05 | Concepts：解决模板接口的类型与约束定义难题

2023-01-25 卢誉声 来自北京

《现代C++20实战高手课》

讲述：卢誉声

时长 21:10　大小 19.34M

你好，我是卢誉声。

在上一讲中，我们了解到 C++ 模板不仅具备强大的泛化能力，自身也是一种“图灵完备”的语言，掀起了 C++ 之父 Bjarne Stroustrup 自己都没料到的“模板元编程”这一子领域。

但是，使用模板做泛型编程，最大的问题就是缺少良好的接口，一旦使用过程中出现偏差，报错信息我们难以理解，甚至无从下手。更糟的是，使用模板的代码几乎无法做到程序 ABI 层面兼容。这些问题的根本原因是 C++ 语言本身缺乏模板参数约束能力，因此，既能拥有良好接口、高性能表达泛化，又能融入语言本身是非常困难的。

好在 C++20 标准及其后续演进中，为我们带来了 Concepts 核心语言特性变更来解决这一难题。那么它能为我们的编程体验带来多大的革新？能解决多少模板元编程的历史遗留问题？今天我们一起探究 Concepts。

课程配套代码，点击这里即可获取。

定义 Concepts

首先我们看看 Concepts 是什么，它可不是横空出世的，C++20 为模板参数列表添加了一个特性——约束，采用约束表达式对模板参数进行限制。约束表达式可以使用简单的编译期常量表达式，也可以使用 C++20 引入的 requires 表达式，并且支持约束的逻辑组合，这是对 C++20 之前 enable_if 和 type_traits 的进一步抽象。

在约束的基础上，C++20 正式提出了 Concepts，也就是由一组由约束组成的具名集合。

我们可以将一组通用的约束定义为一个 concept，并且，在定义模板函数与模板类中，直接使用这些 concept 替换通用的 typename 和 class，所以 concept 的定义必定是约束的表达式，定义方式就像这样。

复制代码

```
template <参数模板>
concept 名称 = 约束表达式;
```

看一个最简单的例子如何定义一个 concept，使用 type_traits 的简单版本。

复制代码

```
template<class T, class U>
concept Derived = std::is_base_of<U, T>::value;
```

这里定义了一个名为 Derived 的 concept，有两个类型参数 T 和 U，其中的约束定义为 `std::is_base_of<U, T>::value`，也就是判定 U 是否为 T 的基类。相比于传统基于 SFINAE 和 enable_if 的方式，这种约束定义明显更加清晰。

我们再来看一个更加具体的 concept。

复制代码

```
class BaseClass {
public:
    int32_t getValue() const {
        return 1;
```

```
    }
};

template<class T>
concept DerivedOfBaseClass = std::is_base_of_v<BaseClass, T>;
```

在这段代码中，我先定义了一个基类 BaseClass，该类定义了一个成员函数 getValue。我又定义了名为 DerivedOfBaseClass 的 concept。需要注意的是，我在这里使用了一个 C++17 标准之后引入的工具变量模板 is_base_of_v，相当于 `is_base_of<BaseClass, T>::value`。

简单来说，这里定义的 concept，可以判定模板参数 T 是否为 BaseClass 的派生类，通过一些现代 C++ 语法变换，我们定义的 concept 更易读和使用。

有了定义好的 concept，如何使用呢，我写了一个例子。

复制代码

```
template <DerivedOfBaseClass T>
void doGetValue(const T& a) {
    std::cout << "Get value:" << a.getValue() << std::endl;
}

class DerivedClass: public BaseClass {
public:
    int32_t getValue() const {
        return 2;
    }
};

int32_t c2() {
    DerivedClass d;
    doGetValue(d);

    BaseClass b;
    doGetValue(b);

    return 0;
}
```

我们先从代码的第 6 行开始，定义一个名为 DerivedClass 的类，它继承 BaseClass，并重新定义了函数 getValue 的具体实现。接着，在函数 c2 中，我们定义了两个类型分别为

DerivedClass 和 BaseClass 的对象，并调用函数 doGetValue，在编译时进一步验证我们编写的基于 concept 的代码。

doGetValue 是代码开头定义的一个模板函数，它的模板参数很特别，采用 DerivedOfBaseClass 定义了 T，而非 typename/class，这个意思是实例化时传入的模板参数 T，必须符合 DerivedOfBaseClass 这个 concept 的要求。

现在，我们编译运行这段代码，可以看到输出是后面这样。

```
Get value:2
Get value:1
```

那么，如果参数不是 BaseClass 的派生类，会发生什么呢？我们来看看后面这段代码。

复制代码

```
class NonDerivedClass {
public:
    int32_t getValue() const {
        return 3;
    }
};

int32_t c2() {
    NonDerivedClass n;
    doGetValue(n);

    return 0;
}
```

这段代码在编译时会报错，报错信息是这样。

```
error C2672: “doGetValue”: 未找到匹配的重载函数
message : 可能是“void doGetValue(const T &)”
message : 未满足关联约束
: message : 计算结果为 false 的概念“DerivedOfBaseClass<NonDerivedClass>”
: message : 未满足约束
```

这段报错信息中，我们很容易知道，由于 NonDerivedClass 并非 BaseClass 的派生类，编译时发生了错误。

从这个案例看，如果 C++ 模板通过 concept 进行了“约束”，调用者再也不需要从难以理解的模板编译错误中寻找问题根源了。基于 concept 的模板编译错误信息，极具指导性，**足够简单、易于理解和纠错。**

不过讲到这里，关于这段代码，模板参数 T 的概念是 DerivedOfBaseClass，函数 doGetValue 的参数必须符合 DerivedOfBaseClass 这个概念。

```
1 template<class T>
2 concept DerivedOfBaseClass = std::is_base_of_v<BaseClass, T>;
```

你可能会有一个疑问：**为何不使用虚函数来解决这个问题呢？**如果将 getValue 定义成虚函数，并将 doGetValue 的参数类型设定成 const BaseClass&，不也可以实现一样的效果吗？

事实上，虚函数是基于虚函数表等特性来实现的，会对调用性能产生一定的损耗，也可能因为不同编译器内存模型，产生 ABI 兼容性问题。但是，如果用模板，编译器就可以通过编译期判定，直接消除虚函数造成的性能副作用，同时，编译器也可以充分利用各种跨函数调用的优化方式，生成性能更好的代码，有效提升生成代码的质量。因此，很多工程场景下，这种方法比基于虚函数实现的多态更加合理。

所以，我们可以看到，**通过约束与 concept 这两个 C++ 核心语言特性变更（高级抽象），实现了对模板参数列表与参数的约束的逻辑分离**。这不仅能提升模板函数或类接口的质量，还可以彻底提升代码的可读性。

如果从语言设计的角度进一步探讨，Concepts，本质就是让开发者能够定义在模板参数列表中直接使用的“类型”，与我们在函数的参数列表上使用的由 class 定义的类型，理论上讲，是一样的。

所以，在面向对象的编程思想中，我们思考的如何设计清晰且可复用的 class，那从此以后，在泛型编程中我们就需要转变一下，思考如何设计清晰且可复用的 concept。可以说，**从**

C++20 标准及其演进标准之后，concept 之于 C++ 泛型编程，正如 class 之于 C++ 面向对象。

了解了使用 Concepts 的优点，接下来，我们看看它的高级用法。我们会从 requires 关键字定义的约束表达式开始，掌握逻辑操作符的组合用法，之后会了解一下 requires 子句的概念、约束顺序规则，涵盖 Concepts 的各个重要方面。

约束表达式

定义 Concepts 时我们提到，一个 concept 被定义为约束表达式（constraint expression）。那什么是约束表达式呢？

从定义上来说，约束表达式是“用于描述模板参数要求的操作符与操作数的序列”，你也可以简单理解为布尔常量表达式。**约束表达式，本身是通过逻辑操作符的方式进行组合的，用于定义更复杂的 concept。**

约束的逻辑操作符一共有三种。

- 合取式（conjunctions）
- 析取式（disjunctions）
- 原子约束（atomic constraints）

编译器实例化一个模板函数或者模板类时，会按照一定顺序，逐一检查模板参数是否符合所有的约束要求（检查顺序具体可参考课后小知识）。我们来深入理解一下这几种逻辑操作符。

合取式

合取（conjunctions）通俗易懂的说法就是“逻辑与”，AND。在约束表达式中，合取式就是通过 && 操作符，把两个约束表达式连接到一起的。

我们来看三个例子。

复制代码

```
template<class T>
concept Integral = std::is_integral_v<T>;

```

```
template<class T>
concept SignedIntegral = Integral<T> && std::is_signed_v<T>;

template<class T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```


首先，定义了一个名为 Integral 的 concept，表示参数模板类型需要为整型。

接着，定义了名为 SignedIntegral 的 concept，表示参数模板类型需要为有符号整型。定义体就是一个合取式，`&&` 表示这个 concept 必须同时满足 `Integral<T>` 这一 concept 和 `std::is_signed_v<T>` 这一编译时常量表达式。

最后，我还定义了名为 UnsignedIntegral 的 concept，表示参数模板类型需要为无符号整型。其定义体表示必须满足 `Integral<T>` 和 `!SignedIntegral<T>` 这两个 concept。

编译器在处理合取式的时候，要求左右两侧约束都必须满足。检测过程遵循逻辑与表达式自左向右的短路运算原则，也就是说，如果左侧表达式不满足要求，右侧表达式也不会执行。因此，即使右侧表达式执行存在问题，也不会被执行，引发检测失败。你可以结合后面的代码加深理解。

复制代码

```
template<typename T>
constexpr bool get_value() { return T::value; }

template<typename T>
    requires (sizeof(T) > 1 && get_value<T>())
void f(T) {
    std::cout << "template version" << std::endl;
}

void f(int32_t) {
    std::cout << "int version" << std::endl;
}

void c15() {
    f('A');
}
```

我们在调用函数 f 的时候，由于 `'A'` 的类型为 char，其 sizeof 为 1，因此 sizeof(T) > 1 为 false。所以 `get_value<T>()` 是不会执行的，这里并不会引发编译错误（char 类型没

有::value）。虽然有些反直觉，但这就是合取式的短路运算原则。

析取式

析取（disjunctions）就是“逻辑或”，OR。在约束表达式中，析取式就是通过 || 操作符将两个约束表达式连接到一起。具体我们来看代码。

```
复制代码
1 template <class T>
2 concept Integral = std::is_integral_v<T>;
3
4 template <class T>
5 concept FloatingPoint = std::is_floating_point_v<T>;
6
7 template <class T>
8 concept Number = Integral<T> || FloatingPoint<T>;
```

这里定义了三个 concept：Integral、FloatingPoint 和 Number，其中 Number 这个 concept 通过 || 将 Integral 和 FloatingPoint 这两个 concept 连接在一起，表达只要为整型或者浮点型即可。

编译器在处理析取式的时候，**要求左右两侧约束满足其一即可，检测过程遵循逻辑或表达式自左向右的短路运算原则**，也就是只要左侧表达式满足要求，右侧表达式就不会执行。因此，即使这时右侧表达式执行存在问题，也不会被执行引发检测失败。

原子约束

原子约束（atomic constraints）是最后一种约束表达式，本身是一个很简单的概念，但是对编译器解析约束表达式非常重要，我们单独讲一下。

原子约束，由表达式 E 与 E 的参数映射组成。参数映射指的是 E 中受约束实体的模板参数（template parameter）与实例化时使用的模板实参（template argument）之间的映射关系。原子约束是在约束规范化过程中形成的，一个原子约束不能包含逻辑与 / 或表达式。

编译器在实例化过程中，会检查参数是否满足原子约束。编译器会根据参数映射关系，将模板实参替换成表达式 E 中的形参。

- 如果替换后的表达式是一个非法类型或者非法表达式，说明当前实例化参数不满足约束。
- 否则，编译器会对表达式的值进行左右值转换，只有得到的右值类型是 bool 类型，并且值为 true 时，编译器才认定为满足约束，否则就是不满足约束。

值得一提的是，E 的值必须是 bool 类型，不允许通过任何隐式转换变为 bool 型（这个和 C++ 中的 if 不一样）。听起来有些复杂，我们看一段代码，很好理解。

复制代码

```
template<typename T>
struct S {
    constexpr operator bool() const { return true; }
};

template<typename T>
    requires (S<T>{})
void f1(T) {
    std::cout << "Template" << std::endl;
}

template<typename T>
    requires (1)
void f2(T) {
    std::cout << "Template" << std::endl;
}

template<typename T>
    requires (static_cast<bool>(S<T>{}))
void f3(T) {
    std::cout << "Template" << std::endl;
}
```

在这段代码中，函数 f1 和 f2 的约束都会编译失败，只有 f3 才是正确的原子约束表达式。

另外，如果在源代码中，两个原子约束表达式相同，参数映射也相等，那么这两个原子约束就是完全相同的。我们看下面这段代码。

复制代码

```
template <class T>
concept Floating = std::is_floating_point_v<T>;

template <class T>
concept BadNumber = std::is_floating_point_v<T> && std::is_floating_point_v<T>;

```

```cpp
template <class T>
concept Number = Floating<T> && std::is_floating_point_v<T>;

template <Floating T> // #1
void func(T) {}

template <BadNumber T> // #2
void func(T) {}

template <Number T> // #3
void func(T) {}
```


在这段代码中，如果调用 func 同时匹配 #1 和 #2，会相互冲突导致编译失败，而同时匹配 #1 和 #3 不会。这是因为，BadNumber 中第一个 `is_floating_v` 和 Floating 中的 `is_floating_v`，并不会识别为相同的原子约束，导致编译器认为匹配了两个版本，不知道选择哪个版本而失败。

Number 中第一个原子约束直接使用了 Floating，所以 Number 属于 Floating 的派生约束。虽然两个约束都能匹配，但 Number 是比 Floating 更精准的匹配，所以编译器最后会选择 #3 版本，并不会发生编译错误。

作为补充，编译器为了后续进行统一的语法和语义分析，会在约束表达式的解析过程中对约束表达式进行规范化。

学习了三种约束表达式，我们讨论几个重要细节，包括 requires 表达式、requires 子句以及约束顺序等高级话题。

requires 关键字

我们在前面已经看到了由 type_traits 和 requires 构成的 concept，针对 requires 表达式和 requires 子句这两个概念，我们简单说明一下。

requires 表达式

跟 type_traits 类似，requires 表达式本身就是一个谓词。

requires 与普通约束表达式不同，如果其定义体在完成参数替换后，存在非法的类型或表达式，或者 requires 定义中的约束存在冲突时，会返回 false。反之，如果完成参数替换后语法检查以及约束检查全部成功之后，才会返回 true。

定义方式是这样。


```
requires (可选参数) { // 表达式结果必须为 bool 类型
    表达式_1
    表达式_2
    ...
}
```

“可选参数”声明了一系列局部变量（不支持提供默认参数），大括号中的所有表达式都可以访问这些变量，如果表达式使用了未声明变量，编译时会报错。

requires 大括号内，可以定义几种不同的表达式，分别用于约束接口或函数行为、变量、类型，同时还可以对约束进行组合和嵌套。在这里，我用一个例子来说明这几种不同表达式的用法。

复制代码

```
template<typename T>
concept Histogram = requires(Histo h1, Histo h2) {
    h1.getMoments();          // 要求有getMoments接口
    T::count;                 // 要求有静态变量count
    h1.moments;               // 要求有成员变量moments
    h1 + h2;                  // 要求对象能够进行+操作

    typename T::type;         // 要求存在类型成员type
    typename std::vector<T>; // 要求能够模板实例化并与std::vector组合使用

    { h1.getSubHistogram() } -> same_as<T>;     // 要求接口返回类型与T一致
    { h1.getUnit() } -> convertible_to<float>; // 要求接口返回类型能转换成float，本质
    { h1 = std::move(h2); } noexcept;           // 要求表达式不能抛出异常

    requires sizeof(T) > 4;
}
```

requires 表达式定义中的约束分为四种类型，分别是：

- 基本约束：第 3 到 6 行，这种独立的、不以关键词开头的表达式语句，都是基本约束，只会进行词法、语法和语义的正确性检查，并不会真实执行。编译器检查通过，则约束检查通过，否则检查失败。

- 类型约束：第 8 到 9 行，这种使用 typename 开头的表达式语句是类型约束，表达式用于描述一个类型，如果类型存在，则约束检查通过，否则检查失败。
- 组合约束：第 11 到 13 行，这种类似“**{} [noexcept] -> 约束**”形式的都是组合约束，编译器会执行{}中的语句，并检查其结果类型是否符合后续约束，如果符合约束则检查通过，否则检查失败。此外，还可以通过可选的 noexcept 检查表达式是否会抛出异常。
- 嵌套约束：第 15 行，requires 开头的就是嵌套约束，用于嵌套新的 requires 表达式，如果 requires 表达式结果为 true 则检查通过，否则检查失败。

requires 子句

下面我们看一看，相较于 requires 表达式这一谓词，requires 中出现的另一个关键字——requires 子句，又是怎么回事？

由于 requires 子句和 requires 表达式并不是相同的概念，所以我们可能会看到这种代码：


```
1 export template <typename T1, typename  T2>
2     requires requires (T1 x, T2 y) { x + y; }
3 std::common_type<T1, T2> func(
4     T1 arg1, T2 arg2
5 ) {
6     return arg1 + arg2;
7 }
```

我们在模板头上定义了一条 requires 子句，它表达了模板参数应该在什么条件下工作，在这里我们还可以定义更复杂或具体的约束表达式。

这里有两个 requires，但是含义完全不同，`requires (T1 x,T2 y) { x + y; }` 就是 requires 子句，而前面的 requires 就是 requires 子句的开头，后面所需的是一个约束表达式，只不过 requires 表达式是约束表达式的一种，所以这是合法的代码。

requires 子句存在的意义是判断它所约束的声明在“上下文”中是否可行。所谓上下文，分为三种。

1. 函数模板：是在执行重载决议中进行的。

2. 模板类：在决策适合的特化版本当中。

3. 模板类中的成员函数：决策当显式实例化时是否生成该函数。

约束顺序

在前面，我们看到了给模板施加约束后，受约束的版本比未受约束的版本更优。但是，如果两个版本同样含有约束且都满足，哪个最优呢？

编译器在后续分析前，会将模板中所有的具名 concept 和 requires 表达式都替换成其定义，接着进行正规化，直到所有的约束变成原子约束及其合取式或析取式为止。然后分析约束之间的蕴含关系，并根据约束偏序选择最优的版本。

这里解释一下蕴含关系。针对约束 P 和约束 Q，只有通过 P 和 Q 中的原子约束证明 P 蕴含 Q，才认定约束 P 蕴含约束 Q（编译器并不会分析表达式和类型来判定蕴含关系，比如 N>0 并不蕴含 N>=0）。

蕴含关系非常重要，决定了约束的偏序。如果声明 D1 所受约束蕴含 D2 所受约束（或者 D2 不受约束），并且声明 D2 所受约束并不蕴含声明 D1 所受约束，我们就可以认为，声明 D1 的约束比声明 D2 的约束更加精准。这说明，当编译器选择声明版本时，如果参数同时符合 D1 和 D2 所受约束，编译器会选择 D1，也就不会引起编译错误。

只有了解并利用约束的偏序规则，我们才能更好地组织代码。

总结

今天我们了解了什么是 Concepts，它是由一组由约束组成的具名集合，约束支持普通编译期常量表达式，同时支持采用 requires 表达式，对模板参数进行更复杂的约束检查，并且支持约束的逻辑组合，这是对 C++20 之前 enable_if 和 type_traits 的进一步抽象。

在现代 C++20 标准及其后续演进中，约束的顺序通过概念约束进行决策，而约束的合取式、析取式以及 Concepts，在模板函数重载决议与类模板特化决策过程中，扮演了核心角色。编译器通过约束的偏序规则决策出最优解。

Concepts 这种高级抽象，妥善解决了模板接口的类型与约束定义难题，同时也改进了约束顺序决策。

结合 C++ 泛型编程，约束表达是一个较为复杂的议题，如何正确且有效地利用这一全新特性呢？下一讲，我们将通过实战案例来学习。

课后思考

我们不止一次提到编译时谓词，你如何理解“编译时计算”和“谓词”？

不妨在这里分享你的见解，与大家一起分享。我们一同交流。下一讲见！

课后小知识

- **编译器实例化检查顺序**

编译器实例化一个模板函数或者模板类时，会按照一定顺序，逐一检查模板参数是否符合所有的约束要求。看这段代码，我们说明一下约束的检查顺序。

```
template <std::integral T1, std::integral T2, int32_t V> // #1
    requires (std::is_convertible_v<T1, T2> && V > 0) // #2
std::common_type_t<T1, T2> add(T1 a, T2 b, std::integral auto c) // #3
    requires requires (T1 x, T2 y) { x + y + V; } // #4
{
    return a + b + c + V;
}
```

- 依次检查模板参数列表中引入的类型模板参数和非类型模板参数约束表达式，如示例代码中#1部分就是类型模板与非类型模板参数约束；
- 检查在模板参数列表后通过 requires 子句引入的约束表达式，如示例代码中 #2 部分；
- 检查**缩略函数模板**中引入的约束表达式，如示例代码 #3 部分 std::integral auto c 中 std::integral 就是对缩略函数参数 c 添加的约束；
- 检查在函数参数列表后通过 requires 子句引入的约束表达式，如示例代码中 #4 部分。

这里解释一下“**缩略函数模板（Abbreviated function template）**”。该特性在 C++20 中引入，支持通过 auto 省略传统的模板参数列表定义，如下代码中两者定义是等价的：

```
template <typename T1, typename T2>
std::common_type_t<T1, T2> sub(T1 a, T2 b) {
    return a - b;
}

auto sub(auto a, auto b) {
    return a - b;
}
```

第二个 sub 定义中虽然没有通过 template 定义模板，但由于函数参数列表中使用了 auto，因此，与第一种采用传统 template 模板参数列表定义的形式是等价的。这个特性让很多简单模板函数的定义变得非常简洁。

上一篇　04丨Concepts背景：C++泛型编程之殇

下一篇　06丨Concepts实战：写个向量计算模板库

精选留言 (1)

写留言

peter

2023-01-25 来自北京

请教老师两个问题：

Q1：b不是派生类，为什么不报错？

BaseClass b;　doGetValue(b); b是BaseClass，不是BaseClass的派生类，为什么不报错？

Q2：原子约束的f1和f2为什么失败？

f1前面加的 requires (S<T>{})表示什么意思？为什么失败？

原子约束有关键字吗？或者说，怎么看出来一个约束是原子约束？

另外，struct S中的constexpr operator什么意思？

作者回复: Q1：std::is_base_of<T1,T2>严格来说是判断T2是否是T1的派生类或相同类型，而不只是派生类。

Q2：原子约束f1和f2的失败原因是这两个都需要隐式转换成bool，所以原文中说“E的值必须是bool类型，不允许通过任何隐式转换变为bool型”。

F1的requires中的S<T>{}是构造了一个S<T>类型的对象，这个构造对象的语法就是C++11里的统一初始化表达式，相当于S<T>()。然后S这个类有一个operator bool()的成员函数，可以将一个S类型的对象隐式转换成bool类型，这种隐式转换在原子约束中是不允许的。

原子约束是一个概念，没有关键字。如果想要容易理解一点，非严格来说，基本上不是合取或者析取的合法约束一般就是原子约束了，也就是约束的基本组成部分。

1