原型模式

原型模式（Prototype Pattern）用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。

举例：

假设平时我们都要记笔记，自然有一些固定的格式，那我们是不是就没有必要辛辛苦苦写格式，是不是就可以直接将写好的当成模板进行修改呢。

介绍

意图

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。

主要解决

在运行期建立和删除原型。

何时使用

1、当一个系统应该独立于它的产品创建，构成和表示时。

2、当要实例化的类是在运行时刻指定时，例如，通过动态装载。

3、为了避免创建一个与产品类层次平行的工厂类层次时。

4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

如何解决

利用已有的一个原型对象，快速地生成和原型对象一样的实例。

关键代码

1、实现克隆操作，在 JAVA 继承 Cloneable，重写 clone()，在 .NET 中可以使用 Object 类的 MemberwiseClone() 方法来实现对象的浅拷贝或通过序列化的方式来实现深拷贝。

2、原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些"易变类"拥有稳定的接口。

应用实例

1、细胞分裂。

2、JAVA 中的 Object clone() 方法。

优点

1、性能提高。

2、逃避构造函数的约束。

缺点

1、配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。

2、必须实现 Cloneable 接口。

使用场景

1、资源优化场景。

2、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。

3、性能和安全要求的场景。

4、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。

5、一个对象多个修改者的场景。

6、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。

7、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。

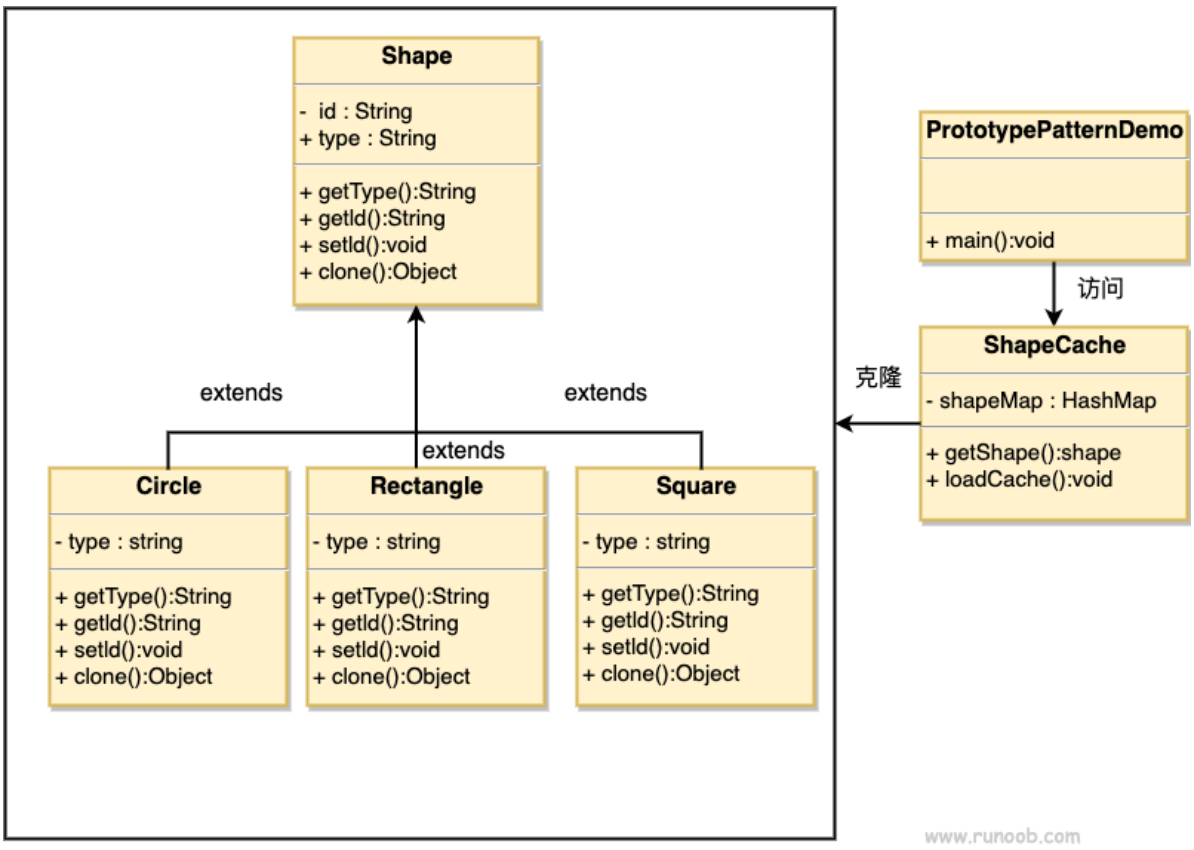
注意事项

与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝实现 Cloneable，重写，深拷贝是通过实现 Serializable 读取二进制流。

实现

我们将创建一个抽象类 *Shape* 和扩展了 *Shape* 类的实体类。下一步是定义类 *ShapeCache*，该类把 shape 对象存储在一个 *Hashtable* 中，并在请求的时候返回它们的克隆。

PrototypePatternDemo 类使用 *ShapeCache* 类来获取 *Shape* 对象。



步骤 1

创建一个实现了 *Cloneable* 接口的抽象类。

Shape.java

```
public abstract class Shape implements Cloneable {

    private String id;

    protected String type;

    abstract void draw();

    public String getType(){

        return type;

    }

    public String getId() {

        return id;

    }

    public void setId(String id) {

        this.id = id;

    }
```

```
    public Object clone() {

        Object clone = null;

        try {

            clone = super.clone();

        } catch (CloneNotSupportedException e) {

            e.printStackTrace();

        }

        return clone;

    }

 }
```

步骤 2

创建扩展了上面抽象类的实体类。

Rectangle.java

```
public class Rectangle extends Shape {

   public Rectangle(){

     type = "Rectangle";

   }

   @Override
   public void draw() {

    System.out.println("Inside Rectangle::draw() method.");

   }

 }
```

Square.java

```
public class Square extends Shape {

   public Square(){

        type = "Square";

   }

   @Override

   public void draw() {
```

```
      System.out.println("Inside Square::draw() method.");

   }

}
```

Circle.java

```
public class Circle extends Shape {

    public Circle(){

        type = "Circle";

    }

    @Override
    public void draw() {

     System.out.println("Inside Circle::draw() method.");

 }

}
```

步骤 3

创建一个类，从数据库获取实体类，并把它们存储在一个 *Hashtable* 中。

ShapeCache.java

```
import java.util.Hashtable;

public class ShapeCache {

      private static Hashtable<String, Shape> shapeMap = new Hashtable<String,
Shape>();

      public static Shape getShape(String shapeId) {

        Shape cachedShape = shapeMap.get(shapeId);

        return (Shape) cachedShape.clone();

      }

    // 对每种形状都运行数据库查询，并创建该形状

    // shapeMap.put(shapeKey, shape);

    // 例如，我们要添加三种形状

    public static void loadCache() {

       Circle circle = new Circle();
```

```
        circle.setId("1");
        shapeMap.put(circle.getId(),circle);
        Square square = new Square();
        square.setId("2");
        shapeMap.put(square.getId(),square);
        Rectangle rectangle = new Rectangle();
        rectangle.setId("3");
        shapeMap.put(rectangle.getId(),rectangle);
    }
}
```

步骤 4

PrototypePatternDemo 使用 *ShapeCache* 类来获取存储在 *Hashtable* 中的形状的克隆。

PrototypePatternDemo.java

```
public class PrototypePatternDemo {

  public static void main(String[] args) {
      ShapeCache.loadCache();
      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());
      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());
      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

步骤 5

执行程序，输出结果：

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

扩展：

clone()

概述

创建并返回此对象的副本，避免对原型对象的污染操作。

要求

1. 实现 Cloneable 接口
2. 重写 clone 方法

特点

1. clone得到的对象虽然拥有和本体相同的属性和方法，但是地址和原型不同
2. clone不会调用构造方法

案例代码

实体类

```
public class Student implements Cloneable {
    private int age;

    public void eat() {
        System.out.println("吃饭");
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    // Constructor、Getters、Setters
}
```

测试类

```
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Student student = new Student();

        student.setAge(18);

        Student cloneStudent = (Student) student.clone();

        cloneStudent.eat(); // 吃饭

        System.out.println(cloneStudent.getAge());  // 18

        // 地址不同
        System.out.println(cloneStudent == student); // false
    }
}
```

浅拷贝和深拷贝

浅拷贝概述

浅拷贝是按**位**拷贝对象，它会创建一个**新对象**，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是内存地址（引用类型），拷贝的就是内存地址，因此如果其中一个对象改变了这个地址，就会影响到另一个对象。即默认拷贝构造函数只是对对象进行浅拷贝复制(逐个成员依次拷贝)，即只复制对象空间而不复制对象资源。

浅拷贝特点

1. 对于基本数据类型的成员对象，因为**基础数据类型是值传递**的，所以是直接将属性值赋值给新的对象。基础类型的拷贝，其中一个对象修改该值，不会影响另外一个。
2. 对于引用类型，比如数组或者类对象，因为**引用类型**是**引用传递**，所以浅拷贝只是把**内存地址赋值**给了成员变量，它们指向了同一内存空间。改变其中一个，会对另外一个也产生影响。

浅拷贝案例代码

实体类

```
public class Car {
    private String brand;
    private String color;

    // Constructor、Getters、Setters、toString
}

public class Student implements Cloneable {
    private int age;
    private Car car;

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    // Constructor、Getters、Setters、toString
}
```

测试类

```
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Student student = new Student();

        student.setAge(18);

        Student cloneStudent = (Student) student.clone();

        student.setAge(20);
        car.setColor("红色");

        System.out.println(cloneStudent.getAge());  // 20
        System.out.println(student.getAge());   // 18
        System.out.println(cloneStudent.getCar());  // 红色
        System.out.println(student.getCar());   // 红色
```

```
    }
}
```

深拷贝概述

深拷贝，在拷贝引用类型成员变量时，为引用类型的数据成员另辟了一个独立的内存空间，实现真正内容上的拷贝。从而解决了上面浅拷贝修改汽车颜色会影响到本体的情况。

深拷贝特点

1. 对于基本数据类型的成员对象，因为基础数据类型是值传递的，所以是直接将属性值赋值给新的对象。基础类型的拷贝，其中一个对象修改该值，不会影响另外一个（和浅拷贝一样）。
2. 对于引用类型，比如数组或者类对象，深拷贝会新建一个对象空间，然后拷贝里面的内容，所以它们指向了不同的内存空间。改变其中一个，不会对另外一个也产生影响。
3. 对于有多层对象的，每个对象都需要实现 Cloneable 并重写 clone() 方法，进而实现了对象的**串行层层拷贝**。
4. 深拷贝相比于浅拷贝**速度较慢**并且**花销较大**。

深拷贝案例代码

实体类

```
public class Car implements Cloneable {
    private String brand;
    private String color;

    // Constructor、Getters、Setters、toString

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Student implements Cloneable {
    private int age;
    private Car car;

    // Constructor、Getters、Setters、toString

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Student student = (Student) super.clone();

        student.car = (Car) car.clone();

        return student;
    }
}
```

测试类

```
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Student student = new Student();
```

```
        student.setAge(18);

        Student cloneStudent = (Student) student.clone();

        student.setAge(20);
        car.setColor("红色");

        System.out.println(cloneStudent.getAge());  // 20
        System.out.println(student.getAge());   // 18
        System.out.println(cloneStudent.getCar());  // 红色
        System.out.println(student.getCar());   // 黑色
    }
}
```