

# 没那么简单的单例模式

原创 小牛呼噜噜 小牛呼噜噜 2022-05-30 08:30

收录于合集  
#Java 进阶 14 #Java 21



微信扫一扫  
关注该公众号

- 什么是单例
- 单例的应用场景
- 单例的实现方式
  - 1. 懒汉式单例--简单版本
  - 2. 懒汉式单例 -- synchronized 版
  - 3. 懒汉式单例 -- 双重校验锁 synchronized版
  - 4. 懒汉式单例 -- 双重校验锁 volatile版
  - 5. 饿汉式单例
  - 6. 懒汉式单例--静态工厂版
  - 7. 枚举 实现单例
- 尾语
- 参考资料：



单例(Singleton)可以说是最简单的设计模式之一，而且基本上哪怕你没特别了解过，也能够随手写出，但是单例真有这么简单吗？



## 什么是单例

单例对象的类必须保证只有一个实例存在，自行提供这个实例，并向整个系统提供这个实例。上述定义总结以下特点大致有3点：

1. 单例类只有一个实例对象；
2. 该单例对象必须由单例类自行创建；
3. 单例类对外提供一个访问该单例的全局访问点。

## 单例的应用场景

单例模式的核心精髓其实是避免创建不必要的对象不必要的对象一般是：

1. 频繁创建的一些类，又频繁被销毁
2. “昂贵的对象”，有些对象创建的成本比其他对象要高得多，比如占用资源较多，或实例化耗时较长
3. 系统要求单一控制逻辑的操作，或者对象需要被共享的情况
4. ....

常见的使用场合：数据库的连接池、Spring中的全局访问点BeanFactory，Spring下的Bean、多线程的线程池、网络连接池等等单例模式的优点：

不仅可以减少每次创建对象的时间开销，还可以节约内存空间；能够避免由于操作多个实例导致的逻辑错误；如果一个对象有可能贯穿整个应用程序，能够起到了全局统一管理控制的作用。

缺点：



单例模式一般没有接口，没有抽象层，扩展困难。如果要扩展，得修改原来的代码单例模式的功能代码通常写在一个类中，其职责过重，如果功能设计不合理，则很容易违背单一职责原则不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。比如单例模式下去将对象转成json 会出现互相引用的问题。

## 单例的实现方式

对单例的实现一般可以分为两大类——懒汉式和饿汉式他们的区别在于：懒汉式：**全局的单例实例，默认不会实例化，直到首次使用时才实例化**，通俗点讲"一个懒汉,不愿意动弹。等到饭点了，他才开始想办法搞食物"

饿汉式：全局的单例实例在**类装载时**就实例化，并且创建单例对象。通俗点讲"一个饿汉,很勤快就怕自己饿着。总是先把食物准备好，等啥时候到饭点了，他随时拿来吃"

### 1. 懒汉式单例--简单版本

我们首先来写一个最简单的懒汉实现单例的方式：

```
/**
 * 懒汉 - 最简单的版本
 */
public class SingletonEasy {
    private static SingletonEasy instance;

    private SingletonEasy() {}//将构造器 私有化，防止外部调用

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

使用方式：**SingletonEasy singletonEasy = SingletonEasy.\_getInstance\_();** SingletonEasy的instance 默认为空，直到程序获取instance时，先进行判断instance 是否为空，如果instance 为空就new一个，反之直接返回已存在的instance 我们以这种方式实现的单例是**线程不安全的**，在大部分情况下是没问题的，但是当突然有一天有多个访问者（线程）同时去获取对象实例时，

```
if (instance == null) {
    instance = new SingletonEasy();
}
```

他们发现都不存在instance，然后就会导致 创建多个同样的实例的问题。那怎么解决这种问题呢？

### 2. 懒汉式单例 -- synchronized 版

其实遇到上面的问题，我们很容易想到一个解决方案 **加锁synchronized**

```
/**
 * 懒汉 - 加锁synchronized
 */
public class SingleSyn {
    private static SingleSyn instance;

    private SingleSyn() {}//将构造器 私有化，防止外部调用
}

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

加锁之后，如果有多个访问者（线程）访问getInstance()方法，当一个线程获得锁之后，进行**判空、对象创建、获得返回值**的操作，其他的线程必须等待其完成，才能继续执行这样加锁之后懒汉模式虽然解决了线程并发问题（**线程安全的**），但由于**把锁加到方法上**后，所有的访问都因需要锁占用导致资源的浪费，这其实非常影响程序的性能,效率很低。那我们可以怎样优化呢？

3. 懒汉式单例 -- 双重校验锁 synchronized版

```
/**
 * 懒汉 - 双层校验锁
 */
public class SingleDoubleCheck {

    private static SingleDoubleCheck instance = null;

    private SingleDoubleCheck(){}//将构造器 私有化，防止外部调用

    public static SingleDoubleCheck getInstance() {

        if (instance == null) { //part 1

            synchronized (SingleDoubleCheck.class) {

                if (instance == null) { //part 2

                    instance = new SingleDoubleCheck();//part 3

                }

            }

        }

        return instance;

    }

}
```

我们来仔细看下它的妙处：在多线程的环境下，当一个线程执行getInstance()时先判断单例对象是否已经初始化，如果已经初始化，就直接返回单例对象，如果未初始化，就在同步代码块中先进行初始化，然后返回，效率很高。

- 1. 在多线程的环境下，当一个线程执行getInstance()时
- 2. 程序到达part 1处的 if(instance == null) 先判断单例对象是否已经初始化，如果已经初始化，就直接返回单例对象，如果未初始化，则进入后续同步块逻辑；

此处 解决了 懒汉式单例 -- synchronized 版 的缺陷，不会影响到其他线程的getInstance()方法。

- 3. 程序进入同步块， 当一个线程获得锁之后，进行 判空(part2处的instance == null)、对象创建、获得返回值 的操作，其他的线程必须等待其完成，才能继续执行。

此处实现了 懒汉式单例 -- synchronized 版 的功能，保证了线程安全。这种写法，理论上既线程安全又效率高，可惜事实并非如此。问题出现在了 part 3处 instance = new SingleDoubleCheck(); 我们来看下整个类的字节码（JVM指令集）：

```
$ javap -c SingleDoubleCheck.class

Compiled from "SingleDoubleCheck.java"
public class com.zj.ideaprojects.test.SingleDoubleCheck {

    public static com.zj.ideaprojects.test.SingleDoubleCheck getInstance();

    Code:
        0: getstatic      #2                // Field instance:Lcom/zj/ideaprojects/test/SingleDoubleCheck;
        3: ifnonnull      37
        6: ldc           #3                // class com/zj/ideaprojects/test/SingleDoubleCheck
        8: dup
        9: astore_0
       10: monitorenter
       11: getstatic      #2                // Field instance:Lcom/zj/ideaprojects/test/SingleDoubleCheck;
       14: ifnonnull      27
       17: new           #3                // class com/zj/ideaprojects/test/SingleDoubleCheck
       20: dup
       21: invokespecial #4                // Method "<init>":()V
       24: putstatic      #2                // Field instance:Lcom/zj/ideaprojects/test/SingleDoubleCheck;
       27: aload_0
       28: monitorexit
       29: goto          37
       32: astore_1
       33: aload_0
       34: monitorexit
       35: aload_1
       36: athrow
       37: getstatic      #2                // Field instance:Lcom/zj/ideaprojects/test/SingleDoubleCheck;
       40: areturn

    Exception table:
        from    to      target type
         11     29      32     any
         32     35      32     any

    static {};

    Code:
        0: aconst_null
        1: putstatic      #2                // Field instance:Lcom/zj/ideaprojects/test/SingleDoubleCheck;
        4: return
}
```

内容比较多，我们直接看 instance = new SingleDoubleCheck() 相关的部分，可以发现在JVM字节码中 instance = new SingleDoubleCheck() 是有4个操作的

```
11: getstatic    #2          // 获取指定类的静态域instance 索引#2，并将其值压入栈顶
14: ifnonnull    27          // 不为空

17: new          #3          //1. 创建对象SingleDoubleCheck，并将对象引用压入栈
20: dup          //2. 将操作数栈顶的数据复制一份，并将其压入栈，此时栈中有两个引用值
21: invokespecial #4          //3. pop出栈引用值, 调用SingleDoubleCheck 其构造函数，完成对象的初始化
24: putstatic    #2          //4. SingleDoubleCheck对象指向指定类的静态域instance 索引#2
```

new指令并不能完全创建一个对象，对象只有在调用初始化方法完成后（即调用了 invokespecial指令之后），对象才创建成功。所以 `instance = new SingleDoubleCheck()` 并非一个原子操作（atomic）

原子操作就是不可分割的操作，在计算机中，就是指不会因为线程调度被打断的操作。

而在我们现代的计算机中CPU是乱序执行。CPU的速度是超级快的，但同时其价格也是非常昂贵的。为了"充分"压榨CPU，我们要把CPU的时间进行分片，让各个程序在CPU上轮转，造成一种多个程序同时在运行的假象，即并发。

并发是针对单核 CPU 提出的，而并行则是针对多核 CPU 提出的。和单核 CPU 不同，多核 CPU 真正实现了“同时执行多个任务”

在CPU中为了能够让指令的执行尽可能地同时运行起来，采用了指令流水线。一个 CPU 指令的执行过程可以分成 4 个阶段：取指、译码、执行、写回。这 4 个阶段分别由 4 个独立物理执行单元来完成。理想的情况是：指令之间无依赖，可以使流水线的并行度最大化但是如果两条指令的前后存在依赖关系，比如数据依赖，控制依赖等，此时后一条语句就必需等到前一条指令完成后，才能开始。所以CPU为了提高流水线的运行效率，对无依赖的前后指令做适当的乱序和调度

接着上面的内容，在生成字节码后，JVM 的编译器同样也会对其指令进行重排序的优化（指令重排）。

所谓指令重排是指在不改变原语义的情况下，通过调整指令的执行顺序让程序运行的更快。JVM中并没有规定编译器优化相关的内容，也就是说JVM可以自由的进行指令重排序的优化。

无论是编译期的指令重排还是\*\* CPU 的乱序执行\*\*，主要都是为了让 CPU 内部的指令流水线可以“填满”，提高指令执行的并行度。

指令重排 对于非原子性的操作，在不影响最终结果的情况下，其拆分成的原子操作可能会被重新排列执行顺序。`instance = new SingleDoubleCheck()` 的操作1234可能变成1243。这样会存在一个 `instance已经不为null但是SingleDoubleCheck仍没有完成初始化` 的状态这个时候其他的线程过来，走到 `part 1 if (instance == null)` 处时会产生：明明instance不为空，但是SingleDoubleCheck却没有的问题这种问题我们如何解决呢？

#### 4. 懒汉式单例 -- 双重校验锁 volatile版

不过好在JDK1.5及之后版本增加了volatile关键字。**volatile**保证该变量对所有线程的可见性,还有一个语义是禁止指令重排序优化，这样可以保证instance变量被赋值的时候对象已经是初始化完成的，从而避免了上面说到的问题。

```
/**
 * 懒汉 - 双层校验锁2
 */
public class SingleVolatile {
    private static volatile SingleVolatile instance;// 加上volatile关键字

    private SingleVolatile() {}//将构造器 私有化，防止外部调用

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

我们查看一下 这个文件的字节码：



```
$ javap -c SingleVolatile.class
Compiled from "SingleVolatile.java"
public class test.SingleVolatile {
    public static test.SingleVolatile getInstance();
    Code:
        0: getstatic   #2          // Field instance:Ltest/SingleVolatile;
        3: ifnonnull   37
        6: ldc         #3          // class test/SingleVolatile
        8: dup
        9: astore_0
       10: monitorenter
       11: getstatic   #2          // Field instance:Ltest/SingleVolatile;
       14: ifnonnull   27
       17: new         #3          // class test/SingleVolatile
       20: dup
       21: invokespecial #4          // Method "<init>":()V
       24: putstatic   #2          // Field instance:Ltest/SingleVolatile;
       27: aload_0
       28: monitorexit
       29: goto        37
       32: astore_1
       33: aload_0
       34: monitorexit
       35: aload_1
       36: athrow
       37: getstatic   #2          // Field instance:Ltest/SingleVolatile;
       40: areturn
    Exception table:
        from    to  target type
         11     29    32   any
         32     35    32   any

    public static void main(java.lang.String[]);
    Code:
        0: invokestatic #5          // Method getInstance():Ltest/SingleVolatile;
        3: pop
        4: return
}
```

可以看出和SingleDoubleCheck.class的字节码基本一模一样，看不出啥区别



那我们继续对SingleVolatile.class文件反汇编一下：

```
-server

-Xcomp

-XX:+UnlockDiagnosticVMOptions

-XX:+PrintAssembly

-XX:CompileCommand=compileonly,*SingleVolatile.getInstance

VM参数我贴了一下，大家感兴趣可以去试试
```

```
...
0x000001cdb13c7313: mov     dword ptr [r11+68h],r10d
0x000001cdb13c7317: mov     r10,76bf9bc68h ; {oop(a 'java/lang/Class' = 'test/SingleVolatile'
0x000001cdb13c7321: shr     r10,9h
0x000001cdb13c7325: mov     r11,1cbbd065000h
0x000001cdb13c732f: mov     byte ptr [r11+r10],r12l
0x000001cdb13c7333: lock add dword ptr [rsp],0h ;*putstatic instance
                                ; - test.SingleVolatile::getInstance@24 (line 13)

0x000001cdb13c7338: jmp     1cdb13c71e4h
0x000001cdb13c733d: mov     rdx,7c0060828h ; {metadata('test/SingleVolatile')}
...
```

汇编代码比较长，省略了很多，根据'putstatic'我们定位到第7行 0x000001cdb13c7333: lock add dword ptr [rsp],0h ;\*putstatic instance

我们再对SingleDoubleCheck.class 反汇编一下： VM参数：

```
-server

-Xcomp

-XX:+UnlockDiagnosticVMOptions

-XX:+PrintAssembly

-XX:CompileCommand=compileonly,*SingleDoubleCheck.getInstance
```

它的汇编代码,我们根据'putstatic'同样截取一段:

```
...
0x00000209690592e4: mov     rax,76bf9bd90h    ; {oop(a 'java/lang/Class' = 'test/SingleDoubleChe
0x00000209690592ee: mov     rsi,qword ptr [rsp+20h]
0x00000209690592f3: mov     r10,rsi
0x00000209690592f6: shr     r10,3h
0x00000209690592fa: mov     dword ptr [rax+68h],r10d
0x00000209690592fe: shr     rax,9h
0x0000020969059302: mov     rsi,20974cf5000h
0x000002096905930c: mov     byte ptr [rax+rsi],0h ;*putstatic instance
                                ; - test.SingleDoubleCheck::getInstance@24 (line 18)

0x0000020969059310: mov     rax,76bf9bd90h    ; {oop(a 'java/lang/Class' = 'test/SingleDoubleChe
0x000002096905931a: lea     rax,[rsp+28h]
0x000002096905931f: mov     rdi,qword ptr [rax+8h]
...
```

我们发现第9行 `0x000002096905930c: mov byte ptr [rax+rsi],0h ;*putstatic instance` 这个时候我们发现了区别,加了 "Volatile"关键字后, 汇编代码中 多了一个 `lock`, 其他的都是正常赋值的汇编语句我们知道在汇编中 LOCK指令前缀功能如下:

- 被修饰的汇编指令成为“原子的”
- 与被修饰的汇编指令一起提供内存屏障效果（LOCK指令可不是内存屏障，不能画等号哦）

内存屏障（Memory Barrier）这里就不展开说了，再说文章越写越多了，我们这里只要知道：它的几个作用：

1. 确保一些特定操作执行的顺序,让cpu必须按照顺序执行指令
2. 另一个作用是强制更新一次不同CPU的缓存，保证任何试图读取该数据的线程将得到会是最新值

instance声明为volatile之后，告诉JVM编译器**不允许指令重排优化**，告诉CPU**不允许乱序执行**。这样就保证new 对象等等过程中，一个写操作完成之前，不会调用读操作。这样避免了上面示例3中的说到的问题。

这样懒汉单例 就比较完美了，即保证了效率也是线程安全的。

### 5. 饿汉式单例

本文到现在一直介绍懒汉实现单例，我们来看下饿汉是怎么实现单例的

```
/**
 * 饿汉
 */
public class SingleHungry {
    private static SingleHungry instance = new SingleHungry();

    private SingleHungry() {
    }

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

这是 饿汉实现单例的标准写法，没啥大问题，线程安全的，执行效率高缺点：类加载时instance就初始化了，造成资源的浪费；开发者无法手动控制类实例化的时机

### 6. 懒汉式单例--静态工厂版

介绍一下 《Effective Java》第3版 给出的方法：

```
/**
 * 单例 - 静态工厂
 */
public class SingleStatic {
    private static class SingletonHolder{
        public static SingleStatic instance = new SingleStatic();
    }

    private SingleStatic(){}

    public static SingleStatic newInstance(){
        return SingletonHolder.instance;
    }
}
```

使用方式: `SingleStatic singleStatic = SingleStatic._newInstance_();` 我们来看下这种实现方法的巧妙之处:

- 从内部来看 对于**静态内部类**SingletonHolder，它是一个**饿汉式**的单例实现，在 SingletonHolder初始化的时候会由ClassLoader来保证同步，使INSTANCE是一个单例。
- 同时，由于SingletonHolder是一个内部类，只在外部类的Singleton的getInstance()中被使用，所以它被加载的时机也就是在getInstance()方法第一次被调用的时候。从外部看来，又的确是**懒汉式**的实现

使用类的静态内部类实现的单例模式，既保证了线程安全有保证了懒加载，同时不会因为加锁的方式耗费性能。推荐这种实现方法

## 7. 枚举 实现单例

最后再介绍一个《Effective Java》第3版推荐的写法

```
public enum Singleton {
    INSTANCE;
    public void funDo() {
        System.out.println("doSomething");
    }
}
```

使用方式：`Singleton.INSTANCE.funDo()` 这种方法充分 利用枚举的特性，让JVM来帮我们保证线程安全和单一实例的问题。除此之外，写法极其简洁。分外优雅！

### 尾语


虽然本文核心通篇是：单例可以 避免创建不必要的对象，减少每次创建对象的时间开销，还可以节约内存空间 这样可能会让一些人误以为：“JAVA创建对象的代价非常昂贵， 应该要 尽可能地避免创建对象”事实恰恰相反，由于小对象的构造器只做很少量的显式工作，所以小对象 的创建和回收动作是非常廉价的，特别是在现代的 JVM 实现上更是如此 通过创建附加的 对象，提升程序的清晰性、简洁性和功能性，所以通常是件好事 。

单例模式真的是最简单的设计模式吗？当我们去看其字节码、汇编是如何实现的原理时，往往发现其中细节无数充满前人的智慧结晶，平时我们日常学习中不能过于功利只盯着面试题去刷，也要深入底层去挖掘实现的细节和设计原理。感谢看您到最后。

### 参考资料：

- 《深入理解计算机系统》
- 《Effective Java》
- 《Java虚拟机规范》
- 《汇编语言》王爽
- <https://www.cnblogs.com/xrq730/p/7048693.html>
- <https://www.cnblogs.com/Mainz/p/3556430.html>
- <https://coolshell.cn/articles/265.html>
- <https://zhuanlan.zhihu.com/p/413889872>

本篇文章到这里就结束啦，如果喜欢的话，多多支持，欢迎关注！



帐号已迁移

1篇原创内容

>

公众号

收录于合集 #Java 进阶 14

< 上一篇

下一篇 >

深挖面向对象编程三大特性 --封装、继承、多态

"类加载器"与"双亲委派机制"一网打尽

[阅读原文](#)

喜欢此内容的人还喜欢

DeepFace-EMD: 一种大幅提升遮挡人脸识别准确度的好方法

Kiwi技术团队



第三章——进程之间的并发控制和死锁

Phoenix的狼窝



测试闪存微FCM32F103C8T6的USB-CDC功能，没想到竟如此简单

一起共享美好

