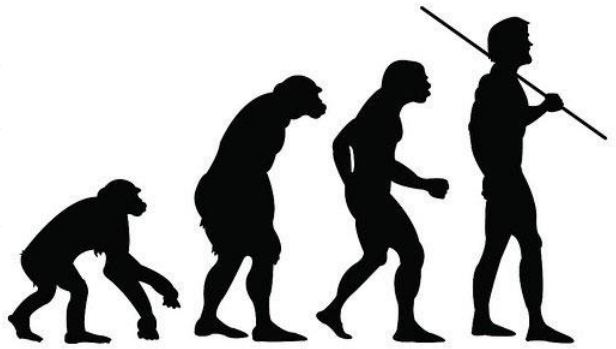

LINQ进化史

北京理工大学计算机学院
金旭亮

```
static void FindOddNumbers()
{
    //生成从1到10的整数集合
    var nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    //找出所有的奇数，放到List<int>集合中
    var result = new List<int>();
    foreach (var num in nums)
    {
        if (IsOdd(num))
        {
            result.Add(num);
        }
    }
    //输出结果
    foreach (var num in result)
    {
        Console.WriteLine(num);
    }

}
```

原始形态

将数据处理功能外置为独立的方法，提升代码可维护性

```
static IEnumerable<int> FilterIntegers(IEnumerable<int> list)
{
    //找出所有的奇数，放到List<int>集合中
    var result = new List<int>();
    foreach (var num in list)
    {
        if (IsOdd(num))
        {
            result.Add(num);
        }
    }
    return result;
}
```

引入委托进行重构

```
public delegate bool PredicateDelegate(int value);
```

新写一个方法：

```
static IEnumerable<int> FilterIntegers(IEnumerable<int> list,
    PredicateDelegate predicate)
{
    //找出所有的偶数，放到List<int>集合中
    var result = new List<int>();
    foreach (var num in list)
    {
        if (predicate(num))
        {
            result.Add(num);
        }
    }
    return result;
}
```

```
static bool IsEven(int num)
{
    return num % 2 == 0;
}
```

现在在外部就可以“临时”指定到底是需要偶数还是奇数

系统具有了更强的可扩展性.....

```
var result = FilterIntegers(nums,IsEven);
//result = FilterIntegers(nums, IsOdd);
```

引入“泛型”进行重构……

```
public delegate bool PredicateGenericDelegate<T>(T value);
```

```
static IEnumerable<T> Filter<T>(IEnumerable<T> list,
    PredicateGenericDelegate<T> predicate)
{
    //找出所有的偶数，放到List<int>集合中
    var result = new List<T>();
    foreach (var num in list)
    {
        if (predicate(num))
        {
            result.Add(num);
        }
    }
    return result;
}
```

重构后的代码，已经与具体的数据类型“脱钩”……

使用预定义的Func<>，无需再自定义一个委托

```
static IEnumerable<T> Filter2<T>(IEnumerable<T> list,
    Func<T, bool> predicate)
{
    //找出所有的偶数，放到List<int>集合中
    var result = new List<T>();
    foreach (var num in list)
    {
        if (predicate(num))
        {
            result.Add(num);
        }
    }
    return result;
}
```

将Lambda表达式直接作为方法参数，现在啥都不需要了，两行代码搞定一切.....

```
static void FindOddOrEvenNumbersUseLambda()
{
    //生成从1到10的整数集合
    var nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    //现在在外部就可以“临时”指定到底是需要偶数还是奇数
    var result = Filter2(nums, num => num % 2 == 0);
    //result = Filter2(nums, num => num % 2 != 0);
    //输出结果
    foreach (var num in result)
    {
        Console.WriteLine(num);
    }

}
```

Cool!

引入**扩展方法**特性改造Filter方法

```
static class MyExtensions
{
    1 reference
    public static IEnumerable<T> Filter<T>(this IEnumerable<T> list,
        Func<T, bool> predicate)
    {
        //找出所有的偶数，放到List<int>集合中
        var result = new List<T>();
        foreach (var num in list)
        {
            if (predicate(num))
            {
                result.Add(num);
            }
        }
        return result;
    }
}
```

现在，所有的集合都自动地拥有此特性！

```
static void FindOddOrEvenNumbersUseExtensionMethod()
{
    //生成从1到10的整数集合
    var nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    var result = nums.Filter(num => num % 2 == 0);
    //result = nums.Filter(nums, num => num % 2 != 0);
    //输出结果
    foreach (var num in result)
    {
        Console.WriteLine(num);
    }

}
```

使用yield特性，进一步优化性能

```
public static IEnumerable<T> AdvancedFilter<T>(this IEnumerable<T> list,
 Func<T, bool> predicate)
{
    //找出所有的偶数，放到List<int>集合中

    foreach (var num in list)
    {
        if (predicate(num))
        {
            yield return num;
        }
    }
}
```

当第一个Filter得到一个数时，它会立即把这个数传给下一下Filter，而无需等待第一个Filter处理完全部的数.....
大大节约了内存，并提升了性能

优化后的扩展方法，支持级联调用！

```
static void FindOddOrEvenNumbersUseExtensionMethod2()
{

    var nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    var result = nums.AdvancedFilter(num => num % 2 != 0)
        .AdvancedFilter(num => num % 3 != 0);


    foreach (var num in result)
    {
        Console.WriteLine(num);
    }

}
```

把AdvancedFilter改为Where，我们实际上就实做出来了.NET基类库中的Where扩展方法……

```
static void FindOddOrEvenNumbersUseWhere()
{

    var nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    var result = nums.Where(num => num % 2 != 0)
        .Where(num => num % 3 != 0);


    foreach (var num in result)
    {
        Console.WriteLine(num);
    }

}
```

最后再进一步，改写为LINQ查询语句：

```
static void FindOddOrEvenNumbersUseLINQ()
{
    var nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    var result = from num in nums
                 where num % 2 != 0 && num % 3 != 0
                 select num;
    foreach (var num in result)
    {
        Console.WriteLine(num);
    }
}
```

LINQ查询与Where扩展方法，本质上是一致的，但LINQ方式更易读

小结

这一讲我们从一个最原始的版本开始，一步步地进行重构和优化，最终走向“LINQ”这个里程碑。

LINQ真是一个很Cool的技术，在后面的课程中，我们将经常使用它来处理数据。