

An Introduction to Perl for Biologists



Beginning Perl for Bioinformatics



O'REILLY®

James Tisdall

前 言

什么是生物信息学？

生物学数据正在飞速增长。一段时间以来，GenBank 和 PDB (Protein Data Bank) 等公共数据库都在以指数级别增长。随着万维网 (World Wide Web) 的到来，以及快速的网络连接，在世界上的任意一个地方，都可以快速、简便且廉价地获取到这些数据库中的数据和大量具有特定用途的程序。作为结果，在生物学研究的进步中，基于计算机的工具现在正起着越来越关键的作用。

生物信息学是一个快速发展的学科领域，它应用计算工具和技术来管理并分析生物学数据。生物信息学这个术语还相对较新，在此处的定义，它还包含了“计算生物学”和其他专业术语的含义。在生物学研究中使用计算机，要比生物信息学专业术语的使用早许多年。比如，通过 X 射线晶体数据来确定蛋白质的 3D 结构，一直以来都是依赖于计算机分析。在本书中，我把在生物学研究中计算机的使用就看做生物信息学。但是，需要强调的重要一点是，其他人可能会把这些术语严格区分开来。特别的是，当指代对于 *C. elegans* (线虫)、*Arabidopsis* (拟南芥) 和 *Homo sapiens* (人) 等物种的全基因组进行大规模测序和分析时的数据和技术时，通常都会使用生物信息学这个术语。

生物信息学能做什么

这是在实际中应用生物信息学的一个简单的例子。假设你发现了一个非常有趣的小鼠 DNA 的片段，你猜想它可能会为人类的致命脑肿瘤的发生发展提供线索。在对 DNA 进行测序后，你使用基于网络的序列比对工具，比如 BLAST，在 GenBank 和其他数据资源中进行了检索。尽管你找到了一些相关的序列，但是你并没有找到你猜想的它和脑肿瘤之间联系的直接的证据或其他信息。你知道这些公共的遗传数据库每天都在快速增长这。你可能会想每天都进行一次检索，把结果和上一次的检索结果进行比较，来看看在数据库中是不是有新的东西出现。但这每天都会花费一两个小时的时间！幸运的是，你会 Perl。经过一天的努力，你（使用 BioPerl 等模块）编写了一个程序，可以自动每天对你的 DNA 序列在 GenBank 中进行一次 BLAST，并把结果和前一天的结果进行比较，如果有任何变换它就会发送邮件给你。这个程序是如此的有用，以至于你开始把它应用在其他序列上，而你的同时也开始使用它。在几个月的时间内，你一天的努力节省了你团队数周的时间。这个例子来源于真实事件。现在你已经可以使用现成的程序来达成你的目的，甚至有网站可以让你提交自己的 DNA 序列和邮箱地址，它们会为你完成所有的工作！

当你把计算的强大能力应用的生物学问题中时，这只是一个很小的例子。这就是生物信息学。

关于本书

本书是一个指南，直到生物学家如何去编程，并且它是为程序员新手设计的。除了个别以外，所有的例子和练习题使用的都是生物学数据。本书的目的有两个：教授编程技能，同时把它们应用到感兴趣的生物学领域中。

我想让你快速上手，并且尽快且愉快地进行编程。我会进行简洁的解释，而不会把一切都囊括其中。我不会对编程概念进行严格的定义，因为这可能会喧兵夺主，干扰你的学习。

Perl 语言使得通过快速编写真实的程序来开始学习称为可能。当你继续阅读本书和在线的 Perl 文档时，你会逐渐补充细节性的知识，在实践中学习，不断提高你对编程概念的理解。

根据你学习风格的不同，可能会通过不同的方式找到这些材料。一种方式，就像国王郑重地对爱丽丝说的那样，“从开始的地方开始吧，一直读到末尾，然后停止。”（《爱丽丝漫游仙境》中的这句话常常被用来作为算法的一个诙谐的定义。）作为一本叙述性的教材，本书的材料就是以从头到尾阅读的方式进行组织的。

另一种方式是把程序摘抄下来，放到你的计算机中，运行它们，看看它们都做了什么，可能还尝试一下去改变程序中的某个地方，看看你的修改有什么影响。这种方式可能会和快速浏览各章节的文字结合起来。当程序员学习一个新的编程语言时，这是最常用的一种学习方式。基本上，你通过人造例子进行学习，目的是真实的程序。

任何因为生物信息学而学习 perl 编程的读者，都应该尝试一下大多数章节末尾的练习题。它们基本上是按照由易到难的顺序进行排列的，并且排在后面的一些练习题相当有难度，可能作为课堂项目更加合适一些。因为在 Perl 中处理问题的方法不止一种，所以对于每一个练习题来说都没有绝对正确的唯一答案。如果你是一个程序员新手，你可以尝试尽量用不同的方法来解答一个练习题，那么对于这个练习题你就已经很好的掌握了。对于练习题，我推荐的解答可以在 <http://www.oreilly.com/catalog/begperlbio> 找到。

我希望，本书中的材料不仅仅作为实践性的指南。如果你觉得生物信息学本身是一个前途无限的研究方向或者是正在进行的研究的附属物，它还能引领你编写出研究性的程序。

本书为谁编写

本书是为生物学家编写的实践性编程指南。

在生物学研究和发展中，编程技能现在已经是必需的了。历史上，对于实验室的生物学家来说，编程并不被看做一个关键性的技能。然而，如今生物学的发展趋势使得对于大规模数据的计算机分析成为了许多研究程序的重中之重。本书意在作为繁忙的生物学家手头的一本不算厚的教程，帮助它们获得实践性的生物信息学编程技能。所以，如果你是一个午休要学习编程的生物学家，这本书就是为你编写的。它的目的是教会你快速、愉快地编写有用的实践性的生物信息学程序。

本书把编程作为一个重要的实验室技能，它是一个编程指南，包括了一堆的手册或者编程技术，这些在实验室中都是急需且非常有用的。但是它的初衷还是教授编程，而不是构建一个全面的工具包。

在实验室的实验台和计算机程序之间，确实有一些技能和方法上的相似之处。不管是在一天的课程中，还是在整个的生物学研究职业生涯中，许多人确实都能够从跑胶转到编写 Perl 程序。当然，编程是一个有着自己独特方法和专业术语的独特学科，所以必须以其独特的方式学习特有的术语。但确实有“杂交”存在（如果你能原谅这两个学科之间的隐喻）。

本书的练习题由不同难度的题目组成，因此既可以把它当做课堂教材，也可以用来自学。（几乎）所有的例子和练习都是基于真实的生物学问题，所以本书将对大多数常见的生物信息学编程问题和最常见的基于计算机的生物学数据进行一个很好的介绍。

本书的网站，<http://www.oreilly.com/catalog/begperlbio>，包含了书中的所有程序代码便于下载，既包括练习和解答，也包括勘误表和其他的一些信息。¹

¹程序代码，或者简称代码，表示一个计算机程序——一个程序员写到一个文件中的真实的 Perl 语言命令。

为什么我应该学习编程？

因为许多研究人员把他们的工作描述为“生物信息学”，但它们并不编写程序，而是使用其他人编写的程序，所以你可能会问，“我真的需要学习编程才能做生物信息学吗？”在某种层面上讲，答案是否定的，你并不需要学习编程。使用现成的工具，你可以完成很多的工作，而且有相关的书籍和文档帮助你学习这些工具的使用。但是从另一个更高的层面上讲，答案因问题的不同而异。当你用现成的工具无法完成你要做的东西时会发生什么？当你无法找到一个工具来完成你实际的一个任务、而且你找不到一个人可以为你编写程序时会发生什么？

从这点来看，你需要学习编程。即使你仍然主要依赖于现有的程序和工具，去学习编程从而能够写出小的程序也是值得的。小的程序会令人难以置信的有用。比如，通过一定的练习，你会学会编写程序来运行其他的程序，从而节省你大把的时间，避免你坐在计算机前手动去做这些事情。

许多科学家都是从编写小的程序开始的，然后发现他们真的很喜欢编程。作为一名程序员，你永远都不需要担心找不到满足你需要的合适的工具，因为你可以自己编写它们。本书就是助你起步的。

本书的结构

本书总共包含十三章和两个附录。下面是对它们的一个简短介绍：

第 1 章

本章涵盖了分子生物学中一些关键性的概念，以及生物学和计算机科学是纠缠在一起的。

第 2 章

本章向你展示如何让 Perl 在你的计算机上运行起来。

第 3 章

第 3 章对程序员完成他们工作的方式进行了概述。解释了一些好的程序员使用的最重要的实践性的策略，并对如何寻找编程过程中遇到的问题的答案进行了详细的规划。通过简短的描述性实例的分析，使这些观点更加具体化，展示程序员是如何针对一个问题寻找解决方案的。

第 4 章

在第 4 章中，你开始使用 DNA 和蛋白质来编写 Perl 程序。编写程序把 DNA 转录成 RNA，把序列片段连接起来，获得 DNA 的反向互补序列，从文件读取序列数据，等等。

第 5 章

本章继续示范 Perl 语言的基础，编写程序查找 DNA 或蛋白质中的基序，实现通过键盘与用户的交互，把数据保存到文件中，使用循环和条件测试，使用正则表达式，以及操作字符串和数组。

第 6 章

本章在两个方向上拓展了 Perl 的基础知识：一个是子程序，它是进行结构化编程的一种重要的方式，一个是 Perl 调试器的使用，它可以在运行 Perl 程序的时候对其进行详细的检查。

第 7 章

遗传突变对于生物学来说是最基本的，通过使用 Perl 中的随机数生成器可以把以随机事件对其进行建模。本章使用随机数来生成 DNA 序列数据集，不断得突变 DNA 序列。循环，子程序和词汇作用域也会在本章中进行讨论。

第 8 章

本章演示了如何使用遗传密码把 DNA 翻译成蛋白质。它还涵盖了 Perl 编程语言的更多内容，比如散列数据类型、排序和未排序的数组、折半查找、关系型数据库、DBM 以及如何处理 FASTA 格式的序列数据。

第 9 章

本章包含对 Perl 正则表达式的介绍。本章的主要焦点是编写程序来计算一个 DNA 序列的限制酶切图谱。

第 10 章

GenBank (Genetic Sequence Data Bank) 对于现代生物学和生物信息学是非常重要的。在本章中，你会学习如何编写程序从 GenBank 文件和库中提取信息。你也会制作一个数据库，来创建自己的对于 GenBank 库的快速访问和检索。

第 11 章

本章编写一个能解析 PDB (Protein Data Bank) 文件的程序。在编写这个程序的过程中，会遇到一些有趣的 Perl 技术，比如查找并迭代大量的文件，以及在

Perl 程序中控制其他的生物信息学程序。

第 12 章

第 12 章逐步编写一些可以解析 BLAST 输出文件的代码。此外还会提到 BioPerl 项目与它的 BLAST 解析器，以及在 Perl 中格式化输出的其他一些方法。

第 13 章

第 13 章展望了一些超越本书范畴的主题。

附录 A

此处收集的是 Perl 和生物信息学编程的相关资源，比如书籍和网站。

附录 B

这是对本书中涉及到的 Perl 知识点的总结，还有一些其他的東西。

本书中使用的约定

本书中使用下面这些约定：

斜体 (*Italic*)

用于命令、文件名、目录名、变量、模块、URLs 和术语的第一次使用

☐☐☐Constant width☐

用于代码实例和展示命令的输出



该图标表示提醒，注意旁边的文字非常重要。



该图标表示和旁边文字相关的一个警告。

评论和问题

请把关于本书的评论和问题邮寄给出版商：

O' Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

本书有一个网页，上面罗列了勘误表、例子以及其他的一些信息。你可以访问该网页：

<http://www.oreilly.com/catalog/begperlbio>

要对本书进行评论或者询问技术性的问题，请发邮件至：

bookquestions@oreilly.com

关于书籍、会议、资源中心和 O' Reilly 网络的更多信息，请访问 O' Reilly 的网站：

<http://www.oreilly.com>

致谢

我想感谢我的编辑 Lorrie Lejeune, 以及 O' Reilly & Associates 中的每一个人, 感谢它们的技能、热情、支持和耐心; 感谢我的技术审阅人 Cynthia Gibas、Joel Greshock、Ian Korf、Andrew Martin、Jon Orwant 和 Clay Shirky, 感谢它们的帮助和细节性的审阅。我还要感谢 M. Immaculada Barrasa、Michael Caudy、Muhammad Muquit 和 Nat Torkington, 感谢他们对特定章节给出的杰出的帮助。

此外, 还要感谢 James Watson, 是他的经典教材 *The Molecular Biology of the Gene*² 首次让我对生物学产生了兴趣; 感谢 Larry Wall 发明发展了 Perl; 感谢 Murray Hill, NJ 的贝尔实验室中的同事, 是他们教授了我计算机科学。感谢 Beverly Emmanuel、David Searls 以及后来的 Chris Overton, 他们在宾夕法尼亚大学和费城儿童医院中创建了计算生物学和信息实验室, 负责人类基因组计划中的第 22 号染色体, 它们给我提供了我的第一份生物信息学工作。感谢贝尔实验室和 Upenn 的计算机与信息科学系的 Mitch Marcus, 他坚持让我借走它的 *Programming Perl*³ 并进行尝试。我还要感谢墨卡托遗传学和福克斯·蔡斯癌症中心的同事, 感谢他们对我生物信息学工作的支持。

最后, 我要感谢支持我写作的朋友们, 尤其是我的父母 Edward 和 Geraldine, 我的兄弟姐妹 Judi、John 和 Thom, 我的妻子 Elizabeth, 以及我的孩子们 Rose、Eamon 和 Joe。

²译者注: 《基因的分子生物学》。

³译者注: 《Perl 语言编程》。

目 录

第 1 章 生物学和计算机科学	1
1.1 DNA 的组成	3
1.2 蛋白质的组成	4
1.3 <i>In Silico</i>	5
1.4 计算的局限	6
第 2 章 Perl 语言入门	7
2.1 低而长的学习曲线	8
2.2 Perl 的优势	9
2.2.1 易于编程	9
2.2.2 快速原型	9
2.2.3 可移植性，速度和程序维护	9
2.2.4 Perl 的版本	10
2.3 在你的计算机中安装 Perl	11
2.3.1 也许 Perl 已经安装上了！	11
2.3.2 没有网络可用？	11
2.3.3 下载	12
2.3.4 二进制 vs. 源代码	12
2.3.5 安装	12
2.4 如何运行 Perl 程序	14
2.4.1 Unix 或 Linux	14
2.4.2 Macs	15
2.4.3 Windows	15
2.5 文本编辑器	16
2.6 寻求帮助	17
第 3 章 编程的艺术	19
3.1 学习编程的不同方法	20
3.2 编辑-运行-修正（还有保存）	21
3.2.1 保存和备份	21
3.2.2 错误信息	21
3.2.3 调试	22
3.3 编程文化	23
3.3.1 开源程序	23

3.4	编程策略	24
3.5	编程过程	26
3.5.1	构思阶段	26
3.5.2	算法	27
3.5.3	伪代码和代码	28
3.5.4	注释	29
第 4 章	序列和字符串	31
4.1	序列数据的表征	32
4.2	存储 DNA 序列的程序	35
4.2.1	控制流	35
4.2.2	再说注释	36
4.2.3	命令解释	36
4.2.4	语句	36
4.3	连接 DNA 片段	39
4.4	转录：从 DNA 到 RNA	42
4.5	使用 Perl 文档	44
4.6	在 Perl 中计算反向互补	45
4.7	蛋白质，文件和数组	48
4.8	从文件中读取蛋白质序列数据	49
4.9	数组	53
4.10	标量上下文和列表上下文	57
4.11	练习题	59
第 5 章	基序和循环	61
5.1	流程控制	62
5.1.1	条件语句	62
5.1.2	循环	65
5.2	代码布局	68
5.3	查找基序	70
5.3.1	获取用户的键盘输入	72
5.3.2	使用 join 把数组合并成标量	73
5.3.3	do-until 循环	73
5.3.4	正则表达式	73
5.4	计数核苷酸	76
5.5	把字符串拆解成数组	77
5.6	操作字符串	83
5.7	写入文件	87
5.8	练习题	91
第 6 章	子程序和 Bugs	93
6.1	子程序	94
6.1.1	子程序的优势	94
6.1.2	编写子程序	94

6.2	作用域和子程序	97
6.2.1	参数	97
6.2.2	作用域	98
6.3	命令行参数和数组	102
6.4	传递数据给子程序	105
6.4.1	子程序：通过值传递	105
6.4.2	子程序：通过引用传递	105
6.5	模块和子程序库	109
6.6	修复代码中的 Bugs	110
6.6.1	use warnings; 和 use strict;	110
6.6.2	使用注释和 Print 语句修复 Bugs	111
6.6.3	Perl 调试器	111
6.7	练习题	123
第 7 章	突变和随机化	125
7.1	随机数生成器	126
7.2	使用随机化的一个程序	127
7.2.1	为随机数生成器设置种子	129
7.2.2	控制流	129
7.2.3	造句	130
7.2.4	随机选取数组的一个元素	130
7.2.5	格式化	131
7.2.6	计算随机位置的另一种方法	131
7.3	模拟 DNA 突变的程序	133
7.3.1	伪代码设计	133
7.3.2	改进设计	136
7.3.3	组合子程序来模拟突变	137
7.3.4	程序中的一个 Bug?	141
7.4	生成随机 DNA	144
7.4.1	自下而上 vs. 自上而下	144
7.4.2	生成一系列随机 DNA 的子程序	144
7.4.3	把设计变成代码	145
7.5	分析 DNA	149
7.5.1	关于代码的一些注释	154
7.6	练习题	155
第 8 章	遗传密码	157
8.1	散列	158
8.2	生物学的数据结构和算法	160
8.2.1	基因表达数据库	160
8.2.2	使用未排序数组的基因表达数据	160
8.2.3	使用排序数组和折半查找的基因表达数据	161
8.2.4	使用散列的基因表达数据	162

8.2.5	关系数据库	163
8.2.6	DBM	164
8.3	遗传密码	165
8.3.1	背景	165
8.3.2	把密码子翻译成氨基酸	165
8.3.3	遗传密码的冗余性	168
8.3.4	使用散列表示遗传密码	169
8.4	把 DNA 翻译成蛋白质	173
8.5	从文件中读取 FASTA 格式的 DNA	176
8.5.1	FASTA 格式	176
8.5.2	读取 FASTA 文件的设计	177
8.5.3	读取 FASTA 文件的子程序	179
8.5.4	输出格式化的序列数据	181
8.5.5	读入 DNA 输出蛋白质的主程序	183
8.6	阅读框	185
8.6.1	什么是阅读框?	185
8.6.2	翻译阅读框	185
8.7	练习题	190
第 9 章	限制酶图谱和正则表达式	193
9.1	正则表达式	194
9.2	限制酶切图谱和限制性内切酶	196
9.2.1	背景	196
9.2.2	程序规划	196
9.2.3	限制性内切酶数据	197
9.2.4	逻辑操作符和范围操作符	203
9.2.5	寻找限制性内切位点	205
9.3	Perl 的操作	209
9.3.1	运算和括号的优先级	209
9.4	练习题	210
第 10 章	GenBank	211
10.1	GenBank 文件	213
10.2	GenBank 库	216
10.3	分割序列和注释	218
10.3.1	使用数组	218
10.3.2	使用标量	221
10.4	解析注释	225
10.4.1	使用数组	225
10.4.2	何时使用正则表达式	228
10.4.3	主程序	230
10.4.4	在顶层解析注释	233
10.4.5	解析 FEATURES 表	237

10.5 使用 DBM 对 GenBank 进行索引	244
10.5.1 DBM 基础	244
10.5.2 一个用于 GenBank 的 DBM 数据库	244
10.6 练习题	248
第 11 章 PDB	249
11.1 PDB 概述	250
11.2 文件和文件夹	251
11.2.1 打开目录	251
11.2.2 递归	255
11.2.3 处理大量文件	257
11.3 PDB 文件	259
11.3.1 PDB 文件格式	264
11.3.2 SEQRES	264
11.4 解析 PDB 文件	269
11.4.1 提取一级序列	273
11.4.2 查找原子坐标	274
11.5 控制其他程序	279
11.5.1 Stride 二级结构预测器	279
11.5.2 解析 Stride 的输出	282
11.6 练习题	284
第 12 章 BLAST	287
12.1 获取 BLAST	288
12.2 字符串匹配和同源	289
12.3 BLAST 输出文件	290
12.4 解析 BLAST 输出	295
12.4.1 提取注释和比对	295
12.4.2 解析 BLAST 比对	299
12.5 呈现数据	305
12.5.1 printf 函数	305
12.5.2 here 文档	306
12.5.3 format 和 write	306
12.6 BioPerl	309
12.6.1 示例模块	309
12.6.2 BioPerl 指南脚本	309
12.7 练习题	315
第 13 章 进阶主题	317
13.1 程序涉及的艺术	318
13.2 网页编程	319
13.3 算法和序列比对	320
13.4 面向对象编程	321

13.5 Perl 模块	322
13.5.1 BioPerl	322
13.6 复杂的数据结构	323
13.7 关系数据库	324
13.8 芯片和 XML	325
13.9 图形编程	326
13.10 网络建模	327
13.11 DNA 计算机	328
附录 A 资源	329
A.1 Perl	330
A.1.1 网站	330
A.1.2 CPAN (Comprehensive Perl Archive Network) : Perl 综合典藏网	330
A.1.3 FAQs (Frequently Asked Questions) : 常见问答集	330
A.1.4 在线手册	331
A.1.5 书籍	331
A.1.6 会议	332
A.1.7 新闻组	332
A.2 计算机科学	333
A.2.1 算法	333
A.2.2 软件工程	333
A.2.3 计算机科学理论	334
A.2.4 通用编程	334
A.3 Linux	335
A.4 生物信息学	336
A.4.1 书籍	336
A.4.2 政府组织	336
A.4.3 会议	336
A.5 分子生物学	337
附录 B Perl 概要	339
B.1 命令解释	340
B.2 注释	341
B.3 标量值和标量变量	342
B.3.1 字符串	342
B.3.2 数字	342
B.3.3 标量变量	343
B.4 赋值	344
B.5 语句和块	345
B.6 数组	346
B.7 散列	347
B.8 操作符	349
B.9 操作符优先级	350

B.10 基本操作符	351
B.10.1 算术操作符	351
B.10.2 位操作符	351
B.10.3 字符串操作符	352
B.10.4 文件测试操作符	352
B.11 条件和逻辑操作符	353
B.11.1 真和假	353
B.11.2 逻辑操作符	353
B.11.3 使用逻辑操作符控制流程	354
B.11.4 if 语句	355
B.12 绑定操作符	357
B.13 循环	358
B.14 输入/输出	361
B.14.1 从文件获取输入	361
B.14.2 从 STDIN 获取输入	361
B.14.3 从命令行指定的文件中获取输入	362
B.14.4 输出命令	362
B.15 正则表达式	365
B.15.1 概述	365
B.15.2 元字符	366
B.15.3 捕获匹配的模式	368
B.15.4 元符号	368
B.15.5 扩展正则表达式序列	370
B.15.6 模式修饰符	370
B.16 标量和列表上下文	371
B.17 子程序和模块	373
B.18 内置函数	376

图 片

1.1	DNA 的两条链	3
4.1	替换操作符	44
4.2	tr 语句	48
8.1	遗传密码	166
B.1	图解数组	347
B.2	图解散列	348

表 格

4.1	标准的 IUB/IUPAC 核酸代码	32
4.2	标准的氨基酸代码	33
11.1	PDB 文件中参考的数据库	265
12.1	BioPerl 模块	310
B.1	复制操作符简写	344
B.2	文件测试操作符	352
B.3	字母数字元符号	369
B.4	扩展正则表达式序列	370
B.5	模式修饰符	370
B.6	Perl 的内置函数	376

程 序

例 4.1: 把 DNA 存储到计算机中	35
例 4.2: 连接 DNA	39
例 4.3: 把 DNA 转录成 RNA	42
例 4.4: 计算 DNA 一条链的反向互补链	45
例 4.5: 从文件中读取蛋白质序列数据	49
例 4.6: 从文件中读取蛋白质序列数据, 第二次尝试	50
例 4.7: 从文件中读取蛋白质序列数据, 第三次尝试	53
例 4.8: 变量上下文和列表上下文	57
例 5.1: if-elsif-else	65
例 5.2: 从文件中读取蛋白质序列数据, 第四次尝试	66
例 5.3: 查找基序	70
例 5.4: 确定核苷酸的频率	78
例 5.5: 演示 Perl 对数字和字符串的智能化处理	81
例 5.6: 确定核苷酸频率, 第二次尝试	83
例 5.7: 确定核苷酸频率, 第三次尝试	87
例 6.1: 把 ACGT 附加到 DNA 上的子程序	95
例 6.2: 不使用 my 变量导致的陷阱	99
例 6.3: 通过命令行计算 DNA 中 G 的数目	102
例 6.4: 有一两个 bug 的程序	112
例 7.1: 使用随机数的儿童游戏	127
例 7.2: 突变 DNA	138
例 7.3: 生成随机 DNA	145
例 7.4: 计算成对随机 DNA 序列的平均一致性百分比	149
例 8.1: 把 DNA 翻译成蛋白质	173
例 8.2: 读取 FASTA 文件并提取序列数据	181
例 8.3: 读取 DNA 的 FASTA 文件, 翻译成蛋白质, 并格式化输出	183
例 8.4: 从六个阅读框上翻译 DNA 序列	187
例 9.1: 把 IUB 的模糊代码翻译成正则表达式	200
例 9.2: 解析 REBASE 数据文件的子程序	201
例 9.3: 为用户的查询制作限制酶切图谱	205
例 10.1: 从 GenBank 文件中提取注释和序列	218
例 10.2: 从 GenBank 记录中提取注释和序列	223
例 10.3: 使用数组解析 GenBank 的注释	225

例 10.4: 使用数组解析 GenBank 注释, 第二次尝试	226
例 10.5: GenBank 库的子程序	230
例 10.6: 解析 GenBank 注释	234
例 10.7: 测试解析特征的子程序	240
例 10.8: 一个 GenBank 库的 DBM 索引	245
例 11.1: 列出文件夹 (或目录) 的内容	251
例 11.2: 列出文件夹及其子文件夹的内容	253
例 11.3: 一个罗列文件系统的递归子程序	255
例 11.4: 演示 File::Find	257
例 11.5: 从 PDB 文件中提取序列链	269
例 11.6: 从 PDB 文件中提取原子坐标	275
例 11.7: 调用其他程序进行二级结构预测	279
例 12.1: 从 BLAST 输出文件中提取注释和比对	295
例 12.2: 从 BLAST 输出文件中解析比对	300
例 12.3: here 文档示例	306
例 12.4: 生出 FASTA 输出的 format 函数示例	307

第 1 章 生物学和计算机科学

目录

1.1 DNA 的组成	3
1.2 蛋白质的组成	4
1.3 <i>In Silico</i>	5
1.4 计算的局限	6

涉足计算机程序设计和生物学领域，总会发现有许多激动人心的事情，随处可见的新技术和新成果便是其中之一。

当然，生物学是一门古老的学科，但其中许多有趣的研究方向都源于新技术和新想法。现代科学中的遗传学起源于广受赞誉的孟德尔的工作¹，迄今为止仅有 100 多年的历史，但已成为现代生物学中举足轻重的一门学科。脱氧核糖核酸（DNA）和第一个蛋白质的结构是大约 50 年前解析出来的²，而克隆 DNA 的聚合酶链式反应（PCR）技术才出现了 20 多年³。人类基因组计划（Human Genome Project, HGP）旨在破译人类基因的遗传信息，刚刚过去的十年⁴见证了该计划的启动和完成⁵现在，我们正处于生物学研究的黄金年代，这是人类医学史、科学史和哲学史上至关重要的一个时代。

计算机科学是一个相对崭新的学科。算法早在古代就已经出现（欧几里得），而对计算机器的痴迷也有一定的历史了（比如，帕斯卡的机械计算器，以及 19 世纪巴贝奇的蒸汽动力计算器）。但程序设计仅仅在 50 年前才出现，和第一台大型、可编程的电子数字积分计算机 ENIAC 的建造处于同一个时代⁶。之后，程序设计便急速发展，现在仍是如此。互联网⁷和个人电脑⁸才出现了 20 多年，而万维网更是只有短短 10 年的历史⁹。今天，我们的通讯、运输、农业、金融、政府、商业、艺术都和计算机以及程序设计密不可分，科学研究亦是如此。

计算机程序设计领域的急速发展着实令人兴奋，但这也要求相关的专业从业人员要与时俱进。在某中程度上，程序设计代表了过程性知识——即如何做事的知识。计算机在我们社会和历史中的重要性，从使用计算机引起的过程性知识的海量增长中就可见一斑。同

¹译者注：孟德尔的研究论文发表于 1865 年。

²译者注：DNA 的双螺旋结构和第一个蛋白质的结构分别于 1953 年和 1958 年被解析出来。

³译者注：PCR 技术出现于 1983 年。

⁴译者注：指 1990 年到 2000 年。

⁵译者注：1990 年 HGP 正式启动，2000 年工作草图完成。

⁶译者注：ENIAC 诞生于 1946 年。

⁷译者注：1973 年 ARPA 网扩展成互联网，1983 年 ARPA 网将其网络核心协议由 NCP 改变为 TCP/IP 协议。

⁸译者注：世界公认第一部个人电脑是 1971 年 Kenbak Corporation 推出的 Kenbak-1。

⁹译者注：1991 年因特网上的万维网公共服务首次亮相。

时我们也看到计算和算法的概念已被广泛使用，比如在艺术、法律和科学中就随处可见。计算机已经成为启发人们解开事物本质的制胜法宝。有感于此，把发生分子生物学过程的细胞看成一种特殊的计算机器，这绝对是一个诱人的想法。

反之亦然，生物学中的重大发现在计算机科学中也有所体现，如进化程序、神经网络和模拟退火等。不切当的启发存在一定的危险性，这是确确实实存在的，但不可否认，生物学和计算机科学两个领域观点的交换和启发已经成为各自领域取得重大发现的推动力。

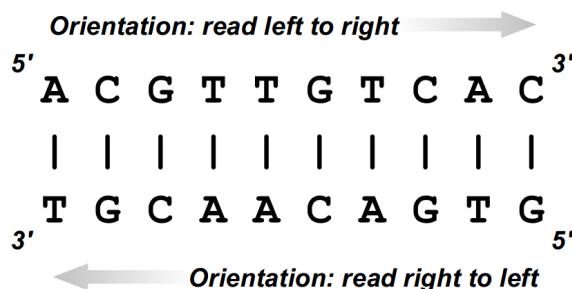


图 1.1: DNA 的两条链

1.1 DNA 的组成

为非生物学家着想，有必要回顾一下 DNA 和蛋白质的基本概念和相关术语。如果你是生物学家，可以直接跳过下面的两小节。

DNA 是由四种分子组成的聚合物。这四种分子常被叫做碱基或者核苷酸，它们是腺嘌呤、胞嘧啶、鸟嘌呤和胸腺嘧啶，单字母缩写分别为 A、C、G 和 T。¹⁰（关于 DNA 是如何表述成计算机数据的，请参看第 4 章。）这些碱基首尾相连形成 DNA 的一条单链。

在细胞中，DNA 通常以双链形式存在，两条链以著名的双螺旋形式互相缠绕在一起。双螺旋的两条链拥有互相配对的碱基，称为碱基对。一条链上的 A 总是和另一条链上的 T 相配对，而 G 则和 C 配对。

DNA 链是有方向性的。核苷酸的一端叫做 5' 端，另一端叫做 3' 端。当核苷酸连接形成 DNA 链的时候，一个核苷酸的 5' 端总是和另一个核苷酸的 3' 端相连接。此外，当细胞使用 DNA 时，如转录成 RNA 的过程，也是从 5' 端到 3' 端一个碱基一个碱基地进行。所以，当在纸上书写 DNA 时，方向是从左到右的，这和 DNA 碱基的 5' 端到 3' 端的方向是对应的。一个编码基因可能出现在两条链的任何一条上，所以当查找或分析 DNA 时一定要对两条链进行分析。

当两条链形成双螺旋时（见图 1.1），它们的方向是相反的，也就是说，一条链的方向是 5' 端到 3' 端，而另一条链的方向则是 3' 端到 5' 端。所以，观察双螺旋的任何一个末端，都是一条链的 3' 端和另一条链的 5' 端。

因为碱基对都是 A-T 配对和 C-G 配对，同时两条链的方向又是相反的，因此常用反向互补这个术语来描述两条链上碱基之间的关系。说“反向”是因为两条链的方向相反，说“互补”是因为某个碱基总是和它的互补碱基相配对——A 和 T 配对、C 和 G 配对。

基于以上事实，对于任何一条 DNA 单链，很容易就可以得到双螺旋中对应的另一条链。只需要简单得把碱基换成它们的互补碱基：A 换成 T、T 换成 A、C 换成 G、G 换成 C。因为 DNA 的书写方向是从 5' 端到 3' 端，所以在对 DNA 互补之后，还需要把它反向写出来才行。

GenBank(the Genetic Sequence Data Bank)数据库(<http://www.ncbi.nlm.nih.gov>)存储着绝大多数已知的序列数据。我们将会在第 10 章对 GenBank 进行详细介绍。

¹⁰它们因最早被发现的地方而得名：腺体、细胞、鸟粪和胸腺

1.2 蛋白质的组成

蛋白质和 DNA 类似，它们也是聚合物，由数量不等的简单分子组成一个长串。DNA 是由四种核苷酸组成的，而蛋白质则是由 20 中氨基酸构成的。这些氨基酸出现的顺序并不固定，它们的名字和单字母、三字母缩写都罗列在了表 4.2 中。

氨基酸由氨基基团和羧基基团构成。相邻氨基酸的氨基基团和羧基基团会形成化学键，叫做肽键。氨基酸都有从骨干上突出出来的侧链，20 种氨基酸的侧链各不相同。侧链的化学属性在决定蛋白质属性方面起着关键作用。

蛋白质通常有比 DNA 更加复杂的 3D 结构。肽键有相当大的旋转自由度，从而允许蛋白质形成多种 3D 结构。和 DNA 的双螺旋结构不同，蛋白质可以由一条或多条由氨基酸组装成的肽链构成，并且可以折叠成各种不同的形状。¹¹蛋白质的氨基酸序列叫做蛋白质的一级结构，一级结构卷曲形成的 α -螺旋、 β -折叠和转角等局部结构叫做二级结构，最终的折叠和组装叫做蛋白质的三级结构和四级结构（参看第 11 章）。

可利用的蛋白质一级序列数据要比二级结构或高级结构的数据多。事实上，有大量的蛋白质一级序列数据可供使用（因为大量的 DNA 已被测序，从 DNA 中识别出蛋白质的一节序列并不是非常困难）。

PDB (Protein Data Bank) 数据库储存着成千上万个蛋白质的结构信息，这些信息都是最近几十年的工作成果积累起来的。我们将在第 10 章详细介绍 PDB，当然你也可以先去 PDB 的网站 (<http://www.rcsb.org/pdb/>) 逛逛，熟悉一下这个最为基本的生物信息学资源。

¹¹为了集中精力学习 Perl 语言，我尽量避开绝大部分令人迷惑的生物学知识。但我禁不住还是要提醒一句，其实 DNA 也有更加复杂的 3D 结构。DNA 可以以单链、双链和三链形式存在，并且在细胞周期的大部分时间内它都卷曲压缩在一个很小的空间内。

1.3 *In Silico*

最近, *in silico* 这个新词已经成为指代在计算机中进行生物学的专用术语, 和 *in vivo*、*in vitro* 这两个传统术语一起来描述进行实验研究的位置。

对于非生物学家来说, *in vitro* 表示“在玻璃中”, 换言之, 在实验试管中; *in vivo* 则表示“在生命中”, 换言之, 在生物活体内。*in silico* 这个术语的由来, 源于大多数计算机芯片主要是由硅构成的这个事实。就我个人而言, 我更喜欢 *in algorithmo* 这样的术语, 毕竟有许多不需要基于硅的计算方法, 像 DNA 计算、量子计算、光学计算等奇妙的计算过程。

可以在线获取的大量生物学数据, 使得生物学研究处于和物理学、天文学相类似的境地。在这些学科中, 基于现代仪器的实验会产出海量的数据, 为了分析处理这些数据, 计算机不仅是无价的, 而且是必需的。事实上, 在计算机中对实验进行完全模拟是非常有可能的。比如, 物理学中用计算机进行模拟的一个早期应用, 就是对音乐厅的声学进行建模, 通过改变音乐厅的设计来研究最终的声学效果, 这显然比通过建造数十个音乐厅来检验声学效果要廉价的多。

自从计算机问世以来, 在生物学中也发生着类似的转变, 随着 HGP 和对多种生物 DNA 测序的进行, 这种趋势在近几年急剧加大。需要收集、检索和分析的实验数据对单个生物学家来说实在是太过巨大了, 这就迫使生物学家使用计算机来处理这些信息。

除了生物学数据的存储和提取, 现在通过计算机模拟来研究生命系统也已经成为可能。获取人类和其他一些生物的基因, 这已经成为计算机标准且常规的处理。当确定 DNA 的序列后, 可以把它们存储在计算机中, 然后写程序来识别限制酶切位点、进行限制酶切消化、制作限制酶图谱 (参看第 9 章)。与之类似, 基因识别程序可以从测序的 DNA 中识别出潜在的外显子和内含子。(到本书撰写之际, 并不是很精确, 不同生物体的识别结果差异很大。) 也可以对细胞过程进行建模, 利用该模型来研究如基因调控变化的影响等生物学过程。

现在, 利用芯片技术 (玻璃片上点上数千个样本, 利用仪器进行检测) 仅仅通过一次实验就可以获得成千上万个基因的表达水平。借助计算机来解析基因之间复杂的相互作用。比如, 我们期望能够找到所有相关的基因, 在细胞中它们通过蛋白质产物参与相同的生化途径。芯片会产出海量的数据, 和其他实验数据不同, 这些数据需要在计算机中进行存储和分析。

在我进入贝尔实验室成为程序员的第一天, 我导师就告诉我, 他的模拟计算速度太快了——仅需过夜就可完成, 但这却让他头疼, 因为他没有足够的时间来分析上一次的模拟了! 尽管计算机有着各种令人头疼的问题和陷阱, 但使用计算机来模拟实验确实给生物学带来了极大的益处。

1.4 计算的局限

计算机科学中一些非常有趣的结果证明了人类知识的局限性。在生物学中有一些开放性的问题，有人认为利用更多的计算资源就可以解决这些问题，但这并不总是对的，因为有些问题本身就是无解的，换句话说，它们不可能被任何程序解决。此外，有些问题也许是可以解决的，但是随着问题量的增长，在实际中是没法解决这些问题的。这样的问题是不治的，称为 NP 完全。即使使用百万台计算机，每台的计算能力都是现在最强大计算机的一百万倍，也有可能需要十亿年才能计算出一个 NP 完全问题的答案。

你有可能会遇到 NP 完全问题，但这是非常罕见的，你的选择就是别去招惹这种无法解决的问题。我提及它们更多的是因为有趣，而不是因为菜鸟程序员在实际中会遇到这样的问题。但随着你在程序员的路上越走越远，尝试更多复杂程序的时候，这些局限，尤其是某些生物学问题的不治本性，会对你的编程产生实质性的影响。

第 2 章 Perl 语言入门

目录

2.1 低而长的学习曲线	8
2.2 Perl 的优势	9
2.3 在你的计算机中安装 Perl	11
2.4 如何运行 Perl 程序	14
2.5 文本编辑器	16
2.6 寻求帮助	17

Perl 语言是一种流行的编程语言，在生物信息学和网络编程中广泛应用。Perl 之所以能被生物学家广泛使用，是因为它特别适合用来解决生物信息学问题。

Perl 也是一个应用程序，就像你在计算机中安装的其他应用程序一样。对于大多数生物学实验室使用的各种操作系统来说，Perl（完全免费）都存在其中并时刻运行着。¹ 计算机中的 Perl 应用程序²读取 Perl 语言程序（比如你将在本书学习过程中写的任何一个程序）³，把它翻译成计算机可以理解的指令，并运行（或“执行”）它。

所以，Perl 这个词既指你用来编写脚本的程序语言，也表示计算机中运行这些脚本的应用程序/解释器。⁴你可以通过上下文的语境来区分 Perl 的具体含义。

所有的计算机语言，包括 Perl，都需要一个翻译器应用程序（包括解释器和编译器两种）来把程序翻译成计算机可以实际运行的指令。Perl 的翻译器应用程序通常指的就是 Perl 解释器，当然它也包括 Perl 编译器。你会发现一般把 Perl 程序叫做 Perl 脚本或者 Perl 源码。程序、应用程序、脚本和可执行文件这四个术语在某种程度上是通用的，在本书中我都把它们叫做“程序”。

¹操作系统管理者程序的运行和其他一些计算机提供的基本服务，如文件存储等。

²译者注：此处指的是实际编译并运行程序的解释器，一般使用 perl（小写 p）来表示。

³译者注：此处指 Perl 脚本。指代程序语言时一般用 Perl（大写 P）来表示。

⁴译者注：一般情况下，前者使用 Perl 表示，后者使用 perl 表示。

2.1 低而长的学习曲线

对于 Perl 来说，令人高兴的一点就是你很快就可以学会 Perl 程序的编写。大体上来看，Perl 的学习曲线是比较低的。这也就是说你可以轻松入门，不需要学习掌握太多的知识就可以写出非常实用的程序。

Perl 提供了不同的编程范式，这已经超出了本书的范畴，所以我并不打算对此进行深入探讨，但我还是要说一句，最流行的编程范式叫做命令式编程，这也是你将在本书中学习到的编程范式。另一种比较流行的编程范式叫做面向对象编程，Perl 对这种编程范式也有很好的支持。其他编程范式还包括函数式编程和逻辑编程。

尽管你可以轻松入门，但如果想学习 Perl 中的所有知识，那将花费你大量的精力。大多数人都只学习像本书中的知识点一样的基本知识，当需要时再学习其他的相关主题。

在继续进行之前，我们先讨论几个基本概念：

什么是计算机程序？

计算机程序就是用特定编程语言写成的一系列指令，这些指令可以被计算机读取。一个程序可以非常简单，简单到就像下面这个把 DNA 序列打印到计算机屏幕上的 Perl 语言程序一样：

```
1 | print 'ACCTGGTAACCCGGAGATTCCAGCT';
```

就像在计算机中存储各种数据（不限于程序）一样，要在文件中编写和保存 Perl 语言程序。文件以成组分层的形式进行存储，这些不同的组在 Macintosh 和 Windows 系统中叫做文件夹，在 Unix 和 Linux 系统中叫做目录，其实两者是完全可以互换使用的。

什么是编程语言？

有一系列详细定义的语法规则来指导计算机程序的编写。通过学习编程语言的这些语法规则，你可以编写出在计算机上运行的程序。编程语言和英语等我们日常使用的口语是非常相似的，只是针对计算机系统进行了严格而特定的定义。经过一定的训练，就可以轻松阅读并编写计算机程序。有许多不同的编程语言，在本书中你将学习 Perl 语言。

程序员编写的程序也叫做源代码，有时也简称为源或代码。源代码需要翻译成机器语言，计算机才能够运行它。机器语言都是二进制数字，非常难于阅读和编写，通常把它叫做二进制可执行程序。就像在本章中你将看到的那样，使用 Perl 解释器（或编译器），你可以把 Perl 程序变成可以运行的程序。

什么是计算机？

哦，……

好吧，这是个愚蠢的问题。计算机就是你在计算机商城里买到的机器。但实际上，你应该有一个清晰的认识，明白计算机到底是怎样的一台机器。本质上来说，计算机就是一台可以运行不同程序的机器。正是这种最基本的灵活性和适用性，使得计算机成为如此有用且通用的工具。它是可编程的，你将学习如何使用 Perl 编程语言来对它进行编程。

2.2 Perl 的优势

接下来的几个小节将对 Perl 的几个优点进行说明。

2.2.1 易于编程

编程语言的易用性体现在不同的方面。我说“容易”指的是对于程序员来说易于编程。Perl 的一些特性，使得解决常见的生物信息学问题非常容易。它可以处理 ASCII 文本文件或平面文件中的信息，而许多重要的生物学数据就是存储在这样的文件中，GenBank 和 PDB 等数据库就是如此。（关于 ASCII 的讨论请参看第 4 章；GenBank 和 PDB 分别是第 10 章和第 11 章的讨论主题。）使用 Perl，可以容易得处理 DNA 和蛋白质的长序列，方便得控制一个或多个其他程序。最后再举一个例子，Perl 曾用来把生物学研究实验室概况和研究成果放在实验室的动态网站上。Perl 可以胜任所有的这些任务，当然绝不止这些。

尽管 Perl 是一种非常适用于生物信息学的编程语言，但它绝不是唯一的选择，也不一定总是最好的选择。其他编程语言，如 C 和 Java，也在生物信息学中使用。如何选择一种编程语言，这取决于需要解决的问题、程序员的能力、可利用的系统等因素。

2.2.2 快速原型

在生物学研究中使用 Perl 的另一个非常重要的优势就是，程序员可以非常迅速地写出一个典型的 Perl 程序（这被称为快速原型）。与 C 和 Java 相比，许多问题用很少的几行 Perl 代码就可以解决，这也直接导致了它在生物学研究中的流行。在一个研究中，经常需要写一些处理新问题的程序，这些程序仅会偶尔使用或使用一次，或者需要被频繁修改。使用 Perl，你只需折腾几分钟或者几个小时就可以写出这样的程序，继续后续的研究工作。当在工作中选择使用 Perl 时，这种快速原型的能力常常是需要重点考虑的因素。常常可以遇到同时精通 Perl 和 C 的程序员，他们声称在编程时 Perl 要比 C 快五到十倍。这种差距在人手不足的研究实验室中尤其显著。

2.2.3 可移植性，速度和程序维护

可移植性表示编程语言可以在各种不同的计算机操作系统上运行。Perl 的可移植性就不错，在生物学实验室中使用的所有现代计算机中，它都可以使用。如果你在自己的 Mac 中用 Perl 编写一个分析 DNA 的程序，然后把它放到 Windows 计算机中，你会发现它通常可以正常运行，或者仅需很小的修改就可运行。

速度表示程序运行的速度。在这方面，Perl 虽然不是最好的，但表现也不错。考虑到运行的速度，最常见的选择就是 C。用 C 编写的程序，其运行速度通常比对应的 Perl 程序快两倍甚至更多。（有许多利用编译器等来提高 Perl 运行速度的方法，但依然……。）

在许多组织中，都是先用 Perl 来编写程序，然后再把那些确实需要提高运行速度的部分用 C 重写。事实上，运行速度只是偶尔才会成为需要考虑的关键因素。

编程相对是比较昂贵的，因为它需要时间和熟练的人员，它是劳动密集型的。另一方面，计算机和计算机时间（在有了中央处理单元之后通常叫做 CPU 时间）却是相对便宜的。无论如何，在一天内的大部分时间里，大多数桌面计算机都出于空闲状态。所以最好的策略就是节省程序员的时间，让计算机工作。一般来说使用 Perl 就可以了，除非你的程序必须运行四秒而非十秒。

程序维护是保证程序正常运行的常规工作，比如给程序添加特性、扩展程序以便能够处理更多类型的输入、移植程序以便在其他计算机系统上运行、修复 bug 等工作。编写程序需要花费一定的时间、精力和财力，而好的程序在维护阶段的花费远比第一次编写时的花费要高。编写程序时使用一种语言和特定的风格是非常重要的，因为这将使得维护工作相对容易一些，Perl 就是这样的编程语言。（就像其他编程语言一样，你可以用 Perl 写出晦涩难懂、难于维护的代码，但我会给你一些指导，使你的程序可以被其他程序员轻松读懂。）

2.2.4 Perl 的版本

向其他所有流行软件一样，Perl 也在不断成长，在近 15 年的历史中不断发展变化⁵。Perl 的作者拉里·沃尔和庞大的开发社区会定期发布新的版本。这些新的版本都进行了认真的设计，以便向下兼容旧版本写成的大部分程序，但是偶尔也会添加一些新的特性，这些特性在旧版本的 Perl 中无法正常工作。

本书假设你安装了 Perl 5 或者更高的版本。如果你的计算机中安装了 Perl，那很有可能是 Perl 5，但最好去确认一下。在 Unix 或者 Linux 系统中，以及 MS-DOS 或者 MacOS X 的命令行窗口中，命令 `perl -v` 会显示 Perl 的版本号。在我的计算机中，使用的是 Perl 5.6.1⁶。数字 5.6.1 比 5 “大”，也就是说它满足要求。如何计算机中 Perl 的版本号小（很有可能是 4.036），你最好安装一个最新版本的 Perl，这样本书中的大部分程序才能正常运行。

未来的版本会如何呢？Perl 总实在进化，Perl 6 已经近在眼前了⁷。本书中的代码在 Perl 6 中能否依然正常工作？答案是肯定的。尽管 Perl 6 要添加一些新的东西进去，运行本书中的 Perl 5 代码不会存在太大的问题。

⁵译者注：Perl 诞生于 1987 年。

⁶译者注：现在最新的版本是 5.22.0（2015-06-01）。

⁷译者注：Perl 6 正在开发中，它将会与现在的 Perl 版本有很大的不同，但相信还要开发一段很长的时间。

2.3 在你的计算机中安装 Perl

接下来的几个小节将指导你在最常见的计算机系统中安装 Perl。

2.3.1 也许 Perl 已经安装上了！

许多计算机，尤其是 Unix 和 Linux 计算机，已经预装了 Perl。（注意 Unix 和 Linux 本质上是一样的操作系统；Linux 是 Unix 操作系统的克隆，或者说是功能性的拷贝。）所以，首先确认一下 Perl 是不是已经被安装上了。在 Unix 和 Linux 系统中，在命令提示符后面输入一下命令：

```
1 | $ perl -v
```

如果 Perl 已经安装上了，你会看到和我自己 Linux 计算机上类似的信息：

```
1 | This is perl, v5.6.1 built for i686-linux
2 |
3 | Copyright 1987-2001, Larry Wall
4 |
5 | Perl may be copied only under the terms of either the Artistic License or the
6 | GNU General Public License, which may be found in the Perl 5 source kit.
7 |
8 | Complete documentation for Perl, including FAQ lists, should be found on
9 | this system using 'man perl' or 'perldoc perl'. If you have access to the
10 | Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

如果没有安装 Perl，你会看到如下类似的信息：

```
1 | perl: command not found
```

如果你看到这样的信息，并且你使用的是大学或公司的公用 Unix 系统，一定要向系统管理员进行确认，也许 Perl 已经安装上了，但是你的环境设置导致你无法找到它。（或者，系统管理员可能会说：“你要用 Perl？没问题，我来给你安装。”）

在 Windows 或者 Macintosh 中，可以在程序目录中或者使用查找程序查找 *perl*。在 MS-DOS 的命令窗口或者 MacOS X 的 shell 窗口中，你同样可以尝试输入 `perl -v`。（注意 MacOS X 其实是 Unix 系统！）

2.3.2 没有网络可用？

如果你没有网络可用，你可以把计算机给你可以联网的朋友，让他帮忙安装 Perl。你也可以利用朋友的计算机，通过 Zip 磁盘片或者刻录一张光盘把 Perl 软件复制到你的计算机中。通过各种途径（咨询当地的软件商店），你也可以获得商业的含有 Perl 的包装光盘，此外，像 O'Reilly 出版的《Perl Resource Kits》等书籍中也会包含 Perl 的光盘。

除了安装 Perl，本书中的所有内容都不要联网。在上下班坐地铁的时候，或者其他类似的情况，如果你想做做练习，也未尝不可。除了安装 Perl，本书中需要联网的内容主要包括：从书籍网站上下载示例代码了，这样你就不用一行一行输入计算机了；下载并练习

测试题；从各种生物学数据库中查找生物学数据；在计算机中没有安装 Perl 文档的情况下联网查阅 Perl 文档。

要知道，如果你想做生物信息，因特网是必不可少的。你可以在没有联网的情况下通过本书学习编程的基本原理，但是你需要联网才能下载生物信息学软件和数据。

2.3.3 下载

Perl 是一个应用程序，所以在你的计算机中下载和安装 Perl 的过程和安装其他应用程序是完全一样的。

有一个网站是学习 Perl 相关知识的起点，它就是 <http://www.perl.com/>。网站主页上有一个下载点击按钮，通过它你可以找到在计算机上安装 Perl 所需要的所有东西。在下载页面，有获取帮助的连接和其他相关链接。所以即使本书中的内容过时了，你仍然可以访问 Perl 站点，找到安装 Perl 所需要的相关内容。

下载和安装 Perl 通常都比较容易，事实上，大部分时间都还是比较惬意的。然而，有时为了让它能够正常工作，你还是需要付出一些努力的。如果你是编程方面的新手，并且碰到了一些棘手的问题，你最好向实验室中使用 Perl 的专业程序员、管理员、教师等人寻求帮助。

简言之，在计算机中安装 Perl 的基本步骤如下所示：

1. 确认一下 Perl 是否已经安装上了；如果已经安装了，确认一下 Perl 的版本不低于 Perl 5。
2. 连接网络，打开 Perl 官网的主页 <http://www.perl.com/>。
3. 进入下载页面，确定你要下载的 Perl 发行版。
4. 下载正确的 Perl 发行版。
5. 在你的计算机中安装此发行版。

2.3.4 二进制 vs. 源代码

当从站点 <http://www.perl.com> 中下载 Perl 的时候，你需要在二进制发行版和源代码发行版中进行选择。在计算机中安装 Perl 的最好选择就是使用已经编译好的二进制版本，因为它最容易安装。然而，如果没有二进制版本可用，或者你想配置安装 Perl 时的各种选项，就可以下载 Perl 的源代码。源代码本身是用 C 语言编写的，你可以使用 C 编译器对它进行编译。对于新手来说，从源代码进行编译实在是太过复杂了，所以还是尽量寻找适合计算机操作系统的二进制版本吧。

2.3.5 安装

接下来的几个小节是在特定平台上安装 Perl 的操作指南。

Unix 和 Linux

如果你的 Unix 或者 Linux 计算机中没有安装 Perl，首先去寻找二进制包进行安装。在 <http://www.perl.com> 站点的下载页面上，你会看到二进制发行版的副标题。选择 Unix 或 Linux，看看是否有适用于你的操作系统的二进制版本。有许多版本可以下载安装，一旦你下载了二进制版本，根据网站上的指南就可以把 Perl 安装上了。许多 Linux 发行版

都在它们的网站上维护着最新的 Perl 二进制版本。比如，如果你使用的是红帽 Linux 系统，你需要首先确认系统的版本（使用 `uname -a`），然后找到对应的 rpm 文件，下载之后进行安装。红帽 Linux 系统有 Perl 的 rpm 文件，用户可以使用以下命令进行安装：

```
1 | rpm -Uvh perl.rpm
```

（perl.rpm 文件的实际名称可能有所变化）。

如果没有适用于你使用的 Unix 或 Linux 的 Perl 二进制版本，你就必须从源代码进行编译了。这种情况下，还是从 Perl 网站开始，点击下载按钮，选择源代码发行版。源代码中有一个 *INSTALL* 文件，它是指导你进行操作的指南，从下载源代码、在系统中进行安装、从源代码编译成二进制可执行文件，到最后安装二进制可执行文件，每一个步骤都有详细的说明。

前面已经提到，相比安装已经编译好的二进制版本，从源代码进行编译是一个漫长的过程，并且需要仔细阅读操作指南，但通常这还是值得的。为了从源代码安装 Perl，你需要一个 C 编译器。如今，某些 Unix 系统中并没有一个完整的 C 编译器，而在 Linux 中通常都安装有一个叫做 *gcc* 的免费的 C 编译器。当然，你也可以在任何缺少 C 编译器的 Unix（或 Windows，或 Mac）系统中安装 *gcc*。

Macintosh

MacPerl 网站 <http://www.macperl.com/> 上（你可以从 Perl 网站上点击下载按钮进入该网站）清晰得解释了 MacPerl 的安装步骤，此处是一个简要的概述。

在 MacPerl 网站上，点击获取 MacPerl，按照指引即可下载该应用程序。它会下载到你的桌面上，双击即可进行安装。如果你没有 Aladdin Stuffit Expander（大多数 Mac 都有），双击它就不会有什么效果，所以你需要去网站 <http://www.aladdinsys.com> 下载并安装 Stuffit。

在 MacOS Finder 中 MacPerl 作为一个独立的应用程序进行安装，而在 Macintosh 程序员工作台则作为一个工具进行安装。你想要的很可能是一个独立的应用程序。Perl 5 可以适用于 MacOS 7.0 及其后续版本。至于哪个 Perl 版本适用于你自己的硬件和 MacOS 版本，在 MacPerl 网站上可以找到详细的说明。

Windows

对于不同的 Windows 版本，有好多二进制发行版可供选用。因为 Windows 是和英特尔 32 位芯片密切相关的，所以这些二进制发行版通常叫做 Wintl 或者 Win32 二进制包。现在标准的 Perl 发行版是来自 ActiveState 的 ActivePerl，网站是 <http://www.activestate.com/ActivePerl/> 在上面你可以找到完整的安装说明。你也可以在 Perl 网站上通过下载按钮找到 ActivePerl，在二进制发行版副标题下，找到 Perl for Win32，点击 ActivePerl 站点链接即可。

在 ActiveState 网站的 ActivePerl 页面中，点击下载按钮，你将下载 Windows-Intel 二进制包。注意安装它需要一个叫做 Windows Installer 的程序，如果你的计算机中没有安装，可以在 ActivePerl 中找到它。

2.4 如何运行 Perl 程序

如何运行 Perl 的详细步骤，取决于你的操作系统。你安装 Perl 时参考的操作指南中包含了你需要知道的所有相关内容，此处我仅进行简短的总结，这对于起步来说足够了。

2.4.1 Unix 或 Linux

在 Unix 或 Linux 中，你通常会在命令行中运行 Perl 程序。如果你位于程序所在的目录中，假设这个 Perl 程序文件叫做 *this_program*，你通过输入 `perl this_program` 就可以运行它。如果你在另外的目录中，就必须给出程序的路径名，比如：

```
1 | perl /usr/local/bin/this_program
```

因为不同的计算机可能会把 Perl 安装在不同的目录中，所以通常情况下，你会在 *this_program* 文件的第一行指定系统中 Perl 的正确路径。在我的计算机中，我的 Perl 程序中的第一行是：

```
1 | #!/usr/bin/perl
```

你可以通过键入 `which perl` 来找到你系统中 Perl 的安装路径。
使用 `chmod` 命令，你可以赋予程序可执行权限。比如，你可以键入：

```
1 | chmod 755 this_program
```

如果你的第一行设置正确，并且也使用了 `chmod` 命令，你可以直接键入 Perl 程序的文件名来运行它。所以，如果你在程序所在的目录中，你可以键入 `./this_program`；如果程序所在的目录包含在 `$PATH` 或者 `$path` 环境变量中，你可以键入 `this_program`。⁸

如果你的 Perl 程序没有正常运行，命令窗口中的 shell 会给出错误信息，这些错误信息也许会让你晕头转向。比如，我的 Linux 系统中的 `bash` 给出了如下所示的错误信息：

```
1 | bash: ./my_program: No such file or directory
```

出现该错误信息有两种可能：在当前目录中确实没有叫做 *my_program* 的程序，或者 *my_program* 的第一行没有给出 Perl 的正确路径。你需要好好检查一下，尤其是当运行来源于 CPAN（参看附录 A）的程序时，这些程序第一行中 Perl 的路径往往都不相同。另外，如果你键入 `my_program`，可能会得到这样的错误信息：

```
1 | bash: my_program: command not found
```

这表示操作系统找不到该程序。但它就在当前目录中呀！问题可能出在你的 `$PATH` 或 `$path` 环境变量并不包含当前目录，所以系统根本就没在当前路径中寻找该程序。这种情况下，根据使用的 shell 不同，你需要修改 `$PATH` 或 `$path` 环境变量，或者键入 `./my_program` 而非 `my_program`。

⁸`$PATH` 是 `sh`、`bash` 和 `ksh` 等 shell 的环境变量；而 `$path` 则适用于 `csh` 和 `tcsh`。

2.4.2 Macs

在 Mac 系统中，保存 Perl 程序时，推荐使用的方式是作为“droplets”；MacPerl 文档中给出了简要的说明。基本上你只需要用 MacPerl 应用程序打开 Perl 程序，另存为时选择 Droplet 类型选项即可。

你可以通过拖放文件到 droplet 上来把它作为输入文件（通过 @ARGV 数组，参看第 6 章的讨论）。

最新的 MacOS X 其实是一个 Unix 系统，因此你也可以通过命令行来运行 Perl 程序，就像前述在 Unix 和 Linux 系统中的操作那样。

2.4.3 Windows

在 Windows 系统中，Perl 程序通常和 *.pl* 文件名后缀相关联。在 Perl 安装过程中，通过修改注册表的设置实现了这种关联。有了这种关联，在 MS-DOS 命令窗口中，你就可以键入 `this_program` 或者 `perl this_program.pl` 来运行 *this_program.pl* 程序。Windows 中有一个 PATH 变量，存储着系统从中寻找程序的文件夹，在 Perl 安装过程中，该变量被修改以便收录包含 Perl 应用程序的文件夹的路径，该路径通常为 `c:\perl`。如果你想运行一个 Perl 程序，该程序所在的文件夹并为被 PATH 变量所收录，你可以键入该程序的完整路径，如 `perl c:\windows\desktop\my_program.pl`。

2.5 文本编辑器

既然已经设置好了计算机，也安装上了 Perl，接下来就需要选择一个文本编辑器并学习其基本知识了。文本编辑器用来键入程序等文档，并把文档内容保存到文件中。所以要编写一个 Perl 程序，你需要使用一个文本编辑器。尽管某些文本编辑器学习起来比较容易，但如果以前你从未使用过文本编辑器，学习起来工作量也是不小的。下面是一些最为流行的编辑器，此处依然按照操作系统的类型进行介绍：

Unix 或 Linux

vi 和 Emacs 都是比较复杂（但是很棒）的编辑器。pico、xedit 和其他一些编辑器（nedit、gedit、kedit）非常容易使用，学习起来也不难，但功能并不是很强大。在 StarOffice⁹中包含了一个和 Microsoft Word 相兼容的编辑器，而且是免费的（但一定要确保把文件保存为 ASCII 或者纯文本）。

Macintosh

MacPerl 内置的编辑器就很不错。有一个很好的收费编辑器叫做 BBEdit，它对 Perl 进行了优化，相对应的免费版本叫做 BBEdit Lite。你也可以使用共享编辑器 Alpha 或 Microsoft Word（确保把文件保存为 ASCII 纯文本）。

Windows

Notepad 已经比较流行了，用起来也令人满意；Microsoft Word 也是可以使用的，但一定要保存为 ASCII 或纯文本。在基于 Windows 的计算机中进行 Perl 编程，强烈推荐使用 Emacs 的 Windows 版本，虽然学习起来有些复杂。当然，还有好多其他的编辑器，我就使用 Unix 编辑器 vi 的一个免费版本——移植到 Windows 中的 Vim。

还有许多其他的文本编辑器可以使用。许多计算机上都有多种编辑器可供选择。（有些程序员在他们的职业生涯中会试图编写一个编辑器，或者对已有的编辑器进行某些功能上的扩展，所以可供选择的编辑器真的是非常众多的。）

有些编辑器非常容易学习和使用；而有些编辑器则有众多的特性、相应的指南书籍、讨论组和网站等，学习它们还是要花费一定精力的。如果你是一个程序员菜鸟，不要自找麻烦，选择一个易用的编辑器吧。日后，如果你喜欢冒险，可以进一步学习使用一个“神”级的编辑器，它的某些特性会让你事半功倍。不知道在你的计算机中有哪些可用的编辑器？可以向程序员或其他用户咨询一下，或者去查阅计算机系统中附带的文档。

⁹译者注：现在比较流行的是 OpenOffice 和 LibreOffice。

2.6 寻求帮助

确保你有必需的文档。如果你向前述那样安装了 Perl，作为 Perl 默认安装的一部分，文档已经安装到了计算机中，而 Perl 发行版中附带的指南解释了如何去获取这些文档。还有一个很棒的在线文档，可以在 Perl 网站主页上找到它。

各种编程资源是寻找编程问题答案的好地方，而 Perl 的资源对使用 Perl 编程来说是必需的。去附录 A 看一下吧，从中你可以找到书籍、在线文档、可用的程序、新闻组、存档、期刊和会议等各种资源。

既然你已经开始编程，你将接触到最重要的书籍、网站、网络新闻组及其可搜索的存档、当地的大师（附近该方面的专家）以及程序文档。这些文档包括程序手册（打印版或在线版）和常见问答集（FAQs）。

多数编程语言都有一个标准的文档集，包括语言定义和使用的完整说明。Perl 的该文档集作为在线手册包含在了程序中。尽管编程手册通常写的不咋地，但最好是做好不得不面并“生啃”它们的准备。视若无睹、充耳不闻的能力有时候也是一种宝贵的财富。Perl 的手册并不是很糟糕，它最大的问题就是，像其他语言的手册一样，把所有的细节都放在了里面，所以刚开始可能会有种卷帙浩繁的感觉。不过，教程文档可以帮助初学者对 Perl 文档进行梳理，这绝对是初学者的福音。

最后，我极力主张程序员菜鸟去找一些经验丰富的 Perl 程序员，他们可以帮你解决一些偶尔遇到的问题。他可以是课程的教师或助教、同事，或者是沦落到当地计算机商店的某个人、在在线新闻组（有专门针对 Perl 初学者的新闻组）中回答你提问的那个人。在你的初学阶段，偶然和一个经验丰富的用户进行的交谈，会使你避免数个小时钻牛角尖的苦思冥想，这是非常有可能的。大部分程序员都非常乐意向初学者伸出援手，或者提供一定的建议。在编程界，友好、学术的气氛十分盛行。

但是还有一个警告：如果有人重复提问可以在 FAQs 或其他标准文档中可以找到答案的问题，这往往会激怒那些专家们。对于这种类型的问题，你有时会看到 RTFM——Read The F(ine) Manual¹⁰——的答复。所以在重复提问之前，一定要去 FAQs 中查看一下是否已有解答，以避免浪费别人宝贵的时间。

（我忍不住要提起一件偶然发生的轶事。）在我学习编程时接受的第一个任务中，有一个问题令我寸步难行，而这个问题看上去并没有很明显的解决方案。于是我向被公认为实验室里最好的程序员求助。我对遇到的困境详细得进行了描述，而他则耐心得聆听着。当我解释完后，他笑着向我说道：“男儿有泪不轻谈，自力更生当为先！”听到此话，我不免有些垂头丧气，丈二和尚摸不着头脑。但完全想不到的是，他给出的建议完全是在假装深沉，而他最终也向我进行了解释，并指点我找到了解决办法。

¹⁰译者注：RTFM: Read The F***ing Manual。与之类似的还有 STFW: Search The F***ing Web 等。

第 3 章 编程的艺术

目录

3.1 学习编程的不同方法	20
3.2 编辑-运行-修正（还有保存）	21
3.3 编程文化	23
3.4 编程策略	24
3.5 编程过程	26

本章将概述程序员是如何完成他们的任务的。如果你已经安装了 Perl，而且想立即编写生物信息学中的实用程序，可以跳过本章直接阅读第 4 章。

初进生物学实验室的人对各种试管仪器会有一种莫名的崇拜，与之类似，刚刚接触编程的初学者也会对程序员的世界充满了好奇，认为它们如同一个充满了怪异术语和高深技能的神秘黑盒子。为了让读者尽快融入这个大家庭，我会对重要的知识和技能进行简短的介绍，每个程序员都要学习并使用到这些内容。其中有两个是最为重要的，一个是优秀程序员在实践过程中所使用的编程策略，另一个则是如何找到在编程过程中遇到的各种问题的答案。通过一些简单的描述性实例的学习，我们将总结出程序员寻找问题解决方案的方法。附录 A 中罗列了一些非常好的 Perl 和生物信息学的资源，在解决实践过程中遇到的问题时，它们会对你有所帮助的。

3.1 学习编程的不同方法

学习编程的最佳方法是什么？答案取决于你要完成的任务。有许多引领你入门的方法，比如：

- 参加各种类型的学习班
- 阅读像本书这样的教程性的书籍
- 死啃编程手册
- 拜其他的程序员为师
- 研究你所需要的一个程序
- 尝试上述几种或所有方法，直到对编程能够驾轻就熟为止

答案还取决于你打算如何学习编程。有人喜欢参加培训班，因为培训班会把知识点整理得有条有理，同时授课老师也会解答学生的各种问题。另外一些人则可能会更加喜欢自学。

不管是那种学习编程的方法，有些东西却是共通的。如果以前你从未写过程序，下面几个小节的内容是你首先必须要了解的。

3.2 编辑-运行-修正（还有保存）

就像跳舞、弹琴、烹饪或者其他家庭中的活动一样，学习编程需要你实际动手去做，这是最重要的一点。你可以阅读源码，但如果不动手去编写、调试程序，你永远也无法真正编写出程序。

当学习 Perl 语言编程时，你要弄明白 Perl 是如何工作的，就像在接下来的几章中你将看到的那样。此外，你还要学习大量的程序实例。在接下来几章的最后有一些练习题供你练手，此时你要尝试编写自己的程序。只有这样的实战练习才能使你成长为一个真正的程序员。

为了帮助你写出第一个程序，同时弄明白它的工作原理，我会对一些内容进行简要的概述，这些在编写程序过程中都是至关重要的。

你用计算机做什么？程序员使用计算机时，大部分工作都无外乎在编辑器中编写或修正程序，然后运行程序并检查它的运行状态，再根据运行状态回去检查和修正程序，如此循环往复。对于一个程序员来说，通常一半以上的时间都用在了编辑程序上。

3.2.1 保存和备份

即使你只写了几行代码，也一定要保存它，请牢记这一点。实际上，在编辑代码的过程中，应该有定期保存程序每一个版本的概念。这样在进行了大量编辑后，即使计算机崩溃了，你也不会丢失数个小时的劳动成果。此外，要确保在另一个磁盘上对你的工作进行了备份。如果你的硬盘损坏了，上面的所有数据都会丢失。所以，对你的工作进行定期（每日）备份至关重要，可以备份到磁带、软盘、ZIP 磁盘片、另一个硬盘、光盘等其他媒介上。不管怎样，当你的磁盘损坏时，因为备份的存在，你的工作内容将不会全部丢失。

除了备份硬盘，还应记住，要对你的程序进行定期保存，保留它的每一个版本¹。这样，当你需要回退到程序的历史版本时，你可以轻松做到。

还有一点，一定要确保你的备份确实可以使用。举个例子，如果你把数据备份到了磁带上，每次备份一段时间后，都要尝试从磁带上恢复文件，来确保软件以及磁带本身都可以正常使用。以防系统崩溃，你可能要定期把程序打印出来（“白纸黑字”），这样会多一份保险。最后一点，比较好的一个策略是不要把鸡蛋放在一个篮子里，把你的备份放在一个远离计算机的地方，这样即使遇到火灾或者其他灾难，你仍有备份可用。

3.2.2 错误信息

修复错误是编写程序过程中的基本步骤。在编辑完一个程序后，接下来你要做的就是运行它，看看程序能否正常工作。多数情况下，你会发现有一些像忘记添加分号这样的拼写错误。结果可想而知，因为程序中存在语法错误，运行时你会看到系统给出的各种错误信息。这时，你将不得不仔细查看错误信息，去重新编辑程序以消除这些讨厌的错误。

有时，这些错误信息是比较隐秘的。这种错误导致的结果就是，Perl 解释器很难精确定位错误所在，它仅知道某个地方出错了，也仅此而已。所以，它会猜测问题的根源所在，在这个过程中，它可能会给出一些与错误本身无关的信息。

牢记一点，在处理错误信息时，要对后面的视而不见，只关注第一个或前两个错误信息即可。修复这两个最先出现的问题，然后尝试重新运行一次程序。错误信息通常都比

¹译者注：推荐使用 Git 等版本控制工具。

较冗长，甚至可能有数页之多。除了第一个错误，把其他的都忽略掉吧！还有一点需要注意，第一个错误信息中给出的代码行号通常都是正确的，有时它会有一行之差，但不会偏差太大。稍后，我们会练习制作错误并解读错误信息。

3.2.3 调试

也许你编写了一个程序，Perl 解释器可以正常运行它而没有任何报错。然而你却发现，程序并没有像你期望的那样工作。现在，你需要回过头来，去查阅程序代码，尝试找出问题所在。

也许你仅仅是犯了一个低级错误，比如，本应该使用减法的地方你却用了加法。也有可能你误解了文档中的内容，以错误的方式编写了程序（重新阅读文档吧）。还有可能你对要完成的任务考虑不周、计划不足（想清楚具体策略后重新编写相应部分的代码吧）。有时你并不能找到问题所在，那就四处求助吧（可以尝试在新闻组存档和 FAQs 中搜索一下，或者向同事求助）。

对于那些难于发现的错误，有一个叫做调试器的程序专门对付它们。调试器允许你一步一步运行程序，看看在运行过程中到底发生了什么。（第 6 章中对 Perl 的调试器进行了深入讲解。）

当然，还有其他工具或者技术可供选用。比如，在程序中加入 `print` 语句，把那些中间值或中间结果输出出来，这也可以帮助你检查程序。此外，也有专门的程序，在程序运行过程中它会对其进行观察，然后输出相应的报告，比如，告诉你程序的哪一部分消耗的时间最多。所有这些工具，以及其他类似的工具，对于编程来说都是必需的，一定要学会如何去使用它们。

3.3 编程文化

编程是解决问题的实践，它是一个重复、循序渐进的过程。单打独斗未尝不可，但它通常是一个群体性的活动（让很多初学者惊讶吧）。它需要你培养特定的解决问题的技能，还要求你学习一些新的工具。编程有时难以捉摸，会让人厌恶至极。另一方面，对于那些有能力的程序员来说，当成功编写出程序后，也会有极大的成就感。

计算机程序无所不是，从一无是处，到给人美学及理智上的刺激，再到催生全新的知识。它们是如此美丽！（它们也有可能是破坏性的、愚蠢的、糊涂的甚至狠毒的，毕竟是人类创造了它们。）编写程序是一个重复、循序渐进的构建过程，从开始时的只砖片瓦，到最后的高楼大厦，看到这样的过程，谁能不由衷自豪呢？对于初学者来说，编程中的这种循序渐进的过程，简直就是一步一步学习语言的过程的缩影。

自从 20 世纪中期人们开始编写并积累程序，编程文化就在不断形成。慢慢地，我们积累出了坚固的编程文化底蕴。程序会反映出其他程序对它的影响，这就是文化氛围的力量，而程序员菜鸟也会从中受益良多。

3.3.1 开源程序

编程越来越重要，随之就产生了经济价值，导致的后果便是，好多程序的源代码被隐藏了起来，以此来保存它的商业价值、提升竞争力。

然而，对于那些最好、最有用的程序的源代码来说，每个人都可以免费获取进行查阅。可以免费获取的源代码通常被称作开源。（对于开源程序代码来说，有各种各样的版权附加其上，但这些版权都允许任何人去查阅源代码。）开源运动对待程序源代码的态度，和科学家发表研究结果的态度非常类似：公布于众以供他人检查和讨论。

对于程序员菜鸟来说，这些程序的源代码非常有用，因为从中可以学习到专业程序员是如何编写代码的。开源的程序包括 Perl 解释器和大量的 Perl 代码、Linux 操作系统、Apache 网络服务器、Netscape 网页浏览器、sendmail 邮件传输代理等等。

3.4 编程策略

为了让初学者对编程过程有一个直观的认识，通过几个教学性的实例研究，让我们看看程序员高手是如何思考来解决问题的。

假设你从测序实验室中拿到了一大批 DNA 序列，现在想对其中的调控元件²进行计数。如果你是一个专业的生物信息学程序员，你会怎么做？有两种可能的解决策略：找一个现成的程序，或者自己动手写一个。

非常有可能已经有这样一个程序，正好是你需要的，可以正常运行而且是免费的。大多数时候，在网络上都可以找到你所需要的程序，从而避免了重复发明轮子的代价。这就是性价比最高的编程——最小的工作量换来了最大的成果。图书馆里呆一天，胜过实验室里忙半年，（他山之石可以攻玉）这是实验工作者的经典谚语。

时刻注意收集整理可用的程序，这是编程艺术中很重要的一部分。之后，你就可以使用合适的程序来完成自己的工作，或者对已有的程序进行修改来满足自己的需要。当然，还是要注意版权声明的，但是大部分都是免费的，尤其是针对学术和非营利机构。大部分的 Perl 模块都有版权，虽然有一定的限制，但都允许你使用并修改它。版权的细节可以在 Perl 网站上可以，也会和特定的模块附在一起。

如何找到这些已经存在且免费的很棒的程序呢？Perl 社区已经把这样的程序代码收集整理起来，放在了 Perl 综合典藏网（CPAN）上，网址是 <http://www.CPAN.org>。去随便看看吧，你会发现它是按照主题进行整理的，所以可以快速查找网络、统计或图形等主题的程序。在我们这个例子中，你需要查找 BioPerl 模块，它包含了非常多的实用的生物信息学功能模块。所谓模块就是 Perl 代码的集合，你可以在 Perl 程序中轻松地加载并使用它们。

最有用的代码就是那些模块，它把许多相关的功能整合在了一起。在编写新程序的过程中，这些模块给予我们了极大的灵活性。尽管你仍然不得不去编程，但工作量已经大大减少，只需利用它们拼凑出一个完成的程序即可。还是以寻找调控元件为例，通过搜索你可能会发现一个方便使用的模块，它包含了所有的调控元件，还附带有相应的代码——给定调控元件后可以在 DNA 序列库中进行查找！而你需要做的仅仅是把这些现成的代码整合起来，给它提供一个 DNA 库，然后只需要很少的编程你就可以完成任务了。

还有好多其他地方可以找到现成的代码。你可以使用自己最喜欢的搜索引擎在网上进行搜索，可以通过浏览生物信息学链接合集来寻找程序，也可以在我们提到的新闻组、相关专家等资源那里进行寻找。

如果你没有找到问题的突破点，并且你也明白，自己写程序需要花费大量的时间，你就会想去图书馆的文献中自己查找一下，或者在图书管理员的帮助下进行查找。你可以在 MEDLINE 数据库中查找调控元件相关的文章，这些文章中通常会刊登有作者使用的代码（一个用 Perl 语言编写的程序）。你也可以在会议记录、书籍和期刊中进行查找。会议和商品展销会都是不错的地方，可以去逛逛，在那里你可以和其他人进行交流、向他们进行提问。

多数情况下，你都会如愿以偿，尽管你也有一定的付出，但是你为自己和实验室节省了数天、数周甚至数月的时间。

但是，针对修改现成的代码还有一句警告：有时修改现成的代码可能会比从头写一个完整的程序更加困难，这取决于有多少代码需要修改。为什么会这样呢？不同的程序有不

²所谓调控元件是指细胞用来控制编码区域的 DNA 区段，它将帮助决定这段编码区域是否以及何时翻译成蛋白质。

同的作者，要弄明白程序各个部分的作用，有时候还是比较困难的。如果你都不理解程序第一部分使用的方法，修改又从何谈起？（稍后，我们会详谈如何编写可读的代码，以及在代码中添加注释的重要性。）这个因素本身在编程代价中就占了很大一部分。许多程序阅读、理解起来非常困难，因此也难以维护。而基于种种原因，测试这样的程序也会非常困难。要确保你的修改能够正常工作，有时会花费大量的时间与精力。

好吧，如果说你已经用三天的时间来寻找现成的程序，最终却一无所获，（也许有一个现成的程序，但是需要三万美元才能买到，这已经超出了你的预算；同时你身边的程序员专家忙得不可开交，也没有时间来给你写程序。）那你将不得不自己动手写一个程序了。

如何从头写一个在 DNA 序列中计算调控元件数目的程序呢？听我慢慢道来。

3.5 编程过程

现在，你被指派去写一个在 DNA 序列中计算调控元件数目的程序。如果以前从未编写过程序，那你很可能会毫无头绪。为了写出这个程序，我们先讨论一下你需要知道的内容。

我们将逐步进行，这里是对这些步骤的简要概述：

1. 确定必需的输入，比如用户提供的数据或信息。
2. 对程序进行整体构思，包括程序计算输出结果的基本方法——算法。
3. 决定结果的输出形式；比如，输出到文件，或者进行图形化展示。
4. 通过添加更多的细节改善整体构思。
5. 编写 Perl 程序代码。

对于更短或者更长的程序来说，这些步骤可能有所不同，但对于你的大多数编程来说，这就是最基本的步骤。

3.5.1 构思阶段

首先，对于程序如何工作，你需要酝酿一个计划。这是对于程序的总体构思，而且是一个关键的步骤，这样的构思通常在实际开始编写程序之前就要完成。编程常和厨房中的食谱相类比，毕竟它们都是完成某项任务的特定操作指南。比如，你要知道程序需要什么样的输入和输出。在我们这个例子中，输入的是新的 DNA 序列。接下来，你需要一个策略，就是程序如何对输入进行计算处理得到我们想要的输出。

在我们这个例子中，程序首先要从用户那里收集信息：也就是说，DNA 序列在哪里？（这样的信息可以是存储 DNA 序列文件的文件名。）程序需要用户键入数据文件的文件名，可能是在计算机屏幕上进行输入，有可能是通过网页进行提交。然后，程序需要检查一下文件是否存在（有时文件并不存在，这时有发生，比如用户拼错了文件名，此时程序要给出警示），然后打开这个文件，在进行计算处理之前读入 DNA 序列。

该步骤虽然简单，但仍值得进行一定的注解。你可以直接把 DNA 序列放在程序代码中，这样就不用去写这部分代码了。但是把程序设计成读入 DNA 序列还是非常有用的，因为当你拿到新的 DNA 序列时就不用每次都重写程序了。这种想法非常简单，甚至是显而易见的，但却非常有效。

程序用来处理的数据叫做输入。输入可以有多种来源，如文件、其他程序、运行程序的用户、网站上填写的表单、电子邮件信息等。多数程序都读入某种形式的输入，但也有程序没有输入。

让我们把调控元件添加到实际的程序代码中吧。就像我们处理 DNA 序列一样，你可以从文件中读入调控元件；文件中存储的是调控元件的列表，这样程序就可以查找不同的调控元件了。但是，在这种情况下，我们要使用的调控元件列表并不会改变，那为什么还要麻烦用户输入这样一个文件的文件名呢？

既然有了 DNA 序列和调控元件列表，我们就要概括地决定程序如何在 DNA 序列中查找每一个调控元件。这一步显然是最为关键的一步，它将一步定乾坤。比如，如果程序的运行速度是一个重要的考虑因素，你就要让程序运行的足够快。

在这个工作中，此处的问题就是要选择正确的算法。算法就是处理问题的构思（马上我就会对此进行详述）。比如，你打算交替读入调控元件，在读入下一个调控元件之前，从头到尾在 DNA 中对当前元件进行查找。或者也有可能，你打算对 DNA 序列只进行一次

从头到尾的读取，然后在 DNA 序列的每一个位置对每一个调控元件都进行一次查找，看看此处是否存在该调控元件。这两种方法有没有优劣之别？是否可以先对调控元件列表进行排序，这样查找就可以更快一些？此时，我们会说算法的选择至关重要。

构思的最后部分就是把结果以某种形式输出出来。也许你想把结果展示在网页上，或者在计算机屏幕上输出简单的列表，或者保存到一个可供打印的文件中，也可能是上述所有形式。在这个阶段，你可能需要让用户提供一个文件名来保存输出。

这就是如何展示结果的问题，这也确实是一个至关重要的问题。最理想的解决办法就是以某种方式把结果展示出来，这种方式使得用户仅需一瞥就可抓住计算过程中的重要特性。你可以使用图形、颜色、地图，甚至可以把异常结果用跳跃的小球标示出来，总之有很多方法。如果一个程序的输出结果很难解读，显然这并不是一个好的程序。事实上，让重要结果很难寻找或理解的输出，会彻底抹杀掉你在编写这个优雅程序过程中的所有付出。现在说的已经足够多了。

有非常多的策略，程序员可以使用这些策略来帮助他们进行好的整体构思。通常情况下，除了最小的程序外，所有的程序都会被分割成多个微小但却互相关联的部分。（在后续章节的学习中，我们会看到好多这样的例子。）每个部分是什么？它们之间如何进行关联？软件工程这个领域处理的就是这样的问题。此时此刻，我只想指出它们是非常重要的，提及一些程序员实现构思的途径。

有许多构思的方法，每种方法都有各自的拥护者。最好的办法就是了解一下有哪些可用的方法，并在处理任务时使用最合适的方法。比如，在本书中，我会教授一种叫做命令式编程的编程范式，它的理念是把一个问题分割成相互配合的程序或子程序（参看第 6 章），这就是所谓的结构化设计。另外一种流行的编程范式叫做面向对象编程，Perl 也支持这种范式。

如果你在一个大的项目中和一堆程序员共事，构思阶段可能会非常正式，甚至可能会由其他人来完成，而不是由程序员自己来完成。另一方面，你可能会发现有些程序员一上来就编写程序，他们会边写程序边制定计划。没有一种适用于所有程序员的方法。（萝卜白菜，各有所爱。）但不管你怎样实现它，作为初学者，在开始编写代码之前，你最好还是要要在头脑中有一定的构思。

3.5.2 算法

算法就是构思、计划，计算机程序的计算过程。（如果不使用正式的数学语言，这确实是一个难以定义的术语，不过此处也算是一个合理的定义。）一个算法要通过使用特定的计算机语言编程来实现，但算法本身是计算的思路。使用伪代码可以对算法进行很好的表述，它不是一个真正的计算机程序，但却展示了程序的思路。

大部分程序都只做简单的事情。它们从用户那里获取文件名，打开文件并读入数据，然后进行简单的计算后把结果展示出来。此处你将学习算法的类型。

不管怎样，算法学都是一门深奥、成果卓越的学科，对生物信息学也产生了深远的影响。通过算法，可以找到新的分析生物学数据的方法，发现新的科学成果。在生物学中，确实有不少这样的问题，这些问题的解决必须要等待新算法的发明。

算法学中有许多巧妙的技术。作为一个程序员菜鸟，现在还没有必要为这些担心。在这个阶段，本章作为编程方面的入门教程中的导言章节，对算法方法进行深入讲解并不明智。你的首要任务就是学习如何用某种编程语言进行编程。当然，如果对算法念念不忘，你可以去学习相关的技术。买一本像样的教科书放在身边作为参考书，对一个严谨的程序

员来说绝对是值得的。（参看附录 A）。

在现在这个例子中，就是在 DNA 序列中对调控元件进行计数的这个例子，我给出了一种策略：交替读取调控元件，在处理下一个调控元件之前，在 DNA 序列中进行从头到尾的查找。其他算法也是可以的。事实上，这是字符串匹配这个普遍性问题的一个实例。字符串匹配在生物信息学中最重要的一部分，而对它的研究已经催生了许多巧妙的算法。

算法常常和这样的问题或技术一起被提起，并且也有许多资料也可查阅。对于实用主义的程序员来说，最无价的材料莫过于用特定语言写成的算法代码，你可以直接把他们用在自己的程序中。可以从附录 A 开始，使用其中列出的代码合集和书籍，能够相对轻松的把许多算法技术整合到你的 Perl 代码中。

3.5.3 伪代码和代码

现在你有了整体构思，包括输入、算法和输出。你该如何把这些想法应用到程序的设计中呢？

一个常用的实践策略就是从编写伪代码开始。伪代码是一种非正式的程序，其中没有具体的细节，也不需要遵守正式的语法规则。³伪代码不会向程序那样实际运行，它的目的仅仅是以一种快速、非正式的方法把程序的整体构思具体化而已。

举个例子，在实际的 Perl 程序中你可能会写一些叫做子程序的代码（参看第 6 章）。在这个例子中，子程序会从用户的键入内容中获取答案；这个子程序可能会像这样：

```
1 | sub getanswer {
2 |     print "Type in your answer here :";
3 |     my $answer = <STDIN>;
4 |     chomp $answer;
5 |     return $answer;
6 | }
```

但是在伪代码中，你可能仅仅需要这样写：

```
1 | getanswer
```

至于细节日后再说。

对于上面讨论的内容，这里是程序伪代码的一个例子：

```
1 | get the name of DNAfile from the user
2 |
3 | read in the DNA from the DNAfile
4 |
5 | for each regulatory element
6 |     if element is in DNA, then
7 |         add one to the count
8 |
9 | print count
```

³语法规则指的是就是语法的规则。根据英语语法，“Go to school”是对的，而“School go to”是错的。程序语言也有这样的语法规则。

3.5.4 注释

注释是 Perl 源代码的一部分，旨在帮助理解程序的所作所为。从 # 开始到行末的所有内容都被看作注释，会被 Perl 解释器忽略掉。（唯一的例外是大多数 Perl 程序的第一行，是像这样的一行：`#!/usr/bin/perl`；参看第 4 章中的第 4.2.3 小节。）

注释对于保持代码可用是非常重要的。注释通常包括对于程序主要目的和整体构思的讨论、如何使用程序的实例，以及散布程序各处的细节注解，来解释为什么那段代码在这里、它是做什么用的。一般来说，好的程序员会把好的注释作为程序完整的一部分来进行编写。在本书的所有编程实例中，你都会看到注释。

你的代码不只会被计算机读入，也会被人查阅，这一点非常重要！

在调试行为异常的程序时，注释往往非常有用。如果无法确认程序出错的地方，你可以尝试选择性地注释掉不同部分的代码。如果你发现当注释掉某一部分代码后问题消失了，就可以把出错范围缩小到你注释掉的那部分代码，当这部分代码足够短时，你就找到问题的所在了。这常常是一个非常有用的调试方法。

在你把伪代码转换成 Perl 源代码时，也可以使用注释。伪代码不是 Perl 代码，所以对于任何没有注释掉的伪代码，Perl 解释器都会报错。在伪代码所在行的开头加上 # 就可以把它注释掉了：

```
1 | #get the name of DNAfile from the user
2 |
3 | #read in the DNA from the DNAfile
4 |
5 | #for each regulatory element
6 | #   if element is in DNA, then
7 | #       add one to the count
8 |
9 | #print count
```

在你把伪代码扩展成 Perl 代码时，移除 # 就可以对 Perl 代码取消注释了。用这种方法时，Perl 代码和伪代码会混杂在一起，但你可以正常运行或测试 Perl 代码部分，因为 Perl 解释器会把注释行简单地忽略掉。

你可以把伪代码整个保留在程序中，只需把它们注释掉即可。这种做法会保留下程序构思的提纲，当你或其他人试图阅读或修改代码时它们就会派上用场。

现在我们已经可以开始进行真正的 Perl 编程了。在第 4 章中，你会学习 Perl 的语法规则，并开始用 Perl 进行编程。当你开始编程时，牢记首先要对你的程序进行构思，之后才是你将花费大量时间重复去做的事情：编辑程序、运行程序，然后修正程序。

第 4 章 序列和字符串

目录

4.1 序列数据的表征	32
4.2 存储 DNA 序列的程序	35
4.3 连接 DNA 片段	39
4.4 转录：从 DNA 到 RNA	42
4.5 使用 Perl 文档	44
4.6 在 Perl 中计算反向互补	45
4.7 蛋白质，文件和数组	48
4.8 从文件中读取蛋白质序列数据	49
4.9 数组	53
4.10 标量上下文和列表上下文	57
4.11 练习题	59

在本章中，我们将开始编写 Perl 程序，来处理 DNA 和蛋白质的生物序列数据。在你的计算机上，一旦有了这样的序列，你就可以编写程序对序列数据进行下列的处理了：

- 把 DNA 转录成 RNA
- 把序列连接起来
- 获取反向互补序列
- 从文件中读取序列

此外，你还将编写获取序列信息的程序。DNA 的 GC 含量如何？蛋白质的疏水性如何？你将学习相关的编程技术，利用它们就可以来回答这些类似的问题了。

在本章中你将学习到的技能涉及 Perl 语言的基础，此处列出了其中的一部分：

- 标量变量
- 数组变量
- 替换、翻译等字符串的操作
- 从文件中读取数据

4.1 序列数据的表征

本书的大部分内容都是处理表征 DNA 和蛋白质生物学序列的符号。在生物信息学中，使用这些符号来表征序列，这些符号和生物学家日常工作中用来表征序列的符号是完全一样的。

如前所述，DNA 是由四种基础分子组成的，它们就是核酸，也叫做核苷酸或碱基；而蛋白质则是由 20 中基础分子组成的，它们就是氨基酸，也叫做残基。蛋白质的片段叫做肽（即缩氨酸）。不管是 DNA 还是蛋白质，本质上都是多聚物，由构成它们的基础分子首尾相连而形成。所以，仅仅通过碱基或氨基酸序列就可以表征 DNA 或蛋白质的基本结构。

这些都是最基本的定义。我假设你对这些内容都比较熟悉，或者打算查阅一本入门性的参考书来学习更加详细的内容。表 4.1 中列出的是碱基；在碱基上加上一个糖基，就可以得到核苷：腺苷、鸟苷、胞苷、胸苷和尿苷；在核苷上再加一个磷酸基团，就可以得到核苷酸：腺苷酸、鸟苷酸、胞苷酸、胸苷酸和尿苷酸。核酸就是由核苷酸通过化学键相连形成的序列。肽是由数个氨基酸相连而成的，更长的话就叫做多肽了。蛋白质是由一个或多个多肽组成的生物学功能基团。残基指的就是多肽链中的氨基酸。

为了方便，如表 4.1 和表 4.2 所示，常用一个字母或三个字母的代码来表示核酸和氨基酸。（本书中主要用单字母代码来表示氨基酸。）

表 4.1: 标准的 IUB/IUPAC 核酸代码

代码	核酸
A	腺嘌呤
C	胞嘧啶
G	鸟嘌呤
T	胸腺嘧啶
U	尿嘧啶
M	A 或 C (aMino, 氨基)
R	A 或 G (puRine, 嘌呤)
W	A 或 T (Weak, 作用力弱)
S	C 或 G (Strong, 作用力强)
Y	C 或 T (pYrimidine, 嘧啶)
K	G 或 T (keto, 酮基)
V	A 或 C 或 G (非 T)
H	A 或 C 或 T (非 G)
D	A 或 G 或 T (非 C)
B	C 或 G 或 T (非 A)
N	A 或 G 或 C 或 T (aNy, 任何一个碱基)

表 4.1 中的核酸代码不仅包括四种基本核酸的字母缩写，还定义了二个核酸、三个核酸或四个核酸所有可能组合的单字母缩写。在本书的大多数例子中，我仅使用 A、C、G、T、U 和 N。A、C、G 和 T 字母代表了 DNA 的核酸，而当 DNA（脱氧核糖核酸）转录成 RNA（核糖核酸）时 U 将替换 T。当测序仪无法确定碱基时，常用 N 来表示“未知碱基”。稍后，在第 9 章中，在编程处理限制性酶切图谱时，我们还需要用到表示各种核酸组合的其他代码。有时，也会使用这些单字母代码的小写形式，这对 DNA 来说比较常见，但在蛋白质中则很少这样使用。

对于表 4.1 和表 4.2 中的代码来说，计算机科学中的术语和生物学中的术语还是有一

表 4.2: 标准的氨基酸代码

单字母代码	氨基酸	三字母代码
A	Alanine, 丙氨酸	Ala
B	Aspartic acid or Asparagine, 天冬氨酸 或天冬酰胺	Asx
C	Cysteine, 半胱氨酸	Cys
D	Aspartic acid, 天冬氨酸	Asp
E	Glutamic acid, 谷氨酸	Glu
F	Phenylalanine, 苯丙氨酸	Phe
G	Glycine, 甘氨酸	Gly
H	Histidine, 组氨酸	His
I	Isoleucine, 异亮氨酸	Ile
K	Lysine, 赖氨酸	Lys
L	Leucine, 亮氨酸	Leu
M	Methionine, 甲硫氨酸	Met
N	Asparagine, 天冬酰胺	Asn
P	Proline, 脯氨酸	Pro
Q	Glutamine, 谷氨酰胺	Gln
R	Arginine, 精氨酸	Arg
S	Serine, 丝氨酸	Ser
T	Threonine, 苏氨酸	Thr
V	Valine, 缬氨酸	Val
W	Tryptophan, 色氨酸	Trp
X	Unknown, 未知氨基酸	Xxx
Y	Tyrosine, 酪氨酸	Tyr
Z	Glutamic acid or Glutamine, 谷氨酸或 谷氨酰胺	Glx

定差别的。从计算机科学的角度来看，这两个表定义了两个按字母顺序排列的有限的符号集合，使用它们可以来构建字符串。字符串指的就是符号序列。比如，这句话本身就是一个字符串（this sentence is a string）。语言就是一个（有限或无限）字符串的集合。在本书中，语言主要就是 DNA 和蛋白质的序列数据。和在序列数据中的具有生物学含义的表征不同，生物信息学家常常会把真正的 DNA 或蛋白质序列称作“字符串”。两个不同学科中的术语会交叉使用，这就是一个例子。

如同你在表格中看到的那样，我们会使用简单的字母来表征数据，这和在纸张上手写时使用的方式是一样的。计算机实际上会用另外的代码来表征简单的字母，但你不用担心这些，只要记住在使用文本编辑器时将它们保存为 ASCII 或者纯文本即可。

ASCII 是计算机在内存中存储文本（和控制信息）数据的一种方式。当文本编辑器等程序读取数据时，计算机知道它正在读取 ASCII，因为计算机知道每个代码代表什么，所以它就会在屏幕上以一种容易识别的方式把相应的字母显示出来。总而言之，知道 ASCII 是计算机表征文本的一种代码就足够了。¹

¹一种新的编码字符叫做 Unicode，它可以表征世界上所有语言中的所有符号，因此现在已得到广泛使用，并且 Perl 对 Unicode 也有很好的支持。

4.2 存储 DNA 序列的程序

让我们来写一个小程序吧，它把 DNA 存储在变量中，然后把它打印输出到屏幕上。我们会用最常用的方式——由 A、C、G 和 T 四个字母组成的字符串——来书写 DNA，并且把存储 DNA 的变量叫做 `$DNA`。换言之，`$DNA` 就是程序中 DNA 序列数据的名称。注意这一点，在 Perl 中，变量就是你打算处理的数据的名称，使用该名称，你可以对数据进行完全的访问。例 4.1 是完整的程序。

例 4.1：把 DNA 存储到计算机中

```
1 |#!/usr/bin/perl -w
2 |# Example 4-1    Storing DNA in a variable, and printing it out
3 |
4 |# First we store the DNA in a variable called $DNA
5 |$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
6 |
7 |# Next, we print the DNA onto the screen
8 |print $DNA;
9 |
10|# Finally, we'll specifically tell the program to exit.
11|exit;
```

在第 2 章中，我们已经学习了文本编辑器和运行 Perl 程序的知识，运用这些知识，输入例子中的代码（或者从书籍官网上把它复制下来）并保存到一个文件中。牢记一定要把程序保存成 ASCII 或者纯文本格式，否则 Perl 在读取该文件时可能会出现问题。

接下来就是运行程序了。运行程序的具体步骤取决于你使用的计算机（参看第 2 章）。我们假定程序是你计算机中一个叫做 *example4-1* 的文件。回顾第 2 章中相关的知识，如果你想在 Unix 或者 Linux 中运行这个程序，需要在 shell 窗口中键入：

```
1 |perl example4-1
```

在 Mac 中，使用 MacPerl 应用程序打开这个文件，并把它保存为 droplet，然后双击该 droplet 即可。在 Windows 中，在 MS-DOS 命令窗口中键入：

```
1 |perl example4-1
```

如果你成功运行了该程序，在你的计算机屏幕上你将看到相应的输出。

4.2.1 控制流

例 4.1 展示了所有 Perl 程序都将用到的许多理念，其中一个便是控制流的理念，即计算机是以什么顺序来执行程序中的语句的。

所有的程序都从第一行开始，除非明确指明了其他的运行顺序，否则它将一条一条地执行语句，直到程序的最后一行。例 4.1 只是简单的从头到尾执行程序，并没有其他的运行支路。

在后续的章节中，你将学习到如何编程控制程序的执行顺序。

4.2.2 再说注释

现在让我们看一下例 4.1 程序中的细节。你会发现其中有许多空行，它们的存在使得程序更容易被人阅读。另外，注意以 # 起始的注释。在第 3 章中提到过，当 Perl 运行时，它会把这些注释和空行全部忽略掉。事实上，对于 Perl 来说，下面的程序和例 4.1 中的程序是完全一样的：

```
1 |#!/usr/bin/perl -w
2 |$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'; print $DNA; exit;
```

在例 4.1 中，我使用了大量的注释。代码开始的注释指明了程序的用途、作者以及其他信息，当其他人需要理解代码时，这些信息会对他有很大的帮助。注释还解释了代码每一部分的作用，有时还对代码的工作原理进行了阐释。

了解注释的重要性是非常必要的。在大多数高校的计算机科学课程的课堂作业中，没有注释的程序通常会得到很低的分数甚至不及格；而在工作中不对代码进行注释的程序员，他的职业生涯将是短暂而失败的。

4.2.3 命令解释

程序的第一行以 # 起始，这使得它看上去像是一行注释，但又不像是有什么信息含量的注释：

```
1 |#!/usr/bin/perl -w
```

这是比较特殊的一行，叫做命令解释，它告诉运行 Unix 或者 Linux 的计算机，这是一个 Perl 程序。在不同的计算机中，这一行可能会有些许的差异。在某些计算机中，这一行并不是必需的，因为计算机可以通过其他信息识别出 Perl。在 Windows 计算机中，通常会配置成把以 .pl 结尾的任何程序都假定成 Perl 程序。在 Unix 或 Linux 中、Windows 的命令窗口中、或者 MacOS X 的 shell 中，你可以键入 perl my_program，这样你的 Perl 程序 my_program 中就不需要这样特殊的一行了。然而，通常都会写上这一行，所以在我们所有程序的开头也都会有它。

注意在第一行代码中使用了 -w 标志。“w”代表警告 warnings，它会使得 Perl 在遇到错误时打印出相应的信息。通常来说，错误信息都会给出出现错误的对应行号。有的时候，行号是错误的，但错误一般就出现信息指出的对应行的附近。在本书的后面，你会看到 -w 的另一种写法：use warnings;。

4.2.4 语句

例 4.1 中的下一行代码把 DNA 存储到了一个变量中：

```
1 |$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
```

这在计算机语言中是非常常见、非常重要的，所以我们将对它进行详细的解释。总的来说，你将会看到 Perl 和编程语言的一些基本特性，所以不要跳读了，停下来慢慢阅读学习吧。

这行代码叫做语句。在 Perl 中，语句以分号 (;) 结尾。用分号来结尾，就像英语中用句号来结尾一样。

更准确的来说，这一行是一个赋值语句。在该程序中，它作用就是把 DNA 存储到 \$DNA 变量中。正如在接下来的小节中你将看到的那样，此处有许多基本的事件发生。

变量

首先，让我们来看看 \$DNA 变量。它的名字有些随意，你可以给它起另外一个名字，程序仍会正常运行。举个例子，如果你把下面这两行：

```
1 | $DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';  
2 | print $DNA;
```

换成这样的两行：

```
1 | $A_poem_by_Seamus_Heaney = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';  
2 | print $A_poem_by_Seamus_Heaney;
```

程序仍将像原来那样正常运行，把 DNA 打印输出到计算机屏幕上。不管怎样，计算机程序中变量的名字都是由你来起的。（要满足特定的限制：在 Perl 中，变量的名字只能由大小写字母、数字和下划线 _ 组成，而且第一个字符不能是数字。）

前面已经强调过，使用空行和注释可以使代码更加易读，变量的命名也存在同样的问题。不管变量名是 \$DNA 还是 \$A_poem_by_Seamus_Heaney，对于计算机来说都没有什么特殊的含义，但对于阅读程序的人来说就不一样了。有意义的变量名，可以清晰地表明程序中变量的作用，使得程序更加容易理解。其他的名字可能会使得程序的功能和变量的作用不甚明朗。使用精心选择的变量名是自文档化代码的一部分（self-documenting code）。精心选择变量名的话，你仍然需要注释，但就不需要那么多的注释了。

你会注意到 \$DNA 变量名以一个美元符号起始。在 Perl 中，这样的变量叫做标量变量，它是存储单个数据项目的变量。在存储字符串或者各种各样的数字（如，字符串 hello，或者 25、6.234、3.5E10、-0.8373 这样的数字）时，需要使用标量变量。一个标量变量一次只能存储数据中的一个项目。

字符串

在例 4.1 中，标量变量 \$DNA 存储着用 A、C、G 和 T 表征的 DNA。如前所述，在计算机科学中，字母序列叫做字符串。在 Perl 中，你需要把它放在引号中来表明它是字符串。可以使用单引号，就像例 4.1 中那样，也可以使用双引号。（稍后你会看到两者的区别。）因此，DNA 就被表征成：

```
1 | 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'
```

赋值

在 Perl 中，要把一个变量设成特定的值，需要使用 = 符号。= 符号因此被叫做赋值操作符。在例 4.1 中，值

```
1 | 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'
```

被赋给了 `$DNA` 变量。赋值后，你可以通过变量名来获取它的值，就像例 4.1 中的 `print` 语句那样。

在赋值语句中，每个部分的顺序是非常重要的。要赋给变量的值在赋值操作符的右边，而需要赋值的变量总在赋值操作符的左边。在编程手册中，有时你可能会遇到 *lvalue* 和 *rvalue* 这样的术语，它们分别指代赋值操作符左边和右边的项目。

在编程语言中，使用 `=` 符号进行赋值有一段很长的历史。然而，这也导致了某种形式的混乱：比如说，在数学中，使用 `=` 时表示等号两边的数是相等的。所以，一定要牢记，在 Perl 中，`=` 符号并不表示相等，而是把值赋给一个变量。（稍后我们会看到如何表示相等）。

来总结一下，对于这个语句，我们都学习到了那些知识：

```
1 | $DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
```

这是一个赋值语句，它把表征 DNA 的字符串赋给了变量 `$DNA`，作为这个变量的值。

打印输出

该语句：

```
1 | print $DNA;
```

把 `ACGGGAGGACGGGAAAATTACTACGGCATTAGC` 打印输出到计算机屏幕上。注意，`print` 语句处理的是标量变量，它把它的值打印输出出来，此处就是变量 `$DNA` 包含的字符串。稍后，你将看到更多关于打印输出的内容。

退出

最后，`exit;` 语句告诉计算机退出程序。在 Perl 语言中，在程序的最后并不需要 `exit` 语句，一旦运行到结尾，程序就会自动退出。但放上这么一个语句也没什么坏处，还明确表示了程序的结束。你会看到，在正常结束之前，程序会因为某些错误而退出，所以 `exit` 语句还是非常有用的。

4.3 连接 DNA 片段

现在，我们对例 4.1进行简单的修改，来演示一下如何把 DNA 片段连接起来。所谓连接指的就是把一个东西附加在另一个东西的结尾上。生物学家都知道，在生物学实验室中把 DNA 序列连接起来是非常常见的工作，比如把克隆插入到细胞载体中，或者在基因表达过程中把剪切的外显子连接起来。许多生物信息学的软件包都能够进行这样的工作，因此这里只是把它作为一个实例来讲解。

例 4.2演示了关于字符串、变量和打印输出语句的更多内容。

例 4.2: 连接 DNA

```
1  #!/usr/bin/perl -w
2  # Example 4-2   Concatenating DNA
3
4  # Store two DNA fragments into two variables called $DNA1 and $DNA2
5  $DNA1 = 'ACGGGAGGACGGGAAAAATTACTACGGCATTAGC';
6  $DNA2 = 'ATAGTGCCGTGAGAGTGATGTAGTA';
7
8  # Print the DNA onto the screen
9  print "Here are the original two DNA fragments:\n\n";
10
11 print $DNA1, "\n";
12
13 print $DNA2, "\n\n";
14
15 # Concatenate the DNA fragments into a third variable and print them
16 # Using "string interpolation"
17 $DNA3 = "$DNA1$DNA2";
18
19 print "Here is the concatenation of the first two fragments (version 1):\n\n";
20
21 print "$DNA3\n\n";
22
23 # An alternative way using the "dot operator":
24 # Concatenate the DNA fragments into a third variable and print them
25 $DNA3 = $DNA1 . $DNA2;
26
27 print "Here is the concatenation of the first two fragments (version 2):\n\n";
28
29 print "$DNA3\n\n";
30
31 # Print the same thing without using the variable $DNA3
32 print "Here is the concatenation of the first two fragments (version 3):\n\n";
33
34 print $DNA1, $DNA2, "\n";
35
36 exit;
```

就像你看到的那样，这里三个变量：`$DNA1`、`$DNA2` 和 `$DNA3`。为了对运行过程

进行说明，我添加了一些 `print` 语句，这样打印到计算机屏幕上的输出就不仅仅是一个接一个 DNA 片段了，看起来会更加明了一些。

下面是例 4.2 的输出：

```

1 | Here are the original two DNA fragments:
2 |
3 | ACGGGAGGACGGGAAAATTACTACGGCATTAGC
4 | ATAGTGCCGTGAGAGTGATGTAGTA
5 |
6 | Here is the concatenation of the first two fragments (version 1):
7 |
8 | ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA
9 |
10 | Here is the concatenation of the first two fragments (version 2):
11 |
12 | ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA
13 |
14 | Here is the concatenation of the first two fragments (version 3):
15 |
16 | ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA

```

例 4.2 和例 4.1 有许多相似之处。让我们来看一下两者的不同吧。作为开始，我们会发现 `print` 语句中多了一些看起来并不直观的东西：

```

1 | print $DNA1, "\n";
2 | print $DNA2, "\n\n";

```

像前述一样，`print` 语句中有包含 DNA 的变量，但现在后面又多了逗号和 `"\n"` 或者 `"\n\n"`。这些都是打印换行符的指令。换行符在纸张页面或者屏幕上并不可见，但它告诉计算机转换到新行的开头，以便输出后续内容。`"\n"`，一个换行符，只是简单得定位到下一行的开头而已；`"\n\n"`，两个新行，移动到下一行，然后定位到后面一行的开头，在两者之间留下一个空行。

看一下例 4.2 的代码，确保已经明白这些换行符是如何决定输出效果的。空行指的就是没有任何打印输出的一行。取决于你的操作系统，它可能仅仅是一个换行符，也可能是换页符和回车符的组合（在这种情况下，它也被叫做空白行），还有可能包含空格和制表符等非打印的空白字符。注意，包裹在双引号中的换行符表示它们是字符串的一部分。（如前所述，这里是单引号好双引号的区别之一：`"\n"` 会打印输出换行符，而 `'\n'` 而打印输出 `\n` 本身。）

注意在 `print` 语句中还有逗号。逗号分隔列表中的项目，`print` 语句会打印输出列表中的所有项目。仅此而已。

现在，让我们看一下把 `$DNA1` 和 `$DNA2` 两个 DNA 片段连接到 `$DNA3` 变量中的语句吧：

```

1 | $DNA3 = "$DNA1$DNA2";

```

对 `$DNA3` 进行的赋值是一个典型的赋值操作，就像你在例 4.1 看到的那样，变量名后面跟着 `=` 符号，`=` 后则是要被赋予的值。

赋值语句中右边的值是包裹在双引号中的字符串。双引号会把字符串中的变量替换为变量的值，这叫做字符串内插。²所以事实上，此处的字符串就是 \$DNA1 变量中的 DNA，后面紧跟着 \$DNA2 变量中的 DNA。两个 DNA 片段连接后就被赋值给了变量 \$DNA3。


在把连接结果赋值给 \$DNA3 变量后，你把她打印出来，后面跟着一个空行：

```
1 | print "$DNA3\n\n";
```

Perl 的中心思想之一就是 “There’s more than one way to do it.”（不只一种方法来做一件事）。所以，程序的下面一部分就演示了连接两个字符串的另外一种办法——使用点操作符。当把点操作符放在两个字符串中间时，它会把原来的两个字符串连接起来，产生一个新的字符串。所以这一行：

```
1 | $DNA3 = $DNA1 . $DNA2;
```

演示了点操作符的使用。



在计算机语言中，一个操作符需要一些参数——在这个例子中，就是 \$DNA1 和 \$DNA2 字符串——并对它们进行操作，然后返回一个值——在这个例子中，就是连接后保存在 \$DNA3 变量中的字符串。算术中最为常见的操作符加减乘除都是需要两个数字作为参数、并返回一个数字作为返回值的操作符。

最后，来练习一下 Perl 的另外一种方法，仅仅使用 print 语句就可以完成同样的连接任务：

```
1 | print $DNA1, $DNA2, "\n";
```

此处的 print 语句有用逗号分隔开的三部分：存储在两个变量中的两个 DNA 片段和一个换行符。使用下面这一个 print 语句你也可以得到同样的结果：

```
1 | print "$DNA1$DNA2\n";
```

也许 Perl 的口号应该改成 “There are more than two ways to do it.”（不只两种方法来做一件事）。

在结束这一小节之前，让我们来看看 Perl 变量的其他用法吧。你已经看到，使用变量可以存储 DNA 序列数据的字符串。还有其他类型的数据，编程语言也需要变量来存储它们。在 Perl 中，一个像 \$DNA 这样的标量变量可以存储字符串、整数、浮点数（有小数点的数字）、布尔值（true 或 false）等。当需要的时候，Perl 能够知道变量中存储的是哪种类型的数据。现在，试着在例 4.1 或例 4.2 中添加如下几行，在标量变量中存储一个数字并把它打印出来：

```
1 | $number = 17;
2 | print $number, "\n";
```

²有时在字符串内插时你会加上大括号。大括号可以确保变量名不会与双引号中的字符串混在一起。举个例子，你有一个叫做 \$prefix 的变量，想把它内插到 I am \$prefixinterested 字符串中，Perl 可能无法识别出这个变量，而把它错当成一个并不存在的 \$prefixinterested 变量。但是 I am \${prefix}interested 对于 Perl 来说就没有歧义了。

4.4 转录：从 DNA 到 RNA

作为生物信息学中的 Perl 程序员,你很大一部分时间都是在做类似于例 4.1和例 4.2那样的事情:你获取一些数据,可能是 DNA、蛋白质、GenBank 条目或者其他数据,处理这些数据,并把处理结果输出打印出来。

例 4.3是处理 DNA 的另一个程序:它把 DNA 转录成 RNA。在细胞中,把 DNA 转录成 RNA 是由精巧、复杂且有勘误功能的分子机器完成的。³此处仅是简单的替换而已。当 DNA 转录成 RNA 时,所有的 T 都会被替换成 U,这也正是我们的程序所要做的全部工作。

4

例 4.3: 把 DNA 转录成 RNA

```
1 #!/usr/bin/perl -w
2 # Example 4-3   Transcribing DNA into RNA
3
4 # The DNA
5 $DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
6
7 # Print the DNA onto the screen
8 print "Here is the starting DNA:\n\n";
9
10 print "$DNA\n\n";
11
12 # Transcribe the DNA to RNA by substituting all T's with U's.
13 $RNA = $DNA;
14
15 $RNA =~ s/T/U/g;
16
17 # Print the RNA onto the screen
18 print "Here is the result of transcribing the DNA to RNA:\n\n";
19
20 print "$RNA\n";
21
22 # Exit the program.
23 exit;
```

这是例 4.3的输出:

```
1 Here is the starting DNA:
2
3 ACGGGAGGACGGGAAAATTACTACGGCATTAGC
4
5 Here is the result of transcribing the DNA to RNA:
6
7 ACGGGAGGACGGGAAAAUUACUACGGCAUUAGC
```

³简要说来, DNA 的编码链是另一条链(模板链)的反向互补链,模板链作为合成 RNA 的模板,把 T 替换成 U,生成反向互补链。两次反向互补后,除了 T→U 的替换外,就和编码链完全一样了。

⁴很明显,我们忽略了切除内含子的过程。T 代表胸腺嘧啶, U 代表尿嘧啶。

这个简短的程序展示了 Perl 重要的特性：轻松处理 DNA 字符串等文本数据的能力。类似的处理可能会有多种：翻译、反转、替换、删除和重排序等等。总的来说，Perl 在此类任务中的便利是其能够在生物信息学领域成功及在程序员中广泛应用的主要原因。

首先，程序生成了一个 DNA 的拷贝，并把它存储在叫做 `$RNA` 的变量中：

```
1 | $RNA = $DNA;
```

注意在执行这个语句后，叫做 `$RNA` 的这个变量存储的就是 DNA。⁵你可以给变量起任何你喜欢的名字，这是完全合法的，但不准确的变量名可能会导致一些混乱。在这个例子中，拷贝完变量值后，紧跟着的是内容丰富的注释，之后便是让 `$RNA` 变量真正包含 RNA 的语句，所以此处给它起名为 `$RNA` 完全没有问题。有一个让 `$RNA` 只包行 RNA 而不包含其他内容的方法：

```
1 | ($RNA = $DNA) =~ s/T/U/g;
```

在例 4.3 中，转录过程发生在这个语句中：

```
1 | $RNA =~ s/T/U/g;
```

在这个语句中有两个新的项目：绑定操作符 (`=~`) 和替换命令 `s/T/U/g`。

很明显，绑定操作符 `=~` 用于包含字符串的变量，如此处的 `$RNA` 变量包含 DNA 序列数据。绑定操作符表示“把右边的操作应用到左边变量中的字符串上”。

如图 4.1 所示，对替换操作符需要进行一些详细的解释。该命令的不同部分用正斜线相分隔（或限定）。首先，`s` 表明这是一个替换（substitution）。在第一个正斜线 `/` 后面跟着的是 `T`，它代表字符串中要被替换的元素。在第二个正斜线 `/` 后面跟着的是 `U`，它代表将要取代 `T` 的元素。最后，在第三个正斜线后面跟着的是 `g`，这个 `g` 代表全局（global）替换，它是可以出现在语句该部分中众多可能的修饰符中的一个。全局替换表示“从头到尾对字符串进行替换”，也就是说，对字符串中每一处可能的地方都进行替换。

因此，这个语句含义就是“把 `$RNA` 变量存储的字符串数据中的所有 `T` 都替换成 `U`”。

替换操作符是使用正则表达式的一个例子。正则表达式对于文本处理来说至关重要，在后续的章节中你将看到，正则表达式是 Perl 最为强大的特性之一。

⁵回顾一下第 4.2.4 小节中关于赋值语句中各个部分顺序的重要性的讨论。此处，`$DNA` 的值，也就是存储在 `$DNA` 变量中的 DNA 序列数据，被赋给了 `$RNA` 变量。如果你写成了 `$DNA = $RNA;`，`$RNA` 变量的值（空值）将会被赋给 `$DNA` 变量，导致变量中的 DNA 序列数据被清空，最终留下了两个空的变量。

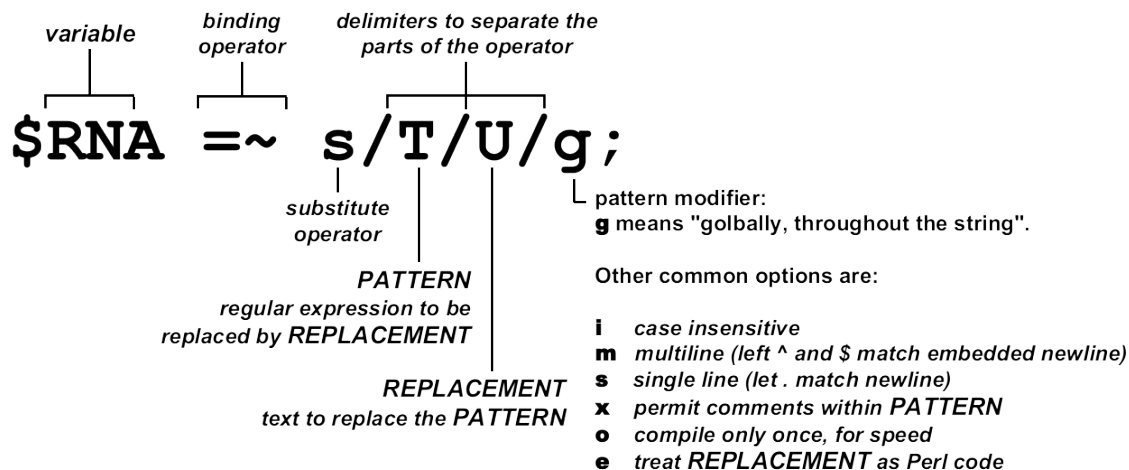


图 4.1: 替换操作符

4.5 使用 Perl 文档

对于 Perl 程序员来说，最重要的资源便是 Perl 的文档。它应该已经安装在了你的电脑上，另外通过因特网在 Perl 网站上也可以找到它。对于不同的计算机系统来说，Perl 文档的格式可能有少许的差别，但网络版对任何一个人来说都是一样的，这也正式我在本书中参考的版本。参看附录 A 中的参考资料，你会找到对于 Perl 文档不同资源的讨论。

来试一下，让我们找找 `print` 操作符的文档吧。首先，打开你的网页浏览器，进入 <http://www.perl.com> 网站，然后点击文档链接，依次选择“Perl 的内置函数”（“Perl’s Builtin Functions”）和“按字母顺序罗列的 Perl 函数”（“Alphabetical Listing of Perl’s Functions”）⁶。你会看到一个把 Perl 函数按字母顺序进行排列的一个冗长的列表。一旦你找到这个页面，你可能需要把它收藏到浏览器的书签中，因为你会频繁地访问该页面。现在点击 `print` 来查看 `print` 操作符的文档吧。

看一下文档中的例子，看看 Perl 语言的特性是如何被运用的。这往往是你找到所需内容的最快方法。

一旦在你的屏幕中打开了文档，你会发现通过阅读它会找到一些答案，但也会产生其他一些疑问。文档试图以一种简洁的形式把所有的内容都包含进去，但这却会让初学者们心生胆怯。比如，`print` 函数文档的开头还比较简单：“打印输出一个字符串或者一个逗号分隔的字符串列表。如果成功将返回 TRUE”。但之后就是一堆的胡言乱语了（在你学习的现阶段它看起来确实是这样）：文件句柄？输出流？列表上下文？

文档中的所有信息都是必需的，毕竟，你需要在某个地方找到这样的全部内容！通常情况下，你可以忽略掉那些对你来说毫无意义的内容。

Perl 文档中也包含了一些对学习 Perl 有很大帮助的教程。有时，它们会假定你掌握了比一个编程语言初学者应当掌握的知识更多的知识，但你会发现它们仍然非常有用。翻阅文档是在学习 Perl 语言过程中快速成长的绝佳途径。

⁶译者注：打开<http://www.perl.com>网站，点击右侧的“Document”链接，之后点击左侧的“Functions”链接，最后点击“Perl functions A-Z”即可。

4.6 在 Perl 中计算反向互补

第 1 章中已经提到了，DNA 聚合物是由核苷酸构成的。考虑到 DNA 双螺旋中两条链的亲密关系，最好能编写这样一个程序：给出一条链，输出另一条链。这样的工作对许多生物信息学应用程序来说都是非常重要的一部分。比如，当在数据库中查询某条 DNA 时，常常也需要自动查询该 DNA 的反向互补序列，因为你手上的序列有可能是某个已知基因的负链。

回到正题，这是例 4.4，它使用了一些新的 Perl 特性。就像你将看到的那样，它首先尝试一种方法，失败了，然后尝试另外一种方法，最终取得了成功。

例 4.4：计算 DNA 一条链的反向互补链

```
1  #!/usr/bin/perl -w
2  # Example 4-4    Calculating the reverse complement of a strand of DNA
3
4  # The DNA
5  $DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
6
7  # Print the DNA onto the screen
8  print "Here is the starting DNA:\n\n";
9
10 print "$DNA\n\n";
11
12 # Calculate the reverse complement
13 # Warning: this attempt will fail!
14 #
15 # First, copy the DNA into new variable $revcom
16 # (short for REVerse COMplement)
17 # Notice that variable names can use lowercase letters like
18 # "revcom" as well as uppercase like "DNA". In fact,
19 # lowercase is more common.
20 #
21 # It doesn't matter if we first reverse the string and then
22 # do the complementation; or if we first do the complementation
23 # and then reverse the string. Same result each time.
24 # So when we make the copy we'll do the reverse in the same statement.
25 #
26
27 $revcom = reverse $DNA;
28
29 #
30 # Next substitute all bases by their complements,
31 # A->T, T->A, G->C, C->G
32 #
33
34 $revcom =~ s/A/T/g;
35 $revcom =~ s/T/A/g;
36 $revcom =~ s/G/C/g;
37 $revcom =~ s/C/G/g;
```

```
38 |
39 | # Print the reverse complement DNA onto the screen
40 | print "Here is the reverse complement DNA:\n\n";
41 |
42 | print "$revcom\n";
43 |
44 | #
45 | # Oh-oh, that didn't work right!
46 | # Our reverse complement should have all the bases in it, since the
47 | # original DNA had all the bases-but ours only has A and G!
48 | #
49 | # Do you see why?
50 | #
51 | # The problem is that the first two substitute commands above change
52 | # all the A's to T's (so there are no A's) and then all the
53 | # T's to A's (so all the original A's and T's are all now A's).
54 | # Same thing happens to the G's and C's all turning into G's.
55 | #
56 |
57 | print "\nThat was a bad algorithm, and the reverse complement was wrong!\n";
58 | print "Try again ... \n\n";
59 |
60 | # Make a new copy of the DNA (see why we saved the original?)
61 | $revcom = reverse $DNA;
62 |
63 | # See the text for a discussion of tr///
64 | $revcom =~ tr/ACGTacgt/TGCAtgca/;
65 |
66 | # Print the reverse complement DNA onto the screen
67 | print "Here is the reverse complement DNA:\n\n";
68 |
69 | print "$revcom\n";
70 |
71 | print "\nThis time it worked!\n\n";
72 |
73 | exit;
```

这是你屏幕上例 4.4 的输出:

```
1 | Here is the starting DNA:
2 |
3 | ACGGGAGGACGGGAAAATTACTACGGCATTAGC
4 |
5 | Here is the reverse complement DNA:
6 |
7 | GGAAAAGGGGAAGAAAAAAGGGGAGGAGGGGA
8 |
9 | That was a bad algorithm, and the reverse complement was wrong!
10 | Try again ...
```

```
11 |
12 | Here is the reverse complement DNA:
13 |
14 | GCTAATGCCGTAGTAATTTTCCCGTCCTCCCGT
15 |
16 | This time it worked!
```

通过从不同的末端开始读起，也就是说一条链从左向右读，另一条链从右向左读，你可以检查一下 DNA 的两条链是不是反向互补的。当你读这两条链的时候，比较它们对应的碱基，应该总是 C 和 G 对应、A 和 T 对应。

在第一次尝试中，试着从原始的 DNA 和反向互补后的 DNA 中读几个字符，你会发现反向互补的第一次尝试失败了。这是一个错误的算法。

这是在你程序中经常要经历的一种体验。你写一个程序来完成某项任务，但却发现它并没有像你期望的那样工作。在这种情况下，我们需要运用已掌握的知识来尝试解决全新的问题。它并没有如期望那样完成任务，哪里出错了呢？

你会发现这样的经历会非常相似：你编写代码，但它并不工作！然后你就修正语法（这通常是最简单的，而且利用错误信息提供的线索就可以轻松修正语法），或者对问题进行思考，找到问题所在，并尝试设计一种新的可以成功的方法。通常情况下，这需要你浏览语言的文档，查找语言工作过程的细节内容，同时期望能找到一个解决问题的特性。如果这个问题在计算机上能够解决，那么你用 Perl 也可以将它解决。问题是，能够精确到什么程度？

在例 4.4 中，计算反向互补的第一次尝试失败了。使用四个全局替换，序列中的每一个碱基都作为一个整体进行了处理。还需要另外的方法。你可以从左向右查阅 DNA，每次只查找一种碱基，把它替换成互补的碱基，然后再在 DNA 中查找另外一种碱基，一直到字符串的结尾。然后把字符串反转过来，任务就完成了。事实上，这是一个非常好的方法，在 Perl 中也不难实现。但还需要学习语言的其他内容才行，这将在第 5 章中进行讲解。

然而，在这个例子中，`tr` 操作符——表示直译或翻译——正好适合这项任务。它看起来像替换命令一样，都使用三个正斜线来分隔不同的部分。

`tr` 正是我们所需要的，它会一次性把一个字符集合翻译成新的字符。图 4.2 展示了它的工作原理：需要翻译的字符集合位于前两个正斜线之间，而将要替换它们的新的字符集合则位于第二个和第三个正斜线之间。第一个集合中的每一个字符都将被翻译成第二个集合中同样位置的字符。举个例子，在例 4.4 中，`C` 是第一个集合中的第二个字符，所以它会被翻译成第二个集合中的第二个字符，也就是 `G`。最后，DNA 序列数据可以使用大写或者小写字母（虽然在本程序中 DNA 都是大写的），这两种情况在例 4.4 的 `tr` 语句中都进行了考虑。

`reverse` 函数进行的工作也是我们需要的，虽然有点大材小用了。它被设计用来反转元素的顺序，包括字符串，就像在例 4.4 中看到的那样。

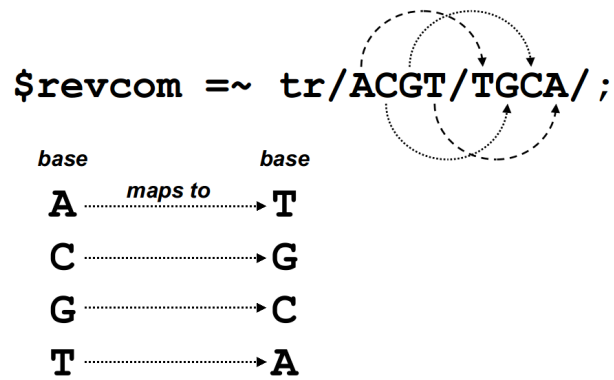


图 4.2: tr 语句

4.7 蛋白质，文件和数组

到现在为止，我们已经编写了处理 DNA 序列数据的程序，现在，我们也将要处理同样重要的蛋白质序列数据。接下来的几个小节将要学习一些新的知识，这里是一个简单的概述：

- 在 Perl 程序中如何使用蛋白质序列数据
- 如何从一个文件中读入蛋白质序列数据
- Perl 语言中的数组

在本章的剩余部分中，蛋白质和 DNA 序列数据都将使用到。

4.8 从文件中读取蛋白质序列数据

程序要和计算机磁盘中的文件进行交互。这些文件可以存储在任何永久性存储介质中——硬盘、光盘、软盘、Zip 磁盘、磁带等。

让我们看看如何从文件中读取蛋白质序列数据吧。首先，在你的计算机上创建一个文件（使用文本编辑器），并在其中存储一些蛋白质序列数据。给这个文件起名为 *NM_021964fragment.pep*（你可以在书籍网站上下载到该文件）。你将使用下列数据（人类锌指蛋白 NM_021964 的一部分）：

```
1 | MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
2 | SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDDEEQMETHERLPQ
3 | GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

你可以给它起任意一个名字，只要和文件夹中已有的文件不重名即可。

就像好的变量名对于理解程序至关重要一样，好的文件名和文件夹名也是非常重要的。如果你有一个项目，这个项目会生成大量的文件，那你就需要认真考虑如何对这些文件和文件夹命名和组织了。不管是对于某一个研究人员还是对于一个庞大的多国团队来说，这都是非常现实的问题。花一些功夫来给文件起一个富含信息量的名字是非常重要的。

文件名 *NM_021964fragment.pep* 来源于蛋白质的 GenBank ID。同时，它还表明了这只是数据的一部分，而文件的后缀 *.pep* 则提醒你文件中保存的是肽或蛋白质序列数据。当然，其他的命名方案可能会更加适合于你。不管怎样，关键的一点就是不需要打开文件，仅仅通过文件名就可以对文件保存的数据有所了解。

你已经创建或者下载了保存有蛋白质序列数据的文件，那我们就来编写一个程序吧，这个程序从文件中读入蛋白质序列数据并把它保存到变量中。例 4.5 是第一次尝试，随着学习我们会逐步对它进行扩展。

例 4.5：从文件中读取蛋白质序列数据

```
1 | #!/usr/bin/perl -w
2 | # Example 4-5    Reading protein sequence data from a file
3 |
4 | # The filename of the file containing the protein sequence data
5 | $proteinfilename = 'NM_021964fragment.pep';
6 |
7 | # First we have to "open" the file, and associate
8 | # a "filehandle" with it. We choose the filehandle
9 | # PROTEINFILE for readability.
10 | open( PROTEINFILE, $proteinfilename );
11 |
12 | # Now we do the actual reading of the protein sequence data from the file,
13 | # by using the angle brackets < and > to get the input from the
14 | # filehandle. We store the data into our variable $protein.
15 | $protein = <PROTEINFILE>;
16 |
17 | # Now that we've got our data, we can close the file.
18 | close PROTEINFILE;
19 |
```

```

20 | # Print the protein onto the screen
21 | print "Here is the protein:\n\n";
22 |
23 | print $protein;
24 |
25 | exit;

```

下面是例 4.5 的输出：

```

1 | Here is the protein:
2 |
3 | MNIDDKLEGLFLKCGGIDEMQSSRTMVMGGVSGQSTVSGELQD

```

注意只有文件的第一行被打印输出出来，稍后我会解释为什么会这样。

让我们仔细研究一下例 4.5 吧。在把文件名保存到 `$proteinfilename` 变量中后，下面这个语句会打开文件：

```

1 | open(PROTEINFILE, $proteinfilename);

```

打开文件后，你就可以对它进行各种操作了，比如读取、写入、查找、定位到文件的特定位置、清除文件中的所有数据，等等。注意，程序假设保存在 `$proteinfilename` 变量中的文件是存在的，而且可以打开。你将会看到如何来确认这一点，现在你可以进行一下这样的尝试：更改 `$proteinfilename` 变量中文件的名字，这样计算机上就没有叫原来那个名字的文件了，之后运行一下程序。如果文件并不存在，你会看到一些错误信息。

如果你查阅 `open` 函数的文档，你会看到好多选项，大多数选项都是在打开文件后让你精确指定如何使用文件的。

让我们来看一下 `PROTEINFILE` 这个数据，它叫做文件句柄。没有必要理解文件句柄真正是什么，它们就是你处理文件时使用的东西。对于文件句柄，不一定必须使用大写字母，但这是一个广为接受的惯例。在使用 `open` 语句对文件句柄赋值后，对文件进行的任何交互操作都将通过使用文件句柄来进行。

使用这个语句，就可以把数据读入到程序中了：

```

1 | $protein = <PROTEINFILE>;

```

为什么要把文件句柄 `PROTEINFILE` 包裹在尖括号中呢？这些尖括号叫做输入操作符。通过把文件句柄包裹在尖括号中，你就可以从程序外部的来源中读入数据了。在这个例子中，我们读入了 `NM_021964fragment.pep` 文件中的数据，该文件的名字保存在 `$proteinfilename` 变量中，而 `open` 语句则把它和文件句柄关联了起来。数据现在就保存到了 `$protein` 变量中，之后可以把它打印输出出来。

然而，就像我们前面已经注意到的那样，只有这个多行文件的第一行被打印了出来。这是问什么呢？因为对于读取文件还有一些知识需要学习。

有许多方法可以读取整个文件，例 4.6 演示了其中的一种。

例 4.6：从文件中读取蛋白质序列数据，第二次尝试

```

1 | #!/usr/bin/perl -w
2 | # Example 4-6   Reading protein sequence data from a file, take 2

```

```
3 |
4 | # The filename of the file containing the protein sequence data
5 | $proteinfilename = 'NM_021964fragment.pep';
6 |
7 | # First we have to "open" the file, and associate
8 | # a "filehandle" with it. We choose the filehandle
9 | # PROTEINFILE for readability.
10 | open( PROTEINFILE, $proteinfilename );
11 |
12 | # Now we do the actual reading of the protein sequence data from the file,
13 | # by using the angle brackets < and > to get the input from the
14 | # filehandle. We store the data into our variable $protein.
15 | #
16 | # Since the file has three lines, and since the read only is
17 | # returning one line, we'll read a line and print it, three times.
18 |
19 | # First line
20 | $protein = <PROTEINFILE>;
21 |
22 | # Print the protein onto the screen
23 | print "\nHere is the first line of the protein file:\n\n";
24 |
25 | print $protein;
26 |
27 | # Second line
28 | $protein = <PROTEINFILE>;
29 |
30 | # Print the protein onto the screen
31 | print "\nHere is the second line of the protein file:\n\n";
32 |
33 | print $protein;
34 |
35 | # Third line
36 | $protein = <PROTEINFILE>;
37 |
38 | # Print the protein onto the screen
39 | print "\nHere is the third line of the protein file:\n\n";
40 |
41 | print $protein;
42 |
43 | # Now that we've got our data, we can close the file.
44 | close PROTEINFILE;
45 |
46 | exit;
```

下面是例 4.6的输出:

```
1 | Here is the first line of the protein file:
2 |
```

```
3 | MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
4 |
5 | Here is the second line of the protein file:
6 |
7 | SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDDEEQMETHERLPQ
8 |
9 | Here is the third line of the protein file:
10 |
11 | GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

这个程序中比较有趣的一点就是，它演示了如何成功得从文件中读取数据。每当把数据读取到 `$protein` 这样的标量变量中后，都会对文件的下一行进行读取。上一次读取到哪儿了，现在需要读取哪一行，程序对这些信息都有所记录。

另一方面，程序的缺陷也是显而易见的。对于输入文件的每一个行，都需要编写一行代码来读入，这样太不方便了。但是，在 Perl 中有两个特性可以很好的解决这个问题：数组（下一小节进行介绍）和循环（参看第 5 章）。

4.9 数组

在计算机语言中，数组是存储多个标量值的变量。这些标量的值可以是数字、字符串，或者，像此处的例子一样，也可以是蛋白质序列数据输入文件中的每一行。让我们看看如何来使用数组吧。例 4.7 演示了如何使用数组来把输入文件中的所有行都读入进来。

例 4.7：从文件中读取蛋白质序列数据，第三次尝试

```

1  #!/usr/bin/perl -w
2  # Example 4-7   Reading protein sequence data from a file, take 3
3
4  # The filename of the file containing the protein sequence data
5  $proteinfilename = 'NM_021964fragment.pep';
6
7  # First we have to "open" the file
8  open( PROTEINFILE, $proteinfilename );
9
10 # Read the protein sequence data from the file, and store it
11 # into the array variable @protein
12 @protein = <PROTEINFILE>;
13
14 # Print the protein onto the screen
15 print @protein;
16
17 # Close the file.
18 close PROTEINFILE;
19
20 exit;
```

下面是例 4.7 的输出：

```

1  MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
2  SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDDEEQMETHERLPQ
3  GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

就像你看到的，这就是文件中的全部数据。我们成功了！

使用数组的方便性显而易见——只需要一行代码就可以把所有的数据都读入到程序中了。

注意，和标量变量以美元符号 (\$) 起始不同，数组变量是以 @ 符号起始的。此外，还要注意，print 函数不仅可以处理标量变量，同样可以处理数组变量。在 Perl 中，数组被广泛使用，所以在本书的剩余部分你会看到大量的数组应用的实例。

数组是存储多个标量值的变量。数组中的每一个项目或者元素都是标量值，可以通过在数组中的位置（它的下标或者偏移量）对它进行引用。让我们来看几个数组和数组常用操作的实例吧。我们定义一个叫做 @bases 的输出，其中存储着 A、C、G 和 T 四个碱基。现在我们对它应用几个常见的数组操作。

这是一个代码片段，它演示了如何初始化数组，以及如何使用下标来访问数组中的单个元素：

```
1 | # Here's one way to declare an array, initialized with a list of four scalar values.
2 | @bases = ('A', 'C', 'G', 'T');
3 |
4 | # Now we'll print each element of the array
5 | print "Here are the array elements:";
6 | print "\nFirst element: ";
7 | print $bases[0];
8 | print "\nSecond element: ";
9 | print $bases[1];
10 | print "\nThird element: ";
11 | print $bases[2];
12 | print "\nFourth element: ";
13 | print $bases[3];
```

这个代码片段将会输出：

```
1 | First element: A
2 | Second element: C
3 | Third element: G
4 | Fourth element: T
```

你可以像这样把元素一个接一个的输出出来：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | print "\n\nHere are the array elements: ";
3 | print @bases;
```

它会产生这样的输出：

```
1 | Here are the array elements: ACGT
```

你也可以输出用空格分隔的元素（注意 print 语句中的双引号）：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | print "\n\nHere are the array elements: ";
3 | print "@bases";
```

它会产生这样的输出：

```
1 | Here are the array elements: A C G T
```

使用 pop，你可以从数组的末尾拿掉一个元素：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | $base1 = pop @bases;
3 | print "Here's the element removed from the end: ";
4 | print $base1, "\n\n";
5 | print "Here's the remaining array of bases: ";
6 | print "@bases";
```

它会产生这样的输出：

```
1 | Here's the element removed from the end: T
2 |
3 | Here's the remaining array of bases: A C G
```

使用 `shift`，你可以从数组的开头拿掉一个碱基：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | $base2 = shift @bases;
3 | print "Here's an element removed from the beginning: ";
4 | print $base2, "\n\n";
5 | print "Here's the remaining array of bases: ";
6 | print "@bases";
```

它会产生这样的输出：

```
1 | Here's an element removed from the beginning: A
2 |
3 | Here's the remaining array of bases: C G T
```

使用 `unshift`，你可以把一个元素添加到数组的开头：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | $base1 = pop @bases;
3 | unshift (@bases, $base1);
4 | print "Here's the element from the end put on the beginning: ";
5 | print "@bases\n\n";
```

它会产生这样的输出：

```
1 | Here's the element from the end put on the beginning: T A C G
```

使用 `push`，你可以把一个元素添加到数组的末尾：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | $base2 = shift @bases;
3 | push (@bases, $base2);
4 | print "Here's the element from the beginning put on the end: ";
5 | print "@bases\n\n";
```

它会产生这样的输出：

```
1 | Here's the element from the beginning put on the end: C G T A
```

你可以反转数组：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | @reverse = reverse @bases;
3 | print "Here's the array in reverse: ";
4 | print "@reverse\n\n";
```

它会产生这样的输出：

```
1 | Here's the array in reverse: T G C A
```

你可以获取数组的长度：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | print "Here's the length of the array: ";
3 | print scalar @bases, "\n";
```

它会产生这样的输出：

```
1 | Here's the length of the array: 4
```

使用 Perl 的 `splice` 函数，你可以在数组的任意一个位置插入一个元素：

```
1 | @bases = ('A', 'C', 'G', 'T');
2 | splice ( @bases, 2, 0, 'X' );
3 | print "Here's the array with an element inserted after the 2nd element: ";
4 | print "@bases\n";
```

它会产生这样的输出：

```
1 | Here's the array with an element inserted after the 2nd element: A C X G T
```

4.10 标量上下文和列表上下文

依赖于使用的上下文环境不同，Perl 的许多操作符都会有不同的表现。在 Perl 中有标量上下文和列表上下文两种上下文环境。在例 4.8 对它们都进行了演示。

例 4.8：变量上下文和列表上下文

```
1 |#!/usr/bin/perl -w
2 |# Example 4-8 Demonstration of "scalar context" and "list context"
3 |
4 |@bases = ( 'A', 'C', 'G', 'T' );
5 |
6 |print "@bases\n";
7 |
8 |$a = @bases;
9 |
10| print $a, "\n";
11|
12| ($a) = @bases;
13|
14| print $a, "\n";
15|
16| exit;
```

下面是例 4.8 的输出：

```
1 | A C G T
2 | 4
3 | A
```

首先，例 4.8 语句了一个包含四个碱基的数组。然后，赋值语句尝试把数组（它是一种列表）复制给 \$a 这个标量变量：

```
1 | $a = @bases;
```

在这种标量上下文中，数组会对它的大小进行求值，也就是数组中的元素数目。语句的左边是一个标量变量，这表明是一个标量上下文。

之后，例 4.8 尝试把数组（重复一下，它是一种列表）赋值给另一个列表，在这个例子中，这个列表仅有 \$a 这么一个变量：

```
1 | ($a) = @bases;
```

在这种列表上下文中，数组会把它的元素展开成一个列表。语句的左边是包裹在括号中的列表，这表明是一个列表上下文。如果左侧没有足够变量用来赋值，那么将只有数组中的部分元素会被赋值给变量。Perl 的这种行为在很多情况下都会出现。Perl 的许多特性都被设计成在不同的上下文环境中表现出不同的行为。在附录 B 中有更多关于标量上下文和列表上文的讨论。

现在，你已经看到使用字符串和数组来保存序列和文件数据，学习了 Perl 的基本语法，包括变量、赋值、打印和读入文件。把 DNA 转录成了 RNA，还计算出了 DNA 一条链的反向互补序列。到第 5 章的结尾，你将学习到 Perl 语言编程的所有基础知识。

4.11 练习题

习题 4.1

探索编程语言对语法错误的敏感性。对于一个可以运行的程序，试着把其中任意一个语句末尾的分号删除掉，看看会产生什么样的错误信息。试着改变其他的语法项：添加一个小括号或者大括号，把“print”或其他保留字拼错，键入或者删除任何一些内容。程序员对这样的错误习以为常。即使精通语言后，在你一点点编写代码时，仍然会常常出现这样的语法错误。注意一个错误是如何导致多行错误报告的。Perl 报告的出现错误的行是不是完全准确？

习题 4.2

编写一个程序，把一个整数保存到变量中并把它打印出来。

习题 4.3

编写一个把 DNA（原始序列可以是大写或者小写）以小写形式（acgt）输出的程序；再编写一个把 DNA 以大写形式（ACGT）输出的程序。使用 *tr///* 函数。

习题 4.4

任务和练习 4.3 中的一样，但这次使用 `\U` 和 `\L` 字符串指令符来进行大小写的转换。举个例子，`print "\U$DNA"` 会把 `$DNA` 中的数据以大写形式输出出来。

习题 4.5

有时，信息也可以从 RNA 流向 DNA。编写一个把 RNA 反转录成 DNA 的程序。

习题 4.6

读取两个文件的数据，在输出第一个文件的内容后紧接着输出第二个文件的内容。

习题 4.7

这是一个更有难度的练习题。编写一个程序，读去一个文件，然后逆序输出它的每一行，也就是首先输出它的最后一行。你可能需要看一下 *push*、*pop*、*shift* 和 *unshift* 这四个函数，从中选择一个或多个来完成此练习。你可能需要预习一下第 5 章，这样就可以在程序中使用循环了。但取决于你采取的方案，这并不是必需的。或者，你可以对包含所有行的数组使用 *reverse* 函数。

第 5 章 基序和循环

目录

5.1 流程控制	62
5.2 代码布局	68
5.3 查找基序	70
5.4 计数核苷酸	76
5.5 把字符串拆解成数组	77
5.6 操作字符串	83
5.7 写入文件	87
5.8 练习题	91

本章将在第 4 章的基础上进一步介绍 Perl 语言的基础知识。到本章结束的时候，你将学会：

- 在 DNA 或蛋白质中查找基序
- 通过键盘与用户进行交互
- 把数据写入文件
- 使用循环
- 使用基本的正则表达式
- 根据条件测试的结果采取不同的行动
- 通过字符串和数组的操作对序列数据进行细致的处理

所有这些主题，加上在第 4 章学习的知识，将使你能够编写出实用的生物信息学程序。在本章中，你将学习编写一个在序列数据中查找基序的程序。

5.1 流程控制

流程控制指的就是程序中语句执行的顺序。除非明确指明不按顺序执行，否则程序将从最顶端的第一个语句开始，顺序执行到最底端的最后一个语句。有两种方式可以告诉程序不按照顺序执行：条件语句和循环。条件语句只会在条件测试成功的前提下执行相应的语句，否则就会直接跳过这些语句。循环会一直重复那些语句，直到相应的测试失败为止。


5.1.1 条件语句

让我们再看一下 *open* 语句吧。回忆一下，当你试图打开一个并不存在的文件时，你会看到错误信息。在你尝试打开文件之前，可以对文件是否存在进行明确的测试。事实上，类似的测试都是计算机语言最强大的特性之一。*if*、*if-else* 和 *unless* 条件语句就是 Perl 语言中用来进行类似测试的三个语句。

这类语句的最主要特性就是可以对条件进行测试。条件测试的结果可以是 `true` 或者是 `false`。如果测试结果为 `true`，那么后面的语句就会被执行；与之相反，如果测试结果为 `false`，这些语句就会被跳过而不执行。

那么，“什么是真呢”？对于这个问题，不同计算机语言的回答会稍有不同。

本节包含了一些演示 Perl 条件语句的实例，其中的真-假条件测试就是看看两个数字是否相等。注意，因为一个等号 `=` 是用来对变量进行赋值的，所以两个数字的相等要用两个等号 `==` 来表示。

 `=` 表示赋值、`==` 表示数字相等，这两者非常容易让人混淆，可是说是编程中最常见的“bug”了，所以一定要小心奥！

下面这个例子演示了条件测试的结果是 `true` 还是 `false`。平时你很少会用到这么简单的测试；通常，你会测试那些预先并不知道结果的值，比如读入变量的值，或者函数调用的返回值。

条件测试为 `true` 的 *if* 语句：

```
1 | if( 1 == 1 ) {
2 |     print "1 equals 1\n\n";
3 | }
```

它会输出：

```
1 | 1 equals 1
```

我们进行的测试是 `1 == 1`，用口语来说就是，“1 是否等于 1”。1 确实等于 1，所以条件测试的结果为 `true`，因此和 *if* 语句相关联的语句就会被执行，信息被打印了出来。

你也可以这样写：

```
1 | if( 1 ) {
2 |     print "1 evaluates to true\n\n";
3 | }
```

它会输出：

```
1 | 1 evaluates to true
```

条件测试为 `0` 的 `if` 语句：

```
1 | if( 1 == 0 ) {  
2 |     print "1 equals 0\n\n";  
3 | }
```

它没有任何输出！我们进行的测试是 `1 == 0`，用口语来说就是，“1 是否等于 0”。1 当然不等于 0 了，所以条件测试的结果为 `0`，因此和 `if` 语句相关联的语句就不会被执行，不会输出任何信息。

你也可写成：

```
1 | if( 0 ) {  
2 |     print "0 evaluates to true\n\n";  
3 | }
```

因为 0 的测试结果为 `0`，所以它不会产生任何输出，因此和 `if` 语句相关联的语句会完全被跳过。

还有一种编写短小的 `if` 语句的方法，它模拟的是英语中 `if` 的用法。在英语中，你可以说，“If you build it, they will come”，或者，“They will come if you build it”，这两种说法是完全等同的。Perl 也不甘示弱，它也允许你把 `if` 放在对应的动作之后：

```
1 | print "1 equals 1\n\n" if (1 == 1);
```

它和本节中的第一个例子做的事情是完全一样的，输出：

```
1 | 1 equals 1
```

现在，让我们看一下条件测试为 `0` 的 `if-else` 语句：

```
1 | if( 1 == 1 ) {  
2 |     print "1 equals 1\n\n";  
3 | } else {  
4 |     print "1 does not equal 1\n\n";  
5 | }
```

它会输出：

```
1 | 1 equals 1
```

当测试结果为 `0` 时，`if-else` 会做某件事情，当测试结果为 `0` 时，它则会另一件事情。这是条件测试为 `0` 的 `if-else` 语句：

```
1 | if( 1 == 0 ) {  
2 |     print "1 equals 0\n\n";  
3 | } else {
```

```

4 | print "1 does not equal 0\n\n";
5 | }

```

它会输出：

```

1 | 1 does not equal 0

```

最后的例子是 `unless`，它和 `if` 完全相反。它就像英语中的“unless”那样来使用：比如，“Unless you study Russian literature, you are ignorant of Chekov”¹。如果测试结果为 `0`，它不会采取任何行动；而当测试结果为非 `0` 时，相应的语句就会被执行。如果“you study Russian literature”是非 `0` 的，那么“you are ignorant of Chekov”²。

```

1 | unless( 1 == 0 ) {
2 |     print "1 does not equal 0\n\n";
3 | }

```

它会输出：

```

1 | 1 does not equal 0

```

条件测试和括号成对

对这些语句和它们的条件测试，还有两点注释：

第一点，在这些语句的条件部分，有许多测试可以使用。除了在前述实例中使用的数字相等 `==` 外，你还可以对不等 `!=`、大于 `>` 和小于 `<` 等进行测试。

与之类似，你也可以使用 `eq` 操作符对字符串相等进行测试：如果两个字符串是一样的，测试结果就是 `0`。也有一些文件测试操作符，允许你对文件进行相应的测试：是否存在，是否为空，是否设置了特定的权限，等等（参看附录 B）。另一个比较常用的是对变量名进行测试：如果变量存储的是 `0`，那它就被认为是非 `0` 的；任何其他数字都被认为是非 `0` 的。如果变量中有非空的字符串，那它就是非 `0` 的；空的字符串用 `""` 或者 `' '` 来表示，它是非 `0` 的。

第二点，注意条件测试之后的语句都被包裹在了成对的大括号内。用大括号括起来的语句叫做块，这在 Perl 中非常常见。³ 括号成对出现是最普遍的编程特性，比如成对的小括号、中括号、尖括号和大括号：`()`、`[]`、`<>` 和 `{}`。左右括号的数目相等，且它们都出现在正确的位置，这对于 Perl 程序正常运行来说是至关重要的。

成对使用括号是很难做到的，所以，当你因为少写了括号而导致程序报错时，请不要大惊小怪。这是一个比较常见的语法错误，你需要查阅代码找到缺少括号的地方并把它添加上。随着代码越来越复杂，要找到成对括号出错的地方并对它进行修正会更加具有挑战性。即使括号都在正确的位置，当你阅读代码时，要分清哪些语句是一组也是非常困难的。当然你也可以采取一定的措施来避免这样的问题：每一行代码做的事情不要太多，使用缩进让代码块更加突出明显一些（参看第 5.2 节）。⁴

¹译者注：除非你研究俄国文学，否则你不会契诃夫有所了解。

²译者注：如果“你研究俄国文学”是非 `0` 的，那么“你对契诃夫毫无了解”。

³虽然这看起来有些奇怪，但块中的最后一个语句并不需要用分号来结尾。

⁴有些文本编辑器可以帮你找到配对的括号（举个例子，在 vi 编辑器中，在一个括号上键入百分号 `%` 就会让光标跳跃到与之配对的括号上去）。

回过头来再看看条件语句。就像例 5.1 中演示的那样，*if-else* 还有 *if-elsif-else* 这样的形式。第一个 *if* 和 *elsifs* 中的条件被交替测试，一旦测试结果为真，相应的代码块就会被执行，并且剩余的条件测试也会被忽略掉。假设没有一个条件的测试结果为真，如果存在 *else* 代码块，那它就会被执行。*else* 代码块是可有可无的。

例 5.1: if-elsif-else

```
1  #!/usr/bin/perl -w
2  # Example 5-1    if-elsif-else
3
4  $word = 'MNIDDKL';
5
6  # if-elsif-else conditionals
7  if ( $word eq 'QSTVSGE' ) {
8
9      print "QSTVSGE\n";
10
11 }
12 elsif ( $word eq 'MRQQDMISHDEL' ) {
13
14     print "MRQQDMISHDEL\n";
15
16 }
17 elsif ( $word eq 'MNIDDKL' ) {
18
19     print "MNIDDKL--the magic word!\n";
20
21 }
22 else {
23
24     print "Is \"$word\" a peptide? This program is not sure.\n";
25
26 }
27
28 exit;
```

注意 *else* 代码块的 *print* 语句中的 `\`，它可以在双引号包裹起来的字符串中打印出双引号 (")。反斜线告诉 Perl，把后面紧跟的 `"` 当成引号本身，而不是把它当做字符串结束的标志。此外还请注意使用了 *eq* 来检测字符串的相等。

例 5.1 的输出：

```
1  MNIDDKL--the magic word!
```

5.1.2 循环

循环允许你重复执行被成对大括号包裹起来的语句块。在 Perl 中有多种方法实现循环：*while* 循环、*for* 循环、*foreach* 循环等等。例 5.2（来源于第 4 章）演示了如何使用 *while* 循环从一个文件读取蛋白质序列数据。

例 5.2：从文件中读取蛋白质序列数据，第四次尝试

```

1  #!/usr/bin/perl -w
2  # Example 5-2   Reading protein sequence data from a file, take 4
3
4  # The filename of the file containing the protein sequence data
5  $proteinfilename = 'NM_021964fragment.pep';
6
7  # First we have to "open" the file, and in case the
8  # open fails, print an error message and exit the program.
9  unless ( open( PROTEINFILE, $proteinfilename ) ) {
10
11      print "Could not open file $proteinfilename!\n";
12      exit;
13  }
14
15  # Read the protein sequence data from the file in a "while" loop,
16  # printing each line as it is read.
17  while ( $protein = <PROTEINFILE> ) {
18
19      print " ##### Here is the next line of the file:\n";
20
21      print $protein;
22  }
23
24  # Close the file.
25  close PROTEINFILE;
26
27  exit;

```

下面是例 5.2 的输出：

```

1  ##### Here is the next line of the file:
2  MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
3  ##### Here is the next line of the file:
4  SVLQDRSMPHQEILAADEVVLQESEMRQQDMISHDELMVHEETVKNDDEEQMETHERLPQ
5  ##### Here is the next line of the file:
6  GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR

```

在 *while* 循环中，注意在循环至文件的下一行时 *\$protein* 变量是如何一次次被赋值的。在 Perl 中，赋值语句会返回赋予的值。此处，条件测试检测的是赋值语句是否成功得读取了另一行。如果有另一行被读入，就会发生赋值行为，条件测试就为 `1`，新的一行会被存储在 *\$protein* 变量中，而包含两个 *print* 语句的代码块也会被执行。如果没有更多行了，赋值就会是未定义，条件测试就为 `0`，而程序也会跳过包含两个 *print* 语句的代码块，退出 *while* 循环，继续执行程序的后部分（在这个例子中，就是 *close* 和 *exit* 函数）。

open 和 unless

open 是一个系统调用，因为要打开一个文件，Perl 必须向操作系统提出请求。操作系统可以是 Unix 或 Linux、Microsoft Windows 或 Apple Maintosh 等的某个版本。文件被操作系统管理者，因此也只有操作系统能够访问它。

检查系统调用的成功与否是一个好习惯，尤其是当打开文件时更是如此。如果系统调用失败了，而你也没有对它进行检查，那么程序将继续运行，可能会尝试读取或写入一个你并没有打开过的文件。你应该总是检查这种失败，当文件无法打开时，要立即告知程序的使用者。当打开文件失败时，通常情况下你会希望立即退出程序，并尝试打开另一个文件。

在例 5.2 中，*open* 系统调用是 *unless* 条件测试的一部分。

unless 和 *if* 相反。就像在你英语中你可以说 “do the statements in the block if the condition is true”（“如果条件为真，就执行代码块中的语句”）一样，你也可以反过来说 “do the statements in the block unless the condition is true”（“除非条件为真，否则就执行代码块中的语句”）⁵。如果成功打开了文件，*open* 系统调用就会返回真值；在这个例子的 *unless* 语句条件测试中，如果 *open* 系统调用失败了，那么代码块中的语句就会被执行，程序会输出错误信息，随后退出。

总结一下，在 Perl 中，条件和循环都是比较简单的概念，学起来并不困难。它们是编程语言最强大的特性之一。条件可以使你的程序适应不同的状况，使用这种方式，就可以针对不同的输入采取不同的方案了。在人工智能的计算机程序中，它占了相当大的比重。循环充分利用了计算机的速度，利用循环，仅仅使用几行代码，你就可以处理大量的输入，或者对计算进行重复迭代与提炼。

⁵译者注：根据语境，此处应为：“do the statements in the block unless the condition is false”（“除非条件为假，否则就执行代码块中的语句”）。

5.2 代码布局

一旦你开始使用循环和条件语句，你就需要认真考虑格式的问题了。当格式化 Perl 代码时你有多种选择。对于 `while` 循环中的 `if` 语句，比较一下下面几种不同的格式化方法：

格式 A

```
1 | while ( $alive ) {  
2 |     if ( $needs_nutrients ) {  
3 |         print "Cell needs nutrients\n";  
4 |     }  
5 | }
```

格式 B

```
1 | while ( $alive )  
2 | {  
3 |     if ( $needs_nutrients )  
4 |     {  
5 |         print "Cell needs nutrients\n";  
6 |     }  
7 | }
```

格式 C

```
1 | while ( $alive )  
2 | {  
3 |     if ( $needs_nutrients )  
4 | {  
5 |     print "Cell needs nutrients\n";  
6 | }  
7 | }
```

格式 D

```
1 | while($alive){if($needs_nutrients){print "Cell needs nutrients\n";}}
```

对于 Perl 解释器来说，所有这些代码片段都是一样的，因为 Perl 并不依赖于每一行中语句的布局方式，它只关心句法元素的正确顺序。有些元素在它们中间需要空白（比如空格、制表符或者换行符），这样就可以把它们区别开了，但通常来说，对于你使用空白来布局代码的方式，Perl 不会进行特别的限定。

格式 A 和 B 是布局代码最常见的方式。它们都使程序结构更加清晰，便于人们阅读。对于 `while` 和 `if` 这种附带着代码块的语句来说，注意观察是如何排列大括号、以及缩进代码块中的语句的。这种布局使得语句附带的代码块的范围更加清晰一些。（这对于长的、复杂的代码块是非常关键的。）代码块中的语句都进行了缩进，通常你使用制表符键或者四个或八个空格就可以实现这种缩进。（许多文本编辑器可以使你在敲击制表符键时输入空格，或者你可以对它们进行配置，使得用制表符键就可以输入四个、八个或者任意数目的空格。）使用这种方式，程序的总体结构变得更加清晰了，你可以轻而易举得看出那些

语句是归类在循环或条件语句附带的代码块中的。就个人而言，尽管我也非常乐意使用格式 B，但我更加喜欢格式 A 的布局。

格式 C 是格式化代码的反面教材。代码的流控制很不清晰，举个例子，很难看出 *print* 语句是否在 *while* 语句的代码块中。

格式 D 演示了，在对代码不进行任何格式化的情况下，即使对于这样一个简单的代码片段来说，要阅读它是多么的困难。

从 Perl 的手册中也可以找到 Perl 的风格指南，或者输入下面的命令也可以：

```
1 | perldoc perlstyle
```

对于如何编写可读性好的代码，Perl 风格指南给出了不少的建议和意见。然而，这些并不是规则，你可以使用在经过众多格式化练习后找到的最适合你自己的代码布局方案。

5.3 查找基序

在生物信息学中我们最常做的事情之一就是查找基序，即特定的 DNA 或蛋白质片段。这些基序可能是 DNA 的调控元件，也可能是已知在多个物种中保守的蛋白质片段。（PROSITE 网站——<http://www.expasy.ch/prosite/>——有关于蛋白质基序的更多信息。）

在生物学序列中你要查找的基序往往不是特定的一个序列，它们可能有一些变体——比如，某个位置上的碱基或者残基是什么并不重要。它们也有可能有不同的长度。但通常它们都可以用正则表达式来进行表征，在接下来的例 5.3、第 9 章以及本书中的许多地方，你都可以看到更多关于正则表达式的讨论。

Perl 有一系列在字符串中进行查找的便利的特性，这同样使得 Perl 成为生物信息学领域中流行的语言。例 5.3 对这种字符串搜索的能力进行了演示；它做的事情真的非常实用，许多类似的程序在生物科学研究中一直被使用着。它做的事情包括：

- 从文件中读入蛋白质序列数据
- 为了便于搜索，把所有的序列数据都放到一个字符串中
- 查找用户通过键盘键入的基序

例 5.3：查找基序

```
1  #!/usr/bin/perl -w
2  # Example 5-3   Searching for motifs
3
4  # Ask the user for the filename of the file containing
5  # the protein sequence data, and collect it from the keyboard
6  print "Please type the filename of the protein sequence data: ";
7
8  $proteinfilename = <STDIN>;
9
10 # Remove the newline from the protein filename
11 chomp $proteinfilename;
12
13 # open the file, or exit
14 unless ( open( PROTEINFILE, $proteinfilename ) ) {
15
16     print "Cannot open file \"$proteinfilename\"\n\n";
17     exit;
18 }
19
20 # Read the protein sequence data from the file, and store it
21 # into the array variable @protein
22 @protein = <PROTEINFILE>;
23
24 # Close the file - we've read all the data into @protein now.
25 close PROTEINFILE;
26
27 # Put the protein sequence data into a single string, as it's easier
28 # to search for a motif in a string than in an array of
29 # lines (what if the motif occurs over a line break?)
```

```
30 $protein = join( ' ', @protein );
31
32 # Remove whitespace
33 $protein =~ s/\s//g;
34
35 # In a loop, ask the user for a motif, search for the motif,
36 # and report if it was found.
37 # Exit if no motif is entered.
38 do {
39     print "Enter a motif to search for: ";
40
41     $motif = <STDIN>;
42
43     # Remove the newline at the end of $motif
44
45     chomp $motif;
46
47     # Look for the motif
48
49     if ( $protein =~ /$motif/ ) {
50
51         print "I found it!\n\n";
52
53     }
54     else {
55
56         print "I couldn't find it.\n\n";
57     }
58
59     # exit on an empty user input
60 } until ( $motif =~ /^s*$/ );
61
62 # exit the program
63 exit;
```

这是例 5.3 典型的一个输出：

```
1 | Please type the filename of the protein sequence data:
2 | NM_021964fragment.pep
3 | Enter a motif to search for: SVLQ
4 | I found it!
5 |
6 | Enter a motif to search for: jkl
7 | I couldn't find it.
8 |
9 | Enter a motif to search for: QDSV
10 | I found it!
11 |
12 | Enter a motif to search for: HERLPQGLQ
```

```
13 | I found it!
14 |
15 | Enter a motif to search for:
16 | I couldn't find it.
```

就像你在结果中看到的那样，这个程序查找用户通过键盘键入的基序。利用这样一个程序，你就不用再在海量的数据中手工进行查找了。让计算机来做这样的工作，它比人工做的更快、更准确。

如果程序不仅报告它找到了基序，还能告诉我们在哪里找到了它，那就更好了。在第 9 章中你将看到如何实现这个目标，那一章的一个练习要求你对这个程序进行修改，使得它能够报告基序的位置。

接下来的小节将对例 5.3 中这些新的内容进行介绍与讨论：

- 获取用户的键盘输入
- 把文件的所有行合并到一个标量变量中
- `do-until` 循环
- 正则表达式和字符组
- 模式匹配

5.3.1 获取用户的键盘输入

在例 4.5 中你第一次遇到文件句柄。在例 5.3 中（在例 4.3 中也是这样），使用文件句柄和尖括号输入操作符从打开的文件中读入数据并保存到数组中，就像这样：

```
1 | @protein = <PROTEINFILE>;
```

Perl 使用同样的语法获取用户的键盘输入。在例 5.3 中，为了获取用户的键盘输入，使用了一个叫做 STDIN（standard input（标准输入）的简写）的特殊的文件句柄，就像这行从用户键盘输入获取文件名的代码一样：

```
1 | $proteinfilename = <STDIN>;
```

文件句柄可以和一个文件相关联，也可以和用户回答程序问题时的键盘输入相关联。

就像这个代码片段一样，如果你用来存储输入的变量是一个以美元符号（\$）起始的标量变量，那么将只有一行被读入，这往往也是在这种情况下你希望的结果。

在例 5.3 中，用户被要求输入一个包含蛋白质序列数据的文件的文件名。在通过这种方式得到文件名后，打开文件之前还需要一步操作。当用户键入文件名并通过 Enter 键（也叫做 Return 键）换行时，文件名和其末尾的换行符一起被保存进了变量中。这个换行符并不是文件名的一部分，所以在使用 *open* 系统调用之前应该把它去除掉。Perl 函数 *chomp* 会去除掉字符串末尾的换行符 *newlines*（或者它的表亲换行符 *linefeeds* 和回车符 *carriage returns*）。（更加古老的 *chop* 函数会删除最后一个字符，不管这个字符是什么；这会导致一些问题，所以 *chomp* 被引入了进来，而它也总是被优先考虑使用的。）

所以这一个部分还有一点附带的知识：删除获取的用户键盘输入中的换行符。试着把 *chomp* 函数注释掉，你会看到 *open* 调用失败，因为并没有末尾包含换行符的文件名。（对于文件名中可以使用的字符，操作系统有自己的规定。）

5.3.2 使用 join 把数组合并成标量

常常可以发现，蛋白质序列数据被打断成了短的片段，每个片段有大约 80 个字符。原因很简单：当要把数据打印到纸张上或展示在屏幕上时，根据纸张或屏幕的空间需要把它打断成数行。你的数据被打断成了片段，但对你的 Perl 程序来说却多有不便。当你要查找一个基序，而它却被换行符分割开了，这会怎样呢？你的程序将无法找到这个基序。事实上，例 5.3 中查找的一些基序确实被换行符分割开了。在 Perl 中，你可以使用 *join* 函数来处理类似的片段数据。在例 5.3 中，*join* 把存储在 `@protein` 数组中的数据的各行合并成一个单独的字符串，并把它保存在了新的标量变量 `$protein` 中：

```
1 | $protein = join( ' ', @protein );
```

当合并数组时，你需要指定一个字符串，这个字符串会放在数组的各个元素之间。在这个例子中，你指定的是空字符串，它会放在输入文件的各行之间。空字符串就夹在成对的单引号 `' '` 中（当然也可以使用双引号 `""`）。

回忆一下例 4.2，在那个例子中，我介绍了好几种连接两个 DNA 片段的方法。*join* 函数的作用与之类似，它取出数组元素的变量值并把它们连接成一个单独的标量值。回忆一下例 4.2 中的下列语句，它是连接两个字符串的方法之一：

```
1 | $DNA3 = $DNA1 . $DNA2;
```

另一种实现同样连接的方法是使用 *join* 函数：

```
1 | $DNA3 = join( "", ($DNA1, $DNA2) );
```

在这个例子中，我没有给出数组名，而是指定了一个标量元素的列表：

```
1 | ($DNA1, $DNA2)
```

5.3.3 do-until 循环

在例 5.3 中有一种新的循环，这就是 *do-until* 循环，它首先执行一个代码块，之后再行条件检测。有时它会比常见的先测试、如果测试成功就执行代码块的策略更加方便。在这个例子中，你想提示用户，获取用户的输入，查找基序并报告查找结果。在重复这些动作之前，你进行条件测试来看看用户是否输入了一个空行。输入空行意味着用户没有更多的基序来查找了，所以你就退出循环。

5.3.4 正则表达式

正则表达式使你可以轻松处理各种各样的字符串，比如 DNA 和蛋白质序列数据。正则表达式的强大之处就在于，如果想对字符串进行一些处理，你通常可以使用 Perl 的正则表达式来完成该任务。

有些正则表达式非常简单。举个例子，你可以把你想搜索的具体文字当成正则表达式来使用：如果我想在本书的文字中查找“bioinformatics”（“生物信息学”），我可以使用这样的正则表达式：

```
1 | /bioinformatics/
```

然而，有些正则表达式会更加复杂。在本节中，我会通过例 5.3 来演示它们的使用。

正则表达式和字符组

正则表达式使用特定的类似于通配符的操作符来匹配一个或多个字符串。正则表达式可以非常简单——一个单词，它匹配单词本身，也可以非常复杂，可以匹配一个大的单词集合（甚至每一个单词！）。

在例 5.3 中，当你把蛋白质序列数据合并到 `$protein` 标量变量中后，你还需要删除掉换行符和其他所有不是序列数据的字符。这可能会包括行中的数字、注释、信息行或标题行，等等。在这个例子中，你需要删除掉换行符和空格或制表符等那些存在其中的非打印字符。下面这行例 5.3 中的代码会删除空白：

```
1 | $protein =~ s/\s//g;
```

标量变量 `$protein` 中的序列数据会被这个语句所改变。在例 4.3 中你首次看到绑定操作符 `=` 和替换函数 `s///`，那时它们被用来把一个字符替换成另一个字符。而此处，它们的使用则有少许的不同。你把用 `\s` 表示的空白字符组中的任意一个字符替换成了空字符串，第二个和第三个正斜线之间什么都没有表示的就是空字符串。换言之，你删除了空白字符组中的任意一个字符，这是对字符串进行的全局操作，这是通过语句末尾的 `g` 来实现的。

`\s` 是众多元字符中的一个。你已经见过了 `\n` 元字符。`\s` 元字符匹配空格、制表符、换行符、换页符和回车符中的任意一个。`\s` 也可以写成：

```
1 | [ \t\n\f\r]
```

这个表达式是字符组的一个例子，用中括号包裹起来。字符组匹配中括号里的字符中的任意一个。空格就用空格来表示，而其他空白字符则有自己的元字符：制表符用 `\t` 表示，换行符用 `\n` 表示，换页符用 `\f` 表示，回车符用 `\r` 表示。回车符使得下一个字符被写在行首，而换页符则会转换到下一行。两者结合起来实际上就是换行符的作用。

我提到的每一个 `s///` 命令中都有一定形式的正则表达式，它位于前两个正斜线 `/` 之间。在 `s/C/G/g` 中处在该位置的是 `C` 这样的单个字母。`C` 是有效正则表达式的一个例子。

在例 5.3 中，也使用了正则表达式。下面这行代码：

```
1 | } until ( $motif =~ /^s*$/ );
```

用口语来说（用英语来说），就是检测 `$motif` 变量中的空行。如果用户没有任何输入，或者只是输入了一些空白字符，空白字符用 `\s*` 来表示，该匹配就会成功，程序则会退出。完整的正则表达式是：

```
1 | /^s*$/
```

把它翻译过来就是：匹配这样的字符串，从开头（用 `^` 表示）到结尾（用 `$` 表示）只有零个或多个（用 `*` 表示）空白字符（用 `\s` 表示）。

如果觉得这些晦涩难懂，那就先别管它们了，很快你就会对这些术语越来越熟悉了。正则表达式是处理序列和其他基于文本的数据的很好的方法，而 Perl 则对正则表达式尤其擅长，使得它们易用、强大且灵活。附录 A 中的许多参考资料都有关于正则表达式的相关内容，而在附录 B 中则有一个简短的总结。

使用 =~ 和正则表达式进行模式匹配

真正的查找基序是例 5.3 中的这一行：

```
1 | if ( $protein =~ /$motif/ ) {
```

此处，绑定操作符 =~ 在蛋白质 \$protein 中查找存储在 \$motif 变量值中的正则表达式。通过这种特性，你可以把变量的值内插到字符串匹配中。（Perl 字符串中的内插指的是把变量的值插入到字符串中，就像当你要把字符串连接起来时，在例 4.2 中首次看到的那样）。真正的基序，也就是字符串变量 \$motif 的值，就是你的正则表达式。这些最简单的正则表达式就是由一些字符组成的字符串，就像 AQQK 基序一样。

你也可以使用例 5.3 来探索正则表达式的更多特性。你可以键入任意的正则表达式，在蛋白质中进行查找。参考正则表达式的文档，运行程序，去探索吧！这里是键入的正则表达式的一些例子：

- 查找 A、紧跟 D 或 S、之后紧跟 V 的基序：

```
1 | Enter a motif to search for: A[DS]V
2 | I couldn't find it.
```

- 查找 K、N、零个或多个 D、两个或更多个 E（注意 {2,} 表示两个或更多）的基序：

```
1 | Enter a motif to search for: KND*E{2,}
2 | I found it!
```

- 查找两个 E、紧跟着任意字符、之后紧跟另外两个 E 的基序：

```
1 | Enter a motif to search for: EE.*EE
2 | I found it!
```

在最后一个查找中，注意点号表示除换行符以外的任意字符，“.” 代表零个或多个这样的字符。（如果你想匹配点号本身，需要使用反斜线对它进行转义。）

5.4 计数核苷酸

对于一段 DNA 序列，有许多信息你可能想知道。它是编码的还是非编码的？⁶它是否含有调控元件？它是否和其他已知的 DNA 相关，如果相关，是怎么相关呢？DNA 中四种核苷酸的数目各是多少？事实上，在许多物种中，编码区域都有特定的核苷酸偏向性，所以，最后这个问题对于基因识别来说是非常重要的。此外，不同的物种有不同的核苷酸使用模式。所以对核苷酸进行计数不但有趣而且非常有用。

接下来的小节中有两个程序，例 5.4 和例 5.6，它们对 DNA 序列中每种核苷酸都进行计数。这两个程序展示了 Perl 的一些新的知识：

- “拆解”字符串
- 查看字符串的特定位置
- 对数组进行迭代
- 对字符串的长度进行迭代

为了对 DNA 中的每一种核苷酸进行计数，你需要查看每一个碱基，看看它是什么，并且要为每一种碱基记录一个计数，一共四个计数。我们将通过两种方法来实现：

- 把 DNA 拆解成单个碱基，存储到数组中，然后对数组进行迭代（换言之，一个接一个的对数组元素进行处理）
- 在计数时使用 *substr* 这个 Perl 函数对 DNA 字符串的位置进行迭代

首先，让我们从该任务的伪代码开始。之后，我们会给出更加详细的伪代码，最终，编写出这两种方案的 Perl 程序。

下面的伪代码描述了基本的要求：

```
1 | for each base in the DNA
2 |   if base is A
3 |     count_of_A = count_of_A + 1
4 |   if base is C
5 |     count_of_C = count_of_C + 1
6 |   if base is G
7 |     count_of_G = count_of_G + 1
8 |   if base is T
9 |     count_of_T = count_of_T + 1
10| done
11|
12| print count_of_A, count_of_C, count_of_G, count_of_T
```

就像你看到的一样，这是一个非常简单的想法，它模拟了你手工计数时的工作。（如果你想计算所有人类基因中碱基的相对频率，手工计数就不现实了，因为数量太过巨大，你将不得不使用类似的程序。这就是生物信息学。）现在，让我们看看如何用 Perl 来实现它吧。

⁶编码 DNA 是编码蛋白质的 DNA，也就是说，它是基因的一部分。在包括人类的许多生物中，大部分 DNA 是不编码的——它不是基因的一部分，也不编码蛋白质。在人类基因组中，大约 98-99% 的 DNA 是非编码的。

5.5 把字符串拆解成数组

假设你打算把 DNA 字符串拆解成数组。所谓拆解，我指的是字符串中的每一个字母分离开来，就像把字符串分离成 bit 一样。换句话说，DNA 字符串中表示碱基的字母都被分离开，每一个字母都将成为数据中的标量值。然后一个接一个的查看数组元素（每一个都是一个单独的字符），这样你就可以进行计数了。这与第 5.3.2 小节中的 `join` 函数是相反的，它把数组中的字符串合并成单个的标量值。（在把字符串拆解成数组后，如果你愿意，可以使用 `join` 再把这个数组合并成和原来一模一样的字符串。）

在刚才的伪代码中，我将加上下面的这些操作指令：从文件中获取 DNA，操作文件数据使 DNA 序列成为单一的字符串。所以，首先，你把包含原始文件数据行的数组合并起来，通过删除空白把它清理干净，直到只有序列保留下来为止，然后，把它拆解成数组。当然，关键的一点就是最后的这个数组就是我们所需要的，使用它可以在循环中方便地进行计数。与包含行、换行符和其他可能的无用的字符的数组不同，这个数组就是包含单个碱基的数组。

```
1 | read in the DNA from a file
2 |
3 | join the lines of the file into a single string $DNA
4 |
5 | # make an array out of the bases of $DNA
6 | @DNA = explode $DNA
7 |
8 | # initialize the counts
9 | count_of_A = 0
10 | count_of_C = 0
11 | count_of_G = 0
12 | count_of_T = 0
13 |
14 | for each base in @DNA
15 |
16 |     if base is A
17 |         count_of_A = count_of_A + 1
18 |     if base is C
19 |         count_of_C = count_of_C + 1
20 |     if base is G
21 |         count_of_G = count_of_G + 1
22 |     if base is T
23 |         count_of_T = count_of_T + 1
24 | done
25 |
26 | print count_of_A, count_of_C, count_of_G, count_of_T
```

如前所述，这里的伪代码更加详细一些。通过把 DNA 字符串拆解成包含单个字符的数组，它提供了一种查看每个碱基的办法。为了保证计数正确，它还把所有的计数都初始化为 0。如果你理解了程序中的初始化，就很容易看出具体发生了什么，这可以防止代码中出现某些错误。（然而，这并不是规定。有时，你可能更加偏好直到使用标量时，一直使

它的值保持为未定义。) 如果你尝试把变量当做数值来使用, 比如把它和另一个数字相加, Perl 会假定这个未初始化的变量的值为 0。但是这种情况下你通常会得到一个警告信息。

我们已经设计好了程序, 现在就把它转化成 Perl 代码。例 5.4 是一个可以工作的程序。随着本章的学习, 你将看到其他完成该任务的方法, 它的速度会更快一些, 但此处速度并不是我们关注的焦点。

例 5.4: 确定核苷酸的频率

```
1  #!/usr/bin/perl -w
2  # Example 5-4   Determining frequency of nucleotides
3
4  # Get the name of the file with the DNA sequence data
5  print "Please type the filename of the DNA sequence data: ";
6
7  $dna_filename = <STDIN>;
8
9  # Remove the newline from the DNA filename
10 chomp $dna_filename;
11
12 # open the file, or exit
13 unless ( open( DNAFILE, $dna_filename ) ) {
14
15     print "Cannot open file \"$dna_filename\"\n\n";
16     exit;
17 }
18
19 # Read the DNA sequence data from the file, and store it
20 # into the array variable @DNA
21 @DNA = <DNAFILE>;
22
23 # Close the file
24 close DNAFILE;
25
26 # From the lines of the DNA file,
27 # put the DNA sequence data into a single string.
28 $DNA = join( '', @DNA );
29
30 # Remove whitespace
31 $DNA =~ s/\s//g;
32
33 # Now explode the DNA into an array where each letter of the
34 # original string is now an element in the array.
35 # This will make it easy to look at each position.
36 # Notice that we're reusing the variable @DNA for this purpose.
37 @DNA = split( '', $DNA );
38
39 # Initialize the counts.
40 # Notice that we can use scalar variables to hold numbers.
41 $count_of_A = 0;
```

```

42 $count_of_C = 0;
43 $count_of_G = 0;
44 $count_of_T = 0;
45 $errors      = 0;
46
47 # In a loop, look at each base in turn, determine which of the
48 # four types of nucleotides it is, and increment the
49 # appropriate count.
50 foreach $base (@DNA) {
51
52     if ( $base eq 'A' ) {
53         ++$count_of_A;
54     }
55     elsif ( $base eq 'C' ) {
56         ++$count_of_C;
57     }
58     elsif ( $base eq 'G' ) {
59         ++$count_of_G;
60     }
61     elsif ( $base eq 'T' ) {
62         ++$count_of_T;
63     }
64     else {
65         print "!!!!!!! Error - I don't recognize this base: $base\n";
66         ++$errors;
67     }
68 }
69
70 # print the results
71 print "A = $count_of_A\n";
72 print "C = $count_of_C\n";
73 print "G = $count_of_G\n";
74 print "T = $count_of_T\n";
75 print "errors = $errors\n";
76
77 # exit the program
78 exit;

```

为了演示例 5.4，我创建了下面这个小的 DNA 文件，把它命名为 *small.dna*：

```

1 | AAAAAAAAAAAAAAGGGGGGTTTTCCCCCCCC
2 | CCCCCGTCGTAGTAAAGTATGCAGTAGCVG
3 | CCCCCCCCCCGGGGGGGGAAAAAAAAAAAAATTTTTTAT
4 | AAACG

```

你可以使用钟爱的文本编辑器在计算机上键入 *small.dna* 文件，也可以从本书的网站上下载它。

注意文件中有一个 V，这是一个错误。⁷下面是例 5.4 的输出：

⁷DNA 序列数据的文件有时会包含 N 这样的字符，它表示“未知碱基”，以及其他特定的字符。有时你不

```
1 | Please type the filename of the DNA sequence data: small.dna
2 | !!!!!!!! Error - I don't recognize this base: V
3 |
4 | A = 40
5 | C = 27
6 | G = 24
7 | T = 17
8 | errors = 1
```

现在来看看程序中出现的新的知识。打开和读取序列数据与前面的程序是一样的。第一个新知识出现在这一行中：

```
1 | @DNA = split( ' ', $DNA );
```

它的注释解释说它会把 `$DNA` 拆解成包含单个字符的 `@DNA` 数组。

`split` 与 `join` 是相对的，最好花些时间去看看这两个命令的文档。使用 `split` 时，如果第一个参数是空字符串，将会把字符串拆解成单个的字符，而这正是我们所需要的。⁸

接下来，有五个标量变量被初始化为了 0，就是 `$count_of_A` 等变量。所谓初始化就是给它赋一个初始值，在这个例子中，就是 0。

例 5.4 阐明了类型和初始化的概念。一个变量的类型决定了它可以存储的数据类型，比如，字符串或数字。到现在为止，我们使用 `$DNA` 这样的标量变量来存储由 A、C、G 和 T 字母构成的字符串。例 5.4 演示了你也可以用标量变量来存储数字。比如，`$count_of_A` 变量就记录着字符 A 的计数。

标量变量可以存储整数（0、1、-1、2、-2、……），如 6.544 这样的小数或浮点数，如 6.544E6 这样用科学计数法表示的数字，它可以转换成 6.544×10^6 或者 6,544,000。（附录 B 中有更多关于数字类型的介绍。）

在例 5.4 中，从 `$count_of_A` 到 `$count_of_T` 的变量都被初始化为 0。初始化一个变量意味着在声明该变量后给它一个值。如果你不初始化你的变量，它的值将被假定为 'undef'。在 Perl 中，对于一个未定义的变量，如果在数字上下文中使用时它的值为 0，在字符串操作中使用它就是空字符串。尽管 Perl 程序员通常不去初始化变量，但对于其他的语言来说这确实是一个关键的步骤。比如，在 C 中，未初始化的变量就会有不可预料的值，这可能会导致莫名其妙的输出。你应当养成初始化变量的习惯，这会使得程序更容易阅读和维护，这非常重要。

所谓声明变量就是指变量的名字及其它属性，如初始值和作用域（对于作用域，请参看第 6 章以及 `my` 变量的讨论）。许多语言要求你在使用变量之前先声明它们。对于本书来说，到目前为止，声明还并不是必需的。从下一章开始将需要声明。在 Perl 中，除了初始化变量的值外，你还可以声明一个变量的作用域（参看第 6 章以及 `my` 变量的讨论）。许多语言还要求你声明变量的类型，比如“整数”或“字符串”，但 Perl 并不要求你这么

得不去查看一下数据来源的文档，比如 ABI 测序仪或 GenBank 文件等，看看它们使用了那些字符，各代表什么含义。

⁸就像你在 `split` 函数的文档中所看到的那样，第一个参数可以使任何的正则表达式，比如 `/\s+/`（一个或多个相邻的空白字符）。

Perl 在判断标量变量是什么时非常智能。比如，你可以把数字 1234（没有引号）赋值给变量，也可以把字符串 '1234'（有引号）赋值给它。当打印输出时，Perl 把变量当成字符串来处理，而当在算术运算中使用时 Perl 则会把它看做数字，所有这些你都不需担心。例 5.5 演示了 Perl 的这种能力。换句话说，在指定变量的数据类型时，Perl 并不是很严格。

例 5.5：演示 Perl 对数字和字符串的智能化处理

```

1 |#!/usr/bin/perl -w
2 |# Example 5-5   Demonstration of Perl's built-in knowledge about numbers and strings
3 |
4 |$num = 1234;
5 |
6 |$str = '1234';
7 |
8 |# print the variables
9 |print $num, " ", $str, "\n";
10 |
11 |# add the variables as numbers
12 |$num_or_str = $num + $str;
13 |
14 |print $num_or_str, "\n";
15 |
16 |# concatenate the variables as strings
17 |$num_or_str = $num . $str;
18 |
19 |print $num_or_str, "\n";
20 |
21 |exit;
```

例 5.5 的输出：

```

1 |1234 1234
2 |2468
3 |12341234
```

例 5.5 演示了 Perl 对标量变量数据类型的智能化处理，它是字符串还是数字，你是要像数字那样对它进行加减，还是像字符串那样把它连接起来。Perl 会具体问题具体分析，这使得程序员的工作更加简单一些。Perl 为你“作对的事情”。

接下来就是一种循环，foreach 循环。该循环作用于数组的元素。这一行：

```
1 |foreach $base (@DNA) {
```

对 @DNA 数组的元素进行循环处理，每循环一次，标量变量 \$base（或者你起的任意一个名字）就会被设定成数组中的下一个元素。

循环的主体检查每一个碱基，当它发现某个碱基时就会增加它的计数。在 Perl 中，有四种方法给一个数字加 1。在这个例子中，你把 ++ 放在变量的前面，就像这样：

```
1 | ++$count;
```

你也可以把 ++ 放在变量的后面：

```
1 | $count++;
```

你可以把它写成这样，加法和赋值的组合：

```
1 | $count = $count + 1;
```

或者，作为它的简写，也可以这样：

```
1 | $count += 1;
```

这几乎成了选择的烦恼。++ 这种写法对于增加计数来说比较方便，就像我们此处这样。+= 这种写法节省一些输入，在加除 1 以外的数字时比较常用。

例 5.5 中的 foreach 循环也可以写成这样：

```
1 | foreach (@DNA) {
2 |
3 |     if ( /A/ ) {
4 |         ++$count_of_A;
5 |     } elsif ( /C/ ) {
6 |         ++$count_of_C;
7 |     } elsif ( /G/ ) {
8 |         ++$count_of_G;
9 |     } elsif ( /T/ ) {
10 |         ++$count_of_T;
11 |     } else {
12 |         print "!!!!!!! Error - I don't recognize this base: ";
13 |         print;
14 |         print "\n";
15 |         ++$errors;
16 |     }
17 | }
```

foreach 循环的这种写法：

```
1 | foreach(@DNA) {.
```

没有标量值。在 foreach 循环中，如果你不指明存储从数组中读取的标量的变量（在例 5.5 的循环中担当该职责的是 \$base），Perl 就会使用特殊变量 \$_。

此外，在不提供参数的情况下，Perl 的许多内置函数都会对这个特殊变量进行操作。在这个例子中，条件测试是简单的模式，Perl 假定你对 \$_ 变量进行模式匹配，所以这和你使用 \$_ =~ /A/ 是完全一样的。最后，在错误信息中，print; 语句打印出 \$_ 变量的值。

尽管在本书中，我不会过多的使用它，但这个不需要命名的特殊变量 \$_ 会出现在许多 Perl 程序中。

5.6 操作字符串

要查看每一个字符，其实也没必要把字符串拆解成数组。事实上，有时你会想避免那种做法。一个大的字符串会占用你计算机的大量内存，对于大的数组也是如此。当你把字符串拆解成数组时，原始的字符串仍然存在着，而你还要对每一个字符制作一个拷贝，作为元素存储到创建的数组中。如果你有一个大的字符串，它已经占用了一大部分可用内存，创建另一个数组可能会导致内存不足。当内存不足时，计算机的性能会大打折扣，它会慢得像乌龟一样，甚至崩溃或冻结（“挂起”）。到现在为止，这些都还不是什么令人担忧的因素，但当你处理大的数据集（比如人类基因组）时，你将不得不考虑这些问题。

那么假设你不想制作一个 DNA 序列数据的拷贝存进另一个变量。有没有什么方法可以直接查看 \$DNA 字符串并对碱基进行计数吗？答案是肯定的。下面是伪代码，紧随其后的是 Perl 程序：

```
1 read in the DNA from a file
2
3 join the lines of the file into a single string of $DNA
4
5 # initialize the counts
6 count_of_A = 0
7 count_of_C = 0
8 count_of_G = 0
9 count_of_T = 0
10
11 for each base at each position in $DNA
12
13     if base is A
14         count_of_A = count_of_A + 1
15     if base is C
16         count_of_C = count_of_C + 1
17     if base is G
18         count_of_G = count_of_G + 1
19     if base is T
20         count_of_T = count_of_T + 1
21 done
22
23 print count_of_A, count_of_C, count_of_G, count_of_T
```

例 5.6是查看 DNA 字符串中每个碱基的程序。

例 5.6：确定核苷酸频率，第二次尝试

```
1 #!/usr/bin/perl -w
2 # Example 5-6    Determining frequency of nucleotides, take 2
3
4 # Get the DNA sequence data
5 print "Please type the filename of the DNA sequence data: ";
6
7 $dna_filename = <STDIN>;
```

```
8
9 chomp $dna_filename;
10
11 # Does the file exist?
12 unless ( -e $dna_filename ) {
13
14     print "File \"$dna_filename\" doesn't seem to exist!!\n";
15     exit;
16 }
17
18 # Can we open the file?
19 unless ( open( DNAFILE, $dna_filename ) ) {
20
21     print "Cannot open file \"$dna_filename\"\n\n";
22     exit;
23 }
24
25 @DNA = <DNAFILE>;
26
27 close DNAFILE;
28
29 $DNA = join( ' ', @DNA );
30
31 # Remove whitespace
32 $DNA =~ s/\s//g;
33
34 # Initialize the counts.
35 # Notice that we can use scalar variables to hold numbers.
36 $count_of_A = 0;
37 $count_of_C = 0;
38 $count_of_G = 0;
39 $count_of_T = 0;
40 $errors      = 0;
41
42 # In a loop, look at each base in turn, determine which of the
43 # four types of nucleotides it is, and increment the
44 # appropriate count.
45 for ( $position = 0 ; $position < length $DNA ; ++$position ) {
46
47     $base = substr( $DNA, $position, 1 );
48
49     if ( $base eq 'A' ) {
50         ++$count_of_A;
51     }
52     elsif ( $base eq 'C' ) {
53         ++$count_of_C;
54     }
55     elsif ( $base eq 'G' ) {
56         ++$count_of_G;
```

```

57     }
58     elsif ( $base eq 'T' ) {
59         ++$count_of_T;
60     }
61     else {
62         print "!!!!!!! Error - I don't recognize this base: $base\n";
63         ++$errors;
64     }
65 }
66
67 # print the results
68 print "A = $count_of_A\n";
69 print "C = $count_of_C\n";
70 print "G = $count_of_G\n";
71 print "T = $count_of_T\n";
72 print "errors = $errors\n";
73
74 # exit the program
75 exit;

```

下面是例 5.6 的输出：

```

1 Please type the filename of the DNA sequence data: small.dna
2 !!!!!!!! Error - I don't recognize this vase: V
3 A = 40
4 C = 27
5 G = 24
6 T = 17
7 errors = 1

```

在例 5.6 中，我添加了一行代码，来看看文件是否存在：

```

1 unless ( -e $dna_filename ) {

```

有许多各种各样的文件测试操作符，可以参看附录 B 或-X 下的 Perl 文档⁹。注意文件有许多属性，比如大小、权限、文件系统中的位置、文件类型等，对于许多这样的属性都可以简单地使用文件测试操作符进行检测。

此外，注意我保留了关于正则表达式的详细注释，因为正则表达式难于阅读，此处的少许注释可以帮助读者跳过代码。

所有的代码都非常熟悉，除了 `for` 循环，需要对它进行一些解释：

```

1 for ( $position = 0 ; $position < length $DNA ; ++$position ) {
2
3     # the statements in the block
4 }

```

这里的 `for` 循环和下面的 `while` 循环是完全等价的：

⁹译者注：在终端中使用 `perldoc -f -X` 即可查看文件测试操作符的文档。


```
1 | $position = 0;
2 |
3 | while( $position < length $DNA ) {
4 |
5 |     # the same statements in the block, plus ...
6 |
7 |     ++$position;
8 | }
```

花点时间，把这两个循环比较一下。你会看到同样的语句，但是出现在了不同的地方。

如你所见，在 *for* 循环的循环语句中引入了计数器 (*\$position*) 的初始化和增量，而在 *while* 循环中，它们则是单独的语句。在 *for* 循环中，初始化和增量语句都包裹在了括号中，而在 *while* 循环的括号中只有条件测试。在 *for* 循环中，你把初始化语句放在了第一个分号的前面，把增量语句放在了第二个分号的后面。初始化语句在循环开始之前就执行了，而且只初始化一次，而增量语句则是在每次循环迭代完成后、条件测试之前执行的。如此所述，它实际上就是完全等价于 *while* 循环的一种简写方法。

条件测试检测处理到达字符串的位置是否小于字符串的长度。它使用的是 Perl 的 *length* 函数。显然，你不能检测超过字符串长度的字符。但对于字符串和数组位置的索引还要多说几句。

默认情况下，Perl 假定字符串的索引起始于 0，它的最后一个字符的索引要比字符串的长度小一。问什么不使用起始于 1、终止于字符串长度的索引，而是用这样的索引呢？这其中有很多原因，但它们都有些深奥；可以参看一下关于启蒙 (enlightenment) 的文档。稍感欣慰的是，许多其他编程语言使用的也是 Perl 的这种索引方式。（当然，也有编程语言选择了直观的索引方式，从 1 开始进行索引。好吧。）

这种索引方式对于生物学家来说非常重要，因为他们已经习惯了从 1 开始对序列进行索引，而不是像 Perl 这样从 0 开始。在打印输出结果之前，你有时需要把位置索引加 1，这样对于非程序员来说就比较容易理解了。这可能稍微有些烦人，但你会慢慢习惯的。

对于数组元素的索引也是一样的，数组的第一个元素的索引是 0，最后一个元素的索引是 *\$length-1*。

不管怎么说，当 *\$position* 的值小于等于 *length-1* 时，条件测试的结果为真，当 *\$position* 的值和字符串的长度相等时，测试就会失败。举个例子，假设你有一个包含“seeing”文本的字符串，它的长度为 6 个字符。其中，“s”的位置索引为 0，“g”的位置索引为 5，比字符串的长度 6 小一。

在回来看看代码块，你使用 *substr* 函数来查看字符串：

```
1 | $base = substr($DNA, $position, 1);
```

对于处理字符串来说，这是相当常用的一个函数，你还可以用它来进行插入或删除。在这个例子中，你只想查看一个字符，所以你对字符串 *\$DNA* 使用了 *substr* 函数，让它找到位置索引为 *\$position* 的那个字符，并把结果保存到标量变量 *\$base* 中。然后，就像前面的例 5.4 程序那样，对碱基进行计数。

5.7 写入文件

例 5.7 演示了对 DNA 字符串中的核苷酸进行计数的另一种方法。它使用了一个 Perl 的技巧，这种技巧就是专门为类似的工作设计的。在 *while* 循环的测试中，它进行了一个全局的正则表达式查找，就像你将看到的，这是一种对字符串中字符进行计数的简洁的方法。

Perl 比较好的一点就是，对于那些相当常见的事情，它很可能提供了一种相对简洁的处理方法。（而它的弊端就是对于 Perl，有大量知识需要你去学习。）

例 5.7 的结果，不仅会输出打印到屏幕，还会写入到文件中。实现写入文件的代码如下：

```
1 | # Also write the results to a file called "countbase"
2 |
3 | $outputfile = "countbase";
4 |
5 | unless ( open(COUNTBASE, ">$outputfile") ) {
6 |
7 |     print "Cannot open file \"$outputfile\" to write to!!\n\n";
8 |     exit;
9 | }
10 |
11 | print COUNTBASE "A=$a C=$c G=$g T=$t errors=$e\n";
12 |
13 | close(COUNTBASE);
```

正如你所见，要写入到文件，你要像读取文件那样调用 *open*，但有一点不同：在文件名前使用的是大于号 *>*。文件句柄成了 *print* 语句的第一个参数（但其后并没有逗号），这使得 *print* 语句把它的输出定向到了文件。¹⁰

例 5.7 检查 DNA 字符串中每个碱基的第三个版本的 Perl 程序。

例 5.7：确定核苷酸频率，第三次尝试

```
1 | #!/usr/bin/perl -w
2 | # Example 5-7   Determining frequency of nucleotides, take 3
3 |
4 | # Get the DNA sequence data
5 | print "Please type the filename of the DNA sequence data: ";
6 |
7 | $dna_filename = <STDIN>;
8 |
9 | chomp $dna_filename;
10 |
11 | # Does the file exist?
12 | unless ( -e $dna_filename ) {
13 |
14 |     print "File \"$dna_filename\" doesn't seem to exist!!\n";
```

¹⁰在这种情况下，如果文件已经存在了，它将先被清空然后写入。但也是可以指定其他的行为方式的。如前所述，Perl 文档中有关于 *open* 函数的所有细节，可以设定选项来读取、写入文件，以及其他操作。

```
15     exit;
16 }
17
18 # Can we open the file?
19 unless ( open( DNAFILE, $dna_filename ) ) {
20
21     print "Cannot open file \"$dna_filename\"\n\n";
22     exit;
23 }
24
25 @DNA = <DNAFILE>;
26
27 close DNAFILE;
28
29 $DNA = join( ' ', @DNA );
30
31 # Remove whitespace
32 $DNA =~ s/\s//g;
33
34 # Initialize the counts.
35 # Notice that we can use scalar variables to hold numbers.
36 $a = 0;
37 $c = 0;
38 $g = 0;
39 $t = 0;
40 $e = 0;
41
42 # Use a regular expression "trick", and five while loops,
43 # to find the counts of the four bases plus errors
44 while ( $DNA =~ /a/ig ) { $a++ }
45 while ( $DNA =~ /c/ig ) { $c++ }
46 while ( $DNA =~ /g/ig ) { $g++ }
47 while ( $DNA =~ /t/ig ) { $t++ }
48 while ( $DNA =~ /[^\acgt]/ig ) { $e++ }
49
50 print "A=$a C=$c G=$g T=$t errors=$e\n";
51
52 # Also write the results to a file called "countbase"
53 $outputfile = "countbase";
54
55 unless ( open( COUNTBASE, ">$outputfile" ) ) {
56
57     print "Cannot open file \"$outputfile\" to write to!!\n\n";
58     exit;
59 }
60
61 print COUNTBASE "A=$a C=$c G=$g T=$t errors=$e\n";
62
63 close(COUNTBASE);
```

```

64 |
65 | # exit the program
66 | exit;

```

当你运行例 5.7 时，会看到类似的输出：

```

1 | Please type the filename of the DNA sequence data: small.dna
2 | A=40 C=27 G=24 T=17 errors=1

```

在你运行例 5.7 后，计数碱基的输出文件中包含以下内容：

```

1 | A=40 C=27 G=24 T=17 errors=1

```

while 循环：

```

1 | while($dna =~ /a/ig){$a++}

```

有它自己的条件测试，包裹在括号中，这是一个字符串匹配的表达式：

```

1 | $dna =~ /a/ig

```

这个表达式查找正则表达式 `/a/`，也就是 `a` 这个字母。既然它使用了 `i` 修饰符，那这就是不区分大小写的匹配，它将匹配 `a` 或者 `A`。它还使用了全局修饰符，这意味它将匹配字符串中的所有 `a`。（没有全局修饰符时，在每次循环迭代时，如果在 `$dna` 中有“`a`”，它只会持续返回 `0`。）

现在，*while* 循环中的这个字符串匹配表达式，使得 *while* 循环在每一次匹配正则表达式时都执行它的代码块。所以，附加了这只有一个语句的代码块：

```

1 | {$a++}

```

在每次匹配正则表达式时递增计数器。换句话说，你会对所有的 `a` 进行计数。

对于这第三个版本的程序来说，还有一点需要提及一下。你会注意到其中有些语句发生了变化，被缩短了。有些变量的名字更短了，有些语句放在了一行中，同时末尾的 *print* 语句也更加简洁了。这些都只是书写的另一种方式而已。在你编程时，你将会发现自己会遇到不同的书写方式：适当的去尝试一下吧。

在第三个版本的程序中，计数碱基的方式比较灵活。比如，不需要单独指定各个碱基，你就可以对 ACGT 以外的所有碱基进行计数。在后面的章节中，你将使用这样的 *while* 循环来取得良好的效果。然后，还有一种更快的计数碱基的方法。你可以使用第 4 章中提到的 *tr* 转换函数，它的速度更快一些，如果你有大量的 DNA 需要计数，它将会非常有用：

```

1 | $a = ($dna =~ tr/Aa//);
2 | $c = ($dna =~ tr/Cc//);
3 | $g = ($dna =~ tr/Gg//);
4 | $t = ($dna =~ tr/Tt//);

```

tr 函数返回它在字符串中找到的特定字符的数目，并且，如果替换的字符集为空的话，原始的字符串并不会被改变。这使得它成为一个很好的字符计数器。注意使用 *tr* 时，

你需要同时指定大小写字母。此外，因为 *tr* 并不接受字符组，所以没法直接对非碱基的字符进行计数。但是，你可以这样做：

```
1 | $basecount = ($dna =~ tr/ACGTacgt//);  
2 | $nonbase = (length $dna) - $basecount;
```

使用 *tr* 的这个程序要比例 5.7 中使用 *while* 循环的程序运行快。

你可能会觉得计数碱基的程序有三个（实际上是四个）版本有点多了，尤其是每个版本中的大部分代码都是一样的。程序中唯一改变的就是计数碱基的部分。有没有什么办法，能够让我们方便的只改变计数碱基的部分呢？在第 6 章中，你将看到这样的方法，使用子程序来把程序进行分割。

5.8 练习题

习题 5.1

使用循环写一个永不挂起的程序。每次迭代时，条件测试都应该永远为真。注意，有的系统会注意到你在一个无穷的循环中，从而自动结束程序。依据你使用的操作系统的不同，结束程序的方法也会有所不同。在 Unix 和 Linux、Windows MS-DOS 命令窗口或者 MacOS X shell 窗口中，使用 Ctrl-C 就可以结束程序。

习题 5.2

提示用户输入两个（短的）DNA 字符串。通过使用 `.` 赋值操作符，把第二个附加到第一个 DNA 字符串的后面，从而把两者连接起来。先打印输出连接后的两个字符串，然后打印输出第二个字符串，但要与连接后字符串末尾对应的字符串对齐。举个例子，如果输入的字符串是 AAAA 和 TTTT，则要打印输出：

```
AAAATTTT
      TTTT
```

习题 5.3

编写一个程序，输出从 1 到 100 的所有数字。当然，你的程序代码应该远远小于 100 行。

习题 5.4

编写一个程序，计算 DNA 链的反向互补链。不要使用 `s///` 或者 `tr` 函数。使用 `substr` 函数，当你计算反向互补链时，要一次检查原始 DNA 中的一个碱基。（提示：你可能会发现检查原始 DNA 字符串时，尽管从左到右也是可以的，但从右向左要更加方便一些）

习题 5.5

编写一个程序，报告蛋白质序列中疏水氨基酸的百分比。（想要知道哪些氨基酸是疏水性的，可以参看任何关于蛋白质、分子生物学或细胞生物学导论性的介绍。在附录 A 中你会找到一些有用的资源。）

习题 5.6

编写一个程序，检查一下作为参数提供的两个 DNA 字符串是不是反向互补的。使用 Perl 的内置函数 `split`、`pop`、`shift` 和 `eq`（`eq` 实际上是一个操作符）。

习题 5.7

编写一个程序，报告序列的 GC 含量。（换言之，就是给出 DNA 中 G 和 C 的百分比）。

习题 5.8

修改例 5.3，使它不仅能通过正则表达式找到基序，还能打印输出它找到的基序。举个例子，如果你使用正则表达式查找基序 `EE.*EE`，你的程序应该输出 `EETVKNDEE`。你可以使用特殊变量 `$&`。在一次成功的模式匹配后，这个特殊变量会保存匹配到的模式。

习题 5.9

编写一个程序，交换 DNA 字符串中特定位置的两个碱基。（提示：你可以使用 Perl 的 `substr` 函数或 `slice` 函数。）

习题 5.10

编写一个程序，写入一个临时文件然后把它删除掉。*unlink* 函数可以删除一个文件，举个例子，这样来使用：

```
1 | unlink "tmpfile";
```

但要检查一下看看 `unlink` 是否成功了。

第 6 章 子程序和 Bugs

目录

6.1 子程序	94
6.2 作用域和子程序	97
6.3 命令行参数和数组	102
6.4 传递数据给子程序	105
6.5 模块和子程序库	109
6.6 修复代码中的 Bugs	110
6.7 练习题	123

在本章中，你将学习到以下两个主题的基础知识：

- 子程序
- 使用 Perl 调试器

子程序是结构化组织程序的一个重要方法。第 7 章将学习如何通过随机化来模拟 DNA 的突变，在那里你将使用到子程序。Perl 调试器会用“慢镜头”的形式来检查一个程序的行为，帮助你找到那些讨厌的 bugs。

6.1 子程序

子程序是结构化组织程序的一个重要方法，所有主流的编程语言中都使用子程序。

子程序把一些代码包裹起来，给它起一个名字，并提供方法把一些值提供给它进行计算，然后返回计算结果。这样程序的其余部分就可以仅仅通过使用它的名字来使用子程序中的代码了，把需要的值传递给子程序代码并收集运算结果。这种子程序的使用或“调用”通常称作调用子程序。你可以把子程序看做程序中的一个程序，就像你运行程序得到结果一样，程序调用子程序得到结果。一旦你有了一个子程序，就可以在程序中使用它了，只需要知道传递哪些值、收集哪种类型的值就行了。

6.1.1 子程序的优势

子程序提供了一些好处。它赋予了程序抽象化和模块化的能力，通过把代码组织成有特定输入输出的代码块就可以创建大的程序了。

假设你需要计算一些东西，比如在一个或多个不同程序的多个地方计算分布的平均值。通过把这种计算编写成子程序，只需要写一次，你就可以在任何需要它的时候调用它了，这会使你的程序：

- 更短，因为你在重用代码。
- 更容易测试，因为你可以单独对子程序进行测试。
- 更容易理解，因为它使得程序具有良好的组织、更加简洁。
- 更加稳健，因为重用子程序使得代码量减少了，出错的几率也小了。
- 编写起来更加快速，因为你可能已经编写了好多进行基本统计的子程序，这样你就可以直接调用计算平均值的那个子程序而不用重写了。还有更好的一种可能，你找到了一个别人编写的很好的统计学库，这样你就完全不用自己去写了。

还有一个更加微妙的事情，也这正是它的强大之处。子程序本身就可以调用其他的子程序，也就是说，一个子程序可以根据计算需要使用其他的子程序。¹通过编写一系列的子程序，每一个子程序只做一件或很少的几件事，你可以通过各种形式把它们组合编写成新的子程序，然后你还可以继续组合这些新的子程序，依此类推，最后可能会形成一个巨大且灵活的程序系统。把问题分解成可以方便组合的一系列的子程序，使你可以创造不断增长且适应各种条件的程序，而你只需要很少的付出就可以实现这一点。

所有这一切的技巧就在于你如何把代码分割成一系列的子程序。你希望子程序能够封装一些通用且有用的东西，并且不会只被调用一次（尽管有时这也非常有用）。有一些经验法则：子程序应该只做一件事情并把它做好，并且子程序的代码最好不要超过一页或者两页。这些并不是真正的法则，例外时有发生，但它可以帮你把代码分割成易于管理的代码块，这非常适合子程序。

6.1.2 编写子程序

让我们看看子程序是如何使用及定义的吧。

要使用一个子程序，你需要把数据作为参数传递给子程序，然后收集子程序的运算结果值。举个例子，你想要一个子程序，可以把“ACGT”附加到给定 DNA 的末尾，并返回得到的新的、更长的 DNA。让我们把这个子程序叫做 *addACGT* 吧。在 Perl 中，你要调用

¹子程序甚至可以调用它自己本身，这就是所谓的递归，这使得计算过程非常优雅（参看第 11 章）。

一个子程序，通常只需要键入它的名字，并在后面跟上用小括号包裹起来的参数列表就可以了（如果有参数的话）。比如，这是带着 `$dna` 这个参数调用 `addACGT` 子程序的方法：

```
1 | addACGT($dna);
```

当调用子程序时，较旧版本的 Perl 要求在子程序名称的前面加上 `&` 字符（与字符）。现在这样写也是可以的（比如写成：`&addACGT`），但如今一般都把与字符省略掉。²

例 6.1 演示了一个子程序，从中可以看到它的工作细节。

例 6.1：把 ACGT 附加到 DNA 上的子程序

```
1 | #!/usr/bin/perl -w
2 | # Example 6-1    A program with a subroutine to append ACGT to DNA
3 |
4 | # The original DNA
5 | $dna = 'CGACGTCTTCTCAGGCGA';
6 |
7 | # The call to the subroutine "addACGT".
8 | # The argument being passed in is $dna; the result is saved in $longer_dna
9 | $longer_dna = addACGT($dna);
10 |
11 | print "I added ACGT to $dna and got $longer_dna\n\n";
12 |
13 | exit;
14 |
15 | #####
16 | # Subroutines for Example 6-1
17 | #####
18 |
19 | # Here is the definition for subroutine "addACGT"
20 |
21 | sub addACGT {
22 |     my ($dna) = @_;
23 |
24 |     $dna .= 'ACGT';
25 |     return $dna;
26 | }
```

例 6.1 会产生这样的输出：

```
1 | I added ACGT to CGACGTCTTCTCAGGCGA and got CGACGTCTTCTCAGGCGAACGT
```

现在来看看这些代码，看看子程序是如何被定义的、以及在 Perl 程序中使用的。

首先注意的一点，从宏观上来看，程序现在有两小部分。第一部分从程序的开头开始，到 `exit` 命令结束。随后的一部分是子程序的定义（为了便于理解在开头添加了大量的注释），在这个例子中，就是 `addACGT` 子程序的定义。为了方便阅读，通常把所有的子程序定义都集中放在程序的末尾，而且会以字母顺序或其他方便的形式对子程序进行排列。

²即使在较新版本的 Perl 中，有时也需要加上与字符。在第 11 章的第 11.2.3 小节中你会看到这样的例子，那一小节介绍的是 `File::Find` 模块。（还可以参看文档中的 `defined` 和 `undef` 函数，或者 `perlref` 手册页）。

实际上，把子程序的定义放在程序中的任何地方都是合法的。这是因为 Perl 首先通读代码，在开始运行程序之前，做一些检查语法、学习子程序定义之类的事情。特别的，子程序代码可以紧跟在程序中使用它们的地方的后面（许多人认为法则是放在其前面，但并不需要这样）。并且，不一定非要把子程序都集中放在一起，可以散落在程序代码的各个地方。但是把它们集中放在程序末尾的这种方法，会使阅读程序更加容易一些。可能的一个例外就是，在代码的某一部分使用一个小的子程序，比如使用 `sort` 函数时这种情况就时有发生。像这种情况，把子程序的定义放在使用它的地方，可以避免读者在子程序定义和它的使用之处两个地方来回翻页。通常，不看子程序定义而把程序整个阅读一遍会更加方便，先对程序的整个流程有个了解，需要时再回去仔细看看子程序。

如你所见，例 6.1 非常简单。它首先把某个 DNA 保存到变量 `$dna` 中，然后把那个变量作为参数去调用子程序，看起来就像这样：`addACGT($dna)`。子函数的调用是通过它的名字来实现的，并在其后紧跟用小括号包裹起来的子程序参数。也有可能没有参数，或者有不只一个参数，这时要用逗号把它们分割开来。可以把子程序的返回值保存起来。在这个程序中，子程序的返回值保存在了叫做 `$longer_dna` 的变量中，随后就把这个变量打印了出来，然后程序就退出了。

从程序开始到 `exit` 语句的这一部分叫做主程序或程序的主体。通过查看这一部分代码，而不需要去看子程序的细节，你就可以看出程序从头到尾发生了什么。

既然你已经看完了例 6.1 中的主程序，是时候去看一下子程序的定义、以及它是如何使用作用域这个概念的了。

6.2 作用域和子程序

一个子程序的定义包括三部分：子程序定义的保留字³——`sub`，子程序的名字——在这里是 `addACGT` 和一个包裹在成对大括号中的代码块。这里的代码块和前面看到的循环和条件语句中把语句集中在一起的代码块是一样的。

在例 6.1 中，子程序的名字是 `addACGT`，而代码块就是名字后面的所有代码。下面把子程序的定义重复了一遍：

```
1 sub addACGT {
2     my($dna) = @_ ;
3
4     $dna .= 'ACGT' ;
5     return $dna ;
6 }
```

现在，让我们来看看子程序的代码块吧。

一个子程序就像是单独的一个主程序的辅助程序一样，它需要有自己的变量。在本书的子程序中你将使用两种类型的变量：⁴

- 传递给子程序的参数
- 其他通过 `my` 声明的变量，它们的作用域会被限制在子程序中

参数就是使用或调用子程序时传递给它的值。参数值通过 `@_` 这个特殊变量传递给子程序，在下一小节你将会看到。

其他子程序可能使用到的变量，需要与程序其他部分使用到的变量区分开来，这样它们就只能在子程序自己的范围内发挥作用了。这可以通过使用 `my` 声明变量来实现，稍后再对此进行详细解释。

最后，大部分子程序都通过 `return` 函数返回它们的结果。就像我们的子程序 `addACGT` 中 `return $dna;` 一样，它可以返回单独的一个标量，也可以像 `return ($dna1, $dna2);` 这样返回一个标量列表，或者像 `return @lines;` 这样返回一个数组，等等。

6.2.1 参数

调用子函数意味着要键入它的名字，给它适当的参数，通常还要收集它的结果。参数 (*argument, parameter*) ⁵通常包含子程序要计算的数据。在例 6.1 中，使用参数 `$dna` 调用子程序 `addACGT`：

```
1 $longer_dna = addACGT($dna);
```

³保留字是 Perl 语言中定义的基本字，如 `if`、`while`、`foreach` 和 `sub`。

⁴在本书的子程序中，我们不会使用全局变量，全局变量能同时被主程序和子程序看到；我们也不会使用通过 `local` 声明的变量，它和 `my` 的作用域限制有所不同。

⁵译者注：parameter 是指函数定义中的参数，而 argument 指的是函数调用时的实际参数。简略描述为：parameter= 形参 (formal parameter)，argument= 实参 (actual parameter)。在不很严格的情况下，现在二者可以混用，一般用 argument，而 parameter 则比较少用。While defining method, variables passed in the method are called parameters. 当定义方法时，传递到方法中的变量称为参数。While using those methods, values passed to those variables are called arguments. 当调用方法时，传给变量的值称为引数。（有时 argument 被翻译为“引数”）

最基本的一点，当程序员想要使用子程序时，可以使用子程序可以接受的任何参数、你需要计算的数据（在这个例子中就是需要把 ACGT 附加上去的任何 DNA）、出现在子程序 `@_` 数组中的每一个参数的值来调用它。

当使用特定参数调用子程序时，你在调用中使用的参数的名字在子程序内部就无关紧要了，只有被实际传递到子程序内部的参数的值才是最重要的。子程序通常从 `@_` 数组中收集这些值，并把它们赋给新的变量，这些变量的名字和你调用子程序时使用的变量名可能一样、也可能不一样。唯一保持不变的是值的顺序，而非包含值的变量的名字。

下面是它的工作原理。子程序代码块的第一行是：

```
1 | my($dna) = @_;
```

调用子程序时的参数的值被传递到了子程序中的特殊数组变量 `@_` 中。因为它以 `@` 字符开头，所以你知道这是一个数组。它有一个简短的名字“`_`”，这是一个特殊的数组变量，是 Perl 程序预先定义好的。（你不能再为自己的数组起这样的名字了。）`@_` 数组包含着传递到子程序中所有的标量值，这些标量值就是调用子程序时参数的值。在这个例子中，只有一个标量值：DNA 字符串，是作为参数传递给子程序的 `$dna` 变量的值。

如果子程序有更多的参数——比如一个 DNA 的参数，一个相关蛋白的参数，还有一个基因名的参数——它们都被传递了进去，并在子程序内部被赋给了用 `my` 声明的变量：


```
1 | my($dna,$protein,$name_of_gene) = @_;
```

如果没有参数，在子程序中就可以省略这样的语句了。

在子程序中，这个语句：

```
1 | my($dna) = @_;
```

执行后，传递进去的值就赋给了子程序的变量 `$dna`。接下来的小节将解释为什么这个变量对于子程序来说是一个新的变量。子程序中的变量也可以起任何名字，并不一定要和参数的名字一样，虽然这个例子中它们是一样的。关于作用域非常酷的一点就是，名字一样不一样是无所谓的。

当心这个常见的错误，在子程序中为参数命名时忘记使用 `@_` 数组， 换言之，使用语句 `my($dna);` 代替了 `my($dna) = @_;`。如果你犯了这种错误，即使已经声明了变量名，参数的值也不会出现在你的子程序中。

6.2.2 作用域

通过保证子程序使用到的所有变量只在子程序中有效，你就可以安全得在任何地方调用子程序了。通过使用 `my` 来声明这些变量，就可以使它们只在子程序中有效了。`my` 是 Perl 定义的关键字，它可以把变量限制在使用它们的代码块中（在这个例子中，这个代码块就是子程序）。⁶

⁶作用域有不同的类别。`my` 实现的类别叫做词法作用域，也叫做静态作用域。在 Perl 中还有一种方法，就是使用 `local` 来声明变量，但你几乎总会希望使用 `my`。

把变量隐藏起来，使它们仅局限在程序的特定部分，这就是作用域的概念。在 Perl 中，使用 `my` 声明变量就是所谓的词法作用域，这是使你的程序模块化的一个至关重要的部分。

像这样用 `my` 来声明一个变量：

```
1 | my($x);
```

或者：

```
1 | my $x ;
```

或者，变量的声明和变量值的初始化一块进行：

```
1 | my($x) = '49';
```

又或者，如果你在子程序中收集一个参数：

```
1 | my($x) = @_;
```

一旦通过这种方式声明了一个变量，它就只存在于声明所在的代码块中，直到代码块的末尾。所以，在一个子程序中，如果你像这样声明了所有的变量（包括参数和其他所有的变量），它们就只在子程序中有效。如果某个变量和程序别处的另一个变量同名，你也无须担心什么，因为 `my` 声明实际上创建了一个新的变量，它只在包裹起来的代码块中有效，代码块外面使用的任何重名的变量都和它完全无关。

例子中演示的在子程序中收集参数，都用小括号把变量包裹了起来。因为 `@_` 是一个数组，用小括号把新变量包裹起来就把它放在了数组上下文⁷中，这可以保证它们能被正确地初始化（参看第 4 章）。



在子程序中，永远使用 `my` 这样的关键字来声明你的所有变量，即使这些变量并不是作为参数传递进来的。

为什么使用作用域呢？例 6.2 演示了不使用作用域带来的麻烦。回忆一下，子程序的优势之一就是，对于那些有用的代码只需写一次，就可以在需要的时候重复使用了。例 6.2 这个程序，在主程序中的一个变量和它调用的子程序中的一个变量重名了。在你编写完主程序一段时间后（比如说半年后）再编写子程序时，或者调用别人写的子程序时，这种情况非常容易发生。

例 6.2：不使用 `my` 变量导致的陷阱

```
1 | #!/usr/bin/perl -w
2 | # Example 6-2   Illustrating the pitfalls of not using my variables
3 |
4 | $dna = 'AAAAA';
5 |
6 | $result = A_to_T($dna);
7 |
8 | print "I changed all the A's in $dna to T's and got $result\n\n";
```

⁷译者注：即列表上下文。

```

9 |
10 | exit;
11 |
12 | #####
13 | # Subroutines
14 | #####
15 | sub A_to_T {
16 |     my ($input) = @_;
17 |
18 |     $dna = $input;
19 |
20 |     $dna =~ s/A/T/g;
21 |
22 |     return $dna;
23 | }

```

例 6.2 给出这样的输出：

```
1 | I changed all the A's in TTTTT to T's and got TTTTT
```

我们期望的应该是这样的输出：

```
1 | I changed all the A's in AAAAA to T's and got TTTTT
```

要想获得这个期望的输出，可以把子程序 `A_to_T` 的定义改成下面这样，使用 `my` 声明变量把子程序中的 `$dna` 变成 `my` 变量：

```

1 | sub A_to_T {
2 |     my($input) = @_;
3 |     my($dna) = $input;
4 |     $dna =~ s/A/T/g;
5 |     return $dna;
6 | }

```

例 6.2 错在哪儿了呢？当程序进入子程序后，使用变量 `$dna` 进行字符串处理来把 A 变成 T 时，Perl 语言看到在主程序中已经有一个叫做 `$dna` 的变量了，因此就继续使用它了。当程序从子程序中返回后，运行 `print` 语句时，它仍然使用这同一个（仅此一个）变量 `$dna`。所以，当它打印输出结果时，变量 `$dna` 中就不是原来的 DNA 了，而是在子程序中经过处理已经改变了的 DNA。

现在类似的事情经常会发生。程序员倾向于大量使用特定的变量名：常见的变量名如 `$tmp`、`$temp`、`$x`、`$a`、`$number`、`$variable`、`$var`、`$array`、`$input`、`$output`、`$result`、`$data`、`$file`、`$filename` 等等。生物信息学家则偏爱 `$dna`、`$protein`、`$motif`、`$sequence` 这样的变量名。当你开始使用别人写的子程序库、你的程序越来越庞大时，让 Perl 语言来考虑避免重名的问题会更加容易、也更加安全一些。

事实上，从现在开始，我们将不在使用未经声明的变量。从这里开始，我们的所有变量，即使是主程序中的变量，我们都会用 `my` 来声明。通过在你的程序中添加下面这条指令，可以使这条纪律得到强制执行：

```
1 | use strict;
```

它会有这样的效果，要求把程序中的所有变量都声明为 `my` 变量。

和学习第 4 章与第 5 章的简单而快乐的日子相比，这些代码中看似无关紧要、稍显杂乱的东西可能会让你倍感束缚，但你应该知道许多语言都要求对它们的所有变量进行声明。事实上，当你编写简短的 Perl 程序时，不强制使用严格的作用域还是非常方便的，比如，在你尝试教授编程时，不会想一开始就让成千上万的细节把学生压的喘不过气来。

你编写程序时不小心把变量名拼写错了，使用严格的作用域的另一个好处就会凸显出来。如果变量没有被声明过，Perl 就会使用拼错的名字创建一个新的变量。程序可能不能正常工作，而且要找出问题所在可能会比较困难。通过使用严格的作用域，对于所有没有声明的、拼错的变量，Perl 都会进行抱怨给出提示信息，这会节省你数小时甚至数天抓耳挠腮的工作，向惨不忍睹的代码说再见吧。

最后，让我们再通过例 6.1 来回顾一下作用域、参数和子程序的工作原理吧。要调用子程序，需要键入它的名字 `addACGT`，把参数 `$dna` 传递给它，并且通过给 `$longer_dna` 赋值把结果（如果有的话）收集起来：

```
1 | $longer_dna = addACGT($dna);
```

子程序的第一行，从特殊变量 `@_` 中获取参数的值，并把它保存到自己的变量 `$dna` 中，因为使用了 `my`，所以这个变量在子程序外是看不到的。尽管子程序外的原始变量也叫做 `$dna`，子程序中叫做 `$dna` 的变量实际上是一个全新的变量（只不过名字相同而已），因为 `my` 的使用使得它只属于子程序。这个新变量只在程序进入子程序运行时是有效的。注意例 6.2 末尾 `print` 语句的输出，尽管子程序中叫做 `$dna` 的变量被延长了，但子程序外的原始变量 `$dna` 并没有被改变。

6.3 命令行参数和数组

例 6.3是使用子程序的另一个程序。不需要交互式回答程序的提示，你使用命令行就可以把程序需要的信息（比如文件名，或 DNA 字符串）告诉它。比如，当你计划在特定时间运行程序、而你又不在于的时候，这会非常有用。

例 6.3还演示了使用数组的一些更多的知识。你将看到，如何通过使用下标来访问数组中特定的元素。

对于命令行程序来说，你键入程序名，如果有参数的话，之后紧跟程序的参数，然后按 Enter（或 Return）键就可以使程序运行了。在例 6.3中，当用户键入程序名，它会要求在程序名后给出参数，在这个例子中，就是 DNA 字符串，它会计算 DNA 中 G 的数目。所以程序运行后会返回类似这样的结果：

```
1 | AAGGGGTTTCCC
2 |
3 | The DNA AAGGGGTTTCCC has 4 G's in it!
```

当然，好多程序都有图形用户界面（GUI）。它会让程序占据计算机屏幕的一部分或者全部，界面通常包括菜单、按钮、从键盘键入值来设定参数的输入框之类的东西。

然而，许多程序还是从命令行运行。即使最新的基于 Unix 构建的 MacOS X，现在也提供了一个命令行。（尽管大多数 Windows 用户并不怎么使用 MS-DOS 命令窗口，它仍然非常有用，比如说，运行 Perl 程序。）如前所述，以非交互的形式运行程序，通过命令行传递参数，这可以使程序运行实现自动化，比如在半夜三更没有人真正坐在电脑前的时候运行程序。

例 6.3计算 DNA 字符串中 G 的数目。

例 6.3：通过命令行计算 DNA 中 G 的数目

```
1 | #!/usr/bin/perl -w
2 | # Example 6-3   Counting the number of G's in some DNA on the command line
3 |
4 | use strict;
5 |
6 | # Collect the DNA from the arguments on the command line
7 | #   when the user calls the program.
8 | # If no arguments are given, print a USAGE statement and exit.
9 |
10 | # $0 is a special variable that has the name of the program.
11 | my ($USAGE) = "$0 DNA\n\n";
12 |
13 | # @ARGV is an array containing all command-line arguments.
14 | #
15 | # If it is empty, the test will fail and the print USAGE and exit
16 | #   statements will be called.
17 | unless (@ARGV) {
18 |     print $USAGE;
19 |     exit;
20 | }
```

```

21 |
22 | # Read in the DNA from the argument on the command line.
23 | my ($dna) = $ARGV[0];
24 |
25 | # Call the subroutine that does the real work, and collect the result.
26 | my ($num_of_Gs) = countG($dna);
27 |
28 | # Report the result and exit.
29 | print "\nThe DNA $dna has $num_of_Gs G's in it!\n\n";
30 |
31 | exit;
32 |
33 | #####
34 | # Subroutines for Example 6-3
35 | #####
36 |
37 | sub countG {
38 |
39 |     # return a count of the number of G's in the argument $dna
40 |
41 |     # initialize arguments and variables
42 |     my ($dna) = @_;
43 |
44 |     my ($count) = 0;
45 |
46 |     # Use the fourth method of counting nucleotides in DNA, as shown in
47 |     # Chapter Four, "Motifs and Loops"
48 |     $count = ( $dna =~ tr/Gg// );
49 |
50 |     return $count;
51 | }

```

现在让我们看看程序是如何工作的，同时检查并解释一下其中的新特性。作为开始，注意这新的一行代码：

```
1 | use strict;
```

从现在开始，我将使用它来确保所有的变量都用 `my` 进行了声明，这样就可以强制执行词法作用域了。

Perl 预先设置了一些特殊变量，这样你就可以轻而易举地从命令行中使用参数了。每一个 Perl 程序都有一个数组变量 `@ARGV`，它包含了所有的命令行参数。此外，还有一个叫做 `$0`（是零不是字母 o）的特殊变量，它包含的是在命令行中调用程序时的程序名。

注意，在例 6.3 中，在 `$USAGE` 变量中定义了一个提示信息，开头就是变量 `$0` 的值，后面紧跟着程序需要的参数的指示。这是非常常用且实用的做法。如果用户没有给程序提供所需的信息，通过某种形式的测试进行检测后，程序会打印输出如何正确使用它的信息提示，然后退出。

事实上，这个程序确实进行了检测，看看在命令行上是否键入了参数。它检测 `@ARGV` 中是否包含内容，这种情况下它会被测试为 `1`；或者如果它完全是空的，这种情况下它就

会被测试为 `␣`。如果程序需要用户提供一个参数，你可以使用 `unless` 条件测试，如果 `@ARGV` 是空的，就打印输出 `$USAGE` 语句并退出程序：

```
1 | unless(@ARGV) {  
2 |     print $USAGE;  
3 |     exit;  
4 | }
```

接下来的代码演示了关于数组的一些新的东西，即，如何通过使用下标从数组中提取出一个元素。换句话说，它演示了如何获取第一个、第四个或任何一个元素。例 6.3 中的代码演示了如何提取出第一个元素，如你所见，它的索引值是 0：

```
1 | my($dna) = $ARGV[0];
```

你已经检测确保数组不是空的，那就已经知道肯定有第一个元素了。为了获取数组 `@ARGV` 中的第一个元素，把 `@` 换成 `$`，并在其后跟上用中括号包裹起来的所需的下标：第一个元素的下标是 0，第二个元素的下标是 1，依此类推。这个语法表明，既然你现在想看数组中的一个元素，并且它是一个标量变量，那你就使用美元符号，就像在其他标量变量上那样使用。

在例 6.3 中，你把命令行数组 `@ARGV` 的第一个（也是唯一的一个）元素复制到了变量 `$dna` 中。

最后是子程序的调用，其中并没有什么新的东西，但却实现了第 5 章最后一段的梦想：

```
1 | my($num_of_Gs) = countG ( $dna );
```

6.4 传递数据给子程序

在后续章节中，当开始解析 GenBank、PDB 和 BLAST 文件时，你将需要向子程序传递更加复杂的参数，来从记录数据中解析出多个字段。接下来的几个小节将演示在 Perl 是如何实现这一点的。你可以跳过这一小节，等学习到第 10 章的时候再回来仔细阅读。

6.4.1 子程序：通过值传递

到目前为止，我们所有的子程序的参数都非常简单。这些参数的值被复制并传递给子程序，而且子程序中这些值的变化不会影响到主程序中相应参数的值。这叫做**通过值传递**或者**通过值调用**。举个例子：

```
1 |#!/usr/bin/perl -w
2 |# Example of pass-by-value (a.k.a. call-by-value)
3 |
4 |use strict;
5 |
6 |my $i = 2;
7 |
8 |simple_sub($i);
9 |
10| print "In main program, after the subroutine call, \$i equals $i\n\n";
11|
12| exit;
13|
14|#####
15|# Subroutines
16|#####
17|sub simple_sub {
18|
19|    my($i) = @_;
20|
21|    $i += 100;
22|
23|    print "In subroutine simple_sub, \$i equals $i\n\n";
24|}
```

这会得到下面的输出：

```
1 |In subroutine simple_sub, $i equals 102
2 |
3 |In main program, after the subroutine call, $i equals 2
```

6.4.2 子程序：通过引用传递

如果你的参数更加复杂，比如说混合了标量、数组和散列，对于 Perl 来说通常不能把它们区分开来。Perl 把所有的参数当成一个单独的数组，就是 `@_` 这个特殊数组，传递

给子程序，如果参数中有数组或者散列，在子程序中，它们的元素就会“扁平化”后保存进 @_ 这一个数组中。下面是一个例子：

```

1  #!/usr/bin/perl -w
2  # Example of problem of pass-by-value with two arrays
3
4  use strict;
5
6  my @i = ('1', '2', '3');
7  my @j = ('a', 'b', 'c');
8
9  print "In main program before calling subroutine: i = " . "@i\n";
10 print "In main program before calling subroutine: j = " . "@j\n";
11
12 reference_sub(@i, @j);
13
14 print "In main program after calling subroutine: i = " . "@i\n";
15 print "In main program after calling subroutine: j = " . "@j\n";
16
17 exit;
18
19 #####
20 # Subroutines
21 #####
22
23 sub reference_sub {
24
25     my(@i, @j) = @_;
26
27     print "In subroutine : i = " . "@i\n";
28     print "In subroutine : j = " . "@j\n";
29
30     push(@i, '4');
31
32     shift(@j);
33 }

```

下面的输出说明这种方法存在问题：

```

1  In main program before calling subroutine: i = 1 2 3
2  In main program before calling subroutine: j = a b c
3  In subroutine : i = 1 2 3 a b c
4  In subroutine : j =
5  In main program after calling subroutine: i = 1 2 3
6  In main program after calling subroutine: j = a b c

```

如你所见，在子程序中，@i 和 @j 中的所有元素都被组合进了 @_ 这个数组中。你开始使用的这两个数组之间的区别，在子程序中荡然无存。当你通过下面这一语句试图把这

两个数组找回来时：

```
1 | my(@i, @j) = @_;
```

Perl 把所有的元素都赋给了第一个数组 `@i`。这种行为使得向子程序中传递多个数组变得有些不确定。

此外，一切照旧，因为你使用了词法作用域（`my` 变量），主程序中原始的数组并不会被子程序所影响。

为了避免这种问题，你可以以一种叫做通过引用传递或通过引用调用的方式向子程序中传递参数。通过使用引用，你可以向子程序中传递标量、数组、散列等各种组合形式的参数，子程序可以将它们区分开来。这也是有代价的：代码看起来会有一些复杂。但它的回报通常值得我们这么去做。

通过引用传递的参数行为有一个很大的不同。当以这种形式传递参数变量时，在子程序中你对参数变量值做的任何事情都会影响到主程序中参数的值。

要调用一个以引用形式传递参数的子程序时，方法和以前一样，但有一点不同：你必须在变量名前加一个反斜线。在本节通过引用传递参数的例子中，可以像这样实现子程序的调用：

```
1 | reference_sub(\@i, \@j);
```

就像你在这里看到的一样，参数是两个数组，为了保持传递给 `reference_sub` 子程序后两者的区别，通过引用的形式进行了传递，方法就是在它们的名字前加上反斜线。

在子程序中，也有一些变化。首先，参数从 `@_` 数组中收集起来，并保存为标量变量，这是因为不管引用的是标量、数组、散列或者其他什么，引用都是存储在标量变量中的一种特殊类型的数据。如下所示，进行参数的收集：

```
1 | my($i, $j) = @_;
```

从 `@_` 数组中读取参数后，保存为标量。

对于这些引用的参数，子程序还要进行进一步的处理。当使用它们时，需要对它们进行解引用。要解引用一个被引用的参数，你需要在引用前添加上表明变量类型的符号：对于标量来说是 `$`，对于数组来说是 `@`，对于散列来说是 `%`。所以在这些变量的名字前有两个符号：从左到右分别是它们本来的符号和表明这个变量是引用的 `$` 符号。这些行：

```
1 | push(@$i, '4');
2 | shift(@$j);
```

就是子程序中操作变量的代码。`push` 向 `@i` 数组的末尾添加了“4”这个元素，而 `shift` 则移除了 `@j` 数组的第一个元素。因为这些数组是以引用的形式传递进来的，所以在子程序中它们的名字就是 `@$i` 和 `@$j`。（如果你想看一下 `@j` 数组的第三个元素，通常情况下它就是 `$j[2]`，所以你应该使用 `$j[2]`。）

在子程序中你对参数做的任何改变都会在主程序中产生影响。因为引用就是对实际参数的引用，它们并不是通过值进行传递时它们的值的拷贝。所以，就像你在例子中看到的那样，在调用子程序后，主程序中的数组也被相应的改变了：

```

1  #!/usr/bin/perl
2  # Example of pass-by-reference (a.k.a. call-by-reference)
3
4  use strict;
5  use warnings;
6
7  my @i = ('1', '2', '3');
8  my @j = ('a', 'b', 'c');
9
10 print "In main program before calling subroutine: i = " . "@i\n";
11 print "In main program before calling subroutine: j = " . "@j\n";
12
13 reference_sub(\@i, \@j);
14
15 print "In main program after calling subroutine: i = " . "@i\n";
16 print "In main program after calling subroutine: j = " . "@j\n";
17
18 exit;
19
20 #####
21 # Subroutines
22 #####
23
24 sub reference_sub {
25     my($i, $j) = @_;
26
27     print "In subroutine : i = " . "@$i\n";
28     print "In subroutine : j = " . "@$j\n";
29
30     push(@$i, '4');
31     shift(@$j);
32 }

```

This gives the following output:

```

1  In main program before calling subroutine: i = 1 2 3
2  In main program before calling subroutine: j = a b c
3  In subroutine : i = 1 2 3
4  In subroutine : j = a b c
5  In main program after calling subroutine: i = 1 2 3 4
6  In main program after calling subroutine: j = b c

```

现在，子程序可以区分开作为参数传递进去的两个数组了。在子程序中对变量进行的修改，在子程序结束后回到主程序中时仍然有效。这是通过引用传递的基本属性。

6.5 模块和子程序库

当开始收集子程序时，你会发现自己不断地把它们从现有的程序中复制粘贴到新的程序中。这样，子程序就出现了许多程序中。这会使你的程序代码列表显得有些繁琐和重复。这也会使子程序的修改变得更加复杂，因为你不得不修改所有的子程序拷贝。

总之，子程序非常棒，但如果你不得不把它们复制粘贴到你写的每一个新程序中，那就太麻烦了。所以是时候开始把子程序收集到一个便于使用的文件中了，这就是模块或者库。

这是它的工作原理。你把所有可以重复使用的子程序放进一个单独的文件里。（或者，随着你编写越来越多的代码，事情会变得复杂起来，你可能会想把它们组织到不同的文件中。）之后在你的程序只需要把文件的名字写上，然后说声变：子程序的定义就被读进来了，就像它们本身就在你的程序中一样。要实现这一点，使用 Perl 的内置函数 *use* 即可，它会把子程序的库文件读进来。

让我们把这个模块叫做 *BeginPerlBioinfo.pm* 吧。你可以把所有的子程序定义都放在里面，就像它们出现在程序代码中一样。然后就像在本书的学习过程中键入子程序的定义，你就可以创建模块了；或者，更加简便的方法，从书籍的网页上直接把它下载下来。但需要牢记一点，当创建模块或者向模块中添加东西时，模块的最后一行必须是 `1;`，否则它不会工作。这个 `1;` 应该是 *.pm* 文件的最后一行，而不是最后一个子程序的一部分。如果你忘记了这一行，你会看到类似这样的错误信息：

```
1 | BeginPerlBioinfo.pm did not return a true value at jkl line 14.  
2 | BEGIN failed--compilation aborted at jkl line 14.
```

现在，要使用 *BeginPerlBioinfo.pm* 中的任何子程序，你只需要在靠近代码顶部（靠近 *use strict* 语句）的地方中加上这样一条语句：

```
1 | use BeginPerlBioinfo;
```

注意名字中故意去掉了 *.pm*：它是 Perl 处理模块名的方式。

最后还有一点需要知道，使用模块载入子程序时，Perl 程序需要知道到哪里去找到这个模块。如果你在同一个文件夹中进行所有的工作，一切都没有问题。如果 Perl 抱怨没法找到 *BeginPerlBioinfo.pm*，那就给出模块的全路径名吧。如果全路径名是 */home/tisdall/book/BeginPerlBioinfo.pm*，在程序中就可以这样写：

```
1 | use lib '/home/tisdall/book';  
2 | use BeginPerlBioinfo;
```

还有其他告诉 Perl 去哪里寻找模块的方法，去查阅一下 *use* 的 Perl 文档吧。

从第 8 章开始，我会定义子程序并给出代码，但你应该把它们放到模块中，然后键入：

```
1 | use BeginPerlBioinfo;
```

在书籍的网站上也可以下载到这个模块。

6.6 修复代码中的 Bugs

现在让我们谈谈，当你的程序出现问题时，该如何去处理。

一个程序会因为各种各样的原因出现问题。也许它完全无法运行。看一下错误信息，尤其是错误信息的第一行或前两行，通常它会带你找到问题的所在，将是语法的某个地方，同时它还会给出相应的解决办法，就是使用正确的语法（举例来说，括号要配对，或者，每一个语句都要以一个分号结尾）。

你的程序可能会运行，但不是你期望的那样。然后你发现程序的逻辑存在一定的问題。可能在某种情况下，你应该买的时候却卖了，比如应该相减却相加了，或者当你想用 `==` 测试两个数字是否相等时却使用了赋值操作符 `=`。再或者，问题可能在于你完成任务的方案设计存在缺陷，只有当你实际尝试它的时候缺陷才会暴露出来。

然而，有的时候问题可能不是明显，这时你就不得不开挂了。

幸运的是，Perl 有好多方法可以帮助你寻找并修复程序中的 bugs。语句 `use strict;` 和 `use warnings;` 的使用应该成为一种习惯，因为使用它们你可以捕获许多错误。Perl 调试器则给了你完全的自由，可以在程序运行时对程序进行详细的检查。

6.6.1 `use warnings;` 和 `use strict;`

一般来说，当程序的语法出现错误时，可以很容易的进行识别，因为 Perl 解释器给出的错误信息通常就可以指引你找到问题的所在。但是当程序不以你期望的形式工作时，要找到问题所在通常会更加困难一些。如果你开启了警告功能、并且强制使用严格的声明，许多这样的问题都可以被捕获。

你可能注意到了，到现在为止，本书中出现的所有程序都以这样的命令解释器行起始：

```
1 |#!/usr/bin/perl -w
```

`-w` 开启了 Perl 的警告功能，这会尝试寻找代码中潜在的问题，并对此给出警告。它会找出常见的问题，比如声明了不止一次的变量之类的问题，以及不是语法错误但会导致 bugs 的东西。

开启警告的另外一种方法就是在靠近程序顶部的地方加上下面这个语句：

```
1 |use warnings;
```

如果你使用的 Perl 的版本比较老，`use warnings;` 语句可能在其中并不存在。所以如果你的 Perl 对此进行抱怨，那就把这个语句删掉，用 `-w` 命令来替代吧。你既可以在命令解释器行上使用它，也可以在命令行中使用它：

```
1 |$ perl -w my_program
```

然而，在不同的操作系统之间，使用 `use warnings;` 会更加灵活一些。所以，从现在开始，这就是我在代码中开启警告功能的方式。另外一个你应该使用的重要的帮手是下面这个语句，也把它放在靠近程序顶部（紧邻 `use warnings;`）的地方：

```
1 |use strict;
```

前面已经提到，这会强制你去声明变量。（它还有一些选项，但那已经超出本书的范围了。）它会找到拼错的变量、会干扰程序其他部分的未声明的变量，等等。



当编写 Perl 代码时，最好永远同时使用 `use strict;` 和 `use warnings;`。

6.6.2 使用注释和 Print 语句修复 Bugs

有时，通过选择性的把程序的一部分注释掉，你就可以识别出表现异常的代码，最终找到出问题的那一部分。你也可以在有问题的程序的可以部分添加 `print` 语句，来检查某个变量的行为。这些都是历史悠久的编程技术，在几乎所有的编程语言中都能很好的工作。

把代码的一部分注释掉，这会非常有用，尤其是当你从 Perl 中得到的错误信息没有直接指出出问题的那行代码时，而这时时有发生。当出现这种情况时，你可以通过不断的试验，发现当注释掉代码的一小部分时，错误信息消失了，这样你就知道是哪一部分出错了。

通过添加 `print` 语句，也可以很快查明问题所在，尤其是当你差不多已经知道是哪儿出问题时。然而，作为程序员菜鸟，你可能会发现使用 Perl 调试器要比添加 `print` 语句更容易一些。在调试器中，你可以在任意行轻松得设置 `print` 语句。比如，下面的调试器命令是要在 48 行之前打印出 `$i` 和 `$k` 的值：

```
1 | a 48 print "$i $k\n"
```

一旦你学会了如何使用它，这种方法通常就会比手工编辑 Perl 程序来添加 `print` 语句更加快捷和容易一些。使用这种方法部分是出于个人喜好的关系，因为有些极端的好的 Perl 程序员更加喜欢使用添加 `print` 语句这种古典的方法。

6.6.3 Perl 调试器

处理程序中那些不是很明显的 bugs 使，我最喜欢的方法还是使用 Perl 调试器。对于代码中存在 bugs 的程序来说，问题在于一旦程序开始运行，你看到的所有内容就是它的输出了，你无法看到程序运行的具体步骤。Perl 调试器可以让你一步一步仔细的检查程序，而这往往可以让你快速找到问题所在。你也会发现，仅需少许的练习，就可以轻松得使用它。

有些状况，Perl 调试器也不能很好的应对，比如，依赖于时间的交互式的进程。调试器每次只能检查一个程序，在检查的过程中，它会把程序中断，这就使得它没法考虑依赖于时间的其他的进程了。

对于大多数情况来说，Perl 调试器都是强大、基本的编程工具。本小节将介绍它的最主要的特性。

有 bugs 的程序

例 6.4 程序中有一些 bugs，我们来对其进行检查。程序会处理一条序列和两个碱基，（如果它能够在序列中找到这两个碱基的话）就把从这两个碱基到序列末尾的所有内容都输出出来。这两个碱基可以以命令行参数的形式传递给程序，如果不给参数的话，程序将默认使用 TA 这两个碱基。

在例 6.4中有一个新的东西。*next* 语句会影响循环中的控制流，它会立即使程序流进入循环的下一个迭代，直接跳过后面的所有内容。此外，你还需要回忆一下 `$_`，在例 5.5的 `foreach` 循环的相关内容中我们对它进行了讨论。

例 6.4：有一两个 bug 的程序

```
1  #!/usr/bin/perl
2  # Example 6-4   A program with a bug or two
3  #
4  # An optional argument, for where to start printing the sequence,
5  #   is a two-base subsequence.
6  #
7  # Print everything from the subsequence ( or TA if no subsequence
8  # is given as an argument) to the end of the DNA.
9
10 # declare and initialize variables
11 my $dna = 'CGACGTCTTCTAAGGCGA';
12 my @dna;
13 my $receivingcommittment;
14 my $previousbase = '';
15
16 my $subsequence = '';
17
18 if (@ARGV) {
19     my $subsequence = $ARGV[0];
20 }
21 else {
22     $subsequence = 'TA';
23 }
24
25 my $base1 = substr( $subsequence, 0, 1 );
26 my $base2 = substr( $subsequence, 1, 1 );
27
28 # explode DNA
29 @dna = split( '', $dna );
30
31 ##### Pseudocode of the following loop:
32 #
33 # If you've received a committment, print the base and continue.  Otherwise:
34 #
35 # If the previous base was $base1, and this base is $base2, print them.
36 #   You have now received a committment to print the rest of the string.
37 #
38 # At each loop, save the previous base.
39
40 foreach (@dna) {
41     if ($receivingcommittment) {
42         print;
43         next;
```

```

44     }
45     elsif ( $previousbase eq $base1 ) {
46         if (/ $base2/) {
47             print $base1, $base2;
48             $recievingcommitment = 1;
49         }
50     }
51     $previousbase = $_;
52 }
53
54 print "\n";
55
56 exit;

```

下面是两次运行例 6.1 的输出：

```

1 $ perl example 6-4 AA
2
3 $ perl example 6-4
4 TA

```

嗯？当使用参数 AA 调用程序时，应该输出 AAGGCGA，而没有参数运行程序时，应该输出 TAAGGCGA。在这个程序中肯定有一个 bug。但是，如果你仔细检查这个程序，其中并没有什么明显的错误。是时候使用调试器了。接下来就针对例 6.4 的真实的调试会话，其中穿插着解释发生了什么及其原因的注释。

如何启动和停止调试器

调试器以交互的形式运行，你可以通过键盘控制它。⁸启动调试器的最常用的方法，就是在命令行中给 Perl 添加 `-d` 开关。既然使用有 bug 的例 6.4 来演示调试器的使用，下面就是启动程序的方式：

```
1 | perl -d example6-4
```

另外，你也可以给命令解释器添加 `-d` 标志：

```
1 | #!/usr/bin/perl -d
```

在类似 Unix 和 Linux 这种命令解释器起作用的系统上，这种写法会自动启动调试器。要停止调试器，输入 `q` 即可。

调试器命令总结

首先，让我们试着找找当不使用参数调用例 6.4 时其中的 bug 吧：

⁸你也可以让它自动运行，把调试结果保存到文件中。

```

1 $ perl -d example6-4
2
3 Loading DB routines from perl5db.pl version 1.39_10
4 Editor support available.
5
6 Enter h or 'h h' for help, or 'man perldebug' for more help.
7
8 main::(example6-4.pl:11):      my $dna = 'CGACGTCTTCTAAGGCGA';
9   DB<1>

```

这是刚刚开始，让我们先停在这里，看看其中的一些东西。开始是一些信息，现在看来它们可能毫无意义，之后，就是一个非常棒的信息——使用命令 `h` 和 `h h` 可以获得更多的帮助。让我们试试 `h`：

```

1   DB<1> h
2 List/search source lines:          Control script execution:
3   l [ln|sub]   List source code      T           Stack trace
4   - or .      List previous/current line s [expr]   Single step [in expr]
5   v [line]    View around line       n [expr]    Next, steps over subs
6   f filename  View source in file    <CR/Enter> Repeat last n or s
7   /pattern/ ?patt? Search forw/backw r           Return from subroutine
8   M           Show module versions  c [ln|sub]  Continue until position
9 Debugger controls:                L           List break/watch/actions
10  o [...]     Set debugger options   t [n] [expr] Toggle trace [max depth] ][tra
11  <[<]|{[{}]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub] [cnd] Set breakpoint
12  ! [N|pat]   Redo a previous command B ln|*      Delete a/all breakpoints
13  H [-num]    Display last num commands a [ln] cmd  Do cmd before line
14  = [a val]   Define/list an alias    A ln|*      Delete a/all actions
15  h [db_cmd]  Get help on command     w expr      Add a watch expression
16  h h        Complete help page       W expr|*    Delete a/all watch exprs
17  |[[]db_cmd  Send output to pager    ![!] syscmd Run cmd in a subprocess
18  q or ^D    Quit                    R           Attempt a restart
19 Data Examination:      expr      Execute perl code, also see: s,n,t expr
20  x|m expr     Evals expr in list context, dumps the result or lists methods.
21  p expr       Print expression (uses script's current package).
22  S [![pat]    List subroutine names [not] matching pattern
23  V [Pk [Vars]] List Variables in Package. Vars can be ~pattern or !pattern.
24  X [Vars]     Same as "V current_package [Vars]". i class inheritance tree.
25  y [n [Vars]] List lexicals in higher scope <n>. Vars same as V.
26  e           Display thread id      E Display all thread ids.
27 For more help, type h cmd_letter, or run man perldebug for all docs.
28   DB<1>

```

这有点不容易阅读，但是你看到了调试器命令的简要总结。你也可以使用 `h h` 命令，它会给出数屏的大量信息。`| h h` 命令会逐页显示这些信息，一次只显示一页。调试器命令开头的管道会把输出传送给分页器，当你敲击键盘上的空格键时分页器通常会显示下一页。你最好尝试一下。但是现在，我们要把精力集中在少数几个最有用的命令上。但是别忘了，键入 `h` 命令会给你关于命令的帮助信息。

使用调试器逐步运行语句

言归正传。当你启动调试器后，你会看到它停止在真实的 Perl 代码的第一行上：

```
1 | main::(example6-4:11):    my $dna = 'CGACGTCTTCTAAGGCGA';
```

现在，对于调试器，有重要的一点你需要理解：它显示的是将要执行的那一行，而不是已经执行的行。

所以，实际上，例 6.4 现在还什么事情也没有做。你从命令总结中可以看到，`p` 会让调试器打印出值。如果你想让它打印 `$dna` 的值，你可以这么做：

```
1 | DB<1> p $dna
2 |
3 | DB<2>
```

因为其中没有任何东西，所以它不会有什么显示，现在它还没有看到 `$dna` 变量呢。所以你应该执行这个语句。有两个命令可以使用：`n` 和 `s` 都可以执行显示的语句。（两者的区别在于：`n` 或“next”在子程序调用时不会进入子程序，而把它看做一个单独的语句；而 `s` 或“single step”会进入子程序，并且一步步运行它。）一旦你使用了其中的一个命令，你就可以敲击 Enter 键来重复同样的命令了。

因为没有子程序，所以在选择 `n` 和 `s` 时不用左右为难，我们使用 `n`：

```
1 | DB<2> n
2 | main::(example6-4:12):    my @dna;
3 | DB<3>
```

这会显示下一行（在提示符的末尾你可以看到程序的行号）。如果你想看更多的行，可以使用 `v` 或者“view”命令：

```
1 | DB<2> v
2 | 9
3 | 10      # declare and initialize variables
4 | 11:     my $dna = 'CGACGTCTTCTAAGGCGA';
5 | 12==>   my @dna;
6 | 13:     my $receivingcommittment;
7 | 14:     my $previousbase = '';
8 | 15
9 | 16:     my $subsequence = '';
10 | 17
11 | 18:     if (@ARGV) {
12 | DB<2>
```

当前行——接下来将被执行的行——会以箭头（`==>`）突出显示。

`v` 看起来是一个非常有用的命令。通过帮助命令 `h v`，让我们来看看它的更多信息吧。

```
1 | DB<2> h v
2 | v [line]      View window around line.
```

3 DB<3>

实际上，不止这些——通过重复键入 `v` 可以持续显示程序的更多代码，减号 (`-`) 会上翻一屏。这些足够了。

既然 `$dna` 已经声明和初始化了，程序的第一个语句看起来出了一点错误：

```
1 DB<3> p $dna
2 CGACGTCTTCTAAGGCGA
3 DB<4>
```

这正是我们所期望的。这里没有 bug，所以让我们继续检查剩余的行，并把各种值打印出来：

```
1 DB<4> n
2 main::(example6-4.pl:13):      my $receivingcommittment;
3 DB<4> n
4 main::(example6-4.pl:14):      my $previousbase = '';
5 DB<4> n
6 main::(example6-4.pl:16):      my $subsequence = '';
7 DB<4> n
8 main::(example6-4.pl:18):      if (@ARGV) {
9 DB<4> p @ARGV
10
11 DB<5> v
12 15
13 16:      my $subsequence = '';
14 17
15 18==>    if (@ARGV) {
16 19:      my $subsequence = $ARGV[0];
17 20      }
18 21      else {
19 22:      $subsequence = 'TA';
20 23      }
21 24
22 DB<5> n
23 main::(example6-4.pl:22):      $subsequence = 'TA';
24 DB<5> n
25 main::(example6-4.pl:25):      my $base1 = substr( $subsequence, 0, 1 );
26 DB<5> p $subsequence
27 TA
28 DB<6> n
29 main::(example6-4.pl:26):      my $base2 = substr( $subsequence, 1, 1 );
30 DB<6> n
31 main::(example6-4.pl:29):      @dna = split( '', $dna );
32 DB<6> p $base1
33 T
34 DB<7> p $base2
35 A
```

36 | DB<8>

到现在为止，一切都和预期一样：使用的是默认子序列 TA，\$base1 和 \$base2 变量也被设成了子程序的第一个碱基 T 和第二个碱基 A。让我们继续：

```
1 | DB<8> n
2 | main::(example6-4:39):    foreach (@dna) {
3 |   DB<8> p @dna
4 | CGACGTCTTCTAAGGCGA
5 |   DB<9> p "@dna"
6 | C G A C G T C T T C T A A G G C G A
7 |   DB<10>
```

这里展示了一个 Perl 和打印数组的技巧：通常打印出来时元素之间没有空格，但是在 print 语句中通过把数组包裹进双引号中会使元素以空格分隔的形式展示出来。

一如既往，一切看起来都还正常，并且我们要进入循环了。先让我们看一下这整个的循环：

```
1 | DB<10> v
2 | 37      #
3 | 38      # At each loop, save the previous base.
4 | 39
5 | 40==>   foreach (@dna) {
6 | 41:         if ($receivingcommitment) {
7 | 42:             print;
8 | 43:             next;
9 | 44         }
10 | 45:         elsif ( $previousbase eq $base1 ) {
11 | 46:             if (/ $base2/) {
12 | DB<10> v
13 | 44         }
14 | 45:         elsif ( $previousbase eq $base1 ) {
15 | 46:             if (/ $base2/) {
16 | 47:                 print $base1, $base2;
17 | 48:                 $receivingcommitment = 1;
18 | 49             }
19 | 50         }
20 | 51:         $previousbase = $_;
21 | 52     }
22 | 53
23 | DB<10>
```

尽管 v 命令的输出结果中有一些重复的行，你还是看到了整个的循环。现在你知道这里的某个地方出现了问题：就像现在运行的这样，当你不给参数测试程序时，它会使用默认参数 TA，到现在为止它看起来还是正常的。然而，在你的测试中，它本应该打印出从 TA 第一次出现到最后的所有字符串，但实际上它却只打印出了 TA。哪里错了呢？

设置断点

要找出出错的地方，你可以在代码中设置断点。所谓断点指的就是程序中的一个点，你告诉调试器在这个地方停止执行，这样你就可以在其附近检查代码了。Perl 调试器允许你以多种方式设置断点。它们让你运行程序，只在语句到达断点的时候停止检查。通过这种方式，你就不用一步一步执行代码中的每一行了。（如果你有 5000 行的代码，而错误则发生在你读入输入的第 12000 行、敲击一行首次使用到的代码时，你会为有这样的特性而感到高兴。）

注意一旦发现起始的两个碱基、循环中打印输出剩余字符串的部分，就是开始于第 41 行的 `if` 代码块：

```
1 | if ($receivingcommittment) {
2 |     print;
3 |     next;
4 | }
```

让我们看看 `$receivingcommittment` 变量。

这里是实现的一种方法。我们在第 41 行设置断点。键入 `b 41`，然后键入 `c` 继续，程序会持续运行直到它到达第 41 行：

```
1 | DB<10> b 41
2 | DB<11> c
3 | main::(example6-4:41):      if ($receivingcommittment) {
4 | DB<11> p
5 | C
6 | DB<11>
```

最后的命令 `p`，会打印输出 `foreach` 循环到达的 `@dna` 数组的那个元素。既然你并没有为循环指定特定的变量，那么它就使用默认的 `$_` 变量。许多 Perl 命令，比如 `print` 和模式匹配，在没有其他变量可用的情况下，会操作默认的 `$_` 变量。（它是子程序用来存储参数的默认数组 `@_` 的表亲。）所以 `p` 这个调试器命令显示，你正在操作 `@dna` 数组中的第一个字符 `C`。

一切正常。但最好在变量 `$receivingcommittment` 的值发生改变时，程序能够暂停，然后一步一步运行，看看为什么程序没有打印出剩余的字符串。回忆一下，这个变量是一个标志，它的改变会让程序打印出剩余的字符串。首先让我们删除所有其他的断点：

```
1 | DB<11> B *
2 | Deleting all breakpoints...
3 | DB<12>
```

你可以像这样使用 `w` 来“watch”一下变量：

```
1 | DB<12> w $receivingcommittment
2 | DB<13> c
3 | TA
4 | Debugged program terminated. Use q to quit or R to restart,
```

```

5 | use o inhibit_exit to avoid stopping after program termination,
6 | h q, h R or h o to get additional info.
7 | DB<13>

```

等一会！当 `$receivingcommitment` 改变值时，`w` 命令应该有所显示。但是当使用 `c` 命令继续运行程序时，它直接运行到了结尾，这意味着 `$receivingcommitment` 的值从没有发生改变。所以让我们重新运行程序，并在改变值的那一行暂停一下：

```

1 | DB<13> R
2 | Warning: some settings and command-line options may be lost!
3 |
4 | Loading DB routines from perl5db.pl version 1.39_10
5 | Editor support available.
6 |
7 | Enter h or 'h h' for help, or 'man perldebug' for more help.
8 |
9 | main::(example6-4.pl:11):          my $dna = 'CGACGTCTTCTAAGGCGA';
10 | DB<12> v 47
11 | 44          }
12 | 45:          elsif ( $previousbase eq $base1 ) {
13 | 46:              if (/ $base2/) {
14 | 47:                  print $base1, $base2;
15 | 48:                  $receivingcommitment = 1;
16 | 49              }
17 | 50          }
18 | 51:          $previousbase = $_;
19 | 52      }
20 | 53
21 | DB<13> b 48
22 | DB<14> c
23 | main::(example6-4.pl:48):          $receivingcommitment = 1;
24 | DB<14> n
25 | main::(example6-4.pl:51):          $previousbase = $_;
26 | DB<14> p $receivingcommitment
27 | TA
28 | DB<15>

```

嗯？代码说它把 1 这个值赋给了变量，但是在你使用 `n` 执行代码后，尝试打印出它的值时，并没有输出正确的值。

如果你仔细检查程序，会看到在第 66 行，你把 `$receivingcommitment` 错误的拼写成了 `$receivingcommitment`。这就就是了所有的一切；修正它并重新运行一次：

```

1 | $ perl example6-4
2 | TAAGGCGA

```

成功了！

修复另一个 bug

现在，这修复了当你使用参数运行例 6.4 时的其他 bug 吗？

```
1 | $ perl example6-4 AA
2 | GACGTCTTCTAAGGCGA
```

又一次，嗯？你期望的是 AAGGCGA。在程序中是不是有另外一个 bug？让我们再一次尝试一下调试器：

```
1 | $ perl -d example6-4 AA
2 |
3 | Loading DB routines from perl5db.pl version 1.39_10
4 | Editor support available.
5 |
6 | Enter h or 'h h' for help, or 'man perldebug' for more help.
7 |
8 | main::(example6-4_fix1.pl:11): my $dna = 'CGACGTCTTCTAAGGCGA';
9 |   DB<1> n
10 | main::(example6-4_fix1.pl:12): my @dna;
11 |   DB<1> n
12 | main::(example6-4_fix1.pl:13): my $receivingcommittment;
13 |   DB<1> n
14 | main::(example6-4_fix1.pl:14): my $previousbase = '';
15 |   DB<1> n
16 | main::(example6-4_fix1.pl:16): my $subsequence = '';
17 |   DB<1> n
18 | main::(example6-4_fix1.pl:18): if (@ARGV) {
19 |   DB<1> n
20 | main::(example6-4_fix1.pl:19):     my $subsequence = $ARGV[0];
21 |   DB<1> n
22 | main::(example6-4_fix1.pl:25): my $base1 = substr( $subsequence, 0, 1 );
23 |   DB<1> n
24 | main::(example6-4_fix1.pl:26): my $base2 = substr( $subsequence, 1, 1 );
25 |   DB<1> n
26 | main::(example6-4_fix1.pl:29): @dna = split( '', $dna );
27 |   DB<1> p $subsequence
28 |
29 |   DB<2> p $base1
30 |
31 |   DB<3> p $base2
32 |
33 |   DB<4>
```

好了，因为某种原因，\$subsequence、\$base1 和 \$base2 变量都没有设置正确。为什么会这样呢？

检查一下第 19 行，在 if 语句的代码块中，你使用同样的名字 \$subsequence 声明了一个新的 my 变量。这就是你设置的变量，但在 if 语句结束后它就消失了，因为它是

一个 `my` 变量，所以它的作用域只在代码块中。

所以，又一次，通过删除第 19 行中的 `my` 声明，把它改成 `$subsequence = $ARGV[0]`；这个赋值，你修复了问题。重新运行程序：

```
1 | $ perl example6-4
2 | TAAGGCGA
3 | $ perl example6-4 AA
4 | AAGGCGA
```

最后终于成功了。

再说 `use warnings`; 和 `use strict`;

例 6.4 某种程度上是人为的。它证明，如果开启了警告模式，这些问题都可以轻松地报告出来。所以，让我们来看一个实际的例子，它展示了 `use strict`; 和 `use warnings`; 的优势，就像在本章前面讨论的那样。

如果你在原始的例 6.4 靠近程序顶部的地方添加上 `use warnings`; 指令，你会得到下面的输出：

```
1 | $ perl example6-4
2 | Name "main::recievingcommitment" used only once: possible typo at example6-4 line 50.
3 | TA
```

如你所见，警告模式立即发现了第一个 bug。它注意到有一个变量只用了一次，这通常是变量拼写错误的标志。（我一直不能正确拼写“receiving”和“commitment”。）所以修正第 66 行的拼写错误，然后重新运行程序：

```
1 | $ perl example6-4 AA
2 | substr outside of string at example6-4 line 28.
3 | Use of uninitialized value $base2 in regexp compilation at example6-4 line 48.
4 | Use of uninitialized value $base2 in print at example6-4 line 49.
5 | GACGTCTTCTAAGGCGA
```

所以，第一个 bug 被修复了。第二个 bug 仍然存在，还有一些可能难以理解的警告信息。但是只关注第一个错误信息，看到它抱怨的是第 26 行：

```
1 | my $base2 = substr($subsequence, 1, 1);
```

所以，`$subsequence` 有一些问题。通常，错误信息有一行的错位，所以很可能错误开始于之前的那一行，就是 `$subsequence` 被 `substr` 初次操作的那行。但此处并不是这种情况。

不过，警告已经直接指出了问题的所在。在这个例子中，你还要主动一些，回去看看 `$subsequence` 变量，注意到第 20 行 `if` 代码块中多了一次 `my` 声明，这导致变量不能够被正确的初始化。现在这并不一定总是一个 bug——在代码块中声明一个有作用域的变量，覆盖掉代码块外面重名的另一个变量。事实上，这是完全合法的，所以编写警告的程序员并没有把它标识成一个明显的错误。然而，在这里它看起来却导致了一个真正的问题！

最后一点:如果你回去看最原始的、有 bug 的程序,注意到在程序中没有 `use strict;`。如果你把它添加上,然后在无参数的情况下运行程序,你看到如下信息:

```
1 | $ perl example6-4
2 | Global symbol "$recievingcommitment" requires explicit package name at example6-4 line 1.
3 | Execution of example6-4 aborted due to compilation errors.
```

修正拼错的变量,然后在有参数的情况下运行程序,你将得到:

```
1 | $ perl example6-4 AA
2 | GACGTCTTCTAAGGCGA
```

你会看到 `use strict;` 对修复另一个 bug 毫无帮助。记住,最好同时使用 `use strict;` 和 `use warnings;`。

6.7 练习题

习题 6.1

编写一个子程序，把两个 DNA 字符串连接起来。

习题 6.2

编写一个子程序，报告 DNA 中每种核苷酸的百分比。你已经看到了加法操作符 +。你也会用到除法操作符 / 和乘法操作符 *。计算每种核苷酸的数目，除以 DNA 的总长，然后乘以 100 就可以得到百分比了。你的参数应该是 DNA 和你想计算的核苷酸。如果需要的话，可以用 `int` 函数来删除小数点后的数字。

习题 6.3

编写一个子程序，给用户一些提示信息，并收集用户的答案。子程序的参数应该是提示信息，而返回值应该是用户的（一行的）答案。

习题 6.4

编写一个子程序，来查找 `-help`、`-h` 和 `--help` 这样的命令行参数。回忆一下，命令行参数都在 `@ARGV` 数组中。从主程序中调用你的子程序。如果你给出了任意可用的命令行参数，当你把它们传送到子程序中时，它应该返回一个真值。在这个例子中，可以让程序打印输出 `$USAGE` 变量中的帮助信息，然后退出。

习题 6.5

编写一个子程序，来检查一下一个文件是否存在、是不是一个普通文件，是不是大小不为零。使用 `-e` 操作符（参看附录 B）。

习题 6.6

在一个子程序中使用习题 6.3，一直进行提示，直到用户输入一个有效的文件，或者已经进行了五次失败的尝试。

习题 6.7

编写一个包含子程序的模块，报告关于 DNA 序列的多种统计信息，比如它的长度、GC 含量、有没有 poly-T 序列（许多 `$DNA` 序列 5'（左）端大多数是 T 的长的延伸），或者其他感兴趣的信息。

习题 6.8

编写一个子程序，做一些生物学家通常做的事情。（这是在实验室中逛逛、写一个有用的程序的好机会！）

习题 6.9

阅读调试器的文档，通过在你的程序中运行它来熟悉调试器的使用。

习题 6.10

编写一个子程序，改变一个文件中存储在数组中的一些行。对于数组来说，通过引用的方式进行传递。给子程序传递数组的引用、一个正则表达式和一个替换正则表达式的字符串。数组中的所有行都应该通过正则表达式的查找，用替换字符串替换掉找到的匹配。

第 7 章 突变和随机化

目录

7.1 随机数生成器	126
7.2 使用随机化的一个程序	127
7.3 模拟 DNA 突变的程序	133
7.4 生成随机 DNA	144
7.5 分析 DNA	149
7.6 练习题	155

正如每一个生物学家所知道的那样，突变是生物学中的一个基本主题。在细胞中，DNA 上的突变时时刻刻都在发生着。绝大多数突变都不影响蛋白质行使功能，是良性的。也有一部分突变确实会影响到蛋白质，导致肿瘤等疾病的发生。突变也会造成后代无法存活，它们在发育过程中就会死亡；有时，突变也能导致进化的改变。许多细胞都有很复杂的机制，来对突变进行修复。

DNA 的突变可能来源于辐射、化学制剂、复制错误等原因。我们将使用 Perl 的随机数生成器，把突变看成随机化事件来对其进行建模。

随机化是一种计算机技术，它经常会在日常使用的密码等程序中突然出现，比如你生成一个不容易被猜到的密码。但随机化也是算法中的一个重要分支：许多最快的算法都用到了随机化。

使用随机化，可以来模拟和研究 DNA 突变的机制，以及突变对相关蛋白质生物活性的影响。模拟是研究系统和预测结果的一个强有力的工具，随机化让你可以更好地模拟生物系统中的“有序混沌”。使用计算机程序来模拟突变，这将有助于进化、疾病以及分裂和 DNA 修复机制等基本细胞过程的研究。细胞发育和功能的计算机模型，现在还在它们的早起阶段，在未来的几年中它将会更加精确且有用，而突变就是这些模型将要囊括在内的一个基本的生物学机制。

从编程技术以及对进化、突变和疾病建模的立场来看，随机化是一个强大的编程技巧，而幸运的是，它非常容易使用。

在本章中，我们将要完成以下内容：

- 在数组中随机选取一个索引，在字符串中随机选取一个位置：这些是在 DNA（或其他数据）中选取随机位置的基本工具
- 使用随机数对突变进行建模，学习如何随机选取 DNA 中的一个核苷酸并把它突变成其他（随机）的核苷酸
- 使用随机数来生成 DNA 序列数据集，这可以用来研究实际基因组的随机化程度
- 重复突变 DNA 来研究在进化过程中突变随时间累积的影响

7.1 随机数生成器

随机数生成器是你调用的一个子程序。对于大多数实践操作来说，你不需要知道它里面是什么。你从计算机中得到的随机数数值，和真实世界中测量到的随机事件有一定的差别，比如，检测的核衰变事件。有些计算机确实连接着盖革计数器等设备，这样就可以有一个真实随机事件的来源了。但我敢打赌，你的计算机上并没有这样的设备。你有的只是一个代替盖革计数器的算法，它就是随机数生成器。

随机数生成器输出的数字并不是真正随机的，因此它们被叫做伪随机数。一个随机数生成器，作为一种算法，是可以被预测的。随机数生成器需要一个种子作为输入，通过改变种子，你可以得到一系列不同的（伪）随机数。

随机数生成器生成的结果给出的是数值的均匀分布，这是随机化最重要的特性之一，并且很大程度上也决定了要根据期望的随机范围的大小来调整算法的使用。

对于随机数生成器来说，另一个要牢记在心的就是你初始化使用的种子本身也应该是随机选择的。如果你每次都使用同样的数字作为种子，那么每次你都将得到同样的“随机数字”序列。（这就并不随机了！）试着选一个具有随机性的种子，比如某些随时间任意改变的计算机事件计算出来的数字。¹

在接下来的例子中，我使用一个简单的方法来挑选种子，这对于大多数用途来说都是没有问题的。如果你使用随机数对存在紧要的隐私问题的数据（比如病人的记录）进行加密，你应该进一步参阅 Perl 中关于 Perl 提供给随机数生成器的几个高级选项的文档。在本书中，我使用的方法对于大多数情况的用途来说都已经足够了。

¹即使这样，对于紧要的用途来说，你还没有跳出如来佛的五指山。除非你小心地选择种子，否则黑客还是可以猜出你是如何选择种子，从而破解你的随机数和密码。本章中使用的生成种子的方法，`time|$$`，是可以被黑客中的“有志青年”所破解的。一个更好的选择是 `time() ^ ($$+<<15)`。如果程序安全非常重要，你就应该好好查阅 Perl 的文档，以及 CPAN 中的 `Math::Random` 和 `Math::TrulyRandom` 模块。

7.2 使用随机化的一个程序

例 7.1 通过一个简单的程序介绍了随机化，它通过随机组合句子的片段来构造一个故事。这并不是一个生物信息学的程序，但是我发现这是学习随机化基础知识的一个有效的方法。你将学习如何从数组中随机选取一个元素，这会在后续突变 DNA 的程序实例中得到运用。

例子声明了几个包含句子片段的数组，然后把它们随机组合成完整的句子。这是一个微不足道的孩子的游戏，但它演示了一些编程要点。

例 7.1：使用随机数的儿童游戏

```
1  #!/usr/bin/perl -w
2  # Example 7-1  Children's game, demonstrating primitive artificial intelligence,
3  #  using a random number generator to randomly select parts of sentences.
4
5  use strict;
6  use warnings;
7
8  # Declare the variables
9  my $count;
10 my $input;
11 my $number;
12 my $sentence;
13 my $story;
14
15 # Here are the arrays of parts of sentences:
16 my @nouns = (
17     'Dad',      'TV',      'Mom',      'Groucho',
18     'Rebecca', 'Harpo',    'Robin Hood', 'Joe and Moe',
19 );
20
21 my @verbs = (
22     'ran to', 'giggled with', 'put hot sauce into the orange juice of',
23     'exploded', 'dissolved', 'sang stupid songs with',
24     'jumped with',
25 );
26
27 my @prepositions = (
28     'at the store',
29     'over the rainbow',
30     'just for the fun of it',
31     'at the beach',
32     'before dinner',
33     'in New York City',
34     'in a dream',
35     'around the world',
36 );
37
38 # Seed the random number generator.
```

```

39 # time|$$ combines the current time with the current process id
40 # in a somewhat weak attempt to come up with a random seed.
41 srand( time | $$ );
42
43 # This do-until loop composes six-sentence "stories".
44 # until the user types "quit".
45 do {
46
47     # (Re)set $story to the empty string each time through the loop
48     $story = '';
49
50     # Make 6 sentences per story.
51     for ( $count = 0 ; $count < 6 ; $count++ ) {
52
53         # Notes on the following statements:
54         # 1) scalar @array gives the number of elements in the array.
55         # 2) rand returns a random number greater than 0 and
56         #    less than scalar(@array).
57         # 3) int removes the fractional part of a number.
58         # 4) . joins two strings together.
59         $sentence =
60             $nouns[ int( rand( scalar @nouns ) ) ] . " "
61             . $verbs[ int( rand( scalar @verbs ) ) ] . " "
62             . $nouns[ int( rand( scalar @nouns ) ) ] . " "
63             . $prepositions[ int( rand( scalar @prepositions ) ) ] . '. ';
64
65         $story .= $sentence;
66     }
67
68     # Print the story.
69     print "\n", $story, "\n";
70
71     # Get user input.
72     print "\nType \"quit\" to quit, or press Enter to continue: ";
73
74     $input = <STDIN>;
75
76     # Exit loop at user's request
77 } until ( $input =~ /\s*q/i );
78
79 exit;

```

下面是例 7.1 一些典型的输出：

```

1 | Joe and Moe jumped with Rebecca in New York City. Rebecca exploded
   | Groucho in a dream. Mom ran to Harpo over the rainbow. TV giggled
   | with Joe and Moe over the rainbow. Harpo exploded Joe and Moe at
   | the beach. Robin Hood giggled with Harpo at the beach.
2 |

```

```

3 | Type "quit" to quit, or press Enter to continue:
4 |
5 | Harpo put hot sauce into the orange juice of TV before dinner. Dad ran
   |   to Groucho in a dream. Joe and Moe put hot sauce into the orange
   |   juice of TV in New York City. Joe and Moe giggled with Joe and Moe
   |   over the rainbow. TV put hot sauce into the orange juice of Mom
   |   just for the fun of it. Robin Hood ran to Robin Hood at the beach.
6 |
7 | Type "quit" to quit, or press Enter to continue: quit

```

例子的结构非常简单。使用以下语句强制对变量进行声明并开启警告模式：

```

1 | use strict;
2 | use warnings;

```

之后，对变量进行声明，并使用值对数组进行初始化。

7.2.1 为随机数生成器设置种子

接下来，通过调用内置函数 `srand`，为随机数生成器设置种子。它需要一个参数，就是前面讨论的随机数生成器的种子。如前所述，为了得到一系列不同的随机数，你必须使用不同的种子。尝试把它改成像这样的语句：

```

1 | srand(100);

```

然后，多次运行该程序。每次，你都会得到完全相同的结果。²你使用的种子：

```

1 | time|$$

```

每次都会计算返回不同的种子。

`time` 返回代表时间的一个数，`$$` 返回代表运行的 Perl 程序的 ID（每次你运行程序它都会改变）的一个数，而 `|` 表示位元的或运算，它把两个数的位组合起来（更多细节请参看 Perl 文档）。还有选取种子的其他方法，但就让我们使用最流行的这种方法吧。

7.2.2 控制流

程序中的主循环是 `do-until` 循环。当你想在每次循环中采取任何行动（比如询问用户是否要继续）之前就做一些事情（比如打印出一个小故事）时，这种循环就非常方便了。`do-until` 循环首先执行代码块中的语句，然后进行测试，决定它是否应该重复执行代码块中的语句。注意，这和你以前见过的其他类型的循环正好相反，它们是先进行测试后执行代码块。

既然总是向 `$story` 变量上附加内容，那就需要在每次循环的开始先把它清空。忘记需要在特定的地方把以某种形式递增的变量进行重置，这非常常见，所以在你编程时一定要留意这一点。线索就是不断增长的长字符串或大数字。

²最新的随机数生成器会自动更改随机数序列，所以如果该实验不成功，很有可能你在使用一个非常新的随机数生成器。然而，有时你想重复一个随机数序列。注意，如果你像 `srand` 这样调用 `srand`，更新版本的 Perl 会自动给你一个好的种子。

`for` 循环包含着程序的主要工作。就像你前面看到的那样，这个循环初始化了一个计数器，执行测试，并在代码块的最后对计数器进行递增。

7.2.3 造句

在例 7.1 中，注意造句用的语句蔓延了数行代码。这有一点点复杂，而这正是整个程序的真正内容，所以附加了一些注释帮助理解它。注意语句进行了精心的格式化，这样它就整洁地排列在了八行中。变量名也是进行选择，这样这个过程就清晰了许多——你使用一个名词、一个动词、一个名词和一个介词短语进行造句。

然而，即使是这样，在中括号中也有多层嵌套的表达式，它用来指定数组的位置，要理解这些代码你需要进行一点仔细的分析。你会看到，你用以空格分隔的句子片段构造了一个字符串，并用一个句点和空格将其结束。字符串是通过多次使用点字符串连接操作符构建出来的，这些点字符串连接操作符被放在了每一行的开头，这样就使得整个语句的结构清晰了许多。

7.2.4 随机选取数组的一个元素

让我们仔细看看其中一个语句成分选择器吧：

```
1 | $verbs[int(rand(scalar @verbs))]
```

对于这种多层嵌套的括号，要由内向外进行阅读和计算。所以包裹在括号最内层的表达式是：

```
1 | scalar @verbs
```

从语句前面的注释中你可以看到，内置函数 *scalar* 返回数组元素的个数。例子中的数组 `@verbs` 有七个元素，所以这个表达式返回 7。

所以现在你得到的是：

```
1 | $verbs[int(rand(7))]
```

而嵌套在最内层的表达式就成了：

```
1 | rand(7)
```

代码中语句前帮助性的注释提醒你，这个语句返回一个大于 0、小于 7 的（伪）随机数。这个数是一个浮点数（有一个分数的十进制数）。回想一下，一个有七个元素的数组的元素索引是从 0 到 6。

所以现在你得到的类似于：

```
1 | $verbs[int(3.47429)]
```

而你想要对这个表达式进行计算：

```
1 | int(3.47429)
```

int 函数会丢掉浮点数的小数部分，仅返回它的整数部分，在这个例子中就是 3。
所以你来到了最后的一步：

```
1 | $verbs[3]
```

这会给你 `@verbs` 数组的第四个元素，注释中已经给你了足够的提示。

7.2.5 格式化

为了随机选取一个动词，你调用了几个函数：

scalar 确定数组的大小

rand 从数组大小决定的范围内选取一个随机数字

int 把 *rand* 返回的浮点数转变成用于索引数组元素的整数值

使用嵌套的括号，这些函数调用被组合到了一行中。有时，这会生成难于阅读的代码，对于作者这种辛苦工作得来的成果，某些吹毛求疵的读者可能会直喊头疼，因为它们并不讨人喜欢。你可以使用一些额外的临时变量，试着重写这些行的代码。比如，你可以这样写：

```
1 | $verb_array_size = scalar @verbs;
2 | $random_floating_point = rand ( $verb_array_size );
3 | $random_integer = int $random_floating_point;
4 | $verb = $verbs[$random_integer];
```

并且，对其他造句部分也进行类似的改写，最后通过这样的语句你就可以构造出句子了：

```
1 | $sentence = "$subject $verb $object $prepositional_phrase. ";
```

这是风格的问题。当你编程时，你总是会进行类似的抉择。例 7.1 中的排版风格是基于这样的得失权衡：既要把整个任务表达清晰（得），又要避免难于阅读的高度嵌套的函数调用（失）。使用这种排版的另一个原因就是，在后面的程序中，你经常需要从数组中随机选取一个元素，所以你将对这种特殊的函数调用的嵌套习以为常。事实上，如果你将要多次重复同样的事情，你可能会对这样的调用编写一个小的子程序。

就像在大多数代码中一样，易读性这里是这里最重要的因素。你必须能够阅读和理解代码，不管是你自己的还是别人的代码，这通常都要比实现其他的动人的目标重要，比如最快的速度、使用最少的内存以及最简练的程序。它并不总是很重要，但通常来说最好先把它写的易读一些，之后如果需要的话再返回去尝试提高其速度（或者其他）。你甚至可以把更加易读的代码作为注释写在那儿，这样阅读代码的人就能够对程序以及你是如何提高程序速度（或者其他）的有一个清晰的理解。

7.2.6 计算随机位置的另一种方法

Perl 通常有不同的方法来完成同一个任务。下面就是编写这个随机数选择的另一种方法；它使用的同样的函数调用，但是没有使用小括号：

```
1 | $verbs[int rand scalar @verbs]
```

这种函数链在 Perl 中很常见，其中的每一个函数都需要一个参数。要计算表达式，Perl 首先把 `@verbs` 作为 *scalar* 的参数，这会返回数组的大小。然后它把得到的值作为 *rand* 的参数，这会返回一个大于 0、小于数组大小的浮点数。然后，它把浮点数作为 *int* 的参数，这会返回小于浮点数的最大整数。换句话说，它计算的数字和用于数组 `@verbs` 的下标是完全一样的。

为什么 Perl 允许这样呢？因为这样的计算非常频繁，并且，根据“让计算机干活”的精神，Perl 的设计者拉里·沃尔决定让你（以及他自己）免于键入这些括号并使其配对的烦恼。

已经走了这么远了，拉里决定为了简单还要再进一步。你可以省略 *scalar* 和 *int* 函数的调用，直接使用：

```
1 | $verbs[rand @verbs]
```

这里发生了什么？既然 *rand* 已经期望一个标量值，它就会把 `@verbs` 放在一个标量上下文中，也就是简单的返回数组的大小。拉里聪明的设计了数组的下标（当然，它总是整数值），这样当需要下标时，会自动提取浮点数值值的整数部分，所以就不需要 *int* 了。

7.3 模拟 DNA 突变的程序

例 7.1 给你了突变 DNA 时需要的工具。在接下来的例子中，你将照常使用由字母 A、C、G 和 T 构成的字符串来表示 DNA。你将在字符串中随机选取位置，然后使用 *substr* 函数来改变 DNA。

这次，让我们换一种思路，在给出整个程序之前，首先来编写一些将要使用到的子程序。

7.3.1 伪代码设计

从简单的伪代码开始，这是把 DNA 中一个随机位置上的核苷酸突变成一个随机核苷酸的子程序的设计：

1. 选取 DNA 字符串中一个随机的位置。
2. 选择一个随机的核苷酸。
3. 把 DNA 随机位置上的核苷酸替换成随机的核苷酸。

这看上去简洁且直指要害，所以你决定把前两句分别写成子程序。

在字符串中选取一个随机位置

怎样才能在一个字符串中随机选取一个位置呢？回忆一下，内置函数 *length* 返回的就是字符串的长度，此外，字符串中的位置是从 0 到 *length-1* 进行编号的，就像数组中的位置一样。所以你可以使用和例 7.1 一样的通用策略，编写成子程序：

```

1 | # randomposition
2 | #
3 | # A subroutine to randomly select a position in a string.
4 | #
5 | # WARNING: make sure you call srand to seed the
6 | # random number generator before you call this function.
7 |
8 | sub randomposition {
9 |
10 |     my($string) = @_;
11 |
12 |     # This expression returns a random number between 0 and length-1,
13 |     # which is how the positions in a string are numbered in Perl.
14 |
15 |     return int(rand(length($string)));
16 | }
```

如果不计算注释的话，*randomposition* 实际上是一个简短的函数。这和例 7.1 中选取一个随机的数组元素的想法是一样的。

当然，如果你亲自编写这段代码，需要进行一点测试，来看看这个子程序能不能工作：

```

1 | #!/usr/bin/perl -w
2 | # Test the randomposition subroutine
```

```

3 |
4 | my $dna = 'AACCGTTAATGGGCATCGATGCTATGCGAGCT';
5 |
6 | srand(time|$$);
7 |
8 | for (my $i=0 ; $i < 20 ; ++$i ) {
9 |   print randomposition($dna), " ";
10 | }
11 |
12 | print "\n";
13 |
14 | exit;
15 |
16 | sub randomposition {
17 |   my($string) = @_ ;
18 |   return int rand length $string;
19 | }

```

下面是测试的一些典型输出（你的结果可能会有所不同）：

```

1 | 28 26 20 1 29 7 1 27 2 24 8 1 23 7 13 14 2 12 13 27

```

注意 for 循环的新的写法：

```

1 | for (my $i=0 ; $i < 20 ; ++$i ) {

```

这里演示了你可以在 for 循环中使用 my 对计数器变量（在这个例子中就是 \$i）进行声明，从而使其进入循环。

选择一个随机的核苷酸

接下来，让我们写一个子程序，从四个核苷酸中随机选取一个：

```

1 | # randomnucleotide
2 | #
3 | # A subroutine to randomly select a nucleotide
4 | #
5 | # WARNING: make sure you call srand to seed the
6 | # random number generator before you call this function.
7 |
8 | sub randomnucleotide {
9 |
10 |   my(@nucs) = @_ ;
11 |
12 |   # scalar returns the size of an array.
13 |   # The elements of the array are numbered 0 to size-1
14 |   return $nucs[rand @nucs];
15 | }

```


又一次，这个子程序简洁、悦目。（大多数有用的子程序都是这样的，尽管编写一个简短的子程序并不保证它是有用的。事实上，你将会看到还可以对这个子程序进行一点改进。）

让我们也对它进行以下测试：

```

1  |#!/usr/bin/perl -w
2  |# Test the randomnucleotide subroutine
3  |
4  |my @nucleotides = ('A', 'C', 'G', 'T');
5  |
6  |srand(time|$$);
7  |
8  |for (my $i=0 ; $i < 20 ; ++$i ) {
9  |    print randomnucleotide(@nucleotides), " ";
10 |}
11 |
12 |print "\n";
13 |
14 |exit;
15 |
16 |sub randomnucleotide {
17 |    my(@nucs) = @_;
18 |
19 |    return $nucs[rand @nucs];
20 |}

```

下面是一些典型的输出（它是随机的，所以理所当然，有很大的可能性你的输出会与此不同）：

```

1  |C A A A A T T T T T A C A C T A A G G G

```

把随机的核苷酸放到随机的位置

现在轮到第三个、也是最后一个子程序了，它要进行实际的突变。下面是代码：

```

1  |# mutate
2  |#
3  |# A subroutine to perform a mutation in a string of DNA
4  |#
5  |
6  |sub mutate {
7  |
8  |    my($dna) = @_;
9  |    my(@nucleotides) = ('A', 'C', 'G', 'T');
10 |
11 |    # Pick a random position in the DNA
12 |    my($position) = randomposition($dna);

```

```
13 |
14 |     # Pick a random nucleotide
15 |     my($newbase) = randomnucleotide(@nucleotides);
16 |
17 |     # Insert the random nucleotide into the random position in the DNA.
18 |     # The substr arguments mean the following:
19 |     #   In the string $dna at position $position change 1 character to
20 |     #   the string in $newbase
21 |     substr($dna,$position,1,$newbase);
22 |
23 |     return $dna;
24 | }
```

这里还是一个简短的程序。当你查看它时，会发现它阅读、理解起来都相对比较容易。通过选取一个随机的位置、随机的核苷酸，并把字符串中那个位置的核苷酸替换成那个随机的核苷酸，你实现了突变。（如果你忘记了 *substr* 是如何使用的，可以参看附录 B 或者其他的 Perl 文档。如果你像我一样，你可能不得不多次查阅文档，尤其是要确保以正确的顺序使用参数。）

这里使用的声明变量的风格有点不同。以前是在程序的开头声明变量，而这里则是在第一次使用变量的时候才对它们分别进行声明。每种编程风格都各有利弊。把所有变量放在程序的顶部是一种很好的组织形式，而且对阅读代码也有所帮助；而随用随声明在编写程序时看起来则是一种更加自然的方式。选择权在你手中。

此外，注意这个子程序的大部分是如何基于其他的子程序构建起来的，仅仅需要添加一点代码。这使得代码的易读性大大提高。此时，你可能会觉得你已经把任务进行了很好的分解，并且每一个小部分都比较容易完成，而最后它们也能很好的在一起协作。但真是这样的吗？

7.3.2 改进设计

你可能会对自己这么快就编写完程序而颇感自豪，但你注意到了一些事情。你总是要声明讨厌的 *@nucleotides* 数组变量，然后把它传递给 *randomnucleotide* 子程序。但你使用这个数组的唯一地方只在 *randomnucleotide* 子程序的内部。所以为什么不把设计改变一下呢？下面是一个新的尝试：

```
1 | # randomnucleotide
2 | #
3 | # A subroutine to randomly select a nucleotide
4 | #
5 | # WARNING: make sure you call srand to seed the
6 | #   random number generator before you call this function.
7 |
8 | sub randomnucleotide {
9 |     my(@nucs) = ('A', 'C', 'G', 'T');
10 |
11 |     # scalar returns the size of an array.
12 |     # The elements of the array are numbered 0 to size-1
```

```

13 | return $nucs[rand @nucs];
14 | }

```

注意这个函数现在没有参数了，要像这样调用它：

```

1 | $randomnucleotide = randomnucleotide( );

```

它从一个特定的数据集中选取一个随机的元素。当然，你总是在思考，并且会说“要是有一个从任意数组中随机选取一个元素的子程序该会多方便呀。我可能现在并不需要它，但我敢打赌很快我就会需要这样的子程序！”所以，你定义了两个子程序，而不是一个：

```

1 | # randomnucleotide
2 | #
3 | # A subroutine to randomly select a nucleotide
4 | #
5 | # WARNING: make sure you call srand to seed the
6 | # random number generator before you call this function.
7 |
8 | sub randomnucleotide {
9 |     my(@nucleotides) = ('A', 'C', 'G', 'T');
10 |
11 |     # scalar returns the size of an array.
12 |     # The elements of the array are numbered 0 to size-1
13 |     return randomelement(@nucleotides);
14 | }
15 |
16 | # randomelement
17 | #
18 | # A subroutine to randomly select an element from an array
19 | #
20 | # WARNING: make sure you call srand to seed the
21 | # random number generator before you call this function.
22 |
23 | sub randomelement {
24 |
25 |     my(@array) = @_;
26 |
27 |     return $array[rand @array];
28 | }

```

回头看一下，你会注意到并不需要更改 *mutate* 子程序，改变的只是 *randomnucleotide* 的内部构造，而不是它的行为。

7.3.3 组合子程序来模拟突变

现在，所有的材料都到位了，所以你要编写如例 7.2 一样的主程序，来看看新的子程序是否工作。

例 7.2: 突变 DNA

```
1  #!/usr/bin/perl -w
2  # Example 7-2   Mutate DNA
3  #  using a random number generator to randomly select bases to mutate
4
5  use strict;
6  use warnings;
7
8  # Declare the variables
9
10 # The DNA is chosen to make it easy to see mutations:
11 my $DNA = 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA';
12
13 # $i is a common name for a counter variable, short for "integer"
14 my $i;
15
16 my $mutant;
17
18 # Seed the random number generator.
19 # time|$$ combines the current time with the current process id
20 srand( time | $$ );
21
22 # Let's test it, shall we?
23 $mutant = mutate($DNA);
24
25 print "\nMutate DNA\n\n";
26
27 print "\nHere is the original DNA:\n\n";
28 print "$DNA\n";
29
30 print "\nHere is the mutant DNA:\n\n";
31 print "$mutant\n";
32
33 # Let's put it in a loop and watch that bad boy accumulate mutations:
34 print "\nHere are 10 more successive mutations:\n\n";
35
36 for ( $i = 0 ; $i < 10 ; ++$i ) {
37     $mutant = mutate($mutant);
38     print "$mutant\n";
39 }
40
41 exit;
42 #####
43 # Subroutines for Example 7-2
44 #####
45
46 # Notice, now that we have a fair number of subroutines, we
47 # list them alphabetically
```

```
48
49 # A subroutine to perform a mutation in a string of DNA
50 #
51 # WARNING: make sure you call srand to seed the
52 # random number generator before you call this function.
53
54 sub mutate {
55
56     my ($dna) = @_ ;
57
58     my (@nucleotides) = ( 'A', 'C', 'G', 'T' );
59
60     # Pick a random position in the DNA
61     my ($position) = randomposition($dna);
62
63     # Pick a random nucleotide
64     my ($newbase) = randomnucleotide(@nucleotides);
65
66     # Insert the random nucleotide into the random position in the DNA
67     # The substr arguments mean the following:
68     # In the string $dna at position $position change 1 character to
69     # the string in $newbase
70     substr( $dna, $position, 1, $newbase );
71
72     return $dna;
73 }
74
75 # A subroutine to randomly select an element from an array
76 #
77 # WARNING: make sure you call srand to seed the
78 # random number generator before you call this function.
79
80 sub randomelement {
81
82     my (@array) = @_ ;
83
84     return $array[ rand @array ];
85 }
86
87 # randomnucleotide
88 #
89 # A subroutine to select at random one of the four nucleotides
90 #
91 # WARNING: make sure you call srand to seed the
92 # random number generator before you call this function.
93
94 sub randomnucleotide {
95
96     my (@nucleotides) = ( 'A', 'C', 'G', 'T' );
```

```

97
98     # scalar returns the size of an array.
99     # The elements of the array are numbered 0 to size-1
100     return randomelement(@nucleotides);
101 }
102
103 # randomposition
104 #
105 # A subroutine to randomly select a position in a string.
106 #
107 # WARNING: make sure you call srand to seed the
108 # random number generator before you call this function.
109
110 sub randomposition {
111
112     my ($string) = @_;
113
114     # Notice the "nested" arguments:
115     #
116     # $string is the argument to length
117     # length($string) is the argument to rand
118     # rand(length($string)) is the argument to int
119     # int(rand(length($string))) is the argument to return
120     # But we write it without parentheses, as permitted.
121     #
122     # rand returns a decimal number between 0 and its argument.
123     # int returns the integer portion of a decimal number.
124     #
125     # The whole expression returns a random number between 0 and length-1,
126     # which is how the positions in a string are numbered in Perl.
127     #
128
129     return int rand length $string;
130 }

```

下面是例 7.2 的一些典型的输出：

```

1 Mutate DNA
2
3 Here is the original DNA:
4
5 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
6
7 Here is the mutant DNA:
8
9 AAAAAAAAAAAAAAAAAAAAAAGAAAAAAAAA
10
11 Here are 10 more successive mutations:
12

```

```

13 | AAAAAAAAAAAAAAAAAAAGACAAAAAAAA
14 | AAAAAAAAAAAAAAAAAAAGACAAAAAAAA
15 | AAAAAAAAAAAAAAAAAAAGACAAAAAAAA
16 | AAAAAAAAAAAAAAACAAGACAAAAAAAA
17 | AAAAAAAAAAAAAAACAAGACAAAAAAAA
18 | AAAAAAAAAAAAAAACAAGACAAAAAAAA
19 | AAAAAAAGAAAAACAAGACAAAAAAAA
20 | AAAAAATAAGAAAAACAAGACAAAAAAAA
21 | AAAAAATAAGAAAAACAAGACAAAAAAAA
22 | AAAAAATTAGAAAAACAAGACAAAAAAAA

```

例 7.2 在编程上有一定的挑战，但你最终看到（模拟的）DNA 突变时还是会颇感欣慰。编写一个图形化的展示如何呢，这样每次碱基突变的时候，它都会有一个小的爆炸特效，而且颜色也高亮显示，这样你就可以实时观看突变的发生了。

在你对此进行嘲笑之前，你应该知道好的图形化展示对于大多数程序的成功是多么重要。这听起来可能有点像是一个雕虫小技的图形，但是如果你能够演示大多数常见的突变，比如用这种方式演示 BRCA 乳腺癌基因，它就会非常有用。

7.3.4 程序中的一个 Bug?

言归正传，在查看例 7.2 的输出时你可能已经注意到了某些事情。看一下 “10 more successive mutations” 部分的前两行吧，它们是完全一样的！在欣喜若狂、对自己能够完成如此漂亮的工作而颇感自豪之际，竟然发现了一个 bug？

你该如何追踪它呢？就像在第 6 章学习的那样，你可能想用 Perl 调试器来一步步地运行程序。但是这一次，不要这么做，停下来先思考一下你的设计吧。你使用随机选取的碱基来替换随机位置上的碱基。啊！有的时候，你随机选取的某个位置上的碱基和你随机选取用来进行替换该位置碱基的碱基是完全一样的！偶尔，你用一个碱基替换了它本身！³

我们假设，你觉得这种方式不是很有用，对于每一成功的突变，你都应该看到一个碱基的改变。你该如何修改代码来实现这一点呢？让我们先从 *mutate* 子程序的伪代码开始吧：

```

1 | Select a random position in the string of DNA
2 |
3 | Repeat:
4 |
5 |     Choose a random nucleotide
6 |
7 | Until: random nucleotide differs from the nucleotide in the random position
8 |
9 | Substitute the random nucleotide into the random position in the DNA

```

这看起来应该管用，所以你修改了 *mutate* 子程序，把它改名为 *mutate_better* 子程序：

³这有多频繁呢？对于 DNA 中的每一个碱基，其出现的概率都为 1/4。对于一个由四种碱基等概率出现构成的 DNA 来说。这种情况发生的频率就高达 1/4！

```

1  # mutate_better
2  #
3  # Subroutine to perform a mutation in a string of DNA--version 2, in which
4  # it is guaranteed that one base will change on each call
5  #
6  # WARNING: make sure you call srand to seed the
7  # random number generator before you call this function.
8
9  sub mutate_better {
10
11     my($dna) = @_ ;
12     my(@nucleotides) = ('A', 'C', 'G', 'T');
13
14     # Pick a random position in the DNA
15     my($position) = randomposition($dna);
16
17     # Pick a random nucleotide
18     my($newbase);
19
20     do {
21         $newbase = randomnucleotide(@nucleotides);
22
23         # Make sure it's different than the nucleotide we're mutating
24     }until ( $newbase ne substr($dna, $position,1));
25
26     # Insert the random nucleotide into the random position in the DNA
27     # The substr arguments mean the following:
28     # In the string $dna at position $position change 1 character to
29     # the string in $newbase
30     substr($dna,$position,1,$newbase);
31
32     return $dna;
33 }

```

当你用这个子程序替换掉 *mutate* 后，运行代码，你会得到下面的输出：

```

1  Mutate DNA
2
3  Here is the original DNA:
4
5  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
6
7  Here is the mutant DNA:
8
9  AAAAAAAAAAAAAAATAAAAAAAAAAAAAA
10
11 Here are 10 more successive mutations:
12
13 AAAAAAAAAAAAAAATAAAAAAACAAAAAA

```



```
14 | AAAAAATAAAAAATAAAAAACAACAAAAA
15 | AAATATAAAAAATAAAAAACAACAAAAA
16 | AAATATAAAAAATAAAAAACAACAAAAA
17 | AATTATAAAAAATAAAAAACAACAAAAA
18 | AATTATTAAAAAATAAAAAACAACAAAAA
19 | AATTATTAAAAAATAAAAAACAACACAA
20 | AATTATTAAAAAGTAAAAACAACACAA
21 | AATTATTAAAAAGTGAAAAACAACACAA
22 | AATTATTAAAAAGTGATAAAAAACAACAA
```

看起来，在每次迭代的时候，确实都有一个碱基发生了改变。

对于声明变量还需要注意一点。在 *mutate_better* 的代码中，如果你在循环中声明了 `$newbase` 变量，因为循环被包裹在了代码块中，所以变量 `$newbase` 在循环外就是不可见的。具体来说，在突变过程中实际进行碱基替换的 `substr` 调用中，它没法使用。所以，在 *mutate_better* 中，你必须要在循环外声明这个变量。

对于那些喜欢随用随声明变量的程序员来说，这常常是一些混乱的源头，这也是一个强有力的证据，让你养成在程序顶部收集所有变量定义的习惯。

即使这样，有的时候你还是想把变量隐藏在代码块中，因为那里是你唯一会用到这个变量的地方。这时你就会想在代码块中对其进行声明。（如果代码块很长，可能就会在代码块的顶部进行声明？）

7.4 生成随机 DNA

有时出于测试的目的，生成随机的数据会非常有用。同样可以使用随机 DNA 来研究生物体中真实 DNA 的组织方式。在本小节中，我们就来编写一些生成随机 DNA 序列的程序。

这样的随机 DNA 序列在很多方面都非常有用。比如，流行的 BLAST 程序（参看第 12 章），就要依赖于随机 DNA 的属性来获得对序列相似性打分进行评估的分析和经验性结果，以及用于对“击中”进行排序的统计结果，这会被 BLAST 反馈给用户。

假设我们需要的是一系列长短不一的随机 DNA 片段。你的程序必须要设定一个最大和最小的长度，以及生成 DNA 片段的数目。

7.4.1 自下而上 vs. 自上而下

在例 7.2 中，你编写了一些最基本的子程序，然后一个子程序调用这些基本的子程序，最后实现的是主程序。如果你忽略伪代码，这就是*自下而上*设计的一个例子：从最基本的砖瓦开始，然后把它们组装成高楼大厦。

现在来看看另一种设计，它从主程序开始，然后是其中的子程序调用，当你发现需要子程序时你才编写它们。这就叫做*自上而下*设计。

7.4.2 生成一系列随机 DNA 的子程序

考虑到我们生成随机 DNA 的目标，你需要的可能是一个直接生成数据的子程序：

```
1 | @random_DNA = make_random_DNA_set( $minimum_length, $maximum_length, $size_of_set );
```

这看起来没有问题，但还是要看如何来真正完成整个任务。（对你来说这就是*自上而下*的设计！）所以你需要一步步深入，编写 *make_random_DNA_set* 子程序的伪代码：

```
1 | repeat $size_of_set times:
2 |
3 |     $length = random number between minimum and maximum length
4 |
5 |     $dna = make_random_DNA ( $length );
6 |
7 |     add $dna to @set
8 | }
9 |
10 | return @set
```

现在，继续*自上而下*的设计，你需要 *make_random_DNA* 子程序的伪代码：

```
1 | from 1 to $length
2 |
3 |     $base = randomnucleotide
4 |
5 |     $dna .= $base
```

```
6 }  
7  
8 return $dna
```

不需要更加深入了，因为在例 7.2 中你已经编写了 *randomnucleotide* 子程序。

(你是否对伪代码中不配对的大括号感到厌烦？此处，你依赖于缩进，并用右大括号来表明代码块。既然是伪代码，只要它能工作，一切都是允许的。)

7.4.3 把设计变成代码

现在我们已经有了自上而下的设计，该如何编写代码呢？我们还是继续自上问下的设计，来看看它是如何工作的。

例 7.3 按照伪代码中自上而下的设计顺序一步步前进，从主程序开始，之后才是子程序。

例 7.3：生成随机 DNA

```
1  #!/usr/bin/perl -w  
2  # Example 7-3   Generate random DNA  
3  #   using a random number generator to randomly select bases  
4  
5  use strict;  
6  use warnings;  
7  
8  # Declare and initialize the variables  
9  my $size_of_set    = 12;  
10 my $maximum_length = 30;  
11 my $minimum_length = 15;  
12  
13 # An array, initialized to the empty list, to store the DNA in  
14 my @random_DNA = ();  
15  
16 # Seed the random number generator.  
17 # time|$$ combines the current time with the current process id  
18 srand( time | $$ );  
19  
20 # And here's the subroutine call to do the real work  
21 @random_DNA =  
22     make_random_DNA_set( $minimum_length, $maximum_length, $size_of_set );  
23  
24 # Print the results, one per line  
25 print "Here is an array of $size_of_set randomly generated DNA sequences\n";  
26 print "   with lengths between $minimum_length and $maximum_length:\n\n";  
27  
28 foreach my $dna (@random_DNA) {  
29  
30     print "$dna\n";  
31 }  
32
```

```
33 print "\n";
34
35 exit;
36
37 #####
38 # Subroutines
39 #####
40
41 # make_random_DNA_set
42 #
43 # Make a set of random DNA
44 #
45 #   Accept parameters setting the maximum and minimum length of
46 #   each string of DNA, and the number of DNA strings to make
47 #
48 # WARNING: make sure you call srand to seed the
49 # random number generator before you call this function.
50
51 sub make_random_DNA_set {
52
53     # Collect arguments, declare variables
54     my ( $minimum_length, $maximum_length, $size_of_set ) = @_;
55
56     # length of each DNA fragment
57     my $length;
58
59     # DNA fragment
60     my $dna;
61
62     # set of DNA fragments
63     my @set;
64
65     # Create set of random DNA
66     for ( my $i = 0 ; $i < $size_of_set ; ++$i ) {
67
68         # find a random length between min and max
69         $length = randomlength( $minimum_length, $maximum_length );
70
71         # make a random DNA fragment
72         $dna = make_random_DNA($length);
73
74         # add $dna fragment to @set
75         push( @set, $dna );
76     }
77
78     return @set;
79 }
80
81 # Notice that we've just discovered a new subroutine that's
```

```
82 # needed: randomlength, which will return a random
83 # number between (or including) the min and max values.
84 # Let's write that first, then do make_random_DNA
85
86 # randomlength
87 #
88 # A subroutine that will pick a random number from
89 # $minlength to $maxlength, inclusive.
90 #
91 # WARNING: make sure you call srand to seed the
92 # random number generator before you call this function.
93
94 sub randomlength {
95     # Collect arguments, declare variables
96     my ( $minlength, $maxlength ) = @_;
97
98     # Calculate and return a random number within the
99     # desired interval.
100     # Notice how we need to add one to make the endpoints inclusive,
101     # and how we first subtract, then add back, $minlength to
102     # get the random number in the correct interval.
103     return ( int( rand( $maxlength - $minlength + 1 ) ) + $minlength );
104 }
105
106
107 # make_random_DNA
108 #
109 # Make a string of random DNA of specified length.
110 #
111 # WARNING: make sure you call srand to seed the
112 # random number generator before you call this function.
113
114 sub make_random_DNA {
115     # Collect arguments, declare variables
116     my ($length) = @_;
117
118     my $dna;
119
120     for ( my $i = 0 ; $i < $length ; ++$i ) {
121         $dna .= randomnucleotide();
122     }
123
124     return $dna;
125 }
126
127 }
128
129 # We also need to include the previous subroutine
130 # randomnucleotide.
```

```

131 # Here it is again for completeness.
132
133 # randomnucleotide
134 #
135 # Select at random one of the four nucleotides
136 #
137 # WARNING: make sure you call srand to seed the
138 # random number generator before you call this function.
139
140 sub randomnucleotide {
141
142     my (@nucleotides) = ( 'A', 'C', 'G', 'T' );
143
144     # scalar returns the size of an array.
145     # The elements of the array are numbered 0 to size-1
146     return randomelement(@nucleotides);
147 }
148
149 # randomelement
150 #
151 # randomly select an element from an array
152 #
153 # WARNING: make sure you call srand to seed the
154 # random number generator before you call this function.
155
156 sub randomelement {
157
158     my (@array) = @_;
159
160     return $array[ rand @array ];
161 }

```

下面是例 7.3 的输出：

```

1 Here is an array of 12 randomly generated DNA sequences
2   with lengths between 15 and 30:
3
4 TACGCTTGTGTTTTCGGGGAC
5 GGGGTGTGGTAAGGCTGTCTCAGATGTGC
6 TGAACGACAACCTCCTGGACTTTACT
7 ATCTATGCTTTGCCATGCTAGT
8 CCGCTCATTCTCTTCCTCGGC
9 TGTACCCCTAATACTACTTTAGCCGAATTTA
10 ATAGGTCGGGGCGACAGCGCCGG
11 GATTGACCTCTGTAA
12 AAAATCTCTAGGATCGAGC
13 GTATGTGCTTGGGTAAAT
14 ATGGAGTTGCGAGGAAGTAGCTGAGT
15 GGCCCATGACCAGCATCCAGACAGCA

```

7.5 分析 DNA

在处理随机化的最后这个例子中，你将收集 DNA 的一些统计信息，来回答这个问题：平均来说，对于两个随机的 DNA 序列，它们的碱基相同的百分比是多少？尽管一些简单的数学计算就可以帮你回答这个问题，但这个程序的要点在于表明你现在已经有必需的编程能力来提问并回答关于 DNA 序列的问题了。（如果你在使用真实的 DNA，比如收集到的在不同生物中略有不同的某个特定基因，这个问题就更加有趣了。稍后你可能会想尝试一下。）

让我们生成一个随机 DNA 的集合，所有的 DNA 长度都相等，然后对这个集合思考下面的问题：对于集合中成对的 DNA 序列来说，相同碱基位置的百分比平均是多少？

像平常一样，我们先尝试用伪代码勾画出程序的思路：

```

1 | Generate a set of random DNA sequences, all the same length
2 |
3 | For each pair of DNA sequences
4 |
5 |     How many positions in the two sequences are identical as a fraction?
6 |
7 | }
8 |
9 | Report the mean of the preceding calculations as a percentage

```

显而易见，要编写这个代码，你至少可以重用已经完成的一部分工作。你已经知道如何生成一系列随机的 DNA 序列。此外，虽然你还没有按照位置逐个比较两条序列碱基的子程序，但你知道如何查找 DNA 字符串中的位置。所以，这样的子程序并不难写。事实上，我们写一些伪代码，来把一个序列上的每一个核苷酸和另一个序列上相同位置的核苷酸进行比较：

```

1 | assuming DNA1 is the same length as DNA2,
2 |
3 | for each position from 1 to length(DNA)
4 |
5 |     if the character at that position is the same in DNA_1 and DNA_2
6 |
7 |         ++$count
8 |     }
9 | }
10 |
11 | return count/length

```

这个问题已经迎刃而解了。当然，你还需要编写一些代码，来挑选成对的序列，收集计算结果，最终获得结果的平均值并以百分比的形式将它报告出来。所有这些都在主程序中。例 7.4 就是这样的尝试，所有的内容都在其中。

例 7.4：计算成对随机 DNA 序列的平均一致性百分比

```

1 | #!/usr/bin/perl -w

```

```

2  # Example 7-4   Calculate the average percentage of positions that are the same
3  # between two random DNA sequences, in a set of 10 sequences.
4
5  use strict;
6  use warnings;
7
8  # Declare and initialize the variables
9  my $percent;
10 my @percentages;
11 my $result;
12
13 # An array, initialized to the empty list, to store the DNA in
14 my @random_DNA = ();
15
16 # Seed the random number generator.
17 # time|$$ combines the current time with the current process id
18 srand( time | $$ );
19
20 # Generate the data set of 10 DNA sequences.
21 @random_DNA = make_random_DNA_set( 10, 10, 10 );
22
23 # Iterate through all pairs of sequences
24 for ( my $k = 0 ; $k < scalar @random_DNA - 1 ; ++$k ) {
25     for ( my $i = ( $k + 1 ) ; $i < scalar @random_DNA ; ++$i ) {
26
27         # Calculate and save the matching percentage
28         $percent = matching_percentage( $random_DNA[$k], $random_DNA[$i] );
29         push( @percentages, $percent );
30     }
31 }
32
33 # Finally, the average result:
34 $result = 0;
35
36 foreach $percent ( @percentages ) {
37     $result += $percent;
38 }
39
40 $result = $result / scalar(@percentages);
41
42 #Turn result into a true percentage
43 $result = int( $result * 100 );
44 print "In this run of the experiment, the average percentage of \n";
45 print "matching positions is $result%\n\n";
46
47 exit;
48
49 #####
50 # Subroutines

```



```
51 #####
52
53 # matching_percentage
54 #
55 # Subroutine to calculate the percentage of identical bases in two
56 # equal length DNA sequences
57
58 sub matching_percentage {
59
60     my ( $string1, $string2 ) = @_;
61
62     # we assume that the strings have the same length
63     my ($length) = length($string1);
64     my ($position);
65     my ($count) = 0;
66
67     for ( $position = 0 ; $position < $length ; ++$position ) {
68         if (
69             substr( $string1, $position, 1 ) eq substr( $string2, $position, 1 )
70         )
71         {
72             ++$count;
73         }
74     }
75
76     return $count / $length;
77 }
78
79 # make_random_DNA_set
80 #
81 # Subroutine to make a set of random DNA
82 #
83 #   Accept parameters setting the maximum and minimum length of
84 #   each string of DNA, and the number of DNA strings to make
85 #
86 # WARNING: make sure you call srand to seed the
87 # random number generator before you call this function.
88
89 sub make_random_DNA_set {
90
91     # Collect arguments, declare variables
92     my ( $minimum_length, $maximum_length, $size_of_set ) = @_;
93
94     # length of each DNA fragment
95     my $length;
96
97     # DNA fragment
98     my $dna;
99
```

```
100     # set of DNA fragments
101     my @set;
102
103     # Create set of random DNA
104     for ( my $i = 0 ; $i < $size_of_set ; ++$i ) {
105
106         # find a random length between min and max
107         $length = randomlength( $minimum_length, $maximum_length );
108
109         # make a random DNA fragment
110         $dna = make_random_DNA($length);
111
112         # add $dna fragment to @set
113         push( @set, $dna );
114     }
115
116     return @set;
117 }
118
119 # randomlength
120 #
121 # A subroutine that will pick a random number from
122 # $minlength to $maxlength, inclusive.
123 #
124 # WARNING: make sure you call srand to seed the
125 # random number generator before you call this function.
126
127 sub randomlength {
128
129     # Collect arguments, declare variables
130     my ( $minlength, $maxlength ) = @_;
131
132     # Calculate and return a random number within the
133     # desired interval.
134     # Notice how we need to add one to make the endpoints inclusive,
135     # and how we first subtract, then add back, $minlength to
136     # get the random number in the correct interval.
137     return ( int( rand( $maxlength - $minlength + 1 ) ) + $minlength );
138 }
139
140 # make_random_DNA
141 #
142 # Make a string of random DNA of specified length.
143 #
144 # WARNING: make sure you call srand to seed the
145 # random number generator before you call this function.
146
147 sub make_random_DNA {
148
```

```

149     # Collect arguments, declare variables
150     my ($length) = @_;
151
152     my $dna;
153
154     for ( my $i = 0 ; $i < $length ; ++$i ) {
155         $dna .= randomnucleotide();
156     }
157
158     return $dna;
159 }
160
161 # randomnucleotide
162 #
163 # Select at random one of the four nucleotides
164 #
165 # WARNING: make sure you call srand to seed the
166 # random number generator before you call this function.
167
168 sub randomnucleotide {
169
170     my (@nucleotides) = ( 'A', 'C', 'G', 'T' );
171
172     # scalar returns the size of an array.
173     # The elements of the array are numbered 0 to size-1
174     return randomelement(@nucleotides);
175 }
176
177 # randomelement
178 #
179 # randomly select an element from an array
180 #
181 # WARNING: make sure you call srand to seed the
182 # random number generator before you call this function.
183
184 sub randomelement {
185
186     my (@array) = @_;
187
188     return $array[ rand @array ];
189 }

```

例 7.4 中的代码看起来和前面例子中的代码有些重复，确实是这样的。为了便于描述，我把子程序代码都放在了程序中。（在第 8 章中，你将开始使用模块，就会避免这种重复。）

下面是例 7.4 的输出：

```
1 | In this run of the experiment, the average number of
```

2 | matching positions is 24%

好吧，这看起来还算合理。你可能会说，这是显而易见的：四分之一的位点相匹配，因为一共有四种碱基。但是，这点并不足以验证基本概率，它只是让你明白，你已经有足够的编程技能傍身，来编写一些针对 DNA 序列提问和回答问题的程序了。

7.5.1 关于代码的一些注释

注意，在主程序中，当调用：

```
1 | @random_DNA = make_random_DNA_set( 10, 10, 10 );
```

时，你并不需要声明并初始化像 `$minimum_length` 这样的变量。你只需要在调用子程序时填入真实的数值即可。（然而，把这样的东西存储在程序顶部声明的变量中，通常都是一个比较好的做法，因为这样比较容易寻找并修改它们。）这个例子中，你把最大和最小的长度都设置成了 10，同时要求生成 10 条序列。

让我们重申一下刚刚解决的问题。你需要比较所有的 DNA 对，对每一对 DNA，都要计算有相同核苷酸的位置的百分比。然后，你要得到这些百分比的平均值。

在例 7.4 主程序中实现这一点的代码如下所示：

```
1 | # Iterate through all pairs of sequences
2 | for (my $k = 0 ; $k < scalar @random_DNA - 1 ; ++$k) {
3 |     for (my $i = ($k + 1) ; $i < scalar @random_DNA ; ++$i) {
4 |
5 |         # Calculate and save the matching percentage
6 |         $percent = matching_percentage($random_DNA[$k], $random_DNA[$i]);
7 |         push(@percentages, $percent);
8 |     }
9 | }
```

为了比较每一对 DNA，你使用了嵌套的循环。所谓嵌套循环就是指在一个循环中还有另一个循环。这在编程中非常常见，但一定要小心处理它们。这看起来可能有一点复杂，花些时间来看看嵌套循环是如何工作的，因为当从一个集合中选取两个（或多个）元素的组合时这很常用。

这里的嵌套循环要查看 $(n * (n - 1)) / 2$ 对序列，这是数据集大小的平方函数。它会变得非常大！试着逐渐增加数据集的大小，然后重新运行程序，你会发现运算时间增加了，而且远比线性增加要大。

看看循环是如何工作的？首先，序列 0（以 `$k` 进行索引）依次跟序列 1、2、3、……9（以 `$i` 进行索引）进行配对。之后，序列 1 依次和 2、3、……9 进行配对，如此往复，最后，序列 8 和序列 9 进行配对。（回忆一下，数组元素的计数是从 0 开始的，所以有 10 个元素的数组的最后一个元素的索引是 9。此外，回忆一下，标量 `@random_DNA` 返回的是数组中元素的数目。）

你可能发现这样做是比较值得的，就是把序列数目设置成一个小的数值，比如 3 或者 4，然后思考（手中拿着纸笔）在程序运行过程中嵌套循环是如何工作的，变量 `$k` 和 `$i` 是如何一步步改变的。或者，你也可以使用 Perl 调试器来看看到底发生了什么。

7.6 练习题

习题 7.1

编写一个程序，询问你选取一个氨基酸，然后（随机）猜测你选的是哪个氨基酸。

习题 7.2

编写一个程序，选取四个核苷酸中的一个，然后一直进行提示，直到你正确猜出选取的是哪个核苷酸。

习题 7.3

编写一个子程序，随机打乱数组的元素。子程序以一个数组作为参数，返回具有相同元素、但打乱成随机顺序的数组。原始数组中的每一个元素在输出的数组中都出现且仅出现一次，就像洗牌一样。

习题 7.4

编写一个程序来突变蛋白质序列，就像例 7.2 中突变 DNA 的代码一样。

习题 7.5

编写一个子程序，给定一个密码子（一个长 3bp 的 DNA 片段），返回密码子中的一个随机突变。

习题 7.6

有些版本的 Perl 为随机数生成器自动设置种子，这样在使用 `rand` 生成随机数之前就不用多余使用 `srand` 来设置种子了。实验一下，看看 `rand` 是否会自动调用 `srand`，还是需要你自己明确地去调用 `srand`，就像你在本章的代码中看到的那样。

习题 7.7

有时，在随机选取的时候并不是所有的选项都会被选择到。编写一个子程序，随机返回一个核苷酸，而且是按照指定的每个核苷酸的几率。给这个子程序传递四个数字作为参数，代表每种核苷酸的几率。如果每种核苷酸的几率都是 0.25，这个子程序在选取每种核苷酸时就是完全均等的。为了检查错误，子程序还要确保四个几率之和为 1。

提示：实现这一点的一种方法就是，把从 0 到 1 的范围分成四个区间，每个区间的长度和相应核苷酸的几率相对应。然后，简单的从 0 到 1 之间随机选取一个数字，看看它落在了哪个区间中，就返回相对应的核苷酸即可。

习题 7.8

这是一个有一定难度的练习题。Perl 中的 `study` 函数可能提高 DNA 或蛋白质中基序的查找速度。查看关于这个函数的 Perl 文档。它的用法非常简单：给定存储在 `$sequence` 变量中的一些序列数据，在进行搜索之前键入：

```
1 | study $sequence;
```

基于你已经阅读的文档中的相关内容，你认为 `study` 会提高 DNA 或蛋白质的搜索速度吗？

可以得到大量更多的得分！现在阅读标准模块 Benchmark 的 Perl 文档。（键入 `perldoc Benchmark`，或者访问 <http://www.perl.com> 上的 Perl 主页。）编写一个程序，在使用和不使用 `study` 的情况下，分别检测 DNA 和蛋白质基序的搜索速度，看看你的猜测是不是正确。

第 8 章 遗传密码

目录

8.1 散列	158
8.2 生物学的数据结构和算法	160
8.3 遗传密码	165
8.4 把 DNA 翻译成蛋白质	173
8.5 从文件中读取 FASTA 格式的 DNA	176
8.6 阅读框	185
8.7 练习题	190

到现在为止，我们已经使用 Perl 进行了基序的查找、模拟 DNA 突变、生成随机序列，以及把 DNA 转录成 RNA。这些都是非常重要的主题，它们可以作为你在研究生物学系统过程中使用到的计算技术的好的入门指引。

在本章中，我们将编写 Perl 程序，来模拟遗传密码是如何指导 DNA 翻译成蛋白质的。开始我先介绍散列这种数据类型。之后，在对不同的数据结构（散列、数组和数据库）如何对实验信息存取进行简要的讨论后，我们将编写一个程序，来把 DNA 翻译成蛋白质。我们也将继续探讨正则表达式，并编写处理 FASTA 文件的代码。

8.1 散列

在 Perl 中有三种主要的数据类型。你已经学习了其中的两种：标量变量和数组。现在，我们将开始学习第三种：散列（也叫做关联数组）。

散列提供了与键相关联的值的快速查找。举个例子，比如说你有一个叫做 `%english_dictionary` 的散列。（是的，散列以百分号起始。）如果你想查找单词 “recreant” 的定义，可以这样：

```
1 | $definition = $english_dictionary{'recreant'};
```

标量 `'recreant'` 是键，返回的标量定义是值。就像你在这个例子中看到的这样，当你访问单个元素时，散列（就像数组那样）把起始的字符换成了美元符号，因为散列查找返回的值是一个标量值。通过它们使用的括号类型，你可以把散列查找和数组元素区分开来：数组使用中括号 `[]`，而散列使用大括号。

如果你想给键赋值，只需要一个类似的简单语句：

```
1 | $english_dictionary{'recreant'} = "One who calls out in surrender.";
```

此外，如果你想用一些键值对初始化一个散列，方法类似于初始化数组，不同的是每一对都成了键-值对：

```
1 | %classification = (  
2 |     'dog',      'mammal',  
3 |     'robin',    'bird',  
4 |     'asp',      'reptile',  
5 | );
```

它用值 `'mammal'` 对键 `'dog'` 进行了初始化，依此类推。还有另一种书写方法，可以使其含义更加清晰一些。下面这些代码所做的事情和前面的代码完全一样，但把键-值的关系表现地更加明确一些：

```
1 | %classification = (  
2 |     'dog'    => 'mammal',  
3 |     'robin' => 'bird',  
4 |     'asp'    => 'reptile',  
5 | );
```

你可以得到散列的所有键构成的数组：

```
1 | @keys = keys %my_hash;
```

你也可以得到散列的所有值构成的数组：

```
1 | @values = values %my_hash;
```

在许多不同的情形下你都会用到散列，尤其是当你的数据以键-值形式存在时，或者你需要快速查找到某个键的值时。举个例子，在本章的后面部分，我们将编写程序，使用

散列来提取一个基因的信息。基因的名字就是键，关于基因的信息就是键的值。从数学上来讲，一个 Perl 散列表示的永远是一个有穷函数。

“散列”这个名字来源于散列函数，如果你留心去寻找，几乎关于算法的任何一本书中都会对它进行定义。让我们跳过它们深层次的工作细节，只讨论它们的表现。

8.2 生物学的数据结构和算法

生物学家研究生物学数据，并且试图阐明在生命系统中基于它的存在结构是如何发挥功能的。生物信息学常被用来尽可能地对这样的存在结构进行建模。（不要太咬文嚼字了，我只是概括而言的！）

生物信息学也会采取略有不同的方案。它会考虑对于这些数据可以做什么，然后尝试去阐明如何对它们进行组织才能实现目标。换句话说，通过以一种方便的数据结构来表征数据，它会尝试去产生一种算法。

既然你已经学习了 Perl 中的三种数据类型，就是标量、数组和散列，现在是时候来看一下与算法和数据结构相关的主题了。在第 3 章中，我们已经讨论了算法。现在讨论的重点就是算法中数据组织形式的重要性，换言之，就是算法中的数据结构。

此处最为关键的一点是，不同的算法通常需要不同的数据结构。

8.2.1 基因表达数据库

让我们考虑一个典型问题。假设你在研究一种生物，它总共大约有 30,000 个基因。（是的，没错，这就是人类。）假设你正在研究一种类型的细胞，在某个特定的环境条件下它还没有被深入研究过，你想知道，对于每一个基因来说，它是否表达了。¹你有一个很好的芯片设备，它把那个细胞的表达信息都告诉你了。现在，对于每一个基因，你要去查找一下看看在细胞中它是否表达了。你必须在你的网站上实现这种查找功能，这样在你即将发表的文章中看到你结果的访问者就可以找到基因的表达数据了。

有许多不同的方法可以实现。让我们看一下几个不同的实现方法，作为对算法和数据结构的艺术和科学的简洁的入门介绍。

你的数据是什么？为简单起见，假设你有这个生物所有基因的名字，以及在你的实验中表示表达水平的基因的表达数值。所有未表达的基因的表达数值都是 0。

8.2.2 使用未排序数组的基因表达数据

现在，假设你想知道基因是否表达了，而不是具体的表达水平，并且你想使用数组来解决这个编程问题。毕竟，现在你已经对数组非常熟悉了。你该怎么做呢？

你可能只在数组中存储那些表达基因的基因名，而丢弃其他的基因名。假如有 8,000 个表达基因。之后，要进行任何查询，只需要遍历数组，把查询的基因名和数组中的每一个基因名进行比较，直到你找到它或者没有找到它但已经到达了数组的末尾。

这样是可行的，但也存在问题。最主要的，它非常慢。如果你只是偶尔查询一下，这并不是问题，但如果有许多人访问你的网站对这套新的表达数据进行查询，问题就大了。平均来说，查找一个表达基因需要遍历 4,000 个基因名字，而查找一个未表达的基因则需要进行 8,000 次比较。

此外，如果某个人查找的是你的研究中没有的一个基因，因为你丢弃了所有未表达基因的基因名，所以你就没法对其进行回应。查询给出的结果是没有找到这个基因，而不是说要查询的基因并不在你的实验结果中的错误信息。如果要查询的基因虽然并不在你的研究结果中，但却在这类细胞中表达（你刚好错过了它），那这个查询就是一个假阴性了。

¹对于非生物学家：当一个基因转录成 RNA 后，就可以进一步翻译出蛋白质了，就说这个基因表达了。

你可能更希望遇到这种情况时，你的程序可以报告给用户，说那个名字的基因并没有在实验中被研究。

所以你打算把 30,000 个基因都存储到数组中。（当然，现在进行查找会更慢一些。）但是，如何把表达基因和未表达基因区分开来呢？你可以把每个基因的名字都存储到数组中，并且把表达测量值附加到每个基因名的后面，然后你就可以准确无误的知道某个基因是不是在你的实验中并不存在了。

然而，这个程序仍然有点慢。你仍然不得不去遍历整个数组，直到你找到那个基因或者确定它并没有被研究为止。如果它是数组中的第一个元素，你立马就可找到它，否则你可能不得不等到它遍历到数组的最后一个元素。平均来说，你将不得不遍历一半的数组。另外，你还不得不把需要查找的基因名和数组中的基因名一个一个进行比较。对于每次查询来说，平均都要进行 15,000 次的比较，这非常慢。（实际上，在现代的计算机上，这其实也并不是慢的可怕。但是我想指出这一点，这样的东西确实意味着一个运行很慢的程序。）

另外一个问题就是，在一个标量中你存储了两个值：基因名和表达测量值。处理这样的数据，你必须还要把基因名和基因的表达测量值分割开来。

虽然有这些缺点，但这种方法确实是可行的。现在，我们来讨论一下另外一种方法。

8.2.3 使用排序数组和折半查找的基因表达数据

你可能尝试把所有的基因名按照字母顺序排序后存储在数组中，然后使用下面这种查找技术。首先，看一下中间的元素。（就像我们已经看到的，使用 `scalar @array` 表达式你可以得到数组的大小）。按照字母顺序，如果你的基因名排在中间元素的前面，你就可以忽略数组的后半部分了，并找到数组剩余的前半部分的中间元素。如此循环往复，每一步都可以把查找范围缩小到前一步元素数目的一半，直到最终找到对应的匹配，或者发现根本不存在。下面是用伪代码进行实现：

```
1 | Given a sorted array, and an element:
2 |
3 | Until you find the element or discover it's not there,
4 |
5 |     Pick the midpoint of the array, $array[scalar(@array)/2]
6 |
7 |     Compare your element with the element at the midpoint
8 |
9 |     If that matches your element, you're done.
10 |
11 |     Else, ignore the half of the array that your element is not in
12 | }
```

在 Perl 中要按照字母顺序比较两个字符串，你可以使用 `cmp` 操作符，如果两个字符串一样它会返回 0，如果它们按照字母顺序排列就会返回 -1，如果它们按照字母顺序的逆序排列就会返回 1。比如，下面这个会返回 0：

```
1 | 'ZZZ' cmp 'ZZZ';
```

这个返回-1:

```
1 | 'AAA' cmp 'ZZZ';
```

最后, 这个返回 1:

```
1 | 'ZZZ' cmp 'AAA';
```

这种算法叫做折半查找, 它会明显提高在数组中进行查找的速度。比如, 要查找 30,000 个基因, 最多只需要大约 15 次的循环即可。(和未排序数组平均进行 15,000 次的比较相比。)当然, 你还必须要对列表进行排序, 这也需要一定的时间。如果你需要不停地增加元素, 你就不得不把它们插入到合适的位置, 或者把它们添加到末尾然后对整个数组进行重新排序。所有这样的插入或者排序都可能会相当的慢。但是, 如果你仅需要进行一次排序, 然后进行大量的查找, 折半查找还是值得考虑的。

既然我们已经谈到了它, 就来看看如何对数据进行排序。这是按照字母顺序对由字符串构成的数组进行排序的方法:

```
1 | @array = sort @array;
```

这是以升序对由数字构成的数组进行排序的方法:

```
1 | @array = sort { $a <=> $b } @array;
```

还可以进行多种其他形式的排序, 但这些是最常见的。更多的细节, 可以参看 Perl 文档中关于 *sort* 函数的说明。

8.2.4 使用散列的基因表达数据

你还可以使用散列来查找你数据中的某个基因。要实现这一点, 你需要以基因名作为键、以表达测量值作为值载入散列。然后对散列一个简单的调用, 使用待查找基因的基因名作为键, 就可以返回这个基因的实验结果, 这就是你要的答案。与把基因名和表达值存储到一个标量字符串中相比, 这个过程要清晰多了。在这里, 键是一个标量, 而值则是另外一个标量。

此外, 取决于散列的构建方式, 你可以很快得到你要的答案, 因为现在的散列都不需要进行繁琐的查找就可以找到某个键的值。使用散列通常要比折半查找快很多。此外, 你还可以知道查找的基因在数据中是否存在, 因为你可以明确询问某个散列值是否被定义了, 就像这样:

```
1 | if( defined $myhash{'mykey'} ) { ... }
```

另外, 如果你开启了警告模式, 你会得到一个错误信息, 因为你提到的是一个未定义的值。

相比于折半查找, 散列的另一个优势在于, 你可以向散列中添加或删除元素, 而不需要对整个数组进行重排序。

最后, 因为散列是作为一个基本的数据类型内置在 Perl 中的, 所以它们非常容易使用, 而且你不需要进行太多的编程就可以实现你的目的。通常情况下, 节省编程的时间要

比节省程序运行的时间更加重要一些。我在第 3 章中提到过这一点，但此处还是有必要强调一下。对于一个程序员来说，懒惰的方法通常都是最有效的方法：让机器来干活吧！

但是，不要想当然的认为散列永远都是最好的方法。比如，散列并不以排序的顺序存储其中的元素，所以如果你需要以排序的方式查看数据，就不得不对它进行明确的排序，就像这样：

```
1 | @sorted_keys = sort keys %my_hash;
```

这样就可以了，但对于大的数组来说，它可能会有点慢。（当然，你也可以对值进行排序。）

对于这个表达数据的例子，我们总结一下关于数据结构的讨论，下面是对 Perl 中不同数据结构属性的信息描述，包括对基因名数据集进行的查找、添加和删减、以及保持排序顺序：

- 如果你只需要看看某个东西是不是在数据集中，并不需要按照顺序把它们罗列出来，那就使用散列。
- 如果你需要一个排序的数据集以及相对快速的查找，而不需要频繁添加或删减元素，使用排序数组结合折半查找就可以了。
- 如果你不需要对元素进行排序，但是需要快速找到最新添加的元素，使用数组并结合 Perl 函数 *push* 和 *pop* 就可以了。
- 如果你不需要对元素进行排序，但是需要添加元素，使用 Perl 的数组结合函数 *push* 和 *shift* 就可以了。当总是需要移除“最老”的元素（待在数组中时间最长的元素）时，这种方案尤其有用。

更多信息，可以参看附录 A 和 *Mastering Algorithms with Perl* (O' Reilly 出版)，尤其是后者。

8.2.5 关系数据库

数据库是存储和访问海量数据的程序。它们提供了最常用的数据类型形式在算法中使用。有一些流行的数据库，其中一些非常好的还是免费的（最好的那些都非常昂贵），而 Perl 提供了对所有最流行数据库的访问方法。比如，Perl 中的 DBI 模块，提供了方便的方法，可以在 Perl 程序中对关系数据库进行访问。

大多数数据库都叫做关系型，这描述了它们存储数据的方式。这种类型数据库的另外一个比较常见的名字是关系数据库管理系统，简称 RDMS。

关系数据库把数据组织成表格进行存储。数据通常通过一种查询语言进行输入和提取，它叫做结构化查询语言，简称 SQL。这是一种非常简单的语言，可以在表格中访问数据，同时跟随表格之间的链接。

关系数据库是存储和提取海量数据最流行的方法，但它们确实需要一定的学习。对关系数据库进行编程已经超出了本书讨论的范畴，但如果你最终需要使用 Perl 进行大量的编程，你会发现知道使用数据库的基础知识是一个宝贵的技能。参看第 13 章中的相关讨论。

尤其是，把你的基因表达数据存储到一个关系数据库中，然后在程序中使用来对网站上的查询做出回应，这完全合情合理。

8.2.6 DBM

Perl 有一个简单的、内置的方法来存储散列数据，叫做数据库管理器（DBM）。它使用起来非常简单：在启动之后，它把一个散列“绑定”到计算机硬盘上的一个文件，这样你就可以把散列保存下来以便日后对其进行重用了。这实际上是一个简单（且非常有用）的数据库。除了初始化以外，你就像使用散列一样使用它。你可以把你的基因和表达数据存储到一个 DBM 文件中，然后像散列一样使用它。在第 10 章中有关于 DBM 的更多讨论。

8.3 遗传密码

遗传密码就是细胞把包含在 DNA 中的信息翻译成氨基酸的方式，氨基酸进而形成在细胞中真正发挥功能的蛋白质。

8.3.1 背景

此处是这对非生物学家的简短介绍。

如前所述，DNA 编码蛋白质的一级结构（也就是氨基酸序列）。DNA 有四种核苷酸，而蛋白质有 20 种氨基酸。编码的过程就是从 DNA 中找到三个核苷酸，把它们作为一组“翻译”成一个氨基酸或者终止信号。这样每一组的三个核苷酸叫做密码子。稍后我们将会看到编码和翻译过程的细节。

事实上，转录首先利用 DNA 制造 RNA，然后翻译再利用 RNA 制造蛋白质。这就是分子生物学的中心法则。但是，在本课程中，我将把从 DNA 到蛋白质的整个过程不准确地简称为“翻译”。

之所以要进行这样的区分，是因为通过使用字符串来表征 DNA、RNA 和蛋白质，在计算机中可以非常简单地对整个过程进行模拟。事实上，就像在第 4 章中演示的那样，把 DNA 转录成 RNA 确实非常简单。在你的计算机模拟中，可以简单的跳过这一步，因为它只不过是把一个字母换成了另一个字母而已。（当然，细胞中的真实过程要复杂得多。）

注意，使用四种碱基，DNA 的每三个碱基构成一组，这样可以表征 $4 \times 4 \times 4 = 64$ 可能的氨基酸。因为只有 20 种氨基酸外加一个终止信号，遗传密码进化出了冗余性，所以某些氨基酸由不止一个密码子所表征。每一种可能的 DNA 三碱基——每一个密码子——都表征某个氨基酸（除了三个密码子表征终止信号外）。

图 8.1 中的表格展示了不同的碱基是如何组合形成一个氨基酸的。对于遗传密码，有一些有趣的现象需要注意一下。对于我们的目的来说，最重要的就是冗余性——不止一个密码子翻译成同一个氨基酸。很快你就会看到，我们将使用字符类和正则表达式对其进行编程。²

细胞中的翻译机器实际上会从 RNA 的某处起始，并“读取”一个接一个的密码子，把编码的氨基酸附加到不断增长的蛋白质序列的尾部。例 8.1 模拟了这个过程，一次读取 DNA 字符串的三个碱基，然后把编码的氨基酸符号连接到不断增长的蛋白质字符串的尾部。在细胞中，当遇到三个终止密码子中的任意一个时，这个过程就会停止。

8.3.2 把密码子翻译成氨基酸

第一个任务，就是让后面的程序实现翻译的过程，就是把三个核苷酸的密码子翻译成氨基酸。对于研究三个核苷酸密码子编码一个氨基酸的遗传密码来说，这是最重要的一步。

下面是一个子程序，在给定三字母的 DNA 密码子后，它会返回一个（以单字母缩写表示的）氨基酸：

```
1 | # codon2aa
2 | #
```

²此外，还要注意图 8.1 中的遗传密码是基于 RNA 的，尿嘧啶替代胸腺嘧啶出现在了其中。在我们的程序中，我们将直接把 DNA 翻译成氨基酸，所以我们的代码中将使用胸腺嘧啶而不是尿嘧啶。

		Second Position									
		U		C		A		G			
First Position	U	UUU	Phe	UCU	Ser	UAU	Tyr	UGU	Cys	U	Third Position
		UUC		UCC		UAC		UGC		C	
		UUA	Leu	UCA		UAA	Stop	UGA	Stop	A	
		UUG		UCG		UAG	Stop	UGG	Trp	G	
	C	CUU	Leu	CCU	Pro	CAU	His	CGU	Arg	U	
		CUC		CCC		CAC		CGC		C	
		CUA		CCA		CAA	Gln	CGA		A	
		CUG		CCG		CAG		CGG		G	
	A	AUU	Ile	ACU	Thr	AAU	Asn	AGU	Ser	U	
		AUC		ACC		AAC		AGC		C	
		AUA		ACA		AAA	Lys	AGA	Arg	A	
		AUG	Met (start)	ACG		AAG		AGG		G	
	G	GUU	Val	GCU	Ala	GAU	Asp	GGU	Gly	U	
		GUC		GCC		GAC		GGC		C	
		GUA		GCA		GAA	Glu	GGA		A	
		GUG		GCG		GAG		GGG		G	

图 8.1: 遗传密码

```
3 | # A subroutine to translate a DNA 3-character codon to an amino acid
4 |
5 | sub codon2aa {
6 |     my($codon) = @_ ;
7 |
8 |     if ( $codon =~ /TCA/i ) { return 'S' } # Serine
9 |     elsif ( $codon =~ /TCC/i ) { return 'S' } # Serine
10 |    elsif ( $codon =~ /TCG/i ) { return 'S' } # Serine
11 |    elsif ( $codon =~ /TCT/i ) { return 'S' } # Serine
12 |    elsif ( $codon =~ /TTC/i ) { return 'F' } # Phenylalanine
13 |    elsif ( $codon =~ /TTT/i ) { return 'F' } # Phenylalanine
14 |    elsif ( $codon =~ /TTA/i ) { return 'L' } # Leucine
15 |    elsif ( $codon =~ /TTG/i ) { return 'L' } # Leucine
16 |    elsif ( $codon =~ /TAC/i ) { return 'Y' } # Tyrosine
17 |    elsif ( $codon =~ /TAT/i ) { return 'Y' } # Tyrosine
18 |    elsif ( $codon =~ /TAA/i ) { return '_' } # Stop
19 |    elsif ( $codon =~ /TAG/i ) { return '_' } # Stop
20 |    elsif ( $codon =~ /TGC/i ) { return 'C' } # Cysteine
21 |    elsif ( $codon =~ /TGT/i ) { return 'C' } # Cysteine
22 |    elsif ( $codon =~ /TGA/i ) { return '_' } # Stop
23 |    elsif ( $codon =~ /TGG/i ) { return 'W' } # Tryptophan
24 |    elsif ( $codon =~ /CTA/i ) { return 'L' } # Leucine
25 |    elsif ( $codon =~ /CTC/i ) { return 'L' } # Leucine
26 |    elsif ( $codon =~ /CTG/i ) { return 'L' } # Leucine
27 |    elsif ( $codon =~ /CTT/i ) { return 'L' } # Leucine
```

```

28  elif ( $codon =~ /CCA/i ) { return 'P' } # Proline
29  elif ( $codon =~ /CCC/i ) { return 'P' } # Proline
30  elif ( $codon =~ /CCG/i ) { return 'P' } # Proline
31  elif ( $codon =~ /CCT/i ) { return 'P' } # Proline
32  elif ( $codon =~ /CAC/i ) { return 'H' } # Histidine
33  elif ( $codon =~ /CAT/i ) { return 'H' } # Histidine
34  elif ( $codon =~ /CAA/i ) { return 'Q' } # Glutamine
35  elif ( $codon =~ /CAG/i ) { return 'Q' } # Glutamine
36  elif ( $codon =~ /CGA/i ) { return 'R' } # Arginine
37  elif ( $codon =~ /CGC/i ) { return 'R' } # Arginine
38  elif ( $codon =~ /CGG/i ) { return 'R' } # Arginine
39  elif ( $codon =~ /CGT/i ) { return 'R' } # Arginine
40  elif ( $codon =~ /ATA/i ) { return 'I' } # Isoleucine
41  elif ( $codon =~ /ATC/i ) { return 'I' } # Isoleucine
42  elif ( $codon =~ /ATT/i ) { return 'I' } # Isoleucine
43  elif ( $codon =~ /ATG/i ) { return 'M' } # Methionine
44  elif ( $codon =~ /ACA/i ) { return 'T' } # Threonine
45  elif ( $codon =~ /ACC/i ) { return 'T' } # Threonine
46  elif ( $codon =~ /ACG/i ) { return 'T' } # Threonine
47  elif ( $codon =~ /ACT/i ) { return 'T' } # Threonine
48  elif ( $codon =~ /AAC/i ) { return 'N' } # Asparagine
49  elif ( $codon =~ /AAT/i ) { return 'N' } # Asparagine
50  elif ( $codon =~ /AAA/i ) { return 'K' } # Lysine
51  elif ( $codon =~ /AAG/i ) { return 'K' } # Lysine
52  elif ( $codon =~ /AGC/i ) { return 'S' } # Serine
53  elif ( $codon =~ /AGT/i ) { return 'S' } # Serine
54  elif ( $codon =~ /AGA/i ) { return 'R' } # Arginine
55  elif ( $codon =~ /AGG/i ) { return 'R' } # Arginine
56  elif ( $codon =~ /GTA/i ) { return 'V' } # Valine
57  elif ( $codon =~ /GTC/i ) { return 'V' } # Valine
58  elif ( $codon =~ /GTG/i ) { return 'V' } # Valine
59  elif ( $codon =~ /GTT/i ) { return 'V' } # Valine
60  elif ( $codon =~ /GCA/i ) { return 'A' } # Alanine
61  elif ( $codon =~ /GCC/i ) { return 'A' } # Alanine
62  elif ( $codon =~ /GCG/i ) { return 'A' } # Alanine
63  elif ( $codon =~ /GCT/i ) { return 'A' } # Alanine
64  elif ( $codon =~ /GAC/i ) { return 'D' } # Aspartic Acid
65  elif ( $codon =~ /GAT/i ) { return 'D' } # Aspartic Acid
66  elif ( $codon =~ /GAA/i ) { return 'E' } # Glutamic Acid
67  elif ( $codon =~ /GAG/i ) { return 'E' } # Glutamic Acid
68  elif ( $codon =~ /GGA/i ) { return 'G' } # Glycine
69  elif ( $codon =~ /GGC/i ) { return 'G' } # Glycine
70  elif ( $codon =~ /GGG/i ) { return 'G' } # Glycine
71  elif ( $codon =~ /GGT/i ) { return 'G' } # Glycine
72  else {
73      print STDERR "Bad codon \"$codon\"!!\n";
74      exit;
75  }
76 }

```


这个代码非常清晰、简单，其排版布局也使得整个过程一目了然。然后，它运行起来却要耗费一定的时间。比如，对于代表甘氨酸的密码子 GGT，它需要一个一个的进行测试，直到到达最后一行才会测试成功，这是一个大量的字符串比较。但不管怎么说，这些代码实现了最终的目的。

在代码中关于错误信息的部分出现了一些新的东西。回忆一下第 4 章中的文件句柄，以及它们是如何访问文件中的数据。在第 5 章中，我们提到 STDIN 这个特殊的文件句柄会从键盘上读取用户的输入。STDOUT 和 STDERR 也是特殊的文件句柄，在 Perl 程序中你总是可以使用它们。STDOUT 把输出定向到屏幕（通常情况下）或者其他标准的输出位置。在 `print` 语句中没有指定文件句柄的时候，默认就会使用 STDOUT。`print` 语句可以使用一个文件句柄作为可选的参数，但到目前为止，我们都把结果直接打印到了默认的 STDOUT。在这个例子中，错误信息被定向到了 STDERR，通常情况下就是被打印到屏幕，但在许多计算机系统中，它们可以被定向到一个特定的错误文件或者其他地方。另外，有时你会想把 STDOUT 定向到一个文件或者其他地方，而让 STDERR 错误信息显示在你的屏幕上。我之所以提到这些选项，是因为你很可能在 Perl 代码中碰到它们，但在本书中我们不会过多地使用它们（更多内容请参看附录 B）。

8.3.3 遗传密码的冗余性

我已经提到过遗传密码的冗余性，而刚才的子程序也清晰的展示了这种冗余性。能够在你的子程序中将这种冗余性表现的淋漓尽致，这可能比较有趣。注意这些具有冗余性的密码子的前两个碱基通常都是一样的，只有第三个碱基发生了变化。在正则表达式中你已经使用过字符集，可以来匹配任意的一个字符集合。现在，让我们尝试把子程序重写一下，对每一个冗余的密码子组都只进行一次测试：

```

1  # codon2aa
2  #
3  # A subroutine to translate a DNA 3-character codon to an amino acid
4  #   Version 2
5
6  sub codon2aa {
7      my($codon) = @_;
8
9      if ( $codon =~ /GC./i)          { return 'A' }    # Alanine
10     elsif ( $codon =~ /TG[TC]/i)    { return 'C' }    # Cysteine
11     elsif ( $codon =~ /GA[TC]/i)    { return 'D' }    # Aspartic Acid
12     elsif ( $codon =~ /GA[AG]/i)    { return 'E' }    # Glutamic Acid
13     elsif ( $codon =~ /TT[TC]/i)    { return 'F' }    # Phenylalanine
14     elsif ( $codon =~ /GG./i)        { return 'G' }    # Glycine
15     elsif ( $codon =~ /CA[TC]/i)    { return 'H' }    # Histidine
16     elsif ( $codon =~ /AT[TCA]/i)   { return 'I' }    # Isoleucine
17     elsif ( $codon =~ /AA[AG]/i)    { return 'K' }    # Lysine
18     elsif ( $codon =~ /TT[AG]|CT./i){ return 'L' }    # Leucine
19     elsif ( $codon =~ /ATG/i)        { return 'M' }    # Methionine
20     elsif ( $codon =~ /AA[TC]/i)    { return 'N' }    # Asparagine
21     elsif ( $codon =~ /CC./i)        { return 'P' }    # Proline
22     elsif ( $codon =~ /CA[AG]/i)    { return 'Q' }    # Glutamine

```

```

23 elif ( $codon =~ /CG.|AG[AG]/i) { return 'R' } # Arginine
24 elif ( $codon =~ /TC.|AG[TC]/i) { return 'S' } # Serine
25 elif ( $codon =~ /AC./i) { return 'T' } # Threonine
26 elif ( $codon =~ /GT./i) { return 'V' } # Valine
27 elif ( $codon =~ /TGG/i) { return 'W' } # Tryptophan
28 elif ( $codon =~ /TA[TC]/i) { return 'Y' } # Tyrosine
29 elif ( $codon =~ /TA[AG]|TGA/i) { return '_' } # Stop
30 else {
31     print STDERR "Bad codon \"$codon\"!!\n";
32     exit;
33 }
34 }

```

使用字符集和正则表达式，现在的代码清晰的展示了遗传密码的冗余性。此外，还要注意，现在表示氨基酸的单字母代码已经是按照字母顺序排列的了。

像 `[TC]` 的字符集匹配一个单字符，匹配 `T` 或者匹配 `C`。`.` 是一个正则表达式，它匹配除换行符以外的任意字符。代表缬氨酸的 `/GT./i` 表达式匹配 `GTA`、`GTC`、`GTG` 和 `GTT`，所有这些密码子编码的都是缬氨酸。（当然，点号匹配任意其他字符，但是我们假设的是 `$codon` 只包含 `A`、`C`、`G` 和 `T` 四个字符。）正则表达式后面的 `i` 表示既匹配大写字母，也匹配小写字母，比如 `/T/i` 就匹配 `T` 或者 `t`。

这些正则表达式中一个新的特性就是使用了竖线或者管道 (`|`) 来分隔两种匹配选择。因此，对于丝氨酸来说，`/TC.|AG[TC]/` 匹配的是 `/TC./` 或者 `/AG[TC]/`。在这个程序中，对于每一个正则表达式来说你仅仅需要两种选择，但你可以自己的喜好和需要使用尽可能多的竖线。

你也可以用小括号把正则表达式的一部分进行分组，并在其中使用竖线。举个例子，`/give me a (break|meal)/` 能匹配 “give me a break” 或者 “give me a meal”。

8.3.4 使用散列表示遗传密码

如果你考虑使用散列来完成这个翻译过程，你会看到这才是比较自然的方法。对于每一个密码子键，都有一个氨基酸值相对应。下面是代码：

```

1 #
2 # codon2aa
3 #
4 # A subroutine to translate a DNA 3-character codon to an amino acid
5 # Version 3, using hash lookup
6
7 sub codon2aa {
8     my($codon) = @_;
9
10    $codon = uc $codon;
11
12    my(%genetic_code) = (
13
14        'TCA' => 'S',    # Serine
15        'TCC' => 'S',    # Serine

```

```
16 'TCG' => 'S',    # Serine
17 'TCT' => 'S',    # Serine
18 'TTC' => 'F',    # Phenylalanine
19 'TTT' => 'F',    # Phenylalanine
20 'TTA' => 'L',    # Leucine
21 'TTG' => 'L',    # Leucine
22 'TAC' => 'Y',    # Tyrosine
23 'TAT' => 'Y',    # Tyrosine
24 'TAA' => '_',    # Stop
25 'TAG' => '_',    # Stop
26 'TGC' => 'C',    # Cysteine
27 'TGT' => 'C',    # Cysteine
28 'TGA' => '_',    # Stop
29 'TGG' => 'W',    # Tryptophan
30 'CTA' => 'L',    # Leucine
31 'CTC' => 'L',    # Leucine
32 'CTG' => 'L',    # Leucine
33 'CTT' => 'L',    # Leucine
34 'CCA' => 'P',    # Proline
35 'CCC' => 'P',    # Proline
36 'CCG' => 'P',    # Proline
37 'CCT' => 'P',    # Proline
38 'CAC' => 'H',    # Histidine
39 'CAT' => 'H',    # Histidine
40 'CAA' => 'Q',    # Glutamine
41 'CAG' => 'Q',    # Glutamine
42 'CGA' => 'R',    # Arginine
43 'CGC' => 'R',    # Arginine
44 'CGG' => 'R',    # Arginine
45 'CGT' => 'R',    # Arginine
46 'ATA' => 'I',    # Isoleucine
47 'ATC' => 'I',    # Isoleucine
48 'ATT' => 'I',    # Isoleucine
49 'ATG' => 'M',    # Methionine
50 'ACA' => 'T',    # Threonine
51 'ACC' => 'T',    # Threonine
52 'ACG' => 'T',    # Threonine
53 'ACT' => 'T',    # Threonine
54 'AAC' => 'N',    # Asparagine
55 'AAT' => 'N',    # Asparagine
56 'AAA' => 'K',    # Lysine
57 'AAG' => 'K',    # Lysine
58 'AGC' => 'S',    # Serine
59 'AGT' => 'S',    # Serine
60 'AGA' => 'R',    # Arginine
61 'AGG' => 'R',    # Arginine
62 'GTA' => 'V',    # Valine
63 'GTC' => 'V',    # Valine
64 'GTG' => 'V',    # Valine
```

```

65 'GTT' => 'V',      # Valine
66 'GCA' => 'A',      # Alanine
67 'GCC' => 'A',      # Alanine
68 'GCG' => 'A',      # Alanine
69 'GCT' => 'A',      # Alanine
70 'GAC' => 'D',      # Aspartic Acid
71 'GAT' => 'D',      # Aspartic Acid
72 'GAA' => 'E',      # Glutamic Acid
73 'GAG' => 'E',      # Glutamic Acid
74 'GGA' => 'G',      # Glycine
75 'GGC' => 'G',      # Glycine
76 'GGG' => 'G',      # Glycine
77 'GGT' => 'G',      # Glycine
78 );
79
80 if(exists $genetic_code{$codon}) {
81     return $genetic_code{$codon};
82 }else{
83     print STDERR "Bad codon \"\$codon\"!!\n";
84     exit;
85 }
86 }

```

子程序非常简单：它先初始化了一个散列，然后对散列中单个的参数依次进行查找。散列有 64 个键，每一个键都代表一个密码子。

注意，有 *exists* 这样一个函数，如果散列中存在 *\$codon* 这个键，他就会返回 *true*。它的作用和 *codon2aa* 子程序前两个版本中的 *else* 语句完全一样。³

另外，还要注意，为了让子程序既可以处理大写字母也可以处理小写字母，你把输入的参数都转换成了大写字母，这样就可以和 *%genetic_code* 散列中的数据进行比较了。你不能把正则表达式作为散列的键，它必须是简单的标量值，比如一个字符串或者一个数字，所以必须首先进行大小写的转换。（还有一种选择，你可以让散列的大小翻倍。）类似的，也不能把字符集作为散列的键，所以对于 64 个密码子你都必须单个进行指定。

你可能会想为什么要费事的把代码中最后的那一部分放在子程序中呢，为什么不仅仅声明并初始化散列、然后不需要进入子程序而是直接对散列进行查询呢？好吧，实际上子程序还对不存在的键进行了一点错误检查，这样在你使用散列的时候，就不用每次都自己进行错误检查了，因为子程序已经帮你做了。

另外，把这部分代码放在子程序中，也是为未来考虑的一点保险措施。使用我们的子程序，你编写的代码只需要负责密码子的翻译即可，这样就可以很方便的转换为另一种进行翻译的方式。也许在将来 Perl 会添加进一种新的数据类型，也有可能你想从数据库或者 DBM 文件中进行查询。这样，你需要做的仅仅是修改子程序的内部代码而已。只要对子程序的接口界面保持不变——也就是说，只要它还是把一个密码子作为参数并返回一个单字母表示的氨基酸——在程序的其他部分你就不需要担心它是如何实现翻译的了。我们的子程序成了一个黑盒子。这是使用子程序对程序进行模块化和组织的一个非常显著的优势。

³在散列中，一个键可能存在，但它的值可能是未定义的。*defined* 函数能检查值是不是已经被定义了。另外，当然，值也可能是 0 或者空字符串，在这种情况下，像 *if (\$hash{\$key})* 这样的测试会失败，因为，即使键存在而且值也被定义了，但在条件测试中，这样的值会被测试为 *false*。

之所以使用子程序表示遗传密码，还有一个好的生物学上的原因。事实上，因为对于哺乳动物、植物、昆虫和酵母来说，DNA 编码氨基酸的方式都有所不同，对于线粒体来说更是如此，所以有不止一种遗传密码。因此如果你对遗传密码进行了模块化，你只需要对程序进行简单的修改就可以适用于一系列的物种了。

散列的另一优势是它非常快。不幸的是，我们的子程序在每次被调用时都要声明整个散列，即使是仅进行一次查询也是如此。这并不高效，事实上，它还有些慢。还有其他更加快速的方法，可以把遗传密码散列作为全局变量只声明一次，但这对于现在的我们来说还有点遥不可及。我们现在这个版本的优势在于它非常易读。所以，让我们对这个散列版本的 *codon2aa* 感到心满意足，并把它放到 *BeginPerlBioinfo.pm* 文件（参看第 6 章）的模块中吧。

现在我们已经找到一个比较满意的方法，把密码子翻译成了氨基酸，在接下来的小节和例子中我们将会使用它。

8.4 把 DNA 翻译成蛋白质

例 8.1展示了新的 *codon2aa* 子程序如何把一整个 DNA 序列翻译成蛋白质。

例 8.1：把 DNA 翻译成蛋白质

```

1  #!/usr/bin/perl -w
2  # Example 8-1   Translate DNA into protein
3
4  use strict;
5  use warnings;
6  use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8  # Initialize variables
9  my $dna      = 'CGACGTCTTCGTACGGGACTAGCTCGTGTCGGTCGC';
10 my $protein = '';
11 my $codon;
12
13 # Translate each three-base codon into an amino acid, and append to a protein
14 for ( my $i = 0 ; $i < ( length($dna) - 2 ) ; $i += 3 ) {
15     $codon = substr( $dna, $i, 3 );
16     $protein .= codon2aa($codon);
17 }
18
19 print "I translated the DNA\n\n$dna\n\n into the protein\n\n$protein\n\n";
20
21 exit;
```

就像在第 6 章中讨论的那样，要使它正常工作，你需要把提供子程序的 *BeginPerlBioinfo.pm* 模块放到一个程序可以找到的单独的文件中。你还需要把 *codon2aa* 子程序添加到这个文件中。还有一种办法，你可以直接把子程序 *condon2aa* 的代码添加到例 8.1 中的程序中，然后移除对 *BeginPerlBioinfo.pm* 模块的引用。

下面是例 8.1 的输出：

```

1  I translated the DNA
2
3  CGACGTCTTCGTACGGGACTAGCTCGTGTCGGTCGC
4
5      into the protein
6
7  RRLRTGLARVGR
```

例 8.1 中的所有元素你都已经在前面见到过了，除了循环遍历 DNA 的方法，就是下面这个语句：

```

1  for(my $i=0; $i < (length($dna) - 2); $i += 3) {
```

回忆一下，`for` 由用两个分号隔开的三部分组成。第一部分初始化一个计数器：`my $i=0` 静态限定了 `$i` 变量的范围，这样它就只能在代码块中被使用了，而代码中其他

部分出现的其他 `$i`（好吧，在这个例子中，并没有其他这样的变量，但确实可能会出现这种情况）现在在代码块中则是不可用的。`for` 循环的第三部分会在代码块中的所有语句执行完后、重新回到循环顶部前增加计数器。

```
1 | $i += 3
```

因为在遍历 DNA 时每次要处理三个碱基，所以你把计数器增加三。
`for` 循环中间的第二部分测试判断是否要继续循环：

```
1 | $i < (length($dna) - 2)
```

关键点在于，如果有两个、一个或者没有碱基剩余时，你应该退出程序，因为它们不足以构成一个密码子了。现在，对于一个特定长度的 DNA 字符串来说，碱基位置的计数就是从 0 到 `length-1`。所以如果位置计数器 `$i` 增加到了 `length-2`，就只剩余两个碱基（`length-2` 和 `length-1` 位置处的碱基）了，这时你应该退出程序。当位置计数器 `$i` 小于 `length-2` 时，剩余的碱基至少还有三个，这足以构成一个密码子了。所以要想测试成功，只能：

```
1 | $i < (length($dna) - 2)
```

（注意一下，小于号后面的表达式是如何被包裹在小括号中的；我们将在第 9 章的第 9.3.1 小节中对其进行讨论。）

这行代码：

```
1 | $codon = substr ($dna, $i 3);
```

实际上从 DNA 中提取了三碱基的密码子。通过调用 `substr` 函数，提取出 `$dna` 字符串上位置 `$i` 处长度为 3 的子字符串，并把它保存到了变量 `$codon` 中。

如果你知道，你需要进行很多从 DNA 到蛋白质的翻译，你可以把例 8.1 转换成一个子程序。当你编写一个子程序时，你需要考虑你想把那些参数传递给子程序。所以你意识到，早晚有一天会遇到这样的情况，你有一些很长的 DNA 序列，但却只想翻译整个序列中指定的一部分。你是否应该给子程序添加两个参数来指定起始和终止位点呢？你可以这么做，但你决定先不这么做。这是一种主观判断——把代码集合分解成有用片段的艺术的一部分。但是最好有一个只负责翻译过程的子程序，然后，如果有需要，你可以把它作为从序列中选择终点的更大的子程序的一部分。此处的考虑是，你通常仅仅需要把整条序列都进行翻译，所以每次都键入 0 作为起始点、`length($dna)-1` 作为终止点可能会让人厌烦。当然，这取决于你手头的工作，所以此处的选择仅供你在编写代码时思考之用。

你应该移除末尾输出信息的 `print` 语句，因为与子程序相比，它更加适合于主程序。

不管怎样，你已经思考了整个设计思路，现在只需要一个子程序，它需要一个包含 DNA 的参数，并返回翻译后的肽链：

```
1 | # dna2peptide
2 | #
3 | # A subroutine to translate DNA sequence into a peptide
4 |
```

```

5 sub dna2peptide {
6
7     my($dna) = @_;
8
9     use strict;
10    use warnings;
11    use BeginPerlBioinfo;      # see Chapter 6 about this module
12
13    # Initialize variables
14    my $protein = '';
15
16    # Translate each three-base codon to an amino acid, and append to a protein
17    for(my $i=0; $i < (length($dna) - 2) ; $i += 3) {
18        $protein .= codon2aa( substr($dna,$i,3));
19    }
20
21    return $protein;
22 }

```

现在把子程序 *dna2peptide* 添加到 *BeginPerlBioinfo.pm* 模块中吧。

注意，在你把例 8.1 转换成子程序时，删除了一个变量：变量 *\$codon*。为什么呢？

好吧，一个原因就是你可以把它删除。在例 8.1 中，你使用 *substr* 从 *\$dna* 中提取密码子，把它保存到变量 *\$codon* 中，然后传递到子程序 *codon2aa* 中。新的方法删除这个中间人。直接把提取密码子的对 *substr* 的调用作为参数传递给子程序 *codon2aa*，这样值像以前一样传递了进去，但却没有必要先把它复制到变量 *\$codon* 中去了。

这在某种程度上提高了效率和速度。因为复制字符串是计算机程序做的非常慢的一件事情，删除一大批字符串的复制，是提高程序运行速度一种简单、有效的方法。

但是这有没有削弱程序的易读性呢？这由你来判断。我觉得有那么一点，但是对于我来说，无论如何，循环前面的注释看起来已经让一切都清晰明了了。编写易读的代码非常重要，所以如果你确实需要大幅提升子程序的速度，但却发现使代码更加难读了，一定要确保包含足够的注释，让读者可以理解所发生的一切。

use 函数的调用第一次被包含在了子程序中而不是主程序中：

```

1 use strict;
2 use warnings;
3 use BeginPerlBioinfo;

```

这可能和主程序中的调用存在冗余，但这没有任何坏处（Perl 会进行检查，并且只载入模块一次）。如果在一个没有载入这些模块的模块中调用这个子程序，它还是可以正常工作。

现在，让我们来看看如何处理文件中的 DNA。

8.5 从文件中读取 FASTA 格式的 DNA

在生物信息学短暂的历史中，许多不同的生物学家和程序员发明了各种在计算机文件中格式化序列数据的方法，导致生物信息学家不得不去处理这些不同的格式。我们需要从这些文件中提取出序列数据和注释信息，对于每种不同的格式，这都需要编写代码进行处理。

有许多这样的格式，仅仅针对 DNA 的常用格式可能就有 20 种之多。当你在实验室中分析序列时，格式的这种多样化会让人头疼不已：因为对于你用来分析序列的各种程序，都需要把一种格式转换成另一种格式。下面是几个最流行的格式：

FASTA

FASTA 和基本局部相似性比对搜索技术 (BLAST⁴, Basic Local Alignment Search Technique) 程序非常流行，它们都使用 FASTA 格式。因为它的简洁性，FASTA 格式可能是所有格式中除 GenBank 格式以外使用最为广泛的一种格式。

Genetic Sequence Data Bank (GenBank⁵)

GenBank 收集了所有公开发表的遗传数据。除了 DNA 序列，它还包括了许多相关的信息。它非常重要，所以我们将第 10 章中对 GenBank 文件进行深入探讨。

欧洲分子生物学实验室 (EMBL, European Molecular Biology Laboratory)

EMBL 数据库大体上和 GenBank、DDBJ (日本 DNA 数据库, DNA Data Bank of Japan) 存储有相同的数据，但格式稍有不同。

简单数据，或美国应用生物系统公司 (ABI, Applied Biosystems) 测序仪的输出

这是没有进行任何格式化的 DNA 序列数据，仅仅是代表碱基的字符而已。ABI 的测序机器和其他机器以及程序把它直接输出到文件中。

蛋白质识别资源 (PIR, Protein Identification Resource)

PIR 是一个良好组织的蛋白质序列数据集。

遗传学电脑集团 (GCG, Genetics Computer Group)

Accelrys 的 GCG 程序 (亦称 GCG Wisconsin package) 在许多大型研究机构中使用。要想被它们的程序使用，数据必须以 GCG 格式进行存储。

在这六种序列格式中，GenBank 和 FASTA 是迄今为止最常用的格式。接下来的几个小节将向你介绍 FASTA 格式数据的读取和处理过程。

8.5.1 FASTA 格式

让我们编写一个子程序，来处理 FASTA 格式的数据。它本身就非常有用，而且可以作为后续章节处理 GenBank、PDB 和 BLAST 的热身。

FASTA 格式基本上就是序列数据行，在其末尾有换行符，这样就可以把它打印到纸上或者显示在计算机屏幕上。行的长度没有特别指定，但是为了兼容，最好把长度限制在 80 个字符以内。此外，还有一个头信息，就是文件开头以大于号 > 起始的一行或数行，它可以包含任意文字 (或者没有文字)。通常，标题行包括 DNA 或者它的来源基因的名字，一般用竖线将其和序列的其他信息、生成它的实验或者其他类似的非序列的信息分隔开来。

⁴译者注：BLAST 是 Basic Local Alignment Search Tool 的缩写。

⁵译者注：GenBank 指的是 Genetic Sequence Database。

许多使用 FASTA 格式的软件都坚持认为只能有一行头信息，其他则允许多行头信息的存在。我们的子程序对一行或多行头信息以及以 # 起始的注释都能够进行处理。

下面是一个 FASTA 文件。我们把它叫做 *sample.dna*，并且在许多程序中都将使用到它。你应该从书籍网站上复制、下载它，或者用你自己的数据制作出自己的文件。

```

1 | > sample dna | (This is a typical fasta header.)
2 | agatggcggcgctgaggggtcttgggggctctaggccggccacctactgg
3 | tttgcagcgggagacgacgcatggggcctgcgcaataggagtacgctgcct
4 | gggaggcggtgactagaagcgggaagtagttgtgggcgcctttgcaaccgcc
5 | tgggacgccgccgagtggtctgtgcagggttcgcggtcgctggcggggggt
6 | cgtgagggagtgcgccgggagcggagatatggaggagatggttcagacc
7 | cagagcctccagatgccggggaggacagcaagtccgagaatgggggagaat
8 | gcgcccactctactgcatctgccgcaaaccggacatcaactgcttcatgat
9 | cgggtgtgacaactgcaatgagtgggttccatggggactgcatccggatca
10 | ctgagaagatggccaaggccatccgggagtggtactgtcgggagtgcaga
11 | gagaaagaccccaagctagagattcgctatcggcacaagaagtcacggga
12 | gcgggatggcaatgagcgggacagcagtgagccccgggatgagggtggag
13 | ggcgcaagaggcctgtccctgatccagacctgcagcgccgggcagggtca
14 | gggacaggggttggggccatgcttgctcggggctctgcttcgccccacaa
15 | atcctctccgcagcccttggtggccacaccagccagcatcaccagcagc
16 | agcagcagcagatcaaacggtcagcccgcatgtgtggtgagtgtgaggca
17 | tgtcggcgcaactgaggactgtggtcactgtgatttctgtcgggacatgaa
18 | gaagttcgggggcccccaacaagatccggcgagaagtgcgggctgcgccagt
19 | gccagctgcggggcccggaatcggtacaagtacttcccttcctcgctctca
20 | ccagtgcgcccctcagagtccctgccaaaggccccgccggccactgcccac
21 | ccaacagcagccacagccatcacagaagttagggcgcatccgtgaagatg
22 | agggggcagtgggcgtcatcaacagtcaaggagcctcctgaggctacagcc
23 | acacctgagccactctcagatgaggaccta

```

8.5.2 读取 FASTA 文件的设计

在第 4 章中，你学习了如何读入序列数据。此处，你只需要将其进行扩展以便能够处理标题行。你还将学习到如何丢弃空行以及以井号（磅字符）# 起始的行，也就是 Perl 和其他语言以及文件格式中的注释。（这样的行在刚刚展示的 FASTA 文件 *sample.dna* 中并没有出现。）

当读入数据时有两种选择。你可以从打开的文件中一次读入一行，随读入随进行处理。或者，你也可以一次性把整个文件都读入到数组中，然后对数组进行操作。对于非常大的文件来说，尤其是你寻找一些小的信息片段时，最好一次只读入文件的一行。（这是因为把一个大的文件读入数组会占用大量的内存空间。如果你的计算机不够强大，可能会导致系统崩溃。）

对于小的、正常大小的文件来说，把所有数据读入数组的好处在于，之后你可以轻松地遍历数据对它进行操作。这就是我们的子程序所做的事情，但是一定要记住，对于大文件来说这种方法可能会导致内存空间问题，还有其他的方法可以采用。

让我们编写一个子程序，给定一个包含 FASTA 格式数据的文件名参数，它会返回序列数据。

在动手之前，你应该考虑你是否只需要一个子程序，还是一个子程序打开并读取文件、另一个子程序调用它并提取序列数据。我们使用两个子程序，一定要牢记在心，每次当你需要编写类似的程序处理其他格式时，你都可以重用这里处理任意文件的子程序。

我们由伪代码开始：

```
1 | subroutine get data from a file
2 |
3 |   argument = filename
4 |
5 |   open file
6 |     if can't open, print error message and exit
7 |
8 |   read in data and
9 |
10 |  return @data
11 |}
12 |
13 | Subroutine extract sequence data from fasta file
14 |
15 |   argument = array of file data in fasta format
16 |
17 |   Discard all header lines
18 |   (and blank and comment lines for good measure)
19 |   If first character of first line is >, discard it
20 |
21 |   Read in the rest of the file, join in a scalar,
22 |   edit out nonsequence data
23 |
24 |   return sequence
25 |}
```

在第一个从文件中获取数据的子程序中，存在这样一个问题，当文件不能被读取时，最好的处理办法是什么。此处，我们采用了最极端的方案：尖叫“着火了！”然后跳出。但是你可能不一定想让你的程序在无法打开文件时立即停止。也许，你想通过键盘或者网页向用户询问文件名，并给他们三次机会来键入正确的文件名。又或者，如果文件无法打开，你就使用默认的文件进行替代。

当你无法打开文件时，也许你可以返回 `false` 值，比如一个空数组。然后，调用这个子程序的程序就可以退出，重试，或者其他它想进行的操作。但是，如果你成功打开了文件，但却完全是空的呢？这时，你成功打开了文件，并返回一个空数组，而调用这个子程序的程序可能会错误的认为文件无法打开。所以，这种方案并不可取。

还有其他的选择，比如返回特殊的“未定义”（`undefined`）值。我们还是言归正传，但牢记这么一点是非常重要的，处理错误很重要，有时非常晦涩，这也是编写强健代码的一部分，这种代码可以很好的处理异常情况。

第二个子程序，处理存储 FASTA 格式序列的数组，并返回未格式化的序列字符串。

8.5.3 读取 FASTA 文件的子程序

既然你已经思考了问题、编写了伪代码、想到了设计子程序的不同方案以及不同选择的利弊，现在就可以真正开始编写代码了：

```
1  # get_file_data
2  #
3  # A subroutine to get data from a file given its filename
4
5  sub get_file_data {
6
7      my($filename) = @_;
8
9      use strict;
10     use warnings;
11
12     # Initialize variables
13     my @filedata = ( );
14
15     unless( open(GET_FILE_DATA, $filename) ) {
16         print STDERR "Cannot open file \"$filename\"\n\n";
17         exit;
18     }
19
20     @filedata = <GET_FILE_DATA>;
21
22     close GET_FILE_DATA;
23
24     return @filedata;
25 }
26
27 # extract_sequence_from_fasta_data
28 #
29 # A subroutine to extract FASTA sequence data from an array
30
31 sub extract_sequence_from_fasta_data {
32
33     my(@fasta_file_data) = @_;
34
35     use strict;
36     use warnings;
37
38     # Declare and initialize variables
39     my $sequence = '';
40
41     foreach my $line (@fasta_file_data) {
42
43         # discard blank line
```

```

44 |     if ($line =~ /\s*$/) {
45 |         next;
46 |
47 |         # discard comment line
48 |     } elsif ($line =~ /\s*#/) {
49 |         next;
50 |
51 |         # discard fasta header line
52 |     } elsif ($line =~ /^>/) {
53 |         next;
54 |
55 |         # keep line, add to sequence string
56 |     } else {
57 |         $sequence .= $line;
58 |     }
59 | }
60 |
61 | # remove non-sequence data (in this case, whitespace) from $sequence string
62 | $sequence =~ s/\s//g;
63 |
64 | return $sequence;
65 | }

```

注意，在 *extract_sequence_from_fasta_data* 的代码中你并没有检查文件中存储的内容：它是不是真的是以 FASTA 格式存储的 DNA 或者蛋白质序列？当然，你可以编写一个子程序——把它叫做 *is_fasta*——来检查数据看看它是不是我们所期望的那样。但这里我把它留作课后练习。

对于 *extract_sequence_from_fasta_data* 子程序要进行一些注释。下面这行代码包括一个在循环中使用的变量的声明。

```

1 | foreach my $line (@fasta_file_data) {

```

在 for 循环中你已经见过这样的情况。在使用 *\$line* 的地方使用 *my* 对变量进行声明，这非常方便，因为它们往往都有比较常见的名字，而且在循环外不会被使用到。

一些正则表达式需要进行简要的注释。这一行代码：

```

1 | if ($line =~ /\s*$/) {

```

其中，*\s* 匹配空白，也就是空格、制表符、换页符、回车符或者换行符。*\s** 匹配任意数量的空白（即使没有）。*^* 匹配行首，而 *\$* 则匹配行尾。所以综合起来，这个正则表达式匹配的是没有或者只有空白的空行。

这个正则表达式表示的是行首没有或者只有空白、之后紧跟井号：

```

1 | } elsif ($line =~ /\s*#/) {

```

这个正则表达式匹配的是行首的大于号：

```
1 } elsif($line =~ /^>/) {
```

最后，下面这个语句删除空白，包括换行符：

```
1 $sequence =~ s/\s//g;
```

我们已经把这两个新的子程序放到了我们的 *BeginPerlBioinfo.pm* 模块中。现在，我们来为这些子程序编写一个主程序，看看它的输出。首先，还需要编写一个子程序，来处理长序列的输出。

8.5.4 输出格式化的序列数据

当你试图打印输出“原始的”序列数据时，如果数据长度远远超过纸张的宽度，会出现问题。实际上，如果你尝试让它适合纸张，80 个字符大约是你应该使用的最大的长度。我们编写一个 *print_sequence* 子程序，它以一些序列和行的长度作为参数，把序列打断成要求长度的多行并打印输出出来。它和 *dna2peptide* 子程序有很高的相似度。这是代码：

```
1 # print_sequence
2 #
3 # A subroutine to format and print sequence data
4
5 sub print_sequence {
6
7     my($sequence, $length) = @_;
8
9     use strict;
10    use warnings;
11
12    # Print sequence in lines of $length
13    for ( my $pos = 0 ; $pos < length($sequence) ; $pos += $length ) {
14        print substr($sequence, $pos, $length), "\n";
15    }
16 }
```

代码依赖于 *substr* 的行为，它会在字符串后面输出部分子字符串，即使其长度小于要求的长度。你可以看到在 *BeginPerlBioinfo.pm* 模块（参看第 6 章）中出现了新的 *print_sequence* 子程序。记住，一定要把语句 1；作为模块的最后一行。例 8.2 演示的是主程序。

例 8.2：读取 FASTA 文件并提取序列数据

```
1 #!/usr/bin/perl -w
2 # Example 8-2    Read a fasta file and extract the sequence data
3
4 use strict;
5 use warnings;
6 use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8 # Declare and initialize variables
```

```
9 | my @file_data = ();
10 | my $dna      = '';
11 |
12 | # Read in the contents of the file "sample.dna"
13 | @file_data = get_file_data("sample.dna");
14 |
15 | # Extract the sequence data from the contents of the file "sample.dna"
16 | $dna = extract_sequence_from_fasta_data(@file_data);
17 |
18 | # Print the sequence in lines 25 characters long
19 | print_sequence( $dna, 25 );
20 |
21 | exit;
```

下面是例 8.2 的输出：

```
1 | agatggcggcgctgaggggtcttgg
2 | gggctctaggccggccacctactgg
3 | tttgcagcggagacgacgcatgggg
4 | cctgcgcaataggagtacgtgcct
5 | gggaggcgtgactagaagcggaagt
6 | agttgtgggcgcctttgcaaccgcc
7 | tgggacgccgccgagtggctgtgc
8 | aggttcgcgggtcgctggcgggggt
9 | cgtgagggagtgcgccgggagcgga
10 | gatatggagggagatggttcagacc
11 | cagagcctccagatgccggggagga
12 | cagcaagtccgagaatggggagaat
13 | gcgcccattctactgcatctgccgca
14 | aaccggacatcaactgcttcatgat
15 | cgggtgtgacaactgcaatgagtgg
16 | ttccatggggactgcatccggatca
17 | ctgagaagatggccaaggccatccg
18 | ggagtgggtactgtcgggagtgcaga
19 | gagaaagaccccaagctagagattc
20 | gctatcggcacaagaagtcacggga
21 | gcgggatggcaatgagcgggacagc
22 | agtgagccccgggatgagggtggag
23 | ggcgcaagaggcctgtccctgatcc
24 | agacctgcagcgccgggcagggtca
25 | gggacaggggttggggccatgcttg
26 | ctcggggctctgcttcgccccacaa
27 | atcctctccgcagcccttggtggcc
28 | acaccagccagcatcaccagcagc
29 | agcagcagcagatcaaacggtcagc
30 | ccgcatgtgtggtgagtgtgaggca
31 | tgtcggcgcaactgaggactgtggtc
32 | actgtgatttctgtcgggacatgaa
33 | gaagttcgggggccccacaagatc
```

```
34 | cggcagaagtgccggctgcgccagt
35 | gccagctgcggggcccggaatcgta
36 | caagtacttcccttcctcgctctca
37 | ccagtgacgccctcagagtccctgc
38 | caaggccccgccggccactgcccac
39 | ccaacagcagccacagccatcacag
40 | aagttagggcgcatccgtgaagatg
41 | agggggcagtggcgtcatcaacagt
42 | caaggagcctcctgaggctacagcc
43 | acacctgagccactctcagatgagg
44 | accta
```

8.5.5 读入 DNA 输出蛋白质的主程序

现在，这是本小节的最后一个程序。我们向上面的程序中添加从 DNA 到蛋白质的翻译过程，最后输出蛋白质而不是 DNA。注意例 8.3 是多么的精炼！随着你不断向我们的模块中积累有用的子程序，编写程序会越来越容易。

例 8.3：读取 DNA 的 FASTA 文件，翻译成蛋白质，并格式化输出

```
1 | #!/usr/bin/perl -w
2 | # Example 8-3   Read a fasta file and extract the DNA sequence data
3 | # Translate it to protein and print it out in 25-character-long lines
4 |
5 | use strict;
6 | use warnings;
7 | use BeginPerlBioinfo;    # see Chapter 6 about this module
8 |
9 | # Initialize variables
10 | my @file_data = ();
11 | my $dna       = '';
12 | my $protein    = '';
13 |
14 | # Read in the contents of the file "sample.dna"
15 | @file_data = get_file_data("sample.dna");
16 |
17 | # Extract the sequence data from the contents of the file "sample.dna"
18 | $dna = extract_sequence_from_fasta_data(@file_data);
19 |
20 | # Translate the DNA to protein
21 | $protein = dna2peptide($dna);
22 |
23 | # Print the sequence in lines 25 characters long
24 | print_sequence( $protein, 25 );
25 |
26 | exit;
```

下面是例 8.3 的输出：


```
1 | RWRR_GVLGALGRPPTGLQRRRRMG
2 | PAQ_EYAAWEA_LEAEVVVGAFATA
3 | WDAAEWSVQVRGSLAGVVRECAGSG
4 | DMEGDGSDPEPPDAGEDSKSENGEN
5 | APIYCICRKPDIKCFMIGCDNCNEW
6 | FHGDCIRITEKMAKAIREWYCRECR
7 | EKDPKLEIRYRHKKSRERDGNERDS
8 | SEPRDEGGGRKRPVDPDLQRRAGS
9 | GTGVGAMLARGSASPHKSSPQPLVA
10 | TPSQHHQQQQQIKRSARMCGECEA
11 | CRRTEDCGHCDFCRDMKKFGGPNKI
12 | RQKCRLRQCQLRARESYKYFPSSLS
13 | PVTPSESLPRRRRPLPTQQQPQPSQ
14 | KLGRIREDEGAVASSTVKEPPEATA
15 | TPEPLSDEDL
```

8.6 阅读框

生物学家都知道，给定一个 DNA 序列，需要检查这个 DNA 所有的六种阅读框，来寻找细胞用来编码蛋白质的编码区域。

8.6.1 什么是阅读框？

通常情况下你都不知道细胞实际上是从你研究的 DNA 的那个地方开始把它翻译成蛋白质的。只有大约 1-1.5% 的人类 DNA 是基因⁶，也就是用来翻译成蛋白质的 DNA 片段。此外，基因通常都是以片段的形式存在，在转录/翻译的过程中它们会被剪切、拼接在一起。

如果你不知道翻译从何处起始，你就必须要考虑六种可能的阅读框。因为密码子长 3 个碱基，所以翻译可以发生在三个“框”内，比如从第一个碱基、第二个碱基或者第三个碱基开始。（从第四个碱基开始实际上是和从第一个碱基开始完全一样的。）每一种起始位置都会给出一系列不同的密码子，最终就是一系列不同的氨基酸。

此外，转录和翻译可以发生在 DNA 的两条链上，也就是 DNA 序列或者它的反向互补序列都可能包含最终被翻译的 DNA 编码。反向互补序列也可以从三种不同框的任意一种内进行阅读。所以，寻找编码区域、也就是编码蛋白质的 DNA 片段时，总共有六种阅读框需要考虑。

因此这非常常见，通过检查 DNA 序列的所有六种阅读框来寻找最终的蛋白质翻译，找到没有终止密码子的连续的长的氨基酸片段。

终止密码子肯定会打断 DNA→protein 的翻译过程。在翻译过程（实际上是从 RNA 到蛋白质，但是我故意模糊简化了生物化学）中，如果碰到一个终止密码子，翻译就会停止，不断增长的肽链也会停止增长。

不包含终止密码子的长的 DNA 片段叫做开放阅读框（ORFs, open reading frames），是判断正在研究的 DNA 中存在基因的一个重要的依据。所以基因识别程序需要进行这种阅读框的分析，这也正是我们在本章中学习的内容。

8.6.2 翻译阅读框

基于上述事实，我们编写一些代码，来把 DNA 进行六个阅读框的翻译。

在现实生活中，你可能会四处寻找已经写好的、可以完成该任务的子程序。基于任务的基本属性——研究 DNA 的工作者一定会做的事情——你很可能找到这样的代码。但是这里是一个指南，而不是现实世界，所以还是让我们披挂上阵吧。

这个问题并没有听上去那么吓人。所以，检查一下你的子程序行囊，（利用手头可以使用的子程序储备，）想想你现在身处何方、该如何到达目的地。

查看一下我们已经编写的子程序，回想一下 *dna2peptide*。你可能会想到添加一些参数来指定起始和结束位点。现在就让我们动手吧。

还记得吗，尽管我们在第 4 章中计算了反向互补序列，但我们并没有把它编写成子程序。所以我们先从这里开始：

⁶译者注：此段中的基因实际上指的是编码区，或者说是外显子。

```

1  # revcom
2  #
3  # A subroutine to compute the reverse complement of DNA sequence
4
5  sub revcom {
6
7      my($dna) = @_;
8
9      # First reverse the sequence
10     my($revcom) = reverse($dna);
11
12     # Next, complement the sequence, dealing with upper and lower case
13     # A->T, T->A, C->G, G->C
14     $revcom =~ tr/ACGTacgt/TGCAtgca/;
15
16     return $revcom;
17 }

```

现在，这是一点伪代码，展示将要翻译 DNA 特定区间的子程序的思路：

```

1  Given DNA sequence
2
3  subroutine translate_frame ( DNA, start, end )
4
5      return dna2peptide( substr( DNA, start, end - start + 1 ) )
6
7  }

```

这就很好！幸运的是，当把 DNA 传递进已经写好的 *dna2peptide* 子程序中时，Perl 的内置函数 *substr* 使得它可以非常容易的使用指定的起始和终止位点。

注意序列的长度是 $\text{end} - \text{start} + 1$ 。考虑一个简单的例子：如果你从位置 3 起始、到位置 5 终止，你会得到位置 3、4 和 5 的碱基，总共有三个碱基，正好等于 $5 - 3 + 1$ 。

处理类似这样的索引要非常小心，否则代码可能无法正常工作。对于许多程序来说，这是数学最讨人烦的地方。



小心索引！

你需要决定，你是要从 0 开始对位置进行计数，这是 Perl 的处理方式，还是把序列的第一个字符作为位置 1，这是生物学家的处理方式。我们采用生物学家的处理方式。当传递给 Perl 的函数 *substr* 时，位置要减一，当然，最终还是 Perl 的处理方式。

纠正后的伪代码是这样子的：

```

1  Given DNA sequence
2
3  subroutine translate_frame ( DNA, start, end )
4
5      # start and end are numbering the sequence from 1 to length
6

```

```

7 | return dna2peptide( substr( DNA, start - 1, end - start + 1 ) )
8 | }

```

序列的长度并没有随索引的改变而改变，因为：

```

1 | (end - 1) - (start - 1) + 1 = end - start + 1

```

现在我们来编写这个子程序：

```

1 | # translate_frame
2 | #
3 | # A subroutine to translate a frame of DNA
4 |
5 | sub translate_frame {
6 |
7 |     my($seq, $start, $end) = @_;
8 |
9 |     my $protein;
10 |
11 |     # To make the subroutine easier to use, you won't need to specify
12 |     # the end point--it will just go to the end of the sequence
13 |     # by default.
14 |     unless($end) {
15 |         $end = length($seq);
16 |     }
17 |
18 |     # Finally, calculate and return the translation
19 |     return dna2peptide ( substr ( $seq, $start - 1, $end - $start + 1 ) );
20 | }

```

例 8.4从六个阅读框上翻译 DNA。

例 8.4： 从六个阅读框上翻译 DNA 序列

```

1 | #!/usr/bin/perl -w
2 | # Example 8-4   Translate a DNA sequence in all six reading frames
3 |
4 | use strict;
5 | use warnings;
6 | use BeginPerlBioinfo;    # see Chapter 6 about this module
7 |
8 | # Initialize variables
9 | my @file_data = ();
10 | my $dna       = '';
11 | my $revcom    = '';
12 | my $protein   = '';
13 |
14 | # Read in the contents of the file "sample.dna"
15 | @file_data = get_file_data("sample.dna");
16 |

```

```

17 # Extract the sequence data from the contents of the file "sample.dna"
18 $dna = extract_sequence_from_fasta_data(@file_data);
19
20 # Translate the DNA to protein in six reading frames
21 #   and print the protein in lines 70 characters long
22 print "\n -----Reading Frame 1-----\n\n";
23 $protein = translate_frame( $dna, 1 );
24 print_sequence( $protein, 70 );
25
26 print "\n -----Reading Frame 2-----\n\n";
27 $protein = translate_frame( $dna, 2 );
28 print_sequence( $protein, 70 );
29
30 print "\n -----Reading Frame 3-----\n\n";
31 $protein = translate_frame( $dna, 3 );
32 print_sequence( $protein, 70 );
33
34 # Calculate reverse complement
35 $revcom = revcom($dna);
36
37 print "\n -----Reading Frame 4-----\n\n";
38 $protein = translate_frame( $revcom, 1 );
39 print_sequence( $protein, 70 );
40
41 print "\n -----Reading Frame 5-----\n\n";
42 $protein = translate_frame( $revcom, 2 );
43 print_sequence( $protein, 70 );
44
45 print "\n -----Reading Frame 6-----\n\n";
46 $protein = translate_frame( $revcom, 3 );
47 print_sequence( $protein, 70 );
48
49 exit;

```

下面是例 8.4的输出：

```

1  -----Reading Frame 1-----
2
3  RWRR_GVLGALGRPPTGLQRRRRMGPAQ_EYAAWEA_LEAEVVVGAFATAWDAAEWSVQVRGSLAGVVRE
4  CAGSGDMEGDGSDPEPPDAGEDSKSENGENAPIYICIRKPDINCFMIGCDNCNEWFHGDCIRITEKMAKA
5  IREWYCRECREKDPKLEIRYRHKKSRERDGNERDSSEPRDEGGGRKRPVPDPDLQRRAGSGTGVGAMLAR
6  GSASPHKSSPQPLVATPSQHHQQQQQQIKRSARMCGECEACRRTEDCGHCDFCRDMKKFGGPNKIRQKCR
7  LRQCQLRARESYKYFPSSLSPVTPSESLPRPRRPLPTQQQPQPSQKLGRIREDEGAVASSTVKEPPEATA
8  TPEPLSDEDL
9
10 -----Reading Frame 2-----
11
12 DGAEGSWGL_AGHLLVCSGDDAWGLRNRSTLPGRRD_KRK_LWAPLQPPGTPPSGLCRFAGRWRGS_GS
13 APGAEIWRMVQTSLSLQMPGRTASPRMGRMPSTASAAANTSTAS_SGVTTAMSGMSGTASGSLRRWPRP

```

```
14 | SSGTVGSAERKTPS_RFAIGTRSHGSGMAMSGTAVSPGMRVEGARGLSLIQTCSAGQQGQGLGPCLLG
15 | ALLRPTNPLRSPWPHPASITSSSSSRNGQPACVVSVRHVGALRTVVTVISVGT_RSSGAPTRSGRSAG
16 | CASASCGPGNRTSTSLPRSHQ_RPQSPCQGPAGHCPPNSSSHSHRS_GASVKMRGQWRHQQSRSLRLRQP
17 | HLSHSQMRT
18 |
19 | -----Reading Frame 3-----
20 |
21 | MAALRGLGGSRPATYWFAAETTHGACAIGVRCLGGVTRSGSSCGRLCNRLGRRRVVCAGSRVAGGGREGV
22 | RRERRYGGRWFRPRASRCRGQVREWGECALLHLPQTGHQLLHDRV_QLQ_VVPWGLHPDH_EDGQGH
23 | PGVVLSGVQRERQARDSLSAQEVTGAGWQ_AGQQ_APG_GWRAQEACP_SRPAAPGRVRDRGWGHACSG
24 | LCFAPQILSAALGGHTQPASPAADQTVSPHVW_V_GMSAH_GLWSL_FLSGHEEVRGPQQDPAEVPA
25 | APVPAAGPGIVQVLPFLALTSDALRVPKAPPATAHPTAATAITEVRAHP_R_GGSGVINSQGAS_GYSH
26 | T_ATLR_GP
27 |
28 | -----Reading Frame 4-----
29 |
30 | _VLI_EWLRCGCSLRRLLDC_ _RHCPLIFTDAP_LL_WLWLLGGQWPAGPWQGL_GRHW_ERGREVLVR
31 | FPGPQLALAQPALLPDLVGAPELLHVPTEITVTTVLSAPTCLTLTTHAG_PFDLLLLLLVMLAGCGHQGL
32 | RRGFVGRSRAPSKHGPNPCP_PCPALQVWIRDRPLAPSTLIPGLTAVPLIAIPLP_LLVPPIANL_LGVFL
33 | SALPTVPLPDGLGHLLSDPDVPMELIAVVTPDHEAVDVRFAADAVIDGRILPILGLAVLPGIWRLWV_T
34 | ISLHISAPGALPHDPRQRPANLHRPLGGVPGGCKGAHNYFRF_SRLPGSVLLLRPHASSPLQTSRWPA_
35 | SPQDPSAPPS
36 |
37 | -----Reading Frame 5-----
38 |
39 | RSSSESGSGVAVASGGS�TVDDATAPSSSRMRPNFCDGCGCCWVGSGRRGLGRDSEGVTGESEEGKYLYD
40 | SRARSWHWRSRHFCRILLGPPNFFMSRQKSQ_PQSSVRRHASHSPHMRADRLICCCCW_CWLGVATKGC
41 | GEDLWGEAEPRASMAPTPVPDPARRCRSGSGTGLLRPPPSRGSLLSRSLPSRSRDFLCR_RISSLGFSF
42 | LHSRQYHSRMALAIIFSVIRMQSPWNHSLQLSHPIKQLMSGRLRQM_MGAFSPFSDLLSSPASGGSGSEP
43 | SPSISPLPAHSLTTPASDPRTCTDHAASQAVAKAPTTTSASSHASQAAYSYCAGPMRRLRCKPVGGRPR
44 | APKTPQRRH
45 |
46 | -----Reading Frame 6-----
47 |
48 | GPHLRVAQVWL_PQEAP_LLMTPLPPHLHGCALTSMVAVAAGWAVAGGALAGTLRASLVRARKGSTCTI
49 | PGPAAGTGAAGTSAGSCWGPRTSSCPDRNHSDHSPQCADMPHTHTCGLTV_SAAAAAGDAGWVPPRAA
50 | ERICGAKQSPEQAWPQPLSLTLPGAAGLDQGGQASCALHHPGAHCCPAHCHPAPVTSCADSESLAWGLSL
51 | CTPDSTTPGWPWPSSQ_SGCSPHGTTHCSCHTRS_SS_CPVCGRCSRWAHSPHSRTCCPPRHLEALGLNH
52 | LPPYLRSRRTPSRPPPATREPAQTTRRRPRRLQRRPQLLPLLVTPPRQRTPIAQAPCVVSAANQ_VAGLE
53 | PPRPLSAAI
```

8.7 练习题

习题 8.1

编写一个子程序，检查一个字符串，如果它是一个 DNA 序列就返回 `true`。编写另外一个子程序来检查蛋白质序列数据。

习题 8.2

编写一个程序，在未排序的数组中通过基因名查找一个基因。

习题 8.3

编写一个程序，在一个排序的数组中通过基因名查找一个基因。使用 Perl 的 `sort` 函数来对数组进行排序。额外的得分：编写一个折半查找的子程序进行搜索。

习题 8.4

编写一个子程序，把一个元素插入到一个排序的数组中。提示：就像在第 4 章中演示的那样，使用 Perl 的 `splice` 函数插入元素。

习题 8.5

编写一个程序，在散列中通过基因名查找一个基因。从你自己的工作中获取基因，或者尝试从 www.ncbi.nlm.nih.gov 或附录 A 中的一个网站下载一个物种的全部基因列表。为所有的基因制作一个散列（键是基因名，值是基因 ID 或者基因序列）。提示：你可能需要编写一个简短的 Perl 程序，来重新格式化你手头的基因列表，这样就容易把它做成 Perl 的散列了。

习题 8.6

编写一个子程序，检查数组中的数据，如果是 FASTA 格式就返回 `true`。注意 FASTA 格式使用标准的 IUB/IUPAC 氨基酸和核苷酸代码，以及代表未知长度的空位的破折号 (-)。此外，对于氨基酸来说，星号 (*) 代表终止密码子。在正则表达式中使用星号时一定要小心，使用 `*` 对它进行转义，来匹配真正的星号。

剩下的问题就是，DNA 的突变对它们编码的蛋白质的影响。这要把第 7 章中随机化和突变的主题与本章中遗传密码的主题结合起来。

习题 8.7

对于每一个密码子，看看密码子上单核苷酸的突变会产生什么影响：是编码同样的氨基酸，还是新的密码子会编码一个完全不同的氨基酸？新编码的是那种氨基酸？编写一个子程序，给定一个密码子，返回密码子上任意一个突变导致的编码出的新的所有的氨基酸列表。

习题 8.8

编写一个子程序，给定一个氨基酸，随机把它改变成习题 8.7 计算出来的氨基酸中的一种。

习题 8.9

编写一个程序，随机突变蛋白质中的氨基酸，但是要限定在原始密码子的单核苷酸突变能导致的突变范围内，就像习题 8.7 和 8.8 中的那样。

习题 8.10

有些密码子比其他的密码子更容易出现在随机 DNA 中。比如，在 64 种可能的密码子中，有 6 种编码丝氨酸，但只有 2 中编码苯丙氨酸。编写一个子程序，给定一个氨基酸，返回它被随机生成的密码子编码的可能性（参看第 7 章）。

习题 8.11

编写一个子程序，以一个氨基酸、位置 1、2 或 3 和一个核苷酸作为它的参数。然后，对于编码这个特定氨基酸的每一个密码子（可能有一到六个不等的密码子），都在指定的位置上突变称指定的核苷酸。最后，返回突变后的密码子编码的氨基酸集。

习题 8.12

编写一个程序，给定两个氨基酸，返回它们潜在（而非指定）密码子上单核苷酸突变从而导致一个氨基酸的密码子突变成另一个氨基酸的密码子的概率。

第 9 章 限制酶图谱和正则表达式

目录

9.1 正则表达式	194
9.2 限制酶切图谱和限制性内切酶	196
9.3 Perl 的操作	209
9.4 练习题	210

在本章中，我会对 Perl 的正则表达式和操作符进行概述，这是我们一直在使用的 Perl 语言的两个基本特性。我们也会探讨一个标准的、基础的分子生物学技术的编程：找到一个序列的限制酶图谱。对 DNA 进行限制酶切消化是把它“指纹化”的最原始的方法之一，现在可以在计算机上对其进行模拟了。

在实验室中，限制酶图谱和与之相关的限制酶切消化都是非常常见的计算，并且多种软件包都可以进行这样的计算。它们是意欲进行克隆实验的基本工具，比如，应用它，可以把一个期望的 DNA 片段插入到克隆载体中。限制酶图谱在序列测序项目中也有其应用，比如鸟枪法或直接测序。

9.1 正则表达式

现在我们已经使用了一段时间的正则表达式了。本小节将补充一些背景知识，并把本书前面部分中对正则表达式进行的那些零散的讨论串联起来。

正则表达式有趣、主要，而且应用潜能无限。Jeffrey Friedl 的精通正则表达式 (O'Reilly 出版) 这本书就对它们进行了全面深入的讲解。Perl 尤其擅长使用正则表达式，Perl 的文档中对此进行了很好的讲解。当处理序列或者 GenBank、PDB 和 BLAST 文件等生物学数据时，正则表达式非常有用。

正则表达式是用一个字符串来表征并搜索多个字符串的方法。尽管严格来说它们并不是一样的东西，但是你可以把正则表达式看做是一种高度发展的通配符集合，这会对你有所帮助。正则表达式中的特殊字符通常叫做元字符。

许多人对通配符都非常熟悉，在搜索引擎或者纸牌游戏中都可以发现它的身影。举个例子，你可能会发现，通过键入 `biolog*` 可以指代所有以 `biolog` 起始的单词。或者，你可能发现自己的牌中有五张尖儿。（不同的情况下可能会使用不同的通配符。Perl 的正则表达式中使用 `*` 表示“前面的字符重复 0 次或多次”，而不是刚才那个通配符例子中的“后面跟任何字符”。）

在计算机科学中，不管是在实践中还是在理论上，这种类型的通配符或者元字符都有一个重要的历史。被知名的逻辑学家发明之后，在实践中，星号字符就被叫做克莱尼闭包¹了。作为对理论的致敬，我会提到有一个简单的计算机模型，它没有图灵机那么强大，但完全可以处理用正则表达式描述的同样类型的语言。这种机器模型叫做有限自动机。现在理论知识已经够多了。

我们已经看到了好多正则表达式的例子，在 DNA 或者蛋白质序列中进行查找。此处，我将对正则表达式背后的一些基础观念进行简要的介绍，作为多一些术语的引言。在附录 B 中有一个非常有用的正则表达式特性的总结。最后，我们将看看如何在 Perl 的文档中对它们进行更加深入的学习。

所以让我们从一个实际的例子开始吧，对于逐章阅读至今的读者来说这应该已经非常熟悉了：使用字符集来查找 DNA。假设有一个小的基序，你想在你的 DNA 库中进行查找，这个基序有六个碱基对之长：CT 后面跟着 C 或 G 或 T，之后是 ACG。这个基序中的第三个核苷酸不是 A，但可以是 C、G 或者 T。你可以使用字符集 `[CGT]` 制作一个正则表达式来表示这个变异的位点。这样，基序就可以用这样一个正则表达式来进行表征了：`CT[CGT]ACG`。这是一个有六个碱基对之长的基序，在它的第三个位置是一个 C、G 或者 T。如果你的 DNA 保存在标量变量 `$dna` 中，你就可以检测一下其中是否存在这个基序，在模式匹配语句的条件测试中使用正则表达式即可，就像这样：

```
1 if( $dna =~ /CT[CGT]ACG/ ) {  
2   print "I found the motif!!\n";  
3 }
```

正则表达式基于三个基础的理念：

重复（或闭包）

星号（`*`），也叫做克莱尼闭包或克莱尼星号，表示前面的字符重复 0 次或多次。比如，`abc*` 匹配所有这些字符串：`ab`，`abc`，`abcc`，`abccc`，`abcccc` 等

¹译者注：亦称克莱尼星号。

等。这个正则表达式匹配无穷多个字符串。

择一

在 Perl 中，模式 `(a|b)`（读作：a or b）匹配字符串 a 或者字符串 b。

连接

这是显而易见的一点。在 Perl 中，字符串 `ab` 表示字符 a 后面跟着（与之相连接）字符 b。

使用小括号进行分组是非常重要的：它们也是元字符。所以，举个例子，字符串 `(abc|def)z*x` 匹配 `abcx`、`abczx`、`abczzx`、`defx`、`defzx`、`defzzzzzx` 等等这样的字符串。用白话来说，它匹配的是 `abc` 或者 `def` 后面跟着零个或多个 `z`，并且最后是一个 `x`。这个例子把分组、择一、闭包和连接的理念整合在了一起。通过把这三个基础的理念整合起来，就可以真正看出正则表达式的强大之处来了。

Perl 有许多正则表达式的特性。它们基本上都是我们刚刚提到的三个基础理念——重复、择一和连接的快捷键而已。比如，前面演示的字符集使用择一可以写成 `(C|G|T)`。另外一个常见的特性是点号，它可以用来表示除换行符以外的任意字符。所以 `ACG.*GCA` 表示任意起始于 `ACG`、终止于 `GCA` 的 DNA。用白话来说，它读作：ACG 后面跟着零个或多个字符，然后是 `GCA`。

在 Perl 中，正则表达式通常被用作模式匹配界定符的正斜线包裹在中间。查看 `m/` 的文档（或者是附录 B），其中包括影响正则表达式行为的一些选项。就像你将看到的那样，正则表达式也被用在许多 Perl 的内置命令中。

Perl 文档是最基本的：从 <http://www.perldoc.com/perl5.6/pod/perlre.html#top> 上的 Perl 手册中的 *perlre* 小节开始吧。

9.2 限制酶切图谱和限制性内切酶

分子生物学的重大发现之一就是限制性内切酶的发现，它为生物学研究的当今黄金时代铺平了道路。为了非生物学家，同时也为后面准备好编程材料，这里先对其进行一个概述。

9.2.1 背景

限制性内切酶是把 DNA 切割成短的、特定序列的蛋白质，比如，最常见的限制性内切酶 *EcoRI* 和 *HindIII* 就在实验室中被广泛应用。*EcoRI* 切割找到的 GAATTC，切割位点在 G 和 A 之间。实际上，它同时切割互补的两条链，在每一端都留下一个突出部分。这些在单链上有几个碱基的“粘性末端”，使得片段有可能被重排，比如在克隆和测序中它使得把 DNA 插入到载体中成为可能。*HindIII* 切割 AAGCTT，切割位点在两个 A 之间。有些限制性内切酶会在中间进行切割，从而产生没有突出的“平滑末端”。已知大约有 1,000 种限制性内切酶。

如果你查看限制性内切酶 *EcoRI* 的反向互补链，你会看到同样的序列 GAATTC。这是一种生物学的回文，也就是说正反读都一样。许多限制性位点都是回文结构。

在实验室中，计算限制酶图谱是一个常见且实用的生物信息学工作。通过计算限制酶图谱，可以对实验进行规划，找到切割 DNA 并插入基因的最佳方法，进行特定位点的突变，或者是其他重组 DNA 技术的各种应用。通过初步的计算，实验室科学家就可以不用在实验台上不断得进行试错性的实验了。在 <http://rebase.neb.com/rebase/rebase.html> 上可以查阅更多关于限制性内切酶的相关内容。

现在我们要编写一个在实验室中非常有用的程序：它会在 DNA 序列中查找限制性内切酶，并报告限制酶切图谱以及这个限制性内切酶出现在 DNA 上的精确位点。

9.2.2 程序规划

在前面的第 5 章中，你已经看到如何在文本中查找正则表达式了。所以你也知道如何使用 Perl 来查找序列中的基序。现在让我们想想，该如何利用这些技术来生成限制酶切图谱。下面是需要询问的一些问题：

我在哪里可以找到限制性内切酶的数据？

限制性内切酶的数据可以在限制性酶切酶数据库（REBASE, Restriction Enzyme Database）中找到，它的网址是 <http://rebase.neb.com/rebase/rebase.html>。

我该如何使用正则表达式表征限制性内切酶？

浏览那个网站，你会看到限制性内切酶用它们自己的语言进行了表征。我们会尝试把这种语言翻译成正则表达式的语言。

我该如何存储限制性内切酶的数据？

大约有 1,000 种有名字和定义的限制性内切酶。这使得可以使用散列提供的快速的键-值查找对它们进行处理。当你编写一个真正的应用程序时，比如应用于网站，最好是创建一个 DBM 文件来存储这些信息，当程序需要进行查找时立马就可以使用它。我会在第 10 章中对 DBM 文件进行介绍。此处，我们仅仅是指出这种方式而已。我们将只在程序中保存几个限制性内切酶的定义。

我该如何接受用户的查询？

你可以询问一个限制性内切酶的名字，或者可以让用户直接键入一个正则表达式。我们将采用第一种方法。此外，你想要让用户指定使用的序列。再重复一次，为了简化问题，你将只从一个 DNA 样本文件中读入数据。

我该如何向用户报告返回限制酶切图谱呢？

这是一个非常重要的问题。最简单的方式就是生成一个位置以及在此处找到的限制性内切酶的名字的列表。因为它呈现的信息非常简单，所以对于进一步的处理非常有用。

但是如果你不想进行进一步的处理，只想把限制酶切图谱展示给用户呢？这时，进行一个图形化展示就会更加有用，可能是打印出序列，并在其上用一条线来标明限制性内切酶存在的位置。

你可以使用各种精致花哨的东西，但我们现在只采用简单的方式来进行处理，输出一个列表。

所以，我们的规划就是编写一个程序，包括把限制性内切酶的数据翻译成正则表达式，并把它存储为以限制性内切酶名字作为键的值中。DNA 序列数据从文件中读入，用户将被提示输入限制性内切酶的名字。相应的正则表达式会从散列中提取出来，我们将会查找正则表达式所有的出现，以及出现的位置。最后，返回找到的位置列表。

9.2.3 限制性内切酶数据

访问 REBASE 网站时你会看到，有多种格式的限制性内切酶数据可供使用。在查看之后，你决定使用存储在 *bionet* 文件中的信息，它有一个相当简单的排版。下面是这个文件的头信息以及一些限制性内切酶的信息：

```

1 REBASE version 104                                     bionet.104
2
3 =====
4 REBASE, The Restriction Enzyme Database   http://rebase.neb.com
5 Copyright (c)  Dr. Richard J. Roberts, 2001.   All rights reserved.
6 =====
7
8 Rich Roberts                                           Mar 30 2001
9
10 AaaI (XmaIII)                C^GGCCG
11 AacI (BamHI)                 GGATCC
12 AaeI (BamHI)                 GGATCC
13 AagI (ClaI)                  AT^CGAT
14 AaqI (ApaLI)                 GTGCAC
15 AarI                         CACCTGCNNNN^
16 AarI                         ^NNNNNNNNNGCAGGTG
17 AatI (StuI)                  AGG^CCT
18 AatII                        GACGT^C
19 AauI (Bsp1407I)              T^GTACA
20 AbaI (BclI)                  T^GATCA
21 AbeI (BbvCI)                 CC^TCAGC
22 AbeI (BbvCI)                 GC^TGAGG

```

23	AbrI (XhoI)	C^TCGAG
24	AcaI (AsuII)	TTCGAA
25	AcaII (BamHI)	GGATCC
26	AcaIII (MstI)	TGCGCA
27	AcaIV (HaeIII)	GGCC
28	AccI	GT^MKAC
29	AccII (FnuDII)	CG^CG
30	AccIII (BspMII)	T^CCGGA
31	Acc16I (MstI)	TGC^GCA
32	Acc36I (BspMI)	ACCTGCNNNN^
33	Acc36I (BspMI)	^NNNNNNNNNGCAGGT
34	Acc38I (EcoRII)	CCWGG
35	Acc65I (KpnI)	G^GTACC
36	Acc113I (ScaI)	AGT^ACT
37	AccB1I (HgiCI)	G^GYRCC
38	AccB2I (HaeII)	RGCGC^Y
39	AccB7I (PflMI)	CCANNNN^NTGG
40	AccBSI (BsrBI)	CCG^CTC
41	AccBSI (BsrBI)	GAG^CGG
42	AccEBI (BamHI)	G^GATCC
43	AceI (TseI)	G^CWGC
44	AceII (NheI)	GCTAG^C
45	AceIII	CAGCTCNNNNNNN^
46	AceIII	^NNNNNNNNNNNGAGCTG
47	AciI	C^CGC
48	AciI	G^CGG
49	AclI	AA^CGTT
50	AclNI (SpeI)	A^CTAGT
51	AclWI (BinI)	GGATCNNNN^

你的第一个任务就是读入这个文件，获取每一个酶的名字和识别位点（或说是限制性内切位点）。现在为了简化问题，简单的把括号中的限制性内切酶的名字丢掉。

该如何读入这个数据呢？

```

1 Discard header lines
2
3 For each data line:
4
5     remove parenthesized names, for simplicity's sake
6
7     get and store the name and the recognition site
8
9     Translate the recognition sites to regular expressions
10    --but keep the recognition site, for printing out results
11 }
12
13 return the names, recognition sites, and the regular expressions

```

这是一个高等级的粗犷的伪代码，所以让我们来把它细化扩展开。（注意大括号并没

有配对。这样是完全可以的，因为对于伪代码来说没有什么语法规则，只要它能使用就行了！）下面是丢弃头信息行的伪代码：

```
1 foreach line
2
3     if /Rich Roberts/
4
5         break out of the foreach loop
6
7 }
```

这是基于文件的格式，你寻找的字符串是数据行开始之前的最后一行文本。（当然，如果文件的格式发生了变化，它可能就无法再使用了。）

现在，我们来进一步展开伪代码，想想如何来完成下面的任务：

```
1 # Discard header lines
2 # This keeps reading lines, up to a line containing "Rich Roberts"
3 foreach line
4     if /Rich Roberts/
5         break out of the foreach loop
6 }
7
8 For each data line:
9
10     # Split the two or three (if there's a parenthesized name) fields
11     @fields = split( " ", $_);
12
13     # Get and store the name and the recognition site
14     $name = shift @fields;
15
16     $site = pop @fields;
17
18     # Translate the recognition sites to regular expressions
19     --but keep the recognition site, for printing out results
20 }
21
22 return the names, recognition sites, and the regular expressions
```

这并不是翻译，但让我们来看看你都做了哪些事情。

首先，你想从一个字符串中提取出名字和识别位点的数据。在 Perl 中把一行分割成单词的最常用的方法，就是使用 Perl 的内置函数 *split*，尤其是当字符串有着很规整的格式时。

对于有空白的一行，如果其中有两三个单词被空白分隔开来，你可以把它们提取保存到数组中，使用下面这个简单的 *split* 调用即可（它处理的是存储在特殊变量 `$_` 中的行）：

```
1 @fields = split( " ", $_);
```

取决于是否有用括号括起来的酶的别名，@fields 数组可能有两个或三个元素。但你总可以这样提取出第一个和最后一个元素：

```
1 | $name = shift@fields;
2 | $site = pop@fields;
```

现在，你面临的问题是要把识别位点翻译成正则表达式。

仔细查看这些识别位点，并阅读网站上找到的 REBASE 的文档后，你知道切割位点是用脱字符 (^) 来表示的。这对于在序列中查找位点用的正则表达式的制作并没有什么帮助，所以你应该把它删除掉（参看第 9.4 节中的 Exercise 9.6）。

还要注意，识别位点中的碱基并不仅仅只是 A、C、G 和 T 四种碱基，它们还使用到了表 4.1 中的更多的扩展字符。这些额外的字母包括代表两个、三个或四个碱基的所有可能组合的字母。在这方面，它们真的很像字符集的简写。奥！我们来编写一个子程序，把这些代码都替换成字符集，这样我们就有了我们要的正则表达式。

当然，REBASE 使用它们，是因为一个限制性内切酶可能匹配几个不同的识别位点。

例 9.1 是一个子程序，对于给定的一个字符串，它会把这样的代码转换成字符集。

例 9.1：把 IUB 的模糊代码翻译成正则表达式

```
1 | # Example 9-1 Translate IUB ambiguity codes to regular expressions
2 | # IUB_to_regexp
3 | #
4 | # A subroutine that, given a sequence with IUB ambiguity codes,
5 | # outputs a translation with IUB codes changed to regular expressions
6 | #
7 | # These are the IUB ambiguity codes
8 | # (Eur. J. Biochem. 150: 1-5, 1985):
9 | # R = G or A
10 | # Y = C or T
11 | # M = A or C
12 | # K = G or T
13 | # S = G or C
14 | # W = A or T
15 | # B = not A (C or G or T)
16 | # D = not C (A or G or T)
17 | # H = not G (A or C or T)
18 | # V = not T (A or C or G)
19 | # N = A or C or G or T
20 |
21 | sub IUB_to_regexp {
22 |
23 |     my ($iub) = @_ ;
24 |
25 |     my $regular_expression = '';
26 |
27 |     my %iub2character_class = (
28 |
29 |         A => 'A',
```



```

30     C => 'C',
31     G => 'G',
32     T => 'T',
33     R => '[GA]',
34     Y => '[CT]',
35     M => '[AC]',
36     K => '[GT]',
37     S => '[GC]',
38     W => '[AT]',
39     B => '[CGT]',
40     D => '[AGT]',
41     H => '[ACT]',
42     V => '[ACG]',
43     N => '[ACGT]',
44 );
45
46 # Remove the ^ signs from the recognition sites
47 $iub =~ s/\^//g;
48
49 # Translate each character in the iub sequence
50 for ( my $i = 0 ; $i < length($iub) ; ++$i ) {
51     $regular_expression .= $iub2character_class{ substr( $iub, $i, 1 ) };
52 }
53
54 return $regular_expression;
55 }

```

看起来你已经准备好编写一个子程序，从 REBASE 的数据文件中提取数据。但还有一个重要的问题没有解决：你想要返回的数据有多精确？

对于原始的 REBASE 文件的每一行，你打算返回三个数据项：酶的名字、识别位点和正则表达式。这并不能很容易的转换成散列。你可以返回一个数组，把这三个数据项连续的存储在一起。这是可行的：要读入数据，你必须从数组中读取成组的三个项目。这是完全可以的，但使得查询有些困难。随着你学习更多 Perl 的进阶知识，你会发现你可以创建自己的复杂数据结构。

既然你已经学习了 *split*，也许你可以使用一个散列，它的键是酶的名字，它的值是包含用空白分隔开的识别位点和正则表达式的字符串。然后你就可以对数据进行快速的查找，并使用 *split* 提取出需要的值。例 9.2展示了这种方法。

例 9.2：解析 REBASE 数据文件的子程序

```

1 # Example 9-2 Subroutine to parse a REBASE datafile
2 # parseREBASE-Parse REBASE bionet file
3 #
4 # A subroutine to return a hash where
5 #   key   = restriction enzyme name
6 #   value = whitespace-separated recognition site and regular expression
7
8 sub parseREBASE {
9

```

```
10     my ($rebasefile) = @_;  
11  
12     use strict;  
13     use warnings;  
14     use BeginPerlBioinfo;    # see Chapter 6 about this module  
15  
16     # Declare variables  
17     my @rebasefile = ();  
18     my %rebase_hash = ();  
19     my $name;  
20     my $site;  
21     my $regex;  
22  
23     # Read in the REBASE file  
24     my $rebase_filehandle = open_file($rebasefile);  
25  
26     while (<$rebase_filehandle>) {  
27  
28         # Discard header lines  
29         ( 1 .. /Rich Roberts/ ) and next;  
30  
31         # Discard blank lines  
32         /^\\s*$/ and next;  
33  
34         # Split the two (or three if includes parenthesized name) fields  
35         my @fields = split( " ", $_ );  
36  
37         # Get and store the name and the recognition site  
38  
39         # Remove parenthesized names, for simplicity's sake,  
40         # by not saving the middle field, if any,  
41         # just the first and last  
42         $name = shift @fields;  
43  
44         $site = pop @fields;  
45  
46         # Translate the recognition sites to regular expressions  
47         $regex = IUB_to_regex($site);  
48  
49         # Store the data into the hash  
50         $rebase_hash{$name} = "$site $regex";  
51     }  
52  
53     # Return the hash containing the reformatted REBASE data  
54     return %rebase_hash;  
55 }
```

这个 *parseREBASE* 子程序做了大量的工作。对于一个子程序来说，做的事情是不是太多了，要不要重新编写它？当你编写代码时，这是你应该问自己的一个很好的问题。在这

个例子中，就先这样吧。然而，除了做了大量的事情以外，它也采用了一些新的处理方法，现在我们就来看看吧。

9.2.4 逻辑操作符和范围操作符

你使用 *foreach* 循环来处理存储在 `@rebasefile` 数组中的 *bionet* 文件的每一行。

在这个循环中，你使用了一个 Perl 的新特性来跳过头信息行，这就是范围操作符 (`..`)，它出现在这一行中：

```
1 | ( 1 .. /Rich Roberts/ ) and next;
```

它的作用是跳过从第一行到包含“Rich Roberts”这一行的所有行，换句话说，就是头信息行。（范围操作符必须至少有一个可以作为数字的终点，就像这样，它才能正常工作。）

and 函数是一个逻辑操作符。在绝大多数编程语言中都有逻辑操作符。在 Perl，它们变得非常流行，所以尽管我们在本书中并没有大量使用它们，但你将来会经常遇到使用它们的代码。事实上，随着本书的深入，你将开始越来越多的看到它们。

逻辑操作符可以用来测试两个条件是不是都为 `0`，例如：

```
1 | if( $string eq 'kinase' and $num == 3 ) {
2 |     ...
3 | }
```

只有当两个条件都为 `0` 时，整个语句才为 `0`。

类似的，通过逻辑操作符，使用 *or* 操作符，你可以测试是不是至少有一个条件为 `0`，例如：

```
1 | if( $string eq 'kinase' or $num == 3 ) {
2 |     ...
3 | }
```

这里，如果两个条件至少有一个为 `0`，那么 *if* 语句就为 `0`。

此外，还有一个 *not* 逻辑操作符，一个否定操作符，使用它你可以测试某个东西是不是为 `0`：

```
1 | if( not 6 == 9 ) {
2 |     ...
3 | }
```

`6 == 9` 返回 `0`，但它被 *not* 操作符否定了，所以真个条件测试返回 `0`。

此外还有非常相关的操作符，与 *and* 相关的是 `&&`，与 *or* 相关的是 `||`，与 *not* 相关的是 `!`。它们在行为上有少许的不同（实际上，优先级不同）。大多数 Perl 代码都使用我演示的版本，但两者都很常见。

当你对优先级有所怀疑时，你总是可以把表达式用小括号包裹起来，确保你的语句的执行结果就是你所期望的那样。（参看本章后面的第 9.3.1 小节。）

逻辑操作符也有一定的求值顺序，这使得它们在控制程序流程方面非常有用。让我们看一下 *and* 操作符是如何对它的两个参数进行求值的。它首先对左边的参数进行求值，当且仅当它为 `1` 时，才会对右边的参数求值并返回结果。如果左边参数的求值结果为 `0`，右边的参数就永远不会被求值了。所以 *and* 操作符的行为就像一个小的 *if* 语句。举个例子，下面这两个语句就是完全等同的：

```
1 | if( $verbose ) {
2 |     print $helpful_but_verbose_message;
3 | }
4 |
5 | $verbose and print $helpful_but_verbose_message;
```

当然，*if* 语句要更加灵活一些，因为它允许你轻松地代码块中添加更多的语句，对于 *elsif* 和 *else* 条件和它们的代码块也是一样。但对于简单的情况，*and* 操作符会更好一些。²

逻辑操作符 *or* 会对左边的参数求值，如果它为 `1` 就会返回求值结果；如果左边的参数求值结果不为 `1`，*or* 操作符就会对右边的参数求值并返回结果。所以还有另一种方法来编写单行的语句，在 Perl 程序中你会经常看到：

```
1 | open(MYFILE, $file) or die "I cannot open file $file: $!";
```

这和我们常用的基本上是等同的：

```
1 | unless(open(MYFILE, $file)) {
2 |     print "I cannot open file $file\n";
3 |     exit;
4 | }
```

言归正传，来看一下 *parseREBASE* 子程序中的这一行：

```
1 | ( 1 .. /Rich Roberts/ ) and next;
```

左边的参数是 `1 .. /Rich Roberts/` 这个范围。当你在这些行的范围之内时，范围操作符返回 `1` 值。因为它为 `1`，所以 *and* 逻辑操作符就会继续看看另一边的值是不是也为 `true`，它会找到 *next* 函数，它的求值为 `true`，而它则会把你带到包裹在 *foreach* 循环中的“下一个”迭代。所以当你第一行和 *Rich Roberts* 这一行之间时，会直接跳过循环的剩余部分。

类似的，这一行：

```
1 | /^\\s*$/ and next;
```

也会把你带回到 *foreach* 的下一个迭代，当左边的参数为 `1`、也就是匹配一个空行时。

在设计阶段，我们已经讨论过 *parseREBASE* 子程序的其他部分了。

²你甚至可以把逻辑操作符一个接一个地串成长串，来构建更加复杂的表达式，并用括号对它们进行分组。个人而言，我不太喜欢这种风格，但在 Perl 中，方法不止一种！

9.2.5 寻找限制性内切位点

所以现在是时候来编写主程序了，看看我们的代码的实际表现。让我们从一个小的伪代码开始，看看我们还需要做什么：

```

1 | #
2 | # Get DNA
3 | #
4 | get_file_data
5 |
6 | extract_sequence_from_fasta_data
7 |
8 | #
9 | # Get the REBASE data into a hash, from file "bionet"
10 | #
11 | parseREBASE('bionet');
12 |
13 | for each user query
14 |
15 |     If query is defined in the hash
16 |         Get positions of query in DNA
17 |
18 |     Report on positions, if any
19 |
20 | }
```

你现在需要编写一个子程序，在 DNA 中寻找查询的位置。记住例 5.7 在 `foreach` 循环中进行全局查找的技巧，并且牢记在心。说到做到：

```

1 | Given arguments $query and $dna
2 |
3 | while ( $dna =~ /$query/ig ) {
4 |     save the position of the match
5 | }
6 |
7 | return @positions
```

在你之前使用这个技巧的时候，你仅仅计数了有多少匹配，而没有处理匹配的位置。我们来看看文档找点线索，尤其是文档中的内置函数列表。看起来，`pos` 函数可以解决这个问题。它给出了 `m//g` 查找中最后一次变量匹配的位置。例 9.3 演示了主程序，其后是需要的子程序。这是一个简单的子程序，因为像 `pos` 这样的 Perl 函数使得问题简单了许多。

例 9.3：为用户的查询制作限制酶切图谱

```

1 | #!/usr/bin/perl -w
2 | # Example 9-3   Make restriction map from user queries on names of restriction enzymes
3 |
4 | use strict;
5 | use warnings;
```

```
6 use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8 # Declare and initialize variables
9 my %rebase_hash          = ();
10 my @file_data            = ();
11 my $query                = '';
12 my $dna                  = '';
13 my $recognition_site     = '';
14 my $regex                = '';
15 my @locations            = ();
16
17 # Read in the file "sample.dna"
18 @file_data = get_file_data("sample.dna");
19
20 # Extract the DNA sequence data from the contents of the file "sample.dna"
21 $dna = extract_sequence_from_fasta_data(@file_data);
22
23 # Get the REBASE data into a hash, from file "bionet"
24 %rebase_hash = parseREBASE('bionet');
25
26 # Prompt user for restriction enzyme names, create restriction map
27 do {
28     print "Search for what restriction site for (or quit)?: ";
29
30     $query = <STDIN>;
31
32     chomp $query;
33
34     # Exit if empty query
35     if ( $query =~ /^\\s*$/ ) {
36
37         exit;
38     }
39
40     # Perform the search in the DNA sequence
41     if ( exists $rebase_hash{$query} ) {
42
43         ( $recognition_site, $regex ) = split( " ", $rebase_hash{$query} );
44
45         # Create the restriction map
46         @locations = match_positions( $regex, $dna );
47
48         # Report the restriction map to the user
49         if (@locations) {
50             print "Searching for $query $recognition_site $regex\\n";
51             print "A restriction site for $query at locations:\\n";
52             print join( " ", @locations ), "\\n";
53         }
54         else {
```

```

55         print "A restriction enzyme $query is not in the DNA:\n";
56     }
57 }
58 print "\n";
59 } until ( $query =~ /quit/ );
60
61 exit;
62
63 #####
64 #
65 # Subroutine
66 #
67 # Find locations of a match of a regular expression in a string
68 #
69 #
70 # return an array of positions where the regular expression
71 # appears in the string
72 #
73
74 sub match_positions {
75
76     my ( $regexp, $sequence ) = @_ ;
77
78     use strict;
79
80     use BeginPerlBioinfo;    # see Chapter 6 about this module
81
82     #
83     # Declare variables
84     #
85
86     my @positions = ();
87
88     #
89     # Determine positions of regular expression matches
90     #
91
92     while ( $sequence =~ /$regexp/ig ) {
93
94         push( @positions, pos($sequence) - length($&) + 1 );
95     }
96
97     return @positions;
98 }

```

下面是例 9.3的一些示例输出:

```

1 | Search for what restriction enzyme (or quit)?: AceI
2 | Searching for AceI G^CWGC GC[AT]GC

```

```
3 | A restriction site for AceI at locations:
4 | 54 94 582 660 696 702 840 855 957
5 |
6 | Search for what restriction enzyme (or quit?): AccII
7 | Searching for AccII CG^CG CGCG
8 | A restriction site for AccII at locations:
9 | 181
10 |
11 | Search for what restriction enzyme (or quit?): AaeI
12 | A restriction site for AaeI is not in the DNA:
13 |
14 | Search for what restriction enzyme (or quit?): quit
```

注意子程序 *match_positions* 中的 `length($&)`。\$& 是在正则表达式匹配成功后设置的一个特殊变量。它表示匹配正则表达式的序列。因为 *pos* 给出的是匹配序列后面的第一个碱基的位置，所以你必须减去匹配序列的长度并加一（让碱基开始于位置 1 而非位置 0），这样才是匹配开始的位置。其他的特殊变量包括包含字符串中成功匹配前面的所有内容的 `$``，和包含字符串中成功匹配后面的所有内容的 `$'`。所以，举个例子：`'123456' =~ /34/` 匹配成功，并把这些特殊变量设置为：`$` = '12'`，`$& = '34'`，以及 `$' = '56'`。

不可否认我们此处完成的仅仅是个赤裸的骨架而已，但它确实可以工作。参看本章末尾的练习题，其中给出了扩展这个代码的方法。

9.3 Perl 的操作

在这本指南性的编程书籍中，我们没有讨论基本的算术运算，就已经做的很好的，因为实际上你并不需要比增加计数器的加法更多的内容。

然而，包括 Perl 在内的编程语言的一个重要部分，就是进行数学计算的能力。参看附录 B，它展示了 Perl 中的基本运算。

9.3.1 运算和括号的优先级

运算有一套优先级规则。这使得语言可以在一行中有多个运算时决定先进行哪个运算。运算的顺序会改变运算的结果，就像下面的例子演示的那样。

假设你有 $8 + 4 / 2$ 这样的代码。如果你先进行除法预算，你会得到 $8 + 2$ ，或者是 10。然而，如果你先进行加法运算，你就会得到 $12 / 2$ ，或者是 6。

现在编程语言对运算都赋予了优先级。如果你知道这些优先级，你可以编写 $8 + 4 / 2$ 这样的表达式，并且你知道它的运算结果。但是这并不可靠。

首先，要是你得到了错误的结果会怎样呢？或者，要是其他查看代码的人并没有你这样的记忆力会怎样呢？再或者，要是你记住了一种语言的优先级、但 Perl 采用的是不同的优先级会怎样呢？（不同的语言确实有不同的优先级规则。）

有一种解决办法，这就是使用括号。对于例 9.3，如果你简单的添加上括号： $(8 + (4 / 2))$ ，对于你自己、其他读者以及 Perl 程序来说，都可以明确无误地知道你想首先进行除法运算。注意包裹在另外一对括号中的“内部”括号，会被首先求值。

记住，在复杂的表达式中使用括号来指明运算的顺序。另外，它会让你避免大量的程序调试！

9.4 练习题

习题 9.1

修改例 9.3，让它可以从命令行接收 DNA；如果它没有被指定，提示用户键入 FASTA 文件名并读入 DNA 序列数据。

习题 9.2

修改 Exercise 9.1，从 *bionet* 文件中读入全部的 REBASE 限制性内切位点数据，并把它制作成一个散列。

习题 9.3

修改 Exercise 9.2，如果不存在 DBM 文件，就通过存储的 REBASE 散列生成一个 DBM 文件，如果 DBM 文件存在，就直接使用 DBM 文件。（提前查阅第 10 章中关于 DBM 的更多的信息。）

习题 9.4

修改例 5.3，报告它找到的基序的位置，即使基序在序列数据中出现多次。

习题 9.5

通过打印出序列、并用酶的名字把限制性内切位点标记出来，为限制酶切图谱添加一个切割位点的图形化展示。你能制作一个处理多个限制性内切酶的图谱吗？你该如何处理重叠的限制性内切位点呢？

习题 9.6

编写一个子程序，返回限制酶切消化片段，就是进行限制性酶切反应后留下的 DNA 片段。记住要考虑切割位点的位置。（这需要你以另一种方式解析 REBASE 的 *bionet* 文件。如果你愿意的话，可以把没有用 ^ 指明切割位点的限制性内切酶忽略掉。）

习题 9.7

扩展限制酶切图谱软件，对于非回文识别位点，把相反的另一条链也考虑在内。

习题 9.8

对于没有括号的算术运算，编写一个子程序，根据 Perl 的优先级规则为其添加上合适的括号。（警告：这是一个相当有难度的练习题，除非你确信有足够的课余时间，否则请跳过该题。对于优先级规则，可以参看 Perl 的文档。）

第 10 章 GenBank

目录

10.1	GenBank 文件	213
10.2	GenBank 库	216
10.3	分割序列和注释	218
10.4	解析注释	225
10.5	使用 DBM 对 GenBank 进行索引	244
10.6	练习题	248

GenBank (Genetic Sequence Data Bank) 是一个快速增长的国际性知识库, 存储的是各种各样生物的已知遗传序列。它的使用对于现代生物学和生物信息学来说至关重要。

本章将向你展示如何编写 Perl 程序来从 GenBank 文件和库中提取信息。练习题包括查找模式、创建特定的库, 以及解析平面文件格式来提取 DNA、注释和特征。你将学习如何制作一个 DBM 数据库, 实现自己在 GenBank 库中对特定数据进行快速的访问与查找。

Perl 是处理 GenBank 文件的一个优秀的工具。它可以让你提取并使用序列以及 FEATURES 表和其他等注释中的任何细节性的数据。当我第一次使用 Perl 的时候, 我编写了一个程序, 检索 GenBank 中所有被注释为位于人类第 22 号染色体上的序列记录。我发现许多基因, 它们的信息被深深地隐藏在了注释中, 以至于 GDB (Genome Database) 这个主要的基因图谱数据库都没有把它们包含在自己的染色体图谱中。我相信, 当你开始应用 Perl 处理 GenBank 文件的时候, 你也会对其中的信息有与我同样强烈的感觉。

大多数生物学家对 GenBank 都非常熟悉。研究人员可以进行检索, 比如对于某个查询序列的 BLAST 检索, 也可以把相关序列的一系列 GenBank 文件作为结果收集起来。因为 GenBank 记录是由发现序列的特定科学家进行维护的, 所以如果你发现了某个新的有趣的序列, 你也可以把它发布到 GenBank 上。

GenBank 文件中除了序列数据以外, 还有一大堆的信息, 包括登录号和基因名这样的识别号、系谱分类和发表文献的参考信息等。一个 GenBank 文件可能还包含详细的 FEATURES 表, 对序列的情况进行了总结, 比如调控区域、蛋白质翻译以及外显子和内含子的定位区域。

GenBank 有时被看做是数据银行 (*databank*) 或者数据商店 (*data store*), 这和数据库 (*database*) 有一定的区别。数据库通常给数据强加一个关系型的结构, 包括相关的指数、链接和查询语言。相比之下, GenBank 就是一个平面文件 (*flat file*), 换言之, 就是一个对于人类易读的 ASCII 码文本文件。¹

¹GenBank 也有 ASN.1 格式的发布, 你需要使用 NCBI 提供特定工具才能对其进行处理。

从它毫不起眼的诞生之初，GenBank 就开始了飞速增长，而在增长过程中平面文件则凸显了其不足之处。随着知识体的快速发展，尤其是像遗传数据这种知识的快速增长，要想让数据银行的这种设计与与时俱进已经很难了。对 GenBank 进行重新整理设计的许多共组已经完成了，但是平面文件——顶着它逐渐褪色的皇冠——仍然存在着。

归咎于 GenBank 记录中某些小节内容的一定灵活性，从中提取要寻找的信息可能会比较复杂。这种灵活性有好的一面，它允许你把你认为是最重要的东西都放到数据的注释中；但同时它也有不好的一面，因为同样是灵活性，它会使编写程序来寻找并提取需要的注释信息变得更加困难。正因为如此，现在的趋势是让注释中的内容更具结构化。

Perl 的数据结构和正则表达式的应用使得它成为处理平面文件的优秀工具，尤其适合用来处理 GenBank 数据。使用 Perl 的这些特性，基于前面章节训练的技能，你可以编写程序来访问 GenBank 中对于科学界来说日益积累的遗传知识。

因为这是本不需要编程经验的初学者指南书籍，所以你不要指望从中找到完美无瑕、包治百病的软件。与之相反，你会找到对于对于 GenBank 文件进行解析和构建快速查找表的详尽介绍。如果你还从没有做过类似的事情，我强烈推荐你去探索一下 NIH (National Institutes of Health) (<http://www.ncbi.nlm.nih.gov>) 的 NCBI (National Center for Biotechnology Information)。当你开始之后，停止在 <http://www.ebi.ac.uk> 的 EBI (European Bioinformatics Institute) 和 <http://www.embl-heidelberg.de/> 的生物信息学分支 EMBL (European Molecular Biology Laboratory) 即可。它们都是大型的、由重金赞助支持的政府性的生物信息学中心，并且它们都有（并且发布了）大量的顶尖的生物信息学软件。

10.1 GenBank 文件

主要的遗传信息库就是 NCBI GenBank、欧洲的 EMBL 和日本的 DDBJ (DNA Data Bank of Japan)。因为有国际合作协定，它们都有几乎完全相同的信息。GenBank 或者它的镜像站点中的每一个条目或者记录都包含确定的描述性的遗传信息，存储在 ASCII 格式的文件中。每一个记录都用特定的标准格式进行编写与组织，这样不管是人还是计算机程序都可以比较容易地从中提取需要的信息。

让我们看一个相对较短的 GenBank 记录，在编写代码之前先看看字段是如何定义的。我会把这些信息保存在一个叫做 *record.gb* 的文件中，便于后面程序的使用。

```
1 LOCUS      AB031069      2487 bp      mRNA                      PRI      27-MAY-2000
2 DEFINITION Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
3           complete cds.
4 ACCESSION  AB031069
5 VERSION    AB031069.1  GI:8100074
6 KEYWORDS   .
7 SOURCE     Homo sapiens embryo male lung fibroblast cell_line:HuS-L12 cDNA to
8           mRNA.
9   ORGANISM Homo sapiens
10           Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
11           Mammalia; Eutheria; Primates; Catarrhini; Hominidae; Homo.
12 REFERENCE 1 (sites)
13   AUTHORS  Fujino,T., Hasegawa,M., Shibata,S., Kishimoto,T., Imai,Si. and
14           Takano,T.
15   TITLE     PCCX1, a novel DNA-binding protein with PHD finger and CXXC domain,
16           is regulated by proteolysis
17   JOURNAL   Biochem. Biophys. Res. Commun. 271 (2), 305-310 (2000)
18   MEDLINE   20261256
19 REFERENCE 2 (bases 1 to 2487)
20   AUTHORS  Fujino,T., Hasegawa,M., Shibata,S., Kishimoto,T., Imai,S. and
21           Takano,T.
22   TITLE     Direct Submission
23   JOURNAL   Submitted (15-AUG-1999) to the DDBJ/EMBL/GenBank databases.
24           Tadahiro Fujino, Keio University School of Medicine, Department of
25           Microbiology; Shinanomachi 35, Shinjuku-ku, Tokyo 160-8582, Japan
26           (E-mail:fujino@microb.med.keio.ac.jp,
27           Tel:+81-3-3353-1211(ex.62692), Fax:+81-3-5360-1508)
28 FEATURES   Location/Qualifiers
29     source   1..2487
30             /organism="Homo sapiens"
31             /db_xref="taxon:9606"
32             /sex="male"
33             /cell_line="HuS-L12"
34             /cell_type="lung fibroblast"
35             /dev_stage="embryo"
36     gene     229..2199
37             /gene="PCCX1"
```

```
38 CDS 229..2199
39 /gene="PCCX1"
40 /note="a nuclear protein carrying a PHD finger and a CXXC
41 domain"
42 /codon_start=1
43 /product="protein containing CXXC domain 1"
44 /protein_id="BAA96307.1"
45 /db_xref="GI:8100075"
46 /translation="MEGDGSDPEPPDAGEDSKSENGENAPIYICIRKPDINCFMIGCD
47 NCNEWFHGD CIRITEKMAKAIREWYCREKREKDPKLEIRYRHKKS RERDGNERSSEP
48 RDEGGGRKRPVPDPDLQRRAGSGTGVGAMLARGSASPHKSSPQLVATPSQHHQQQQQ
49 QIKRSARMCGECEACRRTEDCGHCDFCRDMKKFGGPNKIRQKCRLRQCQLRARES YKY
50 FPSSLSPVTPSESLPRRRPLPTQQQPQPSQKLGRIREDEGAVASSTVKEPPEATATP
51 EPLSDEDLPLDPDLYQDFCAGAFDDHGLPWMSDTEESPFDPALRKRAVKVKHVKRRE
52 KKSEKKKEERYKRHRQKQKHKDKWKHPPERADAKDPASLPQCLGPGCVRPAQPSSKYCS
53 DDCGMKLAANRIYEILPQRIQQWQQSPCIAEEHGKKLLERIRREQQSARTRLQEMERR
54 FHELEAII LRAKQQAVREDEESNEGDSDDTDLQIFCVSCGHPINPRVALRHMERCYAK
55 YESQTSFGSMYPTRIEGATRLFCDVYNPQSKTYCKRLQVLCPEHSRDPKVPADDEVCGC
56 PLVRDVFELTGDFCRLPKRQCNRHYCWEKLRRAEVDLERVRVWYKLDELFEQERNVRT
57 AMTNRAGLLALMLHQTIQHDLPLTTDLRSSADR"
58 BASE COUNT 564 a 715 c 768 g 440 t
59 ORIGIN
60 1 agatggcggc gctgaggggt cttgggggct ctaggccggc cacctactgg tttgcagcgg
61 61 agacgacgca tggggcctgc gcaataggag tacgctgcct gggaggcgtg actagaagcg
62 121 gaagtagttg tgggcgcctt tgcaaccgcc tgggacgccg ccgagtggtc tgtgcaggtt
63 181 cgcgggtcgc tggcgggggt cgtgagggag tgcgccggga gcggagatat ggaggagat
64 241 ggttcagacc cagagcctcc agatgccggg gaggacagca agtccgagaa tgggggagaat
65 301 gcgcccattc actgcatctg ccgcaaaccg gacatcaact gcttcatgat cgggtgtgac
66 361 aactgcaatg agtgggttcca tggggactgc atccggatca ctgagaagat ggccaaggcc
67 421 atccgggagt ggtactgtcg ggagtgcaga gagaaagacc ccaagctaga gattcgctat
68 481 cggcacaaga agtcacggga gcgggatggc aatgagcggg acagcagtga gccccgggat
69 541 gaggggtggag ggcgcaagag gcctgtccct gatccagacc tgcagcgccg ggcaggggtca
70 601 gggacagggg ttggggccat gcttgctcgg ggctctgctt cgccccacaa atcctctccg
71 661 cagcccttgg tggccacacc cagccagcat caccagcagc agcagcagca gatcaaagg
72 721 tcagcccgca tgttgtgtga gtgtgaggca tgtcggcgca ctgaggactg tggtcactgt
73 781 gattttctgtc gggacatgaa gaagttcggg ggccccacaa agatccggca gaagtgccgg
74 841 ctgcgccagt gccagctgcg ggcccgggaa tcgtacaagt acttcccttc ctcgctctca
75 901 ccagtgacgc cctcagagtc cctgccaaagg ccccgccggc cactgcccac ccaacagcag
76 961 ccacagccat cacagaagtt agggcgcatc cgtgaagatg agggggcagt ggcgtcatca
77 1021 acagtcaagg agcctcctga ggctacagcc acacctgagc cactctcaga tgaggacct
78 1081 cctctggatc ctgacctgta tcaggacttc tgtgcagggg cctttgatga ccatggcctg
79 1141 ccctggatga gcgacacaga agagtcccca ttctggacc ccgcgctgcg gaagagggca
80 1201 gtgaaagtga agcatgtgaa gcgtcgggag aagaagtctg agaagaagaa ggaggagcga
81 1261 tacaagcggc atcggcagaa gcagaagcac aaggataaat ggaaacaccc agagagggct
82 1321 gatgccaaagg accctgcgtc actgccccag tgcctggggc ccggctgtgt gcgccccgcc
83 1381 cagcccagct ccaagtattg ctcagatgac tgtggcatga agctggcagc caaccgcata
84 1441 tacgagatcc tccccagcg catccagcag tggcagcaga gcccttgcat tgctgaagag
85 1501 cacggcaaga agctgctcga acgcattcgc cgagagcagc agagtgcccg cactgcctt
86 1561 caggaaatgg aacgccgatt ccatgagctt gaggccatca ttctacgtgc caagcagcag
```

```

87      1621 gctgtgcgcg aggatgagga gagcaacgag ggtgacagtg atgacacaga cctgcagatc
88      1681 ttctgtgttt cctgtgggca ccccatcaac ccacgtgttg ccttgcgcca catggagcgc
89      1741 tgctacgcca agtatgagag ccagacgtcc tttgggtcca tgtacccac acgcattgaa
90      1801 ggggccacac gactcttctg tgatgtgtat aatcctcaga gcaaaacata ctgtaagcgg
91      1861 ctccagggtgc tgtgccccga gcactcacgg gaccccaaag tgccagctga cgaggatatgc
92      1921 gggtgcccc ttgtacgtga tgtctttgag ctcacgggtg acttctgccg cctgccaag
93      1981 cgccagtgc atcgccatta ctgctgggag aagctgcggc gtgcggaagt ggacttggag
94      2041 cgcgtgcgtg tgtggtacaa gctggacgag ctgtttgagc aggagcgcaa tgtgcgcaca
95      2101 gccatgacaa accgcgcggg attgctggcc ctgatgctgc accagacgat ccagcacgat
96      2161 cccctcacta ccgacctgcg ctccagtgcc gaccgctgag cctcctggcc cggacccctt
97      2221 acaccctgca ttccagatgg gggagccgcc cggtgcccgt gtgtccgttc ctccactcat
98      2281 ctgtttctcc ggttctccct gtgcccattc accggttgac cgccatctg cctttatcag
99      2341 agggactgtc cccgtcgaca tgttcagtgc ctggtggggc tgcggagtcc actcatcctt
100     2401 gcctcctctc cctgggtttt gttaataaaa ttttgaagaa accaaaaaaa aaaaaaaaaa
101     2461 aaaaaaaaaa aaaaaaaaaa aaaaaaa
102 //

```

即使你已经习惯了看 GenBank 文件，当你考虑如何编写一个程序来提取数据的各个部分时，再花一些时间从头到尾看一下也是值得的。比如，你该如何提取序列数据呢？FEATURES 表和它的各个子字段的格式是什么样子的？

在一个典型的 GenBank 条目中，压缩进了大量信息，能把这些不同的部分分隔开来是非常重要的。比如，如果你能提取出序列，你就可以查找基序、计算序列的统计信息、寻找它和其他序列的相似性，等等。类似的，你可能想把数据注释的各个部分分割开来或者进行解析。在 GenBank 中，这包括 ID 号、基因名、属种和发表文献等。注释中的 FEATURES 表部分包含了 DNA 的特定信息，像外显子、调控区域、重要突变的位置等。

GenBank 文件的格式规范，以及关于 GenBank 的各种其他信息可以在 GenBank 的版本注释文件 *gbrel.txt* 文件中找到，这个文件位于 GenBank 的网站 <ftp://ncbi.nlm.nih.gov/genbank/gbrel.txt> 上。

gbrel.txt 给出了 GenBank 文件结构的完整详细的信息，对于程序员有很大的帮助，所以当你的检索越来越复杂的时候，你可能会想去看一看它。作为一名 Perl 程序员，你并不需要所有的细节，因为你可以使用正则表达式或者 *split* 函数来解析数据。你需要把数据提取出来，让你的程序可以使用它。就像你在本章中将要看到的，完成该任务的代码其实非常简单。

10.2 GenBank 库

GenBank 以一系列的库进行发布，也就是包含连续的多个记录的平面文件。²对于 2001 年 8 月份发布的 GenBank 的 125.0 版，一共有 243 个文件，大多数文件的大小都超过了 200 MB。总算起啦，GenBank 包含了来自 12,813,526 条报道序列的 12,813,516 个位点和 13,543,364,296 个碱基。GenBank 库同样以压缩格式进行发布，这也就意味着你可以下载相对较小的文件，但是在你获取到它们后你需要对它们进行解压缩。解压缩后数据的总量大约有 50 GB。从 1982 年开始，大约每 14 各月 GenBank 中序列的数目就会翻番。

根据它们包含的序列类型，或者是系统发育，或者是测序技术，GenBank 库又进一步组织、分成不同的类。下面是这些类别：

- PRI (primate sequences) : 灵长类动物序列
- ROD (rodent sequences) : 啮齿类动物序列
- MAM (other mammalian sequences) : 其他哺乳动物序列
- VRT (other vertebrate sequences) : 其他脊椎动物序列
- INV (invertebrate sequences) : 无脊椎动物序列
- PLN (plant, fungal, and algal sequences) : 植物、真菌和藻类序列
- BCT (bacterial sequences) : 细菌序列
- VRL (viral sequences) : 病毒序列
- PHG (bacteriophage sequences) : 噬菌体序列
- SYN (synthetic and chimeric sequences) : 合成和嵌合序列
- UNA (unannotated sequences) : 未注释序列
- EST (EST (expressed sequence tags) sequences) : EST (表达序列标签) 序列
- PAT (patent sequences) : 专利序列
- STS (STS (sequence tagged sites) sequences) : STS (序列标签位点) 序列
- GSS (GSS (genome survey sequences) sequences) : GSS (基因组勘测序列) 序列
- HTG (HTGS (high throughput genomic sequencing data) sequences) : HTGS (高通量基因组测序数据) 序列
- HTC (HTC (high throughput cDNA sequencing data) sequences) : HTC (高通量 cDNA 测序数据) 序列

有些类别非常大：最大的就是 EST (表达序列标签) 类别，它由 123 个库文件构成！人类 DNA 的一部分存储在 PRI 类别中，它包含 13 个库文件（本书撰写期间），总共大约有 3.5 GB 的数据。人类的数据还存储在 STS、GSS、HTGS 和 HTC 类别中。单单是 GenBank 中人类的数据就有近 5 百万记录条目，序列的碱基数超过了 8 兆（万亿）。

<http://www.ncbi.nlm.nih.gov/> 上的 Entrez 和 BLAST 等公共数据库服务器，可以让你访问进行良好维护和升级的序列数据和程序，但是许多研究人员发现它们需要编写自己的程序来处理和分析这些数据。问题在于，数据量实在太大了。对于大多数的研究目的，你只需要从 NCBI 或者其他地方下载选定的一部分记录，但有时候你需要全部的数据集。

可以构建一个 (Windows、Mac、Unix 或者 Linux) 桌面工作站，把所有的 GenBank 都包含在内，但是一定要确保购买了一个足够大的硬盘！然而，把所有数据都下载到你的硬盘中是非常困难的。一个叫做 *mirror.pl* 的程序可以帮你完成这个工作。即使是大学

²数据也以 ASN.1 格式进行发布。

标准的高速因特网连接，下载这些数据也是一个非常耗时的工作；如果使用调制解调器下载全部的数据集一定会让你抓狂的。最好的解决方案是只下载你需要的文件，而且是压缩格式的文件。比如，EST 数据，它大约是整个数据库的一半，除非你真的需要它，否则不要去下载它。如果你需要下载 GenBank，我推荐你联系 NCBI 的服务台。它们会帮助你获取最新的信息。

既然你在学习编程，那在一个小的、只有五条记录的库文件上练习就完全够了，当然，你编写的程序在真实的文件中也是完全可以工作的。

10.3 分割序列和注释

在上一章中，你看到了如何使用 Perl 的数组操作来检查文件的行。通常，你会把数据保存到一个数组中，每一行都是数组的一个元素。

让我们看看从 GenBank 文件中提取注释和 DNA 的两种方法。在第一种方法中，你会把文件一股脑的都放进数组中，然后像上一章的程序一样逐行进行处理。在第二种方法中，你会把整个的 GenBank 记录放到一个标量变量中，然后使用正则表达式来解析信息。是不是某种方法比另一种更好一些？并不一定，这取决于你的数据。每一种方法都有自己的优缺点，但不管怎样，它们都可以完成任务。

我已经把五条 GenBank 记录放在了一个叫做 *library.gb* 的文件中。就像前面一样，你可以从本书的网站上下载这个文件。在接下来的几个例子中，你将使用这个数据文件和 *record.gb* 这个文件。

10.3.1 使用数组

例 10.1 演示了第一种方法，它对包含 GenBank 记录行的数组进行操作。主程序后面跟着的是真正起作用的子程序。

例 10.1：从 GenBank 文件中提取注释和序列

```

1  #!/usr/bin/perl
2  # Example 10-1   Extract annotation and sequence from GenBank file
3
4  use strict;
5  use warnings;
6  use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8  # declare and initialize variables
9  my @annotation = ();
10 my $sequence   = '';
11 my $filename   = 'record.gb';
12
13 parse1( \@annotation, \$sequence, $filename );
14
15 # Print the annotation, and then
16 #   print the DNA in new format just to check if we got it okay.
17 print @annotation;
18
19 print_sequence( $sequence, 50 );
20
21 exit;
22
23 #####
24 # Subroutine
25 #####
26
27 # parse1
28 #
```

```

29 # -parse annotation and sequence from GenBank record
30
31 sub parse1 {
32
33     my ( $annotation, $dna, $filename ) = @_;
34
35     # $annotation-reference to array
36     # $dna          -reference to scalar
37     # $filename     -scalar
38
39     # declare and initialize variables
40     my $in_sequence = 0;
41     my @GenBankFile = ();
42
43     # Get the GenBank data into an array from a file
44     @GenBankFile = get_file_data($filename);
45
46     # Extract all the sequence lines
47     foreach my $line (@GenBankFile) {
48
49         if ( $line =~ /^\\n/ ) {      # If $line is end-of-record line //n,
50             last;                    # break out of the foreach loop.
51         }
52         elsif ($in_sequence) {        # If we know we're in a sequence,
53             $$dna .= $line;           # add the current line to $$dna.
54         }
55         elsif ( $line =~ /^ORIGIN/ ) { # If $line begins a sequence,
56             $in_sequence = 1;         # set the $in_sequence flag.
57         }
58         else {                        # Otherwise
59             push( @$annotation, $line ); # add the current line to @annotation.
60         }
61     }
62
63     # remove whitespace and line numbers from DNA sequence
64     $$dna =~ s/[\s0-9]//g;
65 }

```

下面是例 10.1 输出的序列数据的开头和结尾部分：

```

1 | agatggcggcgctgaggggtcttgggggctctaggccggccacctactgg
2 | ttgcagcggagacgacgcatggggcctgcgcaataggagtacgctgcct
3 | gggaggcgctgactagaagcggagtagttgtgggcgcctttgcaaccgcc
4 | tgggacgccgccgagtggtctgtgcaggttcgcgggtcgctggcgggggt
5 | cgtgagggagtgcgccgggagcggagatatggagggagatggttcagacc
6 | ...
7 | cgggtgcccgtgtgtccgttcctccactcatctgtttctccggttctccct
8 | gtgcccacccacgggttgaccgcccatctgcctttatcagagggactgtc
9 | cccgtcgacatgttcagtgctggtggggctgcggagtccactcatcctt

```

```

10 | gcctcctctccctgggttttgttaataaaattttgaagaaacccaaaaaaa
11 | aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

```

在例 10.1 中，子程序 `parse1` 中的 `foreach` 循环把存储在数组 `@GenBankFile` 中的 GenBank 文件的行进行逐行处理。它充分利用了 GenBank 文件的结构，其中的注释从开头开始一直到下面这一行结束：

```
ORIGIN
```

其后便是序列，一直到记录终止行 `//` 为止。循环使用一个标识变量 `$in_sequence` 来记住它已经找到了 `ORIGIN` 行、现在正在读取序列行。

`foreach` 循环有一个新的特性：Perl 的内置函数 `last`，它会跳出包裹在最内层的循环。这会由记录终止行 `//` 触发，只有当整个记录都被处理后会到达这一行。

为了寻找记录终止行，使用了一个正则表达式。为了能够正确地匹配记录终止行中的（正）斜杠，你必须在每一个前面都放上反斜杠对它们进行转义，这样 Perl 就不会把它们解释成模式的提前终止了。正则表示式也以换行符结束 `\\//\\n`，所以把它放到匹配的定界符中间：`/\\//\\n/`。（当你在一个正则表达式中有许多正斜杠时，你可以使用其他的定界符把正则表达式包裹起来，并在其前面使用 `m`，这样就可以避免在正斜杠前面使用反斜杠了。就像这样：`m!//\\n!`）。

对于子程序 `parse1` 来说，比较有趣的一点是 `foreach` 循环对 GenBank 记录进行逐行处理时的检测顺序。当你逐行阅读记录时，你会想首先收集注释行，当读到序列开始行 `ORIGIN` 时设置一个标识，然后收集序列行，直到记录终止行 `//` 为止。

注意检测的顺序是完全相反的。首先，你检测记录终止行，如果 `$in_sequence` 标识被设置了就收集序列，然后检测序列开始行 `ORIGIN`。最后，你收集注释信息。

逐行读取文件和使用标识变量来标记文件小节的技术，是一个很常见的编程技术。所以，花点时间来想一下，如果你改变了检测的顺序，循环的行为会有什么变化。如果你在检测记录终止行之前就收集序列行，那么你将永远也不会进行记录终止行的检测！

使用其他方法收集注释和序列行也是可以的，尤其是当你多次遍历数组的行时。你可以扫描整个数组，记住序列开始行和记录终止行的行号，然后返回去，使用数组切片（在例 9.2 的子程序 `parseREBASE` 中对其进行过介绍）提取注释和序列。下面是一个例子：

```

1 | # find line numbers of ORIGIN and // in the GenBank record
2 |
3 | $linenumber = 0;
4 | foreach my $line (@GenBankFile) {
5 |     if ( $line =~ /^\\/\\n/ ) { # end-of-record // line
6 |         $end = $linenumber;
7 |         last;
8 |     } elsif ( $line =~ /^ORIGIN/ ) { # end annotation, begin sequence
9 |         $origin = $linenumber;
10 |    }
11 |    $linenumber++;
12 | }
13 |
14 | # extract annotation and sequence with "array splice"
15 |
16 | @annotation = @GenBankFile[0..($origin-1)];
17 | @sequence   = @GenBankFile[($origin+1)..($end-1)];

```

10.3.2 使用标量

第二种把 GenBank 记录中的注释和序列分开的方法是，把整个记录读取到一个标量变量中，然后用正则表达式操作它。对于某些类型的数据来说，（和例 10.1 中的遍历数组相比，）这种方法可能是解析输入更加方便的一种方法。

通常对于字符串数据来说，都是把一行存储到单个的标量变量中，如果字符串末尾还有换行符，也会把换行符包含在内。然而，有时你也可以把多行连接在一起，把连接起来的单个字符串保存到一个单独的标量变量中。这种多行组成的字符串并不常见。还记得吗，在例 6.2 和例 6.3 中你曾经使用它们来从 FASTA 文件中收集序列。正则表达式有专门的模式修饰符，让你可以轻松处理含有换行符的多行字符串。

模式修饰符

到现在为止，我们使用过的模式修饰符包括用于全局匹配的 `/g` 和不去分大小写进行匹配的 `/i`。让我们看一下另外两个模式修饰符，它们可以影响正则表达式处理含有换行符的标量的行为。

回顾一下，前面的正则表达式已经使用过脱字符号（`^`）、点号（`.`）和美元符号（`$`）这三个元字符。`^` 默认会把一个正则表达式锚定到字符串的开头，所以 `/^THE BEGUINE/` 匹配以“THE BEGUINE”起始的字符串。与之类似，`$` 把一个正则表达式锚定到字符串的末尾，点号（`.`）会匹配除换行符以外的任意一个字符。

下面的这些模式修饰符会影响这三个元字符的行为：

- `/s` 修饰符假设你想把真个字符串作为单独的一行，即使其中有换行符也是一样，所以它使得点号元字符可以匹配包括换行符在内的任意一个字符。
- `/m` 修饰符假设你像把整个字符串作为含有换行符的当行进行处理，所以它使得 `^` 和 `$` 可以匹配字符串内部换行符的后面和前面。

模式修饰符实例

下面是展示脱字符号（`^`）、点号（`.`）和美元符号（`$`）默认行为的例子：

```
1 | use warnings;  
2 | "AAC\nGTT" =~ /^.*$/;  
3 | print $&, "\n";
```

它演示的是在没有使用 `/m` 和 `/s` 修饰符的情况下的默认行为，它会输出警告信息：

```
1 | Use of uninitialized value in print statement at line 3.
```

`print` 语句试图输出 `$&`，这是一个特殊变量，它总是会被赋值为上一个成功匹配的模式。在这个例子中，无法匹配模式，变量 `$&` 并没有被赋值，所以在尝试输出一个未初始化的值的时候你会得到警告信息。

为什么匹配不能够成功呢？首先，让我们检查一下 `^.*$` 这个模式。它以 `^` 起始，表示必须从字符串的开头进行匹配。它以 `$` 结尾，表示必须同时对字符串的末尾（字符串的末尾可能会包含一个换行符，但不允许有更多的换行符存在）进行匹配。`.*` 表示它必须匹配除换行符以外的任意一个字符（`.`）零次或者多次（`*`）。所以，换言之，模式 `^.*$`

匹配不包含换行符在内的任意一个字符串，除非可能存在的单个换行符是最后的字符。但是上面这个例子中的字符串“ACC\nGTT”，却包含一个不是最后字符的内部的换行符，所以模式匹配失败。

在下面两个例子中，模式修饰符 `/m` 和 `/s` 会改变元字符 `^`、`$` 和点号的默认行为：

```
1 | "AAC\nGTT" =~ /^.*$/m;
2 | print $&, "\n";
```

这个代码片段会输出 AAC，演示了 `/m` 修饰符的作用。`/m` 会扩展 `^` 和 `$` 的含义，这样它们也会对包含在内部的换行符的周边进行匹配了。此处，模式会从字符串的开头一直匹配到第一个内部换行符为止。

接下来的这个代码片段演示了 `/s` 修饰符的作用：

```
1 | "AAC\nGTT" =~ /^.*$/s;
2 | print $&, "\n";
```

这会输出：

```
1 | AAC
2 | GTT
```

`/s` 修饰符会改变点号元字符的含义，这样它能够匹配包含换行符在内的任意一个字符了。通过使用 `/s` 修饰符，模式会从字符串的开头开始一直匹配到字符串的结尾，包括换行符在内的所有东西。注意当它输出时，它把内部的换行符也输出出来了。

把注释和序列分割开来

既然你已经学习了模式匹配修饰符，那么接下来正则表达式将是你把 GenBank 文件作为一个标量进行解析的主要工具，让我们试着把注释和序列分割开来吧。

第一步是把 GenBank 记录存储到一个标量变量中。回顾一下，GenBank 记录开始于以“LOCUS”起始的一行，终止于记录终止分隔符，也就是包含两个正斜杠的行。

首先，你想把 GenBank 记录读取进来，并存储到一个标量变量中。有一个叫做输入记录分隔符的设备，它用特殊变量 `$/` 表示，可以让你指定输入记录的分隔符。输入记录分隔符通常设定为换行符，所以当从一个文件句柄中获取记录保存到标量时得到的是一行。可以像下面这样把它设置成 GenBank 的记录终止分隔符：

```
1 | $/ = "//\n";
```

这样，当从文件句柄读取一个标量时，会把到 GenBank 记录终止分隔符的所有数据都读进来。所以例 10.2 中的 `$record = <GBFILE>` 这行语句会把多行的 GenBank 记录存储到标量变量 `$record` 中。稍后你会看到，可以持续进行调用，把 GenBank 库文件中连续的多个 GenBank 记录读取进来。

在读取完记录后，你要使用 `/s` 和 `/m` 模式修饰符，把它解析成注释和序列两个部分。提取注释和序列是比较容易的，而解析注释则将占据本章剩余的大部分内容。

例 10.2: 从 GenBank 记录中提取注释和序列

```

1  #!/usr/bin/perl
2  # Example 10-2  Extract the annotation and sequence sections from the first
3  #   record of a GenBank library
4
5  use strict;
6  use warnings;
7  use BeginPerlBioinfo;    # see Chapter 6 about this module
8
9  # Declare and initialize variables
10 my $annotation      = '';
11 my $dna              = '';
12 my $record           = '';
13 my $filename         = 'record.gb';
14 my $save_input_separator = $/;
15
16 # Open GenBank library file
17 unless ( open( GBFILE, $filename ) ) {
18     print "Cannot open GenBank file \"$filename\"\n\n";
19     exit;
20 }
21
22 # Set input separator to "//\n" and read in a record to a scalar
23 $/ = "//\n";
24
25 $record = <GBFILE>;
26
27 # reset input separator
28 $/ = $save_input_separator;
29
30 # Now separate the annotation from the sequence data
31 ( $annotation, $dna ) = ( $record =~ /^(LOCUS.*ORIGIN\s*\n)(.*)\\/\n/s );
32
33 # Print the two pieces, which should give us the same as the
34 #   original GenBank file, minus the // at the end
35 print $annotation, $dna;
36
37 exit;

```

该程序的输出和前面展示的 GenBank 文件的内容是完全一样的，但是没有最后一行，也就是没有记录终止分隔符 //。

让我们把焦点放在从 \$record 变量中解析注释和序列的正则表达式上。这是到现在为止最复杂的一个正则表达式：

```
1 | $record = /^(LOCUS.*ORIGIN\s*\n)(.*)\\/\n/s.
```

在正则表达式中有两对括号：(LOCUS.*ORIGIN\s*\n) 和 (.*). 小括号是元字符，它的目的是捕获数据中匹配小括号内模式的部分，换言之，就是此处的注释和序列。还要

注意，模式匹配返回的是一个数组，这个数组的元素就是匹配的被小括号括起来的模式。在你用正则表达式中成对的小括号对注释和序列进行匹配之后，就可以简单地把匹配到的模式赋值给 `$annotation` 和 `$dna` 这两个变量了，就像这样：

```
1 | ($annotation, $dna) = ($record =~ /^(\LOCUS.*ORIGIN\s*\n)(.*)\\\/\n/s);
```

注意，在模式的最后，我们加上了 `/s` 这个模式匹配修饰符，就像你在前面看到的那样，它允许点号去匹配包括内部换行符在内的任意一个字符。（当然，因为我们把整个的 GenBank 记录都放到了 `$record` 变量中，所以其中有很多个嵌入的内部换行符。）

接下来，先看一下第一对小括号：

```
1 | (\LOCUS.*ORIGIN\s*\n)
```

因为前面有一个 `^` 元字符，所以整个表达式都被锚定在了字符串的开头。（`/s` 并不改变正则表达式中 `^` 字符的含义。）

在小括号内部，你从 GenBank 记录开头的 `LOCUS` 字符串出现的地方开始匹配，接着是使用 `.*` 表示的任意数目的包括换行符在内的任意字符，然后是 `ORIGIN` 字符串，用 `\s*` 表示其后可能有一些空白，最后是一个换行符 `\n`。它匹配的其实就是 GenBank 记录中的注释部分。

接下来，让我们再看一下第二个小括号和剩余的部分：

```
1 | (.*)\\\/\n
```

这个相对简单一些。`.*` 匹配的是包括换行符在内的任意一个字符，因为在模式匹配的后面使用了 `/s` 模式修饰符。小括号后面紧跟着的是记录终止行 `//`，以及最后的换行符。反斜杠前面的正斜杠表明你是想要匹配它们本身。它们并不是模式匹配操作符的分隔符。最后的结果就是，GenBank 记录中的注释和序列被分开，分别保存到了变量 `$annotation` 和变量 `$sequence` 中。尽管我使用的正则表达式需要一定的解释，但它吸引人的地方在于只需要一行 Perl 代码就可以同时把注释和序列提取出来。

10.4 解析注释

既然现在你已经成功把序列提取出来了，那么接下来就让我们解析 GenBank 文件的注释吧。

看看 GenBank 记录，你会发现思考如何把有用的信息提取出来还是比较有趣的。FEATURES 表肯定是我们此处的重中之重。它的结构比较繁杂，哪些是需要保留的，哪些是不需要的呢？比如，有时你只想看看像“endonuclease”这个的一个单词是否在记录的某个地方出现了。像这种情况，你只需要一个可以在注释中查找任意正则表达式的子程序即可。有时这就足够了，但当需要更详尽的调研时，Perl 有你需要的工具来保证任务顺利完成。

10.4.1 使用数组

例 10.3 解析了 GenBank 文件中注释的一部分信息。它通过把数据保存到数组中完成了该任务。

例 10.3：使用数组解析 GenBank 的注释

```
1  #!/usr/bin/perl -w
2  # Example 10-3   Parsing GenBank annotations using arrays
3
4  use strict;
5  use warnings;
6  use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8  # Declare and initialize variables
9  my @genbank   = ();
10 my $locus      = '';
11 my $accession = '';
12 my $organism   = '';
13
14 # Get GenBank file data
15 @genbank = get_file_data('record.gb');
16
17 # Let's start with something simple.  Let's get some of the identifying
18 # information, let's say the locus and accession number (here the same
19 # thing) and the definition and the organism.
20
21 for my $line (@genbank) {
22     if ( $line =~ /^LOCUS/ ) {
23         $line =~ s/^LOCUS\s*//;
24         $locus = $line;
25     }
26     elsif ( $line =~ /^ACCESSION/ ) {
27         $line =~ s/^ACCESSION\s*//;
28         $accession = $line;
29     }
30     elsif ( $line =~ /^  ORGANISM/ ) {
```

```

31     $line =~ s/^\s*ORGANISM\s*//;
32     $organism = $line;
33 }
34 }
35
36 print "*** LOCUS ***\n";
37 print $locus;
38 print "*** ACCESSION ***\n";
39 print $accession;
40 print "*** ORGANISM ***\n";
41 print $organism;
42
43 exit;

```

下面是例 10.3 的输出：

```

1  *** LOCUS ***
2  AB031069      2487 bp      mRNA          PRI          27-MAY-2000
3  *** ACCESSION ***
4  AB031069
5  *** ORGANISM ***
6  Homo sapiens

```

现在，我们来稍微扩充一下程序，让它可以处理 DEFINITION 字段。注意，DEFINITION 字段的内容可能不止一行。要收集该字段，使用在例 10.1 中学到的技巧：当你在收集定义的“状态”时就设置一个标识。毋庸置疑，标识变量还是叫做 \$flag。

例 10.4：使用数组解析 GenBank 注释，第二次尝试

```

1  #!/usr/bin/perl -w
2  # Example 10-4   Parsing GenBank annotations using arrays, take 2
3
4  use strict;
5  use warnings;
6  use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8  # Declare and initialize variables
9  my @genbank      = ();
10 my $locus        = '';
11 my $accession    = '';
12 my $organism     = '';
13 my $definition   = '';
14 my $flag         = 0;
15
16 # Get GenBank file data
17 @genbank = get_file_data('record.gb');
18
19 # Let's start with something simple. Let's get some of the identifying
20 # information, let's say the locus and accession number (here the same
21 # thing) and the definition and the organism.

```

```

22
23 for my $line (@genbank) {
24     if ( $line =~ /^LOCUS/ ) {
25         $line =~ s/^LOCUS\s*//;
26         $locus = $line;
27     }
28     elsif ( $line =~ /^DEFINITION/ ) {
29         $line =~ s/^DEFINITION\s*//;
30         $definition = $line;
31         $flag      = 1;
32     }
33     elsif ( $line =~ /^ACCESSION/ ) {
34         $line =~ s/^ACCESSION\s*//;
35         $accession = $line;
36         $flag      = 0;
37     }
38     elsif ($flag) {
39         chomp($definition);
40         $definition .= $line;
41     }
42     elsif ( $line =~ /^  ORGANISM/ ) {
43         $line =~ s/^  \s*ORGANISM\s*//;
44         $organism = $line;
45     }
46 }
47
48 print "*** LOCUS ***\n";
49 print $locus;
50 print "*** DEFINITION ***\n";
51 print $definition;
52 print "*** ACCESSION ***\n";
53 print $accession;
54 print "*** ORGANISM ***\n";
55 print $organism;
56
57 exit;

```

例 10.4的输出:

```

1  *** LOCUS ***
2  AB031069      2487 bp      mRNA          PRI          27-MAY-2000
3  *** DEFINITION ***
4  Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1, complete
5  cds.
6  *** ACCESSION ***
7  AB031069
8  *** ORGANISM ***
9  Homo sapiens

```

当从含有多行小节的文件中提取信息时，从循环的一次跌倒到下一次迭代，使用标识去记住你现在在文件的哪一部分，这是一种非常常见的技术。随着文件和其字段越来越复杂，在代码中需要一次使用多个标识，记住是处于文件的哪一部分，需要从中提取什么信息。这是完全可行的，但随着文件越来越复杂，代码也会越来越复杂，要阅读和修改它就变得困难起来了。所以，让我们看看如何使用正则表达式这个媒介来解析注释吧。

10.4.2 何时使用正则表达式

我们已经使用了两种方法来解析 GenBank 文件：使用正则表达式和循环处理存储行的数组并设置标识。本章前面的小节中，在分割注释和序列的时候，这两种方法我们都使用过。两种方法没有优劣之分，都完全适用，因为在 GenBank 文件中，注释后面紧跟着的就是序列，两者被 ORIGIN 行明确分割开来，这是一个比较简单的结构。然而，要解析注释看起来就要复杂一些了，因此，让我尝试使用正则表达式来完成这个任务吧。

作为开始，我们先把先前的代码整理一下，把它们整理进一些便捷的子程序，这样我们就可以把精力集中在注释的解析上了。你可能会想从库（一个库文件包含一条或多条 GenBank 记录）中一次只获取一条 GenBank 记录，提取出注释和序列，然后如果需要的话就解析注释。这样是非常有用的，比如当你在 GenBank 库中寻找一些基序的时候。然后你就可以查找基序，如果找到了，你再解析注释来找到和序列相关的其他信息。

前面已经提到了，我们将使用文件 *library.gb*，你可以从本书的网站上下载到它。

既然处理注释数据有些复杂，就让我们花一分钟来把我们的任务分割成几个比较容易处理的子程序吧。下面是伪代码：

```
1 sub open_file
2     given the filename, return the filehandle
3
4 sub get_next_record
5     given the filehandle, get the record
6     (we can get the offset by first calling "tell")
7
8 sub get_annotation_and_dna
9     given a record, split it into annotation and cleaned-up sequence
10
11 sub search_sequence
12     given a sequence and a regular expression,
13     return array of locations of hits
14
15 sub search_annotation
16     given a GenBank annotation and a regular expression,
17     return array of locations of hits
18
19 sub parse_annotation
20     separate out the fields of the annotation in a convenient form
21
22 sub parse_features
23     given the features field, separate out the components
```

思路就是让每一个子程序都只完成一个重要的任务，然后把它们组合成最后有用的程序。其中的某些也可以组合进其他的子程序里，比如，你可能想要打开一个文件并从中获得记录，只需要一个子程序调用就可以了。

你设计这些子程序来处理库文件，也就是含有多个 GenBank 记录的文件。你把文件句柄作为一个参数传递给子程序，这样你的子程序就可以访问打开的用文件句柄代表的库文件了。这样做，你就要有一个 `get_next_record` 函数，便于在循环中使用。使用 Perl 函数 `tell` 可以让你保存任何感兴趣的记录的字节偏移量，稍后再回来，快速地从这个字节偏移量处提取记录。（所谓 *byte offset*（字节偏移量）就是到达文件中感兴趣的信息所在地所要经过的字符数。）操作系统对 Perl 的支持，可以让你立即跳转到任意字节偏移量的地方，就算是巨大的文件也没问题，这样就不需要像通常那样，打开文件后从头开始读取直到到达你要去的那个地方。

当你处理大文件使，使用字节偏移量是非常重要的。Perl 有内置的变量，像是 `seek` 可以让你立即跳转到已打开文件的任意一个地方。思路就是，当你在一个文件中寻找某些东西的时候，你可以使用 Perl 函数 `tell` 把字节偏移量保存下来。然后，当你想返回到文件中的那个地方的时候，你可以直接使用字节偏移量作为参数调用 Perl 函数 `seek`。在本章的后面，当你创建 DBM 文件，基于它们的索引号来查找记录的时候，你会看到这样的用法。要点在于，对于一个 250-MB 的文件，要从头开始查找某些东西会花费很唱得时间，有一些办法可以避免这一点。

根据设计，通过三步来完成对数据的解析：

1. 首先，你把注释和序列（此处，你要清洁数据，进行删除空白之类的工作，这样就可以得到一个简单的序列字符串了）分隔开来。就算在这一步，你也可以在序列中查找基序，在注释中寻找文本。
2. 然后，提取出各个字段。
3. 最后，解析特征表。

这三步看起来一气呵成。依据目的的不同，你可以在任意深度上对数据进行解析。

下面使用伪代码描述的主程序，它演示了如何使用这些子程序：

```

1 | open_file
2 |
3 | while ( get_next_record )
4 |
5 |     get_annotation_and_dna
6 |
7 |     if ( search_sequence for a motif AND
8 |         search_annotation for chromosome 22 )
9 |
10 |         parse_annotation
11 |
12 |         parse_features to get sizes of exons, look for small sizes
13 |     }
14 | }
15 |
16 | return accession numbers of records meeting the criteria

```

这个例子演示了如何使用这些子程序来回答一个问题，比如：在 22 号染色体上有哪

些基因包含特定的基序，并且有小的外显子？

10.4.3 主程序

让我们使用例 10.5来测试一下这些子程序吧，其中的一些子程序定义将会添加到 *BeginPerlBioinfo.pm* 模块里面：

例 10.5: GenBank 库的子程序

```

1  #!/usr/bin/perl
2  # Example 10-5 - test program of GenBank library subroutines)
3
4  use strict;
5  use warnings;
6
7  # Don't use BeginPerlBioinfo
8  # Since all subroutines defined in this file
9  # use BeginPerlBioinfo;      # see Chapter 6 about this module
10
11 # Declare and initialize variables
12 my $fh;      # variable to store filehandle
13 my $record;
14 my $dna;
15 my $annotation;
16 my $offset;
17 my $library = 'library.gb';
18
19 # Perform some standard subroutines for test
20 $fh = open_file($library);
21
22 $offset = tell($fh);
23
24 while ( $record = get_next_record($fh) ) {
25
26     ( $annotation, $dna ) = get_annotation_and_dna($record);
27
28     if ( search_sequence( $dna, 'AAA[CG].' ) ) {
29         print "Sequence found in record at offset $offset\n";
30     }
31     if ( search_annotation( $annotation, 'homo sapiens' ) ) {
32         print "Annotation found in record at offset $offset\n";
33     }
34
35     $offset = tell($fh);
36 }
37
38 exit;
39
40 #####

```

```
41 # Subroutines
42 #####
43
44 # open_file
45 #
46 #   - given filename, set filehandle
47
48 sub open_file {
49
50     my ($filename) = @_;
51     my $fh;
52
53     unless ( open( $fh, $filename ) ) {
54         print "Cannot open file $filename\n";
55         exit;
56     }
57     return $fh;
58 }
59
60 # get_next_record
61 #
62 #   - given GenBank record, get annotation and DNA
63
64 sub get_next_record {
65
66     my ($fh) = @_;
67
68     my ($offset);
69     my ($record) = '';
70     my ($save_input_separator) = $/;
71
72     $/ = "///\n";
73
74     $record = <$fh>;
75
76     $/ = $save_input_separator;
77
78     return $record;
79 }
80
81 # get_annotation_and_dna
82 #
83 #   - given filehandle to open GenBank library file, get next record
84
85 sub get_annotation_and_dna {
86
87     my ($record) = @_;
88
89     my ($annotation) = '';
```

```
90     my ($dna)           = '';
91
92     # Now separate the annotation from the sequence data
93     ( $annotation, $dna ) = ( $record =~ /^(\LOCUS.*ORIGIN\s*\n)(.*)\\/\n/s );
94
95     # clean the sequence of any whitespace or / characters
96     # (the / has to be written \/ in the character class, because
97     #   / is a metacharacter, so it must be "escaped" with \)
98     $dna =~ s/[\\s\\/]/g;
99
100     return ( $annotation, $dna );
101 }
102
103 # search_sequence
104 #
105 #   - search sequence with regular expression
106
107 sub search_sequence {
108
109     my ( $sequence, $regularexpression ) = @_ ;
110
111     my (@locations) = ();
112
113     while ( $sequence =~ /$regularexpression/ig ) {
114         push( @locations, pos );
115     }
116
117     return (@locations);
118 }
119
120 # search_annotation
121 #
122 #   - search annotation with regular expression
123
124 sub search_annotation {
125
126     my ( $annotation, $regularexpression ) = @_ ;
127
128     my (@locations) = ();
129
130     # note the /s modifier-. matches any character including newline
131     while ( $annotation =~ /$regularexpression/isg ) {
132         push( @locations, pos );
133     }
134
135     return (@locations);
136 }
```

对于我们小巧的 GenBank 库，例 10.5会生成以下输出：


```

1 | Sequence found in record at offset 0
2 | Annotation found in record at offset 0
3 | Sequence found in record at offset 6256
4 | Annotation found in record at offset 6256
5 | Sequence found in record at offset 12366
6 | Annotation found in record at offset 12366
7 | Sequence found in record at offset 17730
8 | Annotation found in record at offset 17730
9 | Sequence found in record at offset 22340
10 | Annotation found in record at offset 22340

```

tell 函数会报告到达文件当前读取位置的字节偏移量，所以你需要首先调用 *tell* 一次，然后读取记录得到从记录开头开始的正确的偏移量。

10.4.4 在顶层解析注释

现在，让我们来解析注释。

前面已经提到过，NCBI 上有一个文档，对 GenBank 记录结构的细节进行了描述。这个文件就是 *gbrel.txt*，它是 GenBank 发布的一部分，可以从 NCBI 网站或者它们的 FTP 站点上找到这个文件。每次发布的时候（现在是每两个月发布一个新版本），它都会更新，其中会注明格式发生的变化。如果你编程处理 GenBank 记录，你应该读读这个文档，并且在手边保留一份拷贝供参考用，还要定期去查看一下宣告的 GenBank 记录格式的变化。

如果你回头去看看本章前面部分那个完整的 GenBank 记录，你会发现注释部分有特定的结构。其中有一些字段，像是 LOCUS、DEFINITION、ACCESSION、VERSION、KEYWORDS、SOURCE、REFERENCE、FEATURES 和 BASE COUNT 等，这些字段都起始于一行的开头。有些字段还有子字段，尤其是 FEATURE 字段，它的结构异常复杂。

但是现在，我们只提取顶层的字段。你需要使用一个正则表达式，来匹配从一行开头的单词到另一行开头的别的单词之前的换行符之间的所有内容。

下面是匹配一个我们定义的字段的正则表达式：

```
1 | /^[A-Z].*\n(^\.s.*\n)*$/m
```

这个正则表达式什么含义？首先，它有 */m* 模式匹配修饰符，表示脱字符 *^* 和美元符号 *\$* 也可以匹配包含在内部的换行符附近的位置（而不仅仅匹配整个字符串的开头和结尾，这是它的默认行为）。

正则表达式的第一部分：

```
1 | ^[A-Z].*\n
```

匹配一行开头的大写字母，紧跟着任意数目的字符（换行符除外），最后是一个换行符。它对你试图匹配的字段的第一行进行了很好的描述。

正则表达式的第二部分：

```
1 | (^\.s.*\n)*
```

匹配一行开头的空格或制表符 `\s`，紧跟着任意数目的字符（换行符除外），最后是一个换行符。它用小括号括了起来，后面紧跟着使用了 `*`，这表示可以有 0 个或者多个这样的行。它匹配一个字段中的后续行，就是那些以空白开头的行。一个字段可能没有额外的这样的行，也有可能有多于一个这样的后续行。

所以，正则表达式的这两部分组合起来，匹配字段及其后面可选的附加行。

例 10.6 演示了一个子程序，对于给定的存储在标量变量中的 GenBank 记录的注释部分，它会返回一个散列，散列的键就是顶层的字段名，散列的值就是这些字段的具体内容。

例 10.6：解析 GenBank 注释

```

1  #!/usr/bin/perl
2  # Example 10-6 - test program for parse_annotation subroutine
3
4  use strict;
5  use warnings;
6  use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8  # Declare and initialize variables
9  my $fh;
10 my $record;
11 my $dna;
12 my $annotation;
13 my %fields;
14 my $library = 'library.gb';
15
16 # Open library and read a record
17 $fh = open_file($library);
18
19 $record = get_next_record($fh);
20
21 # Parse the sequence and annotation
22 ( $annotation, $dna ) = get_annotation_and_dna($record);
23
24 # Extract the fields of the annotation
25 %fields = parse_annotation($annotation);
26
27 # Print the fields
28 foreach my $key ( keys %fields ) {
29     print "***** $key *****\n";
30     print $fields{$key};
31 }
32
33 exit;
34
35 #####
36 # Subroutine
37 #####
38
39 # parse_annotation

```

```

40 #
41 #  given a GenBank annotation, returns a hash  with
42 #  keys: the field names
43 #  values: the fields
44
45 sub parse_annotation {
46
47     my ($annotation) = @_;
48     my (%results)     = ();
49
50     while ( $annotation =~ /^([A-Z].*\n(\s.*\n)*)/gm ) {
51         my $value = $&;
52         ( my $key = $value ) =~ s/^([A-Z]+).*/$1/s;
53         $results{$key} = $value;
54     }
55
56     return %results;
57 }

```

在子程序 *parse_annotation* 中，注意变量 *\$key* 和 *\$value* 是如何限制在 *while* 代码块作用范围内的。这种做法的一个好处就是，你不需要在每次进行循环的时候都对变量进行重新初始化。此外还要注意，散列的键是字段名，而散列的值则是整个字段的内容。

你可能需要花些时间来理解按键提取字段名的整个正则表达式：

```
1 | (my $key = $value) =~ s/^([A-Z]+).*/$1/s;
```

它首先把 *\$value* 的值赋给了 *\$key*。然后，它把 *\$key* 中的所有内容（注意针对嵌入换行符的 */s* 修饰符）都替换成了 *\$1*，它是一个特殊变量，保存的是第一对小括号之间的模式（*[A-Z]+*）。这个模式是一个或多个大写字母（锚定在字符串的开头，也就是字段名），所以它把 *\$key* 的值设置成了字段名中的第一个单词。

对于例 10.6，你会得到下面的输出（这次尝试仅仅获取了 GenBank 库的第一条记录）：

```

1 | ***** SOURCE *****
2 | SOURCE      Homo sapiens embryo male lung fibroblast cell_line:HuS-L12 cDNA to
3 |              mRNA.
4 |   ORGANISM  Homo sapiens
5 |              Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
6 |              Mammalia; Eutheria; Primates; Catarrhini; Hominidae; Homo.
7 | ***** DEFINITION *****
8 | DEFINITION  Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
9 |              complete cds.
10 | ***** KEYWORDS *****
11 | KEYWORDS    .
12 | ***** VERSION *****
13 | VERSION     AB031069.1  GI:8100074
14 | ***** FEATURES *****
15 | FEATURES    Location/Qualifiers

```

```

16      source      1..2487
17                  /organism="Homo sapiens"
18                  /db_xref="taxon:9606"
19                  /sex="male"
20                  /cell_line="HuS-L12"
21                  /cell_type="lung fibroblast"
22                  /dev_stage="embryo"
23      gene         229..2199
24                  /gene="PCCX1"
25      CDS          229..2199
26                  /gene="PCCX1"
27                  /note="a nuclear protein carrying a PHD finger and a CXXC
28                  domain"
29                  /codon_start=1
30                  /product="protein containing CXXC domain 1"
31                  /protein_id="BAA96307.1"
32                  /db_xref="GI:8100075"
33                  /translation="MEGDGSDPEPPDAGEDSKSENGENAPIYCICRKPDI
34                  NCNEWFHGD CIRITEKMAKAIREWYCRECREKDPKLEIRYRHKKSRE
35                  RDEGGGRKRPVDPDLQRRAGSGTGVGAMLARGSASPHKSSPQPLVATPSQ
36                  QHHQQQQQ QIKRSARMCGECEACRRTEDCGHCDFCRDMKKFGGPNKIRQK
37                  CRLRQCQLRARES YKYFPSSLSPVTPSESLPRRRPLPTQQQPQPSQKLGR
38                  IREDEGAVASSTVKEPPEATATP EPLSDEDLPDLYQDFCAGAFDDHGLP
39                  WMSDTEESPFLDPALRKRAVKVKHVKRRE KKSEKKKEERYKRHRQKQKH
40                  KDKWKHPERADAKDPASLPQCLGPGCVRPAQPSSKYCS DDCGMKLAANRI
41                  YEILPQRIQQWQQSPCIAEEHGKLLERIRREQQSARTRLQEMERR FHE
42                  LEAII LRKQQAVREDEESNEGDSDDTDLQIFCVSCGHPINPRVALRHMER
43                  CYAK YESQTSFGSMYPTRIEGATRLFCDVYNPQSKTYCKRLQVLCPEHSR
44                  DPKVPADEVCGC PLVRDVFELTGDFCRLPKRQCNRHWCWEKLRRAEVDLER
45                  VRVWYKLDLFEQERNVRT AMTNRAGLLALMLHQTIQHDLPTTDLRSSADR"
46 ***** REFERENCE *****
47 REFERENCE      2 (bases 1 to 2487)
48 AUTHORS       Fujino,T., Hasegawa,M., Shibata,S., Kishimoto,T., Imai,S.
49               and
50               Takano,T.
51 TITLE         Direct Submission
52 JOURNAL        Submitted (15-AUG-1999) to the DDBJ/EMBL/GenBank databases.
53               Tadahi-ro Fujino, Keio University School of Medicine, Department of
54               Microbiology; Shinanomachi 35, Shinjuku-ku, Tokyo 160-8582, Japan
55               (E-mail:fujino@microb.med.keio.ac.jp,
56               Tel:+81-3-3353-1211(ex.62692), Fax:+81-3-5360-1508)
57 ***** ACCESSION *****
58 ACCESSION      AB031069
59 ***** LOCUS *****
60 LOCUS          AB031069      2487 bp      mRNA          PRI          27-MAY-2000
61 ***** ORIGIN *****
62 ORIGIN
63 ***** BASE *****
64 BASE COUNT      564 a      715 c      768 g      440 t

```

如你所见,这种方法完全可行,除了阅读正则表达式比较困难以外(随着练习的增多

这也会越来越容易），整个代码都简单明了，就是几个简短的子程序而已。

10.4.5 解析 FEATURES 表

让我们更进一步，解析一下下一个层面的 FEATURES 表，它由 *source*、*gene* 和 *CDS* 这几个特征键 (*features keys*) 组成。（参看本节后面的更加全面的特征键列表。）在本章末尾的练习题中，你将会看到进一步深入 FEATURES 表的挑战。

要研究 FEATURES 表，你应该首先去看一下前面提到的 NCBI 中的 *gbrel.txt* 文档。然后你要再好好研究一下 FEATURES 表的更加详尽的文档，在 <http://www.ncbi.nlm.nih.gov/collab/FT/index.html> 上可以找到它。

特征

尽管我们的 GenBank 条目非常简单，只包含三个特征，实际上特征非常多。注意解析代码会把它们全部找到，因为代码只是处理文档的结构，并不针对特定的特征。

下面是 GenBank 记录中定义的特征的一个列表。尽管非常长，但我认为最好还是通读一下，对可能会出现在 GenBank 记录中的信息有一个大体的了解。

allele

等位基因，废弃；参看变异 (*variation*) 特征键

attenuator

弱化子，和转录终止相关的序列

C_region

C-免疫特征区

CAAT_signal

真核生物启动子区的 CAAT 盒 CAAT box in eukaryotic promoters

CDS

编码蛋白质中氨基酸的序列（包括终止密码子）

conflict

不同测定结果所得差异序列

D-loop

置换环³

D_segment

D-免疫特征区⁴

enhancer

增强启动子功能的顺式作用增强子

exon

外显子，编码剪接 mRNA 部分的区域

gene

基因，确定一个功能性基因的区域，可能包括上游（启动子、增强子，等）和下游的调控元件，每一个都有特定的名字

GC_signal

³译者注：指 DNA 双链的局部，由具有互补性单链 DNA 与之结合所产生的环状结构。

⁴译者注：免疫功能中免疫球蛋白重链的多变区

	真核生物启动子中的 GC 盒
<i>iDNA</i>	重组引入的插入 DNA
<i>intron</i>	内含子，被 mRNA 剪接切除的转录区域
<i>J_region</i>	J-免疫特征区 ⁵
<i>LTR</i>	长末端重复序列 (Long terminal repeat)
<i>mat_peptide</i>	成熟肽编码区域 (不包括终止密码子)
<i>misc_binding</i>	其他结合位点
<i>misc_difference</i>	其他特征区
<i>misc_feature</i>	无法用任何其他特征描述的重要生物功能区
<i>misc_recomb</i>	其他重组特征区
<i>misc_RNA</i>	不能用其他 RNA 名字定义的转录特征区
<i>misc_signal</i>	其他信号区
<i>misc_structure</i>	其他 DNA 或 RNA 结构
<i>modified_base</i>	被修饰的核苷酸碱基
<i>mRNA</i>	信使 RNA (Messenger RNA)
<i>mutation</i>	突变，废弃；参看变异 (variation) 特征键
<i>N_region</i>	N-免疫特征区
<i>old_sequence</i>	修订自旧版本的序列
<i>polyA_signal</i>	polyA 信号，剪切的多聚腺苷酸信号
<i>polyA_site</i>	polyA 位点，mRNA 的多聚腺苷酸添加位点
<i>precursor_RNA</i>	还不是成熟 RNA 产物的前体 RNA
<i>prim_transcript</i>	

⁵译者注：免疫功能中的连接区，位于免疫球蛋白等分子的 V 区与 C 区之间。

<i>primer</i>	初始（未加工的）转录本
<i>primer_bind</i>	PCR 中使用的引物结合区域
<i>promoter</i>	启动子，转录起始的区域
<i>protein_bind</i>	蛋白质结合在 DNA 或 RNA 上的非共价结合位点
<i>RBS</i>	核糖体结合位点
<i>rep_origin</i>	双链 DNA 的复制起始区
<i>repeat_region</i>	包含重复的子序列的重复序列
<i>repeat_unit</i>	重复序列区域的单个重复单元
<i>rRNA</i>	核糖体 RNA（Ribosomal RNA）
<i>S_region</i>	S-免疫特征区
<i>satellite</i>	卫星 DNA 重复序列
<i>scRNA</i>	小胞浆 RNA
<i>sig_peptide</i>	信号肽编码区域
<i>snRNA</i>	小核 RNA
<i>source</i>	一个 GenBank 记录代表的序列数据的生物学来源；每个记录都有一个或多个必须的特征；对于那些已经被收录进 NCBI 分类学数据库的生物，会有一个与之相关的 <code>/db_xref="taxon:NNNN"</code> 分类号（其中的 NNNNN 就是 NCBI 分类学数据库中为该生物分配的数字识别号）
<i>stem_loop</i>	DNA 或 RNA 中的发卡环结构
<i>STS</i>	序列标签位点（Sequence Tagged Site）：操作上和 PCR 实验中使用的引物结合的唯一序列
<i>TATA_signal</i>	真核生物启动子中的 TATA 盒
<i>terminator</i>	终止子，导致转录终止的序列

transit_peptide

转运肽编码区域

transposon

转座子元件 (Transposable element, TN)

tRNA

转运 (Transfer RNA)

unsure

作者不确定该区域中的序列

V_region

V-免疫特征区

variation

变异, 一个相关的群体包含稳定的突变

-

占位符 (连字符)

-10_signal

原核生物启动子中的 Pribnow 盒

-35_signal

原核生物启动子中的-35 盒

3' clip

转录本前体在加工过程中被切除的 3' 端区域

3' UTR

3' 端非翻译区 (untranslated region) (后缀)

5' clip

转录本前体在加工过程中被切除的 5' 端区域

5' UTR

5' 端非翻译区 (untranslated region) (先导)

这些特征键都可以有它们自己的附加特征, 在这里以及后面的联系中你会看到的。

解析

例 10.7 寻找出现的特征, 并返回用它们填充的数组。它并不会去寻找上一小节中列出的全部特征, 它只寻找 GenBank 记录中出现的那些特征, 返回它们以备后续之用。

比较常见的一种情况是, 在一个记录中有多个同样的特征。比如, 在一个 GenBank 记录的 FEATURES 表中可能会有好几个外显子。正因为如此, 我们把特征作为元素存储到一个数组中, 而不是以特征名作为键存储到一个散列中 (它只允许你存储一个, 比如, 仅仅一个外显子而已)。

例 10.7: 测试解析特征子程序

```

1 |#!/usr/bin/perl
2 |# - main program to test parse_features
3 |
4 |use strict;
5 |use warnings;
6 |use BeginPerlBioinfo;    # see Chapter 6 about this module

```



```

7
8 # Declare and initialize variables
9 my $fh;
10 my $record;
11 my $dna;
12 my $annotation;
13 my %fields;
14 my @features;
15 my $library = 'library.gb';
16
17 # Get the fields from the first GenBank record in a library
18 $fh = open_file($library);
19
20 $record = get_next_record($fh);
21
22 ( $annotation, $dna ) = get_annotation_and_dna($record);
23
24 %fields = parse_annotation($annotation);
25
26 # Extract the features from the FEATURES table
27 @features = parse_features( $fields{'FEATURES'} );
28
29 # Print out the features
30 foreach my $feature ( @features ) {
31
32     # extract the name of the feature (or "feature key")
33     my ($featurename) = ( $feature =~ /^ {5}(\S+)/ );
34
35     print "***** $featurename *****\n";
36     print $feature;
37 }
38
39 exit;
40
41 #####
42 # Subroutine
43 #####
44
45 # parse_features
46 #
47 # extract the features from the FEATURES field of a GenBank record
48
49 sub parse_features {
50
51     my ($features) = @_;    # entire FEATURES field in a scalar variable
52
53     # Declare and initialize variables
54     my (@features) = ();    # used to store the individual features
55

```

```

56 | # Extract the features
57 | while ( $features =~ /^ {5}\S.*\n(^ {21}\S.*\n)*/gm ) {
58 |     my $feature = $&;
59 |     push( @features, $feature );
60 | }
61 |
62 | return @features;
63 | }

```

例 10.7会给出下面的输出：

```

1 | ***** source *****
2 |     source          1..2487
3 |                     /organism="Homo sapiens"
4 |                     /db_xref="taxon:9606"
5 |                     /sex="male"
6 |                     /cell_line="HuS-L12"
7 |                     /cell_type="lung fibroblast"
8 |                     /dev_stage="embryo"
9 | ***** gene *****
10 |     gene            229..2199
11 |                     /gene="PCCX1"
12 | ***** CDS *****
13 |     CDS             229..2199
14 |                     /gene="PCCX1"
15 |                     /note="a nuclear protein carrying a PHD finger and a CXXC
16 |                     domain"
17 |                     /codon_start=1
18 |                     /product="protein containing CXXC domain 1"
19 |                     /protein_id="BAA96307.1"
20 |                     /db_xref="GI:8100075"
21 |                     /translation="MEGDGSDPEPPDAGEDSKSENGENAPIYICICRKPDI
22 |                     NCNEWFHGD CIRITEKMAKAIREWYCRECREKDPKLEIRYRHKKS
23 |                     RDEGGGRKRPVDPDLQRRAGSGTGVGAMLARGSASPHKSSPQLVATPSQHH
24 |                     QIKRSARMCGECEACRRTEDCGHCDFCRDMKKFGGPNKIRQKCLRQCLRARE
25 |                     SPSSLSPVTPSESLPRPRRPLPTQQQPQPSQKLGRIREDEGAVASSTVKEP
26 |                     EPLSDEDLPLDPDLYQDFCAGAFDDHGLPWMSDTEESPFLDPALRKRAVKV
27 |                     KKSEKKKEERYKRHRQKQKHDKWKHPPERADAKDPASLPQCLGPGCVRPA
28 |                     DDCGMKLAANRIYEILPQRIQQWQQSPCIAEEHGKKLLERIRREQQSARTR
29 |                     FHELEAIIILRAKQQA VREDEESNEGDSDDTDLQIFCVSCGHPINPRVAL
30 |                     YESQTSFGSMYPTRIEGATRLFCDVYNPQSKTYCKRLQVLCPEHSRDPKVP
31 |                     PLVRDVFELTGDFCRLPKRQCNRHYCWEKLRRAEVDLERVVRVWYKLDELFE
32 |                     AMTNRAGLLALMLHQTIQHDLTTDLRSSADR

```

在例 10.7的子程序 `parse_features` 中，提取特征的正则表达式，和例 10.6中使用的解析顶层注释所使用的正则表达式非常相似。让我看一下例 10.7中最基本的解析代码：

```

1 | while( $features =~ /^ {5}\S.*\n(^ {21}\S.*\n)*/gm ) {

```

从整体上看，简单来说，这个正则表达式寻找特定格式的特征，第一行以 5 个空白起始，后面是可有可无的以 21 个空白起始的行。

首先，注意模式修饰符 `/m` 使得 `^` 元字符可以匹配嵌入换行符后面的位置。此外 `{5}` 和 `{21}` 是量词，指定前面的项目必须出现正好 5 次和 21 次，在这两个例子中前面的项目都是一个空白。

根据特征的第一行和可选的后续行，正则表达式分为两部分。第一部分 `^ {5} \S . * \n` 表示一行的开头 (`^`) 有 5 个 (`{5}`) 空白，后面紧跟着的是一个非空白字符 (`\S`)，之后是任意数目的非换行符字符 (`. *`)，最后是一个换行符 (`\n`)。正则表达式的第二部分 (`^ {21} \S . * \n`)`*`，表示一行的开头 (`^`) 有 21 个 (`{21}`) 空白，之后是任意数目的非换行符字符 (`. *`)，最后是一个换行符 (`\n`)；并且这样的行可以有 0 个或者多个，这是用包裹住整个表达式的 (`() *`) 来表明。

主程序中也有一个简短的正则表达式处理类似的行，来从特征中提取出特征名（也叫特征键）。

所以，又一次成功了。FEATURES 表现在被详细地分解或说“解析”了，一直到了能把各个特征分割开的水平。解析 FEATURES 表的下一步就是从每一个特征中提取出具体的信息了。这包括定位（和特征名在同一行上，也可能在其他行上）；用正斜杠表明的限定词，包括一个限定词名称，如何合适的话，还有一个等号和各种各样的附加信息，附加信息可能持续多行。

我会把这最后的一步作为练习。这是对我们用来解析特征的方法的一个理所当然的扩充。在尝试从一个特征中解析定位和限定词之前，你可能想去参考一下 NCBI 网站上关于 FEATURES 表结构的完整细节的文档。

我用来解析 FEATURES 表的方法保留了信息的结构。然后，有时，你可能只想看看像是“endonuclease”这样的某些单词是否在记录中的某个地方出现了。如果是这样，回忆一下你在例 10.5 中创建的 `search_annotation` 子程序，它在整个注释中查找正则表达式。在多数情况下，这就是你真正需要的。然而，就想你刚刚看到的那样，当你真的需要对 FEATURES 表进行深入解析的时候，Perl 有它独特的特性，可以让这个工作变得可行甚至非常简单。

10.5 使用 DBM 对 GenBank 进行索引

DBM 表示数据库管理 (Database Management)。Perl 提供了内置函数，让 Perl 程序员可以访问 DBM 文件。

10.5.1 DBM 基础

当你打开一个 DBM 文件时，你就像使用一个散列那样访问它：你给它键，它返回值，并且你可以添加和删除键-值对。DBM 之所以有用，是因为它把键-值数据保存在了你计算机中的永久硬盘上。它可以在你运行程序的间歇保存信息，也可以作为在需要同样数据的不同程序间共享信息的一种方法。在吃尽计算机内容之前，一个 DBM 文件可能会变得非常大，这样它会使你的程序以及其他所有东西慢得像乌龟爬一样。

有两个函数把一个散列“绑定”到一个 DBM 文件上，它们就是 *dbmopen* 和 *dbmclose*，绑定之后你就使用散列就行了。正如你已经看到的那样，使用散列的话，就像它的定义那样，查找会非常容易。对一个叫做 `%my_hash` 的散列，你键入 `keys %my_hash` 就可以得到这个散列的所有键了。之后，你键入 `values %my_hash` 就可以得到所有的只。对于大的 DBM 文件来说，你可能不会想这么做。Perl 函数 *each* 允许你一次读取一个键-值对，这样就能节省你运行程序的内存了。也有一个 *delete* 函数可以删除键的定义：

```
1 | delete $my_hash{'DNA'}
```

它会把键从散列中完全删除掉。

DBM 文件是一个非常简单的数据库。它们没有 *MySQL*、*Oracle* 或者 *PostgreSQL* 这些关系数据库强大，然而，通常对于一个问题来说，它就是我们真正需要的，而这样简单的数据库表现也异常出色。当你有一个键-值数据集（或者多个这样的数据集）时，考虑使用 DBM 吧。对于 Perl 来说，它真的非常容易使用。

使用 DBM 主要的问题在于，有多种稍有不同的 DBM 实现——*NDBM*、*GDBM*、*SDBM* 和 *Berkeley DB*。它们之间的区别很小，但确实存在。但对于绝大多数的目的来说，这些实现是完全通用的。新版本的 Perl 默认使用 *Berkeley DB*，如果你想的话，对于你的 Perl 来说它非常容易获得并安装上。如果你真的不需要长的键或者值，这并不是一个问题。一些老的 DBMs 需要你给键添加空字节 (null bytes) 并且从值中删除它们。

```
1 | $value = $my_hash{"$key\0"};  
2 | chop $value;
```

如果你不需要这样做，那就再好不过了。*Berkeley DB* 可以很好得处理长字符串（其他的一些 DBM 实现有一定的限制）。因为在生物学中，你可能会有一些长的字符串，所以如果你没有 *Berkeley DB* 的话，我推荐你安装上。

10.5.2 一个用于 GenBank 的 DBM 数据库

你已经看到了，如何从一个 GenBank 记录或者 GenBank 记录库中提取信息。你刚刚看到了在程序运行时 DBM 文件是如何把你的散列数据保存到你的硬盘上的。你也看到了使用 *tell* 和 *seek* 来快速访问一个文件中的某个位置。

现在，我们把这三个想法组合起来，使用 DBM 来构建一个关于 GenBank 库信息的数据库。在一定程度上这非常简单：你提取出索引号作为键，把 GenBank 库中记录的字节偏移量存储为值。你要添加一些代码，对于给定的一个库和某个偏移量，返回在那个偏移量处的记录，并且编写主程序，允许用户使用索引号交互式得获取 GenBank 记录。完成后，如果给它一个索引号，你的程序应该会非常快得返回一个 GenBank 记录。

此处基本的想法会在本章末尾的练习题中进行扩展，扩展到一个相当大的程度。你现在在可能会想先去看一下，这样你对我现在介绍的这种技术的强大就有一定的了解了。

为了避免后面过度的杂乱，现在先给出打开（如果需要会先创建）一个 DBM 文件的代码片段：

```
1 unless(dbmopen(%my_hash, 'DBNAME', 0644)) {
2
3     print "Cannot open DBM file DBNAME with mode 0644\n";
4     exit;
5 }
```

`%my_hash` 就像 Perl 的其他散列一样，但是通过这个语句，它会被绑定到 DBM 文件上。`DBNAME` 是将要实际创建的 DBM 文件的基名。某些 DBM 版本会创建一个完全叫做这个名字的文件，其他的则会创建使用文件扩展名`.dir`和`.pag`的两个文件。

另一个参数叫做模式（*mode*）。Unix 或者 Linux 用户对于使用这种形式的文件权限会非常熟悉。有多种可能存在，下面是最常见的几个：

0644

你可以读取并写入，其他人只能读取。

0600

你有你自己可以读取或写入。

0666

任何人都可以读取或写入。

0444

任何人都可以读取（但是没人能够写入）。

0400

只有你可以读取（其他人都不能对它进行任何操作）。

当 DBM 文件创建的时候它会被授予一定的权限，如果你尝试使用更多的权限去打开它，`dbmopen` 函数的调用会失败。通常，如果只有所有者被允许写入的话，所有者会对文件使用 0644 模式，而读取者则会使用 0444 模式。如果想要让任何人都可以读取或者写入文件，所有者会赋予它 0666 模式。

基本上就这些，DBM 文件就是这么简单。例 10.8 演示了一个 DBM 文件，它存储的键-值对中，键是 GenBank 记录的索引号，值是记录的字节偏移量。

例 10.8：一个 GenBank 库的 DBM 索引

```
1 #!/usr/bin/perl
2 # Example 10-8 - make a DBM index of a GenBank library,
3 #     and demonstrate its use interactively
4
5 use strict;
6 use warnings;
```

```
7 use BeginPerlBioinfo;    # see Chapter 6 about this module
8
9 # Declare and initialize variables
10 my $fh;
11 my $record;
12 my $dna;
13 my $annotation;
14 my %fields;
15 my %dbm;
16 my $answer;
17 my $offset;
18 my $library = 'library.gb';
19
20 # open DBM file, creating if necessary
21 unless ( dbmopen( %dbm, 'GB', 0644 ) ) {
22     print "Cannot open DBM file GB with mode 0644\n";
23     exit;
24 }
25
26 # Parse GenBank library, saving accession number and offset in DBM file
27 $fh = open_file($library);
28
29 $offset = tell($fh);
30
31 while ( $record = get_next_record($fh) ) {
32
33     # Get accession field for this record.
34     ( $annotation, $dna ) = get_annotation_and_dna($record);
35
36     %fields = parse_annotation($annotation);
37
38     my $accession = $fields{'ACCESSION'};
39
40     # extract just the accession number from the accession field
41     # -remove any trailing spaces
42     $accession =~ s/^ACCESSION\s*//;
43
44     $accession =~ s/\s*$//;
45
46     # store the key/value of accession/offset
47     $dbm{$accession} = $offset;
48
49     # get offset for next record
50     $offset = tell($fh);
51 }
52
53 # Now interactively query the DBM database with accession numbers
54 # to see associated records
55
```

```
56 | print "Here are the available accession numbers:\n";
57 |
58 | print join( "\n", keys %dbm ), "\n";
59 |
60 | print "Enter accession number (or quit): ";
61 |
62 | while ( $answer = <STDIN> ) {
63 |     chomp $answer;
64 |     if ( $answer =~ /^\\s*q/ ) {
65 |         last;
66 |     }
67 |     $offset = $dbm{$answer};
68 |
69 |     if ( defined $offset ) {
70 |         seek( $fh, $offset, 0 );
71 |         $record = get_next_record($fh);
72 |         print $record;
73 |     }
74 |     else {
75 |         print "Do not have an entry for accession number $answer\n";
76 |     }
77 |
78 |     print "\nEnter accession number (or quit): ";
79 | }
80 |
81 | dbmclose(%dbm);
82 |
83 | close($fh);
84 |
85 | exit;
```

下面是例 10.8 截断的输出：

```
1 | Here are the available accession numbers:
2 | XM_006271
3 | NM_021964
4 | XM_009873
5 | AB031069
6 | XM_006269
7 | Enter accession number (or quit): NM_021964
8 | LOCUS      NM_021964    3032 bp    mRNA          PRI      14-MAR-2001
9 | DEFINITION  Homo sapiens zinc finger protein 148 (pHZ-52) (ZNF148), mRNA.
10 | ...
11 | //
12 |
13 | Enter accession number (or quit): q
```

10.6 练习题

习题 10.1

去逛逛 NCBI、EMBL 和 EBI 的网站，熟悉一下它们的使用。

习题 10.2

阅读 GenBank 格式的文档 *gbrel.txt*。

习题 10.3

编写一个子程序，通过值传递一个散列。现在重写它，通过指针传递散列。

习题 10.4

设计由几个子程序构成的模块，来处理下面这些类型的数据：一个包含记录的平面文件，记录由位于一行的基因名和位于后续行的各种附加信息组成，最后是一个空白行。你的子程序应该能够读取数据，之后对和一个基因名相关的信息进行快速查询。你还应该能够添加新的记录到这个平面文件中。现在重用这些模块，来构建一个地址簿程序。

习题 10.5

进一步深入 FEATURES 表。解析表中特征的下一层信息，主要是特征名、定位和限定词这几个特征。对于字段结构的定义，可以查阅文档 *gbrel.txt*。

习题 10.6

编写一个程序，以一个长的 DNA 序列作为输入，输出以频率进行排序的所有四碱基子序列（一共有 256 个）的计数。一个四碱基子序列可以起始于 1、2、3 等各个位置。（这种类型的词频分析在许多研究领域都非常常见，包括语言学、计算机科学和音乐学。）

习题 10.7

扩展习题 10.6 中的程序，让它可以对一个 GenBank 库中的所有序列进行计数。

习题 10.8

对于给定的一个氨基酸，找到一个 DNA 序列或者一个 GenBank 库中它临近氨基酸的出现频率。

习题 10.9

从 GenBank 记录库的注释中提取出所有的单词（除了“the”或者其他无用的类似单词以外）。对于找到的每一个单词，都把库中 GenBank 记录的偏移量添加到 DBM 文件中，DBM 文件的键就是单词，值使用空格分隔开的偏移量字符串。换言之，一个键对应的值可以用空格分隔开的偏移量列表。然后，通过一个简单的查找，你就可以露艾苏找到含有像“fibroblast”这样的单词的所有记录了，之后，提取出这些偏移量，使用它们在库中进行找寻（seek）。与 GenBank 库先比，你的 DBM 文件有多大？如果要针对所有 GenBank 中的注释构建一个搜索引擎，又该如何呢？如果是仅仅针对人类的 DNA 呢？

习题 10.10

编写一个程序，从 GenBank 的 GBPRI 分类中构建一个个性化的肿瘤基因库。

第 11 章 PDB

目录

11.1 PDB 概述	250
11.2 文件和文件夹	251
11.3 PDB 文件	259
11.4 解析 PDB 文件	269
11.5 控制其他程序	279
11.6 练习题	284

人类基因组计划 (Human Genome Project) 在解码人类基因 DNA 序列上的成功, 已经俘获了公众的想象力, 但另一个项目却没有获得如此的关注度, 它也会得到同样具有变革性的结果。这个项目是进行国际性的合作努力, 使用高通量分析技术, 在基因组范围水平上确定大量蛋白质的 3D 结构。国际性的合作是结构基因组学这个新兴领域的基础。

最近技术上的突破, 促进了确定蛋白质结构这场竞赛的加速。存储所有这些数据的商店就是 PDB (Protein Data Bank), 可以在网站 <http://www.rcsb.org/pdb/> 上找到它。

找到氨基酸序列也就是一级序列, 仅仅是蛋白质研究的开始。蛋白质会进行局部折叠, 折叠成 α -螺旋、 β -折叠和 β -转角之类的二级结构。两个或三个临近的二级结构可能会组合成叫做“基元”或者“超二级结构”的常见折叠花式, 比如 β -片层或者 α - α 螺旋组合单元。这些建筑模块会进一步折叠成蛋白质的 3D 或者三级结构。最后, 一个或者多个三级结构可能作为亚单元组合成酶或者病毒的四级结构。

在不知道一个蛋白质是如何折叠成一个 3D 结构之前, 你很难直到这个蛋白质到底有什么功能, 以及它是如何发挥功能的。即使你知道这个蛋白质在疾病中发挥作用, 要想找到一个可能的治疗方案, 通常还是需要知道它的三级结构。了解蛋白质的三级结构和它的活性位点 (它可能会包含那些在一级结构上相距甚远、但是在蛋白质折叠后却紧邻在一起的氨基酸), 对于筛选新的药物靶点是至关重要的。

现在, 包括人类在内的不少生物的基本遗传信息已经被解码了, 接下来生物学家面临的主要挑战就是尽可能多的去了解这些基因编码的蛋白质, 以及它们之间的相互作用。

事实上, 现代生物学主要的问题之一就是蛋白质的一级氨基酸序列是如何决定它最终的 3D 结构的。如果能够找到一种计算方法, 可以从蛋白质的氨基酸序列可靠地预测出它的折叠, 那么生物学和医学的作用将是意义深远的。

在本章中, 你将学习 PDB 文件的基础, 以及如何从中解析出需要的信息。你将会探索更加有趣的 Perl 技术, 寻找大量的文件并进行循环处理, 以及在 Perl 程序中控制其他的生物信息学程序。本章末尾的练习题, 在此处介绍的基本知识的基础上, 让你挑战获取 PDB 数据的更多信息。

11.1 PDB 概述

大分子（包括蛋白质、肽、病毒、蛋白质和核酸的复合物、核酸、以及碳水化合物）3D 结构信息的主要资源就是 PDB。并且，它的格式实际上就是交换结构信息的标准格式。PDB 中的大多数结构都是通过 X 射线衍射（X-ray）diffraction 和核磁共振（NMR, nuclear magnetic resonance）实验确定的。

1971 年 PDB 诞生之初，只有七个蛋白质，随后迅速增长到了 20,000 个结构。随着结构基因组学中国际性合作的增加，PDB 仍然在延续它快速增长的态势。在短短几年的时间内，已知结构的数目将会接近 100,000。

PDB 文件和 GenBank 记录相似，都是人类可读的 ASCII 平面文件。文件中的文本符合特定的格式，所以可以编写计算机程序从中提取信息。和 GenBank 在一个“库”文件中保存许多记录不一样，PDB 对于每一个结构都用一个文件来存储。

经常处理 PDB 文件的生物信息学家，会抱怨 PDB 格式在一致性上存在着严重的问题。比如，为了满足新知识的需要，随着字段和数据格式的不断变革，有些旧的文件就过时了。现在保持 PDB 数据一致性的工作正在进行中，直到这些工作完成、开发出一个新的数据格式之前，现有数据格式的不一致性仍然是程序员必须面对的一项挑战。如果你对 PDB 文件进行大量的编程，你会发现数据中有许多的不一致甚至错误，尤其是那些老的文件中。另外，许多能够成功解析新文件的工具，在老的文件上就不一定好用了。

随着你逐渐称为一个经验更加丰富的程序员，你面对的 PDB 这样那样的问题就显得更加重要的。比如，随着 PDB 的发展，你编写的和它进行交互的代码也要不断改进。你必须要时刻关注世界的变化，注意维护你的代码，让它与时俱进。随着数据库间链接得到了更好的支持，你的代码应该充分利用这些链接提供的新的机遇。数据存储的新标准正在建立中，你的代码也要更新把它们包含进去。

PDB 网站上包含了大量关于如果下载搜有文件的信息。它们也以便于获取的免费的 CD 集进行发布，这对于那些没有高速网路连接的人来说是一个巨大的便利。

11.2 文件和文件夹

PDB 以目录中的文件形式进行发布。每一个蛋白质结构都有自己单独的文件。PDB 包含大量的数据，所以要处理它是一种挑战。在本节中，你将学习处理组织在目录和子目录中的大量文件。

你会发现，常常需要编写程序来操作大量的文件。比如，你可能会把多次的测序放在一个目录中，根据测序上机的日期分成子目录，把测序仪产生的数据放在对应日期的子目录中。在短短几年之后，你可能就会有相当数量的文件。

然后，有一天，你发现了一个新的 DNA 序列，看起来参与细胞分裂。你进行了一个 BLAST 搜索（参看第 12 章），但是没有找到对于新 DNA 的显著结果。这个时候，你想直到在以前的测序中是不是看到过这个 DNA。¹你需要做的就是，针对多种多样的测序子目录中的成百上千的文件，运行比对的子程序。但这可能会重复耗费好几天，对于坐在计算机屏幕前的你来说这绝对是无聊透顶的工作。

你可以编写一个程序，在很短时间内完成这个工作！然后你需要做的就是回到座位上，检查一下结果中你的程序有没有找到显著的匹配。然而要编写这样的程序，你需要知道如何在 Perl 中操作所有的文件和文件夹。接下来的小节将向你演示如何去做。

11.2.1 打开目录

文件系统以树状结构进行组织。这个比喻是非常贴切的。从树的任何地方开始，你可以沿着树干，得到源于从你开始之处的任何树叶。如果你从树根开始，你可以得到所有的树叶。类似的，在文件系统中，如果你从一个特定的目录开始，你可以得到源于你开始之处的子目录中的所有文件。如果你从文件系统的根（非常奇怪，它也被叫做“顶”）开始，你可以得到所有的文件。

在打开、读取、写入和关闭文件方面，你已经练习了很多。我将演示一个简单的方法，让你可以打开一个文件夹（也叫做目录），获取这个文件夹中所有文件的文件名。在那之后，你将看到如何从一个特定的地方开始，获取所有目录和子目录中的所有文件的名称。

从一些伪代码开始，让我们看一下列出文件夹中所有文件的 Perl 的方式：

```
1 | open folder
2 |
3 | read contents of folder (files and subfolders)
4 |
5 | print their names
```

例 11.1 演示了真实的 Perl 代码。

例 11.1：列出文件夹（或目录）的内容

```
1 | #!/usr/bin/perl
2 | # Example 11-1   Demonstrating how to open a folder and list its contents
3 |
4 | use strict;
5 | use warnings;
```

¹你可能会把所有的测序结构保存为一个大的 BLAST 库进行比较；可以使用本节介绍的技术构建这样一个 BLAST 库。

```

6 use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8 my @files = ();
9 my $folder = 'pdb';
10
11 # open the folder
12 unless ( opendir( FOLDER, $folder ) ) {
13     print "Cannot open folder $folder!\n";
14     exit;
15 }
16
17 # read the contents of the folder (i.e. the files and subfolders)
18 @files = readdir(FOLDER);
19
20 # close the folder
21 closedir(FOLDER);
22
23 # print them out, one per line
24 print join( "\n", @files ), "\n";
25
26 exit;

```

因为你在一个包含 PDB 文件的文件夹中运行这个程序，所以你将会看到：

```

1 | .
2 | ..
3 | 3c
4 | 44
5 | pdb1a4o.ent

```

如果你想列出当前目录中的文件，你可以把代表当前目录的“.”这个特殊名字赋值给目录名，就像这样：

```

1 | my $folder = '.';

```

在 Unix 或者 Linux 系统中，特殊文件“.”和“..”分别代表当前目录和父目录。它们并不是“真实”的文件，至少不是你想读取的文件。使用优秀的、了不起的 *grep* 函数，你可以避免把它们罗列出来。*grep* 允许你根据测试选择数组中的元素，比如一个正则表达式。下面演示的是如何把数组中的“.”和“..”过滤掉：

```

1 | @files = grep( !/^\.\.?$/, @files );

```

因为感叹号这个取反操作符，*grep* 选择了不匹配正则表达式的所有行。正则表达式 `/^\.\.?$/` 寻找这样的行，它以点号 `.`（因为点号是一个元字符，所以用反斜杠进行了转义）起始（一行的开头用 `^` 元字符表示），后面跟着 0 个或者 1 个点号 `\.?`（`?` 匹配前面的元素 0 次或者 1 次），再后面就没有其他任何东西了（`$` 元字符表示字符串的结尾）。

实际上，当读取一个目录的时候，这非常常用，所以通常把它们组合到一步中：

```
1 | @files = grep ( !/^\.\.?$/, readdir(FOLDER));
```

好了，现在所有的文件都罗列出来了。但是稍等：如果这些文件不是常规文件而是子文件夹呢？你可以使用便捷的文件测试操作符来检测每一个文件名，这样就可以打开每一个子文件夹把其中的文件罗列出来了。首先是一些伪代码：

```
1 | open folder
2 |
3 | for each item in the folder
4 |
5 |     if it's a file
6 |         print its name
7 |
8 |     else if it's a folder
9 |         open the folder
10 |        print the names of the contents of the folder
11 |    }
12 | }
```

例 11.2演示了这个程序。

例 11.2：列出文件夹及其子文件夹的内容

```
1 | #!/usr/bin/perl
2 | # Example 11-2   Demonstrating how to open a folder and list its contents
3 | #   -distinguishing between files and subfolders, which
4 | #       are themselves listed
5 |
6 | use strict;
7 | use warnings;
8 | use BeginPerlBioinfo;    # see Chapter 6 about this module
9 |
10 | my @files = ();
11 | my $folder = 'pdb';
12 |
13 | # Open the folder
14 | unless ( opendir( FOLDER, $folder ) ) {
15 |     print "Cannot open folder $folder!\n";
16 |     exit;
17 | }
18 |
19 | # Read the folder, ignoring special entries "." and ".."
20 | @files = grep ( !/^\.\.?$/, readdir(FOLDER) );
21 |
22 | closedir(FOLDER);
23 |
24 | # If file, print its name
25 | # If folder, print its name and contents
26 | #
27 | # Notice that we need to prepend the folder name!
```

```
28 | foreach my $file (@files) {
29 |
30 |     # If the folder entry is a regular file
31 |     if ( -f "$folder/$file" ) {
32 |         print "$folder/$file\n";
33 |
34 |         # If the folder entry is a subfolder
35 |     }
36 |     elsif ( -d "$folder/$file" ) {
37 |
38 |         my $folder = "$folder/$file";
39 |
40 |         # open the subfolder and list its contents
41 |         unless ( opendir( FOLDER, "$folder" ) ) {
42 |             print "Cannot open folder $folder!\n";
43 |             exit;
44 |         }
45 |
46 |         my @files = grep ( !/^\.\.?$/, readdir(FOLDER) );
47 |
48 |         closedir(FOLDER);
49 |
50 |         foreach my $file (@files) {
51 |             print "$folder/$file\n";
52 |         }
53 |     }
54 | }
55 |
56 | exit;
```

下面是例 11.2 的输出：

```
1 | pdb/3c/pdb43c9.ent
2 | pdb/3c/pdb43ca.ent
3 | pdb/44/pdb144d.ent
4 | pdb/44/pdb144l.ent
5 | pdb/44/pdb244d.ent
6 | pdb/44/pdb244l.ent
7 | pdb/44/pdb344d.ent
8 | pdb/44/pdb444d.ent
9 | pdb/pdb1a4o.ent
```

注意，代码中 `$file` 和 `@files` 这样的变量名是如何被重用的，方法就是在内层的代码块中使用 `my` 限定词汇作用域。如果程序的整体结构不是这么简短，这样阅读起来就会相当困难。当程序中出现 `$file` 的时候，它是表示此处的 `$file` 还是彼处的 `$file`？这个代码就是一个引起麻烦的反面例子。它确实可以工作，但尽管它很简短，任然非常难于阅读。

事实上，例 11.2 存在一个深层次的问题，它的设计并不好。通过对例 11.1 进行扩充，它现在可以罗列出子目录了。但是如果还有更深层次的子目录呢？

11.2.2 递归

如果你有一个子程序，可以罗列出目录的内容，并且可以通过递归调用自己来罗列出它找到的任何子目录的内容，那么你就可以在顶层目录调用它，而它最终则会罗列出所有的文件。

让我们编写另一个程序来完成这个工作吧。一个递归子程序被简单的定义为可以调用自己的子程序。下面是伪代码和代码（例 11.3，之后是对递归工作原理的讨论：

```

1  subroutine list_recursively
2
3      open folder
4
5      for each item in the folder
6
7          if it's a file
8              print its name
9
10         else if it's a folder
11             list_recursively
12     }
13 }
```

例 11.3：一个罗列文件系统的递归子程序

```

1  #!/usr/bin/perl
2  # Example 11-3   Demonstrate a recursive subroutine to list a subtree of a filesystem
3
4  use strict;
5  use warnings;
6  use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8  list_recursively('pdb');
9
10 exit;
11
12 #####
13 # Subroutine
14 #####
15
16 # list_recursively
17 #
18 #   list the contents of a directory,
19 #       recursively listing the contents of any subdirectories
20
21 sub list_recursively {
22
23     my ($directory) = @_;
24
25     my @files = ();
```

```
26
27     # Open the directory
28     unless ( opendir( DIRECTORY, $directory ) ) {
29         print "Cannot open directory $directory!\n";
30         exit;
31     }
32
33     # Read the directory, ignoring special entries "." and ".."
34     #
35     @files = grep ( !/^\.\/\.$/, readdir(DIRECTORY) );
36
37     closedir(DIRECTORY);
38
39     # If file, print its name
40     # If directory, recursively print its contents
41
42     # Notice that we need to prepend the directory name!
43     foreach my $file (@files) {
44
45         # If the directory entry is a regular file
46         if ( -f "$directory/$file" ) {
47
48             print "$directory/$file\n";
49
50             # If the directory entry is a subdirectory
51         }
52         elsif ( -d "$directory/$file" ) {
53
54             # Here is the recursive call to this subroutine
55             list_recursively("$directory/$file");
56         }
57     }
58 }
```

下面是例 11.3 的输出（注意它和例 11.2 的输出是完全一样的）：

```
1 | pdb/3c/pdb43c9.ent
2 | pdb/3c/pdb43ca.ent
3 | pdb/44/pdb144d.ent
4 | pdb/44/pdb144l.ent
5 | pdb/44/pdb244d.ent
6 | pdb/44/pdb244l.ent
7 | pdb/44/pdb344d.ent
8 | pdb/44/pdb444d.ent
9 | pdb/pdb1a4o.ent
```

看一下例 11.3 的代码，把它和例 11.2 比较以下。如你所见，程序大体上是一样的。例 11.2 整体就是一个主程序；而例 11.3 拥有和它几乎完全一样的代码，只不过这些代码被打包成了一个子程序，然后在一个简短的主程序中调用这个子程序。例 11.3 的主程序

只是简单的调用了一个递归函数，给它一个目录名（我计算机中存在的一个目录；当你尝试在自己的计算机中运行这个程序的时候，你可能需要修改目录名）即可。下面就是这个调用

```
1 | list_recursively('pdb');
```

我对此有些失望，不知道你是不是也有这种感觉。这看上去和其他的子程序调用并没有什么区别。显然，递归必须在子程序内部进行定义。这出现在 *list_recursively* 子程序的最末尾，当程序发现（使用 `-d` 文件测试操作符）目录列出的一个内容本身就是一个目录时，就会进行递归处理，和例 11.2 中的代码相比这有着显著的不同。在这一点上，例 11.2 有再一次寻找常规文件和目录的代码，而例 11.3 中的这个子程序通过简单的调用一个子程序就实现了这一点，这里的这个子程序就是它本身，叫做 *list_recursively* 的子程序：

```
1 | list_recursively("$directory/$file");
```

这就是递归。

正如你在此处所见，有很多时候，数据——比如文件系统的层次结构——正好能够匹配递归程序的这种能力。在子程序的末尾进行递归调用，这意味着它是一个特殊类型的递归，叫做尾递归 (*tail recursion*)。尽管递归会很慢，归因于它创建的所有子程序调用，关于尾递归的好消息就是许多编译器会对代码进行优化，让它运行更快一些。使用递归可以得到简洁、易于理解的程序。（尽管 Perl 并不对它进行优化，现在 Perl6 的计划中包含对尾递归进行优化的支持。）

11.2.3 处理大量文件

Perl 有可以处理各种任务的模块。有些模块是作为标准和 Perl 一块发布的，更多的则可以从 CPAN (<http://www.CPAN.org/>) 或者其他地方下载安装。

上一小节的例 11.3 演示了如何定位一个给定目录中的所有文件和目录。在所有近期版本的 Perl 中都有一个叫做 *File::Find* 的标准模块。你可以在你的手册页中找到它：比如，在 Unix 或者 Linux 上，你可以使用命令 `perldoc File::Find`。这个模块让处理一个给定目录的所有文件变得简单且高效，它可以进行你指定的各种操作。

例 11.4 使用了 *File::Find*。对于这个有用的模块的更多例子，可以参考它的文档。这个实例演示的功能和例 11.3 是完全一样的，只不过现在使用了 *File::Find*。它只是简单的把文件和目录罗列出来。注意，你会发现，如果你找到一个好的模块，你只需要编写很少的代码即可，所以开始使用模块吧！

例 11.4：演示 File::Find

```
1 | #!/usr/bin/perl
2 | # Example 11-4   Demonstrate File::Find
3 |
4 | use strict;
5 | use warnings;
6 | use BeginPerlBioinfo;    # see Chapter 6 about this module
7 |
```

```
8 use File::Find;
9
10 find( \&my_sub, ('pdb') );
11
12 sub my_sub {
13     -f and ( print $File::Find::name, "\n" );
14 }
15
16 exit;
```

注意，通过在 *my_sub* 子程序前面使用反斜杠字符，把指针传递给了它。就像在第 6 章中提到的那样，你还需要在它名字的前面加上 & 字符。

find 的调用也可以这样实现：

```
1 find sub { -f and (print $File\dotFind\dotname, "\n") }, ('pdb');
```

它把一个匿名子程序放在了 *my_sub* 子程序指针出现的地方，对于这种简短的子程序来说，这也是一种比较简洁的写法。

下面是它的输出：

```
1 pdb/pdb1a4o.ent
2 pdb/44/pdb144d.ent
3 pdb/44/pdb144l.ent
4 pdb/44/pdb244d.ent
5 pdb/44/pdb244l.ent
6 pdb/44/pdb344d.ent
7 pdb/44/pdb444d.ent
8 pdb/3c/pdb43c9.ent
9 pdb/3c/pdb43ca.ent
```

作为使用 Perl 处理文件的最后一个例子，下面是一个在命令行中使用的单行程序，它的作用和上面这个程序是完全一样的：

```
1 perl -e 'use File::Find;find sub{-f and (print $File::Find::name,"\n")},("pdb")'
```

尽管它不可避免的让人困惑，但对于那些崇尚简洁的人来说，这简直太酷了！此外还要注意，对于 Unix 操作系统的用户来说，`ls -R pdb` 和 `find pdb -print` 也可以完成同样的工作，而且键入的字符更少。

之所以使用你定义的一个子程序，是因为它可以让你对找到的文件进行任意的测试，然后对这些文件进行任意的处理。模块化则是另外一个例子：*File::Find* 模块可以轻而易举地对一个文件结构中的所有文件和目录进行递归，让你随意处理找到的文件和目录。

11.3 PDB 文件

下面是一个真实 PDB 文件的一部分：

```
1  HEADER      SUGAR BINDING PROTEIN                      03-MAR-99    1C1F
2  TITLE      LIGAND-FREE CONGERIN I
3  COMPND     MOL_ID: 1;
4  COMPND     2 MOLECULE: CONGERIN I;
5  COMPND     3 CHAIN: A;
6  COMPND     4 FRAGMENT: CARBOHYDRATE-RECOGNITION-DOMAIN;
7  COMPND     5 BIOLOGICAL_UNIT: HOMODIMER
8  SOURCE      MOL_ID: 1;
9  SOURCE      2 ORGANISM_SCIENTIFIC: CONGER MYRIASTER;
10 SOURCE      3 ORGANISM_COMMON: CONGER EEL;
11 SOURCE      4 TISSUE: SKIN MUCUS;
12 SOURCE      5 SECRETION: NON-CLASSICAL
13 KEYWDS      GALECTIN, LECTIN, BETA-GALACTOSE-BINDING, SUGAR BINDING
14 KEYWDS      2 PROTEIN
15 EXPDTA      X-RAY DIFFRACTION
16 AUTHOR      T.SHIRAI,C.MITSUYAMA,Y.NIWA,Y.MATSUI,H.HOTTA,T.YAMANE,
17 AUTHOR      2 H.KAMIYA,C.ISHII,T.OGAWA,K.MURAMOTO
18 REVDAT      2 14-OCT-99 1C1F 1 SEQADV HEADER
19 REVDAT      1 08-OCT-99 1C1F 0
20 JRNL        AUTH T.SHIRAI,C.MITSUYAMA,Y.NIWA,Y.MATSUI,H.HOTTA,
21 JRNL        AUTH 2 T.YAMANE,H.KAMIYA,C.ISHII,T.OGAWA,K.MURAMOTO
22 JRNL        TITL HIGH-RESOLUTION STRUCTURE OF CONGER EEL GALECTIN,
23 JRNL        TITL 2 CONGERIN I, IN LACTOSE- LIGANDED AND LIGAND-FREE
24 JRNL        TITL 3 FORMS: EMERGENCE OF A NEW STRUCTURE CLASS BY
25 JRNL        TITL 4 ACCELERATED EVOLUTION
26 JRNL        REF STRUCTURE (LONDON) V. 7 1223 1999
27 JRNL        REFN ASTM STRUE6 UK ISSN 0969-2126 2005
28 REMARK      1
29 REMARK      2
30 REMARK      2 RESOLUTION. 1.6 ANGSTROMS.
31 REMARK      3
32 REMARK      3 REFINEMENT.
33 REMARK      3 PROGRAM : X-PLOR 3.1
34 REMARK      3 AUTHORS : BRUNGER
35 REMARK      3
36 REMARK      3 DATA USED IN REFINEMENT.
37 REMARK      3 RESOLUTION RANGE HIGH (ANGSTROMS) : 1.60
38 REMARK      3 RESOLUTION RANGE LOW (ANGSTROMS) : 8.00
39 REMARK      3 DATA CUTOFF (SIGMA(F)) : 3.000
40 REMARK      3 DATA CUTOFF HIGH (ABS(F)) : NULL
41 REMARK      3 DATA CUTOFF LOW (ABS(F)) : NULL
42 REMARK      3 COMPLETENESS (WORKING+TEST) (%) : 85.0
43 REMARK      3 NUMBER OF REFLECTIONS : 17099
44 REMARK      3
```

```
45 REMARK 3
46 REMARK 3 FIT TO DATA USED IN REFINEMENT.
47 REMARK 3 CROSS-VALIDATION METHOD : THROUGHOUT
48 REMARK 3 FREE R VALUE TEST SET SELECTION : RANDOM
49 REMARK 3 R VALUE (WORKING SET) : 0.201
50 REMARK 3 FREE R VALUE : 0.247
51 REMARK 3 FREE R VALUE TEST SET SIZE (%) : 5.000
52 REMARK 3 FREE R VALUE TEST SET COUNT : 855
53 REMARK 3 ESTIMATED ERROR OF FREE R VALUE : NULL
54 REMARK 3
55 ...
56
57 (file truncated here)
58
59
60 REMARK 4
61 REMARK 4 1CIF COMPLIES WITH FORMAT V. 2.3, 09-JULY-1998
62 REMARK 7
63 REMARK 7 >>> WARNING: CHECK REMARK 999 CAREFULLY
64 REMARK 8
65 REMARK 8 SIDE-CHAINS OF SER123 AND LEU124 ARE MODELED AS ALTERNATIVE
66 REMARK 8 CONFORMERS.
67 REMARK 9
68 REMARK 9 SER1 IS ACETYLATED.
69 REMARK 10
70 REMARK 10 TER
71 REMARK 10 SER: THE N-TERMINAL RESIDUE WAS NOT OBSERVED
72 REMARK 100
73 REMARK 100 THIS ENTRY HAS BEEN PROCESSED BY RCSB ON 07-MAR-1999.
74 REMARK 100 THE RCSB ID CODE IS RCSB000566.
75 REMARK 200
76 REMARK 200 EXPERIMENTAL DETAILS
77 REMARK 200 EXPERIMENT TYPE : X-RAY DIFFRACTION
78 REMARK 200 DATE OF DATA COLLECTION : NULL
79 REMARK 200 TEMPERATURE (KELVIN) : 291.0
80 REMARK 200 PH : 9.00
81 REMARK 200 NUMBER OF CRYSTALS USED : 1
82 REMARK 200
83 REMARK 200 SYNCHROTRON (Y/N) : Y
84 REMARK 200 RADIATION SOURCE : PHOTON FACTORY
85 REMARK 200 BEAMLINE : BL6A
86 REMARK 200 X-RAY GENERATOR MODEL : NULL
87 REMARK 200 MONOCHROMATIC OR LAUE (M/L) : M
88 REMARK 200 WAVELENGTH OR RANGE (A) : 1.00
89 REMARK 200 MONOCHROMATOR : NULL
90 REMARK 200 OPTICS : NULL
91 REMARK 200
92 ...
93
```

```

94 (file truncated here)
95
96
97 REMARK 500
98 REMARK 500 GEOMETRY AND STEREOCHEMISTRY
99 REMARK 500 SUBTOPIC: COVALENT BOND ANGLES
100 REMARK 500
101 REMARK 500 THE STEREOCHEMICAL PARAMETERS OF THE FOLLOWING RESIDUES
102 REMARK 500 HAVE VALUES WHICH DEVIATE FROM EXPECTED VALUES BY MORE
103 REMARK 500 THAN 4*RMSD (M=MODEL NUMBER; RES=RESIDUE NAME; C=CHAIN
104 REMARK 500 IDENTIFIER; SSEQ=SEQUENCE NUMBER; I=INSERTION CODE).
105 REMARK 500
106 REMARK 500 STANDARD TABLE:
107 REMARK 500 FORMAT: (10X,I3,1X,A3,1X,A1,I4,A1,3(1X,A4,2X),12X,F5.1)
108 REMARK 500
109 REMARK 500 EXPECTED VALUES: ENGH AND HUBER, 1991
110 REMARK 500
111 REMARK 500  M RES CSSEQI ATM1  ATM2  ATM3
112 REMARK 500    HIS A  44  N   -  CA   -  C   ANGL. DEV.  =-10.3 DEGREES
113 REMARK 500    LEU A 132  CA   -  CB   -  CG  ANGL. DEV.  = 12.5 DEGREES
114 REMARK 700
115 REMARK 700 SHEET
116 REMARK 700 DETERMINATION METHOD: AUTHOR-DETERMINED
117 REMARK 999
118 REMARK 999 SEQUENCE
119 REMARK 999 LEU A 135 IS NOT PRESENT IN SEQUENCE DATABASE
120 REMARK 999
121 DBREF  1C1F A    1   136  SWS    P26788    LEG_CONMY    1   135
122 SEQADV 1C1F LEU A  135  SWS  P26788                SEE REMARK 999
123 SEQRES  1 A  136  SER GLY GLY LEU GLN VAL LYS ASN PHE ASP PHE THR VAL
124 SEQRES  2 A  136  GLY LYS PHE LEU THR VAL GLY GLY PHE ILE ASN ASN SER
125 SEQRES  3 A  136  PRO GLN ARG PHE SER VAL ASN VAL GLY GLU SER MET ASN
126 SEQRES  4 A  136  SER LEU SER LEU HIS LEU ASP HIS ARG PHE ASN TYR GLY
127 SEQRES  5 A  136  ALA ASP GLN ASN THR ILE VAL MET ASN SER THR LEU LYS
128 SEQRES  6 A  136  GLY ASP ASN GLY TRP GLU THR GLU GLN ARG SER THR ASN
129 SEQRES  7 A  136  PHE THR LEU SER ALA GLY GLN TYR PHE GLU ILE THR LEU
130 SEQRES  8 A  136  SER TYR ASP ILE ASN LYS PHE TYR ILE ASP ILE LEU ASP
131 SEQRES  9 A  136  GLY PRO ASN LEU GLU PHE PRO ASN ARG TYR SER LYS GLU
132 SEQRES 10 A  136  PHE LEU PRO PHE LEU SER LEU ALA GLY ASP ALA ARG LEU
133 SEQRES 11 A  136  THR LEU VAL LYS LEU GLU
134 FORMUL  2  HOH  *81(H2 O1)
135 HELIX   1  1 GLY A  66  ASN A  68  5
136 |
136 SHEET  1  S1 1 GLY A  3  VAL A  6  0
137 SHEET  1  S2 1 PHE A 121 GLY A 126  0
138 SHEET  1  S3 1 ARG A  29 GLY A  35  0
139 SHEET  1  S4 1 LEU A  41 ASN A  50  0
140 SHEET  1  S5 1 GLN A  55 THR A  63  0
141 SHEET  1  S6 1 GLN A  74 SER A  76  0

```

```

142 SHEET      1  F1  1  ALA  A 128  GLU  A 136  0
143 SHEET      1  F2  1  PHE  A   16  ILE  A   23  0
144 SHEET      1  F3  1  TYR  A   86  TYR  A   93  0
145 SHEET      1  F4  1  LYS  A   97  ILE  A  102  0
146 SHEET      1  F5  1  ASN  A  107  PRO  A  111  0
147 CRYST1     94.340   36.920   40.540  90.00  90.00  90.00 P 21 21 2      4
148 ORIGX1      1.000000   0.000000   0.000000      0.000000
149 ORIGX2      0.000000   1.000000   0.000000      0.000000
150 ORIGX3      0.000000   0.000000   1.000000      0.000000
151 SCALE1      0.010600   0.000000   0.000000      0.000000
152 SCALE2      0.000000   0.027085   0.000000      0.000000
153 SCALE3      0.000000   0.000000   0.024667      0.000000
154 ATOM        1  N    GLY  A   2      1.888  -8.251  -2.511  1.00 36.63
|  N
155 ATOM        2  CA   GLY  A   2      2.571  -8.428  -1.248  1.00 33.02
|  C
156 ATOM        3  C    GLY  A   2      2.586  -7.069  -0.589  1.00 30.43
|  C
157 ATOM        4  O    GLY  A   2      2.833  -6.107  -1.311  1.00 33.27
|  O
158 ATOM        5  N    GLY  A   3      2.302  -6.984   0.693  1.00 24.67
|  N
159 ATOM        6  CA   GLY  A   3      2.176  -5.723   1.348  1.00 18.88
|  C
160 ATOM        7  C    GLY  A   3      0.700  -5.426   1.526  1.00 16.58
|  C
161 ATOM        8  O    GLY  A   3     -0.187  -6.142   1.010  1.00 12.47
|  O
162 ATOM        9  N    LEU  A   4      0.494  -4.400   2.328  1.00 15.00
|  N
163 ...
164
165 (file truncated here)
166
167
168 ATOM       1078  CG   GLU  A 136     -0.873   9.368  16.046  1.00 38.96
|  C
169 ATOM       1079  CD   GLU  A 136     -0.399   9.054  17.456  1.00 44.66
|  C
170 ATOM       1080  OE1  GLU  A 136      0.789   8.749  17.641  1.00 47.97
|  O
171 ATOM       1081  OE2  GLU  A 136     -1.236   9.099  18.361  1.00 47.75
|  O
172 ATOM       1082  OXT  GLU  A 136      0.764  12.146  12.712  1.00 26.22
|  O
173 TER        1083      GLU  A 136
174 HETATM     1084  O    HOH    200     -1.905  -7.624   2.822  1.00 14.50
|  O
175 HETATM     1085  O    HOH    201     -8.374   7.981   9.202  1.00 20.77

```

```

| 0
176 | HETATM 1086 0 HOH 202 -4.047 9.199 11.632 1.00 38.24
| 0
177 | HETATM 1087 0 HOH 203 6.172 14.210 8.483 1.00 14.50
| 0
178 | HETATM 1088 0 HOH 204 2.903 7.804 15.329 1.00 24.51
| 0
179 | HETATM 1089 0 HOH 205 16.654 0.676 11.968 1.00 10.49
| 0
180 | ...
181 |
182 | (file truncated here)
183 |
184 |
185 | HETATM 1157 0 HOH 286 6.960 14.840 -3.025 1.00 35.59
| 0
186 | HETATM 1158 0 HOH 287 -3.222 10.410 7.061 1.00 38.91
| 0
187 | HETATM 1159 0 HOH 288 28.306 0.551 4.876 1.00 52.13
| 0
188 | HETATM 1160 0 HOH 290 21.506 -12.424 9.751 1.00 31.68
| 0
189 | HETATM 1161 0 HOH 291 12.951 10.424 -7.324 1.00 46.10
| 0
190 | HETATM 1162 0 HOH 292 18.119 -15.184 14.793 1.00 56.82
| 0
191 | HETATM 1163 0 HOH 293 13.501 22.220 8.216 1.00 43.30
| 0
192 | HETATM 1164 0 HOH 294 13.916 -11.387 9.695 1.00 47.13
| 0
193 | MASTER 240 0 0 1 11 0 0 6 1163 1 0 11
194 | END

```

PDB 文件非常长，主要是因为需要存储分子中每个原子的信息。这还算是一个相对简短的例子，如果全部列出来，它会非常长——有 28 页纸之多。此处，我把它截取了一下，缩减到了三页纸多点，把主要的部分全部都展示了出来，让你能有一个大体上的了解。

PDB 网站上有基本的文档，当你阅读 PDB 文件并且编程处理它们的时候，你会需要这个文档的。PDB 内容指南 (Protein Data Bank Contents Guide, http://www.rcsb.org/pdb/docs/format/pdbguide2.2/guide2.2_frame.html) 可以说是最好的参考资料，里面有 FAQs 和一些额外的文档。

在接下来的小节里，我会从这些文件中提取信息。因为这些文件描述的主要是大分子 3D 结构的信息，所以这些文件通常被图形程序所使用，来展示分子的空间结构。本书所讨论的范围并不包括图形，尽管如此，你还是将会看到如何从这些文件中提取出空间坐标。PDB 文件中最大的一部分是包含原子坐标的 ATOM 记录类型行。因为这样详细的程度，PDB 文件通常都比 GenBank 记录要长。（注意术语上的不一致——PDB 的基本单元就是文件，它包含了一个结构；而 GenBank 的基本单元是记录，它包含了一个条目。）

11.3.1 PDB 文件格式

让我们看一个 PDB 文件，文档告诉了我们 PDB 文件中的信息是如何组织的。基于这些信息，你可以解析文件，从中提取出你感兴趣的信息。

PDB 文件有含有 80 列的多行组成，每一行都以某个预定义的记录名起始，以换行符终止。（“列”表示一行中的位置：第一个字符在第一列上，以此类推。）空列用空白填充。一个记录类型是有相同记录名的一行或多行。不同的记录类型有在行中定义的不同类型字段。它们也据功能进行分组。

SEQRES 记录类型是一级结构部分（Primary Structure Section）中的四大记录类型之一，它描述了肽或核苷酸序列的一级结构：

DBREF

指向序列数据库中的记录

SEQADV

记录 PDB 与其他序列数据库的冲突

SEQRES

骨干残基的一级序列

MODRES

记录标准残基上的修饰

上一节的 PDB 记录例子中的 DBREF 和 SEQADV 记录类型，给出了参考信息，以及 PDB 和原始数据库之间的冲突细节。（例子中不包含 MODRES 记录类型。）下面是这个记录中的这些记录类型：

```
1 DBREF  1C1F  A      1   136  SWS      P26788  LEG_CONMY      1   135
2 SEQADV 1C1F  LEU  A   135  SWS      P26788                      SEE REMARK 999
```

简单来说，DBREF 行声明有一个叫做 *1C1F* 的 PDB 文件（来自于叫做 *pdb1c1f.ent* 的文件），在原始的 Swiss-Prot（SWS）数据库中 A 链的残基从 1 开始编号一直到 136，在那个数据中它的 ID 号为 P26788，名字为 LEG_CONMY（在许多数据库中这些都是一样的），在 PDB 数据库中残基从 1 开始编号到 135。原始数据库和 PDB 编号上的差异在 SEQADV 记录类型中进行了解释，它让你参考 REMARK 的 999 行（此处未显示），在那里你会发现 PDB 记录对于 Swiss-Prot 序列上第 135 位的亮氨酸有歧义（也许这是两个不同小组测定的结构，它们在这个位点上有分歧）。²

你可以看到，要通过程序解析这两行的信息，需要好几步，比如跟随到达 PDB 记录其他行的链接，它会进一步解释重复，并且识别其他的数据库。

在生物信息学中，数据库之间的链接是非常重要的。表 11.1 展示了 PDB 文件中参考的数据库。如你所知，有许多的生物学数据库，这儿展示的主要是和蛋白质或者结构数据相关的数据库。

11.3.2 SEQRES

刚刚起步，我们先用 Perl 来尝试一个相对简单的任务：提取氨基酸序列数据。要提取氨基酸的一级序列信息，你需要解析 SEQRES 记录类型。下面是先前那个 PDB 文件中

²在老的 PDB 文件中，不同数据库的交叉引用是一个问题：它可能缺失，或者隐藏在 REMARK 的 999 行的某个地方。

表 11.1: PDB 文件中参考的数据库

Database
BioMagResBank
BLOCKS
European Molecular Biology Laboratory
GenBank
Genome Data Base
Nucleic Acid Database
PROSITE
Protein Data Bank
Protein Identification Resource
SWISS-PROT
TREMBL

的 SEQRES 行:

```
1 | SEQRES      1 A  136  SER GLY GLY LEU GLN VAL LYS ASN PHE ASP PHE THR VAL
```

下面展示了 PDB 内容指南中定义的 SEQRES 记录类型。SEQRES 部分，是一个相对简单的记录类型，它的定义被全部展示了出来，帮助你熟悉一下这种文档。

1	SEQRES			
2				
3	Overview			
4				
5	SEQRES records contain the amino acid or nucleic acid sequence of residues in			
6	each chain of the			
7	macromolecule that was studied.			
8				
9	Record Format			
10				
11	COLUMNS	DATA TYPE	FIELD	DEFINITION
12	-----			
13	1 - 6	Record name	"SEQRES"	
14				
15	9 - 10	Integer	serNum	Serial number of the SEQRES record
16				for the current chain. Starts at 1
17				and increments by one each line.
18				Reset to 1 for each chain.
19				
20	12	Character	chainID	Chain identifier. This may be any
21				single legal character, including a
22				blank which is used if there is
23				only one chain.
24				
25	14 - 17	Integer	numRes	Number of residues in the chain.

26 This value is repeated on every
27 record.
28
29 20 - 22 Residue name resName Residue name.
30
31 24 - 26 Residue name resName Residue name.
32
33 28 - 30 Residue name resName Residue name.
34
35 32 - 34 Residue name resName Residue name.
36
37 36 - 38 Residue name resName Residue name.
38
39 40 - 42 Residue name resName Residue name.
40
41 44 - 46 Residue name resName Residue name.
42
43 48 - 50 Residue name resName Residue name.
44
45 52 - 54 Residue name resName Residue name.
46
47 56 - 58 Residue name resName Residue name.
48
49 60 - 62 Residue name resName Residue name.
50
51 64 - 66 Residue name resName Residue name.
52
53 68 - 70 Residue name resName Residue name.
54
55 Details
56
57 * PDB entries use the three-letter abbreviation for amino acid names and the
58 one-letter code for nucleic acids.
59
60 * In the case of non-standard groups, a hetID of up to three (3) alphanumeric
61 characters is used. Common HET names appear in the HET dictionary.
62
63 * Each covalently contiguous sequence of residues (connected via the "backbone"
64 atoms) is represented as an individual chain.
65
66 * Heterogens which are integrated into the backbone of the chain are listed as
67 being part of the chain and are included in the SEQRES records for that chain.
68
69 * Each set of SEQRES records and each HET group is assigned a component number.
70 The component number is assigned serially beginning with 1 for the first set
71 of SEQRES records. This number is given explicitly in the FORMUL record, but
72 only implicitly in the SEQRES record.
73
74 * The SEQRES records must list residues present in the molecule studied, even

```

75   if the coordinates are not present.
76
77   * C- and N-terminus residues for which no coordinates are provided due to
78     disorder must be listed on SEQRES.
79
80   * All occurrences of standard amino or nucleic acid residues (ATOM records)
81     must be listed on a SEQRES record. This implies that a numRes of 1 is valid.
82
83   * No distinction is made between ribo- and deoxyribonucleotides in the SEQRES
84     records. These residues are identified with the same residue name (i.e., A,
85     C, G, T, U, I).
86
87   * If the entire residue sequence is unknown, the serNum in column 10 is "0",
88     the number of residues thought to comprise the molecule is entered as numRes
89     in columns 14 - 17, and resName in columns 20 - 22 is "UNK".
90
91   * In case of microheterogeneity, only one of the sequences is presented. A
92     REMARK is generated to explain this and a SEQADV is also generated.
93
94   Verification/Validation/Value Authority Control
95
96   The residues presented on the SEQRES records must agree with those found in
97   the ATOM records.
98
99   The SEQRES records are checked by PDB using the sequence databases and
100  information provided by the depositor.
101
102  SEQRES is compared to the ATOM records during processing, and both are checked
103  against the sequence database. All discrepancies are either resolved or
104  annotated in the entry.
105
106  Relationships to Other Record Types
107
108  The residues presented on the SEQRES records must agree with those found in
109  the ATOM records. DBREF refers to the corresponding entry in the sequence
110  databases. SEQADV lists all discrepancies between the entry's sequence for
111  which there are coordinates and that referenced in the sequence database.
112  MODRES describes modifications to a standard residue.
113
114  Example
115
116      1          2          3          4          5          6          7
117  123456789012345678901234567890123456789012345678901234567890
118  SEQRES  1 A   21  GLY ILE VAL GLU GLN CYS CYS THR SER ILE CYS SER LEU
119  SEQRES  2 A   21  TYR GLN LEU GLU ASN TYR CYS ASN
120  SEQRES  1 B   30  PHE VAL ASN GLN HIS LEU CYS GLY SER HIS LEU VAL GLU
121  SEQRES  2 B   30  ALA LEU TYR LEU VAL CYS GLY GLU ARG GLY PHE PHE TYR
122  SEQRES  3 B   30  THR PRO LYS ALA
123  SEQRES  1 C   21  GLY ILE VAL GLU GLN CYS CYS THR SER ILE CYS SER LEU

```

```
124 SEQRES  2 C  21  TYR GLN LEU GLU ASN TYR CYS ASN
125 SEQRES  1 D  30  PHE VAL ASN GLN HIS LEU CYS GLY SER HIS LEU VAL GLU
126 SEQRES  2 D  30  ALA LEU TYR LEU VAL CYS GLY GLU ARG GLY PHE PHE TYR
127 SEQRES  3 D  30  THR PRO LYS ALA
128
129 Known Problems
130
131 Polysaccharides do not lend themselves to being represented in SEQRES.
132
133 There is no mechanism provided to describe sequence runs when the exact
134 ordering of the sequence is not known.
135
136 For cyclic peptides, PDB arbitrarily assigns a residue as the N-terminus.
137
138 For microheterogeneity only one of the possible residues in a given position
139 is provided in SEQRES.
140
141 No distinction is made between ribo- and deoxyribonucleotides in the SEQRES
142 records. These residues are identified with the same residue name (i.e., A,
143 C, G, T, U).
```

这行的结构包括显而易见的 SEQRES 记录类型，以及指定给链中特定位置或者列的字段。稍后你会看到如何利用这些位置来解析信息。注意，文档中包含了大量的细节，当你处理这样复杂的实验数据时可能会需要。

除了序列不断积累这个还算标准的问题外，多链会让它更加复杂。通过阅读刚才展示的文档，你会看到在 SEQRES 这个识别符后面，还有一个表示这条链的行数的数字，接下来的字段就是链信息（尽管在老的记录中，它是可选的，并且很可能会空着）。在这些字段后是表示链中残基总数的一个数字。最后，在这些内容之后，使用三字母代码表示的残基。为了实现我们的编程目的，哪些信息是需要的，哪些又可以被忽略掉呢？

11.4 解析 PDB 文件

首先，例 11.5 演示了主程序和三个子程序，在本节将会对它们进行讨论。

例 11.5：从 PDB 文件中提取序列链

```

1  #!/usr/bin/perl
2  # Example 11-5   Extract sequence chains from PDB file
3
4  use strict;
5  use warnings;
6  use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8  # Read in PDB file:  Warning - some files are very large!
9  my @file = get_file_data('pdb/c1/pdb1c1f.ent');
10
11 # Parse the record types of the PDB file
12 my %recordtypes = parsePDBrecordtypes(@file);
13
14 # Extract the amino acid sequences of all chains in the protein
15 my @chains = extractSEQRES( $recordtypes{'SEQRES'} );
16
17 # Translate the 3-character codes to 1-character codes, and print
18 foreach my $chain (@chains) {
19     print "$chain\n";
20     print iub3to1($chain), "\n";
21 }
22
23 exit;
24
25 #####
26 # Subroutines for Example 11-5
27 #####
28
29 # parsePDBrecordtypes
30 #
31 #-given an array of a PDB file, return a hash with
32 #   keys   = record type names
33 #   values = scalar containing lines for that record type
34
35 sub parsePDBrecordtypes {
36
37     my @file = @_;
38
39     use strict;
40     use warnings;
41
42     my %recordtypes = ();
43

```

```
44     foreach my $line (@file) {
45
46         # Get the record type name which begins at the
47         # start of the line and ends at the first space
48
49         # The pattern (\S+) is returned and saved in $recordtype
50         my ($recordtype) = ( $line =~ /^(\S+)/ );
51
52         # .= fails if a key is undefined, so we have to
53         # test for definition and use either .= or = depending
54         if ( defined $recordtypes{$recordtype} ) {
55             $recordtypes{$recordtype} .= $line;
56         }
57         else {
58             $recordtypes{$recordtype} = $line;
59         }
60     }
61
62     return %recordtypes;
63 }
64
65 # extractSEQRES
66 #
67 #-given an scalar containing SEQRES lines,
68 #   return an array containing the chains of the sequence
69
70 sub extractSEQRES {
71
72     use strict;
73     use warnings;
74
75     my ($seqres) = @_ ;
76
77     my $lastchain = '';
78     my $sequence  = '';
79     my @results   = ();
80
81     # make array of lines
82
83     my @record = split( /\n/, $seqres );
84
85     foreach my $line (@record) {
86
87         # Chain is in column 12, residues start in column 20
88         my ($thischain) = substr( $line, 11, 1 );
89         my ($residues)  = substr( $line, 19, 52 );    # add space at end
90
91         # Check if a new chain, or continuation of previous chain
92         if ( "$lastchain" eq "" ) {
```

```
93         $sequence = $residues;
94     }
95     elsif ( "$thischain" eq "$lastchain" ) {
96         $sequence .= $residues;
97
98         # Finish gathering previous chain (unless first record)
99     }
100     elsif ($sequence) {
101         push( @results, $sequence );
102         $sequence = $residues;
103     }
104     $lastchain = $thischain;
105 }
106
107 # save last chain
108 push( @results, $sequence );
109
110 return @results;
111 }
112
113 # iub3to1
114 #
115 #-change string of 3-character IUB amino acid codes (whitespace separated)
116 #   into a string of 1-character amino acid codes
117
118 sub iub3to1 {
119
120     my ($input) = @_ ;
121
122     my %three2one = (
123         'ALA' => 'A',
124         'VAL' => 'V',
125         'LEU' => 'L',
126         'ILE' => 'I',
127         'PRO' => 'P',
128         'TRP' => 'W',
129         'PHE' => 'F',
130         'MET' => 'M',
131         'GLY' => 'G',
132         'SER' => 'S',
133         'THR' => 'T',
134         'TYR' => 'Y',
135         'CYS' => 'C',
136         'ASN' => 'N',
137         'GLN' => 'Q',
138         'LYS' => 'K',
139         'ARG' => 'R',
140         'HIS' => 'H',
141         'ASP' => 'D',
```

```

142         'GLU' => 'E',
143     );
144
145     # clean up the input
146     $input =~ s/\n/ /g;
147
148     my $seq = '';
149
150     # This use of split separates on any contiguous whitespace
151     my @code3 = split( ' ', $input );
152
153     foreach my $code (@code3) {
154
155         # A little error checking
156         if ( not defined $three2one{$code} ) {
157             print "Code $code not defined\n";
158             next;
159         }
160         $seq .= $three2one{$code};
161     }
162     return $seq;
163 }

```

一定要注意这一点，调用读取 PDB 文件的子程序 `get_file_data` 的主程序中，包含了一个对于大的 PDB 文件的警告。（比如，PDB 文件 *1gav* 就有 3.45 MB 之大。）另外，主程序中，把整个文件读取进来之后，子程序 `parsePDBrecordtypes` 把输入文件中的所有行都进行了拷贝，用记录类型分隔开。在这一点上，程序运行时会占用文件大小两倍大的内存空间。这种设计的有点在于它的清晰化和模块化，但主内存不够用的时候它也会导致问题。如果不保存从文件中读取进来的结果，而是直接把文件数据传递给 `parsePDBrecordtypes` 子程序，这样占用的内存会小一些，就想这样：

```

1 | # Get the file data and parse the record types of the PDB file
2 | %recordtypes = parsePDBrecordtypes(get_file_data('pdb/c1/pdb1c1f.ent'));

```

进一步节省内存也是可能的。比如，你可以重写程序，让它在把数据解析成记录类型的时候一次只读入文件的一行。我之所以指出这一点，是让你明白在处理大文件的时候，有许多种选择，这在实践中是非常重要的。尽管如此，现在我们还是坚持刚才的设计。这样可能比较耗费内存，但却让整个程序结构更加清晰明了。

在第 10 章中，我演示了两种方法，把 GenBank 文件解析成序列和注释，之后又把注释一层层得解析出来。

第一种方法是循环处理存储着记录行的数据。回忆一下，因为多行字段的结构，当循环的时候我们需要设置一个标识来记录输入行处于那个字段中。³

另外一种方法，更加适用于 GenBank 文件，使用的正则表达式。哪种方法对于 PDB 文件最适合呢？（又或者你探索一下第三种方法？）

³在 GenBank 中，多行信息集叫做字段；在 PDB 中，它们叫做记录类型。就像在生物学中，不同的研究人员可能会使用他们自己的术语来描述结构或者概念一样，在计算机科学中，大家对术语也有一定的创造性。这也是整合生物学数据资源时存在的比较有趣的困难之一。

有好多方法可以提取出这个信息。PDB 使得收集记录类型比较容易，因为它们在第一行都起始于同样的关键字。在上一章中，使用的正则表达式这种技术来解析文件顶层的字段，这种方法对于 PDB 文件来说略显笨重。（参看本章末尾的练习题。）比如，下面的这个匹配所有临近 SEQRES 行的正则表达式可以它们整合进一个标量字符串中：

```
1 | $record =~ /SEQRES.*\n(SEQRES.*\n)*;/
2 | $seqres = $&;
```

正则表达式使用 `SEQRES.*\n` 匹配单独的 SEQRES 行，然后使用 `(SEQRES.*\n)*` 匹配零行或者多行附加行。注意最后的 `*` 表明匹配前面的项目零次或者多次，也就是用小括号括起来的表达式 `(SEQRES.*\n)`。此外，还要注意 `.*` 匹配另一个或者多个非换行符。最后，第二行把用 `$&` 表示的匹配到的模式捕获起来，保存到了变量 `$seqres` 中。

要扩充它来捕获所有的记录类型，可以参看本章末尾的练习题。

对于 PDB 文件来说，每一行都起始于一个关键字，明确表明了该行属于哪种记录类型。在文档中，你会发现，在每一组中，每一种记录类型的行都彼此相邻。在这种情况下，简单的对所有行进行重复收集记录类型，这看起来是最简单的编程策略了。

例 11.5 包含一个叫做 *parsePDBrecordtypes* 的子程序，它从包含 PDB 记录行的数组中解析 PDB 记录类型。这是一个简洁的程序，它可以完成它需要完成的工作。注释对发生的事情进行了很好地描述，如你所知，对于编写好的代码来说注释是至关重要的一个因素。简单来说，对每一行的记录类型进行检查，然后把它添加到散列的值里面去，散列的键是记录类型。最后，散列从子程序中返回出来。

11.4.1 提取一级序列

既然记录类型已经被解析出来了，就让我们看看子程序 *extractSEQRES* 是如何提取一级氨基酸序列的。

你需要把每条链单独提取出来，返回对应这些链的一个或多个序列字符串的一个数组，而不是仅仅一条序列。

例 11.4 中前面的解析，只把需要的 SEQRES 记录类型保留了下来，它包含多行内容，以单个标量字符串的形式存储为散列的值，其对应的键为 ‘SEQRES’。先前对行进行循环处理（与对多行字符串使用正则表达式不同）的 *parsePDBrecordtypes* 子程序的成功导致了此处同样的方法。Perl 函数 *split* 可以让你把一个多行的字符串转换成一个数组。

在对 SEQRES 记录类型中的行进行循环处理的时候，注意，当一个新的链开始时，会把先前的链保存到数组 `@results` 中，重置变量 `$sequence`，同时也把 `$lastchain` 标识重置为新的链。此外，当处理完所有的行后，要确保把最后的序列链保存进 `@results` 数组。

还要注意（查阅这个函数的 Perl 文档进行确认），根据你给它的参数，*split* 会完成你所期望的工作。

例 11.5 中的第三个也是最后一个子程序叫做 *iub3to1*。因为 PDB 中的序列信息使用三字母进行编码的，所以你需要这个子程序来把这些序列转变成单字母编码的形式。它直截了当的使用了一个散列查找来完成这个转换。

我们现在已经把问题分解成了几个互相协作的子程序。如何最优得把一个问题分解成协作的子程序，这总是非常有趣。你可以把 *iub3to1* 的调用放在 *extractSEQRES* 子程序里

面，这可能是把这些子程序组合在一起的一个比较简洁明了的方法，毕竟，除了 PDB 文件格式，你很可能不会用到三字母编码的氨基酸字符串。

在这个关口，最重要的一个论点就是需要指出，在一个简短的主程序中把几个简短的子程序组合起来，足够来完成解析 PDB 文件这样复杂的任务了。

11.4.2 查找原子坐标

到现在为止，我只是对蛋白质结构进行了简单的概述，并没有试图进行更加详细的介绍。但是，在解析 PDB 文件的时候，你还是要面对一大堆的细节信息，关于结构和确定结构所使用的实验条件。现在，我将演示一个简短的程序，从 PDB 文件中提取出原子坐标。我不会进行全面的讲解，要想了解更多，你需要去仔细阅读 PDB 文档，以及蛋白质结构、X 射线衍射和 NMR 技术的参考资料。

刚才说，我们要从 ATOM 记录类型中提取坐标。ATOM 记录类型是 MODEL、ATOM、SIGATM、ANISOU、SIGUIJ、TER、HETATM 和 ENDMDL 等处理原子坐标数据的众多记录类型中的一种。此外，还有几个处理坐标转换的记录类型：ORIGXn、SCALEn、MTRIXn 和 TVECT。

下面是 PDB 文档中介绍每个 ATOM 记录的字段定义的部分：

1	ATOM			
2				
3	Overview			
4				
5	The ATOM records present the atomic coordinates for standard residues.			
6	They also present the occupancy and temperature factor for each atom.			
7	Heterogen coordinates use the HETATM record type. The element symbol			
8	is always present on each ATOM record; segment identifier and charge			
9	are optional.			
10				
11	Record Format			
12				
13	COLUMNS	DATA TYPE	FIELD	DEFINITION
14	-----			
15	1 - 6	Record name	"ATOM "	
16				
17	7 - 11	Integer	serial	Atom serial number.
18				
19	13 - 16	Atom	name	Atom name.
20				
21	17	Character	altLoc	Alternate location indicator.
22				
23	18 - 20	Residue name	resName	Residue name.
24				
25	22	Character	chainID	Chain identifier.
26				
27	23 - 26	Integer	resSeq	Residue sequence number.
28				

29	27	AChar	iCode	Code for insertion of residues.
30				
31	31 - 38	Real(8.3)	x	Orthogonal coordinates for X in
32				Angstroms.
33				
34	39 - 46	Real(8.3)	y	Orthogonal coordinates for Y in
35				Angstroms.
36				
37	47 - 54	Real(8.3)	z	Orthogonal coordinates for Z in
38				Angstroms.
39				
40	55 - 60	Real(6.2)	occupancy	Occupancy.
41				
42	61 - 66	Real(6.2)	tempFactor	Temperature factor.
43				
44	73 - 76	LString(4)	segID	Segment identifier, left-justified.
45				
46	77 - 78	LString(2)	element	Element symbol, right-justified.
47				
48	79 - 80	LString(2)	charge	Charge on the atom.

下面是一个典型的 ATOM 行：

```
1 | ATOM      1  N   GLY A   2      1.888 -8.251 -2.511  1.00 36.63
  | N
```

让我们来做一些非常简单的事情：提取出每个原子的 x 坐标、y 坐标和 z 坐标，以及其序列号（每个原子在分子中独一无二的整数序号）和元素符号。例 11.6 是一个完成该任务的子程序，以及一个执行该子程序的主程序。

例 11.6：从 PDB 文件中提取原子坐标

```
1 |#!/usr/bin/perl
2 |# Example 11-6   Extract atomic coordinates from PDB file
3 |
4 |use strict;
5 |use warnings;
6 |use BeginPerlBioinfo;    # see Chapter 6 about this module
7 |
8 |# Read in PDB file
9 |my @file = get_file_data('pdb/c1/pdb1c1f.ent');
10 |
11 |# Parse the record types of the PDB file
12 |my %recordtypes = parsePDBrecordtypes(@file);
13 |
14 |# Extract the atoms of all chains in the protein
15 |my %atoms = parseATOM( $recordtypes{'ATOM'} );
16 |
17 |# Print out a couple of the atoms
18 |print $atoms{'1'},    "\n";
```

```

19 print $atoms{'1078'}, "\n";
20
21 exit;
22
23 #####
24 # Subroutines of Example 11-6
25 #####
26
27 # parseATOM
28 #
29 # -extract x, y, and z coordinates, serial number and element symbol
30 #   from PDB ATOM record type
31 #   Return a hash with key=serial number, value=coordinates in a string
32
33 sub parseATOM {
34
35     my ($atomrecord) = @_ ;
36
37     use strict;
38     use warnings;
39     my %results = ();
40
41     # Turn the scalar into an array of ATOM lines
42     my (@atomrecord) = split( /\n/, $atomrecord );
43
44     foreach my $record (@atomrecord) {
45         my $number = substr( $record, 6, 5 );    # columns 7-11
46         my $x      = substr( $record, 30, 8 );   # columns 31-38
47         my $y      = substr( $record, 38, 8 );   # columns 39-46
48         my $z      = substr( $record, 46, 8 );   # columns 47-54
49         my $element = substr( $record, 76, 2 );  # columns 77-78
50
51         # $number and $element may have leading spaces: strip them
52         $number =~ s/^\s*//;
53         $element =~ s/^\s*//;
54
55         # Store information in hash
56         $results{$number} = "$x $y $z $element";
57     }
58
59     # Return the hash
60     return %results;
61 }

```

子程序 *parseATOM* 非常简短：ATOM 记录严格规范的格式使得从中解析信息非常直截了当。首先，你把包含 ATOM 行的标量参数分割成一个由行组成的数组。

然后，对于每一行，使用 *substr* 函数提取出该行特定的列，这些列包含了我们需要的数据：原子的序列号，x、y 和 z 坐标，以及元素符号。

最后，把结果保存到一个散列中，散列的键就是序列号，散列的值是包含其他四个相

关键字的字符串。现在，这可能不总是返回数据最便捷的方法。有一点要注意，散列并不对键进行排序，所以如果你要以序列号对原子进行排序，那就还需要额外的一步。尤其，要根据序列号的排序存储信息，数组是一个比较合理的选择。或者，如果你真正想要的是找到所有的金属原子，这种情况下，推荐使用另外一种数据结构。但不管怎样，这个简短的子程序演示了找到并报告信息的一种方法。

通常会发生这样的事情，你真正需要的是对数据进行重新格式化，用于其他的程序。使用这个子程序的技术，你可以看到，如何去提取需要的数据，以及通过添加 `print` 语句来数据格式化成想要的格式。看一下 `printf` 和 `sprintf` 函数，它们可以对格式进行更加细致的控制。对于真正的任务繁重的格式化，有一个 `format` 函数，在 O'Reilly's 详尽的 *Programming Perl* 的一书中有单独的一章对它进行介绍。（参看本书的第 12 章和附录 B。）

下面是例 11.6 的输出：

```
1 | 1.888   -8.251   -2.511   N
2 | 18.955  -10.180   10.777   C
```

现在，你至少可以从 PDB 文件中提取出主要的原子坐标部分了。重申一次，好消息是：你不需要一个冗长或者特别复杂的程序就可以完成它需要完成的任务。

这个程序进行了一定的设计，使得其中的部分可以在后续的工作中用于其他的目的。比如，你要解析所有的记录类型，而不仅仅是 ATOM 记录类型。让我们看下一个非常简短的程序，它只是从一个输入文件中把 ATOM 记录类型行解析了出来。如果仅仅是为了解决这个问题，你可以编写一个更加简短的程序。下面就是这个程序：

```
1 | while(<>) {
2 |     /^ATOM/ or next;
3 |
4 |     my($n, $x, $y, $z, $element)
5 |       = ($_ =~ /^.{6}.{5}.{19}.{8}.{8}.{8}.{22}(.)/);
6 |
7 |     # $n and $element may have leading spaces: strip them
8 |     $n      =~ s/^\s*//;
9 |     $element =~ s/^\s*//;
10 |
11 |     if (($n == 1) or ($n == 1078)) {
12 |         printf "%8.3f%8.3f%8.3f %2s\n", $x, $y, $z, $element;
13 |     }
14 | }
```

对于每一行，正则表达式匹配会提取出需要的信息。回忆一下，一个包含小括号元字符的正则表达式会返回一个数组，数组的元素就是小括号中匹配的字符串的特定部分。你把这些小括号中匹配到的子字符串赋值给了 `$number`、`$x`、`$y`、`$z` 和 `$element` 这五个变量。

实际上，正则表达式只是简单的使用了点号和量词操作符 `{num}` 来表示字符的个数。用这种方法，你可以从用脱字符 `^` 这个元字符表示的字符串的开头开始，指定这行信息中的特定的列，把你需要的信息用小括号括起来返回回来。

比如，你需要开头的六个字符，所以你用 `^.{6}` 指定它们，但是你需要接下来的五个字符，因为它们包含了原子的序列号，所以使用 `(.{5})` 来指定这个字段。

坦率的说，对于现在的这个目的来说，我认为使用 `substr` 更加清晰一些，但是我也想给你演示使用能够正则表达式的另外一种方法。

我们已经看到了使用 `printf` 函数比使用 `print` 函数有更多的选项可以控制输出的格式。

这个程序中还有一个重要的捷径。它并没有指定打开和读取的文件。在 Perl 中，你可以在命令行中给出输入文件的文件名（或者把它拖放到 Mac droplet 中），程序就会从那个文件中读取它的输入。像程序的第一行演示的那样，只需要使用尖括号就可以从文件中读取了。你可以把 `open` 函数中所有的调用和测试是否成功的测试全部丢弃掉，仅仅使用尖括号。你也可以在命令行中这样调用它，假设你把程序保存到了一个叫做 `get_two_atoms` 的文件中：

```
1 | %perl get_two_atoms pdb1a4o.ent
```

此外，你还可以通过管道把数据传递给程序，使用下面的命令：

```
1 | % cat pdb1a40.cat | perl get_two_atoms
```

或者进行重定向：

```
1 | % perl get_two__atoms < pdb1a40.ent
```

另外，在你的程序中也可以 `<STDIN>` 而不是 `<>` 来读取数据。

11.5 控制其他程序

Perl 使得在你的 Perl 程序中运行其他程序并且收集它们的输出非常简单。这是一个异常有用的能力。对于大多数程序来说，Perl 都可以非常简单的完成这个任务。

很多时候，你可能需要运行一些特定的程序，比如针对 PDB 中的每一个文件提取出二级结构信息。程序本身可能并没有办法让它“针对所有的文件运行自己”。另外，程序的输出中可能会有各种各样无关的信息。你需要的只是一个更加简单的报告，仅仅呈现你感兴趣的信息，可能要以一种特定的格式作为其他程序输入！使用 Perl，你可以编写一个程序来精确地完成该任务。

自动化执行的一种重要类型的程序就是网站上提供的在线的有用的程序或数据。使用合适的 Perl 模块，你可以连接到网站上，发送你的输入，收集输出，然后按照你的意愿进行解析和重格式化。这实际上并不难！作为 *Programming Perl* 的姊妹篇，O’ Reilly 的 *Perl Cookbook* 是一个简单程序和有用描述的优秀资源，可以帮助你快速起步。

Perl 是自动化其他成语的一种杰出的方法。下一小节将演示一个实例，使用一个 Perl 程序来启动其他的程序，并且收集、解析、重格式化和输出结果。这个程序会控制同一台计算机上的另一个程序。实例基于 Unix 或者 Linux 平台环境；关于如何在你的 Windows 或者 Macintosh 平台上实现同样的功能，请参阅你的 Perl 文档。

11.5.1 Stride 二级结构预测器

我们将会使用一个外部程序，基于一个 PDB 文件的 3D 坐标来计算它的二级结构。作为一个二级结构任务引擎，我使用叫做 *stride* 的程序，它会输出二级结构报告。*stride* 可以从 EMBL (http://www.embl-heidelberg.de/stride/stride_info.html⁴) 上获取到，它可以在 Unix、Linux、Windows、Macintosh 和 VMS 操作系统上运行。这个程序的工作原理非常简单，就是把一个 PDB 文件名作为命令行参数给它，并在之后的 *call_stride* 子程序中收集输出。

例 11.7 是完整的程序，包括两个子程序和一个主程序，程序的后面是讨论。

例 11.7：调用其他程序进行二级结构预测

```
1 | #!/usr/bin/perl
2 | # Example 11-7   Call another program to perform secondary structure prediction
3 |
4 | use strict;
5 | use warnings;
6 |
7 | # Call "stride" on a file, collect the report
8 | my (@stride_output) = call_stride('pdb/c1/pdb1c1f.ent');
9 |
10 | # Parse the stride report into primary sequence, and secondary
11 | #   structure prediction
12 | my ( $sequence, $structure ) = parse_stride(@stride_output);
13 |
14 | # Print out the beginnings of the sequence and the secondary structure
15 | print substr( $sequence, 0, 80 ), "\n";
```

⁴译者注：原链接已失效，请前往 <http://webclu.bio.wzw.tum.de/stride/>。

```
16 | print substr( $structure, 0, 80 ), "\n";
17 |
18 | exit;
19 |
20 | #####
21 | # Subroutine for Example 11-7
22 | #####
23 |
24 | # call_stride
25 | #
26 | # -given a PDB filename, return the output from the "stride"
27 | #     secondary structure prediction program
28 |
29 | sub call_stride {
30 |
31 |     use strict;
32 |     use warnings;
33 |
34 |     my ($filename) = @_ ;
35 |
36 |     # The stride program options
37 |     my ($stride) = '/usr/local/bin/stride';
38 |     my ($options) = '';
39 |     my (@results) = ();
40 |
41 |     # Check for presence of PDB file
42 |     unless ( -e $filename ) {
43 |         print "File \"$filename\" doesn't seem to exist!\n";
44 |         exit;
45 |     }
46 |
47 |     # Start up the program, capture and return the output
48 |     @results = ` $stride $options $filename `;
49 |
50 |     return @results;
51 | }
52 |
53 | # parse_stride
54 | #
55 | # -given stride output, extract the primary sequence and the
56 | #     secondary structure prediction, returning them in a
57 | #     two-element array.
58 |
59 | sub parse_stride {
60 |
61 |     use strict;
62 |     use warnings;
63 |
64 |     my (@stridereport) = @_ ;
```



```
65     my ($seq)          = '';
66     my ($str)          = '';
67     my $length;
68
69     # Extract the lines of interest
70     my (@seq) = grep( /^SEQ /, @stridereport );
71
72     my (@str) = grep( /^STR /, @stridereport );
73
74     # Process those lines to discard all but the sequence
75     # or structure information
76     for (@seq) { $_ = substr( $_, 10, 50 ) }
77     for (@str) { $_ = substr( $_, 10, 50 ) }
78
79     # Return the information as an array of two strings
80     $seq = join( ' ', @seq );
81     $str = join( ' ', @str );
82
83     # Delete unwanted spaces from the ends of the strings.
84     # ($seq has no spaces that are wanted, but $str may)
85     $seq =~ s/(\s+)$//;
86
87     $length = length($1);
88
89     $str =~ s/\s{$length}$//;
90
91     return ( ( $seq, $str ) );
92 }
```

正如你在子程序 *call_stride* 中看到的那样，针对程序名 (*\$stride*) 和你想传递的选项 (*\$options*) 我们都单独创建了变量。因为这是程序中你可能需要修改的部分，所以把它们做成变量放在代码的顶部，这样就容易找到并修改它们了。子程序的参数是 PDB 文件的文件名 (*\$filename*)。（当然，如果你认为选项会经常改变，也可以把它们做成子程序的另外一个参数。）

既然你正在处理的是一个读入文件的程序，那么需要进行一点点的错误检查来看看通过名字指定的这个文件是不是真的存在。使用 *-e* 文件测试符即可。或者你可以跳过这一步，让 *stride* 程序来检查它是否存在，然后捕获它的错误输出。但是这样的话，就需要从 *stride* 的输出中解析错误输出，而这则需要了解 *stride* 是如何报告错误的。这会让事情变得复杂起来，所以我还是坚持使用 *-e* 文件测试符。

实际上，程序的运行和收集它的输出只发生在一行代码中。需要运行的程序被包裹在反引号中，它会运行程序（首先展开变量），并且返回输出结果，存储在由每一行组成的数组中。

还有其他的方法可以运行程序。一种常见的方法是进行调用 *system* 函数。它和反引号的行为有所不同：它并不会返回它调用的命令的输出（它仅仅返回退出状态，也就是表示命令运行成功或失败的一个整数）。其他方法还包括 *qx*、*open* 系统调用，以及 *fork* 和 *exec* 函数。

11.5.2 解析 Stride 的输出

此处我不想太过深入得讲解 *stride* 输出结果的解析。我们只看一下提取一级序列和二级结构预测的代码。可以看看本章末尾的练习题，挑战一下从 PDB 文件的 HELIX、SHEET 和 TURN 记录类型中提取出二级结构信息，然后把它们输出成类似此处 *stride* 输出的格式。

下面是一个典型的 *stride* 输出的一部分（并不是全部输出）：

```

1 | SEQ  1      MDKNELVQKAKLAEQAERYDDMAACMKSVTEQGAELSNEERNLLSVAYKN      50
  | 1A40
2 | STR          HHHHHHHHHHHHHHHH  HHHHHHHHHHHHHHTTT  HHHHHHHHHHHHHHH
  | 1A40
3 | REM
  | 1A40
4 | REM          .          .          .          .          .
  | 1A40
5 | SEQ  51     VVGARRSSWRVVSIEQKEKKQQMAREYREKIETELRDICNDVLSLLEKF      100
  | 1A40
6 | STR          HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHT
  | 1A40
7 | REM
  | 1A40
8 | REM          .          .          .          .          .
  | 1A40
9 | SEQ 101     LIPNAAESKVFYFLKMKGDYYRYLAEVAAGDDKKGIVDQSQQAYQEAFEIS      150
  | 1A40
10 | STR         TTTTT HHHHHHHHHHHHHHHHHHHHHH  HHHHHHHHHHHHHHHHHHHHHH
  | 1A40
11 | REM
  | 1A40
12 | REM          .          .          .          .
  | 1A40
13 | SEQ 151     KKEMIRLGLALNFSVFYYACSLAKTAFDEAIAELLIMQLLRDNLTLW      197
  | 1A40
14 | STR         TTTTHHHHHHHHHHHHHH  HHHHHHHHHHHHHH  HHHHHHHHHHH
  | 1A40

```

注意，每一行的都是以一个识别码开始的，这使得收集不同的记录类型简单了许多。不需要去查阅文档（有点危险，但有时也是权宜之计）你就可以看出，一级序列有 SEQ 这个关键词，结构预测有 STR 这个关键词，并且我们感兴趣的数据位于每一行的第 11 列到第 60 列之间。（现在我们会把其他的都忽略掉。）

下面这个列表展示了 *stride* 使用的单字母二级结构代码：

使用 `substr` 函数，两个 `for` 循环会改变两个数组的每一行，把这些字符串的第 11 位到第 60 位之间的部分保存起来。这正是我们所需要的信息所处的位置。

现在，让我们检查一下例 11.7 中的子程序 *parse_stride*，它以 *stride* 的输出作为输入，返回一级序列和二级结构预测这两个字符串的组成的数组。

这是一个非常有“Perl 风格”的子程序，它使用了许多处理文本的特性。让人感兴趣

H	α-螺旋 (Alpha helix)
G	3-10 螺旋 (3-10 helix)
I	π-螺旋 (PI helix)
E	延伸构造 (Extended conformation)
B or b	孤立桥 (Isolated bridge)
T	转角 (Turn)
C	卷曲 (Coil, 不属于上面的任何一类)

的是程序的简洁，正是许多 Perl 内置函数使其成为了可能。

首先，你在子程序中使用参数 `@_` 获取了 *stride* 程序的输出。接着，使用 *grep* 函数提取出感兴趣的那些行，在输出中很容易就可以把它们识别出来，因为它们都起始于明确的识别符 `SEQ` 和 `STR`。

接下来，你只想把这些行中包含序列或者结构信息的位置（或列）保存起来，你并不需要关键字、位置号，一级行尾的 PDB 记录名。

最后，把数组连接成单个的字符串。此处，有一个细节需要处理：你需要把这些字符串末尾的不需要的空白全部去除掉。注意 *stride* 有时会在结构预测中留有空白，在这个例子中，结构预测的末尾就有一些空白。但是你不应该把这些字符串末尾的所有空白全部去掉，而是去掉序列字符串末尾的所有空白，因为它们仅仅是行中多余的空白而已。现在，看看序列字符串末尾有多少空白，就要丢掉结构预测字符串末尾同样数目的空白，这样就可以保留下与未确定的二级结构相对应的空白了。

例 11.7中包含一个主程序调用这两个子程序，因为子程序非常简单，就把它们都包含在内了（所以此处没有必要使用 *BeginPerlBioinfo* 模块）。下面是例 11.7的输出：

```
1 | GGLQVKNFDFTVGKFLTVGGFINNSPQRFSVNVGESMNSLSLHLDHRFNYGADQNTIVMNSTLKGDNGWETEQRSTNFTL
2 |      TTTTTTBTBT EEEEEETTTT EEEEEEEEEETEEEEEEEEEEEEETEEEEEEEEETTGGG B
   | EEE
```

第一行显示的是氨基酸，而第二行则是二级结构的预测。对于改善输出的子程序，可以参看下一小节。

11.6 练习题

练习 11.1

使用 `File::Find` 和文件测试操作符来找到你计算机硬盘上最老的和最大的文件。（当你的硬盘空间不够用的时候，你可以删除它们，或者把它们保存到别的地方去。）

练习 11.2

找到你计算机中的所有 Perl 程序。

提示：使用 `File::Find`。所有的 Perl 程序都有什么共同点？

练习 11.3

解析你计算机上所有 PDB 文件的 HEADER、TITLE 和 KEYWORDS 记录类型。制作一个散列，其键就是这些记录类型单词，值则是包含这些单词的文件名列表。把它保存为一个 DBM 文件，并且为它构建一个查询程序。最后，你应该能够在询问一个单词的情况下，比如，`sugar` 这个单词，得到在 HEADER、TITLE 或者 KEYWORDS 记录中包含这个单词的所有 PDB 文件列表。

练习 11.4

使用正则表达式（在第 10 章中使用过），而不是对由输入行组成的数组进行循环处理（本章中就是这种方法），解析出 PDB 文件的记录类型。

练习 11.5

编写一个程序，提取出 PDB 文件中 HELIX、SHEET 和 TURN 记录类型中包含的二级结构信息。同时输出二级结构和一级序列，这样就比较容易能看出某个残基是处于哪种二级结构中了。（考虑对二级结构使用一种特殊的字母表，这样的话，举个例子，螺旋中的每个残基就都可以用 H 来表示了。）

练习 11.6

编写一个程序，找到一个给定目录中的所有 PDB 文件，并且运行一个程序（比如 `stride`），或者你在练习 11.5 中编写的程序，报告每个 PDB 文件中的二级结构。把结果保存到一个 DBM 文件中，用文件名作为键。

练习 11.7

编写一个子程序，对于给定的两个字符串，把它们输出出来，让一个字符串在另一个字符串的上面，但是要有换行（类似与 `stride` 程序的输出）。使用这个模块打印出例 11.7 中的字符串。

练习 11.8

编写一个递归的子程序，来确定一个数组的大小。你可能想要使用 `pop` 或者 `unshift` 函数。（暂时忽略掉 `scalar @array` 会返回 `@array` 的大小这个事实！）

练习 11.9

编写一个递归的子程序，从一个 PDB 文件的 SEQRES 记录类型中提取出一级氨基酸序列。

练习 11.10

（额外加分）给定一个原子和距离，找到 PDB 文件中中和这个原子相距距离范围以内的所有其他原子。

练习 11.11

(额外加分) 编写一个程序，找到一级氨基酸序列和 α -螺旋定位之间的某种相关性。

第 12 章 BLAST

目录

12.1 获取 BLAST	288
12.2 字符串匹配和同源	289
12.3 BLAST 输出文件	290
12.4 解析 BLAST 输出	295
12.5 呈现数据	305
12.6 BioPerl	309
12.7 练习题	315

在生物科学研究中，查找序列的相似性是非常重要的。比如，一个研究人员发现了一个可能非常重要的 DNA 或者蛋白质序列，他想知道这条序列是否已经被别的研究人员发现并且研究表征过了。如果没有的话，这个研究人员想知道这条序列是否和某个物种中的某条序列比较类似。这些信息对于物种中这条序列的功能会提供非常重要的线索。

BLAST (Basic Local Alignment Search Tool) 是在生物科学研究中最流行的软件工具之一。对于一条查询序列，它会在一个已知序列库中进行测试，来查找相似性。BLAST 实际上是一个程序集，根据查询-数据库对的不同有不同的版本，比如核酸-核酸、蛋白质-核酸、蛋白质-蛋白质、核酸-蛋白质等等。

本章将解析这个程序的核酸-核酸版本的输出，也就是 *BLASTN* 的输出。简便起见，此处我会简单地用 BLAST 来指代它。本章的主要目标就是来演示如何编写代码使用正则表达式来解析一个 BLAST 输出文件。代码非常简单而是很基本，但是它确实可以完成这个工作。一旦你理解了基本知识，你就可以向你的解析器中添加更多的特性，或者从网上找一个更加精致的 BLAST 输出解析器。不管哪种情况，你都需要对输出解析器有足够的了解，才能去使用或者扩展它们。

本章还会对 BioPerl 进行简要的介绍，它是一个用于生物信息学的 Perl 模块集。BioPerl 项目是一个开源项目的实例，作为使用 Perl 的生物信息学程序员，你可以好好地利用它。Perl 编程语言本身就是一个开源项目。程序以及它的源代码都可以随意使用和修改，只有几个非常合理的限制，并且是免费的奥。

12.1 获取 BLAST

有许多不同的 BLAST 实现，最流行的可能是 NCBI (National Center for Biotechnology Information) 免费提供的版本：<http://www.ncbi.nlm.nih.gov/BLAST/>。NCBI 网站提供了一个可以公开使用的 BLAST 服务器、全面的数据库集和组织良好的文档和指南文集，还有可供下载的 BLAST 软件。

另外一个比较流行的实现是华盛顿大学的 WU-BLAST。包含其他 WU-BLAST 服务器列表的主网站可以在 <http://blast.wustl.edu> 找到¹。旧版本的 WU-BLAST 可以免费获取。如果你是一个研究人员或者非盈利的组织，并且同意开发和维护这个程序的华盛顿大学的许可协议，那么新版的 WU-BLAST 也可以免费获得。如果你在一个大型的研究机构工作，你可能已经有了一个 WU-BLAST 程序的站点许可证了。如果你是一个营利性的公司，那么就需要为新版本的 WU-BLAST 程序支付一笔非常昂贵的费用了（如果你想在自己的计算机上运行 BLAST，旧版本的程序仍然是免费的）。宾夕法尼亚州立大学也开发了一些 BLAST 程序，可以在 <http://bio.cse.psu.edu/> 找到²。除了 NCBI 和 WU-BLAST，还有许多其他可以使用的 BLAST 服务器网站。在谷歌 (Google, <http://www.google.com>) 中搜索 “BLAST server” 就能得到很多结果。

当研究人员使用 BLAST 的时候，它们面临的一个大问题就是是在公用的 BLAST 服务器上运行呢，还是在本地运行呢。使用公用的服务器，有一些显著的优势，最大的优势在于 BLAST 服务器使用的数据库（比如 GenBank）总是最新的。要让你自己的这些数据库时刻保持最新，需要大量的硬盘空间，以及拥有高端处理器、大量内存和高速网络连接的计算机，还需要花费大量的时间来设置并监视更新数据库的软件。另一方面，可能你有自己的序列库，想用它来进行 BLAST 搜索，你需要经常搜索或者进行大量的搜索，又或者你有一些原因不得不使用内部自己的 BLAST 引擎。这种情况下，对硬件进行投资、在本地运行它就是比较明智的了。

BLAST 的在线文档非常详尽，包括了程序用来计算相似度的统计方法的细节内容。在接下来的小节中，我会对这些进行简单的介绍，但是你应该到 BLAST 的主页上以及 NCBI 的网站上找到这些优秀的材料，从头到尾把整个内容通读一下，详读需要查阅的内容。此处我们的兴趣并不在于理论知识，而是解析程序的输出。

¹译者注：最新版本是 2009-10-30 发布的 AB-BLAST (<http://blast.advbiocomp.com/>)。

²译者注：最新版本是 2010-01-12 发布的 LASTZ (http://www.bx.psu.edu/miller_lab)。

12.2 字符串匹配和同源

字符串匹配是计算机科学中的术语，指在另一个字符串中找到某个嵌入在内的字符串的算法。它有一个悠久且成果卓著的历史，使用不同的技术，开发出了许多字符串匹配算法，用于各种不同的情况。（参看附录 A 中 Gusfield 的书对其进行了精彩的讲解，且侧重点在于生物学领域。）我们已经进行了不少的字符串匹配，使用绑定操作符用正则表达式来查找基序和其他的文本。

BLAST 是一个基本的字符串匹配程序。字符串匹配算法的细节，以及 BLAST 中使用的算法，已经超出了本书的范畴。但是首先我想来定义几个常常被混淆或者混用的术语。此外，我还会对 BLAST 涉及的统计进行一个简要的介绍。

生物学的字符串匹配用来查找相似，它是同源的一个指标。查询序列和数据库中序列的相似度 (*similarity*) 可以用百分同一性 (*percent identity*) 来衡量，或者是查询序列和数据库中一个序列对应区域完全匹配的碱基数目。它也可以用保守 (*conservation*) 的程度来衡量，它会找到等价 (冗余) 密码子或者不改变蛋白质功能的具有相似属性的氨基酸残基之间的匹配。（参看第 8 章）。序列之间同源 (*homology*) 表示序列在进化上是相关的。两个序列要么同源，要么不同源，没有同源度这种说法。³

冒着把一个复杂主题过度简化的风险，我要对 BLAST 的统计中的一些方面进行总结。（完整的细节可以参看 BLAST 的文档。）BLAST 搜索的输出中会报告它找到的匹配的一些值和统计信息，主要基于原始值 *S*、打分算法的参数，以及查询和数据库的属性。原始值 *S* (*raw score S*) 是对相似性和匹配大小的一种度量。BLAST 的输出罗列了按照 *E* 值排序的击中 (*hit*)。粗略来说，一个匹配的 *E* 值/期望值 (*E value, expect value*) 衡量的是在一个随机生成的具有同样大小和组成的数据库中字符串匹配（允许空位）的几率。*E* 值越接近 0，这个匹配越不可能随机发生。换言之，*E* 值越小，匹配越好。作为 BLASTN 的一个经验准则，*E* 值小于 1 的可能是一个比较可靠的击中，*E* 值小于 10 的可能还需要看看，但这并不是一个硬性规定。（当然，对于蛋白质来说，即使只有很小的百分同一性，也可能是同源的；它的相似度百分比通常比同源 DNA 要高。）

现在，既然你已经找我的基础知识，就让我们编写代码来解析 BLAST 的输出吧。首先，你需要把击中分隔开，然后提取出序列，最后找到注释把 *E* 值这个统计量显示出来。

³译者注：相似不一定同源。

12.3 BLAST 输出文件

下面是 BLAST 输出文件的一部分。通过把第 8 章中 *sample.dna* 这个文件的几行输入到 NCBI 网站的 BLAST 程序中，在使用默认参数的情况下得到了这个文件。然后我把输出以文本形式保存到 *blast.txt* 文件中，在本书的网站上可以找到它。在贯穿本章的解析工作中，我会重复使用它。因为输出有数页之长，此处我把它截断了，只显示文件的开头、中间和结尾部分。

```

1 | BLASTN 2.1.3 [Apr-11-2001]
2 |
3 | Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
4 | Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
5 | "Gapped BLAST and PSI-BLAST: a new generation of protein database search
6 | programs", Nucleic Acids Res. 25:3389-3402.
7 | RID: 991533563-27495-9092
8 | Query=
9 |         (400 letters)
10 |
11 | Database: nt
12 |         868,831 sequences; 3,298,558,333 total letters
13 |
14 |                                     Score
15 | E
16 | Sequences producing significant alignments: (bits)
17 | Value
18 | dbj|AB031069.1|AB031069 Homo sapiens PCCX1 mRNA for protein cont... 793
19 | 0.0
20 | ref|NM_014593.1| Homo sapiens CpG binding protein (CGBP), mRNA 779
21 | 0.0
22 | gb|AF149758.1|AF149758 Homo sapiens CpG binding protein (CGBP) m... 779
23 | 0.0
24 | ref|XM_008699.3| Homo sapiens CpG binding protein (CGBP), mRNA 765
25 | 0.0
26 | emb|AL136862.1|HSM801830 Homo sapiens mRNA; cDNA DKFZp434F174 (f... 450
27 | e-124
28 | emb|AJ132339.1|HSA132339 Homo sapiens CpG island sequence, subcl... 446
29 | e-123
30 | emb|AJ236590.1|HSA236590 Homo sapiens chromosome 18 CpG island D... 406
31 | e-111
32 | dbj|AK010337.1|AK010337 Mus musculus ES cells cDNA, RIKEN full-l... 234
33 | 3e-59
34 | dbj|AK017941.1|AK017941 Mus musculus adult male thymus cDNA, RIK... 210
35 | 5e-52
36 | gb|AC009750.7|AC009750 Drosophila melanogaster, chromosome 2L, r... 46
37 | 0.017
38 | gb|AE003580.2|AE003580 Drosophila melanogaster genomic scaffold ... 46

```

```
| 0.017
28 | ref|NC_001905.1| Leishmania major chromosome 1, complete sequence      40
| 1.0
29 | gb|AE001274.1|AE001274 Leishmania major chromosome 1, complete s...    40
| 1.0
30 | gb|AC008299.5|AC008299 Drosophila melanogaster, chromosome 3R, r...    38
| 4.1
31 | gb|AC018662.3|AC018662 Human Chromosome 7 clone RP11-339C9, comp...    38
| 4.1
32 | gb|AE003774.2|AE003774 Drosophila melanogaster genomic scaffold ...    38
| 4.1
33 | gb|AC008039.1|AC008039 Homo sapiens clone SCb-391H5 from 7q31, c...    38
| 4.1
34 | gb|AC005315.2|AC005315 Arabidopsis thaliana chromosome II sectio...    38
| 4.1
35 | emb|AL353748.13|AL353748 Human DNA sequence from clone RP11-317B...    38
| 4.1
36
37 ALIGNMENTS
38 >dbj|AB031069.1|AB031069 Homo sapiens PCCX1 mRNA for protein containing CXXC
39 domain 1,
40     complete cds
41     Length = 2487
42
43 Score = 793 bits (400), Expect = 0.0
44 Identities = 400/400 (100%)
45 Strand = Plus / Plus
46
47 Query: 1   agatggcggcgctgaggggtcttgggggctctaggccggccacctactggtttgcagcgg 60
48           |||
49 Sbjct: 1   agatggcggcgctgaggggtcttgggggctctaggccggccacctactggtttgcagcgg 60
50
51 Query: 61   agacgacgcatggggcctgcgcaataggagtacgctgcctgggaggcgtgactagaagcg 120
52           |||
53 Sbjct: 61   agacgacgcatggggcctgcgcaataggagtacgctgcctgggaggcgtgactagaagcg 120
54
55 Query: 121  gaagtagttgtgggcgcctttgcaaccgcctgggacgccgccgagtggtctgtgcaggtt 180
56           |||
57 Sbjct: 121  gaagtagttgtgggcgcctttgcaaccgcctgggacgccgccgagtggtctgtgcaggtt 180
58
59 Query: 181  cgcgggctcgctggcgggggctcgtgagggagtgcgccgggagcggagatatggagggagat 240
60           |||
61 Sbjct: 181  cgcgggctcgctggcgggggctcgtgagggagtgcgccgggagcggagatatggagggagat 240
62
63 Query: 241  ggttcagacccagagcctccagatgccggggaggacagcaagtccgagaatggggagaat 300
64           |||
65 Sbjct: 241  ggttcagacccagagcctccagatgccggggaggacagcaagtccgagaatggggagaat 300
66
67 Query: 301  gcgcccactctactgcatctgccgcaaaccggacatcaactgcttcatgatcgggtgtgac 360
```

```
68      |||
69 Sbjct: 301 gcgcccactctactgcatctgccgcaaaccggacatcaactgcttcatgatcgggtgtgac 360
70
71 Query: 361 aactgcaatgagtgggttccatggggactgcatccggatca 400
72      |||
73 Sbjct: 361 aactgcaatgagtgggttccatggggactgcatccggatca 400
74
75 >ref|NM_014593.1| Homo sapiens CpG binding protein (CGBP), mRNA
76
77
78 ... (file truncated here)
79
80
81 >dbj|AK010337.1|AK010337 Mus musculus ES cells cDNA, RIKEN full-length
82 enriched library,
83     clone:2410002I16, full insert sequence
84     Length = 2538
85
86 Score = 234 bits (118), Expect = 3e-59
87 Identities = 166/182 (91%)
88 Strand = Plus / Plus
89
90 Query: 219 gagcggagatatggagggagatgggttcagacccagagcctccagatgccggggaggacag 278
91      |||
92 Sbjct: 260 gagcggagatatggaaggagatggctcagacctggaacctccggatgccggggacgacag 319
93
94 Query: 279 caagtccgagaatggggagaaatgcgcccactctactgcatctgccgcaaaccggacatcaa 338
95      |||
96 Sbjct: 320 caagtctgagaatggggagaaacgctcccatctactgcatctgtcgcaaaccggacatcaa 379
97
98 Query: 339 ctgcttcatgatcgggtgtgacaactgcaatgagtgggttccatggggactgcatccggat 398
99      |||
100 Sbjct: 380 ttgcttcatgattggatgtgacaactgcaacgagtgggttccatggagactgcatccggat 439
101
102 Query: 399 ca 400
103      ||
104 Sbjct: 440 ca 441
105 Score = 44.1 bits (22), Expect = 0.066
106 Identities = 25/26 (96%)
107 Strand = Plus / Plus
108
109 Query: 118 gcggaagtagttgtgggcgcctttgc 143
110      |||
111 Sbjct: 147 gcggaagtagttgcgggcgcctttgc 172
112
113 >dbj|AK017941.1|AK017941 Mus musculus adult male thymus cDNA, RIKEN
114 full-length enriched library, clone:5830420C16, full insert sequence
115     Length = 1461
116
```

```
117 | Score = 210 bits (106), Expect = 5e-52
118 | Identities = 151/166 (90%)
119 | Strand = Plus / Plus
120 |
121 | Query: 235 ggagatgggttcagacccagagcctccagatgccggggaggacagcaagtccgagaatggg 294
122 |          ||||| ||||| || ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
123 | Sbjct: 1048 ggagatggctcagacctggaacctccggatgccggggacgacagcaagtctgagaatggg 1107
124 |
125 | Query: 295 gagaatgcgcccactctactgcatctgccgcaaaccggacatcaactgcttcatgatcggg 354
126 |          ||||| || ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||
127 | Sbjct: 1108 gagaacgctcccatctactgcatctgtcgcaaaccggacatcaattgcttcatgattgga 1167
128 |
129 | Query: 355 tgtgacaactgcaatgagtgggtccatggggactgcatccggatca 400
130 |          ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
131 | Sbjct: 1168 tgtgacaactgcaacgagtgggtccatggagactgcatccggatca 1213
132 |
133 | Score = 44.1 bits (22), Expect = 0.066
134 | Identities = 25/26 (96%)
135 | Strand = Plus / Plus
136 |
137 | Query: 118 gcggaagtagttgtgggcgcctttgc 143
138 |          ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
139 | Sbjct: 235 gcggaagtagttgcgggcgcctttgc 260
140 |
141 | >gb|AC009750.7|AC009750 Drosophila melanogaster, chromosome 2L, region 23F-24A,
142 | BAC clone
143 |
144 |
145 | ... (file truncated here)
146 |
147 |
148 | >emb|AL353748.13|AL353748 Human DNA sequence from clone RP11-317B17 on
149 | chromosome 9, complete
150 |     sequence [Homo sapiens]
151 |     Length = 179155
152 |
153 | Score = 38.2 bits (19), Expect = 4.1
154 | Identities = 22/23 (95%)
155 | Strand = Plus / Plus
156 |
157 | Query: 192 ggcgggggctcgtgagggagtgcg 214
158 |          |||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
159 | Sbjct: 48258 ggcgtgggctcgtgagggagtgcg 48280
160 |
161 | Database: nt
162 | Posted date: May 30, 2001 3:54 AM
163 | Number of letters in database: -996,408,959
164 | Number of sequences in database: 868,831
165 |
```

```
166 Lambda      K      H
167      1.37      0.711      1.31
168
169 Gapped
170 Lambda      K      H
171      1.37      0.711      1.31
172
173 Matrix: blastn matrix:1 -3
174 Gap Penalties: Existence: 5, Extension: 2
175 Number of Hits to DB: 436021
176 Number of Sequences: 868831
177 Number of extensions: 436021
178 Number of successful extensions: 7536
179 Number of sequences better than 10.0: 19
180 length of query: 400
181 length of database: 3,298,558,333
182 effective HSP length: 20
183 effective length of query: 380
184 effective length of database: 3,281,181,713
185 effective search space: 1246849050940
186 effective search space used: 1246849050940
187 T: 0
188 A: 30
189 X1: 6 (11.9 bits)
190 X2: 15 (29.7 bits)
191 S1: 12 (24.3 bits)
192 S2: 19 (38.2 bits)
```

如你所见，文件包含了三大部分：在开头的是一些头信息，之后是对比对的总结和详细的比对信息，末尾是附加的一些参数和统计信息总结。

12.4 解析 BLAST 输出

那么为什么要解析 BLAST 的输出呢？一个原因是看看你的 DNA 在持续增长数据库中是否有新的匹配。你可以编写一个程序来自动执行，每天进行一次 BLAST 查找，然后通过解析击中的总结列表来和前一天的总结列表进行比较，对比它的结果和前一天的结果。如果有新的结果出现，你也可以让程序发送邮件给你。

12.4.1 提取注释和比对

例 12.1 由一个主程序和两个新的子程序组成。*parse_blast* 和 *parse_blast_alignment* 这两个子程序使用正则表达式来从标量字符串中提取大量的数据位。我之所以选取这种方法，是因为数据本身，数据虽然是结构化的，但是每一行并没有明确指定的功能。（参看第 10 章和第 11 章中的讨论。）

例 12.1：从 BLAST 输出文件中提取注释和比对

```

1  #!/usr/bin/perl
2  # Example 12-1   Extract annotation and alignments from BLAST output file
3
4  use strict;
5  use warnings;
6  use BeginPerlBioinfo;    # see Chapter 6 about this module
7
8  # declare and initialize variables
9  my $beginning_annotation = '';
10 my $ending_annotation    = '';
11 my %alignments           = ();
12 my $filename             = 'blast.txt';
13
14 parse_blast( \$beginning_annotation, \$ending_annotation, \%alignments,
15             $filename );
16
17 # Print the annotation, and then
18 #   print the DNA in new format just to check if we got it okay.
19 print $beginning_annotation;
20
21 foreach my $key ( keys %alignments ) {
22     print "$key\nXXXXXXXXXXXX\n", $alignments{$key}, "\nXXXXXXXXXXXX\n";
23 }
24
25 print $ending_annotation;
26
27 exit;
28
29 #####
30 # Subroutines for Example 12-1
31 #####
32

```

```
33 # parse_blast
34 #
35 # -parse beginning and ending annotation, and alignments,
36 #     from BLAST output file
37
38 sub parse_blast {
39
40     my ( $beginning_annotation, $ending_annotation, $alignments, $filename ) =
41         @_;
42
43     # $beginning_annotation-reference to scalar
44     # $ending_annotation    -reference to scalar
45     # $alignments           -reference to hash
46     # $filename             -scalar
47
48     # declare and initialize variables
49     my $blast_output_file = '';
50     my $alignment_section = '';
51
52     # Get the BLAST program output into an array from a file
53     $blast_output_file = join( ' ', get_file_data($filename) );
54
55     # Extract the beginning annotation, alignments, and ending annotation
56     ( $$beginning_annotation, $alignment_section, $$ending_annotation ) =
57         ( $blast_output_file =~ /(.*^ALIGNMENTS\n)(.*)"(^ Database:.*)/ms );
58
59     # Populate %alignments hash
60     # key = ID of hit
61     # value = alignment section
62     %$alignments = parse_blast_alignment($alignment_section);
63 }
64
65 # parse_blast_alignment
66 #
67 # -parse the alignments from a BLAST output file,
68 #     return hash with
69 #     key = ID
70 #     value = text of alignment
71
72 sub parse_blast_alignment {
73
74     my ($alignment_section) = @_;
75
76     # declare and initialize variables
77     my (%alignment_hash) = ();
78
79     # loop through the scalar containing the BLAST alignments,
80     # extracting the ID and the alignment and storing in a hash
81     #
```

```

82 | # The regular expression matches a line beginning with >,
83 | # and containing the ID between the first pair of | characters;
84 | # followed by any number of lines that don't begin with >
85 |
86 | while ( $alignment_section =~ /^>.*\n^(?!>).*\n+/gm ) {
87 |     my ($value) = $&;
88 |     my ($key) = ( split( /\|/, $value ) )[1];
89 |     $alignment_hash{$key} = $value;
90 | }
91 |
92 | return %alignment_hash;
93 | }

```

主程序仅仅是调用了用来解析的子程序，并且输出结果而已。初始值为空的参数，是通过指针进行传递的。（参看第 6 章）。

子程序 *parse_blast* 进行的是顶层解析的工作，它把 BLAST 输出文件的三部分分割了开来：开头的注释部分，中间的比对部分，和末尾的注释部分。然后，它调用了 *parse_blast_alignment* 子程序来从中间的比对部分中提取出单个的比对。我们首先使用我们的老朋友——第 8 章中的 *get_file_data* 子程序从指定的文件中读取数据。用 *join* 函数把数组中的文件数据存储到一个标量字符串中。

BLAST 输出文件中的三大部分是通过下面的语句分割开的：

```

1 | ($$beginning_annotation, $alignment_section, $$ending_annotation)
2 | = ($blast_output_file =~ /(.*^ALIGNMENTS\n)(.*)"(^ Database:.*)/ms);

```

模式匹配包含了三个用小括号括起来的表达式：

```

1 | (.*^ALIGNMENTS\n)
   | 会返回到 $$beginning_annotation;，

```

```

1 | (.*)
   | 会保存到 $alignment_section;，最后：

```

```

1 | (^ Database:.*)
   | 会保存到 $$ending_annotation。

```

在三个标量中有两个变量的开头使用的是 \$\$ 而非 \$，这表明它们是标量变量的指针。回忆一下当它们作为参数传递给子程序的时候，在它们的前面都有一个斜杠，就像这样：

```

1 | parse_blast(\$beginning_annotation, \$ending_annotation, \%alignments, $filename);

```

从第 6 章开始，我们就已经见过变量的指针了。让我们简单复习一下。在 *parse_blast* 子程序中，只有一个 \$ 的这些变量都是标量变量的指针。如果要表示真正的标量变量，还需要额外的一个 \$。指针就是这么使用的，它们需要额外的一个特殊字符来表明它们指代的到底是哪种类型的变量。所以一个标量变量的指针需要用 \$\$ 来起始，一个数组变量的指针需要用 @\$ 来起始，一个散列变量的指针需要用 %\$ 来起始。

前面代码片段中的正则表达式中, `(.*^ALIGNMENTS\n)` 会匹配所有内容一直到行尾的 `ALIGNMENTS` 单词; 然后, `(.*)` 匹配所有内容; 之后 `(^ Database:.*)` 会匹配以两个空格和 `Database` 单词起始的一行, 以及文件之后的所有剩余内容。小括号中的这三个表达式正好对应 BLAST 输出文件中的那三大部分: 开头的注释, 比对部分, 以及结尾的注释。

保存在 `$alignment_section` 变量中的比对部分用子程序 `parse_blast_alignment` 来进行分割。子程序中有一个重要的循环:

```
1 while($alignment_section =~ /^>.*\n(?:?!>).*\n)/gm) {
2   my($value) = $&;
3   my($key) = (split(/\|/, $value)) [1];
4   $alignment_hash{$key} = $value;
5 }
```

你可能会认为这个正则表达式实在是糟糕透顶了。第一眼看上去, 正则表达式所做的事情是在让人费解, 所以让我们仔细看一下。有几个新的东西需要学习一下。

这五行代码组成了一个 `while` 循环, 它会持续尽可能多地去匹配字符串中出现的模式 (这要归因于 `while` 循环中模式匹配的 `/g` 全局修饰符)。程序每次循环的时候, 模式匹配都会找到它的值 (也就是整个比对), 进而确定键。键和值都保存到散列 `%alignment_hash` 中。

当解析第 12.3 节中的 BLAST 输出时, 下面是这个 `while` 循环找到的一个匹配的例子:

```
1 >emb|AL353748.13|AL353748 Human DNA sequence from clone RP11-317B17 on
2 chromosome 9, complete
3         sequence [Homo sapiens]
4         Length = 179155
5
6 Score = 38.2 bits (19), Expect = 4.1
7 Identities = 22/23 (95%)
8 Strand = Plus / Plus
9
10 Query: 192   ggcggggggtcgtgagggagtgcg 214
11           |||| |
12 Sbjct: 48258 ggcggtgggtcgtgagggagtgcg 48280
```

这段文字起始于以一个 `>` 字符开始的行。在完整的 BLAST 输出中, 像这样的部分一个接一个。你想做的是从以 `>` 起始的一行开始匹配, 把随后相邻的所有行都包含在内, 但是不包括以 `>` 字符起始的行。你还想提取出识别符, 它出现在第一行 (比如, 在这个比对中就是 `AL353748.13`) 的前两个竖线 `|` 字符之间。

让我们来剖析一下这个正则表达式:

```
1 $alignment_section =~ /^>.*\n(?:?!>).*\n)/gm
```

出现在代码的 `while` 循环中的这个模式匹配, 有助于多行匹配的 `m` 修饰符。 `m` 修饰符允许 `^` 去匹配多行字符串中任意的行开头, 也允许 `$` 去匹配任意的行结尾。

正则表达式可以如下进行分解。第一部分是：

```
1 | ^>.*\n
```

它会寻找 BLAST 输出中以 > 起始的行，后面跟着 .*，它会匹配任意数量的任意字符（换行符除外），一直到第一个换行符为止。换言之，它匹配比对的第一行。

下面是正则表达式的剩余部分：

```
1 | (^(!>).*\n)+
```

在你见过 ^ 会匹配行首之后，你将看到否定性前瞻断言/前向否定断言/前向否定匹配 (*negative lookahead assertion*)，(!>)，它会确保不会紧跟着一个 >。接下来，.* 匹配非换行符的所有字符，直到行尾的最后 \n。所有的这些都被包裹在小括号中，并且使用了 +，所有它会匹配所有符合要求的行。

现在，既然你已经匹配了整个的比对，你想把键提出来，用你的键和价值来填充散列。在 while 循环中，你刚刚匹配的比对会自动被 Perl 设置成特殊变量 \$& 的值，把它保存到 \$value 变量中。现在，你需要从比对中提取出你的键来。可以在保存到 \$value 的比对的第一行中找到它，位于第一个和第二个 | 符号之间。

使用 split 函数就可以提取出整个用于识别的键，它会把字符串打断成一个数组。split 的调用：

```
1 | split(/\|/, $value)
```

把 \$value 按照 | 字符打断成了多个片段。也就是说，| 符号用来决定一个列表元素终止和下一个元素起始的位置。（记住，竖线 | 是一个元字符，必须像 \| 这样对它进行转义。）通过用小括号把对 split 的调用包裹起来，然后添加一个数组偏移量 ([1])，你可以把键分离出来，保存到 \$key 中。

现在让我们后退一步，来整体看一下例 12.1。注意，它非常简短——只有两页多一点，这还把注释也算在内了。尽管这并不是一个简单的程序，因为它里面使用了复杂的正则表达式，但如果你能够在 BLAST 输出文件和解析它的正则表达式上花点功夫，你还是能够理解它的。

正则表达式有许多复杂的特性，也正因为如此，它们可以做大量有用的事情。作为一个 Perl 程序员，你花在学习它们上的努力是非常值得的，在以后的编程之路上会给你巨额的回报。

12.4.2 解析 BLAST 比对

让我们把对 BLAST 输出文件的解析更进一步。注意，有些比对包含不知一个比对字符串——比如，ID 为 AK017941.1 的比对，再次展示如下：

```
1 | >dbj|AK017941.1|AK017941 Mus musculus adult male thymus cDNA, RIKEN
2 | full-length enriched
3 |     library, clone:5830420C16, full insert sequence
4 |     Length = 1461
5 |
```

```

6 | Score = 210 bits (106), Expect = 5e-52
7 | Identities = 151/166 (90%)
8 | Strand = Plus / Plus
9 |
10 | Query: 235 ggagatgggttcagacccagagcctccagatgccggggaggacagcaagtccgagaatggg 294
11 |          ||||| ||||| || ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
12 | Sbjct: 1048 ggagatggctcagacctggaacctccggatgccggggacgacagcaagtctgagaatggg 1107
13 |
14 | Query: 295 gagaatgcgcccactctactgcatctgccgcaaaccggacatcaactgcttcatgatcggg 354
15 |          ||||| || ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
16 | Sbjct: 1108 gagaacgctcccatctactgcatctgtcgcaaaccggacatcaattgcttcatgattgga 1167
17 |
18 | Query: 355 tgtgacaactgcaatgagtgggtccatggggactgcatccggatca 400
19 |          ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
20 | Sbjct: 1168 tgtgacaactgcaacgagtgggtccatggagactgcatccggatca 1213
21 |
22 | Score = 44.1 bits (22), Expect = 0.066
23 | Identities = 25/26 (96%)
24 | Strand = Plus / Plus
25 |
26 | Query: 118 gcggaagtagttgtgggcgcctttgc 143
27 |          ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| ||||| |||||
28 | Sbjct: 235 gcggaagtagttgcgggcgcctttgc 260

```

要解析这样的比对，我们必须把每一个匹配的字符串都解析出来，在 BLAST 中它的专用术语叫做高分值片段对 (*high-scoring pairs, HSPs*)。

每一个 HSP 也都包含一些注释，然后才是 HSP 本身。让我们把每一个 HSP 解析成注释、查询字符串和主题字符串，以及字符串起始和终止的位置。更多的解析也是有可能的，比如，你可以提取出注释中的特定特征，以及 HSP 中相同和不同碱基的位置。

例 12.2 包含了一对子程序：第一个把比对解析成它们的 HSPs，第二个提取出序列和它们终止的位置。主程序对例 12.1 进行了扩展，使用了这两个新的子程序。

例 12.2：从 BLAST 输出文件中解析比对

```

1 | #!/usr/bin/perl
2 | # Example 12-2 Parse alignments from BLAST output file
3 |
4 | use strict;
5 | use warnings;
6 | use BeginPerlBioinfo; # see Chapter 6 about this module
7 |
8 | # declare and initialize variables
9 | my $beginning_annotation = '';
10 | my $ending_annotation = '';
11 | my %alignments = ();
12 | my $alignment = '';
13 | my $filename = 'blast.txt';
14 | my @HSPs = ();
15 | my ( $expect, $query, $query_range, $subject, $subject_range ) =

```

```

16  ( ' ', ' ', ' ', ' ', ' ' );
17
18  parse_blast( \beginning_annotation, \ending_annotation, \%alignments,
19  \filename );
20
21  $alignment = $alignments{'AK017941.1'};
22
23  @HSPs = parse_blast_alignment_HSP($alignment);
24
25  ( $expect, $query, $query_range, $subject, $subject_range ) =
26  extract_HSP_information( $HSPs[1] );
27
28  # Print the results
29  print "\n-> Expect value:  $expect\n";
30  print "\n-> Query string:  $query\n";
31  print "\n-> Query range:  $query_range\n";
32  print "\n-> Subject String: $subject\n";
33  print "\n-> Subject range: $subject_range\n";
34
35  exit;
36
37  #####
38  # Subroutines for Example 12-2
39  #####
40
41  # parse_blast_alignment_HSP
42  #
43  # -parse beginning annotation, and HSPs,
44  #   from BLAST alignment
45  #   Return an array with first element set to the beginning annotation,
46  #   and each successive element set to an HSP
47
48  sub parse_blast_alignment_HSP {
49
50      my ($alignment) = @_;
51
52      # declare and initialize variables
53      my $beginning_annotation = '';
54      my $HSP_section          = '';
55      my @HSPs                 = ();
56
57      # Extract the beginning annotation and HSPs
58      ( $beginning_annotation, $HSP_section ) =
59      ( $alignment =~ /(.*?)(^ Score =.+)/ms );
60
61      # Store the $beginning_annotation as the first entry in @HSPs
62      push( @HSPs, $beginning_annotation );
63
64      # Parse the HSPs, store each HSP as an element in @HSPs

```

```

65     while ( $HSP_section =~ /(^\s*Score =.*\n)(^(!\s*Score =).*\n)+/gm ) {
66         push( @HSPs, $& );
67     }
68
69     # Return an array with first element = the beginning annotation,
70     # and each successive element = an HSP
71     return (@HSPs);
72 }
73
74 # extract_HSP_information
75 #
76 # -parse a HSP from a BLAST output alignment section
77 #     - return array with elements:
78 #     Expect value
79 #     Query string
80 #     Query range
81 #     Subject string
82 #     Subject range
83
84 sub extract_HSP_information {
85
86     my ($HSP) = @_;
87
88     # declare and initialize variables
89     my ($expect)      = '';
90     my ($query)       = '';
91     my ($query_range) = '';
92     my ($subject)     = '';
93     my ($subject_range) = '';
94
95     ($expect) = ( $HSP =~ /Expect = (\S+)/ );
96
97     $query = join( ' ', ( $HSP =~ /^Query(.*)\n/gm ) );
98
99     $subject = join( ' ', ( $HSP =~ /^Sbjct(.*)\n/gm ) );
100
101     $query_range = join( '...', ( $query =~ /(\d+).*\D(\d+)/s ) );
102
103     $subject_range = join( '...', ( $subject =~ /(\d+).*\D(\d+)/s ) );
104
105     $query =~ s/[^acgt]//g;
106
107     $subject =~ s/[^acgt]//g;
108
109     return ( $expect, $query, $query_range, $subject, $subject_range );
110 }

```

例 12.2给出如下的输出：

```

1 -> Expect value: 5e-52
2
3 -> Query string: ggagatgggttcagacccagagcctccagatgccggggaggacagcaagtccgagaatggg
4 gagaatgcgcccactctactgcatctgccgcaaaccggacatcaactgcttcattgatcggggtgtgacaactgcaatgagt
5 ggttccatggggactgcatccggatca
6
7 -> Query range: 235..400
8
9 -> Subject String: ggagatggctcagacctggaacctccggatgccggggacgacagcaagtctgagaatggg
10 gagaacgctcccactctactgcatctgtcgcaaaccggacatcaattgcttcattgattggatgtgacaactgcaacgagt
11 ggttccatgggagactgcatccggatca
12
13 -> Subject range: 1048..1213

```

让我们讨论一下例 12.2 和它子程序的新特性。首先注意，例 12.1 中的两个新子程序已经放到了 *BeginPerlBioinfo.pm* 模块里面，所以就不需要再次在此处把它们打印出来了。

例 12.2 的主程序，开头部分和例 12.1 完全一样。它调用 *parse_blast* 子程序来把 BLAST 输出文件中的注释和比对分割开来。

接下来的一行从 `%alignment` 散列中提取出一个比对，随后它就被用作了 *parse_blast_alignment_HSP* 子程序的参数，这个子程序会返回由注释（第一个元素）和 HSPs 组成的数组 `@HSPs`。

最后，例 12.2 通过调用 *extract_HSP_information* 子程序对单个的 HSP 进行了更低层次的解析，并且把从一个 HSP 中提取的各部分打印了出来。

例 12.2 演示的和我们的设计互相矛盾。有些子程序通过指针调用它们的参数，而其他的则通过值调用它们（参看第 6 章）。你可能会问：这样是不是不太好呢？

答案是：不一定。子程序 *parse_blast* 混合了好几个参数，而且其中一个并不是标量类型。回忆一下，在 Perl 中，这可能是使用指针进行调用的一个好地方。其他的子程序并没像它这样混合参数的类型。然而，也可以设计来通过指针调用它们的参数。

继续讨论我们的代码，来看一下子程序 *parse_blast_alignment_HSP*。它处理 BLAST 输出中的一个比对，把单个的 HSP 字符串匹配分割开来。此处使用的技术还是正则表达式，在一个单个的包含所有比对行的字符串上使用正则表达式，这个比对是作为输入参数指定的。

第一个正则表达式解析出注释以及包含 HSPs 的部分：

```

1 | ($beginning_annotation, $HSP_section )
2 |
3 | = ($alignment =~ /(.*?)(^ Score =.*)/ms);

```

正则表达式中的第一个小括号是 `(.*?)`。这是在第 9 章中提到的非贪婪匹配或者最小化匹配，它会匹配尽可能短的字符串。默认 `*` 是贪婪的，会匹配尽可能长的字符串。此处，它会匹配第一行以 `Score =` 开头的行之前的所有内容（如果没有 `*` 后面的 `?` 的话，它会匹配最后一行以 `Score =` 开头的行之前的所有内容）。这正好是开头的注释和 HSP 字符串匹配之间的分隔行。

下一个循环和正则表达式把单个的 HSP 字符串匹配分割开来：

```

1 | while($HSP_section =~ /(^ Score =.*\n)(^(! Score =).*\n)/gm) {
2 |

```

```
3 | push(@HSPs, $&);  
4 |  
5 | }
```

这和你前面看到的 `while` 循环类型是一样的，都是全局字符串匹配，只要能找到匹配，它就会持续进行循环。另外的修饰符 `\m` 是多行修饰符，它让元字符 `$` 和 `^` 可以匹配嵌入在内部的换行符之前和之后的位置。

第一对小括号中的表达式——`(^ Score =.*\n)`——匹配以 `Score =` 起始的一行，这正是引起 HSP 字符串匹配部分的行。

第二对小括号中的代码——`(^(?! Score =).*\n)+`——匹配不以 `Score =` 起始的一行或多行（因为在小括号外面紧跟着 `+`）。小括号括起来的部分的开头是 `?!`，它就是你在例 12.2 中遇到的否定性前瞻断言。所以，总体来看，正则表达式会捕获以 `Score =` 起始的行，以及随后的不以 `Score =` 起始的相邻行。

12.5 呈现数据

直到现在为止，我们都还依赖于 *print* 语句来格式化输出。在本节中，我将介绍打印输出的其他三个 Perl 特性：

- *printf* 函数
- *here* 文档
- *format* 和 *write* 函数

关于这些 Perl 输出特性的完整内容，已经超出了本书的范畴，但我还是会告诉你足够多的知识，让你对它们的用法有一个基本的了解。

12.5.1 printf 函数

printf 函数和 *print* 函数非常类似，但是有一些额外的特性，可以让你指定如何输出特定的数据。Perl 的 *printf* 函数是从 C 语言中同名的函数借鉴而来的。下面是 *printf* 语句的一个例子：

```
1 my $first  = '3.14159265';
2 my $second = 76;
3 my $third  = "Hello world!";
4
5 printf STDOUT "A float: %6.4f An integer: %-5d and a string: %s\n",
6     $first, $second, $third;
```

这个代码片段会输出如下内容：

```
1 | A float:  3.1416 An integer: 76    and a string: Hello world!
```

printf 函数的参数包括一个格式字符串，后面紧跟着一个值列表，这些值会按照格式字符串指定的格式输出出来。格式字符串除了指定值列表输出格式的指令外，也可能包括任意的文本。（你也可能会指定一个可选的文件句柄，它的使用方式和在 *print* 函数中的使用方式是完全一样的。）

指令由一个百分号以及紧随其后的必需的转换指定符组成，刚才例子中的转换指定符有用于浮点数的 *f*、用于整数的 *d* 和用于字符串的 *s*。转换指定符指明了变量中的数据该以何种类型输出出来。在 *%* 和转换指定符之间，可能还会有 0 个或者多个标识，一个可选的最小字段宽度，一个可选的精度，以及一个可选的长度修饰符。格式字符串后面的值列表中的数据必须和指令中的类型一一对应。

对于这些标识和指定符（有一些在附录 B 中罗列了出来）来说，有许多可能的选项。下面是对例 12.3 的解释。首先，指令 *%6.4f* 指定要输出一个浮点数（也就是小数），总的宽度是六个字符（如果有必要就用空白填充），并且小数部分最多只能有四位。你看一下输出，尽管 *\$f* 这个浮点数给出的 π 的值已经到了小数点后八位，但是例子中指定的精度是小数点四位，结果中确实是按要求输出的。

指令 *%-5d* 指定输出一个整数，其字段宽度为 5；*-* 标识会让数字在字段中居左对齐。最后，指令 *%s* 输出一个字符串。

12.5.2 here 文档

现在，我们将简单来看一下 `here` 文档。这是指定多行输出文本的比较简单的一种方法，在其中还可以嵌入一些变量用于变量内插，用这种方式的话，输出的格式就和你在代码中看到的格式几乎是完全一样的——也就是说，不需要大量的 `print` 语句以及嵌入的换行符 `\n` 字符。我们将用例 12.3 及其输出来进行讨论。

例 12.3: here 文档示例

```
1  #!/usr/bin/perl
2  # Example 12-3   Example of here document
3
4  use strict;
5  use warnings;
6
7  my $DNA = 'AAACCCCCCGGGGGGGGTTTTTT';
8
9  for ( my $i = 0 ; $i < 2 ; ++$i ) {
10     print <<HEREDOC;
11         On iteration $i of the loop!
12         $DNA
13
14 HEREDOC
15 }
16
17 exit;
```

下面是例 12.3 的输出：

```
1  On iteration 0 of the loop!
2  AAACCCCCCGGGGGGGGTTTTTT
3
4  On iteration 1 of the loop!
5  AAACCCCCCGGGGGGGGTTTTTT
```

在例 12.3 中，一个 `here` 文档放在了 `for` 循环中，这样你就可以在输出中看到 `$i` 变量的变化了。就像在双引号字符串中对变量进行内插一样，在 `here` 文档中也可以以同样的方式进行变量内插。每次进行循环的时候，`here` 文档的内容都会在变量内插后输出出来。终止字符串可以是你指定的任意字符串，在这个例子中是 `HEREDOC`。（处理缩进这样的事情有很多选项，在此处我不会进行讨论，你可以查阅 Perl 的文档。）`here` 文档对许多任务都非常顺手，比如，你有一个很长的多行文档，每次输出它的时候都只有很小的一点改动。典型的例子就是商业信函，其中只有地址需要改变。使用 `here` 文档在最终的输出中保留了它在代码中看起来的样子，同时还允许变量内插。

12.5.3 format 和 write

最后，让我们来看一下 `format` 和 `write` 函数。`format` 被用来设计生成报表，它可以处理页码、页眉，以及居中、居左和居右对齐等各种排版选项。在格式化方面，它参考的

12.6 BioPerl

BioPerl 是一个重要的 Perl 代码集合，专门用于生物信息学，这个项目从 1998 年开始一直在发展。尽管 BioPerl 使用了 Perl 语言设计中更加高级的面向对象的风格，此处还是可以对它进行一个简单的了解，知道它是如何组织的、如何去使用它。

BioPerl 模块的主要关注点是进行序列处理，提供对各种生物学数据库（包括本地和基于网络的数据库）的访问，以及解析各种程序的输出。

在 <http://www.bioperl.org/> 上可以找到 BioPerl。它的一些特性依赖于已经安装的额外的 Perl 模块——可以从 CPAN (<http://www.cpan.org/>) 上进行获取。这种情形非常常见，随着你进行更多的 Perl 编程，你会对从 CPAN 上获取安装模块越来越熟悉的。BioPerl 指南包括在三大主流操作系统——Unix 或 Linux、Mac 和 Windows——上安装 BioPerl 和其他模块的介绍。

BioPerl 并不提供完整的程序，而是提供一个巨大——还在持续增长——的模块集，可以用来完成常见的任务，包括你在本书中已经看到过的一些任务。你需要自己编写代码，来让这些模块进行协作。通过提供这些准备妥当并且（通常）易用的模块，BioPerl 使得用 Perl 来开发生物信息学应用更加快捷、简便。对于大多数模块都有示例性的程序，你可以从查看并修改它们起步。

就像许多开源项目一样，BioPerl 也有着片段化和文档不均一的问题，这要归因于有大量的志愿者参与以及贡献者在地理上过于分散。但是，最近这个项目的工作使得 0.7 版本在 2001 年三月发布⁴，这显著改善了该项目。尤其是，现在已经有足够的模块使用指南信息，让你可以充分利用这些代码。

有些困难仍然存在。大部分代码都是在 Unix 或者 Linux 系统上开发的。并不是所有的都可以在 Macs 或者 Windows 操作系统上工作，但大部分还是可以的。在 BioPerl 网站上还有一些文档讨论在非 Unix 计算机上使用 BioPerl，但底线是你可能会发现有些事情并不能正常工作。

如果你打算尝试一下 BioPerl（我也强烈推荐你去尝试一下），你需要确保已经安装了最新版本的 Perl。你至少需要版本 5.004，如果从 Perl 网站 <http://www.perl.com> 上下载安装最新的稳定版本会更好一些。

12.6.1 示例模块

为了让你对 BioPerl 可以让哪些任务变得更加简单有一个了解，表 12.1 展示了一个最有用的模块中的代表性模块。

12.6.2 BioPerl 指南脚本

BioPerl 有一个指南脚本，帮助你尝试这个包各个部分的功能。在本节中，我将展示如何入手并运行一些示例性的计算程序。

我已经提到过，你应该学习如何从 CPAN 上下载代码，来安装 BioPerl 这样的模块。Perl 编程环境现在之所以非常有用，很大程度上是因为在 CPAN 上有各种各样的模块可以使用。这是一个设计性的决策：把精力集中放在核心 Perl 语言上，Perl 的设计者可以集中力量让这门语言尽可能的好。然后 Perl 模块的开发者可以把精力集中在他们各自的

⁴译者注：最新版本是 2014 年发布的 1.6.924。

表 12.1: BioPerl 模块

模块	描述
Bio::Seq	有特征的序列对象
Bio::SimpleAlign	最为一个序列集的多序列比对
Bio::Species	通用物种对象
Bio::DB::Ace	针对 ACeDB 服务器的数据库对象接口
Bio::DB::GDB	针对 GDB HTTP 查询的数据库对象接口
Bio::DB::GenBank	GenBank 的数据库对象接口
Bio::DB::GenPept	GenPept 的数据库对象接口
Bio::DB::NCBIHelper	查询 NCBI 数据库有用的常规程序集合
Bio::DB::SwissProt	针对 SWISS-PROT 检索的数据库对象接口
Bio::Index::Fasta	索引 FASTA 文件的接口
Bio::Index::GenBank	索引 GenBank 序列文件 (GenBank 格式的平面文件) 的接口
Bio::Location::Simple	对序列进行简单定位的实现
Bio::Location::Split	对有多个位置的序列进行定位的实现
Bio::SeqFeature::FeaturePair	处理成对的特征信息, 比如, BLAST 击中
Bio::SeqFeature::Generic	通用的 SeqFeature
Bio::SeqFeature::Similarity	基于相似性的序列特征
Bio::SeqFeature::SimilarityPair	基于两条序列的相似性的序列特征
Bio::SeqFeature::Gene::Exon	表征一个外显子的特征
Bio::SeqFeature::Gene::GeneStructure	表征一个基因任意复杂结构的特征
Bio::SeqFeature::Gene::Transcript	表征一个转录本的特征
Bio::SeqFeature::Gene::TranscriptI	表征一个有外显子、启动子、UTR 和 poly(A) 位点的转录本的特征接口
Bio::Tools::Blast	BioPerl 的 BLAST 序列分析对象
Bio::Tools::BPbl2seq	使用 BLAST 算法进行双序列比对的轻量级 BLAST 解析器
Bio::Tools::BPlite	轻量级的 BLAST 解析器
Bio::Tools::BPpsilite	用于 PSIBLAST 报告的轻量级 BLAST 解析器
Bio::Tools::CodonTable	BioPerl 的密码子表对象
Bio::Tools::Fasta	BioPerl 的 FASTA 实用对象
Bio::Tools::IUPAC	从一个含糊的序列对象生成多个唯一的序列对象
Bio::Tools::RestrictionEnzyme	用于限制性核酸内切酶对象的 BioPerl 对象
Bio::Tools::SeqPattern	用于序列模式或基序的 BioPerl 对象
Bio::Tools::SeqStats	处理单个特定序列统计信息的对象
Bio::Tools::SeqWords	处理一条序列的 n-mer 统计信息的对象
Bio::Tools::Blast::HSP	BioPerl 的 BLAST 高分片段对对象
Bio::Tools::Blast::HTML	用于 HTML 格式的 BLAST 报告的 BioPerl 实用模块
Bio::Tools::Blast::Sbjct	BioPerl 的 BLAST “击中”对象
Bio::Tools::Blast::Run::LocalBlast	本地运行 BLAST 分析的 BioPerl 模块
Bio::Tools::Blast::Run::Webblast	使用 HTTP 接口运行 BLAST 分析的 BioPerl 模块
Bio::Tools::Prediction::Exon	预测的外显子特征
Bio::Tools::Prediction::Gene	预测的基因结构特征
Bio::Variation::AChange	用于多肽的序列改变集
Bio::Variation::AARreverseMutation	氨基酸改变的点突变和密码子信息
Bio::Variation::Allele	等位基因特异的属性的序列对象
Bio::Variation::DNAMutation	DNA 水平的突变集

模块上。通过各种手段，在 CPAN 网站上好好浏览一番，看看哪些模块对你比较有用，对此有一个大概的了解。

此处，我不想对如何安装 BioPerl 进行详细的讲解：已经说过，可以在 BioPerl 网站上找到它，或者你可以访问 CPAN 网站来寻找一些信息。

所以，让我们假设你已经安装上了 BioPerl 模块，并且已经浏览了 BioPerl 网站上的指南。现在，让我们看一下如何来尝试一些 BioPerl 程序。

进入你计算机上 BioPerl 软件构建安装的目录。比如，在我的 Linux 计算机上，我把下载的 *bioperl-0.7.0.tar.gz* 文件放到了 */usr/local/src* 目录中，然后使用下面的命令将其解压缩：

```
1 | tar xvzf bioperl-0.7.0.tar.gz
```

它会创建 */usr/local/src/bioperl-0.7.0* 这个源目录。在安装上这个模块之后（请查阅文档），你就可以运行指南脚本了。

切换到源目录中，键入 `perl bptutorial.pl`。下面是运行结果（我也把给出作者和版权信息的指南的头信息显示出来了）：

```
1 | % head bptutorial.pl
2 | # $Id: ch12,v 1.44 2001/10/10 20:37:42 troutman Exp mam $
3 |
4 | =head1 BioPerl Tutorial
5 |
6 | Cared for by Peter Schattner <schattner@alum.mit.edu>
7 |
8 | Copyright Peter Schattner
9 |
10 | This tutorial includes "snippets" of code and text from various
11 | BioPerl documents including module documentation, example scripts
12 | % perl bptutorial.pl
13 |
14 | The following numeric arguments can be passed to run the corresponding demo-script.
15 | 1 => access_remote_db ,
16 | 2 => index_local_db ,
17 | 3 => fetch_local_db ,          (# NOTE: needs to be run with demo 2)
18 | 4 => sequence_manipulations ,
19 | 5 => seqstats_and_seqwords ,
20 | 6 => restriction_and_sigcleave ,
21 | 7 => other_seq_utilities ,
22 | 8 => run_standaloneblast ,
23 | 9 => blast_parser ,
24 | 10 => bplite_parsing ,
25 | 11 => hmmer_parsing ,
26 | 12 => run_clustalw_tcoffee ,
27 | 13 => run_psw_bl2seq ,
28 | 14 => simplealign_univaln ,
29 | 15 => gene_prediction_parsing ,
30 | 16 => sequence_annotation ,
```

```

31 17 => largeseqs ,
32 18 => liveseqs ,
33 19 => demo_variations ,
34 20 => demo_xml ,
35
36 In addition the argument "100" followed by the name of a single
37 bioperl object will display a list of all the public methods
38 available from that object and from what object they are inherited.
39
40 Using the parameter "0" will run all tests.
41 Using any other argument (or no argument) will run this display.
42
43 So typical command lines might be:
44 To run all demo scripts:
45 > perl -w bptutorial.pl 0
46 or to just run the local indexing demos:
47 > perl -w bptutorial.pl 2 3
48 or to list all the methods available for object Bio::Tools::SeqStats -
49 > perl -w bptutorial.pl 100 Bio::Tools::SeqStats
50
51 %

```

现在，让我们来试一下选项 9——BLAST 解析器和选项 1——access_remote_db。
所以下面先从 BLAST 解析器开始：

```

1 % perl bptutorial.pl 9
2
3 Beginning blast.pm parser example...
4
5 QUERY NAME      : gi|1401126
6 QUERY DESC     : UNKNOWN
7 LENGTH         : 504
8 FILE           : t/blast.report
9 DATE           : Thu, 16 Apr 1998 18:56:18 -0400
10 PROGRAM        : TBLASTN
11 VERSION        : 2.0.4 [Feb-24-1998]</b>
12 DB-NAME        : Non-redundant GenBank+EMBL+DDBJ+PDB sequences
13 DB-RELEASE     : Apr 16, 1998  9:38 AM
14 DB-LETTERS     : 677679054
15 DB-SEQUENCES   : 336723
16 GAPPED        : YES
17 TOTAL HITS     : 100
18 CHECKED ALL    : YES
19 FILT FUNC     : NO
20 SIGNIF HITS    : 4
21 SIGNIF CUTOFF  : 1.0e-05 (EXPECT-VALUE)
22 LOWEST EXPECT  : 0.0
23 HIGHEST EXPECT : 1e-05
24 HIGHEST EXPECT : 7.6 (OVERALL)

```

```

25 MATRIX           : BLOSUM62
26 FILTER           : NONE
27 EXPECT           : 10
28 LAMBDA, K, H     : 0.270, 0.0470, 0.230 (SHARED STATS)
29 WORD SIZE        : 13
30 S                : 42, 74 (SHARED STATS)
31 GAP CREATION      : 11
32 GAP EXTENSION     : 1
33
34 Number of hits is 4
35 Fraction identical for hit 1 is 0.25
36 Sequence identities for hsp of hit 1 are 66-68 70 73 76 79 80 87-89 114 117
37 119 131 144 146 149 150 152 156 162 165 168 170 171 176 178-182 184 187 190
38 191 205-207 211 214 217 222 226 241 244 245 249 256 266-268 270 278 284 291
39 296 304 306 309 311 316 319 324
40 %

```

这是解析 BLAST 输出的一种有趣的方式！现在，让我们再看一下访问远程数据库：

```

1 % perl bptutorial.pl 1
2 Beginning remote database access example...
3 seq1 display id is MUSIGHBA1
4 seq2 display id is AF303112
5 Display id of first sequence in stream is AF041456
6 %

```

好吧，它就像一个输出一样，只有很少的信息，但是看上去你可以推断远程数据库的访问成功了。（补充一句，如果你失败了，可能是因为你还在防火墙之后，它阻止了访问——这在大学或者大型公司中并不罕见。）

文档建议在 Perl 调试器下运行 *bptutorial.pl* 脚本，这样可以一步步观察到底发生了什么。我非常赞同它的建议，当不会在此处把它的输出也展示出来。自己去试一下吧！

既然上一个例子并不是那么有趣，就让我们再尝试一个吧。下面是序列操作的指南：

```

1 % perl bptutorial.pl 4
2
3 Beginning sequence_manipulations and SeqIO example...
4 First sequence in fasta format...
5 >Test1
6 AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAGAGTGTCTG
7 TGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTTAAATTTTATTGACTTAGG
8 TCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAAATTACAGAGTAC
9 ACAACATCCATGAAACGCATTAGCACCACC
10 Seq object display id is Test1
11 Sequence is AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAGAGTGTCTGATAG
12 CAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTTAAATTTTATTGACTTAGGTCACCTAAATACTTTAACCAATATA
13 GGCATAGCGCACAGACAGATAAAAAATTACAGAGTACACAACATCCATGAAACGCATTAGCACCACC
14 Sequence from 5 to 10 is TTTCAT
15 Acc num is unknown

```



```
16 | Moltype is dna
17 | Primary id is Test1
18 | Truncated Seq object sequence is TTTCAT
19 | Reverse complemented sequence 5 to 10 is GTGCTA
20 | Translated sequence 6 to 15 is LQRAICLCVD
21 |
22 | Beginning 3-frame and alternate codon translation example...
23 | ctgagaaaataa translated using method defaults : LRK*
24 | ctgagaaaataa translated as a coding region (CDS): MRK
25 |
26 | Translating in all six frames:
27 |   frame: 0 forward: LRK*
28 |   frame: 0 reverse-complement: LFSQ
29 |   frame: 1 forward: *ENX
30 |   frame: 1 reverse-complement: YFLX
31 |   frame: 2 forward: EKI
32 |   frame: 2 reverse-complement: IFS
33 | Translating with all codon tables using method defaults:
34 | 1 : LRK*
35 | 2 : L*K*
36 | 3 : TRK*
37 | 4 : LRK*
38 | 5 : LSK*
39 | 6 : LRKQ
40 | 9 : LSN*
41 | 10 : LRK*
42 | 11 : LRK*
43 | 12 : SRK*
44 | 13 : LGK*
45 | 14 : LSNY
46 | 15 : LRK*
47 | 16 : LRK*
48 | 21 : LSN*
49 | %
```

这更加有趣了，因为 BioPerl 的这一部分做了许多我们在本书中已经做过的事情。

我希望对于 BioPerl 的这个简短的浏览能够勾起你的好奇心。去探索一下这些模块集绝对是一个不错的想法。有一个解析 BLAST 输出的 Perl 模块，叫做 *BPLite.pm*，它可能也比较有趣：现在它还不是 BioPerl 项目的一部分。

12.7 练习题

习题 12.1

基本的字符串匹配。编写一个程序，在目标字符串中查找一个查询字符串。比如，如果查询字符串是“gone”，它会在目标字符串“goof through the way-gone-osphere”的 22 位置找到一个匹配。不要使用正则表达式或者任何 Perl 内置的字符串匹配工具；相反，在字符串中检查每一个单独的位置，比较字符，发明你自己的算法。

习题 12.2

探索 <http://www.ncbi.nlm.nih.gov/BLAST> 上的 NCBI BLAST 网页。熟悉 BLAST 各个组成程序的目的和使用，阅读指南信息理解统计结果的含义。

习题 12.3

探索 <http://www.bioperl.org> 上的 BioPerl 网页。下载代码，并在你的计算机上安装它。

习题 12.4

在 NCBI 网站上进行 BLAST 检索。先针对 DNA 数据库检索 DNA，然后针对蛋白质数据库检索同样的 DNA，最后比较它们的输出。

习题 12.5

对相关的序列进行两次 BLAST 检索。解析检索的 BLAST 结果，对于每一个检索都提取出头注释中的前 10 个击中。编写一个程序，报告两次检索的异同之处。

习题 12.6

编写一个程序，使用 BioPerl 来在 NCBI 网站上进行 BLAST 检索，然后使用 BioPerl 来解析 BLAST 的输出。

习题 12.7

使用 BioPerl 模块，混合你自己的代码，编写一个程序，在一个 DNA 序列集上运行 BLAST，对每个 BLAST 的击中列表进行排序，把排序后的 IDs 保存进数组。允许用户查看每一个列表、多个列表共有的击中，以及多个列表中每个独有的击中。对于每一个击中，可以让用户获取整个的 GenBank 记录。

习题 12.8

对子程序 *extract_HSP_information* 中的代码编写解释说明。一定要参考作为代码输入的数据的格式。

第 13 章 进阶主题

目录

13.1 程序涉及的艺术	318
13.2 网页编程	319
13.3 算法和序列比对	320
13.4 面向对象编程	321
13.5 Perl 模块	322
13.6 复杂的数据结构	323
13.7 关系数据库	324
13.8 芯片和 XML	325
13.9 图形编程	326
13.10 网络建模	327
13.11 DNA 计算机	328

本书的初衷是帮助你学习基本的 Perl 语言编程。在本章中，我会介绍一些深入学习 Perl 涉及到的主题。

13.1 程序涉及的艺术

我强调程序设计的艺术，这也暗示了程序要以何种方式展示出来。通常的过程就是先讨论问题和想法，写出伪代码，然后编写一组小的、互相协作的子程序，最后搭建出完整的程序。在某些点上，你已经看到了，完成同一个任务有不只一种方法。这是程序员心态的一个重要部分：或者利用已掌握的知识，或者进行不断的尝试。

另一个已经提过的主题也解释了使用问题解决策略的程序员所依赖的东西。它们包括知道如何充分利用可检索的新闻组档案、书籍和语言文档等资源的信息，对于调试工具有足够的实践经验，理解基本的算法和数据结构设计和分析。

随着技能的提升，你的程序会更加复杂，你会发现这些策略起的作用越来越重要。要设计、编程解决复杂的问题，或者处理大量的复杂数据，都需要更加高深的问题解决策略。所以，花一些精力学习像计算机科学和生物学家那样进行思考，是非常值得的。

13.2 网页编程

因特网是生物信息学数据最主要的来源。从 FTP 站点到使用网页的程序，学习 Perl 的生物信息学家需要有能力去访问这些网络资源。如今大概每一个实验室都必须要有自己的网页，并且许多经费也需要它。你需要学习关于 HTML 和 XML 标记语言¹的基础知识，这些标记语言被用来显示网页，要了解网络服务器和网页浏览器之间的区别，以及生活中类似的事情。

流行的 *CGI.pm* 模块使得创建交互式的网页变得相当简单，以及其他的一些可用的模块使得因特网编程任务也不是那么痛苦了。比如，你可以为你自己的网页编写代码，让访问者尝试你最新的序列分析器或者检索你特定目的的数据库。你也可以向你自己的程序中添加代码，让它们可以和其他的网站进行交互，自动访问和获取数据。在地理上分散各地的合作者们可以在一个项目中利用这样的网页编程进行亲密无间的协作。

¹译者注：还有 HTML5、XHTML、Markdown 等。

13.3 算法和序列比对

你会想花一些时间去探索一下算法中的表中结果，在附录 A 中有相关的推荐资料。一个入门的好的切入点是最基本的序列比对方法，比如 Smith-Waterman 算法。在算法的术语中，并行、随机和近似的主题都值得你至少去混个眼熟。

序列比对是算法家族中的一个子集，这就是字符串匹配算法，用来寻找相同或相似的程度，或者寻找序列间同源的证据。Smith-Waterman 算法、空位的处理、预处理和并行技术的使用以及多序列比对等等都是这个主题中的一部分。

13.4 面向对象编程

面向对象编程（object-oriented programming）是程序设计的一种风格，它为数据和子程序提供了一个明确定义的界面（在面向对象编程中叫做方法）。面向对象编程学起来并不难，它让某些本来很难的事情变得简单了（反之亦然，但你并没有必要用它来处理所有的事情！）。自从几年前这个特性被添加到 Perl 语言中以来，大量的 Perl 代码都开始用面向对象的风格进行编写。

13.5 Perl 模块

我多次提到模块，而 CPAN 这个 Perl 代码的大仓库中有大量的可以使用的模块。大部分都是免费的，但是最好检查一下版权限制，看看 Perl FAQs 中关于版权议题的讨论。最近的大多数模块，包括 CPAN 中的大量代码，都开始使用面向对象编程的风格进行编写。要想理解这种风格，你需要扩充你的 Perl 知识，但是你不需要对面向对象编程进行很深入的学习就可以在你的程序中使用大多数的模块。

13.5.1 BioPerl

生物信息学中一个重要的并且在稳步发展的 Perl 模块套件就是 BioPerl 项目，你可以在网站 <http://www.bioperl.org> 上找到它。这些模块赋予你很多的能力，都是可以直接使用的。

13.6 复杂的数据结构

Perl 可以处理复杂的数据结构，在许多编程的情况下这是非常有用的。当然这也需要你去学习，这样才能读懂你可能会遇到的大量已有的 Perl 代码。

比如，在本书中，你已经解析了很多数据。为了完成这个任务，你编写了一组子程序，每一个都非常简短，每一个都用来解析数据不同层面的结构。通过使用复杂的数据结构，你可以用反映数据的结构的形式来存储你的解析过程。这和使用面向对象的方法访问已经解析的数据结合起来，是实现数据解析的一个非常有用的方法。

复杂的数据结构依赖于指针，我在通过指针进行访问以及 *File::Find* 的讨论中简单提过它。

13.7 关系数据库

关系数据库是 Perl 程序员和生物信息学家需要了解的另外一个领域。总有一天，你会发现使用使用平面文件或者 DBM 没法管理中型或大型项目的数据，这时就要考虑关系数据库了。尽管需要花点力气才能配置好并进行编程，但是它提供了一个标准且可靠的方法来存储数据，并且针对它可以询问各种问题。在本书中，我们简单讨论了以下关系数据库，但实际上使用了一个简单的 DBM 数据库。然而，在你工作的历程中，你很可能会遇到 Oracle、MySQL、PostgreSQL、Sybase 和其他的一些数据库。Perl 模块 DBI，它本身就表示 Database Independence，使得在不（太）考虑实际使用的哪个数据库的前提下编写操作关系数据库的代码成为可能。

事实上，编写处理数据库的代码并不是很难。最困难的部分其实是要把正确的库存储到数据库中，确保有正确的 Perl 模块可以使用，以及你知道如何从你的程序中连接数据库。一旦你把这些都搞定了，使用数据库通常来说就非常容易了。

都知道，关系数据库有它们自己的知识，需要大量的知识来设计和操作好的数据库。许多程序员就专门研究这些议题，其实不少生物信息学家也专门做这个，因为对于设计更好的生物学数据库来说有很多有趣的研究问题。

13.8 芯片和 XML

芯片（用于研究基因表达的小型化的基于芯片的“实验室”）和 XML（Extensible Markup Language，可扩展标记语言）是两个结合在一起的现代发展领域。现在整个基因组都可以使用，芯片技术让你可以一次检测成千上万个基因转录本的相对水平，通过它们的帮助，我们希望理解细胞中成千上万个基因和基因产物之间的通路和相互作用。简单来说，XML 是一个新的、改良版的 HTML，它是作为存储和互换数据的标准而出现的。（本书就是通过广泛使用 XML 进行编写的。²）XMI 正在成为许多新的实验数据类型的重要的接口。

²译者注：本书是基于 \LaTeX 进行排版的。

13.9 图形编程

用好的图形展示数据，对于让你的同事能够充分理解你的结果是至关重要的。图形编程语言展示数据和结果，并且通过绚丽且易于导航的界面和软件应用进行交互。许多生物信息学的程序都处理大量的数据，一个图形用户界面（GUI, graphical user interface）可以很容易把有助于你工作的应用和浪费你时间的应用区分开来。像常见于网页上的 GUIs，不仅对于展示输出结果至关重要，对于用户数据的收集也是非常重要的。

通过点击的方法与软件应用进行交互是最基本的标准。一个好的 GUI 可以让一个应用或者程序更加易用。然而，一个复杂的 GUIs 以及图形数据展示，与更加简单的图形相比，其可移植性要差一些。你可能要去摸索一下 Tk、GD 以及其他一些 Perl 模块的图形能力。

13.10 网络建模

生物学系统，比如基因和基因产物，相互作用的网络，可以进行建模，用图算法进行研究。尽管和“图形”这个名词非常相似，但图算法是完全不同的一个东西，它基于图论的离散数学领域。举个例子，利用图和其他许多变体（比如佩特里网（Petri net））的算法，可以存储并研究生化通路和细胞内以及细胞间信号通路的属性。

13.11 DNA 计算机

对于有超前思维的科学家来说，了解研究计算方面的新动向既有趣也有启发性，比如 DNA 计算机、光计算和量子计算。DNA 计算机尤其有趣。它们使用标准的分子生物学实验室中的技术作为通用计算机的模型。它们可以执行算法、存储数据，从常见的行为来看就行一台“真”的计算机一样。在本书编写时，它们还是不切实际的，但光想想就足够激动人心的了，也许某一天真的会实现，谁知道呢？³

³译者注：该领域已经取得了不错的进展，请参看 DNA 运算（维基百科）。

附录 A 资源

目录

A.1 Perl	330
A.2 计算机科学	333
A.3 Linux	335
A.4 生物信息学	336
A.5 分子生物学	337

对于 Perl 和生物信息学编程来说，有大量的相关资源与材料。此处并不对其进行穷举，但是它包含一些在线的资源和一些印刷版的资源，我认为在你拓展 Perl 编程技能的时候，你会发现这些资源比较有趣且有用。

A.1 Perl

Perl 的文档非常详尽。它包括 FAQs（常见问题集，附带解答）列表，指南，以 Unix 风格的 man 手册页形式整理的精确定义，以及特定领域的讨论。有大量的网站，一个叫做 CPAN 的组织良好的有用的 Perl 程序仓库，具有可检索档案的新闻组，会议，和许多好的书籍。非常值得花一定的时间去寻找并结交当地的 Perl 社团。不要害怕去叨扰你的同事，随着你编程技能的提升，他们也会慢慢开始向你进行咨询。

我前面已经提到过，Perl 是免费的。它是更加庞大的开源运动的一部分，它包括 Linux、Apache 网络服务器等的开发。既然 Perl 是免费的，它就依赖于同道中人组成的一个社区团体来开发代码并撰写文档。正因为如此，你可能注意到了有不少文档都有点破碎（对某些来说简直是支离破碎）。尽管如此，这些项目的支持水平绝不亚于最好的商业软件包的支持程度。

A.1.1 网站

<http://www.perl.com>

这是 Perl 所有内容的起点。不管怎么样，去看看吧。在这里，你会发现更多关于 Perl 编程各方各面的站点。在这其中，你可能会发现 <http://www.perl.org> 尤其有用。

A.1.2 CPAN (Comprehensive Perl Archive Network) : Perl 综合典藏网

<http://www.cpan.org/>

CPAN 是一个非常重要的资源，也是寻找 Perl 模块的地方。此外，它还是其他软件、文档和网页链接的仓库。在花时间编写自己的程序之前，先到这里看看是不是已经有写好的程序了。

A.1.3 FAQs (Frequently Asked Questions) : 常见问答集

<http://www.perl.com/pub/v/faqs>

FAQs 是一个新手最常询问的问题的摘要，同时附带解答，这些解答通常都非常有用。作为一个程序员菜鸟，要想尽快上手，花点时间去读一下 FAQs 绝对是一个不错的选择，必要时可以进行跳读。

你至少应该花费足够的时间来阅读 FAQs，对哪些问题在 FAQs 中有对应的存档要有一个大概的了解。在向当地专家寻求帮助或者在新闻组中提问之前，一定要先去检查一下 FAQs。重复询问那些在 FAQs 中已经进行了详尽解答的问题，通常会让人生厌，尤其是在 Perl 的新闻组中。

你会发现 Perl 的 FAQs 分成了几个部分。当查阅 FAQs 时，看看它们上次更新的日期。这对于 Perl 来说不算是个大问题，当通常来说，你在网上会找到许多过时的信息。

初学者

在 FAQs 和文档中有许多专门针对初学者的资料。除了本书以外，还有许多其他适用于初学者的书籍，在该附录的其他地方提到了。在 <http://learn.perl.org>（当我撰写本书

时这还是一个比较新的站点，但看起来非常有前途）上也有一些关于 Perl 的在线指南和初学者文章。此外，还有一些邮件列表，你可以去订阅，包括叫做 `beginners@perl.org` 的邮件列表，通过访问 `http://lists.perl.org` 你可以订阅它。

A.1.4 在线手册

`http://www.perl.com/pub/v/documentation`

Perl 的手册是在线的，位于前面提到的 Perl 网站上。同时它也应该安装在了你的计算机上。通过键入 `perldoc perl` 你可以访问它。在 Unix/Linux 系统上，你也可以键入 `man perl` 来得到初始的 man 手册。如其所述，手册被分割成了几部分。比如，要找到 Perl 内置函数的手册，需要键入 `man perlfunc` 或者 `perldoc`。也有 HTML 版本的手册，可以把它们安装在你本地的计算机上。这是我最喜欢的获取文档的方法，它会给你链接使得导航更加容易，并且如果它被安装在了本地，甚至在没有联网的情况下都可以使用它。

A.1.5 书籍

有许多 Perl 的书籍。其中不少都非常出色，但有些也不好。下面是一个简短的 Perl 书籍列表，我发现在我的工作中它们是最有用的。

Programming Perl, Third Edition, Larry Wall、Tom Christiansen 和 Jon Orwant 著，O’ Reilly & Associates 出版¹。这是 Perl 语言发明人撰写的关于 Perl 的标准书籍。尽管它滞后于最新版本的 Perl，但它非常好得解释了一切。所以你安装的绝对权威还是在线的手册。*Programming Perl* 涵盖了大量的内部细节，所以它更适合作为参考、指南，当你需要深入细节的内容时，可以把它作为绝妙的故事来看。它展示了语言背后的一些哲学，所以可以通过理解一些计算机科学的思维方式。如果你正好有早期的版本，也是完全可以的；我个人尤其喜欢它的第一版。

Perl Cookbook, Tom Christiansen 和 Nathan Torkington 著，O’ Reilly & Associates 出版。它被宣称为 *Programming Perl* 的姊妹篇，确实如此。在这里，你会发现使用 Perl 来完成不同任务的实例。在许多情况下它都非常有用，如果你要进行许多的 Perl 编程，花费至少几个小时去研读它是非常值得的。

Mastering Algorithms with Perl, Jon Orwant、Jarkko Hietaniemi 和 John Macdonald 著，O’ Reilly & Associates 出版。我已经提到过学习算法的重要性，而该书就用 Perl 演示了许多重要的算法。它解释概念并给出代码，但它并不教授分析和测试算法的数学知识。真正严谨的学习算法的学生可以在 Corman、Leiserson 和 Rivest 著的 *Introduction to Algorithms* 中找到相应的信息。即使你是一个程序员新手，这也是一本很有价值的书，你会找到许多你可以使用的代码。

Mastering Regular Expressions, Jeffrey R. Friedl 著，O’ Reilly & Associates 出版²。一本关于重要主题的好书，很好地涵盖了 Perl。

Elements of Programming in Perl, Andrew L. Johnson 著，Manning Publications 出版。这是针对初学者的另一本书。这本书非常好，我推荐把它作为本书的补充。

¹ 《Perl 语言编程》（第四版）：<http://item.jd.com/11544992.html>。

² 《精通正则表达式》（第 3 版）：<http://item.jd.com/11070361.html>。

Learning Perl, Third Edition, Randall L. Schwartz 和 Tom Christiansen 著, O' Reilly & Associates 出版³。这是 Perl 的经典入门指南书籍。它的编写和组织都非常好。如果你从头到尾学习了 *Beginning Perl for Bioinformatics*, 那你阅读 *Learning Perl* 应该没有什么困难。

Object-Oriented Perl, Damian Conway 著, Manning Publications 出版。一本很棒的书, 其中涵盖的主题对于程序员菜鸟和老鸟都能受益匪浅。

A.1.6 会议

O' Reilly 开源大会 (*O' Reilly Open Source Convention*)。该大会现在包括一年一度的 Perl 会议。这是一个机会, 可以参加各种各样的讨论和报告, 结识形形色色的 Perl 实践者。此外还有常规的 YAPC (yet another Perl conference) 会议; 你可以在 Perl 的站点上找到它的详细信息。

A.1.7 新闻组

Perl 新闻组是程序员的一个重要资源。如果你从未看到过它们, 那是因为它们通过网络 (以及其他方式) 进行交流。它们让你可以给网络上的一大组人写一个信息, 可以针对成百上千个特定主题中的一个。如果你遇到了一个问题, 在 Perl 文档和 FAQs 中都找不到解答方法, 在新闻组中搜索针对这个问题的主题往往能得到答案。如果找不到现成的答案, 你也可以在新闻组中发表一个问题, 但这通常并不是必需的。

我想强调一下这个资源真的非常有用。弊端就是这通常倾向于“低信噪比”: 换言之, 在新闻组中常常有大量的无信息材料。但它还是值得一看的, 即使是负面的回复 (没有给出问题的已知解决方法) 也会节省你的时间和精力。

在 comp.lang.perl 层级中有许多和 Perl 相关的新闻组。搜索引擎 deja.com (最近卖给了 google.com, 但是仍然可以访问)⁴ 让你可以搜索这些新闻组的档案。对于特定新闻组的更多信息可以在 Perl 的 FAQs 中找到。比如, 许多特定的 Perl 模块都有它们自己的新闻组、邮件列表或者网站。CPAN 网站是另一个可以找到可检索新闻组档案的地方。

³ 《Perl 语言入门》(第 6 版): <http://item.jd.com/10972653.html>。

⁴ 请使用: <https://groups.google.com>。

A.2 计算机科学

尽管你是通过编程编写生物学应用，你还是会发现自己常常一不留神就进入了传统计算机科学的世界。这里是一些已经发表的资源，可以帮助你找到自己的方向。

A.2.1 算法

Mastering Algorithms with Perl, Jon Orwant、Jarkko Hietaniemi 和 John Macdonald 著，O'Reilly & Associates 出版。对于使用 Perl 编程的非计算机专业的科学家来说，这是最好的书籍。

Introduction to Algorithms, Thomas H. Cormen、Charles E. Leiserson 和 Ronald L. Rivest 著，MIT Press and McGraw-Hill 出版⁵。这绝对是关于算法的一本好书——从许多方面来说，这是最好的一本书。不管对于研究生还是大学生，它都是标准的大学教材之一（按理来说就是标准的教材）。不管是作为教材书籍还是作为参考书籍，它都完全能够胜任。它的目标读者是计算机科学专业的学生，所以里面涉及相当数量的数学知识，但即使是非数学的程序员来说，也会发现这本书非常有用。

Fundamentals of Algorithmics, Gilles Brassard 和 Paul Bratley 著，Prentice Hall 出版。对于算法技术的浅显概述。

Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Dan Gusfield 著，Cambridge University Press 出版。这本书专注于字符串相关的算法，包括序列比对等主题。它非常详尽，但即使是这样，也不是面面俱到，因为这是一个一场庞大的领域。这是专门针对字符串算法的最后的资源，有大量关于生物学序列相似性的信息。

下面的书籍共进阶学习使用。

The Design and Analysis of Computer Algorithms, Alfred V. Aho、John E. Hopcroft 和 Jeffrey D. Ullman 著，Addison-Wesley 出版。这是关于算法科学的经典书籍。

Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Frank Thomson Leighton 著，Morgan Kaufmann 出版。一个全面且严谨的教材和参考。

Randomized Algorithms, Rajeev Motwani 和 Prabhakar Raghavan 著，Cambridge University Press 出版⁶。一本清晰而严谨的书。

A.2.2 软件工程

Software Engineering, Second Edition, Ian Sommerville 著，Addison-Wesley 出版⁷。一本好的通用的书，它涵盖了重要的主题，同时避开了对立的竞争性理论的相关讨论。

⁵ 《算法导论》（原书第 3 版）：<http://item.jd.com/11144230.html>。

⁶ 《随机算法》：<http://item.jd.com/10000060.html>。

⁷ 《软件工程》（原书第 9 版）：<http://item.jd.com/10645053.html>。

A.2.3 计算机科学理论

Introduction to Automata Theory, Languages, and Computation, Second Edition, John E. Hopcroft、Rajeev Motwani 和 Jeffrey D. Ullman 著, Addison-Wesley 出版⁸。关于计算机科学理论的经典教材。

Computers and Intractability: A Guide to the Theory of Np-Completeness, Michael R. Garey 和 David S. Johnson 著, W.H. Freeman & Co 出版。关于这个主题的经典、超赞的一本书。

A.2.4 通用编程

The Unix Programmers Manual, Steven V. Earhart 等著, Harcourt、Brace 和 Jovanovich School 出版。关于 Unix (不管是那个版本的 Unix) 的这个手册, 是计算机科学中重点针对编程的速成课。交互式程序的设计, 以及管道、重定向、进程的概念, 等等都已经称为编程中巨大成功的范例之一。该手册对系统进行了概述: 第一部分描述用户程序; 第二部分和第三部分描述编程界面。可编程的 shell, 以及 grep、awk 和 sed 程序是 Perl 的主要灵感。

The C Programming Language, Brian W. Kernighan 和 Dennis M. Ritchie 著, Prentice Hall PTR 出版⁹。C 和 C++ 是生物信息学中的重要编程语言, 而这本经典的书籍教授的是 C。如果你研读了全书, 并且尝试了所有的编程练习, 那么你已经有了良好的编程训练。

Structure and Interpretation of Computer Programs, Harold Abelson、Gerald Jay Sussman 和 Juke Sussman 著, MIT Press 出版¹⁰。一本真的非常有趣的书籍, 在学习 Lisp 方言的过程中对编程进行了深入讲解。

The Unix Programming Environment, Brian W. Kernighan 和 Robert Pike 著, Prentice Hall 出版¹¹。这本书非常有趣, 并且它讨论了好的软件设计。

⁸ 《自动机理论、语言和计算导论》(原书第 3 版): <http://item.jd.com/10058560.html>。

⁹ 《C 程序设计语言》(第 2 版): <http://item.jd.com/10057446.html>。

¹⁰ 《计算机程序的构造和解释》(原书第 2 版): <http://item.jd.com/10057478.html>。

¹¹ 《UNIX 编程环境》: <http://item.jd.com/11423589.html>。

A.3 Linux

如果你有一个 Linux 操作系统，你就有了整个系统的源代码（对于一些 Unix 系统来说也是这样）。（如果它没有被安装，你可以从发行版 CD 中得到它，从网站 <http://www.linux.org> 或者制造你使用的 Linux 版本的公司的网站上得到发行版 CD。）这是一个巨大的资源。你可以看看任何一个程序，甚至是操作系统，是如何被编写的。现在你真的进入编程的世界了。

A.4 生物信息学

生物信息学是一个相对较新的学科，吸引了众多的目光，所以可用的资源也在飞速增长。下面是一些帮助你入门的书籍和其他资源。

A.4.1 书籍

Developing Bioinformatics Computer Skills, by Cynthia Gibas and Per Jambeck 著, O' Reilly & Associates 出版¹²。对于初学者来说，这是相当好的一本书。它涵盖了 Linux 工作站的构建，以及许多优秀且廉价的生物信息学程序的安装与使用。它教授的是如何去使用生物信息学程序，而非如何去编程。它是现有的最实用的一本生物信息学书籍。

Introduction to Computational Biology: Maps, Sequences and Genomes, Michael S. Waterman 著, CRC Press 出版¹³。这是一本经典的书籍，主要从统计学的角度进行讲解。

Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins, Second Edition Andreas D. Baxevanis 和 B.F. Francis Ouellette 编著, John Wiley & Sons 出版。包括由众多作者编写的多个章节，涉及比较广泛的主题。

A.4.2 政府组织

最基本的东西。下面这些网站是最重要的由政府资助的生物信息学组织。

<http://www.ncbi.nlm.nih.gov/>:

NCBI (National Center for Biotechnology Information) : 国家生物技术信息中心, 美国政府中心。

<http://www.embl.org/>:

EMBL (European Molecular Biology Laboratory) : 欧洲分子生物学实验室, 欧洲联合实验室。

<http://www.ebi.ac.uk/>:

EMBL 中的 EBI (European Bioinformatics Institute) : 欧洲生物信息研究所。

A.4.3 会议

生物信息学一直是各种生物学会会议的一部分，比如，关于测序的冷泉港会议 (Cold Spring Harbor conferences)。现在有许多涉及该方面的会议，通常都被冠以“基因组学”。下面是几个有趣的会议：

- ISMB (*Intelligent Systems for Molecular Biology*) : 国际分子生物学智能系统会议，现在是它的第九个年头¹⁴
- 生物信息学开放源代码会议 (*Bioinformatics Open Source Conference*) , <http://www.bioinformatics.org/>
- RECOMB (*Conference on Computational Molecular Biology*) : 计算分子生物学会会议

¹² 《生物信息学中的计算机技术》。

¹³ 《计算生物学导论：图谱、序列和基因组》：<http://item.jd.com/10005674.html>。

¹⁴ 译者注：指的是 2001 年。一年一届，2015 年举办的是第 23 届。

A.5 分子生物学

Recombinant DNA, James Watson 等著, W.H. Freeman & Co 出版。相对于这么一个快速发展的领域,该书显得有点陈旧了,但它还是一本引领程序员和计算机科学家进入生物信息学领域的佳作。许多标准的技术都用精彩的插图进行了清晰简洁的解释。去找一下它的第二版,看看能不能找到。

Molecular Biology of the Gene, Fourth Edition, James Watson 等著, Addison-Wesley 出版¹⁵。分子生物学的经典书籍。它非常详尽,从知识面的覆盖来说,它确实有些过时了,但仍不失为经典之作。对于基础知识可以好好参考该书。

Molecular Cell Biology, Fourth Edition, Harvey Lodish 等著, W.H. Freeman & Co 出版。关于细胞生物学的优秀且宽泛的介绍性概述。

《Lewin 基因 X (中文版)》¹⁶: <http://item.jd.com/11159665.html>。该书对分子生物学和分子遗传学进行了精彩的论述,内容涵盖了基因的结构、序列、组织和表达,是分子生物学和分子遗传学最经典的名著之一。

¹⁵ 《基因的分子生物学》(第七版): <http://item.jd.com/11672603.html>。

¹⁶ 译者补充。

附录 B Perl 概要

目录

B.1 命令解释	340
B.2 注释	341
B.3 标量值和标量变量	342
B.4 赋值	344
B.5 语句和块	345
B.6 数组	346
B.7 散列	347
B.8 操作符	349
B.9 操作符优先级	350
B.10 基本操作符	351
B.11 条件和逻辑操作符	353
B.12 绑定操作符	357
B.13 循环	358
B.14 输入/输出	361
B.15 正则表达式	365
B.16 标量和列表上下文	371
B.17 子程序和模块	373
B.18 内置函数	376

本附录对 Perl 编程语言的相关内容进行了总结，这些知识对于你阅读本书大有裨益。它并不是对 Perl 语言进行的全面的总结。牢记一点，你不需要知道 Perl 的所有知识，就可以去使用它。本附录的原始材料是来源于 *Programming Perl, Third Edition* (O' Reilly & Associates) 的。

B.1 命令解释

本书中的 Perl 程序都起始于这样一行：

```
1 |#!/usr/bin/perl -w
```

在 Unix（或 Linux）系统中，文件的第一行可以包含程序的名称和一些可选的标志。该行必须以 `#!` 起始，后面紧跟程序（这我们的例子中，就是 Perl 解释器）的全路径¹名，之后是可选的由一个或多个标志做成的标志组。

如果 Perl 程序文件名为 *myprogram*，并且有可执行的权限，你直接键入 *myprogram*（或者可能是 *./myprogram*，或者是程序的绝对或相对路径名）就可以运行程序了。

Unix 操作系统会启动由命令解释行指定的程序，并把文件中第一行之后的所有内容作为程序的输入。所以，在这个例子中，它会启动 Perl 解释器，并把文件中的程序交由 Perl 解释器来运行。

上述其实是在命令行中输入：

```
1 |/usr/bin/perl -w myprogram
```

的一种简写。

¹译者注：即绝对路径。

B.2 注释

从 # 起始，一直到该行的结尾都是注释的内容。注释会被 Perl 解释器忽略掉，它仅供程序员阅读。在注释中可以包含任何文本内容。

B.3 标量值和标量变量

一个标量值是数据的单一项目，比如一个字符串或者一个数字。

B.3.1 字符串

字符串是标量值，书写形式就是包裹在单引号内的文本，就像这样：

```
1 | 'This is a string in single quotes.'
```

或者使用双引号，像这样：

```
1 | "This is a string in double quotes."
```

单引号包裹的字符串会被原样输出。使用双引号时，你可以在字符串中包含变量，在输出时变量的值会被插入或者“以内插值替换”。你还可以使用 `\n` 这样的命令来表示一个新行（参看表 B.3）：

```
1 | $aside = '(or so they say)';  
2 | $declaration = "Misery\n $aside \nloves company.";   
3 | print $declaration;
```

这个代码片段会输出：

```
1 | Misery  
2 | (or so they say)  
3 | loves company.
```

B.3.2 数字

数字可以是如下的标量值：

- 整数：

3
-4
0

- 浮点数（小数）：

4.5326

- 科学计数法（指数）（ 3.13×10^{23} 或 313000000000000000000000）：

3.13E23

- 十六进制（以 16 为基数）：

0x12bc3

- 八进制（以 8 为基数）：

05777

- 二进制（以 2 为基数）：

0b10101011

$3 + i$ 这样的复数（或虚数），以及 $1/3$ 这样的分数（或比率，或有理数），使用起来会有点麻烦。Perl 可以处理分数，但在内部会把它们转换成浮点数，这会使某些操作符出现错误（计算机语言中存在该问题的不仅仅是 Perl）：

```
1 | if ( 10/3 == ( (1/3) * 10 ) ) {  
2 |     print "Success!";  
3 | }else {  
4 |     print "Failure!";  
5 | }
```

这会输出：

```
1 | Failure!
```

要准确地处理分数、复数或其他许多数学结构的有理运算，可以使用相应的数学模块，此处不进行赘述。

B.3.3 标量变量

标量值可以存储在标量变量中。一个标量变量用变量名前面的 $\$$ 来进行表明。变量名以字母或下划线起始，可以包含任意数量的字母、下划线和数字。但是，数字不可以是变量名的第一个字符。下面是标量变量合法名称的一些例子：

```
1 | $Var  
2 | $var_1
```

下面是变量变量的一些不合法的名称：

```
1 | $lvar  
2 | $var!iable
```

变量名是大小写敏感的： $\$dna$ 和 $\$DNA$ 是两个不同的变量。

这些用来生成合法标量变量名的规则（除以 $\$$ 起始外），同样适用于数组和散列的变量名，以及子程序的命名。

一个标量变量可能存储前面提到的任意一种变量值，比如字符串或者不同类型的数字。

B.4 赋值

使用赋值语句，把标量值赋值给标量变量。例如：

```
1 | $thousand = 1000;
```

把 1,000 这个标量值赋值给了标量变量 `$thousand`。

赋值语句和初等数学里面的等号看起来非常相似，但意义完全不同。赋值语句是一个操作指令，而不是一个论断。它表示的不是“`$thousand` 等于 1,000”，而是“把标量值 1,000 存储到标量变量 `$thousand` 中去”。不管怎样，在这个语句执行后，标量变量 `$thousand` 的值确实等于 1,000 了。

你可以把多个值赋值给多个标量变量，方法就是把变量和值包裹在括号中并用逗号分隔开来，这实际上是构造了列表：

```
1 | ($one, $two, $three) = ( 1, 2, 3 );
```

除了 `=` 以外，还有许多复制操作符，它们都是长表达式的简写。比如，`$a += $b` 等同于 `$a = $a + $b`。表 B.1 给出了完整的列表（它包含了本书中没有提到的一些操作符）。

表 B.1: 复制操作符简写

操作符实例	等同于
<code>\$a += \$b</code>	<code>\$a = \$a + \$b</code> (加法)
<code>\$a -= \$b</code>	<code>\$a = \$a - \$b</code> (减法)
<code>\$a *= \$b</code>	<code>\$a = \$a * \$b</code> (乘法)
<code>\$a /= \$b</code>	<code>\$a = \$a / \$b</code> (除法)
<code>\$a **= \$b</code>	<code>\$a = \$a ** \$b</code> (求幂)
<code>\$a %= \$b</code>	<code>\$a = \$a % \$b</code> (取模, <code>\$a/\$b</code> 的余数)
<code>\$a x= \$b</code>	<code>\$a = \$a x \$b</code> (把字符串 <code>\$a</code> 重复 <code>\$b</code> 次)
<code>\$a &= \$b</code>	<code>\$a = \$a & \$b</code> (位与)
<code>\$a = \$b</code>	<code>\$a = \$a \$b</code> (位或)
<code>\$a ^= \$b</code>	<code>\$a = \$a ^ \$b</code> (位异或)
<code>\$a >>= \$b</code>	<code>\$a = \$a >> \$b</code> (<code>\$a</code> 右移 <code>\$b</code> 位)
<code>\$a <<= \$b</code>	<code>\$a = \$a << \$b</code> (<code>\$a</code> 左移 <code>\$b</code> 位)
<code>\$a &&= \$b</code>	<code>\$a = \$a && \$b</code> (逻辑与)
<code>\$a = \$b</code>	<code>\$a = \$a \$b</code> (逻辑或)
<code>\$a .= \$b</code>	<code>\$a = \$a . \$b</code> (把字符串 <code>\$b</code> 附加到 <code>\$a</code> 后)

B.5 语句和块

程序是由语句构成的，而语句通常组合成块。

语句以分号 (;) 结束，而对于块中的最后一个语句来说分号是可有可无的。

一个块通常就是用大括号包裹起来的一个或多个语句。下面是一个例子：

```
1 {  
2   $thousand = 1000;  
3   print $thousand;  
4 }
```

块可以独立存在，但通常都会与循环或 `if` 语句结构关联在一起。

B.6 数组

数组是有序的零个或多个标量值的集合，通过位置进行索引。一个数组变量起始于一个 `@` 符号，后面是一个合法的变量名。比如，下面是两个可能的数组变量名：

```
1 | @array1
2 | @dna_fragments
```

你可以把标量值赋值给数组，方法就是把这些标量值放在列表中，用逗号分隔开、并用成对的括号包裹起来。比如，你可以把空列表赋值给数组：

```
1 | @array = ( );
```

或者，把一个或多个标量值赋值给数组：

```
1 | @dna_fragments = ('ACGT', $fragment2, 'GGCGGA');
```

注意，在一个列表中，完全可以指定像 `$fragment2` 这样的标量变量。存放到数组中的，是它现在的值，而非变量名。

数组中单个的标量值（元素）通过它们在数组中的位置进行索引。索引数组从 0 开始。在数组名前面使用 `$`，在其后面紧跟用中括号 `[]` 包裹起来的元素索引数字，这样你就可以指定数组中特定的一个元素了，就想这样：

```
1 | $dna_fragments[2]
```

考虑到先前对数组进行的赋值，它现在的值就等于 `'GGCGGA'`。注意数组有三个标量值，分别用 0、1 和 2 进行索引。第三个、也就是最后一个元素的索引值是 2，比总的元素数目 3 小 1，这是因为第一个元素的索引值是 0。

使用复制操作符 `=`，你可以把数组复制一份，就像下面这个例子中一样，它为现有的数组 `@input` 复制了一份拷贝 `@output`：

```
1 | @output = @input;
```

如果你在标量上下文中对数组进行求值，得到的值是数组中元素的数目。所以如果数组 `@input` 有五个元素，下面的这个例子就会把 5 这个值赋值给 `$count`：

```
1 | $count = @input;
```

图 B.1 展示了含有三个元素的数组 `@myarray`，它演示了数组的有序属性。对于其中的每一个元素，都可以通过它在数组中的位置找到。

Arrays:**@myarray=('DNA', 'RNA', 'Protein');****Positions: 0 1 2****Scalar values:**

DNA	RNA	Protein
------------	------------	----------------

图 B.1: 图解数组

B.7 散列

散列（也叫做关联数组）是零个或多个成对标量值的集合，这些成对的标量值叫做键和值。其中的值通过键进行索引。一个散列标量起始于一个 % 符号，后面是一个合法的变量名。比如，可能的散列变量名：

```
1 | %hash1
2 | %genes_by_name
```

你可以使用一个简单的赋值语句把值赋值给键。比如，假设你有一个叫做 %baseball_stadiums 散列和一个叫做 Phillies 的键，你想把值 Veterans Stadium 赋值给这个键。下面的这个语句就可以完成赋值：

```
1 | $baseball_stadiums{'Phillies'} = 'Veterans Stadium';
```

注意，单独的一个散列值用散列名前面的 \$ 而非 % 进行指代；这和数组中使用的方法非常类似，当指代单独的一个数组值时使用 \$ 而非 @。

你可以把多个键、值赋值给散列，方法就是把它们的标量值放在列表中，用逗号分隔开、并用成对的括号把它们包裹起来。每一个连续的标量对都会称为散列中的一个键和一个值。比如，你可以把空列表赋值给散列：

```
1 | %hash = ( );
```

你也可以把一个或多个标量键-值对赋值给散列：

```
1 | %genes_by_name = ('gene1', 'AACCCGGTTGGTT', 'gene2', 'CCTTTCGGAAGGTC');
```

还有另一种方法也可以完成同样的事情，但它使得键-值对的关系更加一目了然。下面做的事情和前面的例子完全一样：

```
1 | %genes_by_name = (
2 |   'gene1' => 'AACCCGGTTGGTT',
3 |   'gene2' => 'CCTTTCGGAAGGTC'
4 | );
```

要提取和某个特定键相关联的值，只需要在散列名前使用 \$，并在其后紧跟包裹键的标量值的成对大括号 { } 即可：

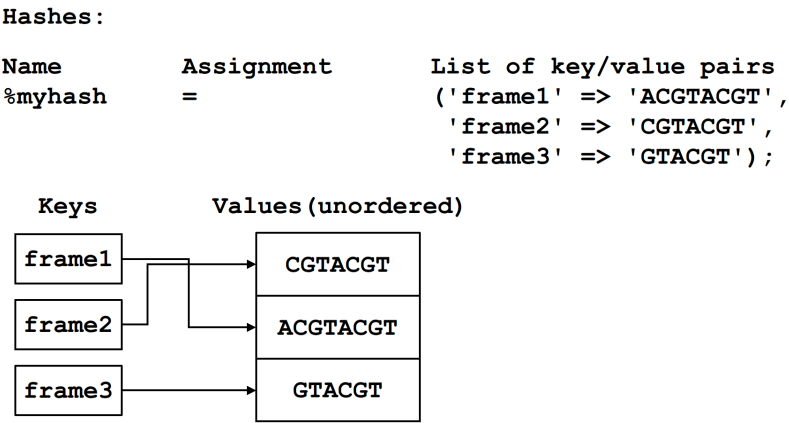


图 B.2: 图解散列

```
1 | $genes_by_name{'gene1'}
```

考虑到先前在散列 %genes_by_name 中对 'gene1' 进行的赋值，它会返回 'AACCCGGTTGGTT' 这个值。图 B.2展示了一个含有三个键的散列。

B.8 操作符

操作符是代表对值进行加减等基本操作的函数。它们的使用非常频繁，是 Perl 编程语言的核心部分。它们其实就是需要参数的函数。举个例子，+ 是把两个数进行相加的操作符，就像这样：

```
1 | 3 + 4;
```

操作符通常有一个、两个或三个运算对象。在刚才的例子中，有 3 和 4 这么两个运算对象。

操作符可以出现在它们的运算对象的前面、中间或者后面。比如，加法运算符 + 就出现在它的运算对象的中间。

B.9 操作符优先级

操作符优先级决定了运算的顺序。比如，在 Perl 中，下面这个表达式：

```
1 | 3 + 3 * 4
```

并不是从左到右进行求值的，也就是说，并不是先计算 3 加 3 等于 6、然后把 6 乘以 4 得到 24 这个值的。优先级规则决定先乘法后加法，最终得到 15 这个结果。在 *perlop* 手册页和大多数 Perl 书籍中都可以找到优先级规则。但是，我建议你使用括号来使你的代码更加易读，同时避免 bugs。它们使得表达式清晰明确。第一个例子：

```
1 | (3 + 3) * 4
```

求值后得到 24，而第二个例子：

```
1 | 3 + (3 * 4)
```

求之后得到 15。

B.10 基本操作符

关于操作符运算的更多信息，请查阅和 Perl 绑定在一起的 `perlop` 文档。

B.10.1 算术操作符

Perl 有五个基本的算术操作符：

+	加法
-	减法
*	乘法
/	除法
**	求幂

这些操作符对整数和浮点数值都适用（如果你不小心地话，也可以对字符串使用）。

Perl 还有一个 `%` 操作符，它会计算两个整数的余数：

```
1 | % modulus
```

比如，`17 % 3` 的值为 2，因为当你把 17 除以 3 后得到的余数是 2。

Perl 还有自增和自减操作符：

```
1 | ++  0 0
2 | --  0 0
```

和前面的六个操作符不同，它们会改变变量的值。`$x++` 会给 `$x` 加一，把值从 4 变到 5（或者从 `a` 变到 `b`）。

B.10.2 位操作符

所有的标量，不管是数字还是字符串，“在底层”都是用一串单个的位（比特）来表示的。偶尔你需要操作这些位，而 Perl 提供了五个操作符可供使用：

&	位与
	位或
^	位异或
»	右移
«	左移

B.10.3 字符串操作符

使用点操作符，可以把字符串连接起来——头尾相连：

```
1 | 'This is a ' . 'joined string'
```

这会得到' This is a joined string' 这个值。

使用 x 操作符也可以把一个字符串进行重复：

```
1 | print "Hear ye! " x 3;
```

这会输出：

```
1 | Hear ye! Hear ye! Hear ye!
```

B.10.4 文件测试操作符

文件测试操作符是一元操作符，它会检测文件的特定属性，比如对于 `-e $file`，当文件 `$file` 存在时它会返回 `0`。表 B.2列出了一些可用的文件测试操作符。

表 B.2: 文件测试操作符

操作符	含义
-r	文件是可读的
-w	文件是可写的
-x	文件是可执行的
-o	文件为“你”所有
-e	文件存在
-z	文件字节单位的大小为零
-s	文件的大小非零（返回以字节为单位的大小）
-f	文件是一个普通文件
-d	文件是一个目录（也就是文件夹）
-l	文件是一个符号链接
-t	文件句柄打开的是终端
-T	文件是一个文本文件
-B	文件是一个二进制文件
-M	文件最后一次被修改后至今（程序启动时）的天数
-A	文件最后一次被访问后至今（程序启动时）的天数
-C	文件最后一次节点编号被变更后至今（程序启动时）的天数

B.11 条件和逻辑操作符

本节涵盖条件语句和逻辑操作符的相关内容。

B.11.1 真和假

在条件测试中，一个表达式求值的结果为 `true`（真）或 `false`（假）。基于求值结果的不同，一个语句或者代码块有可能执行，也有可能不执行。

在条件中，一个标量的值可能为 `true`（真）或 `false`（假）。如果一个字符串是空字符串（用“”或“”表示），那么它的值就为 `false`；如果字符串不是空字符串，那么它的值就为 `true`。

与之类似，如果一个数组或者散列为空，它的值就是 `false`，不为空值就是 `true`。

如果一个数字是 0，它的值就为 `false`；数字不是 0，值就为 `true`。

在 Perl 中，大多数东西在求值后都会返回一些值（比如：算数运算表达式返回的数字，或者子程序返回的数组），因此你可以在 Perl 的条件测试中使用它们。有时，你可能会得到一个未定义的值，例如当你把一个数字和一个还没有赋值的变量相加时，此时一切就会可能和预期相差甚远。举个例子：

```
1 | use strict;
2 | use warnings;
3 | my $a;
4 | my $b;
5 | $b = $a + 2;
```

会得到警告：

```
1 | Use of uninitialized value in addition (+) at - line 5.
```

使用 Perl 的函数 `defined` 来检测值是定义的还是未定义的。

B.11.2 逻辑操作符

一共有四种逻辑操作符：

```
1 | not
2 | and
3 | or
4 | xor
```

`not`（非）会把 `true`（真）值变 `false`（假），把 `false`（假）值变 `true`（真）。用代码来更好的阐释以下：

```
1 | if(not $done) {...}
```

只有在 `$done` 的值为 `false`（假）时，代码才会执行。

`and`（与）是一个二元操作符，只有当两边操作数的值都为 `true`（真）时，它才会返回 `true`（真）。如果有一个操作数为 `false`（假），运算符都会返回 `false`（假）：

```
1 | 1 and 1 returns true
2 | 'a' and '' returns false
3 | '' and 0 returns false
```

`or`（或）也是一个二元操作符，只要两个操作数中至少有一个为 `true`（真），它就会返回 `true`（真）。如果两个操作数都为 `false`（假），它会返回 `false`（假）：

```
1 | 1 or 1 returns true
2 | 'a' or '' returns true
3 | '' or 0 returns false
```

`xor`（异或）只有在两个操作数中一个为 `true`（真）一个为 `false`（假）的情况下才会返回 `true`（真）。如果两个都为 `true`（真）或者都为 `false`，`xor` 会返回 `false`（假）：

```
1 | 1 xor 0 returns true
2 | 0 xor 1 returns true
3 | 1 xor 1 returns false
4 | 0 xor 0 returns false
```

这几个操作符大多数都有变体：

```
1 | ! for not
2 | && for and
3 | || for or
```

除了优先级不同以外，其他没有什么区别。一些老版本的 Perl 可能只有：

```
1 | !
2 | ||
3 | &&
```

而没有 `not` 或者 `and`。

B.11.3 使用逻辑操作符控制流程

要想依据上一个动作的结果来决定下一步的动作，最简便也是最流行的做法就是用逻辑操作符把语句链接组合在一起。比如，在 Perl 程序中，用来打开文件的下面这一个语句就非常常见：

```
1 | open(FH, $filename) or die "Cannot open file $filename: $!";
```

这个语句中 `or` 的使用展示了关于二元逻辑操作符的另一个重要的事情：它们是从左向右对参数进行求值的。在这个例子中，如果文件打开成功，`or` 操作符永远都不会去检

查第二个操作数的值（die 会输出字符串中的信息并退出程序，如果使用了 \$! 还会有一些额外的信息）。之所以会这样，是因为如果一个操作数为 true（真），or 就为 true，所以它根本不需要去检查第二个操作数的值。然而，如果文件打开失败，or 就需要去检查一下第二个操作数的值是 true 还是 false，所以它就会继续执行 die 语句。

与之类似，你也可以使用 and 语句，在只有第一个操作成功的情况下才去测试第二个操作。

xor 不会用于控制流程，因为每次都要对它的两个参数进行求值。

对于这种逻辑操作符控制的流程链我使用的并不多，我主要使用 if 语句。这是因为，我尝尝发现我需要在测试后面添加一些语句，这种情况下，如果使用 if 语句包裹代码块修改起来就比较方便，如果使用的逻辑操作符修改起来就麻烦多了。

B.11.4 if 语句

在 if 语句以及它们的变体和循环中，条件测试非常常见。下面是一个 if 语句的例子：

```
1 | if (open (FH, $filename) ) {  
2 |     print "Hurray, I opened the file."  
3 | }
```

if 语句后面紧跟着一个包裹在小括号中的条件表达式，条件表达式的后面是包裹在大括号 { } 中的代码块。当条件表达式求值为 true（真）时，代码块中的语句就会执行。

if 语句后面也可能会跟着一个 else，当条件表达式求值为 false 时，其中的代码块就会执行：

```
1 | if ( open(FH, $filename) ) {  
2 |     print "Hurray, I opened the file."  
3 | } else {  
4 |     print "Rats. The file did not open."  
5 | }
```

if 表达式中也可能会包括可选的不定个数的 elsif 语句，当前面的所有条件语句都不为 true（真）时，它会检查额外的条件语句：

```
1 | if ( open(FH, $file1) ) {  
2 |     print "Hurray, I opened file 1."  
3 | } elsif ( open(FH, $file2) ) {  
4 |     print "Hurray, I opened file 2."  
5 | } elsif ( open(FH, $file3) ) {  
6 |     print "Hurray, I opened file 3."  
7 | } else {  
8 |     print "None of the dadblasted files would open."  
9 | }
```

在上面这个例子中，如果 file1 成功打开，if 语句就不会再去尝试打开其他的文件了。

也有一个 `unless` 语句，除了条件取反外，和 `if` 语句完全一样。所以，下面这两个语句是完全等价的：

```
1 unless ( open(FH, $filename) ) {  
2   print "Rats. The file did not open."  
3 }  
4  
5 if ( not open(FH, $filename) ) {  
6   print "Rats. The file did not open."  
7 }
```

B.12 绑定操作符

绑定操作符用于字符串的模式匹配、替换和转换。它们和指定模式的正则表达式配合使用。下面是一个例子：

```
1 | 'ACGTACGTACGTACGT' =~ /CTA/
```

模式就是用反斜杠 `/` 包裹起来的字符串 `CTA`。字符串绑定操作符是 `=~`，它告诉程序对哪个字符串进行搜索，如果字符串中有这个模式，它就会返回 `true`。

另一个字符串绑定操作符是 `!~`，如果字符串中没有这个模式，它就会返回 `true`。

```
1 | 'ACGTACGTACGTACGT' !~ /CTA/
```

这等同于：

```
1 | not 'ACGTACGTACGTACGT' =~ /CTA/
```

使用字符串绑定操作符，你可以把一个模式替换成另一个模式。在接下来的这个例子中，`s/thine/nine/` 是替换命令，它会把第一个出现的 `thine` 替换成字符串 `nine`：

```
1 | $poor_richard = 'A stitch in time saves thine.';
2 | $poor_richard =~ s/thine/nine/;
3 | print $poor_richard;
```

这会得到如下输出：

```
1 | A stitch in time saves nine.
```

最后，转换（或者翻译）操作符 `tr` 会替换字符串中的字符。它有很多的用处，我已经提到过了两个。第一，可以用它来把碱基变成它们的互补碱基 `A→T`、`C→G`、`G→C`、`T→A`：

```
1 | $DNA = 'ACGTTTAA';
2 | $DNA =~ tr/ACGT/TGCA/;
```

这会得到如下的值：

```
1 | TGCAAATT
```

第二，`tr` 操作符计算一个字符串中特定字符的数目，在下面这个例子中，它计算 DNA 序列字符串中 `G` 的个数：

```
1 | $DNA = 'ACGTTTAA';
2 | $count = ($DNA =~ tr/A//);
3 | print $count;
```

它会输出值 3。这展示了，一个模式匹配可以返回在一个字符串中进行转换的次数，随后这个计数被赋值给了变量 `$count`。

B.13 循环

循环会重复执行代码块中的语句，直到条件测试的值发生改变。在 Perl 中有多种类型的循环：

```
1 while(CONDITION) {BLOCK}
2 until(CONDITION) {BLOCK}
3 for(INITIALIZATION ; CONDITION ; RE-INITIALIZATION ) {BLOCK}
4 foreach VAR (LIST) {BLOCK}
5 for VAR (LIST) {BLOCK}
6 do {BLOCK} while (CONDITION)
7 do {BLOCK} until (CONDITION)
```

while 循环首先测试条件是否为 true：如果为 `0`，它就会执行代码块，然后返回到条件重复上面的过程；如果为 `1`，它什么也不会做，同时循环结束。比如：

```
1 $i = 3;
2 while ( $i ) {
3     print "$i\n";
4     $i--;
5 }
```

这会得到下面的输出：

```
1 3
2 2
3 1
```

循环的过程是这样子的。标量变量 `$i` 首先被初始化为 3（这并不是循环的部分）。接着进入循环，`$i` 被测试来看一下它是否有个 true（非零）值。如果是，数字 3 就会被输出出来，并且减量操作符会被应用到 `$i` 上，这会使它的值减小为 2。现在代码块就结束了，循环从条件测试再次开始。因为是 true 值 2，所以测试成功，值被打印出来，同时减一。循环又一次从 `$i` 的测试开始，它现在是 true 值 1，1 会被打印出来，同时减小为 0。循环再一次开始，0 被测试来看看它是不是为 true，因为它不为 `0`，所以现在循环就结束了。

循环通常都遵循同样的模式：先初始化一个变量，然后调用一个循环，它会测试变量的值，然后执行一个代码块，代码块中会有语句改变变量的值。

for 循环让这些简便了许多，它把变量的初始化和变量值的改变都放在了循环语句中。下面这个例子和刚才的例子是完全等价的，输出也完全一样：

```
1 for ( $i = 3 ; $i ; $i-- ) {
2     print "$i\n";
3 }
```

要想对一个数组中的元素进行循环处理，使用 foreach 循环是一个比较便捷的方式。下面是一个例子：

```
1 | @array = ('one', 'two', 'three');
2 |
3 | foreach $element (@array) {
4 |     print $element"\n";
5 | }
```

它会输出：

```
1 | one
2 | two
3 | three
```

`foreach` 循环指定了一个标量变量 `$element` 来依次表示数组中的每一个元素。（你可以使用任意的变量名，甚至是 `none`，在这种情况下，会自动使用特殊变量 `$_`。）小括号中的数组会被循环处理，执行后面的代码块。你也可以使用 `for` 来代替这个循环中的 `foreach`，效果是完全一样的。

进行第一个循环的时候，数组第一个元素的值会赋值给 `foreach` 中的变量 `$element`。每当循环成功执行依次，数组下一个元素的值就会赋值给 `foreach` 中的变量 `$element`。当循环到达数组的末尾时，循环就会结束。

然而，有一点需要强调一下。如果在代码块中，你改变了循环变量 `$element` 的值，数组也会随之改变，即使你离开了 `foreach` 循环，这种改变也会一直有效。举个例子：

```
1 | @array = ('one', 'two', 'three');
2 |
3 | foreach $element (@array) {
4 |     $element = 'four';
5 | }
6 |
7 | foreach $element (@array) {
8 |     print $element,"\n";
9 | }
```

会输出：

```
1 | four
2 | four
3 | four
```

在 `do-until` 循环中，代码块在条件测试之前就会执行，并且测试会一直成功，直到条件为 `true` 为止：

```
1 | $i = 3;
2 | do {
3 |     print $i,"\n";
4 |     $i--;
5 | } until ( $i );
```

会输出：

```
1 | 3
```

在 do-while 循环中，代码块在条件测试之前就会执行，并且当条件为 true 时，测试会一直成功。

```
1 | $i = 3;  
2 | do {  
3 |   print $i, "\n";  
4 |   $i--;  
5 | } while ( $i );
```

会输出：

```
1 | 3  
2 | 2  
3 | 1
```

B.14 输入/输出

本节涵盖向程序输入信息、并从程序中获取输出的相关内容。

B.14.1 从文件获取输入

Perl 有许多便捷的方法可以把信息输入到程序中。在本书中，我强调的是打开文件并读取其中的信息，因为它使用非常频繁，并且在所有不同操作系统中它的操作都是一样的。你已经见过 `open` 和 `close` 系统调用，以及当你打开文件的时候如果把它和文件句柄相关联，之后使用文件句柄来读取数据。像下面这个例子：

```
1 | open(FILEHANDLE, "informationfile");
2 | @data_from_informationfile = <FILEHANDLE>;
3 | close(FILEHANDLE);
```

上述代码会打开文件 *informationfile*，并把它和文件句柄 `FILEHANDLE` 关联在一起。之后在尖括号 `< >` 中使用文件句柄把文件的内容读取进来，并且把这些内容存储到数组 `@data_from_informationfile` 中。最后，通过再一次调用已经打开的文件句柄把文件关闭掉。

B.14.2 从 STDIN 获取输入

Perl 允许你读入通过标准输入 (STDIN) 自动发送到程序的任何输入。STDIN 是一个默认一直打开的文件句柄。你的程序可能会期望通过这种方式获取一些输入。比如，在 Mac 中，你可以通过拖放一个文件的图标到你的 Perl 程序的 applet 上，让这个文件的内容出现在 STDIN 中。在 Unix 系统中，你可以使用 shell 命令的管道把其他程序的输出作为你的程序的标准输入，就想这样：

```
1 | someprog | my_perl_program
```

通过下面这种方式，你也可以使用管道把文件的内容作为你的程序的输入：

```
1 | cat file | my_perl_program
```

或者这样：

```
1 | my_perl_program < file.
```

之后，你的程序就可以读入 STDIN 的（来自程序或者文件的）数据了，就像这些数据来在于你已经打开的一个文件一样：

```
1 | @data_from_stdin = <STDIN>;
```

B.14.3 从命令行指定的文件中获取输入

你可以在命令行中指定你的输入文件。`<>` 是 `<ARGV>` 的简写。ARGV 文件句柄把数组 `@ARGV` 作为一个文件名的列表，并且一次一行得返回所有这些文件的内容。Perl 把所有的命令行参数都放在数组 `@ARGV` 中。其中有些可能是特殊的标志，当同时也有制定的数据文件时，它们应该从 `@ARGV` 中读取并删除掉。当看到 `< >` 命令时，Perl 假定 `@ARGV` 中的所有内容都代表一个输入文件名。使用尖括号 `< >`，而不需要文件句柄，就可以让程序获取文件的内容，就像这样：

```
1 | @data_from_files = <>;
```

比如，在 Microsoft、Unix 或者 MacOS X 上，你在命令行中指定输入文件，就像这样：

```
1 | % my_program file1 file2 file3
```

B.14.4 输出命令

`print` 语句是从 Perl 程序中输出数据最常用的方法。`print` 语句把用逗号分隔开的一堆标量作为它的参数。一个数组也可以作为参数，在这种情况下，数组中的元素会一个接一个得全部输出出来：

```
1 | @array = ('DNA', 'RNA', 'Protein');  
2 | print @array;
```

这会输出：

```
1 | DNARNAProtein
```

如果你想在数组的元素之间加上空格，使用 `print` 语句的时候就把它放在双引号中，就像这样：

```
1 | @array = ('DNA', 'RNA', 'Protein');  
2 | print "@array";
```

这会输出：

```
1 | DNA RNA Protein
```

在 `print` 语句和参数之间，可以指定一个文件句柄作为可选的间接对象，就像这样：

```
1 | print FH "@array";
```

`printf` 函数给予用户更多的控制，来格式化输出数字。比如，你可以指定字段宽度，精度或说小数点后的数字位数，以及字段中的值是右对齐或者左对齐。在第 12 章中我已经展示了大多数常见的选项，推荐你去随 Perl 附带的 Perl 文档中常阅更加详细的内容。

`sprintf` 函数和 `printf` 函数相关，它会格式化字符串而不是把它输出出来。

当生成报表时，可以使用 `format` 和 `write` 命令来格式化多行的输出。`format` 是一个非常有用的命令，但在实际使用中用的并不是很多。详细的细节可以查阅 Perl 文档，在 O'Reilly's *Programming Perl* 中有整整一章的内容介绍 `format`。你还可以在本书的第 12 章的看到 `format`。

输出至 STDOUT、STDERR 和文件

标准输出的文件句柄是 `STDOUT`，它是 Perl 程序的默认输出位置，所以不需要明确指出。下面这两个语句是完全等价的，除非你使用 `select` 改变了默认的输出文件句柄：

```
1 | print "Hello biology world!\n";
2 | print STDOUT "Hello biology world!\n";
```

注意 `STDOUT` 后面没有逗号。`STDOUT` 通常指向计算机屏幕，但是可以在命令行中把它重定向到其他程序或者文件。下面这行 Unix 命令通过管道把 `my_program` 的 `STDOUT` 定向到了 `your_program` 的 `STDIN`：

```
1 | my_program | your_program
```

而下面这行 Unix 命令则把 `my_program` 的输出定向到了 `outputfile` 文件：

```
1 | my_program > outputfile
```

把特定的错误信息定向到预先定义的标准错误文件句柄 `STDERR`，或者你已经打开用于输出的、通过特定文件句柄调用的文件，这非常常见。下面是这两种情况的例子：

```
1 | print STDERR "If you reached this part of the program, something is terribly wrong!";
2 |
3 | open(OUTPUTFD, ">output_file");
4 | print OUTPUTFD "Here is the first line in the output file output_file\n";
```

`STDERR` 默认也是定向到计算机屏幕，但是可以在命令行中把它定向到一个文件。不同操作系统中的实现方法不一样，例如下面这个例子（在 Unix 系统中使用 `sh` 或者 `bash`）：

```
1 | myprogram 2>myprogram.error
```

在 Perl 程序中，你也可以把 `STDERR` 定向到一个文件，使用下面这行代码，它会在把第一个输出信息输出到 `STDERR` 之前对其重定向。这是重定向 `STDERR` 最便捷的方法：

```
1 | open (STDERR, ">myprogram.error") or die "Cannot open error file myprogram.error:$_\n";
```

使用这种方法的问题在于原来的 `STDERR` 丢失了。下面这种方法摘抄自 *Programming Perl*，它会保存并恢复原来的 `STDERR`。

```
1 | open ERRORFILE, ">myprogram.error"
2 | or die "Can't open myprogram.error";
```



```
3 | open SAVEERR, ">&STDERR";
4 | open STDERR, ">&ERRORFILE";
5 |
6 | print STDERR "This will appear in error file myprogram.error\n";
7 |
8 | # now, restore STDERR
9 | close STDERR;
10 | open STDERR, ">&SAVEERR";
11 |
12 | print STDERR "This will appear on the computer screen\n";
```

在本书中，还有许多关于文件句柄的内容没有涉及到。并且，重定向 STDERR 等预先定义的文件句柄可能会出现问题，尤其是当你的程序越来越大、并且依赖于众多的模块和子程序库时。一种比较安全的做法是定义一个与错误文件相关联的新文件句柄，把所有的错误信息都输出到它里面去：

```
1 | open (ERRORMESSAGES, ">myprogram.error")
2 |   or die "Cannot open myprogram.error:$!\n";
3 |
4 | print ERRORMESSAGES "This is an error message\n";
```

注意一下 die 函数，以及与之密切相关的 warn 函数，它们会把错误信息输出到 STDERR。

B.15 正则表达式

正则表达式实际上是 Perl 语言里面的另外一种编程语言。在 Perl 中，它们有许多的特性。首先，我总结一下在 Perl 中正则表达式是如何工作的；之后，我会展示它们的一些特性。

B.15.1 概述

正则表达式描述字符串中的模式。使用单个正则表达式描述的模式可能会匹配众多不同的字符串。

正则表达式在模式匹配中使用，也就是说，当你要看看某个特定的模式是不是存在于一个字符串中时使用正则表达式。它们也可以改变字符串，就像替换模式的 `s///` 操作符，每当找到一个模式就会进行替换。另外，它们也在 `tr` 函数中使用，它会把字符串中的一些字符转换成要替代的其他字符。正则表达式是大小写敏感的，除非明确告诉它不区分大小写。

最简单的模式匹配是匹配它自身的一个字符串。举个例子，要看模式 ‘`abc`’ 是不是出现在了字符串 ‘`abcdefghijklmnopqrstuvwxyz`’ 中，在 Perl 中就可以这么写：

```
1 | $alphabet = 'abcdefghijklmnopqrstuvwxyz';
2 | if( $alphabet =~ /abc/ ) {
3 |     print $&;
4 | }
```

`=~` 操作符把模式匹配绑定到一个字符串上。`/abc/` 就是模式 `abc`，包裹在反斜杠 `//` 中表明这是一个正则表达式模式。如果有匹配，`$&` 就被设置成匹配到的模式。在这个例子下，匹配成功，因为 ‘`abc`’ 出现在了字符串 `$alphabet` 中，而代码仅仅会输出 `abc`。

正则表达式由两类字符构成。一类是匹配它们自身的字符，比如 ‘`a`’ 或者 ‘`z`’。一类是在正则表达式中有特殊含义的元字符 (metacharacter)。比如，小括号 `()` 不匹配它们自身，而是用于把其他的字符进行分组。如果你想匹配一个字符串中的元字符，就像 `(`，你编写模式时，必须要在元字符的前面加上反斜线，就像 `\(` 这样。

在正则表达式背后有三个基本的思想。第一个基本思想就是串联：在正则表达式模式（就是例子中两个反斜杠 `//` 之间的字符串）中相邻的两个项目，必须匹配可以待匹配字符串（刚才例子中的 `$alphabet`）中相邻的两个项目。所以要匹配 ‘`abc`’ 后面紧跟着 ‘`def`’，在正则表达式中就把它们串联起来：

```
1 | $alphabet = 'abcdefghijklmnopqrstuvwxyz';
2 | if( $alphabet =~ /abcdef/ ) {
3 |     print $&;
4 | }
```

这会输出：

```
1 | abcdef
```

第二个主要的思想是择一。被元字符 `|` 分隔开的项目会匹配其中的任何一个。例如：

```

1 | $alphabet = 'abcdefghijklmnopqrstuvwxyz';
2 | if( $alphabet =~ /a(b|c|d)c/ ) {
3 |     print $&;
4 | }
5 |
6 | ☐ ☐ ☐ ☐
7 |
8 | \begin{lstlisting}
9 | abc.
```

这个例子还演示了在正则表达式中如果使用小括号进行分组。小括号是元字符，它不会匹配字符串中它们自身，而是对选择项进行分组。对于 `b|c|d` 来说，它表示模式中那个位置可以是 `b`、`c` 和 `d` 中的任何一个。因为 `$alphabet` 中的那个位置是 `b`，所以这种择一匹配，实际上就是整个模式 `a(b|c|d)c` 能够匹配 `$alphabet`。（补充一点：`ab|cd` 表示 `(ab)|(cd)` 而不是 `a(b|c)d`。）

正则表达式第三个主要的思想就是重复（或称闭包）。它表现在模式中的量词元字符 `*`，有时也被称作克林星号，这源于正则表达式的发明人之一。当 `*` 出现在一个项目后面时，表示这个项目在字符串的那个位置可能出现 0 次、1 次或者任意次。所以，在下面这个例子中，所有的模式匹配都会成功：

```

1 | 'AC' =~ /AB*C/;
2 | 'ABC' =~ /AB*C/;
3 | 'ABBBBBBBBBBBC' =~ /AB*C/;
```

B.15.2 元字符

下面这些都是元字符：

```

1 | \ | ( ) [ { ^ $ * + ? .
```

使用 `\` 进行转义

元字符前面的反斜线 `\` 会让它匹配这些字符本身。举个例子，`\\` 会匹配字符串中的单个 `\`。

使用 `|` 进行择一匹配

如前所述，管道符 `|` 表示择一匹配。

使用 `()` 进行分组

如前所述，小括号 `()` 用于分组。

字符集

中括号 `[]` 指定一个字符集。一个字符集可以匹配指定字符中的任意一个字符。举个例子，`[abc]` 会匹配那个位置上的 `a` 或者 `b` 或者 `c`（所以和 `a|b|c` 的效果是完全一样的）。`A-Z` 是一个范围，它匹配任意的一个大写字母，`a-z` 匹配任意的一个小写字母，而 `0-9` 匹配任意的一个数字。举个例子，`[A-Za-z0-9]` 匹配那个位置上的任意单个字母或数字。如果字符集的第一个字符是 `^`，会匹配除指定字符外的任意一个字符。比如，`[^0-9]` 匹配非数字的任意一个字符。

使用. 匹配任意一个字符

句点或圆点 `.` 代表除换行符以外的任意一个字符。（使用模式修饰符 `/s` 可以让它匹配也匹配换行符。）所以，`.` 就像是一个指定了每一个字符的字符集一样。

使用 ^ 和 \$ 匹配字符串的开头和结尾

元字符 `^` 不匹配任意一个字符，而是表明其后的项目必须位于字符串的开头。与之类似，元字符 `$` 也不匹配任意一个字符，而是表明其前面的项目必须位于字符串的末尾（或者说在最后的换行符之前）。举个例子：如果字符串以 `Watson and Crick` 起始，`/^Watson and Crick/` 就能够匹配成功了；如果字符串以 `Watson and Crick` 或者 `Watson and Crick\n` 结尾，`/Watson and Crick$/` 就能够匹配成功。

量词：* + {MIN,} {MIN,MAX} ?

这些元字符表明项目的重复次数。元字符 `*` 表示前面的项目出现零次、一次或者多次。元字符 `+` 表示前面的项目出现一次或者多次。大括号 `{ }` 元字符让你指定前面项目出现的确切次数或者一个范围。比如，`{3}` 表示前面的项目正好出现了三次；`{3,7}` 表示前面的项目出现三次、四次、五次、六次或者七次；而 `{3,}` 表示前面的项目出现了三次或者更多次。元字符 `?` 匹配前面的项目零次或者一次。

通过? 限定量词进行最小化匹配

刚才介绍的量词默认都是贪婪的（进行最大化匹配），也就是说，它们会匹配尽可能多的项目。有时，你想进行最小化匹配，匹配尽可能少的项目。在 `* + { }` 每一个的后面加上 `?` 就可以实现这一点。比如，`*?` 会尝试进行尽可能少的匹配，在它尝试匹配前面项目一次或多次之前，它可能会先去尝试匹配前面项目零次。下面是一个最大化匹配的例子：

```
1 | 'hear ye hear ye hear ye' =~ /hear.*ye/;
2 | print $&;
```

它匹配 ‘hear’ 后面紧跟着 `.*`（尽可能多的字符），再后面是 ‘ye’，这会输出：

```
1 | hear ye hear ye hear ye
```

下面是一个最小化匹配的例子：

```
1 | 'hear ye hear ye hear ye' =~ /hear.*?ye/;
2 | print $&;
```

它匹配 ‘hear’ 后面紧跟着 .*?（尽可能少的字符），再后面是 ‘ye’，这会输出：

```
1 | hear ye
```

B.15.3 捕获匹配的模式

如果想知道匹配到的字符串，你可以用小括号把模式的那些部分包裹起来。比如：

```
1 | $alphabet = 'abcdefghijklmnopqrstuvwxyz';
2 | $alphabet =~ /k(lmnop)q/;
3 | print $1;
```

会输出：

```
1 | lmnop
```

在正则表达式中，你可以随意放置任意多的小括号。Perl 会自动把匹配到的子字符串存储到叫做 \$1、\$2 等的特殊变量中去。按照左小括号从左到右出现的顺序，对匹配进行编号。

下面是一个更加复杂的例子，演示了字符串中匹配模式的捕获：

```
1 | $alphabet = 'abcdefghijklmnopqrstuvwxyz';
2 | $alphabet =~ /(((a)b)c)/;
3 | print "First pattern = ", $1, "\n";
4 | print "Second pattern = ", $2, "\n";
5 | print "Third pattern = ", $3, "\n";
```

这会输出：

```
1 | First pattern = abc
2 | Second pattern = ab
3 | Third pattern = a
```

B.15.4 元符号

元符号（转义字符）是两个或多个字符序列，在正常字符的前面有一个反斜线。在 Perl 的正则表达式中（对于其中的大多数来说在双引号括起来的字符串中也是一样），这些元符号有特殊的含义。元符号不是很多，但它们逗非常有用。表 B.3 罗列了大部分的元符号。如果元符号匹配一个项目，“原子型”这一列就标明为“是”，如果它仅仅进行位置判定就标明为“否”，如果它触发了其他的行为就标明为“-”。

表 B.3: 字母数字元符号

符号	原子型	含义
\0	是	匹配空字符 (ASCII NULL)
\NNN	是	匹配八进制表示的字符, 最大到 377
\n	是	匹配第 <i>n</i> 个先前捕获的字符串 (十进制)
\a	是	匹配响铃符 (BEL)
\A	否	位于字符串的开头时为真
\b	是	匹配退格符 (BS)
\B	否	位于单词边界时为真
\b	否	不位于单词边界时为真
\cX	是	匹配控制字符 Ctrl-X
\d	是	匹配任意一个数字字符
\D	是	匹配任意一个非数字字符
\e	是	匹配转义 (escape) 符 (ASCII ESC, 而非反斜线)
\E	-	结束大小写 (\L、\U) 或元引用 (\Q) 的转换
\f	是	匹配进纸符 (FF)
\G	否	位于前一个 m//g 匹配结尾的位置时为真
\l	-	仅将下一个字符转换为小写
\L	-	将后面的字符全部转换为小写, 直到 \E 为止
\n	是	匹配换行符 (通常是 NL, 但在 Macs 中是 CR)
\Q	-	引用 (do-meta, 转义) 元字符, 直到 \E 为止
\r	是	匹配回车符 (通常是 CR, 但在 Macs 中是 NL)
\s	是	匹配任意一个空白字符
\S	是	匹配任意一个非空白字符
\t	是	匹配制表符 (HT)
\u	-	仅将下一个字符转换为大写 (单词首字母大写)
\U	-	将后面的字符全部转换为大写 (非首字母大写), 直到 \E 为止
\w	是	匹配任意一个“单词”字符 (字母数字加上 _)
\W	是	匹配任意一个非单词字符
\x{abcd}	Yes	匹配十六进制表示的字符
\z	No	仅位于字符串末尾时为真
\Z	No	位于字符串末尾或者可有可无的换行符前时为真

B.15.5 扩展正则表达式序列

表 B.4包括一些有用的特性，它们被添加到 Perl 的正则表达式中，扩展了其能力。

表 B.4: 扩展正则表达式序列

扩展	原子型	含义
(?##...)	否	注释，弃用
(?:...)	是	小括号仅用于分组聚类，不用于捕获
(?imsx-imsx)	否	启用/弃用模式修饰符
(?imsx-imsx:...)	是	小括号仅用于分组聚类，外加修饰符
(?=...)	否	如果向前判断成功就为真
(?!...)	否	如果向前判断失败就为真
(?<=...)	否	如果向后判断成功就为真
(?<!...)	否	如果向后判断失败就为真
(?>...)	是	匹配非回溯的子模式
(? {...})	否	执行嵌入的 Perl 代码
(? ? {...})	是	匹配来自嵌入 Perl 代码的正则表达式
(? (...))	是	使用 if-then-else 模式进行匹配
(? (...) ...)	是	使用 if-then 模式进行匹配

B.15.6 模式修饰符

模式修饰符是放在反斜杠后面的单字母命令。它们被用来限定正则表达式或者替换，改变一些正则表达式特性的行为。表 B.5罗列了最常用的模式修饰符，之后给出一个例子。

表 B.5: 模式修饰符

修饰符	含义
/i	忽略大小写之间的区别
/s	使. 匹配换行符
/m	使 ^ and \$ 匹配嵌入的 \n 的紧邻位置（每行的开头和结尾）
/x	忽略（大多数）空白，允许在模式中添加注释
/o	仅仅编译模式一次
/g	找到所有的匹配，而不仅仅是第一个

作为一个例子，假设你在一个文本中查找一个名字，但是你不知道这个名字是首字母大写的还是全部都是大写。你可以使用 /i 修饰符，就想这样：

```
1 | $text = "WATSON and CRICK won the Nobel Prize";
2 | $text =~ /Watson/i;
3 | print $&;
```

这会匹配（因为 /i 使得大小写之间的区别被忽略掉了）并输出匹配到的字符串 WATSON。

B.16 标量和列表上下文

Perl 中的每一个操作都会在标量上下文或者列表上下文中求值。依据所处上下文的不同，许多操作符的行为也会有所不同，在列表上下文中返回列表，在标量上下文中返回标量。

标量和列表上下文的最简单的例子就是赋值语句。如果左边（将要被赋值的变量）是一个标量变量，右边（要赋予的值）就会在标量上下文中就行求值。在下面的这个例子中，右边是一个有两个元素的数组 `@array`。当左边是一个标量变量时，它会使得 `@array` 在标量上下文中就行求值。在标量上下文中，一个数组返回这个数组中元素的个数：

```
1 | @array = ('one', 'two');
2 | $a = @array;
3 | print $a;
```

这会输出：

```
1 | 2
```

如果你用小括号把 `$a` 包裹起来，你就把它变成了只有一个元素的列表，这会使得 `@array` 在列表上下文中就行求值：

```
1 | @array = ('one', 'two');
2 | ($a) = @array;
3 | print $a;
```

这会输出：

```
1 | one
```

注意，当给列表进行赋值时，如果没有足够的变量存储所有的值，多余的值会被简单的丢弃掉。要捕获所有的变量，你可以这么做：

```
1 | @array = ('one', 'two');
2 | ($a, $b) = @array;
3 | print "$a $b";
```

这会输出：

```
1 | one two
```

类似的，如果左边的变量数目多于右边的变量数目，多余的变量会直接被赋值为非定义的值 `undef`。

当查阅 Perl 函数和操作符的问当时，一定要注意文档中对于标量上下文和列表上下文的描述。如果你的程序表现很诡异，通常情况下，是因为它在和你预期不同的上下文中进行的求值。

下面是预估标量上下文和列表上下文的一些常用的准则：

- 以下情况下得到的是列表上下文：函数调用（在参数位置的所有内容都在列表上下文中就行求值）和列表赋值。
- 以下情况下得到的是标量上下文：字符串和数字操作符（像 `.` 和 `+` 等操作符的参数都被假设为标量）；布尔测试，比如 `if ()` 语句的条件测试和 `||` 逻辑操作符的参数；标量赋值。

B.17 子程序和模块

要定义子程序，使用关键词 `sub`，后面写上子程序的名字，之后就是用大括号 `{ }` 包括起来的代码块，代码块就是子程序的主体。下面是一个简单的例子：

```
1 | sub a_subroutine {
2 |     print "I'm in a subroutine\n";
3 | }
```

通常来说，你要调用子程序，只需要使用子程序的名字，后面跟上用小括号包裹起来的参数列表即可：

```
1 | a_subroutine();
```

参数可以以标量列表的形式传递给子程序。如果把一个数组作为参数，数组的元素会被展开成标量列表。子程序把所有的标量值以列表进行接收，保存到特殊变量 `@_` 中。下面这个例子展示了子程序的定义，以及使用一些参数调用子程序：

```
1 | sub concatenate_dna {
2 |     my ($dna1, $dna2) = @_;
3 |     my ($concatenation);
4 |     $concatenation = "$dna1$dna2";
5 |     return $concatenation;
6 | }
7 |
8 | print concatenate_dna('AAA', 'CGC');
```

这会输出：

```
1 | AAACGC
```

参数 ‘AAA’ 和 ‘CGC’ 作为标量列表传递给子程序，子程序代码块中的第一个语句：

```
1 | my ($dna1, $dna2) = @_;
```

把保存在特殊变量 `@_` 中的这个列表赋值给变量 `$dna1` 和 `$dna2`。

把变量 `$dna1` 和 `$dna2` 声明为 `my` 变量以保证它们只局限在自程序的代码块中。通常情况下，你应该把所有的变量都声明为 `my` 变量；当你在程序开头处加上 `use strict;` 语句后这就变成了强制行为。然而，使用不用 `my` 声明的全局变量也是可能的，这样它就可以在程序的任何地方都可以使用了，包括子程序内部。在本书中，我还没有使用过全局变量。

下面这个语句：

```
1 | my ($concatenation);
```

声明了供子程序使用的另一个变量。
在这个语句之后：

```
1 | $concatenation = "$dna1$dna2";
```

完成了子程序的任务，子程序使用 `return` 语句定义了它的值：

```
1 | return $concatenation;
```

调用子程序后返回的值可以在你想用的任何地方使用。在这个例子中，它直接作为 `print` 函数的参数。

如果把数组作为参数，数组中的元素会被展开存储到 `@_` 列表中，就行下面这个例子演示的一样：

```
1 | sub example_sub {
2 |     my (@arguments) = @_;
3 |
4 |     print "@arguments\n";
5 | }
6 |
7 | my @array = ('two', 'three', 'four');
8 |
9 | example_sub('one', @array, 'five');
```

这会输出：

```
1 | one two three four five
```

注意，下面这个例子试图在子程序的参数中混用数组和标量，但这并不会工作：

```
1 | # This won't work!!
2 | sub bad_sub {
3 |     my (@array, $scalar) = @_;
4 |
5 |     print $scalar;
6 | }
7 |
8 | my @arr = ('DNA', 'RNA');
9 | my $string = 'Protein';
10 |
11 | bad_sub(@arr, $string);
```

在这个例子中，赋值语句左边的子程序变量 `@array` 会把右边的 `@_` 整个列表（也就是 `'DNA' 'RNA' 'Protein'`）都囊括进来。子程序变量 `$scalar` 不会被赋值，所以子程序不会如预期一样把 `'Protein'` 打印输出出来。要把单独的数组和散列传递给子程序，你需要使用引用；请参看第 6 章中的第 6.4.1 小节。下面是一个简短的例子：

```
1 sub good_sub {  
2     my ($arrayref, $hashref) = @_;  
3  
4     print "@$arrayref", "\n";  
5  
6     my @keys = keys %$hashref;  
7  
8     print "@keys", "\n";  
9 }  
10  
11 my @arr = ('DNA', 'RNA');  
12 my %nums = ( 'one' => 1, 'two' => 2 );  
13  
14 good_sub(\@arr, \%nums);
```

这会输出：

```
1 DNA RNA  
2 one two
```

B.18 内置函数

Perl 有大量的内置函数。表 B.6只是罗列了其中的一部分，对它们进行了简短的描述。

表 B.6: Perl 的内置函数

函数	概要
abs VALUE	返回它的数值参数的绝对值
atan2 Y, X	返回从 $-\pi$ 到 π 之间的 Y/X 的反正切值
chdir EXPR	切换工作路径到 EXPR（或者默认切换到家目录）
chmod MODE LIST	把 LIST 中的文件的权限修改为 MODE
chomp (VARIABLE or LIST)	如果有的话，删除字符串末尾的换行符
chop (VARIABLE or LIST)	删除字符串末尾的字符
chown UID, GID, LIST	把 LIST 中的文件的所有者和所属组修改为以数字表示的 UID 和 GID
close FILEHANDLE	关闭和 FILEHANDLE 相关联的文件、套接字或者管道
closedir DIRHANDLE	关闭和 DIRHANDLE 相关联的目录
cos EXPR	返回以弧度表示的 EXPR 的余弦值
dbmclose HASH	打断 DBM 文件和散列之间的绑定
dbmopen HASH, DBNAME, MODE	以 MODE 权限把 DBM 文件绑定到一个散列上
defined EXPR	如果 EXPR 有一个定义的值返回真，否则返回假
delete EXPR	删除散列或数组的元素（或切片）
die LIST	输出包含 LIST 的错误信息，退出程序
each HASH	一次一个得遍历散列的键或者键-值对
exec PATHNAME LIST	终止程序，以参数 LIST 执行程序 PATHNAME
exists EXPR	如果散列的键或者数组索引存在返回真
exit EXPR	以返回值 EXPR 退出程序
exp EXPR	返回 e（自然对数的底）的 EXPR 次方的值
format	声明一个格式供 write 函数使用
grep EXPR, LIST	返回 EXPR 为真的 LIST 的元素列表

待续...

(续表B.6)

函数	概要
<code>gmtime</code>	获取格林尼治标准时间；星期天是第 0 天，一月是第 0 月，年是自 1900 年至今的年数。例子： <code>(\$sec,\$min,\$hour,\$mday,\$mon,\$year,\$wday,\$yday,\$isdaylightsavingstime) = gmtime;</code>
<code>goto LABEL</code>	程序控制，跳转至标记为 LABEL 的语句
<code>hex EXPR</code>	返回十六进制数 EXPR 的十进制值
<code>index STR, SUBSTR</code>	给出 STR 中 SUBSTR 第一次出现的位置
<code>int EXPR</code>	给出数字 EXPR 的整数部分
<code>join EXPR, LIST</code>	把 LIST 中的多个字符串合并成单个的字符串，用 EXPR 分隔开
<code>keys HASH</code>	返回散列 HASH 所有键的列表
<code>last LABEL</code>	默认立即跳出最内部的循环，或者跳出标记为 LABEL 的循环
<code>lc EXPR</code>	返回 EXPR 字符串的小写格式
<code>lcfirst EXPR</code>	返回 EXPR 的首字母小写格式
<code>length EXPR</code>	返回 EXPR 的字符长度
<code>localtime</code>	获取地方时间，格式同 <code>gmtime</code> 函数
<code>log EXPR</code>	返回数字 EXPR 的自然对数
<code>m/PATTERN/</code>	匹配正则表达式 PATTERN 的匹配操作符，通常简写为 <code>/PATTERN/</code>
<code>map BLOCK LIST (or map EXPR, LIST)</code>	针对 LIST 的每个元素，对 BLOCK 或者 EXPR 进行求值，返回返回值的列表
<code>mkdir FILENAME</code>	创建目录 <i>FILENAME</i>
<code>my EXPR</code>	把 EXPR 中的变量限定内内层的代码块中
<code>next LABEL</code>	默认进入内层循环的下一个迭代，或者进入用 LABEL 标记的循环的下一个迭代
<code>oct EXPR</code>	返回八进制数 EXPR 的十进制值
<code>open FILEHANDLE, EXPR</code>	打开一个文件，以 EXPR 中的选项把它关联至 FILEHANDLE
<code>opendir DIRHANDLE, EXPR</code>	打开目录 EXPR，并且关联至句柄 DIRHANDLE

待续...

(续表B.6)

函数	概要
pop ARRAY	移除并返回数组 ARRAY 的最后一个元素
pos SCALAR	给出上一个 m//g 查找在字符串 SCALAR 中的位置
print FILEHANDLE LIST	把字符串列表输出至 FILEHANDLE (默认是 STDOUT)
printf FILEHANDLE FORMAT, LIST	把用 FORMAT 格式指定的字符串和变量 LIST 输出至 FILEHANDLE
push ARRAY, LIST	把 LIST 的元素放到数组 ARRAY 的末尾
rand EXPR	给出 0 至 (小于) EXPR (默认为 1) 之间的伪随机十进制数
readdir DIRHANDLE	返回目录 DIRHANDLE 的内容列表
redo LABEL	在不对条件进行再求值的前提下重新运行循环代码块
ref EXPR	如果是引用返回真, 否则返回假: 如果为真, 返回表示引用类型的值
rename OLDNAME, NEWNAME	修改文件的名字
return EXPR	返回值 EXPR, 退出当前子程序
reverse LIST	以逆序返回 LIST, 或者在标量上下文中反转字符串
rindex STR, SUBSTR	与 index 函数类似, 但返回的是 STR 中 SUBSTR 最后一次出现的位置
rmdir FILENAME	删除目录 FILENAME
s/PATTERN/REPLACEMENT/	用字符串 REPLACEMENT 替换匹配的正则表达式 PATTERN
scalar EXPR	强制在标量上下文中对 EXPR 求值
seek FILEHANDLE, OFFSET, WHENCE	把 FILEHANDLE 的文件指针定位到 OFFSET 字节 (在 WHENCE 是 0 的情况下, 如果 WHENCE 是 1 就定位到当前位置加上 OFFSET, 如果 WHENCE 是 2 就定位到距离末尾的 OFFSET 字节处)
shift ARRAY	删除并返回数组的第一个元素
sin EXPR	返回以弧度表示的 EXPR 的正弦值
sleep EXPR	让程序沉睡 EXPR 秒

待续...

(续表B.6)

函数	概要
sort USERSUB LIST (or sort BLOCK LIST)	以 USERSUB 或者 BLOCK 指定的顺序对 LIST 进行排序（默认按照标准字符串的顺序）
splice ARRAY, OFFSET, LENGTH, LIST	从 OFFSET 开始，删除 ARRAY 中的 LENGTH 个元素，如果有 LIST 就把它们替换成 LIST
split /PATTERN/, EXPR	在出现/PATTERN/的地方对字符串 EXPR 进行分割，返回列表
sprintf FORMAT, LIST	返回一个格式化的字符串，就像 printf 函数一样
sqrt EXPR	返回数字 EXPR 的平方根
srand EXPR	为 rand 操作符设定随机数种子；只在 5.004 版本之前的 Perl 中需要
stat (FILEHANDLE or EXPR)	返回文件 EXPR 或者它的文件句柄 FILEHANDLE 的统计信息。例子：(<code>\$dev,\$inode,\$mode,\$num_of_links,\$uid,\$gid,\$rdev,\$size,\$accesstime,\$modifiedtime,\$changetime,\$blksize,\$blocks</code>) = stat \$filename;
study SCALAR	尝试对接下来针对字符串 SCALAR 的模式匹配进行优化
sub NAME BLOCK	使用 BLOCK 中的程序代码定义一个名为 NAME 的子程序
substr EXPR, OFFSET, LENGTH, REPLACEMENT	返回字符串 EXPR 从 OFFSET 位置开始长度为 LENGTH 的子字符串；如果有 REPLACEMENT 就把子字符串替换成 RELACEMENT
system PATHNAME LIST	使用参数 LIST 执行程序 PATHNAME；返回程序的退出状态而非其输出；使用反引号捕获输出。例子： <code>@output = `bin/who`;</code>
tell FILEHANDLE	返回在 FILEHANDLE 中的当前文件位置，以字节表示
tr/ORIGINAL/REPLACEMENT/	把 ORIGINAL 中的每一个字符转换成 REPLACEMENT 中对应的字符
truncate (FILEHANDLE or EXPR), LENGTH	截断文件 EXPR 或者使用 FILEHANDLE 打开至 LENGTH 字节
uc EXPR	返回字符串 EXPR 的大写形式

待续...

(续表B.6)

函数	概要
ucfirst EXPR	返回字符串 EXPR 的首字母大写形式
undef EXPR	返回未定义值；如果 EXPR 是一个已定义的变量或者子程序，它就不再是已定义的了；当你不需要保存值的时候可以用它进行赋值
unlink LIST	删除 LIST 中的文件
unshift ARRAY, LIST	把 LIST 中的元素添加到数组 ARRAY 的开头
use MODULE	载入模块 MODULE
values HASH	返回散列 HASH 的所有值的列表
wantarray	在子程序中，如果调用的程序预期返回一个列表值，它就会返回真
warn LIST	输出包括 LIST 在内的错误信息
write FILEHANDLE	按照 format 函数中的定义，把格式化的记录写入到 FILEHANDLE（默认为 STDOUT）

版权页

我们希望看到的结果是读者的评论、我们自己的实验，以及来自分布式频道的反馈。独特的封面反映了我们对于不同技术主题的独特态度，把个性和人性引入到了可能枯燥无味的主题中。

Beginning Perl for Bioinformatics 这本书封面上的动物是青铜蛙 (*Rana clamitans*) 和美国牛蛙 (*Rana catesbeiana*) 的蝌蚪。

蝌蚪是青蛙和蟾蜍的幼虫。它们是水生动物，刚孵化出来的时候，有着大而圆的头部和长而扁的尾巴。经过一个复杂的变态过程，蝌蚪从小的鱼形生物变成了更为人们熟知的青蛙和蟾蜍。根据物种的不同，变态过程需要从 10 天到 3 年不等的时间。

在变态的第一个阶段，蝌蚪的后腿先萌芽，头部开始扁平起来，尾巴逐渐变短。在生命的早期，蝌蚪主要以硅藻、水藻和少量的浮游生物为食。随着变态的继续，当它的消化系统从以素食为主变成肉食时，它停止进食，并开始重吸收自己的尾巴作为营养来源。在变态的最后阶段，蝌蚪的前腿出现、颌形成、骨架硬化，并且随着肺的发育腮会逐渐消失。一个短暂的时间之后，蝌蚪从水中出来，重吸收尾巴的最后部分，开始像青蛙或者蟾蜍那样跳跃。

Mary Anne Weeks Mayo 是 *Beginning Perl for Bioinformatics* 的出版商编辑和文字编辑。Matt Hutchinson 和 Jane Ellin 进行了质量控制。Edie Shapiro、Matt Hutchinson 和 Derek DiMatteo 提供了出版援助。Ellen Troutman-Zaig 编写了索引。

Ellie Volckhausen 基于 Edie Freedman 的系列设计，设计了本书的封面。封面图片是 Lorrie Lejeune 创作的原版插图。Emma Colby 使用 Adobe's ITC Garamond 字体、基于 Quark™XPress 4.1 对封面进行了排版。

Melanie Wang 基于 David Futato 的系列设计，设计了内部的版式布局。Neil Walls 使用 Mike Sierra 创建的工具把文件从 SGML 转换到了 FrameMaker 5.5.6。文本的字体是 Linotype Birka，标题的字体是 Adobe Myriad Condensed，代码的字体是 LucasFont's TheSans Mono Condensed。书中出现的插图都是由 Robert Romano 和 Jessamyn Read 使用 Macromedia FreeHand 9 和 Adobe Photoshop 6 制作的。忠告和警告由 Christopher Bing 绘制。Lorrie Lejeune 撰写了该版权页。