CPP-Summit

吴咏炜
博览首席咨询师

C++ 编译期编程的过去、现在和未来

议程

1 简介

2 一点点历史

3 模板元编程

4 constexpr

5 变参模板和静态反射

1

简介

什么是编译期编程？

运行期 vs 编译期

普通编程（运行期）

- 代码在**运行**程序的时候执行

编译期编程

- 代码在**编译**程序的时候执行

为什么要使用编译期编程？

奇技淫巧？

制式武器 vs 自定义武器

演讲本身

- 示例为主
  - 有完整可编译的代码
- 展示可能性和方向
- 不讲解的内容
  - 完整可用的库
  - 完整的知识点教授
  - 技巧和陷阱

2

一点点历史

非类型模板参数

```
template<class T, int size> class buffer;
buffer<char*, 1000> glob;
```

(Stroustrup, 1988)

C++ is a powerful enough language—the first such language in our experience—to allow the construction of generic programming components that combine mathematical *precision*, *beauty*, and *abstractness* with the *efficiency* of non-generic hand-crafted code.

—Alex Stepanov

(Stroustrup, 1994)

模板的“滥用”

```
// Prime number computation by Erwin Unruh
template <int i> struct D { D(void*); operator int(); };

template <int p, int i> struct is_prime {
    enum { prim = (p%i) && is_prime<(i > 2 ? p : 0), i -1> :: prim };
    };

template < int i > struct Prime_print {
    Prime_print<i-1> a;
    enum { prim = is_prime<i, i-1>::prim };
    void f() { D<i> d = prim; }
    };

struct is_prime<0,0> { enum {prim=1}; };
struct is_prime<0,1> { enum {prim=1}; };
struct Prime_print<2> { enum {prim = 1}; void f() { D<2> d = prim; } };
#ifndef LAST
#define LAST 10
#endif
main () {
    Prime_print<LAST> a;
    }
```

(Unruh, 1994)

标准 C++ 下可“工作”的版本

```cpp
// Prime number computation by Erwin Unruh

template <int p, int i> struct is_prime {
  enum { prim = (p==2) || (p%i) && is_prime<(i>2?p:0), i-1> :: prim };
};

template<> struct is_prime<0,0> { enum {prim=1}; };
template<> struct is_prime<0,1> { enum {prim=1}; };

template <int i> struct D { D(void*); };

template <int i> struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i, i-1>::prim };
  void f() { D<i> d = prim ? 1 : 0; a.f();}
};

template<> struct Prime_print<1> {
  enum {prim=0};
  void f() { D<1> d = prim ? 1 : 0; };
};

int main() {
  Prime_print<18> a;
  a.f();
}
```

某编译器下的输出

```
unruh.cpp:15:19: error: no viable conversion from 'int' to 'D<17>'
...
unruh.cpp:15:19: error: no viable conversion from 'int' to 'D<13>'
...
unruh.cpp:15:19: error: no viable conversion from 'int' to 'D<11>'
...
unruh.cpp:15:19: error: no viable conversion from 'int' to 'D<7>'
...
unruh.cpp:15:19: error: no viable conversion from 'int' to 'D<5>'
...
unruh.cpp:15:19: error: no viable conversion from 'int' to 'D<3>'
...
unruh.cpp:15:19: error: no viable conversion from 'int' to 'D<2>'
...
```

https://godbolt.org/z/o9G9M8Mx9

3

模板元编程

阶乘 – 数学定义

$$n! = n \cdot (n-1)!$$
$$0! = 1$$

阶乘 – 模板元编程

```cpp
template <int N>
struct factorial {
    static const int value = N * factorial<N - 1>::value;
};

template <>
struct factorial<0> {
    static const int value = 1;
};
```

“函数”的定义

阶乘 – 模板元编程的展开

```
factorial<3>::value
3 * factorial<2>::value
3 * (2 * factorial<1>::value)
3 * (2 * (1 * factorial<0>::value))
3 * (2 * (1 * 1))
```

"函数"的调用

C++ 里的函数式编程

- 一个类模板表达了一个“函数”
- 模板的实例化相当于“纯函数调用”
- 一个“变量”（类模板里的静态成员变量）只能赋值一次
- 实例化的结果在下面的编译过程中被记住，相当于记忆化
- ……

编译期条件

```
template <bool Condition, typename Then, typename Else>
struct conditional;

template <typename Then, typename Else>
struct conditional<true, Then, Else> {
    using type = Then;
};

template <typename Then, typename Else>
struct conditional<false, Then, Else> {
    using type = Else;
};
```

编译期循环

```
template <bool Condition, typename Body>
struct loop_result;

template <typename Body>
struct loop_result<true, Body> {
    using type = typename loop_result<Body::next_type::condition,
                                      typename Body::next_type>::type;
};

template <typename Body>
struct loop_result<false, Body> {
    using type = typename Body::type;
};

template <typename Body>
struct loop {
    using type = typename loop_result<Body::condition, Body>::type;
};
```

编译期循环 – 用继承简化

```cpp
template <bool Condition, typename Body>
struct loop_result;

template <typename Body>
struct loop_result<true, Body>
    : loop_result<Body::next_type::condition,
                  typename Body::next_type> {};

template <typename Body>
struct loop_result<false, Body> : Body {};

template <typename Body>
struct loop : loop_result<Body::condition, Body> {};
```

阶乘 – 改用循环

```cpp
template <int N, int Last, int Result>
struct factorial_loop {
    static const bool condition = (N <= Last);
    using type = integral_constant<int, Result>;
    using next_type = factorial_loop<N + 1, Last, Result * N>;
};

template <int N>
struct factorial : loop<factorial_loop<1, N, 1>>::type {};
```

汇编结果

```cpp
printf("%d\n",
        factorial<10>::value);
```

编译期计算的结果

```asm
.LC0:
        .string "%d\n"
main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 3628800
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        pop     rbp
        ret
```

https://godbolt.org/z/j59ErdnTj https://godbolt.org/z/Ea8xcff7e

模板元编程之质数计算

算法

1. 产生 2..*n* 的列表

2. 列表首项是质数

4. 再递归筛

```
primesTo n = sieve [2..n]
           where
           sieve (x:xs) = x:(sieve $
                             filter (\a -> a `mod` x /= 0)
                                    xs)
           sieve [] = []
```

3. 剩余项里过滤掉首项除得尽的数

5. 空列表即递归终结

工具 – 编译期列表

```
struct nil {};

template <typename Head, typename Tail = nil>
struct list {};
```

```
// 可理解成
struct list {
    any head;
    list* tail{nullptr};
};
```

工具 – 编译期过滤

类模板声明（原型）

```
template <template <typename> class Pred, typename List>
struct filter;

template <template <typename> class Pred, typename Head, typename Tail>
struct filter<Pred, list<Head, Tail> > {
    typedef
        typename conditional<Pred<Head>::value,
                             list<Head, typename filter<Pred, Tail>::type>,
                             typename filter<Pred, Tail>::type>::type type;
};

template <template <typename> class Pred>
struct filter<Pred, nil> {
    typedef nil type;
};
```

条件分支

特化（主要过滤式）

递归调用

特化（终结条件）

工具 - 整数和类型的相互转换

```
template <typename T, T Val>
struct integral_constant {
    static const T value = Val;
    typedef T value_type;
    typedef integral_constant type;
};

// 类型
integral_constant<int, 42>
integral_constant<bool, true>
// 值
integral_constant<int, 42>::value    // 42
integral_constant<bool, true>::value // true
```

工具 – 产生序列

主模板

```
template <int First, int Last>
struct range {
    typedef list<integral_constant<int, First>,
                 typename range<First + 1, Last>::type>
        type;
};

template <int Last>
struct range<Last, Last> {
    typedef nil type;
};
```

特化（终结条件）

筛法求质数

```
template <typename T>
struct sieve_prime;

template <typename Head, typename Tail>
struct sieve_prime<list<Head, Tail> > {
    template <typename T>
    struct is_not_divisible
        : integral_constant<bool, (T::value % Head::value) != 0> {};

    typedef list<Head, typename sieve_prime<
                           typename filter<is_not_divisible, Tail>::type>::type>
        type;
};

template <>
struct sieve_prime<nil> {
    typedef nil type;
};
```

```
template <int N>
struct primes_to : sieve_prime<typename range<2, N + 1>::type>::type {};
```

```
        mov     esi, 2
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        mov     esi, 3
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        mov     esi, 5
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
```

Output (0/0) x86-64 gcc 11.2 - *1862ms (81029B) ~3668 lines filtered*

Output of x86-64 gcc 11.2 (Compiler #1)

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

https://godbolt.org/z/xE7MKca4s

C++ 模板是图灵完全的！

求质数的等效 Scheme 代码

```scheme
(define (sieve-prime lst)
  (cond
    [(null? lst) '()]
    [else (let ([n (car lst)])
            (let ([is-not-divisible
                   (lambda (m) (not (= (remainder m n) 0)))])
              (cons n (sieve-prime (filter is-not-divisible
                                           (cdr lst))))))]))

(define (primes-to n) (sieve-prime (range 2 (+ n 1))))
```

模板元编程的问题

- 难写，“土”
- 编译慢（慢于 Haskell/Scheme 代码的运行速度）
- 可能耗费大量内存
- 容易让编译器崩溃，甚至让系统失去响应

4

constexpr

const 的问题之一

```cpp
template <int N>
struct factorial {
  static const int value =
    N * factorial<N - 1>::value;
};

template <>
struct factorial<0> {
  static const int value = 1;
};
```

```cpp
template <typename T>
void print_value(const T& value)
{
  …
}

print_value(factorial<10>::value);
```

undefined reference to `factorial<10>::value'

https://godbolt.org/z/TW4GTKPYc

constexpr 变量

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

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

```
template <typename T>
void print_value(const T& value)
{
  …
}

print_value(factorial<10>::value);
```

https://godbolt.org/z/c3Gh13eWb

C++11 的 constexpr 函数

```
constexpr int factorial(int n)
{
    return n == 0 ? 1 : n * factorial(n - 1);
}
```

https://godbolt.org/z/8Gfjxx8fv

C++14 的 constexpr 函数

```
constexpr int factorial(int n)
{
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

https://godbolt.org/z/3PE43PTn4

C++17 的编译期筛子

array 支持 constexpr 方法

```
template <int N>
constexpr auto sieve_prime()
{
    array<bool, N + 1> sieve{};
    for (int i = 2; i <= N; ++i) {
        sieve[i] = true;
    }
    for (int p = 2; p * p <= N; p++) {
        if (sieve[p]) {
            for (int i = p * p; i <= N; i += p) {
                sieve[i] = false;
            }
        }
    }
    return sieve;
}
```

编译期数质数（C++20）

```
template <size_t N>
constexpr size_t prime_count(const array<bool, N>& sieve)
{
    return count(cbegin(sieve), cend(sieve), true);
}
```

count 是 constexpr 函数

编译期数质数（C++17）

```
template <size_t N>
constexpr size_t prime_count(const array<bool, N>& sieve)
{
    size_t count = 0;
    for (size_t i = 2; i < sieve.size(); ++i) {
        if (sieve[i]) {
            ++count;
        }
    }
    return count;
}
```

结果转为 array

```
template <int N>
constexpr auto get_prime_array()
{
    constexpr auto sieve = sieve_prime<N>();
    array<int, prime_count(sieve)> result{};
    for (size_t i = 2, j = 0; i < sieve.size(); ++i) {
        if (sieve[i]) {
            result[j] = i;
            ++j;
        }
    }
    return result;
}
```

```
        .long   2
        .long   3
        .long   5
        .long   7
        .long   11
        .long   13
        .long   17
```

Output (0/0) x86-64 gcc 11.2 - 956ms (76770B) ~4228 lines filtered

Output of x86-64 gcc 11.2 (Compiler #1)

A Wrap lines Select all

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193
```

https://godbolt.org/z/hjezWEf7v

编译性能

模板元编程

N = 1000

- GCC：约 1.2 秒
- Clang：编译器崩溃
- MSVC：fatal error C1202

constexpr 函数

N = 10000

- GCC：约 0.7 秒
- Clang：约 0.8 秒
- MSVC：约 1.3 秒

一些限制

- 编译期常量只能作为模板参数传递
  - 函数参数在函数里永远不能用作编译期常量
- C++20 之前 vector/string 完全不能在 constexpr 函数里使用
  - 不允许任何动态内存分配
- C++20 开始 vector/string 可以在 constexpr 函数里临时使用
  - 可以有动态内存分配；但函数返回前必须完全释放
  - 不能传递给运行期；不能声明为 constexpr 变量
  - 编译期用 vector 筛子求质数示例：https://godbolt.org/z/6c833fE4r

5

变参模板和静态反射

变参模板的主要用法

- 转发不定数量的参数到其他函数（通常结合转发引用）
- 通过递归调用或折叠表达式来对参数进行遍历处理

折叠表达式

```
template <typename... Args>
constexpr bool is_any_null(const Args&... args)
{
    return (... || (args == nullptr));
}
```

对反射结构体的编译期遍历

```
template <typename T, typename F, size_t... Is>
constexpr void forEachImpl(T&& obj, F&& f, std::index_sequence<Is...>)
{
    using TDecay = std::decay_t<T>;
    (void(f(typename TDecay::template _field<T, Is>(obj).name(),
            typename TDecay::template _field<T, Is>(obj).value())),
     ...);
}

template <typename T, typename F>
constexpr void forEach(T&& obj, F&& f)
{
    using TDecay = std::decay_t<T>;
    forEachImpl(std::forward<T>(obj), std::forward<F>(f),
                std::make_index_sequence<TDecay::_field_count>{});
}
```

(罗能, 2020)

打印反射结构体

```
template <typename T>
void dumpObj(const T& obj, std::ostream& os = std::cout,
             const char* fieldName = "", int depth = 0)
{
    if constexpr (IsReflected_v<T>) {
        os << indent(depth) << fieldName << (*fieldName ? ": {\n" : "{\n");
        forEach(obj, [depth, &os](const char* fieldName, auto&& value) {
            dumpObj(value, os, fieldName, depth + 1);
        });
        os << indent(depth) << "}" << (depth == 0 ? "\n" : ",\n");
    } else {
        os << indent(depth) << fieldName << ": " << obj << ",\n";
    }
}
```

dumpObj 的输出效果

```
DEFINE_STRUCT(
    Point,
    (double) x,
    (double) y);

DEFINE_STRUCT(
    Rect,
    (Point) p1,
    (Point) p2,
    (uint32_t) color);

dumpObj(Rect{
    {1.2, 3.4},
    {5.6, 7.8},
    12345678,
});
```

```
{
    p1: {
        x: 1.2,
        y: 3.4,
    },
    p2: {
        x: 5.6,
        y: 7.8,
    },
    color: 12345678,
}
```

https://godbolt.org/z/384af5xc9

静态反射

- 标准中尚未支持
- 目前需要通过宏和编译期编程技巧来实现
- 编译时静态展开，高性能！

将来的静态反射？

反射

```
template <typename T>
void dumpObj(const T& obj, std::ostream& os = std::cout,
             const char* fieldName = "", int depth = 0)
{
    if constexpr (std::is_class_v<T>) {
        os << indent(depth) << fieldName << (*fieldName ? ": {\n" : "{\n");
        template for (constexpr meta::info member :
                          meta::members_of(^T, meta::is_nonstatic_data_member)) {
            dumpObj(obj.[:member:], os, meta::name_of(member), depth + 1);
        }
        os << indent(depth) << "}" << (depth == 0 ? "\n" : ",\n");
    } else {
        os << indent(depth) << fieldName << ": " << obj << ",\n";
    }
}
```

编译期遍历

逆反射

https://cppx.godbolt.org/z/88vvchrna

总结

- 编译期编程提供了在编译期进行推理、计算的能力
- 模板元编程在 C++ 发展的早期提供编译期编程的能力
- 现代 C++ 提供了更多编译期编程的特性，更加易用
- 未来 C++ 可能提供更多编译期的支持功能，如静态反射
  - 希望大约会是在 26……

参考资料

参考资料

Wyatt Childers et al. 2022. Scalable Reflection in C++ (P1240R2). http://wg21.link/p1240r2

Bjarne Stroustrup. 1988. Parameterized Types for C++. Proc. USENIX C++ Conference, Denver, CO. Also, USENIX Computer Systems, Vol 2 No 1. Winter 1989. https://www.usenix.org/legacy/publications/compsystems/1989/win_stroustrup.pdf

Bjarne Stroustrup. 1994. *The Design and Evolution of C++*. Addison-Wesley.

Andrew Sutton et al. 2020. Expansion Statements (P1306R1). http://wg21.link/p1306r1

Andrew Sutton et al. 2021. The Syntax of Static Reflection" (P2320R0). http://wg21.link/p2320r0

Erwin Unruh. 1994. Primzahlen. http://www.erwin-unruh.de/primorig.html

Todd L. Veldhuizen. 2003. C++ Templates are Turing Complete. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670

罗能. 2020. 如何优雅的实现 C++ 编译期静态反射. https://netcan.github.io/2020/08/01/%E5%A6%82%E4%BD%95%E4%BC%98%E9%9B%85%E7%9A%84%E5%AE%9E%E7%8E%B0C-%E7%BC%96%E8%AF%91%E6%9C%9F%E9%9D%99%E6%80%81%E5%8F%8D%E5%B0%84/

胶片网址

https://github.com/adah1972/cpp_summit_2022

谢谢观看