雾里看花：真正意义上的理解 C++ 模板(Template)

在 C++ 中，模板（Template）这个概念已经存在二十多年了。作为 C++ 最重要的一个语言构成之一，相关的讨论数不胜数。很可惜的是，相关深入的有价值的讨论很少，尤其是提供多个视角看待这一技术。很多文章在谈论模板的时候往往会把它和各种语法细节缠绕在一起，容易给人一种云里雾里的感觉。类似的例子还发生在其它话题上面，比如介绍协程就和各种 IO 混在一起谈，谈到反射似乎就限定了 Java，C# 中的反射。这样做并不无道理，但是往往让人感觉抓不到本质。看了一大堆，但却不得其要领，反倒容易把不同的概念混淆在一起。

就我个人而言，讨论一个问题喜欢多层次，多角度的去讨论，而不仅限某一特定的方面。这样一来，既能更好的理解问题本身，也不至于让自己的视野太狭隘。故本文将尝试从模板诞生之初开始讲起，以四个角度来观察，理清模板这一技术在 C++ 中的发展脉络。注意，本文并不是教学文章，不会深入语法细节。更多的谈论设计哲学和 trade-off 。掌握一点点模板的基础知识就能看懂，请放心食用。当然这样可能严谨性有所缺失，如有错误欢迎评论区讨论。

我们主要讨论四个主题：

- 控制代码生成，实现泛型
- 做泛型约束
- 编译期计算
- 对类型做计算

其中第一个主题一般认为就是普通的 Template。而后三者一般被规划到「TMP」中去。TMP 即 Template meta programming 也就是模板元编程。因为模板设计之初的意图并不是实现后面这三个功能，但是能以比较抽搐的语法来实现这些功能，代码写起来也比较抽象难懂，所以一般叫做元编程。

代码生成，实现泛型

事实上，这一点正是模板被设计之初的用法，用于实现泛型。在加入模板之前，常常使用宏来实现泛型。考虑下面这个简单的示例：

```
#define add(T) _ADD_IMPL_##T

#define ADD_IMPL(T)             \
T _ADD_IMPL_##T (T a, T b) {    \
    return a + b;               \
}

ADD_IMPL(int);
ADD_IMPL(float);

int main()
{
    add(int)(1, 2);
    add(float)(1.0f, 2.0f);
}
```

原理倒是很简单，其实就是把函数里面的类型替换成了宏参数。然后通过IMPL来「实例化」一个函数定义，最后直接使用就行了。但是上面的代码有很多缺点：

- 代码可读性差，宏的拼接和代码逻辑耦合
- 没法调试，宏只有展开后才能调试 ，报错信息不好阅读

- 在使用对应的函数之前，必须手动实例化，比如上面的`ADD_IMPL(int)`
- 需要显式写出对应的泛型类型，没法自动推导泛型类型

这些问题，在模板中都被解决了：

```
template<typename T>
T add(T a, T b)
{
    return a + b;
}

template int add<>(int, int); // 显式实例化

int main()
{
    add<int>(1, 2); // 显式指定模板参数 T
    add(1, 2); // 自动推导模板参数 T
    add(1.0f, 2.0f); // 自动推导并且隐式实例化
}
```

- 模板就是占位符，不需要拼接字符串，完全不影响代码的可读性
- 模板的报错信息相对友好，当类型不匹配时，会提示对应的类型
- 模板既可以隐式实例化也可以显式实例化
- 既可以自动推导模板参数，也可以显式指明模板参数

除此之外，模板还支持偏特化，特化，可变模板参数，类成员模板等等一系列特性，这些都是宏做不到的。而且通过模板这一特性的的确确实现了 STL 这样通用的标准库。经常听见很多人问为什么 C 语言没有别的语言那样的容器，算法标准库呢？一个很重的原因就是 C 语言抽象能力不够，没法实现一套这样通用的方案。而且结合后面聊的三个主题，我们可以通过模板实现一些更高级的代码生成。比如经常说的**编译期打表，打函数表**这种。

别的我都同意，但是你这上面的 **模板的报错信息相对友好，当类型不匹配时，会提示对应的类型。**难道这不是五十步笑百步吗？甚至有过之而无不及。轻松产生几百，几千行的代码报错，只有 C++ 的模板能做到吧。

哎，别急，这就是下面我要讲到的问题。

对类型做约束

首先我们要讨论的问题是，为什么 C++ 的编译错误信息这么长？而且有时候非常难读懂。

函数重载

```
struct A{};

int main()
{
    std::cout << A{} << std::endl;
}
```

在我的 GCC 编译器上，足足产生了 239 行报错信息。不过好消息是 GCC 把重点信息标记出来了，如下所示：

```
no match for 'operator<<' (operand types are 'std::ostream' {aka 'std::basic_ostream<char>'} and 'A')
    9 |     std::cout << A{} << std::endl;
      |     ~~~~~~~~~ ^~ ~~~
      |          |       |
      |          |       A
      |          std::ostream {aka std::basic_ostream<char>}
```

那大概还是能读懂的的，意思就是没有找到匹配的重载函数。也就是说我们需要为A重载operator<<。当然这个只是入门级别的，还是能轻松看懂的。但是我们好奇的是，剩下的两百行报错在干嘛呢？其实关键就在于重载函数和隐式类型转换。让我们来看其中一段信息。

```
note:   template argument deduction/substitution failed:
note:   cannot convert 'A()' (type 'A') to type 'const char*'
    9 |     std::cout << A{} << std::endl;
```

意思就是尝试用A类型匹配const char*这个重载（通过隐式类型转换），结果失败了。标准库类似这样的函数，都实现了很多的重载函数，比如这个 operator<<就重载了int, bool, long, double 等等，将近几十个函数。结果报错信息就是把所有重载函数尝试失败的原因都列出来，于是轻松就有几百行了，再加上标准库诡异的命名，看起来就像天书一样。

模板

函数重载是导致报错信息难以读懂的一部分原因，但不是全部。实际上如上面所示，仅仅是把所有可能性枚举出来，不过几百行报错。要知道我们还能产出上千行呢，量级上的差距可不是能用数量轻松堆叠出的。况且本小节要说的是约束，和报错有什么关系呢。 来看下面这个简单的例子：

```
struct A{};
struct B{};

template<typename T>
void test(T a, T b)
{
    std::cout << a << b << std::endl;
}

int main()
{
    test(A{}, B{}); // 短短几行报错
    test(A{}, A{}); // 几百行报错
}
```

究竟为什么会出现如此大的差距呢？还记得我们在第一部分里面说的模板相比于宏的两个优点吗？一个是自动类型推导，一个是隐式实例化。对于模板报错来说也基本是从这两个角度入手，test(A{}, B{})这里模板参数推导失败了。因为test函数隐含了一个重要的条件，那就是 a 和 b 的类型是一样的，于是实际上它报的错误是找不到匹配的函数，然后把推断失败的模板函数的原因列出来。而第二个函数test(A{}, A{})则是模板参数推导成功了，进入到实例化的阶段了，但是在实例化的阶段出错了。也就是说T已经被推断为A了，尝试把A代入函数体的时候，出错了。于是就只能把替换失败的原因列出来了。

那对类型做约束有什么用呢？看下面这个例子

```
struct A{};

template<typename T>
void print1(T x)
{
    std::cout << x << std::endl;
}

template<typename T>
// requires requires (T x) { std::cout << x; }
// C++20加入的requires语法，意思就是要求std::cout << x`是合法的。
void print2(T x)
{
    print1(x);
    std::cout << x << std::endl;
}


int main()
{
    print2(A{});
}
```

短短几行，在我的 GCC 上产生了 700 行的编译错误。稍微改动一下，把注释掉的那行代码加上。相比之下这种情况的代码报错只有短短几行：

```
In substitution of 'template<class T>  requires requires(T x) {std::cout << x;} void print2(T) [with T = A]':
required from here
required by the constraints of 'template<class T>  requires requires(T x) {std::cout << x;} void print2(T)'
in requirements with 'T x' [with T = A]
note: the required expression '(std::cout << x)' is invalid
   15 | requires requires (T x) { std::cout << x; }
```

意思就是`A`类型的实例`x`不满足`requires`语句`std::cout << x`。事实上通过这样的语法，我们就可以把错误限制在类型推断的阶段，而不去进行实例化。于是报错就友好一万倍了。也就是说通过`requires`我们能阻止编译错误的传播。 但是可惜的是，相关的约束语法是在 C++20 才加入的。那在这之前呢？

C++20之前

在 C++20 之前，我们并没有这么好用的方法。只能通过一种叫做 SFINAE的技术来实现类似的功能，对类型实现约束。比如上面那个功能，在 C++20 之前只能这么写：

```
template<typename T, typename = decltype(std::cout << std::declval<T>())>
void print2(T x)
{
    print1(x);
    std::cout << x << std::endl;
}
```

具体的规则在这里就不介绍了，感兴趣的可以去搜搜相关的文章看看。

结果就是`typename = decltype(std::cout << std::declval<T>())`这行代码，完全让人不知所云。只有深入了解相关的规则之后才能看懂这究竟是在干嘛。被口诛笔伐自然是少不了的，如此常用的功能直到 C++20 才加入，实在是很让人很流汗黄豆啊。不过据 C++ 之父本人 自述 ，其实他早就意识到这个问题了，需要对泛型添加某种约束，不过一直拖到 C++20 而已（笑）。别的支持泛型的语言，也基本都有类似的东西。例如 Rust 和 C# 都通过`where`来表达类似的约束。

编译期计算

意义

首先要肯定的一点是，编译期计算肯定是有用的。具体到特定场景，意义有多大，倒是没法判断。有很多人谈编译期计算色变，什么代码难懂，屠龙技，没有价值云云。这样的确很容易误导初学者。事实上相关的需求的确存在。如果编程语言没有这个功能，但是确有需求，程序员也会想方设法的通过其它的办法来实现。

我将举两个例子来说明：

- 首先是编译器对常量表达式的优化，相信这个大家都并不陌生。极其简单的情况，像`1+1+x`这样的表达式，必然是会被优化成`2+x`。事实上现代编译器对于类似的情况能做的优化非常多，比如这个 问题。提问者提问 C 语言的`strlen`函数在参数是常量字符串的时候，会不会把函数调用直接优化成一个常量。比如`strlen("hello")`会不会直接优化成`5`。从主流编译器的实验结果来看，答案是肯定的。类似的情况数不胜数，不知不觉中你就在使用编译期计算。只是它被归到编译器优化的一部分去了。而编译器的优化能力总归是有上限的，允许使用者自己定义这种优化规则，会更加灵活和自由。比如在 C++ 里面明确了`strlen`是`constexpr`的，这种优化必然会发生
- 其而是在程序语言发展早期，编译器优化能力还没那么强的时候。就已经开始广泛的使用外部脚本语言提前算好数据（甚至生成好代码）用来减少运行时开销了。典型的例子是算好三角函

数表这种常量表，然后运行期直接用就行了（比如在编译代码之前，运行一段python脚本用来生成一些需要的代码）

C++ 的编译期计算有明确的语义保证，并且内嵌于语言之中，能和其它部分良好的交互。从这个角度来说，很好的解决了上面两点问题。当然很多人对它的讨伐并不无道理，通过模板元编程进行的编译期计算。代码丑陋且晦涩难懂，牵扯到的语法细节多，并且大大拖慢编译时间，增加二进制文件大小。无可否认的是，这些问题的确存在。但是随着 C++ 版本的不断更新，编译期计算现在已经非常容易理解了，不再需要去写那些复杂的模板元代码，新手也能很快学会。因为和运行期代码几乎一样了。接下来伴随着它的发展史，我们将逐步阐明。

发展史

从历史上看，TMP 是一个偶然事件。在标准化 C++ 语言的过程中发现它的模板系统恰好是图灵完备的，即原则上能够计算任何可计算的东西。第一个具体演示是 Erwin Unruh 编写的一个程序，该程序计算素数，尽管它实际上并未完成编译：素数列表是编译器在尝试编译代码时生成的错误消息的一部分。具体的示例，请参考 这里。

作为入门级别的编程案例，可以展示一个编译期计算阶乘的方法：

```
template<int N>
struct Factorial
{
    enum { value = N * Factorial<N - 1>::value };
};

template<>
struct Factorial<0>
{
    enum { value = 1 };
};

std::cout << Factorial<5>::value << std::endl; // 输出 120
```

这段代码即使在 C++11 之前也能通过编译，在那之后 C++ 引入了很多新的东西用于简化编译期计算。最重要的就是constexpr关键字了。可以发现在 C++11 之前，我们并没有合适的办法表示编译期常量这一概念，只能借用enum来表达。而 C++11 之后，我们可以这么写：

```
template<int N>
struct Factorial
{
    constexpr static int value = N * Factorial<N - 1>::value;
};

template<>
struct Factorial<0>
{
    constexpr static int value = 1;
};
```

尽管进行了一些简化，但实际上我们仍然是借助模板来进行编译期计算。这样写出的代码是难以读懂的，主要原因有以下两点：

- 模板参数只能是编译期常量，并没有编译期变量的概念，无论是全局还是局部都没有
- 只能通过递归而不能通过循环来进行编程

想象一下，如果平常写代码，把变量和循环给你禁了，那写起来是有多难受啊。

那有没有满足上面两个特征的编程语言呢？其实满足上面两点的编程语言，一般称为 pure functional 也即纯函数式的编程语言。Haskell 就是一个典型的例子。 但是 Haskell 它有强大的模式匹配，在熟悉了 Haskell 的思维之后，也能写出短小优美的代码（而且 Haskell 本身也能用 do 语法模拟出局部变量，因为

使用局部变量，其实就相当于把它作为函数参数一级级传递下去）。而 C++ 这些都没有，属于是把别人缺点都继承来了，优点一个没有，自然是被声讨的对象。所幸的是，上面这些问题都在cosntexpr function中都被解决了。

```
constexpr std::size_t factorial(std::size_t N)
{
    std::size_t result = 1;
    for (std::size_t i = 1; i <= N; ++i)
    {
        result *= i;
    }
    return result;
}

int main()
{
    constexpr auto a = factorial(5); // 编译期计算
    std::size_t& n = *new std::size_t(6);
    auto b = factorial(n); // 运行期计算
    std::cout << a << std::endl;
    std::cout << b << std::endl;
}
```

事实上，C++ 允许在一个函数前面直接加上constexpr关键字修饰。表示这个函数既可以在运行期调用，也可以在编译期调用，而函数本身的内容几乎不需要任何改变。这样一来，我们可以直接把运行期的代码复用到编译期。也允许使用循环和局部变量进行编程，可以说和平常写的代码没有任何区别。很令人震惊对吧，所以编译期计算在 C++ 里面早已经是一件司空见惯的事情了，用户压根就不需要去写复杂的模板元。在 C++20 之后几乎所有的标准库函数也都是constexpr的了，我们可以轻松的调用它们，比如编译期排序。

```
constexpr auto sort(auto && range)
{
    std::sort(std::begin(range), std::end(range));
    return range;
}

int main()
{
    constexpr auto arr = sort(std::array{1,3,4,2,3});
    for(auto i : arr)
        std::cout << i;
}
```

真正意义上的代码复用！如果你想要这个函数只在编译期执行，你也可以用consteval标记它。同时，在 C++20 中还允许了编译期动态内存分配，可以在constexpr function中使用new来进行内存分配，但是编译期分配的内存必须要在编译期释放。你也可以直接在编译期使用vector和string这样的容器。而且请注意，相比于利用模板进行编译期计算，constexpr函数的编译速度会快很多 。如果你好奇编译期是如何实现这一强大的特性的，可以认为，C++编译器内部内嵌了一个小的解释器，这样遇到constexpr函数的时候用这个解释器解释一下，再把计算结果返回就行了。

相信你已经充分见识到 C++ 在编译期计算方面所做的努力，编译期计算早就和模板元脱离关系了，在 C++ 中已经成为一种非常自然的特性，不需要特殊的语法，却能发挥强大的威力。所以以后千万不要一谈到 C++ 的编译期计算就十分恐慌，以为是什么屠龙之技。现在它早已经变得十分温柔美丽。

尽管编译期计算已经脱离了模板元的魔爪，但是 C++ 并没有。还有两种情况，我们不得不编写蹩脚的模板元代码。

无法迈过的坎

对类型做计算

如何判断两个类型相等呢，或者说判断两个变量的类型相等。可能有人会想，这不是多此一举吗，变量的类型都是编译期已知的，还需要判断吗？其实这个问题可以说是伴随着泛型编程而出现的，考虑下面的示例：

```
template<typename T>
void test()
{
    if(T == int){...}
}
```

这样的代码是符合我们直觉的，可惜 C++ 并不允许你这么写。不过在 Python / Java 等语言中确实有这种写法，但是它们的判断大多都是在运行时的。C++ 的确允许我们在编译期对类型进行操作，但是可惜的是类型并不能作为一等公民，作为普通的值，只能作为模板参数。我们只能写出如下的代码：

```
template<typename T>
void test()
{
    if constexpr(std::is_same_v<T, int>){...}
}
```

类型只能存在于模板参数里面，这直接导致上一小节的`constexpr`编译计算提到的优势全都消失了。我们又回到了刀耕火种的时代，没有变量和循环。

下面是判断两个`type_list`满足不满足子序列关系的代码:

```
template<typename ...Ts>
struct type_list{};

template<typename SubFirst, typename ...SubRest, typename SuperFirst, typename ...SuperRest>
constexpr auto is_subsequence_of_impl(type_list<SubFirst, SubRest...>, type_list<SuperFirst, SuperRest...>)
{
    if constexpr (std::is_same_v<SubFirst, SuperFirst>)
        if constexpr (sizeof...(SubRest) == 0)
            return true;
        else
            return is_subsequence_of(type_list<SubRest...>{}, type_list<SuperRest...>{});
    else
        if constexpr (sizeof...(SuperRest) == 0)
            return false;
        else
            return is_subsequence_of(type_list<SubFirst, SubRest...>{}, type_list<SuperRest...>{});
}

template<typename ...Sub, typename ...Super>
constexpr auto is_subsequence_of(type_list<Sub...>, type_list<Super...>)
{
    if constexpr (sizeof...(Sub) == 0)
        return true;
    else if constexpr (sizeof...(Super) == 0)
        return false;
    else
        return is_subsequence_of_impl(type_list<Sub...>{}, type_list<Super...>{});
}

int main()
{
    static_assert(is_subsequence_of(type_list<int, double>{}, type_list<int, double, float>{}));
    static_assert(!is_subsequence_of(type_list<int, double>{}, type_list<double, long, char, double>{}));
    static_assert(is_subsequence_of(type_list<>{}, type_list<>{}));
}
```

写起来非常难受，我把相同的代码逻辑用`constexpr`函数写一遍，把类型参数换成`std::size_t`

```
constexpr bool is_subsequence_of(auto&& sub, auto&& super)
{
    std::size_t index = 0;
    for (std::size_t i = index; index < sub.size() && i < super.size() ; i++)
    {
        if(super[i] == sub[index])
        {
            index++;
```

```
        }
    }
    return index == sub.size();
}

static_assert(is_subsequence_of(std::array{1,2}, std::array{1,2,3}));
static_assert(!is_subsequence_of(std::array{1,2,4}, std::array{1,2,3}));
```

瞬间清爽一万倍，仅仅是因为在 C++ 中类型不是一等公民，只能作为模板参数，在涉及到类型相关的计算的时候，我们就不得不编写繁琐的模板元代码。事实上对类型做计算的需求一直都存在，典型的例子是`std::variant`。在编写`operator=`的时候，我们需要从一个类型列表里面（`variant`的模板参数列表）里面查找某个类型并返回一个索引，其实就是从一个数组里面查找一个满足特定条件的元素。相关的实现这里就不展示了。其实可怕的并不是使用模板元编程本身，而是就 C++ 自身而言，把类型当作值这样的改动是完全不可接受 unacceptable 的。也就是说这样的状况会一直持续下去，以后都不会有什么本质上的改变，这一事实才是最让人悲伤的。不过仍然要清楚的一个事实是，支持对类型做计算的语言并不多，像 Rust 对于这方面的支持几乎没有。C++ 的代码虽然写起来蹩脚，但是至少能写。

但是还好这里有还有另外一条路可以走。就是通过一些手段把类型映射到值。例如把类型映射到字符串，匹配类型可以类似于匹配字符串，只要对字符串进行计算就好了，也能实现一定程度上的`type as value`。C++23之前 并没有标准化的手段进行这种映射，通过一些特殊的编译器扩展能做到，可以参考 C++ 中如何优雅进行 enum 到 string 的转换

```
template<typename ...Ts>
struct type_list{};

template<typename T, typename ...Ts>
constexpr std::size_t find(type_list<Ts...>)
{
    //type_name用于获取编译期类型名
    constexpr std::array arr{ type_name<Ts>()... };
    for(auto i = 0; i < arr.size(); i++)
    {
        if(arr[i] == type_name<T>())
        {
            return i;
        }
    }
}
```

在 C++23 之后也可以直接用typeid实现映射，而不使用字符串映射。但是类型映射到值简单，把值映射到类型回去可一点都不简单，除非你利用 STMP 这种黑魔法，才能方便的把值映射回类型。但是，如果静态反射将来被引入，那么这种从类型和值的双向映射会非常简单。这样的话虽然不能直接支持把类型当成值来进行操作，但是也基本差不多了。不过还有很长一段路要走，具体什么时候能加入标准，还是个未知数。如果对静态反射感兴趣，可以下面这篇文章

编译期变量之痛

除了上面说的对类型做计算不得不用到模板元编程之外，如果需要在编译期计算的同时实例化模板，也不得不用模板元编程。

```
consteval auto test(std::size_t length)
{
    return std::array<std::size_t, length>{};
    //error length is not constant expression
}
```

报错的意思就是`length`不是编译期常量，一般认为它属于编译期变量。这样就很让人讨厌了，考虑如下需求：我们要实现一个完全类型安全的`format`。也就是说根据第一个常量字符串的内容，来约束后面函数参数的个数。比如是`"{}"` 的话，后面`format`的函数参数个数就是`1`个

```
consteval auto count(std::string_view fmt)
{
    std::size_t num = 0;
```

```
    for(auto i = 0; i < fmt.length(); i++)
    {
        if(fmt[i] == '{' && i + 1 < fmt.length())
        {
            if(fmt[i + 1] == '}')
            {
                num += 1;
            }
        }
    }
    return num;
}

template<typename ...Args>
constexpr auto format(std::string_view fmt, Args&&... args) requires (sizeof ...(Args) == count(fmt)) {}
```

事实上我们并没有办法保证一个函数参数是编译期常量，所以上面的代码是没法编译通过的。想要编译期常量，只能把这部分内容填到模板参数里面去，比如上面的函数可能会最后修改成`format<"{}">(1)`这样的形式。虽然只是形式上的差别，但这无疑给使用者带来了困难。这样来看，也就不难理解为什么`std::make_index_sequence`这样的东西大行其道了。想要真正意义上可以做模板参数的编译期变量，也可以通过 STMP 这种黑魔法做到，但是如前文所述，难以在日常的编程中真正使用它。

心中所向往的

非常值得一提的是，有一个比较新的语言叫 Zig。它解决了上述提到的问腿，不仅支持编译期变量，还支持把把类型作为一等公民来进行操作。得益于 Zig 独特的`comptime`机制，被它标记的变量或代码块都是在编译期执行的。这样一来，我们就可以写出如下的代码：

```
const std = @import("std");

fn is_subsequence_of(comptime sub: anytype, comptime super: anytype) bool
{
    comptime
    {
        var subIndex = 0;
        var superIndex = 0;
        while (superIndex < super.len and subIndex < sub.len) : (superIndex += 1)
        {
            if (sub[subIndex] == super[superIndex])
            {
                subIndex += 1;
            }
        }
        return subIndex == sub.len;
    }
}

pub fn main() !void
{
    comptime var sub = [_]type{ i32, f32, i64 };
    comptime var super = [_]type{ i32, f32, i64, i32, f32, i64 };
    std.debug.print("{}\n", .{is_subsequence_of(sub, super)});

    comptime var sub2 = [_]type{ i32, f32, bool, i64 };
    comptime var super2 = [_]type{ i32, f32, i64, i32, f32 };
    std.debug.print("{}\n", .{is_subsequence_of(sub2, super2)});
}
```

写出了我们梦寐以求的代码，啊，实在是太优雅了！在对类型计算这方面 Zig 可以说是完胜目前的 C++，感兴趣的读者可以自己去 Zig 官网了解一下，不过在类型计算以外的其它方面，比如泛型和代码生成，Zig 其实做的并不好，这并不是本文的重点，所以就不讨论了。

结尾

到这里文章就结束了，主要对 C++ 模板的不同方面进行探索和讨论。把这庞大的怪物一层层拆开后其实也没那么可怕了，也让我们更加接近它的本质。稍微总结一下吧：

- 模板元编程并不等于编译期计算，现在的 C++ 编译期计算和运行期代码逻辑几乎一致。除非需要对类型进行计算，否则不需要模板元编程
- `requires`解决了 C++ 代码报错冗长的问题？`requires`的确让模板报错更加清晰了，但是并没法完全解决 C++ 报错冗长。因为另一罪魁祸首函数重载和隐式类型转换仍然存在