Google 技术写作

原文：Technical Writing
翻译：陈皓
修订：伍俊滨

第一部分

语法

词性	定义	例
Noun 名词	人，地方，概念或事物	**Sam** runs **races**. 山姆赛跑。
Pronoun 代词	替代另一个名词的名词	Sam runs races. **He** likes to compete. 山姆赛跑。他喜欢竞争。
Adjective 形容词	修饰名词的单词或短语	Sam wears **blue** shoes 山姆穿蓝色的鞋子。
Verb 动词	一个动作词或短语	Sam **runs** races. 山姆跑比赛。
Adverb 副词	修饰动词，形容词或其他副词的单词或短语	Sam runs **slowly**. 山姆跑得慢。
Preposition 介词	指定两个名词的位置关系的单词或短语	Sam's sneakers are seldom **on** his shelf. 山姆的运动鞋很少在他的架子上。
Conjunction 连词	连接两个名词或短语的单词	Sam's trophies **and** ribbons live only in his imagination. 山姆的奖杯和缎带只存在于他的想象中。
Transition 过渡	连接两个句子的单词或短语	Sam runs races weekly. **However**, he finishes races weakly. 山姆每周参加比赛。但是，他无力完成比赛。

名词

名词代表人，地方或事物。**朱迪（Judy）**，**南极洲（Antarctica**）和 **锤子（Hammer**）都是名词，无形的概念（例如**健壮性 robustness** 和**完美性 perfection**）也是如此。例如，我们在下面的示例中加粗了名词：

In the **framework**, an **object** must copy any underlying **values** that the **object** wants to change. The **protos** in the **codebase** are huge, so copying the **protos** is unacceptably expensive.

代词

代词是一个间接层，它指向或替代了其他名词或句子。例如：Janet writes great code. **She** is a senior staff engineer. 示例中，第一句话将Janet建立为名词。第二句用代词“ She”代替名词“Janet”。

在以下示例中，代词**This**代替了它前面的整个句子：

Most applications aren't sufficiently tested. **This** is poor engineering.

动词

动词是一个动作词或短语。当你想要表示两个名词（一个行为者和一个目标）之间的关系时，该动词就起作用了。动词标识行为者对目标的作用。每个句子必须至少包含一个动词。例如，以下每个句子包含一个动词：

- Sakai **prefers** pasta. 酒井法子喜欢面食。
- Rick **likes** the ocean. 瑞克喜欢大海。
- Smurfs **are** blue. 蓝精灵是蓝色的。
- Jess **suffers** from allergies. 杰西有过敏症。

有些句子会包含多个动词，如：

- Nala **suffers** from allergies and **sneezes** constantly.
  娜娜过敏，经常打喷嚏。
- Chung **likes** snacks **to eat** while **riding** the train.
  Chung喜欢在火车上吃零食。

根据时态和词缀变化，一个动词可以包含一个单词或多个单词。例如：

- Tina **was eating** breakfast a few hours ago.
  蒂娜几小时前正在吃早餐。

- Tina **is eating** lunch right now.
  蒂娜现在在吃午餐。
- Tina **will eat** dinner tonight at 7:00.
  蒂娜将在7点吃晚餐。

形容词和副词

形容词修饰名词。例如，在下面的句子中，注意形容词如何修饰后面的名词：

Tom likes **red** balloons. He prepares **delicious** food. He fixed **eight** bugs at work.

大多数副词修饰动词。例如，注意下面句子中的副词是如何(有效地)修饰动词的：

Jane **efficiently** fixes bugs.

副词不一定紧挨着动词。例如，在下面的句子中，副词(effective)与动词(fixes)相距两个单词

Jane fixes bugs **efficiently**.
副词也可以修饰形容词或其他副词。

连词和过渡

连词连接句子中的短语或名词；过渡连接句子本身。最重要的连词如下：

- and
- but
- or

例如，在下面的句子中，and连接了“code”和“documentation”，而but连接了句子的前半部分和后半部分。

Natasha writes great internal code **and** documentation **but** seldom works on open-source projects.

Natasha 编写了大量的内部代码和文档，但是很少在开源项目上工作。

技术写作中最重要的过度词如下：

- however
- therefore
- for example

在下面的段落中，请注意过渡如何连接句子并使其上下文相关：

> Juan is a wonderful coder. **However**, he rarely writes sufficient tests. **For example**, Juan coded a 5,000 line FFT package that contained only a single 10-line unit test.
>
> Juan 是一个出色的程序员。然而，他很少编写足够的测试。例如，Juan编码了一个5000行的FFT包，却只包含一个10行的单元测试。

单词

定义新术语或不熟悉的术语

在写作或编辑时，识别那些目标受众可能不熟悉的术语。当你发现此类术语时，请采取以下两种策略之一：

- 如果该术语已经存在，请链接到现有的具体解释。（不要重新发明轮子）
- 如果你的文档中引入了该术语，请定义该术语。如果你的文档引入了许多术语，请将定义收集到词汇表中。

始终使用术语

如果在方法中途更改变量的名称，则代码将无法编译。同样，如果你在文档中间重命名术语，则你的想法将无法编译（在用户头脑中）。

修养：在整个文档中始终使用相同的明确词或术语。一旦你将某个组件命名为**thingy**之后，不要将其重命名为 **thingamabob**。例如，以下段落错误地将 **Protocol Buffers** 重命名为 **protobufs**：

> **Protocol Buffer** 提供了自己的定义语言。……。这就是 **protobufs** 赢得如此众多县博览会的原因。

是的，技术写作是残酷和充满限制的，但是至少技术写作提供了一个很好的解决方法。即，当引入冗长的概念名称或产品名称时，你也可以指定该名称的缩写形式。然后，你可以在整个文档中使用该简称。例如，以下段落很好：

> **Protocol Buffer**（或简称 **protobuf**）提供了自己的定义语言。…… 这就是 **protobuf** 赢得如此众多县博览会的原因。

正确使用首字母缩写词

在文档或章节中首次使用不熟悉的首字母缩写词时，请拼写完整的术语，然后将首字母缩写词放在括号中。拼写版本和首字母缩写用黑体字标出。例如：

本文档适用于**远程触觉网络 Telekinetic Tactile Network（TTN）**的新手或需要了解如何通过手指运动订购TTN替换零件的工程师 。

然后可以使用首字母缩略词，如以下示例所示：

如果不存在缓存条目，则混合器将调用 **OttoGroup Server**（**OGS**）来为请求获取Ottos。OGS是一个存放所有可使用的Otto的存储库。OGS以逻辑树结构组织，具有一个根节点和两个级别的叶节点。OGS根将请求转发到叶子并收集响应。
另外，不要在同一文档中的首字母缩写词和扩展版本之间来回切换。

使用首字母缩写词还是完整术语？

当然，你可以正确地引入和使用首字母缩写词，但是你真的要使用首字母缩写词吗？好吧，首字母缩略词确实减少了句子的大小。例如，*TTN* 比*Telekinetic Tactile Network* 短很多。但是，首字母缩略词实际上只是抽象层。读者必须在头脑中将最近学到的首字母缩略词扩展到整个术语。例如，读者在*脑海*中将 *TTN* 转换为 *Telekinetic Tactile Network*，因此“较短”的首字母缩略词实际上要比整个术语花费更长的时间。

大量使用的首字母缩写词基本上会变成另外一个新词。在出现许多情况后，读者通常停止将首字母缩略词展开成具体的单词。例如，许多Web开发人员已经忘记了*HTML* 这个术语展开后是什么。

这是首字母缩写词的准则：

- 不要定义只会使用几次的首字母缩写词。
- 请定义同时满足以下两个条件的首字母缩写词：
    - 该首字母缩写词明显短于整个术语。
    - 该首字母缩写词在文档中很多次出现。

消除代词歧义

许多代词指向先前引入的名词。这种代词类似于编程中的指针。像编程中的指针一样，代词往往会引入错误。代词使用不当会就像程序中的 `nullptr` 空指针错误一样在

读者的脑海中造成错误的认知 。在许多情况下，你应该简单地避免代词，而就直接重复使用该名词。但是，代词的效用有时会非常有用。

请考虑以下代词准则：

- 引入名词*后才*使用代词；在介绍名词之前，切勿使用代词。
- 代词应尽可能靠近指称名词。根据经验，如果将名词与代词分隔开的单词超过五个，请考虑重复使用名词，而不要使用代词。
- 如果在名词和代词之间引入第二个名词，请重复使用名词，而不要使用代词。

it 和 they

以下代词在技术文档中引起最大的混乱：

- it
- they，them 和 their

例如，在下面的句子中，**它**是指Python还是C ++？

Python是解释型语言，而C ++是编译型语言。**它**具有几乎类似邪教的追随者。

再举一个例子，**它们** 在接下来的句子中指的是什么？

将 Frambus 或 Carambola 与 HoobyScooby 或 BoiseFram 一起使用时要小心，因为**它们的**核心可能会导致意外的大量脱机。

this 和 that

考虑另外两个问题代词：

- this
- that

例如，在下面有歧义的句子中，“**这**”可能是指Frambus，Foo或两者：

你可以使用 Frambus 或 Foo 来计算导数。**这** 不是最佳的。

使用以下的战术来消除歧义**这个**和**那个**：

- 将 **this** 或 **that** 替换为相关的名词。
- 在 **this** 或 **that** 后马上使用那个名词。

例如，以下两个句子中的任何一个都消除了前面的示例的歧义：

Overlapping functionality is not optimal.
This overlapping functionality is not optimal.

主动语态与被动语态

技术写作中的绝大多数句子都应该是主动语态。本单元教你如何执行以下操作：

- 区分被动语态和主动语态。
- 将被动语态转换为主动语态，因为主动语态通常更清晰。

用简单的句子区分主动语态和被动语态

在主动的语态句子中，主语作用于目标。也就是说，主动语态句子遵循以下公式：

主动语态句=主语+动词+目标

被动的语态句子则反过来。即，被动语态语句通常遵循以下公式：

被动语态句=目标+动词+主语

主动语态示例

例如，这是一个简短而主动语态句子：

The cat sat on the mat. 猫坐在垫子上。

- 主语：The cat
- 动词：sat
- 目标：the mat

被动语态示例

相比之下，这是被动语态中的同一句话：

The mat was sat on by the cat. 垫子被猫坐着。

- 目标：The mat
- 被动动词：was sat

- 主语：the cat

一些被动的语态句子省略了主语。例如：

```
The mat was sat on。
```

- 主语：*不明*
- 被动动词：was sat
- 目标：the mat

谁或什么坐在垫子上？一只猫？一只狗？霸王龙？读者只能猜测。技术文档中的好句子可以确定谁对谁做事。

识别被动动词

被动动词通常具有以下公式：

```
passive verb = be 的形式 + 动词过去分词
```

尽管上述公式令人生畏，但实际上非常简单：

- **be** 在一个被动动词中通常是下列词语之一：
  - is / are
  - was / were
- **past participle verb 过去分词动词** 通常是一个普通的动词加上过去式的后缀 ed。例如，以下是过去分词动词：
  - interpreted
  - generated
  - formed

不幸的是，某些过去分词动词是不规则的；也就是说，过去分词形式不以后缀*ed*结尾。例如：

- sat
- known
- frozen

将*be*和过去分词的形式放在一起会产生被动动词，例如：

- was interpreted
- is generated
- was formed
- is frozen

如果短语中包含一个主语，介词通常会跟在被动动词之后。(这个介词通常是帮助你辨别被动语态的关键线索)下面的例子结合了被动动词和介词：

- was interpreted as 被解释为
- is generated by 由……生成
- was formed by 由……形成
- is frozen by 被……冻结

祈使动词通常是主动的

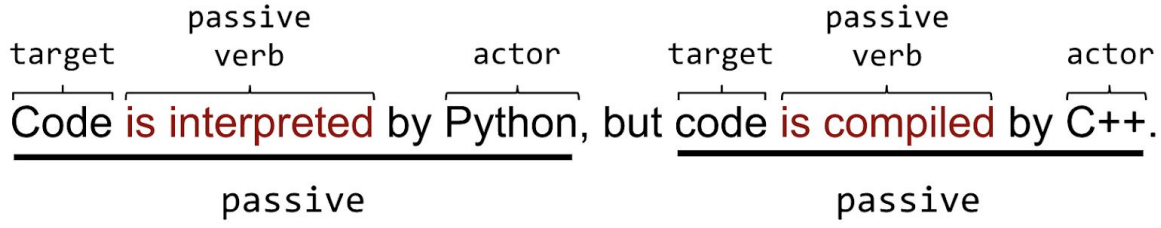
将祈使动词开头的句子很容易错误地归为被动。一个**祈使动词**是一个命令。编号列表中的许多项目都以祈使动词开头。例如，以下列表中的“ *Open*”和“ *Set* ”都是祈使动词：

1. Open the configuration file.
2. Set the `Frombus` variable to `False`.

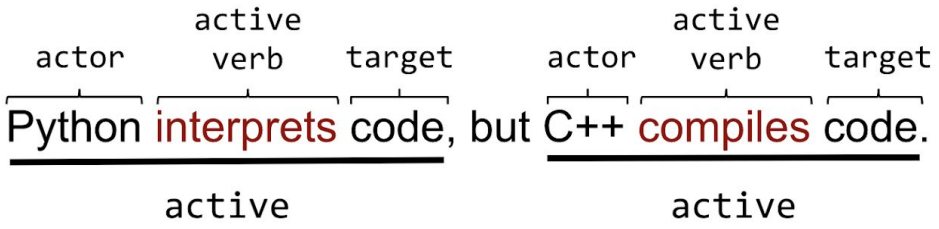
以祈使动词开头的句子通常采用主动语态，即使它们没有明确提及主语。相反，以命令式动词开头的句子*暗示*一个主语。这个隐含的主语就是“**你**”。

用更复杂的句子区分主动语态和被动语态

许多句子包含多个动词，其中有些是主动的，有些是被动的。例如，以下句子包含两个动词，两个动词均为被动语态：



完全转换为主动语态：



首选主动语态而不是被动语态

大部分时间使用主动态。谨慎使用被动语态。主动语态具有以下优点：

- 大多数读者会在心理上将被动语态转换为主动语态。为什么要使读者的处理时间更长？通过坚持主动语态，读者可以跳过预处理阶段，直接进入编译阶段。
- 被动语态会使你的想法模糊不清，使他们的句子变得无聊。被动语态间接报告操作。
- 一些被动语态的句子完全忽略了主语，这迫使读者猜测主语是谁。
- 主动语态通常比被动语态更短。

Be bold—be active.

清晰的句子

喜剧作家寻求最有趣的结果，恐怖作家寻求最恐怖的结果，技术作家寻求最清晰的结果。在技术写作中，清晰度优先于所有其他规则。本单元提供了几种使句子清晰的方法。

选择强动词

许多技术写作者认为，动词是句子中最重要的部分。选择正确的动词，句子的其余部分都会顺理成章。不幸的是，有些写作者只重复使用了少量温和的动词，就像每天为客人提供千篇一律的饼干和生菜一样。选择正确的动词需要花费更多时间，但会产生更令人满意的结果。

为了吸引和教育读者，请选择精确，有力的特定动词。减少不精确，虚弱或通用的动词，例如：

- *be的*形式：is，are，am，was，are等。
- occur
- happen

例如，考虑以下句子中的弱动词如何增强有代入感的句子

弱动词	强动词
The error **occurs** when clicking the Submit button.	Clicking the Submit button **triggers** the error.
This error message **happens** when...	The system **generates** this error message when...
We **are** very careful to ensure...	We carefully **ensure**...

许多写作者都依赖于*be的*形式，好像它们是货架上唯一的香料。撒上不同的动词，能让自己的散文变得更开胃。也就是说，一种形式的*be*有时是动词的最佳选择，因此不必觉得你必须从写作中消除每种形式的*be*。

请注意，一般动词通常有病变的信号，例如：

- 句子中不精确的主语或没有主语
- 被动语态句子

示例一

```
When a variable declaration doesn't have a data type, a compiler error happens.
```

被改写成下面的形式会更好：

- When a variable declaration doesn't **specify** a data type, the compiler **generates** an error message.
- If you **declare** a variable but don't **specify** a data type, the compiler **generates** an error message.

示例二

```
Compiler errors occur when you leave off a semicolon at the end of a statement.
```

被改成下面的形式会更好：

- Compilers **issue** errors when you **omit** a semicolon at the end of a statement.
- A missing semicolon at the end of a statement **triggers** compiler errors.

减少使用 there is/there are

以**There is** 或 **There is** 开头的句子将普通名词嫁接到普通动词上。这种乱点鸳鸯谱的方式会使读者感到厌烦。通过提供真实的主语和真实的动词来表达才是对读者的真爱。

在最佳情况下，你可以简单地删除“**There is**”或“**There are**”（以及句子后面的另一个单词或两个单词）。例如，考虑以下句子：

There is a variable called `met_trick` that stores the current accuracy.

删除 **There is** 用更好的主语替换通用主题。例如，以下任一句子比原始句子更清晰：

A variable named `met_trick` stores the current accuracy.
The `met_trick` variable stores the current accuracy.

你有时可以通过将真实的主语和真实的动词从句子的末尾移到开头来修复 **There is**或 **There are** 句子。例如，请注意，**You** 的代词出现在以下句子的结尾：

There are two disturbing facts about Perl you should know.

用 You 替换 there is ：

You should know two disturbing facts about Perl.

在其他情况下，写作者以 **There is** 或 **There are** 开始句子，以避免创建真实的主语或动词的麻烦。如果不存在任何主题，请考虑创建一个。例如，以下 **There is** 句子不能识别接收实体：

There is no guarantee that the updates will be received in sequential order.

用有意义的主语（例如:**client**）代替“ There is”可以为读者带来更清晰的体验：

Clients might not receive the updates in sequential order.

练习

1. There is a lot of overlap between X and Y.
2. There is no creator stack for the main thread.
3. There is a low-level, TensorFlow, Python interface to load a saved model.
4. There is a sharding function named `distribute` that assigns keys.

可以改写为：

1. X and Y overlap a lot.

2. The main thread does not provide a creator stack.
3. TensorFlow provides a low-level Python interface to load a saved model.
4. The `distribute` sharding function assigns keys.

最小化特定的形容词和副词（可选）

形容词和副词在小说和诗歌中表现出色。由于形容词，普通的草能变成**杂草 prodigal** 和**葱绿 verdant**，而毫无生气的头发变换到的东西**柔滑 silky** 和**流动 flowing**。副词能让马跑地**疯狂 madly**和**自由 freely**，让狗叫得**大声 loudly** 和**凶猛 ferociously**。不幸的是，形容词和副词有时会加入很多噪音。那是因为形容词和副词的定义过于松散，对技术读者而言主观。更糟糕的是，形容词和副词会使技术文档听起来像营销材料一样危险。例如，请考虑以下技术文档中的内容：

Setting this flag makes the application run screamingly fast.

诚然，**screamingly fast 令人尖叫的速度** 能引起了读者的注意，但不一定是一种很好的方式。向你的读者提供事实数据，而不是像市场营销人员讲话。将无定形副词和形容词重构为客观的数字信息。例如：

Setting this flag makes the application run 225-250% faster.

前面的更改是否会剥夺其某些魅力的句子？是的，有一点，但是修改后的句子获得了准确性和可信度。

简短的句子

软件工程师通常愿意精简代码，主要出于以下原因：

- 简短的代码更易于他人阅读。
- 简短的代码更易于维护。
- 多余的代码可能会引入潜在的故障。

以上规则，同样适用于技术写作：

- 简短的文档可读性更好。
- 简短的文档更易于维护。
- 多余的文档行会引入额外的问题。

寻找最短的文档实现需要时间，但最终还是值得的。短句子比长句子更有效地进行交流，并且短句子通常比长句子更容易理解。

一个句子只聚焦在一个想法

将每句话聚焦在一个想法上思想或概念上。就像程序中的语句执行单个任务一样，句子也应该执行单个想法。例如，以下很长的句子包含多种想法：

The late 1950s was a key era for programming languages because IBM introduced FORTRAN in 1957 and John McCarthy introduced Lisp the following year, which gave programmers both an iterative way of solving problems and a recursive way.

1950年代后期是编程语言的关键时代，因为IBM于1957年推出了FORTRAN，而John McCarthy于次年推出了Lisp，这为程序员提供了解决问题的迭代方法和递归的方法。

将长句子分解为一连串的单意识句子会产生以下结果：

The late 1950s was a key era for programming languages. IBM introduced FORTRAN in 1957. John McCarthy invented Lisp the following year. Consequently, by the late 1950s, programmers could solve problems iteratively or recursively.

1950年代后期是编程语言的关键时代。IBM在1957年推出了FORTRAN。第二年，John McCarthy发明了Lisp。因此，到1950年代后期，程序员可以使用迭代或递归的方法来解决问题。

练习

将以下长句子转换为一系列短句子。不要修改太多；最后只剩下几句话而不是只有一句话。

In bash, use the if, then, and fi statements to implement a simple conditional branching block in which the if statement evaluates an expression, the then statement introduces a block of statements to run when the if an expression is true, and the fi statement marks the end of the conditional branching block.

可以改写成

In bash, use an if, then, and fi statement to implement a simple conditional branching block. The if statement evaluates an expression. The then statement introduces a

block of statements to run when the if an expression is true. The fi statement marks the end of the conditional branching block.

将长句子转换为列表

许多冗长的技术语句中，都有一个渴望摆脱困境的清单。例如，考虑以下句子：

To alter the usual flow of a loop, you may use either a **break** statement (which hops you out of the current loop) or a **continue** statement (which skips past the remainder of the current iteration of the current loop).

要更改循环的通常流程，可以使用**break**语句（使你跳出当前循环）或**continue**语句（跳过当前循环的当前迭代的其余部分）。

当你看到连词**or**长句子时，请考虑将该句子重构为项目符号列表。当你看到长句子中嵌入的项目或任务列表时，请考虑将该句子重构为项目符号或编号列表。例如，前面的示例包含连词**或or**，因此让我们将长句子转换为以下项目符号列表：

要更改循环的通常流程，请调用以下语句之一：

- `break`，使你跳出当前循环。
- `continue`，跳过当前循环的当前迭代的其余部分。

练习

将以下句子重构为更简短的内容。确保你的答案包含一个列表：

1. To get started with the Frambus app, you must first find the app at a suitable store, pay for it using a valid credit or debit card, download it, configure it by assigning a value for the `Foo` variable in the `/etc/Frambus` file, and then run it by saying the magic word twice.
2. KornShell was invented by David Korn in 1983, then a computer scientist at Bell Labs, as a superset of features, enhancements, and improvements over the Bourne Shell (which it was backwards compatible with), which was invented by Stephen Bourne in 1977 who was also a computer scientist at Bell Labs.

可以改写为

Take the following steps to get started with the Frambus app:
请按照以下步骤使用 Frambus 应用：

1. Find the app at a suitable store.

   在应用商店搜索该应用。

2. Pay for the app using a valid credit or debit card.

   使用有效的信用卡或借记卡购买该应用。

3. Download the app.

   下载安装应用。

4. Configure the app by assigning a value for the Foo variable in the /etc/Frambus file.

   在 /etc/Frambus 文件中，为 Foo 变量分配一个值来配置应用。

5. Run the app by saying the magic word twice.

The following two Bell Labs computer scientists invented popular shells:

- Stephen Bourne invented the Bourne Shell in 1977.
- David Korn invented the KornShell in 1983.

The KornShell's features are a backwards-compatible superset of the Bourne Shell's.

消除或减少多余的单词

许多句子都包含填充词，即文本的垃圾食品，它占用空间而不滋养读者。例如，看看是否可以在以下句子中找到不必要的单词：

```
An input value greater than 100 causes the triggering of logging.
```

替换**causes the triggering of 为**动词 **trigger** 可以产生较短的句子：

```
An input value greater than 100 triggers logging.
```

通过练习，你会发现多余的单词，并享受删除他们的快乐。例如，考虑以下句子：

```
This design document provides a detailed description of Project Frambus.
```

句子 **provides a detailed description of** 可以缩减为动词 **details**，句子变为：

```
This design document details Project Frambus.
```

下表建议了一些常见的啰嗦的句子的替换：

啰嗦	简洁
at this point in time	now
determine the location of	find
is able to	can

减少从句（可选）

一个**从句**是一个句子中的独立逻辑片断，其中包含一个主语和动作。每个句子包含以下内容：

- 一个主句
- 零个或多个从句

从句在主从句中会修改了主要的意思。还会暗示着，从句比主句重要。例如，考虑以下句子：

```
Python是一种解释型编程语言，于1991年发明。
```

- 主句：`Python是一种解释型编程语言`
- 从句：`于1991年发明`

通常，你可以通过引入从句的词来识别它们。以下列表（绝不完整）显示了引入从句的常用词：

- which
- that
- because
- whose
- until
- unless
- since

有些从句以逗号开头，有些则没有。例如，以下句子中突出显示的从属子句以单词“ **因为because”**开头，并且不包含逗号：

I prefer to code in C++ because I like strong data typing.

我更喜欢用C ++编写代码，因为我喜欢强大的数据类型。

编辑时，请仔细检查从属子句。记住`one sentence = one idea`公式。句子中的从句是*扩展*单个概念还是将其分支成一个单独的概念？如果是后者，请考虑将有问题的从属子句分成单独的句子。

练习

判断哪些句子包含从句，这些从句应该被分支成独立的句子。(不要重写句子，只要找出需要重写的句子就可以了)

Python is an interpreted language, which means that the language can execute source code directly.

这个句子里的从句扩展了主句的意思，所以这个句子还可以。

Bash is a modern shell scripting language that takes many of its features from KornShell 88, which was developed at Bell Labs.

第一个从句扩展了主句的意思，而第二个从句则向另一个方向发展。把这个句子分成两部分。

Lisp is a programming language that relies on Polish prefix notation, which is one of the systems invented by the Polish logician Jan Łukasiewicz.

第一个从句扩展了主句的意思，而第二个从句则向另一个方向发展。把这个句子分成两部分。

I don't want to say that Fortran is old, but only radiocarbon dating can determine its true age.

这个句子里的从句扩展了主句的意思，所以这个句子还可以。

区分 that 和 which

that 和 **which** 都是用来引入了从句的。它们之间有什么区别？好吧，在某些国家，这两个词几乎可以互换。但是，不可避免的是，机智的美国读者会愤怒地宣布你再次混淆了这两个词。

在美国，使用 **which** 从句意味着从句是不必要的部分，而使用 **that** 则表明是一个重要的短语，句子不能没有。例如：

> Python is an interpreted language, **which** means the processor runs the program directly.
>
> Python是一种解释性语言，这意味着处理器可以直接运行程序。

> FORTRAN is perfect for mathematical calculations **that** don't involve linear algebra.
> FORTRAN是完美的数学计算是不涉及线性代数。

这样的解释有用吗？可能不会。相反，请尝试以下操作：如果你大声朗读句子并听到从属子句之前的停顿，则使用**which**。如果你没有听到暂停，使用 **that**。返回并阅读两个例句。你听到第一句话中的停顿了吗？

在**which**前面放置逗号；在 **that** 之前不要使用逗号。

列表和表格

读者通常喜欢列表，好的列表可以将混乱转变为有序。因此，在写作时，尽可能将散文转换为列表。

选择正确的列表类型

以下列表在技术写作中占主导地位：

- 无序列表 bullets lists
- 有序列表 numbered lists
- 嵌入式列表 embedded lists

未排序的项目使用**无序列表**；排序的项目使用 **有序列表**。换一种说法：

- 如果修改列表的顺序不会改变含义，则使用**无序列表**。
- 如果修改列表的顺序会改变含义，则使用**有序列表**。

下面例子我们使用了**无序列表**，因为重新排列项目不会改变含义：

Bash提供以下字符串操作机制：

- 子字符串会从字符串开头删除
- 将整个文件读入到一个字符串变量

相比之下，下面例子使用有序列表，因为重新排列其项目会改变列表的含义：

请执行以下步骤来重新配置服务器：

1. 停止服务器。
2. 编辑配置文件。
3. 重新启动服务器。

嵌入式列表（有时称为 **run-in** 列表）包含在一个句子。例如，下面句子包含了四个项目的嵌入式列表。

llamacatcher API使调用者可以创建和查询美洲驼、分析羊驼、删除骆驼和跟踪单峰骆驼。

一般而言，嵌入式列表是展示信息较差的方法。尽量将**嵌入式列表**转换为无序列表或有序列表。

例如，上面的例子可以转化为无序列表：

llamacatcher API使调用者可以执行以下操作：

- 创建和查询美洲驼。
- 分析羊驼。
- 删除骆驼。
- 跟踪单峰骆驼。

将以下段落转换为列表形式：

今天，在工作中，我必须编写三个单元测试的代码，编写设计文档，并查看Janet的最新文档。下班后，我必须不用水洗车，然后不用毛巾擦干。

以下是列表的形式。

我今天工作必须执行以下操作：

- 编码三个单元测试。
- 编写设计文档。
- 查看Janet的最新文档。

下班后，我必须执行以下操作：

1. 不用水洗我的车。
2. 不用毛巾擦干我的车。

保持清单项目平行

如何区分好的列表和坏的列表？好的列表是平行的，坏的列表则相反。**平行**列表中的项目看起来属于一类。换句话说，平行列表中的所有项目都符合下面的参数：

- 语法
- 逻辑类别
- 大小写
- 标点

相反，**非平行**列表中的至少一项不符合上面参数。

例如，以下列表是平行的，因为所有项都是复数名词（语法），可食用（逻辑类别），小写（大小写），并且没有句点或逗号（标点符号）。

- carrots
- potatoes
- cabbages

相比之下，以下列表在所有四个参数上都不是平行的：

- carrots
- potatoes
- The summer light obscures all memories of winter.

以下列表是并行的，因为所有项目都是完整的句子，并带有完整的句子大写和标点符号：

- Carrots contain lots of Vitamin A.（胡萝卜含有大量的维生素A）
- Potatoes taste delicious.（土豆味道鲜美）

- Cabbages provide oodles of Vitamin K. (卷心菜提供大量的维生素K）

列表中的第一项建立了读者的预期，后面项目反复强化这种预期。。

列表项使用动词开头

推荐在有序列表中，项目使用动词开头，例如 **打开open** 或 **启动 start**。下面是一些示例，请注意以下列表中所有的项目是如何以动词开头的：

1. Download the Frambus app from Google Play or iTunes.

   下载 Frambus 应用（Google Play 或 iTunes）。
2. Configure the Frambus app's settings.

   配置 Frambus 应用的设置。
3. Start the Frambus app.

   启动 Frambus 应用。

以下有序列表是不平行的，因为前两项以动词开头，而第三项则不是：

1. Instantiate the Froobus class.

   实例化 Froobus 类。
2. Invoke the Froobus.Salmonella() method.

   调用 Froobus.Salmonella() 方法。
3. The process stalls.

   这个过程停滞不前。

正确使用标点符号

如果列表项是句子，请使用首字母大写并使用标点符号。否则，请勿使用句子大写和标点符号。例如，以下列表项是一个句子，因此使“ **Most”**中**M**大写， 并在句末加句号。

- Most carambolas have five ridges.

但是，下面的列表项不是句子，所以使 **the** 中的 **t** 小写并不加标点。

- the color of lemons

创建有用的表格

分析型的头脑倾向于使用表格。如果一个页面上包含多个段落和一个表格，工程师往往将目光移向表格。

创建表格时，请遵循以下原则：

- 用有意义的表头标记每列。不要让读者猜测每一列的内容。
- 避免在表格单元格中放入太多文本。如果一个表格单元格包含两个以上的句子，请问问自己该信息是否属于其他格式。
- 尽管不同的列可以保存不同类型的数据，但是要*在*各个列中尽可能的做到平行性。例如，特定表格列中的单元格不应是数字数据和著名的马戏团大象的混合。

注意：某些表格不能很好地呈现所有形式的信息。例如，在笔记本电脑上看起来不错的表格在手机上可能看起来很糟糕。

介绍每个列表和表格

我们建议在每个列表和表格前加上一句话，告诉读者该列表或表格代表什么。换句话说，给出列表或表格上下文：即使用冒号结束的介绍性句子。

尽管不是必需的，但我们建议将“ **following** ”一词放入介绍的句子中。参考如下例子：：

The following list identifies key performance parameters:

以下列表确定了关键性能的参数:

Take the following steps to install the Frambus package:

采取以下步骤安装 Frambus 软件包:

The following table summarizes our product's features against our key competitors' features:

下表总结了我们产品和主要竞争对手的特点：

练习

为下表写一个介绍性句子：

语言能力	发明者	推出年份	关键特点
Lisp	约翰·麦卡锡	1958年	递归
C++	比尼亚·斯特鲁斯特鲁普（Bjarne Stroustrup）	1979年	面向对象
Python	吉多·范·罗苏姆（Guido van Rossum）	1994年	简单

The following table contains a few key facts about some popular programming languages:

下表展示了一些流行编程语言的关键信息：

The following table identifies the inventor, year of invention, and key feature of three popular programming languages:

下表列出了三种流行编程语言的发明者、发明年份和主要特征：

段落

本节提供了一些构建内聚段落的指导原则。但是首先，先读一下下面的这段话：

写作的流程很*简单*：理清主题各部分之间的依赖关系，并以逻辑流的形式呈现这些部分，使读者能够理解你。

写一个精彩的开头句

开头句子是任何段落中最重要的句子。忙碌的读者只关注开头的句子，有时会跳过后面的句子。因此，将精力集中在开头句上。

好的开头句确立了段落的中心点。以下段落就是个好例子：

循环会多次运行同一代码块。例如，假设你编写了一段代码来检测输入行是否以句号结尾。要计算一百万条输入行，那请创建一个运行一百万次的循环。

前面的开头句将段落的主题确立为循环的简介。相比之下，以下开篇句子将读者引向错误的方向：

代码块是相同功能内的连续代码集。例如，假设你编写了一段代码来检测输入行是否以句点结尾。要评估一百万条输入行，请创建一个运行一百万次的循环。

注意：有效的开头句可以采用多种形式。也就是说，并非所有出色的段落都以陈述主题的句子开头。例如，以反问开头的段落可以吸引读者。

一个段落聚焦一个主题

一个段落应代表一个独立的逻辑单元。将每个段落限制为当前主题。不要描述将来的话题会发生什么或过去的话题会发生什么。当我们要修改编辑时，一定要毫不犹豫地删除（或移至另一段）任何与当前主题无直接关联的句子。

下面例子中，假设开头句正确表达了段落的主题，你能找到其中不符合该主题的句子吗？

毕达哥拉斯定理指出，直角三角形的两条直角边的平方和等于斜边的平方。三角形的周长等于三个边的总和。你可以使用勾股定理来测量对角线距离。例如，如果你知道乒乓球桌的长度和宽度，则可以使用勾股定理确定对角线距离。要计算乒乓球桌的周长，请将长度和宽度相加，然后将其乘以2。

我们删去了第二和第五句话，得出了一个专门针对勾股定理的段落：

毕达哥拉斯定理指出，直角三角形的两条直角边的平方和等于斜边的平方。~~三角形的周长等于三个边的总和~~。你可以使用勾股定理来测量对角线距离。例如，如果你知道乒乓球桌的长度和宽度，则可以使用勾股定理确定对角线距离。~~要计算乒乓球桌的周长，请将长度和宽度相加，然后将其乘以2。~~

段落不要太长或太短

长段落在视觉上令人生畏。很长的段落构成了可怕的“文字墙”，读者会忽略。读者通常欢迎只有三到五个句子的段落，同时忽略超过七个句子的段落。修订时，请考虑将很长的段落分为两个单独的段落。

相反，段落也不要太短。如果你的文档包含大量的单句段落，则说明你语言组织不够。请设法将这些零散段落组合成连贯的段落，或者组合成列表形式。

回答what，why 以及 how

好的段落回答以下三个问题：

1. **What -** 告诉读者这是什么
2. **Why -** 为什么让读者知道这一点很重要
3. **How -** 读者应如何使用这些知识。或者，读者应该如何知道你的观点是正确的？

下面例子就回答了 What，Why 以及 How：

<Start of What> 该 grap() 函数返回一个数据集的平均值和中位数之间的增量。**<End of What>** **<Start of Why>** 许多人毫不怀疑地相信，平均值之道总是真理。但是，平均值很容易受到几个非常大或非常小的数据点的影响。**<End of Why>** **<Start of How>**调用garp()可以帮助确定是否有几个非常大或非常小的数据点对均值的影响太大。相比较高的grap()值，较小的grap()值说明平均值更有意义。**<End of How>**

受众

我们认为你应该喜欢数学。因此，本章节以一个公式开始：

好的文档 = 你的受众完成一项任务所需的知识和技能 - 你的受众当前的知识和技能

换句话说，请确保你的文档提供了受众所需的信息，而受众还没有这些信息。因此，本章节说明了如何执行以下操作：

- 定义你的受众群体。
- 确定你的读者需要学习的内容。
- 提供合适你读者的文档。

如以下视频所示，针对*错误的*受众可能会很混乱：https://youtu.be/eFtXIrmsMwI

定义你的受众

认真的文档工作会花费了大量时间和精力来定义他们的受众。这些工作可能进行用户调查，用户体验研究，聚焦小组和文档测试。你可能没有那么多时间，所以本章节采用一种更简单的方法。

首先确定你的受众**角色**。示例角色包括：

- 软件工程师
- 技术非工程师角色（例如技术项目经理）
- 科学家
- 科学领域的专业人员（例如医师）
- 工科本科生
- 工科研究生
- 非技术职位

我们很高兴地认识到，许多非技术人员具有出色的技术和数学技能。但是，角色仍然是定义受众时必不可少的。具有相同角色的人通常对某些基本技能和知识有一定的共识。例如：

- 大多数软件工程师都知道流行的排序算法，Big O 符号和至少一种编程语言。因此，你可以假定软件工程师了解O(n)的含义，但不能假定非技术角色也可以了解O(n)。
- 针对同一研究项目，医生的专业报告和普通受众的报纸文章应该看起来有很大的不同。
- 大学教授对研究生进行一种新型的机器学习方法的教学方法，会完全不同于对本科生的教学方法。

如果处于同一角色的每个人都共享完全相同的知识，那么写作会容易得多。不幸的是，同一角色内的知识也会迅速产生分支。Amal是Python方面的专家，Sharon的专业知识是C++，Micah的专业知识是Java。Kara喜欢Linux，但是David只知道iOS。

角色本身不足以定义受众。也就是说，你还必须考虑受众对知识的*接近程度*。Project Frombus的软件工程师对相关的Project Dingus有所了解，但对无关的Carambola项目一无所知。普通的心脏病专家比普通的软件工程师对耳朵问题的了解要多，但比听觉专家要少得多。

时间也会造成差距。例如，几乎所有软件工程师都学过微积分。但是，大多数软件工程师在工作中并不使用微积分，因此他们对微积分的了解逐渐消失。相反，与同一项目的新工程师相比，经验丰富的工程师通常对当前项目的了解要多得多。

样本受众分析

以下是虚拟Zylmon项目的样本受众分析：

Zylmon项目的目标受众包括以下角色：

- 软件工程师
- 技术产品经理

目标受众在以下方面有相近的知识：

- 我的目标受众已经知道Zyljeune API，它们与Zylmon API有点相似。
- 我的目标读者知道C ++，但通常没有在新的Winged Victory开发环境中构建C ++程序。
- 我的目标受众是大学里的线性代数，但团队的许多成员都需要复习矩阵乘法。

确定受众可以学到什么

写下目标受众学习以实现目标所需的一切清单。在某些情况下，列表应包含目标受众需要*执行的*任务。例如：

阅读文档之后，受众将知道如何执行以下任务：

- 使用Zylmon API按价格列出酒店。
- 使用Zylmon API可以按位置列出酒店。
- 使用Zylmon API通过用户评分列出酒店。

请注意，你的受众有时必须按一定顺序完成任务。例如，你的受众可能需要*在*学习如何编写特定类型的程序*之前，先*学习如何在新的开发环境中构建和执行程序。

如果你正在编写设计规范，那么你的列表应该关注目标受众应该学习的信息，而不是精通特定任务：例如：

阅读设计规范后，受众将学到以下内容：

- Zylmon胜过Zyljeune的三个原因。
- Zylmon花了5.25个工程年来开发的五个原因。

使文档适合受众

满足受众需求的写作需要无私的同理心。你必须创建满足受众好奇心而不是你自己的解释。为了使文档适合受众，你如何跳出自己的脚步？不幸的是，我们无法提供简单的答案。但是，我们可以提供一些要重点关注的参数。

词汇和概念

使你的词汇与受众匹配。请参阅单词以获取帮助。

注意亲近。你团队中的人员可能理解你团队的缩写，但是，其他团队中的人员是否也理解相同的缩写？随着目标受众的扩大，假设你必须进行更多说明。

同样，软件团队中经验丰富的人员可能了解团队项目的实施细节和数据结构，但几乎其他所有人（包括团队的新成员）都不了解。除非你是专门为团队中其他经验丰富的成员撰写的，否则通常你必须解释的内容超出了你的预期。

知识的魔咒

专家经常遭受知识的魔咒，也就是说，他们对主题的专业理解破坏了对新手的解释。作为专家，很容易忘记新手不知道你已经知道的知识。新手可能不理解解释，而这些解释需要专家们不停地用和各种引用以及更深的知识来进行各种解释。

从新手的角度来看，知识的磨咒就好像是由于模块尚未编译而导致的“找不到文件”链接器错误。

简单的话

英语已经成为全球技术交流的主要语言。但是，英语并不是大多数技术读者的母语。因此，更喜欢简单的单词而不是复杂的单词。避免使用奥术，过时或过于复杂的英语单词；倍半数[1]和稀有词会排斥大多数读者。

文化中立和成语

使你的作品保持文化中立。不需要读者了解NASCAR[2]，板球或相扑的复杂性，以了解某个软件的工作原理。例如，下面的句子（加上像苹果派一样的美国人的棒球比喻）可能会使某些巴黎读者感到困惑：

```
If Frambus 5.0 was a solid single, Frambus 6.0 is a stand-up double.
```

[1] 一种测量单位，一英尺半

[2] （美国）全国运动汽车竞赛协会（National Association for Stock Car Auto Racing的缩写）

习惯用语是短语，其整体含义不同于该短语中各个单词的字面含义。例如，以下短语是成语：

- a piece of cake[3]
- Bob's your uncle[4]

蛋糕？鲍勃？大多数美国读者都认可第一个习惯用法。大多数英国读者都认可第二种成语。如果你是专门为英国观众写的书，那么 *鲍勃的叔叔* 就可以了。但是，如果你要为国际读者写作，那么用*此任务*替换该惯用法*很容易*。

习语在我们的演讲中根深蒂固，以至于习语的特殊非文字含义对我们来说是不可见的。也就是说，习语或成语是知识魔咒的另一种形式。

请注意，受众中的某些人使用翻译软件来阅读你的文档。比起简单朴素的英语，翻译软件往往在文化参考和习语方面很无力。

文档

你可以写句子，也可以写段落。但是，你可以将所有这些段落组织成一份连贯的文档吗？

说明文件范围

一个好的文档首先要定义其范围。例如：

```
本文档描述了Project Frambus的总体设计。
```

更好的文档还定义了未覆盖的范围，即目标读者可能希望文档涵盖但是文档并没有未涵盖主题。例如：

```
本文档未未涉及 Froobus 项目的相关技术设计。
```

这些作用域和非作用域语句不仅使读者受益，而且使作者（你）受益。在编写时，如果文档的内容偏离范围声明，则必须重新调整文档的范围或修改范围声明。在查看初稿时，请删除（或分支到另一个文档）任何不利于满足范围说明的部分。

---

[3] 小菜一碟
[4] 易如反掌

陈述你的受众

好的文档明确指定了它的受众。例如：

```
我为支持Project Frambus的测试工程师编写了此文档。
```

除了受众的角色之外，良好的受众声明还可以指定任何必备的知识或经验。例如：

```
本文档假定你了解矩阵乘法以及如何冲泡一杯真正好的茶。
```

在某些情况下，受众声明还必须指定先决条件文件。例如：

```
在阅读本文档之前，你必须阅读“ Project Froobus：A New Hope”。
```

预先建立关键观点

工程师和科学家们很忙，他们不一定会阅读你全部76页的设计文档。想象一下，你的同龄人可能只会阅读第一页的第一段。在查看文档时，请确保文档的开头回答了读者的基本问题。

专业作家将大量精力放在第一页上，以增加读者进入第二页的几率。但是，任何长文档的第一页都是最难写的页面。因此，准备好多次修改页面。

始终为长的工程文档写一份总结性摘要（TL; DR[5]）。尽管这个摘要必须非常简短，但是应该花很多时间来编写它。无聊或令人困惑的执行摘要是一个危险信号，会让你的读者直接你的文档丢到垃圾桶。

为受众写作

本课程反复强调定义受众的重要性。在本节中，我们将重点放在受众定义上，以将其组织为文档。

定义受众

回答以下问题有助于你确定文档应包含的内容：

[5] Too Long, Don’t Read

- 谁是你的目标受众？
- 你的读者在阅读文档之前已经知道什么？
- 你的读者在阅读你的文档后应该知道或能够做什么？

例如，假设你已经发明了一种新的排序算法。以下列表包含对以上问题的一些潜在答案：

- 我的目标受众是组织中的所有软件工程师。
- 我的大多数目标受众在学校期间都学习了排序算法。但是，多年以来，大约有25%的目标受众尚未实现或评估排序算法。
- 阅读此文档后：
    1. 读者知道该算法如何工作。
    2. 读者可以用所需的语言来实现算法。
    3. 读者知道在什么情况下该算法要优于流行的快速排序算法。
    4. 读者了解某些情况下的性能下降。

组织

定义受众之后，整理文档以提供读者在阅读文档后应该知道或能够做的事情。例如，文档的大纲可能如下所示：

1. 算法概述
    1. Big O 复杂度
    2. 用伪代码实现
2. 用C语言实现的示例
    1. 使用其他语言实现的提示
3. 更深入的算法分析
    1. 最佳数据集
    2. 边界案例问题

此外，使用受众群体定义可帮助你选择编写文档的正确方法。例如，目标受众研究了排序算法，但是大约四分之一的受众可能不记得不同算法的细节。因此，你的文档可能应该插入指向 `quicksort` 的现有教程的链接，而不是试图解释 `quicksort`。

将主题分为几个部分

将代码模块化为文件，类和方法，可以让代码更易于阅读，理解，维护和重用。文档模块化可为你带来相同的好处。你可能对代码中的功能模块化有很强的直觉，但是如何将这些原理应用于你的写作中？

想象一下，你有一个空罐子，需要把一大堆岩石，粗砂砾和沙子打包。你将如何打包它们以确保可以将所有物料放入罐子中？显然，你需要先放置大石头，然后倒入砾石中，然后用沙子填充剩余的间隙。如果试图以相反的顺序执行此操作，将会失败。

读者的脑袋就像一个空罐子，你的信息通常分为三种粒度：岩石，砾石和沙子。

信息大纲是岩石，帮助读者构造基本结构，就像岩石首先填充罐子，以更好接受其余内容。但是，你如何确定什么是岩石什么是砾石？

一种策略是花 2~5 分钟的时间，发散性地随意记录或默念些什么。没错，这需要集中注意力，检查你在这个过程是否做到了以下事项？

- 用模糊不清的方式描述概念？
- 列出你的受众达到目标所需要完成的步骤？
- 用列表描述一个系统的性质？

上面提到模糊不清的概念可能就是主题的核心，如果你的演讲没有做到这一点，请回过头来试试这种方式。

标点（可选）

逗号

编程语言强制执行有关标点的明确规则。相反，在英语中，有关逗号的规则有些模糊。作为指导，在读者自然会在句子中某处停顿的地方插入逗号。用音乐上来做类比，如果句号是整个音符的休止符，那么逗号可能是半音符或四分音符的休止符。换句话说，逗号的暂停要短于一段时间。例如，如果你大声阅读以下句子，则可能会在单词 *just* 之前短暂*停留*：

```
C behaves as a mid-level language, just a couple of steps up in abstraction from assembly language.
```

有些情况下*需要*逗号。例如，使用逗号分隔嵌入式列表中的项目，如下所示：

```
我们公司使用C++，Python，Java和JavaScript。
```

你可能想知道列表的最终逗号，该逗号插入在项目N-1和N之间。该逗号（称为**串行逗号**或**牛津逗号）是**有争议的。我们建议仅提供最后一个逗号，因为技术写作需要选择最不明确的解决方案。也就是说，我们实际上更喜欢通过将嵌入列表转换为项目符号列表来规避争议。

在表达条件的句子中，在条件和结果之间放置逗号。例如，以下两个句子在正确的位置提供逗号：

If the program runs slowly, try the `--perf` flag.

If the program runs slowly, then try the `--perf` flag.

你还可以在一对逗号之间进行快速定义或离题，如以下示例所示：

Python, an easy-to-use language, has gained significant momentum in recent years.

最后，避免使用逗号将两个独立的想法粘贴在一起。例如，以下句子中的逗号是被称为**逗号拼接**的标点重罪：

萨曼莎（Samantha）是一位出色的编码员，她编写了大量测试。

使用句号而不是逗号来分隔两个独立的思想。例如：

萨曼莎（Samantha）是一位出色的编码员。她写了大量的测试。

分号

一个时期将不同的思想分开；分号统一了高度相关的思想。例如，请注意以下句子中的分号如何结合了第一和第二个想法：

更新配置文件后，重新运行Frambus；更新现有源代码后，请勿重新运行Frambus。

分号之前和之后的思想必须全部是语法完整的句子。例如，以下分号是*错误的*，因为分号后面的段落不是完整的句子：

Rerun Frambus after updating your configuration file; not after updating existing source code.

在使用分号之前，请先问问自己，如果你将思想转到分号的相对两侧，该句子是否仍然有意义。例如，反转前面的示例仍会产生一个*有效的*句子：

Don't rerun Frambus after updating existing source code; rerun Frambus after updating your configuration file.

你几乎应该始终使用逗号（而不是分号）来分隔嵌入列表中的项目。例如，以下分号的使用是*不正确的*：

Style guides are bigger than the moon; more essential than oxygen; and completely inscrutable.

许多句子在分号后立即放置过渡词或短语。在这种情况下，请在转换后放置逗号。请注意以下两个示例中过渡后的逗号：

Frambus provides no official open source package for string manipulation; however, subsets of string manipulation packages are available from other open source projects.

Even seemingly trivial code changes can cause bugs; therefore, write abundant unit tests.

破折号

破折号是引人注目的标点符号，具有丰富的标点符号可能性。破折号比逗号表示更长的暂停（更大的中断）。如果逗号是四分音符休止符，则破折号是半音符休止符。例如：

C++ is a rich language—one requiring extensive experience to master.

编写者有时使用一对破折号来阻止题外话，如以下示例所示：

Protocol Buffers—often nicknamed **protobufs**—encode structured data in an efficient yet extensible format.

我们可以在前面的示例中使用逗号而不是破折号吗？当然。为什么我们选择破折号而不是逗号？感觉。艺术。经验。请记住，英语的标点符号是糊状且具有延展性的。

括号

使用括号可以保留次要点和题外话。括号告诉读者，所附文字并不重要。

有关句号和括号的规则使许多写作者大跌眼镜。以下是标准：

- 如果一对圆括号包含整个句子，则句号在右圆括号内。
- 如果一对括号使一个句子结尾但不包含整个句子，则句点就在右括号之外。

例如：

(Incidentally, Protocol Buffers make great birthday gifts.)

Binary mode relies on the more compact native form (described later in this document).

Markdown （可选）

Markdown是一种轻量级的标记语言，许多技术专业人员使用它来创建和编辑技术文档。使用Markdown，你可以在纯文本编辑器(如vi或Emacs)中编写文本，插入特殊字符来创建标题、加粗、项目符号等。例如，下面的示例显示了一个用Markdown格式化的简单技术文档

```
## bash and ksh

**bash** closely resembles an older shell named **ksh**.  The key
*practical* difference between the two shells is as follows:

*  More people know bash than ksh, so it is easier to get help for
bash problems than ksh problems.
```

上述的文档会被渲染成如下的样子：

bash and ksh

bash closely resembles an older shell named **ksh**. The key *practical* difference between the two shells is as follows:

- More people know bash than ksh, so it is easier to get help for bash problems than ksh problems.

Markdown解析器将Markdown文件转换成HTML。然后，浏览器可以将生成的HTML显示给读者。我们建议你通过学习下面的教程来熟悉Markdown

- www.markdowntutorial.com
- Mastering Markdown

小结

- 始终使用术语。

- 避免模棱两可的代词。
- 首选主动语态而不是被动语态。
- 选择强动词。
- 选择具体的动词而不是模糊的动词。
- 将每个句子聚焦在一个事上。
- 将一些长句子转换为列表。
- 消除不必要的单词。
- 如果次序重要，请使用编号列表；与次序无关，则使用项目符号列表。
- 保持列表项的平行属行。
- 用祈使性单词开始编号的列表项。
- 正确地介绍列表和表格。
- 创建漂亮的开头句，以建立本段落的中心思想。
- 将每个段落集中在一个主题上。
- 确定你的受众需要学习的内容和目标。
- 让文档贴近你的受众。
- 在文档开头处说明文档的主要观点。

第二部分

自我编辑

想象一下，你刚刚编写了文档的初稿。你如何做得更好？在大多数情况下，达到可最终发布的文档是一个反复迭代的过程。从零开始形成初稿通常是最困难的步骤。编写完初稿后，还要确保能留出大量时间来完善文档。

本章节中的编辑技巧可以帮助你将初稿转换为文档，从而更清晰地传达你的受众所需的信息。你可以使用其中的一个小或全部的技巧；但重要的是找到适合你的策略，然后将该策略作为写作日常的一部分。

注意：本章节的技巧以技术写作一的基本写作和编辑技能为基础。本单元总结了该课程中有用的编辑技术。有关更详细的复习，请访问技术写作的第一部分 自学单元。

采用样式指南

公司，组织和大型开源项目经常采用已有的风格样式。Google Developers网站上的许多文档项目都遵循《 Google Developer文档样式指南》 。如果你以前从未使用过样式指南，那么乍一看，《 Google Developer文档样式指南》可能看起来有点吓人，它提供了有关语法，标点符号，格式设置和计算机接口文档等主题的详细指南。所以，你可能更喜欢从 《样式指南精要》 开始。

注意：对于较小的项目，例如团队文档或小型开源项目，《样式指南精要》已经够你用了。

精要内容中列出的一些准则在第一部分中有介绍。你应该可以回想起一些：

- 使用主动语态来明确谁在执行操作。
- 将顺序步骤格式化为 有序列表。
- 将其他列表格式化为无序列表。

精要还介绍了许多其他技术，这些技术在编写技术文档时可能会有用，例如：

- 使用第二人称。称呼你的的受众称为“你”而不是“我们”。
- 将条件子句放在指令之前，而不是之后。
- 代码相关的文本格式设置代码字体。

像受众一样思考

你的受众是谁？退后一步，尝试从他们的角度阅读草稿。确保文档的目的明确，并为读者可能不熟悉的任何术语或概念提供定义和解释。

给你的受众设置一个“人设”可能会对写作有所帮助。其中可以包含以下任何属性：

- 角色，例如：*系统工程师* 或 *QA测试人员*。
- 最终目标，例如：*还原数据库*。
- 知识和经验的一组假设。例如，你可以假设受众的“人设”是：
    - 熟悉Python。
    - 运行Linux操作系统。
    - 平时在命令行的环境工作。

然后，你可以模拟他们来阅读初稿。告诉受众你所做的任何假设可能特别有用。你还可以提供指向资源的链接，如果他们需要复习特定主题，他们可以在其中了解更多信息。

请注意，过分依赖角色（或两个角色）可能会导致文档过于狭窄而无法对大多数读者有用。

有关技术写作一的复习和更多信息，请参阅受众自学单元。

大声朗读

不同的上下文或写作风格可能会疏远、吸引甚至打扰你的受众。给定文档设定什么样的样式在一定程度上取决于受众。例如，旨在招募新志愿者的开放源项目贡献者指南可能采用更非正式和对话的风格，而商业企业应用程序的开发者指南则更可能采用更正式的风格。

要检查你的写作是否有对话性，请大声朗读。识别尴尬的措词，太长的句子或其他不自然的内容。另外，你也可以尝试请其他人为你朗读你的草稿。

有关调整写作风格以适合受众的更多信息，请参阅样式和作者语气。

稍后再回来

在你编写了第一稿（或第二稿或第三稿）之后，请将其放在一边。一小时（或两到三个）后再回来阅读，这样在阅读时会有新鲜感。于是，你几乎总是会注意到一些可以改进的地方。

改变场景

一些写作者喜欢打印他们的文档，并且手里拿着一支红色铅笔，查看纸质的文档副本。这种换一种场景来审阅自己的作品时，可以帮助你找到需要改进的地方。如果要使这个经典技巧更富现代感，你可将文稿复制到其他文档中并更改字体，大小和颜色。

寻找同伴编辑

就像工程师需要同伴人审查他们的代码一样，写作者也需要其他编辑者来向他们提供有关文档的反馈。请他人来审阅你的文档，并给你具体的建设性意见。你的同伴编辑者不必是文档技术主题的主题专家，但他们确实需要熟悉你遵循的样式指南。

组织大型文档

你如何将大量信息整理成一个文档或网站？或者说，你如何将当前杂乱的文档或网站重组为通俗易懂的东西？以下策略可以帮助你：

- 整理组织文档
- 添加导航
- 逐步展开信息

何时编写大型文档

你可以将一组信息组织成较长的独立文档或一组较短的相关文档。一组较短的相关文档通常以网站，Wiki或类似结构化格式发布。

有些读者喜欢阅读长文档，有些没那么喜欢。这两类的读者会有以下不同的搜索行为：

- Hong 发现阅读长文档很困难并且容易迷失方向。他更喜欢使用站点搜索来查找答案。

- Rose喜欢独立的长文档。她经常使用浏览器内置的页面搜索功能查找到有用的信息。

那么，你应该将信息组织成一个独立文档还是网站中的一组文档？请参考以下准则：

- 对于新手来说，，How-to 操作指南、入门简介、和概念描述以较短的文档展示时，会更友好。例如，一个对文档完全陌生的读者，可能很难记住很多新的术语，概念和事实。请记住，你的受众阅读文档时，可能是想快速全面地了解该主题。
- 深入的教程，最佳实践指南和命令行参考页可以以长文档的方式出现，尤其是针对那些已经对工具和主题有一定经验的读者。
- 出色的教程一般可以依靠叙述的方式来引导读者完成较长文档中的一系列相关任务。但是，即使是大型教程，很多时候分割成较小的部分会更好。
- 许多较长的文档不需要一次读完。例如，用户通常浏览参考页面以搜索某个命令的参数或标志的说明。

本章节的其余部分介绍了一些编写长文档的技巧，例如教程和一些概念性指南。

整理文件

本节提出了一些规划长文档的技巧，包括创建大纲和起草引言。在完成文档的初稿之后，根据概述和简介进行复审，以确保你没有遗漏任何本来打算涵盖的内容。

文档大纲

从结构化的高层级的大纲开始，可以帮助你对相关的主题进行分组并确定哪里需要更多详细信息。大纲可帮助你在开始写作之前就先讨论主题。

你可能会发现将大纲视为文档的叙述很有用。这世上还没有一个编写大纲的标准方法，但是以下准则提供的实用技巧可能会对你有帮助：

- 在要求读者执行任务之前，请向他们解释为什么要执行任务。例如，以下要点说明了本教程中有关审核和改善网页可访问性的部分大纲：
    - 介绍浏览器插件；说明我们将使用审计报告的结果来修复一些错误。
    - 列出运行插件和审计网页可访问性的步骤。
- 将大纲的每个步骤限制为，描述概念或完成特定任务。
- 结构化大纲，以便文档在与读者最相关的时候才引入相关的信息。例如，当你的读者刚开始使用基础知识时，他们可能不需要在文档的简介部分中了解（或想要了解）项目的历史。如果你觉得项目的历史记录很有用，请在文档末尾添加指向此类信息的链接。

- 在概念性信息和实际步骤之间交替的文档可能是一种特别吸引人的学习方式。考虑解释一个概念，然后说明读者如何将其应用于示例项目或自己的工作中。
- 如果你要与将要审阅和测试文档的贡献者团队合作，则提纲特别有用。在开始写文档之前，请与你的撰稿人分享你的大纲，看看他们是否有任何建议。

大纲练习

对于本练习，请查看并更新以下长篇教程简介的高层级大纲。你可以重新排列，添加和删除主题。

```
##项目的历史

描述该项目的发展历史。

##先决条件

列出读者在开始之前应熟悉的概念，以及任何软件或硬件要求。

##系统设计

描述系统如何工作。

##受众

描述本教程的目标对象。

##设置教程

解释如何配置环境以跟进本教程。

##  故障排除

说明在实践本教时如何诊断并解决在以下情况下可能发生的潜在问题。

##有用的术语

列出读者需要遵循的术语定义教程。
```

以下是一种可能的解决方案：

```
##观众

描述了本教程的目标读者。

##先决条件
```

```
列出了读者在开始之前应熟悉的概念以及
所有软件或硬件要求。

##设置本教程

介绍如何配置你的环境以遵循本教程。

##有用的术语

列出了读者阅读本
教程需要了解的术语定义。
```

介绍文档

如果文档的读者找不到与该主题相关的内容，则他们基本上会忽略这个文档。要为你的用户设置基本规则，建议你提供一个包含以下信息的简介：

- 文档涵盖的内容。
- 你希望读者具备哪些先验知识。
- 该文件未涵盖的内容。

请记住，你想使文档易于维护，因此请勿尝试介绍中的所有内容。

下面的段落演示了前面列表中的思想，作为一个名为Froobus的假设文档发布平台的概述：

```
本文档说明了如何使用Froobus系统发布Markdown文件。

Froobus是一个运行在Linux服务器上的将Markdown文件转成HTML页面的发布系统。

本文档适用于熟悉Markdown语法。要了解语法，请参阅 Markdown 参考。你还需要了解在
Linux终端下的一些命令。

本文档不包含有关安装或配置Froobus发布系统的内容。有关Froobus的安装，请参阅使用入门
手册。
```

完成初稿后，请对照概述中设置的期望检查整个文档。你的简介是否提供你所涵盖主题的准确概述？你可能会发现把这种审核认为是文档的质量保证（QA）是很有效的。

介绍练习

对于本练习，请查看并修订以下介绍，以获取一种称为F@的假设编程语言的最佳实践指南。删除你认为与此无关的任何信息，并添加你认为丢失的任何信息。

```
本指南列出了使用F@编程语言的最佳实践。
F@是在2011年开发的一个开源社区项目。本指南提供了F@的代码风格指南。除了本指南中的最佳实践之外，确保你已安装并运行F@命令行Linter。该编程语言在健康行业中被广泛采用。
如果你有有关最佳做法列表的补充建议，请到F@的github上给我们开issue。
```

以下是一种可能的修改：

```
本指南列出了使用F@编程语言的最佳实践。

在阅读本指南之前，请先完成F@开发人员的入门教程。本指南提供了F@代码风格指南。除了
本指南中的最佳实践之外，确保你已安装并运行F@命令行Linter。

如果你有有关最佳做法列表的补充建议，请到F@的github上给我们开issue。
```

添加导航

为读者提供导航和路标，可确保他们能够找到所需的内容以及不需要的信息。

清晰的导航包括：

- 简介和摘要部分
- 一个清晰有逻辑的主题
- 有助于用户理解主题的标题和副标题
- 介绍该工具的概述
- 目录菜单，向用户显示他们在文档中的位置
- 链接到相关资源或更深入的信息
- 链接到下一步学习

以下各节中的技巧可以帮助你写出不错文档标题。

首选基于任务的标题

选择描述你的读者正在从事的任务的标题。避免使用不熟悉的术语或工具的标题。例如，假设你正在写创建新网站的过程。要创建站点，读者必须初始化`Froobus`框架。要初始化`Froobus`框架，必须运行 `carambola` 命令行工具。乍一看，在说明中添加以下任一标题似乎合乎逻辑：

- 运行`carambola`命令
- 初始化`Froobus`框架

除非你的读者已经非常熟悉该主题的术语和概念，否则最好使用更熟悉的标题，例如“*创建站点*”。

在每个标题下提供文字

大多数读者至少喜欢在每个标题下进行简短介绍以提供一些背景信息。避免在第二级标题之后放置三级标题，如以下示例所示：

```
##创建网站
###运行carambola命令
```

在此示例中，简要介绍可以帮助读者确定方向：

```
##创建网站
要创建站点，请运行`carambola`命令行工具。该命令会显示一系列提示，以帮助你配置站点。
###运行carambola命令
```

标题练习

帮助读者浏览你的文档可帮助他们找到成功使用你的工具所需的信息。通常，清晰，井井有条的目录或大纲就像地图一样，可以帮助用户导航工具的功能。

对于本练习，请改进以下大纲。你可以重新排列，添加和删除主题，也可以创建辅助条目。

```
关于本教程
进阶主题
建立资产导航树
定义资源路径
定义和建设项目
启动开发环境
定义和建设资源
下一步是什么
定义图像资源
受众
参考
建立图像资源
定义图像项目
建立影像专案
设置教程
选择教程资产根
关于本指南
```

以下是一种可能的修改：

```
##关于本教程

###受众
```

###关于本指南

###高级主题

##设置本教程

###选择本教程根目录

###启动开发环境

###构建资产导航树

###定义资源路径

##定义和构建资源

###定义图像资源

###构建图像资源

##定义和构建项目

###定义图像项目

###构建图像项目

##定义和构建数据库

###定义数据库

###建立数据库

##推送，发布和查看数据库

###推送数据库

###发布数据库

###查看数据库

##配置“点数据”的显示规则

###定义，配置和构建矢量数据

##另请参阅

###示例数据文件

```
##下一步做什么
```

循序递近

对于许多乐于按自己的节奏阅读文档的读者而言，学习新的概念，想法和技术是一个有益的体验。但是，过快地面对太多新概念和说明可能会令人不知所措。读者很可能会接受较长的文档的形是，文档会逐步地在需要的时候层层递近。以下技术可以帮助你写出这样的层层递进的文档：

- 尽可能地，在需要用到的地方引入新的术语和概念。
- 分解大块文字。为了避免在一个页面上出现多个较大的段落，请在适当的地方引入表格，图表，列表和标题。
- 分解大量步骤。如果你的复杂步骤列表特别长，请尝试将它们重新排列为较短的列表，以解释如何完成子任务。
- 从简单的示例和说明开始，然后逐步添加更多有趣和复杂的技术。例如，在创建表单的教程中，首先说明如何处理文本响应，然后介绍其他技术来处理多个选择，图像和其他响应类型。

插图

还记得老师给你分配的阅读篇章吗？你翻阅了教科书的指定部分，非常希望...是的，图片！查看插图比阅读文本有趣得多。实际上，在阅读技术资料时，绝大多数成年人还是小孩，他们仍然渴望图片而不是文字。

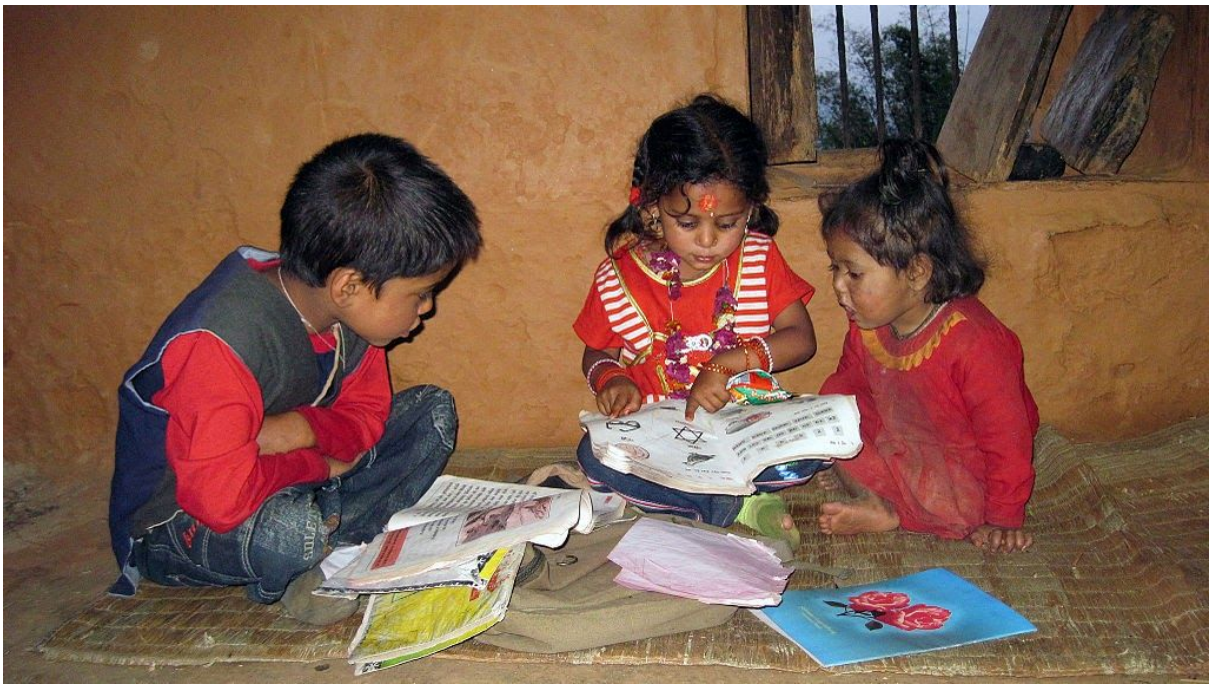


图1.良好的图片以文本无法不可比拟的方式吸引读者。

根据Sung和Mayer（2012）的研究，提供任何图片（无论好坏）都会使读者更喜欢该文档。但是，只有*教育性的*图片才能帮助读者学习。本章节提供了几种方法来帮助你创建真正胜过千言的图示。

首先写标题

通常，在创建插图之前写标题会很有帮助。然后，创建最能说明标题的插图。此过程可帮助你检查插图是否符合目标。

好的标题具有以下特征：

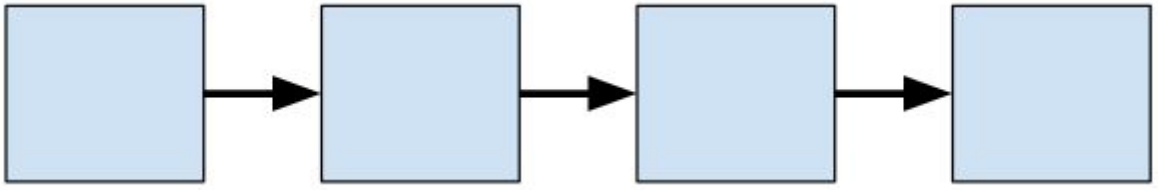
- 他们很**简短**。通常，标题只是几个单词。
- 他们解释了**要点**。查看此图形后，读者应该记住什么？
- 他们**聚焦**读者的注意力。当照片或图表包含很多细节时，聚焦尤其重要。

示例：

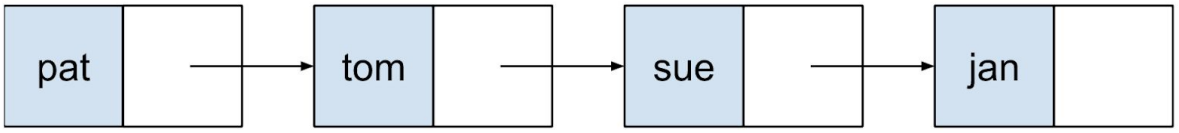
目标受众：CS本科生参加“数据结构入门”课程。

考虑以下三个图形，每个图形使用相同的标题。

标题A。单向链表包含内容和指向下一个节点的指针。



标题B。单向链表包含内容和指向下一个节点的指针。



标题C。单向链表包含内容和指向下一个节点的指针。

前三个图中的哪一个最能说明其标题？

- 图A很糟糕。链条很漂亮，但是没有信息。链还错误地暗示了一个单链表同时指向向后和向前。

- 图B是可以的。这个插图帮助学生认识到第一项指向第二项，第二项指向第三项，以此类推。但是，虽然标题同时引用了*内容*和*指针*，但是插图显示的是指针，而不是内容。
- 图C是最好的、最有启发意义的选择。插图清楚地描述了每个节点的内容部分和指针部分。

限制单个图片中的信息量

很少的智力工作可以像学习一幅精美的画作一样有收获，它逐渐揭示了洞察力和意义。人们为在世界艺术博物馆中做到这一点付出了很多钱。

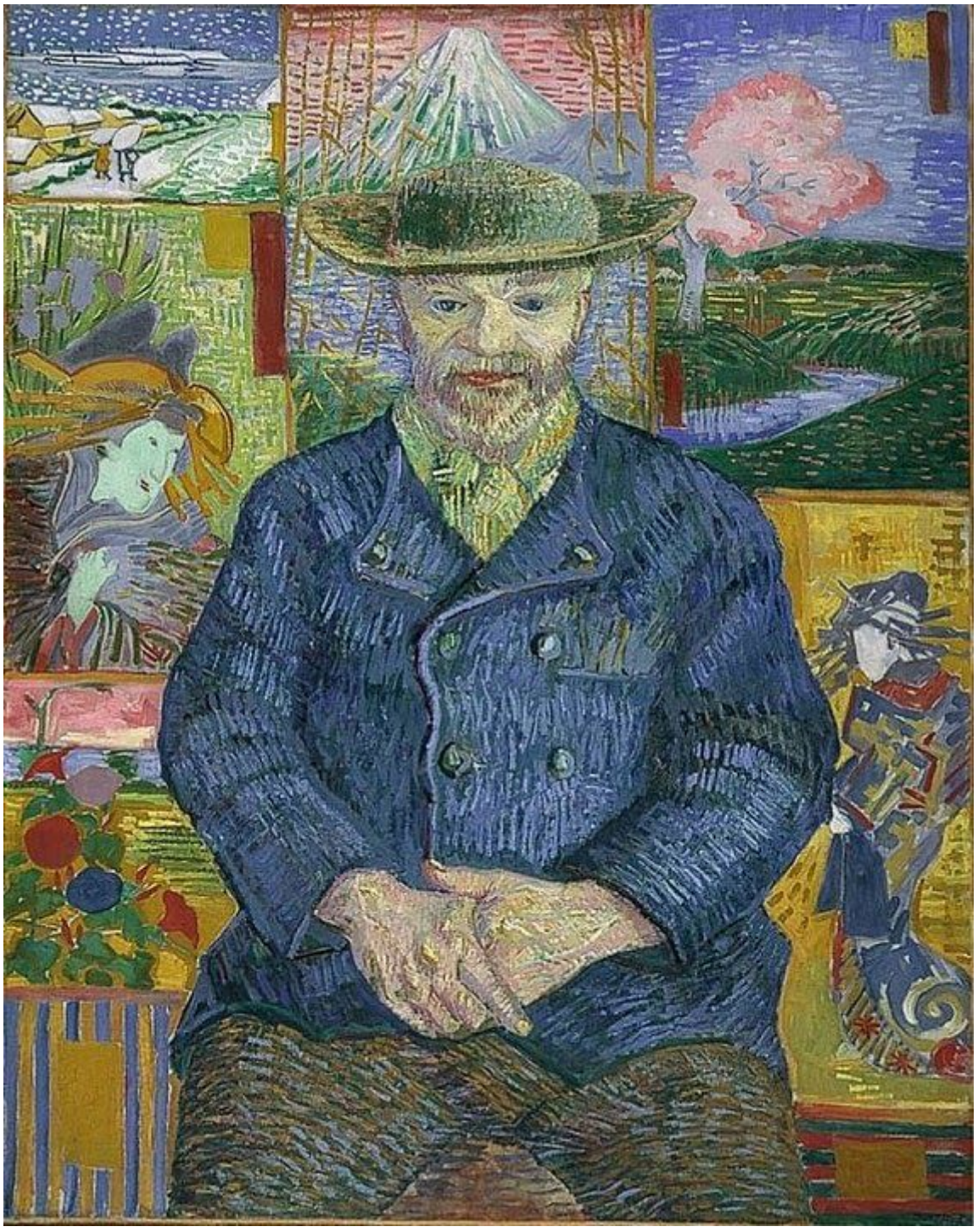


图2.你会很高兴研究梵高的这幅画。

Pere Tanguy的画像，文森特·梵高-罗丹博物馆[公共领域]

相比之下，如下所示的高度复杂的技术插图往往会阻止大多数读者：

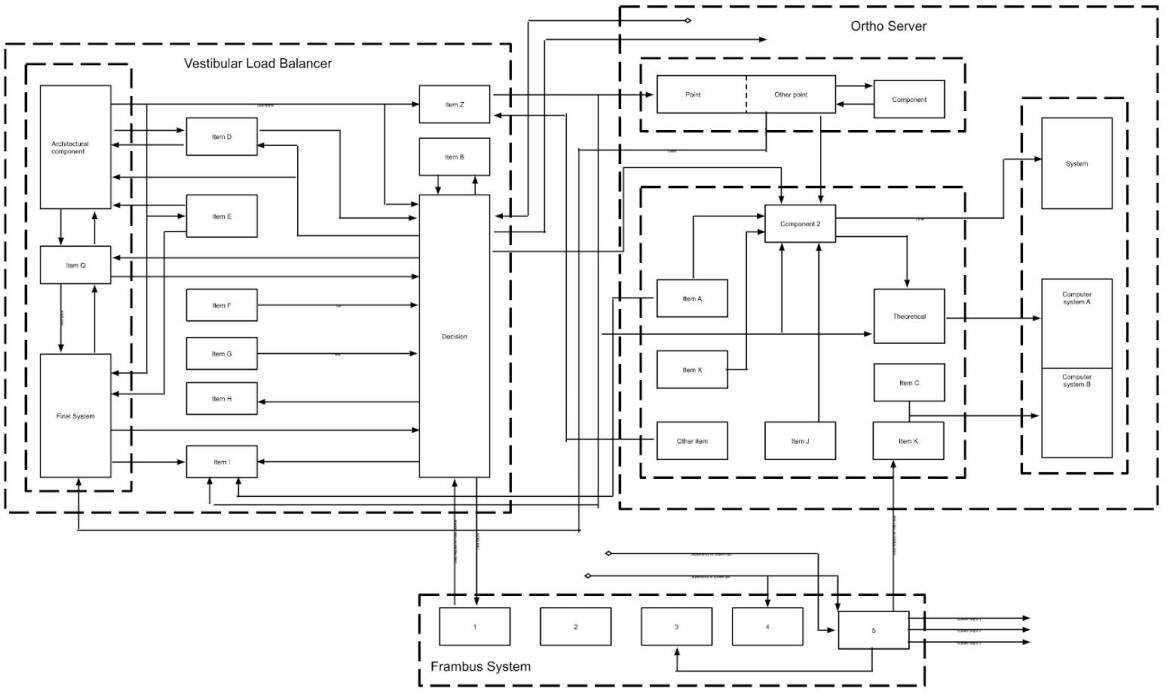


图3.复杂的框图使读者不知所措。

就像你避免过长的句子一样，也要努力避免视觉冲击。根据以往经验，不要在一个图表中放置超过一个段落的信息。（另一种经验法则是避免插图需要五个以上的无序列表来解释。）我听到你说："但是，现实生活中的技术系统可能比图3所示的系统复杂得多。"你是对的，但是你可能不会觉得要在单个段落中解释这个复杂的系统。

将视觉混乱变成连贯且有用的东西的诀窍是将复杂的系统组织成子系统，如下图所示：

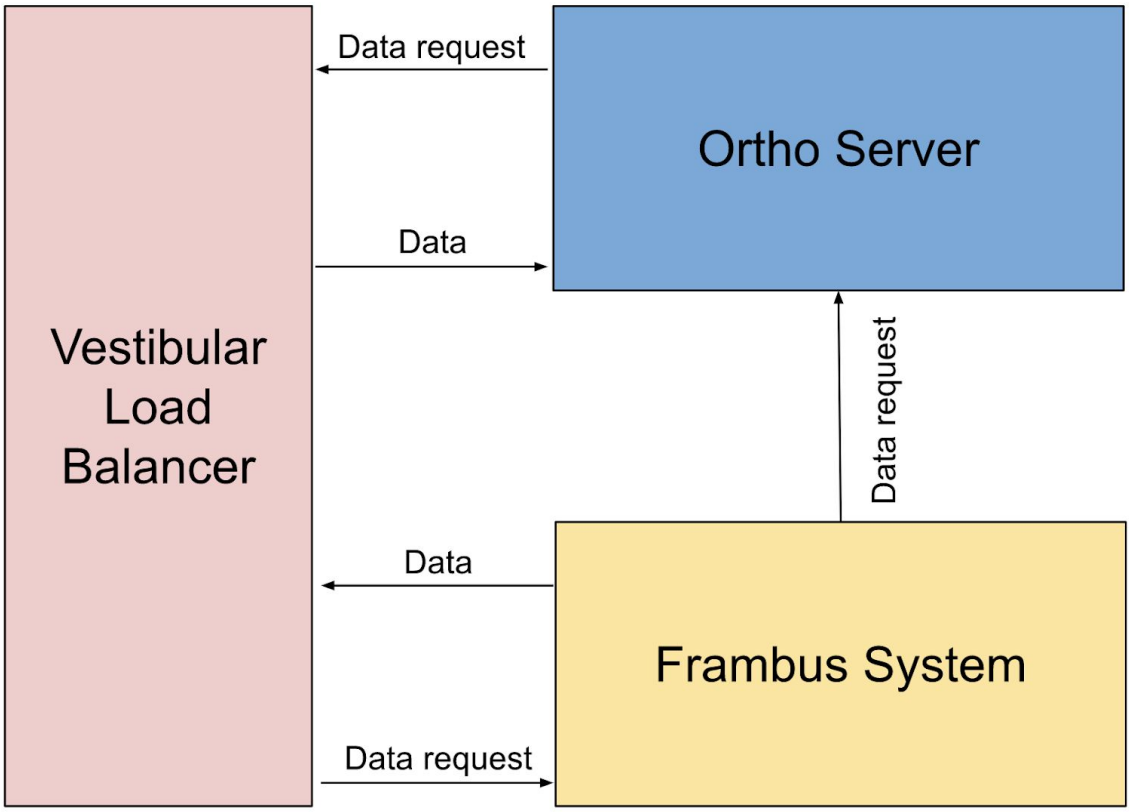


图4.分为三个子系统的复杂系统。

显示“大图”之后，分别提供每个子系统的图示。

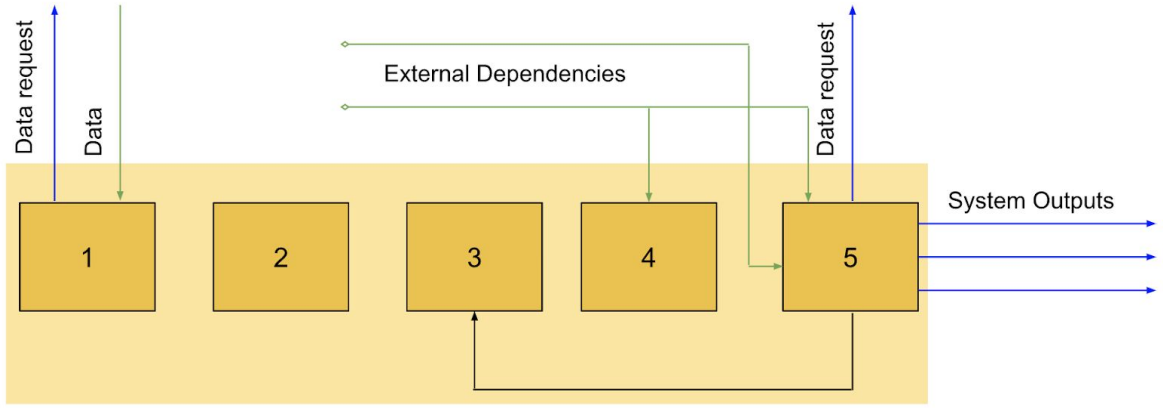


图5.复杂系统的一个子系统的扩展细节。

另外，可以从简单的“大图片”开始，然后在每个后续插图中逐步扩展细节。

吸引读者的注意力

面对如下复杂的屏幕截图时，读者难以确定相关内容：

图6.读者不知道该关注什么。

添加视觉提示，例如下图中的红色椭圆，可以帮助读者专注于屏幕截图的相关部分：

图7.读者会关注于破坏图案的形状。

标注提供了另一种吸引读者注意力的方法。对于图片和艺术线条，标注可以帮助我们的眼睛找到合适的落地位置。图片中的标注通常比图片的段落说明更好，因为标注将读者的注意力集中在图片的最重要的地方。然后，在你的解释中，你可以直接关注图表的相关部分，而不必花费时间描述你所讨论的图像的哪个部分。

在示例图像中，标注和箭头快速将读者引导至目标。

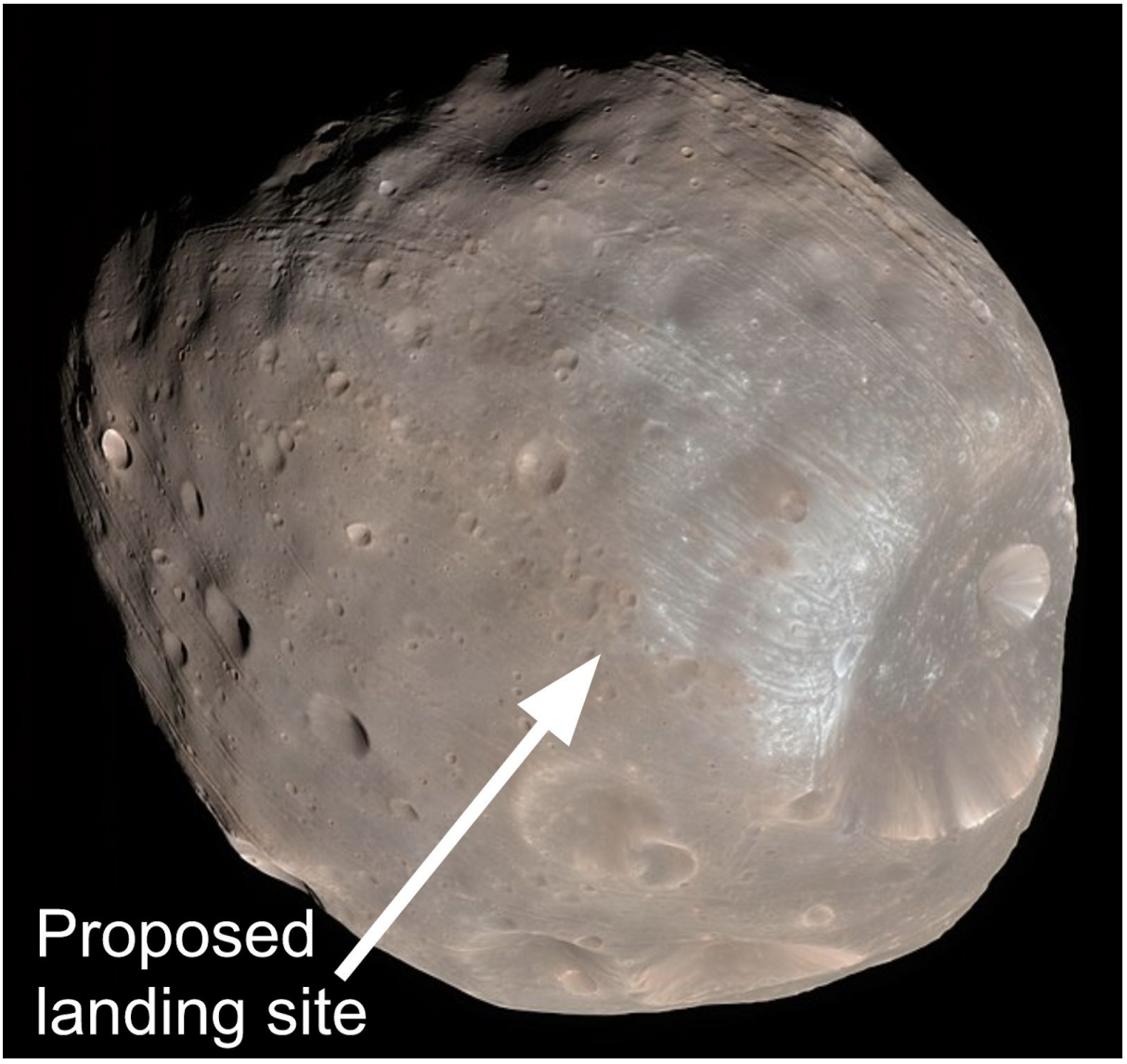


图8.标注引导读者的视线。

NASA / JPL-Caltech /亚利桑那大学[公共领域]

插图重构

与写作一样，插图的初稿也不够好。修改插图以澄清内容。修改时，请问自己以下问题：

- 如何简化图示？
- 我应该将此插图分为两个或更多个简单的插图吗？
- 插图中的文字是否易于阅读？文字与背景是否有足够的对比？
- 什么是重点？

例如，考虑伦敦地铁地图的 演变。在1931年之前，绘制了地铁地图，并按比例绘制了地上道路和像轨道一样弯曲的地铁线。

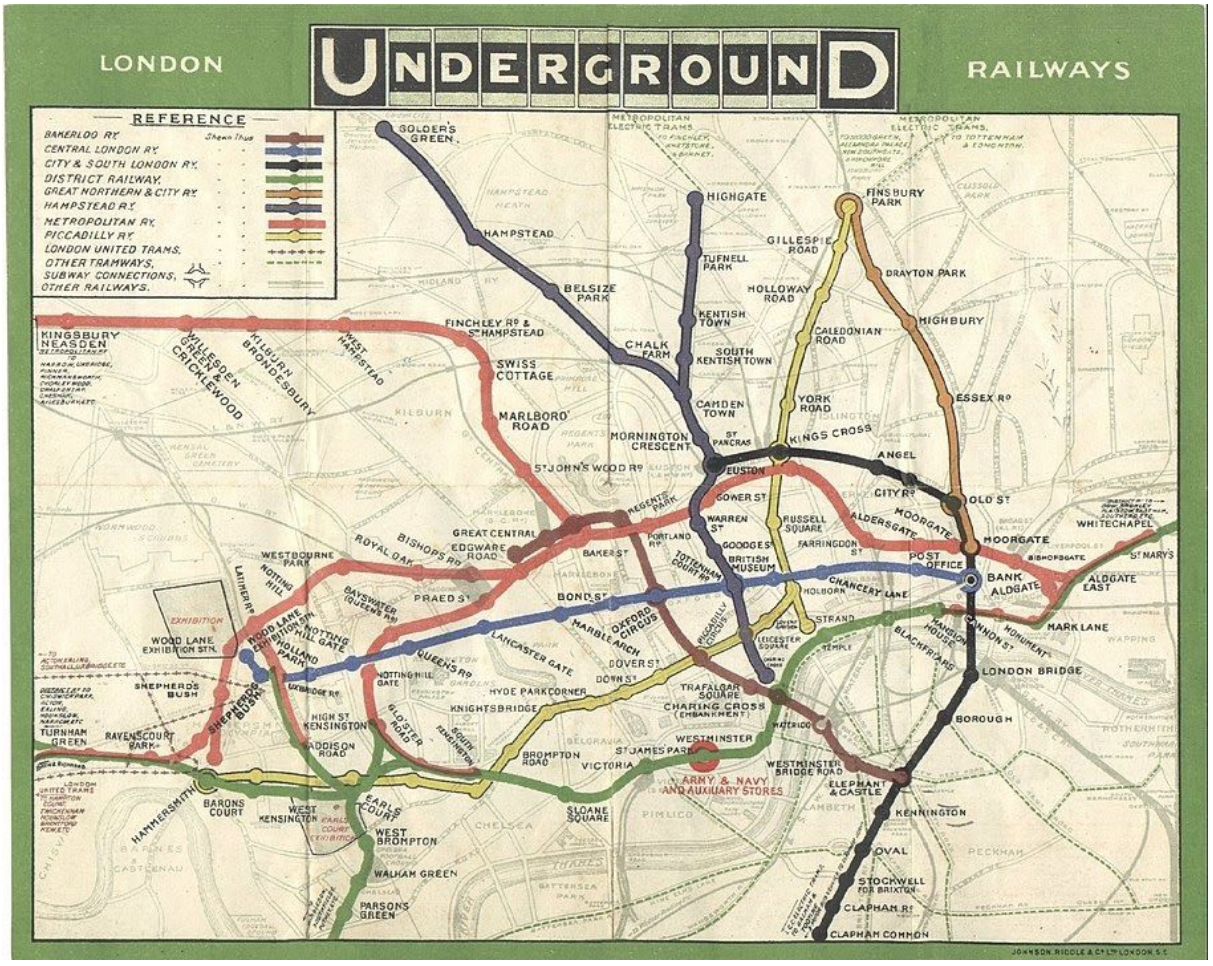
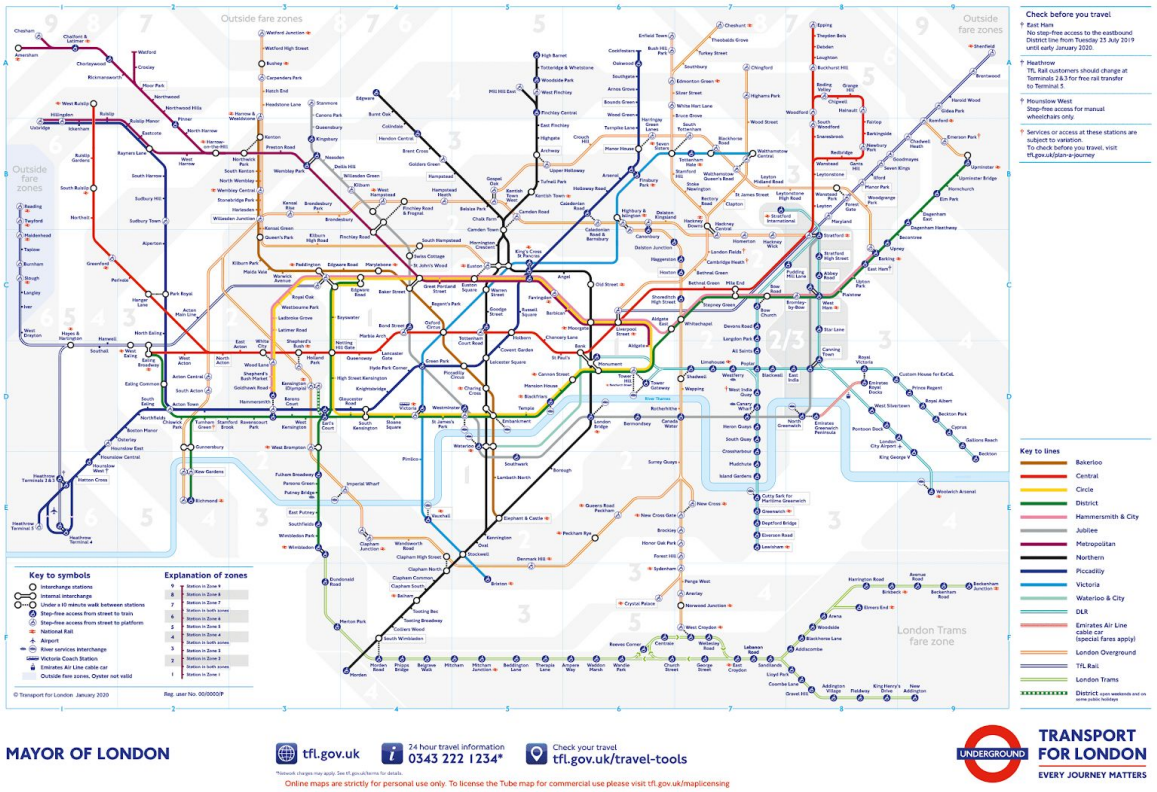


图9. 1908年的按地面街道绘制的伦敦地铁地图

[公共区域]

1931年，哈里·贝克（Harry Beck）革新了一种新型的公共交通地图，通过取消地上标记和比例尺来简化旧地图。相反，他的设计专注于使用地图的人们真正关心的是：从A站到B站。即使1931年的地图取得了成功，但贝克仍然多年对地图进行反复的简化和并使用清晰易读。现在考虑一下现代的地铁图，尽管出现了新的线路和车站，但它们仍然与贝克的设计保持接近。

Tube map



考虑以下原始插图：

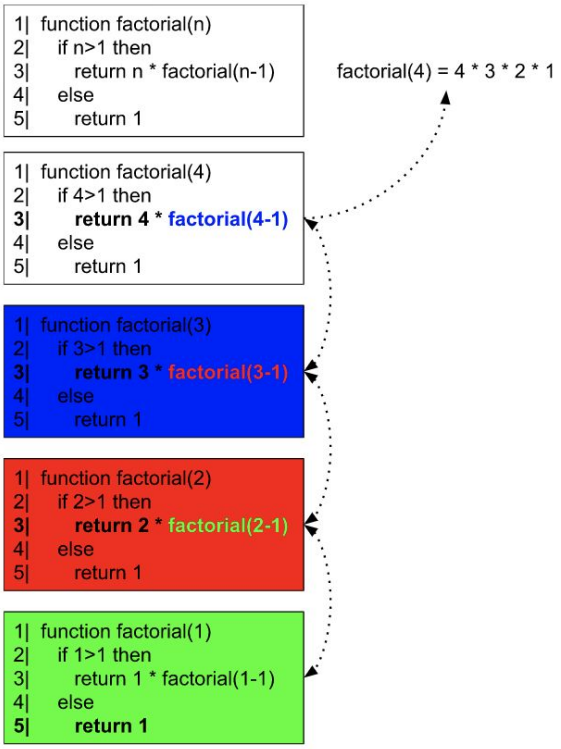


图10. 复杂图。

上图的内容应该是：

对于递归函数，在return语句中调用函数本身，直到获得基本解为止。

考虑这个图表的复杂性是以什么方式隐藏了要点？你将如何解决这些问题？

该图中可能存在的一些问题包括：

- **问题**：鲜艳的色彩使读者的注意力从图片的其他部分移开。
  解决方案：仔细选择颜色，以免它们压倒图片。
- **问题**：图片的色彩对比度不足。这使得某些视力较弱或某些类型的色盲的人无法访问该图。
  解决方案：消除不必要的颜色使用，并确保颜色 符合标准的颜色对比建议。
- **问题**：当前箭头指向两个方向，这使得该图的流程变得不清楚。
  解决方案：将箭头分为两部分，其中一组说明调用功能，另一组说明从功能返回。

当然，图中还有其他未在这里指出的问题。

这是一个改进的图示：

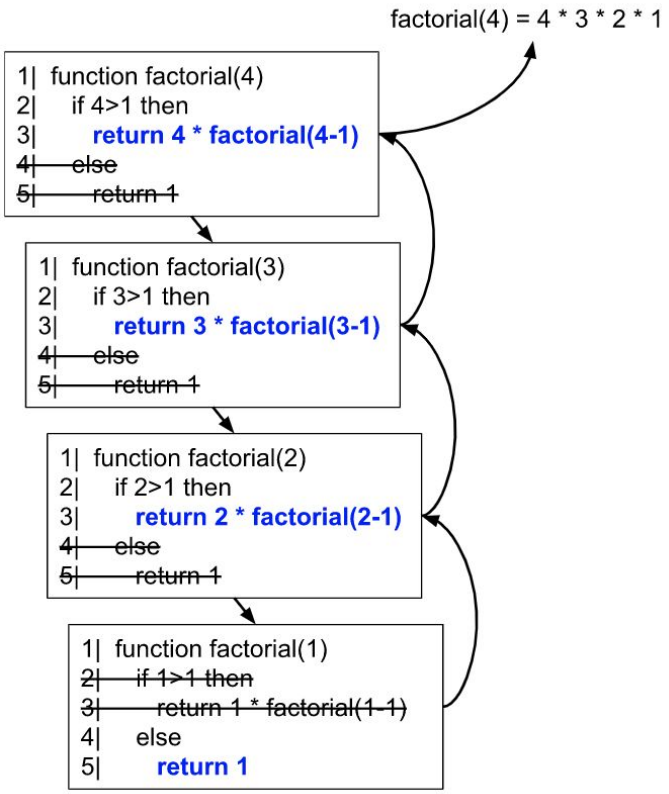


图11.上图的简化版本。

你在改进的插图中看到哪些缺陷？

这是两个仍然存在的缺陷：

- 该图仍然太复杂。要解释此插图，需要的不仅仅是段落。考虑删除多余的信息或添加说明标签如何简化解释。
- 当函数之间相互调用或返回数据时，分开显示箭头有助于显示，但返回箭头可能会受益于告诉读者返回值是什么的标签。

插图工具

有许多工具可用于创建图表。这里推荐三个免费的作图工具：

- Google Drawings
- Draw.IO
- LucidChart

从这些工具导出图表供文档使用时，通常最好将文件导出为SVG或可缩放矢量图形。可伸缩矢量图形可以轻松地根据空间限制来缩放图表，因此无论大小如何，你最终都可以得到高质量的图像。

示例代码

好的示例代码通常是最好的文档。即使你的段落和列表像水一样清澈，程序员仍然更喜欢好的示例代码。毕竟，文本是与代码不同的语言，并且它是读者最终关心的代码。尝试用文本描述代码就像尝试用英语解释一首意大利诗。

好的样本是**正确**，**简洁的**代码，你的读者可以**快速理解它们**，并以**最小的副作用轻松地重用它们**。

正确

示例代码应满足以下条件：

- 构建没有错误。
- 执行它要执行的任务。
- 尽可能是可以上生产的代码。例如，该代码不应包含任何安全漏洞。
- 遵循特定于语言的约定。

示例代码是直接影响用户编写代码的方式。因此，示例代码应示例最佳的编码方法。如果编码的方法不止一种，请以你的团队认为最好的方式对其进行编码。如果你的团队尚未考虑每种方法的利弊，请花一些时间去对比一下各种方式的好坏和优缺。

始终测试你的示例代码。随着时间的流逝，系统会发生变化，示例代码可能会出错。与其他任何代码一样，测试并维护你的示例代码。

许多团队将其单元测试用作示例程序，这有时是个坏主意。因为，单元测试的主要目标是测试，而示例程序的唯一目的是教育。

一个**程序片断**是示例程序的一个碎片，可能只有一行或几行长。这种程序片段繁多的文档通常会随着时间的推移而降级，因为团队倾向于不像完整的示例程序那样严格地测试程序片段。

运行示例代码

好的文档说明了如何运行示例代码。例如，在运行示例之前，你的文档可能需要告知用户执行以下活动：

- 安装某个库。
- 配置某些环境变量的值。
- 配置集成开发环境（IDE）。

用户并不总是正确执行上述活动。在某些情况下，用户喜欢直接在文档中运行或（使用）示例代码。（如："单击此处运行此代码。"）

编写者应考虑描述示例代码的预期输出或结果，尤其是对于难以运行的示例代码。

简洁

示例代码应该简短，仅包括基本组件。当C语言新手想学习如何调用该`malloc`函数时，请给该程序员一个简短的代码片段，而不是整个Linux源代码树。不相关的代码可能会使你的受众分散注意力并使他们困惑。也就是说，切勿使用错误的做法来缩短代码；总是喜欢正确而不是简洁。

易理解

请遵循以下建议来创建清晰的示例代码：

- 选择自描述性的类，方法和变量名。
- 避免读者难以理解的编程技巧。
- 避免深层嵌套的代码。
- 可选：使用粗体或彩色字体将读者的注意力吸引到示例代码的特定部分。但是，明智地使用突出显示——太多的突出显示意味着读者不会特别关注任何内容。

示例

以下哪个示例示例程序中的代码行会更有帮助？假设目标受众由`go.so`API的新软件工程师组成。

1. `MyLevel = go.so.Level(5, 28, 48)`
2. `MyLevel = go.so.Level(rank=5, 28, 48)`
3. `MyLevel = go.so.Level(rank=5, dimension=28, opacity=48)`

答案**3**是这里的最佳选择。尽管试图使示例代码尽可能短，但省略参数名称会使新手学习起来更加困难。

注释

遵循以下示例代码注释的建议：

- 保持简短，但易懂清楚始终高于简洁。
- 避免编写*明显*代码的注释，但也要记住，对你（专家）而言显而易见的东西可能对新手而言并不明显。
- 将你的评论精力集中在任何非直觉的代码上。
- 当你的读者非常有技术有经验，不解释*什么what*——代码是什么，解释*为什么 why*——代码为什么这么干。

你应该将代码的描述放在代码注释中还是在示例代码之外的文本（段落或列表）中？请注意，复制并粘贴代码的读者不仅会拷走代码，还会拷走其中的注释。因此，将代码的说明放入代码注释中以便一同复制粘贴。相比之下，当你必须解释冗长或难以解释的概念时，通常应将注释文本放在示例程序之前。

注意：如果必须牺牲能上生产环境质量的代码，以便使代码更短且更易于理解，请在注释中解释你的决定。

示例

你在以下代码片段的注释中看到什么问题？假设代码针对的是不熟悉`br`API但对流的概念有一定经验的程序员：

```
/ *从文本文件创建路径为 /tmp/myfile 的流 */

mystream = br.openstream(pathname="/tmp/myfile" , mode= "z" )
```

注释包含以下缺陷：

- 该注释只是说明了代码中相当明显的部分。
- 该代码段未解释的不直觉的部分。即，`mode`参数是什么，`z`值是什么意思？

可重用

为了使读者轻松地重用示例代码，请提供以下内容：

- 运行示例代码所需的所有信息，包括所有依赖关系和设置。
- 可以以有用的方式扩展或自定义的代码。

拥有简洁明了且易于理解的示例代码是一个很好的开始。但是，如果毁了读者的应用程序，他们将不会高兴。因此，在编写示例代码时，请考虑由于将代码集成到另一个程序中而引起的任何潜在副作用。没有人想要不安全或效率很低的代码。

正反示例

除了告诉读者 *要做什么*，有时还要明智的向读者展示了*什么不该做*。例如，许多编程语言都允许程序员在等号的两侧放置空格。现在，假设你正在使用某种语言（例如 bash）编写教程，该语言不允许在等号的两边使用空格。在这种情况下，同时展示好例子和反例子将使读者受益。例如：

正确的示例

```
#有效的字符串分配。

s="The Rain in Maine."
```

错误的示例

```
#由于在字符串的两边都有空格，因此字符串分配无效

#等于符号。

s = "The Rain in Maine."
```

并列示列

一个好的示例代码集展示了**一系列的复杂度**。

完全不了解某种技术的读者通常渴望获得一些简单的示例来上手。示例代码集中的第一个也是最基本的示例通常称为 Hello World程序。掌握了基础知识之后，工程师们需要更复杂的程序。一组好的示例代码提供了一系列简单，适当和复杂的示例程序。

以下哪一项是一组好的函数示例，以向新手介绍编程函数概念的教程？

1. 以下是一组函数：
    1. 一个不带参数且不返回任何东西的函数。
    2. 一个带有一个参数但不返回任何东西的函数。
    3. 一个具有一个参数并返回一个值的函数。
    4. 具有三个参数并返回一个值的函数。
2. 以下是一组函数：
    1. 具有三个参数并返回一个值的函数。
3. 以下是一组函数：
    1. 一个具有一个参数并返回一个值的函数。
    2. 具有三个参数并返回一个值的函数。

最好的答案是**1**。提供涵盖一系列复杂性的样本通常是最明智的选择，尤其是对于新来者。抵制诱惑，*匆匆*走向非常复杂的示例程序，绕过新人渴望的初级和中级示例程序。

小结

技术写作两个涵盖了以下技术写作中级课程：

- 采用统一的样式。
- 换位思考。
- 大声朗读文档（对自己）。
- 编写初稿后，离开一会儿再回来查看文档。
- 寻找一个好的同伴编辑。
- 文档大纲。或是先写下类别，这后再组织起来。
- 介绍文档的范围和所有先决条件。
- 首选基于任务的标题。
- 循序渐进地写作（在某些情况下）。
- *在*创建插图*之前*，请考虑写标题。
- 将信息量限制在一个图示中。
- 通过标注来集中读者的注意力。
- 创建简单易懂的示例代码。
- 保持代码注释简短，但更推崇清楚而不是简洁。
- 避免编写*显而易见* 的代码注释。
- 将你的注释集中在任何非直觉的代码上。
- 不仅提供示例，还提供反例。
- 提供一系列复杂度的代码示例。
- 有连续修订记录的版本管理。

- 为不同类别的用户提供不同的文档类型。
- 与读者已经熟悉的东西进行比较和对比。
- 在教程中，通过示例来增强概念。
- 在教程中，指出危险之处。

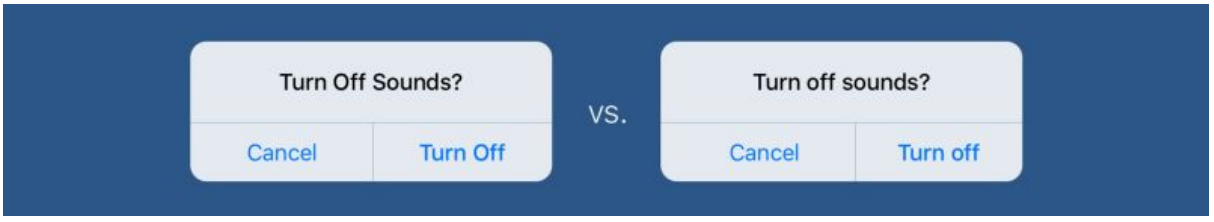
技术写作资源

这里总结了其他技术写作资源。

编辑风格指南

一个**编辑风格指南**规定了编辑团队的指导方针。例如，你的组织应采用以下哪个规则作为标题？

- 标题用 sentence case（仅将每个标题的首字母大写）
- 标题用 title case （将每个标题中的单词首字母大写）



（译者注：左边中是Title Case，右边是 Sentence case）

不要浪费时间和精力争论这类问题。而是要求你的组织采用统一的编辑风格指南。一个**编辑风格指南**提供了写作的一种约定。是 Sentence case 还是Title case 并不重要。整个团队采用统一的编辑风格指南才很重要。

嗯，但是要用哪个编辑风格指南呢？你可能已经熟悉通用的编辑样式指南（例如《芝加哥样式手册》或《 牛津大学样式指南》）。但是，你的工程团队应使用专门从事*技术*写作的编辑风格指南。所以，我们建议选择以下选项之一：

- 在谷歌的开发者文档风格指南规定了谷歌有关的项目，任何人书写的开发者文档的编辑准则。
- 《Microsoft写作风格指南》为编写技术文档的任何人提供了指南。

不要编写自己的编辑风格指南。创建和维护编辑风格指南需要大量资源，并且会引起巨大冲突。就是说，有团队创造了新术语，而现有的编辑风格指南中没有出现。发生这种情况时，组织可以执行以下任一操作：

- 要求编辑风格指南的维护者添加新术语。
- 创建并维护你自己的**用法指南**或**样式表**，以将团队的专业词汇的拼写和单词用法编入规章。

开源文档的机会

Google的"Season of Docs" 计划旨在促进开源项目与技术作家之间的协作。对于从事开源项目的人们来说，这是一个提高他们的技术写作技能的好机会。

文档季节每年运行一次。要随时了解2020年计划，你可以加入邮件列表或Slack工作区，如有关讨论频道的页面中所述。