The implementation of iterators in C# and its consequences (part 1)

devblogs.microsoft.com/oldnewthing/20080812-00

August 12, 2008

Raymond Chen

Like anonymous methods, iterators in C# are very complex syntactic sugar. You could do it all yourself (after all, you *did* have to do it all yourself in earlier versions of C#), but the compiler transformation makes for much greater convenience.

The idea behind iterators is that they take a function with `yield return` statements (and possible some `yield break` statements) and convert it into a state machine. When you `yield return`, the state of the function is recorded, and execution resumes from that state the next time the iterator is called upon to produce another object.

Here's the basic idea: All the local variables of the iterator (treating iterator parameters as pre-initialized local variables, including the hidden `this` parameter) become member variables of a helper class. The helper class also has an internal *state* member that keeps track of where execution left off and an internal *current* member that holds the object most recently enumerated.

```
class MyClass {
 int limit = 0;
 public MyClass(int limit) { this.limit = limit; }


 public IEnumerable<int> CountFrom(int start)
 {
  for (int i = start; i <= limit; i++) {
   yield return i;
  }
 }
}
```

The `CountFrom` method produces an integer enumerator that spits out the integers starting at `start` and continuing up to and including `limit`. The compiler internally converts this enumerator into something like this:

```
class MyClass_Enumerator : IEnumerable<int> {
 int state$0 = 0;// internal member
 int current$0;  // internal member
 MyClass this$0; // implicit parameter to CountFrom
 int start;      // explicit parameter to CountFrom
 int i;          // local variable of CountFrom


 public int Current {
  get { return current$0; }
 }


 public bool MoveNext()
 {
  switch (state$0) {
  case 0: goto resume$0;
  case 1: goto resume$1;
  case 2: return false;
  }


resume$0:;
  for (i = start; i <= this$0.limit; i++) {
   current$0 = i;
   state$0 = 1;
   return true;
resume$1:;
  }


  state$0 = 2;
  return false;
 }
 … other bookkeeping, not important here …
}


public IEnumerable<int> CountFrom(int start)
{
 MyClass_Enumerator e = new MyClass_Enumerator();
 e.this$0 = this;
 e.start = start;
 return e;
}
```

The enumerator class is auto-generated by the compiler and, as promised, it contains two internal members for the state and current object, plus a member for each parameter (including the hidden `this` parameter), plus a member for each local variable. The `Current` property merely returns the current object. All the real work happens in `MoveNext`.

To generate the `MoveNext` method, the compiler takes the code you write and performs a few transformations. First, all the references to variables and parameters need to be adjusted since the code moved to a helper class.

- `this` becomes `this$0`, because inside the rewritten function, `this` refers to the auto-generated class, not the original class.
- `m` becomes `this$0.m` when `m` is a member of the original class (a member variable, member property, or member function). This rule is actually redundant with the previous rule, because writing the name of a class member `m` without a prefix is just shorthand for `this.m`.
- `v` becomes `this.v` when `v` is a parameter or local variable. This rule is actually redundant, since writing `v` is the same as `this.v`, but I call it out explicitly so you'll notice that the storage for the variable has changed.

The compiler also has to deal with all those `yield return` statements.

> Each `yield return x` becomes
>
> ```
> current$0 = x;
> state$0 = n;
> return true;
> resume$n:;
> ```
>
> where `n` is an increasing number starting at 1.

And then there are the `yield break` statements.

> Each `yield break` becomes
>
> ```
> state$0 = n2;
> return false;
> ```
>
> where `n2` is one greater than the highest state number used by all the `yield return` statements. Don't forget that there is also an implied `yield break` at the end of the function.

Finally, the compiler puts the big state dispatcher at the top of the function.

At the start of the function, insert

```
switch (state$0) {
case 0: goto resume$0;
case 1: goto resume$1;
case 2: goto resume$2;
…
case n: goto resume$n;
case n2: return false;
}
```

with one `case` statement for each state, plus the initial zero state and the final `n2` state.

Notice that this transformation is quite different from the enumeration model we built based on coroutines and fibers. The C# method is far more efficient in terms of memory usage since it doesn't consume an entire stack (typically a megabyte in size) like the fiber approach does. Instead it just borrows the stack of the caller, and anything that it needs to save across calls to `MoveNext` are stored in a helper object (which goes on the heap rather than the stack). This fake-out is normally quite effective—most people don't even realize that it's happening—but there are places where the difference is significant, and we'll see that shortly.

Exercise: Why do we need to write `state$0 = n2;` and add the `case n2: return false;` ? Why can't we just transform each `yield break` into `return false;` and stop there?

Raymond Chen

Follow