文章已收录我的仓库：Java学习笔记与免费书籍分享

设计动机

正如其名，单例模式保证一个类只有一个实例，那么为什么需要设计单例模式？

对一些类来说，只有一个实例是很重要的，例如一台电脑只应该由一个文件系统，生产厂商不应该为一台电脑配置两个文件系统；一个应用应该有一个专属的日志对象，而不应该一会儿写到这里一会儿写到那里；一个程序中往往只有一个线程池，由一个线程池管理线程，而不应该使用多个线程池，那样会使得线程乱套并且难以维护；在抽象工厂中，具体的工厂类也只应该存在一个...

诸如此类的要求，我们都需要保证一个类只有一个实例，此时便可以使用单例模式。

设计

我们必须要防止用户实例化多个对象，解决办法是让类自己保存它的唯一实例，并将构造函数对外隐藏，并暴露特定静态方法返回唯一实例，这样用户将无法自己实例化多个对象而只能通过类对外暴露的静态方法获取唯一实例。

代码示例

假设需求如下：一个应用程序的全局状态中需要同一个日志对象。

一、懒汉模式

懒汉模式并不直接初始化实例，而是等到实例被使用时才初始化它，避免不必要的资源浪费。

```
//日志文件类
class LogFile {
    //唯一实例
    private static LogFile logFile = null;

    //构造方法对外隐藏
    private LogFile(){};
```

```
    //对外暴露的方法
    public static LogFile getInstance() {
        //懒汉模式
        if (logFile == null) {
            logFile = new LogFile();
        }
        return logFile;
    }
}

public class Test {
    public static void main(String[] args) {
        var s1 = LogFile.getInstance();
        var s2 = LogFile.getInstance();
        System.out.println(s1 == s2); //输出true，产生的是同一个对象
    }
}
```

这里的代码在单线程中运行良好，logFile属于临界区资源，因此这样的写法是线程不安全的，一开始实例为null，线程A执行完if判断句后在执行`logFile = new LogFile()`前被调度到线程B，此时线程B看到的实例也为空，因为A还没有初始化，所以线程B初始化实例，当回到线程A时，线程A将继续执行构造logFile的语句，此时logFile已经被初始化两次了，它们A与B拿到的已经不是同一个实例了。

一个简单的解决办法是为它们上锁：

```
public static LogFile getInstance() {
    synchronized (LogFile.class) {
        //懒汉模式
        if (logFile == null) {
            logFile = new LogFile();
        }
    }
    return logFile;
}
```

但是这样上锁效率太低了，还不如采用饿汉式，因为即时当logFile不为空时，多个线程也必须排队获取实例，而事实上并不需要排队，当logFile不为空时，多个线程应当可以同时获取logFile实例，因为它们仅仅只是读取实例而并不会更改实例，共享读是线程安全的。

一个更好的解决办法是采用双重检查锁定：

```
public static LogFile getInstance() {
    if (logFile == null) {
        synchronized (LogFile.class) {
            //懒汉模式
            if (logFile == null) {
                logFile = new LogFile();
            }
```

```
            }
        }
        return logFile;
    }
```

通过在外层加一重判断，我们解决了上述所说的问题，现在代码的效率已经够高了——仅仅在最开始的阶段才会涉及到加锁。

注意，上面的代码仍然是线程不安全的，如果要想线程安全，我们必须为logFile实例的声明加上`volatile`关键字，即：

```
private volatile static LogFile logFile;
```

要想理解这一点，我们必须要理解`new`一个对象的过程，大致的过程如下：

1. 申请内存空间，对空间内字段采用默认初始化(此时对象为null)。
2. 调用类的构造方法，进行初始化(此时对象为null)。
3. 返回地址(执行完成后对象不为null)。

如果不加`volatile`关键字，Java虚拟机可能在保证可串行化的前提下发生指令重排，即虚拟机可能先执行第3步再执行第2步(比较罕见的)，初始化对象时虚拟机考虑的仅仅只是单线程的情况，此时的指令重排并不会影响到单线程的运行，因此为了加快速度，指令重排这种情况是可能出现的。

如果从多线程角度来看，如果发生了指令重排，线程A在new对象时执行第一部后先执行了第三步，此时对象已经不为null了，但是对象还没被构造好，虽然这个时候线程A还持有锁，但这对线程B毫无影响——线程B闯入发现对象不为null而直接拿走一个还未构造完全的对象实例——根本不会通过第一层判断而申请锁。

加上`volatile`关键字可以保证可见性并且禁止指令重排。

但从解决问题的角度来看，我们还有更好的解决办法——静态内部类：

```
//日志文件类
class LogFile {
    //实例交给静态内部类保管
    private static class LazyHolder {
        private static LogFile logFile = new LogFile();
    }

    //构造方法对外隐藏
    private LogFile(){};

    //对外暴露的方法
    public static LogFile getInstance() {
        return LazyHolder.logFile;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        var s1 = LogFile.getInstance();
        var s2 = LogFile.getInstance();
        System.out.println(s1 == s2); //输出true，产生的是同一个对象
    }
}
```

静态内部类的效果是最好的，静态内部类只有在其成员变量或方法被引用时才会加载，也就是说只有当我们第一次访问类的时候实例才会被初始化完成，我们将实例委托给静态内部类帮忙初始化，虚拟机对静态内部类的加载是线程安全的，我们避免自己采用上锁机制而委托给虚拟机，这样的效率是非常高的。

懒汉式可以避免无用垃圾对象的产生——只有在它们使用时才初始化它，但我们也必须为此多编写一些代码来保证它的安全性，如果某个类并不是很常用的话，使用懒汉式可以一定程度的节约资源。

二、饿汉式

饿汉式模式在加载时便初始化单例，使得用户获取时实例已经被初始化。

```
//日志文件类
class LogFile {
    //唯一实例
    private static LogFile logFile = new LogFile();

    //构造方法对外隐藏
    private LogFile(){};

    //对外暴露的方法
    public static LogFile getInstance() {
        return logFile;
    }
}

public class Test {
    public static void main(String[] args) {
        var s1 = LogFile.getInstance();
        var s2 = LogFile.getInstance();
        System.out.println(s1 == s2); //输出true，产生的是同一个对象
    }
}
```

饿汉式是线程安全的，因为logFile已经被初始完成，因此饿汉式比懒汉式效率更高，但与此同时，如果实例在全局都没用上的话，饿汉式模式将会产生垃圾从而消耗资源。

优缺点总结

主要优点:

1. 提供了对唯一实例的受控访问。
2. 系统中内存只存在一个对象，节约系统的的资源。
3. 单例模式可以允许可变的数目的实例。

主要缺点:

1. 可扩展性比较差。
2. 单例类，职责过重，在一定程度上违背了"单一职责原则"。
3. 滥用单例将带来一些负面的问题，如为了节省资源将数据库连接池对象设计为单例类，可能会导致共享连接池对象的程序过多而出现的连接池溢出(大家都用一个池子，可能池子吃不消)，如果实例化对象长时间不用系统就会被认为垃圾对象被回收，这将导致对象状态丢失。