深入研究Task实例方法ContinueWith的参数TaskContinuationOptions

揭秘：

该章节的性质和上一个章节类似，也是一个扩展的章节，主要来研究Task类下的实例方法ContinueWith中的参数TaskContinuationOptions。

通过F12查看TaskContinuationOptions的源码，知道主要有这么几个参数：

①. LazyCancellation：在延续取消的情况下，防止延续的完成直到完成先前的任务。

(下面的例子task2取消，原先的延续关系不复存在，task1和task3可以并行执行)

②. ExecuteSynchronously：希望执行前面那个task的thread也在执行本延续任务

(下面的例子执行task2的Thread和执行task1的Thread是同一个，所有二者的线程id相同)

③. NotOnRanToCompletion和OnlyOnRanToCompletion

NotOnRanToCompletion：延续任务必须在前面task非完成状态才能执行

OnlyOnRanToCompletion：延续任务必须在前面task完成状态才能执行

(下面例子：注释掉异常的这句代码task2不能执行，task3能执行；不注释，task2能执行，task3不能执行)

线程的分类

线程分类：
用户级线程和内核级线程（轻量级进程）

用户级线程：
在一个纯粹的用户级线程软件中，有关线程管理的所有工作都由应用程序完成，内核意识不到线程的存在。

使用用户级线程而非内核级线程有很多优点：
（1）由于所有线程管理数据结构都在一个进程的用户地址空间中，线程切换不需要内核态特权，因此，进程不需要为了线程管理而切换到内核态，这节省了两次状态转换（从用户态到内核态，从内核态返回到用户态）的开销。
（2）调度可以是应用程序相关的。一个应用程序可能更适合简单的轮转调度算法，二个应用程序可能更适合基于优先级的调度算法。可以做到为应用程序量身定做调度算法而不影响底层的操作系统调度程序。
（3）用户级线程可以在任何操作系统中运行，不需要对底层内核进行修改以支持用户级线程。线程库是一组供所有应用程序共享的应用程序共享的应用程序级别的函数。

用户级线程相对于内核级线程的缺点：
（1）在典型的操作系统中，许多系统调用会引起阻塞。因此，当用户级线程执行一个系统调用时，不进这个线程会被阻塞，进程中所有线程都会被阻塞。
（2）在纯粹的用户级线程中，一个多线程应用程序不能利用多处理技术。内核一次只把一个进程分配给一个处理器，因此一个进程中只有一个线程可以执行。

解决办法：
（1）把应用程序写成一个多进程应用程序而非多线程应用程序，但这种方法消除了线程的主要优点：每次切换都变成了进程间的切换，而不是线程间的切换，导致开销过大。
（2）使用jacketing技术，jacketing目标是把一个产生阻塞的系统调用转换成一个非阻塞的系统调用。

内核级线程：
在一个纯粹的内核级应用软件中，有关线程的所有工作都是由内核完成的，应用程序部分没有进行线程管理的代码，只有一个到内核线程设施的应用程序编程接口（API）。

内核为进程及其内部的每个线程维护上下文信息，调度是基于线程完成的。

该方法克服了用户级线程的两个基本缺陷：
首先，内核可以同时把同一个进程中的多个线程调度到多个处理器中，再者，如果进程中的一个线程被阻塞，内核可以调度同一个线程中的另一个线程。

内核级线程方法的优点是：内核例程自身也是可以使用多线程的。
缺点：在把控制从一个线程传送到同一个进程内的另一个线程时，需要到内核的状态切换。

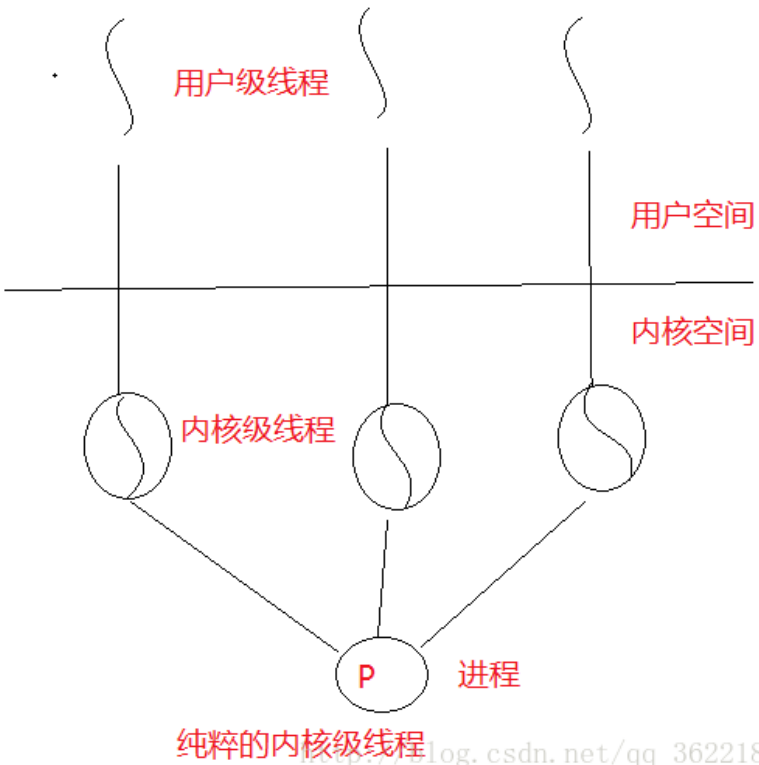
下表是单处理器VAX机上运行类UNIX操作系统的测量结果。
进行了两种测试：Null Fork（测试创间，调度，执行和完成一个调用空过程的进程/线程的时间，即就是派生一个进程、线程的开销）和Single-Wait（测量进程/线程给正在等待的进程/线程发信号，然后在

某个条件上等待所需要的时间，即就是两个进程/线程的同步时间）。

操作	用户级线程	内核级线程	进程
Null Fork	3 4	9 4 8	1 1 3 0 0
Single-Wait	3 7	4 4 1	1 8 4 0

从上表可以看出，用户级线程和内核级线程之间、内核级线程和进程之间都有一个数量级以上的性能差异。

从表面上看，虽然使用内核级进程和多线程技术会比使用单线程的进程有明显的速度提高，使用用户级线程却比内核级线程有额外的提高。不过额外的提高取决于应用程序的性质。如果应用程序中大多数线程切换都需要内核态的访问，那么基于用户级线程的方案不会比基于内核级线程的方案好多少。

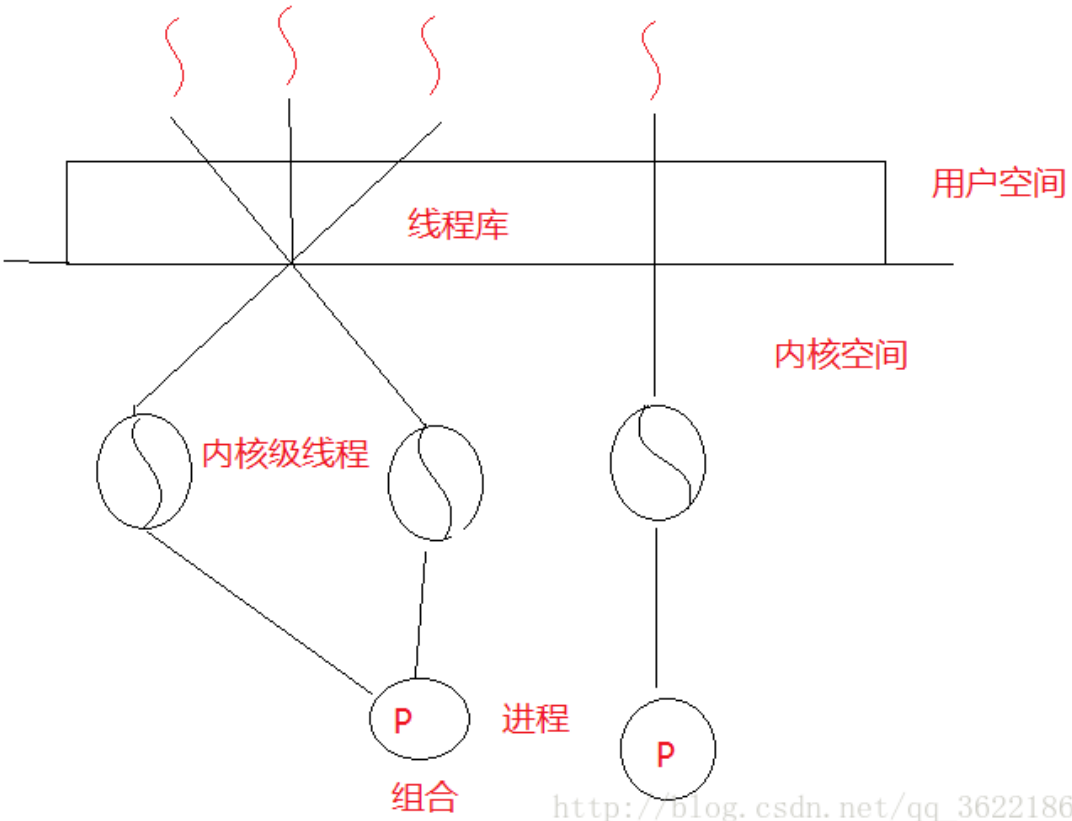


混合方法：

在混合系统中，线程创建完全在用户空间中完成，现成的调度和同步也在应用程序中进行。

一个应用程序中的多个用户及线槽被映射到一些（小于或者等于用户级线程的数目）内核到线程上。

程序员可以为特定的应用程序和处理器调节内核级线程的数目，以达到最佳效果。



在混合方法中，同一个应用程序中的多个线程可以在多个处理器上并行的运行，某个会引起阻塞的系统调用不会阻塞整个进程。

c#之task与thread区别及其使用

1.什么是thread
当我们提及多线程的时候会想到thread和threadpool，这都是异步操作，threadpool其实就是thread的集合，具有很多优势，不过在任务多的时候全局队列会存在竞争而消耗资源。thread默认为前台线程，主程序必须等线程跑完才会关闭，而threadpool相反。
总结：threadpool确实比thread性能优，但是两者都没有很好的api区控制，如果线程执行无响应就只能等待结束，从而诞生了task任务。
2.什么是task
task简单地看就是任务，那和thread有什么区别呢？Task的背后的实现也是使用了线程池线程，但它的性能优于ThreadPoll,因为它使用的不是线程池的全局队列，而是使用的本地队列，使线程之间的资源竞争减少。同时Task提供了丰富的API来管理线程、控制。但是相对前面的两种耗内存，Task依赖于CPU对于多核的CPU性能远超前两者，单核的CPU三者的性能没什么差别。

内核态和用户态的区别

就像世界上的人并不平等一样，并不是所有的程序都是平等的。世界上有的人占有资源多，有的人占有资源少，有的人来了，别人得让出资源，有的人则专门为别人让出资源。程序也是这样，有的程序可以访问计算机的任何资源，有的程序则只能访问非常受限的少量资源。而操作系统作为计算机的管理者，自然不能和被管理者享受一样的待遇，它应该享有更多的方便或特权。为了区分不用程序的不同权利，人们发胆了内核和用户态的概念。

那么什么是内核态，什么是用户态呢？只要想一想现实生活中，处于社会核心的人与处于社会边缘的人有什么区别就能明白处于核心的人拥有的资源多！因此，内核态就是拥有资源多的状态，或者说访问资源多的状态，我们也称之为特权态。相对来说，用户态就是非特权态，在此种状态下访问的资源将受到限制。如果一个程序运行在特权态，则该程序就可以访问计算机的任何资源，即它的资源访问权限不受限制。如果一个程序运行在用户态，则其资源需求将受到各种限制。

例如，如果要访问操作系统的内核数据结构，如进程表，则需要在特权态下才能办到。如果要访问用户程序里的数据，则在用户态下就可以了。

由于内核态的程序可以访问计算机的所有资源，这种程序的可靠性和安全性就显得十分重要。试想如果一个不可靠的程序在内核态下修改了操作系统的各种内核数据结构，结果会怎样呢？整个系统有可能崩溃。而运行于用户态的程序就比较简单了，如果其可靠性和安全性出了问题，其造成的损失只不过是让用户程序崩溃，而操作系统将继续运行。

很显然，内核态和用户态各有优势：运行在内核态的程序可以访问的资源多，但可靠性、安全性要求高，维护管理都较复杂；用户态程序访问的资源受限，但可靠性、安全性要求低，自然编写维护起来都较简单。一个程序到底应该运行在内核态还是用户态取决于其对资源和效率的需求。

一般来说，一个程序能够运行于用户态，就应该让它运行在用户态。只在迫不得已的情况下，才让程序运行于内核态。只要看看一个国家的治理就清楚了。我们拿什么标准来判断什么事情应该归国家领导管理。凡是牵扯到计算机本体根本运行的事情都应该在内核态下执行，只与用户数据和应用相关的东西则放在用户态执行。另外，对时序要求特别高的事情，也应该在内核态做。你有没有想过，国家领导出门怎么不塞车呢？

那么什么样的功能应该在内核态下实现呢？ 首先，CPU管理和内存管理都应该在内核态实现。这些功能可不可以在用户态下实现呢？当然能，但是不太安全。就像一个国家的军队（CPU和内存在计算机里的地位就相当于一个国家的军队的地位）交给老百姓来管一样，是非常危险的。所以从保障计算机安全的角度来说，CPU和内存的管理必须在内核态实现。

诊断与测试程序也需要在内核态下实现。因为诊断和测试需要访问计算机的所有资源，否则怎么判断计算机是否正常呢？就像中医治病，必须把脉触摸病人。你不让中医触摸，他怎么能看病呢（当然，很多人认为中医是伪科学，根本治不了病，本书对此问题不做讨论）？输入输出管理也一样，因为要访问各种设备和底层数据结构，也必须在内核态实现。

对于文件系统来说，则可以一部分放在用户态，一部分放在内核态。文件系统本身的管理，即文件系统的宏数据部分的管理，必须放在内核态，不然任何人都可能破坏文件系统的结构；而用户数据的管理，则可以放在用户态。编译器、网络管理的部分功能、编辑器用户程序，自然都可以放在用户态下执行。图3.8描述的是Windows操作系统的内核态与用户态的界线。

用户态和内核态的转换

1）用户态切换到内核态的3种方式

a. 系统调用
这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。
b. 异常
当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。
c. 外围设备的中断
当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

这3种方式是系统在运行时由用户态转到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。

2）具体的切换操作
从触发方式上看，可以认为存在前述3种不同的类型，但是从最终实际完成由用户态到内核态的切换操作上来说，涉及的关键步骤是完全一致的，没有任何区别，都相当于执行了一个中断响应的过程，因为系统调用实际上最终是中断机制实现的，而异常和中断的处理机制基本上也是一致的，关于它们的具体区别这里不再赘述。关于中断处理机制的细节和步骤这里也不做过多分析，涉及到由用户态切换到内核态的步骤主要包括：
[1] 从当前进程的描述符中提取其内核栈的ss0及esp0信息。
[2] 使用ss0和esp0指向的内核栈将当前进程的cs,eip,eflags,ss,esp信息保存起来，这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。
[3] 将先前由中断向量检索得到的中断处理程序的cs,eip信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。