Item 37：Make `std::threads` unjoinable on all paths

每个`std::thread`对象处于两个状态之一：*joinable or unjoinable*。*joinable*状态的`std::thread`对应于正在运行或者可能正在运行的异步执行线程。比如，一个blocked或者等待调度的`std::thread`是*joinable*，已运行结束的`std::thread`也可以认为是*joinable*

*unjoinable*的`std::thread`对象比如：

- **Default-constructed std::threads**。这种`std::thread`没有函数执行，因此无法绑定到具体的线程上
- **已经被moved的`std::thread`对象**。move的结果就是将`std::thread`对应的线程所有权转移给另一个`std::thread`
- **已经joined的`std::thread`**。在join之后，`std::thread`执行结束，不再对应于具体的线程
- **已经detached的`std::thread`**。detach断开了`std::thread`与线程之间的连接

（译者注：`std::thread`可以视作状态保存的对象，保存的状态可能也包括可调用对象，有没有具体的线程承载就是有没有连接）

`std::thread`的可连接性如此重要的原因之一就是当连接状态的析构函数被调用，执行逻辑被终止。比如，假定有一个函数`doWork`，执行过滤函数`filter`，接收一个参数`maxVal`。`doWork`检查是否满足计算所需的条件，然后通过使用0到maxVal之间的所有值过滤计算。如果进行过滤非常耗时，并且确定doWork条件是否满足也很耗时，则将两件事并发计算是很合理的。

我们希望为此采用基于任务的设计（参与Item 35），但是假设我们希望设置做过滤线程的优先级。Item 35阐释了需要线程的基本句柄，只能通过`std::thread`的API来完成；基于任务的API（比如futures）做不到。所以最终采用基于`std::thread`而不是基于任务

代码如下：

```
constexpr auto tenMillion = 10000000; // see Item 15 for constexpr
bool doWork(std::function<bool(int)> filter, int maxVal = tenMillion) // return
whether computation was performed; see Item2 for std::function
{
  std::vector<int> goodVals;
  std::thread t([&filter, maxVal, &goodVals]
                {
                  for (auto i = 0; i <= maxVal; ++i)
                  {
                    if (filter(i)) goodVals.push_back(i);
                  }
                });
  auto nh = t.native_handle(); // use t's native handle to set t's priority
  ...
  if (conditionsAreStatisfied()) {
    t.join(); // let t finish
    performComputation(goodVals); // computation was performed
    return true;
  }

  return false; // computation was not performed
}
```

在解释这份代码为什么有问题之前，看一下tenMillion的初始化可以在C++14中更加易读，通过单引号分隔数字：

```
constexpr auto tenMillion = 10'000'000; // C++14
```

还要指出，在开始运行之后设置t的优先级就像把马放出去之后再关上马厩门一样（译者注：太晚了）。更好的设计是在t为挂起状态时设置优先级（这样可以在执行任何计算前调整优先级），但是我不想你为这份代码考虑这个而分心。如果你感兴趣代码中忽略的部分，可以转到Item 39，那个Item告诉你如何以挂起状态开始线程。

返回`doWork`。如果`conditionsAreSatisfied()`返回真，没什么问题，但是如果返回假或者抛出异常，`std::thread`类型的`t`在`doWork`结束时会调用`t`的析构器。这造成程序执行中止。

你可能会想，为什么`std::thread`析构的行为是这样的，那是因为另外两种显而易见的方式更糟：

- **隐式join**。这种情况下，`std::thread`的析构函数将等待其底层的异步执行线程完成。这听起来是合理的，但是可能会导致性能异常，而且难以追踪。比如，如果`conditonAreStatisfied()`已经返回了假，`doWork`继续等待过滤器应用于所有值就很违反直觉。
- **隐式detach**。这种情况下，`std::thread`析构函数会分离其底层的线程。线程继续运行。听起来比join的方式好，但是可能导致更严重的调试问题。比如，在`doWork`中，`goodVals`是通过引用捕获的局部变量。可能会被lambda修改。假定，lambda的执行时异步的，`conditionsAreStatisfied()`返回假。这时，`doWork`返回，同时局部变量`goodVals`被销毁。堆栈被弹出，并在`doWork`的调用点继续执行线程

  某个调用点之后的语句有时会进行其他函数调用，并且至少一个这样的调用可能会占用曾经被`doWork`使用的堆栈位置。我们称为`f`，当`f`运行时，`doWork`启动的lambda仍在继续运行。该lambda可以在堆栈内存中调用`push_back`，该内存曾是`goodVals`，位于`doWork`曾经的堆栈位置。这意味着对`f`来说，内存被修改了，想象一下调试的时候痛苦

标准委员会认为，销毁连接中的线程如此可怕以至于实际上禁止了它（通过指定销毁连接中的线程导致程序终止）

这使你有责任确保使用`std::thread`对象时，在所有的路径上最终都是unjoinable的。但是覆盖每条路径可能很复杂，可能包括`return, continue, break, goto or exception`，有太多可能的路径。

每当你想每条路径的块之外执行某种操作，最通用的方式就是将该操作放入本地对象的析构函数中。这些对象称为RAII对象，通过RAII类来实例化。（RAII全称为 Resource Acquisition Is Initialization）。RAII类在标准库中很常见。比如STL容器，智能指针，`std::fstream`类等。但是标准库没有RAII的`std::thread`类，可能是因为标准委员会拒绝将`join`和`detach`作为默认选项，不知道应该怎么样完成RAII。

幸运的是，完成自行实现的类并不难。比如，下面的类实现允许调用者指定析构函数`join`或者`detach`：

```
class ThreadRAII {
public:
  enum class DtorAction{ join, detach }; // see Item 10 for enum class info
  ThreadRAII(std::thread&& t, DtorAction a): action(a), t(std::move(t)) {} // in
dtor, take action a on t
  ~ThreadRAII()
  {
    if (t.joinable()) {
      if (action == DtorAction::join) {
        t.join();
      } else {
```

```
        t.detach();
      }
    }
  }
  std::thread& get() { return t; } // see below
private:
  DtorAction action;
  std::thread t;
};
```

我希望这段代码是不言自明的，但是下面几点说明可能会有所帮助：

- 构造器只接受`std::thread`右值，因为我们想要move`std::thread`对象给`ThreadRAII`（再次强调，`std::thread`不可以复制）
- 构造器的参数顺序设计的符合调用者直觉（首先传递`std::thread`，然后选择析构执行的动作），但是成员初始化列表设计的匹配成员声明的顺序。将`std::thread`成员放在声明最后。在这个类中，这个顺序没什么特别之处，调整为其他顺序也没有问题，但是通常，可能一个成员的初始化依赖于另一个，因为`std::thread`对象可能会在初始化结束后就立即执行了，所以在最后声明是一个好习惯。这样就能保证一旦构造结束，所有数据成员都初始化完毕可以安全的异步绑定线程执行
- `ThreadRAII`提供了`get`函数访问内部的`std::thread`对象。这类似于标准智能指针提供的`get`函数，可以提供访问原始指针的入口。提供`get`函数避免了`ThreadRAII`复制完整`std::thread`接口的需要，因为着`ThreadRAII`可以在需要`std::thread`上下文的环境中使用
- 在`ThreadRAII`析构函数调用`std::thread`对象t的成员函数之前，检查t是否joinable。这是必须的，因为在unjoinbale的`std::thread`上调用`join or detach`会导致未定义行为。客户端可能会构造一个`std::thread`t，然后通过t构造一个`ThreadRAII`，使用`get`获取t，然后移动t，或者调用`join or detach`，每一个操作都使得t变为unjoinable
  如果你担心下面这段代码

  ```
  if (t.joinable()) {
    if (action == DtorAction::join) {
      t.join();
    } else {
      t.detach();
    }
  }
  ```

  存在竞争，因为在`t.joinable()`和`t.join or t.detach`执行中间，可能有其他线程改变了t为unjoinable，你的态度很好，但是这个担心不必要。`std::thread`只有自己可以改变`joinable or unjoinable`的状态。在`ThreadRAII`的析构函数中被调用时，其他线程不可能做成员函数的调用。如果同时进行调用，那肯定是有竞争的，但是不在析构函数中，是在客户端代码中试图同时在一个对象上调用两个成员函数（析构函数和其他函数）。通常，仅当所有都为const成员函数时，在一个对象同时调用两个成员函数才是安全的。

在`doWork`的例子上使用`ThreadRAII`的代码如下：

```
bool doWork(std::function<bool(int)> filter, int maxVal = tenMillion)
{
  std::vector<int> goodVals;
  ThreadRAII t(std::thread([&filter, maxVal, &goodVals] {
      for (auto i = 0; i <= maxVal; ++i) {
        if (filter(i)) goodVals.push_back(i);
      }
```

```
    }),
    ThreadRAII::DtorAction::join
  );
  auto nh = t.get().native_handle();
  ...
  if (conditonsAreStatisfied()) {
    t.get().join();
    performComputation(goodVals);
    return true;
  }
  return false;
}
```

这份代码中，我们选择在`ThreadRAII`的析构函数中异步执行`join`的动作，因为我们先前分析中，`detach`可能导致非常难缠的bug。我们之前也分析了`join`可能会导致性能异常（坦率说，也可能调试困难），但是在未定义行为（`detach`导致），程序终止（`std::thread`默认导致），或者性能异常之间选择一个后果，可能性能异常是最好的那个。

哎，Item 39表明了使用`ThreadRAII`来保证在`std::thread`的析构时执行`join`有时可能不仅导致程序性能异常，还可能导致程序挂起。“适当”的解决方案是此类程序应该和异步执行的lambda通信，告诉它不需要执行了，可以直接返回，但是C++11中不支持可中断线程。可以自行实现，但是这不是本书讨论的主题。（译者注：关于这一点，C++ Concurrency in Action 的section 9.2 中有详细讨论，也有中文版出版）

Item 17说明因为`ThreadRAII`声明了一个析构函数，因此不会有编译器生成移动操作，但是没有理由`ThreadRAII`对象不能移动。所以需要我们显式声明来告诉编译器自动生成：

```
class ThreadRAII {
public:
  enum class DtorAction{ join, detach }; // see Item 10 for enum class info
  ThreadRAII(std::thread&& t, DtorAction a): action(a), t(std::move(t)) {} // in
dtor, take action a on t
  ~ThreadRAII()
  {
    if (t.joinable()) {
      if (action == DtorAction::join) {
        t.join();
      } else {
        t.detach();
      }
    }
  }

  ThreadRAII(ThreadRAII&&) = default;
  ThreadRAII& operator=(ThreadRAII&&) = default;
  std::thread& get() { return t; } // see below
private:
  DtorAction action;
  std::thread t;
};
```

需要记住的事

- 在所有路径上保证`thread`最终是unjoinable
- 析构时`join`会导致难以调试的性能异常问题
- 析构时`detach`会导致难以调试的未定义行为
- 声明类数据成员时，最后声明`std::thread`类型成员