

## 26 | 单例模式：如何创建单一对象优化系统性能？

2019-07-23 刘超

Java性能调优实战

[进入课程 >](#)



讲述：李良

时长 09:35 大小 8.77M



你好，我是刘超。


从这一讲开始，我们将一起探讨设计模式的性能调优。在《Design Patterns: Elements of Reusable Object-Oriented Software》一书中，有 23 种设计模式的描述，其中，单例设计模式是最常用的设计模式之一。无论是在开源框架，还是在我们的日常开发中，单例模式几乎无处不在。

### 什么是单例模式？

它的核心在于，单例模式可以保证一个类仅创建一个实例，并提供一个访问它的全局访问点。

该模式有三个基本要点：一是这个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

结合这三点，我们来实现一个简单的单例：

 复制代码

```
1 // 饿汉模式
2 public final class Singleton {
3     private static Singleton instance=new Singleton();// 自行创建实例
4     private Singleton(){}// 构造函数
5     public static Singleton getInstance(){// 通过该函数向整个系统提供实例
6         return instance;
7     }
8 }
```

由于在一个系统中，一个类经常会被使用在不同的地方，**通过单例模式，我们可以避免多次创建多个实例，从而节约系统资源。**

## 饿汉模式

我们可以发现，以上第一种实现单例的代码中，使用了 static 修饰了成员变量 instance，所以该变量会在类初始化的过程中被收集进类构造器即 <clinit> 方法中。在多线程场景下，JVM 会保证只有一个线程能执行该类的 <clinit> 方法，其它线程将会被阻塞等待。

等到唯一的一次 <clinit> 方法执行完成，其它线程将不会再执行 <clinit> 方法，转而执行自己的代码。也就是说，static 修饰了成员变量 instance，在多线程的情况下能保证只实例化一次。

这种方式实现的单例模式，在类加载阶段就已经在堆内存中开辟了一块内存，用于存放实例化对象，所以也称为饿汉模式。


饿汉模式实现的单例的优点是，可以保证多线程情况下实例的唯一性，而且 getInstance 直接返回唯一实例，性能非常高。

然而，在类成员变量比较多，或变量比较大的情况下，这种模式可能会在没有使用类对象的情况下，一直占用堆内存。试想下，如果一个第三方开源框架中的类都是基于饿汉模式实现

的单例，这将会初始化所有单例类，无疑是灾难性的。

## 懒汉模式

懒汉模式就是为了避免直接加载类对象时提前创建对象的一种单例设计模式。该模式使用懒加载方式，只有当系统使用到类对象时，才会将实例加载到堆内存中。通过以下代码，我们可以简单地了解下懒加载的实现方式：


 复制代码

```
1 // 懒汉模式
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     private Singleton(){}// 构造函数
5     public static Singleton getInstance(){// 通过该函数向整个系统提供实例
6         if(null == instance){// 当 instance 为 null 时，则实例化对象，否则直接返回对象
7             instance = new Singleton();// 实例化对象
8         }
9         return instance;// 返回已存在的对象
10    }
11 }
```

以上代码在单线程下运行是没有问题的，但要运行在多线程下，就会出现实例化多个类对象的情况。这是怎么回事呢？

当线程 A 进入到 if 判断条件后，开始实例化对象，此时 instance 依然为 null；又有线程 B 进入到 if 判断条件中，之后也会通过条件判断，进入到方法里面创建一个实例对象。

所以我们需要对该方法进行加锁，保证多线程情况下仅创建一个实例。这里我们使用 Synchronized 同步锁来修饰 getInstance 方法：

 复制代码

```
1 // 懒汉模式 + synchronized 同步锁
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     private Singleton(){}// 构造函数
5     public static synchronized Singleton getInstance(){// 加同步锁，通过该函数向整个系统提
6         if(null == instance){// 当 instance 为 null 时，则实例化对象，否则直接返回对象
7             instance = new Singleton();// 实例化对象
8         }
9         return instance;// 返回已存在的对象
10    }
11 }
```

```
10     }
11 }
```


但我们前面讲过，同步锁会增加锁竞争，带来系统性能开销，从而导致系统性能下降，因此这种方式也会降低单例模式的性能。

还有，每次请求获取类对象时，都会通过 `getInstance()` 方法获取，除了第一次为 `null`，其它每次请求基本都是不为 `null` 的。在没有加同步锁之前，是因为 `if` 判断条件为 `null` 时，才导致创建了多个实例。基于以上两点，我们可以考虑将同步锁放在 `if` 条件里面，这样就可以减少同步锁资源竞争。

 复制代码

```
1 // 懒汉模式 + synchronized 同步锁
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     private Singleton(){}// 构造函数
5     public static Singleton getInstance(){// 加同步锁，通过该函数向整个系统提供实例
6         if(null == instance){// 当 instance 为 null 时，则实例化对象，否则直接返回对象
7             synchronized (Singleton.class){
8                 instance = new Singleton();// 实例化对象
9             }
10        }
11        return instance;// 返回已存在的对象
12    }
13 }
```

看到这里，你是不是觉得这样就可以了呢？答案是依然会创建多个实例。这是因为当多个线程进入到 `if` 判断条件里，虽然有同步锁，但是进入到判断条件里面的线程依然会依次获取到锁创建对象，然后再释放同步锁。所以我们还需要在同步锁里面再加一个判断条件：

 复制代码

```
1 // 懒汉模式 + synchronized 同步锁 + double-check
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     private Singleton(){}// 构造函数
5     public static Singleton getInstance(){// 加同步锁，通过该函数向整个系统提供实例
6         if(null == instance){// 第一次判断，当 instance 为 null 时，则实例化对象，否则直接返回
7             synchronized (Singleton.class){// 同步锁
8                 if(null == instance){// 第二次判断
9                     instance = new Singleton();// 实例化对象
10                }
11            }
12        }
13        return instance;
14    }
15 }
```

```


9         instance = new Singleton();// 实例化对象
10     }
11 }
12 }
13     return instance;// 返回已存在的对象
14 }
15 }

```

以上这种方式，通常被称为 Double-Check，它可以大大提高支持多线程的懒汉模式的运行性能。那这样做是不是就能保证万无一失了呢？还会有什么问题吗？

其实这里又跟 Happens-Before 规则和重排序扯上关系了，这里我们先来简单了解下 Happens-Before 规则和重排序。

我们在第二期[加餐](#)中分享过，编译器为了尽可能地减少寄存器的读取、存储次数，会充分复用寄存器的存储值，比如以下代码，如果没有进行重排序优化，正常的执行顺序是步骤 1\2\3，而在编译期间进行了重排序优化之后，执行的步骤有可能就变成了步骤 1/3/2，这样就能减少一次寄存器的存取次数。

 复制代码

```

1 int a = 1;// 步骤 1: 加载 a 变量的内存地址到寄存器中，加载 1 到寄存器中，CPU 通过 mov 指令把
2 int b = 2;// 步骤 2 加载 b 变量的内存地址到寄存器中，加载 2 到寄存器中，CPU 通过 mov 指令把 2
3 a = a + 1;// 步骤 3 重新加载 a 变量的内存地址到寄存器中，加载 1 到寄存器中，CPU 通过 mov 指令

```

在 JMM 中，重排序是十分重要的一环，特别是在并发编程中。如果 JVM 可以对它们进行任意排序以提高程序性能，也可能会给并发编程带来一系列的问题。例如，我上面讲到的 Double-Check 的单例问题，假设类中有其它的属性也需要实例化，这个时候，除了要实例化单例类本身，还需要对其它属性也进行实例化：

 复制代码

```

1 // 懒汉模式 + synchronized 同步锁 + double-check
2 public final class Singleton {
3     private static Singleton instance= null;// 不实例化
4     public List<String> list = null;//list 属性
5     private Singleton(){
6         list = new ArrayList<String>();
7     }// 构造函数

```

```

8      public static Singleton getInstance(){// 加同步锁，通过该函数向整个系统提供实例
9          if(null == instance){// 第一次判断，当 instance 为 null 时，则实例化对象，否则直接返回
10             synchronized (Singleton.class){// 同步锁
11                 if(null == instance){// 第二次判断
12                     instance = new Singleton();// 实例化对象
13                 }
14             }
15         }
16         return instance;// 返回已存在的对象
17     }
18 }

```

在执行 `instance = new Singleton();` 代码时，正常情况下，实例过程这样的：

给 Singleton 分配内存；

调用 Singleton 的构造函数来初始化成员变量；

将 Singleton 对象指向分配的内存空间（执行完这步 singleton 就为非 null 了）。

如果虚拟机发生了重排序优化，这个时候步骤 3 可能发生在步骤 2 之前。如果初始化线程刚好完成步骤 3，而步骤 2 没有进行时，则刚好有另一个线程到了第一次判断，这个时候判断为非 null，并返回对象使用，这个时候实际没有完成其它属性的构造，因此使用这个属性就很可能导致异常。在这里，Synchronized 只能保证可见性、原子性，无法保证执行的顺序。

这个时候，就体现出 Happens-Before 规则的重要性了。通过字面意思，你可能会误以为是前一个操作发生在后一个操作之前。然而真正的意思是，前一个操作的结果可以被后续的操作获取。这条规则规范了编译器对程序的重排序优化。

我们知道 volatile 关键字可以保证线程间变量的可见性，简单地说就是当线程 A 对变量 X 进行修改后，在线程 A 后面执行的其它线程就能看到变量 X 的变动。除此之外，volatile 在 JDK1.5 之后还有一个作用就是阻止局部重排序的发生，也就是说，volatile 变量的操作指令都不会被重排序。所以使用 volatile 修饰 instance 之后，Double-Check 懒汉单例模式就万无一失了。

 复制代码

```

1 // 懒汉模式 + synchronized 同步锁 + double-check
2 public final class Singleton {

```



```

3     private volatile static Singleton instance= null;// 不实例化
4     public List<String> list = null;//list 属性
5     private Singleton(){
6         list = new ArrayList<String>();
7     }// 构造函数
8     public static Singleton getInstance(){// 加同步锁，通过该函数向整个系统提供实例
9         if(null == instance){// 第一次判断，当 instance 为 null 时，则实例化对象，否则直接返回
10             synchronized (Singleton.class){// 同步锁
11                 if(null == instance){// 第二次判断
12                     instance = new Singleton();// 实例化对象
13                 }
14             }
15         }
16         return instance;// 返回已存在的对象
17     }
18 }


```

## 通过内部类实现

以上这种同步锁 + Double-Check 的实现方式相对来说，复杂且加了同步锁，那有没有稍微简单一点儿的可以实现线程安全的懒加载方式呢？

我们知道，在饿汉模式中，我们使用了 static 修饰了成员变量 instance，所以该变量会在类初始化的过程中被收集进类构造器即 <clinit> 方法中。在多线程场景下，JVM 会保证只有一个线程能执行该类的 <clinit> 方法，其它线程将会被阻塞等待。这种方式可以保证内存的可见性、顺序性以及原子性。

如果我们在 Singleton 类中创建一个内部类来实现成员变量的初始化，则可以避免多线程下重复创建对象的情况发生。这种方式，只有在第一次调用 getInstance() 方法时，才会加载 InnerSingleton 类，而只有在加载 InnerSingleton 类之后，才会实例化创建对象。具体实现如下：

 复制代码

```

1 // 懒汉模式 内部类实现
2 public final class Singleton {
3     public List<String> list = null;// list 属性
4
5     private Singleton() { // 构造函数
6         list = new ArrayList<String>();
7     }
8
9     // 内部类实现

```

```
10     public static class InnerSingleton {
11         private static Singleton instance=new Singleton();// 自行创建实例
12     }
13
14     public static Singleton getInstance() {
15         return InnerSingleton.instance;// 返回内部类中的静态变量
16     }
17 }
```

## 总结

单例的实现方式其实有很多，但总结起来就两种：饿汉模式和懒汉模式，我们可以根据自己的需求来做选择。

如果我们在程序启动后，一定会加载到类，那么用饿汉模式实现的单例简单又实用；如果我们是写一些工具类，则优先考虑使用懒汉模式，因为很多项目可能会引用到 jar 包，但未必会使用到这个工具类，懒汉模式实现的单例可以避免提前被加载到内存中，占用系统资源。

## 思考题

除了以上那些实现单例的方式，你还知道其它实现方式吗？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。



# Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超

金山软件西山居技术经理



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 答疑课堂：模块四热点问题解答

下一篇 27 | 原型模式与享元模式：提升系统性能的利器

## 精选留言 (16)

 写留言



豆泥丸

2019-07-23

最安全的枚举模式，反射和序列化都是单例。

展开

作者回复: 对的，我们之前序列化优化这一讲中的问答题就是与枚举实现单例相关，《Effective Java》作者也是强烈推荐枚举方式实现单例。



9



Loubobooo

2019-07-23

使用枚举来实现单例模式，具体代码如下：`public class SingletonExample5 {  
 private static SingletonExample5 instance = null;`

```
// 私有构造函数  
private SingletonExample5(){...
```

展开 ▾

作者回复: 很赞！这是一种懒加载模式的枚举实现。



7



**行者**

2019-07-23

枚举也是一种单例模式，同时是饿汉式。

相比Double Check，以内部类方式实现单例模式，代码简洁，性能相近，在我看来是更优的选择。

展开 ▾

作者回复: 也可以使用枚举实现懒汉模式，可以根据本讲中的使用内部类方式实现懒加载。



3



**-W.LI-**

2019-07-23

枚举底层实现就是静态内部类吧

展开 ▾

作者回复: 对的，枚举是一种语法糖，在Java编译后，枚举类中的枚举会被声明为static，接下来就跟我们文中讲的一样了。



3



**我又不乱来**

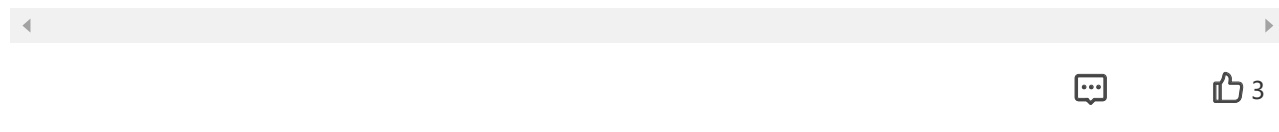
2019-07-23

枚举天生就是单例，但是不清楚这么实现。

注册式单例，spring应该是用的这种。这个也不太清楚，超哥有机会讲一下spring的实现

方式和枚举方式实现的单例。谢谢🙏

作者回复: Spring中的bean的单例虽然是一种单例效果，但实现方式是通过容器缓存实现，严格来说是一种享元模式。



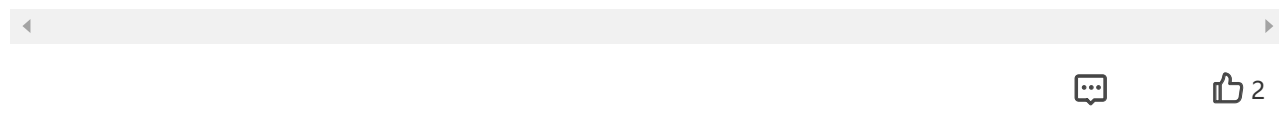
**我知道了**

2019-07-23

枚举实现单例

展开 ▾

作者回复: 对的

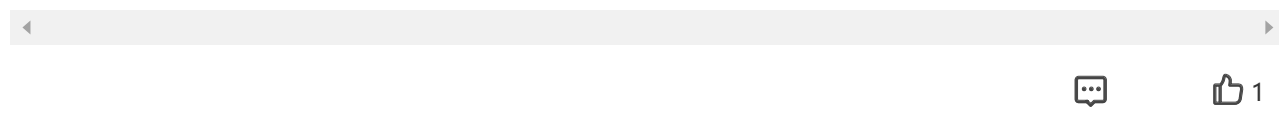


**QQ怪**

2019-07-23

这一节虽然都懂，但是评论区补充的我还是第一次见到，get到了，有收获，哈哈

作者回复: 互相学习，共同进步



**Aaron**

2019-07-28

谢谢超哥分享，写了这么多代码，也看了不少博客也写了不少博客，今天算是彻底搞懂了，很多时候都是觉得似乎懂了，自己做发现还有点模糊，工作忙反正平时都那么写没深究。get到了很多干活。谢谢🙏



**WL**

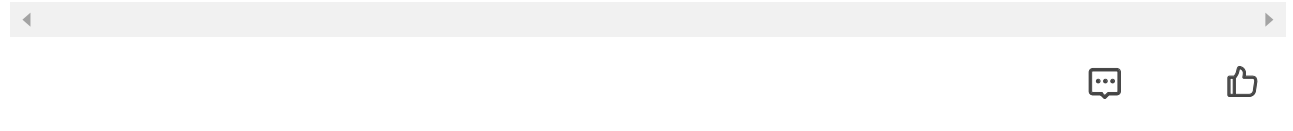
2019-07-24

老师请问您讲的单例模式三个基本要点,但是我使用spring的框架并里面默认类不都是单

例的吗,但是并没有满足您说的要点, 用tomcat也是, tomcat里面的servlet应该也是单例的吧, 好像也没有满足您说的三个要点, 请问spring和tomcat是咋实现的, 是数据懒汉还是饿汉的实现方式.

展开 ▾

作者回复: Spring是通过容器管理来实现单例的, 也是基于这三个基本点。



**colin**

2019-07-24

看评论涨知识

展开 ▾



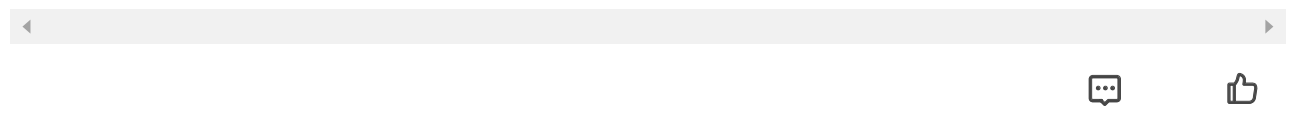
**Jxin**

2019-07-23

1.可能大部分同学都知道, 但为了少部分同学, 我在老师这个单例上补个点。其它线程空指针异常确实是指令重排导致的, 但其原因还有一个。加锁并不能阻止cpu调度线程执行体, 所以时间片还是会切的(假设单核), 所以其他线程依旧会执行锁外层的if(), 并发情况下就可能拿到仅赋值引用, 未在内存空间存储数据的实例(null实例), 进而空指针。  
2.给老师的代码补段骚的: ...

展开 ▾

作者回复: 虽然有点绕, 还是值得表扬的。我们还是鼓励简单易懂的编程风格。



**码德纽@宝**

2019-07-23

枚举模式, 之外还有CAS方式单例模式

展开 ▾



**Zed**

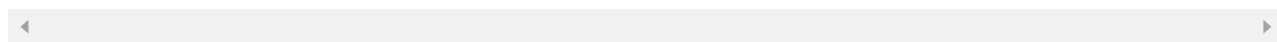
2019-07-23

## 容器类管理

```
class InstanceManager {  
    private static Map<String, Object> objectMap = new HashMap<>();  
    private InstanceManager(){}...
```

展开 ▾

作者回复: Spring中bean的单例就是使用容器来实现的，便于管理。



1

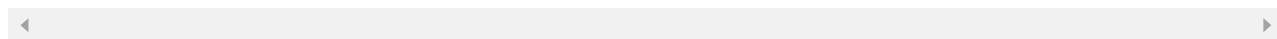


**OMT**

2019-07-23

看过其他文章有分析单例反射和序列化问题。  
枚举单例可以反编译查看。

作者回复: 对的，在第9讲中，我们的问答题也是关于枚举实现单例来解决序列化问题。



**计科一班**

2019-07-23

枚举单例

展开 ▾



**nightmare**

2019-07-23

666

展开 ▾

