Application of Closure Function

Practices and Reflections on 'Fast' Access Memory

Hongjia Liu @ AxiomQuant

2022.08.12

Press Space for next page →

Seminar Outline

All from a small thought at work...

- 🤔 **When does it apply?** - "acceleration" for large number of repetitive calculations
- 🔪 **Where to start?** - all starts with the computer memory model
- 👷 **What is a closure?** - "a closure gives you access to an outer function's scope from an inner function" [1]
- 🧰 **How to use?** - combining closures with the scenarios we mentioned at the beginning
- 🧪 **Why use it** - the results of experiment

[1] closure: developer.mozilla.org

"large" number of "repetitive" calculations

```
# Hard Code: The variables of the calculation are fixed
# and the rules of the calculation are also fixed
class Base:
  def cal(self, value) -> None:
    self.num = self.num + 1
    self.cum = self.cum + value
    self.avg = self.cum / self.num

b = Base()
... b.cal(value)  # will be called for many times

# The variables of the calculation are NOT fixed
class MutableVars:
  def __init__(self, data: Dict[str, int]):
    self.data = data
  def cal(self, value) -> None:
    self.data["num"] = self.data["num"] + 1
    self.data["cum"] = self.data["cum"] + value
    self.data["avg"] = self.data["cum"] / self.data["num"]

global_data = get_dict()
m = MutableVars(global_data)
... m.cal(value)
```

```
class Base:
  def cal(self, value) -> None:
    self.num = self.num + 1
    self.cum = self.cum + value
    self.avg = self.cum / self.num

class MutableVars:
  def __init__(self, data):
    self.data = data
  def cal(self, value) -> None:
    self.data["num"] = self.data["num"] + 1
    self.data["cum"] = self.data["cum"] + value
    self.data["avg"] = self.data["cum"] / self.data["num"]

# Variables and rules are both NOT fixed
class Mutable:
  def __init__(self, rules) -> None:
    self.rules = rules
    self.loadVars()
  def cal() -> None:
    for rule in self.rules:
      if rule["rule"] == "*":
        self.vars[rule["name3"]] = self.vars[rule["name1"]] * self.vars[rule["name2"]]
      elif ...
```

Which is fast?

```
class Base:
  def cal(self, value) -> None:
    self.num = self.num + 1
    self.cum = self.cum + value
    self.avg = self.cum / self.num
```

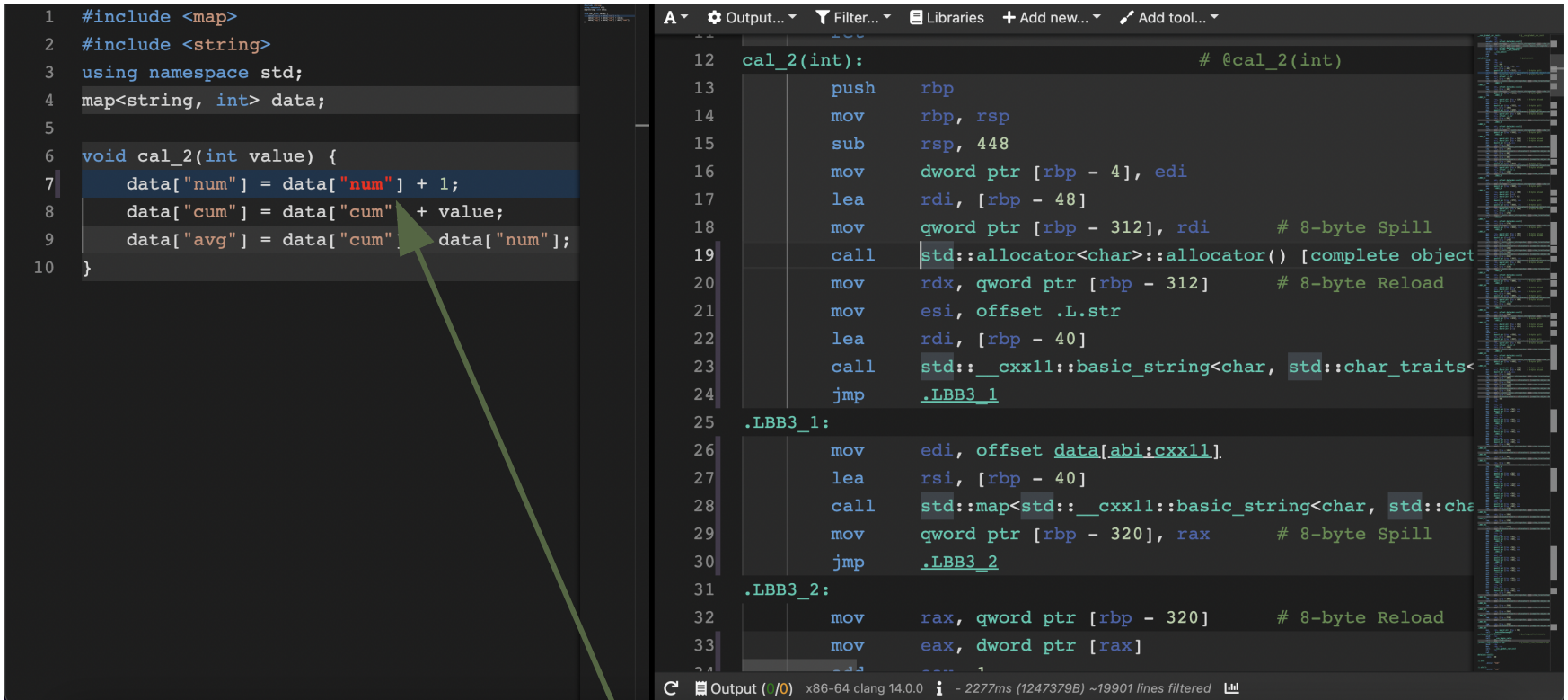
```
class MutableVars:
  def __init__(self, data):
    self.data = data
  def cal(self, value) -> None:
    self.data["num"] = self.data["num"] + 1
    self.data["cum"] = self.data["cum"] + value
    self.data["avg"] = self.data["cum"] / self.data["num"]
```

Which is fast?

```
1  int num = 0;
2  int cum = 0;
3  int avg = 0;
4
5  void cal_1(int value) {
6      num = num + 1;
7      cum = cum + value;
8      avg = cum / num;
9  }
```

```
1  cal_1(int):                                  # @cal_1(int)
2          push    rbp
3          mov     rbp, rsp
4          mov     dword ptr [rbp - 4], edi
5          mov     eax, dword ptr [num]
6          add     eax, 1
7          mov     dword ptr [num], eax
8          mov     eax, dword ptr [cum]
9          add     eax, dword ptr [rbp - 4]
10         mov     dword ptr [cum], eax
11         mov     eax, dword ptr [cum]
12         cdq
13         idiv    dword ptr [num]
14         mov     dword ptr [avg], eax
15         pop     rbp
16         ret
```

Which is fast?



```cpp
#include <map>
#include <string>
using namespace std;
map<string, int> data;

void cal_2(int value) {
    data["num"] = data["num"] + 1;
    data["cum"] = data["cum"] + value;
    data["avg"] = data["cum"] / data["num"];
}
```

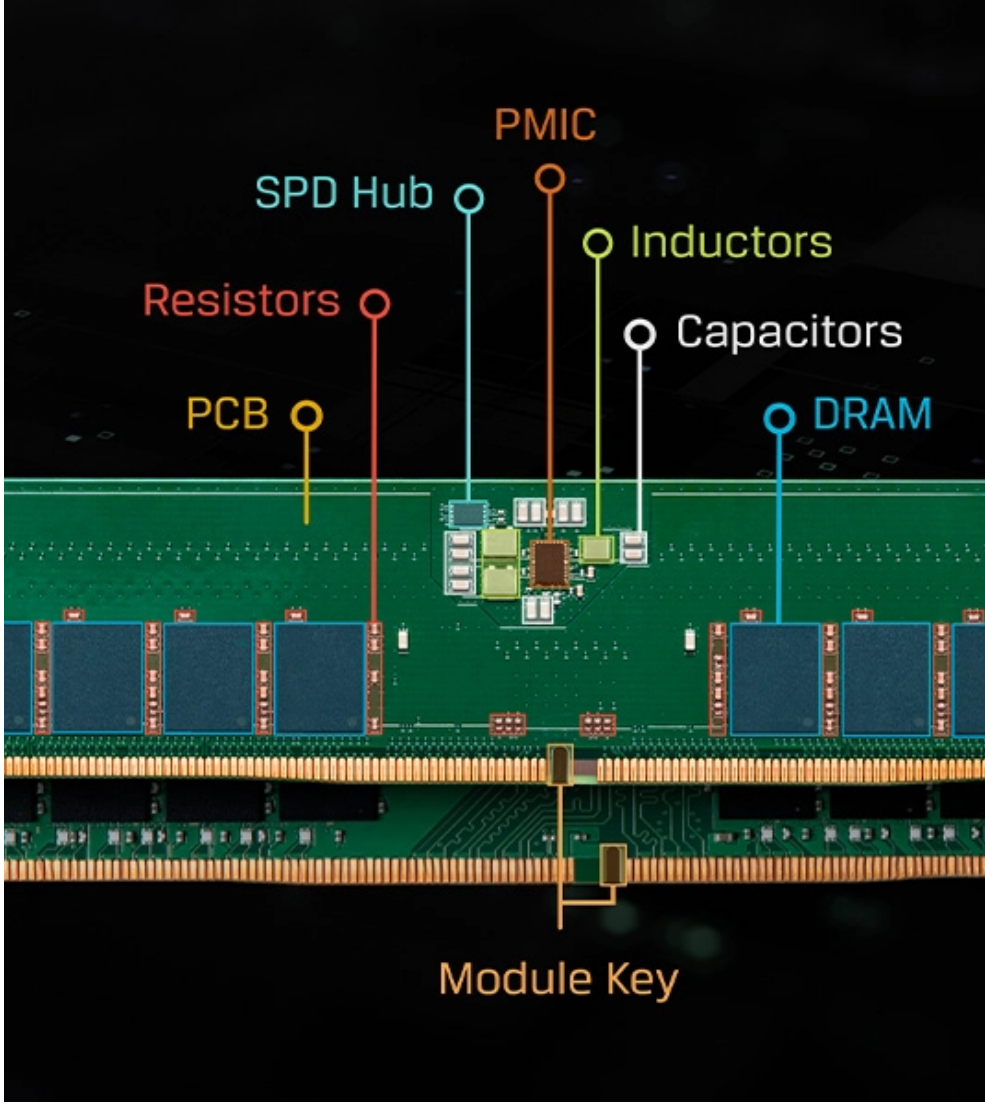
```
cal_2(int):                                # @cal_2(int)
        push    rbp
        mov     rbp, rsp
        sub     rsp, 448
        mov     dword ptr [rbp - 4], edi
        lea     rdi, [rbp - 48]
        mov     qword ptr [rbp - 312], rdi      # 8-byte Spill
        call    std::allocator<char>::allocator() [complete object
        mov     rdx, qword ptr [rbp - 312]      # 8-byte Reload
        mov     esi, offset .L.str
        lea     rdi, [rbp - 40]
        call    std::__cxx11::basic_string<char, std::char_traits<
        jmp     .LBB3_1
.LBB3_1:
        mov     edi, offset data[abi:cxx11]
        lea     rsi, [rbp - 40]
        call    std::map<std::__cxx11::basic_string<char, std::cha
        mov     qword ptr [rbp - 320], rax      # 8-byte Spill
        jmp     .LBB3_2
.LBB3_2:
        mov     rax, qword ptr [rbp - 320]      # 8-byte Reload
        mov     eax, dword ptr [rax]
```

What if we do this process of hashing well in advance?

Before Closure Function ...

Let’s first look at the computer memory structure.

```
0x8f---
    stack  call a function
           ...
0x6f---
    heap   auto a = New Base()
           ...
0x5f---
    static    num = 1
              ...
0x4f---
    code  mov rbp rsp
          ...
0x00---
```



'Living' in Stack

```
0x8f---
    stack  call a function
           begin cal_twice
           res = cal(value)   res = res + 1
           end cal_twice
           begin cal
           temp = value + 1   ...
           end cal
0x4f---
    code  mov rbp rsp
          load codes...
          cal_twice 1
0x00---
```

```
def cal(value):
  temp = value + 1
  return temp

def cal_twice(value):
  res = cal(value)
  res = res + 1
  return res

cal_twice(1)
```

Where will "temp" be?

```
0x8f---
    stack  call a function
           begin cal_twice
           res = cal(value)   res = res + 1
           end cal_twice
           begin cal
           temp = value + 1   ...
           end cal
0x4f---
    code  mov rbp rsp
          load codes...
          cal_twice 1
0x00---

def cal(value):
  temp = value + 1
  return temp

def cal_twice(value):
  res = cal(value)
  res = res + 1
  return res

cal_twice(1)
```

Where will "temp" be?

```
0x8f---
    stack  call a function
           begin cal_twice
           res = cal(value)   res = res + 1
           end cal_twice



0x4f---
    code  mov rbp rsp
          load codes...
          cal_twice 1
0x00---

def cal(value):
  temp = value + 1
  return temp

def cal_twice(value):
  res = cal(value)
  res = res + 1
  return res

cal_twice(1)
```

Here comes the closure function

MDN: A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
function useCountEffect() {
  count = 0;  // It will not be discarded because it is on the "stack"!
  return function() {
    count++;
    console.log()
  }
}

count1 = useCountEffect();
count2 = useCountEffect();

count1();  // 1
count1();  // 2
count2();  // 1
```

Closure Function in Our cases

```
class Field:
  value = 0

data = {"num": Field(), "cum": Field(), "avg": Field()}

def cal1(value):
  data["num"].value = data["num"].value + 1
  data["cum"].value = data["cum"].value + value
  data["avg"].value = data["cum"].value / data["num"].value

cal1(1)   ...

def makeCal2():
  field_num = data["num"]
  field_cum = data["cum"]
  field_avg = data["avg"]
  def cal(value):
    field_num.value = field_num.value + 1
    field_cum.value = field_cum.value + value
    field_avg.value = field_cum.value / field_num.value
  return cal

cal = makeCal2()

cal(1)   ...
```

Begin With Experiment

https://github.com/PiperLiu/talks/tree/master/assets/20220812_closure

```
➜  ~ python3 --version
Python 3.8.9
➜  ~ ~/Github/downloads/pypy-c-jit-105934-1b027cda9f26-macos_arm64/bin/pypy3.8 --version
Python 3.8.13 (1b027cda9f26605e3acc92009338eefbc7300418, Aug 07 2022, 10:04:36)
[PyPy 7.3.10-alpha0 with GCC Apple LLVM 13.1.6 (clang-1316.0.21.2.5)]
➜  ~ g++ -v
Apple clang version 13.1.6 (clang-1316.0.21.2.5)
Target: arm64-apple-darwin21.6.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
➜  ~ node --version
v16.16.0
➜  ~ go version
go version go1.19 darwin/arm64

➜  ~ # just make all!
➜  ~ make all
```

Data

```
NAMES = 5
LENGTH = 61
RULES = 3
```


```
[
    {
        "name1": "AYS1azSoNDFkfQmARkPe5oy61Q3MgThDhxtpa8HEHKOF4achFETVdwUpD1SdO",
        "name2": "vB4l3N4zmH3vpzGN84msceIdHo7fW5isUnriUkGXPTmvkaSaLsGURoWgxoYk4",
        "name3": "AYS1azSoNDFkfQmARkPe5oy61Q3MgThDhxtpa8HEHKOF4achFETVdwUpD1SdO",
        "rule": "-"
    },
    {
        "name1": "vB4l3N4zmH3vpzGN84msceIdHo7fW5isUnriUkGXPTmvkaSaLsGURoWgxoYk4",
        "name2": "RYyOk9a0GBOGMNoIg5IuObLOxlqE0fVs4t32Zc6HLQCyWPIhBVW4BVetWo7gs",
        "name3": "u7eYG8odMoentoaBOg0R7pBpkI456Eux0umSgxLZGoTbnfthtboIkPm5Cvfg1",
        "rule": "|"
    },
    {
        "name1": "RYyOk9a0GBOGMNoIg5IuObLOxlqE0fVs4t32Zc6HLQCyWPIhBVW4BVetWo7gs",
        "name2": "AYS1azSoNDFkfQmARkPe5oy61Q3MgThDhxtpa8HEHKOF4achFETVdwUpD1SdO",
        "name3": "AYS1azSoNDFkfQmARkPe5oy61Q3MgThDhxtpa8HEHKOF4achFETVdwUpD1SdO",
        "rule": "+"
    }
]
```

```python
# 0
class Calculator:
    def __init__(self, rules) -> None:  ...
    def cal(self):
        for rule in self.rules:
            name1, name2, name3, rule = rule['name1'], rule['name2'], rule['name3'], rule['rule']
            if rule == '*':
                self.values[name3] = self.values[name1] * self.values[name2]
            elif rule == '+':  ...

# 1
class Calculator:
    def __init__(self, rules) -> None:  ...
    def cal(self):
        for rule in self.rules:
            name1, name2, name3, rule = rule['name1'], rule['name2'], rule['name3'], rule['rule']
            if rule == '*':
                def func():
                    self.values[name3] = self.values[name1] * self.values[name2]
                self.funcs.append(func)
            elif rule == '+':  ...
    def cal(self):
        for func in self.funcs:
            func()
```

```
# 0
class Calculator:
    def __init__(self, rules) -> None:  ...
    def cal(self):
        for rule in self.rules:
            name1, name2, name3, rule = rule['name1'], rule['name2'], rule['name3'], rule['rule']
            if rule == '*':
                self.values[name3] = self.values[name1] * self.values[name2]
            elif rule == '+':  ...

# 2
class Calculator:
    def __init__(self, rules) -> None:  ...
    def cal(self):
        for rule in self.rules:
            name1, name2, name3, rule = rule['name1'], rule['name2'], rule['name3'], rule['rule']
            field1, field2, field3 = self.values[name1], self.values[name2], self.values[name3]
            if rule == '*':
                def func():
                    field3.value = field1.value * field2.value
                self.funcs.append(func)
            elif rule == '+':  ...
    def cal(self):
        for func in self.funcs:
            func()
```

```python
# 0
class Calculator:
    def __init__(self, rules) -> None:  ...
    def cal(self):
        for rule in self.rules:
            name1, name2, name3, rule = rule['name1'], rule['name2'], rule['name3'], rule['rule']
            if rule == '*':
                self.values[name3] = self.values[name1] * self.values[name2]
            elif rule == '+':  ...

# 2
class Calculator:
    def __init__(self, rules) -> None:  ...
    def cal(self):
        for rule in self.rules:
            name1, name2, name3, rule = rule['name1'], rule['name2'], rule['name3'], rule['rule']
            field1, field2, field3 = self.values[name1], self.values[name2], self.values[name3]
            if rule == '*':
                def func():
                    field3.value = field1.value * field2.value
                self.funcs.append(func)
            elif rule == '+':  ...
    def cal(self):
        for func in self.funcs:
            func()
```

```python
# 0
class Calculator:
    def __init__(self, rules) -> None:  ...
    def cal(self):
        for rule in self.rules:
            name1, name2, name3, rule = rule['name1'], rule['name2'], rule['name3'], rule['rule']
            if rule == '*':
                self.values[name3] = self.values[name1] * self.values[name2]
            elif rule == '+':  ...

# 2
class Calculator:
    def __init__(self, rules) -> None:  ...
    def cal(self):
        for rule in self.rules:
            name1, name2, name3, rule = rule['name1'], rule['name2'], rule['name3'], rule['rule']
            field1, field2, field3 = self.values[name1], self.values[name2], self.values[name3]
            if rule == '*':
                def makefunc(field1, field2, field3):
                    def func():
                        field3.value = field1.value * field2.value
                    return func
                self.funcs.append(makefunc(field1, field2, field3))
            elif rule == '+':  ...
    def cal(self):
        for func in self.funcs:
            func()
```

results

NAMES = 1500, LENGTH = 100, RULES = 500000

`CPython`	0.224: 0.156, 0.155, 0.157	1.218: 0.110, 0.119, 0.114	1.027: 0.085, 0.082, 0.084
`PyPy`	0.228: 0.147, 0.146, 0.151	0.384: 0.252, 0.168, 0.171	0.440: 0.109, 0.063, 0.054
`C++`	3.491: 1.485, 1.514, 1.517	3.908: 1.472, 1.451, 1.453	5.109: 0.014, 0.013. 0.013
`GoLang`	0.053: 0.053, 0.052, 0.053	0.074: 0.052, 0.051, 0.050	0.115: 0.004, 0.004, 0.004
`JavaScript`	0.36: 0.035, 0.030, 0.026	0.406: 0.037, 0.036, 0.028	50.098: 0.019, 0.012, 0.008

Conclusion

- Unless the computation is very intensive and recompilation is not very convenient, it is better to write it as hard code.

Thanks