Неочевидности в использовании C#-замыканий

7 августа 2013 Lambda Closures C# .NET

Язык C# даёт нам возможность пользоваться замыканиями — мощным механизмом, который позволяет анонимным методам и лямбдам захватывать свободные переменные в своём лексическом контексте. И в .NET-мире многие программисты очень любят использовать замыкания, но немногие понимают, как они действительно работают. Начнём с простого примера:

```
public void Run()
{
  int e = 1;
  Foo(x => x + e);
}
```

Ничего сложного тут не происходит: мы просто «захватили» локальную переменную `e` в лямбду, которая передаётся в некоторый метод `Foo`. Посмотрим, во что компилятор развернёт такую конструкцию:

```
public void Run()
{
  DisplayClass c = new DisplayClass();
  c.e = 1;
  Foo(c.Action);
}
private sealed class DisplayClass
{
  public int e;
  public int Action(int x)
  {
    return x + e;
  }
}
```

Как видно из примера, для нашего замыкания создаётся дополнительный класс, который содержит захватываемую переменную и целевой метод. Это знание поможет нам осознать

поведение замыканий в различных ситуациях.

Цикл for

Наверное, это самый классический пример, который приводят все:

```
public void Run()
{
  var actions = new List<Action>();
  for (int i = 0; i < 3; i++)
    actions.Add(() => Console.WriteLine(i));
  foreach (var action in actions)
    action();
}
```

В этом примере сделана типичная ошибка. Начинающие программисты думаю, что этот код выведет `"0 1 2"`, но на самом деле он выведет `"3 3 3"`. Такое странное поведение легко понять, если взглянуть на развёрнутую версию этого метода:

```
public void Run()
{
  var actions = new List<Action>();
  DisplayClass c = new DisplayClass();
  for (c.i = 0; c.i < 3; c.i++)
    list.Add(c.Action);
  foreach (Action action in list)
    action();
}

private sealed class DisplayClass
{
  public int i;

  public void Action()
  {
    Console.WriteLine(i);
  }
}
```

В таком случае часто говорят, что переменная замыкается по ссылке, а не по значению. Эту особенность замыканий многие осуждают, как непонятную, хотя она является достаточно логичной для тех, кто хорошо представляет, что скрыто под капотом замыканий. Эту тему очень подробно обсуждает Эрик Липперт в постах О вреде замыканий на переменных цикла и Замыкания на переменных цикла. Часть 2.

Цикл foreach

Посмотрим более интересный пример:

```
public void Run()
{
```

```
  var actions = new List<Action>();
  foreach (var i in Enumerable.Range(0, 3))
    actions.Add(() => Console.WriteLine(i));
  foreach (var action in actions)
    action();
}
```

Что выведет это код? Увы, однозначного ответа на этот вопрос нету. Дело в том, что в ранних версиях C# поведение foreach было подобно поведению for: переменная цикла создавалась один раз и захватывалась во всех лямбдах. А в C# 5.0 это поведение поменяли (тут Эрик Липперт признаётся, что Microsoft всё-таки сделали breaking change). Теперь этот код выводит `"0 1 2"` . Заметьте, что это особенность именно языка, а не платформы. Если вы работаете из VisualStuido 2012 и меняете TargetFramework на 3.5, то ничего не поменяется, а вот из VisualStudio 2010 вы сможете пронаблюдать старое поведение. На Stackoverflow Джон Скит объясняет почему было решено сделать различное поведение для `for` и `foreach`. Взглянем на новый вариант развёрнутой версии кода:

```
public void Run()
{
  var actions = new List<Action>();
  foreach (int i in Enumerable.Range(0, 3))
  {
    DisplayClass c = new DisplayClass();
    c.i = i;
    list.Add(c1.Action);
  }
  foreach (Action action in list)
    action();
}

private sealed class DisplayClass
{
  public int i;

  public void Action()
  {
    Console.WriteLine(i);
  }
}
```

Легко можно заметить разницу: в C# 5.0 на каждую итерацию цикла `foreach` мы имеем новый экземпляр сгенерированного класса, обеспечивающего логику замыкания. На Хабре можно почитать поподробнее про замыкания в новой версии C#.

Замыкание нескольких переменных

Рассмотрим ситуацию в которой у нас есть несколько переменных, которые замыкаются в различных переменных:

```
public void Run()
```

```
{
  int x = 1, y = 2;
  Foo(u => u + x, u => u + y);
}
```

Можно подумать, что в этом случае у нас сгенерируется два дополнительных класса, каждый из которых будет отвечать за единственную переменную. Но на самом деле будет только один сгенерированный класс:

```
public void Run()
{
  DisplayClass c = new DisplayClass();
  c.x = 1;
  c.y = 2;
  Foo(c.ActionX, c.ActionY);
}

private sealed class DisplayClass
{
  public int x;
  public int y;

  public int ActionX(int u)
  {
    return u + x;
  }

  public int ActionY(int u)
  {
    return u + y;
  }
}
```

Таким образом, лямбды оказываются «связаны»: сборщик мусора доберётся до них только после того, как не останется ссылок ни на одну из них. Представьте ситуацию, в которой первая лямбда используется при инициализации долгоживущего объекта, а вторая — по окончанию работы с ним. И пусть таких объектов будет много. В этом случае инициализирующие лямбды будут болтаться в памяти очень долго, хотя никто их больше никогда не будет вызывать.

Scope

Есть ещё одна особенность работы замыканий, о которой полезно знать. Рассмотрим пример:

```
public void Run(List<int> list)
{
  foreach (var element in list)
  {
    var e = element;
    if (Condition(e))
      Foo(x => x + e);
  }
```

```
}
```

А теперь вопрос: в каком месте будет создан объект замыкания? Не смотря на то, что лямбда создаётся внутри `if`-а, объект будет создаваться в том же `scope` -е, что и захватываемая переменная:

```
public void Run(List<int> list)
{
  foreach (int element in list)
  {
    DisplayClass c = new DisplayClass();
    c.e = element;
    if (Condition(c.e))
      Foo(c.Action);
  }
}

private sealed class DisplayClass
{
  public int e;

  public int Action(int x)
  {
    return x + e;
  }
}
```

Такая особенность может иметь значение в случае, если `list` очень большой, а условие `Condition(e)` выполняется весьма редко. Ведь будет происходить бесполезное создание экземпляров класса `DisplayClass` , что негативно скажется на памяти и производительности. Мы можем исправить эту ситуацию:

```
public void Run(List<int> list)
{
  foreach (var element in list)
    if (Condition(element))
    {
      var e = element;
      Foo(x => x + e);
    }
}
```

Данный метод будет разворачиваться более оптимально, ведь теперь конструктор `DisplayClass` будет вызываться только тогда, когда он действительно нужен:

```
public void Run(List<int> list)
{
  foreach (int element in list)
    if (Condition(element))
    {
```

```
      DisplayClass c = new DisplayClass();
      c.e = element;
      Foo(c.Action);
    }
}

private sealed class DisplayClass
{
  public int e;

  public int Action(int x)
  {
    return x + e;
  }
}
```

Задачи

На приведённую тему есть три задачки в ProblemBook.NET: ClosureAndForeach, ClosureAndFor, ClosureAndVariable.

Для дизассемблирования удобно пользоваться утилитой dotPeek от JetBrains с включённой опцией `Show compiler-generated code`. Приведённый в статье код немного причёсан по сравнению с дизассемблированной версией для повышения читаемости.

Поделиться:
