单例模式的多种实现

作者：DongXiaokai0819
原文链接：https://ld246.com/article/1571748176619
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

单例模式

单例模式来确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单类，它提供全局访问的方法。

单例模式确保一个类只有一个实例，并提供一个全局访问点。

单例模式是一种对象创建型模式。

单例模式又被称为单件模式或单态模式。

单例模式的要点有三个：

1. 某个类只能有一个实例
2. 必须自行创建这个实例
3. 必须自行向整个系统提供这个实例

单例模式的经典实现

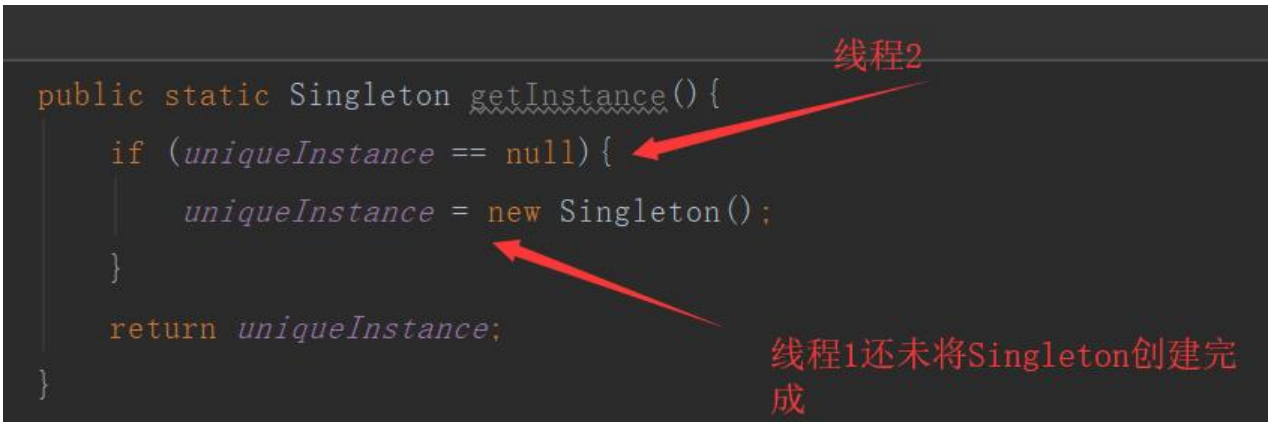
```
public class Singleton {
    private static Singleton uniqueInstance;//利用一个静态变量来记录Singleton类的唯一实例
    //这里有其它的有用实例化变量

    private Singleton(){};//私有构造器，只有自己才可new自己

    public static Singleton getInstance(){
        if (uniqueInstance == null){//如果uniqueInstance是空的，表示还没有创建实例
            uniqueInstance = new Singleton();//如果uniqueInstance不存在，我们就调用私有构造器
成一个实例
            //并把它赋值给uniqueInstance
            //如果我们不需要这个实例，那么这个实例就永远不会产生，这就是延迟实例化
        }
        return uniqueInstance;
    }

    public static void operation(){//类中的其它有用的方法，最好是static的
    }
}
```

模式分析：

- 单例类拥有一个私有构造函数，确保用户无法通过new关键字直接实例化它。
- 该模式还包含一个静态私有成员变量与静态共有的工厂方法，该工厂方法负责检验实例的存在性并迟实例化自己，然后存储在静态成员的变量中，以确保只有一个实例被创建。

多线程情况下的单例模式

上述单例模式的经典实现，在多线程的情况下是有问题的。



在当线程1还未new 出Singleton对象时线程2判断为true，这样就会导致创建了俩个Singleton对象。

如何避免这种情况呢，只要把getinstance()变成同步方法，多线程的灾难就轻而易举的解决了，如下示：

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

如果你的应用程序可以接受getInstance()造成的额外负担，直接在方法上加上synchronized 关键字最简单的实现方式，这可能会造成程序的执行效率大大下降，如果将getInstance()的程序使用在频繁行的地方，就必须得重新考虑了。

懒汉式-双重检查加锁

顺着上述直接加synchronized 关键字同步的思路，我们可以将加锁的范围尽量缩小。

```
public static Singleton getInstance(){
    if(instance == null) {//判断是否有Singleton实例对象
        synchronized (Singleton.class) {//类锁  A处
            if(instance == null) {//再次进行判断  B处
                instance = new Singleton();//如果没有进行创建  C处
            }
        }
    }
    return instance ;
}
```

双重检查锁定背后的理论是：在 B处的第二次检查可以使得创建两个不同的 Singleton 对象成为不可。假设现在有俩个线程，产生了如下所示的事件序列：

1. 线程 1 进入 getInstance() 方法。
2. 由于 instance 为 null，线程 1 在 A处进入 synchronized 块，获取Singleton类锁。
3. 线程 1 被线程 2 预占。
4. 线程 2 进入 getInstance() 方法。
5. 由于 instance 仍旧为 null，线程 2 试图获取 A处的锁。然而，由于线程 1 持有该锁，线程 2 在 A 阻塞。

6. 线程 2 被线程 1 预占。

7. 线程 1 执行，由于在 B 处实例仍旧为 null，线程 1 还创建一个 Singleton 对象并将其引用赋值给 instance。

8. 线程 1 退出 synchronized 块并从 getInstance() 方法返回实例。

9. 线程 1 被线程 2 预占。

10. 线程 2 获取 A 处的锁并检查 instance 是否为 null。

11. 由于 instance 是非 null 的，并没有创建第二个 Singleton 对象，由线程 1 创建的对象被返回。

双重检查锁定背后的理论是完美的。不幸地是，现实完全不同。

双重检查锁定的问题是：并不能保证它会在单处理器或多处理器计算机上顺利运行。

双重检查锁定失败的问题并不归咎于 JVM 中的实现 bug，而是归咎于 Java 平台内存模型。内存模允许所谓的“**无序写入**”，这也是这些习语失败的一个主要原因。

无序写入

为解释该问题，需要重新考察上述清单 4 中的C 行。此行代码创建了一个 Singleton 对象并初始化变 instance 来引用此对象。这行代码的问题是：在 Singleton 构造函数体执行之前，变量 instance 可成为非 null 的。

什么？这一说法可能让您始料未及，但事实确实如此。在解释这个现象如何发生前，请先暂时接受这事实，我们先来考察一下双重检查锁定是如何被破坏的。

假设清单 4 中代码执行以下事件序列：

1. 线程 1 进入 getInstance() 方法。

2. 由于 instance 为 null，线程 1 在 A 处进入 synchronized 块。

3. 线程 1 前进到C 处，但在构造函数执行之前，使实例成为非 null。

4. 线程 1 被线程 2 预占。

5. 线程 2 检查实例是否为 null。因为实例不为 null，线程 2 将 instance 引用返回给一个构造完整但分初始化了的 Singleton对象。

6. 线程 2 被线程 1 预占。

7. 线程 1 通过运行 Singleton 对象的构造函数并将引用返回给它，来完成对该对象的初始化。

对象的初始化，并不是有序的，并不是一次性完成的。

一种可能的情况是，生成了一个实例，但还未将该实例的属性值初始化（即还未执行构造方法），而时instance 已经不为null。所以上述事件序列线程2还未等到线程1将Singleton对象完全初始化完成得知instance不为null，便把这个不完全的instance 返回了。线程1继续执行初始化，然后将完全实化的instance 返回。这样线程1与线程2便返回了俩个不完全相同的实例对象。

解决双重检查加锁的问题

解决Singleton 实例对象在不完全初始化的情况下返回而产生了俩种实例的情况，只需要在private volatile static Singleton instance;加一个volatile 关键字即可。这样就保证了instance对象在多个线程间的可见性。

但是，现在双重检查加锁的这种方式，现在已经不推荐使用了，那么现在如何来实现线程安全的单例式呢?

饿汉式-类初始化模式

```
public class SingleEHan {
    private static final SingleEHan singleEHan = new SingleEHan();
    private SingleEHan(){}

    public static SingleEHan getInstance(){
        return singleEHan;
    }
}
```

在JVM中，对类的加载和初始化，由虚拟机保证线程安全。

但是如果SingleEHan 这个类很大的话，可能会占据很多的内存空间。可以考虑下面的懒汉式-类初始模式。

懒汉式-类初始化模式/延迟占位模式

```
public class SingleInit {
    private SingleInit(){}

    //定义一个私有类，来持有当前类的实例
    private static class InstanceHolder{
        public static SingleInit instance = new SingleInit();
    }

    public static SingleInit getInstance(){
        return InstanceHolder.instance;
    }
}
```

该实现方式，同样也是利用了JVM保证了类的加载和初始化时的线程安全。

JVM在对SingleInit 进行加载的时候，是不会对私有类进行初始化的，只有当调用getInstance()方法，JVM才会将私有类初始化，并由私有类来代替，返回实例出去。

延迟占位模式其实是个很用处很广泛的模式，下面举个栗子~

```
public class InstanceLazy {
    private Integer value;
    private Integer val ;//可能是一个很大的对象，也可能是一个非常大的数组,但是这个属性可能平用的不是很多，所以可以在需要这个属性val的时候，再将其进行初始化，采用延迟占位同时也保证了程安全。

    public InstanceLazy(Integer value) {
        this.value = value;
    }

    public Integer getValue() {
        return value;
    }

    private static class ValHolder {
        public static Integer vHolder = new Integer(1000000);
```

```
    }

    public Integer getVal() {
        return ValHolder.vHolder;
    }
}
```

单例模式的优缺点

优点：

1. 提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以何时访问它。

2. 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象单例模式无疑可以提高系统的性能。

缺点：

1. 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难

2. 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类即充当了工厂角色，提供工厂方法，又充当了产品角色，包含一些业务方法，将产品的创建和产品本身的功能融合在了一起。

单例模式的适用环境

- 系统只需要一个实例对象的时候
- 系统需要考虑资源消耗太大只允许创建一个对象
- 客户端调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问实例。

参考

- 《Head First 设计模式》
- 《软件体系结构与设计》
- https://blog.csdn.net/chenchaofuck1/article/details/51702129