单例模式的写法大致分为4类：

#### 1、饱汉模式

饱汉是变种最多的单例模式。我们从饱汉出发，通过其变种逐渐了解实现单例模式时需要关注的问题。

1. **基础的饱汉**

先不初始化单例，等第一次使用的时候再初始化，即“懒加载”。

public class Singleton1 {

private static Singleton1 *singleton* = null;  
  
 public Singleton1() {  
 }  
  
 private static Singleton1 getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new Singleton1();  
 }  
 return *singleton*;  
 }  
}

启动快，节省资源；线程不安全，if语句存在竟态条件。

1. **饱汉-变种1**

使用synchronized关键字修饰getInstance()方法，达到绝对的线程安全。

public class Singleton1\_1 {

private static Singleton1\_1 *singleton* = null;  
  
 public Singleton1\_1() {  
 }  
  
 public synchronized static Singleton1\_1 getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new Singleton1\_1();  
 }  
 return *singleton*;  
 }  
}

线程安全；并发性能极差，事实上完全退化到了串行。单例只需要初始化一次，但就算初始化以后，synchronized的锁也无法避开，从而getInstance()完全编程了串行操作。

1. 饱汉-变种2

DCL 1.0

基本上，把synchronized移动到代码内部是没什么意义的，每次调用getInstance()还是要进行同步。同步本身没有问题，但是我们只希望在第一次创建instance实例的时候进行同步，因此有了下面的写法：变种2在变种1的外层又套上了一层check，加上synchronized内存的check，即所谓“双重检查锁”（Double Check Lock,简称DCL）。

public class Singleton1\_2 {

private static Singleton1\_2 *singleton* = null;  
  
 private int f1 = 1;  
 private int f2 = 2;  
   
 public Singleton1\_2() {  
 }  
  
 public static Singleton1\_2 getInstance() {  
 //当instance不为null时，可能指向一个“被部分初始化的对象”  
 if (*singleton* == null) {  
 synchronized (Singleton1\_2.class) {  
 if (*singleton* == null) {  
 *singleton* = new Singleton1\_2();  
 }  
 }  
 }  
 return *singleton*;  
 }  
}

DLC仍是线程不安全的，由于指令重排序，你可能会得到“半个对象”，即“部分初始化”问题。

1. **饱汉-变种3**

DCL 2.0

针对变种DCL 1.0的“半个对象”问题，变种3在instance上增加了volatile关键字。

public class Singleton1\_3 {

private static volatile Singleton1\_3 *singleton* = null;  
  
 private int f1 = 1;  
 private int f2 = 2;  
  
 public Singleton1\_3() {  
 }  
  
 public static Singleton1\_3 getInstance() {  
 if (*singleton* == null) {  
 //must be a complete instance  
 synchronized (Singleton1\_3.class) {  
 if (*singleton* == null) {  
 *singleton* = new Singleton1\_3();  
 }  
 }  
 }  
 return *singleton*;  
 }  
}

多线程环境下，变种3更适合用于性能敏感的场景。但后面我们将了解到，就算线程安全的，还有一些办法能够破坏单例。当然，还有很多方式，能通过与volatile类似的方式防止部分初始化。

#### 2、饿汉模式

类加载时初始化单例，以后访问时直接返回即可。

public class Singleton2 {

private static final Singleton2 *singleton* = new Singleton2();  
  
 public Singleton2() {  
 }  
  
 public static Singleton2 getInstance() {  
 return *singleton*;  
 }  
}

线程安全（得益于类加载机制）；使用时没有延迟；可能造成资源浪费（如果类加载后就一直不使用单例的话）。

注意：单线程环境下，饿汉与饱汉在性能上没差别；但在多线程环境下，由于饱汉需要加锁，饿汉的性能反而更优。

#### 3、Holder模式

我们既希望饿汉模式中静态变量的方便和线程安全；又希望通过懒加载规避资源浪费。Holder模式满足了这两点要求：核心仍然是静态变量，足够方便和线程安全；通过静态的Holder类持有真正实例，间接实现了懒加载。

public class Singleton3 {

*/\*\*  
 \* 类级内部类，即静态的成员内部类，该内部类的实例与外部类的实例没有绑定关系  
 \* 只有被调用时才会装载，从而实现了延迟加载。  
 \*/* private static class SingletonHolder {  
 //静态初始化器，由JVM来保证线程安全  
 private static final Singleton3 *singleton* = new Singleton3();  
 }  
  
 public static Singleton3 getInstance() {  
 return SingletonHolder.*singleton*;  
 }  
}

Holder模式仅增加了一个静态内部类的成本，与饱汉的变种3效果相当（略优）。

#### 4、枚举模式

基础的枚举：将枚举的静态成员变量作为单例的实例：

public enum Singleton4 {

SINGLETON;  
}

代码量比饿汉式更少。但用户只能直接访问实例Singleton4.SINGLETON——事实上，这样的访问方式作为单例使用也是恰当的，只是牺牲了静态工厂方法的优点，如无法实现懒加载。

#### 丑陋但好用的语法糖

Java的枚举是一个“丑陋但好用的语法糖”。

枚举型单例模式的本质：通过反编译打开语法糖，就看到枚举型的本质，简化如下

public class Singleton4 extends Enum<Singleton4> {

...

public static final Singleton4 SINGLETON = new Singleton4();

...

}

本质上和饿汉模式相同，区别仅在于公有的静态成员变量。

用枚举实现的一些trick：

虽然枚举相当灵活，但如何恰当的使用枚举有一定难度。一个足够简单的典型例子是TimeUnit类，建议有时间耐心阅读。

上面已经看到，枚举型单例的本质仍然是一个普通的类。实际上，我们可以在枚举型型单例上增加任何普通类可以完成的功能。要点在于枚举实例的初始化，可以理解为实例化了一个匿名内部类。为了更明显，我们在Singleton4\_1中定义一个普通的私有成员变量，一个普通的公有成员方法，和一个公有的抽象成员方法，如下：

public enum Singleton4\_1 {

*/\*\*  
 \*  
 \*/  
 SINGLETON*("枚举是最简单的单例模式，但不是最易读的模式") {  
 @Override  
 public void testAbsMethod() {  
 print();  
 System.*out*.println("枚举很难看，但是灵活多变");  
 }  
 };  
  
 private String comment = null;  
  
 Singleton4\_1(String comment) {  
 this.comment = comment;  
 }  
  
 public void print() {  
 System.*out*.println("comment=" + comment);  
 }  
  
 abstract public void testAbsMethod();  
  
 public static Singleton4\_1 getInstance() {  
 return *SINGLETON*;  
 }  
}

枚举类Singleton4\_1中的每一个枚举实例不仅继承了父类Singleton4\_1的成员方法print()，还必须实现父类Singleton4\_1的抽象成员方法testAbsMethod()。

#### 总结

上面的分析都忽略了反射和序列化的问题。通过反射或序列化，我们仍然能给个访问私有构造器，创建新的实例破坏单例模式。此时，只有枚举模式能天然防范这一问题。

下面忽略反射和序列化的问题，做个总结：

