12 | Ranges（二）：用“视图”破除函数式编程之困

2023-02-10 卢誉声 来自北京

讲述：卢誉声

时长 13:04 大小 11.94M

你好，我是卢誉声。

上一讲，我们重点讨论了 C++ 传统函数式编程的困境，介绍了 Ranges 的概念，了解到 range 可以视为对传统容器的一种泛化，都具备迭代器等接口。但与传统容器不同的是，range 对象不一定直接拥有数据。

在这种情况下，range 对象就是一个视图（view）。这一讲，我们来讨论一下视图，它是 Ranges 中提出的又一个核心概念，是 Ranges 真正解放函数式编程的重要驱动力（项目的完整代码，你可以这里获取）。

视图

视图也叫范围视图（range views），它本质是一种轻量级对象，用于间接表示一个可迭代的序列。Ranges 也为视图实现了视图的迭代器，我们可以通过迭代器来访问视图。

对于传统 STL 中大部分可接受迭代器参数的算法函数，在 C++20 中都针对视图和视图迭代器提供了重载版本，**比如 ranges::for_each 等函数，这些算法函数在 C++20 中叫做 Constraint Algorithm**。

那么 Ranges 库提供的视图有哪些呢？

我把视图类型和举例梳理了一张表格，供你参考。

视图	说明	举例
[begin, end)	通过一对迭代器创建视图，也支持通过容器 begin 和 end 迭代器为容器创建视图	views:all 函数返回的视图
[start, size)	通过容器或视图的起始位置和新视图的元素数量创建视图	views::counted 函数返回的视图
[start, predicate)	通过容器或视图的起始位置和序列终止条件创建视图	views::take_while 函数返回的视图
[start..)	未指定范围的视图	views::iota 函数返回的视图

所有的视图类型与函数，都定义在 std::ranges::views 命名空间中，标准库也为我们提供了 std::views 作为这个命名空间的一个别名，所以实际开发时我们可以直接使用 std::views。

后面是直接使用 std::views 的代码。后面我再解释 iota、take 的涵义，你可以先忽略这个细节。


```
#include <ranges>
#include <cstdint>
#include <iostream>

int main() {
    for (int32_t i : std::views::iota(1) | std::views::take(4)) {
        std::cout << i << " ";
    }

    return 0;
}
```

基础视图接口

了解了视图概念还不够，我们再聊聊 C++ 标准中基础接口的详细设计，以及自定义实现方法。

C++ Ranges 定义了一个标准接口 ranges::view_interface（本质是一个抽象类）。我们首先来看一下如何使用该类，自定义自己的视图类。

```
template <class Element, size_t Size>
class ArrayView : public std::ranges::view_interface<ArrayView<Element, Size>>
public:
    using Container = std::array<Element, Size>;

    ArrayView() = default;
    ArrayView(const Container& container) :
        _begin(container.cbegin()), _end(container.cend())
    {}

    auto begin() const {
        return _begin;
    }

    auto end() const {
        return _end;
    }

private:
    typename Container::const_iterator _begin;
    typename Container::const_iterator _end;
};
```

可以看到，代码中定义了 ArrayView 类，该类型表示 array 容器的视图。我们定义了三个成员函数。

- 构造函数：包含默认构造函数和通过 array 对象创建视图的构造函数。构造函数将 _begin 和 _end 初始化为 array 的 cbegin 和 cend。
- begin：返回 _begin，Ranges 可以通过该函数获取 begin 迭代器。
- end：返回 _end，Ranges 可以通过该函数获取 end 迭代器。

这样我们就可以将其作为视图来使用，你可以对照示例代码来理解。

```cpp
int main() {
    std::array<int, 4> array = { 1, 2, 3, 4 };
    ArrayView arrayView { array };

    for (auto v : arrayView) {
        std::cout << v << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

在这段代码中，创建 array 对象后创建视图，由于视图类中定义了 begin 和 end 成员函数，因此可以用 C++ 的 for 循环直接遍历这个视图。

除此以外，ranges::view_interface 中还定义了几个成员函数，当视图类满足特定约束时基类会提供默认实现，开发者必要时可以覆盖其实现，具体可以参考后面这张表。

成员函数	说明
empty	判断视图对象是否为空。当视图类满足 sized_range 和 forward_range 时，view_interface 会提供默认实现。
data	返回视图对象内部数据缓冲区的指针。 当视图类满足 contiguous_range 时 view_interface 会提供默认实现。
size	返回视图范围内的元素数量。当视图类满足 forward_range 并且其迭代器类型满足 sized_sentinel_for 约束时 view_interface 会提供默认实现。
front	返回视图的第一个元素。当视图类满足 forward_range 时 view_interface 会提供默认实现。
back	返回视图的第一个元素。当视图类满足 bidirectional_range 和 common_range 时 view_interface 会提供默认实现。
operator bool	判断视图是否不为空，当视图类实现 empty 成员函数时 view_interface 会提供默认实现。
operator []	获取视图的第 n 个元素，当视图类满足 random_access_range 成员函数时 view_interface 会提供默认实现。

从这里我们就可以看出视图的本质就是对一个可迭代序列的间接引用，视图自身不存储数据，只是引用了可迭代序列的一部分数据。

虽然 Ranges 提供了视图的基础接口。但总体来说，我们自己实现所有的视图接口可就太麻烦了。

因此，Ranges 提供了视图工厂和适配器，为我们提供了便利的构造视图的方法——这是我们使用视图的主要方法。接下来，我们就来仔细看一下视图工厂与适配器的细节。

工厂

视图工厂提供了一些常用的视图类，以及基于这些视图类构建视图对象的工具函数。

我们先来看一段代码，感性地认识一下如何使用视图工厂。

复制代码

```cpp
#include <array>
#include <ranges>
#include <iostream>
#include <sstream>

int main() {
    namespace ranges = std::ranges;
    namespace views = std::views;

    // iota_view与iota
    for (int32_t i : ranges::iota_view{ 0, 10 }) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    for (int32_t i : views::iota(1) | views::take(4)) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    // istream_view与istream
    std::istringstream textStream{ "hello, world" };
    for (const std::string& s : ranges::istream_view<std::string>{ textStream }
        std::cout << "Text: " << s << std::endl;
    }

    std::istringstream numberStream{ "3 1 4 1 5" };
    for (int n : views::istream<int32_t>(numberStream)) {
        std::cout << "Number: " << n << std::endl;
    }

    return 0;
}
```

在这段代码中，我们使用了两个视图工厂——ranges::iota_view 和 ranges::istream_view。

views::iota 是 ranges::iota_view 的工具函数，有了它，就能更方便地创建一个 iota_view 对象。调用 views::iota 时，我们也使用了视图适配器 views::take(4) 创建一个新视图，包含前一个视图的前 4 个元素。类似于 L | R 这种语法就是所谓的视图管道，允许我们将多个视图连接在一起，让分步的数据处理变得简洁优雅。

另一个视图工厂是 ranges::istream_view，作用是创建一个从输入流中不断读取数据的视图类。在遍历这个视图的时候，视图会尝试从输入流中读取数据，直到输入流结束为止。views::istream（也就是 ranges::istream_view 的工具函数），可以返回一个 istream_view 对象。

由此可见，使用视图工厂是非常简单的。后面表格里整理了 C++20 中提供的所有工厂，供你参考。

序号	工厂	说明
1	ranges::empty_view 或 views::empty	不包含元素的空视图。
2	ranges::single_view 或 views::single	只包含一个值（可以为任意类型）的视图。
3	ranges::iota_view 或 views::iota	创建一个由等差序列组成的视图。
4	ranges::istream _view 或 views::istream	创建的视图会不断从输入流中通过 >> 操作符读取输入作为元素。

除此之外，还有一些在 C++23 中提供的新视图工厂，待后续讨论 C++23 时我再介绍。

适配器

除了直接创建视图，Ranges 还提供了一系列工具，可以将一个或者多个视图转换成一个新的视图，用来支持数据处理和运算工作。

这些用视图作为参数的“工厂”就是视图适配器。比如说，上一节中用到的 views::take 返回的就是类型为 take_view 的视图适配器对象。

Ranges 也支持通过嵌套和组合的方式来使用视图适配器。后面的例子是一个典型的函数式编程案例。

复制代码

```
#include <vector>
#include <ranges>
#include <iostream>
#include <random>
#include <algorithm>

int main() {
    namespace ranges = std::ranges;
    namespace views = std::views;

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, 10);

    // 步骤6：输出
    ranges::for_each(
        // 步骤5：将键值对序列[(0,a1),(1,a2),...,(n, an)]转换为[0+a1,1+a2,...,n+an]
        views::transform(
            // 步骤4：选取结果键值对的至多前3个键值对（不足3个则全部返回）
            views::take(
                // 步骤3：从随机数键值对中筛选数值大于5的键值对
                views::filter(
                    // 步骤2：生成随机数键值对序列[(0,a1),(1,a2),...,(n, an)]
                    views::transform(
                        // 步骤1：生成序列[0,10)
                        views::iota(0, 10),
                        [&distrib, &gen](auto index) { return std::make_pair(in
                    ),
                    [](auto element) { return element.second > 5;  }
                ),
                3
            ),
            [](auto element) { return element.first + element.second; }
        ),
        [](auto number) {
            std::cout << number << " ";
        }
    );
    std::cout << std::endl;

    return 0;
}
```

这段代码是一个典型的函数式编程案例。你可以结合代码来理解这几个步骤。

第一步，使用 views::iota 生成一个[0,10) 的等差序列，相当于一个 range。

第二步，使用 views::transform 为等差序列中的每一个数生成一个随机数，返回一个由随机数键值对组成的序列，序列形式为[(0,a1),(1,a2),…,(n, an)]。

第三步，使用 views::filter 从随机数键值对序列中筛选所有随机数大于 5 的键值对，生成一个新的序列。

第四步，使用 views::take 从 filter 输出的键值对序列中选取前 3 个键值对，如果 filter 输出的数量不足 3 个则返回所有元素，也就是这里肯定不会产生越界。

第五步，使用 views::transform 将 take 返回的键值对序列[(0,a1),(1,a2),…,(n, an)]转换为[0+a1,1+a2,…,n+an]的求和序列。

第六步，使用 views::for_each 输出结果。

我们可以看出，整段代码是用嵌套函数的形式编写的，而且在 transform、filter 和 for_each 中，都使用了 Lambda 表达式作为高阶函数。这段编码已经非常简洁了，除了部分 C++ 无法避免的语法特性，可读性堪比其他函数式编程语言。

不过，如果你还不熟悉函数式编程，那么看到这种深度的括号嵌套应该会感到非常头痛。对此，我跟你说一个有关 Lisp 的地狱笑话。

> 某个程序员偷到了一个系统代码的最后一页，结果发现那一页上全部都是右括号。

或许，你现在应该理解了这个笑话的梗在哪里。

如果我们要用传统 STL 算法来实现这个函数式编程的过程，大概情况会是这样的。

复制代码

```
#include <vector>
#include <iostream>
#include <random>
#include <algorithm>
#include <numeric>

int main() {
```

```
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, 10);

    std::vector<int> rangeNumbers(10, 0);
    std::iota(rangeNumbers.begin(), rangeNumbers.end(), 0);

    std::vector<std::pair<int, int>> rangePairs;
    std::transform(rangeNumbers.begin(), rangeNumbers.end(), std::back_inserter
        return std::make_pair(index, distrib(gen));
    });

    std::vector<std::pair<int, int>> filteredPairs;
    std::copy_if(rangePairs.begin(), rangePairs.end(), std::back_inserter(filte
        return element.second > 5;
    });

    std::vector<std::pair<int, int>> leadingPairs;
    std::copy_n(filteredPairs.begin(), 3, std::back_inserter(leadingPairs));

    std::vector<int> resultNumbers;
    std::transform(leadingPairs.begin(), leadingPairs.end(), std::back_inserter
        return element.first + element.second;
    });

    std::for_each(resultNumbers.begin(), resultNumbers.end(), [](int number) {
        std::cout << number << " ";
    });

    return 0;
}
```

由于传统算法为了通用性，所以算法函数的输入都是迭代器，我们也就不得不创建大量的临时变量存储中间结果。有了对比，我们可以直观感受到，相比采用视图的方法，传统 STL 算法写的代码可读性就差了很多，而且也没有提供越界检查功能。

除了上述案例中用到的适配器，Ranges 还提供了大量的适配器。如果你感兴趣的话，可以查一下 <ranges> 头文件的文档，进一步了解所有的适配器。

视图管道

在实际编码时，虽然有适配器的帮助，但是大量的函数嵌套还是非常影响代码可读性。为此，C++ 还提供了视图管道（pipeline）来帮助我们更好地组织代码。比如前面的代码就还有改进空间，我们可以修改成后面这样。

复制代码

```cpp
#include <vector>
#include <ranges>
#include <iostream>
#include <random>
#include <algorithm>

int main() {
    namespace ranges = std::ranges;
    namespace views = std::views;

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1, 10);

    ranges::for_each(
        views::iota(0, 10) |
        views::transform([&distrib, &gen](int index) { return std::make_pair(in
        views::filter([](const auto& element) { return element.second > 5;  })
        views::take(3) |
        views::transform([](const auto& element) { return element.first + eleme
        [](int number) {
            std::cout << number << " ";
        }
    );
    std::cout << std::endl;

    return 0;
}
```

这个代码中，除了 ranges::for_each 属于算法函数，其他的嵌套视图都修改成了通过|这个视图管道操作符连接的形式。比如原本代码中的 transform(iota(), fn)，我们可以修改成 iota() | transform(fn) 这种形式。

所谓的视图管道依赖于 **Range 适配器对象（range adaptor object）**这个概念。简单来说，Range 适配器对象需要重载 operator() 操作符，并且满足后面这三个条件。

1. 参数列表为 (R, …args)。

2. 第一个参数是另一个 Range 适配器对象 R。

3. 可以存在后续参数…args，也可以不存在后续参数。

也就是说，Range 适配器对象必定是一个仿函数（functor）。由于第一个参数可以接收另一个适配器对象，因此我们可以像上一节中那样实现视图适配器的嵌套。**那么视图管道又是怎么实现的呢？**

首先，Range 适配器对象中有一个特例，就是如果后续参数...args 不存在时，我们就把这种适配器对象叫做 **range 适配器闭包对象（range adaptor closure object）**。假设有一个适配器闭包对象 C，其参数列表只有一个参数 R，并且 R 也是一个适配器对象，那么我们可以这样将两者嵌套调用。

```
C(R)
```

这时，C++ Ranges 就提供了视图管道，让我们可以将这种函数调用写成：

```
R | C
```

所以视图管道，本质上就是一个对 “range 适配器闭包对象函数调用”的语法糖。

那么，**普通的 Range 适配器对象如何转换成闭包对象呢**？很简单，只需要将除了第一个参数的后续参数...args 通过 binding 绑定上固定的参数，生成只有一个参数的偏函数就可以了。

此外，视图管道还可以复合使用，假设 R、C 和 D 都是 range 适配器闭包对象，我们就可以写成这样。

```
R | C | D
(R | C) | D
D(C(R))
```

这三者是完全等价的。所以可以看出 | 管道操作符的结合方向是自左向右结合。如果想要改变结合性，我们可以使用括号，后面这种形式代码就是等价的。

```
R | (C | D)
D(C)(R)
```

复制代码

这样一来，通过视图管道和视图适配器，我们就能组织出 C++ 中非常优雅的函数式代码了。

需要额外说明的是，C++20 中暂时只能使用标准库中定义的视图类型，我们自己哪怕实现了满足 range 适配器闭包对象的接口也无法在视图管道中使用，用户自定义的适配器闭包对象类型在 C++23 中才会得到支持。不过现阶段我们也有变通的方法可以将自定义的类型组合到视图管道中，我们将会在下一讲中具体讨论。

总结

通过两讲的内容，我们一起了解了 Ranges 的来龙去脉。

这一讲我们学习了 Ranges 的另一个重要概念——视图，我们通过它来间接引用特定范围的数据，而非拥有数据。在视图的基础上，通过视图工厂、视图适配器和视图管道，我们可以让复杂的数据处理变得简洁优雅。

我们还讨论了 range 适配器闭包对象，这种对象只需要满足后面三个条件中的一个。

1. 只有一个参数，参数类型为 Range 适配器对象。

2. 将另一个 range 适配器对象的后续参数…args 绑定固定参数后生成的仿函数（functor）。

3. 使用视图管道操作符 | 连接两个 range 适配器闭包对象后返回的对象。

课后思考

我们在讨论 Ranges 视图适配器的时候，曾提到除了课程里用到的适配器，Ranges 还提供了大量的适配器。你能否查阅相关文档进一步了解这些适配器，并结合一段简短的代码来展示其使用？

欢迎把你的代码贴出来，与大家一起分享。我们一同交流。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 1　提建议

上一篇　11丨Ranges（一）：数据序列处理的新工具

下一篇　13丨Ranges实战：数据序列函数式编程

精选留言 (3)

写留言

大熊猫有宝贝

2023-04-03 来自上海

工厂和工具函数之间的关系该怎么理解呢？

作者回复: 视图工厂包含：

（1）视图类：视图对象的类型

（2）工具函数：用于从特定的视图类型，创建视图对象

tang_ming_wu

2023-03-14 来自广东

对比函数式编程实现和传统编程实现，我个人觉得函数式编程只是伪需求和一小部分人的自嗨：（1）不方便调试（2）不方便设计解耦（3）不方便维护（4）不方便阅读理解。

作者回复: 首先，函数式编程本身并不是伪需求，如果你了解并习惯了函数式编程的一些基本原则和解决问题的思路，你会发现在处理特定问题时采用函数式编程的思路是很顺畅的。

其次，现在讨论函数式编程时很多人可能的确在“自嗨”，因为在我看来函数式编程并不比传统的过程化或者面向对象等范式更”高级”，我认为这些范式没有绝对优劣。以我个人为例，在处理复杂数据尤其是并行编程时，会优先选用函数式编程的思路组织计算过程，在实现函数细节时，我会更习惯使用过程化/结构化的思路，在组织整个系统的架构或对系统进行建模时，我可能更倾向于基于面向对象的设计方法并进行模块划分——在合理的场景使用每个人所认定的更合理的解决方案就行。计算机世界没有“银弹”，不同的技术或者方法论对我们来说都只是解决实际问题的一个“工具”，我们不需要教条

化，该用什么就用什么，仅此而已。

但是，我们依然需要去学习，去尝试使用一些不同的思路。由于我们大部分人入门时都以结构化/过程与面向对象为主要范式的语言开始学习，自然会习惯采用这些熟悉的思路去解决问题，一旦换了一种思维方式（比如函数式编程），一开始就会无所适从并且抵触。反之，如果我们一开始学习的不是C/C++/Java这类语言，而是LISP或者Erlang这类函数式语言，那么你可能会觉得用函数式的思路去解决问题是一个非常自然的事情，就和你现在习惯使用C++编写代码一样。

同时，我也不觉得函数式编程有什么地方会让我们不方便调试，反而由于习惯了变量的不可变性（无副作用），很多时候反倒容易在调试更快确定发生问题的位置。

至于设计解耦也和函数式编程完全没有关系。也许因为我的例子中只演示了如何使用函数式编程处理数据，所以你觉得函数式编程的耦合性很高。但耦合性更多体现在系统的模块划分和组织，而不是我们针对某一个数据的处理过程中。我们使用函数式编程来组织系统时，可以根据函数和业务的关系分析函数之间的内聚性，然后进行模块划分，并不会将函数都放在一个模块中，依然可以实现高内聚、低耦合的系统设计。我的示例中只是在实现一个算法，这些代码必然是要以一种高内聚的方式来组织的，而且你会发现数据处理代码的各个部分由于只关注如何解决一个问题，因此数据流的每一个处理函数反倒有更强的内聚性。

代码的可维护性就更取决于我们自己的系统设计和编码习惯，和函数式编程没有任何必然关系。这里也需要再提一下，函数式编程中的变量不可变性有时候对于代码维护反倒是有益处的。

说实话，我一开始也不习惯（自然也不认同）函数式编程的思路和理念， 但在实际工作中慢慢去尝试换一种思路解决问题后，突然发现自己已经非常习惯在数据处理过程中使用map/filter/reduce之类的高阶函数来组织代码，现在非常享受这种分离关注点、无副作用和函数幂等性带给我的“快乐”，也许这也不是纯粹的函数式编程，但在汲取其中一些核心理念后，我在处理复杂数据时的编码风格的确产生了很大改变，并且让我感到受益良多。


全程不笑

2023-02-10 来自广东

老师好，请教个问题，我的环境是ubuntu20.04， gcc版本11.1.0。
工厂小节中的示例代码编译报错了， istream_view与istream相关。
第25行的视图初始化应该为小括号吧，大括号{}我这边编译报错。另外28行的views::istream也编译报错，替换成ranges::istream_view运行正常，不知道是不是我环境的问题？

作者回复: 目前这个时间点，gcc 对新标准支持并不好。从支持程度上来说，Visual C++ > clang > gcc。

你可以尝试 Visual Studio 2022 来进行编译。另外建议gcc升级到12.2。