C++11 新特性：Lambda 表达式 | DevBean's World

C++11 新特性：Lambda 表达式

豆子 | 2012年5月15日 | C++ | 6条评论

参考文章：https://blogs.oracle.com/pcarlini/entry/c_1x_tidbits_lambda_expressions

或许，Lambda 表达式算得上是 C++ 11 新增特性中最激动人心的一个。这个全新的特性听起来很深奥，但却是很多其他语言早已提供（比如 C#）或者即将提供（比如 Java）的。简而言之，Lambda 表达式就是用于创建匿名函数的。GCC 4.5.x 和 Microsoft Visual Studio 早已提供了对 lambda 表达式的支持。在 GCC 4.7 中，默认是不开启 C++ 11 特性的，需要添加 `-std=c++11` 编译参数。而 VS2010 则默认开启。

为什么说 lambda 表达式如此激动人心呢？举一个例子。标准 C++ 库中有一个常用算法的库，其中提供了很多算法函数，比如 sort() 和 find()。这些函数通常需要提供一个“谓词函数 predicate function”。所谓谓词函数，就是进行一个操作用的临时函数。比如 find() 需要一个谓词，用于查找元素满足的条件；能够满足谓词函数的元素才会被查找出来。这样的谓词函数，使用临时的匿名函数，既可以减少函数数量，又会让代码变得清晰易读。

下面来看一个例子：

C++

```
#include <algorithm>
#include <cmath>

void abssort(float *x, unsigned N)
{
  std::sort(x,
            x + N,
            [](float a, float b) { return std::abs(a) < std::abs(b); });
}
```

从上面的例子来看，尽管支持 lambda 表达式，但 C++ 的语法看起来却很“神奇”。lambda 表达式使用一对方括号作为开始的标识，类似于声明一个函数，只不过这个函数没有名字，也就是一个匿名函数。这个匿名函数接受两个参数， a 和 b ；其返回值是一个 bool 类型的值，注意，返回值是自动推断的，不需要显式声明，不过这是有条件的！条件就是，lambda 表达式的语句只有一个 return。函数的作用是比较 a、b 的绝对值的大小。然后，在此例中，这个 lambda 表达式作为一个闭包被传递给 `std::sort()` 函数。

下面，我们来详细解释下这个神奇的语法到底代表着什么。

我们从另外一个例子开始：

C++

```
std::cout << [](float f) { return std::abs(f); } (-3.5);
```

输出值是什么？3.5！注意，这是一个函数对象（由 lambda 表达式生成），其实参是 -3.5，返回值是参数的绝对值。lambda 表达式的返回值类型是语言自动推断的，因为 std::abs() 的返回值就是 float。注意，前面我们也提到了，只有当 lambda 表达式中的语句“足够简单”，才能自动推断返回值类型。

C++ 11 的这种语法，其实就是匿名函数声明之后马上调用（否则的话，如果这个匿名函数既不调用，又不作为闭包传递给其它函数，那么这个匿名函数就没有什么用处）。如果你觉得奇怪，那么来看看 JavaScript 的这种写法：

JavaScript

```
function() {} ();

function(a) {} (-3.5);
```

C++ 11 的写法完全类似 JavaScript 的语法。

如果我不想让 lambda 表达式自动推断类型，或者是 lambda 表达式的内容很复杂，不能自动推断怎么办？比如， std::abs(float) 的返回值是 float，我想把它强制转型为 int。那么，此时，我们就必须显式指定 lambda 表达式返回值的类型：

C++

```
std::cout << [](float f) -> int { return std::abs(f); } (-3.5);
```

这个语句与前面的不同之处在于，lambda 表达式的返回时不是 float 而是 int。也就是说，上面语句的输出值是 3。返回值类型的概念同普通的函数返回值类型是完全一样的。

引入 lambda 表达式的前导符是一对方括号，称为 lambda 引入符（lambda-introducer）。lambda 引入符是有其自己的作用的，不仅仅是表明一个 lambda 表达式的开始那么简单。lambda 表达式可以使用与其相同范围 scope 内的变量。这个引入符的作用就是表明，其后的 lambda 表达式以何种方式使用（正式的术语是“捕获”）这些变量（这些变量能够在 lambda 表达式中被捕获，其实就是构成了一个闭包）。目前为止，我们看到的仅仅是一个空的方括号，其实，这个引入符是相当灵活的。例如：

C++

```
float f0 = 1.0;
std::cout << [=](float f) { return f0 + std::abs(f); } (-3.5);
```

其输出值是 4.5。[=] 意味着，lambda 表达式以传值的形式捕获同范围内的变量。另外一个例子：

C++

```
float f0 = 1.0;
std::cout << [&](float f) { return f0 += std::abs(f); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

输出值是 4.5 和 4.5。[&] 表明，lambda 表达式以传引用的方式捕获外部变量。那么，下一个例子：

```C++
float f0 = 1.0;
std::cout << [=](float f) mutable { return f0 += std::abs(f); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

这个例子很有趣。首先，[=] 意味着，lambda 表达式以传值的形式捕获外部变量。C++ 11 标准说，如果以传值的形式捕获外部变量，那么，lambda 体不允许修改外部变量，对 f0 的任何修改都会引发编译错误。但是，注意，我们在 lambda 表达式前声明了 mutable 关键字，这就允许了 lambda 表达式体修改 f0 的值。因此，我们的例子本应报错，但是由于有 mutable 关键字，则不会报错。那么，你会觉得输出值是什么呢？答案是，4.5 和 1.0。为什么 f0 还是 1.0？因为我们是传值的，虽然在 lambda 表达式中对 f0 有了修改，但由于是传值的，外部的 f0 依然不会被修改。

上面的例子是，所有的变量要么传值，要么传引用。那么，是不是有混合机制呢？当然也有！比如下面的例子：

```C++
float f0 = 1.0f;
float f1 = 10.0f;
std::cout << [=, &f0](float a) { return f0 += f1 + std::abs(a); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

这个例子的输出是 14.5 和 14.5。在这个例子中，f0 通过引用被捕获，而其它变量，比如 f1 则是通过值被捕获。

下面我们来总结下所有出现的 lambda 引入符：

- [] // 不捕获任何外部变量
- [=] // 以值的形式捕获所有外部变量
- [&] // 以引用形式捕获所有外部变量
- [x, &y] // x 以传值形式捕获，y 以引用形式捕获
- [=, &z] // z 以引用形式捕获，其余变量以传值形式捕获
- [&, x] // x 以值的形式捕获，其余变量以引用形式捕获

另外有一点需要注意。对于 [=] 或 [&] 的形式，lambda 表达式可以直接使用 this 指针。但是，对于 [] 的形式，如果要使用 this 指针，必须显式传入：

```C++
[this]() { this->someFunc(); }();
```

至此，我们已经大致了解了 C++ 11 提供的 lambda 表达式的概念。建议通过结合 lambda 表达式与 std::sort() 或 std::for_each() 这样的标准函数来尝试使用一下吧！

« 上一篇 下一篇 »

C++11 新特性：Lambda 表达式

豆子 | 2012年5月15日 | C++ | 6条评论

参考文章：https://blogs.oracle.com/pcarlini/entry/c_1x_tidbits_lambda_expressions

或许，Lambda 表达式算得上是 C++ 11 新增特性中最激动人心的一个。这个全新的特性听起来很深奥，但却是很多其他语言早已提供（比如 C#）或者即将提供（比如 Java）的。简而言之，Lambda 表达式就是用于创建匿名函数的。GCC 4.5.x 和 Microsoft Visual Studio 早已提供了对 lambda 表达式的支持。在 GCC 4.7 中，默认是不开启 C++ 11 特性的，需要添加 `-std=c++11` 编译参数。而 VS2010 则默认开启。

为什么说 lambda 表达式如此激动人心呢？举一个例子。标准 C++ 库中有一个常用算法的库，其中提供了很多算法函数，比如 sort() 和 find()。这些函数通常需要提供一个“谓词函数 predicate function”。所谓谓词函数，就是进行一个操作用的临时函数。比如 find() 需要一个谓词，用于查找元素满足的条件；能够满足谓词函数的元素才会被查找出来。这样的谓词函数，使用临时的匿名函数，既可以减少函数数量，又会让代码变得清晰易读。

下面来看一个例子：

C++

```
#include <algorithm>
#include <cmath>

void abssort(float *x, unsigned N)
{
  std::sort(x,
            x + N,
            [](float a, float b) { return std::abs(a) < std::abs(b); });
}
```

从上面的例子来看，尽管支持 lambda 表达式，但 C++ 的语法看起来却很“神奇”。lambda 表达式使用一对方括号作为开始的标识，类似于声明一个函数，只不过这个函数没有名字，也就是一个匿名函数。这个匿名函数接受两个参数， a 和 b ；其返回值是一个 bool 类型的值，注意，返回值是自动推断的，不需要显式声明，不过这是有条件的！条件就是，lambda 表达式的语句只有一个 return。函数的作用是比较 a、b 的绝对值的大小。然后，在此例中，这个 lambda 表达式作为一个闭包被传递给 `std::sort()` 函数。

下面，我们来详细解释下这个神奇的语法到底代表着什么。

我们从另外一个例子开始：

C++

```
std::cout << [](float f) { return std::abs(f); } (-3.5);
```

输出值是什么？3.5！注意，这是一个函数对象（由 lambda 表达式生成），其实参是 -3.5，返回值是参数的绝对值。lambda 表达式的返回值类型是语言自动推断的，因为 std::abs() 的返回值就是 float。注意，前

面我们也提到了，只有当 lambda 表达式中的语句“足够简单”，才能自动推断返回值类型。

C++ 11 的这种语法，其实就是匿名函数声明之后马上调用（否则的话，如果这个匿名函数既不调用，又不作为闭包传递给其它函数，那么这个匿名函数就没有什么用处）。如果你觉得奇怪，那么来看看 JavaScript 的这种写法：

JavaScript

```
function() {} ();

function(a) {} (-3.5);
```

C++ 11 的写法完全类似 JavaScript 的语法。

如果我不想让 lambda 表达式自动推断类型，或者是 lambda 表达式的内容很复杂，不能自动推断怎么办？比如， std::abs(float) 的返回值是 float，我想把它强制转型为 int。那么，此时，我们就必须显式指定 lambda 表达式返回值的类型：

C++

```
std::cout << [](float f) -> int { return std::abs(f); } (-3.5);
```

这个语句与前面的不同之处在于，lambda 表达式的返回时不是 float 而是 int。也就是说，上面语句的输出值是 3。返回值类型的概念同普通的函数返回值类型是完全一样的。

引入 lambda 表达式的前导符是一对方括号，称为 lambda 引入符（lambda-introducer）。lambda 引入符是有其自己的作用的，不仅仅是表明一个 lambda 表达式的开始那么简单。lambda 表达式可以使用与其相同范围 scope 内的变量。这个引入符的作用就是表明，其后的 lambda 表达式以何种方式使用（正式的术语是“捕获”）这些变量（这些变量能够在 lambda 表达式中被捕获，其实就是构成了一个闭包）。目前为止，我们看到的仅仅是一个空的方括号，其实，这个引入符是相当灵活的。例如：

C++

```
float f0 = 1.0;
std::cout << [=](float f) { return f0 + std::abs(f); } (-3.5);
```

其输出值是 4.5。[=] 意味着，lambda 表达式以传值的形式捕获同范围内的变量。另外一个例子：

C++

```
float f0 = 1.0;
std::cout << [&](float f) { return f0 += std::abs(f); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

输出值是 4.5 和 4.5。[&] 表明，lambda 表达式以传引用的方式捕获外部变量。那么，下一个例子：

C++

```
float f0 = 1.0;
std::cout << [=](float f) mutable { return f0 += std::abs(f); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

这个例子很有趣。首先，[=] 意味着，lambda 表达式以传值的形式捕获外部变量。C++ 11 标准说，如果以传值的形式捕获外部变量，那么，lambda 体不允许修改外部变量，对 f0 的任何修改都会引发编译错误。但是，注意，我们在 lambda 表达式前声明了 mutable 关键字，这就允许了 lambda 表达式体修改 f0 的值。因此，我们的例子本应报错，但是由于有 mutable 关键字，则不会报错。那么，你会觉得输出值是什么呢？答案是，4.5 和 1.0。为什么 f0 还是 1.0？因为我们是传值的，虽然在 lambda 表达式中对 f0 有了修改，但由于是传值的，外部的 f0 依然不会被修改。

上面的例子是，所有的变量要么传值，要么传引用。那么，是不是有混合机制呢？当然也有！比如下面的例子：

C++

```
float f0 = 1.0f;
float f1 = 10.0f;
std::cout << [=, &f0](float a) { return f0 += f1 + std::abs(a); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

这个例子的输出是 14.5 和 14.5。在这个例子中，f0 通过引用被捕获，而其它变量，比如 f1 则是通过值被捕获。

下面我们来总结下所有出现的 lambda 引入符：

- [] // 不捕获任何外部变量
- [=] // 以值的形式捕获所有外部变量
- [&] // 以引用形式捕获所有外部变量
- [x, &y] // x 以传值形式捕获，y 以引用形式捕获
- [=, &z] // z 以引用形式捕获，其余变量以传值形式捕获
- [&, x] // x 以值的形式捕获，其余变量以引用形式捕获

另外有一点需要注意。对于 [=] 或 [&] 的形式，lambda 表达式可以直接使用 this 指针。但是，对于 [] 的形式，如果要使用 this 指针，必须显式传入：

C++

```
[this]() { this->someFunc(); }();
```

至此，我们已经大致了解了 C++ 11 提供的 lambda 表达式的概念。建议通过结合 lambda 表达式与 std::sort() 或 std::for_each() 这样的标准函数来尝试使用一下吧！

« 上一篇 下一篇 »

C++11 新特性：Lambda 表达式

豆子 | 2012年5月15日 | C++ | 6条评论

参考文章：https://blogs.oracle.com/pcarlini/entry/c_1x_tidbits_lambda_expressions

或许，Lambda 表达式算得上是 C++ 11 新增特性中最激动人心的一个。这个全新的特性听起来很深奥，但却是很多其他语言早已提供（比如 C#）或者即将提供（比如 Java）的。简而言之，Lambda 表达式就是用于创建匿名函数的。GCC 4.5.x 和 Microsoft Visual Studio 早已提供了对 lambda 表达式的支持。在 GCC 4.7 中，默认是不开启 C++ 11 特性的，需要添加 `-std=c++11` 编译参数。而 VS2010 则默认开启。

为什么说 lambda 表达式如此激动人心呢？举一个例子。标准 C++ 库中有一个常用算法的库，其中提供了很多算法函数，比如 sort() 和 find()。这些函数通常需要提供一个“谓词函数 predicate function”。所谓谓词函数，就是进行一个操作用的临时函数。比如 find() 需要一个谓词，用于查找元素满足的条件；能够满足谓词函数的元素才会被查找出来。这样的谓词函数，使用临时的匿名函数，既可以减少函数数量，又会让代码变得清晰易读。

下面来看一个例子：

C++

```
#include <algorithm>
#include <cmath>

void abssort(float *x, unsigned N)
{
  std::sort(x,
            x + N,
            [](float a, float b) { return std::abs(a) < std::abs(b); });
}
```

从上面的例子来看，尽管支持 lambda 表达式，但 C++ 的语法看起来却很“神奇”。lambda 表达式使用一对方括号作为开始的标识，类似于声明一个函数，只不过这个函数没有名字，也就是一个匿名函数。这个匿名函数接受两个参数， `a` 和 `b` ；其返回值是一个 bool 类型的值，注意，返回值是自动推断的，不需要显式声明，不过这是有条件的！条件就是，lambda 表达式的语句只有一个 return。函数的作用是比较 a、b 的绝对值的大小。然后，在此例中，这个 lambda 表达式作为一个闭包被传递给 `std::sort()` 函数。

下面，我们来详细解释下这个神奇的语法到底代表着什么。

我们从另外一个例子开始：

C++

```
std::cout << [](float f) { return std::abs(f); } (-3.5);
```

输出值是什么？3.5！注意，这是一个函数对象（由 lambda 表达式生成），其实参是 -3.5，返回值是参数的绝对值。lambda 表达式的返回值类型是语言自动推断的，因为 std::abs() 的返回值就是 float。注意，前面我们也提到了，只有当 lambda 表达式中的语句“足够简单”，才能自动推断返回值类型。

C++ 11 的这种语法，其实就是匿名函数声明之后马上调用（否则的话，如果这个匿名函数既不调用，又不作为闭包传递给其它函数，那么这个匿名函数就没有什么用处）。如果你觉得奇怪，那么来看看 JavaScript 的这种写法：

```JavaScript
function() {} ();

function(a) {} (-3.5);
```

C++ 11 的写法完全类似 JavaScript 的语法。

如果我不想让 lambda 表达式自动推断类型，或者是 lambda 表达式的内容很复杂，不能自动推断怎么办？比如， std::abs(float) 的返回值是 float，我想把它强制转型为 int。那么，此时，我们就必须显式指定 lambda 表达式返回值的类型：

```C++
std::cout << [](float f) -> int { return std::abs(f); } (-3.5);
```

这个语句与前面的不同之处在于，lambda 表达式的返回时不是 float 而是 int。也就是说，上面语句的输出值是 3。返回值类型的概念同普通的函数返回值类型是完全一样的。

引入 lambda 表达式的前导符是一对方括号，称为 lambda 引入符（lambda-introducer）。lambda 引入符是有其自己的作用的，不仅仅是表明一个 lambda 表达式的开始那么简单。lambda 表达式可以使用与其相同范围 scope 内的变量。这个引入符的作用就是表明，其后的 lambda 表达式以何种方式使用（正式的术语是“捕获”）这些变量（这些变量能够在 lambda 表达式中被捕获，其实就是构成了一个闭包）。目前为止，我们看到的仅仅是一个空的方括号，其实，这个引入符是相当灵活的。例如：

```C++
float f0 = 1.0;
std::cout << [=](float f) { return f0 + std::abs(f); } (-3.5);
```

其输出值是 4.5。[=] 意味着，lambda 表达式以传值的形式捕获同范围内的变量。另外一个例子：

```C++
float f0 = 1.0;
std::cout << [&](float f) { return f0 += std::abs(f); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

输出值是 4.5 和 4.5。[&] 表明，lambda 表达式以传引用的方式捕获外部变量。那么，下一个例子：

```C++
float f0 = 1.0;
std::cout << [=](float f) mutable { return f0 += std::abs(f); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

这个例子很有趣。首先， [=] 意味着，lambda 表达式以传值的形式捕获外部变量。C++ 11 标准说，如果以传值的形式捕获外部变量，那么，lambda 体不允许修改外部变量，对 f0 的任何修改都会引发编译错误。但是，注意，我们在 lambda 表达式前声明了 mutable 关键字，这就允许了 lambda 表达式体修改 f0 的值。因此，我们的例子本应报错，但是由于有 mutable 关键字，则不会报错。那么，你会觉得输出值是什么呢？答案是，4.5 和 1.0。为什么 f0 还是 1.0？因为我们是传值的，虽然在 lambda 表达式中对 f0 有了修改，

但由于是传值的，外部的 f0 依然不会被修改。

上面的例子是，所有的变量要么传值，要么传引用。那么，是不是有混合机制呢？当然也有！比如下面的例子：

```C++
float f0 = 1.0f;
float f1 = 10.0f;
std::cout << [=, &f0](float a) { return f0 += f1 + std::abs(a); } (-3.5);
std::cout << '\n' << f0 << '\n';
```

这个例子的输出是 14.5 和 14.5。在这个例子中，f0 通过引用被捕获，而其它变量，比如 f1 则是通过值被捕获。

下面我们来总结下所有出现的 lambda 引入符：

- [] // 不捕获任何外部变量
- [=] // 以值的形式捕获所有外部变量
- [&] // 以引用形式捕获所有外部变量
- [x, &y] // x 以传值形式捕获，y 以引用形式捕获
- [=, &z] // z 以引用形式捕获，其余变量以传值形式捕获
- [&, x] // x 以值的形式捕获，其余变量以引用形式捕获

另外有一点需要注意。对于 [=] 或 [&] 的形式，lambda 表达式可以直接使用 this 指针。但是，对于 [] 的形式，如果要使用 this 指针，必须显式传入：

```C++
[this]() { this->someFunc(); }();
```

至此，我们已经大致了解了 C++ 11 提供的 lambda 表达式的概念。建议通过结合 lambda 表达式与 std::sort() 或 std::for_each() 这样的标准函数来尝试使用一下吧！

« 上一篇 下一篇 »