



下载APP



08 | 锁：并发操作中，解决数据同步的四种方法

2021-05-26 LMOS

《操作系统实战45讲》

课程介绍 >



讲述：陈晨

时长 17:30 大小 16.04M



你好，我是 LMOS。

我们在前面的课程中探索了，开发操作系统要了解的最核心的硬件——CPU、MMU、Cache、内存，知道了它们的工作原理。在程序运行中，它们起到了至关重要的作用。

在开发我们自己的操作系统以前，还不能一开始就把机器跑起来，而是先要弄清楚数据同步的问题。如果不解决掉数据同步的问题，后面机器跑起来，就会出现很多不可预知的结果。



通过这节课，我会给你讲清楚为什么在并发操作里，很可能得不到预期的访问数据，还会带你分析这个问题的原因以及解决方法。有了这样一个研究、解决问题的过程，对最重要的几种锁（**原子变量，关中断，信号量，自旋锁**），你就能做到心中有数了。

非预期结果的全局变量

来看看下面的代码，描述的是一个线程中的函数和中断处理函数，它们分别对一个全局变量执行加 1 操作，代码如下。

[复制代码](#)

```
1  int a = 0;
2  void interrupt_handle()
3  {
4      a++;
5  }
6  void thread_func()
7  {
8      a++;
9  }
10
```

首先我们梳理一下编译器的翻译过程，通常编译器会把 `a++` 语句翻译成这 3 条指令。

1. 把 `a` 加载某个寄存器中。
2. 这个寄存器加 1。
3. 把这个寄存器写回内存。

那么不难推断，可能导致结果不确定的情况是这样的：`thread_func` 函数还没运行完第 2 条指令时，中断就来了。

因此，CPU 转而处理中断，也就是开始运行 `interrupt_handle` 函数，这个函数运行完 `a=1`，CPU 还会回去继续运行第 3 条指令，此时 `a` 依然是 1，这显然是错的。

下面来看一下表格，你就明白了。

显然在 `t2` 时刻发生了中断，导致了 `t2` 到 `t4` 运行了 `interrupt_handle` 函数，`t5` 时刻 `thread_func` 又恢复运行，导致 `interrupt_handle` 函数中 `a` 的操作丢失，因此出错。

方法一：原子操作 拿下单体变量

要解决上述场景中的问题，有这样两种思路。一种是**把 a++ 变成原子操作**，这里的原子是不可分隔的，也就是说要 a++ 这个操作不可分隔，即 a++ 要么不执行，要么一口气执行完；另一种就是**控制中断**，比如在执行 a++ 之前关掉中断，执行完了之后打开中断。

我们先来看看原子操作，显然靠编译器自动生成原子操作不太可能。第一，编译器没有这么智能，能检测哪个变量需要原子操作；第二，编译器必须要考虑代码的移植性，例如有些硬件平台支持原子操作的机器指令，有的硬件平台不支持原子操作。

既然实现原子操作无法依赖于具体编译器，那就需要我们自己动手，x86 平台支持很多原子指令，我们只需要直接应用这些指令，比如原子加、原子减，原子读写等，用汇编代码写出对应的原子操作函数就行了。

好在现代 C 语言已经支持嵌入汇编代码，可以在 **C 函数中按照特定的方式嵌入汇编代码**了，实现原子操作就更方便了，代码如下。

[复制代码](#)

```
1 //定义一个原子类型
2 typedef struct s_ATOMIC{
3     volatile s32_t a_count; //在变量前加上volatile，是为了禁止编译器优化，使其每次都从
4 }atomic_t;
5 //原子读
6 static inline s32_t atomic_read(const atomic_t *v)
7 {
8     //x86平台取地址处是原子
9     return (*(volatile u32_t*)&(v->a_count));
10 }
11 //原子写
12 static inline void atomic_write(atomic_t *v, int i)
13 {
14     //x86平台把一个值写入一个地址处也是原子的
15     v->a_count = i;
16 }
17 //原子加上一个整数
18 static inline void atomic_add(int i, atomic_t *v)
19 {
20     __asm__ __volatile__ ("lock; " "addl %1,%0"
21                          : "+m" (v->a_count)
22                          : "ir" (i));
23 }
24 //原子减去一个整数
25 static inline void atomic_sub(int i, atomic_t *v)
```

```
26 {
27     __asm__ __volatile__("lock;" "subl %1,%0"
28                          : "+m" (v->a_count)
29                          : "ir" (i));
30 }
31 //原子加1
32 static inline void atomic_inc(atomic_t *v)
33 {
34     __asm__ __volatile__("lock;" "incl %0"
35                          : "+m" (v->a_count));
36 }
37 //原子减1
38 static inline void atomic_dec(atomic_t *v)
39 {
40     __asm__ __volatile__("lock;" "decl %0"
41                          : "+m" (v->a_count));
42 }
43
```

以上代码中，加上 lock 前缀的 addl、subl、incl、decl 指令都是原子操作，lock 前缀表示锁定总线。

我们还是来看看 GCC 支持嵌入汇编代码的模板，不同于其它 C 编译器支持嵌入汇编代码的方式，为了优化用户代码，GCC 设计了一种特有的嵌入方式，它规定了汇编代码嵌入的形式和嵌入汇编代码需要由哪几个部分组成，如下面代码所示。

```
1 __asm__ __volatile__(代码部分:输出部分列表: 输入部分列表:损坏部分列表);
```

[复制代码](#)

可以看到代码模板从 __asm__ 开始（当然也可以是 asm），紧跟着 __volatile__，然后是跟着一对括号，最后以分号结束。括号里大致分为 4 个部分：

1. 汇编代码部分，这里是实际嵌入的汇编代码。
2. 输出列表部分，让 GCC 能够处理 C 语言左值表达式与汇编代码的结合。
3. 输入列表部分，也是让 GCC 能够处理 C 语言表达式、变量、常量，让它们能够输入到汇编代码中去。

4. 损坏列表部分，告诉 GCC 汇编代码中用到了哪些寄存器，以便 GCC 在汇编代码运行前，生成保存它们的代码，并且在生成的汇编代码运行后，恢复它们（寄存器）的代码。

它们之间用冒号隔开，如果只有汇编代码部分，后面的冒号可以省略。但是有输入列表部分而没有输出列表部分的时候，输出列表部分的冒号就必须写，否则 GCC 没办法判断，同样的道理对于其它部分也一样。

这里不会过多展开讲这个技术，详情可参阅 [GCC 手册](#)。你可以重点看 GAS 相关的章节。

下面将用上面一个函数 `atomic_add` 为例子说一下，如下所示。

[复制代码](#)

```
1 static inline void atomic_add(int i, atomic_t *v)
2 {
3     __asm__ __volatile__ ("lock;" "addl %1,%0"
4                          : "+m" (v->a_count)
5                          : "ir" (i));
6 }
7 // "lock;" "addl %1,%0" 是汇编指令部分，%1,%0是占位符，它表示输出、输入列表中变量或表态式
8 //: "+m" (v->a_count) 是输出列表部分，“+m”表示(v->a_count)和内存地址关联
9 //: "ir" (i) 是输入列表部分，“ir”表示i是和立即数或者寄存器关联
```

有了这些原子操作函数之后，前面场景中的代码就变成下面这样了：无论有没有中断，或者什么时间来中断，都不会出错。

[复制代码](#)

```
1 atomic_t a = {0};
2 void interrupt_handle()
3 {
4     atomic_inc(&a);
5 }
6 void thread_func()
7 {
8     atomic_inc(&a);
9 }
```

好，说完了原子操作，我们再看看怎么用中断控制的思路解决数据并发访问的问题。

方法二：中断控制 搞定复杂变量

中断是 CPU 响应外部事件的重要机制，时钟、键盘、硬盘等 IO 设备都是通过发出中断来请求 CPU 执行相关操作的（即执行相应的中断处理代码），比如下一个时钟到来、用户按下了键盘上的某个按键、硬盘已经准备好了数据。

但是中断处理代码中如果操作了其它代码的数据，这就需要相应的控制机制了，这样才能保证在操作数据过程中不发生中断。

你或许在想，可以用原子操作啊？不过，**原子操作只适合于单体变量**，如整数。操作系统的数据结构有的可能有几百字节大小，其中可能包含多种不同的基本数据类型。这显然用原子操作无法解决。

下面，我们就要写代码实现关闭开启、中断了，x86 CPU 上关闭、开启中断有专门的指令，即 cli、sti 指令，它们主要是对 CPU 的 eflags 寄存器的 **IF 位**（第 9 位）进行清除和设置，CPU 正是通过此位来决定是否响应中断信号。这两条指令只能 Ring0 权限才能执行，代码如下。

[复制代码](#)

```
1 //关闭中断
2 void hal_cli()
3 {
4     __asm__ __volatile__("cli": : : "memory");
5 }
6 //开启中断
7 void hal_sti()
8 {
9     __asm__ __volatile__("sti": : : "memory");
10 }
11 //使用场景
12 void foo()
13 {
14     hal_cli();
15     //操作数据.....
16     hal_sti();
17 }
18 void bar()
19 {
20     hal_cli();
21     //操作数据.....
22     hal_sti();
23 }
```

你可以自己思考一下，前面这段代码效果如何？

它看似完美地解决了问题，其实有重大缺陷，`hal_cli()`，`hal_sti()`，**无法嵌套使用**，看一个例子你就明白了，代码如下。

[复制代码](#)

```
1 void foo()
2 {
3     hal_cli();
4     //操作数据第一步.....
5     hal_sti();
6 }
7 void bar()
8 {
9     hal_cli();
10    foo();
11    //操作数据第二步.....
12    hal_sti();
13 }
```

上面代码的关键问题在 `bar` 函数在关中断下调用了 `foo` 函数，`foo` 函数中先关掉中断，处理好数据然后开启中断，回到 `bar` 函数中，`bar` 函数还天真地以为中断是关闭的，接着处理数据，以为不会被中断抢占。

那么怎么解决上面的问题呢？我们只要修改一下开启、关闭中断的函数就行了。

我们可以这样操作：在关闭中断函数中先保存 `eflags` 寄存器，然后执行 `cli` 指令，在开启中断函数中直接恢复之前保存的 `eflags` 寄存器就行了，具体代码如下。

[复制代码](#)

```
1 typedef u32_t cpuflg_t;
2 static inline void hal_save_flags_cli(cpuflg_t* flags)
3 {
4     __asm__ __volatile__(
5         "pushfl \t\n" //把eflags寄存器压入当前栈顶
6         "cli \t\n" //关闭中断
7         "popl %0 \t\n" //把当前栈顶弹出到flags为地址的内存中
8         : "=m"(*flags)
9         :
10        : "memory"
11    );
12 }
```

```
13 static inline void hal_restore_flags_sti(cpuflg_t* flags)
14 {
15     __asm__ __volatile__(
16         "pushl %0 \t\n" //把flags为地址处的值寄存器压入当前栈顶
17         "popfl \t\n"    //把当前栈顶弹出到flags寄存器中
18         :
19         : "m"(*flags)
20         : "memory"
21     );
22 }
```

从上面的代码中不难发现，硬件工程师早就想到了如何解决在嵌套函数中关闭、开启中断的问题：pushfl 指令把 eflags 寄存器压入当前栈顶，popfl 把当前栈顶的数据弹出到 eflags 寄存器中。

hal_restore_flags_sti() 函数的执行，是否开启中断完全取决于上一次 eflags 寄存器中的值，并且 popfl 指令只会影响 eflags 寄存器中的 IF 位。这样，无论函数嵌套调用多少层都没有问题。

方法三：自旋锁 协调多核心 CPU

前面说的控制中断，看似解决了问题，那是因为以前是单 CPU，同一时刻只有一条代码执行流，除了中断会中止当前代码执行流，转而运行另一条代码执行流（中断处理程序），再无其它代码执行流。这种情况下只要控制了中断，就能安全地操作全局数据。

但是我们都知道，现在情况发生了改变，CPU 变成了多核心，或者主板上安装了多颗 CPU，同一时刻下系统中存在多条代码执行流，控制中断只能控制本地 CPU 的中断，无法控制其它 CPU 核心的中断。

所以，原先通过控制中断来维护全局数据安全的方案失效了，这就需要全新的机制来处理这样的情况，于是就轮到自旋锁登场了。


我们先看看自旋锁的原理，它是这样的：首先读取锁变量，判断其值是否已经加锁，如果未加锁则执行加锁，然后返回，表示加锁成功；如果已经加锁了，就要返回第一步继续执行后续步骤，因而得名自旋锁。为了让你更好理解，下面来画一个图描述这个算法。

自旋锁原理示意图

这个算法看似很好，但是想要正确执行它，就**必须保证读取锁变量和判断并加锁的操作是原子执行的**。否则，CPU0 在读取了锁变量之后，CPU1 读取锁变量判断未加锁执行加锁，然后 CPU0 也判断未加锁执行加锁，这时就会发现两个 CPU 都加锁成功，因此这个算法出错了。

怎么解决这个问题呢？这就要找硬件要解决方案了，x86 CPU 给我们提供了一个原子交换指令，xchg，它可以让寄存器里的一个值跟内存空间中的一个值做交换。例如，让 eax=memlock，memlock=eax 这个动作是原子的，不受其它 CPU 干扰。

下面我们就去实现自旋锁，代码如下所示。

 复制代码

```

1 //自旋锁结构
2 typedef struct
3 {
4     volatile u32_t lock; //volatile可以防止编译器优化，保证其它代码始终从内存加载lock
5 } spinlock_t;
6 //锁初始化函数
7 static inline void x86_spin_lock_init(spinlock_t * lock)
8 {
9     lock->lock = 0; //锁值初始化为0是未加锁状态
10 }
11 //加锁函数
12 static inline void x86_spin_lock(spinlock_t * lock)
13 {
14     __asm__ __volatile__ (
15         "1: \n"
16         "lock; xchg  %0, %1 \n" //把值为1的寄存器和lock内存中的值进行交换
17         "cmpl  $0, %0 \n" //用0和交换回来的值进行比较
18         "jnz   2f \n" //不等于0则跳转后面2标号处运行
19         "jmp  3f \n" //若等于0则跳转后面3标号处返回
20         "2: \n"
21         "cmpl  $0, %1 \n" //用0和lock内存中的值进行比较
22         "jne   2b \n" //若不等于0则跳转到前面2标号处运行继续比较
23         "jmp  1b \n" //若等于0则跳转到前面1标号处运行，交换并加锁
24         "3: \n" :
25         : "r"(1), "m"(*lock));
26 }
27 //解锁函数
28 static inline void x86_spin_unlock(spinlock_t * lock)
29 {
30     __asm__ __volatile__(
31         "movl  $0, %0 \n" //解锁把lock内存中的值设为0就行
32         :
33         : "m"(*lock));
34 }


```

上述代码的中注释已经很清楚了，关键点在于 `xchg` 指令，`xchg %0, %1`。

其中，`%0` 对应 `"r"(1)`，表示由编译器自动分配一个通用寄存器，并填入值 1，例如 `mov eax, 1`。而 `%1` 对应 `"m"(*lock)`，表示 `lock` 是内存地址。把 1 和内存中的值进行交换，若内存中是 1，则不会影响；因为本身写入就是 1，若内存中是 0，一交换，内存中就变成了 1，即加锁成功。

自旋锁依然有中断嵌套的问题，也就是说，在使用自旋锁的时候我们仍然要注意中断。

在中断处理程序访问某个自旋锁保护的某个资源时，依然有问题，所以我们要写的自旋锁函数必须适应这样的中断环境，也就是说，它需要在处理中断的过程中也能使用，如下所示。

 复制代码

```
1 static inline void x86_spin_lock_disable_irq(spinlock_t * lock,cpuflg_t* flags
2 {
3     __asm__ __volatile__(
4         "pushfq                \n\t"
5         "cli                    \n\t"
6         "popq %0                \n\t"
7         "1:                    \n\t"
8         "lock; xchg %1, %2 \n\t"
9         "cmpl $0,%1            \n\t"
10        "jnz 2f                \n\t"
11        "jmp 3f                \n\t"
12        "2:                    \n\t"
13        "cmpl $0,%2            \n\t"
14        "jne 2b                \n\t"
15        "jmp 1b                \n\t"
16        "3:                    \n\t"
17        : "=m"(*flags)
18        : "r"(1), "m"(*lock));
19 }
20 static inline void x86_spin_unlock_enabled_irq(spinlock_t* lock,cpuflg_t* flag
21 {
22     __asm__ __volatile__(
23         "movl $0, %0\n\t"
24         "pushq %1 \n\t"
25         "popfq \n\t"
26         :
27         : "m"(*lock), "m"(*flags));
28 }
```

以上代码实现了关中断下获取自旋锁，以及恢复中断状态释放自旋锁。在中断环境下也完美地解决了问题。

方法四：信号量 CPU 时间管理大师

无论是原子操作，还是自旋锁，都不适合长时间等待的情况，因为有很多资源（数据）它有一定的时间性，你想去获取它，CPU 并不能立即返回给你，而是要等待一段时间，才能把数据返回给你。这种情况，你用自旋锁来同步访问这种资源，你会发现这是对 CPU 时间的巨大浪费。

下面我们看看另一种同步机制，既能对资源数据进行保护（同一时刻只有一个代码执行流访问），又能在资源无法满足的情况下，让 CPU 可以执行其它任务。

如果你翻过操作系统的理论书，应该对信号量这个词并不陌生。信号量是 1965 年荷兰学者 Edsger Dijkstra 提出的，是一种用于资源互斥或者进程间同步的机制。这里我们就来看看如何实现这一机制。

你不妨想象这样一个情境：微信等待你从键盘上的输入信息，然后把这个信息发送出去。

这个功能我们怎么实现呢？下面我们就来说说实现它的一般方法，当然具体实现中可能不同，但是原理是相通的，具体如下。

1. 一块内存，相当于缓冲区，用于保存键盘的按键码。
2. 需要一套控制机制，比如微信读取这个缓冲区，而该缓冲区为空时怎么处理；该缓冲区中有了按键码，却没有代码执行流来读取，又该怎么处理。

我们期望是这样的，一共有三点。

1. 当微信获取键盘输入信息时，发现键盘缓冲区中是空的，就进入等待状态。
2. 同一时刻，只能有一个代码执行流操作键盘缓冲区。

3. 当用户按下键盘时，我们有能力把按键码写入缓冲区中，并且能看一看微信或者其它程序是否在等待该缓冲区，如果是就重新激活微信和它的程序，让它们重新竞争读取键盘缓冲区，如果竞争失败依然进入等待状态。

其实以上所述无非是三个问题：**等待、互斥、唤醒（即重新激活等待的代码执行流）**。

这就需要一种全新的数据结构来解决这些问题。根据上面的问题，这个数据结构至少需要一个变量来表示互斥，比如大于 0 则代码执行流可以继续运行，等于 0 则让代码执行流进入等待状态。还需要一个等待链，用于保存等待的代码执行流。

这个数据结构的实现代码如下所示。

[复制代码](#)

```
1 #define SEM_FLG_MUTEX 0
2 #define SEM_FLG_MULTI 1
3 #define SEM_MUTEX_ONE_LOCK 1
4 #define SEM_MULTI_LOCK 0
5 //等待链数据结构，用于挂载等待代码执行流（线程）的结构，里面有用于挂载代码执行流的链表和计数器
6 typedef struct s_KWLST
7 {
8     spinlock_t wl_lock;
9     uint_t    wl_tdnr;
10    list_h_t  wl_list;
11 }kwlst_t;
12 //信号量数据结构
13 typedef struct s_SEM
14 {
15     spinlock_t sem_lock; //维护sem_t自身数据的自旋锁
16     uint_t    sem_flg; //信号量相关的标志
17     sint_t    sem_count; //信号量计数值
18     kwlst_t   sem_waitlst; //用于挂载等待代码执行流（线程）结构
19 }sem_t;
```

搞懂了信号量的结构，我们再来看看信号量的一般用法，注意信号量在使用之前需要**先进初始化**。这里假定信号量数据结构中的 `sem_count` 初始化为 1，`sem_waitlst` 等待链初始化为空。

使用信号量的步骤，我已经给你列好了。

第一步，获取信号量。

1. 首先对用于保护信号量自身的自旋锁 `sem_lock` 进行加锁。
2. 对信号值 `sem_count` 执行“减 1”操作，并检查其值是否小于 0。
3. 上步中检查 `sem_count` 如果小于 0，就让进程进入等待状态并且将其挂入 `sem_waitlst` 中，然后调度其它进程运行。否则表示获取信号量成功。当然最后别忘了对自旋锁 `sem_lock` 进行解锁。

第二步，代码执行流开始执行相关操作，例如读取键盘缓冲区。

第三步，释放信号量。

1. 首先对用于保护信号量自身的自旋锁 `sem_lock` 进行加锁。
2. 对信号值 `sem_count` 执行“加 1”操作，并检查其值是否大于 0。
3. 上步中检查 `sem_count` 值如果大于 0，就执行唤醒 `sem_waitlst` 中进程的操作，并且需要调度进程时就执行进程调度操作，不管 `sem_count` 是否大于 0（通常会大于 0）都标记信号量释放成功。当然最后别忘了对自旋锁 `sem_lock` 进行解锁。

这里我给你额外分享一个小技巧，**写代码之前我们常常需要先想清楚算法步骤，建议你像我这样分条列出，因为串联很容易含糊其辞，不利于后面顺畅编码。**

好，下面我们来看看实现上述这些功能的代码，按照理论书籍上说，信号量有两个操作：`down`，`up`，代码如下。

 复制代码

```
1 //获取信号量
2 void krlsem_down(sem_t* sem)
3 {
4     cpuflg_t cpufg;
5     start_step:
6     krlspinlock_cli(&sem->sem_lock,&cpufg);
7     if(sem->sem_count<1)
8     { //如果信号量值小于1,则让代码执行流（线程）睡眠
9         krlwlst_wait(&sem->sem_waitlst);
10        krlspinunlock_sti(&sem->sem_lock,&cpufg);
11        krlschedul(); //切换代码执行流，下次恢复执行时依然从下一行开始执行，所以要goto开始
```



```
12         goto start_step;
13     }
14     sem->sem_count--; //信号量值减1,表示成功获取信号量
15     krlspinunlock_sti(&sem->sem_lock,&cpufg);
16     return;
17 }
18 //释放信号量
19 void krlsem_up(sem_t* sem)
20 {
21     cpuflg_t cpufg;
22     krlspinlock_cli(&sem->sem_lock,&cpufg);
23     sem->sem_count++; //释放信号量
24     if(sem->sem_count<1)
25     { //如果小于1,则说数据结构出错了,挂起系统
26         krlspinunlock_sti(&sem->sem_lock,&cpufg);
27         hal_sysdie("sem up err");
28     }
29     //唤醒该信号量上所有等待的代码执行流(线程)
30     krlwlst_allup(&sem->sem_waitlst);
31     krlspinunlock_sti(&sem->sem_lock,&cpufg);
32     krlsched_set_schedflgs();
33     return;
34 }
```

上述代码中的 `krlspinlock_cli` , `krlspinunlock_sti` 两个函数, 只是对前面自旋锁函数的一个封装, `krlschedul`、`krlwlst_wait`、`krlwlst_allup`、`krlsched_set_schedflgs` 这几个函数会在进程相关课程进行探讨。

重点回顾

又到了这节课结束的时候, 我们回顾一下今天都讲了什么。我把这节课的内容为你梳理一下, 要点如下。

1. 原子变量, 在只有**单个变量全局数据**的情况下, 这种变量非常实用, 如全局计数器、状态标志变量等。我们利用了 CPU 的原子指令实现了一组操作原子变量的函数。

2. 中断的控制。当要操作的数据很多的情况下, 用原子变量就不适合了。但是我们发现在单核心的 CPU, 同一时刻只有一个代码执行流, 除了响应中断导致代码执行流切换, 不会有其它条件会干扰全局数据的操作, 所以我们只要在操作全局数据时关闭或者开启中断就行了, 为此我们开发了控制中断的函数。

3. 自旋锁。由于多核心的 CPU 出现，控制中断已经失效了，因为**系统中同时有多个代码执行流**，为了解决这个问题，我们开发了自旋锁，自旋锁要么一下子获取锁，要么循环等待最终获取锁。

4. 信号量。如果长时间等待后才能获取数据，在这样的情况下，前面中断控制和自旋锁都不能很好地解决，于是我们开发了信号量。信号量由一套数据结构和函数组成，它能使获取数据的代码执行流进入睡眠，然后在相关条件满足时被唤醒，这样就能让 CPU 能有时间处理其它任务。所以信号量同时解决了三个问题：**等待、互斥、唤醒**。

思考题

请用代码展示一下自旋锁或者信号量，可能的使用形式是什么样的？

期待你在留言区的分享，也欢迎你把这节课的内容分享给身边的朋友，跟他一起学习交流。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

👍 赞 20 💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | Cache与内存：程序放在哪儿？

下一篇 09 | 瞧一瞧Linux：Linux的自旋锁和信号量如何实现？

更多学习推荐

极客时间 | 训练营

大厂面试必考 100 题

2021 最新版 | 算法篇

限量免费领取  仅限前 99 名



精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。