编程的智慧

编程是一种创造性的工作，是一门艺术。精通任何一门艺术，都需要很多的练习和领悟，所以这里提出的“智慧”，并不是号称一天瘦十斤的减肥药，它并不能代替你自己的勤奋。然而由于软件行业喜欢标新立异，喜欢把简单的事情搞复杂，我希望这些文字能给迷惑中的人们指出一些正确的方向，让他们少走一些弯路，基本做到一分耕耘一分收获。

反复推敲代码

既然“天才是百分之一的灵感，百分之九十九的汗水”，那我先来谈谈这汗水的部分吧。有人问我，提高编程水平最有效的办法是什么？我想了很久，终于发现最有效的办法，其实是反反复复地修改和推敲代码。

在 IU 的时候，由于 Dan Friedman 的严格教导，我们以写出冗长复杂的代码为耻。如果你代码多写了几行，这老顽童就会大笑，说：“当年我解决这个问题，只写了 5 行代码，你回去再想想吧……” 当然，有时候他只是夸张一下，故意刺激你的，其实没有人能只用 5 行代码完成。然而这种提炼代码，减少冗余的习惯，却由此深入了我的骨髓。

有些人喜欢炫耀自己写了多少多少万行的代码，仿佛代码的数量是衡量编程水平的标准。然而，如果你总是匆匆写出代码，却从来不回头去推敲，修改和提炼，其实是不可能提高编程水平的。你会制造出越来越多平庸甚至糟糕的代码。在这种意义上，很多人所谓的“工作经验”，跟他代码的质量，其实不一定成正比。如果有几十年的工作经验，却从来不回头去提炼和反思自己的代码，那么他也许还不如一个只有一两年经验，却喜欢反复推敲，仔细领悟的人。

有位文豪说得好：“看一个作家的水平，不是看他发表了多少文字，而要看他的废纸篓里扔掉了多少。” 我觉得同样的理论适用于编程。好的程序员，他们删

掉的代码，比留下来的还要多很多。如果你看见一个人写了很多代码，却没有删掉多少，那他的代码一定有很多垃圾。

就像文学作品一样，代码是不可能一蹴而就的。灵感似乎总是零零星星，陆陆续续到来的。任何人都不可能一笔呵成，就算再厉害的程序员，也需要经过一段时间，才能发现最简单优雅的写法。有时候你反复提炼一段代码，觉得到了顶峰，没法再改进了，可是过了几个月再回头来看，又发现好多可以改进和简化的地方。这跟写文章一模一样，回头看几个月或者几年前写的东西，你总能发现一些改进。

所以如果反复提炼代码已经不再有进展，那么你可以暂时把它放下。过几个星期或者几个月再回头来看，也许就有焕然一新的灵感。这样反反复复很多次之后，你就积累起了灵感和智慧，从而能够在遇到新问题的时候直接朝正确，或者接近正确的方向前进。

写优雅的代码

人们都讨厌“面条代码”（spaghetti code），因为它就像面条一样绕来绕去，没法理清头绪。那么优雅的代码一般是什么形状的呢？经过多年的观察，我发现优雅的代码，在形状上有一些明显的特征。

如果我们忽略具体的内容，从大体结构上来看，优雅的代码看起来就像是一些整整齐齐，套在一起的盒子。如果跟整理房间做一个类比，就很容易理解。如果你把所有物品都丢在一个很大的抽屉里，那么它们就会全都混在一起。你就很难整理，很难迅速的找到需要的东西。但是如果你在抽屉里再放几个小盒子，把物品分门别类放进去，那么它们就不会到处乱跑，你就可以比较容易的找到和管理它们。

优雅的代码的另一个特征是，它的逻辑大体上看起来，是枝丫分明的树状结构（tree）。这是因为程序所做的几乎一切事情，都是信息的传递和分支。你可以把代码看成是一个电路，电流经过导线，分流或者汇合。如果你是这样思考

的，你的代码里就会比较少出现只有一个分支的 if 语句，它看起来就会像这个样子：

```
if (...) {
  if (...) {
    ...
  } else {
    ...
  }
} else if (...) {
  ...
} else {
  ...
}
```

注意到了吗？在我的代码里面，if 语句几乎总是有两个分支。它们有可能嵌套，有多层的缩进，而且 else 分支里面有可能出现少量重复的代码。然而这样的结构，逻辑却非常严密和清晰。在后面我会告诉你为什么 if 语句最好有两个分支。

写模块化的代码

有些人吵着闹着要让程序“模块化”，结果他们的做法是把代码分部到多个文件和目录里面，然后把这些目录或者文件叫做“module”。他们甚至把这些目录分放在不同的 VCS repo 里面。结果这样的作法并没有带来合作的流畅，而是带来了许多的麻烦。这是因为他们其实并不理解什么叫做“模块”，肤浅的把代码切割开来，分放在不同的位置，其实非但不能达到模块化的目的，而且制造了不必要的麻烦。

真正的模块化，并不是文本意义上的，而是逻辑意义上的。一个模块应该像一个电路芯片，它有定义良好的输入和输出。实际上一种很好的模块化方法早已经存在，它的名字叫做“函数”。每一个函数都有明确的输入（参数）和输出（返回值），同一个文件里可以包含多个函数，所以你其实根本不需要把代码分开在多个文件或者目录里面，同样可以完成代码的模块化。我可以把代码全都写在同一个文件里，却仍然是非常模块化的代码。

想要达到很好的模块化，你需要做到以下几点：

- 避免写太长的函数。如果发现函数太大了，就应该把它拆分成几个更小的。通常我写的函数长度都不超过 40 行。对比一下，一般笔记本电脑屏幕所能容纳的代码行数是 50 行。我可以一目了然的看见一个 40 行的函数，而不需要滚屏。只有 40 行而不是 50 行的原因是，我的眼球不转的话，最大的视角只看得到 40 行代码。

  如果我看代码不转眼球的话，我就能把整片代码完整的映射到我的视觉神经里，这样就算忽然闭上眼睛，我也能看得见这段代码。我发现闭上眼睛的时候，大脑能够更加有效地处理代码，你能想象这段代码可以变成什么其它的形状。40 行并不是一个很大的限制，因为函数里面比较复杂的部分，往往早就被我提取出去，做成了更小的函数，然后从原来的函数里面调用。

- 制造小的工具函数。如果你仔细观察代码，就会发现其实里面有很多的重复。这些常用的代码，不管它有多短，提取出去做成函数，都可能是会有好处的。有些帮助函数也许就只有两行，然而它们却能大大简化主要函数里面的逻辑。

  有些人不喜欢使用小的函数，因为他们想避免函数调用的开销，结果他们写出几百行之大的函数。这是一种过时的观念。现代的编译器都能自动的把小的函数内联（inline）到调用它的地方，所以根本不产生函数调用，也就不会产生任何多余的开销。

  同样的一些人，也爱使用宏（macro）来代替小函数，这也是一种过时的观念。在早期的 C 语言编译器里，只有宏是静态“内联”的，所以他们使用宏，其实是为了达到内联的目的。然而能否内联，其实并不是宏与函数的根本区别。宏与函数有着巨大的区别（这个我以后再讲），应该尽量避免使用宏。为了内联而使用宏，其实是滥用了宏，这会引起各种各样的麻烦，比如使程序难以理解，难以调试，容易出错等等。

- 每个函数只做一件简单的事情。有些人喜欢制造一些“通用”的函数，既可以做这个又可以做那个，它的内部依据某些变量和条件，来“选择”这个函数所要做的事情。比如，你也许写出这样的函数：

```
void foo() {
  if (getOS().equals("MacOS")) {
    a();
  } else {
    b();
  }
  c();
  if (getOS().equals("MacOS")) {
    d();
  } else {
    e();
  }
}
```

写这个函数的人，根据系统是否为“MacOS”来做不同的事情。你可以看出这个函数里，其实只有 `c()`是两种系统共有的，而其它的 `a()`, `b()`, `d()`,`e()`都属于不同的分支。

这种“复用”其实是有害的。如果一个函数可能做两种事情，它们之间共同点少于它们的不同点，那你最好就写两个不同的函数，否则这个函数的逻辑就不会很清晰，容易出现错误。其实，上面这个函数可以改写成两个函数：

```
void fooMacOS() {
  a();
  c();
  d();
}
```

和

```
void fooOther() {
  b();
  c();
  e();
}
```

如果你发现两件事情大部分内容相同，只有少数不同，多半时候你可以把相同的部分提取出去，做成一个辅助函数。比如，如果你有个函数是这样：

```
void foo() {
  a();
  b()
  c();
  if (getOS().equals("MacOS")) {
    d();
  } else {
    e();
  }
}
```

其中 a()，b()，c()都是一样的，只有 d()和 e()根据系统有所不同。那么你可以把 a()，b()，c()提取出去：

```
void preFoo() {
  a();
  b()
  c();
```

然后制造两个函数：

```
void fooMacOS() {
  preFoo();
  d();
}
```

和

```
void fooOther() {
  preFoo();
  e();
}
```

这样一来，我们既共享了代码，又做到了每个函数只做一件简单的事情。这样的代码，逻辑就更加清晰。

- 避免使用全局变量和类成员（class member）来传递信息，尽量使用局部变量和参数。有些人写代码，经常用类成员来传递信息，就像这样：

```
class A {
  String x;

  void findX() {
     ...
     x = ...;
  }

  void foo() {
    findX();
    ...
    print(x);
  }
}
```

首先，他使用 `findX()`，把一个值写入成员 `x`。然后，使用 `x` 的值。这样，`x` 就变成了 `findX` 和 `print` 之间的数据通道。由于 `x` 属于 `class A`，这样程序就失去了模块化的结构。由于这两个函数依赖于成员 `x`，它们不再有明确的输入和输出，而是依赖全局的数据。`findX` 和 `foo` 不再能够离开 `class A` 而存在，而且由于类成员还有可能被其他代码改变，代码变得难以理解，难以确保正确性。

如果你使用局部变量而不是类成员来传递信息，那么这两个函数就不需要依赖于某一个 class，而且更加容易理解，不易出错：

```
String findX() {
   ...
   x = ...;
   return x;
}
void foo() {
  int x = findX();
  print(x);
}
```

写可读的代码

有些人以为写很多注释就可以让代码更加可读，然而却发现事与愿违。注释不但没能让代码变得可读，反而由于大量的注释充斥在代码中间，让程序变得障眼难读。而且代码的逻辑一旦修改，就会有很多的注释变得过时，需要更新。修改注释是相当大的负担，所以大量的注释，反而成为了妨碍改进代码的绊脚石。

实际上，真正优雅可读的代码，是几乎不需要注释的。如果你发现需要写很多注释，那么你的代码肯定是含混晦涩，逻辑不清晰的。其实，程序语言相比自然语言，是更加强大而严谨的，它其实具有自然语言最主要的元素：主语，谓语，宾语，名词，动词，如果，那么，否则，是，不是，..... 所以如果你充分利用了程序语言的表达能力，你完全可以用程序本身来表达它到底在干什么，而不需要自然语言的辅助。

有少数的时候，你也许会为了绕过其他一些代码的设计问题，采用一些违反直觉的作法。这时候你可以使用很短注释，说明为什么要写成那奇怪的样子。这样的情况应该少出现，否则这意味着整个代码的设计都有问题。

如果没能合理利用程序语言提供的优势，你会发现程序还是很难懂，以至于需要写注释。所以我现在告诉你一些要点，也许可以帮助你大大减少写注释的必要：

1. 使用有意义的函数和变量名字。如果你的函数和变量的名字，能够切实的描述它们的逻辑，那么你就不需要写注释来解释它在干什么。比如：

```
// put elephant1 into fridge2
put(elephant1, fridge2);
```

   由于我的函数名 `put`，加上两个有意义的变量名 `elephant1` 和 `fridge2`，已经说明了这是在干什么（把大象放进冰箱），所以上面那句注释完全没有必要。

4. 局部变量应该尽量接近使用它的地方。有些人喜欢在函数最开头定义很多局部变量，然后在下面很远的地方使用它，就像这个样子：

```
void foo() {
  int index = ...;
```

```
7.   ...
8.   ...
9.   bar(index);
10.   ...
11. }
```

由于这中间都没有使用过 index，也没有改变过它所依赖的数据，所以这个变量定义，其实可以挪到接近使用它的地方：

```
void foo() {
  ...
  ...
  int index = ...;
  bar(index);
  ...
}
```

这样读者看到 bar(index)，不需要向上看很远就能发现 index 是如何算出来的。而且这种短距离，可以加强读者对于这里的“计算顺序”的理解。否则如果 index 在顶上，读者可能会怀疑，它其实保存了某种会变化的数据，或者它后来又被修改过。如果 index 放在下面，读者就清楚的知道，index 并不是保存了什么可变的值，而且它算出来之后就没变过。

如果你看透了局部变量的本质——它们就是电路里的导线，那你就能更好的理解近距离的好处。变量定义离用的地方越近，导线的长度就越短。你不需要摸着一根导线，绕来绕去找很远，就能发现接收它的端口，这样的电路就更容易理解。

12. 局部变量名字应该简短。这貌似跟第一点相冲突，简短的变量名怎么可能有意义呢？注意我这里说的是局部变量，因为它们处于局部，再加上第 2 点已经把它放到离使用位置尽量近的地方，所以根据上下文你就会容易知道它的意思：

比如，你有一个局部变量，表示一个操作是否成功：

```
boolean successInDeleteFile = deleteFile("foo.txt");
if (successInDeleteFile) {
  ...
```

```
} else {
  ...
}
```

这个局部变量 successInDeleteFile 大可不必这么啰嗦。因为它只用过一次，而且用它的地方就在下面一行，所以读者可以轻松发现它是 deleteFile 返回的结果。如果你把它改名为 success，其实读者根据一点上下文，也知道它表示"success in deleteFile"。所以你可以把它改成这样：

```
boolean success = deleteFile("foo.txt");
if (success) {
  ...
} else {
  ...
}
```

这样的写法不但没漏掉任何有用的语义信息，而且更加易读。successInDeleteFile 这种"camelCase"，如果超过了三个单词连在一起，其实是很碍眼的东西，所以如果你能用一个单词表示同样的意义，那当然更好。

13. 不要重用局部变量。很多人写代码不喜欢定义新的局部变量，而喜欢“重用”同一个局部变量，通过反复对它们进行赋值，来表示完全不同意思。比如这样写：

```
14. String msg;
15. if (...) {
16.   msg = "succeed";
17.   log.info(msg);
18. } else {
19.   msg = "failed";
20.   log.info(msg);
21. }
```

虽然这样在逻辑上是没有问题的，然而却不易理解，容易混淆。变量 msg 两次被赋值，表示完全不同的两个值。它们立即被 log.info 使用，没有传递到其它地方去。这种赋值的做法，把局部变量的作用域不必要

的增大，让人以为它可能在将来改变，也许会在其它地方被使用。更好的做法，其实是定义两个变量：

```
if (...) {
  String msg = "succeed";
  log.info(msg);
} else {
  String msg = "failed";
  log.info(msg);
}
```

由于这两个 msg 变量的作用域仅限于它们所处的 if 语句分支，你可以很清楚的看到这两个 msg 被使用的范围，而且知道它们之间没有任何关系。

22. 把复杂的逻辑提取出去，做成“帮助函数”。有些人写的函数很长，以至于看不清楚里面的语句在干什么，所以他们误以为需要写注释。如果你仔细观察这些代码，就会发现不清晰的那片代码，往往可以被提取出去，做成一个函数，然后在原来的地方调用。由于函数有一个名字，这样你就可以使用有意义的函数名来代替注释。举一个例子：

```
...
// put elephant1 into fridge2
openDoor(fridge2);
if (elephant1.alive()) {
  ...
} else {
   ...
}
closeDoor(fridge2);
...
```

如果你把这片代码提出去定义成一个函数：

```
void put(Elephant elephant, Fridge fridge) {
  openDoor(fridge);
  if (elephant.alive()) {
    ...
  } else {
     ...
  }
  closeDoor(fridge);
}
```

这样原来的代码就可以改成：

```
...
put(elephant1, fridge2);
...
```

更加清晰，而且注释也没必要了。

33. 把复杂的表达式提取出去，做成中间变量。有些人听说“函数式编程”是个好东西，也不理解它的真正含义，就在代码里使用大量嵌套的函数。像这样：

```
34. Pizza pizza = makePizza(crust(salt(), butter()),
35.    topping(onion(), tomato(), sausage()));
```

这样的代码一行太长，而且嵌套太多，不容易看清楚。其实训练有素的函数式程序员，都知道中间变量的好处，不会盲目的使用嵌套的函数。他们会把这代码变成这样：

```
Crust crust = crust(salt(), butter());
Topping topping = topping(onion(), tomato(), sausage());
Pizza pizza = makePizza(crust, topping);
```

这样写，不但有效地控制了单行代码的长度，而且由于引入的中间变量具有“意义”，步骤清晰，变得很容易理解。

36. 在合理的地方换行。对于绝大部分的程序语言，代码的逻辑是和空白字符无关的，所以你可以在几乎任何地方换行，你也可以不换行。这样的语言设计，是一个好东西，因为它给了程序员自由控制自己代码格式的能力。然而，它也引起了一些问题，因为很多人不知道如何合理的换行。

有些人喜欢利用 IDE 的自动换行机制，编辑之后用一个热键把整个代码重新格式化一遍，IDE 就会把超过行宽限制的代码自动折行。可是这种自动这行，往往没有根据代码的逻辑来进行，不能帮助理解代码。自动换行之后可能产生这样的代码：

```
if (someLongCondition1() && someLongCondition2() && someLongCondition3() &&
  someLongCondition4()) {
  ...
}
```

由于 someLongCondition4()超过了行宽限制，被编辑器自动换到了下面一行。虽然满足了行宽限制，换行的位置却是相当任意的，它并不能帮助人理解这代码的逻辑。这几个 boolean 表达式，全都用&&连接，所以它们其实处于平等的地位。为了表达这一点，当需要折行的时候，你应该把每一个表达式都放到新的一行，就像这个样子：

```
if (someLongCondition1() &&
    someLongCondition2() &&
    someLongCondition3() &&
    someLongCondition4()) {
  ...
}
```

这样每一个条件都对齐，里面的逻辑就很清楚了。再举个例子：

```
log.info("failed to find file {} for command {}, with exception {}", file, command,
  exception);
```

这行因为太长，被自动折行成这个样子。file，command 和 exception 本来是同一类东西，却有两个留在了第一行，最后一个被折到第二行。它就不如手动换行成这个样子：

```
log.info("failed to find file {} for command {}, with exception {}",
  file, command, exception);
```

把格式字符串单独放在一行，而把它的参数一并放在另外一行，这样逻辑就更加清晰。

为了避免 IDE 把这些手动调整好的换行弄乱，很多 IDE（比如 IntelliJ）的自动格式化设定里都有“保留原来的换行符”的设定。如果你发现 IDE 的换行不符合逻辑，你可以修改这些设定，然后在某些地方保留你自己的手动换行。

说到这里，我必须警告你，这里所说的“不需注释，让代码自己解释自己”，并不是说要让代码看起来像某种自然语言。有个叫 Chai 的 JavaScript 测试工具，可以让你这样写代码：

```
expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.length(3);
expect(tea).to.have.property('flavors').with.length(3);
```

这种做法是极其错误的。程序语言本来就比自然语言简单清晰，这种写法让它看起来像自然语言的样子，反而变得复杂难懂了。

写简单的代码

程序语言都喜欢标新立异，提供这样那样的“特性”，然而有些特性其实并不是什么好东西。很多特性都经不起时间的考验，最后带来的麻烦，比解决的问题还多。很多人盲目的追求“短小”和“精悍”，或者为了显示自己头脑聪明，学得快，所以喜欢利用语言里的一些特殊构造，写出过于“聪明”，难以理解的代码。

并不是语言提供什么，你就一定要把它用上的。实际上你只需要其中很小的一部分功能，就能写出优秀的代码。我一向反对“充分利用”程序语言里的所有特性。实际上，我心目中有一套最好的构造。不管语言提供了多么“神奇”的，“新”的特性，我基本都只用经过千锤百炼，我觉得值得信奈的那一套。

现在针对一些有问题的语言特性，我介绍一些我自己使用的代码规范，并且讲解一下为什么它们能让代码更简单。

- 避免使用自增减表达式（i++，++i，i--，--i）。这种自增减操作表达式其实是历史遗留的设计失误。它们含义蹊跷，非常容易弄错。它们把读和写这两种完全不同的操作，混淆缠绕在一起，把语义搞得乌七八糟。含有它们的表达式，结果可能取决于求值顺序，所以它可能在某种编译器下能正确运行，换一个编译器就出现离奇的错误。

其实这两个表达式完全可以分解成两步，把读和写分开：一步更新 i 的值，另外一步使用 i 的值。比如，如果你想写 `foo(i++)`，你完全可以把它拆成 `int t = i; i += 1; foo(t);`。如果你想写 `foo(++i)`，可以拆成 `i += 1; foo(i);` 拆开之后的代码，含义完全一致，却清晰很多。到底更新是在取值之前还是之后，一目了然。

有人也许以为 i++或者++i 的效率比拆开之后要高，这只是一种错觉。这些代码经过基本的编译器优化之后，生成的机器代码是完全没有区别的。自增减表达式只有在两种情况下才可以安全的使用。一种是在 for 循环的 update 部分，比如 `for(int i = 0; i < 5; i++)`。另一种情况是写成单独的一行，比如 `i++;`。这两种情况是完全没有歧义的。你需要避免其它的情况，比如用在复杂的表达式里面，比如 `foo(i++)`，`foo(++i) + foo(i)`，..... 没有人应该知道，或者去追究这些是什么意思。

- 永远不要省略花括号。很多语言允许你在某种情况下省略掉花括号，比如 C，Java 都允许你在 if 语句里面只有一句话的时候省略掉花括号：

```
if (...)
  action1();
```

咋一看少打了两个字，多好。可是这其实经常引起奇怪的问题。比如，你后来想要加一句话 `action2()`到这个 if 里面，于是你就把代码改成：

```
if (...)
  action1();
  action2();
```

为了美观，你很小心的使用了 `action1()`的缩进。咋一看它们是在一起的，所以你下意识里以为它们只会在 if 的条件为真的时候执行，然而 `action2()`却其实在 if 外面，它会被无条件的执行。我把这种现象叫做“光学幻觉”（optical illusion），理论上每个程序员都应该发现这个错误，然而实际上却容易被忽视。

那么你问，谁会这么傻，我在加入 `action2()`的时候加上花括号不就行了？可是从设计的角度来看，这样其实并不是合理的作法。首先，也许你以后又想把 `action2()`去掉，这样你为了样式一致，又得把花括号拿掉，烦不烦啊？其次，这使得代码样式不一致，有的 if 有花括号，有的又没有。况且，你为什么需要记住这个规则？如果你不问三七二十一，只要是 if-else 语句，把花括号全都打上，就可以想都不用想了，就当 C 和 Java 没提供给你这个特殊写法。这样就可以保持完全的一致性，减少不必要的思考。

有人可能会说，全都打上花括号，只有一句话也打上，多碍眼啊？然而经过实行这种编码规范几年之后，我并没有发现这种写法更加碍眼，反而由于花括号的存在，使得代码界限明确，让我的眼睛负担更小了。

- 合理使用括号，不要盲目依赖操作符优先级。利用操作符的优先级来减少括号，对于 `1 + 2 * 3` 这样常见的算数表达式，是没问题的。然而有些人如此的仇恨括号，以至于他们会写出 `2 << 7 - 2 * 3` 这样的表达式，而完全不用括号。

  这里的问题，在于移位操作<<的优先级，是很多人不熟悉，而且是违反常理的。由于 `x << 1` 相当于把 x 乘以 2，很多人误以为这个表达式相当于`(2 << 7) - (2 * 3)`，所以等于 250。然而实际上<<的优先级比加法+还要低，所以这表达式其实相当于 `2 << (7 - 2 * 3)`，所以等于 4！

  解决这个问题的办法，不是要每个人去把操作符优先级表给硬背下来，而是合理的加入括号。比如上面的例子，最好直接加上括号写成 `2 << (7 - 2 * 3)`。虽然没有括号也表示同样的意思，但是加上括号就更加清晰，读者不再需要死记<<的优先级就能理解代码。

- 避免使用 continue 和 break。循环语句（for，while）里面出现 return 是没问题的，然而如果你使用了 continue 或者 break，就会让循环的逻辑和终止条件变得复杂，难以确保正确。

出现 continue 或者 break 的原因，往往是对循环的逻辑没有想清楚。如果你考虑周全了，应该是几乎不需要 continue 或者 break 的。如果你的循环里出现了 continue 或者 break，你就应该考虑改写这个循环。改写循环的办法有多种：

1. 如果出现了 continue，你往往只需要把 continue 的条件反向，就可以消除 continue。
2. 如果出现了 break，你往往可以把 break 的条件，合并到循环头部的终止条件里，从而去掉 break。
3. 有时候你可以把 break 替换成 return，从而去掉 break。
4. 如果以上都失败了，你也许可以把循环里面复杂的部分提取出来，做成函数调用，之后 continue 或者 break 就可以去掉了。

下面我对这些情况举一些例子。

情况 1：下面这段代码里面有一个 continue：

```
List<String> goodNames = new ArrayList<>();
for (String name: names) {
  if (name.contains("bad")) {
    continue;
  }
  goodNames.add(name);
  ...
}
```

它说：“如果 name 含有'bad'这个词，跳过后面的循环代码……”注意，这是一种“负面”的描述，它不是在告诉你什么时候“做”一件事，而是在告诉你什么时候“不做”一件事。为了知道它到底在干什么，你必须搞清楚 continue 会导致哪些语句被跳过了，然后脑子里把逻辑反个向，你才能知道它到底想做什么。这就是为什么含有 continue 和 break 的循环不容易理解，它们依靠“控制流”来描述“不做什么”，“跳过什么”，结果到最后你也没搞清楚它到底“要做什么”。

其实，我们只需要把 continue 的条件反向，这段代码就可以很容易的被转换成等价的，不含 continue 的代码：

```
List<String> goodNames = new ArrayList<>();
for (String name: names) {
  if (!name.contains("bad")) {
    goodNames.add(name);
    ...
  }
}
```

goodNames.add(name);和它之后的代码全部被放到了 if 里面，多了一层缩进，然而 continue 却没有了。你再读这段代码，就会发现更加清晰。因为它是一种更加“正面”地描述。它说：“在 name 不含有'bad'这个词的时候，把它加到 goodNames 的链表里面……”

情况 2：for 和 while 头部都有一个循环的“终止条件”，那本来应该是这个循环唯一的退出条件。如果你在循环中间有 break，它其实给这个循环增加了一个退出条件。你往往只需要把这个条件合并到循环头部，就可以去掉 break。

比如下面这段代码：

```
while (condition1) {
  ...
  if (condition2) {
    break;
  }
}
```

当 condition 成立的时候，break 会退出循环。其实你只需要把 condition2 反转之后，放到 while 头部的终止条件，就可以去掉这种 break 语句。改写后的代码如下：

```
while (condition1 && !condition2) {
  ...
}
```

这种情况表面上貌似只适用于 break 出现在循环开头或者末尾的时候，然而其实大部分时候，break 都可以通过某种方式，移动到循环的开头或者末尾。具体的例子我暂时没有，等出现的时候再加进来。

情况 3：很多 break 退出循环之后，其实接下来就是一个 return。这种 break 往往可以直接换成 return。比如下面这个例子：

```
public boolean hasBadName(List<String> names) {
    boolean result = false;

    for (String name: names) {
        if (name.contains("bad")) {
            result = true;
            break;
        }
    }
    return result;
}
```

这个函数检查 names 链表里是否存在一个名字，包含“bad”这个词。它的循环里包含一个 break 语句。这个函数可以被改写成：

```
public boolean hasBadName(List<String> names) {
    for (String name: names) {
        if (name.contains("bad")) {
            return true;
        }
    }
    return false;
}
```

改进后的代码，在 name 里面含有“bad”的时候，直接用 `return true` 返回，而不是对 result 变量赋值，break 出去，最后才返回。如果循环结束了还没有 return，那就返回 false，表示没有找到这样的名字。使用 return 来代替 break，这样 break 语句和 result 这个变量，都一并被消除掉了。

我曾经见过很多其他使用 continue 和 break 的例子，几乎无一例外的可以被消除掉，变换后的代码变得清晰很多。我的经验是，99%的

> break 和 continue，都可以通过替换成 return 语句，或者翻转 if 条件的方式来消除掉。剩下的 1%含有复杂的逻辑，但也可以通过提取一个帮助函数来消除掉。修改之后的代码变得容易理解，容易确保正确。

写直观的代码

我写代码有一条重要的原则：如果有更加直接，更加清晰的写法，就选择它，即使它看起来更长，更笨，也一样选择它。比如，Unix 命令行有一种“巧妙”的写法是这样：

```
command1 && command2 && command3
```

由于 Shell 语言的逻辑操作 `a && b` 具有“短路”的特性，如果 `a` 等于 false，那么 `b` 就没必要执行了。这就是为什么当 command1 成功，才会执行 command2，当 command2 成功，才会执行 command3。同样，

```
command1 || command2 || command3
```

操作符||也有类似的特性。上面这个命令行，如果 command1 成功，那么 command2 和 command3 都不会被执行。如果 command1 失败，command2 成功，那么 command3 就不会被执行。

这比起用 if 语句来判断失败，似乎更加巧妙和简洁，所以有人就借鉴了这种方式，在程序的代码里也使用这种方式。比如他们可能会写这样的代码：

```
if (action1() || action2() && action3()) {
  ...
}
```

你看得出来这代码是想干什么吗？action2 和 action3 什么条件下执行，什么条件下不执行？也许稍微想一下，你知道它在干什么：“如果 action1 失败了，执行 action2，如果 action2 成功了，执行 action3”。然而那种语义，并不是直接的“映射”在这代码上面的。比如“失败”这个词，对应了代码里的哪一个字呢？

你找不出来，因为它包含在了||的语义里面，你需要知道||的短路特性，以及逻辑或的语义才能知道这里面在说“如果 action1 失败……”。每一次看到这行代码，你都需要思考一下，这样积累起来的负荷，就会让人很累。

其实，这种写法是滥用了逻辑操作&&和||的短路特性。这两个操作符可能不执行右边的表达式，原因是为了机器的执行效率，而不是为了给人提供这种“巧妙”的用法。这两个操作符的本意，只是作为逻辑操作，它们并不是拿来给你代替 if 语句的。也就是说，它们只是碰巧可以达到某些 if 语句的效果，但你不应该因此就用它来代替 if 语句。如果你这样做了，就会让代码晦涩难懂。

上面的代码写成笨一点的办法，就会清晰很多：

```
if (!action1()) {
  if (action2()) {
    action3();
  }
}
```

这里我很明显的看出这代码在说什么，想都不用想：如果 action1()失败了，那么执行 action2()，如果 action2()成功了，执行 action3()。你发现这里面的一一对应关系吗？if=如果，!=失败，…… 你不需要利用逻辑学知识，就知道它在说什么。

写无懈可击的代码

在之前一节里，我提到了自己写的代码里面很少出现只有一个分支的 if 语句。我写出的 if 语句，大部分都有两个分支，所以我的代码很多看起来是这个样子：

```
if (...) {
  if (...) {
    ...
    return false;
  } else {
    return true;
  }
} else if (...) {
```

```
  ...
  return false;
} else {
  return true;
}
```

使用这种方式，其实是为了无懈可击的处理所有可能出现的情况，避免漏掉 corner case。每个 if 语句都有两个分支的理由是：如果 if 的条件成立，你做某件事情；但是如果 if 的条件不成立，你应该知道要做什么另外的事情。不管你的 if 有没有 else，你终究是逃不掉，必须得思考这个问题的。

很多人写 if 语句喜欢省略 else 的分支，因为他们觉得有些 else 分支的代码重复了。比如我的代码里，两个 else 分支都是 `return true`。为了避免重复，他们省略掉那两个 else 分支，只在最后使用一个 `return true`。这样，缺了 else 分支的 if 语句，控制流自动“掉下去”，到达最后的 `return true`。他们的代码看起来像这个样子：

```
if (...) {
  if (...) {
    ...
    return false;
  }
} else if (...) {
  ...
  return false;
}
return true;
```

这种写法看似更加简洁，避免了重复，然而却很容易出现疏忽和漏洞。嵌套的 if 语句省略了一些 else，依靠语句的“控制流”来处理 else 的情况，是很难正确的分析和推理的。如果你的 if 条件里使用了&&和||之类的逻辑运算，就更难看出是否涵盖了所有的情况。

由于疏忽而漏掉的分支，全都会自动“掉下去”，最后返回意想不到的结果。即使你看一遍之后确信是正确的，每次读这段代码，你都不能确信它照顾了所有的情况，又得重新推理一遍。这简洁的写法，带来的是反复的，沉重的头脑开

销。这就是所谓“面条代码”，因为程序的逻辑分支，不是像一棵枝叶分明的树，而是像面条一样绕来绕去。

另外一种省略 else 分支的情况是这样：

```
String s = "";
if (x < 5) {
  s = "ok";
}
```

写这段代码的人，脑子里喜欢使用一种“缺省值”的做法。s 缺省为 null，如果 x<5，那么把它改变（mutate）成“ok”。这种写法的缺点是，当 x<5 不成立的时候，你需要往上面看，才能知道 s 的值是什么。这还是你运气好的时候，因为 s 就在上面不远。很多人写这种代码的时候，s 的初始值离判断语句有一定的距离，中间还有可能插入一些其它的逻辑和赋值操作。这样的代码，把变量改来改去的，看得人眼花，就容易出错。

现在比较一下我的写法：

```
String s;
if (x < 5) {
  s = "ok";
} else {
  s = "";
}
```

这种写法貌似多打了一两个字，然而它却更加清晰。这是因为我们明确的指出了 x<5 不成立的时候，s 的值是什么。它就摆在那里，它是""（空字符串）。注意，虽然我也使用了赋值操作，然而我并没有“改变”s 的值。s 一开始的时候没有值，被赋值之后就再也没有变过。我的这种写法，通常被叫做更加“函数式”，因为我只赋值一次。

如果我漏写了 else 分支，Java 编译器是不会放过我的。它会抱怨：“在某个分支，s 没有被初始化。”这就强迫我清清楚楚的设定各种条件下 s 的值，不漏掉任何一种情况。

当然，由于这个情况比较简单，你还可以把它写成这样：

```
String s = x < 5 ? "ok" : "";
```

对于更加复杂的情况，我建议还是写成 if 语句为好。

正确处理错误

使用有两个分支的 if 语句，只是我的代码可以达到无懈可击的其中一个原因。这样写 if 语句的思路，其实包含了使代码可靠的一种通用思想：穷举所有的情况，不漏掉任何一个。

程序的绝大部分功能，是进行信息处理。从一堆纷繁复杂，模棱两可的信息中，排除掉绝大部分“干扰信息”，找到自己需要的那一个。正确地对所有的“可能性”进行推理，就是写出无懈可击代码的核心思想。这一节我来讲一讲，如何把这种思想用在错误处理上。

错误处理是一个古老的问题，可是经过了几十年，还是很多人没搞明白。Unix 的系统 API 手册，一般都会告诉你可能出现的返回值和错误信息。比如，Linux 的 read 系统调用手册里面有如下内容：

```
RETURN VALUE
On success, the number of bytes read is returned...

On error, -1 is returned, and errno is set appropriately.



ERRORS



EAGAIN, EBADF, EFAULT, EINTR, EINVAL, ...
```

很多初学者，都会忘记检查 read 的返回值是否为-1，觉得每次调用 read 都得检查返回值真繁琐，不检查貌似也相安无事。这种想法其实是很危险的。如果

函数的返回值告诉你，要么返回一个正数，表示读到的数据长度，要么返回-1，那么你就必须要对这个-1 作出相应的，有意义的处理。千万不要以为你可以忽视这个特殊的返回值，因为它是一种“可能性”。代码漏掉任何一种可能出现的情况，都可能产生意想不到的灾难性结果。

对于 Java 来说，这相对方便一些。Java 的函数如果出现问题，一般通过异常（exception）来表示。你可以把异常加上函数本来的返回值，看成是一个“union 类型”。比如：

```
String foo() throws MyException {
  ...
}
```

这里 MyException 是一个错误返回。你可以认为这个函数返回一个 union 类型：{String, MyException}。任何调用 foo 的代码，必须对 MyException 作出合理的处理，才有可能确保程序的正确运行。Union 类型是一种相当先进的类型，目前只有极少数语言（比如 Typed Racket）具有这种类型，我在这里提到它，只是为了方便解释概念。掌握了概念之后，你其实可以在头脑里实现一个 union 类型系统，这样使用普通的语言也能写出可靠的代码。

由于 Java 的类型系统强制要求函数在类型里面声明可能出现的异常，而且强制调用者处理可能出现的异常，所以基本上不可能出现由于疏忽而漏掉的情况。但有些 Java 程序员有一种恶习，使得这种安全机制几乎完全失效。每当编译器报错，说“你没有 catch 这个 foo 函数可能出现的异常”时，有些人想都不想，直接把代码改成这样：

```
try {
  foo();
} catch (Exception e) {}
```

或者最多在里面放个 log，或者干脆把自己的函数类型上加上 throws Exception，这样编译器就不再抱怨。这些做法貌似很省事，然而都是错误的，你终究会为此付出代价。

如果你把异常 catch 了，忽略掉，那么你就不知道 foo 其实失败了。这就像开车时看到路口写着“前方施工，道路关闭”，还继续往前开。这当然迟早会出问题，因为你根本不知道自己在干什么。

catch 异常的时候，你不应该使用 Exception 这么宽泛的类型。你应该正好 catch 可能发生的那种异常 A。使用宽泛的异常类型有很大的问题，因为它会不经意的 catch 住另外的异常（比如 B）。你的代码逻辑是基于判断 A 是否出现，可你却 catch 所有的异常（Exception 类），所以当其它的异常 B 出现的时候，你的代码就会出现莫名其妙的问题，因为你以为 A 出现了，而其实它没有。这种 bug，有时候甚至使用 debugger 都难以发现。

如果你在自己函数的类型加上 `throws Exception`，那么你就不可避免的需要在调用它的地方处理这个异常，如果调用它的函数也写着 `throws Exception`，这毛病就传得更远。我的经验是，尽量在异常出现的当时就作出处理。否则如果你把它返回给你的调用者，它也许根本不知道该怎么办了。

另外，try { ... } catch 里面，应该包含尽量少的代码。比如，如果 `foo` 和 `bar` 都可能产生异常 A，你的代码应该尽可能写成：

```
try {
  foo();
} catch (A e) {...}

try {
  bar();
} catch (A e) {...}
```

而不是

```
try {
  foo();
  bar();
} catch (A e) {...}
```

第一种写法能明确的分辨是哪一个函数出了问题，而第二种写法全都混在一起。明确的分辨是哪一个函数出了问题，有很多的好处。比如，如果你的 catch 代

码里面包含 log，它可以提供给你更加精确的错误信息，这样会大大地加速你的调试过程。

正确处理 null 指针

穷举的思想是如此的有用，依据这个原理，我们可以推出一些基本原则，它们可以让你无懈可击的处理 null 指针。

首先你应该知道，许多语言（C，C++，Java，C#，……）的类型系统对于 null 的处理，其实是完全错误的。这个错误源自于 Tony Hoare 最早的设计，Hoare 把这个错误称为自己的“billion dollar mistake”，因为由于它所产生的财产和人力损失，远远超过十亿美元。

这些语言的类型系统允许 null 出现在任何对象（指针）类型可以出现的地方，然而 null 其实根本不是一个合法的对象。它不是一个 String，不是一个 Integer，也不是一个自定义的类。null 的类型本来应该是 NULL，也就是 null 自己。根据这个基本观点，我们推导出以下原则：

- 尽量不要产生 null 指针。尽量不要用 null 来初始化变量，函数尽量不要返回 null。如果你的函数要返回“没有”，“出错了”之类的结果，尽量使用 Java 的异常机制。虽然写法上有点别扭，然而 Java 的异常，和函数的返回值合并在一起，基本上可以当成 union 类型来用。比如，如果你有一个函数 find，可以帮你找到一个 String，也有可能什么也找不到，你可以这样写：

```
public String find() throws NotFoundException {
  if (...) {
    return ...;
  } else {
    throw new NotFoundException();
  }
}
```

Java 的类型系统会强制你 catch 这个 NotFoundException，所以你不可能像漏掉检查 null 一样，漏掉这种情况。Java 的异常也是一个比较容易滥用的东西，不过我已经在上一节告诉你如何正确的使用异常。

Java 的 try...catch 语法相当的繁琐和蹩脚，所以如果你足够小心的话，像 `find` 这类函数，也可以返回 null 来表示“没找到”。这样稍微好看一些，因为你调用的时候不必用 try...catch。很多人写的函数，返回 null 来表示“出错了”，这其实是对 null 的误用。“出错了”和“没有”，其实完全是两码事。“没有”是一种很常见，正常的情况，比如查哈希表没找到，很正常。“出错了”则表示罕见的情况，本来正常情况下都应该存在有意义的值，偶然出了问题。如果你的函数要表示“出错了”，应该使用异常，而不是 null。

- 不要把 null 放进“容器数据结构”里面。所谓容器（collection），是指一些对象以某种方式集合在一起，所以 null 不应该被放进 Array，List，Set 等结构，不应该出现在 Map 的 key 或者 value 里面。把 null 放进容器里面，是一些莫名其妙错误的来源。因为对象在容器里的位置一般是动态决定的，所以一旦 null 从某个入口跑进去了，你就很难再搞明白它去了哪里，你就得被迫在所有从这个容器里取值的位置检查 null。你也很难知道到底是谁把它放进去的，代码多了就导致调试极其困难。

  解决方案是：如果你真要表示“没有”，那你就干脆不要把它放进去（Array，List，Set 没有元素，Map 根本没那个 entry），或者你可以指定一个特殊的，真正合法的对象，用来表示“没有”。

  需要指出的是，类对象并不属于容器。所以 null 在必要的时候，可以作为对象成员的值，表示它不存在。比如：

```
class A {
  String name = null;
  ...
}
```

之所以可以这样，是因为 null 只可能在 A 对象的 name 成员里出现，你不用怀疑其它的成员因此成为 null。所以你每次访问 name 成员时，检查它是否是 null 就可以了，不需要对其他成员也做同样的检查。

- 函数调用者：明确理解 null 所表示的意义，尽早检查和处理 null 返回值，减少它的传播。null 很讨厌的一个地方，在于它在不同的地方可能表示不同的意义。有时候它表示“没有”，“没找到”。有时候它表示“出错了”，“失败了”。有时候它甚至可以表示“成功了”，…… 这其中有很多误用之处，不过无论如何，你必须理解每一个 null 的意义，不能给混淆起来。

  如果你调用的函数有可能返回 null，那么你应该在第一时间对 null 做出“有意义”的处理。比如，上述的函数 `find`，返回 null 表示“没找到”，那么调用 `find` 的代码就应该在它返回的第一时间，检查返回值是否是 null，并且对“没找到”这种情况，作出有意义的处理。

  “有意义”是什么意思呢？我的意思是，使用这函数的人，应该明确的知道在拿到 null 的情况下该怎么做，承担起责任来。他不应该只是“向上级汇报”，把责任踢给自己的调用者。如果你违反了这一点，就有可能采用一种不负责任，危险的写法：

```
public String foo() {
  String found = find();
  if (found == null) {
    return null;
  }
}
```

  当看到 find()返回了 null，foo 自己也返回 null。这样 null 就从一个地方，游走到了另一个地方，而且它表示另外一个意思。如果你不假思索就写出这样的代码，最后的结果就是代码里面随时随地都可能出现 null。到后来为了保护自己，你的每个函数都会写成这样：

```
public void foo(A a, B b, C c) {
  if (a == null) { ... }
  if (b == null) { ... }
  if (c == null) { ... }
```

```
  ...
}
```

- 函数作者：明确声明不接受 null 参数，当参数是 null 时立即崩溃。不要试图对 null 进行“容错”，不要让程序继续往下执行。如果调用者使用了 null 作为参数，那么调用者（而不是函数作者）应该对程序的崩溃负全责。

  上面的例子之所以成为问题，就在于人们对于 null 的“容忍态度”。这种“保护式”的写法，试图“容错”，试图“优雅的处理 null”，其结果是让调用者更加肆无忌惮的传递 null 给你的函数。到后来，你的代码里出现一堆堆 nonsense 的情况，null 可以在任何地方出现，都不知道到底是哪里产生出来的。谁也不知道出现了 null 是什么意思，该做什么，所有人都把 null 踢给其他人。最后这 null 像瘟疫一样蔓延开来，到处都是，成为一场噩梦。

  正确的做法，其实是强硬的态度。你要告诉函数的使用者，我的参数全都不能是 null，如果你给我 null，程序崩溃了该你自己负责。至于调用者代码里有 null 怎么办，他自己该知道怎么处理（参考以上几条），不应该由函数作者来操心。

  采用强硬态度一个很简单的做法是使用 `Objects.requireNonNull()`。它的定义很简单：

  ```
  public static <T> T requireNonNull(T obj) {
    if (obj == null) {
      throw new NullPointerException();
    } else {
      return obj;
    }
  }
  ```

  你可以用这个函数来检查不想接受 null 的每一个参数，只要传进来的参数是 null，就会立即触发 `NullPointerException` 崩溃掉，这样你就可以有效地防止 null 指针不知不觉传递到其它地方去。

- 使用@NotNull 和@Nullable 标记。IntelliJ 提供了@NotNull 和 @Nullable 两种标记，加在类型前面，这样可以比较简洁可靠地防止 null 指针的出现。IntelliJ 本身会对含有这种标记的代码进行静态分析，指出运行时可能出现 `NullPointerException` 的地方。在运行时，会在 null 指针不该出现的地方产生 `IllegalArgumentException`，即使那个 null 指针你从来没有 deference。这样你可以在尽量早期发现并且防止 null 指针的出现。
- 使用 Optional 类型。Java 8 和 Swift 之类的语言，提供了一种叫 Optional 的类型。正确的使用这种类型，可以在很大程度上避免 null 的问题。null 指针的问题之所以存在，是因为你可以在没有“检查”null 的情况下，“访问”对象的成员。

  Optional 类型的设计原理，就是把“检查”和“访问”这两个操作合二为一，成为一个“原子操作”。这样你没法只访问，而不进行检查。这种做法其实是 ML，Haskell 等语言里的模式匹配（pattern matching）的一个特例。模式匹配使得类型判断和访问成员这两种操作合二为一，所以你没法犯错。

  比如，在 Swift 里面，你可以这样写：

```
let found = find()
if let content = found {
  print("found: " + content)
}
```

  你从 `find()`函数得到一个 Optional 类型的值 `found`。假设它的类型是 `String?`，那个问号表示它可能包含一个 String，也可能是 nil。然后你就可以用一种特殊的 if 语句，同时进行 null 检查和访问其中的内容。这个 if 语句跟普通的 if 语句不一样，它的条件不是一个 Bool，而是一个变量绑定 `let content = found`。

  我不是很喜欢这语法，不过这整个语句的含义是：如果 found 是 nil，那么整个 if 语句被略过。如果它不是 nil，那么变量 content 被绑定到

found 里面的值（unwrap 操作），然后执行 `print("found: " + content)`。由于这种写法把检查和访问合并在了一起，你没法只进行访问而不检查。

Java 8 的做法比较蹩脚一些。如果你得到一个 Optional 类型的值 found，你必须使用“函数式编程”的方式，来写这之后的代码：

```
Optional<String> found = find();
found.ifPresent(content -> System.out.println("found: " + content));
```

这段 Java 代码跟上面的 Swift 代码等价，它包含一个“判断”和一个“取值”操作。ifPresent 先判断 found 是否有值（相当于判断是不是 null）。如果有，那么将其内容“绑定”到 lambda 表达式的 content 参数（unwrap 操作），然后执行 lambda 里面的内容，否则如果 found 没有内容，那么 ifPresent 里面的 lambda 不执行。

Java 的这种设计有个问题。判断 null 之后分支里的内容，全都得写在 lambda 里面。在函数式编程里，这个 lambda 叫做“continuation”，Java 把它叫做 “Consumer”，它表示“如果 found 不是 null，拿到它的值，然后应该做什么”。由于 lambda 是个函数，你不能在里面写 `return` 语句返回出外层的函数。比如，如果你要改写下面这个函数（含有 null）：

```
public static String foo() {
  String found = find();
  if (found != null) {
    return found;
  } else {
    return "";
  }
}
```

就会比较麻烦。因为如果你写成这样：

```
public static String foo() {
  Optional<String> found = find();
  found.ifPresent(content -> {
```

```
    return content;    // can't return from foo here
  });
  return "";
}
```

里面的 `return a`，并不能从函数 `foo` 返回出去。它只会从 lambda 返回，而且由于那个 lambda（Consumer.accept）的返回类型必须是 `void`，编译器会报错，说你返回了 String。由于 Java 里 closure 的自由变量是只读的，你没法对 lambda 外面的变量进行赋值，所以你也不能采用这种写法：

```
public static String foo() {
  Optional<String> found = find();
  String result = "";
  found.ifPresent(content -> {
    result = content;    // can't assign to result
  });
  return result;
}
```

所以，虽然你在 lambda 里面得到了 found 的内容，如何使用这个值，如何返回一个值，却让人摸不着头脑。你平时的那些 Java 编程手法，在这里几乎完全废掉了。实际上，判断 null 之后，你必须使用 Java 8 提供的一系列古怪的函数式编程操作：`map`, `flatMap`, `orElse` 之类，想法把它们组合起来，才能表达出原来代码的意思。比如之前的代码，只能改写成这样：

```
public static String foo() {
  Optional<String> found = find();
  return found.orElse("");
}
```

这简单的情况还好。复杂一点的代码，我还真不知道怎么表达，我怀疑 Java 8 的 Optional 类型的方法，到底有没有提供足够的表达力。那里面少数几个东西表达能力不咋的，论工作原理，却可以扯到 functor，continuation，甚至 monad 等高深的理论…… 仿佛用了 Optional 之后，这语言就不再是 Java 了一样。

所以 Java 虽然提供了 Optional，但我觉得可用性其实比较低，难以被人接受。相比之下，Swift 的设计更加简单直观，接近普通的过程式编程。你只需要记住一个特殊的语法 `if let content = found {...}`，里面的代码写法，跟普通的过程式语言没有任何差别。

总之你只要记住，使用 Optional 类型，要点在于“原子操作”，使得 null 检查与取值合二为一。这要求你必须使用我刚才介绍的特殊写法。如果你违反了这一原则，把检查和取值分成两步做，还是有可能犯错误。比如在 Java 8 里面，你可以使用 `found.get()`这样的方式直接访问 found 里面的内容。在 Swift 里你也可以使用 `found!`来直接访问而不进行检查。

你可以写这样的 Java 代码来使用 Optional 类型：

```
Option<String> found = find();
if (found.isPresent()) {
  System.out.println("found: " + found.get());
}
```

如果你使用这种方式，把检查和取值分成两步做，就可能会出现运行时错误。`if (found.isPresent())`本质上跟普通的 null 检查，其实没什么两样。如果你忘记判断 `found.isPresent()`，直接进行 `found.get()`，就会出现 `NoSuchElementException`。这跟 `NullPointerException` 本质上是一回事。所以这种写法，比起普通的 null 的用法，其实换汤不换药。如果你要用 Optional 类型而得到它的益处，请务必遵循我之前介绍的“原子操作”写法。

防止过度工程

人的脑子真是奇妙的东西。虽然大家都知道过度工程（over-engineering）不好，在实际的工程中却经常不由自主的出现过度工程。我自己也犯过好多次这种错误，所以觉得有必要分析一下，过度工程出现的信号和兆头，这样可以在初期的时候就及时发现并且避免。

过度工程即将出现的一个重要信号，就是当你过度的思考“将来”，考虑一些还没有发生的事情，还没有出现的需求。比如，“如果我们将来有了上百万行代码，有了几千号人，这样的工具就支持不了了”，“将来我可能需要这个功能，所以我现在就把代码写来放在那里”，“将来很多人要扩充这片代码，所以现在我们就让它变得可重用”……

这就是为什么很多软件项目如此复杂。实际上没做多少事情，却为了所谓的“将来”，加入了很多不必要的复杂性。眼前的问题还没解决呢，就被“将来”给拖垮了。人们都不喜欢目光短浅的人，然而在现实的工程中，有时候你就是得看近一点，把手头的问题先搞定了，再谈以后扩展的问题。

另外一种过度工程的来源，是过度的关心“代码重用”。很多人“可用”的代码还没写出来呢，就在关心“重用”。为了让代码可以重用，最后被自己搞出来的各种框架捆住手脚，最后连可用的代码就没写好。如果可用的代码都写不好，又何谈重用呢？很多一开头就考虑太多重用的工程，到后来被人完全抛弃，没人用了，因为别人发现这些代码太难懂了，自己从头开始写一个，反而省好多事。

过度地关心“测试”，也会引起过度工程。有些人为了测试，把本来很简单的代码改成“方便测试”的形式，结果引入很多复杂性，以至于本来一下就能写对的代码，最后复杂不堪，出现很多 bug。

世界上有两种“没有 bug”的代码。一种是“没有明显的 bug 的代码”，另一种是“明显没有 bug 的代码”。第一种情况，由于代码复杂不堪，加上很多测试，各种 coverage，貌似测试都通过了，所以就认为代码是正确的。第二种情况，由于代码简单直接，就算没写很多测试，你一眼看去就知道它不可能有 bug。你喜欢哪一种“没有 bug”的代码呢？

根据这些，我总结出来的防止过度工程的原则如下：

1. 先把眼前的问题解决掉，解决好，再考虑将来的扩展问题。
2. 先写出可用的代码，反复推敲，再考虑是否需要重用的问题。
3. 先写出可用，简单，明显没有 bug 的代码，再考虑测试的问题。