第2章 建造者模式

提出问题

> 问题描述：要求根据原始数据创建文本和HTML两种风格的列表。假设列表只到二级，原始数据格式为 `标题1:子标题1,子标题2;标题2,子标题1,子标题2` ，一级列表之间采用分号分隔，子列表之间采用逗号分隔。

文本列表格式如下：

```
标题1
-子标题1
-子标题2
标题2
-子标题1
-子标题2
```

HTML列表格式如下：

```
<li>标题1
<ul><li>子标题1</li>
<li>子标题2</li>
</ul></li>
<li>标题2
<ul><li>子标题1</li>
<li>子标题2</li>
</ul></li>
```

根据问题描述，对象的构建过程比较复杂，需要先对源数据进行解析，再逐个创建列表的各个部分，最终组成完整的列表对象。这里设计`TextListBuilder`和`HTMLListBuilder`两个类用于创建两种格式的列表，主要创建过程封装在`build()`方法中。具体代码如下所示：

```
public class Test {
    public static void main(String[] args) {
        String data = "标题1:子标题1,子标题2;标题2:子标题1,子标题2";
        HTMLListBuilder html = new HTMLListBuilder();
        html.build(data);
        TextListBuilder text = new TextListBuilder();
        text.build(data);
        System.out.println(text.toString());
        System.out.println(html.toString());
    }
}
class TextListBuilder {
    private StringBuilder value;
    public TextListBuilder() {
        value = new StringBuilder();
    }
    public void build(String data){
        String[] ulList = data.split(";");
        for(String ul : ulList) {
            String[] items = ul.split(":");
            String title = items[0];
            value.append(title + "\n");
            String[] liList = items[1].split(",");
            for(String liTitle : liList) {
                value.append("-" + liTitle + "\n");
            }
            //value.append("");
        }
    }
    public String toString() {
        return value.toString();
    }
}
class HTMLListBuilder {
    private StringBuilder value;
    public HTMLListBuilder() {
        value = new StringBuilder();
    }
    public void build(String data){
        String[] ulList = data.split(";");
        for(String ul : ulList) {
            String[] items = ul.split(":");
            String title = items[0];
            value.append("<li>" + title + "\n<ul>");
            String[] liList = items[1].split(",");
            for(String liTitle : liList) {
```

```
                value.append("<li>" + liTitle + "</li>\n");
            }
            value.append("</ul></li>\n");
        }
    }
    public String toString() {
        return value.toString();
    }
}
```

这里对象的创建流程和对象表示混合在一起，这可能会导致几个问题：

- 代码复用性较低。仔细观察如上代码，对象创建流程较复杂，且各类对象创建流程相似，有区别的是每个步骤对象表示形式，例如HTML格式中列表项是`<li>标题<li>`，而文本格式中列表项是`-标题`。创建流程代码没有得到有效复用，如果扩展一个新的格式，需要重复将相似的流程代码再写一遍。因此，复用性低导致功能扩展成本较大。
- 代码可维护性较低。如果对象创建流程发生变化，则可能需要修改所有`build()`方法中的代码。

模式名称

建造者模式：Builder

设计意图

建造者模式将复杂对象的构造过程与其表示分离，使得相同的构造过程可以创建不同表示的对象。

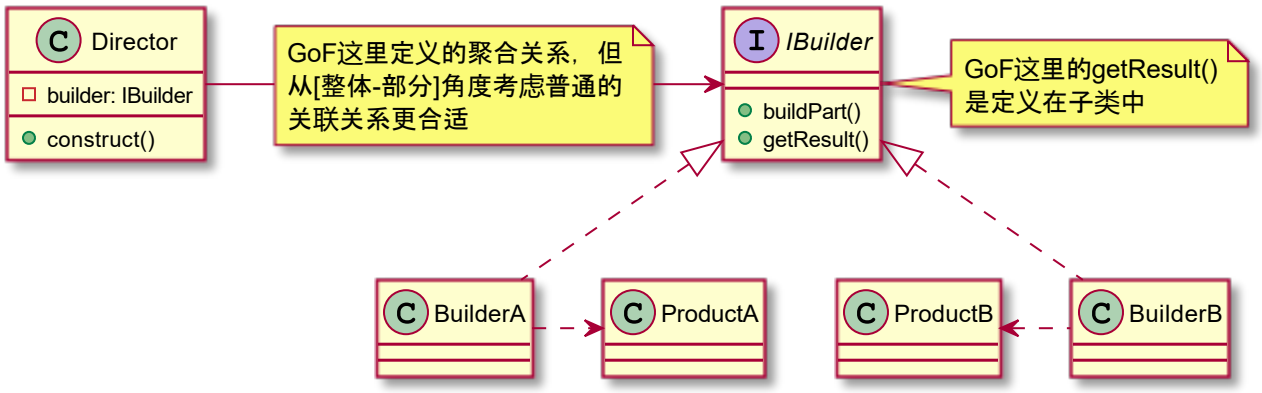
> Separate the construction of a complex object from its representation so that the same construction process can create different representations.

设计结构

建造者模式结构如下图所示，参与的角色包括四种：

- 指挥者（Director）：负责定义复杂对象创建流程，主要实现`construct()`方法；
- 抽象建造者（IBuilder）：负责定义复杂对象各个部分创建的接口；

- 具体建造者（ConcreteBuilder）：实现`IBuilder`接口，构造和装配各个配件；提供获取创建对象实例的方法（例如`getResult()`）。
- 具体产品（Product）：需要创建的复杂对象。



根据不同理解可能导致的设计结构差异，给出几点思考：

- GoF将`Director`和`IBuilder`之间的关系设计为聚合。思考两者是不是"整体-部分"关系？用普通的关联关系是否更恰当？
- GoF在`IBuilder`接口中没有给出`getResult()`方法。如果在接口中给出`getResult()`方法，那么返回值适合设计为什么类型？Object类或一个抽象的产品类？
- 建造者模式中没有给出抽象产品类。GoF的解释是，一般情况下具体产品类之间的表现形式差别较大，通过公共的抽象类可能无法有效获取具体产品类的信息。
- 指挥者主要定义对象创建的流程，那么流程也可能多元化，这里的`Director`是否能设计多个？GoF在讨论建造者模式效果时，提到过一个具体的建造者类可以被多种指挥者利用
  。`[The code is written once; then different Directors can reuse it to build Product variants`

除了以上几点，可能还存在一些细节上的推敲，例如`Director`的`construct()`函数能否直接返回被创建对象，还是只执行创建命令。

解决问题

基于建造者模式对前面的设计进行改进，将对象的创建过程与表示进行分离，创建过程封装在一个指挥者类中，对象表示封装在建造者类中。那么现在分析一个列表有哪些具体操作会影响其表示。问题描述中的列表包括一个一级标题和多个子标题，将其构建过程分解为添加一级标题、添加子标题和添加结束标签三个基本步骤。因此，接口`IBuilder`定义如下：

```
public interface IBuilder {
    public void appendListTitle(String title); // 添加一级标题
    public void appendListItem(String item); // 添加一项子标题
    public void appendListTail(); // 添加结束标签
    public StringBuilder getResult();
}
```

遵循依赖倒置原则，指挥者的`construct()`方法实现依赖于接口`IBuilder`，其具体实现如下：

```
public class Director {
    private IBuilder builder;
    public void construct(String data) {
        String[] ulList = data.split(";");
        for(String ul : ulList) {
            String[] items = ul.split(":");
            String title = items[0];
            builder.appendListTitle(title);
            String[] liList = items[1].split(",");
            for(String liTitle : liList) {
                builder.appendListItem(liTitle);
            }
            builder.appendListTail();
        }
    }
}
```

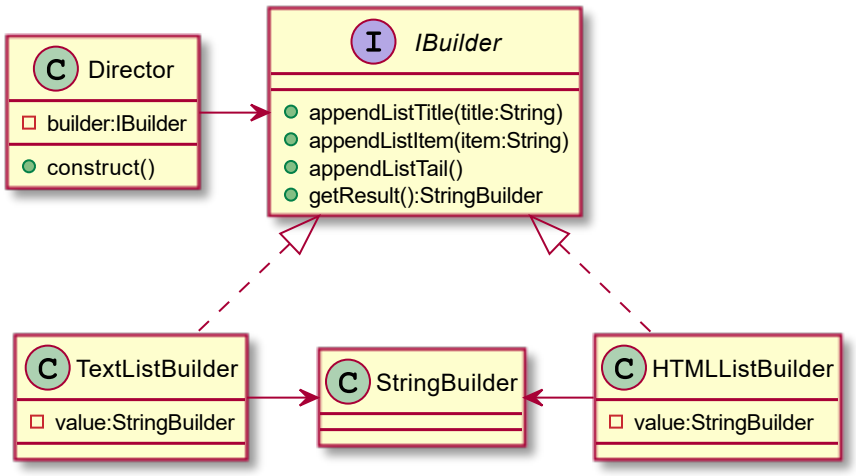
接下来是具体建造类的实现：

```
public class HTMLListBuilder implements IBuilder{
    private StringBuilder value;

    @Override
    public void appendListTitle(String title) {
        value.append("<li>" + title + "\n<ul>");
    }

    @Override
    public void appendListItem(String item) {
        value.append("<li>" + item +"</li>\n");
    }

    @Override
    public void appendListTail() {
        value.append("</ul></li>\n");
    }

    @Override
    public StringBuilder getResult() {
        return value;
    }
}

public class TextListBuilder implements IBuilder{
    private StringBuilder value;

    @Override
    public void appendListTitle(String title) {
        value.append(title + "\n");
    }

    @Override
    public void appendListItem(String item) {
        value.append("-" + item + "\n");
    }

    @Override
    public void appendListTail() {    }

    @Override
    public StringBuilder getResult() {
        return value;
    }
}
```

由于这里最终的产品是一个文本，产品类直接采用的系统类`StringBuilder`。实现所有角色后，进行测试：

```
public class Test {
    public static void main(String[] args) {
        String data = "标题1:子标题1,子标题2;标题2:子标题1,子标题2";
        Director director = new Director();
        IBuilder builder = new HTMLListBuilder();
        director.setBuilder(builder);
        director.construct(data);
        System.out.println(builder.getResult());

        builder = new TextListBuilder();
        director.setBuilder(builder);
        director.construct(data);
        System.out.println(builder.getResult());
    }
}
```

根据以上设计，类图结构如下：



效果与适用性

建造者模式将对象的创建流程与表示分离，能达到以下效果：

- 产品对象的内部表示易于扩展。只对指挥者提供统一操作接口，对其隐藏产品对象具体的内部表示、结构和组装方式。当需要新增其他表示的产品对象时，只需要定义一个新的建造者类。

- 促进代码模块化并提高复用率。将对象的创建流程和表示分别封装，促进了模块化。建造者类和指挥者类之间两两配对以实现各种场景应用，这比为每个场景开发独立代码的复用率要高。
- 能够更精细化地控制对象创建过程。建造者模式中地对象创建流程不是一步到位的，而是分步骤进行的，指挥者能针对每个步骤进行更精细化地控制。

适用性：

- 创建复杂对象的流程需要独立于对象内部结构、表示和组成方式的时候；
- 需要创建流程生成不同表示的对象的时候。