原型模式

题目链接

原型模式-矩形原型

什么是原型模式

原型模式一种创建型设计模式，该模式的核心思想是基于现有的对象创建新的对象，而不是从头开始创建。

在原型模式中，通常有一个原型对象，它被用作创建新对象的模板。新对象通过复制原型对象的属性和状态来创建，而无需知道具体的创建细节。

为什么要使用原型模式

如果一个对象的创建过程比较复杂时（比如需要经过一系列的计算和资源消耗），那每次创建该对象都需要消耗资源，而通过原型模式就可以复制现有的一个对象来迅速创建/克隆一个新对象，不必关心具体的创建细节，可以降低对象创建的成本。

下面是一个简短的Python代码示例了模拟了上面的问题：

```
import copy

class ComplexObject:
    def __init__(self, data):
        # 耗时的资源型操作
        self.data = data

    def clone(self):
        # 复制
        return copy.deepcopy(self)
# 创建原型对象
original_object = ComplexObject(data="large date")
# 创建新对象，直接拷贝原对象
new_object = original_object.clone()
```

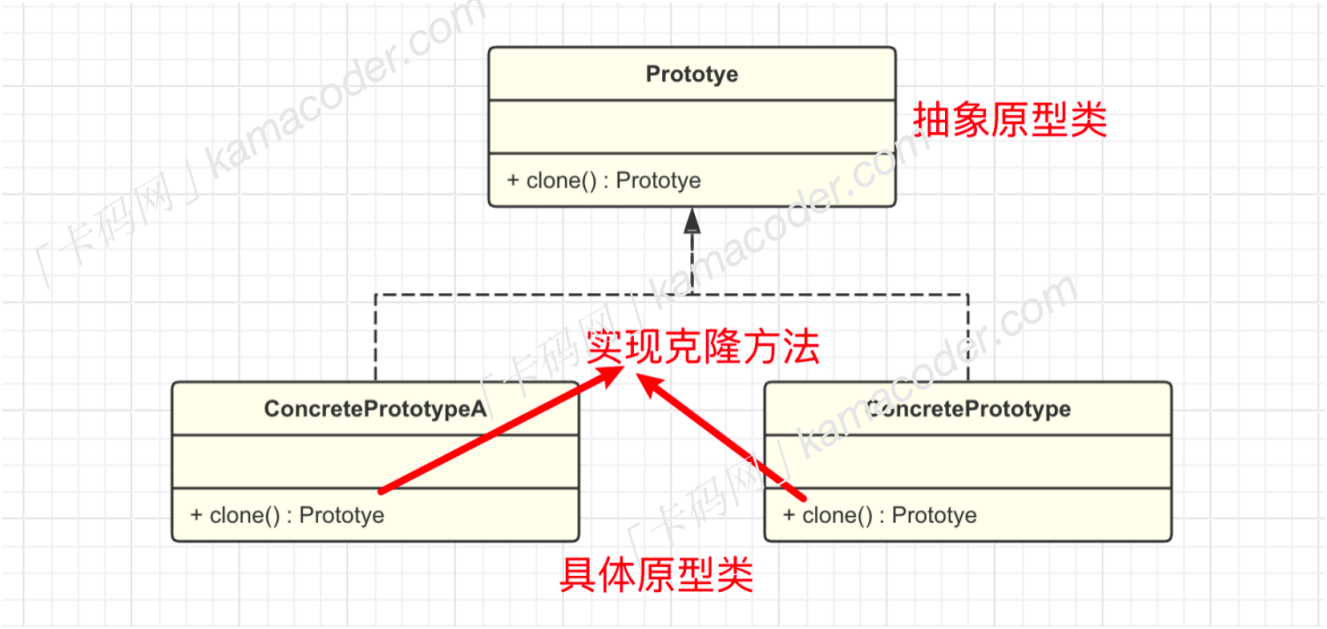
原型模式的基本结构

实现原型模式需要给【原型对象】声明一个克隆方法，执行该方法会创建一个当前类的新对象，并将原始对象中的成员变量复制到新生成的对象中，而不必实例化。并且在这个过程中只需要调用原型对象的克隆方法，而无需知道原型对象的具体类型。

原型模式包含两个重点模块：

- 抽象原型接口prototype：声明一个克隆自身的方法clone
- 具体原型类ConcretePrototype：实现clone方法，复制当前对象并返回一个新对象。

在客户端代码中，可以声明一个具体原型类的对象，然后调用clone()方法复制原对象生成一个新的对象。



原型模式的基本实现

原型模式的实现过程即上面描述模块的实现过程：

- 创建一个抽象类或接口，声明一个克隆方法clone
- 实现具体原型类，重写克隆方法
- 客户端中实例化具体原型类的对象，并调用其克隆方法来创建新的对象。

```
// 1. 定义抽象原型类
public abstract class Prototype implements Cloneable {
    public abstract Prototype clone();
}

// 2. 创建具体原型类
public class ConcretePrototype extends Prototype {
    private String data;

    public ConcretePrototype(String data) {
        this.data = data;
    }
```

```
    @Override
    public Prototype clone() {
        return new ConcretePrototype(this.data);
    }

    public String getData() {
        return data;
    }
}

// 3. 客户端代码
public class Client {
    public static void main(String[] args) {
        // 创建原型对象
        Prototype original = new ConcretePrototype("Original Data");

        // 克隆原型对象
        Prototype clone = original.clone();

        // 输出克隆对象的数据
        System.out.println("Clone Data: " + ((ConcretePrototype)
clone).getData());
    }
}
```

什么时候实现原型模式

相比于直接实例化对象，通过原型模式复制对象可以减少资源消耗，提高性能，尤其在对象的创建过程复杂或对象的创建代价较大的情况下。当需要频繁创建相似对象、并且可以通过克隆避免重复初始化工作的场景时可以考虑使用原型模式，在克隆对象的时候还可以动态地添加或删除原型对象的属性，创造出相似但不完全相同的对象，提高了灵活性。

但是使用原型模式也需要考虑到如果对象的内部状态包含了引用类型的成员变量，那么实现深拷贝就会变得较为复杂，需要考虑引用类型对象的克隆问题。

原型模式在现有的很多语言中都有应用，比如以下几个经典例子。

- Java 提供了 `Object` 类的 `clone()` 方法，可以实现对象的浅拷贝。类需要实现 `Cloneable` 接口并重写 `clone()` 方法。
- 在 .NET 中，`ICloneable` 接口提供了 `Clone` 方法，可以用于实现对象的克隆。
- Spring 框架中的 Bean 的作用域之一是原型作用域（Prototype Scope），在这个作用域下，Spring 框架会为每次请求创建一个新的 Bean 实例，类似于原型模式。

本题代码

```
import java.util.Scanner;

// 抽象原型类
abstract class Prototype implements Cloneable {
    public abstract Prototype clone();
    public abstract String getDetails();

    // 公共的 clone 方法
    public Prototype clonePrototype() {
        try {
            return (Prototype) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }
}

// 具体矩形原型类
class RectanglePrototype extends Prototype {
    private String color;
    private int width;
    private int height;

    // 构造方法
    public RectanglePrototype(String color, int width, int height) {
        this.color = color;
        this.width = width;
        this.height = height;
    }

    // 克隆方法
    @Override
    public Prototype clone() {
        return clonePrototype();
    }

    // 获取矩形的详细信息
    @Override
    public String getDetails() {
        return "Color: " + color + ", Width: " + width + ", Height: " +
height;
    }
}
```

```
// 客户端程序
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取需要创建的矩形数量
        int N = scanner.nextInt();

        // 读取每个矩形的属性信息并创建矩形对象
        for (int i = 0; i < N; i++) {
            String color = scanner.next();
            int width = scanner.nextInt();
            int height = scanner.nextInt();

            // 创建原型对象
            Prototype originalRectangle = new RectanglePrototype(color,
width, height);

            // 克隆对象并输出详细信息
            Prototype clonedRectangle = originalRectangle.clone();
            System.out.println(clonedRectangle.getDetails());
        }
    }
}
```

其他语言代码

Java

使用原型注册表来实现

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

// 原型模式抽象类Shape，表示形状的基类
abstract class Shape {
    protected int height;
    protected int width;
    protected String color;

    public Shape() {}

    public Shape(int height, int width, String color) {
```

```
        this.height = height;
        this.width = width;
        this.color = color;
    }

    // 抽象方法 clone，用于克隆形状
    public abstract Shape clone();

    //get、set方法。
    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}

// Rectangle 类，继承自 Shape 表示矩形
class Rectangle extends Shape {
    public Rectangle(int height, int width, String color) {
        super(height, width, color);
    }

    // 克隆方法，返回一个新的矩形对象
    @Override
    public Shape clone() {
        return new Rectangle(this.height, this.width, this.color);
    }
}
```

```
// 原型注册表类，用于存储和管理形状
class ShapeRegistry {
    private Map<String, Shape> shapeMap = new HashMap<>();

    // 注册形状，使用形状 ID 作为键
    public void registerShape(String shapeId, Shape shape) {
        shapeMap.put(shapeId, shape);
    }
    // 获取形状，通过克隆返回一个新的对象
    public Shape getShape(String shapeId) {
        Shape shape = shapeMap.get(shapeId);
        return shape != null ? shape.clone() : null;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        ShapeRegistry shapeRegistry = new ShapeRegistry();

        // 注册一个默认的矩形形状
        shapeRegistry.registerShape("defaultRectangle", new Rectangle(10,
20, "red"));

        String input = scanner.nextLine();
        String[] parts = input.split(" ");
        if (parts.length == 3) {
            String color = parts[0];
            try {
                int width = Integer.parseInt(parts[1]);
                int height = Integer.parseInt(parts[2]);

                //创建并注册一个新的矩形
                Shape newRectangle = new Rectangle(height, width, color);
                shapeRegistry.registerShape("userRectangle", newRectangle);

                int num = scanner.nextInt();
                for (int i = 0; i < num; i++) {
                    Shape clonedShape =
shapeRegistry.getShape("userRectangle");
                    if (clonedShape != null) {
                        System.out.println("Color: " +
clonedShape.getColor() +
                                ", Width: " + clonedShape.getWidth() +
                                ", Height: " + clonedShape.getHeight());
```

```
                    } else {
                        System.out.println("在注册表中找不到形状.");
                    }
                }
            } catch (NumberFormatException e) {
                System.out.println("输入错误，请重新输入.");
            }
        } else {
            System.out.println("输入格式错误，请按格式输入：颜色 宽度 高度.");
        }
        scanner.close();
    }
}
```

C++

```
#include <iostream>
#include <string>
#include <vector>

// 抽象原型类
class Prototype {
public:
    virtual Prototype* clone() const = 0;
    virtual std::string getDetails() const = 0;
    virtual ~Prototype() {}
};

// 具体矩形原型类
class RectanglePrototype : public Prototype {
private:
    std::string color;
    int width;
    int height;

public:
    // 构造方法
    RectanglePrototype(std::string color, int width, int height) :
color(color), width(width), height(height) {}

    // 克隆方法
    Prototype* clone() const override {
        return new RectanglePrototype(*this);
    }
```

```
    // 获取矩形的详细信息
    std::string getDetails() const override {
        return "Color: " + color + ", Width: " + std::to_string(width) + ",
Height: " + std::to_string(height);
    }
};

// 客户端程序
int main() {
    std::vector<Prototype*> rectangles;

    // 读取需要创建的矩形数量
    int N;
    std::cin >> N;

    // 读取每个矩形的属性信息并创建矩形对象
    for (int i = 0; i < N; i++) {
        std::string color;
        int width, height;

        std::cin >> color >> width >> height;

        // 创建原型对象
        Prototype* originalRectangle = new RectanglePrototype(color, width,
height);

        // 将原型对象保存到向量中
        rectangles.push_back(originalRectangle);
    }

    // 克隆对象并输出详细信息
    for (const auto& rectangle : rectangles) {
        Prototype* clonedRectangle = rectangle->clone();
        std::cout << clonedRectangle->getDetails() << std::endl;

        // 释放克隆对象的内存
        delete clonedRectangle;
    }

    // 释放原型对象的内存
    for (const auto& rectangle : rectangles) {
        delete rectangle;
    }

    return 0;
```

```
}
```

Python

```python
from abc import ABC, abstractmethod

# 抽象原型类
class Prototype(ABC):
    @abstractmethod
    def clone(self):
        pass

    @abstractmethod
    def get_details(self):
        pass

    # 公共的 clone 方法
    def clone_prototype(self):
        try:
            return self.clone()
        except Exception as e:
            print(e)
            return None

# 具体矩形原型类
class RectanglePrototype(Prototype):
    def __init__(self, color, width, height):
        self.color = color
        self.width = width
        self.height = height

    # 克隆方法
    def clone(self):
        return RectanglePrototype(self.color, self.width, self.height)

    # 获取矩形的详细信息
    def get_details(self):
        return f"Color: {self.color}, Width: {self.width}, Height:
{self.height}"

# 客户端程序
if __name__ == "__main__":
    # 读取需要创建的矩形数量
    N = int(input())

    # 读取每个矩形的属性信息并创建矩形对象
```

```
    for _ in range(N):
        line = input()
        parts = line.split()
        color = parts[0]
        width = int(parts[1])
        height = int(parts[2])

        # 创建原型对象
        original_rectangle = RectanglePrototype(color, width, height)

        # 克隆对象并输出详细信息
        cloned_rectangle = original_rectangle.clone()
        print(cloned_rectangle.get_details())
```

Go

```
package main

import (
    "fmt"
)

// 抽象原型类
type Prototype interface {
    clone() Prototype
    getDetails() string
}

// 具体矩形原型类
type RectanglePrototype struct {
    color  string
    width  int
    height int
}

// 构造方法
func NewRectanglePrototype(color string, width, height int)
*RectanglePrototype {
    return &RectanglePrototype{
        color:  color,
        width:  width,
        height: height,
    }
}

// 实现 Prototype 接口的 clone 方法
```

```
func (r *RectanglePrototype) clone() Prototype {
    return &RectanglePrototype{
        color:  r.color,
        width:  r.width,
        height: r.height,
    }
}

// 获取矩形的详细信息
func (r *RectanglePrototype) getDetails() string {
    return fmt.Sprintf("Color: %s, Width: %d, Height: %d", r.color,
r.width, r.height)
}

// 客户端程序
func main() {
    // 读取需要创建的矩形数量
    var N int
    fmt.Scan(&N)

    // 读取每个矩形的属性信息并创建矩形对象
    for i := 0; i < N; i++ {
        var color string
        var width, height int
        fmt.Scan(&color, &width, &height)

        // 创建原型对象
        originalRectangle := NewRectanglePrototype(color, width, height)

        // 克隆对象并输出详细信息
        clonedRectangle := originalRectangle.clone()
        fmt.Println(clonedRectangle.getDetails())
    }
}
```