

## 2.1 线程管理的基础

每个程序至少有一个线程：执行`main()`函数的线程，其余线程有其各自的入口函数。线程与原始线程(以`main()`为入口函数的线程)同时运行。如同`main()`函数执行完会退出一样，当线程执行完入口函数后，线程也会退出。在为一个线程创建了一个 `std::thread` 对象后，需要等待这个线程结束；不过，线程需要先进行启动。下面就来启动线程。

### 2.1.1 启动线程

第1章中，线程在 `std::thread` 对象创建(为线程指定任务)时启动。最简单的情况下，任务也会很简单，通常是无参数无返回的函数。这种函数在其所属线程上运行，直到函数执行完毕，线程也就结束了。在一些极端情况下，线程运行时，任务中的函数对象需要通过某种通讯机制进行参数的传递，或者执行一系列独立操作；**可以通过通讯机制传递信号，让线程停止**。线程要做什么，以及什么时候启动，其实都无关紧要。总之，使用C++线程库启动线程，可以归结为构造 `std::thread` 对象：

```
1 void do_some_work();
2 std::thread my_thread(do_some_work);
```

为了让编译器识别 `std::thread` 类，这个简单的例子也要包含 `<thread>` 头文件。如同大多数C++标准库一样，`std::thread` 可以用可调用类型构造，将带有函数调用符类型的实例传入 `std::thread` 类中，替换默认的构造函数。

```
1 class background_task
2 {
3 public:
4     void operator()() const
5     {
6         do_something();
7         do_something_else();
8     }
9 };
10
```

```
11 background_task f;  
12 std::thread my_thread(f);
```

代码中，提供的函数对象会复制到新线程的存储空间当中，函数对象的执行和调用都在线程的内存空间中进行。函数对象的副本应与原始函数对象保持一致，否则得到的结果会与我们的期望不同。

有件事需要注意，当把函数对象传入到[线程构造函数](#)中时，需要避免“[最令人头痛的语法解析](#)”(C++'s most vexing parse, [中文简介](#))。如果你传递了一个临时变量，而不是一个命名的变量；C++编译器会将其解析为函数声明，而不是类型对象的定义。

例如：

```
std::thread my_thread(background_task());
```

这里相当与声明了一个名为my\_thread的函数，这个函数带有一个参数(函数指针指向没有参数并返回background\_task对象的函数)，返回一个 std::thread 对象的函数，而非启动了一个线程。

使用在前面命名函数对象的方式，或使用多组括号①，或使用新统一的初始化语法②，可以避免这个问题。

如下所示：

```
1 std::thread my_thread((background_task())); // 1  
2 std::thread my_thread{background_task()}; // 2
```

使用lambda表达式也能避免这个问题。lambda表达式是C++11的一个新特性，它允许使用一个可以捕获局部变量的局部函数(可以避免传递参数，参见2.2节)。想要具体的了解lambda表达式，可以阅读附录A的A.5节。之前的例子可以改写为lambda表达式的类型：

```
1 std::thread my_thread([]{  
2     do_something();  
3     do_something_else();  
4 });
```

启动了线程，你需要明确是要等待线程结束([加入式](#)——参见2.1.2节)，还是让其自主运行([分离式](#)——参见2.1.3节)。如果 std::thread 对象销毁之前还没有做出决定，程序就会终止(std::thread 的析构函数会调用 std::terminate() )。因此，即便是有异常存在，也需要确保

线程能够正确的**加入(joined)**或**分离(detached)**。2.1.3节中，会介绍对应的方法来处理这两种情况。需要注意的是，必须在 `std::thread` 对象销毁之前做出决定，否则你的程序将会终止(`std::thread`的析构函数会调用`std::terminate()`，这时再去决定会触发相应异常)。

如果不等待线程，就必须保证线程结束之前，可访问的数据得有效性。这不是一个新问题——单线程代码中，对象销毁之后再去访问，也会产生未定义行为——不过，线程的生命周期增加了这个问题发生的几率。

这种情况很可能发生在**线程还没结束，函数已经退出**的时候，这时线程函数还持有**函数局部变量的指针或引用**。下面的清单中就展示了这样的一种情况。

清单2.1 函数已经结束，线程依旧访问局部变量


```

1  struct func
2  {
3      int& i;
4      func(int& i_) : i(i_) {}
5      void operator() ()
6      {
7          for (unsigned j=0 ; j<1000000 ; ++j)
8          {
9              do_something(i);          // 1. 潜在访问隐患：悬空引用
10         }
11     }
12 };
13
14 void oops()
15 {
16     int some_local_state=0;
17     func my_func(some_local_state);
18     std::thread my_thread(my_func);
19     my_thread.detach();                // 2. 不等待线程结束
20 }                                     // 3. 新线程可能还在运行

```


这个例子中，已经决定不等待线程结束(使用了`detach()`②)，所以当`oops()`函数执行完成时③，新线程中的函数可能还在运行。如果线程还在运行，它就会去调用`do_something(i)`函数①，这时就会访问已经销毁的变量。如同一个单线程程序——**允许在函数完成后继续持有局部变量的指针或引用**；当然，这从来就不是一个好主意——这种情况发生时，**错误并不明显，会使多线程更容易出错。**


处理这种情况的常规方法：使线程函数的功能齐全，将数据复制到线程中，而非复制到共享数据中。如果使用一个可调用的对象作为线程函数，这个对象就会复制到线程中，而后原始对象就会

立即销毁。但对于对象中包含的指针和引用还需谨慎，例如清单2.1所示。使用一个能访问局部变量的函数去创建线程是一个糟糕的主意(除非十分确定线程会在函数完成前结束)。此外，可以通过`join()`函数来确保线程在函数完成前结束。


## 2.1.2 等待线程完成

如果需要等待线程，相关的 `std::thread` 实例需要使用`join()`。清单2.1中，将

`my_thread.detach()` 替换为 `my_thread.join()`，就可以确保局部变量在线程完成后，才被销毁。在这种情况下，因为原始线程在其生命周期中并没有做什么事，使得用一个独立的线程去执行函数变得收益甚微，但在实际编程中，**原始线程**要么有自己的工作要做；要么会启动多个子线程来做一些有用的工作，并等待这些线程结束。

`join()`是简单粗暴的等待线程完成或不等待。当你需要对等待中的线程有更灵活的控制时，比如，**看一下某个线程是否结束**，或者**只等待一段时间(超过时间就判定为超时)**。想要做到这些，你需要使用其他机制来完成，比如条件变量和**期待(futures)**，相关的讨论将会在第4章继续。调用`join()`的行为，还清理了线程相关的存储部分，这样 `std::thread` 对象将不再与已经完成的线程有任何关联。这意味着，**只能对一个线程使用一次`join()`**；一旦已经使用过`join()`，`std::thread` **对象就不能再次加入了**，对其使用`joinable()`时，将返回`false`。

## 2.1.3 特殊情况下的等待

如前所述，需要对一个还未销毁的 `std::thread` 对象使用`join()`或`detach()`。如果想要分离一个线程，可以在线程启动后，直接使用`detach()`进行分离。如果打算等待对应线程，则需要细心挑选调用`join()`的位置。当在线程运行之后产生异常，在`join()`调用之前抛出，就意味着**这次调用会被跳过**。

避免应用被抛出的异常所终止，就需要作出一个决定。通常，当倾向于在无异常的情况下使用`join()`时，需要在异常处理过程中调用`join()`，从而避免生命周期的问题。下面的程序清单是一个例子。

清单 2.2 等待线程完成

```
1 struct func; // 定义在清单2.1中
2 void f()
3 {
```

```
4   int some_local_state=0;
5   func my_func(some_local_state);
6   std::thread t(my_func);
7   try
8   {
9       do_something_in_current_thread();
10  }
11  catch(...)
12  {
13      t.join(); // 1
14      throw;
15  }
16  t.join(); // 2
17 }
```

清单2.2中的代码使用了 `try/catch` 块确保访问本地状态的线程退出后，函数才结束。当函数正常退出时，会执行到②处；当函数执行过程中抛出异常，程序会执行到①处。`try/catch` 块能轻易的捕获轻量级错误，所以这种情况，并非放之四海而皆准。如需确保线程在函数之前结束——查看是否因为线程函数使用了局部变量的引用，以及其他原因——而后再确定一下程序可能会退出的途径，无论正常与否，可以提供一個简洁的机制，来做解决这个问题。

一种方式是使用“资源获取即初始化方式”(RAII, Resource Acquisition Is Initialization)，并且提供一个类，在析构函数中使用`join()`，如同下面清单中的代码。看它如何简化`f()`函数。

### 清单 2.3 使用RAII等待线程完成

```
1  class thread_guard
2  {
3      std::thread& t;
4  public:
5      explicit thread_guard(std::thread& t_):
6          t(t_)
7      {}
8      ~thread_guard()
9      {
10         if(t.joinable()) // 1
11         {
12             t.join();      // 2
13         }
14     }
15     thread_guard(thread_guard const&)=delete; // 3
16     thread_guard& operator=(thread_guard const&)=delete;
17 };
```

```
18
19 struct func; // 定义在清单2.1中
20
21 void f()
22 {
23     int some_local_state=0;
24     func my_func(some_local_state);
25     std::thread t(my_func);
26     thread_guard g(t);
27     do_something_in_current_thread();
28 } // 4
```

当线程执行到④处时，**局部对象就要被逆序销毁了**。因此，`thread_guard`对象`g`是第一个被销毁的，这时线程在析构函数中被加入②到原始线程中。即使`do_something_in_current_thread`抛出一个异常，这个销毁依旧会发生。

在`thread_guard`的析构函数的测试中，首先判断线程是否已加入①，如果没有会调用`join()`②进行加入。这很重要，因为`join()`只能对给定的对象调用一次，所以对给已加入的线程再次进行加入操作时，将会导致错误。

拷贝构造函数和拷贝赋值操作被标记为 `=delete` ③，是**为了不让编译器自动生成它们**。直接对一个对象进行拷贝或赋值是危险的，因为这可能会**弄丢已经加入的线程**。通过删除声明，任何尝试给`thread_guard`对象赋值的操作都会引发一个编译错误。想要了解删除函数的更多知识，请参阅附录A的A.2节。

如果不想等待线程结束，可以分离(*detaching*)线程，从而避免异常安全\*(*exception-safety*)问题。不过，这就打破了线程与 `std::thread` 对象的联系，即使线程仍然在后台运行着，分离操作也能确保 `std::terminate()` 在 `std::thread` 对象销毁才被调用。

## 2.1.4 后台运行线程

使用`detach()`会让线程在后台运行，这就意味着主线程不能与之产生直接交互。也就是说，不会等待这个线程结束；如果线程分离，那么就不可能有 `std::thread` 对象能引用它，分离线程的确在后台运行，所以分离线程不能被加入。不过C++运行库保证，当线程退出时，相关资源的能够正确回收，后台线程的归属和控制C++运行库都会处理。

通常称分离线程为**守护线程(daemon threads)**，UNIX中守护线程是指，没有任何显式的用户接口，并在后台运行的线程。这种线程的特点就是长时间运行；线程的生命周期可能会从某一个应用起始到结束，可能会在后台监视文件系统，还有可能对缓存进行清理，亦或对数据结构进行优化。



另一方面，分离线程的另一方面只能确定线程什么时候结束，*发后即忘*(fire and forget)的任务就使用到线程的这种方式。

如2.1.2节所示，调用 `std::thread` 成员函数`detach()`来分离一个线程。之后，相应的 `std::thread` 对象就与实际执行的线程无关了，并且这个线程也无法加入：

```
1  std::thread t(do_background_work);
2  t.detach();
3  assert(!t.joinable());
```

为了从 `std::thread` 对象中分离线程(前提是有可进行分离的线程),不能对没有执行线程的 `std::thread` 对象使用`detach()`,也是`join()`的使用条件，并且要用同样的方式进行检查——当 `std::thread` 对象使用`t.joinable()`返回的是`true`，就可以使用`t.detach()`。

试想如何能让一个文字处理应用同时编辑多个文档。无论是用户界面，还是在内部应用内部进行，都有很多的解决方法。虽然，这些窗口看起来是完全独立的，每个窗口都有自己独立的菜单选项，但他们却运行在同一个应用实例中。一种内部处理方式是，让每个文档处理窗口拥有自己的线程；每个线程运行同样的代码，并隔离不同窗口处理的数据。如此这般，打开一个文档就要启动一个新线程。因为是对独立的文档进行操作，所以没有必要等待其他线程完成。因此，这里就可以让文档处理窗口运行在分离的线程上。

下面代码简要的展示了这种方法：

清单2.4 使用分离线程去处理其他文档

```
1  void edit_document(std::string const& filename)
2  {
3      open_document_and_display_gui(filename);
4      while(!done_editing())
5      {
6          user_command cmd=get_user_input();
7          if(cmd.type==open_new_document)
8          {
9              std::string const new_name=get_filename_from_user();
10             std::thread t(edit_document,new_name);  // 1
11             t.detach();  // 2
12         }
13         else
14         {
15             process_user_input(cmd);
16         }
```

```
17     }  
18 }
```

如果用户选择打开一个新文档，需要启动一个新线程去打开新文档①，并分离线程②。与当前线程做出的操作一样，新线程只不过是打开另一个文件而已。所以，`edit_document`函数可以复用，通过传参的形式打开新的文件。

这个例子也展示了传参启动线程的方法：不仅可以向 `std::thread` 构造函数①传递函数名，还可以传递函数所需的参数(实参)。C++线程库的方式也不是很复杂。当然，也有其他方法完成这项功能，比如:使用一个带有数据成员的成员函数，代替一个需要传参的普通函数。