C# 笔记

郑华

2018 年 11 月 8 日

第一章　基础

面试题学习 C# Unity :https://blog.csdn.net/qq_25601345/article/details/77102775#_Toc449099466

1.1　类型装换

1.1.1　数据类型

值类型　原类型（*Sbyte*、*Byte*、*Short*、*Ushort*、*Int*、*Uint*、*Long*、*Ulong*、*Char*、*Float*、*Double*、*Bool*、*Decimal*）、枚举 *(enum)*、结构 *(struct)* 等，是**在栈中分配内存**，在申明的同时就初始化，以确保数据不为 *NULL*

引用类型　类、数组、接口 *Interface*、委托 *Delegate*、字符串 *String* 等，在**堆中分配内存**，初始化为 *NULL*

区别

- **引用型**是需要 **GARBAGE COLLECTION** 来回收内存的
- **值型**超出了作用范围，系统就会自动释放
- 值型数据传递是通过拷贝传递参数，引用型则是通过引用。

1.1.2　装箱、拆箱

装箱　**将值类型转换为引用类型**，用于在垃圾回收堆中存储值类型，是值类型到 *object* 类型或到此值类型所实现的任何接口类型的隐式转换

```
//  这是一个装箱的过程，是将值类型转换为引用类型的过程
int val = 100;
```

```
object obj = val;
```

拆箱 将引用类型转换为值类型，从 *object* 类型到值类型或从接口类型到实现该接口的值类型的显式转换

```
// 这是一个拆箱的过程，是将值类型转换为引用类型，再由引用类型转换为值类型的过程
int val = 100;
object obj = val;
int num = (int) obj;
```

装箱拆箱效率

装箱

- 首先从托管堆中为新生成的引用对象分配内存
- 然后将值类型的数据拷贝到刚刚分配的内存中
- 返回托管堆中新分配对象的地址

进行一次装箱要进行分配内存和拷贝数据这两项比较影响性能的操作。

拆箱

- 首先获取托管堆中属于值类型那部分字段的地址，这一步是严格意义上的拆箱
- 将引用对象中的值拷贝到位于线程堆栈上的值类型实例中

严格意义上的拆箱，并不影响性能，但伴随这之后的拷贝数据的操作就会同 boxing 操作中一样影响性能.

如何避免装箱 ->

学会查看 IL 反编译语言：http://www.cnblogs.com/caokai520/p/4921706.html

https://blog.csdn.net/qq_32452623/article/details/53910726

- 警惕隐式类型转换–使用合理的方式进行类型转换 `s = s + 1;`
- 使用泛型—运行时绑定数据类型, 减少装箱与拆箱 `ArrayList(非泛型)`和`List(泛型)`

1.1.3 Ref、Out

使用 ref 或 out 关键字都是通过引用传递参数的，它们的区别是：

- **使用 ref 型参数时，传入**的参数**必须先被初始化**。
- **对 out 而言，在方法中**必须 对其完成**初始化**。

使用 ref 和 out 时，在方法的参数和执行方法时，都要加 Ref 或 Out 关键字。以满足匹配. 除此之外

- **out** 适合用在**需要 retrun 多个返回值**的地方
- **ref** 则用在**需要被调用的方法修改调用者**的引用的时候

->Notice:**ref 是有进有出，而 out 是只出不进。**

1.1.4 StringBuilder 、String

String 字符串一旦创建就不可修改大小，每次使用 System.String 类中的方法之一时，都要在内存中创建一个新的字符串对象，这就需要为该新对象分配新的空间。在需要对字符串执行重复修改的情况下，与创建新的 String 对象相关的系统开销可能会非常昂贵。

string 是 c# 中的类，String 是.NET Framework 的类. C# string 映射为 Framework 的 String。如果用 string, 编译器会把它编译成 String，所以如果直接用 String 就可以让编译器少做一点点工作。

StringBuilder 如果要修改字符串而不创建新的对象，则可以使用 System.Text.StringBuilder 类，StringBuilder 可以自由扩展大小，对字符串修改比较频繁的情况使用 StringBuilder。

StringBuilder 并不会重新创建一个 string 对象，如果 stringbuilder 没有预先定义长度，默认长度为 16，大于 16 而小于 32, 时，会自动重新分配内存为 32，即 16 的倍数。使用 StringBuilder 需要预先知道长度，避免浪费空间。

1.2 类

Sealed Sealed 访问修饰符用于类时，该类是密封类，可防止其他类继承此类。

在方法中使用时则可防止派生类重写此方法。

1.3 继承

1.4 多态

1.5 预处理

1.6 C# 属性 (Property)

1.7 泛型

`foreach` 是只读的。不能一边遍历一边修改。

1.8 反射

第二章　高级

2.1　接口

IEumerable 接口　IEnumerable

只包含一个方法GetEnumerator()，它返回一个可用于循环访问集合的IEnumerator 对象，继承实现接口，完成该方法之后，就可以在调用时用foreach 了。

```
public interface IEnumerable{
    // 返回一个循环访问集合的枚举器
    IEnumerator GetEnumerator();
}
```

IEnumerator 接口　IEnumerator

它是一个真正的集合访问器 (枚举器)，没有它，就不能使用foreach语句遍历集合或数组，因为只有IEnumerator 对象才能访问集合中的项，假如连集合中的项都访问不了，那么进行集合的循环遍历是不可能的事情了。

```
public interface IEnumerator{
    // 获取集合中的当前元素
    object Current {get;}
    // 将枚举数推进到集合的下一个元素
    bool MoveNext();
    // 将枚举数设置为其初始位置，改位置位于集合中第一个元素之前
    void Reset()
}
```

参考　https://www.cnblogs.com/aehyok/p/3642103.html

2.2 枚举

2.3 Lambda

语法 创建 Lambda 表达式的简单语法形式：输入参数 => 表达式或语句块。其中，=> 为 *Lambda* 运算符，可读作“*goes to*”。

表达式 基本形式：(输入参数) => 表达式，当 *lambda* 表达式有且只有一个输入参数的时候，括号“*()*”是可选的。括号内存在多个输入参数时使用“,”进行分割。

```
( ) => true;
x => x == 1;
(x) => x == 1;
(x, y) => x == y;
```

语句 基本形式：(输入参数) => { 表达式 }

```
delegate void MyDel(string s);
// ...
MyDel myDel = n => { var s = n + "␣Fanguzai!"; Console.WriteLine(s); };
myDel("Hi,");
```

2.4 正则表达式

2.5 文件操作

2.6 属性

2.7 集合

2.8 委托

函数指针

定义委托 delegate result-type Identifier ([parameters]);

- `result-type`: 返回值的类型，和**方法**的返回值类型**一致**
- `Identifier`: 委托的名称
- `parameters`: 参数，要引用的**方法**带的**参数**

实例化委托 Identifier objectName = new Identifier(functionName)

- **Identifier：** 这个是委托名字
- **objectName：** 委托的实例化对象
- **functionName：**是该委托对象所指向的函数的名字

  -> 对于这个函数名要特别注意：定义这个委托对象肯定是在类中定义的，那么如果所指向的函数也在该类中，不管该函数是静态还是非静态的，那么就直接写函数名字就可以了；如果函数是在别的类里面定义的 *public*、*internal*，但是如果是静态，那么就直接用类名. 函数名，如果是非静态的，那么就类的对象名. 函数名，这个函数名与该对象是有关系的，比如如果函数中出现了 *this*，表示的就是对当前对象的调用。

委托推断 Identifier objectName = functionName

当我们需要定义委托对象并实例化委托的时候，就**可以只传送函数的名称**，即函数的地址.

这里面的`functionName` 与实例化委托的`functionName` 是一样的，没什么区别，满足上面的规则。

匿名委托 DelegateTest anonDel = delegate([parameters])//implements

直接定义一个 lambda 来实现临时函数。

多播委托 deleIntifier += functionX

前面使用的每个委托都只包含一个方法调用，调用委托的次数与调用方法的次数相同，如果

要调用多个方法，就需要多次给委托赋值，然后调用这个委托。**委托也可以包含多个方法，这时候要向委托对象中添加多个方法，这种委托称为多播委托**，多播委托有一个方法列表，如果调用多播委托，就可以连续调用多个方法，即先执行某一个方法，等该方法执行完成之后再执行另外一个方法，这些方法的参数都是一样的，**这些方法的执行是在一个线程中执行的**，而不是每个方法都是一个线程，**最终将执行完成所有的方法**。

多播委托包含一个逐个调用的委托集合。**如果通过委托调用的一个方法抛出了异常，整个迭代就会停止**。

委托运算符 「= 、+=、-=」

= 「Identifier objName= new Identifier(functionName)」 或者 「objName=functionName1」

这里的“=”号表示**清空** `objectName` 的方法列表，然后将 `functionName` **加入**到 `objectName` 的**方法列表**中

+= 「Identifier objName += new Identifier(functionName)」 或者 「objName+=functionName1」

这里的“+=”号表示**在原有的方法列表不变的情况下**，将 `functionName1` **加入**到 `objectName` 的方法列表中。可以在方法列表中加上多个相同的方法，执行的时候也会执行完所有的函数，哪怕有相同的，就会多次执行同一个方法。

-= 「Identifier objName -= new Identifier(functionName)」 或者 「objName-=functionName1」

这里的“-=”号表示在 `objectName` 的方法列表中减去一个`functionName1`。**可以在方法列表中多次减去相同的方法**，减一次只会减一个方法，如果列表中无此方法，那么减就没有意义，对原有列表无影响，也不会报错。

2.8.1 Action

`Action` 是无返回值的泛型委托。

- `Action` 表示无参，无返回值的委托
- `Action<int,string>` 表示有传入参数 int,string 无返回值的委托
- `Action<int,string,bool>` 表示有传入参数 int,string,bool 无返回值的委托
- `Action<int,int,int,int>` 表示有传入 4 个 int 型参数，无返回值的委托

- Action 至少 0 个参数，至多 16 个参数，无返回值。

2.9 事件

是委托 (函数指针) 的集合执行体。

2.9.1 自定义事件

声明一个委托 Delegate result-type delegateName ([parameters]);

这个委托可以在类 A 内定义也可以在类 A 外定义

```
public delegate void DelegateClick (int a);
```

声明一个基于某个委托的事件 Event delegateName eventName;

eventName 不是一个类型，而是一个具体的对象，这个具体的对象只能在类 A 内定义而不能在类 A 外定义

```
public event DelegateClick Click;
```

在类 A 中定义一个触发该事件的方法 ReturnType FunctionName（[parameters]）...

```
ReturnType FunctionName（[parameters]）
{
    If(eventName != null)
    {
        eventName([parameters]);
        或者eventName.Invoke([parameters]);
    }
}

public class butt
{
      public event DelegateClick Click;
```

```
        public void OnClick(int a)
        {
            if(Click != null)
                Click.Invoke(a);
              //Click(a);//这种方式也是可以的
            MessageBox.Show("Click();");
        }
    }
```

触发事件之后，事件所指向的函数将会被执行。这种执行是通过事件名称来调用的，就像委托对象名一样的。

触发事件的方法**只能在 A 类中定义**，事件的实例化，以及实例化之后的实现体都只能在 A 类外定义。

初始化 A 类的事件 在类 B 中定义一个类 A 的对象，并且让类 A 对象的那个事件指向类 B 中定义的方法，这个方法要与事件关联的委托所限定的方法吻合。

```
butt b = new butt();
b.Click += new DelegateClick (Fm_Click); //事件是基于委托的，所以委托推断一样适用，下面的
    语句一样有效：b.Click += Fm_Click;
```

触发 A 类的事件 在 B 类中去调用 A 类中的触发事件的方法：用 A 类的对象去调用 A 类的触发事件的方法。

```
b.OnClick(10000);
```

2.9.2 控件事件

控件事件委托 EventHandler Public delegate void EventHandler(object sender,EventArgs e);

委托 EventHandler 参数 一般第一个参数都是object sender，第二个参数可以是**任意类型**，不同的委托可以有不同的参数，**只要它派生于 EventArgs 即可**。

实例

```
    // 第1步：定义参数
    public class TimeInfoEventArgs : EventArgs
    {
        public TimeInfoEventArgs(int hour,int minute,int second)
        {
```

```
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
    public readonly int hour;
    public readonly int minute;
    public readonly int second;
}

// 第2步：定义委托
// 定义名为SecondChangeHandler的委托，封装不返回值的方法，
// 该方法带参数，一个clock类型对象参数，一个TimeInfoEventArgs类型对象
public delegate void SecondChangeHandler(object clock, TimeInfoEventArgs
    timeInformation);


// 第3步：定义事件发送者
// 被其他类观察的钟（Clock）类，该类发布一个事件：SecondChange。观察该类的类订阅了该事
    件。
public class Clock
{
    // 代表小时，分钟，秒的私有变量
    int _hour;
    public int Hour
    {
        get { return _hour; }
        set { _hour = value; }
    }
    private int _minute;
    public int Minute
    {
        get { return _minute; }
        set { _minute = value; }
    }
    private int _second;
    public int Second
    {
        get { return _second; }
        set { _second = value; }
    }

    // 要发布的事件
    public event SecondChangeHandler SecondChange;
```

```
        // 触发事件的方法
        protected void OnSecondChange(object clock,TimeInfoEventArgs timeInformation)
        {
            // Check if there are any Subscribers
            if (SecondChange != null)
            {
                // Call the Event
                SecondChange(clock, timeInformation);
            }
        }
        // 让钟（Clock）跑起来，每隔一秒钟触发一次事件
        public void Run()
        {
            for (; ; )
            {
                // 让线程Sleep一秒钟
                Thread.Sleep(1000);
                // 获取当前时间
                System.DateTime dt = System.DateTime.Now;
                // 如果秒钟变化了通知订阅者
                if (dt.Second != _second)
                {
                    // 创造TimeInfoEventArgs类型对象，传给订阅者
                    TimeInfoEventArgs timeInformation =new TimeInfoEventArgs(dt.Hour, dt.
                        Minute, dt.Second);

                    // 通知订阅者
                    OnSecondChange(this, timeInformation);
                }
                // 更新状态信息
                _second = dt.Second;
                _minute = dt.Minute;
                _hour = dt.Hour;
            }
        }
    }

/* ====================== Event Subscribers ============================== */
        // 一个订阅者。DisplayClock订阅了clock类的事件。它的工作是显示当前时间。
        public class DisplayClock
        {
            // 传入一个clock对象，订阅其SecondChangeHandler事件
            public void Subscribe(Clock theClock)
            {
```

```
            theClock.SecondChange += new SecondChangeHandler(TimeHasChanged);
        }

        // 实现了委托匹配类型的方法
        public void TimeHasChanged(object theClock, TimeInfoEventArgs ti)
          {
            Console.WriteLine("Current Time: {0}:{1}:{2}",
              ti.hour.ToString(),
              ti.minute.ToString(),
              ti.second.ToString());
          }
    }

    // 第二个订阅者，他的工作是把当前时间写入一个文件
    public class LogClock
    {
        public void Subscribe(Clock theClock)
        {
            theClock.SecondChange += new SecondChangeHandler(WriteLogEntry);
        }
        // 这个方法本来应该是把信息写入一个文件中
        // 这里我们用把信息输出控制台代替
        public void WriteLogEntry(object theClock, TimeInfoEventArgs ti)
        {
            Clock a = (Clock)theClock;
            Console.WriteLine("Logging to file: {0}:{1}:{2}",
              a.Hour.ToString(),
              a.Minute.ToString(),
              a.Second.ToString());
        }
    }

/* ======================== Test Application ================================ */
    // 测试拥有程序
    public class Test
    {
        public static void Main()
        {
            // 创建clock实例
            Clock theClock = new Clock();
            // 创建一个DisplayClock实例，让其订阅上面创建的clock的事件
            DisplayClock dc = new DisplayClock();
            dc.Subscribe(theClock);
            // 创建一个LogClock实例，让其订阅上面创建的clock的事件
```

```
            LogClock lc = new LogClock();
            lc.Subscribe(theClock);
            // 让钟跑起来
            theClock.Run();
        }
    }
```

https://blog.csdn.net/chaixinke/article/details/45396269

2.10 where T:

泛型的 Where 能够对类型参数作出限定。有以下几种方式。

- `where T : struct` 限制类型**参数 T 必须继承自 System.ValueType**。
- `where T : class` 限制类型**参数 T 必须是引用类型**，也就是不能继承自 System.ValueType。
- `where T : new()` 限制类型**参数 T 必须有一个缺省的构造函数**
- `where T : NameOfClass` 限制类型**参数 T 必须继承自某个类**或实现某个接口。