Mastering reactivity

Introduction

您现在拥有一整套有用的技术，让您能够创建各种有用的应用程序。接下来，我们将把注意力转向作为Shiny魔力基础的反应性理论：

- 在第13章中，您将了解为什么需要反应式编程模型，以及一些关于R之外的反应式编程的历史。
- 在第14章中，您将了解反应图的全部细节，该图确切决定了反应成分的更新时间。
- 在第15章中，您将了解底层构建块，特别是观察器（observers）和定时失效（timed invalidation）。
- 在第16章中，您将学习如何使用`reactiveVal()`和`observe()`来逃避反应图的约束。

对于Shiny应用程序的常规开发，您当然不需要了解所有这些细节。但提高您的理解力将帮助您从一开始就编写正确的应用程序，当某些事情出现意外时，您可以更快地缩小潜在问题的范围。

Chapter 13. Why reactivity?

13.1 Introduction

Shiny的最初印象通常是“魔法”。当你开始时，魔术很棒，因为你可以非常快速地制作简单的应用程序。但软件中的魔力通常会导致幻灭：如果没有一个坚实的模型基础，当你冒险超越其演示和示例的边界时，很难预测软件将如何发挥作用。当事情没有按照你预期的方式进行时，调试几乎是不可能的。

幸运的是，Shiny是“好”的魔法。正如Tom Dale在他的Ember.js JavaScript框架时所说：“我们做了很多魔法，但它是*很好的魔法*，这意味着它会分解成合理的原语。”这是Shiny团队渴望的Shiny的质量，特别是在反应式编程方面。当你一层层深入剖析反应式编程时，你不会找到一堆启发式、特殊情况和黑客式的代码；相反，你会发现一个有“魔法”，但最终又相当直接的编程机制。一旦你形成了一个准确的反应式编程模型，你会看到Shiny的底层里什么都没有：魔力来自以一致的方式组合的简单概念。

在本章中，我们将引入反应式编程的深层次概念，然后简要介绍与Shiny相关的反应性历史。

13.2 Why do we need reactive programming?

反应式编程是一种编程风格，专注于随时间变化的值，以及依赖于这些值的计算和操作。反应性对shiny应用程序很重要，因为它们是交互式的：用户更改输入控件（拖动滑块，键入文本框，检查复选框，……），引起code在服务器上运行（读取CSV，子设置数据，拟合模型，……），最终输出更新（绘图重新绘制，表格更新，……）。这与大多数R代码截然不同，后者通常处理相当静态的数据。

为了让Shiny应用程序具有最大的用处，我们需要在输入发生变化时更新反应式表达式和输出。我们希望输出与输入保持同步，同时确保我们永远不会做超过必要的工作。为了了解为什么反应性在这里如此有帮助，我们将尝试解决一个没有反应性的简单问题。

13.2.1 Why can't you use variables?

从某种意义上说，你已经知道如何处理“随着时间的推移而变化的值”：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，但它们不是为了在变化时帮助你。举这个将温度从摄氏度转换为华氏度的简单例子：

```
temp_c <- 10
temp_f <- (temp_c * 9 / 5) + 32
temp_f
#> [1] 50
```

到目前为止还不错：`temp_c`变量的值为10，`temp_f`变量的值为50，我们可以更改`temp_c`：

```
temp_c <- 30
```

但更改`temp_c`不会影响`temp_f`：

```
temp_f
#> [1] 50
```

变量可以随着时间的推移而变化，但它们永远不会自动改变。

13.2.2 What about functions?

相反，您可以用一个函数来解决这个问题：

```
temp_c <- 10
temp_f <- function() {
  message("Converting")
  (temp_c * 9 / 5) + 32
}
temp_f()
#> Converting
#> [1] 50
```

（这是一个有点奇怪的函数，因为它没有任何参数，而是从其封闭环境访问`temp_c`，但它是完全有效的R代码。）

这解决了reactivity试图解决的第一个问题：每当您访问`temp_f()`时，您都会获得最新的计算：

```
temp_c <- -3
temp_f()
#> Converting
#> [1] 26.6
```

然而，它并没有最小化计算。每次您调用`temp_f()`，它都会重新计算，即使`temp_c`改变：

```
temp_f()
#> Converting
#> [1] 26.6
```

在这个琐碎的例子中，计算资源消耗很少，所以不必要地重复它，没什么大不了的，但仍然没有必要：如果输入没有改变，我们为什么需要重新计算输出？

13.2.3 Event-driven programming

由于变量和函数都不起作用，我们需要创造一些新的东西。在过去的几十年里，我们会直接跳到*事件驱动的编程*。事件驱动编程是一个有吸引力的简单范式：您注册回调函数，这些函数将响应事件执行。

我们可以使用R6实现一个非常简单的事件驱动工具包，如下例所示。在这里，我们定义了一个具有三种重要方法的`DynamicValue`：`get()`和`set()`访问和更改基础值，以及`onUpdate()`注册代码以在值修改时运行。如果您不熟悉R6，不要担心细节，而是专注于以下示例。

```
DynamicValue <- R6::R6Class("DynamicValue", list(
  value = NULL,
  on_update = NULL,

  get = function() self$value,

  set = function(value) {
    self$value <- value
    if (!is.null(self$on_update))
      self$on_update(value)
    invisible(self)
  },

  onUpdate = function(on_update) {
    self$on_update <- on_update
    invisible(self)
  }
))
```

因此，如果Shiny是在五年前发明的，它可能看起来更像这样，其中`temp_c`在需要时使用`<<-`更新`temp_f`。

```
temp_c <- DynamicValue$new()
temp_c$onUpdate(function(value) {
  message("Converting")
  temp_f <<- (value * 9 / 5) + 32
})

temp_c$set(10)
#> Converting
temp_f
#> [1] 50

temp_c$set(-3)
#> Converting
temp_f
#> [1] 26.6
```

事件驱动编程解决了不必要的计算问题，但它创造了一个新问题：您必须仔细跟踪哪些输入影响哪些计算。不久之后，您开始权衡正确性（只要有任何变化，只需更新所有内容）与性能（尝试只更新必要的部分，并希望您没有错过任何边缘情况），因为两者都很难做到。

13.2.4 Reactive programming

反应式编程通过结合上述解决方案的功能，优雅地解决了这两个问题。现在，我们可以向您展示一些真正的shiny代码，使用特殊的shiny模式，`reactiveConsole(TRUE)`可以直接在控制台中尝试反应性。

```
library(shiny)
reactiveConsole(TRUE)
```

与事件驱动编程一样，我们需要某种方法来表明我们有一个特殊类型的变量。在Shiny中，我们用`reactiveVal()`创建一个**反应值**。反应值具有特殊的语法，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）。

```
temp_c <- reactiveVal(10) # create
temp_c()                  # get
#> [1] 10
temp_c(20)                # set
temp_c()                  # get
#> [1] 20
```

现在，我们可以创建一个依赖于此值的反应式表达式：

```
temp_f <- reactive({
  message("Converting")
  (temp_c() * 9 / 5) + 32
})
temp_f()
#> Converting
#> [1] 68
```

正如您在创建应用程序时所学到的，反应式表达式会自动跟踪其所有依赖项。因此，稍后，如果`temp_c`更改，`temp_f`将自动更新：

```
temp_c(-3)
temp_c(-10)
temp_f()
#> Converting
#> [1] 14
```

但是，如果`temp_c()`没有更改，那么`temp_f()`不需要重新计算，只需从缓存中检索即可：

```
temp_f()
#> [1] 14
```

反应表达式有两个重要属性：

- 它很**懒惰**：在被召唤之前，它不会做任何工作。
- 它被**缓存**：它在第二次和之后的调用中不做任何工作，因为它缓存了之前的结果。

我们将在第14章中讲到这些重要属性。

13.3 A brief history of reactive programming

如果您想了解更多关于其他语言的响应式编程，一点历史可能会有所帮助。您可以在第一个电子表格VisiCalc中看到40多年前反应式编程的起源：

> 我想象了一个神奇的黑板，如果你擦除一个数字并写一个新东西，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

电子表格与反应式编程密切相关：您使用公式声明单元格之间的关系，当一个单元格更改时，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反应式编程！

虽然反应性的想法已经存在了很长时间，但直到20世纪90年代末，它们才在学术计算机科学中得到了认真的研究。反应式编程的研究由FRAN，函数式反应动画启动（functional reactive animation），这是一个将随着时间的推移变化和用户输入纳入函数式编程语言的新型系统。这催生了丰富的文献，但对编程实践影响不大。

直到2010年代，反应式编程才通过快节奏的JavaScript UI框架世界进入编程主流。Knockout、Ember和Meteor（Joe Cheng对Shiny的个人灵感）等开创性框架表明，反应式编程可以使UI编程大大简化。在短短几年内，反应式编程通过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反应性的，要么旨在与反应式后端携手并进。

值得记住的是，“反应式编程”是一个相当通用的术语。虽然所有反应式编程库、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、设计和实现方面差异很大。在这本书中，每当我们提到“反应式编程”时，我们都会特别指在Shiny中实现的反应式编程。因此，如果您阅读了不专门针对Shiny的关于反应式编程的材料，那么这些概念甚至术语不太可能与编写Shiny应用程序有关。对于确实对其他反应式编程框架有一定经验的读者来说，Shiny的方法与Meteor和MobX相似，与ReactiveX系列或任何自称为功能反应式编程的东西非常不同。

13.4 Summary

既然您已经了解了为什么需要反应式编程，并了解了一点历史，下一章将讨论基础理论的更多细节。最重要的是，您将巩固对反应图的理解，该图连接反应值、反应式表达式和观察器（observer），并准确控制何时运行。

Chapter 14. The reactive graph

14.1 Introduction

要理解反应式计算，您必须首先理解反应性图。在本章中，我们将深入研究图表的细节，更加关注事情发生的确切顺序。特别是，您将了解无效的重要性，这个过程是确保Shiny完成最低限度工作的关键。您还将了解reactlog软件包，该软件包可以自动为真实应用程序绘制反应图。

如果你已经有一段时间没有看第3章了，我强烈建议你在继续之前重新熟悉它。它为我们将在这里更详细地探索的概念奠定了基础。

14.2 A step-by-step tour of reactive execution

为了解释反应性计算执行的过程，我们将使用图14.1所示的图形。它包含三个反应式输入、三个反应式表达式和三个输出。回想一下，反应式输入和表达式统称为反应式生产者；反应式表达式和输出是反应式消费者。

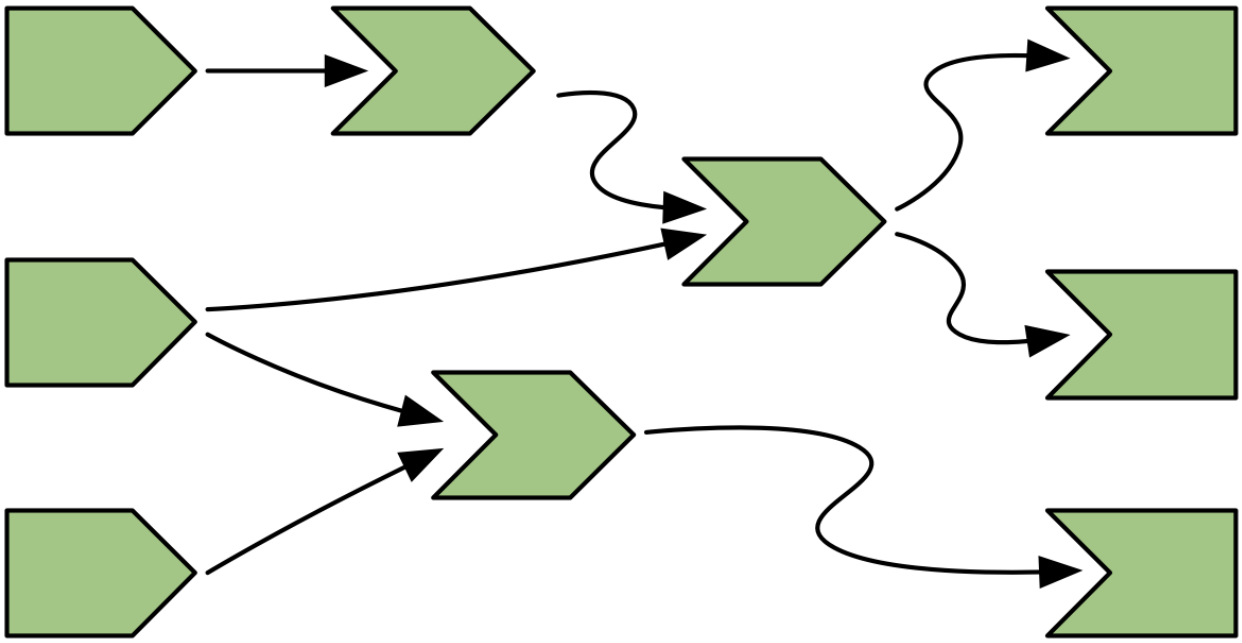

图14.1：包含三个输入、三个达式和三个输出的假想应用程序的完整反应图。

组件之间的连接是定向的，箭头指示反应方向。方向可能会让你感到惊讶，因为很容易想到消费者依赖一个或多个生产商。然而，不久，您将看到反应性流更准确地向相反的方向建模。

底层应用程序并不重要，但如果它能帮助你拥有一些具体的东西，你可以假装它来自这个不太有用的应用程序。

```
ui <- fluidPage(
  numericInput("a", "a", value = 10),
  numericInput("b", "b", value = 1),
  numericInput("c", "c", value = 1),
  plotOutput("x"),
  tableOutput("y"),
  textOutput("z")
)

server <- function(input, output, session) {
  rng <- reactive(input$a * 2)
  smp <- reactive(sample(rng(), input$b, replace = TRUE))
  bc <- reactive(input$b * input$c)

  output$x <- renderPlot(hist(smp()))
  output$y <- renderTable(max(smp()))
  output$z <- renderText(bc())
}
```

我们开始吧！

14.3 A session begins

图14.2显示了应用程序启动和服务器功能首次执行后的反应图。

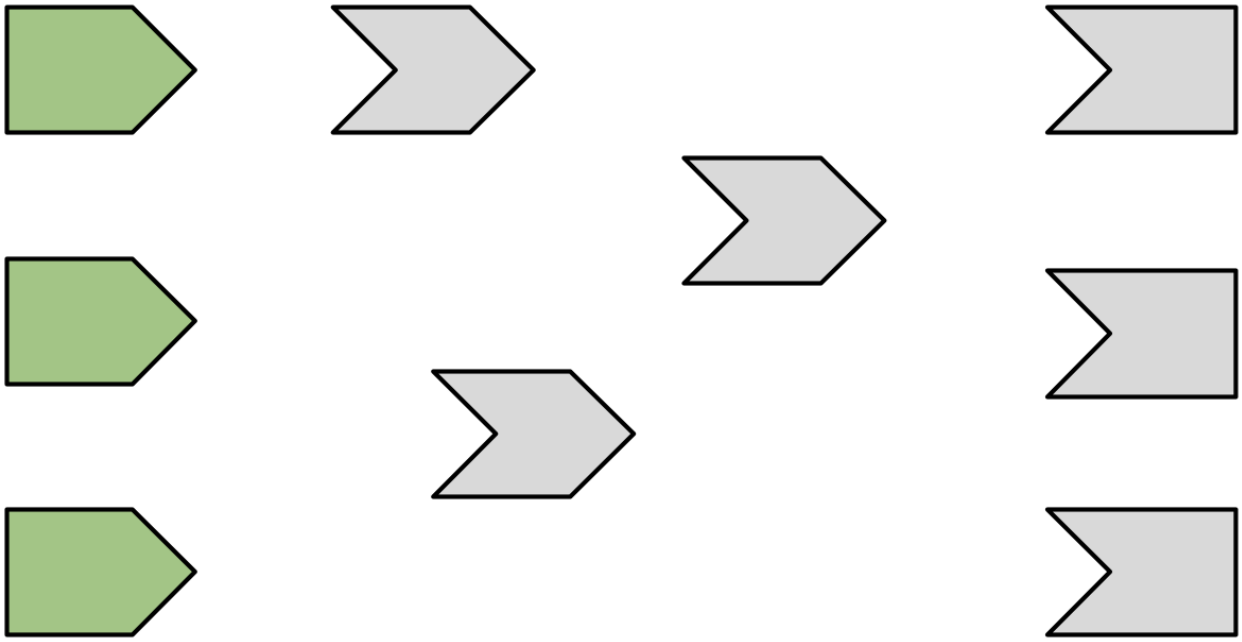

图14.2：应用程序加载后的初始状态。对象之间没有连接，所有反应式表达式都无效（灰色）。有六个反应式消费者和六个反应式生产者。

这张图中有三个重要信息：

- 元素之间没有联系，因为Shiny对反应者之间的关系没有*先验*的了解。
- 所有反应式表达式和输出都处于启动状态（**无效**（灰色）），这意味着它们尚未运行。
- 反应式输入已准备就绪（绿色），表示其值可用于计算。

14.3.1 Execution begins

现在我们开始执行阶段，如图14.3所示。

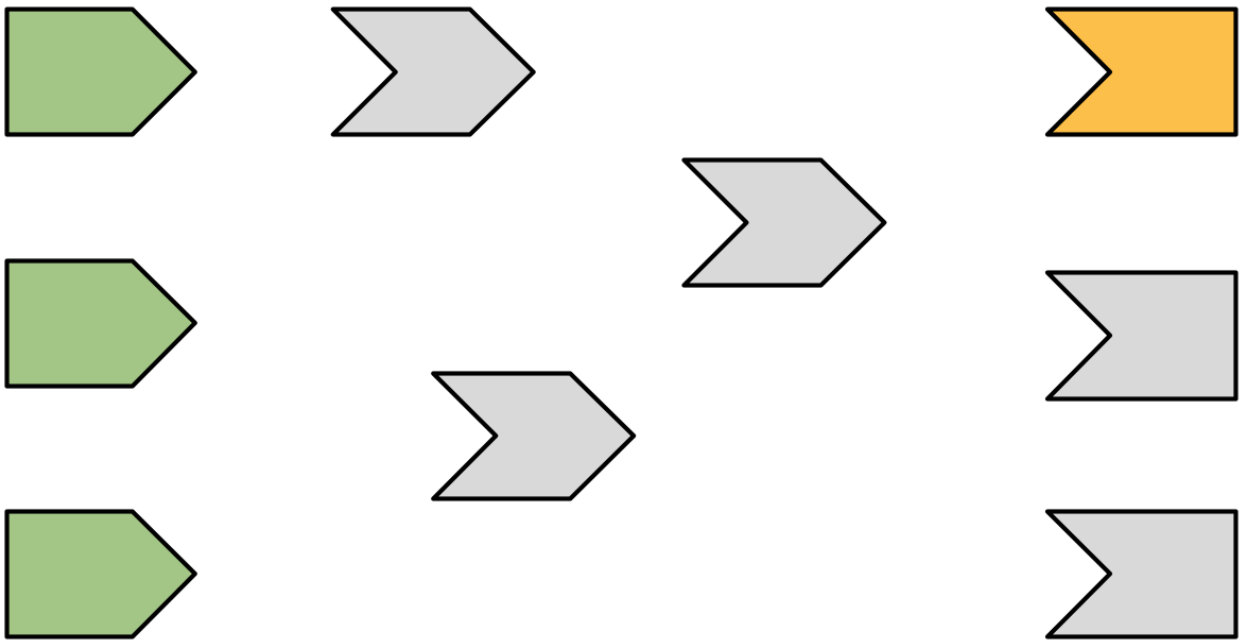

图14.3：Shiny开始执行任意观察器/输出（橙色）。

在这个阶段，Shiny选择一个无效的输出并开始执行它（橙色）。您可能想知道Shiny如何决定执行哪些无效输出。简而言之，你应该表现得好像它是随机的：你的观察器（observer）和输出不应该关心它们以什么顺序执行，因为它们被设计为独立运行47。

14.3.2 Reading a reactive expression

执行输出可能需要一个来自反应的值，如图14.4所示。

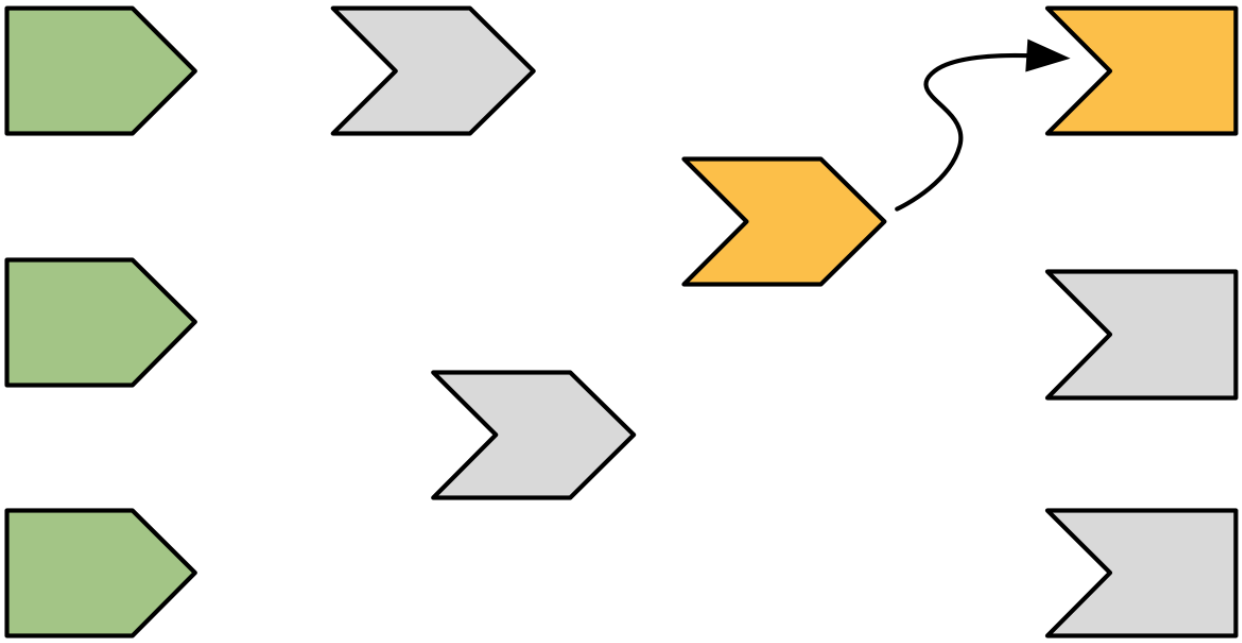

图14.4：输出需要反应式表达式的值，因此它开始执行表达式。

阅读反应性图，以两种方式改变图表：

- 反应式表达式也需要开始计算其值（变成橙色）。请注意，输出仍在计算中：它正在等待反应表式达式返回其值，以便其自己的执行可以继续，就像R中的常规函数调用一样。
- Shiny记录了输出和反应表达式之间的关系（即我们画一个箭头）。箭头的方向很重要：反应式表达式记录输出使用它；输出不记录它使用表达式。这是一个微妙的区别，但当你了解到无效时，它的重要性会变得更加清晰。

14.3.3 Reading an input

这个特殊的反应表式达式碰巧读取一个反应式输入。同样，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。

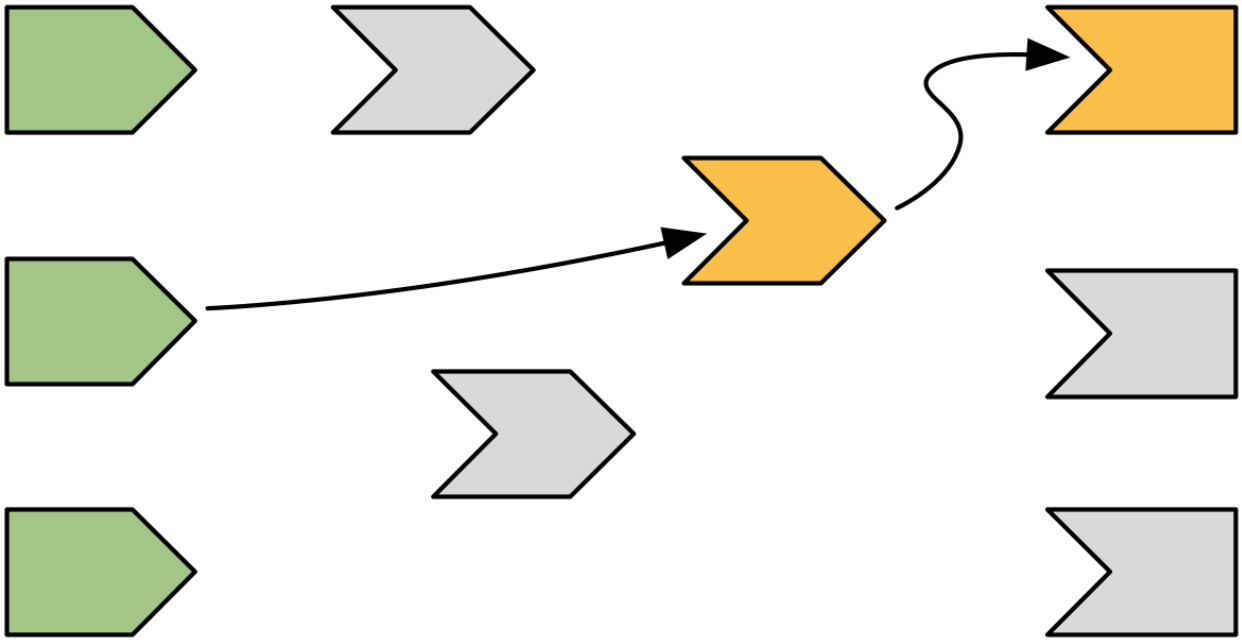

图14.5：反应式表达式也从反应值读取，因此我们添加另一个箭头。

与反应式表达式和输出不同，反应式输入没有要执行的内容，因此它们可以立即返回。

14.3.4 Reactive expression completes

在我们的示例中，反应式表达式读取另一个反应式表达式，而反应式表达式又读取另一个输入。我们将跳过这些步骤的逐一描述，因为它们重复了我们已经描述的内容，并直接跳转到图14.6。

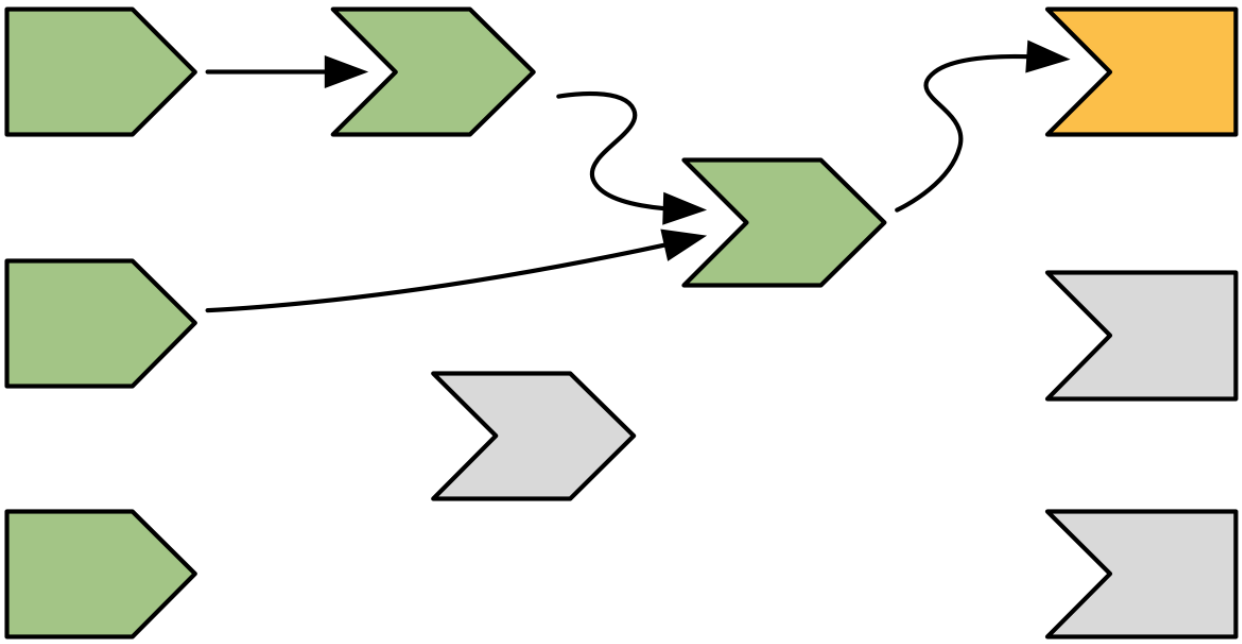

图14.6：反应式表达式已完成计算，因此变为绿色。

现在，反应式表达式已完成执行，它会变成绿色，以表示它已准备就绪。它缓存结果，因此除非其输入发生变化，否则不需要重新计算。

14.3.5 Output completes

现在，反应表达式已返回其值，输出可以完成执行，并将颜色更改为绿色，如图14.7所示。

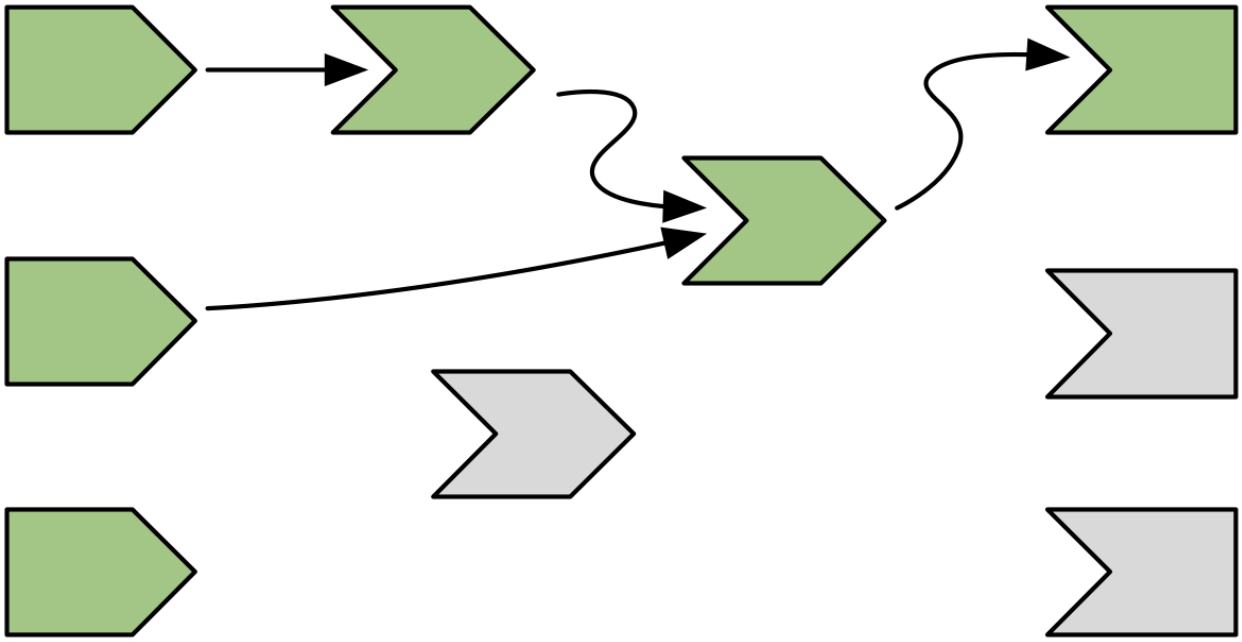

图14.7：输出已完成计算并变成绿色。

14.3.6 The next output executes

现在第一个输出已经完成，Shiny选择另一个输出来执行。此输出变为橙色，图14.8，并开始从反应式生产者读取值。

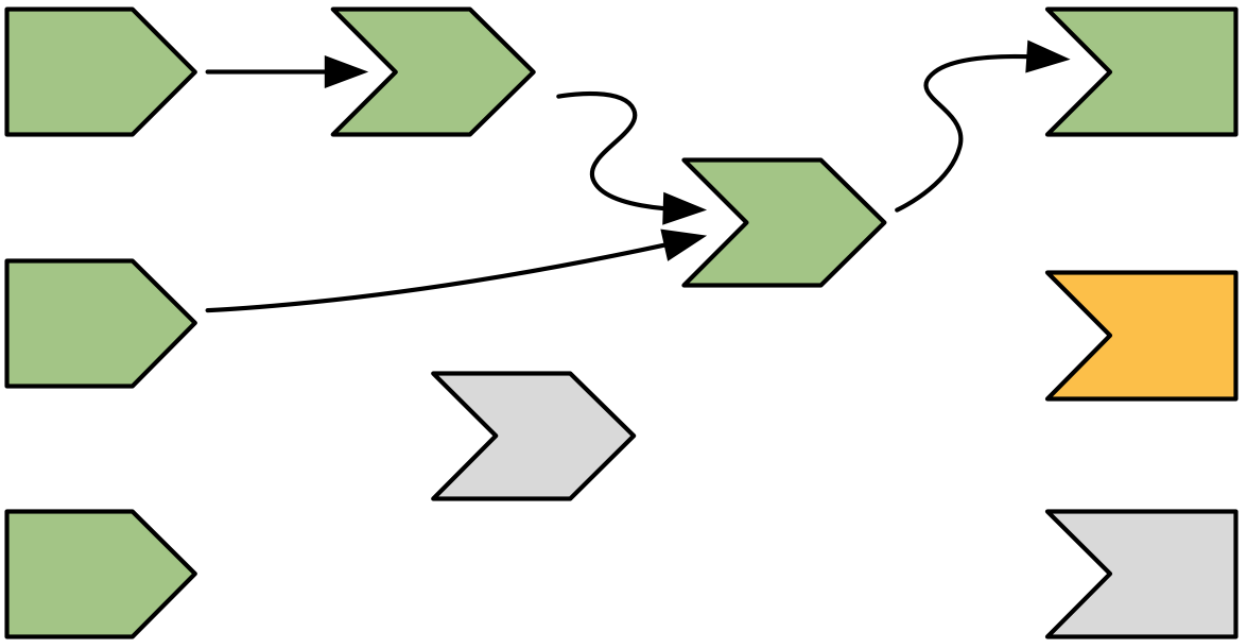

图14.8：下一个输出开始计算，变成橙色。

完整的反应可以立即返回其值；无效的反应将启动自己的执行图。这个循环将重复，直到每个无效的输出进入完整（绿色）状态。

14.3.7 Execution completes, outputs flushed

现在，所有输出都已完成执行并处于空闲状态，图14.9。

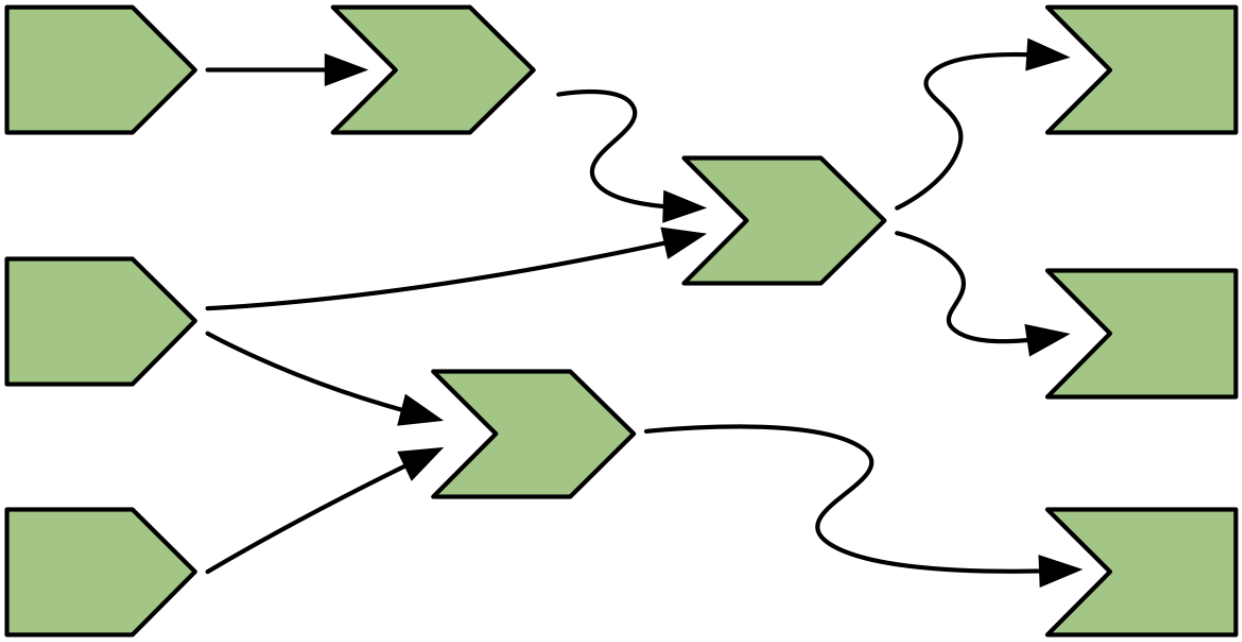

图14.9：所有输出和被动表达式都已完成并变成绿色。

这一轮反应执行已经完成，在一些外部力量作用在系统上（例如，shiny应用程序的用户在用户界面中移动滑块）之前，不会进行更多工作。用反应性的话来说，本次会议现在处于静止。

让我们在这里停一会儿，想想我们做了什么。我们读取了一些输入，计算了一些值，并生成了一些输出。但更重要的是，我们还发现了反应性控件之间的关系。当反应式输入发生变化时，我们确切地知道我们需要更新哪些反应。

14.4 An input changes

上一步中断了，我们的shiny会话处于完全空闲状态。现在想象一下，应用程序的用户会更改滑块的值。这导致浏览器向服务器函数发送消息，指示Shiny更新相应的反应式输入。这启动了一个**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。

14.4.1 Invalidating the inputs

失效阶段从更改的输入/值开始，我们将用灰色填充，这是我们通常的失效颜色，如图14.10所示。

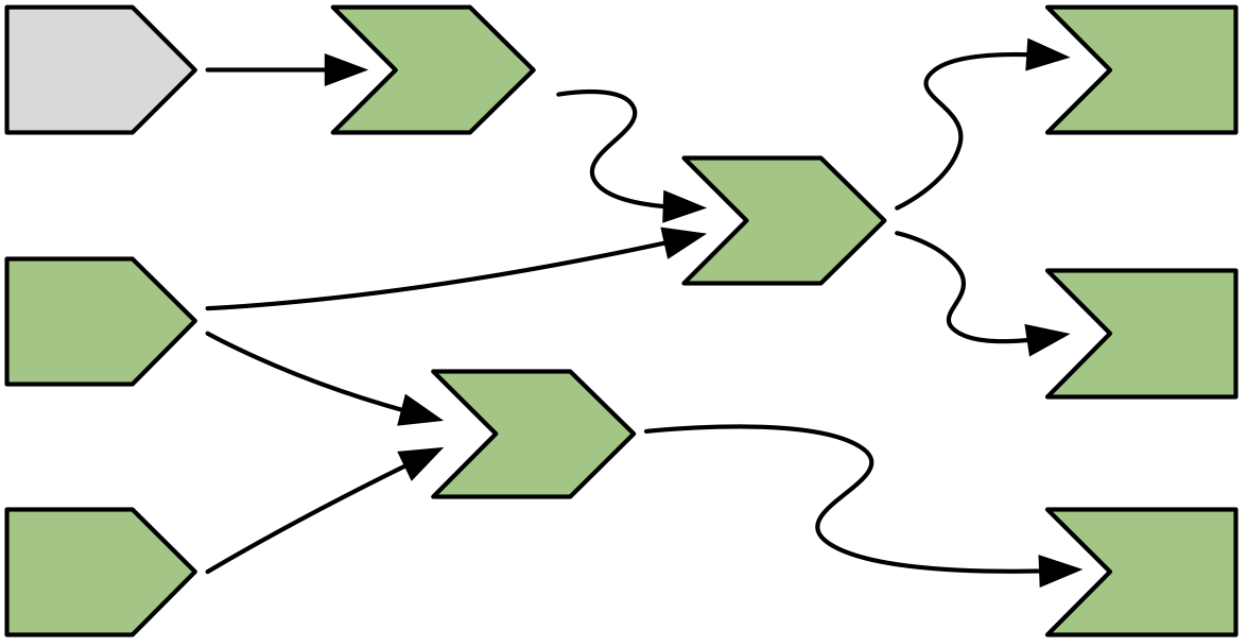

图14.10：用户与应用程序交互，使输入无效。

14.4.2 Notifying dependencies

现在，我们按照之前绘制的箭头，将每个节点涂成灰色，并将箭头涂成浅灰色。这产生了图14.11。

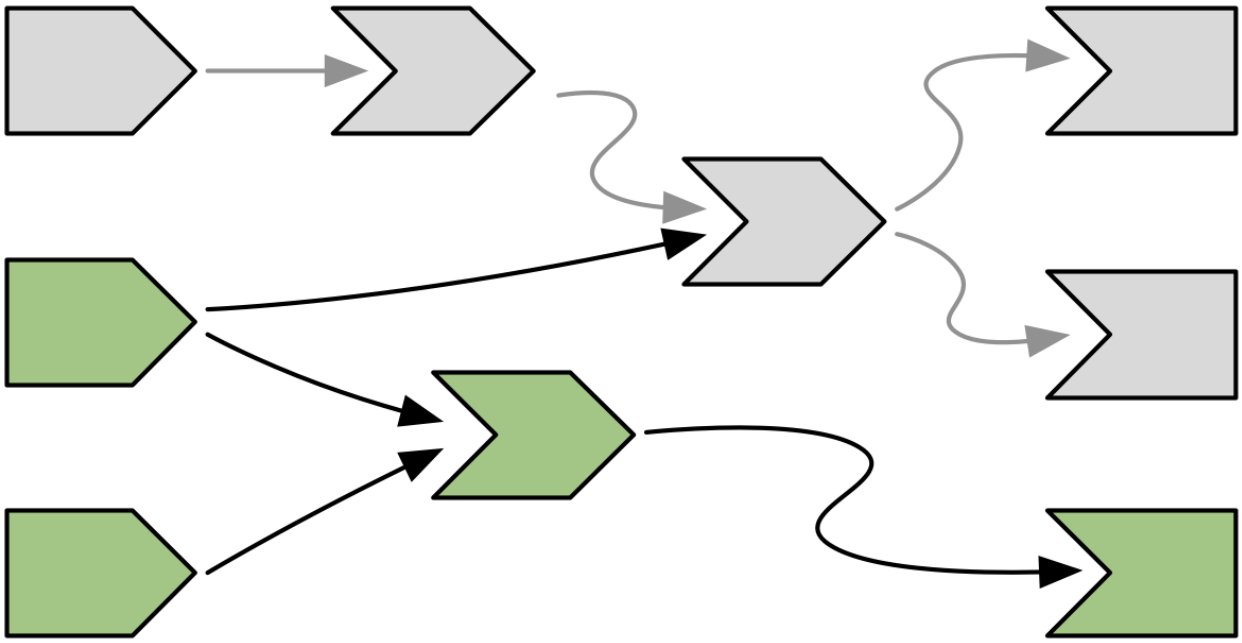

图14.11：无效从输入中流出，从左到右跟随每个箭头。Shiny在失效期间跟随的箭头被涂成浅灰色。

14.4.3 Removing relationships

接下来，每个无效的反应式表达式和输出都会“擦掉”所有进出它的箭头，产生图14.12，并完成无效阶段。

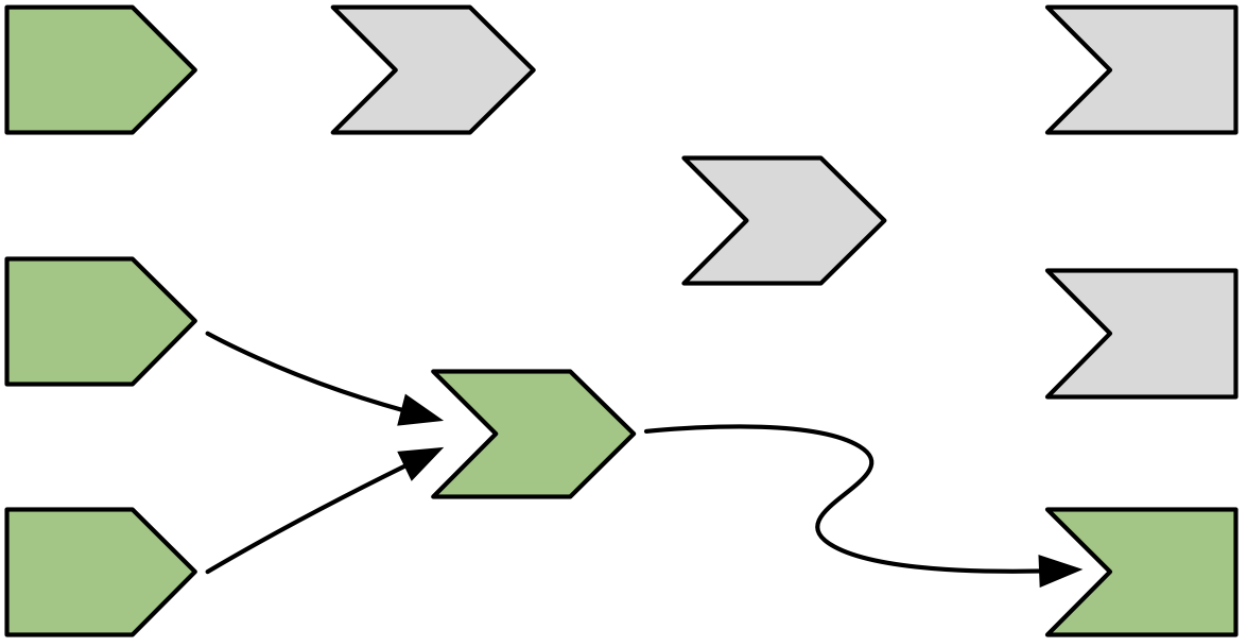

图14.12：无效的节点移除它们之前的所有关系，以便重新发现它们

从节点发出的箭头是一次性通知，下一次，值更改时将触发。现在他们已经变的独立了，他们已经完成了他们的目的，我们可以移除他们。

不太清楚为什么我们擦除进入无效节点的箭头，即使它们来自的节点没有失效。虽然这些箭头代表尚未触发的通知，但无效的节点不再关心它们：反应式消费者只关心通知，以便使自己失效，这种情况已经发生了。

我们如此重视这些关系，现在我们已经把它们扔掉了，这似乎很反常！但这是Shiny反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质时擦除箭头，并让Shiny在重新执行这些节点时重新发现它们周围的关系。我们将在第14.5节中回到这个重要主题。

14.4.4 Re-execution

现在，我们的情况与我们执行第二个输出时非常相似，混合了有效和无效的反应。是时候做我们当时所做的了：执行无效的输出，一次一个，从图14.13开始。

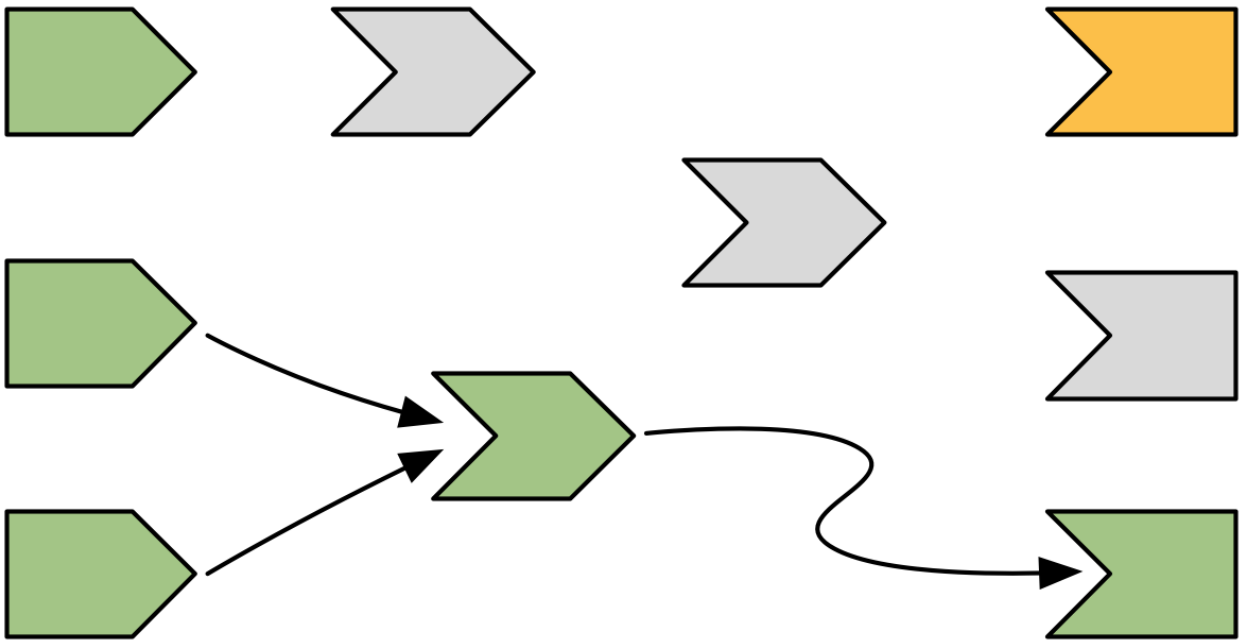

图14.13：现在重新执行的方式与先前执行方式相同，但由于我们不是从头开始，所以需要做的工作更少。

同样，我不会向您展示细节，但最终结果将是一个静止的反应图，所有节点都用绿色标记。这个过程的美妙之处在于，Shiny已经完成了最低限度的工作—我们只完成了更新实际受更改输入影响的输出所需的工作。

14.4.5 Exercises

1. 绘制以下server函数的被动图，然后解释为什么不运行反应。

```
server <- function(input, output, session) {
  sum <- reactive(input$x + input$y + input$z)
  prod <- reactive(input$x * input$y * input$z)
  division <- reactive(prod() / sum())
}
```

2. 以下反应图通过使用`Sys.sleep()`模拟长期运行的计算：

```
x1 <- reactiveVal(1)
x2 <- reactiveVal(2)
x3 <- reactiveVal(3)

y1 <- reactive({
  Sys.sleep(1)
  x1()
})
y2 <- reactive({
  Sys.sleep(1)
  x2()
})
y3 <- reactive({
  Sys.sleep(1)
  x2() + x3() + y2() + y2()
})

observe({
  print(y1())
  print(y2())
  print(y3())
})
```

   如果`x1`发生变化，图表需要多长时间才能重新计算？`x2`或`x3`呢？

3. 如果您尝试创建带有周期的反应图，会发生什么？

```
x <- reactiveVal(1)
y <- reactive(x + y())
y()
```

14.5 Dynamism

在第14.4.3节中，您了解到Shiny“忘记了”反应组件之间的联系，它花费了大量精力进行记录。这使得Shiny具有反应动态，因为它可以在您的应用程序运行时发生变化。这种动态是如此重要，我想用一个简单的例子来加强它：

```
ui <- fluidPage(
  selectInput("choice", "A or B?", c("a", "b")),
  numericInput("a", "a", 0),
  numericInput("b", "b", 10),
  textOutput("out")
)

server <- function(input, output, session) {
  output$out <- renderText({
    if (input$choice == "a") {
      input$a
    } else {
      input$b
    }
  })
}
```

您可能期望反应图看起来像图14.14。

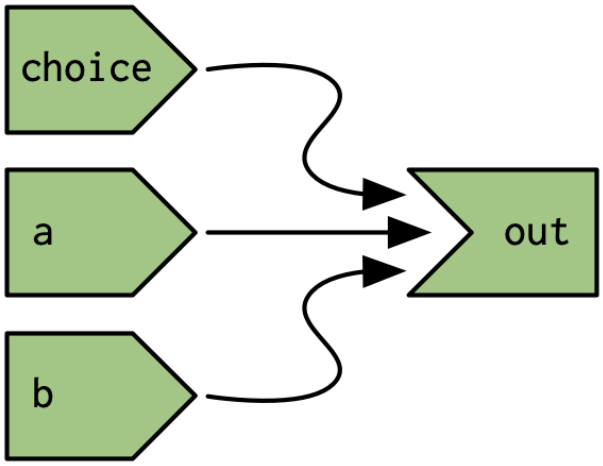

图14.14：如果Shiny静态分析反应性，反应图将始终将`choice`、`a`和`b`连接到`out`。

但是，由于Shiny在输出无效后动态重建图形，它实际上看起来像图14.15中的任何一个图形，这取决于`input$choice`的值。这确保了当输入无效时，Shiny可以完成最少的工作量。其中，如果`input$choice`设置为“b”，则`input$a`的值不会影响`output$out`，也无需重新计算。

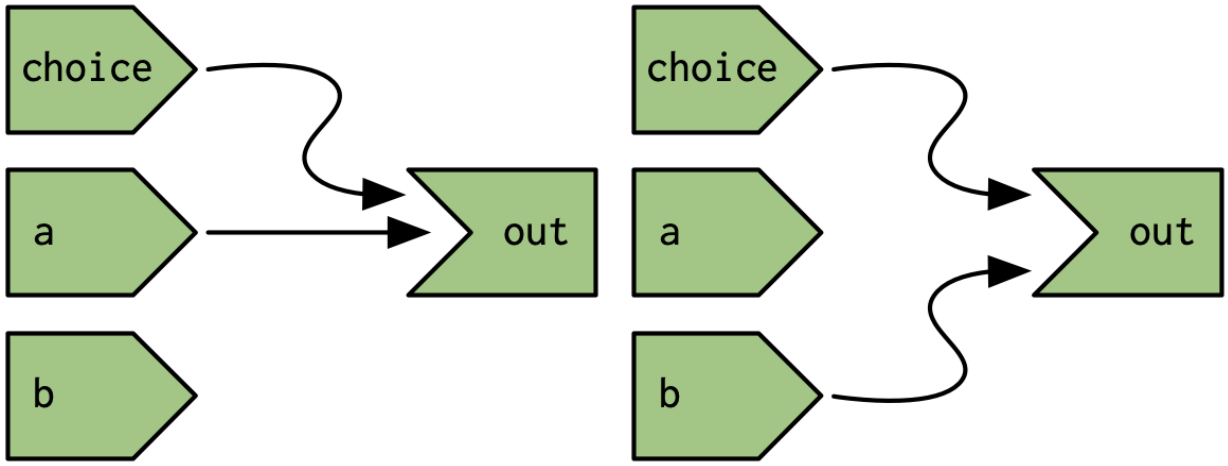

图14.15：但Shiny的反应图是动态的，因此该图要么连接out到`choice`和`a`（左）或`choice`和`b`（右）。

值得注意的是（就像Yindeng Jiang在[他们的博客](#)中所做的那样），一个小的变化将导致输出总是依赖于`a`和`b`：

```
output$out <- renderText({
  a <- input$a
  b <- input$b

  if (input$choice == "a") {
    a
  } else {
    b
  }
})
```

这对正常R代码的输出没有影响，但这里有所不同，因为当您从`input`读取值时，而不是当您使用该值时，会建立反应依赖关系。

14.6 reactlog package

手工绘制反应图是一种强大的技术，可以帮助您理解简单的应用程序，并建立反应式编程的准确模型。但对于拥有许多移动部件的真实应用程序来说，这样做是痛苦的。如果我们能用Shiny对它的了解自动绘制图表，那不是很好吗？这是reactlog包的工作，它生成所谓的**reactlog**，它显示了反应图如何随着时间的推移而演变。

要查看reactlog，您需要首先安装reactlog软件包，打开`reactlog::reactlog_enable()`，然后启动应用程序。然后您有两个选项：

- 当应用程序运行时，按Cmd + F3（Windows上的Ctrl + F3），以显示到该点生成的反应日志。
- 应用程序关闭后，运行`shiny::reactlogShow()`以查看完整会话的日志。

reactlog使用与本章相同的图形约定。最大的区别是，reactlog绘制了每个依赖项，即使它目前没有使用，以保持自动布局的稳定。当前不活跃的连接（但过去或将来会活跃）的连接以细虚线绘制。

图14.16显示了reactlog为我们上面使用的应用程序绘制的反应图。此屏幕截图中有一个惊喜：有三个额外的反应输入（`clientData$output_x_height`、`clientData$output_x_width`和`clientData$pixelratio`）没有出现在源代码中。这些之所以存在，是因为绘图对输出的大小有隐式依赖；每当输出更改大小时，绘图需要重新绘制。

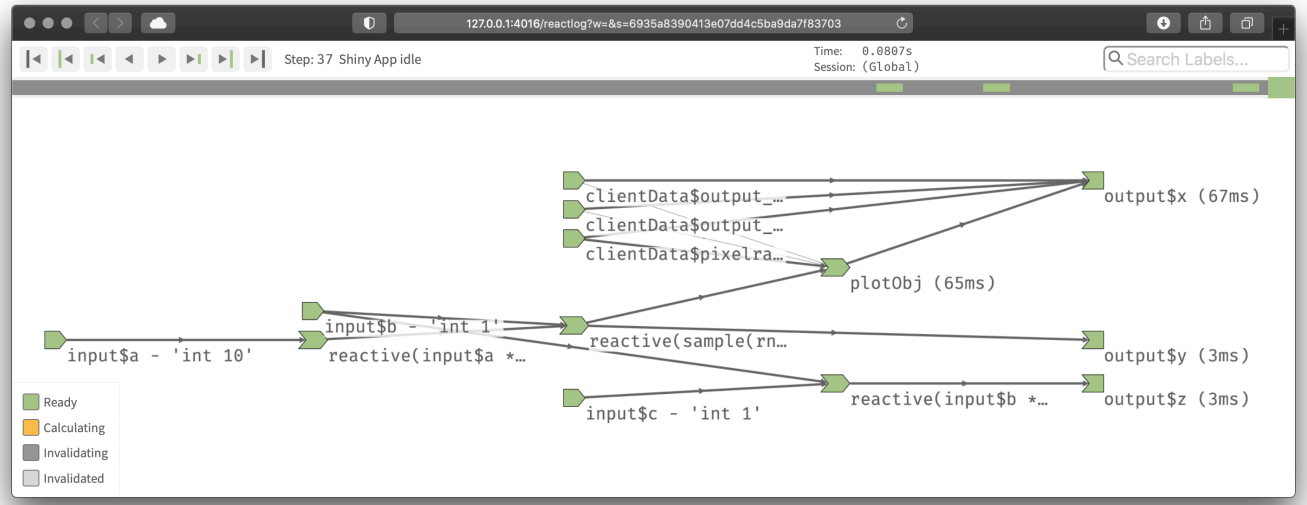

图14.16：由reactlog绘制的我们假设应用程序的反应图

请注意，虽然反应的输入和输出有名称，但有反应的表达式和观察者没有，所以它们被标记为其内容。为了使事情更容易理解，您可能需要使用`label`参数来标记`reactive()`和`observe()`，然后它们将出现在reactlog中。您可以使用表情符号使特别重要的反应在视觉上脱颖而出。

14.7 Summary

在本章中，您精确地了解了反应图是如何运作的。特别是，您第一次了解了无效阶段，这不会立即导致重新计算，而是将反应式消费者标记为无效，以便在需要时重新计算。失效周期也很重要，因为它清除了以前发现的依赖项，以便它们可以自动重新发现，使反应图形具有动态。

现在您已经掌握了大局，下一章将提供一些关于反应值、表达式和输出的底层数据结构的额外细节，我们将讨论定时失效的相关概念。

Chapter 15. Reactive building blocks

现在您已经了解了支撑反应图的理论，并且您有一些实践经验，现在是时候更详细地讨论反应性如何适合编程语言R了。反应式编程有三个基本组成部分：反应式值、反应式表达式和观察器。您已经看到了反应值和表达式的重要部分，因此本章将花更多时间在观察器和输出上（正如您将了解到的那样，这是一种特殊类型的观察器）。您还将学习控制反应图的另外两种工具：隔离（isolation）和定时失效（timed invalidation）。

本章将再次使用反应式控制台，这样我们就可以直接在控制台中尝试反应式编程，而无需每次都启动shiny应用程序。

```
library(shiny)
reactiveConsole(TRUE)
```

15.1 Reactive values

有两种类型的反应值：

- 单个反应值，由`reactiveVal()`创建。
- 反应值列表，由`reactiveValues()`创建。

它们在获取和设置值方面的编程略有不同：

```
x <- reactiveVal(10)
x()       # get
#> [1] 10
x(20)     # set
x()       # get
#> [1] 20

r <- reactiveValues(x = 10)
r$x       # get
#> [1] 10
r$x <- 20 # set
r$x       # get
#> [1] 20
```

不幸的是，这两个相似的对象具有相当不同的接口，但没有办法将它们标准化。然而，虽然它们看起来不同，但它们的行为是一样的，所以你可以根据你喜欢的语法在它们之间进行选择。在这本书中，我使用`reactiveValues()`，因为语法更容易一目了然，但在我自己的代码中，我倾向于使用`reactiveVal()`，因为语法清楚地表明正在发生一些事情。

值得注意的是，这两种类型的响应式值都有所谓的引用语义。大多数R对象都有copy-on-modify语义，这意味着如果您为两个名称分配相同的值，一旦您修改一个名称，连接就会中断：

```
a1 <- a2 <- 10
a2 <- 20
a1 # unchanged
#> [1] 10
```

反应值的情况并非如此——它们总是保留对相同值的引用，以便修改任何副本会修改所有值：

```
b1 <- b2 <- reactiveValues(x = 10)
b1$x <- 20
b2$x
#> [1] 20
```

我们将回到为什么你可能会在第16章中创建自己的反应值。否则，您遇到的大多数反应值将来自server函数的`input`参数。这些与您自己创建的`reactiveValues()`有点不同，因为它们是只读的：您无法修改值，因为Shiny会根据浏览器中的用户操作自动更新它们。

15.1.1 Exercises

1. 这两个反应值列表之间有什么区别？比较获取和设置单个反应值的语法。

```
l1 <- reactiveValues(a = 1, b = 2)
l2 <- list(a = reactiveVal(1), b = reactiveVal(2))
```

2. 设计并执行一个小实验，以验证`reactiveVal()`也具有引用语义。

15.2 Reactive expressions

回想一下，反应性有两个重要属性：它是懒惰的和缓存的。这意味着它只有在实际需要时才起作用，如果在修改值的情况下，连续调用两次，它会返回之前的值。

有两个重要细节我们尚未涵盖：反应式表达式如何处理错误，以及为什么`on.exit()`在它们内部工作。

15.2.1 Errors

反应式表达式缓存错误的方式与缓存值的方式完全相同。例如，以这个反应为例：

```
r <- reactive(stop("Error occured at ", Sys.time(), call. = FALSE))
r()
#> Error: Error occured at 2022-08-23 23:10:12
```

如果我们等一两秒钟，我们可以看到我们遇到与以前相同的错误：

```
Sys.sleep(2)
r()
#> Error: Error occured at 2022-08-23 23:10:12
```

当涉及到反应图时，错误也以与值相同的方式处理：错误通过反应图传播的方式与正确的反应值完全相同。唯一的区别是当错误到达输出或观察器时会发生什么：

- 输出中的错误将显示在app中。
- 观察器中的错误将导致当前会话终止。如果您不希望这种情况发生，您需要在`try()`或`tryCatch()`中包装代码。

同一系统中`req()`命令（第8.1.2节）会发出特殊类型的错误。这个特殊错误导致观察器和输出停止他们正在做的事情，但不会在其他方面失败或引起错误。默认情况下，它将引起输出重置为初始空白状态，但如果您使用`req(..., cancelOutput = TRUE)`，它们将保留当前显示。

15.2.2 `on.exit()`

您可以将`reactive(x())`视为`function() x()`的快捷方式，自动添加懒惰和缓存。如果您想了解Shiny是如何实现的，这一点很重要，但这意味着您可以使用只能在函数内部工作的函数。其中最有用的是`on.exit()`，它允许您在反应表达式完成时运行代码，无论反应是成功返回错误还是失败。这就是`on.exit()`在第8.2.2节中起作用的原因。

15.2.3 Exercises

1. 使用reactlog包观察通过以下应用程序中的反应传播的错误，确认它遵循与值传播相同的规则。

```
ui <- fluidPage(
  checkboxInput("error", "error?"),
  textOutput("result")
)
server <- function(input, output, session) {
  a <- reactive({
    if (input$error) {
      stop("Error!")
    } else {
      1
    }
  })
  b <- reactive(a() + 1)
  c <- reactive(b() + 1)
  output$result <- renderText(c())
}
```

2. 修改上述应用程序以使用`req()`而不是`stop()`验证事件是否仍然以相同的方式传播。当您使用cancelOutput参数时会发生什么?

15.3 Observers and outputs

观察器和输出是反应图中的终端节点。它的两个重要特性与反应式表达式不同：

- 他们渴望（eager）和健忘（forgetful）—他们尽可能快地跑，他们不记得他们之前的行动。这种渴望是“传染性的”，因为如果他们使用反应式表达式，该反应式也会被评估。
- 观察器返回的值被忽略，因为它们被设计为调用其副作用的函数，如`cat()`或`write.csv()`

观察器和输出由相同的底层工具提供动力：`observe()`会设置一个代码块，每次更新它使用的一个反应值或表达式时都会运行。请注意，当您创建它时，观察者会立即运行—它必须这样做才能确定其反应依赖性。

```
y <- reactiveVal(10)
observe({
  message("`y` is ", y())
})
#> `y` is 10

y(5)
#> `y` is 5
y(4)
#> `y` is 4
```

在这本书中，我很少使用`observe()`，因为它是低级工具，为用户友好的`observeEvent()`提供动力。一般来说，你应该坚持使用`observeEvent()`，除非不可能让它做你想做的事。在这本书中，我只向您展示一个需要`observe()`的案例，即第16.3.3节。

`observe()`还为无效输出提供动力。反应输出是一种特殊类型的观察器，具有两个重要属性：

- 当您将它们分配到`output`，它们就会被定义，即`output$text <- ...`创建观察器。
- 它们在不可见时，检测的能力有限（即它们在非活动选项卡中），因此它们不必重新计算。

重要的是要注意，`observe()`和反应输出不会“做”某事，而是“创造”某事（然后根据需要采取行动）。这有助于你理解这个例子中发生了什么：

```
x <- reactiveVal(1)
y <- observe({
  x()
  observe(print(x()))
})
#> [1] 1
x(2)
#> [1] 2
#> [1] 2
x(3)
#> [1] 3
#> [1] 3
#> [1] 3
```

对`x`的每次更改都会触发观察器。观察器本身调用`observe()`设置的*另一个*观察器。因此，每次`x`发生变化时，它都会得到另一个观察器，因此其值会再次打印。

作为一般规则，您只应在server函数的顶层创建观察器或输出。如果您发现自己试图嵌套它们或在输出中创建观察器，请坐下来勾勒出您试图创建的反应图—几乎肯定有更好的方法。在更复杂的应用程序中直接发现这个错误可能更难，但您始终可以使用reactlog：只需在观察器（或输出）中寻找意外的流失，然后跟踪创建它们的原因。

15.4 Isolating code

为了完成这一章，我将讨论两个重要的工具来准确控制反应图的无效方式和时间。在本节中，我将讨论`isolate()`，这是一个为`observeEvent()`和`eventReactive()`提供动力的工具，可以避免在不需要时创建反应依赖关系。在下一节中，您将了解`invalidateLater()`，它允许您在计划中生成反应性无效。

15.4.1 `isolate()`

观察器通常与反应值耦合，以跟踪随时间变化的状态变化。例如，以这个代码为例，它跟踪`x`更改的次数：

```
r <- reactiveValues(count = 0, x = 1)
observe({
  r$x
  r$count <- r$count + 1
})
```

如果您要运行它，您将立即陷入无限循环中，因为观察器将对`x`**和**`count`进行反应依赖；由于观察器修改`count`，它将立即重新运行。

幸运的是，Shiny提供了`isolate()`来解决这个问题。此函数允许您访问反应值或表达式的当前值，**而无需**依赖它：

```
r <- reactiveValues(count = 0, x = 1)
```

```
class(r)
#> [1] "rv_flush_on_write" "reactivevalues"
observe({
  r$x
  r$count <- isolate(r$count) + 1
})

r$x <- 1
r$x <- 2
r$count
#> [1] 2

r$x <- 3
r$count
#> [1] 3
```

与`observe()`一样，很多时候您不需要直接使用`isolate`，因为有两个有用的函数概括了最常见的用法：`observeEvent()`和`eventReactive()`。

15.4.2 `observeEvent()`和`eventReactive()`

当您看到上面的代码时，您可能还记得第3.6节，并想知道为什么我没有使用`observeEvent()`

```
observeEvent(r$x, {
  r$count <- r$count + 1
})
```

事实上，我可以这样做，因为`observeEvent(x,y)`等价于`observe({x;isolate(y)})`。它优雅地将你想听的内容与你想采取的行动脱钩。`eventReactive ()`为`reactives`执行类似的工作：`eventReactive(x,y)`等效于`reactive({x;isolate(y)})`。

`observeEvent()`和`eventReactive()`有额外的参数，允许您控制其操作的详细信息：

- 默认情况下，这两个函数都将忽略任何产生`NULL`的事件（或在操作按钮的特殊情况下，0）。使用`ignoreNULL = FALSE`也处理`NULL`值。
- 默认情况下，当您创建它们时，这两个函数都会运行一次。使用`ignoreInit = TRUE`跳过此运行。
- 仅对于`observeEvent()`，您可以使用`once=TRUE`,只运行一次处理程序。

这些很少需要，但了解起来很好，以便您可以在需要时从文档中查找详细信息。

15.4.3 Exercises

1. 使用server函数完成下面的应用程序，该功能仅在按下按钮时才会更新来自`x`的`out`值。

```
ui <- fluidPage(
  numericInput("x", "x", value = 50, min = 0, max = 100),
  actionButton("capture", "capture"),
  textOutput("out")
)
```

15.5 Timed invalidation

`isolate()`减少反应图失效的时间。本节的这个主题，`invalidateLater()`恰恰相反：当数据没有变化时，它允许您使反应图无效。您在第3.5.1节中看到了`reactiveTimer()`的例子，但现在是时候讨论为它提供动力的底层工具了：`invalidateLater()`

`invalidateLater(ms)`引起任何反应式消费者在未来`ms`毫秒后失效。它可用于创建动画和连接到Shiny反应框架之外的数据源，这些数据源可能会随着时间的推移而变化。例如，以下的反应式表达式将每半秒自动生成10个新的随机数：

```
x <- reactive({
  invalidateLater(500)
  rnorm(10)
})
```

该观察者将用随机数递增累积求和：

```
sum <- reactiveVal(0)
observe({
  invalidateLater(300)
  sum(isolate(sum()) + runif(1))
})
```

在下面的部分中，您将学习如何使用`invalidateLater()`从磁盘读取不断变化的数据，如何避免`invalidateLater()`陷入无限循环，以及一些重要的失效发生时间细节。

15.5.1 Polling

`invalidateLater()`的一个有用应用程序是将Shiny连接到R之外正在变化的数据。例如，您可以使用以下反应器每秒重新读取一个csv文件：

```
data <- reactive({
  on.exit(invalidateLater(1000))
  read.csv("data.csv")
})
```

这将不断变化的数据连接到Shiny的反应图，但它有一个严重的缺点：当您使反应无效时，您也会使所有下游反应式消费者无效，因此即使数据相同，所有下游工作也必须重做。

为了避免这个问题，Shiny提供了`reactivePoll()`，它需要两个函数：一个执行相对不消耗资源的检查，以查看数据是否已更改；另一个相对好资源的函数，实际上进行计算。我们可以使用`reactivePoll()`重写之前的反应式表达式，如下所示。

```
server <- function(input, output, session) {
  data <- reactivePoll(1000, session,
    function() file.mtime("data.csv"),
    function() read.csv("data.csv")
  )
}
```

在这里，我们习惯了`file.mtime()`，它返回上次修改文件的时间，作为不消耗资源的检查，看看我们是否需要重新加载文件：

在文件更改时，阅读文件是一项常见的任务，因此Shiny提供了一个更具体的助手，只需要文件名和阅读器函数：

```
server <- function(input, output, session) {
  data <- reactiveFileReader(1000, session, "data.csv", read.csv)
}
```

如果您需要从其他来源（例如数据库）读取不断变化的数据，您需要想出自己的`reactivePoll()`代码。

15.5.2 Long running reactive

如果您正在执行长时间运行的计算，您需要考虑一个重要问题：何时应该执行`invalidateLater()`？例如，以这个反应为例：

```
x <- reactive({
  invalidateLater(500)
  Sys.sleep(1)
  10
})
```

假设Shiny在时间0开始反应运行，它将在时间500ms请求无效。反应需要1000ms才能运行，所以现在是时间1000ms，它立即失效，必须重新计算，然后设置另一个无效：我们被困在一个无限循环中。

另一方面，如果您在最后运行`invalidateLater()`，它将在完成后500毫秒无效，因此反应将每1500毫秒重新运行一次。

```
x <- reactive({
  on.exit(invalidateLater(500), add = TRUE)
  Sys.sleep(1)
  10
})
```

这是我们更喜欢`invalidateLater()`而不是我们之前使用的更简单的`reactiveTime()`的主要原因：它可以让您更好地控制失效发生的确切时间。

15.5.3 Timer accuracy

`invalidateLater()`中指定的毫秒数是一个礼貌的请求，而不是一个要求。当您要求无效时，R可能正在做其他事情，因此您的请求必须等待。这实际上意味着这个数字是最小的，无效化可能需要比您预期的更长的时间。在大多数情况下，这并不重要，因为微小的差异不太可能影响用户对应用程序的感知。但是，在会积累许多小错误的情况下，您应该计算确切的经过时间，并用它来调整您的计算。

例如，以下代码根据速度和经过的时间计算距离。与其假设`invalidateLater(100)`，总是延迟整整100毫秒，不如计算经过的时间，并将其用于计算位置。

```
velocity <- 3
r <- reactiveValues(distance = 1)

last <- proc.time()[[3]]
observe({
  cur <- proc.time()[[3]]
  time <- last - cur
  last <<- cur

  r$distance <- isolate(r$distance) + velocity * time
  invalidateLater(100)
})
```

如果您没有仔细做动画，请随意忽略`invalidateLater()`的固有变体。请记住，这是一个礼貌的请求，而不是一个要求。

15.5.4 Exercises

1. 为什么这种反应永远不会被执行？你的解释应该谈论反应图和无效。

   ```
   server <- function(input, output, session) {
     x <- reactive({
       invalidateLater(500)
       rnorm(10)
     })
   }
   ```

2. 如果您熟悉SQL，请使用`reactivePoll()`在添加新行时仅重新读取虚构的“结果”表。您可以假设结果表有一个`timestamp`字段，其中包含添加记录的日期时间。

15.6 Summary

在本章中，您了解了有关使Shiny工作的构建块的更多信息：反应值、反应式表达式、观察器和定时评估。现在，我们将把注意力转向反应值和观察器的特定组合，这使我们能够摆脱反应图的一些约束（无论好坏）。

Chapter 16. Escaping the graph

16.1 Introduction

Shiny的反应式编程框架非常有用，因为它会自动确定当输入更改时更新所有输出。

在本章中，您将学习如何将`reactiveValues()`和`observe()`/`observeEvent()`结合起来，将反应图的右侧连接到左侧。这些技术很强大，因为它们可以让您手动控制图形的某些部分。但它们也很危险，因为它们允许您的应用程序做不必要的工作。最重要的是，您现在可以创建无限循环，您的应用程序会陷入永无止境的更新循环中。

如果您觉得本章中探讨的想法很有趣，您可能还想看看shinySignals和rxtools软件包。这些都是实验包，旨在探索“高阶”反应，即从其他反应以编程方式创建的反应。我不建议你在“真实”应用程序中使用它们，但阅读源代码可能会有启发性。

```
library(shiny)
```

16.2 What doesn't the reactive graph capture?

在第14.4节中，我们讨论了当用户导致输入无效时会发生什么。还有两种重要情况，可能会使输入无效：

- 您调用设置`value`参数的`update`函数。这会向浏览器发送一条消息，以更改输入的值，然后通知R输入值已更改。
- 您修改了反应值的值（使用`reactiveVal()`或`reactiveValues()`创建）。

重要的是要明白，在这两种情况下，反应值和观察器之间*都没有*产生反应依赖关系。虽然这些操作导致图表无效，但它们不会通过新的连接记录。

为了使这个想法具体化，请采用以下简单的应用程序，反应图如图16.1所示。

```
ui <- fluidPage(
  textInput("nm", "name"),
  actionButton("clr", "Clear"),
  textOutput("hi")
)
server <- function(input, output, session) {
  hi <- reactive(paste0("Hi ", input$nm))
  output$hi <- renderText(hi())
  observeEvent(input$clr, {
    updateTextInput(session, "nm", value = "")
  })
}
```

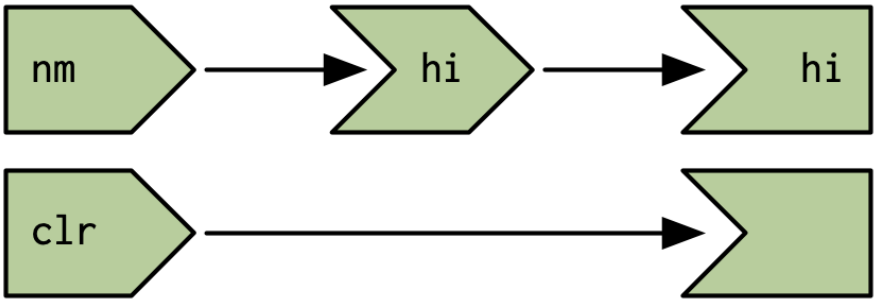

图16.1：反应图不记录未命名的观察器和`nm`输入之间的连接；这种依赖关系超出了其范围。

当你按下清除按钮时会发生什么？

1. `input$clr`无效，然后使观察器无效。
2. 观察器重新计算，重新创建对`input$clr`的依赖性，并告诉浏览器更改输入控件的值。
3. 浏览器更改`nm`的值。
4. `input$nm`无效，无效`hi()`，然后`output$hi`。
5. `output$hi`重新计算，迫使`hi()`重新计算。

这些操作都不会改变反应图，因此它仍然如图16.1所示，该图不会捕获从观察器到`input$nm`的连接。

16.3 Case studies

接下来，让我们看看一些有用的案例，您可以结合`reactiveValues()`和`observeEvent()`或`observe()`来解决其他非常具有挑战性（如果不是不可能的话）的问题。这些是您自己应用程序的有用模板。

16.3.1 One output modified by multiple inputs

首先，我们将解决一个非常简单的问题：我想要一个由多个事件更新的通用文本框。

```
ui <- fluidPage(
  actionButton("drink", "drink me"),
  actionButton("eat", "eat me"),
  textOutput("notice")
)
server <- function(input, output, session) {
  r <- reactiveValues(notice = "")
  observeEvent(input$drink, {
    r$notice <- "You are no longer thirsty"
  })
  observeEvent(input$eat, {
    r$notice <- "You are no longer hungry"
  })
  output$notice <- renderText(r$notice)
}
```

在下一个示例中，事情变得稍微复杂一些，我们有一个带有两个按钮的应用程序，可以让你增加和减少值。我们使用`reactiveValues()`来存储当前值，然后在按下适当的按钮时使用`observeEvent()`来增加和减少值。这里的主要额外复杂性是，`r$n`的新值取决于以前的值。

```
ui <- fluidPage(
  actionButton("up", "up"),
  actionButton("down", "down"),
  textOutput("n")
)
server <- function(input, output, session) {
  r <- reactiveValues(n = 0)
  observeEvent(input$up, {
    r$n <- r$n + 1
  })
  observeEvent(input$down, {
    r$n <- r$n - 1
  })

  output$n <- renderText(r$n)
}
```

图16.2显示了本示例的反应图。再次注意，反应图不包括从观察器回到反应值的任何连接。

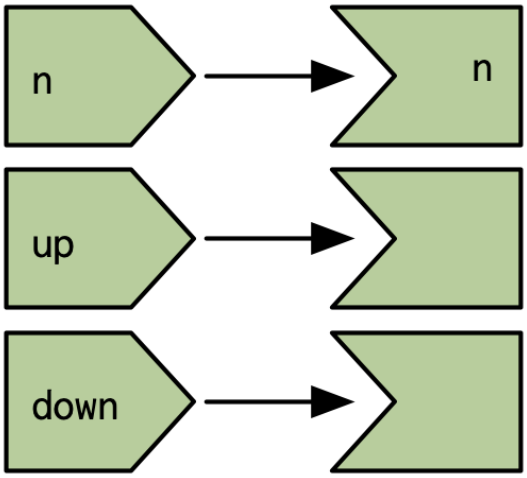

图16.2：反应图不捕获从观察器到输入值的连接

16.3.2 Accumulating inputs

如果您想积累数据以支持数据输入，这是一个类似的模式。这里的主要区别是，在用户单击添加按钮后，我们使用`updateTextInput()`重置文本框。

```
ui <- fluidPage(
  textInput("name", "name"),
  actionButton("add", "add"),
  textOutput("names")
)
server <- function(input, output, session) {
  r <- reactiveValues(names = character())
  observeEvent(input$add, {
    r$names <- c(input$name, r$names)
    updateTextInput(session, "name", value = "")
  })

  output$names <- renderText(r$names)
}
```

我们可以通过提供删除按钮，并确保添加按钮不会创建重复名称，来使其更加有用：

```
ui <- fluidPage(
  textInput("name", "name"),
  actionButton("add", "add"),
  actionButton("del", "delete"),
  textOutput("names")
)
server <- function(input, output, session) {
  r <- reactiveValues(names = character())
  observeEvent(input$add, {
    r$names <- union(r$names, input$name)
    updateTextInput(session, "name", value = "")
  })
  observeEvent(input$del, {
    r$names <- setdiff(r$names, input$name)
```

```
    updateTextInput(session, "name", value = "")
  })

  output$names <- renderText(r$names)
}
```

16.3.3 暂停动画

另一个常见的用例是提供一个启动和停止按钮，用于控制一些重复发生的事件。此示例使用一个`running`反应值来控制数字是否递增，并使用`invalidateLater()`来确保观察器在运行时每250毫秒失效一次。

```
ui <- fluidPage(
  actionButton("start", "start"),
  actionButton("stop", "stop"),
  textOutput("n")
)
server <- function(input, output, session) {
  r <- reactiveValues(running = FALSE, n = 0)

  observeEvent(input$start, {
    r$running <- TRUE
  })
  observeEvent(input$stop, {
    r$running <- FALSE
  })

  observe({
    if (r$running) {
      r$n <- isolate(r$n) + 1
      invalidateLater(250)
    }
  })
  output$n <- renderText(r$n)
}
```

请注意，在这种情况下，我们无法轻松使用`observeEvent()`，因为我们根据`running()`是`TRUE`还是`FALSE`执行不同的操作。由于我们不能使用`observeEvent()`我们必须使用`isolate()`——如果我们不这样做，该观察者也会对`n`进行反应依赖，它会更新`n`，因此它会卡在无限循环中。

希望这些例子能开始让您了解`reactiveValues()`和`observe()`编程的感觉。这非常势在必行：当这种情况发生时，就做那个；当那种情况发生时，做另一件事。这使得小规模更容易理解，但当更大的部分开始相互作用时更难理解。因此，一般来说，您会希望尽可能谨慎地使用它，并保持隔离，以便尽可能少的观察器修改反应值。

16.3.4 Exercises

1. 提供一个server函数，在单击`Normal`时从正态分布中绘制100个随机数的直方图，以及100个来自均匀分布的随机数。

```
ui <- fluidPage(
  actionButton("rnorm", "Normal"),
  actionButton("runif", "Uniform"),
  plotOutput("plot")
)
```

2. 从上面修改您的代码以使用此UI：

```
ui <- fluidPage(
  selectInput("type", "type", c("Normal", "Uniform")),
  actionButton("go", "go"),
  plotOutput("plot")
)
```

3. 从上一个答案中重写代码，以消除使用`observe()`/`observeEvent()`，并仅使用`reactive()`。为什么你可以为第二个用户界面这样做，但不能为第一个用户界面这样做?

16.4 Anti-patterns

一旦你掌握了这种模式，就很容易养成坏习惯：

```
server <- function(input, output, session) {
  r <- reactiveValues(df = cars)
  observe({
    r$df <- head(cars, input$nrows)
  })

  output$plot <- renderPlot(plot(r$df))
  output$table <- renderTable(r$df)
}
```

在这个简单的案例中，与使用`reactive()`的替代方案相比，此代码没有做太多的额外工作：

```
server <- function(input, output, session) {
  df <- reactive(head(cars, input$nrows))

  output$plot <- renderPlot(plot(df()))
  output$table <- renderTable(df())
}
```

但仍然有两个缺点：

- 如果表格或绘图位于当前不可见的选项卡中，观察器仍将绘制它们。
- 如果`head()`抛出错误，`observe()`将终止应用程序，它不会被传播。但`reactive()`将传播它，因此它显示的反应是抛出错误。

随着应用程序变得更加复杂，情况会越来越糟。很容易恢复到第13.2.3节中描述的事件驱动编程情况。您最终会做很多艰苦的工作来分析应用程序中的事件流，而不是依靠Shiny自动处理它。

比较两个反应图是信息丰富的。图16.3显示了第一个示例中的图表。它具有误导性，因为它看起来不像`nrows`连接到`df()`。如图16.4所示，使用反应式，使精确的连接易于看到。拥有一个尽可能简单的反应图，对人类和Shiny都很重要。一个简单的图表对人类来说更容易理解，一个简单的图表对Shiny来说更容易优化。

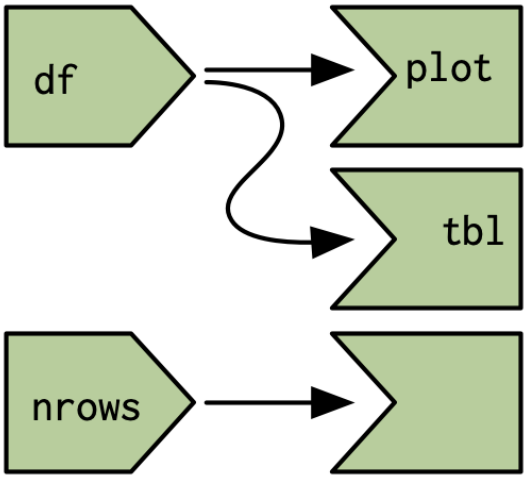

图16.3：使用反应值和观察器使部分图表断开连接

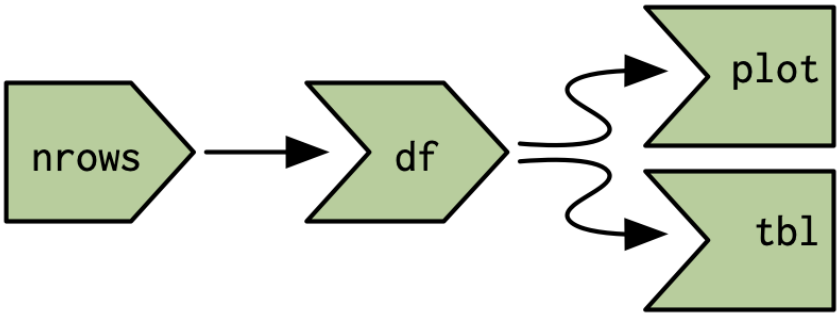

图16.4：使用反应使组件之间的依赖性非常清晰。

16.5 Summary

在最后四章中，您对Shiny使用的反应式编程模型有了更多的了解。您已经了解了为什么反应式编程很重要（它允许Shiny完成所需的工作，而不是更多），以及反应图的细节。您还了解了一些基本构建块，如何在shiny隐藏环境下工作，以及如何在需要时使用它们来摆脱反应图的约束。

这本书的其余部分通过软件工程的视角讨论了Shiny。在接下来的七章中，您将学习如何在扩展大小和影响力不断增长时，让您的shiny应用程序保持可维护、性能和安全。