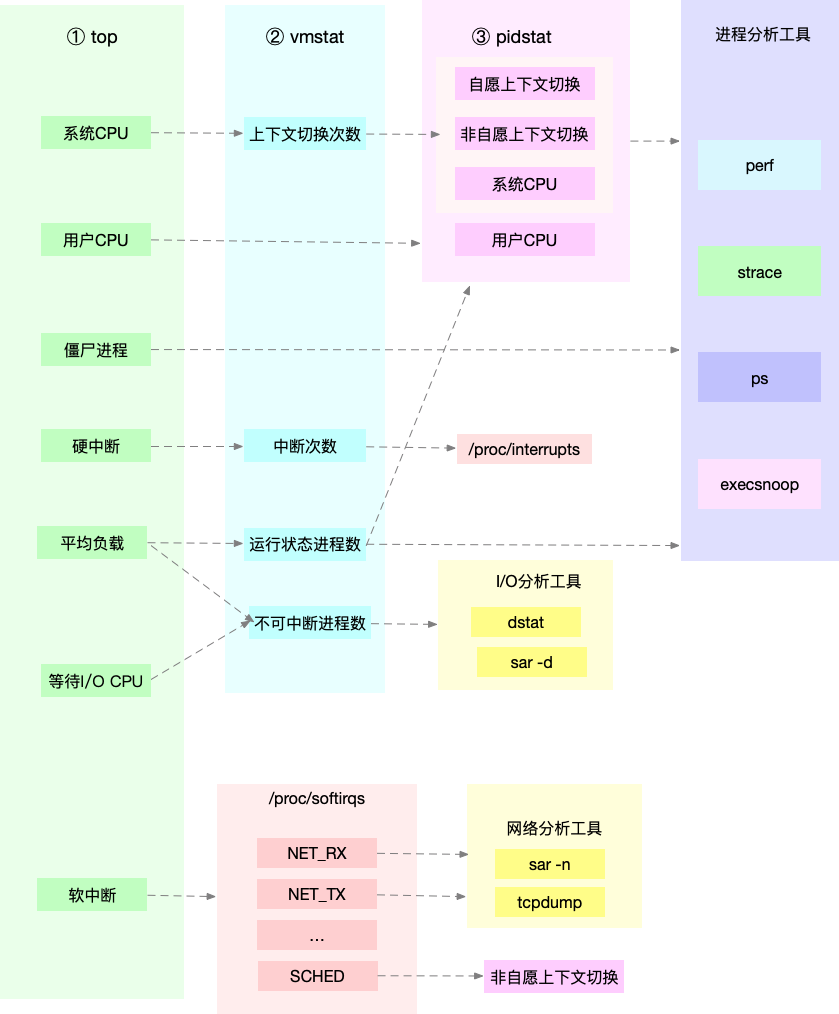
额外参考文档：<http://www.brendangregg.com/>

《Systems Performance: Enterprise and the Cloud》。这本书也出了中文版，名字是《性能之巅：洞悉系统、企业与云计算》

<http://www.brendangregg.com/linuxperf.html>

**cpu**

排查思路



**平均负载**

平均负载是指单位时间内，系统处于可运行状态和不可中断状态的平均进程数，也就是平均活跃进程数。所以，它不仅包括了正在使用 CPU 的进程，还包括等待 CPU 和等待 I/O 的进程。

所谓可运行状态的进程，是指正在使用 CPU 或者正在等待 CPU 的进程，也就是我们常用 ps 命令看到的，处于 R 状态（Running 或 Runnable）的进程。

不可中断状态的进程则是正处于内核态关键流程中的进程，并且这些流程是不可打断的，比如最常见的是等待硬件设备的 I/O 响应，也就是我们在 ps 命令中看到的 D 状态（Uninterruptible Sleep，也称为 Disk Sleep）的进程。

比如当平均负载为 2 时，意味着什么呢？在只有 2 个 CPU 的系统上，意味着所有的 CPU 都刚好被完全占用。在 4 个 CPU 的系统上，意味着 CPU 有 50% 的空闲。而在只有 1 个 CPU 的系统中，则意味着有一半的进程竞争不到 CPU。

当平均负载高于 CPU 数量 70% 的时候，你就应该分析排查负载高的问题了

**cpu使用率**

CPU 使用率是单位时间内 CPU 使用情况的统计，以百分比的方式展示

在上一期我曾提到，Linux 作为一个多任务操作系统，将每个 CPU 的时间划分为很短的时间片，再通过调度器轮流分配给各个任务使用，因此造成多任务同时运行的错觉。为了维护 CPU 时间，Linux 通过事先定义的节拍率（内核中表示为 HZ），触发时间中断，并使用全局变量 Jiffies 记录了开机以来的节拍数。每发生一次时间中断，Jiffies 的值就加 1。节拍率 HZ 是内核的可配选项，可以设置为 100、250、1000 等。不同的系统可能设置不同数值，你可以通过查询 /boot/config 内核选项来查看它的配置值。

$ grep 'CONFIG\_HZ=' /boot/config-$(uname -r)

CONFIG\_HZ=250

同时，正因为节拍率 HZ 是内核选项，所以用户空间程序并不能直接访问。为了方便用户空间程序，内核还提供了一个用户空间节拍率 USER\_HZ，它总是固定为 100，也就是 1/100 秒。这样，用户空间程序并不需要关心内核中 HZ 被设置成了多少，因为它看到的总是固定值 USER\_HZ。

# 只保留各个CPU的数据

$ cat /proc/stat | grep ^cpu

cpu 280580 7407 286084 172900810 83602 0 583 0 0 0

cpu0 144745 4181 176701 86423902 52076 0 301 0 0 0

cpu1 135834 3226 109383 86476907 31525 0 282 0 0 0

这里的输出结果是一个表格。其中，第一列表示的是 CPU 编号，如 cpu0、cpu1 ，而第一行没有编号的 cpu ，表示的是所有 CPU 的累加。其他列则表示不同场景下 CPU 的累加节拍数，它的单位是 USER\_HZ，也就是 10 ms（1/100 秒），所以这其实就是不同场景下的 CPU 时间。

而我们通常所说的 CPU 使用率，就是除了空闲时间外的其他时间占总 CPU 时间的百分比，用公式来表示就是：

不过先不要着急计算，你能说出，直接用 /proc/stat 的数据，算的是什么时间段的 CPU 使用率吗？看到这里，你应该想起来了，这是开机以来的节拍数累加值，所以直接算出来的，是开机以来的平均 CPU 使用率，一般没啥参考价值。事实上，为了计算 CPU 使用率，性能工具一般都会取间隔一段时间（比如 3 秒）的两次值，作差后，再计算出这段时间内的平均 CPU 使用率，即

就是我们用各种性能工具所看到的 CPU 使用率的实际计算方法。

当然，各种各样的性能分析工具已经帮我们计算好了。不过要注意的是，性能分析工具给出的都是间隔一段时间的平均 CPU 使用率，所以要注意间隔时间的设置，特别是用多个工具对比分析时，你一定要保证它们用的是相同的间隔时间。比如，对比一下 top 和 ps 这两个工具报告的 CPU 使用率，默认的结果很可能不一样，因为 top 默认使用 3 秒时间间隔，而 ps 使用的却是进程的整个生命周期。

**cpu使用率跟平均负载的关系**

CPU 密集型进程，使用大量 CPU 会导致平均负载升高，此时这两者是一致的；

I/O 密集型进程，等待 I/O 也会导致平均负载升高，但 CPU 使用率不一定很高；

大量等待 CPU 的进程调度也会导致平均负载升高，此时的 CPU 使用率也会比较高。

cpu使用率会超过100%，如果有4个cpu理论可以到达400%

**cpu使用率过高怎么办**

CPU 使用率是最直观和最常用的系统性能指标，更是我们在排查性能问题时，通常会关注的第一个指标。所以我们更要熟悉它的含义，尤其要弄清楚用户（%user）、Nice（%nice）、系统（%system） 、等待 I/O（%iowait） 、中断（%irq）以及软中断（%softirq）这几种不同 CPU 的使用率。比如说：

用户 CPU 和 Nice CPU 高，说明用户态进程占用了较多的 CPU，所以应该着重排查进程的性能问题。

系统 CPU 高，说明内核态占用了较多的 CPU，所以应该着重排查内核线程或者系统调用的性能问题。

I/O 等待 CPU 高，说明等待 I/O 的时间比较长，所以应该着重排查系统存储是不是出现了 I/O 问题。

软中断和硬中断高，说明软中断或硬中断的处理程序占用了较多的 CPU，所以应该着重排查内核中的中断服务程序。

碰到 CPU 使用率升高的问题，你可以借助 top、pidstat 等工具，确认引发 CPU 性能问题的来源；再使用 perf 等工具，排查出引起性能问题的具体函数。

CPU 使用率描述了非空闲时间占总 CPU 时间的百分比，根据 CPU 上运行任务的不同，又被分为用户 CPU、系统 CPU、等待 I/O CPU、软中断和硬中断等。

用户 CPU 使用率，包括用户态 CPU 使用率（user）和低优先级用户态 CPU 使用率（nice），表示 CPU 在用户态运行的时间百分比。用户 CPU 使用率高，通常说明有应用程序比较繁忙。

系统 CPU 使用率，表示 CPU 在内核态运行的时间百分比（不包括中断）。系统 CPU 使用率高，说明内核比较繁忙。

等待 I/O 的 CPU 使用率，通常也称为 iowait，表示等待 I/O 的时间百分比。iowait 高，通常说明系统与硬件设备的 I/O 交互时间比较长。

软中断和硬中断的 CPU 使用率，分别表示内核调用软中断处理程序、硬中断处理程序的时间百分比。它们的使用率高，通常说明系统发生了大量的中断。

除了上面这些，还有在虚拟化环境中会用到的窃取 CPU 使用率（steal）和客户 CPU 使用率（guest），分别表示被其他虚拟机占用的 CPU 时间百分比，和运行客户虚拟机的 CPU 时间百分比。

**cpu上下文切换**

CPU 寄存器，是 CPU 内置的容量小、但速度极快的内存。而程序计数器，则是用来存储 CPU 正在执行的指令位置、或者即将执行的下一条指令位置。它们都是 CPU 在运行任何任务前，必须的依赖环境，因此也被叫做 CPU 上下文。

CPU 上下文切换，就是先把前一个任务的 CPU 上下文（也就是 CPU 寄存器和程序计数器）保存起来，然后加载新任务的上下文到这些寄存器和程序计数器，最后再跳转到程序计数器所指的新位置，运行新任务。

根据任务的不同，CPU 的上下文切换就可以分为几个不同的场景，也就是进程上下文切换、线程上下文切换以及中断上下文切换。

**进程上下文切换**

**特权模式切换**

Linux 按照特权等级，把进程的运行空间分为内核空间和用户空间，分别对应着下图中， CPU 特权等级的 Ring 0 和 Ring 3。

内核空间（Ring 0）具有最高权限，可以直接访问所有资源；

用户空间（Ring 3）只能访问受限资源，不能直接访问内存等硬件设备，必须通过系统调用陷入到内核中，才能访问这些特权资源。

换个角度看，也就是说，进程既可以在用户空间运行，又可以在内核空间中运行。进程在用户空间运行时，被称为进程的用户态，而陷入内核空间的时候，被称为进程的内核态。

从用户态到内核态的转变，需要通过系统调用来完成。比如，当我们查看文件内容时，就需要多次系统调用来完成：首先调用 open() 打开文件，然后调用 read() 读取文件内容，并调用 write() 将内容写到标准输出，最后再调用 close() 关闭文件。

那么，系统调用的过程有没有发生 CPU 上下文的切换呢？答案自然是肯定的。

CPU 寄存器里原来用户态的指令位置，需要先保存起来。接着，为了执行内核态代码，CPU 寄存器需要更新为内核态指令的新位置。最后才是跳转到内核态运行内核任务。而系统调用结束后，CPU 寄存器需要恢复原来保存的用户态，然后再切换到用户空间，继续运行进程。所以，一次系统调用的过程，其实是发生了两次 CPU 上下文切换。

不过，需要注意的是，系统调用过程中，并不会涉及到虚拟内存等进程用户态的资源，也不会切换进程。这跟我们通常所说的进程上下文切换是不一样的：进程上下文切换，是指从一个进程切换到另一个进程运行。而系统调用过程中一直是同一个进程在运行。

所以，系统调用过程通常称为特权模式切换，而不是上下文切换。但实际上，系统调用过程中，CPU 的上下文切换还是无法避免的。

**进程上下文切换**

进程上下文切换跟系统调用又有什么区别呢？首先，你需要知道，进程是由内核来管理和调度的，进程的切换只能发生在内核态。所以，进程的上下文不仅包括了虚拟内存、栈、全局变量等用户空间的资源，还包括了内核堆栈、寄存器等内核空间的状态。因此，进程的上下文切换就比系统调用时多了一步：在保存当前进程的内核状态和 CPU 寄存器之前，需要先把该进程的虚拟内存、栈等保存下来；而加载了下一进程的内核态后，还需要刷新进程的虚拟内存和用户栈。如下图所示，保存上下文和恢复上下文的过程并不是“免费”的，需要内核在 CPU 上运行才能完成。

根据 Tsuna 的测试报告，每次上下文切换都需要几十纳秒到数微秒的 CPU 时间。这个时间还是相当可观的，特别是在进程上下文切换次数较多的情况下，很容易导致 CPU 将大量时间耗费在寄存器、内核栈以及虚拟内存等资源的保存和恢复上，进而大大缩短了真正运行进程的时间。这也正是上一节中我们所讲的，导致平均负载升高的一个重要因素。

另外，我们知道， Linux 通过 TLB（Translation Lookaside Buffer）来管理虚拟内存到物理内存的映射关系。当虚拟内存更新后，TLB 也需要刷新，内存的访问也会随之变慢。特别是在多处理器系统上，缓存是被多个处理器共享的，刷新缓存不仅会影响当前处理器的进程，还会影响共享缓存的其他处理器的进程。

**什么时间会进程上下文切换**

显然，进程切换时才需要切换上下文，换句话说，只有在进程调度的时候，才需要切换上下文。Linux 为每个 CPU 都维护了一个就绪队列，将活跃进程（即正在运行和正在等待 CPU 的进程）按照优先级和等待 CPU 的时间排序，然后选择最需要 CPU 的进程，也就是优先级最高和等待 CPU 时间最长的进程来运行。那么，进程在什么时候才会被调度到 CPU 上运行呢？最容易想到的一个时机，就是进程执行完终止了，它之前使用的 CPU 会释放出来，这个时候再从就绪队列里，拿一个新的进程过来运行。其实还有很多其他场景，也会触发进程调度，在这里我给你逐个梳理下。

其一，为了保证所有进程可以得到公平调度，CPU 时间被划分为一段段的时间片，这些时间片再被轮流分配给各个进程。这样，当某个进程的时间片耗尽了，就会被系统挂起，切换到其它正在等待 CPU 的进程运行。

其二，进程在系统资源不足（比如内存不足）时，要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行。

其二，进程在系统资源不足（比如内存不足）时，要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行。

其三，当进程通过睡眠函数 sleep 这样的方法将自己主动挂起时，自然也会重新调度。

其四，当有优先级更高的进程运行时，为了保证高优先级进程的运行，当前进程会被挂起，由高优先级进程来运行。

最后一个，发生硬件中断时，CPU 上的进程会被中断挂起，转而执行内核中的中断服务程序。

**线程上下文切换**

线程与进程最大的区别在于，线程是调度的基本单位，而进程则是资源拥有的基本单位。说白了，所谓内核中的任务调度，实际上的调度对象是线程；而进程只是给线程提供了虚拟内存、全局变量等资源。

所以，对于线程和进程，我们可以这么理解：当进程只有一个线程时，可以认为进程就等于线程。当进程拥有多个线程时，这些线程会共享相同的虚拟内存和全局变量等资源。这些资源在上下文切换时是不需要修改的。另外，线程也有自己的私有数据，比如栈和寄存器等，这些在上下文切换时也是需要保存的。

这么一来，线程的上下文切换其实就可以分为两种情况：

第一种， 前后两个线程属于不同进程。此时，因为资源不共享，所以切换过程就跟进程上下文切换是一样。

第二种，前后两个线程属于同一个进程。此时，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据。

到这里你应该也发现了，虽然同为上下文切换，但同进程内的线程切换，要比多进程间的切换消耗更少的资源，而这，也正是多线程代替多进程的一个优势。

**中断上下文切换**

为了快速响应硬件的事件，中断处理会打断进程的正常调度和执行，转而调用中断处理程序，响应设备事件。而在打断其他进程时，就需要将进程当前的状态保存下来，这样在中断结束后，进程仍然可以从原来的状态恢复运行。

跟进程上下文不同，中断上下文切换并不涉及到进程的用户态。所以，即便中断过程打断了一个正处在用户态的进程，也不需要保存和恢复这个进程的虚拟内存、全局变量等用户态资源。中断上下文，其实只包括内核态中断服务程序执行所必需的状态，包括 CPU 寄存器、内核堆栈、硬件中断参数等。

对同一个 CPU 来说，中断处理比进程拥有更高的优先级，所以中断上下文切换并不会与进程上下文切换同时发生。同样道理，由于中断会打断正常进程的调度和执行，所以大部分中断处理程序都短小精悍，以便尽可能快的执行结束。

另外，跟进程上下文切换一样，中断上下文切换也需要消耗 CPU，切换次数过多也会耗费大量的 CPU，甚至严重降低系统的整体性能。所以，当你发现中断次数过多时，就需要注意去排查它是否会给你的系统带来严重的性能问题。

**每秒上下文切换多少次才算正常**

这个数值其实取决于系统本身的 CPU 性能。在我看来，如果系统的上下文切换次数比较稳定，那么从数百到一万以内，都应该算是正常的。但当上下文切换次数超过一万次，或者切换次数出现数量级的增长时，就很可能已经出现了性能问题。这时，你还需要根据上下文切换的类型，再做具体分析。比方说：

自愿上下文切换变多了，说明进程都在等待资源，有可能发生了 I/O 等其他问题；

非自愿上下文切换变多了，说明进程都在被强制调度，也就是都在争抢 CPU，说明 CPU 的确成了瓶颈；

中断次数变多了，说明 CPU 被中断处理程序占用，还需要通过查看 /proc/interrupts 文件来分析具体的中断类型。

**进程**

ps 的输出中，名字括在中括号里的，一般都是内核线程。

**进程状态**

R 是 Running 或 Runnable 的缩写，表示进程在 CPU 的就绪队列中，正在运行或者正在等待运行。

D 是 Disk Sleep 的缩写，也就是不可中断状态睡眠（Uninterruptible Sleep），一般表示进程正在跟硬件交互，并且交互过程不允许被其他进程或中断打断。

Z 是 Zombie 的缩写，如果你玩过“植物大战僵尸”这款游戏，应该知道它的意思。它表示僵尸进程，也就是进程实际上已经结束了，但是父进程还没有回收它的资源（比如进程的描述符、PID 等）。

S 是 Interruptible Sleep 的缩写，也就是可中断状态睡眠，表示进程因为等待某个事件而被系统挂起。当进程等待的事件发生时，它会被唤醒并进入 R 状态。

I 是 Idle 的缩写，也就是空闲状态，用在不可中断睡眠的内核线程上。前面说了，硬件交互导致的不可中断进程用 D 表示，但对某些内核线程来说，它们有可能实际上并没有任何负载，用 Idle 正是为了区分这种情况。要注意，D 状态的进程会导致平均负载升高， I 状态的进程却不会。

T 或者 t，也就是 Stopped 或 Traced 的缩写，表示进程处于暂停或者跟踪状态。向一个进程发送 SIGSTOP 信号，它就会因响应这个信号变成暂停状态（Stopped）；再向它发送 SIGCONT 信号，进程又会恢复运行（如果进程是终端里直接启动的，则需要你用 fg 命令，恢复到前台运行）。而当你用调试器（如 gdb）调试一个进程时，在使用断点中断进程后，进程就会变成跟踪状态，这其实也是一种特殊的暂停状态，只不过你可以用调试器来跟踪并按需要控制进程的运行。

X，也就是 Dead 的缩写，表示进程已经消亡，所以你不会在 top 或者 ps 命令中看到它。

s 表示这个进程是一个会话的领导进程，而 + 表示前台进程组.进程组表示一组相互关联的进程，比如每个子进程都是父进程所在组的成员；而会话是指共享同一个控制终端的一个或多个进程组。

**中断**

中断是系统用来响应硬件设备请求的一种机制，它会打断进程的正常调度和执行，然后调用内核中的中断处理程序来响应设备的请求。

举例：比如说你订了一份外卖，但是不确定外卖什么时候送到，也没有别的方法了解外卖的进度，但是，配送员送外卖是不等人的，到了你这儿没人取的话，就直接走人了。所以你只能苦苦等着，时不时去门口看看外卖送到没，而不能干其他事情。在订外卖的时候，你就跟配送员约定好，让他送到后给你打个电话，那你就不用苦苦等待了，就可以去忙别的事情，直到电话一响，接电话、取外卖就可以了。这里的“打电话”，其实就是一个中断。没接到电话的时候，你可以做其他的事情；只有接到了电话（也就是发生中断），你才要进行另一个动作：取外卖。

中断其实是一种异步的事件处理机制，可以提高系统的并发处理能力。由于中断处理程序会打断其他进程的运行，所以，为了减少对正常进程运行调度的影响，中断处理程序就需要尽可能快地运行。如果中断本身要做的事情不多，那么处理起来也不会有太大问题；但如果中断要处理的事情很多，中断服务程序就有可能要运行很长时间。特别是，中断处理程序在响应中断时，还会临时关闭中断。这就会导致上一次中断处理完成之前，其他中断都不能响应，也就是说中断有可能会丢失。

那么还是以取外卖为例。假如你订了 2 份外卖，一份主食和一份饮料，并且是由 2 个不同的配送员来配送。这次你不用时时等待着，两份外卖都约定了电话取外卖的方式。但是，问题又来了。当第一份外卖送到时，配送员给你打了个长长的电话，商量发票的处理方式。与此同时，第二个配送员也到了，也想给你打电话。但是很明显，因为电话占线（也就是关闭了中断响应），第二个配送员的电话是打不通的。所以，第二个配送员很可能试几次后就走掉了（也就是丢失了一次中断）。

**软中断**

如果你弄清楚了“取外卖”的模式，那对系统的中断机制就很容易理解了。事实上，为了解决中断处理程序执行过长和中断丢失的问题，Linux 将中断处理过程分成了两个阶段，也就是上半部和下半部：

上半部用来快速处理中断，它在中断禁止模式下运行，主要处理跟硬件紧密相关的或时间敏感的工作。

下半部用来延迟处理上半部未完成的工作，通常以内核线程的方式运行。

举例：网卡接收到数据包后，会通过硬件中断的方式，通知内核有新的数据到了。这时，内核就应该调用中断处理程序来响应它。你可以自己先想一下，这种情况下的上半部和下半部分别负责什么工作呢？对上半部来说，既然是快速处理，其实就是要把网卡的数据读到内存中，然后更新一下硬件寄存器的状态（表示数据已经读好了），最后再发送一个软中断信号，通知下半部做进一步的处理。而下半部被软中断信号唤醒后，需要从内存中找到网络数据，再按照网络协议栈，对数据进行逐层解析和处理，直到把它送给应用程序。所以，这两个阶段你也可以这样理解：

上半部直接处理硬件请求，也就是我们常说的硬中断，特点是快速执行；

而下半部则是由内核触发，也就是我们常说的软中断，特点是延迟执行。

实际上，上半部会打断 CPU 正在执行的任务，然后立即执行中断处理程序。而下半部以内核线程的方式执行，并且每个 CPU 都对应一个软中断内核线程，名字为 “ksoftirqd/CPU 编号”，比如说， 0 号 CPU 对应的软中断内核线程的名字就是 ksoftirqd/0。

不过要注意的是，软中断不只包括了刚刚所讲的硬件设备中断处理程序的下半部，一些内核自定义的事件也属于软中断，比如内核调度和 RCU 锁（Read-Copy Update 的缩写，RCU 是 Linux 内核中最常用的锁之一）等。

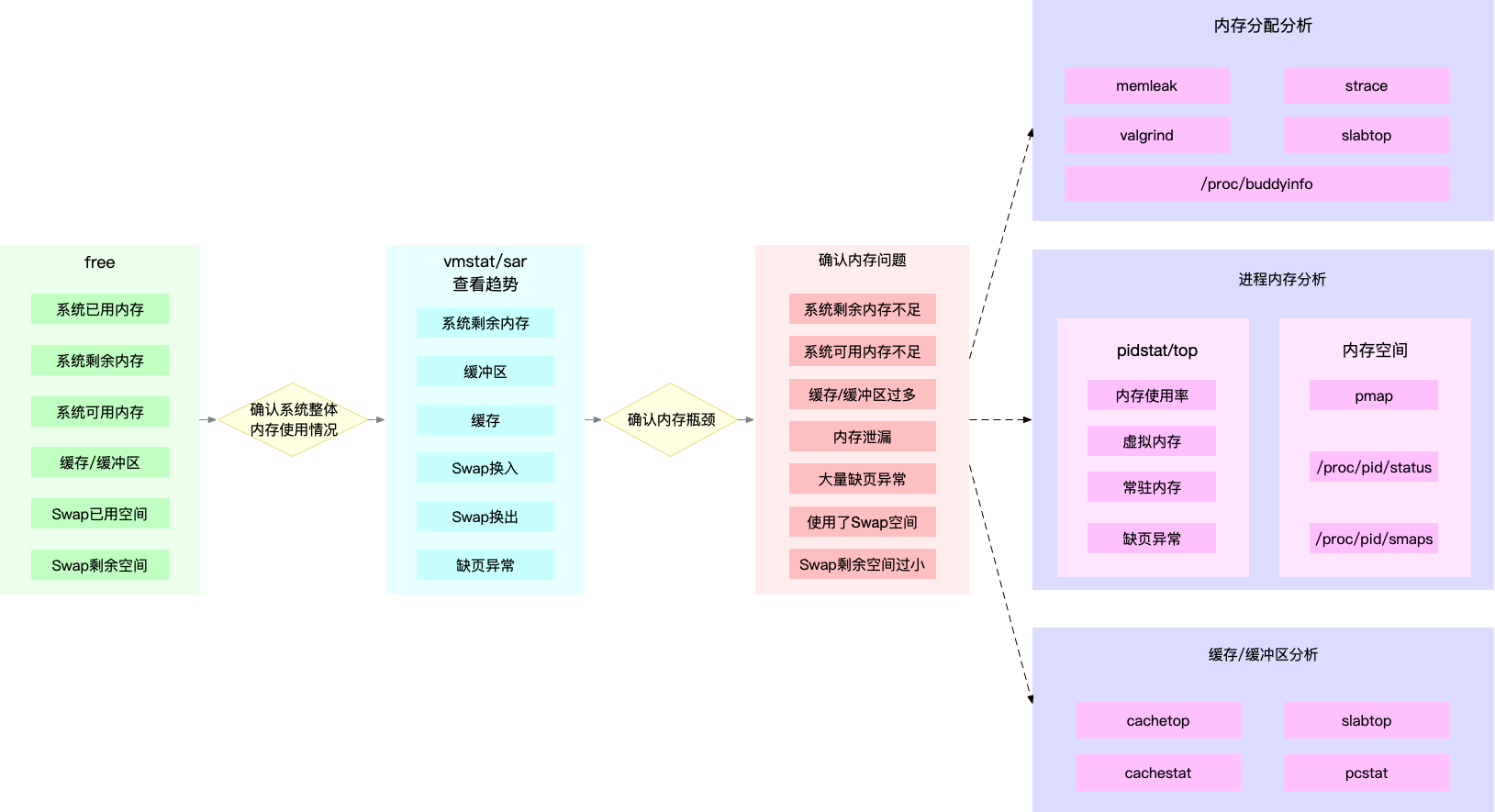
/proc/softirqs 提供了软中断的运行情况；

/proc/interrupts 提供了硬中断的运行情况。

它是系统运行以来的累积中断次数

**内存**





**内存映射**

Linux 内核给每个进程都提供了一个独立的虚拟地址空间，并且这个地址空间是连续的。这样，进程就可以很方便地访问内存，更确切地说是访问虚拟内存。

既然每个进程都有一个这么大的地址空间，那么所有进程的虚拟内存加起来，自然要比实际的物理内存大得多。所以，并不是所有的虚拟内存都会分配物理内存，只有那些实际使用的虚拟内存才分配物理内存，并且分配后的物理内存，是通过内存映射来管理的。

**内存映射**，其实就是将虚拟内存地址映射到物理内存地址。为了完成内存映射，内核为每个进程都维护了一张页表，记录虚拟地址与物理地址的映射关系

页表实际上存储在 CPU 的内存管理单元 MMU 中，这样，正常情况下，处理器就可以直接通过硬件，找出要访问的内存。而当进程访问的虚拟地址在页表中查不到时，系统会产生一个缺页异常，进入内核空间分配物理内存、更新进程页表，最后再返回用户空间，恢复进程的运行。MMU 并不以字节为单位来管理内存，而是规定了一个内存映射的最小单位，也就是页，通常是 4 KB 大小。这样，每一次内存映射，都需要关联 4 KB 或者 4KB 整数倍的内存空间。

页的大小只有 4 KB ，导致的另一个问题就是，整个页表会变得非常大。比方说，仅 32 位系统就需要 100 多万个页表项（4GB/4KB），才可以实现整个地址空间的映射。为了解决页表项过多的问题，Linux 提供了两种机制，也就是多级页表和大页（HugePage）。

多级页表就是把内存分成区块来管理，将原来的映射关系改成区块索引和区块内的偏移。由于虚拟内存空间通常只用了很少一部分，那么，多级页表就只保存这些使用中的区块，这样就可以大大地减少页表的项数。Linux 用的正是四级页表来管理内存页，如下图所示，虚拟地址被分为 5 个部分，前 4 个表项用于选择页，而最后一个索引表示页内偏移。

大页，顾名思义，就是比普通页更大的内存块，常见的大小有 2MB 和 1GB。大页通常用在使用大量内存的进程上，比如 Oracle、DPDK 等。

**虚拟内存空间分布**

通过这张图你可以看到，用户空间内存，从低到高分别是五种不同的内存段。

1. 只读段，包括代码和常量等。

2. 数据段，包括全局变量等。

3. 堆，包括动态分配的内存，从低地址开始向上增长。

4. 文件映射段，包括动态库、共享内存等，从高地址开始向下增长。

5. 栈，包括局部变量和函数调用的上下文等。栈的大小是固定的，一般是 8 MB。

在这五个内存段中，堆和文件映射段的内存是动态分配的。比如说，使用 C 标准库的 malloc() 或者 mmap() ，就可以分别在堆和文件映射段动态分配内存。

**内存分配与回收**

malloc() 是 C 标准库提供的内存分配函数，对应到系统调用上，有两种实现方式，即 brk() 和 mmap()。

对小块内存（小于 128K），C 标准库使用 brk() 来分配，也就是通过移动堆顶的位置来分配内存。这些内存释放后并不会立刻归还系统，而是被缓存起来，这样就可以重复使用。

而大块内存（大于 128K），则直接使用内存映射 mmap() 来分配，也就是在文件映射段找一块空闲内存分配出去。

回收

1. 回收缓存，比如使用 LRU（Least Recently Used）算法，回收最近使用最少的内存页面；

2. 回收不常访问的内存，把不常用的内存通过交换分区直接写到磁盘中；

其中，第二种方式回收不常访问的内存时，会用到交换分区（以下简称 Swap）。Swap 其实就是把一块磁盘空间当成内存来用。它可以把进程暂时不用的数据存储到磁盘中（这个过程称为换出），当进程访问这些内存时，再从磁盘读取这些数据到内存中（这个过程称为换入）。所以，你可以发现，Swap 把系统的可用内存变大了。不过要注意，通常只在内存不足时，才会发生 Swap 交换。并且由于磁盘读写的速度远比内存慢，Swap 会导致严重的内存性能问题。

3. 杀死进程，内存紧张时系统还会通过 OOM（Out of Memory），直接杀掉占用大量内存的进程。

OOM（Out of Memory），其实是内核的一种保护机制。它监控进程的内存使用情况，并且使用 oom\_score 为每个进程的内存使用情况进行评分：一个进程消耗的内存越大，oom\_score 就越大，越容易被 OOM 杀死；一个进程运行占用的 CPU 越多，oom\_score 就越小。

可以通过 /proc 文件系统，手动设置进程的 oom\_adj ，从而调整进程的 oom\_score。oom\_adj 的范围是 [-17, 15]，数值越大，表示进程越容易被 OOM 杀死；数值越小，表示进程越不容易被 OOM 杀死，其中 -17 表示禁止 OOM。

**buffer/cache**

从 free 的手册中，你可以看到 buffer 和 cache 的说明。Buffers是内核缓冲区用到的内存，对应的是 /proc/meminfo 中的 Buffers 值。Cache是内核页缓存和 Slab 用到的内存，对应的是 /proc/meminfo 中的 Cached 与 SReclaimable 之和。

从proc帮助手册，

Buffers是对原始磁盘块的临时存储，也就是用来缓存磁盘的数据，通常不会特别大（20MB 左右）。这样，内核就可以把分散的写集中起来，统一优化磁盘的写入，比如可以把多次小的写合并成单次大的写等等。

Cached是从磁盘读取文件的页缓存，也就是用来缓存从文件读取的数据。这样，下次访问这些文件数据时，就可以直接从内存中快速获取，而不需要再次访问缓慢的磁盘。

SReclaimable是 Slab 的一部分。Slab 包括两部分，其中的可回收部分，用 SReclaimable 记录；而不可回收部分，用 SUnreclaim 记录。

**磁盘和文件写案例**

实验1 终端1： echo 3 > /proc/sys/vm/drop\_caches

vmstat 1

终端2： dd if=/dev/urandom of=/tmp/file bs=1M count=500

在终端1 可以观察到cache 上升比较明显，buffer 没怎么改变。

实验2 终端1： 同上

终端2： dd if=/dev/urandom of=/dev/sdb1 bs=1M count=2048

在终端1 可以观察到cache、buffer 都有上升，buffer 明显

对比两个案例，写文件时会用到 Cache 缓存数据，而写磁盘则会用到 Buffer 来缓存数据。所以，回到刚刚的问题，虽然文档上只提到，Cache 是文件读的缓存，但实际上，Cache 也会缓存写文件时的数据。

磁盘和文件读案例

# 首先清理缓存$ echo 3 > /proc/sys/vm/drop\_caches# 运行dd命令读取文件$ dd if=/dev/sda1 of=/dev/null bs=1M count=1024

读文件时数据会缓存到 Cache 中，而读磁盘时数据会缓存到 Buffer 中

结论：Buffer 是对磁盘数据的缓存，而 Cache 是文件数据的缓存，它们既会用在读请求中，也会用在写请求中。

**缓存命中率**

所谓缓存命中率，是指直接通过缓存获取数据的请求次数，占所有数据请求次数的百分比。命中率越高，表示使用缓存带来的收益越高，应用程序的性能也就越好。

**swap**

在内存资源紧张时，Linux 通过直接内存回收和定期扫描的方式，来释放文件页和匿名页，以便把内存分配给更需要的进程使用。

文件页的回收比较容易理解，直接清空缓存，或者把脏数据写回磁盘后，再释放缓存就可以了。

而对不常访问的匿名页，则需要通过 Swap 换出到磁盘中，这样在下次访问的时候，再次从磁盘换入到内存中就可以了。开启 Swap 后，你可以设置 /proc/sys/vm/min\_free\_kbytes ，来调整系统定期回收内存的阈值，也可以设置 /proc/sys/vm/swappiness ，来调整文件页和匿名页的回收倾向。swappiness 0-100，值越大，越倾向使用swap。即使配置了0，也会使用swap

开启swap

Linux 本身支持两种类型的 Swap，即 Swap 分区和 Swap 文件。以 Swap 文件为例，在第一个终端中运行下面的命令开启 Swap，我这里配置 Swap 文件的大小为 8GB：

# 创建Swap文件

$ fallocate -l 8G /mnt/swapfile

# 修改权限只有根用户可以访问

$ chmod 600 /mnt/swapfile

# 配置Swap文件$ mkswap /mnt/swapfile

# 开启Swap

$ swapon /mnt/swapfile

这里我还是推荐 proc 文件系统，用来查看进程 Swap 换出的虚拟内存大小，它保存在 /proc/pid/status 中的 VmSwap 中

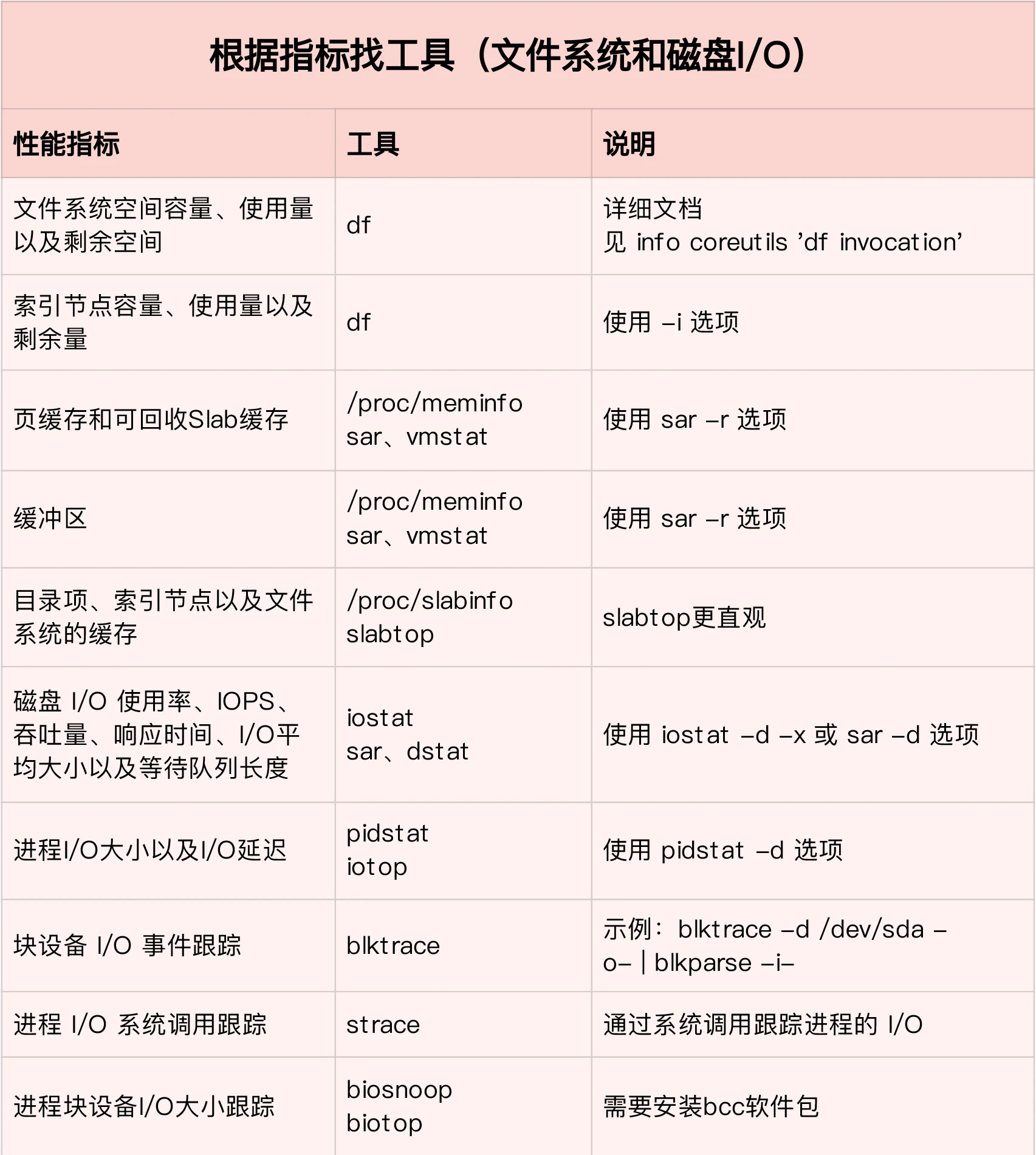
关闭 Swap：

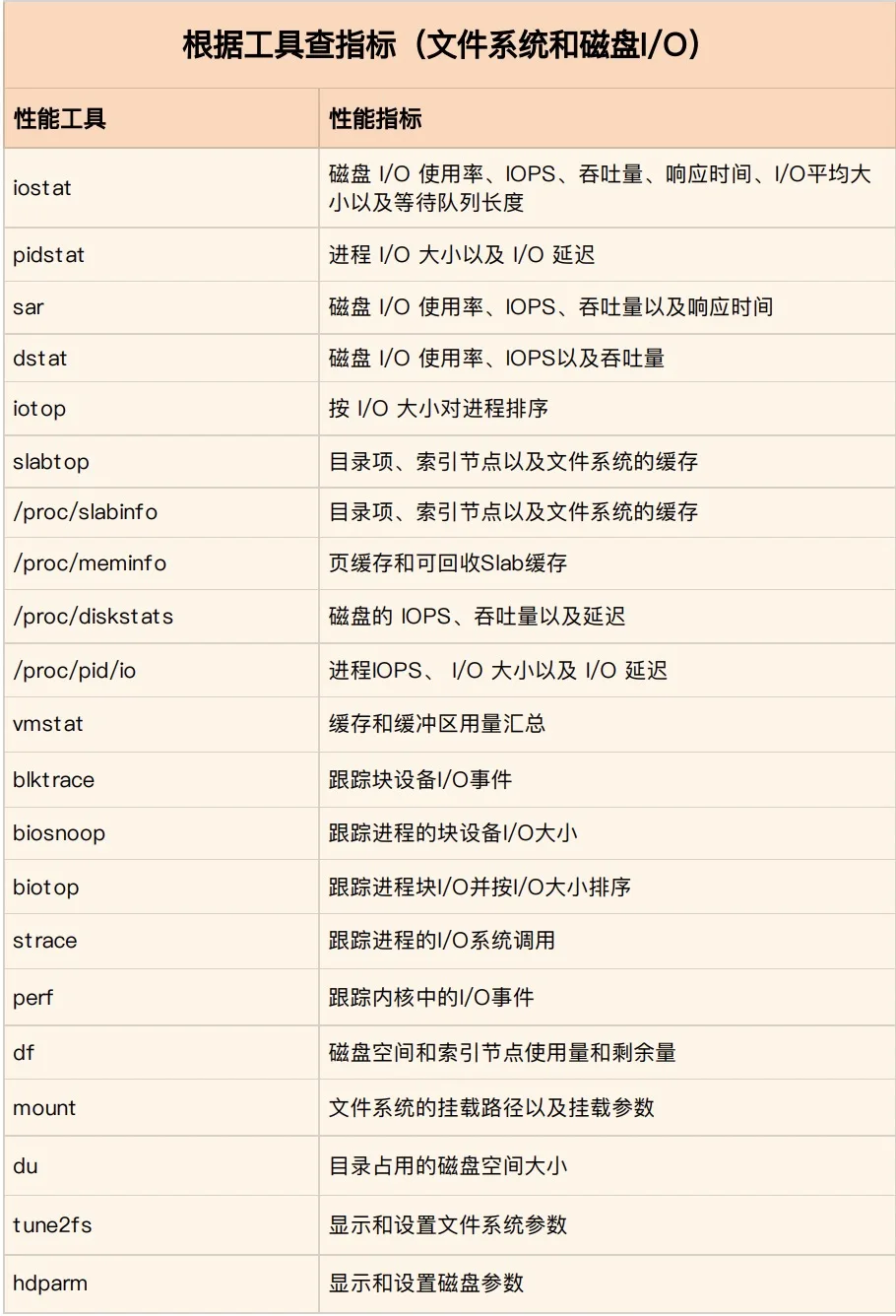
$ swapoff -a

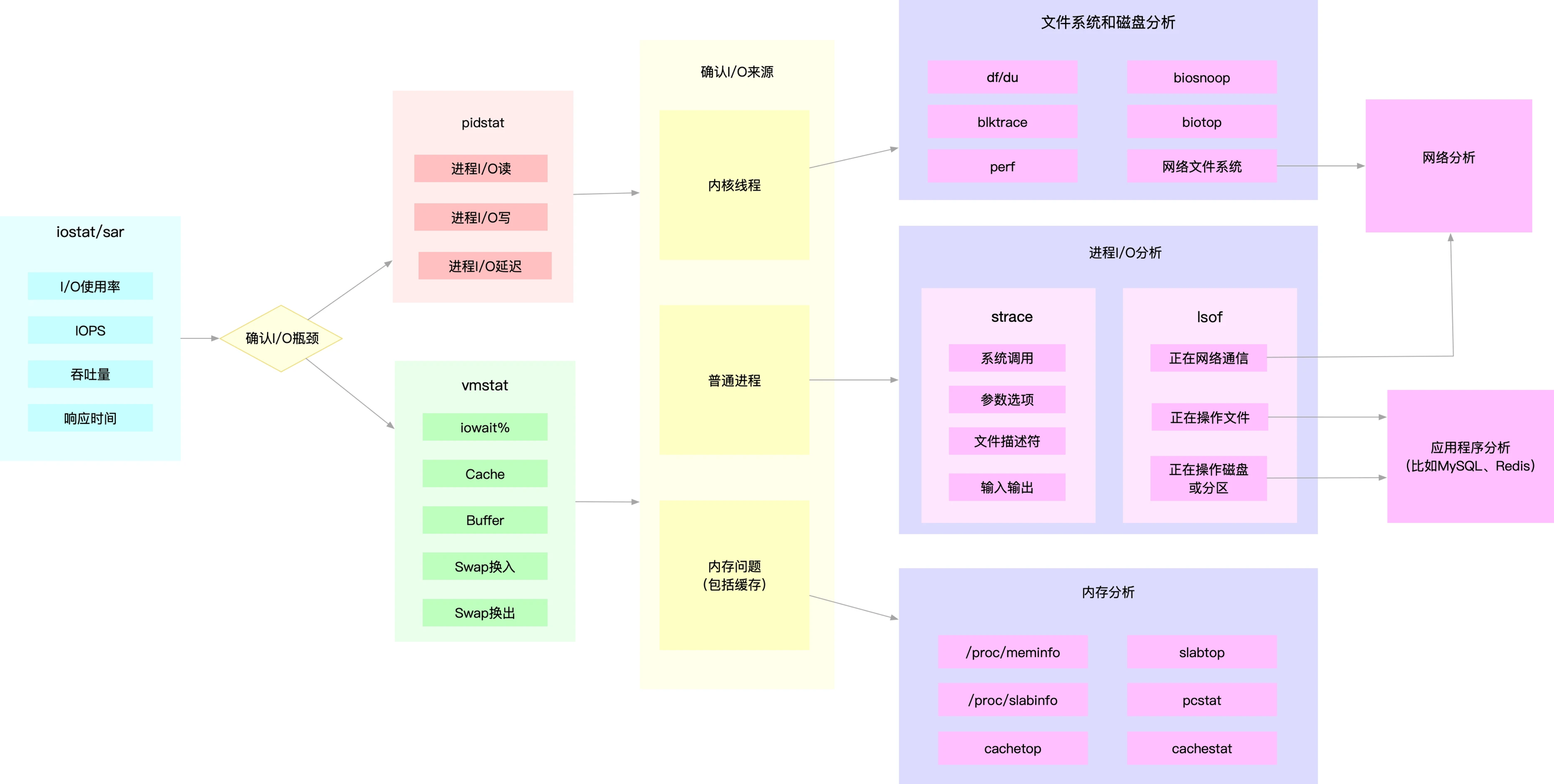
**I/O**

排查思路









**索引节点和目录项**

文件系统，本身是对存储设备上的文件，进行组织管理的机制。组织方式不同，就会形成不同的文件系统。你要记住最重要的一点，在 Linux 中一切皆文件。不仅普通的文件和目录，就连块设备、套接字、管道等，也都要通过统一的文件系统来管理。为了方便管理，Linux 文件系统为每个文件都分配两个数据结构，索引节点（index node）和目录项（directory entry）。它们主要用来记录文件的元信息和目录结构。

索引节点，简称为 inode，用来记录文件的元数据，比如 inode 编号、文件大小、访问权限、修改日期、数据的位置等。索引节点和文件一一对应，它跟文件内容一样，都会被持久化存储到磁盘中。所以记住，索引节点同样占用磁盘空间。

目录项，简称为 dentry，用来记录文件的名字、索引节点指针以及与其他目录项的关联关系。多个关联的目录项，就构成了文件系统的目录结构。不过，不同于索引节点，目录项是由内核维护的一个内存数据结构，所以通常也被叫做目录项缓存。换句话说，索引节点是每个文件的唯一标志，而目录项维护的正是文件系统的树状结构。目录项和索引节点的关系是多对一，你可以简单理解为，一个文件可以有多个别名。

举个例子，通过硬链接为文件创建的别名，就会对应不同的目录项，不过这些目录项本质上还是链接同一个文件，所以，它们的索引节点相同。

硬连接文件的目录项和原文件目录项指向同一个索引节点，索引节点的引用计数会加1，如果原目录项被删除（即原文件被删除），或硬连接被删除，只要索引节点的引用计数不为0，其它目录项就可以访问到文件数据。硬连接不能指向目录。 软连接不仅会有新的目录项还会创建一个新的索引节点，索引节点对应的逻辑块中存储的数据是被链接文件的目录和索引节点（重点）。即软连接和被链接文件具有不同的索引节点，如原文件被删除了，因为原文件的索引节点没有被其他引用，所以变为0，那么索引节点和索引节点对应的数据块就会被删除，那么此时软连接访问时发现自己的索引节点中存储的被链接文件的目录在磁盘中不存在了，就会出错，现象是软连接变红

这里有两点需要你注意。

第一， 目录项本身就是一个内存缓存，而索引节点则是存储在磁盘中的数据。在前面的 Buffer 和 Cache 原理中，我曾经提到过，为了协调慢速磁盘与快速 CPU 的性能差异，文件内容会缓存到页缓存 Cache 中。那么，你应该想到，这些索引节点自然也会缓存到内存中，加速文件的访问。

第二， 第二，磁盘在执行文件系统格式化时，会被分成三个存储区域，超级块、索引节点区和数据块区。其中，

超级块，存储整个文件系统的状态。

索引节点区，用来存储索引节点。

数据块区，则用来存储文件数据。

**虚拟文件系统vfs**

VFS 定义了一组所有文件系统都支持的数据结构和标准接口。这样，用户进程和内核中的其他子系统，只需要跟 VFS 提供的统一接口进行交互就可以了，而不需要再关心底层各种文件系统的实现细节。

文件系统可以分为三类。

第一类是基于磁盘的文件系统，也就是把数据直接存储在计算机本地挂载的磁盘中。常见的 Ext4、XFS、OverlayFS 等，都是这类文件系统。

第二类是基于内存的文件系统，也就是我们常说的虚拟文件系统。这类文件系统，不需要任何磁盘分配存储空间，但会占用内存。我们经常用到的 /proc 文件系统，其实就是一种最常见的虚拟文件系统。此外，/sys 文件系统也属于这一类，主要向用户空间导出层次化的内核对象。

第三类是网络文件系统，也就是用来访问其他计算机数据的文件系统，比如 NFS、SMB、iSCSI 等。

“Linux 一切皆文件”的深刻含义。无论是普通文件和块设备、还是网络套接字和管道等，它们都通过统一的 VFS 接口来访问。

**磁盘**

磁盘是可以持久化存储的设备，根据存储介质的不同，常见磁盘可以分为两类：机械磁盘和固态磁盘。

第一类，机械磁盘，也称为硬盘驱动器（Hard Disk Driver），通常缩写为 HDD。机械磁盘主要由盘片和读写磁头组成，数据就存储在盘片的环状磁道中。在读写数据前，需要移动读写磁头，定位到数据所在的磁道，然后才能访问数据。显然，如果 I/O 请求刚好连续，那就不需要磁道寻址，自然可以获得最佳性能。这其实就是我们熟悉的，连续 I/O 的工作原理。与之相对应的，当然就是随机 I/O，它需要不停地移动磁头，来定位数据位置，所以读写速度就会比较慢。

第二类，固态磁盘（Solid State Disk），通常缩写为 SSD，由固态电子元器件组成。固态磁盘不需要磁道寻址，所以，不管是连续 I/O，还是随机 I/O 的性能，都比机械磁盘要好得多。

机械磁盘的最小读写单位是扇区，一般大小为 512 字节。

而固态磁盘的最小读写单位是页，通常大小是 4KB、8KB 等。

除了可以按照存储介质来分类，另一个常见的分类方法，是按照接口来分类，比如可以把硬盘分为 IDE（Integrated Drive Electronics）、SCSI（Small Computer System Interface） 、SAS（Serial Attached SCSI） 、SATA（Serial ATA） 、FC（Fibre Channel） 等。不同的接口，往往分配不同的设备名称。比如， IDE 设备会分配一个 hd 前缀的设备名，SCSI 和 SATA 设备会分配一个 sd 前缀的设备名。如果是多块同类型的磁盘，就会按照 a、b、c 等的字母顺序来编号。

**通用块层**

跟我们上一节讲到的虚拟文件系统 VFS 类似，为了减小不同块设备的差异带来的影响，Linux 通过一个统一的通用块层，来管理各种不同的块设备。通用块层，其实是处在文件系统和磁盘驱动中间的一个块设备抽象层。

它主要有两个功能 。第一个功能跟虚拟文件系统的功能类似。向上，为文件系统和应用程序，提供访问块设备的标准接口；向下，把各种异构的磁盘设备抽象为统一的块设备，并提供统一框架来管理这些设备的驱动程序。第二个功能，通用块层还会给文件系统和应用程序发来的 I/O 请求排队，并通过重新排序、请求合并等方式，提高磁盘读写的效率。

对 I/O 请求排序的过程，也就是我们熟悉的 I/O 调度。事实上，Linux 内核支持四种 I/O 调度算法，分别是 NONE、NOOP、CFQ 以及 DeadLine

**I/O 栈**

存储系统的 I/O 栈，由上到下分为三个层次，分别是文件系统层、通用块层和设备层

文件系统层，包括虚拟文件系统和其他各种文件系统的具体实现。它为上层的应用程序，提供标准的文件访问接口；对下会通过通用块层，来存储和管理磁盘数据。

通用块层，包括块设备 I/O 队列和 I/O 调度器。它会对文件系统的 I/O 请求进行排队，再通过重新排序和请求合并，然后才要发送给下一级的设备层。

设备层，包括存储设备和相应的驱动程序，负责最终物理设备的 I/O 操作。

存储系统的 I/O ，通常是整个系统中最慢的一环。所以， Linux 通过多种缓存机制来优化 I/O 效率。比方说，为了优化文件访问的性能，会使用页缓存、索引节点缓存、目录项缓存等多种缓存机制，以减少对下层块设备的直接调用。

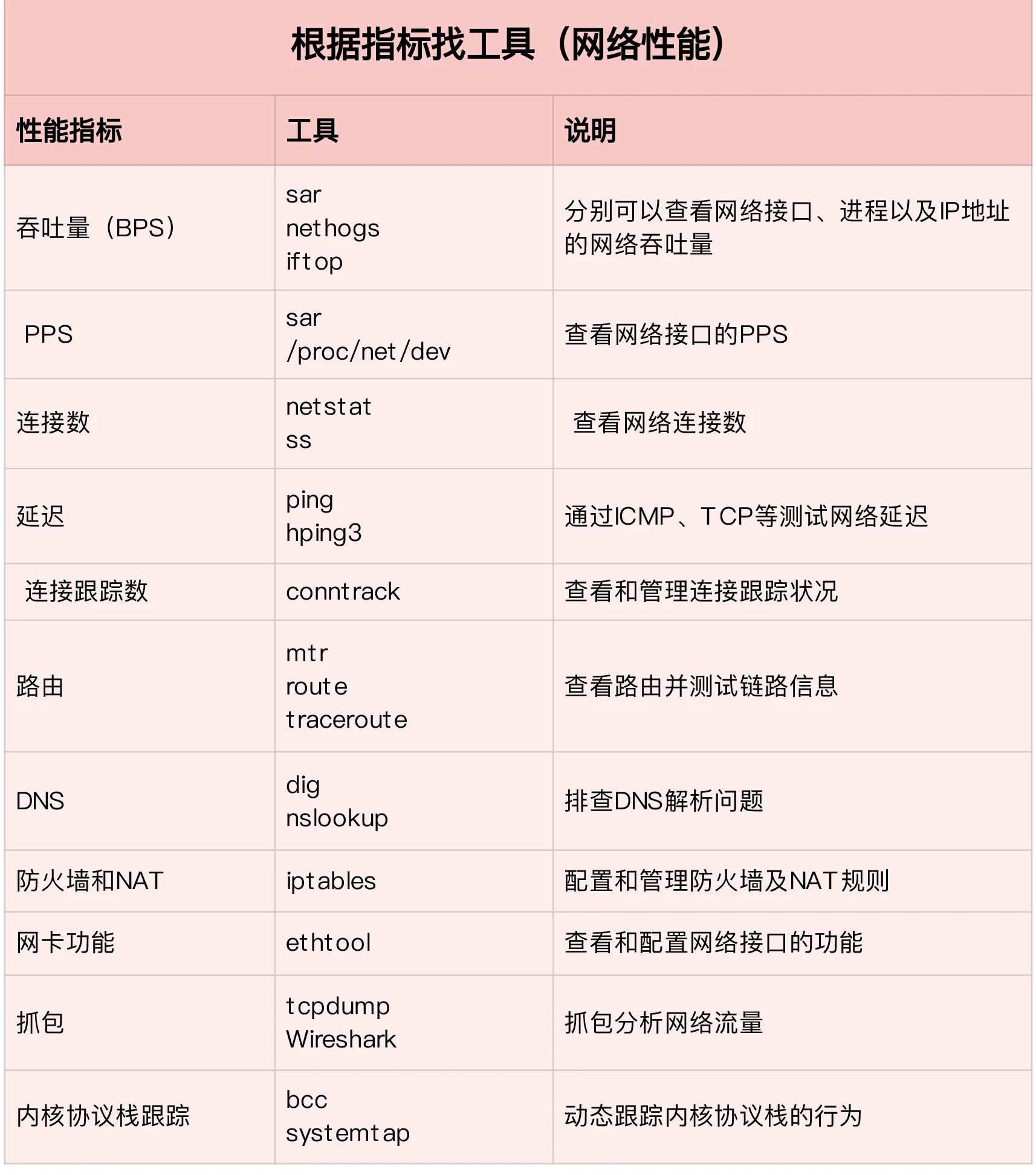
**磁盘性能指标**

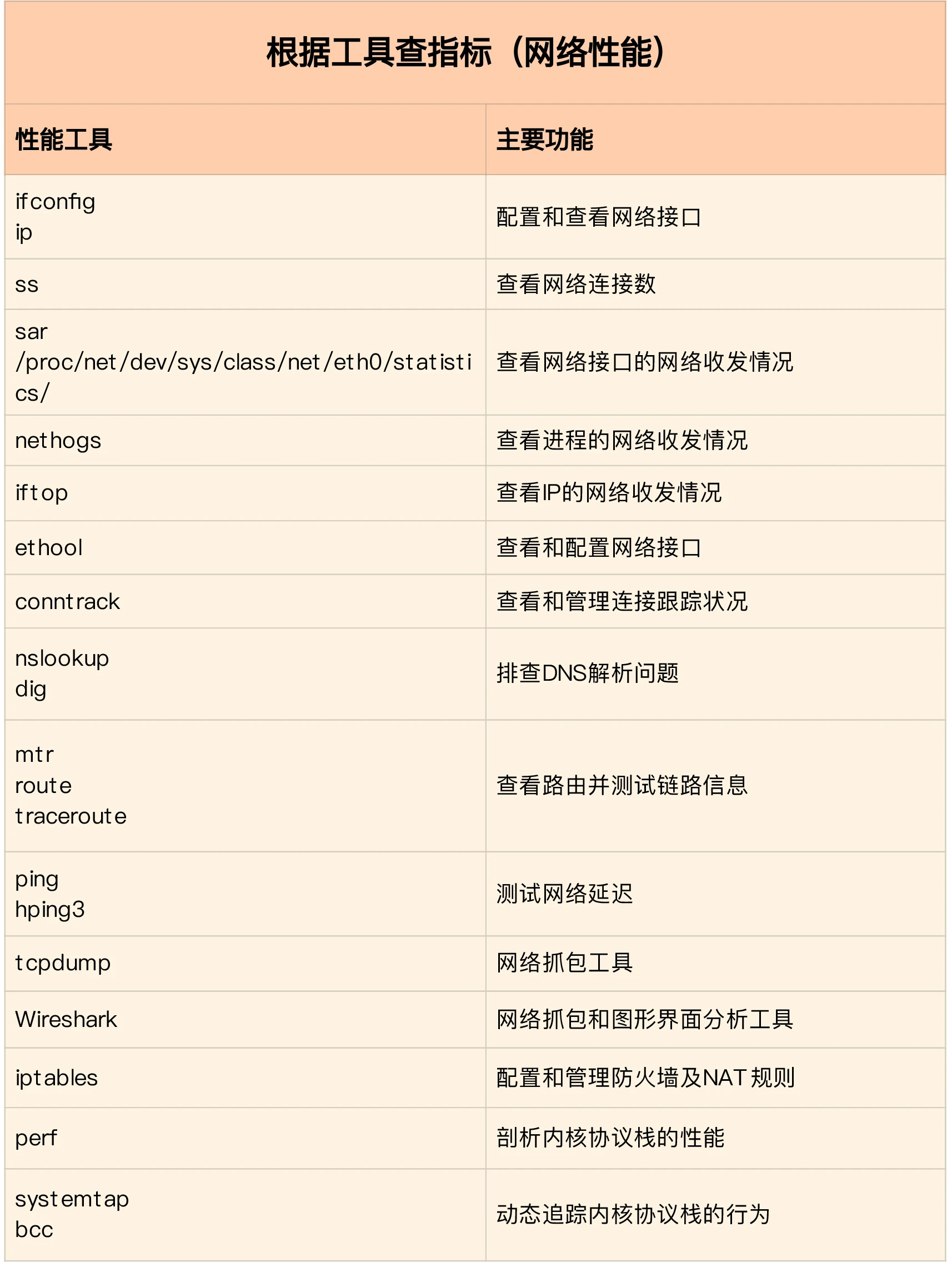
使用率、饱和度、IOPS、吞吐量以及响应时间等。这五个指标，是衡量磁盘性能的基本指标。

1. 使用率，是指磁盘处理 I/O 的时间百分比。过高的使用率（比如超过 80%），通常意味着磁盘 I/O 存在性能瓶颈。
2. 饱和度，是指磁盘处理 I/O 的繁忙程度。过高的饱和度，意味着磁盘存在严重的性能瓶颈。当饱和度为 100% 时，磁盘无法接受新的 I/O 请求。
3. IOPS（Input/Output Per Second），是指每秒的 I/O 请求数。
4. 吞吐量，是指每秒的 I/O 请求大小。
5. 响应时间，是指 I/O 请求从发出到收到响应的间隔时间。

不要孤立地去比较某一指标，而要结合读写比例、I/O 类型（随机还是连续）以及 I/O 的大小，综合来分析。举个例子，在数据库、大量小文件等这类随机读写比较多的场景中，IOPS 更能反映系统的整体性能；而在多媒体等顺序读写较多的场景中，吞吐量才更能反映系统的整体性能。

**网络**





**网络模型**

OSI 模型把网络互联的框架分为应用层、表示层、会话层、传输层、网络层、数据链路层以及物理层等七层，每个层负责不同的功能。其中，

应用层，负责为应用程序提供统一的接口。

表示层，负责把数据转换成兼容接收系统的格式。

会话层，负责维护计算机之间的通信连接。

传输层，负责为数据加上传输表头，形成数据包。

网络层，负责数据的路由和转发。

数据链路层，负责 MAC 寻址、错误侦测和改错。

物理层，负责在物理网络中传输数据帧。

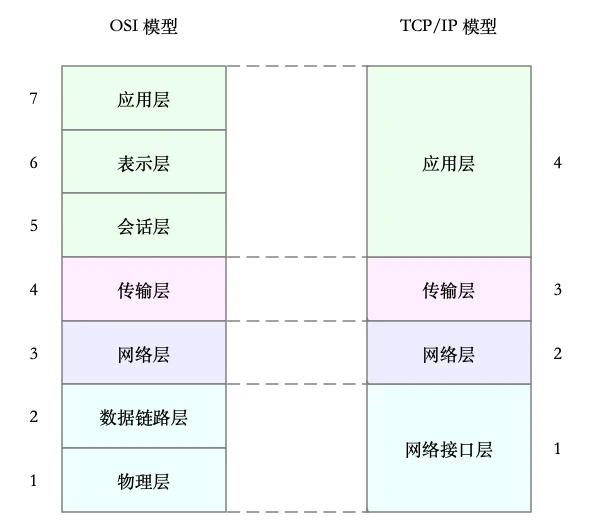
TCP/IP 模型，把网络互联的框架分为应用层、传输层、网络层、网络接口层等四层，其中，

应用层，负责向用户提供一组应用程序，比如 HTTP、FTP、DNS 等。

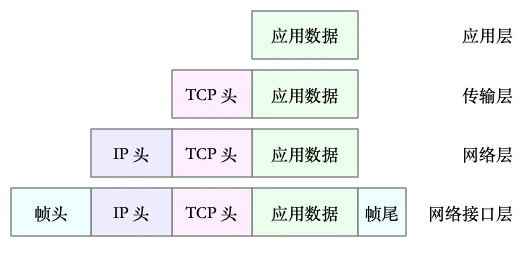
传输层，负责端到端的通信，比如 TCP、UDP 等。

网络层，负责网络包的封装、寻址和路由，比如 IP、ICMP 等。

网络接口层，负责网络包在物理网络中的传输，比如 MAC 寻址、错误侦测以及通过网卡传输网络帧等。



**Linux 网络栈**



传输层在应用程序数据前面增加了 TCP 头；

网络层在 TCP 数据包前增加了 IP 头；

而网络接口层，又在 IP 数据包前后分别增加了帧头和帧尾。

这些新增的头部和尾部，增加了网络包的大小，但我们都知道，物理链路中并不能传输任意大小的数据包。网络接口配置的最大传输单元（MTU），就规定了最大的 IP 包大小。在我们最常用的以太网中，MTU 默认值是 1500（这也是 Linux 的默认值）。一旦网络包超过 MTU 的大小，就会在网络层分片，以保证分片后的 IP 包不大于 MTU 值。显然，MTU 越大，需要的分包也就越少，自然，网络吞吐能力就越好。

**网络包的接收流程**

我们先来看网络包的接收流程。当一个网络帧到达网卡后，网卡会通过 DMA 方式，把这个网络包放到收包队列中；然后通过硬中断，告诉中断处理程序已经收到了网络包。接着，网卡中断处理程序会为网络帧分配内核数据结构（sk\_buff），并将其拷贝到 sk\_buff 缓冲区中；然后再通过软中断，通知内核收到了新的网络帧。接下来，内核协议栈从缓冲区中取出网络帧，并通过网络协议栈，从下到上逐层处理这个网络帧。比如，

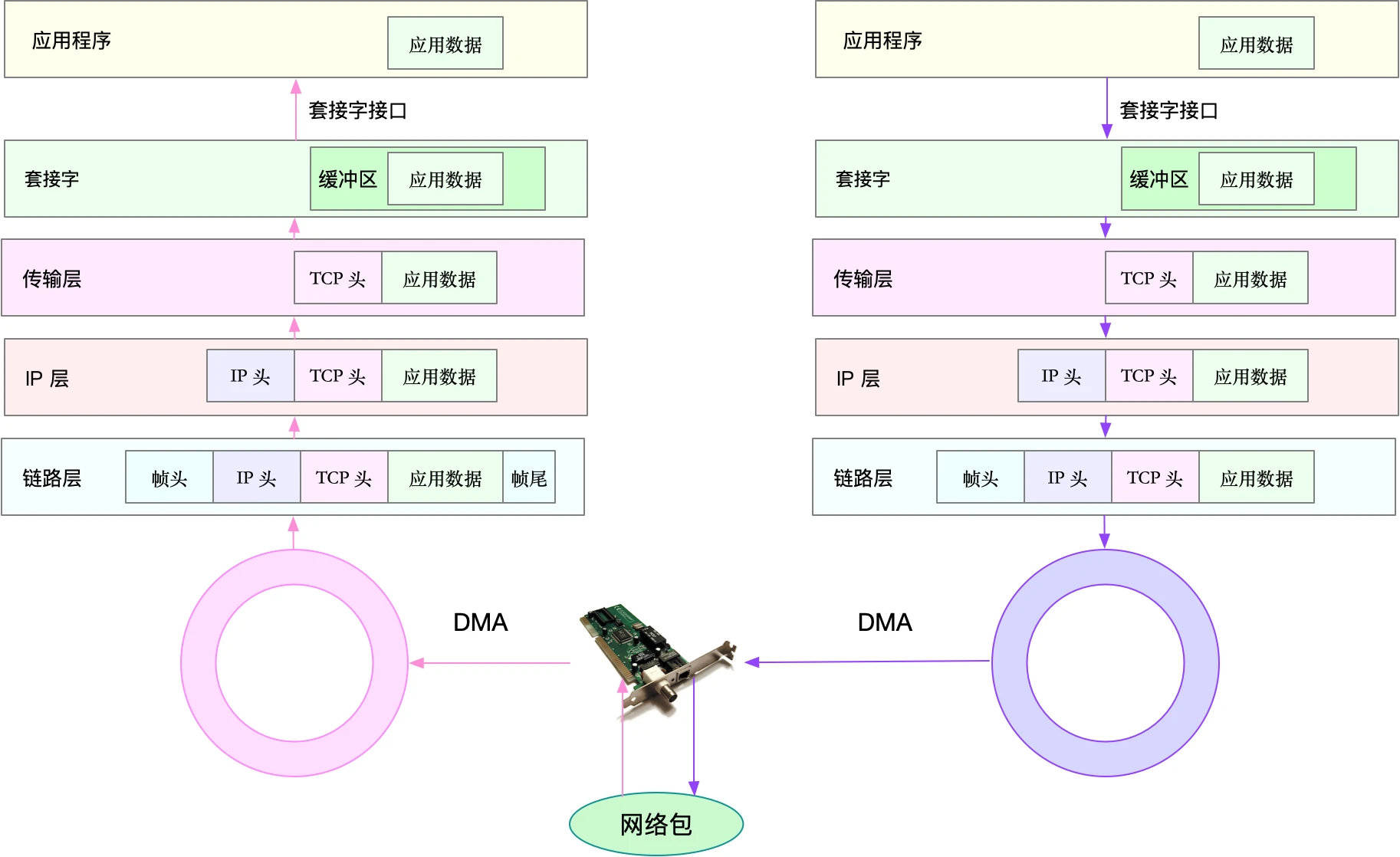
在链路层检查报文的合法性，找出上层协议的类型（比如 IPv4 还是 IPv6），再去掉帧头、帧尾，然后交给网络层。

网络层取出 IP 头，判断网络包下一步的走向，比如是交给上层处理还是转发。当网络层确认这个包是要发送到本机后，就会取出上层协议的类型（比如 TCP 还是 UDP），去掉 IP 头，再交给传输层处理。

传输层取出 TCP 头或者 UDP 头后，根据 < 源 IP、源端口、目的 IP、目的端口 > 四元组作为标识，找出对应的 Socket，并把数据拷贝到 Socket 的接收缓存中。

最后，应用程序就可以使用 Socket 接口，读取到新接收到的数据了。

为了更清晰表示这个流程，我画了一张图，这张图的左半部分表示接收流程，而图中的粉色箭头则表示网络包的处理路径。



**网络包的发送流程**

了解网络包的接收流程后，就很容易理解网络包的发送流程。网络包的发送流程就是上图的右半部分，很容易发现，网络包的发送方向，正好跟接收方向相反。

首先，应用程序调用 Socket API（比如 sendmsg）发送网络包。由于这是一个系统调用，所以会陷入到内核态的套接字层中。套接字层会把数据包放到 Socket 发送缓冲区中。

接下来，网络协议栈从 Socket 发送缓冲区中，取出数据包；再按照 TCP/IP 栈，从上到下逐层处理。比如，传输层和网络层，分别为其增加 TCP 头和 IP 头，执行路由查找确认下一跳的 IP，并按照 MTU 大小进行分片。

分片后的网络包，再送到网络接口层，进行物理地址寻址，以找到下一跳的 MAC 地址。然后添加帧头和帧尾，放到发包队列中。

这一切完成后，会有软中断通知驱动程序：发包队列中有新的网络帧需要发送。

最后，驱动程序通过 DMA ，从发包队列中读出网络帧，并通过物理网卡把它发送出去。

**性能指标**

我们通常用带宽、吞吐量、延时、PPS（Packet Per Second）等指标衡量网络的性能。

带宽，表示链路的最大传输速率，单位通常为 b/s （比特 / 秒）。

吞吐量，表示单位时间内成功传输的数据量，单位通常为 b/s（比特 / 秒）或者 B/s（字节 / 秒）。吞吐量受带宽限制，而吞吐量 / 带宽，也就是该网络的使用率。

延时，表示从网络请求发出后，一直到收到远端响应，所需要的时间延迟。在不同场景中，这一指标可能会有不同含义。比如，它可以表示，建立连接需要的时间（比如 TCP 握手延时），或一个数据包往返所需的时间（比如 RTT）。

PPS，是 Packet Per Second（包 / 秒）的缩写，表示以网络包为单位的传输速率。PPS 通常用来评估网络的转发能力，比如硬件交换机，通常可以达到线性转发（即 PPS 可以达到或者接近理论最大值）。而基于 Linux 服务器的转发，则容易受网络包大小的影响。

**性能测试**

**转发性能**

可以用linux pktgen 来测试。

**TCP/UDP 性能**

使用iperf 或者 netperf

yum -y install iperf3

然后，在目标机器上启动 iperf 服务端：

# -s表示启动服务端，-i表示汇报间隔，-p表示监听端口

$ iperf3 -s -i 1 -p 10000

在另一台机器上运行 iperf 客户端，运行测试：

# -c表示启动客户端，192.168.0.30为目标服务器的IP

# -b表示目标带宽(单位是bits/s)

# -t表示测试时间

# -P表示并发数，-p表示目标服务器监听端口

$ iperf3 -c 192.168.0.30 -b 1G -t 15 -P 2 -p 10000

**http性能**

要测试 HTTP 的性能，也有大量的工具可以使用，比如 ab、webbench 等，都是常用的 HTTP 压力测试工具。其中，ab 是 Apache 自带的 HTTP 压测工具，主要测试 HTTP 服务的每秒请求数、请求延迟、吞吐量以及请求延迟的分布情况等。

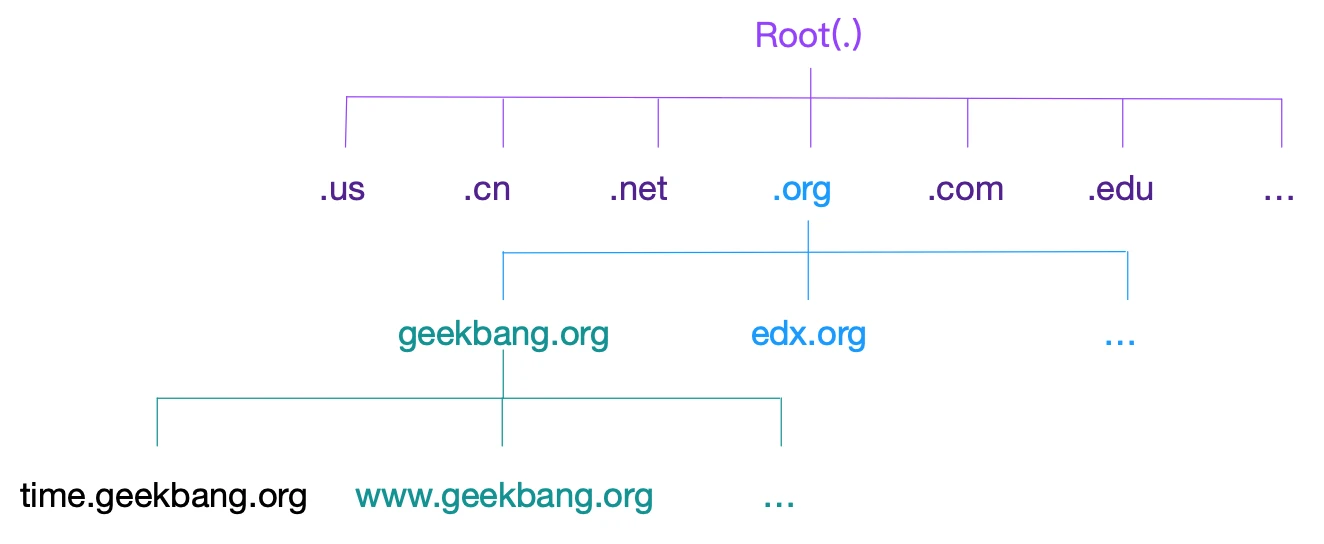
**应用负载性能**

为了得到应用程序的实际性能，就要求性能工具本身可以模拟用户的请求负载，而 iperf、ab 这类工具就无能为力了。幸运的是，我们还可以用 wrk、TCPCopy、Jmeter 或者 LoadRunner 等实现这个目标。

**域名与 DNS 解析**

DNS（Domain Name System），即域名系统，是互联网中最基础的一项服务，主要提供域名和 IP 地址之间映射关系的查询服务。

域名我们本身都比较熟悉，由一串用点分割开的字符组成，被用作互联网中的某一台或某一组计算机的名称，目的就是为了方便识别，互联网中提供各种服务的主机位置。要注意，域名是全球唯一的，需要通过专门的域名注册商才可以申请注册。为了组织全球互联网中的众多计算机，域名同样用点来分开，形成一个分层的结构。而每个被点分割开的字符串，就构成了域名中的一个层级，并且位置越靠后，层级越高。我们以极客时间的网站 time.geekbang.org 为例，来理解域名的含义。这个字符串中，最后面的 org 是顶级域名，中间的 geekbang 是二级域名，而最左边的 time 则是三级域名。



在域名解析的过程中，所有域名都以点结束

域名解析服务（DNS），而对应的服务器就是域名服务器，网络协议则是 DNS 协议。这里注意，DNS 协议在 TCP/IP 栈中属于应用层，不过实际传输还是基于 UDP 或者 TCP 协议（UDP 居多） ，并且域名服务器一般监听在端口 53 上。

防ddos攻击 sync 攻击

1. 半开状态的连接数是有限制的，执行下面的命令，你就可以看到，默认的半连接容量只有 256：

$ sysctl net.ipv4.tcp\_max\_syn\_backlognet.ipv4.tcp\_max\_syn\_backlog = 256

你可以用下面的命令，将其增大为 1024：$ sysctl -w net.ipv4.tcp\_max\_syn\_backlog=1024

1. 连接每个 SYN\_RECV 时，如果失败的话，内核还会自动重试，并且默认的重试次数是 5 次。你可以执行下面的命令，将其减小为 1 次：

$ sysctl -w net.ipv4.tcp\_synack\_retries=1net.ipv4.tcp\_synack\_retries = 1

1. TCP SYN Cookies 也是一种专门防御 SYN Flood 攻击的方法。SYN Cookies 基于连接信息（包括源地址、源端口、目的地址、目的端口等）以及一个加密种子（如系统启动时间），计算出一个哈希值（SHA1），这个哈希值称为 cookie。然后，这个 cookie 就被用作序列号，来应答 SYN+ACK 包，并释放连接状态。当客户端发送完三次握手的最后一次 ACK 后，服务器就会再次计算这个哈希值，确认是上次返回的 SYN+ACK 的返回包，才会进入 TCP 的连接状态。因而，开启 SYN Cookies 后，就不需要维护半开连接状态了，进而也就没有了半连接数的限制。注意，开启 TCP syncookies 后，内核选项 net.ipv4.tcp\_max\_syn\_backlog 也就无效了。你可以通过下面的命令，开启 TCP SYN Cookies：

$ sysctl -w net.ipv4.tcp\_syncookies=1

net.ipv4.tcp\_syncookies = 1

**网络延迟**

网络数据传输所用的时间。 我们更常用的是双向的往返通信延迟，比如 ping 测试的结果，就是往返延时 RTT（Round-Trip Time）。

很多网络服务会把 ICMP 禁止掉，这也就导致我们无法用 ping ，来测试网络服务的可用性和往返延时。这时，你可以用 traceroute 或 hping3 的 TCP 和 UDP 模式，来获取网络延迟。

-c表示发送3次请求，-S表示设置TCP SYN，-p表示端口号为80

$ hping3 -c 3 -S -p 80 baidu.com

# --tcp表示使用TCP协议，-p表示端口号，-n表示不对结果中的IP地址执行反向域名解析$ traceroute --tcp -p 80 -n baidu.com

**NAT**

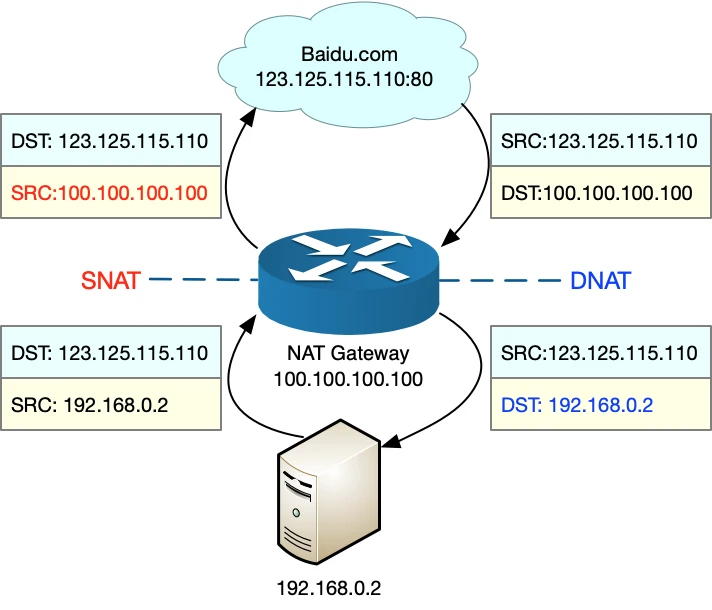
**nat 原理**

NAT 技术可以重写 IP 数据包的源 IP 或者目的 IP，被普遍地用来解决公网 IP 地址短缺的问题。它的主要原理就是，网络中的多台主机，通过共享同一个公网 IP 地址，来访问外网资源。同时，由于 NAT 屏蔽了内网网络，自然也就为局域网中的机器提供了安全隔离。

你既可以在支持网络地址转换的路由器（称为 NAT 网关）中配置 NAT，也可以在 Linux 服务器中配置 NAT。如果采用第二种方式，Linux 服务器实际上充当的是“软”路由器的角色。

NAT 的主要目的，是实现地址转换。根据实现方式的不同，NAT 可以分为三类：

1. 静态 NAT，即内网 IP 与公网 IP 是一对一的永久映射关系；
2. 动态 NAT，即内网 IP 从公网 IP 池中，动态选择一个进行映射；
3. 网络地址端口转换 NAPT（Network Address and Port Translation），即把内网 IP 映射到公网 IP 的不同端口上，让多个内网 IP 可以共享同一个公网 IP 地址。NAPT 是目前最流行的 NAT 类型，我们在 Linux 中配置的 NAT 也是这种类型。而根据转换方式的不同，我们又可以把 NAPT 分为三类。
   1. 第一类是源地址转换 SNAT，即目的地址不变，只替换源 IP 或源端口。SNAT 主要用于，多个内网 IP 共享同一个公网 IP ，来访问外网资源的场景。
   2. 第二类是目的地址转换 DNAT，即源 IP 保持不变，只替换目的 IP 或者目的端口。DNAT 主要通过公网 IP 的不同端口号，来访问内网的多种服务，同时会隐藏后端服务器的真实 IP 地址。
   3. 第三类是双向地址转换，即同时使用 SNAT 和 DNAT。当接收到网络包时，执行 DNAT，把目的 IP 转换为内网 IP；而在发送网络包时，执行 SNAT，把源 IP 替换为外部 IP。



**iptables 与 NAT**

如何在 Linux 中实现 NAT 的功能呢

Linux 内核提供的 Netfilter 框架，允许对网络数据包进行修改（比如 NAT）和过滤（比如防火墙）。在这个基础上，iptables、ip6tables、ebtables 等工具，又提供了更易用的命令行接口，以便系统管理员配置和管理 NAT、防火墙的规则。

iptables 中，Linux 支持 4 种表，包括 filter（用于过滤）、nat（用于 NAT）、mangle（用于修改分组数据） 和 raw（用于原始数据包）等。每个表中可以包含多条链，nat 表中，内置 PREROUTING、POSTROUTING、OUTPUT 等

在 Linux 中实现 NAT 的功能，主要是在 nat 表进行操作。而 nat 表内置了三个链：

PREROUTING，用于路由判断前所执行的规则，比如，对接收到的数据包进行 DNAT。

POSTROUTING，用于路由判断后所执行的规则，比如，对发送或转发的数据包进行 SNAT 或 MASQUERADE。

OUTPUT，类似于 PREROUTING，但只处理从本机发送出去的包。

**SNAT**

根据刚才内容，我们知道，SNAT 需要在 nat 表的 POSTROUTING 链中配置。我们常用两种方式来配置它。

第一种方法，是为一个子网统一配置 SNAT，并由 Linux 选择默认的出口 IP。这实际上就是经常说的 MASQUERADE：

$ iptables -t nat -A POSTROUTING -s 192.168.0.0/16 -j MASQUERADE

第二种方法，是为具体的 IP 地址配置 SNAT，并指定转换后的源地址：

$ iptables -t nat -A POSTROUTING -s 192.168.0.2 -j SNAT --to-source 100.100.100.100

**DNAT**

再来看 DNAT，显然，DNAT 需要在 nat 表的 PREROUTING 或者 OUTPUT 链中配置，其中， PREROUTING 链更常用一些（因为它还可以用于转发的包）。

$ iptables -t nat -A PREROUTING -d 100.100.100.100 -j DNAT --to-destination 192.168.0.2

**双向地址转换**

双向地址转换，就是同时添加 SNAT 和 DNAT 规则，为公网 IP 和内网 IP 实现一对一的映射关系，即：

$ iptables -t nat -A POSTROUTING -s 192.168.0.2 -j SNAT --to-source 100.100.100.100

$ iptables -t nat -A PREROUTING -d 100.100.100.100 -j DNAT --to-destination 192.168.0.2

在使用 iptables 配置 NAT 规则时，Linux 需要转发来自其他 IP 的网络包，所以你千万不要忘记开启 Linux 的 IP 转发功能。

$ sysctl net.ipv4.ip\_forward

net.ipv4.ip\_forward = 1

**命令**

**stress**

stress 是一个 Linux 系统压力测试工具，这里我们用作异常进程模拟平均负载升高的场景。

stress --cpu 1 --timeout 600 模拟一个 CPU 使用率 100% 的场景

stress -i 1 --timeout 600 模拟io压力

stress -c 8 --timeout 600 // 模拟8个进程

**sysstat**

sysstat 包含了常用的 Linux 性能工具，用来监控和分析系统的性能。包含mpstat、pidstat等命令

**mpstat**

mpstat 是一个常用的多核 CPU 性能分析工具，用来实时查看每个 CPU 的性能指标，以及所有 CPU 的平均指标。

mpstat -P ALL 5 //-P ALL 表示监控所有CPU，后面数字5表示间隔5秒后输出一组数据

**pidstat**

pidstat 是一个常用的进程性能分析工具，用来实时查看进程的 CPU、内存、I/O 以及上下文切换等性能指标。

watch -d uptime //-d 参数表示高亮显示变化的区域. 持续观察uptime 的输出

pidstat -u 5 1 // -u 输出cpu使用率， 间隔5秒输出1组数据

pidstat -w 5 # 每隔5秒输出1组数据. -w 选项，你就可以查看每个进程上下文切换的情况

这个结果中有两列内容是我们的重点关注对象。一个是 cswch ，表示每秒自愿上下文切换（voluntary context switches）的次数，另一个则是 nvcswch ，表示每秒非自愿上下文切换（non voluntary context switches）的次数。

所谓自愿上下文切换，是指进程无法获取所需资源，导致的上下文切换。比如说， I/O、内存等系统资源不足时，就会发生自愿上下文切换。

而非自愿上下文切换，则是指进程由于时间片已到等原因，被系统强制调度，进而发生的上下文切换。比如说，大量进程都在争抢 CPU 时，就容易发生非自愿上下文切换。

$ pidstat -wt 1 # -wt 参数表示输出线程的上下文切换指标

pidstat -d -p 4344 1 3 -d 展示 I/O 统计数据，-p 指定进程号，间隔 1 秒输出 3 组数据

**vmstat**

vmstat 是一个常用的系统性能分析工具，主要用来分析系统的内存使用情况，也常用来分析 CPU 上下文切换和中断的次数。 vmstat 只给出了系统总体的上下文切换情况，要想查看每个进程的详细情况，就需要使用我们前面提到过的 pidstat 了

vmstat 5 //每隔5秒输出1组数据

cs（context switch）是每秒上下文切换的次数。

in（interrupt）则是每秒中断的次数。

r（Running or Runnable）是就绪队列的长度，也就是正在运行和等待 CPU 的进程数。

b（Blocked）则是处于不可中断睡眠状态的进程数。

buff 和 cache 就是我们前面看到的 Buffers 和 Cache，单位是 KB。

bi 和 bo 则分别表示块设备读取和写入的大小，单位为块 / 秒。因为 Linux 中块的大小是 1KB，所以这个单位也就等价于 KB/s。

系统的就绪队列过长，也就是正在运行和等待 CPU 的进程数过多，导致了大量的上下文切换，而上下文切换又导致了系统 CPU 的占用率升高

**dstat**

可以同时查看 CPU 和 I/O 这两种资源的使用情况，便于对比分析

**sysbench**

是一个多线程的基准测试工具，一般用来评估不同系统参数下的数据库负载情况。

# 以10个线程运行5分钟的基准测试，模拟多线程切换的问题

$ sysbench --threads=10 --max-time=300 threads run

**top**

top 显示了系统总体的 CPU 和内存使用情况，以及各个进程的资源使用情况

按1 可以查看各个cpu 的使用率

top 并没有细分进程的用户态 CPU 和内核态 CPU

**命令参数：**

-b 批处理

-c 显示完整的治命令

-I 忽略失效过程

-s 保密模式

-S 累积模式

-i<时间> 设置间隔时间

-u<用户名> 指定用户名

-p<进程号> 指定进程

-n<次数> 循环显示的次数

**输出**

**第一行，任务队列信息，同 uptime 命令的执行结果，具体参数说明情况如下：**

14:06:23 — 当前系统时间

up 70 days, 16:44 — 系统已经运行了70天16小时44分钟（在这期间系统没有重启过的吆！）

2 users — 当前有2个用户登录系统

load average: 1.15, 1.42, 1.44 — load average后面的三个数分别是1分钟、5分钟、15分钟的负载情况。

load average数据是每隔5秒钟检查一次活跃的进程数，然后按特定算法计算出的数值。如果这个数除以逻辑CPU的数量，结果高于5的时候就表明系统在超负荷运转了。

**第二行，Tasks — 任务（进程），具体信息说明如下：**

系统现在共有206个进程，其中处于运行中的有1个，205个在休眠（sleep），stoped状态的有0个，zombie状态（僵尸）的有0个。

**第三行，cpu状态信息，具体属性说明如下：**

5.9%us — 用户空间占用CPU的百分比。

3.4% sy — 内核空间占用CPU的百分比。

0.0% ni — 改变过优先级的进程占用CPU的百分比

90.4% id — 空闲CPU百分比

0.0% wa — IO等待占用CPU的百分比

0.0% hi — 硬中断（Hardware IRQ）占用CPU的百分比

0.2% si — 软中断（Software Interrupts）占用CPU的百分比

**备注：在这里CPU的使用比率和windows概念不同，需要理解linux系统用户空间和内核空间的相关知识！**

**第四行,内存状态，具体信息如下：**

32949016k total — 物理内存总量（32GB）

14411180k used — 使用中的内存总量（14GB）

18537836k free — 空闲内存总量（18GB）

169884k buffers — 缓存的内存量 （169M）

**第五行，swap交换分区信息，具体信息说明如下：**

32764556k total — 交换区总量（32GB）

0k used — 使用的交换区总量（0K）

32764556k free — 空闲交换区总量（32GB）

3612636k cached — 缓冲的交换区总量（3.6GB）

备注：

第四行中使用中的内存总量（used）指的是现在系统内核控制的内存数，空闲内存总量（free）是内核还未纳入其管控范围的数量。纳入内核管理的内存不见得都在使用中，还包括过去使用过的现在可以被重复利用的内存，内核并不把这些可被重新使用的内存交还到free中去，因此在linux上free内存会越来越少，但不用为此担心。

如果出于习惯去计算可用内存数，这里有个近似的计算公式：第四行的free + 第四行的buffers + 第五行的cached，按这个公式此台服务器的可用内存：18537836k +169884k +3612636k = 22GB左右。

对于内存监控，在top里我们要时刻监控第五行swap交换分区的used，如果这个数值在不断的变化，说明内核在不断进行内存和swap的数据交换，这是真正的内存不够用了。

**第六行，空行。**

**第七行以下：各进程（任务）的状态监控，项目列信息说明如下：**

PID — 进程id

USER — 进程所有者

PR — 进程优先级

NI — nice值。负值表示高优先级，正值表示低优先级

VIRT — 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES

RES — 进程使用的、未被换出的物理内存大小，但不包括 Swap 和共享内存

，单位kb。RES=CODE+DATA

SHR — 共享内存大小，单位kb 。比如与其他进程共同使用的共享内存、加载的动态链接库以及程序的代码段等。

S — 进程状态。D=不可中断的睡眠状态 R=运行 S=睡眠 T=跟踪/停止 Z=僵尸进程

%CPU — 上次更新到现在的CPU时间占用百分比

%MEM — 进程使用的物理内存百分比

TIME+ — 进程使用的CPU时间总计，单位1/100秒

COMMAND — 进程名称（命令名/命令行）

注意点

第一， 虚拟内存通常并不会全部分配物理内存。从上面的输出，你可以发现每个进程的虚拟内存都比常驻内存大得多。

第二， 第二，共享内存 SHR 并不一定是共享的，比方说，程序的代码段、非共享的动态链接库，也都算在 SHR 里。当然，SHR 也包括了进程间真正共享的内存。所以在计算多个进程的内存使用时，不要把所有进程的 SHR 直接相加得出结果。

**perf**

perf 是 Linux 2.6.31 以后内置的性能分析工具。它以性能事件采样为基础，不仅可以分析系统的各种事件和内核性能，还可以用来分析指定应用程序的性能问题。

perf top 它能够实时显示占用 CPU 时钟最多的函数或者指令，因此可以用来查找热点函数，使用界面如下所示

$ **perf top**

Samples: 833 of event 'cpu-clock', Event count (approx.): 97742399

Overhead Shared Object Symbol

7.28% perf [.] 0x00000000001f78a4

4.72% [kernel] [k] vsnprintf

4.32% [kernel] [k] module\_get\_kallsym

3.65% [kernel] [k] \_raw\_spin\_unlock\_irqrestore

...

输出结果中，第一行包含三个数据，分别是采样数（Samples）、事件类型（event）和事件总数量（Event count）。比如这个例子中，perf 总共采集了 833 个 CPU 时钟事件，而总事件数则为 97742399。另外，采样数需要我们特别注意。如果采样数过少（比如只有十几个），那下面的排序和百分比就没什么实际参考价值了。再往下看是一个表格式样的数据，每一行包含四列，分别是：

第一列 Overhead ，是该符号的性能事件在所有采样中的比例，用百分比来表示。

第二列 Shared ，是该函数或指令所在的动态共享对象（Dynamic Shared Object），如内核、进程名、动态链接库名、内核模块名等。

第三列 Object ，是动态共享对象的类型。比如 [.] 表示用户空间的可执行程序、或者动态链接库，而 [k] 则表示内核空间。

最后一列 Symbol 是符号名，也就是函数名。当函数名未知时，用十六进制的地址来表示。还是以上面的输出为例，我们可以看到，占用 CPU 时钟最多的是 perf 工具自身，不过它的比例也只有 7.28%，说明系统并没有 CPU 性能问题。 perf top 的使用你应该很清楚了吧。

**perf record/ perf report**

perf top 虽然实时展示了系统的性能信息，但它的缺点是并不保存数据，也就无法用于离线或者后续的分析。perf record 则提供了保存数据的功能，保存后的数据，需要你用 perf report 解析展示

-p 指定进程

-g 开启调用关系的采样，方便我们根据调用链来分析性能问题

**perf 显示的是16进制地址而不是函数名**

解决方法：

第一个方法，在容器外面构建相同路径的依赖库。这种方法从原理上可行，不推荐

第二个方法，在容器内部运行 perf。不过，这需要容器运行在特权模式下，但实际的应用程序往往只以普通容器的方式运行。所以，容器内部一般没有权限执行 perf 分析。

第三个方法，指定符号路径为容器文件系统的路径。

第四个方法，在容器外面把分析纪录保存下来，再去容器里查看结果。这样，库和符号的路径也就都对了。

**ab**

ab（apache bench）是一个常用的 HTTP 服务性能测试工具

# 并发10个请求测试Nginx性能，总共测试100个请求

ab -c 10 -n 100 <http://192.168.0.10:10000/>

-t 设置时长 # -c表示并发请求数为1000，-n表示总的请求数为100

**siege**

siege is a HTTP/FTP load tester and benchmarking utility

siege -c 254 -d 0.8 -i -r 900000000 -f request.txt > res.txt

-c 模拟254 个用户同时访问

-d 间隔0.8秒访问

-f 指定url 文件， -i 从文件中随机访问

-r 跑多少次

**dd**

dd if=/dev/urandom of=/tmp/file bs=1M count=500 //通过读取随机设备，生成一个 500MB 大小的文件

dd if=/dev/zero of=/dev/vdb bs=1M count=1 //初始化一个磁盘

**sar**

是一个系统活动报告工具，既可以实时查看系统的当前活动，又可以配置保存和报告历史统计数据。

sar -n DEV 1 # -n DEV 表示显示网络收发的报告，间隔1秒输出一组数据

对于 sar 的输出界面，我先来简单介绍一下，从左往右依次是：第一列：表示报告的时间。第二列：IFACE 表示网卡。第三、四列：rxpck/s 和 txpck/s 分别表示每秒接收、发送的网络帧数，也就是 PPS。第五、六列：rxkB/s 和 txkB/s 分别表示每秒接收、发送的千字节数，也就是 BPS。

# 间隔1秒输出一组数据# -r表示显示内存使用情况，-S表示显示Swap使用情况$

sar -r -S 1

输出

kbcommit，表示当前系统负载需要的内存。它实际上是为了保证系统内存不溢出，对需要内存的估计值。%commit，就是这个值相对总内存的百分比。

kbactive，表示活跃内存，也就是最近使用过的内存，一般不会被系统回收。

kbinact，表示非活跃内存，也就是不常访问的内存，有可能会被系统回收。

# 数字1表示每隔1秒输出一组数据

sar -n DEV 1

-n 参数就可以查看网络的统计信息，比如网络接口（DEV）、网络接口错误（EDEV）、TCP、UDP、ICMP 等等

查看系统当前的网络吞吐量和 PPS

输出：

rxpck/s 和 txpck/s 分别是接收和发送的 PPS，单位为包 / 秒。

rxkB/s 和 txkB/s 分别是接收和发送的吞吐量，单位是 KB/ 秒。

rxcmp/s 和 txcmp/s 分别是接收和发送的压缩数据包数，单位是包 / 秒。

%ifutil 是网络接口的使用率，即半双工模式下为 (rxkB/s+txkB/s)/Bandwidth，而全双工模式下为 max(rxkB/s, txkB/s)/Bandwidth。

**ethtool**

Bandwidth 可以用 ethtool 来查询

**hping3**

是一个可以构造 TCP/IP 协议数据包的工具，可以对系统进行安全审计、防火墙测试等。

hping3 模拟 SYN FLOOD

运行 hping3 命令，来模拟 Nginx 的客户端请求：

# -S参数表示设置TCP协议的SYN（同步序列号），-p表示目的端口为80

# -i u100表示每隔100微秒发送一个网络帧

# 注：如果你在实践过程中现象不明显，可以尝试把100调小，比如调成10甚至1

$ hping3 -S -p 80 -i u100 192.168.0.30

--rand-source 选项，来随机化源 IP

**free**

free 显示的是整个系统的内存使用情况.如果你想查看进程的内存使用情况，可以用 top 或者 ps 等工具

第一列，total 是总内存大小；

第二列，used 是已使用内存的大小，包含了共享内存；

第三列，free 是未使用内存的大小；

第四列，shared 是共享内存的大小；

第五列，buff/cache 是缓存和缓冲区的大小；

最后一列，available 是新进程可用内存的大小。available 不仅包含未使用内存，还包括了可回收的缓存，所以一般会比未使用内存更大。不过，并不是所有缓存都可以回收，因为有些缓存可能正在使用中。

**tcpdump**

是一个常用的网络抓包工具，常用来分析各种网络问题。

tcpdump -i eth0 -n tcp port 80 # -i eth0 只抓取eth0网卡，-n不解析协议名和主机名# tcp port 80表示只抓取tcp协议并且端口号为80的网络帧

从 tcpdump 的输出中，你可以发现192.168.0.2.18238 > 192.168.0.30.80 ，表示网络帧从 192.168.0.2 的 18238 端口发送到 192.168.0.30 的 80 端口，也就是从运行 hping3 机器的 18238 端口发送网络帧，目的为 Nginx 所在机器的 80 端口。Flags [S] 则表示这是一个 SYN 包。

**bcc-tools**

**cachestat**

yum -y install bcc-tools

export PATH=$PATH:/usr/share/bcc/tools

centos7.3 以上可用，不然得升级内核

提供了整个操作系统缓存的读写命中情况。

**cachetop**

yum -y install bcc-tools

可以在<https://github.com/iovisor/bcc>下载到源码

提供了每个进程的缓存命中情况。

**strace**

最常用的跟踪进程系统调用的工具

strace -p pid

-f 跟踪线程

**df**

查看文件系统的磁盘空间使用情况

df -h

查看索引节点的使用情况

df -i

**slabtop**

查看各种缓存

**fio**

来测试磁盘的 IOPS、吞吐量以及响应时间等核心指标

# 随机读

fio -name=randread -direct=1 -iodepth=64 -rw=randread -ioengine=libaio -bs=4k -size=1G -numjobs=1 -runtime=1000 -group\_reporting -filename=/dev/sdb

# 随机写

fio -name=randwrite -direct=1 -iodepth=64 -rw=randwrite -ioengine=libaio -bs=4k -size=1G -numjobs=1 -runtime=1000 -group\_reporting -filename=/dev/sdb

# 顺序读

fio -name=read -direct=1 -iodepth=64 -rw=read -ioengine=libaio -bs=4k -size=1G -numjobs=1 -runtime=1000 -group\_reporting -filename=/dev/sdb

# 顺序写

fio -name=write -direct=1 -iodepth=64 -rw=write -ioengine=libaio -bs=4k -size=1G -numjobs=1 -runtime=1000 -group\_reporting -filename=/dev/sdb

direct，表示是否跳过系统缓存。上面示例中，我设置的 1 ，就表示跳过系统缓存。

iodepth，表示使用异步 I/O（asynchronous I/O，简称 AIO）时，同时发出的 I/O 请求上限。在上面的示例中，我设置的是 64。

rw，表示 I/O 模式。我的示例中， read/write 分别表示顺序读 / 写，而 randread/randwrite 则分别表示随机读 / 写。

ioengine，表示 I/O 引擎，它支持同步（sync）、异步（libaio）、内存映射（mmap）、网络（net）等各种 I/O 引擎。上面示例中，我设置的 libaio 表示使用异步 I/O。

bs，表示 I/O 的大小。示例中，我设置成了 4K（这也是默认值）。

filename，表示文件路径，当然，它可以是磁盘路径（测试磁盘性能），也可以是文件路径（测试文件系统性能）。示例中，我把它设置成了磁盘 /dev/sdb。不过注意，用磁盘路径测试写，会破坏这个磁盘中的文件系统，所以在使用前，你一定要事先做好数据备份。

**iostat**

iostat 是最常用的磁盘 I/O 性能观测工具，它提供了每个磁盘的使用率、IOPS、吞吐量等各种常见的性能指标，当然，这些指标实际上来自 /proc/diskstats。

# -d表示显示I/O性能指标，-x表示显示扩展统计（即所有I/O指标）

$ iostat -x -d 1



%util ，就是我们前面提到的磁盘 I/O 使用率；

r/s+ w/s ，就是 IOPS；

rkB/s+wkB/s ，就是吞吐量；

r\_await+w\_await ，就是响应时间。

**iotop**

它是一个类似于 top 的工具，你可以按照 I/O 大小对进程排序，然后找到 I/O 较大的那些进程

**lsof**

可以查看进程打开了哪些文件

lsof -p 8100

输出中文件描述符那列w表示以写打开，u表示读写

**filetop**

它是 bcc 软件包的一部分，基于 Linux 内核的 eBPF（extended Berkeley Packet Filters）机制，主要跟踪内核中文件的读写情况，并输出线程 ID（TID）、读写大小、读写类型以及文件名称。

**ps**

ps -efT | grep 514

查找514线程属于哪个进程

**ifconfig/ip**

查看网络接口的配置和状态

ifconfig eth0

ip -s addr show dev eth0

第一，网络接口的状态标志。ifconfig 输出中的 RUNNING ，或 ip 输出中的 LOWER\_UP ，都表示物理网络是连通的，即网卡已经连接到了交换机或者路由器中。如果你看不到它们，通常表示网线被拔掉了。

第二，MTU 的大小。MTU 默认大小是 1500，根据网络架构的不同（比如是否使用了 VXLAN 等叠加网络），你可能需要调大或者调小 MTU 的数值。

第三，网络接口的 IP 地址、子网以及 MAC 地址。这些都是保障网络功能正常工作所必需的，你需要确保配置正确。

第四，网络收发的字节数、包数、错误数以及丢包情况，特别是 TX 和 RX 部分的 errors、dropped、overruns、carrier 以及 collisions 等指标不为 0 时，通常表示出现了网络 I/O 问题。其中：

1. errors 表示发生错误的数据包数，比如校验错误、帧同步错误等；
2. dropped 表示丢弃的数据包数，即数据包已经收到了 Ring Buffer，但因为内存不足等原因丢包；
3. overruns 表示超限数据包数，即网络 I/O 速度过快，导致 Ring Buffer 中的数据包来不及处理（队列满）而导致的丢包；
4. carrier 表示发生 carrirer 错误的数据包数，比如双工模式不匹配、物理电缆出现问题等；
5. collisions 表示碰撞数据包数。

**netstat/ss**

查看套接字、网络栈、网络接口以及路由表的信息

ss -ltnp

netstat -nlp

接收队列（Recv-Q）和发送队列（Send-Q）需要你特别关注，它们通常应该是 0。当你发现它们不是 0 时，说明有网络包的堆积发生。

当然还要注意，在不同套接字状态下，它们的含义不同。当套接字处于连接状态（Established）时，Recv-Q 表示套接字缓冲还没有被应用程序取走的字节数（即接收队列长度）。而 Send-Q 表示还没有被远端主机确认的字节数（即发送队列长度）。当套接字处于监听状态（Listening）时，Recv-Q 表示全连接队列的长度。而 Send-Q 表示全连接队列的最大长度。

-s 显示协议栈信息

**ping**

测试远程主机的连通性和延时，而这基于 ICMP 协议

ping 的输出，可以分为两部分。

第一部分，是每个 ICMP 请求的信息，包括 ICMP 序列号（icmp\_seq）、TTL（生存时间，或者跳数）以及往返延时。

第二部分，则是三次 ICMP 请求的汇总。

**modeprobe**

modprobe可载入指定的个别模块，或是载入一组相依的模块。 内核模块

modprobe pktgen 加载pktgen模块

**iperf/netperf**

用来测试tcp/udp性能

yum -y install iperf3

然后，在目标机器上启动 iperf 服务端：

# -s表示启动服务端，-i表示汇报间隔，-p表示监听端口

$ iperf3 -s -i 1 -p 10000

在另一台机器上运行 iperf 客户端，运行测试：

# -c表示启动客户端，192.168.0.30为目标服务器的IP

# -b表示目标带宽(单位是bits/s)

# -t表示测试时间

# -P表示并发数，-p表示目标服务器监听端口

$ iperf3 -c 192.168.0.30 -b 1G -t 15 -P 2 -p 10000

nslookup/dig

查域名ip

dig +trace +nodnssec time.geekbang.org

nslookup -debug time.geekbang.org 增加调试输出

**traceroot**

# --tcp表示使用TCP协议，-p表示端口号，-n表示不对结果中的IP地址执行反向域名解析$ traceroute --tcp -p 80 -n baidu.com

用来测试网络延时

**/proc**

可以man proc 查看帮助文档

它们都是 CPU 使用率相关的重要指标，你还会在很多其他的性能工具中看到它们。下面，我来依次解读一下。

user（通常缩写为 us），代表用户态 CPU 时间。注意，它不包括下面的 nice 时间，但包括了 guest 时间。

nice（通常缩写为 ni），代表低优先级用户态 CPU 时间，也就是进程的 nice 值被调整为 1-19 之间时的 CPU 时间。这里注意，nice 可取值范围是 -20 到 19，数值越大，优先级反而越低。

system（通常缩写为 sys），代表内核态 CPU 时间。

idle（通常缩写为 id），代表空闲时间。注意，它不包括等待 I/O 的时间（iowait）。

iowait（通常缩写为 wa），代表等待 I/O 的 CPU 时间。

irq（通常缩写为 hi），代表处理硬中断的 CPU 时间。

softirq（通常缩写为 si），代表处理软中断的 CPU 时间。

steal（通常缩写为 st），代表当系统运行在虚拟机中的时候，被其他虚拟机占用的 CPU 时间。

guest（通常缩写为 guest），代表通过虚拟化运行其他操作系统的时间，也就是运行虚拟机的 CPU 时间。

guest\_nice（通常缩写为 gnice），代表以低优先级运行虚拟机的时间。

**/proc/interrupts**

/proc 实际上是 Linux 的一个虚拟文件系统，用于内核空间与用户空间之间的通信。/proc/interrupts 就是这种通信机制的一部分，提供了一个只读的中断使用情况。

watch -d cat /proc/interrupts # -d 参数表示高亮显示变化的区域

**/proc/stat**

提供的就是系统的 CPU 和任务统计信息

**/proc/[pid]/stat**

Linux 也给每个进程提供了运行情况的统计信息

/proc/softirqs 提供了软中断的运行情况；

/proc/interrupts 提供了硬中断的运行情况。

/proc/sys/vm/drop\_caches

# 清理文件页、目录项、Inodes等各种缓存

$ echo 3 > /proc/sys/vm/drop\_caches

这里的 /proc/sys/vm/drop\_caches ，就是通过 proc 文件系统修改内核行为的一个示例，写入 3 表示清理文件页、目录项、Inodes 等各种缓存。这几种缓存的区别你暂时不用管，后面我们都会讲到。

默认值0

To free pagecache, use  
echo 1 > /proc/sys/vm/drop\_caches  
to free dentries and inodes, use  
echo 2 > /proc/sys/vm/drop\_caches  
to free pagecache, dentries and inodes, use  
echo 3 >/proc/sys/vm/drop\_caches

/proc/slabinfo 查看各种缓存大小