

单例模式

概念

- 定义：单例模式是一种**创建型设计模式**，目的是确保**一个类只有一个实例**，并提供一个**全局访问点**；在游戏开发中，单例模式常用于管理全局数据、游戏状态或者资源管理
- 实现：单例模式有两种实现方式：饿加载（Eager Initialization）和懒加载（Lazy Initialization）；区别在于实例化对象的时机
 - 饿加载：在类加载时就创建单例实例
 - 优点：实现简单，线程安全
 - 缺点：如果单例实例比较大且不一定会用到会浪费内存
 - 懒加载：在第一次需要使用时才创建单例实例
 - 优点：节省资源，只有在需要时才创建实例
 - 缺点：需要处理线程安全问题；加上线程锁每次调用时的同步检测影响性能

实例

- 饿加载：在类声明时就实例化

```
1 public class Singleton {  
2     // 在类声明时就实例化  
3     private static Singleton instance = new Singleton();  
4     private Singleton () {}  
5     // 通过getInstance()接口提供全局访问  
6     public static Singleton GetInstance() {  
7         return instance;  
8     }  
9 }
```

- 懒加载（不加线程锁）：在多线程环境下是不安全的，因为多个线程可能同时进入GetInstance()方法并通过if (instance == null)检查；这会导致多个线程同时创建多个实例
 - 假设两个线程A和B同时调用getInstance()方法
 - 线程A检查instance是否为null，发现是null
 - 线程B也检查instance是否为null，也发现是null
 - 线程A创建一个新的Singleton实例并赋值给instance

- 线程B也创建一个新的Singleton实例并赋值给instance
- 最终会有两个Singleton实例存在，违反了单例模式的原则

```
1 public class Singleton {
2     private static Singleton instance;
3     private Singleton (){}
4
5     // 通过getInstance()接口提供全局访问
6     public static Singleton GetInstance() {
7         if (instance == null) {
8             instance = new Singleton();
9         }
10        return instance;
11    }
12 }
```

- 懒加载（加线程锁）：通过在GetInstance()方法上使用synchronized关键字，确保了在多线程环境下只有一个线程能够进入该方法并创建实例，从而避免了多个线程同时创建多个实例的问题
 - 使用synchronized关键字会导致每次调用getInstance()方法时都进行同步检查，可能会影响性能

```
1 public class Singleton {
2     // 在类声明时不实例化
3     private static Singleton instance;
4     private Singleton (){}
5     // 通过getInstance()接口提供全局访问
6     public static synchronized Singleton GetInstance() {
7         if (instance == null) {
8             instance = new Singleton();
9         }
10        return instance;
11    }
12 }
```


- 懒加载（双重校验锁）：第一次检查instance是否为null时不进行同步，只有在instance为null时才进行同步检查，从而减少了不必要的同步开销，同时保证了线程安全



volatile关键字：用于确保变量的可见性和防止指令重排序

当一个线程修改了volatile修饰的变量的值，新的值会立即被刷新到主内存中，其他线程读取该变量时会得到最新的值

此外，还防止编译器和CPU对变量的读写操作进行重排序；在写入volatile变量之前的所有操作都不会被重排序到写入之后，从而确保变量在被其他线程访问之前已经初始化

 lock关键字：lock关键字用于确保一段代码在同一时间只能被一个线程执行；当一个线程进入lock块时，其他试图进入该块的线程会被阻塞，直到当前线程退出lock块

```
1 public class Singleton {
2     private static volatile Singleton instance;
3     private Singleton (){}
4
5     // 类声明时就创建一个对象充当线程锁
6     private static readonly object lockObj = new Object();
7
8     public static Singleton GetInstance() {
9         if (instance == null) {
10             lock (lockObj) {
11                 if (instance == null) {
12                     instance = new Singleton();
13                 }
14             }
15         }
16         return instance;
17     }
18 }
```

- 懒加载（C#独有的）：在C#中，Lazy<T>类提供了一种线程安全的懒加载机制，并从内部实现了线程安全

```
1 public class Singleton
2 {
3     private static readonly Lazy<Singleton> instance =
4         new Lazy<Singleton>(() => new Singleton());
5     private Singleton(){}
6     public static Singleton GetInstance()
7     {
8         return instance.Value;
9     }
10 }
```