结构型模式-享元模式

概述：
　　由于对象创建的开销，面向对象的系统可能会面临性能问题；性能问题通常在资源受限的嵌入式系统中出现，比如智能手机和平板电脑；大型复杂系统中也可能会出现同样的问题，因为要在其中创建大量对象(用户)，这些对象需要同时并存．这个问题之所以会发生，是因为当我们创建一个新对象时，需要分配额外的内存，虽然虚拟内存理论上为我们提供了无限制的内存空间，但现实却并非如此．如果一个系统耗尽了所有的物理内存，就会开始将内存页替换到二级存储设备，通常是硬盘驱动器(Hard Disk Drive HDD) 在多数情况下，由于内存和硬盘之间的性能差异，这是不能接受的；固态硬盘(Solid State Drive，SSD) 的性能一般比硬盘更好，但并非人人都使用SSD；SSD并不会很快全面替代硬盘．除内存使用之外，计算性能也是一个考虑点．图形软件 包括计算机游戏，应该能够极快地渲染3D信息(例如 有成千上万颗树的森林或满是士兵的村庄)，如果一个3D地带的每个对象都是单独创建，未使用数据共享，那么性能将是无法接受的．作为软件工程师，我们应该编写更好的软件来解决软件问题，而不是要求客户购买更多更好的硬件．
　　享元设计模式通过为相似对象引入数据共享来最小化内存使用来提升性能，一个享元就是一个包含状态独立的不可变数据的共享对象，状态的可变(又称非固有的)数据不应是享元的一部分，因为每个对象的这种信息都不同，无法共享．如果享元需要非固有的数据，应该由客户端代码显示地提供．

　　用一个例子可能有助于解释实际场景中如何使用享元模式，在游戏中玩家共享一些状态，如外在表现和行为．例如在cs游戏中，同一团队的所有玩家看起来都是一样的(外在表现)．同一个游戏中(两个团队的)所有玩家都有一些共同的动作，比如跳起、低头等(行为)．这意味着我们可以创建一个享元来包含所有共同的数据，当然玩家也有许多因人而异的可变数据，这些数据不是享元的一部分．比如，枪支、健康状况和地理位置等．

应用场景：
　　1)应用需要使用大量的对象
　　2)对象太多，存储/渲染它们的代价太大．
　　3)对象ID对于应用不重要，对象共享会造成ID比较的失败，所以不能依赖对象ID(那些在客户端代码看来不同的对象，最终具有相同的ID)

未使用享元模式示例代码：
　　我们需要模拟一片树林，其中有不同年龄的苹果树、梨树和樱桃树分布在不同的位置．我们先定义Tree这个类，并设置TreeType为几种树木的枚举．

In [1]:
```
from enum import Enum
TreeType = Enum('TreeType', 'apple_tree cherry_tree peach_tree')
class Tree:
    def __init__(self, tree_type, age, x, y):
        self.tree_type = tree_type
        self.age = age
        self.x = x
        self.y= y
    def __str__(self):
        s = '类型:%s 年龄:%s 坐标:%s %s' %(self.tree_type, self.age, self.x, self.y)
        return s

if __name__ == '__main__':
    x = Tree(TreeType.apple_tree, 20, 100, 200)
    print(x)
```

```
类型:TreeType.apple_tree 年龄:20 坐标:100 200
```

　　如此我们就成功创建了一个Tree的类，其中的tree_type用于标识每个实例到底是哪一种树．但是，这样一来我们就会发现，当树林中树木非常多的时候，我们的树对象的数量将会急剧膨胀，在数量很大或者资源很紧张的时候是不可接受的．所以我们需要考虑使用享元模式来提取通用数据以节约资源．

In [9]:
```
#coding:utf8

from random import randint

from enum import Enum
TreeType = Enum('TreeType', 'apple_tree cherry_tree peach_tree')

class Tree(object):
    pool = {}
    def __new__(cls, tree_type):
        t = cls.pool.get(tree_type, None)
        if t:
           pass
        else:
            t = object.__new__(cls)
            cls.pool[tree_type] = t
            t.tree_type = tree_type
        return t

    def render(self, age, x, y):
        s = "类型: %s 年龄: %s 位置 %s %s" %(self.tree_type, age, x, y)
        print(s)
        return s

def main():
    age_min, age_max = 1, 30
    min_point, max_point = 1, 100
    tree_count = 0

    for _ in range(10):
        t1 = Tree(TreeType.apple_tree)
        t1.render(randint(age_min,age_max), randint(min_point, max_point), randint(min_point, max_point))
        tree_count += 1

    for _ in range(3):
        t1 = Tree(TreeType.cherry_tree)
        t1.render(randint(age_min,age_max), randint(min_point, max_point), randint(min_point, max_point))
        tree_count += 1

    for _ in range(5):
        t1 = Tree(TreeType.peach_tree)
        t1.render(randint(age_min, age_max), randint(min_point, max_point), randint(min_point, max_point))
        tree_count += 1

    print('总共种树: %s' %tree_count)
    print('总共种了%s种树' %len(Tree.pool))

    t4 = Tree(TreeType.apple_tree)
    x = t4.render(4,4, 4)
    print(x, 'x')
    t5 = Tree(TreeType.apple_tree)
    y = t5.render(5,5,5)
    print(y, 'y')
    print(x, 'x')

    t6 = Tree(TreeType.peach_tree)
    z = t6.render(6,6,6)
    print(z, 'z')
    print("id_t4: %s, id_t5: %s, id_t6:%s" %(id(t4), id(t5), id(t6)))


if __name__ == '__main__':
    main()
```

```
类型: TreeType.apple_tree 年龄: 10 位置 64 47
类型: TreeType.apple_tree 年龄: 13 位置 99 57
类型: TreeType.apple_tree 年龄: 17 位置 49 14
类型: TreeType.apple_tree 年龄: 8 位置 1 73
类型: TreeType.apple_tree 年龄: 16 位置 79 73
类型: TreeType.apple_tree 年龄: 22 位置 59 88
类型: TreeType.apple_tree 年龄: 2 位置 30 99
类型: TreeType.apple_tree 年龄: 27 位置 58 70
类型: TreeType.apple_tree 年龄: 2 位置 27 41
类型: TreeType.apple_tree 年龄: 2 位置 94 40
类型: TreeType.cherry_tree 年龄: 25 位置 47 23
类型: TreeType.cherry_tree 年龄: 11 位置 10 17
类型: TreeType.cherry_tree 年龄: 6 位置 59 33
类型: TreeType.peach_tree 年龄: 16 位置 19 9
类型: TreeType.peach_tree 年龄: 28 位置 17 7
类型: TreeType.peach_tree 年龄: 1 位置 61 47
类型: TreeType.peach_tree 年龄: 1 位置 29 23
类型: TreeType.peach_tree 年龄: 21 位置 91 71
总共种树: 18
总共种了3种树
类型: TreeType.apple_tree 年龄: 4 位置 4 4
类型: TreeType.apple_tree 年龄: 4 位置 4 4 x
类型: TreeType.apple_tree 年龄: 5 位置 5 5
类型: TreeType.apple_tree 年龄: 5 位置 5 5 y
类型: TreeType.apple_tree 年龄: 4 位置 4 4 x
类型: TreeType.peach_tree 年龄: 6 位置 6 6
类型: TreeType.peach_tree 年龄: 6 位置 6 6 z
id_t4: 4577608704, id_t5: 4577608704, id_t6:4577608928
```

将Tree类如此修改一番，我们就实现了享元模式，简单做一个讲解：为Tree类型提供了一个类属性pool代表这个类的对象池——可以理解为一个对象数据的缓存，而def __new__方法，将Tree类改造为一个元类，实现了引用自身的目的. 于是在每次创建新的对象时先到pool中检查是否有该类型的的对象存在，如果存在就直接返回之前创建好的那个对象，如果不存在则在pool中添加这个新的对象，如此一来就实现了对象的复用. 这里要注意的是，我们使用了__new__方法后，移除了__init__方法，并把age，x和y等可变数据都放到了其他地方，就是为了保留最通用的数据，这也是实现享元模式的核心.

可以看到，最后的内存单元编号证明，同样类型的树木对象共享了同一个树木类型数据；但会改变的部分数据——年龄 位置又互不影响，在树木数量超大时，将有效提升性能.

最后要注意：享元模式不能依赖对象的id，在上面的测试实例中可以看到，因为使用了享元模式，所以本来是不同的两棵树，在做id对比时，却是同一个对象.

优点：

1)系统中存在大量的相似对象时，可以选择享元模式提高资源利用率. 若假设一个电商平台，每个买家和卖家建立起买卖关系后，买家对象和卖家对象都是占用资源的，如果一个卖家同时与多个买家建立起买卖关系时候，此时享元模式的优势就体现出来了.

2)需要缓冲池的场景中，可以使用享元模式. 如进程池，线程池等技术，就可以使用享元模式(事实上，很多的池技术中已经使得了享元模式)

缺点：

1) 享元模式虽然节约了系统资源，但同时也提高了系统的复杂性，尤其当遇到外部状态和内部状态混在一起时，需要先将其进行分离，才可以使用享元模式.否则会引起逻辑混乱或业务风险.

2)享元模式中需要额外注意线程安全问题.

In [ ]: