实现单例模式的8种方式

单例模式是一种常用的软件设计模式，其定义是单例对象的类智能允许一个实例存在。

实现单例模式的8种方式

单例的实现主要时通过以下两个步骤：
1.将该类的构造方法定义为私有方法，这样其他处的代码就无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例。
2.在该类内提供一个静态方法，当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保存的引用。

第一种：饿汉式(静态常量)【可用】

```
public class Singleton01 {
    private  static Singleton01 instance = new Singleton01();

    private Singleton01(){

    }

    public static Singleton01 getInstance(){
        return instance;
    }
}
```

优点：这种写法比较简单，就是在类装载的时候就完成实例化，避免了线程同步问题。
缺点：在类装载的时候就完成实例化，没有达到Lazy Loading的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费。

第二种：饿汉式（静态代码块）【可用】

```
public class Singleton02 {
    private  static Singleton02  instance ;

    static {
```

```
            instance = new Singleton02();
        }

        private Singleton02(){

        }

        public static Singleton02 getInstance(){
            return instance;
        }
    }
```

这种写法和上面的方法其实类似，只不过是将类实例的过程放到了静态代码块中。也是在类装载的时候就执行静态代码块中的代码，初始化类的实例。优缺点和第一种方法一样。

第三种：懒汉式(线程不安全)【不可用】

```
public class Singleton03 {
    private  static Singleton03 instance ;

    private Singleton03(){

    }

    public static Singleton03 getInstance(){
        if(null !=instance){
            instance = new Singleton03();
        }
        return instance;
    }
}
```

这种写法起到了Lazy Loading的效果，但是只能在单线程下适用，如果在多线程下，一个线程进入了 if(null !=instance) 判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例，所以在多线程环境下不可使用这种方式。

第四种：懒汉式(线程安全，同步方法)【不推荐用】

```
public class Singleton04 {
    private  static Singleton04 instance ;
```

```
    private Singleton04(){

    }

    public static synchronized Singleton04 getInstance(){
        if(null !=instance){
            instance = new Singleton04();
        }
        return instance;
    }
}
```

优点：解决上面第三种实现方式的线程不安全问题，对getInstance()方法进行加锁操作，实现了线程同步。
缺点：效率太低，每个线程在获得类的实例时，执行getInstance()方法都要进行同步，而其实这个方法只执行一次实例化就够了，后面要获得该实例直接return.

第五种：懒汉式(同步代码块)【不可用】

```
public class Singleton05 {
    private  static Singleton05 instance ;

    private Singleton05(){

    }

    public static  Singleton05 getInstance(){
        if(null !=instance){
            synchronized (Singleton05.class){
                instance = new Singleton05();
            }
        }
        return instance;
    }
}
```

由于第四种实现方法同步效率太低，所以摒弃同步方法，改为同步产生实例化的代码块，但是这种方式并不能起到线程同步的作用，跟第三种实现方法一样，一个线程进入到if(null !=instance)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例，所以在多线程环境下不可使用这种方式。

第六种：懒汉式(双重检查)【推荐用】

```
public class Singleton06 {
    private  static Singleton06 instance ;

    private Singleton06(){

    }

    public static Singleton06 getInstance(){
        if(null !=instance){
            synchronized (Singleton06.class){
                if(instance !=null){
                    instance = new Singleton06();
                }
            }
        }
        return instance;
    }
}
```

Double-Check概念对于多线程开发者来说并不陌生，如代码中所示，我们进行了两次if(null !=instance)检查，这样就可以保证线程安全了，这样，实例化代码只用执行一次，后面再次访问时，直接return 实例化对象即可。

第七种：静态内部类【推荐用】

```
public class Singleton07 {
    private Singleton07(){

    }

    private static class SingletonInstance{
        private  static Singleton07 instance  = new Singleton07();

    }

    public static Singleton07 getInstance() {
       return SingletonInstance.instance;
```

```
    }
}
```

这种方式跟饿汉式方式采用的机制类似，但又有不同。两者都是采用了类装载的机制来保证初始化实例时只有一个线程，不同的地方在饿汉式方式只要Singleton01类被装载就会实例化，没有Lazy_Loading的作用，而静态内部类方式在Singleton07类被转载时并不会立即实例化，而是在需要实例化时，调用getInstance()方法，才会装载SingletonInstance类，从而完成Singleton07的实例化。

第八种：枚举【推荐用】

```
public enum  Singleton08 {
    INSTANCE;
    public void whateverMethod(){

    }
}
```

借助JDK1.5中添加的枚举来实现单例模式，不仅仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象.

单例模式的作用：
节省内存和计算，保证结果正确，方便管理
单例模式适用场景：
1.无状态的工具类：比如日志工具类，不管是在哪里使用，我们需要的只是它帮我们记录日志信息，除此之外，并不需要在它的实例对象上存储任何状态，这时我们就只需要一个实例对象即可。
2.全局信息类：网站计数器，一般也是使用单例模式，如果你存在多个计数器，每个用户的访问都刷新计数器的值，这样的话你的实计数的值是难以同步，但是如果采用单例模式实现就不会存在这种问题，而且可以避免线程安全问题。
3.多线程的线程池也是采用单例模式，这是由于线程池需要方便对池中的线程进行控制。
4.web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。
5.Windows的Task Manager(任务管理器)就是典型的单例模式。
6.Windows的Recycle Bin(回收站)也是单例模式。
7.操作系统的文件系统也是单例模式，一个操作系统只能有一个文件系统。
单例模式应用场景出现一般发生在一下条件下：
1.资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如日志文件，web应用配置等等。
2.控制资源的情况下，方便资源之间的互相通信，如线程池。