写在前面的话：

这周末我一个同学在群上说找到一篇挺有意思的文章（就是下面要说的可读性代码的心理学），说要翻译出来，我就主动请缨了，跟他合作翻译这篇文章，在看这篇文章的同时，我突然间想到，为什么代码的可读性那么多人重视呢？当然我也认为代码的可读性很重要，能给我们的协作开发带来好处。我突然联想起我前一段时间在工作之余看的一本书，叫《人类简史》，它在介绍人类这一物种的历史的同时，也对人类为什么能成为地球霸主（位于食物链顶端）做出了解释，它认为其中一个原因就是，人类演化出想象的能力，人类能脸不红，心不跳的说出他从未见过的东西。认为它真的存在那样。比如说，神、科幻小说里说的技术、以及那些概念（国家、主权、科学、民主以及各种主义）。并且让所有人相信，利用这些想象的概念建立了能让陌生人也能合作的框架。重点是合作。这种几百万人为了同一个目标而奋斗的合作能力，这让人类能够战胜其他物种的原因之一。再往下说就离题了，我想强调的是合作能力的重要性，它让我们人类达成了今天这样的成就，我想把它搬到我们公司研发的身上，就是说，个体能力可以不强，但是协作开发的能力一定要强，怎么提高协作开发的能力呢？方法之一就是代码的可读性，我认为代码的可读性是我们协作的基础，代码都看不懂，协作从何谈起。又怎么提高我们研发的生产效率呢？所以我想把下面这篇文章介绍给大家。

原文：<https://medium.com/@egonelbre/psychology-of-code-readability-d23b1ff1258a>

代码可读性心理学

Psychology of Code Readability

By no means should this be regarded as truth, but rather a model that I’ve found extremely helpful in understanding and finding better ways of writing code.

我发现了一个很有用的模型，这个模型虽然不是真理，但是却非常能够帮助我理解和编写出更好的代码。

I think one of the things every programmer strives for is writing better code. Readability is one of the aspects of “good code”. There have been many papers and books written on the topic, however I find many of them lacking. Not because of the recommendations, but rather the analysis part.

写出更好的代码是每个程序员都在努力的目标。代码的可读性是“好代码”的一个判断标准之一。关于这个主题的论文和书已经有非常多了，然而我发现他们都存在缺点。不是因为给出的那些建议不好，而是他们都少了分析的那一部分。

（我的理解：就是说，市面上的书都是说该怎么怎么做，并没有说出为什么怎么做。）

What makes some piece of code more readable than another? It’s one thing to say that it uses better variable names, but what makes a certain variable name easier to read? I really mean digging deeper into human psyche. It is our brain that is doing all the processing after all.

到底是什么东西，让有些代码的可读性就是比另外一些代码强？有一种说法是更好的变量名，可究竟是什么东西让变量名更易读？我的意思是要深入到人类的心里层面，毕竟我们的大脑接管了所有的处理过程。

心理学入门

Psychology Primer

As any programmer knows we have limited capacity to think about things. This is our [working memory limit](https://en.wikipedia.org/wiki/Working_memory" \l "Capacity). There’s an old myth going around that we can hold 7±2 objects in our head. It is known as [“The Magical Number Seven”](https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two) and it isn’t entirely accurate. This number has been refined to 4±1 and some even suggest there isn’t a limit, but [rather a degradation of ideas over time](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2974097/). For all intents and purposes we can assume that we have a small number of ideas we can process in our head at a given time. The exact number isn’t that important.

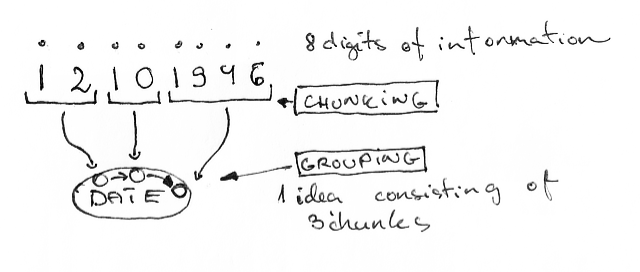
任何程序员都知道，我们的思考容量是有限的。这就是我们的运行内存限制。有一个古老的传说，相传我们的大脑可以容纳 7±2 个物体，它被称为“神奇的 7 号”，不过这并不准确，这个数字有点限制到 4±1，而有些建议则认为其没有受到限制，而是随着时间推算思想受到了退化。出于所有意图和目的，我们可以假设我们的大脑在给定的时间内，能处理的东西只是一个很小的数字，具体是数字是多少并不重要。

But some would still confidently say that they can handle problems involving more than 4 ideas. Luckily there’s another process going on in our brain called [chunking](https://en.wikipedia.org/wiki/Chunking_%28psychology%29). Our brain automatically groups information pieces into larger pieces (chunks).

但有些人还是坚持认为他们能够同时处理超过 4 个主意。幸运的是，我们的大脑中正在进行另一个叫做分块的过程。 我们的大脑会自动将信息片段分成更大的片段（块）。

Dates and phone-numbers are good examples of this:

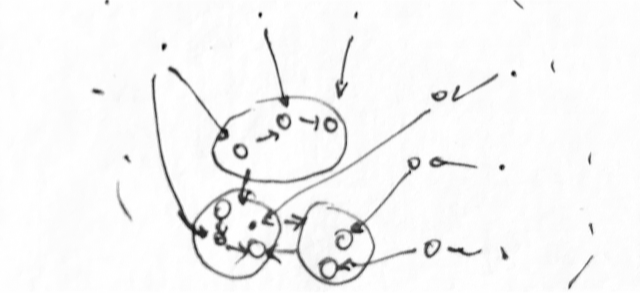
日期和电话号码就是很好的例子：



From these **chunks** we build up our **long term memory**. I like to imagine it as a large web of consisting many chunks, chunk sequences and groupings.

通过这些信息块，我们建立起我们的长期记忆。我喜欢把它们想象成一个由许多块、块序列和块分组组成的大网络。

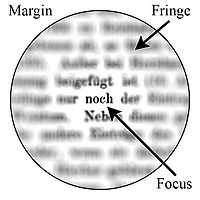
（分块记忆，举例，电话号码的分段记忆。由此推出，代码的方法编写，一个方法只做一件事，基于大脑用分块来储存信息来解释为什么这么做）



You might guess from this image that moving from one place to another in memory is slow. And you would be right. In UX there’s a concept called singular [focus of attention](https://en.wikipedia.org/wiki/Attention). Which means that we can focus at a single thing at a time. It also has a friend called [locus of attention](http://www.usabilityfirst.com/glossary/locus-of-attention/), which says that our attention is also localized in space.

你可以从这张图片得出一个结论，记忆块之间的信息交流是很慢的。你是对的，在 UX 领域中，这个结论被称为单一关注焦点理论。也就是说，我们一次只能关注单一的事物。也可以说我们只能有一个关注点，我们关注的地方也只能是个局部的。

（也就是，在一堆信息中突然插入一段不相关的信息，大脑会花时间建立联系，由此推出一个方法只做一件事！！）



You might think this is the same thing as working memory limit, however there is a slight difference. **Working memory capacity** talks how big our focusing area is, the **focus/locus of attention** say that we can only do that when there is a place in our brain that contains the ideas.

同理，我们人脑的运行的内存也是有限的。然而，有一个细微的不同在于，我们的人脑内存有多大，聚焦的范围也就能有多大。单一关注焦点理论也说明了只有我们大脑中有存在一个包含这些想法的地方时，大脑才能正常工作。



The **focus and locus of attention** are important to know, because [switching cost](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3197943/) is significant. It is even slower when we need to create new **chunks** and **groupings**. It also goes the other way, the more familiar something is the less time it takes to make it our focus.

认识焦点和关注点非常重要，因为切换他们的成本是非常高的。我们人脑创建一个认知事物的块和分组是很慢的。同样的，如果事物之间相似度很高，创建分组的时间就会缩短，那我们就能更快的把我们的关注点聚焦到另一个事物。

（也就是，在一堆信息中突然插入一段不相关的信息，大脑会花时间建立联系，由此推出一个方法只做一件事！！这段话也是说明这个的）

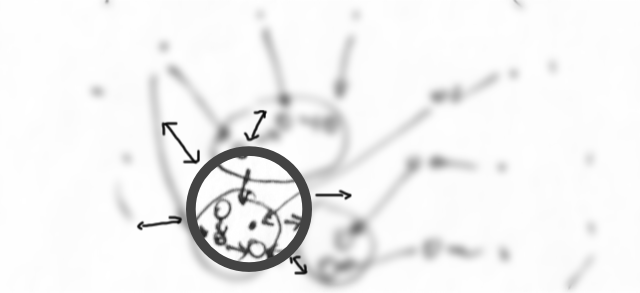
We also remember things better when we are in a similar context. This is called [encoding specificity principle](https://en.wikipedia.org/wiki/Encoding_specificity_principle). This means by designing our encoding and recalling conditions we can design better what we remember.

当我们处在类似的环境时，我们也能更好地记住事情。这被称为编码特异性原则。这意味着，通过设计编码和回忆条件，我们可以更好地设计我们记忆中的内容。

**译者注：**情境相似性是心理学中编码特异性原则，描述的是，当回忆时的背景与识记时的背景相匹配时，记忆效果最好。触景生情，睹物思人。

[In an experiment](https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2044-8295.1975.tb01468.x) divers were assigned to memorize words on land and under water. Then recall them on land or in water. The best results were for people who memorized and recalled on land. Surprisingly the second best were the people that memorized and recalled on water. This showed that the context where you learn things has an impact on how well you can remember things.

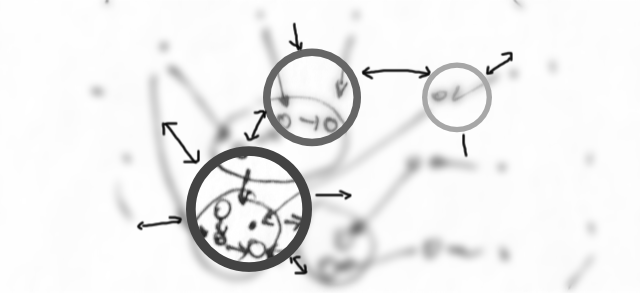
在一个实验中，潜水员被分配在陆地和水下记忆单词。然后在陆地或水中回忆它们。最好的结果是那些在陆地上记忆和回忆的人。令人惊讶的是，第二好的是那些在水中记忆和回忆的人。这表明，你学习事物的环境会影响你对事物的回忆能力。



To make things shorter, I’ll use **context** to refer to “focus and locus of attention” and how it relates to other chunks and loci. Effectively our brain is moving from one **context** to another. When we move our focus of attention we also remember what our previous contexts were, until our memory fades.

为了缩短篇幅，我会用上文来指代关注点以及它与其它块的联系，以及他们之间是如何联系的。我们的大脑承上启下的能力是挺强的，当我们转移注意力的时候，我们依旧能够记住上文出现的内容，直到我们的记忆变淡

**译者注：**这样的描述给我的感觉非常像进程和线程在竞争CPU的样子。



From these contexts and chunks we build up [mental representations](https://en.wikipedia.org/wiki/Mental_representation) and [a mental model](https://en.wikipedia.org/wiki/Mental_model). There’s a slight difference between these two things. **Mental representation** is our internal cognitive symbol for representing the external world or a mental processes. **Mental model** can be thought of as a explanation of a mental representation. Often these terms are used interchangeably.

根据这些上下文和块，我们可以构建出[心理表征](https://en.m.wikipedia.org/wiki/Mental_representation" \t "_blank)和一个[心理模型](https://en.m.wikipedia.org/wiki/Mental_model" \t "_blank)，两者之间存在着细微的不同。心里表征是我们内在的认知中对外部世界或者心里过程的符号。心理模型可以被认为是心理表征的解释。在大多数情况下，这些术语通常可以互换使用。

Mental models have a vital importance in our ability to precisely describe a solution to a problem. There are many different mental models possible for a single problem each having their own benefits and problems.

心理模型对于我们精确描述问题的解决方案的能力至关重要。对于一个问题，有许多不同的心理模型，每种模型都有各种好处和问题。

All of these ideas sound nice and precise, however our brains are quite imprecise. There are many other problems with our brain.

所有这些想法听起来都很准确，但是我们的大脑是非常不精确的。我们的大脑还有很多其它的问题。

Our brains need to do more work when dealing with **abstractions**.

我们的大脑在处理抽象概念时需要做更多的努力。

When ideas are **similar** their chunks are related and linked in our brains in a similar way. This leads to our brain being unable to “rebuild the contexts properly” because we are **uncertain** which chunk is the right one. Example: I and 1; O and 0.

当一些想法相似，我们把新想法以相似的方法建立起区块并与已存在的区块建立联系并连接这导致了我们的大脑不能正确地“重构上文说的结论”（context）因为当新区块与就区块起冲突，我们不确定哪个是对的。比如说 l 和 1 ， O 和 0。

**Ambiguity** is another source for **uncertainty**. When a thing is ambiguous then there are multiple interpretations for the same thing. Homonyms are the best example of this property. Example: Crane — the bird or the machine.

歧义是不确定性的另一个来源。当一件事模棱两可时，对同一件事情就会有多种解释。同义词是此属性的最佳示例。例如：Crane-意思可能是鹤，也可能是起重机。

（起变量名不要有歧义！，原因下面有解释）

**Uncertainty** causes us to slow down. It might be a few milliseconds, but that can be enough to disrupt our state of flow or make us use more working memory than necessary.

不确定性会让减缓我们的速度。这可能只有几毫秒的时间，但是却足够打乱我们的状态或者让我们使用更多的工作内存。

There are of course interruptions that can disrupt our working memory, but there are also “smaller interruptions” called **noise**. If someone is saying random numbers and you are trying to calculate, then we can end-up accidentally start processing them and use up some of our **working memory**. This can happen also visually on screen when there are many irrelevant things between the important things.

当然，中断可以打断我们的工作记忆，但是还有一些小中断被称为噪声。如果一个人在说一些随机数，你试着对这些数字做计算，我们最终会意外得停止，因为这个处理过程会耗尽我们的一些工作记忆。当重要事物之间存在许多不相关的事物时，这也可以在屏幕上直观地发生。

（同样这言论也可以对一个方法只做一件事做出解释，不相干的事情会占用我们大脑的工作内存，内存满了就宕机了）

Our brains also have trouble **processing negation**, with support from many studies. The effect of negation depends [on the context](https://langcog.stanford.edu/papers/NF-cogsci2014.pdf), but negation should be used with care.

在许多研究表明，我们的大脑也难以处理否定。否定的影响取决于上下文，否定应谨慎使用。

All of these together add up to [cognitive load](https://en.wikipedia.org/wiki/Cognitive_load). It is the total amount of mental effort being used. Our **processing capacity** decreases with prolonged cognitive load and it is restored with rest. With prolonged cognitive load our minds also start to wander.

所有这些共同增加了认知负荷。认知负荷是被使用的精力的总量。长期的认知负荷会使我们处理能力下降，这通过休息来恢复。长期的认知负荷，也会使得我们的大脑开始走神。

（小休五到十分钟，番茄时间之类的）

**译者注：**认知负荷理论假设人类的认知结构由工作记忆和长时记忆组成。其中工作记忆也可称为短时记忆，它的容量有限，一次只能存储3-5条基本信息或信息块。当要求处理信息时，工作记忆一次只能处理2-3条信息，因为存储在其中的元素之间的交互也需要工作记忆空间，这就减少了能同时处理的信息数。

（一个方法只做一件事）

If this is new information to you, then I highly suggest taking a break now. These form fundamental properties that code analysis will rest upon.

如果这对你来说是新信息，我强烈建议现在休息一下。这些构成了代码分析所依赖的基本条件。

-------------------------------------------------

I’m going to use the term **programming artifact**. By that I mean everything that is created as a result of programming. It might be a method you write, type declarations for a function, variable names, comments, Unreal Engine Blueprints, UML diagrams etc. Effectively anything that is a direct result of programming.

我将使用编程工件这个术语。指代跟编程相关的所有内容。它可能是您编写的方法、函数的类型声明、变量名称、注释、虚幻的引擎蓝图（**Unreal Engine Blueprints**）、UML图等。实际上就是编程的直接结果。

Here are a few recommendations, rules-of-thumb and paradigms analyzed in the context of psychology. By no means is this an exhaustive list or even a guide on what exactly to do. Probably there are many places where the analysis could be better, but this is more about showing how we can gain deeper insight into code readability by using psychology.

这里有一些在心理学背景下分析的建议，经验法则和范例。这绝不是一份详尽的清单，甚至也不是关于究竟要做什么的指南。有很多地方可能还能做得更好，但更多的是展示如何使用心理学来深入了解代码可读性。

名称的范围

Scope of a name

Length is not a virtue in a name; clarity of expression is. — Rob Pike

长度不是名字中的优点，表达的清晰度是。—罗布·派卡

Let’s take a simple for loop:

让我们使用一个简单的for循环：

**A.** for(i=0 to N)

**B.** for(theElementIndex=0 to theNumberOfElementsInTheList)

（我在工作中经常使用长名字，太长确实也要花时间看）

Most programmers would recommend A. Why?

大多数程序员会推荐A。为什么呢？

**B.** uses longer names which prevents us from recognizing this as a single chunk. The longer name also doesn’t help creating a better context, effectively it is just noise.

**B.** 使用较长的名称，这使我们无法将其识别为单个块。更长的名字也无助于创建一个更好的上下文，实际上它只是一个噪音。

However, let’s imagine different ways of writing packages / units / modules / namespaces:

但是，让我们想象一下编写包/单元/模块/命名空间的不同方式：

**A.** strings.IndexOf(x, y)

**B.** s.IndexOf(x, y)

**C.** std.utils.strings.IndexOf(x, y)

**D.** IndexOf(x, y)

In example **B.** the namespace s is too short and doesn’t help “to find the right chunk”.

在例子**B.** 中命令空间s太短，不能帮助“找到正确的信息块”。

In example **C.** the namespace std.utils.strings is too long, most of it’s unnecessary, because strings itself is descriptive enough. (Unless you need to use multiple of them).

在例子**C.** 中，命名空间std.utils.strings太长，大部分都是不必要的，因为strings本身具有足够的描述性（除非你需要使用其中的多个）。

In example **D.** without namespaces, then the call becomes ambiguous, you might be unsure where the IndexOf comes from and what it is related to.

在例子**D.** 中，如果没有命名空间，那么调用就变的模棱两可，你可能无法确定IndexOf来自任何以及处理它与什么相关。

（就是说，IndexOf这个方法不知道是干嘛用的）

It’s important to mention that, if all of code is dealing with strings it will be quite easy to assume that IndexOf is some string related function. In such cases, even the strings part might be too noisy. For example: int16.Add(a, b) compared to a + b, would be much harder to read.

需要注意的是，如果所有代码都在处理字符串，那么很容易假定IndexOf是一些与字符串相关的函数。在这种情况下，甚至strings部分也可能太嘈杂了。例如：int16.Add(a, b)比a + b更难以阅读。

（变量名没有统一的说明要如何做，也就是说受人的主观意愿的影响大，有些人觉得这样足够了，有些人不认为。所以，我认为在我们团队内部统一变量命名规则会很好。）

变量状态

State of a variable

With variables it would be easy to conclude that “modification is bad, because it makes harder to track what is happening”. But, lets take these examples:

对于变量，很容易得出这样的结论：“修改是不好的，因为它使跟踪正在发生的事情变得更加困难”。但是，让我们以以下例子为例：

// **A.**

func foo() (int, int) {

sum, sumOfSquares := 0, 0

for \_, v := range values {

sum += v

sumOfSquares += v \* v

}

return sum, sumOfSquares

}

// **B.**

func GCD(a, b int) int {

for b != 0 {

a, b = b, a % b

}

return a

}

// **C.**

func GCD(a, b int) int {

if b == 0 {

return a

}

return GCD(b, a % b)

}

Here foo is probably easiest to understand. Why? The problem isn’t modifying the variables, but rather how they are modified. **A** doesn’t have any complex interactions, which both **B** and **C** do. I would also guess, that even though **C** doesn’t have modifications, our brain still processes it as such.

在这里，foo可能是最容易理解的。为什么呢？问题不是修改变量，而是如何修改它们。**A**不存在任何复杂的相互作用，**B**和**C**都存在。我也会猜测，即使**C**没有修改，我们的大脑仍然是这样处理它的。

// **D.**

sum = sum + v.x

sum = sum + v.y

sum = sum + v.z

sum = sum + v.w

// **E.**

sum1 = v.x

sum2 := sum1 + v.y

sum3 := sum2 + v.z

sum4 := sum3 + v.w

Here is another example where the modification based version (**D**) is easier to follow. **E** introduces new variables for the same idea, effectively, the different variables become noise.

这里是另一个示例，其中基于修改的版本（**D**）更容易理解。**E**为相同的思想引入新的变量，有效地将不同的变量转换为噪声。

惯用语法

Idioms

Let’s take another for loop:

让我们再来一次for循环：

**A.** for(i = 0; i < N; i++)

**B.** for(i = 0; N > i; i++)

**D.** for(i = 0; i <= N-1; i += 1)

**C.** for(i = 0; N-1 >= i; i += 1)

How long did it take for you to figure out what each line is doing? For anyone who has been coding for a while, **A** probably took the least time. Why is that?

你花了多长时间才弄清楚每一行都做了什么？对于任何已经编程了一段时间的人来说，**A**可能花的时间最少的。为什么会这样呢？

The main reason is **familiarity**. To be more precise, we have a **chunk** in our long-term-memory for **A**, however not for any of the others. This means that we need to do more processing, before we can extract the meaning and concept from it.

主要原因是**熟悉**。更准确的说，我们的长期记忆中有一块关于A的信息块，而不是其他的。这意味着我们需要做更多的处理，然后才能从中提取含义和概念。

（大家都知道的一些做法）

For any complete beginner, all of these would be processed quite similarly. They wouldn’t notice that one is “better” than any other.

对于任何一个完全的初学者来说，所有这些都会被处理得非常相似。他们不会注意到一个比其它任何一个都“更好”。

A proficient programmers reads **A** as a single chunk or idea “i is looped for N items”. However a beginner reads this as “We initialize i to zero. Then we test whether each time we are still smaller than N. Then we add one to i.”

熟练的程序员将**A**理解为“i的N次循环”。但是初学者认为这是“我们初始化为零。然后每次循环都测试i是否比N小。然后我们在i中加1。”

**A** is what you call the “idiomatic way” of writing the for loop. It’s not really better in terms of intrinsic complexity. However, most programmers can read it more easily, because it is part of our common vocabulary.

**A**是你所称为for循环的“惯用方式”。就内在的复杂性而言，这并不是真的更好。但是，大多数程序员可以更容易地阅读它，因为它是我们常用词汇表的一部分。

Most languages have an idiomatic way of writing things. There are even papers and books about them, starting with [APL idioms](http://www.softwarepreservation.org/projects/apl/Papers/MYFAVORITEIDIOM), [C++ idioms](http://www.dre.vanderbilt.edu/~sutambe/documents/More%20C++%20Idioms.pdf) and more structural idioms like in [GoF Design Patterns](https://en.wikipedia.org/wiki/Design_Patterns). These books can be regarded as a vocabulary for writing sentences and paragraphs, such that it will be recognized by people.

大多数语言都有一种惯用的写作方式。甚至还有关于它们的论文和书籍，从APL惯用语法、C++惯用语法和像是在GoF设计模式中的更加结构化的惯用语法。这些书可以看作是写句子和段落的词汇，这样才能被人们所认可。

There’s however a downside to all of this. The more idioms there are, the bigger vocabulary you have to have to understand something. Languages with unlimited flexibility often suffer due to this. People end up creating “idioms” that help them write more concise code, however everybody else will be slowed down by them.

然而，所有这些都有不利的一面。惯用语法越多，不得不去理解的词汇量就越大。具有高度灵活性的语言常常因此而受到影响。人们最终会创建“惯用语法”，帮助他们编写更简洁的代码，但是其他人都会被它们拖慢。

（代码潜规则，不利用新手，新手需要记住很多潜规则。有些潜规则无法避免，

最好写个文档，让每个刚入职的员工先看一遍，熟悉）

一致性

Consistency

With regards to repeated structures names such as “model” and “controller” act as a chunk to remind of how these structures relate to each other.

对于重复结构，诸如“模型”和“控制器”这样的名称作为信息块来提醒这些结构是如何相互关联的。

Frameworks, micro-architectures and game engines all try to create and enforce such relations. This means people have to spend less time figuring out how things communicate and are wired up. Once you grok the structures it becomes easier to jump from one code base to another.

框架、微体系结构和游戏引擎都试图创建和加强这种关系。这意味着人们可以花费更少的时间去弄清楚事物是如何沟通和连接起来的。一旦你通过感觉意会了这个结构，就更容易从一个代码库跳到另一个代码库。

However the main factor with all of this is **consistency**. The more consistent the code base is in naming, formatting, structure, interaction etc. the easier it is to jump into arbitrary code and understand it.

然而，所有这些的主要因素是**一致性**。代码库在命名、格式化、结构、交互等方面越一致。跳入任意代码并理解它就越容易。

（一致性，也就是说要用个通用的规则，比如说变量名都用骆驼峰之类的）

不确定性

Uncertainty

As previously mentioned uncertainty can cause stuttering when reading or writing code.

如前所述，当阅读或编写代码时，不确定性会导致不顺畅的工作。

Let’s take ambiguity as our first example. The simplest example would be [1,2,3].filter(v => v >= 2). The question is, what will this print, is it “2 and 3” or “1”. It’s a simple question, but it can cause a reading/writing stutter when you don’t use it day-in-out.

让我们以模糊度作为我们的第一个例子。最简单的例子是[1, 2, 3].filter(v => v >= 2)。问题是，这个印刷品是“2和3”还是“1”？这是一个简单的问题，但当你不使用它时，它会导致读写工作的不顺畅。

**译者注：**是过滤出大于等于2的元素？还是过滤掉大于等于2的元素？

（到底想要【1】还是【2，3】）

The source of the stutter is **ambiguity**. In the real-world there are two uses for it, one is to keep the part that is getting stuck in the filter and the other that passes through the filter. For example when you have gold in water, then you want to get rid of the water. When you have dirt in the water, you probably want to get rid of the dirt.

工作不顺畅的根源是含糊不清。在现实世界中，它有两种用途，一种是保留被卡在过滤器中的部分，另一种是通过过滤器。例如，当你有金子落入水中，那么你想摆脱水。当你在水中有污垢时，你可能想要清除这些污垢。

Even if we precisely define what filter does, it can still cause stutter because it’s hardwired with two meanings in our brain. The common solution is to use functions such as select, discard, keep.

即时我们精确地定义了filter（过滤器）的作用，它仍然会导致工作不顺畅，因为它在我们的大脑中有两个含义。常见的解决方案是使用诸如select、discard、keep等函数。

We can also attach meaning in different ways, such as types. For example: instead of GetUser(string) you can use type CustomerID string to ensure GetUser(CustomerID) to make clear that the interpretation is “get user using a customer id” instead of other possibilities such as “get user by name”.

我们还可以以不同的方式附加含义，例如类型。例如：你可以使用CustomerID类型的字符串代替GetUser(String)，以确保GetUser(CustomerID)解释为“使用客户ID获取用户”，而不是“按名称获取用户”等其它可能性。

**Similarity** is also easy to conceptually understand. For example having variables such as total1, total2, total3 can lead to situation where you make copy paste mistakes or over a longer piece of code lose track what it meant. For example name such as sum, sum\_of\_squares, total\_error can provide more meaning.

相似性在概念上也很容易理解，例如，拥有诸如total1、total2、total3这样的变量可能会导致复制粘贴错误或在代码较长的时候，无法跟踪它的含义。例如，sum、sum\_of\_squares、total\_error等名称可以提供更多含义。

Having multiple names for the same thing can also be source of confusion when moving between packages. For example in one package you use variable name c, cl and in another client in the third source. It’s interesting to think about special variables such as this and self.

当在包之间移动时，为同一件事情设置多个名称也可能是混淆的根源。例如，在一个包中使用变量名称c、c1，在另一个地方使用变量名client，在第三个地方使用变量名source。想一想特殊的变量，比如this和self，是很有趣的。

Ambiguity and similarity is not a problem just at the source level. Eric Evans noted this in DDD with the [Ubiquitous Language](https://martinfowler.com/bliki/UbiquitousLanguage.html) pattern. The notion is that in different contexts such as billing and shipping, words such as “client” can have widely different usages and meanings, so it’s helpful to keep a vocabulary around to ensure that everyone communicates clearly.

歧义与相似并不仅仅是来源层的问题。Eric Evans用无处不在的语言模式在DDD中注意了这一点。这个概念是，在不同的上下文中，例如账单和发货，诸如“client”这样的词可以有宽泛而不同的用法和含义，所以保持词汇量有助于确保每个人都清楚地沟通。

注释

Comments

We have all seen the “stupid beginner examples” of commenting:

我们都看到了“愚蠢的初学者的例子”的注释：

// makes variable i go from 0 to 99

for(var i = 0; i < 100; i++) {

// sets value 4 to variable a

var a = 4;

（愚蠢指的是每行加注释吧）

While it may look stupid, it might have some purpose. Think about learning a second or third language. You usually learn the new language by understanding the translation in your primary language. These are the “chunks” written out explicitly.

虽然它看起来很愚蠢，但可能有它的目的。考虑学习第二或第三语言。你通常通过理解你的主要语言的翻译来学习新的语言。这些是明确写出的“信息块”。

Once you have learned “chunk” the comments become noise, because you already know that information by looking at the second line.

一旦你学会了“信息块”，这些注释就会变成噪音，因为你已经通过看第二行就知道了这些信息。

As programmers get better, the intent of comments becomes to condense information and to provide a context for understanding code. Why was a particular approach taken when doing X or what needs to be considered when modifying the code.

当程序员变得更好时，注释的目的就变成了压缩信息和提供理解代码的上下文。为什么在执行X时采用了特定的方法，或者在修改代码时需要考虑什么。

Effectively, it’s for setting up the right mental model for reading the code.

实际上，这是为了建立正确的阅读代码的心理模型。

（也就是说，只在关键或难以理解或在潜规则代码处加上注释）

上下文

Contexts

Working memory limitation leads us to decompose and partition our code into different interacting pieces. We must be mindful in how we relate different pieces and how they interact.

工作记忆限制导致我们分解和划分我们的代码到不同的交互部件。我们必须注意我们如何将不同的部分联系起来，以及它们是如何相互作用的。

For example when we have a very deep inheritance chain and we use things from all different inheritance levels, the class might be too complicated, even if each class has maybe two methods and each method is five lines of code. The class and all the parents form a single “whole”. Illustratively you can count each “inheritance step” as a “single idea” that you need to remember when you use that particular class.

例如，当我们有一个非常深的继承链，并且使用来自所有不同继承级别的东西时，类可能太复杂了，即时每个类可能有两种方法，而且每个方法都是五行代码。所有类和父类组成一个单一的“整体”。举例说明，你可以将每个“继承步骤”计算为使用该特定类时需要记住的“单个想法”。

The other side of contexts is moving between function calls. Each call is a “context in our mental model”, so we need to remember where we came from and how it relates to the current situation. The deeper the call stack, the more stuff we have to keep in mind.

上下文的另一面是在函数调用之间移动。每一个调用都是一个“心理模型中的上下文”，所以我们需要记住我们来自何处以及它是如何与当前的情况相关联的。调用堆栈越深，我们需要记住的东西就越多。

One way to reduce the depth of our mental model contexts is to clearly separate them. One of such examples is early return:

减少我们的心理模型上下文的深度的另一种方法是清楚地将它们分开。其中一个例子就是提前返回：

public void SomeFunction(int age)

{

if (age >= 0) {

// Do Something

} else {

System.out.println("invalid age");

}

}

public void SomeFunction(int age)

{

if (age < 0){

System.out.println("invalid age");

return;

}

// Do Something

}

In the first version when we read the “Do Something” part we understand it only happens when the age is positive. However, when we reach the “else” part we have forgotten what the condition was, because at that point the distance from the condition can be quite far away.

在第一个版本中，我们读到“Do Something”的部分时，我们知道只有当年龄是非负数的时候才会发生。然而，当我们到达else部分时，我们已经忘记了条件是什么，因为在这一点上，与条件的距离可能很远。

The second version is somewhat nicer. We have lost the necessity to keep multiple “contexts” in our head, but can focus instead of a single context that is setup and verified by multiple checks in the beginning.

第二个版本则要好一些。我们已经失去了在头脑中保留多个“上下文”的必要性，我们可以集中注意力，而不是在开始时通过多次检查来设置和验证单个上下文。

经验法则

Rules of thumb

One of the usual recommendations is “don’t have global variables”. But, when a variable is set during startup and never changed again, is that a problem? The problem isn’t in the “variableness” or “globalness” of something, but rather in how it affects our capability to understand code. When something is modified at a distance then we cannot build a contained model of it. The “globalness” of course clutters the namespace (depending on the language) and means there are more places it can be accessed from. Of course there are many other things that have same properties, such as “Singleton”. So, why is it considered better than a global variable?

通常的建议之一是“没有全局变量”。但是，当一个变量在启动时被设置，并且再也不会改变，这是一个问题吗？问题不在于事物的“多样性”或“全局性”，而在于它如何影响我们理解代码的能力。当某物在一定时间间隔内被修改时，我们就无法建立包含它在内的模型。当然，“全局性”把命名空间（取决于语言）弄得很乱，这意味着可以从更多的地方访问它。当然，还有许多其他的东西具有相同的属性，比如单例。那么，为什么人们认为它比全局变量更好呢？

Single responsibility principle (SRP) is easy to understand with these concepts. It tries to ensure that we have proper chunks for a thing. This constraint often makes chunks smaller. Having a single responsibility also means that we end up with things that have working memory need. However, we need to consider that when we separate a class or function into multiple pieces we introduce many new artifacts. When these artifacts are deeply bound together we may not even gain the benefits of SRP.

单一责任原则（SRP）很容易理解这个概念。它试图确保我们有适合某件事的信息块。这个约束通常使信息块变小。拥有单一的责任也意味着我们最终会有工作记忆所需的东西。但是，我们需要考虑的是，当我们将类或函数分离为多个部分时，我们引入了许多新的构件。当这些构件深深地结合在一起时，我们甚至可能得不到SRP的好处。

[Carmack’s comments on inlined functions](http://number-none.com/blow/john_carmack_on_inlined_code.html) is a good example of this. The three examples he gave were these:

Carmack对内联函数的评价就是一个很好的例子。他列举了三个例子如下：

**// A**

void MinorFunction1( void ) {

}

void MinorFunction2( void ) {

}

void MinorFunction3( void ) {

}

void MajorFunction( void ) {

MinorFunction1();

MinorFunction2();

MinorFunction3();

}

**// B**

void MajorFunction( void ) {

MinorFunction1();

MinorFunction2();

MinorFunction3();

}

void MinorFunction1( void ) {

}

void MinorFunction2( void ) {

}

void MinorFunction3( void ) {

}

**// C.**

void MajorFunction( void ) {

{ // MinorFunction1

}

{ // MinorFunction2

}

{ // MinorFunction3

}

}

By making pieces smaller we made the chunks smaller, however understanding the system became harder. We cannot read our code from top-to-bottom and understand what it does, but instead we have to jump around in the code base to read it. Version C preserves the linear ordering while still maintaining the conceptual chunks.

我们通过使部件更小，从而使信息块更小，但是理解系统变得更加困难。我们不能自上而下的阅读我们的代码，也不能理解它的作用，相反，我们必须在代码库中跳来跳去去阅读它。在保持概念快的同时，**C**版保留了现行排序。

概要

Summary

Overall we can summarize the code readability as trying to balance different aspects:

总之，我们可以将代码可读性概括为试图平衡不同方面：

1.Names help us retrieve the right **chunks** from memory and help us figure out their meaning. Too long a name can end up being noisy in our code. Too short a name may not help us figure out its true meaning. Bad names are misleading and confusing.

1.名字帮助我们从记忆中检索出正确的信息块，并帮助我们理解它们的意义。在我们的代码中，太长的名字可能会引起噪音。太短的名字可能无法帮助我们找出它真正的含义。不好的名字是误导和令人困惑的。

2.To minimize the cost of **shifting attention**, we try to write all related code close together. To minimize the burden to our **working memory**, we try to split the code into smaller and more fathomable units.

为了将注意力转移的成本降到最低，我们尝试将所有相关代码紧密地写在一起。为了将工作记忆的负担降到最低，我们尝试将代码分割成更小、更可以理解的单元。

3.Using **common vocabulary** allows the author as well as the team to rely on previous code-reading experience. That means reading, understanding and contributing to code is easier. Using unique solutions in place where a common one would do, can slow down new readers of that code.

使用通用词汇可以让作者和团队依赖以前的代码阅读经验。这意味着阅读、理解和贡献代码更容易。在一个普通的解决方案可以解决的问题中使用独特的解决方案，可以让代码的新读者阅读变得迟钝。

In practice there is no “perfect” way of organizing code, but there are many trade-offs. While I focused on readability, it is never the end goal, there are many other things to consider like reliability, maintainability, performance, speed of prototyping.

在实践中，没有“完美”的组织代码方式，但有许多权衡。虽然我关注的是可读性，但它永远不是最终的目标，还有许多其他的事情需要考虑，比如可靠性、可维护性、性能、原型的速度。