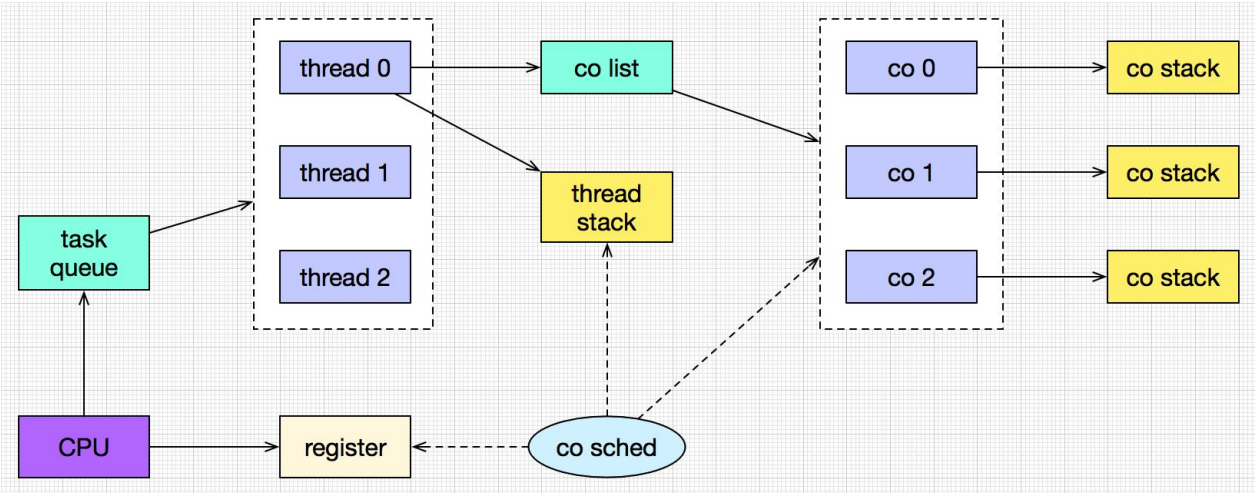
作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

协程的含义

协程表示一个独立的功能逻辑，每个协程有独立的协程上下文，协程调度切换协程上下文；多个协程共用 1 个线程，单个线程某个时刻只有 1 个协程在运行。
协程的核心是切换协程上下文，包括切换寄存器、切换协程栈。

协程的实现有多种方式，且比较复杂。这里使用单线程和独立协程栈的方式。
协程调度器，管理一组协程，切换协程上下文，运行某个协程。
协程使用独立栈，每个协程一个栈。
主协程，复用线程栈，管理协程调度。
子协程，处理自定义功能。

协程的示意图：



用 C 和汇编实现完整的协程功能

编写代码： co_handler.h

```
#ifndef _CO_HANDLER_H_
#define _CO_HANDLER_H_

#define CO_STACK_SIZE 4096 // 协程栈的大小
#define CO_LIST_SIZE 9     // 协程队列的大小
#define CO_STATUS_OK 1     // 协程的状态。正常。
#define CO_STATUS_EXIT 2   // 协程的状态。退出。

// 协程的对象
struct co
{
    uint64_t rsp;    // +0  寄存器，rsp
```

```
    uint64_t rbp;    // +8  寄存器，rbp
    uint64_t rax;    // +16 寄存器，rax
    uint64_t rbx;    // +24 寄存器，rbx
    uint64_t rcx;    // +32 寄存器，rcx
    uint64_t rdx;    // +40 寄存器，rdx
    char *stack_low; // 栈的下界地址
    char name[30];   // 名称
    uint32_t status; // 状态
};

typedef void (*co_func)(); // 协程的函数

struct co *co_main;    // 主协程。负责调度子协程
struct co *co_running; // 正在运行的子协程

// 限于篇幅，队列使用数组实现。使用链表更合适。
struct co *co_list[CO_LIST_SIZE]; // 协程队列
uint32_t co_list_cursor;          // 轮询数组的下标。

// 创建协程
extern struct co *co_create(char *name, co_func func);

// 运行某个子协程。主协程调用此方法。
extern void co_resume(struct co *co_sub);

// 当前协程切换出去。子协程调用此方法。
extern void co_yield ();

// 运行协程管理器
extern void co_handler_run();

#endif
```

编写代码： co_switch_context.s

```
.text
.global co_switch_context

# 切换协程的上下文
co_switch_context :

    # 备份上下文到 from_co
    movq %rsp, 0(%rdi)  # 寄存器，rsp
    movq %rbp, 8(%rdi)  # 寄存器，rbp
    movq %rax, 16(%rdi) # 寄存器，rax
    movq %rbx, 24(%rdi) # 寄存器，rbx
    movq %rcx, 32(%rdi) # 寄存器，rcx
    movq %rdx, 40(%rdi) # 寄存器，rdx

    # 从 to_co 恢复上下文
```

```
    movq 0(%rsi), %rsp  # 寄存器，rsp
    movq 8(%rsi), %rbp  # 寄存器，rbp
    movq 16(%rsi), %rax # 寄存器，rax
    movq 24(%rsi), %rbx # 寄存器，rbx
    movq 32(%rsi), %rcx # 寄存器，rcx
    movq 40(%rsi), %rdx # 寄存器，rdx

    retq
```

编写代码： co_handler.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include "co_handler.h"

// 协程切换上下文。汇编代码
extern void co_switch_context(struct co *from_co, struct co *to_co);

// 协程，执行完了，清理自身
void co_clean_self()
{
    printf("\n ## clean ##  %s  \n", co_running->name);
    co_running->status = CO_STATUS_EXIT; // 退出状态
    co_yield ();                         // 切换到主协程
}

// 创建协程
struct co *co_create(char *name, co_func func)
{
    // 申请协程对象
    struct co *co_new = malloc(sizeof(struct co));
    memset(co_new, 0, sizeof(struct co));
    strcpy(co_new->name, name);
    co_new->status = CO_STATUS_OK;

    // 没有协程函数，直接返回。表示主协程
    if (NULL == func)
    {
        return co_new;
    }

    // 申请协程栈
    co_new->stack_low = malloc(CO_STACK_SIZE);
    // 栈的上界
    uint64_t *tmp_addr = (uint64_t *)(co_new->stack_low + CO_STACK_SIZE);
    uint64_t tmp_up = (uint64_t)tmp_addr;
```

```
    // 把清理函数，压入栈顶。
    --tmp_addr;
    *tmp_addr = (uint64_t)co_clean_self;
    // 记录 rbp
    co_new->rbp = (uint64_t)tmp_addr;

    // 把协程函数，压入栈顶。
    --tmp_addr;
    *tmp_addr = (uint64_t)func;

    // 记录 rsp
    co_new->rsp = (uint64_t)tmp_addr;
    printf(" ## create ##  %13s  stack = %#llX ~ %#llX \n",
           name, (uint64_t)co_new->stack_low, tmp_up);
    return co_new;
}

// 协程切换。
void co_switch(struct co *from_co, struct co *to_co)
{
    // 更新正在运行的协程。
    // 放在切换上下文之前，因为切换后，协程栈上的 rip 可能导致跳转到其他地方。
    co_running = to_co;

    // 协程切换上下文
    co_switch_context(from_co, to_co);
}

// 运行某个子协程。主协程调用此方法。
void co_resume(struct co *co_sub)
{
    printf("\n ## resume ##   %s \n", co_sub->name);
    co_switch(co_main, co_sub);
}

// 当前协程切换出去。子协程调用此方法。
void co_yield ()
{
    co_switch(co_running, co_main);
}

// 清理协程
void co_delete(struct co *co)
{
    if (co->stack_low != NULL)
    {
        free(co->stack_low);
    }
    free(co);
```

```
}

// 从队列找到一个子协程。没有就返回 NULL
struct co *co_pick_from_list()
{
    // 找一圈，如果没有就退出。
    for (uint32_t m = 1; m <= CO_LIST_SIZE; ++m)
    {
        // 使用下标轮询。
        ++co_list_cursor;
        if (co_list_cursor >= CO_LIST_SIZE)
        {
            co_list_cursor = 0;
        }
        // 取一个。
        struct co *tmp = co_list[co_list_cursor];
        if (NULL == tmp)
        {
            continue;
        }
        // 协程退出了。
        if (tmp->status == CO_STATUS_EXIT)
        {
            // 清理这个协程。
            co_delete(tmp);
            co_list[co_list_cursor] = NULL;
            continue;
        }
        // 使用这个协程
        return tmp;
    }
    return NULL;
}

// 运行协程管理器
void co_handler_run()
{
    // 主协程
    co_main = co_create("main_co", NULL);

    // 从队列中挑选协程去运行
    co_list_cursor = 0;
    while (1)
    {
        // 从队列中找一个协程。
        struct co *one = co_pick_from_list();
        if (NULL == one)
        {
            break;
```

```
        }

        // 运行协程
        co_resume(one);
    }

    // 清理主协程
    co_delete(co_main);
    printf("\n === co handler end === \n\n");
}
```

编写代码： co_main.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <syscall.h>
#include "co_handler.h"

#define taskid syscall(SYS_gettid) // 线程 ID
#define co_name co_running->name   // 当前协程的名称

// 模拟 tcp 发送。非阻塞。
void func_tcp_send()
{
    int32_t tmp1 = 1;
    printf(" thread = %d co = %13s  1  at %p \n", taskid, co_name, &tmp1);
    co_yield ();
    int32_t tmp2 = 1;
    printf(" thread = %d co = %13s  2  at %p \n", taskid, co_name, &tmp2);
    co_yield ();
    int32_t tmp3 = 1;
    printf(" thread = %d co = %13s  3  at %p \n", taskid, co_name, &tmp3);
}

// 模拟写文件。非阻塞。
void func_write_file()
{
    int32_t tmp1 = 1;
    printf(" thread = %d co = %13s  1  at %p \n", taskid, co_name, &tmp1);
    co_yield ();
    int32_t tmp2 = 1;
    printf(" thread = %d co = %13s  2  at %p \n", taskid, co_name, &tmp2);
    co_yield ();
    int32_t tmp3 = 1;
    printf(" thread = %d co = %13s  3  at %p \n", taskid, co_name, &tmp3);
}

int main()
```

```
{
    int tmp3 = 1;
    printf(" main at %p \n", &tmp3);

    // 子协程。放入队列。
    struct co *co_tcp = co_create("Send_TCP_co", func_tcp_send);
    co_list[0] = co_tcp;
    struct co *co_file = co_create("Write_FILE_co", func_write_file);
    co_list[3] = co_file;

    // 运行协程管理器
    co_handler_run();

    return 0;
}
```

编译代码：

```
gcc co_switch_context.s co_handler.c co_main.c -std=gnu99 -o co_main
```

运行代码：

```
[root@local coroutine]# ./co_main
 main at 0x7fff89497f7c
 ## create ##    Send_TCP_co  stack = 0XB16080 ~ 0XB17080
 ## create ##  Write_FILE_co  stack = 0XB17100 ~ 0XB18100

 ## resume ##   Write_FILE_co
 thread = 83942 co = Write_FILE_co  1  at 0xb180dc

 ## resume ##   Send_TCP_co
 thread = 83942 co =   Send_TCP_co  1  at 0xb1705c

 ## resume ##   Write_FILE_co
 thread = 83942 co = Write_FILE_co  2  at 0xb180d8

 ## resume ##   Send_TCP_co
 thread = 83942 co =   Send_TCP_co  2  at 0xb17058

 ## resume ##   Write_FILE_co
 thread = 83942 co = Write_FILE_co  3  at 0xb180d4

 ## clean ##  Write_FILE_co

 ## resume ##   Send_TCP_co
 thread = 83942 co =   Send_TCP_co  3  at 0xb17054

 ## clean ##  Send_TCP_co

 === co handler end ===
```

分析结果：
创建 2 个协程，模拟 tcp 发送、模拟写文件，采用非阻塞模式。每个协程，执行 3 次输出函数，期间触发协程调度 co_yield。2 个协程交叉运行，属于同一个线程 83942。
函数 main 的变量的地址为 0x7fff89497f7c，在线程栈。协程栈使用函数 malloc 分配，在堆区。2 个协程栈的区间为 0XB16080 ~ 0XB17080、0XB16080 ~ 0XB17080。协程函数的变量的地址在协程栈，比如 0xb180dc 属于区间 0XB17100 ~ 0XB18100。

结构体 struct co 定义协程，核心属性包括寄存器、协程栈。(这里做了简化，可以包含更多寄存器)
全局变量 co_main 表示主协程。主协程没有协程函数，使用线程栈，主要作用为调度子协程。
全局变量 co_running 表示正在运行的子协程，记录协程调度之后运行的协程 `co_running = to_co` 。
函数 co_create 创建协程，使用函数 malloc 分配协程栈，把清理函数压入栈顶 `*tmp_addr = (uint64_t)co_clean_self` ，把协程函数压入栈顶 `*tmp_addr = (uint64_t)func` ，记录 rbp `co_new->rbp = (uint64_t)tmp_addr` 。
函数 co_switch_context 使用汇编实现，切换协程的上下文，包括栈寄存器 rbp、rsp，通用寄存器 rax、rbx、rcx、rdx。
函数 co_handler_run 实现简单版的协程调度器，从协程队列中取协程，然后切换协程。