Why the compiler can’t autoconvert foreach to for

devblogs.microsoft.com/oldnewthing/20040421-00

April 21, 2004

Raymond Chen

People have discovered that the “natural” C# loop construct

```
ArrayList list = ...;
foreach (Object o in list) {
  ... do something with o ...
}
```

is fractionally slower than the corresponding manual loop:

```
ArrayList list = ...;
for (int i = 0; i < list.Length; i++) {
    Object o = list[i];
  ... do something with o ...
}
```

The first thing that needs to be said here is that

> The performance difference is almost certainly insignificant.

Don’t go running around changing all your **foreach** loops into corresponding **for** loops thinking that your program will magically run faster. It almost certainly won’t, because loop overhead is rarely where a non-benchmark program spends most of its time.

My topic for today is not how to make your code faster by abandoning your **foreach** loops. My topic is to answer the question, “Why doesn’t the compiler autoconvert the **foreach** into the corresponding **for**, so I don’t lose readability but get to take advantage of the performance benefit.”

The reason is that the two loops are in fact not identical.

The semantics for enumeration is that you aren’t allowed to change the object being enumerated while an enumeration is in progress. If you do, then the enumerator will throw an InvalidOperationException the next time you talk to it. On the other hand, the **for** loop

doesn't care if you change the collection while you're enumerating it. If you insert items into the collection inside the **for** loop, the loop will keep on going and depending on where the insertion happened, you might double-enumerate an item.

If the compiler changed the **foreach** to a **for**, then a program that used to throw an exception would now run without a hiccup. Whether you consider this an "improvement" is a matter of opinion. (Depending on the circumstances, it may be better for the program to crash than to produce incorrect results.)

Now, the compiler might be able to prove that you don't change the collection inside the loop, but that is often hard to prove. For example, does this loop change the collection?

```
ArrayList list = target.GetTheList();
foreach (Object o in list) {
  o.GetHashCode();
}
```

Well, it doesn't look like it. But who knows, maybe `target` looks like this:

```
class Sneaky {
  ArrayList list_;
  public Sneaky(ArrayList list) { list_ = list; }
  public override int GetHashCode()
  {
    list_.Add(this);
    return base.GetHashCode();
  }
}
class SneakyContainer {
  public ArrayList GetTheList()
  {
    ArrayList list = new ArrayList();
    list.Add(new Sneaky(list));
    return list;
  }
}
class Program {
  static public void Main()
  {
    SneakyContainer target = new SneakyContainer();
    ArrayList list = target.GetTheList();
    foreach (object o in list) {
      o.GetHashCode();
    }
  }
}
```

Ah, little did you know that `o.GetHashCode()` modifies the ArrayList. And yet it looked so harmless!

If the `SneakyContainer` class came from another assembly, then the compiler must assume the worst, because it's possible that somebody will make that assembly sneaky after you compiled your assembly.

If that's not a messed-up enough reason, here's another: The `ArrayList` class is not sealed. Therefore, somebody can override its IEnumerable.GetEnumerator and return a nonstandard enumerator. For example, here's a class that always returns an empty enumerator:

```
class ApparentlyEmptyArrayList : ArrayList {
  static int[] empty = new int[] { };
  public override IEnumerator GetEnumerator()
    { return empty.GetEnumerator(); }
}
```

"Who would be so crazy as to override the enumerator?"

Well, this one is rather bizarro, but more generally one might override the enumerator in order to add a filter or to change the order of enumeration.

So you can't even trust that your ArrayList really is an ArrayList. It might be an ApparentlyEmptyArrayList!

Now if the compiler wanted to do this rewrite optimization, not only would it have to prove that the object being enumerated is not modified inside the enumeration, it also has to prove that the object really is an ArrayList and not a derived class that may have overridden the GetEnumerator method.

Given the late-binding nature of cross-assembly classes, the number of cases where the compiler can prove these requirements is very restricted indeed, to the point where the number of places where the optimization can safely be performed without changing semantics becomes so vanishingly small as to be not worth the effort.

Raymond Chen

Follow