|  |
| --- |
| 定义：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。  1）一个类只有一个实例这是最基本的  2）它必须自行创建这个实例  3）它必须自行向整个系统提供这个实例  1）懒汉式  在类加载时，不创建实例，因此类加载速度快，但运行时获取对象的速度慢。  A.构造函数定义为私有  B.成员变量为static的，没有初始化  C.公开访问点getInstance()：public和synchronized的-----同步保证多线程时的正确性  2）饿汉式  在类加载时就完成了初始化，所以类加载较慢，但获取对象的速度快。  A.私有构造函数  B.静态私有成员--在类加载时已初始化  C.公开访问点getInstance()-----不需要同步，因为在类加载时已经初始化完毕，也不需要判断  特点：线程安全的；但是资源效率不高，可能后面没有调用getInstance()。  3）双重检查模式 （DCL）  懒汉式的加强版。  if (instance== null) {  synchronized (Singleton.class) {  if (instance== null) {  instance= new Singleton();  }  }  }  4）静态内部类单例模式  第一次加载Singleton类时并不会初始化sInstance，只有第一次调用getInstance方法时虚拟机加载SingletonHolder 并初始化sInstance。 |

单例模式的概念

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

关键点：

1）一个类只有一个实例这是最基本的

2）它必须自行创建这个实例

1. 它必须自行向整个系统提供这个实例
2. 懒汉模式（类加载时不初始化）

|  |
| --- |
| package Singleton;public class LazySingleton {  //懒汉式单例模式  //比较懒，在类加载时，不创建实例，因此类加载速度快，但运行时获取对象的速度慢 private static LazySingleton intance = null;  //静态私用成员，没有初始化  private LazySingleton() {  //私有构造函数  }  public static synchronized LazySingleton getInstance() //静态，同步，公开访问点  {  if(intance == null){  intance = new LazySingleton();  }  return intance;  }  } |

关键点：

1）构造函数定义为私有----不能在别的类中来获取该类的对象，只能在类自身中得到自己的对象

2）成员变量为static的，没有初始化----类加载快，但访问类的唯一实例慢，static保证在自身类中获取自身对象

3）公开访问点getInstance： public和synchronized的-----public保证对外公开，同步保证多线程时的正确性（因为类变量不是在加载时初始化的）

优缺点见代码注释。

2. 饿汉式单例模式（在类加载时就完成了初始化，所以类加载较慢，但获取对象的速度快）

|  |
| --- |
| package Singleton;  public class EagerSingleton {  //饿汉单例模式  //在类加载时就完成了初始化，所以类加载较慢，但获取对象的速度快    private static EagerSingleton instance = new EagerSingleton();//静态私有成员，已初始化    private EagerSingleton()  {  //私有构造函数  }    public static EagerSingleton getInstance()  //静态，不用同步（类加载时已初始化，不会有多线程的问题）  {  return instance;  }  } |

关键点：（代码注释已写）

1）私有构造函数

2）静态私有成员--在类加载时已初始化

3）公开访问点getInstance-----不需要同步，因为在类加载时已经初始化完毕，也不需要判断null，直接返回

优缺点见代码注释。

1. 双重检查模式 （DCL）

|  |
| --- |
| public class Singleton {  private volatile static Singleton instance;  private Singleton (){  }  public static Singleton getInstance() {  if (instance== null) {  synchronized (Singleton.class) {  if (instance== null) {  instance= new Singleton();  }  }  }  return singleton;  }  } |

这种写法在getSingleton方法中对singleton进行了两次判空，第一次是为了不必要的同步，第二次是在singleton等于null的情况下才创建实例。在这里用到了volatile关键字，不了解volatile关键字的可以查看Java多线程（三）volatile域这篇文章，在这篇文章我也提到了双重检查模式是正确使用volatile关键字的场景之一。

在这里使用volatile会或多或少的影响性能，但考虑到程序的正确性，牺牲这点性能还是值得的。 DCL优点是资源利用率高，第一次执行getInstance时单例对象才被实例化，效率高。缺点是第一次加载时反应稍慢一些，在高并发环境下也有一定的缺陷，虽然发生的概率很小。DCL虽然在一定程度解决了资源的消耗和多余的同步，线程安全等问题，但是他还是在某些情况会出现失效的问题，也就是DCL失效，在《java并发编程实践》一书建议用静态内部类单例模式来替代DCL。

1. 静态内部类单例模式

|  |
| --- |
| public class Singleton {  private Singleton(){  }  public static Singleton getInstance(){  return SingletonHolder.sInstance;  }  private static class SingletonHolder {  private static final Singleton sInstance = new Singleton();  }  } |

第一次加载Singleton类时并不会初始化sInstance，只有第一次调用getInstance方法时虚拟机加载SingletonHolder 并初始化sInstance ，这样不仅能确保线程安全也能保证Singleton类的唯一性，所以推荐使用静态内部类单例模式。