4.1 介绍

在前一章，我们讨论了shiny的UI部分，这次讨论shiny的server部分，server部分的代码使你的shiny部分更加的生动。
在shiny的server部分，你使用反应式编程表达你的逻辑。反应式编程是一种优雅的、强大的的编程范例。但是刚开始可能会让人疑惑，因为不同于写脚本，反应式编程的中心思想是依赖关系图。所以当输出改变的时候，输出也会自动的相应的发生变化。这使得一个app的流程变得非常简单，但是需要时间去了解如何将各个小组件组合在一起。
本章将对响应式编程进行简要的介绍，教你使用最基础的响应式编程构造shiny app。将从server部分开始，详细的介绍输入和输出的工作原理。接下来我们将讨论最简单的响应式（输入和输出直接相连）。最后讨论反应表达式如何消除重复的工作，最后，我们将回答新手遇到的一些常见问题。

```
library(shiny)
```

4.2 server函数

正如你所以见到的，每一个shiny的框架都是这样的：

```
library(shiny)

ui <- fluidPage(
  #front end interface 前端，用户直接交互
)

server <- function(input, output, session) {
  # back end logic 后端，运行流程图
}

shinyApp(ui, server)
```

上一章介绍了前端（UI)的基础知识。UI这个对象包含呈现给用户的HTML。UI部分简单，是因为每一个用户都获得相同的HTML。而每个shiny用户需要一个独立的程序，所以server就更加复杂。也就是说，用户A移动滑块的时候，用户B不会受到用户A操作的影响。
为了实现独立性，shiny每次启动server部分的时候，都会先创建一个新的对话。当server函数被调用的时候，会在本地创建一个新的环境来保证函数独立于其他的会话里的函数。这导致每一个会话都是独特的。这也就是为什么每一次函数式编程都放在server里面。

server部分需要三个参数：`input`、`output`、`session`。你自己永远不会调用server部分。所以你自己不会创建这些对象。相反，当shiny被运行的那一刻（这个时候自动创建一个会话），从而server部分开始运行在会话里。目前我们将着重介绍`input`和`output`参数，`session`在以后的章节再介绍。

4.2.1 输入

`input`参数类似于列表的一个对象，包含所有由浏览器端发送的数据，这些数据在`input`里的名字由输入的进行命名。比如说，你的UI有个数值输入id叫count的，代码如下：

```
ui <- fluidPage(
  numericInput("count", label = "Number of values", value = 100)
)
```

那么你想获得这个数值，就可以使用`input$count`。这个初始值是100，具体多少会根据用户的调整自动更新。
和R里面的list不同，这个`input`列表里面的值在server部分只能被读取，不能被修改。像下main的操作，就会出错：

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

shinyApp(ui, server)
#> Error: Attempted to assign value to a read-only reactivevalues object
```

这个错误发生的原因是因为，`input`反映的是UI端的变化情况。shiny'的UI端是一切值的来源。如果修改`input`的值，可能会导致数值不一样。也就是说，你滑块在你看到的是这个值，但是在server部分被你修改成另外一个值。这样会增加编程的复杂度。在第八章，将了解如何使用`updateNumericInput()`修改浏览器中的值，然后`input$count`将会自动的改变。

还有一个关于`input`输出更加重要的事情：哪一个被读取是可选择的。读取一个`input`，必须要放在由`renderText()`或者`reactive()`创建的反应式上下文中（就相当于绘画一样，你必须在纸上画，而不能去纸背后的画架上画）。很快将要介绍这一点。这是一个很重要约束，使得你可以自动更新输出（当输入变化的时候）。下面就是一个错误的示例：

```
server <- function(input, output, session) {
  message("The value of input$count is ", input$count)
}

shinyApp(ui, server)
#> Error: Operation not allowed without an active reactive context.
#> (You tried to do something that can only be done from inside
#> a reactive expression or observer.)
```

4.2.2 输出

`output`和`input`非常相似。同样是一个类似于列表的对象。列表里面的变量的名字由输出的id决定。和`input`最主要的不同就是`output`是发送数据而不是接收数据。`output`里面的数据要始终和`render()`这一系列的函数配套使用。就像是下面的案例：

```
ui <- fluidPage(
  textOutput("greeting")
)

server <- function(input, output, session) {
  output$greeting <- renderText("Hello human!")
}
```

注意输出的id在UI端引用，在server部分不引用。
这个`render**`函数有两个做了两件事：

1. 设置了一个特殊的上下文（输入变化的时候，输出自动变化输出）

2. 将R代码结果转换为适合网页显示的HTML。

和`input`一样，`output`也很挑剔

1. 忘记使用`render`函数：

```
server <- function(input, output, session) {
  output$greeting <- "Hello human"
}
shinyApp(ui, server)
#> Error: Unexpected character output for greeting
```

2. 试图从output读取数据：

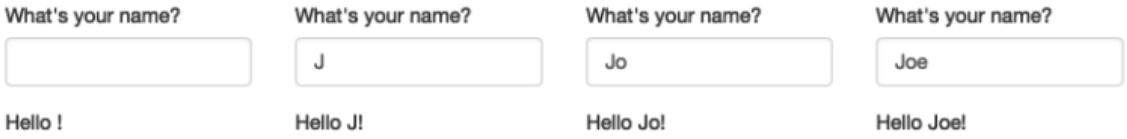
```
server <- function(input, output, session) {
  message("The greeting is ", output$greeting)
}
shinyApp(ui, server)
#> Error: Reading objects from shinyoutput object not allowed.
```

4.3 反应式编程

一个shiny app只有输入或者只有输出是非常无聊的。真正有趣的shiny app是两个都有，让我们看这个简单的程序：

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

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

很难在这本书里面说明这个工作原理。但是如果运行这个app。在框里写入字符串，下面问候语就会更新。



这就是shiny的一个优点：不需要告诉shiny什么时候更新输出，因为shiny自动更新。这个是怎么工作的？到底在这个函数上发生了什么？我们剖析server里面的这个函数。

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

这个代码很容易理解为：使用`paste0`函数将"hello "和`input$name`粘贴起来然后传递给`output$greeting`。但是这个思维是错误的，应该这样想：使用这个`renderText()`模型，只需要构造一次。shiny在`input$name`变化后自动更新。
这个shiny app可以正常运行不是因为代码告诉shiny创建一个字符串并传递给UI部分，而是告诉shiny在需要的时候如何创造一个字符串。到底是什么时候运行取决于shiny：可能是在打开shiny的时候就运行；可能要等一段时间再运行；或者运行很多次；或者根本不会运行。这不是因为shiny反复无常，只是shiny是决定什么时候执行代码，而不是你决定（其实你确定shiny执行的方式：你所创建的反应表达式）。你只是提供了食谱，而不是要求做出三明治。

4.3.1 命令式与声明式编程

这两种编程形式的区别就像是命令和配方的区别：

1. 命令式编程中，你发出特定的指令，代码立刻执行。就像是使用R代码加载数据、数据转换、可视化等，结果马上出现。
2. 声明式编程中，你表达你的目标和约束，然后依靠别人决定什么时候执行，这种风格就是声明式编程。

对命令式编程说“做一个三明治”。对声明式编程说“当我打开冰箱的时候，里面有个三明治”。命令式编程是果断地，声明式编程是被动的。

大部分时候，声明式编程极大的解放了程序员：你描述了目标，而软件在无需干预的时候就实现了这些目标。声明式也有缺点：明确的知道你想什么，但是无法写出一个框架给系统执行。本书的目的是帮助你加深对基础理论的理解，减少这样的情况的发生。

4.3.2 偷懒机制

shiny中使用的声明式编程的优点就是运行app可以偷懒。shiny app只会执行所需要的最小工作量（在输入变化更新输出的时候）。但是这样的范式导致找程序bug的时候非常麻烦，比如能否看到下面的bug：

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

如果仔细看上面的代码，会发现拼写错误：`greeting`被写成`greetning`。这在shiny不会发生错误，因为`greetnig`在shiny的UI部分不存在，导致这一部分的`renderText()`始终不会被运行。
如果在你写的shiny app里面，你不能搞清楚为什么你的代码不运行，你应该检查你的UI和server部分是否使用了相同的变量名字。

4.3.3 反应图

shiny的偷懒机制还有一个更加重要的属性。在大部分R代码中，代码都是从第一行按照顺序运行到最后一行的。但是在shiny里面并不是这样的。因为代码只会在需要的时候才会运行。为了理解这样的执行顺序，您应该了解一些反应图（描述输出和输出的联系）。上面的app的反应图非常简单：

反应图的输入和输出都有对应的形状，不论输出什么时候需要输入，都要将输入和输出联系到一起，这个图告诉你，`greeting`需要被计算当`name`发生改变的时候，我们通常将这种关系描述为：`greeting`对`name`有反应式依赖。

我们可以将它们紧紧放在一起，表示他们是一起的，但是这通常适用于最简单的app。

这个反应图是一个强大的工具用来帮助理解shiny app是如何工作的，当你的shiny app变得更加复杂的时候，从全局的角度快熟绘制草图，提醒你将所有的零件如何组合在一起，在未来，我们将介绍如何使用reactlog来绘制反应图。

4.3.4 反应表达式

在反应图里看到最重要的成分就是反应表达式，我们在不久的将来详细介绍反应表达式。现在只是将它看为：通过观察反应图的节点来减少重复代码的工具。
在简单的app里面，我们不需要反应表达式。但是无论如何，我都会创建一个，下面的代码和对印的反应图：

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

反应表达式接受输入并产生输出，所以上面的`text`的形状将输入和输出的特征结合起来。上面的图形的形状帮助你理解反应式如何将输入和输出结合起来。

4.3.5 执行顺序

一个很重要一点就是，shiny app里面的代码执行顺序取决于你的反应图而不是你的代码从上到下的顺序。这个和R的代码从上向下运行是完全不一样的。我们看下面的例子：

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

上面的代码的代码顺序被颠倒了，你可能认为会发生执行错误因为`text()`函数执行的时候还没有被创建（从代码的行数顺序上来看）。但是请记住，shiny 是懒惰的，所以代码只会在会话开始，`text()`被创建之后。
相反，上面的代码和原来的代码运行是一样的，因为都有着相同的反应图。但是重新修改你的代码，增加代码的可读性，确保反应表达式只引用上面的内容，而不是下面的内容，这样使得代码更加容易理解（写代码要规范）。

这个概念非常重要，与很多的其他的R代码定义不同，因此，再说一遍也不为过：代码的运行顺序是由反应图决定的，而不是代码在server部分从上向下的位置决定的。

4.4.6 练习

4.4 反应式

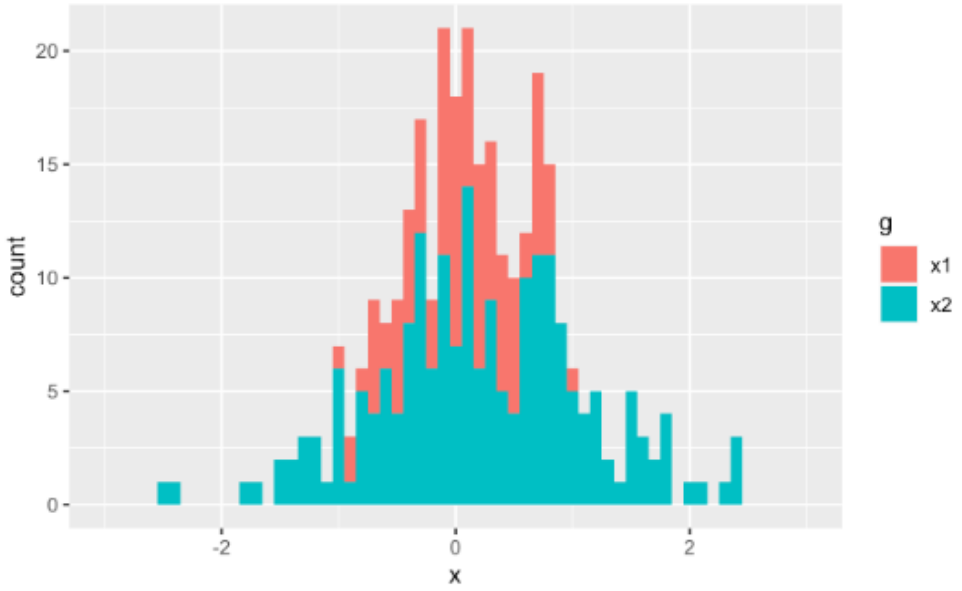
4.4.1 动机

想象我需要两个仿真数据，用这两个数据做假设检验和画图。我做了一些实验，并提出以下两个功能：使用`histgram()`做两个分布的直方图，使用`t_test()`做t检验比较均值，并将结果显示出来。

```
library(ggplot2)

histogram <- 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, fill = g)) +
    geom_histogram(binwidth = binwidth) +
    coord_cartesian(xlim = xlim)
}

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

  sprintf(
    "p value: %0.3f\n[%0.2f, %0.2f]",
    test$p.value, test$conf.int[1], test$conf.int[2]
  )
}
```

如果我再生成两个数据，我就可以使用上面的函数比较两个变量：

```
x1 <- rnorm(100, mean = 0, sd = 0.5)
x2 <- rnorm(200, mean = 0.15, sd = 0.9)

histogram(x1, x2)
cat(t_test(x1, x2))
#> p value: 0.039
#> [-0.32, -0.01]
```



实际中，你可能需要大量的探索，这里跳过探索部分，尽可能快速的将代码转到shiny app。将命令性的代码转到shiny app里面是一项重要的技术。提取的代码越多，越容易理解（熟能生巧）。这是一个很好的软件工程，因为他帮助隔离问题：app的功外部函数集中于计算，app内部的函数集中于响应用户操作，

4.4.2 app

我倾向于使用两个顺手的工具来探索这一系列的仿真。shiny app非常擅长做这两个方面，因为这样你减少不必要的工作：修改代码，再运行代码。我将下面的代码包装到shiny app里面，这样我可以使用交互式来修改输入。

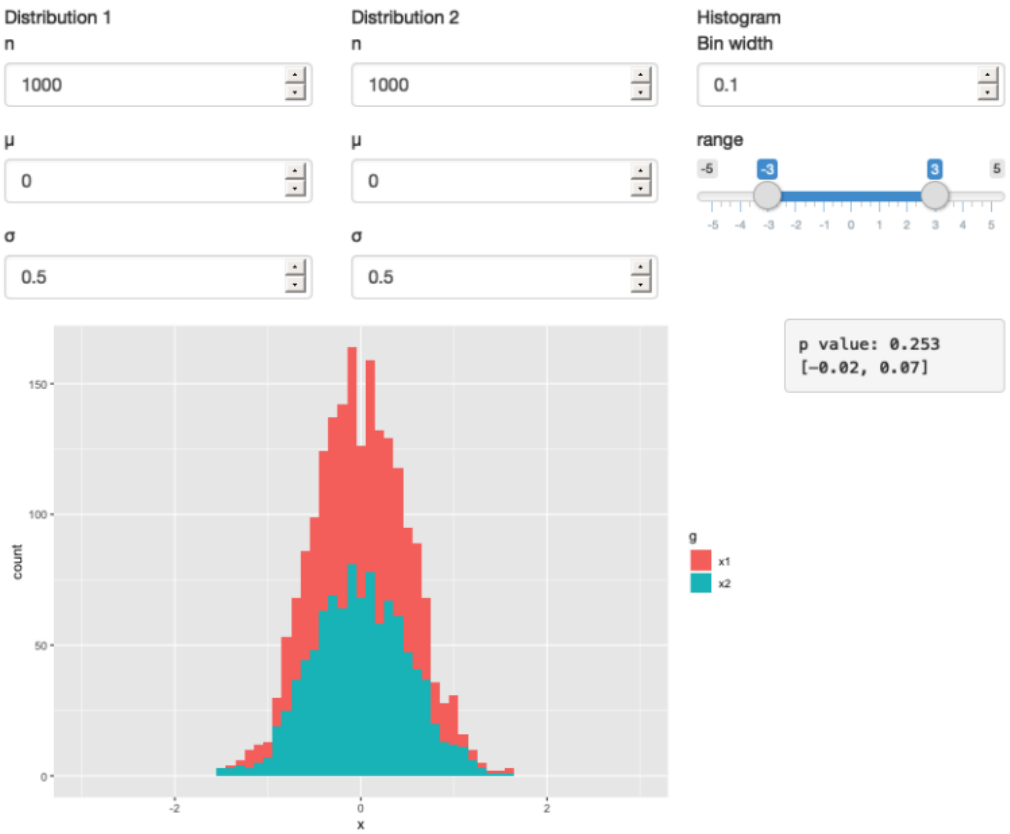
从UI部分开始着手，第一行有三列输出控件（分布1、分布2、画图控制）。第二行是两列（比较宽的输出画图，比较窄的输出检验结果）。

```
ui <- fluidPage(
  fluidRow(
    column(4,
      "Distribution 1",
      numericInput("n1", label = "n", value = 1000, min = 1),
      numericInput("mean1", label = "μ", value = 0, step = 0.1),
      numericInput("sd1", label = "σ", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Distribution 2",
      numericInput("n2", label = "n", value = 1000, min = 1),
      numericInput("mean2", label = "μ", value = 0, step = 0.1),
      numericInput("sd2", label = "σ", value = 0.5, min = 0.1, step = 0.1)
    ),
    column(4,
      "Histogram",
      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部分：在创建好两个分布之后，调用两个响应函数：`histgram()`和`t_test()`函数。

```
server <- function(input, output, session) {
  output$hist <- renderPlot({
    x1 <- rnorm(input$n1, input$mean1, input$sd1)
    x2 <- rnorm(input$n2, input$mean2, input$sd2)

    histogram(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)
  })
}
```

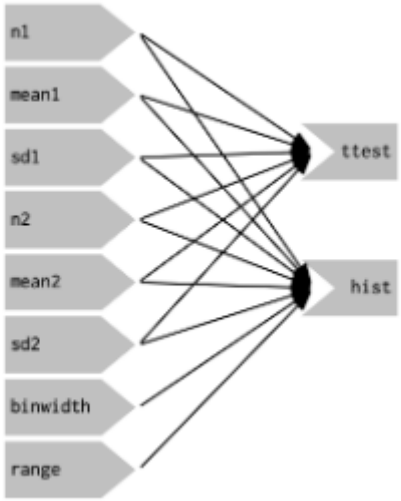


4.4.3 反应式流程图

在绘制这个app的反应图的时候，shiny足够聪明更新输出当输入发生变化，但是在输出中选择性的运行代码块不够聪明，换句话来说：要么整体执行，要么不执行。
比如来说，看来自server部分的代码：

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

阅读上面的代码会发现，只需要更新`x1`当`n1`、`mean1`或者`sd1`变化；只需要更新`x2`当`n2`、`mean2`或者`sd2`变化。但是上面代码中，当其中的一个`x1`、`n1`、`mean1`、`x2`、`n2`、`mean2`发生变化的时，shiny会更新所有的`x1`、`x2`，这导致下面的反应图如下：



你将注意到上面的反应图关系非常密集，基本上每一个输入都直接联系着输出。这样照成两个问题：

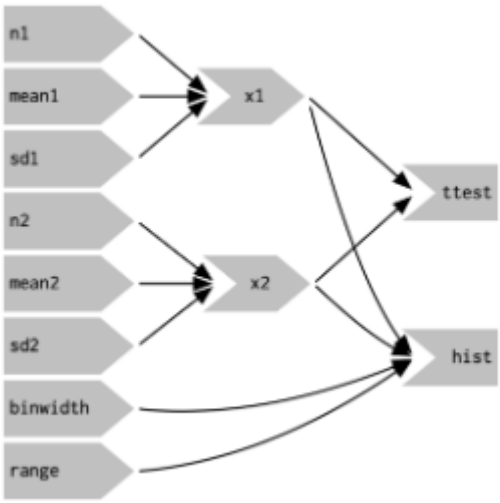
1. 因为太多的关系，导致这个app很难理解，app中没有任何代码块可以单独分析。
2. 这个app不高效，如果只是改变了这个直方图的一个参数，这个数据就会被重新计算，导致你的`x1`和`x2`就会更新。
   在更改直方图或者t检验的一些参数的时候，你肯定希望数据保持不改变，幸运的是可以使用下面的方法优化代码。

4.4.4 简化反应图

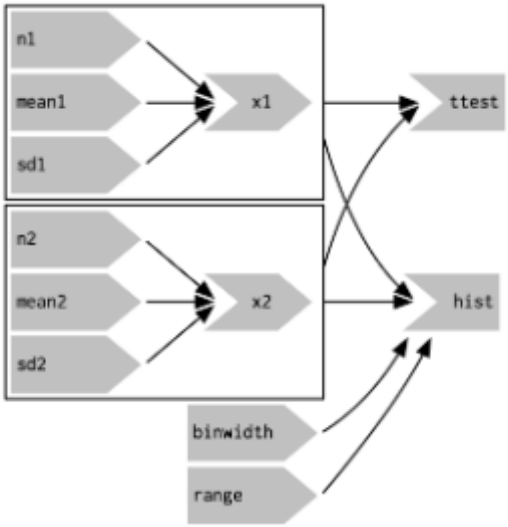
在下面的代码中，重构代码，将重复的代码提取出来作为新的反应表达式`x1`、`x2`，用来模型两个分布。使用`reactive()`将结果分配给变量。在后面使用这些表达式，可以像是使用函数一样调用他们。

```
server <- function(input, output, session) {
  x1 <- reactive(rnorm(input$n1, input$mean1, input$sd1))
  x2 <- reactive(rnorm(input$n2, input$mean2, input$sd2))

  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1(), x2())
  })
}
```

上面的代码的反应图如下图，这个流程图简化了代码而且更加容易理解，因为每个部分都是相互独立的。分布参数控制着`x1`、`x2`。这个改写后的代码减少了计算量，使得计算更加高效。。现在可以直接修改参数`binwidth`和`range`来控制直方图，而不再修改数据。



为了强调这种模块化，加入边框，模块化可以将重复的代码提取出来重用，同时保证了与应用程序中的其他功能隔离。对于更加复杂的技术，模块是极为有用且强大的工具。



要遵守编程里面的“事不过三”原则：当代码复制粘贴超过三次，就应该考虑减少重复（通常是写函数）。这个是非常重要的原则：使得代码更加容易阅读，减少代码，更加容易维护代码。

4.4.5 为什么需要反应表达式

当你第一次使用反应式代码的时候，你可能会问为什么需要反应表达式，为什么你不能利用现有的工具减少代码的重复：创建性的变量或者写函数？不幸的是这些在反应式环境都没有被用到。

如果你想使用一个变量来减少代码的重复，你可能将代码写成下面的这样：

```
server <- function(input, output, session) {
  x1 <- rnorm(input$n1, input$mean1, input$sd1)
  x2 <- rnorm(input$n2, input$mean2, input$sd2)

  output$hist <- renderPlot({
    histogram(x1, x2, binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1, x2)
  })
}
```

如果写成上面的代码，代码会报错，因为你试图从反应式目录外面获得输入变量，如果不是这样，依然会有错误，因为 `x1`、`x2` 只会运行一次，当会话开始的时候，而不是当输入更新的时候。
上面是写成变量的形式，下面是写成函数的形式：

```
server <- function(input, output, session) {
  x1 <- function() rnorm(input$n1, input$mean1, input$sd1)
  x2 <- function() rnorm(input$n2, input$mean2, input$sd2)

  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = input$binwidth, xlim = input$range)
  }, res = 96)

  output$ttest <- renderText({
    t_test(x1(), x2())
  })
}
```

写成函数的形式依然有错误：任何输入将造成所有的输出被重新计算。t检验和画直方图将会重新计算 `x1()`、`x2()`。但是在反应式编程中，将自动获得这些结果的缓存。并且只在输入发生变化的时候。

总结：将输入保存到变量里面，导致只是计算一次；将输入放到函数里面，导致函数在被调用的时候，数据会被重新计算。而反应式编程里面，只有当输入发生变化的时候，才会重新计算。

4.4.6 练习

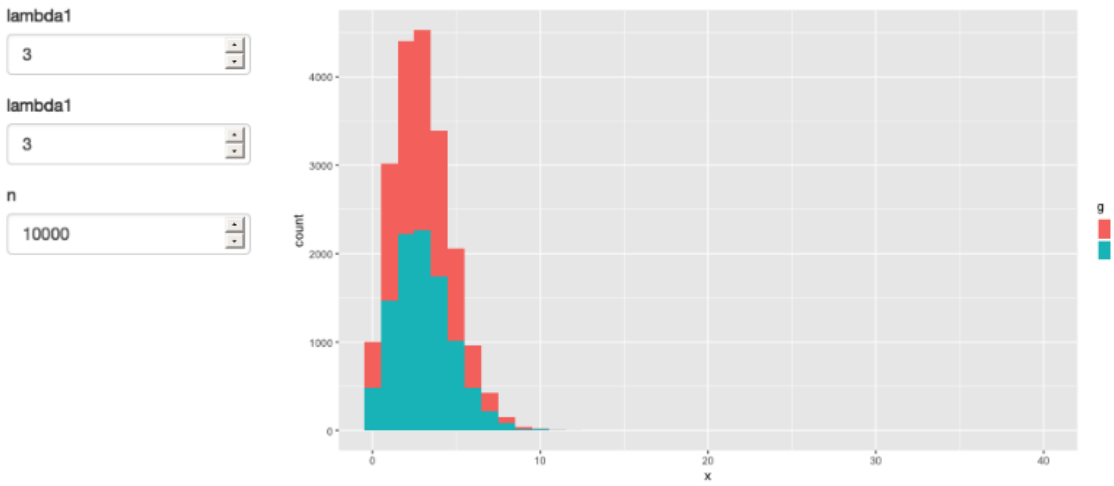
4.5 控制评估时间

现在你已经知道反应式编程的基础的内容。我们将讨论更加高级的内容：增加或者减少反应式编程执行的频率。这里只是简单的介绍，在未来的章节，将详细介绍这个技术。

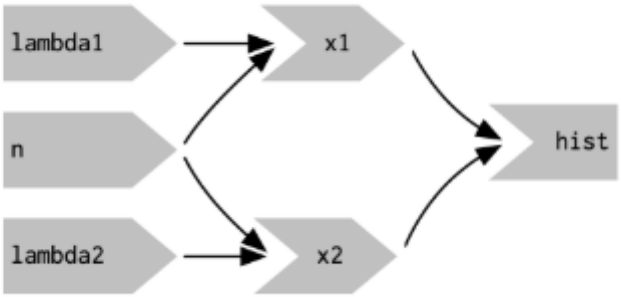
为了解释这个基础的原理，我将简化我的仿真app：每个分布只有一个独一无二的参数，还有一个共享参数：`n`。这样修改使得shiny app的代码更小。

```
ui <- fluidPage(
  fluidRow(
    column(3,
      numericInput("lambda1", label = "lambda1", value = 3),
      numericInput("lambda2", label = "lambda2", value = 3),
      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({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

这个shiny app是这样的：



这个shiny app的反应图是这样的：



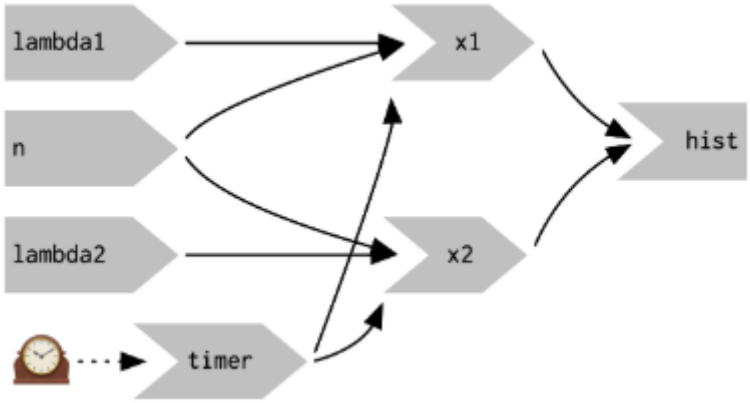
4.5.1 定时失效

想象一下你希望你希望改变这样的一个事实：不停的重新数据的仿真，来达到看的图是动态的图，而不是静态的图。因此，我们使用`reactiveTimer()`来增加更新频率。

`reactiveTimer()`是一个反应表达式，依赖于隐藏的输入，如果希望反应式更新的频率和默认的频率不一样，就可以使用这个函数。下面的代码中，将在一秒钟更新两次你的分布图。这样的数据足够您观察你的数据分布。而不会过快使得你眼花缭乱。(啥意思？就是说当你确定好`x1`和`x2`的参数之后，shiny app会每隔1/2秒就运行`x1`和`x2`这个反应式，虽然你只是修改了一次参数，但是这个反应式编程会自动的按照你指定的频率不断的刷新`x1`和`x2`的值)

```
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({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
```

```
}
```



注意，我们在反应式编程中使用`timer()`计算`x1()`和`x2()`的原理是：我们不断 的调用它，但是不是使用他的值，而是让`x1`和`x2`依赖于`timer`什么时候运行，而不担心`x1`和`x2`去确切的值是多少。

4.5.2 点击响应

在上面的情况中，如果仿真每隔1s运行会怎么样，上面的代码是1/2s更新。这导致shiny app需要不断的计算，如果有人疯狂的点击按钮，那么shiny app会创建大量的待办事项，导致shiny app可能无法响应别的事项而崩溃。这将给用户带来不良的用户体验。
如果你的计算花费的时间非常长（或者计算昂贵）你可能需要用户点击“开始计算”按钮来执行计算。这就需要用到`actionButton()`。

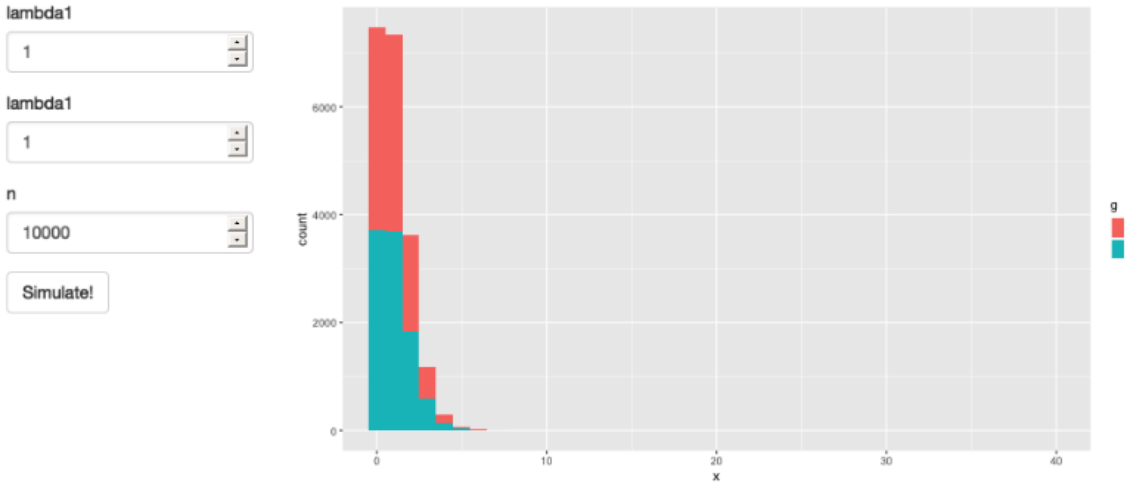
```
ui <- fluidPage(
  fluidRow(
    column(3,
      numericInput("lambda1", label = "lambda1", value = 3),
      numericInput("lambda2", label = "lambda2", value = 3),
      numericInput("n", label = "n", value = 1e4, min = 0),
      actionButton("simulate", "Simulate!")
    ),
    column(9, plotOutput("hist"))
  )
)
```

这个也叫依赖性反应：

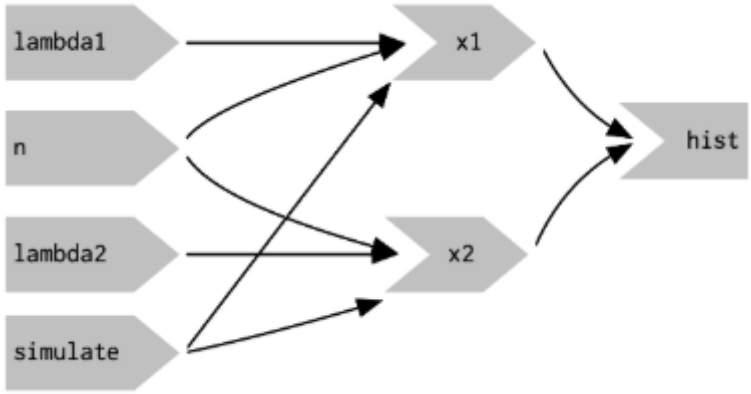
```
server <- function(input, output, session) {
  x1 <- reactive({
    input$simulate
    rpois(input$n, input$lambda1)
  })
  x2 <- reactive({
    input$simulate
    rpois(input$n, input$lambda2)
  })
  output$hist <- renderPlot({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

`x1()`和`x2()`需要点击`simulate!`这个按钮才会开始计算（依赖性）。

这个shiny app的外观如下：



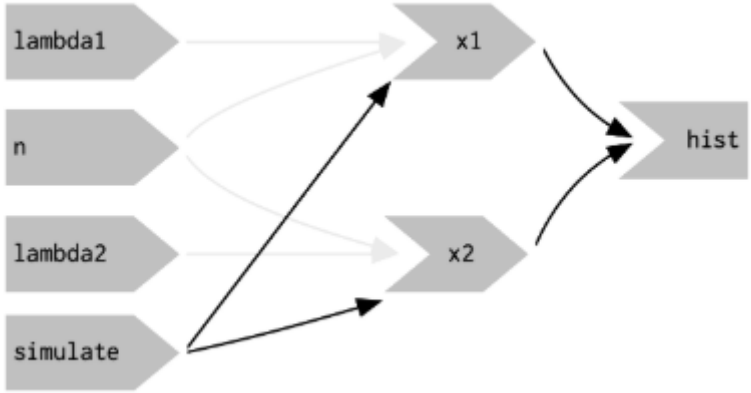
这个shiny app的反应图是：



如果只是更新`lambda1`、`n`、`lambda2`。这个计算不会开始，当点击`simulate!`后，才会开始计算。上面的写法没有达到我们的要求，当`lambda1`、`n`、`lambda2`更改的时候，也会重新计算，我们要替代现有的依赖。
为了解决这个问题，使用一个新的工具。直接使用值而不需要任何依赖。这个工具为`eventReactive()`。这个函数参数有两个部分，一个参数是调整依赖性的，一个参数是发送特定值去计算的。这允许app只在点击`simulate`时候开始计算`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({
    histogram(x1(), x2(), binwidth = 1, xlim = c(0, 40))
  }, res = 96)
}
```

这个代码的反应图如下：



这下`x1()`和`x2()`不会在改变`lambda1`、`lambda2`、`n`的时候触发计算。虽然`x1`和`x2`需要`lambda1`、`lambda2`、`n`的值，但是是否开始计算，不再依赖这几个值。

4.6 观察者

到目前位置，我们已经探索了shiny app的内在机制，但是有时候应该注意这个shiny app的外部和对这个世界产生的副作用。比如将文件保存到网络硬盘里、发送一个获得数据的api、更新数据库、或者最常见的就是将调试信息打印到控制台上。这些信息不会改变你的shiny app的外观，因此你无法使用`render()`函数来输出。相反你需要一个`observer`。

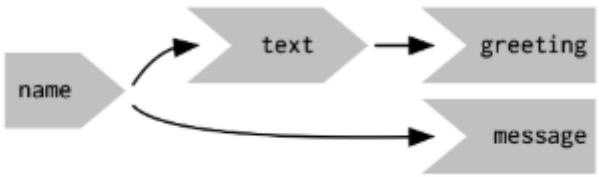
有很多方法创建一个监视器。将在以后章节详细介绍，现在我想展示`observeEvent()`。这个有助于新手的你调试程序。

`observeEvent()`是和`eventReactive()`非常相似的，有两个重要的参数，`eventExpr`和`handlerExpr`。第一个参数是输入或者依赖性的表达式。第二个参数是将要运行的代码。小案例：下面的代码在更新`input$name`的时候，会向控制台发生一条信息。

```
server <- function(input, output, session) {
  text <- reactive(paste0("Hello ", input$name, "!"))

  output$greeting <- renderText(text())
  observeEvent(input$name, {
    message("Greeting performed")
  })
}
```

`observeEvent()`和`evetReactive()`有两个重要的区别：

1. 不可将`observeEvent()`给任何变量赋值
2. 不能被别的反应式调用



观察者和输出是非常接近的，可以这样认为`output`是用来更新HTML部分的。而`Observers`是用来更新控制台的。

这一章介绍的反应式编程到此为止，下一章是一个使用shiny进行数据处理的实战。

内容可能会一直更新，可以查看我的微信公众号：pypi

获得最新的关于这部分的内容

微信扫一扫：

知乎：https://www.zhihu.com/people/fa-fa-1-94

csdn：https://blog.csdn.net/yuanzhoulvpi

github: https://github.com/yuanzhoulvpi2017/master_shiny_CN

如果有错误，欢迎指正，邮箱联系我：yuanzhoulvpi@outlook.com