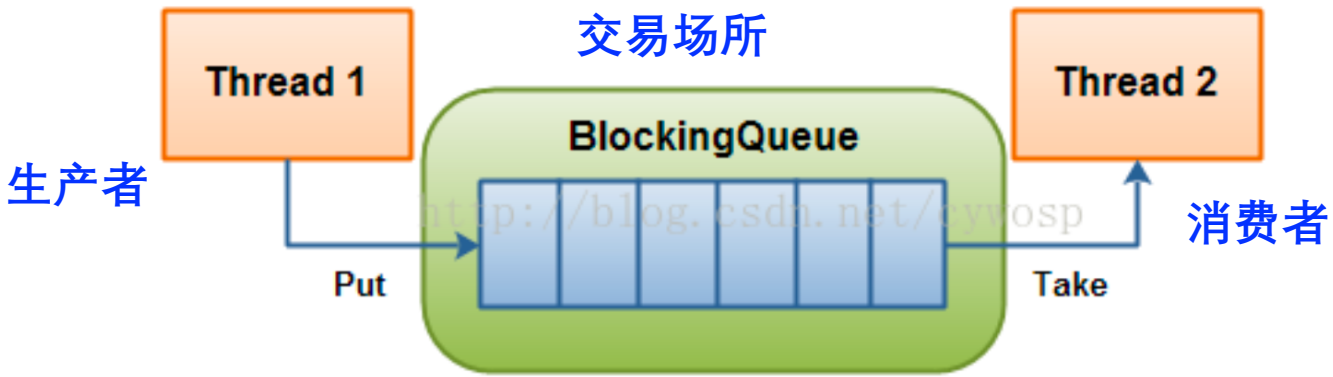
0114生产者消费者模型

基于BlockingQueue的生产者消费者模型

`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)



这个东西，是不是很像管道？
其实管道自带互斥和同步机制！
所以我们以前学的：
进程间通信的本质是：生产者消费者模型！

模型具体实现看代码

```
7   void* consumer(void *args)
8   {
9       BlockQueue<int> *bqueue = (BlockQueue<int>*)args;
10      while(true)
11      {
12          //不断消费
13          int a = 0;
14          bqueue->pop(&a);
15          std::cout << "消费一个数据: " << a << std::endl;
16          sleep(1); // 消费慢一点
17      }
18      return nullptr;
19  }
```

如果我们让消费慢一点

我们期待的现象就是
生产者一下子生产满了
然后生产一个消费一个

现象也确实如此

```
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0114]$ ./my
bash: ./my: No such file or directory
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0114]$ ./cp
生产一个数据: 消费一个数据: 11

生产一个数据: 2
生产一个数据: 3
生产一个数据: 4
生产一个数据: 5
生产一个数据: 6
消费一个数据: 2
生产一个数据: 7
消费一个数据: 3
生产一个数据: 8
消费一个数据: 4
生产一个数据: 9
消费一个数据: 5
生产一个数据: 10
^C
(base) [yufc@VM-12-12-centos:~/Core/BitCodeField/0114]$
```

当然，我们可以添加策略
可以让 bq里面的数据达到一定程度，或者下降到一定程度才开始生产或消费

```
65              // 此时满了！生产者应该等待!
66              pthread_cond_wait(&__Full, &__mtx);
67          }
68          // 2. 访问临界资源
69          __bq.push(in);
70          if (__bq.size() >= __capacity / 2)
71              pthread_cond_signal(&__Empty); // 唤醒消费者
72          pthread_mutex_unlock(&__mtx);       // 解锁
73      }
74      void pop(T *out)
75      {
76          pthread_mutex_lock(&__mtx);
77          if (__isQueueEmpty())
78          {
79              pthread_cond_wait(&__Empty, &__mtx);
80          }
81          *out = __bq.front();
82          __bq.pop();
83          pthread_mutex_unlock(&__mtx);
84          pthread_cond_signal(&__Full);
85      }
86  };
```

添加策略

```
    // 此时，pthread_cond_wait，会自动帮助我们先层获取锁(这些动作都是原子的，不用我们操心)
    // 但是，谁来唤醒呢？生产者的唤醒是由消费者做的！消费者给bq腾出位置，生产者就能生产了！
    if (this->__isQueueFull())
    {
        // 此时满了！生产者应该等待！
        pthread_cond_wait(&__Full, &__mtx); //首先这里是一个函数，它有可能调用失败！
    }
    // 2. 访问临界资源
    __bq.push(in);
```

此时代码还有问题！

1. pthread_cond_wait 可能调用失败！
2. pthread_cond_wait 可能存在伪唤醒！

```
    // 但是，谁来唤醒呢？生产者的唤醒是由消费者做的！消费者给bq腾出位置，生产者就能生产了！
    while(this->__isQueueFull())
    {
        // 此时满了！生产者应该等待！
        pthread_cond_wait(&__Full, &__mtx); //首先这里是一个函数，它有可能调用失败！
    }
    // 2. 访问临界资源
    __bq.push(in);
    if (__bq.size() >= __capacity / 2)
        pthread_cond_signal(&__Empty); // 唤醒消费者
    pthread_mutex_unlock(&__mtx);      // 解锁
}
void pop(T *out)
{
```

所以，应该用 while 进行判断！

基于生产者消费者模型的任务派发

```
void *consumer(void *args)
{
    BlockQueue<Task> *bqueue = (BlockQueue<Task> *)args;
    while (true)
    {
        // 不断消费
        int a = 0;
        Task t; // 获取任务
        bqueue->pop(&t);
        std::cout << "consumer: "
                  << "(" << t.__x << "," << t.__y << ")" << " = " << t() << std::endl;
        sleep(1); // 消费慢一点
    }
    return nullptr;
}
void *productor(void *args)
{
    BlockQueue<Task> *bqueue = (BlockQueue<Task> *)args;
    // int a = 1;
    while (true)
    {
        // 不断生产
        int x = rand() % 10 + 1;
        int y = rand() % 5 + 1;
        usleep(rand() % 1000);
        Task t(x, y, myadd); // 制作任务
        bqueue->push(t); //生产任务
        std::cout << "productor: "
                  << "(" << t.__x << "," << t.__y << ")" << " = " << "?" << std::endl;
    }
    return nullptr;
}
```

```cpp
int main()
{
    srand((uint_fast64_t)time(nullptr) ^ getpid() ^ 0x213);
    // 我们先弄一个生产，一个消费
    // BlockQueue<int> *bqueue = new BlockQueue<int>();
    BlockQueue<Task> *bqueue = new BlockQueue<Task>();

    pthread_t c[2], p[2];
    pthread_create(c, nullptr, consumer, bqueue); // 把bq传过去，在两个线程内部都可以看到这个阻塞队列了
    pthread_create(c + 1, nullptr, consumer, bqueue);
    pthread_create(p, nullptr, productor, bqueue);
    pthread_create(p + 1, nullptr, productor, bqueue);

    pthread_join(*c, nullptr);
    pthread_join(*(c + 1), nullptr);
    pthread_join(*p, nullptr);
    pthread_join(*(p + 1), nullptr);

    delete bqueue;
```

改成多生产多消费的样子
这个bq也可以支持

多生产和多消费最大意义是什么？

记住：我们期望的并发不是拿任务的时候并发，而是处理任务的时候并发

因此，多生产多消费最大的意义，就是可以让生产过程和消费过程并发！

```cpp
#include <iostream>
#include <pthread.h>


//封装一个锁
class Mutex
{
private:
    pthread_mutex_t *__pmtx;
public:
    Mutex(pthread_mutex_t *mtx)
        :__pmtx(mtx){}
    ~Mutex()
```

我们把锁封装一下
让代码更加优雅

```cpp
class lockGuard
{
public:
    lockGuard(pthread_mutex_t *mtx)
        :__mtx(mtx)
    {
        __mtx.lock();
    }
    ~lockGuard()
    {
        __mtx.unlock();
    }
private:
    Mutex __mtx;
};
```

```cpp
    // pthread_mutex_unlock(&__mtx);  // 不再需要解锁!
    // 这里lockGuard的析构
}
void pop(T *out)
{
    lockGuard lockguard(__mtx);
    // pthread_mutex_lock(&__mtx);
    while (__isQueueEmpty())
    {
        pthread_cond_wait(&__Empty, &__mtx);
    }
    *out = __bq.front();
    __bq.pop();
    pthread_cond_signal(&__Full);
    // pthread_mutex_unlock(&__mtx); // 不用自己解锁了
}
};
```

这里漏了个&

自动调用构造

不用解锁
lockGuard会自动调用
析构!

这个叫做RAII风格的加锁方式！