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

devblogs.microsoft.com/oldnewthing/20080813-00

August 13, 2008

Raymond Chen

Now that you have the basic idea behind iterators under your belt, you can already answer some questions on iterator usage. Here's a scenario based on actual events:

> I have an iterator that is rather long and complicated, so I'd like to refactor it. For illustrative purposes, let's say that the enumerator counts from 1 to 100 twice. (In real life, of course, the iterator will not be this simple.)
>
> ```
> IEnumerable<int> CountTo100Twice()
> {
>  int i;
>  for (i = 1; i <= 100; i++) {
>   yield return i;
>  }
>  for (i = 1; i <= 100; i++) {
>   yield return i;
>  }
> }
> ```
>
> As we learned in Programming 101, we can pull common code into a subroutine and call the subroutine. But when I do this, I get a compiler error:
>
> ```
> IEnumerable<int> CountTo100Twice()
> {
>  CountTo100();
>  CountTo100();
> }
>
>
> void CountTo100()
> {
>  int i;
>  for (i = 1; i <= 100; i++) {
>   yield return i;
>  }
> }
> ```
>
> What am I doing wrong? How can I move the "count to 100" into a subroutine and call it twice from the `CountTo100Twice` function?

As we saw last time, iterators are not coroutines. The technique above would have worked great had we built iterators out of, say, fibers instead of building them out of state machines. As state machines, all `yield return` statements must occur at the "top level". So how do you iterate with the help of subroutines?

You make the subroutine its own iterator and suck the results out from the main function:

```
IEnumerable<int> CountTo100Twice()
{
 foreach (int i in CountTo100()) yield return i;
 foreach (int i in CountTo100()) yield return i;
}


IEnumerable<int> CountTo100()
{
 for (i = 1; i <= 100; i++) {
  yield return i;
 }
}
```

Exercise: Consider the following fragment:

```
 foreach (int i in CountTo100Twice()) {
  …
 }
```

Explain what happens on the 150th call to `MoveNext()` in the above loop. Discuss its consequences for recursive enumerators (such as tree traversal).

Raymond Chen

Follow