A.5 Lambda函数

lambda函数在 C++ 11中的加入很是令人兴奋，因为lambda函数能够大大简化代码复杂度(语法糖：利于理解具体的功能)，避免实现调用对象。 C++ 11的lambda函数语法允许在需要使用的时候进行定义。能为等待函数，例如 std::condition_variable (如同4.1.1节中的例子)提供很好谓词函数，其语义可以用来快速的表示可访问的变量，而非使用类中函数来对成员变量进行捕获。

最简单的情况下，lambda表达式就一个自给自足的函数，不需要传入函数仅依赖管局变量和函数，甚至都可以不用返回一个值。这样的lambda表达式的一系列语义都需要封闭在括号中，还要以方括号作为前缀：

```
[]{  // lambda表达式以[]开始
  do_stuff();
  do_more_stuff();
}();  // 表达式结束，可以直接调用
```

例子中，lambda表达式通过后面的括号调用，不过这种方式不常用。一方面，如果想要直接调用，可以在写完对应的语句后，就对函数进行调用。对于函数模板，传递一个参数进去时很常见的事情，甚至可以将可调用对象作为其参数传入；可调用对象通常也需要一些参数，或返回一个值，亦或两者都有。如果想给lambda函数传递参数，可以参考下面的lambda函数，其使用起来就像是一个普通函数。例如，下面代码是将vector中的元素使用 std::cout 进行打印：

```
std::vector<int> data=make_data();
std::for_each(data.begin(),data.end(),[](int i){std::cout<<i<<"\n";});
```

返回值也是很简单的，当lambda函数体包括一个return语句，返回值的类型就作为lambda表达式的返回类型。例如，使用一个简单的lambda函数来等待 std::condition_variable (见4.1.1节)中的标志被设置。

清单A.4 lambda函数推导返回类型

```
std::condition_variable cond;
bool data_ready;
std::mutex m;
void wait_for_data()
```

```
{
  std::unique_lock<std::mutex> lk(m);
  cond.wait(lk,[]{return data_ready;});  // 1
}
```

lambda的返回值传递给cond.wait()①，函数就能推断出data_ready的类型是bool。当条件变量从等待中苏醒后，上锁阶段会调用lambda函数，并且当data_ready为true时，仅返回到wait()中。

当lambda函数体中有多个return语句，就需要显式的指定返回类型。只有一个返回语句的时候，也可以这样做，不过这样可能会让你的lambda函数体看起来更复杂。返回类型可以使用跟在参数列表后面的箭头(->)进行设置。如果lambda函数没有任何参数，还需要包含(空)的参数列表，这样做是为了能显式的对返回类型进行指定。对条件变量的预测可以写成下面这种方式：

```
cond.wait(lk,[]()->bool{return data_ready;});
```

还可以对lambda函数进行扩展，比如：加上log信息的打印，或做更加复杂的操作：

```
cond.wait(lk,[]()->bool{
  if(data_ready)
  {
    std::cout<<”Data ready”<<std::endl;
    return true;
  }
  else
  {
    std::cout<<”Data not ready, resuming wait”<<std::endl;
    return false;
  }
});
```

虽然简单的lambda函数很强大，能简化代码，不过其真正的强大的地方在于对本地变量的捕获。

A.5.1 引用本地变量的Lambda函数

lambda函数使用空的 `[]` (lambda introducer)就不能引用当前范围内的本地变量；其只能使用全局变量，或将其他值以参数的形式进行传递。当想要访问一个本地变量，需要对其进行捕获。最简单的方式就是将范围内的所有本地变量都进行捕获，使用 `[=]` 就可以完成这样的功能。函数被创建的时候，就能对本地变量的副本进行访问了。

实践一下，看一下下面的例子：

```
std::function<int(int)> make_offseter(int offset)
{
  return [=](int j){return offset+j;};
}
```

当调用make_offseter时，就会通过 `std::function<>` 函数包装返回一个新的lambda函数体。

这个带有返回的函数添加了对参数的偏移功能。例如：

```
int main()
{
  std::function<int(int)> offset_42=make_offseter(42);
  std::function<int(int)> offset_123=make_offseter(123);
  std::cout<<offset_42(12)<<",“<<offset_123(12)<<std::endl;
  std::cout<<offset_42(12)<<",“<<offset_123(12)<<std::endl;
}
```

屏幕上将打印出54,135两次，因为第一次从make_offseter中返回，都是对参数加42的；第二次调用后，make_offseter会对参数加上123。所以，会打印两次相同的值。

这种本地变量捕获的方式相当安全，所有的东西都进行了拷贝，所以可以通过lambda函数对表达式的值进行返回，并且可在原始函数之外的地方对其进行调用。这也不是唯一的选择，也可以通过选择通过引用的方式捕获本地变量。在本地变量被销毁的时候，lambda函数会出现未定义的行为。

下面的例子，就介绍一下怎么使用 `[&]` 对所有本地变量进行引用：

```
int main()
{
  int offset=42;  // 1
  std::function<int(int)> offset_a=[&](int j){return offset+j;};  // 2
  offset=123;  // 3
  std::function<int(int)> offset_b=[&](int j){return offset+j;};  // 4
  std::cout<<offset_a(12)<<",”<<offset_b(12)<<std::endl;  // 5
  offset=99;  // 6
  std::cout<<offset_a(12)<<",”<<offset_b(12)<<std::endl;  // 7
}
```

之前的例子中，使用 [=] 来对要偏移的变量进行拷贝，offset_a函数就是个使用 [&] 捕获offset的引用的例子②。所以，offset初始化成42也没什么关系①；offset_a(12)的例子通常会依赖与当前offset的值。在③上，offset的值会变为123，offset_b④函数将会使用到这个值，同样第二个函数也是使用引用的方式。

现在，第一行打印信息⑤，offset为123，所以输出为135,135。不过，第二行打印信息⑦就有所不同，offset变成99⑥，所以输出为111,111。offset_a和offset_b都对当前值进行了加12的操作。

尘归尘，土归土， C++ 还是 C++ ；这些选项不会让你感觉到特别困惑，你可以选择以引用或拷贝的方式对变量进行捕获，并且你还可以通过调整中括号中的表达式，来对特定的变量进行显式捕获。如果想要拷贝所有变量，而非一两个，可以使用 [=] ，通过参考中括号中的符号，对变量进行捕获。下面的例子将会打印出1239，因为i是拷贝进lambda函数中的，而j和k是通过引用的方式进行捕获的：

```
int main()
{
  int i=1234,j=5678,k=9;
  std::function<int()> f=[=,&j,&k]{return i+j+k;};
  i=1;
  j=2;
  k=3;
  std::cout<<f()<<std::endl;
}
```

或者，也可以通过默认引用方式对一些变量做引用，而对一些特别的变量进行拷贝。这种情况下，就要使用 [&] 与拷贝符号相结合的方式对列表中的变量进行拷贝捕获。下面的例子将打印出5688，因为i通过引用捕获，但j和k 通过拷贝捕获：

```
int main()
{
  int i=1234,j=5678,k=9;
  std::function<int()> f=[&,j,k]{return i+j+k;};
  i=1;
  j=2;
  k=3;
  std::cout<<f()<<std::endl;
}
```

如果你只想捕获某些变量，那么你可以忽略=或&，仅使用变量名进行捕获就行；加上&前缀，是将对应变量以引用的方式进行捕获，而非拷贝的方式。下面的例子将打印出5682，因为i和k是通过

引用的范式获取的，而j是通过拷贝的方式：

```
int main()
{
  int i=1234,j=5678,k=9;
  std::function<int()> f=[&i,j,&k]{return i+j+k;};
  i=1;
  j=2;
  k=3;
  std::cout<<f()<<std::endl;
}
```

最后一种方式，是为了确保预期的变量能被捕获，在捕获列表中引用任何不存在的变量都会引起编译错误。当选择这种方式，就要小心类成员的访问方式，确定类中是否包含一个lambda函数的成员变量。类成员变量不能直接捕获，如果想通过lambda方式访问类中的成员，需要在捕获列表中添加this指针，以便捕获。下面的例子中，lambda捕获this后，就能访问到some_data类中的成员：

```
struct X
{
  int some_data;
  void foo(std::vector<int>& vec)
  {
    std::for_each(vec.begin(),vec.end(),
         [this](int& i){i+=some_data;});
  }
};
```

并发的上下文中，lambda是很有用的，其可以作为谓词放在 `std::condition_variable::wait()` (见4.1.1节)和 `std::packaged_task<>` (见4.2.1节)中；或是用在线程池中，对小任务进行打包。也可以线程函数的方式 `std::thread` 的构造函数(见2.1.1)，以及作为一个并行算法实现，在parallel_for_each()(见8.5.1节)中使用。