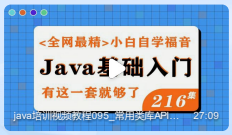
Lambda表达式

播报 编辑 讨论 上传视频

匿名函数



秒懂百科
一看就懂的视频百科

Lambda 表达式（lambda expression）是一个匿名函数，Lambda表达式基于数学中的λ演算得名，直接对应于其中的lambda抽象（lambda abstraction），是一个匿名函数，即没有函数名的函数。Lambda表达式可以表示闭包（注意和数学传统意义上的不同）。

中文名	Lambda表达式	学　科	程序设计
外文名	Lambda expression	属　性	匿名函数
名称起源	λ演算	关联函数	lambda抽象

Lambda表达式的概述图（1张）

目录

C#表达式

播报 编辑

C#的Lambda 表达式都使用 Lambda 运算符 =>，该运算符读为“goes to”。语法如下：

```
(object argOne, object argTwo) => {; /*Your statement goes here*/}
```

函数体多于一条语句的可用大括号括起。

类型

可以将此表达式分配给委托类型，如下所示：

```
delegate int del(int i);
del myDelegate=x=>{return x*x;};
int j = myDelegate(5); //j=25
```

创建表达式目录树类型：

```
using System.Linq.Expressions;
//...
Expression <del>=x=>x*x;
```

=> 运算符具有与赋值运算符 (=) 相同的优先级，并且是右结合运算符。

Lambda 用在基于方法的 LINQ 查询中，作为诸如 Where 和 Where 等标准查询运算符方法的参数。

使用基于方法的语法在 Enumerable 类中调用 Where 方法时（像在 LINQ to Objects 和 LINQ to XML 中那样），参数是委托类型 System.:::Func<(Of <(T, TResult>)>)。使用 Lambda 表达式创建委托最为方便。例如，当您在 System.Linq.:::.Queryable 类中调用相同的方法时（像在 LINQ to SQL 中那样），则参数类型是 System.Linq.Expressions.:::.Expression<Func>，其中 Func 是包含至多五个输入参数的任何 Func 委托。同样，Lambda 表达式只是一种用于构造表达式目录树的非常简练的方式。尽管事实上通过 Lambda 创建的对象的类型是不同的，但 Lambda 使得 Where 调用看起来类似。

在前面的示例中，请注意委托签名具有一个 int 类型的隐式类型输入参数，并返回 int。可以将 Lambda 表达式转换为该类型的委托，因为该表达式也具有一个输入参数 (x)，以及一个编译器可隐式转换为 int 类型的返回值。（以下几节中将对类型推理进行详细讨论。）使用输入参数 5 调用委托时，它将返回结果 25。

在 is 或 as 运算符的左侧不允许使用 Lambda。

适用于匿名方法的所有限制也适用于 Lambda 表达式。有关更多信息，请参见匿名方法（C# 编程指南）。

特殊

下列规则适用于 Lambda 表达式中的变量范围：

捕获的变量将不会被作为垃圾回收，直至引用变量的委托超出范围为止。

在外部方法中看不到 Lambda 表达式内引入的变量。

Lambda 表达式无法从封闭方法中直接捕获 ref 或 out 参数。

Lambda 表达式中的返回语句不会导致封闭方法返回。

Lambda 表达式不能包含其目标位于所包含匿名函数主体外部或内部的 goto 语句、break 语句或 continue 语句。

Lambda表达式的本质是“匿名方法”，即当编译我们的程序代码时，“编译器”会自动将“Lambda表达式”转换为“匿名方法”，如下例：

```
string[] names={"agen","balen","coure","apple"};
string[] findNameA=Array.FindAll<string>(names,delegate(string v){return v.StartsWith("a");});
string[] findNameB=Array.FindAll<string>(names,v=>v.StartsWith("a"));
```

上面中两个FindAll方法的反编译代码如下：

```
string[] findNameA=Array.FindAll<string>(names,delegate(string v){return v.StartsWith("a");});
string[] findNameB=Array.FindAll<string>(names,delegate(string v){return v.StartsWith("a");});
```

从而可以知道“Lambda表达式”与“匿名方法”是可以划上等号的，只不过使用“Lambda表达式”输写代码看上去更直观漂亮，不是吗？

Lambda表达式的语法格式：

参数列表 => 语句或语句块 [1]

其中“参数列”中可包含任意个参数(与委托对应)，如果参数列中有0个或1个以上参数，则必须使用括号括住参数列，如下：

```
x => x + 1                          // Implicitly typed, expression body
x => { return x + 1; }              // Implicitly typed, statement body
(int x) => x + 1                    // Explicitly typed, expression body
(int x) => { return x + 1; }        // Explicitly typed, statement body
(x, y) => x * y                     // Multiple parameters
() => Console.WriteLine()           // No parameters
async (t1,t2) => await t1 + await t2    // Async
delegate (int x) { return x + 1; }  // Anonymous method expression
delegate { return 1 + 1; }          // Parameter list omitted
```

如果“语句或语句块”有返回值时，如果只有一条语句则可以不输写“return”语句，编译器会自动处理，否则必须加上，如下示例：

“Lambda表达式”是委托的实现方法，所以必须遵循以下规则：

1）“Lambda表达式”的参数数量必须和“委托”的参数数量相同；

2）如果“委托”的参数中包括有ref或out修饰符，则“Lambda表达式”的参数列中也必须包括有修饰符；

例子：

```
class Test
{
    delegate int AddHandler(int x,int y);
    static void Print(AddHandler add);
    {
        Console.Write(add(1, 3));
    }
    static void Main(string[] args)
    {
        Print((x,y) => x+y);
        Print((x,y) => {int v=x*10; return y+v;});
        Console.ReadKey();
    }
}
```

注： 如果包括有修饰符，则"Lambda表达式"中的参数列中也必须加上参数的类型

3）如果"委托"有返回类型，则"Lambda表达式"的"语句或语句块"中也必须返回相同类型的数据；

4）如果"委托"有几种数据类型格式而在"Lambda表达式"中"编译器"无法推断具体数据类型时，则必须手动明确数据类型。

例子：

(错误代码)

```
class Test
{
    delegate AddHandler<T> (Tx, Ty);

    static void Print(AddHandler<int> test)
    {
        Console.WriteLine("int type:{0}",test(1, 2));
    }
    static void Print(AddHandler<double> test)
    {
        Console.WriteLine("doubletype:{0}",test(1d, 2d));
    }
    static void Main(string[] args)
    {
        Print((x, y) => x+y);
        Console.ReadKey();
    }
}
```

当我们编译以下代码时，编译器将会显示以下错误信息：

```
在以下方法或属性之间调用不明确：
"ConsoleApplication1.Test.Print(ConsoleApplication1.Test.AddHandler<int>)"
和"ConsoleApplication1.Test.Print(ConsoleApplication1.Test.AddHandler<double>)"
```

所以我们必须明确数据类型给编译器，如下：

```
Print((int x, int y) => x+y);
```

这样我们的代码就能编译通过了。

Java表达式

播报 编辑

Java 8的一个大亮点是引入Lambda表达式，使用它设计的代码会更加简洁。当开发者在编写Lambda表达式时，也会随之被编译成一个函数式接口。下面这个例子就是使用Lambda语法来代替匿名的内部类，代码不仅简洁，而且还可读。

没有使用Lambda的老方法：

```
button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent actionEvent){
        System.out.println("Action detected");
    }
});
```

使用Lambda：

```
button.addActionListener( actionEvent -> {
    System.out.println("Action detected");
});
```

让我们来看一个更明显的例子。

不采用Lambda的老方法：

```
Runnable runnable1=new Runnable(){
@Override
public void run(){
    System.out.println("Running without Lambda");
}
};
```

使用Lambda：

```
Runnable runnable2=()->System.out.println("Running from Lambda");
```

正如你所看到的，使用Lambda表达式不仅让代码变的简单、而且可读、最重要的是代码量也随之减少很多。然而，在某种程度上，这些功能在Scala等这些JVM语言里已经被广泛使用。

并不奇怪，Scala社区是难以置信的，因为许多Java 8里的内容看起来就像是从Scala里搬过来的。在某种程度上，Java 8的语法要比Scala的更详细但不是很清晰，但这并不能说明什么，如果可以，它可能会像Scala那样构建Lambda表达式。

一方面，如果Java继续围绕Lambda来发展和实现Scala都已经实现的功能，那么可能就不需要Scala了。另一方面，如果它只提供一些核心的功能，例如帮助匿名内部类，那么Scala和其他语言将会继续茁壮成长，并且有可能会凌驾于Java之上。其实这才是最好的结果，有竞争才有进步，其它语言继续发展和成长，并且无需担心是否会过时。

C++表达式

播报 编辑

ISO C++ 11 标准的一大亮点是引入Lambda表达式。基本语法如下：

```
[capture list] (parameter list) -> return type { function body }
```

其中除了"[ ]"（其中捕获列表可以为空）和"复合语句"（相当于具名函数定义的函数体），其它都是可选的。它的类型是单一的具有成员operator()的非联合的类类型，称为闭包类型（closure type）。

C++中，一个lambda表达式表示一个可调用的代码单元。我们可以将其理解为一个未命名的内联函数。它与普通函数不同的是，lambda必须使用尾置返回来指定返回类型。

例如调用<algorithm>中的std::sort，ISO C++ 98 的写法是要先写一个compare函数：

```
bool compare(int& a,int& b)
{
    return a>b;
}
```

然后，再这样调用：

```
sort(a, a+n, compare);
```

然而，用ISO C++ 11 标准新增的Lambda表达式，可以这么写：

```
sort(a, a+n, [](int a,int b){return a>b;});//降序排序
```

这样一来，代码明显简洁多了。

由于Lambda的类型是单一的，不能通过类型名来显式声明对应的对象，但可以利用auto关键字和类型推导：

```
auto f=[](int a,int b){return a>b;};
```

和其它语言的一个较明显的区别是Lambda和C++的类型系统结合使用，如：

```
auto f=[x](int a,int b){return a>x;};//x被捕获复制
int x=0, y=1;
auto g=[&](int x){return ++y;};//y被捕获引用，调用g后会修改y，需要注意y的生存期
bool(*fp)(int, int)=[](int a,int b){return a>b;};//不捕获时才可转换为函数指针
```

Lambda表达式可以嵌套使用。

ISO C++14支持基于类型推断的泛型lambda表达式。上面的排序代码可以这样写：

```
sort(a, a+n, [](const auto& a,const auto& b){return a>b;});//降序排序：不依赖a和b的具体类型
```

因为参数类型和函数模板参数一样可以被推导而无需和具体参数类型耦合，有利于重构代码；和使用auto声明变量的作用类似，它也允许避免书写过于复杂的参数类型。特别地，不需要显式指出参数类型使使用高阶函数变得更加容易。

lambda表达式有些部分是可以省略的，所以一个最简单的lambda表达式可以是下面这样，这段代码是可以通过编译的：

[] {}; // lambda expression

Python表达式

播报 编辑

Lambda表达式是Python中一类特殊的定义函数的形式，使用它可以定义一个匿名函数。与其它语言不同，Python的Lambda表达式的函数体只能有单独的一条语句，也就是返回值表达式语句。其语法如下： [2]

lambda 形参列表 : 函数返回值表达式语句

下面是个Lambda表达式的例子：

```
#!/usr/bin/envpython
li=[{"age":20,"name":"def"},{"age":25,"name":"abc"},{"age":10,"name":"ghi"}]
li=sorted(li, key=lambda x:x["age"])
print(li)
```

如果不用Lambda表达式，而要写成常规的函数，那么需要这么写：

```
#!/usr/bin/envpython
def comp(x):
    return x["age"]
li=[{"age":20,"name":"def"},{"age":25,"name":"abc"},{"age":10,"name":"ghi"}]
li=sorted(li, key=comp)
print(li)
```

词条图册

更多图册 >

Lambda表达式的示例(1)

参考资料

1 Expressions . Microsoft C# Language Specification. 2017-07-01[引用日期2018-06-06]

2 PEP 312 -- Simple Implicit Lambda . Python Enhancement Proposals. 2003-02-11[引用日期2018-06-06]

猜你喜欢

相关搜索