Contents

Chapter 1

Your First Shiny App

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 使用反应式表达式减少重复

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$count 在 R 中表示不同的内容。这将使编程变得具有挑战性！稍后，在 Chapter 6 中，你将学习如何使用诸如 updateNumericInput() 修改浏览器中的值之类的功能，然后 input$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。

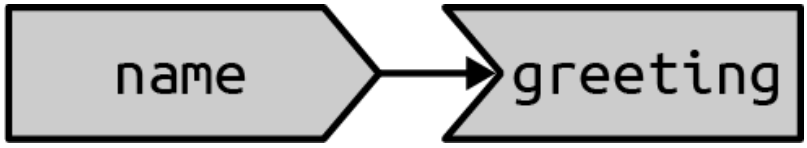

Figure 3.1: 响应图显示了输入和输出的连接方式

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 Chapter 12 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

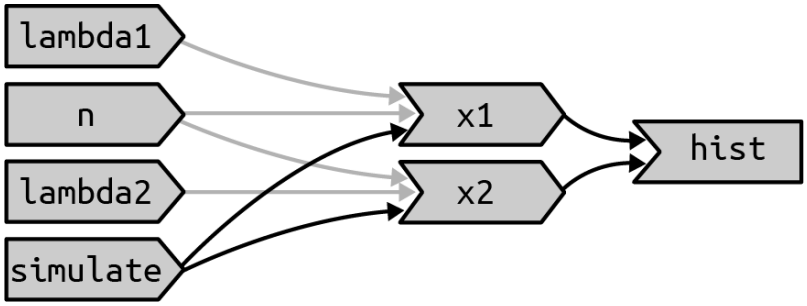

Figure 3.3: 根据需要，lambda1、lambda2、n 不再对 x1、 x2 具有响应性依赖：更改它们的值将不会触发计算。将箭头保留为非常浅的灰色只是为了提醒你 x1 和 x2 继续使用这些值，但不再对它们产生响应性依赖。

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，比如说调式消息、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

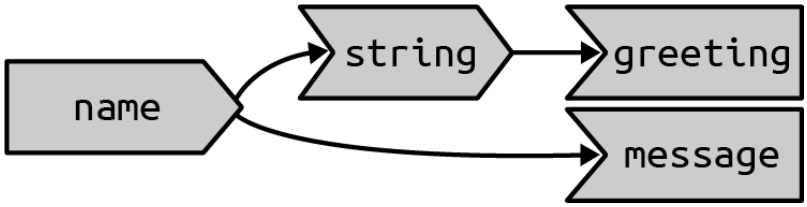

Figure 3.4

- 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()）。

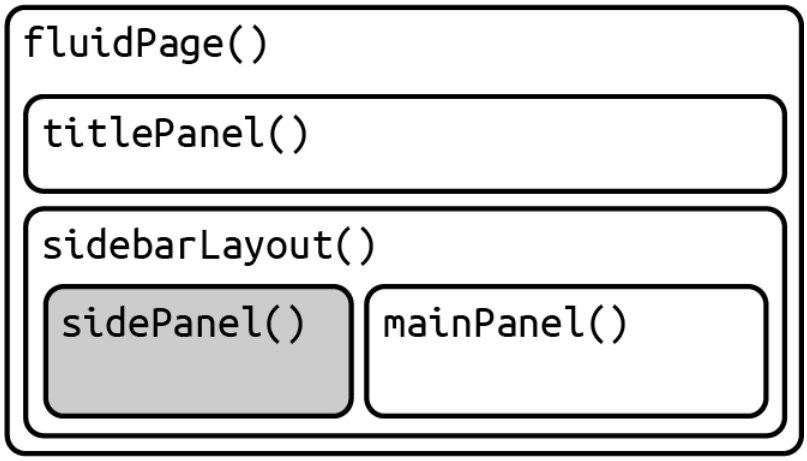

Figure 4.1: 带有侧边栏的基本应用程序的结构

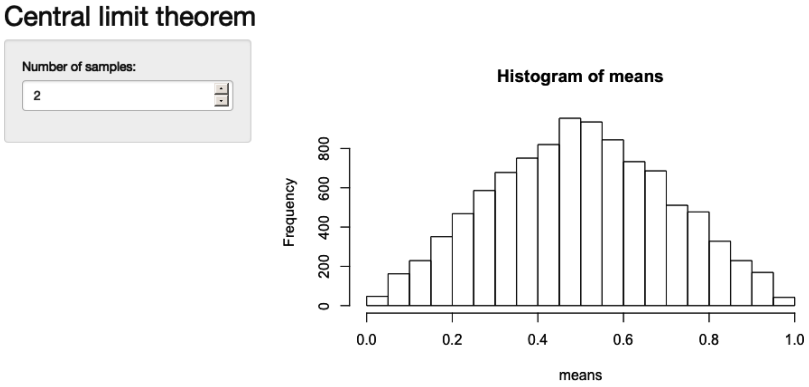

Figure 4.2: 常见的应用程序设计是将控件放在侧边栏中并在主面板中显示结果

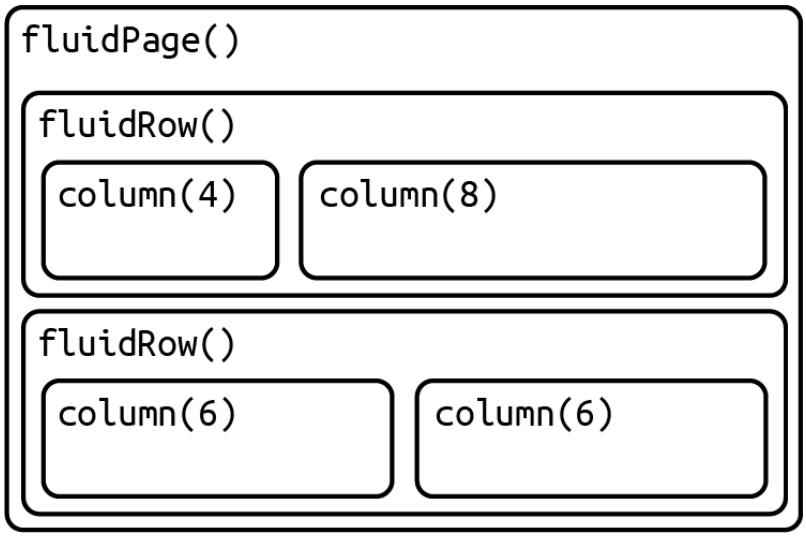

Figure 4.3: 简单多行应用程序的底层结构

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()。如果你想知道用户选择了哪个选项卡，你可以提供参数 id，它将成为 tabsetPanel() 的输入。

注意：tabsetPanel()可以在应用程序中的任何位置使用；如果需要的话，将选项卡集嵌套在其他组件（包括选项卡集!）中是完全可以的。

4.2.2 导航列表和导航栏

由于选项卡是水平显示的，因此可以使用的选项卡数量存在根本限制，特别是当它们具有长标题时。 navbarPage() 与 navbarMenu() 提供两种替代布局，让你可以使用更多带有更长标题的选项卡。

navlistPanel() 类似于 tabsetPanel()，但它不是水平运行选项卡标题，而是在侧边栏中垂直显示它们。它还允许你添加带有纯字符串的标题。另一种方法是使用 navbarPage()：它仍然水平运行选项卡标题，但你可以使用 navbarMenu() 添加下拉菜单以获得额外的层次结构级别。

4.3 Bootstrap

要继续您的应用程序定制之旅，您需要更多地了解 Shiny 使用的Bootstrap框架。Bootstrap 是 HTML 约定、CSS 样式和 JS 片段的集合，它们捆绑成一种方便的形式。Bootstrap 源自最初为 Twitter 开发的框架，在过去 10 年中已发展成为网络上最流行的 CSS 框架之一。

作为一个 Shiny 开发者，你不需要对 Bootstrap 考虑太多，因为 Shiny 函数会自动为你生成 bootstrap 兼容的 HTML。但很高兴知道 Bootstrap 的存在，因为这样：

- 您可以使用 bslib::bs_theme() 自定义代码的视觉外观
- 您可以使用class参数通过 Bootstrap 类名来自定义一些布局、输入和输出
- 您可以创建自己的函数来生成Shiny 未提供的Bootstrap 组件

也可以使用完全不同的 CSS 框架。许多现有的 R 包通过包装 Bootstrap 的流行替代品使这一切变得容易：

- shiny.semantic，由 Appsilon 开发，建立在 formantic UI 之上。
- shinyMobile 由 RInterface 开发，构建于框架 7之上，专为移动应用程序而设计。
- shinymaterial 由 Eric Anderson开发，建立在 Google 的 Material 设计框架之上。
- shinydashboard 也由 RStudio 提供，提供了一个旨在创建仪表板的布局系统。

您可以在 awesome-shiny-extensions 找到更完整且积极维护的列表。

4.4 主题

Bootstrap 在 R 社区中如此普遍，以至于很容易产生风格疲劳：一段时间后，每个 Shiny 应用程序和 Rmd 开始看起来都一样。解决方案是使用 bslib 包进行主题化。bslib 是相对较新

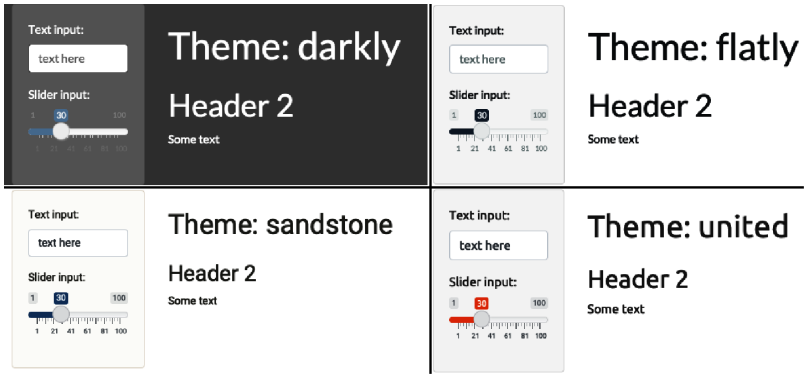

Figure 4.4: 同一个应用程序具有四个 bootswatch 主题：darkly、flatly、sandstone 和 United

的软件包，它允许您覆盖许多 Bootstrap 默认值，以创建独特的外观。

4.4.1 shiny 主题

更改应用程序整体外观的最简单方法是使用的参数选择预制的“bootswatch”主题。

预览和自定义主题的一种简单方法是使用 `bslib::bs_theme_preview(theme)`。这将打开一个闪亮的应用程序，显示应用许多标准控件时主题的外观，并为您提供用于自定义最重要参数的交互式控件。

4.4.2 plot 主题

如果您对应用程序的样式进行了大量自定义，您可能还需要自定义绘图以匹配。幸运的是，这真的很容易，这要归功于 thematic 包，它自动为 ggplot2、lattice和 base 提供主题。只需在您的服务器调用 `thematic_shiny()` 功能即可。这将自动确定应用程序主题的所有设置。

Chapter 5

图形

5.1 交互性

plotOutput() 最酷的事情之一是，它不仅可以作为显示绘图的输出，还可以作为响应指针事件的输入。这允许你创建交互式图形，用户可以直接与绘图上的数据进行交互。交互式图形是一种强大的工具，具有广泛的应用范围。

5.1.1 基础知识

绘图可以响应四种不同的鼠标23事件：click、dblclick（双击）、hover（当鼠标在同一位置停留一小会儿时）和brush（矩形选择工具）。要将这些事件转换为闪亮的输入，你可以向相应的参数提供一个字符串 plotOutput()，例如 `plotOutput("plot", click = "plot_click")`。这将创建一个 `input$plot_click` 可用于处理绘图上的鼠标单击的。

5.1.2 点击

点事件返回一个相对丰富的列表，其中包含大量信息。最重要的组成部分是 x 和 y，它们给出了数据坐标中事件的位置。但我不会谈论这种数据结构，因为你只在相对罕见的情况下需要。相反，你将使用 nearPoints() 帮助程序，它返回一个数据框，其中包含与点击接近的行（一个观测值），处理一堆繁琐的细节。

这里我们给出 nearPoints() 四个参数：绘图下的数据框、输入事件以及轴上变量的名称。如果你使用 ggplot2，则只需提供前两个参数，因为 xvar 和 yvar 可以从绘图数据结构自动估算。

你可能想知道到底 nearPoints() 返回了什么。这是一个使用的好地方 browser()。

另一种使用方法 nearPoints() 是 `allRows = TRUE` 与 `addDist = TRUE` 和一起使用。这将返回带有两个新列的原始数据框：

- `dist_` 给出行和事件之间的距离（以像素为单位）。
- `selected_` 表示它是否靠近单击事件（即，是否是在 `allRows = FALSE` 时返回的行）。

5.1.3 其他点事件

同样的方法同样适用于 click、dblclick 和 hover：只需更改参数的名称即可。clickOpts() 如果需要，你可以通过提供 dblclickOpts() 或 hoverOpts() 来代替给出输入 ID 的字符串，从而获得对事件的额外控制。

你可以在一张图上使用多种交互类型。只需确保向用户解释他们可以做什么：使用鼠标事件与应用程序交互的一个缺点是它们不能立即被发现。

5.1.4 画笔

在绘图上选择点的另一种方法是使用画笔(brushing)，即由四个边定义的矩形选择。click在Shiny中，一旦你掌握了 click 和 nearPoints() 画笔的使用就很简单：你只需切换到brush 参数和 brushedPoints() 助手即可。

brushOpts() 用于控制颜色 (fill 和 stroke)，或 direction = “x” 或 “y” 将 brush 限制为单一维度（例如，对于刷牙时间序列很有用）。

5.1.5 修改绘图

当你在与之交互的同一个绘图中显示变化时，交互性的真正魅力就显现出来了。

reactiveVal() 与 reactive() 类似。你可以通过调用其初始值来创建响应式值 reactiveVal()，并以与响应式相同的方式检索该值。最大的区别在于，你还可以更新响应性值，并且引用它的所有响应性使用者都将重新计算。响应式值使用特殊的语法进行更新 - 你可以像函数一样调用它，第一个参数是新值。

5.1.6 交互限制

在我们继续之前，了解交互式图中的基本数据流以了解它们的局限性非常重要。基本流程是这样的：

1. JavaScript 捕获鼠标事件。
2. Shiny 将鼠标事件数据发送回 R，告诉应用程序输入现在已过时。

3. 所有下游响应性消费者都被重新计算。
4. plotOutput() 生成一个新的 PNG 并将其发送到浏览器。

对于本地应用程序，瓶颈往往是绘制绘图所需的时间。根据情 plot 的复杂程度，这可能需要几分之一秒的时间。但对于托管应用程序，您还必须考虑将事件从浏览器传输到 R，然后将渲染的绘图从 R 传输回浏览器所需的时间。

5.2 动态高度和宽度

本章的其余部分不如交互式图形那么令人兴奋，但包含一些需要在某个地方介绍的重要材料。

首先，可以使绘图大小具有响应性，因此宽度和高度会根据用户操作而变化。为此，请为零参数函数 renderPlot() 的参数 width 和 height 提供值 - 现在必须在服务器而不是 UI 中定义这些函数，因为它们可以更改。这些函数应该没有参数并返回所需的像素大小。它们在响应性环境中进行评估，以便您可以动态调整绘图的大小。

5.3 图像

如果您想显示现有图像（而不是绘图），则可以使用 renderImage()。

renderImage()需要返回一个列表。唯一关键的参数是src图像文件的本地路径。您还可以额外提供：

- contentType，定义图像的 MIME 类型。如果未提供，Shiny 会根据文件扩展名进行猜测，因此仅当您的图像没有扩展名时才需要提供此文件
- 图像的 width 和 height（如果已知）
- 任何其他参数（例如class或 alt）将作为属性添加到 HTML `<img>` 中的标记中

您还必须提供 deleteFile 值。不幸的是，renderImage() 最初设计用于处理临时文件，因此它在渲染图像后会自动删除图像。这显然是非常危险的，因此在 Shiny 1.5.0 中行为发生了变化。现在，shiny 不再删除图像，而是强制您明确选择您想要的行为。

Chapter 6

向用户反馈

你通常可以通过让用户更深入地了解正在发生的事情来提高应用程序的可用性。当输入没有意义时，这可能会采取更好的消息形式，或者在需要很长时间的操作时采取进度条。

我们将从**验证**技术开始，当输入（或输入组合）处于无效状态时通知用户。然后，我们将继续进行**通知**，向用户发送一般消息，以及**进度条**，进度条提供由许多小步骤组成的耗时操作的详细信息。最后，我们将讨论**危险**的操作，以及如何通过确认对话框或撤消操作的能力让用户安心。

6.1 验证

你可以向用户提供的第一个也是最重要的反馈是他们给了你错误的输入。

6.1.1 验证输入

向用户提供额外反馈的一个好方法是通过 shinyFeedback 包。使用它是一个两步过程。首先，添加 useShinyFeedback() 到ui. 这将设置所需的 HTML 和 JavaScript，以实现有吸引力的错误消息显示。然后在你的 server() 函数中，你调用反馈函数之一：feedback()、feedbackWarning()、feedbackDanger()和 feedbackSuccess()。他们都有三个关键参数：

1. inputId，应放置反馈的输入的 id。
2. show，逻辑决定是否显示反馈。
3. text，要显示的文本。

它们还具有 color 和 icon 可用于进一步自定义外观的参数。

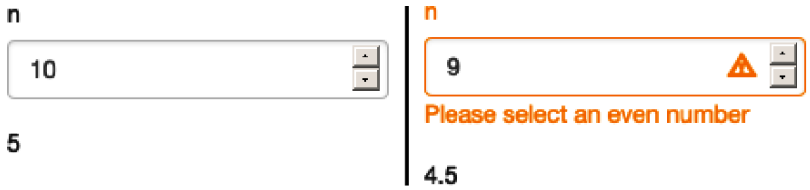

Figure 6.1: 用于 feedbackWarning() 显示无效输入的警告。左侧的应用程序显示有效的输入，右侧的应用程序显示无效（奇数）输入并带有警告消息。

请注意，虽然显示了错误消息，但输出仍会更新。通常你不希望出现这种情况，因为无效的输入可能会导致你不希望向用户显示的无信息 R 错误。要阻止输入触发响应性更改，你需要一个新工具：req()，“required”的缩写。当 req() 的输入不正确时，它会发送一个特殊信号来告诉 Shiny 响应式没有所需的所有输入，因此应该“暂停”。

6.1.2 用 req() 取消执行

It’s easiest to understand req() by starting outside of validation. You may have noticed that when you start an app, the complete reactive graph is computed even before the user does anything. This works well when you can choose meaningful default values for your inputs. But that’s not always possible, and sometimes you want to wait until the user actually does something. This tends to crop up with three controls:

- 在 textInput() 中，你已经使用过 `value = ""` 并且在用户键入某些内容之前不想执行任何操作。
- 在 selectInput() 中，你提供了一个空选择 `""` 并且在用户做出选择之前你不想执行任何操作。
- 在 fileInput() 中，在用户上传任何内容之前，其结果为空。

我们需要某种方法来“暂停”响应，以便在某些条件成立之前不会发生任何事情。这就是 req() 的工作，它在允许响应式生产者继续之前检查所需的值。

req() 通过发出特殊条件信号来工作[1]。这种特殊情况会导致所有下游响应和输出停止执行。从技术上讲，它使任何下游响应性消费者处于无效状态。

req() 被设计为仅当用户提供了值时才会继续，而不管输入控件 `req(input$x)` 的类型。如果需要，你还可以与自己的逻辑语句一起使用。

[1] https://adv-r.hadley.nz/conditions.html

6.1.3 req() 及验证

6.1.4 验证输出

当问题与单个输入相关时，shinyFeedback 非常有用。但有时无效状态是输入组合的结果。在这种情况下，将错误放在输入旁边（你会将其放在哪个输入旁边？）并没有真正意义，而是将其放在输出中更有意义。

你可以使用 shiny 内置的工具 validate() 来做到这一点。当在响应或输出内部调用时，validate(message) 停止执行其余代码，而是在任何下游输出中显示 message。

6.2 通知

如果没有问题而你只是想让用户知道发生了什么，那么你需要一个通知。在 Shiny 中，通知是通过 showNotification() 和堆叠在页面右下角创建的。共有三种基本使用方法 showNotification()：

1. 显示在固定时间后自动消失的瞬时通知。
2. 在进程启动时显示通知并在进程结束时将其删除。
3. 使用渐进式更新来更新单个通知。

6.2.1 瞬时通知

showNotification() 最简单的使用方法是使用单个参数调用它：你想要显示的消息。默认情况下，该消息将在 5 秒后消失，你可以通过设置覆盖该消息 duration，或者用户可以通过单击关闭按钮提前将其关闭。如果你想让通知更加突出，你可以将 type 参数设置为“message”、“warning”或“error”之一。

6.2.2 完成后移除

将通知的存在与长时间运行的任务联系起来通常很有用。在这种情况下，你希望在任务开始时显示通知，并在任务完成时删除通知。为此，你需要：

- 设置duration = NULL，closeButton = FALSE使通知保持可见，直到任务完成。
- 存储 showNotification() 返回的 id 值，然后将该值传递给 removeNotification()。最可靠的方法是使用on.exit()，这可以确保无论任务如何完成（成功或有错误），通知都会被删除。你可以在更改和恢复状态中了解更多 on.exit() 信息。

一般来说，这些类型的通知将以响应方式存在，因为这可以确保长时间运行的计算仅在需要时重新运行。

6.2.3 渐进式更新

多次调用 showNotification() 通常会创建多个通知。你可以通过捕获第一个调用的 id 并在后续调用中使用它来更新单个通知。如果你的长时间运行的任务有多个子组件，这非常有用。

6.3 进度条

对于长时间运行的任务，最好的反馈类型是进度条。除了告诉你在此过程中所处的位置外，它还可以帮助你估计需要多长时间。不幸的是，这两种技术都存在相同的主要缺点：要使用进度条，你需要能够将大任务划分为已知数量的小块，每个小块花费大致相同的时间。这通常很困难，特别是因为底层代码通常是用 C 编写的，并且无法向你传达进度更新。

6.3.1 shiny

要使用 Shiny 创建进度条，你需要使用 withProgress() 和 incProgress()。你首先将其包装在 withProgress(). 这会在代码启动时显示进度条，并在完成后自动将其删除。然后在每一步之后调用 incProgress()，第一个参数是进度条增量的量。默认情况下，进度条从 0 开始，到 1 结束，因此增量除以 1 除以步数将确保进度条在循环结束时完成。

6.3.2 waiter

内置进度条对于基础知识来说非常有用，但如果你想要提供更多视觉选项的东西，你可以尝试 waiter 包。waiter 包使用 R6 对象将所有与进度相关的功能捆绑到一个对象中。

默认显示是页面顶部的细进度条，但有多种方法可以自定义输出：

- 你可以用使用以下之一覆盖默认值 theme：
  - overlay：隐藏整个页面的不透明进度条
  - overlay-opacity：覆盖整个页面的半透明进度条
  - overlay-percent：不透明的进度条，还显示数字百分比。
- 你可以通过设置 selector 参数将其覆盖在现有输入或输出上，而不是显示整个页面的进度条

6.3.3 旋转器

有时你并不确切知道操作需要多长时间，而你只想显示一个动画旋转器，让用户放心某些事情正在发生。

一个更简单的替代方案是使用Dean Attali 的 shinycssloaders 包。它使用 JavaScript 来监听 Shiny 事件，因此它甚至不需要服务器端的任何代码。相反，你只需使用 shinycssloaders::withSpinner() 包装你想要在无效时自动获取微调器的输出即可。

6.4 确认和撤销

有时某个操作可能存在危险，你要么想确保用户确实想要执行该操作，要么想让他们能够在为时已晚之前退出。

6.4.1 显式确认

保护用户免遭意外执行危险操作的最简单方法是要求明确的确认。最简单的方法是使用一个对话框，强制用户从一小组操作中进行选择。在 Shiny 中，你可以使用 modalDialog()。这被称为“模式”对话框，因为它创建了一种新的交互“模式”；在处理完该对话框之前，你无法与主应用程序交互。

创建对话框时需要考虑一些小但重要的细节：

- 这些按钮应该怎么称呼？最好是描述性的，因此避免使用是/否或继续/取消，以重述关键动词。
- 您应该如何订购按钮？您是先取消（如 Mac），还是先继续（如 Windows）？您最好的选择是镜像您认为大多数人会使用的平台。
- 你能让危险的选择更明显吗？在这里，我习惯于 `class = "btn btn-danger"` 突出显示按钮的样式。

6.4.2 撤消操作

显式确认对于不经常执行的破坏性操作最有用。如果你想减少频繁操作所造成的错误，你应该避免它。在这种情况下，更好的方法是在实际执行操作之前等待几秒钟，让用户有机会注意到任何问题并撤消它们。

6.4.3 垃圾

对于几天后您可能会后悔的操作，更复杂的模式是在计算机上实现垃圾箱或回收站等功能。当您删除文件时，它不会被永久删除，而是会被移至保留单元，这需要单独的操作才能清空。这就像类固醇的“撤消”选项；你有很多时间为你的行为后悔。这也有点像确认；您必须执行两个单独的操作才能使删除永久化。

Chapter 7

上传和下载

与用户之间传输文件是应用程序的常见功能。您可以使用它上传数据进行分析，或将结果下载为数据集或报告。

7.1 上传

7.1.1 用户界面

支持文件上传所需的 UI 很简单：只需添加 fileInput() 到您的 UI 中即可。与大多数其他 UI 组件一样，只有两个必需参数： id 和 label。

7.1.2 服务器

服务器上的处理fileInput()比其他输入稍微复杂一些。大多数输入返回简单向量，但fileInput()返回包含四列的数据框：

- name：用户计算机上的原始文件名。
- size：文件大小，以字节为单位。默认情况下，用户只能上传最大 5 MB 的文件。您可以通过在启动 Shiny 之前设置 shiny.maxRequestSize 选项来增加此限制。例如，允许最多 10 MB 运行 `options(shiny.maxRequestSize = 10 * 1024^2)`。
- type：文件的“MIME 类型” 。这是文件类型的正式规范，通常源自扩展名，并且在 Shiny 应用程序中很少需要。
- datapath：数据在服务器上上传的路径。将此路径视为临时路径：如果用户上传更多文件，则该文件可能会被删除。数据始终保存到临时目录并指定临时名称。

7.1.3 上传数据

如果用户上传数据集，则需要注意两个细节：

1. 在页面加载时 `input$upload` 已初始化为 NULL，因此您需要 `req(input$upload)` 确保代码等待第一个文件上传。
2. 参数 accept 允许您限制可能的输入。最简单的方法是提供文件扩展名的字符向量，例如 accept = ".csv"。但该accept 参数只是对浏览器的建议，并不总是强制执行，因此最好自己验证它。在 R 中获取文件扩展名的最简单方法是 `tools::file_ext()`，只需注意它会删除扩展名中的前缀.。

7.2 下载

7.2.1 基础知识

同样，用户界面很简单：使用 downloadButton(id) 或 downloadLink(id) 为用户提供单击以下载文件的内容。

与其他输出不同，downloadButton() 它不与渲染函数配对。相反，您使用 downloadHandler()。

downloadHandler()有两个参数，两个都是函数：

- filename 应该是一个不带参数的函数，返回文件名（作为字符串）。该函数的作用是创建将在下载对话框中向用户显示的名称。
- content 应该是一个带有一个参数的函数，file 该参数是保存文件的路径。该函数的作用是将文件保存在 Shiny 知道的地方，以便它可以将其发送给用户。

这是一个不寻常的界面，但它允许 Shiny 控制文件的保存位置（以便可以将其放置在安全位置），同时您仍然可以控制该文件的内容。

7.2.2 下载报告

除了下载数据之外，您可能还希望应用程序的用户下载一份报告，该报告总结了 Shiny 应用程序中交互式探索的结果。这是相当大量的工作，因为您还需要以不同的格式显示相同的信息，但这对于高风险的应用程序非常有用。

生成此类报告的一种有效方法是使用参数化的 RMarkdown 文档。参数化的 RMarkdown 文件在 YAML 元数据中有一个 params 字段。

还有一些其他技巧值得了解：

- RMarkdown 在当前工作目录中工作，这在许多部署场景中都会失败（例如在 shinyapps.io 上）。您可以通过在应用程序启动时将报告复制到临时目录（即在服务器功能之外）来解决此问题
- 默认情况下，RMarkdown 将在当前进程中渲染报表，这意味着它将继承 Shiny 应用程序的许多设置（如加载的包、选项等）。为了提高鲁棒性，我建议使用 callr 包在单独的 R 会话中运行 render()

shinymeta 包解决了一个相关问题：有时您需要能够将 Shiny 应用程序的当前状态转换为可以在将来重新运行的可重现报告。

Chapter 8

动态 UI

创建动态用户界面有以下三种关键技术：

- 使用 update 函数族修改输入控件的参数。
- 用于 tabsetPanel() 有条件地显示和隐藏部分用户界面。
- 使用 uiOutput() 和 renderUI() 通过代码生成用户界面的选定部分。

这三个工具使你能够通过修改输入和输出来响应用户。我将演示一些更有用的方法来应用它们，但最终你只会受到你的创造力的限制。同时，这些工具可能会使你的应用程序更加难以推理，因此请谨慎部署它们，并始终努力使用最简单的技术来解决你的问题。

8.1 更新输入

每个输入控件都与一个**更新函数**配对，该函数允许你在创建控件后对其进行修改。更新函数看起来与其他 Shiny 函数略有不同：它们都将输入的名称（作为字符串）作为 inputId 参数。其余参数对应于输入构造函数的参数，可以在创建后进行修改。

8.1.1 分层选择框

一个特别重要的应用是通过逐步过滤，可以更轻松地从一长串可能的选项中进行选择。这通常是“分层选择框”的问题。更新功能的一个更复杂但特别有用的应用是允许跨多个类别进行交互式钻取。

8.1.2 循环引用

如果你想使用更新函数来更改输入的当前 value，那么我们需要讨论一个重要问题。从 Shiny 的角度来看，使用更新功能进行修改 value 与用户通过单击或键入来修改值没有什么不同。这意味着更新函数可以以与人类完全相同的方式触发反应性更新。这意味着你现在已经超出了纯反应式编程的范围，并且你需要开始担心循环引用和无限循环。

8.1.3 相互关联的输入

当应用程序中有多个“事实来源”时，很容易出现循环引用。

8.2 动态可见性

复杂性的下一步是有选择地显示和隐藏部分 UI。如果你了解一点 JavaScript 和 CSS，还有更复杂的方法，但有一种不需要任何额外知识的有用技术：使用选项卡集隐藏可选 UI。这是一个聪明的技巧，允许你根据需要显示和隐藏 UI，而无需从头开始重新生成它。

这里有两个主要想法：

- 使用带有隐藏选项卡的选项卡集面板。
- updateTabsetPanel() 用于从服务器切换选项卡。

8.2.1 条件性的用户界面

8.2.2 向导界面

你还可以使用这个想法来创建一个“向导”，这是一种界面，可以通过将大量信息分布在多个页面上来更轻松地收集信息。

8.3 使用代码创建 UI

- uiOutput() 在你的 ui 上插入一个占位符。这会留下一个“漏洞”，你的服务器代码可以稍后填充。
- renderUI() 在 server() 内部被调用以使用动态生成的 UI 填充占位符。

8.3.1 入门

你会注意到应用程序加载后只需要几分之一秒的时间即可显示。这是因为它是响应式的：应用程序必须加载、触发响应式事件，该事件调用服务器函数，生成要插入到页面中的 HTML。这是 renderUI() 的缺点之一；过度依赖它可能会导致用户界面滞后。

这种方法还有另一个问题：当你更改控件时，你会丢失当前选择的值。维护现有状态是使用代码创建 UI 的一大挑战。这就是有选择地显示和隐藏 UI 是一种更好的方法（如果它适合你）的原因之一 - 因为你没有销毁和重新创建控件，因此不需要执行任何操作来保留值。但是，在许多情况下，我们可以通过将新输入的 value 值设置为现有控件的当前值来解决问题。

8.3.2 多重控制

当你生成任意数量或类型的控件时，动态 UI 最有用。这意味着你将使用代码生成 UI，我建议使用函数式编程来完成此类任务。

8.3.3 对话框

在我们结束之前，想提一下相关的技术：对话框。你已经在显式确认中看到了它们，其中对话框的内容是固定的文本字符串。但因为 modalDialog() 是从服务器函数内部调用的，所以你实际上可以像 renderUI() 一样动态创建内容. 如果你想迫使用户在继续常规应用程序流程之前做出某些决定，那么这是一项非常有用的技术。

Chapter 9

9.1 基本思想

要使该应用程序可添加书签，我们需要做三件事：

- 添加 bookmarkButton() 到 UI。这会生成一个按钮，用户单击该按钮即可生成可添加书签的 URL。
- 转变ui成一个函数。您需要这样做，因为添加书签的应用程序必须重播添加书签的值：实际上，Shiny 修改了value每个输入控件的默认值。这意味着不再是单个静态 UI，而是依赖于 URL 中参数的多个可能的 UI；即它必须是一个函数。
- 添加 enableBookmarking = ”url” 到shinyApp()通话中。

9.1.1 更新 URL

另一种选择是自动更新浏览器中的 URL，而不是提供显式按钮。这允许您的用户在浏览器中使用用户书签命令，或从地址栏中复制并粘贴 URL。

自动更新 URL 需要服务器函数中的一些样板。

9.1.2 存储更丰富的状态

到目前为止，我们已经使用了 enableBookmarking = ”url” 直接将状态存储在 URL 中的方法。这是一个很好的起点，因为它非常简单并且适用于您可能部署 Shiny 应用程序的任何地方。然而，正如您可以想象的那样，如果您有大量输入，URL 将会变得很长，并且显然无法捕获上传的文件。

对于这些情况，您可能需要使用 enableBookmarking = ”server”，它将状态保存到 .rds 服务器上的文件中。这总是会生成一个短的、不透明的 URL，但需要在服务器上进行额外的存储。

服务器书签的主要缺点是它需要将文件保存在服务器上，并且这些文件需要保留多长时间并不明显。如果您为复杂状态添加书签并且从不删除这些文件，那么随着时间的推移，您的应用程序将占用越来越多的磁盘空间。如果您删除了这些文件，一些旧书签将停止工作。

9.2 书签挑战

自动书签依赖于反应图。它使用保存的值播种输入，然后重播所有反应式表达式和输出，只要您的应用程序的反应图很简单，这就会生成与您看到的相同的应用程序。本节简要介绍了一些需要额外注意的情况：

如果您的应用使用随机数，即使所有输入都相同，结果也可能不同。如果始终生成相同的数字确实很重要，那么您需要考虑如何使随机过程可重现。最简单的方法是使用 repeatable();

如果您有选项卡并且想要添加书签并恢复活动选项卡，请确保在您的调用中为 tabsetPanel() 提供 id；

如果存在不应添加书签的输入，例如它们包含不应共享的私人信息，请在服务器功能中某处包括对 setBookmarkExclude() 的调用。例如，setBookmarkExclude(c(”secret1”, ”secret2”)) 将确保 secret1 和 secret2 输入未添加书签。

如果您在自己的 reactiveValues() 对象中手动管理反应状态（正如我们将在逃离响应图中讨论的那样），您将需要使用 onBookmark() 和 onRestore() 回调来手动保存和加载附加状态。有关详细信息，请参阅高级书签。

Chapter 10

Tidy Evaluation

10.1 动机

env-variable An environment variable is a “programming” variable that you create with `<-`. `input$var` is an env-variable.

data-variable A data frame variable is a “statistical” variable that lives inside a data frame. carat is a data-variable.

有了这些新术语，我们可以使间接问题变得更加清晰：我们有一个数据变量(carat) 存储在环境变量(`input$var`) 中，并且我们需要某种方法来告诉 dplyr。有两种略有不同的方法可以执行此操作，具体取决于你正在使用的函数是“数据屏蔽”函数还是“整齐选择”函数。

10.2 数据屏蔽

数据屏蔽函数允许你使用“当前”数据框中的变量，而无需任何额外的语法。它在许多 dplyr 函数中使用，例如 arrange()、filter()、group_by()、mutate() 和 summarise()，以及 ggplot2 中的 aes()。数据屏蔽很有用，因为它允许你使用数据变量而无需任何附加语法。

在数据屏蔽函数内部，你可以使用 .data 或者 .env 如果你想明确你正在谈论的是数据变量还是环境变量。

```
diamonds %>% filter(.data$carat > .env$min)
```

10.3 整齐选择

除了数据屏蔽之外，整洁评估还有另一个重要部分：整洁选择。Tidy-selection 提供了一种按位置、名称或类型选择列的简洁方法。它用在 dplyr::select() 和d plyr::across() 以及 tidyr 中的许多函数中，例如 pivot_longer(), pivot_wider(), separate(), extract(), 和 unite()。

10.3.1 间接

要间接引用变量，请使用 any_of() 或则 all_of()：两者都期望包含数据变量名称的字符向量环境变量。唯一的区别是，如果你提供输入中不存在的变量名，会发生什么：all_of() 会抛出错误，而 any_of() 会默默地忽略它。

10.3.2 Tidy-Selection and Data-Masking

当你使用使用 tidy-selection 的函数时，使用多个变量非常简单：你只需将变量名称的字符向量传递到 any_of() 或 all_of() 即可。如果我们也能在数据屏蔽函数中做到这一点，不是很好吗？这就是 dplyr 1.0.0 中添加 across() 函数的想法。它允许你在数据屏蔽函数中使用整洁选择。

across() 通常与一个或两个参数一起使用。第一个参数选择变量，在 group_by() 或 distinct() 等函数中很有用。

第二个参数是应用于每个选定列的函数（或函数列表）。这使得它非常适合 mutate() 与 summarise() 当你希望以某种方式转换每个变量的情况。

10.4 parse() and eval()

这是一种很诱人的方法，因为它只需要学习很少的新想法。但它有一些主要缺点：因为您将字符串粘贴在一起，所以很容易意外创建无效代码，或者可能被滥用的代码来执行您不想要的操作。如果它是一个只有您使用的闪亮应用程序，那么这并不是非常重要，但这不是一个好习惯 - 否则很容易在您更广泛共享的应用程序中意外地创建安全漏洞。我们将在**??**中回顾这个想法。

（如果这是你能找到解决问题的唯一方法，你不应该感到难过，但是当你有更多的心理空间时，我建议花一些时间弄清楚如何在不进行字符串操作的情况下做到这一点。这将帮助您成为一名更好的 R 程序员。）

Part II

Mastering Reactivity

Chapter 11

为什么是响应性？

11.1 介绍

11.2 为什么我们需要响应式编程？

响应式编程是一种编程风格，重点关注随时间变化的值以及依赖于这些值的计算和操作。响应性对于 Shiny 应用程序来说很重要，因为它们是交互式的：用户更改输入控件（拖动滑块、输入文本框、检查复选框等），最终导致逻辑在服务器上运行（读取 CSV、子集数据、拟合模型等）导致输出更新（绘图重绘、表格更新……）。这与大多数 R 代码有很大不同，大多数 R 代码通常处理相当静态的数据。

为了让闪亮的应用程序发挥最大作用，我们需要响应式表达式和输出，当且仅当它们的输入发生变化时才进行更新。我们希望输出与输入保持同步，同时确保我们不会做不必要的工作。为了了解为什么响应性在这里如此有用，我们将尝试解决一个没有响应性的简单问题。

11.2.1 为什么不能使用变量？

从某种意义上说，你已经知道如何处理“随时间变化的值”：它们被称为“变量”。R 中的变量表示值，它们可以随着时间的推移而变化，但它们的设计初衷并不是为了在它们变化时为你提供帮助。变量可以随着时间的推移而改变，但它们永远不会自动改变。

11.2.2 函数呢？

函数的问题在于每次运行函数都要进行重复的计算。即使计算成本很低，不必要地重复它并不是什么大问题，但仍然没有必要：如果输入没有改变，为什么我们需要重新计算输出？

11.2.3 事件驱动编程

由于变量和函数都不起作用，我们需要创建一些新的东西。在过去的几十年里，我们会直接跳到事件驱动编程。事件驱动编程是一种非常简单的范例：你注册将响应事件而执行的回调函数。事件驱动编程解决了不必要的计算问题，但它产生了一个新问题：你必须仔细跟踪哪些输入影响哪些计算。不久之后，你就开始在正确性（只要有任何变化就更新所有内容）与性能（尝试仅更新必要的部分，并希望不会错过任何边缘情况）之间进行权衡，因为两者都很难做到。

11.2.4 响应式编程

在 Shiny 中，我们使用 reactiveVal() 创造了一个响应值。响应式值具有特殊语法来用于获取其值（像零参数函数一样调用它）和设置其值（通过像单参数函数一样调用它来设置其值）。

响应式表达式有两个重要的属性：

- 它很懒：在被调用之前它不会做任何工作。
- 它被缓存：它在第二次和后续调用时不会执行任何工作，因为它缓存了先前的结果。

我们将在响应图中回顾这些重要的属性。

11.3 响应式编程简史

如果你想了解有关其他语言的响应式编程的更多信息，了解一些历史可能会有所帮助。你可以在 40 多年前的第一个电子表格 VisiCalc 中看到响应式编程的起源：

> 我想象了一块神奇的黑板，如果你擦掉一个数字并在上面写下一个新的东西，所有其他数字都会自动改变，就像用数字进行文字处理一样。
> ——丹·布里克林

电子表格与响应式编程密切相关：你使用公式声明单元格之间的关系，当一个单元格发生更改时，其所有依赖项都会自动更新。所以你可能已经在不知不觉中完成了一堆响应式编程！

虽然响应性的想法已经存在很长时间了，但直到 20 世纪 90 年代末，学术计算机科学才开始认真研究它们。响应式编程的研究是由 FRAN 启动的，功能响应式动画是一种新颖的系统，用于将随时间的变化和用户输入合并到函数式编程语言中。这催生了丰富的文献，但对编程实践影响甚微。

直到 2010 年代，响应式编程才通过 JavaScript UI 框架的快节奏世界迅速进入编程主流。Knockout、Ember和Meteor（Joe Cheng 对 Shiny 的个人灵感）等开创性框架表明，响应式编程可以使 UI 编程变得更加容易。在短短几年内，响应式编程已经通过 React、Vue.js 和 Angular 等非常流行的框架主导了 Web 编程，这些框架要么本质上是响应式的，要么被设计为与响应式后端协同工作。

值得记住的是，“响应式编程”是一个相当笼统的术语。虽然所有响应式编程库、框架和语言都广泛关注编写响应不断变化的值的程序，但它们在术语、设计和实现方面存在巨大差异。在本书中，每当我们提到“响应式编程”时，我们特指的是在 Shiny 中实现的响应式编程。因此，如果你阅读的响应式编程材料并非专门针对 Shiny，那么这些概念甚至术语不太可能与编写 Shiny 应用程序相关。对于对其他响应式编程框架有一定经验的读者来说，Shiny 的方法类似于 Meteor 和 MobX ，但与 ReactiveX 系列或任何标榜自己为函数式响应式编程的东西有很大不同。

Chapter 12

响应图

12.1 响应式执行的逐步浏览

为了解释响应式执行的过程，我们将使用Figure 12.1所示的图形。它包含三个响应性输入、三个响应性表达式和三个输出。响应式输入和表达式统称为响应式生产者；响应式表达式和输出是响应式消费者。

组件之间的连接是有方向的，箭头指示响应的方向。这一方向可能会让你感到惊讶，因为很容易想到消费者依赖于一个或多个生产者。然而，很快你就会发现响应流在相反方向上的建模更加准确。

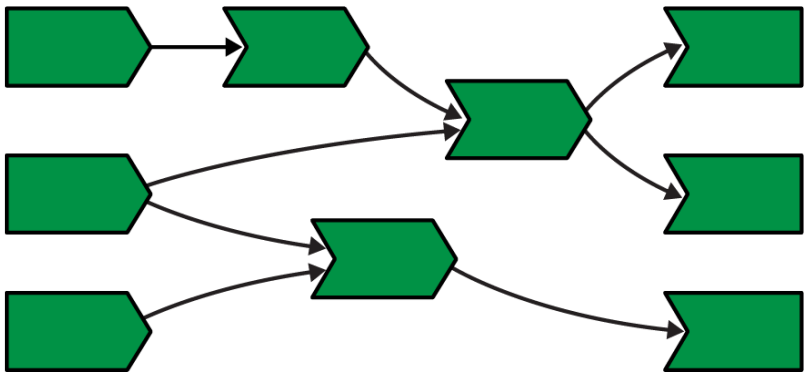

Figure 12.1: 一个虚构应用程序的完整响应图，包含三个输入、三个响应表达式和三个输出。

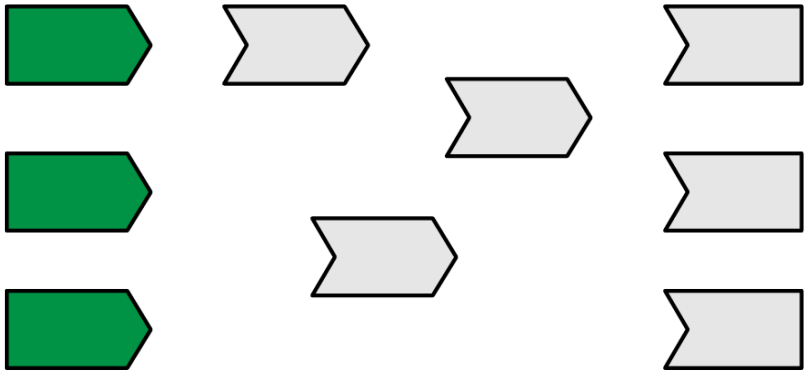

Figure 12.2: 应用程序加载后的初始状态。对象之间没有连接，所有响应式表达式都无效（灰色）。有六个响应性消费者和六个响应性生产者。

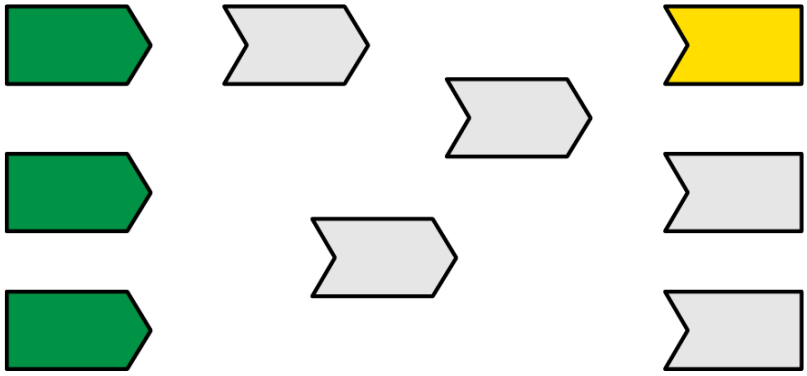

Figure 12.3: Shiny 开始执行任意观察者/输出，颜色为橙色。

12.2 会话开始

Figure 12.2显示了应用程序启动且服务器功能首次执行后的响应图。

该图中传达了三个重要信息：

- 元素之间没有联系，因为 Shiny对响应之间的关系没有先验知识。
- 所有响应式表达式和输出都处于起始状态，无效（灰色），这意味着它们尚未运行。
- 响应输入已准备就绪（绿色），表明它们的值可用于计算。

12.2.1 执行开始

现在我们开始执行阶段，如Figure 12.3所示。 在此阶段，Shiny 选择一个无效的输出并开始执行它（橙色）。你可能想知道 Shiny 如何决定执行哪些无效输出。简而言之，你应该表现

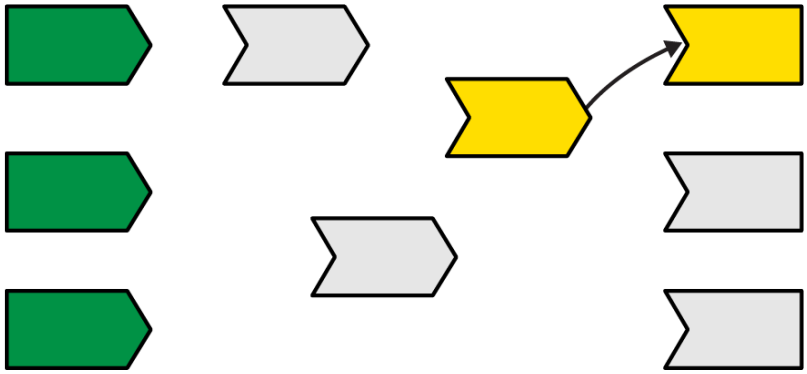

Figure 12.4: 输出需要响应式表达式的值，因此它开始执行该表达式。

得好像它是随机的：你的观察者和输出不应该关心它们执行的顺序，因为它们被设计为独立运行。

12.2.2 读取响应式表达式

执行输出可能需要来自响应性的值，如图14.4所示。 读取响应式会以两种方式改变图表：

- 响应式表达式还需要开始计算其值（变成橙色）。请注意，输出仍在计算：它正在等待响应式表达式返回其值，以便其自身的执行可以继续，就像 R 中的常规函数 调用一样。
- Shiny 记录了输出和响应表达式之间的关系（即我们画了一个箭头）。箭头的方向很重要：表达式记录了它被输出使用；输出不记录它使用该表达式。这是一个微妙的区别，但当你了解失效时，其重要性将变得更加清晰。

12.2.3 读取输入

这个特定的响应式表达式恰好读取响应式输入。再次建立了依赖/依赖关系，因此在图14.5中我们添加了另一个箭头。

与响应式表达式和输出不同，响应式输入不需要执行任何内容，因此它们可以立即返回。

12.2.4 响应式表达式完成

在我们的示例中，响应式表达式读取另一个响应式表达式，后者又读取另一个输入。我们将跳过这些步骤的详细描述，因为它们是我们已经描述过的内容的重复，并直接跳至Figure

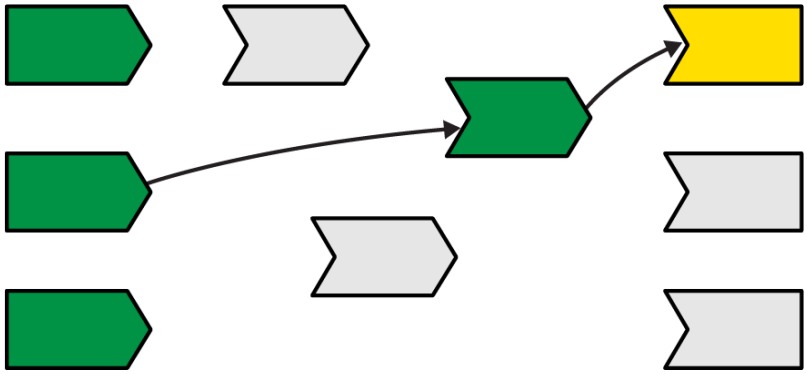

Figure 12.5: 响应表达式也读取响应值，因此我们添加另一个箭头。

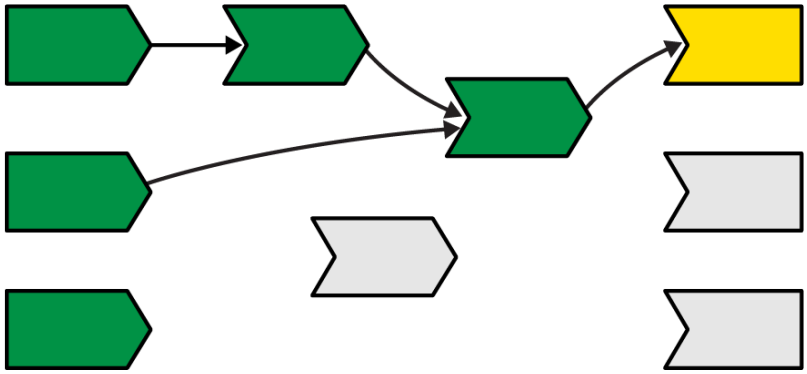

Figure 12.6: 响应式表达式已完成计算，因此变为绿色。

12.6。 现在响应式表达式已完成执行，它会变成绿色，表明它已准备就绪。它会缓存结果，因此不需要重新计算，除非其输入发生变化。

12.2.5 输出完成

现在响应式表达式已经返回了它的值，输出可以完成执行，并将颜色更改为绿色，如**??**所示。

12.2.6 执行下一个输出

12.2.7 执行完成，输出刷新

12.3 输入变化

上一步结束时，我们的 Shiny 会话处于完全空闲状态。现在想象一下应用程序的用户更改滑块的值。这会导致浏览器向服务器函数发送一条消息，指示 Shiny 更新相应的响应式输入。这将启动**无效阶段**，该阶段由三个部分组成：使输入无效、通知依赖项，然后删除现有连接。

12.3.1 使输入无效

12.3.2 通知依赖关系

现在，我们沿着之前绘制的箭头，将每个节点着色为灰色，并将我们遵循的箭头着色为浅灰色。得出**??**。

12.3.3 删除关系

接下来，每个无效的响应表达式和输出都会“擦除”所有进出它的箭头，产生**??**，并完成无效阶段。

从节点发出的箭头是一次性通知，将在下次值更改时触发。现在他们已经开火了，他们的目的已经达到，我们可以消灭他们了。

不太明显的是为什么我们删除进入无效节点的箭头，即使它们来自的节点没有无效。虽然这些箭头代表尚未触发的通知，但失效的节点不再关心它们：响应性消费者只关心通知，以便使自己失效，而这已经发生了。

我们如此看重这些关系，现在却把它们抛弃了，这似乎有些反常！但这是 Shiny 响应式编程模型的关键部分：尽管这些特定的箭头很重要，但它们现在已经过时了。确保我们的图表保持准确的唯一方法是在箭头变得陈旧时删除它们，并让 Shiny 在它们重新执行时重新发现这些节点周围的关系。我们将在动态性中回到这个重要的主题。

12.3.4 重新执行

12.4 动态性

在删除关系中，你了解到 Shiny“忘记”了它花费大量精力记录的响应组件之间的连接。这使得 Shiny 具有响应性动态，因为它可以在你的应用程序运行时发生变化。

12.5 总结

在本章中，已经准确了解了响应图的运作方式。特别是，第一次了解了失效阶段，该阶段不会立即导致重新计算，而是将响应式消费者标记为无效，以便在需要时重新计算它们。失效周期也很重要，因为它清除了以前发现的依赖关系，以便可以自动重新发现它们，从而使响应式图形变得动态。

Chapter 13

反应式构建块

13.1 响应值

有两种类型的响应值：

- 由 reactiveVal() 创建的单个响应值。
- 由 reactiveValues() 创建的响应值列表。

大多数 R 对象都具有 `copy_on_modify` 语义，这意味着如果您将相同的值分配给两个名称，则一旦您修改一个名称，连接就会中断。对于反应性值来说，情况并非如此——它们总是保留对同一值的引用，以便修改任何副本都会修改所有值。

13.2 隔离代码

为了结束本章，我将讨论两个重要的工具，用于精确控制反应图失效的方式和时间。在本节中，我将讨论 isolate()，该工具为 observeEvent() 和 eventReactive() 提供支持，并且可以让您避免在不需要时创建反应性依赖项。在下一节中，您将了解 invalidateLater()，它允许您按计划生成反应性失效。

13.2.1 isolate()

观察者通常与反应值结合在一起，以便跟踪状态随时间的变化。

13.2.2 observeEvent() 和 eventReactive()

observeEvent() 和 eventReactive() 具有允许您控制其操作细节的附加参数：

1. 默认情况下，这两个函数都会忽略产生的任何事件NULL（或者在操作按钮的特殊情况下，忽略 0）。使用 ignoreNULL = FALSE 来处理 NULL 值。
2. 默认情况下，这两个函数在您创建它们时都会运行一次。用 ignoreInit = TRUE 跳过此运行。
3. 仅对 observeEvent()，您可以设置 once = TRUE 来仅处理程序一次。

13.3 定时失效

isolate() 减少反应图失效的时间。本节的主题 invalidateLater() 做了相反的事情：它允许您在没有数据更改时使反应图无效。

13.3.1 轮询

一个有用的 invalidateLater() 应用是将 Shiny 连接到 R 外部正在更改的数据。

Chapter 14

逃离响应图

14.1

Part III

最佳实践

Chapter 15

一般准则

Chapter 16

Shiny 模块

16.1 模块基础知识

模块与应用程序非常相似。就像应用程序一样，它由两部分组成：

- 生成规范模块 UI 的函数 ui。
- 在函数内部运行代码的模块服务器函数 server。

这两个函数都有标准形式。它们都接受一个 id 参数并使用它来命名模块。要创建模块，我们需要从应用程序 UI 和服务器中提取代码并将其放入模块 UI 和服务器中。

Chapter 17

测试

测试首先声明意图（as.vector() strips names），然后使用常规 R 代码生成一些测试数据。然后使用期望（以 expect_ 开头的函数）将测试数据与预期结果进行比较。第一个参数是要运行的一些代码，第二个参数描述了预期结果。

17.1 测试函数

17.1.1 基本结构

测试分为三个级别：

文件 所有测试文件都位于 tests/testthat 中，并且每个测试文件应对应于 R/ 中的一个代码文件，例如 R/module.R 中的代码应由 tests/testthat/test-module.R 中的代码进行测试。幸运的是，你不必记住该约定：只需用 usethis::use_test() 自动创建或定位与当前打开的 R 文件相对应的测试文件。

测试 每个文件都被分解为测试，即对 test_that() 的调用。测试通常应该检查函数的单个属性。很难准确描述这意味着什么，但一个很好的启发是，你可以轻松地在 test_that() 的第一个参数中描述测试。

期待 每个测试都包含一个或多个期望，其函数以 expect_。它们准确地定义了你期望代码执行的操作，无论是返回特定值、引发错误还是其他操作。在本章中，我将讨论对 Shiny 应用程序最重要的期望，但你可以在 testthat 网站上查看完整列表。

测试的艺术在于弄清楚如何编写测试来明确定义函数的预期行为，而不依赖于将来可能改变的偶然细节。

17.1.2 主要期望

在测试函数时，你会经常使用两个期望：expect_equal() 和 expect_error()。与所有期望函数一样，第一个参数是要检查的代码，第二个参数是预期结果：在 expect_equal() 的情况下为预期值，在 expect_error() 的情况下为预期错误文本。

使用 expect_equal() 时请记住，你不必测试整个对象：通常最好只测试你感兴趣的组件。

对一些 expect_equal() 的特殊情况，可以节省你的打字时间：

- expect_true(x) 和 expect_false(x) 相当于 expect_equal(x, TRUE) 和 expect_equal(x, FALSE)。expect_null(x) 相当于 expect_equal(x, NULL)。
- expect_named(x, c("a", "b", "c")) 相当于 expect_equal(names(x), c("a", "b", "c"))，但有选项 ignore.order 和 ignore.case。expect_length(x, 10) 相当于 expect_equal(length(x), 1

还有一些函数可以实现针对向量 expect_equal() 的宽松版本：

- expect_setequal(x, y) 测试 x 中的每个值是否出现在 y 中，以及 y 中的每个值是否出现在 x 中。
- expect_mapequal(x, y) 测试 x 和 y 具有相同的名称并且 x[names(y)] == y。

测试代码是否生成错误通常很重要，你可以使用 expect_error()。请注意，expect_error() 的第二个参数是正则表达式 - 目标是找到与你期望的错误匹配的一小段文本，并且不太可能与你不期望的错误匹配。

或者使用 expect_snapshot()，我们将很快讨论。expect_error() 还带有变体 expect_warning()，expect_message() 用于以与错误相同的方式测试警告和消息。这些对于测试 Shiny 应用程序很少需要，但对于测试包非常有用。

17.1.3 用户界面函数

你可以使用相同的基本思想来测试从 UI 代码中提取的功能。但这些需要一个新的期望，因为手动输入所有 HTML 会很乏味，所以我们使用快照测试。快照期望与其他期望的主要不同之处在于，期望结果存储在单独的快照文件中，而不是存储在代码本身中。当你设计复杂的用户界面设计系统时，快照测试最有用，这超出了大多数应用程序的范围。

快照测试的关键思想是将预期结果存储在单独的文件中：这样可以将大量数据排除在测试代码之外，并且意味着你无需担心在字符串中转义特殊值。

17.2 工作流程

17.2.1 代码覆盖率

验证你的测试是否测试了你认为他们正在测试的内容非常有用。一个很好的方法是使用“代码覆盖率”来运行测试并跟踪运行的每一行代码。然后，你可以查看结果，看看哪些代码行从未被测试触及，并让你有机会反思是否测试了代码中最重要、风险最高或最难编程的部分。它不能替代对代码的思考——你可能拥有 100% 的测试覆盖率，但仍然存在错误。但它是一个有趣且有用的工具，可以帮助你思考什么是重要的，特别是当你有复杂的嵌套代码时。

17.3 测试响应性

17.3.1 限制

testServer() 是你的应用程序的模拟。模拟很有用，因为它可以让你快速测试响应式代码，但它并不完整。

- 与现实世界不同，时间不会自动前进。因此，如果你想测试依赖于 reactiveTimer() 或 invalidateLater() 的代码，则需要通过调用 `session$elapse(millis = 300)` 来手动提前时间。
- testServer() 忽略用户界面。这意味着输入不会获得默认值，并且 JavaScript 无法工作。最重要的是，这意味着你无法测试这些 update* 函数，因为它们通过将 JavaScript 发送到浏览器来模拟用户交互来工作。

17.4 测试 JavaScript

Chapter 18

性能

18.1 基准

通过基准测试，您可以检查多个用户的应用程序的性能，而无需实际让真实的人接触可能很慢的应用程序。或者，如果您想为数百或数千个用户提供服务，基准测试将帮助您确定每个进程可以处理多少用户，从而确定您需要使用多少台服务器。

基准测试过程由 shinyloadtest 包支持，并具有三个基本步骤：

1. 使用 `shinyloadtest::record_session()` 录制模拟典型用户的脚本。
2. 使用 shinycannon 命令行工具与多个并发用户一起重播该脚本。
3. 使用 `shinyloadtest::report()` 分析结果。

18.2 缓存

缓存是一种非常强大的技术，可以提高代码性能。基本思想是记录每次调用函数的输入和输出。当使用一组已经看到的输入调用缓存函数时，它可以重播记录的输出而无需重新计算。memoise 等软件包提供了用于缓存常规 R 函数的工具。

18.2.1 基础知识

bindCache() 很容易使用。只需将要缓存的 reactive() 或 render* 函数通过管道传递到：bindCache()。

18.2.2 缓存反应式

使用缓存的一个常见地方是与 Web API 结合使用——即使 API 非常快，您仍然必须发送请求，等待服务器响应，然后解析结果。因此，缓存 API 结果通常会带来很大的性能提升。

18.2.3 缓存范围

默认情况下，绘图缓存存储在内存中，永远不会大于 200 MB，在单个进程的所有用户之间共享，并且在应用程序重新启动时丢失。您可以为单个反应或整个会话更改此默认值：

- `bindCache(…, cache = "session")` 将为每个用户会话使用单独的缓存。这确保了私有数据不会在用户之间共享，但它也降低了缓存的好处。
- 使用 `shinyOptions(cache = cachem::cache_mem())` 或 `shinyOptions(cache = cachem::cache_disk` 更改整个应用程序的默认缓存。您可以使用缓存在多个进程之间共享，并在应用程序重新启动时持续存在。

18.3 其他优化

许多应用程序中还出现了另外两种优化：按计划执行数据导入和操作，以及仔细管理用户期望。