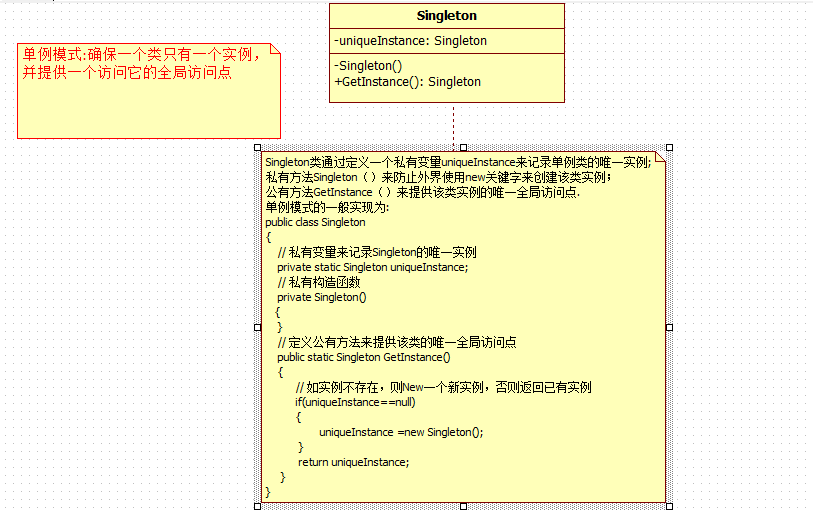
**一、引言**

最近在设计模式的一些内容，主要的参考书籍是《Head First 设计模式》，同时在学习过程中也查看了很多博客园中关于设计模式的一些文章的，在这里记录下我的一些学习笔记，一是为了帮助我更深入地理解设计模式，二同时可以给一些初学设计模式的朋友一些参考。首先我介绍的是设计模式中比较简单的一个模式——单例模式（因为这里只牵涉到一个类）

**二、单例模式的介绍**

说到单例模式,大家第一反应应该就是——什么是单例模式？，从“单例”字面意思上理解为——一个类只有一个实例，所以单例模式也就是**保证一个类只有一个实例的一种实现方法罢了(设计模式其实就是帮助我们解决实际开发过程中的方法, 该方法是为了降低对象之间的耦合度,然而解决方法有很多种,所以前人就总结了一些常用的解决方法为书籍,从而把这本书就称为设计模式)**，下面给出单例模式的一个官方定义：**确保一个类只有一个实例,并提供一个全局访问点。**为了帮助大家更好地理解单例模式,大家可以结合下面的类图来进行理解,以及后面也会剖析单例模式的实现思路:

****

**三、为什么会有单例模式**

看完单例模式的介绍,自然大家都会有这样一个疑问——为什么要有单例模式的？它在什么情况下使用的？从单例模式的定义中我们可以看出——单例模式的使用自然是当我们的系统中某个对象只需要一个实例的情况，例如:操作系统中只能有一个任务管理器,操作文件时,同一时间内只允许一个实例对其操作等,既然现实生活中有这样的应用场景,自然在软件设计领域必须有这样的解决方案了(因为软件设计也是现实生活中的抽象)，所以也就有了单例模式了。

**四、剖析单例模式的实现思路**

了解完了一些关于单例模式的基本概念之后，下面就为大家剖析单例模式的实现思路的，因为在我自己学习单例模式的时候，咋一看单例模式的实现代码确实很简单，也很容易看懂，但是我还是觉得它很陌生（这个可能是看的少的，或者自己在写代码中也用的少的缘故），而且心里总会这样一个疑问——为什么前人会这样去实现单例模式的呢？他们是如何思考的呢？后面经过自己的琢磨也就慢慢理清楚单例模式的实现思路了，并且此时也不再觉得单例模式陌生了，下面就分享我的一个剖析过程的：

我们从单例模式的概念（**确保一个类只有一个实例,并提供一个访问它的全局访问点）**入手，可以把概念进行拆分为两部分：（1）确保一个类只有一个实例；（2）提供一个访问它的全局访问点；下面通过采用两人对话的方式来帮助大家更快掌握分析思路：

菜鸟：怎样确保一个类只有一个实例了？

老鸟：那就让我帮你分析下，你创建类的实例会想到用什么方式来创建的呢？

新手：用new关键字啊，只要new下就创建了该类的一个实例了，之后就可以使用该类的一些属性和实例方法了

老鸟：那你想过为什么可以使用new关键字来创建类的实例吗？

菜鸟：这个还有条件的吗？........., 哦，我想起来了，如果类定义私有的构造函数就不能在外界通过new创建实例了（注：有些初学者就会问，有时候我并没有在类中定义构造函数为什么也可以使用new来创建对象，那是因为编译器在背后做了手脚了，当编译器看到我们类中没有定义构造函数，此时编译器会帮我们生成一个公有的无参构造函数）

老鸟：不错，回答的很对，这样你的疑惑就得到解答了啊

菜鸟：那我要在哪里创建类的实例了？

老鸟：你傻啊，当然是在类里面创建了（注：这样定义私有构造函数就是上面的一个思考过程的，要创建实例，自然就要有一个变量来保存该实例把，所以就有了私有变量的声明,**但是实现中是定义静态私有变量,朋友们有没有想过——这里为什么定义为静态的呢?对于这个疑问的解释为：每个线程都有自己的线程栈，定义为静态主要是为了在多线程确保类有一个实例**）

菜鸟：哦，现在完全明白了，但是我还有另一个疑问——现在类实例创建在类内部，那外界如何获得该的一个实例来使用它了？

老鸟：这个，你可以定义一个公有方法或者属性来把该类的实例公开出去了（注：这样就有了公有方法的定义了，该方法就是提供方法问类的全局访问点）

通过上面的分析，相信大家也就很容易写出单例模式的实现代码了，下面就看看具体的实现代码（看完之后你会惊讶道：真是这样的！）：

[复制代码](javascript:void(0);)

/// <summary>

/// 单例模式的实现

/// </summary>

public class Singleton

{

// 定义一个静态变量来保存类的实例

private static Singleton uniqueInstance;

// 定义私有构造函数，使外界不能创建该类实例

private Singleton()

{

}

/// <summary>

/// 定义公有方法提供一个全局访问点,同时你也可以定义公有属性来提供全局访问点

/// </summary>

/// <returns></returns>

public static Singleton GetInstance()

{

// 如果类的实例不存在则创建，否则直接返回

if (uniqueInstance == null)

{

uniqueInstance = new Singleton();

}

return uniqueInstance;

}

}

[复制代码](javascript:void(0);)

 上面的单例模式的实现在单线程下确实是完美的,然而在多线程的情况下会得到多个Singleton实例,因为在两个线程同时运行GetInstance方法时，此时两个线程判断(uniqueInstance ==null)这个条件时都返回真，此时两个线程就都会创建Singleton的实例，这样就违背了我们单例模式初衷了，既然上面的实现会运行多个线程执行，那**我们对于多线程的解决方案自然就是使GetInstance方法在同一时间只运行一个线程运行就好了**，也就是我们线程同步的问题了(对于线程同步大家也可以参考我[线程同步](http://www.cnblogs.com/zhili/archive/2012/07/21/ThreadsSynchronous.html)的文章),具体的解决多线程的代码如下:

[复制代码](javascript:void(0);)

/// <summary>

/// 单例模式的实现

/// </summary>

public class Singleton

{

// 定义一个静态变量来保存类的实例

private static Singleton uniqueInstance;

// 定义一个标识确保线程同步

private static readonly object locker = new object();

// 定义私有构造函数，使外界不能创建该类实例

private Singleton()

{

}

/// <summary>

/// 定义公有方法提供一个全局访问点,同时你也可以定义公有属性来提供全局访问点

/// </summary>

/// <returns></returns>

public static Singleton GetInstance()

{

// 当第一个线程运行到这里时，此时会对locker对象 "加锁"，

// 当第二个线程运行该方法时，首先检测到locker对象为"加锁"状态，该线程就会挂起等待第一个线程解锁

// lock语句运行完之后（即线程运行完之后）会对该对象"解锁"

lock (locker)

{

// 如果类的实例不存在则创建，否则直接返回

if (uniqueInstance == null)

{

uniqueInstance = new Singleton();

}

}

return uniqueInstance;

}

}

[复制代码](javascript:void(0);)

上面这种解决方案确实可以解决多线程的问题,但是**上面代码对于每个线程都会对线程辅助对象locker加锁之后再判断实例是否存在，对于这个操作完全没有必要的，因为当第一个线程创建了该类的实例之后，后面的线程此时只需要直接判断（uniqueInstance==null）为假，此时完全没必要对线程辅助对象加锁之后再去判断，所以上面的实现方式增加了额外的开销，损失了性能，为了改进上面实现方式的缺陷，我们只需要在lock语句前面加一句（uniqueInstance==null）的判断就可以避免锁所增加的额外开销，这种实现方式我们就叫它 “双重锁定”**，下面具体看看实现代码的：

[复制代码](javascript:void(0);)

/// <summary>

/// 单例模式的实现

/// </summary>

public class Singleton

{

// 定义一个静态变量来保存类的实例

private static Singleton uniqueInstance;

// 定义一个标识确保线程同步

private static readonly object locker = new object();

// 定义私有构造函数，使外界不能创建该类实例

private Singleton()

{

}

/// <summary>

/// 定义公有方法提供一个全局访问点,同时你也可以定义公有属性来提供全局访问点

/// </summary>

/// <returns></returns>

public static Singleton GetInstance()

{

// 当第一个线程运行到这里时，此时会对locker对象 "加锁"，

// 当第二个线程运行该方法时，首先检测到locker对象为"加锁"状态，该线程就会挂起等待第一个线程解锁

// lock语句运行完之后（即线程运行完之后）会对该对象"解锁"

// 双重锁定只需要一句判断就可以了

if (uniqueInstance == null)

{

lock (locker)

{

// 如果类的实例不存在则创建，否则直接返回

if (uniqueInstance == null)

{

uniqueInstance = new Singleton();

}

}

}

return uniqueInstance;

}

}

[复制代码](javascript:void(0);)

**五、C#中实现了单例模式的类**

 理解完了单例模式之后，菜鸟又接着问了：.NET FrameWork类库中有没有单例模式的实现呢？

经过查看，.NET类库中确实存在单例模式的实现类，不过该类不是公开的，下面就具体看看该类的一个实现的(该类具体存在于System.dll程序集，命名空间为System,大家可以用反射工具Reflector去查看源码的):

[复制代码](javascript:void(0);)

// 该类不是一个公开类

// 但是该类的实现应用了单例模式

internal sealed class SR

{

private static SR loader;

internal SR()

{

}

// 主要是因为该类不是公有，所以这个全部访问点也定义为私有的了

// 但是思想还是用到了单例模式的思想的

private static SR GetLoader()

{

if (loader == null)

{

SR sr = new SR();

Interlocked.CompareExchange<SR>(ref loader, sr, null);

}

return loader;

}

// 这个公有方法中调用了GetLoader方法的

public static object GetObject(string name)

{

SR loader = GetLoader();

if (loader == null)

{

return null;

}

return loader.resources.GetObject(name, Culture);

}

}

[复制代码](javascript:void(0);)

**六、总结**

到这里，设计模式的单例模式就介绍完了，希望通过本文章大家可以对单例模式有一个更深的理解，并且希望之前没接触过单例模式或觉得单例模式陌生的朋友看完之后会惊叹：原来如此！