**使用类型 dynamic（C# 编程指南）**

Visual C# 2010 引入了一个新类型 **dynamic**。该类型是一种静态类型，但类型为 **dynamic** 的对象会跳过静态类型检查。大多数情况下，该对象就像具有类型 **object** 一样。在编译时，将假定类型化为 **dynamic** 的元素支持任何操作。因此，您不必考虑对象是从 COM API、从动态语言（例如 IronPython）、从 HTML 文档对象模型 (DOM)、从反射还是从程序中的其他位置获取自己的值。但是，如果代码无效，则在运行时会捕获到错误。

例如，如果以下代码中的实例方法 exampleMethod1 只有一个形参，则编译器会将对该方法的第一个调用 ec.exampleMethod1(10, 4) 识别为无效，原因是它包含两个实参。该调用将导致编译器错误。编译器不会检查对该方法的第二个调用 dynamic\_ec.exampleMethod1(10, 4)，原因是 dynamic\_ec 的类型为 **dynamic**。因此，不会报告编译器错误。但是，该错误不会被无限期疏忽。它将在运行时被捕获，并导致运行时异常。

static void Main(string[] args)

{

ExampleClass ec = new ExampleClass();

// The following call to exampleMethod1 causes a compiler error

// if exampleMethod1 has only one parameter. Uncomment the line

// to see the error.

//ec.exampleMethod1(10, 4);

dynamic dynamic\_ec = new ExampleClass();

// The following line is not identified as an error by the

// compiler, but it causes a run-time exception.

dynamic\_ec.exampleMethod1(10, 4);

// The following calls also do not cause compiler errors, whether

// appropriate methods exist or not.

dynamic\_ec.someMethod("some argument", 7, null);

dynamic\_ec.nonexistentMethod();

}

class ExampleClass

{

public ExampleClass() { }

public ExampleClass(int v) { }

public void exampleMethod1(int i) { }

public void exampleMethod2(string str) { }

}

在这些示例中，编译器的作用是将有关每个语句的预期作用的信息一起打包到类型化为 **dynamic** 的对象或表达式。在运行时，将对存储的信息进行检查，并且任何无效的语句将会导致运行时异常。

大多数动态操作的结果是其本身 **dynamic**。例如，如果将鼠标指针放在以下示例中使用的 testSum 上，则 IntelliSense 将显示类型**“(局部变量)dynamic testSum”**。

dynamic d = 1;

var testSum = d + 3;

// Rest the mouse pointer over testSum in the following statement.

System.Console.WriteLine(testSum);

结果不是 **dynamic** 的操作包括从 **dynamic** 到另一种类型的转换，以及包括类型为 **dynamic** 的参数的构造函数调用。例如，以下声明中 testInstance 的类型为 ExampleClass，而不是 **dynamic**。

var testInstance = new ExampleClass(d);

下一节“转换”中显示了转换示例。

**转换**

动态对象和其他类型之间的转换非常简单。这样，开发人员将能够在动态行为和非动态行为之间切换。

任何对象都可隐式转换为动态类型，如下面的示例所示。

dynamic d1 = 7;

dynamic d2 = "a string";

dynamic d3 = System.DateTime.Today;

dynamic d4 = System.Diagnostics.Process.GetProcesses();

反之，隐式转换也可动态地应用于类型为 **dynamic** 的任何表达式。

int i = d1;

string str = d2;

DateTime dt = d3;

System.Diagnostics.Process[] procs = d4;

**使用类型为 dynamic 的参数重载决策**

如果方法调用中的一个或多个参数具有类型 **dynamic**，或者方法调用的接收方的类型为 **dynamic**，则会在运行时（而不是在编译时）进行重载决策。在下面的示例中，如果唯一可访问的 exampleMethod2 方法定义为接受字符串参数，则将 d1 作为参数发送不会导致编译器错误，但却会导致运行时异常。重载决策之所以会在运行时失败，原因是 d1 的运行时类型为 **int**，而 exampleMethod2 需要字符串。

// Valid.

ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not

// dynamic. A run-time exception is raised because the run-time type of d1 is int.

ec.exampleMethod2(d1);

// The following statement does cause a compiler error.

//ec.exampleMethod2(7);

**动态语言运行时**

动态语言运行时 (DLR) 是 .NET Framework 4 中的一个新 API。它提供了支持 C# 中的 **dynamic** 类型的基础结构，还提供了诸如 IronPython 和 IronRuby 等动态编程语言的实现。有关 DLR 的更多信息，请参见[动态语言运行时概述](https://msdn.microsoft.com/zh-cn/library/dd233052.aspx)。

**COM 互操作**

Visual C# 2010 包括若干功能，这些功能改善了与 COM API（例如 Office 自动化 API）的互操作体验。这些改进之处包括 **dynamic** 类型以及[命名参数和可选参数](https://msdn.microsoft.com/zh-cn/library/dd264739.aspx)的使用。

通过将类型指定为 **object**，许多 COM 方法都允许参数类型和返回类型发生变化。这样，就必须显式强制转换值，以便与 C# 中的强类型变量保持协调。如果使用 [/link (C# Compiler Options)](https://msdn.microsoft.com/zh-cn/library/dd264728.aspx) 选项进行编译，则 **dynamic** 类型的引入使您能够将 COM 签名中出现的 **object** 实例看作 **dynamic** 类型，从而避免了大量的强制转换。例如，下面的语句对比了在使用 **dynamic** 类型和不使用 **dynamic** 类型的情况下如何访问 Microsoft Office Excel 电子表格中的单元格。

// Before the introduction of dynamic.

((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";

Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];

// After the introduction of dynamic, the access to the Value property and

// the conversion to Excel.Range are handled by the run-time COM binder.

excelApp.Cells[1, 1].Value = "Name";

Excel.Range range2010 = excelApp.Cells[1, 1];