建造模式

> *整理自：《java与模式》之建造模式*

在阎宏博士的《JAVA与模式》一书中开头是这样描述建造（Builder）模式的：

建造模式是对象的创建模式。建造模式可以将一个产品的内部表象（internal representation）与产品的生产过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。

产品的内部表象

一个产品常有不同的组成成分作为产品的零件，这些零件有可能是对象，也有可能不是对象，它们通常又叫做产品的内部表象（internal representation）。不同的产品可以有不同的内部表象，也就是不同的零件。使用建造模式可以使客户端不需要知道所生成的产品有哪些零件，每个产品的对应零件彼此有何不同，是怎么建造出来的，以及怎么组成产品。

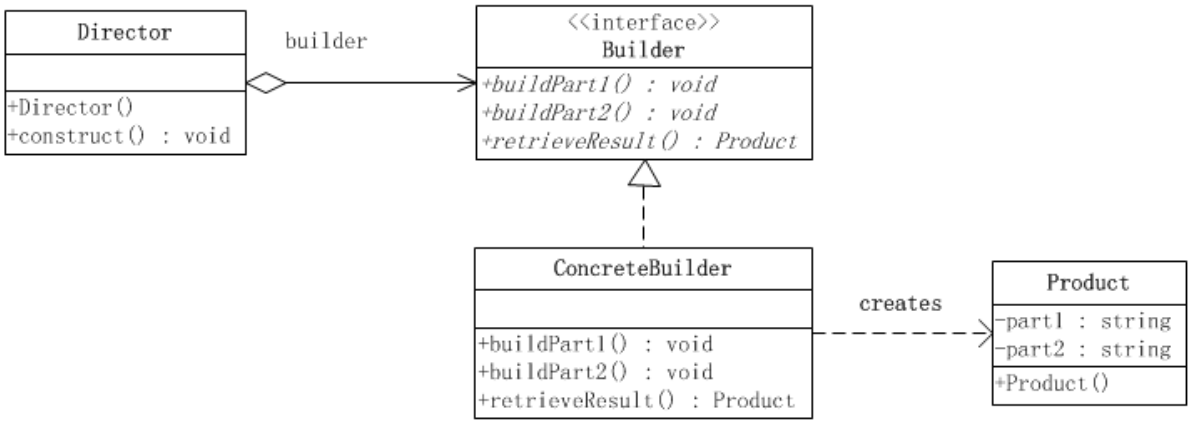
对象性质的建造

有些情况下，一个对象会有一些重要的性质，在它们没有恰当的值之前，对象不能作为一个完整的产品使用。比如，一个电子邮件有发件人地址、收件人地址、主题、内容、附录等部分，而在最起码的收件人地址得到赋值之前，这个电子邮件不能发送。

有些情况下，一个对象的一些性质必须按照某个顺序赋值才有意义。在某个性质没有赋值之前，另一个性质则无法赋值。这些情况使得性质本身的建造涉及到复杂的商业逻辑。这时候，此对象相当于一个有待建造的产品，而对象的这些性质相当于产品的零件，建造产品的过程是建造零件的过程。由于建造零件的过程很复杂，因此，这些零件的建造过程往往被“外部化”到另一个称做建造者的对象里，建造者对象返还给客户端的是一个全部零件都建造完毕的产品对象。

建造模式利用一个导演者对象和具体建造者对象一个个地建造出所有的零件，从而建造出完整的产品对象。建造者模式将产品的结构和产品的零件的建造过程对客户端隐藏起来，把对建造过程进行指挥的责任和具体建造者零件的责任分割开来，达到责任划分和封装的目的。

建造模式的结构



在这个示意性的系统里，最终产品Product只有两个零件，即part1和part2。相应的建造方法也有两个：buildPart1()和buildPart2()、同时可以看出本模式涉及到四个角色，它们分别是：

抽象建造者（Builder）角色：给 出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此接口独立于应用程序的商业逻辑。模式中直接创建产品对象的是具体建造者 (ConcreteBuilder)角色。具体建造者类必须实现这个接口所要求的两种方法：一种是建造方法(buildPart1和 buildPart2)，另一种是返还结构方法(retrieveResult)。一般来说，产品所包含的零件数目与建造方法的数目相符。换言之，有多少 零件，就有多少相应的建造方法。

具体建造者（ConcreteBuilder）角色：担任这个角色的是与应用程序紧密相关的一些类，它们在应用程序调用下创建产品的实例。这个角色要完成的任务包括：1.实现抽象建造者Builder所声明的接口，给出一步一步地完成创建产品实例的操作。2.在建造过程完成后，提供产品的实例。

导演者（Director）角色：担任这个角色的类调用具体建造者角色以创建产品对象。应当指出的是，导演者角色并没有产品类的具体知识，真正拥有产品类的具体知识的是具体建造者角色。

产品（Product）角色：产品便是建造中的复杂对象。一般来说，一个系统中会有多于一个的产品类，而且这些产品类并不一定有共同的接口，而完全可以是不相关联的。

导演者角色是与客户端打交道的角色。导演者将客户端创建产品的请求划分为对各个零件的建造请求，再将这些请求委派给具体建造者角色。具体建造者角色是做具体建造工作的，但是却不为客户端所知。

一般来说，每有一个产品类，就有一个相应的具体建造者类。这些产品应当有一样数目

的零件，而每有一个零件就相应地在所有的建造者角色里有一个建造方法。

源代码

产品类Product

```
public class Product {
    /**
     * 定义一些关于产品的操作
     */
    private String part1;
    private String part2;
    public String getPart1() {
        return part1;
    }
    public void setPart1(String part1) {
        this.part1 = part1;
    }
    public String getPart2() {
        return part2;
    }
    public void setPart2(String part2) {
        this.part2 = part2;
    }
}
```

抽象建造者类Builder

```
public interface Builder {
    public void buildPart1();
    public void buildPart2();
    public Product retrieveResult();
}
```

具体建造者类ConcreteBuilder

```
public class ConcreteBuilder implements Builder {

    private Product product = new Product();
```

```
    /**
     * 产品零件建造方法1
     */
    @Override
    public void buildPart1() {
        //构建产品的第一个零件
        product.setPart1("编号: 9527");
    }
    /**
     * 产品零件建造方法2
     */
    @Override
    public void buildPart2() {
        //构建产品的第二个零件
        product.setPart2("名称: XXX");
    }
    /**
     * 产品返还方法
     */
    @Override
    public Product retrieveResult() {
        return product;
    }

}
```

导演者类Director

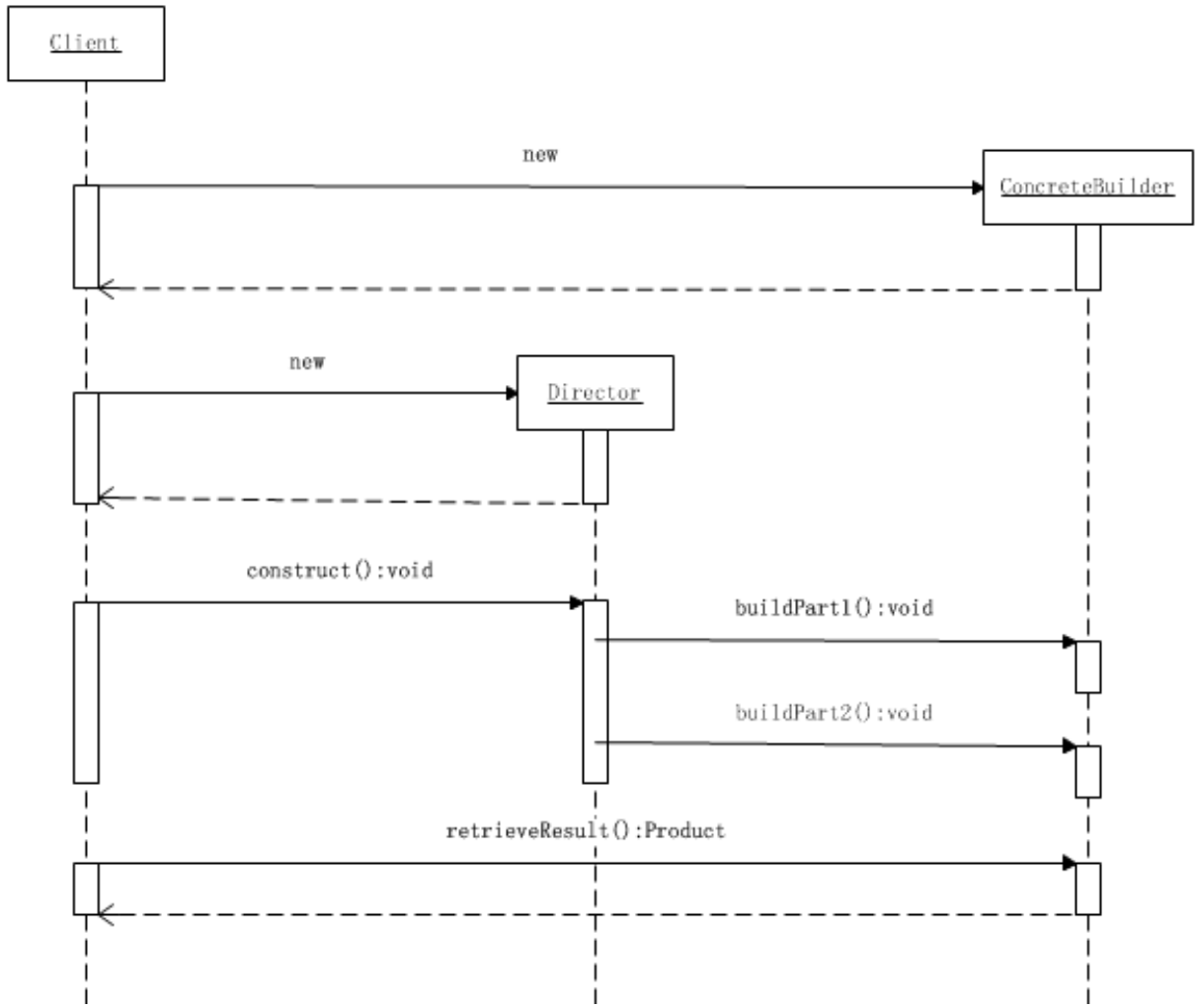
```
public class Director {
    /**
     * 持有当前需要使用的建造器对象
     */
    private Builder builder;
    /**
     * 构造方法，传入建造器对象
     * @param builder 建造器对象
     */
    public Director(Builder builder){
        this.builder = builder;
    }
```

```
    /**
     * 产品构造方法，负责调用各个零件建造方法
     */
    public void construct(){
        builder.buildPart1();
        builder.buildPart2();
    }
}
```

客户端类Client

```
public class Client {
    public static void main(String[]args){
        Builder builder = new ConcreteBuilder();
        Director director = new Director(builder);
        director.construct();
        Product product = builder.retrieveResult();
        System.out.println(product.getPart1());
        System.out.println(product.getPart2());
    }
}
```

时序图



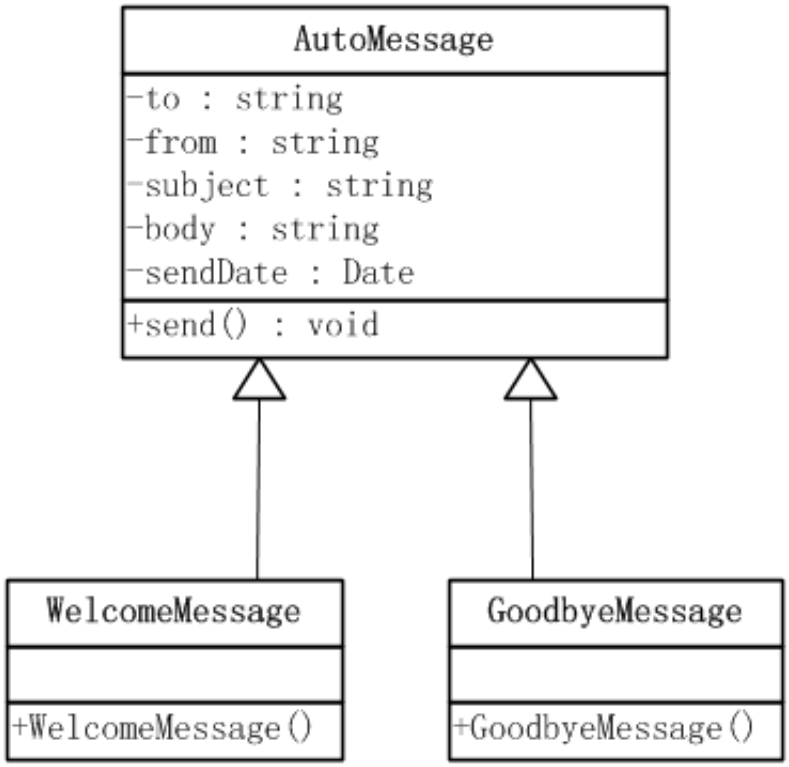
客户端负责创建导演者和具体建造者对象。然后，客户端把具体建造者对象交给导演者，导演者操作具体建造者，开始创建产品。当产品完成后，建造者把产品返还给客户端。

把创建具体建造者对象的任务交给客户端而不是导演者对象，是为了将导演者对象与具体建造者对象的耦合变成动态的，从而使导演者对象可以操纵数个具体建造者对象中的任何一个。

使用场景

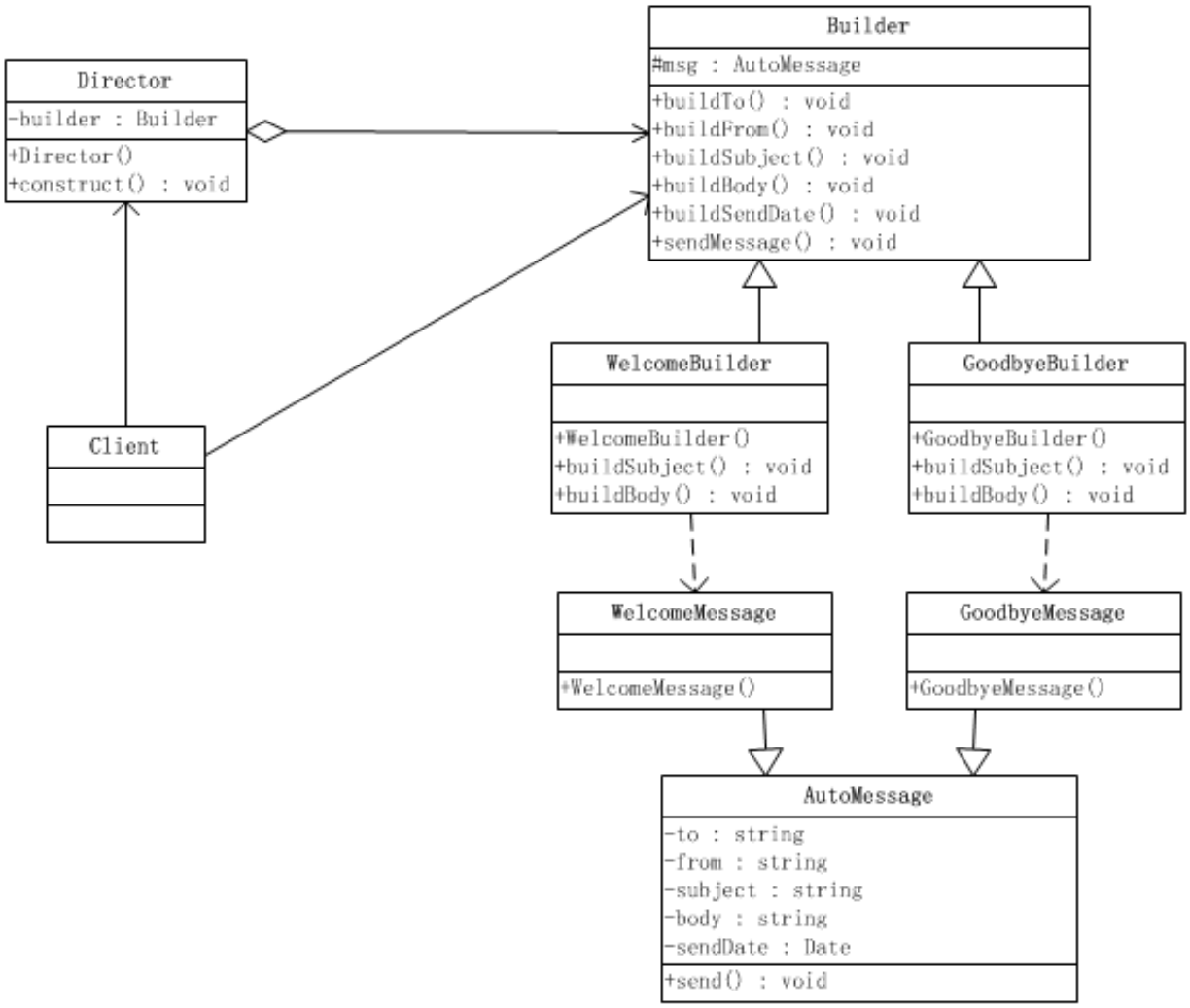
假设有一个电子杂志系统，定期地向用户的电子邮件信箱发送电子杂志。用户可以通过网页订阅电子杂志，也可以通过网页结束订阅。当客户开始订阅时，系统发送一个电子邮件表示欢迎，当客户结束订阅时，系统发送一个电子邮件表示欢送。本例子就是这个系统负责发送“欢迎”和“欢送”邮件的模块。

在本例中，产品类就是发给某个客户的“欢迎”和“欢送”邮件，如下图所示。



虽然在这个例子里面各个产品类均有一个共同的接口，但这仅仅是本例子特有的，并不代表建造模式的特点。建造模式可以应用到具有完全不同接口的产品类上。大多数情况下是不知道最终构建出来的产品是什么样的，所以在标准的建造模式里面，一般是不需要对产品定义抽象接口的，因为最终构造的产品千差万别，给这些产品定义公共接口几乎是没有意义的。

下图所示就是这个系统的类图。



这个系统含有客户端（Client）、导演者（Director）、抽象建造者（Builder）、具体建造者（WelcomeBuilder和GoodbyeBuilder）、产品（WelcomeMessage和GoodbyeMessage）等角色。

源代码

抽象类AutoMessage源代码，send()操作仅仅是示意性的，并没有给出任何发送电子邮件的代码。

```
public abstract class AutoMessage {
    //收件人地址
 private String to;
    //发件人地址
 private String from;
    //标题
 private String subject;
```

```
    //内容
private String body;
    //发送日期
private Date sendDate;
    public void send(){
        System.out.println("收件人地址: " + to);
        System.out.println("发件人地址: " + from);
        System.out.println("标题: " + subject);
        System.out.println("内容: " + body);
        System.out.println("发送日期: " + sendDate);
    }
    public String getTo() {
        return to;
    }
    public void setTo(String to) {
        this.to = to;
    }
    public String getFrom() {
        return from;
    }
    public void setFrom(String from) {
        this.from = from;
    }
    public String getSubject() {
        return subject;
    }
    public void setSubject(String subject) {
        this.subject = subject;
    }
    public String getBody() {
        return body;
    }
    public void setBody(String body) {
        this.body = body;
    }
    public Date getSendDate() {
        return sendDate;
    }
    public void setSendDate(Date sendDate) {
        this.sendDate = sendDate;
    }
```

```
}
```

具体产品类WelcomeMessage

```
public class WelcomeMessage extends AutoMessage {
    /**
     * 构造子
     */
    public WelcomeMessage(){
        System.out.println("发送欢迎信息");
    }
}
```

具体产品类GoodbyeMessage

```
public class GoodbyeMessage extends AutoMessage{
    /**
     * 构造子
     */
    public GoodbyeMessage(){
        System.out.println("发送欢送信息");
    }
}
```

抽象建造者类

```
public abstract class Builder {
    protected AutoMessage msg;
    //标题零件的建造方法
 public abstract void buildSubject();
    //内容零件的建造方法
 public abstract void buildBody();
    //收件人零件的建造方法
 public void buildTo(String to){
        msg.setTo(to);
    }
    //发件人零件的建造方法
 public void buildFrom(String from){
```

```
        msg.setFrom(from);
    }
    //发送时间零件的建造方法
 public void buildSendDate(){
        msg.setSendDate(new Date());
    }
    /**
     * 邮件产品完成后，用此方法发送邮件
     * 此方法相当于产品返还方法
     */
    public void sendMessage(){
        msg.send();
    }
}
```

具体建造者WelcomeBuilder

```
public class WelcomeBuilder extends Builder {
    public WelcomeBuilder(){
        msg = new WelcomeMessage();
    }
    @Override
    public void buildBody() {
        // TODO Auto-generated method stub
        msg.setBody("欢迎内容");
    }

    @Override
    public void buildSubject() {
        // TODO Auto-generated method stub
        msg.setSubject("欢迎标题");
    }

}
```

具体建造者GoodbyeBuilder

```
public class GoodbyeBuilder extends Builder {

    public GoodbyeBuilder(){
```

```
            msg = new GoodbyeMessage();
        }
        @Override
        public void buildBody() {
            // TODO Auto-generated method stub
            msg.setBody("欢送内容");
        }

        @Override
        public void buildSubject() {
            // TODO Auto-generated method stub
            msg.setSubject("欢送标题");
        }

    }
```

导演者Director，这个类提供一个construct()方法，此方法调用建造者的建造方法，包括buildTo()、buildFrom()、buildSubject()、buildBody()、buildSendDate()等，从而一部分一部分地建造出产品对象，既AutoMessage对象。

```
public class Director {
    Builder builder;
    /**
     * 构造子
     */
    public Director(Builder builder){
        this.builder = builder;
    }
    /**
     * 产品构造方法，负责调用各零件的建造方法
     */
    public void construct(String toAddress , String fromAddress){
        this.builder.buildTo(toAddress);
        this.builder.buildFrom(fromAddress);
        this.builder.buildSubject();
        this.builder.buildBody();
        this.builder.buildSendDate();
        this.builder.sendMessage();
    }
}
```

客户端Client

```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Builder builder = new WelcomeBuilder();
        Director director = new Director(builder);
        director.construct("toAddress@126.com", "fromAddress@126.com");

    }

}
```

建造模式分成两个很重要的部分：

1. 一个部分是Builder接口，这里是定义了如何构建各个部件，也就是知道每个部件功能如何实现，以及如何装配这些部件到产品中去；

2. 另外一个部分是Director，Director是知道如何组合来构建产品，也就是说Director负责整体的构建算法，而且通常是分步骤地来执行。

不管如何变化，**建造模式都存在这么两个部分，一个部分是部件构造和产品装配，另一个部分是整体构建的算法。**认识这点是很重要的，因为在建造模式中，强调的是固定整体构建的算法，而灵活扩展和切换部件的具体构造和产品装配的方式。

再直白点说，建造模式的重心在于分离构建算法和具体的构造实现，从而使得构建算法可以重用。具体的构造实现可以很方便地扩展和切换，从而可以灵活地组合来构造出不同的产品对象。

使用建造模式构建复杂对象

考虑这样一个实际应用，要创建一个保险合同的对象，里面很多属性的值都有约束，要求创建出来的对象是满足这些约束规则的。约束规则比如：保险合同通常情况下可以和个人签订，也可以和某个公司签订，但是一份保险合同不能同时与个人和公司签订。这个对象里有很多类似这样的约束，采用建造模式来构建复杂的对象，通常会对建造模式进行一定的简化，因为目标明确，就是创建某个复杂对象，因此做适当简化会使程序更简洁。大致简化如下：

- 由于是用Builder模式来创建某个对象，因此就没有必要再定义一个Builder接口，直接提供一个具体的建造者类就可以了。
- 对于创建一个复杂的对象，可能会有很多种不同的选择和步骤，干脆去掉“导演者”，把导演者的功能和Client的功能合并起来，也就是说,Client这个时候就相当于导演者，它来指导构建器类去构建需要的复杂对象。

保险合同类

```
/**
 * 保险合同对象
 */
public class InsuranceContract {
    //保险合同编号
    private String contractId;
    /**
     * 被保险人员的名称，同一份保险合同，要么跟人员签订，要么跟公司签订
     * 也就是说，“被保险人员”和“被保险公司”这两个属性，不可能同时有值
     */
    private String personName;
    //被保险公司的名称
    private String companyName;
    //保险开始生效日期
    private long beginDate;
    //保险失效日期，一定会大于保险开始生效日期
    private long endDate;
    //其他数据
    private String otherData;
    //私有构造方法
    private InsuranceContract(ConcreteBuilder builder){
        this.contractId = builder.contractId;
        this.personName = builder.personName;
        this.companyName = builder.companyName;
        this.beginDate = builder.beginDate;
        this.endDate = builder.endDate;
        this.otherData = builder.otherData;
    }
    /**
     * 保险合同的一些操作
     */
    public void someOperation(){
```

```
        System.out.println("当前正在操作的保险合同编号为【"+this.contractId+"
】");
    }

    public static class ConcreteBuilder{
        private String contractId;
        private String personName;
        private String companyName;
        private long beginDate;
        private long endDate;
        private String otherData;
        /**
         * 构造方法，传入必须要有的参数
         * @param contractId     保险合同编号
         * @param beginDate        保险合同开始生效日期
         * @param endDate        保险合同失效日期
         */
        public ConcreteBuilder(String contractId,long beginDate,long end
Date){
            this.contractId = contractId;
            this.beginDate = beginDate;
            this.endDate = endDate;
        }
        //被保险人员的名称
        public ConcreteBuilder setPersonName(String personName) {
            this.personName = personName;
            return this;
        }
        //被保险公司的名称
        public ConcreteBuilder setCompanyName(String companyName) {
            this.companyName = companyName;
            return this;
        }
        //其他数据
        public ConcreteBuilder setOtherData(String otherData) {
            this.otherData = otherData;
            return this;
        }
        /**
         * 构建真正的对象并返回
         * @return    构建的保险合同对象
```

```
         */
        public InsuranceContract build(){
            if(contractId == null || contractId.trim().length()==0){
                throw new IllegalArgumentException("合同编号不能为空");
            }
            boolean signPerson = (personName != null && personName.trim(
).length() > 0);
            boolean signCompany = (companyName != null && companyName.tr
im().length() > 0);
            if(signPerson && signCompany){
                throw new IllegalArgumentException("一份保险合同不能同时与个
人和公司签订");
            }
            if(signPerson == false && signCompany == false){
                throw new IllegalArgumentException("一份保险合同不能没有签订
对象");
            }
            if(beginDate <= 0 ){
                throw new IllegalArgumentException("一份保险合同必须有开始生
效的日期");
            }
            if(endDate <=0){
                throw new IllegalArgumentException("一份保险合同必须有失效的
日期");
            }
            if(endDate < beginDate){
                throw new IllegalArgumentException("一份保险合同的失效日期必
须大于生效日期");
            }
            return new InsuranceContract(this);
        }
    }
}
```

客户端类

```
public class Client {
    public static void main(String[]args){
        //创建构建器对象
        InsuranceContract.ConcreteBuilder builder =
```

```
            new InsuranceContract.ConcreteBuilder("9527", 123L, 456L);
        //设置需要的数据，然后构建保险合同对象
        InsuranceContract contract =
            builder.setPersonName("小明").setOtherData("test").build();
        //操作保险合同对象的方法
        contract.someOperation();
    }
}
```

在本例中将具体建造者合并到了产品对象中，并将产品对象的构造函数私有化，防止客户端不使用构建器来构建产品对象，而是直接去使用new来构建产品对象所导致的问题。另外，这个构建器的功能就是为了创建被构建的对象，完全可以不用单独一个类。

在什么情况下使用建造模式

1. 需要生成的产品对象有复杂的内部结构，每一个内部成分本身可以是对象，也可以仅仅是一个对象（即产品对象）的一个组成部分。

2. 需要生成的产品对象的属性相互依赖。建造模式可以强制实行一种分步骤进行的建造过程，因此，如果产品对象的一个属性必须在另一个属性被赋值之后才可以被赋值，使用建造模式是一个很好的设计思想。

3. 在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到。