27 | 原型模式与享元模式：提升系统性能的利器

2019-07-25 刘超

你好，我是刘超。

原型模式和享元模式，前者是在创建多个实例时，对创建过程的性能进行调优；后者是用减少创建实例的方式，来调优系统性能。这么看，你会不会觉得两个模式有点相互矛盾呢？

其实不然，它们的使用是分场景的。在有些场景下，我们需要重复创建多个实例，例如在循环体中赋值一个对象，此时我们就可以采用原型模式来优化对象的创建过程；而在有些场景下，我们则可以避免重复创建多个实例，在内存中共享对象就好了。

今天我们就来看看这两种模式的适用场景，了解了这些你就可以更高效地使用它们提升系统性能了。

原型模式

我们先来了解下原型模式的实现。原型模式是通过给出一个原型对象来指明所创建的对象的类型，然后使用自身实现的克隆接口来复制这个原型对象，该模式就是用这种方式来创建出更多同类型的对象。

使用这种方式创建新的对象的话，就无需再通过new实例化来创建对象了。这是因为Object类的clone方法是一个本地方法，它可以直接操作内存中的二进制流，所以性能相对new实例化来说，更佳。

实现原型模式

我们现在通过一个简单的例子来实现一个原型模式：

```
//实现Cloneable 接口的原型抽象类Prototype
class Prototype implements Cloneable {
    //重写clone方法
    public Prototype clone(){
        Prototype prototype = null;
        try{
            prototype = (Prototype)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return prototype;
    }
}
//实现原型类
class ConcretePrototype extends Prototype{
    public void show(){
        System.out.println("原型模式实现类");
    }
}

public class Client {
    public static void main(String[] args){
        ConcretePrototype cp = new ConcretePrototype();
        for(int i=0; i< 10; i++){
            ConcretePrototype clonecp = (ConcretePrototype)cp.clone();
            clonecp.show();
        }
    }
}
```

要实现一个原型类，需要具备三个条件：

- 实现Cloneable接口：Cloneable接口与序列化接口的作用类似，它只是告诉虚拟机可以安全地在实现了这个接口的类上使用clone方法。在JVM中，只有实现了Cloneable接口的类才可以

被拷贝，否则会抛出CloneNotSupportedException异常。

- 重写Object类中的clone方法：在Java中，所有类的父类都是Object类，而Object类中有一个clone方法，作用是返回对象的一个拷贝。
- 在重写的clone方法中调用super.clone()：默认情况下，类不具备复制对象的能力，需要调用super.clone()来实现。

从上面我们可以看出，原型模式的主要特征就是使用clone方法复制一个对象。通常，有些人会误以为 Object a=new Object();Object b=a; 这种形式就是一种对象复制的过程，然而这种复制只是对象引用的复制，也就是a和b对象指向了同一个内存地址，如果b修改了，a的值也就跟着被修改了。

我们可以通过一个简单的例子来看看普通的对象复制问题：

```
class Student {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name= name;
    }

}
public class Test {

    public static void main(String args[]) {
        Student stu1 = new Student();
        stu1.setName("test1");

        Student stu2 = stu1;
        stu2.setName("test2");

        System.out.println("学生1:" + stu1.getName());
        System.out.println("学生2:" + stu2.getName());
    }
}
```

如果是复制对象，此时打印的日志应该为：

```
学生1:test1
学生2:test2
```

然而，实际上是：

```
学生2:test2
学生2:test2
```

通过clone方法复制的对象才是真正的对象复制，clone方法赋值的对象完全是一个独立的对象。刚刚讲过了，Object类的clone方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。我们可以用 clone 方法再实现一遍以上例子。

```
//学生类实现Cloneable接口
class Student implements Cloneable{
    private String name;  //姓名

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name= name;
    }
   //重写clone方法
   public Student clone() {
        Student student = null;
        try {
            student = (Student) super.clone();
            } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            }
            return student;
   }

}
public class Test {

    public static void main(String args[]) {
        Student stu1 = new Student();  //创建学生1
        stu1.setName("test1");

        Student stu2 = stu1.clone();  //通过克隆创建学生2
        stu2.setName("test2");

        System.out.println("学生1:" + stu1.getName());
        System.out.println("学生2:" + stu2.getName());
    }
}
```

运行结果：

```
学生1:test1
学生2:test2
```

深拷贝和浅拷贝

在调用super.clone()方法之后，首先会检查当前对象所属的类是否支持clone，也就是看该类是否实现了Cloneable接口。

如果支持，则创建当前对象所属类的一个新对象，并对该对象进行初始化，使得新对象的成员变量的值与当前对象的成员变量的值一模一样，但对于其它对象的引用以及List等类型的成员属性，则只能复制这些对象的引用了。所以简单调用super.clone()这种克隆对象方式，就是一种浅拷贝。

所以，当我们在使用clone()方法实现对象的克隆时，就需要注意浅拷贝带来的问题。我们再通过一个例子来看看浅拷贝。

```
//定义学生类
class Student implements Cloneable{
    private String name; //学生姓名
    private Teacher teacher; //定义老师类

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Teacher getTeacher() {
        return teacher;
    }

    public void setName(Teacher teacher) {
        this.teacher = teacher;
    }
  //重写克隆方法
```

```
    //重写克隆方法
    public Student clone() {
        Student student = null;
        try {
            student = (Student) super.clone();
            } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            }
            return student;
    }

}

//定义老师类
class Teacher implements Cloneable{
    private String name;  //老师姓名

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name= name;
    }

    //重写克隆方法，堆老师类进行克隆
    public Teacher clone() {
        Teacher teacher= null;
        try {
            teacher= (Teacher) super.clone();
            } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            }
            return student;
    }

}
public class Test {
```

```
    public static void main(String args[]) {
        Teacher teacher = new Teacher (); //定义老师1
        teacher.setName("刘老师");
        Student stu1 = new Student();  //定义学生1
        stu1.setName("test1");
        stu1.setTeacher(teacher);

        Student stu2 = stu1.clone(); //定义学生2
        stu2.setName("test2");
        stu2.getTeacher().setName("王老师");//修改老师
        System.out.println("学生" + stu1.getName + "的老师是:" + stu1.getTeacher().getName);
        System.out.println("学生" + stu1.getName + "的老师是:" + stu2.getTeacher().getName);
    }
}
```

运行结果：

```
学生test1的老师是：王老师
学生test2的老师是：王老师
```

观察以上运行结果，我们可以发现：在我们给学生2修改老师的时候，学生1的老师也跟着被修改了。这就是浅拷贝带来的问题。

我们可以通过深拷贝来解决这种问题，其实深拷贝就是基于浅拷贝来递归实现具体的每个对象，代码如下：

```
public Student clone() {
    Student student = null;
    try {
        student = (Student) super.clone();
        Teacher teacher = this.teacher.clone();//克隆teacher对象
        student.setTeacher(teacher);
        } catch (CloneNotSupportedException e) {
        e.printStackTrace();
        }
        return student;
}
```

适用场景

前面我详述了原型模式的实现原理，那到底什么时候我们要用它呢？

在一些重复创建对象的场景下，我们就可以使用原型模式来提高对象的创建性能。例如，我在开头提到的，循环体内创建对象时，我们就可以考虑用clone的方式来实现。

例如：

```
for(int i=0; i<list.size(); i++){
  Student stu = new Student();
  ...
}
```

我们可以优化为：

```
Student stu = new Student();
for(int i=0; i<list.size(); i++){
 Student stu1 = (Student)stu.clone();
  ...
}
```

除此之外，原型模式在开源框架中的应用也非常广泛。例如Spring中，@Service默认都是单例的。用了私有全局变量，若不想影响下次注入或每次上下文获取bean，就需要用到原型模式，

我们可以通过以下注解来实现，@Scope(“prototype”)。

享元模式

享元模式是运用共享技术有效地最大限度地复用细粒度对象的一种模式。该模式中，以对象的信息状态划分，可以分为内部数据和外部数据。内部数据是对象可以共享出来的信息，这些信息不会随着系统的运行而改变；外部数据则是在不同运行时被标记了不同的值。

享元模式一般可以分为三个角色，分别为 Flyweight（抽象享元类）、ConcreteFlyweight（具体享元类）和 FlyweightFactory（享元工厂类）。抽象享元类通常是一个接口或抽象类，向外界提供享元对象的内部数据或外部数据；具体享元类是指具体实现内部数据共享的类；享元工厂类则是主要用于创建和管理享元对象的工厂类。

实现享元模式

我们还是通过一个简单的例子来实现一个享元模式：

```
//抽象享元类
interface Flyweight {
    //对外状态对象
    void operation(String name);
    //对内对象
    String getType();
}
```

```
//具体享元类
class ConcreteFlyweight implements Flyweight {
    private String type;

    public ConcreteFlyweight(String type) {
        this.type = type;
    }

    @Override
    public void operation(String name) {
        System.out.printf("[类型(内在状态)] - [%s] - [名字(外在状态)] - [%s]\n", type, name);
    }

    @Override
    public String getType() {
        return type;
    }
}
```

```
//享元工厂类
class FlyweightFactory {
    private static final Map<String, Flyweight> FLYWEIGHT_MAP = new HashMap<>();//享元池，用来存储享元对象

    public static Flyweight getFlyweight(String type) {
        if (FLYWEIGHT_MAP.containsKey(type)) {//如果在享元池中存在对象，则直接获取
            return FLYWEIGHT_MAP.get(type);
        } else {//在响应池不存在，则新创建对象，并放入到享元池
            ConcreteFlyweight flyweight = new ConcreteFlyweight(type);
            FLYWEIGHT_MAP.put(type, flyweight);
            return flyweight;
        }
    }
}
```

```
public class Client {

    public static void main(String[] args) {
        Flyweight fw0 = FlyweightFactory.getFlyweight("a");
        Flyweight fw1 = FlyweightFactory.getFlyweight("b");
        Flyweight fw2 = FlyweightFactory.getFlyweight("a");
        Flyweight fw3 = FlyweightFactory.getFlyweight("b");
        fw1.operation("abc");
        System.out.printf("[结果(对象对比)] - [%s]\n", fw0 == fw2);
        System.out.printf("[结果(内在状态)] - [%s]\n", fw1.getType());
    }
}
```

输出结果：

```
[类型(内在状态)] - [b] - [名字(外在状态)] - [abc]
[结果(对象对比)] - [true]
[结果(内在状态)] - [b]
```

观察以上代码运行结果，我们可以发现：如果对象已经存在于享元池中，则不会再创建该对象了，而是共用享元池中内部数据一致的对象。这样就减少了对象的创建，同时也节省了同样内部数据的对象所占用的内存空间。

适用场景

享元模式在实际开发中的应用也非常广泛。例如Java的String字符串，在一些字符串常量中，会共享常量池中字符串对象，从而减少重复创建相同值对象，占用内存空间。代码如下：

```
String s1 = "hello";
String s2 = "hello";
System.out.println(s1==s2);//true
```

还有，在日常开发中的应用。例如，线程池就是享元模式的一种实现；将商品存储在应用服务的缓存中，那么每当用户获取商品信息时，则不需要每次都从redis缓存或者数据库中获取商品信息，并在内存中重复创建商品信息了。

总结

通过以上讲解，相信你对原型模式和享元模式已经有了更清楚的了解了。两种模式无论是在开源框架，还是在实际开发中，应用都十分广泛。

在不得已需要重复创建大量同一对象时，我们可以使用原型模式，通过clone方法复制对象，这种方式比用new和序列化创建对象的效率要高；在创建对象时，如果我们可以共用对象的内部数据，那么通过享元模式共享相同的内部数据的对象，就可以减少对象的创建，实现系统调优。

思考题

上一讲的单例模式和这一讲的享元模式都是为了避免重复创建对象，你知道这两者的区别在哪儿吗？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

精选留言

木木匠

7

单例模式是针对某个类的单例，享元模式可以针对一个类的不同表现形式的单例，享元模式是单例模式的超集。

2019-07-25

作者回复

言简意赅！

2019-07-27

知行合一

3

new一个对象和clone一个对象，性能差在哪里呢？文中提到直接从内存复制二进制这里不是很理解

2019-07-26

作者回复

一个对象通过new创建的过程为：

1、在内存中开辟一块空间；

2、在开辟的内存空间中创建对象；

3、调用对象的构造函数进行初始化对象。

而一个对象通过clone创建的过程为：

1、根据原对象内存大小开辟一块内存空间；

2、复制已有对象，克隆对象中所有属性值。

相对new来说，clone少了调用构造函数。如果构造函数中存在大量属性初始化或大对象，则使用clone的复制对象的方式性能会好一些。

2019-07-26

QQ怪

3

享元模式可以再次创建对象 也可以取缓存对象

单例模式则是严格控制单个进程中只有一个实例对象

享元模式可以通过自己实现对外部的单例 也可以在需要的使用创建更多的对象

单例模式是自身控制 需要增加不属于该对象本身的逻辑

2019-07-25

作者回复

理解很透彻，点赞

2019-07-26

公号-代码荣耀

1

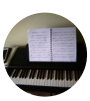
享元模式的实例也需要考虑线程安全哇？

2019-07-28

作者回复

需要的。共享数据尽量不要涉及到线程安全问题，否则就没有什么优势了。例如字符串则利用了不可变性来避免线程安全问题。

2019-07-29



东方奇骥

1

老师，请教一下，文中说的，@service默认是单例模式，若不想影响下次请求，就要使用原型

模式。能举个例子吗，什么时候会影响下次请求，不是很理解，因为我的项目里基本都是单例模式

2019-07-27

作者回复

这里纠正下，不是每次请求，而是每次bean注入或通过上下文获取bean时。

如果我们使用的是单例，假设有一个全局变量private int a=1，我们通过上下文获取到实例，调用A方法修改了变量a=2，此时下一个通过上下文获取到实例调用B方法获取变量，则a=2。

如果我们使用的是原型模式，假设有一个全局变量private int a=1，我们通过上下文获取到实例，调用A方法修改了变量a=2，此时下一个通过上下文获取到实例调用B方法获取变量，则还是a=1。

2019-07-29

Aaron 1

老师请教你个问题，线上短信业务被轰炸，流量费倍增.....求推荐个解决思路，监测发现是爬虫程序

2019-07-26

作者回复

建议加一个图片验证码

2019-07-27

业余草 0

变一下不就是工厂模式吗？

2019-08-01

Aaron 0

谢谢老师提供的思路

2019-07-27

Liam 0

老师好，文中举例Spring的prototype貌似不是原型模式的实现吧，每次spring都是通过反射创建的对象，并没有通过clone的方式吧

2019-07-26

门窗小二 0

通过老师这次的讲解算是彻底明白了单例模式与享元模式的区别，享元模式可以理解为一组单例模式

2019-07-25

作者回复

对的

2019-07-26

程序员人生

0

单例模式的运用场景一般是一个系统中的全局事物，比如数据库连接池，多线程连接池等
享元模式的运用场景则是是业务流程中，需要频繁获取元数据的的情况，比如老师说的用户获取商品的情况。

2019-07-25

Jxin

0

实现一个公共父类，实现原型模式，并反射完成深拷贝。对需要大量创建新对象的类继承该父类。老师这样做行不？反射有开销，继承这种结构也不好（但组合实现感觉不直观）。不确定这样抽象后是否利大于弊。毕竟如果反射开销冲掉了clone带来的性能优化，还不如直接new

2019-07-25

作者回复

想法很好，不建议这样曲线救国。我们没有必要针对每一个类去做原型设计，仅仅针对一些特殊场景的类实现clone方法即可。

2019-07-30

我已经设置了昵称

0

享元模式和策略模式感觉有点像啊，根据某个值，去上下文容器中取对应的handler类处理

2019-07-25

我已经设置了昵称

0

在a=b的地方的代码是否有问题，第二个stu1.setName("test2");应该改为stu2.setName("test2")

2019-07-25

作者回复

是的，感谢提醒！

2019-07-30

-W.LI-

0

老师好，具体啥时候用原型模式啊?循环有很多，中不能每个DTO都搞个原型类吧能透露下评价原则么?谢谢老师pojo,dto,vo，再加一个原型类爆炸。

2019-07-25

作者回复

我相信大部分同学都很少使用到原型模式来创建对象，如果每个类实现clone方法，有点走极端优化了，这里只是针对一些特殊场景，比如在一些循环中创建对象的类，我们可以实现clone方法来复制对象。

2019-07-30

-W.LI-

0

```
// 抽象享元类
interface Flyweight {
// 对外状态对象
void operation(String name);
// 对内对象
```

String getType();

}

老师好!享员工厂为啥不用operation的入参name做key?

2019-07-25

作者回复

因为name是对外状态的一个对象，不存在共享。

2019-07-27

全有

0

享元模式的给工厂类，是用HashMap 来存储共享对象，在多线程下并不安全，同时也没有加锁判定，依然会存在创建个对象，只是会覆盖掉

2019-07-25

a、

0

我觉得单例模式和享元模式的区别，享元模式可以看成是一组单例模式。

2019-07-25