libco源码阅读记录

A、主要数据结构

- stCoRoutineEnv_t：

```
//协程环境，会被定义为线程私有变量
struct stCoRoutineEnv_t
{
    //协程数组
    stCoRoutine_t *pCallStack[ 128 ];
    //协程数量
    int iCallStackSize;
    //协程epoll对象
    stCoEpoll_t *pEpoll;

    //for copy stack log lastco and nextco
    //即将执行的协程以及当前协程
    stCoRoutine_t* pending_co;
    stCoRoutine_t* occupy_co;
};
```

- stCoEpoll_t：

```
struct stCoEpoll_t
{
    //epoll fd
    int iEpollFd;
    static const int _EPOLL_SIZE = 1024 * 10;
    //定时列表
    struct stTimeout_t *pTimeout;
    //超时列表
    struct stTimeoutItemLink_t *pstTimeoutList;
    //触发列表
    struct stTimeoutItemLink_t *pstActiveList;
    //epoll_event
    co_epoll_res *result;

};
```

- stTimeout_t：

```
struct stTimeout_t
{
    //stTimeoutItemLink_t数组
    stTimeoutItemLink_t *pItems;
    //数组大小
    int iItemSize;

    //起始时间（毫秒级）
    unsigned long long ullStart;
    //起始时间所对应的数组下标
    long long llStartIdx;
};
```

- stCoRoutine_t：

```
struct stCoRoutine_t
{
    stCoRoutineEnv_t *env;
    //函数指针
    pfn_co_routine_t pfn;
    //函数参数
    void *arg;
    //协程栈信息
    coctx_t ctx;

    //是否已开始
    char cStart;
    char cEnd;
    //是否是主协程
    char cIsMain;
    //本协程是否使用hook的系统函数
    char cEnableSysHook;
    //是否使用共享栈
    char cIsShareStack;

    void *pvEnv;

    //char sRunStack[ 1024 * 128 ];
    //自定义栈内存信息指针
    stStackMem_t* stack_mem;

    //save satck buffer while confilct on same stack_buffer;
    //共享栈时所需要的成员变量
    //栈顶
    char* stack_sp;
    //本协程使用的栈大小
    unsigned int save_size;
    //每次切换协程时，需要重新申请内存来保存本协程栈数据
    char* save_buffer;

    stCoSpec_t aSpec[1024];
};
```

- coctx_t:

```
//协程栈信息
struct coctx_t
{
//寄存器指针数组
#if defined(__i386__)
    void *regs[ 8 ];
#else
    void *regs[ 14 ];
#endif
//栈大小
    size_t ss_size;
//栈顶指针
    char *ss_sp;

};
```

x86架构下32位有8个通用寄存器，64位有16个通用寄存器，rax、rbx、rcx、rdx、rsi、rdi、rbp、rsp、r8-r15。这16个寄存器除了rbp（保存栈底）、rsp（保存栈顶），其余的除了特定的作用外还可以用来保存临时数据、中间结果或用于计算。 这16个寄存器又分为callee-saved（被调用者保存）和caller-saved（调用者保存）。如下图所示：

Summary Table of Caller-Saved vs Callee-Saved Registers (System V ABI for Linux)

Register	Caller-Saved	Callee-Saved
`rax`	✔ (caller-saved)	
`rcx`	✔ (caller-saved)	
`rdx`	✔ (caller-saved)	
`rsi`	✔ (caller-saved)	
`rdi`	✔ (caller-saved)	
`r8`	✔ (caller-saved)	
`r9`	✔ (caller-saved)	
`r10`	✔ (caller-saved)	
`r11`	✔ (caller-saved)	
`rbx`		✔ (callee-saved)
`rbp`		✔ (callee-saved)
`r12`		✔ (callee-saved)
`r13`		✔ (callee-saved)
`r14`		✔ (callee-saved)
`r15`		✔ (callee-saved)

在属于caller-saved的寄存器中，只有r10、r11没有特定的作用，这应该就是libco不保存这两个寄存器的

原因。r8、r9可能会用于参数传递。如下图所示（参数再多则通过入栈来传递）：

```
00000000000013fe <main>:
    13fe:   f3 0f 1e fa             endbr64
    1402:   55                      push   %rbp
    1403:   48 89 e5                mov    %rsp,%rbp
    1406:   6a 08                   pushq  $0x8          将第八个参数（8）压栈
    1408:   6a 07                   pushq  $0x7          将第七个参数（7）压栈
    140a:   41 b9 06 00 00 00       mov    $0x6,%r9d     将第六个参数（6）复制到r9寄存器
    1410:   41 b8 05 00 00 00       mov    $0x5,%r8d     将第五个参数（5）复制到r8寄存器
    1416:   b9 04 00 00 00          mov    $0x4,%ecx     将第四个参数（4）复制到rcx寄存器
    141b:   ba 03 00 00 00          mov    $0x3,%edx     将第三个参数（3）复制到rdx寄存器
    1420:   be 02 00 00 00          mov    $0x2,%esi     将第二个参数（2）复制到rsi寄存器
    1425:   bf 01 00 00 00          mov    $0x1,%edi     将第一个参数（1）复制到寄存器rdi
    142a:   e8 08 ff ff ff          callq  1337 <_Z4fun8llllllll>
    142f:   48 83 c4 10             add    $0x10,%rsp
    1433:   b8 00 00 00 00          mov    $0x0,%eax
    1438:   c9                      leaveq
    1439:   c3                      retq
```


B、部分API，其他感觉没啥可写的，正常逻辑没啥特殊点

- co_create:

```
int co_create( stCoRoutine_t **ppco,const stCoRoutineAttr_t *attr,pfn_co_routine_t pfn,void *arg )
{
    if( !co_get_curr_thread_env() )
    {
        co_init_curr_thread_env();
    }
    stCoRoutine_t *co = co_create_env( co_get_curr_thread_env(), attr, pfn,arg );
    *ppco = co;
    return 0;
}
```

- co_init_curr_thread_env:

```
static __thread stCoRoutineEnv_t* gCoEnvPerThread = NULL;

void co_init_curr_thread_env()
{
    gCoEnvPerThread = (stCoRoutineEnv_t*)calloc( 1, sizeof(stCoRoutineEnv_t) );
    stCoRoutineEnv_t *env = gCoEnvPerThread;

    env->iCallStackSize = 0;
    //创建主协程
    struct stCoRoutine_t *self = co_create_env( env, NULL, NULL,NULL );
    self->cIsMain = 1;

    env->pending_co = NULL;
    env->occupy_co = NULL;

    coctx_init( &self->ctx );

    env->pCallStack[ env->iCallStackSize++ ] = self;

    stCoEpoll_t *ev = AllocEpoll();
    SetEpoll( env,ev );
}
```

- co_create_env：

```
struct stCoRoutine_t *co_create_env( stCoRoutineEnv_t * env, const
stCoRoutineAttr_t* attr,
    pfn_co_routine_t pfn,void *arg )
{

  stCoRoutineAttr_t at;
  if( attr )
  {
    memcpy( &at,attr,sizeof(at) );
  }
  if( at.stack_size <= 0 )
  {
    //默认栈大小128KB
    at.stack_size = 128 * 1024;
  }
  else if( at.stack_size > 1024 * 1024 * 8 )
  {
    //最大栈8M
    at.stack_size = 1024 * 1024 * 8;
  }

  //4K对齐，glibc默认的页大小，避免页缺失，优化内存访问
  if( at.stack_size & 0xFFF )
  {
    at.stack_size &= ~0xFFF;
    at.stack_size += 0x1000;
  }

  stCoRoutine_t *lp = (stCoRoutine_t*)malloc( sizeof(stCoRoutine_t) );

  memset( lp,0,(long)(sizeof(stCoRoutine_t)));

  lp->env = env;
  lp->pfn = pfn;
  lp->arg = arg;

  stStackMem_t* stack_mem = NULL;
  if( at.share_stack )
  {
    //共享栈
    stack_mem = co_get_stackmem( at.share_stack);
    at.stack_size = at.share_stack->stack_size;
  }
  else
  {
    stack_mem = co_alloc_stackmem(at.stack_size);
  }
  lp->stack_mem = stack_mem;

  lp->ctx.ss_sp = stack_mem->stack_buffer;
  lp->ctx.ss_size = at.stack_size;

  lp->cStart = 0;
```

```
    lp->cEnd = 0;
    lp->cIsMain = 0;
    lp->cEnableSysHook = 0;
    lp->cIsShareStack = at.share_stack != NULL;

    lp->save_size = 0;
    lp->save_buffer = NULL;

    return lp;
}
```

默认栈大小为128K，大抵是因为glibc初始的brk\mmap阈值是128K（可能会被动态调整，调用mallopt设置阈值或者其他的一些东西的时候会把动态调整关闭，32位的最大阈值是512K，64位最大是32M，ptmalloc里再做记录），这能避免内存碎片。但如果是共享栈的时候，每次做协程切换的时候，原来协程的栈的内容每次都会申请内存来保存，这个大小又没按这个阈值来，有点奇怪。save_stack_buffer代码截图如下所示：

```
void save_stack_buffer(stCoRoutine_t* occupy_co)
{
    ///copy out
    stStackMem_t* stack_mem = occupy_co->stack_mem;
    int len = stack_mem->stack_bp - occupy_co->stack_sp;

    if (occupy_co->save_buffer)
    {
        free(occupy_co->save_buffer), occupy_co->save_buffer = NULL;
    }

    occupy_co->save_buffer = (char*)malloc(len); //malloc buf;
    occupy_co->save_size = len;

    memcpy(occupy_co->save_buffer, occupy_co->stack_sp, len);
}
```

- co_resume：

```
void co_resume( stCoRoutine_t *co )
{
    stCoRoutineEnv_t *env = co->env;
    stCoRoutine_t *lpCurrRoutine = env->pCallStack[ env->iCallStackSize - 1 ];
    if( !co->cStart )
    {
        //设置协程的
        coctx_make( &co->ctx,(coctx_pfn_t)CoRoutineFunc,co,0 );
        co->cStart = 1;
    }
    env->pCallStack[ env->iCallStackSize++ ] = co;
    co_swap( lpCurrRoutine, co );
}
```

- coctx_make:

```
int coctx_make(coctx_t* ctx, coctx_pfn_t pfn, const void* s, const void* s1) {
  char* sp = ctx->ss_sp + ctx->ss_size - sizeof(void*);
  //16个字节对齐,gcc现在默认堆栈对齐到16个字节，这是因为一些SSE指令如果没有做16个字节对齐会发生段错误。
  sp = (char*)((unsigned long)sp & -16LL);

  memset(ctx->regs, 0, sizeof(ctx->regs));
  void** ret_addr = (void**)(sp);
  *ret_addr = (void*)pfn;

  ctx->regs[kRSP] = sp;

  ctx->regs[kRETAddr] = (char*)pfn;

  //第一个参数寄存器rdi
  ctx->regs[kRDI] = (char*)s;
  //第二个参数寄存器rsi
  ctx->regs[kRSI] = (char*)s1;
  return 0;
}
```

之所以要做16个字节对齐，是因为gcc现在默认堆栈对齐到16个字节，一些SSE指令如果没有做16个字节

对齐会发生段错误。资料地址：https://sourceforge.net/p/fbc/bugs/659/，测试结果如下图所示：

```
(gdb) l
1       #include <stdio.h>
2
3       void fun()
4       {
5               char arr[8];
6       }
7
8       void fun2()
9       {
10              char arr[24];
(gdb)
11      }
12
13      int main()
14      {
15              return 0;
16      }
(gdb) disassemble fun
Dump of assembler code for function fun:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   %rbp
   0x000000000000114e <+5>:     mov    %rsp,%rbp
   0x0000000000001151 <+8>:     sub    $0x10,%rsp    栈顶向下移16个字节
   0x0000000000001155 <+12>:    mov    %fs:0x28,%rax
   0x000000000000115e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000000000001162 <+25>:    xor    %eax,%eax
   0x0000000000001164 <+27>:    nop
   0x0000000000001165 <+28>:    mov    -0x8(%rbp),%rax
   0x0000000000001169 <+32>:    xor    %fs:0x28,%rax
   0x0000000000001172 <+41>:    je     0x1179 <fun+48>
   0x0000000000001174 <+43>:    callq  0x1050 <__stack_chk_fail@plt>
   0x0000000000001179 <+48>:    leaveq
   0x000000000000117a <+49>:    retq
End of assembler dump.
(gdb) disassemble fun2
Dump of assembler code for function fun2:
   0x000000000000117b <+0>:     endbr64
   0x000000000000117f <+4>:     push   %rbp
   0x0000000000001180 <+5>:     mov    %rsp,%rbp
   0x0000000000001183 <+8>:     sub    $0x20,%rsp    栈顶向下移32个字节
   0x0000000000001187 <+12>:    mov    %fs:0x28,%rax
   0x0000000000001190 <+21>:    mov    %rax,-0x8(%rbp)
   0x0000000000001194 <+25>:    xor    %eax,%eax
   0x0000000000001196 <+27>:    nop
   0x0000000000001197 <+28>:    mov    -0x8(%rbp),%rax
   0x000000000000119b <+32>:    xor    %fs:0x28,%rax
   0x00000000000011a4 <+41>:    je     0x11ab <fun2+48>
   0x00000000000011a6 <+43>:    callq  0x1050 <__stack_chk_fail@plt>
   0x00000000000011ab <+48>:    leaveq
   0x00000000000011ac <+49>:    retq
End of assembler dump.
```

- coctx_swap:

```
//将rsp寄存器指向的地址赋值到rax
leaq (%rsp),%rax
//将寄存器的值赋值到相对于第一个参数偏移对应字节的地方
movq %rax, 104(%rdi)
movq %rbx, 96(%rdi)
movq %rcx, 88(%rdi)
movq %rdx, 80(%rdi)
```

```
    //将rax(也就是rsp)指向地址的值赋值到rax，这个地址的此时值是coctx_swap调用者的
    //下一条指令地址，这是因为调用者在调用coctx_swap函数时会将下一条指令压栈
    movq 0(%rax), %rax
    //保存rax
    movq %rax, 72(%rdi)
    //将寄存器的值赋值到相对于第一个参数偏移对应字节的地方
    movq %rsi, 64(%rdi)
    movq %rdi, 56(%rdi)
    movq %rbp, 48(%rdi)
    movq %r8, 40(%rdi)
    movq %r9, 32(%rdi)
    movq %r12, 24(%rdi)
    movq %r13, 16(%rdi)
    movq %r14, 8(%rdi)
    movq %r15, (%rdi)
    //将rax寄存器的值清空，对自己异或
    xorq %rax, %rax

    //将保存在第二个参数的内容赋值到寄存器
    movq 48(%rsi), %rbp
    movq 104(%rsi), %rsp
    movq (%rsi), %r15
    movq 8(%rsi), %r14
    movq 16(%rsi), %r13
    movq 24(%rsi), %r12
    movq 32(%rsi), %r9
    movq 40(%rsi), %r8
    movq 56(%rsi), %rdi
    movq 80(%rsi), %rdx
    movq 88(%rsi), %rcx
    movq 96(%rsi), %rbx
    //将栈顶往上挪8个字节，为了下面将返回地址压栈
    leaq 8(%rsp), %rsp
    //将返回地址压栈
    pushq 72(%rsi)
    movq 64(%rsi), %rsi
    //弹出栈顶的值，也就是返回地址并跳转
    ret
```

对上面的注释的自测结果如下图所示：

调用coctx_swap函数前的寄存器信息：

```
76          coctx_swap(mainInfo, funInfo);
(gdb) i r
rax            0x555555555179      93824992235897
rbx            0x555555555350      93824992236368
rcx            0x7ffff7cfdb06      140737350982406
rdx            0x1                 1
rsi            0x21000             135168
rdi            0x0                 0
rbp            0x7fffffffe190      0x7fffffffe190
rsp            0x7fffffffe190      0x7fffffffe190
r8             0x7ffff7a56010      140737348198416
r9             0x0                 0
r10            0x22                34
r11            0x246               582
r12            0x555555555090      93824992235664
r13            0x7fffffffe280      140737488347776
r14            0x0                 0
r15            0x0                 0
rip            0x555555555238      0x555555555238 <main()+25>
eflags         0x206               [ PF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

调用coctx_swap函数后的寄存器信息：

```
(gdb)
coctx_swap () at coctx_swap.S:30
30              leaq (%rsp),%rax
(gdb) i r
rax            0x555555555179      93824992235897
rbx            0x555555555350      93824992236368
rcx            0x7ffff7cfdb06      140737350982406
rdx            0x1                 1
rsi            0x5555555580c0      93824992248000
rdi            0x555555558040      93824992247872
rbp            0x7fffffffe190      0x7fffffffe190
rsp            0x7fffffffe188      0x7fffffffe188
r8             0x7ffff7a56010      140737348198416
r9             0x0                 0
r10            0x22                34
r11            0x246               582
r12            0x555555555090      93824992235664
r13            0x7fffffffe280      140737488347776
r14            0x0                 0
r15            0x0                 0
rip            0x5555555552c4      0x5555555552c4 <coctx_swap>
eflags         0x206               [ PF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) x /2xg 0x7fffffffe188     rsp寄存器的值，此时是指向调用coctx_swap后的下一条指令地址
0x7fffffffe188: 0x000055555555524b      0x0000000000000000
(gdb) disassemble 0x000055555555524b
Dump of assembler code for function main():
   0x000055555555521f <+0>:     endbr64
   0x0000555555555223 <+4>:     push   %rbp
   0x0000555555555224 <+5>:     mov    %rsp,%rbp
   0x0000555555555227 <+8>:     lea    0xde2(%rip),%rdi        # 0x555555556010
   0x000055555555522e <+15>:    callq  0x555555555070 <puts@plt>
   0x0000555555555233 <+20>:    callq  0x5555555551a3 <init()>
   0x0000555555555238 <+25>:    lea    0x2e81(%rip),%rsi        # 0x5555555580c0 <funInfo>
   0x000055555555523f <+32>:    lea    0x2dfa(%rip),%rdi        # 0x555555558040 <mainInfo>
   0x0000555555555246 <+39>:    callq  0x5555555552c4 <coctx_swap>
   0x000055555555524b <+44>:    lea    0xdc8(%rip),%rdi        # 0x55555555601a
   0x0000555555555252 <+51>:    callq  0x555555555070 <puts@plt>
   0x0000555555555257 <+56>:    mov    $0x0,%eax
   0x000055555555525c <+61>:    pop    %rbp
   0x000055555555525d <+62>:    retq
End of assembler dump.
```