课程目标

1、掌握原型模式的应用场景及常用写法。

内容定位

1、听说过原型模式，但不知道应用场景的人群。

原型模式

原型模式的应用场景

你一定遇到过大篇幅 getter、setter 赋值的场景。例如这样的代码：

```
public void setParam(ExamPaperVo vo){
        ExamPaper examPaper = new ExamPaper();
        //试卷主键
        examPaper.setExaminationPaperId(vo.getExaminationPaperId());
        //剩余时间
        curForm.setLeavTime(examPaper.getLeavTime());
        //单位主键
        curForm.setOrganizationId(examPaper.getOrganizationId());
        //考试主键
        curForm.setId(examPaper.getId());
        //考场主键
        curForm.setExamroomId(examPaper.getExamroomId());
        //用户主键
        curForm.setUserId(examPaper.getUserId());
        //专业
        curForm.setSpecialtyCode(examPaper.getSpecialtyCode());
        //岗位
        curForm.setPostionCode(examPaper.getPostionCode());
        //等级
        curForm.setGradeCode(examPaper.getGradeCode());
        //考试开始时间
        curForm.setExamStartTime(examPaper.getExamStartTime());
        //考试结束时间
```

```
        curForm.setExamEndTime(examPaper.getExamEndTime());
        //单选题重要数量
        curForm.setSingleSelectionImpCount(examPaper.getSingleSelectionImpCount());
        //多选题重要数量
        curForm.setMultiSelectionImpCount(examPaper.getMultiSelectionImpCount());
        //判断题重要数量
        curForm.setJudgementImpCount(examPaper.getJudgementImpCount());
        //考试时间
        curForm.setExamTime(examPaper.getExamTime());
        //总分
        curForm.setFullScore(examPaper.getFullScore());
        //及格分
        curForm.setPassScore(examPaper.getPassScore());
        //学员姓名
        curForm.setUserName(examPaper.getUserName());
        //分数
        curForm.setScore(examPaper.getScore());
        //是否及格
        curForm.setResult(examPaper.getResult());
        curForm.setIsPassed(examPaper.getIsPassed());
        //单选答对数量
        curForm.setSingleOkCount(examPaper.getSingleOkCount());
        //多选答对数量
        curForm.setMultiOkCount(examPaper.getMultiOkCount());
        //判断答对数量
        curForm.setJudgementOkCount(examPaper.getJudgementOkCount());

        //提交试卷
        service.submit(examPaper);
}
```

代码非常工整，命名非常规范，注释也写的很全面，大家觉得这样的代码优雅吗？我认为，这样的代码属于纯体力劳动。那么原型模式，能帮助我们解决这样的问题。

原型模式（Prototype Pattern）是指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。

原型模式主要适用于以下场景：

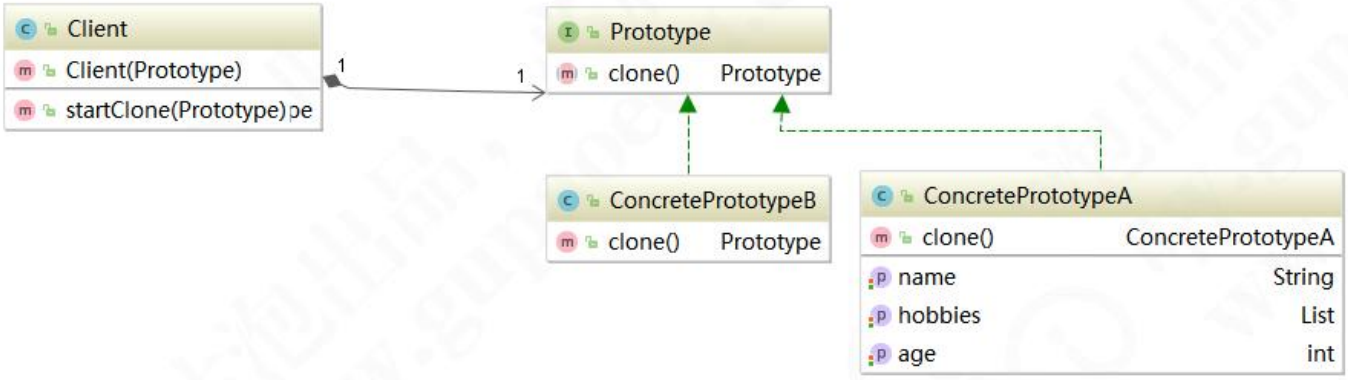
1、类初始化消耗资源较多。

2、new 产生的一个对象需要非常繁琐的过程（数据准备、访问权限等）

3、构造函数比较复杂。

4、循环体中生产大量对象时。

在 Spring 中，原型模式应用得非常广泛。例如 scope=“prototype”，在我们经常用的 JSON.parseObject()也是一种原型模式。下面，我们来看看原型模式类结构图：



简单克隆

一个标准的原型模式代码，应该是这样设计的。先创建原型 Prototype 接口：

```
public interface Prototype{
    Prototype clone();
}
```

创建具体需要克隆的对象 ConcretePrototype

```
import java.util.List;

public class ConcretePrototypeA implements Prototype {
    private int age;
    private String name;
    private List hobbies;

    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }
    public List getHobbies() {
        return hobbies;
    }
    public void setHobbies(List hobbies) {
        this.hobbies = hobbies;
    }

    @Override
    public ConcretePrototypeA clone() {
        ConcretePrototypeA concretePrototype = new ConcretePrototypeA();
        concretePrototype.setAge(this.age);
        concretePrototype.setName(this.name);
        concretePrototype.setHobbies(this.hobbies);
        return concretePrototype;
    }
}
```

创建 Client 对象

```
public class Client {
    private Prototype prototype;
    public Client(Prototype prototype){
        this.prototype = prototype;
    }
    public Prototype startClone(Prototype concretePrototype){
        return (Prototype)concretePrototype.clone();
    }
}
```

测试代码：

```
import java.util.ArrayList;
import java.util.List;
public class PrototypeTest {
    public static void main(String[] args) {
        // 创建一个具体的需要克隆的对象
        ConcretePrototypeA concretePrototype = new ConcretePrototypeA();
        // 填充属性，方便测试
        concretePrototype.setAge(18);
        concretePrototype.setName("prototype");
        List hobbies = new ArrayList<String>();
        concretePrototype.setHobbies(hobbies);
        System.out.println(concretePrototype);
```

```
        // 创建 Client 对象，准备开始克隆
        Client client = new Client(concretePrototype);
        ConcretePrototypeA concretePrototypeClone = (ConcretePrototypeA)
client.startClone(concretePrototype);
        System.out.println(concretePrototypeClone);

        System.out.println("克隆对象中的引用类型地址值：" + concretePrototypeClone.getHobbies());
        System.out.println("原对象中的引用类型地址值：" + concretePrototype.getHobbies());
        System.out.println("对象地址比较："+(concretePrototypeClone.getHobbies() ==
concretePrototype.getHobbies()));
    }
}
```

运行结果：

```
com.gupaoedu.vip.pattern.prototype.simple.ConcretePrototypeA@1540e19d
com.gupaoedu.vip.pattern.prototype.simple.ConcretePrototypeA@677327b6
克隆对象中的引用类型地址值：[]
原对象中的引用类型地址值：[]
对象地址比较：true

Process finished with exit code 0
```

从测试结果看出 hobbies 的引用地址是相同的，意味着复制的不是值，而是引用的地址。这样的话，如果我们修改任意一个对象中的属性值，concretePrototype 和 concretePrototypeCone 的 hobbies 值都会改变。这就是我们常说的浅克隆。只是完整复制了值类型数据，没有赋值引用对象。换言之，所有的引用对象仍然指向原来的对象，显然不是我们想要的结果。下面我们来看深度克隆继续改造。

深度克隆

我们换一个场景，大家都知道齐天大圣。首先它是一只猴子，有七十二般变化，把一根毫毛就可以吹出千万个泼猴，手里还拿着金箍棒，金箍棒可以变大变小。这就是我们耳熟能详的原型模式的经典体现。

创建原型猴子 Monkey 类：

```
import java.util.Date;

/**
 * Created by Tom.
 */
public class Monkey {
    public int height;
    public int weight;
    public Date birthday;

}
```

创建引用对象金箍棒 Jingubang 类：

```
import java.io.Serializable;

/**
 * Created by Tom.
 */
public class JinGuBang implements Serializable {
    public float h = 100;
    public float d = 10;
    public void big(){
        this.d *= 2;
        this.h *= 2;
    }
    public void small(){
        this.d /= 2;
        this.h /= 2;
    }
}
```

创建具体的对象齐天大圣 QiTianDaSheng 类：

```
import java.io.*;
import java.util.Date;

public class QiTianDaSheng extends Monkey implements Cloneable,Serializable {

    public JinGuBang jinGuBang;

    public  QiTianDaSheng(){
        //只是初始化
        this.birthday = new Date();
        this.jinGuBang = new JinGuBang();
    }
```

```
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return this.deepClone();
    }

    public Object deepClone(){
        try{

            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(bos);
            oos.writeObject(this);

            ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
            ObjectInputStream ois = new ObjectInputStream(bis);

            QiTianDaSheng copy = (QiTianDaSheng)ois.readObject();
            copy.birthday = new Date();
            return copy;

        }catch (Exception e){
            e.printStackTrace();
            return null;
        }

    }

    public QiTianDaSheng shallowClone(QiTianDaSheng target){

        QiTianDaSheng qiTianDaSheng = new QiTianDaSheng();
        qiTianDaSheng.height = target.height;
        qiTianDaSheng.weight = target.height;

        qiTianDaSheng.jinGuBang = target.jinGuBang;
        qiTianDaSheng.birthday = new Date();

        return  qiTianDaSheng;
    }
}
```

测试代码：

```
public class DeepCloneTest {

    public static void main(String[] args) {
```

```
        QiTianDaSheng qiTianDaSheng = new QiTianDaSheng();
        try {
            QiTianDaSheng clone = (QiTianDaSheng)qiTianDaSheng.clone();
            System.out.println("深克隆: " + (qiTianDaSheng.jinGuBang == clone.jinGuBang));
        } catch (Exception e) {
            e.printStackTrace();
        }

        QiTianDaSheng q = new QiTianDaSheng();
        QiTianDaSheng n = q.shallowClone(q);
        System.out.println("浅克隆: " + (q.jinGuBang == n.jinGuBang));
    }
}
```

运行结果：

```
深克隆: false
浅克隆: true


Process finished with exit code 0
```

克隆破坏单例模式

如果我们克隆的目标的对象是单例对象，那意味着，深克隆就会破坏单例。实际上防止克隆破坏单例解决思路非常简单，禁止深克隆便可。要么你我们的单例类不实现 Cloneable 接口；要么我们重写 clone()方法，在 clone 方法中返回单例对象即可，具体代码如下：

```
@Override
protected Object clone() throws CloneNotSupportedException {
    return INSTANCE;
}
```

Cloneable 源码分析

先看我们常用的 ArrayList 就实现了 Cloneable 接口，来看代码 clone()方法的实现：

```
public Object clone() {
    try {
        ArrayList<?> v = (ArrayList<?>) super.clone();
        v.elementData = Arrays.copyOf(elementData, size);
```

```
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
    }
}
```