协程概念，原理（c++和node.js实现）

发布于 1 年前 作者 [yyrdl](http://cnodejs.org/user/yyrdl) 10360 次浏览 来自 分享

**协程**

**什么是协程**

wikipedia 的定义： 协程是一个无优先级的子程序调度组件，允许子程序在特点的地方挂起恢复。

线程包含于进程，协程包含于线程。只要内存足够，一个线程中可以有任意多个协程，但某一时刻只能有一个协程在运行，多个协程分享该线程分配到的计算机资源。

**为什么需要协程**

**简单引入**

就实际使用理解来讲，协程允许我们写同步代码的逻辑，却做着异步的事，避免了回调嵌套，使得代码逻辑清晰。code like this:

co(function\*(next){

let [err,data]=yield fs.readFile("./test.txt",next);//异步读文件

[err]=yield fs.appendFile("./test2.txt",data,next);//异步写文件

//....

})()

**异步** 指令执行之后，结果并不立即显现的操作称为异步操作。及其指令执行完成并不代表操作完成。

协程是追求极限性能和优美的代码结构的产物。

**一点历史**

起初人们喜欢同步编程，然后发现有一堆线程因为I/O卡在那里,并发上不去，资源严重浪费。

然后出了异步（select,epoll,kqueue,etc）,将I/O操作交给内核线程,自己注册一个回调函数处理最终结果。

然而项目大了之后代码结构变得不清晰,下面是个小例子。

async\_func1("hello world",func(){

async\_func2("what's up?",func(){

async\_func2("oh ,friend!",func(){

//todo something

})

})

})

于是发明了协程，写同步的代码，享受着异步带来的性能优势。

**程序运行是需要的资源**：

* cpu
* 内存
* I/O (文件、网络，磁盘（内存访问不在一个层级，忽略不计）)

**协程的实现原理（c++和node.js里面的实现）**

**libco 一个C++协程库实现**

libco 是腾讯开源的一个C++协程库，作为微信后台的基础库，经受住了实际的检验。项目地址：<https://github.com/Tencent/libco>

个人源码阅读项目：<https://github.com/yyrdl/libco-code-study> （未完结）

libco源代码文件一共11个，其中一个是汇编代码，其余是C++，阅读起来相对较容易。

在C++里面实现协程要解决的问题有如下几个：

* 何时挂起协程？何时唤醒协程？
* 如何挂起、唤醒协程，如何保护协程运行时的上下文？
* 如何封装异步操作？

**前期知识准备**

1. 现代操作系统是分时操作系统，资源分配的基本单位是进程，CPU调度的基本单位是线程。
2. C++程序运行时会有一个运行时栈，一次函数调用就会在栈上生成一个record
3. 运行时内存空间分为全局变量区（存放函数，全局变量）,栈区，堆区。栈区内存分配从高地址往低地址分配，堆区从低地址往高地址分配。
4. 下一条指令地址存在于指令寄存器IP，ESP寄存值指向当前栈顶地址，EBP指向当前活动栈帧的基地址。
5. 发生函数调用时操作为：将参数从右往左依次压栈，将返回地址压栈，将当前EBP寄存器的值压栈，在栈区分配当前函数局部变量所需的空间，表现为修改ESP寄存器的值。
6. 协程的上下文包含属于他的栈区和寄存器里面存放的值。

**何时挂起，唤醒协程？**

如开始介绍时所说，协程是为了使用异步的优势，异步操作是为了避免IO操作阻塞线程。那么协程挂起的时刻应该是当前协程发起异步操作的时候，而唤醒应该在其他协程退出，并且他的异步操作完成时。

**如何挂起、唤醒协程，如何保护协程运行时的上下文？**

协程发起异步操作的时刻是该挂起协程的时刻，为了保证唤醒时能正常运行，需要正确保存并恢复其运行时的上下文。

所以这里的操作步骤为：

* 保存当前协程的上下文（运行栈，返回地址，寄存器状态）
* 设置将要唤醒的协程的入口指令地址到IP寄存器
* 恢复将要唤醒的协程的上下文

这部分操作相应的源代码：

.globl coctx\_swap//定义该部分汇编代码对外暴露的函数名

#if !defined( \_\_APPLE\_\_ )

.type coctx\_swap, @function

#endif

coctx\_swap:

#if defined(\_\_i386\_\_)

leal 4(%esp), %eax //sp R[eax]=R[esp]+4 R[eax]的值应该为coctx\_swap的第一个参数在栈中的地址

movl 4(%esp), %esp // R[esp]=Mem[R[esp]+4] 将esp指向 &(curr->ctx) 当前routine 上下文的内存地址，ctx在堆区，现在esp应指向reg[0]

leal 32(%esp), %esp //parm a : &regs[7] + sizeof(void\*) push 操作是以esp的值为基准，push一个值,则esp的值减一个单位（因为是按栈区的操作逻辑，从高位往低位分配地址），但ctx是在堆区，所以应将esp指向reg[7]，然后从eax到-4(%eax)push

//保存寄存器值到栈中，实际对应coctx\_t->regs 数组在栈中的位置（参见coctx.h 中coctx\_t的定义）

pushl %eax //esp ->parm a

pushl %ebp

pushl %esi

pushl %edi

pushl %edx

pushl %ecx

pushl %ebx

pushl -4(%eax) //将函数返回地址压栈，即coctx\_swap 之后的指令地址，保存返回地址,保存到coctx\_t->regs[0]

//恢复运行目标routine时的环境（各个寄存器的值和栈状态）

movl 4(%eax), %esp //parm b -> &regs[0] //切换esp到目标 routine ctx在栈中的起始地址,这个地址正好对应regs[0],pop一次 esp会加一个单位的值

popl %eax //ret func addr regs[0] 暂存返回地址到 EAX

//恢复当时的寄存器状态

popl %ebx // regs[1]

popl %ecx // regs[2]

popl %edx // regs[3]

popl %edi // regs[4]

popl %esi // regs[5]

popl %ebp // regs[6]

popl %esp // regs[7]

//将返回地址压栈

pushl %eax //set ret func addr

//将 eax清零

xorl %eax, %eax

//返回，这里返回之后就切换到目标routine了，C++代码中调用coctx\_swap的地方之后的代码将得不到立即执行

ret

#elif

这部分代码只是做了寄存器部分的操作。依赖的结构体定义，见文件coctx.h中：

struct coctx\_t

{

#if defined(\_\_i386\_\_)

void \*regs[ 8 ];//32位机，依次为：ret,ebx,ecx,edx,edi,esi,ebp,eax

#else

void \*regs[ 14 ];//64位机的情况

#endif

size\_t ss\_size;//空间大小

char \*ss\_sp;//ESP

};

调用coctx\_swap 函数只在文件co\_routine.cpp中的co\_swap函数。

保存运行栈的操作见co\_swap函数中调用coctx\_swap之前的部分。具体步骤为取当前栈顶地址 （代码：char c; esp=&c）,若不是共享栈模型则清理下env，若是则判断共享栈区有没有被占用，被占用则从堆区申请内存保存，然后再分配共享栈。

需要注意的是，libco运行时的栈区不在是传统意义上的栈区，其空间实际来自于堆区。

**如何封装异步操作？**

这部分代码见：

* co\_hook\_sys\_call.cpp
* co\_routine.cpp
* co\_epoll.cpp
* co\_epoll.h

核心思想是hook系统本来的I/O接口，比如socket()函数，和epoll(kqueue)结合，采用一个co\_eventloop来统一管理，当发现一个协程发起异步操作时，就将其挂起放入等待队列，唤醒其他异步操作已经完成的协程。可以联系libevent里面的event\_loop，区别在在于一个是操作栈区和寄存器恢复协程，一个是调用绑定的回调函数。

**node.js里面协程**

node.js 的优势：

* node.js天生异步（下面是libuv）
* javascript的闭包特性完成了上下文的保存工作

需要我们做的：

* 实现同步编程

附上 文章开始时的代码：

const fs=require("fs");

const co=require("zco");

co(function\*(next){

let [err,data]=yield fs.readFile("./test.txt",next);//异步读文件

[err]=yield fs.appendFile("./test2.txt",data,next);//异步写文件

//....

})()

**JS 中的Generator**

Generator是一个迭代器生成器,也是node.js中实现协程的关键。

let gen=function \*() {

console.log("ok1");

var a=yield 1;

console.log("a:"+a);

var b=yield 2;

console.log("b:"+b);

}

var iterator=gen();

console.log("ok2");

console.log(iterator.next(100));

console.log(iterator.next(101));

console.log(iterator.next(102));

输出：

ok2

ok1

{ value: 1, done: false }

a:101

{ value: 2, done: false }

b:102

{ value: undefined, done: true }

从这里我们可以看到其执行顺序，以及各个值的变化。iterator.next() 返回的值即yield 之后的表达式的返回值，yield之前的变量的值即iterator.next方法传入的值。通过这个特性，合理包装即可实现coroutine.

以下是zco模块源码，项目地址：<https://github.com/yyrdl/zco> ：

/\*\*

\* Created by yyrdl on 2017/3/14.

\*/

var slice = Array.prototype.slice;

var co = function (gen) {

var iterator,

callback = null,

hasReturn = false;

var \_end = function (e, v) {

callback && callback(e, v); //I shoudn't catch the error throwed by user's callback

if(callback==null&&e){//the error should be throwed if no handler instead of catching silently

throw e;

}

}

var run=function(arg){

try {

var v = iterator.next(arg);

hasReturn = true;

v.done && \_end(undefined, v.value);

} catch (e) {

\_end(e);

}

}

var nextSlave = function (arg) {

hasReturn = false;

run(arg);

}

var next = function () {

var arg = slice.call(arguments);

if (!hasReturn) {//support fake async operation,avoid error: "Generator is already running"

setTimeout(nextSlave, 0, arg);

} else {

nextSlave(arg);

}

}

if ("[object GeneratorFunction]" === Object.prototype.toString.call(gen)) {//todo: support other Generator implements

iterator = gen(next);

} else {

throw new TypeError("the arg of co must be generator function")

}

var future = function (cb) {

if ("function" == typeof cb) {

callback = cb;

}

run();

}

return future;

}

module.exports = co;