一、可读性的重要性

编程有很大一部分时间是在阅读代码，不仅要阅读自己的代码，而且要阅读别人的代码。因此，可读性良好的代码能够大大提高编程效率。

可读性良好的代码往往会让代码架构更好，因为程序员更愿意去修改这部分代码，而且也更容易修改。

只有在核心领域为了效率才可以放弃可读性，否则可读性是第一位。

二、用名字表达代码含义

一些比较有表达力的单词：

单词	可替代单词
send	deliver、dispatch、announce、distribute、route
find	search、extract、locate、recover
start	launch、create、begin、open
make	create、set up、build、generate、compose、add、new

使用 i、j、k 作为循环迭代器的名字过于简单，user_i、member_i 这种名字会更有表达力。因为循环层次越多，代码越难理解，有表达力的迭代器名字可读性会更高。

为名字添加形容词等信息能让名字更具有表达力，但是名字也会变长。名字长短的准则是：作用域越大，名字越长。因此只有在短作用域才能使用一些简单名字。

三、名字不能带来歧义

起完名字要思考一下别人会对这个名字有何解读，会不会误解了原本想表达的含义。

用 min、max 表示数量范围；用 first、last 表示访问空间的包含范围，begin、end 表示访问空间的排除范围，即 end 不包含尾部。

布尔相关的命名加上 is、can、should、has 等前缀。

四、良好的代码风格

适当的空行和缩进。

排列整齐的注释：

```
int a = 1;   // 注释
int b = 11;  // 注释
int c = 111; // 注释
```

语句顺序不能随意，比如与 html 表单相关联的变量的赋值应该和表单在 html 中的顺序一致；

把相关的代码按块组织起来放在一起。

五、编写注释

阅读代码首先会注意到注释，如果注释没太大作用，那么就会浪费代码阅读的时间。那些能直接看出含义的代码不需要写注释，特别是并不需要为每个方法都加上注释，比如那些简单的 getter 和 setter 方法，为这些方法写注释反而让代码可读性更差。

不能因为有注释就随便起个名字，而是争取起个好名字而不写注释。

可以用注释来记录采用当前解决办法的思考过程，从而让读者更容易理解代码。

注释用来提醒一些特殊情况。

用 TODO 等做标记：

标记	用法
TODO	待做
FIXME	待修复
HACH	粗糙的解决方案
XXX	危险！这里有重要的问题

六、如何编写注释

尽量简洁明了：

```
// The first String is student's name
// The Second Integer is student's score
Map<String, Integer> scoreMap = new HashMap<>();
```

```
// Student's name -> Student's score
Map<String, Integer> scoreMap = new HashMap<>();
```

添加测试用例来说明：

```
// ...
// Example: add(1, 2), return 3
int add(int x, int y) {
    return x + y;
}
```

在很复杂的函数调用中对每个参数标上名字：

```
int a = 1;
int b = 2;
int num = add(\* x = *\ a, \* y = *\ b);
```

使用专业名词来缩短概念上的解释，比如用设计模式名来说明代码。

七、提高控制流的可读性

条件表达式中，左侧是变量，右侧是常数。比如下面第一个语句正确：

```
if (len < 10)
if (10 > len)
```

if / else 条件语句，逻辑的处理顺序为：① 正逻辑；② 关键逻辑；③ 简单逻辑。

```
if (a == b) {
    // 正逻辑
} else{
    // 反逻辑
}
```

只有在逻辑简单的情况下使用 ? : 三目运算符来使代码更紧凑，否则应该拆分成 if / else；

do / while 的条件放在后面，不够简单明了，并且会有一些迷惑的地方，最好使用 while 来代替。

如果只有一个 goto 目标，那么 goto 尚且还能接受，但是过于复杂的 goto 会让代码可读性特别差，应该避免使用 goto。

在嵌套的循环中，用一些 return 语句往往能减少嵌套的层数。

八、拆分长表达式

长表达式的可读性很差，可以引入一些解释变量从而拆分表达式：

```
if line.split(':')[0].strip() == "root":
    ...
```

```
username = line.split(':')[0].strip()
if username == "root":
    ...
```

使用摩根定理简化一些逻辑表达式：

```
if (!a && !b) {
    ...
}
```

```
if (!(a || b)) {
    ...
}
```

九、变量与可读性

去除控制流变量 。在循环中通过使用 break 或者 return 可以减少控制流变量的使用。

```
boolean done = false;
while (/* condition */ && !done) {
    ...
    if ( ... ) {
        done = true;
        continue;
    }
}
```

```
while(/* condition */) {
    ...
    if ( ... ) {
        break;
    }
}
```

减小变量作用域 。作用域越小，越容易定位到变量所有使用的地方。

JavaScript 可以用闭包减小作用域。以下代码中 submit_form 是函数变量，submitted 变量控制函数不会被提交两次。第一个实现中 submitted 是全局变量，第二个实现把 submitted 放到匿名函数中，从而限制了起作用域范围。

```
submitted = false;
var submit_form = function(form_name) {
    if (submitted) {
        return;
    }
    submitted = true;
};
```

```
var submit_form = (function() {
    var submitted = false;
    return function(form_name) {
        if(submitted) {
            return;
        }
        submitted = true;
    }
}());  // () 使得外层匿名函数立即执行
```

JavaScript 中没有用 var 声明的变量都是全局变量，而全局变量很容易造成迷惑，因此应当总是用 var 来声明变量。

变量定义的位置应当离它使用的位置最近。

实例解析

在一个网页中有以下文本输入字段：

```
<input type = "text" id = "input1" value = "a">
<input type = "text" id = "input2" value = "b">
<input type = "text" id = "input3" value = "">
<input type = "text" id = "input4" value = "d">
```

现在要接受一个字符串并把它放到第一个空的 input 字段中，初始实现如下：

```
var setFirstEmptyInput = function(new_alue) {
    var found = false;
    var i = 1;
    var elem = document.getElementById('input' + i);
    while (elem != null) {
        if (elem.value === '') {
            found = true;
            break;
        }
        i++;
        elem = document.getElementById('input' + i);
    }
    if (found) elem.value = new_value;
    return elem;
}
```

以上实现有以下问题：

- found 可以去除；
- elem 作用域过大；
- 可以用 for 循环代替 while 循环；

```
var setFirstEmptyInput = function(new_value) {
    for (var i = 1; true; i++) {
        var elem = document.getElementById('input' + i);
        if (elem === null) {
            return null;
        }
        if (elem.value === '') {
            elem.value = new_value;
            return elem;
        }
    }
};
```

十、抽取函数

工程学就是把大问题拆分成小问题再把这些问题的解决方案放回一起。

首先应该明确一个函数的高层次目标，然后对于不是直接为了这个目标工作的代码，抽取出来放到独立的函数中。

介绍性的代码：

```
int findClostElement(int[] arr) {
    int clostIdx;
    int clostDist = Interger.MAX_VALUE;
    for (int i = 0; i < arr.length; i++) {
        int x = ...;
        int y = ...;
        int z = ...;
        int value = x * y * z;
        int dist = Math.sqrt(Math.pow(value, 2), Math.pow(arr[i], 2));
        if (dist < clostDist) {
            clostIdx = i;
            clostDist = value;
        }
    }
    return clostIdx;
}
```

以上代码中循环部分主要计算距离，这部分不属于代码高层次目标，高层次目标是寻找最小距离的值，因此可以把这部分代替提取到独立的函数中。这样做也带来一个额外的好处有：可以单独进行测试、可以快速找到程序错误并修改。

```
public int findClostElement(int[] arr) {
    int clostIdx;
    int clostDist = Interger.MAX_VALUE;
    for (int i = 0; i < arr.length; i++) {
        int dist = computDist(arr, i);
        if (dist < clostDist) {
            clostIdx = i;
            clostDist = value;
        }
    }
    return clostIdx;
}
```

并不是函数抽取的越多越好，如果抽取过多，在阅读代码的时候可能需要不断跳来跳去。只有在当前函数不需要去了解某一块代码细节而能够表达其内容时，把这块代码抽取成子函数才是好的。

函数抽取也用于减小代码的冗余。

十一、一次只做一件事

只做一件事的代码很容易让人知道其要做的事；

基本流程：列出代码所做的所有任务；把每个任务拆分到不同的函数，或者不同的段落。

十二、用自然语言表述代码

先用自然语言书写代码逻辑，也就是伪代码，然后再写代码，这样代码逻辑会更清晰。

十三、减少代码量

不要过度设计，编码过程会有很多变化，过度设计的内容到最后往往是无用的。

多用标准库实现。

参考资料

- Dustin, Boswell, Trevor, 等. 编写可读代码的艺术 [M]. 机械工业出版社, 2012.