从递归说开去

CNX(陈楠昕)

bobchennan@gmail.com

July 26, 2012

声明

大概 YY 了一个周末，看了很多东西，开始做这个 pdf。本文更多是关于一些方法技巧的介绍以及个人一些小的想法，原创性内容较少。

另外本文中牵扯到的语言主要有 C、C++、C#、Python。

意图

为什么想到这个？这个理由会在最后揭晓。大部分我要讲的内容与我的认知过程恰好想反。

About fac

考虑到一个常见的求阶乘程序:

fac-1.c

```
int fac(int n){
    return n?n*fac(n-1):1;
}
```

我们不加参数直接编译它:gcc fac-1.c -o fac-1
在 Windows 环境下大概 13W 左右会崩栈
加上编译优化:gcc fac-1.c -o fac-1 -O2
不会发生崩栈

原因

函数调用以及局部变量维护通过栈来实现,因此反复递归到一定程度就会崩栈。

计算 4!程序本身调用过程如下:

(fac 4)
(* 4 (fac 3))
(* 4 (* 3 (fac 2)))
(* 4 (* 3 (* 2 (fac 1))))
(* 4 (* 3 (* 2 (* 1 (fac 0)))))
(* 4 (* 3 (* 2 (* 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6) 24

步骤 1

考虑到这一类问题特殊性,我们可以考虑通过不回溯的方法实现:

fac-2.c

```
int fac(int n,int v){
    return n?fac(n-1,v*n):v;
}
```

当然这段代码依然需要加编译优化 O2 才能正常运行。
但是这段代码有着明显的特点,即函数最后一步是返回关于自身的调用,并且不需要回溯过程,即没有必要每次保存任何局部变量,这样的递归被称为尾递归。

尾递归

尾递归本质上来说就是一种迭代过程,与循环并无区别,因此程序可改成如下代码:

fac-3.c

```
int fac(int n,int v){
    for(;;){
        if(n==0)return v;
        v*=n;
        --n;
    }
}
```

实际上编译器编译优化就是通过相似原理实现,简单的说就是局部变量不压栈,下次调用直接使用本次空间,或者干脆只使用寄存器。

其他语言

C 与 C++ 依靠编译优化实现了尾递归,其他语言呢？

比如 Python 和 Java,C# 一般来说是没有尾递归自动优化的能力的,递归调用受到调用栈长度的限制。

如何解决此问题呢？

Python

Third, I don't believe in recursion as the basis of all programming. This is a fundamental belief of certain computer scientists, especially those who love Scheme and like to teach programming by starting with a "cons"cell and recursion. But to me, seeing recursion as the basis of everything else is just a nice theoretical approach to fundamental mathematics (turtles all the way down), not a day-to-day tool.(By Guido)

Python2

但是我们依然可以用一些巧妙的方法来解决这个问题：(catch.py)

类似的 C++ 实现

C#

C# 中有着更加神奇的方法,叫做 Continuation。

Continuation 是什么？简而言之,就是一种控制结构,利用它可以把当前的执行状态保存起来,以供以后调用。

简单地说,现在你像持有一个普通对象一样持有一个执行环境,你可以把它保存到任何地方 (不再被限制到栈上),并随时再进入到那个执行环境中继续先前中断的执行过程。

Continuation

打个比方:

话说你在厨房的冰箱前,考虑来点三明治尝尝。如果这时候你做了个 continuation,并将其存放到你的口袋中。然后你把火鸡和面包从冰箱里取出来拿到餐台,并把他们做成了你想要的三明治. 此时,如果你把你的 continuation 从口袋中取出来,并且调用一次的话。你会发现你突然又身处冰箱前, 考虑来点三明治尝尝. 不过,幸运的是, 此时餐台上已经有了个你想要的三明治。而火鸡和面包都不见了. 那么,你就可以径直去把它吃掉了。:-)

Continuation 可以非常容易的实现高级流程控制:大规模退出 (非结构化:goto;结构化:异常)、生成器、回溯。

C#(2)

Listing 1: fac.cs

```
public static int FactorialContinuation(int n,
        Func<int, int> continuation)
{
    if (n == 0) return continuation(1);
    return FactorialContinuation(n - 1,
        r => continuation(n * r));
}
FactorialContinuation(10, x => x)
```

C#(3)

```
public static int FibonacciContinua-
tion(int n, Func<int, int> continua-
tion) {
    if (n < 2) return continuation(n);
    return FibonacciContinuation(n - 1,
       r1 => FibonacciContinuation(n - 2,
           r2 => continuation(r1 + r2)));
}
```

fac-3

C++(2)

fib.cs

My Opinion 1

传统语言中使用堆栈作为函数参数及局部变量存储位置,具有以下优势:

- 存储速度比堆要快,仅次于寄存器

My Opinion 1

传统语言中使用堆栈作为函数参数及局部变量存储位置,具有以下优势:

- 存储速度比堆要快,仅次于寄存器
- 栈数据能够共享

My Opinion 1

传统语言中使用堆栈作为函数参数及局部变量存储位置,具有以下优势:

- 存储速度比堆要快,仅次于寄存器
- 栈数据能够共享
- 便于中断、恢复、嵌套

My Opinion 1

传统语言中使用堆栈作为函数参数及局部变量存储位置,具有以下优势:

- 存储速度比堆要快,仅次于寄存器
- 栈数据能够共享
- 便于中断、恢复、嵌套

缺点:

- 大小、生存时间固定

My Opinion 2

Stack

My Opinion 2

Stack

$\Downarrow$

My Opinion 2

Stack

$\Downarrow$

Heap(or queue?)

函数式语言：Erlang、Scheme 等。

Coroutine

协程看上去与目前所讲的关系不大。

Coroutine

协程看上去与目前所讲的关系不大。
事实上确实没太大关系。

Coroutine

协程看上去与目前所讲的关系不大。
事实上确实没太大关系。
于是我现在开始真正介绍下背景。。

Background

某天晚上洗衣服,我向蒋委询问 new feature 的素材,他跟我提到了下面这个东西 (python 代码):

```
def travel(max):
    i=0
    while i<max:
        yield i
        i=i+1

for n in travel(100):
    print n
```

Again

如何用 C 和 C++ 完成类似功能？

Solution1

```
int function(void) {
    static int i, state = 0;
    switch (state) {
        case 0: goto LABEL0;
        case 1: goto LABEL1;
    }
    LABEL0: /* start of function */
    for (i = 0; i < 10; i++) {
        state = 1; /* so we will come back to LABEL
        return i;
        LABEL1:; /* resume control straight after th
    }
}
```

小技巧

```
switch (count % 8) {
    case 0:         do {  *to = *from++;
    case 7:               *to = *from++;
    case 6:               *to = *from++;
    case 5:               *to = *from++;
    case 4:               *to = *from++;
    case 3:               *to = *from++;
    case 2:               *to = *from++;
    case 1:               *to = *from++;
                    } while ((count -= 8) > 0);
}
```

Solution2

```
int function(void) {
    static int i, state = 0;
    switch (state) {
        case 0: /* start of function */
        for (i = 0; i < 10; i++) {
            state = 1; /* so we will come back to "
            return i;
            case 1:; /* resume control straight aft
        }
    }
}
```

Solution2-2

```
#include <stdio.h>

#define cB static int state=0;switch(state){case 0:
#define cR(x) do{state=__LINE__;return x;\
                    case __LINE__:;}while(0)
#define cF }

int f(void){
    static int i;
    cB;
    for(i=0;i<10;++i)
        cR(i);
    cF;
}

int main(void){
    for(int i=0;i<10;++i)
        printf("%d\n",f());
```

define

define1.c

define(2)

define2.c

What can we do?

下面我们来看一下蒋委问题的一个答案,考虑二叉树遍历,C# 中我们可以通过高阶迭代器实现 (tree.cs)

the same in C

tree.c

What can coroutine do?

比较精妙，说不太清楚。

What can coroutine do?

比较精妙，说不太清楚。

目前资料中提到的比较多的是速度优势，特别是相比continuation。

当然还有以下特点：

- 避免了传统的函数调用栈，使得无限递归成为可能
- 用户态的线程调度，极大降低上下文切换的开销，使得近乎无限并发的“微线程”成为可能
- 由于可以在用户态进行手工线程调度，这样可以避免锁机制

What can coroutine do?

比较精妙，说不太清楚。

目前资料中提到的比较多的是速度优势，特别是相比 continuation。

当然还有以下特点：

- 避免了传统的函数调用栈，使得无限递归成为可能
- 用户态的线程调度，极大降低上下文切换的开销，使得近乎无限并发的“微线程”成为可能
- 由于可以在用户态进行手工线程调度，这样可以避免锁机制

举一个例子，服务器的发展是多进程 => 多线程 => 异步 => 协程。而现在主流的高性能服务器都是基于异步的 (lighttpd、nginx)，但实测协程速度更加惊人 (python stackless)。

Reference


维基百科


各种网络 blog