建造者模式

题目链接

建造者模式-自行车加工

什么是建造者模式

建造者模式（也被成为生成器模式），是一种创建型设计模式，软件开发过程中有的时候需要创建很复杂的对象，而建造者模式的主要思想是**将对象的构建过程分为多个步骤，并为每个步骤定义一个抽象的接口。具体的构建过程由实现了这些接口的具体建造者类来完成。**同时有一个指导者类负责协调建造者的工作，按照一定的顺序或逻辑来执行构建步骤，最终生成产品。

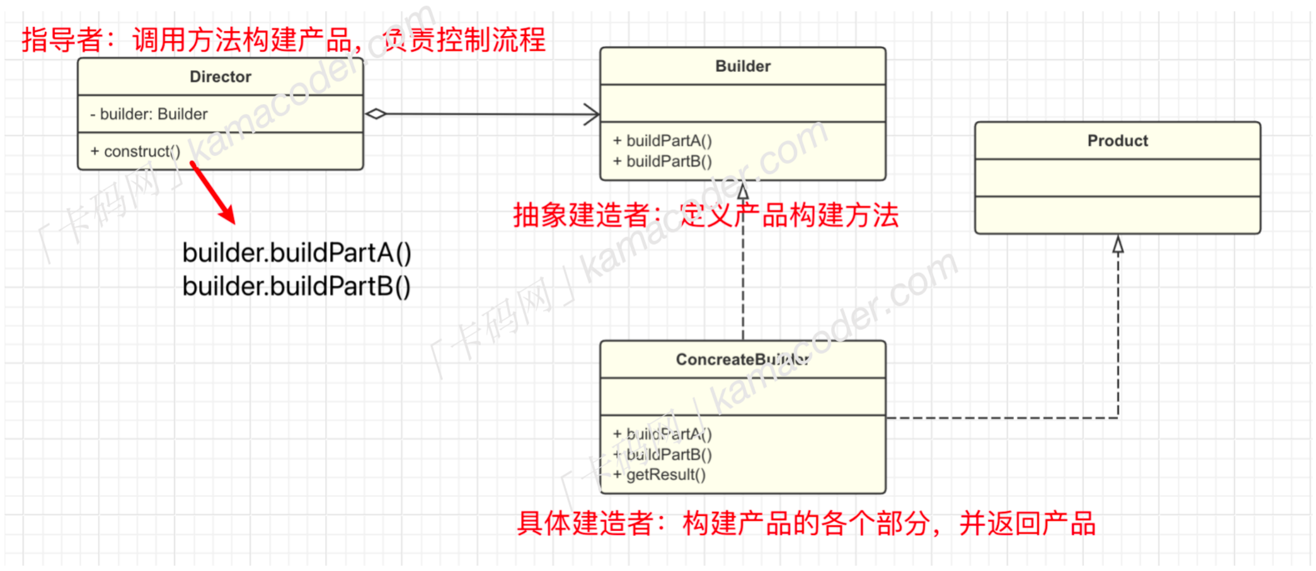
举个例子，假如我们要创建一个计算机对象，计算机由很多组件组成，例如 CPU、内存、硬盘、显卡等。每个组件可能有不同的型号、配置和制造，这个时候计算机就可以被视为一个复杂对象，构建过程相对复杂，而我们使用建造者模式将计算机的构建过程封装在一个具体的建造者类中，而指导者类则负责指导构建的步骤和顺序。每个具体的建造者类可以负责构建不同型号或配置的计算机，客户端代码可以通过选择不同的建造者来创建不同类型的计算机，这样就可以根据需要构建不同表示的复杂对象，更加灵活。

基本结构

建造者模式有下面几个关键角色：

- 产品`Product`：被构建的复杂对象，包含多个组成部分。
- 抽象建造者`Builder`：定义构建产品各个部分的抽象接口和一个返回复杂产品的方法`getResult`
- 具体建造者`Concrete Builder`：实现抽象建造者接口，构建产品的各个组成部分，并提供一个方法返回最终的产品。
- 指导者`Director`：调用具体建造者的方法，按照一定的顺序或逻辑来构建产品。

在客户端中，通过指导者来构建产品，而并不和具体建造者进行直接的交互。



简易实现

建造者模式的实现步骤通常包括以下几个阶段

1. 定义产品类：产品类应该包含多个组成部分，这些部分的属性和方法构成了产品的接口

```
// 产品类
class Product {
    private String part1;
    private String part2;

    public void setPart1(String part1) {
        this.part1 = part1;
    }

    public void setPart2(String part2) {
        this.part2 = part2;
    }

    // 其他属性和方法
}
```

2. 定义抽象建造者接口：创建一个接口，包含构建产品各个部分的抽象方法。这些方法通常用于设置产品的各个属性。

```java
// 抽象建造者接口
interface Builder {
    void buildPart1(String part1);
    void buildPart2(String part2);
    Product getResult();
}
```

3. 创建具体建造者：实现抽象建造者接口，构建具体的产品。

```java
// 具体建造者类
class ConcreteBuilder implements Builder {
    private Product product = new Product();

    @Override
    public void buildPart1(String part1) {
        product.setPart1(part1);
    }

    @Override
    public void buildPart2(String part2) {
        product.setPart2(part2);
    }

    @Override
    public Product getResult() {
        return product;
    }
}
```

4. 定义`Director`类： 指导者类来控制构建产品的顺序和步骤。

```java
// 指导者类
class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }
    // 调用方法构建产品
    public void construct() {
        builder.buildPart1("Part1");
        builder.buildPart2("Part2");
    }
}
```

5. 客户端使用建造者模式：在客户端中创建【具体建造者对象】和【指导者对象】，通过指导者来构建产品。

```
// 客户端代码
public class Main{
    public static void main(String[] args) {
        // 创建具体建造者
        Builder builder = new ConcreteBuilder();

        // 创建指导者
        Director director = new Director(builder);

        // 指导者构建产品
        director.construct();

        // 获取构建好的产品
        Product product = builder.getResult();

        // 输出产品信息
        System.out.println(product);
    }
}
```

使用场景

使用建造者模式有下面几处优点:

- 使用建造者模式可以**将一个复杂对象的构建与其表示分离**，通过将构建复杂对象的过程抽象出来，可以使客户端代码与具体的构建过程解耦
- **同样的构建过程可以创建不同的表示**，可以有多个具体的建造者(相互独立），可以更加灵活地创建不同组合的对象。

对应的，建造者模式适用于复杂对象的创建，当对象构建过程相对复杂时可以考虑使用建造者模式，但是当产品的构建过程发生变化时，可能需要同时修改指导类和建造者类，这就使得重构变得相对困难。

建造者模式在现有的工具和库中也有着广泛的应用，比如JUnit 中的测试构建器`TestBuilder`就采用了建造者模式，用于构建测试对象。

本题代码

```
import java.util.Scanner;

// 自行车产品
class Bike {
```

```java
    private String frame;
    private String tires;

    public void setFrame(String frame) {
        this.frame = frame;
    }

    public void setTires(String tires) {
        this.tires = tires;
    }

    @Override
    public String toString() {
        return frame + " " + tires;
    }
}

// 自行车建造者接口
interface BikeBuilder {
    void buildFrame();
    void buildTires();
    Bike getResult();
}

// 山地自行车建造者
class MountainBikeBuilder implements BikeBuilder {
    private Bike bike;

    public MountainBikeBuilder() {
        this.bike = new Bike();
    }

    @Override
    public void buildFrame() {
        bike.setFrame("Aluminum Frame");
    }

    @Override
    public void buildTires() {
        bike.setTires("Knobby Tires");
    }

    @Override
    public Bike getResult() {
        return bike;
    }
```

```java
}

// 公路自行车建造者
class RoadBikeBuilder implements BikeBuilder {
    private Bike bike;

    public RoadBikeBuilder() {
        this.bike = new Bike();
    }

    @Override
    public void buildFrame() {
        bike.setFrame("Carbon Frame");
    }

    @Override
    public void buildTires() {
        bike.setTires("Slim Tires");
    }

    @Override
    public Bike getResult() {
        return bike;
    }
}

// 自行车Director，负责构建自行车
class BikeDirector {
    public Bike construct(BikeBuilder builder) {
        builder.buildFrame();
        builder.buildTires();
        return builder.getResult();
    }
}

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

        int N = scanner.nextInt();  // 订单数量
        scanner.nextLine();

        BikeDirector director = new BikeDirector();

        for (int i = 0; i < N; i++) {
            String bikeType = scanner.nextLine();
            BikeBuilder builder;
            // 根据输入类别，创建不同类型的具体建造者
            if (bikeType.equals("mountain")) {
                builder = new MountainBikeBuilder();
            } else {
                builder = new RoadBikeBuilder();
            }
            // Director负责指导生产产品
            Bike bike = director.construct(builder);
            System.out.println(bike);
        }
    }
}
```

其他语言代码

Java

使用模版方法模式和建造者模式实现自行车工厂。

```
import java.util.ArrayList;
import java.util.Scanner;

//抽象类，模版方法模式创建
abstract class Bicycle{
    private ArrayList<String> fittingsList = new ArrayList<String>();

    //选用不同配件的方法
    protected void useAluminumFrame(){};
    protected void useKnobbyTires(){};
    protected void useCarbonFrame(){};
    protected void useSlimTires(){};

    //设置配件列表
    public final void setFittingsList(ArrayList<String> list){
        this.fittingsList = list;
    }

    //选用配件，构建自行车
    public final void maker(){
    for (String fitting : fittingsList) {
            switch (fitting) {
                case "useAluminumFrame":
                    useAluminumFrame();
                    break;
                case "useKnobbyTires":
                    useKnobbyTires();
                    break;
                case "useCarbonFrame":
                    useCarbonFrame();
                    break;
                case "useSlimTires":
                    useSlimTires();
                    break;
                default:
                    System.out.println("不清楚的配件：" + fitting + "请重新输
入！");
            }
        }
    }
}

//山地自行车类，并重写父类方法
class MountainBicycle extends Bicycle{
    @Override
    protected void useAluminumFrame(){
        System.out.print("Aluminum Frame ");
    }
    @Override
    protected void useKnobbyTires(){
        System.out.println("Knobby Tires ");
    }
}

//公路自行车类，并重写父类方法
class RoadBicycle extends Bicycle{
    @Override
    protected void useCarbonFrame(){
        System.out.print("Carbon Frame ");
    }
    @Override
    protected void useSlimTires(){
        System.out.println("Slim Tires ");
    }
}

//定义建造者接口
interface Builder{
    void setFittingList(ArrayList<String> list);
    Bicycle getBicycle();
```

```java
}

//山地自行车建造类
class MountainBicycleBuilder implements Builder{
    private MountainBicycle mountainBicycle = new MountainBicycle();

    public void setFittingList(ArrayList<String> list){
        this.mountainBicycle.setFittingsList(list);
    }

    public Bicycle getBicycle(){
        return this.mountainBicycle;
    }
}

//公路自行车建造者类
class RoadBicycleBuilder implements Builder{
    private RoadBicycle roadBicycle = new RoadBicycle();

    public void setFittingList(ArrayList<String> list){
        this.roadBicycle.setFittingsList(list);
    }

    public Bicycle getBicycle(){
        return this.roadBicycle;
    }
}

//导演类
class Director{
    private ArrayList<String> arrayList = new ArrayList<String>();
    private MountainBicycleBuilder mountainBicycleBuilder = new
MountainBicycleBuilder();
    private RoadBicycleBuilder roadBicycleBuilder = new
RoadBicycleBuilder();

    public MountainBicycle getMountainBicycle(){
    //清理场景
        this.arrayList.clear();
        //选用配件
        this.arrayList.add("useAluminumFrame");
        this.arrayList.add("useKnobbyTires");
        this.mountainBicycleBuilder.setFittingList(this.arrayList);
        return (MountainBicycle)this.mountainBicycleBuilder.getBicycle();
    }
    public RoadBicycle getRoadBicycle(){
            //清理场景
        this.arrayList.clear();
        //选用配件
        this.arrayList.add("useCarbonFrame");
        this.arrayList.add("useSlimTires");
        this.roadBicycleBuilder.setFittingList(this.arrayList);
        return (RoadBicycle)this.roadBicycleBuilder.getBicycle();
    }

}

//主程序类
public class Main{
    public static void main (String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            int num = scanner.nextInt();
            scanner.nextLine();
            Director director = new Director();

            for (int i = 0; i < num; i++) {
                String type = scanner.nextLine().toLowerCase().trim();
                Bicycle bicycle;
                if ("mountain".equals(type)) {
                    bicycle = director.getMountainBicycle();
                } else if ("road".equals(type)) {
                    bicycle = director.getRoadBicycle();
                } else {
                    System.out.println("无效输入. 请输入 'mountain' 或
'road'.");
                    continue;
                }
                bicycle.maker();
            }
        } catch (Exception e) {
            System.out.println("An error occurred: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }
}
```

Cpp

```cpp
#include <iostream>
#include <string>

// 自行车产品
class Bike {
public:
    std::string frame;
    std::string tires;

    void setFrame(const std::string& frame) {
        this->frame = frame;
    }

    void setTires(const std::string& tires) {
        this->tires = tires;
    }

    friend std::ostream& operator<<(std::ostream& os, const Bike& bike) {
        os << bike.frame << " " << bike.tires;
        return os;
    }
};

// 自行车建造者接口
class BikeBuilder {
public:
    virtual void buildFrame() = 0;
    virtual void buildTires() = 0;
    virtual Bike getResult() = 0;
};

// 山地自行车建造者
class MountainBikeBuilder : public BikeBuilder {
private:
    Bike bike;

public:
    void buildFrame() override {
        bike.setFrame("Aluminum Frame");
    }

    void buildTires() override {
        bike.setTires("Knobby Tires");
    }
    Bike getResult() override {
        return bike;
    }
};

// 公路自行车建造者
class RoadBikeBuilder : public BikeBuilder {
private:
    Bike bike;

public:
    void buildFrame() override {
        bike.setFrame("Carbon Frame");
    }

    void buildTires() override {
        bike.setTires("Slim Tires");
    }

    Bike getResult() override {
        return bike;
    }
};

// 自行车Director，负责构建自行车
class BikeDirector {
public:
    Bike construct(BikeBuilder& builder) {
        builder.buildFrame();
        builder.buildTires();
        return builder.getResult();
    }
};

int main() {
    int N;
    std::cin >> N;  // 订单数量

    BikeDirector director;

    for (int i = 0; i < N; i++) {
        std::string bikeType;
        std::cin >> bikeType;

        BikeBuilder* builder;
        // 根据输入类别，创建不同类型的具体建造者
        if (bikeType == "mountain") {
            builder = new MountainBikeBuilder();
        } else {
            builder = new RoadBikeBuilder();
        }

        // Director负责指导生产产品
        Bike bike = director.construct(*builder);
        std::cout << bike << std::endl;

        // 释放动态分配的对象
        delete builder;
    }

    return 0;
}
```

Python

```python
# 自行车产品
class Bike:
    def __init__(self):
        self.frame = None
        self.tires = None

    def set_frame(self, frame):
        self.frame = frame

    def set_tires(self, tires):
        self.tires = tires

    def __str__(self):
        return f"{self.frame} {self.tires}"

# 自行车建造者接口
class BikeBuilder:
    def build_frame(self):
        pass

    def build_tires(self):
        pass

    def get_result(self):
        pass
```

```
# 山地自行车建造者
class MountainBikeBuilder(BikeBuilder):
    def __init__(self):
        self.bike = Bike()

    def build_frame(self):
        self.bike.set_frame("Aluminum Frame")

    def build_tires(self):
        self.bike.set_tires("Knobby Tires")

    def get_result(self):
        return self.bike

# 公路自行车建造者
class RoadBikeBuilder(BikeBuilder):
    def __init__(self):
        self.bike = Bike()

    def build_frame(self):
        self.bike.set_frame("Carbon Frame")

    def build_tires(self):
        self.bike.set_tires("Slim Tires")

    def get_result(self):
        return self.bike

# 自行车Director，负责构建自行车
class BikeDirector:
    def construct(self, builder):
        builder.build_frame()
        builder.build_tires()
        return builder.get_result()

def main():
    N = int(input())  # 订单数量

    director = BikeDirector()

    for _ in range(N):
        bike_type = input()

        # 根据输入类别，创建不同类型的具体建造者
        if bike_type == "mountain":
```

```python
            builder = MountainBikeBuilder()
        else:
            builder = RoadBikeBuilder()

        # Director负责指导生产产品
        bike = director.construct(builder)
        print(bike)

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

Go

```go
package main

import "fmt"

// 自行车产品
type Bike struct {
    frame string
    tires string
}

func (b *Bike) setFrame(frame string) {
    b.frame = frame
}

func (b *Bike) setTires(tires string) {
    b.tires = tires
}

func (b *Bike) String() string {
    return b.frame + " " + b.tires
}

// 自行车建造者接口
type BikeBuilder interface {
    buildFrame()
    buildTires()
    getResult() *Bike
}

// 山地自行车建造者
type MountainBikeBuilder struct {
    bike *Bike
}
```

```go
func NewMountainBikeBuilder() *MountainBikeBuilder {
    return &MountainBikeBuilder{
        bike: &Bike{},
    }
}

func (mbb *MountainBikeBuilder) buildFrame() {
    mbb.bike.setFrame("Aluminum Frame")
}

func (mbb *MountainBikeBuilder) buildTires() {
    mbb.bike.setTires("Knobby Tires")
}

func (mbb *MountainBikeBuilder) getResult() *Bike {
    return mbb.bike
}

// 公路自行车建造者
type RoadBikeBuilder struct {
    bike *Bike
}

func NewRoadBikeBuilder() *RoadBikeBuilder {
    return &RoadBikeBuilder{
        bike: &Bike{},
    }
}

func (rbb *RoadBikeBuilder) buildFrame() {
    rbb.bike.setFrame("Carbon Frame")
}

func (rbb *RoadBikeBuilder) buildTires() {
    rbb.bike.setTires("Slim Tires")
}

func (rbb *RoadBikeBuilder) getResult() *Bike {
    return rbb.bike
}

// 自行车Director，负责构建自行车
type BikeDirector struct{}

func (bd *BikeDirector) construct(builder BikeBuilder) *Bike {
    builder.buildFrame()
    builder.buildTires()
    return builder.getResult()
}

func main() {
    var N int
    fmt.Scan(&N) // 订单数量

    director := &BikeDirector{}

    for i := 0; i < N; i++ {
        var bikeType string
        fmt.Scan(&bikeType)

        var builder BikeBuilder
        // 根据输入类别，创建不同类型的具体建造者
        if bikeType == "mountain" {
            builder = NewMountainBikeBuilder()
        } else {
            builder = NewRoadBikeBuilder()
        }
        // Director负责指导生产产品
        bike := director.construct(builder)
        fmt.Println(bike)
    }
}
```