第41章 概念和约束

《现代C++语言核心特性解析》 谢丙堃

使用std::enable_if约束模板

- 例子

```
template <class T, class U =
std::enable_if_t<std::is_integral_v<T>>>
struct X {};
X<int> x1; // 编译成功
X<std::string> x2; // 编译失败
```

概念的背景介绍

- 概念是对C++核心语言特性中模板功能的扩展。它是编译时进行评估，对类模板、函数模板以及类模板的成员函数进行约束：它限制了能被接受为模板形参的实参集。

将std::enable_if改写为concept

- 使用concept关键字或者直接使用requires

```
template <class C>
concept IntegerType = std::is_integral_v<C>;

template <IntegerType T>
struct X {};

template <class T>
requires std::is_integral_v<T>
struct X {};
```

使用concept和约束表达式定义概念

- IntegerType是概念名，这里的std::is_integral_v<C>我们称之为约束表达式。

```
template <class C>
concept IntegerType = std::is_integral_v<C>;

template <IntegerType T>
struct X {};
```

使用concept和约束表达式定义概念

- 约束表达式还支持一般的逻辑操作，包括合取和析取：

```
// 合取
template <class C>
concept SignedIntegerType = std::is_integral_v<C> && std::is_signed_v<C>;

// 析取
template <class C>
concept IntegerFloatingType = std::is_integral_v<C> || std::is_floating_point_v<C>;
```

使用concept和约束表达式定义概念

- requires子句必须是一个类型为bool的常量表达式，除此之外还有一些额外要求：
  - 是一个初等表达式，或者带括号的任意表达式；
  - 使用&&或者||运算符连接的上述表达式

使用concept和约束表达式定义概念

- 约束模板实参的方法很多，当一个模板同时具备多种约束时，如何确定优先级?
  - 模板形参列表中的形参的约束表达式，其中检查顺序就是按照形参出现顺序。
  - 模板形参列表之后的 requires子句中的约束表达式。
  - 简写函数模板声明中每个拥有受约束auto占位符类型的形参所引入的约束表达式。
  - 函数模板声明尾部 requires子句中的约束表达式。

原子约束

- 原子约束是表达式和表达式中模板形参到模板实参映射的组合（简称为形参映射）。

```
template <int N> constexpr bool Atomic = true;
template <int N> concept C = Atomic<N>;
template <int N> concept Add1 = C<N + 1>;
template <int N> concept AddOne = C<N + 1>;
template <int M> void f()
requires Add1<2 * M> {};
template <int M> void f()
requires AddOne<2 * M> && true {};

f<0>(); // 编译成功
```

原子约束

- 形参映射不同的情况。

```
template <int N> void f2()
requires Add1<2 * N> {};
template <int N> void f2()
requires Add1<N * 2> && true {};

f2<0>(); // 编译失败
```

requires表达式

- 该表达式同样是一个纯右值表达式，表达式为true时表示满足约束条件，反之false表示不满足约束条件。

```
template <class T>
concept Check = requires {
    T().clear();
};

template <Check T>
struct G {};

G<std::vector<char>> x; // 编译成功
G<std::string> y; // 编译成功
G<std::array<char, 10>> z; // 编译失败
```

requires表达式

- 支持形参列表。
  - a.clear()和a+b是要求序列

```
template <class T>
concept Check = requires(T a, T b) {
    a.clear();
    a + b;
};
```

requires表达式

- 4种要求序列:
  - 简单要求
  - 类型要求
  - 复合要求
  - 嵌套要求

requires表达式

- 简单要求
  - 不以requires关键字开始的要求，它只断言表达式的有效性，并不做表达式的求值操作。

```
template<typename T> concept C =
requires (T a, T b) {
    a + b;
};
```

requires表达式

- 类型要求
  - 以typename关键字开始的要求，紧跟typename的是一个类型名，通常可以用来检查嵌套类型、类模板以及别名模板特化的有效性。

```
template<typename T, typename T::type = 0> struct S;
template<typename T> using Ref = T&;
template<typename T> concept C = requires {
    typename T::inner; // 要求嵌套类型
    typename S<T>; // 要求类模板特化
    typename Ref<T>; // 要求别名模板特化
};
```

requires表达式

- 复合要求
  - 断言一个复合要求需要按照以下顺序：
    1. 替换模板实参到{E}中的表达式E，检测表达式的有效性。
    2. 如果使用了noexcept，则需要检查并确保{E}中的表达式E不会有抛出异常的可能。
    3. 如果使用了->后的返回类型约束，则需要将模板实参替换到返回类型约束中，并且确保表达式E的结果类型，即decltype((E))，满足返回类型约束。

requires表达式

- 复合要求例子:

```
template <class T>
concept Check = requires(T a, T b) {
    {a.clear()} noexcept;
    {a + b} noexcept -> std::same_as<int>;
};
```

requires表达式

- 嵌套要求
  - 以requires开始的要求，它通常是根据局部形参来指定其他额外的要求。

```
template <class T>
concept Check = requires(T a, T b) {
    requires std::same_as<decltype((a + b)), int>;
};
```

约束可变参数模版

- 是将各个实参替换到概念的约束表达式后合取各个结果。

```
template<class T> concept C1 = true;
template<C1... T> struct s1 {};

// 相当于(C1<T> && ...)，也就是约束每一个参数
```

约束类模板特化

- 约束可以影响类模板特化的结果，在模板实例化的时候编译器会自动选择更满足约束条件的特化版本进行实例化。

```
template<typename T> concept C = true;
template<typename T> struct X {
    X() { std::cout << "1.template<typename T> struct X" << std::endl; }
};
template<typename T> struct X<T*> {
    X() { std::cout << "2.template<typename T> struct X<T*>" << std::endl; }
};
template<C T> struct X<T> {
    X() { std::cout << "3.template<C T> struct X<T>" << std::endl; }
};

X<int*> s1;
X<int> s2;
```

约束auto

- 对auto和decltype(auto)的约束可以扩展到普遍情况，例如：

```
template <class C>
concept IntegerType = std::is_integral_v<C>;

IntegerType auto i1 = 5.2; // 编译失败
IntegerType auto i2 = 11; // 编译成功

IntegerType decltype(auto) i3 = 4.8; // 编译失败
IntegerType decltype(auto) i4 = 7; // 编译成功
```

感谢您的观看

欢迎关注