

系列文章目录地址:

[.NET 面试题解析\(00\)-开篇来谈谈面试 & 系列文章索引](#)

弱小和无知不是生存的障碍，傲慢才是！——《三体》

常见面试题目:

1. const 和 readonly 有什么区别?
2. 哪些类型可以定义为常量? 常量 const 有什么风险?
3. 字段与属性有什么异同?
4. 静态成员和非静态成员的区别?
5. 自动属性有什么风险?
6. 特性是什么? 如何使用?
7. 下面的代码输出什么结果? 为什么?

```
List<Action> acs = new List<Action>(5);
for (int i = 0; i < 5; i++)
{
    acs.Add(() => { Console.WriteLine(i); });
}
acs.ForEach(ac => ac());
```

8. C#中的委托是什么? 事件是不是一种委托?

字段与属性的恩怨



常量

常量的基本概念就不细说了，关于常量的几个特点总结一下：

- 常量的值必须在编译时确定，简单说就是在定义是设置值，以后都不会被改变了，她是编译常量。
- 常量只能用于简单的类型，因为常量值是要被编译然后保存到程序集的元数据中，只支持基元类型，如 `int`、`char`、`string`、`bool`、`double` 等。
- 常量在使用时，是把常量的值内联到 IL 代码中的，常量类似一个占位符，在编译时被替换掉了。正是这个特点导致常量的一个风险，就是[不支持跨程序集版本更新](#)；

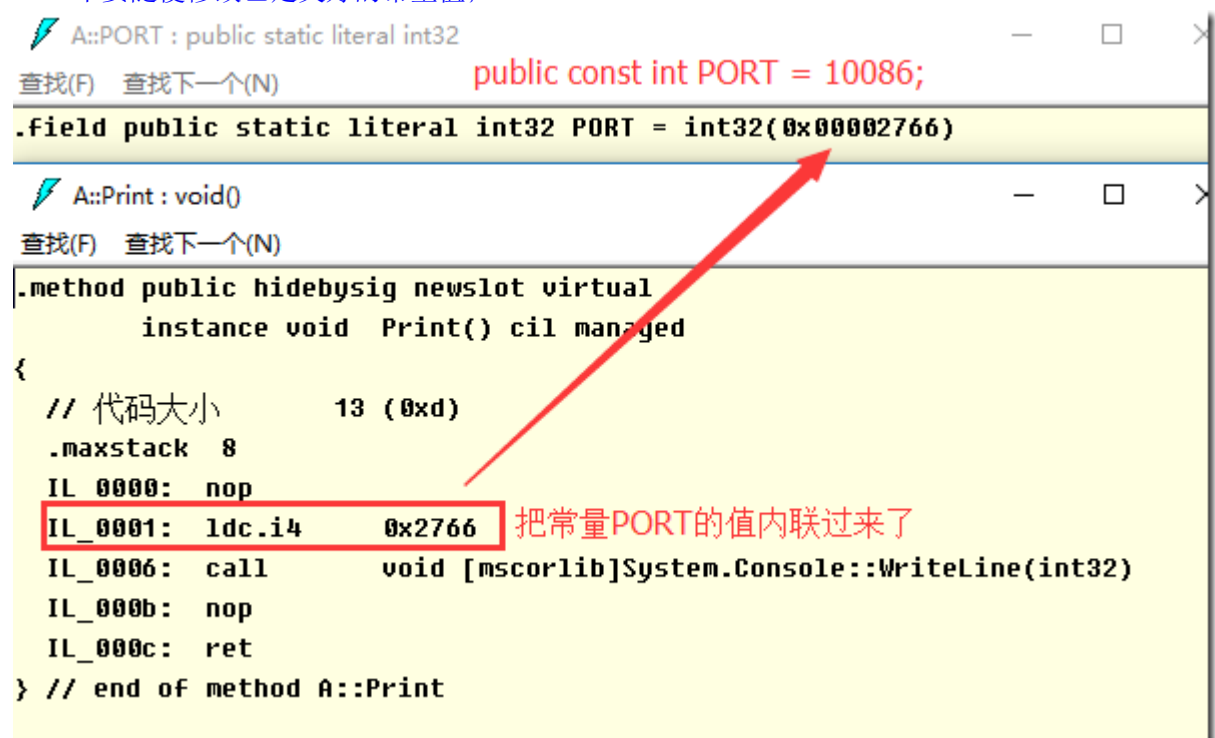
关于常量[不支持跨程序集版本更新](#)，举个简单的例子来说明：

```
public class A
{
    public const int PORT = 10086;

    public virtual void Print()
    {
        Console.WriteLine(A.PORT);
    }
}
```

上面一段非常简单代码，其生产的 IL 代码如下，在使用常量变量的地方，把她的值拷过来了（把常量的值内联到使用的地方），与常量变量 A.PORT 没有关系了。假如 A 引用了 B 程序集（B.dll 文件）中的一个常量，如果后面单独修改 B 程序集中的常量值，只是重新编译了 B，而没有编译程序集 A，就会出问题了，就是上面所说的不支持跨程序集版本更新。常量值更新后，所有使用该常量的代码都必须重新编译，这是我们在使用常量时必须要注意的一个问题。

- 不要随意使用常量，特别是有可能变化的数据；
- 不要随便修改已定义好的常量值；



```

A::PORT : public static literal int32
查找(F) 查找下一个(N) public const int PORT = 10086;
.field public static literal int32 PORT = int32(0x00002766)

A::Print : void()
查找(F) 查找下一个(N)
.method public hidebysig newslot virtual
    instance void Print() cil managed
{
    // 代码大小      13 (0xd)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldc.i4      0x2766 把常量PORT的值内联过来了
    IL_0006: call        void [mscorlib]System.Console::WriteLine(int32)
    IL_000b: nop
    IL_000c: ret
} // end of method A::Print

```

🤖 补充一下枚举的本质

接着上面的 const 说，其实枚举 enum 也有类似的问题，其根源和 const 一样，看看代码你就明白了。下面的是一个简单的枚举定义，她的 IL 代码定义和 const 定义是一样一样的啊！枚举的成员定义和常量定义一样，因此枚举其实本质上就相当是一个常量集合。

```
public enum EnumType : int
{
    None=0,
    Int=1,
    String=2,
}
```

field public static literal	valuetype CLRTest.ConsoleTest.BlogTest/EnumType None = i
EnumType::Int : public static literal	valuetype CLRTest.ConsoleTest.BlogTest/EnumType
查找(F) 查找下一个(N)	
field public static literal	valuetype CLRTest.ConsoleTest.BlogTest/EnumType Int = in
EnumType::String : public static literal	valuetype CLRTest.ConsoleTest.BlogTest/EnumType
查找(F) 查找下一个(N)	
field public static literal	valuetype CLRTest.ConsoleTest.BlogTest/EnumType String =

🤔 关于字段

字段本身没什么好说的，这里说一个字段的内联初始化问题吧，可能容易被忽视的一个小问题（不过好像也没什么影响），先看看一个简单的例子：

```
public class SomeType
{
    private int Age = 0;
    private DateTime StartTime = DateTime.Now;
    private string Name = "三体";
}
```

定义字段并初始化值，是一种很常见的代码编写习惯。但注意了，看看 IL 代码结构，一行代码（定义字段+赋值）被拆成了两块，最终的赋值都在构造函数里执行的。

```
.field private valuetype [mscorlib]System.DateTime StartTime
```

定义字段: private DateTime StartTime

SomeType::.ctor : void()

查找(F) 查找下一个(N)

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
```

```
{
```

```
// 代码大小      37 (0x25)
```

```
.maxstack 8
```

```
IL_0000: ldarg.0
```

```
IL_0001: ldc.i4.0
```

```
IL_0002: stfld      int32 CLRTest.ConsoleTest.BlogTest/SomeType::Age
```

```
IL_0007: ldarg.0
```

```
IL_0008: call      valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime
```

```
IL_000d: stfld      valuetype [mscorlib]System.DateTime CLRTest.ConsoleTest.BlogT
```

```
IL_0012: ldarg.0
```

```
IL_0013: ldstr      bytearray (09 4E 53 4F)
```

```
IL_0018: stfld      string CLRTest.ConsoleTest.BlogTest/SomeType::Name
```

```
IL_001d: ldarg.0
```

```
IL_001e: call      instance void [mscorlib]System.Object::.ctor()
```

```
IL_0023: nop
```

```
IL_0024: ret
```

```
} // end of method SomeType::.ctor
```

在构造函数里设置字段值: StartTime=DateTime.Now;

那么问题来了, 如果有多个构造函数, 就像下面这样, 有多半个构造函数, 会造成在两个构造函数.ctor 中重复产生对字段赋值的 IL 代码, 这就造成了不必要的代码膨胀。这个其实也很好解决, 在非默认构造函数后加一个“:this()”就 OK 了, 或者显示的在构造函数里初始化字段。

```
public class SomeType
{
    private DateTime StartTime = DateTime.Now;

    public SomeType() { }

    public SomeType(string name)
    {
    }
}
```

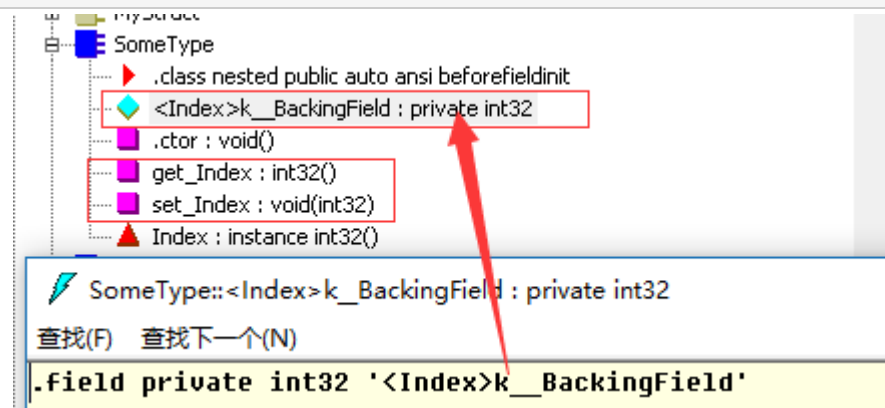


属性的本质

属性是面向对象编程的基本概念，提供了对私有字段的访问封装，在 C# 中以 `get` 和 `set` 访问器方法实现对可读可写属性的操作，提供了安全和灵活的数据访问封装。我们看看属性的本质，主要手段还是 IL 代码：

```
public class SomeType
{
    public int Index { get; set; }

    public SomeType() { }
}
```



上面定义的属性 `Index` 被分成了三个部分：

- 自动生成的私有字段“`<Index>k__BackingField`”
- 方法：`get_Index()`，获取字段值；
- 方法：`set_Index(int32 'value')`，设置字段值；

因此可以说属性的本质还是方法，使用面向对象的思想把字段封装了一下。在定义属性时，我们可以自定义一个私有字段，也可以使用自动属性“`{get; set;}`”的简化语法形式。

使用自动属性时需要注意一点的是，私有字段是由编译器自动命名的，是不受开发人员控制的。正因为这个问题，曾经在项目开发中遇到一个因此而产生的 **Bug**：

这个 **Bug** 是关于序列化的，有一个类，定义很多个（自动）属性，这个类的信息需要持久化到本地文件，当时使用了 .NET 自带的二进制序列化组件。后来因为一个需求变更，把其中一个字段修改了一下，需要把自动属性改为自己命名的私有字段的属性，就像下面实例这样。测试序列化到本地没有问题，反序列化也没问题，但最终 **bug** 还是被测试出来了，问题在与反序列化以前（修改代码之前）的本地文件时，`Index` 属性的值丢失了！！！！

```
private int _Index;
public int Index
{
    get { return _Index; }
    set { _Index = value; }
}
```

因为属性的本质是方法+字段，真正的值是存储在字段上的，字段的名称变了，反序列化以前的文件时找不到对应字段了，导致值的丢失！这也就是使用自动属性可能存在的风险。

委托与事件

什么是委托？简单来说，委托类似于 C 或 C++ 中的函数指针，允许将方法作为参数进行传递。

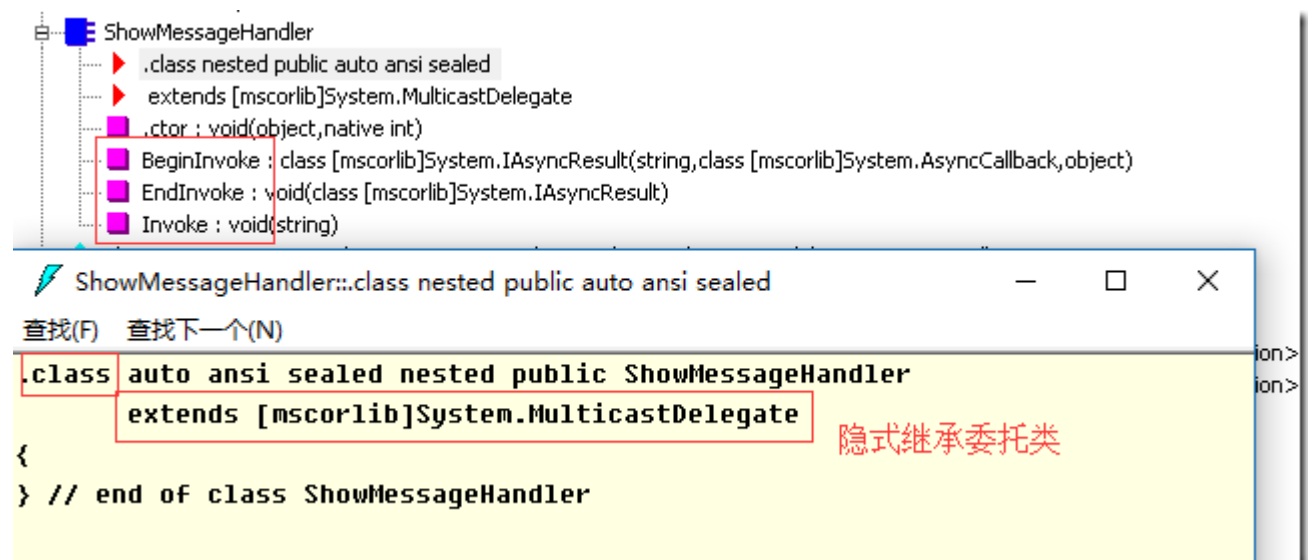
- C# 中的委托都继承自 `System.Delegate` 类型；
- 委托类型的声明与方法签名类似，有返回值和参数；
- 委托是一种可以封装命名（或匿名）方法的引用类型，把方法当做指针传递，但委托是面向对象、类型安全的；

🤖 委托的本质——是一个类

.NET 中没有函数指针，方法也不可能传递，委托之所可以像一个普通引用类型一样传递，那是因为她本质上就是一个类。下面代码是一个非常简单的自定义委托：

```
public delegate void ShowMessageHandler(string mes);
```

看看她生产的 IL 代码



我们一行定义一个委托的代码，编译器自动生成了一堆代码：

- 编译器自动帮我们创建了一个类 `ShowMessageHandler`，继承自 `System.MulticastDelegate`（她又继承自 `System.Delegate`），这是一个多播委托；
- 委托类 `ShowMessageHandler` 中包含几个方法，其中最重要的就是 `Invoke` 方法，签名和定义的方法签名一致；
- 其他两个版本 `BeginInvoke` 和 `EndInvoke` 是异步执行版本；

因此，也就不难猜测，当我们调用委托的时候，其实就是调用委托对象的 `Invoke` 方法，可以验证一下，下面的调用代码会被编译为对委托对象的 `Invoke` 方法调用：

```
private ShowMessageHandler ShowMessage;
```

```
//调用
```

```
this.ShowMessage("123");
```

```
IL_000e: ldstr      "123"
```

```
IL_0013: callvirt instance void CLRTest.ConsoleTest.BlogTest/SomeType/ShowMessage
```

```
IL_0019: pop
```

.NET 的闭包

闭包提供了一种类似脚本语言函数式编程的便捷、可以共享数据，但也存在一些隐患。

题目列表中的第 7 题，就是一个.NET 的闭包的问题。

```
List<Action> acs = new List<Action>(5);
```

```
for (int i = 0; i < 5; i++)
```

```
{
```

```
    acs.Add(() => { Console.WriteLine(i); });
```

```
}
```

```
acs.ForEach(ac => ac()); // 输出了 5 5 5 5 5，全是 5？这一定不是你想要的吧！这是为什么呢？
```

上面的代码中的 `Action` 就是.NET 为我们定义好的一个无参数无返回值的委托，从上一节我们知道委托实质是一个类，理解这一点是解决本题的关键。在这个地方委托方法共享使用了一个局部变量 `i`，那生成的类会是什么样的呢？看看 IL 代码：



共享的局部变量被提升为委托类的一个字段了：

- 变量 *i* 的生命周期延长了；
- for 循环结束后字段 *i* 的值是 5 了；
- 后面再次调用委托方法，肯定就是输出 5 了；

那该如何修正呢？很简单，委托方法使用一个临时局部变量就 OK 了，不共享数据：

```
List<Action> acss = new List<Action>(5);
for (int i = 0; i < 5; i++)
{
    int m = i;
    acss.Add(() => { Console.WriteLine(m); });
}
acss.ForEach(ac => ac()); // 输出了 0 1 2 3 4
```

至于原理，可以自己探索了！

题目答案解析：

1. const 和 readonly 有什么区别？

const 关键字用来声明编译时常量，readonly 用来声明运行时常量。都可以标识一个常量，主要有以下区别：

- 1、初始化位置不同。**const** 必须在声明的同时赋值；**readonly** 即可以在声明处赋值，也可以在构造方法里赋值。
- 2、修饰对象不同。**const** 即可以修饰类的字段，也可以修饰局部变量；**readonly** 只能修饰类的字段。
- 3、**const** 是编译时常量，在编译时确定该值，且值在编译时被内联到代码中；**readonly** 是运行时常量，在运行时确定该值。
- 4、**const** 默认是静态的；而 **readonly** 如果设置成静态需要显示声明。
- 5、支持的类型不同，**const** 只能修饰基元类型或值为 **null** 的其他引用类型；**readonly** 可以是任何类型。

2. 哪些类型可以定义为常量？常量 **const** 有什么风险？

基元类型或值为 **null** 的其他引用类型，常量的风险就是不支持跨程序集版本更新，常量值更新后，所有使用该常量的代码都必须重新编译。

3. 字段与属性有什么异同？

- 属性提供了更为强大的，灵活的功能来操作字段
- 出于面向对象的封装性，字段一般不设计为 **Public**
- 属性允许在 **set** 和 **get** 中编写代码
- 属性允许控制 **set** 和 **get** 的可访问性，从而提供只读或者可读写的功能（逻辑上只写是没有意义的）
- 属性可以使用 **override** 和 **new**

4. 静态成员和非静态成员的区别？

- 静态变量使用 **static** 修饰符进行声明，静态成员在加类的时候就被加载（上一篇中提到过，静态字段是随类型对象存放在 **Load Heap** 上的），通过类进行访问。
- 不带有 **static** 修饰符声明的变量称做非静态变量，在对象被实例化时创建，通过对象进行访问。
- 一个类的所有实例的同一静态变量都是同一个值，同一个类的不同实例的同一非静态变量可以是不同的值。
- 静态函数的实现里不能使用非静态成员，如非静态变量、非静态函数等。

5. 自动属性有什么风险？

因为自动属性的私有字段是由编译器命名的，后期不宜随意修改，比如在序列化中会导致字段值丢失。

6. 特性是什么？如何使用？

特性与属性是完全不相同的两个概念，只是在名称上比较相近。**Attribute** 特性就是关联了一个目标对象的一段配置信息，本质上是一个类，其为目标元素提供关联附加信息，这段附加

信息存储在 dll 内的元数据，它本身没什么意义。运行期以反射的方式来获取附加信息。使用方法可以参考：<http://www.cnblogs.com/anding/p/5129178.html>

7. 下面的代码输出什么结果？为什么？

```
List<Action> acs = new List<Action>(5);
for (int i = 0; i < 5; i++)
{
    acs.Add(() => { Console.WriteLine(i); });
}
acs.ForEach(ac => ac());
```

输出了 5 5 5 5 5，全是 5！因为闭包中的共享变量 i 会被提升为委托对象的公共字段，生命周期延长了

8. C#中的委托是什么？事件是不是一种委托？

什么是委托？简单来说，委托类似于 C 或 C++中的函数指针，允许将方法作为参数进行传递。

- C#中的委托都继承自 System.Delegate 类型；
- 委托类型的声明与方法签名类似，有返回值和参数；
- 委托是一种可以封装命名（或匿名）方法的引用类型，把方法当做指针传递，但委托是面向对象、类型安全的；

事件可以理解为一种特殊的委托，事件内部是基于委托来实现的。

版权所有，文章来源：<http://www.cnblogs.com/anding>