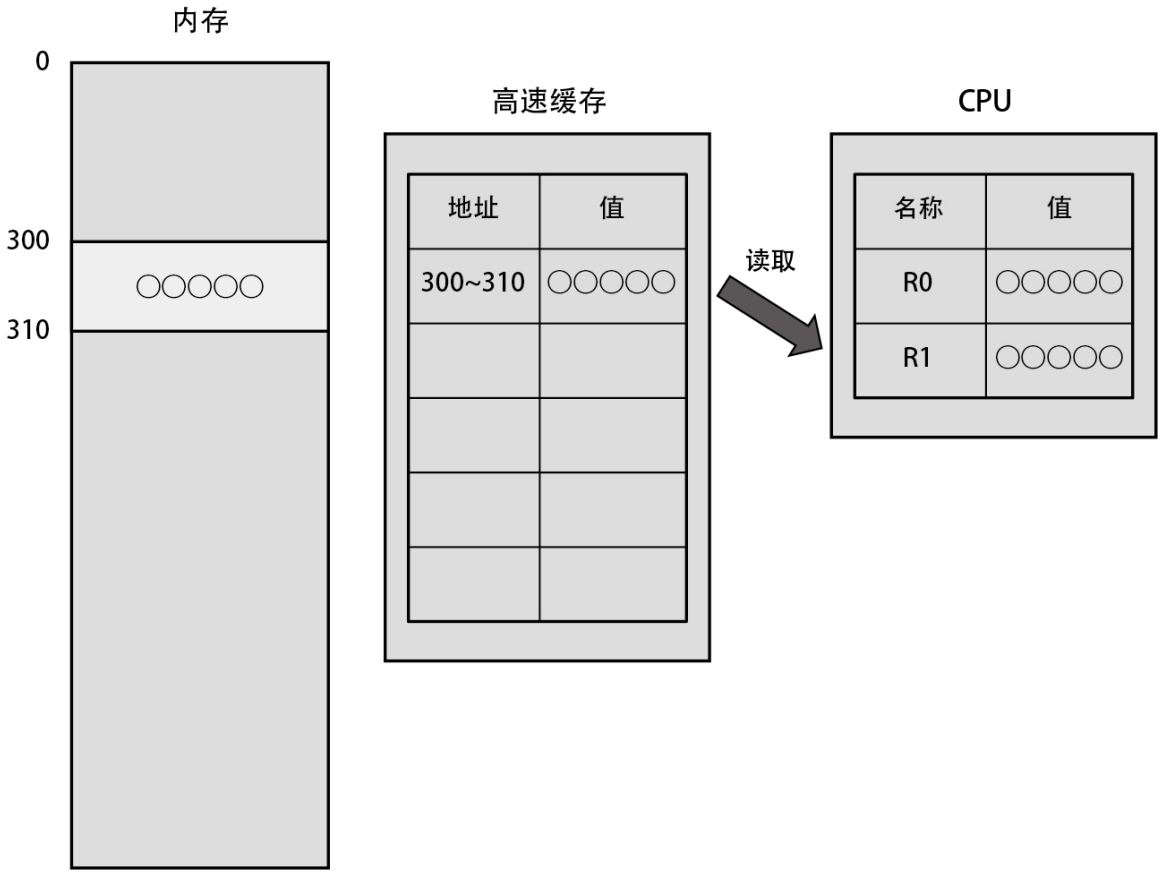

图 6-3　访问存在于高速缓存上的数据（高速）

当我们改写 R0 上的数据时，会发生什么呢？请看图 6-4。

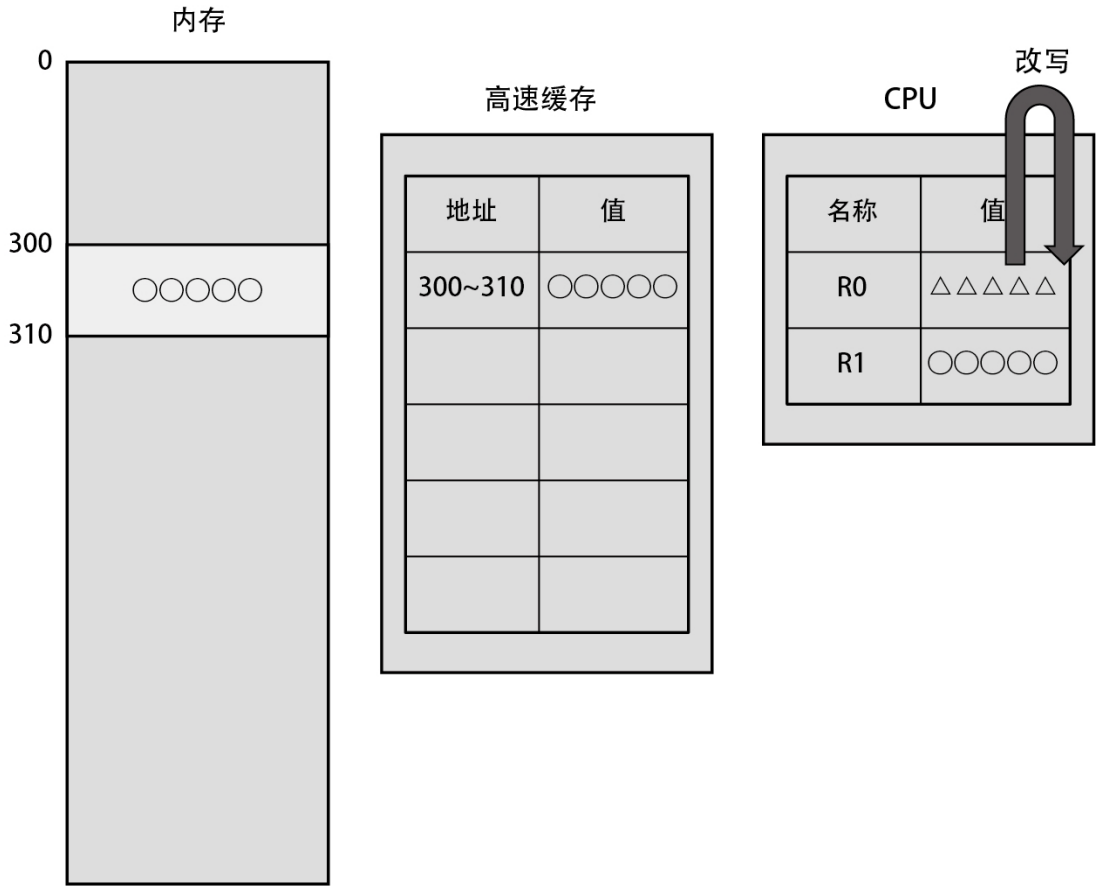

图 6-4　改写 R0 的值

此后，当需要将寄存器上的数据重新写回到地址 300 上时，首先会把改写后的数据写入高速缓存，如图 6-5 所示。此时依然以缓存块大小为单位写入数据。然后，为这些缓存块添加一个标记，以表明这部分从内存读取的数据被改写了。通常我们会称这些被标记的缓存块“脏了”。

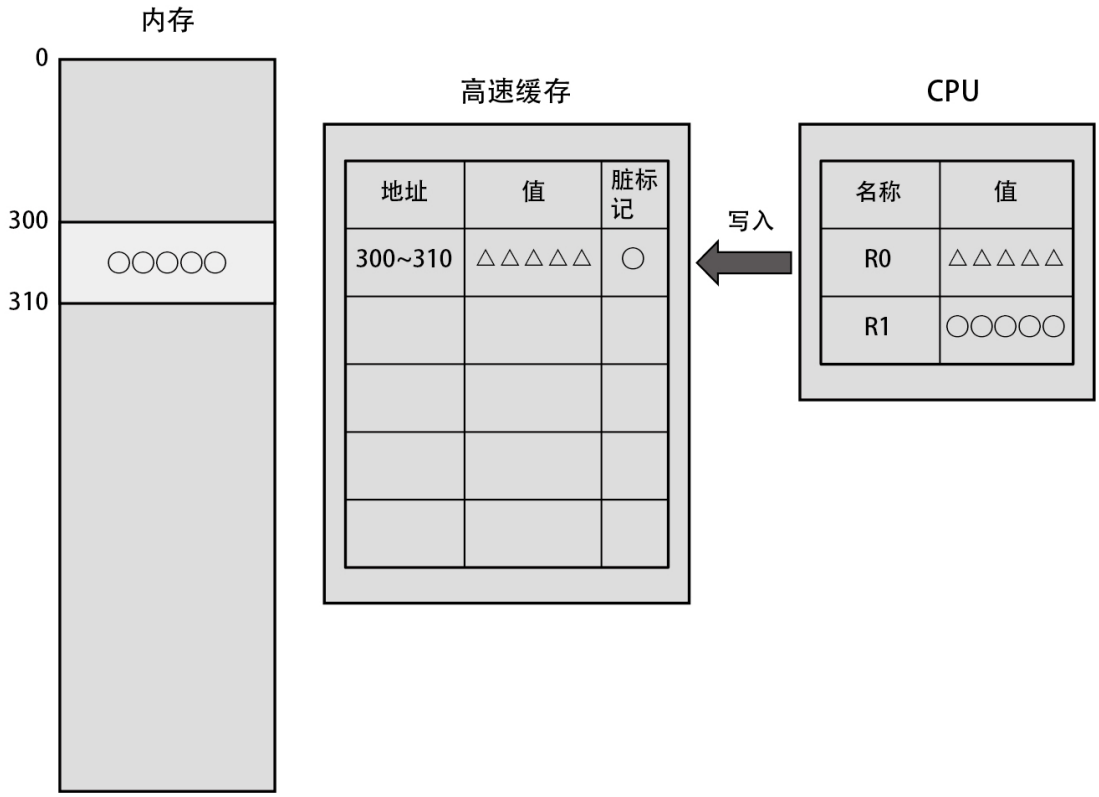

图 6-5　把改写后的数据写入高速缓存

这些被标记的数据会在写入高速缓存后的某个指定时间点，通过后台处理写入内存。随之，这些缓存块就不再脏了[4]，如图 6-6 所示。也就是说，只需要访问速度更快的高速缓存，即可完成图 6-5 中的写入操作。

[4]这种模式称为**回写**（write back）。另外还存在一种名为**直写**（write through）的模式。在直写模式下，缓存块会在变脏的一瞬间被立刻写入内存，但本书不涉及这种模式的内容。

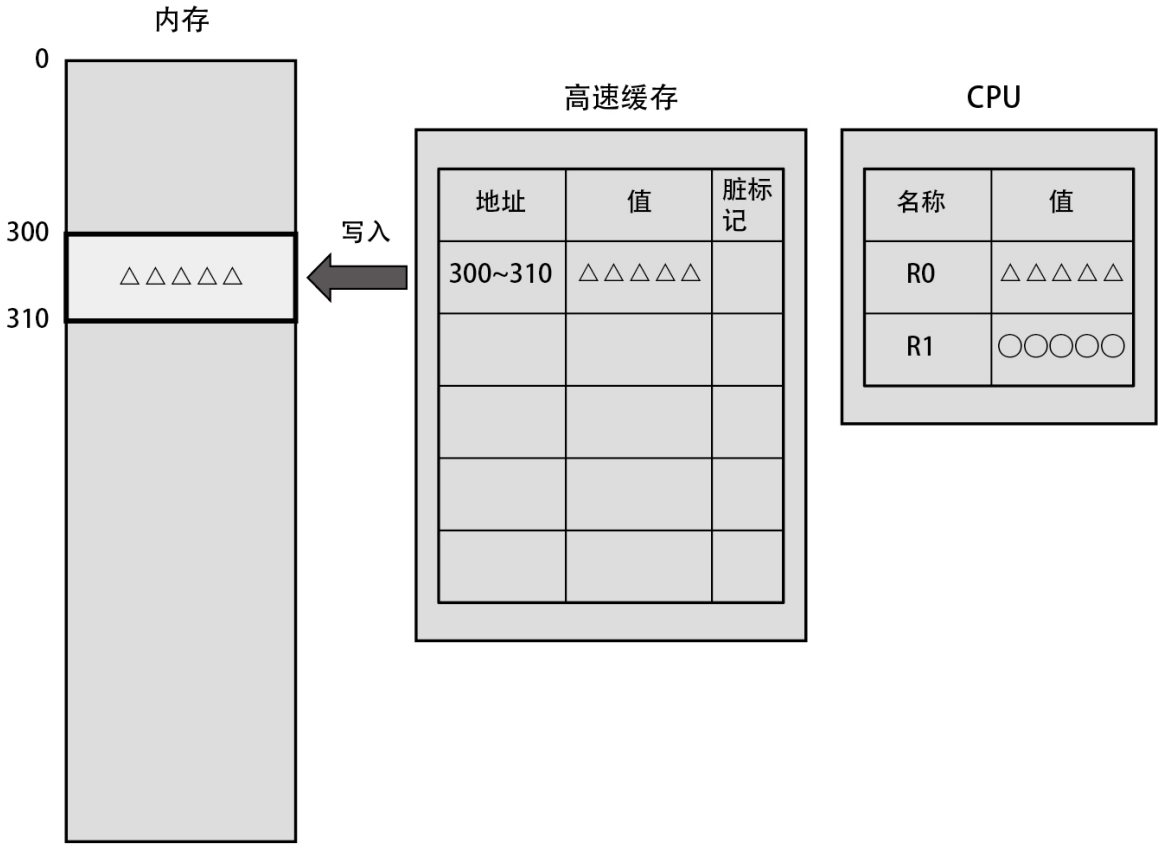

图 6-6　通过后台处理回写到内存中

在进程运行一段时间后，高速缓存中就会充满各种各样的数据，如图 6-7 所示。

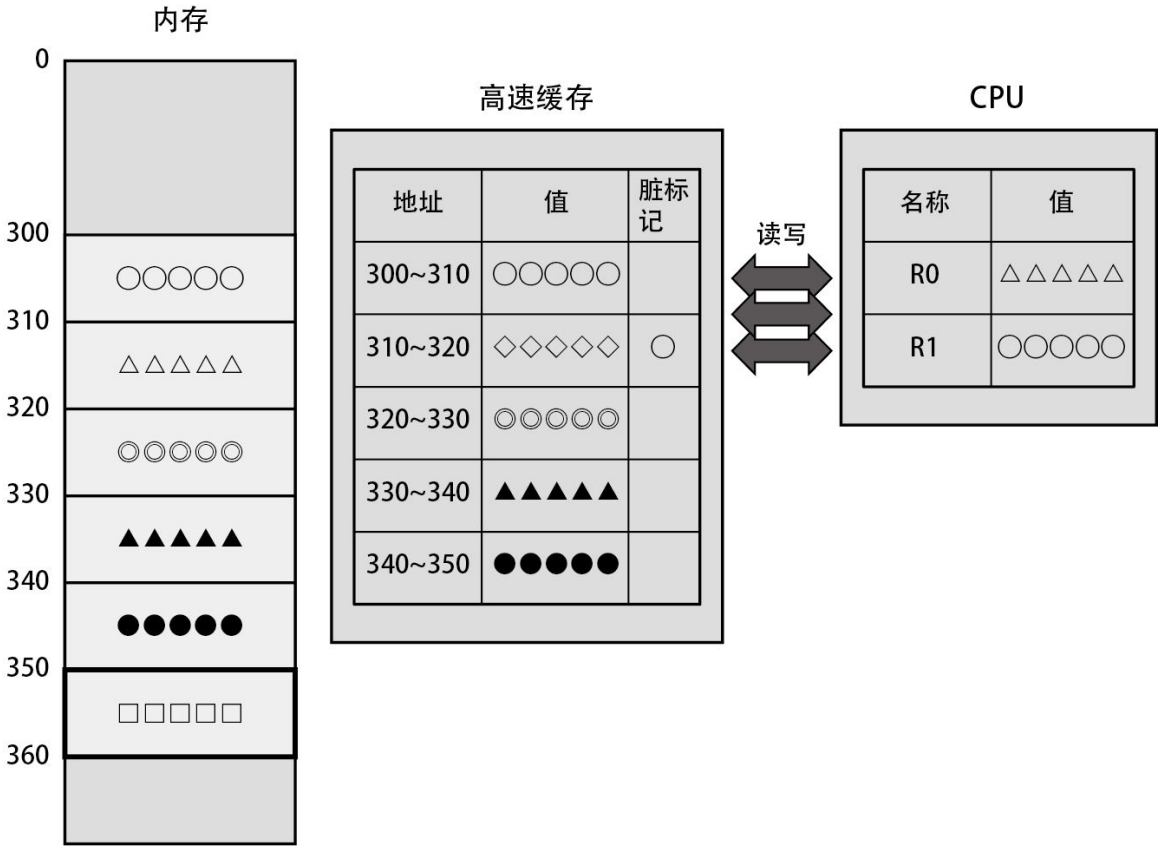

图 6-7　高速缓存中充满各种各样的数据

在这样的状态下，当 CPU 仅访问位于高速缓存上的数据时，访问速度比没有高速缓存时快得多，因为数据访问速度达到了高速缓存的读写速度。

6.2　高速缓存不足时

在高速缓存不足时，如果要读写高速缓存中尚不存在的数据，就要销毁一个现有的缓存块。例如，在图 6-7 的状态下读取地址 350 上的数据时，需要先销毁其中一个缓存块的数据（图 6-8），再把该地址上的数据复制到空出来的缓存块上。

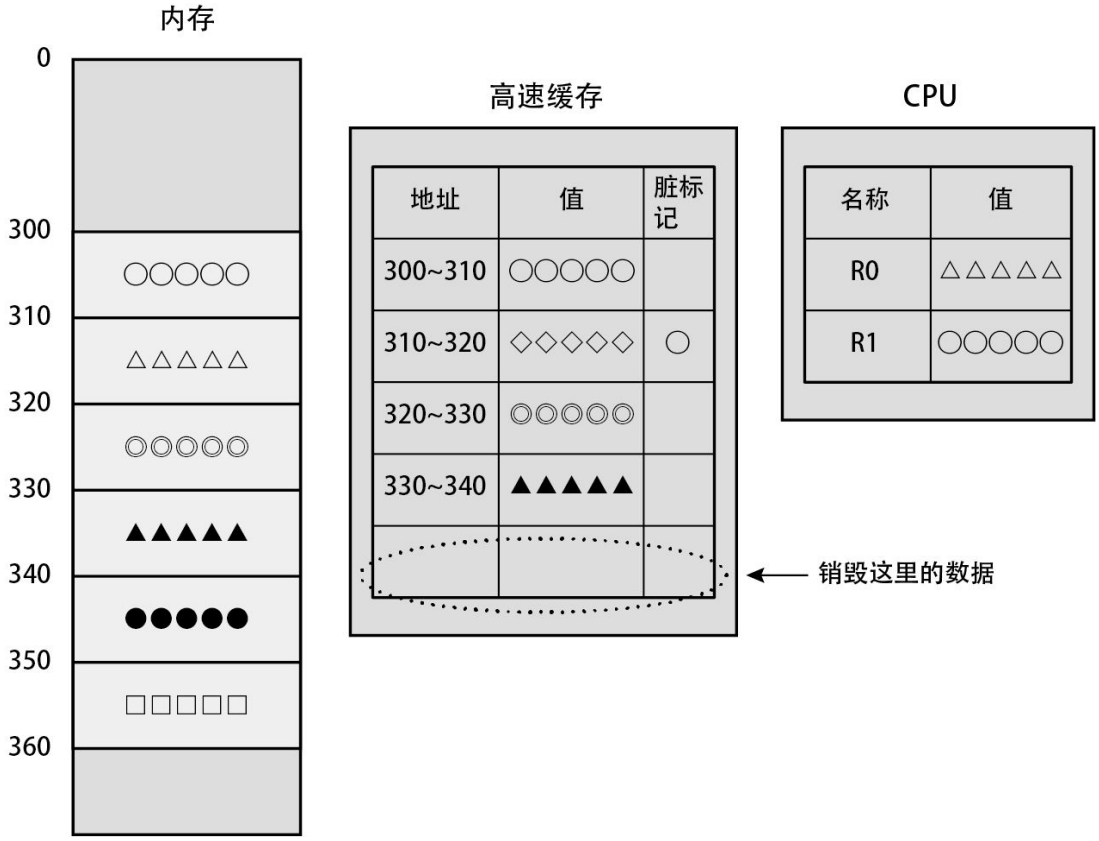

图 6-8　销毁缓存块的数据

把新的数据复制到缓存块上的情形如图 6-9 所示。

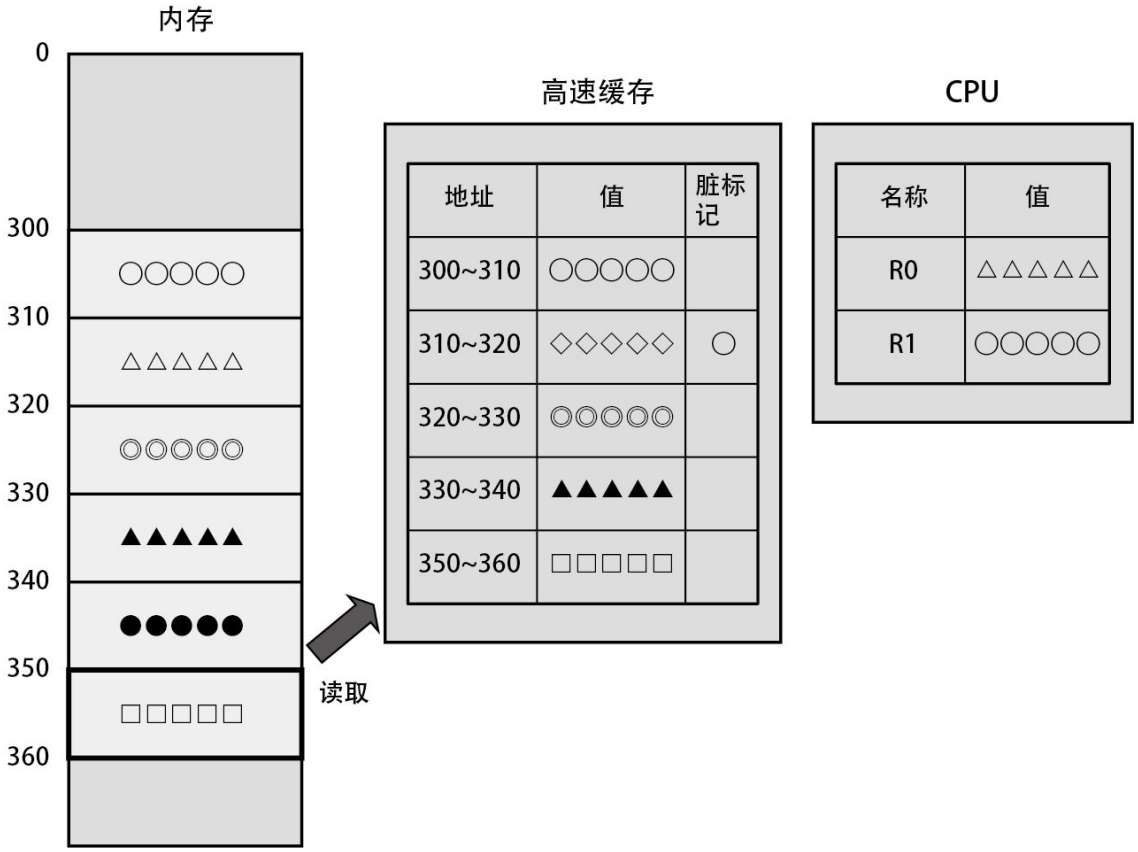

图 6-9　把新的数据复制到缓存块上

当需要销毁的缓存块脏了的时候，数据将在被销毁前被同步到内存中。如果在高速缓存不足，且所有缓存块都脏了的时候向内存发起访问，那么将因高速缓存频繁执行读写处理而发生系统抖动，与此同时性能也会大幅降低。

6.3 多级缓存

在最近的 x86_64 架构的 CPU 中，高速缓存都采用分层结构。各层级在容量、延迟以及“由哪些逻辑 CPU 共享”等方面各不相同。

构成分层结构的各高速缓存分别名为 L1、L2、L3（L 为 Level 的首字母）。不同规格的 CPU 中的缓存层级数量也不同。在各高速缓存中，最靠近寄存器、容量最小且速度最快的是 L1 缓存。层级的数字越大，离寄存器越远，速度越慢，但容量越大。

高速缓存的信息可从 *sys*devices/system/cpu/cpu0/cache/index0/[5]这一目录下的文件中查看。

[5]这是保存 CPU0 的 L1 缓存相关信息的目录。

- **type**：高速缓存中缓存的数据类型。**Data** 代表仅缓存数据，**Code** 代表仅缓存指令，**Unified** 代表两者都能缓存
- **shared_cpu_list**：共享该缓存的逻辑 CPU 列表
- **size**：容量大小
- **coherency_line_size**：缓存块大小

在笔者的计算机上，以上各项的内容如下所示。

文件名	名称	类型	共享的逻辑 CPU	容量大小（KB）	缓存块大小（字节）
index0	L1d	数据	不共享	32	64
index1	L1i	指令	不共享	64	64
index2	L2	数据与指令	不共享	512	64

文件名	名称	类型	共享的逻辑 CPU	容量大小（KB）	缓存块大小（字节）
index3	L3	数据与指令	0 ~ 3 共享、4 ~ 7 共享	8192	64

6.4 关于高速缓存的实验

下面，我们来看一下在高速缓存的影响下，在改变进程访问的数据量后，访问时间会发生什么样的变化。为此，需要编写一个实现下述要求的程序。

① 获取由命令行中的第 1 个参数指定的内存量（单位：KB）。

② 预先设定访问次数，每次访问都对获取的内存区域执行顺序访问。

③ 计算并显示单次访问所需要的时间 [步骤②的时间（单位：纳秒）/ 步骤②的访问次数]。

完成后的程序如代码清单 6-1 所示。

代码清单 6-1 cache 程序（cache.c）

```
#include <unistd.h>
#include <sys/mman.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

#define CACHE_LINE_SIZE  64
#define NLOOP            (4*1024UL*1024*1024)
#define NSECS_PER_SEC    1000000000UL

static inline long diff_nsec(struct timespec before, struct
                             timespec after)
{
    return ((after.tv_sec  NSECS_PER_SEC + after.tv_nsec)
           - (before.tv_sec  NSECS_PER_SEC + before.tv_nsec));
}

int main(int argc, char argv[])
{
    char progname;
    progname = argv[0];

    if (argc != 2) {
        fprintf(stderr, "usage: %s <size[KB]>\n", progname);
```

```
        exit(EXIT_FAILURE);
    }

    register int size;
    size = atoi(argv[1])  1024;
    if (!size) {
        fprintf(stderr, "size should be >= 1: %d
", size);
        exit(EXIT_FAILURE);
    }

    char buffer;
    buffer = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_
                  PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (buffer == (void *) -1)
        err(EXIT_FAILURE, "mmap() failed");

    struct timespec before, after;

    clock_gettime(CLOCK_MONOTONIC, &before);

    int i;
    for (i = 0; i < NLOOP  (size  CACHE_LINE_SIZE); i++) {
        long j;
        for (j = 0; j < size; j += CACHE_LINE_SIZE)
            buffer[j] = 0;
    }

    clock_gettime(CLOCK_MONOTONIC, &after);

    printf("%f\n", (double)diff_nsec(before, after) / NLOOP);

    if (munmap(buffer, size) == -1)
        err(EXIT_FAILURE, "munmap() failed");

    exit(EXIT_SUCCESS);
}
```

编译这个程序。与以往的例子不同的是，这里将启用最优化选项 -O3。这是因为本程序测试的是非常细微的性能差距，最优化后的程序更容易展现出高速缓存所产生的影响。

```
$ cc -O3 -o cache cache.c
```

在笔者的计算机上，各级高速缓存的容量分别为 64 KB、512 KB、8 MB。因此，从 4 KB 开始，一边成倍地增加作为参数的内存量，一边运行程序，直到 32 MB 为止。

```
$ for i in 4 8 16 32 64 128 256 512 1024 2048 4096 8192 ↲
16384 32768 ; do ./cache $i ; done
0.476930
0.363404
0.302903
0.279884
0.504577
0.502791
0.503517
0.602227
0.726228
0.730371
0.728870
1.898528
5.412608
5.282390
$
```

下面把实验结果绘制成如图 6-10 所示的图表。其中，内存量用 2^x（上标 x 为 x 轴上的值）表示。

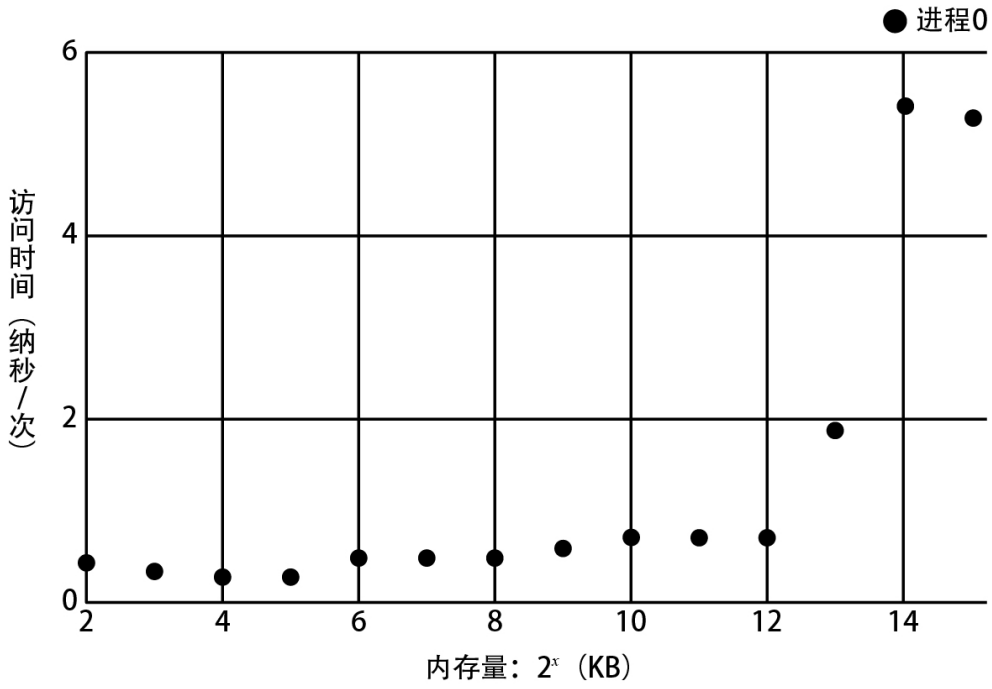

图 6-10　cache 程序的运行结果

在图 6-10 中，中央部分的数据不太直观，因此下面将 y 轴的值变更为“lg（访问时间）”，如图 6-11 所示。

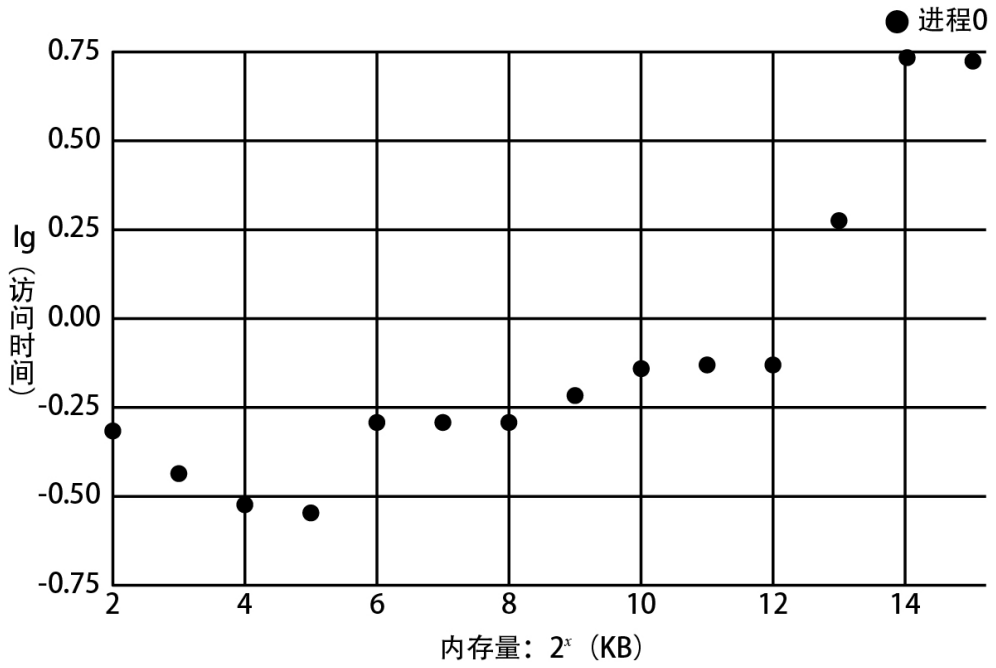

图 6-11　将 y 轴转变为对数轴后，cache 程序的运行结果

可以看到，访问时间大致上以各级高速缓存的容量为边界呈阶梯状变化。推荐大家也在自己的计算机上尝试一下。

在运行时，请大家根据各自计算机上的高速缓存的规格，更改源代码中 CACHE_LINE_SIZE 的值，以及参数的下限值与上限值。另外，如果在性能较低的 CPU 上运行本程序而出现了很久无法运行结束的情况，请适当减小源代码中 NLOOP 的值。

最后补充以下 3 点说明。

- 当内存量为 4 KB 和 16 KB 时，性能比 32 KB 时更好。这是因为测试结果受到了测试程序精度的影响。虽然用汇编语言能编写出精度更高的程序，但为了令源代码易于理解，本书仅采用 C 语言进行编写
- 本程序所测量的值准确来说并非只是单次数据访问的延迟，实际上还包含了其他指令的执行时间，例如为变量 `i` 执行数值递增处理的指令等。因此，实际的延迟会比测量得到的值更短一点
- 对于本实验来说，重要的并非测量得到的值，而是性能随着所访问的内存量的变化而大幅变化这一现象

6.5 访问局部性

通过前面的讲解可以明白，当进程的数据全部存在于高速缓存上时，数据访问速度将变为内存访问速度的几倍，达到高速缓存的访问速度。但在实际的系统中，是否能达到这么理想的状态呢？答案是，大部分情况下可以。大部分程序具有名为**访问局部性**的特征，具体如下所示。

- **时间局部性：在某一时间点被访问过的数据，有很大的可能性在不久的将来会再次被访问，例如循环处理中的代码段**
- **空间局部性：在某一时间点访问过某个数据后，有很大的可能性会继续访问其附近的其他数据，例如遍历数组元素**

因此，从比较短的时间区间上来看，进程倾向于访问很小范围内的内存，这个范围要远小于进程所获取的内存总量。如果这个范围的大小在高速缓存的容量以内，就非常完美了。

6.6 总结

一方面，通过将程序的工作量保持在高速缓存容量的范围内，可以大幅提升程序性能。对于重视运行速度的程序来说，要想最大限度地发挥高速缓存的优势，最重要的是花更多心思在数据结构与算法的设计上，以减小单位时间的内存访问范围。

另一方面，在更改系统设定等导致程序性能大幅下降的情况下，有可能是因为高速缓存容不下程序的数据。