无论是命名变量、函数还是类，都可以使用很多相同的原则。我们喜欢把名字当做一条小小的注释。尽管空间不算很大，但选择一个好名字可以让它承载很多信息。

> **关键思想**
>
> 把信息装入名字中。

- 选择专业的词。
- 避免泛泛的名字（或者说要知道什么时候使用它）。
- 用具体的名字代替抽象的名字。
- 使用前缀或后缀来给名字附带更多信息。
- 决定名字的长度。
- 利用名字的格式来表达含义。

对于空泛名字的裁定

如你所见，在某些情况下空泛的名字也有用处。

> **建议**
>
> 如果你要使用像tmp、it或者retval这样空泛的名字，那么你要有个好的理由。

很多时候，仅仅因为懒惰而滥用它们。这可以理解，如果想不出更好的名字，那么用个没有意义的名字，像foo，然后继续做别的事，这很容易。但如果你养成习惯多花几秒钟想出个好名字，你会发现你的“命名能力”很快提升。

这个宏定义成：

```
#define DISALLOW_EVIL_CONSTRUCTORS(ClassName) \
    ClassName(const ClassName&); \
    void operator=(const ClassName&);
```

通过把这个宏放在类的私有部分中，这两个方法[译注1]成为私有的，所以不能用它们，即使意料之外的使用也是不可能的。

然而DISALLOW_EVIL_CONSTRUCTORS这个名字并不是很好。对于“邪恶”这个词的使用包含了对于一个有争议话题过于强烈的立场。更重要的是，这个宏到底禁止了什么这一点是不清楚的。它禁止了operator=()方法，但这个方法甚至根本就不是构造函数！

这个名字使用了几年，但最终换成了一个不那么嚣张而且更具体的名字：

```
#define DISALLOW_COPY_AND_ASSIGN(ClassName) ...
```

带单位的值

如果你的变量是一个度量的话（如时间长度或者字节数），那么最好把名字带上它的单位。

除了时间，还有很多在编程时会遇到的单位。下表列出一些没有单位的函数参数以及带单位的版本：

函数参数	带单位的参数
Start(int delay)	delay → delay_secs
CreateCache(int size)	size → size_mb
ThrottleDownload(float limit)	limit → max_kbps
Rotate(float angle)	angle → degrees_cw

总结

很难思考巨大的表达式。本章给出了几种拆分表达式的方法，以便读者可以一段一段地消化。

一个简单的技术是引入“解释变量”来代表较长的子表达式。这种方式有三个好处：

- 它把巨大的表达式拆成小段。
- 它通过用简单的名字描述子表达式来让代码文档化。
- 它帮助读者识别代码中的主要概念。

另一个技术是用德摩根定理来操作逻辑表达式——这个技术有时可以把布尔表达式用更整洁的方式重写（例如if (!(a && !b))变成if (!a || b)）。

所以经验原则是：**团队的新成员是否能理解这个名字的含义**？如果能，那可能就没有问题。

例如，对程序员来讲，使用eval来代替evaluation，用doc来代替document，用str来代替string是相当普遍的。因此如果团队的新成员看到FormatStr()可能会理解它是什么意思，然而，理解BEManager可能有点困难。

丢掉没用的词

有时名字中的某些单词可以拿掉而不会损失任何信息。例如，ConvertToString()就不如ToString()这个更短的名字，而且没有丢失任何有用的信息。同样，不用DoServeLoop()，ServeLoop()也一样清楚。

该例子中的大部分格式都很常见，使用CamelCase来表示类名，使用lower_separated来表示变量名。但有些规范也可能会出乎你的意料。

例如，常量的格式是kConstantName而不是CONSTANT_NAME。这种形式的好处是容易和#define的宏区分开，宏的规范是MACRO_NAME。

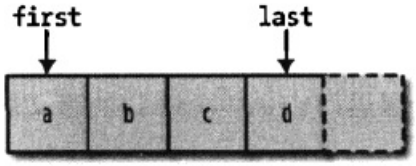
类成员变量和普通变量一样，但必须以一条下划线结尾，如offset_。刚开始看，可能会觉得这个规范有点怪，但是能立刻区分出是成员变量还是其他变量，这一点还是很方便的。例如，如果你在浏览一个大的方法中的代码，看到这样一行：

```
stats.clear();
```

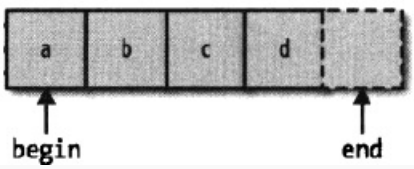
你本来可能要想“stats属于这个类吗？这行代码是否会改变这个类的内部状态？”如果用了member_这个规范，你就能迅速得到结论：“不，stats一定是个局部变量。否则它就会命名为stats_。”

- **使用具体的名字来更细致地描述事物**——ServerCanStart()这个名字就比CanListenOnPort更不清楚。
- **给变量名带上重要的细节**——例如，在值为毫秒的变量后面加上_ms，或者在还需要转义的，未处理的变量前面加上raw_。
- **为作用域大的名字采用更长的名字**——不要用让人费解的一个或两个字母的名字来命名在几屏之间都可见的变量。对于只存在于几行之间的变量用短一点的名字更好。
- **有目的地使用大小写、下划线等**——例如，你可以在类成员和局部变量后面加上"_"来区分它们。

推荐用first和last来表示包含的范围



推荐用begin和end来表示包含/排除范围



下面是个危险的例子：

```
bool read_password = true;
```

这会有两种截然不同的解释：

- 我们需要读取密码。
- 已经读取了密码。

在本例中，最好避免用“read”这个词，用need_password或者user_is_authenticated这样的名字来代替。

通常来讲，加上像is、has、can或should这样的词，可以把布尔值变得更明确。

例如，SpaceLeft()函数听上去像是会返回一个数字，如果它的本意是返回一个布尔值，可能HasSapceLeft()个这名字更好一些。

最后，最好避免使用反义名字。例如，不要用：

```
bool disable_ssl = false;
```

而更简单易读（而且更紧凑）的表示方式是：

```
bool use_ssl = true;
```

在你决定使用一个名字以前，要吹毛求疵一点，来想象一下你的名字会被误解成什么。最好的名字是不会误解的。

当要定义一个值的上限或下限时，max_和min_是很好的前缀。对于包含的范围，first和last是好的选择。对于包含/排除范围，begin和end是最好的选择，因为它们最常用。

当为布尔值命名时，使用is和has这样的词来明确表示它是个布尔值，避免使用反义的词（例如disable_ssl）。

要小心用户对特定词的期望。例如，用户会期望get()或者size()是轻量的方法。

现在，很明显这里有4个测试，每个使用了不同的参数。尽管所有的“脏活”都放在CheckFullName()中，但是这个函数也没那么差：

```
void CheckFullName(string partial_name,
                   string expected_full_name,
                   string expected_error) {
    // database_connection is now a class member
    string error;
    string full_name = ExpandFullName(database_connection, partial_name, &error);
    assert(error == expected_error);
    assert(full_name == expected_full_name); }
```

尽管我们的目的仅仅是让代码更有美感，但这个改动同时有几个附带的效果：

- 它消除了原来代码中大量的重复，让代码变得更紧凑。
- 每个测试用例重要的部分（名字和错误字符串）现在都变得很直白。以前，这些字符串是混杂在像database_connection和error这样的标识之间的，这使得一眼看全这段代码变得很难。
- 现在添加新测试应当更简单

这个故事想要传达的寓意是使代码“看上去漂亮”通常会带来不限于表面层次的改进，它可能会帮你把代码的结构做得更好。

总结

大家都愿意读有美感的代码。通过把代码用一致的、有意义的方式 “格式化”，可以把代码变得更容易读，并且可以读得更快。

下面是讨论过的一些具体技巧：

- 如果多个代码块做相似的事情，尝试让它们有同样的剪影。
- 把代码按“列”对齐可以让代码更容易浏览。
- 如果在一段代码中提到A、B和C，那么不要在另一段中说B、C和A。选择一个有意义的顺序，并始终用这样的顺序。
- 用空行来把大块代码分成逻辑上的“段落”。

不要给不好的名字加注释——应该把名字改好

标记	通常的意义
TODO：	我还没有处理的事情
FIXME：	已知的无法运行的代码
HACK：	对一个问题不得不采用的比较粗糙的解决方案
XXX：	危险！这里有重要的问题

你的团队可能对于是否可以使用及何时使用这些标记有具体的规范。例如，TODO： 可能只用于重要的问题。如果是这样，你可以用像todo:（小写）或者maybe-later:这样的方法表示次要的缺陷。

重要的是你应该可以随时把代码将来应该如何改动的想法用注释记录下来。这种注释给读者带来对代码质量和当前状态的宝贵见解，甚至可能会给他们指出如何改进代码的方向。

给常量加注释

当定义常量时，通常在常量背后都有一个关于它是什么或者为什么它是这个值的“故事”。例如，你可能会在代码中看到如下常量：

```
NUM_THREADS = 8
```

这一行看上去可能不需要注释，但很可能选择用这个值的程序员知道得比这个要多：

```
NUM_THREADS = 8 # as long as it's >= 2 * num_processors, that's good enough.
```

现在，读代码的人就有了调整这个值的指南了（比如，设置成1可能就太低了，设置成50又太夸张了）。

“具名函数参数”的注释

假设你见到下面这样的函数调用：

```
Connect(10, false);
```

因为这里传入的整数和布尔型值，使得这个函数调用有点难以理解。

在像Python这样的语言中，你可以按名字为参数赋值：

```
def Connect(timeout, use_encryption):  ...
```

条件语句中参数的顺序

下面的两段代码哪个更易读？

```
if (length >= 10)
```

还是

```
if (10 <= length)
```

对大多数程序员来讲，第一段更易读。那么，下面的两段呢？

```
while (bytes_received < bytes_expected)
```

还是

```
while (bytes_expected > bytes_received)
```

仍然是第一段更易读。可为什么会这样？通用的规则是什么？你怎么才能决定是写成a<b好一些，还是写成b>a好一些？

下面的这条指导原则很有帮助：

比较的左侧	比较的右侧
“被问询的”表达式，它的值更倾向于不断变化	用来做比较的表达式，它的值更倾向于常量

这条指导原则和英语的用法一致[译注1]。我们会很自然地说：“如果你的年收入至少是10万美元”或者“如果你不小于18岁。”而“如果18岁小于或等于你的年龄”这样的说法却很少见。

这也解释了为什么while(bytes_received < bytes_expected)有更好的可读性。bytes_received是我们在检查的值，并且在循环的执行中它在增长。当用来做比较时，byte_expected则是更“稳定”的那个值。

总结

有几种方法可以让代码的控制流更易读。

在写一个比较时（while (bytes_expected > bytes_received)），把改变的值写在左边并且把更稳定的值写在右边更好一些（while (bytes_received < bytes_expected)）。

你也可以重新排列if/else语句中的语句块。通常来讲，先处理正确的/简单的/有趣的情况。有时这些准则会冲突，但是当不冲突时，这是要遵循的经验法则。

某些编程结构，像三目运算符（:?）、do/while循环，以及goto经常会导致代码的可读性变差。最好不要使用它们，因为总是有更整洁的代替方式。

嵌套的代码块需要更加集中精力去理解。每层新的嵌套都需要读者把更多的上下文“压入栈”。应该把它们改写成更加“线性”的代码来避免深嵌套。

通常来讲提早返回可以减少嵌套并让代码整洁。“保护语句”（在函数顶部处理简单的情况时）尤其有用。

总结

很难思考巨大的表达式。本章给出了几种拆分表达式的方法，以便读者可以一段一段地消化。

一个简单的技术是引入“解释变量”来代表较长的子表达式。这种方式有三个好处：

- 它把巨大的表达式拆成小段。
- 它通过用简单的名字描述子表达式来让代码文档化。
- 它帮助读者识别代码中的主要概念。

另一个技术是用德摩根定理来操作逻辑表达式——这个技术有时可以把布尔表达式用更整洁的方式重写（例如`if (!(a && !b))`变成`if (!a || b)`）。

实际上，让所有的变量都“缩小作用域”是一个好主意，并非只是针对全局变量。

> **关键思想**
>
> **让你的变量对尽量少的代码行可见。**

很多编程语言提供了多重作用域/访问级别，包括模块、类、函数以及语句块作用域。通常越严格的访问控制越好，因为这意味着该变量对更少的代码行“可见”。

为什么要这么做？因为这样有效地减少了读者同时需要考虑的变量个数。如果你能把所有的变量作用域都减半，那么这就意味着同时需要思考的变量个数平均来讲是原来的一半。

另一个对类成员访问进行约束的方法是“尽量使方法变成静态的”。静态方法是让读者知道“这几行代码与那些变量无关”的好办法。

或者还有一种方式是“把大的类拆分成小一些的类”。这种方法只有在这些小一些的类事实上相互独立时才能发挥作用。如果你只是创建两个类来互相访问对方的成员，那你什么目的也没达到。

把大文件拆分成小文件，或者把大函数拆分成小函数也是同样的道理。这么做的一个重要的动机就是数据（即变量）分离。

但是不同的语言有不同的管理作用域的规则。我们接下来给出一些与变量作用域相关的更有趣规则。

总结

本章是关于程序中的变量是如何快速累积而变得难以跟踪的。你可以通过减少变量的数量和让它们尽量“轻量级”来让代码更有可读性。具体有：

- **减少变量**，即那些妨碍的变量。我们给出了几个例子来演示如何通过立刻处理结果来消除“中间结果”变量。
- **减小每个变量的作用域**，越小越好。把变量移到一个有最少代码可以看到它的地方。眼不见，心不烦。
- **只写一次的变量更好**。那些只设置一次值的变量（或者`const`、`final`、常量）使得代码更容易理解。

这是一个不相关子问题的经典例子，应该把它抽取到一个新的函数中，比如ReadFileToString()。现在，你代码库的其他部分可以当做C++语言中确实有ReadFileToString()这个函数。

通常来讲，如果你在想：“我希望我们的库里有XYZ()函数”，那么就写一个！（如果它还不存在的话）经过一段时间，你会建立起一组不错的工具代码，后者可以应用于多个项目。

创建大量通用代码

ReadFileToString()和format_pretty()这两个函数是不相关子问题的好例子。它们是如此基本而广泛适用，所以很可能会在多个项目中重用。代码库常常有个专门的目录来存放这种代码（例如util），这样它们就很方便重用。

通用代码很好，因为“它完全地从项目的其他部分中解耦出来”。像这样的代码容易开发，容易测试，并且容易理解。想象一下如果你所有的代码都如此会怎样！

想一想你使用的众多强大的库和系统，如SQL数据库、JavaScript库和HTML模板系统。你不用操心它们的内部——那些代码与你的项目完全分离。其结果是，你项目的代码库仍然较小。

总结

对本章一个简单的总结就是“把一般代码和项目专有的代码分开”。其结果是，大部分代码都是一般代码。通过建立一大组库和辅助函数来解决一般问题，剩下的只是让你的程序与众不同的核心部分。

这个技巧有帮助的原因是它使程序员关注小而定义良好的问题，这些问题已经同项目的其他部分脱离。其结果是，对于这些子问题的解决方案倾向于更加完整和正确。你也可以在以后重用它们。

进一步的改进

对于当初的大段代码来讲这个新版本算是有了改进。请注意我们甚至不用创建新函数来完成这个清理工作。像前面提到的，“一次只做一件事情”这个想法有助于不考虑函数的边界。

然而，也可以用另一种方法改进这段代码，通过引入3个辅助函数：

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State"][ExitState(hd)]++;
    counts["Http Response"][HttpResponse(hd)]++;
    counts["Content-Type"][ContentType(hd)]++;
}
```

这些函数会抽取出对应的值，或者返回"unknown"。例如：

```
string ExitState(HttpDownload hd) {
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
        return ExitStateTypeName(hd.event_log().exit_state());
    } else {
        return "unknown";
    }
}
```

请注意在这个做法中甚至没有定义任何变量！像第9章所提到的那样，保存中间结果的变量往往可以完全移除。

在这种方法里，我们简单地把问题从不同的角度“切开”。两种方法都很有可读性，因为它们让读者一次只需要思考一件事情。