.NET 编程基础知识 - 文章 - 伯乐在线

一、const和readonly

当编译期常量const被编译成IL时，它就已经被替换成所代表的字面数值。所以更改一个公有的编译期常量的值，需要重新编译所有引用到该常量的代码以保证所有代码使用的是最新的常量值。
相反，运行时常量被编译成IL时引用的是readonly的变量，而不是变量的值，只需要重新编译更改了常量值的代码，就能实现对其它已经发布的代码在二进制层次上的兼容。

二、is、as

分两种情况：
转换的目标类型是引用类型：使用is测试能否转换成功，然后再用as进行转换(as在转换对象为null时会返回null)。as和is不会执行用户自定义的转换，只有当运行时类型是目标类型或者是目标的派生类型，才会转换成功。执行用户自定义的转换可以用强制转换。

转换的目标类型是值类型：不能使用as，可以使用强制转换。

三、条件编译#if #endif和Conditional特性

两者都适用于日常调试。
#if 可以穿插在函数中添加条件性代码，但使用Conditional特性可以限制在函数层面上将条件性的代码（如DEBUG）分离出来，保证代码的良好结构。

四、等同性判断

（1）等同性在数学方面的几个要点：
自反（reflexive）：表示任何对象都和其自身相等，无论a是什么类型，a==a都应该返回true；

对称（symmetric）：意味着等同性判断时的顺序是无关紧要的，若a==b返回true，那么b==a也必然返回true；

可传递（transitive）：含义是若a==b且b==c都返回true，那么a==c也必然返回true；

值相等（对应于值类型）：如果两个值类型的变量类型相同，且包含同样的内容，则“值相等”；

引用相等（对应于引用类型）：如果两个引用类型的变量指向的是同一个对象，则“引用相等”；

（2）C#中等同性判断的四个方法

public static bool ReferenceEquals(object left,object right)

无论比较的是值类型还是引用类型，该方法判断的依据都是对象标识。所以用来比较两个值类型，结果永远都是false，其原因在于装箱。在创建自己的类时，几乎不需要覆写。

public static bool Equals(object left,object right)

当不知道两个变量的运行时类型时，使用该方法进行判断。其内部实现先引用ReferenceEquals进行判断，再拿left和right与null判断，最后使用left.Equals(right)判断。在创建自己的类时，几乎不需要覆写。

public virtual bool Equals(objcet rigth)

有两种情况：

对于引用类型System.Object：使用对象标识作为判断，即比较两个对象是否“引用相等”，默认实现与ReferenceEquals完全一致。新建引用类型时无需覆写。

对于值类型System.ValueType：覆写了System.Object中的该方法，实现了判断是否”值相等“。因为无法得知当前具体的值类型，所以使用了反射，效率较低。建议在新建值类型时覆写该方法，覆写的同时也要覆写GetHashCode()方法。

public static bool operator==（MyClass left,MyClass right）

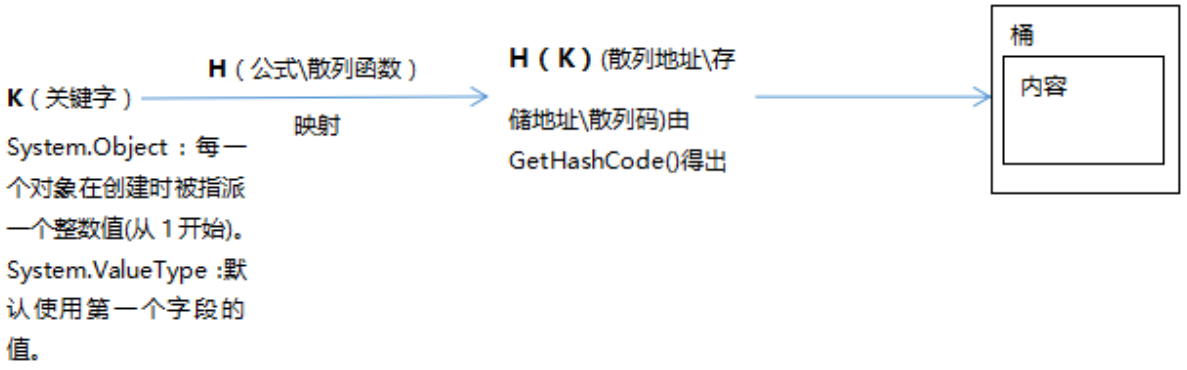
引用类型System.Object：使用对象标识作为判断。

值类型System.ValueType：覆写了System.Object中的该方法，因为无法得知当前具体的值类型，所以使用了反射，效率较低。建议在新建值类型时覆写该方法，覆写的同时也要覆写GetHashCode()方法。

五、GetHashCode()陷阱

在引用类型（System.Object）中：GetHashCode（）能正常工作，虽然它不必然会产生一个高效的分布。

在值类型（System.ValueType）中：只有在struct的第一个字段是只读的情况下，GetHashCode（）才能正常工作，只有当第一个字段包含的值有着相对随机的分布，GetHashCode（）才会产生一个比较高效的散列码。



GetHashCode覆写规则

1.如果两个对象相等（由operator==判断），那么它们必须生成相同散列码。否则，这样的散列码将无法用来查找容器中的对象。System.Object：满足此规则；System.ValueType：并非所有参与等同性判断的属性都会用来进行散列码计算，而是返回struct类型中定义的第一个字段的散列码作为当前值类型的散列码，所以可以认定等同性判断相等的散列码肯定相等（反过来，散列码相等同性判断不一定相等）。满足此规则。

2.对于任何一个对象A，A.GetHashCode()必须保持不变。不管在A上调用什么方法，A.GetHashCode()都必然总是返回同一个值。这样可以确保放在“桶”中的对象总是位于正确的“桶”中。System.Object：满足此规则；System.ValueType：struct中的第一个字段是一个常量字段，不会在生存期发生改变（称为不可变的值类型），满足此规则。

3.对于所有的输入，散列函数应该在所有整数中按照随机分布生成散列码。这样，散列容器才能得到足够的效率提升。System.Object：系统每创建一个对象时会指派一个唯一的对象键（一个整数值），从1开始，每创建一个任意类型的新对象，键值会随之增长，虽然System.Object.GetHashCode()能正常工作，但因为不是随机分布，效率不高，所以没有满足此规则，新建引用类型时建议覆写GetHashCode（）；System.ValueType：依赖于第一个字段的使用方式，如果取任意值，GetHashCode（）可以产生比较均匀的分布，但如果取相同的值则没有此满足规则。

六、委托

内建的委托形式

- Predicate：表示一个提供布尔型返回值的函数；
- Action<T1，T2>：接受任意参数目的参数；
- Func<T1，T2，ResultT>：接受零到多个参数，并返回单一结果；

.NET的委托都是多播委托（multicast delegate），多播委托将会把所有添加到该委托中的所有目标函数组合成一个单一的调用。有两点需要注意：

1. 在多播委托调用过程中，每个目标函数会被依次调用。委托对象本身不会捕获异常。因此，任何目标抛出的异常都会结束委托链的调用，如果有委托调用出现异常，那么这种方式不能保证安全；
2. 整个调用的返回值将为最后一个函数调用的返回值，前面函数的返回值将会被忽略；

解决方法：自己遍历调用列表，调用委托链上的每个目标函数

```
public void Test(Func<bool> func)
{
    bool isContinue = true;
    foreach (Func<bool> f in func.GetInvocationList())
    {
        isContinue &= f();
        if (!isContinue)
        {
            return;
        }
    }
}
```

委托应用于回调

接受Predicate<>、Action<>、Func<>为参数的方法，如List.Find(Predicate p)；有时传入lambda表达式，编译器会把lambda表达式转换成方法，然后创建一个委托，指向该方法，然后调用委托实现回调。

委托应用于事件

.NET的事件模式就是观察者模式

public event EventHandler Log;

编译器会自动创建类似下面的代码，根据需要可以自己编写

private EventHander log;

public event EventHander Log

{

add{log=log+value;}

remove{log=log-value;}

}

事件相当于一个委托集合，可以添加多个同类型委托（通过+=）；

委托可以添加多个同类型方法（通过构造函数或+=）；

七、资源管理

托管堆上的内存由GC（Garbage Collector 垃圾收集器，CLR中包含GC）进行管理，其它资源由开发者负责。

.NET 提供两种管理非托管资源生命周期的机制：终结器（finalizer，由GC调用，调用发生在对象成为垃圾之后的某个时间，时间不可预料）和IDisposable接口。

（1）GC

GC能够判断某个实体目前是否依旧被应用程序的活动对象所引用，对于那些没有被活动对象直接或间接引用的实体，GC会将其判断为垃圾。

GC会在每次运行时压缩托管堆，压缩托管堆能够将当前仍旧使用的对象放在连续的内存中，因此空余空间也是一块连续的内存。

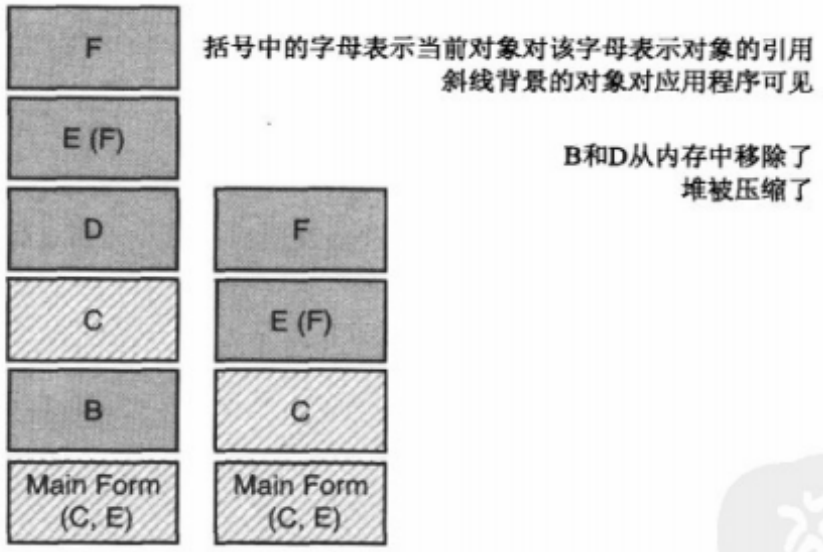


图2-1 GC不仅清理不再使用的内存，还会移动内存中的其他对象，以便压缩正使用的内存并提高可用内存空间

（2）终结器

终结器只是一种防御手段，仅仅能够保证给定类型的对象所分配的非托管资源最终被释放。GC会把需要执行终结的对象放在专门的队列中，然后让另一个线程来执行这些对象的终结器。这样，GC可以继续执行其当前的工作，在内存中移除垃圾对象，而在下一次的GC调用中才会从内存中移除这些已被终结的对象。

可以看到，需要调用终结器的对象将在内存中多停留一个GC周期的时间(实际情况会比这个更复杂一点，详情请查看下面“代”的概念)，所以应该尽量少让代码的逻辑使用到终结器。

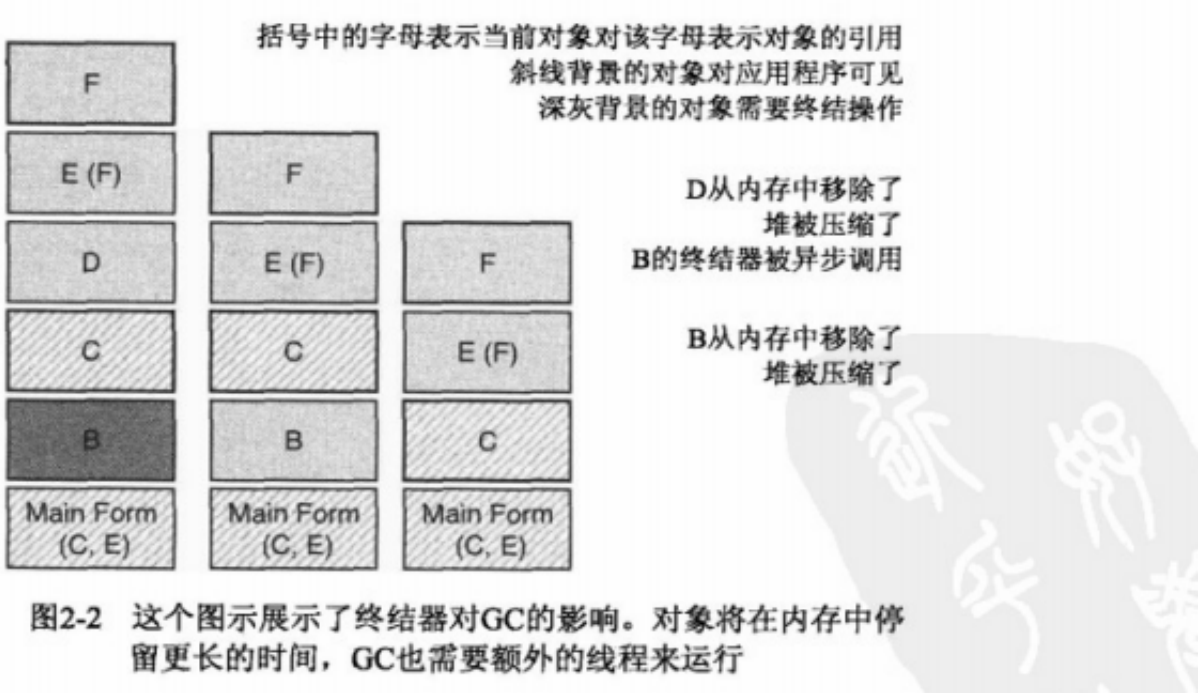


图2-2 这个图示展示了终结器对GC的影响。对象将在内存中停留更长的时间，GC也需要额外的线程来运行

GC为了优化执行，引入了“代”（generation）的概念。可以快速地找到那些更有可能是垃圾的对象。自上一次垃圾收集以来，新创建的对象属于第0代对象。若某个对象在经历过一次垃圾收集之后仍旧存活，那么将成为第1代对象。两次及两次以上垃圾收集后仍没有被销毁的对象就变成了第2代对象。

这样能将局部变量和应用程序生命周期一直使用的对象分开对待。第0代大多属于局部变量。而成员变量

和全局变量则会更快地成为第1代对象，直至第2代。GC将通过减少检查第1代和第2代对象的次数来优化执行过程。在每个周期中，GC都会检查第0代对象。一般来说，大概10个周期的GC中，会有一次去同时检查第0代和第1代对象。

大概100个周期的GC中，会有一次同时检查所有对象。可以看到一个需要总结的对象可能会比普通对象多停留9个GC周期。而若是再次GC的时候仍没有完成终结炒作，那么该对象将继续被提升为第2代。对于第2代的对象，往往需要100次以上的GC周期才会有机会被清除。为了避免这个性能问题，建议使用IDisposable接口。

（3）Dispose()和Close()

使用了非系统资源的类型会自动在终结器中调用Dispose()，以便在使用者忘记的时候仍保证能正常释放资源，但这些资源会在内存中停留更长时间，所以最好的方案还是由使用者自己显示地使用IDisposable接口的Dispose()来释放。Dispose()并不是将对象从内存中移除，而只是让对象释放掉其中的非托管资源。

Dispose()和Close()的区别(Dispose()比Close()要好一些)

Close：清理资源，对象已经不需要被终结，但一般没有调用GC.SuppressFinalize()，所以对象仍旧在终结队列中。

Dispose：清理资源，调用GC.SuppressFinalize()告知GC该对象不再需要被终结

使用IDisposable.Dispose()实现销毁非托管资源的标准销毁模式

1. 释放所有非托管资源；
2. 释放所有托管资源，包括释放事件监听程序；
3. 设定一个状态标识，表示该对象已经被销毁。若是在销毁后再次对用对象的公有方法，那么应该抛出ObjectDisposed异常；
4. 调用GC.SuppressFinalize(this)，跳过终结操作；

（4）using

using语句能以最简单的方式保证用户的对象可以正常销毁，即使对象在调用操作时出现异常。当有多个对象需要销毁时，可以使用多个using块或一个try/finally块。

using(){}=try{}finally{xxx.Dispose();}

下面例子能保证当obj不为null时正确清理到对象，当obj为null时，using(null)也不会报错，但不会做任何清理工作。

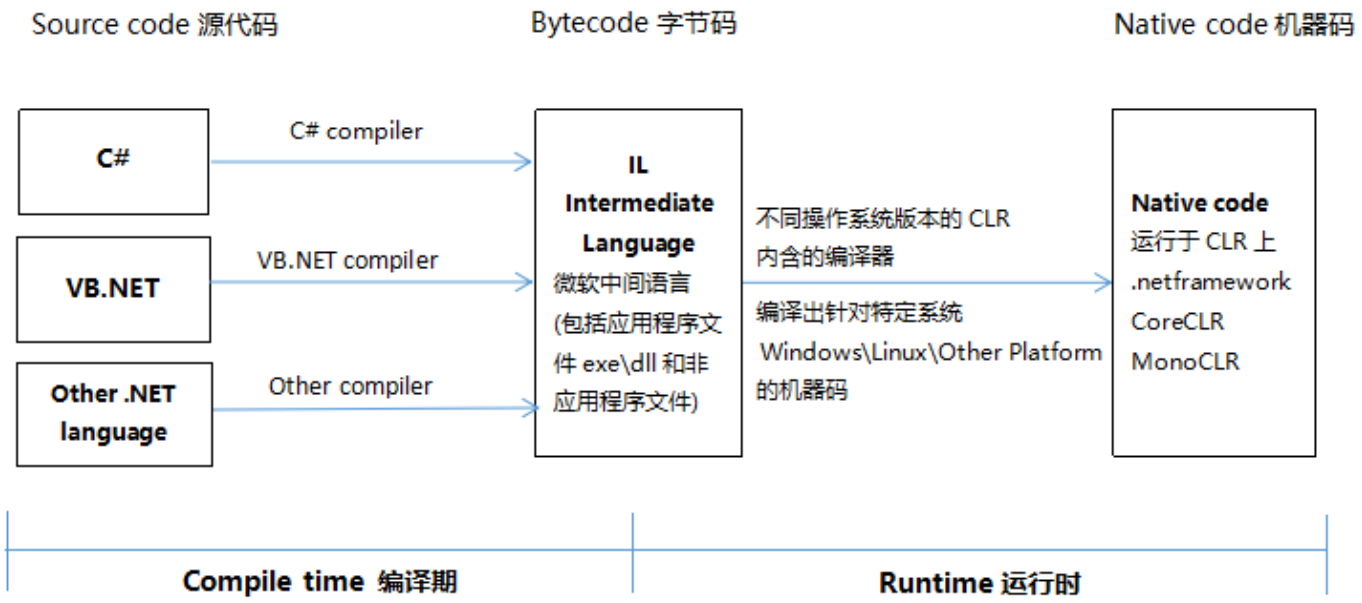
```
object obj=Factory.CreateInstance();
using(obj as IDisposable)
{
Console.Write(obj.ToString());
}
```

八、创建第一个实例所进行的操作顺序

创建某个类型的第一个实例时所进行的操作顺序，创建同样类型的第二个以及以后的实例将从第5步开始执行

1. 静态变量设置为0；
2. 执行静态变量初始化器；
3. 执行基类的静态构造函数；
4. 执行静态构造函数；
5. 实例变量设置为0；
6. 执行实例变量初始化器；
7. 执行基类中合适的实例构造函数；
8. 执行实例构造函数；

九、编译的生命周期



- 小组 - 好的话题、有启发的回复、值得信赖的圈子
- 头条 - 分享和发现有价值的内容与观点
- 相亲 - 为IT单身男女服务的征婚传播平台
- 资源 - 优秀的工具资源导航
- 翻译 - 翻译传播优秀的外文文章
- 文章 - 国内外的精选文章
- 设计 - UI,网页，交互和用户体验
- iOS - 专注iOS技术分享
- 安卓 - 专注Android技术分享
- 前端 - JavaScript, HTML5, CSS
- Java - 专注Java技术分享
- Python - 专注Python技术分享