {}".format(new_name))
        exit()
    for c in reversed(commands):
        try:
            c.undo()
        except AttributeError as e:
            pass


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

返回目录

3.3 解释器模式 这里暂不介绍

- **使用的场景：解释器（Interpreter）模式仅能引起应用的高级用户的兴趣**
- **主要设计思路：**
- **代码示例：**

```
```

返回目录

3.4 观察者模式

- **使用的场景：消息发布 订阅**
- **主要设计思路：被观察者被多个观察者观察着，被观察者有变化，则会处发导致所有观察者也变化**
- **代码示例：**

```
class Publisher:
    def __init__(self):
        self.observers = []

    def add(self, observer):
        if observer not in self.observers:
            self.observers.append(observer)
        else:
            print('Failed to add: {}'.format(observer))

    def remove(self, observer):
        try:
            self.observers.remove(observer)
        except ValueError:
            print('Failed to remove: {}'.format(observer))

    def notify(self):
        print('notify')
        [o.notify(self) for o in self.observers]
```

```
class DefaultFormatter(Publisher):
    def __init__(self, name):
        Publisher.__init__(self)
        self.name = name
        self._data = 0

    def __str__(self):
        return "{}: '{}' has data = {}".format(type(self).__name__, self.name, self._data)

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, new_value):
        try:
            self._data = int(new_value)
        except ValueError as e:
            print('Error: {}'.format(e))
        else:
            self.notify()


class HexFormatter:
    def notify(self, publisher):
        print("{}: '{}' has now hex data = {}".format(type(self).__name__, publisher.name, hex(publisher.data)))


class BinaryFormatter:
    def notify(self, publisher):
        print("{}: '{}' has now bin data = {}".format(type(self).__name__, publisher.name, bin(publisher.data)))


def main():
    df = DefaultFormatter('test1')
    print(df)
    print('==========================')
    hf = HexFormatter()
    df.add(hf)
    hf2 = HexFormatter()
    df.add(hf2)
    df.data = 3

    # df.data = 5
    # print(df.observers)
    # print('==========================')
    # bf = BinaryFormatter()
    # df.add(bf)
    # df.data = 21
    # print(df)
    # print('==========================')
    # df.remove(hf)
    # df.data = 40
    # print(df)
    # print('==========================')
    # df.remove(hf)
    # df.add(bf)
    # df.data = 'hello'
    # print(df)
    # print('==========================')
    # df.data = 15.8
    # print(df)


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

返回目录

3.5 状态模式 State design pattern

- **使用的场景：非计算机的例子包括自动售货机、电梯、交通灯、 暗码锁、停车计时器、自动加油泵及自然语言文法描述。计算机方面的例子包括游戏编程和计算机编程的其他领域、硬件设计、协议设计，以及编程语言解析**
- **主要设计思路：在一个事件驱动系统中，从一个状态转换到另一个状态会触发一个事件/消息。状态设计模式通常使用一个父State类和许多派生的ConcreteState类来实现，父类包含所有状态共同的功能，每个派生类则仅包含特定状态要求的功能。状态模式关注的是实现一个状态机，状机的核心部分是状态和状态之间的转换**
- **代码示例：**

返回目录

3.6 策略模式

- **使用的场景示例：**
  - **i. 使用百度去机场有不同的路线，基于不同的需求考量提供不同的路线**
- **主要的设计思路：**
  - **i. 使用多种算法来解决一个问题，其杀手级特性是能够在运行时透明地切换算法（客户端代码对变化无感知）**
- **对比进行更好的理解：与工厂模式(工厂方法与抽象工厂)进行对比**
- **不同的字符长度，通过不同的算法策略返回是否字母单一，代码示例**

```
import time

SLOW = 3  # 单位为秒
LIMIT = 5  # 字符数
WARNING = 'too bad, you picked the slow algorithm :('
```

```
def pairs(seq):
    n = len(seq)
    for i in range(n):
        yield seq[i], seq[(i + 1) % n]


def allUniqueSort(s):
    if len(s) > LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    srtStr = sorted(s)
    for (c1, c2) in pairs(srtStr):
        if c1 == c2:
            return False
    return True


def allUniqueSet(s):
    if len(s) < LIMIT:
        print(WARNING)
    time.sleep(SLOW)
    return True if len(set(s)) == len(s) else False


def allUnique(s, strategy):
    return strategy(s)


def main():
    while True:
        word = None
        while not word:
            word = input('Insert word (type quit to exit)> ')
            if word == 'quit':
                print('bye')
                return
            strategy_picked = None
            strategies = {'1': allUniqueSet, '2': allUniqueSort}
            while strategy_picked not in strategies.keys():
                strategy_picked = input('Choose strategy: [1] Use a set, [2] Sort andpair> ')
                try:
                    strategy = strategies[strategy_picked]
                    print('allUnique({}): {}'.format(word, allUnique(word,
                                                                     strategy)))
                except KeyError as err:
                    print('Incorrect option: {}'.format(strategy_picked))
                    print()

   if __name__ == '__main__':
       main()</code></pre></li>
```

返回目录

优秀资源

- 关于设计模式优秀资源
- 书籍资料(清单：，密码：)