读懂IL代码就这么简单(三)完结篇

一 前言

写了两篇关于IL指令相关的文章，分别把值类型与引用类型在 堆与栈上的操作区别详细的写了一遍
这第三篇也是最后一篇，之所以到第三篇就结束了，是因为以我现在的层次，能理解到的都写完了，而且个人认为，重要的地方都差不多
写到了，
最后一篇决定把之前的内容全部整合起做一个综合的例子，然后简单的解释下IL指令的含义，及在内存中的变化
如果你没有看前两篇请狂点这里

读懂IL代码就这么简单 (一)

读懂IL代码就这么简单(二)

IL指令大全： IL指令详解

IL反编译工具： ILDasm

注：因本人水平有限，难免有理解错误之处，如有发现，望及时指出，我会立马更正。

二 IL指令详解 (基本介绍)

这次把 类 委托 方法 字段都集合起来，这样的环境就与实际的项目比较接近了，也算接地气了

先看C#代码

```
public delegate void MyDele(string name);
class Program
{
    static void Main(string[] args)
    {

        UserInfo userInfo = new UserInfo();

        PeopleStruct peopleStruct = new PeopleStruct();

        //定义委托
        MyDele myDele = userInfo.PrintName;
        //调用委托
        myDele("Delegate");

        userInfo.PrintName("PrintName");
        userInfo.PrintField();
        //静态方法
        UserInfo.ContactStr("UserInfo", "ContactStr");
        //结构的方法
        peopleStruct.PrintInfo("Color is Yellow");

        //静态类中的静态方法
        StaticUserInfo.PrintName("Static Class Static Method");

        Console.Read();
    }
}

internal class UserInfo
{
    public string Name = "UserInfo Field";

    public void PrintName(string name)
    {
        Console.WriteLine(name);
    }

    public void PrintField()
    {
        Console.WriteLine(Name);
    }

    public static void ContactStr(string Str, string Str2)
    {
        Console.WriteLine(Str + Str2);
    }

}

struct PeopleStruct
{

    public void PrintInfo(string color)
    {
        Console.WriteLine(color);
    }

}

static class StaticUserInfo
{
    public static void PrintName(string name)
    {
        Console.WriteLine(name);
    }
}
```

IL 代码

call可以调用静态方法，实例方法和虚方法

callvirt只能调用实例方法和虚方法，不能调用静态方法

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       106 (0x6a)
  .maxstack  2
  .locals init (class ILDeom3.UserInfo V_0, //只定义变量并不做任何初始化操作
           valuetype ILDeom3.PeopleStruct V_1,
           class ILDeom3.MyDele V_2)
  IL_0000:  nop
      //创建一个值类型的新对象或新实例，并将对象引用推送到计算堆栈上
  IL_0001:  newobj     instance void ILDeom3.UserInfo::.ctor()
      //把栈中顶部的元素弹出(UserInfo 的实例)并赋值给局部变量表中第0个位置的元素(V_0)
  IL_0006:  stloc.0
      //将位于特定索引处的局部变量的 "地址" 加载到计算堆栈上(将指向结构的地址压入栈中)
  IL_0007:  ldloca.s   V_1
      //初始化结构中的属性
  IL_0009:  initobj    ILDeom3.PeopleStruct
      //将局部变量列表中第0个位置(V_0 UerInfo的实例地址)的值压入栈中
  IL_000f:  ldloc.0
      //将指向实现特定方法的本机代码的非托管指针（native int 类型）推送到计算堆栈上。
      //也就是指的将方法指针压入栈中
  IL_0010:  ldftn      instance void ILDeom3.UserInfo::PrintName(string)
     //创建委托的实例并压入栈中
     //这一步会调用委托的构造器，这个构造器需要两个参数，一个对象引用，就是IL_000f:  ldloc.0压入的UserInfo的实例，一个方法的地址。
  IL_0016:  newobj     instance void ILDeom3.MyDele::.ctor(object,native int)
      //弹出栈中值(委托的实例)保存到局部变量表第2个位置(V_2)
  IL_001b:  stloc.2
      //获取局部变量列表中第2个位置上的值上一步保存的值(委托实例)，并压入栈中
  IL_001c:  ldloc.2
      //加载字符串
  IL_001d:  ldstr      "Delegate"
      //调用绑定给委托的PrintName方法
  IL_0022:  callvirt   instance void ILDeom3.MyDele::Invoke(string)
  IL_0027:  nop
      //获取局部变量列表中第0个位置上的值(UserInfo的实例)
  IL_0028:  ldloc.0
  IL_0029:  ldstr      "PrintName"
     //调用PrintName方法
  IL_002e:  callvirt   instance void ILDeom3.UserInfo::PrintName(string)
  IL_0033:  nop
      //获取局部变量列表中第0个位置上的值(UserInfo的实例)
  IL_0034:  ldloc.0
      //调用PrintField方法
  IL_0035:  callvirt   instance void ILDeom3.UserInfo::PrintField()
  IL_003a:  nop
  IL_003b:  ldstr      "UserInfo"
  IL_0040:  ldstr      "ContactStr"
      //因为ContactStr是静态方法所以不需要先加载实例可以直接调用
  IL_0045:  call       void ILDeom3.UserInfo::ContactStr(string,
                                                         string)
  IL_004a:  nop
      //将位于特定索引处的局部变量的 "地址" 加载到计算堆栈上 （将指向结构的地址压入栈中）
  IL_004b:  ldloca.s   V_1
  IL_004d:  ldstr      "Color is Yellow"
      //调用结构中的PrintInfo方法
  IL_0052:  call       instance void ILDeom3.PeopleStruct::PrintInfo(string)
  IL_0057:  nop
  IL_0058:  ldstr      "Static Class Static Method"
  IL_005d:  call       void ILDeom3.StaticUserInfo::PrintName(string)
  IL_0062:  nop
  IL_0063:  call       int32 [mscorlib]System.Console::Read()
  IL_0068:  pop
  IL_0069:  ret
} // end of method Program::Main
```

相信有注释，大家应该都是能够看懂的，IL其实并不难，也并不算底层，只是把C#编译成了中间语言，并非机器语言，CPU照样还是读不懂，

三 IL指令详解 (深入了解)

因这次IL指令，有点长，要画图确实有点扛不住，所以只画重要的地方，还望见谅.

另外 跟园子里的 @冰麟轻武 探讨了跟IL相关的三个内存块 Managed Heap ,Evaluation Stack,Call Stack 了解到了很多之前不明白的知识点，

也纠正了自己以前的一些误区，最后一致认可我们自己的讨论结果，讨论结果如下，

1 Managed Heap(托管堆) 程序运行时会动态的在其中开辟空间来存储变量的值，如new class 时，回收由GC 根据 代龄，和可达对象，来回收相应的内存资源。整个程序共用一个ManagedHeap

2 Evaluation Stack(计算栈)：每个线程都有一个独立的 评估栈，用于程序相关的运算，

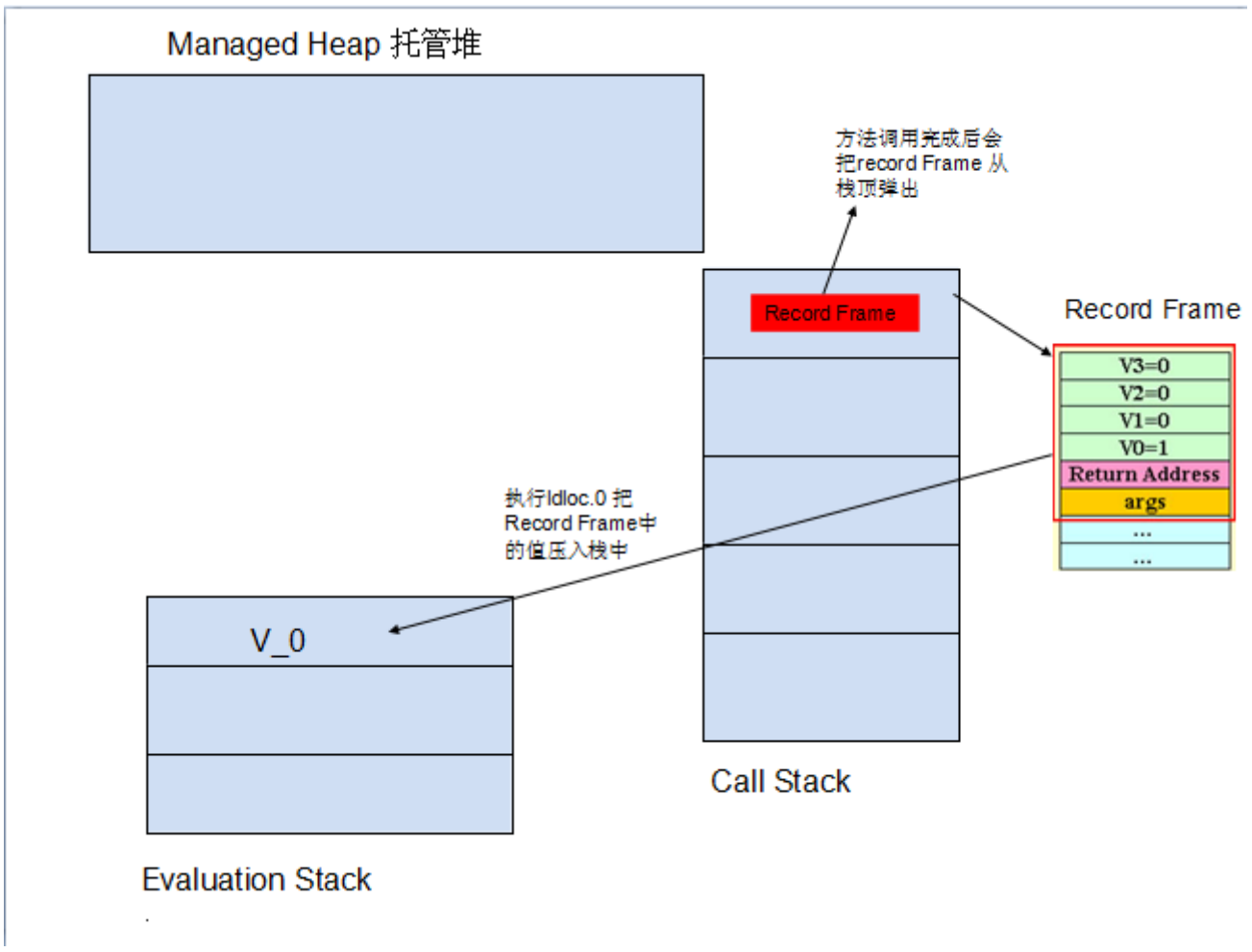
3 Call Stack(调用栈)：讨论的重点就在这里，之前认为Call Stack并不是一个栈，而是一个局部变量列表，用于存放方法的参数，可是我一直有疑问就是值类型应该是存在栈中的，如果Call Stack是个栈，那取值时Call Stack并没有按FILO的原则来，那如果 Call Stack不是个栈那值类型的值 是存在哪里的，然后我与@冰麟轻武就这一问题，讨论起来了

先看官方对Call Stack的解释： 这是由.NET CLR在执行时自动管理的存储单元，每个Thread都有自己专门的Call Stack。每呼叫一次method，就会使得Call Stack上多一个Record Frame；方法执行完毕之后，此Record Frame会被丢弃。重点就在红色这一句中的 Record Frame又是个什么东西他里边有什么东西？然后开始各种假设，最终我们认为这一种理论是比较靠谱一点的如下：

Call Stack本身就是一个栈，每调用一个方法时就会在栈顶部加载一个Record Frame，这个Record Frame里包含了方法所需要的参数(Params)，返回地址（Return Address）和区域变量(Local Variable)，当调用的方法结束时，就自动会把这个Record Frame从栈顶弹出。如此一来，我之前的疑问就可以得到相应的解释了

值类型是存在栈中的，当调用方法里会把方法需要的值重栈中取出，然后在栈中创建一个Record Frame并把赋值给Record Frame中的参数，在这个Record Frame中取数据并不是按FILO原则来的，而可以按索引，也可以按地址 对应IL指令 Ldloc stLoc 等取值与赋值都是针对的Record Frame 。而且我们认为Call Stack是对线程栈的一个统称。

上图

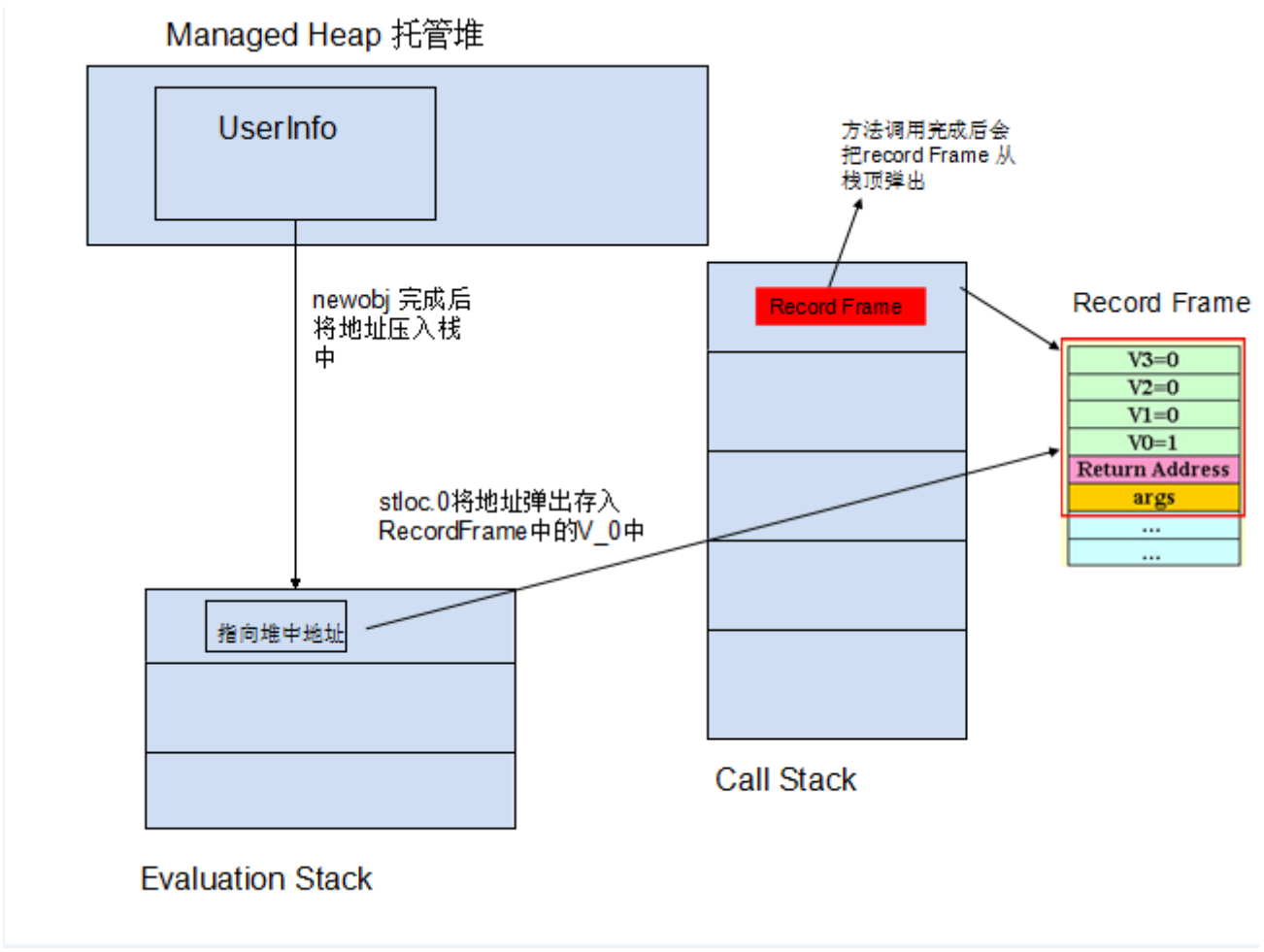


下面图解一下实例化一个类，并调用类中的方法在内存中是如何变化的

.locals init (class ILDeom3.UserInfo V_0,valuetype ILDeom3.PeopleStruct V_1,class ILDeom3.MyDele V_2)

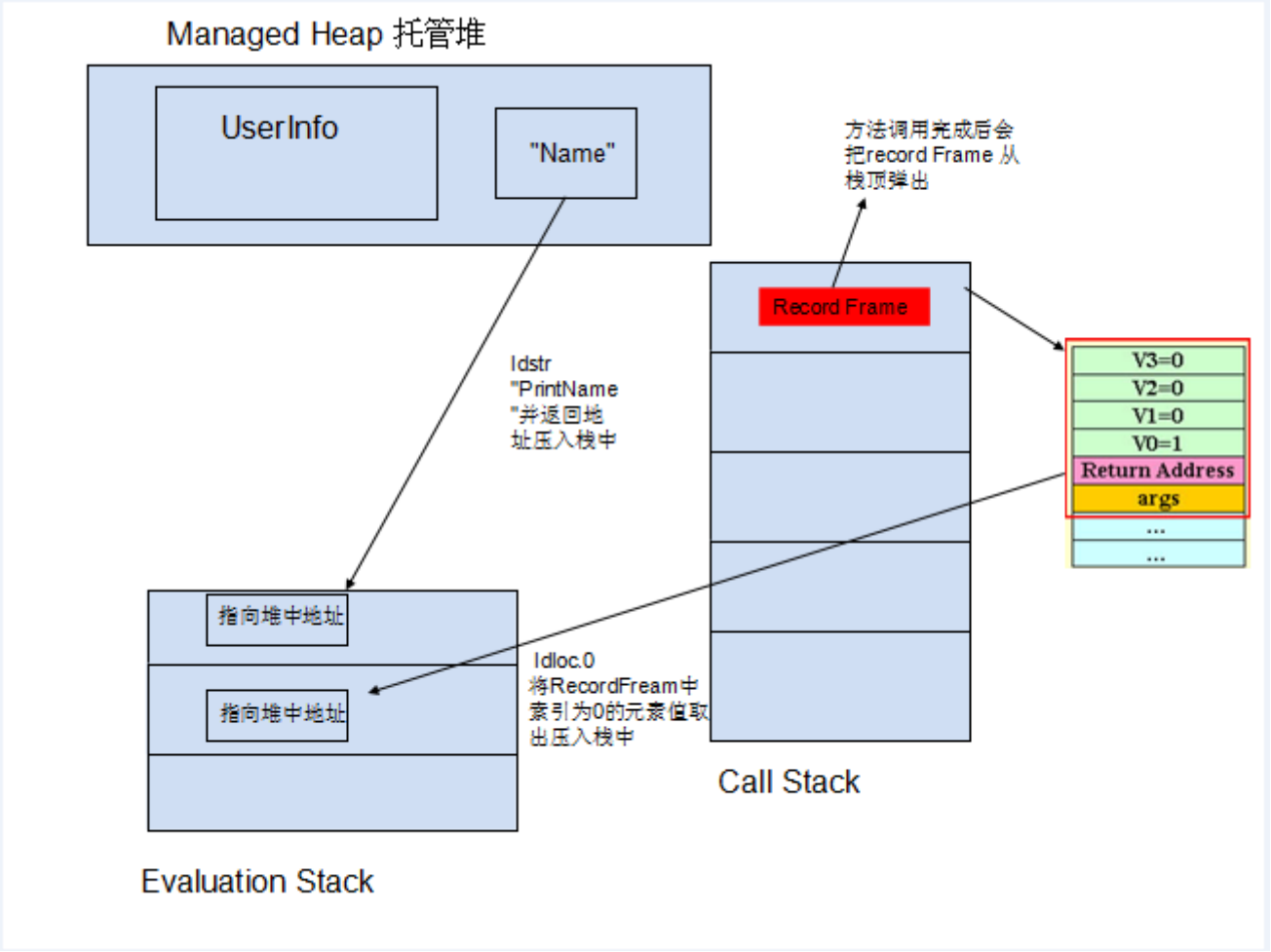
IL_0001: newobj instance void ILDeom3.UserInfo::.ctor()

IL_0006: stloc.0



IL_0028: ldloc.0

IL_0029: ldstr "PrintName"
IL_002e: callvirt instance void ILDeom3.UserInfo::PrintName(string)



四 总结

IL系列终于写完了，也算给自己一个交代了，写文章真的很花时间，就以我这三篇为例，光只是写和画图都有花十几个小时，而且如果是晚上写一般都会超过12点才能完成，更不用说前期的自己学习所用的时间，

但是我觉得真的很值得，充分的把自己的业余时间利用起来了，对于IL也有了一个相对深入的了解，

在此要感谢 园子里朋友的支持，也感谢 @冰麟轻武对我的指点，更要感谢dudu能建立博客园这么好的一个环境。👈

如果您觉得本文有给您带来一点收获，不妨点个**推荐**，为我的付出支持一下，谢谢~

如果希望在技术的道路上能有更多的朋友，那就**关注下我吧**，让我们一起在技术的路上奔跑

原文地址：https://www.cnblogs.com/zery/p/3386898.html

热门文章

在FP与DDD的道路上越走越远
Scala函数式编程——近半年的痛并快乐着
《七周七并发模型》——痛不欲生却欲罢不能
改善用户故事的50个点子
对实例化需求方法的整理与思考
BDD测试框架Spock概要
我为什么想并且要学习Scala
理解Scala中的Extractor
对结合BDD进行DDD开发的一点思考和整理
行为驱动开发BDD概要