使程序运行更高效——原型模式

1.原型模式介绍

原型模式是一个创建型的模式。原型二字表明了该模式应该有一个样板实例，用户从这个样板对象中复制出一个内部属性一致的对象，这个过程也就是我们俗称的“克隆”。被复制的实例就是我们所称的“原型”，这个原型是可定制的。**原型模式多用于创建复杂的或者耗时的实例，因为这种情况下，复制一个已经存在的实例可使程序运行更高效。**

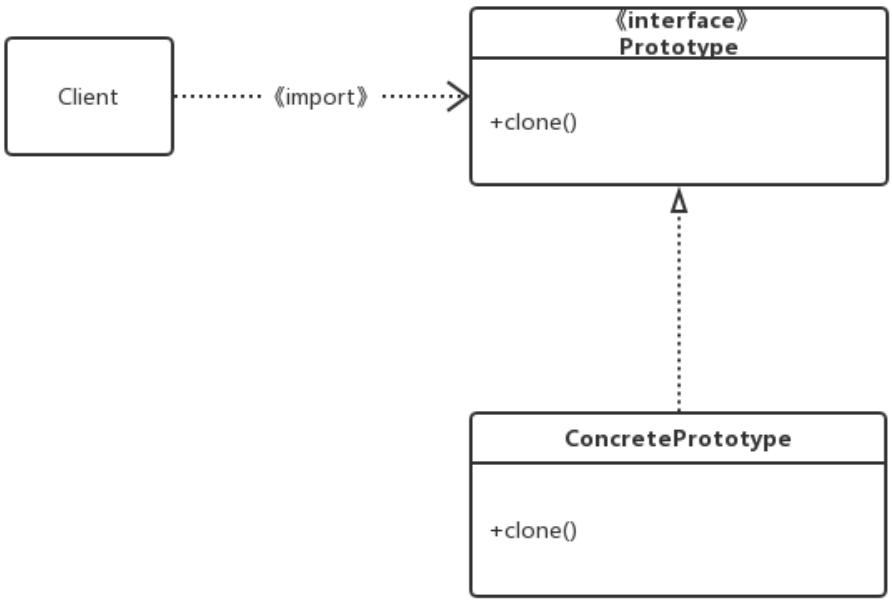
2,原型模式的定义

用原型实例制定创建对象的种类，并通过拷贝这些原型创建新的对象。

3.原型模式的使用场景

1. 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，通过原型拷贝避免这些消耗。
2. 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，这时可以使用原型模式。
3. 一个对象需要提供给其它对象访问，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，即保护性拷贝。
   需要注意的是，通过实现Cloneable接口的原型模式在调用clone函数构造实例时并不一定比通过new操作快，只有当通过new构造对象较为耗时或者说成本较高时，通过clone方法才能够获得效率上的提升。因此，在使用Cloneable时需要考虑构建对象的成本以及做一些效率上的测试。

4,原型模式的UML类图



角色介绍：

- Client: 客户端用户；
- Prototype:抽象类或者接口，声明具备clone能力；
- ConcretePrototype：具体的原型类。

> 5.原型模式的简单实现

下面以简单文档拷贝为例来演示一下简单的原型模式，我们在这个例子中首先创建了一个文档对象，即WordDocument，这个文档中含有文字和图片。用户经过了长时间的内容编辑后，打算对该文档做进一步编辑，但是，这个编辑后的文档是否会被采用还不确定，因此，为了安全起见，用户需要将当前文档拷贝一份，然后再在文档副本上进行修改，这样，这个原始文档就是我们上述所说的样板实例，也就是将要被“克隆”的对象，我们称为原型：

```
/**
*文档类型，扮演的是ConcretePrototype角色，而cloneable是代表*prototype角色
*/
public class WordDocument implements Cloneable {
        // 文本
        private String mText;
        // 图片名称列表
        private ArrayList<String> mImages = new ArrayList<String>();

        public WordDocument() {
                System.out.println("----------WordDocument构造函数-----------");
        }


```

```
        @Override
        protected WordDocument clone() {
                try {
                        WordDocument doc = (WordDocument) super.clone();
                        doc.mText = this.mText;
                        doc.mImages = this.mImages;
                        return doc;
                } catch (Exception e) {
                }
                return null;
        }

        public String getText() {
                return mText;
        }
        public void setText(String mText){
                this.mText=mText;
        }
        public List<String> getImages(){
                return mImages;
        }
        public void addImage(String img){
                this.mImages.add(img);
        }
        /**
         * 打印文档内容
         */
        public void showDocument(){
                System.out.println("----------Word Content Start----------");
                System.out.println("Text:"+mText);
                System.out.println("Image List:");
                for (String imgName:mImages) {
                        System.out.println("image name:"+imgName);
                }
                System.out.println("----------Word Content End------------");
        }
}
```

通过WordDocument类模拟Word文档中的基本元素，即文字和图片。WordDocument在该原型模式示例中扮演的角色为ConcretePrototype,而Cloneable的角色则为Prototype。WordDocument中的clone方法用以实现对象克隆。注意，这个方法并不是Cloneable接口中的，而是Object中的方法。Cloneable也是一个标识接口，它表示这个类的对象是拷贝的。如果没有实现Cloneable接口却调用了clone()函数将抛出异常。在这个示例中，我们通过实现Cloneable接口和覆写clone方法实现原型模式。

Client类：

```
public class Client {
        public static void main(String[] args) {
                // 1.构建文档对象
                WordDocument originDoc = new WordDocument();
                // 2.编辑文档，添加图片等
                originDoc.setText("这是一篇文档");
                originDoc.addImage("图片1");
                originDoc.addImage("图片2");
```

```
                originDoc.addImage("图片3");
                originDoc.showDocument();

                // 以原始文档为原型，拷贝一份副本
                WordDocument doc2 = originDoc.clone();
                doc2.showDocument();

                // 修改文档副本，不会影响原始文档
                doc2.setText("这是修改过的Doc2文本");
                doc2.showDocument();

                originDoc.showDocument();
        }
}

```

输出：

```
----------WordDocument构造函数-----------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
----------Word Content End------------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
----------Word Content End------------
----------Word Content Start----------
Text:这是修改过的Doc2文本
Image List:
image name:图片1
image name:图片2
image name:图片3
----------Word Content End------------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
----------Word Content End------------

```

从输出可看出，doc2是通过originDoc.clone()创建的，并且doc2第一次输出的时候和originDoc输出是一样，即doc2是originDoc的一份拷贝，它们的内容是一样的，而doc2修改了文本内容以后并不影响originDoc的文本内容，这就保证了originDoc的安全性。还需要注意的是，通过clone拷贝对象时并不会执行构造函数。因此，如果在构造函数中需要一些特殊的初始化操作的类型，在使用Cloneable实现拷贝时，需要注意构造函数不会执行的问题。

6.浅拷贝和深拷贝

上述原型模式的实现实际上只是一个浅拷贝，也称影子拷贝，这份拷贝实际上并不是将原始文档的所有字段都重新构造一份，而是 副本文档的字段引用原始文档的字段，如图：

我们知道A引用B就是说两个对象指向同一个地址，当修改A时B也会改变，B修改时A同样会改变。我们直接看下面的例子，将main函数的内容修改为如下：

```
public class Client {
        public static void main(String[] args) {
                // 1.构建文档对象
                WordDocument originDoc = new WordDocument();
                // 2.编辑文档，添加图片等
                originDoc.setText("这是一篇文档");
                originDoc.addImage("图片1");
                originDoc.addImage("图片2");
                originDoc.addImage("图片3");
                originDoc.showDocument();

                // 以原始文档为原型，拷贝一份副本
                WordDocument doc2 = originDoc.clone();
                doc2.showDocument();

                // 修改文档副本，不会影响原始文档
                // doc2.setText("这是修改过的Doc2文本");
                // doc2.showDocument();

                doc2.addImage("哈哈哈.jpg");
                doc2.showDocument();

                originDoc.showDocument();
        }
}

```

输出结果：

```
----------WordDocument构造函数-----------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
----------Word Content End------------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
```

```
----------Word Content End------------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
image name:哈哈哈.jpg
----------Word Content End------------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
image name:哈哈哈.jpg
----------Word Content End------------
```

最后两份文档信息输出是一致的。我们在doc2添加了一张名为“哈哈哈.jpg”的图片，但是，同时也显示在originDoc中了，这是因为上文中WordDocument的clone方法中只是简单地进行浅拷贝，引用类型的新对象doc2的mImages只是单纯地指向了this.mIages引用，并没重新构造一个mImages对象，然后将原文档中的图片添加到新的mImages对象中，这样就导致doc2中的mImages与原始文档中的是同一个对象，因此，修改了其中一个文档中的图片，另一个文档也会受影响。doc2的mImages添加了新的图片，实际上也就是往originDoc里添加了新图片，所以，originDoc里面也有“哈哈哈.jpg”图片文件。那如何解决这个问题呢？答案就是采用深拷贝，即 **在拷贝对象时，对于引用型的字段也要采用拷贝的形式，而不是单纯的形式，而不是单纯引用的形式。** clone方法修改如下：

```
@Override
        protected WordDocument clone() {
                try {
                        WordDocument doc = (WordDocument) super.clone();
                        doc.mText = this.mText;
                        // 浅拷贝
                        // doc.mImages = this.mImages;
                        // 深拷贝
                        doc.mImages = (ArrayList<String>) this.mImages.clone();
                        return doc;
                } catch (Exception e) {
                }
                return null;
        }
```

如上述代码所示，将doc.mImages指向this.mImages的一份拷贝，而不是this.mImages本身， 这样在doc2添加图片时并不会影响originDoc，运行效果如下：

```
----------WordDocument构造函数-----------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
----------Word Content End------------
```

```
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
----------Word Content End------------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
image name:哈哈哈.jpg
----------Word Content End------------
----------Word Content Start----------
Text:这是一篇文档
Image List:
image name:图片1
image name:图片2
image name:图片3
----------Word Content End------------
```

原型模式是非常简单的模式，它的核心问题就是对原始对象进行拷贝，在这个模式的使用过程中需要注意的一点就是：深、浅拷贝的问题。为了减少出错，建议大家在使用该模式时使用深拷贝，避免操作副本时影响原始对象的问题。

5.实例

在开发中，我们有时候会满足一些需求，就是有的对象中的内容只允许客户端程序读取，而不允许修改。在一个客户端中，在用户登录之后，小明会通过一个LoginSession保存用户的登录信息，这些用户信息可能在App的其它模块被用来做登录校验、用户个人信息显示等。但是，这些信息在客户端程序是不允许修改的，而需要在其它模块被调用，因此，需要开放已登录用户信息的访问接口。我们看看小明的代码：

```
/**
 * 用户实体类
 */
public class User {
        public int age;
        public String name;
        public Addresss address;

        @Override
        public String toString() {
                return "User [age=]" + age + ",name=" + name + ",adress=" + address
                                + "]";
        }
}

//用户地址类，存储地址的详细信息
public class Addresss {
```

```
        // 城市
        public String city;
        // 区
        public String district;
        public String street;

        public Addresss(String aCity, String aDist, String aStreet) {
                this.city = aCity;
                this.district = aDist;
                this.street = aStreet;
        }

        @Override
        public String toString() {
                return "Adress [city=" + city + ",district=" + district + ",street="
                                + street + "]";
        }
}
```

登录接口：

```
 //登录接口
public interface Login {
        void login();
}
//登录实现
public class LoginImpl implements Login {

        public void login() {
                // 登录到服务器，获取用户信息
                User loginedUser = new User();
                // 将服务器返回的完整信息设置给loginedUser对象
                loginedUser.age = 22;
                loginedUser.name = "XiaoMing";
                loginedUser.address = new Addresss("北京市", "海淀区", "花园路");
                // 登录完之后将用户信息设置到Sessin中LoginSession.setLoginSession()里
                LoginSession.getLoginSession().setLoginedUser(loginedUser);
        }

}

//登录Session
public class LoginSession {
        static LoginSession sLoginSession = null;
        // 已登录用户
        private User loginUser;

        private LoginSession() {

        }

        public static LoginSession getLoginSession() {
                if (sLoginSession == null) {

                        
```

```
                        sLoginSession = new LoginSession();
                }
                return sLoginSession;
        }

        // 设置已登录的用户信息
        protected void setLoginedUser(User user) {
                loginUser = user;
        }

        public User getLoginedUser() {
                return loginUser;
        }
}
```

上述代码比较简单，就是在用户登录之后通过LoginSession的setLoginedUser函数将登录用的信息设置到Session中，这个setLoginedUser是包级私有的，因此，外部模块无法调用，这在一定程度上满足了小明的需求，也就是外部客户端程序不能修改已登录的用户信息。
不巧的是，小明的开发搭档大明也是一位经验不太丰富的工程师，他在用户个人修改页面写出了类似这样的代码：

```
//获取已登录的User对象
User newUser=LoginSession.getLoginSession().getLoginedUser();
//更新用户
newUser.address=new Address("北京市","朝阳区","大望路");
```

在用户点击更新按钮时，直接调用了类似上述的代码来更新用户地址，而不是网络请求成功 后才调用相关的个人信息更新函数，而且这个修改并不是在LoginSession包中，因为客户端代码只能通过`setLoginedUser()`来更新用户信息，这就奇怪了，小明在更新用户信息的代码下添加了这两行Log输出代码：

```
Log.d("tag","temp user:"+tempUser);
Log.d("tag","已登录用户: "+LoginSession.getLoginSession().getLoginedUser());
```

从Log中可以发现了问题：

```
temp User:User [age=22,name="XiaoMing",address=Address[city=北京市,district=朝阳区,street=大望路]]
已登录用户:User [age=22,name="XiaoMing",address=Address[city=北京市,district=朝阳区,street=大望路]]
```

也就是上述说的，网络请求为成功的情况下修改了用户的address字段！小明感觉自己设置的用户信息更新只限于与LoginSession类在同一个包下的限制瞬间被突破了。这样一来，不管客户端代码无意间写错了代码导致用户信息被修改，还是对于代码理解有误导致的问题，最终结果都是用户信息被修改了，小明找大佬咨询，
“这类问题你们可以使用原型模式来进行保护性拷贝，也就是说在LoginSession的getLoginUser()函数中返回的是已登录用户的一个拷贝，当更新用户地址的网络请求完成时，再通过包级私有的LoginSession中的setLoginedUser更新用户信息，当然，这个网络请求所在的包此时应该与LoginSession一致”，小明和大明这才明白过来，于是在User类中覆写了clone方法：

```
public class User {
        public int age;
        public String name;
```

```
        public Addresss address;

        @Override
        public String toString() {
                return "User [age=]" + age + ",name=" + name + ",adress=" + address
                                + "]";
        }
        @Override
        public User clone() {
                User user = null;
                try {
                        user = (User) super.clone();
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return user;
        }
}
```

并且在LoginSession中将getLoginedUser函数修改如下：

```
public User getLoginedUser() {
                return loginUser.clone();
        }

```

这就使得在任何地方调用getLoginedUser函数获取到的用户对象都是一个拷贝对象，即使客户端代码不小心修改了这个拷贝对象，也不会影响最初的已登录用户对象，对已登录用户信息的修改只能通过setLoginedUser这个方法，而只有与LoginSession在一个包下的类才能访问这个包级私有方法，因此，确保了它的安全性。

6.总结

原型模式本质上就是对象拷贝，与C++中的拷贝构造函数有些类似，它们之间容易出现的问题也都是深拷贝、浅拷贝。使用原型模式可以解决构建复杂对象的资源消耗问题，能够在某些场景下提升创建对象的效率。还有一个重要的用途就是保护性拷贝，也就是某个对象对外可能是只读的，为了防止外部对这个只读对象对象修改，通常可以通过返回一个对象拷贝的形式实现只读的限制。
优点与缺点
优点
原型模式在内存中二进制流的拷贝，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，原型模式可以更好地体现其优点。
缺点
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，在实际开发中当中应注意这个潜在的问题。优点就是减少了约束，缺点也是减少了约束，需要大家在实际应用考虑。
————摘自《Android 源码设计模式解析与实战 第四章》

xyzso1z
原创文章 80　获赞 40　访问量 2万+