第 1 章 Shiny

事实上，R 中存在一些构建 GUI 的包，例如 tcltk，RGtk2，Qt，gWidgets2 等等。通过这些包可以开发一个数据产品，用户通过用户界面去操纵这些数据产品，例如用户想要画图，通过鼠标点击，拖拽就可以绘制，而不需要编写 R 代码。

Shiny 也可以起到类似的作用。Shiny 是一个开源的 R 包，它为使用 R 构建 Web 应用提供了一个优雅有力的 Web 框架。使用 Shiny 不需要 R 用户了解前端，后端知识，需要 R 用户了解网页开发语言的知识，比如 javascript，能够帮助 R 用户高效的开发 web 程序。本章节会介绍 Shiny 的使用，介绍 Shiny 的基本原理，包括输入输出、页面布局、部件等内容。

Shiny 是一种非常高级的数据可视化方法，可以打造出非常令人影响深刻的内容。另外，Shiny 同样可是一个数据探索，成果沟通的工具。本章节会对 Shiny 的相关内容进行介绍。

1.1 Shiny 基础

首先，我们先介绍关于 Shiny 基础的内容，在这一部分，我们会了解什么是 Shiny，Shiny 程序的基本结构，Shiny 程序的运行原理等内容。

1.1.1Shiny 介绍

Shiny 是一个高级的数据可视化，数据展示的工具，他不仅仅能够展示数据，其还有更多的功能。Shiny 可以用于数据交互，数据分析，以及成果沟通。使用 Shiny 可以构建一个类似于小程序一样的东西，实现各种各样的功能。

这个例子中，使用 Shiny 构建了一个用于推荐的软件。需要注意的是，图中所显示的数据都经过了脱敏处理，并不是真实的数据。如图 1.1 所示。

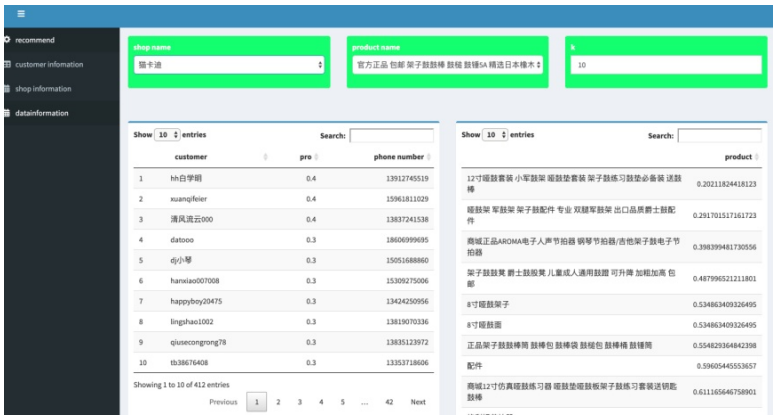

图 1.1　shiny 推荐系统

上面所展示的 Shiny 是一个推荐系统，需要再提醒一下，图中的数据都是经过了脱敏处理，不是真实的数据。从图中可以看到，这个 Shiny 有四个部分，第一个部分是推荐系统，第二个部分分析客户，第三个数据集店铺分析，第四部分会显示所使用的到的数据。

上面的的推荐系统中，可以选择不同的商店，选择不同的商品，然后还有一个调整推荐算法的参数。下面的两个表示输出结果。第一个表输出的内容是可能对所选商品感兴趣的人，越排在前面的人表示越有可能对商品感兴趣，这个结果是更具推荐算法计算出来。第二个表的结果是与所选商品比较相似的商品，商品顺序是通过相似度。

这样的话，可以找到每一个商品可能感兴趣的人，然后推荐给他们，并且可以尝试将相似的商品一同进行推荐。

第二个部分是分析客户，从客户的角度来进行的分析。如图 1.2 所示。

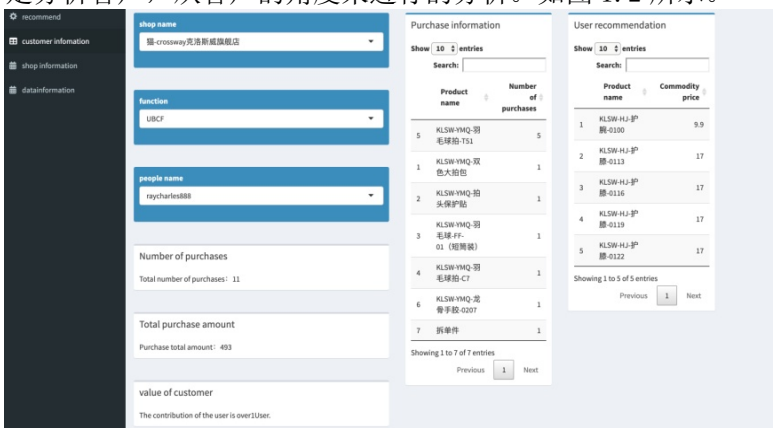

图 1.2　shiny 推荐系统

从图中可以看到，在这个 Shiny 界面轴，可以选择不同的商店，选择不同的推荐算法，以及选择不同的人。这个 Shiny 的目的是希望帮助商家分析自己的客户。从结果中可以看到，客户购买过多少次，花费了多少钱，并给出了这个客户价值。左边的第一张表显示了客户所买的东西，第二张表显示出可以推荐给这个客户的商品，推荐的结果同样是更具推荐算法给出来的。第三个部分是分析商店，如图 1.3 所示。

图 1.3　推荐系统

这个部分对于店铺进行可分析，在这个界面只有一个选项，可以选择不同的商店。在选择了商店之后，会显示出，这个商店有过多少客户，有多少商品、销售额等信息，左边的第一个表给出了店铺中比较有价值的客户（买的多，消费多），第二张表显示出了店铺中比较受欢迎的商品。

这个 Shiny 例子比较复杂，涉及到了很多的内容，但是这个是一个很好的展示。使用 shiny 可以快速的构建一个算法演示或者一个数据产品的 demo。例如，上面的 Shiny 所构建的一套推荐算法，再完善一下可以使用其他的编程工具做成一个产品。但是在构建产品之前，使用 Shiny 可以快速的实现，反馈，然后更改。因为 shiny 非常的简单，使用者不需要了解 web，不需要了解 javascript，只需要了解 R，即可以构建 Shiny。因此，Shiny 是一项非常有价值的技能。

在下面的内容中，会介绍有关 Shiny 的基本原理，构建 Shiny 程序。

1.2Shinyapp 的基本部分

什么是 shiny，Shiny 是一个开源的 R 包，它为使用 R 构建 Web 应用提供了一个优雅有力的 Web 框架。shiny 的特点是不需要 R 用户了解前端，后端知识。不需要 R 用户了解网页开发语言的知识，比如 javascript。为 R 用户量身打造，帮助 R 用户高效的开发 web 程序。

对于 R 用户而言，构建一个 Shiny 程序非常的简单，通常而言，Shiny 程序包含三个部分：

（1）shiny 原程序的组成部分，分为 ui 和 server。

（2）shiny 程序内有哪些元素：小部件，render 函数，input，output。

（3）input 和 output 之间的关系。

我们现来查看一个例子，然后分析其工作原理。首先通过下面的代码打开案例。结果如图 1.4 所示。

```
library(shiny)
runExample("01_hello")
```

图 1.4　shiny 例子

这个例子使用的是自带的 faithful 数据集合，通过改变 slider 滑块的值调整直方图的平滑程度(bin)。shiny 程序包含两个部分：UI 部分和 server 部分，简单来说，

（1）ui 部分就是我们需要设定界面是要怎么展示。

（2）server 部分就是后台的计算是如何进行。

然后我们来看一下上面这个程序的源代码，首先查看的是 UI 部分，ui 部分包括了页面的布局，还包括了页面存在哪些部件。

```
ui<-fluidPage(

#Apptitle----
titlePanel("HelloShiny!"),

#这里指定的是什么布局方式
sidebarLayout(

#Sidebarpanel
sidebarPanel(

#这里是定义一个 slider 作为输入
sliderInput(inputId="bins",
label="Numberofbins:",
min=1,
max=50,
value=30)

),

#主页面，这里主页面是一个绘图
mainPanel(

#Output:Histogram----
plotOutput(outputId="distPlot")
```

```
)
)
)
```

在上面的代码中，主要有三个部分，包括如下：

（1）sidebarLayout 表示边栏布局，用于指定页面是如何布局布局。这种布局是默认的页面布局方式，因为在 R 中创建一个 Shiny 默认就是这样的情况。此布局为输入提供了侧栏，sidebarPanel，为输出提供了大的主区域，mainPanel。当然还有很多其他的布局方式，包括，tabsetPanel()和navlistPanel()函数分割布局，还有一个专门用于Shiny布局的包，dashboard。在这里只介绍这种基础的布局方式。

（2）sliderInput 是一个部件，用于提供输入，也就是图 13.8 中的滑块。

（3）plotOutput(outputId=“distPlot”)是输出，输出一个图，对应的是图 13.8 中的图形。

上面的代码比较简单，主要内容就是这三个部分，首先，定义了布局方式，布局包括一个侧边栏和主区域，在侧变量定义了一个滑块，用于在交互中表示输出。在主区域有一个图形输出。然后是 Server 部分，Server 部分表示了滑块的调整，是如何影响图形的。也就是说，server 部分是定义我们 UI 部分展示的结果是如何计算的。下面的来看一下上面的例子的 server 代码部分：

```
server<-function(input,output){


output$distPlot<-renderPlot({

x<-faithful$waiting
bins<-seq(min(x),max(x),length.out=input$bins+1)

hist(x,breaks=bins,col="#75AADB",border="white",
xlab="Waitingtimetonexteruption(inmins)",
main="Histogramofwaitingtimes")

})

}
```

server 部分是一个函数，这个函数有两个参数，input 和 output，这个两个参数可以认为他们代表输入和输出。上面的代码中 output$distPlot 表示有一个输出，标签为 distPlot，这个标签的对应的是 UI 部分的输出图形的部分，UI 部分的代码为：plotOutput(outputId=“distPlot”)。renderPlot 表示要输出的内容为一副图形，如果是要输出文本，则需要使用 renderText()函数，不同的输出内容，需要使用同的 rend_函数。renderPlot 函数里面是一串用于绘图的函数，这里绘图的方式和在 R 中绘图是一样的，不同点是这里 input$bins 表示的是交互中的输入，对应 UI 部分的滑块，代码为：sliderInput(inputId="bins",label="Numberofbins:",min=1,max=50,value=30)。也就是说，滑块的 inputId 是 bins，当滑块滑动的时候，imput$bins 会记录滑块的结果，然

后再 server 部分绘图的时候，imput$bins 是作为绘图的参数，绘制的图形通过 output$distPlot 表示，进一步通过 plotOutput 函数进行展示。这样的话则构建了输入和输出的关系。

然后使用下面的代码，生成 app。

```
shinyApp(ui,server)
```

这样就构建了一个基础的 shiny 程序。总而言之，shiny 程序分为两个部分，Ui 和 Server

（1）ui 定义了页面的布局。

（2）ui 里面包含的元素包括，页面布局，部件，输出。

（3）server 定义了页面背后的计算。

server 中来来自页面的输入通过 input 进行表示，输出页面通过 output 表示。

上面的总结了关于 Shiny 的核心内容。下面来看第二个例子。

1.3Shiny 简单例子

这个例子展示了通过 verbatimTextOutput 函数直接打印 R 对象，以及使用 tableOutput 函数实现通过 HTML 表格显示数据。运行下列代码，打开这个例子，结果如图 1.5 所示。

```
library(shiny)
runExample("02_text")
```

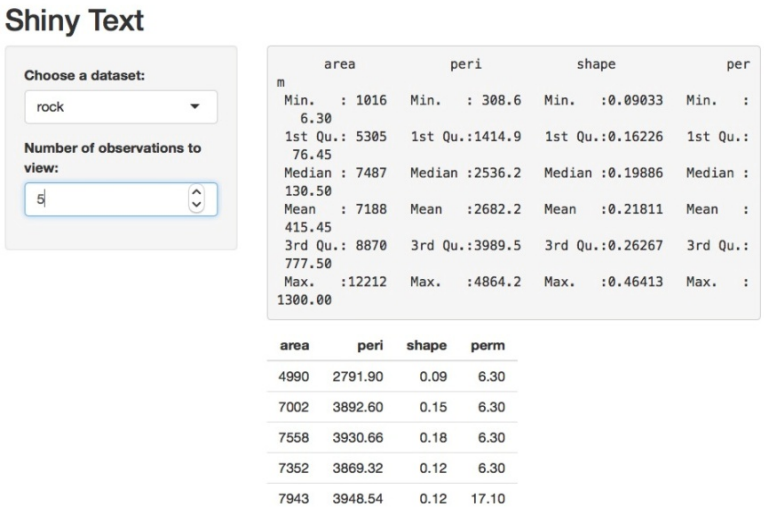

图 1.5　所示

从图 1.5 中可以看到，Shiny 程序有两个输入，用于选择数据集，以及选择显示数据集的行数。有两个输出，包括一个关于数据的统计信息，和数据集的原始数据。选择不同的数据集，则会显示出不同数据集的统计信息和数据集的原始数据。填写不同的行数则会显示不同行数的数据集。

我们来查看一下代码，首先查看 UI 部分：

```
#DefineUIfordatasetviewerapp----
ui<-fluidPage(

#Apptitle----
titlePanel("ShinyText"),#标题

#Sidebarlayoutwithainputandoutputdefinitions----
sidebarLayout(

#Sidebarpanelforinputs----
sidebarPanel(

#Input:Selectorforchoosingdataset----
selectInput(inputId="dataset",
label="Chooseadataset:",
choices=c("rock","pressure","cars")),#这是一个选择输入

#Input:Numericentryfornumberofobstoview----
numericInput(inputId="obs",
label="Numberofobservationstoview:",
value=10)#这是一个数字输入
),

#Mainpanelfordisplayingoutputs----
mainPanel(

#Output:Verbatimtextfordatasummary----
verbatimTextOutput("summary"),#这是一个输入

#Output:HTMLtablewithrequestednumberofobservations----
tableOutput("view")#这是一个表格输入

)
)
)
```

从代码中我们可以看到：

（1）页面布局是默认的布局，边栏布局（sidebarLayout）。

（2）ui 中有两个部件 selectInput 和 numericInput，分别表示一个选择框和数字输入的输入框。两个UI 部件 selectInput 和 numericInput，这两个部件的 ID 分别是 dataset 和 obs。

（3）有两个输出：verbatimTextOutput（“summary”），tableOutput（“view”），这两个输出对应的 ID 分别是“summary”和“view”。分别表示输出文本和输出一个数据表。

Server 部分描述了 Shiny 程序背后的计算，可以看下面的代码。

```
#Defineserverlogictosummarizeandviewselecteddataset----
```

```
server<-function(input,output){

#Returntherequesteddataset----
datasetInput<-reactive({#输入
switch(input$dataset,
"rock"=rock,#对应了不同的数据集合
"pressure"=pressure,
"cars"=cars)
})

#Generateasummaryofthedataset----
output$summary<-renderPrint({#输出一个 summary，这是一个 R 对象
dataset<-datasetInput()
summary(dataset)
})

#Showthefirst"n"observations----
output$view<-renderTable({#另外一个输出
head(datasetInput(),n=input$obs)
})

}
```

上面的代码中输入，输入有两个：

（1）ui 中 selectInput 部件对应的数据为 input$dataset。

（2）ui 中 numericInput 部件对应的数据为 input$obs。

关于输出，同样有两个：

（1）verbatimTextOutput(“summary”)，server 中对应的是 output$summary。

（2）tableOutput(“view”)，server 中对应的是 output$view。

输出通过 render 函数进行传递，举例而言，renderPrint 将 summary(dataset)的结果赋值给了 output$summary。renderTable 将 head(datasetInput(),n=input$obs)的结果为赋值给了 output$view

总而言之，ui 中的元素都存在一个 ID，server 通过$ID 进行提取对应的，例如，input$obs 或者 output$summary。而输出则通过 render 函数进行传递。把握住这几个关键点，则很容的构建 Shiny 程序。

1.1.4Shiny 小结

上面的内容介绍了如何实现一个 shiny 的 web 程序：

（1）程序分为两个部分，Ui 和 Server。

（2）ui 定义我们的网页是如何展示的，Server 定义后台是如何运算的。

下面对于 Shiny 的内容做了一些总结，下面总结了 Shiny 中输出内容与 render_函数之间的关系。如表 1.1 所示。

表 1.1 输入与输出

UI 部分	server 部分
dataTableOutput	renderDataTable
imageOutput	renderImage
plotOutput	renderPlot
verbatimTextOutput	renderPrint
tableOutput	renderTable
textOutput	renderText
uiOutput	renderUI
htmloutput	renderUI

从表中可以看到，如果需要输出一个数据表，则需要在 ui 中使用 dataTableOutput 函数，在 Server 中需要使用 renderDataTable 函数将结果传递到输出对应的 Id。

下面总结了 ui 中有哪些输入函数。

（1）actionButton：按钮。

（2）actionLink：链接。

（3）checkboxGroupInput：复选框。

（4）checkboxInput：复选框。

（5）dateInput：时间框。

（6）dateRangeInput：时间范围。

（7）fileInput：文件输入。

（8）numericInput：数字输入。

（9）passwordInput：密码输入。

（10）selectInput：选择框输入。

（1）selectizeInput:创建选择列表。

（12）sliderInput：滑动条输入。

（13）submitButton:提交按钮。

（14）textInput：文本输入。

上面的总了常用的一些输入部件，在需要使用的时候选择合适的部件进行构成程序。在了解了这些内容之后，可以开始子集构建一个 Shiny 程序。

1.1.5 构建 Shiny 程序

要制作一个 Shiny 程序，首先需要，打开 Rstudio，点击 file，创建一个 shinyapp。然后就可以开始编写 Shiny，在创建好之后，文件中会有一个框架，已经定义好了 Ui 部分和 Server 部分，设定好了默认布局，因此，只需要构建子集需要的输入输出的部件，然后编写 server 部分就可以了。

下面的代码构建了一个 Shiny 程序，使用的数据数据是，ggplot 里面自带的数据 diamonds，绘制不同变量的图形图，结果如图 1.6 所示。

```
#
#ThisisaShinywebapplication.Youcanruntheapplicationbyclicking
#the'RunApp'buttonabove.
#
#Findoutmoreaboutbuildingapplications withShinyhere:
#
#http://shiny.Rstudio.com/
#

library(shiny)
library(ggplot2)
#DefineUIforapplicationthatdrawsahistogram
ui<-fluidPage(

#Applicationtitle
titlePanel("Diamonddata"),

#Sidebarwithasliderinputfornumberofbins
sidebarLayout(
sidebarPanel(
selectInput(inputId="x",label='x',choices=names(diamonds)[c(1,5,6,7,8,9,10)],
selected=T),
selectInput(inputId="y",label='y',choices=names(diamonds)[c(1,5,6,7,8,9,10)],
selected=T)
),


#Showaplotofthegenerateddistribution
mainPanel(
plotOutput("diamondplot")
)
)
)

#Defineserverlogicrequiredtodrawahistogram
server<-function(input,output){
```

```
output$diamondplot<-renderPlot({
#generatebinsbasedoninput$binsfromui.R
plot(x=diamonds[[input$x]],y=diamonds[[input$y]],xlab='x',ylab='y')

})
}

#Runtheapplication
shinyApp(ui=ui,server=server)
```

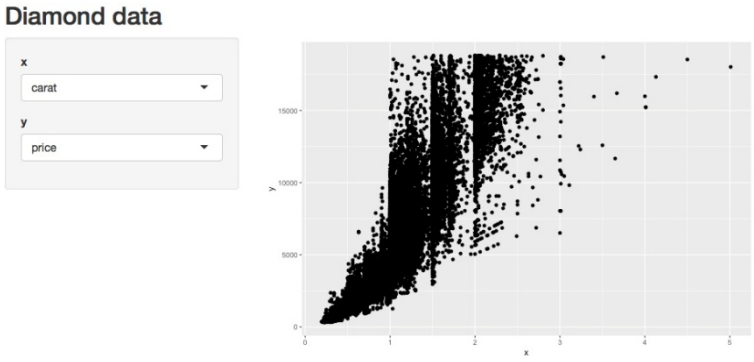

图 1.6　shiny

上面的代码有两个选择输入，分别表示选择什么变量来构建散点图。这种方法可以很好的对数据进行探索性可视化，只需要调整不同的变量，不需要另外的编写代码，就可以绘制不同变量的散点图。下面的代码介绍部署 Shiny，分享构建好的 Shiny 程序。

1.1.6shiny 部署

部署 Shiny 程序是指将构建好的 Shiny 程序部署到服务器上面，这样别人可以浏览器输入相应的连接来访问构建好的 Shiny 程序。如果希望部署 Shiny，有几种方式：

（1）Shinyapps.io。

（2）Shinyserver。

（3）ShinyServerPro。

（4）RstudioConnect。

这里介绍使用 Shinyapps.io 进行 shiny 部署，为什么选择这个呢，因为使用这种方式进行部署是免费的的，当然 Shinyapps.io 有付费的服务，但那是其也有免费的服务，而其他几种方式都是需要付费的。选择 Shinyapps.io 将应用程序部署到 Web，不需要自己的服务器，只需要简单的代码，就可以轻松的部署。想要部署，首先要注册 Shinyapps.io。有 github 账户也可以使用 github 账户进行注册。注册好之后在 R 中下载 rsconnect 包，然后加载包，代码如下：

```
install.packages('rsconnect')
library(rsconnect)
```

然后需要登陆：http://www.shinyapps.io/，账号密码就是申请号的账号密码。需要进行 shiny 配置，Shinyapps.io 会自动生成令牌和密钥，rsconnect 程序包可使用令牌和密钥来访问的帐户。如图 1.7 和图 1.8 所示。

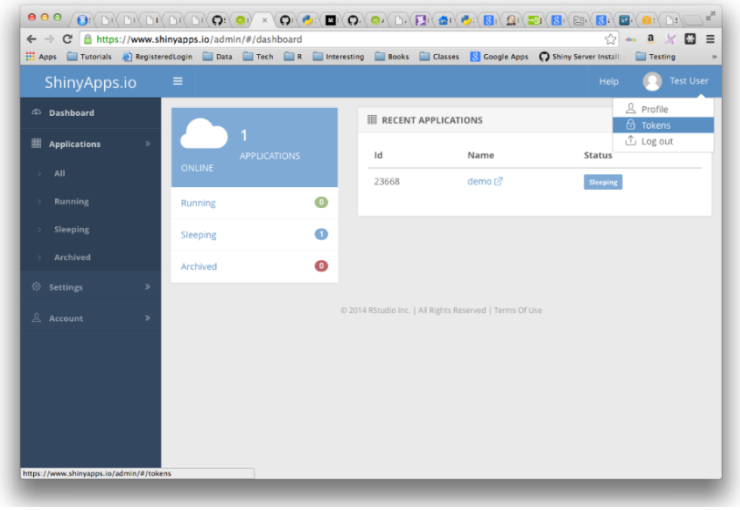

图 1.7　shiny 配置

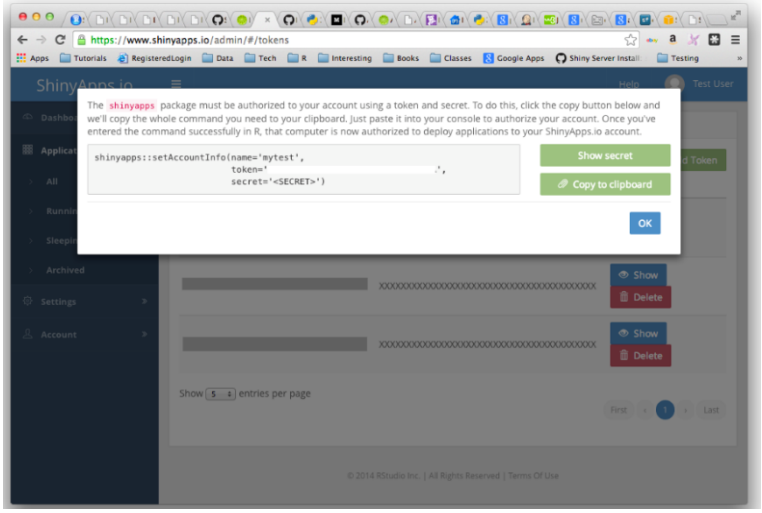

图 1.8　shiny 配置

在图 1.6 中，点击 Tokens，就会出现对应的密匙，如图 1.7 所示。然后复制图 1.6 中的代码，在 Rstudio 中运行。即可完成配置。完成配置之后，编写 shiny 程序。一旦完成了 shiny 程序的构建，就可以看是进行部署，部署 Shiny 的代码如下：

```
library(rsconnect)
deployApp()
```

运行代码之后，如果没有报错，则完成了 Shiny 的部署，任何人可以通过对应的连接来访问构建好的 Shiny 程序。

最后，本章节介绍了关于 Shiny 的核心的一些内容，阅读完本章节之后，就可以快速的构建一个 Shiny 程序。但是需要注意的是，还有很多内容本章节没有提到，比如，更多的布局方式，不如各种部件的实际应用。Shiny 可以快速的去构建一个项目的实例，可以很好的展示结果数据结果。另外，Shiny 同样可以去开发一个成熟的数据产品，然后通过部署的方式进行应用，这对于需要处理数据科学相关工作的人来来说是非常有价值的。

1.2 reactive 与 isolate

在上一部分，大家已经了解到了关于构建 Shiny 程序的核心内容，大家应该可以自己构建一个 Shiny 程序。大家已经了解到了 Shiny 程序的结果，Shiny 程序包含 UI 部分和 Server 部分，其中 UI 部分定义了 Shiny 程序的页面布局，Server 部分定义了 Shiny 程序的计算逻辑。在 UI 部分中，包含了页面的布局，Shiny 程序的部件，Shiny 程序的输出。关于页面布局等内容，会在下文介绍。Shiny 程序的输入输出，通过 Server 部分中的两个函数参数表示，这两个参数分别是 input 和 output。

这些内容都是关于 Shiny 程序的核心内容，了解了这些内容可以非常容易的构建基础的Shiny程序。然后在这个章节，我们介绍一些更加高级的内容，以构建更加高级的Shiny程序。

1.2.1reactive 函数

我们之前介绍的例子中，Shiny 程序的计算部分都是比较简单的，如果大量的慢速计算（例如读取网络中的股票数据），就需要使用到 Shiny 中的 reactive 函数。通过这个函数，可以控制应用程序的哪些部分在何时更新，以防止不必要的计算而降低应用程序的速度。

我们来看一个新的 Shiny 程序，通过股票代码查找股票价格，并将结果显示为折线图。这个 Shiny 程序可以通过股票代码获取股票数据，并且可以筛选股票的时间。程序的界面如图 1.9 所示。

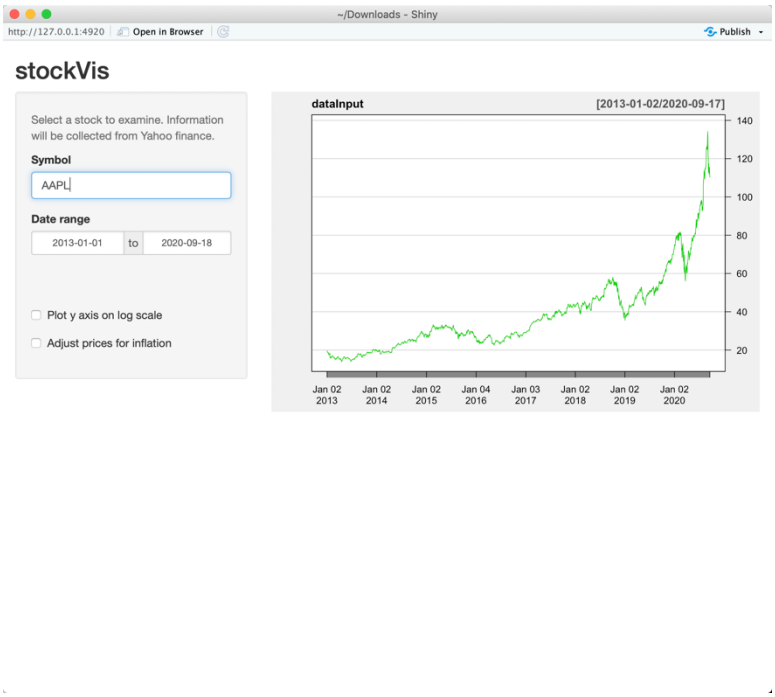

图 1.9　Shiny 程序的界面

图 1.9 中就是这个 Shiny 程序的界面，图 1.9 中显示出了苹果公司的股票数据。程序的代码如下所示。

```
#Loadpackages----
library(shiny)
library(quantmod)

#Sourcehelpers----
source("helpers.R")

#Userinterface----
ui<-fluidPage(
titlePanel("stockVis"),

sidebarLayout(
sidebarPanel(
helpText("Selectastocktoexamine.

InformationwillbecollectedfromYahoofinance."),
textInput("symb","Symbol","SPY"),

dateRangeInput("dates",
"Daterange",
start="2013-01-01",
end=as.character(Sys.Date())),
```

```
br(),
br(),

checkboxInput("log","Plotyaxisonlogscale",
value=FALSE),

checkboxInput("adjust",
"Adjustpricesforinflation",value=FALSE)
),

mainPanel(plotOutput("plot"))
)
)

#Serverlogic
server<-function(input,output){

dataInput<-reactive({
getSymbols(input$symb,src="yahoo",
from=input$dates[1],
to=input$dates[2],
auto.assign=FALSE)
})

output$plot<-renderPlot({

chartSeries(dataInput(),theme=chartTheme("white"),
type="line",log.scale=input$log,TA=NULL)
})

}

#Runtheapp
shinyApp(ui,server)
```

我们来看关于 Shiny 程序的代码，需要注意的是，这个 Shiny 程序还是用了一个金融方面的包，quantmod。这个包可以非常方便的获取金融数据，对金融数据计算相关指标以及对金融数据进行可视化。

这个程序还使用到了另外一个程序，代码source("helpers.R")表示加载helpers.R这个文件的代码，这个代码用于根据通货膨胀调整股票价格。

在 Shiny 程序中的 UI 部分，使用的小部件有 dateRangeInput，这个部件是一个时间范围的选择器。另外一个部件是 checkboxInput，表示复选框。

使用的布局依然是默认布局，程序中 br()表示换行。

在 Shiny 程序的 Server 部分，使用了 reactive 这个函数。大家可能会疑惑这个函数有什么作用，以及为什么需要使用这个。假设假设我们不使用 reactive 这个函数，那么代

码会变成如下所示。

```
output$plot<-renderPlot({
data<-getSymbols(input$symb,src="yahoo",
from=input$dates[1],
to=input$dates[2],
auto.assign=FALSE)

chartSeries(data,theme=chartTheme("white"),
type="line",log.scale=input$log,TA=NULL)
})
```

我们点击了复选框，对坐标轴进行取对数。这个时候，代码中的 input$log 会发生改变，这会导致表达式 renderPlot 会重新计算。然而，每次 renderPlot 计算，都会导致 getSymbols 函数的运行（这个函数用于获取股票数据）。这会导致程序运行速度变慢，并且过于频繁的获取数据或导致服务器拒绝我们的访问。

因此需要使用 reactive 函数，reactive 函数可以获取结果并且进行缓存，并且判断这个结果是否需要更新。例如上面这个例子中，如果使用了 reactive 函数，那么点击复选框，程序不需要重新获取股票数据，即可完成更新。

关于 reactive 函数，做一个简单的总结。

（1）reactive 会计算第一次运行时候的结果并且将结果保存下来。

（2）下次再调用这个结果的时候，会根据其依赖的部件来判断结果是否需要更新。

（3）如果需要更新就重新计算。

例如，上面的 Shiny 程序中，修改股票代码，那么 reactive 函数的结果就会更新。最后需要提的一点是，使用 reactive 函数的结果需要加上括号，对应上文的代码是 dataInput()。

1.2.2isolate 函数

需要注意的是，还有另外一种情况，就是程序有些参数不需要随着输入的变化而变化，例如我们有一个 Shiny 程序，Shiny 程序有一个按钮，只有点击按钮的时候可以获取当前数据集的图形，当时调整其他参数图形并不会发生改变。当遇到这种情况的时候，就需要使用到 isolate 函数。如果不使用 isolate 函数，调整任意的参数，只要和图形有关，都会刷新图形。

我们来看一个例子，代码如下所示。

```
library(shiny)

ui<-pageWithSidebar(
headerPanel("Clickthebutton"),
sidebarPanel(
sliderInput("obs","Numberofobservations:",
min=0,max=1000,value=500),
actionButton("goButton","Go!")
```

```
),
mainPanel(
plotOutput("distPlot")
)
)

server<-function(input,output){
output$distPlot<-renderPlot({

#Takeadependencyoninput$goButton
input$goButton

#Useisolate()toavoiddependencyoninput$obs
dist<-isolate(rnorm(input$obs))
hist(dist)
})
}

#Runtheapp
shinyApp(ui,server)
```

运行这个代码，输出结果如图 1.10 所示。

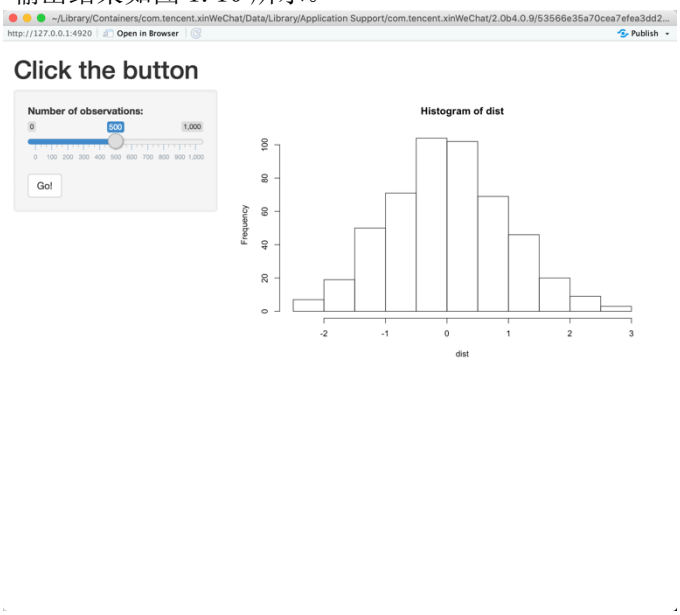

图 1.10　Shiny 程序

如果只是移动滑块，图形是不会产生任何改变的，只有点击了按钮 Go 之后，图形才会发生改变。接下来我们观察一下代码。

UI 部分比较简单，采用的是默认布局，有两个部件，分别是 sliderInput 和 actionButton。UI 中还有一个图形输出。

在 Server 部分，isolate(rnorm(input$obs))代码是关键，这行代码表示隔离 obs 这

个参数，也就表示，如果仅仅 input$obs 发生改变，renderPlot 函数是不会运行的，这也意味着在这个例子中，移动滑块，Shiny 程序不会发生变化。

1.3 Shiny 布局

我们已经了解到了，Shiny 程序的 UI 中，主要有几个部分，页面布局，小部件，输出。

之前的例子中，所使用的布局全部都是默认布局，其中 fluidPage 这个函数设置了 shiny 所需要的所有的 HTML，CSS 和 JavaScript，除了 fluidPage 函数，相似的函数还有 fixedPage 和 fillPage。这两个函数与 fluidPage 类似，不同点在于 fixedPage 具有固定的最大宽度，而 fillPage 则会填充浏览器的整个高度。

在这个章节中我们进一步了解 Shiny 中的布局方式。

1.3.1 侧边栏布局

构建 Shiny 程序默认的布局方式就是侧边栏布局，这种布局方式提供了一个用于输入的侧边栏和一个用于输出的主区域。这种布局的结构如图 1.1 所示：

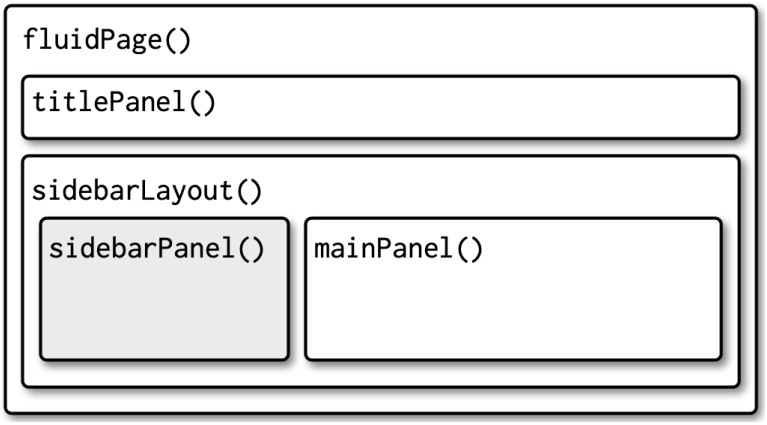

图 1.1　侧边栏布局

侧变量布局的代码结构如下所示。

```
sidebarLayout(

sidebarPanel(
#Inputsexcludedforbrevity
),
mainPanel(
#Outputsexcludedforbrevity
)
)
```

通常而言，侧边栏在左边，如果希望侧边栏在右边，可以添加 position 参数，参考如

下代码。

```
sidebarLayout(position="right",

sidebarPanel(
#Inputsexcludedforbrevity
),
mainPanel(
#Outputsexcludedforbrevity
)
)
```

大多数情况下，这种侧边栏布局已经能够满足构建 Shiny 程序的需要了。

1.3.2 网格布局

网格布局不像侧边栏布局一样，将页面划分成为两个部分。网格布局是将页面划分成为很多的网格。column()函数用于指定 Shiny 程序的某部分占据多少列，创建行则需要使用 fluidRow()函数。网格布局的结构如图 1.12 所示：

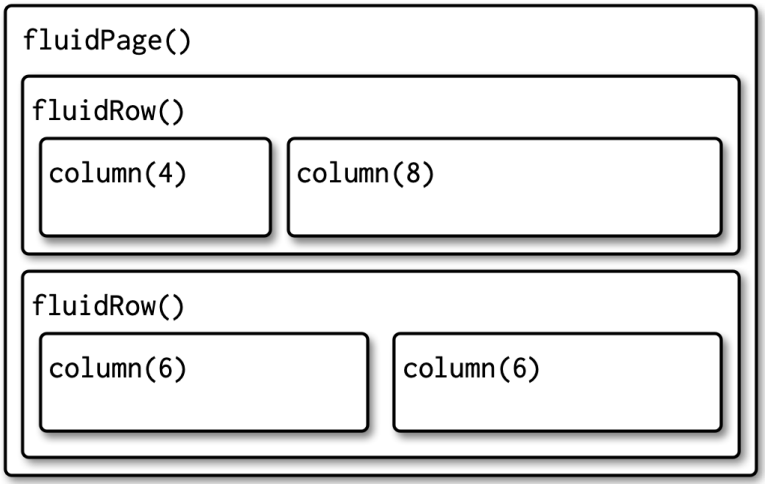

图 1.12　网格布局

我们看一个简单的例子。

```
library(shiny)
ui<-fluidPage(


titlePanel("网格布局"),
fluidRow(
column(4,
sliderInput("obs","Numberofobservations:",
min=0,max=1000,value=500),
actionButton("goButton","Go!")
```

```
),
column(8,
plotOutput("distPlot")
)

)
)
server<-function(input,output){
output$distPlot<-renderPlot({

#Takeadependencyoninput$goButton
input$goButton

#Useisolate()toavoiddependencyoninput$obs
dist<-isolate(rnorm(input$obs))
hist(dist)
})
}

#Runtheapp
shinyApp(ui,server)
```

shiny 程序的界面如图 1.13 所示。

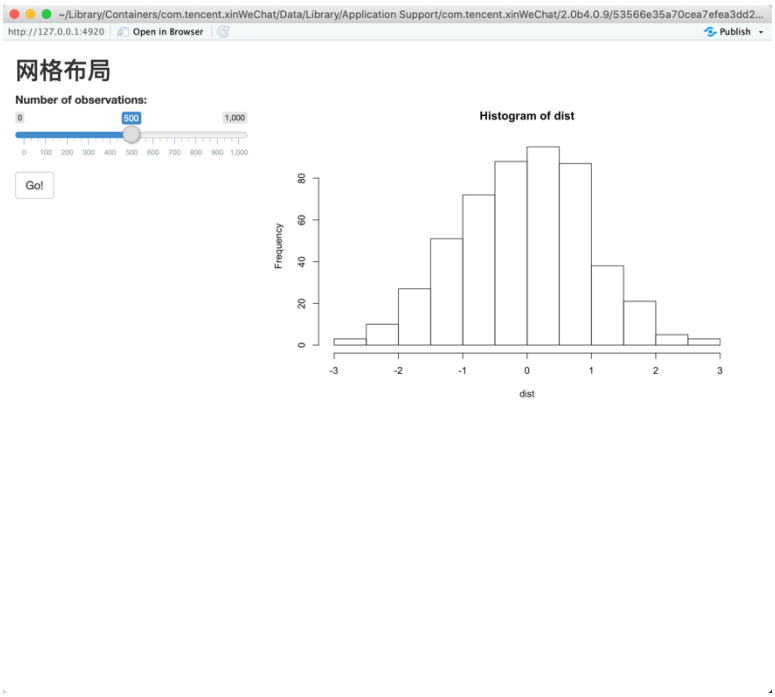

图 1.13　网格布局

在这个例子中通过网格布局模拟了侧边栏布局，我们来看代码，网格布局需要使用两个函数，fluidRow()函数和 column()函数。fluidRow()函数用于创建一行，column（）函数用于划分列。上面的代码中创建了两列，第一列宽为 4，第二例宽为 8.需要注意的是，

整个页面的宽度为 12。

也可以偏移列的位置，以实现对 UI 元素位置的更精确控制。可以通过向 column()函数添加 offset 参数来将列向右移动。

另外，fluidRow()和 column（）之间是可以嵌套的，也可以创建多个行，我们来看另外一个例子，代码如下所示。

```
library(shiny)


ui<-fluidPage(


titlePanel("网格布局"),
fluidRow(
column(4,
sliderInput("obs","Numberofobservations:",
min=0,max=1000,value=500),
actionButton("goButton","Go!")
),
column(8,
plotOutput("distPlot")
)

),

fluidRow(
column(4,
sliderInput("obs","Numberofobservations:",
min=0,max=1000,value=500),
actionButton("goButton","Go!")
),
column(4,offset=2,
sliderInput("obs","Numberofobservations:",
min=0,max=1000,value=500),
actionButton("goButton","Go!")
)

)
)
server<-function(input,output){
output$distPlot<-renderPlot({

#Takeadependencyoninput$goButton
input$goButton

#Useisolate()toavoiddependencyoninput$obs
```

```
dist<-(rnorm(input$obs))
hist(dist)
})
}

#Runtheapp
shinyApp(ui,server)
```

Shiny 程序的输出结果如图 1.14 所示。

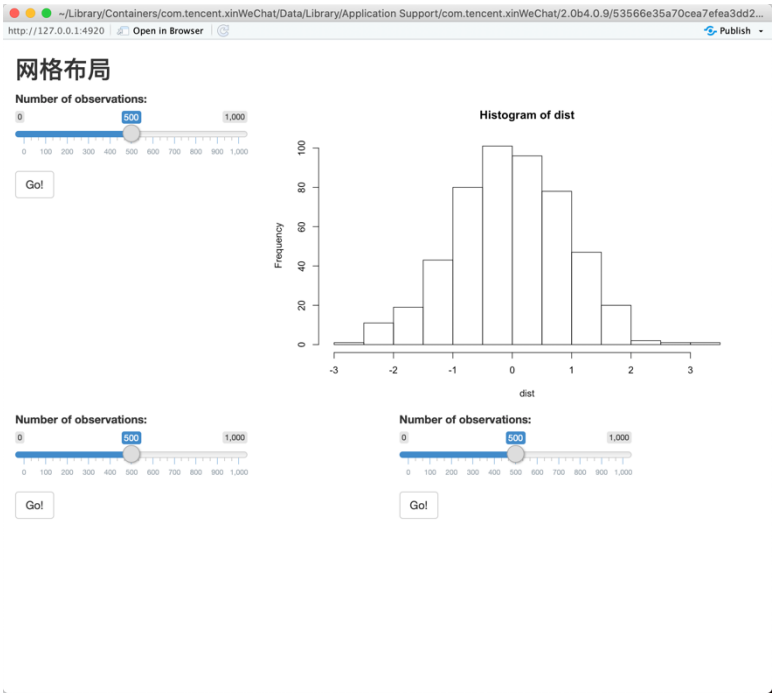

图 1.14　网格布局

在这个例子中，使用了两个 fluidRow()函数，在第二行中，有两列，每一列都是 4 行。代码”column(4,offset=2,…”表示将这一行向右偏移两个单位。从图中可以看到，第二行的两个小部件之间产生了一定的间隔。

网格布局就是将整个页面划分成为不同大小的网格，并且划分的方式完全可以自定义，因此，这种布局方式更加的灵活。

1.3.3 界面细分

有的时候，需要将页面划分成为不同的部分，以展示不同的结果。最常用的工具之一就是选项卡。对应的函数是 tabsetPanel 函数和 tabPanel 函数来完成。我们通过一个简单的例子来了解其使用。

```
library(shiny)
ui<-fluidPage(

titlePanel("Tabsets"),
```

```
sidebarLayout(

sidebarPanel(
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
actionButton(inputId="goButton",label="运行")
),

mainPanel(
tabsetPanel(
tabPanel("Plot",plotOutput("distPlot")),
tabPanel("Summary",verbatimTextOutput("summary")),
tabPanel("Table",tableOutput("table"))
)
)
)
)
server<-function(input,output){
output$distPlot<-renderPlot({

#Takeadependencyoninput$goButton
input$goButton

#Useisolate()toavoiddependencyoninput$obs
dist<-(rnorm(input$obs))
hist(dist)
})

output$summary<-renderPrint({
summary((rnorm(input$obs)))
})

output$table<-renderTable({
table((rnorm(input$obs)))
})
}

#Runtheapp
shinyApp(ui,server)
```

Shiny 程序的输出结果如图 1.15 所示。

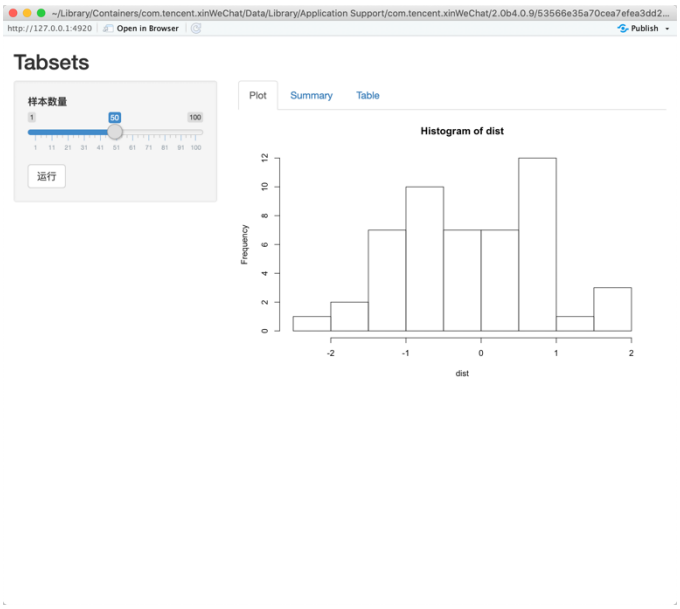

图 1.15　选项卡

从图中可以看到，在主页面中，有三个选项，分别是 Plot，Summary 和 Table。如果选择不同的选项，就会出现不同的界面。

然后我们来看下实现的代码，关键代码如下所示。

```
mainPanel(
tabsetPanel(
tabPanel("Plot",plotOutput("distPlot")),
tabPanel("Summary",verbatimTextOutput("summary")),
tabPanel("Table",tableOutput("table"))
)
)
```

可以看到，在这一部分代码中，首先使用了 tabsetPanel 函数，表示要创建一个选项卡。然后使用 tabPanel 函数，表示创造一个选项。

另外，选项卡可以位于选项卡内容的上方（默认），下方，左侧或右侧，设置 position 参数即可。例如想要将选项卡放在左边，则需要这么设置代码。

```
tabsetPanel(position="left".....,
```

使用这个功能，就可以创建更加细致的 Shiny 程序。另外，还有另外一种实现于选项卡类似的功能，也就是导航列表，其对应的函数是 navlistPanel()。同样，通过一个简单的例子来了解导航列表的实现。

```
library(shiny)
ui<-fluidPage(

titlePanel("列表导航"),

navlistPanel(
"HeaderA",
```

```
tabPanel(sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
plotOutput("distPlot")),
tabPanel("Component2"),
"HeaderB",
tabPanel(sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
verbatimTextOutput("summary")),
tabPanel("Component4"),
"-----",
tabPanel(sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
tableOutput("table"))
)
)
server<-function(input,output){
output$distPlot<-renderPlot({

#Takeadependencyoninput$goButton
input$goButton

#Useisolate()toavoiddependencyoninput$obs
dist<-(rnorm(input$obs))
hist(dist)
})

output$summary<-renderPrint({
summary((rnorm(input$obs)))
})

output$table<-renderTable({
table((rnorm(input$obs)))
})
}

#Runtheapp
shinyApp(ui,server)
```

Shiny 程序的运行结果如图 1.16 所示。

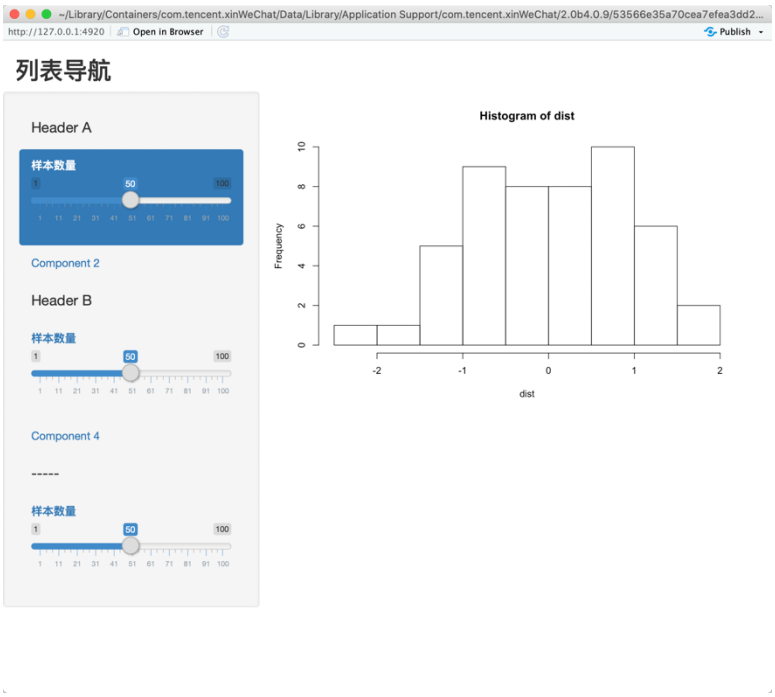

图 1.16　列表导航

从图中可以观察到，Shiny 程序的左边出现了很多的选项，通过点击不同的选项，Shiny 程序的右边就会出现不同的结果。

我们再来看下代码，关键的代码部分如下所示。

```
navlistPanel(
"HeaderA",
tabPanel(sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
plotOutput("distPlot")),
tabPanel("Component2"),
"HeaderB",
tabPanel(sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
verbatimTextOutput("summary")),
tabPanel("Component4"),
"-----",
tabPanel(sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
tableOutput("table"))
)
```

关键的函数是 navlistPanel 函数，使用这个函数表示创建一个列表导航。代码中"HeaderA","HeaderB"和"-----",是选项的标题。通过标题可以将选项进行划分。不同选项的创建依然是使用 tabPanel 函数。

第三种要介绍的是导航栏页面，使用的是 navbarPage()函数。我们来看一个简单的例子。

```
library(shiny)
```

```
ui<-navbarPage("导航栏页面",
tabPanel("Component1",
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
plotOutput("distPlot")),
tabPanel("Component2",
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
verbatimTextOutput("summary")),
tabPanel("Component3",
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
tableOutput("table"))
)
server<-function(input,output){
output$distPlot<-renderPlot({

#Takeadependencyoninput$goButton
input$goButton

#Useisolate()toavoiddependencyoninput$obs
dist<-(rnorm(input$obs))
hist(dist)
})

output$summary<-renderPrint({
summary((rnorm(input$obs)))
})

output$table<-renderTable({
table((rnorm(input$obs)))
})
}
#运行 APP
shinyApp(ui,server)
```

Shiny 程序的界面如图 1.17 所示。

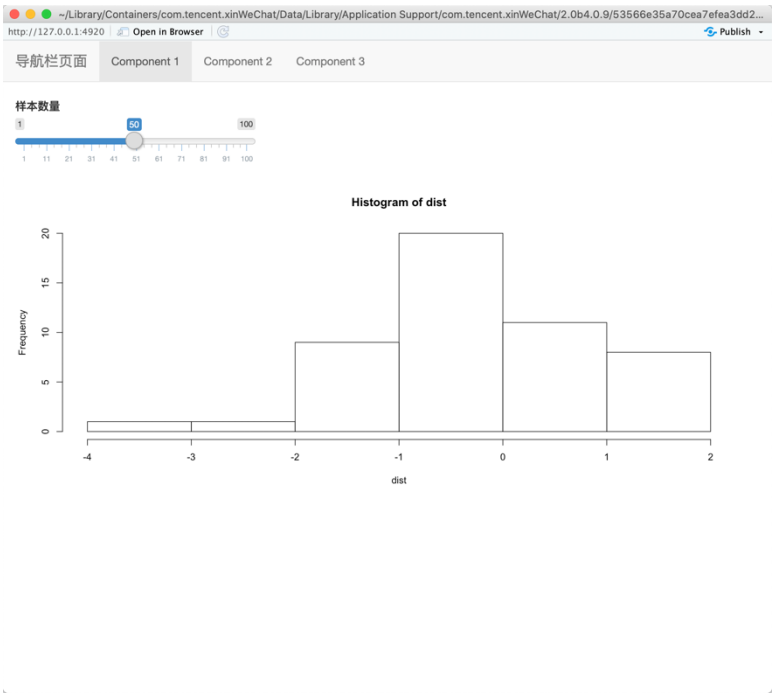

图 1.17　导航栏页面

从图中可以看到，Shiny 界面的上面出现了三个选项，点击不同的选项则会出现不同的界面。

我们来看一下代码，关键代码如下。

```
ui<-navbarPage("导航栏页面",
tabPanel("Component1",
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
plotOutput("distPlot")),
tabPanel("Component2",
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
verbatimTextOutput("summary")),
tabPanel("Component3",
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
tableOutput("table"))
)
```

关键函数是navbarPage函数，表示创建导航栏，不同的导航栏结果同样通过tabPanel函数创建。

另外还可以通过navbarMenu函数添加二级导航，我们将上面例子的代码稍微修改一下。

```
library(shiny)
ui<-navbarPage("导航栏页面",
tabPanel("Component1",
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
plotOutput("distPlot")),
tabPanel("Component2",
```

```
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
verbatimTextOutput("summary")),
tabPanel("Component3",
sliderInput(inputId="obs",label="样本数量",min=1,max=100,value=50),
tableOutput("table")),
navbarMenu("More",
tabPanel("Sub-ComponentA"),
tabPanel("Sub-ComponentB"))
)
server<-function(input,output){
output$distPlot<-renderPlot({

#Takeadependencyoninput$goButton
input$goButton

#Useisolate()toavoiddependencyoninput$obs
dist<-(rnorm(input$obs))
hist(dist)
})

output$summary<-renderPrint({
summary((rnorm(input$obs)))
})

output$table<-renderTable({
table((rnorm(input$obs)))
})
}

#运行 APP
shinyApp(ui,server)
```

Shiny 程序的界面如图 1.18 所示。

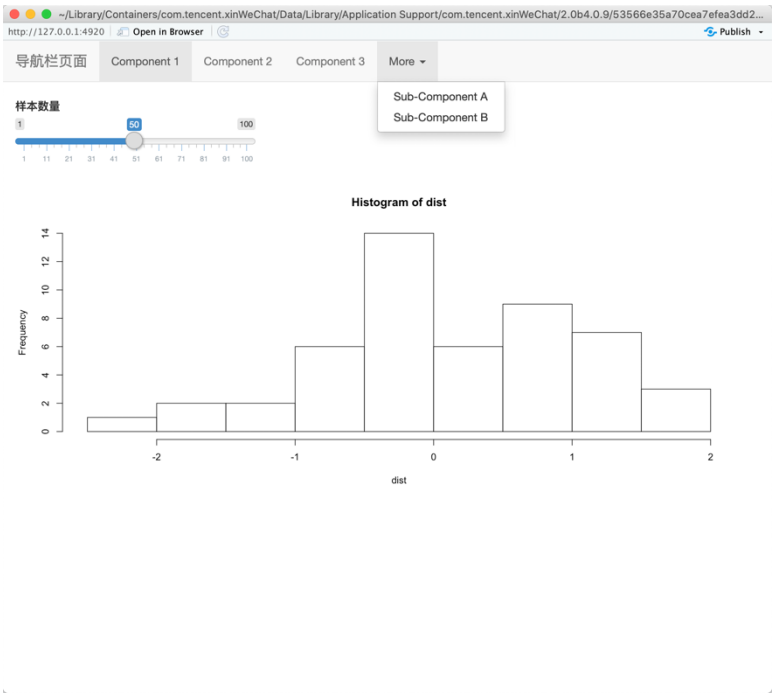

图 1.18　二级导航

从图中可以看到，最后一个选项中，出现了二级导航。对应的代码部分如下所示。

```
navbarmenu("more",
tabpanel("sub-componenta"),
tabpanel("sub-componentb"))
```

以上就是关于 Shiny 布局相关的内容，通过这些内容的学习，大家可以很好的掌握 Shiny 这项工具，能够将 Shiny 应用起来解决遇到的一些问题。

但是依然还有很多与 Shiny 相关的内容这里没有介绍到。例如直接使用 HTML 语言构建 UI，例如 Shiny 的一些扩展包，包括 shinydashboard。关于更多的 Shiny 内容，就需要大家自己去进一步了解与探索了。

1.4 Shiny 拓展

到目前为止，我们已经了解了构建 Shiny 的基本结构，了解了布局等等内容，在本章节我们会介绍额外的一些内容，包括鼠标事件步骤，上传与下载，用户反馈和动态 ui 等内容。

1.4.1outplot 函数

在上文中，我们已经接触到了 Shiny 中的绘图。我们知道了使用 renderPlot 函数来传递一幅图形，使用 plotOutput 函数在 ui 中展示一幅图形。在本小节中，我们将讨论更多

有关绘图的内容，例如响应鼠标事件。

在 Shiny 中，有四种不同类型的鼠标事件，点击（click），双击（dbclick），悬停（hover）和矩阵选择（brush）。想要将这些事件转换成为输入，则需要在 plotOutput 函数中使用新的参数，例如想要捕获点击事件，则需要使用 click 参数，代码可以是’ plotOutput("plot",click="myclick")。如果想好捕获 hover 事件，代码则是’ plotOutput("plot",hover="my_hover")。其他类型的事件也是同理。

我们来看一个关于捕获鼠标点击事件的一个事例，代码如下所示。

```
library(shiny)
library(shinythemes)

#DefineUIforapplicationthatdrawsahistogram
ui<-fluidPage(
theme=shinytheme("darkly"),
#Applicationtitle
titlePanel("点击事件"),

#Sidebarwithasliderinputfornumberofbins
sidebarLayout(

sidebarPanel(
verbatimTextOutput("info")

),

#Showaplotofthegenerateddistribution
mainPanel(
plotOutput("distPlot",click="myclick")

)
)
)

#Defineserverlogicrequiredtodrawahistogram
server<-function(input,output){

output$distPlot<-renderPlot({
#generatebinsbasedoninput$binsfromui.R
plot(iris$Sepal.Length,iris$Sepal.Width)

})

output$info<-renderPrint({
req(input$myclick)
x<-round(input$myclick$x,2)
y<-round(input$myclick$y,2)
```

```
cat("[",x,",",y,"]",sep="")
})
}

#Runtheapplication
shinyApp(ui=ui,server=server)
```

shiny 程序的输出结果如图 1.19 所示：

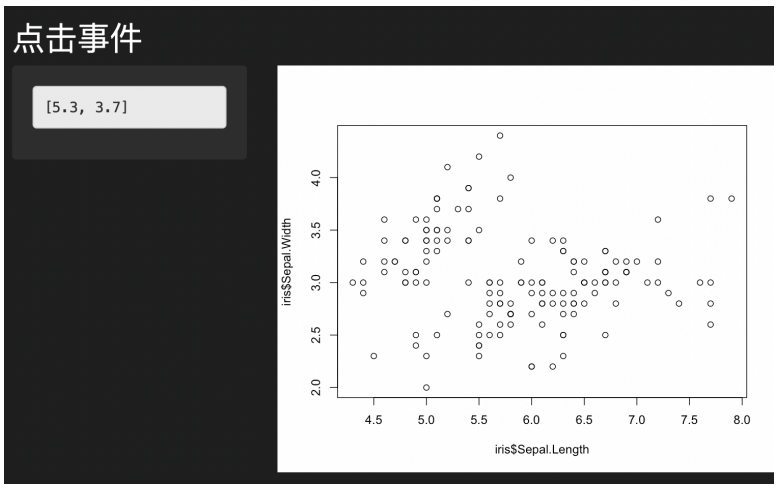

图 1.19　点击

这个 Shiny 程序很简单，首先就是在主页面绘制一幅散点图，然后捕获点击事件，再将点击事件位置结果输出到侧边栏。需要注意的是这里使用了 seq 函数是为了应用程序在第一次单击之前不做任何事情。

除了我们可以获取点击位置的点之外，我们还可以获取点击附近的位置，使用的函数的是 nearPoints()。我们对上面的例子进行简单的修改，代码如下所示。

```
library(shiny)
library(shinythemes)

#DefineUIforapplicationthatdrawsahistogram
ui<-fluidPage(
theme=shinytheme("darkly"),
#Applicationtitle
titlePanel("点击事件"),

#Sidebarwithasliderinputfornumberofbins
sidebarLayout(

sidebarPanel(
verbatimTextOutput("info")

),

#Showaplotofthegenerateddistribution
```

```
mainPanel(
plotOutput("distPlot",click="myclick"),
dataTableOutput("tabl")

)
)
)

#Defineserverlogicrequiredtodrawahistogram
server<-function(input,output){

output$distPlot<-renderPlot({
#generatebinsbasedoninput$binsfromui.R
plot(iris$Sepal.Length,iris$Sepal.Width)

})

output$info<-renderPrint({
req(input$myclick)
x<-round(input$myclick$x,2)
y<-round(input$myclick$y,2)
cat("[",x,",",y,"]",sep="")
})

output$tabl<-renderDataTable({
req(input$myclick)
nearPoints(iris,input$myclick,xvar="Sepal.Length",yvar="Sepal.Width")
})
}

#Runtheapplication
shinyApp(ui=ui,server=server)
```

程序的输出结果如图 1.20 所示。

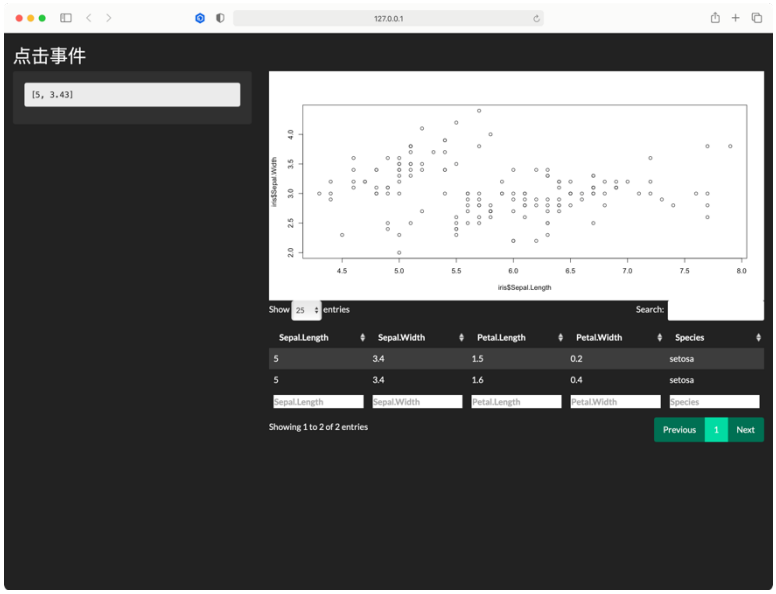

图 1.20　点击事件

上面的代码中，使用了 nearPoints()函数，这个函数的第一个参数是数据集，第二个参数是点击事件，第三个参数和第四个参数分别是图形中 x 轴和 y 轴所对应的变量。我们上文提到过有四种鼠标事件,其他的鼠标事件的实现方式是类似的，想要了解更多的内容，可以在 R 的终端输入’?plotOutput’查看帮助，帮助文档中给出了更多详细的例子，这里就不再过多赘述了。

1.4.2 上传与下载

与用户之间传输文件是应用程序的一项常见功能。很多时候，我们希望能够上传数据然后进一步进行数据分析，或者将分析的下载下来。这个时候就需要了解 shiny 中的上传与下载功能了。

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。

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

fileInput 函数与其他的 ui 部件使用方法类似，只有两个必须的参数就是 id 和 label。与之对应在 server 中的处理则稍微复杂一些。大多数情况下，ui 部件返回的结果都是一个向量，但是 fileInput 返回的结果是包含四列的数据框，分别是 name，size，type 和 datapath。其中 name 表示的是文件在计算机上面的原始文件名。Size 表示的是文件大小，以字节为单位，默认情况下最多只能上传 5MB 大小的文件。Type 是文件类型的正式规范，通常是从扩展名派生的。Datapath 表示的是服务器上已上传数据的路径。将此路径视为短暂路径：如果用户上载了更多文件，则该文件可能会被删除。数据总是保存到一个临时目录并指定一个临时名称。

我们来创建一个最简单的例子，来帮助我们理解，代码如下所示。

```
library(shiny)
library(shinythemes)


ui<-fluidPage(
shiny::fileInput(inputId="upload",NULL,buttonLabel="Upload...",multiple=TRUE),
tableOutput("files")
)
server<-function(input,output,session){
output$files<-renderTable(input$upload)
}
#Runtheapplication
shinyApp(ui=ui,server=server)
```

程序的输出结果如图 1.21 所示。

图 1.21　上传

我们可以点击 Upload，上传一个文件，这里选择一张图片进行上传，上传之后的结果如图 1.22 所示。

图 1.22　上传

上面的代码中，使用 fileInput 函数来实现上传文件，其中 multiple 参数表示的是允许上传多个文件。另外，有几个细节需要注意一下，最一开始没有上传文件，因此 fileInput 的返回值是 null，需要使用 req(input$file)来确保代码等待直到上载第一个文件。fileInput 函数还有一个参数这里没有使用到，accept，这个参数可以限制允许上传的文件类型，例如 accept=’.csv’，表示允许上传 scv 文件。另外，在 R 中获取文件扩展名的最简单方法是 tools::file_ext()。

我们尝试限制文件类型，只允许上传 csv 类型和 tsv 类型的文件，然后展示上传的数据，代码如下所示。

```
library(shiny)
library(shinythemes)
```

```
ui<-fluidPage(
shiny::fileInput("file",NULL,accept=c(".csv",".tsv")),
numericInput("n","Rows",value=5,min=1,step=1),
tableOutput("head")
)

server<-function(input,output,session){
data<-reactive({
req(input$file)

ext<-tools::file_ext(input$file$name)
switch(ext,
csv=read.csv(input$file$datapath,sep=",”),
tsv=vroom::vroom(input$file$datapath,delim="\t"),
validate("Invalidfile;Pleaseuploada.csvor.tsvfile")
)
})

output$head<-renderTable({
head(data(),input$n)
})
}
#Runtheapplication
shinyApp(ui=ui,server=server)
```

程序的输出结果如图 1.23 所示。

图 1.23　上传文件

接着我们上传一个 csv 文件，如图 1.24 所示。

...1	...2	...3	...4	...5	...6	...7	...8	...9	...10	...11
NA	Country	RedMeat	WhiteMeat	Eggs	Milk	Fish	Cereals	Starch	Nuts	Fr&Veg
NA	Albania	10.1	1.4	0.5	8.9	0.2	42.3	0.6	5.5	1.7
NA	Austria	8.9	14.0	4.3	19.9	2.1	28.0	3.6	1.3	4.3
NA	Belgium	13.5	9.3	4.1	17.5	4.5	26.6	5.7	2.1	4.0
NA	Bulgaria	7.8	6.0	1.6	8.3	1.2	56.7	1.1	3.7	4.2

图 1.24　上传文件

这里我随意上传了一个 csv 格式的数据，可以看到，文件上传成功，并且 shiny 还输出了所上传数据的结果。

代码实现的方式其实也非常的简单，文件通过 fileInput 上传成功之后，会生成一个 datapath，也就是文件上传之后在服务器的路径，有了文件路径之后，直接通过读取函数读取就可以获取到上传的文件内容了。

接下来我们开看看下载，同样，用户界面也很简单：使用 downloadButton(id)或 downloadLink(id)为用户提供单击下载文件的功能。需要注意的额是，downloadButton 函数没有与之对应的 render 函数，与之匹配的是 downloadHandler 函数。这个函数有两个关键参数 filename 和 content。filename 应该是一个不带参数的函数，该函数返回文件名。Content 是一个带有 file 参数的函数，file 这是保存文件的路径。此函数的工作是将文件保存在 Shiny 知道的地方，然后将其发送给用户。

我们对上一个例子进行修改，实现下载数据的功能，代码如下所示。

```
library(shiny)
library(shinythemes)


ui<-fluidPage(
shiny::fileInput("file",NULL,accept=c(".csv",".tsv")),
numericInput("n","Rows",value=5,min=1,step=1),
tableOutput("head"),
downloadButton("download","Download.csv")
)

server<-function(input,output,session){
data<-reactive({
req(input$file)

ext<-tools::file_ext(input$file$name)
switch(ext,
csv=read.csv(input$file$datapath,sep=","),
tsv=vroom::vroom(input$file$datapath,delim="\t"),
validate("Invalidfile;Pleaseuploada.csvor.tsvfile")
```

```
)
})

output$head<-renderTable({
head(data(),input$n)
})
output$download<-downloadHandler(
filename=function(){
paste0(input$file$datapath,".tsv")
},
content=function(file){
vroom::vroom_write(data(),file)
}
)

}
#Runtheapplication
shinyApp(ui=ui,server=server)
```

程序的输出结果如图 1.25 所示。

图 1.25　上传文件

我们上传一个文件，然后点击下载，如图 1.26 所示。

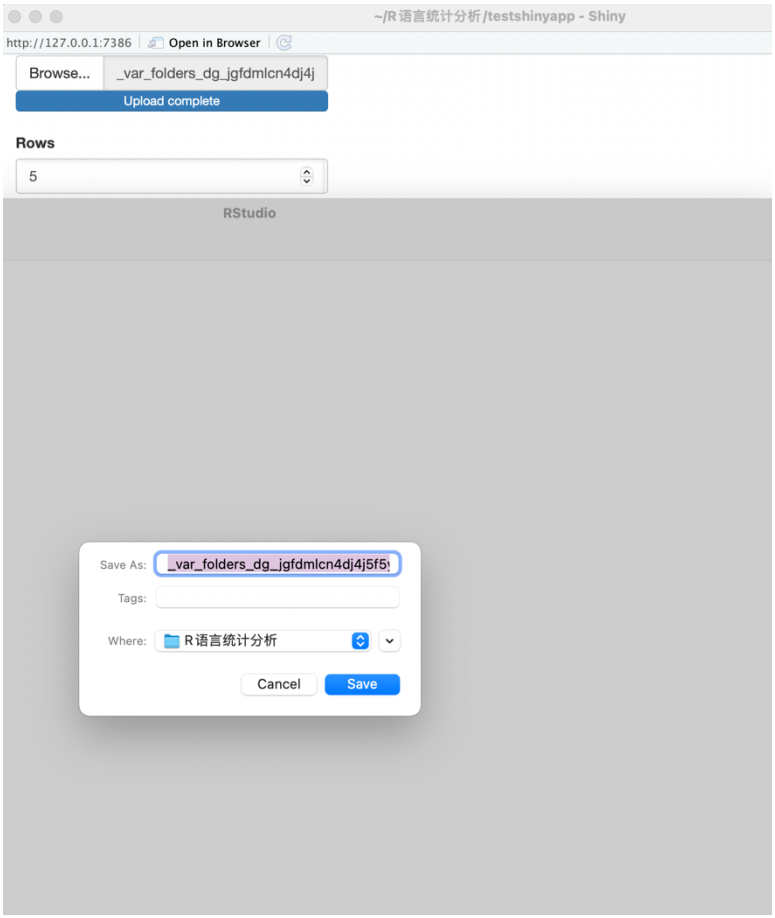

图 1.26　下载文件

这个时候修改文件名，并且选择将要保存文件的位置，最后点击保存，即可保存文件。

1.4.3 动态 UI

再之前的例子中，我们已经创建过很多的 ui，之前所创建的 ui 全部都是静态的，也就是说 ui 不会发生改变。如果想要实现 ui 根据某种逻辑发生改变，通常有三种方式。

（1）使用 update 函数族来修改输入控件的参数。

（2）使用 tabsetPanel()有条件地显示和用户界面的隐藏部分。

（3）使用 uiOutput()和 renderUI()使用代码生成用户界面的选定部分。

接下来我们分别来了解这三种方式，首先来看第一种方式，代码如下所示。

```
library(shiny)
library(shinythemes)


ui<-fluidPage(
numericInput("min","Minimum",0),
```

```
numericInput("max","Maximum",3),
sliderInput("n","n",min=0,max=3,value=1)
)
server<-function(input,output,session){
observeEvent(input$min,{
updateSliderInput(inputId="n",min=input$min)
})
observeEvent(input$max,{
updateSliderInput(inputId="n",max=input$max)
})
}
#Runtheapplication
shinyApp(ui=ui,server=server)
```

程序的输出结果如图 1.27 所示。

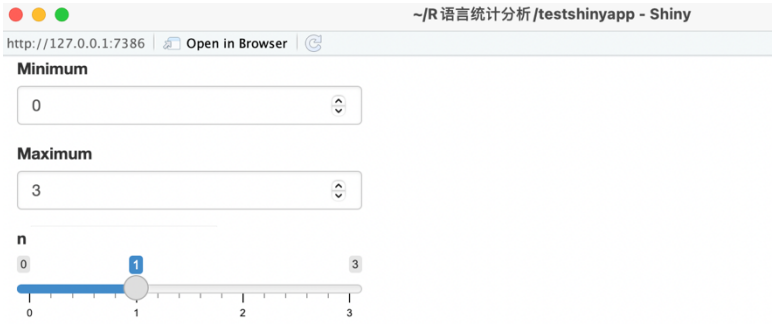

图 1.27　动态 ui

当我们修改 Minimum 和 Maximim 的时候，这个滑块也会发生改变。我们调整 minimum 为 2，Maximum 为 10。结果如图 1.28 所示。

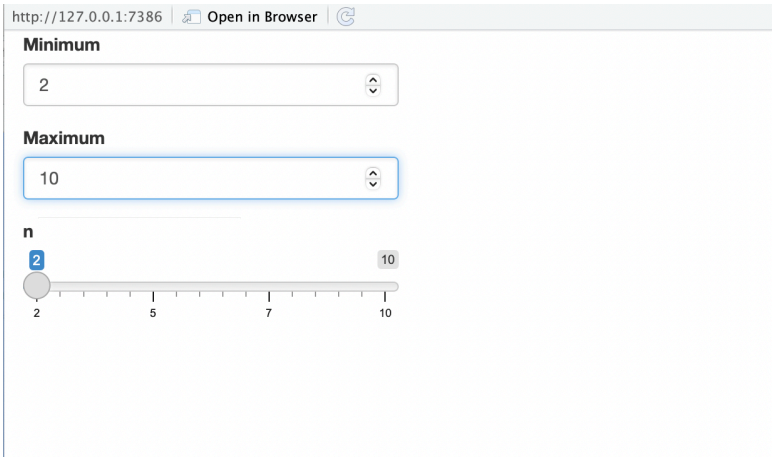

图 1.28　动态 ui

从结果中可以看到，滑块的最小值变成了 2，最大值变成了 10。这种方式的修改其实也非常直观，就是通过 ui 的输出去影响其他 ui 的参数，从而起到动态调整 ui 的效果。

需要注意，这里使用了 observeEvent 函数。（这里需要把这个东西再稍微解释一下）

第二种实现方式是使用 tabsetPanel()有条件地显示和用户界面的隐藏部分，同样，我们先看例子，代码如下所示。

```
library(shiny)
ui<-fluidPage(
sidebarLayout(
sidebarPanel(
selectInput("controller","Show",choices=paste0("panel",1:3))
),
mainPanel(
tabsetPanel(
id="switcher",
type="hidden",
tabPanelBody("panel1","Panel1content"),
tabPanelBody("panel2","Panel2content"),
tabPanelBody("panel3","Panel3content")
)
)
)
)

server<-function(input,output,session){
observeEvent(input$controller,{
updateTabsetPanel(inputId="switcher",selected=input$controller)
})
}
shinyApp(ui,server)
```

代码的输出结果如题 1.29 所示。

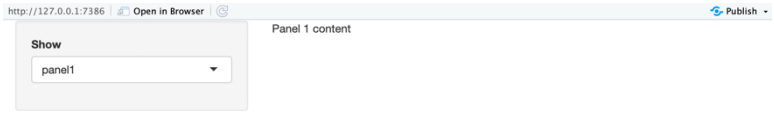

图 1.29　动态 ui

当前 shiny 显示的是第一个面板，我们可以调整 show 的值，从而选择不同的面板，如图 1.30 所示。

图 1.30　动态 ui

这里通过修改 show 的值，从而选择了面板 3。这种实现方式与第一种类似，都使用到了 update 族函数，不同点这里是通过显示不同的面板来实现动态的 ui。

第三种方式是使用 uiOutput()和 renderUI()使用代码生成用户界面的选定部分。这种实现方式其实原理非常的简单，就把 ui 看成与表格，图片等等一样的对象即可，怎么创建展示图片，就怎么创建展示 ui。我们来看一个例子，代码如下所示。

```
library(shiny)
ui<-fluidPage(
textInput("label","label"),
selectInput("type","type",c("slider","numeric")),
uiOutput("numeric")
)
server<-function(input,output,session){
output$numeric<-renderUI({
if(input$type=="slider"){
sliderInput("dynamic",input$label,value=0,min=0,max=10)
}else{
numericInput("dynamic",input$label,value=0,min=0,max=10)
}
})
}
shinyApp(ui,server)
```

shiny 结果如图 1.31 所示。

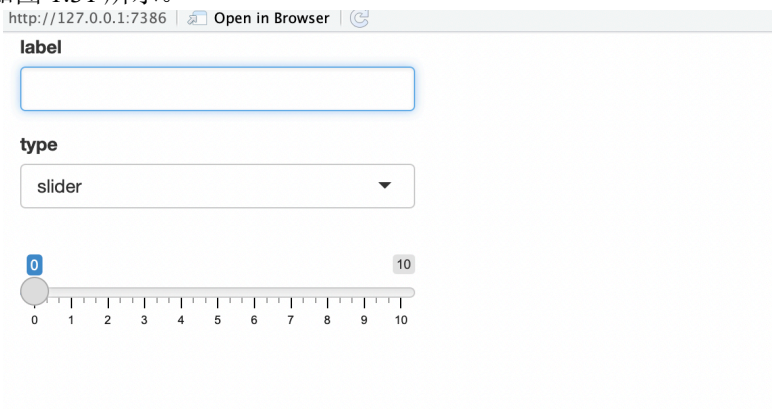

图 1.31　动态 ui

我们调整 label，并且选择不同 type，结果如图 1.32 所示。

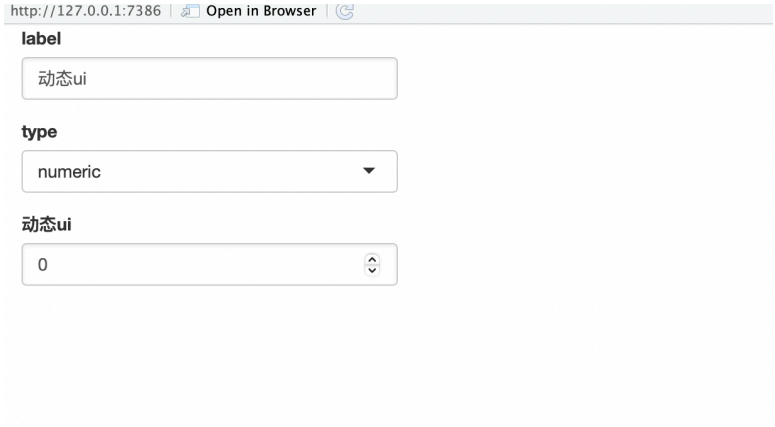

图 1.32　动态 ui

可以看到，ui 从滑块变成了数字输入，并且 ui 的标题也发生了改变。代码的实现逻辑也非常的简单，通过 ui 中部件 ui 的结果来控制 server 中部件 ui 的创建，然后通过 renderUI 来传递所创建 ui 的部件，通过 uiOutput 来展示在 server 创建的部件 ui。

通过这三种方式，就可以实现动态的控制 ui。

1.4.4 用户反馈

构建一个用户优化的 shiny 程序，必不可少的一个要素就是用户反馈。例如，当用户输入错误的时候，通过消息提示用户，或者提供一个进度条让用户知道他们的程序还在运行，需要稍许等待。这些提示能够极大的提高用户的使用体验。

1.验证

首先，我们来考虑验证，这是非常重要的反馈，当用户操作出错的时候，提示用户。就好像编程的时候使用某个函数出错了，如果函数不提供任何的错误信息，那么我们是很难修改的，友好的函数会给出清晰的错误信息，我们可以根据错误信息更加容易的修改代码。同样当用户操作出错但是没有任何错误提示，这会让用户很困惑，从而进一步放弃使用。

向用户提供反馈，可以使用 shinyFeedback 这个包，使用这个包有两个步骤，首先将 useShinyFeedback 函数添加到 ui 里面去,例如：

```
ui<-fluidPage(
shinyFeedback::useShinyFeedback(),
numericInput("n","n",value=10),
textOutput("half")
)
```

然后，在 server 函数中，调用反馈函数，这些函数包括 feedback 函数，feedbackWarning 函数，feedbackDanger 函数和 feedbackSuccess 函数。这些函数都有三个关键参数，inputId，show 和 text。inputId 就是需要验证的 ui 部件的 id；show 参数控制是否显示反馈，其是一个布尔值；text 表示的是现实的内容。

```
server<-function(input,output,session){
half<-reactive({
even<-input$n%%2==0
shinyFeedback::feedbackWarning("n",!even,"请输入偶数")
input$n/2
})

output$half<-renderText(half())
}
```

我们将这两部分代码运行出来，运行结果如图 1.33 所示。

图 1.33　反馈

在这个例子中，我们首先判断用户的输入是不是偶数，如果是偶数那么就正常输出，如果是奇数，那么就输出警告。需要注意的是，虽然输出了警告，却依然输出了计算结果。通常情况下，我们不希望对无效的输入进行计算，这个时候需要使用 req 函数，其中，req 是 require 的缩写。代码如下所示。

```
server<-function(input,output,session){
half<-reactive({
even<-input$n%%2==0
shinyFeedback::feedbackWarning("n",!even,"请输入偶数")
req(even)
input$n/2
})

output$half<-renderText(half())
}
```

当 req（）的输入不为 true 时，它会发送一个特殊的信号，告诉 Shiny 的 reactive，这里没有它所需要的输入，因此程序应该“暂停”。

2. 通知

如果没有问题，只是单纯的想告诉用户发生了一些事情，则需要使用通知。在 shiny 中，创建通知使用的是 showNotification()。

我们可以参考如下代码来了解 showNotification()的作用。

```
library(shiny)
ui<-fluidPage(
actionButton("goodnight","Goodnight")
)
server<-function(input,output,session){
observeEvent(input$goodnight,{
showNotification("Solong")
Sys.sleep(1)
showNotification("Farewell",type="message")
Sys.sleep(1)
showNotification("AufWiedersehen",type="warning")
Sys.sleep(1)
showNotification("Adieu",type="error")
})
}

shinyApp(ui,server)
```

程序的输出结果如题 1.34 所示。

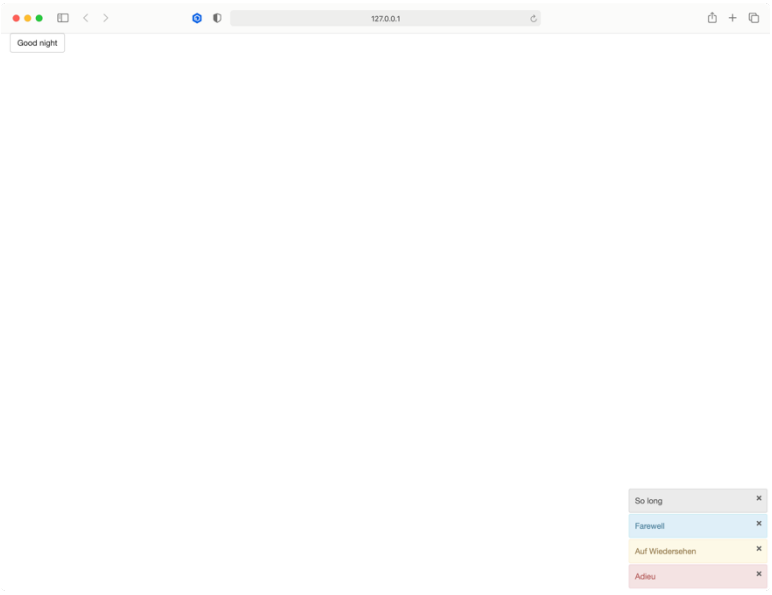

图 1.34 反馈

点击按钮之后，shiny 程序的右下角就出现提示。默认情况下，提示会在 5 秒钟之后消失。可以通过修改 duration 参数调整提示出现的时间。这里还调整了 type，通过调整 type 可以让消息更加突出。

需要注意的是，如果设置了 closeButton=TRUE，那么消息的右上角就会出现一个叉，

点击叉号，消息就会消失。如果设置了 duration=null，则表示消息永远不会消失。

3.进度条

对于长时间运行的任务，进度条是必不可少的，进度条可以告诉用户，程序还在运行，并不是程序出问题了，你只需要等待即可，并且进度条还可以告诉用户大概还需要多长时间。

Shiny 的内容函数实现了进度条的功能，但是进度条的实现有一个困难点，那就是使用进度条需要将程序划分成为已知数量的小块，并且每个小块程序的运行速度大致相同。这挺不容易的，抛开这一点先不谈，我们来看一下如何实现进度条。

Shiny 内置的进度条函数是 withProgress()和 incProgress()，我们来看一个例子，代码如下所示。

```
library(shiny)
ui<-fluidPage(
numericInput("steps","都有少步数",10),
actionButton("go","开始"),
textOutput("result")
)

server<-function(input,output,session){
data<-eventReactive(input$go,{
withProgress(message="计算中",{
for(iinseq_len(input$steps)){
Sys.sleep(0.5)
incProgress(1/input$steps)
}
runif(1)
})
})

output$result<-renderText(round(data(),2))
}
shinyApp(ui,server)
```

程序的输出结果如题 1.35 所示。

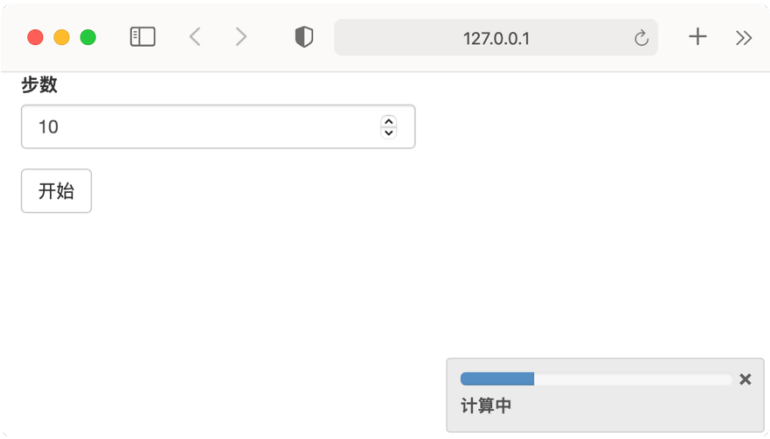

图 1.35 反馈

需要注意的是，想要使用进度条展示运行时间的代码需要放到 withProgress 函数中。这将在代码启动时显示进度条，并在完成后自动将其删除。然后使用 incProgress 函数控制进度条的速度，其第一个参数就是进度条的数量，默认情况下，进度条从 0 开始，到 1 结束，因此以 1 递增除以 step 将确保进度条在循环结束时完成。

1.4.5 确认

有时，某动作可能具有潜在危险，要么需要确保用户的确想这样做，要么要让他们能够在为时已晚之前退出。保护用户免于意外执行危险动作的最简单方法是要求明确确认。最简单的方法是使用一个对话框，该对话框强制用户从一小组动作中选择一个。在 Shiny 中，创建对话框是使用 modalDialog 函数。

例如，你想要用户确定是否要删除某个文件，那么你需要询问用户是否是真的想删除。

```
library(shiny)
modal_confirm<-modalDialog(
"是否真的想这么做？",
title="删除文件",
footer=tagList(
actionButton("cancel","取消"),
actionButton("ok","删除",class="btnbtn-danger")
)
)


ui<-fluidPage(
actionButton("delete","删除所有文件?")
)

server<-function(input,output,session){
observeEvent(input$delete,{
```

```
showModal(modal_confirm)
})

observeEvent(input$ok,{
showNotification("文件删除")
removeModal()
})
observeEvent(input$cancel,{
removeModal()
})
}
shinyApp(ui,server)
```

上面的代码中首先通过 modalDialog 函数创建了一个对话框，对话框有一个标题，有两个按钮。在 server 部分，如果点击了 delete 按钮，那么就会跳出对话框，然后程序会根据对话框的点击结果，进行不同的操作。需要注意的是我们使用 showModal()和 removeModal()来显示和隐藏对话框。

代码的运行结果如题 1.36 所示。

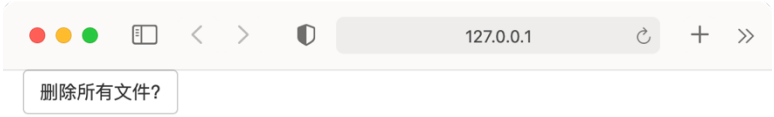

图 1.36 确认

如果点击删除所有文件，那么会出现如图 1.37 结果。

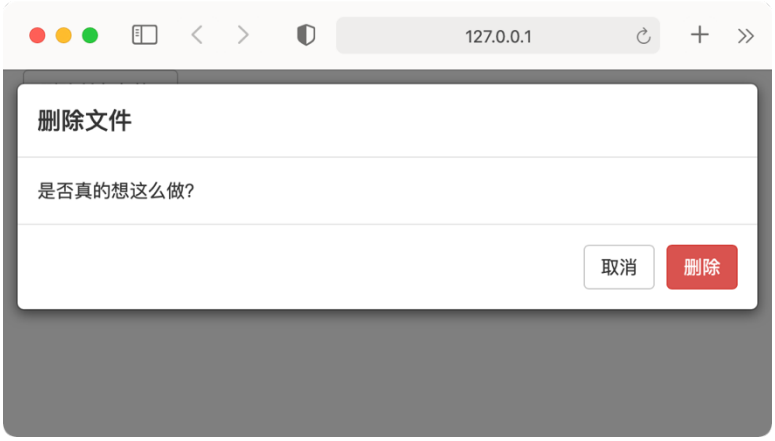

图 1.37 确认

通过这种方式，就可以向用户确认是不是要进行这样的操作。

1.5 如何制作一款成功的 Shiny 应用

制作一款简单的 shiny 程序其实非常的简单，一旦我们了解了 shiny 的基本工作原理，任何 R 用户都可以快速的制作 shiny 程序。所以，对于我们而言，问题应该是如何制作一款成功的 shiny 程序。

1.5.1 成功的 shiny 应用

定义成功并不是一件容易的事情，但是对于应用程序而言，有一些标准可以用于判断。首先，应用应该交付成功。换句话说，开发人员团队能够从规范设计到应用实现再到测试应用再到交付应用。这是一个非常以工程为导向的成功定义，但它也是一个务实的定义：

首先，一个从未达到可用性状态的应用程序不是一个成功的应用程序。

其次，应用应该满足了特定的目的，并且，其应该在大多数情况下，甚至是某种极端情况，是可以正常运行的，

最后，应用还应该是用户友好的，如果这些用户由于应用太难使用，太难理解，太慢或用户体验的设计没有内在逻辑而无法使用该应用程序，则称该应用程序为成功是不合适的。

1.5.2 准备工作

以上从应用的交付，应用的功能完成以及应用的用户友好程度来衡量一款 shiny 应用是否成功。当我们知道了成功的标准之后，我们要开始制作我们的应用，但在我们正式开始动手之前，我们应该做一些准备工作。

首先，我们要明确开发原则，KISS 原则就是一个很好的方法论，KISS 原则的全文是 keepitsimpleandstupid。KISS 原则指出，简单性应该是设计的主要目标，并且应该避免不必要的复杂性。

构建应用程序时，应牢记这一原则。当选择的方式和内容来实现某些功能，尽量做到一个最简单的方案，这样任何人都能够对现有的代码进行理解和维护，未知或难以掌握的技术会减少找到将来可以维护该代码段的人的机会，并降低了协作的流畅度。

然后，工具的选择和团队的结构对于成功也是至关重要的。构建项目的时候需要版本控制的工具，因为我们需要确保代码库的安全，我们需要识别不同版本之间的区别，并有可能及时的回到之前的版本。

将大任务进行分解也是实现复杂项目的必备方法，只有对项目进行分解，才能使得每个开发人员都可以将精力集中在一个节点上，而不必在实现功能时考虑全局基础结构。

从团队的角度来看，完成项目需要一个负责监督整个项目的人员和几个负责特定功能的开发人员。开发负责人将对整个项目有一个完整的了解，并管理团队，以便所有开发人员都能实现相互融合的功能。对于复杂的应用程序，可能很难完全了解整个应用程序在做什么。在大多数情况下，并不需要所有开发人员都拥有完整项目认知。通过定义一个负责人，这个负责人将必须了解整个情况：软件的每个部分正在做什么，如何使所有东西协同工作，避免开发冲突，并且在开发完成之后进行检测，以确保应用程序生成的返回的结果是正确的。开发人员则专注于小功能。如果负责人正确地将团队的开发人员之间的工作分开，他们将专注于应用程序的一个或多个部分，而不必知道应用程序在做什么的每一点。明确分工之后，项目能够更有效率的进行。

1.5.3 工作流程

通过遵循给定的工作流程，构建健壮的，可用于生产环境的 Shiny 应用程序将变得更加容易。我们提倡的内容分为五个步骤：

（1）设计。

（2）原型。

（3）构建。

（4）加强。

（5）部署。

当然，与任何工作流程一样，这一种工作流程也不是一刀切所有问题的解决方案：所有项目都是唯一的，具有不同技术要求，特定的计划和编程团队。但是，遵循此工作流程将帮助开发者养成构建应用程序项目的良好习惯。

1.设计

工作流的第一部分是设计部分。在设计过程中，需要定义应用程序的构建方式，从客户端/最终用户的角度来看，此步骤涉及对他们希望应用程序执行的工作有一个清晰的了解，并与开发人员对程序的功能进行评估，例如，可能执行的操作，执行多少时间。采取以实现所需的功能，等等。

从开发人员团队的角度来看，此步骤还涉及对客户的要求有一个清晰的了解，换句话说，这涉及将需求转换为技术规范。例如，客户可能会写类似“将图保存在数据库中，以便我们以后可以搜索它们”之类的东西：从应用程序用户的角度来看，这是一个明确的功能，从开发人员的角度来看，这一要求在技术实现上，可以有多种方式翻译。

这一步实际上意味着在编程之前需要进行很多思考。此步骤的主要目标是花时间思考应用程序，这个过程不需要任何的实现，避免工作了一段时间后发现某些问题，从而需要返工。

2.原型

shiny 应用程序是一个接口（前端或“ui”），用于将信息传递给在服务器端（后端或“server”）计算并将结果输出给最终用户。

原型包括两个方面，一方面是应用的外观，输入和输出的位置，总体设计，交互等。

另一方面是后端逻辑，这些逻辑将决定的实际输出。

有很多原型设计的工具，这一部分就不做过多的介绍。

3. 构建

当我们知道我们要构建什么应用，并且有了原型之后，就需要开始分解任务，并进一步实现改 shiny 程序。

4. 测试

当我们构建好应用之后，我们还需要进行单元测试，构建可靠的测试套件对于项目的成功至关重要，因为它可以使项目长期稳定，无论您是要添加新功能还是重构现有代码。

5. 部署

当我们的应用完全准备就绪，我们就可以部署我们的应用。

这就是构建 shiny 应用的步骤，当然，这个步骤不仅仅可以用来构建 shiny 应用，构建其他的应用程序同样适合。

1.6 小结

在这一章节，我们介绍了 Shiny 程序的相关内容，涉及的内容也比较的多。读者对这一部分内容的学习，已经能够比较好的掌握如何使用 shiny 去构建一款应用。

当然，关于 shiny 的很多内容这一章节没有涉及到，比如 shiny 的主题，shiny 的仪表盘，plotly 包与 shiny 的结合，shinyDashboard 包，DT 包，更多的 ui 部件等等内容，其实随着开发者不断的开发新的包，关于 shiny 的内容会永远会有新的。因此，在最后，分享一份关于 shiny 的资源，这一份资源叫做 AwesomeShinyExtensions，链接为：https://github.com/nanxstats/awesome-shiny-extensions。

在这个链接中，包好了大量的有关于 shiny 的拓展内容，并且已经归类分组好了，是一份非常好的学习材料。

以上就是本章节关于 Shiny 的内容，Shiny 对于 R 用户而言是一个非常强大的工具，对于每一位 R 用户都是有必要了解的。