快速排序的栈深度

陈昱名

01/ 栈深度

对于栈深度的定义：
编译器通常使用栈来存储递归执行过程中的相关信息，包括每次递归调用的参数等。最新调用的信息存在栈的顶部，而第一次调用的信息存在栈的底部。当一个过程被调用时，其相关信息被压如栈中，当它结束时，其信息被弹出。栈深度是在一次计算中会用到的栈空间的最大值。

02/ 尾递归

如果一个函数中所有递归形式的调用都出现在函数的末尾，我们称这个递归函数是尾递归的。当递归调用是整个函数体中最后执行的语句且它的返回值不属于表达式的一部分时，这个递归调用就是尾递归。尾递归函数的特点是在回归过程中不用做任何操作，这个特性很重要，因为大多数现代的编译器会利用这种特点自动生成优化的代码。

当编译器检测到一个函数调用是尾递归的时候，它就覆盖当前的活动记录而不是在栈中去创建一个新的。编译器可以做到这点，因为递归调用是当前活跃期内最后一条待执行的语句，于是当这个调用返回时栈帧中并没有其他事情可做，因此也就没有保存栈帧的必要了。通过覆盖当前的栈帧而不是在其之上重新添加一个，这样所使用的栈空间就大大缩减了，这使得实际的运行效率会变得更高。

03/ Problem

7-4 Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

a. Argue that TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ correctly sorts the array A.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

04/ Problem 7.4(a)

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

TAIL-RECUTIVE-QUICKSORT-QUICKSORT（A,1,A.length）能正确的对数组A排序

这段代码在本质上与原本的快排并无改动，这个改进的主要思想应该是：用没处理的子节点代替父节点，以减少栈的深度。
假设父节点A，子节点B、C。处理完B后，并不直接处理C而是回溯回A，并把C交到A这一级来处理。通过这种方法的确可以将栈的最大深度降到O(logn)。而并未改变代码的时间复杂度与正确性。

证明

当p=r时，已经正确排序了。对于有n个元素的数组，假设对于任何小于n个元素的数组都可以正确排序。因为Line3的Partition可以把数组分成q左边的比A[q]小的数组，和右边的比A[q]大的数组。因为左数组元素个数必小于n，调用quicksort’后可以正确排序。p=q+1后，quicksort’所要排序的数组A[q+1,…,r]元素也小于n，所以也能正确排序。左右两边都是有序的而且右边的任何一个元素大于等于左边的任何元素，所以整个数组都是有序的。

05/ Problem 7.4(b)

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

请描述一种场景，使得针对一个包含n个元素数组的TAIL-RECUTIVE-QUICKSORT的栈深度是 $\Theta(n)$

这种悲催的情况只能发生在，partition划分出来的q=r，右数组为空，左数组比原数组大小只减少了1.那么Line4就要用n个递归了。栈深度就是 $\Theta(n)$ 了。即这个数组本身即为顺序数组。

06/ Problem 7.4(c)

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

修改TAIL-RECURSIVE-QUICKSORT的代码，使其最坏情况下栈深度是 $\Theta(\lg n)$ 并保持 $O(n \lg n)$ 的期望时间复杂度

由问题b我们可以知道只要每次选择要递归的半个数组尽可能的小就行了，那么我们的策略是每次选取左右数组中较小的那个进行递归调用。那么这样改动对程序本身的时间复杂度没有影响还为$O(n \lg n)$ 。

那么我们只要把Line4-5改成先排序小数组就可以了，因为每次递归总会使得要排序的数组最多只为原来的一半，所以递归深度只有$\Theta(\lg n)$

代码

```
1  TAIL-RECURSIVE-QUICKSORT'(A, p, r)
2      while p < r
3          q = PARTITION(A, p, r)
4          if (q - p < r - q)
5              TAIL-RECURSIVE-QUICKSORT'(A, p, q)
6              p = q + 1
7          else
8              TAIL-RECURSIVE-QUICKSORT'(A, q + 1, r)
9              r = q - 1
```

尾递归是极其重要的，不用尾递归，函数的堆栈耗用难以估量，需要保存很多中间函数的堆栈。
编译器会将这些调用进行优化使之变为简单的跳转，从而节省函数调用在时间和空间上的开销，提高运行效率。
尾递归有时会等同于一个回到函数开始位置的循环，因此，有时也使用尾递归来代替常见的循环、goto或continue语句，不过并不多见。

07/ Example

尾递归与传统递归比较

以下是具体实例:

线性递归:

```
long Rescuvie(long n) {

    return (n == 1) ? 1 : n * Rescuvie(n - 1);

}
```

尾递归:

```
long TailRescuvie(long n, long a) {

    return (n == 1) ? a : TailRescuvie(n - 1, a * n);

}


long TailRescuvie(long n) {//封装用的

    return (n == 0) ? 1 : TailRescuvie(n, 1);

}
```

07/ Example

当n = 5时

对于传统线性递归, 他的递归过程如下:

```php
Rescuvie(5)

{5 * Rescuvie(4)}

{5 * {4 * Rescuvie(3)}}

{5 * {4 * {3 * Rescuvie(2)}}}

{5 * {4 * {3 * {2 * Rescuvie(1)}}}}

{5 * {4 * {3 * {2 * 1}}}}

{5 * {4 * {3 * 2}}}

{5 * {4 * 6}}

{5 * 24}

120
```

对于尾递归, 他的递归过程如下:

```php
TailRescuvie(5)                    // 所以在运算上和内存占用上节省了很多,直接传回结果

TailRescuvie(5, 1)                      return 120
                                             ↑
TailRescuvie(4, 5)                      return 120
                                             ↑
TailRescuvie(3, 20)                     return 120
                                             ↑
TailRescuvie(2, 60)                     return 120
                                             ↑
TailRescuvie(1, 120)                    return 120
                                             ↑
120                              //当运行到最后时,return a => return 120 ,将120返回上一级
```

与普通递归相比，由于尾递归的调用处于方法的最后，因此方法之前所积累下的各种状态对于递归调用结果已经没有任何意义，因此完全可以把本次方法中留在堆栈中的数据完全清除，把空间让给最后的递归调用。这样的优化便使得递归不会在调用堆栈上产生堆积，意味着即时是“无限”递归也不会让堆栈溢出。这便是尾递归的优势。

Thank you for watching