原型模式(Prototype Pattern)：使用原型实例指定创建对象的种类，并且通过拷贝（深、浅）这些原型创建新的对象。

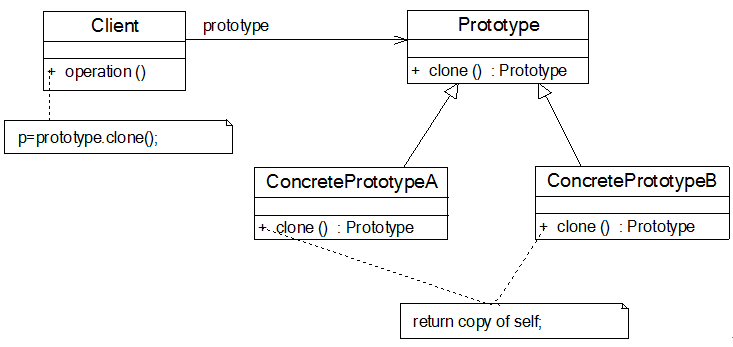
在使用原型模式时，我们需要首先创建一个原型对象，再通过复制这个原型对象来创建更多同类型的对象。

原型模式是一种对象创建型模式。

原型模式的**工作原理**很简单：将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝自己来实现创建过程。由于在软件系统中我们经常会遇到需要创建多个相同或者相似对象的情况，因此原型模式在真实开发中的使用频率还是非常高的。原型模式是一种“另类”的创建型模式，创建克隆对象的工厂就是原型类自身，工厂方法由克隆方法来实现。

需要注意的是通过克隆方法所创建的对象是全新的对象，它们在内存中拥有新的地址，通常对克隆所产生的对象进行修改对原型对象不会造成任何影响，每一个克隆对象都是相互独立的。通过不同的方式修改可以得到一系列相似但不完全相同的对象。

原型模式的结构如图所示：



在原型模式结构图中包含如下几个角色：

      ● Prototype（抽象原型类）：它是声明克隆方法的接口，是所有具体原型类的公共父类，可以是抽象类也可以是接口，甚至还可以是具体实现类。

      ● ConcretePrototype（具体原型类）：它实现在抽象原型类中声明的克隆方法，在克隆方法中返回自己的一个克隆对象。

      ● Client（客户类）：让一个原型对象克隆自身从而创建一个新的对象，在客户类中只需要直接实例化或通过工厂方法等方式创建一个原型对象，再通过调用该对象的克隆方法即可得到多个相同的对象。由于客户类针对抽象原型类Prototype编程，因此用户可以根据需要选择具体原型类，系统具有较好的可扩展性，增加或更换具体原型类都很方便。

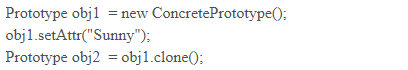
两种在Java语言中常用的克隆实现方法：

1. 通用实现方法

通用的克隆实现方法是在具体原型类的克隆方法中实例化一个与自身类型相同的对象并将其返回，并将相关的参数传入新创建的对象中，保证它们的成员属性相同。示意代码如下所示：



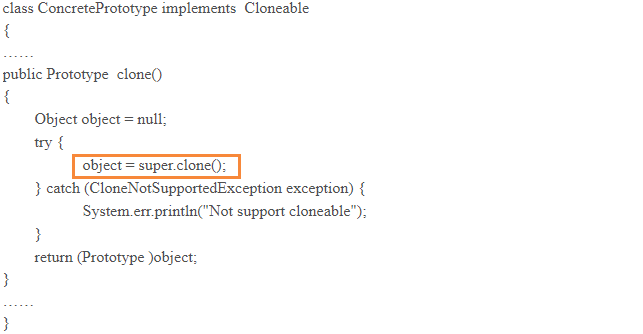
在客户类中我们只需要创建一个ConcretePrototype对象作为原型对象，然后调用其clone()方法即可得到对应的克隆对象，如下代码所示：



这种方法可作为原型模式的通用实现，它与编程语言特性无关，任何面向对象语言都可以使用这种形式来实现对原型的克隆。

2. Java语言中Object类的clone()方法

需要注意的是能够实现克隆的Java类必须实现一个标识接口Cloneable，表示这个Java类**支持被复制**。如果一个类没有实现这个接口但是调用了clone()方法，Java编译器将抛出一个CloneNotSupportedException异常。如下代码所示：



在客户端创建原型对象和克隆对象也很简单，如下代码所示：



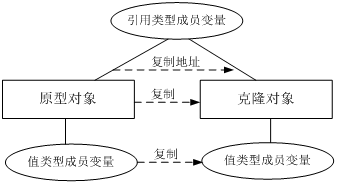
[Java提高篇——对象克隆（复制）](https://www.cnblogs.com/Qian123/p/5710533.html" \l "_label0)

以上算是浅克隆，仅复制了变量的引用并没有开辟新的内存空间，将值复制后再将引用返回给新对象。这样的坏处就是对象里边的对象引用还是指向复制前的。

浅克隆(ShallowClone)和深克隆(DeepClone)。在Java语言中，数据类型分为值类型（基本数据类型）和引用类型，值类型包括int、double、byte、boolean、char等简单数据类型，引用类型包括类、接口、数组等复杂类型。浅克隆和深克隆的主要区别在于**是否支持引用类型的成员变量的复制。**

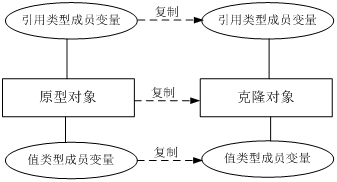
1. 浅克隆(ShallowClone)

如果原型对象的成员变量是值类型，将复制一份给克隆对象；如果原型对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，也就是说原型对象和克隆对象的成员变量指向相同的内存地址。简单来说，在浅克隆中，当对象被复制时只复制它本身和其中包含的值类型的成员变量，而引用类型的成员对象并没有复制，如图所示：



2. 深克隆(DeepClone)

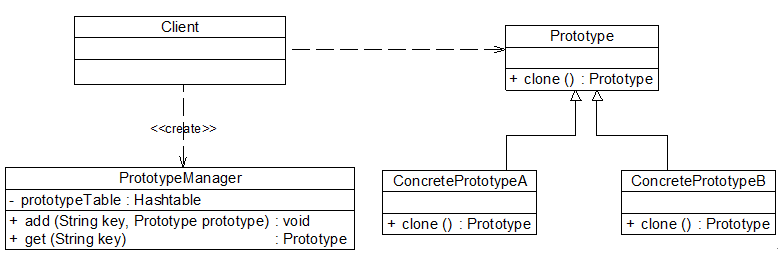
无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象，深克隆将原型对象的所有引用对象也复制一份给克隆对象。简单来说，在深克隆中，除了对象本身被复制外，对象所包含的所有成员变量也将复制，如图所示：



在Java语言中，如果需要实现深克隆，可以通过序列化(Serialization)等方式来实现。序列化就是将对象写到流的过程，写到流中的对象是原有对象的一个拷贝，而原对象仍然存在于内存中。通过序列化实现的拷贝不仅可以复制对象本身，而且可以复制其引用的成员对象，因此**通过序列化将对象写到一个流中，再从流里将其读出来，可以实现深克隆**。需要注意的是能够实现序列化的对象其类必须实现Serializable接口，否则无法实现序列化操作。

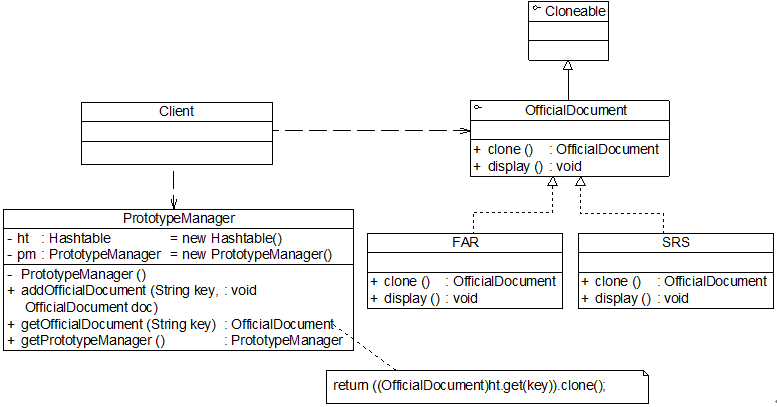
深克隆技术实现了原型对象和克隆对象的完全独立，对任意克隆对象的修改都不会给其他对象产生影响，是一种更为理想的克隆实现方式。

原型管理器(Prototype Manager)是将多个原型对象存储在一个集合中供客户端使用，它是一个专门负责克隆对象的工厂，其中定义了一个集合用于存储原型对象，如果需要某个原型对象的一个克隆，可以通过复制集合中对应的原型对象来获得。在原型管理器中针对抽象原型类进行编程，以便扩展。

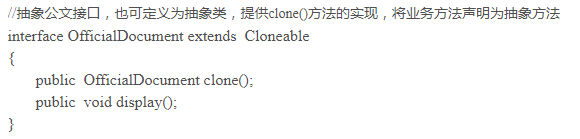


例子：日常办公中有许多公文需要创建、递交和审批，例如《可行性分析报告》、《立项建议书》、《软件需求规格说明书》、《项目进展报告》等，为了提高工作效率，在OA系统中为各类公文均创建了模板，用户可以通过这些模板快速创建新的公文，这些公文模板需要统一进行管理，系统根据用户请求的不同生成不同的新公文。

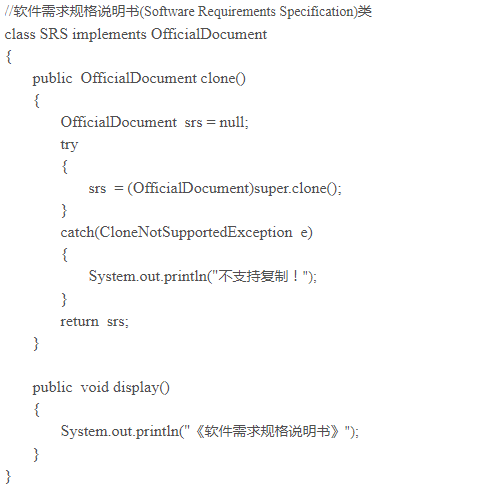
其结构设计如图。

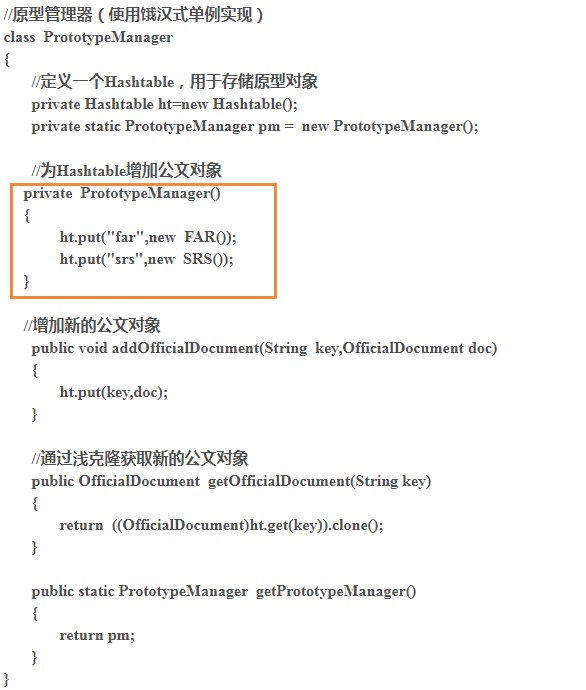


代码实现：

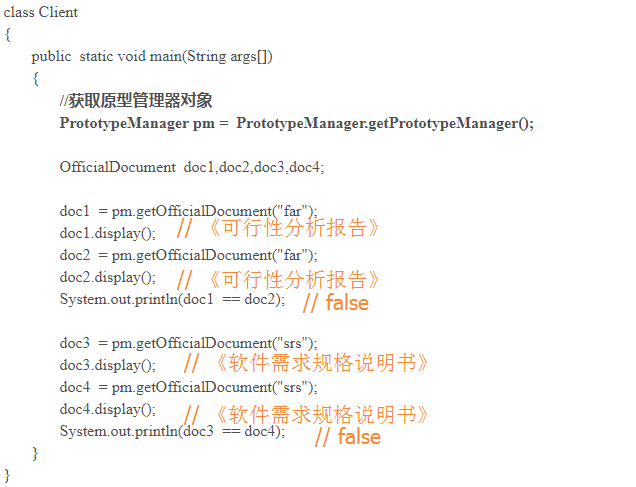








客户端代码如下所示：



在PrototypeManager中定义了一个Hashtable类型的集合对象，使用“键值对”来存储原型对象，客户端可以通过Key（如“far”或“srs”）来获取对应原型对象的克隆对象。PrototypeManager类提供了类似工厂方法的getOfficialDocument()方法用于返回一个克隆对象。在本实例代码中，我们将PrototypeManager设计为单例类，使用饿汉式单例实现，确保系统中有且仅有一个PrototypeManager对象，有利于节省系统资源，并可以更好地对原型管理器对象进行控制。

**原型模式总结**

1. 主要优点

(1) 当创建新的对象实例较为复杂时，使用原型模式可以简化对象的创建过程，通过复制一个已有实例可以提高新实例的创建效率。

(2) 扩展性较好，由于在原型模式中提供了抽象原型类，在客户端可以针对抽象原型类进行编程，而将具体原型类写在配置文件中，增加或减少产品类对原有系统都没有任何影响。

(3) 原型模式提供了简化的创建结构，工厂方法模式常常需要有一个与产品类等级结构相同的工厂等级结构，而原型模式就不需要这样，原型模式中产品的复制是通过封装在原型类中的克隆方法实现的，无须专门的工厂类来创建产品。

(4) 可以使用深克隆的方式保存对象的状态，使用原型模式将对象复制一份并将其状态保存起来，以便在需要的时候使用（如恢复到某一历史状态），可辅助实现撤销操作。

2. 主要缺点

(1) 需要为每一个类配备一个克隆方法，而且该克隆方法位于一个类的内部，当对已有的类进行改造时，需要修改源代码，违背了“开闭原则”。

(2) 在实现深克隆时需要编写较为复杂的代码，而且当对象之间存在多重的嵌套引用时，为了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来可能会比较麻烦。

3. 适用场景

(1) 创建新对象成本较大（如初始化需要占用较长的时间，占用太多的CPU资源或网络资源），新的对象可以通过原型模式对已有对象进行复制来获得，如果是相似对象，则可以对其成员变量稍作修改。

(2) 如果系统要保存对象的状态，而对象的状态变化很小，或者对象本身占用内存较少时，可以使用原型模式配合备忘录模式来实现。

(3) 需要避免使用分层次的工厂类来创建分层次的对象，并且类的实例对象只有一个或很少的几个组合状态，通过复制原型对象得到新实例可能比使用构造函数创建一个新实例更加方便。

[原型模式](https://blog.csdn.net/LoveLion/article/details/17517213)

[对象的克隆——原型模式（一）：大同小异的工作周报，原型模式概述](https://blog.csdn.net/lovelion/article/details/7424559)

[对象的克隆——原型模式（二）：工作周报的原型模式解决方案](https://blog.csdn.net/lovelion/article/details/7424594)

[对象的克隆——原型模式（三）：带附件的周报【浅克隆，深克隆】](https://blog.csdn.net/lovelion/article/details/7424620)

[对象的克隆——原型模式（四）：原型管理器的引入和实现，原型模式总结](https://blog.csdn.net/lovelion/article/details/7424623)