2.2 向线程函数传递参数

清单2.4中，向 `std::thread` 构造函数中的可调用对象，或函数传递一个参数很简单。需要注意的是，默认参数要拷贝到线程独立内存中，即使参数是引用的形式，也可以在新线程中进行访问。再来看一个例子：

```
void f(int i, std::string const& s);
std::thread t(f, 3, "hello");
```

代码创建了一个调用f(3, "hello")的线程。注意，函数f需要一个 `std::string` 对象作为第二个参数，但这里使用的是字符串的字面值，也就是 `char const *` 类型。之后，在线程的上下文中完成字面值向 `std::string` 对象的转化。需要特别要注意，当指向动态变量的指针作为参数传递给线程的情况，代码如下：

```
void f(int i,std::string const& s);
void oops(int some_param)
{
  char buffer[1024]; // 1
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3,buffer); // 2
  t.detach();
}
```

这种情况下，buffer②是一个指针变量，指向本地变量，然后本地变量通过buffer传递到新线程中②。并且，函数有很有可能会在字面值转化成 `std::string` 对象之前*崩溃*(oops)，从而导致一些未定义的行为。并且想要依赖隐式转换将字面值转换为函数期待的 `std::string` 对象，但因 `std::thread` 的构造函数会复制提供的变量，就只复制了没有转换成期望类型的字符串字面值。

解决方案就是在传递到 `std::thread` 构造函数之前就将字面值转化为 `std::string` 对象：

```
void f(int i,std::string const& s);
void not_oops(int some_param)
{
  char buffer[1024];
  sprintf(buffer,"%i",some_param);
```

```
  std::thread t(f,3,std::string(buffer));  // 使用std::string，避免悬垂指针
  t.detach();
}
```

还可能遇到相反的情况：期望传递一个引用，但整个对象被复制了。当线程更新一个引用传递的数据结构时，这种情况就可能发生，比如：

```
void update_data_for_widget(widget_id w,widget_data& data); // 1
void oops_again(widget_id w)
{
  widget_data data;
  std::thread t(update_data_for_widget,w,data); // 2
  display_status();
  t.join();
  process_widget_data(data); // 3
}
```

虽然update_data_for_widget①的第二个参数期待传入一个引用，但是 `std::thread` 的构造函数②并不知晓；构造函数无视函数期待的参数类型，并盲目的拷贝已提供的变量。当线程调用update_data_for_widget函数时，传递给函数的参数是data变量内部拷贝的引用，而非数据本身的引用。因此，当线程结束时，内部拷贝数据将会在数据更新阶段被销毁，且process_widget_data将会接收到没有修改的data变量③。对于熟悉 `std::bind` 的开发者来说，问题的解决办法是显而易见的：可以使用 `std::ref` 将参数转换成引用的形式，从而可将线程的调用改为以下形式：

```
std::thread t(update_data_for_widget,w,std::ref(data));
```

在这之后，update_data_for_widget就会接收到一个data变量的引用，而非一个data变量拷贝的引用。

如果你熟悉 `std::bind` ，就应该不会对以上述传参的形式感到奇怪，因为 `std::thread` 构造函数和 `std::bind` 的操作都在标准库中定义好了，可以传递一个成员函数指针作为线程函数，并提供一个合适的对象指针作为第一个参数：

```
class X
{
public:
  void do_lengthy_work();
};
X my_x;
std::thread t(&X::do_lengthy_work,&my_x); // 1
```

这段代码中，新线程将my_x.do_lengthy_work()作为线程函数；my_x的地址①作为指针对象提供给函数。也可以为成员函数提供参数： `std::thread` 构造函数的第三个参数就是成员函数的第一个参数，以此类推(代码如下，译者自加)。

```
class X
{
public:
  void do_lengthy_work(int);
};
X my_x;
int num(0);
std::thread t(&X::do_lengthy_work, &my_x, num);
```

有趣的是，提供的参数可以*移动*，但不能*拷贝*。"移动"是指:原始对象中的数据转移给另一对象，而转移的这些数据就不再在原始对象中保存了(译者：比较像在文本编辑的"剪切"操作)。 `std::unique_ptr` 就是这样一种类型(译者：C++11中的智能指针)，这种类型为动态分配的对象提供内存自动管理机制(译者：类似垃圾回收)。同一时间内，只允许一个 `std::unique_ptr` 实现指向一个给定对象，并且当这个实现销毁时，指向的对象也将被删除。*移动构造函数*(move constructor)和*移动赋值操作符*(move assignment operator)允许一个对象在多个 `std::unique_ptr` 实现中传递(有关"移动"的更多内容，请参考附录A的A.1.1节)。使用"移动"转移原对象后，就会留下一个*空指针*(NULL)。移动操作可以将对象转换成可接受的类型，例如:函数参数或函数返回的类型。当原对象是一个临时变量时，自动进行移动操作，但当原对象是一个命名变量，那么转移的时候就需要使用 `std::move()` 进行显示移动。下面的代码展示了 `std::move` 的用法，展示了 `std::move` 是如何转移一个动态对象到一个线程中去的：

```
void process_big_object(std::unique_ptr<big_object>);

std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
std::thread t(process_big_object,std::move(p));
```

在 `std::thread` 的构造函数中指定 `std::move(p)` ,big_object对象的所有权就被首先转移到新创建线程的的内部存储中，之后传递给process_big_object函数。

标准线程库中和 `std::unique_ptr` 在所属权上有相似语义类型的类有好几种， `std::thread` 为其中之一。虽然， `std::thread` 实例不像 `std::unique_ptr` 那样能占有一个动态对象的所有权，但是它能占有其他资源：每个实例都负责管理一个执行线程。执行线程的所有权可以在多个 `std::thread` 实例中互相转移，这是依赖于 `std::thread` 实例的*可移动*且*不可复制*性。不可复

制保性证了在同一时间点，一个 `std::thread` 实例只能关联一个执行线程；可移动性使得程序员可以自己决定，哪个实例拥有实际执行线程的所有权。