

# 如何配置Emacs的C/C++语言LSP

---

[索引的问题](#)

[最终效果 \(LSP\)](#)

[准备工作](#)

[编译cc1s](#)

[生成编译信息](#)

[安装插件](#)

## 索引的问题

内核代码使用自己的头文件和一套复杂的编译脚本，用ctags/gtags等基于正则方式的工具产生的索引，在使用中跟lsp方式相比有很多不足：

- 没法直接在变量/成员上查看类型定义
- 在变量/成员上查看引用时，所有同名的（不同结构体中的、甚至不同作用域中的）都会显示（比如找一下某个lock域的引用），有时候几乎不具备参考价值
- 代码被各种复杂的宏定义包住，经常要花很大力气才知道哪些代码在起作用（比如preempt\_enable的实现）
- 代码有多个arch的版本，每次查看定义或者引用都需要反复选择或者过滤
- 很多看起来像函数的代码其实是宏，如果当成函数会导致后面实际调试中找不到符号
- 没法方便的查看caller/callee hierarchy
- 省略（参考lsp能实现的功能，比如变量更名之类的）

以上是C语言，如果是C++代码那tags的问题就更多了

## 最终效果 (LSP)

- 直接查看变量或者结构体成员引用/定义（快捷键：M-?/M-.）

```

579
580 static inline struct kvm_memslots *__kvm_memslots(struct kvm *kvm, int as_id)
581 {
582     as_id = array_index_nospec(as_id, KVM_ADDRESS_SPACE_NUM);
583     return srcu_dereference_check(kvm->memslots[as_id], &kvm->srcu,
584 /include/linux/kvm_host.h
    struct module *module);
void kvm_exit(void);

void kvm_get_kvm(struct kvm *kvm);
void kvm_put_kvm(struct kvm *kvm);

static inline struct kvm_memslots *__kvm_memslots(struct kvm *kvm, int as_id)
{
    as_id = array_index_nospec(as_id, KVM_ADDRESS_SPACE_NUM);
    return srcu_dereference_check(kvm->memslots[as_id], &kvm->srcu,
        lockdep_is_held(&kvm->slots_lock) ||
        !refcount_read(&kvm->users_count));
}

static inline struct kvm_memslots *kvm_memslots(struct kvm *kvm)
{
    return __kvm_memslots(kvm, 0);
}

lockdep_is_held(&kvm->slots_lock) ||
!refcount_read(&kvm->users_count));
585 }
586 }

```

3 references	
/include/linux/kvm_host.h	1
583 return srcu_dereference_check(kvm->memslot..	2
/virt/kvm/kvm_main.c	
708 rcu_assign_pointer(kvm->memslots[i], slots);	
935 rcu_assign_pointer(kvm->memslots[as_id], s..	

- 查看定义时，自动跳转至实际起作用的代码段，不起作用的代码灰色显示（preempt\_disable函数）

```

230
231 #define preempt_enable_no_resched_notrace() \
232 do { \
233     barrier(); \
234     __preempt_count_dec(); \
235 } while (0)
236
237 #else /* !CONFIG_PREEMPT_COUNT */
238
239 /*
240  * Even if we don't have any preemption, we need preempt disable/enable
241  * to be barriers, so that we don't have things like get_user/put_user
242  * that can cause faults and scheduling migrate into our preempt-protected
243  * region.
244  */
245 #define preempt_disable() barrier()
246 #define sched_preempt_enable_no_resched() barrier()
247 #define preempt_enable_no_resched() barrier()
248 #define preempt_enable() barrier()
249 #define preempt_check_resched() do { } while (0)
250
251 #define preempt_disable_notrace() barrier()
252 #define preempt_enable_no_resched_notrace() barrier()
253 #define preempt_enable_notrace() barrier()
254 #define preemptible() 0
255
256 #endif /* CONFIG_PREEMPT_COUNT */
257
258 #ifdef MODULE
259 /*
260  * Modules have no business playing preemption tricks.
261  */
262 #undef sched_preempt_enable_no_resched
263 #undef preempt_enable_no_resched
264 #undef preempt_enable_no_resched_notrace
265 #undef preempt_check_resched
266 #endif

```

未编译代码

- 光标停留在符号上，直接显示符号信息，比如这里atomic\_long\_xchg能看到是个宏

```

72
73 tail = (struct spsc_node *)atomic_long_xchg(&queue->tail, (long)&node->next);
74 WRITE_ONCE(*tail, node);
75 atomic_inc(&queue->job_count);
76
77 /*
78  * In case of first element verify new node will be visible to the consumer
79  * thread when we ping the kernel thread that there is new work to do.
80  */
81 smp_wmb();
82
83 preempt_enable();
84
85 return tail == &queue->head;
86 }
87
88 .1k alikernel-4.19/include/drm/spsc_queue.h 73:42 37%
#define atomic_long_xchg(v, new) \
    (ATOMIC_LONG_PFX(_xchg)((ATOMIC_LONG_PFX(_t) *) (v), (new)))

```

- 查看被调用栈 (caller hierarchy) (快捷键: C-c s l 然后选择c)

```
1 Callers of
2 atomic64_xchg
3 |> spsc_queue_push (spsc_queue.h:64)
4 |> calc_load_nohz_fold (loadavg.c:303)
5 |> calc_global_load (loadavg.c:393)
6 |> do_timer (timekeeping.c:2202)
7 |> tick_periodic (tick-common.c:78)
8 |> tick_handle_periodic (tick-common.c:102)
9 |> tick_device_uses_broadcast (tick-broadcast.c:161)
10 |> tick_set_periodic_handler (tick-broadcast.c:501)
11 |> tick_handle_periodic (tick-common.c:102)
12 |> tick_do_update_jiffies64 (tick-sched.c:56)
13 |> xtime_update (timekeeping.c:2399)
14 |> calc_load_nohz_r_fold (loadavg.c:315)
15 |> calc_global_load (loadavg.c:393)
16 |> propagate_protected_usage (page_counter.c:15)
17 |> page_counter_cancel (page_counter.c:54)
18 |> page_counter_charge (page_counter.c:71)
19 |> page_counter_try_charge (page_counter.c:98)
20 |> page_counter_set_min (page_counter.c:211)
21 |> page_counter_set_low (page_counter.c:228)
22 |> do_shrink_slab (vmscan.c:510)
23 |> shrink_slab_memcg (vmscan.c:632)
24 |> shrink_slab (vmscan.c:734)
```

可展开调用栈

- 查看调用链 (callee hierarchy) (快捷键: C-u C-c s l 然后选择c)

这里能看到\_\_kvm\_memslots调用的array\_index\_nospec其实是一个宏，直接展开成里面包住的函数了

```
1 Callee of
2 kvm_get_dirty_log
3 |> kvm_memslots (kvm_main.c:1164)
4 |> array_index_mask_nospec (kvm_host.h:581)
5 |> __read_once_size (kvm_host.h:582)
6 |> id_to_memslot (kvm_main.c:1165)
7 |> printk (kvm_host.h:607)
8 |> kvm_dirty_bitmap_bytes (kvm_main.c:1169)
9 |> copy_to_user (kvm_main.c:1174)
10 |> check_copy_size (uaccess.h:153)
11 |> _copy_to_user (uaccess.h:154)
```

```
static inline struct kvm_memslots * __kvm_memslots(struct kvm *kvm, int as_id)
{
    as_id = array_index_mask_nospec(as_id, KVM_ADDRESS_SPACE_NUM);
    return srcu_dereference_check(kvm->memslots[as_id], &kvm->srcu,
        lockdep_is_held(&kvm->slots_lock) ||
        !refcount_read(&kvm->users_count));
}
```

```
48 */
49 #define array_index_nospec(index, size) \
50 { \
51     typeof(index) _i = (index); \
52     typeof(size) _s = (size); \
53     unsigned long _mask = array_index_mask_nospec(_i, _s); \
54 \
55     BUILD_BUG_ON(sizeof(_i) > sizeof(long)); \
56     BUILD_BUG_ON(sizeof(_s) > sizeof(long)); \
57 \
58     (typeof(_i)) (_i & _mask); \
59 } \
60
```

- 更多的有待使用中探索

## 准备工作

- Emacs 27 或者更高版本 (建议打开jansson支持, 加速lsp协议传输)
- 升级gcc到9.2.1, 不然ccls没法编译
- 安装llvm和clang

sudo yum -b current install llvm11 clang11 clang11-devel llvm11-devel llvm11-static

- 安装高版本cmake (> 3.8.0)

<https://github.com/Kitware/CMake/releases>

## 编译ccls

- 参考: <https://github.com/MaskRay/ccls/wiki/Build>

```
▼ Bash | 复制代码

1  git clone --depth=1 --recursive https://github.com/MaskRay/ccls
2  cd ccls
3  cmake -H. -BRelease -DCMAKE_BUILD_TYPE=Release
4  cmake --build Release
5
6  # 把 Release/ccls 放到 PATH 中
```

## 生成编译信息

- 参考: <https://github.com/MaskRay/ccls/wiki/Example-Projects>

```
▼ Bash | 复制代码

1  # 进入内核代码目录
2  mkdir out
3  cp configs/xxxxxx.config ./out/.config
4
5  make O=./out CC=/usr/bin/clang olddefconfig
6  make O=./out CC=/usr/bin/clang bzImage # generate .<target>.cmd files
7
8  # 新的内核直接用LLVM=1编译
9  # make O=./out LLVM=1 bzImage
10
11 # modules 建议只编译需要看的
12 make O=./out CC=/usr/bin/clang SUBDIRS=arch/x86/kvm
13
14 # 如果有objtool报错: Segmentation fault
15 # 在对应的目录Makefile里加入:
16 # OBJECT_FILES_NON_STANDARD := y
17 # 或者
18 # OBJECT_FILES_NON_STANDARD_foo.o := y
19
20 # 这是 upstream kernel 里的 scripts/clang-tools 中的脚本
21 gen_compile_commands.py -d ./out # generate compile_commands.json
```

## 安装插件

参考: <https://github.com/MaskRay/ccls/wiki/lsp-mode>

直接git pull更新.emacs.d配置即可