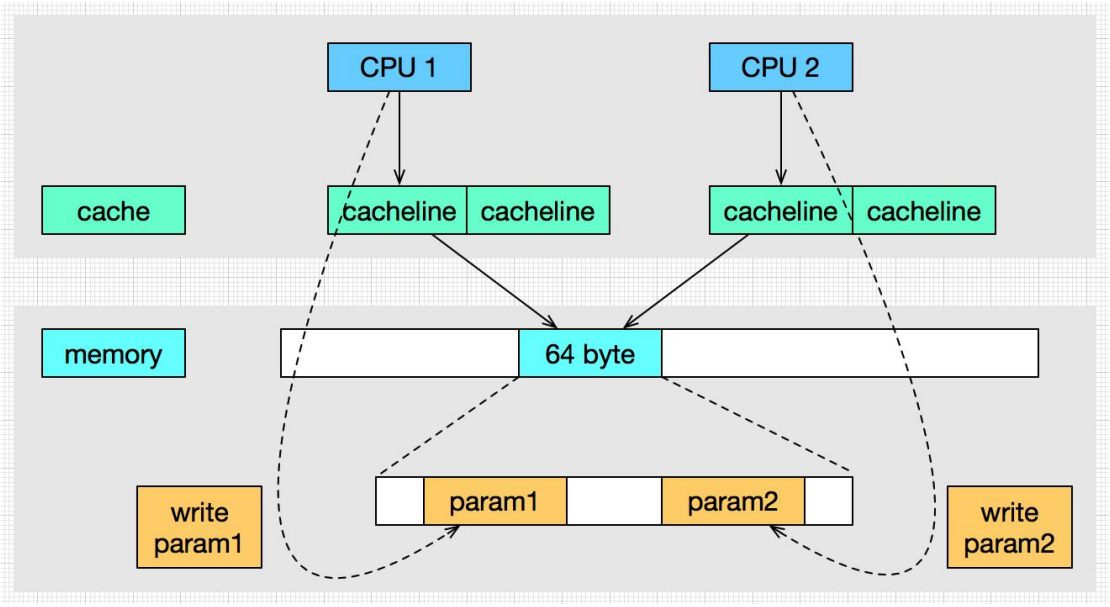
作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

伪共享的含义

现代 CPU 使用多核、多级缓存。多级缓存使用 cacheline 表示一块内存区间，cacheline 的大小为 64 字节，对应内存上的 64 字节，并且内存地址对齐。
多个核的 cacheline，可以映射到同一个内存区间。如果这个内存区间发送数据修改，对应的 cacheline 需要刷新，同步最新的数据。注意，同步操作有成本。

伪共享的含义：多核场景，并发写操作的多个变量在 cacheline 对应的同一个内存区间，缓存频繁同步导致频繁刷新 cacheline，进而导致性能降低。

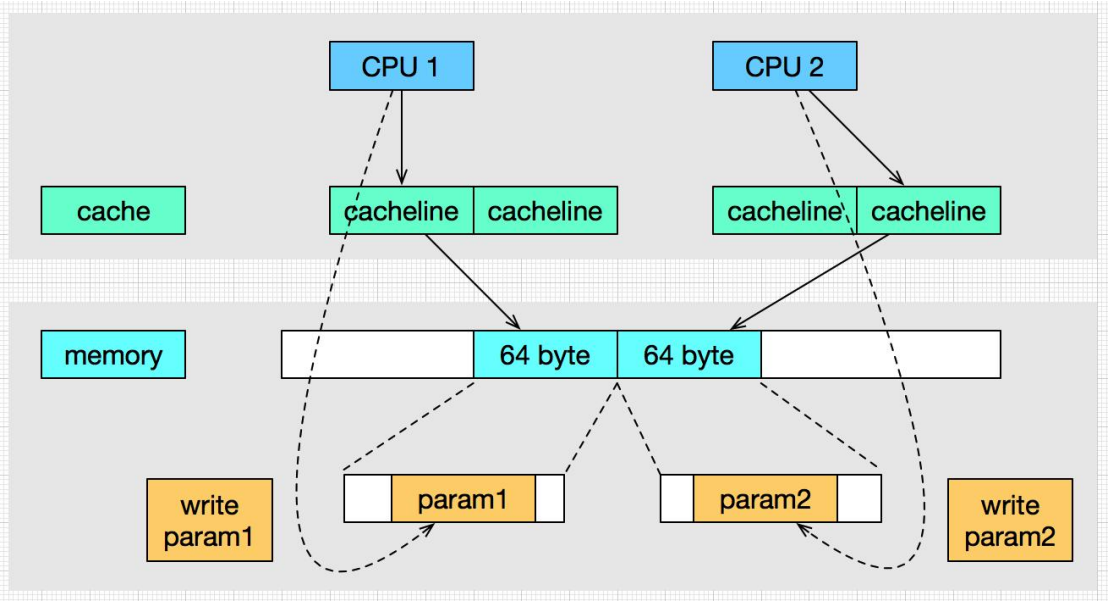
伪共享的示意图：
变量 param1、param2 在同一个 cacheline 内存区间。
CPU1 写变量 param1，导致 CPU2 刷新 cacheline。
CPU2 写变量 param2，导致 CPU1 刷新 cacheline。



避免伪共享的方法

在变量的前后填充冗余字节，使得单个变量独占 64 字节的内存空间，保证多个变量分散在多个 cacheline。
假设变量大小 8 字节，在之前填充 56 字节，在之后填充 56 字节，这样就能保证任何情况下变量独占 64 字节的内存空间。

避免伪共享的示意图：



用 C 分析避免伪共享的效果

编写代码： falseshare.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <pthread.h>
#include <sys/time.h>
#include <time.h>

// 消息，有填充
struct msg_with_pad
{
    char pad_before[56]; // 前方填充，56 字节
    int64_t count;       // 数据，8 字节
    char pad_after[56];  // 后方填充，56 字节
};

// 消息，没有填充
struct msg_no_pad
{
    int64_t count; // 数据，8 字节
};

// 2 个数组。
struct msg_with_pad msg_with_pad_array[3];
struct msg_no_pad msg_no_pad_array[3];
```

```
// 获得毫秒
uint64_t get_millis()
{
    struct timeval val;
    gettimeofday(&val, NULL);
    uint64_t millis = val.tv_sec * 1000;
    millis += val.tv_usec / 1000;
    return millis;
}

// 最大循环次数
int loop_max = 90000000;

// 线程，操作有填充的结构
void *thread_with_pad(void *param)
{
    // 当前线程操作的下标
    int *index_ptr = (int *)param;
    int index = *index_ptr;
    // 循环很多次
    for (int k = 0; k < loop_max; ++k)
    {
        // 写值。触发缓存同步
        msg_with_pad_array[index].count = 3333;
    }
    return NULL;
}

// 线程，操作没有填充的结构
void *thread_no_pad(void *param)
{
    // 当前线程操作的下标
    int *index_ptr = (int *)param;
    int index = *index_ptr;
    // 循环很多次
    for (int k = 0; k < loop_max; ++k)
    {
        // 写值。触发缓存同步
        msg_no_pad_array[index].count = 3333;
    }
    return NULL;
}

int main()
{
    printf(" struct msg_with_pad size = %d \n", sizeof(struct msg_with_pad));
    printf(" struct msg_no_pad   size = %d \n", sizeof(struct msg_no_pad));
    printf("\n");
```

```
    // 多个线程，分别操作多个数组元素
    int index0 = 0;
    int index1 = 1;
    int index2 = 2;

    // 有填充。解决了伪共享。
    uint64_t begin_millis_with_pad = get_millis();
    pthread_t t1;
    pthread_create(&t1, NULL, thread_with_pad, &index0);
    pthread_t t2;
    pthread_create(&t2, NULL, thread_with_pad, &index1);
    pthread_t t3;
    pthread_create(&t3, NULL, thread_with_pad, &index2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    uint64_t cost_millis_with_pad = get_millis() - begin_millis_with_pad;
    printf(" with_pad cost_millis = %llu \n", cost_millis_with_pad);

    // 没有填充。有伪共享问题。
    uint64_t begin_millis_no_pad = get_millis();
    pthread_t t5;
    pthread_create(&t5, NULL, thread_no_pad, &index0);
    pthread_t t6;
    pthread_create(&t6, NULL, thread_no_pad, &index1);
    pthread_t t7;
    pthread_create(&t7, NULL, thread_no_pad, &index2);
    pthread_join(t5, NULL);
    pthread_join(t6, NULL);
    pthread_join(t7, NULL);
    uint64_t cost_millis_no_pad = get_millis() - begin_millis_no_pad;
    printf(" no_pad   cost_millis = %llu \n", cost_millis_no_pad);
    printf("\n");
    return 0;
}
```

编译代码：

```
gcc falseshare.c -std=gnu99 -lpthread -o falseshare
```

运行代码：

```
[root@local falseshare]# ./falseshare
 struct msg_with_pad size = 120
 struct msg_no_pad   size = 8

 with_pad cost_millis = 344
 no_pad   cost_millis = 911

[root@local falseshare]# ./falseshare
 struct msg_with_pad size = 120
```

```
struct msg_no_pad   size = 8

with_pad cost_millis = 309
no_pad   cost_millis = 789
```

分析结果：
struct msg_with_pad 在属性 count 之前、之后分别填充了 56 字节，保证 count 独占 1 个 cacheline。
struct msg_no_pad 只包含属性 count，没有填充额外字节，count 可能与其他变量共用 1 个 cacheline。
数组 msg_with_pad_array，因为有填充字节，多个数组元素的 count 不在同一个 cacheline。
数组 msg_no_pad_array，因为没有填充字节，多个数组元素的 count 可能在同一个 cacheline。
有填充的场景，创建 3 个线程，执行函数 thread_with_pad，每个线程分别操作一个数组元素，触发缓存同步，最后记录耗时。
没有填充的场景，创建 3 个线程，执行函数 thread_no_pad，每个线程分别操作一个数组元素，触发缓存同步，最后记录耗时。
对比耗时，有填充的场景的耗时为 344、309，没有填充的场景的耗时为 911、789，差值达到 2 倍以上。

问题：为什么要在属性 count 之前、之后，各填充 56 字节？
如果只在前方或后方填充 56 字节，则变量 count 可能与其他变量共用 cacheline。此时无法保证 count 独占 cacheline。

前后填充的示意图：