

# 期中周 | 期中测试题，你做对了么？

2023-02-17 卢誉声 来自北京

《现代C++20实战高手课》

课程介绍 >



讲述：卢誉声

时长 02:38 大小 2.41M



你好，我是卢誉声。

为了帮助你巩固知识，提升能力，期中周我给你出了一道实战题目，基于课程里的代码扩展现有协程框架，实现高级任务调度。题目描述你可以通过 [🔗 这个链接](#) 回顾。

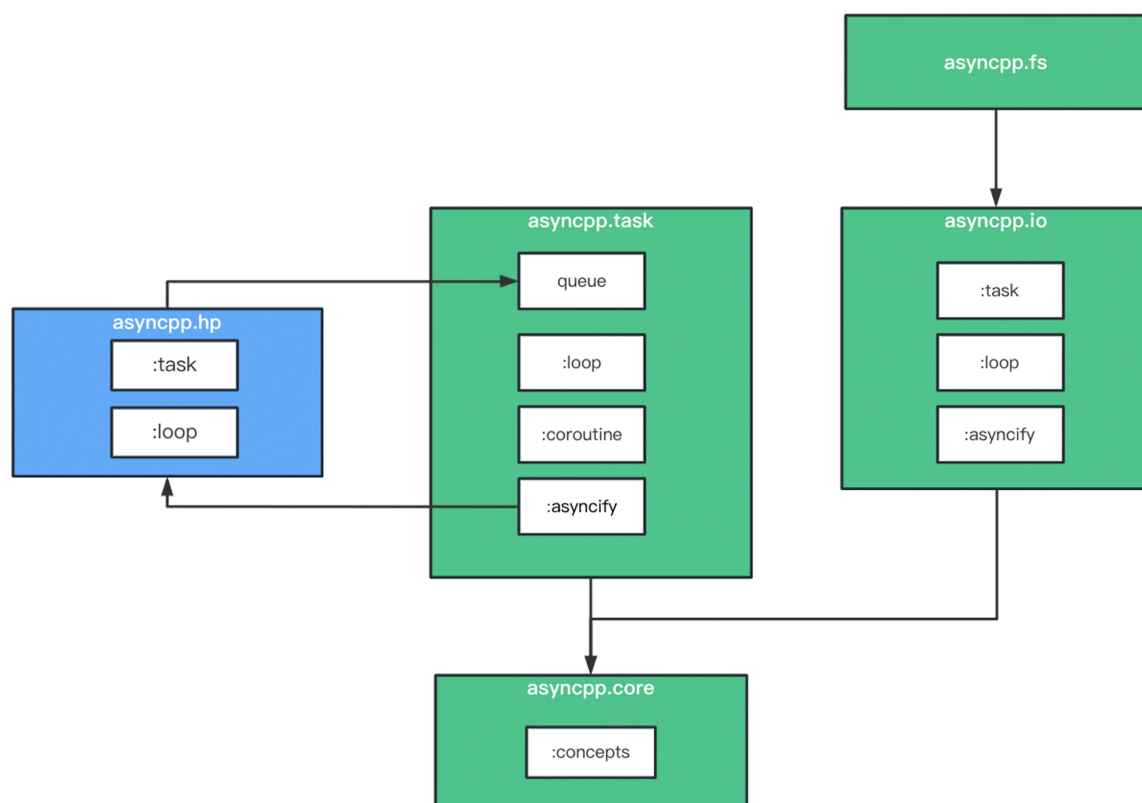
这一讲，我会把参考代码和解题思路公布出来。

## 答案解析

既然要在现有代码上增加功能，我们就有必要先熟悉原有架构，再决定在哪个模块或哪个层面上追加功能。

通过分析，可以发现任务的执行与调度是通过 `asyncpp.task` 模块实现的。同时，我们又是在为现有框架提供高优先级调度的能力。因此，新增的模块应该是供 `asyncpp.task` 使用的。

基于这样的考虑，我们在原有的架构基础上，追加了一个 high performance（`asyncpp.hp`）模块，供 `asyncpp.task` 模块实现高性能的线程调度。补充后的架构图是后面这样。



从图中可以看到，`asyncpp.task` 模块引用了 `asyncpp.hp` 模块。新的模块提供了高优先级线程调度和管理的能力。

我们来看一下具体实现。首先是 `.task` 子模块。

```
1 export module asyncpp.hp:task;
2
3 import asyncpp.core;
4 import <functional>;
5 import <vector>;
6 import <mutex>;
7
8 namespace asyncpp::hp {
9
10 export struct AsyncHpTask {
11     using ResumeHandler = std::function<void()>;
12     using TaskHandler = std::function<void()>;
13 }
```

```

14 // 协程唤醒函数
15 ResumeHandler resumeHandler;
16 // 计算任务函数
17 TaskHandler taskHandler;
18 };
19
20 export class AsyncHpTaskQueue {
21 public:
22     static AsyncHpTaskQueue& getInstance();
23
24     void enqueue(const AsyncHpTask& item) {
25         std::lock_guard<std::mutex> guard(_queueMutex);
26
27         _queue.push_back(item);
28     }
29
30     bool dequeue(AsyncHpTask* item) {
31         std::lock_guard<std::mutex> guard(_queueMutex);
32
33         if (_queue.size() == 0) {
34             return false;
35         }
36
37         *item = _queue.back();
38         _queue.pop_back();
39
40         return true;
41     }
42
43     size_t getSize() const {
44         return _queue.size();
45     }
46
47 private:
48     // 高性能计算任务队列
49     std::vector<AsyncHpTask> _queue;
50     // 高性能计算任务队列互斥锁，用于实现线程同步，确保队列操作的线程安全
51     std::mutex _queueMutex;
52 };
53
54 AsyncHpTaskQueue& AsyncHpTaskQueue::getInstance() {
55     static AsyncHpTaskQueue queue;
56
57     return queue;
58 }
59
60 }

```

这部分代码跟之前的 `asyncpp.io:task` 基本相同，唯一区别是调度的任务类型不同，用于处理高性能计算任务。

接下来的重点是:loop 的实现。

 复制代码

```
1  module;
2
3  #ifndef _WINDOWS_
4  #include <Windows.h>
5  #endif // _WINDOWS_
6
7  export module asyncpp.hp:loop;
8
9  import :task;
10 import asyncpp.task.queue;
11
12 import <thread>;
13 import <chrono>;
14 import <thread>;
15 import <functional>;
16
17 namespace asyncpp::hp {
18     export class AsyncHpLoop {
19     public:
20         // 常量，定义了任务循环的等待间隔时间（单位为毫秒）
21         static const int32_t SLEEP_MS = 100;
22
23         static AsyncHpLoop& start();
24
25     private:
26         // 支持单例模式，将其定义为private，防止外部调用构造函数
27         AsyncHpLoop() {
28             _thread = std::jthread(std::bind(&AsyncHpLoop::loopMain, this));
29             auto nativeWorkerHandle = _thread.native_handle();
30             ::SetThreadPriority(nativeWorkerHandle, THREAD_PRIORITY_HIGHEST);
31         }
32         // 支持单例模式，通过delete修饰符说明拷贝构造函数不可调用
33         AsyncHpLoop(const AsyncHpLoop&) = delete;
34         // 支持单例模式，通过delete修饰符说明赋值操作符不可调用
35         AsyncHpLoop& operator=(const AsyncHpLoop&) = delete;
36
37         void loopExecution() {
38             AsyncHpTask opItem;
39             if (!AsyncHpTaskQueue::getInstance().dequeue(&opItem)) {
40                 return;
41             }
42
43             opItem.taskHandler();
44
45             auto& asyncEventQueue = asyncpp::task::AsyncTaskQueue::getInstance(
46                 asyncEventQueue.enqueue({
47                     .handler = opItem.resumeHandler
48                 }));
49         }
50     };
51 }
```

```

49     }
50
51     void loopMain() {
52         while (true) {
53             loopExecution();
54             std::this_thread::sleep_for(std::chrono::milliseconds(SLEEP_MS))
55         }
56     }
57
58     // jthread对象，为高性能计算线程，jthread让该线程结束之前整个进程都不会结束
59     std::jthread _thread;
60 };
61
62 AsyncHpLoop& AsyncHpLoop::start() {
63     static AsyncHpLoop ioLoop;
64
65     return ioLoop;
66 }
67 }

```

我们在这段代码里不仅使用 `jthread`，还在 `AsyncHpLoop` 构造函数中使用了 `native_handle`，设置了线程优先级，用于高性能的任务调度。

最后，我们在 `asyncpp.task:asyncify` 中，使用了新的模块。

 复制代码

```

1  export module asyncpp.task:asyncify;
2
3  export import asyncpp.task.queue;
4  export import :loop;
5  export import :coroutine;
6
7  import asyncpp.core;
8  import asyncpp.hp;
9
10 namespace asyncpp::task {
11     using asyncpp::core::Invocable;
12
13     // 默认的AsyncTaskSuspend (当任务函数返回类型不为void时)
14     template <typename ResultType>
15     void defaultAsyncAwaitableSuspend(
16         Awaitable<ResultType>* awaitable,
17         AsyncTaskResumer resumer,
18         CoroutineHandle& h
19     ) {
20         auto& asyncTaskQueue = AsyncTaskQueue::getInstance();
21         asyncTaskQueue.enqueue({
22             .handler = [resumer, awaitable] {

```

```

23         awaitable->_taskResult = awaitable->_taskHandler();
24         resumer();
25     }
26 });
27 }
28
29 // 默认的AsyncTaskSuspend (当任务函数返回类型为void时)
30 template <>
31 void defaultAsyncAwaitableSuspend<void>(
32     Awaitable<void>* awaitable,
33     AsyncTaskResumer resumer,
34     CoroutineHandle& h
35 ) {
36     auto& asyncTaskQueue = AsyncTaskQueue::getInstance();
37     asyncTaskQueue.enqueue({
38         .handler = [resumer, awaitable] {
39             awaitable->_taskHandler();
40             resumer();
41         }
42     });
43 }
44
45 template <typename ResultType>
46 void hpAsyncAwaitableSuspend(
47     Awaitable<ResultType>* awaitable,
48     AsyncTaskResumer resumer,
49     CoroutineHandle& h
50 ) {
51     asyncpp::hp::AsyncHpTask operationItem{
52         .resumeHandler = [h] {
53             h.resume();
54         },
55         .taskHandler = [awaitable]() {
56             awaitable->_taskResult = awaitable->_taskHandler();
57         }
58     };
59
60     asyncpp::hp::AsyncHpTaskQueue::getInstance().enqueue(operationItem);
61 }
62
63 export template <Invocable T>
64 auto asyncify(
65     T taskHandler,
66     AsyncTaskSuspend<std::invoke_result_t<T>> suspender =
67     defaultAsyncAwaitableSuspend<std::invoke_result_t<T>>
68 ) {
69     return Awaitable<std::invoke_result_t<T>>{
70         ._taskHandler = taskHandler,
71         ._suspender = suspender
72     };
73 }
74

```

```

75     export template <Invocable T>
76     auto asyncify(
77         T taskHandler,
78         bool highPriority
79     ) {
80         if (highPriority) {
81             return Awaitable<std::invoke_result_t<T>>{
82                 ._taskHandler = taskHandler,
83                 ._suspender = hpAsyncAwaitableSuspend<std::invoke_result_t<T>>
84             };
85         }
86         return Awaitable<std::invoke_result_t<T>>{
87             ._taskHandler = taskHandler,
88             ._suspender = defaultAsyncAwaitableSuspend<std::invoke_result_t<T>>
89         };
90     }
91 }

```


你可以先关注一下第 8 行代码，这里我们导入了 `hp` 模块的符号。接着，还在第 46 行实现了 `hpAsyncAwaitableSuspend`，用于实现高性能版本的调度。

最后，我们在第 76 行实现了类似于之前的 `asyncify` 工具，用于将一个普通的函数 `f` 转换成一个返回 `Awaitable` 对象的函数 `asyncF`。通过这个分区实现的工具，可以让库的用户更容易使用 `coroutine`。

这道题目的源代码，你可以从 [这里](#) 获取。

期中周即将告一段落，你可以再利用周末时间温习一下之前所学。有不明白的地方欢迎和我在留言区交流，下周我们继续回到课程主线，继续学习 `C++ Ranges` 的用法，敬请期待。

分享给需要的人，Ta 购买本课程，你将得 **18 元**

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。