1、简介

建造者模式(Builder Pattern)旨在将一个复杂对象的**构建**与**表示**分离，使得同样的构建过程可以创建不同的表示。简单来说就是使用**多个简单的对象一步一步构建成一个复杂的对象。**这种类型的设计模式属于**创建型模式**，是创建对象的最佳方式之一，其主要特点如下：

- **分离**了部件的构造(由Builder来负责)和装配(由Director负责)。 从而可以构造出复杂的对象。这个模式适用于：某个对象的构建过程复杂的情况。
- 由于实现了**构建和装配的解耦。**不同的构建器，相同的装配，也可以做出不同的对象；相同的构建器，不同的装配顺序也可以做出不同的对象。也就是实现了构建算法、装配算法的解耦，实现了更好的复用。
- 建造者模式可以**将部件和其组装过程分开**，一步一步创建一个复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，而无须知道其内部的具体构造细节。

通过建造者模式，在用户不知道对象的建造过程和细节的情况下就可以直接创建复杂的对象。

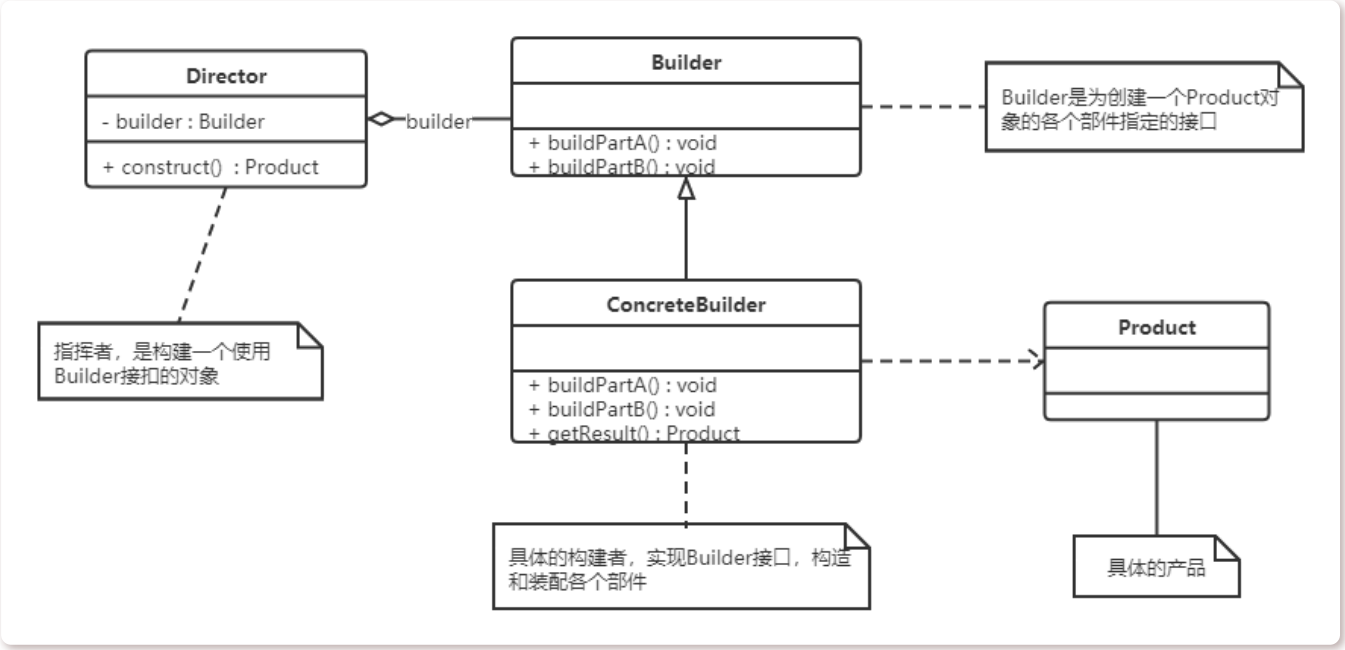
- 用户只需要给出指定复杂对象的类型和内容；
- 建造者模式负责按顺序创建复杂对象（把内部的建造过程和细节隐藏起来）

2、结构

建造者模式包含如下角色：

- **抽象建造者类**(Builder)：这个接口规定要实现复杂对象的那些部分的创建，并不涉及具体的部件对象的创建；
- **具体建造者类**(ConcreteBuilder)：继承实现 Builder 抽象类，完成复杂产品的各个部件的具体创建方法。在构造过程完成后，提供产品的实例；
- **产品类**(Product)：要创建的复杂对象；

- **指挥者类(Director)**：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。



3、实现

3.1、需求场景

可能大家都拥有着自己的一把或多把键盘，而生产键盘是一个复杂的过程，它由键帽、轴体、驱动以及轴下垫等一系列配件组成。熟悉的老手可能了解键帽、轴体等这些都是有不同的，轴体有金粉、风信子、冰淇淋等，键帽有xda、dsa、原厂等。因此针对于复杂的键盘组装过程就可以使用建造者模式；

> 这里为了方便演示，键盘类只设置了键帽和轴体两个成员属性，并只是用字符串类型代表，旨在演示建造者模式。

3.2、产品类

```java
public class Keyboard {
    /** 键帽 */
    private String keycap;

    /** 轴体 */
    private String keyswitch;

    // 省略get、set方法
}
```

3.3、抽象建造者类

```
public abstract class KeyboardBuilder {
    protected Keyboard keyboard;

    public KeyboardBuilder () {
        this.keyboard = new Keyboard();
    }

    public abstract void buildKeycap();
    public abstract void buildKeyswitch();
    public abstract Keyboard createKeyboard();
}
```

3.4、具体建造者类

```
/**
 * @author xbaozi
 * @version 1.0
 * @classname CherryKeyboardBuilder
 * @date 2023-02-14  22:22
 * @description 用于演示建造者模式-樱桃具体建造者类
 */
public class CherryKeyboardBuilder extends KeyboardBuilder {
    @Override
    public void buildKeycap() {
        keyboard.setKeycap("xda高度键帽");
    }

    @Override
    public void buildKeyswitch() {
        keyboard.setKeyswitch("金粉轴");
    }

    @Override
    public Keyboard createKeyboard() {
        return keyboard;
    }
}

/**
 * @author xbaozi
 * @version 1.0
 * @classname DellyouKeyboardBuilder
 * @date 2023-02-14  22:24
 * @description 用于演示建造者模式-达尔优具体建造者类
 */
public class DellyouKeyboardBuilder extends KeyboardBuilder {
```

```
    @Override
    public void buildKeycap() {
        keyboard.setKeycap("dsa高度键帽");
    }

    @Override
    public void buildKeyswitch() {
        keyboard.setKeyswitch("酒红轴");
    }

    @Override
    public Keyboard createKeyboard() {
        return keyboard;
    }
}
```

3.5、指挥者类

```
/**
 * @author xbaozi
 * @version 1.0
 * @classname Director
 * @date 2023-02-14  22:31
 * @description 用于演示建造者模式-指挥者类
 */
public class Director {
    private KeyboardBuilder keyboardBuilder;

    public Director(KeyboardBuilder keyboardBuilder) {
        this.keyboardBuilder = keyboardBuilder;
    }

    public Keyboard construct() {
        // 先组装轴体
        keyboardBuilder.buildKeyswitch();
        // 在组装键帽
        keyboardBuilder.buildKeycap();
        // 完成键盘组装
        return keyboardBuilder.createKeyboard();
    }
}
```

3.6、测试类

```
/**
 * @author xbaozi
 * @version 1.0
 * @classname BuilderTest
```

```
 * @date 2023-02-14  22:36
 * @description 建造者魔术测试类
 */
public class BuilderTest {
    @Test
    public void buiderTest() {
        Keyboard cherryKeyboard = new Director(new CherryKeyboardBuilder()).construct();
        Keyboard dellyouKeyboard = new Director(new DellyouKeyboardBuilder()).construct();
        System.out.println("樱桃键盘：\t" + cherryKeyboard);
        System.out.println("达尔优键盘：\t" + dellyouKeyboard);
    }
}
```

3.7、演示结果

上面示例是 Builder模式的常规用法，导演类 Director 在 Builder模式中具有很重要的作用，它用于指导具体构建者**如何构建产品，控制调用先后次序**，并向调用者返回完整的产品类。

```
"D:\Program Files\Java\jdk-11.0.16.1\bin\java.exe" "-javaagent
樱桃键盘：    Keyboard{keycap='xda高度键帽', keyswitch='金粉轴'}
达尔优键盘：   Keyboard{keycap='dsa高度键帽', keyswitch='酒红轴'}

Process finished with exit code 0
```

4、应用场景

1. 隔离复杂对象的创建和使用，相同的方法，不同执行顺序，产生不同事件结果
2. 多个部件都可以装配到一个对象中，但产生的运行结果不相同
3. 产品类非常复杂或者产品类因为调用顺序不同而产生不同作用
4. 初始化一个对象时，参数过多，或者很多参数具有默认值
5. Builder模式不适合创建差异性很大的产品类
6. 产品内部变化复杂，会导致需要定义很多具体建造者类实现变化，增加项目中类的数量，增加系统的理解难度和运行成本
7. 需要生成的产品对象有复杂的内部结构，这些产品对象具备共性。

5、实操举例

除了上述的用途外，还有另外一个常用的使用方式，就是当一个类构造器需要传入很多参数时，如果创建这个类的实例，代码可读性会非常差，而且很容易引入错误，此时就可以利用建造者模式进行重构。

> 🌈

> ps：这个例子在开发过程中一般不需要我们自己编写，若有导入Lombok包的话，直接在实体类上方加入 `@Builder` 注解即可实现。

这里用项目中常用到的用户类进行举例，用户类一般拥有众多属性，如果直接通过全参构造器进行赋值可读性将会极差，用建造者模式重构如下：

```java
/**
 * @author xbaozi
 * @version 1.0
 * @classname User
 * @date 2023-02-14  22:47
 * @description 建造者模式另一种用途
 */
public class User {
    private String name;
    private String idCard;
    private int age;
    private String gender;
    private String city;

    /**
     * 私有化构造函数，通过建造者进行创建
     * @param builder 建造者
     */
    private User(Builder builder) {
        name = builder.name;
        idCard = builder.idCard;
        age = builder.age;
        gender = builder.gender;
        city = builder.city;
    }

    /**
     * 建造者
     */
    public static final class Builder {
        private String name;
        private String idCard;
        private int age;
        private String gender;
        private String city;

        public Builder name(String name) {
            this.name = name;
            return this;
        }

        public Builder idCard(String idCard) {
```

```
            this.idCard = idCard;
            return this;
        }

        public Builder age(int age) {
            this.age = age;
            return this;
        }

        public Builder gender(String gender) {
            this.gender = gender;
            return this;
        }

        public Builder city(String city) {
            this.city = city;
            return this;
        }

        public User build() {
            return new User(this);
        }
    }
}

/**
 * @author xbaozi
 * @version 1.0
 * @classname BuilderTest
 * @date 2023-02-14  22:36
 * @description 建造者模式测试类
 */
public class BuilderTest {

    @Test
    public void builderTest1 () {
        User user1 = new User.Builder()
                .name("陈宝子")
                .age(18)
                .idCard("100861100")
                .city("广东")
                .gender("男")
                .build();

        User user2 = new User.Builder()
                .name("爱吃鱼蛋")
                .age(81)
                .idCard("10010")
                .city("广东")
```

```
                .gender("男")
                .build();

        System.out.println(user1);
        System.out.println(user2);
    }
}
```

运行结果如下，可以看到通过建造者构建得到的用户类是正常的，重构后的代码在使用起来更方便，某种程度上也可以提高开发效率。这里演示的入参只有五个，如果有更多，那么建造者模式的优势则是更为明显：

```
"D:\Program Files\Java\jdk-11.0.16.1\bin\java.exe" "-javaagent:D:\Progra
User{name='陈宝子', idCard='100861100', age=18, gender='男', city='广东'}
User{name='爱吃鱼蛋', idCard='10010', age=81, gender='男', city='广东'}

Process finished with exit code 0
```

6、优缺点分析

优点：

- 使用建造者模式可以使客户端不必知道产品内部组成的细节。
- 具体的建造者类之间是相互独立的，这有利于系统的扩展。
- 具体的建造者相互独立，因此可以对建造的过程逐步细化，而不会对其他模块产生任何影响。

缺点：

- 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似；如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。
- 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。

> 一般的套路：优点是比较简单，开发效率高，缺点是如果参数真的很多的话鬼知道每个对应的是什么意思啊。

> Builder模式：优点是可以将构造器的setter方法名取成类似注释的方式，这样我们可以很清晰的知道刚才究竟设置的什么值，可读性较高，缺点是比较冗长。

7、抽象工厂模式区别

- 与抽象工厂模式相比，**建造者模式**返回一个组装好的完整产品，而**抽象工厂模式**返回一系列相关的产品，这些产品位于不同的产品等级结构，构成了一个产品簇。
- 在**抽象工厂模式**中，客户端实例化工厂类，然后调用工厂方法获取所需产品对象，而在**建造者模式**中，客户端可以不直接调用建造者的相关方法，而是通过指挥者类来指导如何生成对象，包括对象的组装过程和建造步骤，它**侧重于一步步构造一个复杂对象，返回一个完整的对象。**
- 如果将**抽象工厂模式**看成汽车配件生产工厂，生产一个产品族的产品，那么建造者模式就是一个汽车组装工厂，通过对部件的组装可以返回一辆完整的汽车。

> 
>
> 简单来说，抽象工厂更偏向于生产那些固定量产的产品，而建造者模式更偏向于客制化，自由度更高，专注于产品的构建的细节。