并发面试必备系列之进程、线程与协程

在《Awesome Interviews》归纳的常见面试题中，无论前后端，并发与异步的相关知识都是面试的中重中之重，《并发编程》系列即对于面试中常见的并发知识再进行回顾总结；你也可以前往《Awesome Interviews》，在实际的面试题考校中了解自己的掌握程度。也可以前往《Java 实战》、《Go 实战》等了解具体编程语言中的并发编程的相关知识。

在未配置 OS 的系统中，程序的执行方式是顺序执行，即必须在一个程序执行完后，才允许另一个程序执行；在多道程序环境下，则允许多个程序并发执行。程序的这两种执行方式间有着显著的不同。也正是程序并发执行时的这种特征，才导致了在操作系统中引入进程的概念。**进程是资源分配的基本单位，线程是资源调度的基本单位。**

应用启动体现的就是静态指令加载进内存，进而进入 CPU 运算，操作系统在内存开辟了一段栈内存用来存放指令和变量值，从而形成了进程。早期的操作系统基于进程来调度 CPU，不同进程间是不共享内存空间的，所以进程要做任务切换就要切换内存映射地址。由于进程的上下文关联的变量，引用，计数器等现场数据占用了打段的内存空间，所以频繁切换进程需要整理一大段内存空间来保存未执行完的进程现场，等下次轮到 CPU 时间片再恢复现场进行运算。

这样既耗费时间又浪费空间，所以我们才要研究多线程。一个进程创建的所有线程，都是共享一个内存空间的，所以线程做任务切换成本就很低了。现代的操作系统都基于更轻量的线程来调度，现在我们提到的“任务切换”都是指“线程切换”。

进程与线程

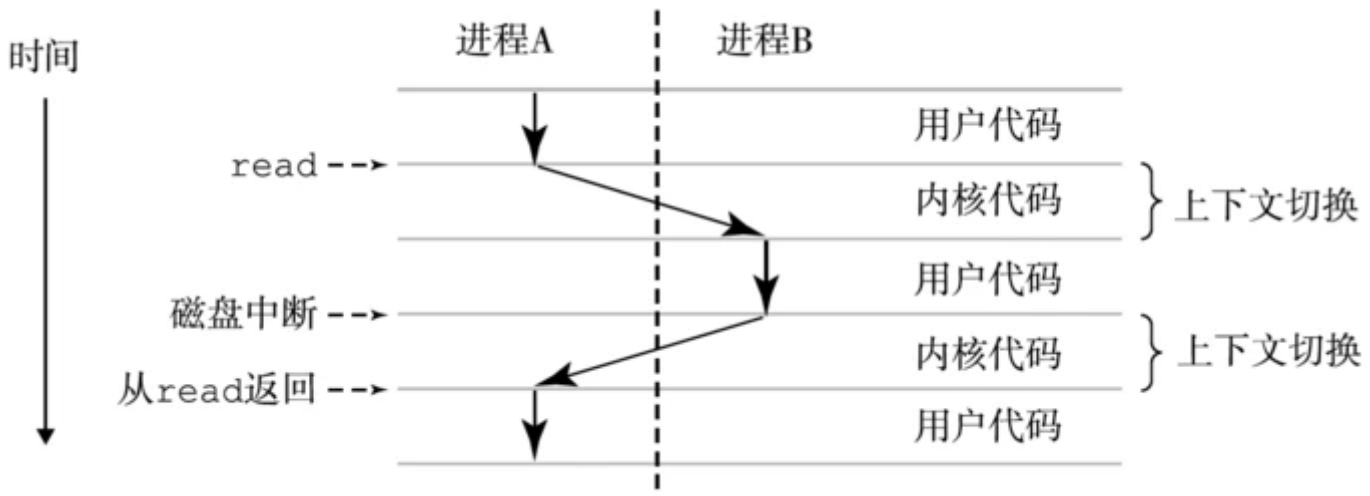
> 本部分节选自《Linux 与操作系统/进程管理》。

在未配置 OS 的系统中，程序的执行方式是顺序执行，即必须在一个程序执行完后，才允许另一个程序执行；在多道程序环境下，则允许多个程序并发执行。程序的这两种执行方式间有着显著的不同。也正是程序并发执行时的这种特征，才导致了在操作系统中引入进程的概念。**进程是资源分配的基本单位，线程是资源调度的基本单位。**

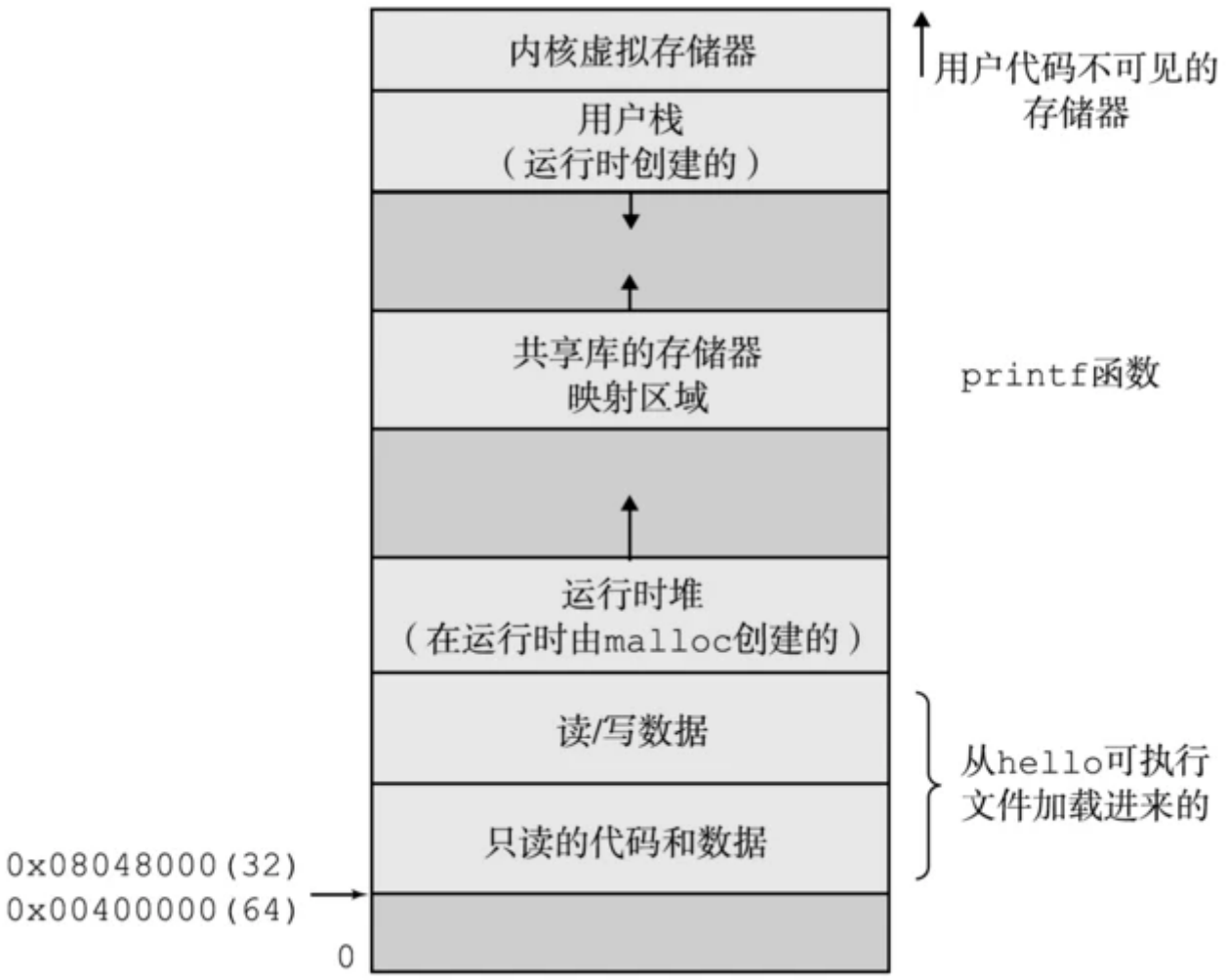
进程（Process）

进程是操作系统对一个正在运行的程序的一种抽象，在一个系统上可以同时运行多个进程，而每个进程都好像在独占地使用硬件。所谓的并发运行，则是说一个进程的指令和另一个进程的指令是交错执行的。无论是在单核还是多核系统中，可以通过处理器在进程间切换，来实现单个 CPU 看上去像是在并发地执行多个进程。操作系统实现这种交错执行的机制称为上下文切换。

操作系统保持跟踪进程运行所需的所有状态信息。这种状态，也就是上下文，它包括许多信息，例如 PC 和寄存器文件的当前值，以及主存的内容。在任何一个时刻，单处理器系统都只能执行一个进程的代码。当操作系统决定要把控制权从当前进程转移到某个新进程时，就会进行上下文切换，即保存当前进程的上下文、恢复新进程的上下文，然后将控制权传递到新进程。新进程就会从上次停止的地方开始。



在虚拟存储管理一节中，我们介绍过它为每个进程提供了一个假象，即每个进程都在独占地使用主存。每个进程看到的是一致的存储器，称为虚拟地址空间。其虚拟地址空间最上面的区域是为操作系统中的代码和数据保留的，这对所有进程来说都是一样的；地址空间的底部区域存放用户进程定义的代码和数据。



- 程序代码和数据，对于所有的进程来说，代码是从同一固定地址开始，直接按照可执行目标文件的内容初始化。
- 堆，代码和数据区后紧随着的是运行时堆。代码和数据区是在进程一开始运行时就被规定了大小，与此不同，当调用如 malloc 和 free 这样的 C 标准库函数时，堆可以在运行时动态地扩展和收缩。
- 共享库：大约在地址空间的中间部分是一块用来存放像 C 标准库和数学库这样共享库的代码和数据的区域。
- 栈，位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用。和堆一样，用户栈在程序执行期间可以动态地扩展和收缩。
- 内核虚拟存储器：内核总是驻留在内存中，是操作系统的一部分。地址空间顶部的区域是为内核保留的，不允许应用程序读写这个区域的内容或者直接调用内核代码定义的函数。

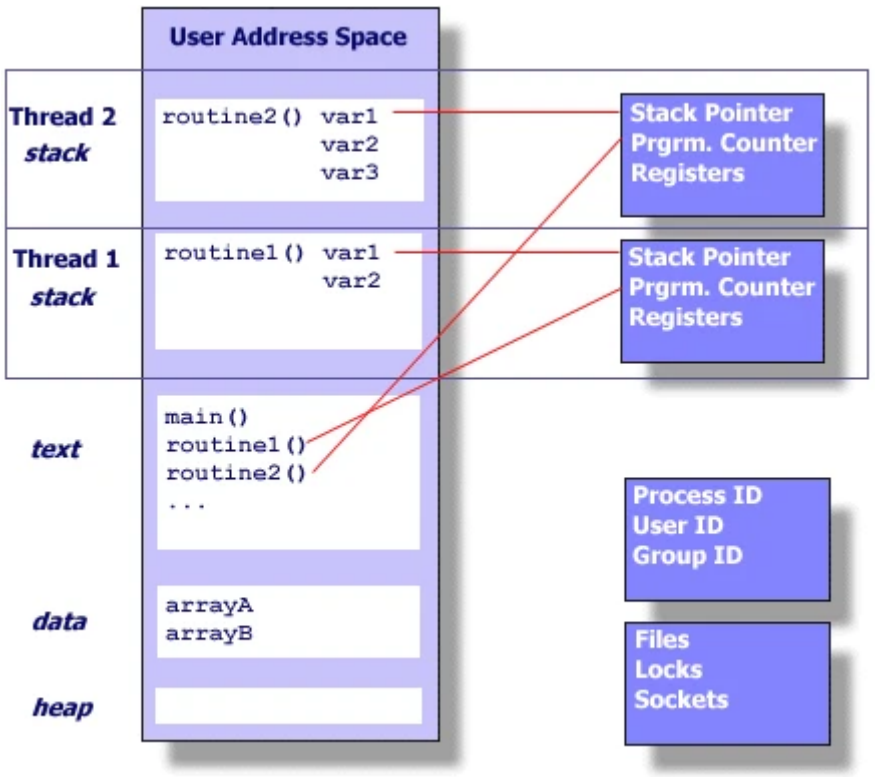
线程（Thread）

在现代系统中，一个进程实际上可以由多个称为线程的执行单元组成，每个线程都运行在进程的上下文中，并共享同样的代码和全局数据。进程的个体间是完全独立的，而线程间是彼此依存的。多进程环境中，任何一个进程的终止，不会影响到其他进程。而多线程环境中，父线程终止，全部子线程被迫终止(没有了资源)。

而任何一个子线程终止一般不会影响其他线程，除非子线程执行了 `exit()` 系统调用。任何一个子线程执行 `exit()`，全部线程同时灭亡。多线程程序中至少有一个主线程，而这个主线程其实就是有 main 函数的进程。它是整个程序的进程，所有线程都是它的子线程；我们通常把具有多线程的主进程称之为**主线程**。

线程共享的环境包括：进程代码段、进程的公有数据、进程打开的文件描述符、信号的处理器、进程的当前目录、进程用户 ID 与进程组 ID 等，利用这些共享的数据，线程很容易的实现相互之间的通讯。线程拥有这许多共性的同时，还拥有自己的个性，并以此实现并发性：

- 线程 ID：每个线程都有自己的线程 ID，这个 ID 在本进程中是唯一的。进程用此来标识线程。
- 寄存器组的值：由于线程间是并发运行的，每个线程有自己不同的运行线索，当从一个线程切换到另一个线程上时，必须将原有的线程的寄存器集合的状态保存，以便 将来该线程在被重新切换到时能得以恢复。
- 线程的堆栈：堆栈是保证线程独立运行所必须的。线程函数可以调用函数，而被调用函数中又是可以层层嵌套的，所以线程必须拥有自己的函数堆栈， 使得函数调用可以正常执行，不受其他线程的影响。
- 错误返回码：由于同一个进程中有很多个线程在同时运行，可能某个线程进行系统调用后设置了 errno 值，而在该 线程还没有处理这个错误，另外一个线程就在此时 被调度器投入运行，这样错误值就有可能被修改。 所以，不同的线程应该拥有自己的错误返回码变量。
- 线程的信号屏蔽码：由于每个线程所感兴趣的信号不同，所以线程的信号屏蔽码应该由线程自己管理。但所有的线程都共享同样的信号处理器。
- 线程的优先级：由于线程需要像进程那样能够被调度，那么就必须要有可供调度使用的参数，这个参数就是线程的优先级。

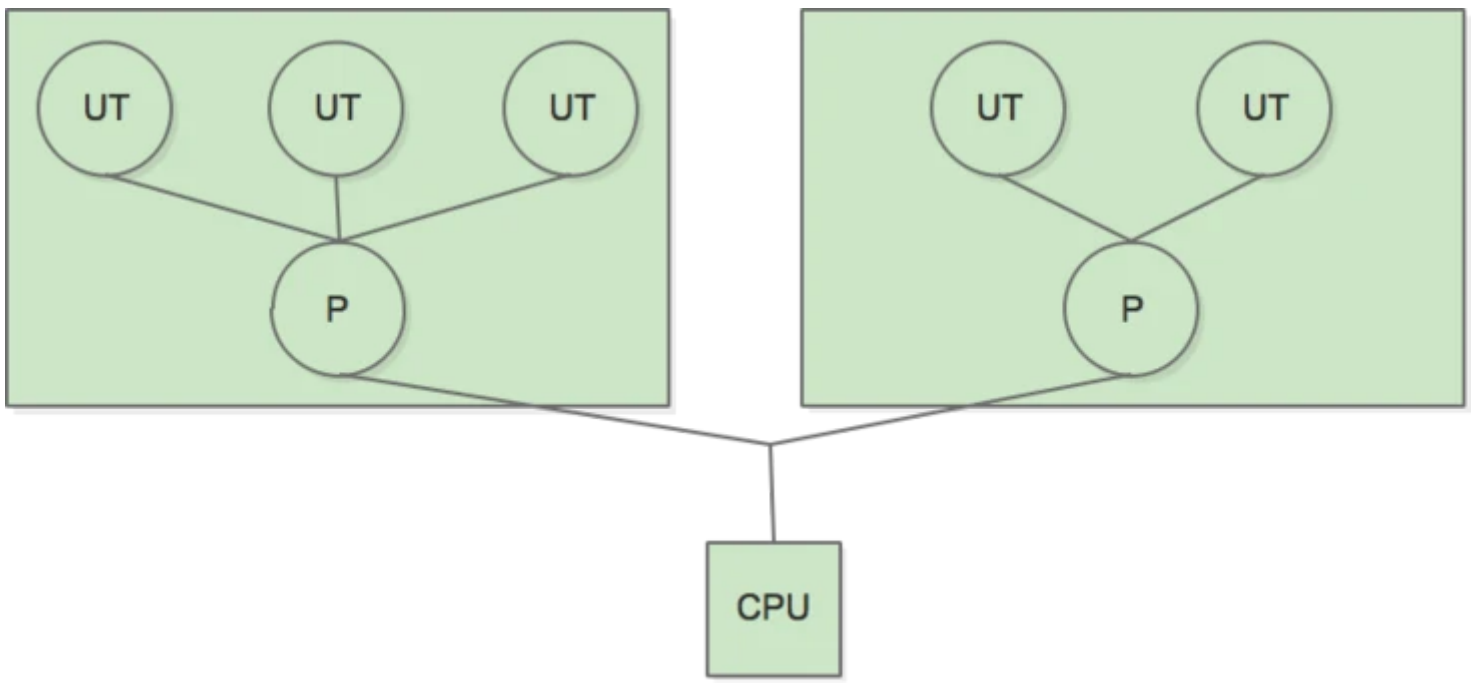


线程模型

线程实现在用户空间下

当线程在用户空间下实现时，操作系统对线程的存在一无所知，操作系统只能看到进程，而不能看到线程。所有的线程都是在用户空间实现。在操作系统看来，每一个进程只有一个线程。过去的操作系统大部分是这种实现方式，这种方式的好处之一就是即使操作系统不支持线程，也可以通过库函数来支持线程。

在这在模型下，程序员需要自己实现线程的数据结构、创建销毁和调度维护。也就相当于需要实现一个自己的线程调度内核，而同时这些线程运行在操作系统的一个进程内，最后操作系统直接对进程进行调度。



这样做有一些优点，首先就是确实在操作系统中实现了真实的多线程，其次就是线程的调度只是在用户态，减少了操作系统从内核态到用户态的切换开销。这种模式最致命的缺点也是由于操作系统不知道线程的存在，因此当一个进程中的某一个线程进行系统调用时，比如缺页中断而导致线程阻塞，此时操作系统会阻塞整个进程，即使这个进程中其它线程还在工作。还有一个问题是假如进程中一个线程长时间不释放 CPU，因为用户空间并没有时钟中断机制，会导致此进程中的其它线程得不到 CPU 而持续等待。

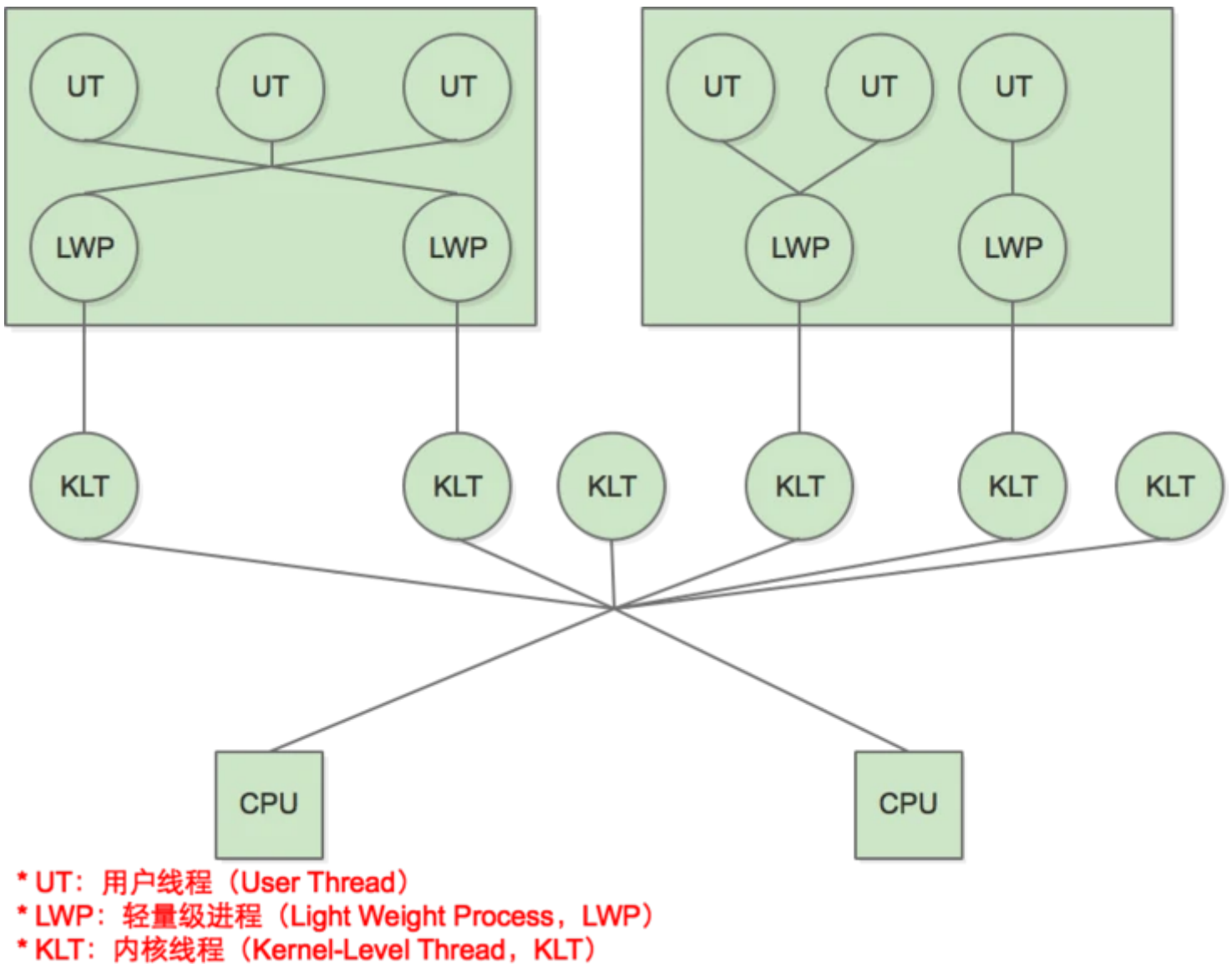
线程实现在操作系统内核中

内核线程就是直接由操作系统内核（Kernel）支持的线程，这种线程由内核来完成线程切换，内核通过操纵调度器（Scheduler）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。每个内核线程可以视为内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内核就叫做多线程内核（Multi-Threads Kernel）。

程序员直接使用操作系统中已经实现的线程，而线程的创建、销毁、调度和维护，都是靠操作系统（准确的说是内核）来实现，程序员只需要使用系统调用，而不需要自己设计线程的调度算法和线程对 CPU 资源的抢占使用。

使用用户线程加轻量级进程混合实现

在这种混合实现下，即存在用户线程，也存在轻量级进程。用户线程还是完全建立在用户空间中，因此用户线程的创建、切换、析构等操作依然廉价，并且可以支持大规模的用户线程并发。而操作系统提供支持的轻量级进程则作为用户线程和内核线程之间的桥梁，这样可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过轻量级进程来完成，大大降低了整个进程被完全阻塞的风险。在这种混合模式中，用户线程与轻量级进程的数量比是不定的，即为 N:M 的关系：

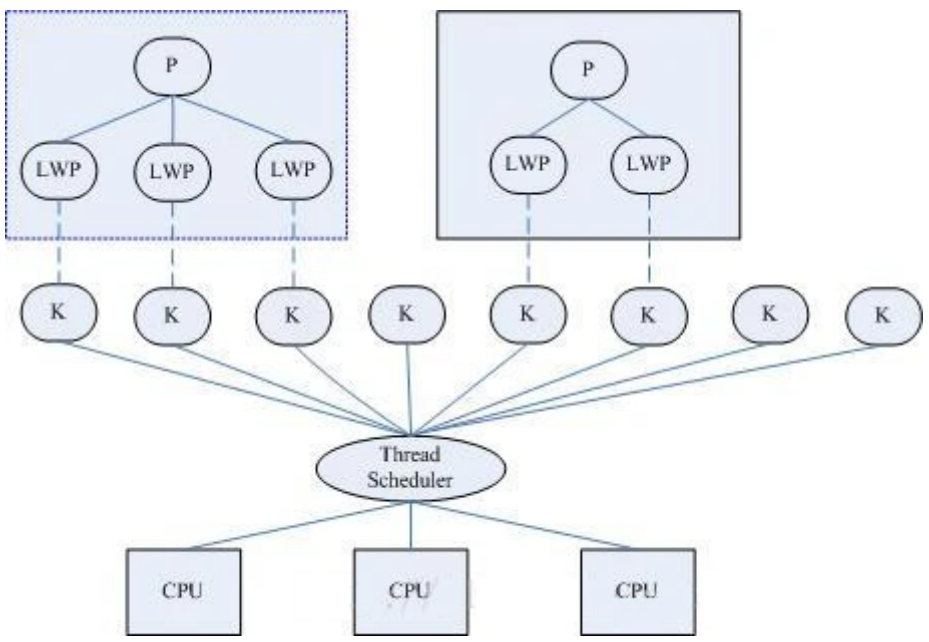


Golang 的协程就是使用了这种模型，在用户态，协程能快速的切换，避免了线程调度的 CPU 开销问题，协程相当于线程的线程。

Linux 中的线程

在 Linux 2.4 版以前，线程的实现和管理方式就是完全按照进程方式实现的；在 Linux 2.6 之前，内核并不支持线程的概念，仅通过轻量级进程（Lightweight Process）模拟线程；轻量级进程是建立在内核之上并由内核支持的用户线程，它是内核线程的高度抽象，每一个轻量级进程都与一个特定的内核线程关联。内核线程只能由内核管理并像普通进程一样被调度。这种模型最大的特点是线程调度由内核完成了，而其他线程操作（同步、取消）等都是核外的线程库（Linux Thread）函数完成的。

为了完全兼容 Posix 标准，Linux 2.6 首先对内核进行了改进，引入了线程组的概念（**仍然用轻量级进程表示线程**），有了这个概念就可以将一组线程组织称为一个进程，不过内核并没有准备特别的调度算法或是定义特别的数据结构来表征线程；相反，线程仅仅被视为一个与其他进程（概念上应该是线程）共享某些资源的进程（概念上应该是线程）。在实现上主要的改变就是在 task_struct 中加入 tgid 字段，这个字段就是用于表示线程组 id 的字段。在用户线程库方面，也使用 NPTL 代替 Linux Thread，不同调度模型上仍然采用 `1 对 1` 模型。



进程的实现是调用 fork 系统调用：`pid_t fork(void);`，线程的实现是调用 clone 系统调用：`int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...)`。与标准 `fork()` 相比，线程带来的开销非常小，内核无需单独复制进程的内存空间或文件描写叙述符等等。这就节省了大量的 CPU 时间，使得线程创建比新进程创建快上十到一百倍，能够大量使用线程而无需太过于操

心带来的 CPU 或内存不足。无论是 fork、vfork、kthread_create 最后都是要调用 do_fork，而 do_fork 就是根据不同的函数参数，对一个进程所需的资源进行分配。

内核线程

内核线程是由内核自己创建的线程，也叫做守护线程（Deamon），在终端上用命令 `ps -Al` 列出的所有进程中，名字以 k 开关以 d 结尾的往往都是内核线程，比如 kthreadd、kswapd 等。与用户线程相比，它们都由 `do_fork()` 创建，每个线程都有独立的 task_struct 和内核栈；也都参与调度，内核线程也有优先级，会被调度器平等地换入换出。二者的不同之处在于，内核线程只工作在内核态中；而用户线程则既可以运行在内核态（执行系统调用时），也可以运行在用户态；内核线程没有用户空间，所以对于一个内核线程来说，它的 0~3G 的内存空间是空白的，它的 `current->mm` 是空的，与内核使用同一张页表；而用户线程则可以看到完整的 0~4G 内存空间。

在 Linux 内核启动的最后阶段，系统会创建两个内核线程，一个是 init，一个是 kthreadd。其中 init 线程的作用是运行文件系统上的一系列"init"脚本，并启动 shell 进程，所以 init 线程称得上是系统中所有用户进程的祖先，它的 pid 是 1。kthreadd 线程是内核的守护线程，在内核正常工作时，它永远不退出，是一个死循环，它的 pid 是 2。

Coroutine | 协程

协程是用户模式下的轻量级线程，最准确的名字应该叫用户空间线程（User Space Thread），在不同的领域中也有不同的叫法，譬如纤程(Fiber)、绿色线程(Green Thread)等等。操作系统内核对协程一无所知，协程的调度完全有应用程序来控制，操作系统不管这部分的调度；一个线程可以包含一个或多个协程，协程拥有自己的寄存器上下文和栈，协程调度切换时，将寄存器上细纹和栈保存起来，在切换回来时恢复先前保运的寄存上下文和栈。

协程的优势如下：

- 节省内存，每个线程需要分配一段栈内存，以及内核里的一些资源
- 节省分配线程的开销（创建和销毁线程要各做一次 syscall）
- 节省大量线程切换带来的开销
- 与 NIO 配合实现非阻塞的编程，提高系统的吞吐

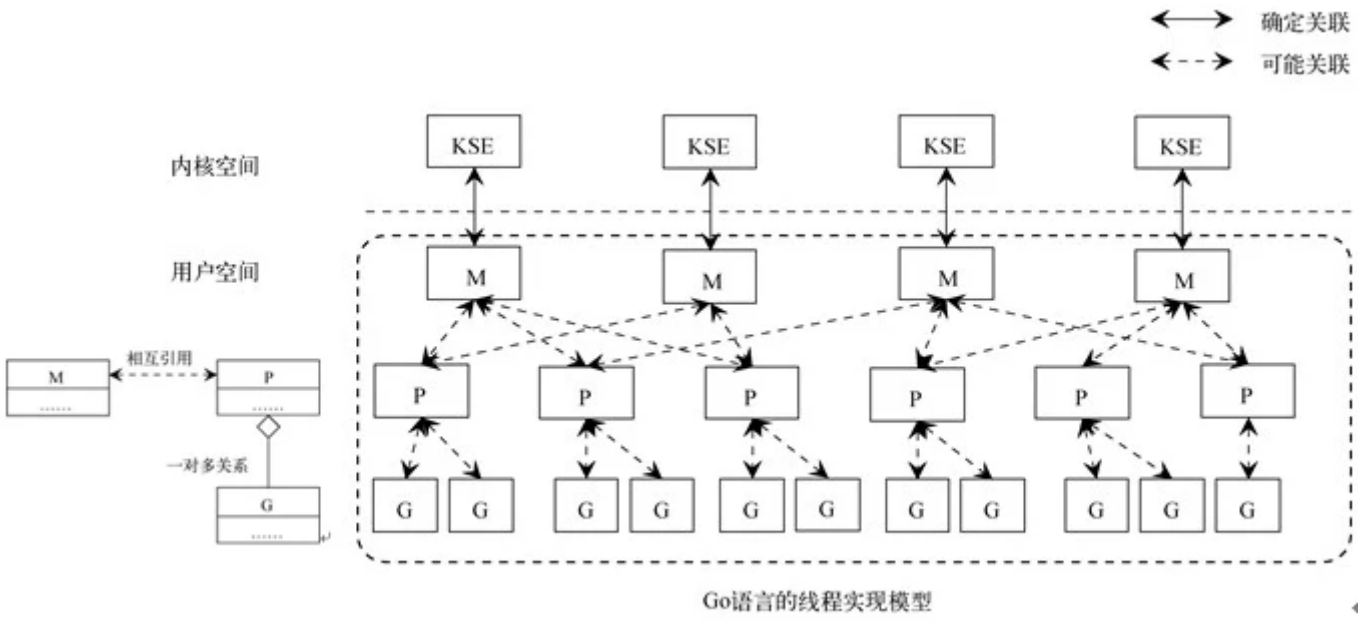
比如 Golang 里的 go 关键字其实就是负责开启一个 Fiber，让 func 逻辑跑在上面。而这一切都是发生的用户态上，没有发生在内核态上，也就是说没有 ContextSwitch 上的开销。协程的实现库中笔者较为常用的譬如 Go Routine、node-fibers、Java-Quasar 等。

Go 的协程模型

Go 线程模型属于多对多线程模型，在操作系统提供的内核线程之上，Go 搭建了一个特有的两级线程模型。Go 中使用使用 Go 语句创建的 Goroutine 可以认为是轻量级的用户线程，Go 线程模型包含三个概念：

- G: 表示 Goroutine，每个 Goroutine 对应一个 G 结构体，G 存储 Goroutine 的运行堆栈、状态以及任务函数，可重用。G 并非执行体，每个 G 需要绑定到 P 才能被调度执行。
- P: Processor，表示逻辑处理器，对 G 来说，P 相当于 CPU 核，G 只有绑定到 P(在 P 的 local runq 中)才能被调度。对 M 来说，P 提供了相关的执行环境（Context），如内存分配状态（mcache），任务队列（G）等，P 的数量决定了系统内最大可并行的 G 的数量（物理 CPU 核数 >= P 的数量），P 的数量由用户设置的 GOMAXPROCS 决定，但是不论 GOMAXPROCS 设置为多大，P 的数量最大为 256。
- M: Machine，OS 线程抽象，代表着真正执行计算的资源，在绑定有效的 P 后，进入 schedule 循环；M 的数量是不定的，由 Go Runtime 调整，为了防止创建过多 OS 线程导致系统调度不过来，目前默认最大限制为 10000 个。

在 Go 中每个逻辑处理器(P)会绑定到某一个内核线程上，每个逻辑处理器（P）内有一个本地队列，用来存放 Go 运行时分配的 goroutine。多对多线程模型中是操作系统调度线程在物理 CPU 上运行，在 Go 中则是 Go 的运行时调度 Goroutine 在逻辑处理器（P）上运行。



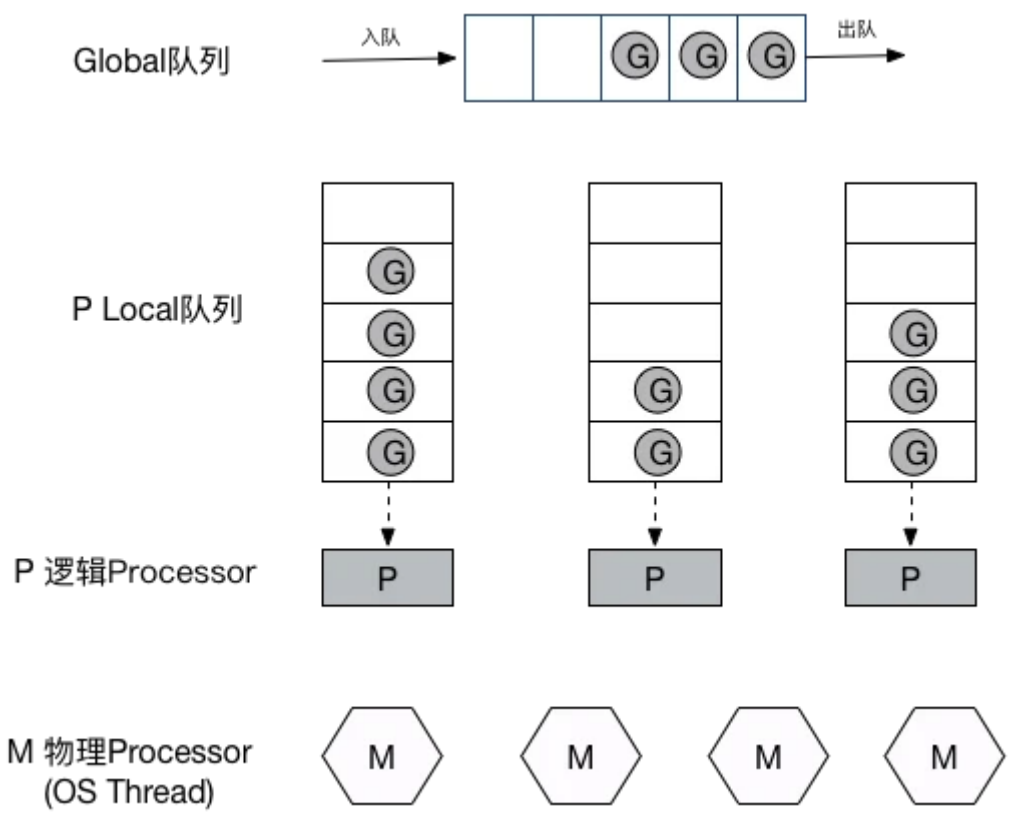
Go语言的线程实现模型

Go 的栈是动态分配大小的，随着存储数据的数量而增长和收缩。每个新建的 Goroutine 只有大约 4KB 的栈。每个栈只有 4KB，那么在一个 1GB 的 RAM 上，我们就可以有 256 万个 Goroutine 了，相对于 Java 中每个线程的 1MB，这是巨大的提升。Golang 实现了自己的调度器，允许众多的 Goroutines 运行在相同的 OS 线程上。就算 Go 会运行与内核相同的上下文切换，但是它能够避免切换至 ring-0 以运行内核，然后再切换回来，这样就会节省大量的时间。

在 Go 中存在两级调度:

- 一级是操作系统的调度系统，该调度系统调度逻辑处理器占用 cpu 时间片运行；
- 一级是 Go 的运行时调度系统，该调度系统调度某个 Goroutine 在逻辑处理上运行。

使用 Go 语句创建一个 Goroutine 后，创建的 Goroutine 会被放入 Go 运行时调度器的全局运行队列中，然后 Go 运行时调度器会把全局队列中的 Goroutine 分配给不同的逻辑处理器（P），分配的 Goroutine 会被放到逻辑处理器（P)的本地队列中，当本地队列中某个 Goroutine 就绪后待分配到时间片后就可以在逻辑处理器上运行了。



Java 协程的讨论

目前，JVM 本身并未提供协程的实现库，像 Quasar 这样的协程框架似乎也仍非主流的并发问题解决方案，在本部分我们就讨论下在 Java 中是否有必要一定要引入协程。在普通的 Web 服务器场景下，譬如 Spring Boot 中默认的 Worker 线程池线程数在 200（50 ~ 500） 左右，如果从线程的内存占用角度来考虑，每个线程上下文约 128KB，那么 500 个线程本身的内存占用在 60M，相较于整个堆栈不过尔尔。而 Java 本身提供的线程池，对于线程的创建与销毁都有非常好的支持；即使 Vert.x 或 Kotlin 中提供的协程，往往也是基于原生线程池实现的。

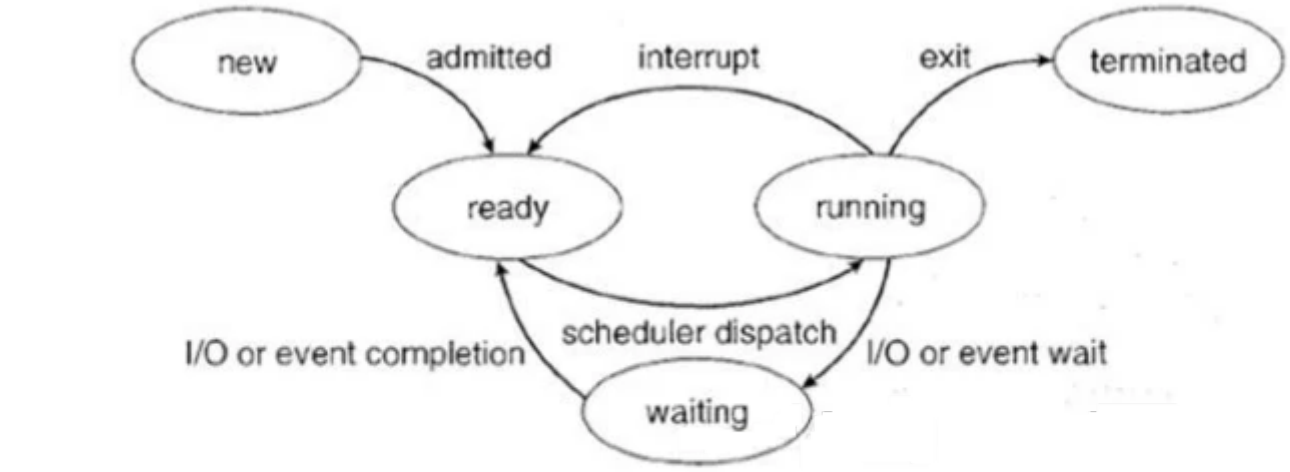
从线程的切换开销的角度来看，我们常说的切换开销往往是针对于活跃线程；而普通的 Web 服务器天然会有大量的线程因为请求读写、DB 读写这样的操作而挂起，实际只有数十个并发活跃线程会参与到 OS 的线程切换调度。而如果真的存在着大量活跃线程的场景，Java 生态圈中也存在了 Akka 这样的 Actor 并发模型框架，它能够感知线程何时能够执行工作，在用户空间中构建运行时调度器，从而支持百万级别的 Actor 并发。

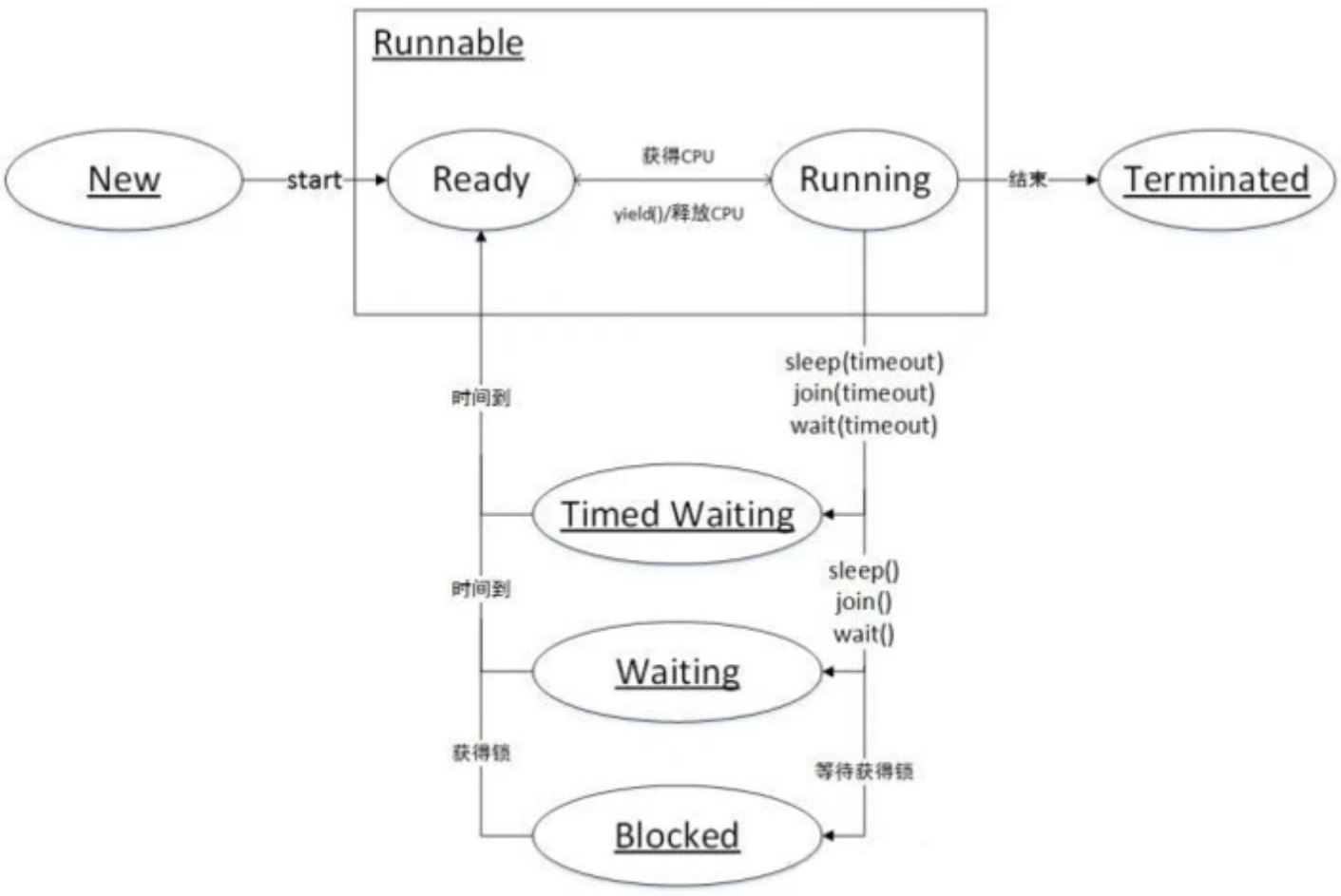
实际上我们引入协程的场景，更多的是面对所谓百万级别连接的处理，典型的就是 IM 服务器，可能需要同时处理大量空闲的链接。此时在 Java 生态圈中，我们可以使用 Netty 去进行处理，其基于 NIO 与 Worker Thread 实现的调度机制就很类似于协程，可以解决绝大部分因为 IO 的等待造成资源浪费的问题。而从并发模型对比的角度，如果我们希望能遵循 Go 中以消息传递方式实现内存共享的理念，那么也可以采用 Disruptor 这样的模型。

Java 线程与操作系统线程

Java 线程在 JDK1.2 之前，是基于称为“绿色线程”（Green Threads）的用户线程实现的，而到了 JDK1.2 及以后，JVM 选择了更加稳健且方便使用的操作系统原生的线程模型，通过系统调用，将程序的线程交给了操作系统内核进行调度。因此，在目前的 JDK 版本中，操作系统支持怎样的线程模型，在很大程度上决定了 Java 虚拟机的线程是怎样映射的，这点在不同的平台上没有办法达成一致，虚拟机规范中也并未限定 Java 线程需要使用哪种线程模型来实现。线程模型只对线程的并发规模和操作成本产生影响，对 Java 程序的编码和运行过程来说，这些差异都是透明的。

对于 Sun JDK 来说，它的 Windows 版与 Linux 版都是使用一对一的线程模型实现的，一条 Java 线程就映射到一条轻量级进程之中，因为 Windows 和 Linux 系统提供的线程模型就是一对一的。也就是说，现在的 Java 中线程的本质，其实就是操作系统中的线程，Linux 下是基于 pthread 库实现的轻量级进程，Windows 下是原生的系统 Win32 API 提供系统调用从而实现多线程。





在现在的操作系统中，因为线程依旧被视为轻量级进程，所以操作系统中线程的状态实际上和进程状态是一致的模型。从实际意义上来讲，操作系统中的线程除去 new 和 terminated 状态，一个线程真实存在的状态，只有：

- `ready`：表示线程已经被创建，正在等待系统调度分配 CPU 使用权。
- `running`：表示线程获得了 CPU 使用权，正在进行运算。
- `waiting`：表示线程等待（或者说挂起），让出 CPU 资源给其他线程使用。

对于 Java 中的线程状态：无论是 Timed Waiting ， Waiting 还是 Blocked，对应的都是操作系统线程的 waiting（等待）状态。而 Runnable 状态，则对应了操作系统中的 ready 和 running 状态。Java 线程和操作系统线程，实际上同根同源，但又相差甚远。

王下邀月熊 Chevalier

爱代码 爱生活 希望成为全栈整合师

22.3k 声望　**8.5k** 粉丝

关注作者