[置顶]设计模式是什么鬼（工厂方法）

凸凹 设计模式 2018年12月17日 1.06K 0 1

//本文作者：凸凹里歐

//本文收录菜单栏：《设计模式是什么鬼》专栏中

工厂是对对象构造、实例化、初始化过程的一种封装，以提供给其他需要对象的地方去使用，以降低耦合，提高系统的扩展性，重用性。众所周知，当我们需要把类实例化成对象的时候，需要用到关键字new，比如Plane = new Plane()，这也是我们最常用的方式了。

然而，这样做的结果就是会把这个对象的诞生过程死死捆绑在我们的代码里，宿主类与实例化过程强耦合。对于一些庞大复杂的系统来说，过多的实例化逻辑于宿主类中会给其后期的维护与扩展带来很多麻烦。

而事实是我们根本不关心到底使用哪个对象；怎样生产出它的实例；制造过程是怎样，我们只在乎谁能给我产品来完成我的任务。为了满足用户需求，解决用户的痛点，工厂粉墨登场。

相信大家都玩过打飞机游戏吧，虽然这个主题的游戏版本繁杂但大同小异，都逃不出主角强大的武器系统，以及敌众我寡的战斗形式，所以敌人的种类就得花样百出以带来丰富多样的游戏体验。那么就从这款游戏入手，开始代码。

凸凹 官方
关注 私信
凸凹里歐，十余年以上开发经验，《设计
列文章作者，旨在用最通俗的方式诠释设

最新文章
设计模式是什么鬼（访问者）
设计模式是什么鬼（命令模式）
设计模式是什么鬼（建造者）
设计模式是什么鬼（抽象工厂）
设计模式是什么鬼（工厂方法）

极客快讯
传Android 要收费，周鸿祎回应
2019年1月4日 124
Python 3.7 上架微软商店
2019年1月3日 90
fish 3.0.0 发布
2019年1月3日 194
股价跳水、变现受阻，美图做中国下戏吗
2019年1月3日 66
中兴通讯5G核心网率先通过IMT202
2019年1月3日 46

首先来定义所有敌人的总抽象，我们想想，敌人们统统都得有一对坐标用来表达位置状态，以便可以把敌人绘制到地图上。为了让子类继承坐标，这里我们使用抽象类来定义敌人。

```
public abstract class Enemy {
    //敌人的坐标，会被子类继承。
    protected int x;
    protected int y;

    //初始化坐标
    public Enemy(int x, int y){
        this.x = x;
        this.y = y;
    }

    //抽象方法，在地图上绘制。
    public abstract void show();

}
```

这里我们只定义一个抽象方法show，可以把敌人绘制在地图上（下一帧会擦除重绘到下一个坐标以实现动画），当然真正的游戏或许还会有move（移动）、attack（攻击）、die（死亡）等等方法我们这里保持简单就忽略掉了。接下来是具体子类，我们这里假设只有两种，敌机类和坦克类。

```
public class Airplane extends Enemy {

    public Airplane(int x, int y){
        super(x, y);//调用父类构造子初始化坐标
    }

    @Override
    public void show() {
        System.out.println("飞机出现坐标：" + x + "," + y);
        System.out.println("飞机向玩家发起攻击……");
    }

}
```

```
public class Tank extends Enemy {

    public Tank(int x, int y){
        super(x, y);
    }

    @Override
    public void show() {
        System.out.println("坦克出现坐标：" + x + "," + y);
        System.out.println("坦克向玩家发起攻击……");
    }

}
```

一如既往地简单，飞机和坦克分别实现不同的show()方法。接下来开始运行游戏并实例化敌人了，重点在于怎样去实例化这些敌人，毋庸置疑要使它们出现在屏幕最上方，也就是纵坐标y等于0，但对于横坐标x我们怎样去初

```
1  public class Client {
2
3      public static void main(String[] args) {
4          int screenWidth = 100;//屏幕宽度
5          System.out.println("游戏开始");
6          Random random = new Random();//准备随机数
7
8          int x = random.nextInt(screenWidth);//生成敌机横坐标随机数
9          Enemy airplan = new Airplane(x, 0);//实例化飞机
10         airplan.show();//显示飞机
11
12         x = random.nextInt(screenWidth);//坦克同上
13         Enemy tank = new Tank(x, 0);
14         tank.show();
15
16         /*输出结果：
17             游戏开始
18             飞机出现坐标：94,0
19             飞机向玩家发起攻击……
20             坦克出现坐标：89,0
21             坦克向玩家发起攻击……
22         */
23 }
```

对，我们在第8行获取了一个从0到屏幕宽度（为了不让敌人出现在屏幕之外）的随机数，作为敌人的横坐标并初始化了敌人，这样每次出现的位置就会不一样了，游戏随机性增强，问题解决了（我们保持简单不考虑敌人自身的宽度）。我们发现从第8行和第12行是在做同样的事情，如果其他地方也需要实例化会出现重复的逻辑，尤其我们还进行了代码省略，实际的逻辑会更复杂，重复代码会更多。如此耗时费力，何不把这些实例化逻辑抽离出来作为一个工厂类？好，开始简单工厂的开发。

```
1  public class SimpleFactory {
2      private int screenWidth;
3      private Random random;//随机数
4
5      public SimpleFactory(int screenWidth) {
6          this.screenWidth = screenWidth;
7          this.random = new Random();
8      }
9
10     public Enemy create(String type){
11         int x = random.nextInt(screenWidth);//生成敌人横坐标随机数
12         Enemy enemy = null;
13         switch (type) {
14         case "Airplane":
15             enemy = new Airplane(x, 0);//实例化飞机
16             break;
17         case "Tank":
18             enemy = new Tank(x, 0);//实例化坦克
19             break;
20         }
21         return enemy;
22     }
23
24 }
```

其实这就是简单工厂了，为客户端省去了很多烦扰，于是我们的代码变得异常简单。

```
1  public class Client {
2
3      public static void main(String[] args) {
4          System.out.println("游戏开始");
5          SimpleFactory factory = new SimpleFactory(100);
6          factory.create("Airplane").show();
7          factory.create("Tank").show();
8      }
9
10 }
```

然而，这个简单工厂并不是一种设计模式，它只是对实例化逻辑进行了一层简单包裹而已，客户端依然是要告诉工厂我要的是哪个产品，虽然没有出现对产品实例化的关键字new，但这依然无疑是另一种形式的耦合。虽然我们在简单工厂中巧妙利用了坐标随机化来丰富游戏性，但又一个问题出现了，虽然坐标随机变化，但敌人的种类总是不变，游戏又开始变得无聊起来，于是随机生产敌人的工厂迫在眉睫。

得难以维护，简单工厂不简单，这显然违反了设计模式原则。

从另一方面来讲，用户的需求是多变的，我们要满足各种复杂情况，其实有些时候客户端目的很明确单纯，就是简单的需要工厂生产一个坦克而已，那么我们还有必要加载实例化这么臃肿一个简单工厂类么？问题显而易见了，简单工厂应对简单情况，而针对我们的场景，简单工厂需要多态化，我们应该对生产方式（工厂方法）进行抽象化。首先，定义一个工厂接口。

```
1  public interface Factory {
2
3      public Enemy create(int screenWidth);
4
5  }
```

这个工厂接口就是工厂方法的核心了，它具备这么一个功能（第3行），可以在屏宽之内来产出一个敌人，这就是我们抽象出来的工厂方法。然后我们来定义这个工厂方法的子类实现，随机工厂。

```
1  public class RandomFactory implements Factory {
2
3      private Random random = new Random();
4
5      @Override
6      public Enemy create(int screenWidth){
7          Enemy enemy = null;
8          if(random.nextBoolean()){
9              enemy = new Airplane(random.nextInt(screenWidth), 0);//实例化飞机
10         }else{
11             enemy = new Tank(random.nextInt(screenWidth), 0);//实例化坦克
12         }
13         return enemy;
14     }
15 }
```

代码非常简洁明了，这个随机工厂无疑具备生产实力，也就是在第6行实现的工厂方法，但其产出方式是随机产出，拒不退换，我只管努力制造，但出来的是飞机还是坦克，这个由天定。

这也许有点霸权主义，我们也许需要加一些其他工厂，比如某局出现了太多的坦克，一个飞机都没有，这是工厂的随机机制造成的，于是我们可以增加一个平衡工厂让飞机和坦克交替生成，这就好比大型网站上的负载均衡的平衡策略一样，让服务器轮流接受请求。

除了以上工厂，我们或许可以为每关做一个脚本工厂，根据主角关卡进度生产该出现的敌人，又或许更具体点为每个产品做一个工厂，总之，我们可以灵活地根据自己的具体需求去实现不同的工厂，每个工厂的生产策略和方式是不同的，最终是由客户端去决定用哪个工厂生产产品。比如，玩家抵达关底，boss要出现了。

```
1  public class Boss extends Enemy {
2
3      public Boss(int x, int y){
4          super(x, y);
5      }
6
7      @Override
8      public void show() {
9          System.out.println("Boss出现坐标：" + x + "," + y);
10         System.out.println("Boss向玩家发起攻击……");
11     }
12
13 }
```

接着来实现Boss的工厂方法，此处要注意Boss出现坐标是在屏幕中央，在第6行处设置横坐标为屏幕的一半。

```
1  public class BossFactory implements Factory {
2
3      @Override
4      public Enemy create(int screenWidth) {
5          // boss应该出现在屏幕中央
6          return new Boss(screenWidth / 2, 0);
7      }
```

```
1  public class Client {
2
3      public static void main(String[] args) {
4          int screenWidth = 100;
5          System.out.println("游戏开始");
6          Factory factory = new RandomFactory();
7          for (int i = 0; i < 10; i++) {
8              factory.create(screenWidth).show();
9          }
10         System.out.println("抵达关底");
11         factory = new BossFactory();
12         factory.create(screenWidth).show();
13
14         /*
15             游戏开始
16             飞机出现坐标：27,0
17             飞机向玩家发起攻击……
18             坦克出现坐标：40,0
19             坦克向玩家发起攻击……
20             飞机出现坐标：30,0
21             飞机向玩家发起攻击……
22             坦克出现坐标：53,0
23             坦克向玩家发起攻击……
24             坦克出现坐标：19,0
25             坦克向玩家发起攻击……
26             飞机出现坐标：18,0
27             飞机向玩家发起攻击……
28             坦克出现坐标：27,0
29             坦克向玩家发起攻击……
30             飞机出现坐标：89,0
31             飞机向玩家发起攻击……
32             飞机出现坐标：24,0
33             飞机向玩家发起攻击……
34             飞机出现坐标：31,0
35             飞机向玩家发起攻击……
36             抵达关底
37             Boss出现坐标：50,0
38             Boss向玩家发起攻击……
39         */
40     }
41
42 }
```

此处我们于第7行循环10次调用随机工厂生成随机敌人，有时出飞机，有时出坦克，玩家永远猜不透。抵达关底后于第11行换成Boss工厂，并生成Boss，如此一来，我们有产品需要就直接问工厂索要便是，至此客户端与敌人的实例化解耦脱钩。

相比简单工厂，工厂方法可以被看做是一个升级为设计模式的变种，其工厂方法的抽象化带来了极大好处，与其把所有生产方式堆积在一个简单工厂类中，不如把生产方式被推迟到具体的子类工厂中实现，工厂本身也是需要分类的，这样后期的代码维护以及对新产品的扩展都会更加方便直观，而不是对单一工厂类翻来覆去地不停改动。

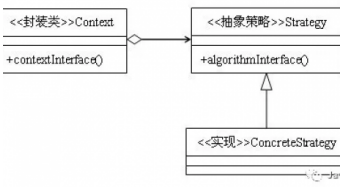
工厂不是万能的，方便面工厂不能生产汽车，手机工厂更不能生产辣条，这本身就看起来很荒诞，妄想吞噬兼备所有产品的工厂不是好的专业工厂。

本文作者：凸凹

设计模式(62)

上一篇：数据库 SQL 优化大总结之：百万级数据库优... 下一篇：推荐几个IDEA插件，Java开发者撸码利器。

相关文章



23种设计模式介绍-策略模式

简说备忘录模式

设计模式是什么鬼（初探）

设计模式是什么鬼（装饰）

23种设计模式介绍-装饰者模式

设计模式六大原则（2）：里氏替换原则

发表评论

评分

您的评论一针见血（必填，该内容可在后台设置）

昵称 邮箱

网址

发表评论