第3章

确保对象的唯一性——单例模式

从本章开始，正式进入设计模式的学习。模式虽多，但难度不一，本章将要介绍的单例模式是结构最简单的设计模式。单例模式用于创建那些在软件系统中独一无二的对象，是一个简单但很实用的设计模式，本书将从它开始为大家逐一展现设计模式的魅力。

3.1 单例模式的动机

对于一个软件系统中的某些类而言，无须创建多个实例。举个大家都熟知的例子——Windows 任务管理器，如图 3-1 所示。可以做一个这样的尝试：在 Windows 任务栏的右键弹出菜单上多次单击“启动任务管理器”，看能否打开多个任务管理器窗口（**注**：计算机中毒或私自修改 Windows 内核者除外）。在正常情况下，无论启动任务管理器多少次，Windows 系统始终只能弹出一个任务管理器窗口，也就是说，在一个 Windows 系统中，任务管理器存在唯一性。为什么要这样设计呢？可以从以下两个方面来分析：其一，如果能弹出多个窗口，且这些窗口的内容完全一致，全部是重复对象，这势必会浪费系统资源（任务管理器需要获取系统运行时的诸多信息，这些信息的获取需要消耗一定的系统资源，包括 CPU 资源及

图 3-1 Windows 任务管理器

内存资源等),而且根本没有必要显示多个内容完全相同的窗口;其二,如果弹出的多个窗口内容不一致,问题就更加严重了,这意味着在某一瞬间系统资源使用情况和进程、服务等信息存在多个状态,例如任务管理器窗口 A 显示“CPU 使用率”为 10%,窗口 B 显示“CPU 使用率”为 15%,到底哪个才是真实的呢?这会给用户带来误解,更不可取。由此可见,确保 Windows 任务管理器在系统中有且仅有一个非常重要。

在实际开发中也经常遇到类似的情况,为了节约系统资源,有时需要确保系统中某个类只有唯一一个实例,当这个唯一实例创建成功之后,无法再创建一个同类型的其他对象,所有的操作都只能基于这个唯一实例。为了确保对象的唯一性,可以通过单例模式来实现,这就是单例模式的动机所在。

3.2 单例模式概述

下面来模拟实现 Windows 任务管理器。假设任务管理器的类名为 TaskManager,在 TaskManager 类中包含了大量的成员方法,例如构造函数 TaskManager(),显示进程的方法 displayProcesses(),显示服务的方法 displayServices()等,该类的示意代码如下:

```
class TaskManager {
    public TaskManager() {…}                    //初始化窗口
    public void displayProcesses() {…}          //显示进程
    public void displayServices() {…}           //显示服务
…
}
```

为了实现 Windows 任务管理器的唯一性,通过以下 3 步对 TaskManager 类进行重构:

(1) 由于每次使用 new 关键字来实例化 TaskManager 类时都将产生一个新对象,为了确保 TaskManager 实例的唯一性,需要禁止类的外部直接使用 new 来创建对象,因此需要将 TaskManager 的构造函数的可见性改为 private,代码如下:

```
private TaskManager() {…}
```

(2) 将构造函数的可见性改为 private 后,虽然类的外部不能再使用 new 来创建对象,但是在 TaskManager 的内部还是可以创建对象的,可见性只对类外有效。因此,可以在 TaskManager 中创建并保存这个唯一实例。为了让外界可以访问这个唯一实例,需要在 TaskManager 中定义一个静态的 TaskManager 类型的私有成员变量,代码如下:

```
private static TaskManager tm = null;
```

(3) 为了保证成员变量的封装性,将 TaskManager 类型的 tm 对象的可见性设置为 private,但外界该如何使用该成员变量并何时实例化该成员变量呢?答案是增加一个公有的静态方法,代码如下:

```
public static TaskManager getInstance() {
    if (tm == null) {
        tm = new TaskManager();              //自行实例化
```

```
        }
        return tm;
    }
```

在 getInstance()方法中首先判断 tm 对象是否存在，如果不存在(即 tm == null 为 true)，则使用 new 关键字创建一个新的 TaskManager 类型的 tm 对象，再返回新创建的 tm 对象；否则直接返回已有的 tm 对象。

需要注意的是 getInstance()方法的修饰符，首先它应该是一个 public 方法，以便外界其他对象使用；其次它使用了 static 关键字，即它是一个静态方法，在类外可以直接通过类名来访问，而无须创建 TaskManager 对象。事实上，在类外也无法创建 TaskManager 对象，因为构造函数是私有的。

思考

为什么要将成员变量 tm 定义为静态变量？

通过以上 3 个步骤，完成了一个最简单的单例类的设计，其完整代码如下：

```
class TaskManager {
    private static TaskManager tm = null;
    private TaskManager() {…}                    //初始化窗口
    public void displayProcesses() {…}           //显示进程
    public void displayServices() {…}            //显示服务

    public static TaskManager getInstance() {
        if (tm == null) {
            tm = new TaskManager();
        }
        return tm;
    }
    …
}
```

在类外无法直接创建新的 TaskManager 对象，但可以通过代码 TaskManager.getInstance()访问实例对象。第一次调用 getInstance()方法时将创建唯一实例，再次调用时将返回第一次创建的实例。

上述代码也是单例模式的一种最典型实现方式，有了以上基础，理解单例模式的定义和结构就非常容易了。单例模式定义如下：

> 单例模式(Singleton Pattern)：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。单例模式是一种对象创建型模式。

单例模式有 3 个要点：①某个类只能有一个实例；②它必须自行创建这个实例；③它必须自行向整个系统提供这个实例。

单例模式是结构最简单的设计模式，在它的核心结构中只包含一个被称为单例类的特

殊类。单例模式结构如图 3-2 所示。

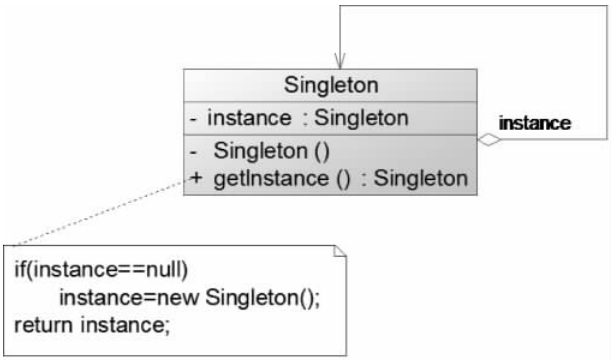

图 3-2　单例模式结构图

从图 3-2 可以看出，单例模式结构图中只包含一个单例角色。

Singleton(单例)：在单例类的内部实现只生成一个实例，同时它提供一个静态的 getInstance()方法，让客户可以访问它的唯一实例；为了防止在外部对单例类实例化，它的构造函数可见性为 private；在单例类内部定义了一个 Singleton 类型的静态对象，作为供外部共享访问的唯一实例。

3.3　负载均衡器的设计

Sunny 软件公司承接了一个服务器负载均衡(Load Balance)软件的开发工作，该软件运行在一台负载均衡服务器上，可以将并发访问和数据流量分发到服务器集群中的多台设备上进行并发处理，提高系统的整体处理能力，缩短响应时间。由于集群中的服务器需要动态删减，且客户端请求需要统一分发，因此需要确保负载均衡器的唯一性，即只能有一个负载均衡器来负责服务器的管理和请求的分发，否则将会带来服务器状态的不一致以及请求分配冲突等问题。如何确保负载均衡器的唯一性是该软件成功的关键。

Sunny 公司开发人员通过分析和权衡，决定使用单例模式来设计该负载均衡器，结构图如图 3-3 所示。

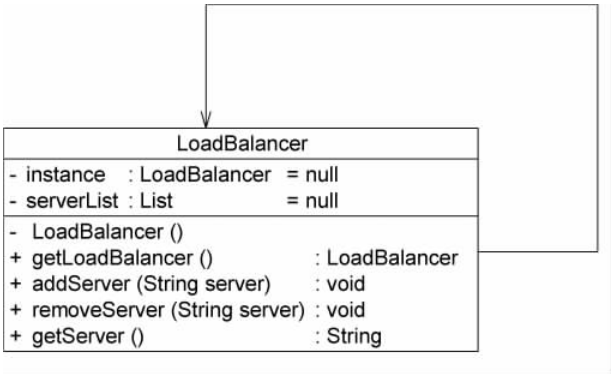

图 3-3　服务器负载均衡器结构图

在图 3-3 中，将负载均衡器 LoadBalancer 设计为单例类，其中包含一个存储服务器信息的集合 serverList，每次在 serverList 中随机选择一台服务器来响应客户端的请求，实现代码如下：

```
import java.util.*;

//负载均衡器 LoadBalancer: 单例类，真实环境下该类将非常复杂，包括大量初始化的工作和业务方
//法，考虑到代码的可读性和易理解性，只列出部分与模式相关的核心代码
class LoadBalancer {
    //私有静态成员变量，存储唯一实例
    private static LoadBalancer instance = null;
    //服务器集合
    private List serverList = null;

    //私有构造函数
    private LoadBalancer() {
        serverList = new ArrayList();
    }

    //公有静态成员方法，返回唯一实例
    public static LoadBalancer getLoadBalancer() {
        if (instance == null) {
            instance = new LoadBalancer();
        }
        return instance;
    }

    //增加服务器
    public void addServer(String server) {
        serverList.add(server);
    }

    //删除服务器
    public void removeServer(String server) {
        serverList.remove(server);
    }

    //使用 Random 类随机获取服务器
    public String getServer() {
        Random random = new Random();
        int i = random.nextInt(serverList.size());
        return (String)serverList.get(i);
    }
}
```

编写如下客户端测试代码：

```
class Client {
    public static void main(String args[]) {
```

```
        //创建 4 个 LoadBalancer 对象
        LoadBalancer balancer1,balancer2,balancer3,balancer4;
        balancer1 = LoadBalancer.getLoadBalancer();
        balancer2 = LoadBalancer.getLoadBalancer();
        balancer3 = LoadBalancer.getLoadBalancer();
        balancer4 = LoadBalancer.getLoadBalancer();

        //判断服务器负载均衡器是否相同
        if (balancer1 == balancer2 && balancer2 == balancer3 && balancer3 == balancer4) {
            System.out.println("服务器负载均衡器具有唯一性!");
        }

        //增加服务器
        balancer1.addServer("Server 1");
        balancer1.addServer("Server 2");
        balancer1.addServer("Server 3");
        balancer1.addServer("Server 4");

        //模拟客户端请求的分发
        for (int i = 0; i < 10; i++) {
            String server = balancer1.getServer();
            System.out.println("分发请求至服务器: " + server);
      }
    }
}
```

编译并运行程序,输出结果如下:

```
服务器负载均衡器具有唯一性!
分发请求至服务器: Server 1
分发请求至服务器: Server 3
分发请求至服务器: Server 4
分发请求至服务器: Server 2
分发请求至服务器: Server 3
分发请求至服务器: Server 2
分发请求至服务器: Server 3
分发请求至服务器: Server 4
分发请求至服务器: Server 4
分发请求至服务器: Server 1
```

虽然创建了 4 个 LoadBalancer 对象,但是它们实际上是同一个对象。因此,通过使用单例模式可以确保 LoadBalancer 对象的唯一性。

3.4 饿汉式单例与懒汉式单例的讨论

Sunny 公司开发人员使用单例模式实现了负载均衡器的设计,但是在实际使用中出现了一个非常严重的问题。当负载均衡器在启动过程中用户再次启动负载均衡器时,系统无

任何异常，但当客户端提交请求时出现请求分发失败。通过仔细分析发现原来系统中还是存在多个负载均衡器对象，导致分发时目标服务器不一致，从而产生冲突。为什么会这样呢？Sunny公司开发人员百思不得其解。

现在对负载均衡器的实现代码进行再次分析。当第一次调用getLoadBalancer()方法创建并启动负载均衡器时，instance对象为null值，因此系统将执行代码“instance = new LoadBalancer();”，在此过程中，由于要对LoadBalancer进行大量初始化工作，需要一段时间来创建LoadBalancer对象。而在此时，如果再一次调用getLoadBalancer()方法（通常发生在多线程环境中），由于instance尚未创建成功，仍为null值，判断条件“instance == null”为真值，因此代码“instance = new LoadBalancer();”将再次执行，导致最终创建了多个instance对象，这违背了单例模式的初衷，也导致系统发生运行错误。

如何解决该问题？至少有两种解决方案。在正式介绍这两种解决方案之前，先介绍一下单例类的两种不同实现方式——饿汉式单例类（Eager Singleton）和懒汉式单例类（Lazy Singleton）。

1. 饿汉式单例类

饿汉式单例类是实现起来最简单的单例类，其结构图如图3-4所示。

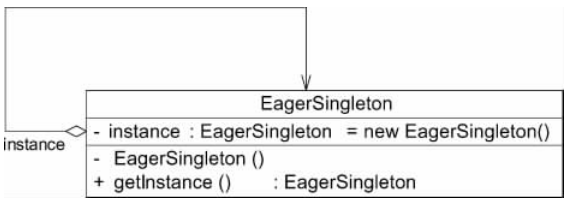

图3-4　饿汉式单例结构图

从图3-4中可以看出，由于在定义静态变量的时候实例化单例类，因此在类加载时就已经创建了单例对象，代码如下：

```
class EagerSingleton {
    private static final EagerSingleton instance = new EagerSingleton();
    private EagerSingleton() { }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

当类被加载时，静态变量instance会被初始化，此时类的私有构造函数会被调用，单例类的唯一实例将被创建。如果使用饿汉式单例来实现负载均衡器LoadBalancer类的设计，则不会出现创建多个单例对象的情况，可确保单例对象的唯一性。

2. 懒汉式单例类与线程锁定

除了饿汉式单例外，还有一种经典的懒汉式单例，也就是前面提到的负载均衡器

LoadBalancer 类的实现方式。懒汉式单例类结构图如图 3-5 所示。

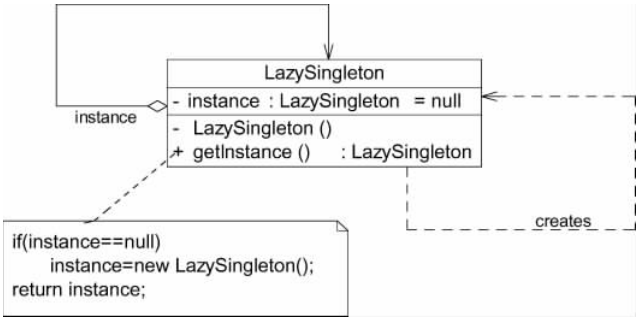

图 3-5 懒汉式单例结构图

从图 3-5 中可以看出，懒汉式单例在第一次调用 getInstance()方法时实例化，在类加载时并不自行实例化，这种技术又称为**延迟加载(Lazy Load)技术**，即需要的时候再加载实例。为了避免多个线程同时调用 getInstance()方法，可以使用关键字 synchronized，代码如下：

```
class LazySingleton {
    private static LazySingleton instance = null;

    private LazySingleton() { }

    synchronized public static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

该懒汉式单例类在 getInstance()方法前面增加了关键字 synchronized 进行线程锁定，以处理多个线程同时访问的问题。上述代码虽然解决了线程安全问题，但是每次调用 getInstance()时都需要进行线程锁定判断，在多线程高并发访问环境中，将会导致系统性能大大降低。如何既解决线程安全问题又不影响系统性能呢？继续对懒汉式单例进行改进。事实上，无须对整个 getInstance()方法进行锁定，只需锁定代码“instance = new LazySingleton();”即可。因此，getInstance()方法可以进行如下改进：

```
public static LazySingleton getInstance() {
    if (instance == null) {
        synchronized (LazySingleton.class) {
            instance = new LazySingleton();
        }
    }
    return instance;
}
```

问题貌似得以解决，事实并非如此。如果使用以上代码来创建单例对象，还是会存在单例对象不唯一。原因如下：

假如某一瞬间线程 A 和线程 B 都在调用 getInstance()方法，此时 instance 对象为 null 值，均能通过“instance == null”的判断。由于实现了 synchronized 加锁机制，线程 A 进入 synchronized 锁定的代码中执行实例创建代码，线程 B 处于排队等待状态，必须等待线程 A 执行完毕后才可以进入 synchronized 锁定代码。但当 A 执行完毕时，线程 B 并不知道实例已经创建，将继续创建新的实例，导致产生多个单例对象，违背单例模式的设计思想。因此需要进行进一步改进，在 synchronized 锁定代码中再进行一次“instance == null”判断，这种方式称为**双重检查锁定**(**Double-Check Locking**)。使用双重检查锁定实现的懒汉式单例类完整代码如下：

```
class LazySingleton {
    private volatile static LazySingleton instance = null;

    private LazySingleton() { }

    public static LazySingleton getInstance() {
        //第一重判断
        if (instance == null) {
            //锁定代码块
            synchronized (LazySingleton.class) {
                //第二重判断
                if (instance == null) {
                    instance = new LazySingleton();      //创建单例实例
                }
            }
        }
        return instance;
    }
}
```

需要注意的是，如果使用双重检查锁定来实现懒汉式单例类，需要在静态成员变量 instance 之前增加修饰符 volatile，被 volatile 修饰的成员变量可以确保多个线程都能够正确处理，且该代码只能在 JDK 1.5 及以上版本中才能正确执行。由于 volatile 关键字会屏蔽 Java 虚拟机所做的一些代码优化，可能会导致系统运行效率降低，因此即使使用双重检查锁定来实现单例模式也不是一种完美的实现方式。

扩展

IBM 公司高级软件工程师 Peter Haggar 2004 年在 IBM developerWorks 上发表了一篇名为《双重检查锁定及单例模式——全面理解这一失效的编程习语》的文章，对 JDK 1.5 之前的双重检查锁定及单例模式进行了全面分析和阐述，参考链接：http://www.ibm.com/developerworks/cn/java/j-dcl.html。

3. 饿汉式单例类与懒汉式单例类比较

饿汉式单例类在类被加载时就将自己实例化，它的优点在于无须考虑多线程访问问题，可以确保实例的唯一性；从调用速度和反应时间角度来讲，由于单例对象一开始就得以创建，因此要优于懒汉式单例。但是无论系统在运行时是否需要使用该单例对象，由于在类加载时该对象就需要创建，因此从资源利用效率角度来讲，饿汉式单例不及懒汉式单例，而且在系统加载时由于需要创建饿汉式单例对象，加载时间可能会比较长。

懒汉式单例类在第一次使用时创建，无须一直占用系统资源，实现了延迟加载。但是必须处理好多个线程同时访问的问题，特别是当单例类作为资源控制器，在实例化时必然涉及资源初始化，而资源初始化很有可能耗费大量时间，这意味着出现多线程同时首次引用此类的概率变得较大，需要通过双重检查锁定等机制进行控制，这将导致系统性能受到一定影响。

3.5 一种更好的单例实现方法

饿汉式单例类不能实现延迟加载，不管将来用不用，它始终占据内存；懒汉式单例类线程安全控制烦琐，而且性能受影响。可见，无论是饿汉式单例还是懒汉式单例都存在这样那样的问题。有没有一种方法，能够将两种单例的缺点都克服，而将两者的优点合二为一呢？答案是肯定的。下面来学习这种更好的被称为 Initialization on Demand Holder（IoDH）的技术。

实现 IoDH 时，需在单例类中增加一个**静态（static）内部类**，在该内部类中创建单例对象，再将该单例对象通过 getInstance()方法返回给外部使用，实现代码如下：

```
//Initialization on Demand Holder
class Singleton {
    private Singleton() {
    }

    private static class HolderClass {
        private final static Singleton instance = new Singleton();
    }

    public static Singleton getInstance() {
        return HolderClass.instance;
    }

    public static void main(String args[]) {
        Singleton s1, s2;
        s1 = Singleton.getInstance();
        s2 = Singleton.getInstance();
        System.out.println(s1 == s2);
    }
}
```

编译并运行上述代码，运行结果为：true，即创建的单例对象 s1 和 s2 为同一对象。由于静态单例对象没有作为 Singleton 的成员变量直接实例化，因此类加载时不会实例化 Singleton。第一次调用 getInstance()时将加载内部类 HolderClass，在该内部类中定义了一个 static 类型的变量 instance，此时会首先初始化这个成员变量，由 Java 虚拟机来保证其线程安全性，确保该成员变量只能初始化一次。由于 getInstance()方法没有被任何线程锁定，因此其性能不会造成任何影响。

通过使用 IoDH，既可以实现延迟加载，又可以保证线程安全，不影响系统性能。因此，IoDH 不失为一种最好的 Java 语言单例模式实现方式；其缺点是与编程语言本身的特性相关，很多面向对象语言不支持 IoDH。

练习

分别使用饿汉式单例、带双重检查锁定机制的懒汉式单例以及 IoDH 技术实现负载均衡器 LoadBalancer。

至此，3 种单例类的实现方式均已学习完毕，它们分别是饿汉式单例、懒汉式单例以及 IoDH。

3.6　单例模式总结

单例模式作为一种目标明确、结构简单、理解容易的设计模式，在软件开发中使用频率相当高，在很多应用软件和框架中都得以广泛应用。

1. 主要优点

单例模式的主要优点如下：

（1）单例模式提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它。

（2）由于在系统内存中只存在一个对象，因此可以节约系统资源。对于一些需要频繁创建和销毁的对象，单例模式无疑可以提高系统的性能。

（3）允许可变数目的实例。基于单例模式，开发人员可以进行扩展，使用与控制单例对象相似的方法来获得指定个数的实例对象，既节省系统资源，又解决了由于单例对象共享过多有损性能的问题。（**注**：自行提供指定数目实例对象的类可称之为多例类。）

2. 主要缺点

单例模式的主要缺点如下：

（1）由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。

（2）单例类的职责过重，在一定程度上违背了单一职责原则。因为单例类既提供了业务方法，又提供了创建对象的方法（工厂方法），将对象的创建和对象本身的功能耦合在一起。

（3）现在很多面向对象语言（如 Java、C＃）的运行环境都提供了自动垃圾回收技术，因

此，如果实例化的共享对象长时间不被利用，系统会认为它是垃圾，会自动销毁并回收资源，下次利用时又将重新实例化，这将导致共享的单例对象状态的丢失。

3. 适用场景

在以下情况下可以考虑使用单例模式：

（1）系统只需要一个实例对象。例如，系统要求提供一个唯一的序列号生成器或资源管理器，或者需要考虑资源消耗太大而只允许创建一个对象。

（2）客户调用类的单个实例只允许使用一个公共访问点。除了该公共访问点，不能通过其他途径访问该实例。

思考

Sunny 软件公司开发人员欲创建一个数据库连接池，将指定个数的（如 2 个或 3 个）数据库连接对象存储在连接池中，客户端代码可以从池中随机取一个连接对象来连接数据库。试通过对单例类进行改造，设计一个能够自行提供指定个数实例对象的数据库连接类。