

享元模式

2020年2月10日 19:28

1. 概念

享元模式 (Flyweight Pattern) 主要用于减少创建对象的数量, 以减少内存占用和提高性能, 提供了减少内存中对象数量从而改善应用所需的对象结构的方式。运用共享技术有效的支持大量细粒度的对象。

享元模式尝试重用现有的同类对象, 如果未找到匹配的对象, 则创建新对象。

- 内部状态: 在享元对象的内部, 不会随着环境的改变而改变的共享部分。
- 外部状态: 在享元对象外部, 随着环境改变而改变, 且不可以共享, 对象得以依赖的一个标记。

例如设计围棋游戏时, 不可能为每个棋子创建一个对象, 可以通过享元模式进行设计, 其中棋子的颜色为内部状态, 具体的方位坐标时棋子的外部状态, 所以只需要根据不同颜色创建棋子对象, 具体使用时利用具体方位坐标来标记一个对象。

2. UML结构图

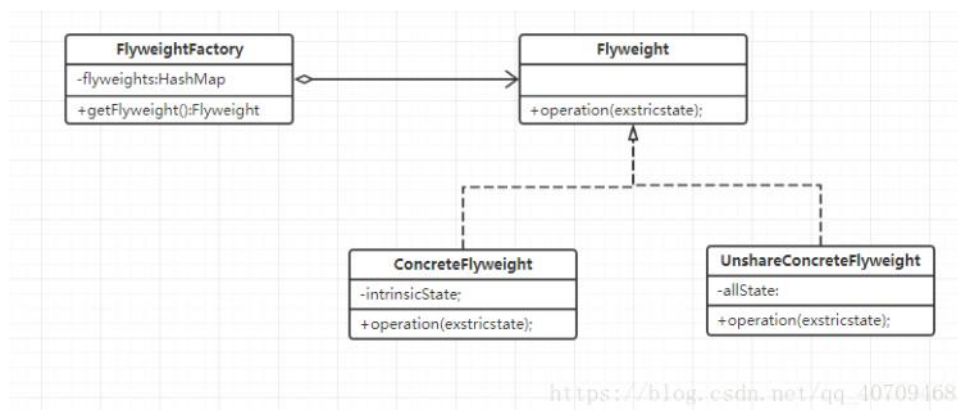


图2.1 享元模式结构图

1. Flyweight (享元抽象类): 一般是接口或者抽象类, 定义了享元类的公共方法。这些方法可以分享内部状态的数据, 也可以调用这些方法修改外部状态。
2. ConcreteFlyweight(具体享元类): 具体享元类实现了抽象享元类的方法, 为享元对象开辟了内存空间来保存享元对象的内部数据。
3. UnshareConcreteFlyweight(非具体享元类): 并不是所有的享元类都需要被共享, 有的享元类就不要被共享, 可以通过享元类来实例一个非共享享元对象。
4. Flyweight(享元工厂类): 享元工厂类创建并且管理享元类, 享元工厂类针对享元类来进行编程, 通过提供一个享元池来进行享元对象的管理。一般享元池设计成键值对, 或者其他的存储结构来存储。当客户端进行享元对象的请求时, 如果享元池中有对应的享元对象则直接返回对应的对象, 否则工厂类创建对应的享元对象并保存到享元池。

3. 实例

一个公司有很多部门, 每个部门的部门经理都经常做报告, 遂通过享元模式对部门经理进行管理。

1. 享元抽象类

```
public interface Employee {
```

```
} public void report(String reportContent);  
}
```

2. 具体享元类

```
public class Manager implements Employee {  
    private String title = "部门经理"; // 内部状态  
    private String department; // 内部状态  
  
    public Manager(String department) {  
        this.department = department;  
    }  
  
    @Override  
    public void report(String reportContent) {  
        System.out.println(reportContent);  
    }  
}
```

3. 享元工厂类

```
public class EmployeeFactory {  
    private static final Map<String, Employee> EMPLOYEE_MAP = new HashMap<String, Employee>();  
    public static Employee getManager(String department){  
        Manager manager = (Manager) EMPLOYEE_MAP.get(department);  
        if (manager == null){  
            manager = new Manager(department);  
            System.out.print("创建" + department + "的部门经理");  
            EMPLOYEE_MAP.put(department,manager);  
        }  
        return manager;  
    }  
}
```

4. 测试

```
private static final String departments[] = {"RD","QA","QM","BD"};  
@Test  
public void flyweightTest(){  
    for (int i = 0; i < 10; i++){  
        String department = departments[(int)(Math.random() * departments.length)];  
        Manager manager = (Manager) EmployeeFactory.getManager(department);  
        reportContent = department + "部门经理准备报告"; // 外部状态, 对象得以依赖的一个标记  
        manager.report(reportContent);  
    }  
}
```

5. 结果

创建BD的部门经理
BD的部门经理准备报告
BD的部门经理准备报告
创建RD部门经理
RD的部门经理准备
BD的部门经理准备
创建QM部门经理
QM的部门经理准备
.....

4. 优/缺点

1. 优点

- 减少对象的创建，降低内存中对象的数量，降低系统的内存，提高效率。
- 减少内存之外的其他资源占用(例如减少时间的占用)。

2. 缺点

- 关注内/外部状态，关注线程安全问题
- 使系统，程序的逻辑更加复杂化

5. 应用场景

- 常用于系统底层的开发，以便解决系统的性能

问题，例如Java中的String类型，使用String常量池，比如我们每次创建字符串对象时，都需

要创建一个新的字符串对象的话，内存开销会很大，所以如果第一次创建了字符串对象"String"，下次再创建相同的字符串"String"时，只是把它的引用指向"String"，这样就实现了"String"字符串在内存中的共享。

- 系统中有大量的相似对象，需要缓冲池的场景，例如数据库连接池。