Item 38：Be aware of varying thread handle destructor behavior.

Item 38：关注不同线程句柄的析构行为

Item 37中说明了joinable的`std::thread`对应于可执行的系统线程。non-defered任务的`future`（参见Item 36）与系统线程有相似的关系。因此，可以将`std::thread`对象和`future`对象都视作系统线程的句柄。

从这个角度来说，有趣的是`std::thread`和`futures`在析构时有相当不同的行为。在Item 37中说明，joinable的`std::thread`析构会终止你的程序，因为两个其他的替代选择--隐式`join`或者隐式`detach`都是更加糟糕的。但是，`futures`的析构表现有时就像执行了隐式`join`，有时又是隐式执行了`detach`，有时又没有执行这两个选择。永远不会造成程序终止。这个线程句柄多种表现值得研究一下。

我们可以观察到实际上`future`是通信信道的一端（被调用者通过该信道将结果发送给调用者）。被调用者（通常是异步执行）将计算结果写入通信信道中（通过`std::promise`对象），调用者使用`future`读取结果。你可以想象成下面的图示，虚线表示信息的流动方向：

但是被调用者的结果存储在哪里？被调用者会在调用者`get`相关的`future`之前执行完成，所以结果不能存储在被调用者的`std::promise`。这个对象是局部的，当被调用者执行结束后，会被销毁。

结果同样不能存储在调用者的`future`，因为`std::future`可能会被用来创建`std::shared_future`(这会将被调用者的结果所有权从`std::future`转移给`std::shared_future`)，而`std::shared_future`在`std::future`被销毁之后被复制很多次。鉴于不是所有的结果都可以被拷贝（有些只能移动）和结果的声明周期与最后一个引用它的`future`一样长，哪个才是被调用者用来存储结果的？这两个问题。

因为与被调用者关联的对象和调用者关联的对象都不适合存储这个结果，必须存储在两者之外的位置。此位置称为共享状态（*shared state*）。共享状态通常是基于堆的对象，但是标准并未指定其类型、接口和实现。标准库的作者可以通过任何他们喜欢的方式来实现共享状态。

我们可以想象调用者，被调用者，共享状态之间关系如下图，虚线还是表示信息的流控方向：

共享状态的存在非常重要，因为`future`的析构行为--这个Item的话题---取决于关联`future`的共享状态。

- Non-defered任务（启动参数为`std::launch::async`）的最后一个关联共享状态的`future`析构函数会在任务完成之前block住。本质上，这种`future`的析构对执行异步任务的线程做了隐式的`join`。
- `future`其他对象的析构简单的销毁。对于异步执行的任务，就像对底层的线程执行`detach`。对于defered任务的最后一种`future`，意味着这个defered任务永远不会执行了。

这些规则听起来好复杂。我们真正要处理的是一个简单的“正常”行为以及一个单独的例外。正常行为是`future`析构函数销毁`future`。那意味着不`join`也不`detach`，只销毁`future`的数据成员（当然，还做了另一件事，就是对于多引用的共享状态引用计数减一。）

正常行为的例外情况仅在同时满足下列所有情况下才会执行：

- 关联`future`的共享状态是被调用了`std::async`创建的
- 任务的启动策略是`std::launch::async`（参见Item 36），原因是运行时系统选择了该策略，或者在对`std::async`的调用中指定了该策略。
- `future`是关联共享状态的最后一个引用。对于`std::future`，情况总是如此，对于`std::shared_future`，如果还有其他的`std::shared_future`引用相同的共享状态没有销毁，就不是。

只有当上面的三个条件都满足时，`future`的析构函数才会表现“异常”行为，就是在异步任务执行完之前block住。实际上，这相当于运行`std::async`创建的任务的线程隐式`join`。

通常会听到将这种异常的析构函数行为称为"Futures from std::async block in their destructors"。作为近似描述没有问题，但是忽略了原因和细节，现在你已经知道了其中三味。

你可能想要了解更加深入。比如“为什么会有这样的规则”（译者注：这里的问题是意译，原文重复了问题本身），这很合理。据我所知，标准委员会希望避免这个问题与隐式`detach`（参见Item 37）相关联，但是不想采取强制程序终止这种激进的方案（因此搞了`join`，同样参见Item 37），所以妥协使用隐式`join`。这个决定并非没有争议，并且认真讨论过在C++14中放弃这种行为。最后，决定先不改变，所以C++11和C++14中这里的行为是一致的。

没有API来提供`future`是否指向`std::async`调用产生的共享状态，因此给定一个`std::future`对象，无法判断是不是会在析构函数block等待异步任务的完成。这就产生了有意思的事情：

```
// this container might block in its dtor, because one or more contained futures
could refer to a shared state for a non-deferred task launched via std::async
std::vector<std::future<void>> futs; // see Item 39 for info on std::future<void>
class Widget // Widget objects might block in their dtors
{
public:
  ...
private:
  std::shared_future<double> fut;
};
```

当然，如果你有办法知道给定的`future`不满足上面条件的任意一条，你就可以确定析构函数不会执行“异常”行为。比如，只有通过`std::async`创建的共享状态才有资格执行“异常”行为，但是有其他创建共享状态的方式。一种是使用`std::packaged_task`，一个`std::packaged_task`对象准备一个函数（或者其他可调用对象）来异步执行，然后将其结果放入共享状态中。然后通过`std::packaged_task`的`get_future`函数获取有关该共享状态的信息：

```
int calcValue(); // func to run
std::packaged_task<int()> pt(calcValue); // wrap calcValue so it can run
asynchrously
auto fut = pt.get_future(); // get future for pt
```

此时，我们知道`future`没有关联`std::async`创建的共享状态，所以析构函数肯定正常方式执行。

一旦被创建，`std::packaged_task`类型的pt可能会在线程上执行。（译者注：后面有些啰嗦的话这里不完整翻译。。大意就是可以再次使用`std::async`来执行，但是那就不用`std::packaged_task`了）

`std::packaged_task`不能拷贝，所以当`pt`被传递给`std::thread`时是右值传递（通过move，参见Item 23）：

```
std::thread t(std::move(pt)); // run pt on t
```

这个例子是你对于`future`的析构函数的正常行为有一些了解，但是将这些语句放在一个作用域的语句块里更容易：

```
{ // begin block
  std::packaged_task<int()> pt(calcValue);
    auto fut = pt.get_future();
  std::thread t(std::move(pt));
  ...
} // end block
```

此处最有趣的代码是在创建`std::thread`对象t之后的"..."。"..."有三种可能性：

- 对t不做什么。这种情况，t会在语句块结束joinable，这会使得程序终止（参见Item 37）
- 对t调用`join`。这种情况，不需要fut的析构函数block，因为`join`被显式调用了
- 对t调用`detach`。这种情况，不需要在fut的析构函数执行`detach`，因为显式调用了

换句话说，当你有一个关联了`std::packaged_task`创建的共享状态的`future`时，不需要采取特殊的销毁策略，通常你会代码中做这些。

需要记住的事

- `future`的正常析构行为就是销毁`future`本身的成员数据
- 最后一个引用`std::async`创建共享状态的`future`析构函数会在任务结束前block