Utilize Servidores de Linguagem para o Desenvolvimento na Árvore Src do FreeBSD

Índice

1. Introdução 1
2. Requisitos 1
3. Configurações do editor 2
4. Banco de dados de compilação 4
5. Finalizando 6

1. Introdução

Este guia é sobre como configurar uma árvore src do FreeBSD com servidores de linguagem realizando a indexação de código fonte. O guia descreve os passos para o Vim/NeoVim e o VSCode. Se você usar um editor de texto diferente, pode usar este guia como referência e procurar os comandos equivalentes para o seu editor preferido.

2. Requisitos

Para seguir este guia, precisamos instalar certos requisitos. Precisamos de um servidor de linguagem, `ccls` ou `clangd`, e opcionalmente um banco de dados de compilação.

A instalação do servidor de linguagem pode ser realizada via `pkg` ou via ports. Se escolhermos `clangd`, precisaremos instalar o `llvm`.

Usando o `pkg` para instalar o `ccls`:

```
# pkg install ccls
```

Se quisermos usar o `clangd`, precisaremos instalar o `llvm` (O comando de exemplo usa o `llvm15`, mas escolha a versão que desejar):

```
# pkg install llvm15
```

Para instalar via ports, escolha a sua combinação favorita de ferramentas de cada categoria abaixo:

- Implementações de servidores de linguagem

    - devel/ccls
    - devel/llvm12 (Outras versões também são aceitáveis, mas as mais novas são melhores. Substitua `clangd12` por `clangdN` no caso de outras versões serem usadas.)
- Editores
    - editors/vim
    - editors/neovim
    - editors/vscode
- Gerador de banco de dados de compilação
    - devel/python (Para a implementação do scan-build-py do llvm)
    - devel/py-pip (Para a implementação do scan-build do rizsotto)
    - devel/bear

3. Configurações do editor

3.1. Vim/Neovim

3.1.1. Plugins de cliente LSP

O gerenciador de plugin integrado é usado para ambos os editores neste exemplo. O plugin do cliente LSP usado é o prabirshrestha/vim-lsp.

Para configurar o plugin do cliente LSP para o Neovim:

```
# mkdir -p ~/.config/nvim/pack/lsp/start
# git clone https://github.com/prabirshrestha/vim-lsp
~/.config/nvim/pack/lsp/start/vim-lsp
```

e para o Vim:

```
# mkdir -p ~/.vim/pack/lsp/start
# git clone https://github.com/prabirshrestha/vim-lsp ~/.vim/pack/lsp/start/vim-lsp
```

Para habilitar o plugin do cliente LSP no editor, adicione o seguinte trecho em ~/.config/nvim/init.vim ao usar o Neovim, ou ~/.vim/vimrc ao usar o Vim:

Para o ccls

```
au User lsp_setup call lsp#register_server({
    \ 'name': 'ccls',
    \ 'cmd': {server_info->['ccls']},
    \ 'allowlist': ['c', 'cpp', 'objc'],
    \ 'initialization_options': {
```

```
    \     'cache': {
    \         'hierarchicalPath': v:true
    \     }
    \ }})
```

Para o clangd

```
au User lsp_setup call lsp#register_server({
    \ 'name': 'clangd',
    \ 'cmd': {server_info->['clangd15', '--background-index', '--header-
insertion=never']},
    \ 'allowlist': ['c', 'cpp', 'objc'],
    \ 'initialization_options': {},
    \ })
```

Dependendo da versão que você instalou para o `clangd`, pode ser necessário atualizar o `server-info` para apontar para o binário correto.

Por favor, consulte https://github.com/prabirshrestha/vim-lsp/blob/master/README.md#registering-servers para aprender sobre como configurar atalhos e as funções para autocompletar o código. O site oficial do clangd é https://clangd.llvm.org, e o repositório do ccls é https://github.com/MaskRay/ccls/.

Abaixo estão as configurações de referência para atalhos de teclado e o autocomplemento de código. Insira o seguinte trecho em ~/.config/nvim/init.vim ou ~/.vim/vimrc para que usuários do Vim possam utilizá-las:

```
function! s:on_lsp_buffer_enabled() abort
    setlocal omnifunc=lsp#complete
    setlocal completeopt-=preview
    setlocal keywordprg=:LspHover

    nmap <buffer> <C-]> <plug>(lsp-definition)
    nmap <buffer> <C-W>] <plug>(lsp-peek-definition)
    nmap <buffer> <C-W><C-]> <plug>(lsp-peek-definition)
    nmap <buffer> gr <plug>(lsp-references)
    nmap <buffer> <C-n> <plug>(lsp-next-reference)
    nmap <buffer> <C-p> <plug>(lsp-previous-reference)
    nmap <buffer> gI <plug>(lsp-implementation)
    nmap <buffer> go <plug>(lsp-document-symbol)
    nmap <buffer> gS <plug>(lsp-workspace-symbol)
    nmap <buffer> ga <plug>(lsp-code-action)
    nmap <buffer> gR <plug>(lsp-rename)
    nmap <buffer> gm <plug>(lsp-signature-help)
endfunction

augroup lsp_install
    au!
    autocmd User lsp_buffer_enabled call s:on_lsp_buffer_enabled()
```

```
augroup END
```

3.2. VSCode

3.2.1. Plugins de cliente LSP

Plugins de cliente LSP são necessários para iniciar o daemon do servidor de linguagem. Pressione `Ctrl+Shift+X` para exibir o painel de pesquisa de extensões online. Digite `llvm-vs-code-extensions.vscode-clangd` quando estiver usando o clangd, ou `ccls-project.ccls` quando estiver usando o ccls.

Em seguida, pressione `Ctrl+Shift+P` para exibir a paleta de comandos do editor. Digite `Preferences: Open Settings (JSON)` na paleta e pressione `Enter` para abrir o settings.json. Dependendo das implementações do servidor de linguagem, adicione um dos seguintes pares chave/valor JSON em settings.json:

Para o clangd

```
[
    /* Begin of your existing configurations */
    ...
    /* End of your existing configurations */
    "clangd.arguments": [
        "--background-index",
        "--header-insertion=never"
    ],
    "clangd.path": "clangd12"
]
```

Para o ccls

```
[
    /* Begin of your existing configurations */
    ...
    /* End of your existing configurations */
    "ccls.cache.hierarchicalPath": true
]
```

4. Banco de dados de compilação

Um banco de dados de compilação contém um array de objetos de comando de compilação. Cada objeto especifica uma maneira de compilar um arquivo de origem. O arquivo de banco de dados de compilação geralmente é chamado de compile_commands.json. O banco de dados é usado por implementações de servidores de linguagem para fins de indexação.

Por favor, consulte https://clang.llvm.org/docs/JSONCompilationDatabase.html#format para obter detalhes sobre o formato do arquivo do banco de dados de compilação.

4.1. Geradores

4.1.1. Usando scan-build-py

Instalação

A ferramenta `intercept-build` do scan-build-py é usada para gerar o banco de dados de compilação.

Primeiro instale o devel/python para obter o interpretador Python. Para obter o `intercept-build` do LLVM:

```
# git clone https://github.com/llvm/llvm-project /path/to/llvm-project
```

onde /path/to/llvm-project/ é o caminho desejado para o repositório. Crie um alias no arquivo de configuração do shell para conveniência:

```
alias intercept-build='/path/to/llvm-project/clang/tools/scan-build-py/bin/intercept-build'
```

Você também pode usar o rizsotto/scan-build em vez do scan-build-py do LLVM. O scan-build-py do LLVM foi incorporado ao repositório do LLVM. Essa implementação pode ser instalada com o comando `pip install --user scan-build`. O script `intercept-build` é instalado por padrão em ~/.local/bin.

Uso

No diretório raiz da árvore src do FreeBSD, gere o banco de dados de compilação com o `intercept-build`:

```
# intercept-build --append make buildworld buildkernel -j`sysctl -n hw.ncpu`
```

A opção `--append` instrui o `intercept-build` a ler um banco de dados de compilação existente (se existir) e adicionar os resultados ao banco de dados. As entradas com chaves de comando duplicadas são mescladas. O banco de dados de compilação gerado por padrão é salvo no diretório de trabalho atual como compile_commands.json.

4.1.2. Usando o devel/bear

Uso

No diretório raiz da árvore src do FreeBSD, para gerar o banco de dados de compilação com o `bear`:

```
# bear --append -- make buildworld buildkernel -j`sysctl -n hw.ncpu`
```

A opção `--append` instrui o `bear` a ler um banco de dados de compilação existente, se estiver

presente, e adicionar os resultados ao banco de dados. As entradas com chave de comando duplicada são mescladas. O banco de dados de compilação gerado por padrão é salvo no diretório de trabalho atual como compile_commands.json.

5. Finalizando

Depois que o banco de dados de compilação for gerado, abra qualquer arquivo de código fonte na árvore src do FreeBSD e o daemon do servidor LSP será lançado em segundo plano. Abrir arquivos de código fonte na árvore src pela primeira vez leva significativamente mais tempo antes que o servidor LSP seja capaz de fornecer um resultado completo, devido à indexação de segundo plano inicial pelo servidor LSP compilando todas as entradas listadas no banco de dados de compilação. No entanto, o daemon do servidor de linguagem não indexa os arquivos de origem que não aparecem no banco de dados de compilação, portanto, nenhum resultado completo é mostrado em arquivos de origem que não estão sendo compilados durante o `make`.