const 表示常量 只读变量

const int a = 10; // a 为常量 const int num; // num是变量,但是只读

struct 结构体的大小 受内存对齐的限制的影响；并且，结构体大小必须是 最大成员的内存大小的整数倍

```
struct Firstchar {
    char c;
    int a;
    int b;
};

struct Firstint {
    int b;
    int a;
    char c;
    char h;
    char i;
    char s;
};
```

```
struct Firstint {
    int b;
    int a;
    char c;
};
```

大小都是 12

委托构造 / 子类继承父类的 构造函数 1、Child(): Father() 2、using Father::Father();

移动构造函数

```
// 添加移动构造函数
Test(Test&& a) : m_num(a.m_num)
{
    a.m_num = nullptr;
    cout << "move construct: my name is sunny" << endl;
}
```

这样 a被创造后，不会仅仅 构造出 对象后，就被析构了

类中

const成员 声明时需要初始化

static成员变量： 基本数据类型的，可以在类内初始化; 其它自定义类型的，需要在类外初始化

POD plain old data

inline 内联函数 用于对编译器的建议

不再调用函数，而是直接把函数展开，（不会频繁开辟栈空间了）但是，只有 编译器认为 此函数较短适合展开时，才会这样去做！！

子类不可以继承 父类的友元关系

子类实现继承，是继承 父类某个对象 而 友元关系涉及到的函数 与父类对象其实没有多大关系

C++11 friend可以用作 模板类

联合体

受限联合体： 之前的 union 不能拥有 非POD 的数据 不能有静态成员 不能拥有引用类型的成员

非受限联合体：只要没有引用类型的成员即可

定位放置 new

可在 栈或堆上放置 对象 关键看 定位的内存地址 是处于栈上还是堆上。

move

```
// 使用智能指针管理一块 int 型的堆内存, 内部引用计数为 1
shared_ptr<int> ptr1(new int(520));
cout << "ptr1管理的内存引用计数: " << ptr1.use_count() << endl;
//调用拷贝构造函数
shared_ptr<int> ptr2(ptr1);
cout << "ptr2管理的内存引用计数: " << ptr2.use_count() << endl;
shared_ptr<int> ptr3 = ptr1;
cout << "ptr3管理的内存引用计数: " << ptr3.use_count() << endl;
//调用移动构造函数
shared_ptr<int> ptr4(std::move(ptr1));
cout << "ptr4管理的内存引用计数: " << ptr4.use_count() << endl;
std::shared_ptr<int> ptr5 = std::move(ptr2);
cout << "ptr5管理的内存引用计数: " << ptr5.use_count() << endl;

cout << *ptr1 << endl;
```

这里转让 指针之后，ptr1被销毁。（因为要保证 内存地址的引用次数 -1）

所以 cout *ptr1 崩溃

int num=3； int b=move(num); num可正常被操作

相当于 普通的复制变量 而且 num后序改变是否影响 b，还看 b复制的是否是 a的内存

```
int mm = 3;
int b = move(mm);

cout << mm << endl;
mm = 5;

cout << mm << endl;
cout << b << endl;
```

结果：5 3

```
int mm = 3;
int *b = move(&mm);

cout << mm << endl;
mm = 5;

cout << mm << endl;
cout << *b << endl;
```

结果：5 5

shared_ptr

1、普通构造函数 2、make_shared 3、p.reset() 重新设置 管理的内存

shared_ptrp(); 如果不分配管理的内存，则 count的结果为0

1. `ptr142.reset(&mm);`:
   - 你将指向栈变量 `mm` 的指针传递给 `shared_ptr`，但是 `shared_ptr` 设计用于管理堆内存（通过 `new` 或 `make_shared` 创建的对象）。当 `shared_ptr` 被销毁或 `reset` 时，它会自动调用 `delete` 来释放它所管理的内存。
   - 因为 `mm` 是在栈上分配的，不能用 `delete` 来释放，所以 `shared_ptr` 在其析构或 `reset` 时试图释放 `mm`，导致崩溃。

reset之后，原管理的内存 会被释放

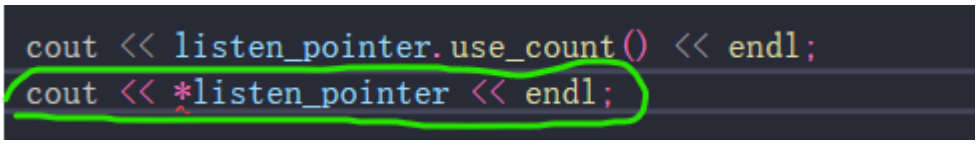
```
int* pp = new int(10);
shared_ptr<int>ptr142;
ptr142.reset(pp);


mm = 4;
cout << mm << endl;
//*ptr142 = 401;
cout << mm << endl;
cout << *ptr142 << endl;

*pp = 20;
cout << *ptr142 << endl;
cout << ptr142.get() << endl;
cout << pp << endl;
```

智能指针 要用来管理 堆上的数据内存

unique_ptr

1、 直接构造 2、不允许拷贝，复制 3、但是允许 return unique_ptr； unique_ptr2=move(unique_ptr1); 或者 reset（可以保证 count《1）

weak_ptr 没有 -> * 重载，不能用来 对内存进行 什么特别的操作，只是用来 观察内存是否被引用的



count 引用计数 expired 是否被释放

lock 获取对象

reset 不监测资源对象

return shared<class T>(this); this 会被析构两次（被引用两次，但是只有一个对象呐） 所以，出错

所以可以用 shared_from_this() : weak_ptr 监听+ lock去返回 shared_ptr

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

struct Test : public enable_shared_from_this<Test>
{
    shared_ptr<Test> getSharedPtr()
    {
        return shared_from_this();
    }
    ~Test()
    {
        cout << "class Test is disstruct ..." << endl;
    }
};

int main()
{
    shared_ptr<Test> sp1(new Test);
    cout << "use_count: " << sp1.use_count() << endl;
    shared_ptr<Test> sp2 = sp1->getSharedPtr();
    cout << "use_count: " << sp1.use_count() << endl;
    return 0;
}
```

```cpp
#include <memory>
using namespace std;

struct TA;
struct TB;

struct TA
{
    weak_ptr<TB> bptr;
    ~TA()
    {
        cout << "class TA is disstruct ..." << endl;
    }
};

struct TB
{
    shared_ptr<TA> aptr;
    ~TB()
    {
        cout << "class TB is disstruct ..." << endl;
```

```
    }
};

void testPtr()
{
    shared_ptr<TA> ap(new TA);
    shared_ptr<TB> bp(new TB);
    cout << "TA object use_count: " << ap.use_count() << endl;
    cout << "TB object use_count: " << bp.use_count() << endl;

    ap->bptr = bp;
    bp->aptr = ap;
    cout << "TA object use_count: " << ap.use_count() << endl;
    cout << "TB object use_count: " << bp.use_count() << endl;
}
```

在测试程序中，共享智能指针 `ap` 、 `bp` 对 `TA` 、 `TB` 实例对象的引用计数变为2， 在共享智能指针离开作用域之后引用计数只能减为1 ，这种情况下不会去删除智能指针管理的内存，导致类 `TA` 、 `TB` 的实例对象不能被析构，最终造成内存泄露。通过使用 `weak_ptr` 可以解决这个问题，只要将类 `TA` 或者 `TB` 的任意一个成员改为 `weak_ptr` ，修改之后的代码如下：

thread

t1.detach(); t1.joinable(); joinable() 线程是否被分离

hardware_concurrency() 获取 CPU核心数

3. 静态函数

`thread` 线程类还提供了一个静态方法，用于获取当前计算机的CPU核心数，根据这个结果在程序中创建出数量相等的线程， 每个线程独自占有一个CPU核心，这些线程就不用分时复用CPU时间片，此时程序的并发效率是最高的 。

C++

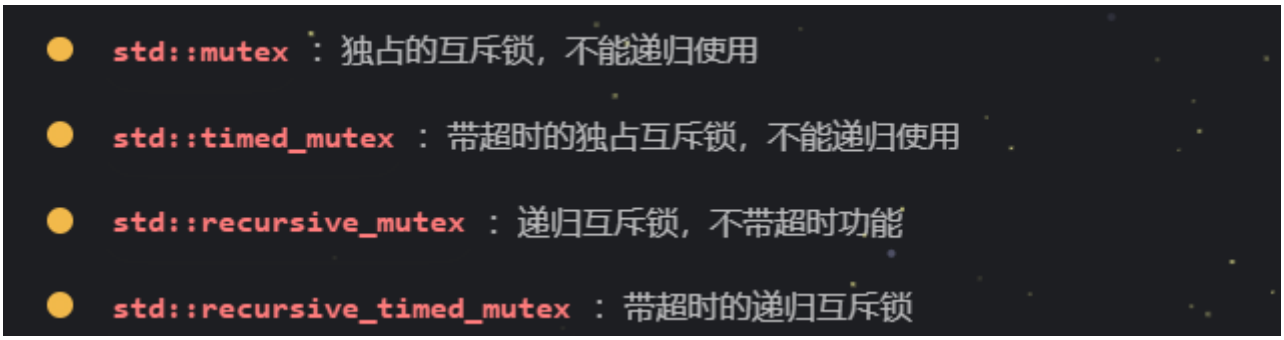
```
1 static unsigned hardware_concurrency() noexcept;
```

call_once 函数只被调用一次（多线程里 只调用一次）

this_thread

yield(); 让出CPU

mutex



递归锁：进入递归后，把锁的使用给下次进入函数时的自己

lock_guard 里面的锁 可以多样

```
    lock_guard<recursive_mutex> locker(m_mutex);
    m_i *= x;
}

void div(int x)
{
    lock_guard<recursive_mutex> locker(m_mutex);
    m_i /= x;
}

void both(int x, int y)
{
    lock_guard<recursive_mutex> locker(m_mutex);
```

超时锁

```cpp
timed_mutex g_mutex;

void work()
{
    chrono::seconds timeout(1);
    while (true)
    {
        // 通过阻塞一定的时长来争取得到互斥锁所有权
        if (g_mutex.try_lock_for(timeout))
        {
            cout << "当前线程ID: " << this_thread::get_id()
                << ", 得到互斥锁所有权..." << endl;
            // 模拟处理任务用了一定的时长
            this_thread::sleep_for(chrono::seconds(10));
            // 互斥锁解锁
            g_mutex.unlock();
            break;
        }
        else
        {
            cout << "当前线程ID: " << this_thread::get_id()
                << ", 没有得到互斥锁所有权..." << endl;
            // 模拟处理其他任务用了一定的时长
            this_thread::sleep_for(chrono::milliseconds(50));
        }
    }
```

condition_variable 只接收 unique_lock 锁

condition_variable_any 可以接收 只要带有 lock unlock方法的任意锁

- `condition_variable` ：需要配合 `std::unique_lock<std::mutex>` 进行wait操作，也就是阻塞线程的操作。
- `condition_variable_any` ：可以和任意带有 `lock()`、`unlock()` 语义的mutex搭配使用，也就是说有四种：
  - `std::mutex` ：独占的非递归互斥锁
  - `std::timed_mutex` ：带超时的独占非递归互斥锁
  - `std::recursive_mutex` ：不带超时功能的递归互斥锁
  - `std::recursive_timed_mutex` ：带超时的递归互斥锁

condition_variable(unique_lock)

condition_variable_any 1/wait() 2.wait(lock,pre) pre “是否继续执行的判定” 谓词， 返回true 则继续执行；false：阻塞

condition_variable_any 和 锁的生命周期绑定

总结：以上介绍的两种条件变量各自有各自的特点，condition_variable 配合 unique_lock 使用更灵活一些，可以在在任何时候自由地释放互斥锁，而condition_variable_any 如果和lock_guard 一起使用必须要等到其生命周期结束才能将互斥锁释放。但是，condition_variable_any 可以和多种互斥锁配合使用，应用场景也更广，而 condition_variable 只能和独占的非递归互斥锁（mutex）配合使用，有一定的局限性。

atomic

原子指的是一系列不可被CPU上下文交换的机器指令，这些指令组合在一起就形成了原子操作。在多核CPU下，当某个CPU核心开始运行原子操作时，会先暂停其它CPU内核对内存的操作，以保证原子操作不会被其它CPU内核所干扰。

由于原子操作是通过指令提供的支持，因此它的性能相比锁和消息传递会好很多。相比较于锁而言，原子类型不需要开发者处理加锁和释放锁的问题，同时支持修改，读取等操作，还具备较高的并发性能，几乎所有的语言都支持原子类型。

可以看出原子类型是无锁类型，但是无锁不代表无需等待，因为原子类型内部使用了 CAS 循环，当大量的冲突发生时，该等待还是得等待！但是总归比锁要好。

C++11内置了整形的原子变量，这样就可以更方便的使用原子变量了。在多线程操作中，使用原子变量之后就不需要再使用互斥量来保护该变量了，用起来更简洁。因为对原子变量进行的操作只能是一个原子操作（ atomic operation ）， 原子操作指的是不会被线程调度机制打断的操作，这种操作一旦开始，就一直运行到结束，中间不会有任何的上下文切换。 多线程同时访问共享资源造成数据混乱的原因就是因为CPU的上下文切换导致的，使用原子变量解决了这个问题，因此互斥锁的使用也就不再需要了。

原子地以 **desired** 替换当前值。按照 **order** 的值影响内存。

```C++
void store( T desired, std::memory_order order = std::memory_order_seq_cst ) noexcept;
void store( T desired, std::memory_order order = std::memory_order_seq_cst ) volatile noexcept;
```

- **desired**：存储到原子变量中的值
- **order**：强制的内存顺序

原子地加载并返回原子变量的当前值。按照 **order** 的值影响内存。直接访问原子对象也可以得到原子变量的当前值。

```C++
T load( std::memory_order order = std::memory_order_seq_cst ) const noexcept;
T load( std::memory_order order = std::memory_order_seq_cst ) const volatile noexcept;
```

- 以上各个 operator 都会有对应的 **fetch_*** 操作，详细见下表：

操作符	操作符重载函数	等级的成员函数	整形	指针	其他
+	atomic::operator+=	atomic::fetch_add	是	是	否
-	atomic::operator-=	atomic::fetch_sub	是	是	否
&	atomic::operator&=	atomic::fetch_and	是	否	否
|	atomic::operator|=	atomic::fetch_or	是	否	否
^	atomic::operator^=	atomic::fetch_xor	是	否	否

调用 atomic API方法时，会有内存顺序约束

- **memory_order_relaxed** ，这是最宽松的规则，它对编译器和CPU不做任何限制，可以乱序
- **memory_order_release** **释放**，设定内存屏障(Memory barrier)，保证它之前的操作永远在它之前，但是它后面的操作可能被重排到它前面
- **memory_order_acquire** **获取**，设定内存屏障，保证在它之后的访问永远在它之后，但是它之前的操作却有可能被重排到它后面，往往和 **Release** 在不同线程中联合使用
- **memory_order_consume** ：改进版的 **memory_order_acquire** ，开销更小
- **memory_order_acq_rel** ，它是 **Acquire** 和 **Release** 的结合，同时拥有它们俩提供的保证。比如你要对一个 **atomic** 自增 1，同时希望该操作之前和之后的读取或写入操作不会被重新排序
- **memory_order_seq_cst** **顺序一致性**， **memory_order_seq_cst** 就像是 **memory_order_acq_rel** 的加强版，它不管原子操作是属于读取还是写入的操作，只要某个线程有用到 **memory_order_seq_cst** 的原子操作，线程中该 **memory_order_seq_cst** 操作前的数据操作绝对不会被重新排在该 **memory_order_seq_cst** 操作之后，且该 **memory_order_seq_cst** 操作后的数据操作也绝对不会被重新排在 **memory_order_seq_cst** 操作前。

release屏障 挡前面 acquire屏障 挡后面

2. `m_value.fetch_add(1)`

- **操作类型**: `fetch_add(1)` 是 `atomic` 类型的成员函数，它提供明确的原子加操作。这种方法是对变量的底层操作，保证在多线程环境中不需要加锁，仍能保证操作的原子性。
- **返回值**: 返回的是加之前的值，与 `m_value++` 一致。
- **优势**: 作为一个显式 API，`fetch_add` 可以接受第二个参数，用来控制内存序（memory ordering）。例如:

```cpp
m_value.fetch_add(1, std::memory_order_relaxed);
```

这种情况下，你可以指定内存顺序模型，以更灵活地控制多线程访问的内存可见性。

区别总结:

1. **语法差异**: `m_value++` 是操作符，语法上更简洁，而 `fetch_add(1)` 是成员函数调用，更加明确。
2. **返回值**: 两者在返回值上表现一致，都是返回操作前的值。
3. **内存顺序控制**: `fetch_add(1)` 提供了更多的控制选项，可以指定内存序（如 relaxed, acquire, release 等），而 `m_value++` 只能使用默认的内存序。
4. **偏好**: 在一般情况下，使用 `m_value++` 更加简洁；但在复杂的多线程编程场景中，`fetch_add` 提供了更细粒度的控制。

其实，`m_value++` 对于 `std::atomic<int>` 类型的变量来说，是一个 **原子操作**。你不需要加锁，它已经在多线程环境下是线程安全的。

解释:

当你使用 `std::atomic` 类型的变量时，如 `std::atomic<int> m_value`，编译器会将 `m_value++` 翻译成原子操作。因此，哪怕是在多线程的场景下，`m_value++` 也是线程安全的，不会出现数据竞争的情况。

原因:

`std::atomic` 提供了对其操作的原子性保证。即使是使用操作符（如 `++` 或 `--`），`std::atomic` 类型变量也能确保操作是不可中断的。`m_value++` 实际上等价于调用 `m_value.fetch_add(1)`，并保证操作的原子性。

对比:

1. **普通类型（如 int）**： 如果你对普通的 `int` 变量进行 `++` 操作，在多线程中可能会出现数据竞争的情况，因为 `++` 不是一个原子操作。它实际上由三步构成:
   - 读取当前值
   - 自增
   - 将自增后的值写回

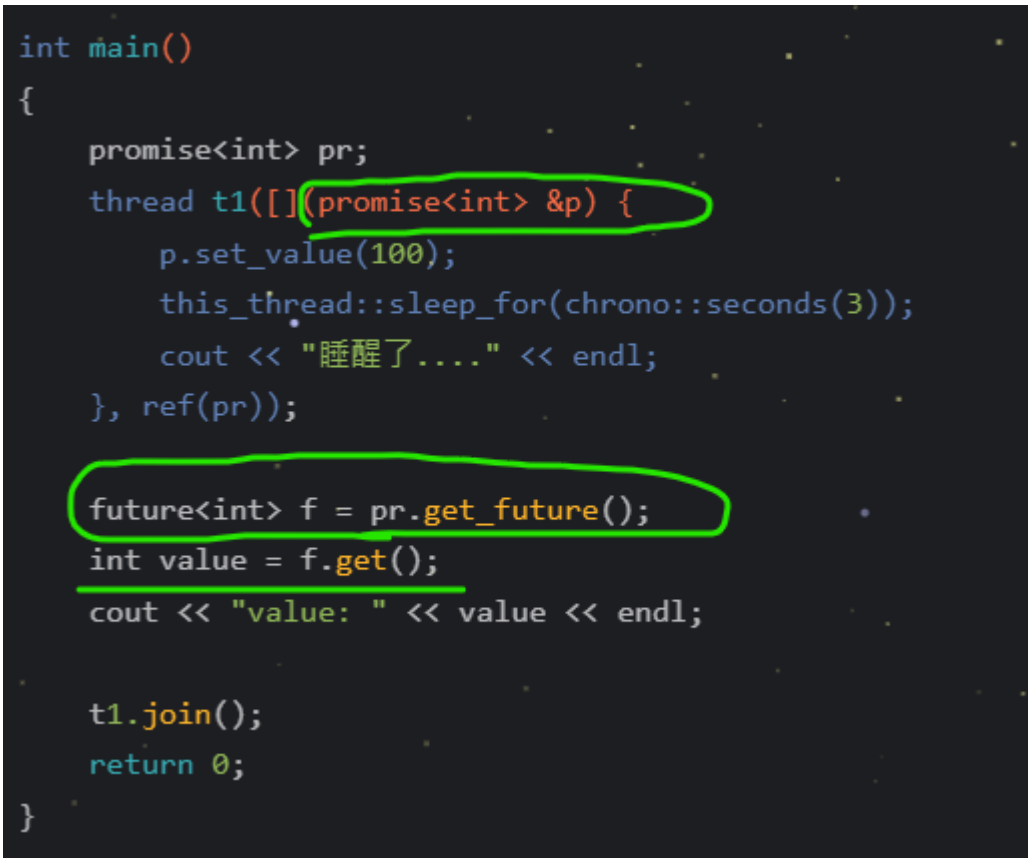
load

store

线程异步：

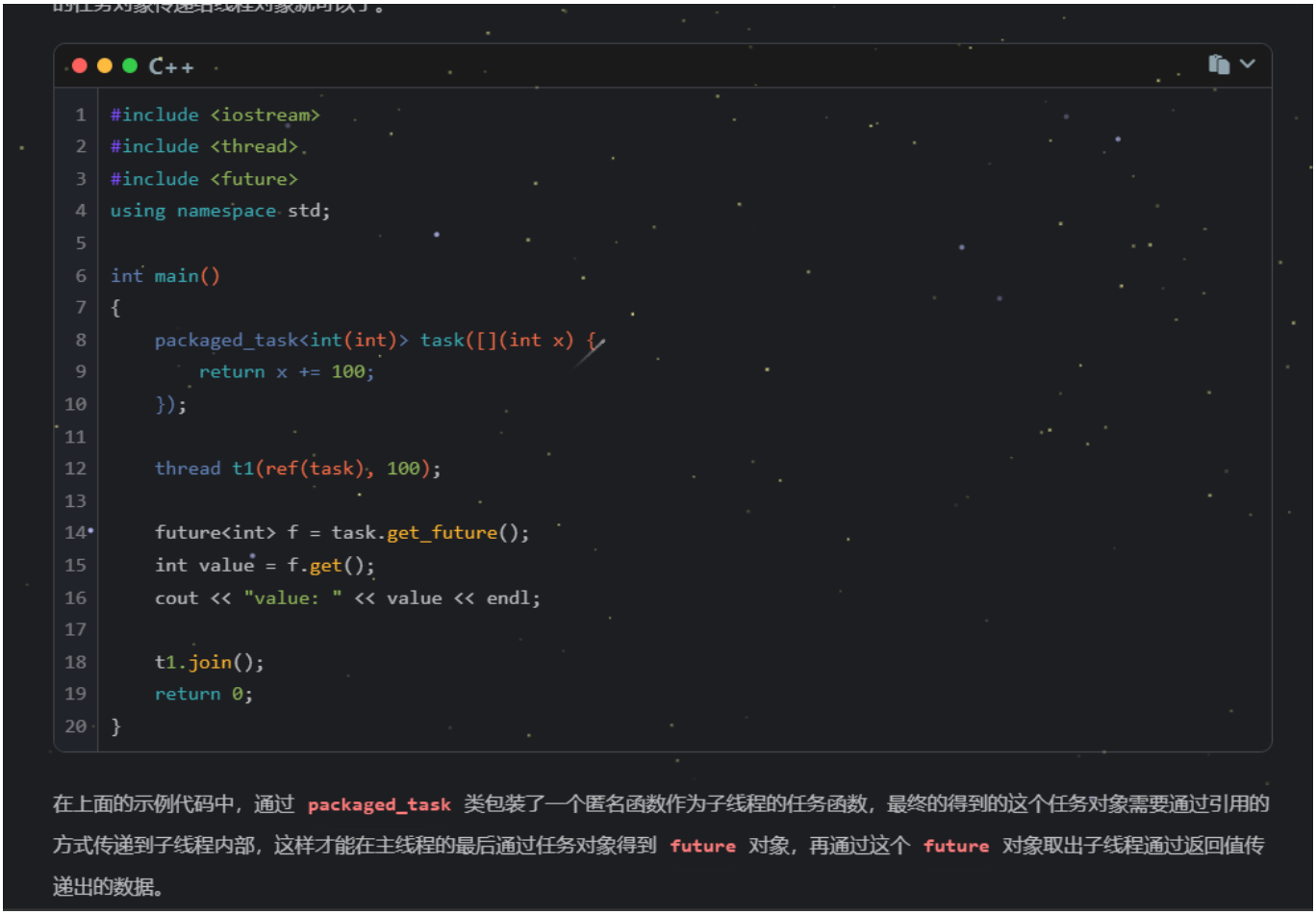
future：getfuture setfuture wait waitfor

promise 传入一个 promise值



```
int main()
{
    promise<int> pr;
    thread t1([](promise<int> &p) {
        p.set_value(100);
        this_thread::sleep_for(chrono::seconds(3));
        cout << "睡醒了...." << endl;
    }, ref(pr));

    future<int> f = pr.get_future();
    int value = f.get();
    cout << "value: " << value << endl;

    t1.join();
    return 0;
}
```

packet_task 包装一个 异步函数;函数 的 返回值 可以是 future的 value



```C++
#include <iostream>
#include <thread>
#include <future>
using namespace std;

int main()
{
    packaged_task<int(int)> task([](int x) {
        return x += 100;
    });

    thread t1(ref(task), 100);

    future<int> f = task.get_future();
    int value = f.get();
    cout << "value: " << value << endl;

    t1.join();
    return 0;
}
```

在上面的示例代码中，通过 **packaged_task** 类包装了一个匿名函数作为子线程的任务函数，最终的得到的这个任务对象需要通过引用的方式传递到子线程内部，这样才能在主线程的最后通过任务对象得到 **future** 对象，再通过这个 **future** 对象取出子线程通过返回值传递出的数据。

async通过返回值 是 future的方式，进程异步

```
cout << "主线程ID: " << this_thread::get_id() << endl;
// 调用函数直接创建线程执行任务
future<int> f = async([](int x) {
    cout << "子线程ID: " << this_thread::get_id() << endl;
    this_thread::sleep_for(chrono::seconds(5));
    return x += 100;
}, 100);

future_status status;
do {
    status = f.wait_for(chrono::seconds(1));
    if (status == future_status::deferred)
    {
        cout << "线程还没有执行..." << endl;
        f.wait();
    }
    else if (status == future_status::ready)
    {
        cout << "子线程返回值: " << f.get() << endl;
    }
    else if (status == future_status::timeout)
    {
        cout << "任务还未执行完毕，继续等待..." << endl;
    }
} while (status != future_status::ready);

return 0;
```

由于指定了 `launch::deferred` 策略，因此调用 `async()` 函数并不会创建新的线程执行任务，当使用 `future` 类对象调用了 `get()` 或者 `wait()` 方法后才开始执行任务（此处一定要注意调用wait_for()函数是不行的）。

通过测试程序输出的结果可以看到，两次输出的线程ID是相同的，任务函数是在主线程中被延迟（主线程休眠了5秒）调用了。

最终总结：

1. 使用async()函数，是多线程操作中最简单的一种方式，不需要自己创建线程对象，并且可以得到子线程函数的返回值。
2. 使用std::promise类，在子线程中可以传出返回值也可以传出其他数据，并且可选择在什么时机将数据从子线程中传递出来，使用起来更灵活。
3. 使用std::packaged_task类，可以将子线程的任务函数进行包装，并且可以得到子线程的返回值。

epoll 同步?

cfd=accept(); int n=epoll();