Item 36:Specify std::launch::async if asynchronicity is essential

Item36： 确保在异步为必须时，才指定 `std::launch::async`

当你调用`std::async`执行函数时（或者其他可调用对象），你通常希望异步执行函数。但是这并不一定是你想要`std::async`执行的操作。你确实通过`std::async`launch policy（译者注：这里没有翻译）要求执行函数，有两种标准policy，都通过`std::launch`域的枚举类型表示（参见Item10关于枚举的更多细节）。假定一个函数**f**传给`std::async`来执行：

- **`std::launch::async`的launch policy**意味着f必须异步执行，即在不同的线程
- **`std::launch::deferred`的launch policy**意味着f仅仅在当调用`get`或者`wait`要求`std::async`的返回值时才执行。这表示f推迟到被求值才延迟执行（译者注：异步与并发是两个不同概念，这里侧重于惰性求值）。当`get`或`wait`被调用，f会同步执行，即调用方停止直到f运行结束。如果`get`和`wait`都没有被调用，f将不会被执行

有趣的是，`std::async`的默认launch policy是以上两种都不是。相反，是求或在一起的。下面的两种调用含义相同

```
auto fut1 = std::async(f); // run f using default launch policy
auto fut2 = std::async(std::launch::async | std::launch::deferred, f); // run f
either async or defered
```

因此默认策略允许f异步或者同步执行。如同Item 35中指出，这种灵活性允许`std::async`和标准库的线程管理组件（负责线程的创建或销毁）避免超载。这就是使用`std::async`并发编程如此方便的原因。

但是，使用默认启动策略的`std::async`也有一些有趣的影响。给定一个线程t执行此语句：

```
auto fut = std::async(f); // run f using default launch policy
```

- 无法预测f是否会与t同时运行，因为f可能被安排延迟运行
- 无法预测f是否会在调用`get`或`wait`的线程上执行。如果那个线程是t，含义就是无法预测f是否也在线程t上执行
- 无法预测f是否执行，因为不能确保`get`或者`wait`会被调用

默认启动策略的调度灵活性导致使用线程本地变量比较麻烦，因为这意味着如果f读写了线程本地存储（thread-local storage, TLS），不可能预测到哪个线程的本地变量被访问：

```
auto fut = std::async(f); // TLS for f possibly for independent thread, but
possibly for thread invoking get or wait on fut
```

还会影响到基于超时机制的wait循环，因为在task的`wait_for`或者`wait_until`调用中（参见Item 35）会产生延迟求值（`std::launch::deferred`）。意味着，以下循环看似应该终止，但是实际上永远运行：

```
using namespace std::literals; // for C++14 duration suffixes; see Item 34
void f()
{
  std::this_thread::sleep_for(1s);
}

auto fut = std::async(f);
while (fut.wait_for(100ms) != std::future_status::ready)
{ // loop until f has finished running... which may never happen!
  ...
}
```

如果f与调用`std::async`的线程同时运行（即，如果为f选择的启动策略是`std::launch::async`），这里没有问题（假定f最终执行完毕），但是如果f是延迟执行，`fut.wait_for`将总是返回`std::future_status::deferred`。这表示循环会永远执行下去。

这种错误很容易在开发和单元测试中忽略，因为它可能在负载过高时才能显现出来。当机器负载过重时，任务推迟执行才最有可能发生。毕竟，如果硬件没有超载，没有理由不安排任务并发执行。

修复也是很简单的：只需要检查与`std::async`的future是否被延迟执行即可，那样就会避免进入无限循环。不幸的是，没有直接的方法来查看future是否被延迟执行。相反，你必须调用一个超时函数----比如`wait_for`这种函数。在这个逻辑中，你不想等待任何事，只想查看返回值是否`std::future_status::deferred`，如果是就使用0调用`wait_for`来终止循环。

```
auto fut = std::async(f);
if (fut.wait_for(0s) == std::future_status::deferred) { // if task is deferred
  ... // use wait or get on fut to call f synchronously
}
else { // task isn't deferred
  while(fut.wait_for(100ms) != std::future_status::ready) { // infinite loop not
possible(assuming f finished)
    ... // task is neither deferred nor ready, so do concurrent word until it's
ready
  }
}
```

这些各种考虑的结果就是，只要满足以下条件，`std::async`的默认启动策略就可以使用：

- task不需要和执行`get or wait`的线程并行执行
- 不会读写线程的线程本地变量
- 可以保证在`std::async`返回的将来会调用`get or wait`，或者该任务可能永远不会执行是可以接受的
- 使用`wait_for or wait_until`编码时考虑deferred状态

如果上述条件任何一个都满足不了，你可能想要保证`std::async`的任务真正的异步执行。进行此操作的方法是调用时，将`std::launch::async`作为第一个参数传递：

```
auto fut = std::async(std::launch::async, f); // launch f asynchronously
```

事实上，具有类似`std::async`行为的函数，但是会自动使用`std::launch::async`作为启动策略的工具也是很容易编写的，C++11版本如下：

```
template<typename F, typename... Ts>
inline
std::future<typename std::result_of<F(Ts...)>::type>
reallyAsync(F&& f, Ts&&... params)
{
  return std::async(std::launch::async, std::forward<F>(f), std::forward<Ts>
(params)...);
}
```

这个函数接受一个可调用对象和0或多个参数params然后完美转发（参见Item25）给`std::async`，使用`std::launch::async`作为启动参数。就像`std::async`一样，返回`std::future`类型。确定结果的类型很容易，因为类型特征`std::result_of`可以提供（参见Item 9 关于类型特征的详细表述）。

`reallyAsync`就像`std::async`一样使用：

```
auto fut = reallyAsync(f);
```

在C++14中，返回类型的推导能力可以简化函数的定义：

```
template<typename f, typename... Ts>
inline
auto
reallyAsync(F&& f, Ts&&... params)
{
  return std::async(std::launch::async, std::forward<T>(f), std::forward<Ts>
(params)...);
}
```

这个版本清楚表明，`reallyAsync`除了使用`std::launch::async`启动策略之外什么也没有做。

需要记住的事

- `std::async`的默认启动策略是异步或者同步的
- 灵活性导致访问**thread_locals**的不确定性，隐含了task可能不会被执行的意思，会影响程序基于`wait`的超时逻辑
- 只有确实异步时才指定`std::launch::async`