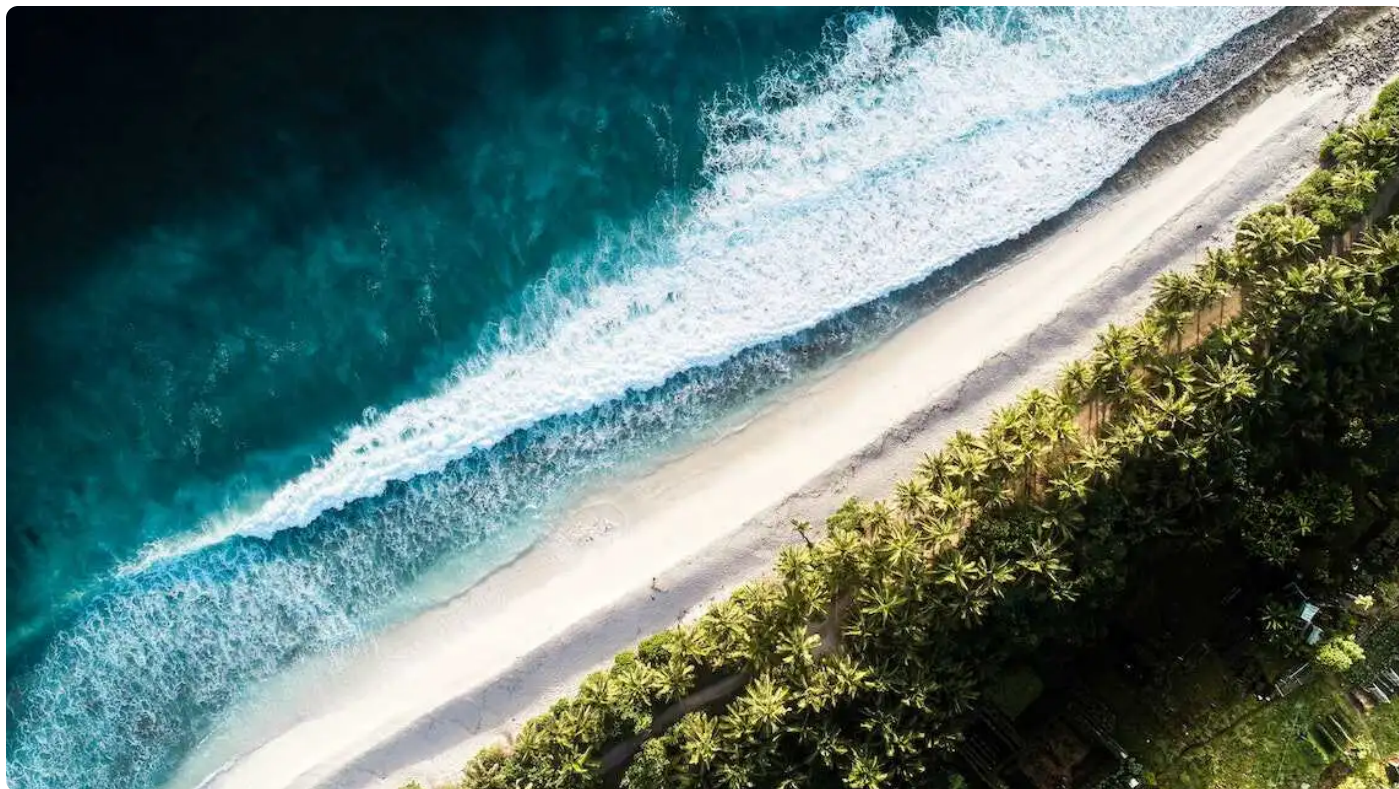


22 | 重大变更（二）：关于C++26的十大猜想

2023-03-13 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 13:59 大小 12.77M



你好，我是卢誉声。

上一讲，我们了解了后续 C++ 标准演进中，极有可能到来的特性或库变更——静态反射、异步任务框架、网络库和 Freestanding 库。

从未来标准的演进路线中，我们其实可以一窥究竟，不难发现，C++ 有着极为清晰的演进路线。

与此同时，C++ 标准还在努力弥补不足之处，比如缺乏标准化的高性能计算能力、异步编程库、针对大数据处理的 Ranges 扩展等等。接下来，就让我们继续漫游之旅，畅想未来 C++ 标准演进可能迎来的另外六个变化吧。

高性能计算支持

对于 C++26 来说，另一个重要目标是提供对高性能计算的支持。

不得不说，C++ 至今依然能焕发生机，得益于人工智能等领域对高性能计算的需求，C++ 几乎是唯一一个同时合理兼顾性能和抽象两个层面的编程语言。

但是，C++ 缺乏对高性能计算的标准支持，无论是基础的多维向量、对 CPU SIMD 的指令封装，还是更高层的线性代数算法都是一片空白。

因此，从 C++20 标准开始，不断有相关特性添加到标准库中，C++20 中的一维数组视图 `span` 和 C++23 中的多维数组视图 `mdspan`，都是线性代数的基础类型向量的前置特性。同时，C++23 的多元索引操作符，也让多维向量访问变得更加方便了。

在 C++26 中，预计添加许多**关键的线性代数支持**。

首先，可能会加入的就是多维数组 `mdarray`。多维数组 `mdarray` 的设计和 C++23 中的多维数组视图 `mdspan` 是一致的，都可以用于表示多维数组。我在这里着重说明它们之间的差异。

序号	对比项	工具	差异	具体说明
1	含义	<code>mdspan</code>	表示的是不拥有数据的引用视图	<code>mdspan</code> 只能包装调用者创建的数组，而 <code>mdarray</code> 会实际创建数组。而大部分情况下，我们肯定是要创建数组的，所以 <code>mdarray</code> 会更实用。同时， <code>mdarray</code> 也提供了接口返回对应的 <code>mdspan</code> ，因此在需要视图的场合也可以满足要求。
		<code>mdarray</code>	<code>mdarray</code> 拥有实际数据	
2	数据不可变性	<code>mdspan</code>	其自身不可修改，但是它引用的数据可以修改	如果 <code>mdspan</code> 是常量，那么只保证浅层常量性。也就是说，我们依然可以通过 [] 访问并修改其中的数据，只是 <code>mdspan</code> 自身不能被修改，但常量 <code>mdarray</code> 会和其他容器一样保证内部的数据也是常量不可修改，所以 <code>mdarray</code> 的各种访问函数也就包含 <code>const</code> 和非 <code>const</code> 的版本。
		<code>mdarray</code>	其自身和内部数据均不可修改	
3	类别	<code>mdspan</code>	视图	<code>mdspan</code> 是一个视图，因此需要通过 <code>AccessorPolicy</code> 控制访问策略，而 <code>mdarray</code> 本质是容器，因此需要通过指定内部容器适配器（ <code>Container Adaptor</code> ）控制访问策略。
		<code>mdarray</code>	容器	
4	移动语义	<code>mdspan</code>	不支持移动构造函数	由于 <code>mdspan</code> 自身不拥有数据，因此无需提供移动构造函数，只提供了拷贝构造函数和赋值操作符重载。它会将内部的数据指针、数组尺寸和布局信息拷贝到另一个 <code>mdspan</code> 对象。而 <code>mdarray</code> 会拥有数据，因此除了拷贝构造函数还提供了移动构造函数，可以转移内部数据的所有权。
		<code>mdarray</code>	支持移动构造函数	

另一个 C++26 可能支持的特性是**提供 SIMD 指令的封装**。

在高性能计算中，SIMD（单指令多数据流）是常见的计算优化处理模式，各个 CPU 指令集都有相应指令提供支持，其目标是通过一条 CPU 指令计算多个多维向量。

目前 C++ 编译器可能会在部分的计算优化中，自动使用这些指令。但是，在高性能计算中，我们往往需要手动指定使用这些指令。由于标准库没有提供这些接口，导致我们只能使用汇编指令或者类似 Intel 的 Intrinsics 接口来实现，经常需要做大量的平台适配工作。

为了解决这个问题，C++ 计划引入针对 SIMD 类型与指令的封装——这也是后续线性代数接口实现的计算基础。

最后，C++26 可能会基于 mdarray、mdspan 提供**线性代数函数库**。

由于 BLAS（Basic Linear Algebra Subprograms）库基本已经成为 C/C++ 中线性代数的事实标准，CBLAS、ATLAS 和 OpenBLAS 等基于 BLAS 接口的库，也成了大部分科学计算库的基础设施（比如 Python 著名的 NumPy 就是基于 BLAS 开发的）。

因此，C++ 的线性代数接口也就以 BLAS 接口为基础设计。C++ 的 BLAS 接口会使用 mdspan 描述向量数据类型，可能使用 SIMD 实现部分计算加速。至于与 BLAS 同时设计的 LAPACK，虽然补充了更多线性代数功能，但由于各种原因可能要留待以后的 C++ 标准决定何时实现。

Coroutines 扩展

C++20 中的 Coroutines 只提供了一套协议，具体实现需要开发者自己来实现。而 C++23 里的 generator 提供了在生成器这种场景下的 coroutine 标准实现。

而在 C++26 中，我们终于可以看到面向普通协程任务的 coroutine 实现——**std::lazy**。

那么什么是 std::lazy 呢？我们可以结合这段代码来理解。

 复制代码

```
1 #include <experimental/lazy>
2 #include <string>
3
```

```

4 struct Person {
5     int id;
6     std::string name;
7 };
8
9 std::lazy<Person> loadPerson(int id);
10 std::lazy<> savePerson(Person p);
11
12 std::lazy<void> modifyPerson() {
13     Person person = co_await loadPerson(1);
14     person.name = "学生名称";
15     co_await savePerson(person);
16 }

```

通过代码可以看到，自定义的 `coroutine` 换成 `lazy` 之后，我们不再需要自己实现协程的调度过程，直接通过 `co_await` 就能调用 `loadPerson` 和 `savePerson`，这可太棒了！

`std::lazy` 甚至可以允许我们指定具体的 `executor` 调度协程，比如后面的代码。

 复制代码

```

1 co_await f().via(e);

```

这里的 `e` 就是一个 `executor`，这样我们可以控制协程到底要通过哪个 `executor` 来唤醒执行，提供了更高的灵活性和更简单的接口。

说白了，`std::lazy` 就是一个通用协程任务封装，起了这么一个名字，是因为标准委员会的 SG1 倾向于将 `task` 这个名字保留给以后的其他概念使用，最后 LEGW（Library Evolution Work Group，即标准委员会负责库改进的工作组）就选用了 `lazy` 这个名字……

Ranges 扩展

到 C++23 为止，Ranges 已经相当完善了，唯一问题就是我们无法像 Python、JavaScript 和 Rust 一样，方便地将序列（包括 `vector`、`map` 和 `set`）打印到控制台上。

在 C++26 中，可能会提出容器和 `ranges` 的格式化接口方案，也就是我们可以写出这种代码：

 复制代码

```

1 #include <ranges>
2 #include <string>
3 #include <vector>

```



```

4 #include <stdint>
5 #include <fmt/ranges.h>
6
7 int main() {
8     std::vector<int32_t> v{
9         1, 2, 3
10    };
11    fmt::print("{}\n", v);
12
13    std::string s = "xyx";
14    auto parts = s | std::views::split('x');
15
16    fmt::print("{}\n", parts);
17    fmt::print("<<{}>>\n", fmt::join(parts, "--"));
18
19    return 0;
20 }

```

可以看出，相比原来的形式，这样的代码无论是调试代码，还是将容器 `ranges` 输出到文件中，编码都会更加方便。

Hive: Bucket Array 容器框架

如今，C++ 依然是高性能交易与游戏编程的核心支撑，在这些领域中，经常需要完成大量数据块的申请、释放与快速检索，我们经常采用一种名为 **Bucket array 的技术** 来解决此类问题。

这种数据结构的原理是，将一个元素数组划分成块的数组，每个块包含多个实际的元素，每个元素有一个布尔标记位，表示这个元素是否被删除了。当遍历元素时，会跳过这些被标记删除的元素，只有一个块中的所有元素被标记删除后，这个块才会被真正释放。这么做，可以避免遍历时遇到全空的块。插入元素时，所有块都满了才会申请新的块。

这种方式大幅度减少了小对象的内存申请（都以块为单位申请内容），避免过多零散的删除操作引起的移位拷贝，所以性能损耗大大降低。而且，还能确保很多相关元素连续存储在一个块中，保证很多场景下随机访问的性能。可以说，这是一种在特定场景下权衡了插入、删除和检索性能的数据结构。

由于在 C++ 的主要支持场景中，会大量用到这种数据结构，因此 C++ 标准提出了相关提案，可能会在 C++26 中正式纳入标准。

多线程无锁内存模型支持

基于多线程的高并发业务也是 C++ 的重要应用场景，尤其在现在的高并发 Web 服务中有大量应用。

在这种场景中，性能的核心瓶颈可能并非长时间的计算，而是持续的高并发访问。这种情况下，多个线程可能需要频繁地同时访问相同数据，自然就需要解决高并发场景下的数据竞争问题。

如果我们采用常用的锁来解决问题，极端情况下会将本可并行的执行流变成串行执行流，严重拖慢并发性能。

为此，有针对不同场景的大量“无锁”内存模型，Hazard Pointer 就是一种面向“单写者、多读者”场景的无锁内存模型与基础数据结构。

你不妨联想一下，在很多 Web 服务中我们可能都会采用类似“读写分离”的机制来处理数据的读写，Hazard Pointer 的原理也是一样的。这种数据结构的特点是，永远只有一个线程具备其所有权，并可以写入数据，其他线程只能访问并读取其中的值，所以我们用这种特性做无锁化的优化非常方便。


为了支持 Hazard Pointer，C++ 采用了 RCU（read copy update）实现了一种面向多读少写特性的链式数据结构，也就是我们熟知的“无锁队列”。

RCU 结合 Hazard Pointer 为我们提供了一种“单写者、多读者”的多读少写的高并发无锁内存模型，可以为我们解决特定的高并发业务提供基础设施支持。

定制点对象调整

最后我们聊聊定制点对象，这是为了解决 ADL（Argument-Dependent Lookup，实参依赖查找）问题提出来的。

本质上，引入它是为了让用户能方便地使用命名空间中的符号。我们还是结合代码示例理解。

 复制代码

```
1 #include <ranges>
2 #include <string>
3 #include <vector>
4 #include <cstdint>
5 #include <iostream>
```

```

6
7 namespace cp {
8     struct Person {
9         friend void swap(Person& lhs, Person& rhs);
10        int32_t id;
11        std::string name;
12    };
13
14    void swap(Person& lhs, Person& rhs) {
15        std::cout << "Person swap" << std::endl;
16
17        Person temp = lhs;
18        lhs = rhs;
19        rhs = temp;
20    }
21
22    std::ostream& operator<<(std::ostream& os, const Person& person) {
23        os << person.id << " " << person.name << std::endl;
24
25        return os;
26    }
27 }
28
29
30 int main() {
31     cp::Person p1{
32         .id = 1,
33         .name = "姓名1"
34     };
35
36     cp::Person p2{
37         .id = 2,
38         .name = "姓名2"
39     };
40
41     std::cout << p1 << std::endl;
42     // 通过p2找到cp::operator<<
43     operator<<(std::cout, p2) << std::endl;
44
45     // 通过p1, p2找到cp::swap
46     swap(p1, p2);
47
48     cp::operator<<(std::cout, p1) << std::endl;
49
50     return 0;
51 }

```

可以看到，在代码第 8 行定义了 **Person** 类，第 14 行定义了 **swap** 函数，第 22 行定义了 **<<** 的操作符重载。这些符号都被定义在命名空间 **cp** 中。

现在来看一下代码 41 和 43 行，我们都知道其实 C++ 的操作符重载只是一个语法糖，`std::cout << p1` 的本质相当于 `operator<<(std::cout, p1)`，所以 43 行的用法没有什么问题。

但是，代码 46 行就很奇怪了，我们分明没有 `using cp::swap`，也没有 `using namespace cp`，为什么这里 C++ 能找到 `cp::swap` 这个函数呢？

命名空间是为了解决名称隔离的问题引入的，但也让我们引用符号变得比较麻烦。比如说，我们为了避免命名空间污染，基本上不会使用 `using namespace`，而是在每次引用符号时，都加上完整的命名空间（比如使用 `cp::swap`）。

不过有些问题，我们必须交给编译器来解决。比如说，代码中的 `operator<<` 重载。既然 `std::cout << p1` 只是 `operator<<(std::cout, p1)` 的语法糖，而 `operator<<(std::ostream&, const Person&)` 是在 `cp` 这个命名空间中定义的函数，那么编译器要怎么找到 `cp::operator<<` 这个函数呢？

为此，C++ 提出了 ADL。简单来说，就是调用函数时，只要有一个参数的类型属于函数所在的命名空间，那么调用的时候就不用加命名空间前缀（当然实际情况肯定复杂得多...）。

这样一来，编译器就可以通过 `operator<<(std::cout, p1)` 将 `std::operator<<` 或者 `cp::operator<<` 作为候选符号，最后匹配满足调用条件的版本。

这个能力，最终在 C++ 标准中从操作符重载被扩展到了普通函数，就变成了我们现在所知的 ADL，这也解释了为什么代码 46 行能够编译成功。

不过，这种特性在后续过程中被“合法”运用到了 C++ 的其他部分，来实现用户自定义的扩展（想想模板元编程是怎么出现的）。比如 C++20 Ranges 中的 `begin`、`end` 等函数，都需要类似的特性。但这种场景下使用 ADL 并不在语言设计的预期中，自然会存在各种各样的问题。

因此，C++ 在标准中不断对它修修补补。CPO（定制点对象）就是其中一种。你可以这样理解，C++ 给我们提供了一个以特定对象作为扩展点，让开发者可以针对自己的类型定制对应的行为，这就是为什么叫做定制点。

不过 CPO 依然存在很多问题，在准备推出的 `execution` 中，就有很多符号需要此类支持以供第三方定制扩展。因此，C++ 本来准备引入 `tag_invoke` 来缓解这个问题，但这只不过又是一个给 ADL 收烂摊子的方案。

所幸，由于 `execution` 特性延迟，标准委员会才有机会在 `tag_invoke` 这种治标不治本的方案进入标准之前，在 **C++26** 之后（大概率在 **C++26**），彻底消除在扩展点中使用 **ADL** 带来的问题，最终完全统一扩展点的技术方案。

总结

人工智能技术发展和高性能计算领域，既需要接近硬件层，又需要提供上层接口。唯一一个兼顾性能和抽象两方面的编程语言，几乎只有 **C++**。

不过，虽然 **C++** 具备如此素质，但标准库一致缺乏对这些领域所依赖的底层技术的支持。因此工程师们被迫引入大量第三方库，但集成过程里的问题（比如不同库之间的适配问题、体系结构、操作系统的兼容性问题）也是层出不穷，令人头大。

如今，从 **C++11** 再到 **C++20** 以及后续演进标准，我们能清楚看到 **C++** 的改变以及演进路线，**C++** 标准将提供更多“标准化的解决方案”，这些发展有望大幅降低开发者编写代码的复杂度。

由此，我们有理由推测，之后 **C++** 在机器学习和高性能计算等重要应用领域中，会大放异彩，越战越勇。

我们在课程中曾自己实现了几乎所有的 **C++ Coroutines** 接口约定。但是可以看到，在后续的标准演进中，`coroutine` 库即将到来，届时，我们就可以通过 `std::lazy` 来彻底简化异步编程的工作。这可太让人期待了。

最后，**Ranges** 针对 `format` 的扩展也让代码调试与容器输出变得更加方便。

这些新特性的提出和对现有标准中的特性的补足，将会把 **C++** 推向一个全新的高度。**C++26** 或其后续标准，将是一个极为令人激动的重大变更。让我们一起期待它们的到来，以及编译器对新标准的支持吧。

课后思考

最后给你留两道思考题。

1. 在人工智能领域，**C++** 无处不在。那么，你能说出哪些技术或工具在使用 **C++** 呢？

2. 我在这一讲中提到了名为 **Bucket array** 的技术，用于解决内存碎片的问题。那么，你在日常工作中，是如何避免内存碎片导致的性能下降问题呢？

欢迎说出你的实践，与大家一起分享。我们一起讨论，下一讲见。

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。

上一篇 21 | 重大变更（一）：关于C++26的十大猜想

下一篇 23 | 未来展望：透过未来标准演进看C++设计哲学

精选留言 (1)

 写留言



peter

2023-03-14 来自北京

C++会用于网站后端开发吗？网站开发，一般也就是Java、PHP、微软ASP那一套、后来的python，基本就这些了。难道C++会被用来开发后端的某一个模块？

作者回复: 一般大多数互联网公司不会用C++编写增删改查这种业务代码，但是很多需要大量计算的任务会采用C++来编写，然后通过RPC/消息队列等方式与后台前端集成。或者可以通过互操作接口直接通过业务代码的语言做C++的wrapper（比如Python就很多，比如大多数深度学习框架）。

另外，网站系统的运行时无一例外，底层都是C/C++实现的。比如说 node 的 libuv、Java的底层实现等。

