A Comprehensive Discussion on C++20 Coroutines

Alibaba Cloud Community February 28, 2024

This article is an introduction to C++20 coroutines. It provides a comprehensive explanation of the concept and implementation of coroutines

By Qianyi

Concept of Coroutines

Let's start with the concept of coroutines. If you already have a good understanding of the background knowledge, you can skip this section (or take a quick look because we may have some different ideas about coroutines). Here I would like to go a little deeper, as this is crucial to

whether you can correctly comprehend and resolve complex coroutine issues later on.

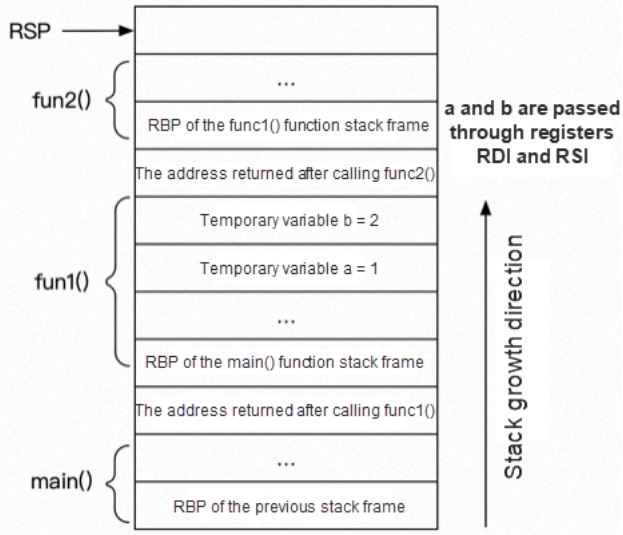
Coroutines, also known as micro-threads or fibers, generally refer to a logical entity that can be paused and resumed. A regular function has two main operations or behaviors: **call and return**. When this function is called, the current execution is paused, and the program jumps to the starting position of this function to execute it. After the function execution is completed, it returns a result or throws an exception. This calling process is usually a one-time action. If the function is called again, it is treated as another independent action. However, coroutines are different. The operations and behaviors of coroutines include **create, suspend, resume**, and **destroy**. Coroutines allow the called function to retain its temporary information and suspend when it reaches a certain position during execution. Later, it can return to the same position and state to resume execution. Therefore, in a sense, coroutines can be seen as a generalization of regular functions.

When a program starts executing, the operating system creates a process. The operating system's kernel also creates a scheduling entity, which is a data structure that stores information related to the process. Then, the program jumps to the code entry point for execution. The operating system often executes multiple processes simultaneously. These processes are executed in parallel or in a sequential manner on a physical multi-core CPU. When the number of processes exceeds the number of CPU cores, the processes are queued. The operating system handles all the scheduling work for these entities, including monitoring, counting, and switching. A process usually has only one regular execution flow (main thread), which corresponds to the serial execution of the `main()` function in C/C++ language. The `main()` function may call other functions, and those functions may further call other functions. Eventually, the process exits after the `main()` function execution is completed. When a process executes code, the information of the current function, such as the execution state and temporary result, is stored in the CPU's registers. In order to call a new function, the current register values must be saved in memory because the new function will also use these registers for calculations.

Now, which registers need to be saved? Where do the parameters passed to the new function go? Where is the return value stored after the call? These details are specified by **calling conventions**. This article will not delve into the details of calling conventions. If you are interested, please refer to [1].

In the AMD64/x86_64 architecture, function calls are implemented using stacks, and each function has its own stack frame. Below is an example of code and the corresponding stack state during its execution.

```c
int fun3() {
    return 1;
}

int func2(int a, int b) {
    return a + b;
}

int func1() {
    int a = 1;
    int b = 2;
    int c = func2(a, b);
    return fun3() + c;
}


int main() {
    func1();
    return 0;
}
```



When the code is executed to `main()->func1()->func2()`, the memory stack is arranged as shown in the diagram. When a new function is called, the stack space grows upwards to create a stack frame for that function. This includes creating local variables and saving the registers used by the previous function. After the function call returns, the stack frame is reclaimed, the previous register values are restored from the stack, and the execution jumps back to the previous code address. However, accessing the recycled stack frame after the function returns is risky, as the space may be overwritten by temporary data from other functions. Therefore, it is not recommended to return pointers to a function's local variables in C/C++ code. The stack grows and shrinks as functions are called and return. What is the default size of this stack? On a Linux system, you can execute `ulimit -s` to see that the default value is 8,192 KB. If the local variables defined in a program are too large, or if the function calls are too deep (especially during recursive operations), there is a risk of stack overflow.

The memory area opposite the stack is the heap, typically allocated during program execution using `brk(2)` or `mmap(2)`. In C/C++ programs, `malloc(3)/free(3)` or `new/delete` is used (discussion here is limited to the semantics of memory allocation). Allocations are generally an encapsulation by the CRT or a memory allocation library like jemalloc. Heap memory isn't automatically reclaimed when a function call ends; the program must explicitly release it after use. Fundamentally, the heap and stack are concepts in memory logic; they differ only in terms of timing and usage, but there's no essential difference.

If a program logically requires more execution flows, it can create threads to concurrently execute another function. Here, a thread is typically a concept supported by the operating system, corresponding to a kernel scheduling entity and capable of parallel execution with the main thread on different CPU cores. When does the OS kernel schedule tasks? Naturally, when the execution stream jumps into kernel code. Besides system calls dropping into the kernel and triggering additional scheduling, hardware clock interrupts are also necessary to forcibly interrupt the currently executing program to switch to kernel logic. Otherwise, a program in an infinite user-mode loop would never yield execution rights. A created thread is similar to the main thread, as it has a fully independent thread stack for preserving its function call state.

So, with all this groundwork laid, what exactly are coroutines? As mentioned earlier, coroutine functions involve operations and behaviors like **create, suspend, resume, and destroy**. Creation and destruction are straightforward and also steps in creating threads. The additional operations, suspend and resume, are because the execution logic is completely simulated in user mode without a kernel scheduling entity, and thus naturally requires user-mode management of these execution flows' suspension and resumption. If you're familiar with the history of Linux threads, you'll know that before the Linux kernel supported the concept of threads, the multiple switchable "threads" simulated in user mode were actually an implementation of coroutines. In multi-threaded programming, one generally doesn't add active switching logic within the thread; the kernel often passively converts the thread from an execution state to a waiting state—for example, when I/O operations are incomplete or when resources to be acquired are temporarily occupied. At most, some system calls may be made to inform the kernel to temporarily yield the CPU and check if other tasks can be performed. However, coroutines operate differently, requiring user logic to actively switch out and yield execution rights when subsequent execution conditions are not met.

Let's tentatively refer to this type of coroutine as the first generation of coroutines. It is simply a theoretical deduction of the concept of threads. However, coroutines are a very broad concept in general, especially with the new trend brought by the `async/await` semantics. In recent years, almost all new programming languages have introduced coroutine models based on the `async/await` semantics (except for the Go language 😁). This article will not delve into the history of the development of `async/await` in Microsoft. If you are interested, you can refer to relevant information.

So, how can we implement coroutines in the C/C++ language? The first idea that comes to mind is to implement it in a thread-like manner (first-generation coroutine abstraction), where a separate memory stack is created for each coroutine to save the context and call functions. This method of switching between coroutines is relatively simple, and it only requires the user code to save the context itself and then jump directly to the target function position for execution. This is similar to how the operating system kernel switches between multiple execution streams. It is also analogous to the concept of **stackful coroutine** mentioned in C++20 coroutine-related materials. However, creating a stackful coroutine has a relatively high cost. Independent coroutine stacks need to be created and memory must be pre-allocated before the coroutine function can be executed, which limits the number of concurrent coroutines. Moreover, if the pre-allocated stack is too large, it will cause waste, and if it is too small, it will result in stack overflow when the function call depth increases. In reality, from the perspective of memory allocation in the Linux kernel, memory pages are only allocated when they are actually used. In other words, a coroutine stack wastes at most one physical page more than the memory actually used. Additionally, the branch prediction mechanism of modern CPUs includes the prediction of the return stack buffer (RSB). The constant manual stack switching involved in coroutine execution disrupts the prediction mechanism of the RSB. The more complex and frequent the switching, the greater the impact. Nevertheless, the advantage of this model is that it is almost transparent to the compiler. The transformation of existing code into coroutines is relatively simple. You only need to modify the position where the coroutine is created and add active switching points in the code. If a hook is used to automatically add `yield`, you don't even need to modify the existing code.

> Let's briefly mention the **shared stack coroutine** (copying the stack coroutine). Creating an additional stack for each coroutine results in excessive resource consumption, so we can create only one stack. When switching between coroutines, the used stack is copied to another memory, freeing up the stack for the new coroutine. When switching back, the previous stack is copied back. Shared stack coroutine solves the problem of wasted pre-allocated memory but introduces the overhead of stack backup and restoration. For optimal performance, it is necessary to

> minimize the depth of function calls and avoid allocating excessively large data structures on the stack. Therefore, shared stack coroutine is only an optimization method and is generally not singled out for comparison and discussion.

The opposite of the stackful coroutine is the **stackless coroutine**, which is the mode adopted by C++20. In this mode, the created coroutines are very lightweight. Initially, all the **temporary variables** of the coroutine function and context information such as calling parameters are saved on the heap. Since most of the information is stored on the heap, switching from the coroutine function can be very fast. It doesn't require much stack restoration to jump back to the original code position and resume execution. The created coroutine doesn't allocate a new memory stack but restores the context at the call/restore position. Therefore, it is called a stackless coroutine. However, adapting existing code to this mode cannot simply transform it into a coroutine. It requires refactoring or even rewriting the old code to complete the transformation.

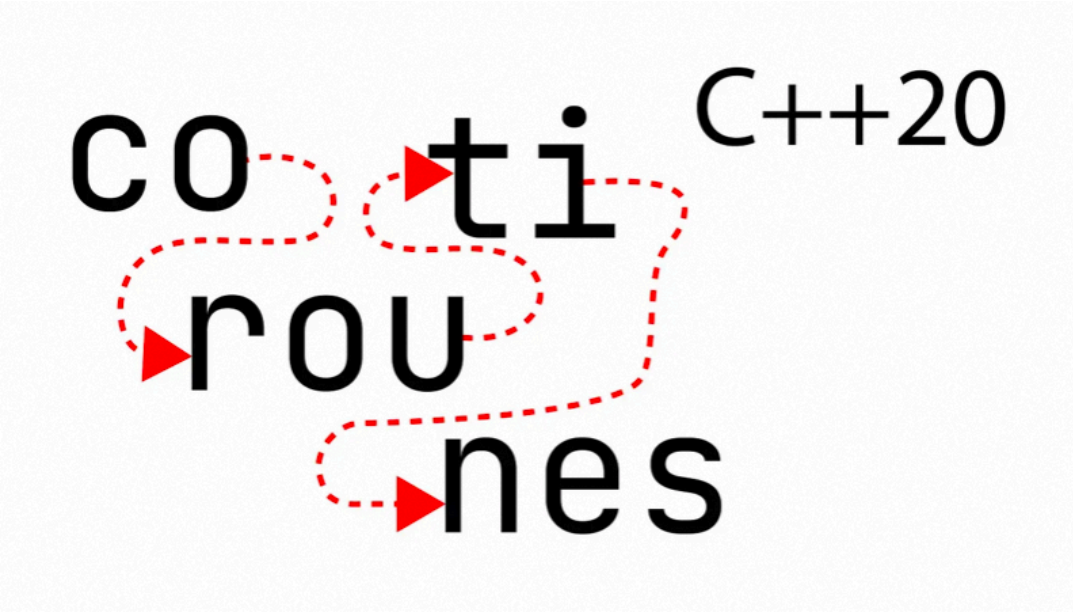
It should be noted that the discussion on stackful and stackless coroutines is from the perspective of the C/C++ language. Some languages, such as Python, don't allocate a **stack** for execution streams to save context in the stack space, so they are not included in the discussion. Even when discussing the concept of coroutines in a purely theoretical way, mentioning the stack seems unprofessional. Stacks are not the only way to implement function calls; it's just the method used by most modern mainstream programming languages. Although this doesn't change the intended meaning of this article, it's important to clarify that this is only a discussion on engineering in the current C/C++ language and not a purely theoretical analysis. Additionally, I'm just a regular programmer who writes code. I can't discuss the semantics of `async/await` from a scientific or mathematical perspective. 😁

> Explaining too much may confuse the reader, but not explaining enough may leave the reader in the dark. This article introduces the specific implementation principle to help readers quickly understand the abstract theory, but it may result in a narrow understanding of coroutine theory. Please keep this in mind while reading.

What is the future direction of coroutines? Currently, coroutines primarily focus on solving the problem of "formalized synchronous writing" of asynchronous code, which means helping programmers write asynchronous logic in an approximate synchronous way, so that code with logical relationships is not fragmented in form. However, this abstraction comes with a runtime cost. In pure theory, coroutines are generally not as efficient as threads combined with callbacks, at least not at the current level of automatic optimization. We cannot rely on coroutines for extreme performance. **The correct direction is to seek a balance between maintenance costs and performance, or to reduce development costs.**

Some people say that a program is a state machine, and threads and coroutines are for those who can't write a good state machine [2]. Does this make sense? In fact, it does make some sense. From the perspective of final execution, a program is a state machine executed on the CPU. However, in terms of program design, higher-level abstractions and methods help in writing logic that is easier for humans to understand and improve. High-level abstractions are essential infrastructure for building large software systems. A little runtime cost is worth it. With technological advancements, the runtime cost will gradually be optimized and reduced. We must always remind ourselves that code is written for humans to read, not for machines.

Implementation of C++20 Coroutines



Why do we spend so much time discussing the background and principles? I must admit that it took me several readings of the cppreference document to understand the coroutine mechanism of C++20. The coroutine

mechanism currently implemented in C++20 is not intended for end users to use directly. Instead, it provides coroutine library authors with compiler support and syntactic sugar. In my opinion, the best way to understand the current implementation is to see it from the perspective of the designers.

As mentioned earlier, the coroutine implementation in C++20 is a Stackless implementation. The coroutines it creates store the necessary data for their execution on the heap. After a coroutine is switched out and saves its context information on the heap, the previous function execution is resumed at the original stack position. When the coroutine switches back, it also restores the previous state on the stack of the currently calling recovery operation and then executes it. What support does the compiler provide in this process? The compiler supports automatic saving and restoring of coroutine context, automatic variable capturing, and the save mechanism on the heap. This is similar to the go language, which supports returning function local variables and the compiler will automatically save them on the heap. Therefore, the current coroutines in C++20 are just a basic implementation with additional support from the compiler. This aligns with the style of C++, where the compiler provides minimal features that cannot be achieved by the standard library and leaves the rest to the standard library.

The coroutine code in C++20 will be expanded into more complex code by the compiler, similar to syntactic sugar. This requires following certain conventions when writing user-side code, and then relying on the compiler to generate the subsequent parts using a codegen-like method. If you are familiar with JavaScript and have studied the React lifecycle, you may already have an idea of what this entails. This is why I still found myself confused after reading many demos of C++20 coroutine code. In essence, it is important to understand the process after the code is generated, and then you can understand why these predefined types and callbacks are necessary to help the compiler expand the code.

Start with a "Simple" Demo.

Here is a simple demo code of a C++20 coroutine, which can be executed with Compiler Explorer [3] if you don't have a higher version of the compiler:

```
#include <iostream>
#include <coroutine>

template <bool READY>
struct Awaiter {
    bool await_ready() noexcept {
        std::cout << "await_ready: " << READY << std::endl;
```

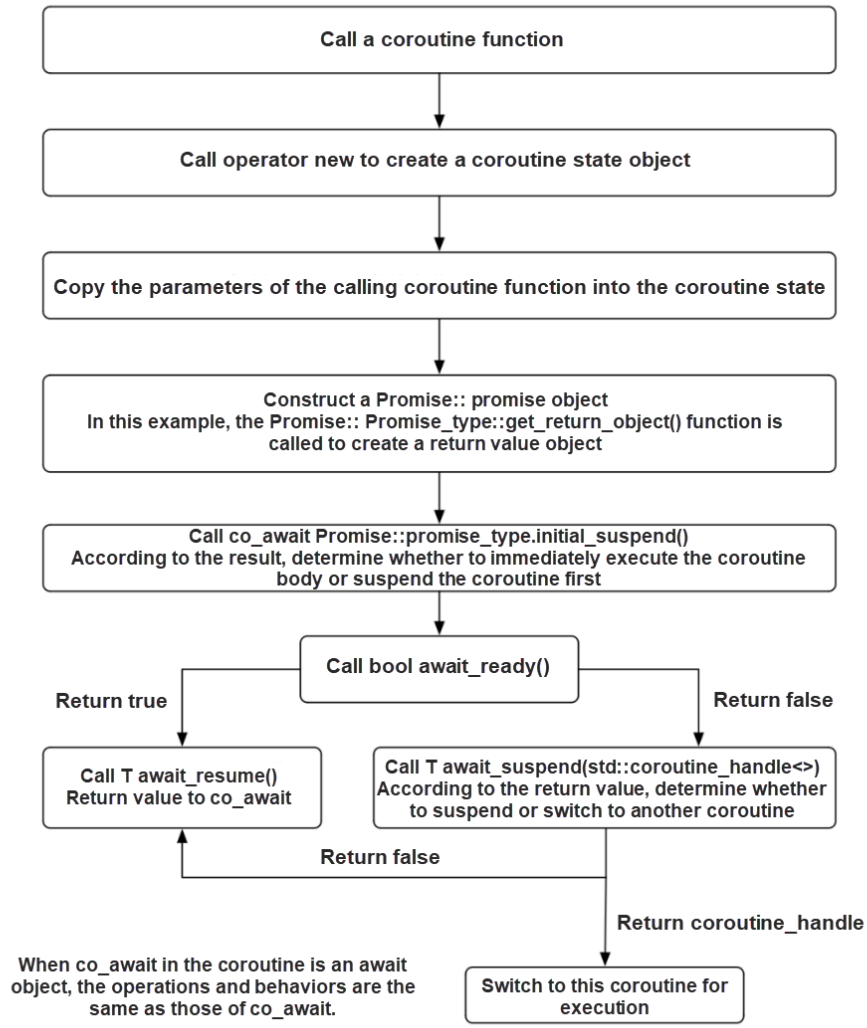
```
        return READY;
    }
    void await_resume() noexcept {
        std::cout << "await_resume" << std::endl;
    }
    void await_suspend(std::coroutine_handle<>) noexcept {
        std::cout << "await_suspend" << std::endl;
    }
};

struct TaskPromise {
    struct promise_type {
        TaskPromise get_return_object() {
            std::cout << "get_return_object" << std::endl;
            return TaskPromise{std::coroutine_handle<promise_typ
        }
        Awaiter<true> initial_suspend() noexcept {
            std::cout << "initial_suspend" << std::endl;
            return {};
        }
        Awaiter<true> final_suspend() noexcept {
            std::cout << "final_suspend" << std::endl;
            return {};
        }
        void unhandled_exception() {
            std::cout << "unhandled_exception" << std::endl;
        }
        void return_void() noexcept {
            std::cout << "return_void" << std::endl;
        }
    };

    void resume() {
        std::cout << "resume" << std::endl;
        handle.resume();
    }

    std::coroutine_handle<promise_type> handle;
};

TaskPromise task_func() {
    std::cout << "task first run" << std::endl;
    co_await Awaiter<false>{};
    std::cout << "task resume" << std::endl;
}

int main() {
    auto promise = task_func();
    promise.resume();

    return 0;
}
```

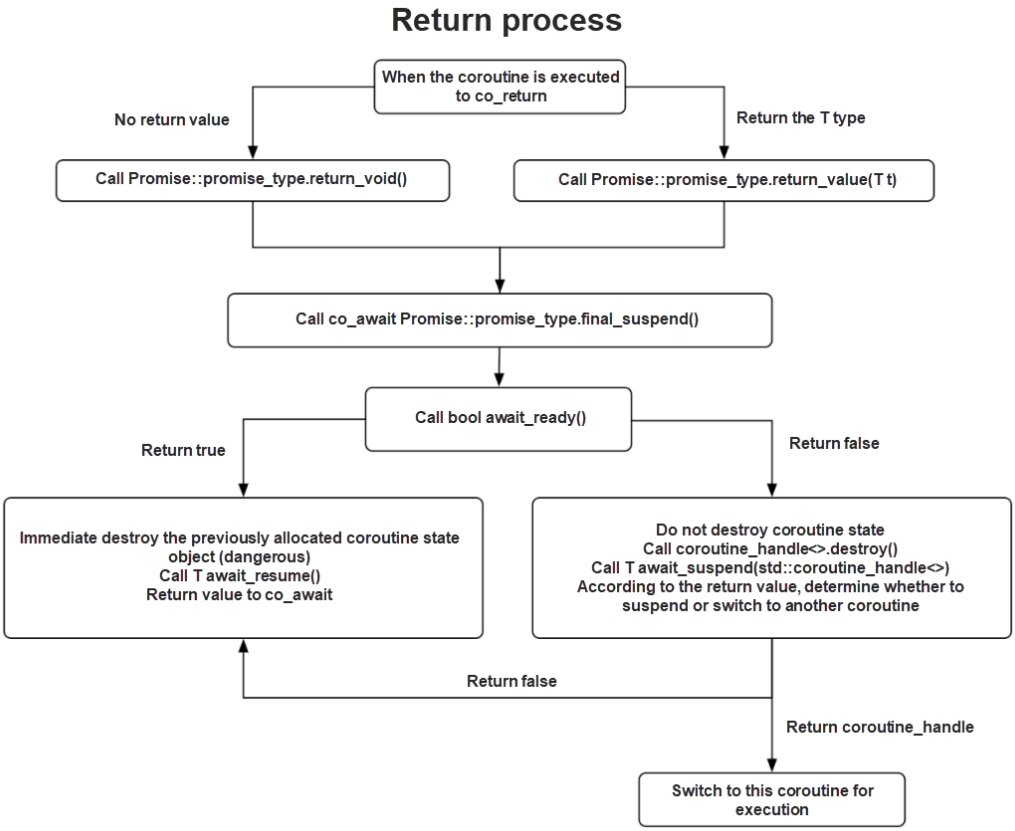
The following is the output of this code [4] after running:

```
get_return_object
initial_suspend
await_ready: 1
await_resume
task first run
await_ready: 0
await_suspend
resume
await_resume
task resume
return_void
final_suspend
await_ready: 1
await_resume
```

Although I have made an effort to make the first demo code as short as possible, it is still much longer than coroutine demos in other languages. The reason is quite simple: if C++ wants programmers to customize the behavior of any step in any phase of coroutine creation and execution, then enough callback functions must be defined to specify the behavior of each phase. Without referring to any documentation, it can be difficult to understand the above code. However, if we recall the implementation principle of the "compiler expansion" code mentioned earlier and consider the execution order of these functions, we can roughly infer the order of function calls after the code is expanded by the compiler. In fact, cppreference provides the specific execution process [5], but it may not be clear enough. Here, we will describe and explain the process in more detail:

Creation and Execution Process





1. Call `operator new` to apply for space and initialize the coroutine state. The coroutine state is a class automatically generated by the compiler based on the coroutine function. Each different coroutine generates this class separately.
2. Copy or move the parameters of the calling coroutine function to the coroutine state object [6]. The parameters must be saved onto the heap to be retained during switching. In addition, if other stack variables are defined inside the function body, they also need to be placed on the heap. This is not defined in the demo. This is done by automatic analysis by the compiler. Some stackless C/C++ coroutine libraries that do not rely on the compiler to implement will create temporary variables for each function and allocate them on the heap. Of course, the native support of the compiler will be much simpler and more natural. This raises a question: If some temporary objects have already left the scope when the coroutine returns and theoretically do not need to be captured, does this mean there is no need to save them additionally? Can they be destructed directly? The answer is No, because there is a rule in C++ that the order of object destruction is the reverse order of construction. Therefore, once an intermediate object is saved on the heap, it may be forced to save more objects to ensure the destructuring order. It is hoped that subsequent standards will revise

the behavior specifications of coroutine functions here, which can improve performance.

3. Construct the `Promise::promise_type` object of the coroutine, which is also saved in the coroutine state. The `Promise` object is the return value object of the coroutine specified in C++, corresponding to the `TaskPromise` class [7] in the demo code. If the user-defined `Promise::promise_type` has a constructor that accepts all coroutine arguments, that constructor is called. Otherwise, the default constructor is called. The `promise_type.get_return_object()` function is then called to create the coroutine function's return value object `TaskPromise`. This object is returned to the position where the coroutine function is called when the coroutine is first suspended. `coroutine_handle<promise_type>` and `promise_type` can be converted to each other by using the `handle::from_promise()` and `handle.promise()` interfaces. We will not detail the principle here. This involves the implementation of some compiler builtin, which is related to the memory layout of specific objects. The coroutine handle `std::coroutine_handle<>` is an object that can operate coroutines, which is similar to the `std::thread` object that can operate threads. This is for the purpose that coroutine functions can be called like normal functions, so the way to obtain this handle is somewhat awkward.
4. Call `Promise::promise_type.initial_suspend()`, and the latter will return an `awaitable` object that has three defined member functions. All you need to know is that when the `await_ready()` member returns true, it will not suspend by default and will call the `await_resume()` function immediately. Otherwise, it suspends the coroutine and calls the `await_suspend()` function immediately. Most of the time, `co_await` only needs to know whether to execute immediately, so the standard library provides the default implementations of `Awaiter`: `std::suspend_never` and `std::suspend_always` classes. The former never suspends, and the latter always suspends. Why is it made so complicated? This is for scalability. You can use this returned object to control whether a coroutine is executed immediately or suspended immediately after it is created, and whether additional operations need to be done before execution. This allows you to conveniently support the scheduling mechanism of the coroutine. By the way, the return value of the `await_resume()` function in the demo is void, but it can be any type. It is used as the return value of the `co_await` expression. It is not returned in the demo. In addition, the return value of the `await_suspend()` function in the demo is void, but it can also be bool (returning false will cause no suspension), or even the

`coroutine_handle` object of other coroutines. At this time, it will switch to the coroutine for execution. This provides basic support for switching and scheduling. We will discuss this mechanism in detail later.

5. Choose to directly execute or suspend according to Step 4. If you directly suspend, the `Promise` object will be returned to the caller immediately. Otherwise, it will not return to the caller until the coroutine is explicitly suspended or execution is completed. In the demo, the function does not suspend by default, and will not be returned to the caller until the first suspension. The caller immediately `resume()` the coroutine function, and then the coroutine function completes execution and exits. The preceding code does not explicitly write the return statement. The compiler will add `co_return` at the end, which will call the `promise.return_void()` function. No value is returned here. If `co_return` returns a value `T t`, you need to define a function called `void return_value(T t)`. `return_value()` and `return_void()` cannot coexist.
6. Finally, the compiler calls the `co_await Promise::promise_type.final_suspend()` function to end the coroutine. The function will be called whether it ends with an exception or exits normally. Note that `final_suspend()` returns another `awaitable` object, but when `std::suspend_always()` is used here to return "suspend", the coroutine will not immediately destroy the internal status information (otherwise it will be destroyed directly), because some information is still saved in `Promise`. The best practice is to write `coroutine_handle<>.destroy()` in the destructor of `Promise`. It is better to make the lifecycle of the `handle` consistent with the `Promise` object. Otherwise, operating the `Promise` object after the coroutine exits will be UAF. The demo does not need this currently, so it does not follow this practice for simplicity.
7. As for `promise.unhandled_exception()`, it is called when an uncaptured exception occurs in the coroutine. However, note that exceptions thrown before `promise.get_return_object()` will not come here. For example, the `std::bad_alloc` exception caused by `new` will not be called here. We will not detail the exception handling process here. It is clearly described in cppreference.

If we combine the preceding `task_func` code running result with this process introduction, the code after the compiler is expanded can be roughly imagined:

```
TaskPromise task_func() {
    // No parameters and local variables.
```

```
    auto state = new __TaskPromise_state_(); // has TaskPromise:
    TaskPromise coro = state.promise.get_return_object();
    try {
        co_await p.inital_suspend();
        std::cout << "task first run" << std::endl;
        co_await Awaiter<false>{};
        std::cout << "task resume" << std::endl;
    } catch (...) {
        state.promise.unhandled_exception();
    }
    co_await state.promise.final_suspend();
}
```

Expanding `co_await` can be troublesome, especially because the meaning of the synchronized return value of `await_suspend()` is different. We will not detail the compiler code expansion here. As the basic principles are clearly explained, we will first focus on how to apply them[8].

As mentioned earlier, the coroutine function must return a `Promise` object that conforms to the specification, and there must be a `Promise::promise_type` inside this object. The name cannot be changed, but it can be defined elsewhere with another name. In the class, using is declared as this name. This `Promise::promise_type` must implement the necessary functions mentioned in the preceding process. Otherwise, the compilation will not pass.

The following is an example to demonstrate the behavior of the three interface functions of the `Awaiter` object, namely `await_ready()`, `await_suspend()`, and `await_resume()`:

```
#include <iostream>
#include <coroutine>
#include <future>
#include <thread>

struct TaskPromise {
    struct promise_type {
        TaskPromise get_return_object() {
            std::cout << "get_return_object(), thread_id: " << s
            return TaskPromise{std::coroutine_handle<promise_typ
        }
        std::suspend_always initial_suspend() noexcept { return
        std::suspend_always final_suspend() noexcept { return {}
        void unhandled_exception() {}
        void return_void() noexcept {}
        size_t data = 0;
    };
    std::coroutine_handle<promise_type> handle;
```

```
};

struct Awaiter {
    bool await_ready() noexcept {
        std::cout << "await_ready(), thread_id: " << std::this_t
        return false;
    }
    void await_suspend(std::coroutine_handle<TaskPromise::promis
        std::cout << "await_suspend(), thread_id: " << std::this
        auto thread = std::thread([=]() {
            std::this_thread::sleep_for(std::chrono::seconds(1))
            handle.promise().data = 1;
            handle.resume();
        });
        thread.join();
    }
    void await_resume() noexcept {
        std::cout << "await_resume(), thread_id: " << std::this_
    }
};

TaskPromise task_func() {
    std::cout << "task_func() step 1, thread_id: " << std::this_
    co_await Awaiter{};
    std::cout << "task_func() step 2, thread_id: " << std::this_
}

int main() {
    std::cout << "main(), thread_id: " << std::this_thread::get_
    auto promise = task_func();
    std::cout << "main(), data: " << promise.handle.promise().da
    promise.handle.resume();
    std::cout << "main(), data: " << promise.handle.promise().da

    return 0;
}
```

The following result is returned:

```
main(), thread_id: 0x1d9d91ec0
get_return_object(), thread_id: 0x1d9d91ec0
main(), data: 0, thread_id: 0x1d9d91ec0
task_func() step 1, thread_id: 0x1d9d91ec0
await_ready(), thread_id: 0x1d9d91ec0
await_suspend(), thread_id: 0x1d9d91ec0
await_resume(), thread_id: 0x16dce7000
task_func() step 2, thread_id: 0x16dce7000
main(), data: 1, thread_id: 0x1d9d91ec0
```

It is easy to understand this code by combining the log and the preceding process description. The `handle` parameter for calling `await_suspend()` in line 26 of the code is passed in with the help of the compiler. This handle is a `void *` pointer, and the cost of value passing is very low. Line 31 of the code is called on another thread, and the subsequent coroutine code is also run on another thread. This also reveals the cross-thread passing ability of the coroutine. As long as the coroutine handle is passed, the execution of the coroutine can be resumed on any thread. Then, when the coroutine is passed across threads, you still need to pay attention to the thread safety issues. The execution stream can move arbitrarily, so other synchronization issues also require additional attention.

Implement a Simple Generator

Generators are very common in languages such as Python/Js. They could not be simply implemented before C++20, but now by using the proceding coroutine mechanism, simple implementations can be easily written.

Suppose there is a need to obtain the Fibonacci sequence, each time we need to get a subsequent number in the sequence. What should you do if you simply implement it in C++? You need to define a class for the Fibonacci sequence generator. This class has a function in the shape of `size_t next()`. Each time it is called, it will return the next number. Then the relevant variables must be saved in the class member variables so that the next time the `next()` function is called, it will calculate the value to be returned based on the previous value. The code for this class is simple, so I will not discuss it in detail here. The following is the implementation in the form of coroutine:

```
#include <iostream>
#include <coroutine>

template <typename T>
struct Generator {
    struct promise_type {
        Generator get_return_object() {
            return Generator{std::coroutine_handle<promise_type>
        }
        std::suspend_always initial_suspend() noexcept { return
        std::suspend_always final_suspend() noexcept { return {}
        void unhandled_exception() {}
        void return_value(T t) noexcept {
            v = t;
        }
        std::suspend_always yield_value(T t) {
            v = t;
```

```
            return {};
        }
        T v{};
    };
    bool has_next() {
        return !handle.done();
    }
    size_t next() {
        handle.resume();
        return handle.promise().v;
    }
    std::coroutine_handle<promise_type> handle;
};

Generator<size_t> fib(size_t max_count) {
    co_yield 1;
    size_t a = 0, b = 1, count = 0;
    while (++count < max_count - 1) {
        co_yield a + b;
        b = a + b;
        a = b - a;
    }
    co_return a + b;
}

int main() {
    size_t max_count = 10;
    auto generator = fib(max_count);
    size_t i = 0;
    while (generator.has_next()) {
        std::cout << "No." << ++i << ": " << generator.next() <<
    }
    return 0;
}
```

The following result is returned:

```
No.1: 1
No.2: 1
No.3: 2
No.4: 3
No.5: 5
No.6: 8
No.7: 13
No.8: 21
No.9: 34
No.10: 55
```

`co_await` and `co_return` have been described earlier. The new identifier co_yield is regarded as a syntactic sugar for co_await, which can pass a

value more conveniently than co_await does [9].

The `Generator` here is defined as a template class that can be used to quickly define various types of `Generators`. If there are other needs, it can be quickly implemented. Of course, the `Generator` here is not very comprehensive. In general, it is also necessary to consider that the lifecycle of the coroutine state corresponding to the handle should be consistent with the returned Promise object, so after `final_suspend()` returns `std::suspend_always`, Promise needs to release the relevant memory of the coroutine corresponding to the handle. After adding release logic to the destructor, you need to improve the functions of `Promise` class, including structure, copy, and other functions (the 3-5 rule). The more refined `Generator` code is as follows:

```
template <typename T>
class Generator {
public:
    struct promise_type;
    using promise_handle_t = std::coroutine_handle<promise_type>
    explicit Generator(promise_handle_t h) : handle(h) {}
    explicit Generator(Generator &&generator) : handle(std::exch
    ~Generator() {
        if (handle) {
            handle.destroy();
        }
    }
    Generator(Generator &) = delete;
    Generator &operator=(Generator &) = delete;

    struct promise_type {
        Generator get_return_object() {
            return Generator{std::coroutine_handle<promise_type>
        }
        std::suspend_always initial_suspend() noexcept { return
        std::suspend_always final_suspend() noexcept { return {}
        void unhandled_exception() {}
        void return_value(T t) noexcept {
            v = t;
        }
        std::suspend_always yield_value(T t) {
            v = t;
            return {};
        }
        T v{};
    };
    bool has_next() {
        return !handle.done();
    }
    size_t next() {
```

```
        handle.resume();
        return handle.promise().v;
    }
private:
    std::coroutine_handle<promise_type> handle;
};
```

Implementing this `Generator` is once and for all. Of course, it would be easier if the standard library had the `std::generator` template, but this wish would not be realized until C++23.

You may find it not difficult to write a simple class to implement relevant logic. However, if the algorithm is complex and needs to save a lot of variable values, it is then not so easy to extract these variables into member variables and implement phased returns in the `next()` function. Besides, the code written is definitely not as easy to maintain as this coroutine version.

Now that you have the `Generator`, can other functional programming functions such as `filter` and `map` in Python be implemented? Of course you can, but that's a long way off. This article will not continue to discuss functional programming. Let's return to coroutines.

A Generic Coroutine Return Class: Task

Up to now, there are already some open source coroutine libraries based on C++20 coroutine basic support, such as cppcoro [10], the coroutine library coro [11] in follyl, and open source async_simplep [12] provided by Alibaba. I have to say that although the name of async_simple is not "high-end", the code quality is quite good. 😄.

These coroutine libraries will provide some common auxiliary classes such as `coroutine types` and `awaitable types`, as well as some synchronization mechanisms such as locks based on C++20 coroutines, which can easily implement coroutine logic. Among these infrastructures, the most basic one is the implementation of a generic coroutine return type `Task<T>`.

A fully functional and highly scalable `Task<T>` is too complex, but a simple demo is relatively easy. After referring to the general idea of the preceding coroutine library and other references, here is a simple implementation. For the sake of simplicity, the saving and handling of all C++ exceptions are ignored. In fact, we usually do not use the exception mechanism in C++ projects. If there are unexpected exceptions, we will just `let it crash`.

```
#include <iostream>
#include <functional>
#include <deque>
#include <optional>
#include <coroutine>
#include <thread>

template <typename T>
class Task {
public:
    struct promise_type;
    using promise_handle_t = std::coroutine_handle<promise_type>

    explicit Task(promise_handle_t h) : handle(h) {}
    Task(Task &&task) noexcept : handle(std::exchange(task.handl
    ~Task() { if (handle) { handle.destroy(); } }

    template <typename R>
    struct task_awaiter {
        explicit task_awaiter(Task<R> &&task) noexcept : task(st

        task_awaiter(task_awaiter &) = delete;
        task_awaiter &operator=(task_awaiter &) = delete;

        bool await_ready() noexcept { return false; }
        void await_suspend(std::coroutine_handle<> handle) noexc
            task.finally([handle]() { handle.resume(); });
        }
        R await_resume() noexcept { return task.get_result(); }

    private:
        Task<R> task;
    };

    struct promise_type {
        Task get_return_object() {
            return Task(promise_handle_t::from_promise(*this));
        }

        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}

        template <typename U>
        task_awaiter<U> await_transform(Task<U> &&task) {
            return task_awaiter<U>(std::move(task));
        }

        void unhandled_exception() {}

        void return_value(T t) {
            data_ = t;
            notify_callbacks();
```

```
        }

        void on_completed(std::function<void(T)> &&callback) {
            if (data_.has_value()) {
                callback(data_.value());
            } else {
                callbacks_.push_back(callback);
            }
        }

        T get() {
            return data_.value();
        }

    private:
        void notify_callbacks() {
            for (auto &callback : callbacks_) {
                callback(data_.value());
            }
            callbacks_.clear();
        }

        std::optional<T> data_;
        std::deque<std::function<void(T)>> callbacks_;
    };

    T get_result() {
        return handle.promise().get();
    }

    void then(std::function<void(T)> &&callback) {
        handle.promise().on_completed([callback](auto data) {
            callback(data);
        });
    }

    void finally(std::function<void()> &&callback) {
        handle.promise().on_completed([callback](auto result) {
            callback();
        });
    }

private:
    promise_handle_t handle;
};

Task<int> task1() {
    std::cout << "task1 run" << std::endl;
    co_return 1;
}

Task<int> task2() {
    std::cout << "task2 run" << std::endl;
```

```
    co_return 2;
}

Task<int> call_task() {
    std::cout << "call_task" << std::endl;
    int data1 = co_await task1();
    std::cout << "call_task task1 data: " << data1 << std::endl;
    int data2 = co_await task2();
    std::cout << "call_task task2 data: " << data2 << std::endl;
    co_return data1 + data2;
}

int main() {
    Task<int> task = call_task();
    task.then([](int data) {
        std::cout << "call_task data: " << data << std::endl;
    });
    return 0;
}
```

The following result is returned:

```
call_task
task1 run
call_task task1 data: 1
task2 run
call_task task2 data: 2
call_task data: 3
```

You can add some logs to the functions in the source code, like in the previous demo, to understand the process. Alternatively, you can use a debugger to step through the code. In reality, this demo does not actually implement waiting and awakening. There is even some awakening logic that is not executed. Specifically, if these coroutine tasks need to be scheduled to other threads for execution, the concurrency safety of the internal data structures of these objects should also be considered. If you use `std::mutex` directly without proper handling, deadlocks are more likely to occur in this function than in normal ones. Additionally, this demo also illustrates that once you start using the coroutine asynchronous method of C++20, you should continuously modify the entire project's asynchronous functions from the entry point onwards.

If you have had the patience to read this far, you should now have a basic understanding of C++20 coroutines. With this article as a starting point, you can move on to reading more advanced articles. It is still challenging to write a coroutine framework for a production environment using the current support provided by the C++ standard library. I recommend reading the