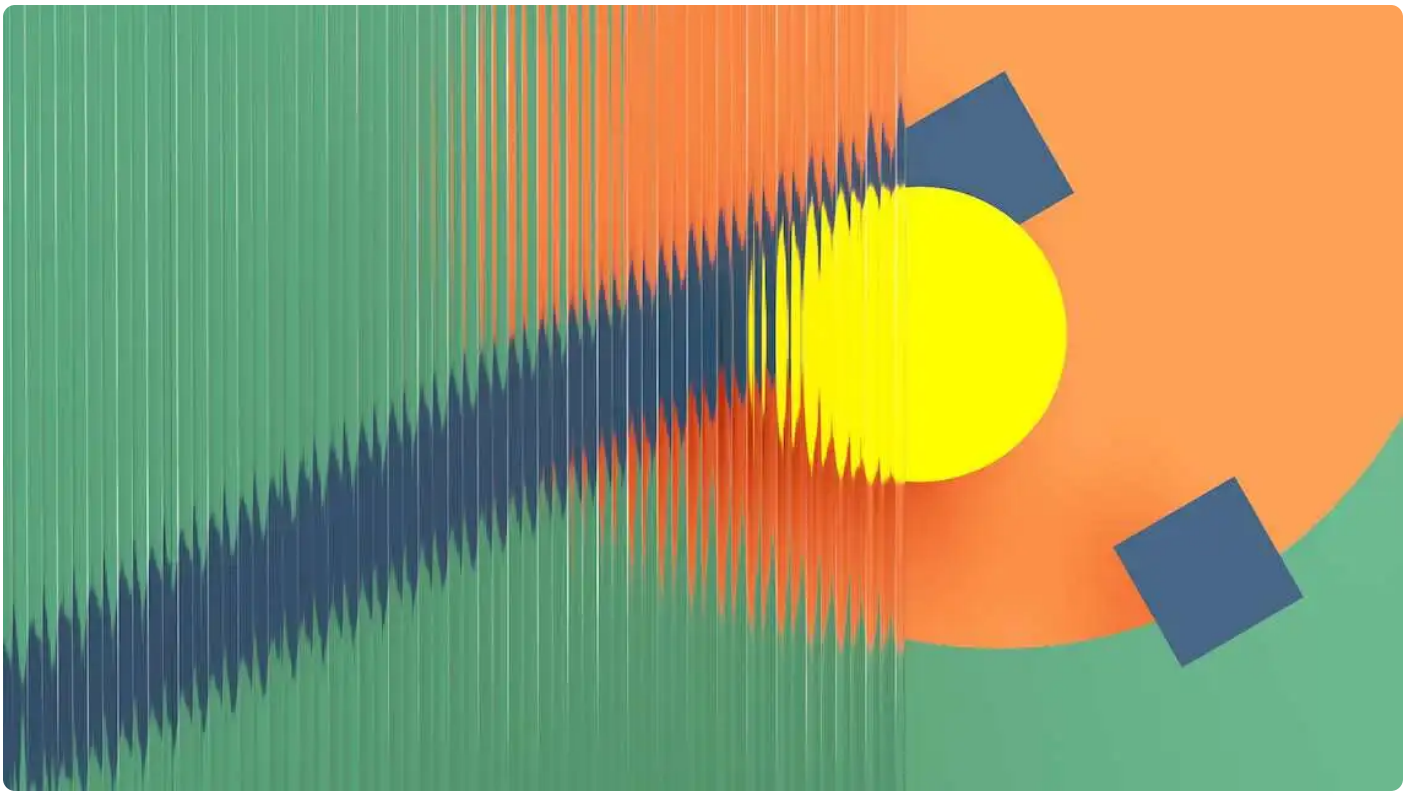
06 | Concepts实战：写个向量计算模板库

2023-01-27 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 20:07　大小 18.38M

你好，我是卢誉声。

Concept 之于 C++ 泛型编程，正如 class 之于 C++ 面向对象。在传统的 C++ 面向对象编程中，开发者在写代码之前，要思考好如何设计“类”，同样地，在 C++20 及其后续演进标准之后，我们编写基于模板技术的泛型代码时，必须先思考如何设计好“concept”。

具体如何设计呢？今天我们就来实战体验一下，使用 C++ 模板，编写一个简单的向量计算模板库。

在开发过程中，我们会大量使用 Concepts 和约束等 C++20 以及后续演进标准中的特性，重点展示如何基于模板设计与开发接口（计算上如何通过 SIMD 等指令进行性能优化不是关注的重心）。

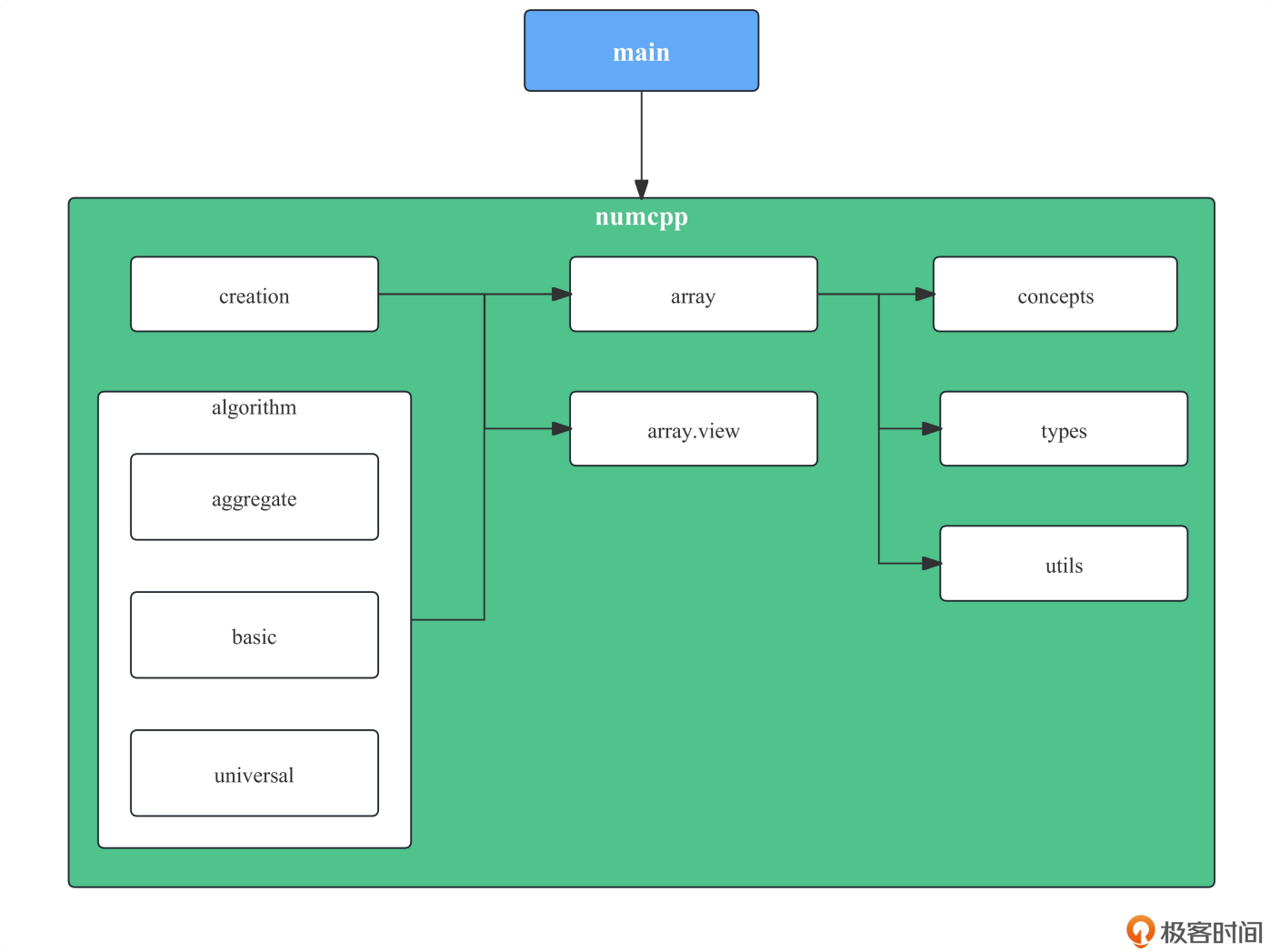
完成整个代码实现后，我们会基于今天的开发体验，对 Concepts 进行归纳总结，进一步深入理解（课程配套代码，点击这里即可获取）。

好，话不多说，我们直接开始。

模块组织方式

对于向量计算模板库这样一个项目，我们首先要考虑的就是如何组织代码。

刚刚学的 C++ Modules 正好可以派上用场，作为工程的模块组织方式。后面是项目的具体结构。



实现向量计算库的接口时，我们会部分模仿 Python 著名的函数库 NumPy。因此，向量库的模块命名为 numcpp，namespace 也会使用 numcpp。

先梳理一下每个模块的功能。

- main：系统主模块，调用 numcpp 模块完成向量计算；
- numcpp：工程主模块，负责导入其他分区并重新导出；
- numcpp:creation：向量创建模块，提供了基于 NDArray 类的向量创建接口；
- numcpp:algorithm：计算模块，负责导入各类算法子模块；
- numcpp:algorithm.aggregate：聚合计算模块，负责提供各类聚合计算函数。比如 sum、max 和 min 等聚合统计函数；
- numcpp:algorithm.basic：基础计算模块，负责提供各类基础计算函数。比如加法、减法和点积等聚合统计函数；
- numcpp:algorithm.universal：通用计算模块，负责提供各类通用计算函数。比如 reduce 和 binaryMap 等向量通用计算接口，aggregate 和 basic 中部分函数就会基于该模块实现。

有了清晰的模块划分，我们先从接口开始编写，再使用 Concepts 来约束设计的模板，这将是今天的学习重点。

接口设计

首先，我们来设计 numcpp 模块的接口模块 numcpp.cpp。


```
export module numcpp;

export import :array;
export import :array.view;
export import :creation;
export import :algorithm;
```

在这段代码中，我们使用 export module 定义模块 numcpp，然后使用 export import 导入几个需要导出的子分区，包括 array、array.view、creation 和 algorithm。

接下来，还要创建向量的相关接口。


```
auto arr1 = numcpp::zeros<int32_t>({1, 2, 3});
auto arr2 = numcpp::array({1, 2, 3});
auto arr3 = numcpp::array<std::initializer_list<std::initializer_list<int32_t>>
auto arr4 = numcpp::array<std::vector<std::vector<std::vector<int32_t>>>>({
```

```
    {{1, 2}, {3, 4}},
    {{5, 6}, {7, 8}}
});
auto arr5 = numcpp::ones<int32_t>({ 1, 2, 3, 4, 5 });
```


在这段代码中：

- 第 1 行，通过 zeros 生成三维向量，初始化列表中的参数{1, 2, 3}表示这个向量的 shape。其中，三个维度的项目数分别为 1、2 和 3，而 zeros 表示生成的向量元素初始值都是 0。
- 第 2 行，通过 array 创建了一维向量，初始化列表中的参数{1, 2, 3}表示这个向量的元素为 1、2 和 3。
- 第 3 行，通过 array 创建了二维向量，初始化列表中的参数{{1, 2}, {3, 4}}表示这个向量是一个 2 行 2 列的向量，第一行的值为 1 和 2，第二行的值为 3 和 4。
- 第 4 行，通过 array 创建了三维向量。初始化列表中的参数表示这个向量是一个 222 的向量。一共八个元素，按照排列顺序分别是 1、2、3、4、5、6、7 和 8。
- 第 8 行，通过 ones 创建了五维向量，初始化列表中的参数{1, 2, 3, 4, 5}表示这个向量的 shape。

接下来，是索引接口——用于从数组中获取到特定的值。

复制代码

```
int32_t value = arr1[{0, 1, 2}];
std::cout << "Value: " << value << std::endl;
```

这一小段代码中，通过[]来获取特定位置的元素，在{}中需要给定位置的准确索引。那么，这行代码的意思就是，获取 arr1 中三个维度分别为 0、1、2 位置的元素。

从功能上讲，numcpp 也支持为多维数组创建视图，所谓视图就是一个多维数组的一个切片，但是数据依然直接引用原本的数组，创建视图的方法是这样的。

复制代码

```
auto view1 = arr1.view(
    {0, 2}
);

```

```
auto view2 = arr3.view({
    {0, 1},
    {1, 2}
});
```


这段代码中：

- 第 1 行，调用 view 函数，在数组 arr1 中创建了一个子视图，从第一个维度切出了 0 到 2 这个区间，注意这里的区间是左闭右开。
- 第 5 行，类似地，调用 view 函数，在数组 arr3 中创建了一个子视图，第一个维度切出了 0 到 1，第二个维度切出了 1 到 2。由于 arr3 是一个 2 × 2 的向量，因此最后得到的视图是一个 2 × 1 的向量。

现在，我们继续为接口添加“基础计算”功能。

复制代码

```
auto arr6 = numcpp::array<std::vector<std::vector<int32_t>>>({ { 1, 2, 3 }, { 4
printArray(arr6);
auto arr7 = numcpp::array<std::vector<std::vector<double>>>({ { 3.5, 3.5, 3.5 }
printArray(arr7);

auto arr8 = arr6 + arr7;
printArray(arr8);
auto arr9 = arr6 - arr7;
printArray(arr9);
auto arr10 = numcpp::dot(arr6, arr7);
printArray(arr10);
```

在这段代码中，分别在第 6、78、10 行调用了向量基本运算符——向量加法、减法和点积。它们的计算结果分别是返回两个向量所有相同位置元素和、差和乘积，并生成一个新的向量。

需要注意的是，这里用 dot 而非重载 operator* 来实现向量点积。这是因为向量之间的乘法不止一种。我在这里定义一个 dot 函数是为了避免引起误解，也减少用户的记忆负担。

这些基础计算接口都要求两个输入向量的 shape 完全一致，如果不同会直接抛出参数异常。

接着，我们为接口添加“聚合计算”功能。

复制代码

```
double sum = numcpp::sum(arr10);
std::cout << "Array10 sum: " << sum << std::endl;

double maxElement = numcpp::max(arr10);
std::cout << "Array10 max: " << maxElement << std::endl;

double minElement = numcpp::min(arr10);
std::cout << "Array10 min: " << minElement << std::endl;
```

在这段代码中，我们分别在第 1 行、第 4 行和第 7 行调用了 sum、max、min 函数，计算向量中所有元素的和、最大值和最小值。由于这几个函数都是聚合计算，只会返回一个简单的浮点数。

现在，numcpp 的接口已经定义得差不多了。接下来，我们进入 numcpp 的内部实现细节，这将会涉及到重要的 C++ Modules 和 Concepts 的具体应用。

Concepts 设计

根据前面的接口定义，我们先来考虑有哪些 Concepts 需要被应用到实现部分的代码中，也就是 concepts.cpp 的编写，这是今天最关键的部分。

看这里的代码，我定义了所有为该工程服务的通用 concept。

复制代码

```
export module numcpp:concepts;

import <concepts>;
import <type_traits>;

namespace numcpp {
    // 直接使用type_traits中的is_integral_v进行类型判断
    template <class T>
    concept Integral = std::is_integral_v<T>;

    // 直接使用type_traits中的is_floating_v进行类型判断
    template <class T>
    concept FloatingPoint = std::is_floating_point_v<T>;

    // 约束表达式为Integral和FloatingPoint的析取式
    template <class T>
    concept Number = Integral<T> || FloatingPoint<T>;

    // 预先定义了IteratorMemberFunction这个类型，表示一个返回值为T::iterator，参数列表为
```

```
    template <class T>
    using IteratorMemberFunction = T::iterator(T::*)();

    // 用IteratorMemberFunction<T>(&T::begin)从T中获取成员函数的函数指针，
    // 使用decltype获取类型，判断该类型是否为一个成员函数
    template <class T>
    concept HasBegin = std::is_member_function_pointer_v<
        decltype(
            IteratorMemberFunction<T>(&T::begin)
            )
    >;

    // 使用IteratorMemberFunction<T>(&T::end)从T中获取成员函数的函数指针，
    // 使用decltype获取类型，判断该类型是否为一个成员函数。
    // 如果类型不包含end成员函数，或者end函数的函数签名不同，都会违反这个约束
    template <class T>
    concept HasEnd = std::is_member_function_pointer_v<
        decltype(
            IteratorMemberFunction<T>(&T::begin)
            )
    >;

    // 约束表达式为HasBegin和HasEnd的合取式
    template <class T>
    concept IsIterable = HasBegin<T> && HasEnd<T>;

    // 首先使用IsIterable<T>约束类型必须可遍历，
    // 再使用Number<typename T::value_type>约束类型的value_type，
    // 嵌套类型必须符合Number这个概念的约束，
    // 因此，约束表达式也是一个合取式
    template <class T>
    concept IsNumberIterable = IsIterable<T> && Number<typename T::value_type>;

    // 使用了requires表达式，采用std::common_type_t判断From和To是否有相同的类型，如果存在
    template <class From, class To>
    concept AnyConvertible = requires {
      typename std::common_type_t<From, To>;
    }
}
```

这段代码不多，但至关重要，演示了如何在工程代码中合适地使用 Concepts。我们定义了几个核心 concept。

- Integral：约束类型必须为整型。
- FloatingPoint：约束类型必须为浮点型。
- Number：约束类型必须为数字型，接受整型或浮点型的输入。

- HasBegin：约束类型必须存在 begin 这一成员函数。如果类型不包含 begin 成员函数，或者 begin 函数的函数签名不同，都会违反这个约束。
- HasEnd：约束类型必须存在 end 成员函数。
- IsIterable：约束类型是一个可以遍历的类型。
- IsNumberIterable：约束类型是一个可以遍历的类型，并且其值类型必须为数值类型。
- AnyConvertible：约束两个类型任一方向可以隐式转换。

好了，现在我们有了接口定义和 Concepts 定义。准备工作结束，接下来就是根据这些定义来实现具体的功能。我们会涵盖向量模块、构建模块、视图模块和计算模块。这些模块都是 C++20 标准后定义的标准 Modules，让我们先从向量模块的实现开始。

再说明一下，模块本身的功能不是今天的重点，所以我们只会在涉及到 Concepts 和 C++20 及其后续演进标准的部分着重讲解。

向量模块

向量模块是这个库的主要模块，主要定义了多维数组类型，而这个类型又使用了一些通用类型和通用工具函数，分别定义在 types 和 utils 分区中，所以，我们接下来就分别看看这些实现。

首先，我们在 types.cpp 中定义了部分基础类型。

复制代码

```
export module numcpp:types;

import <vector>;
import <cstdint>;
import <tuple>;

namespace numcpp {
    export using Shape = std::vector<size_t>;

    export class SliceItem {
    public:
        SliceItem(): _isPlaceholder(true) {}

        SliceItem(int32_t value) : _value(value) {
        }

```

```
17          int32_t getValue() const {
18              return _value;
19          }
20
21          bool isPlaceholder() const {
22              return _isPlaceholder;
23          }
24
25          std::tuple<int32_t, bool> getValidValue(size_t maxSize, bool isStart) c
26              int32_t signedMaxSize = static_cast<int32_t>(maxSize);
27
28              if (_isPlaceholder) {
29                  return std::make_tuple(isStart ? 0 : signedMaxSize, true);
30              }
31
32              if (_value > signedMaxSize) {
33                  return std::make_tuple(signedMaxSize, true);
34              }
35
36              if (_value < 0) {
37                  int32_t actualValue = maxSize + _value;
38
39                  return std::make_tuple(actualValue, actualValue >= 0);
40              }
41
42              return std::make_tuple(_value, true);
43          }
44
45      private:
46          int32_t _value = 0;
47          bool _isPlaceholder = false;
48      };
49
50      export const SliceItem SLICE_PLACEHOLDER;
51  }
```


在这段代码中，第 8 行定义了 Shape 类型，该类型是 std::vector<size_t> 的类型别名，用于描述多维数组每个维度的元素数量，vector 的长度也就是多维数组的维度数量。

接着，我们又定义了 SliceItem 类型，用于描述视图切片区间的元素，具体功能实现与 Python 中的类似。

接着是一个简单的工具 utils.cpp 的实现。


```
1  export module numcpp:utils;
2
```

```
import :types;

namespace numcpp {
    export size_t calcShapeSize(const Shape& shape) {
        if (!shape.size()) {
            return 0;
        }

        size_t size = 1;
        for (size_t axis : shape) {
            size *= axis;
        }

        return size;
    }
}
```


这个模块非常简单，目前只包含 calcShapeSize 函数，用于计算一个 shape 的实际元素总数，其数字为 shape 中所有维度元素数量之乘积。

有了基本工具后，我们必须先实现多维数组——这是向量计算的基本单位，并将其放入在 array 模块分区（源代码在 array.cpp 中）

复制代码

```
export module numcpp:array;
import <cstdint>;
import <cstring>;
import <vector>;
import <memory>;
import <algorithm>;
import <stdexcept>;
import <tuple>;
import <array>;
import :concepts;
import :types;
import :array.view;
import :utils;

namespace numcpp {
    export template <Number DType>
    class NDArray {
    public:
        using dtype = DType;
        NDArray(
            const Shape& shape,
            const DType* buffer = nullptr,
            size_t offset = 0
```

```
        ) : _shape(shape) {
            size_t shapeSize = calcShapeSize(shape);
            _data = std::make_shared<DType[]>(shapeSize);
            if (!buffer) {
                return;
            }
            memcpy(_data.get(), buffer + offset, shapeSize * sizeof(DType));
        }

        NDArray(
            const Shape& shape,
            const std::vector<DType>& buffer,
            size_t offset = 0
        ) : _shape(shape) {
            size_t shapeSize = calcShapeSize(shape);
            _data = std::make_shared<DType[]>(shapeSize);
            if (!buffer) {
                return;
            }
            if (offset >= buffer.size()) {
                return;
            }
            size_t actualCopySize = std::min(buffer.size() - offset, shapeSize)
            memcpy(_data.get(), buffer.data() + offset, actualCopySize * sizeof
        }

        NDArray(
            const Shape& shape,
            DType initialValue
        ) : _shape(shape) {
            size_t shapeSize = calcShapeSize(shape);
            _data = std::make_shared<DType[]>(shapeSize);
            std::fill(_data.get(), _data.get() + shapeSize, initialValue);
        }

        NDArray(
            const Shape& shape,
            std::shared_ptr<DType[]> data
        ) : _data(data), _shape(shape) {
        }

        const Shape& getShape() const {
            return _shape;
        }

        size_t getShapeSize() const {
            return calcShapeSize(_shape);
        }

        NDArray<DType> reshape(const Shape& newShape) const {
            size_t originalShapeSize = calcShapeSize(_shape);
            size_t newShapeSize = calcShapeSize(newShape);
```

```
76              if (originalShapeSize != newShapeSize) {
77                  return false;
78              }
79              return NDArray(newShape, _data);
80          }
81
82          DType& operator[](std::initializer_list<size_t> indexes) {
83              if (indexes.size() != _shape.size()) {
84                  throw std::out_of_range("Indexes size must equal to shape size
85              }
86              size_t flattenIndex = 0;
87              size_t currentRowSize = 1;
88              auto shapeDimIterator = _shape.cend();
89              for (auto indexIterator = indexes.end();
90                  indexIterator != indexes.begin();
91                  --indexIterator) {
92                  auto currentIndex = *(indexIterator - 1);
93                  flattenIndex += currentIndex * currentRowSize;
94                  auto currentDimSize = *(shapeDimIterator - 1);
95                  currentRowSize *= currentDimSize;
96                  -- shapeDimIterator;
97              }
98              return _data.get()[flattenIndex];
99          }
100
101         DType operator[](std::initializer_list<size_t> indexes) const {
102             if (indexes.size() != _shape.size()) {
103                 throw std::out_of_range("Indexes size must equal to shape size
104             }
105             size_t flattenIndex = 0;
106             size_t currentRowSize = 1;
107             auto shapeDimIterator = _shape.cend();
108             for (auto indexIterator = indexes.end();
109                 indexIterator != indexes.begin();
110                 --indexIterator) {
111                 auto currentIndex = *(indexIterator - 1);
112                 flattenIndex += currentIndex * currentRowSize;
113                 auto currentDimSize = *(shapeDimIterator - 1);
114                 currentRowSize *= currentDimSize;
115                 --shapeDimIterator;
116             }
117             return _data.get()[flattenIndex];
118         }
119
120         NDArrayView<DType> view(std::tuple<SliceItem, SliceItem> slice) {
121             return NDArrayView<DType>(_data, _shape, { slice });
122         }
123
124         NDArrayView<DType> view(std::initializer_list<std::tuple<SliceItem, Sli
125             return NDArrayView<DType>(_data, _shape, slices);
126         }
127
```

```
128         NDArrayView<DType> view(std::initializer_list<std::tuple<SliceItem, Sli
129             return NDArrayView<DType>(_data, _shape, slices);
130         }
131 
132         const std::shared_ptr<DType[]>& getData() const {
133             return _data;
134         }
135 
136         std::shared_ptr<DType[]>& getData() {
137             return _data;
138         }
139 
140         NDArray<DType> clone() {
141             size_t shapeSize = calcShapeSize(_shape);
142             std::shared_ptr<DType[]> newData = std::make_shared<DType[]>(shapeS
143             memcpy(newData.get(), _data.get(), shapeSize);
144             return NDArray<DType>(_shape, newData);
145         }
146 
147     private:
148         std::shared_ptr<DType[]> _data;
149         Shape _shape;
150 
151 
```


从这段代码开始，开始使用前面定义的 concept，我们重点看。

第 16 行，定义了 NDArray 类型。这个类型是一个类模板，**模板参数 DType 使用了名为 Number 的 concept**。NDArray 包含两个属性。

- _data，其类型为 shared_ptr 智能指针，通过引用计数来避免执行多维数组之间的拷贝，几乎没有性能损耗。如果真的想要复制一份新的数据，需要调用一百四十行的 clone 成员函数生成一个真正的拷贝。
- _shape，其类型为我们在之前定义的 Shape，用于描述多维数组每个维度的元素数量。

第 148 行，我们定义了一个类型为 DType[]的智能指针，这是从 C++20 开始支持的一个新特性。

由于 C++ 中 new T 和 new T[] 返回的指针，需要通过不同的方式释放内存（前者需要通过 delete，后者需要通过 delete [] 释放）。在 C++20 之前，开发者需要向智能指针传递一个 deleter 函数，告知智能指针如何释放内存。而 C++20 之后，编译器会自动根据传入的类型，确定使用哪种方式释放内存，这也是 type_traits 带来的便利，因为我们可以在模板类型中确定用户传入的类型是否是一个数组。

构建模块

实现了向量模块之后，我们来看构建模块的具体实现。

构建模块实现在:creation 分区中，creation.cpp 中的代码如下所示。

复制代码

```
export module numcpp:creation;
import :array;
import :concepts;

import <cstring>;
import <memory>;

namespace numcpp {
    // 使用了名为IsNumberIterable的concept，用于获取不包含子数组的数组的元素数量
    export template <IsNumberIterable ContainerType>
    void makeContainerShape(Shape& shape, const ContainerType& container) {
        shape.push_back(container.size());
    }

    // 使用了名为IsIterable的concept，用于获取不满足IsNumberIterable约束的数组的元素数量
    export template <IsIterable ContainerType>
    void makeContainerShape(Shape& shape, const ContainerType& container) {
        shape.push_back(container.size());
        makeContainerShape(shape, *(container.begin()));
    }

    /*
     * 用于帮助调用者获取一个多维容器类型的实际元素类型
     * 该结构体定义也是一个递归定义
```

```
 */

// 如果第34行或第40行都不匹配，编译器会选用这一默认版本
export template <typename>
struct ContainerValueTypeHelper {
};

// 当模板参数类型符合IsNumberIterable这一concept的时候会选用这一版本
export template <IsNumberIterable ContainerType>
struct ContainerValueTypeHelper<ContainerType> {
    using ValueType = ContainerType::value_type;
};

// 当模板参数类型符合IsIterable这一concept的时候会选用这一版本
export template <IsIterable ContainerType>
struct ContainerValueTypeHelper<ContainerType> {
    using ValueType = ContainerValueTypeHelper<
        typename ContainerType::value_type
    >::ValueType;
};


/*
 * fillContainerBuffer成员函数
 * 该成员函数有两个重载版本，
 * 负责将多维容器中的数据拷贝到多维数组对象的数据缓冲区中
 */

// 通过IsNumberIterable这一concept来约束调用该版本的参数必须是元素类型为Number的可迭代
export template <IsNumberIterable ContainerType>
typename ContainerType::value_type* fillContainerBuffer(
    typename ContainerType::value_type* dataBuffer,
    const ContainerType& container
) {
    using DType = ContainerType::value_type;

    DType* nextDataBuffer = dataBuffer;
    for (auto it = container.begin();
        it != container.end();
        ++it) {
        *nextDataBuffer = *it;
        ++nextDataBuffer;
    }

    return nextDataBuffer;
}

// 通过IsIterable这一concept来约束调用该版本的参数必须是可迭代容器
// 由于存在IsNumberIterable的版本，因此如果容器元素类型为Number，则不会匹配该版本
export template <IsIterable ContainerType>
typename ContainerValueTypeHelper<ContainerType>::ValueType* fillContainerB
    typename ContainerValueTypeHelper<ContainerType>::ValueType* dataBuffer
```

```
77          const ContainerType& container
78      ) {
79          using DType = ContainerValueTypeHelper<ContainerType>::ValueType;
80
81          DType* nextDataBuffer = dataBuffer;
82          for (const auto& element : container) {
83              nextDataBuffer = fillContainerBuffer(nextDataBuffer, element);
84          }
85
86          return nextDataBuffer;
87      }
88
89      export template <IsIterable ContainerType>
90      NDArray<typename ContainerValueTypeHelper<ContainerType>::ValueType> array(
91          const ContainerType& container
92      ) {
93          Shape shape;
94          makeContainerShape(shape, container);
95          size_t shapeSize = calcShapeSize(shape);
96
97          using DType = ContainerValueTypeHelper<ContainerType>::ValueType;
98          auto dataBuffer = std::make_shared<DType[]>(shapeSize);
99          fillContainerBuffer(dataBuffer.get(), container);
100
101         return NDArray<DType>(shape, dataBuffer);
102     }
103
104     export template <Number DType>
105     NDArray<DType> array(
106         const std::initializer_list<DType>& container
107     ) {
108         Shape shape;
109         makeContainerShape(shape, container);
110         size_t shapeSize = calcShapeSize(shape);
111
112         using ContainerType = std::initializer_list<DType>;
113         auto dataBuffer = std::make_shared<DType[]>(shapeSize);
114         fillContainerBuffer(dataBuffer.get(), container);
115
116         return NDArray<DType>(shape, dataBuffer.get());
117     }
118
119     export template <Number DType>
120     NDArray<DType> zeros(const Shape& shape) {
121         return NDArray<DType>(shape, 0);
122     }
123
124     export template <Number DType>
125     NDArray<DType> ones(const Shape& shape) {
126         return NDArray<DType>(shape, 1);
127     }
128
```

这段代码中，你可以重点关注第 17 行，我们利用了模板约束的偏序特性，实现了一个递归的 makeContainerShape 函数，并定义了函数的终止条件。这也是 C++ 模板元编程中递归函数的常见实现方式。只不过，相比传统的 SAFINE 方式，concept 为我们提供了更清晰简洁的实现方式。

视图模块

构建模块实现完后，我们来看视图模块的具体实现。

对于一个向量计算库来说，很多时候都需要从多维数组中进行灵活地切片，并生成多维数组的视图。这个时候，就需要数组视图的功能，这里我们在 array_view.cpp 中实现了 array.view 模块，代码如下所示。

复制代码

```
export module numcpp:array.view;

import <memory>;
import <stdexcept>;
import <iostream>;
import <algorithm>;
import :concepts;
import :types;

namespace numcpp {
    export template <Number DType>
    class NDArrayView {
    public:
        NDArrayView(
            std::shared_ptr<DType[]> data,
            Shape originalShape,
            std::vector<std::tuple<SliceItem, SliceItem>> slices
        ) : _data(data), _originalShape(originalShape), _slices(slices) {
            this->generateShape();
        }

        std::shared_ptr<DType[]> getData() const {
            return _data;
        }

        DType& operator[](std::initializer_list<size_t> indexes) {
            if (indexes.size() != _shape.size()) {
                throw std::out_of_range("Indexes size must equal to shape size
            }

            size_t flattenIndex = 0;
            size_t currentRowSize = 1;
```

```
            auto shapeDimIterator = _shape.cend();
            auto originalShapeDimIterator = _originalShape.cend();

            for (auto indexIterator = indexes.end();
                indexIterator != indexes.begin();
                --indexIterator) {
                auto currentIndex = *(indexIterator - 1);
                auto currentDimOffset = *(originalShapeDimIterator - 1);
                flattenIndex += (currentDimOffset + currentIndex) * currentRowS

                auto currentDimSize = *(shapeDimIterator - 1);
                currentRowSize *= currentDimSize;
                --shapeDimIterator;
            }

            return _data.get()[flattenIndex];
        }

        bool isValid() const {
            return _isValid;
        }

        const Shape& getShape() const {
            return _shape;
        }

    private:
        void generateShape() {
            _isValid = true;
            _shape.clear();
            _starts.clear();

            auto originalShapeDimIterator = _originalShape.begin();

            for (const std::tuple<SliceItem, SliceItem>& slice : _slices) {
                auto originalShapeDim = *originalShapeDimIterator;

                SliceItem start = std::get<0>(slice);
                SliceItem end = std::get<1>(slice);

                auto [actualStart, startValid ] = start.getValidValue(originalS
                auto [actualEnd, endValid] = end.getValidValue(originalShapeDim

                if ((!startValid && !endValid) ||
                    actualStart > actualEnd
                ) {
                    _isValid = false;

                    break;
                }

                if (actualStart < 0) {
```

```
                    actualStart = 0;
                }

                _shape.push_back(static_cast<size_t>(actualEnd - actualStart));
                _starts.push_back(static_cast<size_t>(actualStart));

                ++originalShapeDimIterator;
            }
        }

    private:
        std::shared_ptr<DType[]> _data;
        Shape _originalShape;
        std::vector<std::tuple<SliceItem, SliceItem>> _slices;
        Shape _shape;
        std::vector<size_t> _starts;
        bool _isValid = false;
    };
}
```

这段代码没有使用 concept，但使用了 Modules，理解它对理解视图很有帮助，因此我们简单看下。

类成员函数 _data 和 _originalShape 分别来源于原数组的数据指针和 Shape，这样在原数组的数据发生变化时，视图依然可以引用相关数据，毕竟视图的本质就是数组的引用，所以存储数据的引用也是合情合理的。_slices 用于生成该视图的切片数据。_shape、_starts 是根据多维数组原始 shape 和切片综合计算得到的新视图的 shape，以及视图相对于原数组在各个维度上的起始索引。

计算模块

了解了向量模块、构建模块和视图模块的实现，我们最后讲解一下计算模块。

计算模块中主要实现了各类算法，算法分为基础算法、聚合算法和通用算法，模块的接口代码实现在 algorithm/algorithm.cpp，主要导入并重新导出了所有的子模块。因此，我们有了如下所示的模块设计。

复制代码

```
export module numcpp:algorithm;

export import :algorithm.basic;
export import :algorithm.aggregate;
```

```
export import :algorithm.universal;
```

基础计算

首先，我们看一下基础算法的实现，基础算法的实现在 algorithm/basic.cpp 中。后面是具体代码。

复制代码

```
export module numcpp:algorithm.basic;

import <memory>;
import <stdexcept>;
import <type_traits>;

import :types;
import :concepts;
import :array;
import :utils;

namespace numcpp {
    export template <Number DType1, Number DType2>
        requires (AnyConvertible<DType1, DType2>)
    NDArray<std::common_type_t<DType1, DType2>> operator+(
        const NDArray<DType1>& lhs,
        const NDArray<DType2>& rhs
    ) {
        using ResultDType = std::common_type_t<DType1, DType2>;

        std::shared_ptr<DType1[]> lhsData = lhs.getData();
        Shape lhsShape = lhs.getShape();

        std::shared_ptr<DType2[]> rhsData = rhs.getData();
        Shape rhsShape = rhs.getShape();

        if (lhsShape != rhsShape) {
            throw std::invalid_argument("Lhs and rhs of operator+ must have the
        }

        size_t shapeSize = calcShapeSize(lhsShape);
        std::shared_ptr<ResultDType[]> resultData = std::make_shared<ResultDTyp
        ResultDType* resultDataPtr = resultData.get();
        const DType1* lhsDataPtr = lhsData.get();
        const DType2* rhsDataPtr = rhsData.get();

        for (size_t datumIndex = 0; datumIndex != shapeSize; ++datumIndex) {
            resultDataPtr[datumIndex] = lhsDataPtr[datumIndex] + rhsDataPtr[dat
        }

        return NDArray(lhsShape, resultData);
```

```
    }

    export template <Number DType1, Number DType2>
        requires (AnyConvertible<DType1, DType2>)
    NDArray<std::common_type_t<DType1, DType2>> operator-(
        const NDArray<DType1>& lhs,
        const NDArray<DType2>& rhs
        ) {
        using ResultDType = std::common_type_t<DType1, DType2>;

        std::shared_ptr<DType1[]> lhsData = lhs.getData();
        Shape lhsShape = lhs.getShape();

        std::shared_ptr<DType2[]> rhsData = rhs.getData();
        Shape rhsShape = rhs.getShape();

        if (lhsShape != rhsShape) {
            throw std::invalid_argument("Lhs and rhs of operator+ must have the
        }

        size_t shapeSize = calcShapeSize(lhsShape);
        std::shared_ptr<ResultDType[]> resultData = std::make_shared<ResultDTyp
        ResultDType* resultDataPtr = resultData.get();
        const DType1* lhsDataPtr = lhsData.get();
        const DType2* rhsDataPtr = rhsData.get();

        for (size_t datumIndex = 0; datumIndex != shapeSize; ++datumIndex) {
            resultDataPtr[datumIndex] = lhsDataPtr[datumIndex] - rhsDataPtr[dat
        }

        return NDArray(lhsShape, resultData);
    }
}
```


我们在代码中实现了向量加法和向量减法。仔细观察两个函数的声明，你会发现，我们除了在模板参数列表中使用 Number 来限定 T1 和 T2 的基本类型，**还在参数列表后使用了 requires 子句**——要求 T1 和 T2 必须是可以相互转换的数值类型，才能进行算术运算。

聚合计算

接下来，我们来看一下聚合算法的实现，聚合算法实现在 algorithm/aggreagte.cpp 中。该模块实现了 sum 和 max 函数，分别用于求一个向量中所有元素的和，以及一个向量中所有元素的最大值。

由于并不涉及有关 concept 的代码逻辑，为了让你聚焦主线，代码实现部分我们省略一下，这部分你可以参考完整的项目代码。

通用函数

通用函数是为用户对向量执行计算提供一个计算框架。在基础计算和聚合计算中我们看到了两类通用的计算需求。

1. 基础计算中对两个向量的元素逐个计算转换，生成新的计算结果并生成新的向量，新向量的 shape 和输入向量是保持一致的，我们将这种计算需求称之为 binaryMap（二元映射）。

2. 聚合计算中对一个向量中的元素逐个计算，处理各个元素的时候还需要考虑前面几个元素的处理结果，最后返回聚合计算的结果，这种计算需求我们称之为 reduce。

对这两个通用函数的实现在 algorithms/universal.cpp 中。

复制代码

```
export module numcpp:algorithm.universal;

import <functional>;
import <numeric>;
import :types;
import :concepts;
import :array;
import :utils;

namespace numcpp {
    export template <Number DType>
    using ReduceOp = std::function<DType(DType current, DType prev)>;

    export template <Number DType>
    DType reduce(
        const NDArray<DType>& ndarray,
        ReduceOp<DType> op,
        DType init = static_cast<DType>(0)
    ) {
        using ResultDType = DType;

        std::shared_ptr<DType[]> data = ndarray.getData();
        Shape shape = ndarray.getShape();

        const DType* dataPtr = data.get();
        size_t shapeSize = calcShapeSize(shape);

        return std::reduce(
            dataPtr,
            dataPtr + shapeSize,
            init,
            op
```

```
33          );
34      }
35
36      export template <Number DType1, Number DType2>
37          requires (AnyConvertible<DType1, DType2>)
38      using BinaryMapOp = std::function<
39          std::common_type_t<DType1, DType2>(DType1 current, DType2 prev)
40      >;
41
42      export template <Number DType1, Number DType2>
43          requires (AnyConvertible<DType1, DType2>)
44      NDArray<std::common_type_t<DType1, DType2>> binaryMap(
45          const NDArray<DType1>& lhs,
46          const NDArray<DType2>& rhs,
47          BinaryMapOp<DType1, DType2> op
48      ) {
49          using ResultDType = std::common_type_t<DType1, DType2>;
50
51          std::shared_ptr<DType1[]> lhsData = lhs.getData();
52          Shape lhsShape = lhs.getShape();
53
54          std::shared_ptr<DType2[]> rhsData = rhs.getData();
55          Shape rhsShape = rhs.getShape();
56
57          if (lhsShape != rhsShape) {
58              throw std::invalid_argument("Lhs and rhs of operator+ must have the
59          }
60
61          size_t shapeSize = calcShapeSize(lhsShape);
62          std::shared_ptr<ResultDType[]> resultData = std::make_shared<ResultDTyp
63          ResultDType* resultDataPtr = resultData.get();
64          const DType1* lhsDataPtr = lhsData.get();
65          const DType2* rhsDataPtr = rhsData.get();
66
67          for (size_t datumIndex = 0; datumIndex != shapeSize; ++datumIndex) {
68              resultDataPtr[datumIndex] = op(lhsDataPtr[datumIndex], rhsDataPtr[d
69          }
70
71          return NDArray(lhsShape, resultData);
72      }
73  }
```


在这段代码中，第 36 行定义了 BinaryMap 操作所需的函数类型，BinaryMap 函数需要的是两个序列相同位置的两个元素，并计算返回一个数值。在这里，我们**通过 requires (AnyConvertible<DType1, DType2>) 这一约束表达式**进行约束。

第 42 行定义了 binaryMap 函数。这个函数的内部实现和基础计算模块中的加法减法是一样的，只不过最后加减法改成了调用 op 而已。这里我们用跟第 36 行一样的约束表达式对函数

进行约束。

深入理解 Concepts

好的 concept 设计可以从根本上，解决 C++ 泛型编程中缺乏好的接口定义的问题。因此，在学习了实际工程中设计和使用了 Concepts 的方法后，我们有必要探讨一下，什么才是好的 concept 设计？

对比有助于我们加深理解，先看看我们所熟悉的面向对象编程的情况。在类的设计中，我们经常会提到三个基本特性：封装、继承与多态。在 C++ 中使用面向对象的思想设计类时，需要考虑如何通过组合或继承的方式来提升类的复用性，同时通过继承和函数重载实现面向对象的“多态”特性。

而这些问题和思想在 Concepts 和泛型编程中也同样存在。

首先，我们需要考虑**通过组合来提升 concept 的复用性**。在这一讲中，我们先定义了 Integral 和 FloatingPoint 这两个基本 concept，然后通过组合定义了 Number 这一 concept。

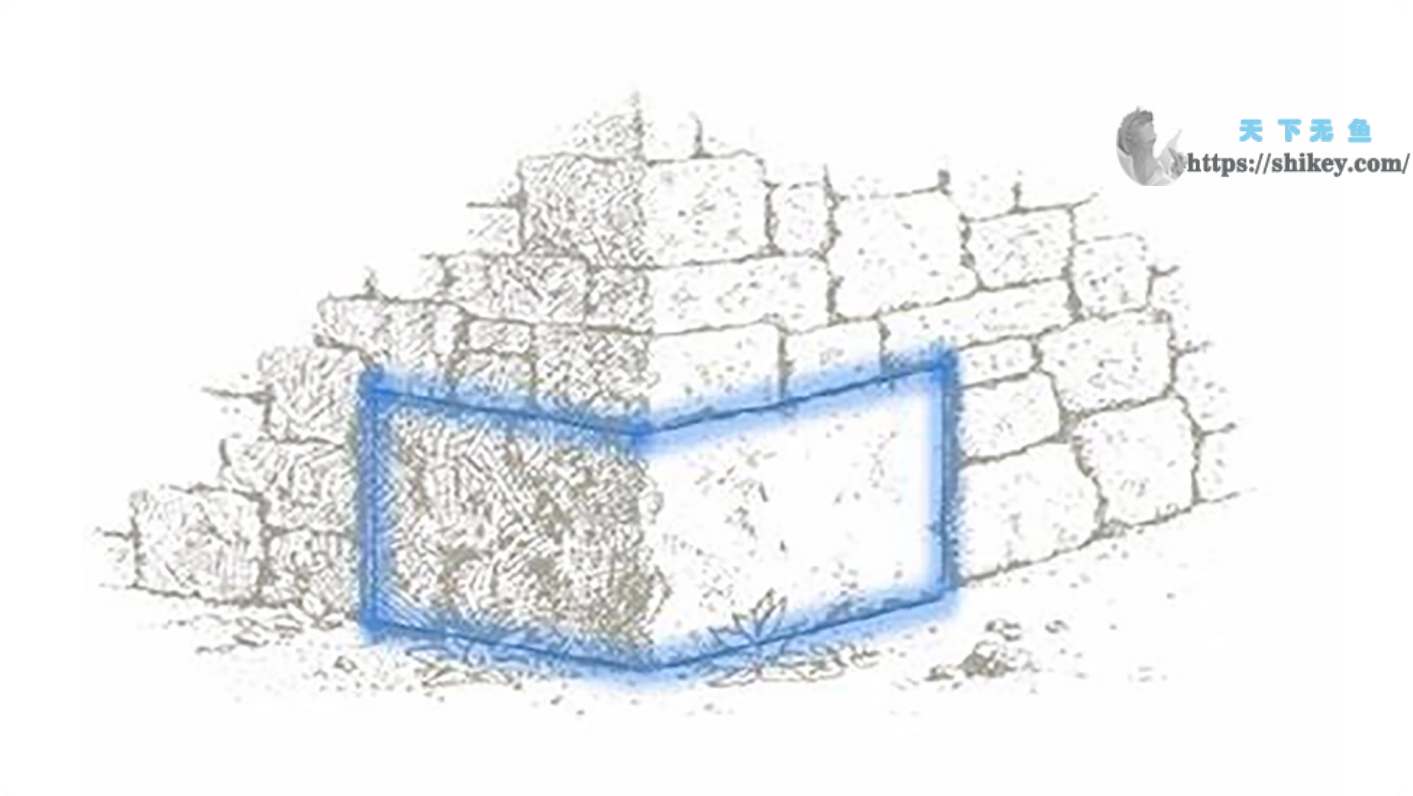
作为类比，考虑面向对象的思路设计类时，我们可能也会先设计一个 Number 类，然后设计继承 Number 类的 Integral 和 FloatingPoint 类。也就是说，在面向对象思想中，公有继承包含了 is-a 这个隐喻。

那么在泛型编程的 concept 组合中，我们不也包含了 is-a 的隐喻？只不过是倒过来的，Integral is a Number, FloatingPoint is a Number，同理于 IsNumberIterable 和 IsIterable。所以，组合与继承并非面向对象的“专利”——我们可以在泛型编程中，使用组合与继承来实现面向模板的类型，也就是 Concepts。

其次，concept 的设计也使得泛型编程能够**更好地具备“多态”的特性**。

作为类比，考虑面向对象的思路设计类时，对一个 Integral 的 print 函数和一个 Number 类的 print 函数，我们可以通过继承与覆盖（C++ 中的虚函数）实现“多态”。

这一讲中，我们定义 creation 模块时用到的 fillContainerBuffer、makeContainerShape 和 ContainerValueTypeHelper 这些约束表达式，就利用了 concept 的“原子约束”特性选择不同的模版版本，实现了泛型编程中的“多态”特性。



到这里，我们应该就可以理解，为什么说 Concepts 与约束是 C++20 以及后续演进标准之后，实现泛型编程的复用和“多态”特性的重要基石了。

此外，Concepts 还**给模板元编程带来了巨大提升**。

模板元编程已经成为现代 C++ 不可或缺的一部分，因此学习和掌握模板元编程的基本概念变得越来越重要。模板元编程的本质就是**以 C++ 模板为语言，以编译期常量表达式为计算定义，以编译期常量（包括普通常量与类型的编译期元数据）为数据，最终实现在编译时完成所有的运算（包括类型运算与数值运算）**。

> 模板元编程属于 C++ 中“戏弄”编译器而产生的副产品，因此在 C++11 之前并无法得到最高优先级的支持，使得我们不得不定义大量的类型，通过类型中的嵌套类型与枚举类型完成编译期计算。所以说，在 C++11 之前虽然能实现这些功能，却鲜有人将模板元编程应用到实际工程中。
>
> C++11 首次强调了模板元编程在 C++ 中的重要地位。这个版本开始，引入了 constexpr（允许我们定义编译器可计算的表达式），还提供了模板元编程的一些配套工具库（比如 type_traits），这意味着编译时计算在现代 C++ 中是核心议题。

虽然 C++11 提供了很多模板元编程的基础设施，但缺乏一种标准的抽象手段来描述模板参数的约束，这也使得模板元编程中，各个模板之间缺乏描述调用关系的简单手段。尤其是递归计

算的定义令人更加头痛。

对于代码中的 ContainerValueTypeHelper 的实现来说，在使用 concept 后，代码更加简洁易懂，这就是 concept 为模板元编程带来的重要提升。

我们知道，SFINAE 是自模板技术诞生以来就存在的一个规则。该规则让开发者可以通过一些方式，让编译器根据模板参数类型，选择合适的模板函数与模板类。但是，在 C++11 标准中加入了 type_traits 后，我们就可以在模板中通过标准库获取静态元数据，并决定模板类与函数的匹配与调用路径。

不过，这种在模板参数或函数参数列表中填充 type_traits 的方式，会让开发者的代码变得难以维护，而且用户更是难以阅读调用时的错误消息，这让 type_traitis 冲突时的偏序规则难以捉摸。而 Concepts 与约束的提出，正好完美地解决了这些问题。由此，C++20 就成了继 C++11 后让模板元编程脱胎换骨的一个标准。

总结

在 C++20 及其后续演进标准中，提供了使用编译期常量表达式编写模板参数约束的能力，并通过 Concepts 提供了为约束表达式起名的能力。

设计 Concepts 是一件非常重要的事情。我们通过实战案例展示了如何利用 concept 这一核心语言特性变更，实现了编译时模板匹配和版本选择时的 SFINAE 原则，并通过“原子约束”的特性实现了根据不同的约束选择不同的模板版本。

通过这三讲的内容相信你也感受到了，我们在现代 C++ 时代绕不开泛型编程。掌握 C++ 模板元编程的基础知识，并将这些新特性应用到编写的代码中来改善编程体验和编译性能，对一名 C++ 开发者来说至关重要。

课后思考

从我们已经讲解的现代 C++ 特性中可以了解到，所有特性都是为 C++ 编译时计算提供服务的，这也再次印证了 C++ 设计哲学——“不为任何抽象付出不可接受的多余运行时性能损耗”。事实上，编译时计算变得越来越重要了。那么，根据经验来说，你觉得哪些代码或功能可以开始向编译时计算开始迁移？

不妨在这里分享你的见解，我们一同交流。下一讲见！

上一篇 05丨Concepts：解决模板接口的类型与约束定义难题

下一篇 07丨Coroutines背景：异步I/O的复杂性

精选留言 (3)

写留言

黄骏
2023-01-29 来自湖北

内容太干了。哈哈。
nits：concepts.cpp的line 38应该是IteratorMemberFunction(&T::end)吧？

> 作者回复: 是的，应该把begin换成end。
> 已进行修正。

1

Geek_7c0961
2023-01-31 来自美国

concepts 这个feature太强了, 让我想起了rust中的traits 和 traits objects.

> 作者回复: Concepts的确是非常重要的feature，让C++编译时静态化编码整体提升了一个档次。现代C++演进有一种“Rust化”的趋势，当然了，C++还是有着独特的气质，而且对向前兼容性方面几乎是无敌的。

Geek_e04349
2023-01-27 来自云南

感觉 SliceItem 的 getValidValue 使用 optional 返回会更符合其语义

作者回复: 这里的确是可以使用std::optional。