第 14 章 选取名称

为变量、方法和其他实体选择名称是软件设计中最被低估的方面之一。好的名称是一种形式的文档：它们使代码更易于理解。它们减少了对其他文档的需求，并使检测错误更加容易。相反，名称选择不当会增加代码的复杂性，造成的歧义和误解可能会导致代码缺陷。名称选择是复杂性逐步累积的原因之一。为特定变量选择一个平庸的名称，而不是最好的名称，可能不会对系统的整体复杂性产生太大影响。但是，软件系统具有成千上万个变量，为所有这些选择好的名称将对复杂性和可管理性产生重大影响。

14.1 示例：坏名称会导致代码缺陷

有时，即使是一个取名不当的变量也会产生严重的后果。我曾经修复过的最具挑战性的代码缺陷就是由于名称选取不当造成的。在 1980 年代末和 1990 年代初，我的研究生和我创建了一个名为 Sprite 的分布式操作系统。在某个时候，我们注意到文件偶尔会丢失数据：即使用户未修改文件，数据块之一也会突然变为全零。该问题并不经常发生，因此很难追踪。一些研究生试图找到该错误，但他们未能取得进展，最终放弃了。但是，我认为任何未解决的代码缺陷都是无法忍受的，因此我决定对其进行跟踪。

结果花了六个月的时间，但我最终找到并修复了该缺陷。这个问题实际上很简单（就像大多数缺陷一样，一旦您找出原因之后）。文件系统代码将变量名 `block` 用于两个不同的目的。在某些情况下，`block` 是指磁盘上的物理块号；在其它情况下，`block` 是指文件中的逻辑块号。不幸的是，在代码的某处有一个包含逻辑块号的块变量，但是在某个需要物理块号的情况下意外地使用了它。结果，磁盘上无关的块被重置为零了。

在跟踪该错误时，包括我自己在内的几个人都阅读了有问题的代码，但我们从未注意到问题所在。当我们看到变量 `block` 用作物理块号时，我们本能地假设它确实拥有物理块号。经过很长时间的排查，最终表明损坏一定发生在特定的语句中，然后我才能越过该名称所创建的思维障碍，并检查它的值究竟来自何处。如果对不同类型的块使用不同的变量名（例如 `fileBlock` 和 `diskBlock`），则错误很可能不会发生；程序员会知道在哪种情况下不能使用 `fileBlock`。甚至更好的是给这两种不同的块定义不同的类型，这样它们就不可能互换。

不幸的是，大多数开发人员没有花太多时间在思考名称上面。他们倾向于使用想到的第一个名称，只要它的含义与被命名的事物合理相近即可。例如，块与磁盘上的物理块和文件内的逻辑块都非常接近，这肯定不是一个可怕的名称。即使如此，它还是导致花费了大量时间来追踪一个细微的错误。因此，您不应该只选择“合理相近”的名称。花一些额外的时间来选择准确、明确且直观的好名称。额外的时间花费将很快收回成本，随着时间的流逝，您将学会快速选择好名称。

14.2 创造画面

选择名称时，目标是能在读者的脑海中创造出关于被命名事物的本质的画面。一个好的名称传达了很多有关底层实体是什么以及（同样重要的）不是什么的信息。在考虑一个特定的名称时，请问自己：“如果有人孤立地看到这个名称，而没有看到其声明、文档或使用该名称的任何代码，他们是否能够猜到该名称指的是什么？还有其他名称可以让这个画面更清晰吗？”当然，一个名称可以包含多少信息是有限制的。如果名称包含两个或三个以上的单词，则会变得笨拙。因此，挑战是仅通过几个单词就能捕获到实体的最重要的方面。

名称是一种抽象形式：名称提供了一种简化的方式来思考更复杂的底层实体。与其他抽象形式一样，最好的名称会突出底层实体最重要的东西，而忽略那些次要的细节。

14.3 名称应该是精确的

良好的名称具有两个属性：精确性和一致性。让我们从精确性开始。名称最常见的问题是太笼统或含糊不清。因此，读者很难说出这个名称指的是什么。读者可能会认为该名称所指的是与现实不符的事物，如上面的 `block` 缺陷所示。考虑以下方法声明：

```
/**
 * Returns the total number of indexlets this object is managing.
 */
int IndexletManager::getCount() {...}
```

术语 `count` 太笼统了：对什么计数？如果有人看到此方法的调用，除非他们阅读了它的文档，否则他们不太可能知道它的作用。像 `numActiveIndexlets` 这样的更精确的名称会更好：很多读者可能无需查看其文档就能猜测该方法返回的内容。

 危险信号：模糊的名称 

> 如果变量或方法的名称足够广泛，可以指代许多不同的事物，那么它不会向开发人员传递太多信息，因此其底层的实体很可能会被误用。

以下是其他一些来自学生项目的名称不够精确的示例：

- 构建图像界面文本编辑器的项目使用名称 `x` 和 `y` 来表示字符在文件中的位置。这些名称太笼统了。他们可能意味着很多事情。例如，它们也可能代表屏幕上字符的坐标（以像素为单位）。单独看到名称 `x` 的人不太可能会认为它是指字符在一行文本中的位置。如果使用诸如 `charIndex` 和 `lineIndex` 之类的名称来反映代码实现的特定抽象，该代码将更加清晰。
- 另一个编辑器项目包含以下代码：

  ```
  // Blink state: true when cursor visible.
  private boolean blinkStatus = true;
  ```

`blinkStatus` 这个名称无法传达足够的信息。`status` 一词对于布尔值来说太含糊了：它不提供关于真值或假值含义的任何线索。`blink` 一词也含糊不清，因为它并没有将其含义表述清楚。以下是更好的选择：

```
// Controls cursor blinking: true means the cursor is visible,
// false means the cursor is not displayed.
private boolean cursorVisible = true;
```

名称 `cursorVisible` 传达了更多信息；例如，它允许读者猜测真值的含义（通常，布尔变量的名称应始终为谓词）。名称中也不再包含 `blink` 一词，因此，如果读者想知道为什么光标不总是可见，则必须查阅文档，此信息不那么重要。

- 一个实现共识协议的项目包含以下代码：

```
// Value representing that the server has not voted (yet) for
// anyone for the current election term.
private static final String VOTED_FOR_SENTINEL_VALUE = "null";
```

此值的名称表示它是特殊的，但没有说明特殊含义是什么。使用更具体的名称（例如 `NOT_YET_VOTED`）会更好。

- 在没有返回值的方法中使用了名为 `result` 的变量。这个名称有多个问题。首先，它会产生误导，让人以为它将成为方法的返回值。其次，它除了是某种计算值外，实际上没有提供关于持有内容的任何信息。它的名称应提供有关 `result` 实际是什么的信息，例如 `mergedLine` 或 `totalChars`。在实际上确实具有返回值的方法中，使用 `result` 名称是合理的。该名称仍然有点通用，但是读者可以查看方法的文档以了解其含义，这有助于知道什么值最终将成为返回值。
- Linux 内核包含两个描述网络套接字结构的结构：`struct socket` 和 `struct sock`。`struct sock` 包含一个 `struct socket` 作为其第一个元素，它实际上是 `struct socket` 的子类。这些名称如此相似，以至于很难记住哪个是哪个。选择易于区分的并阐明了这两个类型之间的关系的名称会更好，例如 `struct sock_base` 和 `struct inet_sock`。

像所有规则一样，有关选择精确名称的规则也有一些例外。例如，如果循环仅包含几行代码，也可以将通用名称（如 `i` 和 `j`）用作循环迭代变量。如果您可以直接看到一个变量的完整使用范围，那么该变量的含义可能在代码中就很明显了，因此您不需要长名称。例如以下代码：

```
for  (i = 0; i < numLines; i++) {
    ...
}
```

从这段代码中可以很明显地看到 `i` 正被用来迭代某个实体中的每一行。如果循环太长，以至于您无法一次看到全部内容，或者如果很难从代码中找出迭代变量的含义，那么应该使用更具描述性的名称。

名称也可能太具体，例如这个用来删除文本范围的方法的申明：

```
void delete(Range selection) {...}
```

`selection` 参数的名称过于具体，因为它暗示要删除的文本是当前在用户界面中选取的。但是，可以在任意范围的文本上调用此方法，无论是否选取。因此，这个参数名称应选取更通用的，例如 `range`。

如果您发现很难为特定变量想出一个精确、直观且不太长的名称，那么这是一个危险信号。这表明该变量可能没有清晰的定义或目的。发生这种情况时，请考虑其它因素。例如，也许您正在尝试使用单个变量来表示多个事物；如果是这样，将这种表示分成多个变量可能会让每个变量的定义更简单。选取好名称的过程可以通过识别弱点来改善您的设计。

 危险信号：难以选取名称 

> 如果很难为变量或方法找到一个简单的名称，该名称能够清晰地描述底层对象，那么这暗示底层对象的设计可能不够简洁。

14.4 命名要确保一致性

好的名称的第二个重要属性是一致性。在任何程序中，都会反复使用某些变量。例如，文件系统反复操作块号。对于每种常见用途，请选择一个用于该目的的名称，并在各处使用相同的名称。例如，文件系统可能总是使用 `fileBlock` 来保存文件中的块索引。一致的命名方式与复用一个通用的类一样，可以减轻认知负荷：一旦读者在一个上下文中看到了该名称，当他们在不同上下文中看到该名称时，就可以重用其知识并立即做出假设。

一致性具有三个要求：首先，始终将通用名称用于给定目的；其次，除了给定目的外，切勿使用通用名称；第三，确保给定的目的足够窄，以使所有具有该名称的变量都具有相同的行为。在本章开头的文件系统缺陷案例中违反了第三项要求。文件系统使用 `block` 来表示具有两种不同行为的变量（文件块和磁盘块），这导致对变量含义的错误假设，进而导致代码缺陷。

有时您将需要多个变量来引用相同的事物。例如，一个复制文件数据的方法将需要两个块号，一个为源，一个为目标。发生这种情况时，请对每个变量使用通用名称，但要添加一个可区分的前缀，例如 `srcFileBlock` 和 `dstFileBlock`。

循环是一致性命名可以提供帮助的另一个领域。如果将诸如 `i` 和 `j` 之类的名称用于循环变量，则始终在最外层循环中使用 `i`，而在嵌套的循环中始终使用 `j`。这使读者可以在看到给定名称时对代码中发生的事情做出即时的（安全的）假设。

14.5 避免多余的单词

名称中的每一个单词都应该提供有用的信息，没有帮助澄清变量含义的单词只会增加混乱（例如，它们可能会导致更多的行要换行）。一个常见的错误是向名称中添加通用的名词，例如 `field` 或 `object`，例如 `fileObject`。在这种情况下，单词 `Object` 可能不会提供有用的信息（还有不是对象的文件吗？），因此应该从名称中省略。

一些编码风格会在名称中包含类型信息，例如 `filePtr` 表示一个指向文件对象的指针变量。一个极端的例子是在微软的 C 编程中使用的匈牙利命名法。在匈牙利命名法中，每个变量名称都有一个前缀，表示其完整的类型。例如，名称 `arru8NumberList` 表示该变量是一个无符号 8 位整数的数组。尽管我过去也会在变量名称中包含类型信息，但不再推荐这样做。随着现代 IDE 的出现，很容易从变量名称跳转到其声明（或者 IDE 甚至可以自动显示类型信息），因此不需要在变量名称中包含此信息。

另一个多余单词的例子是当一个类的实例变量重复了类的名称时，例如在名为 File 的类中有一个名为 fileBlock 的实例变量。该变量是 File 类的组成部分在上下文中很明显，因此将类名包含在变量名称中没有提供任何有用的信息，只需将变量命名为 `block`（除非该类包含多个不同类型的块）。

14.6 不同的观点：Go 样式指南

并非所有人都同意我对命名的看法。一些使用 Go 语言的开发人员认为，名称应该非常简短，通常只能是一个字符。在关于 Go 的名称选择的演示中，Andrew Gerrand 指出“长名称模糊了代码的作用。”[1] 他介绍了此代码示例，该示例使用单字母变量名：

```
func RuneCount(b []byte) int {
    i, n := 0, 0
    for i < len(b) {
        if b[i] < RuneSelf {
            i++
        } else {
            _, size := DecodeRune(b[i:])
            i += size
        }
        n++
    }
    return n
}
```

并认为它比以下使用更长名称的版本更具可读性：

```
func RuneCount(buffer []byte) int {
    index, count := 0, 0
    for index < len(buffer) {
        if buffer[index] < RuneSelf {
            index++
        } else {
            _, size := DecodeRune(buffer[index:])
            index += size
        }
        count++
    }
    return count
}
```

就个人而言，我不觉得第二版比第一版更难读。比如，与 `n` 相比，名称 `count` 为变量的行为提供了更好的线索。在第一个版本中，我最终通读了代码，才弄清楚 `n` 的含义，而第二个版本中我并没有这种需要。但是，如果在整个系统中一致地使用 `n` 来表示计数（而不表示任何其它内容），那么其他开发人员可能会清楚知道该短名称。

Go 文化鼓励在多个不同的事物上使用相同的短名称：`ch` 用于字符或通道，`d` 用于数据、差异或距离，等等。对我来说，像这样的模棱两可的名称很可能导致混乱和错误，就像在文件块的示例中一样。

总的来说，我认为可读性必须由读者而不是作者来决定。如果您使用简短的变量名编写代码，并且阅读该代码的人很容易理解，那么很好。如果您开始抱怨代码很含糊，那么您应该考虑使用更长的名称（在网络上搜索 `go language short name` 会发现一些这样的抱怨）。同样，如果我开始抱怨长变量名使我的代码难以阅读，那么我会考虑使用较短的变量名。

Gerrand 发表了一个我同意的评论：“名称声明与其使用之间的距离越大，名称就应该越长。”前面关于使用名为 `i` 和 `j` 的循环变量的讨论是此规则的示例。

14.7 结论

精心选取的名称能大大提高代码的可读性。当有人第一次遇到该变量时，他们对行为的第一次猜测就是正确的，而不需要太多的思考。选取好名称是第 3 章讨论的投资思维的一个示例：如果您花一些额外的时间来选取好名称，将来您将更容易处理代码。此外，您引入代码缺陷的可能性更小。培养命名技巧也是一项投资。当您第一次决定不再满足于平庸的名称时，您会发现想出好名称的过程既令人沮丧又耗时。但是，随着您获得更多的经验，您会发现命名变得更加容易。最终，您将几乎不需要花费额外的时间来选取好名称，因此您几乎可以毫不费力地获得它的好处。

[1] https://talks.golang.org/2014/names.slide#1