互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 13 章　原型模式

本章内容

- ❑ 个性化电子账单
- ❑ 原型模式的定义
- ❑ 原型模式的应用
- ❑ 原型模式的注意事项
- ❑ 最佳实践

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.1 个性化电子账单

现在电子账单越来越流行了，比如你的信用卡，到月初的时候银行就会发一份电子邮件到你邮箱中，说你这个月消费了多少，什么时候消费的，积分是多少等等，这是每个月发一次，还有一种也是银行发的邮件你肯定非常有印象：广告信，现在各大银行的信用卡部门都在拉拢客户，电子邮件是一种廉价、快捷的通讯方式，你用纸质的广告信那个费用多高呀，比如我行今天推出一个信用卡刷卡抽奖活动，通过电子账单系统可以一个晚上发送给 600 万客户，为什么要用电子账单系统呢？直接找个发垃圾邮件工具不就解决问题了吗？是个好主意，但是这个方案在金融行业是行不通的，为什么？因为银行发送该类邮件是有要求的：

- 个性化服务

一般银行都要求个性化服务，发过去的邮件上总有一些个人信息吧，比如“XX 先生”，“XX 女士”等等。

- 递送成功率

邮件的递送成功率有一定的要求，由于大批量的发送邮件会被接收方邮件服务器误认是垃圾邮件，因此在邮件头要增加一些伪造数据，以规避被反垃圾邮件引擎误认为是垃圾邮件。

从这两方面考虑广告信的发送也是电子账单系统（电子账单系统一般包括：账单分析、账单生成器、广告信管理、发送队列管理、发送机、退信处理、报表管理等）的一个子功能，我们今天就来考虑一下广告信这个模块怎么开发的。那既然是广告信，肯定需要一个模版，然后再从数据库中把客户的信息一个一个的取出，放到模版中生成一份完整的邮件，然后扔给发送机进行发送处理，类图如图 13-1 所示。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

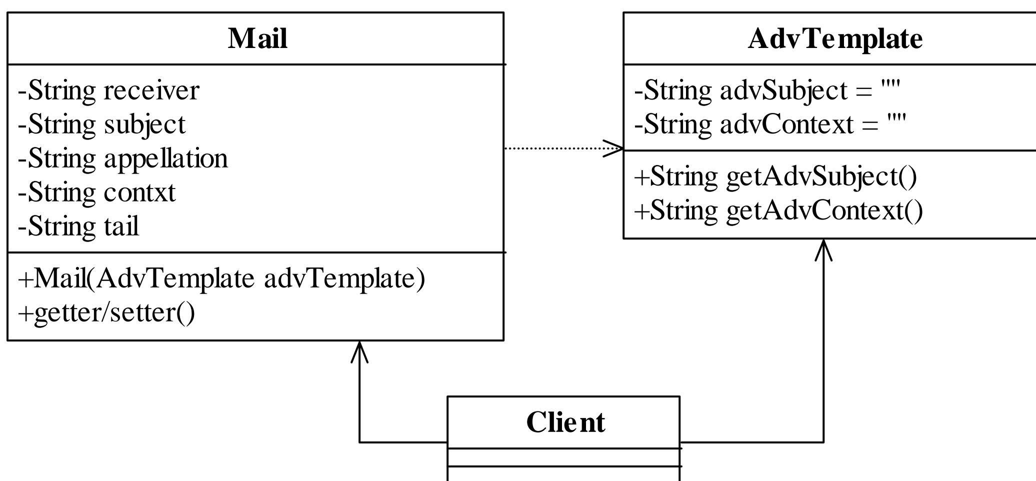

图13-1　发送电子账单类图

在类图中 AdvTemplate 是广告信的模板，一般都是从数据库取出，生成一个 BO 或者是 DTO，我们这里使用一个静态的值来作代表；Mail 类是一封邮件类，发送机发送的就是这个类。我们先来看 AdvTemplate，如代码清单 13-1 所示。

代码清13-1　广告信模版代码

```
public class AdvTemplate {
      //广告信名称
      private String advSubject ="XX 银行国庆信用卡抽奖活动";
      //广告信内容
      private String advContext = "国庆抽奖活动通知：只要刷卡就送你 1 百万！....";
      //取得广告信的名称
      public String getAdvSubject(){
              return this.advSubject;
      }
      //取得广告信的内容
      public String getAdvContext(){
              return this.advContext;
      }
}
```

邮件类 Mail 如代码清单 13-2 所示。

代码清13-2　邮件类代码

```
public class Mail {
      //收件人
      private String receiver;
      //邮件名称
      private String subject;
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
    //称谓
    private String appellation;
    //邮件内容
    private String contxt;
    //邮件的尾部，一般都是加上"XXX 版权所有"等信息
    private String tail;
    //构造函数
    public Mail(AdvTemplate advTemplate){
            this.contxt = advTemplate.getAdvContext();
            this.subject = advTemplate.getAdvSubject();
    }
    //以下为 getter/setter 方法
    public String getReceiver() {
            return receiver;
    }

    public void setReceiver(String receiver) {
            this.receiver = receiver;
    }

    public String getSubject() {
            return subject;
    }

    public void setSubject(String subject) {
            this.subject = subject;
    }

    public String getAppellation() {
            return appellation;
    }

    public void setAppellation(String appellation) {
            this.appellation = appellation;
    }

    public String getContxt() {
            return contxt;
    }

    public void setContxt(String contxt) {
            this.contxt = contxt;
    }
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        public String getTail() {
                return tail;
        }

        public void setTail(String tail) {
                this.tail = tail;
        }
   }
```

Mail 类就是一个业务对象，虽然比较长，还是比较简单的。我们再来看业务场景类是如何对邮件继续处理的，如代码清单 11-3 所示。

代码清13-3　场景类

```
   public class Client {
        //发送账单的数量，这个值是从数据库中获得
        private static int MAX_COUNT = 6;

        public static void main(String[] args) {
                //模拟发送邮件
                int i=0;
                //把模板定义出来，这个是从数据库中获得
                Mail mail = new Mail(new AdvTemplate());
                mail.setTail("XX 银行版权所有");
                while(i<MAX_COUNT){
                        //以下是每封邮件不同的地方
                        mail.setAppellation(getRandString(5)+" 先生（女士）");
                        mail.setReceiver(getRandString(5) + "@" +
getRandString(8)+".com");
                        //然后发送邮件
                        sendMail(mail);
                        i++;
                }
        }
        //发送邮件
        public static void sendMail(Mail mail){
                System.out.println("标题："+mail.getSubject() + "\t 收件人：
"+mail.getReceiver()+"\t....发送成功！");
        }
        //获得指定长度的随机字符串
        public static String getRandString(int maxLength){
                String source
="abcdefghijklmnopqrskuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
                StringBuffer sb = new StringBuffer();
                Random rand = new Random();
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
                for(int i=0;i<maxLength;i++){

        sb.append(source.charAt(rand.nextInt(source.length())));
                }
                return sb.toString();
        }
}
```

运行结果如下所示。

```
标题：XX 银行国庆信用卡抽奖活动        收件人：fjQUm@ZnkyPSsL.com ....发送成功！
标题：XX 银行国庆信用卡抽奖活动        收件人：ZIKnC@NOKdloNM.com ....发送成功！
标题：XX 银行国庆信用卡抽奖活动        收件人：zNkMI@HpMMSZaz.com ....发送成功！
标题：XX 银行国庆信用卡抽奖活动        收件人：oMTFA@uBwkRjxa.com ....发送成功！
标题：XX 银行国庆信用卡抽奖活动        收件人：TquWT@TLLVNFja.com ....发送成功！
标题：XX 银行国庆信用卡抽奖活动        收件人：rkQbp@mfATHDQH.com ....发送成功！
```

由于是随机数，每次运行都有所差异，不管怎么样，我们这个电子账单发送程序是编写出来了，也能正常发送出来。我们再来仔细的想想，这个程序是否有问题？Look here，这是一个线程在运行，也就是你发送是单线程的，那按照一封邮件发出去需要 0.02 秒（够小了，你还要到数据库中取数据呢），600 万封邮件需要……我算算（掰指头计算中……），恩，是 33 个小时，也就是一个整天都发送不完毕，今天发送不完毕，明天的账单又产生了，积累积累，激起甲方人员一堆抱怨，那怎么办？

好办，把 sendMail 修改为多线程，但是你只把 sendMail 修改为多线程还是有问题的呀，你看哦，产生第一封邮件对象，放到线程 1 中运行，还没有发送出去；线程 2 呢也启动了，直接就把邮件对象 mail 的收件人地址和称谓修改掉了，线程不安全了，好了，说到这里，你会说这有 N 多种解决办法，我们不多说，我们今天就说一种，使用一种新型模式来解决这个问题：对象的拷贝功能来解决这个问题，类图稍作修改，如图 13-2 所示。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

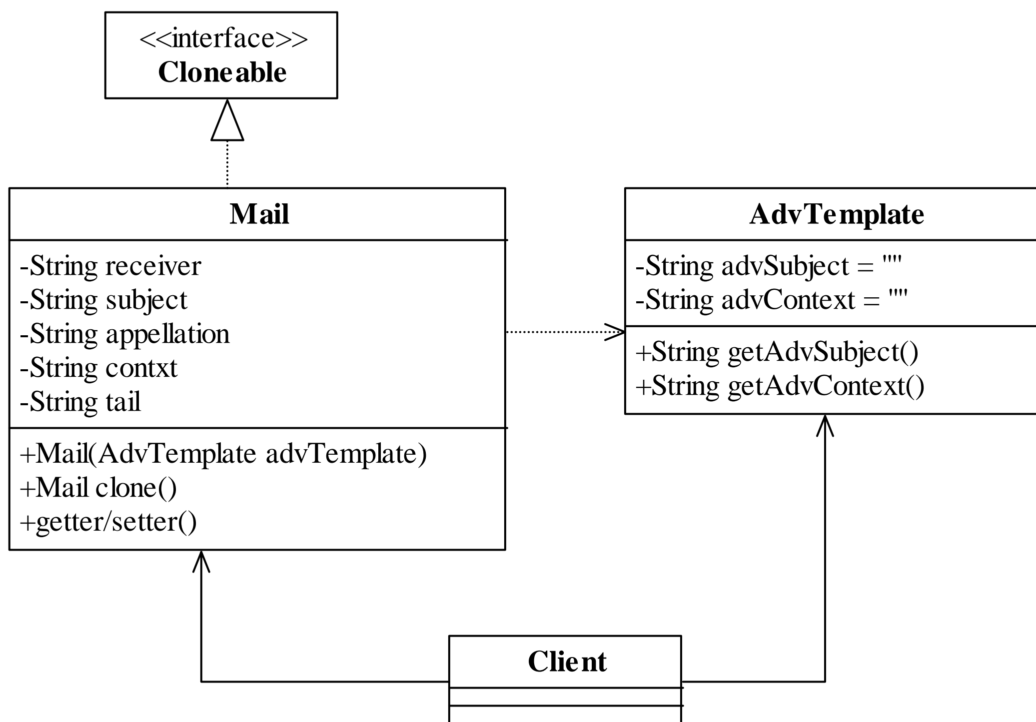

图13-2　修正后的发送电子账单类图

增加了一个 Cloneable 接口（Java 自带的一个接口）， Mail 实现了这个接口，在 Mail 类中覆写 clone()方法，我们来看 Mail 类的改变，如代码清单 13-4 所示。

代码清13-4　修正后的邮件类

```
public class Mail implements Cloneable{
        //收件人
        private String receiver;
        //邮件名称
        private String subject;
        //称谓
        private String appellation;
        //邮件内容
        private String contxt;
        //邮件的尾部，一般都是加上"XXX 版权所有"等信息
        private String tail;
        //构造函数
        public Mail(AdvTemplate advTemplate){
                this.contxt = advTemplate.getAdvContext();
                this.subject = advTemplate.getAdvSubject();
        }

        @Override
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
public Mail clone(){
        Mail mail =null;
        try {
                mail = (Mail)super.clone();
        } catch (CloneNotSupportedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
        return mail;
}

//以下为 getter/setter 方法
public String getReceiver() {
        return receiver;
}

public void setReceiver(String receiver) {
        this.receiver = receiver;
}

public String getSubject() {
        return subject;
}

public void setSubject(String subject) {
        this.subject = subject;
}

public String getAppellation() {
        return appellation;
}

public void setAppellation(String appellation) {
        this.appellation = appellation;
}

public String getContxt() {
        return contxt;
}

public void setContxt(String contxt) {
        this.contxt = contxt;
}
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        public String getTail() {
                return tail;
        }

        public void setTail(String tail) {
                this.tail = tail;
        }
}
```

注意看黑体部分，实现了一个接口，并重写了 clone 方法，大家可能看着这个类有点奇怪，先保留你的好奇，我们继续讲下去，稍后会给你清晰的解答。我们再来看场景 Client 的变化，如代码清单 13-5 所示。

代码清13-5　修正后的场景类

```
public class Client {
        //发送账单的数量，这个值是从数据库中获得
        private static int MAX_COUNT = 6;

        public static void main(String[] args) {
                //模拟发送邮件
                int i=0;
                //把模板定义出来，这个是从数据中获得
                Mail mail = new Mail(new AdvTemplate());
                mail.setTail("XX 银行版权所有");
                while(i<MAX_COUNT){
                        //以下是每封邮件不同的地方
                        Mail cloneMail = mail.clone();
                        cloneMail.setAppellation(getRandString(5)+" 先生（女士）");
                        cloneMail.setReceiver(getRandString(5) + "@" + getRandString(8)+".com");
                        //然后发送邮件
                        sendMail(cloneMail);
                        i++;
                }
        }
}
```

运行结果不变，一样完成了电子广告信的发送功能，而且 sendMail 即使是多线程也没有关系。注意看 Client 类中的黑体字 mail.clone()这个方法了吗？把对象拷贝一份，产生一个新的对象，和原有对象一样，然后再修改细节的数据，如设置称谓、设置收件人地址等等。这种不通过 new 关键字来产生一个对象，而是通过对象拷贝来实现的模式就叫做原型模式。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

13.2 原型模式的定义

原型模式（Prototype Pattern）的简单程度是仅次于单例模式和迭代器模式，正是由于简单，使用的场景才非常的多，其定义如下：

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

原型模式的通用类图如图 13-3 所示。

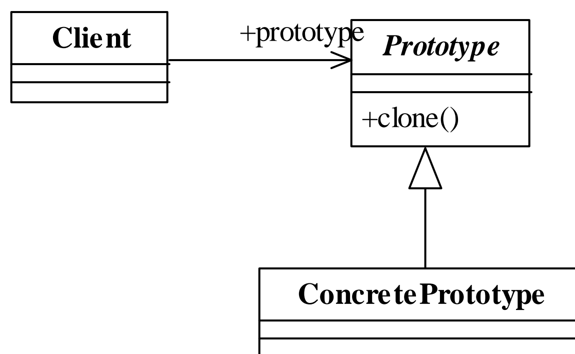

图13-3　原型模式的通用类图

简单，太简单了，原型模式的核心是一个 clone 方法，通过该方法进行对象的拷贝，Java 提供了一个 Cloneable 接口来标示这个对象是可拷贝的，为什么说是“标示”呢？翻开 JDK 的帮助看看 Cloneable 是一个方法都没有的，这个接口只是一个标记作用，在 JVM 中具有这个标记的对象才有可能被拷贝，那怎么才能从“有可能被拷贝”转换为“可以被拷贝”呢？方法是覆盖 clone()方法，是的，你没有看错是重写 clone()方法，看看我们上面 Mail 类中的 clone 方法，如代码清单 13-6 所示。

代码清13-6　邮件类中的 clone 方法

```
@Override
public Mail clone(){}
```

读者请注意，在 clone()方法上增加了一个注解@Override，没有继承一个类为什么可以覆写呢？想想看，在 Java 中所有类的老祖宗是谁？对嘛，Object 类，每个类默认都是继承了这个类，所以这个用上覆写是非常正确的，——覆写了 Object 类中的 clone 方法！

在 Java 中原型模式是如此的简单，我们来看通用源代码，如代码清单 13-7 所示。

代码清13-7　原型模式通用源码

```
public class PrototypeClass  implements Cloneable{
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        //覆写父类 Object 方法
        @Override
        public PrototypeClass clone(){
                PrototypeClass prototypeClass = null;
                try {
                        prototypeClass = (PrototypeClass)super.clone();
                } catch (CloneNotSupportedException e) {
                        //异常处理
                }
                return prototypeClass;
        }
}
```

实现一个接口，然后重写 clone 方法，就完成了原型模式！

13.3 原型模式的应用

13.3.1 原型模式的优点

- 性能优良

原型模式是在内存二进制流的拷贝，要比直接 new 一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好的体现其优点。

- 逃避构造函数的约束

这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的（见“原型模式的注意事项”），优点就是减少了约束，缺点也是减少了约束，双刃剑，需要大家在实际应用时考虑。

13.3.2 原型模式的使用场景

- 资源优化场景

类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。

- 性能和安全要求的场景

通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。

- 一个对象多个修改者的场景

一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过 clone 的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与 Java 融为浑然一体，大家可以随手拿来使用。

13.4 原型模式的注意事项

原型模式虽然很简单，但是在 Java 中使用原型模式也就是 clone 方法还是有一些注意事项的，我们通过几个例子一个一个解说（如果你对 Java 不是很感冒的话，可以跳开以下部分）。

13.4.1 构造函数不会被执行

一个实现了 Cloneable 并重写了 clone 方法的类 A,有一个无参构造或有参构造 B，通过 new 关键字产生了一个对象 S，再然后通过 S.clone()方式产生了一个新的对象 T，那么在对象拷贝时构造函数 B 是不会被执行的，我们来写一小段程序来说明这个问题，如代码清单 13-8 所示。

代码清13-8　简单的可拷贝对象

```
public class Thing implements Cloneable{
	public Thing(){
		System.out.println("构造函数被执行了...");
	}

	@Override
	public Thing clone(){
		Thing thing=null;
		try {
			thing = (Thing)super.clone();
		} catch (CloneNotSupportedException e) {
			e.printStackTrace();
		}
		return thing;
	}
}
```

然后我们再来写一个 Client 类，进行对象的拷贝，如代码清单 13-9 所示。

代码清13-9　简单的场景类

```
public class Client {

	public static void main(String[] args) {
		//产生一个对象
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
                Thing thing = new Thing();
                //拷贝一个对象
                Thing cloneThing = thing.clone();
        }
}
```

运行结果如下所示。

```
构造函数被执行了...
```

对象拷贝时构造函数确实没有被执行，这点从原理来讲也是可以讲得通的，Object 类的 clone 方法的原理是从内存中（具体的说就是堆内存）以二进制流的方式进行拷贝，重新分配一个内存块，那构造函数没有被执行也是非常正常的了。

13.4.2 浅拷贝和深拷贝

在解释什么是浅拷贝什么是深拷贝前，我们先来看个例子，如代码清单 13-10 所示。

代码清13-10 浅拷贝

```
public class Thing implements Cloneable{
        //定义一个私有变量
        private ArrayList<String> arrayList = new ArrayList<String>();

        @Override
        public Thing clone(){
                Thing thing=null;
                try {
                        thing = (Thing)super.clone();
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return thing;
        }
        //设置 HashMap 的值
        public void setValue(String value){
                this.arrayList.add(value);
        }
        //取得 arrayList 的值
        public ArrayList<String> getValue(){
                return this.arrayList;
        }
}
```

在 Thing 类中增加一个私有变量 arrayLis，类型为 ArrayList,然后通过 setValue 和 getValue 分别

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

进行设置和取值，我们来看场景类是如何拷贝的，如代码清单 13-11 所示。

代码清13-11 浅拷贝测试

```
public class Client {

    public static void main(String[] args) {
        //产生一个对象
        Thing thing = new Thing();
        //设置一个值
        thing.setValue("张三");
        //拷贝一个对象
        Thing cloneThing = thing.clone();
        cloneThing.setValue("李四");
        System.out.println(thing.getValue());
    }
}
```

读者猜想一下运行结果应该是什么？是仅一个“张三”吗？运行结果如下所示。

```
[张三, 李四]
```

怎么会这样呢？怎么会有李四呢？让我来给你解释，是因为 Java 做了一个偷懒的拷贝动作，Object 类提供的方法 clone 只是拷贝本对象，其对象内部的数组、引用对象等都不拷贝，还是指向原生对象的内部元素地址，这种拷贝就叫做浅拷贝，确实是非常浅，两个对象共享了一个私有变量，你改我改大家都能改，是一种非常不安全的方式，在实际项目中使用还是比较少的（当然，这是也是一种“危机”环境的一种救命方式）。你可能会比较奇怪，为什么在 Mail 那个类中就可以使用 String 类型，而不会产生由浅拷贝带来的问题呢？内部的数组和引用对象才不拷贝，其他的原始类型比如 int,long,String(Java 就希望你把 String 认为是基本类型，String 是没有 clone 方法的)等都会被拷贝的。

注意 使用 clone 方法拷贝时，满足两个条件的对象才不会被拷贝：一是类的成员变量，而不是方法内的变量；二是必须是一个对象，而不是一个原始类型

浅拷贝是有风险的，那怎么才能深入的拷贝呢？我们修改一下程序就可以深拷贝，如代码清单 13-12 所示。

代码清13-12 深拷贝

```
public class Thing implements Cloneable{
    //定义一个私有变量
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        private ArrayList<String> arrayList = new ArrayList<String>();

        @Override
        public Thing clone(){
                Thing thing=null;
                try {
                        thing = (Thing)super.clone();
                        thing.arrayList = 
(ArrayList<String>)this.arrayList.clone();
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return thing;
        }
    }
```

仅仅增加了黑体部分，对私有的类变量进行独立的拷贝。Client 类没有任何改变，运行结果如下所示。

```
[张三]
```

该方法就实现了完全的拷贝，两个对象之间没有任何的瓜葛了，你修改你的，我修改我的，不相互影响，这种拷贝就叫做深拷贝，深拷贝还有一种实现方式就是通过自己写二进制流来操作对象，然后实现对象的深拷贝，这个大家有时间自己实现一下。

注意 深拷贝和浅拷贝建议不要混合使用，特别是是在涉及到类的继承，父类有多个引用的情况就非常的复杂，建议的方案是深拷贝和浅拷贝分开实现。

13.4.3 clone 与 final 两对冤家

俗话说，不是冤家不聚头，对象的 clone 与对象内的 final 关键字是有冲突的，我们举例来说明这个问题，如代码清单 13-13 所示。

代码清13-13 增加 final 关键字的拷贝

```
public class Thing implements Cloneable{
        //定义一个私有变量
        private final ArrayList<String> arrayList = new ArrayList<String>();

        @Override
```

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章


```
        public Thing clone(){
                Thing thing=null;
                try {
                        thing = (Thing)super.clone();
                        this.arrayList = 
(ArrayList<String>)this.arrayList.clone();
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return thing;
        }
   }
```

黑体部分仅仅增加了一个 final 关键字，然后编译器就报斜体部分错误，正常呀，final 类型你还想重设值呀！完蛋了，你要实现深拷贝的梦想在 final 关键字的威胁下破灭了,路总是有的，我们来想想怎么修改这个方法：删除掉 final 关键字，这是最便捷最安全最快速的方式。你要使用 clone 方法就在类的成员变量上不要增加 final 关键字。

注意　要使用 clone 方法，类的成员变量上不要增加 final 关键字。

13.4 最佳实践

原型模式先产生出一个包含大量共有信息的类，然后可以拷贝出副本，修正细节信息，建立了一个完整的个性对象。不知道大家有没有看过施瓦辛格演的《第六日》这个电影，电影的主线也就是一个人被复制，然后正本和副本对掐，我们今天讲的原型模式也就是由一个正本可以创建多个副本的概念，可以这样理解:一个对象的产生可以不由零起步，直接从一个已经具备一定雏形的对象克隆，然后再修改为生产需要的对象。也就是说，产生一个人，可以不从 1 岁长到 2 岁，再 3 岁…，也可以直接找一个人，从其身上获得 DNA，然后克隆一个，直接修改一下就是 30 岁了!，我们讲的原型模式也就是这样的功能，是紧跟时代潮流的哇！