Revisions to previous discussion of the implementation of anonymous methods in C#

devblogs.microsoft.com/oldnewthing/20170717-00

July 17, 2017

Raymond Chen

Welcome to CLR Week.

Yes, it's been a long time since the last CLR Week. Some people might consider that a feature.

Anyway, I'm going to start by calling attention to some revisions to previous discussion of the implementation of anonymous methods in C#.

- Part 1
- Part 2
- Part 3

The first revision is one most people are well aware of, namely that the scope of the control variable of a `foreach` statement is now the controlled statement. What this means for you is that closing over the loop control variable of a `foreach` statement is not dangerous. Note, however, that closing over the loop control variable of a `for` statement is still dangerous.

The second revision is that noncapturing lambdas are no longer wrappers around a static method. Even if the lambda captures nothing, it is still converted to an instance method (of an anonymous type).

The reason given by Kevin Pilch-Bisson is that "delegate invokes are optimized for instance methods and have space on the stack for them. To call a static method they have to shift parameters around."

Let's unpack that explanation.

Recall that instance methods have a hidden `this` parameter, whereas static methods do not. Suppose you want to forward a call from one method to another. For concreteness, let's say you have

```
class C1
{
 public void M1(int x, int y, int z)
 {
  System.Console.WriteLine("From {0} to {1} via {2}", x, y, z);
 }
 static public void S1(int x, int y, int z)
 {
  System.Console.WriteLine("From {0} to {1} via {2}", x, y, z);
 }
}

class C2
{
 private C1 c1 = new C1();
 static private C1 s1 = new C1();

 public void M2(int x, int y, int z)
 {
  c1.M1(x, y, z);
 }
 static public void S2(int x, int y, int z)
 {
  C1.S1(x, y, z);
 }
 public void M2S(int x, int y, int z)
 {
  C1.S1(x, y, z);
 }
 static public void S2M(int x, int y, int z)
 {
  s1.M1(x, y, z);
 }
}
```

Since the layouts for the parameters to both `C1.M1()` and `C2.M2()` method match, `C2.M2()` can get away with the following:

- Fetch `this.c1`.
- Validate that the fetched value is not null.
- Replace `this` with the fetched value.
- Jump to `C1.M1`.

The assembly for `C2.M2` on x86 would go something like this:

```
; fastcall convention passes
; the first parameter (this) in ecx
; the second parameter (x) in edx
; remaining parameters (y, z) on the stack

C2.M2:
    mov  ecx, [ecx].c1  ; fetch this.c1
    cmp  ecx, [ecx]     ; null check
    jmp  C1.M1          ; all the other parameters are already set
```

Similarly, forwarding a call from one static method to another can reuse the stack frame as-is:

```
C2.S2:
    jmp C1.S1           ; all parameters are already set properly
```

However, forwarding from an instance method to a static method or vice versa isn't so easy. The compiler would either have to generate a traditional non-tail call:

```
C2.M2S:
    mov  ecx, edx       ; put x into ecx
    mov  edx, [esp][4]  ; put y into edx
    push edx, [esp][8]  ; push z
    call C1.S1
    ret  8

C2.S2M:
    push [esp][4]       ; push z
    push edx            ; push y
    mov  edx, ecx       ; put x into edx
    mov  ecx, [C2.s1]   ; put C2.s1 into ecx
    cmp  ecx, [ecx]     ; null check
    call C1.M1          ; call it
    ret  8
```

Or maybe the compiler plays funny stack rewriting games:[1]

```
C2.M2S:
    mov  ecx, edx       ; put x into ecx
    pop  eax            ; pop return address
    pop  edx            ; pop y into edx
                        ; leave z on the stack
    push eax            ; restore return address
    jmp  C1.S1

C2.S2M:
    pop  eax            ; pop return address
    push edx            ; push y
    push eax            ; restore return address
    mov  edx, ecx       ; put x into edx
    mov  ecx, [C2.s1]   ; put C2.s1 into ecx
    cmp ecx, [ecx]      ; null check
    jmp C1.M1
```

Both of these are worse than the case where the call is forwarded to a function of matching ilk.

Since delegate invoke is done instance-style, the code to dispatch the delegate to the lambda is more efficient if the lambda is also instance.

Since the language specification does not specify the nature of the lambda, whether the delegate represents a static or instance method is an implementation detail that can change at any time.

And it did.

[1] Note that these stack rewriting games are not available to x64 because of alignment requirements. On x64, we are forced to generate a traditional non-tail call.

Raymond Chen

Follow