编程的智慧

编程是创造性的工作，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。然而我希望它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。这是一种历史遗留的错觉。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，这也是一种历史遗留的错觉。在早期的C语言编译器里，只有macro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。

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。

2. 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 = ...;
  ...
  ...
  bar(index);
  ...
}
```

由于这中间都没有使用过x，也没有改变过x的值所依赖的数据，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用来保存了某种会变化的数据，这些数据在index被复制之后被修改过，或者index变量后来又被修改过。如果index放在下面，读者就清楚的知道，index并不是保存了什么后来变化过的值，而且它算出来就没变过。

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

3. 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 successInFindingFile = findFile("foo.txt");
if (successInFindingFile) {
  ...
} else {
  ...
}
```

这个局部变量successInFindingFile大可不必这么详细。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findFile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 findFile"。所以你可以把它改成这样：

```
boolean success = findFile("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。successInFindingFile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。

4. 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.isDead()) {
  ...
} 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.isDead()) {
    ...
  } 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。

5. 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：

```
Pizza pizza = makePizza(crust(salt(), butter()),
   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。

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”。

上面这种“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。

- 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”它的成员。

  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，如果你得到一个Optional类型的值`found`，假设它的类型是`String?`（那个问号表示它可能是String，也可能是

nil）。然后你就可以用下面这样的代码，同时进行null检查和访问其中的内容：

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

这是一种特殊的if语句，它的条件不是一个普通的Bool，而是一个变量绑定语句`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的内容是nil，那么整个if语句被略过。如果它不是nil，那么变量content得到其中的值（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。如果你只使用这种写法，而不使用`found!`这样的强行访问，那么NullPointerException就不会出现。

Java 8的做法比较蹩脚一些。如果你得到一个Optional类型的值found，你必须使用一种奇怪的“函数式编程”方式，来写这之后的代码：

```
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。如果有，那么把内容放进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的这种设计有一个严重的问题。一旦你使用这种方式，判断null之后的内容，全都得写在lambda里面，使用`andThen`串接起来。这种方式在函数式编程里面，叫做CPS形式（Continuation-Passing Style）。这种样式的代码虽然看起来很酷，可是难写难读，一般只用在函数式语言编译器的内部。（题外话：所谓的“王垠40行代码”，就是用来自动把普通的代码编译成CPS的形式。）所以Java的Optional类型，并不是真正好用的，其实还不如直接使用null。相比之下，Swift的设计要简单直观很多。

总之你只要记住，使用Optional类型，要点在于进行“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。