


41 | 单例模式（上）：为什么说支持懒加载的双重检测不比饿汉式更优？

王争 · 设计模式之美



单例模式（上）

从今天开始，我们正式进入到设计模式的学习。我们知道，经典的设计模式有 23 种。其中，常用的并不是很多。据我的工作经验来看，常用的可能都不到一半。如果随便抓一个程序员，让他说一说最熟悉的 3 种设计模式，那其中肯定会包含今天要讲的单例模式。

网上有很多讲解单例模式的文章，但大部分都侧重讲解，如何实现一个线程安全的单例。我今天也会讲到各种单例的实现方法，但是，这并不是我们专栏学习的重点，我重点还是希望带你搞清楚下面这样几个问题（第一个问题会在今天讲解，后面三个问题放到下一节课中讲解）。

为什么要使用单例？

单例存在哪些问题？

单例与静态类的区别？

有何替代的解决方案？

话不多说，让我们带着这些问题，正式开始今天的学习吧！


为什么要使用单例？

单例设计模式（Singleton Design Pattern）理解起来非常简单。一个类只允许创建一个对象（或者实例），那这个类就是一个单例类，这种设计模式就叫作单例设计模式，简称单例模式。

对于单例的概念，我觉得没必要解释太多，你一看就能明白。我们重点看一下，为什么我们需要单例这种设计模式？它能解决哪些问题？接下来我通过两个实战案例来讲解。

实战案例一：处理资源访问冲突

我们先来看第一个例子。在这个例子中，我们自定义实现了一个往文件中打印日志的 Logger 类。具体的代码实现如下所示：

 复制代码

```
1 public class Logger {
2     private FileWriter writer;
3
4     public Logger() {
5         File file = new File("/Users/wangzheng/log.txt");
6         writer = new FileWriter(file, true); //true表示追加写入
7     }
8
9     public void log(String message) {
10        writer.write(message);
11    }
12 }
13
14 // Logger类的应用示例：
15 public class UserController {
16     private Logger logger = new Logger();
17
18     public void login(String username, String password) {
19         // ...省略业务逻辑代码...
20         logger.log(username + " logged!");
21     }
22 }
```

```

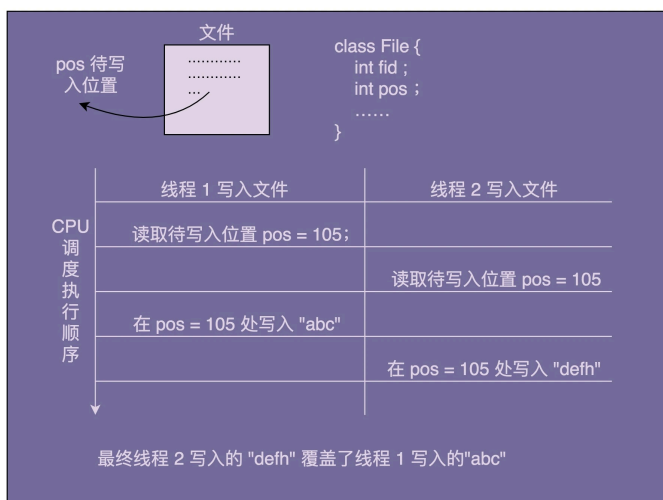
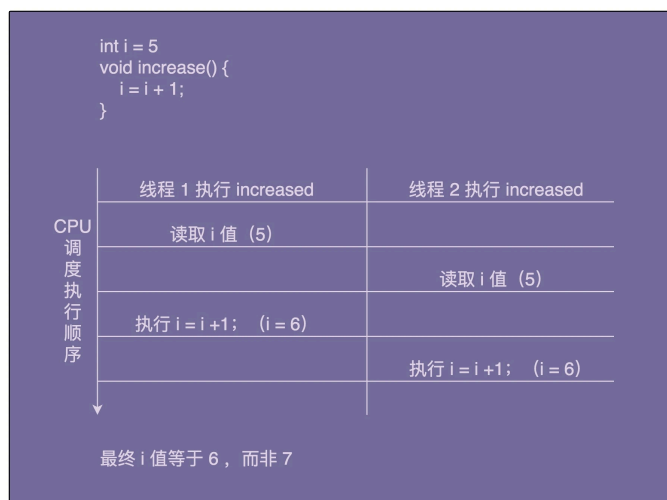
22 }
23
24 public class OrderController {
25     private Logger logger = new Logger();
26
27     public void create(OrderVo order) {
28         // ...省略业务逻辑代码...
29         logger.log("Created an order: " + order.toString());
30     }
31 }

```

看完代码之后，先别着急看我下面的讲解，你可以先思考一下，这段代码存在什么问题。


在上面的代码中，我们注意到，所有的日志都写入到同一个文件 `/Users/wangzheng/log.txt` 中。在 `UserController` 和 `OrderController` 中，我们分别创建两个 `Logger` 对象。在 Web 容器的 Servlet 多线程环境下，如果两个 Servlet 线程同时分别执行 `login()` 和 `create()` 两个函数，并且同时写日志到 `log.txt` 文件中，那就有可能存在日志信息互相覆盖的情况。

为什么会出现互相覆盖呢？我们可以这么类比着理解。在多线程环境下，如果两个线程同时给同一个共享变量加 1，因为共享变量是竞争资源，所以，共享变量最后的结果有可能并不是加了 2，而是只加了 1。同理，这里的 `log.txt` 文件也是竞争资源，两个线程同时往里面写数据，就有可能存在互相覆盖的情况。



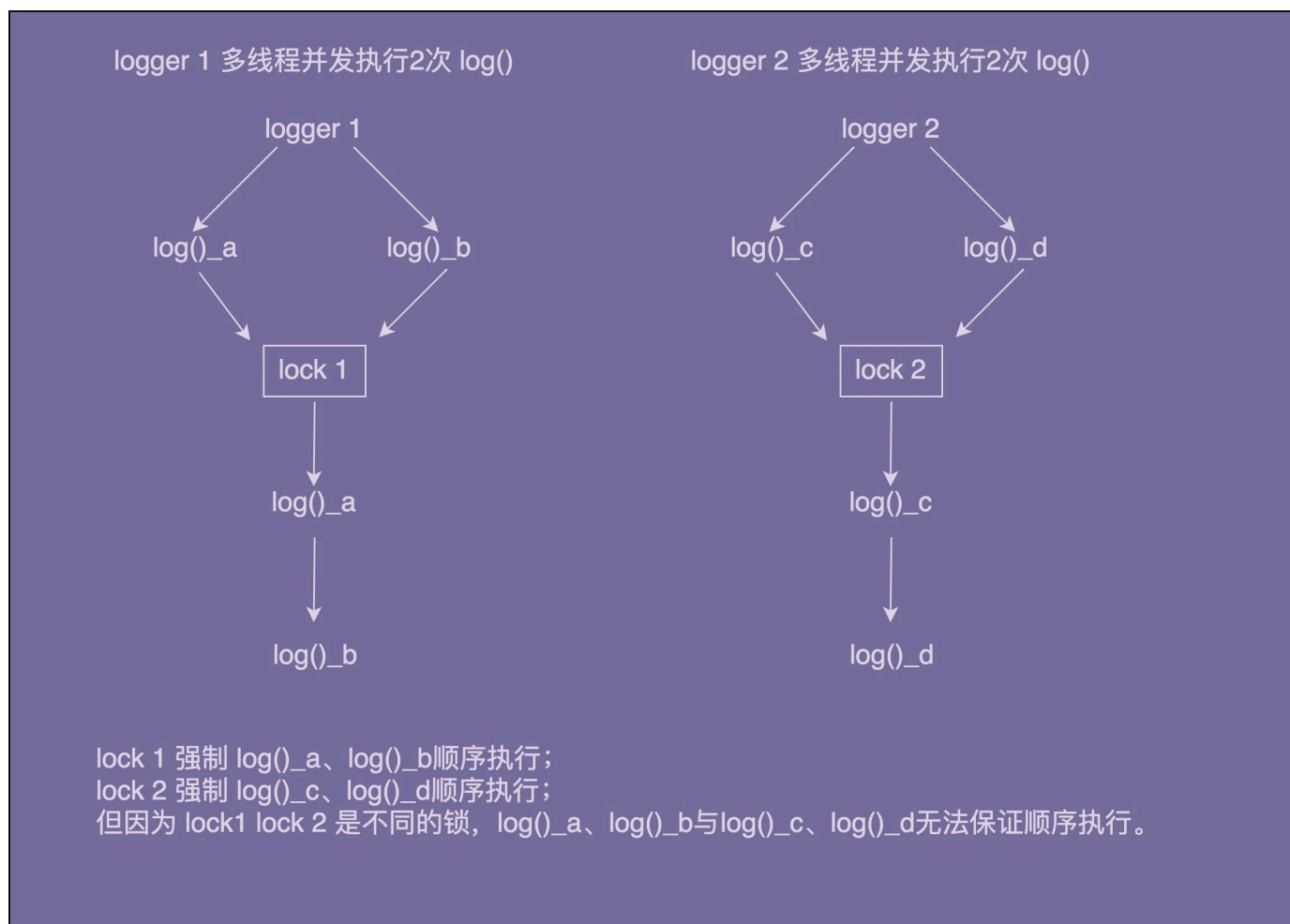
那如何来解决这个问题呢？我们最先想到的就是通过加锁的方式：给 `log()` 函数加互斥锁（Java 中可以通过 `synchronized` 的关键字），同一时刻只允许一个线程调用执行 `log()` 函

数。具体的代码实现如下所示：

 复制代码

```
1 public class Logger {
2     private FileWriter writer;
3
4     public Logger() {
5         File file = new File("/Users/wangzheng/log.txt");
6         writer = new FileWriter(file, true); //true表示追加写入
7     }
8
9     public void log(String message) {
10        synchronized(this) {
11            writer.write(mesasge);
12        }
13    }
14 }
```

不过，你仔细想想，这真的能解决多线程写入日志时互相覆盖的问题吗？答案是否定的。这是因为，这种锁是一个对象级别的锁，一个对象在不同的线程下同时调用 `log()` 函数，会被强制要求顺序执行。但是，不同的对象之间并不共享同一把锁。在不同的线程下，通过不同的对象调用执行 `log()` 函数，锁并不会起作用，仍然有可能存在写入日志互相覆盖的问题。



我这里稍微补充一下，在刚刚的讲解和给出的代码中，我故意“隐瞒”了一个事实：我们给 log() 函数加不加对象级别的锁，其实都没有关系。因为 FileWriter 本身就是线程安全的，它的内部实现中本身就加了对象级别的锁，因此，在外层调用 write() 函数的时候，再加对象级别的锁实际上是多此一举。因为不同的 Logger 对象不共享 FileWriter 对象，所以，FileWriter 对象级别的锁也解决不了数据写入互相覆盖的问题。

那我们该怎么解决这个问题呢？实际上，要想解决这个问题也不难，我们只需要把对象级别的锁，换成类级别的锁就可以了。让所有的对象都共享同一把锁。这样就避免了不同对象之间同时调用 log() 函数，而导致的日志覆盖问题。具体的代码实现如下所示：

```
1 public class Logger {
2     private FileWriter writer;
3 }
```

复制代码

```

4     public Logger() {
5         File file = new File("/Users/wangzheng/log.txt");
6         writer = new FileWriter(file, true); //true表示追加写入
7     }
8
9     public void log(String message) {
10        synchronized(Logger.class) { // 类级别的锁
11            writer.write(mesasge);
12        }
13    }
14 }


```

除了使用类级别锁之外，实际上，解决资源竞争问题的办法还有很多，分布式锁是最常听到的一种解决方案。不过，实现一个安全可靠、无 bug、高性能的分布式锁，并不是件容易的事情。除此之外，并发队列（比如 Java 中的 BlockingQueue）也可以解决这个问题：多个线程同时往并发队列里写日志，一个单独的线程负责将并发队列中的数据，写入到日志文件。这种方式实现起来也稍微有点复杂。

相对于这两种解决方案，单例模式的解决思路就简单一些了。单例模式相对于之前类级别锁的好处是，不用创建那么多 Logger 对象，一方面节省内存空间，另一方面节省系统文件句柄（对于操作系统来说，文件句柄也是一种资源，不能随便浪费）。

我们将 Logger 设计成一个单例类，程序中只允许创建一个 Logger 对象，所有的线程共享使用的这一个 Logger 对象，共享一个 FileWriter 对象，而 FileWriter 本身是对象级别线程安全的，也就避免了多线程情况下写日志会互相覆盖的问题。

按照这个设计思路，我们实现了 Logger 单例类。具体代码如下所示：

 复制代码

```

1     public class Logger {
2         private FileWriter writer;
3         private static final Logger instance = new Logger();
4
5         private Logger() {
6             File file = new File("/Users/wangzheng/log.txt");
7             writer = new FileWriter(file, true); //true表示追加写入
8         }
9
10        public static Logger getInstance() {

```

```

11     return instance;
12 }
13
14 public void log(String message) {
15     writer.write(mesasge);
16 }
17 }
18
19 // Logger类的应用示例:
20 public class UserController {
21     public void login(String username, String password) {
22         // ...省略业务逻辑代码...
23         Logger.getInstance().log(username + " logged!");
24     }
25 }
26
27 public class OrderController {
28     public void create(OrderVo order) {
29         // ...省略业务逻辑代码...
30         Logger.getInstance().log("Created a order: " + order.toString());
31     }
32 }


```

实战案例二：表示全局唯一类

从业务概念上，如果有些数据在系统中只应保存一份，那就比较适合设计为单例类。

比如，配置信息类。在系统中，我们只有一个配置文件，当配置文件被加载到内存之后，以对象的形式存在，也理所应当只有一份。

再比如，唯一递增 ID 号码生成器（[第 34 讲](#)中我们讲的是唯一 ID 生成器，这里讲的是唯一递增 ID 生成器），如果程序中有两个对象，那就会存在生成重复 ID 的情况，所以，我们应该将 ID 生成器类设计为单例。

 复制代码

```

1 import java.util.concurrent.atomic.AtomicLong;
2 public class IdGenerator {
3     // AtomicLong是一个Java并发库中提供的一个原子变量类型，
4     // 它将一些线程不安全需要加锁的复合操作封装为了线程安全的原子操作，
5     // 比如下面会用到的incrementAndGet()。
6     private AtomicLong id = new AtomicLong(0);

```



```
7     private static final IdGenerator instance = new IdGenerator();
8     private IdGenerator() {}
9     public static IdGenerator getInstance() {
10         return instance;
11     }
12     public long getId() {
13         return id.incrementAndGet();
14     }
15 }
16
17 // IdGenerator使用举例
18 long id = IdGenerator.getInstance().getId();
```

实际上，今天讲到的两个代码实例（Logger、IdGenerator），设计的都并不优雅，还存在一些问题。至于有什么问题以及如何改造，今天我暂时卖个关子，下一节课我会详细讲解。

如何实现一个单例？

尽管介绍如何实现一个单例模式的文章已经有很多了，但为了保证内容的完整性，我这里还是简单介绍一下几种经典实现方式。概括起来，要实现一个单例，我们需要关注的点无外乎下面几个：

构造函数需要是 private 访问权限的，这样才能避免外部通过 new 创建实例；

考虑对象创建时的线程安全问题；

考虑是否支持延迟加载；

考虑 getInstance() 性能是否高（是否加锁）。

如果你对这块已经很熟悉了，你可以当作复习。注意，下面的几种单例实现方式是针对 Java 语言语法的，如果你熟悉的是其他语言，不妨对比 Java 的这几种实现方式，自己试着总结一下，利用你熟悉的语言，该如何实现。

1. 饿汉式

饿汉式的实现方式比较简单。在类加载的时候，instance 静态实例就已经创建并初始化好了，所以，instance 实例的创建过程是线程安全的。不过，这样的实现方式不支持延迟加载

（在真正用到 IdGenerator 的时候，再创建实例），从名字中我们也可以看出这一点。具体的代码实现如下所示：

 复制代码

```
1 public class IdGenerator {
2     private AtomicLong id = new AtomicLong(0);
3     private static final IdGenerator instance = new IdGenerator();
4     private IdGenerator() {}
5     public static IdGenerator getInstance() {
6         return instance;
7     }
8     public long getId() {
9         return id.incrementAndGet();
10    }
11 }
```

有人觉得这种实现方式不好，因为不支持延迟加载，如果实例占用资源多（比如占用内存多）或初始化耗时长（比如需要加载各种配置文件），提前初始化实例是一种浪费资源的行为。最好的方法应该在用到的时候再去初始化。不过，我个人并不认同这样的观点。

如果初始化耗时长，那我们最好不要等到真正要用它的时候，才去执行这个耗时长的初始化过程，这会影响到系统的性能（比如，在响应客户端接口请求的时候，做这个初始化操作，会导致此请求的响应时间变长，甚至超时）。采用饿汉式实现方式，将耗时的初始化操作，提前到程序启动的时候完成，这样就能避免在程序运行的时候，再去初始化导致的性能问题。

如果实例占用资源多，按照 fail-fast 的设计原则（有问题及早暴露），那我们也希望在程序启动时就将这个实例初始化好。如果资源不够，就会在程序启动的时候触发报错（比如 Java 中的 PermGen Space OOM），我们可以立即去修复。这样也能避免在程序运行一段时间后，突然因为初始化这个实例占用资源过多，导致系统崩溃，影响系统的可用性。

2. 懒汉式

有饿汉式，对应的，就有懒汉式。懒汉式相对于饿汉式的优势是支持延迟加载。具体的代码实现如下所示：

```
1 public class IdGenerator {
2     private AtomicLong id = new AtomicLong(0);
3     private static IdGenerator instance;
4     private IdGenerator() {}
5     public static synchronized IdGenerator getInstance() {
6         if (instance == null) {
7             instance = new IdGenerator();
8         }
9         return instance;
10    }
11    public long getId() {
12        return id.incrementAndGet();
13    }
14 }
```

不过懒汉式的缺点也很明显，我们给 `getInstance()` 这个方法加了一把大锁

(`synchronized`)，导致这个函数的并发度很低。量化一下的话，并发度是 1，也就相当于串行操作了。而这个函数是在单例使用期间，一直会被调用。如果这个单例类偶尔会被用到，那种实现方式还可以接受。但是，如果频繁地用到，那频繁加锁、释放锁及并发度低等问题，会导致性能瓶颈，这种实现方式就不可取了。

3. 双重检测

饿汉式不支持延迟加载，懒汉式有性能问题，不支持高并发。那我们再来看一种既支持延迟加载、又支持高并发的单例实现方式，也就是双重检测实现方式。

在这种实现方式中，只要 `instance` 被创建之后，即便再调用 `getInstance()` 函数也不会再进入到加锁逻辑中了。所以，这种实现方式解决了懒汉式并发度低的问题。具体的代码实现如下所示：

```
1 public class IdGenerator {
2     private AtomicLong id = new AtomicLong(0);
3     private static IdGenerator instance;
4     private IdGenerator() {}
5     public static IdGenerator getInstance() {
6         if (instance == null) {
```

```
7     synchronized(IdGenerator.class) { // 此处为类级别的锁
8         if (instance == null) {
9             instance = new IdGenerator();
10        }
11    }
12 }
13 return instance;
14 }
15 public long getId() {
16     return id.incrementAndGet();
17 }
18 }
```

实际上，上述实现方式存在问题：CPU 指令重排序可能导致在 `IdGenerator` 类的对象被关键字 `new` 创建并赋值给 `instance` 之后，还没来得及初始化（执行构造函数中的代码逻辑），就被另一个线程使用了。这样，另一个线程就使用了一个没有完整初始化的 `IdGenerator` 类的对象。要解决这个问题，我们只需要给 `instance` 成员变量添加 `volatile` 关键字来禁止指令重排序即可。

4. 静态内部类

我们再来看一种比双重检测更加简单的实现方法，那就是利用 Java 的静态内部类。它有点类似饿汉式，但又能做到了延迟加载。具体是怎么做到的呢？我们先来看它的代码实现。

```
1 public class IdGenerator {
2     private AtomicLong id = new AtomicLong(0);
3     private IdGenerator() {}
4
5     private static class SingletonHolder{
6         private static final IdGenerator instance = new IdGenerator();
7     }
8
9     public static IdGenerator getInstance() {
10         return SingletonHolder.instance;
11     }
12
13     public long getId() {
14         return id.incrementAndGet();
15     }
16 }
```

SingletonHolder 是一个静态内部类，当外部类 IdGenerator 被加载的时候，并不会创建 SingletonHolder 实例对象。只有当调用 getInstance() 方法时，SingletonHolder 才会被加载，这个时候才会创建 instance。instance 的唯一性、创建过程的线程安全性，都由 JVM 来保证。所以，这种实现方法既保证了线程安全，又能做到延迟加载。

5. 枚举

最后，我们介绍一种最简单的实现方式，基于枚举类型的单例实现。这种实现方式通过 Java 枚举类型本身的特性，保证了实例创建的线程安全性和实例的唯一性。具体的代码如下所示：

```
1 public enum IdGenerator {
2     INSTANCE;
3     private AtomicLong id = new AtomicLong(0);
4
5     public long getId() {
6         return id.incrementAndGet();
7     }
8 }
```

重点回顾

好了，今天的内容到此就讲完了。我们来总结回顾一下，你需要掌握的重点内容。

1. 单例的定义

单例设计模式（Singleton Design Pattern）理解起来非常简单。一个类只允许创建一个对象（或者叫实例），那这个类就是一个单例类，这种设计模式就叫作单例设计模式，简称单例模式。

2. 单例的用处

从业务概念上，有些数据在系统中只应该保存一份，就比较适合设计为单例类。比如，系统的配置信息类。除此之外，我们还可以使用单例解决资源访问冲突的问题。

3. 单例的实现

单例有下面几种经典的实现方式。

饿汉式

饿汉式的实现方式，在类加载的期间，就已经将 instance 静态实例初始化好了，所以，instance 实例的创建是线程安全的。不过，这样的实现方式不支持延迟加载实例。

懒汉式

懒汉式相对于饿汉式的优势是支持延迟加载。这种实现方式会导致频繁加锁、释放锁，以及并发度低等问题，频繁的调用会产生性能瓶颈。

双重检测

双重检测实现方式既支持延迟加载、又支持高并发的单例实现方式。只要 instance 被创建之后，再调用 getInstance() 函数都不会进入到加锁逻辑中。所以，这种实现方式解决了懒汉式

并发度低的问题。

静态内部类

利用 Java 的静态内部类来实现单例。这种实现方式，既支持延迟加载，也支持高并发，实现起来也比双重检测简单。

枚举

最简单的实现方式，基于枚举类型的单例实现。这种实现方式通过 Java 枚举类型本身的特性，保证了实例创建的线程安全性和实例的唯一性。

课堂讨论

1. 在你所熟悉的编程语言的类库中，有哪些类是单例类？又为什么要设计成单例类呢？
2. 在第一个实战案例中，除了我们讲到的类级别锁、分布式锁、并发队列、单例模式等解决方案之外，实际上还有一种非常简单的解决日志互相覆盖问题的方法，你想到了吗？

可以在留言区说一说，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

单例模式是一种常用的设计模式，它保证一个类只能创建一个实例。本文通过实例和讲解深入浅出地介绍了单例模式的应用和实现方式。文章首先介绍了为什么需要使用单例模式，通过实战案例展示了单例模式的应用场景和解决问题的能力。其中，通过Logger类的例子说明了在多线程环境下，使用单例模式可以避免资源访问冲突的问题。另外，通过IdGenerator类的例子展示了单例模式在表示全局唯一类时的应用。文章还介绍了如何实现一个单例，包括构造函数的私有访问权限、线程安全、延迟加载和性能高效等方面的考虑。单例的实现方式包括饿汉式、懒汉式、双重检测、静态内部类和枚举。每种实现方式都有其优势和适用场景，读者可以根据具体需求选择合适的方式。总的来说，本文通过实例和讲解，对读者理解和掌握单例模式具有一定的指导意义。

© 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。



西南偏北 置顶

2020-02-05

这真的是看过的关于讲单例的最好的文章

共 13 条评论 >

👍 329



KK

2020-04-05

饿汉式和懒汉式的名字为什么这么起呀？可以解释一下吗？

作者回复: 着急吃-》饿汉式

不着急吃-》懒汉式

共 3 条评论 >

👍 8



星之所在

2020-06-10

争哥我想细问下，我用一个静态变量也可以实现单例的效果，为啥还要用单例设计模式？是为了代码后续扩展性，还是静态变量用多了影响整个代码？

作者回复: 静态变量没法替代单例啊。单例是类本身不允许多个实例。但是静态变量，我可以定义多个啊，这个怎么解决呢？



👍 6



_Walker

2020-05-05

最后枚举那个解释有些含糊其辞呀😂要是能详细解释一下就好了

作者回复: 不是重点,你自己研究研究吧😂



👍 3



西柚

2020-06-08

老师讲的太好了，逻辑清晰、缜密。思考问题的方式非常值得学习~

作者回复: 信小争哥就对了



👍 2



skying

2020-07-11

争哥，你好！

我这边想模拟出 你文章中 的Logger类写入文件会重复的场景。

但没复现出来，

不知道你这边有 样例代码没有。

作者回复: 比较难模拟出来的，因为毕竟要并发极端情况下（有竞争的情况下，就像我画的那张图一样）才会发生覆盖的情况。我的样例都在文章里了。

共 2 条评论 >

👍 1



子夜2104

2020-05-19

我们现在用的高版本的 Java 已经在 JDK 内部实现中解决了这个问题（解决的方法很简单，只要把对象 new 操作和初始化操作设计为原子操作，就自然能禁止重排序）。

老师，请问是哪个版本解决了这个问题呢？

作者回复: 好像jdk5之后就解决了，我有点记不清了😂

共 8 条评论 >

👍 1



鹤涵

2020-11-26

1. Spring中的一些连接工厂类，Service类都默认是单例模式 这些对象是一般消耗资源或者类似于工具类没有共享变量竞争问题。
2. FileWriter设计成 static final的可以使用jvm类加载特性解决竞争问题。但是可测试性变差

作者回复: 嗯嗯 🤔🤔🤔🤔🤔🤔





西门吹牛

2020-07-01

其实有一点不太理解，希望老师解答。

双层检查，加volatile，根据java内存模型，volatile保证的是可见性，也就是说，给变量赋值的操作，会被另一线程看到，这样，另一线程拿到的还是地址，这时候，内存一定初始化完成了吗？

是不是可以理解为，这个volatile 的写操作 happen-before 与后续的读操作，就相当于于是，从语言层面考虑，而不是指令层面的。

如果从指令层面考虑，这个new操作，会有三条指令，赋值完成就相当与写操作完成，对象还没初始化完成，这时候别的线程读到还没初始化的地址值会报空指针异常；

如果从语言层面考虑，这个new语句，相当于写操作完成，就代表这个操作对应的所有指令都完成了，所以后续能读到已经初始化的值。

这个happen-before规则是从语言层面考虑还是指令层面？

作者回复: 我建议你去看下极客的并发专栏😂

共 2 条评论 >



Douglas

2020-02-05

争哥新年好， 有个问题想请教一下，单例的实现中看到过一种实现方式，包括在spring源码中有类似的实现，代码如下

```
1. public class Singleton {
    private static volatile Singleton instance=null;
    private Singleton() {
    }

    public static Singleton getInstance() {
        Singleton temp=instance; // 为什么要用局部变量来接收
        if (null == temp) {
            synchronized (Singleton.class) {
                temp=instance;
                if (null == temp) {
                    temp=new Singleton();
                    instance=temp;
                }
            }
        }
    }
}
```

```
        return instance;
    }
}
```

spring源码 如 ReactiveAdapterRegistry。

JDK 源码 如 AbstractQueuedSynchronizer。

很多地方 都有用 局部变量 来接收 静态的成员变量， 请问下 这么写有什么性能上的优化点吗？

jcu 包下面类似的用法太多。想弄明白为什么要这样写

2. 看jdk 官方的文档（JMM）有说明 指令重排发生的地方有很多，编译器，及时编译，CPU 在硬件层面的优化，看spring 比较新的代码也使用volatile来修饰，你说的new 关键字和初始化 作为原子操作 可以说一下 大概的jdk版本吗

共 42 条评论 >

 116