第8章

单例（Singleton）模式

通常，对象通过在自身属性上执行任务来承担自己的职责，除了需要维护自身的一致性外，无须承担其他任何职责。然而，仍有一些对象承担了更多的职责，例如对真实世界的实体进行建模、协调工作或者对整个系统的状态进行建模。当系统的其他对象都依赖于特殊对象所承担的职责时，我们需要通过某种方式找到这个承担职责的对象。例如，可能需要找到一个代表指定机器的对象，或者是从数据库获取数据来创建自身的客户对象，又或者是初始化系统内存用以恢复的对象。

在某些场景，你需要找到一个承担职责的对象，并且这个对象是它所属类的唯一实例。例如，焰火工厂可能需要唯一的一个 Factory 对象。此时，可以使用单例模式，单例模式的英文为 Singleton。

单例模式的意图是为了确保一个类有且仅有一个实例，并为它提供一个全局访问点。

单例模式机制

单例模式的机制比其意图更加容易记忆。解释如何保证一个类有且仅有一个实例，要比解释为何需要这种限制要简单得多。正如 *Design Patterns* 一书就把单例模式归类为“创建型”模式。当然，理解模式的方式是没有限制的，只要能帮助你记忆、识别和运用这些模式。对于单例模式，它的意图说明了该单例对象承担了其他对象所要依赖的职责。

创建一个担当着独一无二角色的对象，有多种方式。但是，不管你如何创建一个单例对象，

都必须确保其他开发人员不能创建该单例对象的新的实例。

挑战 8.1

怎样才能阻止其他开发人员不能创建类的新实例？

答案参见第 312 页

设计一个单例类时，需要确定何时实例化该类的单例对象。一种做法是创建这个类的实例，并将它作为该类的静态成员变量。例如，SystemStartup 类可能包括这一行：

```
private static Factory factory = new Factory();
```

这个类通过一个公共的 getFactory()静态方法获得该类的唯一实例。

如果不希望提前创建单例实例，还可以在第一次需要该实例时，延迟初始化它。例如，SystemStartup 类可能采用如下方式获取单个实例：

```
public static Factory getFactory() {
    if (factory == null)
        factory = new Factory();
    // ...
    return factory;
}
```

挑战 8.2

为什么要决定延迟初始化一个单例实例，而不是在声明时就初始化？

答案参见第 312 页

无论哪种场景，单例模式都建议提供一个公共的静态方法去访问单例对象。如果该方法创建了一个对象，它就要保证只有一个实例可以被创建。

单例和线程

如果想在一个多线程环境下延迟初始化一个单例模型，必须小心谨慎地避免多个线程同时

初始化该单例对象。在多线程环境下，无法保证在其他线程开始执行该方法时，当前线程已经完整地执行完该方法。这可能出现两个线程同时初始化一个单例对象的情况。假设第一个线程发现该单例对象为 null，紧接着第二个线程运行，也会发现该单例对象为 null。然后两个线程都会对该单例对象进行初始化。为了避免这种竞争，需要用锁机制去协调不同线程对同一方法的执行。

Java 支持多线程开发。它可以为每个对象提供一个锁，用于表示对象是否已经被线程所占用。为确保仅有一个线程初始化单例对象，可以通过对适当的对象进行加锁来同步初始化。其他需要互斥访问单例对象的方法，也可以通过相同的锁机制进行同步。若要了解在并发模式下如何进行面向对象编程，可以参考 *Concurrent Programming in Java™* 一书。书中建议使用属于当前类的锁进行同步，就像如下代码：

```
package com.oozinoz.businessCore;
import java.util.*;

public class Factory {
    private static Factory factory;
    private static Object classLock = Factory.class;
    private long wipMoves;

    private Factory() {
        wipMoves = 0;
    }
    public static Factory getFactory() {
        synchronized (classLock) {
            if (factory == null)
                factory = new Factory();

            return factory;
        }
    }

    public void recordWipMove() {
            // 挑战!
    }
}
```

getFactory()方法保证：当一个线程开始初始化单例的实例时，如果另一个线程也开始相同的操作，则第二个线程就会等待，直到获得 classLock 对象的锁。当它获得锁后，就会发现

该单例不再为 null（因为类仅可能有一个实例，所以可以使用单个静态锁）。

wipMoves 变量记录了半成品（WIP）的进展状况。每次把箱子放入一个新机器时，引起移动或记录移动步骤的子系统必须调用工厂单例类的 recordWipMove()方法。

挑战 8.3

写出 Factory 类 recordWipMove()方法的代码。

答案参见第 312 页

识别单例

这种独一无二的对象并不罕见。事实上，应用程序中的很多对象都承担了唯一的职责，既然如此，为何要创建拥有相同职责的两个对象？同样的，几乎每个类都拥有独一无二的角色，又何必为相同的类重复开发两次？然而，允许类只能拥有一个实例的单例类却极为罕见。事实上，一个对象或一个类是唯一的，并不意味着就是单例模式。考虑图 8.1 所示的类。

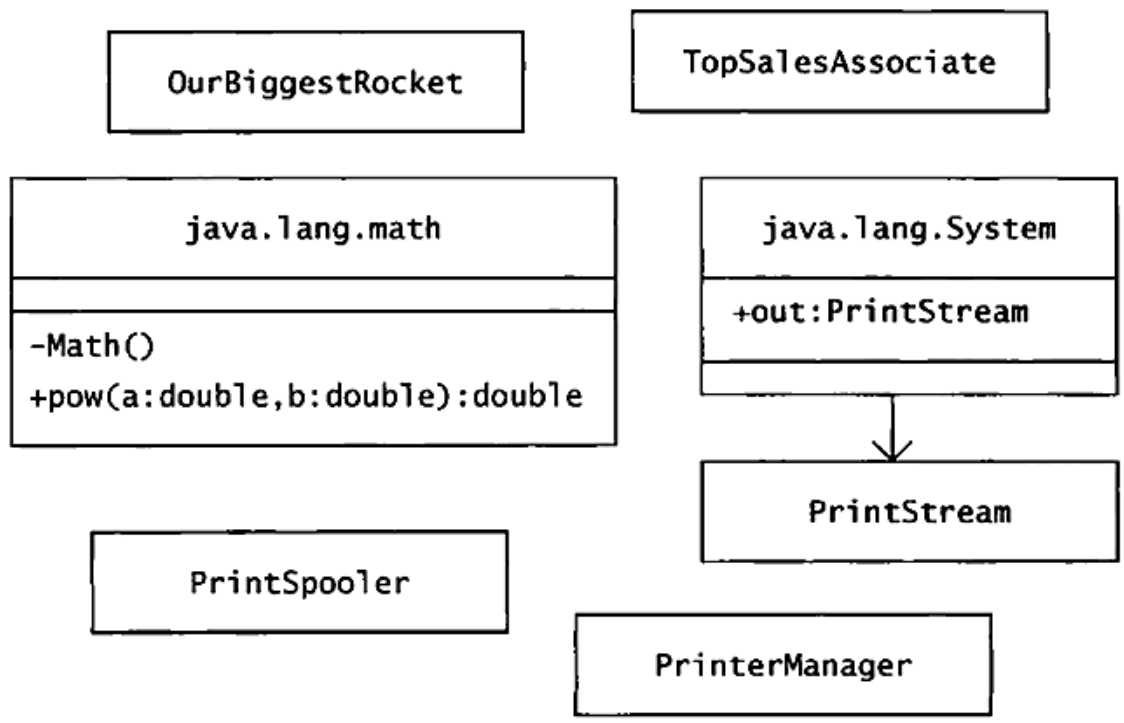

图 8.1 哪些类适用于单例模式

挑战 8.4

对于图 8.1 中的每个类，说说它们是否适用于单例类，为什么？

答案参见第 313 页

单例模式或许是最负盛名的一个模式了，但它很容易被误用，因此不要轻易使用。不要让单例模式成为创建全局变量的一种花哨方法。单例模式常常会引入一些耦合。应该减少运用单例模式的类的数量；最好的方式是，类只需知道与它协作的对象，却不必了解创建它所需要的限制。请注意：倘若需要为测试提供子类或不同的版本，由于单例只能拥有一个实例，因此它可能不是最佳选择。

小结

单例模式保证了类仅有一个实例，并为其提供了一个全局访问点。通过延迟初始化（仅在第一次使用它时才初始化），一个单例对象是达到此目的的通用做法。在多线程环境下，必须小心管理线程间的协作，因为它们访问单例对象方法与数据的时间，可能只有毫厘之差。

对象具有唯一性，并不意味着使用了单例模式。单例模式通过隐藏构造函数，提供对象创建的唯一入口点，从而将类的职责集中在类的单个实例中。