

第一部分 Getting started

Introduction

接下来四章的目标是让你尽快编写Shiny的应用程序。在第1章中，我将从小而完整的部分开始，向您展示应用程序的所有主要部分以及它们是如何组合在一起的。然后在第2章和第3章中，您将开始了解Shiny应用程序的两个主要部分的细节：前端（用户在浏览器中看到的内容）和后端（使其全部工作的代码）。我们将在第4章中以一个案例研究结束，以帮助巩固您迄今为止学到的概念。

Chapter 1. Your first Shiny app

1.1 Introduction

在本章中，我们将创建一个简单的Shiny应用程序。首先，我将向您展示Shiny应用程序所需的最小模版，然后您将学习如何启动和停止它。接下来，您将学习每个Shiny应用的两个关键组件：定义应用程序外观的UI（用户界面的缩写, user interface）和定义应用程序工作方式的函数（server function）。Shiny使用反应式编程（reactive programming）在输入更改时自动更新输出，因此我们将通过学习Shiny应用程序的第三个重要组件：反应式表达式，来结束本章。

如果您还没有安装Shiny，请使用以下软件立即安装：

```
install.packages("shiny")
```

如果您已经安装了Shiny，请使用packageVersion（“shiny”）检查您是否安装了1.5.0或更高版本。

然后加载到您当前的R会话中：

```
library(shiny)
```

1.2 Create app directory and file

有几种方法可以创建Shiny应用程序。最简单的方法是为你的应用程序创建一个新目录，并在其中放一个名为“app.R”的文件。这个“app.R”文件将用于告诉Shiny，你的应用应该是什么样子，以及它应该如何表现。

尝试创建一个新目录，并添加一个“app.R”文件，如下所示：

```
library(shiny)
ui <- fluidPage(
  "Hello, world!"
)
server <- function(input, output, session) {
}
shinyApp(ui, server)
```

这是一个完整的Shiny应用程序！仔细看上面的代码，我们的“app.R”做了四件事：

1. 它调 library(shiny) 来加载 shiny 的包。

2. 它定义了用户界面，即人类与之交互的HTML网页。在这种情况下，它是一个包含单词“Hello, world!”的页面。
3. 它通过定义了一个server函数，来指定我们的应用程序的行为。它目前是空的，所以我们的应用程序什么都不做，但我们很快就会回来重新访问它。
4. 它执行 `shinyApp(ui, server)`，从 UI 和 server 构建并启动一个Shiny应用程序。

RStudio提示：在RStudio中创建新应用程序有两种便捷的方法：

- 通过单击“文件|新项目”，然后选择“新目录”和“闪亮的Web应用程序”，一步创建一个包含基本应用程序的新目录和 `app.R` 文件。
- 如果您已经创建了 `app.R` 文件，您可以通过键入“shinyapp”并按Shift+Tab快速添加应用程序样板。

1.3 Running and stopping

有几种方法可以运行此应用程序：

- 单击文档工具栏中的**Run App**（图1.1）按钮。
- 使用键盘快捷键: `Cmd/Ctrl + Shift + Enter`。
- 如果您不使用RStudio，您可以 `(source())` 整个文档，或调用 `shiny::runApp()` 与包含 `app.R` 的目录的路径。

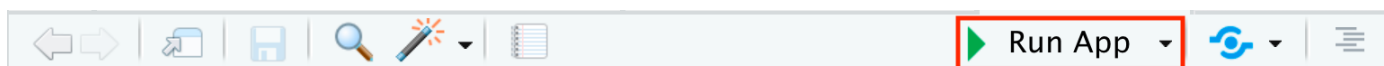


图1.1：在源窗格的右上角可以找到“运行应用程序”按钮。

选择这些选项之一，并检查您是否看到与图1.2中相同的应用程序。恭喜！你已经制作了你的第一个Shiny应用程序。

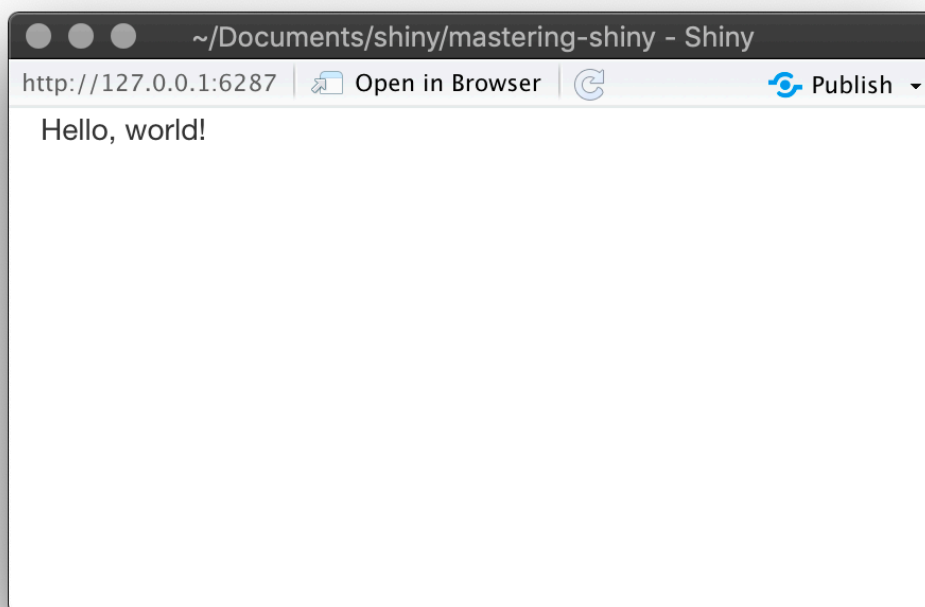


图1.2：运行上述代码时，您将看到非常基本的闪亮应用程序

在关闭应用程序之前，请返回RStudio并查看R控制台。你会注意到如下信息：

```
#> Listening on http://127.0.0.1:3827
```

这会告诉您，您可以找到此应用程序的URL: 127.0.0.1，它是一个标准地址，指向“这台计算机”，3827是一个随机分配的端口号。您可以将该URL输入到任何兼容的网络浏览器中，以打开应用程序的另一个副本。

请注意，R正在运行shiny应用程序，因此R提示符不可见。同时，Rstudio控制台工具栏显示停止标志图标。当Shiny应用程序运行时，它会“屏蔽”R控制台。这意味着在Shiny应用程序停止之前，您无法在R控制台运行新命令。

您可以使用以下任一选项停止应用程序并返回控制台的访问权限：

- 单击Rstudio控制台工具栏上的停止标志图标。
- 单击控制台，然后按 `Esc` (或，如果没有使用Rstudio，则按 `Ctrl + C`)。
- 关闭Shiny的应用程序窗口。

Shiny应用程序开发的基本工作流程是编写一些代码，启动应用程序，运行应用程序，再写一些代码，然后重复。如果你正在使用RStudio，你甚至不需要停止并重新启动应用程序来查看你的更改—你可以按下工具箱中的“重新加载应用程序 (Reload app)”按钮，也可以使用 `Cmd/Ctrl+Shift+Enter` 键盘快捷键。我将在第5章中介绍其他工作流模式。

1.4 Adding UI controls

接下来，我们将在我们的用户界面中添加一些输入和输出，这样就不会那么少了。我们将制作一个非常简单的应用程序，向您显示数据集包中包含的所有内置数据。

用以下代码替换您的 `ui`：

```
ui <- fluidPage(  
  selectInput("dataset", label = "Dataset", choices = ls("package:datasets")),  
  verbatimTextOutput("summary"),  
  tableOutput("table")  
)
```

这个例子使用了四个新功能：

- `fluidPage()` 是一个布局函数，可以设置页面的基本视觉结构。您将在第6.2节中了解有关它们的更多信息。
- `selectInput()` 是一个输入控件，允许用户通过提供值与应用程序交互。在这种情况下，它是一个带有“数据集”标签的选择框，允许您选择R附带的内置数据集之一。您将在第2.2节中了解有关输入的更多信息。
- `verbatimTextOutput()` 和 `tableOutput()` 是输出控件，它们告诉Shiny将渲染输出放在哪里（我们稍后将讨论如何）。`verbatimTextOutput()` 显示代码，`tableOutput()` 则显示表。您将在第2.3节中了解更多关于输出的信息。

布局函数、输入和输出有不同的用途，但它们的运行逻辑基本相同：它们都只是生成HTML的花哨方式方法，如果您在Shiny应用程序之外调用它们中的任何一个，您将在控制台上看到HTML打印出来。不要害怕！四处看看这些不同的布局和控件是如何工作的。

继续，再次运行该应用程序。您现在将看到图1.3，一个包含选择框的页面。我们只看到输入，而不是两个输出，因为我们还没有告诉Shiny输入和输出之间的关系。

Dataset

ability.cov

图1.3：带有UI的数据集应用程序

1.5 Adding behavior

接下来，我们将通过在服务函数（server function）中定义输出，使输出栩栩如生。

Shiny使用反应式编程使应用程序具有交互性。您将在第3章中了解有关反应式编程的更多信息，但现在请注意，它涉及告诉Shiny如何执行计算，而不是命令Shiny实际执行。这就像给某人一个食谱和要求他们给你做三明治的区别。

我们将通过提供这些输出的“食谱”来告诉Shiny如何在示例应用程序中填写 `summary` 和 `table` 输出。用这个替换你的空 `server` 函数：

```
server <- function(input, output, session) {
  output$summary <- renderPrint({
    dataset <- get(input$dataset, "package:datasets")
    summary(dataset)
  })

  output$table <- renderTable({
    dataset <- get(input$dataset, "package:datasets")
    dataset
  })
}
```

赋值运算符 (`<-`) 的左侧, `output$ID`, 表示您正在提供具有该ID的Shiny输出的方法。赋值运算符的右侧使用特定的**渲染函数 (render function)** 来包装您提供的一些代码。每个 `render{Type}` 函数旨在产生特定类型的输出 (例如文本、表格和绘图), 并且通常与 `{type}Output` 函数配对。例如, 在此应用程序中, `renderPrint()` 与 `verbatimTextOutput()` 配对, 以显示具有固定宽度 (逐字) 文本的统计摘要, `renderTable()` 与 `tableOutput()` 配对, 以显示表中的输入数据。

再次运行应用程序, 观察当您更改输入时, 输出会发生什么。图1.4显示了您打开应用程序时应该看到的内容。

Dataset
ability.cov

	Length	Class	Mode
cov	36	-none-	numeric
center	6	-none-	numeric
n.obs	1	-none-	numeric

cov.general	cov.picture	cov.blocks	cov.maze	cov.reading	cov.vocab	center	n.obs
24.64	5.99	33.52	6.02	20.75	29.70	0.00	112.00
5.99	6.70	18.14	1.78	4.94	7.20	0.00	112.00
33.52	18.14	149.83	19.42	31.43	50.75	0.00	112.00
6.02	1.78	19.42	12.71	4.76	9.07	0.00	112.00
20.75	4.94	31.43	4.76	52.60	66.76	0.00	112.00
29.70	7.20	50.75	9.07	66.76	135.29	0.00	112.00

图1.4: 现在我们已经提供了一个连接输出和输入的服务器功能, 我们有一个功能齐全的应用程序

请注意, 每当您更改输入数据集时, `summary` 和 `table` 都会更新。此依赖项是隐式创建的, 因为我们在输出函数中引用了 `input$dataset`。 `input$dataset` 填充了带有id为 `dataset` 的UI组件的当前值, 并且每当该值发生变化时, 输出将自动更新。这是**反应性**的本质: 当输入发生变化时, 输出会自动反应 (重新计算)。

1.6 Reducing duplication with reactive expressions

即使在这个简单的示例中, 我们也有一些重复的代码: 以下行存在于两个输出中。

```
dataset <- get(input$dataset, "package:datasets")
```

在每种编程中, 复制代码都是不当的做法; 它可能在计算上是浪费的, 更重要的是, 它增加了维护或调试代码的难度。这里没那么重要, 但我想在非常简单的语境中说明基本想法。

在传统的R脚本中，我们使用两种技术来处理重复的代码：要么我们使用变量捕获值，要么使用函数捕获计算。不幸的是，这些方法都不起作用，原因您将在第[13.2](#)节中了解，我们需要一种新的机制：**反应式（reactive expressions）**。

您通过使用 `reactive({...})` 包装代码块并将其分配给变量来创建反应式表达式，并通过像函数一样调用它来使用反应式表达式。但是，虽然它看起来像是在调用一个函数，但反应式表达式有一个重要的区别：它只在第一次调用时运行，然后缓存其结果，直到它需要更新。

我们可以更新我们的 `server()` 以使用反应式表达式，如下所示。该应用程序的行为相同，但工作效率更高一点，因为它只需要检索一次数据集，而不是两次。

```
server <- function(input, output, session) {  
  # Create a reactive expression  
  dataset <- reactive({  
    get(input$dataset, "package:datasets")  
  })  
  
  output$summary <- renderPrint({  
    # Use a reactive expression by calling it like a function  
    summary(dataset())  
  })  
  
  output$table <- renderTable({  
    dataset()  
  })  
}
```

我们将多次回到反应式编程，但即使拥有对输入、输出和反应式表达式的粗略知识，也有可能构建非常有用的 Shiny 应用程序！

1.7 Summary

在本章中，您创建了一个简单的应用程序——它不是很令人兴奋或有用，但看到了使用您现有的 R 知识构建 Web 应用程序是多么容易。在接下来的两章中，您将了解有关用户界面和反应式编程的更多信息，这是 Shiny 的两个基本构建块。现在是获取[Shiny cheatsheet](#)的好时机。这是一个很好的资源，可以帮助您记住 Shiny 应用程序的主要组件。

Shiny : : CHEAT SHEET

Basics

A **Shiny** app is a web page (UI) connected to a computer running a live R session (**Server**)



Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){
  shinyApp(ui = ui, server = server)
}
```

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines ui and server into an app. Wrap with **runApp()** if calling from a sourced script or inside a function.

SHARE YOUR APP

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at <http://shinyapps.io>
2. Click the **Publish** icon in the RStudio IDE or run:
`rsconnect::deployApp("path to directory")`

Build or purchase your own Shiny Server at www.rstudio.com/products/shiny-server/



Building an App

Complete the template by adding arguments to **fluidPage()** and a body to the server function.

Add inputs to the UI with 'input()' functions.

Add outputs with 'Output()' functions.

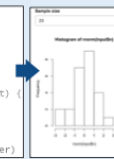
Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with **output\$<id>**
2. Refer to inputs with **input\$<id>**
3. Wrap code in a **render***() function before saving to output

Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

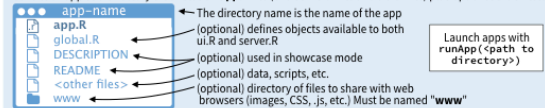


ui.R contains everything you would save to ui.

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that holds an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.



Outputs - render*() and 'Output()' functions work together to add R output to the UI

DT::renderDataTable (expr, options, callback, escape, env, quoted)	dataTableOutput (outputId, icon, ...)
renderImage (expr, env, quoted, deleteFile)	imageOutput (outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)
renderPlot (expr, width, height, res, ..., env, quoted, func)	plotOutput (outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)
renderPrint (expr, env, quoted, func, width)	verbatimTextOutput (outputId)
renderTable (expr, ..., env, quoted, func)	tableOutput (outputId)
renderText (expr, env, quoted, func)	textOutput (outputId, container, inline)
renderUI (expr, env, quoted, func)	uiOutput (outputId, inline, container, ...)
	htmlOutput (outputId, inline, container, ...)

Inputs

collect values from the user

Access the current value of an input object with **input\$<inputid>**. Input values are **reactive**.

actionButton(inputId, label, icon, ...)

actionLink(inputId, label, icon, ...)

checkboxGroupInput(inputId, label, choices, selected, inline)

checkboxInput(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)

fileInput(inputId, label, multiple, accept)

numericInput(inputId, label, value, min, max, step)

passwordInput(inputId, label, value)

radioButtons(inputId, label, choices, selected, inline)

selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also **selectizeInput()**)

sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)

submitButton(text, icon) (Prevents reactions across entire app)

textInput(inputId, label, value)

图1.5: Shiny cheatsheet, 可从<https://www.rstudio.com/resources/cheatsheets>获取

1.8 Exercises

1. 创建一个以名字问候用户的应用程序。你还不知道做这个所需的所有功能，所以我在下面包括了一些代码行。考虑您将使用哪些行，然后将它们复制并粘贴到Shiny应用程序的正确位置。

```
tableOutput("mortgage")
output$greeting <- renderText({
  paste0("Hello ", input$name)
})
numericInput("age", "How old are you?", value = NA)
textInput("name", "What's your name?")
textOutput("greeting")
output$histogram <- renderPlot({
  hist(rnorm(1000))
}, res = 96)
```

2. 假设你的朋友想设计一个应用程序，允许用户在1到50之间设置一个数字（x），并显示将这个数乘以5的结果。这是他们的第一次尝试：

```
library(shiny)

ui <- fluidPage(
  sliderInput("x", label = "If x is", min = 1, max = 50, value = 30),
```



```

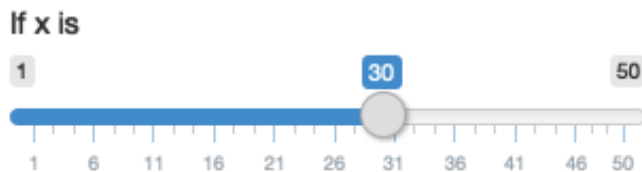
"then x times 5 is",
textOutput("product")
)

server <- function(input, output, session) {
  output$product <- renderText({
    x * 5
  })
}

shinyApp(ui, server)

```

但不幸的是，它有一个错误：

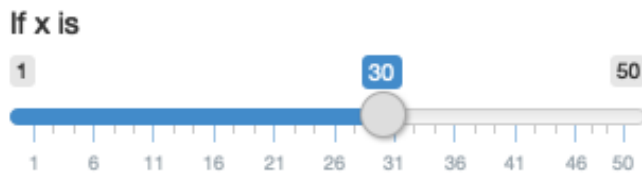


then x times 5 is

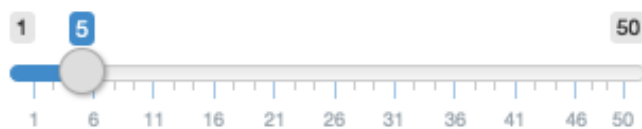
Error: object 'x' not found

你能帮助他们找到并纠正错误吗？

3. 从上一个练习中扩展应用程序，允许用户设置乘数的值 y ，以便应用程序产生 $x * y$ 的值。最终结果应该是这样的：



and y is



then, x times y is

150

4. 以以下应用程序为例，该应用程序为上一个练习中描述的最后一个应用程序添加了一些附加功能。新功能如何通过使用反应式表达式来减少应用程序中重复的代码数量。

```

library(shiny)

ui <- fluidPage(
  sliderInput("x", "If x is", min = 1, max = 50, value = 30),
  sliderInput("y", "and y is", min = 1, max = 50, value = 5),
  "then, (x * y) is", textOutput("product"),
  "and, (x * y) + 5 is", textOutput("product_plus5"),
  "and (x * y) + 10 is", textOutput("product_plus10")
)

```



```

server <- function(input, output, session) {
  output$product <- renderText({
    product <- input$x * input$y
    product
  })
  output$product_plus5 <- renderText({
    product <- input$x * input$y
    product + 5
  })
  output$product_plus10 <- renderText({
    product <- input$x * input$y
    product + 10
  })
}

shinyApp(ui, server)

```

5. 以下应用程序与您在本章前面看到的应用程序非常相似：您从软件包中选择一个数据集（这次我们使用 **ggplot2** 软件包），该应用程序打印出数据的摘要和绘图。它还遵循良好做法，并使用反应式表达式来避免代码的冗余。然而，下面提供的代码中有三个错误。你能找到并修复它们吗？

```

library(shiny)
library(ggplot2)

datasets <- c("economics", "faithfuld", "seals")
ui <- fluidPage(
  selectInput("dataset", "Dataset", choices = datasets),
  verbatimTextOutput("summary"),
  tableOutput("plot")
)

server <- function(input, output, session) {
  dataset <- reactive({
    get(input$dataset, "package:ggplot2")
  })
  output$summmry <- renderPrint({
    summary(dataset())
  })
  output$plot <- renderPlot({
    plot(dataset)
  }, res = 96)
}

shinyApp(ui, server)

```

Chapter 2. Basic UI

2.1 Introduction

现在您已经拥有了一个基本的应用程序，我们可以开始探索使Shiny运行的细节。正如您在上一章中看到的，Shiny鼓励将生成用户界面（前端）的代码与驱动应用程序行为的代码（后端）分开。

在本章中，我们将重点关注前端，并为您介绍Shiny提供的HTML样式的输入和输出风格。这使您能够捕获多种类型的数据并显示多种类型的R输出。在本章节中，您还不能将输入和输出拼接在一起，本章节仅仅是为了熟悉UI的控件，但我们将在第6章中讲解如何拼接。

在这里，我将主要坚持Shiny本身内置的输入和输出。然而，有一个丰富而充满活力的扩展包社区，如 [shinyWidgets](#)、[colorpicker](#) 和 [sortable](#)。您可以在 <https://github.com/nanxstats/awesome-shiny-extensions> 上找到由 [Nan Xiao](#) 维护的其他软件包的全面、积极维护的列表。

像往常一样，我们将从加载shiny的包装开始：

```
library(shiny)
```

2.2 Inputs

正如我们在上一章中看到的，您使用 `sliderInput()`，`selectInput()`，`textInput()` 和 `numericInput()` 等函数将输入控件插入UI中。现在，我们将讨论所有输入函数所支持的通用结构，并快速概述Shiny内置的输入。

2.2.1 Common structure

所有输入函数都有相同的第一个参数：`inputId`。这是用于将前端与后端连接的标识符：如果您的UI有一个带有ID为 "name" 的输入，服务函数（server function）将使用 `input$name` 访问它。

`inputId` 有两个约束：

- 它必须是一个简单的字符串，只包含字母、数字和下划线（不允许空格、破折号、句号或其他特殊字符！）。命名它，就像你在R中命名一个变量一样。
- 它必须是独一无二的。如果它不是唯一的，您将无法在服务函数中引用此控件！

大多数输入函数都有第二个参数，称为 `label`。这用于为控件创建一个人类可读的标签。Shiny不会对这个字符串施加任何限制，但您需要仔细考虑它，以确保您的应用程序可被人类可读！第三个参数通常是 `value`，在可能的情况下，它允许您设置默认值。其余参数是控件独有的。

创建输入时，我建议按位置提供 `inputId` 和 `label` 参数，以及按名称提供所有其他参数：

```
sliderInput("min", "Limit (minimum)", value = 50, min = 0, max = 100)
```

以下各节描述了内置在Shiny中的输入，根据它们创建的控件类型分组。目标是让您快速了解您的选择，而不是详尽地描述所有关键点。我将在下面展示每个控件的最重要参数，但您需要阅读文档以获取完整详细信息。

2.2.2 Free text

使用 `textInput()` 获取少量文本，使用 `passwordInput()` 获取密码，使用 `textAreaInput()` 获取文本段落。

```
ui <- fluidPage(
  textInput("name", "What's your name?"),
  passwordInput("password", "What's your password?"),
  textAreaInput("story", "Tell me about yourself", rows = 3)
)
```

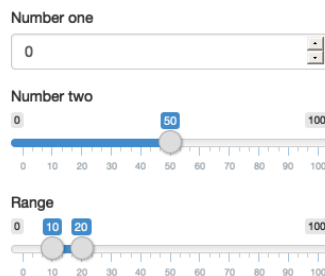


如果您想确保文本具有某些属性，您可以使用 `validate()`。我们将在第8章中讲解该属性。

2.2.3 Numeric inputs

要获取数值，请创建一个带有 `numericInput()` 的受限文本框或带有 `sliderInput()` 的滑块。如果您为 `sliderInput()` 的默认值提供长度为2的数字向量，您将获得一个“范围”滑块。

```
ui <- fluidPage(
  numericInput("num", "Number one", value = 0, min = 0, max = 100),
  sliderInput("num2", "Number two", value = 50, min = 0, max = 100),
  sliderInput("rng", "Range", value = c(10, 20), min = 0, max = 100)
)
```



一般来说，我建议只对小范围或精确值不那么重要的情况使用滑块。试图在小滑块上精确选择一个数字是一种沮丧的练习！

滑块是高度可定制的，有很多方法可以调整其外观。请参阅 `?sliderInput` 和 <https://shiny.rstudio.com/articles/sliders.html> 了解更多详情。

2.2.4 Dates

使用 `dateInput()` 获取一天，或使用 `dateRangeInput()` 获取日期范围。这些提供了一个方便的日历选择器，并且其他参数，如 `datesdisabled` 和 `daysofweekdisabled` 允许您限制有效输入的集合。

```
ui <- fluidPage(  
  dateInput("dob", "When were you born?"),  
  dateRangeInput("holiday", "When do you want to go on vacation next?")  
)
```

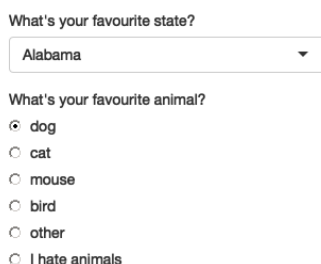


日期格式、语言和一周开始的日期默认为美国标准。如果您正在创建具有国际受众的应用程序，请设置 `format`、`language` 和 `weekstart` 以便日期对您的用户来说是自然的。

2.2.5 Limited choices

有两种不同的方法允许用户从一组预先指定的选项中进行选择：`selectInput()` 和 `radioButtons()`

```
animals <- c("dog", "cat", "mouse", "bird", "other", "I hate animals")  
  
ui <- fluidPage(  
  selectInput("state", "What's your favourite state?", state.name),  
  radioButtons("animal", "What's your favourite animal?", animals)  
)
```



单选按钮（radio buttons）有两个很好的功能：它们显示所有可能的选项，使它们适合短列表，通过 `choiceNames` / `choiceValues` 参数，它们可以显示纯文本以外的选项。`choiceNames` 决定向用户显示的内容；`choiceValues` 决定在服务器函数中返回的内容。

```
ui <- fluidPage(  
  radioButtons("rb", "Choose one:",  
    choiceNames = list(  
      icon("angry"),  
      icon("smile"),
```

```

      icon("sad-tear")
    ),
    choiceValues = list("angry", "happy", "sad")
  )
)
#> This Font Awesome icon ('angry') does not exist:
#> * if providing a custom `html_dependency` these `name` checks can
#>   be deactivated with `verify_fa = FALSE`
#> This Font Awesome icon ('smile') does not exist:
#> * if providing a custom `html_dependency` these `name` checks can
#>   be deactivated with `verify_fa = FALSE`
#> This Font Awesome icon ('sad-tear') does not exist:
#> * if providing a custom `html_dependency` these `name` checks can
#>   be deactivated with `verify_fa = FALSE`

```

Choose one:



使用 `selectInput()` 创建的下拉菜单占用相同的空间，无论选项数量多少，使它们更适合较长的选项。您还可以设置 `multiple = TRUE`，以允许用户选择多个元素。

```

ui <- fluidPage(
  selectInput(
    "state", "What's your favourite state?", state.name,
    multiple = TRUE
  )
)

```

What's your favourite state?

Texas Cal

California

如果您有大量的可能的选项，您可能希望使用“server-side” `selectInput()`，这样一整套可能的选项就不会嵌入到UI中（这可能会使其加载缓慢），而是根据server的需要发送。您可以在<https://shiny.rstudio.com/articles/selectize.html#server-side-selectize>上了解有关此高级主题的更多信息。

没有办法用单选按钮（radio buttons）选择多个值，但有一个概念上相似的替代方案：`checkboxGroupInput()`

```

ui <- fluidPage(
  checkboxGroupInput("animal", "What animals do you like?", animals)
)

```

What animals do you like?

- ☐ dog
- ☐ cat
- ☐ mouse
- ☐ bird
- ☐ other
- ☐ I hate animals

如果您想要单个是/否问题的单个复选框，请使用 `checkboxInput()`

```
ui <- fluidPage(  
  checkboxInput("cleanup", "Clean up?", value = TRUE),  
  checkboxInput("shutdown", "Shutdown?")  
)
```

☒ Clean up?
☐ Shutdown?

2.2.6 File uploads

允许用户使用 `fileInput()` 上传文件：

```
ui <- fluidPage(  
  fileInput("upload", NULL)  
)
```

No file selected

`fileInput()` 需要在服务器端进行特殊处理，并在第9章中详细讨论。

2.2.7 Action buttons

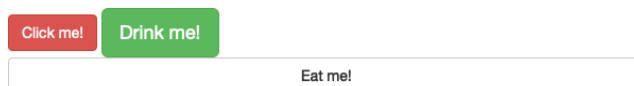
让用户使用 `actionButton()` 或 `actionLink()` 执行操作：

```
ui <- fluidPage(  
  actionButton("click", "Click me!"),  
  actionButton("drink", "Drink me!", icon = icon("cocktail"))  
)  
#> This Font Awesome icon ('cocktail') does not exist:  
#> * if providing a custom `html_dependency` these `name` checks can  
#> be deactivated with `verify_fa = FALSE`
```

操作链接 (actionLink) 和按钮 (actionButton) 时, 常常与服务端函数中的 `observeEvent()` 或 `eventReactive()` 配对。您还没有了解这些重要的功能, 但我们将在第3.5节中重新介绍它们。

您可以使用 "btn-primary", "btn-success", "btn-info", "btn-warning" 或 "btn-danger" 之一来自定义 `class` 参数的外观。您还可以使用 "btn-lg", "btn-sm", "btn-xs" 更改大小。最后, 您可以使用 "btn-block", 使按钮跨越它们嵌入的元素的整个宽度。

```
ui <- fluidPage(  
  fluidRow(  
    actionButton("click", "Click me!", class = "btn-danger"),  
    actionButton("drink", "Drink me!", class = "btn-lg btn-success")  
  ),  
  fluidRow(  
    actionButton("eat", "Eat me!", class = "btn-block")  
  )  
)
```



`class` 参数通过设置底层HTML的 `class` 属性来工作, 这会影响元素的样式。要查看其他选项, 您可以阅读 Bootstrap 的文档, Bootstrap 是 Shiny 使用的 CSS 设计系统:
<http://bootstrapdocs.com/v3.3.6/docs/css/#buttons>。

2.2.8 Exercises

1. 当空间有限时, 使用出现在文本输入区域内的占位符来标记文本框是有用的。您如何调用 `textInput()` 来生成下面的UI?

-
2. 仔细阅读 `sliderInput()` 的文档, 以了解如何创建日期滑块, 如下所示。



-
3. 创建一个滑块输入, 以选择0到100之间的值, 其中滑块上每个可选择值之间的间隔为5。然后, 向输入小部件添加动画, 以便当用户按下播放输入小部件时, 会自动滚动范围。

4. 如果您在 `selectInput()` 中有一个中等长度的列表，那么创建将列表分解成碎片的子标题是有用的。阅读文档以找出方法。（提示：底层HTML称为 `<optgroup>`。）

2.3 Outputs

UI中的输出创建占位符，稍后由server函数填充。与输入一样，输出将唯一的ID作为第一个参数：如果您的UI创建了一个ID为 `"plot"` 的输出，那么您将在server函数中使用 `output$plot` 访问它。

前端的每个 `output` 函数都与后端的 `render` 函数相结合。输出有三种主要类型，对应于您通常在报告中包含的三件事：文本（text）、表格（tables）和绘图（plots）。以下部分向您展示了前端输出函数的基础知识，以及后端的相应 `render` 函数。

2.3.1 Text

使用 `textOutput()` 输出常规文本，使用 `verbatimTextOutput()` 固定代码和控制台输出。

```
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("code")
)
server <- function(input, output, session) {
  output$text <- renderText({
    "Hello friend!"
  })
  output$code <- renderPrint({
    summary(1:10)
  })
}
```

Hello friend!

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.00	3.25	5.50	5.50	7.75	10.00

请注意，只有在需要运行多行代码时，渲染函数中才需要 `{}`。正如您很快就会学到的，您应该在渲染函数中尽可能少地进行计算，这意味着您可以经常省略它们。如果写得更紧凑，以下是上面的另一种写法：

```
server <- function(input, output, session) {
  output$text <- renderText("Hello friend!")
  output$code <- renderPrint(summary(1:10))
}
```

请注意，有两个渲染函数的行为略有不同：

- `renderText()` 将结果组合成一个字符串，通常与 `textOutput()` 配对
- `renderPrint()` 打印结果，就像您在R控制台中一样，通常与 `verbatimTextOutput()` 配对。

我们可以看到两者的区别：

```
ui <- fluidPage(
  textOutput("text"),
  verbatimTextOutput("print")
)
server <- function(input, output, session) {
  output$text <- renderText("hello!")
  output$print <- renderPrint("hello!")
}
```

hello!

```
[1] "hello!"
```

这相当于base-R中 `cat()` 和 `print()` 之间的差异。

2.3.2 Tables

在表格中显示数据框有两个选项：

- `tableOutput()` 和 `renderTable()` 渲染一个静态数据表，同时显示所有数据。
- `dataTableOutput()` 和 `renderDataTable()` 渲染一个动态表，显示固定数量的行以及更改可见行的控件。

`tableOutput()` 对小型固定汇总表（例如模型系数）最有用；如果您想向用户公开一个完整的数据框，`dataTableOutput()` 最合适。如果您想更好地控制 `dataTableOutput()` 的输出，我强烈推荐Greg Lin的 [reactable](#) 包。

```
ui <- fluidPage(
  tableOutput("static"),
  dataTableOutput("dynamic")
)
server <- function(input, output, session) {
  output$static <- renderTable(head(mtcars))
  output$dynamic <- renderDataTable(mtcars, options = list(pageLength = 5))
}
```

mpg	cyl	displacement	horsepower	drat	weight	qsec	vs	am	gear	carb
21.00	6.00	160.00	110.00	3.90	2.62	16.46	0.00	1.00	4.00	4.00
21.00	6.00	160.00	110.00	3.90	2.88	17.02	0.00	1.00	4.00	4.00
22.80	4.00	108.00	93.00	3.85	2.32	18.61	1.00	1.00	4.00	1.00
21.40	6.00	258.00	110.00	3.08	3.21	19.44	1.00	0.00	3.00	1.00
18.70	8.00	360.00	175.00	3.15	3.44	17.02	0.00	0.00	3.00	2.00
18.10	6.00	225.00	105.00	2.76	3.46	20.22	1.00	0.00	3.00	1.00

Show entries

Search:

mpg	cyl	displacement	horsepower	drat	weight	qsec	vs	am	gear	carb
21	6	160	110	3.9	2.62	16.46	0	1	4	4
21	6	160	110	3.9	2.875	17.02	0	1	4	4
22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
18.7	8	360	175	3.15	3.44	17.02	0	0	3	2

Showing 1 to 5 of 32 entries

Previous

1

2

3

4

5

6

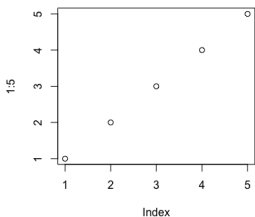
7

Next

2.3.3 Plots

您可以使用 `plotOutput()` 和 `renderPlot()` 显示任何类型的R图形（base、ggplot2或其他）：

```
ui <- fluidPage(
  plotOutput("plot", width = "400px")
)
server <- function(input, output, session) {
  output$plot <- renderPlot(plot(1:5), res = 96)
}
```



默认情况下，`plotOutput()` 将占用其容器的全部宽度（稍后将详细介绍），并且将达到400像素。您可以使用 `height` 和 `width` 参数覆盖这些默认值。我们建议始终设置 `res = 96`，因为这将使您的Shiny绘图尽可能与您在RStudio中看到的内容相匹配。

绘图是特殊的，因为它们也可以作为输入的输出。`plotOutput()` 有许多参数，如 `click`、`dblclick` 和 `hover`。如果您传递这些字符串，如 `click = "plot_click"`，它们将创建一个反应式输入（`input$plot_click`），您可以使用该输入来处理绘图上的用户交互，例如单击绘图。我们将在第7章中的深入讲解。

2.3.4 Downloads

您可以让用户使用 `downloadButton()` 或 `downloadLink()` 下载文件。这些需要server函数中的新技术，因此我们将在第9章中详细讲解。

2.3.5 Exercises

1. 以下哪个渲染函数应该与哪个 `textOutput()` 和 `verbatimTextOutput()` 配对?
 1. `renderPrint(summary(mtcars))`
 2. `renderText("Good morning!")`
 3. `renderPrint(t.test(1:5, 2:6))`
 4. `renderText(str(lm(mpg ~ wt, data = mtcars)))`
2. 从第2.3.3节重新创建Shiny应用程序，这次将高度设置为300px，宽度设置为700px。设置plot为“alt”文本，以便视障用户可以分辨出它是五个随机数的散点图。
3. 更新下面调用 `renderDataTable()` 中的选项，以便显示数据，但所有其他控件都会被抑制（即删除搜索、排序和过滤命令）。你需要阅读 `?renderDataTable` 并在<https://datatables.net/reference/option/>上查看选项。

```
ui <- fluidPage(
  dataTableOutput("table")
)
server <- function(input, output, session) {
  output$table <- renderDataTable(mtcars, options = list(pageLength = 5))
}
```

4. 或者，阅读reactable，并转换上述应用程序以改用它。

2.4 Summary

本章向您介绍了构成Shiny应用程序前端的主要输入和输出功能。这是一个很大的信息转储，所以不要指望在一次阅读后记住所有内容。相反，当您正在寻找特定组件时，请回到本章：您可以快速扫描数字，然后找到您需要的代码。

在下一章中，我们将进入Shiny应用程序的后端：使您的用户界面栩栩如生的R代码。

Chapter 3. Basic reactivity

3.1 Introduction

在Shiny中，您使用反应式编程来表达您的server逻辑。反应式编程是一种优雅而强大的编程范式，但起初可能会让人迷失方向，因为它与编写脚本非常不同。反应式编程的关键思想是指定一个依赖关系图，以便当输入发生变化时，所有相关输出都会自动更新。这使得应用程序的流程变得相当简单，但需要一段时间才能了解它是如何组合在一起的。

本章将简单地介绍反应式编程，教您在Shiny应用程序中使用最常见的反应式编程结构的基础知识。我们将从探究server函数开始，更详细地讨论 `input` 和 `output` 参数的工作原理。接下来，我们将回顾最简单的反应式编程（输入直接连接到输出），然后讨论反应式表达式如何允许您消除重复工作。最后，我们将回顾较新的Shiny用户遇到的一些常见问题。

3.2 The server function

正如你所看到的，每个Shiny应用程序的内部结构都是这样的：

```
library(shiny)

ui <- fluidPage(
  # front end interface
)

server <- function(input, output, session) {
  # back end logic
}

shinyApp(ui, server)
```

上一章涵盖了前端的基础知识，即包含HTML的 `ui` 对象，该对象呈现给应用程序的每个用户。`ui` 很简单，因为每个用户都获得相同的HTML。`server` 更加复杂，因为每个用户都需要获得应用程序的独立版本；当用户A移动滑块时，用户B不应该看到他们的输出变化。

为了实现这种独立性，每次启动新会话时，Shiny都会调用您的 `server()` 函数。就像任何其他R函数一样，当调用 `server` 函数时，它会创建一个独立于该函数的所有其他调用的新本地环境。这允许每个会话具有独特的状态，以及隔离在函数中创建的变量。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 `server` 函数中。

`server` 函数需要三个参数：`input`、`output` 和 `session`。因为您永远不会自己调用 `server` 函数，所以您永远不会自己创建这些对象。相反，它们由Shiny在会话开始时创建，连接到特定的会话。目前，我们将专注于 `input` 和 `output` 参数，并将 `session` 留到以后的章节。

3.2.1 Input

`input` 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包含一个带有 `count` 输入ID的数字输入控件，例如：

```
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。

与典型的列表不同，`input` 对象是只读的。如果您尝试在 `server` 函数中修改，您将收到一个错误：

```
server <- function(input, output, session) {
  input$count <- 10
}

shinyApp(ui, server)
#> Error: Can't modify read-only reactive value 'count'
```

出现此错误是因为 `input` 反映了浏览器中发生的事情，浏览器是Shiny的“单一真相来源”。如果你能修改R中的值，你可能会引入不一致，输入滑块在浏览器中说了一件事，而 `input$count` 在R中说了不同的东西。这将使编程具有挑战性！稍后，在第8章中，您将学习如何使用 `updateNumericInput()` 等函数在浏览器中修改值，然后 `input$count` 将相应更新。

关于 `input` 的更重要的事情是：它选择允许谁来阅读它。要从 `input` 中读取，您必须处于由函数，如 `renderText()` 或 `reactive()`，创建的响应上下文中。我们很快就会看到这个想法，但这是一个重要的约束，允许输出在输入更改时自动更新。此代码说明了如果您犯了这个错误，您将看到错误：

```
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 Output

`output` 与 `input` 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，您使用它来发送输出，而不是接收输入。您总是将 `output` 对象与 `render` 函数一起使用，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中引用，但在server中没有引用。)

渲染功能做两件事：

- 它设置了一个特殊的响应式上下文，自动跟踪输出使用的输入。
- 它将R代码的输出转换为适合在网页上显示的HTML。

像 `input`，`output` 对你如何使用它很挑剔。如果出现以下错误，您将收到一个错误：

- 你忘记了 `render` 功能。

```
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?
```

- 您尝试从输出中读取。

```
server <- function(input, output, session) {  
  message("The greeting is ", output$greeting)  
}  
shinyApp(ui, server)  
#> Error: Reading from shinyoutput object is not allowed.
```

3.3 Reactive programming

如果一个应用程序只有输入或只有输出，那么它会很无聊。当你有一个同时拥有两者的应用程序时，Shiny的真正魔力就会发生。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, "!")  
  })  
}
```

很难在一本书中说明这是如何运作的，但我在图3.1中尽了最大努力。如果您运行该应用程序并在名称框中键入，您将看到问候语在您键入时自动更新。



图3.1：反应性意味着输出会随着输入的变化自动更新，就像在这个应用程序中，我键入“J”、“o”、“e”一样。在<https://hadley.shinyapps.io/ms-connection>上观看动态。

这是Shiny的主体逻辑：您不需要告诉输出何时更新，因为Shiny会自动为您计算出来。它是如何工作的？函数主体到底发生了什么？让我们更准确地探究server函数中的代码：

```
output$greeting <- renderText({  
  paste0("Hello ", input$name, "!")  
})
```

很容易将其解读为：将'hello'和用户名粘贴在一起，然后将其发送到 `output$greeting`。但这个心理模型以一种微妙但重要的方式是错误的。想一想：使用这个模型，你只能发出一次指令。但每次我们更新 `input$name` 时，Shiny都会执行该操作，所以一定还有更多事情发生。

该应用程序之所以有效，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如何在需要时创建字符串。这取决于Shiny何时（即使如此！）应该运行代码。它可能会在应用程序启动后立即运行，可能会晚一点；它可能会运行多次，或者可能永远不会运行！这并不意味着Shiny是反复无常的，只是Shiny有责任决定何时执行代码，而不是你的责任。将您的应用程序视为为Shiny提供方法，而不是给它命令。

3.3.1 Imperative vs declarative programming

命令和声明之间的这种差异是两种重要编程风格之间的关键区别之一：

- 在**命令式 (imperative)** 编程中，您发出特定命令，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。
- 在**声明性 (declarative)** 编程中，您表达更高层次的目标或描述重要的制约因素，并依靠他人来决定如何和/或何时将其转化为行动。这是你在Shiny中使用的编程风格。

用命令式代码，你说“给我做个三明治”。使用声明性代码，你说“每当我查看冰箱里面时，确保冰箱里有一个三明治”。命令式代码是自信的；声明性代码是被动攻击性的。

大多数时候，声明性编程是巨大的自由：你描述你的总体目标，软件会弄清楚如何在没有进一步干预的情况下实现这些目标。缺点是偶尔你确切地知道自己想要什么，但你无法弄清楚如何以声明系统理解的方式构建它。这本书的目标是帮助您发展对基本理论的理解，以便尽可能少发生。

3.3.2 Laziness

Shiny中声明式编程的优势之一是它允许应用程序非常懒惰。Shiny应用程序只会完成更新您目前可以看到的输出控件所需的最低限度的工作。然而，这种懒惰伴随着一个重要的缺点，你应该注意。你能发现下面的server函数出了什么问题吗？

```
server <- function(input, output, session) {  
  output$greteng <- renderText({  
    paste0("Hello ", input$name, "!")  
  })  
}
```

如果你仔细观察，你可能会注意到我写了 `greteng` 而不是 `greeting`。这不会在Shiny中产生错误，但它不会做你想做的事。`greteng` 输出不存在，因此 `renderText()` 中的代码将永远不会运行。

如果您正在处理一个Shiny应用程序，而您无法弄清楚为什么您的代码永远不会运行，请仔细检查您的UI和server函数是否使用相同的标识符。

3.3.3 The reactive graph

Shiny的懒惰还有另一个重要属性。在大多数R代码中，您可以通过从上到下读取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。要了解执行顺序，您需要查看**反应图**，该图描述了输入和输出的连接方式。上面应用程序的反应图非常简单，如图3.2所示。

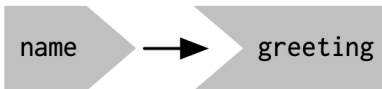


图3.2：反应图显示了输入和输出的连接方式

反应图包含每个输入和输出的一个符号，每当输出访问输入时，我们都会将输入连接到输出。这张图告诉您，每当更改 `name` 时，都需要重新计算 `greeting`。我们经常会将这种关系描述为 `greeting` 对 `name` 有**反应依赖性**。

注意，我们用于输入和输出的图形约定：`name` 输入链接 `greeting` 输出。我们可以将它们紧密地组合在一起，如图3.3所示，以强调它们结合在一起的方式；我们通常不会这样做，因为它只适用于最简单的应用程序。



图3.3：反应图的成分使用的形状唤起了它们的连接方式。

反应图是了解应用程序如何工作的强大工具。随着您的应用程序变得越来越复杂，制作反应图的快速高级草图通常很有用，以提醒您所有部分如何组合在一起。在整本书中，我们将向您展示反应图，以帮助了解示例的工作原理。稍后，在第14章中，您将学习如何使用reactlog为您绘制图表。

3.3.4 Reactive expressions

您将在反应图中看到一个更重要的组成部分：反应式表达式。我们很快就会详细介绍反应式表达式。现在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的重复。

我们非常简单的应用程序中不需要反应表达式，但我无论如何都会添加一个，这样你就可以看到它如何影响反应图，图3.4。

```
server <- function(input, output, session) {  
  string <- reactive(paste0("Hello ", input$name, "!"))  
  output$greeting <- renderText(string())  
}
```

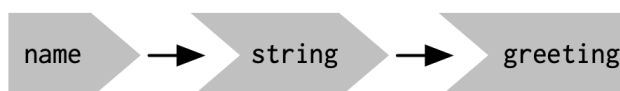


图3.4：反应式表达式在两侧都有角度，因为它将输入连接到输出。

反应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你记住组件是如何结合在一起的。

3.3.5 Execution order

重要的是要了解，您的代码运行顺序仅由反应图决定。这与大多数R代码不同，在大多数R代码中，执行顺序由行顺序决定。例如，我们可以在简单的server函数中翻转两行的顺序：

```
server <- function(input, output, session) {  
  output$greeting <- renderText(string())  
  string <- reactive(paste0("Hello ", input$name, "!"))  
}
```

您可能认为这会产生一个错误，因为 `output$greeting` 指的是尚未创建的反应式表达式 `string`。但请记住，Shiny是懒惰的，因此代码仅在会话开始时，在创建 `string` 后运行。

相反，该代码产生与上述相同的反应图，因此代码的运行顺序完全相同。像这样组织你的代码对人类来说令人困惑，最好避免。相反，请确保反应式表达式和输出仅引用上面定义的东西，而不是下面。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，所以我再说一遍：反应代码的运行顺序仅由反应图决定，而不是由其在server函数中的布局决定。

3.3.6 Exercises

1. 鉴于此用户界面：

```
ui <- fluidPage(  
  textInput("name", "What's your name?"),  
  textOutput("greeting")  
)
```

修复以下三个server函数中每个功能中发现的简单错误。首先尝试仅通过阅读代码来发现问题；然后运行代码以确保您已修复它。

```
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server函数的响应图：

```
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))
```

为什么 `range()` 和 `var()` 是反应性的坏名称？

3.4 Reactive expressions

我们快速浏览了几次反应式表达，以期望了解它们可能会做什么。现在，我们将深入研究更多细节，并展示为什么它们在构建真实应用程序时如此重要。

反应式表达式很重要，因为它们为 *Shiny* 提供了更多的信息，以便在输入更改时可以减少重复计算，使应用程序更高效，并且通过简化反应图，使人类更容易理解应用程序。反应式表达式具有输入和输出的双重属性：

- 与输入一样，您可以在输出中使用反应式表达式的结果。
- 与输出一样，反应式依赖于输入，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**来指代反应式的输入和表达式，使用**消费者**来指代反应式的表达式和输出。图3.5显示了与维恩图的这种关系。

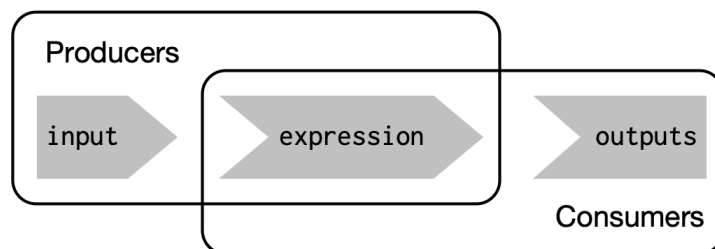


图3.5：输入和表达式是反应式的生产者；表达式和输出是反应式的消费者

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

3.4.1 The motivation

想象一下，我想用图和假设检验来比较两个模拟数据集。我做了一些实验，并提出了以下函数：`freqpoly()` 用频率多边形可视化两个分布，`t_test()` 使用t测试来比较均值，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)
}

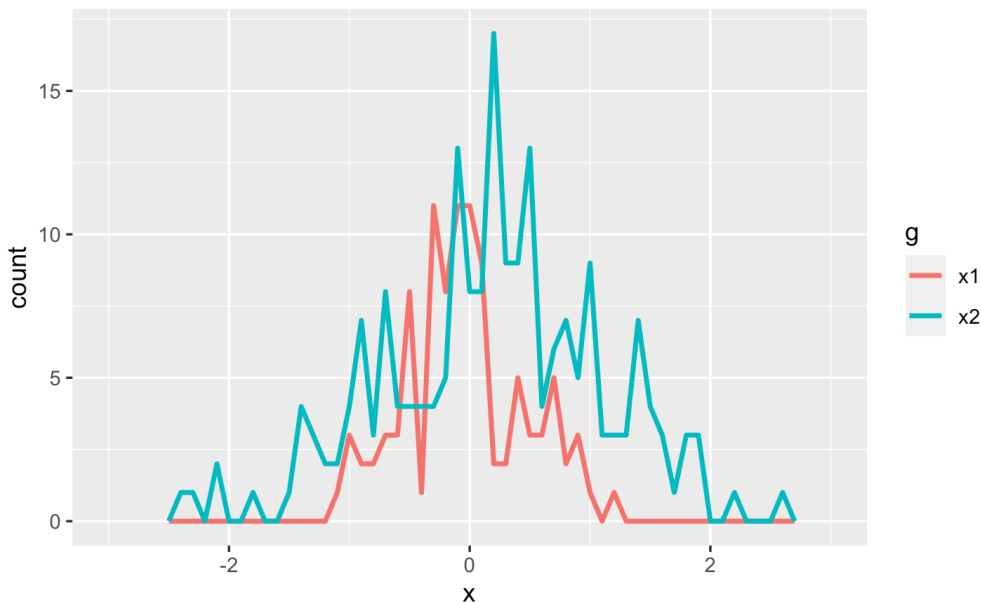
t_test <- function(x1, x2) {
  test <- t.test(x1, x2)
```

```
# 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]
)
}
```

如果我有一些模拟数据，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: 0.005
#> [-0.39, -0.07]
```



在真正的分析中，在获得这些功能之前，你可能会做很多探索。我跳过了这里的探索，这样我们就可以尽快进入应用程序。但是，将命令式代码提取到常规函数中是所有Shiny应用程序的重要技术：您可以从应用程序中提取的代码越多，就越容易理解。这是很好的软件工程，因为它有助于隔离问题：应用程序外部的函数专注于计算，因此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章中再次回到这个想法。

3.4.2 The app

我想用这两个工具来快速探索一系列模拟数据。Shiny应用程序是做到这一点的好方法，因为它可以让您避免繁琐地修改和重新运行R代码。在下面，我把这些碎片包装成一个shiny应用程序，在那里我可以交互式调整输入。

让我们从用户界面开始。我们将在6.2.3节中找到回到 `fluidRow()` 和 `column()` 的确切作用；但你可以从他们的名字中猜出他们的目的 😊。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 = " $\mu$ ", value = 0, step = 0.1),
  numericInput("sd1", label = " $\sigma$ ", value = 0.5, min = 0.1, step = 0.1)
),
column(4,
  "Distribution 2",
  numericInput("n2", label = "n", value = 1000, min = 1),
  numericInput("mean2", label = " $\mu$ ", value = 0, step = 0.1),
  numericInput("sd2", label = " $\sigma$ ", 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"))
)
)

```

Server函数从指定的控件中提取值后，结合了对 `freqpoly()` 和 `t_test()` 函数的调用，完成了上述功能：

```

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)
  })
}

```

 style="zoom:33%;">。" />

图3.6：一个Shiny应用程序，允许您将两个模拟分布与t-test和频率分布进行比较，见<https://hadley.shinyapps.io/ms-case-study-1>。

`server` 和 `ui` 的这个定义产生了图3.6。您可以在<https://hadley.shinyapps.io/ms-case-study-1>上找到实时版本；我建议在阅读之前打开该应用程序并快速播放，以确保您了解其基本操作。

3.4.3 The reactive graph

让我们从绘制这个应用程序的反应图开始。Shiny足够聪明，只有当输入发生变化时，才能更新输出；它不够聪明，只能有选择地在输出中运行代码片段。换句话说，输出是原子类型：它们要么被执行，要么不是作为一个整体。

例如，从server函数上获取此片段：

```
x1 <- rnorm(input$n1, input$mean1, input$sd1)
x2 <- rnorm(input$n2, input$mean2, input$sd2)
t_test(x1, x2)
```

作为一个阅读此代码的人，您可以知道，我们只需要在 `n1`、`mean1` 或 `sd1` 更改时更新 `x1`，我们只需要在 `n2`、`mean2` 或 `sd2` 更改时更新 `x2`。然而，Shiny只看整个输出，所以每次 `n1`、`mean1`、`sd1`、`n2`、`mean2` 或 `sd2` 发生变化时，它都会更新 `x1` 和 `x2`。这导致了图3.7所示的反应图：

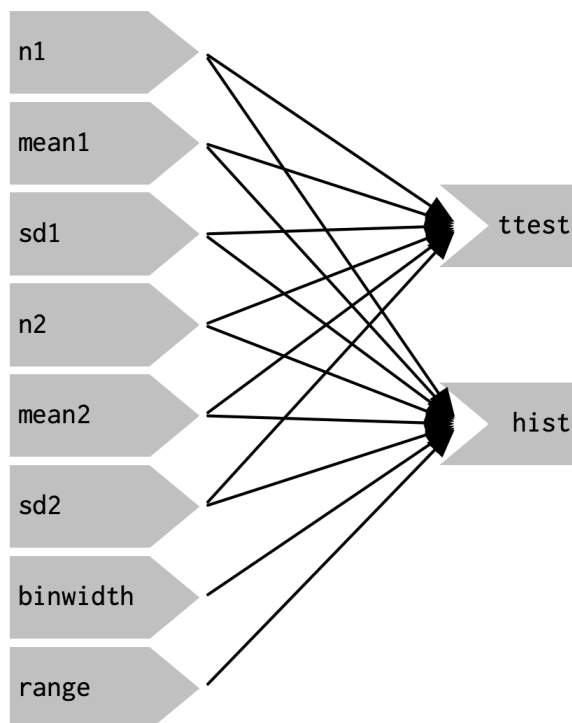


图3.7：反应图显示每个输出都取决于每个输入

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：

- 该应用程序很难理解，因为有这么多个连接。应用程序中没有可以单独拉出和分析的部分。
- 该应用程序效率低下，因为它的工作比必要的多。例如，如果您更改绘图的分隔符，则会重新计算数据；如果您更改 `n1` 的值，`x2` 将更新（在两个地方！）。

应用程序中还有一个主要缺陷：频率分布图和t检验使用单独的随机绘制。这相当具有误导性，因为您预计他们正在处理相同的基础数据。

幸运的是，我们可以通过使用反应式表达式提取重复计算来解决所有这些问题。

3.4.4 Simplifying the graph

在下面的server函数中，我们重构现有代码，将重复的代码拉出两个新的反应表达式，`x1` 和 `x2`，它们模拟来自两个分布的数据。要创建反应式表达式，我们调用 `reactive()` 并将结果分配给一个变量。为了稍后使用该表达式，我们像调用函数一样调用变量。

```
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())  
  })  
}
```

这种变换产生了图3.8所示的简单得多的图表。这个更简单的图表使您更容易理解该应用程序，因为您可以单独理解连接的组件；分布参数的值仅影响通过 `x1` 和 `x2` 的输出。这种重写也使应用程序更有效率，因为它的计算要少得多。现在，当您更改 `binwidth` 或 `range` 时，只有图会更改，而不是基础数据。

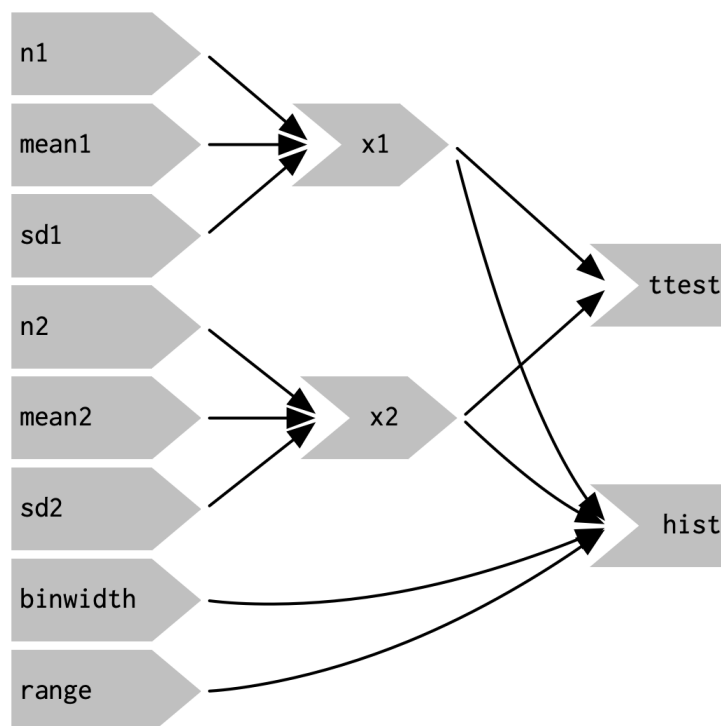


图3.8：使用反应式大大简化了图表，使其更容易理解

为了强调这种模块化，图3.9在独立组件周围绘制框。当我们讨论模块时，我们将在第19章中回到这个想法。模块允许您提取重复的代码以供重用，同时保证它与应用程序中的所有其他内容隔离。模块对于更复杂的应用程序来说是一种非常有用和强大的技术。

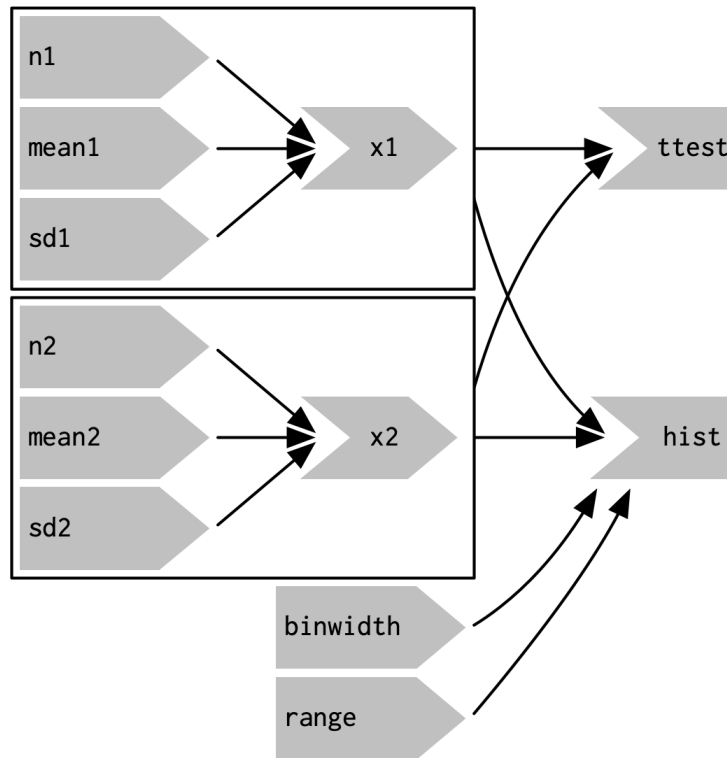


图3.9：模块在应用程序的各个部分之间强制隔离

您可能熟悉编程的“三规则”：每当您复制和粘贴三次内容时，您都应该弄清楚如何减少重复（通常通过编写函数）。这很重要，因为它减少了代码中的重复量，这使得它更容易理解，也更容易随着需求的变化而更新。

然而，在Shiny中，我认为你应该考虑一个规则：每当你复制和粘贴一次东西时，你应该考虑将重复的代码提取为反应式表达式。该规则对Shiny来说更严格，因为反应式表达式不仅使人类更容易理解代码，还提高了Shiny高效运行代码的能力。

3.4.5 Why do we need reactive expressions?

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式表达式。为什么您不能使用现有的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，这两种技术都不在反应式环境中工作。

如果您尝试使用变量来减少重复，您可能会写出这样的东西：

```

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)
  })
}

```

如果您运行此代码，您将收到一个错误，因为您正在尝试在反应式上下文之外访问输入值。即使您没有收到该错误，您仍然会遇到问题：x1 和 x2 只会在会话开始时计算一次，而不是每次更新其中一个输入时。

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())  
  })  
}
```

但它与原始代码存在相同的问题：任何输入都将导致所有输出被重新计算，t检验和频率分布将在单独的样本上运行。反应表达式会自动缓存其结果，并且仅在输入更改时更新。

虽然变量只计算一次值（粥太冷），而函数每次调用时都计算值（粥太热），但反应表达式仅在可能发生变化时计算值（粥恰到好处！）。

3.5 Controlling timing of evaluation

现在您已经熟悉了反应式表达式的基本概念，我们将讨论两种更高级的技术，这些技术允许您增加或减少执行反应式表达式的频率。在这里，我将展示如何使用基本技术；在第15章中，我们将回到它们的基本实现。

为了探索基本想法，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，并强制两个样本共享相同的 `n`。我还将删除绘图控件。这会产生一个较小的UI对象和server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)  
}
```

这会生成图[3.10](#)所示的应用程序和图[3.11](#)所示的反应图。

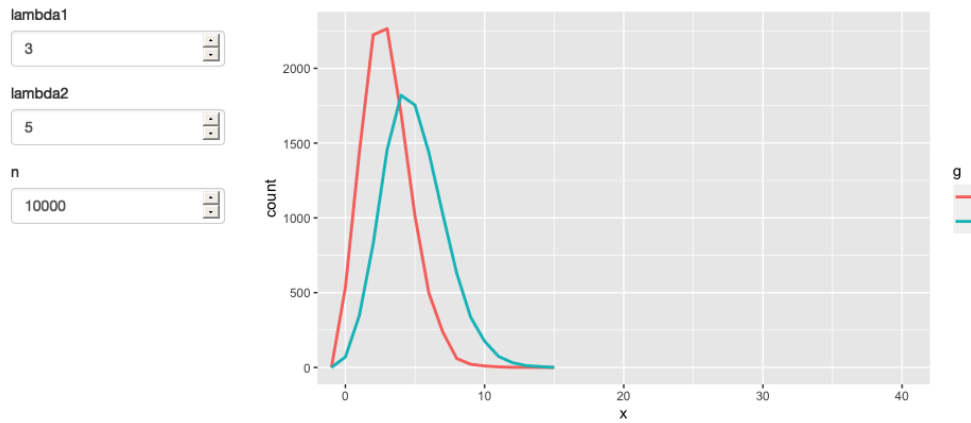


图3.10：一个更简单的应用程序，显示从两个泊松分布中提取的随机数的频率分布图。在<https://hadley.shinyapps.io/ms-simulation-2>上观看动态效果。

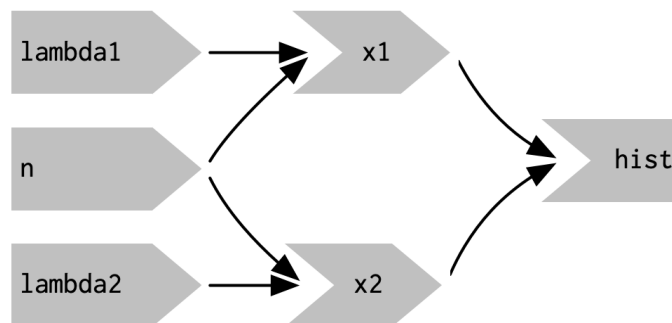


图3.11：反应图

3.5.1 Timed invalidation

想象一下，您想通过不断重新模拟数据来强调这是一个模拟数据的事实，这样您就可以看到动画而不是静态图。我们可以使用新功能：`reactiveTimer()` 增加更新频率。

`reactiveTimer()` 是一个依赖隐藏输入的反应式表达式：当前时间。当您希望反应表达式比其他方式更频繁地使自己无效时，您可以使用 `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。这足以提醒您正在查看模拟，而不会因快速变化而使您头晕目眩。此变化产生了图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)
}

```

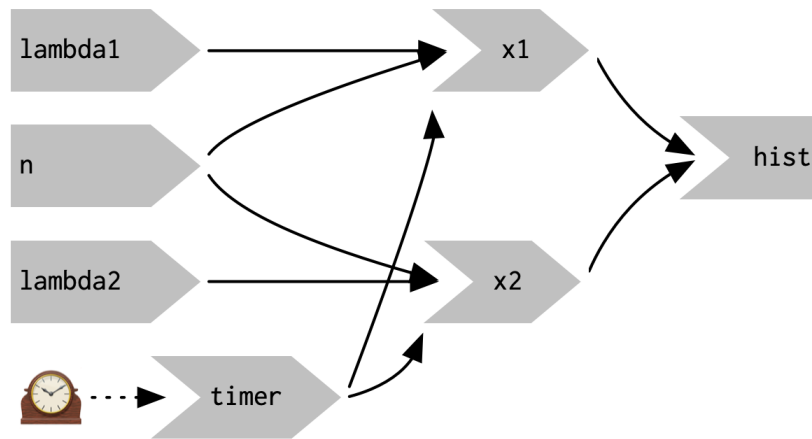


图3.12: `reactiveTimer(500)` 引入了一种新的反应式输入，每半秒自动失效一次

注意我们如何在计算 `x1()` 和 `x2()` 的反应式表达式中使用 `timer()`。我们调用它，但不使用该值。这让 `x1` 和 `x2` 在 `timer` 上接受反应式依赖，而不必担心它返回的确切值。

3.5.2 On click

在上述场景中，考虑一下如果模拟代码需要1秒才能运行，会发生什么。我们每0.5s执行一次模拟，所以Shiny会有越来越多的事情要做，而且永远无法赶上。如果有人快速单击您应用程序中的按钮，并且您正在执行的计算相对昂贵，也会发生同样的问题。有可能为Shiny创建大量积压工作，当它处理积压工作时，它无法响应任何新事件。这导致用户体验不佳。

如果您的应用程序中出现这种情况，您可能希望要求用户通过要求他们单击按钮来选择执行昂贵的计算。这是 `actionButton()` 的绝佳用例：

```

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action-button，我们需要学习一个新工具。为了了解原因，让我们首先使用与上述相同的方法解决这个问题。如上所述，我们指的是 `simulate`，而不使用其值来对它进行反应式依赖。

```
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)
}
```

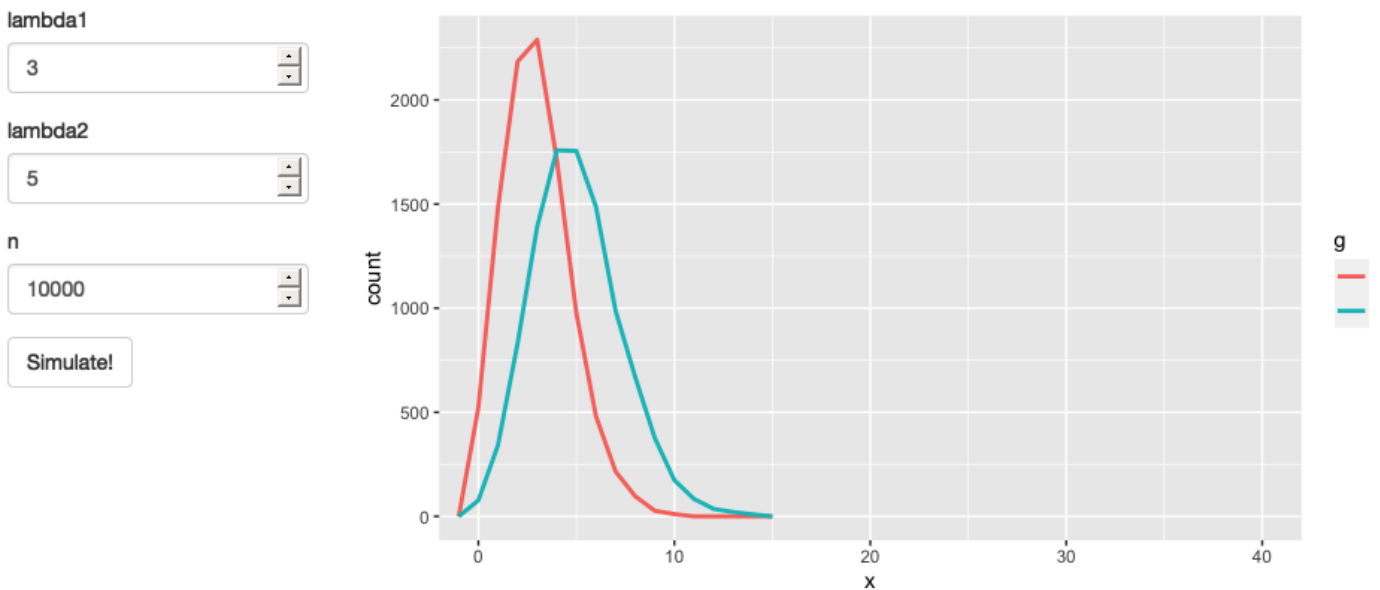


图3.13：带有action-button的应用程序。在<https://hadley.shinyapps.io/ms-action-button>观看动态。

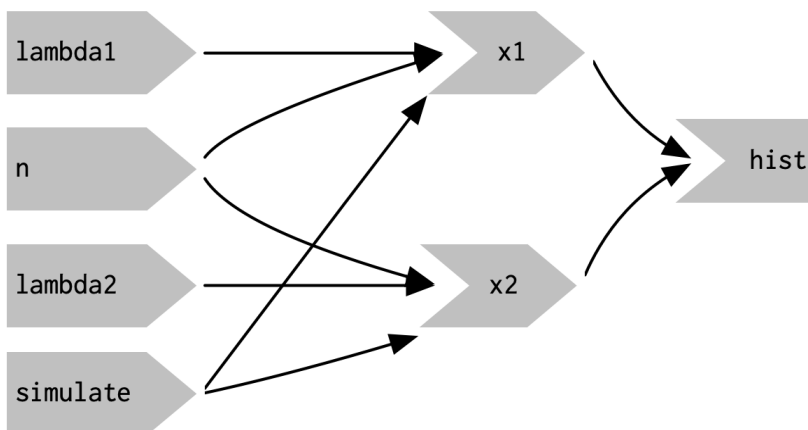


图3.14：这个反应图没有实现我们的目标；我们添加了一个依赖项，而不是替换现有的依赖项。

这产生了图3.13中的应用程序和图3.14中的反应图。这没有实现我们的目标，因为它只是引入了一个新的依赖项：当我们单击模拟按钮时，`x1()` 和 `x2()` 将更新，但当 `lambda1`、`lambda2` 或 `n` 更改时，它们也会继续更新。我们想替换现有的依赖项，而不是增加它们。

为了解决这个问题，我们需要一个新的工具：一种在不被动依赖输入值的情况下使用输入值的方法。我们需要 `eventReactive()` 它有两个参数：第一个参数指定了对什么进行依赖，第二个参数指定了要计算什么。这允许该应用程序在单击 `simulate` 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，根据需要，`x1` 和 `x2` 不再对 `lambda1`、`lambda2` 和 `n` 有反应式依赖：更改它们的值不会触发计算。我把箭头留在了非常浅的灰色中，只是为了提醒你 `x1` 和 `x2` 继续使用这些值，但不再对它们进行反应式依赖。

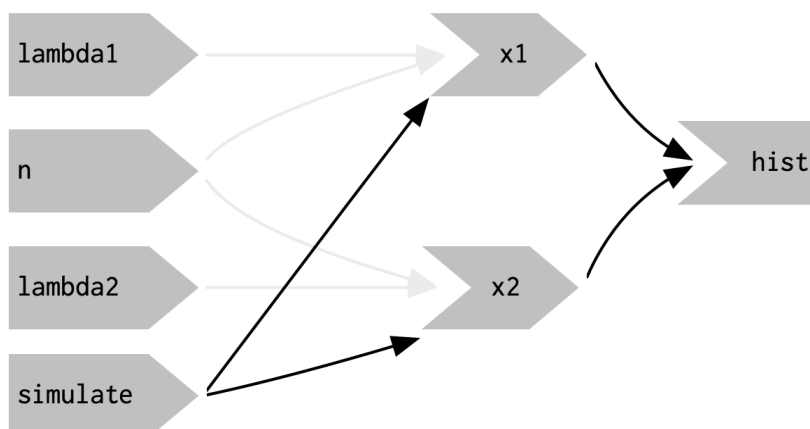


图3.15： `eventReactive()` 可以将依赖项（黑色箭头）与用于计算结果的值（浅灰色箭头）分开。

3.6 Observers

到目前为止，我们专注于应用程序内部发生的事情。但有时您需要接触到应用程序外部，并导致全局其他地方发生副作用。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（最常见的）将调试消息打印到控制台。这些操作不会影响您的应用程序的外观，因此您不应该使用输出和 `render` 函数。相反，您需要使用 **observer**。

创建 **observer** 有多种方法，我们将在稍后的第15.3节中讲解。现在，我想向您展示如何使用 `observeEvent()` 因为它在您第一次学习Shiny时为您提供了一个重要的调试工具。

`observeEvent()` 与 `eventReactive()` 非常相似。它有两个重要的参数： `eventExpr` 和 `handlerExpr`。第一个参数是依赖的输入或表达式；第二个参数是要运行的代码。例如，对 `server()` 的以下修改意味着每次更新该 `name` 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：

- 您不会将 `observeEvent()` 的结果分配给变量, so
- 你不能从其他反应式表达式那里引用它。

Observers和outputs密切相关。您可以认为输出具有特殊的副作用：在用户浏览器中更新HTML。为了强调这种接近性，我们将在反应图中以同样的方式绘制它们。这产生了图3.16所示的以下反应图。

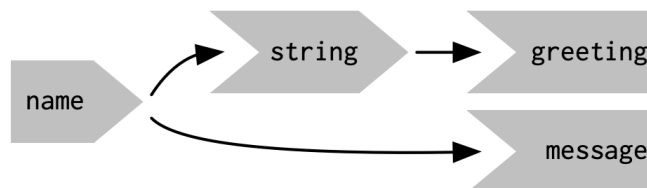


图3.16：在反应图中，观察者看起来与输出相同

3.7 Summary

本章应该提高您对Shiny应用程序后端的理解，即响应用户操作的 `server()` 代码。您还迈出了掌握支撑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；我们将回到第13章的基本理论。反应式编程非常强大，但它也与您最习惯的R编程的命令式风格非常不同。如果所有后果都需要一段时间才能沉淀下来，请不要感到惊讶。

本章总结了我们对Shiny基础的概述。下一章将帮助您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所看到的材料。

4 Case study: ER injuries

4.1 Introduction

在过去的三章中，我向你介绍了一堆新概念。因此，为了帮助他们融入其中，我们现在将浏览一个更丰富的Shiny应用程序，该应用程序探索一个有趣的数据集，并汇集您迄今为止看到的许多想法。我们将首先在Shiny之外进行一些数据分析，然后将其转换为应用程序，从简单开始，然后逐步分层更多细节。

在本章中，我们将用vroom（用于快速文件读取）和tidyverse（用于一般数据分析）来补充Shiny。

```
library(shiny)
library(vroom)
library(tidyverse)
```

4.2 The data

我们将探索消费品安全委员会收集的国家电子伤害监控系统（NEISS）的数据。这是一项长期研究，记录了美国医院代表性样本中的所有事故。这是一个值得探索的有趣数据集，因为每个人都已经熟悉了这个领域，每个观察数据都伴随着一个简短的叙述，解释了事故是如何发生的。您可以在<https://github.com/hadley/neiss>上了解有关此数据集的更多信息。

在本章中，我将只关注2017年的数据。这使数据足够小（~10 MB），易于存储在git中（与书的其余部分一起），这意味着我们不需要考虑快速导入数据的复杂策略（我们将在书的后面回到这些策略中）。您可以在<https://github.com/hadley/mastering-shiny/blob/master/neiss/data.R>上看到我用于创建本章摘录的代码。

如果您想将数据传输到自己的计算机上，请运行以下代码：

```
dir.create("neiss")
#> Warning in dir.create("neiss"): 'neiss' already exists
download <- function(name) {
  url <- "https://github.com/hadley/mastering-shiny/raw/master/neiss/"
  download.file(paste0(url, name), paste0("neiss/", name), quiet = TRUE)
}
download("injuries.tsv.gz")
download("population.tsv")
download("products.tsv")
```

我们将要使用的主要数据集是 `injuries`，其中包含约25万次观测：

```
injuries <- vroom::vroom("neiss/injuries.tsv.gz")
injuries
#> # A tibble: 255,064 × 10
#>   trmt_date    age sex  race body_part  diag  locat...1 prod_...2 weight narra...3
#>   <date>      <dbl> <chr> <chr> <chr>      <chr> <chr>      <dbl> <dbl> <chr>
#> 1 2017-01-01    71 male  white Upper Trunk Contus... Other ...    1807    77.7 71YOM ...
#> 2 2017-01-01    16 male  white Lower Arm Burns,... Home      676    77.7 16YOM ...
#> 3 2017-01-01    58 male  white Upper Trunk Contus... Home      649    77.7 58 YOM...
#> 4 2017-01-01    21 male  white Lower Trunk Strain... Home     4076    77.7 21 YOM...
#> 5 2017-01-01    54 male  white Head Inter ... Other ...    1807    77.7 54 YOM...
#> 6 2017-01-01    21 male  white Hand Fractu... Home     1884    77.7 21 YOM...
#> # ... with 255,058 more rows, and abbreviated variable names 1location,
#> # 2prod_code, 3narrative
```

每行代表一个具有10个变量的事故：

- `trmt_date` 是该人在医院被看到的日期（不是事故发生时间）。
- `age`、`sex` 和 `race` 提供了有关经历事故的人的人口统计信息。
- `body_part` 是身体受伤的位置（如脚踝或耳朵）；`location` 是事故发生的地方（如家庭或学校）。
- `diag` 给出损伤的基本诊断（如骨折或撕裂）。

- `prod_code` 是与伤害相关的主要产品。
- `weight` 是统计权重，如果将该数据集扩展到美国全体人口，将遭受这种伤害的估计人数。
- `narrative` 是一个关于事故如何发生的简短描述。

我们将其与其他两个数据框配对，以获取其他上下文：`products` 让我们从产品代码中查找产品名称，`population` 告诉我们2017年美国每个年龄和性别组合的总人口。

```
products <- vroom::vroom("neiss/products.tsv")
products
#> # A tibble: 38 × 2
#>   prod_code title
#>   <dbl> <chr>
#> 1      464 knives, not elsewhere classified
#> 2      474 tableware and accessories
#> 3      604 desks, chests, bureaus or buffets
#> 4      611 bathtubs or showers
#> 5      649 toilets
#> 6      676 rugs or carpets, not specified
#> # ... with 32 more rows

population <- vroom::vroom("neiss/population.tsv")
population
#> # A tibble: 170 × 3
#>   age sex      population
#>   <dbl> <chr>      <dbl>
#> 1     0 female    1924145
#> 2     0 male     2015150
#> 3     1 female    1943534
#> 4     1 male     2031718
#> 5     2 female    1965150
#> 6     2 male     2056625
#> # ... with 164 more rows
```

4.3 Exploration

在我们创建应用程序之前，让我们稍微探索一下数据。我们将首先看一个具有有趣故事的产品：649，“厕所”。首先，我们将取出与该产品相关的伤害：

```
selected <- injuries %>% filter(prod_code == 649)
nrow(selected)
#> [1] 2993
```

接下来，我们将进行一些基本总结，了解厕所相关伤害的位置、身体部位和诊断。请注意，我按 `weight` 变量加权，以便计数可以解释为整个美国的估计总伤害。

```
selected %>% count(location, wt = weight, sort = TRUE)
#> # A tibble: 6 × 2
#>   location      n
#>   <chr>      <dbl>
```

```

#> 1 Home 99603.
#> 2 Other Public Property 18663.
#> 3 Unknown 16267.
#> 4 School 659.
#> 5 Street Or Highway 16.2
#> 6 Sports Or Recreation Place 14.8

selected %>% count(body_part, wt = weight, sort = TRUE)
#> # A tibble: 24 × 2
#>   body_part      n
#>   <chr>      <dbl>
#> 1 Head 31370.
#> 2 Lower Trunk 26855.
#> 3 Face 13016.
#> 4 Upper Trunk 12508.
#> 5 Knee 6968.
#> 6 N.S./Unk 6741.
#> # ... with 18 more rows

selected %>% count(diag, wt = weight, sort = TRUE)
#> # A tibble: 20 × 2
#>   diag      n
#>   <chr>      <dbl>
#> 1 Other Or Not Stated 32897.
#> 2 Contusion Or Abrasion 22493.
#> 3 Inter Organ Injury 21525.
#> 4 Fracture 21497.
#> 5 Laceration 18734.
#> 6 Strain, Sprain 7609.
#> # ... with 14 more rows

```

正如你可能预料的那样，涉及厕所的伤害最常发生在家里。最常见的身体部位可能是跌倒，诊断似乎相当多样。我们还可以探索跨年龄和性别的模式。我们这里有足够的数据，表格没有那么有用，所以我做了一个图，图[4.1](#)，使模式更加明显。

```

summary <- selected %>%
  count(age, sex, wt = weight)
summary
#> # A tibble: 208 × 3
#>   age sex      n
#>   <dbl> <chr>  <dbl>
#> 1     0 female  4.76
#> 2     0 male   14.3
#> 3     1 female 253.
#> 4     1 male  231.
#> 5     2 female 438.
#> 6     2 male  632.
#> # ... with 202 more rows

summary %>%
  ggplot(aes(age, n, colour = sex)) +

```

```
geom_line() +
labs(y = "Estimated number of injuries")
```

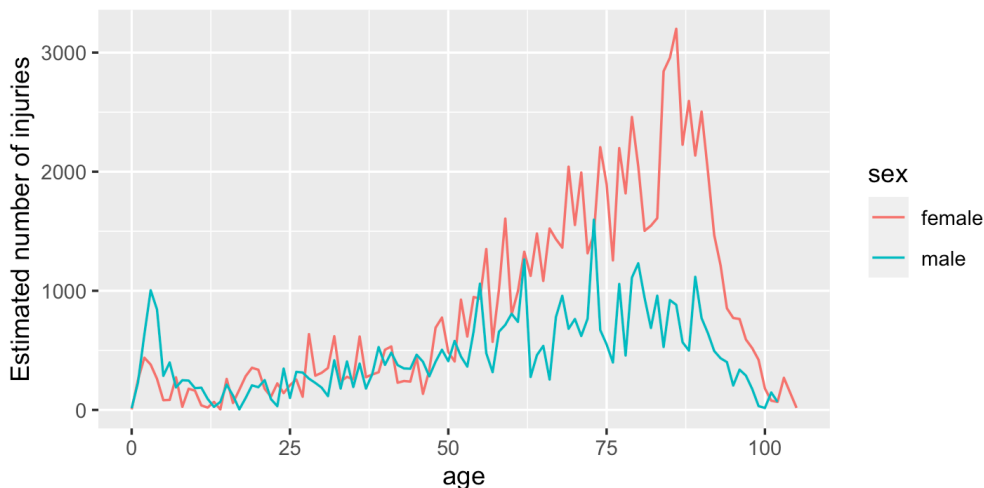


图4.1：按年龄和性别列的厕所造成的伤害估计人数

我们看到年轻男孩在3岁时达到峰值，然后从中年左右开始增加（特别是女性），80岁后逐渐下降。我怀疑高峰期是因为男孩通常站着上厕所，而女性的增加是由于骨质疏松症（即我怀疑女性和男性的受伤率相同，但更多的女性最终进入急诊室，因为她们骨折的风险更高）。

解释这种模式的一个问题是，我们知道老年人比年轻人少，所以受伤的人口较少。我们可以通过将受伤人数与总人口进行比较并计算受伤率来控制这一点。在这里，我使用的比率是每10,000个人中发生事故的比率。

```
summary <- selected %>%
  count(age, sex, wt = weight) %>%
  left_join(population, by = c("age", "sex")) %>%
  mutate(rate = n / population * 1e4)
```

```
summary
#> # A tibble: 208 × 5
#>   age sex      n population  rate
#>   <dbl> <chr>  <dbl>      <dbl> <dbl>
#> 1     0 female   4.76    1924145 0.0247
#> 2     0 male    14.3    2015150 0.0708
#> 3     1 female  253.    1943534 1.30
#> 4     1 male   231.    2031718 1.14
#> 5     2 female  438.    1965150 2.23
#> 6     2 male   632.    2056625 3.07
#> # ... with 202 more rows
```

绘制该比率，图4.2，在50岁后产生了截然不同的趋势：男性和女性之间的差异要小得多，我们不再看到下降。这是因为女性往往比男性活得更长，所以在老年时，活着的女性会更多的人被厕所伤害。

```
summary %>%
  ggplot(aes(age, rate, colour = sex)) +
  geom_line(na.rm = TRUE) +
  labs(y = "Injuries per 10,000 people")
```

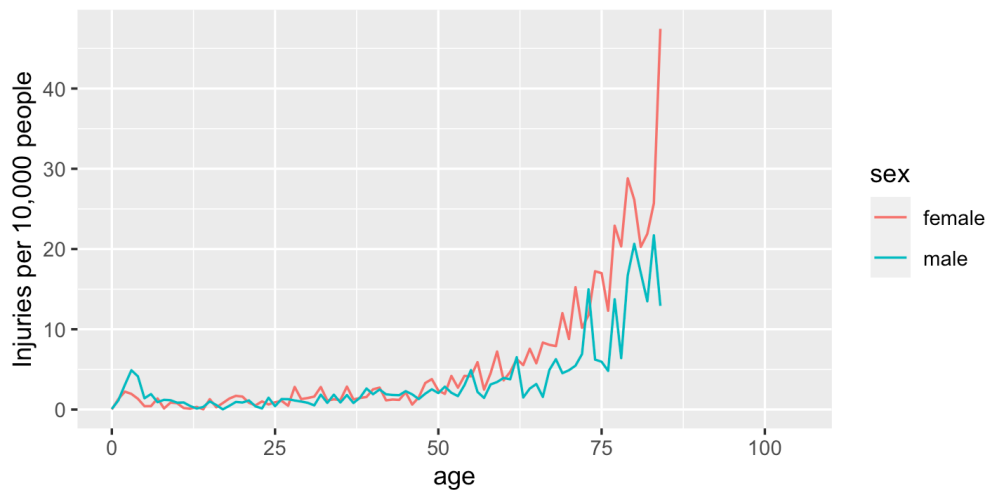


图4.2: 每10,000人的估计受伤率, 按年龄和性别细分 (请注意, 比率只上升到80岁, 因为我找不到80岁以上的人口数据。)

最后, 我们可以看看一些描述。浏览这些是检查我们的假设, 并产生新想法以供进一步探索的非正式方式。在这里, 我抽取了10个随机样本:

```
selected %>%
  sample_n(10) %>%
  pull(narrative)
#> [1] "79 YOM LOW BACK PAIN AFTER SLIPPING OFF TOILET AT HOME 1 WEEK AGO.DX DEGENERATIVE
DISC DISEASE, BLUMBAR, MECHANICAL FALL, SEPSIS"
#> [2] "88YOF PRESENTS AFTER FALLING OFF THE TOILET AT HOME AND HITTING CHEST WALL. DX:
RIGHT CHEST WALL PAIN S/P FALL.*"
#> [3] "73YOF WAS SITTING ON A PUBLIC TOILET AND PASSED OUT FELL OF FONTO HEADCLOSED HEAD
INJURY ADMITTED FOR SYNCOPE"
#> [4] "97 YOM FELL HITTING HEAD ON TOILET SEAT.DX:  NECK PX, BACK PX, FREQUENT FALLS."

#> [5] "32YOM FROM RENEWAL CENTER WAS SNORTING *** AND FELL OFF THE TOILET STRIKING HIS
HEAD DX MINOR CHI"
#> [6] "71YOF LOST BALANCE SITTING ON TOILET AT HOME AND INJURED SHOULDER.  DX:  LT
SHOULDER INJURY."
#> [7] "99YOF FALL OFF TOILET AND C/O SHOULDER PAIN/  CONTUSION R SHOULDER"

#> [8] "26 YOM TRIPPED AND FELL STRIKING KAND ON TOILET SEAT.DX:  R HAND LAC 5 CM."

#> [9] "86YOF TURNED HEAD & BECAME DIZZY, FELL AGAINST TOILET INJURING SHOULDER/CT
SHOULDER"
#> [10] "70 YOF - SYNCOPE - PT WAS SITTING ON TOILET AND FELL @ HOME."
```

在为一个产品进行了这次探索后, 如果我们能够轻松地其他产品进行探索, 而无需重新键入代码, 那就太好了。所以让我们制作一个shiny应用程序吧!

4.4 Prototype

在构建一个复杂的应用程序时，我强烈建议尽可能从简单地分析开始，这样你就可以在开始做更复杂的事情之前确认基本的逻辑。在这里，我将从一个输入（产品代码）、三个表格和一个图开始。

在设计第一个原型时，挑战在于使其“尽可能简单”。快速了解和掌控基础分析内容和设计增量应用程序之间存在着紧密关系。任何一种极端都可能是糟糕的：如果你的设计过于狭隘，你以后会花很多时间重新设计你的应用程序；如果你的设计过于严格，你会花很多时间编写代码，这些代码后来会被搁置在底层。为了帮助取得正确的平衡，在编写代码之前，我经常用铅笔和纸画一些草图来快速探索UI和反应图。

在这里，我决定为输入设置一行（接受我可能会在这个应用程序完成之前添加更多输入），为所有三个表设置一行（给每个表格4列，12列宽度的1/3），然后为绘图设置一行：

```
prod_codes <- setNames(products$prod_code, products$title)

ui <- fluidPage(
  fluidRow(
    column(6,
      selectInput("code", "Product", choices = prod_codes)
    ),
    fluidRow(
      column(4, tableOutput("diag")),
      column(4, tableOutput("body_part")),
      column(4, tableOutput("location"))
    ),
    fluidRow(
      column(12, plotOutput("age_sex"))
    )
  )
)
```

我们还没有讨论过 `fluidRow()` 和 `column()`，但您应该能够从上下文中猜到它们做了什么，我们将在第6.2.3节中讨论它们。另请注意在 `selectInput()` 中的 `choices` 使用 `setNames()`：这在UI中显示产品名称，并将产品代码返回给server函数。

server函数相对简单。我首先将上一节中创建的 `selected` 和 `summary` 变量转换为反应式表达式。这是一个合理的一般模式：您在数据分析中创建变量，将分析分解为一步一步的，并避免多次重新计算，而反应式表达式在Shiny应用程序中发挥着相同的作用。

通常，在启动Shiny应用程序之前，花一点时间清理分析代码是一个好主意，这样在添加反应式表达式的额外复杂性之前，您可以在常规R代码中思考这些问题。

```
server <- function(input, output, session) {
  selected <- reactive(injuries %>% filter(prod_code == input$code))

  output$diag <- renderTable(
    selected() %>% count(diag, wt = weight, sort = TRUE)
  )
  output$body_part <- renderTable(
    selected() %>% count(body_part, wt = weight, sort = TRUE)
  )
}
```

```

output$location <- renderTable(
  selected() %>% count(location, wt = weight, sort = TRUE)
)

summary <- reactive({
  selected() %>%
    count(age, sex, wt = weight) %>%
    left_join(population, by = c("age", "sex")) %>%
    mutate(rate = n / population * 1e4)
})

output$age_sex <- renderPlot({
  summary() %>%
    ggplot(aes(age, n, colour = sex)) +
    geom_line() +
    labs(y = "Estimated number of injuries")
}, res = 96)
}

```

请注意，在这里创建 `summary` 反应式表达式并非绝对必要，因为它仅由单个渲染函数使用。但保持计算和绘图分开是良好做法，因为它使应用程序的流程更容易理解，并在未来更容易维护。

生成的应用程序的屏幕截图如图4.3所示。您可以在<https://github.com/hadley/mastering-shiny/tree/master/neiss/prototype.R>上找到源代码，并在<https://hadley.shinyapps.io/ms-prototype/>上尝试该应用程序的实时版本。

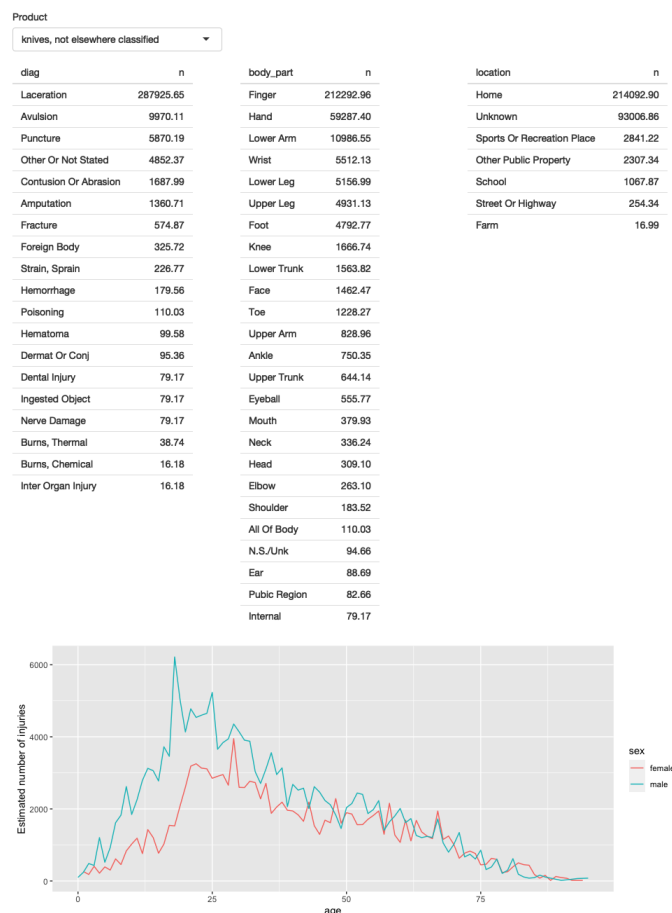


图4.3：NEISS探索应用程序的第一个原型

4.5 Polish tables

现在我们已经有了基本组件并工作了，我们可以逐步改进我们的应用程序。这个应用程序的第一个问题是，它在表格中显示了很多信息，我们可能只想要特定关注的。要解决这个问题，我们首先需要弄清楚如何截断表格。我选择使用forcats函数的组合来做到这一点：我将变量转换为因子，按水平的频率排序，然后在前5名之后将所有水平合并在一起。

```
injuries %>%
  mutate(diag = fct_lump(fct_infreq(diag), n = 5)) %>%
  group_by(diag) %>%
  summarise(n = as.integer(sum(weight)))
#> # A tibble: 6 × 2
#>   diag          n
#>   <fct>      <int>
#> 1 Other Or Not Stated 1806436
#> 2 Fracture          1558961
#> 3 Laceration        1432407
#> 4 Strain, Sprain     1432556
#> 5 Contusion Or Abrasion 1451987
#> 6 Other            1929147
```

因为我知道怎么做，所以我写了一个小函数来为任何变量自动化。细节在这里并不重要，但我们将在第12章中讲到它们。您也可以通过复制和粘贴来解决问题，所以不要担心代码看起来完全陌生。

```
count_top <- function(df, var, n = 5) {
  df %>%
    mutate({{ var }} := fct_lump(fct_infreq({{ var }}), n = n)) %>%
    group_by({{ var }}) %>%
    summarise(n = as.integer(sum(weight)))
}
```

然后我在server函数中使用这个：

```
output$diag <- renderTable(count_top(selected(), diag), width = "100%")
output$body_part <- renderTable(count_top(selected(), body_part), width = "100%")
output$location <- renderTable(count_top(selected(), location), width = "100%")
```

我做了另一个更改来改善应用程序的美感：我强迫所有表格占用最大宽度（即填充它们显示的列）。这使得输出更美观，因为它减少了附带变化的数量。

生成的应用程序的屏幕截图如图4.4所示。您可以在<https://github.com/hadley/mastering-shiny/tree/master/neiss/polish-tables.R>上找到源代码，并在<https://hadley.shinyapps.io/ms-polish-tables>上试用该应用程序的实时版本。

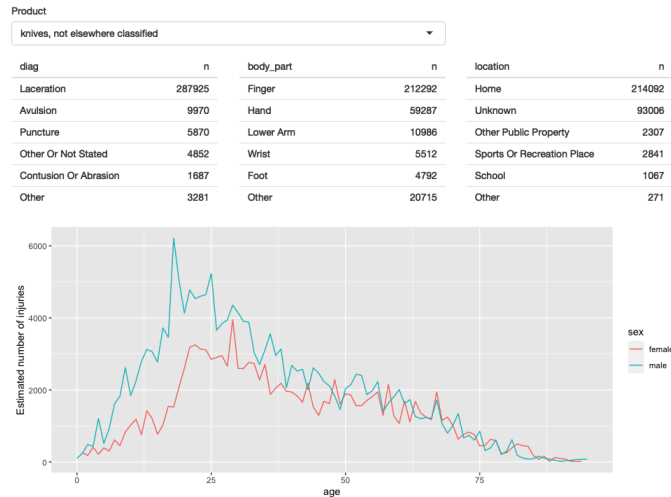


图4.4：应用程序的第二次迭代通过仅显示汇总表中最常见的行来改善显示

4.6 Rate vs count

到目前为止，我们只显示一个情节，但我们想让用户在可视化受伤人数或人口标准化率之间做出选择。首先，我向用户界面添加一个控件。在这里，我选择使用 `selectInput()`，因为它使两种状态都显式，并且将来很容易添加新状态：

```
fluidRow(
  column(8,
    selectInput("code", "Product",
      choices = setNames(products$prod_code, products$title),
      width = "100%"
    )
  ),
  column(2, selectInput("y", "Y axis", c("rate", "count")))
),
```

(我默认 `rate`，因为我认为它更安全；你不需要了解人口分布来正确解释图。)

然后我在生成绘图时以该输入为条件：

```
output$age_sex <- renderPlot({
  if (input$y == "count") {
    summary() %>%
      ggplot(aes(age, n, colour = sex)) +
      geom_line() +
      labs(y = "Estimated number of injuries")
  } else {
    summary() %>%
      ggplot(aes(age, rate, colour = sex)) +
      geom_line(na.rm = TRUE) +
      labs(y = "Injuries per 10,000 people")
  }
}, res = 96)
```

生成的应用程序的屏幕截图如图4.5所示。您可以在<https://github.com/hadley/mastering-shiny/tree/master/neiss/rate-vs-count.R>上找到源代码，并在<https://hadley.shinyapps.io/ms-rate-vs-count>上尝试该应用程序的实时版本。

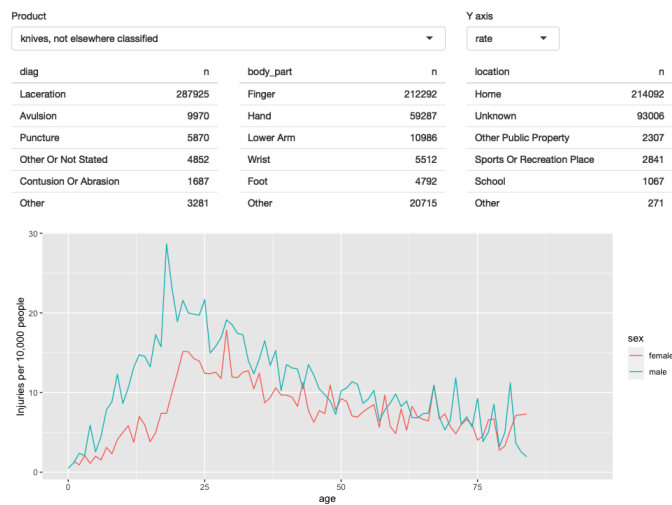


图4.5：在此迭代中，我们让用户能够在y轴上显示计数或人口标准化率之间切换。

4.7 Narrative

最后，我想提供一些访问叙事的方法，因为它们非常有趣，它们提供了一种非正式的方式来交叉检查您在查看图时提出的假设。在R代码中，我一次采样多个叙事，但没有理由在您可以交互式探索的应用程序中这样做。

解决方案有两个部分。首先，我们在用户界面底部添加一行新行。我使用操作按钮触发一个新故事，并将叙事放在 `textOutput()`

```
fluidRow(
  column(2, actionButton("story", "Tell me a story")),
  column(10, textOutput("narrative"))
)
```

然后，我使用 `eventReactive()` 创建一个反应，仅在单击按钮或基础数据更改时更新。

```
narrative_sample <- eventReactive(
  list(input$story, selected()),
  selected() %>% pull(narrative) %>% sample(1)
)
output$narrative <- renderText(narrative_sample())
```

生成的应用程序的屏幕截图如图4.6所示。您可以在<https://github.com/hadley/mastering-shiny/tree/master/neiss/narrative.R>上找到源代码，并在<https://hadley.shinyapps.io/ms-narrative>上尝试该应用程序的实时版本。

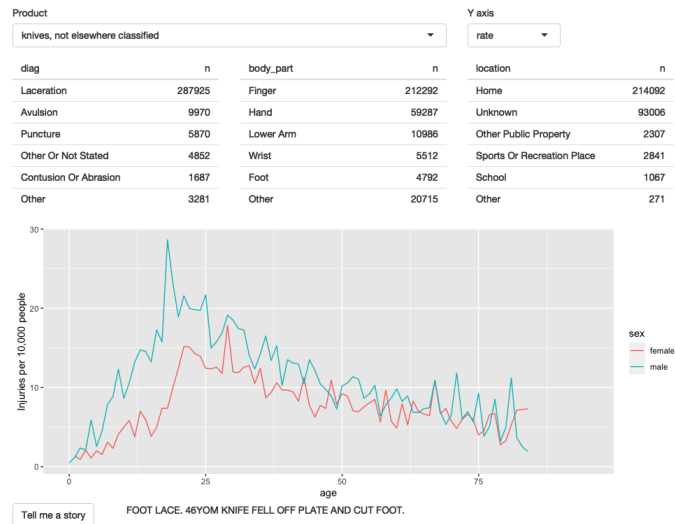


图4.6：最终迭代增加了从所选中提取随机叙事的能力

4.8 Exercises

1. 为每个应用程序绘制反应图。
2. 如果您在减少汇总表的代码中翻转 `fct_infreq()` 和 `fct_lump()` 会发生什么？
3. 添加一个输入控件，让用户决定在汇总表中显示多少行。
4. 提供一种通过前进和后退按钮系统地完成每个叙事的方法。

高级：使叙事列表“循环”，以便从最后一个叙事向前推进，将你带到第一个。

4.9 Summary

现在您已经掌握了shiny应用程序的基础知识，以下七章将为您提供一系列重要技术。一旦你读了关于工作流程的下一章，我建议 you 略读剩下的章节，这样你就可以很好地了解它们所涵盖的内容，然后当你需要shiny应用程序的技术时，再去阅读相关章节。