# 性能评估

## top指标理解

```
top - 10:57:14 up 13 days, 21:42,  5 users,  load average: 51.73, 48.57, 48.33
Tasks: 1076 total,   6 running, 1062 sleeping,   0 stopped,   8 zombie
%Cpu(s): 83.4 us,  9.8 sy,  0.0 ni,  3.9 id,  0.7 wa,  0.9 hi,  1.3 si,  0.0 st
MiB Mem : 257046.2 total,  39035.3 free, 140899.1 used,  77111.8 buff/cache
MiB Swap:  32768.0 total,  24387.7 free,   8380.2 used. 109956.5 avail Mem
```

- load average : 平均负载，指的是特定时间间隔内运行队列中的平均线程数
- %CPU：CPU使用率
  - user(通常1缩写为us)： 代表用户态CPU时间，不包括nice时间，但包括guest时间
  - system(通常缩写为sys)：代表内核态CPU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，nice的可取值范围是-20-19,数值越大，优先级反而越低
  - idle(通常缩写为id)：代表空闲时间，不包括等待I/0的时间
  - 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时，被其他虚拟机占用的CPU时间

## CPU性能指标

### 1 CPU使用率

- **用户CPU使用率**：包括用户态CPU使用率（user），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**：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
- **虚拟化环境会用到窃取CPU使用率和客户CPU使用率**：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时百分比

### 2 平均负载

平均负载等于逻辑CPU个数，表示每个CPU都恰好被充分利用，如果平均负载大于逻辑CPU个数，就表示负载比较重

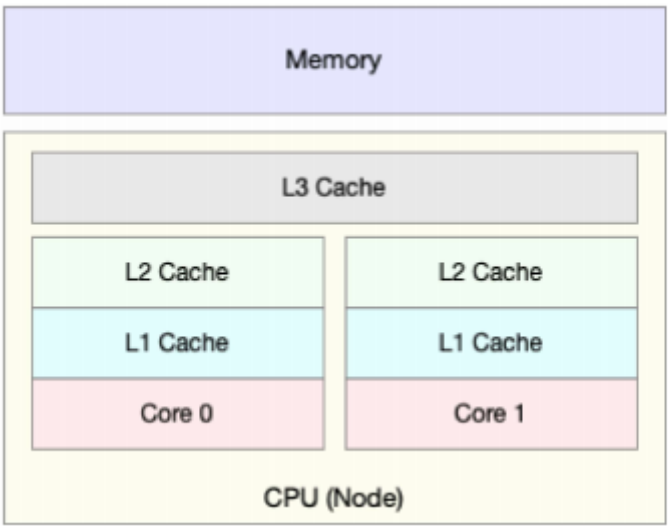
### 3 进程上下文切换

- 无法获取资源而导致的资源上下文切换
- 被系统强制调度导致的非资源上下文切换

二 CPU上下文切换

**4 CPU缓存命中率**

cpu的处理速度比内存的访问速度快得多，这样CPU访问内存的时候，免不了要等待内存的响应，为了协调这两者巨大的性能差距，CPU缓存就出现了




cpu缓存的速度介于CPU和内存之间，缓存的是热店的内存数据，根据不断增长的热店数据，这些缓存按照大小不同分为L1,L2,L3等三级缓存，其实L1和L2常用在单核中，L3则用在多核中

## CPU的多级缓存

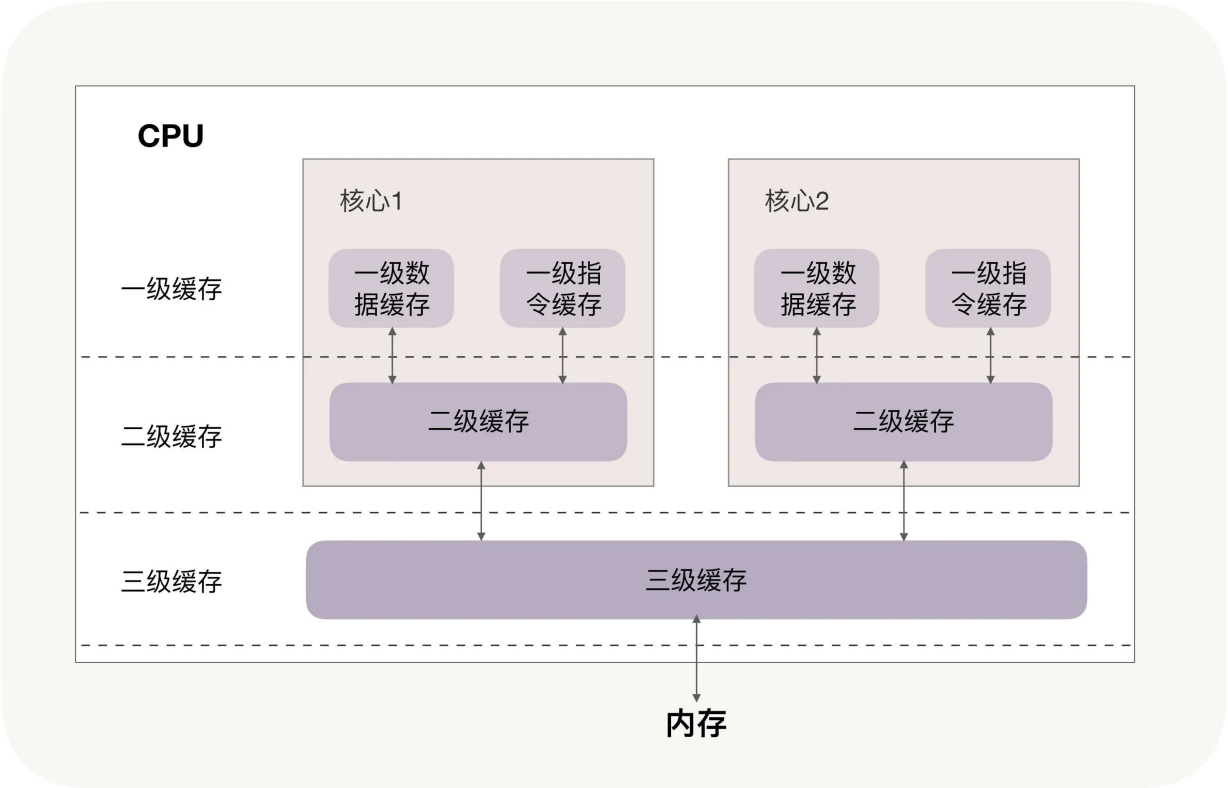
**背景**：代码的执行都依赖 CPU，特别是对于计算密集型的程序，CPU 的执行效率就开始变得至关重要。

由于 CPU 缓存由更快的 SRAM 构成（内存是由 DRAM 构成的），而且离 CPU 核心更近，如果运算时需要的输入数据是从 CPU 缓存，而不是内存中读取时，运算速度就会快很多。

所以，了解 CPU 缓存对性能的影响，便能够更有效地编写我们的代码，优化程序性能。

**CPU的3级缓存：**

程序执行时，会先将内存中的数据载入到共享的三级缓存中，再进入每颗核心独有的二级缓存，最后进入最快的一级缓存，之后才会被 CPU 使用




查看CPU各级缓存的大小

```
[root@bsa5296 nsfocus]# cat /sys/devices/system/cpu/cpu0/cache/index0/size
32K
[root@bsa5296 nsfocus]# cat /sys/devices/system/cpu/cpu0/cache/index1/size
32K
[root@bsa5296 nsfocus]# cat /sys/devices/system/cpu/cpu0/cache/index2/size
1024K
[root@bsa5296 nsfocus]# cat /sys/devices/system/cpu/cpu0/cache/index3/size
14080K
[root@bsa5296 nsfocus]#
```

缓存要比内存快很多。CPU 访问一次内存通常需要 100 个时钟周期以上，而访问一级缓存只需要 4~5 个时钟周期，二级缓存大约 12 个时钟周期，三级缓存大约 30 个时钟周期

## 多级缓存的利用

查看CPU缓存时，会发现又2个一级缓存，比如上图的index0和index1，是因为CPU会区别对待指令与数据，比如，“1+1=2”这个运算，“+”就是指令，会放在**一级指令缓存**中，而“1”这个输入数字，则放在**一级数据缓存**中。

### 提高数据缓存的命中率

经典c++代码案例：二维数组a[i][j]和a[j][i]的性能对比（java代码由于jvm做了内存优化，所以性能差异不大）

前者 array[j][i]执行的时间是后者 array[i][j]的 8 倍之多

**产生差异的原因**：因为二维数组 array 所占用的内存是连续的，比如若长度 N 的值为 2，那么内存中从前至后各元素的顺序是：

```
array[0][0]，array[0][1]，array[1][0]，array[1][1]。
```

array[i][j]访问数组元素，则完全与上述内存中元素顺序一致，因此访问 array[0][0]时，缓存已经把紧随其后的 3 个元素也载入了，CPU 通过快速的缓存来读取后续 3 个元素就可以。如果用 array[j][i]来访问，访问的顺序就是：

```
array[0][0]，array[1][0]，array[0][1]，array[1][1]
```

此时内存时跳跃访问的，如果 N 的数值很大，那么操作 array[j][i]时，是没有办法把 array[j+1][i]也读入缓存的。

### 为什么两者的执行时间有约7，8倍的差距？载入 array[0][0]元素时，缓存一次性会载入多少元素呢?

**coherency_line_size** 配置定义了CPU Cache Line，这里配置的64字节，当载入array[0][0]时，若他们的占用的内存不足64字节，CPU 就会顺序地补足后续元素。顺序访问的 array[i][j]因为利用了这一特点，所以就会比 array[j][i]要快。也正因为这样，此场景下，当元素类型是 4 个字节的整数时，性能就会比 8 个字节的高精度浮点数时速度更快，因为缓存一次载入的元素会更多。

```
[root@bsa3994 ~]# cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
```

对于二维数组，存放的时元素的地址，对于64位的操作系统，地址占用8个字节，（32位的操作系统是4个字节），因此，每批CPU cache line最多只能载入不到8个二维数组元素，所以性能差距大约接近8倍

### 应用：

nginx用于存放域名，http头部等数据的哈希表桶的大小server_names_hash_bucket_size 默认等于CPU cache line的值，当存放的字符串长度大于桶的大小是，需要修改server_names_hash_bucket_size，nginx明确规定应该是CPU cache line的整数倍

**CPU缓存伪共享**：是指多个 CPU 核心在并发访问共享的内存区域时，由于数据在不同 CPU 缓存之间来回传递，导致频繁的缓存一致性协议所带来的性能问题。虽然没有真正的数据冲突，但是由于缓存行的粒度，多个 CPU 同时访问不同的数据，但这些数据在同一个缓存行中，会导致频繁的缓存行失效和更新，进而影响性能。以下是关于伪共享问题的详细解释

- 缓存一致性协议：在多核 CPU 系统中，不同 CPU 核心有各自的缓存，它们分别从主内存中加载数据到缓存中进行处理。为了确保数据一致性，系统实现了缓存一致性协议，当一个 CPU 核心修改了缓存行中的数据，其他共享该缓存行的 CPU 核心需要进行缓存行失效，并重新从主内存中读取数据。这种缓存一致性的机制会导致性能损失。
- 伪共享效应：伪共享是由于多个 CPU 核心同时修改不同的数据，但这些数据被存储在同一个缓存行中，导致频繁的缓存行失效和更新。当一个 CPU 核心修改了其中一个数据，这个缓存行会失效，其他 CPU 核心访问同一缓存行的数据时需要重新从主存中加载，即使它们修改的是不同的数据。这种来回的缓存行变更和刷新会浪费大量的时间和资源，降低应用程序的性能。
- 解决方案：
  - 填充缓存行：在数据结构中增加填充字段，使得不同的数据被存储在不同的缓存行中，避免不必要的缓存行刷新。
  - 使用缓存隔离：通过NUMA（Non-Uniform Memory Access，非一致内存访问）架构或软件指定内存绑定，将相关的数据存储在同一 NUMA 节点上，减少跨节点访存带来的性能损失。
  - 使用特定优化技术：一些编程技术，如缓存友好的数据结构设计、编写 Cache-conscious 的代码等，也可以帮助减少伪共享问题的影响。

**上面案例：**按照 cpu cache line（比如 64 字节）来访问内存时，不会出现多核 CPU 下的伪共享问题，可以尽量减少访问内存的次数。比如，若桶大小为 64 字节，那么根据地址获取字符串时只需要访问一次内存，而桶大小为 50 字节，会导致最坏 2 次访问内存，而 70 字节最坏会有 3 次访问内存。

**查看缓存命中率：**

- 查看三级缓存总的命中率

```
perf stat -p PID -e cache-misses -e references
# 缓存未命中 cache-misses 事件
# 读取缓存次数 cache-references 事件
```

- 查看一级缓存命中情况

```
perf stat -p PID -e L1-dcache-load-misses -e L1-dcache-loads
```

- 查看二级缓存命中情况

```
perf stat -p PID -e LLC-loads,LLC-load-misses
```

**查看进程执行的总指令数：**

```
perf stat -p pid -e instructions -e cycles
#instructions 事件指明了进程执行的总指令数，而 cycles 事件指明了运行的时钟周期，instructions / cycles 就可以得到每时钟周期所执行的指令数,缩写为IPC
#如果缓存未命中，则CPU要等待内存的慢速读取，因此IPC就会很低
```

# 提高指令缓存的命中率

## 分支预测：

CPU 的分支预测是指处理器在执行程序时对分支指令（如条件分支、循环等）进行预测来提高执行效率的一种技术。由于分支指令处于程序执行流的关键位置，正确预测并处理分支指令可以减少循环和条件语句等控制结构带来的延迟，从而提高指令级并行度和整体性能。以下是 CPU 分支预测的详细解释：

- 静态分支预测

静态分支预测是一种简单的预测方式，它不基于运行时信息而只根据固定的规则预测分支的执行方向。例如，一些简单的策略可能认为循环会一直执行（向前跳转），因此预测循环将继续执行。静态分支预测不能适应程序运行时的变化，但可以提供一定的预测准确性。

- 动态分支预测

动态分支预测利用处理器内部的预测器来根据运行时的分支历史信息和分支指令的条件做出预测。处理器会记录分支指令的执行情况，根据历史信息动态地调整预测结果。主要包括两种类型的动态分支预测器：

  - 单向分支预测器：预测分支是向前还是向后跳转。
  - 双向分支预测器：根据两种不同方向来预测分支的执行路径。

- 多级分支预测
- 动态分支预测器的实现

**经典案例：**

遍历一个元素为0-255之间随机数字组成的数组，判断如果小于128，就把元素的值置为0，观察先遍历后排序和先排序后遍历的性能

```java
import java.util.Arrays;
import java.util.Random;

public class PerformanceComparison {
    public static void main(String[] args) {
        int[] randomArray = generateRandomArray(10000000);

        // 直接遍历
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < randomArray.length; i++) {
            if (randomArray[i] < 128) {
                randomArray[i] = 0;
            }
        }
        long endTime = System.currentTimeMillis();
        System.out.println("遍历无序列表的耗时: " + (endTime - startTime) + " 毫秒");

        randomArray = generateRandomArray(10000000);
        Arrays.sort(randomArray);
        //遍历有序列表
        startTime = System.currentTimeMillis();
        for (int i = 0; i < randomArray.length; i++) {
            if (randomArray[i] < 128) {
                randomArray[i] = 0;
            }
        }
        endTime = System.currentTimeMillis();
        System.out.println("遍历有序列表耗时: " + (endTime - startTime) + " 毫秒");
    }

    public static int[] generateRandomArray(int size) {
        int[] randomArray = new int[size];
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            randomArray[i] = random.nextInt(256); // 生成0-255之间的随机数
        }
        return randomArray;
    }
```

```
39 }
```

```
遍历无序列表的耗时： 69 毫秒
遍历有序列表耗时： 23 毫秒
```

当代码中出现 if、switch 等语句时，意味着此时至少可以选择跳转到两段不同的指令去执行。如果分支预测器可以预测接下来要在哪段代码执行（比如 if 还是 else 中的指令），就可以提前把这些指令放在缓存中，CPU 执行时就会很快。当数组中的元素完全随机时，分支预测器无法有效工作，而当 array 数组有序时，分支预测器会动态地根据历史命中数据对未来进行预测，命中率就会非常高。

## 提升多核CPU下缓存的命中率

若进程 A 在时间片 1 里使用 CPU 核心 1，自然也填满了核心 1 的一、二级缓存，当时间片 1 结束后，操作系统会让进程 A 让出 CPU，基于效率并兼顾公平的策略重新调度 CPU 核心 1，以防止某些进程饿死。如果此时 CPU 核心 1 繁忙，而 CPU 核心 2 空闲，则进程 A 很可能会被调度到 CPU 核心 2 上运行，这样，即使我们对代码优化得再好，也只能在一个时间片内高效地使用 CPU 一、二级缓存了，下一个时间片便面临着缓存效率的问题。

因此，操作系统提供了将进程或者线程绑定到某一颗 CPU 上运行的能力。如 Linux 上提供了 sched_setaffinity 方法实现这一功能

当多线程同时执行密集计算，且 CPU 缓存命中率很高时，如果将每个线程分别绑定在不同的 CPU 核心上，性能便会获得非常可观的提升。Perf 工具也提供了 cpu-migrations 事件，它可以显示进程从不同的 CPU 核心上迁移的次数。