

# 11. 线程同步与互斥

## 本节重点：

- 深刻理解线程互斥的原理和操作
- 深刻理解线程同步
- 掌握生产消费模型
- 设计日志和线程池
- 理解线程安全和可重入，掌握锁相关概念

## 1. 线程互斥

### 1-1 进程线程间的互斥相关背景概念

- 临界资源：多线程执行流共享的资源就叫做临界资源
- 临界区：每个线程内部，访问临界资源的代码，就叫做临界区
- 互斥：任何时刻，互斥保证有且只有一个执行流进入临界区，访问临界资源，通常对临界资源起保护作用
- 原子性（后面讨论如何实现）：不会被任何调度机制打断的操作，该操作只有两态，要么完成，要么未完成

### 1-2 互斥量mutex

- 大部分情况，线程使用的数据都是局部变量，变量的地址空间在线程栈空间内，这种情况，变量归属单个线程，其他线程无法获得这种变量。
- 但有时候，很多变量都需要在线程间共享，这样的变量称为共享变量，可以通过数据的共享，完成线程之间的交互。
- 多个线程并发的操作共享变量，会带来一些问题。

```
1 // 操作共享变量会有问题的售票系统代码
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <pthread.h>
7
8 int ticket = 100;
```

```

9
10 void *route(void *arg)
11 {
12     char *id = (char*)arg;
13     while ( 1 ) {
14         if ( ticket > 0 ) {
15             usleep(1000);
16             printf("%s sells ticket:%d\n", id, ticket);
17             ticket--;
18         } else {
19             break;
20         }
21     }
22 }
23
24 int main( void )
25 {
26     pthread_t t1, t2, t3, t4;
27
28     pthread_create(&t1, NULL, route, "thread 1");
29     pthread_create(&t2, NULL, route, "thread 2");
30     pthread_create(&t3, NULL, route, "thread 3");
31     pthread_create(&t4, NULL, route, "thread 4");
32
33     pthread_join(t1, NULL);
34     pthread_join(t2, NULL);
35     pthread_join(t3, NULL);
36     pthread_join(t4, NULL);
37 }
38
39 一次执行结果:
40     thread 4 sells ticket:100
41     ...
42     thread 4 sells ticket:1
43     thread 2 sells ticket:0
44     thread 1 sells ticket:-1
45     thread 3 sells ticket:-2

```

为什么可能无法获得争取结果？

- `if` 语句判断条件为真以后，代码可以并发的切换到其他线程
- `usleep` 这个模拟漫长业务的过程，在这个漫长的业务过程中，可能有很多个线程会进入该代码段
- `--ticket` 操作本身就不是一个原子操作

```
1 取出ticket--部分的汇编代码
2 objdump -d a.out > test.objdump
3 152 40064b: 8b 05 e3 04 20 00      mov     0x2004e3(%rip),%eax      #
   600b34 <ticket>
4 153 400651: 83 e8 01              sub     $0x1,%eax
5 154 400654: 89 05 da 04 20 00      mov     %eax,0x2004da(%rip)      #
   600b34 <ticket>
```

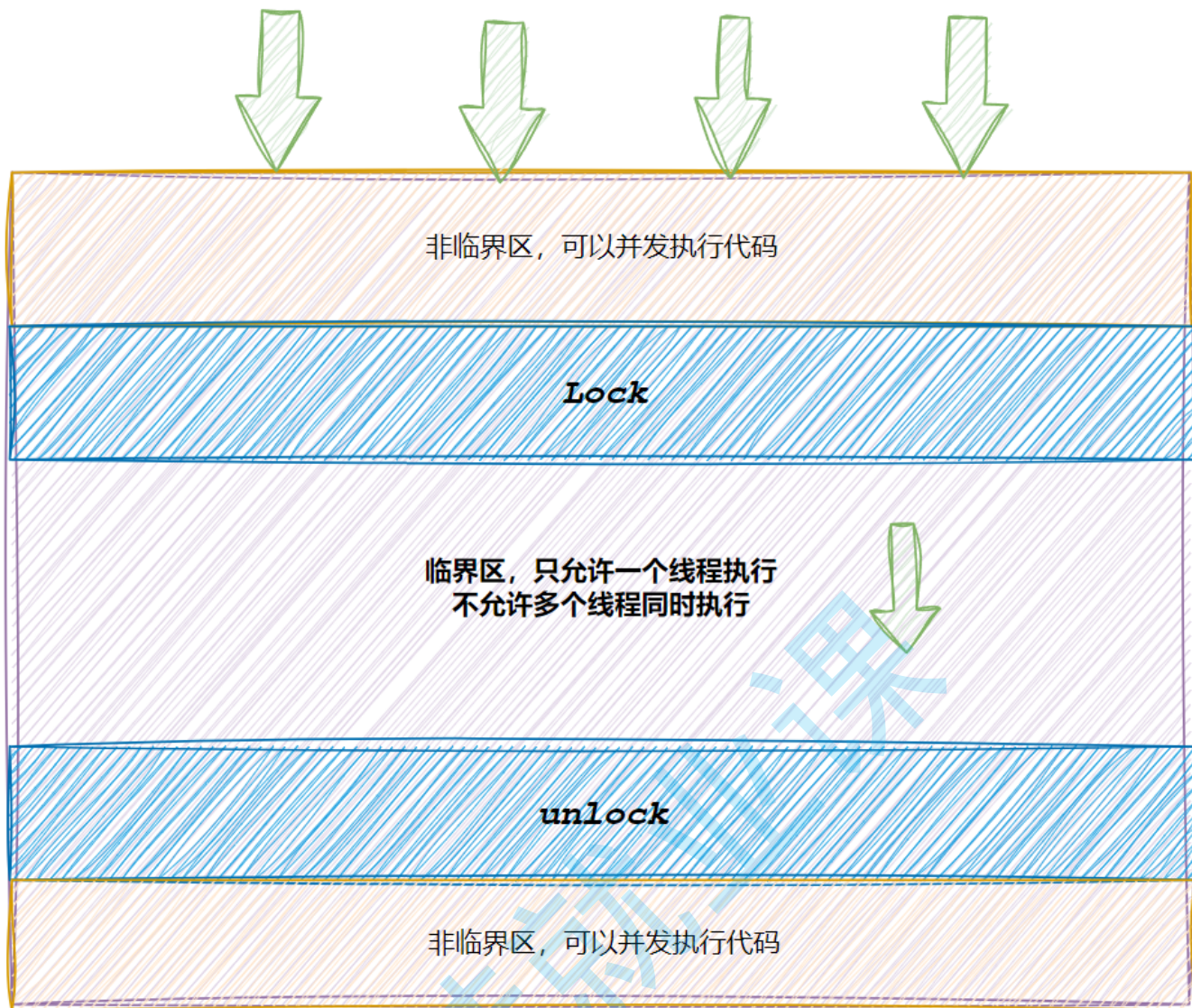
-- 操作并不是原子操作，而是对应三条汇编指令：

- `load`：将共享变量ticket从内存加载到寄存器中
- `update`：更新寄存器里面的值，执行-1操作
- `store`：将新值，从寄存器写回共享变量ticket的内存地址

要解决以上问题，需要做到三点：

- 代码必须要有互斥行为：当代码进入临界区执行时，不允许其他线程进入该临界区。
- 如果多个线程同时要求执行临界区的代码，并且临界区没有线程在执行，那么只能允许一个线程进入该临界区。
- 如果线程不在临界区中执行，那么该线程不能阻止其他线程进入临界区。

要做到这三点，本质上就是需要一把锁。Linux上提供的这把锁叫互斥量。



## 互斥量的接口

### 初始化互斥量

初始化互斥量有两种方法:

- 方法1, 静态分配:

```
1 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```

- 方法2, 动态分配:

```
1 int pthread_mutex_init(pthread_mutex_t *restrict mutex, const  
  pthread_mutexattr_t *restrict attr);
```

2 参数:

3 mutex: 要初始化的互斥量

4 attr: NULL

## 销毁互斥量

销毁互斥量需要注意：

- 使用 `PTHREAD_MUTEX_INITIALIZER` 初始化的互斥量不需要销毁
- 不要销毁一个已经加锁的互斥量
- 已经销毁的互斥量，要确保后面不会有线程再尝试加锁

```
1 int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

## 互斥量加锁和解锁

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex);
2 int pthread_mutex_unlock(pthread_mutex_t *mutex);
3 返回值:成功返回0,失败返回错误号
```

调用 `pthread_lock` 时，可能会遇到以下情况:

- 互斥量处于未锁状态，该函数会将互斥量锁定，同时返回成功
- 发起函数调用时，其他线程已经锁定互斥量，或者存在其他线程同时申请互斥量，但没有竞争到互斥量，那么`pthread_lock`调用会陷入阻塞(执行流被挂起)，等待互斥量解锁。

改进上面的售票系统:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <pthread.h>
6 #include <sched.h>
7
8 int ticket = 100;
9 pthread_mutex_t mutex;
10
11 void *route(void *arg)
12 {
13     char *id = (char*)arg;
14     while ( 1 ) {
15         pthread_mutex_lock(&mutex);
16         if ( ticket > 0 ) {
17             usleep(1000);
18             printf("%s sells ticket:%d\n", id, ticket);
```

```

19         ticket--;
20         pthread_mutex_unlock(&mutex);
21         // sched_yield(); 放弃CPU
22     } else {
23         pthread_mutex_unlock(&mutex);
24         break;
25     }
26 }
27 }
28
29 int main( void )
30 {
31     pthread_t t1, t2, t3, t4;
32
33     pthread_mutex_init(&mutex, NULL);
34
35     pthread_create(&t1, NULL, route, "thread 1");
36     pthread_create(&t2, NULL, route, "thread 2");
37     pthread_create(&t3, NULL, route, "thread 3");
38     pthread_create(&t4, NULL, route, "thread 4");
39
40     pthread_join(t1, NULL);
41     pthread_join(t2, NULL);
42     pthread_join(t3, NULL);
43     pthread_join(t4, NULL);
44     pthread_mutex_destroy(&mutex);
45 }

```

### 1-3 互斥量实现原理探究

- 经过上面的例子，大家已经意识到单纯的 `i++` 或者 `++i` 都不是原子的，有可能会有数据一致性问题
- 为了实现互斥锁操作,大多数体系结构都提供了swap或exchange指令,该指令的作用是把寄存器和内存单元的数据相交换,由于只有一条指令,保证了原子性,即使是多处理器平台,访问内存的总线周期也有先后,一个处理器上的交换指令执行时另一个处理器的交换指令只能等待总线周期。现在我们把lock和unlock的伪代码改一下



```

lock:
    movb $0, %al
    xchgb %al, mutex
    if(al寄存器的内容 > 0){
        return 0;
    } else
        挂起等待;
    goto lock;

unlock:
    movb $1, mutex
    唤醒等待Mutex的线程;
    return 0;

```

## 1-4 互斥量的封装

Lock.hpp

```

1  #pragma once
2
3  #include <iostream>
4  #include <string>
5  #include <pthread.h>
6
7  namespace LockModule
8  {
9      // 对锁进行封装，可以独立使用
10     class Mutex
11     {
12     public:
13         // 删除不要的拷贝和赋值
14         Mutex(const Mutex &) = delete;
15         const Mutex &operator =(const Mutex &) = delete;
16         Mutex()
17         {
18             int n = pthread_mutex_init(&_mutex, nullptr);
19             (void)n;
20         }
21         void Lock()
22         {
23             int n = pthread_mutex_lock(&_mutex);
24             (void)n;
25         }
26         void Unlock()
27         {

```

```

28         int n = pthread_mutex_unlock(&_mutex);
29         (void)n;
30     }
31     pthread_mutex_t *GetMutexOriginal() // 获取原始指针
32     {
33         return &_mutex;
34     }
35     ~Mutex()
36     {
37         int n = pthread_mutex_destroy(&_mutex);
38         (void)n;
39     }
40 private:
41     pthread_mutex_t _mutex;
42 };
43
44 // 采用RAII风格, 进行锁管理
45 class LockGuard
46 {
47 public:
48     LockGuard(Mutex &mutex):_mutex(mutex)
49     {
50         _mutex.Lock();
51     }
52     ~LockGuard()
53     {
54         _mutex.Unlock();
55     }
56 private:
57     Mutex &_mutex;
58 };
59
60 }

```

```

1 // 抢票的代码就可以更新成为
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <pthread.h>
7 #include "Lock.hpp"
8
9 using namespace LockModule;
10
11 int ticket = 1000;


```



```

12 Mutex mutex;
13
14 void *route(void *arg)
15 {
16     char *id = (char *)arg;
17     while (1)
18     {
19         LockGuard lockguard(mutex); // 使用RAII风格的锁
20         if (ticket > 0)
21         {
22             usleep(1000);
23             printf("%s sells ticket:%d\n", id, ticket);
24             ticket--;
25         }
26         else
27         {
28             break;
29         }
30     }
31     return nullptr;
32 }
33
34 int main(void)
35 {
36     pthread_t t1, t2, t3, t4;
37
38     pthread_create(&t1, NULL, route, (void*)"thread 1");
39     pthread_create(&t2, NULL, route, (void*)"thread 2");
40     pthread_create(&t3, NULL, route, (void*)"thread 3");
41     pthread_create(&t4, NULL, route, (void*)"thread 4");
42
43     pthread_join(t1, NULL);
44     pthread_join(t2, NULL);
45     pthread_join(t3, NULL);
46     pthread_join(t4, NULL);
47 }

```

 RAII风格的互斥锁，C++11也有，比如：

```
std::mutex mtx;
```

```
std::lock_guard<std::mutex> guard(mtx);
```

此处我们仅做封装，方便后续使用，详情见C++课程

另外，如果课堂有时间，也可以把我们封装的线程加入进来。

## 2. 线程同步

### 2-1 条件变量

- 当一个线程互斥地访问某个变量时，它可能发现在其它线程改变状态之前，它什么也做不了。
- 例如一个线程访问队列时，发现队列为空，它只能等待，直到其它线程将一个节点添加到队列中。这种情况就需要用到条件变量。

### 2-2 同步概念与竞态条件

- 同步：在保证数据安全的前提下，让线程能够按照某种特定的顺序访问临界资源，从而有效避免饥饿问题，叫做同步
- 竞态条件：因为时序问题，而导致程序异常，我们称之为竞态条件。在线程场景下，这种问题也不难理解

### 2-3 条件变量函数

初始化

```
1 int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t
  *restrict attr);
2 参数：
3     cond: 要初始化的条件变量
4     attr: NULL
```

销毁

```
1 int pthread_cond_destroy(pthread_cond_t *cond)
```

等待条件满足

```
1 int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict
  mutex);
2     参数：
3         cond: 要在这个条件变量上等待
4         mutex: 互斥量，后面详细解释
```

唤醒等待

```
1  int pthread_cond_broadcast(pthread_cond_t *cond);
2  int pthread_cond_signal(pthread_cond_t *cond);
```

简单案例：

- 我们先使用PTHREAD\_COND/MUTEX\_INITIALIZER进行测试，对其他细节暂不追究
- 然后将接口更改成为使用 `pthread_cond_init/pthread_cond_destroy` 的方式，方便后续进行封装

```
1  #include <iostream>
2  #include <string.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
7  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8
9  void *active( void *arg )
10 {
11     std::string name = static_cast<const char*>(arg);
12     while (true){
13         pthread_mutex_lock(&mutex);
14         pthread_cond_wait(&cond, &mutex);
15         std::cout << name << " 活动..." << std::endl;
16         pthread_mutex_unlock(&mutex);
17     }
18 }
19
20 int main( void )
21 {
22     pthread_t t1, t2;
23
24     pthread_create(&t1, NULL, active, (void*)"thread-1");
25     pthread_create(&t2, NULL, active, (void*)"thread-2");
26
27     sleep(3); // 可有可无，这里确保两个线程已经在运行
28     while(true)
29     {
30         // 对比测试
31         // pthread_cond_signal(&cond); // 唤醒一个线程
32         pthread_cond_broadcast(&cond); // 唤醒所有线程
33         sleep(1);
34     }
35 }
```

```
36     pthread_join(t1, NULL);
37     pthread_join(t2, NULL);
38 }
```

```
1 $ ./cond
2 thread-1 活动...
3 thread-2 活动...
4 thread-1 活动...
5 thread-1 活动...
6 thread-2 活动...
```

## 2-4 生产者消费者模型

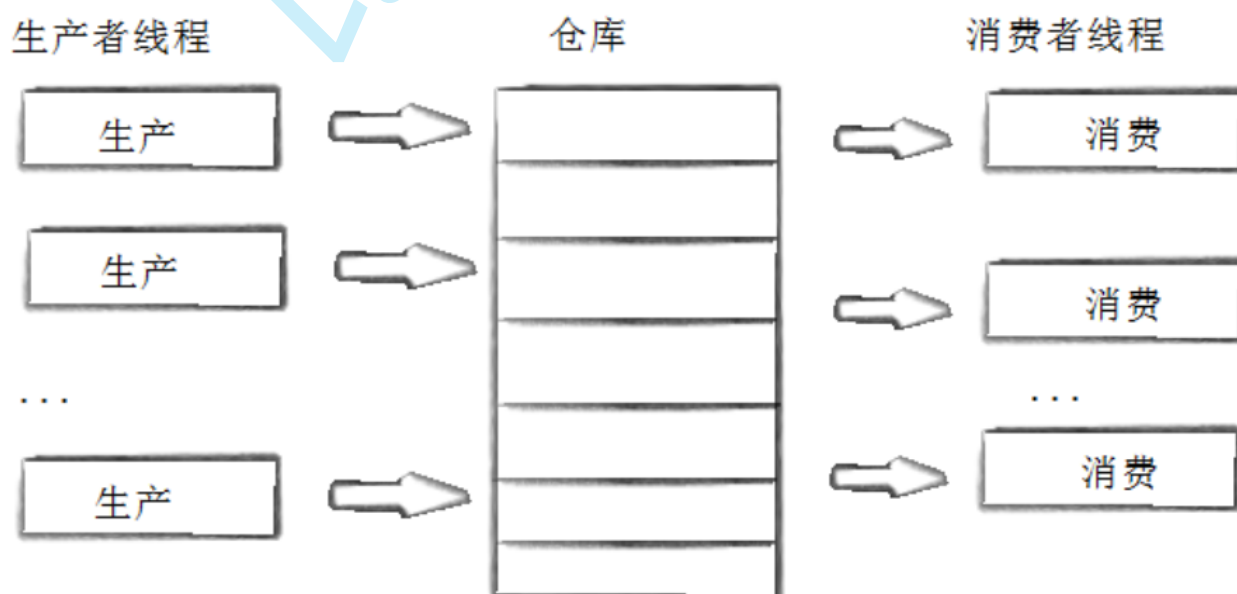
- 321原则(便于记忆)

### 2-4-1 为何要使用生产者消费者模型

生产者消费者模式就是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费者解耦的。

### 2-4-2 生产者消费者模型优点

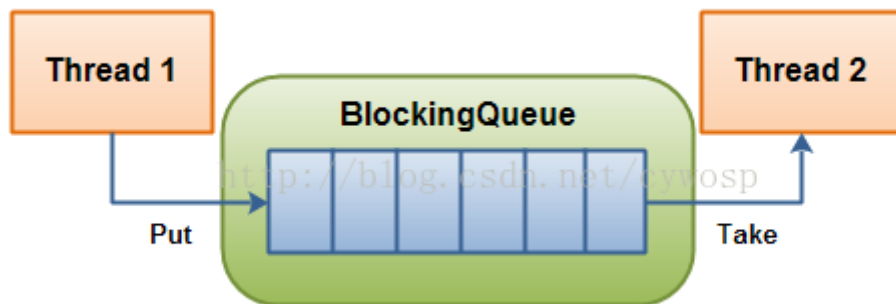
- 解耦
- 支持并发
- 支持忙闲不均



## 2-5 基于BlockingQueue的生产者消费者模型

### 2-5-1 BlockingQueue

在多线程编程中阻塞队列(Blocking Queue)是一种常用于实现生产者和消费者模型的数据结构。其与普通的队列区别在于，当队列为空时，从队列获取元素的操作将会被阻塞，直到队列中被放入了元素；当队列满时，往队列里存放元素的操作也会被阻塞，直到有元素被从队列中取出(以上的操作都是基于不同的线程来说的，线程在对阻塞队列进程操作时会被阻塞)



### 2-5-2 C++ queue模拟阻塞队列的生产消费模型

代码：

- 为了便于同学们理解，我们以单生产者，单消费者，来进行讲解。
- 刚开始写，我们采用原始接口。
- 我们先写单生产，单消费。然后改成多生产，多消费（这里代码其实不变）。

#### BlockQueue.hpp

```
1  #ifndef __BLOCK_QUEUE_HPP__
2  #define __BLOCK_QUEUE_HPP__
3
4  #include <iostream>
5  #include <string>
6  #include <queue>
7  #include <pthread.h>
8
9  template <typename T>
10 class BlockQueue
11 {
12 private:
13     bool IsFull()
14     {
15         return _block_queue.size() == _cap;
16     }
17     bool IsEmpty()
18     {
19         return _block_queue.empty();
20     }
```

```


21 public:
22     BlockQueue(int cap) : _cap(cap)
23     {
24         _producer_wait_num = 0;
25         _consumer_wait_num = 0;
26         pthread_mutex_init(&_mutex, nullptr);
27         pthread_cond_init(&_product_cond, nullptr);
28         pthread_cond_init(&_consum_cond, nullptr);
29     }
30     void Enqueue(T &in) // 生产者用的接口
31     {
32         pthread_mutex_lock(&_mutex);
33         while(IsFull()) // 保证代码的健壮性
34         {
35             // 生产线程去等待,是在临界区中休眠的! 你现在还持有锁呢!!!
36             // 1. pthread_cond_wait调用是: a. 让调用线程等待 b. 自动释放曾经持有的
37             // _mutex锁 c. 当条件满足, 线程唤醒, pthread_cond_wait要求线性
38             // 必须重新竞争_mutex锁, 竞争成功, 方可返回!!!
39             // 之前: 安全
40             _producer_wait_num++;
41             pthread_cond_wait(&_product_cond, &_mutex); // 只要等待, 必定会有唤
42             // 醒, 唤醒的时候, 就要继续从这个位置向下运行!!
43             _producer_wait_num--;
44             // 之后: 安全
45         }
46         // 进行生产
47         // _block_queue.push(std::move(in));
48         // std::cout << in << std::endl;
49         _block_queue.push(in);
50         // 通知消费者来消费
51         if(_consumer_wait_num > 0)
52             pthread_cond_signal(&_consum_cond); // pthread_cond_broadcast
53         pthread_mutex_unlock(&_mutex);
54     }
55     void Pop(T *out) // 消费者用的接口 --- 5个消费者
56     {
57         pthread_mutex_lock(&_mutex);
58         while(IsEmpty()) // 保证代码的健壮性
59         {
60             // 消费线程去等待,是在临界区中休眠的! 你现在还持有锁呢!!!
61             // 1. pthread_cond_wait调用是: a. 让调用进程等待 b. 自动释放曾经持有的
62             // _mutex锁
63             _consumer_wait_num++;
64             pthread_cond_wait(&_consum_cond, &_mutex); // 伪唤醒
65             _consumer_wait_num--;
66         }
67     }

```

```

65         // 进行消费
66         *out = _block_queue.front();
67         _block_queue.pop();
68         // 通知生产者来生产
69         if(_producer_wait_num > 0)
70             pthread_cond_signal(&_product_cond);
71         pthread_mutex_unlock(&_mutex);
72         // pthread_cond_signal(&_product_cond);
73     }
74     ~BlockQueue()
75     {
76         pthread_mutex_destroy(&_mutex);
77         pthread_cond_destroy(&_product_cond);
78         pthread_cond_destroy(&_consum_cond);
79     }
80
81 private:
82     std::queue<T> _block_queue;    // 阻塞队列，是被整体使用的!!!
83     int _cap;                    // 总上限
84     pthread_mutex_t _mutex;       // 保护_block_queue的锁
85     pthread_cond_t _product_cond; // 专门给生产者提供的条件变量
86     pthread_cond_t _consum_cond;  // 专门给消费者提供的条件变量
87
88     int _producer_wait_num;
89     int _consumer_wait_num;
90 };

```

 注意：这里采用模版，是想告诉我们，队列中不仅仅可以防止内置类型，比如int, 对象也可以作为任务来参与生产消费的过程哦。

下面附上一张代码，方便课堂使用

```

1  #pragma once
2
3  #include <iostream>
4  #include <string>
5  #include <functional>
6
7  // 任务类型1
8  // class Task
9  // {
10 // public:
11 //     Task() {}
12 //     Task(int a, int b) : _a(a), _b(b), _result(0)

```



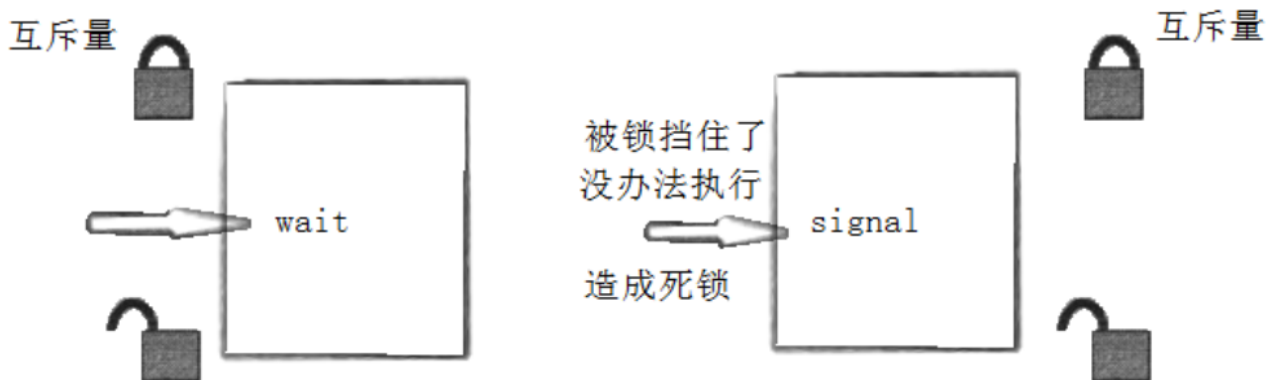
```

13 // {
14 // }
15 // void Excute()
16 // {
17 //     _result = _a + _b;
18 // }
19 // std::string ResultToString()
20 // {
21 //     return std::to_string(_a) + "+" + std::to_string(_b) + "=" +
        std::to_string(_result);
22 // }
23 // std::string DebugToString()
24 // {
25 //     return std::to_string(_a) + "+" + std::to_string(_b) + "=?";
26 // }
27
28 // private:
29 //     int _a;
30 //     int _b;
31 //     int _result;
32 // };
33
34 // 任务类型2
35 using Task = std::function<void()>;

```

## 2-6 为什么 pthread\_cond\_wait 需要互斥量？

- 条件等待是线程间同步的一种手段，如果只有一个线程，条件不满足，一直等下去都不会满足，所以必须要有一个线程通过某些操作，改变共享变量，使原先不满足的条件变得满足，并且友好的通知等待在条件变量上的线程。
- 条件不会无缘无故的突然变得满足了，必然会牵扯到共享数据的变化。所以一定要用互斥锁来保护。没有互斥锁就无法安全的获取和修改共享数据。



- 按照上面的说法，我们设计出如下的代码：先上锁，发现条件不满足，解锁，然后等待在条件变量上不就行了，如下代码：

```

1 // 错误的设计
2 pthread_mutex_lock(&mutex);
3 while (condition_is_false) {
4     pthread_mutex_unlock(&mutex);
5     //解锁之后，等待之前，条件可能已经满足，信号已经发出，但是该信号可能被错过
6     pthread_cond_wait(&cond);
7     pthread_mutex_lock(&mutex);
8 }
9 pthread_mutex_unlock(&mutex);

```

- 由于解锁和等待不是原子操作。调用解锁之后，`pthread_cond_wait` 之前，如果已经有其他线程获取到互斥量，摒弃条件满足，发送了信号，那么 `pthread_cond_wait` 将错过这个信号，可能会导致线程永远阻塞在这个 `pthread_cond_wait`。所以解锁和等待必须是一个原子操作。
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);` 进入该函数后，会去看条件量等于0不？等于，就把互斥量变成1，直到 `cond_wait` 返回，把条件量改成1，把互斥量恢复成原样。

## 2-7 条件变量使用规范

- 等待条件代码

```

1 pthread_mutex_lock(&mutex);
2 while (条件为假)
3     pthread_cond_wait(cond, mutex);
4 修改条件
5 pthread_mutex_unlock(&mutex);

```

- 给条件发送信号代码

```

1 pthread_mutex_lock(&mutex);
2 设置条件为真
3 pthread_cond_signal(cond);
4 pthread_mutex_unlock(&mutex);

```

## 2-8 条件变量的封装

- 基于上面的基本认识，我们已经知道条件变量如何使用，虽然细节需要后面再来进行解释，但这里可以做一下基本的封装，以备后用。

`Cond.hpp`

```

1  #pragma once
2  #include <iostream>
3  #include <string>
4  #include <pthread.h>
5  #include "Lock.hpp"
6
7  namespace CondModule
8  {
9      using namespace LockModule;
10
11     class Cond
12     {
13     public:
14         Cond()
15         {
16             int n = pthread_cond_init(&_cond, nullptr);
17             (void)n; // 酌情加日志, 加判断
18         }
19         void Wait(Mutex &mutex)
20         {
21             int n = pthread_cond_wait(&_cond, mutex.GetMutexOriginal());
22             (void)n;
23         }
24         void Notify()
25         {
26             int n = pthread_cond_signal(&_cond);
27             (void)n;
28         }
29         void NotifyAll()
30         {
31             int n = pthread_cond_broadcast(&_cond);
32             (void)n;
33         }
34         ~Cond()
35         {
36             int n = pthread_cond_destroy(&_cond);
37             (void)n; // 酌情加日志, 加判断
38         }
39     private:
40         pthread_cond_t _cond;
41     };
42 }

```

为了让条件变量更具有通用性，建议封装的时候，不要在Cond类内部引用对应的封装互斥量，要不然后面组合的时候，会因为代码耦合的问题难以初始化，因为一般而言Mutex和Cond基本是一起创建的。

## 2-2 POSIX信号量

POSIX信号量和SystemV信号量作用相同，都是用于同步操作，达到无冲突的访问共享资源目的。但POSIX可以用于线程间同步。

### 初始化信号量

```
1 #include <semaphore.h>
2 int sem_init(sem_t *sem, int pshared, unsigned int value);
3 参数：
4     pshared: 0表示线程间共享，非零表示进程间共享
5     value: 信号量初始值
```

### 销毁信号量

```
1 int sem_destroy(sem_t *sem);
```

### 等待信号量

```
1 功能：等待信号量，会将信号量的值减1
2 int sem_wait(sem_t *sem); //P()
```

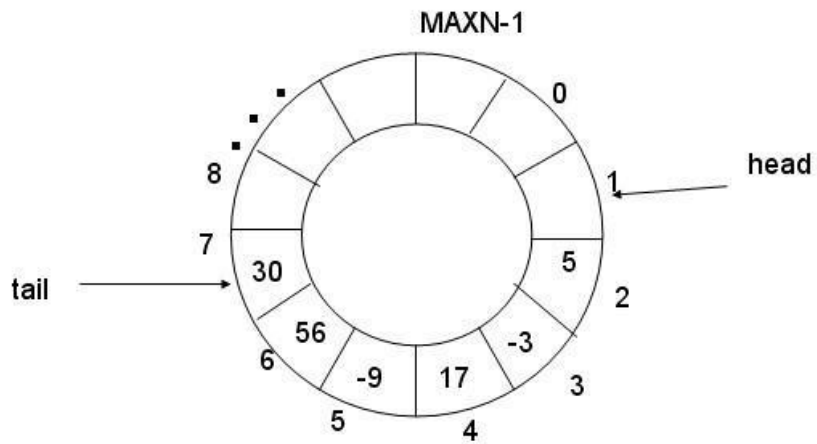
### 发布信号量

```
1 功能：发布信号量，表示资源使用完毕，可以归还资源了。将信号量值加1。
2 int sem_post(sem_t *sem); //V()
```

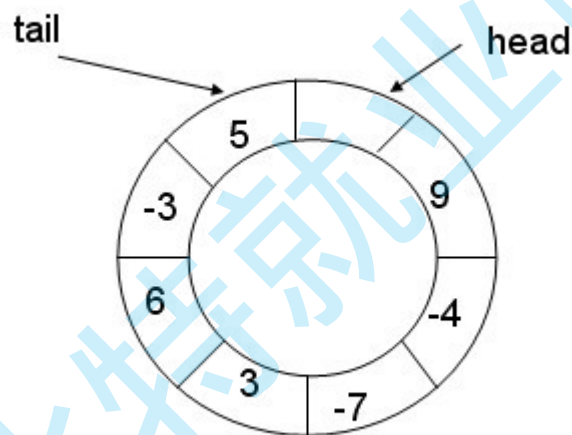
上一节生产者-消费者的例子是基于queue的,其空间可以动态分配,现在基于固定大小的环形队列重写这个程序（POSIX信号量）：

### 2-2-1 基于环形队列的生产消费模型

- 环形队列采用数组模拟，用模运算来模拟环状特性



- 环形结构起始状态和结束状态都是一样的，不好判断为空或者为满，所以可以通过加计数器或者标记位来判断满或者空。另外也可以预留一个空的位置，作为满的状态



- 但是我们现在有信号量这个计数器，就很简单的进行多线程间的同步过程。

```

1 #pragma once
2 #include <iostream>
3 #include <semaphore.h>
4
5 // 随手做一下封装
6 class Sem
7 {
8 public:
9     Sem(int n)
10     {
11         sem_init(&_sem, 0, n);
12     }

```

```

13     void P()
14     {
15         sem_wait(&_sem);
16     }
17     void V()
18     {
19         sem_post(&_sem);
20     }
21     ~Sem()
22     {
23         sem_destroy(&_sem);
24     }
25 private:
26     sem_t _sem;
27 };

```

### 注意：

- 这里我们还是忍住，先进行原始接口的使用
- 先单生产，单消费，然后改成多生产，多消费。
- 关于任务，cond处已经介绍，这里就不再重复了。

```

1  #pragma once
2
3  #include <iostream>
4  #include <string>
5  #include <vector>
6  #include <semaphore.h>
7  #include <pthread.h>
8
9  // 单生产，单消费
10 // 多生产，多消费
11 // "321":
12 // 3: 三种关系
13 // a: 生产和消费互斥和同步
14 // b: 生产者之间:
15 // c: 消费者之间:
16 // 解决方案: 加锁
17 // 1. 需要几把锁? 2把
18 // 2. 如何加锁?
19
20
21 template<typename T>

```

```

22 class RingQueue
23 {
24 private:
25     void Lock(pthread_mutex_t &mutex)
26     {
27         pthread_mutex_lock(&mutex);
28     }
29     void Unlock(pthread_mutex_t &mutex)
30     {
31         pthread_mutex_unlock(&mutex);
32     }
33 public:
34     RingQueue(int cap)
35     : _ring_queue(cap),
36       _cap(cap),
37       _room_sem(cap),
38       _data_sem(0),
39       _producer_step(0),
40       _consumer_step(0)
41     {
42         pthread_mutex_init(&_amp;producer_mutex, nullptr);
43         pthread_mutex_init(&_amp;consumer_mutex, nullptr);
44     }
45     void Enqueue(const T &in)
46     {
47         // 生产行为
48         _room_sem.P();
49         Lock(_amp;producer_mutex);
50         // 一定有空间!!!
51         _ring_queue[_amp;producer_step++] = in; // 生产
52         _amp;producer_step %= _cap;
53         Unlock(_amp;producer_mutex);
54         _data_sem.V();
55     }
56     void Pop(T *out)
57     {
58         // 消费行为
59         _data_sem.P();
60         Lock(_amp;consumer_mutex);
61         *out = _ring_queue[_amp;consumer_step++];
62         _amp;consumer_step %= _cap;
63         Unlock(_amp;consumer_mutex);
64         _room_sem.V();
65     }
66     ~RingQueue()
67     {
68         pthread_mutex_destroy(&_amp;producer_mutex);

```



```

69     pthread_mutex_destroy(&_consumer_mutex);
70 }
71 private:
72     // 1. 环形队列
73     std::vector<T> _ring_queue;
74     int _cap; // 环形队列的容量上限
75
76     // 2. 生产和消费的下标
77     int _producer_step;
78     int _consumer_step;
79
80     // 3. 定义信号量
81     Sem _room_sem; // 生产者关心
82     Sem _data_sem; // 消费者关心
83
84     // 4. 定义锁，维护多生产多消费之间的互斥关系
85     pthread_mutex_t _producer_mutex;
86     pthread_mutex_t _consumer_mutex;
87 };

```

## 3. 线程池

下面开始，我们结合我们之前所做的所有封装，进行一个线程池的设计。在写之前，我们要做如下准备

- 准备线程的封装
- 准备锁和条件变量的封装
- 引入日志，对线程进行封装

### 3-1 日志与策略模式

#### 什么是设计模式

IT行业这么火, 涌入的人很多. 俗话说林子大了啥鸟都有. 大佬和菜鸡们两极分化的越来越严重. 为了让菜鸡们不太拖大佬的后腿, 于是大佬们针对一些经典的常见的场景, 给定了一些对应的解决方案, 这个就是 **设计模式**

#### 日志认识

计算机中的日志是记录系统和软件运行中发生事件的文件，主要作用是监控运行状态、记录异常信息，帮助快速定位问题并支持程序员进行问题修复。它是系统维护、故障排查和安全管理的重要工具。

日志格式以下几个指标是必须得有的

- 时间戳

- 日志等级
- 日志内容

以下几个指标是可选的

- 文件名行号
- 进程，线程相关id信息等

日志有现成的解决方案，如：spdlog、glog、Boost.Log、Log4cxx等等，我们依旧采用自定义日志的方式。

这里我们采用设计模式-策略模式来进行日志的设计，具体策略模式介绍，详情看代码和课程。

我们想要的日志格式如下：

```
1 [可读性很好的时间] [日志等级] [进程pid] [打印对应日志的文件名][行号] - 消息内容，支持可变参数
2 [2024-08-04 12:27:03] [DEBUG] [202938] [main.cc] [16] - hello world
3 [2024-08-04 12:27:03] [DEBUG] [202938] [main.cc] [17] - hello world
4 [2024-08-04 12:27:03] [DEBUG] [202938] [main.cc] [18] - hello world
5 [2024-08-04 12:27:03] [DEBUG] [202938] [main.cc] [20] - hello world
6 [2024-08-04 12:27:03] [DEBUG] [202938] [main.cc] [21] - hello world
7 [2024-08-04 12:27:03] [WARNING] [202938] [main.cc] [23] - hello world
```

Log.hpp

```
1 #pragma once
2
3 #include <iostream>
4 #include <string>
5 #include <fstream>
6 #include <memory>
7 #include <ctime>
8 #include <sstream>
9 #include <filesystem> // C++17, 需要高版本编译器和-std=c++17
10 #include <unistd.h>
11 #include "Lock.hpp"
12
13 namespace LogModule
14 {
15     using namespace LockModule; // 使用我们自己封装的锁，也可以采用C++11的锁。
16
17     // 默认路径和日志名称
18     const std::string defaultpath = "./log/";
19     const std::string defaultname = "log.txt";
```

```

20
21 // 日志等级
22 enum class LogLevel
23 {
24     DEBUG,
25     INFO,
26     WARNING,
27     ERROR,
28     FATAL
29 };
30
31 // 日志转换为字符串
32 std::string LogLevelToString(LogLevel level)
33 {
34     switch (level)
35     {
36     case LogLevel::DEBUG:
37         return "DEBUG";
38     case LogLevel::INFO:
39         return "INFO";
40     case LogLevel::WARNING:
41         return "WARNING";
42     case LogLevel::ERROR:
43         return "ERROR";
44     case LogLevel::FATAL:
45         return "FATAL";
46     default:
47         return "UNKNOWN";
48     }
49 }
50
51 // 根据时间戳，获取可读性较强的时间信息
52 std::string GetCurrTime()
53 {
54     time_t tm = time(nullptr);
55     struct tm curr;
56     localtime_r(&tm, &curr);
57     // 这里如果不好看，可以考虑sprintf
58     // 方法 1
59     // std::stringstream ss;
60     // ss << curr.tm_year + 1900 << "-" << curr.tm_mon << "-" <<
curr.tm_mday << " "
61     // << curr.tm_hour << ":" << curr.tm_min << ":" << curr.tm_sec;
62     // return ss.str();
63     // 方法 2
64     char timebuffer[64];

```

```

65     snprintf(timebuffer, sizeof(timebuffer), "%4d-%02d-%02d
    %02d:%02d:%02d",
66         curr.tm_year + 1900,
67         curr.tm_mon,
68         curr.tm_mday,
69         curr.tm_hour,
70         curr.tm_min,
71         curr.tm_sec);
72     return timebuffer;
73 }
74
75 // 策略模式, 策略接口
76 class LogStrategy
77 {
78 public:
79     virtual ~LogStrategy() = default; // 策略的构造函数
80     virtual void SyncLog(const std::string &message) = 0; // 不同模式核心是刷
    新方式的不同
81 };
82
83 // 控制台日志策略, 就是日志只向显示器打印, 方便我们debug
84 class ConsoleLogStrategy : public LogStrategy
85 {
86 public:
87     void SyncLog(const std::string &message) override
88     {
89         LockGuard LockGuard(_mutex);
90         std::cerr << message << std::endl;
91     }
92     ~ConsoleLogStrategy()
93     {
94         // std::cout << "~ConsoleLogStrategy" << std::endl; // for debug
95     }
96
97 private:
98     Mutex _mutex; // 显示器也是临界资源, 保证输出线程安全
99 };
100
101 // 文件日志策略
102 class FileLogStrategy : public LogStrategy
103 {
104 public:
105     // 构造函数, 建立出来指定的目录结构和文件结构
106     FileLogStrategy(const std::string logpath = defaultpath, std::string
    logfile = defaultname)
107         : _logpath(logpath), _logfile(logfile)
108     {

```

```

109         LockGuard lockguard(_mutex);
110         if (std::filesystem::exists(_logpath))
111             return;
112         try
113         {
114             std::filesystem::create_directories(_logpath);
115         }
116         catch (const std::filesystem::filesystem_error &e)
117         {
118             std::cerr << e.what() << '\n';
119         }
120     }
121     // 将一条日志信息写入到文件中
122     void SyncLog(const std::string &message) override
123     {
124         LockGuard lockguard(_mutex);
125         std::string log = _logpath + _logfilename;
126         std::ofstream out(log.c_str(), std::ios::app); // 追加方式
127         if (!out.is_open())
128             return;
129         out << message << "\n";
130         out.close();
131     }
132     ~FileLogStrategy()
133     {
134         // std::cout << "~FileLogStrategy" << std::endl; // for debug
135     }
136
137     public:
138         std::string _logpath;
139         std::string _logfilename;
140         Mutex _mutex; // 保证输出线程安全，粗狂方式下，可以不用
141     };
142
143     // 具体的日志类
144     class Logger
145     {
146     public:
147         Logger()
148         {
149             // 默认使用显示器策略，如果用户二次指明了策略，会释放在申请，测试的时候注意析
150             UseConsoleStrategy();
151         }
152         ~Logger()
153         {
154

```

```

155     void UseConsoleStrategy()
156     {
157         _strategy = std::make_unique<ConsoleLogStrategy>();
158     }
159     void UseFileStrategy()
160     {
161         _strategy = std::make_unique<FileLogStrategy>();
162     }
163     // 内部类，实现RAII风格的日志格式化和刷新
164     // 这个LogMessage，表示一条完整的日志对象
165     class LogMessage
166     {
167     private:
168         LogLevel _type;           // 日志等级
169         std::string _curr_time;    // 日志时间
170         pid_t _pid;               // 写入日志的时间
171         std::string _filename;     // 对应的文件名
172         int _line;               // 对应的文件行号
173         Logger &_logger;          // 引用外部logger类，方便使用策略进行刷新
174         std::string _loginfo;     // 一条合并完成的，完整的日志信息
175
176     public:
177         // RAII风格，构造的时候构建好日志头部信息
178         LogMessage(LogLevel type, std::string &filename, int line, Logger
&logger)
179             : _type(type),
180               _curr_time(GetCurrTime()),
181               _pid(getpid()),
182               _filename(filename),
183               _line(line),
184               _logger(logger)
185         {
186             // stringstream不允许拷贝，所以这里就当做格式化功能使用
187             std::stringstream ssbuffer;
188             ssbuffer << "[" << _curr_time << "]" "
189                     << "[" << LogLevelToString(type) << "]" "
190                     << "[" << _pid << "]" "
191                     << "[" << _filename << "]" "
192                     << "[" << _line << "]" "
193                     << " - ";
194             _loginfo = ssbuffer.str();
195         }
196         // 重载 << 支持C++风格的日志输入，使用模版，表示支持任意类型
197         template <typename T>
198         LogMessage &operator<<(const T &info)
199         {
200             std::stringstream ssbuffer;

```

```

201         ssbuffer << info;
202         _loginfo += ssbuffer.str();
203         return *this; // 返回当前LogMessage对象，方便下次继续进行<<
204     }
205     // RAII风格，析构的时候进行日志持久化，采用指定的策略
206     ~LogMessage()
207     {
208         if (_logger._strategy)
209         {
210             _logger._strategy->SyncLog(_loginfo);
211         }
212         // std::cout << "~LogMessage" << std::endl;
213     }
214 };
215
216 // 故意拷贝，形成LogMessage临时对象，后续在被<<时，会被持续引用，
217 // 直到完成输入，才会自动析构临时LogMessage，至此也完成了日志的显示或者刷新
218 // 同时，形成的临时对象内包含独立日志数据
219 // 未来采用宏替换，进行文件名和代码行数的获取
220 LogMessage operator()(LogLevel type, std::string filename, int line)
221 {
222     return LogMessage(type, filename, line, *this);
223 }
224
225 private:
226     std::unique_ptr<LogStrategy> _strategy; // 写入日志的策略
227 };
228
229 // 定义全局的logger对象
230 Logger logger;
231
232 // 使用宏，可以进行代码插入，方便随时获取文件名和行号
233 #define LOG(type) logger(type, __FILE__, __LINE__)
234
235 // 提供选择使用何种日志策略的方法
236 #define ENABLE_CONSOLE_LOG_STRATEGY() logger.UseConsoleStrategy()
237 #define ENABLE_FILE_LOG_STRATEGY() logger.UseFileStrategy()
238 }

```

## 使用样例

```

1 #include <iostream>
2 #include "Log.hpp"
3
4 using namespace LogModule;

```



```

5
6 void fun()
7 {
8     int a = 10;
9     LOG(LogLevel::FATAL) << "hello world" << 1234 << ", 3.14" << 'c' << a;
10 }
11
12 int main()
13 {
14     // ENABLE_CONSOLE_LOG_STRATEGY();
15
16     LOG(LogLevel::DEBUG) << "hello world";
17     LOG(LogLevel::DEBUG) << "hello world";
18     LOG(LogLevel::DEBUG) << "hello world";
19     // ENABLE_FILE_LOG_STRATEGY();
20     LOG(LogLevel::DEBUG) << "hello world";
21     LOG(LogLevel::DEBUG) << "hello world";
22
23     LOG(LogLevel::WARNING) << "hello world";
24
25     fun();
26
27     return 0;
28 }

```

#### 注意：

- C++11也已经封装了锁比如：std::lock\_guard<std::mutex> lock(\_mutex);
- 这里我们直接用我们自己封装的锁

## 3-2 线程池设计

### 线程池：

一种线程使用模式。线程过多会带来调度开销，进而影响缓存局部性和整体性能。而线程池维护着多个线程，等待着监督管理者分配可并发执行的任务。这避免了在处理短时间任务时创建与销毁线程的代价。线程池不仅能够保证内核的充分利用，还能防止过分调度。可用线程数量应该取决于可用的并发处理器、处理器内核、内存、网络sockets等的数量。

### 线程池的应用场景：

- 需要大量的线程来完成任务，且完成任务的时间比较短。比如WEB服务器完成网页请求这样的任务，使用线程池技术是非常合适的。因为单个任务小，而任务数量巨大，你可以想象一个热门网站的点击次数。但对于长时间的任务，比如一个Telnet连接请求，线程池的优点就不明显了。因为Telnet会话时间比线程的创建时间大多了。

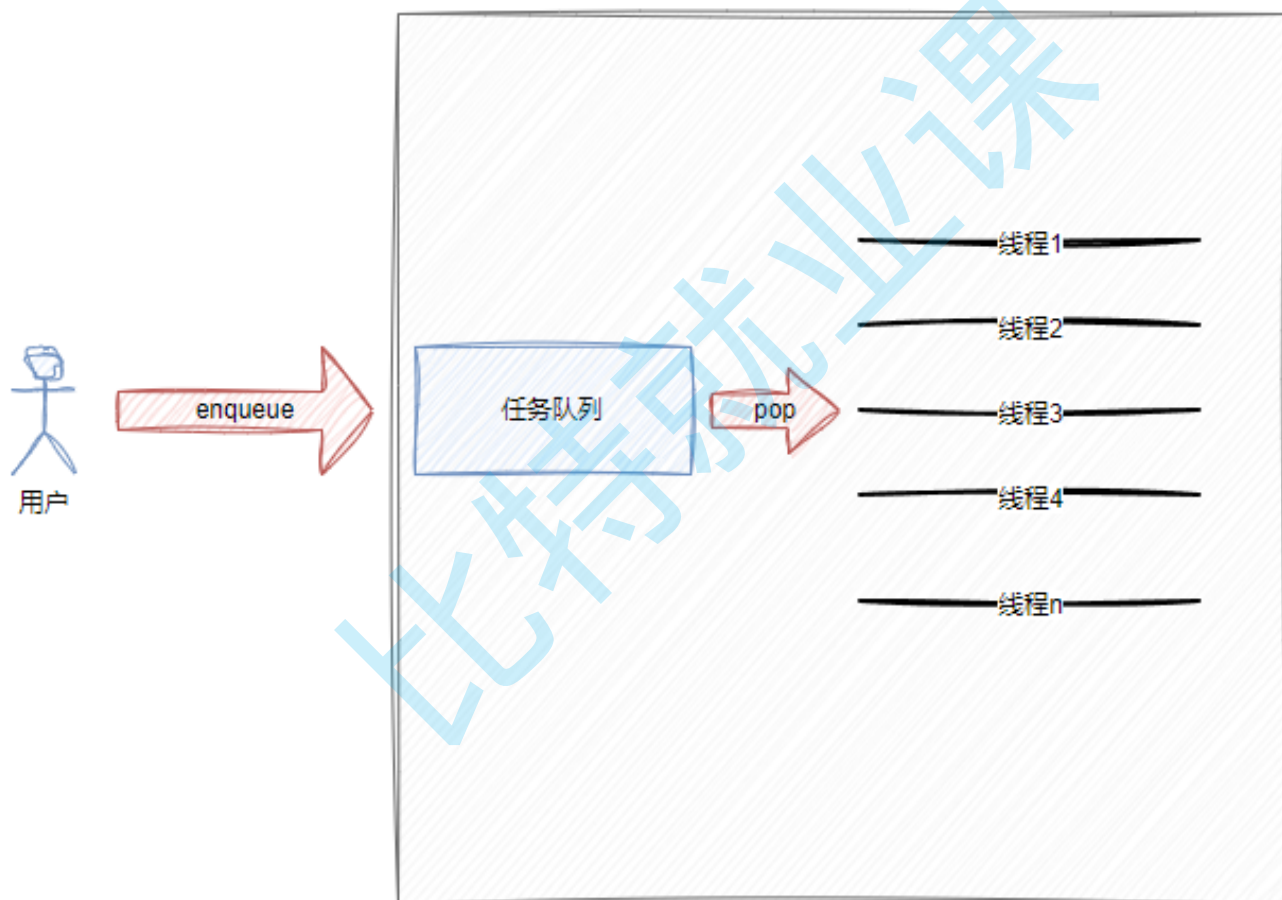
- 对性能要求苛刻的应用，比如要求服务器迅速响应客户请求。
- 接受突发性的大量请求，但不至于使服务器因此产生大量线程的应用。突发性大量客户请求，在没有线程池情况下，将产生大量线程，虽然理论上大部分操作系统线程数目最大值不是问题，短时间内产生大量线程可能使内存到达极限，出现错误。

## 线程池的种类

- a. 创建固定数量线程池，循环从任务队列中获取任务对象，获取到任务对象后，执行任务对象中的任务接口
- b. 浮动线程池，其他同上

此处，我们选择固定线程个数的线程池。

### 线程池



ThreadPool.hpp

```
1 #pragma once
2
3 #include <iostream>
4 #include <vector>
5 #include <queue>
6 #include <memory>
```

```

7 #include <pthread.h>
8 #include "Log.hpp"      // 引入自己的日志
9 #include "Thread.hpp"   // 引入自己的线程
10 #include "Lock.hpp"     // 引入自己的锁
11 #include "Cond.hpp"     // 引入自己的条件变量
12
13 using namespace ThreadModule;
14 using namespace CondModule;
15 using namespace LockModule;
16 using namespace LogModule;
17
18 const static int gdefaultthreadnum = 10;
19
20 // 日志
21 template <typename T>
22 class ThreadPool
23 {
24 private:
25     void HandlerTask() // 类的成员方法，也可以成为另一个类的回调方法，方便我们继续类级
        别的互相调用！
26     {
27         std::string name = GetThreadNameFromNptl();
28         LOG(LogLevel::INFO) << name << " is running...";
29         while (true)
30         {
31             // 1. 保证队列安全
32             _mutex.Lock();
33             // 2. 队列中不一定有数据
34             while (_task_queue.empty() && !_isrunning)
35             {
36                 _waitnum++;
37                 _cond.Wait(_mutex);
38                 _waitnum--;
39             }
40             // 2.1 如果线程池已经退出了 && 任务队列是空的
41             if (_task_queue.empty() && !_isrunning)
42             {
43                 _mutex.Unlock();
44                 break;
45             }
46             // 2.2 如果线程池不退出 && 任务队列不是空的
47             // 2.3 如果线程池已经退出 && 任务队列不是空的 --- 处理完所有的任务，然后在
        退出
48             // 3. 一定有任务，处理任务
49             T t = _task_queue.front();
50             _task_queue.pop();
51             _mutex.Unlock();

```

```

52         LOG(LogLevel::DEBUG) << name << " get a task";
53         // 4. 处理任务, 这个任务属于线程独占的任务
54         t();
55     }
56 }
57
58 public:
59     // 是要有的, 必须是私有的
60     ThreadPool(int threadnum = gdefaultthreadnum) : _threadnum(threadnum),
        _waitnum(0), _isrunning(false)
61     {
62         LOG(LogLevel::INFO) << "ThreadPool Construct()";
63     }
64     void InitThreadPool()
65     {
66         // 指向构建出所有的线程, 并不启动
67         for (int num = 0; num < _threadnum; num++)
68         {
69             _threads.emplace_back(std::bind(&ThreadPool::HandlerTask, this));
70             LOG(LogLevel::INFO) << "init thread " << _threads.back().Name() <<
" done";
71         }
72     }
73     void Start()
74     {
75         _isrunning = true;
76         for (auto &thread : _threads)
77         {
78             thread.Start();
79             LOG(LogLevel::INFO) << "start thread " << thread.Name() << "done";
80         }
81     }
82     void Stop()
83     {
84         _mutex.Lock();
85         _isrunning = false;
86         _cond.NotifyAll();
87         _mutex.Unlock();
88         LOG(LogLevel::DEBUG) << "线程池退出中...";
89     }
90     void Wait()
91     {
92         for (auto &thread : _threads)
93         {
94             thread.Join();
95             LOG(LogLevel::INFO) << thread.Name() << " 退出...";
96         }

```

```

97     }
98     bool Enqueue(const T &t)
99     {
100         bool ret = false;
101         _mutex.Lock();
102         if (_isrunning)
103         {
104             _task_queue.push(t);
105             if (_waitnum > 0)
106             {
107                 _cond.Notify();
108             }
109             LOG(LogLevel::DEBUG) << "任务入队列成功";
110             ret = true;
111         }
112         _mutex.Unlock();
113         return ret;
114     }
115     ~ThreadPool()
116     {}
117
118 private:
119     int _threadnum;
120     std::vector<Thread> _threads; // for fix, int temp
121     std::queue<T> _task_queue;
122     Mutex _mutex;
123     Cond _cond;
124
125     int _waitnum;
126     bool _isrunning;
127 };
128

```

```

1 g++ main.cc -std=c++17 -lpthread // 需要使用C++17

```

```

1 $ ./a.out
2 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [62] - ThreadPool
Construct()
3 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [70] - init thread
Thread-0 done
4 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [70] - init thread
Thread-1 done

```

```
5 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [70] - init thread
   Thread-2 done
6 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [70] - init thread
   Thread-3 done
7 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [70] - init thread
   Thread-4 done
8 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [79] - start thread
   Thread-0done
9 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [79] - start thread
   Thread-1done
10 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [28] - Thread-0 is
    running...
11 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [79] - start thread
    Thread-2done
12 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [79] - start thread
    Thread-3done
13 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [28] - Thread-3 is
    running...
14 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [28] - Thread-2 is
    running...
15 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [79] - start thread
    Thread-4done
16 [2024-08-04 15:09:29] [DEBUG] [206342] [ThreadPool.hpp] [109] - 任务入队列成功
17 [2024-08-04 15:09:29] [DEBUG] [206342] [ThreadPool.hpp] [52] - Thread-0 get a
    task
18 this is a task
19 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [28] - Thread-1 is
    running...
20 [2024-08-04 15:09:29] [INFO] [206342] [ThreadPool.hpp] [28] - Thread-4 is
    running...
21 [2024-08-04 15:09:30] [DEBUG] [206342] [ThreadPool.hpp] [109] - 任务入队列成功
22 [2024-08-04 15:09:30] [DEBUG] [206342] [ThreadPool.hpp] [52] - Thread-3 get a
    task
23 this is a task
24 ...
25 this is a task
26 [2024-08-04 15:09:39] [DEBUG] [206342] [ThreadPool.hpp] [88] - 线程池退出中...
27 [2024-08-04 15:09:44] [INFO] [206342] [ThreadPool.hpp] [95] - Thread-0 退出...
28 [2024-08-04 15:09:44] [INFO] [206342] [ThreadPool.hpp] [95] - Thread-1 退出...
29 [2024-08-04 15:09:44] [INFO] [206342] [ThreadPool.hpp] [95] - Thread-2 退出...
30 [2024-08-04 15:09:44] [INFO] [206342] [ThreadPool.hpp] [95] - Thread-3 退出...
31 [2024-08-04 15:09:44] [INFO] [206342] [ThreadPool.hpp] [95] - Thread-4 退出...
32
```



注意：

- 这里的代码中, 关于锁也可以改成RAII, 其实代码会更加优雅, 但是实际要不要更改, 可以结合课堂情况酌情进行

## 3-3 线程安全的单例模式

### 3-3-1 什么是单例模式

### 3-3-2 单例模式的特点

某些类, 只应该具有一个对象(实例), 就称之为单例.

例如一个男人只能有一个媳妇.

在很多服务器开发场景中, 经常需要让服务器加载很多的数据 (上百G) 到内存中. 此时往往要用一个单例的类来管理这些数据.

### 3-3-3 饿汉实现方式和懒汉实现方式

[洗碗的例子]

- 1 吃完饭, 立刻洗碗, 这种就是饿汉方式. 因为下一顿吃的时候可以立刻拿着碗就能吃饭.
- 2 吃完饭, 先把碗放下, 然后下一顿饭用到这个碗了再洗碗, 就是懒汉方式.

懒汉方式最核心的思想是 "延时加载". 从而能够优化服务器的启动速度.

### 3-3-4 饿汉方式实现单例模式

```
1 template <typename T>
2 class Singleton {
3     static T data;
4 public:
5     static T* GetInstance() {
6         return &data;
7     }
8 };
```

只要通过 Singleton 这个包装类来使用 T 对象, 则一个进程中只有一个 T 对象的实例.

### 3-3-5 懒汉方式实现单例模式

```
1 template <typename T>
2 class Singleton {
3     static T* inst;
```



```

4 public:
5     static T* GetInstance() {
6         if (inst == NULL) {
7             inst = new T();
8         }
9         return inst;
10    }
11 };

```

存在一个严重的问题, 线程不安全.

第一次调用 GetInstance 的时候, 如果两个线程同时调用, 可能会创建出两份 T 对象的实例.

但是后续再次调用, 就没有问题了.

### 3-3-6 懒汉方式实现单例模式(线程安全版本)

```

1 // 懒汉模式, 线程安全
2 template <typename T>
3 class Singleton {
4     volatile static T* inst; // 需要设置 volatile 关键字, 否则可能被编译器优化.
5     static std::mutex lock;
6 public:
7     static T* GetInstance() {
8         if (inst == NULL) { // 双重判定空指针, 降低锁冲突的概率, 提高性能.
9             lock.lock(); // 使用互斥锁, 保证多线程情况下也只调用一次 new.
10            if (inst == NULL) {
11                inst = new T();
12            }
13            lock.unlock();
14        }
15        return inst;
16    }
17 };

```

注意事项:

1. 加锁解锁的位置
2. 双重 if 判定, 避免不必要的锁竞争
3. volatile关键字防止过度优化

### 3-4 单例式线程池

ThreadPool.hpp

```

1 #pragma once
2
3 #include <iostream>
4 #include <vector>
5 #include <queue>
6 #include <memory>
7 #include <pthread.h>
8 #include "Log.hpp"      // 引入自己的日志
9 #include "Thread.hpp"   // 引入自己的线程
10 #include "Lock.hpp"     // 引入自己的锁
11 #include "Cond.hpp"     // 引入自己的条件变量
12
13 using namespace ThreadModule;
14 using namespace CondModule;
15 using namespace LockModule;
16 using namespace LogModule;
17
18 const static int gdefaultthreadnum = 10;
19
20 // 日志
21 template <typename T>
22 class ThreadPool
23 {
24 private:
25     // 是要有的，必须是私有的
26     ThreadPool(int threadnum = gdefaultthreadnum) : _threadnum(threadnum),
27     _waitnum(0), _isrunning(false)
28     {
29         LOG(LogLevel::INFO) << "ThreadPool Construct()";
30     }
31     void InitThreadPool()
32     {
33         // 指向构建出所有的线程，并不启动
34         for (int num = 0; num < _threadnum; num++)
35         {
36             _threads.emplace_back(std::bind(&ThreadPool::HandlerTask, this));
37             LOG(LogLevel::INFO) << "init thread " << _threads.back().Name() <<
38             " done";
39         }
40     }
41     void Start()
42     {
43         _isrunning = true;
44         for (auto &thread : _threads)
45         {
46             thread.Start();
47             LOG(LogLevel::INFO) << "start thread " << thread.Name() << "done";

```

```

46     }
47 }
48 void HandlerTask() // 类的成员方法，也可以成为另一个类的回调方法，方便我们继续类级
    别的互相调用！
49 {
50     std::string name = GetThreadNameFromNptl();
51     LOG(LogLevel::INFO) << name << " is running...";
52     while (true)
53     {
54         // 1. 保证队列安全
55         _mutex.Lock();
56         // 2. 队列中不一定有数据
57         while (_task_queue.empty() && _isrunning)
58         {
59             _waitnum++;
60             _cond.Wait(_mutex);
61             _waitnum--;
62         }
63         // 2.1 如果线程池已经退出了 && 任务队列是空的
64         if (_task_queue.empty() && !_isrunning)
65         {
66             _mutex.Unlock();
67             break;
68         }
69         // 2.2 如果线程池不退出 && 任务队列不是空的
70         // 2.3 如果线程池已经退出 && 任务队列不是空的 --- 处理完所有的任务，然后在
        退出
71         // 3. 一定有任务，处理任务
72         T t = _task_queue.front();
73         _task_queue.pop();
74         _mutex.Unlock();
75         LOG(LogLevel::DEBUG) << name << " get a task";
76         // 4. 处理任务，这个任务属于线程独占的任务
77         t();
78     }
79 }
80 // 复制拷贝禁用
81 ThreadPool<T> &operator=(const ThreadPool<T> &) = delete;
82 ThreadPool(const ThreadPool<T> &) = delete;
83
84 public:
85     static ThreadPool<T> *GetInstance()
86     {
87         // 如果是多线程获取线程池对象下面的代码就有问题了！！
88         // 只有第一次会创建对象，后续都是获取
89         // 双判断的方式，可以有效减少获取单例的加锁成本，而且保证线程安全

```

```
90     if (nullptr == _instance) // 保证第二次之后，所有线程，不用在加锁，直接返回
    _instance单例对象
91     {
92         LockGuard lockguard(_lock);
93         if (nullptr == _instance)
94         {
95             _instance = new ThreadPool<T>();
96             _instance->InitThreadPool();
97             _instance->Start();
98             LOG(LogLevel::DEBUG) << "创建线程池单例";
99             return _instance;
100         }
101     }
102     LOG(LogLevel::DEBUG) << "获取线程池单例";
103     return _instance;
104 }
105
106 void Stop()
107 {
108     _mutex.Lock();
109     _isrunning = false;
110     _cond.NotifyAll();
111     _mutex.Unlock();
112     LOG(LogLevel::DEBUG) << "线程池退出中...";
113 }
114 void Wait()
115 {
116     for (auto &thread : _threads)
117     {
118         thread.Join();
119         LOG(LogLevel::INFO) << thread.Name() << " 退出...";
120     }
121 }
122 bool Enqueue(const T &t)
123 {
124     bool ret = false;
125     _mutex.Lock();
126     if (_isrunning)
127     {
128         _task_queue.push(t);
129         if (_waitnum > 0)
130         {
131             _cond.Notify();
132         }
133         LOG(LogLevel::DEBUG) << "任务入队列成功";
134         ret = true;
135     }
```

```

136     _mutex.Unlock();
137     return ret;
138 }
139 ~ThreadPool()
140 {}
141
142 private:
143     int _threadnum;
144     std::vector<Thread> _threads; // for fix, int temp
145     std::queue<T> _task_queue;
146     Mutex _mutex;
147     Cond _cond;
148
149     int _waitnum;
150     bool _isrunning;
151
152     // 添加单例模式
153     static ThreadPool<T> *_instance;
154     static Mutex _lock;
155 };
156
157 template <typename T>
158 ThreadPool<T> *ThreadPool<T>::_instance = nullptr;
159
160 template <typename T>
161 Mutex ThreadPool<T>::_lock;
162

```

## 测试样例代码

```

1  #include <iostream>
2  #include <functional>
3  #include <unistd.h>
4  #include "ThreadPool.hpp"
5
6  using task_t = std::function<void()>;
7
8  void Download()
9  {
10     std::cout << "this is a task" << std::endl;
11 }
12
13 int main()
14 {
15     ENABLE_CONSOLE_LOG_STRATEGY();

```

```

16
17     int cnt = 10;
18     while(cnt)
19     {
20         ThreadPool<task_t>::GetInstance()->Enqueue(DownLoad);
21         sleep(1);
22         cnt--;
23     }
24
25     ThreadPool<task_t>::GetInstance()->Stop();
26
27     sleep(5);
28
29     ThreadPool<task_t>::GetInstance()->Wait();
30
31     return 0;
32 }

```

```

1 $ ./a.out
2 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [28] - ThreadPool
Construct()
3 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-0 done
4 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-1 done
5 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-2 done
6 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-3 done
7 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-4 done
8 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-5 done
9 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-6 done
10 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-7 done
11 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-8 done
12 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [36] - init thread
Thread-9 done
13 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
Thread-0done
14 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
Thread-1done

```

```
15 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [51] - Thread-0 is
    running...
16 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
    Thread-2done
17 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
    Thread-3done
18 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [51] - Thread-2 is
    running...
19 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
    Thread-4done
20 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [51] - Thread-3 is
    running...
21 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
    Thread-5done
22 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [51] - Thread-4 is
    running...
23 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [51] - Thread-5 is
    running...
24 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
    Thread-6done
25 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [51] - Thread-6 is
    running...
26 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
    Thread-7done
27 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [51] - Thread-7 is
    running...
28 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
    Thread-8done
29 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [45] - start thread
    Thread-9done
30 [2024-08-04 15:03:37] [DEBUG] [206234] [ThreadPool.hpp] [98] - 创建线程池单例
31 [2024-08-04 15:03:37] [DEBUG] [206234] [ThreadPool.hpp] [133] - 任务入队列成功
32 [2024-08-04 15:03:37] [INFO] [206234] [ThreadPool.hpp] [51] - Thread-1 is
    running...
33 [2024-08-04 15:03:37] [DEBUG] [206234] [ThreadPool.hpp] [75] - Thread-0 get a
    task
34 this is a task
35 ....
36 [2024-08-04 15:03:47] [DEBUG] [206234] [ThreadPool.hpp] [102] - 获取线程池单例
37 [2024-08-04 15:03:47] [DEBUG] [206234] [ThreadPool.hpp] [112] - 线程池退出中...
38 [2024-08-04 15:03:52] [DEBUG] [206234] [ThreadPool.hpp] [102] - 获取线程池单例
39 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-0 退出...
40 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-1 退出...
41 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-2 退出...
42 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-3 退出...
43 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-4 退出...
44 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-5 退出...
```

```
45 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-6 退出...
46 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-7 退出...
47 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-8 退出...
48 [2024-08-04 15:03:52] [INFO] [206234] [ThreadPool.hpp] [119] - Thread-9 退出...
```

## 4. 线程安全和重入问题

### 概念

**线程安全：**就是多个线程在访问共享资源时，能够正确地执行，不会相互干扰或破坏彼此的执行结果。一般而言，多个线程并发同一段只有局部变量的代码时，不会出现不同的结果。但是对全局变量或者静态变量进行操作，并且没有锁保护的情况下，容易出现该问题。

**重入：**同一个函数被不同的执行流调用，当前一个流程还没有执行完，就有其他的执行流再次进入，我们称之为重入。一个函数在重入的情况下，运行结果不会出现任何不同或者任何问题，则该函数被称为可重入函数，否则，是不可重入函数。

学到现在，其实我们已经能理解重入其实可以分为两种情况

- 多线程重入函数
- 信号导致一个执行流重复进入函数

### 常见的线程不安全的情况

- 不保护共享变量的函数
- 函数状态随着被调用，状态发生变化的函数
- 返回指向静态变量指针的函数
- 调用线程不安全函数的函数

### 常见不可重入的情况

- 调用了malloc/free函数，因为malloc函数是用全局链表来管理堆的
- 调用了标准I/O库函数，标准I/O库的很多实现都以不可重入的方式使用全局数据结构
- 可重入函数体内使用了静态的数据结构

### 常见的线程安全的情况

- 每个线程对全局变量或者静态变量只有读取的权限，而没有写入的权限，一般来说这些线程是安全的
- 类或者接口对于线程来说都是原子操作
- 多个线程之间的切换不会导致该接口的执行结果存在二义性

### 常见可重入的情况

- 不使用全局变量或静态变量
- 不使用用malloc或者new开辟出的空间
- 不调用不可重入函数
- 不返回静态或全局数据，所有数据都有函数的调用者提供
- 使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据

### 结论

不要被上面绕口令式的话语唬住，你只要仔细观察，其实对应概念说的都是一回事。



### 可重入与线程安全联系

- 函数是可重入的，那就是线程安全的(其实知道这一句话就够了)
- 函数是不可重入的，那就不能由多个线程使用，有可能引发线程安全问题
- 如果一个函数中有全局变量，那么这个函数既不是线程安全也不是可重入的。

### 可重入与线程安全区别

可重入函数是线程安全函数的一种

线程安全不一定是可重入的，而可重入函数则一定是线程安全的。

如果将对临界资源的访问加上锁，则这个函数是线程安全的，但如果这个重入函数若锁还未释放则会产生死锁，因此是不可重入的。

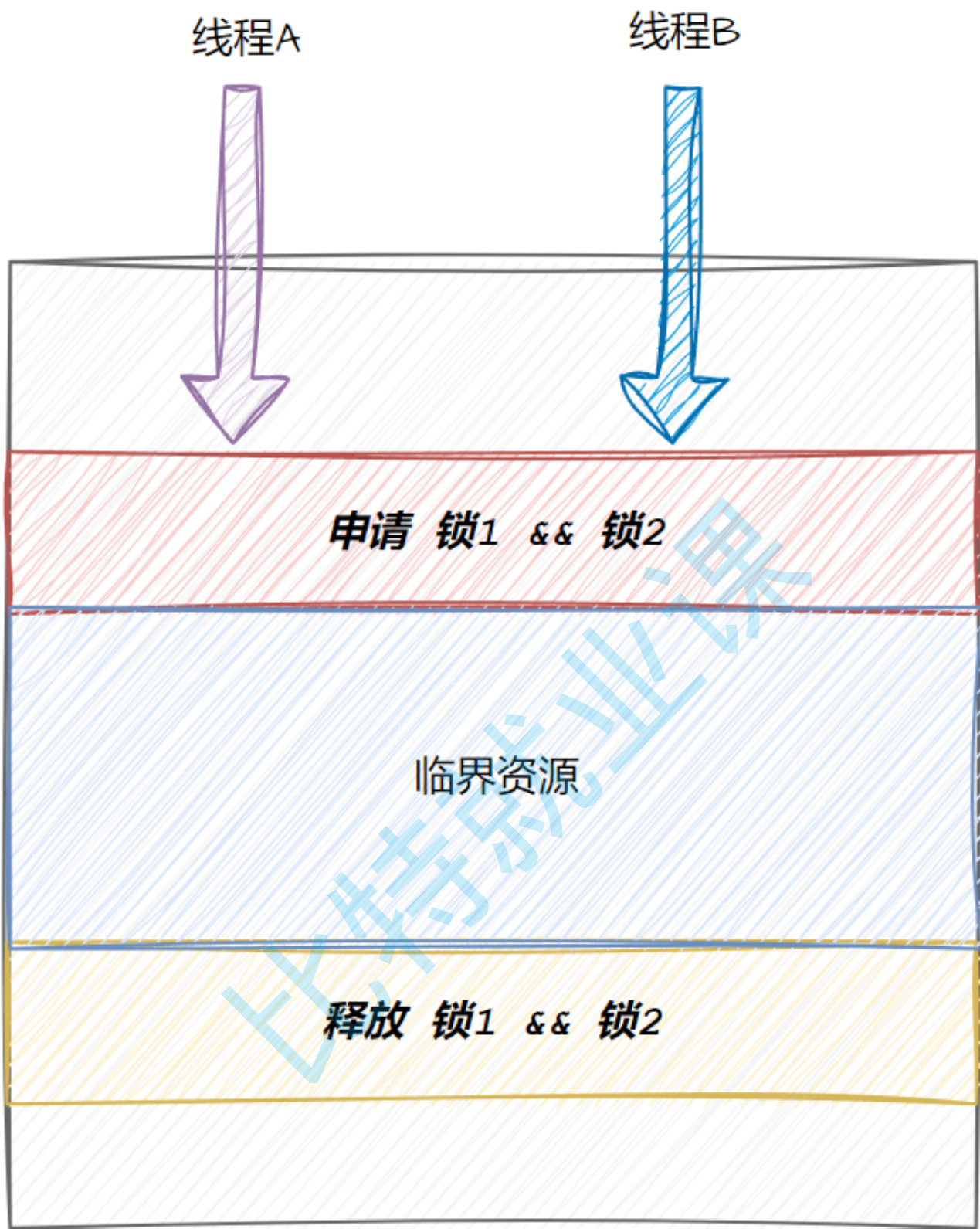
### 注意：

- 如果不考虑 **信号导致一个执行流重复进入函数** 这种重入情况，线程安全和重入在安全角度不做区分
- 但是线程安全侧重说明线程访问公共资源的安全情况，表现的是**并发线程的特点**
- 可重入描述的是一个函数是否能被重复进入，表示的是**函数的特点**

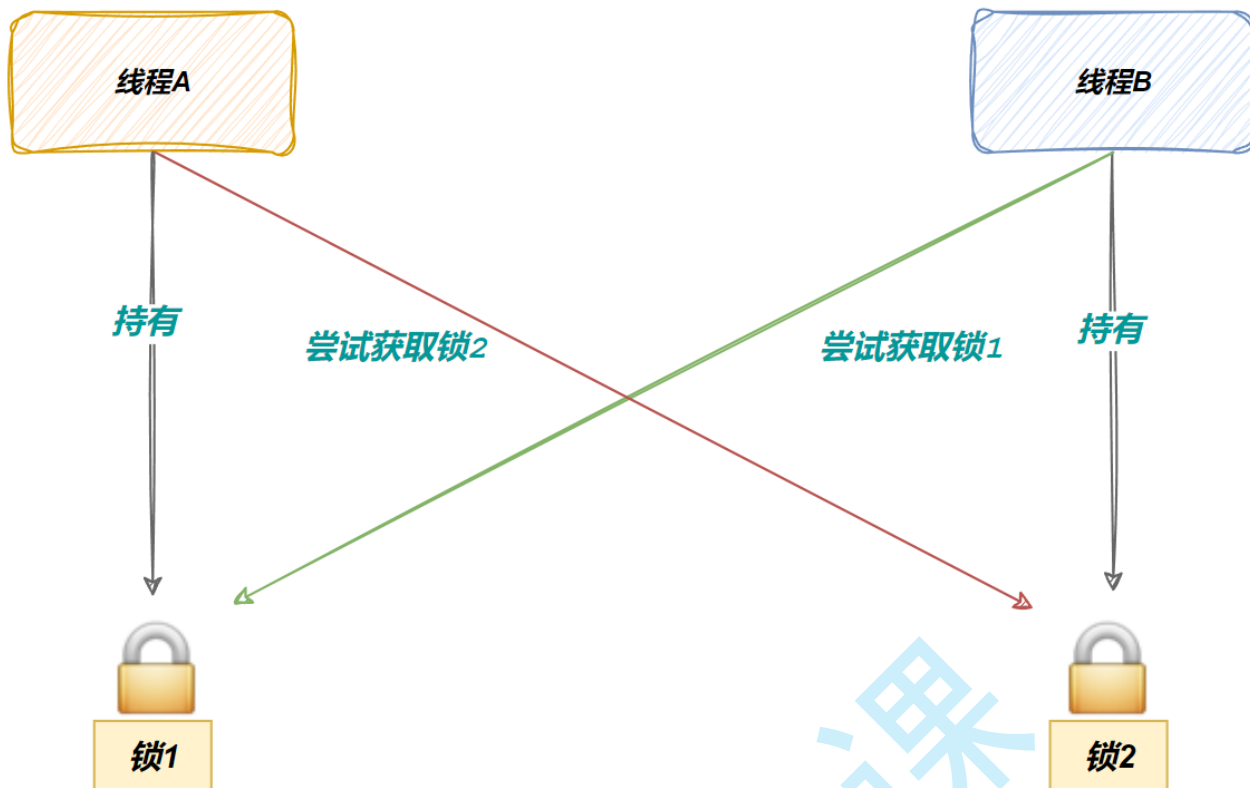
## 5. 常见锁概念

### 5-1 死锁

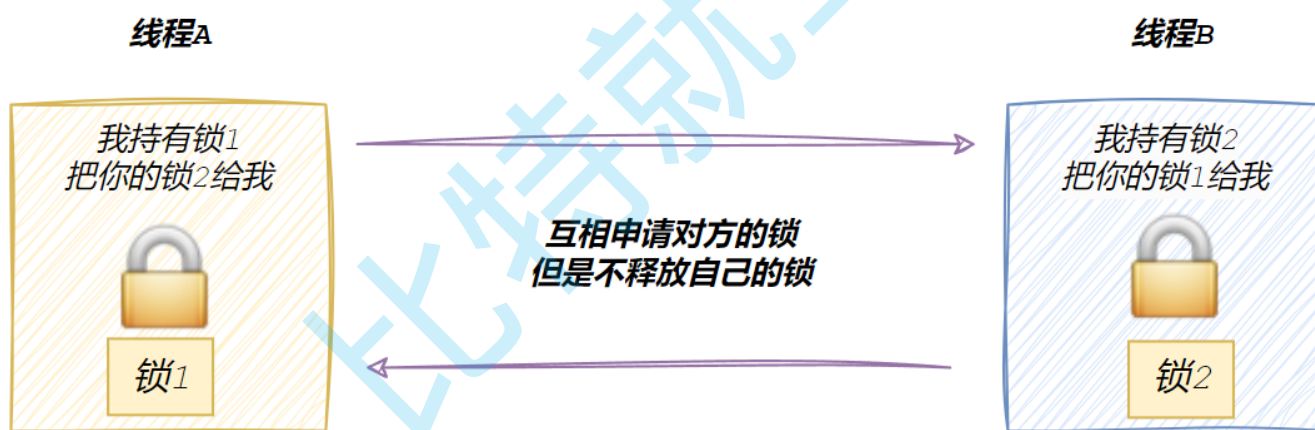
- 死锁是指在一组进程中的各个进程均占有不会释放的资源，但因互相申请被其他进程所站用不会释放的资源而处于的一种永久等待状态。
- 为了方便表述，假设现在线程A，线程B必须同时持有锁1和锁2，才能进行后续资源的访问



申请一把锁是原子的，但是申请两把锁就不一定了

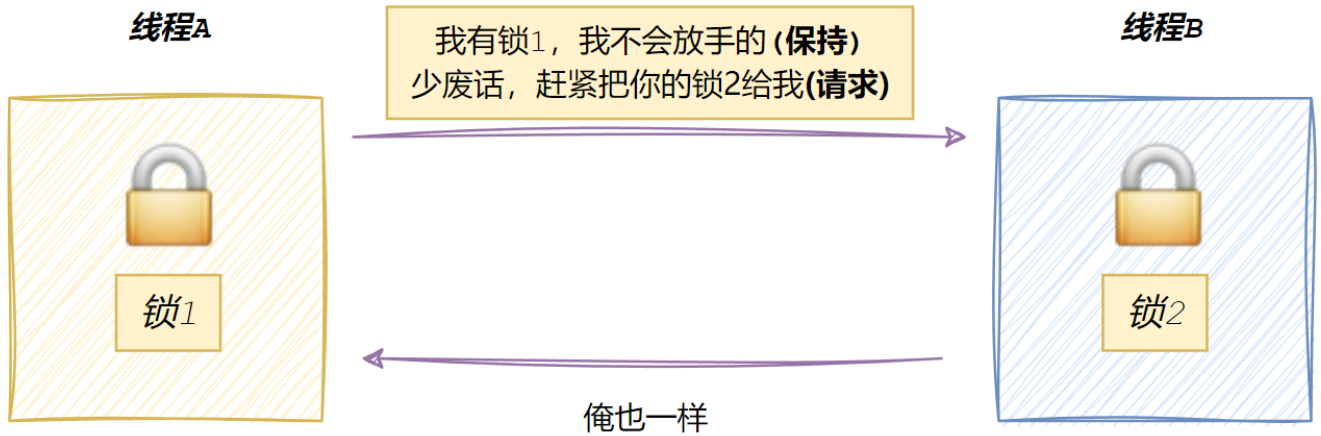


造成的结果是

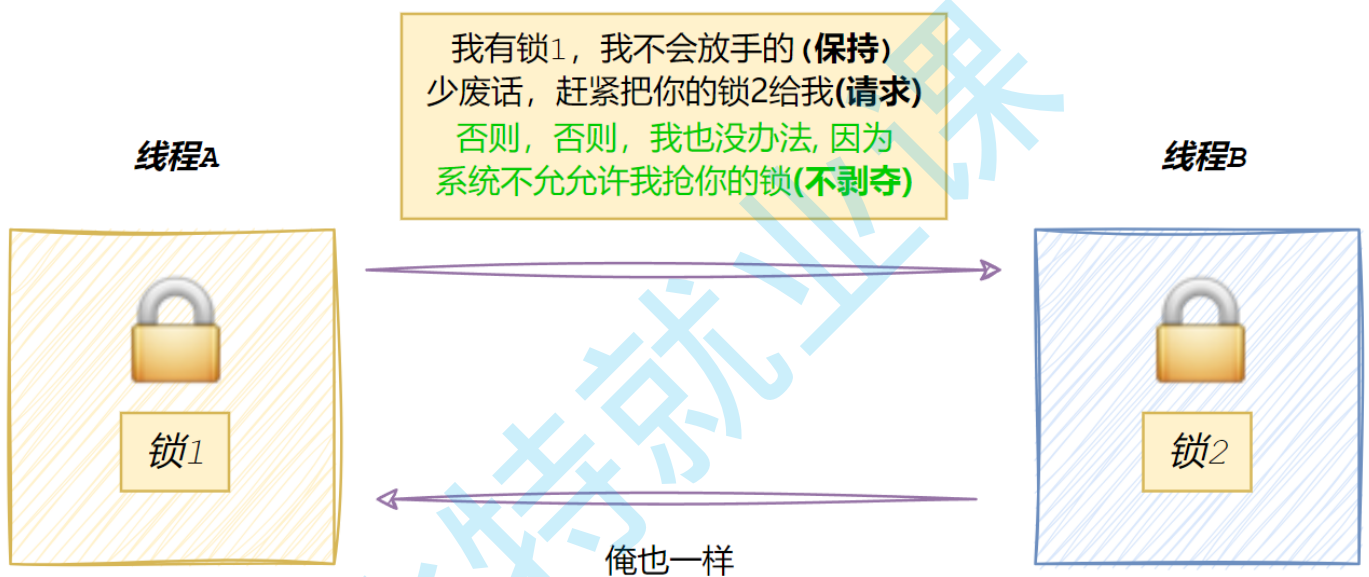


## 5-2 死锁四个必要条件

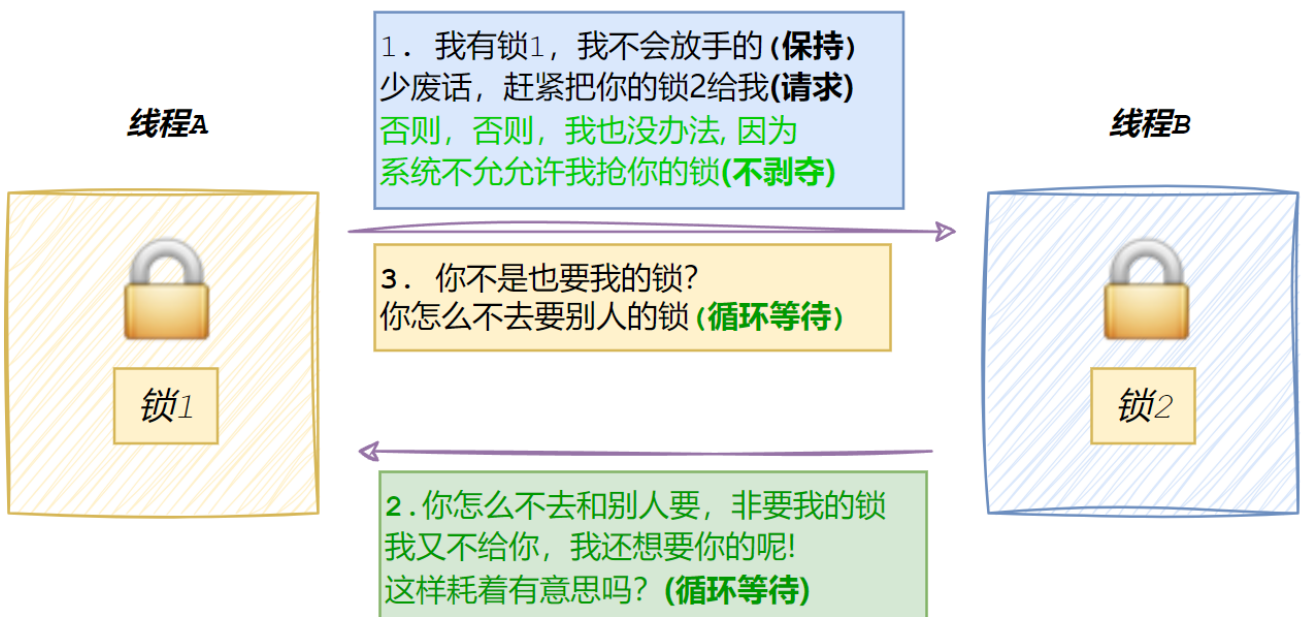
- **互斥条件：**一个资源每次只能被一个执行流使用
  - 好理解，不做解释
- **请求与保持条件：**一个执行流因请求资源而阻塞时，对已获得的资源保持不放



- **不剥夺条件:**一个执行流已获得的资源, 在未使用完之前, 不能强行剥夺



- **循环等待条件:**若干执行流之间形成一种头尾相接的循环等待资源的关系



## 5-3 避免死锁

- 破坏死锁的四个必要条件
  - 破坏循环等待条件问题：资源一次性分配，使用超时机制、加锁顺序一致

```
1 // 下面的C++不写了，理解就可以
2
3 #include <iostream>
4 #include <mutex>
5 #include <thread>
6 #include <vector>
7 #include <unistd.h>
8
9 // 定义两个共享资源（整数变量）和两个互斥锁
10 int shared_resource1 = 0;
11 int shared_resource2 = 0;
12 std::mutex mtx1, mtx2;
13
14 // 一个函数，同时访问两个共享资源
15 void access_shared_resources()
16 {
17     // std::unique_lock<std::mutex> lock1(mtx1, std::defer_lock);
18     // std::unique_lock<std::mutex> lock2(mtx2, std::defer_lock);
19     // // 使用 std::lock 同时锁定两个互斥锁
20     // std::lock(lock1, lock2);
21
22     // 现在两个互斥锁都已锁定，可以安全地访问共享资源
23     int cnt = 10000;
24     while (cnt)
25     {
26         ++shared_resource1;
27         ++shared_resource2;
28         cnt--;
29     }
30
31     // 当离开 access_shared_resources 的作用域时，lock1 和 lock2 的析构函数会被
    自动调用
32     // 这会导致它们各自的互斥量被自动解锁
33 }
34
35 // 模拟多线程同时访问共享资源的场景
36 void simulate_concurrent_access()
37 {
38     std::vector<std::thread> threads;
```



```

39
40 // 创建多个线程来模拟并发访问
41 for (int i = 0; i < 10; ++i)
42 {
43     threads.emplace_back(access_shared_resources);
44 }
45
46 // 等待所有线程完成
47 for (auto &thread : threads)
48 {
49     thread.join();
50 }
51
52 // 输出共享资源的最终状态
53 std::cout << "Shared Resource 1: " << shared_resource1 << std::endl;
54 std::cout << "Shared Resource 2: " << shared_resource2 << std::endl;
55 }
56
57 int main()
58 {
59     simulate_concurrent_access();
60     return 0;
61 }

```

```

1 $ ./a.out // 不一次申请
2 Shared Resource 1: 94416
3 Shared Resource 2: 94536

```

```

1 $ ./a.out // 一次申请
2 Shared Resource 1: 100000
3 Shared Resource 2: 100000

```

- 避免锁未释放的场景

## 5-4 避免死锁算法(不讲)

- 死锁检测算法(了解)
- 银行家算法 (了解)

## 6. STL,智能指针和线程安全

## 6-1 STL中的容器是否是线程安全的？

不是.

原因是, STL 的设计初衷是将性能挖掘到极致, 而一旦涉及到加锁保证线程安全, 会对性能造成巨大的影响.

而且对于不同的容器, 加锁方式的不同, 性能可能也不同(例如hash表的锁表和锁桶).

因此 STL 默认不是线程安全. 如果需要在多线程环境下使用, 往往需要调用者自行保证线程安全.

## 6-2 智能指针是否是线程安全的？

对于 `unique_ptr`, 由于只是在当前代码块范围内生效, 因此不涉及线程安全问题.

对于 `shared_ptr`, 多个对象需要共用一个引用计数变量, 所以会存在线程安全问题. 但是标准库实现的时候考虑到了这个问题, 基于原子操作(CAS)的方式保证 `shared_ptr` 能够高效, 原子的操作引用计数.

## 7. 其他常见的各种锁(不做介绍，具体看加餐课)

- 悲观锁：在每次取数据时，总是担心数据会被其他线程修改，所以会在取数据前先加锁（读锁，写锁，行锁等），当其他线程想要访问数据时，被阻塞挂起。
- 乐观锁：每次取数据时候，总是乐观的认为数据不会被其他线程修改，因此不上锁。但是在更新数据前，会判断其他数据在更新前有没有对数据进行修改。主要采用两种方式：版本号机制和CAS操作。
- CAS操作：当需要更新数据时，判断当前内存值和之前取得的值是否相等。如果相等则用新值更新。若不等则失败，失败则重试，一般是一个自旋的过程，即不断重试。
- 自旋锁，读写锁，加餐课详细介绍