

Contents

1	1
1.1 添加 UI 控件	1
1.2	1
2 基础 UI	3
2.1 输出	3
2.1.1 表格	3
2.1.2 绘图	3
2.1.3 下载	3
3 响应式基础	5
3.1 server 函数	5
3.1.1 input	5
3.1.2 输出	5
3.2 响应式编程	6
3.2.1 命令式编程 imperative programming 与声明式 declarative programming 编程	6
3.2.2 响应图	6
3.2.3 响应表达式	6
3.2.4 执行顺序	7
3.3 响应表达式	7
3.3.1 简化图形	7
3.4 控制评估时间	7
3.4.1 定时失效	7
3.4.2 在点击时执行	8
3.5 观察员	8

I	Shiny in Action	11
4	布局、主题、HTML	13
4.1	单页布局	13
4.1.1	页面功能	13
4.1.2	带侧边栏的页面	13
4.1.3	多行	15

Chapter 1

1.1 添加 UI 控件

- `fluidPage()` 是一个布局函数，用于设置页面的基本视觉结构。
- `selectInput()` 是一个输入控件，允许用户通过提供值与应用程序交互。
- `verbatimTextOutput()` 和 `tableOutput()` 是输出控件，告诉 Shiny 将渲染输出放在哪里。`verbatimTextOutput()` 显示代码并 `tableOutput()` 显示表格。

1.2

您可以通过包装一段代码并将 `reactive({...})` 其分配给变量来创建反应式表达式，并且可以通过像函数一样调用它来使用反应式表达式。但是，虽然看起来您正在调用函数，但响应式表达式有一个重要的区别：它仅在第一次调用时运行，然后缓存其结果，直到需要更新为止。

Chapter 2

基础 UI

2.1 输出

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，并且通常与 `textOutput()`
- `renderPrint()` 打印结果，就像您在 R 控制台中一样，并且通常与 `verbatimTextOutput()`

2.1.1 表格

有两种用于在表中显示数据框的选项：

- `tableOutput()` 与 `renderTable()` 渲染一个静态数据表，一次性显示所有数据。
- `dataTableOutput()` 与 `renderDataTable()` 呈现一个动态表，显示固定数量的行以及用于更改哪些行可见的控件。

`tableOutput()` 对于小型、固定的 `summary`（例如模型系数）最有用；如果您想向用户公开完整的数据框，则 `dataTableOutput()` 最合适。

2.1.2 绘图

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

2.1.3 下载

您可以让用户使用 `downloadButton()` 或 `downloadLink()` 来下载文件。

Chapter 3

响应式基础

3.1 server 函数

3.1.1 input

参数 `input` 是一个类似列表的对象，其中包含从浏览器发送的所有输入数据，根据输入 ID 命名。与普通的列表不同，`input` 对象是只读的。如果你尝试在服务函数内的修改输入，你将收到错误。发生此错误是因为 `input` 反映了浏览器中发生的情况，而浏览器是 Shiny 的“单一事实来源”。如果你可以修改 R 中的值，则可能会导致不一致，即输入滑块在浏览器中表示一件事，而 `input$count` 在 R 中表示不同的内容。这将使编程变得具有挑战性！稍后，在 ?? 中，你将学习如何使用诸如 `updateNumericInput()` 修改浏览器中的值之类的功能，然后 `input$count` 进行相应的更新。

关于 `input` 更重要的一件事是：它对谁可以阅读是有选择性的。要读取 `input`，必须处于由 `renderText()` 或 `reactive()` 函数创建的响应式上下文中。

3.1.2 输出

`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。主要区别在于使用它来发送输出而不是接收输入。你总是要把 `output` 对象与 `render` 函数结合使用。

渲染函数做了两件事：

- 它设置了一个特殊的响应上下文，可以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。

与 `input` 一样，`output` 对如何使用它很挑剔。

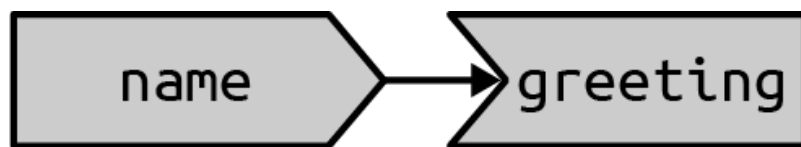


Figure 3.1: 响应图显示了输入和输出的连接方式

3.2 响应式编程

Shiny 的重要思想：你不需要告诉输出何时更新，因为 Shiny 会自动为你计算出来。

3.2.1 命令式编程 **imperative programming** 与声明式 **declarative programming** 编程

命令和 **recipes** 之间的区别是两种重要编程风格之间的主要区别之一：

- 在命令式编程中，你发出特定命令，它会立即执行。这是你在分析脚本中习惯的编程风格：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。
- 在声明式编程中，你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。这是你在 **Shiny** 中使用的编程风格。

命令式代码是 **assertive**；声明式代码是 **passive-aggressive**。

3.2.2 响应图

响应图是了解应用程序工作原理的强大工具。随着你的应用程序变得越来越复杂，制作响应图的快速高级草图通常很有用，以提醒你所有部分如何组合在一起。在本书中，我们将向你展示响应图，以帮助你理解示例的工作原理，稍后在 ?? 中，你将学习如何使用 [reactlog](#) 来为你绘制图表。

3.2.3 响应表达式

你将在响应图中看到一个更重要的组件：响应表达式。响应式表达式接受输入并产生输出，因此它们具有结合输入和输出特征的形状。希望这些形状能帮助你记住组件如何组合在一起。



Figure 3.2: 输入和表达式是响应式生产者；表达式和输出是响应式消费者

3.2.4 执行顺序

重要的是要理解代码运行的顺序完全由响应图决定。这与大多数 R 代码不同，大多数 R 代码的执行顺序由行的顺序决定。

3.3 响应表达式

响应式表达式具有输入和输出的风格：

- 与输入一样，你可以在输出中使用响应式表达式的结果。
- 与输出一样，响应式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（**producer**）来指代响应式输入和表达式，使用消费者（**consumer**）来指代响应式表达式和输出。

3.3.1 简化图形

你可能熟悉编程的“三规则”：每当你将某些内容复制并粘贴三次时，你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，我认为你应该考虑一规则：每当你复制并粘贴某些内容时，你应该考虑将重复的代码提取到响应式表达式中。该规则对于 Shiny 来说更为严格，因为响应式表达式不仅使人们更容易理解代码，还提高了 Shiny 有效重新运行代码的能力。

3.4 控制评估时间

3.4.1 定时失效

想象一下，你想通过不断地重新，以便你看到动画而不是静态图。我们可以通过一个新功能来提高更新频率：`reactiveTimer()`。`reactiveTimer()` 是一个响应式表达式，依赖于隐藏输

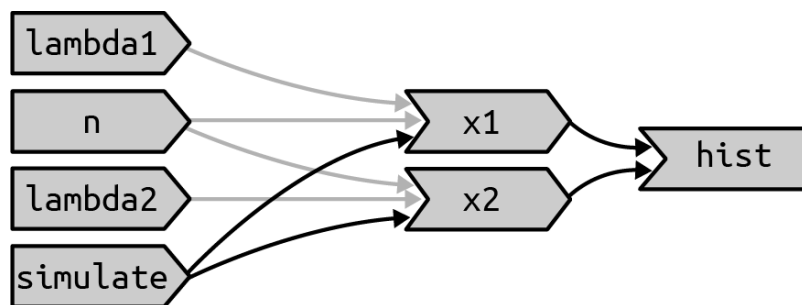


Figure 3.3: 根据需要，`lambda1`、`lambda2`、`n` 不再对 `x1`、`x2` 具有响应性依赖：更改它们的值将不会触发计算。将箭头保留为非常浅的灰色只是为了提醒你 `x1` 和 `x2` 继续使用这些值，但不再对它们产生响应性依赖。

入：当前时间。当你希望响应式表达式比其他方式更频繁地使其自身无效时，可以使用 `reactiveTimer()`。

3.4.2 在点击时执行

假设在一个定时器中，每 0.5 秒执行一次代码，但是代码的执行时间需要 1 秒，那么 Shiny 需要做的事情越来越多。如果用户快速点击更改某个参数同样会直到 Shiny 需要执行的事情越来越多，尤其是在涉及昂贵计算时。

如果你的应用程序中出现这种情况，你可能希望要求用户通过单击按钮来选择执行昂贵的计算。这是 `actionButton()` 一个很好的用例。

我们需要一个新工具 `eventReactive()`：一种使用输入值而不对其产生响应性依赖的方法，它有两个参数：第一个参数指定要依赖的内容，第二个参数指定要计算的内容。

Figure 3.3 体现了这种思想。

3.5 观察员

有些操作不会在页面中展示，但是需要记录，比如说调试消息、发送数据等等，这应该使用输入和输出的 `render`，而是需要使用观察员 `observers`。

`observeEvent()` 它为你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`。第一个参数是要依赖的输入或表达式；第二个参数是将运行的代码。

`observeEvent()` 和 `eventReactive()` 之间有两个重要的区别：

- 你没有将 `observeEvent()` 的结果分配给变量

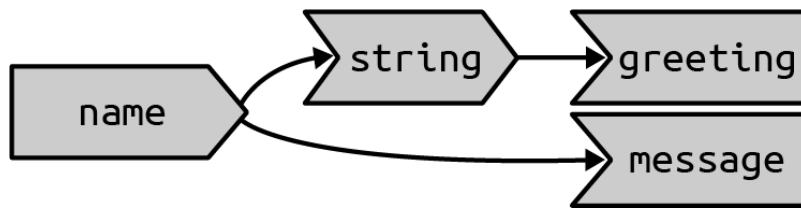


Figure 3.4

- 你无法从其他响应性消费者那里引用它

观察员和产出密切相关。您可以将输出视为具有特殊的副作用：更新用户浏览器中的 HTML。为了强调这种接近性，我们将在响应图中以相同的方式绘制它们。这会产生 [Figure 3.4](#) 所示的响应图。

Part I

Shiny in Action

Chapter 4

布局、主题、HTML

4.1 单页布局

布局函数提供应用程序的高级视觉结构。布局是由函数调用的层次结构创建的，其中 R 中的层次结构与生成的 HTML 中的层次结构相匹配。这有助于你理解布局代码。

4.1.1 页面功能

最重要但最无趣的布局函数是 `fluidPage()`，它看起来是一个非常无聊的应用程序，但幕后有很多工作，因为 `fluidPage()` 设置了 Shiny 所需的所有 HTML、CSS 和 JavaScript。

除了之外 `fluidPage()`，Shiny 还提供了一些其他页面函数，可以在更特殊的情况下派上用场：`fixedPage()` 和 `fillPage()`。`fixedPage()` 工作原理类似 `fluidPage()`，但有一个固定的最大宽度，这可以防止你的应用程序在更大的屏幕上变得不合理的宽度。`fillPage()` 填充浏览器的整个高度，如果你想制作占据整个屏幕的绘图，则非常有用。你可以在他们的文档中找到详细信息。

4.1.2 带侧边栏的页面

要制作更复杂的布局，你需要在 `fluidPage()`。例如，要制作一个左侧输入、右侧输出的两列布局，你可以使用 `sidebarLayout()`（以及它的朋友 `titlePanel()`、`sidebarPanel()` 和 `mainPanel()`）。

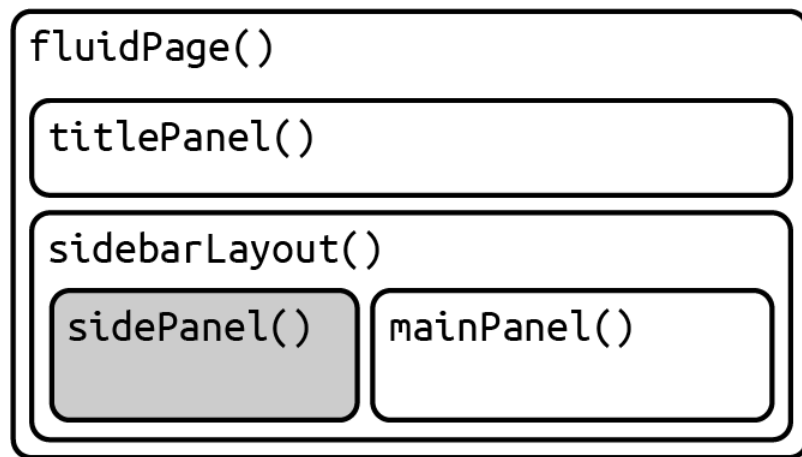


Figure 4.1: 带有侧边栏的基本应用程序的结构

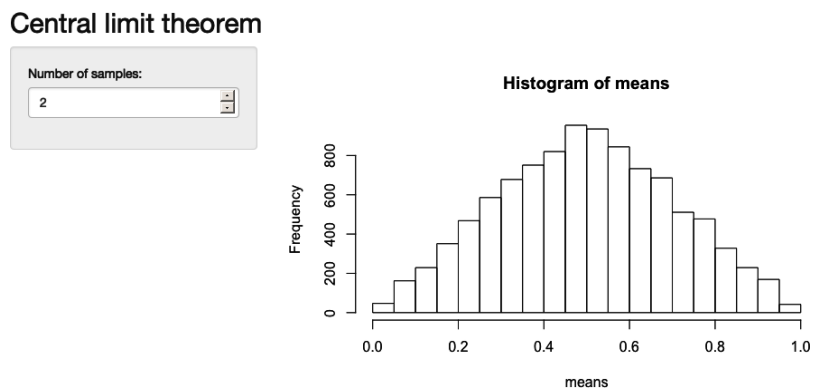


Figure 4.2: 常见的应用程序设计是将控件放在侧边栏中并在主面板中显示结果

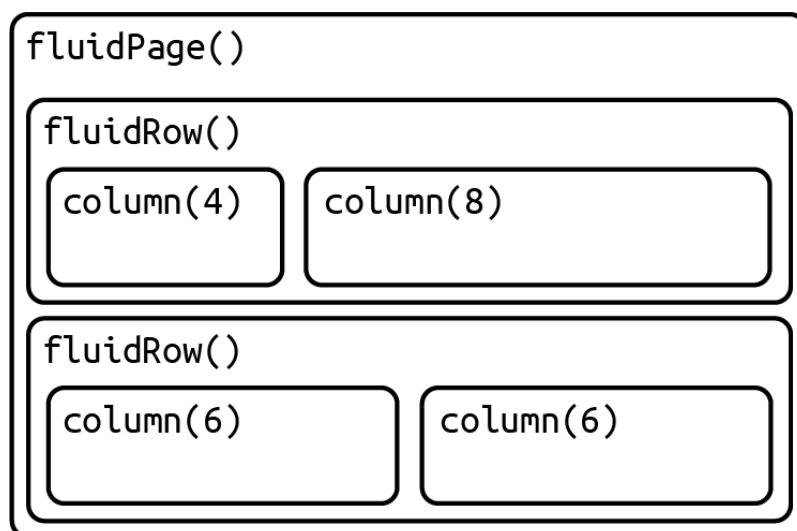


Figure 4.3: 简单多行应用程序的底层结构

4.1.3 多行

在底层，`sidebarLayout()` 它构建在灵活的多行布局之上，你可以直接使用它来创建视觉上更复杂的应用程序。像往常一样，你从 `fluidPage()` 开始。然后，你可以使用 `fluidRow()` 来创建行，并使用 `column()` 来创建列。

每行由 12 列组成，`column()` 第一个参数给出要占用的列数。12 列布局为你提供了极大的灵活性，因为你可以轻松创建 2 列、3 列或 4 列布局，或使用窄列来创建间隔。

4.2 多页面布局

随着你的应用程序变得越来越复杂，可能无法将所有内容都放在一个页面上。在本节中，你将学习 `tabPanel()` 创建多个页面错觉的各种用法。这是一种错觉，因为你仍然拥有一个带有单个底层 HTML 文件的应用程序，但它现在被分成了几部分，并且一次只能看到一个部分。

多页面应用程序与模块配合得特别好，你将在 ?? 中了解这些模块。模块允许你以与划分用户界面相同的方式划分服务器功能，创建仅通过明确定义的连接进行交互的独立组件。

4.2.1 选项卡集

将页面分成多个部分的简单方法是使用 `tabsetPanel()` 以及与它近似的 `tabPanel()`。