如何提高代码的可读性? - 读《编写可读代码的艺术》

笔记本:	MAGIC CS		
创建时间:	2019/11/25 18:54	**更新时间:**	2019/11/25 18:55
作者:	王艺辉		
URL:	https://www.jianshu.com/p/422657e71e3a		

一. 为什么读这本书

很多同行在编写代码的时候往往只关注一些宏观上的主题：架构，设计模式，数据结构等等，却忽视了一些更细节上的点：比如变量如何命名与使用概括。

不同于宏观上的架构，设计模式等需要好几个类，好几个模块才能看出来：代码的可读性是能够立刻从微观上的，一个变量的命名，函数的逻辑划分

宏观层面上的东西固然重要，但是代码的可读性也属于评价代码质量的一个无法让人忽视的指标：它影响了阅读代码的成本（毕竟代码是给人看的）

这里引用《编写可读代码的艺术》这本书里的一句话:

> 对于一个整体的软件系统而言，既需要宏观上的架构决策，设计与指导原则，也必须重视微观上的代码细节。在软件历史中，有许多影响深远的

因此笔者认为代码的可读性可以作为考量一名程序员专业程度的指标。

或许已经有很多同行也正在努力提高自己代码的可读性。然而这里有一个很典型的错觉（笔者之前就有这种错觉）是：越少的代码越容易让人理解。

但是事实上，并不是代码越精简就越容易让人理解。相对于追求最小化代码行数，一个更好的提高可读性方法是**最小化人们理解代码所需要的时间**。

这就引出了这本中的一个核心定理：

> 可读性基本定理：代码的写法应当使别人理解它所需要的时间最小化。

正是这句话深深地吸引了我，于是决定利用这两周的业余时间读完并总结了这本书。

这本书讲的就是关于"如何提高代码的可读性"。
总结下来，这本书从浅入深，在三个层次告诉了我们如何让代码易于理解：

- 表层上的改进：在命名方法（变量名，方法名），变量声明，代码格式，注释等方面的改进。
- 控制流和逻辑的改进：在控制流，逻辑表达式上让代码变得更容易理解。
- 结构上的改进：善于抽取逻辑，借助自然语言的描述来改善代码。

二. 表层的改进

首先来讲最近简单的一层如何改进，涉及到以下几点：

- 如何命名
- 如何声明与使用变量
- 如何简化表达式
- 如何让代码具有美感
- 如何写注释

如何命名

关于如何命名，作者提出了一个关键思想：

> 关键思想：把尽可能多的信息装入名字中。

这里的多指的是有价值的多。那么如何做到有价值呢？作者介绍了以下几个建议：

- 选择专业的词汇，避免泛泛的名字
- 给名字附带更多信息
- 决定名字最适合的长度
- 名字不能引起歧义

选择专业的词汇，避免泛泛的名字

一个比较常见的反例：`get`。

`get`这个词最好是用来做轻量级的取方法的开头，而如果用到其他的地方就会显得很不专业。

举个书中的例子：

`getPage(url)`

通过这个方法名很难判断出这个方法是从缓存中获取页面数据还是从网页中获取。如果是从网页中获取，更专业的词应该是`fetchPage(url)`或者`down`

还有一个比较常见的反例：`returnValue`和`retval`。这两者都是"返回值"的意思，他们被滥用在各个有返回值的函数里面。其实这两个次除了携带他们

那么如何选择一个专业的词汇呢？答案是在非常贴近你自己的意图的基础上，选择一个富有表现力的词汇。

举几个例子：

更为细化功能的单词

- 相对于`make`，选择`create`,`generate`,`build`等词汇会更有表现力，更加专业。
- 相对于`find`，选择`search`,`extract`,`recover`等词汇会更有表现力，更加专业。
- 相对于`retval`，选择一个能充分描述这个返回值的性质的名字，例如：

```
6  var euclidean_norm = function (v){
12     var retval = 0.0;
18     for (var i = 0; i < v.length; i += 1;)
24        retval += v[i] * v[i];
30     return Match.sqrt(retval);
36 }
```

这里的retval表示的是"平方的和"，因此`sum_squares`这个词更加贴切你的意图，更加专业。

但是，有些情况下，泛泛的名字也是有意义的，例如一个交换变量的情景：

```
6  if (right < left){
12     tmp = right;
18     right = left;
24     left = tmp;
30 }
```

像上面这种`tmp`只是作为一个临时存储的情况下，tmp表达的意思就比较贴切了。因此，像`tmp`这个名字，只适用于短期存在而且特性为临时性的变

给名字附带更多信息

除了选择一个专业，贴切意图的词汇，我们也可以通过添加一些前后缀来给这个词附带更多的信息。这里所指的更多的信息有三种：

- 变量的单位
- 变量的属性
- 变量的格式

为变量添加单位

有些变量是有单位的，在变量名的后面添加其单位可以让这个变量名携带更多信息：

- 一个表达时间间隔的变量，它的单位是秒：相对于`duraction`，`ducation_secs`携带了更多的信息
- 一个表达内存大小的变量，它的单位是mb：相对于`size`，`cache_mb`携带了更多的信息。

为变量添加重要属性

有些变量是具有一些非常重要的属性，其重要程度是不允许使用者忽略的。例如：

- 一个UTF-8格式的html字节，相对于`html`，`html_utf8`更加清楚地描述了这个变量的格式。
- 一个纯文本，需要加密的密码字符串：相对于`password`，`plaintext_password`更清楚地描述了这个变量的特点。

为变量选择适当的格式

对于命名，有些既定的格式需要注意：

- 使用大驼峰命名来表示类名：`HomeViewController`。
- 使用小驼峰命名来表示属性名：`userNameLabel`。
- 使用下划线连接词来表示变量名：`product_id`。
- 使用`kConstantName`来表示常量：`kCacheDuraction`。
- 使用MACRO_NAME来表示宏：`SCREEN_WIDTH`。

决定名字最适合的长度

名字越长越难记住，名字越短所持有的信息就越少，如何决定名字的长度呢？这里有几个原则：

- 如果变量的作用域很小，可以取很短的名字
- 驼峰命名中的单元不能超过3个
- 不能使用大家不熟悉的缩写
- 丢掉不必要的单元

如果变量的作用域很小，可以取很短的名字

如果一个变量作用域很小：则可以给它取一个很短的名字也无妨。

看下面这个例子：

```
6  if(debug){
12     map <string,int>m;
18     LookUpNamesNumbers(&m);
24     Print(m);
30 }
```

在这里，变量的类型和使用范围一眼可见，读者可以了解这段代码的所有信息，所以即使是取`m`这个非常简短的名字，也不影响读者来理解作者的意

相反的，如果`m`是一个全局变量，当你看到下面这段代码就会很头疼，因为你不知道它的类型并不明确:

```
6  LookUpNamesNumbers(&m);
12 Print(m);
```

驼峰命名中的单元不能超过3个

我们知道驼峰命名可以很清晰地体现变量的含义，但是当驼峰命名中的单元超过了3个之后，就会很影响阅读体验:

`userFriendsInfoModel`

`memoryCacheCalculateTool`

是不是看上去很吃力？因为我们大脑同时可以记住的信息非常有限，尤其是在看代码的时候，这种短期记忆的局限性是无法让我们同时记住或者瞬
元：

丢掉不必要的单元

有些单元在变量里面是可以去掉的，例如：

`convertToString` 可以省略成 `toString`。

不能使用大家不熟悉的缩写

有些缩写是大家熟知的：

- `doc` 可以代替 `document`
- `str` 可以代替 `string`

但是如果你想用 `BEManager` 来代替 `BackEndManager` 就比较不合适了。因为不了解的人几乎是无法猜到这个名称的意义的。

所以类似这种情况不能偷懒，该是什么就是什么，否则会起到相反的效果。因为它看起来非常陌生，跟我们熟知的一些缩写规则相去甚远。

名字不能引起歧义

有些名字会引起歧义，例如：

- filter：过滤这个词，可以是过滤出符合标准的，也可以是减少不符合标准的：是两种完全相反的结果，所以不推荐使用。
- clip：类似的，到底是在原来的基础上截掉某一段还是另外截出来某一段呢？同样也不推荐使用。
- 布尔值：read_password:是表达需要读取密码，还是已经读了密码呢？所以最好使用 `need_password` 或者 `is_authenticated` 来代替比较好。通常来说，

这一节讲了很多关于如何起好一个变量名的方法。其实有一个很简单的原则来判断这个变量名起的是否是好的：那就是：**团队的新成员是否能迅速**
名字，对谁都好。其实如果你养成习惯多花几秒钟想出个好名字，你会发现你的"命名能力"会很快提升。

如何声明与使用变量

在写程序的过程中我们会声明很多变量（成员变量，临时变量），而我们要知道变量的声明与使用策略是会对代码的可读性造成影响的：

- 变量越多，越难跟踪它们的动向。
- 变量的作用域越大，就需要跟踪它们的动向越久。
- 变量改变的越频繁，就越难跟踪它的当前值。

相对的，对于变量的声明与使用，我们可以从这四个角度来提高代码的可读性：

1. 减少变量的个数
2. 缩小变量的作用域
3. 缩短变量声明与使用其代码的距离
4. 变量最好只写一次

减少变量的个数

在一个函数里面可能会声明很多变量，但是有些变量的声明是毫无意义的，比如：

- 没有价值的临时变量
- 表示中间结果的变量

没有价值的临时变量

有些变量的声明完全是多此一举，它们的存在反而加大了阅读代码的成本。

有些变量的声明完全是多此一举，它们的存在反而加大了阅读代码的成本：

```
let now = datetime.datatime.now()
root_message.last_view_time = now
```

上面这个 `now` 变量的存在是毫无意义的，因为：

- 没有拆分任何复杂的表达式
- `datetime.datatime.now` 已经很清楚地表达了意思
- 只使用了一次，因此而没有压缩任何冗余的代码

所以完全不用这个变量也是完全可以的：

```
root_message.last_view_time = datetime.datatime.now()
```

表示中间结果的变量

有的时候为了达成一个目标，把一件事情分成了两件事情来做，这两件事情中间需要一个变量来传递结果。但往往这件事情不需要分成两件事情来

看一个比较常见的需求，一个把数组中的某个值移除的例子：

```
var remove_value = function (array, value_to_remove){
    var index_to_remove = null;
    for (var i = 0; i < array.length; i+=1){
        if (array[i] === value_to_remove){
            index_to_remove = i;
            break;
        }
    }
    if (index_to_remove !== null){
        array.splice(index_to_remove,1);
    }
}
```

这里面把这个事情分成了两件事情来做：

1. 找出要删除的元素的序号，保存在变量 `index_to_remove` 里面。
2. 拿到 `index_to_remove` 以后使用 `splice` 方法删除它。（这段代码是JavaScript代码）

这个例子对于变量的命名还是比较合格的，但实际上这里所使用的中间结果变量是完全不需要的，整个过程也不需要分两个步骤进行。来看一下如

```
var remove_value = function (array, value_to_remove){
    for (var i = 0; i < array.length; i+=1){
        if (array[i] === value_to_remove){
            array.splice(i,1);
            return;
        }
    }
}
```

上面的方法里面，当知道应该删除的元素的序号 `i` 的时候，就直接用它来删除了应该删除的元素并立即返回。

除了减轻了内存和处理器的负担（因为不需要开辟新的内容来存储结果变量以及可能不用完全走遍整个的for语句），阅读代码的人也会很快领会代

所以在写代码的时候，如果可以“速战速决”，就尽量使用最快，最简洁的方式来实现目的。

缩小变量的作用域

变量的作用域越广，就越难追踪它，值也越难控制，所以我们应该让你的**变量对尽量少的代码可见。**

比如类的成员变量就相当于一个“小型局部变量”。如果这个类比较庞大，我们就会很难追踪它，因为所有方法都可以“隐式”调用它。所以相反地，如

```
//成员变量，比较难追踪
class LargeCass{
  string str_;
```

```
  void Method1(){
     str_ = ...;
     Method2();
  }

  void Method2(){
     //using str_
  }
}
```

降格：

```
//局部变量，容易追踪
class LargeCass{

  void Method1(){
     string str = ...;
     Method2(str);
  }

  void Method2(string str){
     //using str
  }
}
```

所以在设计类的时候如果这个数据（变量）可以通过方法参数来传递，就不要以成员变量来保存它。

缩短变量声明与使用其代码的距离

在实现一个函数的时候，我们可能会声明比较多的变量，但这些变量的使用位置却不都是在函数开头。

有一个比较不好的习惯就是无论变量在当前函数的哪个位置使用，都在一开始（函数的开头）就声明了它们。这样可能导致的问题是：阅读代码的人，声明了好几个变量，也对阅读代码的人的大脑造成了负担，因为人的短期记忆是有限的，特别是记一些暂时还不知道怎么用的东西。

因此，如果在函数内部需要在不同地方使用几个不同的变量，建议在真正使用它们之前再声明它。

变量最好只写一次

操作一个变量的地方越多，就越难确定它的当前值。所以在很多语言里面有其各自的方式让一些变量不可变（是个常量），比如C++里的`const`和Ja

如何简化表达式

有些表达式比较长，很难让人马上理解。这时候最好可以将其拆分成更容易的几个小块。可以尝试下面的几个方法：

- 使用解释变量
- 使用总结变量
- 使用德摩根定理

使用解释变量

有些变量会从一个比较长的算式得出，这个表达式可能很难让人看懂。这时候就需要用一个简短的"解释"变量来诠释算式的含义。使用书中的一个例

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

其实上面左侧的表达式其实得出的是用户名，我们可以用`username`来替换它：

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

使用总结变量

除了以"变量"替换"算式"，还可以用"变量"来替换含有更多变量更复杂的内容，比如条件语句，这时候该变量可以被称为"总结变量"。使用书中的一

```
if(request.user.id == document.owner_id){
```

```
    //do something
}
```

上面这条判断语句所判断的是：“该文档的所有者是不是该用户”。我们可以使用一个总结性的变量`user_owns_document`来替换它：

```
final boolean user_owns_document = (request.user.id == document.owner_id);
if (user_owns_document){
   //do something
}
```

使用德摩根定理

德摩根定理:

1. `not(a or b or c)`等价于`(not a) and (not b) and (not c)`
2. `not(a and b and c)`等价于`(not a) or (not b) or (not c)`

当我们条件语句里面存在外部取反的情况，就可以使用德摩根定理来做个转换。使用书中的一个例子：

```
//使用德摩根定理转换以前
if(!(file_exists && !is_protected)){}

//使用德摩根定理转换以后
if(!file_exists || is_protected){}
```

如何让代码具有美感

在读过一些好的源码之后我有一个感受：好的源码往往都看上去都很漂亮，很有美感。这里说的漂亮和美感不是指代码的逻辑清晰有条理，而是指代码的代码让人赏心悦目，也容易让人读懂。

为了让代码更有美感，采取以下实践会很有帮助：

- 用换行和列对齐来让代码更加整齐
- 选择一个有意义的顺序
- 把代码分成"段落"
- 保持风格一致性

用换行和列对齐来让代码更加整齐

有些时候，我们可以利用换行和列对齐来让代码显得更加整齐。

换行

换行比较常用在函数或方法的参数比较多的时候。

使用换行：

```
- (void)requestWithUrl:(NSString*)url
                method:(NSString*)method
                params:(NSDictionary *)params
               success:(SuccessBlock)success
               failure:(FailuireBlock)failure{

}
```

不使用换行：

```
- (void)requestWithUrl:(NSString*)url method:(NSString*)method params:(NSDictionary *)params success:(SuccessBlock)success failure:(FailuireBlock)fai

}
```

通过比较可以看出，如果不使用换行，就很难一眼看清楚都是用了什么参数，而且代码整体看上去整洁干净了很多。

列对齐

列对齐

在声明一组变量的时候，由于每个变量名的长度不同，导致了在变量名左侧对齐的情况下，等号以及右侧的内容没有对齐：

```
NSString *name = userInfo[@"name"];
NSString *sex = userInfo[@"sex"];
NSString *address = userInfo[@"address"];
```

而如果使用了列对齐的方法，让等号以及右侧的部分对齐的方式会使代码看上去更加整洁：

```
NSString *name    = userInfo[@"name"];
NSString *sex     = userInfo[@"sex"];
NSString *address = userInfo[@"address"];
```

这二者的区别在条目数比较多以及变量名称长度相差较大的时候会更加明显。

选择一个有意义的顺序

当涉及到相同变量（属性）组合的存取都存在的时候，最好以一个有意义的顺序来排列它们：

- 让变量的顺序与对应的HTML表单中<input>字段的顺序相匹配
- 从最重要到最不重要排序
- 按照字母排序

举个例子：相同集合里的元素同时出现的时候最好保证每个元素出现顺序是一致的。除了便于阅读这个好处以外，也有助于能发现漏掉的部分，尤

```
//给model赋值
model.name    = dict["name"];
model.sex     = dict["sex"];
model.address = dict["address"];

 ...

//拿到model来绘制UI
nameLabel.text    = model.name;
sexLabel.text     = model.sex;
addressLabel.text = model.address;
```

把代码分成"段落"

在写文章的时候，为了能让整个文章看起来结构清晰，我们通常会把大段文字分成一个个小的段落，让表达相同主旨的语言凑到一起，与其他主旨

而且除了让读者明确哪些内容是表达同一主旨之外，把文章分为一个个段落的好处还有便于找到你的阅读"脚印"，便于段落之间的导航；也可以让你

其实这些道理同样适用于写代码：如果你可以把一个拥有好几个步骤的大段函数，以空行+注释的方法将每一个步骤区分开来，那么则会对读者理解
实可读性又何尝不是来自于规则，富有美感的代码呢？

```
BigFunction{

    //step1:*****
    ....

    //step2:*****
    ...

    //step3:*****
    ....

}
```

保持风格一致性

有些时候，你的某些代码风格可能与大众比较容易接受的风格不太一样。但是如果你在你自己所写的代码各处能够保持你这种独有的风格，也是可

比如一个比较经典的代码风格问题：

```
if(condition){
```

```
12
18 }
```

or:

```
6  if(condition)
12 {
18
24 }
```

对于上面的两种写法，每个人对条件判断右侧的大括号的位置会有不同的看法。但是无论你坚持的是哪一个，请在你的代码里做到始终如一。因为

我们要知道，一个逻辑清晰的代码也可以因为留白的不规则，格式不对齐，顺序混乱而让人很难读懂，这是十分让人痛心的事情。所以既然你的代

如何写注释

首先引用书中的一句话：

> 注释的目的是尽量帮助读者了解得和作者一样多。

在你写代码的时候，在脑海中可能会留下一些代码里面很难体现出来的部分：这些部分在别人读你的代码的时候可能很难体会到。而这些"不对称"

想要写出好的注释，就需要首先知道：

- 什么不能作为注释
- 什么应该作为注释

什么不能作为注释

我们都知道注释占用了代码的空间，而且实际上对程序本身的运行毫无帮助，所以最好保证它是**物有所值的。**

不幸的是，有一些注释是毫无价值的，它无情的占用了代码间的空间，影响了阅读代码的人的阅读效率，也浪费了写注释的人的时间。这样的注释

- 描述能立刻从代码自身就能立刻理解的代码意图的注释
- 给不好的命名添加的注释

描述能立刻从代码自身就能立刻理解的代码意图的注释

```
6  //add params1 and params2 and return sum of them
12 - (int)addParam1:(int)param1 param2:(int)param2
```

上面这个例子举的比较简单，但反映的问题很明显：这里面的注释是完全不需要的，它的存在反而增加了阅读代码的人的工作量。因为他从方法名

给不好的命名添加的注释

```
6  //get information from internet
12 - (NSString *)getInformation
```

该函数返回的是从网络获取的信息。但这里使用了get前缀，无法看出信息的来源。为了补充信息，使用注释来弥补。但其实这完全不必要。只要取

```
6 - (NSString *)fetchInformation
```

讲完了注释不应该是什么内容，现在讲一下注释应该是什么样的内容：

什么应该作为注释

本书中介绍的注释大概有以下几种：

- 写代码时的思考

- 对代码的评价
- 常量
- 全局观的概述

写代码时的思考

你的代码可能不是一蹴而就的，它的产生可能会需要一些思考的过程。然而很多时候代码本身却无法将这些思考表达出来，所以你就可能有必要通过
让读者理解了**这段代码为什么这么写**。如果遇到了比你高明的高手，在他看到你的注释之后兴许会马上设计出一套更加合适的方案。

对代码的评价

有些时候你知道你现在写的代码是个**临时的方案**：它可能确实是解决当前问题的一个方法，但是：

- 你知道同时它也存在着某些缺陷，甚至是陷阱
- 你不知道有其他的方案可以替代了
- 你知道有哪个方案可以替代但是由于时间的关系或者自身的能力无法实现

也可能你知道你现在实现的这个方案几乎就是"完美的"，因为如果使用了其他的方案，可能会消耗更多的资源等等。

对于上面这些情况，你都有必要写上几个字作为注释来诚实的告诉阅读你的这段代码的人这段代码的情况，比如：

```
//该方案有一个很容易忽略的陷阱：****
//该方案是存在性能瓶颈，性能瓶颈在其中的**函数中
//该方案的性能可能并不是最好的，因为如果使用某某算法的话可能会好很多
```

常量

在定义常量的时候，在其后面最好添加一个关于它是什么或者为什么它是这个值的原因。因为常量通常是不应该被修改的，所以最好把这个常量为

例如：

```
image_quality = 0.72 // 最佳的size/quanlity比率
retry_limit   = 4    // 服务器性能所允许的请求失败的重试上限
```

全局观的概述

对于一个刚加入团队的新人来说，除了团队文化，代码规范以外，可能最需要了解的是当前被分配到的项目的一些"全局观"的认识：比如组织架构，

有时仅仅添加了几句话，可能就会让新人迅速地了解当前系统或者当前类的结构以及作用，而且这些也同样对开发过当前系统的人员迅速回忆出之

这些注释可以在一个类的开头（介绍这个类的职责，以及在整个系统中的角色）也可以在一个模块入口处。书中举了一个关于这种注释的例子：

```
//这个文件包含了一些辅助函数，尾门的文件系统提供了更便利的接口
```

再举一个iOS开发里众所周知的网络框架`AFNetworking`的例子。在`AFHTTPSessionManager`的头文件里说明了这个类的职责：

```
//AFHTTPSessionManager` is a subclass of `AFURLSessionManager` with convenience methods for making HTTP requests. When a `baseURL` is provided, reque
```

在知道了什么不应该是注释以及什么应该是注释以后，我们来看一下一个真正合格的注释应该是什么样子的：

> 注释应当有很高的信息/空间率

也就是说，注释应该用最简短的话来最明确地表达。要做到这一点需要做的努力是：

- **让注释保持紧凑**：尽量用最简洁的话来表达，不应该有重复的内容
- **准确地描述函数的行为**：要把函数的具体行为准确表达出来，不能停留在表明
- **用输入/输出的例子来说明特别的情况**：有时相对于文字，可能用一个实际的参数和返回值就能立刻体现出函数的作用。而且有些特殊情况也可以
- **声明代码的意图**：也就是说明这段代码存在的意义，你为什么当时是这么写的原因

其实好的代码是自解释的，由于其命名的合理以及架构的清晰，几乎不需要注释来向阅读代码的人添加额外的信息，书中有一个公式可以很形象地表

> 好代码 > (坏代码 + 注释)

三. 控制流和逻辑的改进

控制流在编码中占据着很重要的位置，它往往代表着一些核心逻辑和算法。因此，如果我们可以让控制流变得看上去更加"自然"，那么就会对阅读代

那么都有哪相关实践呢?

- 使用符合人类自然语言的表达习惯
- if/else语句块的顺序
- 使用return提前返回

使用符合人类自然语言的表达习惯

写代码也是一个表达的过程，虽然表现形式不同，但是如果我们能够采用符合人类自然语言习惯的表达习惯来写代码，对阅读代码的人理解我们的代

这里有两个比较典型的情景：

1. 条件语句中参数的顺序
2. 条件语句中的正负逻辑

条件语句中参数的顺序：

首先比较一下下面两段代码，哪一个更容易读懂?

```
6  //code 1
12 if(length > 10)
18
24 //code 2
30 if(10 < length)
```

大家习惯上应该会觉得code1容易读懂。

再来看下面一个例子：

```
6  //code 3
12 if(received_number < standard_number)
18
24 //code 4
30 if( standard_number< received_number)
```

仔细看会发现，和上面那一组情况类似，大多数人还是会觉得code3更容易读懂。

那么code1 和 code3有什么共性呢?

它们的共性就是：**左侧都是被询问的内容（通常是一个变量）；右侧都是用来做比较的内容（通常是一个常量）**

这应该是符合自然语言的一个顺序。比如我们一般会说"今天的气温大于20摄氏度"，而不习惯说"20摄氏度小于今天的气温"。

条件语句中的正负逻辑：

在判断一些正负逻辑的时候，建议使用`if(result)`而不是`if(!result)`。

因为大脑比较容易处理正逻辑，比如我们可能比较习惯说“某某某是个男人”，而不习惯说“某某某不是个女人”。如果我们使用了负逻辑，大脑还要对

if/else语句块的顺序

在写if/else语句的时候，可能会有很多不同的互斥情况（好多个`elseif`）。那么这些互斥的情况可以遵循哪些顺序呢？

- **先处理掉简单的情况，后处理复杂的情况**：这样有助于阅读代码的人循序渐进地地理解你的逻辑，而不是一开始就吃掉一个胖子，耗费不少精力
- **先处理特殊或者可疑的情况，后处理正常的情况**：这样有助于阅读代码的人会马上看到当前逻辑的边界条件以及需要注意的地方。

使用return提前返回

在一个函数或是方法里，可能有一些情况是比较特殊或者极端的，对结果的产生影响很大（甚至是终止继续进行）。如果存在这些情况，我们应该把

这样做的好处是可以减少if/else语句的嵌套，也可以明确体现出：“哪些情况是引起异常的”。

再举一个`JSONModel`里的例子，在`initWithDictionary:error`方法里面就有很多return操作，它们都体现出了“在什么情况下是不能成功将字典转化为mod
过了层层考验：

```
6   -(id)initWithDictionary:(NSDictionary*)dict error:(NSError**)err
12  {
18      //check for nil input
24      if (!dict) {
30          if (err) *err = [JSONModelError errorInputIsNil];
36          return nil;
42      }
48
54      //invalid input, just create empty instance
60      if (![dict isKindOfClass:[NSDictionary class]]) {
66          if (err) *err = [JSONModelError errorInvalidDataWithMessage:@"Attempt to initialize JSONModel object using initWithDictionary:error: but the
72          return nil;
78      }
84
90      //create a class instance
96      self = [self init];
102     if (!self) {
108
114         //super init didn't succeed
120         if (err) *err = [JSONModelError errorModelIsInvalid];
126         return nil;
132     }
138
144     //check incoming data structure
150     if (![self __doesDictionary:dict matchModelWithKeyMapper:self.__keyMapper error:err]) {
156         return nil;
162     }
168
174     //import the data from a dictionary
180     if (![self __importDictionary:dict withKeyMapper:self.__keyMapper validation:YES error:err]) {
186         return nil;
192     }
198
204     //run any custom model validation
210     if (![self validate:err]) {
216         return nil;
222     }
228
234     //model is valid! yay!
240     return self;
246 }
```

四. 代码组织的改进

关于代码组织的改进，作者介绍了以下三种方法:

- 抽取出与程序主要目的“不相关的子逻辑”

- 重新组织代码使它一次只做一件事情
- 借助自然语言描述来将想法变成代码

抽取出与程序主要目的“不相关的子逻辑”

一个函数里面往往包含了其主逻辑与子逻辑，我们应该积极地发现并抽取出与主逻辑不相关的子逻辑。具体思考的步骤是：

1. 首先确认这段代码的高层次目标是什么（主要目标）？
2. 对于每一行代码，都要反思一下：“它是直接为了目标而工作么？”
3. 如果答案是肯定的并且这些代码占据着一定数量的行数，我们就应该将他们抽取到独立的函数中。

比如某个函数的目标是**为了寻找距离某个商家最近的地铁口**，那么这其中一定会重复出现一些计算两组经纬度之间距离的子逻辑。但是这些子逻辑
该是无关的。

即是说，像这种类似于工具方法的函数其实是脱离于某个具体的需求的：它可以用在其他的主函数中，也可以放在其他的项目里面。比如**找到离运

而像这种“抽取子逻辑或工具方法”的做法有什么好处呢？

- 提高了代码的可读性：将函数的调用与原来复杂的实现进行替换，让阅读代码的人很快能了解到该子逻辑的目的，让他们把注意力放在更高层的
- 便于修改和调试：因为一个项目中可能会多次调用该子逻辑（计算距离，计算汇率，保留小数点），当业务需求发生改变的时候只需要改变这一
- 便于测试：同理，也是因为可以被多次调用，在进行测试的时候就比较有针对性。

从函数扩大到项目，其实在一个项目里面，有很多东西不是当前这个项目所专有的，它们是可以用在其他项目中的一些“通用代码”。这些通用代码可

我们可以养成这个习惯，“把一般代码与项目专有代码分开”，并不断扩大我们的通用代码库来解决更多的一般性问题。

重新组织代码使它一次只做一件事情

一个比较大的函数或者功能可能由很多任务代码组合而来，在这个时候我们有必要将他们分为更小的函数来调用它们。

这样做的好处是：我们可以清晰地看到这个功能是如何一步一步完成的，而且拆分出来的小的函数或许也可以用在其他的地方。

所以如果你遇到了比较难读懂的代码，可以尝试将它所做的所有任务列出来。可能马上你就会发现这其中有些任务可以转化成单独的函数或者类。

借助自然语言描述来将想法变成代码

在设计一个解决方案之前，如果你能够用自然语言把问题说清楚会对整个设计非常有帮助。因为如果直接从大脑中的想法转化为代码，可能会露掉

但是如果你可以将整个问题和想法滴水不漏地说出来，就可能会发现一些之前没有想到的问题。这样可以不断完善你的思路和设计。

五. 最后想说的

这本书从变量的命名到代码的组织来讲解了一些让代码的可读性提高的一些实践方法。

其实笔者认为代码的可读性也可以算作是一种沟通能力的一种体现。因为写代码的过程也可以被看做是写代码的人与阅读代码的人的一种沟通，只
确，高效地把自己的思考和工作内容以代码的形式表述出来。
所以笔者相信能写出可读性很高的代码的人，TA对于自己的思考和想法的描述能力一定不会很差。

如果你真的打算好好做编程这件事情，建议你从最小的事情上做起：好好为你的变量起个名字。不要再以“我英语不好”或者“没时间想名字”作为托

如果你连起个好的变量名都懒得查个字典，那你怎么证明你在遇到更难的问题的时候能够以科学的态度解决它？
如果你连编程里这种最小的事情都不好好做，那你又怎么证明你对编程是有追求的呢？

本文已经同步到我的个人博客：传送门