Uso de Servidores de Lenguajes para Desarrollo en el Árbol Src de FreeBSD

Tabla de contenidos

1. Introducción

Esta guía es acerca de cómo configurar un árbol src de FreeBSD con servidores de lenguajes para realizar indexado de código fuente.

2. Ports Requeridos

Se necesitan algunos ports a lo largo de esta guía. Escoge tu combinación favorita de herramientas de cada una de las categorías siguientes:

- Implementaciones de servidores de lenguajes
  - devel/ccls
  - devel/llvm12 (Otras versiones son válidas, pero cuanto más nuevo, mejor. Reemplaza `clangd12` con clangdN en caso de usar otra versión.)
- Editores
  - editors/vim
  - editors/neovim
  - editors/vscode
- Generador de base de datos de compilación
  - devel/python (Para la implementación de scan-build-py de llvm)
  - devel/py-pip (Para la implementación de scan-build de rizsotto)
  - devel/bear

3. Configuración del editor

3.1. Vim/Neovim

3.1.1. Plugins de LSP para el cliente

Este ejemplo utiliza el gestor de plugins incorporado en ambos editores. El plugin de LSP para el cliente es prabirshrestha/vim-lsp.

Para configurar el plugin cliente de LSP en Neovim:

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

y para Vim:

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

Para activar el plugin cliente de LSP en el editor, añade el siguiente fragmento en ~/.config/nvim/init.vim cuando uses Neovim, o ~/.vim/vimrc cuando uses Vim:

Para 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 clangd

```
au User lsp_setup call lsp#register_server({
    \ 'name': 'clangd',
    \ 'cmd': {server_info->['clangd12', '--background-index', '--header-
insertion=never']},
    \ 'allowlist': ['c', 'cpp', 'objc'],
    \ 'initialization_options': {},
    \ })
```

Por favor, dirígete a https://github.com/prabirshrestha/vim-lsp/blob/master/README.md#

registering-servers para aprender cómo configurar los atajos de teclado y el auto completado de código. El sitio oficial de clangd es https://clangd.llvm.org, y el enlace al repositorio de ccls es https://github.com/MaskRay/ccls/.

Abajo se muestra la configuración de referencia para los atajos de teclado y el auto completado de código. Pon el siguiente fragmento en ~/.config/nvim/init.vim, o ~/.vim/vimrc para usarlo con Vim:

```
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 LSP para el cliente

Los plugins de cliente de LSP son necesarios para arrancar un demonio de servidor de lenguajes. Presiona `Ctrl+Shift+X` para mostrar el panel de búsquedas online de extensiones. Teclea `llvm-vs-code-extensions.vscode-clangd` cuando utilices clangd, o `ccls-project.ccls` cuando utilices ccls.

Después, presiona `Ctrl+Shift+P` para mostrar la paleta del editor de comandos. Teclea `Preferences: Open Settings (JSON)` y presiona `Enter` para abrir settings.json. Dependiendo de la implementación del servidor de lenguajes, utiliza uno de los siguientes pares de clave/valor de JSON en settings.json:

Para clangd

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

Para ccls

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

4. Base de datos de compilación

Una base de datos de compilación contiene un array de objetos de comandos de compilación. Cada objeto especifica una forma de compilar un fichero fuente. El fichero de la base de datos de compilación es normalmente compiler_commands.json. La base de datos es utilizada por el servidor de lenguajes con propósitos de indexado.

Por favor consulta https://clang.llvm.org/docs/JSONCompilationDatabase.html#format para detalles acerca del formato del fichero de la base de datos de compilación.

4.1. Generadores

4.1.1. Utilizando scan-build-py

4.1.1.1. Instalación

La herramienta `intercept-build` de scan-build-py se utiliza para generar la base de datos de compilación.

Instala devel/python primero para tener el intérprete de python. Para obtener `intercept-build` de LLVM:

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

donde /path/to/llvm-project/ es la ruta que quieres en el repositorio. Crea un alias en el fichero de configuración del shell para tu comodidad:

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

Se puede usar rizsotto/scan-build en lugar del scan-build-py de LLVM. El scan-build-py de LLVM era rizsotto/scan-build que se integró en el árbol de LLVM.Se puede instalar esta implementación mediante `pip install --user scan-build`. El script `intercept-build` está en ~/.local/bin por defecto.

4.1.1.2. Uso

En el directorio de más alto nivel en el árbol src de FreeBSD, genera la base de datos de compilación con `intercept-build`:

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

El flag `--apend` le dice a `intercept-build` que lea una base de datos de compilación existente (si es que existe) y que añada los resultados a dicha base de datos. Las entradas con claves duplicadas de comandos son integradas. La base de datos de compilación generada por defecto se salva en el directorio de trabajo actual como compiler_commands.json.

4.1.2. Usando devel/bear

4.1.2.1. Uso

En el directorio de más alto nivel en el árbol src de FreeBSD, genera la base de datos de compilación con `bear`:

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

El flag `--apend` le dice a `bear` que lea una base de datos de compilación existente (si es que existe) y que añada los resultados a dicha base de datos. Las entradas con claves duplicadas de comandos son integradas. La base de datos de compilación generada por defecto se salva en el directorio de trabajo actual como compiler_commands.json.

5. Final

Una vez que la base de datos de compilación ha sido generada, abre cualquier fichero fuente del árbol src de FreeBSD y el demonio servidor de LSP será arrancado también en segundo plano. Abrir ficheros fuente en el árbol src la primera vez lleva significativamente más tiempo antes de que el servidor LSP sea capaz de completar el resultado, debido al indexado inicial en segundo plano que realiza el servidor LSP mediante la compilación de todas las entradas listadas en la base de datos de compilación. Sin embargo el demonio de servicio de lenguajes no indexa ficheros fuente que no aparecen en la base de datos de compilación, por lo tanto no se muestran resultados completos

para ficheros fuentes que no se compilan durante la ejecución de `make`.