

如何漂亮地写代码

2026-09-23

有人说，AI 时代，没人自己写代码。如果有人说自己必须古法写代码才能解决问题，那么一定是他的模型用得不够好。他肯定是少装了什么 Skill 或者没给 AI 充钱。不错，我承认现在没几个人写代码；但是拿着 Vibe Coding 出来的 AI Slop 往别人的项目乱发 PR，或者往小红书上抱怨自己的“产品”无人问津，就是“程序员”自己的问题了。我们需要学习如何漂亮地写代码，至少掌握鉴赏漂亮代码的能力。

让编程语言有自然语言般的表现力

编程语言是一种语言。不论是自然语言还是形式语言，都有优美和糟糕之分。我们会对优美的文章大加赞叹，而排斥粗鄙之语。我们会对严谨的数学推理和优美的数学工具感兴趣，而对诡辩不感兴趣甚至感到恶心。编程语言也是如此。我们喜爱易读、如自然语言般流畅的代码风格，而拒绝 Review 那些耦合、晦涩、`i++ + ++i` 的屎山。

给计算机安排工作——声明式编程

现代语言给了我们丰富的抽象工具，供我们写出清晰易懂且无需担心性能问题的代码。在此基础上，我们可以弱化实现细节，通过告诉计算机“我想要什么”而不是“应该怎么做”来写出可读性高的代码。

例如，我们在写 SQL 语言时会写出这样的语句：

```
SELECT name FROM users WHERE age ≥ 18;
```

阅读这个句子，我们发现这就是一句语法通顺的英文：

Select the value of `name` from each row in the table `users` where that row's `age` is greater than or equal to `18`.

我们没有像监工一样，命令计算机：

```
for row ← users {
  if row.age ≥ 18 {
    tell me row.name
  }
}
```

而是像向一位绝顶聪明的助手“许愿”：

把 `users` 这张表里面 `age` 大于等于 18 的行挑出来，取出 `name` 给我。

这更像数学里的描述式集合，简洁、纯粹、优雅：

$$\text{names} = \{\text{row.name} \mid \text{row} \in \text{users} \wedge \text{row.name} \geq 18\}$$

再比如这段 Python 代码：

```
l = [i * 2 for i in range(10)]
```

他向计算机声明：我要把 `range(10)` 中每一个 `i` 乘上 2 放到一个列表里。换做以前，我们会写：

```
l = []
for i in range(10):
    l.append(i * 2)
```

例如，我们要将装有学生成绩的数组中及格线以上的学生过滤出来，然后按从分数低到高排列，我们该怎么做？

C 语言是一个古老但仍在系统和高性能编程发挥基石作用的语言。自从 1972 年诞生以来，它就几乎没有添加过新的语法，也自然没有现代语言赖以生存的现代特性。如下所示，这是上述问题的一个典型的 C 语言实现。

```
#include <stdio.h>
#include <stdlib.h>
#define PASS 60

typedef struct {
    char *name;
    unsigned int score;
} Student;

int cmp(const void *a, const void *b) {
    unsigned int score_a = ((const Student *)a)→score;
    unsigned int score_b = ((const Student *)b)→score;

    if (score_a < score_b) return -1;
    if (score_a > score_b) return 1;
    return 0;
}

void filter_and_sort(const Student *src, Student *des, size_t size) {
    size_t index = 0;
    for (size_t i = 0; i < size; i++) {
        if (src[i].score < PASS) continue;
        des[index] = src[i]; // shallow copy
        index++;
    }
    qsort(des, index, sizeof(Student), cmp);
}
```

由于 C++ 可认为是 C 语言的超集（实际上并不准确，因为 C++ 标准不允许 VLA 和一系列 C 特性），于是有人写出这样的 C++ 代码：

```
++
#include <iostream>
#include <vector>
#include <range>
using namespace std;

constexpr int PASS = 60;

struct Student {
    char *name;
    unsigned int score;
} ;

std::vector filter_and_sort(const std::vector& src) {
    std::vector des;
    for (const auto& student : src) {
        if (student.score < PASS) continue;
        des.push_back(student);
    }
}
```

```

    }

    std::ranges::sort(des, [](const auto &a, const auto &b) {
        return a.score < b.score;
    });

    return des;
}

```

我们通过阅读代码并分析逻辑，发现这段实现几乎就是 C 的改写。从可读性上看，我们能不能做得更好？

```

++
#include <vector>
#include <ranges>
using namespace std;
using namespace std::views;
using namespace std::ranges;

constexpr int PASS = 60;

struct Student {
    char *name;
    unsigned int score;
};

std::vector filter_and_sort(const std::vector& src) {
    auto des = src | filter([](const auto &i) { return i.score ≥ PASS; })
                | to<vector>();
    std::ranges::sort(des, std::greater{}, &Student::score);
    return des;
}

```

我们在语言层面直接描述 `filter` 和 `sort` 语义，而无需分析 `for` 语句块的逻辑。

让我们进一步用噪声更小的语言重写这段逻辑：

```

struct Student {
    name: String,
    score: usize,
}

const PASS: usize = 60;

fn filter_and_sort(src: Vec<Student>) → Vec<Student> {
    let mut des = src.into_iter()
        .filter(|s| s.score ≥ PASS)
        .collect();

    des.sort_by(|a, b| b.score.cmp(&a.score));
    des
}

```

我们发现，比起用复杂的流程叙述一个算法，让语言自己描述自己的功能更有可读性。

大胆抽象，交给编译器优化

有人说：让语言承担描述高级算法的职责，是否会拖慢运行速度？我认为，大部分情况是否定的。现代 C++ 和 Rust 遵循所谓零成本抽象原则，指的是不需要为没用到的东西付出代价；你用到的东西，你自己手写也不会比编译器生成的代码更高效。

在现代编译器看来，提供越多的信息，越能给编译器优化带来信心；反之，如果一门语言极其贴近硬件，程序员就不得不考虑编译器没考虑的部分，采用 hacky 手段来尽可能优化程序的性能。所谓“C 语言是最快的语言”显然并不完全成立，对内存掌握不好的程序员，以及视 AI 为神明，认为 AI 能够抹除一切语言区别的 vibe coder 来说更是如此。我们通过 memory aliasing 这个例子来详细说明。

```
void compute(int* input, int* output) {
    if (*input > 10) {
        *output = 1;
    }
    if (*input > 5) {
        *output *= 2;
    }
}
```

这个函数能被优化成下面这个函数吗？

```
void compute(int* input, int* output) {
    int cached_input = *input;
    if (cached_input > 10) {
        *output = 2;
    } else if (cached_input > 5) {
        *output *= 2;
    }
}
```

按照函数想要表达的逻辑来讲，这两个函数本质上做的是相同的事情。让我们把这两个函数编译成 x86-64 汇编：

```
gcc -S main.c -O2 -fno-stack-protector

compute1:
.LFB0:
.cfi_startproc
movl    (%rdi), %eax    ; 第一次读取 *input
cmpl    $10, %eax
jle     .L2
movl    $1, (%rsi)
movl    (%rdi), %eax    ; 第二次读取 *input
.L2:
cmpl    $5, %eax
jle     .L1
sall    (%rsi)
.L1:
ret
.cfi_endproc

compute2:
.LFB1:
.cfi_startproc
movl    (%rdi), %eax    ; 第一次读取 *input
cmpl    $10, %eax
```

```

jle .L6
movl $2, (%rsi)
ret                ; 发生了什么?
.L6:
cmpl $5, %eax
jle .L5
sall (%rsi)
.L5:
ret
.cfi_endproc

```

为什么第二个函数只读一次 `*input`，第一个函数必须读两次？如果在第一个函数加上 `restrict` 修饰符：

```

void compute1(int* restrict input, int* restrict output) {
    if (*input > 10) {
        *output = 1;
    }
    if (*input > 5) {
        *output *= 2;
    }
}

compute1:
.LFB0:
.cfi_startproc
movl (%rdi), %eax    ; 第一次读取 *input
cmpl $10, %eax
jg .L4
cmpl $5, %eax
jg .L6
ret
.p2align 4,,10
.p2align 3
.L4:
movl $2, %eax
movl %eax, (%rsi)
ret
.L6:
movl (%rsi), %eax
addl %eax, %eax
movl %eax, (%rsi)
ret
.cfi_endproc

```

我们发现 `*input` 只被读取了一次。

`restrict` 告诉编译器，这个修饰符修饰的变量不会被程序的其他部分改动，因此可以让编译器激进地优化。

注意看第一个函数的这一行：

```

void compute1(int* input, int* output) {
    if (*input > 10) {
        *output = 1;
    }
}

```

```

    if (*input > 5) {
        *output *= 2;
    }
}

```

这里修改了 `*output`，鉴于 `output` 和 `input` 可能指向同一块内存，编译器不敢贸然做一些例如把 `*input` 放到寄存器里之类的优化。思考：如果我们加上 `restrict` 关键字，并让 `input` 和 `output` 指向了同一块内存，会发生什么事情？

我们运行

```

#include <stdio.h>

int main() {
    int i = 11;
    compute1(&i, &i);
    printf("%d\n", i);
}

```

在 `-O2` 优化程度下，我们期望他输出 `1`，可是他输出了 `2`。但如果我们严格保证两个指针指向的内存不同，那么这个函数不会有任何计算问题。

可是不凑巧的是，普通 C 程序员并不会使用 `restrict` 来优化程序性能，AI 在没有提示词的情况下也不会主动写出 `restrict`，因此编译器难以优化这些 C 语言程序。

得益于 Rust 的“多读单写”原则，同一时间只可能出现某一个变量的唯一可变借用，所以当我们写出

```
fn compute(input: &i32, output: &mut i32);
```

这样的签名时，编译器立即能够知道，`input` 和 `output` 不可能指向同一块内存。指向同一块内存的两个可变借用只可能在 `unsafe` 代码中出现，编译器能够大大方方优化，不必顾及先前提到的 `memory aliasing` 问题。

设计好接口，让你的代码变成文章

我们假设

- `a: Html`
- `b: Json`
- `c: Structure`
- `parse_html: Html → Result<Json, E>`
- `parse_json: Json → Result<Structure, E>`

```

if a.is_ok() {
    let a = a.unwrap();

    let b = parse_html(a);

    if b.is_ok() {
        let b = b.unwrap();

        let c = parse_json(b);

        if c.is_ok() {
            let c = c.unwrap();
            ...
        }
    }
}

```

```

    }
  }
}

```

这段代码显然看着很丑。

但如果我们充分利用所谓自函子范畴上的么半群，让 `Result` 的 `and_then()` 来解释代码逻辑，那么这段代码将变成

```

let result = a
    .and_then(parse_html)
    .and_then(parse_json);

```

我没解释，你看得懂吗？这就是好接口的魅力。

```

pub const fn and_then<U, F>(self, op: F) → Result<U, E>
    where
        F: [const] FnOnce(T) → Result<U, E> + [const] Destruct,
{
    match self {
        Ok(t) ⇒ op(t),
        Err(e) ⇒ Err(e),
    }
}

```

`and_then` 让 `Ok(t)` 传给闭包运行，`Err(e)` 直接保持原样，短路透传，正好符合 `and_then` 函数名语义。其他语言有 `>>=`、`flatMap`、`bind` 等叫法，个人感觉都没有 `and_then` 清晰。漂亮的代码靠清晰的语义取胜，而非靠所谓炫技语法糖。

代码不言自明——少写注释

很多人会把注释的覆盖率当作评判一个项目的代码质量的标准。且看下面一段代码：

```

int sum(int* array, size_t size) {
    ...
}

```

和这一段代码：

```

/** @param array 是一段
**
**
**/

```

不要打断读者的心流状态——少用无意义的中间变量

留下代码六尺巷