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

atomic 的含义

atomic 表示原子操作，atomic 依赖 CAS 指令。
atomic 是提高并发、优化性能的重要方式。多个线程使用 atomic 操作同一个变量，不需要阻塞线程，atomic 广泛使用于并发场景。

CAS 指令保证单次写操作的原子性。多个线程并发写同一个变量，1 个线程写成功，其他线程可能写失败，需要循环执行 CAS 指令。
atomic 的实现过程：开启外层循环，执行 CAS 指令，如果成功就退出循环，否则继续循环。
atomic 的核心代码：

```
// 返回旧值。然后加值
int32_t atomic_get_and_add(int32_t *num, int32_t add)
{
    while (true)
    {
        // 取出旧值。CAS 更新。
        int32_t old_value = *num;
        int32_t new_value = old_value + add;
        bool ok = func_cas(num, old_value, new_value);
        if (ok)
        {
            return old_value;
        }
    }
}
```

使用 C 和汇编实现自定义的 atomic

编写代码： atomic.c

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

// 线程安全的 CAS
bool func_cas(int32_t *param_addr, int32_t old_value, int32_t new_value)
{
    int32_t ret_eax = 0;            // 记录返回的 eax
    asm volatile(                   // 内联汇编
```

```
        "lock \n"                    // 加锁。锁定缓存行
        "cmpxchgl %%ebx, (%%rcx) \n" // 执行 cas
        : "=a"(ret_eax)              // 返回值，eax，写到 ret_eax
        : "a"(old_value),            // 旧值，写到 rax
          "b"(new_value),            // 新值，写到 rcx
          "c"(param_addr));          // 地址，写到 rdx

    if (ret_eax == old_value) // cas 成功
    {
        return true;
    }
    else // cas 失败
    {
        return false;
    }
}

// 加值
void atomic_add(int32_t *num, int32_t add)
{
    while (true)
    {
        // 取出旧值。CAS 更新。
        int32_t old_value = *num;
        int32_t new_value = old_value + 1;
        bool ok = func_cas(num, old_value, new_value);
        if (ok)
        {
            return;
        }
    }
}

// 加 1
void atomic_incr(int32_t *num, int32_t add)
{
    atomic_add(num, 1);
}

// 返回旧值。然后加值
int32_t atomic_get_and_add(int32_t *num, int32_t add)
{
    while (true)
    {
        // 取出旧值。CAS 更新。
        int32_t old_value = *num;
        int32_t new_value = old_value + add;
        bool ok = func_cas(num, old_value, new_value);
        if (ok)
```

```
        {
            return old_value;
        }
    }
}

// 返回旧值。然后加 1
int32_t atomic_get_and_incr(int32_t *num)
{
    return atomic_get_and_add(num, 1);
}

// 返回旧值。然后减值
int32_t atomic_get_and_sub(int32_t *num, int32_t sub)
{
    return atomic_get_and_add(num, -sub);
}

// 返回旧值。然后减 1
int32_t atomic_get_and_decr(int32_t *num)
{
    atomic_get_and_sub(num, 1);
}

// 整数变量。
int32_t count = 0;

// 线程函数
void *thread_func(void *param)
{
    for (int e = 0; e < 300000; ++e)
    {
        // 原子操作
        int32_t old1 = atomic_get_and_add(&count, 3);
        int32_t old2 = atomic_get_and_sub(&count, 1);
    }
    return NULL;
}

int main()
{
    printf(" count before = %d \n", count);

    // 创建多个线程
    pthread_t t1;
    pthread_create(&t1, NULL, thread_func, NULL);
    pthread_t t2;
    pthread_create(&t2, NULL, thread_func, NULL);
    pthread_t t3;
```

```
    pthread_create(&t3, NULL, thread_func, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    printf(" count after  = %d \n", count);
    return 0;
}
```

编译代码：

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

运行代码：

```
atomic# ./atomic
 count before = 0
 count after  = 1800000
```

分析结果：

函数 func_cas 使用 CAS 指令 cmpxchgl，实现原子更新整数变量。

函数 atomic_add、atomic_get_and_add 使用函数 func_cas，实现多线程原子写操作。其他 atomic 函数依赖这 2 个函数。

创建 3 个线程，并发更新同一个变量，先给变量加上 3，再给变量减去 1，循环 300000 次。

计算 `3 * (0+3-1) * 300000 = 1800000` ，与输出结果相同，说明并发写正常。