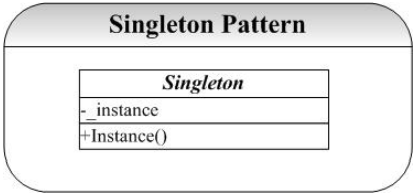
设计模式：

Singleton模式—单例模式

个人认为 Singleton 模式是设计模式中最为简单、最为常见、最容易实现，也是最应该熟悉和掌握的模式。且不说公司企业在招聘的时候为了考察员工对设计的了解和把握，考的最多的就是 Singleton 模式。

Singleton 模式解决问题十分常见，我们怎样去创建一个唯一的变量（对象？在基于对象的设计中我们可以通过创建一个全局变量（对象）来实现，在面向对象和面向过程结合的设计范式（如 C++中）中，我们也还是可以通过一个全局变量实现这一点。  
  
但是当我们遇到了纯粹的面向对象范式中，这一点可能就只能是通过 Singleton 模式来实现了，可能这也正是很多公司在招聘 Java 开发人员时候经常考 Singleton 模式的缘故吧。 Singleton 模式在开发中非常有用，我们开发过程中一些变量必须是唯一的，比如说打印机的实例等等。

**模式选择**



我们通过维护一个 static 的成员变量来记录这个唯一的对象实例。通过提供一个 staitc 的接口instance 来获得这个唯一的实例。Singleton模式经常和Factory（Abstract Factory）模式在一起使用，因为系统中工厂对象一般来说只要一个。

#### 总结

Singleton模式看起来简单，其实上面的单例也是有问题的，比如非线程安全，这里有一篇文章讨论了Singleton模式的几种写法，有兴趣可以看看。

<http://www.cnblogs.com/rush/archive/2011/10/30/2229565.html>

单例模式的六种实现：

# 1.1.1 摘要

       在我们日常的工作中经常需要在应用程序中保持一个唯一的实例，如：IO处理，数据库操作等，由于这些对象都要占用重要的系统资源，所以我们必须限制这些实例的创建或始终使用一个公用的实例，这就是我们今天要介绍的——单例模式（Singleton）。

       使用频率[clip_image001](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302124362695.gif) 高

       单件模式（Singleton）：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

# 1.1.2 正文

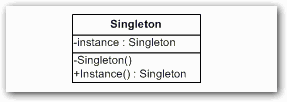
[](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302124381974.gif)

图1单例模式（Singleton）结构图

       单例模式（Singleton）是几个创建模式中最对立的一个，它的主要特点不是根据用户程序调用生成一个新的实例，而是控制某个类型的实例唯一性，通过上图我们知道它包含的角色只有一个，就是Singleton，它拥有一个私有构造函数，这确保用户无法通过new直接实例它。除此之外，该模式中包含一个静态私有成员变量instance与静态公有方法Instance()。Instance()方法负责检验并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。

[](http://images.cnblogs.com/cnblogs_com/rush/201110/20111030212442149.jpg)

图2单例模式（Singleton）逻辑模型

       接下来我们将介绍6中不同的单例模式（Singleton）的实现方式。这些实现方式都有以下的共同点：

* 1. 有一个私有的无参构造函数，这可以防止其他类实例化它，而且单例类也不应该被继承（java中final关键字），如果单例类允许继承那么每个子类都可以创建实例，这就违背了Singleton模式“唯一实例”的初衷。
  2. 单例类被定义为sealed,就像前面提到的该类不应该被继承，所以为了保险起见可以把该类定义成不允许派生，但没有要求一定要这样定义。
  3. 一个静态的变量用来保存单实例的引用。
  4. 一个公有的静态方法用来获取单实例的引用，如果实例为null即创建一个。

## 版本一线程不安全

/// <summary>

/// A simple singleton class implements.

/// </summary>

public sealed class Singleton

{

private static Singleton \_instance = null;

/// <summary>

/// Prevents a default instance of the

/// <see cref="Singleton"/> class from being created.

/// </summary>

private Singleton()

{

}

/// <summary>

/// Gets the instance.

/// </summary>

public static Singleton Instance

{

get { return \_instance ?? (\_instance = new Singleton()); }

}

}

      以上的实现方式适用于单线程环境，因为在多线程的环境下有可能得到Singleton类的多个实例。假如同时有两个线程去判断

（null == \_singleton），并且得到的结果为真，那么两个线程都会创建类Singleton的实例，这样就违背了Singleton模式“唯一实例”的初衷。

## 版本二线程安全

/// <summary>

/// A thread-safe singleton class.

/// </summary>

public sealed class Singleton

{

private static Singleton \_instance = null;

private static readonly object SynObject = new object();

Singleton()

{

}

/// <summary>

/// Gets the instance.

/// </summary>

public static Singleton Instance

{

get

{

// Syn operation.

lock (SynObject)

{

return \_instance ?? (\_instance = new Singleton());

}

}

}

}

        以上方式的实现方式是线程安全的，首先我们创建了一个静态只读的进程辅助对象，由于lock是确保当一个线程位于代码的临界区时，另一个线程不能进入临界区（同步操作）。如果其他线程试图进入锁定的代码，则它将一直等待，直到该对象被释放。从而确保在多线程下不会创建多个对象实例了。只是这种实现方式要进行同步操作，这将是影响系统性能的瓶颈和增加了额外的开销。

## Double-Checked Locking

       前面讲到的线程安全的实现方式的问题是要进行同步操作，那么我们是否可以降低通过操作的次数呢？其实我们只需在同步操作之前，添加判断该实例是否为null就可以降低通过操作的次数了，这样是经典的Double-Checked Locking方法。

/// <summary>

/// Double-Checked Locking implements a thread-safe singleton class

/// </summary>

public sealed class Singleton

{

private static Singleton \_instance = null;

// Creates an syn object.

private static readonly object SynObject = new object();

Singleton()

{

}

public static Singleton Instance

{

get

{

// Double-Checked Locking

if (null == \_instance)

{

lock (SynObject)

{

if (null == \_instance)

{

\_instance = new Singleton();

}

}

}

return \_instance;

}

}

}

       在介绍第四种实现方式之前，首先让我们认识什么是，当字段被标记为beforefieldinit类型时，该字段初始化可以发生在任何时候任何字段被引用之前。这句话听起了有点别扭，接下来让我们通过具体的例子介绍。

/// <summary>

/// Defines a test class.

/// </summary>

class Test

{

public static string x = EchoAndReturn("In type initializer");

public static string EchoAndReturn(string s)

{

Console.WriteLine(s);

return s;

}

}

      上面我们定义了一个包含静态字段和方法的类Test，但要注意我们并没有定义静态的构造函数。

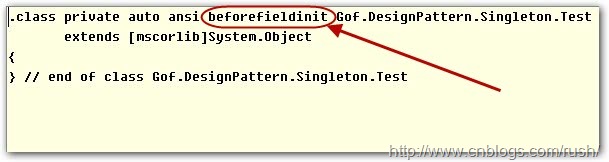
[](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302125045787.jpg)

图3 Test类的IL代码

class Test

{

public static string x = EchoAndReturn("In type initializer");

// Defines a parameterless constructor.

static Test()

{

}

public static string EchoAndReturn(string s)

{

Console.WriteLine(s);

return s;

}

}

    上面我们给Test类添加一个静态的构造函数。

[](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302125208313.jpg)

图4 Test类的IL代码

       通过上面Test类的IL代码的区别我们发现，当Test类包含静态字段，而且没有定义静态的构造函数时，该类会被标记为beforefieldinit。

       现在也许有人会问：“被标记为beforefieldinit和没有标记的有什么区别呢”？OK现在让我们通过下面的具体例子看一下它们的区别吧！

class Test

{

public static string x = EchoAndReturn("In type initializer");

static Test()

{

}

public static string EchoAndReturn(string s)

{

Console.WriteLine(s);

return s;

}

}

class Driver

{

public static void Main()

{

Console.WriteLine("Starting Main");

// Invoke a static method on Test

Test.EchoAndReturn("Echo!");

Console.WriteLine("After echo");

Console.ReadLine();

// The output result:

// Starting Main

// In type initializer

// Echo!

// After echo

}

}

     我相信大家都可以得到答案，如果在调用EchoAndReturn()方法之前，需要完成静态成员的初始化，所以最终的输出结果如下：

[](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302125453969.jpg)

图5输出结果

    接着我们在Main()方法中添加string y = Test.x，如下：

public static void Main()

{

Console.WriteLine("Starting Main");

// Invoke a static method on Test

Test.EchoAndReturn("Echo!");

Console.WriteLine("After echo");

//Reference a static field in Test

string y = Test.x;

//Use the value just to avoid compiler cleverness

if (y != null)

{

Console.WriteLine("After field access");

}

Console.ReadKey();

// The output result:

// In type initializer

// Starting Main

// Echo!

// After echo

// After field access

}

[](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302125587250.jpg)

图6 输出结果

        通过上面的输出结果，大家可以发现静态字段的初始化跑到了静态方法调用之前，Wo难以想象啊！

        最后我们在Test类中添加一个静态构造函数如下：

class Test

{

public static string x = EchoAndReturn("In type initializer");

static Test()

{

}

public static string EchoAndReturn(string s)

{

Console.WriteLine(s);

return s;

}

}

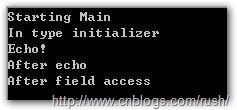
[](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302126157485.jpg)

图7 输出结果

       理论上，type initializer应该发生在”Echo!”之后和”After echo”之前，但这里却出现了不唯一的结果，只有当Test类包含静态构造函数时，才能确保type initializer的初始化发生在”Echo!”之后和”After echo”之前。

所以说要确保type initializer发生在被字段引用时，我们应该给该类添加静态构造函数。接下来让我们介绍单例模式的静态方式。

## 静态初始化

public sealed class Singleton

{

private static readonly Singleton \_instance = new Singleton();

// Explicit static constructor to tell C# compiler

// not to mark type as beforefieldinit

static Singleton()

{

}

/// <summary>

/// Prevents a default instance of the

/// <see cref="Singleton"/> class from being created.

/// </summary>

private Singleton()

{

}

/// <summary>

/// Gets the instance.

/// </summary>

public static Singleton Instance

{

get

{

return \_instance;

}

}

}

        以上方式实现比之前介绍的方式都要简单，但它确实是多线程环境下，C#实现的Singleton的一种方式。由于这种静态初始化的方式是在自己的字段被引用时才会实例化。

       让我们通过IL代码来分析静态初始化。

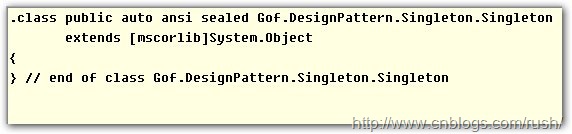
[](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302126427611.jpg)

图8静态初始化IL代码

        首先这里没有beforefieldinit的修饰符，由于我们添加了静态构造函数当静态字段被引用时才进行初始化，因此即便很多线程试图引用\_instance，也需要等静态构造函数执行完并把静态成员\_instance实例化之后可以使用。

## 延迟初始化

/// <summary>

/// Delaies initialization.

/// </summary>

public sealed class Singleton

{

private Singleton()

{

}

/// <summary>

/// Gets the instance.

/// </summary>

public static Singleton Instance { get { return Nested.\_instance; } }

private class Nested

{

// Explicit static constructor to tell C# compiler

// not to mark type as beforefieldinit

static Nested()

{

}

internal static readonly Singleton \_instance = new Singleton();

}

}

   这里我们把初始化工作放到Nested类中的一个静态成员来完成，这样就实现了延迟初始化。

## Lazy<T> type

/// <summary>

/// .NET 4's Lazy<T> type

/// </summary>

public sealed class Singleton

{

private static readonly Lazy<Singleton> lazy =

new Lazy<Singleton>(() => new Singleton());

public static Singleton Instance { get { return lazy.Value; } }

private Singleton()

{

}

}

     这种方式的简单和性能良好，而且还提供检查是否已经创建实例的属性IsValueCreated。

## 具体例子

     现在让我们使用单例模式（Singleton）实现负载平衡器，首先我们定义一个服务器类，它包含服务器名和IP地址如下：

/// <summary>

/// Represents a server machine

/// </summary>

class Server

{

// Gets or sets server name

public string Name { get; set; }

// Gets or sets server IP address

public string IP { get; set; }

}

     由于负载平衡器只提供一个对象实例供服务器使用，所以我们使用单例模式（Singleton）实现该负载平衡器。

/// <summary>

/// The 'Singleton' class

/// </summary>

sealed class LoadBalancer

{

private static readonly LoadBalancer \_instance =

new LoadBalancer();

// Type-safe generic list of servers

private List<Server> \_servers;

private Random \_random = new Random();

static LoadBalancer()

{

}

// Note: constructor is 'private'

private LoadBalancer()

{

// Load list of available servers

\_servers = new List<Server>

{

new Server{ Name = "ServerI", IP = "192.168.0.108" },

new Server{ Name = "ServerII", IP = "192.168.0.109" },

new Server{ Name = "ServerIII", IP = "192.168.0.110" },

new Server{ Name = "ServerIV", IP = "192.168.0.111" },

new Server{ Name = "ServerV", IP = "192.168.0.112" },

};

}

/// <summary>

/// Gets the instance through static initialization.

/// </summary>

public static LoadBalancer Instance

{

get { return \_instance; }

}

// Simple, but effective load balancer

public Server NextServer

{

get

{

int r = \_random.Next(\_servers.Count);

return \_servers[r];

}

}

}

  上面负载平衡器类LoadBalancer我们使用静态初始化方式实现单例模式（Singleton）。

static void Main()

{

LoadBalancer b1 = LoadBalancer.Instance;

b1.GetHashCode();

LoadBalancer b2 = LoadBalancer.Instance;

LoadBalancer b3 = LoadBalancer.Instance;

LoadBalancer b4 = LoadBalancer.Instance;

// Confirm these are the same instance

if (b1 == b2 && b2 == b3 && b3 == b4)

{

Console.WriteLine("Same instance\n");

}

// Next, load balance 15 requests for a server

LoadBalancer balancer = LoadBalancer.Instance;

for (int i = 0; i < 15; i++)

{

string serverName = balancer.NextServer.Name;

Console.WriteLine("Dispatch request to: " + serverName);

}

Console.ReadKey();

}

[](http://images.cnblogs.com/cnblogs_com/rush/201110/201110302126519228.jpg)

图9 LoadBalancer输出结果

# 1.1.3 总结

单例模式的优点：

单例模式（Singleton）会控制其实例对象的数量，从而确保访问对象的唯一性。

1. 实例控制：单例模式防止其它对象对自己的实例化，确保所有的对象都访问一个实例。
2. 伸缩性：因为由类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。

单例模式的缺点：

1. 系统开销。虽然这个系统开销看起来很小，但是每次引用这个类实例的时候都要进行实例是否存在的检查。这个问题可以通过静态实例来解决。
2. 开发混淆。当使用一个单例模式的对象的时候（特别是定义在类库中的），开发人员必须要记住不能使用new关键字来实例化对象。因为开发者看不到在类库中的源代码，所以当他们发现不能实例化一个类的时候会很惊讶。
3. 对象生命周期。单例模式没有提出对象的销毁。在提供内存管理的开发语言（比如，基于.NetFramework的语言）中，只有单例模式对象自己才能将对象实例销毁，因为只有它拥有对实例的引用。在各种开发语言中，比如C++，其它类可以销毁对象实例，但是这么做将导致单例类内部的指针指向不明。

单例适用性

使用Singleton模式有一个必要条件：在一个系统要求一个类只有一个实例时才应当使用单例模式。反之，如果一个类可以有几个实例共存，就不要使用单例模式。

不要使用单例模式存取全局变量。这违背了单例模式的用意，最好放到对应类的静态成员中。

不要将数据库连接做成单例，因为一个系统可能会与数据库有多个连接，并且在有连接池的情况下，应当尽可能及时释放连接。Singleton模式由于使用静态成员存储类实例，所以可能会造成资源无法及时释放，带来问题。

## 参考：

<http://csharpindepth.com/Articles/General/Singleton.aspx>