参考：<http://www.cnblogs.com/lewis0077/p/5128971.html>

[**深入解析单例模式**](http://www.cnblogs.com/lewis0077/p/5128971.html)

　　单例模式在程序设计中非常的常见，一般来说，某些类，我们希望在程序运行期间有且只有一个实例，原因可能是该类的创建需要消耗系统过多的资源、花费很多的时间，或者业务上客观就要求了只能有一个实例。一个场景就是：我们的应用程序有一些配置文件，我们希望只在系统启动的时候读取这些配置文件，并将这些配置保存在内存中，以后在程序中使用这些配置文件信息的时候不必再重新读取。

**定义：**

　　由于某种需要，要保证一个类在程序的生命周期当中只有一个实例，并且提供该实例的全局访问方法。

**结构：**

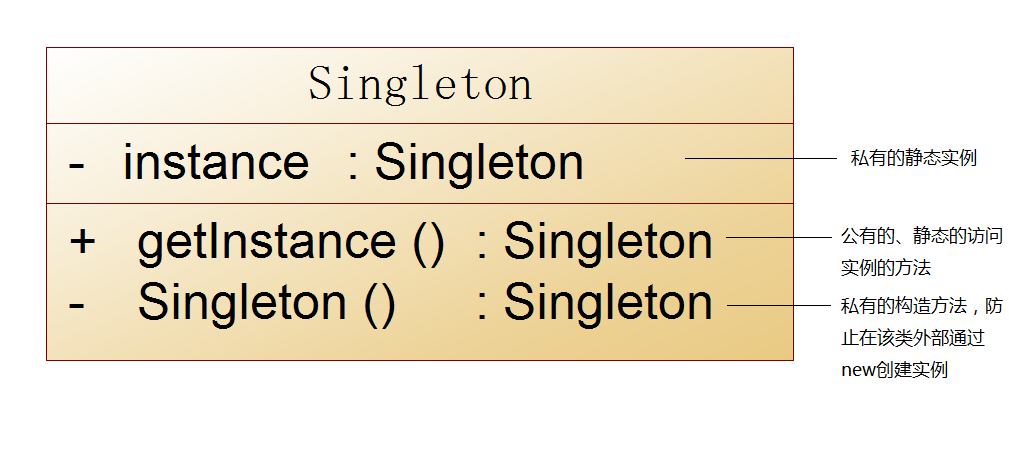
　　一般包含三个要素：

　　1.私有的静态的实例对象 private static instance

　　2.私有的构造函数（保证在该类外部，无法通过new的方式来创建对象实例） private Singleton(){}

　　3.公有的、静态的、访问该实例对象的方法 public static Singleton getInstance(){}

**UML类图：**



**分类：**

　　单例模式就实例的创建时机来划分可分为：懒汉式与饥汉式两种。

　　举个日常生活中的例子：

　　　　妈妈早上起来为我们做饭吃，饭快做好的时候，一般都会叫我们起床吃饭，这是一般的日常情况。如果饭还没有好的时候，我们就自己起来了（这时候妈妈还没有叫我们起床），这种情况在单例模式中称之为饥汉式（妈妈还没有叫我们起床，我们自己就起来的，就是外部还没有调用自己，自己的实例就已经创建好了）。如果饭做好了，妈妈叫我们起床之后，我们才慢吞吞的起床，这种情况在单例模式中称之为懒汉式（饭都做好了，妈妈叫你起床之后，自己才起的，能不懒汉吗？就是外部对该类的方法发出调用之后，该实例才建立的）。

　　懒汉式：顾名思义懒汉式就是应用刚启动的时候，并不创建实例，当外部调用该类的实例或者该类实例方法的时候，才创建该类的实例。是以时间换空间。

　　懒汉式的优点：实例在被使用的时候才被创建，可以节省系统资源，体现了**延迟加载**的思想。

**延迟加载**：通俗上将就是：一开始的时候不加载资源，一直等到马上就要使用这个资源的时候，躲不过去了才加载，这样可以尽可能的节省系统资源。

      懒汉式的缺点：由于系统刚启动时且未被外部调用时，实例没有创建；如果一时间有多个线程同时调用LazySingleton.getLazyInstance()方法很有可能会产生多个实例。

　　　　　　　　　也就是说下面的懒汉式在多线程下，是不能保持单例实例的唯一性的，要想保证多线程下的单例实例的唯一性得用同步，同步会导致多线程下由于争夺锁资源，运行效率不高。

　　饥汉式：顾名思义懒汉式就是应用刚启动的时候，不管外部有没有调用该类的实例方法，该类的实例就已经创建好了。以空间换时间。

　　饥汉式的优点：写法简单，在多线程下也能保证单例实例的唯一性，不用同步，运行效率高。

　　饥汉式的缺点：在外部没有使用到该类的时候，该类的实例就创建了，若该类实例的创建比较消耗系统资源，并且外部一直没有调用该实例，那么这部分的系统资源的消耗是没有意义的。

 下面是懒汉式单例类的演示代码：

[复制代码](javascript:void(0);)

1 package singleton;

2

3 /\*\*

4 \* 懒汉式单例类

5 \*/

6 public class LazySingleton {

7

8 //私有化构造函数，防止在该类外部通过new的形式创建实例

9 private LazySingleton() {

10 System.out.println("生成LazySingleton实例一次！");

11 }

12

13 //私有的、静态的实例，设置为私有的防止外部直接访问该实例变量，设置为静态的，说明该实例是LazySingleton类型的唯一的

14 //若开始时，没有调用访问实例的方法，那么实例就不会自己创建

15 private static LazySingleton lazyInstance = null;

16

17 //公有的访问单例实例的方法，当外部调用访问该实例的方法时，实例才被创建

18 public static LazySingleton getLazyInstance() {

19 //若实例还没有创建，则创建实例；若实例已经被创建了，则直接返回之前创建的实例,即不会返回2个实例

20 if (lazyInstance == null) {

21 lazyInstance = new LazySingleton();

22 }

23 return lazyInstance;

24 }

25 }

[复制代码](javascript:void(0);)

下面测试类：

[复制代码](javascript:void(0);)

1 package singleton;

2

3

4 public class SingletonTest {

5 public static void main(String[] args) {

6 LazySingleton lazyInstance1 = LazySingleton.getLazyInstance();

7 LazySingleton lazyInstance2 = LazySingleton.getLazyInstance();

8 LazySingleton lazyInstance3 = LazySingleton.getLazyInstance();

9 }

10 }

[复制代码](javascript:void(0);)

在上面的测试类SingletonTest 里面，连续调用了三次LazySingleton.getLazyInstance()方法，

控制台输出：

生成LazySingleton实例一次！

下面代码演示饥汉式单例实现：

[复制代码](javascript:void(0);)

1 package singleton;

2

3 public class NoLazySingleton {

4

5 //私有化构造函数，防止在该类外部通过new的形式创建实例

6 private NoLazySingleton(){

7 System.out.println("创建NoLazySingleton实例一次！");

8 }

9

10 //私有的、静态的实例，设置为私有的防止外部直接访问该实例变量，设置为静态的，说明该实例是LazySingleton类型的唯一的

11 //当系统加载NoLazySingleton类文件的时候，就创建了该类的实例

12 private static NoLazySingleton instance = new NoLazySingleton();

13

14 //公有的访问单例实例的方法

15 public static NoLazySingleton getInstance(){

16 return instance;

17 }

18 }

[复制代码](javascript:void(0);)

测试代码：

[复制代码](javascript:void(0);)

package singleton;

public class SingletonTest {

public static void main(String[] args) {

NoLazySingleton instance = NoLazySingleton.getInstance();

NoLazySingleton instance1 = NoLazySingleton.getInstance();

NoLazySingleton instanc2 = NoLazySingleton.getInstance();

NoLazySingleton instanc3 = NoLazySingleton.getInstance();

}

}

[复制代码](javascript:void(0);)

控制台输出：

创建NoLazySingleton实例一次！

上面说到了懒汉式在多线程环境下面是有问题的，下面演示这个多线程环境下很有可能出现的问题：

[复制代码](javascript:void(0);)

1 package singleton;

2

3 /\*\*

4 \* 懒汉式单例类

5 \*/

6 public class LazySingleton {

7

8 //为了易于模拟多线程下，懒汉式出现的问题，我们在创建实例的构造函数里面使当前线程暂停了50毫秒

9 private LazySingleton() {

10 try {

11 Thread.sleep(50);

12 } catch (InterruptedException e) {

13 e.printStackTrace();

14 }

15 System.out.println("生成LazySingleton实例一次！");

16 }

17

18 private static LazySingleton lazyInstance = null;

19

20 public static LazySingleton getLazyInstance() {

21 if (lazyInstance == null) {

22 lazyInstance = new LazySingleton();

23 }

24 return lazyInstance;

25 }

26 }

[复制代码](javascript:void(0);)

下面是测试代码： 我们在测试代码里面 新建了10个线程，让这10个线程同时调用LazySingleton.getLazyInstance()方法

[复制代码](javascript:void(0);)

1 package singleton;

2

3 public class SingletonTest {

4 public static void main(String[] args) {

5 for (int i = 0; i < 10; i++) {

6 new Thread(){

7 @Override

8 public void run() {

9 LazySingleton.getLazyInstance();

10 }

11 }.start();

12 }

13 }

14 }

[复制代码](javascript:void(0);)

结果控制台输出：

生成LazySingleton实例一次！  
生成LazySingleton实例一次！  
生成LazySingleton实例一次！  
生成LazySingleton实例一次！  
生成LazySingleton实例一次！  
生成LazySingleton实例一次！  
生成LazySingleton实例一次！  
生成LazySingleton实例一次！  
生成LazySingleton实例一次！  
生成LazySingleton实例一次！

没错，你没有看错，控制台输出了10次，表示懒汉式单例模式在10个线程同时访问的时候，创建了10个实例，这足以说明懒汉式单例在多线程下已不能保持其实例的唯一性。

那为什么多线程下懒汉式单例会失效？我们下面分析原因：

　　我们不说这么多的线程，就说2个线程同时访问上面的懒汉式单例，现在有两个线程A和B同时访问LazySingleton.getLazyInstance()方法。

假设A先得到CPU的时间切片，A执行到21行处 if (lazyInstance == null) 时，由于lazyInstance 之前并没有实例化，所以lazyInstance == null为true,在还没有执行22行实例创建的时候

此时CPU将执行时间分给了线程B，线程B执行到21行处 if (lazyInstance == null) 时，由于lazyInstance 之前并没有实例化，所以lazyInstance == null为true，线程B继续往下执行实例的创建过程，线程B创建完实例之后，返回。

此时CPU将时间切片分给线程A，线程A接着开始执行22行实例的创建，实例创建完之后便返回。由此看线程A和线程B分别创建了一个实例（存在2个实例了），这就导致了单例的失效。

那如何将懒汉式单例在多线程下正确的发挥作用呢？当然是在访问单例实例的方法处进行同步了

下面是线程安全的懒汉式单例的实现：

[复制代码](javascript:void(0);)

1 package singleton;

2

3

4 public class SafeLazySingleton {

5

6 private SafeLazySingleton(){

7 System.out.println("生成SafeLazySingleton实例一次！");

8 }

9

10 private static SafeLazySingleton instance = null;

11 　　 //1.对整个访问实例的方法进行同步

12 public synchronized static SafeLazySingleton getInstance(){

13 if (instance == null) {

14 instance = new SafeLazySingleton();

15 }

16 return instance;

17 }  
　　　　//2.对必要的代码块进行同步

18 public static SafeLazySingleton getInstance1(){

19 if (instance == null) {

20 synchronized (SafeLazySingleton.class){

21 if (instance == null) {

22 instance = new SafeLazySingleton();

23 }

24 }

25 }

26 return instance;

27 }

28 }

[复制代码](javascript:void(0);)

 对方法同步：

上面的实现 在12行对访问单例实例的整个方法用了synchronized 关键字进行方法同步，这个缺点很是明显，就是锁的粒度太大，很多线程同时访问的时候导致阻塞很严重。

对代码块同步：

在18行的方法getInstance1中，只是对必要的代码块使用了synchronized关键字，注意由于方法时static静态的，所以监视器对象是SafeLazySingleton.class

同时我们在19行和21行，使用了实例两次非空判断，一次在进入synchronized代码块之前，一次在进入synchronized代码块之后，这样做是有深意的。

肯定有小伙伴这样想：既然19行进行了实例非空判断了，进入synchronized代码块之后就不必再次进行非空判断了，如果这样做的话，会导致什么问题？我们来分析一下：

同样假设我们有两个线程A和B，A获取CPU时间片段，在执行到19行时，由于之前没有实例化，所以instance == null 为true,然后A获得监视器对象SafeLazySingleton.class的锁，A进入synchronized代码块里面；

与此同时线程B执行到19行，此时线程A还没有执行实例化动作，所以此时instance == null 为true,B想进入同步块，但是发现锁在线程A手里，所以B只能在同步块外面等待。此时线程A执行实例化动作，实例化结束之后，返回该实例。

随着线程A退出同步块，A也释放了锁，线程B就获得了该锁，若此时不进行第二次非空判断，会导致线程B也实例化创建一个实例，然后返回自己创建的实例，这就导致了2个线程访问创建了2个实例，导致单例失效。若进行第二次非空判断，发现线程A已经创建了实例，instance == null已经不成立了，则直接返回线程A创建的实例，这样就避免了单例的失效。

有细心的网友会发现即便去掉19行非空判断，多线程下单例模式一样有效：

　　线程A获取监视器对象的锁，进入了同步代码块，if(instance == null) 成立，然后A创建了一个实例，然后退出同步块，返回。这时在同步块外面等待的线程B，获取了锁进入同步块，执行if(instance == null)发现instance已经有值了不再是空了，然后直接退出同步块，返回。

　　既然去掉19行，多线程下单例模式一样有效，那为什么还要有进入同步块之前的非空判断（19行）？这应该主要是考虑到多线程下的效率问题：

　　我们知道使用synchronized关键字进行同步，意味着就是独占锁，同一时刻只能有一个线程执行同步块里面的代码，还要涉及到锁的争夺、释放等问题，是很消耗资源的。单例模式，构造函数只会被调用一次。如果我们不加19行，即不在进入同步块之前进行非空判断，如果之前已经有线程创建了该类的实例了，那每次的访问该实例的方法都会进入同步块，这会非常的耗费性能.如果进入同步块之前加上了非空判断，发现之前已经有线程创建了该类的实例了，那就不必进入同步块了，直接返回之前创建的实例即可。这样就基本上解决了线程同步导致的性能问题。

多线程下单例的优雅的解决方案：

上面的实现使用了synchronized同步块，并且用了双重非空校验，这保证了懒汉式单例模式在多线程环境下的有效性，但这种实现感觉还是不够好，不够优雅。

下面介绍一种优雅的多线程下单例模式的实现方案：

[复制代码](javascript:void(0);)

1 package singleton;

2

3 public class GracefulSingleton {

4 private GracefulSingleton(){

5 System.out.println("创建GracefulSingleton实例一次！");

6 }

7   
     //类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例没有绑定关系，而且只有被调用到才会装载，从而实现了延迟加载

8 private static class SingletonHoder{  
　　　　　　　//静态初始化器，由JVM来保证线程安全

9 private static GracefulSingleton instance = new GracefulSingleton();

10 }

11

12 public static GracefulSingleton getInstance(){

13 return SingletonHoder.instance;

14 }

15 }

[复制代码](javascript:void(0);)

上面的实现方案使用一个内部类来维护单例类的实例，当GracefulSingleton被加载的时候，其内部类并不会被初始化，所以可以保证当GracefulSingleton被装载到JVM的时候，不会实例化单例类，当外部调用getInstance方法的时候，才会加载内部类SingletonHoder，从而实例化instance,同时由于实例的建立是在类初始化时完成的，所以天生对多线程友好，getInstance方法也不需要进行同步。

单例模式本质上是控制单例类的实例数量只有一个，有些时候我们可能想要某个类特定数量的实例，这种情况可以看做是单例模式的一种扩展情况。比如我们希望下面的类SingletonExtend只有三个实例，我们可以利用Map来缓存这些实例。

[复制代码](javascript:void(0);)

1 package singleton;

2

3 import java.util.HashMap;

4 import java.util.Map;

5

6 public class SingletonExtend {

7 //装载SingletonExtend实例的容器

8 private static final Map<String,SingletonExtend> container = new HashMap<String, SingletonExtend>();

9 //SingletonExtend类最多拥有的实例数量

10 private static final int MAX\_NUM = 3;

11 //实例容器中元素的key的开始值

12 private static String CACHE\_KEY\_PRE = "cache";

13 private static int initNumber = 1;

14 private SingletonExtend(){

15 System.out.println("创建SingletonExtend实例1次！");

16 }

17

18 //先从容器中获取实例，若实例不存在，在创建实例，然后将创建好的实例放置在容器中

19 public static SingletonExtend getInstance(){

20 String key = CACHE\_KEY\_PRE+ initNumber;

21 SingletonExtend singletonExtend = container.get(key);

22 if (singletonExtend == null) {

23 singletonExtend = new SingletonExtend();

24 container.put(key,singletonExtend);

25 }

26 initNumber++;

27 //控制容器中实例的数量

28 if (initNumber > 3) {

29 initNumber = 1;

30 }

31 return singletonExtend;

32 }

33

34 public static void main(String[] args) {

35 SingletonExtend instance = SingletonExtend.getInstance();

36 SingletonExtend instance1 = SingletonExtend.getInstance();

37 SingletonExtend instance2 = SingletonExtend.getInstance();

38 SingletonExtend instance3 = SingletonExtend.getInstance();

39 SingletonExtend instance4 = SingletonExtend.getInstance();

40 SingletonExtend instance5 = SingletonExtend.getInstance();

41 SingletonExtend instance6 = SingletonExtend.getInstance();

42 SingletonExtend instance7 = SingletonExtend.getInstance();

43 SingletonExtend instance8 = SingletonExtend.getInstance();

44 SingletonExtend instance9 = SingletonExtend.getInstance();

45 System.out.println(instance);

46 System.out.println(instance1);

47 System.out.println(instance2);

48 System.out.println(instance3);

49 System.out.println(instance4);

50 System.out.println(instance5);

51 System.out.println(instance6);

52 System.out.println(instance7);

53 System.out.println(instance8);

54 System.out.println(instance9);

55 }

56 }

[复制代码](javascript:void(0);)

控制台输出：

创建SingletonExtend实例1次！  
创建SingletonExtend实例1次！  
创建SingletonExtend实例1次！  
singleton.SingletonExtend@3a3ee284  
singleton.SingletonExtend@768965fb  
singleton.SingletonExtend@36867e89  
singleton.SingletonExtend@3a3ee284  
singleton.SingletonExtend@768965fb  
singleton.SingletonExtend@36867e89  
singleton.SingletonExtend@3a3ee284  
singleton.SingletonExtend@768965fb  
singleton.SingletonExtend@36867e89  
singleton.SingletonExtend@3a3ee284

从控制台输出情况可以看到 我们成功的控制了SingletonExtend的实例数据只有三个

下面就单例模式总结一下：

我们讲了什么是单例模式，它的结构是怎么样的，并且给出了单例的类图，讲了单例的分类：懒汉式和饥汉式，分别讲了它们在单线程、多线程环境下的实现方式，它们的优点和缺点，以及优雅的单例模式的实现，最后讲了单例模式的扩展，小伙伴们你们清楚了吗？