C++11的伟大标志之一是将并发整合到语言和库中。熟悉其他线程API（比如pthreads或者Windows threads）的开发者有时可能会对C++提供的斯巴达式（译者注：应该是简陋和严谨的意思）功能集感到惊讶，这是因为C++对于并发的大量支持是在编译器的约束层面。由此产生的语言保证意味着在C++的历史中，开发者首次通过标准库可以写出跨平台的多线程程序。这位构建表达库奠定了坚实的基础，并发标准库（tasks, futures, threads, mutexes, condition variables, atomic objects等）仅仅是成为并发软件开发者丰富工具集的基础。

在接下来的Item中，记住标准库有两个futures的模板：`std::future`和`std::shared_future`。在许多情况下，区别不重要，所以我们经常简单的混于一谈为*futures*。

优先基于任务编程而不是基于线程

如果开发者想要异步执行 `doAsyncWork` 函数，通常有两种方式。其一是通过创建 `std::thread` 执行 `doAsyncWork`， 比如

```
int doAsyncWork();
std::thread t(doAsyncWork);
```

其二是将 `doAsyncWork` 传递给 `std::async`， 一种基于任务的策略：

```
auto fut = std::async(doAsyncWork); // "fut" for "future"
```

这种方式中，函数对象作为一个任务传递给 `std::async`。

基于任务的方法通常比基于线程的方法更优，原因之一上面的代码已经表明，基于任务的方法代码量更少。我们假设唤醒`doAsyncWork`的代码对于其提供的返回值是有需求的。基于线程的方法对此无能为力，而基于任务的方法可以简单地获取`std::async`返回的`future`提供的`get`函数获取这个返回值。如果`doAsycnWork`发生了异常，`get`函数就显得更为重要，因为`get`函数可以提供抛出异常的访问，而基于线程的方法，如果`doAsyncWork`抛出了异常，线程会直接终止（通过调用`std::terminate`）。

基于线程与基于任务最根本的区别在于抽象层次的高低。基于任务的方式使得开发者从线程管理的细节中解放出来，对此在C++并发软件中总结了'thread'的三种含义：

- 硬件线程（Hardware threads）是真实执行计算的线程。现代计算机体系结构为每个CPU核心提供一个或者多个硬件线程。
- 软件线程（Software threads）（也被称为系统线程）是操作系统管理的在硬件线程上执行的线程。通常可以存在比硬件线程更多数量的软件线程，因为当软件线程被比如 I/O、同步锁或者条件变量阻塞的时候，操作系统可以调度其他未阻塞的软件线程执行提供吞吐量。
- `std::threads`是C++执行过程的对象，并作为软件线程的handle(句柄)。`std::threads`存在多种状态，1. `null`表示空句柄，因为处于默认构造状态（即没有函数来执行），因此不对应任何软件线程。 2. moved from (moved-to的`std::thread`就对应软件进程开始执行) 3. `joined`（连接唤醒与被唤醒的两个线程） 4. `detached`（将两个连接的线程分离）

软件线程是有限的资源。如果开发者试图创建大于系统支持的硬件线程数量，会抛出`std::system_error`异常。即使你编写了不抛出异常的代码，这仍然会发生，比如下面的代码，即使`doAsyncWork`是`noexcept`

```
int doAsyncWork() noexcept; // see Item 14 for noexcept
```

这段代码仍然会抛出异常。

```
std::thread t(doAsyncWork); // throw if no more
                            // threads are available
```

设计良好的软件必须有效地处理这种可能性（软件线程资源耗尽），一种有效的方法是在当前线程执行`doAsyncWork`，但是这可能会导致负载不均，而且如果当前线程是GUI线程，可能会导致响应时间过长的问题；另一种方法是等待当前运行的线程结束之后再创建新的线程，但是仍然有可能当前运行的线程在等待`doAsyncWork`的结果（例如操作得到的变量或者条件变量的通知）。

即使没有超出软件线程的限额，仍然可能会遇到资源超额的麻烦。如果当前准备运行的软件线程大于硬件线程的数量，系统的线程调度程序会将硬件核心的时间切片，当一个软件线程的时间片执行结束，会让给另一个软件线程，即发生上下文切换。软件线程的上下文切换会增加系统的软件线程管理开销，并且如果发生了硬件核心漂移，这个开销会更高，具体来说，如果发生了硬件核心漂移，（1）CPU cache中关于上次执行线程的数据很少，需要重新加载指令；（2）新线程的cache数据会覆盖老线程的数据，如果将来会再次覆盖老线程的数据，显然频繁覆盖增加很多切换开销。

避免资源超额是困难的，因为软件线程之于硬件线程的最佳比例取决于软件线程的执行频率，（比如一个程序从IO密集型变成计算密集型，执行频率是会改变的），而且比例还依赖上下文切换的开销以及软件线程对于CPU cache的使用效率。此外，硬件线程的数量和CPU cache的速度取决于机器的体系结构，即使经过调校，软件比例在某一种机器平台取得较好效果，换一个其他类型的机器这个调校并不能提供较好效果的保证。

而使用`std::async`可以将调校最优比例这件事隐藏于标准库中，在应用层面不需过多考虑

```
auto fut = std::async(doAsyncWork);  // onus of thread mgmt is
                                                                // on
implement of
                                                                // the
Standard Library
```

这种调用方式将线程管理的职责转交给C++标准库的开发者。举个例子，这种调用方式会减少抛出资源超额的异常，为何这么说调用`std::async`并不保证开启一个新的线程，只是提供了执行函数的保证，具体是否创建新的线程来运行此函数，取决于具体实现，比如可以通过调度程序来将`AsyncWork`运行在等待此函数结果的线程上，调度程序的合理性决定了系统是否会抛出资源超额的异常，但是这是库开发者需要考虑的事情了。

如果考虑自己实现在等待结果的线程上运行输出结果的函数，之前提到了可能引出负载不均衡的问题，`std::async`运行时的调度程序显然比开发者更清楚调度策略的制定，因为运行时调度程序管理的是所有执行过程，而不仅仅个别开发者运行的代码。

如果在GUI程序中使用`std::async`会引起响应变慢的问题，还可以通过`std::launch::async`向`std::async`传递调度策略来保证运行函数在不同的线程上执行。

最前沿的线程调度算法使用线程池来避免资源超额的问题，并且通过窃取算法来提升了跨硬件核心的负载均衡。C++标准实际上并不要求使用线程池或者`work-stealing`算法，而且这些技术的实现难度可能比你想象中更有挑战。不过，库开发者在标准库实现中采用了这些前沿的技术，这使得采用基于任务的方式编程的开发者在这些技术发展中持续获得回报，相反如果开发者直接使用`std::thread`编程，处理资源耗竭，负责均衡问题的责任就压在了应用开发者身上，更不说如何使得开发方案跨平台使用。

对比基于线程的开发方式，基于任务的设计为开发者避免了线程管理的痛苦，并且自然提供了一种获取异步执行的结果的方式。当然，仍然存在一些场景直接使用`std::thread`会更有优势：

- **需要访问非常基础的线程API**。C++并发API通常是通过操作系统提供的系统级API(pthreads 或者 windows threads)来实现的，系统级API通常会提供更加灵活的操作方式，举个例子，C++并发API没有线程优先级和affinities的概念。为了提供对底层系统级线程API的访问，`std::thread`对象提供了`native_handle`的成员函数，而在高层抽象的比如`std::futures`没有这种能力。

- **需要优化应用的线程使用**。举个例子，只在特定系统平台运行的软件，可以调教地比使用C++并行API更好的程序性能。
- **需要实现C++并发API之外的线程技术**。举例来说，自行实现线程池技术。

这些都是在应用开发中并不常见的例子，大多数情况，开发者应该优先采用基于任务的编程方式。

记住

- `std::thread`API不能直接访问异步执行的结果，如果执行函数有异常抛出，代码会终止执行
- 基于线程的编程方式关于解决资源超限，负载均衡的方案移植性不佳
- 基于任务的编程方式`std::async`会默认解决上面两条问题