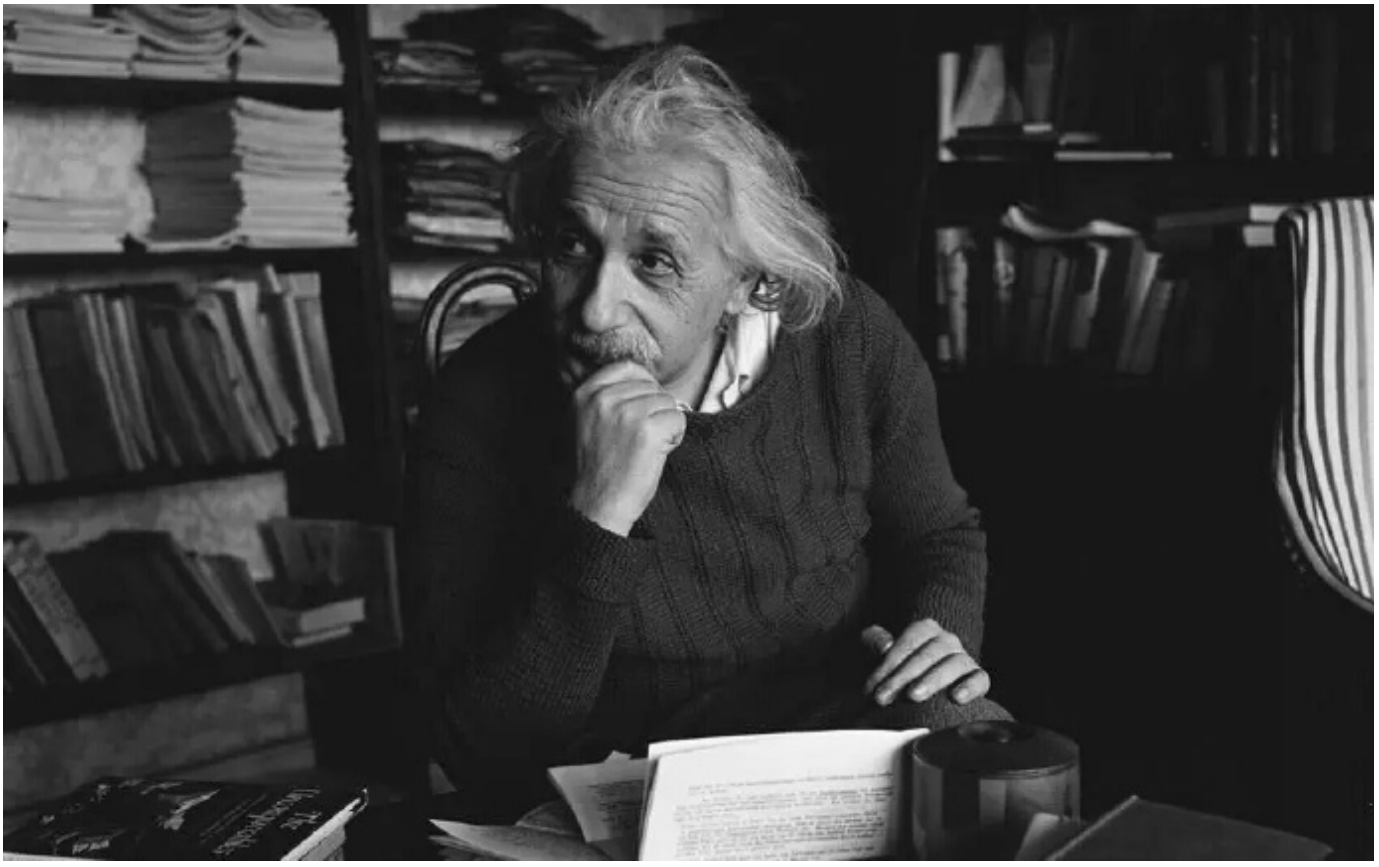
C++ 线程池实现原理 - 文章 - 伯乐在线



背景

多线程编程是C++开发者的一个基本功， 但是很多开发者都是直接使用公司给包装好的线程池库， 没有去了解具体实现，有些实现也都因为高度优化而写得讳莫如深，让初学者看得吃力。

所以写这篇文章主要是想以非常简单的方式讲讲实现原理， 希望初学者看完之后不是觉得「不明觉厉」，而是觉得「原来如此」。

面朝代码

首先先来一段超级简单（注释丰富）的代码展示多线程编程的经典写法。

> 注: 该段代码和完整运行示例请见 limonp-thread-pool-programming-example ， 可以通过以下命令跑通示例代码和输出结果，建议尝试以下。当然记得前提是该机器的网络可以访问 GitHub 的情况下。 可能因为在中国访问 GitHub 并不是很稳定，不是每次都能成功，所以如果卡住的话就多make几次。

```
1
2
3
git clone https://github.com/yanyiwu/practice
cd practice/cpp/limonp-v0.5.1-demo
```

```
make
```

```
#include "limonp/ThreadPool.hpp"#include "limonp/StdExtension.hpp"
using namespace std;
const size_t THREAD_NUM = 4;
// 这个类本身没什么实际含义，只是为了示例多线程编程而写出来的而已。
class Foo {
public:
    // 这个类成员函数Append会在多个线程中被调用，多个线程同时对 chars 这个类成员变量进行写操作，所以需要加锁保证线程安全。
    void Append(char c) {
            limonp::MutexLockGuard lock(mutex_);
            chars.push_back(c);
    }
    string chars; // 多线程共享的对象
    limonp::MutexLock mutex_; // 线程锁
};
void DemoClassFunction() {
    Foo foo;
    cout << foo.chars << endl;
    // 初始化一个线程池。
    limonp::ThreadPool thread_pool(THREAD_NUM);
    thread_pool.Start(); // 启动线程池
    for (size_t i = 0; i < 20; i++) {
            char c = i % 10 + '0';
            // 使用 NewClosure 绑定 foo 对象和 Append 函数和对应参数，构造一个闭包扔进线程池中运行，关于这个 NewClosure 后面会讲。
            thread_pool.Add(limonp::NewClosure(&foo, &Foo::Append, c));
    }
    thread_pool.Stop(); // 等待所有线程工作（工作是指NewClosure生成的闭包函数）都完成，然后停止所有线程。
    cout << foo.chars << endl;
}
int main() {
    DemoClassFunction();
    return 0;
}
```

上面代码注释已经非常详细，一路看下来可能最困惑的应该是 NewClosure 闭包创建函数。 很多人谈到闭包就觉得是一个很高深或者是坑很深的问题，但是此闭包非JavaScript的闭包。 是一个简单版本，只考虑值拷贝，不考虑引用类型的参数。

```
1
limonp::NewClosure(&foo, &Foo::Append, c)
```

所以在该代码中三个参数，第一个是类对象的指针，第二个是成员函数的指针，第三个是int，都是值传递。 在此不支持引用传递的参数。

简单闭包实现

假设你现在已经运行了刚才的这三行命令。

```
1
2
3
git clone https://github.com/yanyiwu/practice
cd practice/cpp/limonp-v0.5.1-demo
make
```

一切运行正常的话，应该目前所在的目录里面有 limonp-0.5.1 这个目录。 然后可以在

```
1
limonp-0.5.1/include/limonp/Closure.hpp
```

文件中看到 NewClosure 函数的定义，NewClosure 是一个模板函数，所以才能支持多种多样的类型。但是也让代码变得晦涩很多。

但是 NewClosure 的实现原理最主要的是生成一个满足 ClosureInterface 接口定义的对象而已。

同时这个 ClosureInterface 也异常的简单，如下：

```
1
2
3
4
5
6
class ClosureInterface {
public:
    virtual ~ClosureInterface() {
        }
    virtual void Run() = 0;
};
```

因为当这个 Closure 对象被扔进线程池之后，其实是进入了一个队列中。 然后线程池的多个线程去从队列中获取Closure指针，然后调用该 Run() 函数。

注意到，NewClosure 其实是生成一个对象指针，我们只是构造了这个对象，但是没有销毁它？ 是否会造成内存泄露？ 显然不会，原因是在线程池中，调用完 Closure 中的 Run() 函数之后， 会 delete 该指针，所以不会造成内存泄露。

到这里只剩下一个最关键的困惑点就是 Run() 的具体实现。 也就是如何通过如下代码将 Append 这个类成员函数变成一个实现了 Run() 函数的 Closure 对象？

```
1
limonp::NewClosure(&foo, &Foo::Append, c)
```

因为 NewClosure 是模板函数，支持各种参数类型，但是对于本文的例子，实际上调用的 NewClosure 函数实现如下：

```
1
2
3
4
template<class R, class Obj, class Arg1>
ClosureInterface* NewClosure(Obj* obj, R (Obj::* fun)(Arg1), Arg1 arg1) {
    return new ObjClosure1<Obj, R (Obj::* )(Arg1), Arg1>(obj, fun, arg1);
}
```

所以实际上创建的对象是 ObjClosure1 ，依然是在源码

```
1
limonp-0.5.1/include/limonp/Closure.hpp
```

中可以找到 ObjClosure1 的实现，如下代码：

```
C++
template <class Obj, class Funct, class Arg1>
class ObjClosure1: public ClosureInterface {
public:
    ObjClosure1(Obj* p, Funct fun, Arg1 arg1) {
         p_ = p;
         fun_ = fun;
         arg1_ = arg1;
        }
    virtual ~ObjClosure1() {
        }
    virtual void Run() {
            (p_->*fun_)(arg1_);
        }
private:
    Obj* p_;
    Funct fun_;
    Arg1 arg1_;
};
```

真相大白

到这里真相基本上就浮出水面了。 类的对象指针，成员函数指针，函数的参数，都作为 ObjClosure1 的类成员变量存储着。 然后在函数 Run() 里面再用起来。

```
1
2
3
virtual void Run() {
        (p_->*fun_)(arg1_);
}
```

可能看到这里的时候有点不理解，这里就涉及到类成员函数指针的调用。 显然因为类的成员函数是需要 this 指针的，这个写过 C++ 的应该都知道。 所以直接像C语言那样调用函数（如下），显然是不可能的，没有传入 this 指针嘛。

```
1
(*fun)(arg1_);
```

所以在 C++ 中，调用类的成员函数指针，是如下这样：

```
1
(p_->*fun_)(arg1_);
```

这样才能把 p_ 当成 this 指针传进去。

这样就实现了之前的 NewClosure 将 「类，类成员函数指针，函数参数」打包成一个闭包供线程池调用的意图。

所以就有了最开始示例代码那种写法。

最后

感觉自己写的还是很通俗易懂的，有具体代码，而且还是可一键下载运行的， 然后最底层的代码实现也都解释了。 应该算是很对得起本文题目了吧。

『题图』

题图是爱因斯坦，爱因斯坦代表高智商的大脑。 而在计算机领域，单核CPU计算性能已经很难有大的突破，发展趋势是多核计算。 所以多线程编程是这个时代的必备基础技能。

- 资源 - 优秀的工具资源导航
- 翻译 - 翻译传播优秀的外文文章
- 文章 - 国内外的精选文章
- 设计 - UI,网页，交互和用户体验
- iOS - 专注iOS技术分享
- 安卓 - 专注Android技术分享
- 前端 - JavaScript, HTML5, CSS
- Java - 专注Java技术分享
- Python - 专注Python技术分享