

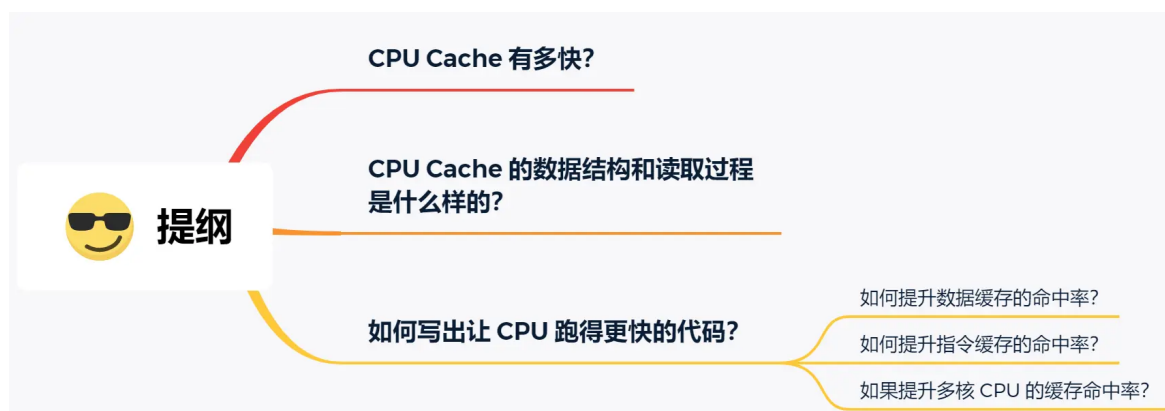
2.3 如何写出让 CPU 跑得更快的代码？

代码都是由 CPU 跑起来的，我们代码写的好与坏就决定了 CPU 的执行效率，特别是在编写计算密集型的程序，更要注重 CPU 的执行效率，否则将会大大影响系统性能。

CPU 内部嵌入了 CPU Cache（高速缓存），它的存储容量很小，但是离 CPU 核心很近，所以缓存的读写速度是极快的，那么如果 CPU 运算时，直接从 CPU Cache 读取数据，而不是从内存的话，运算速度就会很快。

但是，大多数人不知道 CPU Cache 的运行机制，以至于不知道如何才能够写出能够配合 CPU Cache 工作机制的代码，一旦你掌握了它，你写代码的时候，就有新的优化思路了。

那么，接下来我们就来看看，CPU Cache 到底是什么样的，是如何工作的呢，又该如何写出让 CPU 执行更快的代码呢？



CPU Cache 有多快？

你可能会好奇为什么有了内存，还需要 CPU Cache？根据摩尔定律，CPU 的访问速度每 18 个月就会翻倍，相当于每年增长 60% 左右，内存的速度当然也会不断增长，但是增长的速度远小于 CPU，平均每年只增长 7% 左右。于是，CPU 与内存的访问性能的差距不断拉大。

到现在，一次内存访问所需时间是 200~300 多个时钟周期，这意味着 CPU 和内存的访问速度已经相差 200~300 多倍了。

为了弥补 CPU 与内存两者之间的性能差异，就在 CPU 内部引入了 CPU Cache，也称高速缓存。



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

产 1 MB 大小的 CPU Cache 需要 7 美金的成本，而内存只需要 0.015 美金的成本，成本方面相差了 466 倍，所以 CPU Cache 不像内存那样动辄以 GB 计算，它的大小是以 KB 或 MB 来计算的。

在 Linux 系统中，我们可以使用下图的方式来查看各级 CPU Cache 的大小，比如我这手上这台服务器，离 CPU 核心最近的 L1 Cache 是 32KB，其次是 L2 Cache 是 256KB，最大的 L3 Cache 则是 3MB。

```
# 查看 L1 Cache 数据缓存的大小
$ cat /sys/devices/system/cpu/cpu0/cache/index0/size
32K

# 查看 L1 Cache 指令缓存的大小
$ cat /sys/devices/system/cpu/cpu0/cache/index1/size
32K

# 查看 L2 Cache 的大小
$ cat /sys/devices/system/cpu/cpu0/cache/index2/size
256K

# 查看 L3 Cache 的大小
$ cat /sys/devices/system/cpu/cpu0/cache/index3/size
3072K
```

其中，L1 Cache 通常会分为「数据缓存」和「指令缓存」，这意味着数据和指令在 L1 Cache 这一层是分开缓存的，上图中的 index0 也就是数据缓存，而 index1 则是指令缓存，它们的大小通常是一样的。

另外，你也会注意到，L3 Cache 比 L1 Cache 和 L2 Cache 大很多，这是因为 L1 Cache 和 L2 Cache 都是每个 CPU 核心独有的，而 L3 Cache 是多个 CPU 核心共享的。

程序执行时，会先将内存中的数据加载到共享的 L3 Cache 中，再加载到每个核心独有的 L2 Cache，最后进入到最快的 L1 Cache，之后才会被 CPU 读取。它们之间的层级关系，如下图所示：



目录



侧边栏



夜间



技术群



资料



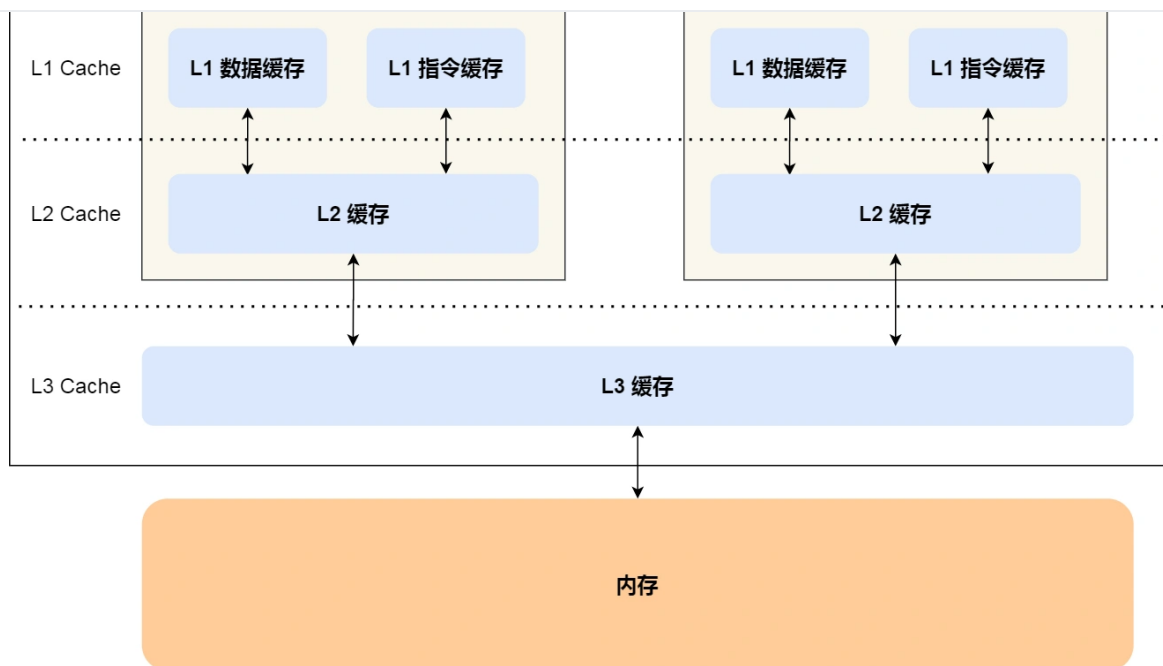
支持我



上一篇



下一篇



越靠近 CPU 核心的缓存其访问速度越快，CPU 访问 L1 Cache 只需要 2~4 个时钟周期，访问 L2 Cache 大约 10~20 个时钟周期，访问 L3 Cache 大约 20~60 个时钟周期，而访问内存速度大概在 200~300 个时钟周期之间。如下表格：

部件	CPU 访问所需时间
L1 Cache	2~4 个时钟周期
L2 Cache	10~20 个时钟周期
L3 Cache	20~60 个时钟周期
内存	200~300 个时钟周期

时钟周期是 CPU 主频的倒数，
比如 2GHZ 主频的 CPU，一个时钟周期是 0.5ns

所以，CPU 从 L1 Cache 读取数据的速度，相比从内存读取的速度，会快 100 多倍。



目录



侧边栏



夜间



技术群



资料



支持我



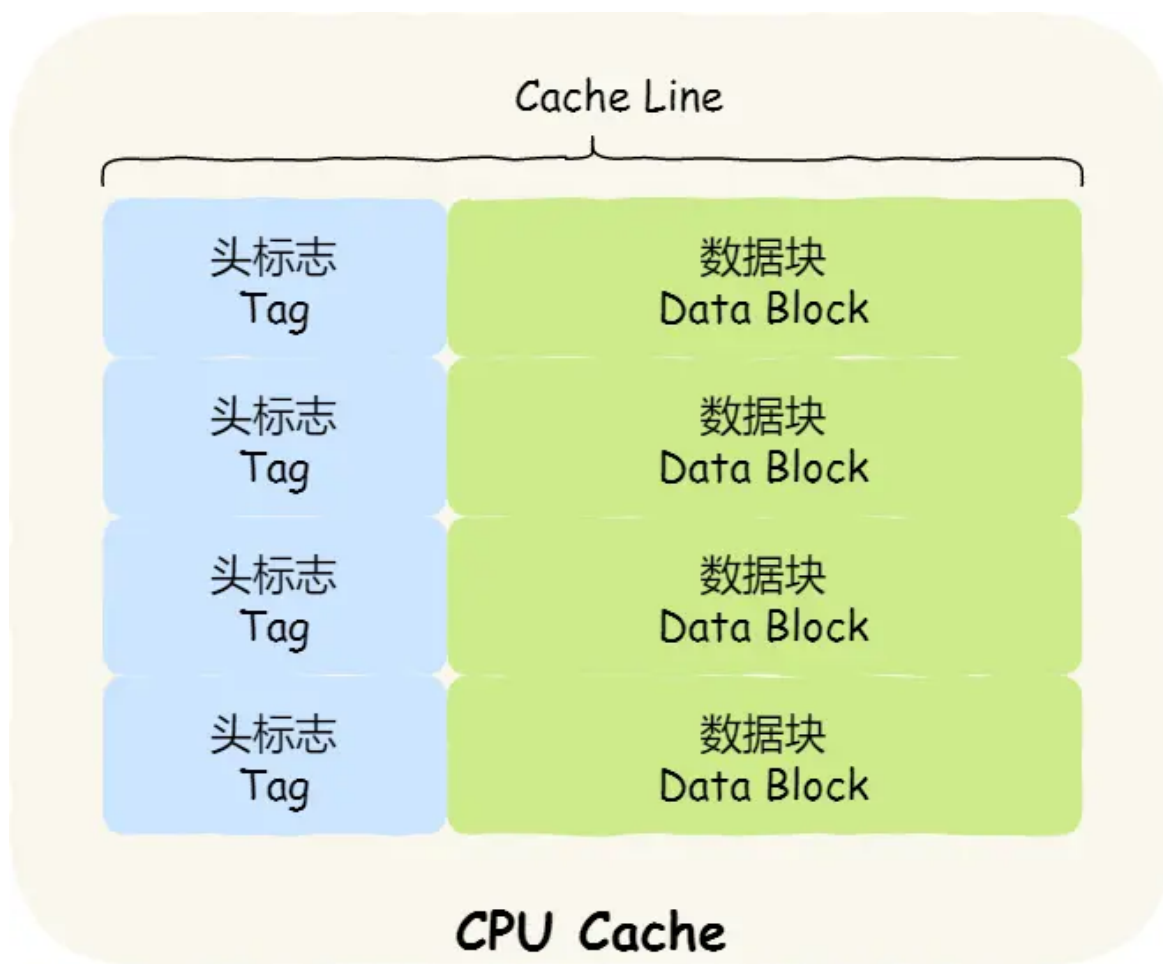
上一篇



下一篇

CPU Cache 的数据结构和读取过程是什么样的？

我们先简单了解下 CPU Cache 的结构，CPU Cache 是由很多个 Cache Line 组成的，Cache Line 是 CPU 从内存读取数据的基本单位，而 Cache Line 是由各种标志（Tag）+ 数据块（Data Block）组成，你可以在下图清晰的看到：



CPU Cache 的数据是从内存中读取过来的，它是以一小块一小块读取数据的，而不是按照单个数组元素来读取数据的，在 CPU Cache 中的，这样一小块一小块的数据，称为 **Cache Line（缓存块）**。

你可以在你的 Linux 系统，用下面这种方式来查看 CPU 的 Cache Line，你可以看我服务器的 L1 Cache Line 大小是 64 字节，也就意味着 **L1 Cache 一次载入数据的大小是 64 字节**。



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



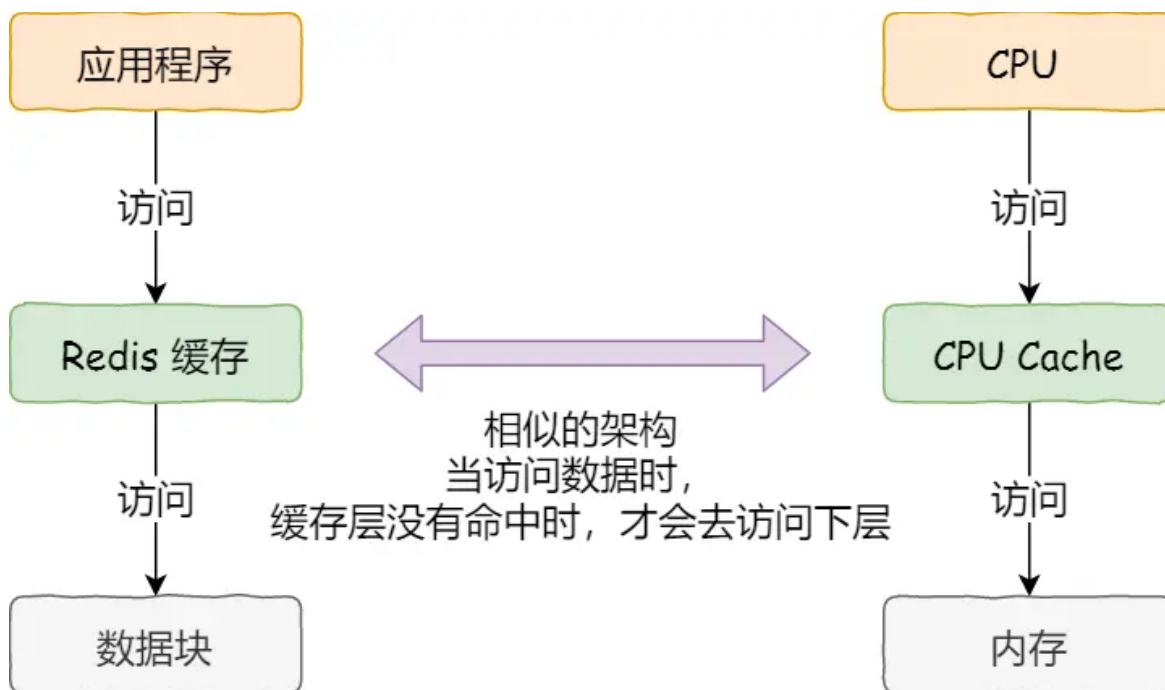
下一篇

```
# 查看 L1 Cache 数据缓存一次载入数据的大小
```

```
$ cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
```

比如，有一个 `int array[100]` 的数组，当载入 `array[0]` 时，由于这个数组元素的大小在内存只占 4 字节，不足 64 字节，CPU 就会**顺序加载**数组元素到 `array[15]`，意味着 `array[0]~array[15]` 数组元素都会被缓存在 CPU Cache 中了，因此当下次访问这些数组元素时，会直接从 CPU Cache 读取，而不用再从内存中读取，大大提高了 CPU 读取数据的性能。

事实上，CPU 读取数据的时候，无论数据是否存放到 Cache 中，CPU 都是先访问 Cache，只有当 Cache 中找不到数据时，才会去访问内存，并把内存中的数据读入到 Cache 中，CPU 再从 CPU Cache 读取数据。



这样的访问机制，跟我们使用「内存作为硬盘的缓存」的逻辑是一样的，如果内存有缓存的数据，则直接返回，否则要访问龟速一般的硬盘。

那 CPU 怎么知道要访问的内存数据，是否在 Cache 里？如果在的话，如何找到 Cache 对应的数据呢？我们从最简单、基础的**直接映射 Cache (Direct Mapped Cache)** 说起，来看看整个 CPU Cache 的数据结构和访问逻辑。

前面，我们提到 CPU 访问内存数据时，是一小块一小块数据读取的，具体这一小块数据的大小，取决于 `coherency_line_size` 的值，一般 64 字节。在内存中，这一块的数据我们称为**内存块 (Block)**，读取的时候我们要拿到数据所在内存块的地址。



目录



侧边栏



夜间



技术群



资料



支持我



上一篇

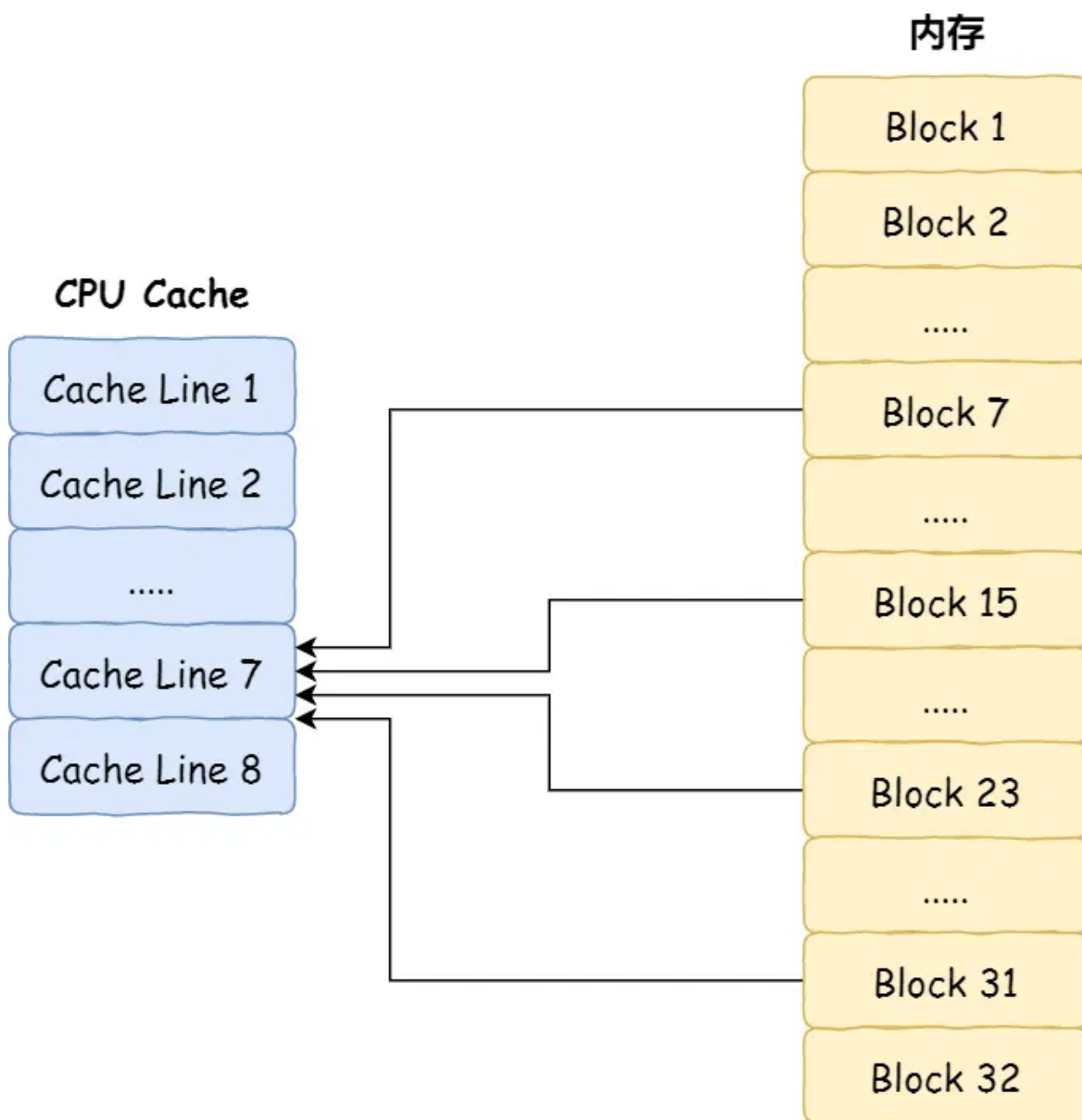


下一篇

就是内存块地址对应的 CPU Cache Line（缓存块）的地址。

举个例子，内存共被划分为 32 个内存块，CPU Cache 共有 8 个 CPU Cache Line，假设 CPU 想要访问第 15 号内存块，如果 15 号内存块中的数据已经缓存在 CPU Cache Line 中的话，则是一定映射在 7 号 CPU Cache Line 中，因为 $15 \% 8$ 的值是 7。

机智的你肯定发现了，使用取模方式映射的话，就会出现多个内存块对应同一个 CPU Cache Line，比如上面的例子，除了 15 号内存块是映射在 7 号 CPU Cache Line 中，还有 7 号、23 号、31 号内存块都是映射到 7 号 CPU Cache Line 中。



因此，为了区别不同的内存块，在对应的 CPU Cache Line 中我们还会存储一个**组标记** (Tag)。这个组标记会记录当前 CPU Cache Line 中存储的数据对应的内存块，我们可以用



目录



侧边栏



夜间



技术群



资料



支持我



上一篇

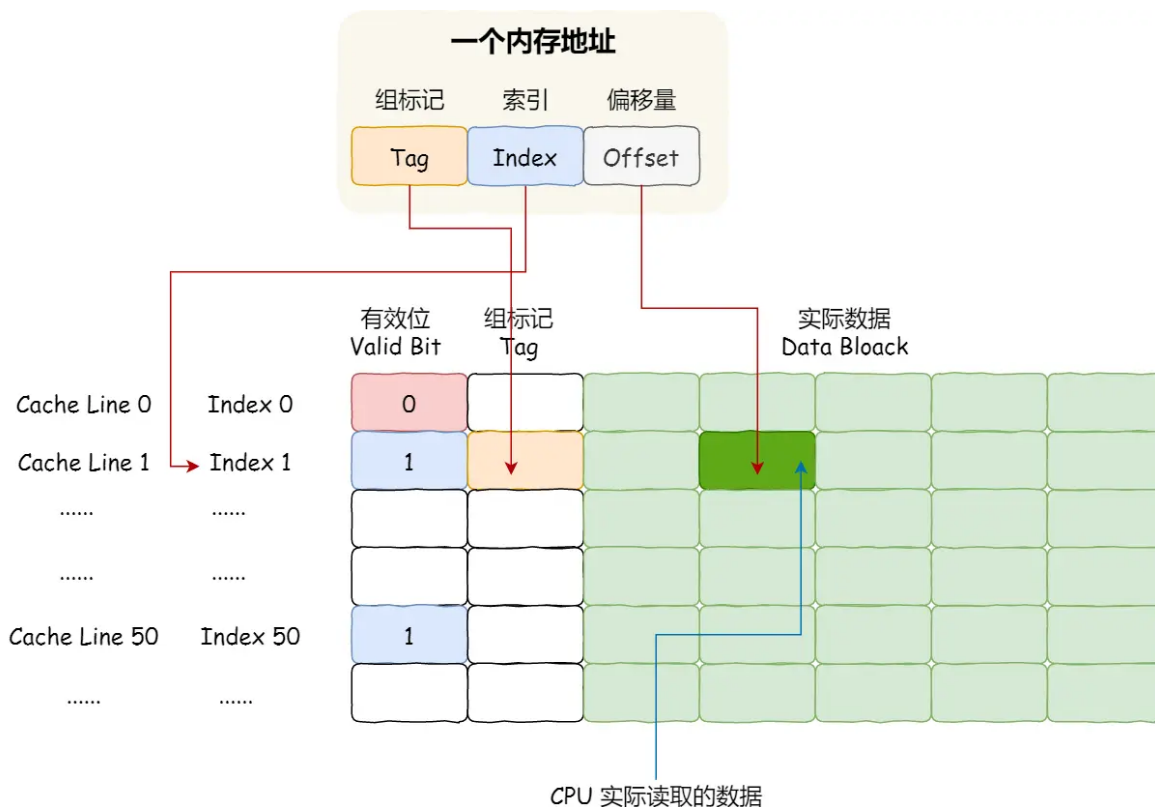


下一篇

- 一个是，从内存加载过来的实际存放**数据 (Data)**。
- 另一个是，**有效位 (Valid bit)**，它是用来标记对应的 CPU Cache Line 中的数据是否是有效的，如果有效位是 0，无论 CPU Cache Line 中是否有数据，CPU 都会直接访问内存，重新加载数据。

CPU 在从 CPU Cache 读取数据的时候，并不是读取 CPU Cache Line 中的整个数据块，而是读取 CPU 所需要的一个数据片段，这样的数据统称为一个**字 (Word)**。那怎么在对应的 CPU Cache Line 中数据块中找到所需的字呢？答案是，需要一个**偏移量 (Offset)**。

因此，一个内存的访问地址，包括**组标记、CPU Cache Line 索引、偏移量**这三种信息，于是 CPU 就能通过这些信息，在 CPU Cache 中找到缓存的数据。而对于 CPU Cache 里的数据结构，则是由**索引 + 有效位 + 组标记 + 数据块**组成。



如果内存中的数据已经在 CPU Cache 中了，那 CPU 访问一个内存地址的时候，会经历这 4 个步骤：

1. 根据内存地址中索引信息，计算在 CPU Cache 中的索引，也就是找出对应的 CPU Cache Line 的地址；
2. 找到对应 CPU Cache Line 后，判断 CPU Cache Line 中的有效位，确认 CPU Cache Line 中数据是否是有效的，如果是无效的，CPU 就会直接访问内存，并重新加载数据，如果数据



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

我们要访问的内存数据，如果不是的话，CPU 就会直接访问内存，开重新加载数据，如果是的话，则往下执行；

4. 根据内存地址中偏移量信息，从 CPU Cache Line 的数据块中，读取对应的字。

到这里，相信你对直接映射 Cache 有了一定认识，但其实除了直接映射 Cache 之外，还有其他通过内存地址找到 CPU Cache 中的数据策略，比如全相连 Cache (*Fully Associative Cache*)、组相连 Cache (*Set Associative Cache*) 等，这几种策略的数据结构都比较相似，我们理解了直接映射 Cache 的工作方式，其他的策略如果你有兴趣去看，相信很快就能理解的了。

如何写出让 CPU 跑得更快的代码？

我们知道 CPU 访问内存的速度，比访问 CPU Cache 的速度慢了 100 多倍，所以如果 CPU 所要操作的数据在 CPU Cache 中的话，这样将会带来很大的性能提升。访问的数据在 CPU Cache 中的话，意味着**缓存命中**，缓存命中率越高的话，代码的性能就会越好，CPU 也就跑的越快。

于是，「如何写出让 CPU 跑得更快的代码？」这个问题，可以改成「如何写出 CPU 缓存命中率高的代码？」。

在前面我也提到，L1 Cache 通常分为「数据缓存」和「指令缓存」，这是因为 CPU 会分别处理数据和指令，比如 $1+1=2$ 这个运算， $+$ 就是指令，会被放在「指令缓存」中，而输入数字 1 则会被放在「数据缓存」里。

因此，**我们要分开来看「数据缓存」和「指令缓存」的缓存命中率。**

如何提升数据缓存的命中率？

假设要遍历二维数组，有以下两种形式，虽然代码执行结果是一样，但你觉得哪种形式效率最高呢？为什么高呢？



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇


```
// 二维数组
```

```
array[N][N] = 0;
```

```
// 形式一:
```

```
for(i = 0; i < N; i+=1) {  
    for(j = 0; j < N; j+=1) {  
        array[i][j] = 0;  
    }  
}
```

```
// 形式二:
```

```
for(i = 0; i < N; i+=1) {  
    for(j = 0; j < N; j+=1) {  
        array[j][i] = 0;  
    }  
}
```

经过测试，形式一 `array[i][j]` 执行时间比形式二 `array[j][i]` 快好几倍。

之所以有这么大的差距，是因为二维数组 `array` 所占用的内存是连续的，比如长度 `N` 的值是 2 的话，那么内存中的数组元素的布局顺序是这样的：



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

```
// 内存中的数组元素的布局顺序
array[0][0], array[0][1], array[1][0], array[1][1]
```

形式一用 `array[i][j]` 访问数组元素的顺序，正是和内存中数组元素存放的顺序一致。当 CPU 访问 `array[0][0]` 时，由于该数据不在 Cache 中，于是会「顺序」把跟随其后的 3 个元素从内存中加载到 CPU Cache，这样当 CPU 访问后面的 3 个数组元素时，就能在 CPU Cache 中成功地找到数据，这意味着缓存命中率很高，缓存命中的数据不需要访问内存，这便大大提高了代码的性能。

而如果用形式二的 `array[j][i]` 来访问，则访问的顺序就是：

```
// 形式二使用 array[j][i] 的访问顺序：
array[0][0], array[1][0], array[0][1], array[1][1]
```

你可以看到，访问的方式跳跃式的，而不是顺序的，那么如果 N 的数值很大，那么操作 `array[j][i]` 时，是没办法把 `array[j+1][i]` 也读入到 CPU Cache 中的，既然 `array[j+1][i]` 没有读取到 CPU Cache，那么就需要从内存读取该数据元素了。很明显，这种不连续性、跳跃式访问数据元素的方式，可能不能充分利用到了 CPU Cache 的特性，从而代码的性能不高。

那访问 `array[0][0]` 元素时，CPU 具体会一次从内存中加载多少元素到 CPU Cache 呢？这个问题，在前面我们也提到过，这跟 CPU Cache Line 有关，它表示 **CPU Cache 一次性能加载数据的大小**，可以在 Linux 里通过 `coherency_line_size` 配置查看它的大小，通常是 64 个字节。

```
# 查看 L1 Cache 数据缓存一次载入数据的大小
$ cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
```

也就是说，当 CPU 访问内存数据时，如果数据不在 CPU Cache 中，则会一次性会连续加载 64 字节大小的数据到 CPU Cache，那么当访问 `array[0][0]` 时，由于该元素不足 64 字



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



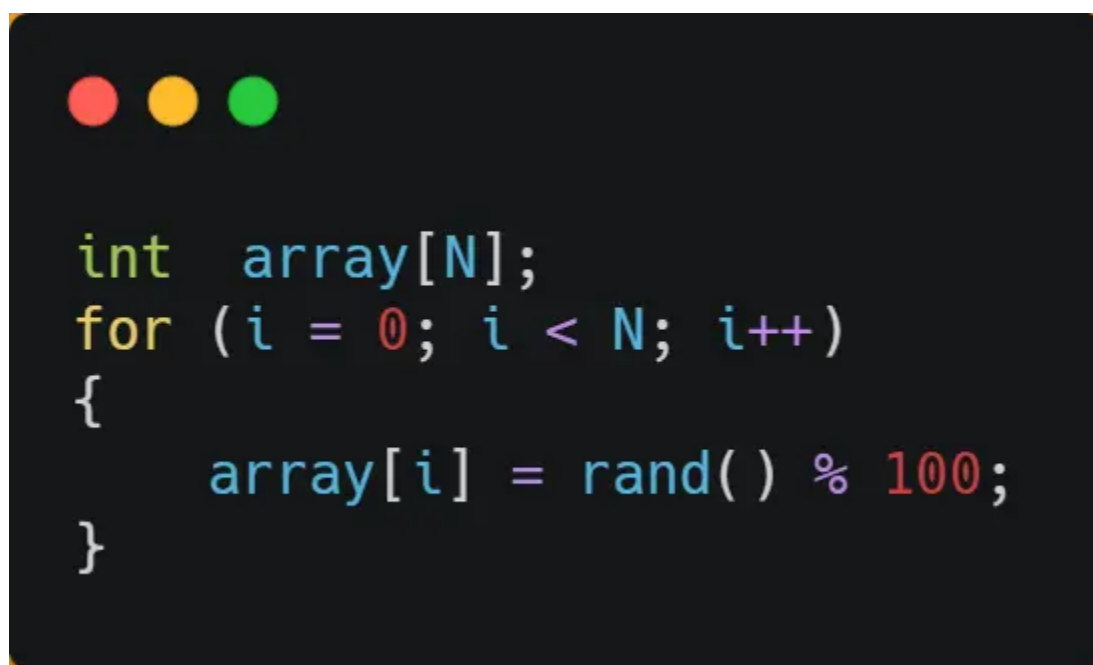
下一篇

因此，遇到这种遍历数组的情况时，按照内存布局顺序访问，将可以有效的利用 CPU Cache 带来的好处，这样我们代码的性能就会得到很大的提升，

如何提升指令缓存的命中率？

提升数据的缓存命中率的方式，是按照内存布局顺序访问，那针对指令的缓存该如何提升呢？

我们以一个例子来看看，有一个元素为 0 到 100 之间随机数字组成的一维数组：



```
int array[N];
for (i = 0; i < N; i++)
{
    array[i] = rand() % 100;
}
```

接下来，对这个数组做两个操作：



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

```
// 操作一：数组遍历
```

```
for(i = 0; i < N; i++) {  
    if (array[i] < 50) {  
        array[i] = 0;  
    }  
}
```

```
// 操作二：排序
```

```
sort(array, array + N);
```

- 第一个操作，循环遍历数组，把小于 50 的数组元素置为 0；
- 第二个操作，将数组排序；

那么问题来了，你觉得先遍历再排序速度快，还是先排序再遍历速度快呢？

在回答这个问题之前，我们先了解 CPU 的**分支预测器**。对于 if 条件语句，意味着此时至少可以选择跳转到两段不同的指令执行，也就是 if 还是 else 中的指令。那么，**如果分支预测可以预测到接下来要执行 if 里的指令，还是 else 指令的话，就可以「提前」把这些指令放在指令缓存中，这样 CPU 可以直接从 Cache 读取到指令，于是执行速度就会很快。**

当数组中的元素是随机的，分支预测就无法有效工作，而当数组元素都是顺序的，分支预测器会动态地根据历史命中数据对未来进行预测，这样命中率就会很高。

因此，先排序再遍历速度会更快，这是因为排序之后，数字是从小到大的，那么前几次循环命中 `if < 50` 的次数会比较多，于是分支预测就会缓存 if 里的 `array[i] = 0` 指令到 Cache 中，后续 CPU 执行该指令就只需要从 Cache 读取就好了。

如果你肯定代码中的 if 中的表达式判断为 true 的概率比较高，我们可以使用显示分支预测工具，比如在 C/C++ 语言中编译器提供了 likely 和 unlikely 这两种宏，如果



目录



侧边栏



夜间



技术群



资料



支持我



上一篇



下一篇

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

if (likely(a == 1)) {
    // do something...
} else {
    // do something...
}
```

实际上，CPU 自身的动态分支预测已经是比较准的了，所以只有当非常确信 CPU 预测的不准，且能够知道实际的概率情况时，才建议使用这两种宏。

如何提升多核 CPU 的缓存命中率？

在单核 CPU，虽然只能执行一个线程，但是操作系统给每个线程分配了一个时间片，时间片用完了，就调度下一个线程，于是各个线程就按时间片交替地占用 CPU，从宏观上看起来各个线程同时在执行。

而现代 CPU 都是多核心的，线程可能在不同的 CPU 核心来回切换执行，这对 CPU Cache 不是有利的，虽然 L3 Cache 是多核心之间共享的，但是 L1 和 L2 Cache 都是每个核心独有的，**如果一个线程在不同核心来回切换，各个核心的缓存命中率就会受到影响**，相反如果线程都在同一个核心上执行，那么其数据的 L1 和 L2 Cache 的缓存命中率可以得到有效提高，缓存命中率高就意味着 CPU 可以减少访问内存的频率。

当有多个同时执行「计算密集型」的线程，为了防止因为切换到不同的核心，而导致缓存命中率下降的问题，我们可以把**线程绑定在某一个 CPU 核心上**，这样性能可以得到非常可观的提升。

在 Linux 上提供了 `sched_setaffinity` 方法，来实现将线程绑定到某个 CPU 核心这一功能。

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)

```
#define _GNU_SOURCE
#include <sched.h>
```

```
int sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);
```

总结

由于随着计算机技术的发展，CPU 与 内存的访问速度相差越来越多，如今差距已经高达好几百倍了，所以 CPU 内部嵌入了 CPU Cache 组件，作为内存与 CPU 之间的缓存层，CPU Cache 由于离 CPU 核心很近，所以访问速度也是非常快的，但由于所需材料成本比较高，它不像内存动辄几个 GB 大小，而是仅有几十 KB 到 MB 大小。

当 CPU 访问数据的时候，先是访问 CPU Cache，如果缓存命中的话，则直接返回数据，就不用每次都从内存读取数据了。因此，缓存命中率越高，代码的性能越好。

但需要注意的是，当 CPU 访问数据时，如果 CPU Cache 没有缓存该数据，则会从内存读取数据，但是并不是只读一个数据，而是一次性读取一块一块的数据存放到 CPU Cache 中，之后才会被 CPU 读取。

内存地址映射到 CPU Cache 地址里的策略有很多种，其中比较简单是直接映射 Cache，它巧妙的把内存地址拆分成「索引 + 组标记 + 偏移量」的方式，使得我们可以将很大的内存地址，映射到很小的 CPU Cache 地址里。

要想写出让 CPU 跑得更快的代码，就需要写出缓存命中率高的代码，CPU L1 Cache 分为数据缓存和指令缓存，因而需要分别提高它们的缓存命中率：

- 对于数据缓存，我们在遍历数据的时候，应该按照内存布局的顺序操作，这是因为 CPU Cache 是根据 CPU Cache Line 批量操作数据的，所以顺序地操作连续内存数据时，性能能得到有效的提升；
- 对于指令缓存，有规律的条件分支语句能够让 CPU 的分支预测器发挥作用，进一步提高执行的效率；

另外，对于多核 CPU 系统，线程可能在不同 CPU 核心来回切换，这样各个核心的缓存命中率就会受到影响，于是要想提高线程的缓存命中率，可以考虑把线程绑定 CPU 到某一个 CPU 核心。

关注作者

[目录](#)[侧边栏](#)[夜间](#)[技术群](#)[资料](#)[支持我](#)[上一篇](#)[下一篇](#)