大家好，我是楼仔！

我理解单例应该是所有模式中，最简单，也是使用最多的一种，我就简单总结一下，首先了解一下单例模式的定义。

> 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。

由单例的定义，可以分析出，实现一个单例，有以下几个要点：

- 构造函数必须私有化，防止外部调用构造函数进行实例；
- 提供静态函数获得该单例。

单例主要有两种种实现方式，懒汉模式和饿汉模式。

懒汉模式

在类加载时，不创建实例，因此类加载速度快，但运行时获取对象的速度慢，代码如下：

```
public class penguin {
    private static volatile penguin m_penguin = null;
    // 避免通过new初始化对象
    private void penguin() {}
    public void beating() {
        System.out.println("打豆豆");
    };
    public static penguin getInstance() {
        if (null == m_penguin) {
            synchronized(penguin.class) {
                if (null == m_penguin) {
                    m_penguin = new penguin();
                }
            }
        }
        return m_penguin;
    }
}
```

懒汉模式实现要点

- 单例使用volatile修饰；
- 单例实例化时，要用synchronized 进行同步处理；
- 双重null判断。

下面模拟一个简单的单例并发测试，可以使用CountDownLatch，使用await()等待锁释放，使用countDown()释放锁从而达到并发的效果，可以见下面的代码：

```
public static void main(String args[]) {
    final CountDownLatch latch = new CountDownLatch(1);
    int threadCount = 20;
    for (int i = 0; i < threadCount; i++) {
        new Thread() {
            @Override
            public void run() {
                try {
                    latch.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(penguin.getInstance().hashCode());
            }
        }.start();
    }
    latch.countDown();
}
```

打印出来的hashCode完全一样，证明单例模式生效，输出如下：

```
1449937592
1449937592
1449937592
1449937592
1449937592
1449937592
1449937592
1449937592
1449937592
1449937592
```

饿汉模式

在类加载时就完成了初始化，所以类加载较慢，但获取对象的速度快，代码如下：

```
public class penguin {
    private static penguin m_penguin = new penguin();
    private void penguin() {}
    public static penguin getInstance() {
        return m_penguin;
    }
}
```

两种实现模式各有优缺点，综合来说，个人比较偏向于懒汉模式。

适用场景

单例模式只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等。如：

- 需要频繁实例化然后销毁的对象。
- 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
- 有状态的工具类对象。
- 频繁访问数据库或文件的对象。

优缺点

优点：

- 1.在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就 防止其它对象对自己的实例化，确保所有的对象都访问一个实例
- 2.单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。
- 3.提供了对唯一实例的受控访问。
- 4.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，当 需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。
- 5.避免对共享资源的多重占用。

缺点：

- 1.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。
- 2.由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。
- 3.单例类的职责过重，在一定程度上违背了“单一职责原则”。
- 4.滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

volatile问题

大家有没有注意到，单例模式中用到了关键字volatile，在PHP和Go中没有类似的关键字，但是JAVA必须加，当初还有疑问，我们先看一下volatile的作用：

> volatile是Java提供的一种轻量级的同步机制。Java 语言包含两种内在的同步机制：同步块（或方法）和 volatile 变量，相比于synchronized（synchronized通常称为重量级锁），volatile更轻量级，因为它不会引起线程上下文的切换和调度。但是volatile 变量的同步性较差（有时它更简单并且开销更低），而且其使用也更容易出错。

我直接总结一下volatile的作用：

- 它会强制将对缓存的修改操作立即写入主存，让所有的线程可见；
- 它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；
- 如果是写操作，它会导致其他CPU中对应的缓存行无效。

再回顾一下单例模式代码：

```
public class penguin {
    private static volatile penguin m_penguin = null;
    // 避免通过new初始化对象
    private void penguin() {}
    public void beating() {
        System.out.println("打豆豆");
    };
    public static penguin getInstance() {      //1
        if (null == m_penguin) {               //2
            synchronized(penguin.class) {      //3
                if (null == m_penguin) {       //4
                    m_penguin = new penguin(); //5
                }
            }
        }
        return m_penguin;                      //6
    }
}
```

在并发情况下，如果没有volatile关键字，在第5行会出现问题。instance = new TestInstance();可以分解为3行伪代码：

```
a. memory = allocate() //分配内存
b. ctorInstanc(memory) //初始化对象
c. instance = memory //设置instance指向刚分配的地址
```

上面的代码在编译运行时，可能会出现重排序从a-b-c排序为a-c-b。在多线程的情况下会出现以下问题。当线程A在执行第5行代码时，B线程进来执行到第2行代码。假设此时A执行的过程中发生了指令重排序，即先执行了a和c，没有执行b。那么由于A线程执行了c导致instance指向了一段地址，所以B线程判断instance不为null，会直接跳到第6行并返回一个未初始化的对象。

关于volatile关键字，后面我会单独拿一篇文章进行讲解~~

后记

单例模式其实还有其它的实现方式，但是主要就用到“懒汉模式”，其它的实现方式，大家感兴趣的话，可以到网上再查一下。其实本来不想写单例模式，因为感觉太简单了，然后这篇文章，应该也是我所有文章中，写的最快的一篇（五一出去旅游，晚上在酒店很快写完了，老婆对我出去玩，回来还抱个电脑很有意见），那就通过这篇文章，简单记录一下吧。

学习交流

可以扫下面二维码，关注「楼仔」公众号。

长按二维码，回复 **「加群」**，欢迎一起学习交流哈~~ 👏👏👏

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。