在R语言中处理和处理字符串

Gaston Sanchez
www.gastonsanchez.com

关于本电子书

摘要

本电子书旨在帮助您开始使用R语言进行字符串操作。 尽管在字符串处理方面，R语言存在一些问题，但我们中的一些人认为R语言在计算字符串和文本方面非常好用。 当涉及字符串操作时，R语言可能不像其他脚本语言那样丰富多样，但如果您了解如何使用，它可以带您走得很远。希望本文本能为您提供足够的材料来进行更高级的字符串和文本处理操作。

关于读者

我假设关于你的三件事情，按重要性递减：

1. 你已经了解 R—这不是一本关于 R入门的教材—。
2. 你已经使用 R处理定量和定性数据，但不一定用它来处理字符串。
3. 你对正则表达式有一些基本知识。

引用

你可以引用这个作品：

Sanchez, G. (2013)在**R**语言中处理和处理字符串
Trowchez Editions. 伯克利, 2013.

http://www.gastonsanchez.com/在R语言中处理和处理字符串.pdf

修订

版本 1.3 (2014年3月)

目录

前言

如果你接受过“经典统计学”（就像我一样），我敢打赌你可能不会把字符字符串视为可以分析的数据。 对于你的分析来说，最重要的是数字或可以映射为数值的事物。 文本和字符字符串？真的吗？你在开玩笑吗？. . .. 那是我在大学毕业后的想法。. 在我统计学的本科学习期间，我的教授们从未提到过使用字符串和文本数据进行分析的应用。 几年后，在研究生学习期间，我有机会参与一些统计文本分析的工作。

更糟糕的是，很多人普遍认为字符串处理是次要且不相关的任务。 人们会对你应用的任何花哨模型、复杂算法和黑盒方法感到印象深刻并钦佩。 每个人都喜欢数据分析的高级烹饪和顶级分析。 但是当涉及到处理和操作字符串时，很多人会认为这就像洗碗或削皮切土豆一样无聊。 如果你想被视为数据厨师，你可能会认为不应该浪费时间在那些无聊的字符串操作上。 是的，没错，处理字符数据不会让你获得米其林星级评价。 但是如果你不通过字符串操作来获得实践经验，你很难成为一名优秀的数据厨师。 说实话，并不总是那么无聊。 无论你喜不喜欢，除非你进行过字符串操作，否则你永远不能自称为数据分析师。 你不必成为专家或字符串处理黑客。 但是你需要了解基础知识，并知道在需要处理文本字符字符串数据时如何进行。

关于如何在 R中操作字符串和文本数据的文档非常稀缺。 这主要是因为 R不被视为一种脚本语言（如Python或Java等）。 然而，我认为我们需要更多关于这个不可或缺的主题的资源。 这项工作是我在这个被大部分人遗忘的领域中的一点建议。

Gaston Sanchez
法国南特
2013年9月

第1章

介绍

在R中处理和处理文本字符串？等一下. . . 你惊呼道，R不是一种
像$Perl$、$Python$或$Ruby$那样的脚本语言。. 为什么你想要使用 R来处理
和处理文本？ 嗯，因为迟早（我会说比较早）你都会
不得不处理某种形式的字符串操作来进行数据分析。 所以最好
为这些任务做好准备，并知道如何在 R环境中执行它们。

我相信会有人告诉你，在R中能做某件事情并不意味着你应该这样做。 我在某种程度上同意。 是的，你可能不应该使用像Python、Perl或Ruby这样的语言来执行那些更好完成的任务。然而，在许多情况下，留在 R环境中更好（即使只是出于方便或在 R中拖延）。 如果必要，我肯定会使用Python，但只要可能，我会尽量使用 R（个人口味）。

1.1 一些资源

关于如何操作字符字符串和文本数据的文档不多
R. 有很多关于统计方法、图形和数据可视化的优秀的R语言书籍，以及在生态学、遗传学、心理学、金融学、经济学等各个领域的应用。但是没有关于操作字符串和文本数据的书籍。

这种资源匮乏的主要原因可能是R语言不被认为是一种“脚本”语言：R语言主要被视为一种用于计算和编程的语言（主要是数值数据）。 引用Hadley Wickham（2010）http://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf

> “R提供了一套可靠的字符串操作，但由于它们是逐渐发展起来的，所以它们可能不一致并且有点难以学习。此外，它们滞后于其他语言的字符串操作功能。”

> 在其他编程语言中，字符串操作的背后，有一些在Ruby或Python等语言中容易做到的事情在R中却相对困难

大多数关于 R的入门书籍都有简短介绍字符串操作的小节，但没有进一步深入讲解 这就是为什么我没有太多推荐的书籍，如果有的话，我会推荐Phil Spector的《使用*R*进行数据处理》

即使出版物不多，我们仍然有在线世界 好消息是网络上有数百个关于处理字符字符串的参考资料 坏消息是它们非常分散和无分类

对于特定的主题和任务，一个好的起点是*S*tack *O*ver*f*low 这是一个面向程序员的问答网站，有很多与 R相关的问题 只需搜索那些标记为 "r"的问题：http://stackoverflow.com/questions/tagged/r 有很多与处理字符和文本相关的帖子，它们可以给你一个解决特定问题的提示 还有 *R-bloggers*，http://www.r-bloggers.com，这是一个为 R爱好者提供博客聚合的网站，上面也可以找到关于处理字符串和文本数据分析的贡献材料

您还可以查看以下与字符串操作相关的资源。 这是一个
非常简短的资源列表，但我发现它们非常有用：

- R Wikibook：编程和文本处理
  http://en.wikibooks.org/wiki/R_Programming/Text_Processing
  R wikibook有一个专门介绍文本处理的部分，值得一看。

- stringr：现代的、包含字符串处理的库by Hadley Wickham
  http://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf
  这是Hadley Wickham在R杂志上介绍stringr包的文章。

- 使用正则表达式在R中进行字符串匹配和修改的简介by Svetlana Eden
  http://biostat.mc.vanderbilt.edu/wiki/pub/Main/SvetlanaEdenRFiles/regExprTalk.pdf

1.2 字符串和数据分析

本书旨在帮助您开始处理R中的字符串。 它提供了一些可以用于字符串操作的资源概述。它涵盖了有用的函数、常见操作和其他技巧。

这本书不是关于文本数据分析，语言分析，文本挖掘或自然
语言处理。 为了这些目的，我强烈建议你看一下

CRAN任务视图上的自然语言处理（NLP）：
http://cran.r-project.org/web/views/NaturalLanguageProcessing.html

然而，即使你不打算进行文本分析，文本挖掘或自然语言处理，我敢打赌你有一些包含字符数据的数据集：行名，列名，日期，货币数量，经度和纬度等。我相信你遇到过以下一种或多种情况：

- 你想在变量名中删除给定的字符
- 你想在数据中替换给定的字符
- 也许你想将标签转换为大写（或小写）
- 你一直在处理xml（或html）文件
- 你一直在excel中修改文本文件，改变标签，类别，一次一个单元格，或进行一千次复制粘贴操作

希望在阅读本书后，你将拥有处理和处理字符串的必要工具，以应对这些（以及许多其他）涉及在R中处理和处理字符串的情况。

1.3 一个玩具示例

为了给你一个关于在R中进行字符串处理的一些事情的想法，让我们玩一个简单的例子。我们将使用数据框USArrests，它已经包含在R中，进行这次简要介绍。 使用函数 head()来查看数据：# 查看USArrests的前几行head(USArrests)

```
##                 谋杀 攻击 城市人口 强奸
## 阿拉巴马        13.2    236       58 21.2
## 阿拉斯加        10.0    263       48 44.5
## 亚利桑那         8.1    294       80 31.0
## 阿肯色           8.8    190       50 19.5
## 加利福尼亚       9.0    276       91 40.6
## 科罗拉多         7.9    204       78 38.7
```

行的标签，如阿拉巴马或阿拉斯加，是显示的字符串。 同样，列的标签——谋杀、攻击、城市人口和强奸——也是字符串。

缩写字符串

假设我们想要缩写州的名称。此外，假设我们想要使用每个名称的前四个字符来缩写。 一种方法是使用函数 substr()，它可以对字符向量进行子字符串处理。 我们只需要指定开始位置为1和结束位置为4：# 州的名称states = rownames(USArrests)

```
# substr
substr(x = states, start = 1, stop = 4)
```

```
## [1] "阿拉巴马" "阿拉斯加" "亚利桑那" "阿肯色" "加利福尼亚" "科罗拉多" "康涅狄格" "特拉华" "佛罗里达" "乔治亚"
## [11] "夏威夷" "爱达荷" "伊利诺伊" "印第安纳" "爱荷华" "堪萨斯" "肯塔基" "路易斯安那" "缅因" "马里兰"
## [21] "马萨诸塞" "密歇根" "明尼苏达" "密西西比" "密苏里" "蒙大拿" "内布拉斯加" "内华达" "新罕布什尔" "新泽西"
## [31] "新墨西哥" "纽约" "北卡罗来纳" "北达科他" "俄亥俄" "俄克拉荷马" "俄勒冈" "宾夕法尼亚" "罗得岛" "南卡罗来纳"
## [41] "南达科他" "田纳西" "德克萨斯" "犹他" "佛蒙特" "弗吉尼亚" "华盛顿" "西弗吉尼亚" "威斯康星" "怀俄明"
```

这可能不是最好的解决方案。 请注意，有四个州具有相同的缩写 "新"（新罕布什尔州，新泽西州，新墨西哥州，纽约州）。同样，北卡罗来纳州和北达科他州有相同的名字 "Nort"。 反过来，南卡罗来纳州和南达科他州有相同的缩写 "Sout"。

通过使用函数abbreviate()，可以更好地缩写州的名称，如下所示：# 缩写州名states2 = abbreviate(states)

```
# 删除向量名称（为了方便起见）
names(states2) = NULL
states2
```

```
##  [1] "Albm" "Alsk" "Arzn" "Arkn" "Clfr" "Clrd" "Cnnc" "Dlwr" "Flrd" "Gerg"
## [11] "Hawa" "Idah" "Illn" "Indn" "Iowa" "Knss" "Kntc" "Losn" "Main" "Mryl"
## [21] "Mssc" "Mchg" "Mnns" "Msss" "Mssr" "Mntn" "Nbrs" "Nevd" "NwHm" "NwJr"
## [31] "NwMx" "NwYr" "NrtC" "NrtD" "Ohio" "Oklh" "Orgn" "Pnns" "RhdI" "SthC"
## [41] "SthD" "Tnns" "Texs" "Utah" "Vrmn" "Vrgn" "Wshn" "WstV" "Wscn" "Wymn"
```

如果我们决定尝试一个有六个字母的缩写，我们只需简单地改变参数 min.ength = 5

```
# 使用5个字母缩写州名
abbreviate(states, minlength = 5)
```

```
##          阿拉巴马          阿拉斯加          亚利桑那          阿肯色      加利福尼亚
##         "阿拉巴马"        "阿拉斯加"        "亚利桑那"        "阿肯色"      "加利福尼亚"
##          科罗拉多      康涅狄格          特拉华          佛罗里达          乔治亚
##         "科罗拉多"        "康涅狄格"        "特拉华"        "佛罗里达"        "乔治亚"
##          夏威夷          爱达荷          伊利诺伊          印第安纳          爱荷华
##         "夏威夷"        "爱达荷"        "疾病"        "印第安"        "爱荷华"
##          堪萨斯          肯塔基          路易斯安那          缅因          马里兰
##         "坎萨斯"        "肯塔基"        "路易斯安那"        "缅因"        "马里兰"
## 麻萨诸塞          密歇根          明尼苏达          密西西比          密苏里
##         "马萨诸塞"        "密歇根"        "明尼苏达"        "Mssss"        "Missr"
##          蒙大拿          内布拉斯加          内华达 新罕布什尔      新泽西
##         "Montn"        "Nbrsk"        "Nevad"        "NwHmp"        "NwJrs"
##          新墨西哥          纽约 北卡罗来纳          北达科他          俄亥俄
##         "NwMxc"        "NwYrk"        "NrthC"        "NrthD"        "Ohio"
##          俄克拉荷马          俄勒冈      宾夕法尼亚      罗得岛 南卡罗来纳
##         "Oklhm"        "Oregn"        "Pnnsy"        "RhdIs"        "SthCr"
##          南达科他          田纳西          德克萨斯          犹他          佛蒙特
##         "SthDk"        "Tnnss"        "德克萨斯"        "犹他"        "Vrmnt"
##          弗吉尼亚      华盛顿西弗吉尼亚          威斯康星          怀俄明
##         "Virgn"        "Wshng"        "WstVr"        "Wscns"        "Wymng"
```

获取最长的名字

现在让我们想象一下，我们需要找到最长的名字。 这意味着我们需要计算每个名字中的字母数。 函数 `nchar()`非常方便用于此目的。

这是我们可以这样做的方式：

```
# 每个名称的大小（以字符为单位）
state_chars = nchar(states)

# 最长的名称
states[which(state_chars == max(state_chars))]
```

```
## [1] "北卡罗来纳州" "南卡罗来纳州"
```

选择州

为了使事情更有趣，假设我们希望选择那些包含字母 `"k"`的州。我们该如何做到这一点？非常简单，我们只需要使用函数 `grep()`来处理正则表达式。 只需将模式指定为`"k"`，如下所示：

```
# 获取包含 'k' 的州名
grep(pattern = "k", x = states, value = TRUE)
```

```
## [1] "阿拉斯加州"       "阿肯色州"       "肯塔基州"       "内布拉斯加州"
## [5] "纽约州"           "北达科他州" "俄克拉荷马州"       "南达科他州"
```

与其获取那些包含 "k"的名称，假设我们希望选择那些包含字母 "w"的州。 同样，可以使用 grep()来实现：# 获取包含 'w' 的州名

```
grep(pattern = "w", x = states, value = TRUE)
```

```
## [1] "特拉华州"         "夏威夷州"       "爱荷华州"       "新罕布什尔州"
## [5] "新泽西州"         "新墨西哥州"     "纽约州"
```

请注意，我们只选择了那些带有小写字母 "w"的州。 但是那些带有大写字母 "W"的州呢？有几种方法可以解决这个问题。 其中一种方法是将搜索的模式指定为字符类 "[wW]"：# 获取带有'w'或'W'的州名grep(pattern = "[wW]", x = states, value = TRUE)## [1]

```
 "特拉华州"
```

```
                        "夏威夷州"       "爱荷华州"       "新罕布什尔州"
## [5] "新泽西州"         "新墨西哥州"     "纽约州"         "华盛顿州"
## [9] "西弗吉尼亚州" "威斯康星州"         "怀俄明州"
```

另一种解决方案是先将州名转换为小写，然后查找字符 "w"，如下所示：# 获取带有'w'的州名

```
grep(pattern = "w", x = tolower(states), value = TRUE)
```

```
## [1] "特拉华州"         "夏威夷"         "爱荷华"         "新罕布什尔"
## [5] "新泽西"           "新墨西哥"       "纽约"           "华盛顿"
## [9] "西弗吉尼亚" "威斯康星"                "怀俄明"
```

或者，我们可以不将州名转换为小写，而是相反地将其转换为大写，然后查找字符 "W"，像这样：# 获取以'W'开头的州名

```
grep(pattern = "W", x = toupper(states), value = TRUE)
```

```
## [1] "特拉华州"         "夏威夷"         "爱荷华"         "新罕布什尔"
## [5] "新泽西"           "新墨西哥"       "纽约"           "华盛顿"
## [9] "西弗吉尼亚" "威斯康星"                "怀俄明"
```

第三种解决方案，也许是最简单的一种，是指定参数ignore.case=TRUE

在 grep()内部：

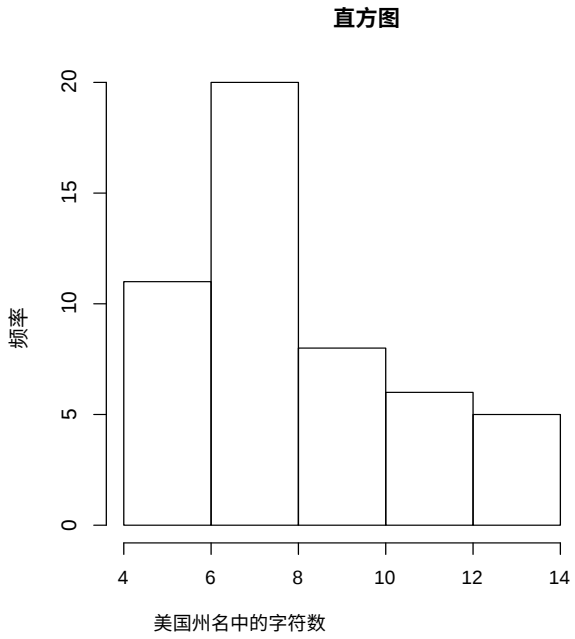
```
# 获取以'w'开头的州名
grep(pattern = "w", x = states, value = TRUE, ignore.case = TRUE)
```

```
## [1] "特拉华州"         "夏威夷州"         "爱荷华州"         "新罕布什尔州"
## [5] "新泽西州"         "新墨西哥州"       "纽约州"           "华盛顿州"
## [9] "西弗吉尼亚州" "威斯康星州"       "怀俄明州"
```

一些计算

除了操作字符串和执行模式匹配操作外，我们还可以进行一些计算。 例如，我们可以询问州名长度的分布。 要找到答案，我们可以使用 nchar()。 此外，我们还可以绘制这种分布的直方图：

```
# 直方图
hist(nchar(states), main = "直方图",
     xlab = "美国州名中的字符数")
```



让我们提出一个更有趣的问题。 州名中元音字母的分布是什么？ 例如，让我们从每个名称中的 a的数量开始。 有一个非常有用的函数可以实现这个目的： regexpr()。 我们可以使用 regexpr()来获取在字符向量中找到搜索模式的次数。 当没有匹配时，我们会得到

一个值-1。

```
# a的位置
positions_a = gregexpr(pattern = "a", text = states, ignore.case = TRUE)

# 有多少个a?
num_a = sapply(positions_a, function(x) ifelse(x[1] > 0, length(x), 0))
num_a
```

```
##  [1] 4 3 2 3 2 1 0 2 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2 1 0 0 0 2 2 0
## [36] 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0
```

如果你检查positions a，你会发现它包含一些负数-1。 这意味着那个名字中没有字母a。 为了得到a的出现次数，我们使用了sapply()的快捷方式。 同样的操作可以通过使用来自"stringr"包的str_count()函数来执行。

-

```
# 加载stringr（记得先安装它）
library(stringr)

# a的总数
str_count(states, "a")
```

```
##  [1] 3 2 1 2 2 1 0 2 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2 1 0 0 0 2 2 0
## [36] 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0
```

注意我们只得到了小写a的数量。 由于str_count()不包含ignore.case参数，我们需要将所有字母转换为小写，然后像这样计算a的数量：# a的总数str_count(tolower(states), "a")

```
##  [1] 4 3 2 3 2 1 0 2 1 1 2 1 0 2 1 2 0 2 1 2 2 1 1 0 0 2 2 2 1 0 0 0 2 2 0
## [36] 2 0 2 1 2 2 0 1 1 0 1 1 1 0 0
```

一旦我们知道如何处理一个元音字母，我们就可以对所有元音字母做同样的处理：

```
# 元音字母的向量
vowels = c("a", "e", "i", "o", "u")

# 用于存储结果的向量
num_vowels = vector(mode = "integer", length = 5)

# 计算每个名称中的元音字母数量
for (j in seq_along(vowels))  {
```

```
    num_aux = str_count(tolower(states), vowels[j])
    num_vowels[j] = sum(num_aux)
}

# 添加元音字母名称
names(num_vowels) = vowels

# 总元音字母数量
num_vowels
```

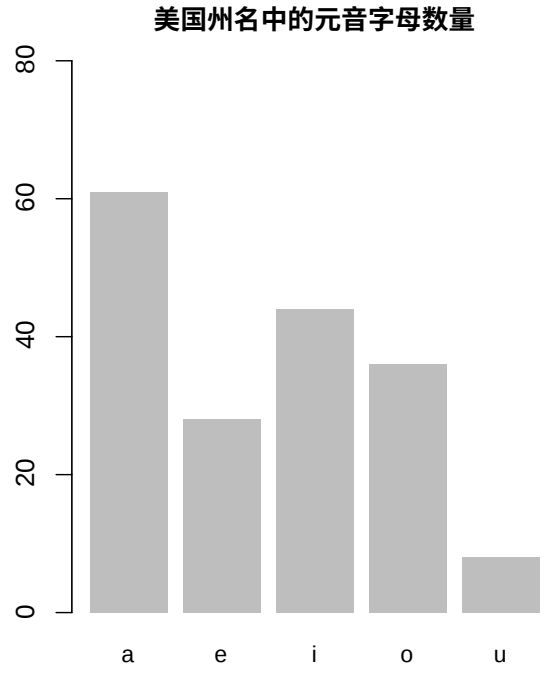
```
##  a  e  i  o  u
## 61 28 44 36  8
```

```
# 按降序排序
sort(num_vowels, decreasing = TRUE)
```

```
##  a  i  o  e  u
## 61 44 36 28  8
```

最后，我们可以用条形图来可视化分布：

```
# 条形图
条形图（num_vowels，主要 = "美国州名中的元音字母数量",
        边界 = NA，ylim = c(0, 80))
```



1.4 概述

前面的例子只是这本书中包含的菜单的开胃菜
美国州名。 本书的其余部分分为六个章节：

第2章：R中的字符字符串

第3章：字符串操作

第4章：使用stringr进行字符串操作

第5章：正则表达式（第一部分）

第6章：正则表达式（第二部分）

第7章：实际应用

第2章致力于向您展示在 R中处理字符字符串的基础知识。 我们在 R中如何获取文本数据以及处理字符时 R对象的行为。

第3章描述了一系列可以在不使用正则表达式的情况下用于操作字符串的函数。 这个想法是覆盖在 R的基本分发中的那些函数。

第4章旨在展示R包中的字符串操作函数`stringr`。 与第3章类似，第4章的内容仅限于那些不使用正则表达式的函数。

第5章是关于在R中使用正则表达式的讨论的第一部分。 基本上，本章的目的是讨论R与正则表达式的工作方式（与其他脚本语言相比）。

第6章继续讨论正则表达式。 它涵盖了基本包和 `stringr`包中的正则表达式函数的应用。

第7章是最后一章，我们将讨论一些具有实际应用的示例。
我们将使用简单的练习来应用本书中涵盖的材料，以激发你对R处理字符字符串的潜力的兴趣。

第2章

R中的字符字符串

R给我们处理文本字符串的主要数据类型是字符。 形式上，R中保存字符字符串的对象的类别是"character"。 我们使用单引号或双引号来表示字符字符串：

```
'使用单引号的字符字符串'
"使用双引号的字符字符串"
```

我们可以在字符串中插入单引号和双引号：

```
"用于统计计算的'R'项目"
'用于统计计算的"R"项目'
```

我们不能在单引号字符串中插入单引号，也不能在双引号字符串中插入双引号（不要这样做！）：

```
"这是完全不可接受的"
'这是绝对错误的'
```

2.1 创建字符字符串

除了单引号 ''或双引号 ""，R还提供了函数character()来创建字符字符串。 更具体地说，character()是创建类型为"character"的向量对象的函数。

2.1.1 空字符串

让我们从最基本的字符串开始：由连续引号产生的空字符串： ""。 从技术上讲， ""是一个没有字符的字符串，因此被称为空字符串：

```
# 空字符串
empty_str = ""

# 显示
empty_str

## [1] ""

# 类型
class(empty_str)

## [1] "character"
```

2.1.2 空字符向量

另一个基本的字符串结构是由函数character()和其参数 length=0生成的空字符向量empty_chr:# 空字符向量empty_chr = charac

```
ter(0)


# 显示
empty_chr

## character(0)

# 类型
class(empty_chr)

## [1] "character"
```

重要的是不要混淆空字符向量character(0)和空字符串 ""；它们之间的主要区别之一是它们的长度不同:# 空字符串的长度length(empty_str)## [1] 1

```
# 空字符向量的长度
length(empty_chr)

## [1] 0
```

注意空字符串empty str的长度为1，而空字符向量
empty_chr的长度为0。

2.1.3 字符()

正如我们已经提到的，字符()是一个用于创建字符向量的函数。 我们只需要指定向量的长度，字符()将生成一个具有相同数量的空字符串的字符向量，例如：

```
# 由5个空字符串组成的字符向量
char_vector = 字符(5)

# 显示
char_vector

## [1] "" "" "" "" ""
```

一旦创建了一个空的字符对象，可以通过给它一个超出其先前范围的索引值来简单地添加新的组件。

```
# 另一个例子
example = 字符(0)
example

## 字符(0)

# 检查其长度
length(example)

## [1] 0


# 添加第一个元素
example[1] = "第一个"
example

## [1] "第一个"

# 再次检查其长度
length(example)
```

```
## [1] 1
```

我们可以添加更多元素，而无需遵循连续的索引范围：

```
example[4] = "第四个"
example
```

```
## [1] "第一个" NA          NA          "第四个"
```

```
length(example)
```

```
## [1] 4
```

请注意，我们从一个单元素向量变成了一个四元素向量，而没有指定第二个和第三个元素。R用缺失值 NA填补了这个空白。

2.1.4 is.character() 和 as.character()

与character()相关的是它的两个姐妹函数：as.character()和is.character()。
这两个函数是用于创建和测试类型为"character"的对象的通用方法。
要测试一个对象是否为"character"类型，可以使用函数is.character()：

```
# 定义两个对象 'a' 和 'b'
a = "测试我"
b = 8 + 9

# 'a' 和 'b' 是字符吗?
is.character(a)
```

```
## [1] TRUE
```

```
is.character(b)
```

```
## [1] FALSE
```

同样，你也可以使用函数 class()来获取对象的类别：

```
# 'a' 和 'b' 的类别
class(a)
```

```
## [1] "character"
```

```
class(b)
```

```
## [1] "numeric"
```

无论好坏，R允许你将非字符对象转换为字符字符串，使用函数as.character()：# 将'b'转换为字符b = as.character(b)b## [1] "17"

2.2 字符串和 R 对象

在继续讨论用于操作字符串的函数之前，我们需要谈论一些重要的技术细节。R有五种主要的数据存储对象类型：向量、因子、矩阵（和数组）、数据框和列表。 我们可以使用这些对象中的每一个来存储字符字符串。 然而，这些对象的行为将取决于我们是否将字符数据与其他类型的数据一起存储。 让我们看看R如何处理具有不同数据类型（例如字符、数值、逻辑）的对象。

2.2.1 带有字符字符串的 R对象的行为

向量 R中的主要且最基本的对象类型是向量。 向量的值必须都是相同的模式。 这意味着任何给定的向量必须明确地是逻辑、数值、复数、字符或原始类型。那么当我们在向量中混合不同类型的数据时会发生什么？

```
# 含有数字和字符的向量
c(1:5, pi, "文本")
```

```
## [1] "1"                 "2"                 "3"
## [4] "4"                 "5"                 "3.14159265358979"
## [7] "文本"
```

正如你所看到的，从整数 (1:5)、数字 pi和一些"文本"组合而成的向量，其所有元素都被视为字符字符串。 换句话说，当我们在向量中组合混合数据时，字符串将占主导地位。这意味着向量的模式将是"字符"，即使我们混合了逻辑值:# 含有数字、逻辑值和字符的向量c(1:5, TRUE, pi, "文本", FALSE)## [1] "1"

```
                        "2"                 "3"
## [4] "4"                 "5"                 "TRUE"
```

```
## [7] "3.14159265358979" "文本"                "FALSE"
```

矩阵当我们在矩阵中混合字符和数字时，与向量相同的行为发生。 同样，所有内容都将被视为字符:

```
# 用数字和字符创建矩阵
rbind(1:5, letters[1:5])
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "1"  "2"  "3"  "4"  "5"
## [2,] "a"  "b"  "c"  "d"  "e"
```

数据框架对于数据框架，情况有些不同。 默认情况下，数据框架中的字符字符串将被转换为因子:

```
# 包含数字和字符的数据框架
df1 = data.frame(numbers = 1:5, letters = letters[1:5])
df1
```

```
##   numbers letters
## 1       1       a
## 2       2       b
## 3       3       c
## 4       4       d
## 5       5       e
```

```
# 检查数据框架结构
str(df1)
```

```
## 'data.frame': 5 obs. of  2 variables:
##  $ numbers: int  1 2 3 4 5
## $ letters: 因子，有5个水平"a","b","c","d",..: 1 2 3 4 5
```

要关闭data.frame()将字符串转换为因子的默认行为，请使用参数stringsAsFactors = FALSE： # 包含数字和字符的数据框df2 = data.frame(numbers = 1:5, letters = letters[1

```
:5],stringsAsFactors = FALSE)df2##
```

```
     numbers letters
## 1       1       a
## 2       2       b
```

```
## 3       3       c
## 4       4       d
## 5       5       e
```

```
# 检查数据框结构
str(df2)
```

```
## 'data.frame': 5 obs. of    2 variables:
##  $ numbers: int   1 2 3 4 5
##  $ letters: chr   "a" "b" "c" "d" ...
```

尽管df1和df2的显示相同，但它们的结构不同。 虽然df1$letters存储为"factor"，df2$letters存储为"character"。

列表 使用列表，我们可以组合任何类型的数据。 列表中每个元素的数据类型将保持其对应的模式：

```
# 具有不同模式的列表
列表(1:5, letters[1:5], rnorm(5))
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] "a" "b" "c" "d" "e"
##
## [[3]]
## [1] -0.6958 -2.2614 -1.4495    1.5161   1.3733
```

2.3 将文本导入 R

我们已经看到如何使用单引号 ''或双引号 ""来表示字符字符串。 但是，我们还需要讨论如何将文本导入 R，即如何导入和读取包含字符字符串的文件。 那么，我们如何将文本导入 R？ 嗯，这基本上取决于我们想要读取的文件的类型格式。

我们将描述两种一般情况。 一种是文件内容可以以表格格式表示（即行和列）。 另一种是内容没有表格结构。 在这种情况下，我们有一些以非结构化形式存在的字符（即仅为字符串行）或至少以非表格格式存在的字符，例如html、xml或其他标记语言格式。

另一个函数是 scan()它允许我们以多种格式读取数据。 通常我们使用 scan()来解析 R脚本，但我们也可以用它来导入文本（字符）

2.3.1 读取表格

如果我们要导入的数据是以某种表格格式的，我们可以使用一组函数来读取表格，如read.table()及其相关函数，例如 read.csv(),read.delim()
,
read.fwf()。 这些函数以表格格式读取文件，并从中创建一个数据框，其中行对应于案例，列对应于文件中的字段。

以表格格式读取文件的函数

函数	描述
read.table()	以表格格式读取文件的主要函数
read.csv()	读取以逗号 ","分隔的csv文件
read.csv2()	读取以分号 ";"分隔的csv文件
read.delim()	读取以制表符 "\t"分隔的文件
read.delim2()	类似于 read.delim()
read.fwf()	读取固定宽度格式文件

让我们看一个简单的例子，从澳大利亚广播公司的网站 *ABC*（http://www.abc.net.au/radio/）中读取一个文件。 特别是，我们将读取一个包含来自ABC广播电台的数据的 csv 文件。这个文件位于：
http://www.abc.net.au/local/data/public/stations/abc-local-radio.csv

要导入文件abc-local-radio.csv，我们可以使用read.table()或 read.csv()（只需选择正确的参数）。 以下是使用read.table()读取文件的代码：# abc广播电台数据的URL

```
abc = "http://www.abc.net.au/local/data/public/stations/abc-local-radio.csv"

# 从URL读取数据
radio = read.table(abc, header = TRUE, sep = ",", stringsAsFactors = FALSE)
```

在这种情况下，文件的位置在对象 abc中定义，它是传递给read.table()的第一个参数。然后我们选择其他参数，如header = TRUE，sep ="，"和stringsAsFactors = FALSE。 参数header = TRUE表示文件的第一行包含列的名称。 分隔符（逗号）由sep = ","指定。 最后，为了将文件中的字符字符串保持为数据框中的"character"，我们使用stringsAsFactors = FALSE。

如果在文件读取操作期间一切顺利，下一步要做的是使用 dim()检查创建的数据框的大小：

```
# 'radio'表的大小
dim(radio)
```

```
## [1] 53 18
```

请注意，数据框 radio是一个具有53行和18列的表。 如果我们使用 str()检查结构，我们将获得每列的信息。 参数vec.len = 1表示我们只想显示每个变量的第一个元素：

```
# 列的结构
str(radio, vec.len = 1)
```

```
## 'data.frame': 53 obs. of    18 variables:
##  $ State            : chr  "QLD" ...
##  $ Website.URL      : chr  "http://www.abc.net.au/brisbane/" ...
##  $ Station          : chr  "612 ABC Brisbane" ...
##  $ Town             : chr  " Brisbane " ...
##  $ Latitude         : num  -27.5 ...
## $ 经度              : 数字 153 ...
## $ 回复号码 : 字符串 "1300 222 612" ...
## $ 询问号码: 字符串 "07 3377 5222" ...
## $ 传真号码          : 字符串 "07 3377 5612" ...
## $ 短信号码          : 字符串 "0467 922 612" ...
## $ 街道号码          : 字符串 "114 Grey Street" ...
## $ 街道区域          : 字符串 "South Brisbane" ...
## $ 街道邮编 : 整数 4101 4700 ...
## $ 邮政信箱          : 字符串 "GPO Box 9994" ...
## $ 邮政区域          : 字符串 "Brisbane" ...
## $ 邮政编码          : 整数 4001 4700 ...
## $ 推特              : 字符串 " 612brisbane" ...
## $ 脸书              : 字符串 " http://www.facebook.com/612ABCBrisbane" ...
```

正如你所看到的，大多数的18个变量都是以"字符"模式。 只有 $纬度, $经度 , $街道邮编 和 $邮政编码 有不同的模式。

2.3.2 读取原始文本

如果我们想要导入文本原样导入（即我们想要读取原始文本），那么我们需要使用readLines()。 如果我们不想让 R假设数据是任何特定的形式，那么我们应该使用这个函数。 我们使用readLines()的方式是通过传递文件名或URL的名称来读取。 输出是一个字符向量，每行文件或URL都有一个元素，包含每行的内容。

让我们看看如何读取文本文件。 对于这个例子，我们将使用来自网站*TEXTFILES.COM*（由Jason *Scott*）的文本文件。http://www.textfiles.com/music/。 这个网站包含了一部分与音乐相关的文本文件。 为了演示目的，让我们考虑“1991年105.3首热门歌曲”根据“现代摇滚”电台*KITS San Francisco*。相应的 txt文件位于：http://www.textfiles.com/music/ktop100.txt。要使用函数readLines()读取文件：# 读取 'ktop100.txt' 文件

```
top105 = readLines("http://www.textfiles.com/music/ktop100.txt")
```

readLines()创建一个字符向量，其中每个元素表示我们尝试读取的URL的行。 要知道top105中有多少元素（即有多少行），我们可以使用函数length()。 要检查第一个元素（即文本文件的第一行），请使用head()# 有多少行length(top105)## [1] 123

```
# 检查第一个元素
head(top105)
```

```
## [1] "From: ed@wente.llnl.gov (Ed Suranyi)"
## [2] "Date: 12 Jan 92 21:23:55 GMT"
## [3] "Newsgroups: rec.music.misc"
## [4] "Subject: KITS' year end countdown"
## [5] ""
## [6] ""
```

通过使用head()提供的输出，前四行包含有关电子邮件主题（KITS年终倒计时）的一些信息。 第五和第六行是空行。 如果我们检查接下来的几行，我们会发现top100中歌曲的列表从第11行开始。

```
# 前5首歌曲
top105[11:15]
```

```
## [1] "1. NIRVANA                          SMELLS LIKE TEEN SPIRIT"
## [2] "2. EMF                              UNBELIEVABLE"
## [3] "3. R.E.M.                           LOSING MY RELIGION"
## [4] "4. SIOUXSIE & THE BANSHEES          KISS THEM FOR ME"
## [5] "5. B.A.D. II                        RUSH"
```

每行都有排名数字，后跟一个点，后跟一个空格，然后是艺术家/组的名称，后跟一堆空格，然后是歌曲的标题。

正如您所看到的，1991年的冠军是*Nirvana*的“Smells like teen spirit”。

KITS排名中的最后几首歌曲呢？ 为了得到答案，我们可以使用tail()函数来检查文件的最后n = 10个元素：# 检查最后10个元素tail(top105, n = 10)

```
##   [1] "101. SMASHING PUMPKINS           SIVA"
##   [2] "102. ELVIS COSTELLO              OTHER SIDE OF ..."
##   [3] "103. SEERS                       PSYCHE OUT"
##   [4] "104. THRILL KILL CULT            SEX ON WHEELZ"
##   [5] "105. MATTHEW SWEET               I'VE BEEN WAITING"
##   [6] "105.3   LATOUR                   PEOPLE ARE STILL HAVING SEX"
##   [7] ""
##   [8] "Ed"
##   [9] "ed@wente.llnl.gov"
## [10] ""
```

请注意，最后四行不包含有关歌曲的信息。 此外，歌曲的数量不止停在105首。 实际上，排名一直到106首歌曲（最后一个数字是105.3）。

我们将在这里结束对本章的讨论。 然而，重要的是要记住，文本文件以各种形式、大小和风格存在。 有关如何在R中导入文件的更多信息，请参阅R数据导入/导出指南（由R核心团队提供），网址为：

http://cran.r-project.org/doc/manuals/r-release/R-data.html

第三章

字符串操作

在前一章中，我们讨论了R如何处理带有字符的对象，以及如何导入文本数据。 现在我们将介绍一些基本（和不那么基本）的用于操作字符字符串的函数。

3.1 多功能的 paste() 函数

函数 paste()可能是我们可以用来创建和构建字符串的最重要的函数之一。paste()接受一个或多个R对象，将它们转换为"character"，然后将它们连接（粘贴）起来形成一个或多个字符字符串。 它的用法如下：

```
paste(..., sep = " ", collapse = NULL)
```

参数 ... 表示它可以接受任意数量的对象。. 参数 sep是一个用作分隔符的字符字符串。参数 collapse是一个可选的字符串，用于指示是否将所有项合并为一个字符串。 这是一个使用 paste()的简单示例：# paste

```
PI = paste("The life of", pi)

PI

## [1] "The life of 3.14159265358979"
```

正如你所看到的，默认分隔符是一个空格（sep = " "）。 但是你可以选择另一个字符，例如sep = "-":

```
# paste
IloveR = paste("I", "love", "R", sep = "-")

IloveR

## [1] "I-love-R"
```

如果我们给 paste()不同长度的对象，那么它将应用循环使用规则。 例如，如果我们将单个字符 "X"与序列 1:5粘贴在一起，并使用分隔符sep = "."，我们会得到以下结果:

```
# 使用不同长度的对象进行粘贴
paste("X", 1:5, sep = ".")

## [1] "X.1" "X.2" "X.3" "X.4" "X.5"
```

为了看到 collapse参数的效果，让我们比较有和没有它的区别：

```
# 使用collapse拼接
paste(1:3, c("!", "?", "+"), sep = "", collapse = "")

## [1] "1!2?3+"

# 不使用collapse拼接
paste(1:3, c("!", "?", "+"), sep = "")

## [1] "1!" "2?" "3+"
```

paste()的一个潜在问题是它将缺失值NA强制转换为字符"NA"： # 使用缺失值NA

```
evalue = paste("'e'的值是", exp(1), NA)

evalue

## [1] "'e'的值是2.71828182845905 NA"
```

除了paste()之外，还有一个函数paste0()，它相当于paste(..., sep = "", collapse)

： # 使用paste0进行拼接

```
paste0("让我们", "合并", "所有", "这些", "单词")

## [1] "让我们合并所有这些单词"
```

3.2 打印字符

R提供了一系列用于打印字符串的函数。 在创建程序化对象类的打印方法时，一些打印函数很有用，特别是在创建 print方法时。 其他函数在 R控制台或给定的文件中打印输出时很有用。 在本节中，我们将描述以下与打印相关的函数：

打印函数

函数	描述
print()	通用打印
noquote()	无引号打印
cat()	连接
format()	特殊格式
toString()	转换为字符串
sprintf()	打印

3.2.1 使用 print()打印值

在 R中，工作主要是通过 print()函数来进行打印。 正如其名称所示，此函数在 R控制台上打印其参数：

```
# 文本字符串
my_string = "使用数据进行编程很有趣"

# 打印字符串
print(my_string)

## [1] "使用数据进行编程很有趣"
```

更准确地说，print()是一个通用函数，这意味着在创建编程类的打印方法时应该使用这个函数。

从前面的例子中可以看出，默认情况下，print()以引号形式显示文本。
如果我们想要打印没有引号的字符字符串，可以设置参数quote = FALSE

```
# 不带引号打印
print(my_string, quote = FALSE)

## [1] 使用数据进行编程很有趣
```

print()产生的输出可以使用各种可选参数进行自定义。 然而，print()显示对象的方式相当有限。 为了获得更多的打印变化，我们可以使用一些更灵活的函数。

3.2.2 使用noquote()打印无引号字符

我们知道可以使用print()和其参数quote = FALSE来打印没有引号的文本。 实现类似输出的另一种选择是使用noquote()。 正如其名称所示，这个函数打印没有引号的字符字符串:

```
# noquote
noquote(my_string)

## [1] 用数据编程很有趣
```

更准确地说，noquote()创建了一个类为"noquote"的字符对象，它总是在不带引号的情况下显示：

```
# 类别 noquote
no_quotes = noquote(c("一些", "引用", "文本", "!%^(&="))

# 显示
no_quotes

## [1] 一些         引用文本     !%^(&=

# 检查类别
class(no_quotes)

## [1] "noquote"

# 测试字符
is.character(no_quotes)

## [1] TRUE

# 即使在下标中也没有引号
no_quotes[2:3]

## [1] 引用文本
```

3.2.3 连接并使用 cat()

另一个非常有用的函数是 cat()它允许我们连接对象并将它们打印到屏幕上或文件中。 它的使用具有以下结构：

cat(..., file = "", sep = " ", fill = FALSE, labels = NULL, append = FALSE)参数 ...

意味着 cat()接受多种类型的数据。 然而，当我们传递数值和/或复数元素时，它们会被R自动转换为字符字符串

猫(). 默认情况下，字符串以空格字符作为分隔符连接。 这可以通过 sep参数进行修改。

如果我们只使用一个字符串来使用 cat()，你会得到一个类似的（尽管不完全相同）结果如 noquote()：

```
# 只是用'cat()'打印
cat(my_string)

## 用数据编程很有趣
```

正如你所看到的， cat()会打印它的参数而不带引号。 本质上， cat()只是显示其内容（在屏幕上或在文件中）。 与 noquote()相比， cat()不会打印数字行指示器（在这种情况下为[1]）。

当我们有两个或更多要连接的字符串时， cat()的有用之处在于：

```
# 连接并打印
cat(my_string, "with R")

## 用数据编程很有趣 with R
```

您可以使用参数 sep来指示一个包含在连接元素之间的字符向量：

```
# 指定'sep'
cat(my_string, "使用R", sep = " =) ")

## 用R进行数据编程很有趣 =) 使用R

# 另一个例子
cat(1:10, sep = "-")

## 1-2-3-4-5-6-7-8-9-10
```

当我们将向量传递给 cat()时，每个元素都被视为单独的参数:

```
# 前四个月
cat(month.name[1:4], sep = " ")

## 一月 二月 三月 四月
```

参数 fill允许我们分割长字符串；这是通过指定字符串宽度为整数实现的:

```
# fill = 30
cat("长字符串", "可以以漂亮的格式显示","通过使用'fill'参数", fill = 30)oo
    ooooooooo## 长字符串## 可以以漂亮的格式显示## 通过使用'fill'参数
```

```
## 长字符串ooooooooooo## 可
以以漂亮的格式显示## 通过使
用'fill'参数
```

最后但并非最不重要的是，我们可以在 cat()中指定一个文件输出。 例如，假设我们想要将输出保存在我们的工作目录中的文件 output.txt中： # 在给定文件中使用catcat(

```
my_string, "使用R", file = "output.txt")
```

3.2.4 使用 format()对字符串进行编码

函数 format()允许我们对 R对象进行格式化以进行漂亮的打印。 基本上，format()使用常见的格式将向量的元素视为字符字符串。 在以不同格式打印数字和数量时，这非常有用。

```
# 默认用法
format(13.7)

## [1] "13.7"

# 另一个例子
format(13.12345678)

## [1] "13.12"
```

一些有用的参数：

- width生成的字符串的（最小）宽度
- trim如果设置为 TRUE，则不会用空格填充
- justify 控制字符串的填充方式。 取值为 "left", "right", "centre", "none"

要控制数字的打印，使用以下参数：

- digits小数点右侧的数字位数。
- scientific使用 TRUE表示科学计数法， FALSE表示标准计数法。

请记住， justify不适用于数值。

```
# 使用'nsmall'
format(13.7, nsmall = 3)

## [1] "13.700"
```

```
# 使用'digits'
format(c(6, 13.1), digits = 2)

## [1] " 6" "13"

# 使用'digits'和'nsmall'
format(c(6, 13.1), digits = 2, nsmall = 1)

## [1] " 6.0" "13.1"
```

默认情况下， `format()`用空格填充字符串，使它们的长度相同。

```
# 对齐选项
format(c("A", "BB", "CCC"), width = 5, justify = "centre")

## [1] "  A  " " BB  " " CCC "

format(c("A", "BB", "CCC"), width = 5, justify = "left")

## [1] "A    " "BB   " "CCC  "

format(c("A", "BB", "CCC"), width = 5, justify = "right")

## [1] "    A" "   BB" "  CCC"

format(c("A", "BB", "CCC"), width = 5, justify = "none")

## [1] "A"   "BB"  "CCC"


# 数字
format(1/1:5, digits = 2)

## [1] "1.00" "0.50" "0.33" "0.25" "0.20"


# 使用'digits'、widths和justify
format(format(1/1:5, digits = 2), width = 6, justify = "c")

## [1] " 1.00 " " 0.50 " " 0.33 " " 0.25 " " 0.20 "
```

为了以序列化格式打印大量数据，我们可以使用参数 `big.mark`或`big.interval`。 例如，下面是如何打印以逗号分隔的序列号的示例 `","`# `big.mark`

```
format(123456789, big.mark = ",")

## [1] "123,456,789"
```

3.2.5 使用C风格的字符串格式化 sprintf()

函数 sprintf()是一个包装器，用于返回一个格式化的字符串，结合了文本和变量值。 关于 sprintf()的一个好处是，它为我们提供了一种非常灵活的方式来将向量元素格式化为字符字符串。 它的用法如下所示：

```
sprintf(fmt, ...)
```

参数 fmt是一个格式字符串的字符向量。 允许的转换规范以符号 %开头，后面跟着数字和字母。 为了演示目的，这里有几种格式化数字 pi的方式：# '%f' 表示'固定点'十进制表示法sprintf("%f", pi)## [1] "3.141593"

```
# 用3位小数的十进制表示
sprintf("%.3f", pi)

## [1] "3.142"

# 用1位整数和0位小数表示
sprintf("%1.0f", pi)

## [1] "3"

# 用3位小数的十进制表示
sprintf("%5.1f", pi)

## [1] "  3.1"

sprintf("%05.1f", pi)

## [1] "003.1"

# 带正号打印
sprintf("%+f", pi)

## [1] "+3.141593"

# 前缀一个空格
sprintf("% f", pi)

## [1] " 3.141593"

# 左对齐
sprintf("%-10f", pi) # 左对齐
```

```
## [1] "3.141593   "
```

```
# 用指数形式表示 'e'
sprintf("%e", pi)
```

```
## [1] "3.141593e+00"
```

```
# 用指数形式表示 'E'
sprintf("%E", pi)
```

```
## [1] "3.141593E+00"
```

```
# 有效数字的数量（默认为6）
sprintf("%g", pi)
```

```
## [1] "3.14159"
```

3.2.6 使用 toString()将对象转换为字符串

函数 toString()允许我们将一个 R对象转换为字符字符串。 这个函数可以作为 format()的辅助函数，从一个向量中的多个对象中生成一个单个字符字符串。 结果将是一个长度为1的字符向量，元素之间用逗号分隔：

```
# 默认用法
toString(17.04)
```

```
## [1] "17.04"
```

```
# 结合两个对象
toString(c(17.04, 1978))
```

```
## [1] "17.04, 1978"
```

```
# 结合多个对象
toString(c("Bonjour", 123, TRUE, NA, log(exp(1))))
```

```
## [1] "Bonjour, 123, TRUE, NA, 1"
```

关于 toString()的一个好处是，你可以指定它的参数 width来固定
最大字段宽度。

```
# 使用'width'
toString(c("一个", "两个", "3333333333"), width = 8)
```

```
## [1] "一个,...."
```

```
# 使用'width'
```

```
toString(c("一个", "两个", "3333333333"), width = 12)
```

```
## [1] "一个, 两个...."
```

3.2.7 比较打印方法

尽管R语言只有一小部分用于打印和格式化字符串的函数，但我们可以使用它们来获得各种各样的输出。 函数的选择（及其参数）将取决于我们想要打印什么，以及我们想要如何打印和在哪里打印它。有时候选择使用哪个函数是直接的。 然而，有时候我们需要尝试和比较不同的方法，直到找到最合适的方法。 为了完成本节，让我们考虑一个具有5个元素的数值向量的简单示例：

```
# 打印方法
print(1:5)
```

```
## [1] 1 2 3 4 5
```

```
# 转换为字符
as.character(1:5)
```

```
## [1] "1" "2" "3" "4" "5"
```

```
# 连接
cat(1:5, sep = "-")
```

```
## 1-2-3-4-5
```

```
# 默认拼接
paste(1:5)
```

```
## [1] "1" "2" "3" "4" "5"
```

```
# 使用collapse参数连接字符串
paste(1:5, collapse = "")
```

```
## [1] "12345"
```

```
# 转换为单个字符串
toString(1:5)
```

```
## [1] "1, 2, 3, 4, 5"
```

```
# 不带引号的输出
noquote(as.character(1:5))
```

```
## [1] 1 2 3 4 5
```

3.3 基本字符串操作

除了创建和打印字符串外，在R中还有一些非常方便的函数来对字符串进行基本操作。 在本节中，我们将回顾以下函数：

字符串操作

函数	描述
nchar()	字符数
tolower()	转换为小写
toupper()	转换为大写
casefold()	大小写折叠
chartr()	字符转换
abbreviate()	缩写
substring()	字符向量的子字符串
substr()	字符向量的子字符串

3.3.1 使用 nchar()计算字符数

用于操作字符字符串的主要函数之一是 nchar()它计算字符串的字符数。 换句话说， nchar()提供了字符串的“长度”：

```
# 有多少个字符?
nchar(c("多少","个","字符? "))
```

```
## [1]  3  4 11
```

```
# 有多少个字符?
nchar("多少个字符? ")
```

```
## [1] 20
```

请注意，第二个例子中单词之间的空格也被计算为字符。

重要的是不要混淆nchar()和length()。 前者给出字符的数量，而后者只给出向量中元素的数量。

```
# 有多少个元素?
length(c("多少","个","字符? "))
```

```
## [1] 3
```

```
# 有多少个元素?
length("多少个字符? ")
```

```
## [1] 1
```

3.3.2 使用tolower()转换为小写

R语言提供了三个用于文本大小写转换的函数。 我们将讨论的第一个函数是tolower()，它将任何大写字符转换为小写：

```
# 转换为小写
tolower(c("所有字符都是小写","ABCDE"))
```

```
## [1] "all characters in lower case" "abcde"
```

3.3.3 使用 toupper()将字符串转换为大写

与 tolower()相反的函数是 toupper. 正如你可能猜到的那样，这个函数将任何小写字符转换为大写:

```
# 转换为大写
toupper(c("所有字符都是大写的","abcde"))
```

```
## [1] "所有字符都是大写的" "ABCDE"
```

3.3.4 使用 casefold()进行大小写转换

第三个大小写转换函数是 casefold()，它是 tolower()和 toupper()的包装器。 它的用法如下:

```
casefold(x, upper = FALSE)
```

默认情况下， casefold()将所有字符转换为小写，但我们可以使用参数 upper = TRUE来表示相反（字符为大写）：

```
# 小写转换
casefold("所有字符都是小写的")
```

```
## [1] "所有字符都是小写的"
```

```
# 大写转换
casefold("所有大写字母", upper = TRUE)
```

```
## [1] "所有大写字母"
```

3.3.5 使用 chartr() 进行字符转换

还有一个函数 chartr()代表字符转换。 chartr()接受三个参数：一个旧字符串，一个新字符串和一个字符向量 x：

```
chartr(old, new, x)
```

chartr()的工作原理是将出现在 x中的 old中的字符替换为 new中指定的字符。 例如，假设我们想要将句子 x中的小写字母 'a'替换为大写字母'A'： # 将 'a' 替换为 'A'

```
chartr("a", "A", "这是一个无聊的字符串")
```

```
## [1] "这是一个无聊的字符串"
```

重要的是要注意， old和 new必须具有相同数量的字符，否则你将会得到一个错误消息，如下所示：

```
# 错误的使用
chartr("ai", "X", "这是一个糟糕的例子")
```

```
## 错误：'old'比'new'更长
```

这是一个更有趣的例子，其中old = "aei"和new = "#!?"。 这意味着任何'a'在'x'中都将被替换为'#'，任何'e'在'x'中都将被替换为'？'，任何'i'在'x'中都将被替换为'？'： # 多个替换

```
crazy = c("为那些疯狂的人干杯", "不合群的人", "叛逆者")
chartr("aei", "#!?", crazy)
```

```
## [1] "H!r!'s to th! cr#zy on!s" "Th! m?sf?ts"
## [3] "Th! r!b!ls"
```

3.3.6 使用abbreviate() 缩写字符串

另一个用于基本字符串操作的有用函数是abbreviate()。 它的使用方法如下：

```
abbreviate(names.org, minlength = 4, dot = FALSE, strict = FALSE,
           method = c("left.keep", "both.sides"))
```

虽然有几个参数，但主要参数是字符向量（names.org），其中包含我们想要缩写的名称：

```
# 一些颜色名称
some_colors = colors()[1:4]
some_colors
```

```
## [1] "white"          "aliceblue"     "antiquewhite"  "antiquewhite1"
```

```
# 缩写（默认用法）
colors1 = abbreviate(some_colors)
colors1
```

```
##         white     aliceblue  antiquewhite antiquewhite1
##        "whit"         "alcb"         "antq"         "ant1"
```

```
# 使用'minlength'进行缩写
colors2 = abbreviate(some_colors, minlength = 5)
colors2
```

```
##         white     aliceblue  antiquewhite antiquewhite1
##       "white"       "alcbl"        "antqw"        "antq1"
```

```
# 缩写
colors3 = abbreviate(some_colors, minlength = 3, method = "both.sides")
colors3
```

```
##         white     aliceblue  antiquewhite antiquewhite1
##         "wht"          "alc"          "ant"          "an1"
```

3.3.7 用 substr()替换子字符串

在处理字符串时，一种常见的操作是提取和替换
一些字符。 对于这样的任务，我们有函数 substr()可以提取或替换

字符向量中的子字符串。 它的用法如下:

```
substr(x, start, stop)
```

x是一个字符向量， start表示要替换的第一个元素， stop表示要替换的最后一个元素:

```
# 提取 'bcd'
substr("abcdef", 2, 4)
```

```
## [1] "bcd"
```

```
# 用井号替换第二个字母
x = c("may", "the", "force", "be", "with", "you")
substr(x, 2, 2) <- "#"
x
```

```
## [1] "m#y"    "t#e"    "f#rce" "b#"     "w#th"   "y#u"
```

```
# 用开心的表情替换第二个和第三个字母
y = c("may", "the", "force", "be", "with", "you")
substr(y, 2, 3) <- ":)"
y
```

```
## [1] "m:)"    "t:)"    "f:)ce" "b:"     "w:)h"   "y:)"
```

```
# 使用循环替换
z = c("may", "the", "force", "be", "with", "you")
substr(z, 2, 3) <- c("#", "@")
z
```

```
## [1] "m#y"    "t@e"    "f#rce" "b@"     "w#th"   "y@u"
```

3.3.8 使用substring() 函数替换子字符串

与 substr()函数密切相关的substring()函数可以从字符向量中提取或替换子字符串。 它的用法如下:

```
substring(text, first, last = 1000000L)
```

text是一个字符向量， first表示要替换的第一个元素， last表示要替换的最后一个元素:

```
# 与'substr'相同
substring("ABCDEF", 2, 4)

## [1] "BCD"

substr("ABCDEF", 2, 4)

## [1] "BCD"


# 提取每个字母
substring("ABCDEF", 1:6, 1:6)

## [1] "A" "B" "C" "D" "E" "F"


# 多次替换并循环使用
text = c("more", "emotions", "are", "better", "than", "less")
substring(text, 1:3) <- c(" ", "zzz")
text

## [1] " ore"      "ezzzions" "ar "      "zzzter"   "t an"     "lezz"
```

3.4 集合操作

R有专门的函数用于对给定向量执行集合操作。 这意味着我们可以对"字符"向量应用集合并、交、差、相等和成员关系等函数。

集合操作

函数	描述
并集()	集合并集
交集()	集合交集
差集()	集合差异
集合相等()	相等的集合
完全相同()	完全相等
is.element()	是元素
%in%()	包含
排序()	排序
paste(rep())	重复

3.4.1 使用 union() 进行集合并集

让我们从union() 开始复习集合函数。 正如其名称所示，当我们想要获取两个字符向量之间的并集元素时，我们可以使用union()。

```
# 两个字符向量
set1 = c("一些", "随机", "词语", "一些")
set2 = c("一些", "许多", "没有", "少量")

# set1和set2的并集
union(set1, set2)
```

```
## [1] "一些"     "随机" "词语" "许多"          "没有"     "一些"
```

注意 union()会丢弃提供的向量中的任何重复值。 在先前的例子中，单词 "一些"在set1中出现了两次，但在union中只出现了一次。
实际上，所有的集合操作函数都会丢弃任何重复的值。

3.4.2 使用intersect()进行集合交集

使用函数intersect()执行集合交集。 当我们希望获取两个向量中共同的元素时，可以使用这个函数：

```
# 两个字符向量
set3 = c("一些", "随机", "一些", "词语")
set4 = c("一些", "许多", "没有", "一些")

# set3和set4的交集
intersect(set3, set4)
```

```
## [1] "一些" "一些"
```

3.4.3 使用 setdiff()进行集合差异

与交集相关，我们可能对获取两个字符向量之间的差异感兴趣。 可以使用 setdiff()来实现：# 两个字符向量

```
set5 = c("一些", "随机", "少数", "词语")
set6 = c("一些", "许多", "没有", "少数")

# set5和set6之间的差异
```

```
setdiff(set5, set6)
```

```
## [1] "随机" "词语"
```

3.4.4 使用 setequal()进行集合相等性测试

函数 setequal()允许我们测试两个字符向量的相等性。 如果向量包含相同的元素， setequal()返回 TRUE (FALSE otherwise)# 三个字符向量set7 = c("一些", "随机", "字符串")set8 = c("一些", "许多", "没有", "少数")set9 = c("字符串", "随机", "一些")

```
# set7 == set8?
setequal(set7, set8)
```

```
## [1] FALSE
```

```
# set7 == set9?
setequal(set7, set9)
```

```
## [1] TRUE
```

3.4.5 使用identical()进行精确相等性测试

有时候 setequal()并不总是我们想要使用的。可能的情况是，我们想要测试两个向量是否逐个元素完全相等。 例如，测试是否set7完全等于 set9。 尽管这两个向量包含相同的元素集合，但它们并不完全相同。 可以使用函数identical()进行这样的测试

```
# set7和set7完全相等吗?
identical(set7, set7)
```

```
## [1] TRUE
```

```
# set7和set9完全相等吗?
identical(set7, set9)
```

```
## [1] FALSE
```

如果我们查阅identical()的帮助文档，我们可以看到这个函数是“测试两个对象是否完全相等的安全可靠的方法”。

3.4.6 使用is.element()检查元素是否包含在内

如果我们想要测试一个元素是否包含在给定的字符字符串集合中，我们可以使用is.element()：# 三个向量

```
set10 = c("一些", "东西", "玩", "的")
elem1 = "玩"
elem2 = "疯狂"

# set10中的elem1?
is.element(elem1, set10)

## [1] TRUE


# set10中的elem2?
is.element(elem2, set10)

## [1] FALSE
```

或者，我们可以使用二元运算符 %in%来测试一个元素是否包含在给定的集合中。函数 %in %如果第一个操作数包含在第二个操作数中，则返回 TRUE，否则返回 FALSE：# set10中的elem1?

```
elem1 %in% set10

## [1] TRUE


# set10中的elem2?
elem2 %in% set10

## [1] FALSE
```

3.4.7 使用 sort()进行排序

函数 sort()允许我们对向量的元素进行排序，可以按升序（默认）或按降序使用参数 decreasing进行排序：

```
set11 = c("今天", "生产", "例子", "漂亮", "一个", "好")

# 排序（降序）
sort(set11)
## [1] "一个"   "漂亮" "例子"   "好"   "生产" "今天"

# 排序（升序）
sort(set11, decreasing = TRUE)
## [1] "今天"   "生产" "好"   "例子"   "漂亮" "一个"
```

如果我们有字母数字字符串， `sort()`在升序排序时会将数字放在前面：

```
set12 = c("今天", "生产", "例子", "漂亮", "1", "好")

# 排序（降序）
sort(set12)
## [1] "1"   "漂亮" "例子"   "好"   "生产" "今天"

# 排序（升序）
sort(set12, decreasing = TRUE)
## [1] "今天"   "生产" "好"   "例子"   "漂亮" "1"
```

3.4.8 使用 `rep()` 进行重复

字符串的一个非常常见的操作是重复，也就是给定一个字符串，我们想要将其重复多次。虽然在 R中没有单独的函数来实现这个目的，但我们可以结合使用 `paste()`和 `rep()`来实现：`# 将 'x' 重复 4 次paste(rep("x", 4), collapse = "")## [1] "xxxx"`

```

```

第4章

使用 stringr进行字符串操作

正如我们在之前的章节中所看到的, R提供了一系列用于基本字符串处理和操作"字符"数据的有用函数。 大多数情况下，这些函数足够使用，并且它们将帮助我们完成工作。 然而，它们也有一些缺点。 例如，考虑以下示例：

```
# 一些文本向量
text = c("一个", "两个", "三个", NA, "五个")

# 每个字符串中有多少个字符?
nchar(text)

## [1] 3 3 5 2 4
```

正如你所看到的，nchar() 给出了 NA 的值为2，就好像它是由两个字符组成的字符串一样。
也许在某些情况下这是可以接受的，但考虑到R语言中的所有操作，最好将NA保留原样，而不是将其视为由两个字符组成的字符串。
在R语言中，将NA保留原样会更好，而不是将其视为由两个字符组成的字符串。

另一个尴尬的例子可以在 paste()中找到。 默认分隔符是一个空格，这通常是我们想要使用的。但这是次要的。 真正令人讨厌的是当我们想要粘贴包含零长度参数的东西时。 在这些情况下，paste()的行为如何? 请参见下面的示例：# 这个工作正常

```
paste("大学", "加利福尼亚", "伯克利")

## [1] "大学加利福尼亚伯克利"

# 这也可以正常工作
paste("大学", "加利福尼亚", "伯克利")
```

```
## [1] "大学加利福尼亚伯克利"


# 这很奇怪
paste("大学", "加利福尼亚", "伯克利", NULL)
## [1] "大学加利福尼亚伯克利"


# 这很丑陋
paste("大学", "加利福尼亚", "伯克利", NULL, character(0),
      "加油熊！")
## [1] "大学加利福尼亚伯克利                    加油熊！ "
```

注意最后一个例子的输出（丑陋的那个）。 对象NULL和character(0)的长度为零，但在paste()中包含时，它们被视为空字符串""。
如果paste()删除长度为零的参数，那将是很好的，不是吗？ 不幸的是，我们无法改变nchar()和paste()。但不要担心。有一个非常好的包可以解决这些问题，并提供几个函数来进行一致的字符串处理。

4.1 包 stringr

感谢Hadley Wickham，我们有了包 stringr，它为处理字符串在 R中添加了更多功能。
根据包的描述（参见http://cran.r-project.org/web/packages/stringr/index.html）
stringr

> “这是一组简单的包装器，使R的字符串函数更一致、更简单、更易于使用。它通过确保：函数和参数名称（和位置）一致，所有函数正确处理NA和零长度字符，每个函数的输出数据结构与其他函数的输入数据结构匹配。”

要安装 stringr，请使用函数install.packages()。 安装后，使用 library()将其加载到当前会话中：# 安装 'stringr'install.packages("stringr")

```


# 加载 'stringr'
library(stringr)
```

4.2 基本字符串操作

stringr提供了基本操作和正则表达式操作的函数。在本章中，我们介绍与基本操作相关的函数。
接下来，在第6章中讨论了使用正则表达式函数和 stringr进行的操作。

下表包含了基本字符串操作的 stringr函数：

函数	描述	类似于
str_c()	字符串连接	paste()
str_length()	字符数	nchar()
str_sub()	提取子字符串	substring()
str_dup()	复制字符	无
str_trim()	去除前导和尾随空格无	
str_pad()	填充字符串	无
str_wrap()	换行字符串段落	strwrap()
str_trim()	修剪字符串	无

正如你所看到的， stringr中的所有函数都以"str "开头，后面跟着与它们执行的任务相关的术语。 例如，str length()给出了字符串中的字符数（即长度）。 此外，一些函数旨在提供更好的替代方案，以取代已有的函数。 这就是str length()的情况，它旨在替代 nchar()。 然而，其他函数没有相应的替代方案，例如str dup()，它允许我们复制字符。

4.2.1 使用str c()进行连接

让我们从str c()开始。 这个函数与 paste()等效，但是不使用空格作为默认分隔符，而是使用空字符串 ""。

```
# 默认用法
str_c("五月", "原力", "与你同在")
```

```
## [1] "五月原力与你同在"
```

```
# 移除长度为零的对象
str_c("五月", "原力", NULL, "与你同在", character(0))
```

```
## [1] "五月原力与你同在"
```

注意str c()和 paste()之间的另一个主要区别：像NULL和character(0)这样的长度为零的参数会被str c()静默地移除。

如果我们想要更改默认分隔符，可以像往常一样通过指定参数sep来实现：

```
# 更改分隔符
str_c("五月", "原力", "与你同在", sep = "_")

## [1] "五月_原力_与你同在"


# 同义函数 'str_join'
str_join("五月", "原力", "与你同在", sep = "-")

## [1] "五月-原力-与你同在"
```

从前面的例子中可以看出，str c()的同义词是str join()。 _ _

4.2.2 使用str length()计算字符数 _

正如我们之前提到的，函数str length()等同于nchar()。 这两个函数都返回字符串的字符数，即字符串的长度（不要与向量的长度length()混淆）。 与nchar()相比，str length()在处理NA值时具有更一致的行为。 与给出NA长度为2不同，str length()保留缺失值NA。# 一些文本（包括NA）

-

```
some_text = c("one", "two", "three", NA, "five")

# 将'str_length'与'nchar'进行比较
nchar(some_text)

## [1] 3 3 5 2 4

str_length(some_text)

## [1]  3  3  5 NA  4
```

此外，str_length()还具有一个很好的特性，它可以将因子转换为字符，而nchar()则无法处理：# 一些因子

```
some_factor = factor(c(1, 1, 1, 2, 2, 2), labels = c("good", "bad"))
some_factor

## [1] good good good bad  bad  bad
## Levels: good bad

# 在因子上尝试'nchar'
```

```
nchar(some_factor)

## Error:  'nchar()' requires a character vector

# 现在与'str_length'进行比较
str_length(some_factor)

## [1] 4 4 4 3 3 3
```

4.2.3 使用str sub() 进行子字符串处理

要从字符向量中提取子字符串，stringr 提供了 str sub()，它等同于 substring()。 函数 str sub() 的使用形式如下：`str_sub(string, start = 1L, end = -1L)`

函数中的三个参数分别是：一个字符串向量，一个表示子字符串第一个字符位置的起始值，以及一个表示子字符串最后一个字符位置的结束值。 下面是一个简单的示例，从一个字符串中提取从第1个到第5个字符：

```
# 一些文本
lorem = "Lorem Ipsum"

# 应用 'str_sub'
str_sub(lorem, start = 1, end = 5)

## [1] "Lorem"

# 等同于 'substring'
substring(lorem, first = 1, last = 5)

## [1] "Lorem"


# 另一个示例
str_sub("adios", 1:3)

## [1] "adios" "dios"  "ios"
```

str sub()的一个有趣特性是它能够在开始和结束位置上使用负索引。 当我们使用负位置时，str sub()会从最后一个字符开始倒数：

```
# 一些字符串
resto = c("brasserie", "bistrot", "creperie", "bouchon")
```

```
# 使用负位置的'str_sub'
str_sub(resto, start = -4, end = -1)

## [1] "erie" "trot" "erie" "chon"

# 与substring相比（无用）
substring(resto, first = -4, last = -1)

## [1] "" "" "" ""
```

与substring类似，我们还可以给str sub()一组将在字符串上循环使用的位置。 但更好的是，我们可以给str sub()一个负序列，substring会忽略这个：# 顺序提取str_sub(lorem, seq_len(nchar(lorem)))## [1] "Lorem Ipsum" "orem Ipsum" "rem Ipsum"

```
                                                "em Ipsum"     "m Ipsum"
##  [6] " Ipsum"       "Ipsum"        "psum"         "sum"          "um"
## [11] "m"

substring(lorem, seq_len(nchar(lorem)))

##  [1] "Lorem Ipsum" "orem Ipsum"    "rem Ipsum"    "em Ipsum"     "m Ipsum"
##  [6] " Ipsum"       "Ipsum"        "psum"         "sum"          "um"
## [11] "m"

# reverse substrings with negative positions
str_sub(lorem, -seq_len(nchar(lorem)))

##  [1] "m"            "um"           "sum"          "psum"         "Ipsum"
##  [6] " Ipsum"       "m Ipsum"      "em Ipsum"     "rem Ipsum"    "orem Ipsum"
## [11] "Lorem Ipsum"

substring(lorem, -seq_len(nchar(lorem)))

##  [1] "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum"
##  [6] "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum"
## [11] "Lorem Ipsum"
```

我们不仅可以使用str_sub()提取子字符串，还可以用它来替换子字符串：

```
# 将'Lorem'替换为'Nullam'
lorem = "Lorem Ipsum"
str_sub(lorem, 1, 5) <- "Nullam"
lorem

## [1] "Nullam Ipsum"
```

```
# 用负位置替换
lorem = "Lorem Ipsum"
str_sub(lorem, -1) <- "Nullam"
lorem

## [1] "Lorem IpsuNullam"


# 多次替换
lorem = "Lorem Ipsum"
str_sub(lorem, c(1, 7), c(5, 8)) <- c("Nullam", "Enim")
lorem

## [1] "Nullam Ipsum"    "Lorem Enimsum"
```

4.2.4 使用str dup() 进行复制 _

处理字符时常见的操作是复制. 问题是 R没有
专门的函数来实现这个目的. 但是 stringr有 : str dup()可以在字符向量中复制和
连接字符串. 它的使用需要两个参数:

```
str_dup(string, times)
```

第一个输入是要重复的字符串. 第二个输入, times, 是每个字符串要复制的次数:

```
# 默认用法
str_dup("hola", 3)

## [1] "holaholahola"


# 使用不同的'times'
str_dup("adios", 1:3)

## [1] "adios"            "adiosadios"       "adiosadiosadios"


# 使用字符串向量
words = c("lorem", "ipsum", "dolor", "sit", "amet")
str_dup(words, 2)

## [1] "loremlorem" "ipsumipsum" "dolordolor" "sitsit"       "ametamet"

str_dup(words, 1:5)
```

```
## [1] "lorem"                 "ipsumipsum"            "dolordolordolor"
## [4] "sitsitsitsit"          "ametametametametamet"
```

4.2.5 使用str pad()进行填充

在 stringr中，我们还可以找到另一个方便的函数str pad()用于填充字符串。 它的默认用法如下：

```
 str_pad(string, width, side = "left", pad = " ")
```

str_pad()的想法是将字符串填充到指定的总宽度，并在前导或尾随字符上进行填充。 默认的填充字符是空格（pad = " "），因此返回的字符串将显示为左对齐（side = "left"），右对齐（side = "right"）或两者都是（side = "both"）。让我们看一些例子：

```
# 默认用法
str_pad("hola", width = 7)

## [1] "    hola"


# 两边填充
str_pad("adios", width = 7, side = "both")

## [1] " adios "


# 左边用'#'填充
str_pad("hashtag", width = 8, pad = "#")

## [1] "#hashtag"


# 两边用'-'填充
str_pad("hashtag", width = 9, side = "both", pad = "-")

## [1] "-hashtag-"
```

4.2.6 使用str wrap() 进行换行

函数str wrap()等同于 strwrap()，可用于将字符串换行为
格式化段落。 将（长）字符串换行的思想是首先将其分成段落

根据给定的宽度进行分段，然后在每行中添加指定的缩进（第一行使用缩进，后续行使用取消缩进）。 其默认用法如下所示：

```
str_wrap(string, width = 80, indent = 0, exdent = 0)
```

例如，考虑以下引用（来自道格拉斯·亚当斯）转换为段落：

```
# 引用（道格拉斯·亚当斯）
some_quote = c(
  "我可能没有去过",
  "我打算去的地方，",
  "但我认为我已经到达了",
  "我需要去的地方")

# 将some_quote放在一个段落中
some_quote = paste(some_quote, collapse = " ")
```

现在，假设我们想要在预定的列宽（例如30）内显示某个引用的文本。 我们可以通过应用str_wrap()函数并设置参数width = 30来实现这一点。

```
# 使用width = 30显示段落
cat(str_wrap(some_quote, width = 30))

## 我可能没有去过我
## 打算去的地方，但我认为我
## 已经到达了我需要去的地方
##
```

除了将（长）段落显示为多行外，我们还可以希望添加一些缩进。 以下是我们如何缩进第一行以及后续行的方法：

```
# 使用首行缩进为2显示段落
cat(str_wrap(some_quote, width = 30, indent = 2), "    \n")

##   我可能没有去过我
## 打算去的地方，但我认为我
## 已经到达了我需要去的地方
##


# 以3个空格缩进显示段落
cat(str_wrap(some_quote, width = 30, exdent = 3), "    \n")

## 我可能没有去过我
##    打算去的地方，但我
```

```
##     认为我已经结束了
##     我需要去的地方
```

4.2.7 使用str trim()修剪

字符串处理的典型任务之一是将文本解析为单词。
通常，我们得到的单词两端都有空格，称为空白字符。 在这种情况下，我们可以使用str trim()函数来删除字符串两端的任意数量的空格。 它的使用只需要两个参数：

```
str_trim(string, side = "both")
```

第一个输入是要修剪的字符串，第二个输入表示要删除的空白字符的位置。

考虑以下字符串向量，其中一些字符串左侧、右侧或两侧都有空格。 在不同的 side设置下，str trim()会对它们进行如下处理# 带有空格的文本

```
bad_text = c("这个", " 例子 ", "有几个                ", "    空格 ")

# 去除左边的空格
str_trim(bad_text, side = "left")
## [1] "这个"              "例子 "             "有几个          " "空格 "
# 去除右边的空格
str_trim(bad_text, side = "right")
## [1] "这个"              " 例子"             "有几个"           "    空格"
# 去除两边的空格
str_trim(bad_text, side = "both")
## [1] "这个"          "例子"                      "有几个" "空格"
```

4.2.8 使用 word()函数提取单词

我们在本章结束时描述 word()函数，该函数旨在从句子中提取单词：

```
word(string, start = 1L, end = start, sep = fixed(" "))
```

我们使用 word()的方式是将一个 string和一个要提取的第一个单词的 start位置以及一个要提取的最后一个单词的 end位置传递给它。 默认情况下，单词之间使用一个空格作为分隔符。

让我们看一些例子：

```
# 一些句子
change = c("成为改变", "你想成为的")

# 提取第一个单词
word(change, 1)

## [1] "成为" "你"

# 提取第二个单词
word(change, 2)

## [1] "the"   "want"

# 提取最后一个单词
word(change, -1)

## [1] "改变" "be"

# 提取除第一个单词之外的所有单词
word(change, 2, -1)

## [1] "the change" "want to be"
```

stringr有更多的函数，但我们将在第6章讨论它们，因为它们与正则表达式有关。

第5章

正则表达式（第一部分）

到目前为止，我们已经看到了一些处理和处理文本的基本和中级函数。 这些是非常有用的函数，它们使我们能够做很多有趣的事情。
然而，如果我们真正想释放字符串处理的力量，我们需要将事情提升到下一个级别，并讨论正则表达式。

正则表达式（也称为 **regex**）是一种用于描述一定数量的文本的特殊文本字符串。 这个“一定数量的文本”被正式称为模式。 因此，我们说正则表达式是描述一组字符串的模式。

几乎所有脚本语言（如Perl、Python、Java、Ruby等）都提供了处理正则表达式的工具。 R提供了一些处理正则表达式的函数，尽管它没有其他脚本语言提供的广泛功能。然而，通过一些变通方法（和一点耐心），它们可以帮助我们解决很多问题。

我假设你已经了解了正则表达式，所以我们不会涵盖关于正则表达式的所有知识。 相反，我们将重点介绍 R如何处理正则表达式，以及在正则表达式操作中需要使用的 R语法。

要了解更多关于正则表达式的一般信息，您可以在以下资源中找到一些有用的信息：

- 正则表达式维基百科
  http://Zh.wikipedia.org/wiki/正则表达式
  对于那些对正则表达式没有经验的读者，一个好的起点是查看维基百科入口。

- Regular-Expressions.info网站（由Jan Goyvaerts创建）
  http://www.regular-expressions.info
  一个充满有关正则表达式信息的优秀网站。 它涵盖了许多不同的主题，资源，大量的示例和教程，适用于初学者和高级水平。

高级水平。

- 掌握正则表达式（由Jeffrey Friedl编写）
  http://regex.info/book.html
  我不确定是否应该包括这个参考资料，但我认为它值得考虑。 这可能是关于正则表达式的权威书籍。 唯一的问题是它是一本更适合有正则表达式经验的读者的书。

5.1 正则表达式基础

使用正则表达式的主要目的是描述用于匹配文本字符串的模式。 简单来说，使用正则表达式就是模式匹配。 匹配的结果要么成功，要么失败。

最简单的模式匹配版本是在字符串中搜索一个或多个特定字符的出现。 例如，我们可能想要在一个大型文本文档中搜索单词"编程"，或者我们可能想要在包含 R脚本的一系列文件中搜索所有出现的字符串"应用"。

通常，正则表达式模式由字母数字字符和特殊字符的组合组成。 正则表达式模式可以是一个简单的字符，也可以由多个具有更复杂结构的字符组成。 在所有情况下，我们构建正则表达式的方式与构建算术表达式的方式相同，通过使用各种运算符来组合较小的表达式。

在各种类型的运算符中，它们的主要用途可以归结为创建正则表达式的四个基本操作：

- 连接
- 逻辑或
- 复制
- 分组

上述操作可以被视为正则表达式运算符的构建块。 通过以多种方式组合它们，我们可以表示非常复杂和复杂的模式。 以下是每个操作的简要描述：

连接 正则表达式的基本类型是通过将一组字符连接在一起形成的，就像 "abcd"一样。 这个正则表达式模式只匹配单个字符串“abcd”。

逻辑或 逻辑运算符 OR，用竖线 |表示，允许我们从多个可能性中选择。 例如，正则表达式 "ab|cd"恰好匹配两个字符串“ab”和“cd”。 使用逻辑运算符，我们可以用一个正则表达式指定多个字符串。 例如，如果我们正在尝试在一堆文档中查找与水文学相关的文本，我们可以搜索类似“水”、“海洋”、“海洋”、“河流”和“湖泊”等词语。 所有这些术语可以组合成一个正则表达式，如"water|ocean|sea|river|lake"。

重复 另一个基本操作是重复，它使我们能够定义一个在多个可能性下匹配的模式。 更具体地说，这个操作使用一系列的正则表达式运算符，称为量词，它们重复前面的正则表达式指定的次数。

分组分组运算符用括号()表示，使我们能够将任意数量的其他运算符和字符作为一个单元来指定。 换句话说，分组序列是一个被括号括起来的表达式，被视为一个单元。 例如，如果我们想要指定字符串集合X、XYX、XYXYX、XYXYXYX等，我们必须写成"(XY)*X"，以表示"XY"模式必须一起重复。

5.2 在R中的正则表达式

有两个主要方面需要考虑关于在R中的正则表达式。 一个与用于正则表达式模式匹配的函数有关。 另一个方面与在R中表达正则表达式模式有关。 在本节中，我们将讨论后一个问题：R如何处理正则表达式。 我发现在讨论函数和如何与正则表达式模式交互之前，先了解R在正则表达式操作方面的特殊性更方便。

5.2.1 R中的正则表达式语法细节

大多数在其他脚本语言中使用的正则表达式操作在R中以相同的方式工作。但并非所有都是如此。 一些正则表达式元素和模式在R中需要特定的语法，我们需要进行描述。

要了解有关在R中使用的正则表达式的规范和技术细节，您应该查看帮助页面：help(regex)或help(regexp)（或者 ?regex）。

```
# 要了解有关R中正则表达式的更多信息，请查看帮
助(regex)
```

关于正则表达式的帮助文档包含了大量的技术信息。
根据您对正则表达式的专业知识水平，您可能会发现提供的信息非常有用，或者只是晦涩难懂。 但不要期望找到任何示例。 帮助内容仅供参考，没有关于R的实际演示。 关于本书，我们将在下一小节中涵盖以下主题：

- 元字符
- 量词
- 序列
- 字符类
- POSIX字符类

5.2.2 元字符

最简单的正则表达式形式是匹配单个字符的表达式。 大多数
字符，包括所有字母和数字，都是正则表达式，可以与它们自身匹配。
例如，模式 "1"可以匹配数字1。 模式 "="可以匹配等号符号。 模式 "blu"可以匹配字母“blu”组成的集合。 然而，有一些特殊字符具有保留状态，它们被称为元字符。 在扩展的正则表达式中，元字符包括：

```
. \ | ( ) [ { $ * + ?
```

例如，模式 "money$"不匹配“money$”。 同样，模式 "what?"不匹配“what?”。 除了少数情况外，元字符在使用正则表达式时具有特殊的含义和目的。

通常情况下，在正则表达式模式中，如果要表示元字符的字面含义，我们需要使用反斜杠 \进行转义。 然而，在 R中，我们需要使用双反斜杠 \进行转义。 下表显示了常见的正则表达式元字符以及在 R中如何进行转义：

元字符和如何在R中转义它们

元字符的字面意义		在R中转义
.	句点或点	\\.
$	美元符号	\\$
*	星号或星号	*
+	加号	\\+
?	问号	\\?
|	竖线或管道符号	\\\|
\	反斜杠	\\\\
^	插入符号	\\^
[	开方括号	\\[
]	闭方括号	\\]
{	开花括号	\\{
}	闭花括号	\\}
(	开圆括号	\\(
)	闭圆括号	\\)

例如，考虑字符字符串 "$money"。 假设我们想要用空字符串 ""替换美元符号 $。 这可以通过函数 sub()来实现。 天真的（但是错误的）方法是简单地尝试用字符 "$"匹配美元符号：# 字符串money = "$money"

```
# 天真但错误的方法
sub(pattern = "$", replacement = "", x = money)

## [1] "$money"
```

如你所见，什么都没有发生。 在大多数脚本语言中，当我们想要在正则表达式中表示任何元字符的字面意义时，我们需要用反斜杠进行转义。 R的问题在于传统的方法不起作用（你会得到一个糟糕的错误）：

```
# 在R中的常规（在其他语言中）但错误的方式
sub(pattern = " \$", replacement = "", x = money)
```

在R中，我们需要用双反斜杠转义元字符。 这是我们如何有效地替换 $符号的方法：# 在R中的正确方式

```
sub(pattern = " \ $", replacement = "", x = money)

## [1] "money"
```

以下是一些愚蠢的例子，展示了如何在R中转义元字符以便替换为空字符串：# 美元符号

```
sub("\ $", "", "$和平-爱")

## [1] "和平-爱"

# 点号
sub("\ .", "", "和平.爱")

## [1] "和平爱"

# 加号
sub("\ +", "", "和平+爱")

## [1] "和平爱"

# 插入符号
sub("\ ^", "", "和平^爱")

## [1] "和平爱"

# 竖线
sub("\ |", "", "和平|爱")

## [1] "和平爱"

# 左圆括号
sub("\ (", "", "和平(爱)")

## [1] "和平爱)"

# 右圆括号
sub("\ )", "", "和平(爱)")

## [1] "和平(爱"

# 左方括号
sub("\ [", "", "和平[爱]")

## [1] "和平爱]"

# 右方括号
sub("\ ]", "", "和平[爱]")

## [1] "和平[爱"

# 左花括号
sub("\{ ", "", "和平 {爱 }")
```

```
## [1] "PeaceLove}"

# closing curly bracket
sub("\} ", "", "Peace {Love}")

## [1] "Peace{Love"

# double backslash
sub("\\  ", "", "Peace \ Love")

## [1] "PeaceLove"
```

5.2.3 序列

序列定义，毫不奇怪，可以匹配的字符序列。 我们在R中有简写版本（或锚点）常用的序列：

R中的锚点序列

锚点	描述
\d	匹配数字字符
\D	匹配非数字字符
\s	匹配空格字符
\S	匹配非空格字符
\w	匹配单词字符
\W	匹配非单词字符
\b	匹配单词边界
\B	匹配非（单词边界）
\h	匹配水平空格
\H	匹配非水平空格
\v	匹配垂直空格
\V	匹配非垂直空格

让我们看一个应用锚定序列的例子，围绕替换操作。 可以使用函数 sub()和 gsub()进行替换。 虽然我们将在本章后面讨论替换函数，但重要的是要记住它们的区别。sub()替换第一个匹配项，而 gsub()替换所有匹配项。

让我们考虑字符串"the dandelion war 2010"，并检查用下划线" "替换每个锚定序列的不同效果。

数字和非数字

```
# 用'_'替换数字
sub("\ d", "_", "the dandelion war 2010")

## [1] "the dandelion war _010"

gsub("\ d", "_", "the dandelion war 2010")

## [1] "the dandelion war ____"


# 用'_'替换非数字
sub("\ D", "_", "the dandelion war 2010")

## [1] "_he dandelion war 2010"

gsub("\ D", "_", "the dandelion war 2010")

## [1] "__________________2010"
```

空格和非空格

```
# 用'_'替换空格
sub("\ s", "_", "the dandelion war 2010")

## [1] "the_dandelion war 2010"

gsub("\ s", "_", "the dandelion war 2010")

## [1] "the_dandelion_war_2010"


# 用'_'替换非空格
sub("\ S", "_", "the dandelion war 2010")

## [1] "_he dandelion war 2010"

gsub("\ S", "_", "the dandelion war 2010")

## [1] "___ _________ ___ ____"
```

单词和非单词

```
# 用'_'替换单词
sub("\ b", "_", "the dandelion war 2010")

## [1] "_the dandelion war 2010"

gsub("\ b", "_", "the dandelion war 2010")

## [1] "_t_h_e_ _d_a_n_d_e_l_i_o_n_ _w_a_r_ _2_0_1_0_"
```

```
# 用'_'替换非单词
sub("\ B", "_", "the dandelion war 2010")

## [1] "t_he dandelion war 2010"

gsub("\ B", "_", "the dandelion war 2010")

## [1] "t_he d_an_de_li_on w_ar 2_01_0"
```

单词边界和非单词边界

```
# 用'_'替换单词边界
sub("\ w", "_", "the dandelion war 2010")

## [1] "_he dandelion war 2010"

gsub("\ w", "_", "the dandelion war 2010")

## [1] "___ _________ ___ ____"


# 用'_'替换非单词边界
sub("\ W", "_", "the dandelion war 2010")

## [1] "the_dandelion war 2010"

gsub("\ W", "_", "the dandelion war 2010")

## [1] "the_dandelion_war_2010"
```

5.2.4 字符类

字符类或字符集是由方括号[]括起来的字符列表。
字符集用于匹配多个字符中的一个。 例如，正则表达式字符类[aA]匹配任何小写字母a或任何大写字母A。 同样，正则表达式[0123456789]匹配任何单个数字。 重要的是不要混淆正则表达式字符类与本地R"字符"类概念。

字符类的一个特殊情况是当我们在列表的开头包含插入符号^时：这表示正则表达式匹配不在列表中的任何字符。 这就是为什么这种配置也被称为类否定或否定字符类的原因。
例如，正则表达式[^xyz]匹配除了字符x、y或z之外的任何字符。

此外，字符集可以被指定为字符范围，例如数字、小写ASCII字母和大写ASCII字母，通过给出第一个和最后一个字符，用连字符分隔。 例如，正则表达式[a-z]匹配任何小写ASCII字母。

字母。 依次，字符类 [0-9] 匹配任何数字。

一些（正则表达式）字符类

锚点	描述
[aeiou]	匹配任何一个小写元音字母
[AEIOU]	匹配任何一个大写元音字母
[0123456789]	匹配任何数字
[0-9]	匹配任何数字（与前一个类相同）
[a-z]	匹配任何小写ASCII字母
[A-Z]	匹配任何大写ASCII字母
[a-zA-Z0-9]	匹配上述类别中的任何一个
[^aeiou]	匹配除小写元音字母之外的任何内容
[^0-9]	匹配除数字之外的任何内容

让我们看一个基本示例。 假设我们有一个包含多个单词的字符向量，并且我们有兴趣匹配那些包含元音字母"e"或 "i"的单词。 为此，我们可以使用字符类"[ei]"： # 一些字符串

```
transport = c("car", "bike", "plane", "boat")

# 寻找 'e' 或 'i'
grep(pattern = "[ei]", transport, value = TRUE)
## [1] "自行车" "飞机"
```

这是另一个带有数字字符类的例子：

```
# 一些数字字符串
numerics = c("123", "17-April", "I-II-III", "R 3.0.1")

# 匹配包含0或1的字符串
grep(pattern = "[01]", numerics, value = TRUE)
## [1] "123"       "17-April" "R 3.0.1"
# 匹配任何数字
grep(pattern = "[0-9]", numerics, value = TRUE)
## [1] "123"       "17-April" "R 3.0.1"
# 非数字
grep(pattern = "[^0-9]", numerics, value = TRUE)
## [1] "17-April" "I-II-III" "R 3.0.1"
```

5.2.5 POSIX字符类

与正则表达式字符类密切相关的是*POSIX*字符类。 在 R中，*POSIX*字符类用双括号内的表达式表示[[]]。 下表显示了在 R中使用的*POSIX*字符类：

在 R中的POSIX字符类

类	描述
[[:lower:]]	小写字母
[[:upper:]]	大写字母
[[:alpha:]]	字母字符（[[:lower:]]和[[:upper:]]）
[[:digit:]]	数字：0、1、2、3、4、5、6、7、8、9
[[:alnum:]]	字母数字字符（[[:alpha:]]和[[:digit:]]）
[[:blank:]]	空白字符：空格和制表符
[[:cntrl:]]	控制字符
[[:punct:]]	标点符号字符：！”＃％＆’（）＊＋，-。／：；[[:space:]] 空格字符：制表符、换行符、垂直制表符、换页符、回车和空格
[[:xdigit:]]	十六进制数字：0-9 A B C D E F a b c d e f
[[:print:]]	可打印字符（[[:alpha:]]，[[:punct:]]和空格）
[[:graph:]]	图形字符（[[:alpha:]]和[[:punct:]]）

例如，假设我们正在处理以下字符串：

```
# la vie（字符串）
la_vie = "La vie en #FFC0CB（玫瑰）;\nCes't la vie!  \ttres jolie"

# 如果打印'la_vie'
print(la_vie)

## [1] "La vie en #FFC0CB（玫瑰）;Ces't la vie! 非常漂亮"


# 如果你猫 'la_vie'
cat(la_vie)

## La vie en #FFC0CB（玫瑰）;
## Ces't la vie! 非常漂亮
```

如果我们对字符串la vie应用一些POSIX字符类的替换，会发生什么la vie:

```
# 移除空格字符
gsub(pattern = "[[:blank:]]", replacement = "", la_vie)## [
1] "Lavieen#FFC0CB(玫瑰);Ces'tlavie!tresjolie"

# 移除数字
gsub(pattern = "[[:punct:]]", replacement = "", la_vie)## [1] "La vie en FFC0
非常漂亮"

# 移除数字
gsub(pattern = "[[:xdigit:]]", replacement = "", la_vie)## [1] "L vi n # (
trs joli"

"# 删除可打印字符
"gsub(pattern = "[[:print:]]", replacement = "", la_vie)## [1] "
"

"# 删除不可打印字符
gsub(pattern = "[^[:print:]]", replacement = "", la_vie)
## [1] "La vie en #FFC0CB (rose);Ces't la vie! tres jolie"

"# 删除图形字符
gsub(pattern = "[[:graph:]]", replacement = "", la_vie)
## [1] "      "       " "

"# 删除非图形字符
gsub(pattern = "[^[:graph:]]", replacement = "", la_vie)
## [1] "Lavieen#FFC0CB(rose);Ces'tlavie!tresjolie"
```

5.2.6 量词

"另一个重要的正则表达式元素是所谓的量词。 "当我们想要匹配满足特定条件的一定数量的字符时，就会使用它们。

"量词指定字符、组或字符类的实例数量。

在输入中存在才能找到匹配项。 下表显示了正则表达式的量词：

在R中的量词

	量词描述
?	前面的项目是可选的，最多匹配一次
*	前面的项目将匹配零次或多次
+	前面的项目将匹配一次或多次
{n}	前面的项目恰好匹配n次
{n,}	前面的项目匹配n次或更多次
{n,m}	前面的项目至少匹配n次，但不超过m次

一些例子

```
# 人名
people = c("rori", "emilia", "matteo", "mehmet", "filipe", "anna", "tyler",
    "rasmus", "jacob", "youna", "flora", "adi")

# 最多匹配'm'一次
grep(pattern = "m?", people, value = TRUE)

##  [1] "rori"  "艾米莉娅" "马特奥" "梅赫梅特" "菲利普" "安娜"      "泰勒"
## [8] "拉斯穆斯" "雅各布" "尤娜" "弗洛拉" "阿迪"


# 匹配'm'恰好一次
grep(pattern = "m {1}", people, value = TRUE, perl = FALSE)

## [1] "艾米莉娅" "马特奥" "梅赫梅特" "拉斯穆斯"


# 匹配'm'零次或多次，并且匹配't'
grep(pattern = "m*t", people, value = TRUE)

## [1] "马特奥" "梅赫梅特" "泰勒"


# 匹配't'零次或多次，并且匹配'm'
grep(pattern = "t*m", people, value = TRUE)

## [1] "艾米莉娅" "马特奥" "梅赫梅特" "拉斯穆斯"


# 匹配'm'一次或多次
grep(pattern = "m+", people, value = TRUE)
```

```
## [1] "emilia" "matteo" "mehmet" "rasmus"

# 匹配'm'一次或多次，然后匹配't'
grep(pattern = "m+.t", people, value = TRUE)

## [1] "matteo" "mehmet"

# 匹配't'恰好两次
grep(pattern = "t {2}", people, value = TRUE)

## [1] "matteo"
```

5.3 正则表达式的函数

一旦我们描述了R如何处理最常见的正则表达式元素，
就该介绍我们可以用来处理正则表达式的函数了。

5.3.1 主要的正则表达式函数

R在基础包中包含一组函数，我们可以用它们来查找模式匹配。
下表列出了这些函数及其简要描述：

R中的正则表达式函数

函数	目的	特点
grep()	查找正则表达式匹配项	匹配的元素（索引或值）
grepl()	查找正则表达式匹配项	匹配的元素（TRUE和FALSE）
regexpr()	查找正则表达式匹配项	第一个匹配的位置
gregexpr()	查找正则表达式匹配	所有匹配的位置
regexec()	查找正则表达式匹配项	regexpr()和gregexpr()的混合
sub()	只替换正则表达式匹配的第一个匹配	
gsub()	替换正则表达式匹配的所有匹配	
strsplit()	根据匹配将向量拆分为子向量	

前面表格中列出的前五个函数用于在字符向量中查找模式匹配。 所有这些函数的目标都是相同的：查找匹配。 它们之间的区别在于输出的格式。 接下来的两个函数——sub()和gsub()——用于替换：查找匹配并进行替换。

它们。 最后一个函数，strsplit()，用于根据正则表达式匹配将字符向量的元素拆分为子字符串。

基本上，所有正则表达式函数都需要两个主要参数：一个 **pattern**（即正则表达式）和一个要匹配的 **text**。 每个函数还有其他附加参数，但主要的是一个pattern和一些文本。 特别是， **pattern**基本上是一个包含正则表达式的字符字符串，用于在给定的 **text**中进行匹配。

您可以通过输入 help(grep)（或者
也可以输入 ?grep）来查看所有grep()类似函数的文档。

```
# 主要正则表达式函数的帮助文档
help(grep)
```

5.3.2 stringr中的正则表达式函数

R包 stringr还提供了几个用于正则表达式操作的函数（见下表）。
更具体地说， stringr提供了用于检测、定位、提取、匹配、替换和拆分字符串的模式匹配函数。

在 stringr中的正则表达式函数

函数	描述
str_detect()	检测字符串中模式的存在或不存在
str_extract()	提取与模式匹配的字符串的第一个片段
str_extract_all()	提取与模式匹配的字符串的所有片段
str_match()	从字符串中提取第一个匹配的组
str_match_all()	从字符串中提取所有匹配的组
str_locate()	在字符串中定位第一个模式的位置
str_locate_all()	在字符串中定位所有模式的位置
str_replace()	替换字符串中第一个匹配模式的出现
str_replace_all()	替换字符串中所有匹配模式的出现
str_split()	将字符串分割成可变数量的片段
str_split_fixed()	将字符串分割成固定数量的片段

记住的一件重要事情是，stringr中的所有模式匹配函数都具有以下一般形式：

```
str_function(string, pattern)
```

它们的共同特点是它们都共享前两个参数：要处理的字符串向量和一个单一的模式（即正则表达式）来匹配。

5.3.3 互补匹配函数

除了主要的grep()函数外，R还有其他相关的匹配函数，如regmatches()，match()，pmatch()，charmatch()。 事实上，regmatches()是唯一设计用于与正则表达式模式配合使用的函数。 其他匹配函数不适用于正则表达式，但我们可以使用它们来匹配字符向量中的术语（例如单词、参数名称）。

补充匹配函数

函数	目的	特点
regmatches()	提取或替换与regexpr()、gregexpr()或regexec()匹配的数据使用	
match()	值匹配	查找（第一个）匹配的位置
pmatch()	部分字符串匹配	查找位置
charmatch()	类似于pmatch()	查找位置

5.3.4 接受正则表达式模式的辅助函数

同样， R包含其他与正则表达式模式交互或接受的函数： apropos(),ls(),browseEnv(), glob2rx(),help.search(),list.files()。 这些函数的主要目的是搜索 R对象或文件，但它们可以接受正则表达式模式作为输入。

辅助函数

函数	描述
apropos()	通过（部分）名称查找对象
browseEnv()	浏览环境中的对象
glob2rx()	将通配符或通配模式转换为正则表达式
help.search()	搜索帮助系统
list.files()	列出目录/文件夹中的文件

第6章

正则表达式（第二部分）

在前一章中，我们讨论了正则表达式的一般性；我们讨论了 R与正则表达式模式的特定方式；我们还快速介绍了可以使用的用于使用正则表达式操作字符串的函数。 在本章中，我们将更详细地描述正则表达式的函数，并展示一些示例来说明它们的用法。

6.1 模式查找函数

让我们从回顾第一个五个 grep()类似函数 grep(), grepl(), regexpr(), gregexpr(),和 regexec()开始。 所有这些函数的目标都是一样的：寻找匹配。
它们之间的区别在于输出的格式。 基本上，这些函数需要两个主要参数：一个模式（即正则表达式）和一个要匹配的文本。 这些函数的基本用法是：

```
grep(pattern, text)
grepl(pattern, text)
regexpr(pattern, text)
gregexpr(pattern, text)
regexec(pattern, text)
```

每个函数还有其他附加参数，但要记住的重要事情是一个模式和一些文本。

6.1.1 函数 grep()

grep()可能是最基本的函数之一，它允许我们在字符串向量中匹配模式。 grep()中的第一个参数是一个正则表达式，用于指定要匹配的模式。 第二个参数是一个包含要搜索的文本字符串的字符向量。
输出是文本向量中元素的索引，其中存在匹配。 如果找不到匹配项，则输出为空的整数向量。

```
# 一些文本
text = c("一个词", "一个句子", "你和我", "三二一")

# 模式
pat = "一个"

# 默认用法
grep(pat, text)

## [1] 1 4
```

从前面的示例输出中可以看出， grep()返回一个数字向量。
这表示第1个和第4个元素包含了匹配项。 相反，第2个和第3个元素没有。

我们可以使用参数 value来修改输出的方式。 如果我们选择value = TRUE，而不是返回索引， grep()返回字符串向量的内容：

```
# 使用'value'（显示匹配的文本）
grep(pat, text, value = TRUE)

## [1] "一个词"                 "三 二 一"
```

另一个有趣的参数是 invert。 我们可以使用这个参数来获取
不匹配的字符串，将其值设置为 TRUE

```
# 使用'invert'（显示不匹配的部分）
grep(pat, text, invert = TRUE)

## [1] 2 3

# 使用'values'也是一样的
grep(pat, text, invert = TRUE, value = TRUE)

## [1] "一个句子" "你和我"
```

总之， grep()可以用来对字符向量进行子集操作，只获取包含（或不包含）匹配模式的元素。

6.1.2 函数 grepl()

函数 grepl()使我们能够执行与 grep()类似的任务。 不同之处在于输出结果不是数值索引，而是逻辑值（TRUE / FALSE）。 因此，你可以将 grepl()看作是 grep-*logical*。 使用前面示例中的相同文本字符串，这是 grepl()的行为：# 一些文本

```
text = c("一个词", "一个句子", "你和我", "三二一")

# 模式
pat = "一个"

# 默认用法
grepl(pat, text)
```

```
## [1]  TRUE FALSE FALSE  TRUE
```

请注意，我们得到的逻辑向量与字符向量的长度相同。 那些与模式匹配的元素的值为 TRUE ; 那些与模式不匹配的元素的值为 FALSE。

6.1.3 函数 regexpr()

为了准确找到给定字符串中模式的位置，我们可以使用 regexpr()
函数。 该函数返回比 grep()更详细的信息，提供给我们：

a) 文本向量中实际包含正则表达式模式的元素，以及

b) 正则表达式模式匹配的子字符串的位置。

```
# 一些文本
text = c("一个词", "一个句子", "你和我", "三二一")

# 默认用法
regexpr("one", text)
```

```
## [1]  1 -1 -1 11
## attr(,"match.length")
## [1]  3 -1 -1  3
## attr(,"useBytes")
## [1] TRUE
```

乍一看，从 regexpr()的输出可能看起来有点凌乱，但解释起来非常简单。 在输出中，我们有三个显示的元素。 第一个元素是一个与文本长度相同的整数向量，给出了第一个匹配的起始位置。 在这个例子中，数字1表示模式"one"从文本的第一个元素的位置1开始。 负索引 -1表示没有匹配；数字11表示在文本的第四个元素中匹配的子字符串的位置。

属性"match.length"给出了每个文本元素中匹配的长度。
同样，负值 -1表示该元素中没有匹配。 最后，属性"useBytes"的值为 TRUE，这意味着匹配是按字节而不是按字符进行的。

6.1.4 函数 gregexpr()

函数 gregexpr()实际上与 regexpr()做的事情几乎相同：通过分别搜索每个元素，确定字符串向量中的模式位置。 唯一的区别是 gregexpr()以列表形式输出。 换句话说， gregexpr()返回一个与 text长度相同的列表，其中每个元素的形式与 regexpr()的返回值相同，只是给出了每个（不相交）匹配的起始位置。

```
# 一些文本
text = c("一个词", "一个句子", "你和我", "三二一")

# 模式
pat = "一个"

# 默认用法
gregexpr(pat, text)
```

```
## [[1]]
## [1] 1
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
```

```
##
## [[3]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"useBytes")
## [1] TRUE
##
## [[4]]
## [1] 11
## attr(,"match.length")
## [1] 3
## attr(,"useBytes")
## [1] TRUE
```

6.1.5 函数 regexec()

函数 regexec()与 gregexpr()非常接近，因为输出也是一个与 text长度相同的列表。 列表的每个元素包含匹配的起始位置。 值为 -1表示没有匹配。 此外，列表的每个元素都具有属性"match.length"，给出匹配的长度（或没有匹配时为-1）： # 一些文本

```
text = c("一个词", "一个句子", "你和我", "三二一")

# 模式
pat = "一个"

# 默认用法
regexec(pat, text)
```

```
## [[1]]
## [1] 1
## attr(,"match.length")
## [1] 3
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
##
```

```
## [[3]]
## [1] -1
## attr(,"match.length")
## [1] -1
##
## [[4]]
## [1] 11
## attr(,"match.length")
## [1] 3
```

来自Spector的例子

```
# 用于提取匹配项的便捷函数
x = regexpr(pat, text)
substring(text, x, x + attr(x, "match.length") - 1)
```

```
## [1] "one" ""      ""    "one"
```

```
# 包括NA
regexpr(pat, c(text, NA))
```

```
## [1]   1 -1 -1 11 NA
## attr(,"match.length")
## [1]   3 -1 -1   3 NA
```

6.2 模式替换函数

有时候我们只想在给定的字符串向量中找到一个模式。 然而，有时我们可能还对用另一个模式替换一个模式感兴趣。为此，我们可以使用替换函数sub()和gsub()。 sub()和gsub()之间的区别在于前者只替换第一次出现的模式，而后者替换所有出现的模式。

替换函数需要三个主要参数：要匹配的正则表达式模式，匹配模式的替换内容，以及要搜索匹配项的文本。基本用法如下：

```
sub(模式，替换内容，文本)
gsub(模式，替换内容，文本)
```

6.2.1 用sub()函数替换第一个出现的内容

函数 sub()用给定的文本中的模式替换第一个出现的模式。 这意味着如果字符串向量的每个元素中有多个模式的出现，只有第一个会被替换。 例如，假设我们有以下包含各种字符串的文本向量：

```
Rstring = c("R基金会",
            "用于统计计算",
            "R是免费软件",
            "R是一个合作项目")
```

假设我们的目标是用新的模式 "RR"替换模式 "R"。 如果我们使用 sub()函数，我们会得到以下结果：

```
# 字符串
Rstring = c("R基金会",
            "用于统计计算",
            "R是免费软件",
            "R是一个合作项目")

# 用'RR'替换'R'
sub("R", "RR", Rstring)
```

```
## [1] "R基金会"                              "用于统计计算"
## [3] "RR是免费软件"                         "RR是一个合作项目"
```

正如你所看到的，在文本向量的每个元素中，只有第一个出现的字母 R被替换。 请注意，第三个元素中的单词 FREE也包含一个 R，但它没有被替换。 这是因为它不是模式的第一个出现。

6.2.2 使用 gsub()替换所有出现

为了替换不仅仅是第一个模式出现，而是所有出现，我们应该使用gsub()（将其视为 *gene ral*substition）。 如果我们使用与上一个示例相同的向量 Rstring和模式，当我们应用 gsub()时，我们得到以下结果# string

```
Rstring = c("R基金会",
            "用于统计计算",
            "R是免费软件",
            "R是一个合作项目")
```

```
# substitute
gsub("R", "RR", Rstring)
```

```
## [1] "The RR Foundation"                "用于统计计算"
## [3] "RR是FRRREE软件"                   "RR是一个合作项目"
```

获得的输出与 sub()几乎相同，只是在Rstring的第三个元素中有所不同。 现在，单词 FR EE中的 R的出现也被考虑在内， gsub()将其更改为 FRRREE。

6.3 分割字符向量

除了查找模式和替换模式的操作之外，另一个常见的任务是基于模式分割字符串。 为了做到这一点，R提供了函数strsplit()，它可以根据正则表达式匹配将字符向量的元素拆分为子字符串。

如果你查看帮助文档 - help(strsplit) - 你会发现strsplit()的基本用法需要两个主要参数：

```
strsplit(x, split)
```

x是字符向量，split是正则表达式模式。 然而，为了保持与其他grep()函数使用的相同符号，最好将x视为文本，将split视为模式。 这样我们可以表示strsplit()的用法为：

```
strsplit(text, pattern)
```

我们可以使用strsplit()的一个典型任务是将字符串分解为单独的组件（即单词）。 例如，如果我们希望在给定的句子中分隔每个单词，我们可以指定一个空格" "作为分割模式：

```
sentence = c("R是一个有许多贡献者的合作项目")

# 拆分成单词
strsplit(sentence, " ")
```

```
## [[1]]
## [1] "R"          "是"         "一个"       "合作"
## [5] "项目"       "与"         "许多"       "贡献者"
```

另一个基本示例可能是通过拆分由连字符"-"
连接的数字集合来分解电话号码的部分 "-"

```
# 电话号码
tels = c("510-548-2238", "707-231-2440", "650-752-1300")

# 将每个号码拆分成各个部分
strsplit(tels, "-")
```

```
## [[1]]
## [1] "510"  "548"  "2238"
##
## [[2]]
## [1] "707"  "231"  "2440"
##
## [[3]]
## [1] "650"  "752"  "1300"
```

6.4 stringr中的函数

在前一章中，我们简要介绍了 R package stringr的函数，用于正则表达式。 正如我们提到的，所有的 stringr函数都共享一个常见的使用结构：

```
str_function(string, pattern)
```

主要的两个参数是：要处理的 string向量和一个单一的 pattern（即正则表达式）来匹配。 此外，所有的函数名都以前缀 str开头，后面跟着要执行的操作的名称。 例如，要 *lo cate*第一个出现的位置，我们应该使用str locate()；要定位所有匹配项的位置，我们应该使用str locate all()。

6.4.1 使用str detect()检测模式

要检测字符串向量中是否存在（或不存在）某个模式，我们可以使用函数str detect()。 实际上，这个函数是 grepl()的一个包装器：# 一些对象

```
some_objs = c("笔", "铅笔", "记号笔", "喷雾")

# 检测手机
```

```
str_detect(some_objs, "笔")
```

```
## [1]  TRUE  TRUE FALSE FALSE
```

```
# 选择检测到的匹配项
some_objs[str_detect(some_objs, "笔")]
```

```
## [1] "笔"      "铅笔"
```

正如你所看到的，str detect()的输出是一个布尔向量（TRUE/FALSE），长度与指定的字符串相同。 如果在字符串中检测到匹配项，则返回 TRUE，否则返回 FALSE。 下面是另一个更详细的示例，其中模式匹配日期的形式为日-月-年：

```
# 一些字符串
strings = c("12 Jun 2002", " 8 September 2004 ", "22-July-2009 ",
            "01 01 2001", "date", "02.06.2000",
            "xxx-yyy-zzzz", "$2,600")

# 日期模式（月份为文本）
dates = "([0-9] {1,2})[- .]([a-zA-Z]+)[- .]([0-9] {4})"

# 检测日期
str_detect(strings, dates)
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

6.4.2 使用str extract()提取第一个匹配项

要提取包含某个模式的字符串，我们可以使用函数str extract()。 实际上，这个函数提取了第一个与给定模式匹配的字符串片段。 例如，假设我们有一个包含一些关于巴黎的推文的字符向量，并且我们想要提取其中的标签。 我们可以通过定义一个 #标签模式来简单地实现这一点#[a-zA-Z]{1} # 关于'巴黎'的推文paris_tweets = c(

```


  "#巴黎充满了文化和美食景点",
  "在#巴黎的运河圣马丁著名的#阿梅丽",
  "当你在#巴黎的时候，在咖啡馆停下来：http://goo.gl/yaCbW",
  "巴黎，光之城")

#标签模式
```

```
hash = "#[a-zA-Z] {1,}"

#提取（第一个）标签
str_extract(paris_tweets, hash)

## [1] "#巴黎" "#巴黎" "#巴黎" NA
```

正如你所看到的，str_extract()的输出是与 string长度相同的向量。 那些不符合模式的元素被标记为 NA。 请注意，str_extract()只匹配第一个模式：它没有提取出标签 "#阿梅丽"。

6.4.3 使用str_extract_all()提取所有匹配项

除了str_extract()之外， stringr还提供了函数str_extract_all()。 正如其名称所示，我们使用str_extract_all()来提取向量字符串中的所有模式。
使用与前一个示例中相同的字符串，我们可以提取所有的标签匹配项
如下所示：

```
# 提取（所有）标签
str_extract_all(paris_tweets, "#[a-zA-Z] {1,}")

## [[1]]
## [1] "#巴黎"
##
## [[2]]
## [1] "#巴黎" "#阿梅丽"
##
## [[3]]
## [1] "#巴黎"
##
## [[4]]
## character(0)
```

与str_extract()相比，str_extract_all()的输出是一个与字符串长度相同的列表。 此外，不匹配模式的元素用一个空字符向量character(0)表示，而不是 NA。

6.4.4 使用str_match()提取第一个匹配组

与str_extract()密切相关的是包 stringr提供了另一个提取函数：
str_match()。 这个函数不仅提取匹配的模式，还显示每个匹配的位置。

在正则表达式字符类模式中匹配的组

```
# 字符串向量
strings = c("12 Jun 2002", " 8 September 2004 ", "22-July-2009 ",
            "01 01 2001", "date", "02.06.2000",
            "xxx-yyy-zzzz", "$2,600")

# 日期模式（月份为文本）
dates = "([0-9] {1,2})[- .]([a-zA-Z]+)[- .]([0-9] {4})"

# 提取第一个匹配的组
str_match(strings, dates)
```

```
##      [,1]                 [,2] [,3]          [,4]
## [1,] "12 Jun 2002"        "12" "Jun"         "2002"
## [2,] "8 September 2004" "8"     "September" "2004"
## [3,] "22-July-2009"       "22" "July"        "2009"
## [4,] NA                   NA   NA            NA
## [5,] NA                   NA   NA            NA
## [6,] NA                   NA   NA            NA
## [7,] NA                   NA   NA            NA
## [8,] NA                   NA   NA            NA
```

请注意，输出不是一个向量，而是一个字符矩阵。 第一列是完整的匹配，其他列是每个捕获的组。 对于那些不匹配的元素，有一个缺失值 NA。

6.4.5 使用str match all()提取所有匹配的组

如果我们要提取字符串向量中的所有模式，而不是使用 str_extract() 我们应该使用 str_extract_all()：

```
# 关于'巴黎'的推文
paris_tweets = c(
  "#巴黎充满了文化和美食景点",
  "在#巴黎的运河圣马丁著名的#阿梅丽",
  "当你在#巴黎的时候，在咖啡馆停下来：http://goo.gl/yaCbW",
  "巴黎，光之城")

# 在'paris_tweets'中匹配（所有）hashtags
str_match_all(paris_tweets, "#[a-zA-Z] {1,}")
```

```
## [[1]]
```

```
##      [,1]
## [1,] "#巴黎"
##
## [[2]]
##      [,1]
## [1,] "#巴黎"
## [2,] "#阿梅丽"
##
## [[3]]
##      [,1]
## [1,] "#巴黎"
##
## [[4]]
## character(0)
```

与str match()相比，str match all()的输出是一个 list。 还要注意，列表的每个元素都是一个具有与hashtag匹配数量相同的行数的矩阵。 反过来，那些不匹配模式的元素用一个空的字符向量character(0)表示，而不是一个 NA。

6.4.6 用str locate()定位第一个匹配项

除了检测、提取和匹配正则表达式模式外， stringr还允许我们定位模式的出现位置。 要定位字符串向量中模式的第一个出现位置，我们应该使用str locate()。

```
# 定位（第一个）井号的位置
str_locate(paris_tweets, "#[a-zA-Z]{1,}")
```

```
##      开始 结束
## [1,]    1    6
## [2,]   14   19
## [3,]   17   22
## [4,]   NA   NA
```

str locate()的输出是一个具有两列的矩阵，行数与（字符串）向量中的元素数量相同。 输出的第一列是起始位置，而第二列是结束位置。

在前面的示例中，结果是一个具有4行和2列的矩阵。 第一行对应于第一条推文的井号。 它从位置1开始，到位置6结束。 第二行对应于第二条推文的井号；它的起始位置是第14个字符，结束位置是第19个字符。

第四行对应于第四条推文。 由于没有井号，该行中的值为NA。 由于没有井号，该行中的值为NA。

6.4.7 使用str_locate_all()定位所有匹配项

为了定位字符串向量中的所有出现模式，而不仅仅是第一个模式，我们应该使用str_locate_all()：

```
# 在'paris_tweets'中定位（所有）hashtags
str_locate_all(paris_tweets, "#[a-zA-Z]{1,}")
```

```
## [[1]]
##      开始 结束
## [1,]     1    6
##
## [[2]]
##      开始 结束
## [1,]    14   19
## [2,]    54   60
##
## [[3]]
##      开始 结束
## [1,]    17   22
##
## [[4]]
##      开始 结束
```

与str_locate()相比，str_locate_all()的输出是一个与提供的字符串长度相同的列表。 列表的每个元素都是一个具有两列的矩阵。 那些不匹配模式的元素用空字符向量表示，而不是 NA。

从应用str_locate_all()到巴黎推文的结果中，你可以看到第二个元素包含了两个hashtags #Paris和 #Amelie的起始和结束位置。 反过来，第四个元素为空，因为它对应的推文中没有hashtags。

6.4.8 用str_replace()替换第一个匹配项

要在字符串中替换第一个匹配模式的出现，我们可以使用str_replace()。 它的用法如下：

```
str_replace(字符串, 模式, 替换)
```

除了其余函数的主要两个输入外，str replace()还需要一个第三个参数来指示替换模式。

假设我们有一个向量中的城市名称：旧金山、巴塞罗那、那不勒斯和巴黎。 假设我们想要用分号替换每个名称中的第一个元音字母。 下面是我们可以这样做的方法：

```
# 城市名称
cities = c("旧金山", "巴塞罗那", "那不勒斯", "巴黎")

# 替换第一个匹配的元音字母
str_replace(cities, "[aeiou]", ";")

## [1] "旧;金山" "巴;塞罗那"                "那;不勒斯"       "巴;黎"
```

现在，假设我们想要替换每个名字中的第一个辅音字母。 我们只需要修改模式并使用否定类：# 替换第一个匹配的辅音字母str_replace(cities, "[^aeiou]", ";")## [1] ";

```
an Francisco" ";arcelona"

                                        ";那不勒斯"       ";巴黎"
```

6.4.9 用str replace all()替换所有匹配项

要在字符串中替换所有匹配模式的出现，我们可以使用str replace all()。
再次考虑一个包含一些城市名称的向量，假设我们想要替换每个名称中的所有元音字母：

```
# 城市名称
cities = c("旧金山", "巴塞罗那", "那不勒斯", "巴黎")

# 替换所有匹配的元音字母
str_replace_all(cities, pattern = "[aeiou]", ";")

## [1] "S;n Fr;nc;sc;" "B;rc;l;n;"       "N;pl;s"         "P;r;s"
```

或者，如果我们想要在每个名称中用分号替换所有辅音字母，我们只需要改变模式并使用否定类：# 替换所有匹配的辅音字母

```
str_replace_all(cities, pattern = "[^aeiou]", ";")

## [1] ";a;;;;a;;i;;o" ";a;;e;o;a"       ";a;;e;"         ";a;i;"
```

6.4.10 使用str split() 进行字符串拆分

与 strsplit()类似，stringr给我们提供了函数str split()来将一个字符向量分成多个部分。 该函数的用法如下：

```
str_split(string, pattern, n = Inf)
```

参数 n是返回的最大片段数。 默认值 (n = Inf)
意味着使用所有可能的拆分位置。

让我们看看 strsplit()的同一个例子，我们希望将一个句子分成单个单词：

```
# 一个句子
sentence = c("R是一个有许多贡献者的合作项目")

# 分成单词
str_split(sentence, " ")
```

```
## [[1]]
## [1] "R"          "是"         "一个"       "合作"
## [5] "项目"       "与"         "许多"       "贡献者"
```

同样，我们可以通过拆分由破折号"-"
连接的数字集来分解电话号码的部分。

```
# 电话号码
tels = c("510-548-2238", "707-231-2440", "650-752-1300")

# 将每个数字拆分为其部分
str_split(tels, "-")
```

```
## [[1]]
## [1] "510"  "548"  "2238"
##
## [[2]]
## [1] "707"  "231"  "2440"
##
## [[3]]
## [1] "650"  "752"  "1300"
```

结果是一个字符向量的列表。 字符串向量的每个元素对应于结果列表中的一个元素。 反过来，列表的每个元素将包含从匹配中出现的拆分向量（即片段数）。

为了展示参数 n的使用，让我们考虑一个包含以下口味的向量"巧克力","香草", "肉桂", "薄荷"和"柠檬"。 假设我们想要将每个口味名称拆分，将元音类作为模式：

```
# 字符串
flavors = c("巧克力", "香草", "肉桂", "薄荷", "柠檬")

# 按元音字母拆分
str_split(flavors, "[aeiou]")
```

```
## [[1]]
## [1] "ch" "c"    "l"   "t"   ""
##
## [[2]]
## [1] "v"   "n"   "ll" ""
##
## [[3]]
## [1] "c"   "nn" "m"   "n"
##
## [[4]]
## [1] "m"   "nt"
##
## [[5]]
## [1] "l" "m" "n"
```

现在让我们将最大片数修改为n = 2。 这意味着str split()
将每个元素分割成最多2个片段。这是我们得到的结果：

```
# 按第一个元音字母分割
str_split(flavors, "[aeiou]", n = 2)
```

```
## [[1]]
## [1] "ch"      "colate"
##
## [[2]]
## [1] "v"      "nilla"
##
## [[3]]
## [1] "c"       "nnamon"
##
## [[4]]
## [1] "m"   "nt"
##
## [[5]]
```

```
## [1] "l"    "mon"
```

6.4.11 使用str split fixed()函数进行字符串拆分

除了str split()之外，还有str split fixed()函数可以将字符串拆分成固定数量的片段。 它的使用形式如下：

```
str_split_fixed(字符串, 模式, n)
```

请注意，参数 n没有默认值。 换句话说，我们需要指定一个整数来表示片段的数量。

再次考虑相同的 flavors向量，以及匹配模式为 "n"的字母。 让我们来看看str split fixed()在n = 2时的行为。

```
# 字符串
flavors = c("巧克力", "香草", "肉桂", "薄荷", "柠檬")

# 将flavors拆分为2个片段
str_split_fixed(flavors, "n", 2)
```

```
##      [,1]         [,2]
## [1,] "巧克力" ""
## [2,] "va"         "尼拉"
## [3,] "ci"         "蒙"
## [4,] "mi"         "t"
## [5,] "柠檬"       ""
```

正如你所看到的，输出是一个字符矩阵，列数为n = 2。 由于"chocolate"不包含任何字母 "n"，所以它在第二列中的对应值保持为空 ""。 相反，与 "lemon"相关联的第二列的值也为空。 但这是因为这种味道被分成了 "lemo"和 ""两部分。

如果我们改变值n = 3，我们将得到一个具有三列的矩阵：

```
# 将味道分成3份
str_split_fixed(flavors, "n", 3)
```

```
##      [,1]         [,2]   [,3]
## [1,] "chocolate" ""      ""
## [2,] "va"         "illa" ""
## [3,] "ci"         ""     "amon"
## [4,] "mi"         "t"    ""
## [5,] "lemo"       ""     ""
```

第7章

实际应用

本章专门介绍一些涉及在 R中处理和处理字符串的实际示例。 主要思想是将迄今为止涵盖的所有材料付诸实践，并了解我们可以在 R中做的各种事情。

我们将描述四个典型的例子。 这些例子并不详尽，只是代表性的：

1. 反转字符串
2. 匹配电子邮件地址
3. 匹配HTML元素（`href`和 `img`的锚点）
4. 一些字符数据的统计和分析

7.1 反转字符串

我们的第一个例子与反转字符字符串有关。 更确切地说，目标是创建一个函数，该函数接受一个字符串并以相反的顺序返回它。 这个练习的技巧取决于我们对 $reversing$一词的理解。 对于一些人来说，反转可能被理解为仅仅将字符集按相反的顺序排列。 对于其他人来说，反转可能被理解为将一组单词按相反的顺序排列。 你能看出区别吗？

让我们考虑以下两个简单的字符串：

- "大气层"
- "宇宙大爆炸理论"

第一个字符串由一个单词（大气层）组成。 第二个字符串由四个单词（宇宙大爆炸理论）组成。 如果我们按字符反转这两个字符串，我们将得到以下结果：

- "erehpsomta"
- "yroeht gnab gib eht"

相反，如果我们按单词反转字符串，我们将得到以下输出：

- "大气层"
- "理论大爆炸宇宙"

对于这个例子，我们将为每种类型的反转操作实现一个函数。

7.1.1 按字符反转字符串

反转字符串的第一种情况是按字符进行。 这意味着我们需要将给定的字符串拆分为不同的字符，然后按相反的顺序将它们连接在一起。 让我们尝试编写第一个函数：

```
# 反转字符串的函数
reverse_chars <- function(string)
{
  # 按字符拆分字符串
  string_split = strsplit(string, split = "")
  # 反转顺序
  rev_order = nchar(string):1
  # 反转字符
  reversed_chars = string_split[[1]][rev_order]
  # 合并反转的字符
  paste(reversed_chars, collapse="")
}
```

让我们用字符和数字向量测试我们的反转函数：

```
# 尝试 'reverse_chars'
reverse_chars("abcdefg")

## [1] "gfedcba"

# 尝试非字符输入
reverse_chars(12345)

## Error:   non-character argument
```

正如你所看到的，当输入为"字符"模式时，reverse_chars()函数工作正常。 然而，当输入为"非字符"时，它会报错。 为了使我们的函数更加健壮，我们可以强制将输入转换为字符。 生成的代码如下所示：

```
# 反转字符串的函数
reverse_chars <- function(string)
{
  string_split = strsplit(as.character(string), split = "")
  reversed_split = string_split[[1]][nchar(string):1]
  paste(reversed_split, collapse="")
}
```

现在，如果我们尝试我们修改后的函数，我们会得到预期的结果：

```
# 一个单词的例子
reverse_chars("atmosphere")

## [1] "erehpsomta"

# 几个单词的例子
reverse_chars("the big bang theory")

## [1] "yroeht gnab gib eht"
```

此外，它还适用于非字符输入：

```
# 尝试 'reverse_chars'
reverse_chars("abcdefg")

## [1] "gfedcba"

# 尝试非字符输入
reverse_chars(12345)

## [1] "54321"
```

如果我们想要将我们的函数与向量（多个元素）一起使用，我们可以将其与 lapply()函数结合使用，如下所示：# 反转向量（按字符）

```

lapply(c("the big bang theory", "atmosphere"), reverse_chars)

## [[1]]
## [1] "yroeht gnab gib eht"
##
## [[2]]
## [1] "erehpsomta"
```

7.1.2 按单词反转字符串

第二种反转操作是按单词反转字符串。 在这种情况下，该过程涉及将字符串按单词拆分，按相反顺序重新排列它们，并将它们粘贴回一个句子中。 下面是我们如何定义我们的reverse words()函数：

```
# 反转字符串的函数
reverse_words <- function(string)
{
  # 按空格拆分字符串
  string_split = strsplit(as.character(string), split = " ")
  # 拆分了多少个词?
  string_length = length(string_split[[1]])
  # 决定要做什么
  if (string_length == 1)  {
    # 一个词（什么都不做）
    reversed_string = string_split[[1]]
  } else {
    # 多个词（合并它们）
    reversed_split = string_split[[1]][string_length:1]
    reversed_string = paste(reversed_split, collapse = " ")
  }
  # 输出
  返回 (reversed_string)
}
```

在reverse words（）函数内部的第一步是根据空格模式" "拆分字符串。 然后我们计算拆分步骤产生的组件数量。 根据这些信息，有两个选项。 如果只有一个单词，那么就不需要做任何操作。如果有多个单词，那么我们需要将它们按相反的顺序重新排列并合并成一个字符串。

一旦我们定义了函数，我们可以尝试在两个字符串示例上运行它，以检查它是否按预期工作：

```
# 示例
reverse_words("大气层")

## [1] "大气层"

reverse_words("宇宙大爆炸理论")

## [1] "理论 爆炸大 宇宙"
```

类似地，要在具有多个元素的向量上使用我们的函数，我们应该在lapply（）函数内调用它，如下所示：# 反转向量（按单词）

```
lapply(c("the big bang theory", "atmosphere"), reverse_words)
```

```
## [[1]]
## [1] "theory bang big the"
##
## [[2]]
## [1] "atmosphere"
```

7.2 匹配电子邮件地址

我们将讨论的第二个实际例子是匹配电子邮件地址。 我们将使用以下形式之一（或类似变体）的常见电子邮件地址进行工作：

```
somename@email.com
somename99@email.com
some.name@email.com
some.name@an-email.com
some.name@an.email.com
```

由于我们的目标是匹配电子邮件地址，这意味着我们需要定义相应的正则表达式模式。 如果我们看一下之前的电子邮件形式，可以看到它们有一个通用的结构，可以分为三个部分。 第一部分是用户名（例如somename99）。 第二部分是@符号。 第三部分是域名（例如an.email.com）。

用户名模式可以定义为：

```
^([a-z0-9_\.-]+)
```

用户名模式以插入符号 ^开头，表示字符串的开头。 然后我们有一个用括号表示的组。 它匹配一个或多个小写字母、数字、下划线、点或连字符。

域名模式可以定义为：

```
([\da-z\.-]+)\.([a-z\.]{2,6})$
```

域名应该是一个或多个小写字母、数字、下划线、点或连字符。 然后是另一个（转义的）点，后面是两到六个字母或点的扩展名。
最后是字符串的结尾（$）。

电子邮件地址的完整正则表达式模式（在 R中）是：

```
"^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$"
```

让我们用一个简单的例子来测试我们的模式：

```
# 模式
email_pat = "^([a-z0-9_ \ .-]+)@([\ da-z\ .-]+)\ .([a-z\ .]{2,6})$"

# 匹配的字符串
grepl(pattern = email_pat, x = "gaston@abc.com")

## [1] TRUE
```

这里是另一个更真实的例子：

```
# 另一个匹配的字符串
grep(pattern = email_pat, x = "gaston.sanchez@research-center.fr")

## [1] 1
```

然而，如果我们有一个超过六个字母的“长”顶级域名（TLD），模式将无法匹配，就像下一个例子中一样：

```
# 不匹配的电子邮件（TLD太长）
grep(pattern = email_pat, x = "gaston@abc.something")

## integer(0)
```

现在让我们将电子邮件模式应用于测试多个字符串是否匹配：

```
# 潜在的电子邮件地址
emails = c(
 "simple@example.com",
 "johnsmith@email.gov",
 "marie.curie@college.edu",
 "very.common@example.com",
 "a.little.lengthy.but.ok@dept.example.com",
 "disposable.style.email.with+symbol@example.com",
 "not_good@email.address")

# 检测模式
str_detect(string=emails, pattern=email_pat)

## [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

请注意，emails中的最后两个元素不是良好定义的电子邮件地址（它们不符合指定的模式）。 第五个地址包含五组字符串，包括一个

+符号。 而第六个地址的域名很长（email.address），其中
address超过六个字母。

7.3 匹配HTML元素

对于我们的第三个示例，我们将处理一些基本的HTML标签。 我们将使用 R邮件列表的网页：http://www.r-project.org/mail.html如果您访问上述网页，您将看到有四个

专门用于 R的常规邮件列表：

- **R-announce**是关于 R开发和新代码可用性的重要公告。
- **R-packages**是关于新的或增强的贡献包可用性的公告列表。
- **R-help**是关于使用 R解决问题和找到解决方案的主要邮件列表。
- **R-devel**是关于 R代码开发的问题和讨论的列表。此外，还有几个特定的**Special I**

nterest Group(SIG) 邮件列表。 下表显示了前5个组：

在 **R**中的前**5**个特殊兴趣小组（**SIG**）

名称	描述
R-SIG-Mac	关于 R的Mac端口的特殊兴趣小组
R-sig-DB	关于数据库接口的SIG
R-SIG-Debian	关于 R的Debian端口的特殊兴趣小组
R-sig-dynamic-models	动态模拟模型的特殊兴趣小组
R-sig-Epi	用于流行病学数据分析的R

作为一个简单的例子，假设我们想要获取所有SIG链接的 href属性。 例如，R-SIG-Mac链接的 href属性是：https://stat.ethz.ch/mailman/listinfo/r-sig-mac而R-sig-D

B链接的 href属性是：https://stat.ethz.ch/mailman/listinfo/r-sig-db

如果我们查看网页的HTML源代码，我们会看到所有链接都可以在这样的行上找到：

```
    <td><a href="https://stat.ethz.ch/mailman/listinfo/r-sig-mac">
<tt>R-SIG-Mac</tt></a></td>
```

7.3.1 获取SIG链接

第一步是创建一个包含邮件列表网页行的字符字符串向量。 我们可以通过将URL名称传递给readLines()来创建这个向量：# 读取HTML内容

```
mail_lists = readLines("http://www.r-project.org/mail.html")
```

一旦我们读取了 R邮件列表网页的HTML内容，下一步是定义匹配SIG链接的正则表达式模式。

```
  '^.*<td> *<a href="(https.*)">.*
```

让我们来检查提议的模式。 通过使用插入符号 (^) 和美元符号 ($)，我们可以将模式描述为整行。 在插入符号旁边，我们匹配零次或多次的任何内容，然后是一个 <td>标签。然后是零次或多次匹配的空格，接着是一个带有其 href属性的锚点标签。 请注意，我们使用双引号来匹配 href属性 ("(https.*)")。 此外，整个正则表达式模式被单引号''包围。 下面是如何获取SIG链接的方法：# SIG的href模式

```
sig_pattern = '^.*<td> *<a href="(https.*)">.*

# 查找SIG href属性
sig_hrefs = grep(sig_pattern, mail_lists, value = TRUE)

# 让我们先看看前5个元素（缩短输出）
shorten_sigs = rep("", 5)
for (i in 1:5)  {
  shorten_sigs[i] = toString(sig_hrefs[i], width=70)
}
shorten_sigs
```

```
## [1] "     <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-mac\">...."
## [2] "     <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-db\"><...."
## [3] "     <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-debia...."
## [4] "     <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-dynam...."
## [5] "     <td><a href=\"https://stat.ethz.ch/mailman/listinfo/r-sig-epi\">...."
```

我们需要摆脱额外的HTML标签。 我们可以使用sub()函数轻松提取注释文件的名称（因为每行只有一个链接，所以我们不需要使用gsub()函数，

虽然我们可以使用）。

```
# 获取第一个匹配的组
sub(sig_pattern, "\ 1", sig_hrefs)
```

```
##  [1] "https://stat.ethz.ch/mailman/listinfo/r-sig-mac"
##  [2] "https://stat.ethz.ch/mailman/listinfo/r-sig-db"
##  [3] "https://stat.ethz.ch/mailman/listinfo/r-sig-debian"
##  [4] "https://stat.ethz.ch/mailman/listinfo/r-sig-dynamic-models"
##  [5] "https://stat.ethz.ch/mailman/listinfo/r-sig-epi"
##  [6] "https://stat.ethz.ch/mailman/listinfo/r-sig-ecology"
##  [7] "https://stat.ethz.ch/mailman/listinfo/r-sig-fedora"
##  [8] "https://stat.ethz.ch/mailman/listinfo/r-sig-finance"
##  [9] "https://stat.ethz.ch/mailman/listinfo/r-sig-geo"
## [10] "https://stat.ethz.ch/mailman/listinfo/r-sig-gr"
## [11] "https://stat.ethz.ch/mailman/listinfo/r-sig-gui"
## [12] "https://stat.ethz.ch/mailman/listinfo/r-sig-hpc"
## [13] "https://stat.ethz.ch/mailman/listinfo/r-sig-jobs"
## [14] "https://stat.ethz.ch/mailman/listinfo/r-sig-mixed-models"
## [15] "https://stat.ethz.ch/mailman/listinfo/r-sig-mediawiki"
## [16] "https://stat.ethz.ch/mailman/listinfo/r-sig-networks"
## [17] "https://stat.ethz.ch/mailman/listinfo/r-sig-phylo"
## [18] "https://stat.ethz.ch/mailman/listinfo/r-sig-qa"
## [19] "https://stat.ethz.ch/mailman/listinfo/r-sig-robust"
## [20] "https://stat.ethz.ch/mailman/listinfo/r-sig-s"
## [21] "https://stat.ethz.ch/mailman/listinfo/r-sig-teaching"
## [22] "https://stat.ethz.ch/mailman/listinfo/r-sig-wiki"
```

正如你所看到的，我们在sub()函数中使用了正则表达式模式 \1。 一般来说，\N会被替换为正则表达式中指定的第N个组。 第一个匹配的组由 \1引用。 在我们的例子中，第一个组是括号中的所有内容，即：(https.*)，实际上这些是我们正在寻找的链接。

7.4 对BioMed Central期刊进行文本分析

对于我们的最后一个应用程序，我们将分析一些文本数据。 我们将分析来自BioMed Central（BMC）的期刊目录，这是一家专门从事开放获取期刊出版的科学出版商。 你可以在这里找到更多关于BMC的信息：http://www.biomedcentral.com/about/catalog

期刊目录的数据以csv格式提供，网址为：http://www.biomedcentral.com/journals/biomedcentraljournallist.txt

要在R中导入数据，我们可以使用read.table()函数读取文件。 只需指定文件的URL位置，并将其传递给read.table()函数。 不要忘记设置参数sep = ","和stringsAsFactors = FALSE# 数据集的链接

```
url = "http://www.biomedcentral.com/journals/biomedcentraljournallist.txt"

# 读取数据 (stringsAsFactors=FALSE)
biomed = read.table(url, header = TRUE, sep = ",", stringsAsFactors = FALSE)
```

我们可以使用str()函数检查数据的结构：

```
# 数据集的结构
str(biomed, vec.len = 1)
```

```
## 'data.frame': 336 obs. of    7 variables:
##  $ Publisher      : chr   "BioMed Central Ltd" ...
##  $ Journal.name   : chr   "AIDS Research and Therapy" ...
##  $ Abbreviation   : chr   "AIDS Res Ther" ...
##  $ ISSN           : chr   "1742-6405" ...
##  $ URL            : chr   "http://www.aidsrestherapy.com" ...
##  $ Start.Date     : int   2004 2011 ...
##  $ Citation.Style: chr   "BIOMEDCENTRAL" ...
```

正如你所看到的，数据框 biomedhas 336 observations and 7 variables. 实际上，除了 Start. Date之外的所有变量都是字符模式。

7.4.1 分析期刊名称

我们将对期刊名称进行简单分析。 目标是研究期刊标题中使用的常见术语。 我们将保持基本水平，但对于更正式（和复杂）的分析，您可以查看包 tm —textmining— （由Ingo F einerer提供）。

为了更好地了解数据的样子，让我们检查一下第一个期刊名称。

```
# first 5 journal names
head(biomed$Journal.name, 5)
```

```
## [1] "AIDS Research and Therapy"
## [2] "AMB Express"
## [3] "Acta Neuropathologica Communications"
## [4] "Acta Veterinaria Scandinavica"
## [5] "Addiction Science & Clinical Practice"
```

正如你所看到的，第五本期刊"Addiction Science & Clinical Practice"有一个和符号。 是否保留和符号和其他标点符号取决于分析的目标。 在我们的案例中，我们将删除这些元素。

预处理

预处理步骤意味着摆脱标点符号。 出于方便的原因，始终建议从数据的一个小子集开始工作。 这样，我们可以在小规模上进行实验，直到我们对正确的操作有信心。 让我们来看看前10本期刊：

```
# 获取前10个名称
titles10 = biomed$Journal.name[1:10]
titles10
```

```
##  [1] "AIDS Research and Therapy"
##  [2] "AMB Express"
##  [3] "Acta Neuropathologica Communications"
##  [4] "Acta Veterinaria Scandinavica"
##  [5] "Addiction Science & Clinical Practice"
##  [6] "Agriculture & Food Security"
##  [7] "Algorithms for Molecular Biology"
##  [8] "Allergy, Asthma & Clinical Immunology"
##  [9] "Alzheimer's Research & Therapy"
## [10] "Animal Biotelemetry"
```

我们想要去掉和其他标点符号一样的&符号 & 这可以通过使用str_replace_all()函数和将模式[[:punct:]]替换为空字符串""(不要忘记加载 stringr package)# remove punctuation

```
titles10 = str_replace_all(titles10, pattern = "[[:punct:]]", "")
titles10
```

```
##  [1] "AIDS Research and Therapy"
##  [2] "AMB Express"
##  [3] "Acta Neuropathologica Communications"
##  [4] "Acta Veterinaria Scandinavica"
##  [5] "Addiction Science  Clinical Practice"
##  [6] "Agriculture  Food Security"
##  [7] "Algorithms for Molecular Biology"
##  [8] "Allergy Asthma  Clinical Immunology"
##  [9] "Alzheimers Research  Therapy"
```

```
## [10] "动物生物遥测"
```

我们成功地用空字符串替换了标点符号，但现在我们有额外的空格。 为了去除空格，我们将再次使用str_replace_all()函数，将任意一个或多个空格 \s+替换为一个空格" "。

```
# 去除额外的空格
titles10 = str_replace_all(titles10, pattern = "  \ s+", " ")
titles10
```

```
##  [1] "AIDS Research and Therapy"
##  [2] "AMB Express"
## [3] "神经病理学通讯"
## [4] "斯堪的纳维亚兽医学"
## [5] "成瘾科学临床实践"
## [6] "农业食品安全"
## [7] "分子生物学算法"
## [8] "过敏哮喘临床免疫学"
## [9] "阿尔茨海默病研究疗法"
## [10] "动物生物遥测"
```

一旦我们对期刊名称的预处理有了更好的理解，我们就可以继续处理所有的336个标题。

```
# 删除标点符号
all_titles = str_replace_all(biomed$Journal.name, pattern = "[[:punct:]]", "")

# 去除额外的空格
all_titles = str_replace_all(all_titles, pattern = "  \ s+", " ")
```

下一步是将标题拆分为不同的术语（输出为列表）。

```
# 按单词拆分标题
all_titles_list = str_split(all_titles, pattern = " ")

# 显示前2个元素
all_titles_list[1:2]
```

```
## [[1]]
## [1] "AIDS"           "研究" "和"       "治疗"
##
## [[2]]
## [1] "AMB"       "表达"
```

摘要统计

到目前为止，我们有一个包含每个期刊名称单词的列表。 知道每个标题中术语数量的分布会很有趣吧？ 这意味着我们需要计算每个标题中有多少个单词。 为了得到这些数字，让我们在 `sapply()`中使用 `length()`；然后让我们制表得到的频率：`#` 每个标题有多少个单词

```
words_per_title = sapply(all_titles_list, length)

# 频率表
table(words_per_title)
```

```
## words_per_title
##   1   2   3   4   5   6   7   8   9
##  17 108  81  55  33  31   6   4   1
```

我们还可以将分布表示为百分比，并且我们可以使用summary()获得一些摘要统计信息`#` 分布

```
100 * round(table(words_per_title)/length(words_per_title), 4)
```

```
## words_per_title
##     1     2     3     4     5     6     7     8     9
##  5.06 32.14 24.11 16.37  9.82  9.23  1.79  1.19  0.30
```

```
# 摘要
summary(words_per_title)
```

```
##    最小值 第一四分位数 中位数    平均值 第三四分位数    最大值
##    1.00    2.00    3.00    3.36    4.00    9.00
```

从摘要统计数据来看，大约有30%的期刊名称有2个单词。
同样，每个标题的中位数单词数为3个单词。

有趣的是，最大值为9个单词。 有9个单词的期刊是哪个？
我们可以通过以下方式找到最长的期刊名称：

```
# 最长的期刊
all_titles[which(words_per_title == 9)]
```

```
## [1] "Journal of Venomous Animals and Toxins including Tropical Diseases"
```

7.4.2 常见词汇

请记住，我们在这个例子中的主要目标是找出期刊标题中最常见的单词。 为了回答这个问题，我们首先需要创建类似于一个词典的东西。 我们如何得到这样的词典呢？ 很简单，我们只需要获取一个包含所有标题中的单词的向量：

```
# 标题中的单词向量
title_words = unlist(all_titles_list)

# 获取唯一的单词
unique_words = unique(title_words)

# 总共有多少个唯一的单词
num_unique_words = length(unique(title_words))
num_unique_words
```

```
## [1] 441
```

将unique()应用于向量title words，我们得到了所需的术语词典，总共有441个单词。

一旦我们有了唯一的单词，我们需要计算每个单词在标题中出现的次数。下面是一种方法：

```
# 用于存储计数的向量
count_words = rep(0, num_unique_words)

# 计算出现次数
for (i in 1:num_unique_words)  {
    count_words[i] = sum(title_words == unique_words[i])
}
```

另一种更简单的方法是使用 `table()` 函数来计算单词出现的次数:

```
# 单词频率表
count_words_alt = table(title_words)
```

在这两种情况下（计算单词或计算单词 alt）_，我们可以用一个简单的表格来查看得到的频率：

```
# 频率表
table(count_words)
```

```
## count_words
```

```
##   1   2   3   4   5   6   7   9  10  12  13  16  22  24  28  30  65  67
## 318  61  24   8   5   7   1   2   2   2   1   2   1   1   1   1   1   1
##  83  86
##   1   1
```

```
# 等效地
table(count_words_alt)
```

```
## count_words_alt
##   1   2   3   4   5   6   7   9  10  12  13  16  22  24  28  30  65  67
## 318  61  24   8   5   7   1   2   2   2   1   2   1   1   1   1   1   1
##  83  86
##   1   1
```

前30个单词

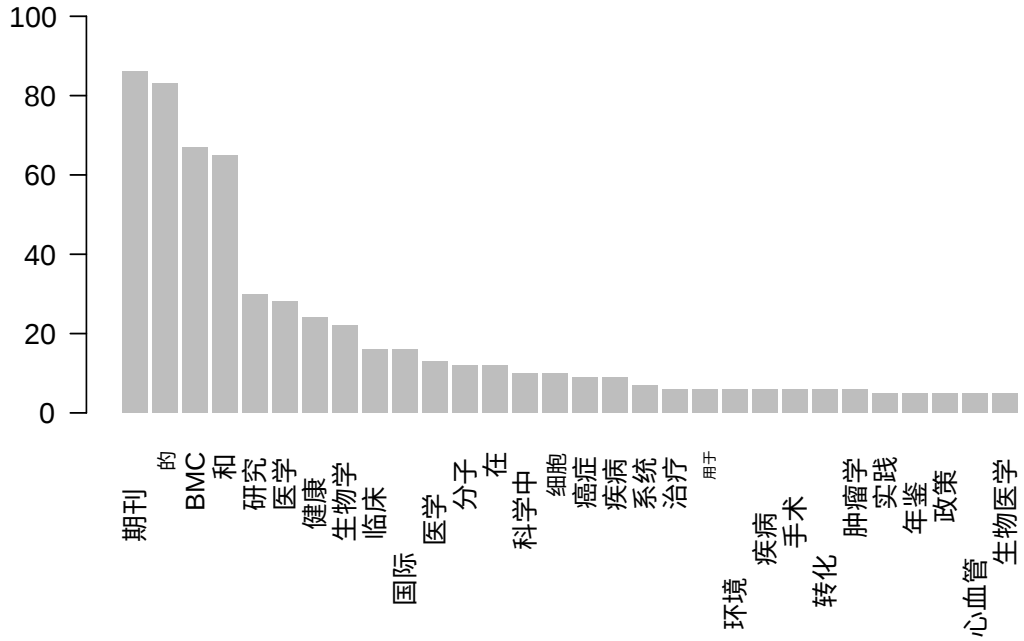
为了说明目的，让我们看一下前30个常见的单词。

```
# 按降序排列的索引值
top_30_order = order(count_words, decreasing = TRUE)[1:30]

# 前30个频率
top_30_freqs = sort(count_words, decreasing = TRUE)[1:30]

# 选择前30个单词
top_30_words = unique_words[top_30_order]
top_30_words
```

```
##  [1] "Journal"          "的"               "BMC"              "和"
## [5] "研究"             "医学"             "健康"             "生物学"
## [9] "临床"             "国际" "医学"                        "分子"
## [13] "在"              "科学"             "细胞"             "癌症"
## [17] "疾病"            "系统"             "治疗"             "用于"
## [21] "环境" "疾病"                        "外科"             "转化"
## [25] "肿瘤学"          "实践"             "年鉴"             "政策"
## [29] "心血管" "生物医学"
```

要可视化前30个单词，我们可以使用barplot()函数绘制条形图：

```
# 条形图
barplot(top_30_freqs, border = NA, names.arg = top_30_words,
        las = 2, ylim = c(0,100))
```



词云

为了完成这一部分，让我们尝试使用一个词云来进行另一种可视化输出，也被称为标签云。 为了获得这种类型的图形显示，我们将使用 *wordcloud*包（由Ian Fel-lows开发）。 如果您还没有下载该包，请记得使用`install.packages()`进行安装。

```
# 安装wordcloud
install.packages("wordcloud")

# 加载wordcloud
library(wordcloud)
```

要绘制一个标签云，只需要使用函数`wordcloud()`即可。 这个函数的最基本用法如下所示：

```
  wordcloud(words, freq}
```

它需要两个主要参数：一个字符向量 `words`和一个数值向量 `freq`，其中包含单词的频率。 在我们的例子中，单词向量对应于唯一单词向量。 而频率向量对应于计数单词。 下面是绘制具有最小频率为6的这些术语的词云的代码。

```
# 词云
wordcloud(unique_words, count_words, scale=c(8,.2), min.freq=6,
          max.words=Inf, random.order=FALSE, rot.per=.15)
```