单例模式

> 创建型设计模式：单例模式
>
> 要点：保证一个类只有一个实例，并且提供一个访问它的全局访问点

大致原理：

使用

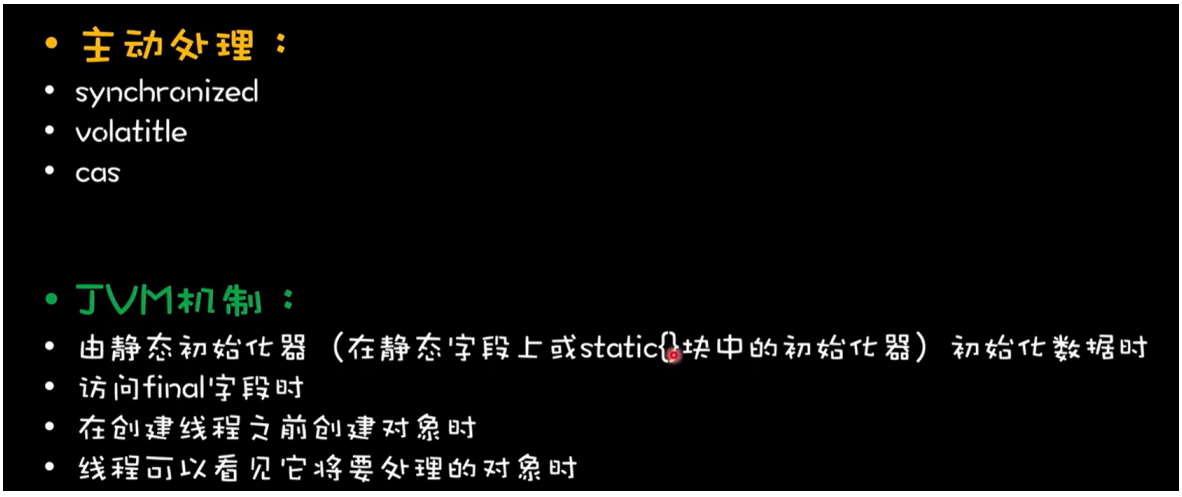
- 私有构造函数
- 私有静态变量
- 公有静态函数

私有构造函数保证了不能通过构造函数来创建对象，只能通过公有静态函数返回唯一私有静态变量

原则：

1. 私有构造 （防止类通过常规方法实例化）
2. 以静态方法或者枚举返回实例。 （保证实例的唯一性）
3. 确保实例只有一个，尤其是多线程环境。（保证创建实例的线程安全）
4. 确保反序列化时不会重新构造对象。（在有序列化反序列化的场景下防止单例被莫名破坏）

单例模式手段：



> 讲讲为什么要使用static和final，参看：这里

static可以保证在一个线程未使用其他同步机制的情况下总是可以读到一个类的静态变量的初始值。

我们都知道，final修饰的变量值不会改变。但是在多线程的环境中，它还会保证两点，**1. 其他线程所看到的final字段必然是初始化完毕的。 2. final修饰的变量不会被程序重排序。**

static保证了变量的初始值，final保证了不被JIT编译器重排序。对于一个单例模式来说，它所在的类在被引用的时候，static会保证它被初始化完毕，且是所有线程所见的初始化，final保证了实例初始化过程的顺寻性。两者结合保证了这个实例创建的唯一性。

案例场景：

1. 数据库连接池不会反复创建
2. spring中一个单例模式bean的生成和使用
3. 在我们平常的代码中需要设置全局的的一些属性保存

六种实现方式

总结：

应用场景	实现方式	原理	优点	缺点	备注
• 初始化时就需创建单例 • 单例对象 要求初始化速度快 & 占用内存小	1. 饿汉式	依赖JVM类加载机制，保证单例只被创建1次	• 线程安全 (即多线程下适用，因为 JVM 只会加载1次单例类) • 初始化速度快 • 占用内存小	单例创建时机不可控制	饿汉式 与 懒汉式最大区别 = 单例创建时机 • 饿汉式：单例创建时机不可控，即类加载时 自动创建 单例 • 懒汉式：单例创建时机可控，即有需要时，才 手动创建 单例
	2. 枚举类型	• 枚举类型 = 不可被继承的类（final） • 每个枚举元素 = 类静态常量 = 依赖JVM类加载机制，保证单例只被创建1次 • 枚举元素 都通过静态代码块来进行初始化 • 构造方法 访问权限 默认 = 私有（private） • 大部分方法都是final	• 线程安全 • 自由序列化 • 实现更加简单、简洁		
• 按需、延迟创建单例 • 单例初始化的操作耗时长 & 应用要求启动速度快 • 单例的占用内存比较大	3. 懒汉式 (基础实现)	1. 类加载时，先不自动创建单例 (即，将单例的引用先赋值为 Null) 2. 需要时才手动 创建 单例	• 按需加载单例 • 节约资源	线程不安全 (即多线程下不适用)	
	4. 同步锁 (懒汉式的改进)	使用同步锁 synchronized 锁住 创建单例的方法 (防止多个线程同时调用，从而避免造成单例被多次创建)	• 线程安全	造成过多的同步开销 (每次访问都要进行线程同步，加锁 = 耗时、耗能)	
	5. 双重校验锁 (懒汉式的改进)	• 校验锁1：若单例已创建，则直接返回已创建的单例，无需再执行加锁操作 • 校验锁2：防止多次创建单例问题	• 线程安全 • 节省资源（不需过多的同步开销）	实现复杂 (多种判断，易出错)	
	6. 静态内部类	• 按需加载：在静态内部类里创建单例，在装载该内部类时才会去创建单例 • 线程安全：类是由 JVM加载，而JVM只会加载1遍，保证只有1个单例	• 线程安全 • 节省资源（不需过多的同步开销） • 实现简单		

静态类调用

```
public class Singleton_OO {

    public static Map<String,String> cache = new ConcurrentHashMap<String,
String>();

}
```

- 以上这种方式在我们平常的业务开发中非常场常见，这样静态类的方式可以在第一次运行的时候直 接初始化Map类，同时这里我们也不需要到**延迟加载**在使用。 （延迟加载：是指类在需要用到的时候才创建，节省内存空间）
- 在不需要维持任何状态下，仅仅用于全局访问，这个使用使用静态类的方式更加方便。
- 但如果需要被继承以及需要维持一些特定状态的情况下，就适合使用单例模式。

懒汉式（线程不安全）

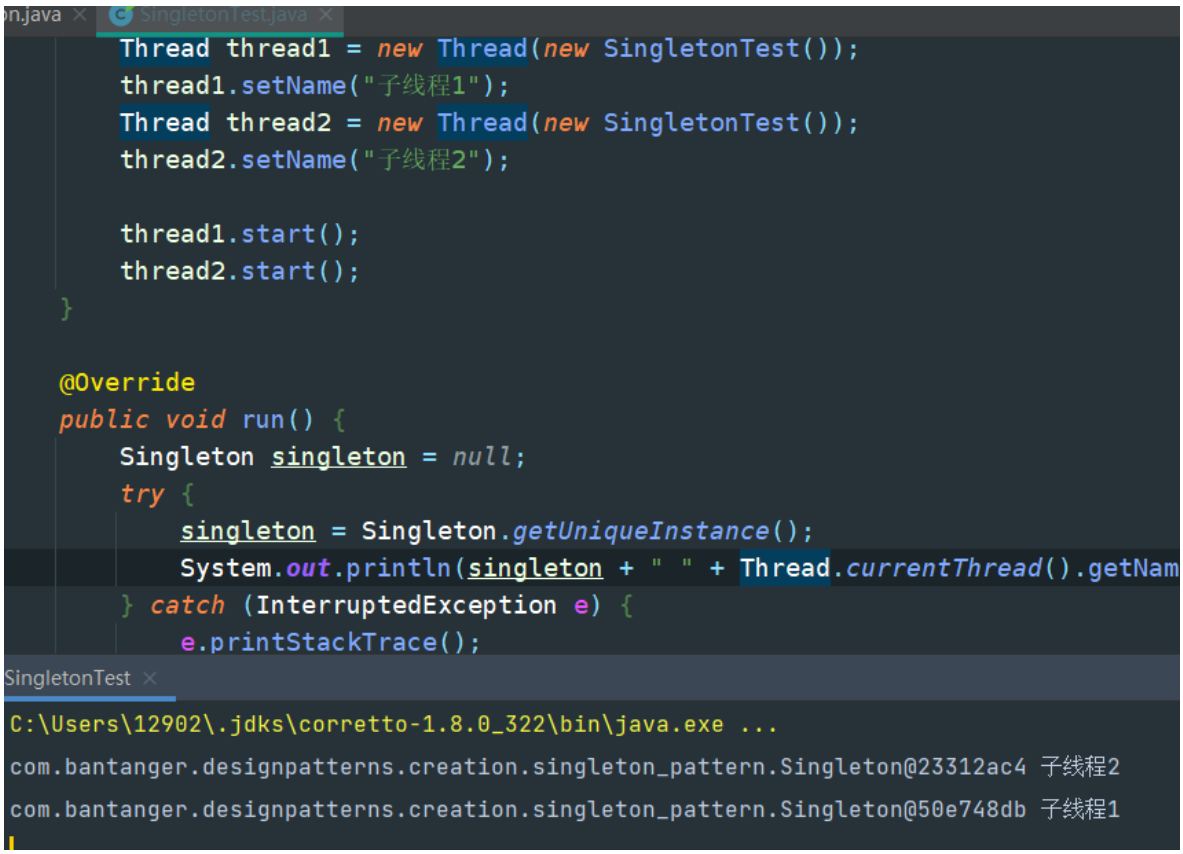
```
/**
 * 懒汉式
 */
public class Singleton {
```

```
5
6       private static Singleton uniqueInstance; // 唯一对象实例
7
8       private Singleton() { // 私有构造函数
9       }
10
11      public static Singleton getUniqueInstance () {
12          if (uniqueInstance == null) {
13              uniqueInstance = new Singleton();
14          }
15          return uniqueInstance;
16      }
17
18  }
```

```
/** 懒汉式 */
public class Singleton {

    private static Singleton uniqueInstance; // 唯一对象实例

    private Singleton() { // 私有构造函数
    }

    public static Singleton getUniqueInstance () {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

}
```

以下实现中，私有静态变量 uniqueInstance 被延迟实例化，这样做的好处是，如果没有用到该类，那么就不会实例化 uniqueInstance，从而节约资源。

这个实现在**多线程环境下是不安全**的，如果多个线程能够同时进入 `if (uniqueInstance == null)`，并且此时 uniqueInstance 为 null，那么会有**多个线程执行** `uniqueInstance = new Singleton();` 语句，这将导致多次实例化 uniqueInstance。



懒汉模式下线程不安全的情况，出现多个实例

懒汉模式（线程安全）

```java
public class Singleton_02 {

    private static Singleton_02 instance;
    private Singleton_02() {
    }

    public static synchronized Singleton_02 getInstance(){
         if (null != instance) return instance;
         instance = new Singleton_02();
         return instance;
    }
}
```

```java
public static synchronized Singleton_02 getInstance(){
     if (null != instance) return instance;
     instance = new Singleton_02();
     return instance;
}
```

使用synchronized同步锁实现线程安全

- 这种模式虽然是安全的，但是由于吧锁加到方法上后，所有的访问都需要锁占用导致资源浪费，如果不是特殊情况，不建议使用

饿汉式（线程安全）

```
public class Singleton_03 {

    private static Singleton_03 instance = new Singleton_03();

    private Singleton_03() {
    }

    public static Singleton_03 getInstance() {
        return instance;
    }

}
```

- 此种方式与我们开头的第一个实例化 Map 基本一致，在程序启动的时候直接运行加载，后续有外 部需要使用的时候获取即可。
- 但此种方式并不是懒加载，也就是说无论你程序中是否用到这样的类都会在程序启动之初进行创 建。
- 那么这种方式导致的问题就像你下载个游戏软件，可能你游戏地图还没有打开呢，但是程序已经将 这些地图全部实例化。到你手机上最明显体验就一开游戏内存满了，手机卡了，需要换了。

使用类的内部类（线程安全）

```
public class Singleton_04 {

    private static class SingletonHolder {
        private static Singleton_04 instance = new Singleton_04();
    }

    private Singleton_04() {
    }

    public static Singleton_04 getInstance() {
        return SingletonHolder.instance;
    }

}
```

- 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。
- 这主要是因为JVM虚拟机可以保证多线程并发访问的正确性，也就是**一个类的构造方法在多线程环境下可以被正确的加载。**
- 此种方式也是非常推荐使用的一种单例模式

双重校验锁-线程安全

uniqueInstance 只需要被实例化一次，之后就可以直接使用了。加锁操作只需要对实例化那部分的代码进行，只有当 uniqueInstance 没有被实例化时，才需要进行加锁。

双重校验锁先判断 uniqueInstance 是否已经被实例化，如果没有被实例化，那么才对实例化语句进行加锁。

```
public class Singleton {

    private volatile static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

考虑下面的实现，也就是只使用了一个 if 语句。在 uniqueInstance == null 的情况下，如果两个线程同时执行 if 语句，那么两个线程就会同时进入 if 语句块内。虽然在 if 语句块内有加锁操作，但是两个线程都会执行 `uniqueInstance = new Singleton();` 这条语句，只是先后的问题，那么就会进行两次实例化，从而产生了两个实例。因此必须使用双重校验锁，也就是需要使用两个 if 语句。

```
if (uniqueInstance == null) {
    synchronized (Singleton.class) {
        uniqueInstance = new Singleton();
    }
}
```

uniqueInstance 采用 volatile 关键字修饰也是很有必要的。`uniqueInstance = new Singleton();` 这段代码其实是分为三步执行。

1. 分配内存空间
2. 初始化对象
3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，有可能执行顺序变为了 1>3>2，这在单线程情况下自然是没有问题。但如果是多线程下，有可能获得是一个还没有被初始化的实例，以致于程序出错。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

> 为什么需要volatile修饰？（以下内容参看java并发编程）

双重锁并不安全，指令可能重排，导致成员域没有初始化，只是成员变量指向了分配好的内存地址

3.8.2 问题的根源

前面的双重检查锁定示例代码的第 7 行（instance = new Singleton();）创建了一个对象。这一行代码可以分解为如下的 3 行伪代码。

```
memory = allocate();      //1：分配对象的内存空间
ctorInstance(memory);     //2：初始化对象
instance = memory;        //3：设置 instance 指向刚分配的内存地址
```

上面 3 行伪代码中的 2 和 3 之间，可能会被重排序（在一些 JIT 编译器上，这种重排序是真实发生的，详情见参考文献 1 的“Out-of-order writes”部分）。2 和 3 之间重排序之后的执行时序如下。

```
memory = allocate();      //1：分配对象的内存空间
instance = memory;        //3：设置 instance 指向刚分配的内存地址
                          //注意，此时对象还没有被初始化！
ctorInstance(memory);     //2：初始化对象
```

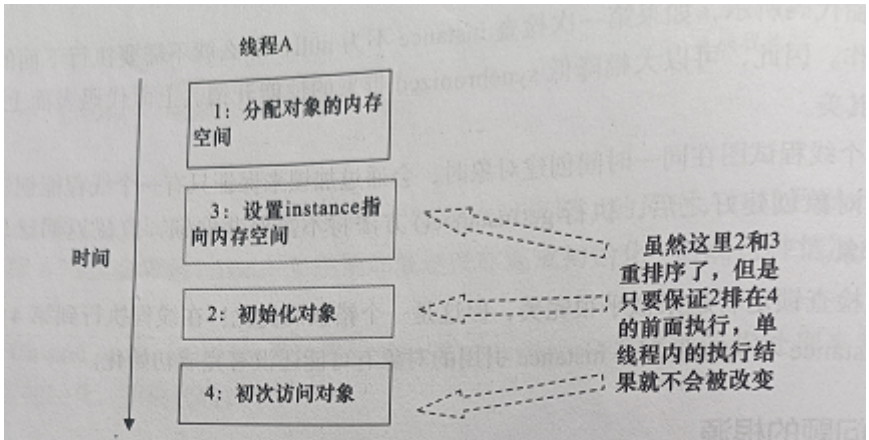


图 3-37　线程执行时序图

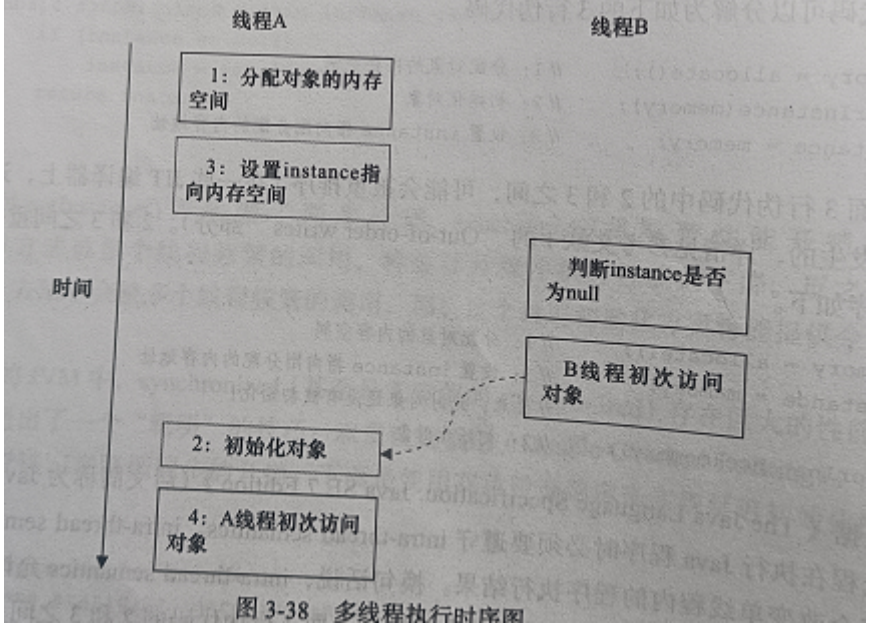


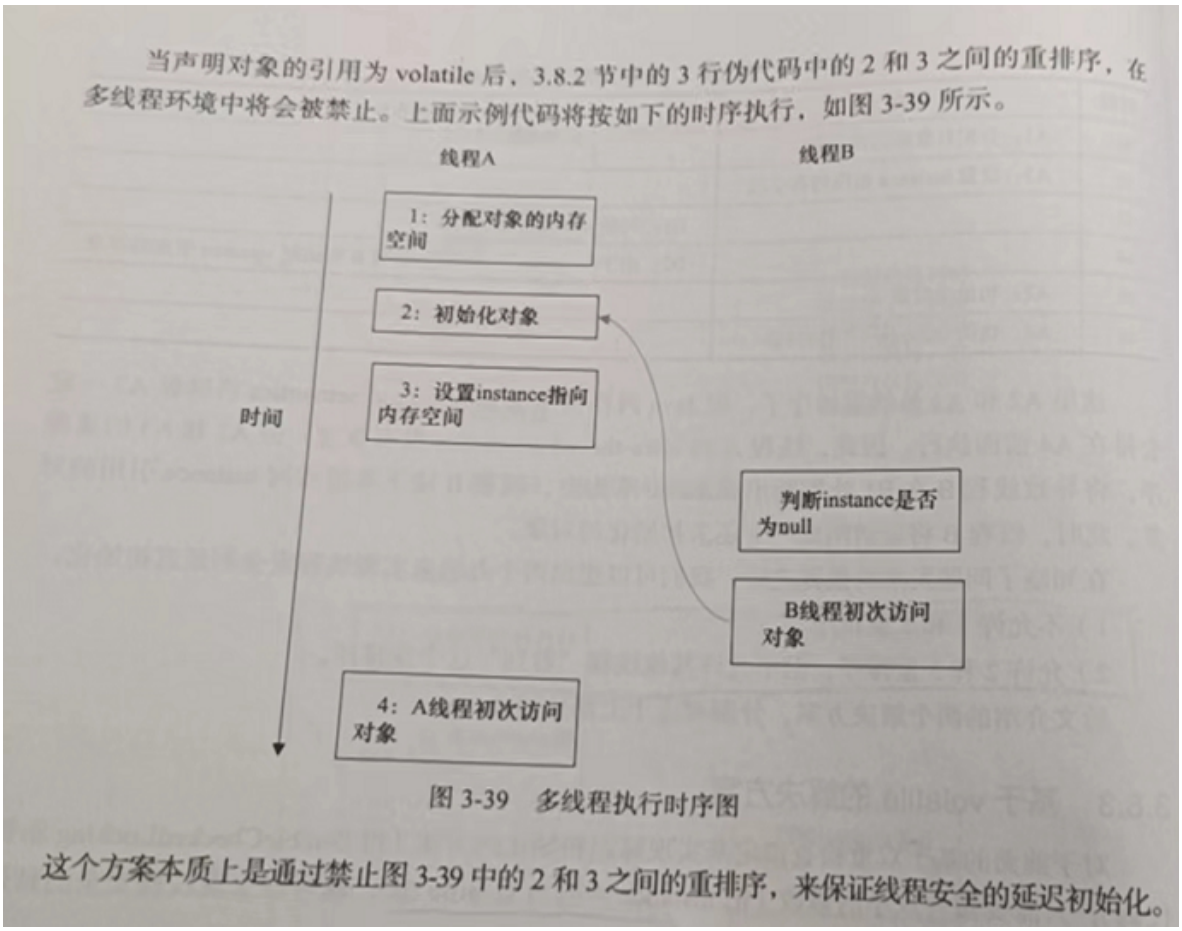
图 3-38　多线程执行时序图

两种解决方法：

1. 不允许指令重排（2，3指令）
2. 允许指令重排，但是不允许其他线程看到这个重排

基于volatile的解决方案

也就是上面代码例子中使用的手段，通过给成员变量添加volatile修饰，从而禁止分配内存指令和初始化指令重排



基于类初始化的解决方案

详细见下面👇

静态内部类实现

当 Singleton 类加载时，静态内部类 SingletonHolder 没有被加载进内存。只有当调用 `getUniqueInstance()` 方法从而触发 `SingletonHolder.INSTANCE` 时 SingletonHolder 才会被加载，此时初始化 INSTANCE 实例。

这种方式不仅具有延迟初始化的好处，而且由虚拟机提供了对线程安全的支持。

```java
public class Singleton {

    private Singleton() {
    }

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getUniqueInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

缺点：不能传参

> 下面来分析虚拟机怎么提供线程安全支持的，参看java并发艺术

JVM在类初始化阶段（Class被加载，且被线程使用之前）会执行类的初始化，在这个期间，JVM会获得一个锁，这个锁可以**同步**多个线程对同一个类的初始化（意思就是一个类**只能由一个线程**去初始化）

当一个线程争抢到锁以后，其余线程因为没有抢到锁而waitSet，也就无法看到类初始化的指令，避免多个线程初始化一个类

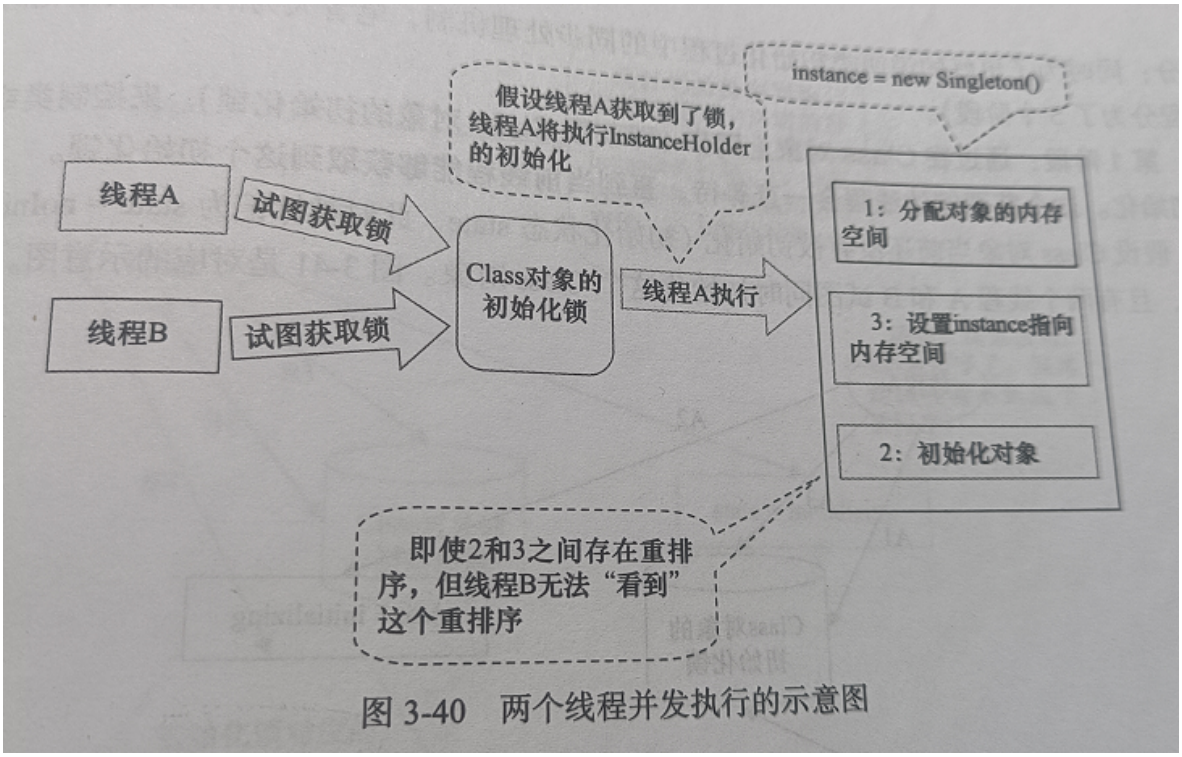


图 3-40　两个线程并发执行的示意图

这里提一嘴，为什么要有类锁呢：本质上是**防止多个线程同时初始化一个类**，导致前一个类被后一个线程初始化的类覆盖，只能由一个线程去初始化，也就是避免重复初始化

对于`<clinit> ()`方法的调用，虚拟机会自己确保其在多线程环境中的安全性。因为`<clinit> ()`方法是带锁线程安全，所以在多线程环境下进行类初始化的话可能会引起多个线程阻塞，并且这种阻塞很难被发现。

虚拟机保证一个类的clinit()在多线程环境中被正确加锁和同步,如果多个线程同时去初始化一个类,只会有一个线程执行,其他阻塞等待,当执行完后,其他线程不会在执行clinit.也就是说clinit会加锁执行,且只会执行一次.可以利用这个特性实现单例模式

枚举类型实现

单例模式最佳实践，实现简单，并且能面对复杂序列化或者反射攻击时，防止实例化多次

```
public enum Singleton {
    uniqueInstance;
}
```

优点：线程安全，自由串行化，单一实例

缺点：存在继承场景下是不能使用的

考虑以下单例模式的实现，该 Singleton 在每次序列化的时候都会创建一个新的实例，为了保证只创建一个实例，必须声明所有字段都是 transient，并且提供一个 readResolve() 方法。

```
public class Singleton implements Serializable {

    private static Singleton uniqueInstance;

    private Singleton() {
    }

    public static synchronized Singleton getUniqueInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

如果不用枚举类型实现单例模式，会出现反射攻击，因为通过setAccessible() 方法可以将私有构造函数的访问权限从private修改成public，然后调用构造函数从而实例化对象，如果要防止这样的攻击，需要在构造函数中添加实例化第二个对象的代码

从上面的讨论可以看出，解决序列化和反射攻击很麻烦，而枚举实现不会出现这两种问题，所以说枚举实现单例模式是最佳实践。

> 讲讲为什么枚举类可以这么轻易做到其他类做不到的事~~(DIO狂喜~~

```
public enum Singleton {
    uniqueInstance;
}
```

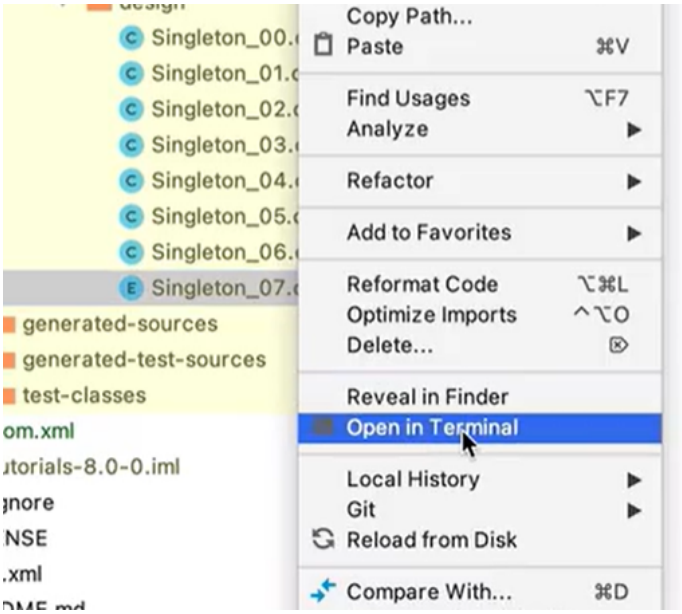
实际上，在Jvm内部的样子是这样的👇

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E>,
Serializable
```

它继承了Enum，进而能方便的得知实际类型，同时实现了序列化接口，能自由串行化

反编译这个类：

操作指令：




```
javap -c Singleon_07.class
```

```
Terminal:  Local  Local (2)  Local (3)  +
fuzhengwei@MacBook-Pro design % ls
Singleton_00.class                  Singleton_03.class                  Singleton_04.class                  Singleton_07.clas
Singleton_01.class                  Singleton_04$1.class                Singleton_05.class
Singleton_02.class                  Singleton_04$SingletonHolder.class  Singleton_06.class
fuzhengwei@MacBook-Pro design % javap -c Singleton_07.class
Compiled from "Singleton_07.java"
public final class cn.bugstack.design.Singleton_07 extends java.lang.Enum<cn.bugstack.design.Singleton_07> {
  public static final cn.bugstack.design.Singleton_07 INSTANCE;

  public static cn.bugstack.design.Singleton_07[] values();

fuzhengwei@MacBook-Pro design % javap -c Singleton_07.class
Compiled from "Singleton_07.java"
public final class cn.bugstack.design.Singleton_07 extends java.lang.Enum<cn.bugstack.design.Singleton_07> {
  public static final cn.bugstack.design.Singleton_07 INSTANCE;

  public static cn.bugstack.design.Singleton_07[] values();
    Code:
       0: getstatic     #1                  // Field $VALUES:[Lcn/bugstack/design/Singleton_07;
       3: invokevirtual #2                  // Method "[Lcn/bugstack/design/Singleton_07;".clone:()Ljava/lang/Object;
       6: checkcast     #3                  // class "[Lcn/bugstack/design/Singleton_07;"
       9: areturn

  public static cn.bugstack.design.Singleton_07 valueOf(java.lang.String);
    Code:
       0: ldc           #4                  // class cn/bugstack/design/Singleton_07
       2: aload_0
       3: invokestatic  #5                  // Method java/lang/Enum.valueOf:(Ljava/lang/Class;Ljava/lang/String;)Ljava/lang/Enum;
       6: checkcast     #4                  // class cn/bugstack/design/Singleton_07
       9: areturn
```

继承了Enum，解释了为什么枚举无法用于继承场景

static final修饰成员变量

保证类初始化的唯一和可见性

其实还有第七种方式：CAS自旋，但比较复杂，还涉及到硬件指令了，稍后我再补充

本篇文章参考：

- java并发编程的艺术
- pdai：https://pdai.tech/md/dev-spec/pattern/2_singleton.html
- 小傅哥的重学设计模式：https://www.bilibili.com/video/BV1KY4y1q7EX/?spm_id_from=333.788&vd_source=2e2bfda6d934ded5322975ea3994ea7c
- 以及各种优秀的帖子，博客
- 还有为我解答疑惑的小伙伴@芒果冰，@俊深哥

感谢开源的互联网