Clang.jl

在JULIA中轻松调用C接口

同济大学

曾富楠

melonedo.github.io

目录

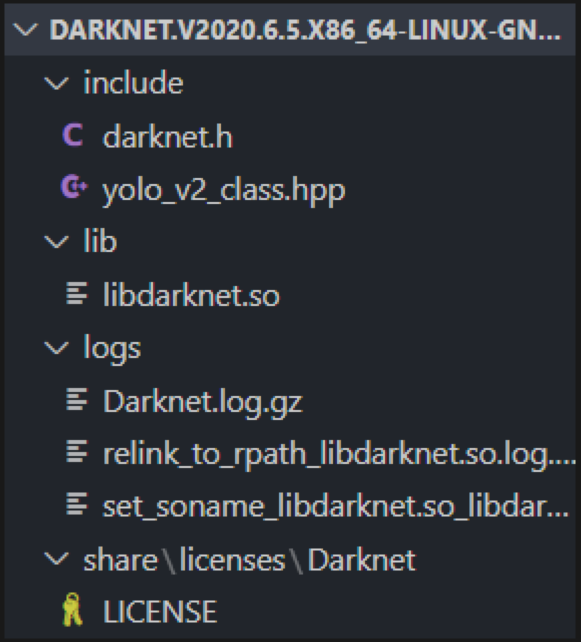
- Julia调用C接口
- 自动生成c接口代码
- Clang.jl 0.14的新特性

C接口

- C库通常包括`.h`头文件和`.c`/`.cpp`源文件
- 头文件作为协议，原样分发
- 源文件编译为（动态）链接库后分发

JLL包

Julia将头文件和库包装在jll包中分发



```
DARKNET.V2020.6.5.X86_64-LINUX-GN...
  include
    darknet.h
    yolo_v2_class.hpp
  lib
    libdarknet.so
  logs
    Darknet.log.gz
    relink_to_rpath_libdarknet.so.log....
    set_soname_libdarknet.so_libdar...
  share\licenses\Darknet
    LICENSE
```

调用C接口

Julia的`ccall`语句可以直接调用C接口，格式为

```
ccall((function_name, library), returntype, (argtype1, ...), a

# 在Julia 1.5中引入
@ccall library.function_name(argvalue1::argtype1, ...)::return
```

在Julia中调用C的动态库只需要根据函数的声明添加对应的接口函数，明确动态库的地址和函数类型。

ccall接口函数

如要调用的c函数声明为

```c
int add(int a, int b);
```

为这个C函数添加Julia接口

```julia
const libadd = "path/to/libadd.so"
function add(a, b)
    @ccall libadd.add(a::Cint, b::Cint)::Cint # Julia 1.5
end
```

调用接口

函数`add`可以和原生的Julia函数同样调用

```
add(1,2) # => 3
```

数据类型兼容

若合理设计数据类型，则在Julia和C间传递数据时，不需要额外的转换函数。

如C中的

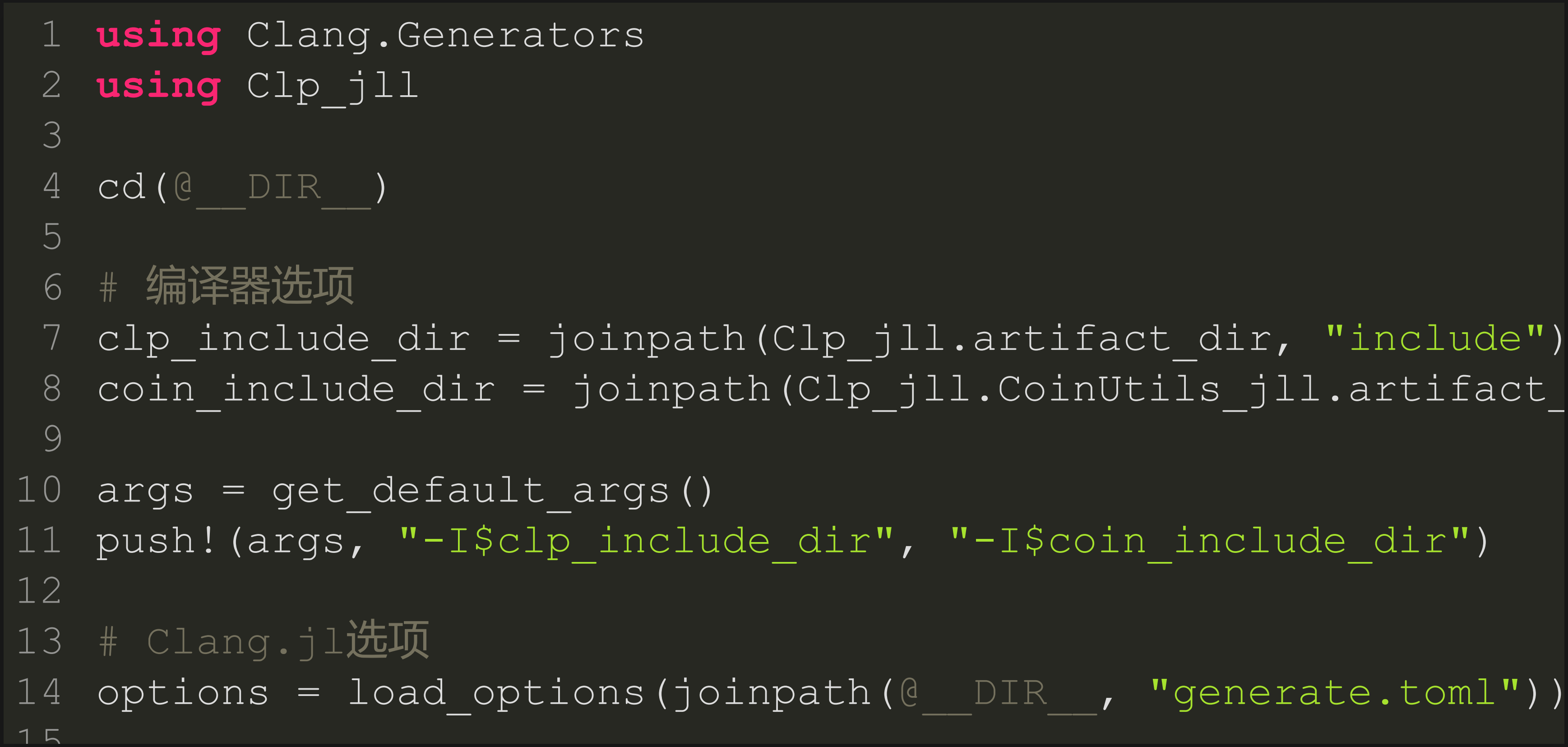
```c
struct S {
    int a;
    float b;
};
```

对应Julia中的

```julia
struct S
    a::Cint
    b::Cfloat
end
```

自动生成JULIA接口

Clang.jl把C头文件中的函数和类型声明转换为Julia中对应的函数和类型。



```julia
using Clang.Generators
using Clp_jll

cd(@__DIR__)

# 编译器选项
clp_include_dir = joinpath(Clp_jll.artifact_dir, "include")
coin_include_dir = joinpath(Clp_jll.CoinUtils_jll.artifact_

args = get_default_args()
push!(args, "-I$clp_include_dir", "-I$coin_include_dir")

# Clang.jl选项
options = load_options(joinpath(@__DIR__, "generate.toml"))
```

Clp.jl/gen/generate.jl

自动生成JULIA接口

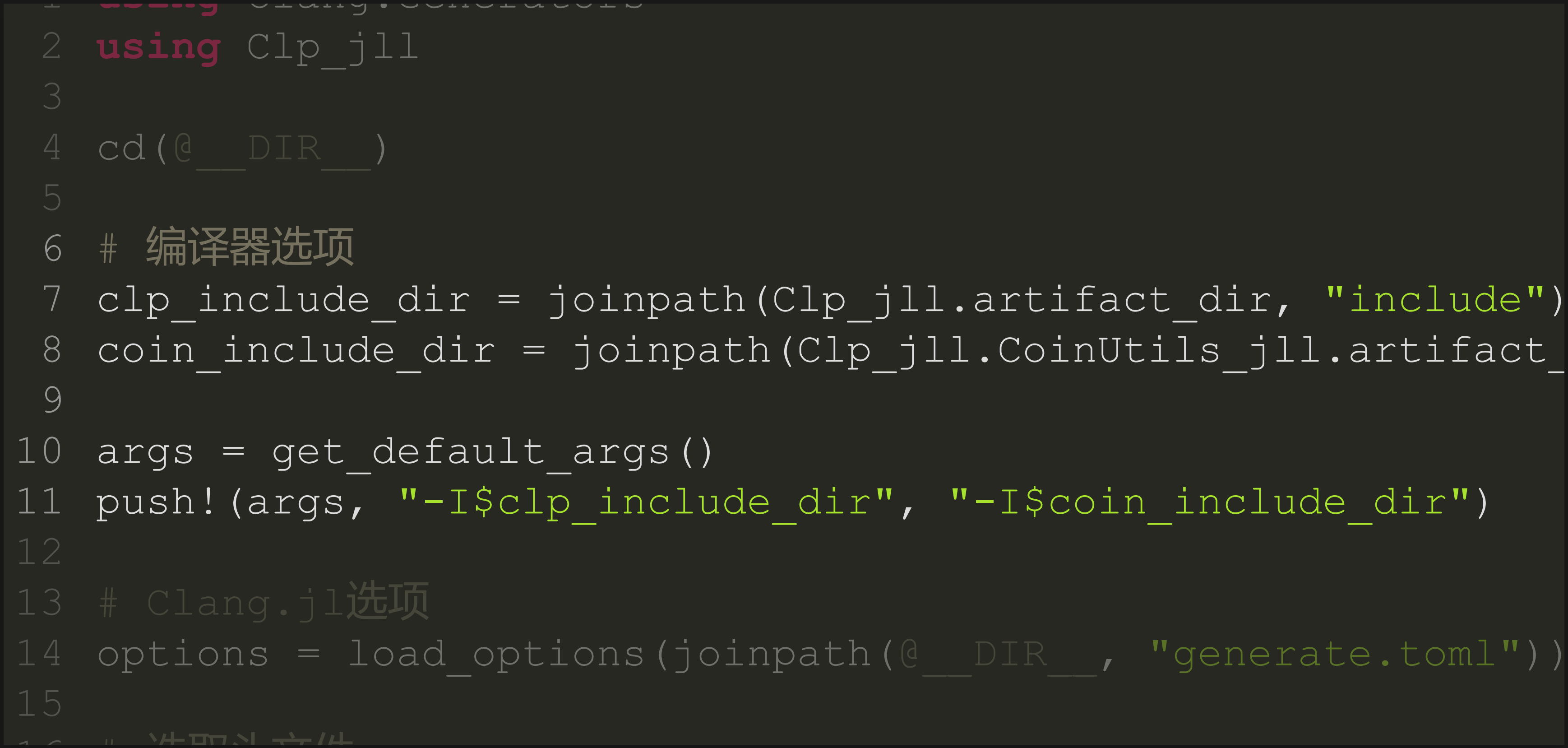
Clang.jl把C头文件中的函数和类型声明转换为Julia中对应的函数和类型。

```julia
using Clang.Generators
using Clp_jll

cd(@__DIR__)

# 编译器选项
clp_include_dir = joinpath(Clp_jll.artifact_dir, "include")
coin_include_dir = joinpath(Clp_jll.CoinUtils_jll.artifact_

args = get_default_args()
push!(args, "-I$clp_include_dir", "-I$coin_include_dir")

# Clang.jl选项
options = load_options(joinpath(@__DIR__, "generate.toml"))
```

Clp.jl/gen/generate.jl

自动生成JULIA接口

Clang.jl把C头文件中的函数和类型声明转换为Julia中对应的函数和类型。

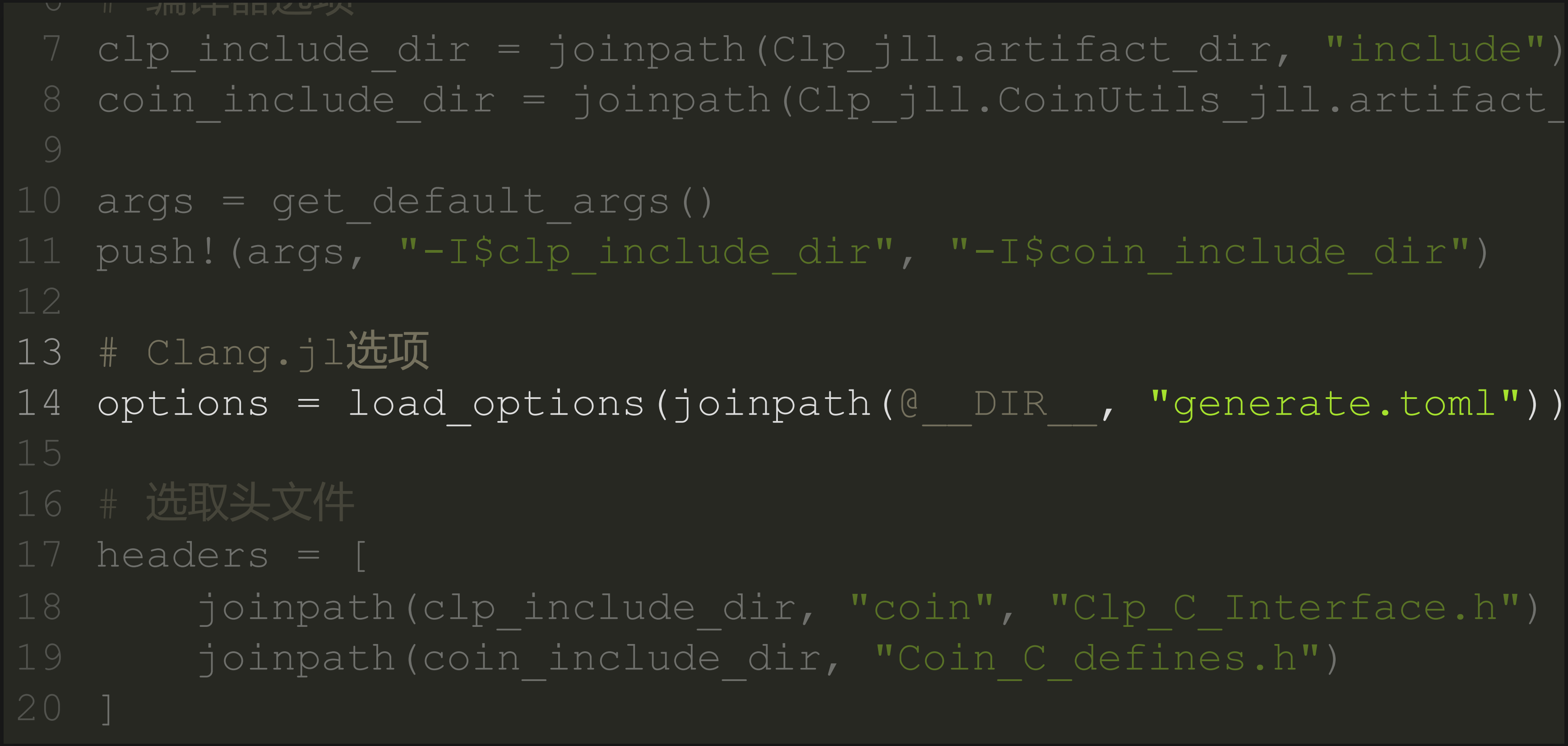


```julia
using Clp_jll

cd(@__DIR__)

# 编译器选项
clp_include_dir = joinpath(Clp_jll.artifact_dir, "include")
coin_include_dir = joinpath(Clp_jll.CoinUtils_jll.artifact_

args = get_default_args()
push!(args, "-I$clp_include_dir", "-I$coin_include_dir")

# Clang.jl选项
options = load_options(joinpath(@__DIR__, "generate.toml"))

```

Clp.jl/gen/generate.jl

自动生成JULIA接口

Clang.jl把C头文件中的函数和类型声明转换为Julia中对应的函数和类型。

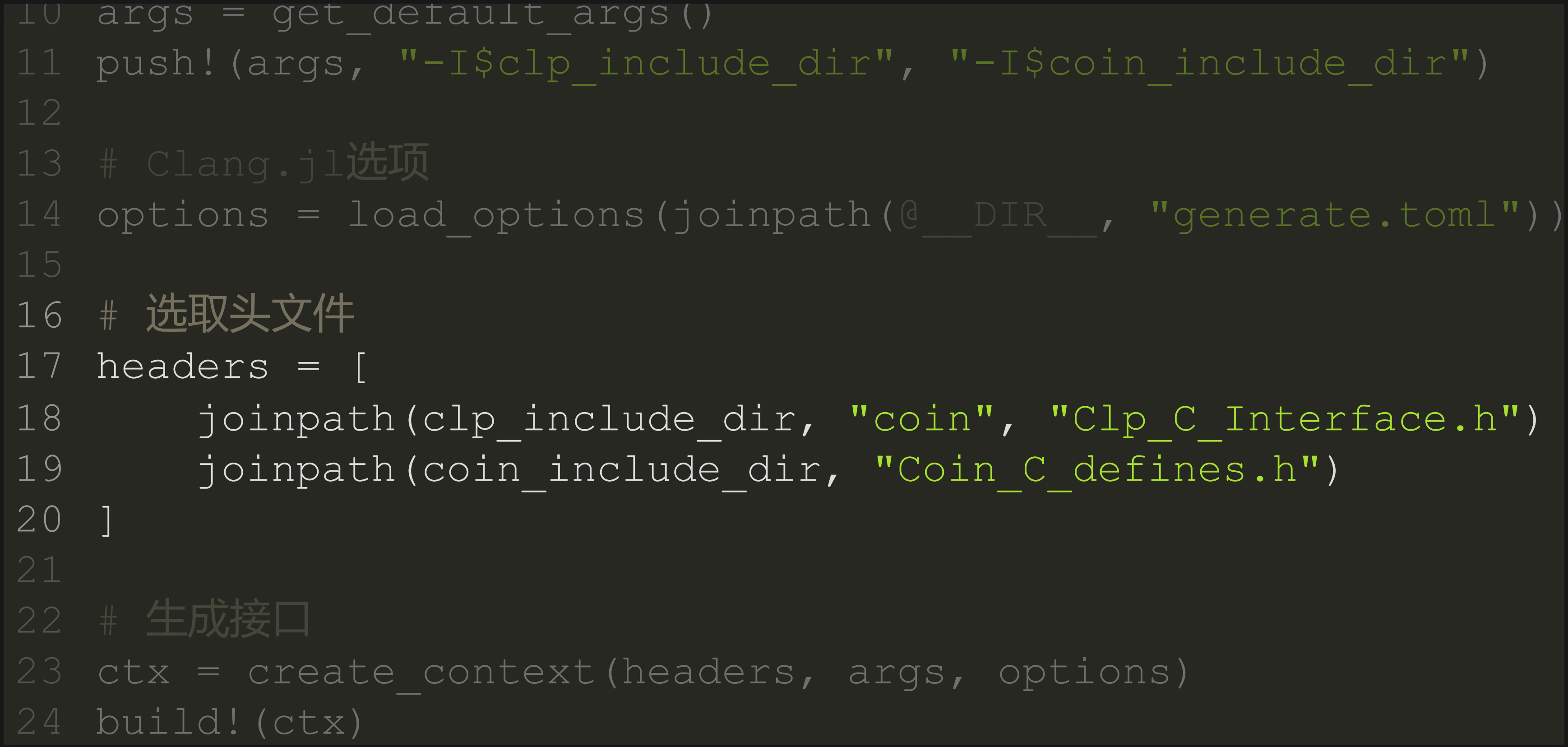


```
# 编译器选项
clp_include_dir = joinpath(Clp_jll.artifact_dir, "include")
coin_include_dir = joinpath(Clp_jll.CoinUtils_jll.artifact_

args = get_default_args()
push!(args, "-I$clp_include_dir", "-I$coin_include_dir")

# Clang.jl选项
options = load_options(joinpath(@__DIR__, "generate.toml"))

# 选取头文件
headers = [
    joinpath(clp_include_dir, "coin", "Clp_C_Interface.h")
    joinpath(coin_include_dir, "Coin_C_defines.h")
]
```

Clp.jl/gen/generate.jl

自动生成JULIA接口

Clang.jl把C头文件中的函数和类型声明转换为Julia中对应的函数和类型。

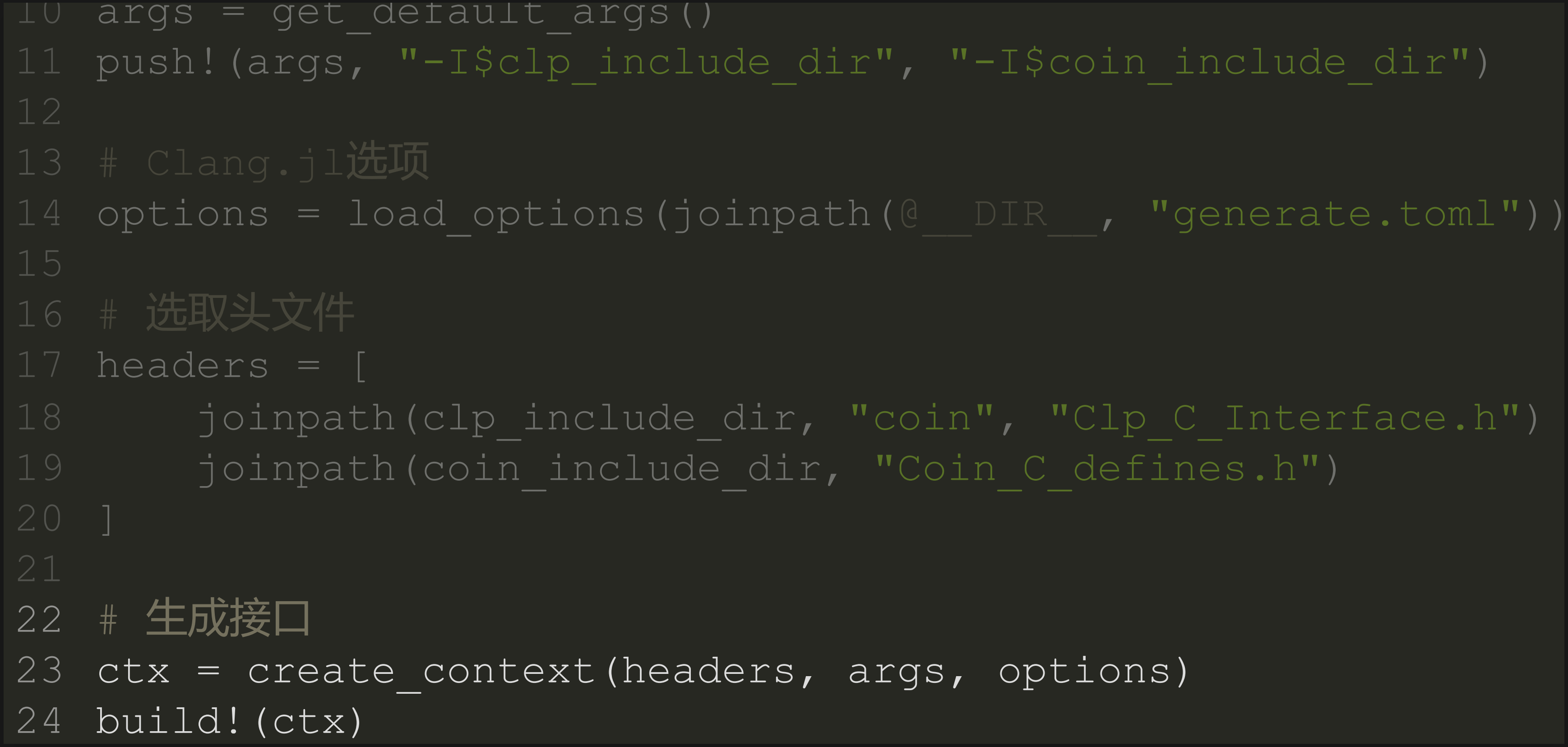


```
args = get_default_args()
push!(args, "-I$clp_include_dir", "-I$coin_include_dir")

# Clang.jl选项
options = load_options(joinpath(@__DIR__, "generate.toml"))

# 选取头文件
headers = [
    joinpath(clp_include_dir, "coin", "Clp_C_Interface.h")
    joinpath(coin_include_dir, "Coin_C_defines.h")
]

# 生成接口
ctx = create_context(headers, args, options)
build!(ctx)
```

Clp.jl/gen/generate.jl

自动生成JULIA接口

Clang.jl把C头文件中的函数和类型声明转换为Julia中对应的函数和类型。



```julia
args = get_default_args()
push!(args, "-I$clp_include_dir", "-I$coin_include_dir")

# Clang.jl选项
options = load_options(joinpath(@__DIR__, "generate.toml"))

# 选取头文件
headers = [
    joinpath(clp_include_dir, "coin", "Clp_C_Interface.h")
    joinpath(coin_include_dir, "Coin_C_defines.h")
]

# 生成接口
ctx = create_context(headers, args, options)
build!(ctx)
```

Clp.jl/gen/generate.jl

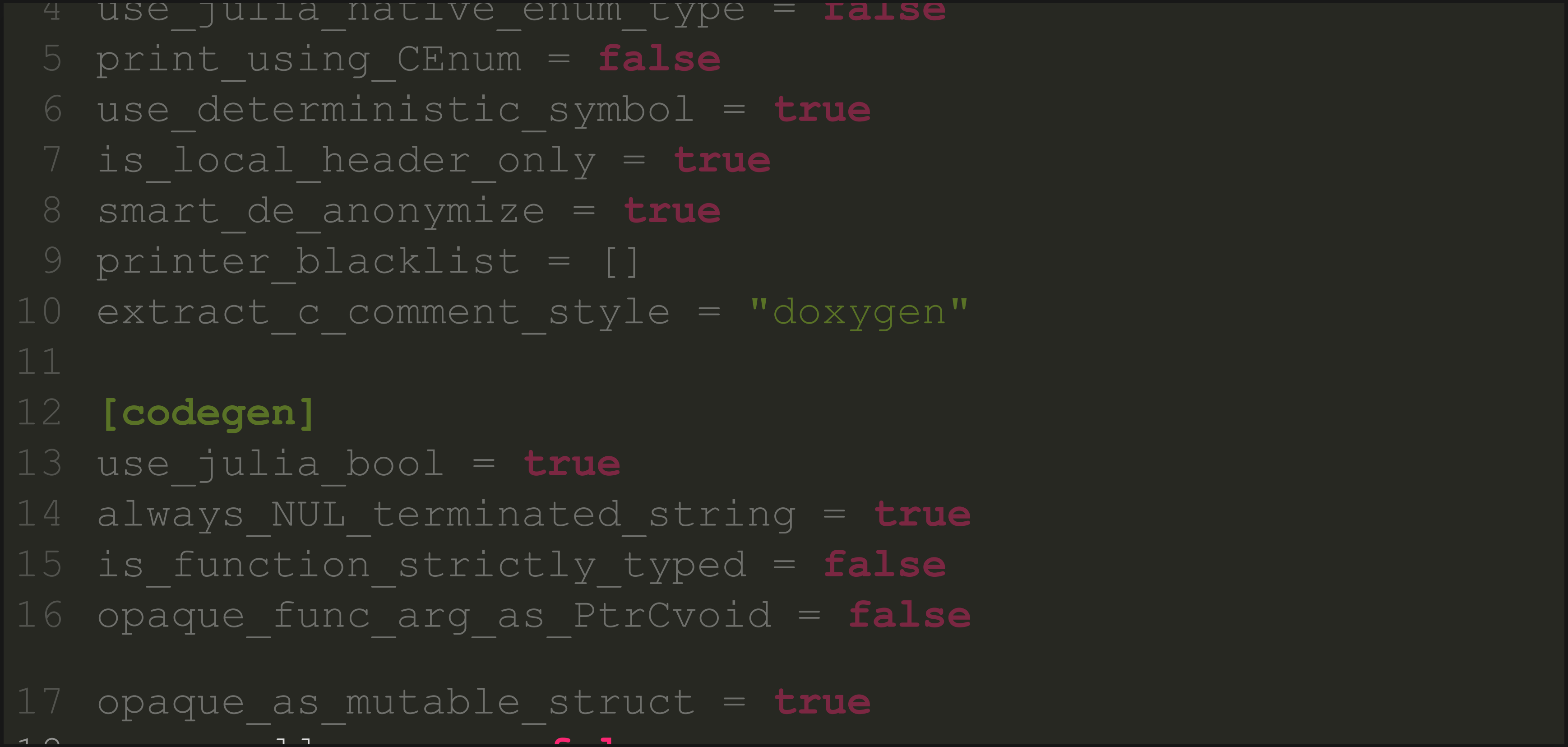
CLANG.JL选项

Clang.jl提供丰富的配置选项，满足各种场景的需求。

```toml
[general]
library_name = "libClp"
output_file_path = "../lib/libClp.jl"
use_julia_native_enum_type = false
print_using_CEnum = false
use_deterministic_symbol = true
is_local_header_only = true
smart_de_anonymize = true
printer_blacklist = []
extract_c_comment_style = "doxygen"

[codegen]
use_julia_bool = true
always_NUL_terminated_string = true
is_function_strictly_typed = false
```

CLANG.JL选项

Clang.jl提供丰富的配置选项，满足各种场景的需求。



```toml
use_julia_native_enum_type = false
print_using_CEnum = false
use_deterministic_symbol = true
is_local_header_only = true
smart_de_anonymize = true
printer_blacklist = []
extract_c_comment_style = "doxygen"

[codegen]
use_julia_bool = true
always_NUL_terminated_string = true
is_function_strictly_typed = false
opaque_func_arg_as_PtrCvoid = false

opaque_as_mutable_struct = true
[illegible]
```

CLANG.JL选项

```
use_ccall_macro = false
```

```
function Clp_resize(model, newNumberRows, newNumberColumns)
    ccall((:Clp_resize, libClp), Cvoid, (Ptr{Clp_Simplex}, Cin
end
```

CLANG.JL选项

```julia
use_ccall_macro = true
```

```julia
function Clp_resize(model, newNumberRows, newNumberColumns)
    @ccall libClp.Clp_resize(model::Ptr{Clp_Simplex}, newNumbe
end
```

CLANG.JL 0.14的新特性

- 可变参数函数
- 位域
- 提取doxygen注释

感谢导师Yupei Qi！

可变参数函数

在Julia 1.5 添加了添加了可变参数调用的支持，比如最经典的`printf`函数

```
int printf(const char *fmt, ...);
```

在Julia中可以用

```
julia> @ccall printf("%s, %d\n"::Cstring; "Hello"::Cstring, 12
Hello, 123
```

的方法调用，只需在可变参数前用分号“；”标注。

可变参数函数接口

为了直接调用`printf`函数，Clang.jl生成如下接口

```
@generated function printf(fmt, va_list...)
    :(@ccall printf(fmt::Ptr{Cchar}; $(to_c_type_pairs(va_list
end
```

- Julia调用时类型不直接给出，必须自动推导参数类型（`to_c_type`）
- `@ccall`的参数是常量，因此用生成函数在编译前指定类型

实例：melonedo/LibCURL.jl

位域

- 指C中长度不是8的整数倍的整数类型
- Julia没有原生支持，需要转化为数组 `NTuple{size, UInt8}`然后重载 `getproperty`和`setproperty!`

```c
struct BitfieldStruct {
    int d:3;
    int e:4;
    unsigned int f:2;
};
```

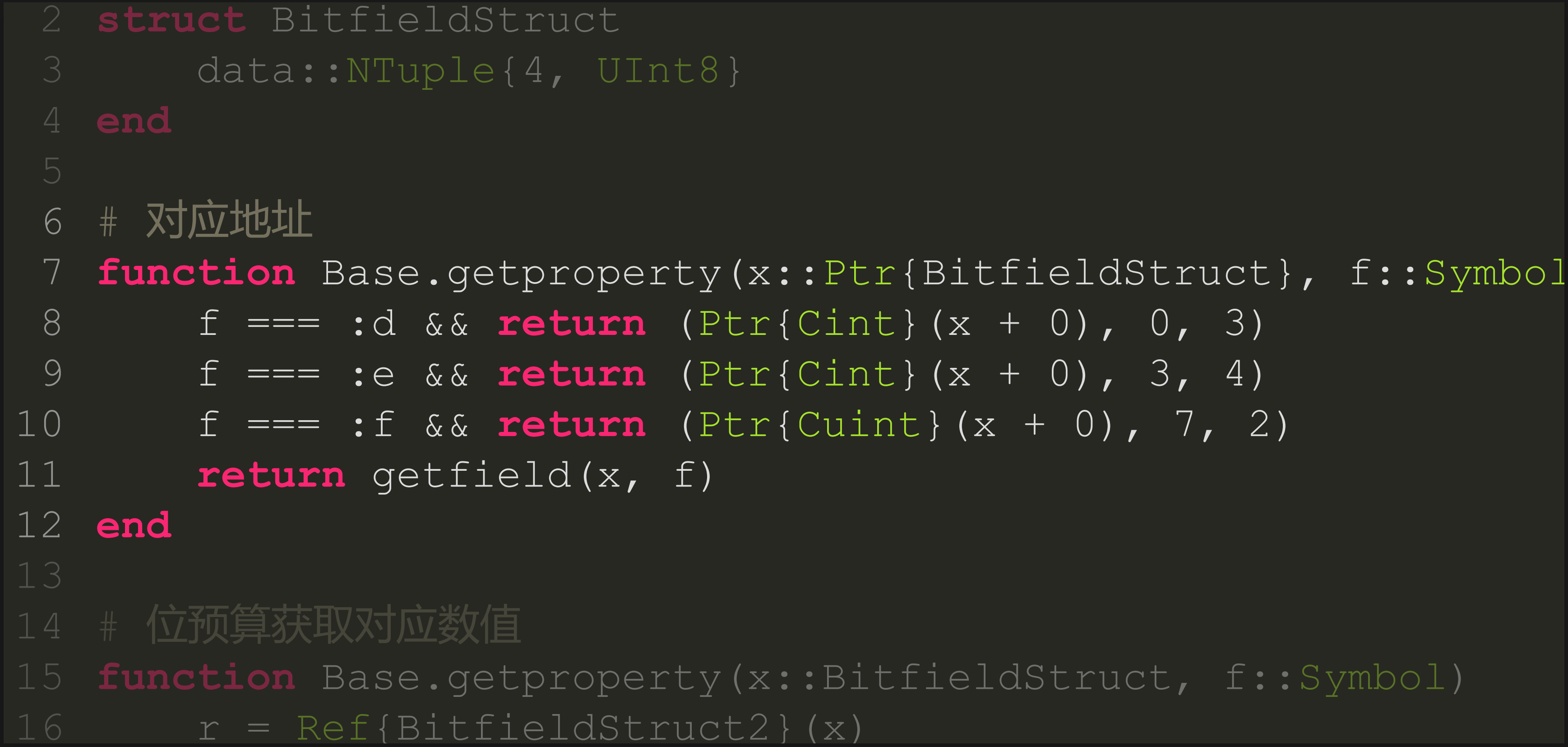
位域接口

```julia
# 4字节长
struct BitfieldStruct
    data::NTuple{4, UInt8}
end

# 对应地址
function Base.getproperty(x::Ptr{BitfieldStruct}, f::Symbol
    f === :d && return (Ptr{Cint}(x + 0), 0, 3)
    f === :e && return (Ptr{Cint}(x + 0), 3, 4)
    f === :f && return (Ptr{Cuint}(x + 0), 7, 2)
    return getfield(x, f)
end

# 位预算获取对应数值
function Base.getproperty(x::BitfieldStruct, f::Symbol)
```

位域接口

```julia
# 4字节长
struct BitfieldStruct
    data::NTuple{4, UInt8}
end

# 对应地址
function Base.getproperty(x::Ptr{BitfieldStruct}, f::Symbol
    f === :d && return (Ptr{Cint}(x + 0), 0, 3)
    f === :e && return (Ptr{Cint}(x + 0), 3, 4)
    f === :f && return (Ptr{Cuint}(x + 0), 7, 2)
    return getfield(x, f)
end

# 位预算获取对应数值
function Base.getproperty(x::BitfieldStruct, f::Symbol)
```

位域接口

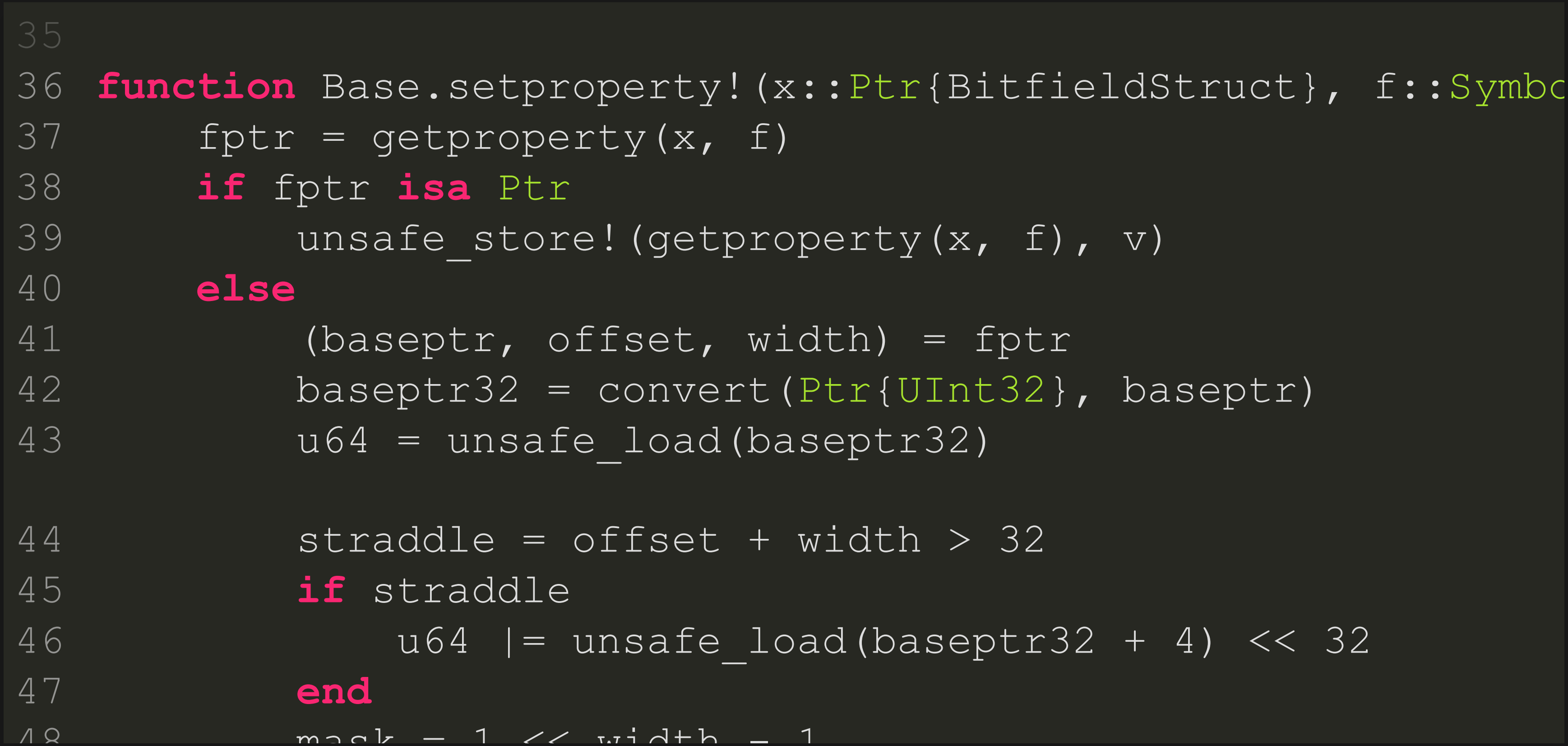


```julia
struct BitfieldStruct
    data::NTuple{4, UInt8}
end

# 对应地址
function Base.getproperty(x::Ptr{BitfieldStruct}, f::Symbol
    f === :d && return (Ptr{Cint}(x + 0), 0, 3)
    f === :e && return (Ptr{Cint}(x + 0), 3, 4)
    f === :f && return (Ptr{Cuint}(x + 0), 7, 2)
    return getfield(x, f)
end

# 位预算获取对应数值
function Base.getproperty(x::BitfieldStruct, f::Symbol)
    r = Ref{BitfieldStruct2}(x)
```

位域接口

```julia
# 位预算获取对应数值
function Base.getproperty(x::BitfieldStruct, f::Symbol)
    r = Ref{BitfieldStruct2}(x)
    ptr = Base.unsafe_convert(Ptr{BitfieldStruct2}, r)
    fptr = getproperty(ptr, f)

    begin
        if fptr isa Ptr
            return GC.@preserve(r, unsafe_load(fptr))
        else
            (baseptr, offset, width) = fptr
            ty = eltype(baseptr)
            baseptr32 = convert(Ptr{UInt32}, baseptr)
            u64 = GC.@preserve(r, unsafe_load(baseptr32))
            if offset + width > 32
```

位域接口



```julia

function Base.setproperty!(x::Ptr{BitfieldStruct}, f::Symbo
    fptr = getproperty(x, f)
    if fptr isa Ptr
        unsafe_store!(getproperty(x, f), v)
    else
        (baseptr, offset, width) = fptr
        baseptr32 = convert(Ptr{UInt32}, baseptr)
        u64 = unsafe_load(baseptr32)

        straddle = offset + width > 32
        if straddle
            u64 |= unsafe_load(baseptr32 + 4) << 32
        end
        mask = 1 << width - 1
```

提取注释中的文档

- C代码中的注释通常使用的doxygen格式可以转换为Julia注释中常用的markdown格式
- 将类型中的成员的注释整合为表格
- 寻找其他函数/类型的名字，并用超链接表示

函数文档示例

C

```
/**
 * \param Comment a \c CXComment_InlineCommand AST node.
 *
 * \param ArgIdx argument index (zero-based).
 *
 * \returns text of the specified argument.
 */
CINDEX_LINKAGE
CXString clang_InlineCommandComment_getArgText(CXComment Comme
                                               unsigned ArgIdx
```

JULIA

```julia
"""
    clang_BlockCommandComment_getArgText(Comment, ArgIdx)
### Parameters
* `Comment`: a `CXComment_BlockCommand` AST node.
* `ArgIdx`: argument index (zero-based).
### Returns
text of the specified word-like argument.
"""
function clang_BlockCommandComment_getArgText(Comment, ArgIdx)
    @ccall libclang.clang_BlockCommandComment_getArgText(Comme
end
```

REPL

```
help?> Clang.clang_BlockCommandComment_getArgText
  clang_BlockCommandComment_getArgText(Comment, ArgIdx)

  Parameters
  ------------

    •  Comment: a CXComment_BlockCommand AST node.

    •  ArgIdx: argument index (zero-based).

  Returns
  ---------

  text of the specified word-like argument.
```

DOCUMENTER

`Clang.LibClang.clang_BlockCommandComment_getArgText` — Method

```
clang_BlockCommandComment_getArgText(Comment, ArgIdx)
```

Parameters

- `Comment`: a `CXComment_BlockCommand` AST node.
- `ArgIdx`: argument index (zero-based).

Returns

text of the specified word-like argument.

类型文档示例

C

```c
/**
 * Describes the kind of error that occurred (if any) in a cal
 * \c clang_saveTranslationUnit().
 */
enum CXSaveError {
  /**
   * Indicates that no error occurred while saving a translati
   */
  CXSaveError_None = 0,

  /**
   * Indicates that an unknown error occurred while attempting
   * the file.
   *
   * This error typically indicates that file I/O failed when
```

JULIA

```julia
"""
    CXSaveError
Describes the kind of error that occurred (if any) in a call t
| Enumerator                      | Note
| :------------------------------ | :-------------------------
| CXSaveError\\_None              | Indicates that no error oc
| CXSaveError\\_Unknown           | Indicates that an unknown
| CXSaveError\\_TranslationErrors | Indicates that errors duri
| CXSaveError\\_InvalidTU         | Indicates that the transla
"""
@cenum CXSaveError::UInt32 begin
    CXSaveError_None = 0
    CXSaveError_Unknown = 1
    CXSaveError_TranslationErrors = 2
    CXSaveError_InvalidTU = 3
```

DOCUMENTER

`Clang.LibClang.CXSaveError` — Type

```
CXSaveError
```

Describes the kind of error that occurred (if any) in a call to `clang_saveTranslationUnit`().

Enumerator	Note
CXSaveError_None	Indicates that no error occurred while saving a translation unit.
CXSaveError_Unknown	Indicates that an unknown error occurred while attempting to save the file. This error typically indicates that file I/O failed when attempting to write the file.
CXSaveError_TranslationErrors	Indicates that errors during translation prevented this attempt to save the translation unit. Errors that prevent the translation unit from being saved can be extracted using `clang_getNumDiagnostics`() and `clang_getDiagnostic`().
CXSaveError_InvalidTU	Indicates that the translation unit to be saved was somehow invalid (e.g., NULL).

感谢观看！