二

27 原型模式与享元模式：提升系统性能的利器

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;
        stu1.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
   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(
        System.out.println(" 学生 " + stu1.getName + " 的老师是:" + stu2.getTeacher(
    }
}
```

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

前面我详讲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", ty
    }

    @Override
    public String getType() {
        return type;
    }
}
// 享元工厂类
class FlyweightFactory {
    private static final Map<String, Flyweight> FLYWEIGHT_MAP = new HashMap<>();//

    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？

上一页 下一页