# 进程、线程之间的爱恨纠葛...

当一个程序开始执行后，在开始执行到执行完毕退出这段时间内，它在内存中的部分就叫称作一个进程。

Linux 是一个多任务的操作系统，也就是说，在同一时间内，可以有多个进程同时执行。我们大家常用的单CPU计算机实际上在一个时间片段内只能执行一条指令。那么Linux是如何实现多进程的同时执行的呢？原来Linux使用了一种称为” 进程调度 “的手段，首先，为每个进程指派一定的运行时间，这个时间通常很短，短到以毫秒为单位，然后依照某种规则，从众多的进程中挑选一个投入运行，其他进程暂时等待，当正在运行的那个进程时间耗尽，或者执行完毕退出，或因某种原因暂停，Linux就会重新调度，挑选一个进程投入运行，因为每个进程占用的时间片段都很短，从使用者的角度来看，就好像多个进程同时运行一样。

在Linux中，每个进程在创建的时都会被分配一个数据结构，称为进程控制块（PCB）。PCB中包含了很多重要的信息，供系统调度和进程本事执行使用，其中最重要的莫过于进程的ID，进程的ID也被称为进程标示符，是一个非负的整数，在Linux操作性系统中唯一的标志一个进程。在最常使用的I386架构上，一个非负的整数的取值是0~32767，这也是我们所可能取到的进程ID，它就是进程的身份证号码。

## **僵尸进程的产生**

僵尸进程就是已经结束的进程，但是还没有从进程表中删除。僵尸进程太多会导致进程表里面条目满了，进而导致系统崩溃，倒是不占用系统资源。

在进程的状态中，僵尸进程是非常特殊的一种，它已经放弃了几乎所有的内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个为位置，记载该进程的退出状态等信息供其他进程收集，除此之外，僵尸进程不再占用任何内存空间，它需要它的父进程来给它收尸，如果父进程没安装SIGCHLD信号处理函数调用wait或waitpid（）等待子进程结束，又没有显示的忽略该信号，那么它就一直处于僵尸状态。如果父进程结束了，那么init进程会自动接手这个子进程，为它收尸，他还是能够被清除的。但是如果父进程是一个循环，不会结束，那么子进程就一直处于僵尸状态。

## **僵尸进程产生的原因:**

每个Linux进程在进程表中都有一个进入点（Entry），核心程序在执行该进程时使用到的一切信息都存储在进入点。当使用ps命令查看系统中的进程信息时，看到的就是进程表中的相关数据。当fork系统调用建立一个新的进程以后，核心进程就会在进程表中给这个新进程分配一个进入点，然后将相关信息存储在该进入点所对应的进程表中，这些信息中有一项是父进程的识别码。当这个进程走完了自己的生命周期后，它会执行exit（）系统调用，此时原来进程表中的数据会被该进程的退出码、执行时所用的CPU时间等数据所取代，这些数据会一直保留到系统将它传递给它的父进程为止。由此可见，僵尸进程的出现时间实在子程序终止后，但是父进程尚未读取这些数据之前。

## **如何避免僵尸进程**

1、父进程通过wait和waitpid等函数等待子进程结束，这会导致父进程挂起

2、如果父进程很忙，那么可以用signal函数为SIGCHLD安装handler，因为子进程结束后，父进程会收到该信号，可以在handler中调用wait回收。

3、如果父进程不关心子进程什么时候结束，那么可以用“singal（SIGCHLD），SIG\_IGN”通知内核，自己对子进程的结束不感兴趣，那么子进程结束后，内核会回收，并不再给父进程发送信号。

4、还有一些技巧，就是fork（）两次，父进程fork一个子进程，然后继续工作，子进程fork一个孙进程后退出，那么孙进程被init接管，孙进程结束后，init会回收，不过子进程回收还要自己做。

## **进程 PK 线程**

我们先打个比方，多线程是十字路口多线程是平面交通系统，造价低，但是红绿灯多，老堵车，而多进程是则是立交桥，虽然造价高，上下坡多耗油，但是不堵车。这是一个抽象的概念。相信大家看完会有这种感觉。

进程和线程是两个相对的概念，通常来说，一个进程可以定义程序的一个实例（Instance）。在Win32中，进程并不执行什么，它只是占据应用程序所使用的地址空间。为了让进程完成一定的工作，进程必须至少占有一个线程，正是这个线程负责包含进程地址空间中的代码。实际上，一个进程可以包含几个线程，它们可以同时执行进程地址空间中的代码。为了做到这一点，每个线程有自己的一组CPU寄存器和堆栈。每个进程中至少有 一个线程在执行其地址空间中的代码。如果没有线程执行进程地址空间中的代码，进程也就没有继续存在的理由，系统将自动清除进程及其地址空间。

## **多线程的实现原理**

创建一个进程时，它的第一个线程称为主线程（Primary thread），由系统自动生成。然后可以由这个主线程生成额外的线程，而这些线程，又可以生成更多的线程。在运行一个多线程的程序时，从表面上看，这些线程似乎在同时运行。而实际情况并非如此，为了运行所有的这些线程，操作系统为每个独立线程安排一些CPU时间。单CPU操作系统以时间片轮转方式向线程提供时间片（Quantum），每个线程在使用完时间片后交出控制，系统再将CPU时间片分配给下一个线程。由于每个时间片足够的短，这样就给人一种假象，好像这些线程在同时运行。创建额外线程的唯一目的就是尽可能地利用CPU时间。

## **多线程的问题**

使用多线程编程可以给程序员带来很大的灵活性，同时也使原来需要复杂技巧才能解决的问题变得容易起来。但是，不应该人为地将编写的程序分成一些碎片，让这些碎片按各自的线程执行，这不是开发应用程序的正确方法。线程很有用，但当使用线程时，可能会在解决老问题的同时产生新问题。例如要开发一个字处理程序，并想让打印功能作为单独的线程自己执行。这听起来是很好的主意，因为在打印时，用户可立即返回，开始编辑文档。但这样一来，在该文档被打印时文档中的数据就有可能被修改，打印的结果就不再是所期望的内容。也许最好不要把打印功能放在单独的线程中，不过如果一定要用多线程的话，也可以考虑用下面的方法解决：第一种方法是锁定正在打印的文档，让用户编辑其他的文档，这样在结束打印之前，该文档不会作任何修改；另一个方法可能更有效一些，即可以把该文档拷贝到一个临时文件中，打印这个临时文件的内容，同时允许用户对原来的文档进行修改。当包含文档的临时文件打印完成时，再删去这个临时文件。通过上面的分析可以看出，多线程在帮助解决问题的同时也可能带来新问题。因此有必要弄清楚，什么时候需要创建多线程，什么时候不需要多线程。总的来说，多线程往往用于在前台操作的同时还需要进行后台的计算或逻辑判断的情况。

## **线程的分类**

在MFC中，线程被分为两类，即工作线程和用户界面线程。如果一个线程只完成后台计算，不需要和用户交互，那么可以使用工作线程；如果需要创建一个处理用户界面的线程，则应使用用户界面线程。这两者的主要区别在于，MFC框架会给用户界面线程增加一个消息循环，这样用户界面线程就可以处理自己消息队列中的消息。这样看来，如果需要在后台作一些简单的计算（如对电子表格的重算），则首先应考虑使用工作线程，而当 后台线程需要处理比较复杂的任务，确切地说，当后台线程的执行过程会随着实际情况的不同而改变时，就应该使用用户界面线程，以便能对不同的消息作出响应。

## **线程的优先级**

当系统需要同时执行多个进程或多个线程时，有时会需要指定线程的优先级。线程的优先级一般是指这个线程的基优先级，即线程相对于本进程的相对优先级和包含此线程的进程的优先级的结合。操作系统以优先级为基础安排所有的活动线程，系统的每一个线程都被分配了一个优先级，优先级的范围从0到31。运行时，系统简单地给第一个优先级为31的线程分配CPU时间，在该线程的时间片结束后，系统给下一个优先级为31的线程分配CPU时间。当没有优先级为31的线程时，系统将开始给优先级为30的线程分配CPU时间，以此类推。除了程序员在程序中改变线程的优先级外，有时程序在执行过程中系统也会自动地动态改变线程的优先级，这是为了保证系统对终端用户的高度响应性。比如用户按了键盘上的某个键时，系统就会临时将处理WM\_KEYDOWN消息的线程的优先级提高2到3。CPU按一个完整的时间片执行线程，当时间片执行完毕后，系统将该线程的优先级减1。

## **线程的同步**

在使用多线程编程时，还有一个非常重要的问题就是线程同步。所谓线程同步是指线程之间在相互通信时避免破坏各自数据的能力。同步问题是由前面说到的Win32系统的CPU时间片分配方式引起的。虽然在某一时刻，只有一个线程占用CPU（单CPU时）时间，但是没有办法知道在什么时候，在什么地方线程被打断，这样如何保证线程之间不破坏彼此的数据就显得格外重要。在MFC中，可以使用4个同步对象来保证多线程同时运行。它们分别是临界区对象（CCriticalSection）、互斥量对象（CMutex）、信号量对象（CS emaphore）和事件对象（CEvent）。在这些对象中，临界区对象使用起来最简单，它的缺点是只能同步同一个进程中的线程。另外，还有一种基本的方法，本文称为线性化方法，即在编程过程中对一定数据的写操作都在一个线程中完成。这样，由于同一线程中的代码总是按顺序执行的，就不可能出现同时改写数据的情况。

## **总结：**

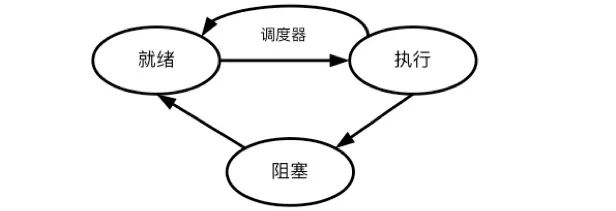
在线程中（相对与进程而言），线程是一个更加接近执行体的概念，它可以与同进程的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。这两者都可以提高程序的并发度，提高程序运行的效率和响应的时间。线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源管理和保护；而进程正好相反。根本的区别就一点：用多进程每个进程有自己的地址空间，线程则共享地址空间，在速度方面：线程产生的速度快，线程间的通讯快，切换快等，因为他们在同一地址空间内。在资源利用率方面：线程的资源率比较好也是因为他们在同一地址空间内。 在同步方面：线程使用公共变量/内存时需要使用同步机制，因为他们在同一地址空间内进程中：子进程是父进程的复制品，子进程获得父进程数据空间、堆和栈的复制品。

# Linux进程及其调度策略

进程是操作系统虚拟出来的概念，用来组织计算机中的任务。它从诞生到随着CPU时间执行，直到最终消失。不过，进程的生命都得到了操作系统内核的关照。就好像疲于照顾几个孩子的母亲内核必须做出决定，如何在进程间分配有限的计算资源，最终让用户获得最佳的使用体验。内核中安排进程执行的模块称为**调度器**（scheduler）。这里将介绍调度器的工作方式。

## **进程状态**

调度器可以切换进程状态（process state）。一个Linux进程从被创建到死亡，可能会经过很多种状态，比如执行、暂停、可中断睡眠、不可中断睡眠、退出等。我们可以把Linux下繁多的进程状态，归纳为三种基本状态。



**图1 进程的基本状态**

进程创建后，就自动变成了就绪状态。如果内核把CPU时间分配给该进程，那么进程就从就绪状态变成了执行状态。

在执行状态下，进程执行指令，最为活跃。正在执行的进程可以主动进入阻塞状态，比如这个进程需要将一部分硬盘中的数据读取到内存中。在这段读取时间里，进程不需要使用CPU，可以主动进入阻塞状态，让出CPU。

当读取结束时，计算机硬件发出信号，进程再从阻塞状态恢复为就绪状态。进程也可以被迫进入阻塞状态，比如接收到SIGSTOP信号。

调度器是CPU时间的管理员。**Linux调度器需要负责做两件事**：一件事是选择某些就绪的进程来执行；另一件事是打断某些执行中的进程，让它们变回就绪状态。不过，并不是所有的调度器都有第二个功能。

调度器在让一个进程变回就绪时，就会立即让另一个就绪的进程开始执行。多个进程接替使用CPU，从而最大效率地利用CPU时间。当然，如果执行中进程主动进入阻塞状态，那么调度器也会选择另一个就绪进程来消费CPU时间。

所谓的**上下文切换**（context switch）就是指进程在CPU中切换执行的过程。内核承担了上下文切换的任务，负责储存和重建进程被切换掉之前的CPU状态，从而让进程感觉不到自己的执行被中断。应用程序的开发者在编写计算机程序时，就不用专门写代码处理上下文切换了。

## **进程的优先级**

调度器分配CPU时间的基本依据，就是进程的优先级。根据程序任务性质的不同，程序可以有不同的执行优先级。根据优先级特点，我们可以把进程分为两种类别。

**1、实时进程**（Real-Time Process）：优先级高、需要尽快被执行的进程。它们一定不能被普通进程所阻挡，例如视频播放、各种监测系统。

**2、普通进程**（Normal Process）：优先级低、更长执行时间的进程。例如文本编译器、批处理一段文档、图形渲染。

普通进程根据行为的不同，还可以被分成互动进程（interactive process）和批处理进程（batch process）。互动进程的例子有图形界面，它们可能处在长时间的等待状态，例如等待用户的输入。一旦特定事件发生，互动进程需要尽快被激活。一般来说，图形界面的反应时间是50到100毫秒。批处理进程没有与用户交互的，往往在后台被默默地执行。

**实时进程**由Linux操作系统创造，普通用户只能创建普通进程。两种进程的优先级不同，**实时进程的优先级永远高于普通进程。**进程的优先级是一个0到139的整数。数字越小，优先级越高。其中，优先级0到99留给实时进程，100到139留给普通进程。

一个普通进程的默认优先级是120。我们可以用命令nice来修改一个进程的默认优先级。例如有一个可执行程序叫app，执行命令：

$nice -n -20 ./app

命令中的-20指的是从默认优先级上减去20。通过这个命令执行app程序，内核会将app进程的默认优先级设置成100，也就是普通进程的最高优先级。命令中的-20可以被换成-20至19中任何一个整数，包括**-20**和**19**。默认优先级将会变成执行时的静态优先级（static priority）。调度器最终使用的优先级根据的是进程的动态优先级：动态优先级 = 静态优先级 – Bonus + 5

如果这个公式的计算结果小于100或大于139，将会取100到139范围内最接近计算结果的数字作为实际的动态优先级。公式中的Bonus是一个估计值，这个数字越大，代表着它可能越需要被优先执行。如果内核发现这个进程需要经常跟用户交互，将会把Bonus值设置成大于5的数字。如果进程不经常跟用户交互，内核将会把进程的Bonus设置成小于5的数。

**O(n)和O(1)调度器**

下面介绍Linux的调度策略。最原始的调度策略是按照优先级排列好进程，等到一个进程运行完了再运行优先级较低的一个，但这种策略完全无法发挥多任务系统的优势。因此，随着时间推移，操作系统的调度器也多次进化。

0先来看Linux 2.4内核推出的O(n)调度器。O(n)这个名字，来源于算法复杂度的大O表示法。大O符号代表这个算法在最坏情况下的复杂度。字母n在这里代表操作系统中的活跃进程数量。O(n)表示这个调度器的时间复杂度和活跃进程的数量成正比。

O(n)调度器把时间分成大量的微小时间片（Epoch）。在每个时间片开始的时候，调度器会检查所有处在就绪状态的进程。调度器计算每个进程的优先级，然后选择优先级最高的进程来执行。一旦被调度器切换到执行，进程可以不被打扰地用尽这个时间片。如果进程没有用尽时间片，那么该时间片的剩余时间会增加到下一个时间片中。

O(n)调度器在每次使用时间片前都要检查所有就绪进程的优先级。这个检查时间和进程中进程数目n成正比，这也正是该调度器复杂度为O(n)的原因。当计算机中有大量进程在运行时，这个调度器的性能将会被大大降低。也就是说，O(n)调度器没有很好的可拓展性。O(n)调度器是Linux 2.6之前使用的进程调度器。

为了解决O(n)调度器的性能问题，O(1)调度器被发明了出来，并从Linux 2.6内核开始使用。顾名思义，O(1)调度器是指调度器每次选择要执行的进程的时间都是1个单位的常数，和系统中的进程数量无关。这样，就算系统中有大量的进程，调度器的性能也不会下降。

O(1)调度器的创新之处在于，它会把进程按照优先级排好，放入特定的数据结构中。在选择下一个要执行的进程时，调度器不用遍历进程，就可以直接选择优先级最高的进程。

和O(n)调度器类似，O(1)也是把时间片分配给进程。优先级为120以下的进程时间片为：

(140–priority)×20毫秒

优先级120及以上的进程时间片为：

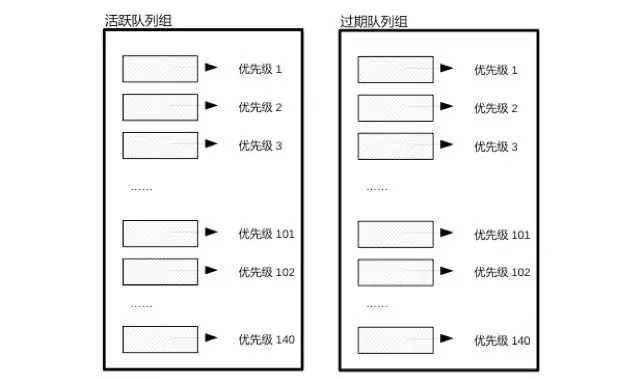
(140–priority)×5 毫秒

O(1)调度器会用两个队列来存放进程。一个队列称为活跃队列，用于存储那些待分配时间片的进程。另一个队列称为过期队列，用于存储那些已经享用过时间片的进程。

**O(1)调度器把**时间片从活跃队列中调出一个进程。这个进程用尽时间片，就会转移到过期队列。当活跃队列的所有进程都被执行过后，调度器就会把活跃队列和过期队列对调，用同样的方式继续执行这些进程。

上面的描述没有考虑优先级。加入优先级后，情况会变得复杂一些。操作系统会创建140个活跃队列和过期队列，对应优先级0到139的进程。一开始，所有进程都会放在活跃队列中。

然后操作系统会从优先级最高的活跃队列开始依次选择进程来执行，如果两个进程的优先级相同，他们有相同的概率被选中。执行一次后，这个进程会被从活跃队列中剔除。如果这个进程在这次时间片中没有彻底完成，它会被加入优先级相同的过期队列中。当140个活跃队列的所有进程都被执行完后，过期队列中将会有很多进程。调度器将对调优先级相同的活跃队列和过期队列继续执行下去。过期队列和活跃队列，如图2所示。



**图2 过期队列和活跃队列（需要替换）**

我们下面看一个例子，有五个进程，如表1所示。



**表1 进程**

Linux操作系统中的进程队列（run queue），如表2所示。



**表2 进程队列**

那么在一个执行周期，被选中的进程依次是先A，然后B和C，随后是D，最后是E。

注意，普通进程的执行策略并没有保证优先级为100的进程会先被执行完进入结束状态，再执行优先级为101的进程，而是在每个对调活跃和过期队列的周期中都有机会被执行，这种设计是为了避免进程饥饿（starvation）。所谓的进程饥饿，就是优先级低的进程很久都没有机会被执行。

我们看到，O(1)调度器在挑选下一个要执行的进程时很简单，不需要遍历所有进程。但是它依然有一些缺点。进程的运行顺序和时间片长度极度依赖于优先级。比如，计算优先级为100、110、120、130和139这几个进程的时间片长度，如表3所示。



**表3 进程的时间片长度**

从表格中你会发现，优先级为110和120的进程的时间片长度差距比120和130之间的大了10倍。也就是说，进程时间片长度的计算存在很大的随机性。O(1)调度器会根据平均休眠时间来调整进程优先级。该调度器假设那些休眠时间长的进程是在等待用户互动。这些互动类的进程应该获得更高的优先级，以便给用户更好的体验。一旦这个假设不成立，O(1)调度器对CPU的调配就会出现问题。

## **完全公平调度器**

从2007年发布的Linux 2.6.23版本起，完全公平调度器（CFS，Completely Fair Scheduler）取代了O(1)调度器。CFS调度器不对进程进行任何形式的估计和猜测。这一点和O(1)区分互动和非互动进程的做法完全不同。

CFS调度器增加了一个虚拟运行时（virtual runtime）的概念。每次一个进程在CPU中被执行了一段时间，就会增加它虚拟运行时的记录。在每次选择要执行的进程时，不是选择优先级最高的进程，而是选择虚拟运行时最少的进程。完全公平调度器用一种叫红黑树的数据结构取代了O(1)调度器的140个队列。红黑树可以高效地找到虚拟运行最小的进程。

我们先通过例子来看CFS调度器。假如一台运行的计算机中本来拥有A、B、C、D四个进程。内核记录着每个进程的虚拟运行时，如表4所示。



**表4 每个进程的虚拟运行时**

系统增加一个新的进程E。新创建进程的虚拟运行时不会被设置成0，而会被设置成当前所有进程最小的虚拟运行时。这能保证该进程被较快地执行。在原来的进程中，最小虚拟运行时是进程A的1 000纳秒，因此E的初始虚拟运行时会被设置为1 000纳秒。新的进程列表如表5所示。



**表5 新的进程列表**

假如调度器需要选择下一个执行的进程，进程A会被选中执行。进程A会执行一个调度器决定的时间片。假如进程A运行了250纳秒，那它的虚拟运行时增加。而其他的进程没有运行，所以虚拟运行时不变。在A消耗完时间片后，更新后的进程列表，如表6所示。



**表6 更新后的进程列表**

可以看到，进程A的排序下降到了第三位，下一个将要被执行的进程是进程E。从本质上看，虚拟运行时代表了该进程已经消耗了多少CPU时间。如果它消耗得少，那么理应优先获得计算资源。

按照上述的基本设计理念，CFS调度器能让所有进程公平地使用CPU。听起来，这让进程的优先级变得毫无意义。CFS调度器也考虑到了这一点。CFS调度器会根据进程的优先级来计算一个时间片因子。同样是增加250纳秒的虚拟运行时，优先级低的进程实际获得的可能只有200纳秒，而优先级高的进程实际获得可能有300纳秒。这样，优先级高的进程就获得了更多的计算资源。

以上就是调度器的基本原理，以及Linux用过的几种调度策略。调度器可以更加合理地把CPU时间分配给进程。现代计算机都是多任务系统，调度器在多任务系统中起着顶梁柱的作用。

# 你知道何为线程与进程吗？？

## 什么是线程

什么是线程？线程与进程与有什么关系？这是一个非常抽象的问题，也是一个特别广的话题，涉及到非常多的知识。我不能确保能把它讲的话，也不能确保讲的内容全部都正确。即使这样，我也希望尽可能地把他讲通俗一点，讲的明白一点，因为这是个一直困扰我很久的，扑朔迷离的知识领域，希望通过我的理解揭开它一层一层神秘的面纱。

## 任务调度

线程是什么？要理解这个概念，须要先了解一下操作系统的一些相关概念。大部分操作系统(如Windows、Linux)的任务调度是采用时间片轮转的抢占式调度方式，也就是说一个任务执行一小段时间后强制暂停去执行下一个任务，每个任务轮流执行。任务执行的一小段时间叫做时间片，任务正在执行时的状态叫运行状态，任务执行一段时间后强制暂停去执行下一个任务，被暂停的任务就处于就绪状态等待下一个属于它的时间片的到来。这样每个任务都能得到执行，由于CPU的执行效率非常高，时间片非常短，在各个任务之间快速地切换，给人的感觉就是多个任务在“同时进行”，这也就是我们所说的并发(别觉得并发有多高深，它的实现很复杂，但它的概念很简单，就是一句话：多个任务同时执行)。多任务运行过程的示意图如下：

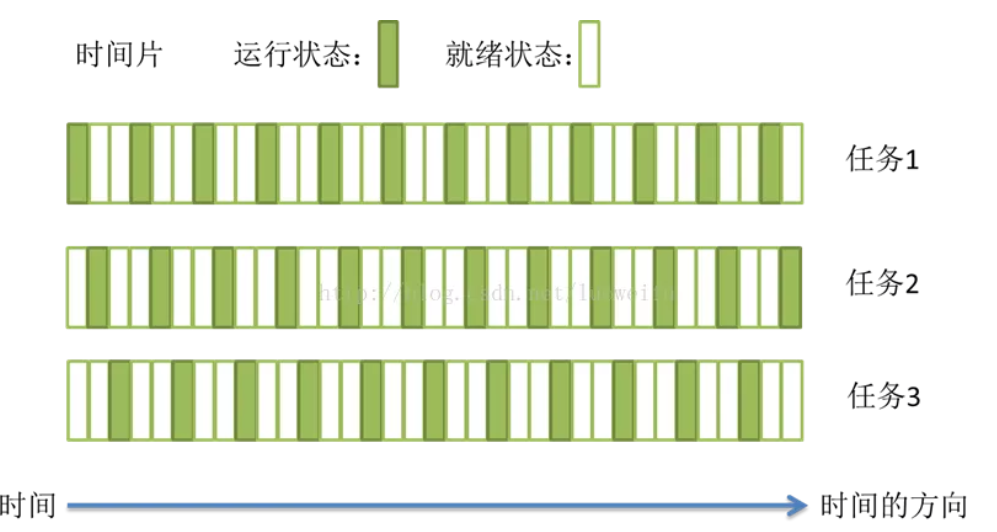


图 1：操作系统中的任务调度

## 进程

我们都知道计算机的核心是CPU，它承担了所有的计算任务；而操作系统是计算机的管理者，它负责任务的调度、资源的分配和管理，统领整个计算机硬件；应用程序侧是具有某种功能的程序，程序是运行于操作系统之上的。

进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用程序运行的载体。进程是一种抽象的概念，从来没有统一的标准定义。进程一般由程序、数据集合和进程控制块三部分组成。程序用于描述进程要完成的功能，是控制进程执行的指令集；数据集合是程序在执行时所需要的数据和工作区；程序控制块(Program Control Block，简称PCB)，包含进程的描述信息和控制信息，是进程存在的唯一标志。

进程具有的特征：

* 动态性：进程是程序的一次执行过程，是临时的，有生命期的，是动态产生，动态消亡的；
* 并发性：任何进程都可以同其他进程一起并发执行；
* 独立性：进程是系统进行资源分配和调度的一个独立单位；
* 结构性：进程由程序、数据和进程控制块三部分组成。

## 线程

在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位。任务调度采用的是时间片轮转的抢占式调度方式，而进程是任务调度的最小单位，每个进程有各自独立的一块内存，使得各个进程之间内存地址相互隔离。

后来，随着计算机的发展，对CPU的要求越来越高，进程之间的切换开销较大，已经无法满足越来越复杂的程序的要求了。于是就发明了线程，线程是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本单位。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间(也就是所在进程的内存空间)。一个标准的线程由线程ID、当前指令指针(PC)、寄存器和堆栈组成。而进程由内存空间(代码、数据、进程空间、打开的文件)和一个或多个线程组成。

## 进程与线程的区别

前面讲了进程与线程，但可能你还觉得迷糊，感觉他们很类似。的确，进程与线程有着千丝万缕的关系，下面就让我们一起来理一理：

* 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；
* 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；
* 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段、数据集、堆等)及一些进程级的资源(如打开文件和信号)，某进程内的线程在其它进程不可见；
* 调度和切换：线程上下文切换比进程上下文切换要快得多。
* 线程与进程关系的示意图：

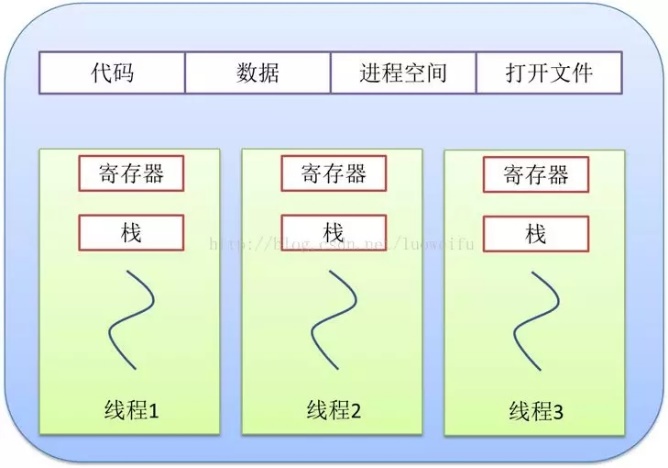


图 2：进程与线程的资源共享关系

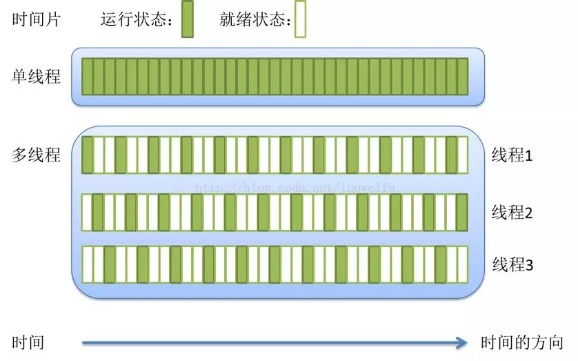


图 3：单线程与多线程的关系

总之，线程和进程都是一种抽象的概念，线程是一种比进程更小的抽象，线程和进程都可用于实现并发。

在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位。它相当于一个进程里只有一个线程，进程本身就是线程。所以线程有时被称为轻量级进程(Lightweight Process，LWP）。



图 4：早期的操作系统只有进程，没有线程

后来，随着计算机的发展，对多个任务之间上下文切换的效率要求越来越高，就抽象出一个更小的概念——线程，一般一个进程会有多个(也可是一个)线程。

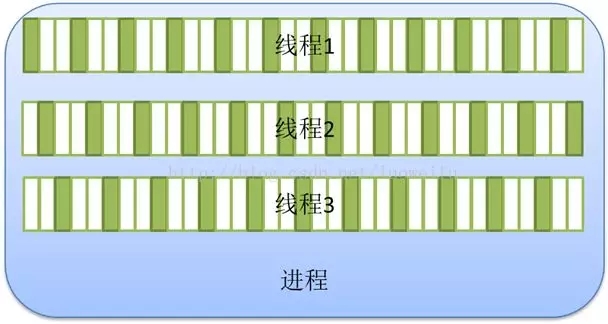


图 5：线程的出现，使得一个进程可以有多个线程

## 多线程与多核

上面提到的时间片轮转的调度方式说一个任务执行一小段时间后强制暂停去执行下一个任务，每个任务轮流执行。很多操作系统的书都说“同一时间点只有一个任务在执行”。那有人可能就要问双核处理器呢？难道两个核不是同时运行吗？

其实“同一时间点只有一个任务在执行”这句话是不准确的，至少它是不全面的。那多核处理器的情况下，线程是怎样执行呢？这就需要了解内核线程。

多核(心)处理器是指在一个处理器上集成多个运算核心从而提高计算能力，也就是有多个真正并行计算的处理核心，每一个处理核心对应一个内核线程。内核线程（Kernel Thread， KLT）就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换，内核通过操作调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。一般一个处理核心对应一个内核线程，比如单核处理器对应一个内核线程，双核处理器对应两个内核线程，四核处理器对应四个内核线程。

现在的电脑一般是双核四线程、四核八线程，是采用超线程技术将一个物理处理核心模拟成两个逻辑处理核心，对应两个内核线程，所以在操作系统中看到的CPU数量是实际物理CPU数量的两倍，如你的电脑是双核四线程，打开“任务管理器\性能”可以看到4个CPU的监视器，四核八线程可以看到8个CPU的监视器。

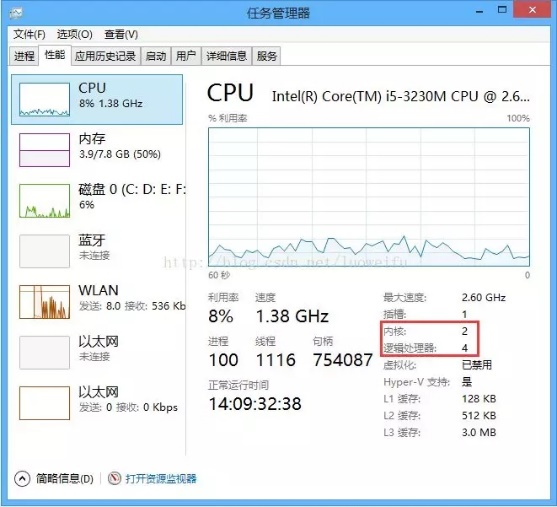


图 6：双核四线程在Windows8下查看的结果

超线程技术就是利用特殊的硬件指令，把一个物理芯片模拟成两个逻辑处理核心，让单个处理器都能使用线程级并行计算，进而兼容多线程操作系统和软件，减少了CPU的闲置时间，提高的CPU的运行效率。这种超线程技术(如双核四线程)由处理器硬件的决定，同时也需要操作系统的支持才能在计算机中表现出来。

程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口——轻量级进程（Light Weight Process，LWP），轻量级进程就是我们通常意义上所讲的线程(我们在这称它为用户线程)，由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。用户线程与内核线程的对应关系有三种模型：一对一模型、多对一模型、多对多模型，在这以4个内核线程、3个用户线程为例对三种模型进行说明。

### 一对一模型

对于一对一模型来说，一个用户线程就唯一地对应一个内核线程(反过来不一定成立，一个内核线程不一定有对应的用户线程)。这样，如果CPU没有采用超线程技术(如四核四线程的计算机)，一个用户线程就唯一地映射到一个物理CPU的线程，线程之间的并发是真正的并发。一对一模型使用户线程具有与内核线程一样的优点，一个线程因某种原因阻塞时其他线程的执行不受影响；此处，一对一模型也可以让多线程程序在多处理器的系统上有更好的表现。

但一对一模型也有两个缺点：1.许多操作系统限制了内核线程的数量，因此一对一模型会使用户线程的数量受到限制；2.许多操作系统内核线程调度时，上下文切换的开销较大，导致用户线程的执行效率下降。

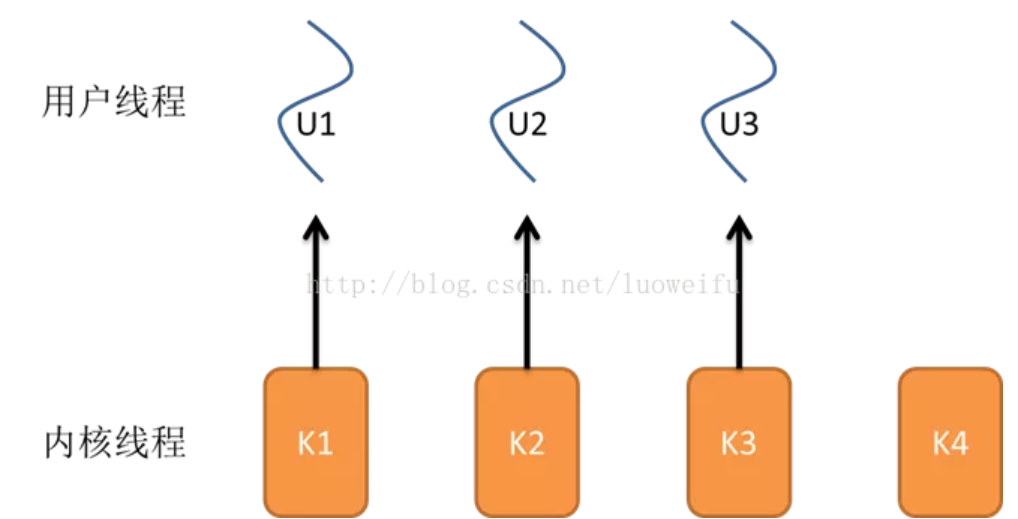


图 7：一对一模型

### 多对一模型

多对一模型将多个用户线程映射到一个内核线程上，线程之间的切换由用户态的代码来进行，因此相对一对一模型，多对一模型的线程切换速度要快许多；此外，多对一模型对用户线程的数量几乎无限制。但多对一模型也有两个缺点：1.如果其中一个用户线程阻塞，那么其它所有线程都将无法执行，因为此时内核线程也随之阻塞了；2.在多处理器系统上，处理器数量的增加对多对一模型的线程性能不会有明显的增加，因为所有的用户线程都映射到一个处理器上了。

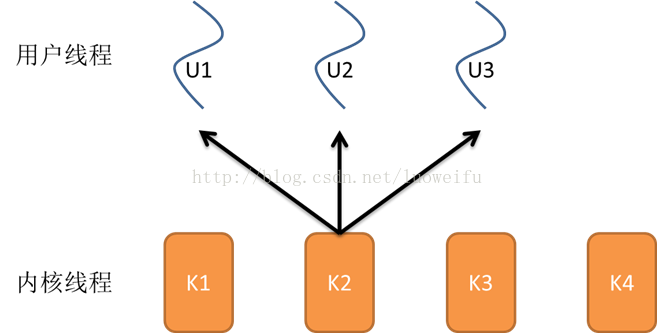


图 8：多对一模型

### 多对多模型

多对多模型结合了一对一模型和多对一模型的优点，将多个用户线程映射到多个内核线程上。多对多模型的优点有：1.一个用户线程的阻塞不会导致所有线程的阻塞，因为此时还有别的内核线程被调度来执行；2.多对多模型对用户线程的数量没有限制；3.在多处理器的操作系统中，多对多模型的线程也能得到一定的性能提升，但提升的幅度不如一对一模型的高。

在现在流行的操作系统中，大都采用多对多的模型。

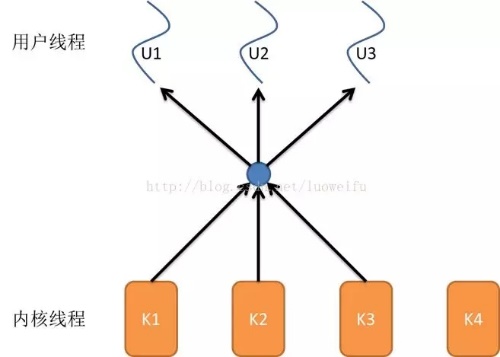


图 9：多对多模型

## 查看进程与线程

一个应用程序可能是多线程的，也可能是多进程的，如何查看呢？在Windows下我们只须打开任务管理器就能查看一个应用程序的进程和线程数。按“Ctrl+Alt+Del”或右键快捷工具栏打开任务管理器。

查看进程数和线程数：



图 10：查看线程数和进程数

在“进程”选项卡下，我们可以看到一个应用程序包含的线程数。如果一个应用程序有多个进程，我们能看到每一个进程，如在上图中，Google的chrome浏览器就有多个进程。同时，如果打开了一个应用程序的多个实例也会有多个进程，如上图中我打开了两个cmd窗口，就有两个cmd进程。如果看不到线程数这一列，可以在点击“查看\选择列”菜单，增加监听的列。

查看CPU和内存的使用率：

在性能选项卡中，我们可以查看CPU和内存的使用率，根据CPU使用记录的监视器的个数还能看出逻辑处理核心的个数，如我的双核四线程的计算机就有四个监视器。

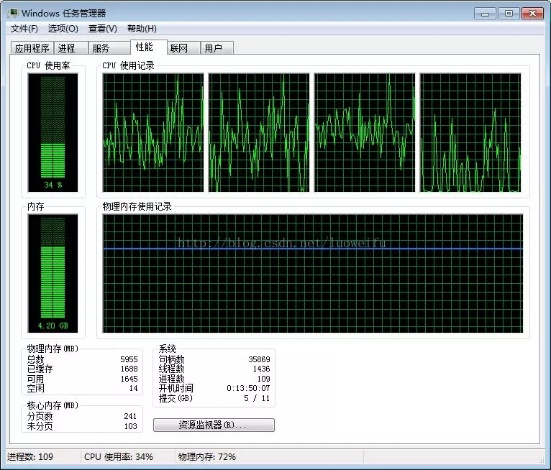


图 11：查看CPU和内存的使用率

## 线程的生命周期

当线程的数量小于处理器的数量时，线程的并发是真正的并发，不同的线程运行在不同的处理器上。但当线程的数量大于处理器的数量时，线程的并发会受到一些阻碍，此时并不是真正的并发，因为此时至少有一个处理器会运行多个线程。

在单个处理器运行多个线程时，并发是一种模拟出来的状态。操作系统采用时间片轮转的方式轮流执行每一个线程。现在，几乎所有的现代操作系统采用的都是时间片轮转的抢占式调度方式，如我们熟悉的Unix、Linux、Windows及Mac OS X等流行的操作系统。

我们知道线程是程序执行的最小单位，也是任务执行的最小单位。在早期只有进程的操作系统中，进程有五种状态，创建、就绪、运行、阻塞(等待)、退出。早期的进程相当于现在的只有单个线程的进程，那么现在的多线程也有五种状态，现在的多线程的生命周期与早期进程的生命周期类似。

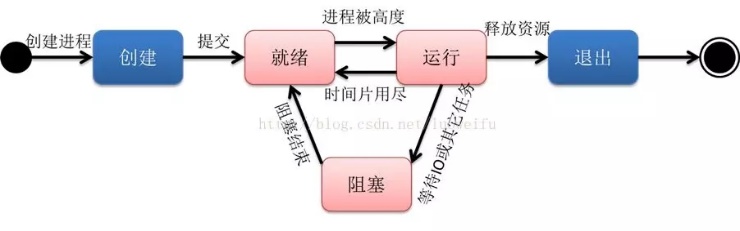


图 12：早期进程的生命周期

进程在运行过程有三种状态：就绪、运行、阻塞，创建和退出状态描述的是进程的创建过程和退出过程。

* 创建：进程正在创建，还不能运行。操作系统在创建进程时要进行的工作包括分配和建立进程控制块表项、建立资源表格并分配资源、加载程序并建立地址空间；
* 就绪：时间片已用完，此线程被强制暂停，等待下一个属于他的时间片到来；
* 运行：此线程正在执行，正在占用时间片；
* 阻塞：也叫等待状态，等待某一事件(如IO或另一个线程)执行完；
* 退出：进程已结束，所以也称结束状态，释放操作系统分配的资源。

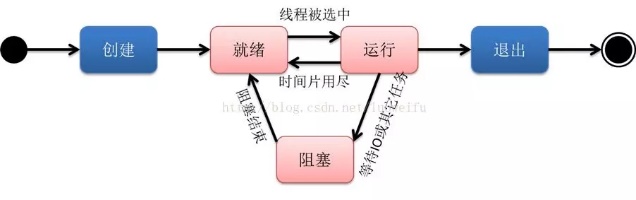


图 13：线程的生命周期

创建：一个新的线程被创建，等待该线程被调用执行；

就绪：时间片已用完，此线程被强制暂停，等待下一个属于他的时间片到来；

运行：此线程正在执行，正在占用时间片；

阻塞：也叫等待状态，等待某一事件(如IO或另一个线程)执行完；

退出：一个线程完成任务或者其他终止条件发生，该线程终止进入退出状态，退出状态释放该线程所分配的资源。

# 一个故事讲完进程、线程和协程

很久以前，有两个程序，暂且称他们旺财和小强吧。

旺财和小强这两个程序都很长，每个都有十几万行。 他们两个的人生价值就是到CPU上去运行，把运行结果告诉人类。

CPU是稀缺资源，只有一个，他们俩必须排着队，轮流使用。

旺财从头到尾执行完了，让出CPU， 让小强从头儿去执行。

人类把这种处理方式叫做**批处理**。



## 进程

长久以来，两人相安无事。 后来CPU的速度越来越快， 远远超过了内存，硬盘的速度。

人类想到，这批处理系统的效率有点低啊，你看当小强需要从硬盘上读取数据的时候，CPU也一直在等待，这是多大的浪费啊！这时候完全可以让旺财来运行一下嘛！

当然得保存好小强的**执行现场**：具体执行到那一行程序指令了， 函数调用到什么层次了，每个函数调用都有什么样的参数，CPU寄存器中的值..... 等等一系列东西。

如果不把小强的执行现场给保存下来，等到小强的数据从银盘读完了，就没法回到中断处来继续执行了。

这个执行现场，再加上小强的代码，就是一个执行中的程序，被称为“**进程**” 。

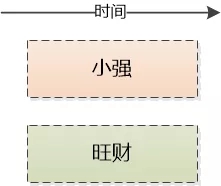
旺财和小强在运行的时候，也被改造成了进程。

人类还规定：进程不能长时间占据CPU， 只能在CPU上执行一小会儿，然后马上切换到别的进程去执行。

旺财和小强不以为意：不就是执行一会儿，歇一会儿，然后继续执行嘛！



但是他们不知道的是，由于CPU运行速度超快，旺财和小强虽然在不断地切换运行，在人类那缓慢的世界里看来，旺财和小强好像是同时在执行一样。  这就是**并发**。



（在人类看来，小强和旺财似乎是在同时执行）

多年以后，他们俩才真正地实现了并行： 在一个豪华电脑中，每人都被分配了一个CPU ， 真正地同时执行， 这是后话了。

## 线程

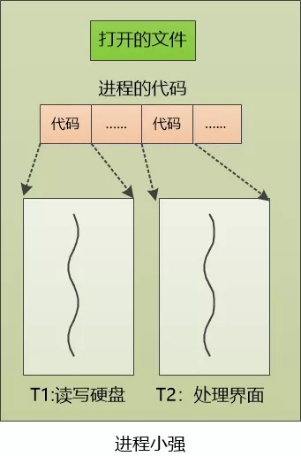
这时候旺财已经有了界面，还能访问网络，每当它联网的时候（这也是个非常非常耗时的操作），就得把CPU让给小强。

即使旺财再次被调度执行，由于网络数据还没有返回，他必须等待，什么事情都做不了，在人类看来，界面根本无法操作，旺财不响应了！  气得人类经常把旺财kill掉。

旺财心里苦，他很纳闷小强怎么就没有问题，小强不是要读写硬盘吗？ 那也是很慢的操作啊。

小强说：“你傻啊，内部只有一个执行的流程，一遇到耗时的操作就得等待，你看看我，内部搞了两个执行流程（**线程**），一个用来读写硬盘（T1)，另外一个处理界面(T2)。我和操作系统商量好了，如果T1在读写硬盘， 就可以调度我的T2来执行，这样界面至少还可以操作。 ”

旺财觉得很有意思，也采用了类似办法。



于是，一个进程中至少有一个执行的流程（主线程），也可以开启新的执行流程（线程）。

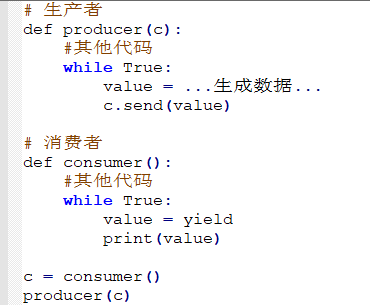
线程变成了最小的调度单位。

## 协程

这一天，旺财被一个叫做生产者和消费者的问题折腾地死去活来，两个线程，一个线程向队列中放数据，另外一个从队列中取数据，处理起两个线程的协作就显得很麻烦，不但需要加锁，还得做好线程的通知和等待。

正在感慨多线程编程之难的时候， 旺财震惊地发现，小强用了一个极为简单的办法把生产者，消费者问题给解决了。

这个方法的代码如下：

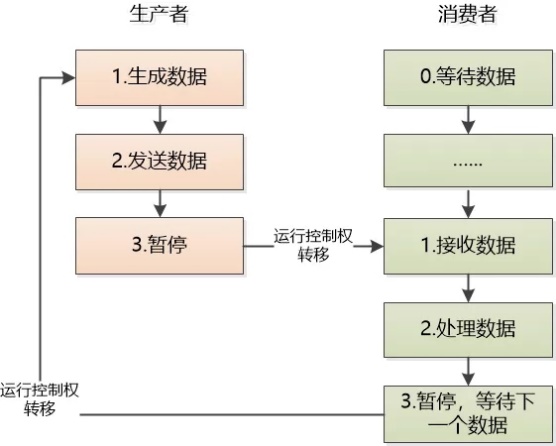


“这....这怎么执行啊，那个yield是怎么回事？”  旺财表示不解。  
“简单啊，你看那个生产者，是不是向消费者发送了数据？ ” 小强说。

“对啊，然后呢，生产者发送了数据以后，会马上进行下一轮循环吗？”

“这就是关键所在了，”小强说，“ 它们是这么执行的：”

1. 生产者发送数据，暂停运行，不进行下一轮循环
2. 消费者其实一直在value = yield 那里等待，直到数据到来，现在数据来了，取出处理（value就是生产者发送过来的数据）。
3. 消费者在循环中再次yield， 暂停执行。
4. 生产者继续下一轮的循环，生成新的消息，发送给消费者。



旺财觉得很吃惊，小强竟然**可以让一个正在执行的程序暂停**，他不由得问道：“你这个暂停是真的停止了了，还是说只是像Java的yield那样，让出CPU进入了就绪状态？ 等待下次调度运行？”

“是真的暂停了，程序就停在那里，等待运行控制权从对方那里转移过来。”

“这不是操作系统干的事情吗？ ” 旺财更加吃惊了。

“正是这样，” 小强得意地说：“我打算把类似生产者，消费者这样的代码称为‘**协程**’， 这个协程有个重要的特点，就是完全被我所调度和掌控， 不用操作系统介入。”

“这个协程和线程似乎很像啊。每次协程停止执行的时候，也得保存现场，要不然没法恢复执行。” 旺财说。

“是啊，只是他们**比线程更加轻量级**，操作系统内核不用参与，相当于用户态线程了，**协程的开销极小**，可以轻松地创建大量的协程来做事情。 对了，也许你注意到了，我这两个协程是'**合作式**'的，它们两个同一时刻只能有一个在运行。 实际上，我在底层可以用一个线程去执行这两个协程。  ”

旺财表示同意：“不错，既然两个程序可以'合作'，那就不用加锁了，也不用在代码里写什么wait和notify了，在程序层面，可以用同步的方式实现异步的功能了！ 代码很清晰，我也搞个协程来玩玩吧！”