C++线程池

自定义一个线程池类，使用有参构造函数实现线程的创建和任务的执行。

析构函数中，设置stop=Ture，并且通知所有线程。

1.万能引用：

模板 &&。发生类型推导，推导规则：引用折叠。

```
void foo(int &&i) {}     // i为右值引用
template<class T>
void bar(T &&t) {}          // t为万能引用
int get_val() { return 5; }
int &&x = get_val();        // x为右值引用
auto &&y = get_val();       // y为万能引用
```

类模板型	T 实际类型	最终类型
T&	R	R&
T&	R&	R&
T&	R&&	R&
T&&	R	R&&
T&&	R&	R&
T&&	R&&	R&&

2.完美转发：

传入的参数以前是左值或者右值，经过转发，再次传入也是原来的类型。

```
#include <iostream>
#include <string>
template<class T>
void show_type(T t)
{
  std::cout << typeid(t).name() << std::endl;
}
template<class T>
void perfect_forwarding(T &&t)
{
  show_type(static_cast<T&&>(t));
}
std::string get_string()
{
  return "hi world";
}
int main()
{
  std::string s = "hello world";
  perfect_forwarding(s);
  perfect_forwarding(get_string());
}
```

标准库提供了函数模板：

```
template<class T>
void perfect_forwarding(T &&t)
{
  show_type(std::forward<T>(t));
}
```

请注意std::move和std::forward的区别，其中std::move一定会将实参转换为一个右值引用，并且使用std::move不需要指定模板实参，模板实参是由函数调用推导出来的。而std::forward会根据左值和右值的实际情况进行转发，在使用的时候需要指定模板实参。

3.bind机制：

绑定，函数与参数绑定在一起，形成新的对象.

```
void add(int x, int y ,int z){}
auto  a = std::bind(add , 1, 2, 3);
a();
```

4.bind与完美转发结合使用：

避免了不必要的拷贝和转换，同时也保持了类型的准确性。接受各种**类型的函数对象和参数**，并正确地将它们绑定在一起。

```
template<class F, class... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args)
{
   return std::bind(std::forward<F>(f), std::forward<Args>(args)...);
}

/ 调用 enqueue 传递一个普通函数指针和参数
int add(int a, int b) { return a + b; }
ThreadPool pool;
auto future1 = pool.enqueue(add, 3, 5);

// 调用 enqueue 传递一个 lambda 表达式和参数
ThreadPool pool;
auto future2 = pool.enqueue([](int x, int y) { return x * y; }, 4, 6);

// 定义一个函数对象类
struct MyFunctionObject {
    int operator()(int x, int y) { return x - y; }
};
ThreadPool pool;
MyFunctionObject myFuncObj;
auto future3 = pool.enqueue(myFuncObj, 8, 2);
```

5.furture:

提供了一种等待异步任务完成并获取结果的方法。

`std::packaged_task<int()>` 是一个模板类，**它用于包装一个可调用对象（函数或者函数对象）**，并且可以异步执行这个可调用对象.

可以使用 `std::packaged_task<int()>` 来封装一个函数或者 lambda 表达式.

```
#include <iostream>
#include <future>
#include <functional>

int add(int a, int b) {
    return a + b;
}

int main() {
    std::packaged_task<int()> task(std::bind(add, 3, 5));
    std::future<int> result = task.get_future();

    // 启动异步任务
    std::thread th(std::move(task));
    th.join();

    // 获取异步任务的结果
    int sum = result.get();
    std::cout << "Sum: " << sum << std::endl;

    return 0;
}
```

使用 `std::future` 时，可以通过 `std::future::get()` 方法获取异步操作的结果。这个方法会阻塞当前线程，直到异步任务完成并返回结果。如果异步任务尚未完成，`get()` 方法将等待直到任务完成并返回结果。

```
std::packaged_task //是一个可调用对象，可以将函数和其参数打包成一个异步任务。
std::function<void()>: //表示一个可调用对象（函数、函数对象、lambda 表达式等）的类型，它可以保存任意可调用对象，并提供了一种统一的方式来调用这些对象。

auto task = std::make_shared< std::packaged_task<int()> >//即使 int() 没有参数，std::bind 仍然可以绑定带有参数的函数，
(
    std::bind(std::forward<F>(f), std::forward<Args>(args)...)
);//将绑定后的 std::packaged_task 对象放入智能指针 task 中，以便能够在将来的某个时间点执行该任务。
```

```
using result_type = typename result_of<F(Args...)>::type;//获取函数的返回值
future<result_type> res = task->get_future();//异步对象，返回的类型是future<函数返回值类型>
res.get()//等待task()执行，get获取。
```