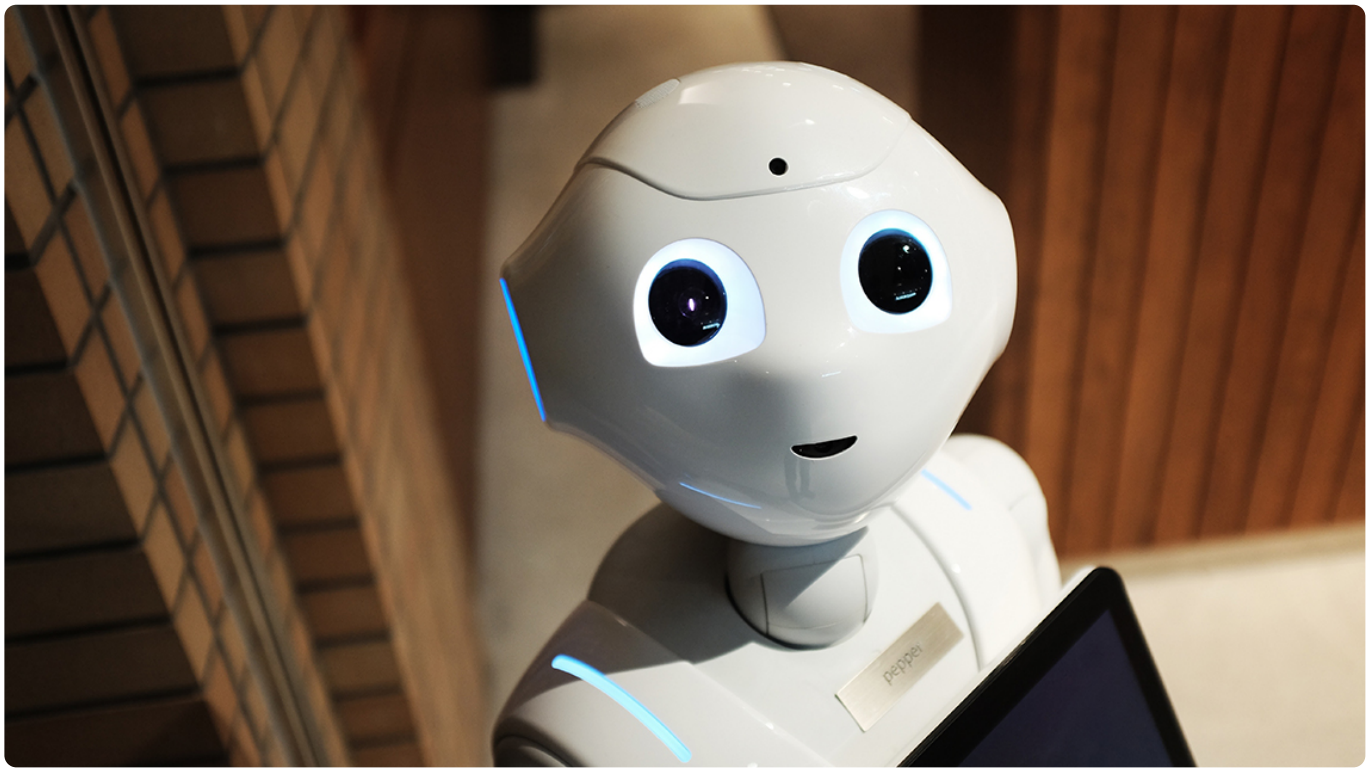
17 | 函数式编程：一种越来越流行的编程范式

2020-01-03 吴咏炜



讲述：吴咏炜
时长 13:13 大小 12.12M

你好，我是吴咏炜。

上一讲我们初步介绍了函数对象和 lambda 表达式，今天我们来讲讲它们的主要用途——函数式编程。

一个小例子

按惯例，我们还是从一个例子开始。想一下，如果给定一组文件名，要求数一下文件里的总文本行数，你会怎么做？

我们先规定一下函数的原型：

复制代码

```
int count_lines(const char** begin,
                const char** end);
```

也就是说，我们期待接受两个 C 字符串的迭代器，用来遍历所有的文件名；返回值代表文件中的总行数。

要测试行为是否正常，我们需要一个很小的 `main` 函数：

复制代码

```
int main(int argc,
         const char** argv)
{
  int total_lines = count_lines(
    argv + 1, argv + argc);
  cout << "Total lines: "
       << total_lines << endl;
}
```

最传统的命令式编程大概会这样写代码：

复制代码

```
int count_file(const char* name)
{
  int count = 0;
  ifstream ifs(name);
  string line;
  for (;;) {
    getline(ifs, line);
    if (!ifs) {
      break;
    }
    ++count;
  }
  return count;
}

int count_lines(const char** begin,
                const char** end)
{
  int count = 0;
  for (; begin != end; ++begin) {
    count += count_file(*begin);
```

```
  }
  return count;
}
```

我们马上可以做一个简单的“说明式”改造。用 `istream_line_reader` 可以简化 `count_file` 成：

复制代码

```
int count_file(const char* name)
{
  int count = 0;
  ifstream ifs(name);
  for (auto&& line :
       istream_line_reader(ifs)) {
    ++count;
  }
  return count;
}
```

在这儿，要请你停一下，想一想如何进一步优化这个代码。然后再继续进行往下看。

如果我们使用之前已经出场过的两个函数，`transform` [1] 和 `accumulate` [2]，代码可以进一步简化为：

复制代码

```
int count_file(const char* name)
{
  ifstream ifs(name);
  istream_line_reader reader(ifs);
  return distance(reader.begin(),
                  reader.end());
}

int count_lines(const char** begin,
                const char** end)
{
  vector<int> count(end - begin);
  transform(begin, end,
            count.begin(),
            count_file);
  return accumulate(
```

```
17      count.begin(), count.end(),
18      0);
19  }
```

这个就是一个非常函数式风格的结果了。上面这个处理方式恰恰就是 map-reduce。`transform` 对应 map，`accumulate` 对应 reduce。而检查有多少行文本，也成了代表文件头尾两个迭代器之间的“距离”（distance）。

函数式编程的特点

在我们的代码里不那么明显的一点是，函数式编程期望函数的行为像数学上的函数，而非一个计算机上的子程序。这样的函数一般被称为纯函数（pure function），要点在于：

> 会影响函数结果的只是函数的参数，没有对环境的依赖
>
> 返回的结果就是函数执行的唯一后果，不产生对环境的其他影响

这样的代码的最大好处是易于理解和易于推理，在很多情况下也会使代码更简单。在我们上面的代码里，`count_file` 和 `accumulate` 基本上可以看做是纯函数（虽然前者实际上有着对文件系统的依赖），但 `transform` 不行，因为它改变了某个参数，而不是返回一个结果。下一讲我们会看到，这会影响代码的组合性。

我们的代码中也体现了其他一些函数式编程的特点：

> 函数就像普通的对象一样被传递、使用和返回。
>
> 代码为说明式而非命令式。在熟悉函数式编程的基本范式后，你会发现说明式代码的可读性通常比命令式要高，代码还短。
>
> 一般不鼓励（甚至完全不使用）可变量。上面代码里只有 `count` 的内容在执行过程中被修改了，而且这种修改实际是 `transform` 接口带来的。如果接口像 [第 13 讲] 展示的 `fmap` 函数一样返回一个容器的话，就可以连这个问题都消除了。（C++ 毕竟不是一门函数式编程语言，对灵活性的追求压倒了其他考虑。）

高阶函数

既然函数（对象）可以被传递、使用和返回，自然就有函数会接受函数作为参数或者把函数作为返回值，这样的函数就被称为高阶函数。我们现在已经见过不少高阶函数了，如：

sort

transform

accumulate

fmap

adder

事实上，C++ 里以 algorithm（算法）[3] 名义提供的很多函数都是高阶函数。

许多高阶函数在函数式编程中已成为基本的惯用法，在不同语言中都会出现，虽然可能是以不同的名字。我们在此介绍非常常见的三个，map（映射）、reduce（归并）和 filter（过滤）。

Map 在 C++ 中的直接映射是 `transform`（在 <algorithm> 头文件中提供）。它所做的事情也是数学上的映射，把一个范围里的对象转换成相同数量的另外一些对象。这个函数的基本实现非常简单，但这是一种强大的抽象，在很多场合都用得上。

Reduce 在 C++ 中的直接映射是 `accumulate`（在 <numeric> 头文件中提供）。它的功能是在指定的范围里，使用给定的初值和函数对象，从左到右对数值进行归并。在不提供函数对象作为第四个参数时，功能上相当于默认提供了加法函数对象，这时相当于做累加；提供了其他函数对象时，那当然就是使用该函数对象进行归并了。

Filter 的功能是进行过滤，筛选出符合条件的成员。它在当前 C++（C++20 之前）里的映射可以认为有两个：`copy_if` 和 `partition`。这是因为在 C++20 带来 ranges 之前，在 C++ 里实现惰性求值不太方便。上面说的两个函数里，`copy_if` 是把满足条件的元素拷贝到另外一个迭代器里；`partition` 则是根据过滤条件来对范围里的元素进行分组，把满足条件的放在返回值迭代器的前面。另外，`remove_if` 也有点相近，通常用于删除满足条件的元素。它确保把不满足条件的元素放在返回值迭代器的前面（但不保证满足条件的元素在函数返回后一定存在），然后你一般需要使用容器的 `erase` 成员函数来将待删除的元素真正删除。

命令式编程和说明式编程

传统上 C++ 属于命令式编程。命令式编程里，代码会描述程序的具体执行步骤。好处是代码显得比较直截了当；缺点就是容易让人只见树木、不见森林，只能看到代码啰嗦地怎么做（how），而不是做什么（what），更不用说为什么（why）了。

说明式编程则相反。以数据库查询语言 SQL 为例，SQL 描述的是类似于下面的操作：你想从什么地方（from）选择（select）满足什么条件（where）的什么数据，并可选指定排序（order by）或分组（group by）条件。你不需要告诉数据库引擎具体该如何去执行这个操作。事实上，在选择查询策略上，大部分数据库用户都不及数据库引擎“聪明”；正如大部分开发者在写出优化汇编代码上也不及编译器聪明一样。

这并不是说说明式编程一定就优于命令式编程。事实上，对于很多算法，命令式才是最自然的实现。以快速排序为例，很多地方在讲到函数式编程时会给出下面这个 Haskell（一种纯函数式的编程语言）的例子来说明函数式编程的简洁性：

复制代码

```
quicksort []     = []
quicksort (p:xs) = (quicksort left)
          ++ [p] ++ (quicksort right)
   where
     left  = filter (< p) xs
     right = filter (>= p) xs
```

这段代码简洁性确实没话说，但问题是，上面的代码的性能其实非常糟糕。真正接近 C++ 性能的快速排序，在 Haskell 里写出来一点不优雅，反而更丑陋 [4]。

所以，我个人认为，说明式编程跟命令式编程可以结合起来产生既优雅又高效的代码。对于从命令式编程成长起来的大部分程序员，我的建议是：

> 写表意的代码，不要过于专注性能而让代码难以维护——记住高德纳的名言：“过早优化是万恶之源。”
>
> 使用有意义的变量，但尽量不要去修改变量内容——变量的修改非常容易导致程序员的思维错误。

类似地，尽量使用没有副作用的函数，并让你写的代码也尽量没有副作用，用返回值来代表状态的变化——没有副作用的代码更容易推理，更不容易出错。

代码的隐式依赖越少越好，尤其是不要使用全局变量——隐式依赖会让代码里的错误难以排查，也会让代码更难以测试。

使用知名的高级编程结构，如基于范围的 for 循环、映射、归并、过滤——这可以让你的代码更简洁，更易于推理，并减少类似下标越界这种低级错误的可能性。

这些跟函数式编程有什么关系呢？——这些差不多都是来自函数式编程的最佳实践。学习函数式编程，也是为了更好地体会如何从这些地方入手，写出易读而又高性能的代码。

不可变性和并发

在多核的时代里，函数式编程比以前更受青睐，一个重要的原因是函数式编程对并行并发天然友好。影响多核性能的一个重要因素是数据的竞争条件——由于共享内存数据需要加锁带来的延迟。函数式编程强调不可变性（immutability）、无副作用，天然就适合并发。更妙的是，如果你使用高层抽象的话，有时可以轻轻松松“免费”得到性能提升。

拿我们这一讲开头的例子来说，对代码做下面的改造，启用 C++17 的并行执行策略 [5]，就能自动获得在多核环境下的性能提升：

复制代码

```
int count_lines(const char** begin,
                const char** end)
{
  vector<int> count(end - begin);
  transform(execution::par,
            begin, end,
            count.begin(),
            count_file);
  return reduce(
    execution::par,
    count.begin(), count.end());
}
```

我们可以看到，两个高阶函数的调用中都加入了 `execution::par`，来启动自动并行计算。要注意的是，我把 `accumulate` 换成了 `reduce` [6]，原因是前者已经定义成从左到

右的归并，无法并行。`reduce` 则不同，初始值可以省略，操作上没有规定顺序，并反过来要求对元素的归并操作满足交换律和结合率（加法当然是满足的），即：

$$A \otimes B = B \otimes A$$
$$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$

当然，在这个例子里，一般我们不会有海量文件，即使有海量文件，并行读取性能一般也不会快于顺序读取，所以意义并不是很大。下面这个简单的例子展示了并行 `reduce` 的威力：

复制代码

```
#include <chrono>
#include <execution>
#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

int main()
{
  vector<double> v(10000000, 0.0625);

  {
    auto t1 = chrono::
      high_resolution_clock::now();
    double result = accumulate(
      v.begin(), v.end(), 0.0);
    auto t2 = chrono::
      high_resolution_clock::now();
    chrono::duration<double, milli>
      ms = t2 - t1;
    cout << "accumulate: result "
         << result << " took "
         << ms.count() << " ms\n";
  }

  {
    auto t1 = chrono::
      high_resolution_clock::now();
    double result =
      reduce(execution::par,
             v.begin(), v.end());
    auto t2 = chrono::
```

```
34        high_resolution_clock::now();
35      chrono::duration<double, milli>
36        ms = t2 - t1;
37      cout << "reduce:     result "
38           << result << " took "
39           << ms.count() << " ms\n";
40    }
41  }
```

在我的电脑（Core i7 四核八线程）上的某次执行结果是：

```
accumulate: result 625000 took 26.122 ms
reduce: result 625000 took 4.485 ms
```

执行策略还比较新，还没有被所有编译器支持。我目前测试下来，MSVC 没有问题，Clang 不行，GCC 需要外部库 TBB（Threading Building Blocks）[7] 的帮助。我上面是用 GCC 编译的，命令行是：

```
g++-9 -std=c++17 -O3 test.cpp -ltbb
```

Y 组合子

限于篇幅，这一讲我们只是很初浅地探讨了函数式编程。对于 C++ 的函数式编程的深入探讨是有整本书的（见参考资料 [8]），而今天讲的内容在书的最前面几章就覆盖完了。在后面，我们还会探讨部分的函数式编程话题；今天我们只再讨论一个有点有趣、也有点烧脑的话题，Y 组合子 [9]。第一次阅读的时候，如果觉得困难，可以跳过这一部分。

不过，我并不打算讨论 Haskell Curry 使用的 Y 组合子定义——这个比较复杂，需要写一篇完整的文章来讨论（[10]），而且在 C++ 中的实用性非常弱。我们只看它解决的问题：如何在 lambda 表达式中表现递归。

回想一下我们用过的阶乘的递归定义：

复制代码

```
1  int factorial(int n)
2  {
3    if (n == 0) {
4      return 1;
```

```
  } else {
    return n * factorial(n - 1);
  }
}
```

注意里面用到了递归，所以你要把它写成 lambda 表达式是有点困难的：

复制代码

```
auto factorial = [](int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * ???(n - 1);
  }
}
```

下面我们讨论使用 Y 组合子的解决方案。

我们首先需要一个特殊的高阶函数，定义为：

$$y(f) = f(y(f))$$

显然，这个定义有点奇怪。事实上，它是会导致无限展开的——而它的威力也在于无限展开。我们也因此必须使用惰性求值的方式才能使用这个定义。

然后，我们定义阶乘为：

$$\text{fact}(n) = \text{If IsZero}(n) \text{ then } 1 \text{ else } n \times \text{fact}(n-1)$$

假设 fact 可以表示成 $y(F)$，那我们可以做下面的变形：

$$\begin{aligned} y(F)(n) &= \text{If IsZero}(n) \text{ then } 1 \text{ else } n \times y(F)(n-1) \\ F(y(F))(n) &= \text{If IsZero}(n) \text{ then } 1 \text{ else } n \times y(F)(n-1) \end{aligned}$$

再把 $y(F)$ 替换成 f，我们从上面的第二个式子得到：

$$F(f)(n) = \text{If IsZero}(n) \text{ then } 1 \text{ else } n \times f(n-1)$$

我们得到了 F 的定义，也就自然得到了 fact 的定义。而且，这个定义是可以用 C++ 表达出来的。下面是完整的代码实现：

复制代码

```
#include <functional>
#include <iostream>
#include <type_traits>
#include <utility>

using namespace std;

// Y combinator as presented by Yegor Derevenets in P0200R0
// <url:http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0200r0.html>
template <class Fun>
class y_combinator_result {
  Fun fun_;
public:
  template <class T>
  explicit y_combinator_result(
    T&& fun)
    : fun_(std::forward<T>(fun))
  {
  }

  template <class... Args>
  decltype(auto)
  operator()(Args&&... args)
  {
    // y(f) = f(y(f))
    return fun_(
      std::ref(*this),
      std::forward<Args>(args)...);
  }
};

template <class Fun>
decltype(auto)
y_combinator(Fun&& fun)
{
  return y_combinator_result<
    std::decay_t<Fun>>(
    std::forward<Fun>(fun));
```

```
}

int main()
{
  //  上面的那个  F
  auto almost_fact =
    [](auto f, int n) -> int {
    if (n == 0)
      return 1;
    else
      return n * f(n - 1);
  };
  // fact = y(F)
  auto fact =
    y_combinator(almost_fact);
  cout << fact(10) << endl;
}
```

这一节不影响后面的内容，看不懂的可以暂时略过。😠

内容小结

本讲我们对函数式编程进行了一个入门式的介绍，希望你对函数式编程的特点、优缺点有了一个初步的了解。然后，我快速讨论了一个会烧脑的话题，Y 组合子，让你对函数式编程的威力和难度也有所了解。

课后思考

想一想，你如何可以实现一个惰性的过滤器？一个惰性的过滤器应当让下面的代码通过编译，并且不会占用跟数据集大小相关的额外空间：

复制代码

```
#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

// filter_view  的定义

int main()
{
  vector v{1, 2, 3, 4, 5};
  auto&& fv = filter_view(
```

```
    v.begin(), v.end(), [](int x) {
      return x % 2 == 0;
    });
  cout << accumulate(fv.begin(),
                     fv.end(), 0)
       << endl;
}
```

结果输出应该是 `6`。

提示：参考 `istream_line_reader` 的实现。

告诉我你是否成功了，或者你遇到了什么样的特别困难。

参考资料

[1] cppreference.com, “std::transform”.
https://en.cppreference.com/w/cpp/algorithm/transform

[1a] cppreference.com, “std::transform”.
https://zh.cppreference.com/w/cpp/algorithm/transform

[2] cppreference.com, “std::accumulate”.
https://en.cppreference.com/w/cpp/algorithm/accumulate

[2a] cppreference.com, “std::accumulate”.
https://zh.cppreference.com/w/cpp/algorithm/accumulate

[3] cppreference.com, “Standard library header <algorithm>”.
https://en.cppreference.com/w/cpp/header/algorithm

[3a] cppreference.com, “标准库头文件 <algorithm>”.
https://zh.cppreference.com/w/cpp/header/algorithm

[4] 袁英杰, “Immutability: The Dark Side”.
https://www.jianshu.com/p/13cd4c650125

[5] cppreference.com, “Standard library header <execution>”. https://en.cppreference.com/w/cpp/header/execution

[5a] cppreference.com, “标准库头文件 <execution>”. https://zh.cppreference.com/w/cpp/header/execution

[6] cppreference.com, “std::reduce”. https://en.cppreference.com/w/cpp/algorithm/reduce

[6a] cppreference.com, “std::reduce”. https://zh.cppreference.com/w/cpp/algorithm/reduce

[7] Intel, tbb. https://github.com/intel/tbb

[8] Ivan Čukić, *Functional Programming in C++*. Manning, 2019, https://www.manning.com/books/functional-programming-in-c-plus-plus

[9] Wikipedia, “Fixed-point combinator”. https://en.wikipedia.org/wiki/Fixed-point_combinator

[10] 吴咏炜, “*Y* Combinator and C++”. https://yongweiwu.wordpress.com/2014/12/14/y-combinator-and-cplusplus/

上一篇　16 | 函数对象和lambda：进入函数式编程

下一篇　18 | 应用可变模板和tuple的编译期技巧

精选留言 (7)

写留言

罗乾林
2020-01-03

参考 istream_line_reader 实现的，望老师斧正

```
template<typename _InIt, typename _Fun>
class filter_view {
public:...
```

展开

作者回复: OK，没啥大问题。

代码风格要稍微说明一下。你似乎是模拟了库代码的风格，这还是有点风险的。在一般的用户代码里，不应该出现双下划线打头、或者下划线加大写字母打头的标识符——这是给系统保留的。

详见：

https://zh.cppreference.com/w/cpp/language/identifiers

2 2

王小品
2020-01-07

Y-Combinator被你说高深了。递归就是自己调用自己。lamda表达式想递归，困难在于不知道自己的函数名，怎么办？调用不了自己，难道还调用不了别人。所以lamda表达式调用了Y-Combinator去间接调用自己，而Y-Combinator只不过：一，记录lamda表达式；二，转调lamda表达式。这就好比普京受制于连任时间限制，如果想继续连任，则找个代言人Y-Combinator继任。代言人的唯一作用就是到期传位普京。

展开

作者回复: 哈哈，有意思的比喻。以前学过函数式编程?

1

三味
2020-01-10

关于这个y_combinator, 代码中有很多我不太理解的：
1. 为什么lambda函数作为参数传递的时候，都是Fun&&？我实际测试直接用Fun，对于当前代码没什么问题，直接用Fun不好么？；
2. 关于forward，题目中所有带有forward的地方，我都直接替换为不带forward的方式，编译和运行也没有什么问题。这里的forward作用究竟是怎样的?

展开

作者回复: 性能。不用引用，对于比较重（比如，有较多按值捕获）的函数对象，拷贝的开销就比较大了。用 Fun&& 和 std::forward，就是要把拷贝尽可能转变成移动。

三味
2020-01-09

这一节是我耗时最长的一节。。因为来回翻阅迭代器那一节。。
写了个std::copy(fv.begin(), fv.end(), std::ostream_iterator<int>(std::cout, " "));结果编译失败。。查了半天。。因为没定义pointer和reference类型。。
还有就是，之前的章节看得不仔细，看别人答案觉得好奇怪，为什么一上来要++(*this)。。后来对比自己的实现，我是按照forward_iterator_tag来定义的，所以写法有些不...

展开

作者回复: 功能看起来没啥问题。嗯，太长，所以贴到别的留言下去了，还是有点怪怪的。

还是要提醒，在一般的用户代码里，不应该出现双下划线打头、或者下划线加大写字母打头的标识符——这是给系统保留的。

详见：

https://zh.cppreference.com/w/cpp/language/identifiers

じJRisenづジ
2020-01-07

#include <iostream>
#include <chrono>
#include <execution>
#include <numeric>
#include <vector> 老师怎么学习库知识？ 提点思路

展开

作者回复: 跟学外语一样，基本诀窍就是多读多写。文档现在都很齐全的，但除了极少数天才式的人物，看了不用就会忘掉吧。而且，没真正用过，碰到一些坑，看了的理解都不一定正确。

另外就是看书了。书比文档更有体系性，更适合完整的学习，需要投入整块的时间。标准库的书，应该就是 Nicolai Josuttis 的那本 The C++ Standard Library 第二版了。中文版刚查了一下，侯捷译，一千多页。

1

廖熊猫
2020-01-03

Y-Combinator主要用到了一个不动点理论，刘未鹏老师的《康托尔、哥德尔、图灵——永恒的金色对角线》这篇文章里面说的相对详细一些。玩过一段函数式...就只有haskell那段代码看懂了😂

作者回复: Y Combinator 只是好玩展示一下，刺激一下大家的好奇心。要进一步了解，是需要看参考资料，或者其他中英文资料的。你说的这篇我之前没看过，内容也不错。

hello world

2020-01-03

请问老师，map和reduce.那是最新的语句吗？还是有第三方库？那个TBB？

> 作者回复: map-reduce 是一种方法，已经有很久了。在 C++ 里的直接对应是 transform 和 accumulate。TBB 见参考资料 [7]。