1.单例模式

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个

类只能存在一个对象实例， 并且该类只提供一个取得其对象实例的方法(静态方法)。

比如 Hibernate 的 SessionFactory，它充当数据存储源的代理，并负责创建

Session 对象。SessionFactory并不是轻量级的，一般情况下，一个项目通常只需要

一个SessionFactory 就够，这是就会使用到单例模式。

注意事项和和使用场景

1) 单例模式保证了 系统内存中该类只存在一个对象，节省了系统资源，对于一

些需要频繁创建销毁的对象，使用单例模式可以提高系统性能

2) 当想实例化一个单例类的时候，必须要记住使用相应的获取对象的方法，而不

是使用 new

3) 单例模式使用的场景：需要频繁的进行创建和销毁的对象、创建对象时耗时

过多或耗费资源过多(即：重量级对象)，但又经常用到的对象、工具类对象、频繁访

问数据库或文件的对象(比如数据源、session 工厂等)

具体应用:JDK的java.lang.Runtime就用到了饿汉式

单例模式有八种方式：

1) 饿汉式(静态常量)

2) 饿汉式（静态代码块）

3） 懒汉式(线程不安全)

4） 懒汉式(线程安全，同步方法)

5） 懒汉式(线程安全，同步代码块)

6） 双重检查

7） 静态内部类

8） 枚举 （与其他相比可以防止反序列化）

1.1饿汉式（静态常量）

1） 构造器私有化（防止 new ）

2） 类的内部创建对象

3） 向外暴露一个静态的公共方法。getInstance

优缺点说明：

1） 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。

2） 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

3） 这种方式基于 classloder 机制避免了多线程的同步问题，不过，instance 在类装

载时就实例化，在单例模式中大多数都是调用 getInstance 方法， 但是导致类装载

的原因有很多种，因此不能确定有其他的方式（或者其他的静态方法）导致类装

载，这时候初始化 instance 就没有达到 lazy loading 的效果

4） 结论：这种单例模式可用，可能造成内存浪费

```
//饿汉式(静态变量)
class Singleton {
  //1. 构造器私有化, 外部能new
```

```
    private Singleton() {}

    //2.本类内部创建对象实例
    private final static Singleton instance = new Singleton();

    //3. 提供一个公有的静态方法，返回实例对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

1.2 饿汉式（静态代码块）

1） 这种方式和上面的方式其实类似，只不过将类实例化的过程放在了静态代码块中，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。优缺点和上面是一样的。

2） 结论：这种单例模式可用，但是可能造成内存浪费

```
//饿汉式(静态代码块)
class Singleton {
    //1. 构造器私有化, 外部能new
    private Singleton() {}

    //2.本类内部创建对象实例
    private  static Singleton instance;

    static { // 在静态代码块中，创建单例对象
        instance = new Singleton();
    }

    //3. 提供一个公有的静态方法，返回实例对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

1.3懒汉式(线程不安全)

1） 起到了 Lazy Loading 的效果，但是只能在单线程下使用。

2） 如果在多线程下，一个线程进入了 if (singleton == null)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

3） 结论：在实际开发中，不要使用这种方式.

```
class Singleton {
  private static Singleton instance;

  private Singleton() {}

  //提供一个静态的公有方法，当使用到该方法时，才去创建 instance
  //即懒汉式
  public static Singleton getInstance() {
    if(instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

1.4懒汉式(线程安全，同步方法)

1） 解决了线程安全问题

2） 效率太低了，每个线程在想获得类的实例时候，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代码就够了，后面的想获得该类实

例，直接 return 就行了。方法进行同步效率太低

3） 结论：在实际开发中，不推荐使用这种方式

```
// 懒汉式(线程安全，同步方法)
class Singleton {
  private static Singleton instance;
```

```
  private Singleton() {}

  //提供一个静态的公有方法，加入同步处理的代码，解决线程安全问题
  //即懒汉式
  public static synchronized Singleton getInstance() {
    if(instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

1.5懒汉式(线程安全，同步代码块)

不仅用到了同步，效率低，而且还可能创建多个实例。就是个废物。(但改进成双

重检查锁式比较常用)

结论:不推荐使用

```
// 懒汉式(线程安全，同步方法)
class Singleton {
  private static Singleton instance;

  private Singleton() {}

  //提供一个静态的公有方法，加入同步处理的代码，解决线程安全问题
  //即懒汉式
  public static Singleton getInstance() {
    if(instance == null) {
      synchronized(Singleton.class){
        instance = new Singleton();
      }
    }
    return instance;
}
```

1.6双重检查锁式

1） Double-Check 概念是多线程开发中常使用到的，如代码中所示，我们进行了两

次 if （singleton == null)检查，这样就可以保证线程安全了。

2） 这样，实例化代码只用执行一次，后面再次访问时，判断 if （singleton ==

null)，直接 return 实例化对象，也避免的反复进行方法同步.

3） 线程安全；延迟加载；效率较高

4） 结论：在实际开发中，推荐使用这种单例设计模式

volitle能保证可见性，也就是说只要一个线程对共享变量修改了，其他线程都能立即看到。当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。有序性也是一定程度的，但也并不能完全保证。因此还有个synchronized关键字。synchronized是啥，是排他锁、同步锁，也就是说同一时刻只会有一个线程获得锁，其他的都在等待锁的释放。所,synchronized就保证了有序性和原子性。因此此程序保证了可见性，原子性和有序性，就满足了并发访问。

```
// 双重检查锁式
class Singleton {
   //volitle能保证可见性
  private static volatile Singleton instance;

  private Singleton() {}

  //提供一个静态的公有方法，加入双重检查代码，解决线程安全问题, 同时解决懒加载问题
  //同时保证了效率, 推荐使用
  public static synchronized Singleton getInstance() {
    if(instance == null) {
       //synchronized保证了有序性和原子性
      synchronized (Singleton.class) {
        if(instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

1.7静态内部类式

1） 这种方式采用了类装载的机制来保证初始化实例时只有一个线程。

2） 静态内部类方式在 Singleton 类被装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 Singleton

的实例化。

3） 类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM 帮助我们

保证了线程的安全性，在类进行初始化时，别的线程是无法进入的。

4） 优点：避免了线程不安全，利用静态内部类特点实现延迟加载，效率高

5） 结论：推荐使用.

```
// 静态内部类完成， 推荐使用
class Singleton {
  private static Singleton instance;

  //构造器私有化
  private Singleton() {}

  //写一个静态内部类,该类中有一个静态属性 Singleton
  private static class SingletonInstance {
    private static final Singleton INSTANCE = new Singleton();
  }

  //提供一个静态的公有方法，直接返回SingletonInstance.INSTANCE
  public static Singleton getInstance() {
    return SingletonInstance.INSTANCE;
  }
}
```

1.8枚举式

1） 这借助 JDK1.5 中添加的枚举来实现单例模式。不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。

2） 这种方式是 Effective Java 作者 Josh Bloch 提倡的方式

3） 结论：推荐使用

```
//使用枚举，可以实现单例，推荐
enum Singleton {
    INSTANCE;
    public void sayOK() {
        System.out.println("ok~");
    }
}
```