面试官：如何优化你的程序

码砖杂役 极客重生 2022-09-15 21:05 Posted on 广东

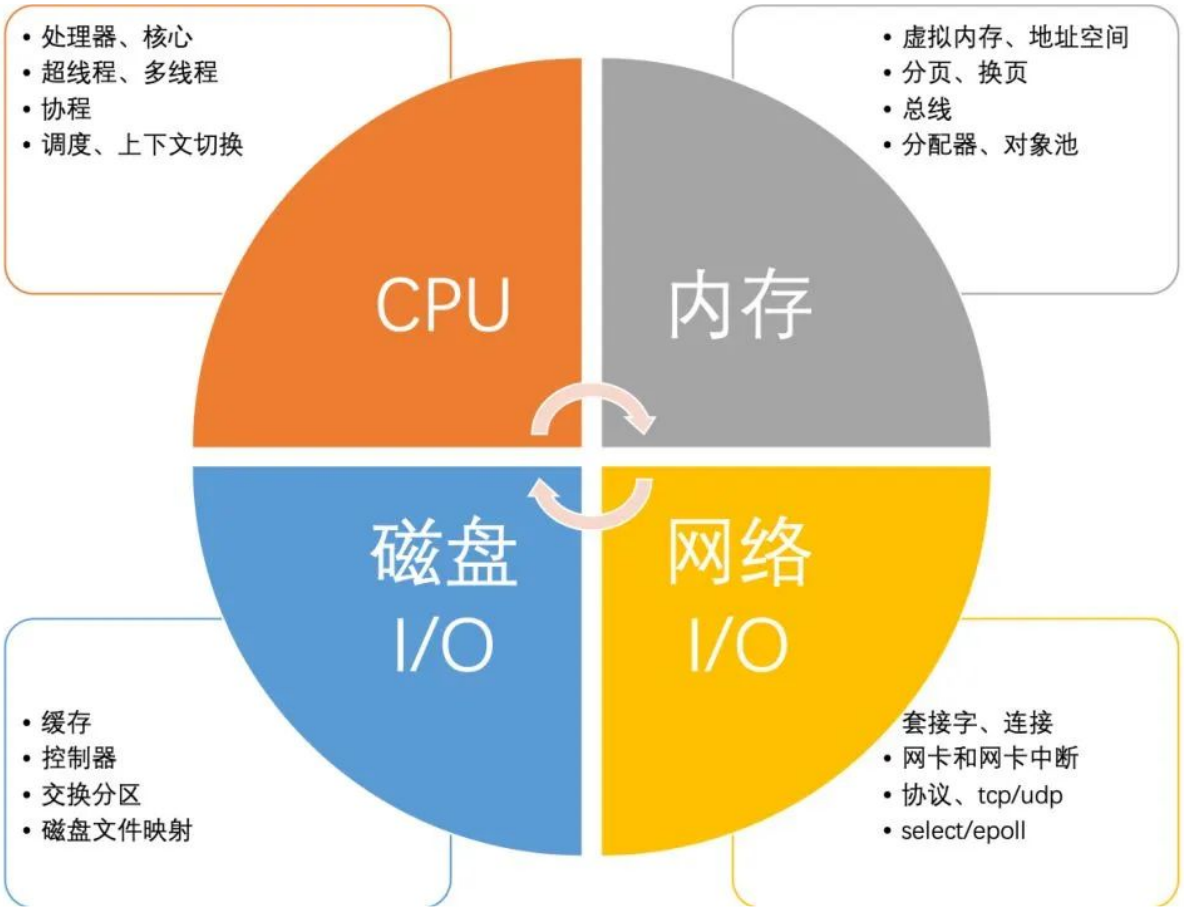
收录于合集

#深入理解Linux系统

49个

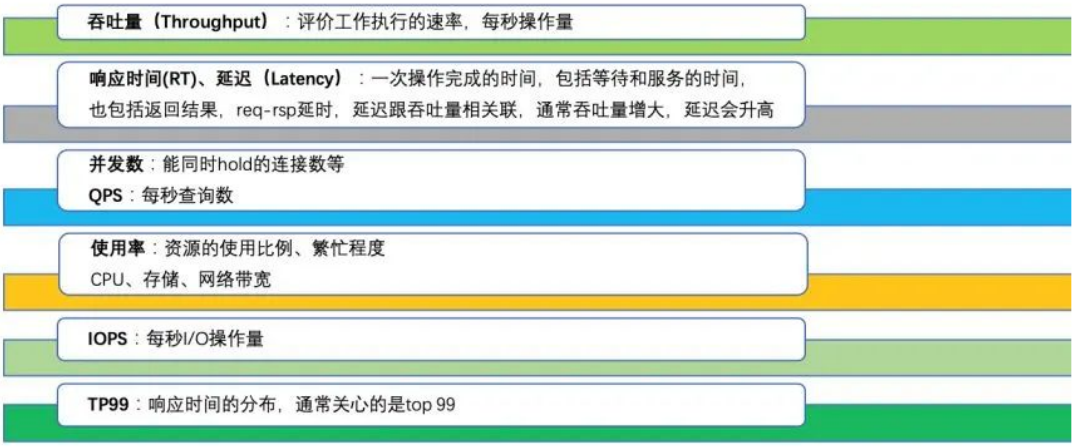
概要

关注&指标&度量，基础理论知识，工具&方法，最佳实践，参考资料



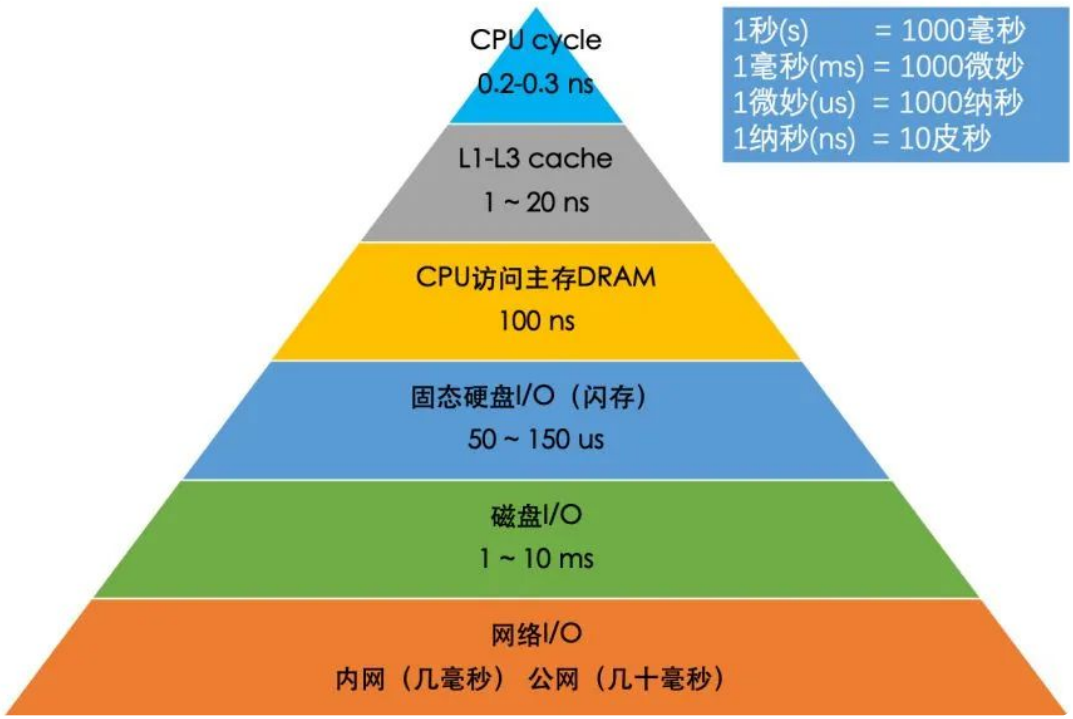
性能优化关注：CPU、内存、磁盘IO、网络IO等四个方面。

性能指标

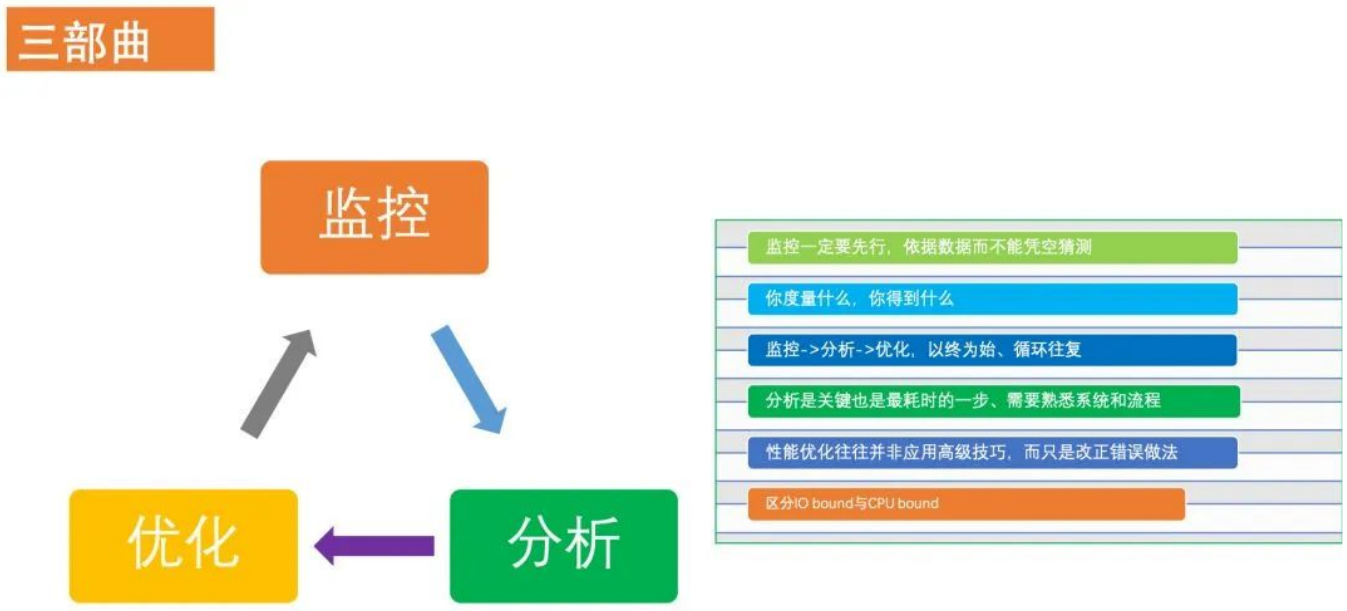


性能指标：吞吐率、响应时间、QPS/IOPS、TP99、资源使用率是我们经常关注的指标。

时间量级

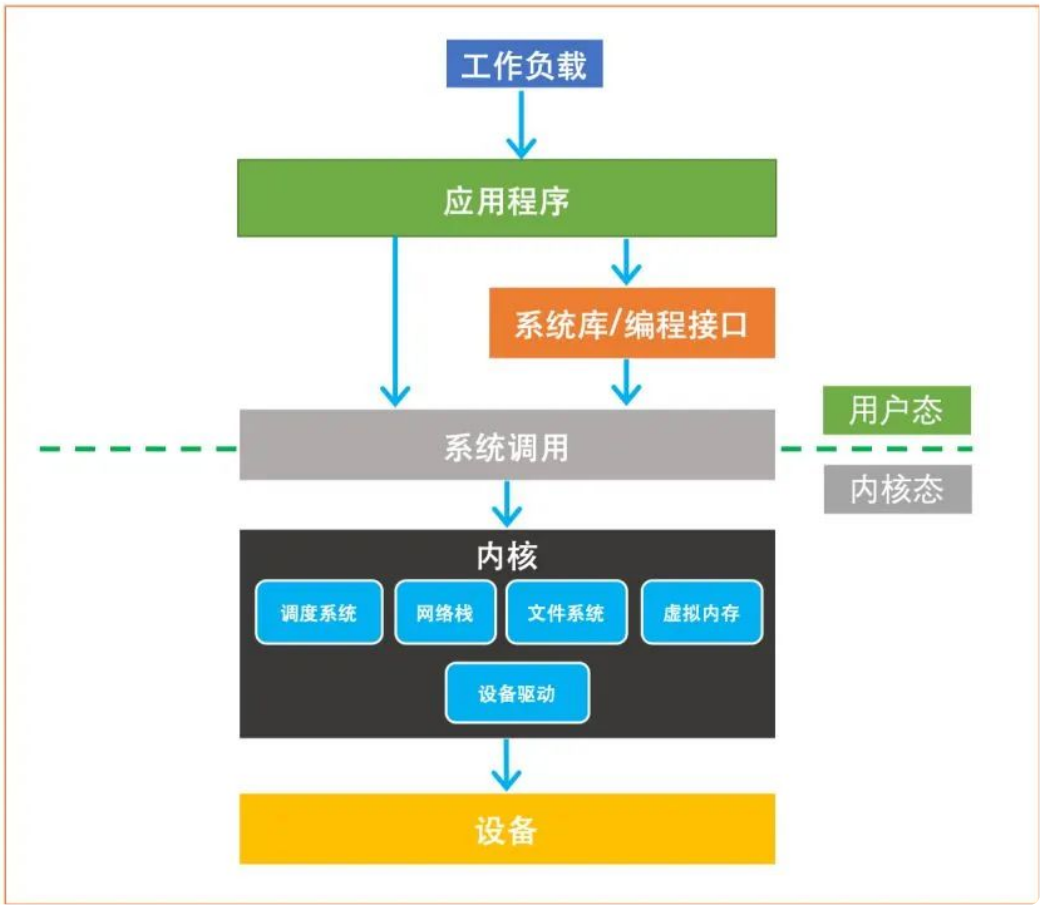


时间度量：从cpu cycle到网络IO，自上到下，时间量级越大。



监控、分析、优化，三部曲，以终为始，循环往复。

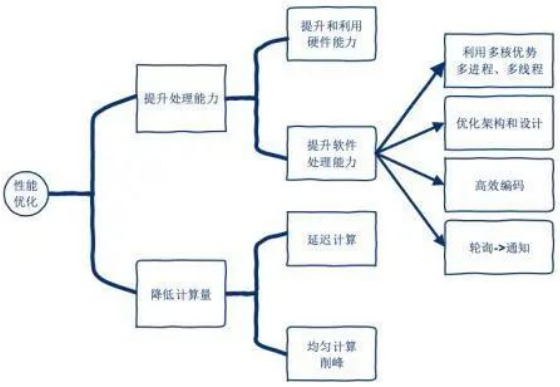
分析视角



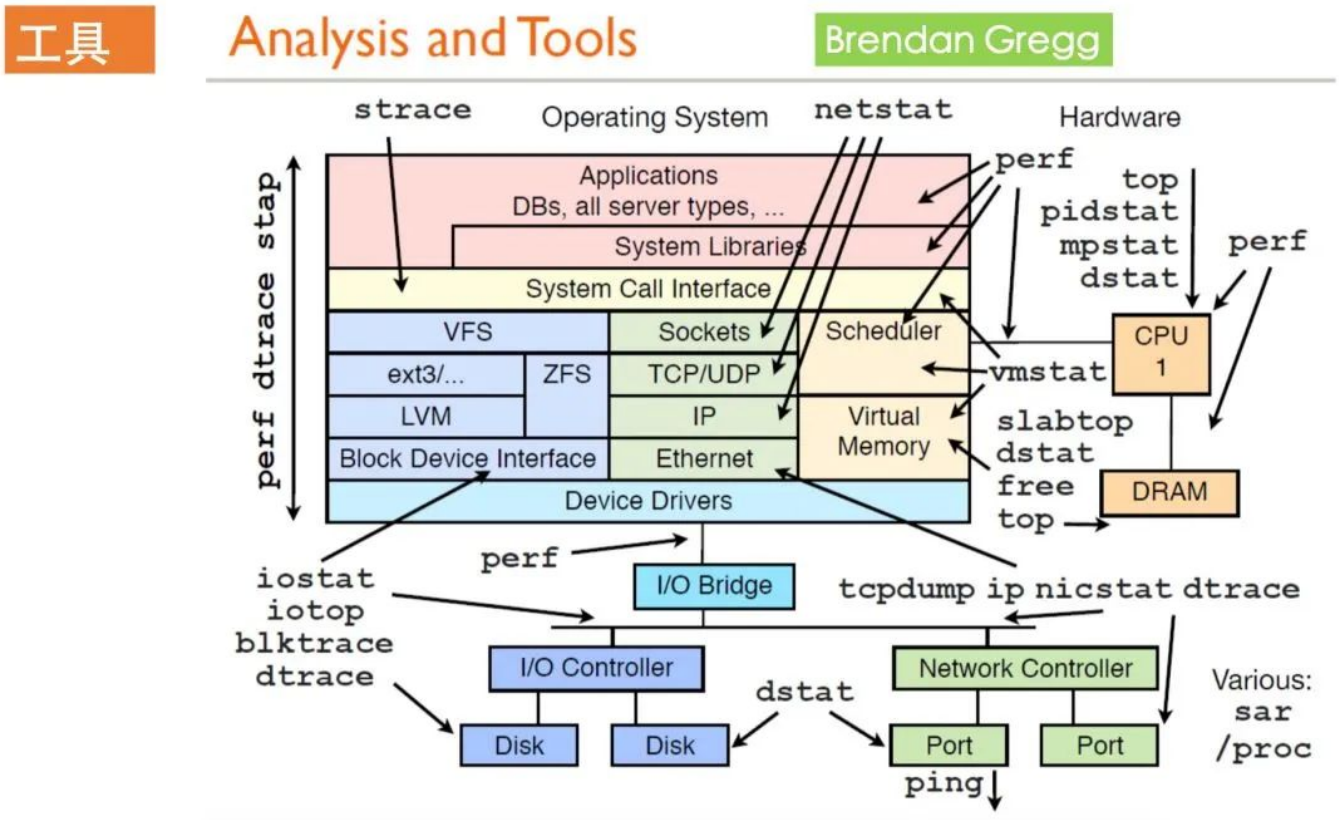
优化性能，需要一些系统编程知识。

提升CPU利用率和减少计算量是优化CPU的主要方向

高处着眼，优化架构和设计往往能获得很明显的提升
低处着手，细节很重要，细节是魔鬼

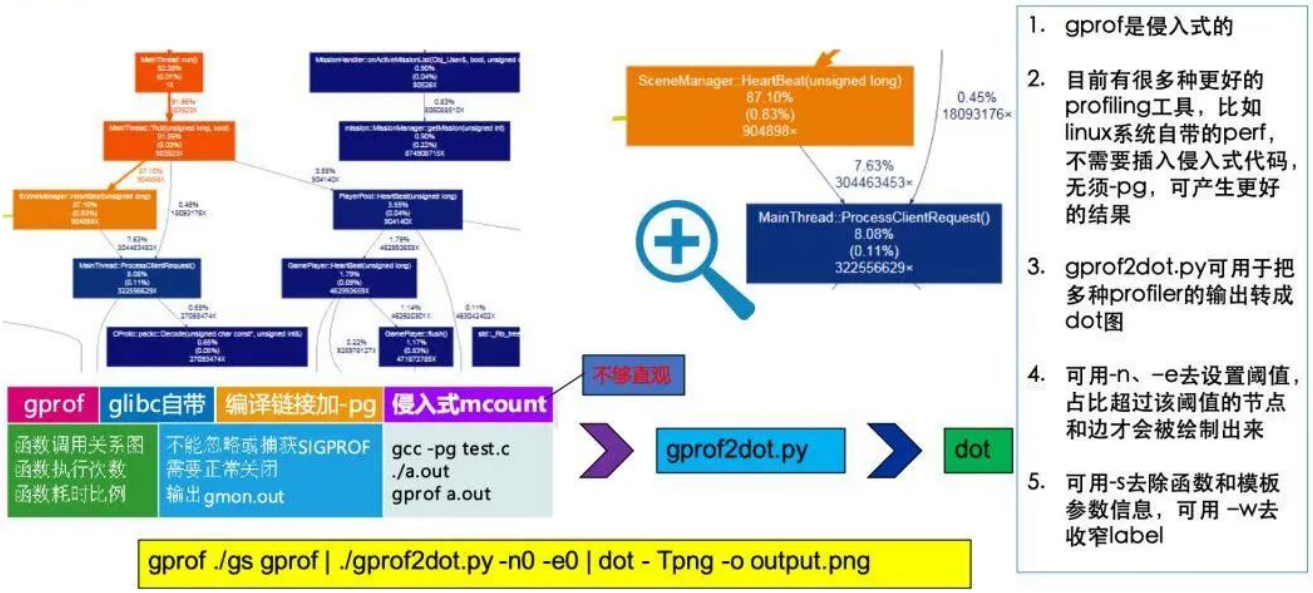


提升处理能力、减少计算量是优化的2个根本方向。



优化大师格雷格画的图，吊炸天，你应该很熟悉，gregg亲手实现了一些工具。

定位CPU瓶颈



1. gprof是侵入式的
2. 目前有很多种更好的profiling工具，比如linux系统自带的perf，不需要插入侵入式代码，无须-pg，可产生更好的结果
3. gprof2dot.py可用于把多种profiler的输出转成dot图
4. 可用-n、-e去设置阈值，占比超过该阈值的节点和边才会被绘制出来
5. 可用-s去除函数和模板参数信息，可用 -w去收窄label

借助工具定位性能瓶颈。gprof2dot.py可以处理多种采样输出数据

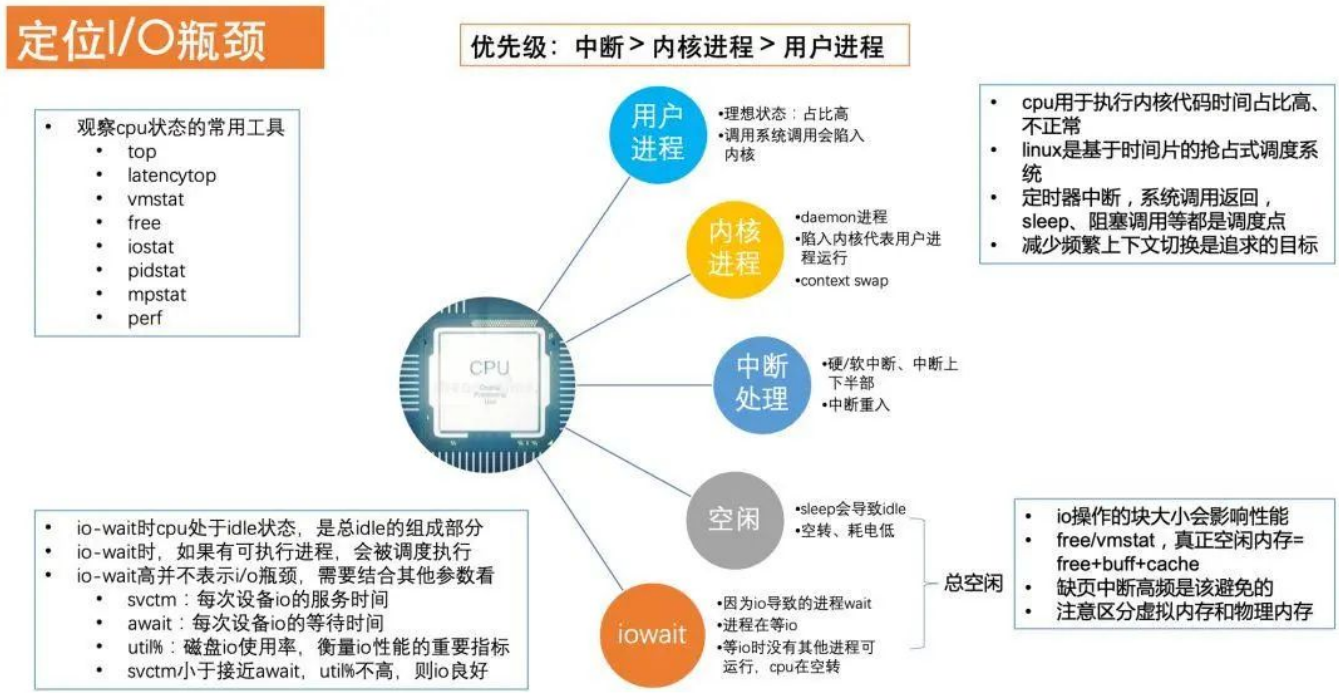
建议使用perf等非侵入式的profiling工具。

PERF

```
$perf list

List of pre-defined events (to be used in -e):

  alignment-faults                                   [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  rNNN                                               [Raw hardware event descriptor]
  cpu/t1=v1[,t2=v2,t3 ...]/modifier                  [Raw hardware event descriptor]
   (see 'man perf-list' on how to encode it)

  mem:<addr>[/len][:access]                          [Hardware breakpoint]
```

- perf可用于多种事件的性能探测
- 比较常用的cpu-cycle，也可以用于诊断缺页、分支预测失败、上下文切换，等各种预定义event。
- 既可诊断用户空间代码，也可用于内核空间代码
- linux系统自带，对程序性能影响小，相比较而言，gprof有数倍的性能衰退

perf不仅仅可以定位cpu瓶颈，还可以查看很多方面，比如缺页，分支预测失败，上下文切换等。



IO瓶颈，你应该知道的知识。

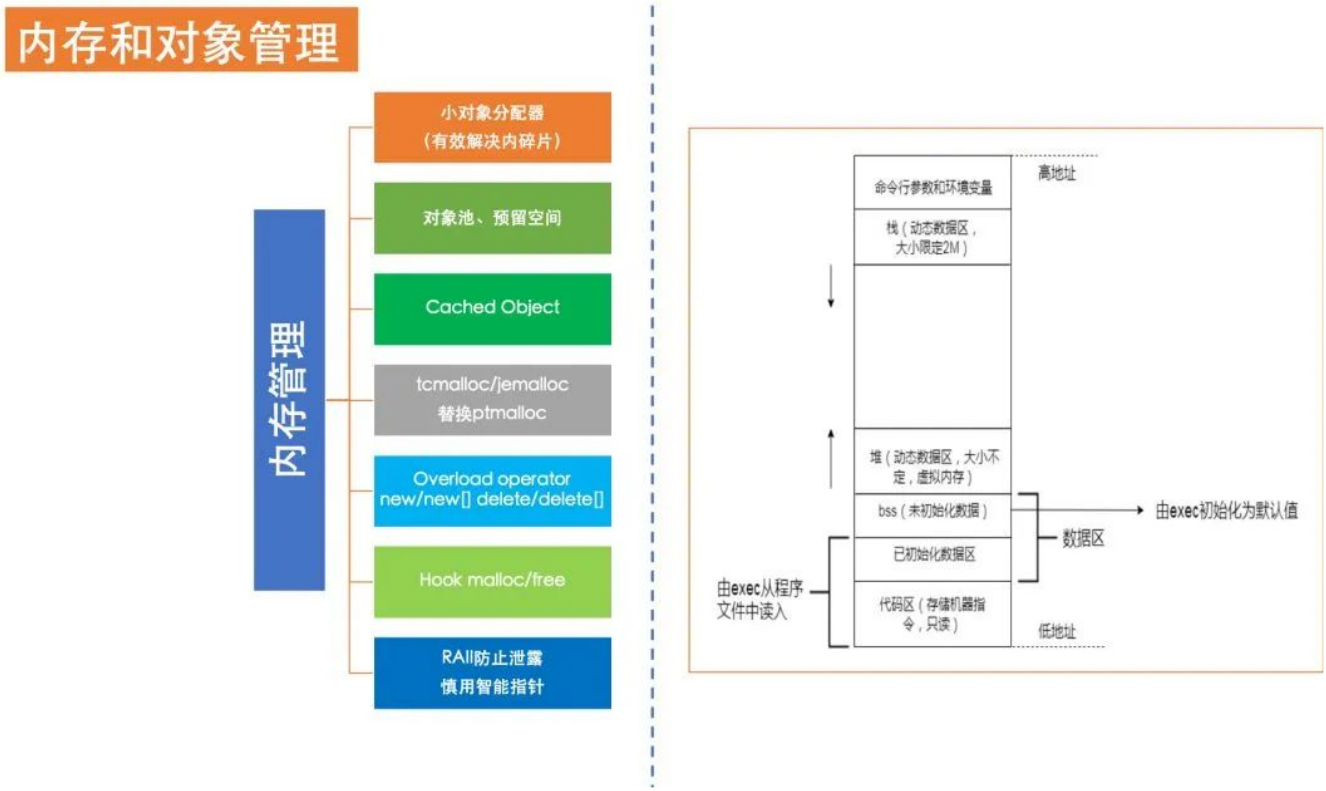
定位锁的瓶颈

- 目前最新的posix mutex实现已经改造成先自旋一小段，抢不到锁，再睡眠等待。
- 锁本身消耗并不大，认为锁是导致性能下降的元凶是不对的，加锁本身可能只有简单的几个汇编语句，比如cas，或者锁总线改一下内存变量
- 锁导致的性能下降主要是因为争用，要减少争用、最小化持有锁的时间。
- 睡眠锁的问题可以用latencytop确认

有关锁的知识，你应该知道的。

多线程的方法面面

多线程的学问很大



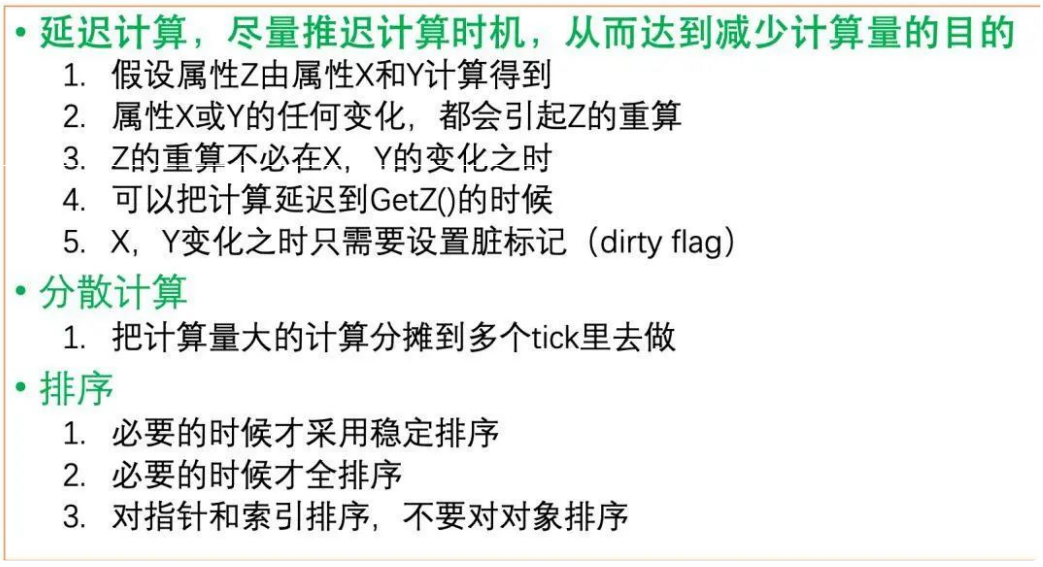
内存管理的方方面面

最佳实践，没有足够理由，你不应该违背。

最佳实践

高效内联	减少/避免拷贝	减少计算	优化循环	调试开关	跳转和异常	其他
• 高频短小函数要内联 • 定义的时候显式inline，而不是依赖编译器优化 • 理解函数调用的开销 • Inline会导致二进制膨胀	• 传参尽量传引用 • 避免返回大的对象 • 减少容器扩容导致的拷贝 • reserve预设容量、避免多次扩容	• 判断前置 • 推迟定义变量 • 控制参数数量 • 有必要才返回值	• 尽量把计算移出循环 • 循环内代码考虑局部性 • 减少不必要的多次循环引用	• 要有开关、确保release版本干净 • 写文件比printf快20倍 • 日志不要太多，通常系统日志消耗占比7%左右	• 使用分支预测、考虑调整分支顺序 • 理解并减少控制跳转 • 理解指令流水线 • 减少函数调用	• 熟悉编译优化选项 • 理解编译优化的局限和限制 • 利用cpu亲和性 • 理解递归的开销和低效

先学会前人宝贵的经验，才能创新。



- **延迟计算，尽量推迟计算时机，从而达到减少计算量的目的**
  1. 假设属性Z由属性X和Y计算得到
  2. 属性X或Y的任何变化，都会引起Z的重算
  3. Z的重算不必在X，Y的变化之时
  4. 可以把计算延迟到GetZ()的时候
  5. X，Y变化之时只需要设置脏标记（dirty flag）
- **分散计算**
  1. 把计算量大的计算分摊到多个tick里去做
- **排序**
  1. 必要的时候才采用稳定排序
  2. 必要的时候才全排序
  3. 对指针和索引排序，不要对对象排序

关于排序，你应该知道的。

参考资料

这些资料不错，你值得拥有。

一般性原则

依据数据而不是凭空猜测

这是性能优化的第一原则，当我们怀疑性能有问题的时候，应该通过测试、日志、profillig来分析出哪里有问题，有的放矢，而不是凭感觉、撞运气。一个系统有了性能问题，瓶颈有可能是CPU，有可能是内存，有可能是IO（磁盘IO，网络IO），大方向的定位可以使用top以及stat系列来定位（vmstat，iostat，netstat...），针对单个进程，可以使用pidstat来分析。

在本文中，主要讨论的是CPU相关的性能问题。按照80/20定律，绝大多数的时间都耗费在少量的代码片段里面，找出这些代码唯一可靠的办法就是profile，我所知的编程语言，都有相关的profile工具，熟练使用这些profile工具是性能优化的第一步。

忌过早优化

> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

我并不十分清楚Donald Knuth说出这句名言的上下文环境，但我自己是十分认同这个观念的。在我的工作环境（以及典型的互联网应用开发）与编程模式下，追求的是快速的迭代与试错，过早的优化往往是无用功。而且，过早的优化很容易拍脑袋，优化的点往往不是真正的性能瓶颈。

忌过度优化

> As performance is part of the specification of a program – a program that is unusably slow is not fit for purpose

性能优化的目标是追求合适的性价比。

在不同的阶段，我们对系统的性能会有一定的要求，比如吞吐量要达到多少多少。如果达不到这个指标，就需要去优化。如果能满足预期，那么就无需花费时间精力去优化，比如只有几十个人使用的内部系统，就不用按照十万在线的目标去优化。

而且，后面也会提到，一些优化方法是“有损”的，可能会对代码的可读性、可维护性有副作用。这个时候，就更不能过度优化。

深入理解业务

代码是服务于业务的，也许是服务于最终用户，也许是服务于其他程序员。不了解业务，很难理解系统的流程，很难找出系统设计的不足之处。后面还会提及对业务理解的重要性。

性能优化是持久战

当核心业务方向明确之后，就应该开始关注性能问题，当项目上线之后，更应该持续的进行性能检测与优化。

现在的互联网产品，不再是一锤子买卖，在上线之后还需要持续的开发，用户的涌入也会带来性能问题。因此需要自动化的检测性能问题，保持稳定的测试环境，持续的发现并解决性能问题，而不是被动地等到用户的投诉。

选择合适的衡量指标、测试用例、测试环境

正因为性能优化是一个长期的行为，所以需要固定衡量指标、测试用例、测试环境，这样才能客观反映性能的实际情况，也能展现出优化的效果。

衡量性能有很多指标，比如系统响应时间、系统吞吐量、系统并发量。不同的系统核心指标是不一样的，首先要明确本系统的核心性能诉求，固定测试用例；其次也要兼顾其他指标，不能顾此失彼。

测试环境也很重要，有一次突然发现我们的QPS高了许多，但是程序压根儿没优化，查了半天，才发现是换了一个更牛逼的物理机做测试服务器。