

The Power of Variable Names

第 11 章

变量名的力量

cc2e.com/1184 内容

- 11.1 选择好变量名的注意事项：第 259 页
- 11.2 为特定类型的数据命名：第 264 页
- 11.3 命名规则的力量：第 270 页
- 11.4 非正式命名规则：第 272 页
- 11.5 标准前缀：第 279 页
- 11.6 创建具备可读性的短名字：第 282 页
- 11.7 应该避免的名字：第 285 页

相关章节

- 子程序名字：第 7.3 节
- 类的名字：第 6.2 节
- 使用变量的一般事项：第 10 章
- 格式化数据声明：第 31.5 节中的“数据声明的布局”
- 注释变量：第 32.5 节中的“注释数据声明”

尽管讨论如何为变量取好的名字与讨论如何高效编程同样重要，我却还没看到任何资料能将创建好的名字的诸多注意事项涵盖一二。很多编程教科书只用几段的篇幅讲讲如何选择缩写，几句老生常谈，指望你自己解决这个问题。而我却要反其道而行之：就如何取一个好的名字给你大量的信息，多得你可能根本都不会用到！

本章所述原则主要适用于为变量——对象和基本数据——命名。不过它们也适用于为类、包、文件以及其他编程实体命名。有关如何为子程序命名请参阅第 7.3 节“好的子程序名字”。

Considerations in Choosing Good Names 选择好变量名的注意事项

你可不能像给狗取名字那样给变量命名——仅仅因为它很可爱或者听上去不错。狗和狗的名字不一样，它们是不同的东西，变量和变量名就本质而言却是同一事物。这样一来，变量的好与坏就在很大程度上取决于它的命名的好坏。在给变量命名的时候需要小心谨慎。

下面举一个使用了不良变量名的例子：



Java示例：糟糕的变量名

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

这段代码究竟在做什么？`x1`、`xx` 和 `xxx` 代表什么？`fido` 又是什么意思？假如说有人告诉你这段代码基于一项余额和一组新开销来计算一位顾客的支付总额，那么你应该使用哪个变量来为该顾客的那组新的花销打印账单呢？

下面是这些代码的另一种写法，它可以使上述问题回答起来非常容易：

Java示例：良好的变量名

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) + monthlyTotal;
balance = balance + Interest( customerID, balance );
```

从上述两段代码的比较中可以看出，一个好的变量名是可读的、易记的和恰如其分的。你可以通过应用多条原则来实现这些目标。

The Most Important Naming Consideration 最重要的命名注意事项



KEY POINT

为变量命名时最重要的考虑事项是，该名字要完全、准确地描述出该变量所代表的事物。获得好名字的一种实用技巧就是用文字表达变量所代表的是什么。通常，对变量的描述就是最佳的变量名。这种名字很容易阅读，因为其中并不包含晦涩的缩写，同时也没有歧义。因为它是对该事物的完整描述，因此不会和其他事物混淆。另外，由于这一名字与所表达的概念相似，因此也很容易记忆。

对于一个表示美国奥林匹克代表团成员数量的变量，你可能会把它命名为 `numberOfPeopleOnTheUsOlympicTeam`。表示运动场中坐椅数量的变量可能会命名为 `numberOfSeatsInTheStadium`。表示某国代表团在现代奥运会上获得的最高分数的变量可能会命名为 `maximumNumberOfPointsInModernOlympics`。表示当前利率的变量最好命名为 `rate` 或 `interestRate`，而不是 `r` 或 `x`。你明白了吧。

请留意上述这些命名所共有的两个特征。首先，它们都很容易理解。事实上它们根本不需要什么解释，因为你可以很轻松地读懂它们。不过第二点，有些名字太长了——长得很不实用。下面我很快就会讲到变量名的长度问题。

表 11-1 中给出了更多变量名称的例子，其中有好的也有差的。

表 11-1 更多变量名的例子，其中有好的也有差的

变量用途	好名字，好描述	坏名字，差描述
到期的支票 累计额	runningTotal, checkTotal	written, ct, checks, CHKTTL, x, x1, x2
高速列车的 运行速度	velocity, trainVelocity, velocityInMph	velt, v, tv, x, x1, x2, train
当前日期	currentDate, todaysDate	cd, current, c, x, x1, x2, date
每页的行数	linesPerPage	lpp, lines, l, x, x1, x2

currentDate 和 todaysDate 都是很好的名字，因为它们都完全而且准确地描述出了“当前日期”这一概念。事实上，这两个名字都用了非常直白的词。程序员们有时候会忽视这些普通词语，而它们往往却是最明确的。cd 和 c 是很糟的命名，因为它们太短，同时又不具有描述性。current 也很糟，因为它并没有告诉你这是当前什么。date 看上去不错，但经过最后推敲它也只是个坏名字，因为这里所说的日期并不是所有的日期均可，而只是特指当前日期；而 date 本身并未表达出这层含义。x、x1 和 x2 永远是坏名字——传统上用 x 代表一个未知量；如果不希望你的变量所代表的是一个未知量，那么请考虑取一个更好的名字吧。



KEY POINT

名字应该尽可能地明确。像 x、temp、i 这些名字都泛泛得可以用于多种目的，它们并没有像应该的那样提供足够信息，因此通常都是命名上的败笔。

Problem Orientation

以问题为导向

一个好记的名字反映的通常都是问题，而不是解决方案。一个好名字通常表达的是“什么”(what)，而不是“如何”(how)。一般而言，如果一个名字反映了计算的某些方面而不是问题本身，那么它反映的就是“how”而非“what”了。请避免选取这样的名字，而应该在名字中反映出问题本身。

一条员工数据记录可以称作 inputRec 或者 employeeData。inputRec 是一个反映输入、记录这些计算概念的计算机术语。employeeData 则直指问题领域，与计算的世界无关。与此类似，对一个用于表示打印机状态的位域来说，bitFlag 就要比 printerReady 更具计算机特征。在财务软件里，calcVal 的计算痕迹也要比 sum 更明显。

Optimum Name Length 最合适的名字长度

变量名的最佳长度似乎应该介于 `x` 和 `maximumNumberOfPointsInModernOlympics` 之间。太短的名字无法传达足够的信息。诸如 `x1` 和 `x2` 这样的名字所存在的问题是，即使你知道了 `x` 代表什么，你也无法获知 `x1` 和 `x2` 之间的关系。太长的名字很难写，同时也会使程序的视觉结构变得模糊不清。



HARD DATA

Gorla、Benander 和 Benander 发现，当变量名的平均长度在 10 到 16 个字符的时候，调试程序所需花费的气力是最小的（1990）。平均名字长度在 8 到 20 个字符的程序也几乎同样容易调试。这项原则并不意味着你应该尽量把变量名控制在 9 到 15 或者 10 到 16 个字符长。它强调的是，如果你查看自己写的代码时发现了很多更短的名字，那么你需要认真检查，确保这些名字含义足够清晰。

你可能已经通过 Goldilocks-and-the-Three-Bears（金发姑娘与三只小熊的经典通话，寓意权衡比较）的方法理解了如何为变量命名，正如表 11-2 所示。

表 11-2 变量名太长、太短或刚好合适的示例

太长:	<code>numberOfPeopleOnTheUsOlympicTeam</code> <code>numberOfSeatsInTheStadium</code> <code>maximumNumberOfPointsInModernOlympics</code>
太短:	<code>n, np, ntm</code> <code>n, ms, nsisd</code> <code>m, mp, max, points</code>
正好:	<code>numTeamMembers, teamMemberCount</code> <code>numSeatsInStadium, seatCount</code> <code>teamPointsMax, pointsRecord</code>

The Effect of Scope on Variable Names 变量名对作用域的影响

交叉参考 关于作用域的详细讨论，见第10.4节“作用域”。

短的变量名总是不好吗？不，不总是这样。当你把一个变量名取得很短的时候，如 `i`，这一长度本身就对该变量做出了一些说明——也就是说，该变量代表的是一个临时的数据，它的作用域非常有限。

阅读该变量的程序员应该会明白，这一数值只会用于几行代码之内。当你把变量命名为 `i` 的时候，你就是在表示，“这是一个普通的循环计数器或者数组下标，在这几行代码之外它没任何作用。”

W.J.Hansen 所做的一项研究表明，较长的名字适用于很少用到的变量或者全局变量，而较短的名字则适用于局部变量或者循环变量（Shneiderman 1980）。不过，短的变量名常常会带来一些麻烦，因此，作为一项防御式编程策略，一些细

心的程序员会避免使用短的变量名。

对位于全局命名空间中的名字加以限定词 如果你在全局命名空间中定义了一些变量（具名常量、类名等），那么请考虑你是否需要采用某种方式对全局命名空间进行划分，并避免产生命名冲突。在 C++ 和 C# 里，你可以使用 `namespace` 关键字来划分全局命名空间。

C++示例：使用namespace关键字来划分全局命名空间

```
namespace UserInterfaceSubsystem {  
    ...  
    // lots of declarations  
    ...  
}  
  
namespace DatabaseSubsystem {  
    ...  
    // lots of declarations  
    ...  
}
```

如果你同时在 `UserInterfaceSubsystem` 和 `DatabaseSubsystem` 命名空间里声明了 `Employee` 类，那么你可以通过写 `UserInterfaceSubsystem::Employee` 或者 `DatabaseSubsystem::Employee` 来确定引用哪一个 `Employee`。在 Java 中，你也可以通过使用包（package）来达到同样的目的。

在那些不支持命名空间或者包的语言里，你同样也可以使用命名规则来划分全局命名空间。其中一项规则要求为全局可见的类加上带有子系统特征的前缀。用户接口部分的雇员类可能命名为 `uiEmployee`，数据库部分的雇员类可能命名为 `dbEmployee`，这样做能把全局命名空间的命名冲突降到最低。

Computed-Value Qualifiers in Variable Names 变量名中的计算值限定词

很多程序都有表示计算结果的变量：总额、平均值、最大值，等等。如果你要用类似于 `Total`、`Sum`、`Average`、`Max`、`Min`、`Record`、`String`、`Pointer` 这样的限定词来修改某个名字，那么请记住把限定词加到名字的最后。

这种方法具有很多优点。首先，变量名中最重要的那部分，即为这一变量赋予主要含义的部分应当位于最前面，这样，这一部分就可以显得最为突出，并会被首先阅读到。其次，采纳了这一规则，你将避免由于同时在程序中使用 `totalRevenue` 和 `revenueTotal` 而产生的歧义。这些名字在语义上是等价的，上述规则可以避免将它们当作不同的东西使用。还有，类似 `revenueTotal`（总收入）、`expenseTotal`（总支出）、`revenueAverage`（评价收入）、`expenseAverage`（平均支出）这组名字的变量具有非常优雅的对称性。而从 `totalRevenue`、`expenseTotal`、`revenueAverage`、`averageExpense` 这组名字中则看不出什么规

律来。总之，一致性可以提高可读性，简化维护工作。

把计算的量放在名字最后的这条规则也有例外，那就是 Num 限定词的位置已经是约定俗成的。Num 放在变量名的开始位置代表一个总数：numCustomers 表示的是员工的总数。Num 放在变量名的结束位置代表一个下标：customerNum 表示的是当前员工的序号。通过 numCustomers 最后代表复数的 s 也能够看出这两种应用之间的区别。然而，由于这样使用 Num 常常会带来麻烦，因此可能最好的办法是避开这些问题，用 Count 或者 Total 来代表员工的总数，用 Index 来指代某个特定的员工。这样，customerCount 就代表员工的总数，customerIndex 代表某个特定的员工。

Common Opposites in Variable Names 变量名中的常用对仗词

交叉参考 用于子程序名的类似对仗词的清单，见第7.3节中的“准确使用对仗词”。

对仗词的使用要准确。通过应用命名规则来提高对仗词使用的一致性，从而提高其可读性。比如像 begin/end 这样的一组用词非常容易理解和记忆。而那些与常用语言相去甚远的词则通常很难记忆，有时甚至会产生歧义。下面是一些常用的对仗词：

- begin/end
- first/last
- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

11.2 Naming Specific Types of Data 为特定类型的数据命名

在为数据命名的时候，除了通常的考虑事项之外，为一些特定类型数据的命名还要求做出一些特殊的考虑。本节将讲述与循环变量、状态变量、临时变量、布尔变量、枚举类型和具名常量有关的考虑事项。

Naming Loop Indexes 为循环下标命名

交叉参考 关于循环的详细讨论，见第16章“控制循环”。

循环是一种极为常见的计算机编程特征，为循环中的变量进行命名的原则也由此应运而生。*i*、*j* 和 *k* 这些名字都是约定俗成的：

Java示例：简单的循环变量名

```
for ( i = firstItem; i < lastItem; i++ ) {  
    data[ i ] = 0;  
}
```

如果一个变量要在循环之外使用，那么就应该为它取一个比 *i*、*j* 或者 *k* 更有意义的名字。举个例子，如果你在从文件中读取记录，并且需要记下所读取记录的数量，那么类似于 *recordCount* 这样的名字就很合适：

Java示例：描述性较好的循环变量名

```
recordCount = 0;  
while ( moreScores() ) {  
    score[ recordCount ] = GetNextScore();  
    recordCount++;  
}  
  
// lines using recordCount  
...
```

如果循环不是只有几行，那么读者会很容易忘记 *i* 本来具有的含义，因此你最好给循环下标换一个更有意义的名字。由于代码会经常修改、扩充，或者复制到其他程序中去，因此，很多有经验的程序员索性不使用类似于 *i* 这样的名字。

导致循环变长的常见原因之一是出现循环的嵌套使用。如果你使用了多个嵌套的循环，那么就应该给循环变量赋予更长的名字以提高可读性：

Java示例：嵌套循环中的好循环变量名

```
for ( teamIndex = 0; teamIndex < teamCount; teamIndex++ ) {  
    for ( eventIndex = 0; eventIndex < eventCount[teamIndex]; eventIndex++ ) {  
        score[ teamIndex ][ eventIndex ] = 0;  
    }  
}
```

谨慎地为循环下标变量命名可以避免产生下标串话(index cross-talk)的常见问题：想用 *j* 的时候写了 *i*，想用 *i* 的时候却写了 *j*。同时，这也使得数据访问变得更加清晰：*score[teamIndex][eventIndex]* 要比 *score[i][j]* 给出的信息更多。

如果你一定要用 *i*、*j* 和 *k*，那么不要把它们用于简单循环的循环下标之外的任何场合——这种传统已经太深入人心了，一旦违背该原则，将这些变量用于其他用途就可能造成误解。要想避免出现这样的问题，最简单的方法就是想出一个比 *i*、*j* 和 *k* 更具描述性的名字来。

Naming Status Variables 为状态变量命名

状态变量用于描述你的程序的状态。下面给出它的命名原则。

为状态变量取一个比 flag 更好的名字 最好是把标记（flag）看做状态变量。标记的名字中不应该含有 flag，因为你从中丝毫看不出该标记是做什么的。为了清楚起见，标记应该用枚举类型、具名常量，或用作具名常量的全局变量来对其赋值，而且其值应该与上面这些量做比较。下面例子中标记的命名都很差：



C++示例：含义模糊的标记

```
if ( flag ) ...
if ( statusFlag & 0x0F ) ...
if ( printFlag == 16 ) ...
if ( computeFlag == 0 ) ...

flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

像 statusFlag = 0x80 这样的语句是反映不出这段代码能做什么的，除非你亲自写了这段代码，或者有文档能告诉你 statusFlag 和 0x80 的含义。下面是作用相同但更为清晰的代码：

C++示例：更好地使用状态变量

```
if ( dataReady ) ...
if ( characterType & PRINTABLE_CHAR ) ...
if ( reportType == ReportType_Annual ) ...
if ( recalcNeeded == false) ...

dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

显然，characterType = CONTROL_CHARACTER 要比 statusFlag = 0x80 更有意义。与之类似，条件判断语句 if (reportType == ReportType_Annual) 要比 if (printFlag == 16) 更为清晰。第二个例子说明你可以结合枚举类型和预定义的具名常量来使用这种方法。下面例子展示了如何使用具名常量和枚举类型来组织例子中的数值：

在C++中声明状态变量

```
// values for CharacterType
const int LETTER = 0x01;
const int DIGIT = 0x02;
const int PUNCTUATION = 0x04;
const int LINE_DRAW = 0x08;
```

```

const int PRINTABLE_CHAR = ( LETTER | DIGIT | PUNCTUATION | LINE_DRAW );
const int CONTROL_CHARACTER = 0x80;

// values for ReportType
enum ReportType {
    ReportType_Daily,
    ReportType_Monthly,
    ReportType_Quarterly,
    ReportType_Annual,
    ReportType_All
};

```

如果你发现自己需要猜测某段代码的含义的时候，就该考虑为变量重新命名。猜测谋杀案中谁是神秘凶手是可行的，但你没有必要去猜测代码。你应该能直接读懂它们。

Naming Temporary Variables 为临时变量命名

临时变量用于存储计算的中间结果，作为临时占位符，以及存储内务管理（housekeeping）值。它们常被赋予 `temp`、`x` 或者其他一些模糊且缺乏描述性的名字。通常，临时变量是一个信号，表明程序员还没有完全把问题弄清楚。而且，由于这些变量被正式地赋予了一种“临时”状态，因此程序员会倾向于比其他变量更为随意地对待这些变量，从而增加了出错的可能。

警惕“临时”变量 临时性地保存一些值常常是很有必要的。但是无论从哪种角度看，你程序中的大多数变量都是临时性的。把其中几个称为临时的，可能表明你还没有弄清它们的实际用途。请考虑下面的示例：

C++示例：不提供信息的“临时”变量名

```

// Compute roots of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
temp = sqrt( b^2 - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );

```

把表达式 `sqrt(b^2 - 4 * a * c)` 的结果存储在一个变量里是很不错的，特别是当这一结果还会被随后两次用到的时候。但是名字 `temp` 却丝毫也没有反映该变量的功能。下面例子显示了一种更好的做法：

C++示例：用真正的变量替代“临时”变量

```

// Compute roots of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
discriminant = sqrt( b^2 - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );

```

就本质而言，这段代码与上面一段是完全相同的，但是它却通过使用了准确而且具有描述性的变量名（discriminant，判别式）而得到了改善。

Naming Boolean Variables 为布尔变量命名

下面是为布尔变量命名时要遵循的几条原则。

谨记典型的布尔变量名 下面是一些格外有用的布尔变量名。

- *done* 用 *done* 表示某件事情已经完成。这一变量可用于表示循环结束或者一些其他的操作已完成。在事情完成之前把 *done* 设为 *false*，在事情完成之后把它设为 *true*。
- *error* 用 *error* 表示有错误发生。在错误发生之前把变量值设为 *false*，在错误已经发生时把它设为 *true*。
- *found* 用 *found* 来表明某个值已经找到了。在还没有找到该值的时候把 *found* 设为 *false*，一旦找到该值就把 *found* 设为 *true*。在一个数组中查找某个值，在文件中搜寻某员工的 ID，在一沓支票中寻找某张特定金额的支票等等的时候，都可以用 *found*。
- *success* 或 *ok* 用 *success* 或 *ok* 来表明一项操作是否成功。在操作失败的时候把该变量设为 *false*，在操作成功的时候将其设为 *true*。如果可以，请用一个更具体的名字代替 *success*，以便更具体地描述成功的含义。如果完成处理就表示这个程序执行成功，那么或许你应该用 *processingComplete* 来取而代之。如果找到某个值就是程序执行成功，那么你也许应该换用 *found*。

给布尔变量赋予隐含“真/假”含义的名字 像 *done* 和 *success* 这样的名字是很不错的布尔变量名，因为其状态要么是 *true*，要么是 *false*；某件事情完成了或者没有完成；成功或者失败。另一方面，像 *status* 和 *sourceFile* 这样的名字却是很糟的布尔变量名，因为它们没有明确的 *true* 或者 *false*。*status* 是 *true* 反映的是什么含义？它表明某件事情拥有一个状态吗？每件事都有状态。*true* 表明某件事情的状态是 OK 吗？或者说 *false* 表明没有任何错误吗？对于 *status* 这样的名字，你什么也说不出来。

为了取得更好的效果，应该把 *status* 替换为类似于 *error* 或者 *statusOK* 这样的名字，同时把 *sourceFile* 替换为 *sourceFileAvailable*、*sourceFileFound*，或者其他能体现该变量所代表含义的名字。

有些程序员喜欢在他们写的布尔变量名前加上 *Is*。这样，变量名就变成了一个问题：*isdone?* *isError?* *isFound?* *isProcessingComplete?* 用 *true* 或 *false* 回答问题也就为该变量给出了取值。这种方法的优点之一是它不能用于那些模糊不清的名字：*isStatus?* 这毫无意义。它的缺点之一是降低了简单逻辑表达式的可读性：*if (isFound)* 的可读性要略差于 *if (found)*。

使用肯定的布尔变量名 否定的名字如 notFound、notdone 以及 notSuccessful 等较难阅读，特别是如果它们被求反：

```
if not notFound
```

这样的名字应该替换为 found、done 或者 processingComplete，然后再用适当的运算符求反。如果你找到了想找的结果，那么就可以用 found 而不必写双重否定的 not notFound 了。

Naming Enumerated Types 为枚举类型命名

交叉参考 关于使用枚举类型的详情，见第12.6节“枚举类型”。

在使用枚举类型的时候，可以通过使用组前缀，如 color_，planet_ 或者 month_ 来明确表示该类型的成员都同属于一个组。下面举一些通过前缀来确定枚举类型元素的例子：

Visual Basic示例：为枚举类型采用前缀命名约定

```
Public Enum Color
    Color_Red
    Color_Green
    Color_Blue
End Enum

Public Enum Planet
    Planet_Earth
    Planet_Mars
    Planet_Venus
End Enum

Public Enum Month
    Month_January
    Month_February
    ...
    Month_December
End Enum
```

与此同时，也有很多命名方法可用于确定枚举类型本身的名字（Color，Planet 或 Month），包括全部大写或者加以前缀（e_Color，e_Planet，e_Month）。有人可能会说，枚举从本质上而言是一个用户定义类型，因此枚举名字的格式应该与其他用户定义的类型如类等相同。与之相反的一种观点认为枚举是一种类型，但它同时也是一种常量，因此枚举类型名字的格式应该与常量相同。本书对枚举类型采用了大小写混合的命名方式。

在有些编程语言里，枚举类型的处理很像类，枚举成员也总是被冠以枚举名字前缀，比如 Color.Color_Red 或者 Planet.Planet_Earth。如果你正在使用这样的编程语言，那么重复上述前缀的意义就不大了，因此你可以把枚举类型自身的名字作为前缀，并把上述名字简化为 Color.Red 和 Planet.Earth。

Naming Constants

为常量命名

交叉参考 关于使用具名常量的详情，见第 12.7 节“具名常量”。在具名常量时，应该根据该常量所表示的含义，而不是该常量所具有的数值为该抽象事物命名。`FIVE` 是个很糟的常量名（不论它所代表的值是否为 5.0）。`CYCLES_NEEDED` 是个不错的名字。`CYCLES_NEEDED` 可以等于 5.0 或者 6.0。而 `FIVE = 6.0` 就显得太可笑了。出于同样原因，`BAKERS_DOZEN` 就是个很糟的常量名；而 `DONUTS_MAX` 则很不错。

11.3 The Power of Naming Conventions 命名规则的力量

有些程序员会抵制标准和约定（convention，规则）——并且有很好的理由：有些标准和约定非常刻板并且低效——它们会毁坏创造性和程序质量。这真让人感到遗憾，因为有效的标准是你所能掌握的最强大的工具之一。本节将讲述为什么、何时以及如何创建自己的变量命名标准。

Why Have Conventions

为什么要有规则

命名规则可以带来以下的好处。

- 要求你更多地按规矩行事。通过做一项全局决策而不是做许多局部决策，你可以集中精力关注代码更重要的特征。
- 有助于在项目之间传递知识。名字的相似性能让你更容易、更自信地理解那些不熟悉的变量原本应该是做什么的。
- 有助于你在新项目中更快速地学习代码。你无须了解 Anita 写的代码是这样的，Julia 是那样的，以及 Kristin 的代码又是另一种样子，而只须面对一组更加一致的代码。
- 有助于减少名字增生（name proliferation）。在没有命名规则的情况下，会很容易地给同一个对象起两个不同的名字。例如，你可能会把总点数既称为 `pointTotal`，也称为 `totalPoints`。在写代码的时候这可能并不会让你感到迷惑，但是它却会让一位日后阅读这段代码的新程序员感到极其困惑。
- 弥补编程语言的不足之处。你可以用规则来仿效具名常量和枚举类型。规则可以根据局部数据、类数据以及全局数据的不同而有所差别，并且可以包含编译器不直接提供的类型信息。

- 强调相关变量之间的关系。如果你使用对象，则编译器会自动照料它们。如果你用的编程语言不支持对象，你可以用命名规则来予以补充。诸如 address、phone 以及 name 这样的名字并不能表明这些变量是否相关。但是假设你决定所有的员工数据变量都应该以 Employee 作为前缀，则 employeeAddress、employeePhone 和 employeeName 就会毫无疑问地表明这些变量是彼此相关的。编程的命名规则可以对你所用的编程语言的不足之处做出弥补。



KEY POINT

关键之处在于，采用任何一项规则都要好于没有规则。规则可能是武断的。命名规则的威力并非来源于你所采取的某个特定规则，而是来源于以下事实：规则的存在为你的代码增加了结构，减少了你需要考虑的事情。

When You Should Have a Naming Convention 何时采用命名规则

没有金科玉律表明何时应该建立命名规则，但是在下列情况下规则是很有价值的。

- 当多个程序员合作开发一个项目时
- 当你计划把一个程序转交给另一位程序员来修改和维护的时候（这几乎总是会发生）
- 当你所在组织中的其他程序员评估你写的程序的时候
- 当你写的程序规模太大，以致于你无法在脑海里同时了解事情的全貌，而必须分而治之的时候
- 当你写的程序生命期足够长，长到你可能会在把它搁置几个星期或几个月之后又重新启动有关该程序的工作时
- 当在一个项目中存在一些不常见的术语，并且你希望在编写代码阶段使用标准的术语或者缩写的时候

你一定会因使用了某种命名规则而受益。上述诸多注意事项将会帮助你决定在一个特定项目中按照何种程度来制定规则里所使用的规则的范围。

Degrees of Formality 正式程度

交叉参考 关于小型项目和大型项目的在正式程度上的区别，见第 27 章“[程序规模对构建的影响](#)”。

不同规则所要求的正式程度也有所不同。一个非正式的规则可能会像“使用有意义的名字”这样简单。下一节将会讲述其他的非正式规则。通常，你所需的正式程度取决于为同一程序而工作的人员数量、程序的规模，以及程序预期的生命周期。对于微小的、用完即弃的项目而言，实施严格的规则可能就太没有必要了。对于多人协作的大型项目而言，无论是在开始阶段还是贯穿整个程序的生命周期，正式规则都是成为提高可读性的必不可少的辅助手段。

11.4 Informal Naming Conventions 非正式命名规则

大多数项目采用的都是类似于本节所讲的相对非正式的命名规则。

Guidelines for a Language-Independent Convention 与语言无关的命名规则的指导原则

下面给出用于创建一种与语言无关的命名规则的指导原则。

区分变量名和子程序名字 本书所采用的命名规则要求变量名和对象名以小写字母开始，子程序名字以大写字母开始：variableName 对 RoutineName()。

区分类和对象 类名字与对象名字——或者类型与该类型的变量——之间的关系会比较棘手。有很多标准的方案可用，如下例所示：

方案1：通过大写字母开头区分类型和变量

```
Widget widget;  
LongerWidget longerWidget;
```

方案2：通过全部大写区分类型和变量

```
WIDGET widget;  
LONGERWIDGET longerWidget
```

方案3：通过给类型加“t_”前缀区分类型和变量

```
t_Widget Widget;  
t_LongerWidget LongerWidget;
```

方案4：通过给变量加“a”前缀区分类型和变量

```
Widget aWidget;  
LongerWidget aLongerWidget;
```

方案5：通过对变量采用更明确的名字区分类型和变量

```
Widget employeeWidget;  
LongerWidget fullEmployeeWidget;
```

每一种方案都有其优点和不足。第一种方案是在大小写敏感语言如 C++ 和 Java 里常用的规则，但是有些程序员对仅依靠大写区分名字感到不大舒服。的确，创建两个只有第一个字母大小写不同的名字所能提供的“心理距离”太短了，二者之间的视觉差异也太小。

在多语言混合编程的环境中，如果任一种语言是大小写不敏感的，则将无法一直使用第一种命名方案。以 Microsoft Visual Basic 为例，Dim widget as Widget 将会引发一处语法错误，因为 widget 和 Widget 会被当做同一个标识符看待。

第二种方案使类型名和变量名之间的差异更加鲜明。然而，由于历史原因，在 C++ 和 Java 里面全部字母大写只用于表示常量，同时这种方案也会与第一种方案一样面临混合语言环境的问题。

第三种方案可用于所有语言，但是很多程序员从审美的角度出发并不喜欢增加前缀。

第四种方案有时会用作第三种方案的备选项，但是它存在的问题是需要改变类的每个实例的名字，而不是仅仅修改类名。

第五种方案要求基于每个变量的实际情况做出更多的考虑。在大多数情况下，要求程序员为每个变量想出一个特别的名字会有助于提高代码的可读性。但是有时候，一个 widget 确实就是一个普通的 widget，在这种情况下你会发现自己会想出一些并不鲜明的名字，如 genericWidget，它的可读性比较差。

简而言之，每一种可选方案都不是十全十美的。本书代码采用的是第五种方案，因为当不要求代码的阅读者熟悉一种不太直观的命名规则时，这种规则做是最容易理解的。

标识全局变量 有一种编程问题很常见，那就是滥用全局变量。假如你在所有的全局变量名之前加上 g_ 前缀，那么程序员在读到变量 g_RunningTotal 之后就会明白这是个全局变量，并且予以相应回应。

标识成员变量 要根据名字识别出变量是类的数据成员。即明确表示该变量既不是局部变量，也不是全局变量。比如说，你可以用 m_ 前缀来标识类的成员变量，以表明它是成员数据。

标识类型声明 为类型建立命名规则有两个好处：首先它能够明确表明一个名字是类型名，其次能够避免类型名与变量名冲突。为了满足这些要求，增加前缀或者后缀是不错的方法。C++ 的惯用方法是把类型名全部大写——例如 COLOR 和 MENU。（这一规则适用于 typedef 和 struct，不适用于类名。）但是这样就会增加与命名预处理常量发生混淆的可能。为了避免出现这样的麻烦，你可以为类型名增加 t_ 前缀，如 t_Color 和 t_Menu。

标识具名常量 你需要对具名常量加以标识，以便明确在为一个变量赋值时你用的是另一个变量的值（该值可能变化），还是一个具名常量。在 Visual Basic 里，还会有另外的可能，那就是该值可能是一个函数的返回值。Visual Basic 不要求在调用函数时给函数名加括号，与之相反，在 C++ 里即使函数没有参数也要使用括号。

给常量命名的方法之一是给常量名增加 `c_` 前缀。这会让你写出类似 `c_RecesMax` 或者 `c_LinesPerPageMax` 这样的名字来。`C++` 和 `Java` 里的规则是全部用大写，以及如果有可能，用下画线来分隔单词，例如 `RECSMAX` 或者 `RECS_MAX`，以及 `LINESPERPAGEMAX` 或者 `LINES_PER_PAGE_MAX`。

标识枚举类型的元素 与具名常量相同，枚举类型的元素也需要加以标识——以便表明该名字表示的是枚举类型，而不是一个变量、具名常量或者函数。标准方法如下：全部用大写，或者为类型名增加 `e_` 或 `E_` 前缀，同时为该类型的成员名增加基于特定类型的前缀，如 `Color_` 或者 `Planet_`。

在不能保证输入参数只读的语言里标识只读参数 有时输入参数会被意外修改。在 `C++` 和 `Visual Basic` 这样的语言里，你必须明确表明是否希望把一个修改后的值返回给调用方子程序。在 `C++` 里分别用 `*`、`&` 和 `const` 指明，在 `Visual Basic` 里分别用 `ByRef` 和 `ByVal` 指明。

交叉参考 可以通过命名方面的约定来增强某种语言的功能，从而弥补该语言自身的缺陷。这种方式正是“深入一种语言去编程”而非仅仅“在一种语言上编程”的典范。第 34.4 节“深入一门语言去编程，不浮于表面”有对这一主题的详细论述。

在其他的语言里，如果你修改了输入变量的取值，那么无论你是否愿意，它的新值都会被返回。特别是当你传递对象的时候。举例来说，在 `Java` 里所有对象都是“按值（by value）”传递的，因此当你把一个对象传递给一个子程序的时候，该对象的内容就可以在被调用子程序中修改（Arnold, Gosling, Holmes 2000）。

在这些语言里，如果你制定了为输入参数增加一个 `const` 前缀（或者 `final`、`nonmodifiable` 等）的命名规则，那么当你看到 `const` 前缀出现在赋值符号左边的时候，就会知道出现了错误。如果你看到 `constMax.SetNewMax(...)`，就会知道这里有大漏洞，因为 `const` 前缀表明了该变量是不应该被修改的。

格式化命名以提高可读性 有两种常用方法可以用来提高可读性，那就是用大小写和分隔符来分隔单词。例如，`GYMNASTICSPOINTTOTAL` 就要比 `gymnastics-PointTotal` 或者 `gymnastics_point_total` 难读得多。`C++`、`Java`、`Visual Basic` 和其他的编程语言允许混合使用大小写字符。另外，`C++`、`Java`、`Visual Basic` 和其他的编程语言也允许使用下画线（`_`）作为分隔符。

尽量不要混用上述方法，那样会使代码难以阅读。如果你老老实实地坚持使用其中任意一种提高可读性的方法，你的代码质量一定会有所改善。人们曾经就诸如变量名的第一个字母是不是应该大写（`TotalPoints` 对 `totalPoints`）的做法的价值展开了非常激烈的讨论，但是只要你和你的团队在使用上保持一致，那么大写小写就没有太大区别。基于 `Java` 经验的影响，同时为了促进不同编程语言之间命名风格的融合，本书对首字母采用小写。

Guidelines for Language-Specific Conventions 与语言相关的命名规则的指导原则

应该遵循你所用语言的命名规则。对于大多数语言，你都可以找到描述其风格原则的参考书。下面将给出 C、C++、Java 和 Visual Basic 的指导原则。

C Conventions

C 的命名规则

深入阅读 描述 C 语言编程风格的经典读物是《C Programming Guidelines》(Plum 1984)。

有很多命名规则特别适用于 C 语言。

- c 和 ch 是字符变量。
- i 和 j 是整数下标。
- n 表示某物的数量。
- p 是指针。
- s 是字符串。
- 预处理宏全部大写 (ALL_CAPS)。这通常也包括 typedef。
- 变量名和子程序名全部小写 (all_lowercase)。
- 下画线 (_) 用做分隔符: letters_in_lowercase 要比 lettersinlowercase 更具可读性。

这些都是属于一般性的、UNIX 风格或者 Linux 风格的 C 编程规则, C 编程规则在不同的环境下也会有所差异。开发 Microsoft Windows 应用的 C 程序员倾向于采用匈牙利命名法，并在变量名中混合使用大小写。在 Macintosh 平台下，C 程序员会倾向于在子程序的名字中混合使用大小写，这是因为 Macintosh 工具箱和操作系统子程序最初是为支持 Pascal 接口而设计的。

C++ Conventions

C++的命名规则

深入阅读 (The Elements of C++ Style) (Misfeldt, Bumgardner, and Gray 2004)一书详细描述了 C++ 编程风格。

以下是围绕着 C++ 编程形成的命名规则。

- i 和 j 是整数下标。
- p 是指针。
- 常量、typedef 和预处理宏全部大写 (ALL_CAPS)。
- 类和其他类型的名字混合大小写 (MixedUpperAndLowerCase())。
- 变量名和函数名中的第一个单词小写，后续每个单词的首字母大写——例如，variableOrRoutineName。
- 不把下画线用做名字中的分隔符，除非用于全部大写的名字以及特定的前缀中（如用于标识全局变量的前缀）。

与 C 编程相比，上述规则还远没有形成标准，并且不同的环境也会形成不同的具体规则。

Java Conventions

Java 的规则

深入阅读 《The Elements of Java Style》，2d ed. (Vermeulen et al. 2000)一书详细描述了 Java 编程风格。

与 C 和 C++ 不同，Java 语言的风格约定从一开始就创建好了。

- i 和 j 是整数下标。
- 常量全部大写 (ALL_CAPS) 并用下画线分隔。
- 类名和接口名中每一个单词的首字母均大写，包括第一个单词——例如，ClassOrInterfaceName。
- 变量名和方法名中第一个单词的首字母小写，后续单词的首字母大写——例如，variableOrRoutineName。
- 除用于全部大写的名字之外，不使用下画线作为名字中的分隔符。
- 访问器子程序使用 get 和 set 前缀。

Visual Basic Conventions

Visual Basic 的命名规则

Visual Basic 还没有固定的规则。下一节将就 Visual Basic 给出一份规则建议。

Mixed-Language Programming Considerations

混合语言编程的注意事项

在混合语言环境中编程时，可以对命名规则（以及格式规则、文档规则等）做出优化以提高整体的一致性和可读性——即使这意味着优化后的规则会与其中某种语言所用的规则相冲突。

在本书里，变量名均以小写开头，这符合 Java 的编程实践传统以及部分但并非全部的 C++ 传统。本书把所有子程序名的首字母大写，这遵循了 C++ 规则。在 Java 中所有的方法名都是以小写字母开始的，但是本书对所有语言的子程序名的首字母都大写，从而提高了整体可读性。

Sample Naming Conventions

命名规则示例

上述的标准规则容易使我们忽略前几页里谈论过的有关命名的若干重要事项——包括变量作用域（私用的，类的或者全局的）、类名、对象名、子程序名和变量名之间的差异等。

在命名规则的指导原则长度超过了几页之后，看上去就显得非常复杂。然而，它们没必要变得如此复杂，你也可以按实际需要来加以应用。变量名包含了以下三类信息：

- 变量的内容（它代表什么）
- 数据的种类（具名常量、简单变量、用户自定义类型或者类）
- 变量的作用域（私用的、类的、包的或者全局的作用域）

根据上述指导原则，表 11-3、表 11-4 和表 11-5 给出了 C、C++、Java 和 Visual Basic 的命名规则。这些特殊规则并非是强制性的，但是它们能帮你了解一份非正式的命名规则应包含哪些内容。

表 11-3 C++和 Java 的命名规则示例

实 体	描 述
ClassName	类名混合使用大小写，首字母大写
TypeName	类型定义，包括枚举类型和 <code>typedef</code> ，混合使用大小写，首字母大写
EnumeratedTypes	除遵循上述规则之外，枚举类型总以复数形式表示
localVariable	局部变量混合使用大小写，首字母小写。其名字应该与底层数据类型无关，而且应该反映该变量所代表的事物
routineParameter	子程序参数的格式与局部变量相同
RoutineName()	子程序名混合使用大小写（第 7.3 节已经讨论过什么是好的子程序名）
m_ClassVariable	对类的多个子程序可见（且只对该类可见）的成员变量名用 <code>m_</code> 前缀
g_GlobalVariable	全局变量名用 <code>g_</code> 前缀
CONSTANT	具名常量全部大写
MACRO	宏全部大写
Base_EnumeratedType	枚举类型名用能够反映其基础类型的、单数形式的前缀——例如， <code>Color_Red</code> , <code>Color_Blue</code>

表 11-4 C 的命名规则示例

实 体	描 述
TypeName	类型名混合使用大小写，首字母大写
GlobalRoutineName()	公用子程序名混合使用大小写
f_FileRoutineName()	单一模块（文件）私用的子程序名用 f_前缀
LocalVariable	局部变量混合使用大小写。其名字应该与底层数据类型无关，而且应该反映该变量所代表的事物
RoutineParameter	子程序参数的格式与局部变量相同
f_FileStaticVariable	模块（文件）变量名用 f_前缀
G_GLOBAL_GlobalVariable	全局变量名以 G_前缀和一个能反映定义该变量的模块（文件）的、全部大写的名字开始——例如，G_SCREEN_Dimensions
LOCAL_CONSTANT	单一子程序或者模块（文件）私用的具名常量全部大写——例如，ROWS_MAX
G_GLOBALCONSTANT	全局具名常量名全部大写，并且以 G_前缀和一个能反映定义该具名常量的模块（文件）的、全部大写的名字开始，如 G_SCREEN_ROWS_MAX
LOCALMACRO()	单一子程序或者模块（文件）私用的宏定义全部大写
G_GLOBAL_MACRO()	全局宏定义全部大写，并且以 G_前缀和一个能反映定义该宏的模块（文件）的全部大写名字开始——例如，G_SCREEN_LOCATION()

由于 Visual Basic 对大小写不敏感，因此需要采取一些特殊的规则来区分类型名和变量名。请见表 11-5。

表 11-5 Visual Basic 的命名规则示例

实 体	描 述
C_ClassName	类名混合使用大小写，首字母大写，并且加 C_前缀
T_TypeName	类型定义，包括枚举类型和 typedef，混合使用大小写，首字母大写，并且加 T_前缀
T_EnumeratedTypes	除遵循上述规则之外，枚举类型总以复数形式表示

续表

实 体	描 述
localVariable	局部变量混合使用大小写，首字母小写。其名字应该与底层数据类型无关，并且应该反映该变量所代表的事物
routineParameter	子程序参数的格式与局部变量相同
RoutineName()	子程序名混合使用大小写（第 7.3 节已经讨论过什么是好的子程序名）
m_ClassVariable	只在一个类范围内对该类的多个子程序可见的成员变量名以 m_ 前缀打头
g_GlobalVariable	全局变量名以 g_ 前缀开始
CONSTANT	具名常量全部大写
Base_EnumeratedType	枚举类型名以能够反映其基础类型的、单数形式的前缀开始——例如，Color_Red, Color_Blue

11.5 Standardized Prefixes 标准前缀

深入阅读 《The Hungarian Revolution》 (Simonyi and Heller 1991)一书有对匈牙利命名法的详细描述。

对具有通用含义的前缀标准化，为数据命名提供了一种简洁、一致并且可读性好的方法。有关标准前缀最广为人知的方案是匈牙利命名法，该方案由一组用于指导变量和子程序命名（而不是指导如何给匈牙利人取名！）的详细原则组成，并且曾经一度被广泛用于 Microsoft Windows 编程。尽管目前匈牙利命名法已经不再得到广泛使用，但是使用简洁准确的缩写词的基本命名标准理念却仍然具有价值。

标准化的前缀由两部分组成：用户自定义类型（UDT）的缩写和语义前缀。

User-Defined Type Abbreviations 用户自定义类型缩写

UDT 缩写可以标识被命名对象或变量的数据类型。UDT 缩写可以被用于表示像窗体、屏幕区域以及字体一类的实体。UDT 缩写通常不会表示任何由编程语言所提供的预置数据类型。

UDT 用很短的编码描述，这些编码是为特定的程序创建的，并且经过标准化以在该程序内使用。这些编码有助于用户理解其所代表的实体，如用 wn 代表窗体，scr 代表屏幕区域。表 11-6 列出了一份 UDT 示例，你可能会在开发文字处理程序的时候用到它们。

表 11-6 用于文字处理程序的 UDT 示例

UDT 缩写	含 义
ch	字符 (Character, 这里的字符不是指 C++ 中的字符, 而是指文字处理程序可能用于表示一份文档中的字符的数据类型)
doc	文档 (Document)
pa	段落 (Paragraph)
scr	屏幕区域 (Screen region)
sel	选中范围 (Selection)
wn	窗体 (window)

当你使用 UDT 的时候, 你还要按与 UDT 同样的缩写去定义编程语言的数据类型。这样, 如果你有表 11-6 所列出的那些 UDT, 你就会看到下面这样的数据声明:

```
CH      chCursorPosition;
SCR     scrUserWorkspace;
DOC     docActive
PA      firstPaActiveDocument;
PA      lastPaActiveDocument;
WN      wnMain;
```

同样, 这些例子是与文字处理程序相关的。要把它们用于你自己的项目, 你需要为环境中最常用的那些 UDT 创建 UDT 缩写。

Semantic Prefixes

语义前缀

语义前缀比 UDT 更进一步, 它描述了变量或者对象是如何使用的。语义前缀与 UDT 不同, 后者会根据项目的不同而不同, 而前者在某种程度上对于不同的项目均是标准的。表 11-7 列出了一组标准的语义前缀。

表 11-7 语义前缀

语义前缀	含 义
c	数量 (count, 如记录、字符或者其他东西的个数)
first	数组中需要处理的第一个元素。first 与 min 类似, 但它是相对于当前操作而不是数组本身的
g	全局变量 (global variable)
i	数组的下标 (index into an array)
last	数组中需要处理的最后一个元素。last 与 first 相对应

续表

语义前缀	含 义
lim	数组中需要处理的元素的上限。lim 不是一个合法的下标。它与 last 都是与 first 相对应的概念。不同之处是 lim 表示的是一个数组中并不存在的上界；而 last 表示的则是最终的、合法的元素。通常，lim 等于 last + 1
m	类一级的变量
max	数组或者其他种类的列表中绝对的最后一个元素。max 反映的是数组本身，而不是针对数组的操作
min	数组或者其他种类的列表中绝对的第一个元素
p	指针（pointer）

语义前缀可以全用小写，也可以混合使用大小写，还可以根据需要与 UDT 和其他的语义前缀结合使用。例如，文档中的第一段应该命名为 pa，以表明它是个段落，还要加上 first 以强调它是第一个段落：即 firstPa。一组段落的下标可以命名为 iPa；cPa 是相应的计数值，段落的总数量；firstPaActiveDocument 和 lastPaActiveDocument 表示当前活动文档中的第一个和最后一个段落。

Advantages of Standardized Prefixes 标准前缀的优点



除了具备命名规则所能提供的一般意义上的优点外，标准前缀还为你带来了另外一些好处。由于很多名字都已经标准化了，因此你在一个程序或者类内需要记忆的名字更少了。

标准前缀能够更为精确地描述一些含义比较模糊的名字。min、first、last 和 max 之间的严格区别就显得格外有用。

标准化的前缀使名字变得更加紧凑。例如，你可以用 cpa 而不是 totalParagraphs 表示段落总数。你可以用 ipa 表示一个段落数组的下标，而不是用 indexParagraphs 或者 paragraphsIndex。

最后，在你用的编译器不能检查你所用的抽象数据类型的时候，标准前缀能帮助你准确地对类型做出判断：paReformat = docReformat 很可能不对，因为 pa 和 doc 是不同的 UDT。

标准前缀的主要缺陷是程序员在使用前缀的同时忽略给变量起有意义的名字。如果 ipa 已经能非常明确地表示一个段落数组的下标，那么程序员就不会主动地去想类似于 ipaActiveDocument 这样有意义的名字。为了提高可读性，应该停下来为数组下标起一个具有描述性的名字。

11.6 Creating Short Names That Are Readable 创建具备可读性的短名字



KEY POINT

从某种程度上说，要求使用短变量名是早期计算的遗留物。早期语言，如汇编、一般的 Basic 和 Fortran 都把变量名的长度限制在 2 到 8 个字符，并要求程序员创建简短的名字。早期的计算科学更多的同数学联系在一起，并大量使用求和及其他等式中的 i、j 和 k 等符号。而在现代语言如 C++、Java 和 Visual Basic 里面，实际上你可以创建任何长度的名字；几乎没有任何理由去缩短具有丰富含义的名字。

如果环境真的要求你创建简短的名字，请注意有些缩短名字的方法要好于其他的方法。你可以通过消除冗余的单词、使用简短的同义词以及使用诸多缩写策略中的任意一种来创建更好的短变量名。熟悉多种缩写技巧会很有用，因为没有哪种方法能够适用于所有的情况。

General Abbreviation Guidelines 缩写的一般指导原则

下面是几项用于创建缩写的指导原则。其中的一些原则彼此冲突，所以不要试图同时应用所有的原则。

- 使用标准的缩写（列在字典中的那些常见缩写）。
- 去掉所有非前置元音。（computer 变成 cmprtr, screen 变成 scrn, apple 变成 appl, integer 变成 intgr。）
- 去掉虚词 and, or, the 等。
- 使用每个单词的第一个或前几个字母。
- 统一地在每个单词的第一、第二或者第三个（选择最合适的一个）字母后截断。
- 保留每个单词的第一个和最后一个字母。
- 使用名字中的每一个重要单词，最多不超过三个。
- 去除无用的后缀——ing, ed 等。
- 保留每个音节中最引人注意的发音。
- 确保不要改变变量的含义。
- 反复使用上述技术，直到你把每个变量名的长度缩减到了 8 到 20 个字符，或者达到你所用的编程语言对变量名的限制字符数。

Phonetic Abbreviations

语音缩写

有些人倡导基于单词的发音而不是拼写来创建缩写。于是 skating 就变成了 sk8ing, highlight 变成了 hilite, before 变成了 b4, execute 变成了 xqt, 诸如此类。这样做很像是要人去猜出个性化汽车牌照的意思，我不提倡这么做。作为一项练习，请猜猜下面这些名字各表示什么：

ILV2SK8 XMEQWK S2DTM8O NXTC TRMN8R

Comments on Abbreviations

有关缩写的评论

在创建缩写的时候，会有很多的陷阱在等着你。下面是一些能够用来避免犯错的规则。

不要用从每个单词中删除一个字符的方式来缩写 键入一个字符算不上是什么额外工作，而节省一个字符带来的便利却很难抵消由此而造成的可读性损失。这就像日历中的“Jun”和“Jul”。你只有在非常着急的情况下才有必要把 June 拼成“Jun”。对于大多数删除一个字母的做法而言，你很难回忆起自己是不是删了一个字符。所以，要么删除不止一个字符，要么就把单词拼写完整。

缩写要一致 应该一直使用相同的缩写。例如，要么全用 Num，要么全用 No，不要两个都用。与之类似，不要在一些名字里缩写某个单词而在其他名字里不缩写。比如，不要在有些地方使用完整的单词 Number，同时在其他地方使用 Num 缩写。

创建你能读出来的名字 用 xPos 而不用 xPstn，用 needsCompu 而不用 ndsCmptg。此处可以借助电话来测试——如果你无法在电话中向他人读出你的代码，就请重新给变量起一个更清晰的名字吧（Kernighan and Plauger 1978）。

避免使用容易看错或者读错的字符组合 为了表示 B 的结尾，ENDB 要比 BEND 更好。如果你使用了一种好的分隔技术，那么就不需要这一条原则，因为 B-END、BEnd 或者 b_end 都不会被读错。

使用辞典来解决命名冲突 创建简短名字会带来的一项麻烦就是命名冲突——缩写后名字相同。例如，如果命名长度被限制为 3 个字符，并且你需要在程序中的同一大段使用 fired 和 full revenue disbursal，你可能会不经意地把缩写都写成了 frd。

避免命名冲突的一种简单做法是使用含义不同的不同单词，这样一来，有一部辞典就显得很方便。在本例中，可以用 dismissed 来代替 fired，以及用 complete revenue disbursal 来代替 full revenue disbursal。这样，3 个字母的缩写就分别变成了 dsm 和 crd，从而消除了命名冲突。

在代码里用缩写对照表解释极短的名字的含义 当编程语言只允许用非常短的名字的时候，增加一张缩写对照表来为用户提供更多的变量含义。把该表格作为注释加到一段代码的开始。下面是一个例子：

Fortran示例：良好的名字对照表

```
C ****
C      Translation Table
C
C      Variable      Meaning
C      -----
C      XPOS          x-Coordinate Position (in meters)
C      YPOS          Y-Coordinate Position (in meters)
C      NDSCMP        Needs Computing (=0 if no computation is needed;
C                           =1 if computation is needed)
C      PTGTTL       Point Grand Total
C      PTVLMX       Point Value Maximum
C      PSCRMX       Possible Score Maximum
C ****
```

你可能会认为这种方法已经过时了，但是在2003年中期，我与一家客户合作，该客户有上万行用RPG语言写成的、变量名被限制在6个字符以内的代码。这些要求极短变量名的问题仍然时不时地出现。

在一份项目级的“标准缩写”文档中说明所有的缩写 代码中的缩写会带来两种常见风险。

- 代码的读者可能不理解这些缩写。
- 其他程序员可能会用多个缩写来代表相同的词，从而产生不必要的混乱。

为了同时解决这两个潜在的问题，你可以创建一份“标准缩写”文档来记录项目中用到的全部编码缩写。这份文档既可以是文字处理程序的文档，也可以是电子表格文档。在很大的项目里，它还可以是一个数据库。这份文档应签入（check in）到版本控制系统里，当任何人于任意时间在代码里创建了一种新的缩写时把它签出（check out）来修改。文档中的词条应该按照完整单词排序，而不是按照缩写排序。

这看上去可能显得非常麻烦，但是除了开始的一点额外工作，它事实上是建立了一种在项目中有效地使用缩写的机制。通过对所有用到的缩写加以说明，就解决了上面描述的两种常见风险中的第一种。程序员如果不费力把标准缩写文档从版本控制系统中check out、输入新的缩写并把它check in回去，就不能创建一个新的缩写。这是件好事。它表明，只有当一个缩写在代码中应用非常广泛，程

程序员不惜花上很多精力来为它编写缩写文档时，这一缩写才的的确确应当被创建。

这种方法通过降低程序员创建多余的缩写的可能性，从而解决了第二种风险。想创建缩写的程序员会把缩写文档 `check out` 并输入新的缩写。如果他想缩写的单词已经有了缩写，该程序员就会注意到它，并且去使用该现有的缩写而不是创建一个新的。



KEY POINT

本原则中体现出来的核心问题，是方便编写代码同方便阅读代码两种理念之间的差异。上面的方法很明显会带来代码编写时的麻烦，但是程序员们在整个项目生命周期里会把更多的时间花在阅读代码而不是编写代码之上。这种方法提高了阅读代码的方便性。当一个项目尘埃落定之后，它可能还会提高编写代码的方便性。

记住，名字对于代码读者的意义要比对作者更重要 去读一读你自己写的并且至少有六个月没看过的代码，注意哪些名字是你需要花功夫才能理解其含义的。应下决心改变导致这种混乱的做法。

11.7 Kinds of Names to Avoid 应该避免的名字

下面就哪些变量名应该避免给出指导原则。

避免使用令人误解的名字或缩写 要确保名字的含义是明确的。例如，`FALSE` 常用做 `TRUE` 的反义词，如果用它作为“Fig and Almond Season”的缩写就很糟糕了。

避免使用具有相似含义的名字 如果你能够交换两个变量的名字而不会妨碍对程序的理解，那么你就需要为这两个变量重新命名了。例如，`input` 和 `inputValue`, `recordNum` 和 `numRecords`, 以及 `fileNumber` 和 `fileIndex` 在语义上非常相似，因此，如果把它们用在同一段代码里，会很容易混淆它们，并且犯下一些微妙且难以发现的错误。

交叉参考 有一个术语用来描述相似变量名之间的差异，即“心理距离 (*psychological distance*)”。在第23.4节有关于它的详细叙述。¹

避免使用具有不同含义但却有相似名字的变量 如果你有两个名字相似但含义不同的变量，那么试着给其中之一重新命名，或者修改你的缩写。避免使用类似于 `clientRecs` 和 `clientReps` 这样的名字。它们之间只有一个字母的差异，并且这个字母很难被注意到。应该采用至少有两个字母不同的名字，或者把不同之处放在名字的开始或者结尾。`clientRecords` 和 `clientReports` 就要比原来的名字好。²

¹译注：即增加标识符之间的 Hamming 距离。

²译注：即增加标识符之间的 Hamming 距离。

避免使用发音相近的名字，比如 *wrap* 和 *rap*。当你试图和别人讨论代码的时候，同音异义字就会产生麻烦。我家猫对于极限编程的抱怨之一是它过于聪明地使用了 *Goal Donor* 和 *Gold Owner* 两个概念，事实上它们读起来很难区分。你最终就会同别人展开类似于这样的对话：

我刚和 *Goal Donor* 谈过话——

你是说 “*Gold Owner*” 还是 “*Goal Donor*”？

我是说 “*Goal Donor*。”

什么？

GOAL --- DONOR!

好了，*Goal Donor*。你不应该大喊大叫，烦死了。

你是说 “*Gold Donut*” 吗？

记住，电话测试也适用于测试发音相近的名字，就像它适用于对付稀奇古怪的缩写名一样。

避免在名字中使用数字 如果名字中的数字真的非常重要，就请使用数组来代替一组单个的变量。如果数组不合适，那么数字就更不合适。例如，要避免使用 *file1* 和 *file2*，或者 *total1* 和 *total2*。你几乎总能想出一种比在名字的最后加上 1 或 2 更好的方法来区分两个变量。我不能说永远不要用数字。有些现实世界的事物（例如 203 国道、Route 66 或者 Interstate 405）中就嵌入了数字。不过在你创建一个含有数字的名字之前，请考虑是否还有更好的选择。

避免在名字中拼错单词 弄清楚单词实际应该怎么拼写是够难的。想让人们想出什么是“正确的”错拼更是勉为其难。比如说，把 *highlight* 错拼为 *hilite* 以省下 3 个字符，让读者很难想起这是 *highlight* 的错拼。是 *highlite*？*hilite*？*hilight*？*hilit*？*jai-a-lai-t*？天知道。

避免使用英语中常常拼错的单词 *Absense*, *acummulate*, *acsend*, *catender*, *concieve*, *defferred*, *definate*, *independance*, *occassionally*, *prefered*, *reciept*, *superseed*以及其他很多英语单词经常会拼错。很多英语手册中会包含一份常常拼错单词的清单。避免在你的变量名中使用这些单词。

不要仅靠大小写来区分变量名 如果你在用一种大小写敏感的语言如 C++ 做开发，你也许会倾向于使用 *frd* 来代表 *fired*，用 *FRD* 代表 *final review duty*，以及用 *Frd* 来代表 *full revenue disbursal*。应该避免这样做。尽管这些名字都是唯一的，但把其中任一名字与某个特殊的含义关联起来的方式却太随心所欲，且让人感到迷惑。*Frd* 会很容易地与 *final review duty* 联系起来，*FRD* 也会被认为是 *full revenue disbursal*，没有逻辑法则能够帮助你或者其他记不住谁是谁。

避免使用多种自然语言 在多语言的项目中，对于全部代码，如类名、变量名等，要强制使用一种自然语言，阅读其他程序员的代码可以称为一种挑战；阅读用火星东南部的语言写成程序代码则是绝无可能的。

一种更微妙的问题产生于英语的变体。如果一个项目在多个说英语的国家进行，就应该以其中一种英语版本为标准，以便你不用一直为在代码中应该使用“color”还是“colour”，“check”还是“cheque”等感到迷惑。

避免使用标准类型、变量和子程序的名字 所有的编程语言指南都会包含一份该语言保留的和预定义的名字列表。请不时读一读这份列表，以确保你自己的命名没有冒犯你所用的语言。例如，下面代码在 PL/I 中是合法的，但除非你是个十足的傻瓜，否则是不会这么用的：

```
if if = then then
    then = else;
else else = if;
```



不要使用与变量含义完全无关的名字 如果你在程序中点缀着诸如 `margaret` 和 `pookie` 这样的名字，就会在事实上保证没有其他人能够理解它。避免用你男朋友的名字、妻子的名字、最喜欢的啤酒的名字或者其他自作聪明的（也就是傻的）名字来为变量命名，除非你的程序真的是与你的男朋友、妻子或者最爱的啤酒有关。即使如此，你也应该明智地认识到这其中的每一项都可能会变的，所以 `boyfriend`、`wife` 和 `favoriteBeer` 这些通用的名字会更好！

避免在名字中包含易混淆的字符 要意识到有些字符看上去是非常接近，很难把它们区分开来。如果两个名字的唯一区别就是这些字符中的一个，那么你区分这些名字就会变得非常困难。例如，请试着把下列每组中不属于该组的名字圈出来：

<code>eyeChart1</code>	<code>eyeChartI</code>	<code>eyeChart1</code>
<code>TTLCONFUSION</code>	<code>TTLCONFUSION</code>	<code>TTLC0NFUSION</code>
<code>hard2Read</code>	<code>hardZRead</code>	<code>hard2Read</code>
<code>GRANDTOTAL</code>	<code>GRANDTOTAL</code>	<code>6RANDTOTAL</code>
<code>tt15</code>	<code>tt1S</code>	<code>tt1S</code>

很难区分的“对”包括 (1(数字 1)和 l(小写字母 L))，(1 和 I(大写字母 i))，(.和,)，(0(零)和 O(大写字母 o))，(2 和 Z)，(;和:)，(S 和 5) 以及 (G 和 6)。

交叉参考 关于使用数据方面的讨论，见第 10 章中“使用变量的一般事项”的核对表。

像这样的细节真有用吗？没错！Gerald Weinberg 报导说，在 20 世纪 70 年代，一条 Fortran FORMAT 语句中的句号错写成了逗号。结果科学家们算错了太空飞船的轨道，导致了太空探测器的丢失——损失高达 16 亿美元（Weinberg 1983）。

cc2e.com/1191

CHECKLIST: Naming Variables**核对表：变量命名****命名的一般注意事项**

- 名字完整并准确地表达了变量所代表的含义吗？
- 名字反映了现实世界的问题而不是编程语言方案吗？
- 名字足够长，可以让你无须苦苦思索吗？
- 如果有计算值限定符，它被放在名字的最后吗？
- 名字中用 Count 或者 Index 来代替 Num 了吗？

为特定类型的数据命名

- 循环下标的名字有意义吗（如果循环的长度超出了一两行代码或者出现了嵌套循环，那么就应该是 i、j 或者 k 以外的其他名字）？
- 所有的“临时”变量都重新命以更有意义的名字了吗？
- 当布尔变量的值为真时，变量名能准确表达其含义吗？
- 枚举类型的名字中含有能够表示其类别的前缀或后缀了吗？例如，把 color_用于 Color_Red, Color_Green, Color_Blue 等了吗？
- 具名常量是根据它所代表的抽象实体而不是它所代表的数字来命名的吗？

命名规则

- 规则能够区分局部数据、类的数据和全局数据吗？
- 规则能够区分类型名、具名常量、枚举类型和变量名吗？
- 规则能够在编译器不强制检测只读参数的语言里标识出子程序中的输入参数吗？
- 规则尽可能地与语言的标准规则兼容吗？
- 名字为了可读性而加以格式化吗？

短名字

- 代码用了长名字吗（除非有必要使用短名字）？
- 是否避免只为了省一个字符而缩写名字的情况？
- 所有单词的缩写方式都一致吗？

- 名字能够读出来吗？
- 避免使用容易被看错或者读错的名字吗？
- 在缩写对照表里对短名字做出说明吗？

常见命名问题：你避免使用……

- ……容易让人误解的名字吗？
- ……有相近含义的名字吗？
- ……只有一两个字符不同的名字吗？
- ……发音相近的名字吗？
- ……包含数字的名字吗？
- ……为了缩短而故意拼错的名字吗？
- ……英语中经常拼错的名字吗？
- ……与标准库子程序名或者预定义变量名冲突的名字吗？
- ……过于随意的名字吗？
- ……含有难读的字符的名字吗？

Key Points

要点

- 好的变量名是提高程序可读性的一项关键要素。对特殊种类的变量，比如循环下标和状态变量，需要加以特殊的考虑。
- 名字要尽可能地具体。那些太模糊或者太通用以致于能够用于多种目的名字通常都是很不好的。
- 命名规则应该能够区分局部数据、类数据和全局数据。它们还应当可以区分类型名、具名常量、枚举类型名字和变量名。
- 无论做哪种类型项目，你都应当采用某种变量命名规则。你所采用的规则的种类取决于你的程序的规模，以及项目成员的人数。
- 现代编程语言很少需要用到缩写。如果你真的要使用缩写，请使用项目缩写词典或者标准前缀来帮助理解缩写。
- 代码阅读的次数远远多于编写的次数。确保你所取的名字更侧重于阅读方便而不是编写方便。