R 语言定义

本册主要对 R 语言，赋值解释（explaining evaluation），解析， 面向对象编程，语言上的计算等方面进行一个介绍。

本文档的当前版本为 0.01 β **草稿**。该文档译自 R-2.3.1 文档（2006 年 6 月 1 日）。

丁国徽（ghding@gmail.com） 译。

本文档的一些发布信息放置在 http://www.biosino.org/R/R-doc/。

ISBN 3-900051-13-5

说明

版权声明

英文文档版权声明：

The R language definition 是一份草稿，里面有很多问题，有些文本是残缺的，也有些地方是 矛盾的。但是对 R 的核心探讨（用原文中的话就是“per se”）还是比较细致了。

我在编译这份文档的过程中，有很多地方也比较地糊涂。只是，匆忙中，先拿出一份草稿（ ^_^，R 官方对应的文档现在也是草稿）。有一个靶子，大家总好讨论一点。 在很多地方都斟酌后，考虑后，我会把它做成比较正规的 PDF 版，也交给 Friedrich Leisch 兄弟。

如果只是用 R 做做统计，绘绘图，可能看看《R 导论》也差不多了。 但是要更细致的了解 R，这本可能就比较的精髓了。可以说，其它几份文档 都是它的派生版本。

不过，刚开始学 R 的时候，看它可能非常的不喜欢。不喜欢去研究 R 底层的人可能也不会去看。毕竟，R 的前台语法定义足够你完成几乎所有 你想解决的事情了（当然，我主要指数据分析方面的）。但是， 理解 `1 + 2` 就是 `"+"(1, 2)` 还是一件比较有用的事情。

任何问题和建议可以给 Email！

感谢身边的朋友！

丁国徽

Email：ghding@gmail.com

2006 年 10 月 5 日

---

1 绪论

R 是为统计计算和图形展示而设计的一个系统。 它包括一种编程语言，高级别图形展示函数，和其它语言的接口以及调试工具。 本手册将会详细描述和定义 R 语言。

R 是统计领域广泛使用的诞生于 1980 年左右的 S 语言的一个分支。 S 的主要设计者 John M. Chambers 因为 S 语言方面的工作获得了 1998 年 ACM 软件系统奖（ACM Software Systems Award）。

该语言的语法表面上类似 C，但在语义上是函数设计语言的（functional programming language） 的变种并且和 Lisp 以及 APL 有很强的兼容性。 特别的是，它允许在“语言上计算”（computing on the language）。 这使得它可以把表达式作为函数的输入参数，而这种做法对 统计模拟和绘图非常有用。

通过命令行运行简单的表达式可以充分地交互使用 R 的功能。 一些用户可能这样做就能满足要求了，但还有一些用户想编写他们自己的函数。 编写函数的用户要么用以一种特别的方式，系统化一些常常重复的工作或者 为新的功能编写扩展包。

本手册的目的是想记录 R 语言的*本质*。 也就是它所工作的对象，表达式赋值过程的细节。这些内容的了解对编写 R 函数非常有用。 另外一些针对特定任务的主题，如绘图，在本册里面只是简单描述，而在其它手册里面 有专门的论述。

尽管手册中的大部分内容同样适用于 S，但 S 和 R 还是有一些 实质上的差异。为了不至于混淆，我们集中描述 R。

R 语言的设计包含了一系列亮点，当然也有让用户惊讶的公共缺陷。 许多设计是基于底层的连贯性考虑，我们会在后面的行文中解释。 它还包括很多有用的快捷方式和特殊用法，使得用户可以很简洁的表述复杂的操作。 一旦用户熟悉底层的概念，这些用法将会变得非常的自然。 在某些情况下，有多种方法完成同一件事情，但是 其中有些技术依赖于语言的实现，另外一些技术则是一个更高层次上的抽象。 在这种情况下，我们会指出首选的用法。

读本册前，我们假定用户对 R 已经有一定的了解。这不是一本 R 的入门读物，而是一本程序员的参考手册。其它文档给出了互补的信息： 特别 Preface (R Introduction) 给出 R 语言的入门介绍和 System and foreign language interfaces (Writing R Extensions) 详细介绍如何用编译好的代码扩充 R 语言。

---

2 对象

在所有编程语言中， 变量提供了一种访问内存中数据的方法。 R 没有提供直接访问计算机内存的方法，但提供了许多我们称之为对象的特殊数据结构。 这些对象通过变量或者符号（symbol）访问。不过在 R 语言里面，符号本身就是对象并且和使用其它对象一样的方式使用。这和许多其它语言不同，但有广泛的影响。

在本章，我们会给出 R 里面各种数据结构的初步描述。对这些数据结构更为详细的讨论会在后面的章节中展开。 R 语言特有的函数 `typeof` 返回 R 对象的*类型*。注意在 R 底层的 C 代码中，所有对象都是指向一个有类型定义 `SEXPREC` 的结构体（structure）的指针；不同的 R 数据类型在 C 里面用决定结构体各部分信息的 `SEXPTYPE` 表示。

下面的表格描述了 `typeof` 可能的返回值以及它们的涵义。

`NULL`	空
`symbol`	一个变量名字
`pairlist`	成对列表对象
`closure`	一个函数
`environment`	一个环境
`promise`	一个用于实现悠闲赋值的对象
`language`	一个 R 语言构建

special	一个不可针对参数求值的内置函数
builtin	一个可针对参数求值的内置函数
logical	含逻辑值的向量
integer	含整数值的向量
double	含实数值的向量
complex	含复数值的向量
character	含字符值的向量
...	特定变量长度参数 ***
any	一个可以匹配任何类型的特殊类型 ***
expression	一个表达式对象
list	一个列表
externalptr	一个外表指针对象
weakref	一个弱引用对象（a weak reference object）
raw	一个字节元素向量

我认为用户不用深入以`***'标记的条目，至少没有想象的那么容易；但是可以多看一些例子。

根据 Becker，Chambers & Wilks (1988)中的说明，函数 `mode` 返回对象的 *模式*信息，并且和其它 S 语言的变种完全兼容。 最后，同样基于 Becker et al. (1988)的考虑，函数 `storage.mode` 返回其参数的*存储模式*（storage mode）。该函数常常用于，在外部语言（如 C 或 FORTRAN）中调用函数时确保 R 对象有被调用的程序所期望的数据对象。（在 S 语言里面，整数值或实数值向量都是 "numeric"模式，因此它们的存储模式需要区分。）

```
> x <- 1:3
> typeof(x)
[1] "integer"
> mode(x)
[1] "numeric"
> storage.mode(x)
[1] "integer"
```

R 在计算过程中，对象常常需要强制转换成 不同的类型（type）。有许多函数可用于显式的强制转换。 在仅仅用 R 语言编程的时候，一个对象的类型通常不会影响计算结果，但是当混合使用外部编程语言或不同的操作系统时，常常需要保证对象类型的正确。

- Basic types: 基本类型
- Attributes: 属性
- Special compound objects: 特殊的混合对象

2.1 基本类型

- Vector objects: 向量对象
- List objects: 列表对象
- Language objects: 语言对象
- Expression objects: 表达式对象
- Function objects: 函数对象
- NULL object: 空对象
- Built-in objects and special forms: 内置对象和特别形态
- Promise objects: 允诺对象
- Dot-dot-dot: ...对象
- Environment objects: 环境对象
- Pairlist objects: 成对列表对象
- Any-type: 任意类型

2.1.1 向量

向量可以看着是由一系列包含数据的紧密联结的单元格子构成。这些单元格通过 类似 `x[5]`的索引操作来访问。更细节的内容可以参考 Indexing。

R 有六个基本（`原子性'）向量类型：逻辑型，整数型，实数型，复数型，字符串（字符）型和原味型（raw）。这些不同向量类型的模式和存储模式如下表所示。

typeof	mode	storage.mode
logical	logical	logical
integer	numeric	integer
double	numeric	double
complex	complex	complex

```
character character character

raw       raw       raw
```

单个的数字，如 `4.2`，以及字符串，如`"four point two"`，仍然是长度为 1 的向量，因为没有更基本的数据类型了。零长度向量是允许的（也是非常有用的）。

字符串向量的模式和存储模式都是 `"character"`。字符向量的单个元素常常是*字符串*。

2.1.2 列表

列表 (“广义向量”) 是另外一种数据存储方式。列表含有元素，每一个元素可以是任意 R 对象类型，也就是说，列表的各个元素可以是不同的数据类型。列表元素的访问可以通过三个不同 索引操作实现。这些在 Indexing 部分将会详细介绍。

列表是向量，并且在不能使用列表时，基本的向量类型可以转换为*原子向量*。

2.1.3 语言对象

三种对象类型构成了 R 语言的全部。它们分别是*调用类型（calls）*，*表达式类型（expressions）* 和*命名类型（names）*。 既然 R 有 `"expression"` 类型的对象，所以我们应该尽量避免在其它地方使用“表达式”这个词。需要注意的是，语法上正确的表达式会被看作是*程序语句（statements）*。 这些对象分别有 `"call"`，`"expression"`，和 `"name"` 三种模式。

这些对象可以利用 `quote` 机制从表达式直接创建，并且可以通过函数 `as.list` 和 `as.call` 与列表相互转换。 解析树的分量可以通过标准的索引操作析取。

- Symbol objects: 符号对象

2.1.3.1 符号对象

符号引用 R 对象。 任意 R 对象的名字常常是符号。 符号可以通过函数 `quote` 创建。

符号的模式是 "name"，存储模式为 "symbol"而类型是 "symbol"。它们可以用函数 as.character 和 as.name 强制转换成 字符串。 它们看上去像解析表达式（parsed expressions）的一部分。读者可以试试 as.list(quote(x + y))。

2.1.4 表达式对象

在 R 里面，我们可以创建类型为 "expression" 的对象。一个 *表达式*（expression）含有一个或多个程序语句。其中，程序语句（statement）指的是语法上正确的一群标记的聚集。 表达式对象是一种特殊语言对象，它包含一些解析过但还未求值的 R 语句。相比其它语言对象，它的主要差别在于一个表达式对象可以包含多个同类型的表达式。另外一个更细微的差别是，"expression"类型的对象仅仅当它显式地传递给函数 eval 时才求值， 而其它语言对象可在一些意想不到的情况下求值。

表达式对象的操作行为非常像列表，它的元素访问方式和列表元素的访问方式一样。

2.1.5 函数对象

在 R 里面，函数是对象并且可以有许多和其它对象类似的操作方法。函数（更准确地说是函数闭包（function closure））有三个基本的组成部分：形式化的参数列表，功能实现的主体和 环境。参数列表是一个以逗号分割的参数的列表。参数可以是符号，或者是 *symbol = default* 的形式，或者是特殊参数 ...。第二种参数形式常常用于设置参数的默认值。如果函数调用时参数缺省，该值会被采用。 ...参数比较特殊，而且可以包含任意多的参数。它通常在参数个数未知或者某些参数会传递给其它函数的情况下使用。

功能实现的主体是解析过的 R 语句。它常常是大括弧里面的一系列程序语句。当然，它也有可能是一个单独的语句，一个符号，甚至是一个常量。

函数的环境 指的是当一个函数被创建时所激活的环境。任何被该环境绑定的符号都可以被函数 *调用*和访问。函数代码及其对应环境中绑定的东西构成的组合体称为`函数闭包'（function closure）。该术语源自函数化编程理论（functional programming theory）。在本文档中，我们主要使用术语`函数'，但使用`闭包'（closure）来强调一个函数相关环境的重要性。

可以通过 formals， body，和 environment 三个构造（这三个构造也可用在赋值语句的左边）来析取和操作闭包的三个部分。 最后一个构造可以用来去掉不想要的环境捕获物（environment capture）。

当一个函数被调用时，一个新的环境（称为*求值环境（evaluation environment）*）将会被创建。该环境的外围（enclosure，见 Environment objects）来自函数闭包的环境。这个新的环境最初由函数的未被求值的参数构成；当求值过程进行时，局部变量将会在该环境中创建。

可以方便地用 `as.list` 和 `as.function` 将函数和列表结构相互转换。 这些方法可以用来实现和 S 兼容，但我们不推崇这样使用。

2.1.6 空对象

`NULL` 是一种非常特殊的对象。它用于表明一个对象不存在。注意不能混淆空对象与零长度的向量/列表。 `NULL` 对象没有类型也没有可以更改的特性。在 R 里面，只用 `NULL` 对象允许被所有的实例对象引用（唯一的一个）。为了检测一个对象是否是 `NULL`，可以使用代码 `is.null`。你不可以设置 `NULL` 的属性。

2.1.7 内置对象和特别形态

有两种类型的对象含有 R 的内置函数， 就是代码列表中显示为 `.Primitive` 的部分。这两种对象的差异在于参数的处理方式。内置函数将对它们所有参数求值并且传给原始的函数，即 *值调用（call-by-value）*。而一些特别函数把没有求值的表达式传给内部函数。

从 R 语言的角度来说，这些对象仅仅是另外一种函数，只是它们的定义不能列出而已。函数 `typeof` 可以把它们与解释型函数（interpreted function）区分开。

2.1.8 允诺对象

允诺对象（promise objects）是 R 的悠闲（lazy）求值机制的一部分。它们含有三个槽（slots）：值，表达式和环境。 当一个函数 被调用，首先参数匹配，然后每个形式参数都会被一个允诺约束。用作形似参数的表达式以及函数调用的环境的指针都保存在允诺里面。

直到该参数被访问，才会有*值*关联允诺。当参数被访问时，保存的表达式会在保存的环境中求值，并返回结果。 结果同样被允诺保存。函数 `substitute` 会提取一个表达式槽里面的内容。这使得程序员既可以访问允诺相关的值也可以访问相关的表达式。

在 R 语言里面，允诺对象常常是隐含的对象。（在以后的 R 发布版本中，它们相对 R 代码是透明的，因为它们总是在被访问的时候求值。）实际的函数参数是这种类型的。函数 `delayedAssign` 可以使一个允诺出现在表达式的外面。通

常还没有办法在 R 代码里面检验一个对象是否是允诺，同时也没有方法用 R 代码确定一个允诺的环境。

2.1.9 ...对象

... 对象类型以列表形式保存。... 的分量在 C 代码里面可以像常规的列表一样访问，但在解释型的代码里面不像一个对象那样容易访问。该对象可以捕获作为一个列表，因此在函数 `table` 的实现代码中，我们可以看到下面的例子

```
    args <- list(...)
## ....
    for (a in args) {
## ....
```

如果一个函数以 ... 作为形式参数，那么任何不匹配形式参数的实际参数都将匹配...。

2.1.10 环境

环境可以简单地看作由两部分组成。一个是包含“符号-值”对集合的*框架（frame）*，另一个是指向外围环境的指针（又称为*（外围）*）（enclosure）。当 R 搜索一个符号的值时，框架将会被检查。如果找到了一个匹配的符号，它的值将会被返回。如果找不到，外围环境将会被访问并且重复这个过程。环境形成一个树形结构，而外围起到一个父节点的角色。环境的树结构的根部是一个空的环境，可以通过没有父节点的 `emptyenv()` 访问。它是基本包环境的直接父节点（可以通过函数 `baseenv()`访问）。 以前， `baseenv()` 可以是 `NULL`，但从版本 2.3.0 开始，不赞成用 `NULL` 作为环境。

环境通过函数调用隐式创建，这些内容在 Function objects 和 Lexical environment 部分将会描述。在这种情况下，环境含有函数的局部变量（包括参数），而它的外围是当前调用函数的环境。环境还可以直接通过 `new.env` 创建。一个环境的框架内容可以通过 `ls`，`get`，`assign` 以及 `eval` 和 `evalq` 访问和操作。

函数 `parent.env` 可以用于访问一个环境的外围。

和其它的 R 对象不一样，环境不是通过拷贝一份新的传递给函数或者用于赋值操作中。因此，如果你将一个的环境赋给好几个符号并且改变其中的一个，那么其它的也都会改变。特别是，把一个属性赋给一个环境会导致一些非常奇异的事情。

2.1.11 成对列表对象

成对列表对象和 Lisp 的点-对列表（dotted-pair list）类似。它们广泛用于 R 的内部。但很少见于解释型的代码里面，尽管它们被 `formals` 返回或通过函数 `pairlist` 创建。一个零长度的成对列表是 `NULL`，这和 Lisp 期望的一样但与零长度列表不一样。 每个这样的对象有三个槽变量，CAR 值，CDR 值和 TAG 值。TAG 值是文本字符串，CAR 和 CDR 分别表示一个以空对象作为终结符的列表的列表项目（头）和剩余项目（尾）（CAR/CDR 术语是传统 Lisp 术语，最初用于 60 年代 IBM 电脑的寄存器）。

R 语言里面成对列表的操作和广义向量(“列表”)完全一样。特别的是，元素也是通过 `[[]]` 方式访问。由于广义向量可以更为高效地应用，一般不赞同使用成对列表。如果通过 R 访问一个内部的成对列表，我们常常把它转换成广义向量（包括取子集的操作）。

在很少情况下，用户可以看到成对列表：其中一个例子是 `.Options`。

2.1.12 “任意” 类型

事实上，不可能有一个对象是“任意”（any）类型的，但它仍然是一个合法的类型值。它用于一些特定环境中（非常非常地少），如 `as.vector(x, "any")` 表明没有必要采用类型的强制转换。

2.2 属性

除了 `NULL`，所有对象有一个或多个相关属性。属性以列表形式保存，其中所有元素都有名字。属性列表可以通过 `attributes` 得到或通过 `attributes<-` 设定。单个的属性分量可以通过 `attr` 和 `attr<-`访问。

一些属性有特别的访问函数 （如和因子相关的 `levels<-`），但这些只在可用的情况下才能使用。为了隐藏实现细节，它们可以进行一些额外的操作。 R 尝试拦截对含有特别属性的 `attr<-` 和 `attributes<-` 的调用，以强迫进行一致性的检验。

矩阵和数组是含有属性 `dim` 及可选属性 `dimnames` 的简单向量。

属性用于实现 R 里面的类结构。如果一个对象有一个 `class` 属性，那么该属性将会在求值过程中被检验。 R 的类结构会在 Object-oriented programming 部分仔细描述。

- Names: 名字
- Dimensions: 维度
- Dimnames: 维度名字
- Classes: 类

- Time series attributes: 时间序列属性

2.2.1 名字

如果存在 `names` 属性，`names` 属性会为一个向量或列表中的每个元素加上标签。当对象被打印时，`names` 属性同样用于为元素加上标签（在存在 `names` 属性的情况下）。 `names` 属性还可以用作索引，例如 `quantile(x)["25%"]`。

可以通过 `names` 和 `names<-` 构造取得和设置名字。 后者将会执行必要的一致性检验以保证名字属性有适合的类型和长度。

成对列表和一维数组的处理比较特殊。对于成对列表对象，一个虚拟的 `names` 属性将被使用；`names` 属性实际上通过列表分量的标签构建。对于一维数组，`names` 属性事实上是访问 `dimnames[[1]]`。

2.2.2 维度

`dim` 属性用于实现数组。数组的内容保存在一个列优先排列（column-major order）的向量中而 `dim` 属性是一个指定数组各维度长度的整数向量。 R 保证了向量的长度是各维度长度的乘积。一个或多个维度的长度可以为 0。

向量和一维数组不一样，因为后者有长度为 1 的 `dim` 属性，而前者没有 `dim` 属性。

2.2.3 维度名字

数组会利用由字符向量构成的 `dimnames` 属性给各个维度命名。 `dimnames` 列表自身也可能有名字，在打印数组的时候这个名字可用作维度名字（extent headings）。

2.2.4 类

R 有一个精心设计的类系统。它通过 `class` 属性控制。该属性是一个含有类列表的字符向量，这些类可以被对象继承。这构成了 R 里面“泛型方法”（generic methods）功能性的基础。

该属性可以没有限制的被用户虚拟访问和操作。对于一个对象是否真的含有类方法期望的组成要素是不会被检验的。因此，改变 `class` 属性需要小心一点。当实在需要的时候，建议使用一些特别的创建和强制转换函数。

2.2.5 时间序列属性

`tsp` 属性用来保存时间序列的参数，起点，终点和频率。该构造主要用于处理有周期性背景的序列数据（如月或季度数据）。

2.3 特殊的复合对象

- Factors: 因子
- Data frame objects: 数据框对象

2.3.1 因子

因子可用于描述含有数目有限值（性别，社会阶层等）的条目。因子有一个 `levels` 属性和`"factor"`类。另外，它还可以拥有一个可选的 `contrasts` 属性。`contrasts` 属性用于控制模型构建函数中的参数设置。 因子可能是完全无序的或者有序的分类。在后面的例子中，可以根据因子的不同类型定义并且有一个对应的 `class` 向量 `c("ordered"," factor")`。

现在，因子通过指定实际水平的整型数组和一个映射整数到名字的字符数组来实现。不幸得是，用户常常利用这种实现方式让一些计算变得比较简单。但是，这只是一个实现的问题，而不能保证 R 的所有实现中都可以这样。

2.3.2 数据框对象

数据框是 R 里面模仿 SAS 或 SPSS 数据集最像的数据结构，即数据的“个体-变量”（cases by variables）矩阵。

数据框是由长度一样（如果是矩阵则是行数一致）的向量，因子和/或矩阵构成的列表。此外，数据框通常有一个 `names` 属性来标记变量和 `row.names` 属性来标记个体。

数据框可以包含一个同其它分量长度一致的列表。该列表可以包含不同长度的元素，这样就提供了一种参差数组（ragged arrays）的数据结构。但是在写本文档的时候，这种数组通常还不能正确处理。

3 表达式的求值

当用户在命令行上键入一行命令（或者从文件中读入一个表达式），首先该命令将会被解析成一个内在的表述方式。 求值程序执行解析后的 R 表达式并且返回表达式的值。所有表达式都有一个值。这就是 R 语言的核心。

本章描述求值程序的基本机制，但不讨论特定函数或者后面独立章节将讨论的函数组或者帮助文档已经提供足够信息内容。

用户可以自己构建表达式并对它们调用求值程序。

- Simple evaluation: 简单求值
- Control structures: 控制结构
- Elementary arithmetic operations: 初等算术操作
- Indexing: 索引
- Scope of variables: 变量作用域

3.1 简单求值

- Constants: 常量
- Symbol lookup: 符号查找
- Function calls: 函数调用
- Operators: 操作符

3.1.1 常量

任何直接在提示符下面键入的数字都是常量，且被求值。

```
> 1
[1] 1
```

常量比较地单调。如果要做更多的事情，我们需要符号。

3.1.2 符号查找

当一个新的变量创建时，它需要一个可以被引用的名字。通常，它还需要一个值。 名字本身就是 符号。当一个符号被求值时， 它的值就会被返回。 后面我们将会仔细解释怎样决定一个和符号相关的值。

在下面简单的例子中，`y` 是一个符号并且它的值是 4。符号也是 R 对象，但是我们很少需要直接处理符号，除了“在语言上编程”（programming on the language）(Computing on the language)。

```
> y <- 4
> y
[1] 4
```

3.1.3 函数调用

R 里面的很多计算都有函数求值。我们也把这称之为函数 *调用*（invocation）。函数调用通过名字和一个以逗号分割的参数列表来实现。

```
> mean(1:10)
[1] 5.5
```

在这个例子中，函数 `mean` 的调用过程中只有一个参数，就是含有 1 到 10 之间整数的向量。

R 含有许多用于各种目的的函数。大多数用来产生一个属于 R 对象的结果，但其它一些函数则利用了它们的副作用，如打印和绘图函数。

函数调用可以用*标签*标记的参数（或*命名*参数），而在 `plot(x, y, pch = 3)` 中，一些参数没有标签，这些参数通过*位置*识别。此时，函数必须通过参数在参数列表中所处的序列顺序来判断它们的意义。因此前面的例子中，`x` 表示横坐标变量，`y` 表示纵坐标变量。使用标签/名字对含有很多可选参数的函数非常方便。

一个特别的函数调用可以出现在赋值操作符的左边，如下所示

```
> class(x) <- "foo"
```

该语句实际所做的就是利用原始的对象和右边部分调用函数 `class<-`。该函数对对象进行修改，返回结果存入原始变量。（至少在概念上，这就是所发生的事情。可能，一些额外的努力将用于避免不必要的数据重复。）

3.1.4 操作符

R 允许使用 C 语言类似的操作符构建算术表达式，例如，

```
> 1 + 2
[1] 3
```

表达式可以用括号合并成组，混以函数调用，然后以一种直接的方式赋给变量

```
> y <- 2 * (a + log(x))
```

R 含有一系列操作符。它们如下表所示。

-	减号，一元操作符或者二元操作符
+	加号，一元操作符或者二元操作符
!	一元否操作符
~	波浪号，用于模型公式，既可以是一元操作符也可以是二元操作符
?	帮助
:	序列，二元操作符（在模型公式中，表示交互效应）
*	乘法，二元操作符
/	除法，二元操作符
^	幂运算符，二元操作符
%*x*%	特殊二元操作符，*x* 可以被任意合法的名字替换
%%	求模，二元操作符
%/%	整除，二元操作符
%*%	矩阵相乘，二元操作符
%o%	外积，二元操作符
%x%	Kronecker 乘积，二元操作符
%in%	匹配操作，二元操作符（在模型公式中，表示嵌套）
<	小于，二元操作符
>	大于，二元操作符
==	等于，二元操作符
>=	大于等于，二元操作符
<=	小于等于，二元操作符

`&`　　与操作，二元操作符，向量模式

`&&`　　与操作，二元操作符，不是向量模式

`|`　　或操作，二元操作符，向量模式

`||`　　或操作，二元操作符，不是向量模式

`<-`　　左赋值，二元操作符

`->`　　右赋值，二元操作符

`$`　　列表子集，二元操作符

除了语法上，操作符使用和函数调用没有差异。事实上，`x + y` 和 `"+"(x, y)` 等价。注意既然 + 不是一个标准的函数名字，，那么它就需要被引号括起来。

R 同时处理数据的整个向量，并且大多数元素操作符和基本的数学函数如 `log` 是向量模式的（和上面表格中提示的一样）。这意味着如果两个一样长度的向量相加会隐式依据向量索引循环计算得到一个含有元素方式加和结果的向量。这种用法同样适合其它操作符，如`-`，`*`，和 `/`，以及可以推广到更高维的结构。需要注意的是，两个矩阵的相乘不会得到通常的矩阵乘积（`%*%`操作符用于这种目的）。一些和向量操作相关的要点将会在 Elementary arithmetic operations 部分讨论。

为访问向量的某个元素，我们常常使用 `x[i]` 语句。

```
> x <- rnorm(5)
> x
[1] -0.12526937 -0.27961154 -1.03718717 -0.08156527  1.37167090
> x[2]
[1] -0.2796115
```

列表分量则更多地用 `x$a` 和 `x[[i]]` 方式访问。

```
> x <- options()
> x$prompt
[1] "> "
```

索引构造（Indexing constructs）同样可以出现在一个赋值操作的右边。 和其它操作符类似，索引实际上也是通过函数实现，可以用 `"["(x, 2)` 代替 `x[2]`。

R 的索引操作含有许多高级特性。这部分内容将在 Indexing 部分进一步描述。

3.2 控制结构

R 里面的计算包括顺序地对*语句*求值。程序语句，如 `x<-1:10` 或 `mean(y)`，可以被分号或者新的一行分割。只要 整个语句在语法上是完整的，该语句就会被求值并且将 *值*返回。一个语句的求值结果可以看作是该语句的值 [1] 这个值通常会赋给一个符号。

分号和换行符可以用来分隔程序语句。分号一般表示一个语句的结束而新的一行只是 *有可能*表示一个语句的结束。如果当前语句在语法上还不完整，换行符会被求值程序忽略掉。如果会话（session）是交互式的，提示符会从 `>` 变为 `+`。

```
> x <- 0; x + 5
[1] 5
> y <- 1:10
> 1; 2
[1] 1
[1] 2
```

语句可以用 { 和 } 组合在一起。一组这样的语句有时会被称为*句块*（block）。单个语句会在其语法完整后键入新的一行时求值。句块不会求值，直到在一个封闭的大括号后面键入新的一行。这一节余下的部分，*语句* 要么指单个语句要么指句块。

```
> { x <- 0
+ x + 5
+ }
[1] 5
```

- if: if 语句
- Looping: 循环控制
- repeat: repeat 语句
- while: while 语句
- for: for 语句
- switch: switch 语句

Footnotes

[1] 求值常常在一个环境中进行。 具体参考 Scope of variables。

3.2.1 if 语句

`if/else` 语句有条件地对两个语句求值。该句式的*条件*语句会被求值，如果它的*值* 是 `TRUE` 那么第一个语句将会被执行；否则第二个语句会被执行。 `if/else` 语句返回所选语句求值结果并作为它的值。语法形式为

```
if ( statement1 )
    statement2
else
```

```
    statement3
```

首先，*statement1* 被求值得到 *value1*。如果 *value1* 是一个首元素为 TRUE 的逻辑向量，那么 *statement2* 将会被求值。如果 *value1* 的第一个元素是 FALSE 那么 *statement3* 将会被求值。如果 *value1* 是一个数值向量，那么 *statement3* 在 *value1* 的第一个元素是零的时候求值，否则 *statement2* 将会被求值。只有 *value1* 的第一个元素才会被使用。其它元素都会被忽略的。如果 *value1* 是逻辑和数值向量以外的向量，将会返回错误。

If/else 语句可以用来防止一些数值计算问题，如对负数进行对数操作。因为 if/else 语句和其它语句一样，你可以把它们的值赋给其它变量。下面的两个例子等价。

```
> if( any(x <= 0) ) y <- log(1+x) else y <- log(x)
> y <- if( any(x <= 0) ) log(1+x) else log(x)
```

`else` 子句是可选的。 语句 `if(any(x <= 0)) x <- x[x <= 0]` 是合法的。如果 `if` 语句不在一个句块中，那么 `else` 子句（假定存在）必须和 *statement1* 在同一行。否则，在 *statement1* 后的新一行将产生一个语法上完整并会被求值的语句。

If/else 语句可以被嵌套。

```
if ( statement1 )
    statement2
else if ( statement3 )
    statement4
else if ( statement5 )
    statement6
else
    statement8
```

一个偶数编号的语句将会被求值并且返回结果值。如果忽略可选的 `else` 子句，并且所有奇数编号的 *statement* 的值都是 FALSE，那么就没有语句会被求值则返回 NULL。

奇数编号的 *statement* 会依次求值，直到一个值为 TRUE，然后对应的偶数编号的 *statement* 会被求值。在这个例子中，*statement6* 当且仅当 *statement1* 为 FALSE，*statement3* 为 FALSE 和 *statement5* 是 TRUE 的情况下才会被求值。`else if` 子句的数目是没有限制的。

3.2.2 循环控制

R 有三种语句实现显式的循环控制。[1] 它们分别是 `for`， `while` 和 `repeat`。两个内置的构造 `next` 和 `break` 提供了对求值过程额外的控制。这三个语句都返回最后语句的求值结果。因此，可以把这些语句的结果值赋给一个符号，尽管这种做法不太常见。 R 还提供了其它一些隐式的循环控制函数，如 `tapply`，`apply`

和 lapply。此外，许多操作，特别算术操作，都是向量模式的，因此你可能不太需要使用循环。

有两种语句可用于显式地循环控制。它们是 break 和 next。 break 语句可以从当前运行的最内部的循环里面跳出。 next 语句会导致控制立即返回到循环的起点，循环的下一次重复（如还有重复的话）然后被执行。当前循环中， next 后面的语句不会被执行。

Footnotes

[1] 循环指的是对某个语句或者语句块进行循环求值。

3.2.3 repeat 语句

repeat 语句会重复对主体部分求值直到明确地要求退出。这就意味着你必须小心使用 repeat，因为可能导致死循环。 repeat 循环的语法如下

```
repeat statement
```

在使用 repeat 语句时，*statement* 必须是一个句块。另外，你需要执行一些计算并且测试语句是否会从循环里面跳出。这样做通常需要两条语句。

3.2.4 while 语句

while 语句和 repeat 语句非常的类似。 while 循环的语法如下

```
while ( statement1 ) statement2
```

其中 *statement1* 会被求值，如果它的值是 TRUE 那么 *statement2* 会被执行。这个过程重复直到 *statement1* 的值是 FALSE。如果 *statement2* 从来都没有被执行，那么 while 语句返回 NULL，否则它将会返回最后一次执行 *statement2* 所得到的值。

3.2.5 for 语句

for 循环的语法如下

```
for ( name in vector )
   statement1
```

其中 *vector* 既可以是向量也可以是列表。*vector* 里面每个元素的值都会赋给变量 *name*，然后执行 *statement1*。一个副作用是循环结束后，变量 *name* 在循环退出后仍存在，并且它的值就是 *vector* 最后一个元素的值。

3.2.6 switch 语句

技术层面上来说，`switch` 仅仅是一个函数，但是它的语义学定义和其它程序语句的控制结构类似。

它的语法如下

```
switch (statement, list)
```

其中 *list* 的元素可能有自己的名字。首先，*statement* 被求值得到结果 *value*。如果 *value* 是 1 到 *list* 长度间的一个数字，那么 *list* 对应元素将会被求值并返回结果。如果 *value* 值过大或者过小， `NULL` 将会被返回。

```
> x <- 3
> switch(x, 2+2, mean(1:10), rnorm(5))
[1]  2.2903605  2.3271663 -0.7060073  1.3622045 -0.2892720
> switch(2, 2+2, mean(1:10), rnorm(5))
[1] 5.5
> switch(6, 2+2, mean(1:10), rnorm(5))
NULL
```

如果 *value* 是字符向量，那么 ... 的元素中名字和 *value* 准确匹配的元素会被执行。如果没有匹配的，返回 `NULL`。

```
> y <- "fruit"
> switch(y, fruit = "banana", vegetable = "broccoli", meat =
"beef")
[1] "banana"
```

`switch` 常用来根据函数一个参数的字符值决定后面的执行语句。

```
> centre <- function(x, type) {
+ switch(type,
+        mean = mean(x),
+        median = median(x),
+        trimmed = mean(x, trim = .1))
+ }
> x <- rcauchy(10)
> centre(x, "mean")
[1] 0.8760325
> centre(x, "median")
[1] 0.5360891
> centre(x, "trimmed")
[1] 0.6086504
```

`switch` 返回值要么是语句的求值结果要么在没有语句被求值时的 `NULL`。

为了从一个已经存在的可选方法列表里面选择一个，`switch` 可能不是最好的实现方案。通常，用 `eval` 和 子集操作符 `[[` 直接运行 `eval(x[[condition]])` 会更好。

3.3 初等算术操作

- Recycling rules: 循环使用规则
- Propagation of names: 名字扩散
- Dimensional attributes: 维度属性
- NA handling: NA 处理

在本节，我们将会讨论用于基本操作符（如两个向量或矩阵的加法和乘法）的一些规则要点。

3.3.1 循环使用规则

如果两个元素个数不一致的数据结构相加，那么短的那个结构会循环使用以达到长的结构的长度。例如，将 `c(1, 2, 3)` 和一个 6 元素长度的向量相加，你实际操作的是 `c(1, 2, 3, 1, 2, 3)`。如果长向量的长度不是短向量的倍数，一个警告将会给出。

从 R 1.4.0 开始，任何含零长度向量的算术操作结果都是零长度向量。

一个例外是，当一个向量和矩阵相加的时候，如果长度不协调，不会给出任何警告。

3.3.2 名字扩散

名字扩散（第一个名字优先。如果它没有名字的，是不是也这样？？—— 第一个*拥有名字*的优先，循环使用导致短的结构丢失名字）。[1]

Footnotes

[1] 译者注：原句为“propagation of names (first one wins, I think - also if it has no names?? —— first one *with names* wins, recycling causes shortest to lose names)”

3.3.3 维度属性

（矩阵+矩阵，维度必须匹配。向量+矩阵：第一个重复使用，然后检验维度是否匹配，否则就报错）

3.3.4 NA 处理

统计学意义上缺失变量（值不知道的变量）的值是 NA。这和一个函数参数的 missing 特性（即一个函数的参数没有提供）不能混淆（见 Arguments）。 因为原子向量的元素必须是一样的类型，因此 NA 值有多种类型。有一种情况对用户非常重要。NA 的默认类型是 logical，除非强制转换成其它类型，因此缺失值可能会触发逻辑索引而不是数值索引（细节见 Indexing）。

含 NA 的数值和逻辑计算通常返回 NA。如果对于 NA 所有取值运算结果都一样，那么就返回这个一样的值。特别是，FALSE & NA 结果是 FALSE， TRUE | NA 结果是 TRUE。 NA 不等于任何其它值（包括自身）；测试一个对象是否为 NA 应该用 is.na。 但是，在函数 match 里面，NA 可以匹配另外一个 NA 值。

结果不明确的数值计算（如 0/0）返回结果是 NaN。这仅仅发生在实数的 double 类型或者复数的虚部中。函数 is.nan 用于检验一个对象是否是 NaN，函数 is.na 对 NaN 也返回 TRUE。 把 NaN 强制转换成逻辑型或整型将返回对应类型的 NA，但是强制转换成字符型将返回 "NaN"。NaN 是不可比较的，因此检验 NaN 是否相等的式子将返回 NA。通过 match 可以匹配 NaN 值（其它值不行，甚至是 NA）。

NA 的字符类型从 R 1.5.0 开始才和字符 "NA" 区分开。程序员如果需要指定一个外在的字符串型 NA，应该使用 as.character(NA) 而不是 "NA"，或者用 is.na<- 对 NA 设置元素。

原味型向量没有 NA 值。

3.4 索引

R 有多种构造允许通过索引操作来访问单个元素或者子集。在基本的向量类型中，可以通过 x[i] 访问第 i 个元素，但在列表，矩阵和多维数组中也有索引。除了用单个的整数进行索引，还有多种形式的索引。索引既可用于提取对象的一部分，也可用于替换对象的一部分（或者增加一部分）。

R 有三种基本的索引操作。语法如下面的例子所示

```
x[i]
x[i, j]
x[[i]]
x[[i, j]]
x$a
```

```
x$"a"
```

对于向量和矩阵，`[[`形式很少使用，尽管它和`[`在语义上稍稍有点不同（例如它去掉了所有 `names` 或 `dimnames` 属性，并且在字符索引的时候采用局部匹配）。当用单个索引处理多维结构时， `x[[i]]` 或者 `x[i]` 将会返回 `x` 的第 `i` 个元素。

对于一个列表，通常使用 `[[` 去选择任意单个的元素，而 `[` 返回所选元素的列表。

`[[` 形式允许使用整数或字符索引选出单个的元素，而 `[` 允许通过向量进行索引。注意，对于一个列表，索引可以使用向量然后向量的任何一个元素将依次用于列表，所选的分量，所选分量的分量，等等。[1] 返回结果仍然是单个元素。

`$`形式用于列表和成对列表的递归对象。它仅仅允许字面上的字符串和符号作为索引。也就是说，这种索引不可计算的：当你需要通过对一个表达式求值确定一个索引，请使用 `x[[expr]]`。当 `$`用于非递归对象时，返回结果是 `NULL`。

- Indexing by vectors: 通过向量进行索引
- Indexing matrices and arrays: 矩阵和数组的索引操作
- Indexing other structures: 其它结构的索引操作
- Subset assignment: 子集赋值

Footnotes

[1] 译者注：我测试了一下，和这里描述的不一样。原文为：“Note though that for a list, the index can be a vector and each element of the vector is applied in turn to the list, the selected component, the selected component of that component, and so on.”

3.4.1 通过向量进行索引

通过向量作为索引，R 实现了一些功能非常强大的构造。首先我们将会讨论简单向量的索引。为了简单起见，假定表达式是 `x[i]`。依据 `i` 类型的不同，有下面几种情况。

- **整数**。`i` 的所有元素必须是一样的符号。如果它们是正数， `x` 中下标和这些数字一样的元素将会被选中。 `i` 中的元素是负数，那么除这些数字对应的元素外的所有元素被选中。

  如果 `i` 是正数，并且大于 `length(x)`，那么对应的所选元素是 `NA`。 `i` 里面有超过一定范围的负数会导致一个错误。

  一个特别的例子是零索引，它不会有任何影响：`x[0]`是一个空向量并且其它含有零的正整数或负整数索引有一样的效果因为零索引会被忽略。

- **其它数值**。非整数值在作为索引前会被转换成整数（直接去掉小数部分）。
- **逻辑值**。索引 `i` 通常和 `x` 长度一致。如果它比较短，那么它的元素将会被循环使用（见 Elementary arithmetic operations）。如果它的长度过长，那么 `x` 在概念上会被 `NA` 扩展。从 `x` 中选中的元素将是对应位置上 `i` 是 `TRUE` 的元素。
- **字符**。`i` 里面的字符串和 `x` 的名字属性进行匹配，其结果整数将会被使用。在精确匹配失败后，`[[` 和 `$`都采用局部匹配，因此如果 `x` 没有一个分量的名字为 `"aa"`并且`"aabb"` 是第一个以`"aa"`作为前缀名字，那么 `x$aa` 将会匹配 `x$aabb`。但是，`[` 需要精确匹配。字符串`""`会被特殊处理：它表示`没有名字'和没有元素匹配(甚至是些没有名字的元素)。注意局部匹配仅用于提取信息，在替换操作时不适用。
- **因子**。其结果等价于 `x[as.integer(i)]`。因子水平不会被采用。如果真的需要，请使用 `x[as.character(i)]` 或者类似的构造。
- **空索引**。表达式 `x[]` 返回 `x`，但结果会扔掉“不相关”的属性，仅保留 `names` 属性和多维数组里面的 `dim` 和 `dimnames` 属性。
- **NULL**。在索引处理时，它似乎就是 `integer(0)`。

索引是缺失值 (如 `NA`)时将返回 `NA`。该规则同样用于逻辑索引，即，`x` 在使用含 `NA` 选择器的索引 `i` 时，返回结果中对应索引 `NA` 的位置上是 `NA`。 但是，需要注意的是，`NA` 有不同的模式—字面上的常量是 `"logical"`模式，但它常常自动强制转换成其它类型。这样做的后果是 `x[NA]` 长度和 `x` 一致，而 `x[c(1, NA)]` 的长度为 2。因为前者是逻辑索引，而后者是整数索引。

用 `[` 作索引操作同样也对名字属性进行相关的子集操作。

3.4.2 矩阵和数组的索引操作

多维结构的子集操作通常和将每个索引变量作一维索引的规则一样，只是 `dimnames` 分量替换了 `names`。但也有一些特殊的规则可以使用。

一般情况下，用对应维度的数字索引访问该结构。在可以忽略 `dim` 和 `dimnames` 属性或者 `c(m)[i]`的结果已经充分的情况下，依然可能用单个索引。注意 `m[1]` 常常和 `m[1, ]` 或 `m[, 1]`不同。

可以用一个整数矩阵作为索引。此时，矩阵的列数对应结构的维度，返回的结果将是一个长度和索引矩阵行数一致的向量。下面的例子展示如何一步提取元素 `m[1, 1]` 和 `m[2, 2]`的方法。

```
> m <- matrix(1:4, 2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> i <- matrix(c(1, 1, 2, 2), 2, byrow = TRUE)
> i
     [,1] [,2]
[1,]    1    1
[2,]    2    2
```

```
> m[i]
[1] 1 4
```

负索引不允许用在索引矩阵里面。但 NA 和零值是允许的：在一个索引矩阵里面，如果一行里面含有零，那么该行会被忽略，如果某一行含有 NA，那么结果对应的元素将是 NA。

无论使用单个的索引还是矩阵索引， names 属性在存在的情况下都会被使用。这里假定结构是一维的。

如果一个索引操作只想得到结构的一个区域，就像在一个三维矩阵里面用 m[2, ,]选择一个切面，则结果中对应的维度属性会被去掉。如果是一个一维结构的结果，将会得到一个向量。有时，这不是我们想要的，那么可以通过在索引操作中加入参数 drop = FALSE 来关闭。注意，这是[函数的一个额外参数，和索引计数无关。因此在一个矩阵中以 1 ×n 的形式选中第一行的正确做法是 m[1, , drop = FALSE]。没有关闭维度去除特性通常是在长度偶尔为 1 的索引里面导致失败的原因。这个规则同样可用于一维数组，其中任何子集操作都返回一个向量，除非使用 drop = FALSE。

注意，向量之所以能区分一维数组主要在于后者有 dim 和 dimnames 属性（二者都是长度为 1）。一维数组不容易通过子集操作得到但他们可以显式创建并通过 table 返回。有时，这种用法非常有用因为 dimnames 列表的元素有时候本身就被命名了。而 names 属性可能不行。

一些操作如 m[FALSE,] 会产生一个维度扩展为零（dimension has zero extent）的结构。R 一般可以敏感地处理这些结构。

3.4.3 其它结构的索引操作

操作符 [是一个泛型函数，它允许增加类方法。 $ 和 [[操作符类似。因此，用户可以对任何结构定义索引操作。现在有这样的一个函数，比如说 [.foo，被调用的时候它有一个参数集，它的第一个参数是一个被索引的结构而其它的都是索引。在使用 $ 时，索引参数是"symbol"模式，甚至在使用 x$"abc" 形式的时候。特别需要明白的是类方法没有必要和基本方法有一样的行为，例如考虑局部匹配。

[类方法的最重要的例子是用于数据框。这里不会仔细讨论这个问题（见 [.data.frame 的帮助文档），但很多时候，如果同时给出两个索引（甚至一个为空），它将为由同样长度的向量构成的列表结构创建矩阵类型的索引。如果只提供单个索引，它将会被解释为索引列表的列—这种情况下， drop 参数会被忽略，同时给出警告。

基本操作符 $ 和 [[可用于环境。此时只允许字符索引，而且不允许局部匹配。

3.4.4 子集赋值

结构的子集赋值只是复合赋值一般机制的一个特例：

```
x[3:5] <- 13:15
```

该命令的结果就像执行了下面的代码

```
`*tmp*` <- x
x <- "[<-"(`*tmp*`,3:5, value=13:15)
```

这样的机制可用于其它函数，而不仅仅是 `[`。赋值函数和前面贴的 `<-` 有一样的名字。它的最后一个名为 `value` 的参数是将被分配的新值。

```
names(x) <- c("a","b")
```

等价于

```
`*tmp*` <- x
x <- "names<-"(`*tmp*`, value=c("a","b"))
```

嵌套复合赋值的可以递归求值

```
names(x)[3] <- "Three"
```

等价于

```
`*tmp*` <- x
x <- "names<-"(`*tmp*`, value="[<-"(names(`*tmp*`), 3,
value="Three"))
```

在外围环境中的复合赋值（用`<<-`）也是允许的：

```
names(x)[3] <<- "Three"
```

等价于

```
`*tmp*` <<- get(x, envir=parent.env(), inherits=TRUE)
names(`*tmp*`)[3] <- "Three"
x <<- `*tmp*`
```

也等价于

```
`*tmp*` <- get(x,envir=parent.env(), inherits=TRUE)
x <<- "names<-"(`*tmp*`, value="[<-"(names(`*tmp*`), 3,
value="Three"))
```

在外围环境中，仅仅目标变量被求值，因此

```
e<-c(a=1,b=2)
i<-1
local({
   e <- c(A=10,B=11)
   i <-2
   e[i] <<- e[i]+1
})
```

在 LHS 和 RHS 中使用 `i` 的局部值，在超赋值语句（superassignment statement）的 RHS 中使用 `e` 的局部值。在环境外把 `e` 的值设为

```
 a  b
 1 12
```

也就是说，超赋值等价于下面三行

```
`*tmp*` <- get(x,envir=parent.env(), inherits=TRUE)
`*tmp*`[i] <- e[i]+1
x <<- `*tmp*`
```

类似的是

```
x[is.na(x)] <<- 0
```

等价于

```
`*tmp*` <- get(x,envir=parent.env(), inherits=TRUE)
`*tmp*`[is.na(x)] <- 0
x <<- `*tmp*`
```

但与下面的代码不一样

```
`*tmp*` <- get(x,envir=parent.env(), inherits=TRUE)
`*tmp*`[is.na(`*tmp*`)] <- 0
x <<- `*tmp*`
```

这两种解释的差异仅仅在于是否有一个局部变量 `x`。应该尽量避免一个局部变量名字和一个超赋值的目标变量名字一样。因为这种情况在 1.9.1 版本或者之前的版本不能正常处理，所以没有必要采用这些代码。

3.5 变量作用域

几乎所有的编程语言都有一套作用域规则，允许同样的名字用于不同的对象。这使得一个函数内的局部变量可以有同全局变量一样的名字。

R 采用 Pascal 类似的*词法作用域*（lexical scoping）模式。但是 R 是一种*函数型编程语言*（functional programming language），允许动态创建和操作函数与语言对象，并且增加了一些特性来反映这些东西。

- Global environment: 全局环境
- Lexical environment: 词法环境
- Stacks: 堆栈
- Search path: 搜索路径

3.5.1 全局环境

全局 环境是用户工作空间的根部。一个 命令行上的赋值操作会导致全局环境中相应对象的改变。它的外围环境在搜索路径里面是下一个环境，如此直到退回到基础环境外围的空环境。

3.5.2 词法环境

对任何函数的调用 都会创建一个*框架*（frame）。该框架包括函数中创建的局部变量，以及在一个环境中求值时组合创建的新环境。

注意以下术语：框架指的是一组变量的集合，环境则是一组框架的嵌套（或者说：内部框架加上外围环境）。

环境可以赋给变量或包含在其它对象里面。但是，没有标准的对象 — 特别是，它们在赋值的时候不会进行拷贝。

闭包（closure；`"function"`模式）对象含定义时就作为它的一部分的环境（默认情况下，环境可以用 `environment<-` 来操作）。当函数随后被调用时，它的求值环境是以闭包的环境作为外围创建。 注意，对于调用者的环境没有这样的必要。

因此，当需要一个函数中的变量时， 首先在求值环境中搜索， 然后再外围，外围的外围，等等；一旦达到全局环境或一个包的环境，搜索路径会继续延伸到基本包的环境。如果仍然没有发现对象，搜索会随后在空环境中进行，并且会失败。

3.5.3 调用堆栈

当函数被调用时， 一个新的求值框架会被创建。在程序执行的任何时刻，通过*调用堆栈*（call stack）可以访问当前激活环境。每当一个函数被调用时，一个被称为上下文（context）的特殊结构会在内部创建并存放在一个上下文的列表里面。当一个函数完成求值，它的上下文会从调用堆栈里面去除。

变量定义高于可以得到的调用堆栈时称为动态作用域。 一个变量的绑定由变量的最近定义决定。这和 R 里面默认的作用域规则相违背。R 里面默认的规则是变量绑定在函数定义的环境中（词法作用域）。 一些函数，特别是使用和操作模型公式的函数，需要通过直接访问调用堆栈来模拟动态作用域。

通过 函数名以 sys. 开头的一族函数来访问调用堆栈。现将它们简单列举如下。

`sys.call`

获得特定上下文的调用。

`sys.frame`

获得特定上下文的求值框架。

`sys.nframe`

获得所有被激活的上下文的环境框架。

`sys.function`

获得在特定上下文中被调用的函数。

`sys.parent`

获得当前函数调用的父节点。

`sys.calls`

获得所有激活的上下文的调用。

`sys.frames`

获得所有被激活的上下文的求值框架。

`sys.parents`

获得所有被激活的上下文的数值标签。

`sys.on.exit`

设置一个特定上下文退出时执行的函数。

`sys.status`

调用 `sys.frames`，`sys.parents` 和 `sys.calls`。

`parent.frame`

获得特定父上下文的求值框架。

3.5.4 搜索路径

除了求值环境结构， R 还有一个用来搜索其它地方找不到的变量的环境搜索路径。它主要用于两件事情：函数的包和被用户邦定的数据。

搜索路径的第一个元素是全局环境，最后一个是基础包。 `Autoloads` 环境用于保存在需要时可能载入的代理对象（proxy objects）。其它环境通过 `attach` 和 `library` 插入。

含有*命名空间*（namespace）的包有不同的搜索路径。当一个 R 对象从这种包里面的一个对象开始搜索时，该包首先被搜索，然后是它的引用，随后是基础命名空间，最后是全局环境和其它正规搜索路径的其它部分。这样做的后果是，在同一包里面其它对象的引用将会在这个包里解决，并且里面的对象不能被全局环境或其它包里面同名的对象屏蔽。

---

4 函数

- Writing functions: 编写函数
- Functions as objects: 函数作为对象
- Evaluation: 求值

4.1 编写函数

虽然 R 是一个非常有用的数据分析工具，但大多数用户很快就发现他们想编写自己的函数。 这是 R 的一个重要优势。用户可以编程而且他们可以在需要的时候把系统层次的函数改变为他们认为更为恰当的函数。

R 还提供了工具去记录你创建的函数的文档。 See Writing R documentation (Writing R Extensions).

- Syntax and examples: 语法和例子
- Arguments: 参数

4.1.1 语法和例子

编写函数的语法如下

```
function ( arglist ) body
```

函数声明的第一个分量就是关键字 `function`。它告诉 R 你想创建一个函数。

参数列表是以逗号分割的形式参数列表。 形式参数可以是符号， *symbol* = *expression* 形式的语句，或者特殊形式参数 ...。

*函数主体*可以是任何合法的 R 表达式。通常，函数主体是一组由大括弧 ({ 和 }) 括起来的表达式。

一般情况下， 函数赋给一个符号，但这不是必要的。调用 `function` 所返回的值就是函数。如果它没有命名，那么它就是一个匿名函数。 匿名函数常常作为参数用于其它函数，如 `apply` 函数族或者 `outer`。

这里是一个简单的例子： `echo <- function(x) print(x)`。因此 `echo` 是一个单参数的函数，当 `echo` 被调用时，它会打印它的参数。

4.1.2 参数

函数的形式参数定义了函数被调用时哪些变量的值需要提供。这些参数的名字可用于函数的主体而且函数的主体在函数调用时可以通过这些名字得到外部所赋的值。

参数默认值可以用 *name* = *expression* 形式指定。在这种情况下，如果用户不给定一个参数的值，则在函数调用时，默认设定表达式会和对应的符号关联。当需要一个值时，*expression* 会在函数的求值框架下求值。 用函数 `missing` 也能设定默认的行为。当根据形式参数的名字调用函数 `missing` 时， 在形式参数不匹配任何实际参数以及在函数主体内没有进行任何修改时返回 `TRUE`。如果需要的话，`missing` 的参数可以有它自己的默认值。 `missing` 函数不会强制参数求值。

特别类型参数 ... 可以包含任意数目的参数。它有各种各样的用途。它允许你编写一个含有任意多参数的函数。 它还可来吸收一些参数传递给一个中间函数，该函数能被其它随后调用的函数提取。

4.2 对象函数

函数是 R 里面第一个类对象。它们可用于任何需要 R 对象的地方。特别的是，他们可以作为参数传给一个函数并且通过函数返回结果值。详见 Function objects 部分。

4.3 求值

- Evaluation environment: 求值环境

- Argument matching: 参数匹配
- Argument evaluation: 参数求值
- Scope: 作用域

4.3.1 求值环境

当一个函数被调用时， 一个新的求值框架会被创建。 在这个框架中，形式参数通过 Argument matching 部分给定的规则匹配给定的参数。函数主体里面的语句随后基于本环境框架求值。 求值框架的外围框架是和被调用的函数关联的环境框架。这可能和 S 不一样。虽然许多函数以 `.GlobalEnv` 作为它们的环境，但事实上不总是这样的，因为一些拥有命名空间的包里面定义的函数通常以包的命名空间作为它们的环境。

4.3.2 参数匹配

函数求值里面发生的第一件事就是 将形式参数和实际的或提供的参数匹配。这由三步完成：

1. **标签的精确匹配**。 对于任何一个有名字的给定参数，形式参数列表会用来仔细搜索对应的名字精确匹配的参数。一个形式参数对多个实际参数是不允许的，反之也然。
2. **标签的局部匹配**。任一留下的有名字的给定参数将和余下的形式参数进行局部匹配。如果给定参数的名字和形式参数的前面部分精确吻合，那么这两个参数看作是相互匹配的。多重匹配是不允许的。注意，如果 `f <- function(fumble, fooey) fbody`，那么 `f(f = 1, fo = 2)` 是违法的，尽管第二个参数仅匹配 `fooey`。而 `f(f = 1, fooey = 2)` *是*合法的，因为第二个参数精确匹配会在考虑局部匹配前去掉了。如果形式参数包括 ...，那么局部匹配仅仅用于优先于它的参数。
3. **位置匹配**。任何没有匹配上的形式参数依次和*没有命名*的给定参数比对。如果有一个... 参数，它将吸收所有余下的参数，无论它加了标签还是没有标签。

如果还有没被匹配上的参数，那么一个错误将会被声明。

参数匹配可以用函数 `match.arg`， `match.call` 和 `match.fun` 来实现扩展。 可以通过 `pmatch` 访问 R 的局部匹配算法。

4.3.3 参数求值

为了了解函数参数的求值情况，知道给定参数和默认参数的不同处理方式是非常重要的。 函数的给定参数在调用函数的求值框架下求值。函数的默认参数在函数的求值框架下求值。

语义上，R 里面调用一个函数，参数是 *值传递*（call-by-value）。通常，给定参数的行为似乎它们是用给定值和对应的形式参数名字初始化后的局部变量。 改变一个函数里面给定参数的值不会影响调用框架里面参数的值。

R 有一种函数参数的悠闲求值方式。参数只有在需要的时候才求值。需要知道的是在一些情况下，这些参数永远不会被求值。因此，在函数中使用参数有时会有副作用。虽然在 **C** 里面，`foo(x = y)`常用来调用 `foo`，其中用到 `y` 并同时把 `y` 赋给 `x`，不过这种方式最好不要用于 R。这里不能保证参数会被求值， 该赋值操作可能不会发生。

值得提醒的是 `foo(x <- y)`的影响。如果该参数被求值，它将会改变调用环境中的 `x` 值，而不是 `foo` 的求值空间。

可以访问事实上的（不是默认的）函数内部作为参数的表达式。这个机制是通过允诺来实现。当一个函数被求值，实际上作为参数的表达式 和一个指向函数调用来源环境的指针一起存储在允诺中。如果参数被求值，函数调用来源环境中存储的表达式将会被求值。既然只有一个指向环境的指针，任何对环境进行的修改在求值过程中都是有效的。结果值随后保存在允诺里面一个单独的地方。随后的求值可以取得该值（不会进行再次求值）。用 `substitute` 可以访问没有求值的表达式。

当一个函数被调用时， 每个形式参数都会被分配到一个相关调用的局部环境中的允诺里面。该允诺包括含有实际参数（如果存在）的表达式槽和含有调用者环境的环境槽。如果没有实际参数对应调用中给定的形式参数以及没有默认的表达式，这就类似被分配到一个形式参数的表达式槽，但拥有局部环境的环境集合。 通过对允诺环境中表达式槽的上下文求值来提供一个允诺值槽的过程 称之为*强制*（forcing）允诺。允诺只有后面在值槽上下文被直接使用时是被强制的。

一个允诺在需要它的值时是强制的。这常常发生在内部函数中， 但可以通过直接对允诺自身求值来强制允诺。这个有时会非常有用，如在默认的表达式倚赖另外一个形式参数的值或局部环境的其它变量时。这可以在下面的例子中看到，孤立的 `label` 保证了它在下一行中改变前标签依赖于 `x` 的值。

```
function(x, label = deparse(x)) {
    label
    x <- x + 1
    print(label)
}
```

一个允诺的表达式槽自身可以包含其它允诺。这在一个未求值的参数作为参数传给另外一个函数的时候发生。当强制一个允诺的时候，其表达式中的其它允诺会被递归强制，就好像它们被求值一样。

4.3.4 作用域

作用域或作用域规则简单说就是求值程序为一个符号寻找值所用规则的集合。每种计算机语言都有这样的一套规则。在 R 里面，这个规则非常的简单，但也确实存在一些机制搅乱常见的或默认的规则。

R 沿用一套称之为*词法作用域的*规则。这表明在表达式创建时变量的 有效绑定可用来为表达式中的自由符号提供值。

作用域的许多有趣的性质 都和函数的求值有关， 现在我们集中描述这个问题。一个符号要么是 有约束的要么是自由的。一个函数的所有形式参数在函数主体中提供了被约束的符号。任何其它在函数主体里面的符号要么是局部变量要么是自由变量。局部变量是在函数中定义的变量。因为 R 没有变量的形式定义，它们只在需要的时候才使用，这样就很难区分一个变量是否是局部变量。局部变量首先需要定义，典型的做法是让它们处在一个赋值操作的左边。 在求值过程中，如果发现一个自由变量，那么 R 会去给它找一个值。作用域规则决定了这个过程如何进行。在 R 里面， 函数的环境首先会被搜索，然后是它的外围，如此直到全局环境。

全局环境指向一个为某个匹配符号逐步搜索的环境的搜索列表。第一个匹配上的值会被采用。

当这些规则结合函数能以值的形式从其它函数返回的事实， 感觉确实不错。但是首先，你必须获得所有特性。[1]

一个简单的例子是，

```
f <- function(x) {
    y <- 10
    g <- function(x) x + y
    return(g)
}
h <- f()
h(3)
```

一个非常有意思的问题是，当 `h` 被求值，什么会发生呢？为了描述这个，我们需要一些的新的考虑。在函数主体里面，变量可以是被约束的，局部的或者自由的。 被约束的变量是这些匹配函数形式参数的变量。局部变量指的是那些在函数主体内创建和定义的变量。自由变量指的是那些既不是局部也不是被约束的变量。当一个函数主体被求值，确定一个局部或被约束变量的值没有问题。作用域规则决定了一个语言如何为自由变量找恰当的值。

当 `h(3)` 被求值，我们发现它的主体就是 `g` 的主体。在那个主体中，`x` 和 `y` 都是自由的。在一个词法作用域定义的语言中， `x` 的值和 3 关联而 `y` 和 10 关联，因此 `h()` 返回值 13。在 R 里面，事实上就是这样发生的。

在 S 里面，因为不同的作用域规则，上面的例子可能会报错并且提示 `y` 找不到。此时除非在你的工作空间中有一个变量 `y`，那么这个值将会被使用。

Footnotes

[1] 译者注：原文为“When this set of rules is combined with the fact that functions can be returned as values from other functions then some rather nice, but at first glance peculiar, properties obtain.”

5 面向对象编程

面向对象编程是一种近年来非常流行的编程方式。它的流行主要来之一个事实就是它让编写和维护一个复杂系统变得比较容易。它通过几种不同的机制来实现这个目标的。

任何面向对象语言的中心都是类（class）和方法（method）这两个概念。类是一个对象的定义。通常一个类包含了多个用于保存类特有信息的*槽变量*（slots）[1]。编程语言中的对象指的是类的具体实例。编程就是基于这些类的对象或实例。

计算过程是通过*方法*（methods）实现。方法主要是 用于实现在某个类的对象上进行专门计算的函数。这就是为什么使得语言面向对象了。在 R 里面，由*泛型函数*（generic functions）来决定适当的方法。泛型函数负责判断它的参数的类并且用这些信息选择恰当的方法。

大多数面向对象语言的另外一个特性是“继承”这个概念。在大多数编程问题中，许多对象通常是相互关联。如果一些东西可以重用，编程会变得比较简单。

如果一个类从另外一个类继承而来，它常常会从父类得到所有的槽变量，而且还可以增加新的槽变量。在方法分发的时候（通过泛型函数），如果一个类的方法不存在，那么就会在它的父类里面搜索。

在本章，我们会讨论这种通用的策略如何在 R 里面已经实现，并且讨论当前设计中一些局限性。大多数对象系统都会体现的一个优点是它有很好的一致性（consistency）。这通过编译器或解释器按照一定的规则检查实现。不幸的是，因为 R 和对象系统融为一体，这个优势不能完全体现。用户在一种直接的方式下面使用对象系统必须十分的小心。虽然可以展示一些非常有趣功能，但这些倾向导致模糊的代码和可能依赖一些不会再被采用的实现细节。

R 里面最广泛的面向对象编程运用是 `print` 方法，`summary` 方法和 `plot` 方法的使用。这些方法允许我们只要有一个泛型函数调用，比如说 `plot`，它会基于它的参数类型分发方法并且针对特定的数据调用相应的绘图函数。

为了使概念更清晰，我们将考虑一个概率论教学用的小系统的实现。在这个系统中，对象是概率函数和计算矩及绘图的方法。概率常常可以通过累计分布函

数来描述，但也可以通过其它方法实现。例如，一个密度函数，它可以单独存在，也可以作为一个矩发生函数存在。

- Definition: 定义
- Inheritance: 继承
- Method dispatching: 方法分发
- UseMethod: UseMethod 函数
- NextMethod: NextMethod 函数
- Group methods: 成组方法
- Writing methods: 编写方法

Footnotes

[1] 译者注：类似“成员变量”，用@符号访问

5.1 定义

和一般的完整面向对象系统不一样的是， R 有一个类系统和基于类对象的分发机制。为解释型代码设置的分发机制依赖于求值框架下的四个特殊对象。它们分别是 `.Generic`， `.Class`， `.Method` 和 `.Group`。还有一个用于内部函数和类型的独立分发机制，但不是这里讨论的内容。

`class` 属性使类系统变得更加容易。该属性是类名字的列表。为创建类`"foo"`的一个对象，可以简单地把字符串"foo"赋给一个对象的类属性。事实上，任何东西都可成为类 `"foo"` 的一个对象。

对象系统通过两个分发函数，`UseMethod` 和 `NextMethod`，来使用 *泛型函数*（generic functions）。对象系统的典型应用是从调用泛型函数开始的。通常，函数非常的简单，只有一行代码。系统函数 `mean` 就是这样的一个函数，

```
> mean
function (x, ...)
UseMethod("mean")
```

在调用 `mean` 的时候，可以给定任意多的参数，但是它的第一个参数是特定的并且第一个参数的类用来决定调用那一种方法。变量 `.Class` 设为 `x` 的类属性，`.Generic` 设为字符串 `"mean"`，然后寻找正确的方法来执行。`mean` 的任何其它参数的类属性都会被忽略。

假定 `x` 有一个依次含有 `"foo"` 和`"bar"` 的类属性列表。 R 可能首先搜索名为 `mean.foo` 的函数，如果该函数找不到，它可能随后去找名为 `mean.bar` 的函数，此时如果还不能成功，则最后寻找函数 `mean.default`。如果最后的搜索仍不能

成功， R 将会报错。因此，记得编写默认的方法是一个好习惯。注意，这里提到的 `mean.foo` 等函数指的是方法。

`NextMethod` 还提供里另外一种分发机制。 函数可以在其内部任何地方调用 `NextMethod`。决定哪个方法会被调用主要基于 `.Class` 和 `.Generic` 的当前值。这里稍稍有点问题，因为方法实际上就是普通的函数，那么用户就可能直接调用它。如果他们这样做，那 `.Generic` 或 `.Class` 就没有赋值了。

如果一个方法被直接调用，它就包括一个对 `NextMethod` 的调用，并且 `NextMethod` 的第一个参数用于决定泛型函数。 如果这个参数没有提供，系统会报错；因此最好记得提供该参数。

在方法直接调用的时候，方法的第一个参数的类属性作为 `.Class` 的值。

方法自身利用 `NextMethod` 提供继承形态。通常，一个特定方法用一些操作启动数据然后根据 `NextMethod` 调用下一个适当的方法。

考虑一下下面这个简单的例子。二维欧氏空间里面的一个点可以通过笛卡尔坐标或极坐标（r-θ）确定。因此，为存储点的位置信息，我们可以定义两个类，`"xypoint"` 和 `"rthetapoint"`。 所有 `xypoint' 数据结构由一个 x-分量和一个 y-分量构成的列表组成。所有 `rthetapoint' 对象由一个 r-分量和一个 theta-分量构成的列表组成。

现在，假定我们想得到任一对象的 x-位置坐标。通过泛型函数，这非常容易实现。 我们如下定义了一个泛型函数 `xpos`。

```
xpos <- function(x, ...)
    UseMethod("xpos")
```

现在我们可以定义方法：

```
xpos.xypoint <- function(x) x$x
xpos.rthetapoint <- function(x) x$r * cos(x$theta)
```

用户可以简单地用任一描述方法作为参数调用函数 `xpos`。内在的分发方法判断对象的类，然后调用恰当的方法。

这使得非常容易加入其它坐标表述方式。我们没有必要写一个新的泛型函数，我们只要写新的方法就可以了。因此，对一个已经存在的系统非常容易扩展的，因为用户只负责处理新的表述方式而不用考虑已有的表述方式。

使用这种思路的一个例子是为不同类型的对象提供专门的打印输出；现在已经为 `print` 设计了 40 种方法。

5.2 继承

一个对象的类属性可以有多个元素。 在一个泛型函数调用时，最初的继承主要是通过 NextMethod 处理。 NextMethod 决定了当前求值的方法，并搜寻下一个类。

注意：这里原文有内容缺失。

5.3 方法分发

泛型函数的语句比较单一。它们常常以 `foo <- function(x, ...) UseMethod("foo", x)` 的形式出现。 UseMethod 的语义是决定一个恰当的方法，然后用同样的参数以调用泛型时一样的参数顺序调用该方法，就好像该调用直接对方法进行操作。

为了确定正确的方法，泛型的第一个参数的类属性会被用于搜寻正确方法。 泛型函数的名字和第一个参数的类属性组成 *generic*.*class* 的形式，然后以此为函数名进行搜索。如果函数找到，则它将会被使用。如果不能找到这样的函数，类属性的第二个元素就会被使用，如此直到所有类属性的元素被用光。如果仍然没有方法被发现，方法 *generic*.*default* 将会被使用。如果泛型函数的第一个参数没有类属性，也是调用 *generic*.*default*。既然引入了命名空间的概念，方法可能不能直接通过名字访问（即 `get("generic.class")` 会失败），但它们可以通过命令 `getS3method("generic","class")` 访问。

任何对象都可以有 class 属性。该属性可以有任何数目的元素。这些元素是定义一个类的字符串。当一个泛型函数被调用时，首先检查它的第一个参数的类。

5.4 UseMethod 函数

UseMethod 是一个比较特殊的函数，它的行为和其它的函数调用有点差异。该函数的调用语法是 UseMethod(*generic*, *object*)，其中 *generic* 是泛型函数的名字， *object* 用于决定哪一个方法将会被采用的对象。 UseMethod 仅可在函数内部被调用。 S 定义里面的 UseMethod 含有一个额外的参数 ...，该参数可能会影响随后被调用方法的一些默认值。 R 在 UseMethod 有多于两个参数时给出警告，并且忽略多余的参数。

UseMethod 通过两种方式改变求值模式。第一种方式是，当它被调用时，它决定下一个被调用的方法（函数），然后利用当前求值环境调用该函数； 这个过程将会被简单描述。第二种方式是，UseMethod 会改变求值环境，但不会返回对函数调用的控制。这就意味着，任何在 UseMethod 调用后面出现的语句肯定不会被执行。

当 UseMethod 被调用时，泛型函数是调用 UseMethod 时的指定值。准备分发的对象要么是当前函数的第一个参数要么是给定的第二个参数。参数的类可以被确定，它的一个元素结合泛型函数的名字可用来确定适当的方法。因此，如果泛型的名字是 foo 而对象的类是 "bar"，那么 R 将会搜索一个名为 foo.bar 的方法。如果不存在这样的方法，那么上面描述的继承机制可用来查找适当的方法。

一旦一个方法已经被确定，R 能以一种特别的方式调用它。和创建新的求值环境不同的是， R 使用当前函数调用（泛型的调用）的环境。任何在调用 UseMethod 前的赋值或求值都将有效。 用于调用泛型函数的参数会再次匹配所选方法的形式参数。

当一个方法被调用时，所使用的参数在参数数目和名字上和调用泛型时采用的都是一样的。利用 R 里面标准的参数匹配规则，这些参数会和所调用方法的参数比较。但第一个参数会被求值。

调用 UseMethod 的一个后果是把一些特殊对象放在求值框架中。它们是 .Class， .Generic 和 .Method。这些特殊对象被 R 用来处理方法分发和继承。 .Class 是对象的类， .Generic 是泛型函数的名字以及 .Method 是当前调用的方法的名字。如果方法通过内部接口调用，可能还会有一个称为.Group 的对象。这些将在 Group methods 部分描述。在初始调用 UseMethod 后，这些特别的变量，不是对象自身，控制了随后方法的选择。

方法的主体然后以标准的方式求值。特别是，主体中的变量搜索方式遵循为方法设置的规则。因此如果一个方法有相关的环境，那么该环境将会被使用。结果是，我们用方法调用代替了泛型调用。泛型框架下的局部赋值会传递给方法调用。 不过，这种*特性*的使用一般不被推崇。需要了解的是，控制不会返回给泛型因此调用 UseMethod 后的任何表达式都不会被执行。

泛型的任何可求值参数在调用 UseMethod 前保留其可求值特性。

泛型调用过程中的参数会使用标准的参数匹配机制在方法参数中再次匹配。第一个参数（也是一个对象）将被求值。

如果没有提供 UseMethod 的第一个参数，则假定是当前函数的名字。如果 UseMethod 有两个参数，则第一个是方法名，第二个假定是方法分发时参考的对象。它将会被求值使得可以确定必须的方法。在这种情况下，泛型调用的第一个参数不会被求值并会被抛弃。没有办法改变方法调用中的其它参数，它们似乎仍在泛型调用中。这和另外一种方法调用中的参数可以改变的 NextMethod 函数形成鲜明对比。

5.5 NextMethod 函数

NextMethod 用于提供简单的继承机制。

调用 NextMethod 产生的方法调用在行为上似乎它们被前面的方法调用。参数以当前调用方法一样的次序和名字传递给后面继承的方法。这意味着它们和泛型调用一样。但是作为参数的表达式是当前方法对应的形式参数名字。因此，在 NextMethod 被调用时，参数已经有它们的对应值了。

未求值的参数仍未求值。缺失的参数仍然缺失。

调用 NextMethod 的语法是 NextMethod(generic, object, ...)。如果没有提供 generic， .Generic 的值将会被使用。如果没有提供 object，当前方法调用的第一个参数将会被使用。... 参数的值用于修改下一个方法的参数。

需要注意的是，下一个方法的选择依赖于 .Generic 和 .Class 的当前值，而不是对象。因此改变 NextMethod 调用中的对象影响下一个方法接受的参数但不影响下一个方法的选择。

方法可以直接调用。如果那样，就不会有 .Generic， .Class 或 .Method。这种情况下， NextMethod 的 generic 参数必须指定。.Class 的值设为当前函数的第一个参数对象的类属性。 .Method 的值是当前函数的名字。默认值的选择保证了一个方法无论在直接调用还是通过泛型调用的情况下行为不会改变。

一个值得讨论的问题是 NextMethod 里面...参数的行为。白皮书（White Book）描述该行为如下：

- 在调用当前方法时用命名参数代替对应的参数。未命名的参数放在参数列表的起始位置。

我们想做的是：

-首先为 NextMethod 进行参数匹配； -如果对象或泛型匹配不错 -首先，如果一个命名的列表元素匹配一个参数（命名的或者没有名字的）， 列表值代替参数值 - 第一个未命名的列表元素

搜索用的值： 类：首先来自 .Class，其次来自方法的第一个参数，最后来自调用 NextMethod 时设定的对象

泛型：首先来自 .Generic，如果没有的话，则来自方法的第一个参数， 如果仍然缺失，则来自 NextMethod 的调用

方法：这可能是当前函数的名字。

5.6 成组方法

对多种内置函数的类型， R 为操作符提供了一个分发机制。这表明操作符如 == 或 < 对特定类的成员可以设定特定行为。函数和操作符已经分成三类。对任何一组分类可以编写成组函数。现在还没有增加组的机制，但可以为一个组内的任一函数编写特定的方法。

下表中列出了不同组（Groups）的函数。

数学

> abs, acos, acosh, asin, asinh, atan, atanh, ceiling, cos, cosh, cumsum, exp, floor, gamma, lgamma, log, log10, round, signif, sin, sinh, tan, tanh, trunc

汇总

> all, any, max, min, prod, range, sum

操作符

> +, -, *, /, ^, < , >, <=, >=, !=, ==, %%, %/%, &, |, !

对于操作符组的操作符，如果两个放在一起的操作数暗示着一个方法，则一个特定的方法会被调用。特别是，两个操作数对应相同的方法，或者一个操作数对应的方法优先级高于另外一个操作数对应的方法。如果它们没有暗示单独的一个方法，那么默认的方法将会被采用。如果另一个操作数没有对应的方法，则成组方法或类方法起支配地位。类方法支配成组方法。

当组是操作符组时，特殊变量 `.Method` 是一个双元素的字符向量。如果对应参数是决定方法的类的一个成员，`.Method` 的元素设为方法的名字。否则，`.Method` 的对应元素设为长度为 0 的字符串""。

5.7 编写方法

用户可以很方便地编写它们自己的方法和泛型函数。 泛型函数可以简单地看作调用 `UseMethod` 的函数。而方法则是通过方法分发的函数。这可以通过调用 `UseMethod` 或 `NextMethod` 实现。

值得记住的是方法可以直接调用。这就意味着它们可以在不调用 `UseMethod` 的情况下直接键入，这会导致特殊变量`.Generic`， `.Class` 和 `.Method` 不会被实例化。在这种情况下，前面讨论的默认规则将用于确定这些值。

泛型函数最常用于为统计对象，一些模型拟合过程中产生的结果通常会提供 `print` 和 `summary` 方法。 为了这样做，每个模型赋予其输出结果一个类属性，然后提供特定的方法处理结果并且给出它的易读版本（信息展示方式比较友好的方式）。用户只需要记住 `print` 或 `summary` 会提供任何分析结果的友好输出。

6 语言上的计算

R 属于一类编程语言。在该类编程语言中，子程序有能力修改或构建其它子程序，求值并且把结果值作为语言自身的一部分。这和 Lisp， Scheme 以及其它“函数编程”（functional programming）语言的变种有点类似，但和 FORTRAN 以及 ALGOL 类的编程语言形成鲜明对比。Lisp 类的语言通过“所有东西都是列表”（everything is a list）这个理念让这个特性发挥到了极致，以致于不能区分程序和数据。

R 提供了比 Lisp 更友好的编程接口。对于用户来说，它至少使用了数学公式和 C 类似的控制结构，但是其引擎事实上和 Lisp 非常类似的。R 允许直接访问 解析后的表达式和函数，并且允许你更改和随后执行它们，或者从草稿中创建全新的函数。

有很多标准的应用采用了这种便利，如表达式的分析计算或通过系数向量产生多项式函数。但是，也有一些非常基础的应用用于 R 里面解释部分的运行。其中的一些应用是作为一个函数在另外一个函数里面作为一部分重用的本质部分，因为调用的 `model.frame` 创建于多个模建和绘图程序（诚然，这样做有点不太漂亮）。其它一些应用简单地允许某些有用函数的优秀接口。如 `curve` 函数，它允许你绘制一个给定函数的表达式（如 `sin(x)`）的图形或者便于绘制数学表达式的图形。

在本章，我们会介绍可以得到的用于语言上的计算的工具集合。

- Direct manipulation of language objects: 直接操作语言对象
- Substitutions: 替换
- More on evaluation: 求值之外的东西
- Evaluation of expression objects: 表达式对象的求值
- Manipulation of function calls: 函数调用的操作
- Manipulation of functions: 函数操作

6.1 直接操作语言对象

有三种语言对象可用于修改，调用，表达式和函数。基于这种认识，我们集中讨论调用对象。就是有时称之为 “未求值表达式”（unevaluated expressions）的对象，尽管这个术语有点让人困惑。获得一个调用对象最直接的方法是把一个表达式参数赋给 `quote` 函数，如

```
> e1 <- quote(2 + 2)
> e2 <- quote(plot(x, y))
```

参数没有被求值，结果简单地解析为参数。对象 e1 和 e2 可以随后用 eval 求值，或简单地当作数据。这就很明显为什么 e2 对象的模式是 "call"，因为它利用一些参数调用了函数 plot。但是 e1 实际上有和调用双参数的二元操作符+完全一样的结构。这个事实可以通过下面的例子更清晰地展示

```
> quote("+"(2, 2))
2 + 2
```

调用对象的分量可以通过列表类似的语法访问，并且可能用 as.list 和 as.call 与列表进行相互转换

```
> e2[[1]]
plot
> e2[[2]]
x
> e2[[3]]
y
```

在使用关键字参数匹配时，关键字可以作为列表的标签：

```
> e3 <- quote(plot(x = age, y = weight))
> e3$x
age
> e3$y
weight
```

在前面的例子中，调用对象的所有分量的模式是 "name"。对于调用对象里面的标识符确实是这样，但是一个调用对象的分量还可以是任意类型的常量，尽管在该调用随后被成功执行时第一个分量最好是一个函数—或者是其它调用对象以对应的子表达式。模式名的对象可以用 as.name 通过字符串创建，因此我们可以如下修改 e2 对象

```
> e2[[1]] <- as.name("+")
> e2
x + y
```

为说明子表达式是自身调用的简单分量，考虑下面的例子

```
> e1[[2]] <- e2
> e1
x + y + 2
```

所有输入的成组括号在解析后的表达式中是受保护的。它们以单参数函数调用的方式展示，因此 4 - (2 - 2) 在前缀方式中变成 "-"(4, "(" ("-"(2, 2)))。在求值过程中，(操作符仅仅返回它的参数。

这有点不幸，但是编写一个同时保护用户输入，以最简形式保存并且保证解析一个语法分析过的表达式时可以返回一样的表达式的解析器/语法分析器不是一件容易的事情。 因为很难做到，所以 R 的解析器不可以很好的可逆，同样它的语法分析器也不行，如下面的例子所示

```
> deparse(quote(c(1, 2)))
[1] "c(1, 2)"
> deparse(1:2)
[1] "c(1, 2)"
> quote("-"(2, 2))
2 - 2
> quote(2 - 2)
2 - 2
```

但是语法分析过的表达式可以得到和原始表达式一样的求值结果（直到精度错误）。

...流控制结构的内部存储...注意和 Splus 不兼容...

6.2 替换

事实上，向上节内容中提到的一样，修改一个表达式内部结构是很少见的。最常见的是，用户简单地想得到一个表达式以分析它并且用它来作标记图形一类的事情。这样的一个例子可见于 `plot.default` 实现代码的起始部分：

```
xlabel <- if (!missing(x))
    deparse(substitute(x))
```

这使得 `plot` 的 `x` 参数变量或表达式随后可用来标记 x-轴。

实现这一要求的是函数 `substitute`。它获得表达式 `x` 并且替换通过形式参数 `x` 传递的表达式。注意，为了保证这样运行， `x` 必须拥有产生它的值的表达式的信息。这和 R 的悠闲求值架构有关（see Promise objects）。 一个形式参数事实上是一个*允诺*，该对象有三个槽变量，一个用于定义它的表达式，一个用于表达式求值的环境，还有一个用于表达式的求值结果。 `substitute` 识别允诺变量并且替换它的表达式槽变量的值。如果 `substitute` 在一个函数内部被调用，该函数的局部变量也受替换支配。

`substitute` 的参数没有必要是一个简单的标识符，它可以是一个含有多个变量的表达式。此时，任何一个变量都会发生替换。同样，`substitute` 有一个额外的参数，它是一个变量可以搜索的环境或者列表。例如：

```
> substitute(a + b, list(a = 1, b = quote(x)))
1 + x
```

注意，引用（quoting）是 `x` 替换所必需的。这种构造方便在图中增加数学表达式，如下面的代码所示

```
> plot(0)
> for (i in 1:4)
+   text(1, 0.2 * i,
+        substitute(x[ix] == y, list(ix = i, y = pnorm(i))))
```

值得注意的是替换是纯词法上实现的；如果它们被求值了，则不会对结果调用对象的意义进行检验。 `substitute(x <- x + 1, list(x = 2))` 会恰当地返回 `2 <- 2 + 1`。但是， R 的有些部分自己定义了什么是有意义和什么无意义的规则，而且事实上就是采用了这些形式上有问题的表达式。例如，使用“图中数学”的特性时常常会有语法上正确，但求值毫无意义的构造，如 {}>=40*" years"。

替换不会对第一个参数求值。这导致如何替换包含在一个变量中的对象的问题。解决问题的方法是再用一次 `substitute` ，如下所示

```
> expr <- quote(x + y)
> substitute(substitute(e, list(x = 3)), list(e = expr))
substitute(x + y, list(x = 3))
> eval(substitute(substitute(e, list(x = 3)), list(e = expr)))
3 + y
```

替换的精确规则如下：第一个参数的解析树的每个符号 和第二个参数匹配，既可以是有标签的列表也可以是环境框架。如果它是一个简单的局部对象，它的值将被插入， *除非*它匹配全局变量。如果它是一个允诺（常常是函数参数），允诺表达式会被替换。如果符号没有被匹配，它不会有任何改变。而在最高层次的替换很少有例外的。[1] 这是从 S 继承而来，原理基本上是变量可能在那个层次上绑定使得替换最好和 `quote` 类似而且没有控制。[2]

如果局部变量在 `substitute` 使用前被替换，允诺替换的规则和 S 相应的规则稍稍有点不同。 R 将使用变量的新值，而 S 将无条件地使用参数表达式 — 除非它是一个常量。这导致一个很古怪的结果，即在 S 里面 `f((1))` 可能和 `f(1)` 差异很大。但 R 的规则相当地清晰，尽管它也有一些比较奇怪的和悠闲求值相关结果。 参看下面的例子

```
logplot <- function(y, ylab = deparse(substitute(y))) {
    y <- log(y)
    plot(y, ylab = ylab)
}
```

这看上去比较直接，但是 y 标签变成了一个比较难看的 `c(...)` 表达式。这是由于悠闲求值的规则导致在 `y` 修改*后* `ylab` 表达式的求值。 解决方法是首先强制 `ylab` 求值，即：

```
logplot <- function(y, ylab = deparse(substitute(y))) {
    ylab
    y <- log(y)
    plot(y, ylab = ylab)
}
```

注意，这种情况下，`eval(ylab)`可能很少使用。如果 `ylab` 是一个语言或表达式对象，那么这将导致这些对象也被求值。但有时结果不是期望的，如传递的数学表达式是 `quote(log[e](y))`。

`substitute` 的一个变种是 `bquote`，它把一些子表达式的值代替它们自己。上面的例子可以如下

```
> plot(0)
> for (i in 1:4)
+   text(1, 0.2 * i,
+        substitute(x[ix] == y, list(ix = i, y = pnorm(i))))
```

也可以更简洁的写成

```
plot(0)
for(i in 1:4)
   text(1, 0.2*i, bquote( x[.(i)] == .(pnorm(i)) ))
```

除了`.()`子表达式的内容被它们的值替换外，其它表达式都被引用。有一个可选的参数计算其它不同环境中的值。 `bquote` 的语法源自 LISP 的后置引用（backquote）宏。

Footnotes

[1] 译者注：原文为：“The special exception for substituting at the top level is admittedly peculiar. ”

[2] 译者注：原文为：“ It has been inherited from S and the rationale is most likely that there is no control over which variables might be bound at that level so that it would be better to just make substitute act as `quote`. ”

6.3 求值之外的东西

在本章前面就引入的 `eval` 函数作为调用对象求值的一种方法。但是，这不是全部。它还可以指定求值发生的环境。 默认情况下，这就是 `eval` 被调用时的求值框架，但是它常常需要设成其它值。 一般情况下，相应的求值框架是当前框架的父框架。特别，当求值对象是函数参数 `substitute` 操作后的结果，它将包含只对调用者有意义的变量（注意，没有理由期望调用者的变量也在被调用者的词法作用域里面）。 既然父框架里面的求值频繁发生，函数 `eval.parent` 可以作为 `eval(expr, sys.frame(sys.parent()))` 的简写方式存在。

另外一个常常出现的例子是一个列表或数据框的求值。例如，当 `data` 参数给定时，可能和 `model.frame` 函数发生关联。通常，模型公式的条目需要在 `data` 里面求值，但是它们偶尔也包括对 `model.frame` 的调用者内部条目的引用。在仿真研究时，这可能非常有用。因为这个原因，不仅需要对一个列表里面的表达式求值，还需要指定一个外围以便于变量在列表中找不到时可以继续搜索。因此，调用有下面的形式

```
eval(expr, data, sys.frame(sys.parent()))
```

注意，在一个给定环境中求值可能事实上改变了那个环境，在赋值操作中这最明显， 如

```
eval(quote(total <- 0), environment(robert$balance)) # rob Rob
```

这对于列表中的求值同样适用，但是原始的列表不会改变，因为我们操作的只是原始列表的一个拷贝。

6.4 表达式对象的求值

模式为 "expression"的对象在 Expression objects 有具体定义。它们和引用对象的列表非常相似。

```
> ex <- expression(2 + 2, 3 + 4)
> ex[[1]]
2 + 2
> ex[[2]]
3 + 4
> eval(ex)
[1] 7
```

注意，对一个表达式对象求值会对每个调用依次求值，但是最终的值是最后一个引用的值。基于这种考虑，它的行为几乎和复合语言对象 `quote({2 + 2; 3 + 4})` 完全一致。但也有一些细微的差别：调用对象在解析树里面不能和子表达式区分。这就说明它们是以子表达式一样的方式自动求值。表达式对象可以在求值过程中被识别，并且在某种意义上，保留了它们的引用。求值程序不会递归对一个表达式对象求值，仅仅当它如前面的例子一样直接传递给函数 `eval` 时求值。它们的差异可以通过下面的例子看出：

```
> eval(substitute(mode(x), list(x = quote(2 + 2))))
[1] "numeric"
> eval(substitute(mode(x), list(x = expression(2 + 2))))
[1] "expression"
```

语法分析器通过创建它的调用来描述一个表达式对象。这和处理数值向量和多个其它没有特别额外表示的对象的方法类似。但是，它会导致一些困惑：

```
> e <- quote(expression(2 + 2))
> e
expression(2 + 2)
> mode(e)
[1] "call"
> ee <- expression(2 + 2)
> ee
expression(2 + 2)
> mode(ee)
[1] "expression"
```

即， e 和 ee 在打印的时候看上去是一样的，但是其中一个是产生表达式对象的调用，而另外一个则是对象本身。

6.5 函数调用的操作

可以通过查看 sys.call 的结果 来了解一个函数是如何被调用的。下面是一个具体的例子，简单展示了一个函数的调用情况：

```
> f <- function(x, y, ...) sys.call()
> f(y = 1, 2, z = 3, 4)
f(y = 1, 2, z = 3, 4)
```

但是，除了调试，这不总是有用的，因为它需要函数跟踪参数匹配以解释函数调用情况。例如，在上面的例子中，它必须可以发现 x 的第二个事实参数被第一个形式参数匹配。

通常，我们需要所有事实参数和对应的形式参数绑定的调用。为达到这个目的，可以采用 match.call。这里是前述例子的一个变种，就是一个通过参数匹配返回它自身调用的函数

```
> f <- function(x, y, ...) match.call()
> f(y = 1, 2, z = 3, 4)
f(x = 2, y = 1, z = 3, 4)
```

注意第二个参数现在和 x 匹配，并且在结果出现在对应的位置中。

该项技术的最初使用是通过一样的参数调用另外一个函数，但可能会删除或增加一些其它的参数。一个典型的应用可以参见 lm 函数代码的起始部分：

```
mf <- cl <- match.call()
mf$singular.ok <- mf$model <- mf$method <- NULL
mf$x <- mf$y <- mf$qr <- mf$contrasts <- NULL
mf$drop.unused.levels <- TRUE
mf[[1]] <- as.name("model.frame")
mf <- eval(mf, sys.frame(sys.parent()))
```

注意结果调用在父框架下被求值。 在该框架下可以确定相关的表达式是有意义的。该调用可以看作是一个列表对象，它的第一个元素是函数名字，其它元素是以形式参数名字作为标签的事实参数表达式。因此，去除不想要的参数的技术可用来分配 NULL，如第 2 和第 3 行所示，和增加一个我们用标签列表赋值的参数（这里传递 drop.unused.levels = TRUE），如第 4 行所示。 为了改变被调用的函数的名字，既可以用这里的 as.name("model.frame")构造也可以用 quote(model.frame) 给列表的第一个元素赋值并且确信该值就是名字。

match.call 函数有一个 expand.dots 的参数，它是一个开关，如果设为 FALSE 将会使得所有... 参数成为标签为 ... 的单个参数。

```
> f <- function(x, y, ...) match.call(expand.dots = FALSE)
> f(y = 1, 2, z = 3, 4)
f(x = 2, y = 1, ... = list(z = 3, 4))
```

... 参数是一个列表（准确地说是成对列表），它和 S 里面调用 `list` 不一样：

```
> e1 <- f(y = 1, 2, z = 3, 4)$...
> e1
$z
[1] 3

[[2]]
[1] 4
```

使用这种形式的 `match.call` 的原因是简单地摆脱任何 ... 参数不至于传递一些函数未知的没有详细说明的参数。下面是一个来自 `plot.formula` 的例子：

```
m <- match.call(expand.dots = FALSE)
m$... <- NULL
m[[1]] <- "model.frame"
```

一个更加精细的应用是函数 `update.default`。可以在该函数中的可选额外参数集合中增加，替换，或取消一些原始调用的参数：

```
extras <- match.call(expand.dots = FALSE)$...
if (length(extras) > 0) {
    existing <- !is.na(match(names(extras), names(call)))
    for (a in names(extras)[existing]) call[[a]] <- extras[[a]]
    if (any(!existing)) {
        call <- c(as.list(call), extras[!existing])
        call <- as.call(call)
    }
}
```

注意，一旦 `extras[[a]] == NULL`，单个修改已经存在的参数需要小心一点。如前所示，在没有强迫的情况下调用一个对象时，连接操作（concatenation）不能使用；这是一个可以论证的程序问题。

为创建函数调用，还有两个额外的函数可以使用，它们分别是 `call` 和 `do.call`。

函数 `call` 允许通过函数名字和参数列表创建一个调用对象

```
> x <- 10.5
> call("round", x)
round(10.5)
```

如上所见， 是 `x` 的值而不是符号 加入了调用中，因此和 `round(x)` 有明显的差异。这种形式用的非常地少，但是当一个函数的名字可以作为一个字符变量时，这会非常有用。

函数 do.call 是相关的，但会立即对调用求值和从含有所有参数的模式为"list"的对象里面获取参数。一个很自然的应用是当我们向把一个函数（如cbind）用于一个列表或数据框的所有对象时。

```
is.na.data.frame <- function (x) {
    y <- do.call("cbind", lapply(x, "is.na"))
    rownames(y) <- row.names(x)
    y
}
```

其它一些应用包括基于 do.call("f", list(...)) 构造的变种。但是，我们必须知道这包括实际参数调用前的参数求值。这可能阻止函数自身的悠闲求值和参数替换方面。一个类似的注意同样适用于 call 函数。

6.6 函数操作

可以操作一个函数或闭包的分量常常非常有用。 R 为这种目的提供了一整套接口函数。

body

返回函数主体的表达式。

formals

返回一个函数的形式参数列表。它是 pairlist。

environment

返回函数的关联环境。

body<-

用给定的表达式设置函数的主体。

formals<-

用给定列表设置函数的形式参数。

environment<-

用给定的环境设置函数的环境。

用 evalq(x <- 5, environment(f)) 可以改变函数环境中不同变量的绑定。

用 as.list 可以把一个函数变为列表。 这样做的结果是形式参数的列表和函数主体的连接。相反，这种列表可以用 as.function 转换为函数。这种用法是为了和 S 兼容。注意当 as.list 被使用时，环境信息会丢失，而 as.function 有一个参数允许重新设置环境。

7 系统和其它语言的接口

- Operating system access: 操作系统访问
- Foreign language interfaces: 其它语言接口
- .Internal and .Primitive: .Internal 和 .Primitive

7.1 操作系统访问

R 通过函数 `system` 访问操作系统的命令层。 不同的操作系统在细节上可能有点不同（见在线帮助），但基本上第一个参数是用于执行的字符串 `command`（不一定通过命令层），第二个参数是 `internal`，如果它是真的话，它会把命令的输出结果输入到一个 R 字符向量中。

函数 `system.time` 和 `proc.time` 用来计时用的（尽管在非 Unix 类型的系统中，可以得到的计时信息非常有限）。

可以用下面的函数访问和获得操作系统环境的信息

`Sys.getenv` 操作系统环境变量

`Sys.putenv`

`Sys.getlocale` 系统本地变量（locale）

`Sys.putlocale`

`Sys.localeconv`

`Sys.time` 当前时间

`Sys.timezone` 时区

在所有系统上，有一套统一的文件读取函数集：

`file.access` 确定文件的权限

`file.append` 连接文件

`file.choose` 提示用户文件名

`file.copy` 拷贝文件

`file.create` 创建或截取文件

`file.exists` 检测文件的存在性

`file.info` 各种文件信息汇总

`file.remove` 删除文件

`file.rename` 重命名文件

`file.show` 显示一个文本文件

`unlink` 去除文件或目录

还有平台依赖的文件名和路径的操作函数。

`basename` 没有目录的文件名

`dirname` 目录名

`file.path` 构建文件的路径

`path.expand` 在 Unix 路径下展开 ~ 路径

7.2 其它语言接口

用编译后的代码为 R 增加函数的技术细节可以参考 System and foreign language interfaces (Writing R Extensions)。

函数 `.C` 和 `.Fortran` 提供了标准接口，允许在程序构建时（build time）使用外部代码或者通过 `dyn.load` 连接 R 和已编译的代码。这个两个函数的设计最初是想对 **C** 和 FORTRAN 的代码分别编译，但是 `.C` 函数可用于其它能产生 C 接口的语言中，如 C++。

函数 `.Call` 和 `.External` 提供了在允许已编译代码（主要是编译后的 **C** 代码）中操作 R 对象的接口。

7.3 .Internal 和 .Primitive

`.Internal` 和 `.Primitive` 接口用于在程序构建时调用编译入 R 的 **C** 代码。参见 See .Internal and .Primitive (Writing R Extensions).

8 异常处理

R 里面的异常处理由两种机制提供。函数 `stop` 或 `warning` 可以直接调用，而 `"warn"`一类的函数可以有选择地用于问题处理的控制。

- stop: 终止
- warning: 警告
- on.exit: on.exit 函数
- Error options: 错误可选项

8.1 终止

`stop` 的调用会暂停当前表达式的求值，打印信息参数，把执行控制返回给顶层。

8.2 警告

函数 `warning` 只有一个字符串类型的参数。调用 `warning` 的行为依赖于可选函数`"warn"`的值。如果 `"warn"` 的值是负的，警告将会被忽略。如果是 0，它们将会被保存，在顶层函数结束后打印出来。如果是 1，它们会在发生时就打印出来，如果是 2（或更大），警告就变成错误了。

如果 `"warn"` 是 0（默认），顶层变量 `last.warning` 将会被创建，随后调用的 `warning` 所产生的信息会保存在这个向量中。如果函数求值结束时，产生的警告信息少于 10 条，它们将会被直接打印。如果超过 10 条，只打印显示有多少条警告信息。这两种情况下， `last.warning` 都是包含信息的向量。

8.3 on.exit 函数

可以在一个函数主体部分中的任何地方插入一个 `on.exit` 的调用。 `on.exit` 调用的作用是保存函数主体的值使得函数跳出后它仍然可以被执行。它允许函数改变一些系统参数和保证在函数结束时它们设有恰当的值。 `on.exit` 保证函数在直接退出或警告退出的情况下一定运行。

`on.exit` 代码求值过程中的错误会导致立即跳至顶层而不进一步执行 `on.exit` 代码。

`on.exit` 只用一个参数，就是一个在函数结束时才会被执行的表达式。

8.4 错误可选项

有几个 `options` 变量可用来控制 R 的错误和警告处理。它们列举如下。

warn

控制警告信息的打印。

warning.expression

设置一个表达式，当警告发生时执行。常规打印的警告在该选项设置时被抑制了。

error

设置一个错误发生时会被执行的表达式。错误和警告信息会在该表达式求值前打印出来了。

通过 `options("error")` 设置的表达式在调用 `on.exit` 前求值。

当有错误发生，我们可以用 `options(error = expression(q("yes")))` 让 R 程序终止。在这种情况下，一个错误会导致 R 终止但全局环境会被保存。

---

9 调试

代码调试通常是一门艺术。 R 提供了多个工具用于帮助用户找到代码中的问题。这些工具会使代码在特定的地方暂停执行，并且可以跟踪计算过程的当前状态。

大多数调试都是通过 `browser` 或 `debug` 来实现的。这两个函数依赖的内在机制是一样的，二者都给用户提供了特别的提示。任何命令都可以在提示下键入。该命令的求值环境 是当前的激活环境。这使得你可以检测任何变量等的当前状态。

R 里面有五个不同解释方式的特殊命令。它们是

<RET>

在函数调试时，去下一个语句处。如果 `browser` 被调用，表示继续执行。

c

cont

继续执行。

n

执行函数中的下一条语句。这和 `browser` 作用类似。

where

显示调用堆栈

Q

挂起执行，立即跳到顶层。

如果一个局部变量的名字和上面列出命令的名字一样，可以用 `get` 访问它的值。把局部变量的名字（要加引号）作为参数调用 `get` 将返回它在当前环境中的值。调试器仅允许对解释型表达式的访问。如果一个函数调用外部语言（如 **C**），那么调试器不能访问外部语言的语句。执行控制将会在 R 里面求值的下一个语句处暂停。符号调试器如 `gdb` 可用来调试编译后的代码。

- browser: 浏览
- debug/undebug: 调试/不调试
- trace/untrace: 跟踪/不跟踪
- traceback: 回溯

9.1 浏览

在函数 `browser` 调用的地方，R 会暂停执行，并且为用户提供特别的提示。赋给 `browser` 的参数会被自动忽略。

```
> foo <- function(s) {
+ c <- 3
+ browser()
+ }
> foo(4)
Called from: foo(4)
Browse[1]> s
[1] 4
Browse[1]> get("c")
[1] 3
Browse[1]>
```

9.2 调试/不调试

可以通过 `debug(`*fun*`)` 对任何函数调用调试器。随后，只要该函数求值，调试器都会被调用。调试器允许你控制函数主体里面的语句求值。在任一语句求值前，该语句会被打印出来并且给出一个特别的提示。可以给定任何命令，包括前面表格中有特殊意义的命令。

通过调用 undebug（以函数作为参数）来关闭调试。

```
> debug(mean.default)
> mean(1:10)
debugging in: mean.default(1:10)
debug: {
    if (na.rm)
        x <- x[!is.na(x)]
    trim <- trim[1]
    n <- length(c(x, recursive = TRUE))
    if (trim > 0) {
        if (trim >= 0.5)
            return(median(x, na.rm = FALSE))
        lo <- floor(n * trim) + 1
        hi <- n + 1 - lo
        x <- sort(x, partial = unique(c(lo, hi)))[lo:hi]
        n <- hi - lo + 1
    }
    sum(x)/n
}
Browse[1]>
debug: if (na.rm) x <- x[!is.na(x)]
Browse[1]>
debug: trim <- trim[1]
Browse[1]>
debug: n <- length(c(x, recursive = TRUE))
Browse[1]> c
exiting from: mean.default(1:10)
[1] 5.5
```

9.3 跟踪/不跟踪

另外一种监控 R 行为的方法是 trace 机制。具体用法是以你想跟踪的函数名字作为参数调用 trace。这里的名字一般不需要用引号括住，但也一些函数名字需要用引号引住以防止语法错误。

当对一个函数调用 trace 时，每次该函数求值，它的调用都会被打印出来。该机制可以通过调用以函数作为参数的 untrace 去除。

```
> get("[<-")
.Primitive("[<-")
> trace("[<-")
> x <- 1:10
> x[3] <- 4
trace: "[<-"(*tmp*, 3, value = 4)
```

9.4 回溯

如果一个错误导致控制跳到顶层，一个名为.Traceback 的特殊变量会在顶层空间中设置。.Traceback 是一个字符向量，它有一个记录错误发生时激活的函数调用的条目。可以通过调用 traceback 来查看 .Traceback 里面的内容。

10 解析器

解析器（parser）可以把文本描述的 R 代码转换内部形式，然后传递给 R 内部可以执行特殊指令的求值程序。这里的内部形式本身就是一个 R 对象，而且可以在 R 系统里面保存和操作。

- The parsing process: 解析过程
- Comments: 注释
- Tokens: 标记
- Expressions: 表达式

10.1 解析过程

- Modes of parsing: 解析模式
- Internal representation: 内在描述
- Deparsing: 语法分析

10.1.1 解析模式

R 里面的解析以三个不同的形式存在：

- “读入-求值-打印”（read-eval-print）循环
- 文本文件的解析
- 字符串的解析

“读入-求值-打印”循环构成了 R 基本命令行操作的接口。它一直运行，直到可以得到完整的 R 表达式，才停止读入文本输入。表达式可以被不同的输入行分割。主要提示符（默认是> ）表示解析器准备读入新的表达式，而延续提示符（默认是+ ）表明解析器在等待输入不完整的表达式的剩余部分。在输入过程中，表达式会被转换为内部形式，解析后的表达式会传递给求值程序并且把结果打印出来（除非特别要求不显示）。如果解析器发现有和语言语法不兼容的地方，会提示“语法错误”，然后解析器自身会重新设置并且在下一个输入行的开始重新等待输入。

文本文件可以用 `parse` 函数解析。特别是，在执行函数 `source` 的时候也是这样，因此使得命令可以保存在外部文件中，而执行的时候就像直接从键盘上键入的一样。注意，尽管整个文件被解析了，但在任何求值操作进行前，首先会进行语法检查。

字符串，或它的向量，都可以通过参数 `text=`赋给 `parse` 解析。字符串会被精确处理，就好像它们是一个输入文件的行。

10.1.2 内在描述

解析后的表达式保存在一个含有解析树的 R 对象中。这种对象的完整描述可以参考 Language objects 和 Expression objects。总之，每个基本的 R 表达式都以函数调用的形式保存，并且是一个首元素为函数名字，其它元素是参数（也可能是 R 表达式）的列表。 列表元素可以被命名，和形式与事实参数的标签匹配对应。注意 *所有* R 的语法元素都是以这种方式处理的，例如：赋值操作 `x <- 1` 编码为 `"<-"(x, 1)`。

10.1.3 语法分析

任何 R 对象可以用 `deparse` 转换成 R 表达式。这常常用于连接结果输出，如为了在图上加标签。注意，只有模式是`"expression"` 的对象在语法分析后结果再解析时不会改变。例如，数值向量 `1:5` 会被语法分析为 `"c(1, 2, 3, 4, 5)"`，接下来的则会解析为对函数 `c` 的调用。如果可能，对语法分析和再解析的表达式求值会得到和原始表达式一样的求值结果，但是也有一些难以使用的例外，主要是一些首先就不是从文本描述中产生的表达式。

10.2 注释

R 里面的注释会被解释器忽略。一行里面， 从 `#` 开始到末尾部分都看作是注释，除非 `#` 字符加了引号。例如，

```
> x <- 1  # This is a comment...
> y <- "  #... but this is not."
```

10.3 标记

标记是编程语言的基本构建块。它们在解析器操作的句法分析（syntactic analysis）前（至少在概念上）的 *词法分析*（lexical analysis）时被识别。

- Literal constants: 字面常量
- Identifiers: 标识符
- Reserved words: 保留字
- Special operators: 特别操作符
- Separators: 分割符

- Operator tokens: 操作标记
- Grouping: 成组
- Indexing tokens: 索引标记

10.3.1 常量

有四种常量类型，逻辑型，数值型，复数型和字符串型。

此外，还有四种特殊常量，`NULL`，`NA`， `Inf` 和 `NaN`。

`NULL` 用于标识空对象。`NA` 用于说明缺失(“不可得到”)值。 `Inf` 表示无穷大，`NaN` 则表示一个在 IEEE 浮点计算中不能表示的值（例如，1/0 和 0/0 的结果）。

逻辑常量要么是 `TRUE` 要么是 `FALSE`。

数值常量遵循和 **C** 语言类似的语法规则。它们有一个零或更多数字构成的整数部分，随后是一个可选的.，一个零或更多数字构成的小数部分，以及一个由 E 或 e 构成的指数部分，有时还有一个可选的符号和零或更多数字构成的字符串。任何一部分都可以是空，但不能同时为空。

```
有效的数值常量： 1 10 0.1 .2 1e-7 1.2e+7 2e 3e+
```

后面两个例子在实践中几乎没有任何用途，但是它们是合法的[1]，并且分别解释为 2 和 3。

数值常量还可以是十六进制值，只是需要以 0x 开始，或在 0x 后加上零或更多的数字，包括 a-f 或 A-F。

注意没有整型常量的隔离类。

还要注意的是，前置符号以一元操作符看待，而不是常量的一部分。

复数常量含有以 i 作为后缀的十进制数值常量。注意只有虚部数字是事实上的常量，其它复数数值可以解析为数字和虚部数字间的一元或二元操作。

```
合法的复数常量： 2i 4.1i 1e-2i
```

字符串常量通过一对单引号 (') 或双引号 (") 界定，它可以包括所有其它可以打印的符号。字符串里面的引号和其它特殊字符可以用 *转义控制序列*（escape sequences）实现：

`\'` 单引号

`\"` 双引号

`\n` 新的一行

`\r` 回车符

`\t` 制表符

`\b` 退格符

`\a` bell

`\f` form feed

`\v` 垂直制表符

`\\` 反斜杠

`\`*`nnn`* 通过给定八进制代码得到字符 - `0 ... 8` 里面的任何一个，两个，三个数字都是合法的。

`\x`*`nn`* 通过给定的十六进制代码得到字符 - 一个或两个十六进制数字（`0 ... 9 A ... F a ... f` 中的一个）的序列。

`\u`*`nnnn`* `\u{`*`nnnn`*`}` （其中要求本地系统支持多字节码，否则报错）。以给定的十六进制数字获得 Unicode 字符 - 最多可以是四个十六进制数字。字符需要是本地系统合法的字符。

`\U`*`nnnnnnnn`* `\U{`*`nnnnnnnn`*`}` （其中要求本地系统支持多字节码并且不是 Windows，否则报错）。 以给定的十六进制数字获得 Unicode 字符 - 最多可以是八个十六进制数字。

单引号可以直接嵌入双引号界定的字符串中，反之也然。

Footnotes

[1] 译者注：我直接在控制台键入，报语法错误

10.3.2 标识符

标识符由字母，数字，点号 (.) 和下划线构成。它们不能以数字和下划线开始，也不能以点号后面跟数字开始。

字母的定义依赖于本地的环境：合法字符的精确集合由 C 表达式 `(isalnum(c) || c == '.' || c == '_')` 给定，并且在许多西欧的本地环境中还包括重音符号。

注意，以点号开头的标识符默认不被 ls 函数列出，并且 ...， ..1 和 ..2 等是比较特殊的。

同样需要注意的是，对象可以有不是标识符的名字。这些通常通过 get 和 assign 读取，尽管它们在一些不会混淆的情况下（很少出现）可以用文本字符串表示（如"x" <- 1）。因为 get 和 assign 没有要求它们所赋的名字一定是标识符，所以它们不能区分下标操作和赋值函数。下面两个例子 *不是* 等价的

```
x$a<-1       assign("x$a",1)

x[[1]]       get("x[[1]]")

names(x)<-nm assign("names(x)",nm)
```

10.3.3 保留字

下面的标识符有特别的含义，不能用作对象的名字

```
if else repeat while function for in next break
TRUE FALSE NULL NA Inf NaN
... ..1 ..2 etc.
```

10.3.4 特别操作符

R 允许用户定义中缀操作符。这些操作符的形式是由%字符界定的字符串。该字符串可以是任何除%外的可打印字符。字符串的转义控制序列在这里将会失效。

下面是一些预定义的该类型操作符

```
%% %*% %/% %in% %o% %x%
```

10.3.5 分隔符

尽管不是严格的标记，伸展空格字符（空格和制表符）可以用来界定模糊的标记（比较一下 x<-5 和 x < -5）。

新行既可以作标记分隔也可以是表达式终结。如果一个表达式可以在一行末结束，解析器会把它当作一行，否则新的一行当作空格处理。分号(;) 可用来分隔同一行中独立的基本表达式。 有多个规则用于关键字 else：在一个组合表达式中， else 前面的新行会被抛弃，而在最外层，新行终止了 if 构造，并且随后的 else 会导致语法错误。这是一种有点反常的行为，这是由于 R 需要能以

交互模式使用，而且必须在用户一旦键入<RET>时能判断一个输入的表达式是否是完整的，不完整的或者不合法的。

逗号(,) 用于分隔函数参数和多重索引。

10.3.6 操作标记

R 使用下面的操作标记

+ - * / %% ^	算术
> >= < <= == !=	关系
! & |	逻辑
~	模型公式
-> <-	赋值
$	列表索引
:	序列

（上面的一些操作符在模型公式里面可能有不同的含义）

10.3.7 成组

普通括弧 — (和) — 用来在表达式中显式把代码成组和为函数定义和函数调用界定参数列表。

大括弧 — { 和 }— 用来界定函数定义，条件表达式和循环架构的句块。

10.3.8 索引标记

[] 和 [[]] 用来进行数组和向量的索引操作。还可以用$操作符对有标签的列表进行索引操作。

10.4 表达式

R 程序由 R 表达式依次组成。表达式可以是仅有一个常量或标识符的简单表达式，也可以是由其它部分（可能它们自身也是表达式）构成的复合表达式。

下面各节中将会详述可用到的各种句法结构。

- Function calls (expressions): 函数调用（表达式）
- Infix and prefix operators: 中缀和前缀操作符
- Index constructions: 索引构造
- Compound expressions: 复合表达式
- Flow control elements: 流控制元素
- Function definitions: 函数定义

10.4.1 函数调用

函数调用的形式是一个函数引用后面伴随着放在括弧里面的一个以逗号分隔的参数列表。

```
function_reference ( arg1, arg2, ..... , argn )
```

函数引用可以是

- 标识符（函数名字）
- 文本字符串（同上，在如果一个函数有一个不是合法标识符的名字时执行）
- 表达式（它可以求值成为一个函数对象）

任何参数都可以加上标签(*tag=expr*)，或者仅仅是一个简单的表达式。它还可以是空，或者是特殊标记 ...， ..2 等。

标签可以是标识符或文本字符串。

例如，

```
f(x)
g(tag = value, , 5)
"odd name"("strange tag" = 5, y)
(function(x) x^2)(5)
```

10.4.2 中缀和前缀操作符

操作符的优先级如下

```
::
$ @
^
```

```
- +                    (一元操作符)
:
%xyz%
* /
+ -                    (二元操作符)
> >= < <= == !=
!
& &&
| ||
~                      (一元操作符和二元操作符)
-> ->>
=                      (作为赋值操作符)
<- <<-
```

幂操作符 ^ 和 左赋值操作符 <- - = <<- 从右往左操作，所有其它操作符从左往右操作。因此 `2 ^ 2 ^ 3` 和 2 ^ 8 相等而不是 4 ^ 3，而 `1 - 1 - 1` 是 -1，而不是 1。

注意用于求模和整除的操作符 `%%` 和 `%/%` 优先级高于乘法和除法。

尽管该操作符要求不太严格，但仍需提起。= 号可用于函数调用时标记参数也可用于函数定义时设定默认值。

从某种意义上说，$也是一种操作符，但不可以放在任意对象的右边，这在 Index constructions 部分进行了一定的讨论。它比其它任一操作符的优先级别都高。

一个一元或二元操作的解析形式完全等价于一个以操作符作为函数名字的函数调用，并且操作数就是函数参数。

括弧可以看作是一个名为"`(`"的一元操作符。这也包括用于设定操作优先（如 `a * (b + c)`）中的括弧。

注意赋值符号是和 算术，关系和逻辑操作符类似的操作符。在赋值操作的目标侧，任何表达式都是允许的（解析器层次）。（从解析器的角度，`2 + 2 <- 5` 是合法的表达式。尽管求值程序反对。）类似的说明还可用于模型公式中的操作符。

10.4.3 索引构造

R 有三个索引构造，其中两个语法类似尽管他们在语义上稍稍有点差异：

```
object [ arg1, ..... , argn ]
object [[ arg1, ..... , argn ]]
```

object 在形式上可以是任何合法的表达式，但它可理解为指向或者操作一个可取子集的对象。参数常常是数字或字符串索引，但其它类型的参数也是允许的（特别是 `drop = FALSE`）。

在内部，这些索引构造以函数调用的形式保存，其中函数名字为分别是 `"["` 和 `"[["`。

第三种索引方式是

```
object $ tag
```

这里， *object* 如前，而 *tag* 是一个标识符或一个文本字符串。内部实现是，它依然以一个名为 `"$"` 的函数调用而保存。

10.4.4 复合表达式

复合表达式的形式如下

```
{ expr1 ; expr2 ; ..... ; exprn }
```

分号可以用换行符代替。在内部，这里以函数调用保存，函数名字就是 `"{"`，而表达式就是参数。

10.4.5 流控制元素

R 含有作为特殊语法结构的流控制结构

```
if ( cond ) expr
if ( cond ) expr1 else expr2
while ( cond ) expr
repeat expr
for ( var in list ) expr
```

这些结构中的表达式是典型的复合表达式。

在循环结构（`while`，`repeat`，`for`）中，我们还可以使用 `break`（终止循环）和 `next`（忽略本次循环进入下次循环）。

在内部，这些结构都以函数调用的方式保存：

```
"if"(cond, expr)
"if"(cond, expr1, expr2)
"while"(cond, expr)
"repeat"(expr)
"for"(var, list, expr)
"break"()
```

```
"next"()
```

10.4.6 函数定义

函数定义的形式是

```
function ( arglist ) body
```

函数主体是表达式，通常是复合表达式。 *arglist* 是一个逗号分隔的条目列表（这些条目可能是标识符），或者是 *identifier* = *default* 的形式，或者是特定标记 ...。*default* 可以是任意合法的表达式。

注意函数参数和列表标签等不一样，它不能有文本字符串的“奇怪名字”（strange names）。

在内部，函数定义也是以函数调用的方式保存。这个函数的名字就是 `function`，它有两个参数，即 the *arglist* 和 *body*。 *arglist* 以有标签的成对列表的形式保存，其中标签就是参数名字而值就是默认的表达式。

Function and Variable Index

- `body`: Function objects
- `body`: Manipulation of functions
- `body<-`: Manipulation of functions
- `break`: Looping
- `browser`: browser
- `debug`: debug/undebug
- `dirname`: Operating system access
- `do.call`: Manipulation of function calls
- `emptyenv`: Environment objects
- `environment`: Manipulation of functions
- `environment`: Function objects
- `environment<-`: Manipulation of functions
- `eval`: More on evaluation
- `file.access`: Operating system access
- `file.append`: Operating system access
- `file.choose`: Operating system access
- `file.copy`: Operating system access
- `file.create`: Operating system access
- `file.exists`: Operating system access
- `file.info`: Operating system access
- `file.path`: Operating system access
- `file.remove`: Operating system access
- `file.rename`: Operating system access
- `file.show`: Operating system access
- `for`: for
- `formals`: Function objects
- `formals`: Manipulation of functions
- `formals<-`: Manipulation of functions
- `get`: Identifiers
- `is.na`: NA handling
- `is.nan`: NA handling
- `match.arg`: Argument matching
- `match.call`: Manipulation of function calls
- `match.call`: Argument matching
- `match.fun`: Argument matching
- `missing`: NA handling
- `mode`: Objects
- `NA`: Indexing by vectors
- `NA`: NA handling
- `names`: Names
- `names<-`: Names
- `NaN`: NA handling
- `new.env`: Environment objects
- `next`: Looping
- `NextMethod`: NextMethod
- `NULL`: NULL object
- `on.exit`: on.exit
- `pairlist`: Pairlist objects
- `path.expand`: Operating system access
- `proc.time`: Operating system access
- `quote`: Language objects

Concept Index

- 符号: Symbol lookup
- 符号: Symbol objects
- 符号: Scope
- 符号: Substitutions
- 赋值: Operators
- 赋值: Subset assignment
- 赋值: Global environment
- 赋值: Function calls
- 赋值: Stacks
- 赋值: Manipulation of function calls
- 赋值: Argument evaluation
- 赋值: UseMethod
- 赋值: Infix and prefix operators
- 赋值: UseMethod
- 赋值: More on evaluation
- 赋值: Scope
- 赋值: Inheritance
- 赋值, 符号: Scope
- 赋值, 悠闲: Substitutions
- 赋值, 悠闲（lazy）: Objects
- 赋值语句: Function objects
- 复合赋值: Subset assignment
- 函数: Object-oriented programming
- 函数: Stacks
- 函数: Internal representation
- 函数: Scope
- 函数: Lexical environment
- 函数: Scope
- 函数: Manipulation of functions
- 函数: Arguments
- 函数: Manipulation of functions
- 函数: Scope
- 函数: Function objects
- 函数: Function calls (expressions)
- 函数: Function objects
- 函数: Writing functions
- 函数: Definition
- 函数: Function objects
- 函数: Argument evaluation
- 函数: Built-in objects and special forms
- 函数: Argument evaluation
- 函数: Promise objects
- 函数: Dot-dot-dot
- 函数: Function calls
- 函数: Syntax and examples
- 函数: Argument evaluation
- 函数: Evaluation environment
- 函数: Manipulation of function calls

- 解析: Parser
- 解析: Computing on the language
- 局部匹配: Indexing by vectors
- 框架: Lexical environment
- 类型: Objects
- 类型: Vector objects
- 类型: Names
- 类型: NA handling
- 类型: Basic types
- 类型: Objects
- 面向对象: Definition
- 面向对象: Object-oriented programming
- 名字: Scope of variables
- 名字: Symbol objects
- 名字: Symbol lookup
- 名字: Argument matching
- 名字: Propagation of names
- 名字: Direct manipulation of language objects
- 名字: Argument evaluation
- 名字: Debugging
- 名字: Arguments
- 名字: NextMethod
- 名字: Method dispatching
- 命名空间: Search path
- 命名类型: Language objects
- 模式: Symbol objects
- 模式: Objects
- 模式: Vector objects
- 强制转换: NA handling
- 强制转换: Classes
- 强制转换: Objects
- 强制转换: Any-type
- 强制转换: Symbol objects
- 求值: Evaluation environment
- 求值, 表达式: Arguments
- 求值, 表达式: Expression objects
- 求值, 表达式: Promise objects
- 求值, 参数: Argument evaluation
- 求值, 符号: Symbol lookup
- 求值, 符号: Attributes
- 求值, 语句: Control structures
- 搜索路径: Search path
- 索引: Indexing by vectors
- 索引: Indexing matrices and arrays
- 索引: Vector objects
- 索引: Indexing matrices and arrays
- 索引: List objects
- 索引: Indexing

- 向量: Vector objects
- 向量: Operators
- 向量: Dimensions
- 原子型: Vector objects
- 允诺: Promise objects
- 值: Symbol lookup
- 注释: Comments
- 作用域: Scope
- 作用域: Stacks
- 作用域: Scope
- 作用域: Scope of variables
- 作用域: More on evaluation

Appendix A References

Richard A. Becker, John M. Chambers and Allan R. Wilks (1988), *The New S Language*. Chapman & Hall, New York. This book is often called the "*Blue Book*".