C++ 反射 TS 初探

CPP开发者 2022-01-23 12:05

翻译 | 杨文波
原文作者 | CLÉMENT PIRELLI

最近，Matus Chochlik[1] 在 clang 的一个分叉中实现了《C++ 反射扩展[2] N4856》技术规范（或 TS），可以在这里[3]把玩。我自然对此很感兴趣，但相关的 cppreference[4] 页面看起来还是非常空洞。

因此，在花了一点时间研究这个 TS 之后，我想解释一下它到底是怎么回事，以及如何用它来正经做些事情。

在这篇文章中，我将解释该规范的基本想法，如何编写一个简单的泛型“枚举到字符串”函数，并略微深入探究一个概念验证性质的序列化函数的细节。

请注意，在所有的代码中，我用 namespace reflect = std::experimental::reflect 以求简短。另外，我还是建议在电脑上而不是在移动设备上阅读本文。

1. 基本想法

基本上整件事情都基于概念，而概念的本质是对通用类型的约束。我这里不做太详细的介绍，但对于不熟悉的人来说，这篇文章[5] 是很好的介绍，你也可以在这里[6]找

到更多关于概念的文档。

反射 TS 增加了一个新的关键字，reflexpr，它返回一个所谓的“反射元对象类型”，基本上就是一个符合 Object 概念的类型。还有其他一些概念对 Object 进行精化，比如 Variable、ObjectSequence、Lambda 等。在这里，精化只是意味着“约束得更多”。

一旦你有了元对象类型，就可以通过元函数询问它一些东西，比如 get_name、is_class、get_public_data_members 等等。所有这些元函数都受到相当合理的约束，例如，你只能对 Record 调用 get_public_data_members，而 Record 是约束了类、结构体和联合体的一个概念。请注意，和其他的 C++ 标准元函数一样，这些元函数通常有 _t 或 _v 的简写形式，所以你可以写出比如 stuff_v<T> 而不是 stuff<T>::value。

2. 枚举到字符串

枚举只有三个专门的元函数，而这里我们只关注 get_enumerators，它会给我们一个 ObjectSequence 类型以便进一步处理：**（下滑滚动看完整代码，下同）**

```
template<typename T> // 令人困惑的是，还有个 reflect::is_enum(_v)，它告诉你
consteval auto enum_names() requires std::is_enum_v<T>
{
    // 这就是 reflexpr；reflected_enum 是一个元对象，它反射 T
    using reflected_enum = reflexpr(T);

    // enum_enumerators 是个 ObjectSequence，它包含每个枚举值的元对象类型
    using enum_enumerators = reflect::get_enumerators_t<reflected_enum>

    // 更多内容马上来
}
```

ObjectSequence 是一个代表一系列 Object 的概念；它本身有几个元函数，特别是 get_element。有了它，我们可以写一个函数，对 ObjectSequence 中每个元素应用另一个元函数并将结果装入一个数组：

```
template<
   // 我们要应用的特征，请别太担心语法
   template<typename> typename Trait_t,
   // 将要应用它到这个序列
   reflect::ObjectSequence Sequence_t,
   // 帅帅的变参，这样我们可以对每个元素应用这个运算
   size_t... ints>
consteval auto make_object_sequence_array(std::index_sequence<ints...>)
{
    // Trait_t 得有一个 value 成员
    return std::array { Trait_t<reflect::get_element_t<ints, Sequence_t
}
```

这有点不好马上消化，但本质上它只是将 Trait_t 应用于 Sequence_t 的每个元素。index_sequence 需要一个大小，这我们可以通过 get_size 元函数获得，该元函数会给出 ObjectSequence 中的元素数量。现在就可以调用 make_object_sequence_array：

```
template<typename T>
consteval auto enum_names() requires std::is_enum_v<T>
{
    using reflected_enum = reflexpr(T);
    using enum_enumerators = reflect::get_enumerators_t<reflected_enum>

    // 拿到 ObjectSequence enum_enumerators 的大小
    constexpr auto T_size = reflect::get_size_v<enum_enumerators>;
    using sequence = std::make_index_sequence<T_size>;

    // 对 ObjectSequence 中的每个元素应用 get_name, 结果以数组形式返回
```

```
    return make_object_sequence_array<reflect::get_name, enum_enumerato
}
```

在 Compiler Explorer[7] 上测试，我们看到它完美地工作了。现在就可以颇为轻松地写出我们的 enum_to_string 函数：

```
template<typename T>
constexpr auto enum_to_string(const T value) requires std::is_enum_v<T>
{
    // 这里又有点容易搞混，有个 reflect::underlying_type，它返回的是某 refle
    using underlying_type = std::underlying_type_t<T>;
    const auto underlying_value = static_cast<underlying_type>(value);

    // 轻松
    return enum_names<T>()[underlying_value];
}
```

我就此搁笔，直到 Barry Revzin[8] 提出来，嗯，这压根不行！它假定枚举从零开始，而且所有的值都是连续的，但情况并不总这样，所以得返工。可以通过在我们的元函数中使用 get_constant 来解决这个问题，它会返回一对常量和名称，像这样：

```
template<typename T> requires reflect::Constant<T> && reflect::Named<T>
struct get_constant_and_name
{
    static constexpr auto value = std::pair { reflect::get_constant_v<T
};
```

把 enum_names 中的 get_name 换成 get_constant_and_name，现在它返回由 pair 组成的数组，这样就行了[9]。现在我们需要对 enum_to_string 函数做点修改，就成了：

```
template<typename T>
constexpr auto enum_to_string(const T value) requires std::is_enum_v<T>
{

    // 此处可以是个 std::find_if, 甚至是 ranges::find，但让我们相对从简
    for(const auto&pair : enum_names<T>())
    {
        // 提醒下， first 是值, second 是名称
        if(value == pair.first)
        {
            return pair.second;
        }
    }

    // 比如用标志位做入参调用可能就会这样
    return "Unnamed value";
}
```

注意，这仍然不能处理一个枚举中有两个枚举项值相同的情况。

3. 序列化—概念验证

这里不会有任何实际的序列化代码——我把它作为练习留给读者。那么，要快速展示的是如何迭代遍历一个类型并相当自动地将其分解为可序列化的部分。为此，我们将使用一个递归的模板函数，草稿我们可以这样打：

```
template<typename T>
```

```
void serialize(const T& value)
{
    // Collection 是我写的一个概念，如你对其实现感兴趣，请稍后留意 Comp
    if constexpr(Collection<T>)
    {
        // 对每个元素调用 serialize，很容易
        for(const auto& element : value)
        {
            serialize(element);
        }
    } else
    if constexpr(std::is_class_v<T>)
    {
        // 在这里我们应该将 T 分解为它的成员，分离出成员变量并对他们调用
    } else
    {
        // 在这里我们处理那些原始类型
    }
}
```

处理原始类型基本上只是在一堆 if constexprs 里处理字符串、算术类型、枚举（对于它，也许应该使用我们的 enum_to_string 函数！），等等。

有意思的部分在于如果 T 是一个类。这时 get_public_data_members 就能派上用场了：

```
if constexpr(std::is_class_v<T>)
{
    // 这又是一个反射 T 的元对象类型
    using Reflected_t = reflexpr(T);

    // 也许我们并不想序列化 private 成员，尽管也有支持它们的 get_data_member
    using data_members = reflect::get_public_data_members_t<Reflected_t

    // 又是从大小中得到一个 index_sequence
    constexpr auto T_size = reflect::get_size_v<data_members>;
```

```
    using sequence = std::make_index_sequence<T_size>;

    // 然后呢？
}
```

这里有点棘手。我们想对 data_members 的每个成员使用 get_pointer 从而得到指向类数据成员的指针[10]，但是如果我们把 get_pointer 用在一个元对象类型之上，它就没有 value 成员，因为元对象类型并不是个 Variable。我通过创建自己的元函数包装 get_pointer 来绕过这个问题。如果它的 T 符合概念 Variable，那么它就调用 get_pointer。否则，它返回 std::monostate，表示这个元对象应该被忽略：

```
template<typename T>
struct get_pointer_or_monostate
{
private:
    static constexpr auto get_value()
    {
        if constexpr(reflect::Variable<T>)
        {
            return reflect::get_pointer<T>::value;
        }
        else
        {
            return std::monostate{};
        }
    }
public:
    static constexpr auto value = get_value();
};
```

有了这个新的（也许非常绕的，如果你找到了更好的方法，请告诉我）元函数，我们现在可以用它来获取数据成员指针和 monostate 组成的元组：

```
// 和 make_object_sequence_array 一样，只是现在返回一个 std::tuple
constexpr auto pointer_or_monostate_tuple = make_object_sequence_tuple<

// 下面是将模板化的 lambda 表达式应用到元组的每个元素上去（ 更好地施展 std::ap
apply_operation_on_tuple([&value](auto current_value)
{
    using current_value_t = decltype(current_value);

    // "not" 挺酷的，来告我呗；这里我们检查 current_value 类型，如果不是 mon
    if constexpr(not std::same_as<current_value_t, std::monostate>)
    {
        // 现在就清楚了，我们能够通过使用指向类数据成员的指针得到该成员的值
        const auto& member = value.*current_value;
        // 这样我们就可以对该成员调用 serialize
        serialize(member);
    }
}, pointer_or_monostate_tuple);
```

好，大致上这就可以了。把这些扔进 Compiler Explorer[11]，我实际上还没有写任何序列化代码，但放了几个 std::cout 进去，打印表明我们正确地遍历要序列化的数据，行了！

4. 结束语

我对 C++ 最感到懊恼的方面之一是它缺乏作为语言核心部分的（好的——对不起，typeid）反射。编译器一定知道的简单东西，比如枚举名称，通常要用当前的 C++ 手动编写，或者不得不依赖像 magic enum[12] 这样的库，并承受由此带来的所有缺点。

就我所知，这个 TS 的命运还没有决定。显而易见的是，我刚才展示的代码并不容易懂，也不容易编写。和使用模板一贯的体验一样（尽管概念已经改善了它），找出某个东西哪里不工作是非常痛苦的。还有对编译速度的担忧，因为这是大量的使用泛型的编译期工作。

尽管如此，TS 为 C++ 特有的许多流行弊病提供了一个难以置信的强大解决方案，所有这些都在一个一致的、建立在标准其他部分之上的软件包中，并且它以大多数模板爱好者相当容易辨识的方式做到了这点。比如，我没有提到对命名空间的反射，但是你能用它做成的事情是难以想象的。

参考文献：

[1] Matus Chochlik: *https://twitter.com/matus_chochlik*

[2] C++ 反射扩展: *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4856.pdf*

[3] 这里: *https://compiler-explorer.com/z/TrYEYhqMK*

[4] cppreference: *https://en.cppreference.com/w/cpp/experimental/reflect*

[5] 这篇文章: *https://www.cppstories.com/2021/concepts-intro/*

[6] 这里: *https://en.cppreference.com/w/cpp/language/constraints*

[7] Compiler Explorer: *https://compiler-explorer.com/z/c7fGojMbT*

[8] Barry Revzin: *https://twitter.com/BarryRevzin*

[9] 行了: *https://compiler-explorer.com/z/8KsnG6eKc*

[10] 指向类数据成员的指针: *https://en.cppreference.com/w/cpp/language/pointer#:~:text=Pointers%20to%20data%20members*

[11] Compiler Explorer: *https://compiler-explorer.com/z/PfM7dTWxr*

[12] magic enum: *https://github.com/Neargye/magic_enum*

- EOF -

推荐阅读 — 点击标题可跳转

1、ProtoBuf 反射详解

2、如何优雅地实现 C++ 编译期静态反射

3、为什么建议少用 if 语句！

看精选C/C++技术文章

CPP开发者

我们在 Github 维护着 9000+ star 的C语言/C++开发资源。日常分享 C语言 ...

24篇原创内容

公众号

点赞和在看就是最大的支持❤️

喜欢此内容的人还喜欢

C语言字符串函数strcat | strcpy | strlen | strcmp的用法及原型

C语言题库

如果你还不会用 git 回滚代码，那你一定要来看看

前端瓶子君