**3.反应机制基础**

**3.1简介**

在Shiny中，您使用反应式编程来表达服务器端逻辑。反应式编程是一种优雅而强大的编程范式，但一开始可能会让人迷失方向，因为它与编写普通脚本的范式截然不同。反应式编程的关键思想是定义依赖关系图，这样当输入发生变化时，所有相关输出都会自动更新。这使应用程序的流程变得相当简单，但需要一段时间才能让您了解所有程序是如何组合在一起的。

本章将概要地介绍反应式编程，教您在Shiny应用程序中使用的最常见反应式编程结构的基础知识。我们将从服务器函数的概览开始，更详细地讨论输入和输出参数是如何工作的。接下来，我们将回顾反应机制的最简单形式(输入直接连接到输出)，然后讨论反应表达式如何帮助您消除重复的工作。最后，我们将看一看新用户遇到的一些常见障碍。

**3.2服务器函数**

正如你所见，每个Shiny应用程序都是这样的：

library(shiny)

ui <- fluidPage(

# front end interface

)

server <- function(input, output, session) {

# back end logic

}

shinyApp(ui, server)

上一章介绍了前端的基础知识，前端的ui对象包含了向应用程序的每个用户呈现的HTML。ui端很简单，因为每个用户都获得相同的HTML。服务器端更复杂，因为每个用户都需要获得独立版本的应用程序；当用户A移动滑块时，用户B不应看到其输出发生变化。

为了实现这种独立性，每次新会话[[1]](#footnote-1)启动时，Shiny都会调用server()函数。与任何其他R函数一样，当调用服务器函数时，它会创建一个新的本地环境，该环境独立于函数的其他每次调用。这允许每个会话具有唯一的状态，并在会话之间隔离在函数内部创建的变量。这就是为什么您将在Shiny中执行的几乎所有反应式编程都将在服务器函数[[2]](#footnote-2)中进行的原因。

服务器函数有三个参数：输入、输出和会话。您不会亲自调用服务器函数，所以您永远不会自己创建这些对象。相反，它们是在会话开始时由Shiny创建的，并连接回特定会话。现在，我们将重点讨论输入和输出参数，并将会话参数留给后面的章节。

**3.2.1输入**

输入参数是一个类似列表的对象，包含从浏览器发送的所有输入数据，根据输入ID命名。例如，如果您的UI包含输入ID为count的数字输入控件，如下所示：

ui <- fluidPage(

numericInput("count", label = "Number of values", value = 100)

)

然后，您可以使用input$count访问该输入的值。它最初将包含值100，并将随着用户在浏览器中更改值而自动更新。

与典型列表不同，输入对象是只读的。如果在服务器函数中试图修改输入，则会出现错误：

server <- function(input, output, session) {

input$count <- 10

}

shinyApp(ui, server)

#> Error: Can't modify read-only reactive value 'count'

出现这种错误是因为输入反映了浏览器中发生的事情，而浏览器是Shiny的“唯一真相来源”。如果您可以修改R中该变量的值，可能会引入不一致性，输入滑块在浏览器中表示一件事，而input$count在R中表示另一件事。这将使编程具有挑战性！稍后，在第8章中，您将学习如何使用updateNumericInput()等函数在浏览器中修改值，然后input$count将相应更新。

关于输入，还有一件更重要的事情：它对允许谁引用它是有选择性的。要引用输入，必须处于由函数renderText()或reactive()等创建的反应上下文中。我们将很快回到这个概念，但这是一个重要的约束，允许输出在输入发生变化时自动更新。这段代码说明如果犯了此错误，您将看到的错误：

server <- function(input, output, session) {

message("The value of input$count is ", input$count)

}

shinyApp(ui, server)

#> Error: Can't access reactive value 'count' outside of reactive consumer.

#> ℹ Do you need to wrap inside reactive() or observer()?

**3.2.2输出**

输出与输入非常相似：它也是一个根据输出ID命名的类似列表的对象。主要区别在于，您使用它发送输出，而不是接收输入。始终将输出对象与渲染函数配合使用，如以下简单示例所示：

ui <- fluidPage(

textOutput("greeting")

)

server <- function(input, output, session) {

output$greeting <- renderText("Hello human!")

}

(请注意，ID在UI中加引号，但在服务器中不加引号。)

渲染函数做两件事：

1、它建立了一个特殊的反应上下文，自动跟踪输出使用的输入。

2、它将R代码的输出转换为适合在网页上显示的HTML。

与输入一样，输出对如何使用它也很挑剔。如果出现以下情况，则会出现错误：

3、您忘记了渲染函数。

server <- function(input, output, session) {

output$greeting <- "Hello human"

}

shinyApp(ui, server)

#> Error: Unexpected character object for output$greeting

#> ℹ Did you forget to use a render function?

4、您尝试读取输出。

server <- function(input, output, session) {

message("The greeting is ", output$greeting)

}

shinyApp(ui, server)

#> Error: Reading from shinyoutput object is not allowed.

**3.3反应式编程**

如果一个应用程序只有输入或输出，那么它将非常无聊。当你有一个同时具备这两种功能的应用程序时，Shiny的真正魔力就出现了。让我们看一个简单的例子：

ui <- fluidPage(

textInput("name", "What's your name?"),

textOutput("greeting")

)

server <- function(input, output, session) {

output$greeting <- renderText({

paste0("Hello ", input$name, "!")

})

}

很难在书中说明这是如何工作的，但我努力做得最好，如图3.1所示。如果您运行应用程序，并在“名称”框中键入，您将看到当您键入[[3]](#footnote-3)时，问候语会自动更新。

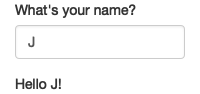


图3.1：反应意味着输出会随着输入的变化而自动更新，如在本应用程序中，我输入“J”、“o”、“e”。在线演示https://hadley.shinyapps.io/ms-connection.

这是Shiny的一个重要思想：您不需要告诉输出何时更新，因为Shiny会自动为您找出来。它是如何工作的？函数体中到底发生了什么？让我们更准确地思考一下服务器函数中的代码：

output$greeting <- renderText({

paste0("Hello ", input$name, "!")

})

这很容易理解为“将‘hello’和用户名粘贴在一起，然后将其发送到output$greeting”。但这种思维模式在一个微妙但重要的方面是错误的。想想看：在这个模型中，您只发出一次指令。但是每次我们更新input$name时，Shinny都会执行该操作，因此肯定有更多的事情发生了。

该应用之所以能够工作，是因为代码没有告诉Shiny创建字符串并将其发送到浏览器，而是告诉Shiny如果需要，它可以如何创建字符串。这取决于Shiny何时(甚至是否！)应该运行代码。它可能会在应用程序启动后立即运行，可能要晚一点；它可能会运行多次，也可能永远不会运行！这并不是说Shiny是反复无常的，只是说由Shiny决定何时执行代码，而不是你。把你的应用程序想象成向Shiny提供食谱，而不是命令。

**3.3.1命令式编程与声明式编程**

命令和食谱之间的这种差异是两种重要编程风格之间的关键差异之一：

1、在命令式编程中，您发出一个特定的命令并立即执行。这是您在分析脚本中习惯的编程风格：您命令R加载数据、转换数据、可视化数据，并将结果保存到磁盘。

2、在声明式编程中，您表达更高级别的目标或描述重要的约束，并依靠其他人来决定如何和/或何时将其转化为行动。这是您在Shiny中使用的编程风格。

在命令式代码中，你说“给我做一个三明治”[[4]](#footnote-4)。在声明式代码中你说“无论我什么时候看冰箱，都要确保冰箱里有一个三明治”。命令式代码是断言的；声明式代码是被动响应的。

大多数情况下，声明式编程非常自由：您描述了您的总体目标，软件会找出如何实现这些目标，而无需进一步干预。不利的一面是，有时你确切地知道自己想要什么，但你无法想出如何以声明式系统理解的方式来构建它[[5]](#footnote-5)。本书的目的是帮助你理解基本理论，以便尽可能少地发生这种情况。

**3.3.2懒惰**

Shiny中声明式编程的优势之一是它允许应用程序非常懒惰。一个Shiny应用程序只会执行所需的最小工作量[[6]](#footnote-6)，来更新你当前可以看到的输出控件。然而，这种惰性带来了一个重要的缺点，你应该意识到。你能发现下面的服务器有什函数问题吗？

server <- function(input, output, session) {

output$greting <- renderText({

paste0("Hello ", input$name, "!")

})

}

如果你仔细看，你可能会注意到我写的是greting而不是greeting。这不会在Shinny中生成错误，但不会实现您想要的功能。greting输出不存在，因此renderText()中的代码永远不会运行。

如果您正在开发一个Shiny应用程序，并且无法弄清楚为什么代码永远无法运行，请仔细检查您的UI和服务器函数是否使用相同的标识符。

**3.3.3反应图**

Shiny的懒惰还有一个重要的特点。在大多数R代码中，您可以通过自上而下阅读代码来理解执行顺序。这在Shiny中不起作用，因为代码仅在需要时运行。为了理解执行顺序，您需要查看反应图，它描述了输入和输出是如何连接的。上述应用程序的反应图非常简单，如图3.2所示。

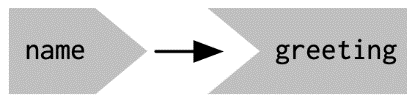


图3.2：反应图显示了输入和输出连接的方式

反应图中每个输入和输出都有一个符号，每当输出引用输入时，我们将输入连接到输出。此图告诉您，无论何时更改名称，都需要重新计算问候语。我们通常将这种关系描述为问候语对名称具有反应依赖。

请注意我们用于输入和输出的图形形状的约定：名称输入自然地适配于问候语输出。我们可以将它们紧密地组合在一起，如图3.3所示，以强调它们的组合方式；我们通常不会这么做，因为它只适用于最简单的应用程序。



图3.3：反应图组件使用的形状反映了它们的连接方式。

反应图是了解应用程序工作方式的强大工具。随着应用程序变得越来越复杂，快速绘制反应图的高级草图通常很有用，以提醒您所有部分是如何组合在一起的。在本书中，我们将向您展示反应图，以帮助您理解示例的工作方式，稍后，在第14章中，您将学习如何使用reactlog来为您绘制反应图。

**3.3.4反应表达式**

在反应图中还有一个更重要的组件：反应表达式。我们将很快详细讨论反应式表达式；现在，可以将它们视为一种工具，通过在反应图中引入额外的节点来减少反应代码中的重复。

在我们非常简单的应用程序中，我们不需要一个反应式表达式，但无论如何，我会添加一个，这样您就可以看到它如何影响反应图，如图3.4所示。

server <- function(input, output, session) {

string <- reactive(paste0("Hello ", input$name, "!"))

output$greeting <- renderText(string())

}

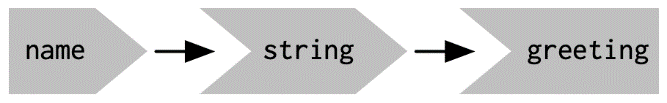


图3.4：由于将输入连接到输出，因此在反应表达式的两侧都绘制了尖角。

反应式表达式接受输入并产生输出，因此它们具有结合了输入和输出特征的形状。希望这些形状能帮助您记住组件是如何装配在一起的。

**3.3.5执行顺序**

理解代码运行的顺序完全由反应图决定，这一点很重要。这与大多数R代码不同，后者的执行顺序由代码行的顺序决定。例如，我们可以翻转这个简单服务器函数中两行的顺序：

server <- function(input, output, session) {

output$greeting <- renderText(string())

string <- reactive(paste0("Hello ", input$name, "!"))

}

您可能认为这会产生错误，因为output$greeting引用了一个尚未创建的反应式表达式string。但请记住，Shiny是懒惰的，因此代码仅在会话启动时运行，这时string已创建了。

相反，这段代码生成与上面相同的反应图，因此代码的运行顺序完全相同。这样组织代码对人类来说是令人困惑的，最好避免这样。相反，请确保反应式表达式和输出仅引用前面已定义的内容，而不是后面定义的内容[[7]](#footnote-7)。这将使您的代码更容易理解。

这个概念非常重要，与大多数其他R代码不同，因此我要再次说明：反应代码的运行顺序仅由反应图决定，而不是由其在服务器函数中的位置决定。

**3.3.6练习**

1、给定此UI：

ui <- fluidPage(

textInput("name", "What's your name?"),

textOutput("greeting")

)

修复以下三个服务器函数中的每个函数中发现的简单错误。首先，试着通过阅读代码来发现问题；然后运行代码以确保已修复它。

server1 <- function(input, output, server) {

input$greeting <- renderText(paste0("Hello ", name))

}

server2 <- function(input, output, server) {

greeting <- paste0("Hello ", input$name)

output$greeting <- renderText(greeting)

}

server3 <- function(input, output, server) {

output$greting <- paste0("Hello", input$name)

}

2、绘制以下服务器函数的反应图：

server1 <- function(input, output, session) {

c <- reactive(input$a + input$b)

e <- reactive(c() + input$d)

output$f <- renderText(e())

}

server2 <- function(input, output, session) {

x <- reactive(input$x1 + input$x2 + input$x3)

y <- reactive(input$y1 + input$y2)

output$z <- renderText(x() / y())

}

server3 <- function(input, output, session) {

d <- reactive(c() ^ input$d)

a <- reactive(input$a \* 10)

c <- reactive(b() / input$c)

b <- reactive(a() + input$b)

}

3、为什么这个代码会失败？

var <- reactive(df[[input$var]])

range <- reactive(range(var(), na.rm = TRUE))

**3.4反应表达式**

我们已经快速浏览了几次反应表达式，所以希望您能够理解它们可能会做什么。现在，我们将深入了解更多细节，并说明它们在构建真正的应用程序时为何如此重要。

反应表达式之所以重要，是因为它们提供了更多的信息，因此当输入发生变化时，它可以进行更少的重新计算，从而提高应用程序的效率，并且通过简化反应图，使人类更容易理解应用程序。反应表达式同时具有输入和输出的味道：

1、与输入一样，您可以在输出中使用反应表达式的结果。

2、与输出一样，反应表达式依赖于输入，并自动知道何时需要更新。

这种二元性意味着我们需要一些新的词汇：我将使用生产者来指代反应式输入和表达式，而用消费者来指代反应表达式和输出。图3.5用维恩图显示了这种关系。

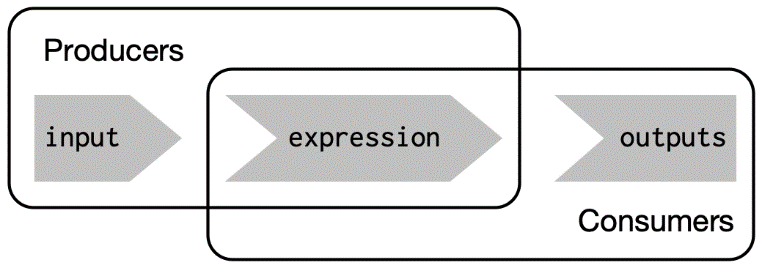


图3.5：输入和表达式为反应式生产者；表达式和输出是反应式消费者

我们需要一个更复杂的应用程序来了解使用反应表达式的好处。首先，我们将通过定义一些常规的R函数来为我们的应用程序提供动力。

**3.4.1动机**

想象一下，我想用一个图和一个假设检验来比较两个相似的数据集。我做了一些实验，得到了以下函数：freqpoly()用频率分布图[[8]](#footnote-8)可视化两个分布，t\_test()使用t检验比较均值，并用字符串总结结果：

library(ggplot2)

freqpoly <- function(x1, x2, binwidth = 0.1, xlim = c(-3, 3)) {

df <- data.frame(

x = c(x1, x2),

g = c(rep("x1", length(x1)), rep("x2", length(x2)))

)

ggplot(df, aes(x, colour = g)) +

geom\_freqpoly(binwidth = binwidth, size = 1) +

coord\_cartesian(xlim = xlim)

}

t\_test <- function(x1, x2) {

test <- t.test(x1, x2)

# use sprintf() to format t.test() results compactly

sprintf(

"p value: %0.3f\n[%0.2f, %0.2f]",

test$p.value, test$conf.int[1], test$conf.int[2]

)

}

如果我有一些相似的数据，我可以使用这些函数来比较两个变量：

x1 <- rnorm(100, mean = 0, sd = 0.5)

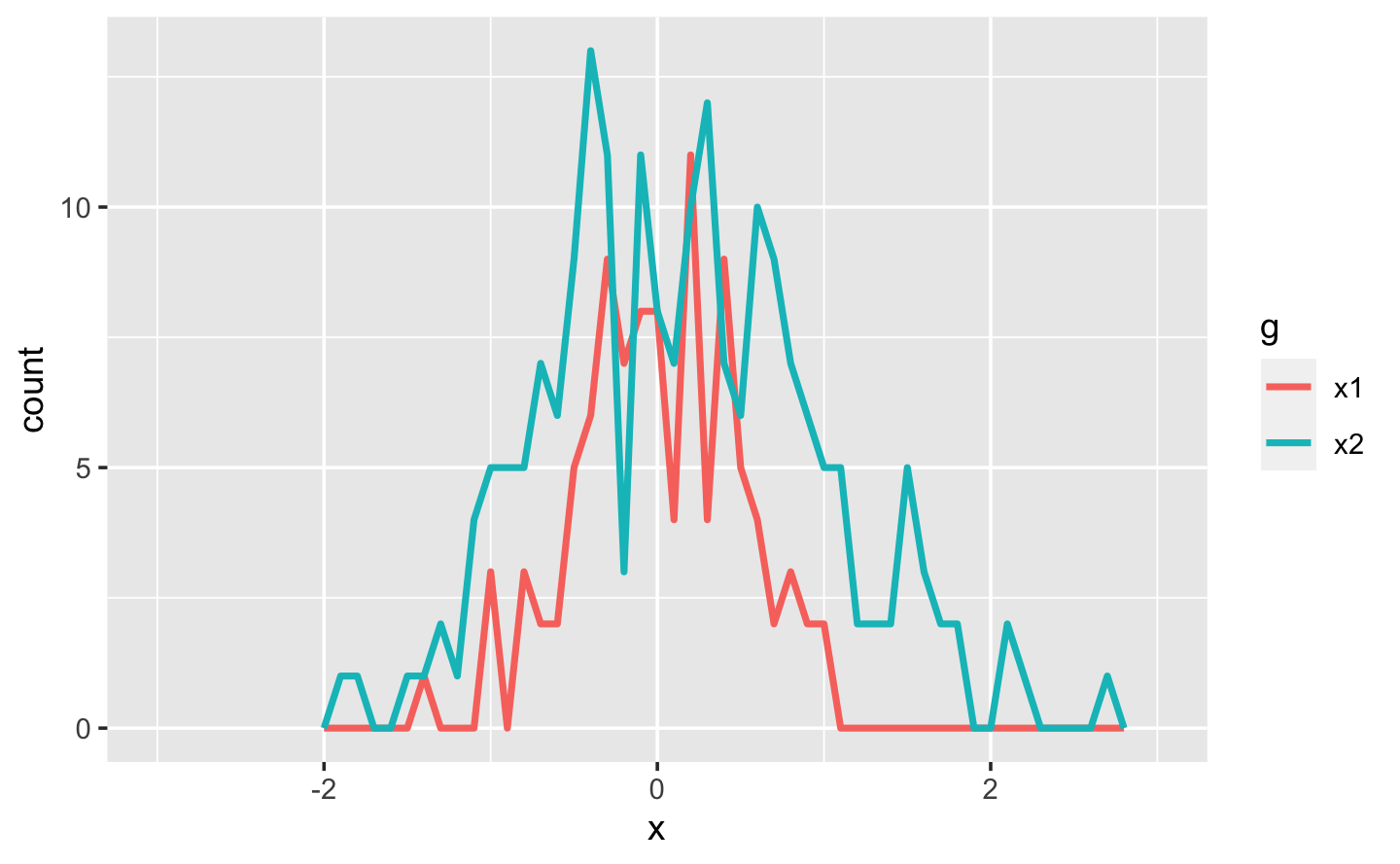
x2 <- rnorm(200, mean = 0.15, sd = 0.9)

freqpoly(x1, x2)

cat(t\_test(x1, x2))

#> p value: 0.047

#> [-0.30, -0.00]



在实际分析中，在编好这些函数之前，您可能已经进行了大量的探索。我跳过了这里的探索，以便我们可以尽快讨论应用程序。但是，将命令式代码提取到常规函数中是所有Shiny应用程序的一项重要技术：从中提取的代码越多，应用程序就越容易理解。这是好的软件工程实践，因为它有助于隔离关注点：应用程序外部的函数专注于计算，因而应用程序内部的代码可以专注于响应用户操作。我们将在第18章再次讨论这个想法。

**3.4.2应用程序**

我想使用这两个工具快速探索一系列模拟。Shiny应用程序是一个很好的方法，因为它可以避免繁琐地修改和重新运行R代码。下面，我将这些片段包装到一个Shiny应用程序中，在那里我可以交互式地调整输入。

让我们从UI开始。在第6.2.3节中，我们将了解fluidRow()和column()的具体操作；但你可以从他们的名字猜出他们的目的😄. 第一行有三列用于输入控件(分布1、分布2和绘图控件)。第二行有一个宽列用于绘图，一个窄列用于假设检验。

ui <- fluidPage(

fluidRow(

column(4,

"Distribution 1",

numericInput("n1", label = "n", value = 1000, min = 1),

numericInput("mean1", label = "µ", value = 0, step = 0.1),

numericInput("sd1", label = "σ", value = 0.5, min = 0.1, step = 0.1)

),

column(4,

"Distribution 2",

numericInput("n2", label = "n", value = 1000, min = 1),

numericInput("mean2", label = "µ", value = 0, step = 0.1),

numericInput("sd2", label = "σ", value = 0.5, min = 0.1, step = 0.1)

),

column(4,

"Frequency polygon",

numericInput("binwidth", label = "Bin width", value = 0.1, step = 0.1),

sliderInput("range", label = "range", value = c(-3, 3), min = -5, max = 5)

)

),

fluidRow(

column(9, plotOutput("hist")),

column(3, verbatimTextOutput("ttest"))

)

)

服务器函数在从指定的分布中绘图后，将对freqpoly()和t\_test()函数的调用组合在一起：

server <- function(input, output, session) {

output$hist <- renderPlot({

x1 <- rnorm(input$n1, input$mean1, input$sd1)

x2 <- rnorm(input$n2, input$mean2, input$sd2)

freqpoly(x1, x2, binwidth = input$binwidth, xlim = input$range)

}, res = 96)

output$ttest <- renderText({

x1 <- rnorm(input$n1, input$mean1, input$sd1)

x2 <- rnorm(input$n2, input$mean2, input$sd2)

t\_test(x1, x2)

})

}

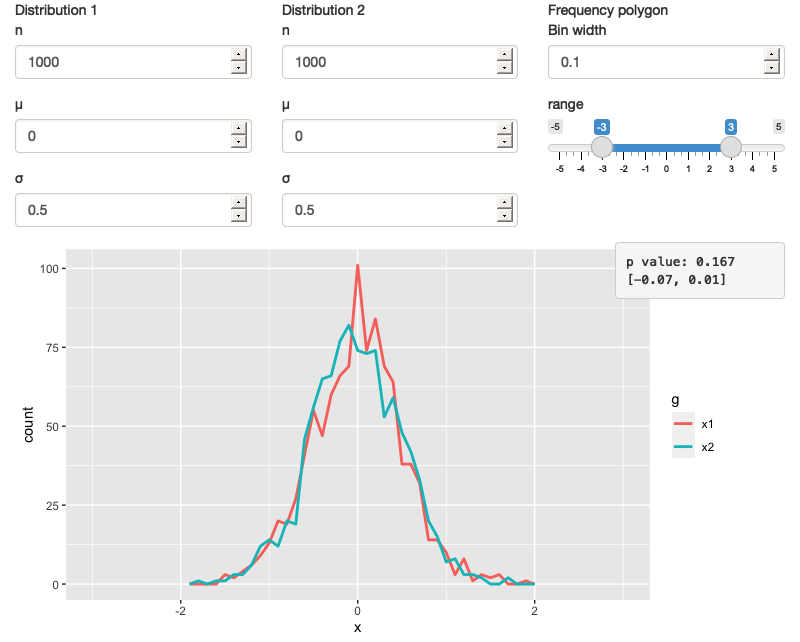


图3.6：一个Shiny应用程序，可以让您使用t检验和频率分布图比较两个相似的分布。在线演示https://hadley.shinyapps.io/ms-case-study-1.

服务器和用户界面的定义生成了如图3.6所示的结果，在线演示可以访问https://hadley.shinyapps.io/ms-case-study-1; 我建议在继续阅读之前打开应用程序并快速转转，以确保您了解其基本运作。

**3.4.3反应图**

让我们从绘制这个应用程序的反应图开始。Shiny足够聪明，只在它所引用的输入发生变化时才更新输出；但还没有聪明到只在输出中有选择地运行代码。换句话说，输出是原子的：它们作为一个整体，要么被执行，要么不是。

例如，看看以下的服务器代码段：

x1 <- rnorm(input$n1, input$mean1, input$sd1)

x2 <- rnorm(input$n2, input$mean2, input$sd2)

t\_test(x1, x2)

作为一名阅读此代码的人，您可以看出，我们只需要在n1、mean1或sd1发生变化时更新x1，而在n2、mean2或sd2发生变化时才需要更新x2。然而，Shiny只将输出视为一个整体，因此每当n1、mean1、sd1、n2、mean2或sd2中的一个发生变化时，它都会更新x1和x2。这导致图3.7所示的反应图：

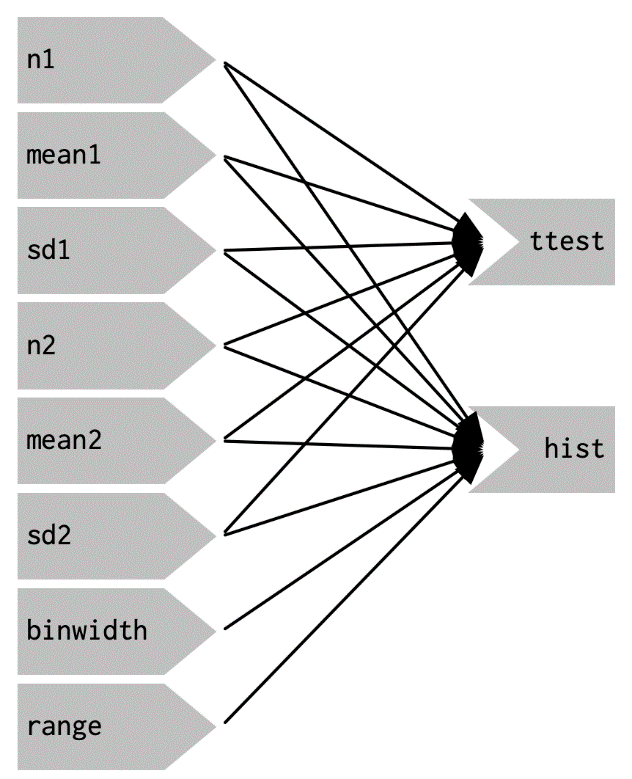


图3.7：反应图显示每个输出引用每个输入

您会注意到该图非常密集：几乎每个输入都直接连接到每个输出。这造成了两个问题：

这个应用程序很难理解，因为有太多的连接。应用程序中没有任何部分可以单独提取和分析。

该应用程序效率低下，因为它所做的工作超出了需要。例如，如果更改绘图的分割区间(binwidth)，将重新计算数据；如果更改n1的值，x2将更新(在两个位置！)。

该应用程序还有一个主要缺陷：频率分布图和t检验使用单独的随机采样。这相当有误导性，因为您希望他们基于相同的数据。

幸运的是，我们可以通过使用反应表达式提取重复的计算来解决所有这些问题。

**3.4.4简化反应图**

在下面的服务器函数中，我们重构现有代码，将重复的代码提取为两个新的反应表达式x1和x2，模拟来自两个分布的数据。为了创建反应表达式，我们调用reactive()并将结果赋给一个变量。稍后引用该表达式时，我们象函数调用一样引用该变量。

server <- function(input, output, session) {

x1 <- reactive(rnorm(input$n1, input$mean1, input$sd1))

x2 <- reactive(rnorm(input$n2, input$mean2, input$sd2))

output$hist <- renderPlot({

freqpoly(x1(), x2(), binwidth = input$binwidth, xlim = input$range)

}, res = 96)

output$ttest <- renderText({

t\_test(x1(), x2())

})

}

这种转换产生了图3.8中所示的基本上更简单的反应图。这个更简单的反应图使您更容易理解应用程序，因为您可以单独理解连接的组件；分布参数的值仅通过x1和x2影响输出。这种重写也使应用程序更高效，因为它的计算量少得多。现在，当您更改binwidth或range时，只有绘图更改，而不是基础数据。

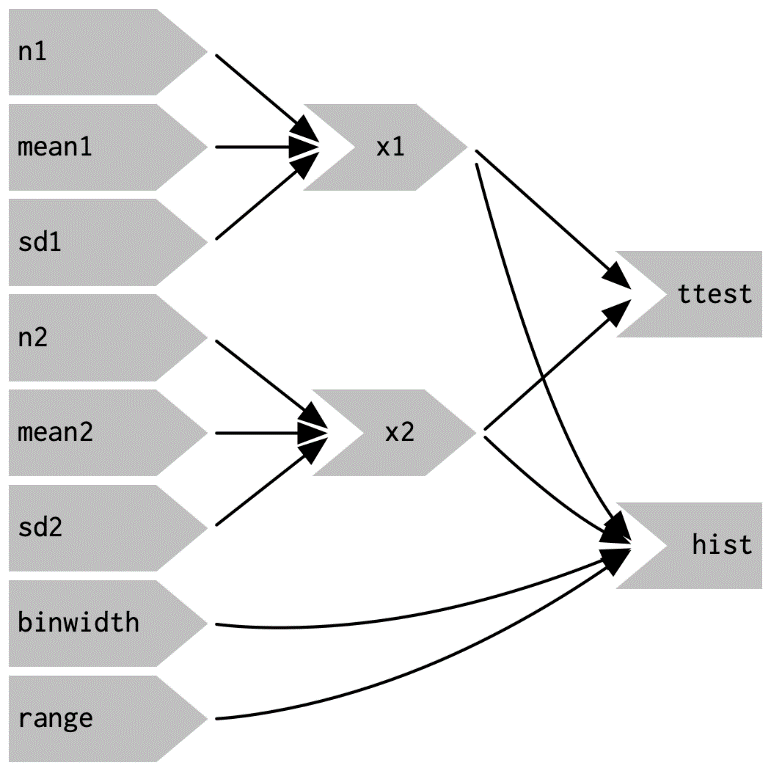


图3.8：使用反应表达式大大简化了反应图，使其更易于理解

为了强调这种模块性，图3.9在独立组件周围绘制了方框。我们将在第19章讨论模块时回到这个概念。模块允许您提取重复代码以供重用，同时保证它与应用程序中的所有其他内容隔离。对于更复杂的应用程序，模块是一种非常有用和强大的技术。

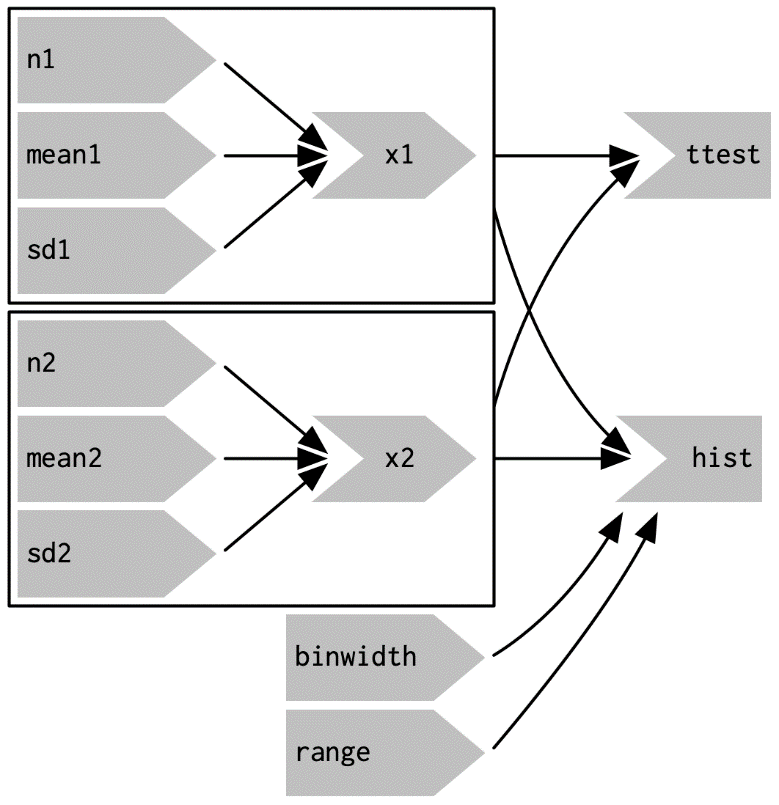


图3.9：模块强制应用程序各部分之间的隔离

您可能熟悉编程的“三法则”：每当您复制和粘贴某个东西三次时，您都应该找出如何减少重复(通常通过编写函数)。这一点很重要，因为它减少了代码中的重复量，这使得代码更容易理解，并且随着需求的变化更容易更新。

然而，在Shiny中，我认为您应该考虑“一法则”：每当您复制和粘贴某个内容一次时，您应该考虑将重复的代码提取到一个反应表达式中。对于Shiny来说，规则更为严格，因为反应表达式不仅让人们更容易理解代码，还提高了Shiny高效重新运行代码的能力。

**3.4.5为什么我们需要反应表达式？**

当您第一次开始使用反应式代码时，您可能想知道为什么我们需要反应表达式。为什么不能使用现有工具来减少代码中的重复：创建新变量和编写函数？不幸的是，这两种技术都不适用于反应式环境。

如果您试图使用变量来减少重复，您可能会编写如下代码：

server <- function(input, output, session) {

x1 <- rnorm(input$n1, input$mean1, input$sd1)

x2 <- rnorm(input$n2, input$mean2, input$sd2)

output$hist <- renderPlot({

freqpoly(x1, x2, binwidth = input$binwidth, xlim = input$range)

}, res = 96)

output$ttest <- renderText({

t\_test(x1, x2)

})

}

如果运行此代码，您将得到一个错误，因为您试图在反应上下文之外访问输入值。即使你没有得到那个错误，你仍然会有一个问题：x1和x2只在会话开始时计算一次，而不是每次更新一个输入时重新计算。

如果您尝试使用一个函数，该应用程序将工作：

server <- function(input, output, session) {

x1 <- function() rnorm(input$n1, input$mean1, input$sd1)

x2 <- function() rnorm(input$n2, input$mean2, input$sd2)

output$hist <- renderPlot({

freqpoly(x1(), x2(), binwidth = input$binwidth, xlim = input$range)

}, res = 96)

output$ttest <- renderText({

t\_test(x1(), x2())

})

}

但它与原始代码有相同的问题：任何输入的改变都会导致重新计算所有输出，t检验和频率分布图将在不同的样本上运行。而反应表达式自动缓存其结果，并且仅在其输入更改时更新[[9]](#footnote-9)。

上面的代码，使用变量只计算一次值(粥太冷)，使用函数每次调用时都计算值(粥太热)，而使用反应表达式仅在值可能发生变化时(粥正好！)才计算值。

**3.5控制执行的时间**

既然您已经熟悉了反应机制的基本概念，我们将讨论两种更高级的技术，它们允许您增加或减少反应表达式的执行频率。在这里，我将展示如何使用基本技术；在第15章中，我们将回到它们的底层实现。

为了探索基本思路，我将简化我的模拟应用程序。我将使用只有一个参数的分布，并强制两个分布样本共享相同的n。我还将删除绘图控件。这将产生更小的UI对象和服务器函数：

ui <- fluidPage(

fluidRow(

column(3,

numericInput("lambda1", label = "lambda1", value = 3),

numericInput("lambda2", label = "lambda2", value = 5),

numericInput("n", label = "n", value = 1e4, min = 0)

),

column(9, plotOutput("hist"))

)

)

server <- function(input, output, session) {

x1 <- reactive(rpois(input$n, input$lambda1))

x2 <- reactive(rpois(input$n, input$lambda2))

output$hist <- renderPlot({

freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))

}, res = 96)

}

这将生成图3.10所示的应用程序和图3.11所示的反应图。

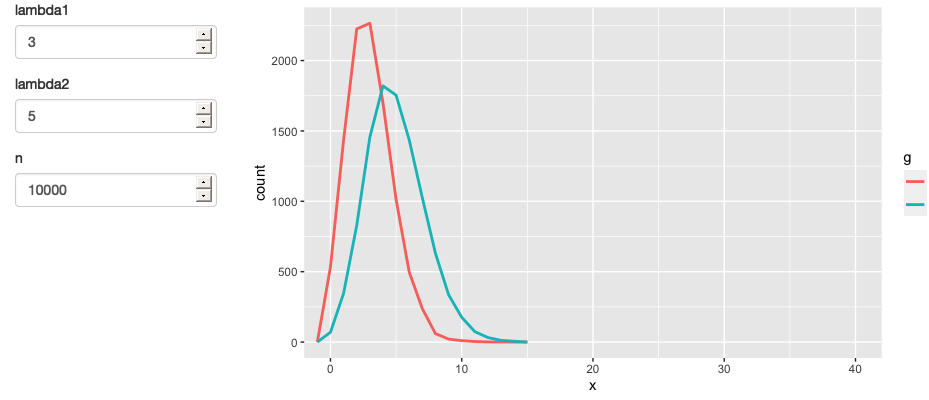


图3.10：一个更简单的应用程序，显示从两个泊松分布中提取的随机数的频率分布图。在线演示访问<https://hadley.shinyapps.io/ms-simulation-2>。

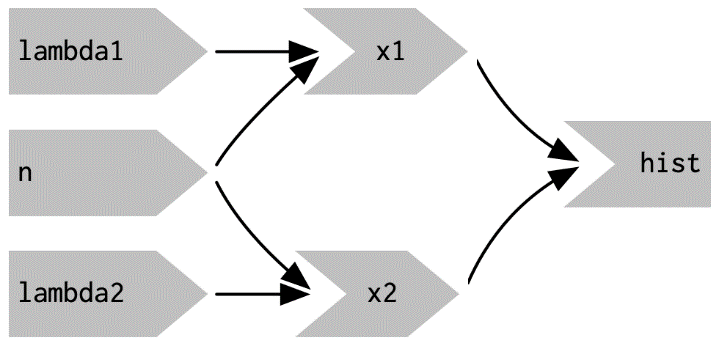


图3.11：反应图

**3.5.1定时失效**

想象一下，您想通过不断重新生成模拟数据来强化这一事实，这是模拟数据，这样您就可以看到动画而不是静态绘图[[10]](#footnote-10)。我们可以使用一个新函数：reactiveTimer()来增加更新频率。

reactiveTimer()是一个反应表达式，它依赖于一个隐藏的输入：当前时间。当您希望反应表达式比其他方式更频繁地使自身无效时，可以使用reactiveTimer()。例如，以下代码使用500毫秒的间隔，因此绘图将每秒更新两次。这足够快，可以提醒您正在看的是模拟数据，而不会因快速变化而让您晕眩。这个变化产生了图3.12所示的反应图。

server <- function(input, output, session) {

timer <- reactiveTimer(500)

x1 <- reactive({

timer()

rpois(input$n, input$lambda1)

})

x2 <- reactive({

timer()

rpois(input$n, input$lambda2)

})

output$hist <- renderPlot({

freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))

}, res = 96)

}

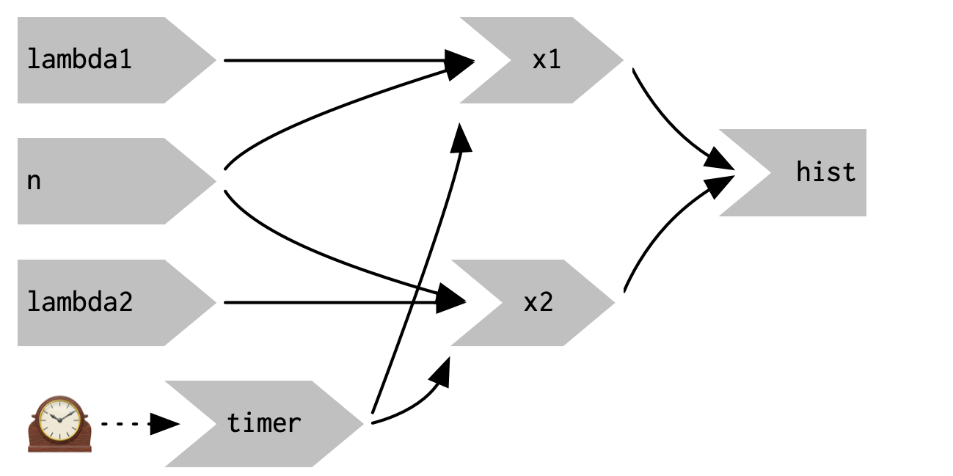


图3.12:reactiveTimer(500)引入了一个新的反应输入，每半秒自动失效一次

注意我们如何在计算x1()和x2()的反应表达式中使用timer()：我们调用它，但不使用返回的值。这使得x1和x2对计时器具有反应依赖，而不必在乎它返回的值到底是什么。

**3.5.2点击**

在上面的场景中，考虑一下如果模拟代码需要运行1秒会发生什么。我们每0.5秒执行一次模拟，因此Shiny将有越来越多的工作要做，永远无法赶上。如果有人在你的应用程序中快速点击按钮，而你所做的计算相对昂贵，同样的问题也会发生。有可能为Shiny造成一个大的工作积压，当它处理积压时，它无法响应任何新事件。这会导致较差的用户体验。

如果您的应用程序中出现这种情况，您可能需要用户通过单击按钮来选择执行昂贵的计算。这是actionButton()的一个很好的用例：

ui <- fluidPage(

fluidRow(

column(3,

numericInput("lambda1", label = "lambda1", value = 3),

numericInput("lambda2", label = "lambda2", value = 5),

numericInput("n", label = "n", value = 1e4, min = 0),

actionButton("simulate", "Simulate!")

),

column(9, plotOutput("hist"))

)

)

要使用动作按钮，我们需要学习一种新工具。为了了解原因，让我们首先使用与上面相同的方法解决这个问题。如上所述，我们引用simulate，但不使用其值以便对其建立反应依赖。

server <- function(input, output, session) {

x1 <- reactive({

input$simulate

rpois(input$n, input$lambda1)

})

x2 <- reactive({

input$simulate

rpois(input$n, input$lambda2)

})

output$hist <- renderPlot({

freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))

}, res = 96)

}

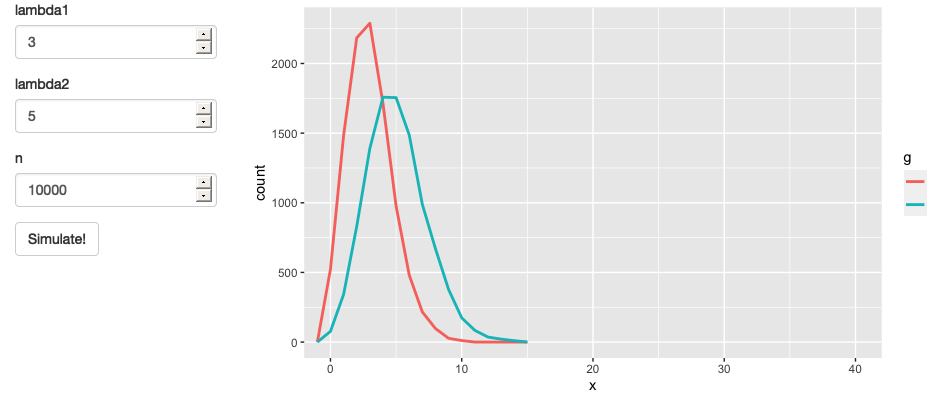


图3.13：带有动作按钮的应用程序。在线演示<https://hadley.shinyapps.io/ms-action-button>。

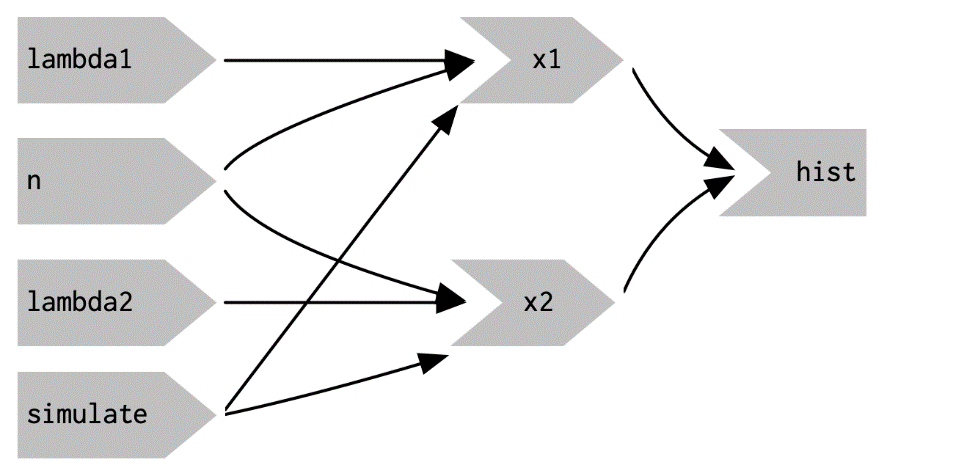


图3.14：这个反应图没有实现我们的目标；我们添加了一个依赖项，而不是替换现有的依赖项。

这产生了图3.13中的应用程序和图3.14中的反应图。这并没有达到我们的目标，因为它只是引入了一个新的依赖关系：当我们单击“模拟”按钮时，x1()和x2()将更新，但当lambda1、lambda2或n发生变化时，它们也将继续更新。我们希望替换现有的依赖项，而不是添加到它们。

为了解决这个问题，我们需要一种新的工具：一种引用输入值而不依赖于它们的方法。我们需要eventReactive()，它有两个参数：第一个参数指定依赖于什么，第二个参数指定计算什么。这允许该应用程序在单击“模拟”时仅计算x1()和x2()：

server <- function(input, output, session) {

x1 <- eventReactive(input$simulate, {

rpois(input$n, input$lambda1)

})

x2 <- eventReactive(input$simulate, {

rpois(input$n, input$lambda2)

})

output$hist <- renderPlot({

freqpoly(x1(), x2(), binwidth = 1, xlim = c(0, 40))

}, res = 96)

}

图3.15显示了新的反应图。请注意，如我们所愿，x1和x2不再对lambda1、lambda2和n具有反应依赖：更改它们的值不会触发计算。我将箭头保留为浅灰色只是为了提醒您x1和x2继续使用这些值，但不再对它们产生反应依赖。

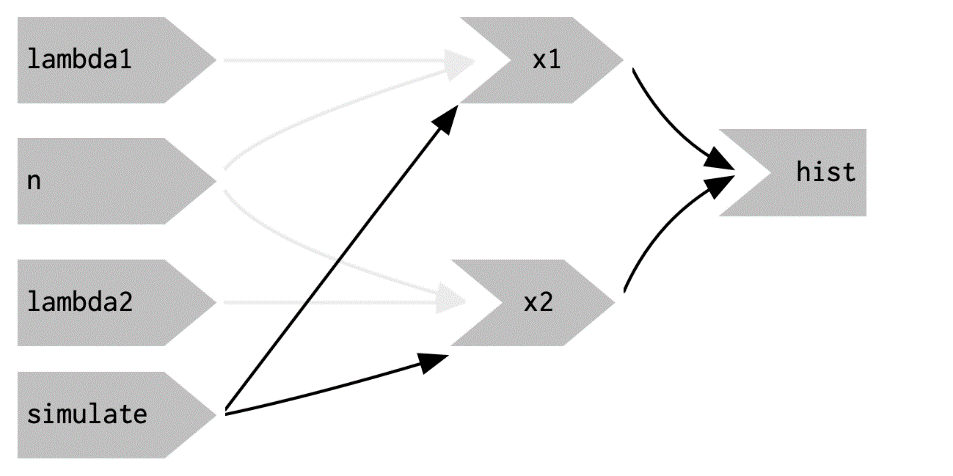


图3.15:eventReactive()可以将依赖性(黑色箭头)与用于计算结果的值(浅灰色箭头)分开。

**3.6观察器**

到目前为止，我们关注的是应用程序内部发生的事情。但有时你需要触达应用程序之外，并导致副作用在世界其他地方发生。这可能是将文件保存到共享网络驱动器、将数据发送到web API、更新数据库，或者(最常见的)将调试消息打印到控制台。这些操作不会影响应用程序的外观，因此不应使用输出和渲染函数。相反，您需要使用观察器。

创建观察器有多种方法，稍后我们将在第15.3节中讨论。目前，我想向您展示如何使用observeEvent()，因为它在您开始学习Shiny时为您提供了一个重要的调试工具。

observeEvent()与eventReactive()非常相似。它有两个重要参数：eventExpr和handlerExpr。第一个参数是要依赖的输入或表达式；第二个参数是将要运行的代码。例如，对server()的以下修改意味着每次更新名字时，都会向控制台发送一条消息：

ui <- fluidPage(

textInput("name", "What's your name?"),

textOutput("greeting")

)

server <- function(input, output, session) {

string <- reactive(paste0("Hello ", input$name, "!"))

output$greeting <- renderText(string())

observeEvent(input$name, {

message("Greeting performed")

})

}

observeEvent()和eventReactive()之间有两个重要区别：

1、您没有将observeEvent()的结果赋给变量，因此

2、你不能从其他反应式消费者那里引用它。

观察器与输出密切相关。您可以认为输出是具有特殊的副作用的观察器：在用户的浏览器中更新HTML。为了强调这种紧密性，我们将在反应图中以相同的方式绘制它们。这产生了如图3.16所示的以下反应图。

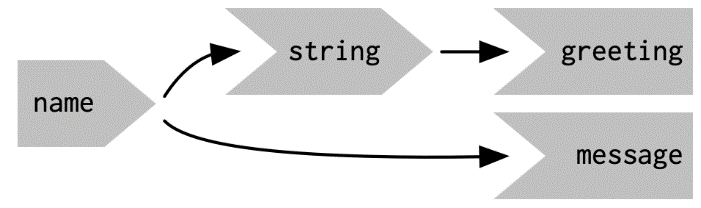


图3.16：在反应图中，观察器看起来与输出相同

**3.7小结**

本章应该已经提高了您对Shiny应用后端的理解，响应用户操作的server()代码。您还迈出了掌握支持Shiny的反应式编程范式的第一步。你在这里学到的东西将带你走很长的路；我们将在第13章回到基本理论。反应机制非常强大，但它与您最习惯的命令式R编程风格也有很大不同。如果需要一段时间才能理解所有后果，请不要感到惊讶。

本章总结了我们对Shiny基础的概述。下一章将通过创建一个更大的Shiny应用程序来支持数据分析，从而帮助您练习到目前为止看到的材料。

1. 与Shiny应用程序的每个连接都会启动一个新会话，无论是来自不同人员的连接，还是来自同一个人的多个选项卡。 [↑](#footnote-ref-1)
2. 主要的例外是有些工作可以在多个用户之间共享。

   例如，所有用户可能都在查看同一个大型csv文件，因此您最好加载一次并在用户之间共享。

   我们将在第23.6.1节中回到这个想法。 [↑](#footnote-ref-2)
3. 如果您正在运行实时应用程序，请注意，您必须非常缓慢地敲键盘，以便输出一次更新一个字母。这是因为Shiny使用了一种称为**去抖动**的技术，这意味着它在发送更新之前要等待几毫秒。

   这大大减少了Shiny需要做的工作量，而不会明显减少应用程序的响应时间。 [↑](#footnote-ref-3)
4. <https://xkcd.com/149/> [↑](#footnote-ref-4)
5. 如果您曾经努力让ggplot2图例看起来与您想要的完全一样，那么您遇到过这个问题！ [↑](#footnote-ref-5)
6. 是的，如果您在浏览器中看不到，Shiny不会更新输出！

   Shiny非常懒惰，除非你能真正看到结果，否则它不会工作。 [↑](#footnote-ref-6)
7. 这种排序的技术术语是“拓扑排序”。 [↑](#footnote-ref-7)
8. 如果您以前没有听说过频率分布图，它只是一个用直线而不是条形绘制的直方图，这使得在同一个图上比较多个数据集更容易。 [↑](#footnote-ref-8)
9. 如果你熟悉记忆，这是一个类似的想法。 [↑](#footnote-ref-9)
10. 《纽约时报》在讨论如何解读乔布斯报告的文章中特别有效地使用了这一技术： <https://www.nytimes.com/2014/05/02/upshot/how-not-to-be-misled-by-the-jobs-report.html> [↑](#footnote-ref-10)