编程语言的发展趋势及未来方向

目录

[编程语言的发展趋势及未来方向 1](#_Toc30366)

[第一章 一、历史回顾及趋势概述 2](#_Toc25588)

[第一节 首先，编程语言的发展非常缓慢。oo等等，但是远没有好上1000倍。 3](#_Toc23345)

[第二节 出现在框架及工具等方面 3](#_Toc25643)

[第三节 我们都不断地提高抽象级别 汇编。面向过程 oo 托管 4](#_Toc14303)

[第二章 我认为，现在影响力较大的趋势主要有3种。声明 动态 并发 4](#_Toc202)

[第一节 首先，我们会越来越多地使用声明式的编程风格。Dsl 4](#_Toc27174)

[第二节 二、声明式编程与DSL 5](#_Toc28969)

[第三节 我之前提到过，现在有两种比较重要的成果，一是DSL（Domain Specific Language，领域特定语言），另一个则是函数式编程。 7](#_Toc9351)

[第四节 三、函数式编程 13](#_Toc15402)

[第五节 关于声明式编程的还有一部分重要的内容，那便是函数式编程 13](#_Toc32567)

[四、动态语言 18](#_Toc31391)

[第三章 我下面继续要讲的是动态语言，这也是我之前提到的三种趋势之一。 19](#_Toc21358)

[第一节 五、元编程 动态语言所擅长的“元编程” 编译器即服务。 22](#_Toc24034)

[第二节 动态语言的另一个关键和有趣之处在于“元编程”。“元编程”实际上是“代码生成”的一种别称， 22](#_Toc24081)

[第三节 其中一种做法，是我们正在努力实现的“编译器即服务”， 23](#_Toc9299)

[第四章 六、并发 24](#_Toc19659)

[七、总结 29](#_Toc12949)

这是Anders Hejlsberg（不用介绍这是谁了吧）在比利时TechDays 2010所做的开场演讲。由于最近我在博客上关于语言的讨论比较多，出于应景，也打算将Anders的演讲完整地听写出来。我希望这个讲座可以从侧面回答某些朋友关于“语言讨论是否有价值”的疑问，并且展示出目前语言的发展状况以及微软在这方面的努力。完整内容将分为多次发表，每次一小部分，包含大约10分钟的演讲内容。等不及的朋友也和可以下载演讲视频一睹为快。现在的第1部分则包含Anders对编程语言发展的历史回顾，以及对趋势简单概述。

# 一、历史回顾及趋势概述

如果没有特别说明，所有的文字都直接翻译自Anders的演讲，并使用我自己的口语习惯表达出来，对于Anders的口误及反复等情况，必要时在译文中自然也会进行忽略。为了方便理解，我也会将视频中关键部分进行截图，而某些代码演示则会直接作为文章内容发表。

（听写开始）

大家好，我是Anders Hejlsberg，现在是微软的Technical Fellow，担任C#编程语言的首席架构师，也参与并领导.NET Framework以及各种语言的开发。我现在打算谈一下……实际上是我脑海中一些影响未来5到10年编程语言设计的内容。比如C#或VB该怎么走，F#该怎么办，这次演讲主要就是讨论这些影响我们的东西。

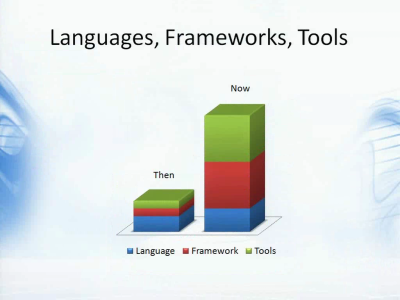
虽然主要内容是谈论未来的，但是我还是想先回顾一下历史。你们有些人可能对这个产品有印象，这是我大约27年前的工作内容，Turbo Pascal，这也是我进入这个领域的起点。我先在拿出这个东西是想展示当年写程序的情况，然后可以讨论目前究竟的发展到哪儿了。

事实上，我现在的机器里正好有TURBO.COM文件，大约39K，嘿，现在还可以运行。我们现在来试着写一点程序。先来创建的程序叫做Hello.pas……（开始写代码）……一个Pascal小程序写好了，我们来运行一下……（出现编译错误）啊噢，有地方我写错了……这个特性在当年是个创新，它会自动打开编辑器，直接把我们带去出错的地方。嗯，我们现在来纠正语法错误，把双引号改成单引号。

再运行一下，现在成功了，（观众掌声）呵呵，谢谢，谢谢。事实上，在27年后这个程序还能在这台机器上运行还真是挺神奇的。

现在，我们来看一下，从那时算起硬件已经发展了……嗯，我那时写Pascal的机器是Z-80，拥有48K内存。从那时算起，我现在这台机器已经有大约10万倍的外部存储容量，1万倍的内存大小，CPU速度也有大约1000倍的提高。

但是如果你关注一下目前的软件……过去27年里编程语言到底进步了多少？呵呵，有趣的是如果你仔细观察这些代码，会发现C#还比Turbo Pascal的版本多一行。这也给我们带来了一些值得关注的东西。



## 首先，编程语言的发展非常缓慢。oo等等，但是远没有好上1000倍。

## 出现在框架及工具等方面

另一方面，你可能会想，那么这些努力都到哪里去了呢？事实上这些努力没有体现在编程语言上，而是

出现在框架及工具等方面了。如果你关注如今我们使用的框架，它们的体积的确有1000倍的增长。例如当年Turbo Pascal所带的框架大约有，比如说100个功能，而现在的.NET Framework里则有一万个类，十万个方法，的确有1000倍的增长。与此类似，如果你观察现在的IDE，我们现在已经有了无数强大的功能，例如语法提示，重构，调试器，探测器等等，这方面的新东西有很多。与此相比，编程语言的改进的确很不明显。

另一方面，如.NET，Java等框架的重要性提高了许多。而编程语言往往都倾向于构建于现有的工具上，而不会从头写起。现在出现的编程语言，例如F#，如果你关注Java领域那么还有Scala，Clojure等等，它们都是基于现有框架构建的。现在已经有太多东西可以直接利用了，每次从头开始的代价实在太高。

## 我们都不断地提高抽象级别 汇编。面向过程 oo 托管

还有件事，便是在过去5、60年的编程历史中，我们都不断地提高抽象级别 ，我们都在不断地让编程语言更有表现力，让我们可以用更少的代码完成更多的工作。我们一开始先使用汇编，然后使用面向过程的语言，例如Pascal和C，然后便是面向对象语言，如C++，随后就进入了托管时代──受托管的执行环境，例如.NET，Java，它们的主要特性有自动的垃圾收集，类型安全等等。我目前还没有看出这样的趋势有停止的迹象，因此我们还会看到抽象级别越来越高的语言，而语言的设计者则必须理解并预测下一个抽象级别是什么样子的。

# 我认为，现在影响力较大的趋势主要有3种。声明 动态 并发

## 首先，我们会越来越多地使用声明式的编程风格。Dsl

这里我主要会提到例如DSL（Domain Specific Language，领域特定语言）以及函数式编程。然后在过去的五年里，我发现对于动态语言的研究变得非常火热，其中对我们产生重大影响的无疑是动态语言所拥有的良好的元编程能力，还有一些非常有趣的东西，例如JavaScript引擎的发展。然后便是并发编程，无论我们愿不愿意，多核的产生都在迫使我们不得不重视并发编程。

有一点值得一提，那便是随着语言的发展，原本的编程语言分类方式也要有所改变了。以前我们经常说面向对象语言，[动态语言](http://en.wikipedia.org/wiki/Dynamic_programming_language)或是函数式语言。但是我们现在发现，这些边界变得越来越模糊，经常会互相学习各自的范式。静态语言中出现了动态类型，动态语言里也出现了静态能力，而如今所有主要的编程语言都受到函数式语言的影响。因此，一个越来越明显的趋势是“多范式程序设计语言”。

在接下来的部分中，我将深入讨论以上提到的这些内容。

## 二、声明式编程与DSL

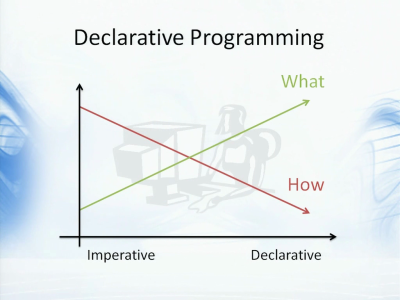
这是[Anders Hejlsberg](http://en.wikipedia.org/wiki/Anders_Hejlsberg)（不用介绍这是谁了吧）在[比利时TechDays 2010所做的开场演讲](http://channel9.msdn.com/posts/adebruyn/TechDays-2010-Developer-Keynote-by-Anders-Hejlsberg/)。由于最近我在博客上关于语言的讨论比较多，出于应景，也打算将Anders的演讲完整地听写出来。在上一部分中，Anders指出语言本身在过去的数十年里并没有明显的发展，并给出了他眼中编程语言发展趋势的预测。在现在的第2部分中，Anders将阐述声明式编程的理念及DSL，并演示C#中一种内部DSL的形式：LINQ。

如果没有特别说明，所有的文字都直接翻译自Anders的演讲，并使用我自己的口语习惯表达出来，对于Anders的口误及反复等情况，必要时在译文中自然也会进行忽略。为了方便理解，我也会将视频中关键部分进行截图，而某些代码演示则会直接作为文章内容发表。

（听写开始，接上篇）



这里先从声明式（Declarative）编程谈起。



目前我们在编写软件时大量使用的是命令式（Imperative）编程语言，例如C#，Java或是C++等等。这些语言的特征在于，写出的代码除了表现出“什么（What）”是你想做的事情之外，更多的代码则表现出实现的细节，也就是“如何（How）”完成工作。这部分代码有时候多到掩盖了我们原来问题的解决方案。比如，你会在代码里写for循环，if语句，a等于b，i加一等等，这体现出机器是如何处理数据。首先，这种做法让代码变得冗余，而且它也很难让执行代码的基础设施更聪明地判断该如何去执行代码。当你写出这样的命令是代码，然后把编译后的中间语言交给虚拟机去执行，此时虚拟机并没有多少空间可以影响代码的执行方式，它只能根据指令一条一条老老实实地去执行。例如，我们现在想要并行地执行程序就很困难了，因为更高层次的一些信息已经丢失了。这样，我们只能在代码里给出“How”，而不能体现出“What”的信息。

有多种方式可以将“What”转化为更为“声明式”的编程风格，我们只要能够在代码中体现出更多“What”，而不是“How”的信息，这样执行环境便可以更加聪明地去适应当前的执行要求。例如，它可以决定投入多少CPU进行计算，你的当前硬件是什么样的，等等。

## 我之前提到过，现在有两种比较重要的成果，一是[DSL](http://en.wikipedia.org/wiki/Domain-specific_language)（Domain Specific Language，领域特定语言），另一个则是[函数式编程](http://en.wikipedia.org/wiki/Functional_programming)。

其实DSL不是什么新鲜的玩意儿，我们平时一直在用类似的东西，比如，SQL，CSS，正则表达式，有的可能更加专注于一个方面，例如[Mathematica](http://en.wikipedia.org/wiki/Mathematica)，[LOGO](http://en.wikipedia.org/wiki/Logo_%28programming_language%29)等等。这些语言的目标都是特定的领域，与之相对的则是[GPPL](http://en.wikipedia.org/wiki/General-purpose_programming_language)（General Purpose Programming Language，通用目的编程语言）。

对于DSL而言其实并没有一个明确的定义，在这里我也不打算为它下个定义，例如[UML](http://en.wikipedia.org/wiki/Unified_Modeling_Language)甚至根本没有特定的语法。不过我这里会谈一些我觉得比较重要的东西。

[Martin Fowler](http://martinfowler.com/)提出DSL应该分为外部DSL及内部DSL两种，我认为这种划分方式还是比较有意义的。外部DSL是自我包含的语言，它们有自己特定语法、解析器和词法分析器等等，它往往是一种小型的编程语言，甚至不会像GPPL那样需要源文件。与之相对的则是内部DSL。内部DSL其实更像是种别称，它代表一类特别API及使用模式。这里我会给你们看一些示例。

这些是我们平时会遇到的一些外部DSL，如这张幻灯片上表现的XSLT，SQL或是Unix脚本。外部DSL的特点是，你在构建这种DSL时，其实扮演的是编程语言设计者的角色，这个工作并不会交给普通人去做。外部DSL一般会直接针对特定的领域设计，而不考虑其他东西。[James Gosling](http://en.wikipedia.org/wiki/James_Gosling)曾经说过这样的话，每个配置文件最终都会变成一门编程语言。你一开始可能只会用它表示一点点东西，然后慢慢你便会想要一些规则，而这些规则则变成了表达式，可能你还会定义变量，进行条件判断等等。而最终它就变成了一种奇怪的编程语言，这样的情况屡见不鲜。

事实上，现在有一些公司也在关注DSL的开发。例如以前在微软工作的[Charles Simonyi](http://en.wikipedia.org/wiki/Charles_Simonyi)提出了[Intentional Programming](http://en.wikipedia.org/wiki/Intentional_programming)的概念，还有一个叫做[JetBrains](http://en.wikipedia.org/wiki/JetBrains)的公司提供一个叫做[MPS（Meta Programming System）](http://en.wikipedia.org/wiki/JetBrains_MPS)的产品。最近微软也提出了自己的[Oslo](http://en.wikipedia.org/wiki/Oslo_%28Microsoft%29)项目，而在Eclipse世界里也有个叫做[Xtext](http://www.eclipse.org/Xtext/)的东西，所以其实在这方面现在也有不少人在尝试。

我在观察外部DSL时，往往会关注它的语法到底提供了多少空间，例如一种XML的方言，利用XML方言的好处在于有不少现成的工具可用，这样可以更快地定义自己的语法。

而内部DSL，正像我之前说的那样，它其实只是一系列特别的API及使用模式的别称。这里则是一些LINQ查询语句，[Ruby on Rails](http://rubyonrails.org/)以及[jQuery](http://jquery.com/)代码。内部DSL的特点是，它其实只是一系列API，但是你可以“假装”它们一种DSL。内部DSL往往会利用一些“流畅化”的技巧，例如像这里的LINQ或jQuery那样把一些方法通过“点”连接起来。有些则利用了元编程的方式，如这里的Ruby on Rails就涉及到了一些元编程。这种DSL可以访问语言中的代码或变量，以及利用如代码补全，重构等母语言的所有特性。

现在我会花几分钟时间演示一下我所创建的DSL，也就是LINQ。我相信你们也已经用过不少LINQ了，不过这里我还是快速的展示一下我所表达的更为“声明式”的编程方式。

public class Product

{

public int ProductID { get; set; }

public string ProductName { get; set; }

public string CategoryName { get; set; }

public int UnitPrice { get; set; }

public static List<Product> GetProducts() { /\* ... \*/ }

}

public partial class \_Default : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

List<Product> products = Product.GetProducts();

List<Product> result = new List<Product>();

foreach (Product p in products)

{

if (p.UnitPrice > 20) result.Add(p);

}

GridView1.DataSource = result;

GridView1.DataBind();

}

}

这里有许多Product对象，那么现在我要筛选出所有单价大于20的那些， 再把他们显示在一个GridView中。传统的做法就是这样，我先得到所有的Product对象，然后foreach遍历每个对象，再判断每个对象的单价，最终把数据绑定到GridView里。运行这个程序……（打开页面）这就是就能得到结果。

好，那么现在我要做一些稍微复杂的事情。可能我不是要展示单价超过20的Product对象，而是要查看每个分类中究竟有多少个单价超过20的对象，然后根据数量进行排序。如果不用DSL完成这个工作，那么我可能会先定义一个对象来表示结果：

class Grouping

{

public string CategoryName { get; set; }

public int ProductCount { get; set; }

}

这是个表示分组的对象，用于保存分类的名称和产品数量。然后我们就会写一些十分丑陋的代码：

Dictionary<string, Grouping> groups = new Dictionary<string, Grouping>();

foreach (Product p in products)

{

if (p.UnitPrice >= 20)

{

if (!groups.ContainsKey(p.CategoryName))

{

Grouping r = new Grouping();

r.CategoryName = p.CategoryName;

r.ProductCount = 0;

groups[p.CategoryName] = r;

}

groups[p.CategoryName].ProductCount++;

}

}

List<Grouping> result = new List<Grouping>(groups.Values);

result.Sort(delegate(Grouping x, Grouping y)

{

return

x.ProductCount > y.ProductCount ? -1 :

x.ProductCount < y.ProductCount ? 1 :

0;

});

我先创建一个新的字典，用于保存分类名称到分组的对应关系。然后我遍历每个Product对象，对于每个单价大于20的对象，如果字典中还没有保存对应的分组则创建一个，然后将数量加一。然后为了排序，我调用Sort方法，于是我要提供一个委托作为排序方法，然后blablablabla……执行之后……（打开页面）我自然可以得到想要的结果。

但是，首先这些代码写起来需要花费一些时间，很显然。然后仔细观察，你会发现这写代码几乎都是在表示“How”，而“What”基本已经丢失了。假设我离开了，现在新来了一个程序员要维护这段代码，他会需要一点时间才能完整理解这段代码，因为他无法直接看清代码的目标。

不过如果这里我们使用DSL，也就是LINQ，就像这样：

var result = products

.Where(p => p.UnitPrice >= 20)

.GroupBy(p => p.CategoryName)

.OrderByDescending(g => g.Count())

.Select(g => new { CategoryName = g.Key, ProductCount = g.Count() });

products……先调用Where……blablabla……再GroupBy等等。由于我们这里可以使用DSL来表示高阶的术语，用以体现我们想做的事情。于是这段代码则更加关注于“What”而不是“How”。我这里不会明确地指示我想要过滤的方式，我也不会明确地说我要建立字典和分类，这样基础结构就可以聪明地，或者说更加聪明地去确定具体的执行方式。你可能比较容易想到我们可以并行地执行这段代码，因为我没有显式地指定做事方式，我只是表示出我的意图。

我们打开页面……（打开页面）很显然我们得到了相同的结果。

这里比较有趣的是，内部DSL是如何设计进C#语法中的，为此我们为C# 3.0添加了一系列的特性，例如Lambda表达式，扩展方法，类型推断等等。这些特性统一起来之后，我们就可以设计出更为丰富的API，组合之后便成为一种内部DSL，就像这里的LINQ查询语言。

除了使用API的形式之外，我们还可以这样做：

var result =

from p in products

where p.UnitPrice >= 20

group p by p.CategoryName into g

orderby g.Count() descending

select new { CategoryName = g.Key, ProductCount = g.Count() };

编译器会简单地将这种形式转化为前一种形式。不过，这里我认为有意思的地方在于，你完全可以创建一门和领域编程语言完全无关的语法，然后等这种语法和API变得流行且丰富起来之后，再来创一种新的表现形式，就如这里的LINQ查询语法。我颇为中意这种语言设计的交流方式。

OK，现在我们回到下面的内容。

## 三、函数式编程

这是[Anders Hejlsberg](http://en.wikipedia.org/wiki/Anders_Hejlsberg)（不用介绍这是谁了吧）在[比利时TechDays 2010所做的开场演讲](http://channel9.msdn.com/posts/adebruyn/TechDays-2010-Developer-Keynote-by-Anders-Hejlsberg/)。由于最近我在博客上关于语言的讨论比较多，出于应景，也打算将Anders的演讲完整地听写出来。在上一部分中，Anders阐述了他眼中声明式编程的理念及DSL，并演示C#中一种内部DSL的形式：LINQ。在这一部分中，Anders谈及了声明式编程的另一个重要组成部分：函数式编程，并使用.NET平台上的函数式编程语言F#进行了演示。

如果没有特别说明，所有的文字都直接翻译自Anders的演讲，并使用我自己的口语习惯表达出来，对于Anders的口误及反复等情况，必要时在译文中自然也会进行忽略。为了方便理解，我也会将视频中关键部分进行截图，而某些代码演示则会直接作为文章内容发表。

（听写开始，接[上篇](http://blog.zhaojie.me/2010/04/trends-and-future-directions-in-programming-languages-by-anders-2-declarative-programming-and-dsl.html)）

## 关于声明式编程的还有一部分重要的内容，那便是[函数式编程](http://en.wikipedia.org/wiki/Functional_programming)

函数式编程已经有很长时间的历史了，当年[LISP](http://en.wikipedia.org/wiki/Lisp_programming_language)便是个函数式编程语言。除了LISP以外我们还有其他许多函数式编程语言，如[APL](http://en.wikipedia.org/wiki/APL_programming_language)、[Haskell](http://en.wikipedia.org/wiki/Haskell_(programming_language))、[Scheme](http://en.wikipedia.org/wiki/Scheme_(programming_language))、[ML](http://en.wikipedia.org/wiki/ML_programming_language)等等。关于函数式编程在学术界已经有过许多研究了，在大约5到10年前许多人开始吸收和整理这些研究内容，想要把它们融入更为通用的编程语言。现在的编程语言，如C#、Python、Ruby、[Scala](http://www.scala-lang.org/)等等，它们都受到了函数式编程语言的影响。

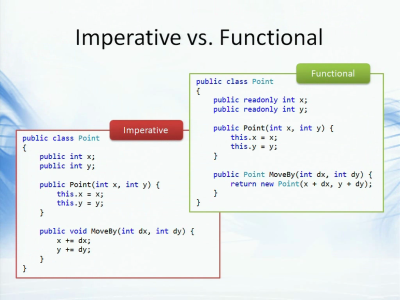
我想在这里先花几分钟时间简单介绍一下我眼中的函数式编程语言。我发现很多人听说过函数式编程语言，但还不十分清楚它们和普通的命令式编程语言究竟有什么区别。如今我们在使用命令式编程语言写程序时，我们经常会写这样的语句，嗨，x等于x加一，此时我们大量依赖的是状态，可变的状态，或者说变量，它们的值可以随程序运行而改变。

可变状态非常强大，但随之而来的便是叫做“副作用”的问题。在使用可变状态时，你的程序则会包含副作用，比如你会写一个无需参数的void方法，然后它会根据你的调用次数或是在哪个线程上进行调用对程序产生影响，因为void方法会改变程序内部的状态，从而影响之后的运行效果。

而在函数式编程中则不会出现这个情况，因为所有的状态都是不可变的。你可以声明一个状态，但是不能改变这个状态。而且由于你无法改变它，所以在函数式编程中不需要变量。事实上对函数式编程的讨论更像是数学、公式，而不像是程序语句。如果你把x = x + 1这句话交给一个程序员看，他会说“啊，你在增加x的值”，而如果你把它交给一个数学家看，他会说“嗯，我知道这不是true”。

然而，如果你给他看这条语言，他会说“啊，y等于x加一，就是把x + 1的计算结果交给y，你是为这个计算指定了一个名字”。这时候在思考时就是另一种方式了，这里y不是一个变量，它只是x + 1的名称，它不会改变，永远代表了x + 1。

所以在函数式编程语言中，当你写了一个函数，接受一些参数，那么当你调用这个函数时，影响函数调用的只是你传进去的参数，而你得到的也只是计算结果。在一个纯函数式编程语言中，函数在计算时不会对进行一些神奇的改变，它只会使用你给它的参数，然后返回结果。在函数式编程语言中，一个void方法是没有意义的，它唯一的作用只是让你的CPU发热，而不能给你任何东西，也不会有副作用。当然现在你可能会说，这个CPU发多少热也是一个副作用，好吧，不过我们现在先不讨论这个问题。



这里的关键在于，你解决问题的方法和以前大不一样了。我这里还是用代码来说明问题。使用函数式语言写没有副作用的代码，就好比在Java或C#中使用final或是readonly的成员。

例如这里，我们有一个Point类，构造函数接受x和y，还有一个MoveBy方法，可以把一个点移动一些位置。 在传统的命令式编程中，我们会改变Point实例的状态，这么做在平时可能不会有什么问题。但是，如果我把一个Point对象同时交给3个API使用，然后我修改了Point，那么如何才能告诉它们状态改变了呢？可能我们可以使用事件，blablabla，如果我们没有事件，那么就会出现那些不愉快的副作用了。

那么使用函数式编程的形式写代码，你的Point类还是可以包含状态，例如x和y，不过它们是readonly的，一旦初始化以后就不能改变了。MoveBy方法不能改变Point对象，它只能创建一个新的Point对象并返回出来。这就是一个创建新Point对象的函数，不是吗？这样就可以让调用者来决定是使用新的还是旧的Point对象，但这里不会有产生副作用的情况出现。

在函数式编程里自然不会只有Point对象，例如我们会有集合，如Dictionary，Map，List等等，它们都是不可变的。在函数式编程中，当我们向一个List里添加元素时，我们会得到一个新的List，它包含了新增的元素，但之前的List依然存在。所以这些数据结构的实现方式是有根本性区别的，它们的内部结构会设法让这类操作变的尽可能高效。

在函数式编程中访问状态是十分安全的，因为状态不会改变，我可以把一个Point或List对象交给任意多的地方去访问，完全不用担心副作用。函数式编程的十分容易并行，因为我在运行时不会修改状态，因此无论多少线程在运行时都可以观察到正确的状态。两个函数完全无关，因此它们是并行还是顺序地执行便没有什么区别了。我们还可以有延迟计算，可以进行Memorization，这些都是函数式编程中十分有趣的方面。

你可能会说，那么我们为什么不都用这种方法来写程序呢？嗯，最终，就像我之前说的那样，我们不能只让CPU发热，我们必须要把计算结果表现出来。那么我们在屏幕上打印内容时，或者把数据写入文件或是Socket时，其实就产生了副作用。因此真实世界中的函数式编程，往往都是把纯粹的部分进行隔离，或是进行更细致的控制。事实上也不会有真正纯粹的函数式编程语言，它们都会带来一定的副作用或是命令式编程的能力。但是，它们默认是函数式的，例如在函数式编程语言中，所有东西默认都是不可变的，你必须做些额外的事情才能使用可变状态或是产生危险的副作用。此时你的编程观念便会有所不同了。

我们在自己的环境中开发出了这样一个函数式编程语言，F#，已经包含在VS 2010中了。F#诞生于微软剑桥研究院，由[Don Syme](http://blogs.msdn.com/dsyme/)提出，他在F#上已经工作了5到10年了。F#使用了另一个函数式编程语言OCaml的常见核心部分，因此它是一个强类型语言，并支持一些如模式匹配，类型推断等现代函数式编程语言的特性。在此之上，F#又增加了异步工作流，度量单位等较为前沿的语言功能。

而F#最为重要的一点可能是，在我看来，它是第一个和工业级的框架和工具集，如.NET和Visual Studio，有深入集成的函数式编程语言。F#允许你使用整个.NET框架，它和C#也有类似的执行期特征，例如强类型，而且都会生成高效的代码等等。我想，现在应该是展示一些F#代码的时候了。

首先我想先从F#中我最喜欢的特性讲起，这是个F#命令行……（打开命令行窗口以及一个F#源文件）……F#包含了一个交互式的命令行，这允许你直接输入代码并执行。例如输入5……x等于5……然后x……显示出x的值是5。然后让sqr x等于x乘以x，于是我这里定义了一个简单的函数，名为sqr。于是我们就可以计算sqr 5等于25，sqr 10等于100。

F#的使用方式十分动态，但事实上它是一个强类型的编程语言。我们再来看看这里。这里我定义了一个计算平方和的函数sumSquares，它会遍历每个列表中每个元素，平方后再把它们相加。让我先用命令式的方式编写这个函数，再使用函数式的方式，这样你可以看出其中的区别。

let sumSquaresI l =

let mutable acc = 0

for x in l do

acc <- acc + sqr x

acc

这里先是命令式的代码，我们先创建一个累加器acc为0，然后遍历列表l，把平方加到acc中，然后最后我返回acc。有几件事情值得注意，首先为了创建一个可变的状态，我必须显式地使用mutable进行声明，在默认情况下这是不可变的。

还有一点，这段代码里我没有提供任何的类型信息。当我把鼠标停留在方法上时，就会显示sumSquaresI方法接受一个int序列作为参数并返回一个int。你可能会想int是哪里来的，嗯，它是由类型推断而来的。编译器从这里的0发现acc必须是一个int，于是它发现这里的加号表示两个int的相加，于是sqr函数返回的是个int，再接下来blablabla……最终它发现这里到处都是int。

如果我把这里修改为浮点数0.0，鼠标再停留一下，你就会发现这个函数接受和返回的类型都变成float了。所以这里的类型推断功能十分强大，也十分方便。

现在我可以选择这个函数，让它在命令行里执行，然后调用sumSquaresI，提供1到100的序列，就能得到结果了。

let rec sumSquaresF l =

match l with

| [] -> 0

| h :: t -> sqr h + sumSquaresF t

那么现在我们来换一种函数式的风格。这里是另一种写法，可以说是纯函数式的实现方式。如果你去理解这段代码，你会发现有不少数学的感觉。这里我定义了sumSqauresF函数，输入一个l列表，然后使用下面的模式去匹配l。如果它为空，则结果为0，否则把列表匹配为头部和尾部，然后便将头部的平方和尾部的平方和相加。

你会发现，在计算时我不会去改变任何一个变量的值，我只是创建新的值。我这里会使用递归，就像在数学里我们经常使用递归，把一个公式分解成几个变化的形式，以此进行递归的定义。在编程时我们也使用递归的做法，然后编译器会设法帮我们转化成尾递归或是循环等等。

于是我们便可以执行sumSquaresF函数，也可以得到相同的结果。当然实际上可能你并不会像之前这样写代码，你可能会使用高阶函数：

let sumSquares l = Seq.sum (Seq.map (fun x -> x \* x) l )

例如这里，我只是把函数x乘以x映射到列表上，然后相加。这样也可以得到相同的结果，而且这可能是更典型的做法。我这里只是想说明，这个语言在编程时可能会给你带来完全不同的感受，虽然它的执行期特征和C#比较接近。

这便是关于F#的内容。

四、动态语言

这是[Anders Hejlsberg](http://en.wikipedia.org/wiki/Anders_Hejlsberg)（不用介绍这是谁了吧）在[比利时TechDays 2010所做的开场演讲](http://channel9.msdn.com/posts/adebruyn/TechDays-2010-Developer-Keynote-by-Anders-Hejlsberg/)。由于最近我在博客上关于语言的讨论比较多，出于应景，也打算将Anders的演讲完整地听写出来。在上一部分中，Anders谈及了声明式编程的另一个重要组成部分：函数式编程，并使用.NET平台上的函数式编程语言F#进行了演示。在这一部分中，Anders讨论了动态语言及JavaScript的相关内容，“动态性”也是Anders眼中编程语言的发展趋势之一。

如果没有特别说明，所有的文字都直接翻译自Anders的演讲，并使用我自己的口语习惯表达出来，对于Anders的口误及反复等情况，必要时在译文中自然也会进行忽略。为了方便理解，我也会将视频中关键部分进行截图，而某些代码演示则会直接作为文章内容发表。

（听写开始，接[上篇](http://blog.zhaojie.me/2010/05/trends-and-future-directions-in-programming-languages-by-anders-3-functional-programming-and-fsharp.html)）

# 我下面继续要讲的是动态语言，这也是我之前提到的三种趋势之一。

我还是尝试着去找到动态语言的定义，但是你也知道……一般地说，动态语言是一些不对编译时和运行时进行严格区分的语言。这不像一些静态编程语言，比如C#，你先进行编译，然后会得到一些编译期错误，稍后再执行，而对于动态语言来说这两个阶段便混合在一起了。我们都熟悉一些动态语言，比如JavaScript，Python，Ruby，LISP等等。

动态语言有一些优势，而静态语言也有着另一些优势，这也是两个阵营争论多年的内容。老实讲，我认为结果不是两者中的任意一个，它们都有各自十分重要的优点，而长期来看，我认为结果应该是两者的杂交产物，我认为在语言发展中也可以看到这样的趋势，这两部分内容正在合并。

许多人认定动态语言执行起来很慢，也没有类型安全等等。我想在这里观察并比较一下，究竟是什么原因会让静态语言和动态语言在这方面有不同的性质。这里有一段有趣的代码，它的语法在JavaScript和C#里都是正确的，这样我们便能比较两种语言是如何处理这段代码的。

首先我们把它看作是一段C#代码，它只是用for循环把一堆整数相加，你肯定不会这么做，这只是一个示例。在C#中，当我们使用var关键字时，它表示“请为我推断这里的类型”，所以在这里a和i的类型都是int。

这断代码在执行的时候，这两个值都是32位整数，而for循环只是简单的使用ADD指令即可，执行起来自然效率很高。

但如果从JavaScript或是动态语言的角度来看……或者说对于动态类型的语言来说，var只代表了“一个值”，它可以是任意类型，我们不知道它究竟是什么。所以当我们使用var a或var i时，我们只是定义了两个值，其中包含了一个“类型”标记，表明在运行时它是个什么类型。在这里它是一个int，因此包含了存储int值的空间。但有些时候，例如要存储一个double值，那么可能便需要更多的空间，还可能是一个字符串，于是便包含一个引用。

所以两者的区别之一便是，表示同样的值在动态语言中会有一些额外的开销，代价较高。而在如今的CPU中，“空间”便等于“速度”，所以较大的值便需要较长时间进行处理，这里便损失了一部分效率。

在JavaScript中，我们如果要处理a加i，那么便不仅仅是一个ADD指令。首先它必须查看两个变量中的类型标记，然后根据类型选择合适的相加操作。于是再去加载两个值，然后再进行加法操作。这里还需要进行越界检查，因为在JavaScript中一旦越界了便要使用double，等等。很明显在这里也有许多开销。一般来说，动态语言是使用解释器来执行的，因此还有一些解释器需要的二进制码。你把这些开销全部加起来以后，便会发现执行代码时需要10倍到100倍的开销。

不过由于近几年来出现的一些动态虚拟机或引擎，目前这些情况改善了许多。比方说，这是传统的情况（上图左），如在IE 6或IE 7里使用的非常缓慢的解释器。目前的情况是，大部分的JavaScript引擎使用了JIT编译器（上图中），于是便省下了解释器的开销，这样性能损失便会减小至3到10倍。而在过去的两三年间，JIT编译器也变得越来越高效，浏览器中新一代的适应性JIT编译器（上图右），如[TraceMonkey](https://wiki.mozilla.org/JavaScript:TraceMonkey)，[V8](http://code.google.com/p/v8/)，还有如今[微软在IE 9中使用的Chakra引擎](http://blogs.msdn.com/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx)。这种适应性的JIT编译器使用了一部分有趣的技术，如Inline Caching、Type Specialization、Hidden Classes、Tracing等等，它们可以将开销降低至2到3倍的范围内，这种效率的提升可谓十分神奇。

在我看来，JavaScript引擎可能已经接近了性能优化的极限，我们在效率上可以提升的空间已经不多。不过我同样认为，如今JavaScript语言的性能已经足够快了，完全有能力统治Web客户端。

有人认为，JavaScript从来不是一种适合进行大规模编程的语言。如今也有一些有趣的工具，如[Google Web Tookit](http://code.google.com/webtoolkit/)，在微软[Nikhil Kothari](http://www.nikhilk.net/)也创建了[Script#](http://projects.nikhilk.net/ScriptSharp)，让你可以编写C#或Java代码，然后将代码编译成JavaScript，这就像是将JavaScript当作是一种中间语言。Google Wave的所有代码都用GWT写成，它的团队坚持认为用JavaScript不可能完成这样的工作，因为复杂度实在太高了。如今在这方面还有一些有趣的开发成果，我不清楚什么时候会结束。不过我认为，这些都不算是大规模的JavaScript开发方案，而编写C#或Java代码再生成JavaScript的方式也不能算是完全正确的做法。我们可以关注这方面的走向。

在.NET 4.0的运行时进行动态编程时，我们引入了一个新功能：[动态语言运行时](http://dlr.codeplex.com/)。可以这样理解，CLR的目的是为静态类型的编程语言提供一个统一的框架或编程模型，而DLR便是在.NET平台上为动态语言提供了统一的编程模型。CLR本身已经有一些支持动态编程能力，如反射，Emit等等。不过在.NET上实现动态语言的时候，总会一遍又一遍地去实现某些功能，还有如动态语言如何与静态语言进行交互，这些都由DLR来提供。DLR的特性包含了，如表达式树、动态分发、Call Site缓存，这可以提高动态代码的执行效率。

在.NET 4.0中我们使用了DLR，不仅仅是IronPython和IronRuby，还有C# 4和VB.NET 10，它们使用DLR实现动态分发功能。因此我们共享了语言的动态能力实现方式，于是这些语言之间可以轻松地进行交互。同样我们可以与其他多样性的技术进行交互，例如使用JavaScript操作Silverlight的DOM，或是与Ruby、Python代码沟通，甚至用来控制Office等自动化服务。

## 五、元编程 动态语言所擅长的“元编程” 编译器即服务。

这是[Anders Hejlsberg](http://en.wikipedia.org/wiki/Anders_Hejlsberg)（不用介绍这是谁了吧）在[比利时TechDays 2010所做的开场演讲](http://channel9.msdn.com/posts/adebruyn/TechDays-2010-Developer-Keynote-by-Anders-Hejlsberg/)。由于最近我在博客上关于语言的讨论比较多，出于应景，也打算将Anders的演讲完整地听写出来。在上一部分中，Anders谈及了他眼中编程语言的另一个发展趋势：动态性。在这一部分中，Anders则讨论了动态语言所擅长的“元编程”，并简单介绍了他为静态类型语言所设计的一种改进方案：编译器即服务。

如果没有特别说明，所有的文字都直接翻译自Anders的演讲，并使用我自己的口语习惯表达出来，对于Anders的口误及反复等情况，必要时在译文中自然也会进行忽略。为了方便理解，我也会将视频中关键部分进行截图，而某些代码演示则会直接作为文章内容发表。

（听写开始，接[上篇](http://blog.zhaojie.me/2010/05/trends-and-future-directions-in-programming-languages-by-anders-4-dynamic-languages.html)）

## 动态语言的另一个关键和有趣之处在于“[元编程](http://en.wikipedia.org/wiki/Metaprogramming)”。“元编程”实际上是“代码生成”的一种别称，

其实在日常应用中我们也经常依赖这种做法。观察动态语言适合元编程的原因也是件十分有趣的事情。

在这个蓝框中是一段Ruby on Rails代码（见上图）。简单地说，这里定义了一个Order类，继承了ActiveRecord，也定义了一些关系，如belongs\_to和has\_many关系。Ruby这种动态语言的关键之处，在于一切事物都是通过执行而得到的，包括类型声明。比如这里的类型申明执行了belongs\_to和has\_many方法的调用，执行belongs\_to会截获一对多或一对一关系所需要的信息，因此在这里语言是在运行的时候，动态为自身生成了代码。

实现这点在动态语言里自然会更容易一些，因为它们没有编译期和执行期的区别。静态类型语言在这方面会比较困难。例如在C#或Java里使用ORM时，传统的做法是让代码生成器去观察数据库，生成一大堆代码，然后再编译，有些复杂。不过我时常想着去改善这一点。

## 其中一种做法，是我们正在努力实现的“编译器即服务”，

我现在先对它进行一些简单的介绍。传统的编译器像是一个黑盒，你在一端输入代码，而另一端便会生成.NET程序集或是对象代码等等。而这个黑盒却很神秘，你目前很难参与或理解它的工作。

你可以想象，一些代码往往是不包含在源文件中的。如果你想要交互式编程的体验，例如一个交互式的提示符，那么代码不是保存在源文件中而是由用户输入的。如果您在实现一个DSL，例如Windows Workflow或是Biztalk，则可能用C#或VB实现了一些需要动态执行的规则，它们也不是保存在源文件中，而可能是放在XML属性中的。此时你想编译它们却做不到，你还是要把它们放入源文件，这就变的复杂了。

另一方面，对于编译器来说，我们不一定需要它生成程序集，有时候需要的是一些树状的表现形式。例如一些由用户反射生成的代码，便可能不要程序集而是一个解析树，然后可以对它进行识别和重写。因此，我们可能越来越需要的是一些API，以此开放编译器的功能。

例如，你可以给它一小段代码，让它返回一段可执行的程序，或是一个可以识别或重写的解析树。这么做可以让静态类型语言获得许多有用的功能，例如元编程，以及可操作的完整的对象模型等等。我们正在朝这方面努力，我也会在下午1点的C# 4.0演讲中谈论更多这方面的内容。

# 六、并发

这是[Anders Hejlsberg](http://en.wikipedia.org/wiki/Anders_Hejlsberg)（不用介绍这是谁了吧）在[比利时TechDays 2010所做的开场演讲](http://channel9.msdn.com/posts/adebruyn/TechDays-2010-Developer-Keynote-by-Anders-Hejlsberg/)。由于最近我在博客上关于语言的讨论比较多，出于应景，也打算将Anders的演讲完整地听写出来。在上一部分中，Anders谈论了“元编程”及他正在努力的“编译器即服务”功能。在这一部分中，Anders则谈论了“并发”，这也是他眼中编程语言发展的三种趋势之一，并演示了.NET 4.0中并行库的神奇效果。

如果没有特别说明，所有的文字都直接翻译自Anders的演讲，并使用我自己的口语习惯表达出来，对于Anders的口误及反复等情况，必要时在译文中自然也会进行忽略。为了方便理解，我也会将视频中关键部分进行截图，而某些代码演示则会直接作为文章内容发表。

（听写开始，接[上篇](http://blog.zhaojie.me/2010/05/trends-and-future-directions-in-programming-languages-by-anders-5-meta-programming.html)）

好，最后我想谈的内容是“并发”。

听说过摩尔定律的请举手……几乎是所有人。那么多少人听说了摩尔定律已经结束了呢？嗯，还是有很多人。我有好消息，也有坏消息。我认为摩尔定律并没有停止。摩尔定律说的是：可以在集成电路上低成本地放置晶体管的数目，约每两年便会增加一倍。有趣的是，这个定律从60年代持续到现在，而从一些迹象上来看，这个定律会继续保持20到30年。

摩尔定理有个推论，便是说时钟速度将根据相同的周期提高，也就是说每隔大约24个月，CPU的速度便会加倍──而这点已经停止了。再来统计一下，你们之中有谁的机器里有20GHz的CPU？看到了没？一个人都没有。但如果你从五年前开始计算的话，现在我们应该已经在使用20GHz的CPU了，但事实并非如此。这点在五年前就停止了，而且事实上最大速度还有些下降，因为发热量实在太大了，会消耗许多能源，让电池用的太快。

有些物理方面的基础因素让CPU不能运行的太快。然而，另一意义上的摩尔定理出现了。我们还是可以看到容量的增加，因为可以在同一个表盘上放置多个CPU了。目前已经有了双核、四核，Intel的CTO在三年前说，十年后我们可以出现80核的处理器。

到了那个时候，你的任务管理器中就可能是这样的。似乎有些吓人，不过这是我们实验室中真实存在的128核机器。你可以看到，计算能力已经完全用上了。这便是个问题，比如你在这台强大的机器上进行一个实验，你自然希望看到100%的使用状况，不过传统的实验都是在一个核上执行的，所以我们面临的挑战是，我们需要换一种写程序的方式来利用此类机器。

我的一个同事，Herb Sutter，他写过一篇文章，谈到“[免费的午餐已经结束了](http://www.gotw.ca/publications/concurrency-ddj.htm)”。没错，我们已经不能写一个程序，然后对客户说：啊，未来的硬件会让它运行的越来越快，我们不用关心太多──不，已经不会这样了，除非你换种不同的写法。实话说，这是个挑战，也是个机遇。说它是个挑战，是因为并发十分困难，至今我们对此还没有简单的答案，稍后我会演示一些正有所改善的东西，但……这也是一个机遇，在这样的机器上，你的确可以用完所有的核，这样便能获得性能提高，不过做法需要有所不同。

多核革命的一个有趣之处在于，它对于并发的思维方式会有所改变。传统的并发思维是在单个CPU上执行多个逻辑任务，使用旧有的分时方式、时间片模型来执行多个任务。但是，你想一下便会发现如今的并发情况正好相反，现在是要将一个逻辑上的任务放在多个CPU上执行。这改变了我们编写程序的方式，这意味着对于语言或是API来说，我们需要有办法来分解任务，把它拆分成多个小任务后独立的执行，而传统的编程语言中并不关注这点。

使用目前的并发API来完成工作并不容易，比如使用Thread，ThreadPool，lock，Monitor等等，你无法太好的进展。不过.NET 4.0提供了一些美妙的事物，我们称之为.NET并行扩展。它是一种现代的并发模型，将逻辑上的任务并发与我们实际使用的的物理模型分离开来。以前我们的API都是直接处理线程，也就是（上图）下方橙色的部分，不过有了.NET并行扩展之后，你可以使用更为逻辑化的编程风格。任务并行库（Task Parallel Library），并行LINQ（Parallel LINQ）以及协调数据结构（Coordination Data Structures）让你可以直接关注逻辑上的任务，而不必关心它们是如何运行的，或是使用了多少个线程和CPU等等。

下面我来简单演示一下它们的使用方式。我带来了一个PLINQ演示，这里是一些代码，读取XML文件的内容。这有个50M大小的popname.xml文件，保存了美国社会安全数据库里的信息，包含某个洲在某一年的人口统计信息。这个程序会读取这个XML文件，把它转化成一系列对象，并存放在一个List中。然后对其执行一个LINQ语句，查找所有在华盛顿名叫Robert的人，再根据年份进行排序：

Console.WriteLine("Loading XML data...");

var popNames =

(from e in XElement.Load("popnames.xml").Elements("Name")

select new

{

Name = (string)e.Attribute("Name"),

State = (string)e.Attribute("State"),

Year = (int)e.Attribute("Year"),

Count = (int)e.Attribute("Count")

})

.ToList();

Console.WriteLine(popNames.Count + " records");

Console.WriteLine();

string targetName = "Robert";

string targetState = "WA";

var querySequential =

from n in popNames

where n.Name == targetName && n.State == targetState

orderby n.Year

select n;

我们来执行一下……首先加载XML文件，然后进行查询。利用PLINQ我们可以做到并行地查询。我们只要拷贝一份代码……改成queryParallel……现在我唯一要做的只是在数据源上使用AsParallel扩展方法，这样便会引入一套新的类型和实现，此时相同的LINQ操作使用的便是并行的实现：

var queryParallel =

from n in popNames.AsParallel()

where n.Name == targetName && n.State == targetState

orderby n.Year

select n;

我们重新执行两个查询。

再次加载XML数据……并行实现使用了1.5秒，我们再试着运行一次，一般结果会更好一些，现在可能刚好在执行一些后台任务。一般我们可以得到更快的结果……这次比较接近了。现在你可以观察到，我们并不需要做太多事情，便可以在我的双核机器上得到并发的效果。

这里我无法保证说，我们只要随时加上AsParallel便可以得到两倍的性能，有时可以有时不行，有些查询能够被并行，有的则不可以。然而，我想你一定同意一点，使用如LINQ这样的DSL能够方便我们编写并行的代码，也更有可能利用起并行效果。虽然不是每次都有效，但是尝试的成本也很低。如果我们使用普通的for循环来编写代码，在某个地方使用线程池等等，便很容易在这些API里失去方向。而这里我们只要简单地尝试一下，便能知道是否可以提高性能了。

这里你已经看到我使用的LINQ查询，而现在也有很多工作是通过循环来完成的。你可以想象主要的运算是从哪里来的，很自然会是在循环里操作数据。如果循环的每个迭代都是独立的，便有很大的机会可以利用并发操作──我知道这里是“如果”，不过长期来看则一定会出现这样的情况。这时候便可以使用并行扩展，或者说是.NET并行扩展里的新API，把循环转化成并行的循环，只要简单的改变……几乎只要用同样的循环体把for重构成Parallel.For就行了。如果你有foreach操作就可以使用Parallel.ForEach，或是一系列顺序执行的语句也可以用上Parallel.Invoke。此时任务并行库会接管并执行这些任务，根据你的CPU数量使用最优化的线程数量，你不需要关注更深的细节，只需要编写逻辑就可以了。

就像我说的那样，可能你会有独立的任务但也可能没有，所以很多时候我们需要编程语言来关注这方面的事情。比如“隔离性（Isolation）”。例如，编译器如何发现这段代码是独立的，可以安全地并发执行，好比我创建了一个对象，在分享给其他人之前，我对它的改变是安全的。但是我一旦把它们共享出去了，那么它们便不安全了。所以如果我们的类型系统可以跟踪到这样的共享，如[Linear Types](http://en.wikipedia.org/wiki/Linear_type_system)──这在学术界也有一些研究。我们也可以在函数的纯洁性（Purity）方面下功夫，如关注某个函数是否有副作用，有些时候编译器可以做这方面的检查，它可以禁止某些操作，以此保证我们写出纯函数。还有便是不可变性（Immutability），目前的C#或VB，我们需要额外的工作才能写出不可变的代码──但本不该这样，我们应该在语言层面上更好的支持不可变性。这些都是在并发方面需要考虑的问题。

如果说有哪个语言特性超出这个范畴，我想说这里还有一个原则：你不该期望C#中出现某个特别的并发模型，而应该是一种通用的，可用于各种不同的并发场景的特性，就像隔离性、纯洁性及不可变性那样。语言拥有这样的特性之后，就可以用于构建各种不同的API，各种并发方式都可以利用到核心的语言特性。

七、[总结](http://blog.zhaojie.me/2010/06/trends-and-future-directions-in-programming-languages-by-anders-7-conclusion.html)

这是[Anders Hejlsberg](http://en.wikipedia.org/wiki/Anders_Hejlsberg)（不用介绍这是谁了吧）在[比利时TechDays 2010所做的开场演讲](http://channel9.msdn.com/posts/adebruyn/TechDays-2010-Developer-Keynote-by-Anders-Hejlsberg/)。由于最近我在博客上关于语言的讨论比较多，出于应景，也打算将Anders的演讲完整地听写出来。在上一部分中，Anders谈论了“并发”，这也是他眼中编程语言发展的三种趋势之一，并演示了.NET 4.0中并行库的神奇效果。现在则是此次演讲的最后一部分，Anders对整场演讲内容进行了总结和回顾（本文较短，主要内容请参考之前的文章）。

如果没有特别说明，所有的文字都直接翻译自Anders的演讲，并使用我自己的口语习惯表达出来，对于Anders的口误及反复等情况，必要时在译文中自然也会进行忽略。为了方便理解，我也会将视频中关键部分进行截图，而某些代码演示则会直接作为文章内容发表。

（听写开始，接[上篇](http://blog.zhaojie.me/2010/05/trends-and-future-directions-in-programming-languages-by-anders-6-concurrency.html)）



OK，我想现在已经讲的差不多了，我来做个总结吧。

在我看来，对于编程语言来说，现在出现了许多有趣的东西，也是令人激动的时刻。在过去，大约1995-2005年，的确可以说是一个有些特别的编程语言的黄金时期。你知道，当Java出现的时候，编程语言的门槛变得平坦了，一切都是Java，天啊其他编程语言都完蛋了，我们也没什么可做的了。然后我们又逐渐发现，这远没有结束，现在回顾起来，会发现又出现了许多有趣的编程语言。我很兴奋，因为新语言代表了我们在编程领域上的进步。

如果要我概括在未来十年编程语言会变成什么样，首先，我认为编程语言应该变得更加“声明式”，我们需要设法为语言引入一些如元编程，函数式编程的能力，同时可能也要寻找让用户有办法扩展语法，使他们可以构造领域特定语言等等。我想在十年以后，动态语言和静态语言的区别也差不多会消失了，这两者会合并为一种单一的常见的编程范式。在并发方面，语言会采纳一些特性，可以利用起隔离性，函数式的纯粹性，以及更好的不可变数据类型的编写方式。不过总体来说我想强调的是，对于编程语言，新的范式则是“多范式”编程语言。

这就是我现在对编程语言的看法，希望我没有给你带来过多内容。如果你对C# 4.0有更多兴趣，今天下午一点我还有一个讲座。多谢捧场，希望你对这次会议感到满意。谢谢。