The implementation of anonymous methods in C# and its consequences (part 2)

devblogs.microsoft.com/oldnewthing/20060803-00

August 3, 2006

Raymond Chen

Last time we took a look at how anonymous methods are implemented. Today we'll look at a puzzle that can be solved with what we've learned. Consider the following program fragment:

```
using System;
class MyClass {
 delegate void DelegateA();
 delegate void DelegateB();
 static DelegateB ConvertDelegate(DelegateA d)
 {
  return (DelegateB)
   Delegate.CreateDelegate(typeof(DelegateB), d.Method);
 }
 static public void Main()
 {
  int i = 0;
  ConvertDelegate(delegate { Console.WriteLine(0); });
 }
}
```

The `ConvertDelegate` method merely converts a `DelegateA` to a `DelegateB` by creating a `DelegateB` with the same underlying method. Since the two delegate types use the same signature, this conversion goes off without a hitch.

But now let's make a small change to that `Main` function:

```
 static public void Main()
 {
  int i = 0;
  // one character change - 0 becomes i
  ConvertDelegate(delegate { Console.WriteLine(i); });
 }
```

Now the program crashes with a `System.ArgumentException` at the point where we try to create the `DelegateB`. What's going on?

First, observe that the overload of `Delegate.CreateDelegate` that was used is one that can only be used to create delegates from static methods. Next, note that in `Test1`, the anonymous method references neither its own members nor any local variables from its lexically-enclosing method. Therefore, the resulting anonymous method is a "static anonymous method of the easy type". Since the anonymous method is a static member, the use of the "static members only" overload of `Delegate.CreateDelegate` succeeds.

However, in `Test2`, the anonymous method dereferences the `i` variable from its lexically-enclosing method. This forces the anonymous method to be a "anonymous method of the hard type", and those anonymous methods use an anonymous instance member function of an anonymous helper class. As a result, `d.Method` is an instance method, and the chosen overload of `Delegate.CreateDelegate` throws an invalid parameter exception since it works only with static methods.

The solution is to use a different overload of `Delegate.CreateDelegate`, one that work with either static or instance member functions.

```
DelegateB ConvertDelegate(DelegateA d)
{
 return (DelegateB)
  Delegate.CreateDelegate(typeof(DelegateB), d.Target, d.Method);
}
```

The `Delegate.CreateDelegate(Type, Object, MethodInfo)` overload creates a delegate for a static method if the `Object` parameter is `null` or a delegate for an instance method if the `Object` parameter is non-`null`. Hardly by coincidence, that is exactly what `d.Target` produces. If the original delegate is for a static method, then `d.Target` is `null`; otherwise, it is the object for which the instance method is to be invoked on.

This fix, therefore, makes the `ConvertDelegate` function handle conversion of delegates for either static or instance methods. Which is a good thing, because it may now be called upon to convert delegates for instance methods as well as static ones.

Okay, this time we were lucky that the hidden gotcha of anonymous methods resulted in an exception. Next time, we'll see a gotcha that merely results in incorrect behavior that will probably take you forever to track down.

Update: This behavior changed in Visual Studio 2015 with the switch to the Roslyn compiler. For performance reasons, anonymous methods are now always instance methods, even if they capture nothing.

Raymond Chen

Follow