**单例设计模式**

Singleton是一种创建型模式，指某个类采用Singleton模式，则在这个类被创建后，只可能产生一个实例供外部访问，并且提供一个全局的访问点。

核心知识点如下：

(1) 将采用单例设计模式的类的构造方法私有化（采用private修饰）。

(2) 在其内部产生该类的实例化对象，并将其封装成private static类型。

(3) 定义一个静态方法返回该类的实例。

[复制代码](javascript:void(0);)

/\*\*

\* 方法一

\* 单例模式的实现：饿汉式,线程安全 但效率比较低

\*/

public class SingletonTest {

// 定义一个私有的构造方法

private SingletonTest() {

}

// 将自身的实例对象设置为一个属性,并加上Static和final修饰符

private static final SingletonTest instance = new SingletonTest();

// 静态方法返回该类的实例

public static SingletonTest getInstancei() {

return instance;

}

}

[复制代码](javascript:void(0);)

方法一就是传说的中的饿汉模式  
优点是：写起来比较简单，而且不存在多线程同步问题，避免了synchronized所造成的性能问题；  
缺点是：当类SingletonTest被加载的时候，会初始化static的instance，静态变量被创建并分配内存空间，从这以后，这个static的instance对象便一直占着这段内存（即便你还没有用到这个实例），当类被卸载时，静态变量被摧毁，并释放所占有的内存，因此在某些特定条件下会耗费内存。

[复制代码](javascript:void(0);)

/\*\*

\*方法二

\* 单例模式的实现：饱汉式,非线程安全

\*

\*/

public class SingletonTest {  
  
 // 定义私有构造方法（防止通过 new SingletonTest()去实例化）

private SingletonTest() {

}

// 定义一个SingletonTest类型的变量（不初始化，注意这里没有使用final关键字）

private static SingletonTest instance;

// 定义一个静态的方法（调用时再初始化SingletonTest，但是多线程访问时，可能造成重复初始化问题）

public static SingletonTest getInstance() {

if (instance == null)

instance = new SingletonTest();

return instance;

}

}

[复制代码](javascript:void(0);)

方法二就是传说的中的饱汉模式  
优点是：写起来比较简单，当类SingletonTest被加载的时候，静态变量static的instance未被创建并分配内存空间，当getInstance方法第一次被调用时，初始化instance变量，并分配内存，因此在某些特定条件下会节约了内存；  
缺点是：并发环境下很可能出现多个SingletonTest实例。

[复制代码](javascript:void(0);)

/\*\*

\*方法三

\* 单例模式的实现：饱汉式,线程安全简单实现

\*

\*/

public class SingletonTest {

// 定义私有构造方法（防止通过 new SingletonTest()去实例化）

private SingletonTest() {

}

// 定义一个SingletonTest类型的变量（不初始化，注意这里没有使用final关键字）

private static SingletonTest instance;

// 定义一个静态的方法（调用时再初始化SingletonTest，使用synchronized 避免多线程访问时，可能造成重的复初始化问题）

public static synchronized SingletonTest getInstance() {

if (instance == null)

instance = new SingletonTest();

return instance;

}

}

[复制代码](javascript:void(0);)

方法三为方法二的简单优化  
优点是：使用synchronized关键字避免多线程访问时，出现多个SingletonTest实例。  
缺点是：同步方法频繁调用时，效率略低。

[复制代码](javascript:void(0);)

/\*\*

\* 方法四

\* 单例模式最优方案

\* 线程安全 并且效率高

\*

\*/

public class SingletonTest {

// 定义一个私有构造方法

private SingletonTest() {

}

//定义一个静态私有变量(不初始化，不使用final关键字，使用volatile保证了多线程访问时instance变量的可见性，避免了instance初始化时其他变量属性还没赋值完时，被另外线程调用)

private static volatile SingletonTest instance;

//定义一个共有的静态方法，返回该类型实例

public static SingletonTest getIstance() {   
 // 对象实例化时与否判断（不使用同步代码块，instance不等于null时，直接返回对象，提高运行效率）

if (instance == null) {  
 //同步代码块（对象未初始化时，使用同步代码块，保证多线程访问时对象在第一次创建后，不再重复被创建）

synchronized (SingletonTest.class) {  
 //未初始化，则初始instance变量

if (instance == null) {

instance = new SingletonTest();

}

}

}

return instance;

}

}

[复制代码](javascript:void(0);)

方法四为单例模式的最佳实现。内存占用地，效率高，线程安全，多线程操作原子性。

（事实上，可以通过Java反射机制来实例化private类型的构造方法，此时基本上会使所有的Java单例实现失效。本帖不讨论反射情况下问题，默认无反射，也是常见的面试已经应用场景）

**饿汉式和懒汉式区别**

从名字上来说，饿汉和懒汉。

饿汉就是类一旦加载，就把单例初始化完成，保证getInstance的时候，单例是已经存在的了，而懒汉比较懒，只有当调用getInstance的时候，才回去初始化这个单例。

另外从以下两点再区分以下这两种方式：

**线程安全**

饿汉式天生就是线程安全的，可以直接用于多线程而不会出现问题。

懒汉式本身是非线程安全的，为了实现线程安全有几种写法，分别是上面的1、2、3，这三种实现在资源加载和性能方面有些区别。

**资源加载和性能**

饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成。

而懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。