C++11线程

在c++ 11之前，c++没有对线程提供语言级别的支持，各种操作系统和编译器实现线程的方法不一样。

C++ 11 增加了线程以及线程相关的类，同一编程风格，简单易用，跨平台。

在linux中编译多线程 程序时，命令：**g++ demo.cpp -o demo -std=c++11 -lpthread**

1. 创建线程

头文件： **#include**

线程类： std::thread

线程对象不能拷贝，不能赋值，但可以交换，可以转移。

1.1构造函数：

thread() noexcept;

> 默认构造函数，构造一个线程对象，不执行任何任务（不会创建/启动子线程）

```
#include <iostream>
#include <Windows.h>
#include <thread>
using namespace std;


void func(int bh, const string& str)
{
    cout << "子线程:" << this_thread::get_id() << endl;    // this_thread表示子线程的id
    for (int ii = 1; ii <= 10; ii++) {
        cout << "第" << ii << "次表白: 亲爱的" << bh << "号, " << str << endl;
        //Sleep(1000);  // 休眠1秒
        this_thread::sleep_for(chrono::seconds(1)); //休眠1秒
    }
}

int main(void)
{
    // 用普通函数创建线程
    thread t1(func, 3, "我是一只傻傻鸟");
    thread t2(func, 8, "我是一只小小鸟");

    thread t3 = move(t2);   // t2被转移资源后，就不再代表线程了，join()会报错，t3才代表线程

    t1.join();  // 回收线程t1的资源
    t3.join();  // 回收线程t3的资源
}
```

template< class Function, class... Args >
explicit thread(Function&& fx, Args&&... args);

> 创建线程对象，在线程中执行任务函数fx中的代码，args是要传递给任务函数fx的参数。
>
> 任务函数fx可以是普通函数、类的非静态成员函数、类的静态成员函数、lambda函数、仿函数。

thread(const thread&) = delete;

> 删除拷贝构造函数，不允许线程对象之间的拷贝。

- **thread(thread&& other) noexcept;**

> 移动构造函数，将线程other的资源所有权转移给新创建的线程对象。*转移之后，原来的线程对象不代表线程。*

1.2 赋值函数

- **thread& operator= (thread&& other) noexcept;**
- **thread& operator= (const other&) = delete;**

> 线程中的资源不能被复制，如果other是右值，会进行资源所有权的转移，如果other是左值，禁止拷贝。

注意：

- 先创建的子线程不一定跑得最快（程序运行的速度有很大的偶然性）。
- 线程的任务函数返回后，子线程将终止。
- 如果主程序（主线程）退出（不论是正常退出还是意外终止），全部的子线程将强行被终止。

```
#include <iostream>
#include <thread>
#include <Windows.h>
using namespace std;

// 普通函数
void func(int bh, const string& str)
{
    for (int ii = 1; ii <= 10; ii++) {
        cout << "第" << ii << "次表白：亲爱的" << bh << "号，" << str << endl;
        Sleep(1000);    // 休眠1秒
    }
}

// 仿函数
class mythread1 {
public:
    void operator()(int bh, const string& str)
    {
        for (int ii = 1; ii <= 10; ii++) {
            cout << "第" << ii << "次表白：亲爱的" << bh << "号，" << str << endl;
            Sleep(1000);    // 休眠1秒
        }
    }
};

// 类的静态成员函数
class mythread2 {
public:
    static void func(int bh, const string& str)
    {
        for (int ii = 1; ii <= 10; ii++) {
            cout << "第" << ii << "次表白：亲爱的" << bh << "号，" << str << endl;
            Sleep(1000);    // 休眠1秒
        }
    }
};

// 类的非静态成员函数
class mythread3 {
public:
    void func(int bh, const string& str)
```

```
    {
        for (int ii = 1; ii <= 10; ii++) {
            cout << "第" << ii << "次表白：亲爱的" << bh << "号，" << str << endl;
            Sleep(1000);    // 休眠1秒
        }
    }
};

//! main函数中的代码叫主程序，主线程或主进程
//! 对象t1是子线程。主线程只有一个，子线程可以有很多，与计算机的硬件资源有关（普通的电脑可以创建几百个子线程，好的服务器可以创建几千个子线程）

int main(void)
{
//TODO: 程序一边执行func()函数中的任务，一边执行main函数中的任务，两个任务是同时执行的

    //thread t1(func, 3, "我是一只傻傻鸟");
    //thread t2(func, 8, "我是一只小小鸟");

    // 用Lambda函数创建线程
    auto f = [](int bh, const string& str)->void
    {
        for (int ii = 1; ii <= 10; ii++) {
            cout << "第" << ii << "次表白：亲爱的" << bh << "号，" << str << endl;
            Sleep(1000);    // 休眠1秒
        }
    };
    //thread t3(f, 3, "我是一只傻傻鸟");

    // 用仿函数创建线程
    //thread t4(mythread1(), 3, "我是一只傻傻鸟"); // 第一个参数用仿函数的匿名对象

    // 用类的静态成员创建线程
    //thread t5(mythread2::func, 3, "我是一只傻傻鸟"); // 类的静态成员函数与普通函数的性质是一样的

    // 用类的普通函数创建线程
    mythread3 myth; // 必须先创建类的对象，必须保证对象的生命周期比子线程要长
    thread t6(&mythread3::func,&myth, 3, "我是一只傻傻鸟");
    // 第一个参数填成员函数的地址，第二个参数填对象的地址，即this指针

    // 主线程（主程序）
    cout << "任务开始" << endl;
    for (int ii = 0; ii < 10; ii++) {
        cout << "执行任务中..." << endl;
        Sleep(1000);
    }
    cout << "任务完成" << endl;

    //t1.join();    // 回收线程t1的资源
    //t2.join();    // 回收线程t2的资源
    //t3.join();    //回收线程t3的资源
    //t4.join();        //回收线程t4的资源
    //t5.join();        //回收线程t5的资源
    t6.join();
}
```

2. 线程资源的回收

虽然同一个进程的多个线程共享进程的栈空间，但是，每个子线程在这个栈中拥有自己私有的栈空间。所以，线程结束时需要回收资源。

子线程的资源一定要回收，如果不回收，会产生僵尸线程，程序还会报错。

回收子线程的资源有两种方法：

- 在主程序中，调用 **join()** 成员函数等待子线程退出，回收它的资源。如果子线程已退出， **join()** 函数立即返回；否则会产生阻塞，直到子线程退出。

- 在子线程中，调用 **detach()** 成员函数分离子线程，子线程退出时，系统自动回收资源。（ **分离后的子线程不能join()** ，会报错）

用 joinable() 成员函数可以判断子线程的分离状态，函数返回布尔类型。

用detach()分离子线程后，运行结果什么也没有，为什么？

- 主程序启动子线程后，再分离子线程，然后主程序就退出了，子线程根本没时间运行。

那怎么办？

- 主程序不能退出。让主程序Sleep(12000)秒，休眠12秒，因为主程序一退出，整个程序都完了

```
int main(){
    thread t1(func, 3, "我是一只傻傻鸟");
    thread t2(func, 8, "我是一只小小鸟");

    t1.detach(); t2.detach();   // 将子线程t1和t2分离出去

    Sleep(12000);
}
```

3. this_thread的全局函数

C++11 提供了**命名空间this_thread**来表示当前线程，该命名空间中有四个函数：**get_id()，sleep_for()，sleep_until()， yield()** 。

this_thread 这个命名空间表示当前进程，与对象中的this指针类似。

- **get_id()**

thread::id this_thread::get_id() noexcept;

该函数用于获取线程ID，thread类也有同名的成员函数。

在多线程的程序中，每个线程都有一个id，用于识别线程，就像身份证号码一样，每个人都有，不会重复
相同的程序，每次运行，产生的线程id是不一样的

```
// 在多线程的程序中，每个线程都有一个id，用于识别线程，就像身份证号码一样，每个人都有，不会重复
// 相同的程序，每次运行，产生的线程id是不一样的
#include <iostream>
#include <Windows.h>
#include <thread>
using namespace std;


void func(int bh, const string& str)
{
    cout << "子线程:" << this_thread::get_id() << endl;    // this_thread表示子线程的id
    for (int ii = 1; ii <= 10; ii++) {
        cout << "第" << ii << "次表白: 亲爱的" << bh << "号, " << str << endl;
        Sleep(1000);    // 休眠1秒
    }
}

int main(void)
{
    // 用普通函数创建线程
    thread t1(func, 3, "我是一只傻傻鸟");
    thread t2(func, 8, "我是一只小小鸟");

    cout <<"主线程:" << this_thread::get_id() << endl; // 这行代码出现在主程序中，this_thread表示
主线程

    // 用线程对象的get_id()成员函数获取子线程的id
    cout << "子线程t1:" << t1.get_id() << endl;
```

```
    cout << "子线程t2:" << t2.get_id() << endl;

    t1.join();  // 回收线程t1的资源
    t2.join();  // 回收线程t2的资源
}
```

- **sleep_for()**　　VS　　Sleep(1000)　　Linux(头文件是 unistd.h) sleep(1)

> template <class Rep, class Period>
>
> void sleep_for (const chrono::duration<Rep,Period>& rel_time);
>
> 该函数让线程休眠一段时间。

eg:　**this_thread::sleep_for(chrono::seconds(1));　//休眠1秒**

- **sleep_until()　2022-01-01 12:30:35**

> template <class Clock, class Duration>
>
> void sleep_until (const chrono::time_point<Clock,Duration>& abs_time);
>
> 该函数让线程休眠至指定时间点。（可实现定时任务）

- **yield()**

> void yield() noexcept;
>
> 该函数让线程主动让出自己已经抢到的CPU时间片。

- **thread类的其他的成员函数**

> void swap(std::thread& other);　// 交换两个线程对象。
>
> static unsigned hardware_concurrency() noexcept;　// 返回硬件线程上下文的数量。
>
> - 这两个函数不属于this_thread命名空间

4. call_once函数

在多线程环境中，某些函数只能被调用一次。例如：初始化某个对象，而这个对象只能被初始化一次。

在线程的任务函数中，可以用 **std::call_once()** 来保证某个函数只被调用一次。

头文件：　**#include**

定义：

template< class callable, class... Args >

void call_once(std::once_flag& flag, Function&& fx, Args&&... args);

> 第一个参数是std::once_flag，用于标记函数fx是否已经被执行过。
>
> 第二个参数是需要执行的函数fx。
>
> 后面的可变参数是传递给函数fx的参数。

`std::once_flag` 的头文件是　**#include**

> once_flag onceflag;　// 定义一个once_flag的全局变量。本质是取值为0和1的锁

```cpp
#include <iostream>
#include <mutex>    // std::call_once()和std::once_flag函数需要包含的头文件
#include <thread>
using namespace std;

once_flag onceflag; // 定义一个once_flag的全局变量。本质是取值为0和1的锁

// 在线程中，打算只调用一次（假设只能在线程的任务函数中调用）
void once_func(const int bh, const string& str)
{
    cout << "once_func()  bh=" << bh << ",str=" << str << endl;
}

// 在多线程的程序中，多个线程可能用同一个线程任务函数，这样的话，每个线程都会执行这行代码，调用
once_func()函数
void func(int bh, const string& str)
{
    call_once(onceflag,once_func,0, "个位观众老爷，我要开始表白了");

    for (int ii = 1; ii <= 3; ii++) {
        cout << "第" << ii << "次表白：亲爱的" << bh << "号，" << str << endl;
        this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒
    }
}



int main(void)
{
    //用普通函数创建线程
    thread t1(func, 3, "我是一只傻傻鸟");
    thread t2(func, 8, "我是一只小小鸟");

    t1.detach();    // 分离子线程t1
    t2.detach();    // 分离子线程t1

    this_thread::sleep_for(chrono::seconds(10));    // 主线程休眠10秒

}
```

5. native_handle 函数

C++11 定义了线程标准，不同的平台和编译器在实现的时候，本质上都是对操作系统的线程库进行封装，会损失一部分功能。

为了弥补C++11线程库的不足，**thread类**提供了**native_handle() 成员函数**，用于**获得与操作系统相关的原生线程句柄**，操作系统原生的线程库就可以用原生的线程句柄操作线程。

Linux中原生的线程库是 pthread ， 头文件是： #include<pthread.h>

> 现在有一个需求，在子线程运行过程中，如果想中止它，怎么办？
>
> - c++11的线程库好像没有这个功能，Linux线程库有这个功能，函数名 pthread_cancel()
>
> int pthread_cancel(pthread_t thread); 需要一个参数(线程id)。这个线程id与c++11的线程id不一样，不是一个东西。
>
> c++11 的thread类没有这个线程id，但是thread类提供了 native_habdle()函数，获得与原生系统相关的原生线程句柄

```cpp
#include <iostream>
#include <thread>
#include <pthread.h>    // linux的pthread线程库头文件
using namespace std;

void func()     // 线程任务函数
{
```

```
    for(int ii=1;ii<=10;ii++){
        cout<<"ii="<<ii<<endl;
        this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒
    }
}
// 现在有一个需求，在子线程运行过程中，如果想中止它，怎么办?
// c++11的线程库好像没有这个功能，Linux线程库有这个功能，函数名 pthread_cancel()
//  int pthread_cancel(pthread_t thread); 需要一个参数，线程id。这个线程id与c++11的线程id不一样，不是一个东西
// c++11 的thread类没有这个线程id，但是thread类提供了 native_habdle()函数，获得与原生系统相关的原生线程句柄

int main(void)
{   // 主程序启动（创建）子线程后，主线程休眠5秒，然后取消子线程
    thread tt(func);     // 创建一个线程

    this_thread::sleep_for(chrono::seconds(5)); // 休眠5秒

    pthread_t thid=tt.native_handle();  // 获得Linux操作系统的线程句柄

    // 用Linux原生的线程库操作线程
    pthread_cancel(rhid);    // 取消线程

    tt.join();  // 等待tt线程退出，回收tt的资源
}
```

命令： **g++ aaa.cpp -o aaa -std=c++11 -lpthread**

注意： 将 pthread线程库连接进来

6. 线程安全

同一进程中的多个线程共享该进程中全部的系统资源。这样的话，**多个线程访问同一共享资源，就会产生冲突**。

单核CPU（一个核）在同一时刻只能运行一个线程。（CPU的调度单位是线程）

> 为什么会冲突呢?
>
> 举个例子，一套房间有多个人住，但是，卫生间只有一个，多个人使用卫生间就会产生冲突。

- **顺序性**

> 程序按照代码的先后顺序执行
>
> CPU为了提高程序整体的执行效率，可能会对代码进行优化，按更高效的顺序执行
>
> CPU虽然不保证完全按照代码的顺序执行，但它会保证最终的结果和按代码顺序执行时的结果一致。

- **可见性**

> 线程操作共享变量时，会将该变量从内存加载到CPU缓存中，修改该变量后，CPU会立即更新缓存，但不一定会立即将它写回内存。这时候，如果其他线程访问该变量，从内存中读到的是旧数据，而非第一个线程操作后的数据。
>
> 当多个线程并发访问共享变量时，一个线程对共享变量的修改，其他线程能够立即看到。

- **原子性**

> CPU执行指令：读取指令，读取内存，执行指令，写回内存
>
> i++　　1)从内存读取i的值 2）把i+1 3) 把结果写回内存
>
> 一个操作（有可能包含多个步骤）要么全部执行（生效），要么全部不执行（都不生效）

- 如何保证线程安全

1.volatile关键字

（1）保证内存变量的可见性　（2）禁止代码优化（重排序）

volatile int aa = 0; 将全局变量声明为volatile后，依旧没有解决问题线程安全问题。

2.原子操作（原子类型）

3.线程同步（锁）

```cpp
#include <iostream>
#include <thread>
using namespace std;

//普通函数
void func(int bh, const string& str)
{
    for (int ii = 1; ii <= 3; ii++) {
        cout << "第" << ii << "次表白: 亲爱的" << bh << "号, " << str << endl;
        this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒
    }
}

// 仿函数
class mythread1 {
public:
    void operator()(int bh, const string& str)
    {
        for (int ii = 1; ii <= 3; ii++) {
            cout << "第" << ii << "次表白: 亲爱的" << bh << "号, " << str << endl;
            this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒
        }
    }
};

// 类的静态成员函数
class mythread2 {
public:
    static void func(int bh, const string& str)
    {
        for (int ii = 1; ii <= 3; ii++) {
            cout << "第" << ii << "次表白: 亲爱的" << bh << "号, " << str << endl;
            this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒
        }
    }
};

// 类的非静态成员函数
class mythread3 {
public:
    void func(int bh, const string& str)
    {
        for (int ii = 1; ii <= 3; ii++) {
            cout << "第" << ii << "次表白: 亲爱的" << bh << "号, " << str << endl;
            this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒
        }
    }
};

int main(void)
{
    // 用普通函数创建线程
    thread t1(func, 3, "我是一只傻傻鸟");
    thread t2(func, 8, "我是一只小小鸟");

    //用Lambda函数创建线程
    auto f = [](int bh, const string& str)->void
    {
        for (int ii = 1; ii <= 3; ii++) {
            cout << "第" << ii << "次表白: 亲爱的" << bh << "号, " << str << endl;
```

```
            this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒
        }
    };
    thread t3(f, 3, "我是一只傻傻鸟");

    // 用仿函数创建线程
    thread t4(mythread1(), 3, "我是一只傻傻鸟");

    // 用类的静态成员函数创建线程
    thread t5(mythread2::func, 3, "我是一只傻傻鸟");

    // 用类的非静态成员函数创建线程
    mythread3 thd;
    thread t6(&mythread3::func, &thd, 3, "我是一只傻傻鸟");
    // 第一个参数是成员函数的地址，第二个参数是对象的地址，即this指针

     // 主线程（主程序）
    cout << "任务开始" << endl;
    for (int ii = 0; ii < 3; ii++) {
        cout << "执行任务中..." << endl;
        this_thread::sleep_for(chrono::seconds(1));
    }
    cout << "任务完成" << endl;


    t1.join();  // 回收t1的资源
    t2.join();  // 回收t2的资源
    t3.join();  // 回收t3的资源
    t4.join();  // 回收t4的资源
    t5.join();  // 回收t5的资源
    t6.join();  // 回收t6的资源
}
```

cout是全局对象，在这个demo程序中，全部的线程共享cout对象，每个线程都用它向屏幕输出数据。如果在同一时间点有多个线程使用cout，结果就会有点乱。

```
#include <iostream>
#include <thread>
using namespace std;

int aa = 0; // 定义全局变量

// 普通函数，把全局变量aa加1000000次
void func()
{
    for (int ii = 0; ii < 1000000; ii++) {
        aa++;
    }
}

int main(void)
{
    //func();
    //func();

    thread t1(func);    // 创建线程t1,把全局变量aa加到1000000次
    thread t2(func);    // 创建线程t2,把全局变量aa加到1000000次

    t1.join();  // 回收t1的资源
    t2.join();  // 回收t2的资源

    cout <<"aa= " << aa << endl;
}
```

创建两个线程t1,t2，全局变量的值应该为2000000，但结果不是

7. 线程同步

线程同步是指多个线程协同工作，协商如何使用共享资源。

C++11的线程同步包含三个方面：（1）互斥锁（互斥量） （2）条件变量 （3）生产/消费者模型

8. 互斥锁

互斥锁和公共厕所一样，如果厕所没人，那就进去上锁，解决问题，解锁出来；如果厕所里有人，那就排队等待。

互斥锁只有加锁和解锁，确保同一时间只有一个线程访问共享资源。

线程在访问共享资源之前，申请加锁，访问完成之后释放锁（解锁）。

如果某线程持有锁，其他的线程形成等待队列。

C++11提供了四种互斥锁：

- **mutex** 互斥锁
- **timed_mutex** 带超时机制的互斥锁
- **recursive_mutex** 递归互斥锁
- **recursive_timed_mutex** 带超时机制的递归互斥锁

包含头文件： **#include**

注意：操作系统提供的互斥锁只有一种，就叫互斥锁。为了方便使用，C++11把它封装成了四个类

1. mutex 类

1.1加锁 lock()

互斥锁有 **锁定和未锁定** 两种状态。

如果互斥锁是未锁定状态，调用 lock()成员函数的 线程会得到互斥锁的所有权，并将其上锁。

如果互斥锁是锁定状态，调用 lock() 成员函数的线程就会阻塞等待，直至互斥锁变成未锁定状态。

1.2 解锁 unlock()

线程加了锁之后，就可以使用共享资源了，用完再解锁。

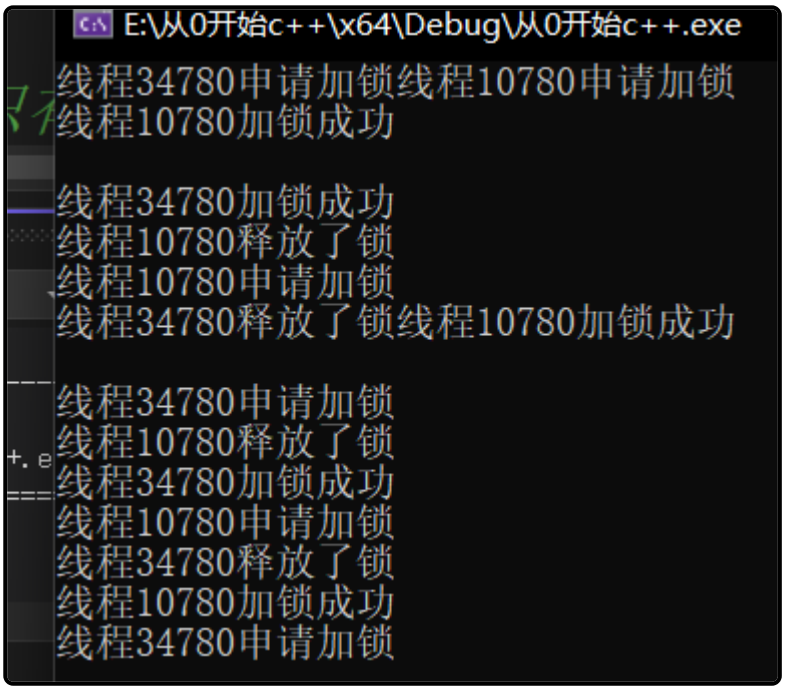
只有持有锁的线程才能解锁。

```
#include <iostream>
#include <mutex>
#include <thread>
using namespace std;

mutex mtx;  // 创建互斥锁对象，保护共享资源cout对象

//普通函数
void func(int bh, const string& str)
{
    for (int ii = 1; ii <= 10; ii++) {
        // 在线程中，每次使用cout全局对象之前，申请加锁
        mtx.lock(); // 申请加锁
        cout << "第" << ii << "次表白：亲爱的" << bh << "号，" << str << endl;
        mtx.unlock();   // 用完了就解锁
        this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒
    }
}
```

```
// cout是全局对象，每个线程都使用它向屏幕输出数据，如果在同一时刻有多个线程使用cout，那么结果会有点乱
//! 用互斥锁给 全局对象cout 加锁

int main(void)
{
    //用普通函数创建线程
    thread t1(func, 1, "我是一只傻傻鸟");
    thread t2(func, 2, "我是一只傻傻鸟");
    thread t3(func, 3, "我是一只傻傻鸟");
    thread t4(func, 4, "我是一只傻傻鸟");
    thread t5(func, 5, "我是一只傻傻鸟");


    t1.join();  // 回收t1的资源
    t2.join();  // 回收t2的资源
    t3.join();  // 回收t3的资源
    t4.join();  // 回收t4的资源
    t5.join();  // 回收t5的资源

}
```

如果持有锁的时间比较长，lock()函数就会阻塞等待。

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

// 创建互斥锁对象，保护共享资源aa
mutex mtx;

int aa = 0; // 定义全局变量

// 普通函数，把全局变量aa加1000000次
void func()
{
    for (int ii = 0; ii < 1000000; ii++) {
        cout << "线程" << this_thread::get_id() << "申请加锁" << endl;
        mtx.lock(); // 使用之前，加锁
        cout << "线程" << this_thread::get_id() << "加锁成功" << endl;
        aa++;
        this_thread::sleep_for(chrono::seconds(5)); // 休眠5秒
        mtx.unlock();   // 使用之后，解锁
        cout << "线程" << this_thread::get_id() << "释放了锁" << endl;
        this_thread::sleep_for(chrono::seconds(1)); // 休眠1秒之后，再回到for循环，再去申请加锁
    }
}

// 两个线程申请锁，只有一个成功，另一个在等待，5秒之后释放锁，等待中的线程申请锁成功，所在两个线程之间切换
int main(void)
{
    //func();
    //func();

    thread t1(func);    // 创建线程t1,把全局变量aa加到1000000次
    thread t2(func);    // 创建线程t2,把全局变量aa加到1000000次

    t1.join();  // 回收t1的资源
    t2.join();  // 回收t2的资源

    cout <<"aa= " << aa << endl;
}
```



1.3 尝试加锁 trylock()

如果互斥锁是未锁定状态，则加锁成功，函数返回 true

如果互斥锁是锁定状态，则加锁失败，函数立即返回 false。（**线程不会阻塞等待**）

应用场景：如果公共卫生间只有一个，加锁失败只能排队。如果公共卫生间有多个，尝试加锁，如果成功，你就进去了；如果失败，不会排队，对另一个卫生间尝试加锁

线程中的共享资源有时候也有多个，处理方法和上面类似。

2. timed_mutex 类（带超时机制的互斥锁）

比mutex类增加了两个成员函数：

bool try_lock_for(时间长度);

bool try_lock_until(时间点);

```
timed_mutex tm;
tm.try_lock_for(chrono::seconds(10));
```

应用场景： 假设卫生间被占用的时间太长，有人憋不住，不得已，可以跑到树林里。

3. recursive_mutex 类（递归互斥锁）

递归互斥锁允许同一线程多次获得互斥锁，可以解决同一线程多次加锁失败而造成死锁问题。

```
// 出现死锁问题
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

class AA {
private:
    mutex m_mutex;
public:
    void func1() {
        m_mutex.lock(); // 申请加锁
        cout << "调用了funcn1()" << endl;
        m_mutex.unlock();   // 解锁
```

```
    }

    void func2() {
        m_mutex.lock(); // 申请加锁
        cout << "调用了funcn2()" << endl;
        func1();
        m_mutex.unlock();    // 解锁
    }
};

int main(void)
{
    AA aa;
    //aa.func1();
    aa.func2(); // 出现了死锁，普通的互斥锁必须在解锁之后才能加锁
    // func2() 加锁后，互斥锁处于锁定状态，再去调用func1()，func1()这边不可能申请到锁，这样就出现了死锁。
}
```

```
// 使用 recursive_mutex 递归互斥锁避免出现死锁
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

class AA {
private:
    recursive_mutex m_mutex;    // 递归互斥锁
public:
    void func1() {
        m_mutex.lock(); // 申请加锁
        cout << "调用了funcn1()" << endl;
        m_mutex.unlock();    // 解锁
    }

    void func2() {
        m_mutex.lock(); // 申请加锁
        cout << "调用了funcn2()" << endl;
        func1();
        m_mutex.unlock();    // 解锁
    }
};

int main(void)
{
    AA aa;
    //aa.func1();
    aa.func2();
}
```

4. lock_guard 类

lock_guard是模板类，可以简化互斥锁的使用，也更安全。

lock_guard 的定义：

```
template<class Mutex>
class lock_guard
{
    explicit lock_guard(Mutex& mtx);
}
// 构造函数的参数是上面四种互斥锁之一
```

lock_guard在构造函数中加锁，在析构函数中解锁。

`lock_guard` 采用了**RAII**思想（在类构造函数中分配资源，在析构函数中释放资源，保证资源在离开作用域时自动释放）。

在程序代码中，我们只需加锁就行了，不用管解锁。（就像智能指针，也是采用RAII思想）

> lock_guard mlock(mtx);

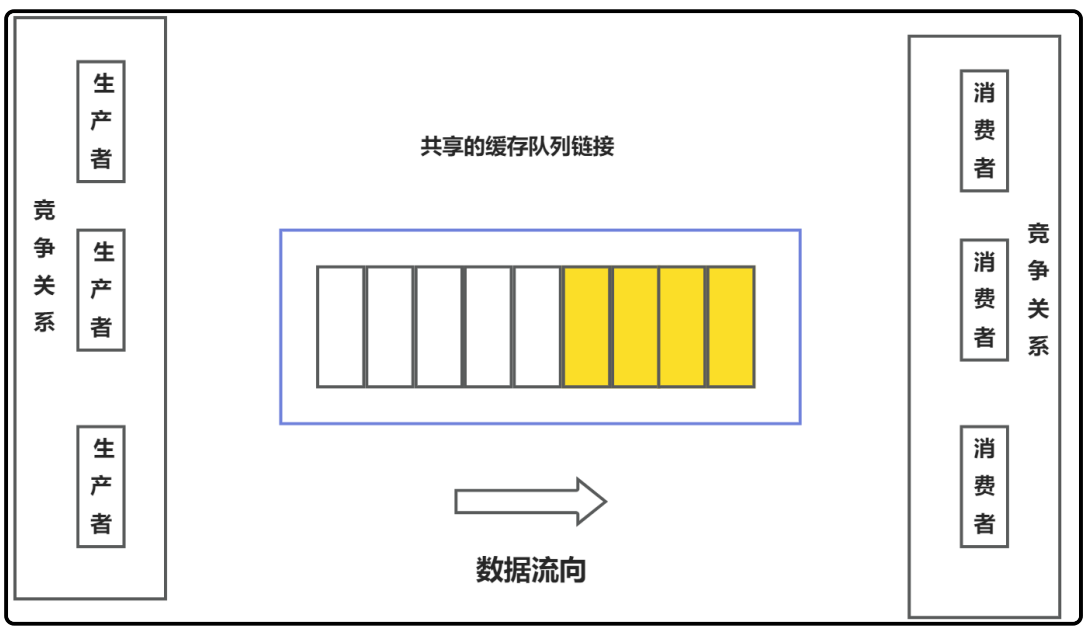
```cpp
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

mutex mtx;  // 创建互斥锁，保护全局变量aa

int aa = 0; // 定义全局变量

// 普通函数，把全局变量aa加1000000次
void func()
{
    for (int ii = 0; ii < 1000000; ii++) {
        lock_guard<mutex> mlock(mtx);
        aa++;
    }
}

int main(void)
{
    //func();
    //func();

    thread t1(func);    // 创建线程t1,把全局变量aa加到1000000次
    thread t2(func);    // 创建线程t2,把全局变量aa加到1000000次

    t1.join();  // 回收t1的资源
    t2.join();  // 回收t2的资源

    cout << "aa= " << aa << endl;
}
```

9. 条件变量

条件变量是一种线程同步机制，当条件不满足时，相关线程被一直阻塞，直到某种条件出现，这些线程才会被唤醒。

为了保护共享资源，**条件变量需要和互斥锁**结合一起使用。

条件变量最常用的就是实现 **生产/消费者模型（高速缓存队列）**

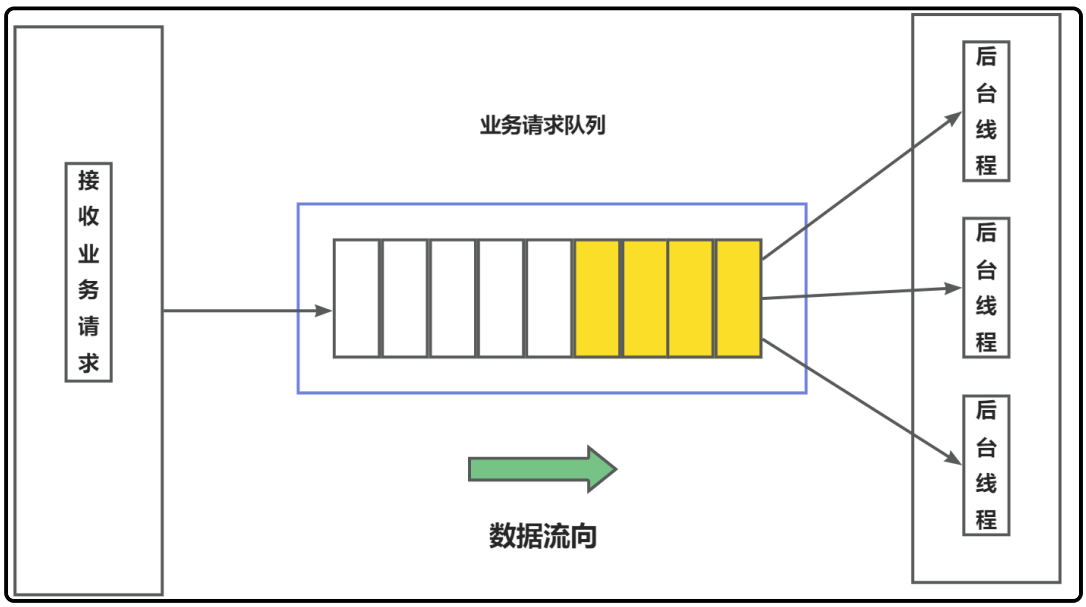


生产者先把需要处理的数据放在缓存队列中，然后向消费者发送通知。消费者接到通知，从缓存队列中把数据拿出来，然后处理它们。

生产者/消费者模型的意义？

例子：某企业售后服务系统，客服收集用户需求，客服不可能直接把问题解决，而是生成工单，然后派发给维修工人。

在这个例子中，客服是数据的生产者，工单是数据的任务队列，维修工人是数据的消费者。



大型网站的后台服务程序有明确的分工，网络通信的线程负责接收客户端的业务请求，然后把业务请求放入队列，后台工作线程负责处理业务请求。**在实际开发中，生产者可以是一个线程，可以是多个线程。而消费者一般是多个线程，这多个线程有一个通俗的名字，叫线程池。**

多个线程共享缓存队列，读写数据的时候可以用互斥锁来保护，当生产者把数据放入缓存队列中后，如何通知消费者线程呢？

```
#include <iostream>
#include <string>
#include <thread>   // 线程类头文件
#include <mutex>    // 互斥锁类的头文件
#include <deque>    // deque容器头文件，双端队列
#include <queue>    // queue容器的头文件，队列
#include <condition_variable>   // 条件变量的头文件
using namespace std;

// 把生产/消费者的实现写在一个类中，这是比较标准的做法
class AA {
private:
    mutex m_mutex;  // 互斥锁
    condition_variable m_cond;  // 条件变量
    queue<string, deque<string>> m_q;   // 缓存队列，底层容器用deque
public:
    //TODO: 生产者线程的任务函数
    void incache(int num)   // 生产数据，num指定数据的个数
    {
        lock_guard<mutex> lock(m_mutex);    // 申请加锁(离开作用域的时候，lock_guard会自动释放锁)
        for (int ii = 0; ii < num; ii++) {
            static int bh = 1;  // 超女编号
            string message = to_string(bh++) + "号超女";   // 拼接出一个数据
            m_q.push(message);  // 把生产出来的数据入队
        }
        // m_cond.notify_one(); //  (给消费者线程发送通知，告诉它们有数据要处理）唤醒一个被当前条件变量阻塞的线程
        m_cond.notify_all();    // 通知所有等待的线程
    }

    // 被条件变量阻塞的消费者线程会形成一个等待队列，轮着来。
    // 但是如果缓存队列中同时有多个数据，那么不应该只让一个消费者线程去处理，合理的做法是让多个线程同时处理，这样的效率才更高。
```

```
    //TODO: 消费者线程的任务函数
    void outcache()
    {
        while (true) {
            string message;
            {
                // 把互斥锁转换为 unique_lock<mutex> ,并申请加锁（离开作用域的时候，unique_lock会自动释放锁）
                unique_lock<mutex> lock(m_mutex);

                // 如果队列中没有数据，这个循环不会中止
                while (m_q.empty()) {   //如果队列为空，进入循环；如果队列不为空，不进入循环，直接处理数据。必须用循环，不能用if
                    m_cond.wait(lock);  // 等待生产者的唤醒信号（通知），如果没有通知，当前线程会一直阻塞
                }

                // 数据元素出队
                message = m_q.front();  // 返回队头的元素
                m_q.pop();  // 再将元素删除（出队）
            //! 离开作用域的时候，unique_lock会自动释放锁
            }

            // 处理出队的数据（把数据消费掉）
            this_thread::sleep_for(chrono::microseconds(1));    // 假设处理数据需要1毫秒
            cout << "线程:" << this_thread::get_id() << ", " << message << endl;
        }
    }
};

//TODO: 使用锁的一个原则：持有锁的时间越短，效率越高
//! 对于消费者线程来说，把数据从队列中拿出来了，就应该立即释放锁。处理数据的时候，已经不需要锁了

int main(void)
{
    AA aa;
    // outcache()成员函数的代码逻辑是死循环，所以，消费者线程创建了，就一直存在，不会退出
    thread t1(&AA::outcache, &aa);  // 创建消费者线程t1
    thread t2(&AA::outcache, &aa);  // 创建消费者线程t2
    thread t3(&AA::outcache, &aa);  // 创建消费者线程t3

    // incache()成员函数只用于生产数据，生产完数据，函数就返回了
    this_thread::sleep_for(chrono::seconds(2)); // 休眠2秒
    aa.incache(3);  //生产3个数据

    this_thread::sleep_for(chrono::seconds(3)); // 休眠3秒
    aa.incache(5);  // 生产5个数据

    t1.join();  // 回收子线程的资源
    t2.join();
    t3.join();
}

// m_cond.notify_one(); 2秒之后，三个消费者线程中的一个线程被唤醒，消费3个数据。5秒之后，另一个线程被唤醒，消费5个数据。因为消费者线程一直再运行，所以程序不会退出
// m_cond.notify_all(); 每生产一批数据后，三个线程都会去竞争它们，如果队列中的数据刚好是三个，那么每个线程抢到一个。
// 如果缓存队列中有5个数据，那么有的线程抢了3个，有的线程抢了1个。

//TODO: 结论：如果生产的数据只有一个，用 notify_one()比较合适；如果生产的数据有多个，用 notify_all()比较合适
```

如果生产的数据只有一个，用 notify_one()比较合适；如果生产的数据有多个，用 notify_all()比较合适

10. 条件变量--生产/消费者模型

C++11 的条件变量提供了**两个类**：

condition_variable： 只支持与普通 mutex搭配，**效率更高**。

condition_variable_any： 是一种通用的条件变量，可以与任意 mutex 搭配，（包含用户自定义的锁类型）

包含头文件： **<condition_variable>**

10.1 condition_variable 类

主要成员函数：

- **condition_variable()**

  > 默认构造函数。

- 2）**condition_variable(const condition_variable &)=delete**

  > 禁止拷贝。

- 3）**condition_variable& condition_variable::operator=(const condition_variable &)=delete**

  > 禁止赋值。

- 4）**notify_one()**

  > 通知一个等待的线程。

- 5）**notify_all()**

  > 通知全部等待的线程。

- 6）**wait(unique_lock lock)**

  > 阻塞当前线程，直到通知到达。(一般用于消费者线程的代码中)

- 7）**wait(unique_lock lock,Pred pred)**

  > 循环的阻塞当前线程，直到通知到达且谓词满足。 （第二个参数是一个谓词）

- 8）**wait_for(unique_lock lock,时间长度)**
- 9）**wait_for(unique_lock lock,时间长度,Pred pred)**
- 10）**wait_until(unique_lock lock,时间点)**
- 11）**wait_until(unique_lock lock,时间点,Pred pred)**

条件变量的wait(mutex)函数

条件变量的wait()函数不只是等待生产者信号这么简单，做了三件事：

> 1. 把互斥锁解锁
> 2. 阻塞，等待被唤醒
> 3. 给互斥锁加锁

```
            cout << "线程: " << this_thread::get_id() << ", 申请加锁..." << endl;
                unique_lock<mutex> lock(m_mutex);   // 申请加锁
                cout << "线程: " << this_thread::get_id() << ", 加锁成功" << endl;
            //TODO: 只有在加锁成功的情况下，才有机会阻塞在条件变量的wait()函数中。
            //TODO: 如果多个消费者线程一起运行，那么，只能有一个线程申请加锁成功，其他消费者线程都会阻塞在申请互斥锁这里

            //? 只有一个线程阻塞在wait()函数中，其他的线程都会阻塞在申请加锁这里
            // 如果队列中没有数据，这个循环不会中止
            //this_thread::sleep_for(chrono::hours(1)); // 让线程休眠1小时
            while (m_q.empty()) {   //如果队列为空，进入循环；如果队列不为空，不进入循环，直接处理数据。必须用循环，不能用if
                m_cond.wait(lock);  // 等待生产者的唤醒信号（通知），如果没有通知，当前线程会一直阻塞
            }
```

1.**因为wait()函数将互斥锁解开了，所以三个消费者线程先后都能加锁成功。**

2.现在wait()函数进行到了第2步， **3个线程都被阻塞到了条件变量的wait()函数中** ， **这个时候互斥锁没有被任何线程持有** 。如果生产者要往缓存队列中放数据，可以加锁成功，生产者往缓存队列中放完数据之后，会发出条件信号。那么这3个线程的wait()函数会接收到条件信号，walt()函数接受到信号后，不一定立即返回。

3.它还需要申请加锁，加锁成功了才会返回。（如果wait()返回了，那么一定申请到了锁）

4.接下来可以让缓存队列中的数据出队，出队后再解锁。

10.2 unique_lock类

template class unique_lock 是模板类，模板参数为**互斥锁类型**。

unique_lock和lock_guard都是管理锁的辅助类，都是RAII风格（在构造时获得锁，在析构时释放锁）。它们的区别在于：为了配合condition_variable， unique_lock还有lock()和unlock()成员函数。

10.2.1 普通的互斥锁，为什么要转换为 unique_lock之后才能用于条件变量？

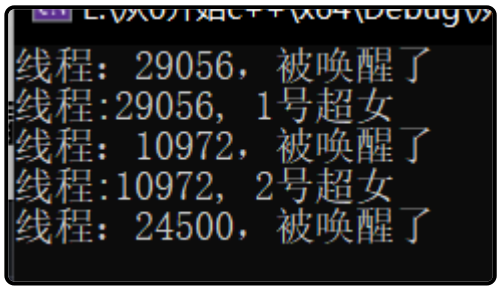
因为在条件变量的wait()函数中，需要解锁和加锁两个功能，lock_guark 类没有这两个成员函数，而unique_lock类有 lock() 和 unlock()这两个成员函数。

10.2.2 条件变量虚假唤醒

```
while (m_q.empty()) {   //如果队列为空，进入循环；如果队列不为空，不进入循环，直接处理数据。必须用循环，不能用if
    m_cond.wait(lock);  // 等待生产者的唤醒信号（通知），如果没有通知，当前线程会一直阻塞
    cout << "线程: " << this_thread::get_id() << ", 被唤醒了" << endl;
}

//main函数中
aa.incache(2);  //生产2个数据
```

条件变量存在虚假唤醒：消费者线程被唤醒后，缓存队列中没有数据。

> 当生产者只生产了2个数据，三个线程都被唤醒了，但是数据只有两个，线程24500并没有拿到数据，对它来说，就是虚假唤醒。



如果消费者线程被虚假唤醒，它应该继续等待下一次通知，所以，这里我们只能用循环，如果用if语句，根本达不到我们想要的效果。

条件变量的wait()函数还有一个重载版本，`wait(unique_lock<mutex> lock,Pred pred)` 第一个参数是unique_lock 第二个参数是 一个谓词。

在这个场景中，谓词用 Lambda函数最简洁。（这个重载版本的wait()函数里面也有一个循环）

m_cond.wait(lock, [this] {return !m_q.empty();});　　等同于 上面的while循环

```
#include <iostream>
#include <string>
#include <thread>   // 线程类头文件
#include <mutex>    // 互斥锁类的头文件
#include <deque>    // deque容器头文件，双端队列
#include <queue>    // queue容器的头文件，队列
#include <condition_variable>   // 条件变量的头文件
using namespace std;

// 把生产/消费者的实现写在一个类中，这是比较标准的做法
class AA {
private:
    mutex m_mutex;  // 互斥锁
    condition_variable m_cond;  // 条件变量
    queue<string, deque<string>> m_q;   // 缓存队列，底层容器用deque
public:
    //TODO: 生产者线程的任务函数
    void incache(int num)   // 生产数据，num指定数据的个数
    {
        lock_guard<mutex> lock(m_mutex);    // 申请加锁(离开作用域的时候，lock_guard会自动释放锁)
        for (int ii = 0; ii < num; ii++) {
            static int bh = 1;  // 超女编号
            string message = to_string(bh++) + "号超女";   // 拼接出一个数据
            m_q.push(message);  // 把生产出来的数据入队
        }
        //m_cond.notify_one();  //  (给消费者线程发送通知，告诉它们有数据要处理）唤醒一个被当前 条件变量阻塞的线程
        m_cond.notify_all();    // 通知所有等待的线程
    }
    // 被条件变量阻塞的消费者线程会形成一个等待队列，轮着来。
    // 但是如果缓存队列中同时有多个数据，那么不应该只让一个消费者线程去处理，合理的做法是让多个线程同时处理，这样的效率才更高。

    //TODO: 消费者线程的任务函数
    void outcache()
    {
        while (true) {
            string message;
            {
                // 把互斥锁转换为 unique_lock<mutex> ,并申请加锁（离开作用域的时候，unique_lock会自动释放锁）
                unique_lock<mutex> lock(m_mutex);   // 申请加锁

                //TODO: 只有在加锁成功的情况下，才有机会阻塞在条件变量的wait()函数中。
                //TODO: 如果多个消费者线程一起运行，那么，只能有一个线程申请加锁成功，其他消费者线程都会阻塞在申请互斥锁这里

                //? 只有一个线程阻塞在wait()函数中，其他的线程都会阻塞在申请加锁这里
                // 如果队列中没有数据，这个循环不会中止

                //while (m_q.empty()) { //如果队列为空，进入循环；如果队列不为空，不进入循环，直接处理数据。必须用循环，不能用if
```

```
                //  m_cond.wait(lock);  // 等待生产者的唤醒信号（通知），如果没有通知，当前线程会一直阻塞
                //}
                m_cond.wait(lock, [this] {return !m_q.empty();} );

                // 数据元素出队
                message = m_q.front();  // 返回对头的元素
                m_q.pop();  // 再将元素删除（出队）
                cout << "线程:" << this_thread::get_id() << ", " << message << endl;
                //! 离开作用域的时候，unique_lock会自动释放锁
                //lock.unlock();    // 手工解锁
            }


            // 处理出队的数据（把数据消费掉）
            this_thread::sleep_for(chrono::microseconds(1));    // 假设处理数据需要1毫秒

        }
    }
};

//TODO: 使用锁的一个原则：持有锁的时间越短，效率越高
//! 对于消费者线程来说，把数据从队列中拿出来了，就应该立即释放锁。处理数据的时候，已经不需要锁了


int main(void)
{
    AA aa;
    // outcache()成员函数的代码逻辑是死循环，所以，消费者线程创建了，就一直存在，不会退出
    thread t1(&AA::outcache, &aa);  // 创建消费者线程t1
    thread t2(&AA::outcache, &aa);  // 创建消费者线程t2
    thread t3(&AA::outcache, &aa);  // 创建消费者线程t3

    // incache()成员函数只用于生产数据，生产完数据，函数就返回了
    this_thread::sleep_for(chrono::seconds(2)); // 休眠2秒
    aa.incache(2);  //生产3个数据

    this_thread::sleep_for(chrono::seconds(3)); // 休眠3秒
    //aa.incache(5);    // 生产5个数据

    t1.join();  // 回收子线程的资源
    t2.join();
    t3.join();
}

// m_cond.notify_one(); 2秒之后，三个消费者线程中的一个线程被唤醒，消费3个数据。5秒之后，另一个线程被唤醒，消费5个数据。因为消费者线程一直再运行，所以程序不会退出
// m_cond.notify_all(); 每生产一批数据后，三个线程都会去竞争它们，如果队列中的数据刚好是三个，那么每个线程抢到一个。
// 如果缓存队列中有5个数据，那么有的线程抢了3个，有的线程抢了1个。

//TODO: 结论：如果生产的数据只有一个，用 notify_one()比较合适；如果生产的数据有多个，用 notify_all()比较合适
```

condition_variable_any这个类创建条件变量，互斥锁用 timed_mutex 带超时机制的互斥锁

```
#include <iostream>
#include <string>
#include <thread>   // 线程类头文件
#include <mutex>    // 互斥锁类的头文件
#include <deque>    // deque容器头文件，双端队列
#include <queue>    // queue容器的头文件，队列
#include <condition_variable>   // 条件变量的头文件
using namespace std;

// 把生产/消费者的实现写在一个类中，这是比较标准的做法
class AA {
private:
    timed_mutex m_mutex;    // 带超时机制的互斥锁
    condition_variable_any m_cond;  // 条件变量
    queue<string, deque<string>> m_q;   // 缓存队列，底层容器用deque
public:
```

```
    //TODO: 生产者线程的任务函数
    void incache(int num)    // 生产数据，num指定数据的个数
    {
        lock_guard<timed_mutex> lock(m_mutex);  // 申请加锁(离开作用域的时候，lock_guard会自动释放锁)
        for (int ii = 0; ii < num; ii++) {
            static int bh = 1;  // 超女编号
            string message = to_string(bh++) + "号超女";    // 拼接出一个数据
            m_q.push(message);  // 把生产出来的数据入队
        }
        //m_cond.notify_one();  //  (给消费者线程发送通知，告诉它们有数据要处理）唤醒一个被当前 条件变量阻塞的线程
        m_cond.notify_all();    // 通知所有等待的线程
    }
    // 被条件变量阻塞的消费者线程会形成一个等待队列，轮着来。
    // 但是如果缓存队列中同时有多个数据，那么不应该只让一个消费者线程去处理，合理的做法是让多个线程同时处理，这样的效率才更高。

    //TODO: 消费者线程的任务函数
    void outcache()
    {
        while (true) {
            string message;
            {
                // 把互斥锁转换为 unique_lock<mutex> ,并申请加锁（离开作用域的时候，unique_lock会自动释放锁）
                unique_lock<timed_mutex> lock(m_mutex); // 申请加锁

                //TODO: 只有在加锁成功的情况下，才有机会阻塞在条件变量的wait()函数中。
                //TODO: 如果多个消费者线程一起运行，那么，只能有一个线程申请加锁成功，其他消费者线程都会阻塞在申请互斥锁这里

                //? 只有一个线程阻塞在wait()函数中，其他的线程都会阻塞在申请加锁这里
                // 如果队列中没有数据，这个循环不会中止

                //while (m_q.empty()) { //如果队列为空，进入循环；如果队列不为空，不进入循环，直接处理数据。必须用循环，不能用if
                //  m_cond.wait(lock);  // 等待生产者的唤醒信号（通知），如果没有通知，当前线程会一直阻塞
                //}
                m_cond.wait(lock, [this] {return !m_q.empty();} );

                // 数据元素出队
                message = m_q.front();  // 返回对头的元素
                m_q.pop();  // 再将元素删除（出队）
                cout << "线程:" << this_thread::get_id() << ", " << message << endl;
                //! 离开作用域的时候，unique_lock会自动释放锁
                //lock.unlock();    // 手工解锁
            }


            // 处理出队的数据（把数据消费掉）
            this_thread::sleep_for(chrono::microseconds(1));    // 假设处理数据需要1毫秒

        }
    }
};

//TODO: 使用锁的一个原则：持有锁的时间越短，效率越高
//! 对于消费者线程来说，把数据从队列中拿出来了，就应该立即释放锁。处理数据的时候，已经不需要锁了


int main(void)
{
    AA aa;
    // outcache()成员函数的代码逻辑是死循环，所以，消费者线程创建了，就一直存在，不会退出
    thread t1(&AA::outcache, &aa);  // 创建消费者线程t1
    thread t2(&AA::outcache, &aa);  // 创建消费者线程t2
    thread t3(&AA::outcache, &aa);  // 创建消费者线程t3

    // incache()成员函数只用于生产数据，生产完数据，函数就返回了
    this_thread::sleep_for(chrono::seconds(2)); // 休眠2秒
    aa.incache(2);  //生产3个数据
```

```
    this_thread::sleep_for(chrono::seconds(3)); // 休眠3秒
    aa.incache(5);  // 生产5个数据

    t1.join();  // 回收子线程的资源
    t2.join();
    t3.join();
}

// m_cond.notify_one(); 2秒之后，三个消费者线程中的一个线程被唤醒，消费3个数据。5秒之后，另一个线程被唤醒，消费5个数据。因为消费者线程一直再运行，所以程序不会退出
// m_cond.notify_all(); 每生产一批数据后，三个线程都会去竞争它们，如果队列中的数据刚好是三个，那么每个线程抢到一个。
// 如果缓存队列中有5个数据，那么有的线程抢了3个，有的线程抢了1个。

//TODO: 结论：如果生产的数据只有一个，用 notify_one()比较合适；如果生产的数据有多个，用 notify_all()比较合适
```

注意：

在C++中，条件表达式在 `while` 或者其他控制流语句中，会被隐式地转换为布尔值，**非零整数**会被视为 `true`，而**0会被视为** `false`。这使得你可以使用类似 `while(表达式)` 来进行循环，

11. 原子类型 atomic

C++ 11提供了 atomic 模板类（结构体），用于支持原子类型，模板参数可以是 bool， char， int， long，long long， 指针类型（不支持浮点类型和自定义数据类型），也就是说，原子类型只支持整型。

原子操作由**CPU指令提供支持，它的性能比锁和消息传递更高**，并且，不需要程序员处理加锁和释放锁的问题，支持修改，读取，交换，比较并交换等操作。

头文件： **#include**

> 互斥锁实现了线程同步的问题，但是，代价比较高，就像交通路口的红绿灯，如果有一个线程持有锁，其他线程就会阻塞等待。
>
> 原子类型就像高架桥，每个方向都可以通行，效率更高。
>
> 原子操作由CPU指令提供支持，是轻量级的锁，不是完全没有锁。（就像汽车通过高架桥一样，虽然不会阻塞，但是，速度也会慢下来）

构造函数：

- **atomic() noexcept = default;**

  > 默认构造函数。

- **atomic(T val) noexcept;**

  > 转换函数。

- **atomic(const atomic&) = delete;**

  > 禁用拷贝构造函数。

赋值函数：

- **atomic& operator=(const atomic&) = delete;**

  > 禁用赋值函数。

常用函数：

- **void store(const T val) noexcept;**

  > 把val的值存入原子变量。

- **T load() noexcept;**

读取原子变量的值。

- **T fetch_add(const T val) noexcept;**

  把原子变量的值与val相加，返回原值。

- **T fetch_sub(const T val) noexcept;**

  把原子变量的值减val，返回原值。

- **T exchange(const T val) noexcept;**

  把val的值存入原子变量，返回原值。

- **T compare_exchange_strong(T &expect,const T val) noexcept;**

  比较原子变量的值和预期值expect，如果当两个值相等，把val存储到原子变量中，函数返回true；如果当两个值不相等，用原子变量的值更新预期值，函数返回false。CAS指令。

- **bool is_lock_free();**

  查询某原子类型的操作是直接用CPU指令（返回true），还是编译器内部的锁（返回false）。

  即 如果编译器不支持原子操作，那么就会用锁来代替，满足c++11标准的要求

原子类型的别名：

原子类型	相关特化类
atomic_bool	std::atomic<bool>
atomic_char	std::atomic<char>
atomic_schar	std::atomic<signed char>
atomic_uchar	std::atomic<unsigned char>
atomic_int	std::atomic<int>
atomic_uint	std::atomic<unsigned>
atomic_short	std::atomic<short>
atomic_ushort	std::atomic<unsigned short>
atomic_long	std::atomic<long>
atomic_ulong	std::atomic<unsigned long>
atomic_llong	std::atomic<long long>
atomic_ullong	std::atomic<unsigned long long>

注意：

- atomic　模板类重载了整数操作的各种运算符
- atomic　模板类的模板参数支持指针，但不表示指针指向的对象是原子类型。
- **原子整型可以用作计数器，布尔型可以用作开关**
- **CAS指令是实现无锁队列的基础**

```
// 用原子操作实现了线程同步，aa=2000000
#include <iostream>
```

```cpp
#include <thread>
#include <mutex>
#include <atomic>
using namespace std;

atomic<int> aa = 0;

void func()
{
    for (int ii = 1; ii <= 1000000; ii++) {
        aa++;
    }
}

int main(void)
{
    thread t1(func);
    thread t2(func);


    t1.join();
    t2.join();
    cout << "aa=" << aa << endl;

}
```

```cpp
#include <iostream>
#include <thread>
#include <atomic>
using namespace std;

int main(void)
{
    //atomic<int> a = 3;    // atomic(T val) noexcept; 转换函数
    //cout << "a=" << a.load() << endl; // 读取原子变量的值
    //a.store(8);   //把8存入到原子变量
    //cout << "a=" << a.load() << endl; // 读取原子变量的值
    //int old;
    //old=a.fetch_add(1);   // 把原子变量的值加1，返回原值
    //cout << "a=" << a.load() <<  "   old=" << old << endl;    // a=9  old =8

    //old = a.fetch_sub(3); // 把原子变量的值减3，返回原值
    //cout << "a=" << a.load() << "   old=" << old << endl; // a=6  old =9

    //old=a.exchange(3);    // 把3存入到原子变量，返回原值
    //cout << "a=" << a.load() << "   old=" << old << endl;

    atomic<int> ii = 3;     // 原子变量
    int expect = 4;     //  期待值
    int val = 5;    // 打算存入原子变量的值

    // 比较原子变量的值和预期值except，如果当两个值相等，把val存入到原子变量中，函数返回true；
    //如果当两个值不相等，用原子变量的值更新预期值，函数返回false。CAS指令
    bool bret = ii.compare_exchange_strong(expect, val);
    cout << "ii=" << ii << "   expect=" << expect << endl;  // ii=5  expect=3
    cout << bret << endl;

}
```