

1 简介

现有的命令行解析器使用方式都很神秘，比如

```
options.add_options()  
    ("d,debug", "Enable debugging") // a bool parameter  
    ("i,integer", "Int param", cxxopts::value<int>())
```

看是能看懂，但是总感觉写着不得劲。或许，我想要的是类似这样的东西：

```
const auto spec = R"  
-d, --debug: # 默认为 bool  
    Enable debugging.  
  
-i, --integer [int]:  
    Int param.  
)"
```

然后自动生成一个结构体

```
struct Options {  
    bool debug;  
    int integer;  
};
```

之后只要 `Options opt = parse(argc, argv);` 就可以了。这就非常清爽了，而且是单独的配置文件，在参数多了之后不需要在 `main()` 里写一长串，人类（和 AI）也很容易阅读。以及，甚至还可以支持 UTF-8:

```
-p, -русский:  
    不管是命令行参数还是描述都可以不是 ASCII。
```

然而在尚不久远的过去，要实现这样的东西，唯一的方式就是用 Python 之类的东西解析这个 DSL，然后生成 C++ 代码，最后编译。这就很麻烦了，毕竟 CMake 这种反人类的东西可不好配置这么复杂的内容。

所幸在 C++26 中，我们获得了静态反射的能力，能够直接在编译期定义新的结构体。而编译期求值的能力早在 C++20 就已经到来了。所以，单头文件的梦想已经可以成真了。

2 安装 C++26

很遗憾，即使是直接从 Github clone 下来的 LLVM，现在也没有完全支持静态反射，甚至 `libcxx` 里都不包括 `<meta>`。目前实现了它的只有 LLVM 的分支 `bloomberg/clang-p2996`。

不过，LLVM 本体就有 4 GB 多，而且二者共享接近 54 万个 commit，差距并没有那么大。我自然是不想克隆两份的。这时，就轮到一些 git 的神秘高级用法了。

```
# 首先 clone; --filter=blob:none 意味着只会按需 clone，不会把之前所有分支的 blob 都复制一份  
git clone --filter=blob:none https://github.com/llvm/llvm-project.git llvm && cd llvm
```

```
# 然后加入 remote 的 p2996 分支  
git remote add bloomberg https://github.com/bloomberg/clang-p2996.git  
git fetch bloomberg p2996:refs/remotes/bloomberg/p2996
```

```
# 再把 remote 分支同步到本地  
git branch p2996 bloomberg/p2996
```

```
# 最后用 worktree 同时保留两个工作区，共用一个 .git  
git worktree add ../llvm-p2996 p2996
```

然后就可以分别编译了。我毕竟是 LLVM 的社区开发者之一，原版 LLVM 肯定是要留下来的。具体而言，我需要启用这些项目：

- clang，不必多说；

- clangd, 也就是 clang 的语言服务器;
- clang-tools-extra, 比如 clang-tidy 之类的;
- compiler-rt, 也就是 ASan, UBSan 什么的;
- libcxx + libcxxabi, 为了 <meta>。

至于 lld 用 LLVM 本体的就行了, 反正链接这种事情也不怎么会变。

实际上, P2996 分支还有各种神秘 bug, 到头来还得自己修。这个还是单开一篇文章说好了。

3 实现

编译原理应该大家都会吧 (真的吗)。要解析, 首先是 lexer, 然后是 parser, 接下来拿着 AST 喂给运行时就可以了。

3.1 Lexer

首先得考虑 UTF-8 的问题。毕竟是生成一个 C++ 结构体, 我们需要保证那些字符真的可以组成标识符。Unicode 倒是提供了一张表, 列出了 XID_Start (可以作为标识符的首字符), 以及 XID_Continue (非首字符) 的所有字符。那么我们只需要把它下载下来, 整理一下就可以了。查找的时候 std::upper_bound 即可, 这个函数也已经是 constexpr 的了。

我们生成的东西类似这样:

```
namespace trait {
    constexpr CodePoint XID_Start[] = { 65, 97, 170, 181, 186, 192, ... }
}
namespace length {
    constexpr unsigned XID_Start[] = { 26, 26, 1, 1, 1, 23, 31, 458, ... }
}
```

实际上是一个区间, 例如前两个就是从 65 (A) 开始, 长度为 26 的区间, 也就是所有的大写字母。第二个明显是所有的小写字母, 以此类推。现在就是时候尝鲜静态反射了: 如何通过 trait::XID_Start 这个名字, 去查找到对应的 length::XID_Start 呢?

只需要首先获取 trait 的标识符 XID_Start, 然后去找 length 命名空间下所有的成员, 逐个对比标识符是否相同就可以了。非常简单。

```
constexpr std::optional<std::meta::info> findLengthTableImpl(std::meta::info trait) {
    const auto name = std::meta::identifier_of(trait);
    for (auto member : std::meta::members_of(^length,
std::meta::access_context::unchecked())) {
        if (std::meta::has_identifier(member) && std::meta::identifier_of(member) == name)
            return member;
    }
    return std::nullopt;
}
```

这里的 std::meta::info 是 ^ 表达式的返回类型。使用的时候, 只需要 findLengthTableImpl(^trait::XID_Start) 即可。

剩下的 UTF-8 解码之类的事情都很标准, 没什么太多说的。不过 tokenization 的过程中, 还需要特别考虑缩进的事情。毕竟描述那个部分出现什么字符串都有可能, 其中的 [,], : 之类的并不能被解析为单独的 token。那么, 我们该如何界定它的开始与结束呢? 我的想法是就靠缩进。

在 : 后, 我要求出现一段连续的空白字符。这个参数描述的每一行必须都以同样的空白字符序列开头, 而这个缩进消失的位置就是下一个命令行参数了。毕竟不会有非常复杂的嵌套, 这样设计应该没有太大问题, 不会像 Python 一样需要游标卡尺。

3.2 Parser

递归下降罢了, LL(1)。

```

spec    := entry* End
entry   := option | group
option  := name (',' name)* type? ':' description?
name    := Shortname | Longname
type    := '[' Ident ('=' Str | '!' | '?')? ']'
group   := '@' Ident ('[' Ident ''])? ':' description?

```

最后解析出来的也不需要是一个树状的 AST，返回一个 `std::array<Argument, N>` 就可以了。毕竟这个东西是需要活到运行时的，所以直接返回 `std::vector<Argument>` 自然是不行的。不过，唯一的缺点是为了得到 N 我们需要 `parse` 两遍，第一遍数出数量，第二遍放入数据。

虽然 C++26 提供了 `std::define_static_array`，但它要求 `Argument` 是一个 structural type，也就是“长得像 struct”的东西。我们的 `Argument` 内部有很多 `std::string_view`，它有私有成员，所以没法这么做。

在获得静态反射之后，我们就可以利用解析结果定义结构体了。首先是把 `string_view` 改写为真正的类型：

```

constexpr std::optional<std::meta::info> resolveType(std::string_view name) {
    if (name == "bool") return ^^bool;
    if (name == "char") return ^^char;
    if (name == "int") return ^^int;
    // ... 一些更多的内置类型

    // 以及，一个给用户的接口：我们会遍历 argparse::types 中的所有类型
    for (auto member : std::meta::members_of(^^argparse::types,
                                              std::meta::access_context::unchecked())) {
        if (!std::meta::has_identifier(member))
            continue;
        if (std::meta::identifier_of(member) != name)
            continue;
        if (std::meta::is_type_alias(member))
            return std::meta::dealias(member);
        if (std::meta::is_type(member))
            return member;
    }
    return std::nullopt;
}

```

接下来，就是真正定义结构体成员的地方了。C++26 提供了显式实例化模板的功能 `std::meta::substitute`，所以我们可以将成员包裹进 `vector` 或者 `optional` 里。考虑到 `vector` 本身还有一个默认的 `Allocator` 模板参数，我们稍微包装了一下 `vector` 本身。

```

template <class T>
using Optional = std::optional<T>;
template <class T>
using Vector = std::vector<T>;

constexpr std::meta::info defineOptions(std::span<const Argument> args,
                                         std::meta::info target) {
    std::vector<std::meta::info> members;

    for (const Argument &argument : args) {
        std::optional<std::meta::info> resolved = resolveType(argument.typeName);
        if (!resolved)
            throw SpecError(std::format("unknown type '{}'", argument.typeName));

        std::meta::info type = *resolved;
        // 处理 ? 和 !
        if (argument.optional)

```

```

        type = std::meta::substitute(^^Optional, {type});
    if (argument.positional)
        type = std::meta::substitute(^^Vector, {type});

    // 新增一个类型为 type, 名称为 field 的成员
    const std::string field = argument.field();
    members.push_back(std::meta::data_member_spec(type, {.name = field}));
}

return std::meta::define_aggregate(target, members);
}

```

在使用的时候，只需要

```

template <String Spec>
struct OptionsDefinition {
    struct Type;
    static constexpr auto arguments = argumentTable<Spec>();
    constexpr { defineOptions(arguments, ^^Type); }
};

```

这里就是将 OptionsDefinition<Spec>::Type 定义为我们上面生成的结构体。我们也选择将 arguments 暴露出来，因为运行期还需要用到它。

3.3 运行时

这就是人体工学设计了。要支持的东西还挺多：

-oout.txt, -o=out.txt, -o out.txt	三种不同的赋值写法
--verbose, --verbose=false, -v true	还有可以省略值的布尔变量
--integer -5	负数也会以 - 开头
-qv	还有缩写（比如 ls -la 就是等价于 ls -l -a）

不过 * 之类的 glob 支持是 shell 自动展开，我们不用管。

好在这一切都是运行期，写正常的 C++ 代码就可以了。当然，之前说过了 argparse::types 可以接受用户自定义的类型，那我们自然也需要接受自定义的解析器。我把它放进了 argparse::convert 里，依靠模板特化 T fromString(string_view)。比如 bool 的解析就是

```

namespace argparse::convert {
template <>
inline bool fromString(std::string_view str) {
    // 只要 ASCII 字符
    if (std::ranges::any_of(str, [](char c) { return (unsigned char) c >= 0x80; }))
        throw InvalidInput();

    std::string s;
    s.reserve(str.size());
    for (char c : str)
        s.push_back(static_cast<char>(std::tolower(static_cast<unsigned char>(c))));
    if (s == "1" || s == "true" || s == "t" || s == "y" || s == "yes" || s == "on")
        return true;
    if (s == "0" || s == "false" || s == "f" || s == "n" || s == "no" || s == "off")
        return false;
    throw InvalidInput();
}
}

```

4 解决符号太长的问题

在 parser 那里我们提到，我们的设计实际上是用 `OptionsDefinition<Spec>::Type`。而 `Spec` 本身可能有几万个字符长，而 `mangle` 之后就更是离谱：

`ELcn29ELcn127ELcn124ELcn29ELcn127ELcn90...`

每一个字符都对应 `ELcn<字节>`，会有 6-7 个字符长。所以每个符号能接近 100 KB。来不及为 VSCode 的崩溃而悲伤，下一个到来的是 clang！什么，我、我打这么长的符号表？

要解决这个问题，只需要略微更改一下就可以了。不直接传入 `String spec`，而是给出

```
class Spec {
    String text = "...";
};
template <class Spec> {...}
```

这样 `mangle` 的符号就只会出现结构体的名字了。不过，我们自然需要限制一下这个 `class` 具有合适的格式，毕竟 `concept` 在 C++20 就已经加入了，不用不就亏了吗？

```
template <class T>
struct IsString : std::false_type {};
template <unsigned Size>
struct IsString<String<Size>> : std::true_type {};

template <class T>
concept Specification = requires {
    requires IsString<std::remove_cvref_t<decltype(T::text)>>::value;
    std::integral_constant<unsigned, T::text.size()>{};
};
```