The Taste of Language 语言的品味

Chunting Gu, 2011/6/3

Agenda

- 一种语言，方式多样
- 语言不同，风格迥异
- 风格虽同，思想有别
- 尾调用/递归
- 函数作为第一级对象

一种语言，方式多样

- Split string in C++ with strtok
- Split string in C++ with boost

- Factorial (阶乘) in C++ with recursion
- Factorial in C++ with iteration
- Factorial in C++ with tail recursion

Split string in C++ with strtok

```
char* strtok(char* str, const char* delimiters);

char str[] = "- This, a sample string.";
char* p = strtok(str," ,.-");
while (p != NULL) {
    cout << p << endl;
    p = strtok(NULL, " ,.-");
}
```

优点：空间效率高，直接改写输入的字符串。
缺点：如果不能改写输入，则不适合；C 风格的用法稍为原始。

Split string in C++ with boost

```
#include "boost/foreach.hpp"
#include "boost/tokenizer.hpp"

string str = "- This, a sample string.";
char_separator<char> sep(" ,.-");
tokenizer<char_separator<char> > tokens(str, sep);
BOOST_FOREACH(string t, tokens) { // Copy happens here!
   cout << t << endl;
}
```

优点：易用？
缺点：需要拷贝每一个子串，空间效率打了折扣。

Factorial in C++ with recursion

```
size_t fac(size_t n) {
    return n == 0 ? 1 : n * fac(n - 1);
}
```

> 递归的实现，简单明了，基本上就是阶乘的定义。但是效率较差。

Factorial in C++ with iteration

```cpp
size_t fac(size_t n) {
    size_t result = n;
    while (n > 1) { result *= --n; }
    return result;
}
```

避免了递归，使用迭代，效率较高。但是不如递归实现明了。

Factorial in C++ with tail recursion

```cpp
size_t __fac(size_t acc, size_t n) {
    if (n == 0) { return acc; }
    return __fac(acc * n, n - 1);
}
size_t fac(size_t n) {
    return n == 0 ? 1 : __fac(n, n - 1);
}
```

使用了尾递归，但是对 C++来说，效率并不会比一般的递归有所改善。

语言不同，风格迥异

- Quick sort in C++
- Quick sort in Haskell

- Fibonacci in C++
- Fibonacci in Haskell

Quick sort in C++

```
template <typename RanIt>
void qsort(RanIt begin, RanIt end) {
  if (end > begin + 1) {
      RanIt w = split(begin, end); // split 为划分算法，从略
      qsort(begin, w);
      qsort(w+1, end);
  }
}
```

STL 风格，使用模板和迭代器，保证效率的同时，又兼顾了复用性。

Quick sort in Haskell

```haskell
qsort []        = []
qsort (x:xs)    = qsort [y | y <- xs, y < x]
                  ++ [x]
                  ++ qsort [y | y <- xs, y >= x]
```

> 类似于数学里的集合表示法，简单明了。

Fibonacci in C++

```cpp
int fib(int n) {
    int a = 0, b = 1, t;
    for (int i = 0; i < n; ++i) {
        t = a + b;
        a = b;
        b = t;
    }
    return a;
}
```

> 避免了递归的实现，使用迭代，效率较高。但是一次只能返回一个结果。
> 可以用 `std::vector` 实现返回数组。

Fabonacci in Haskell

```haskell
fibs :: [Int]
fibs = 0 : 1 : [a + b | (a, b) <- zip fibs (tail fibs)]
```

```haskell
tail [1, 2, 3] = [2, 3]

zip [1, 2, 3] ["one", "two", "three"]
  = [(1, "one"), (2, "two"), (3, "three")]

main = do
        print (fibs !! 7) -- 13
```

这个实现对递归的运用相当精巧。充分显示了 Haskell 的 lazy 特质。

风格虽同，思想有别

- Factorial in C++ with tail recursion
- Factorial in Lua with proper tail recursion

Factorial in C++ with tail recursion

```cpp
size_t __fac(size_t acc, size_t n) {
    if (n == 0) { return acc; }
    return __fac(acc * n, n - 1);
}
size_t fac(size_t n) {
    return n == 0 ? 1 : __fac(n, n - 1);
}
```

> 使用了尾递归，可惜对 C++来说，效率并不会有所改善。

Factorial in Lua with proper tail recursion

```lua
function fac(n)
    function __fac(acc, n)
        if n == 0 then
            return acc
        else
            return __fac(acc * n, n - 1)
        end
    end
    return n == 0 and 1 or __fac(n, n - 1)
end
```

> 因为 Lua 对尾调用/递归有优化（即严格尾调用/递归），所以效率和迭代相仿。

尾调用/递归

> A tail call is a subroutine call that happens inside another procedure and that produces a return value, which is then immediately returned by the calling procedure.
>
> If a subroutine performs a tail call to itself, it is called tail-recursive.

Tail call/recursion examples

`function foo(data)`
`  A(data);`
`  return B(data);`
	B is a tail call.
	A is not.

`function bar(data)`
`  if A(data)`
`  return B(data);`
`  else`
`  return C(data);`
	B and C both are tail calls.
	A is not.

Tail call/recursion examples (cont.)

`function foo(data)` `  return A(data) + 1;`	A is not a tail call.

`function foo(data)`
`  var ret = A(data);`
`  return (ret == 0) ? 1 : ret;`
	A is not a tail call.

函数作为第一级对象

> – SICP
>
> In general, programming languages impose restrictions on the ways in which computational elements can be manipulated.
>
> Elements with the fewest restrictions are said to have *first-class* status. Some of the “rights and privileges” of first-class elements are:

- They may be named by variables.
- They may be passed as arguments to procedures.
- They may be returned as the results of procedures.
- They may be included in data structures.

Lambda expression

> 可以简单地理解成匿名函数（anonymous function）。

```
int A[] = {1, 2, 3, 4, 5};
// Before C++0x
int multiply(int x, int y) { return x * y; }
accumulate(A, A + 5, 1, multiply); // 120
// C++0x
accumulate(A, A + 5, 1, [](int x, int y) { return x * y; });
```

```
(fold-left (lambda (x y) (* x y)) 1 '(1 2 3 4 5)))
; 其实可以这样写：
(fold-left * 1 '(1 2 3 4 5)))
```

Closure

> – SICP
>
> A closure is an implementation technique for representing procedures with free variables.

> – 郑晖《冒号课堂》
>
> 包：函数与其周围的环境变量捆绑打包；
>
> 闭：这些变量是封闭的，只能为该函数所专用。

> – SICP （实现上的代价）
>
> The major implementation cost of first-class procedures is that allowing procedures to be returned as values requires reserving storage for a procedure's free variables even while the procedure is not executing.

First-class function in Lua

```lua
function newCounter (step)
    local i = 0
    return function ()
        i = step(i)
        return i
    end
end
```

```lua
c1 = newCounter(function (i)
    i = i + 1
    return i
end)
print(c1())  --> 1
print(c1())  --> 2

c2 = newCounter(function (i)
    i = i + 2
    return i
end)
print(c2())  --> 2
print(c1())  --> 3
print(c2())  --> 4
```

First-class function in C#

```
public static Func<int> newCounter(Func<int, int> step)
{
    var i = 0;
    Func<int> inc = delegate()
    {
        i = step(i);
        return i;
    };
    return inc;
}
```

First-class function in C# (cont.)

```
static void Main(string[] args)
{
    var c1 = newCounter(delegate(int i) { return i + 1; });
    Console.WriteLine(c1()); // 1
    Console.WriteLine(c1()); // 2

    var c2 = newCounter(delegate(int i) { return i + 2; });
    Console.WriteLine(c2()); // 2
    Console.WriteLine(c1()); // 3
    Console.WriteLine(c2()); // 4
}
```

First-class function in C++0x

```
#include <functional>
using namespace std;
function<int()> newCounter(function<int(int)> step) {
  int i = 0;
  return ([&i, &step]()->int{
     i = step(i);
     return i;
  });
}
auto c1 = newCounter([](int i){ return i + 1; });
cout << c1() << endl;
cout << c1() << endl;
```

```
Crash in VC++ 2010!
Unexpected result in g++ 5.4.2.
```

The End.

Thank you for your time!