

CPP-Summit

2023全球C++及系统软件技术大会

# C++之静态反射

吴咏炜

博览首席咨询师

# ■ 目录

1 背景

2 实现方法

3 枚举的反射

4 结构体的反射

5 总结

# 1

## 背景



Python

反射有啥稀奇?

Java







**关键字：静态**



# 什么是静态反射

程序在**编译期**检视自己的代码并且  
(在一定程度上) 生成新代码的能力  
—Andrew Sutton

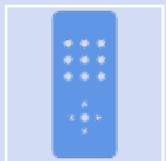
## ■ 静态反射的特点



### 完美契合零开销抽象

你不用的东西，你就不需要付出代价

你使用的东西，你手工写代码也不会更好



### 高性能

能达到一般运行期反射（如 Java 里的）完全不可能做到的性能



### 编译期反射可以用于运行期

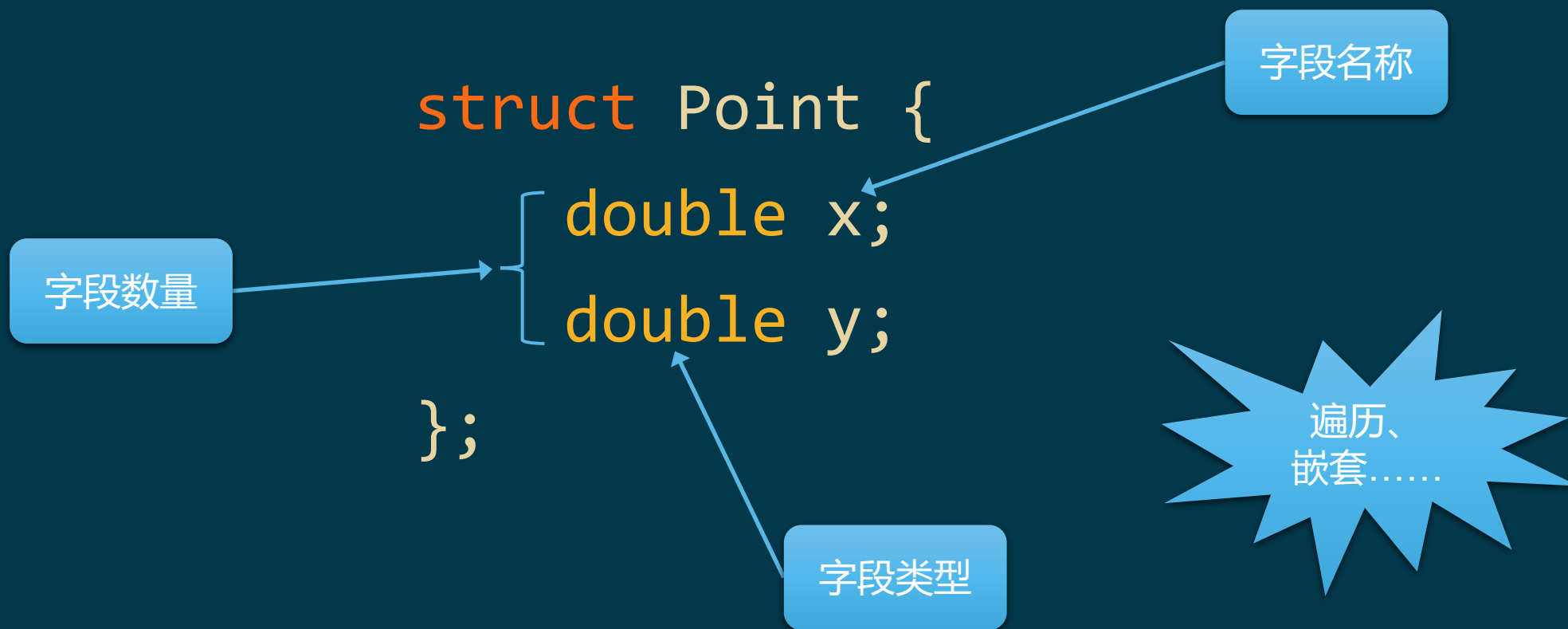
反之则不成立

# 2

## 实现方法



## ■ 静态反射的关注点



# 工具箱

- 模板
- 特化
- 编译期常量、函数、条件语句
- 变参模板、折叠表达式
- 宏展开和宏技巧
- .....

## ■ 宏工具箱

- GET\_ARG\_COUNT 获取宏参数的数量
  - GET\_ARG\_COUNT(a, b, c) → 3
- STRING 把参数变成字符串
  - STRING(foo) → "foo"
- PAIR 将 (type)name 脱去第一层括号
  - PAIR((long)v1) → long v1
- STRIP 将 (type)name 去掉类型部分
  - STRIP((long)v1) → v1
- REPEAT\_ON 重复展开
  - REPEAT\_ON(func, a, b, c) → func(0, a) func(1, b) func(2, c)
- .....





# 可工作的代码！

<https://github.com/adah1972/mozi>

# 3

## 枚举的反射

# ■ 枚举反射的现有方案

- magic\_enum

[https://github.com/Neargye/magic\\_enum](https://github.com/Neargye/magic_enum)

- 需要 C++17 支持，需要较新的编译器
- 优点：不需要特殊的语法；功能较多
- 限制：对枚举值的范围有要求；对不在枚举项列表里的枚举值支持不佳

- better-enums

<https://github.com/aantron/better-enums>

- 支持所有 C++ 标准（对 C++11 有优化），对编译器无特殊要求
- 优点：使用简单，对枚举值范围无要求
- 限制：定义出来的类型本身不是枚举；无法处理不在枚举项列表里的枚举值



## ■ 手工反射方案概要

- 使用宏，生成的实际类型仍然是枚举
- 额外通过 inline constexpr 变量提供枚举项和名字的映射
- 通过 to\_string 等函数重载支持额外的操作

```
DEFINE_ENUM_CLASS(Color, int, red = 1, green, blue);
```

```
cout << to_string(Color::red) << '\n';  
cout << to_string(Color{9}) << '\n';
```

```
red  
(Color)9
```

## ■ 手工反射方案实现

```
#define DEFINE_ENUM_CLASS(e, u, ...) \
    enum class e : u { __VA_ARGS__ }; \
    inline constexpr std::array<std::pair<u, std::string_view>, \
                                GET_ARG_COUNT(__VA_ARGS__)> \
        e##_enum_map_{REPEAT_FIRST_ON(ENUM_ITEM, e, __VA_ARGS__)}; \
    ENUM_FUNCTIONS(e, u)
```

```
enum class Color : int { red = 1, green, blue };
inline constexpr std::array<std::pair<int, std::string_view>, 3>
    Color_enum_map_{
        ENUM_ITEM(0, Color, red = 1),
        ENUM_ITEM(1, Color, green),
        ENUM_ITEM(2, Color, blue),
    };
...
```

## ENUM\_ITEM 的展开

```
inline constexpr std::array<std::pair<int, std::string_view>, 3>
    Color_enum_map_{
        std::pair{to_underlying(Color((eat_assign<Color>)Color::red = 1)),
            remove_equals("red = 1")},
        std::pair{to_underlying(Color((eat_assign<Color>)Color::green)),
            remove_equals("green")},
        std::pair{to_underlying(Color((eat_assign<Color>)Color::blue)),
            remove_equals("blue")},
    };

```



# ENUM\_FUNCTIONS 的展开

```
inline std::string to_string(Color value)
{
    return enum_to_string(to_underlying(value), "Color",
                          Color_enum_map_.begin(), Color_enum_map_.end());
};
```

```
template <typename Int, typename Iterator>
std::string enum_to_string(Int value, const char* enum_name,
                          Iterator first, Iterator last)
{
    std::string result;
    auto it = std::find_if(
        first, last, [&](const auto& pr) { return pr.first == value; });
    if (it != last) {
        result = it->second;
    } else {
        result = "(";
        result += enum_name;
        result += ")";
        result += std::to_string(value);
    }
    return result;
}
```

# ■ 使用提案 P2996R1 中的静态反射

```
template <typename E>
    requires std::is_enum_v<E>
std::string to_string(E value)
{
    template for (constexpr auto e : std::meta::members_of(^E)) {
        if (value == [:e:]) {
            return std::string(std::meta::name_of(e));
        }
    }

    return std::string("(") + std::meta::name_of(^E) + ")" +
        std::to_string(to_underlying(value));
}
```

遍历枚举成员

生成反射信息

获得枚举项名字

获得枚举名字

从反射信息生成 C++ 实体  
(此处是枚举项)

# ■ 使用 P2320 的模拟验证

- 简单版本：线性搜索

<https://cppx.godbolt.org/z/8rWTcf1KP>

- 复杂版本：收集枚举项、排序，而后再进行二分查找

<https://cppx.godbolt.org/z/P5Ycdv3xj>



# 4

## 结构体的反射

# ■ 结构体反射的现有方案

- Boost.PFR
  - 基于 C++14 的通用结构体反射库
  - 优点：类型定义无需宏；简单易用，支持遍历、比较、输出等
  - 限制：不能访问类型和字段的名字
- struct\_pack
  - 基于 C++17 的高性能序列化/反序列化库
  - 优点：类型定义无需宏；支持复杂类型；性能高
  - 限制：仅用于序列化/反序列化；没有设计为用于其他用途，或序列化格式自定义的场景

## ■ 手工反射方案概要

- 使用宏，生成的实际类型仍然是结构体，大小不变化
- 通过额外的静态成员提供后续操作需要的信息
- 通过公用函数模板实现常见的操作

```
DEFINE_STRUCT(  
    S1,  
    (long)v1,  
    (bool)v2  
);  
  
DEFINE_STRUCT(  
    S2,  
    (S1)values,  
    (std::string)msg  
);
```

```
S2 s2{{1, true}, "test"};  
print(s2);  
  
{  
    values: {  
        v1: 1,  
        v2: true  
    },  
    msg: "test"  
}
```

## ■ 一个复杂的应用

```
DEFINE_STRUCT(S1,  
    (uint16_t)v1,  
    (uint16_t)v2,  
    (uint32_t)v3,  
    (uint32_t)v4,  
    (string)msg  
);  
DEFINE_STRUCT(S2,  
    (int)v2,  
    (long)v4  
);  
S1 s1{...};  
...  
S2 s2;  
copy_same_name_fields(s1, s2);
```

```
movzx    eax, WORD PTR s1[rip+2]  
mov      DWORD PTR s2[rip], eax  
mov      eax, DWORD PTR s1[rip+8]  
mov      QWORD PTR s2[rip+8], rax
```



```
s2.v2 = s1.v2;  
s2.v4 = s1.v4;
```



## ■ 手工反射方案实现

```
#define DEFINE_STRUCT(st, ...) \
    struct st { \
        using is_reflected = void; \
        template <typename, size_t> \
        struct _field; \
        static constexpr size_t _size = GET_ARG_COUNT(__VA_ARGS__); \
        REPEAT_ON(FIELD, __VA_ARGS__) \
    }
```

```
struct S1 { \
    using is_reflected = void; \
    template <typename, size_t> \
    struct _field; \
    static constexpr size_t _size = 2; \
    FIELD(0, (long)v1) \
    FIELD(1, (bool)v2) \
};
```

# FIELD 的展开

```
struct S1 {  
    ...  
    long v1;  
    template <typename T>  
    struct _field<T, 0> {  
        using type = decltype(decay_t<T>::v1);  
        static constexpr auto name = STRING(v1);  
        constexpr _field(T&& obj) : obj_(std::forward<T>(obj)) {}  
        constexpr decltype(auto) value() { return (std::forward<T>(obj_).v1); }  
        T&& obj_;  
    };  
    ...  
};
```

可在编译期使用的字段类型

可在编译期使用的字段名称

字段访问函数

外层对象 (s1) 的引用

## ■ 识别对静态反射的支持

```
template <typename T, typename = void>
struct is_reflected_struct : false_type {};
```

```
template <typename T>
struct is_reflected_struct<T, void_t<typename T::is_reflected>>
    : true_type {};
```

```
template <typename T>
inline constexpr static bool is_reflected_struct_v
    = is_reflected_struct<T>::value;
```

## ■ 对字段的访问

```
template <
    size_t I, typename T,
    enable_if_t<is_reflected_struct_v<decay_t<T>>, int> = 0>
constexpr decltype(auto) get(T&& obj)
{
    using DT = decay_t<T>;
    static_assert(I < DT::_size, "Index to get is out of range");
    return typename DT::template _field<T, I>(std::forward<T>(obj)).value();
}
```



## ■ 编译期遍历

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

```
template <typename T, typename F, enable_if_t<is_reflected_v<decay_t<T>>, int> = 0>
constexpr void for_each(T&& obj, F&& f)
{
    using DT = decay_t<T>;
    for_each_impl(std::forward<T>(obj), std::forward<F>(f),
                  make_index_sequence<DT::_size>{});
}
```

`for_each(s1, f) → f(0, S1::_field<S1&, 0>::name, get<0>(s1)); f(1, ...); ...`

## ■ 编译期遍历使用示例

```
template <typename T>
constexpr long add_all_fields(const T& obj)
{
    long result{};
    for_each(obj, [&](auto /*index*/, auto /*name*/, const auto& value) {
        if constexpr (is_integral_v<decay_t<decltype(value)>>) {
            result += value;
        }
    });
    return result;
}

constexpr data::S2 s2{2, 4, true};
constexpr auto result = add_all_fields(s2);
CHECK(result == 7);
```

## ■ 编译期遍历两个对象

```
template <typename T, typename U, typename F, size_t... Is>
constexpr void zip_impl(T&& obj1, U&& obj2, F&& f, index_sequence<Is...>)
{
    (void(std::forward<F>(f)(
        decay_t<T>::template _field<T, Is>::name, decay_t<U>::template _field<U, Is>::name,
        get<Is>(std::forward<T>(obj1)), get<Is>(std::forward<U>(obj2))))) ,
        ...);
}

template <typename T, typename U, typename F,
          std::enable_if_t<(is_reflected_struct_v<decay_t<T>> &&
                           is_reflected_struct_v<decay_t<U>>), int> = 0>
constexpr void zip(T&& obj1, U&& obj2, F&& f)
{
    static_assert(decay_t<T>::_size == decay_t<U>::_size);
    zip_impl(std::forward<T>(obj1), std::forward<U>(obj2),
              std::forward<F>(f), make_index_sequence<decay_t<T>::_size>{});
}
```

比较

复制

## ■ 可扩展的复制框架

```
template <typename T, typename U, typename = void>
struct copier {
    void operator()(const T& src, U& dest) const { dest = src; }
    void operator()(T&& src, U& dest) const { dest = std::move(src); }
};

template <typename T, typename U>
constexpr void copy(T&& src, U& dest)
{
    copier<remove_cvref_t<T>, remove_cvref_t<U>>{}(
        std::forward<T>(src), dest);
}
```

# ■ 反射结构体的复制

```
template <typename T, typename U>
struct copier<T, U,
             std::enable_if_t<is_reflected_struct_v<T> && is_reflected_struct_v<U>>> {
    void operator()(const T& src, U& dest) const
    {
        zip(src, dest,
            [](auto /*name1*/, auto /*name2*/, const auto& value1, auto& value2) {
                copy(value1, value2);
            });
    }
    void operator()(T&& src, U& dest) const
    {
        zip(std::move(src), dest,
            [](auto /*name1*/, auto /*name2*/, auto&& value1, auto& value2) {
                copy(std::forward<decltype(value1)>(value1), value2);
            });
    }
};
```



# ■ 问题

- 为什么不用 `memcpy`?
  - 因为结构体成员里也许有非 POD 类型, 如 `string`
- 为什么不用 “=” 复制?
  - 因为我们可以支持异构结构体复制
- 为什么要支持异构结构体复制?
  - 因为..... 😊

# ■ 异构结构体复制场景 – 字节序转换

```
DEFINE_STRUCT(  
    MsgHdrHost,  
    (uint32_t)src_addr,  
    (uint32_t)dst_addr,  
    (uint16_t)len,  
    (uint16_t)flags  
);
```

```
DEFINE_STRUCT(  
    MsgHdrNet,  
    (net_uint32)src_addr,  
    (net_uint32)dst_addr,  
    (net_uint16)len,  
    (net_uint16)flags  
);
```

```
MsgHdrHost hdr_host{...};  
MsgHdrNet  hdr_net{};  
copy(hdr_host, hdr_net);  
dump(hdr_host);  
dump(hdr_net);
```

01	00	A8	C0	02	00	A8	C0	2C	01	08	10
C0	A8	00	01	C0	A8	00	02	01	2C	10	08

## net\_int 的实现

```
constexpr uint16_t host_to_net(uint16_t value)
{
    return ((value << 8) & 0xFF00) |
           (value >> 8);
}

constexpr uint32_t host_to_net(uint32_t value)
{
    return ((value << 24) & 0xFF000000) |
           ((value << 8) & 0xFF0000) |
           ((value >> 8) & 0xFF00) |
           (value >> 24);
}

...
```

优化效果见 <https://godbolt.org/z/1M6eM5zc8>

```
template <typename T>
class net_int {
public:
    net_int() = default;
    constexpr net_int(T value) :
        net_value_(host_to_net(value)) {}
    constexpr explicit operator T() const
    { return net_to_host(net_value_); }

private:
    T net_value_;
};

using net_uint8 = uint8_t;
using net_uint16 = net_int<uint16_t>;
using net_uint32 = net_int<uint32_t>;
...
```

## ■ 结构体内容提取

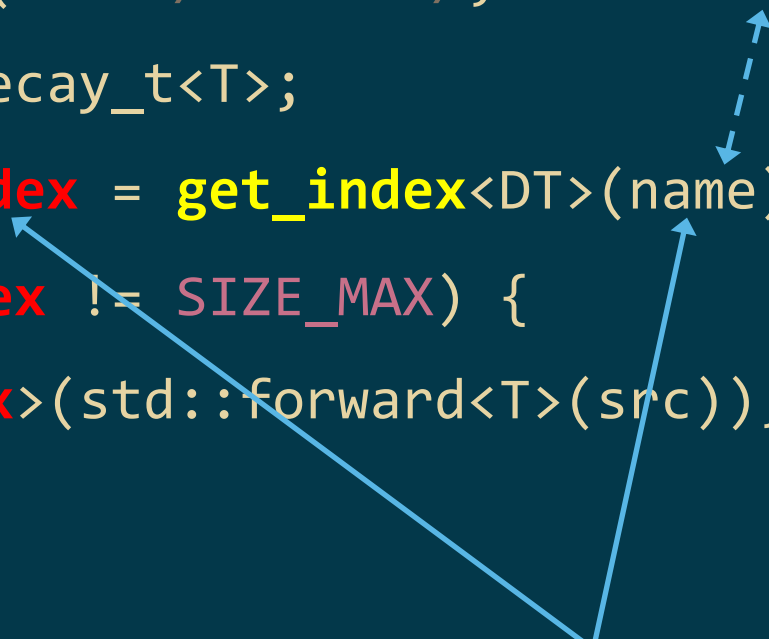
```
DEFINE_STRUCT(  
    BookInfo,  
    (int)id,  
    (int)author_id,  
    (int)publisher_id,  
    (int)publish_year,  
    (string)name,  
    (string)isbn,  
    (string)author,  
    (string)publisher  
);
```

```
SELECT name, publish_year WHERE author_id = ...;
```

```
DEFINE_STRUCT(  
    BookInfoNameYear,  
    (string)name,  
    (int)publish_year  
);  
  
BookInfoNameYear record{};  
vector<BookInfoNameYear> result;  
Container<BookInfo> container;  
while (...) {  
    auto it = container.find(...);  
    ...  
    copy_same_name_fields(*it, record);  
    result.push_back(record);  
}
```

## ■ 编译期的字段名参数问题

```
for_each(dest, [&src](auto /*index*/, auto name, auto& value) {  
    using DT = std::decay_t<T>;  
    constexpr auto index = get_index<DT>(name);  
    if constexpr (index != SIZE_MAX) {  
        copy(get<index>(std::forward<T>(src)), value);  
    }  
});
```



如何通过**参数**字段名来得到**编译期常量**索引？



# ■ 问题：函数的任何参数都不被视为编译期常量

## 解决方案

- Mozi 主干上的解决方法，使用编译期字符串
  - 需要 GCC/Clang 或 C++20
  - 实现原理参见 <https://accu.org/journals/overload/30/172/wu/>
- 把编译期常量参数映射到一个类型，稍后从类型里还原出参数
  - 可以同 `true` 和 `true_type` 的关系类比
  - 一般使用 lambda 实现
  - Mozi 的 no-cts-reflection 分支有实现示意
  - 更多描述参见 <https://mpark.github.io/programming/2017/05/26/constexpr-function-parameters/>

# ■ 从手工方案到 P2996R1 静态反射

- `S::_field` →

`constexpr std::meta::info member : std::meta::nonstatic_data_members_of(^T)`

- `S::_field<T, I>::name` →

`std::meta::name_of(member)`

- `S::_field<T, I>::type` →

`typename [:std::meta::type_of(member):]`

- `for_each(obj, [](..., auto& value) { value ... } )` →

`template for (constexpr std::meta::info member : ...) { obj.[:member:]... }`

- 函数参数不是编译期常量问题自动消解——member 是 constexpr 变量

# ■ 使用提案 P2996R1 中的静态反射

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

遍历结构体数据成员

生成反射信息

获得字段名字

从反射信息生成 C++ 实体 (此处是数据成员)

使用 P2320 的模拟验证: <https://cppx.godbolt.org/z/c7a4jfo74>

## Mozi 里 print 的嵌套数据结构输出效果

```
{  
  1 => {  
    v1: {1},  
    v2: 2,  
    v4: 3  
  },  
  2 => {  
    v1: {1},  
    v2: 2,  
    v4: 3  
  },  
  3 => {  
    v1: {1},  
    v2: 2,  
    v4: 3  
  }  
}
```

```
{  
  value: 37,  
  tup: ({  
    v2: 2,  
    v4: 4  
  }, "Great", {  
    s2: {  
      v1: {1},  
      v2: 2,  
      v4: 3  
    },  
    value: 42  
  }, {  
    {  
      v2: 1,  
      v4: 2  
    },  
    {  
      v2: 3,  
      v4: 4  
    }  
  })  
}
```

# 5

## 总结



# 静态反射 能做的事

- 调试输出
- 自动化的比较
- ORM
- 序列化
- .....

# ■ Mozi 项目

## 已完成

- 枚举和结构体的反射支持
- 可扩展的 copy 机制
- 可扩展的 print 机制
- 单元测试

## 待完成

- 比较机制
- 序列化?
- .....



# ■ 零开销验证

```
#include <stdio.h>
```

```
enum class Number : int {  
    zero, one, two, three  
};
```

```
struct S1 {  
    Number value;  
};
```

```
int main()  
{  
    S1 s1{Number::one};  
    printf("%d\n",  
        static_cast<int>(s1.value));  
}
```

```
#include <stdio.h>
```

```
#include "mozi/enum_reflection_core.hpp"
```

```
#include "mozi/struct_reflection_core.hpp"
```

```
DEFINE_ENUM_CLASS(Number, int,  
    zero, one, two, three  
);
```

```
DEFINE_STRUCT(S1,  
    (Number)value  
);
```

```
int main()  
{  
    S1 s1{Number::one};  
    printf("%d\n",  
        static_cast<int>(s1.value));  
}
```

两段程序产生的汇编代码无实质差异

## ■ 一些参考

- Paul Fultz II. 2012. “Is the C preprocessor Turing complete?”  
<https://pfultz2.com/blog/2012/05/10/turing/>
- Paul. Fultz II. 2012. “C++ Reflection in under 100 lines of code”.  
<https://pfultz2.com/blog/2012/07/31/reflection-in-under-100-lines/>
- 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/>

# ■ 静态反射的标准化

- 2013 年 SG-7 (反射研究组) 成立
- Matúš Chochlík, Axel Naumann, and David Sankel. 2017. “Static Reflection in a Nutshell”. <http://wg21.link/p0578r1>
- David Sankel (Ed.). 2018. “Working Draft, C++ Extensions for Reflection”. <http://wg21.link/n4766>
- Andrew Sutton and Herb Sutter. 2018. “Value-based Reflection”. <http://wg21.link/p0993r0>
- Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2018. “Scalable Reflection in C++”. <http://wg21.link/p1240r0>
- Andrew Sutton, Wyatt Childers, and Daveed Vandevoorde. 2021. “The Syntax of Static Reflection”. <http://wg21.link/p2320r0>
- Wyatt Childers, Peter Dimov, Barry Revzin, Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2023. “Reflection for C++26”. <http://wg21.link/p2996r1>

# 谢谢

