

# Best practices

---

## Introduction

---

当你开始使用Shiny时，即使是小型应用程序也需要很长时间，因为你必须学习基础知识。然而，随着时间的推移，您将更熟悉软件包的基本界面和反应性的关键想法，并且您将能够创建更大、更复杂的应用程序。当您开始编写更大的应用程序时，您将遇到一系列新的挑战：保持复杂且不断增长的代码库井然有序、稳定和可维护。这将包括以下问题：

- “我在这个巨大的文件中找不到我要找的代码。”
- “我已经6个月没有处理过这个代码了，如果我做出任何更改，恐怕我会破坏它。”
- “其他人开始和我一起研究这个应用程序，我们一直站在对方的脚趾上。”
- “该应用程序在我的电脑上工作，但在我的合作者或生产中不起作用。”

在这本书的“最佳实践”中，您将从软件工程中学习一些关键概念和工具，这些概念和工具将帮助您克服这些挑战：

- 在第17章中，我将简要介绍软件工程的伟大想法。
- 在第18章中，我将向您展示如何从Shiny应用程序中提取代码到独立应用程序中，并讨论您为什么想这样做。
- 在第19章中，您将了解Shiny的模块系统，该系统允许您将耦合的UI和服务器代码提取到隔离和可重用的组件中。
- 在第20章中，我将向您展示如何将您的应用程序变成R包，并激励为什么这些投资会为更大的应用程序获得回报。
- 在第21章中，您将学习如何将现有的非正式测试转换为自动测试，每当您的应用程序更改时，这些测试都可以轻松重新运行。
- 在第23章中，您将学习如何识别和解决应用程序中的性能瓶颈，确保即使被数百名用户使用，它们也能保持快速。

当然，你不能在一本书的一部分中学习关于软件工程的所有知识，所以我也会给你指出了解更多信息的好地方。

## Chapter 17. General guidelines

---

### 17.1 Introduction

本章介绍了编写Shiny应用程序时所需的最重要的软件工程技能：代码组织、测试、依赖管理、源代码控制、持续集成和代码审查。这些技能并不特定于Shiny应用程序，但如果您想编写复杂的应用程序，这些应用程序随着时间的推移更容易维护，而不是更难，您需要了解所有这些技能。

提高你的软件工程技能是一段终生的旅程。当你开始学习时，会有挫折感，但要明白每个人都会经历同样的问题，如果你坚持下去，你会克服它们。大多数人在学习新技术时都会经历同样的演变：“我不明白它，每次使用它时必须查找它”到“我模糊地理解它，但仍然经常阅读文档”到最终“我理解它，可以流畅地使用它”。到达最后阶段需要时间和练习。

我建议每周留出一些时间来练习你的软件开发技能。在此期间，尽量避免触及应用程序的行为或外观，而是集中精力使应用程序更容易理解和开发。这将使您的应用程序在未来更容易更改，随着您提高软件开发技能，您对应用程序的首次尝试也将变得更高质量。

## 17.2 Code organization

任何傻瓜都可以编写计算机能够理解的代码。优秀的程序员编写人类可以理解的代码。——马丁·福勒

提高应用程序质量的最明显方法之一是提高其代码的可读性和可理解性。世界上最好的程序员无法维护他们无法理解的代码库，所以这是一个很好的起点。

成为一名优秀的程序员意味着培养对未来需要与这个代码库互动的其他人的同理心（即使只是未来的你！）。像所有形式的同理心一样，这需要练习，只有在您多次这样做后才会变得更容易。随着时间的推移，您会开始注意到某些实践可以提高代码的可读性。没有通用规则，但一些一般准则包括：

- 变量和函数名称是否清晰简洁？如果没有，哪些名字会更好传达代码的意图？
- 我有什么注释需要解释复杂的代码吗？
- 整个功能适合我的屏幕吗？还是可以打印在一张纸上？如果没有，有办法把它分成更小的碎片吗？
- 我是否在整个应用程序中多次复制和粘贴相同的代码块？如果是这样，是否有办法使用函数或变量来避免重复？
- 我的应用程序的所有部分都纠缠在一起，还是我可以单独管理应用程序的不同组件？

没有灵丹妙药来解决所有这些问题——很多时候它们涉及主观判断——但有两个特别重要的工具：

- **函数**，第18章的主题，允许您减少UI代码的重复，使您的server函数更容易理解和测试，并允许您更灵活地组织应用程序代码。
- **Shiny模块**是第19章的主题，可以编写孤立的、可重复使用的代码，以协调前端和后端行为。模块允许您优雅地分离问题，以便（例如）应用程序中的单个页面可以独立运行，或者不再需要复制和粘贴重复的组件。

## 17.3 Testing

为应用程序制定测试计划对于确保其持续稳定性至关重要。如果没有测试计划，每个更改都会危及应用程序。当应用程序足够小，你可以把它全部放在脑子里时，你可能会觉得不需要额外的测试计划。当然，测试非常简单的应用程序似乎比它的价值更麻烦。然而，一旦其他人开始为您的应用程序做出贡献，或者当您花了足够的时间离开它，以至于您忘记了这一切是如何结合在一起时，缺乏计划可能会造成痛苦。

测试计划可以完全是手动的。一个很好的开始是一个简单的文本文件，给出一个脚本来检查是否一切顺利。然而，随着应用程序变得越来越复杂，该脚本将不得不增长，您要么花越来越多的时间手动测试应用程序，要么开始跳过一些脚本。

因此，下一步是开始自动进行一些测试。自动化需要时间设置，但随着时间的推移会得到回报，因为您可以更频繁地运行测试。出于这个原因，如第21章所述，为Shiny开发了各种形式的自动测试。正如该章将解释的那样，您可以开发：

- 确认单个函数正确行为的单元测试。
- 集成测试以确认反应者之间的相互作用。
- 功能测试，以验证浏览器的端到端体验
- 负载测试，以确保应用程序能够承受您预期的流量。

编写自动测试的美妙之处在于，一旦您花时间编写它，您将不再需要再次手动测试应用程序的这一部分。您甚至可以利用持续集成（稍后会详细介绍）在发布应用程序之前每次更改代码时运行这些测试。

## 17.4 Dependency management

如果您曾经尝试过，在R中重现其他人编写的一些分析，甚至试图重运行您前段时间编写的一些分析或Shiny应用程序，您可能在依赖性方面遇到麻烦。应用程序的依赖项是它运行所需的源代码之外的任何内容。这些可能包括硬盘驱动器上的文件、外部数据库或API，或应用程序使用的其他R包。

对于您将来可能想要复制的任何分析，请考虑使用[renv](#)，它使您能够创建可重现的R环境。使用renv，您可以捕获应用程序使用的确切软件包版本，以便当您在另一台计算机上使用此应用程序时，您可以使用完全相同的软件包版本。这对在生产中运行的应用程序至关重要，不仅仅是因为它在第一次运行时就获得了正确的版本，而且因为它还随着时间的推移将您的应用程序与版本变化隔离开来。

另一个管理依赖性的工具是[config](#)包。配置包实际上并不管理依赖项本身，但它确实为您提供了一个方便的地方来跟踪和管理R包以外的依赖项。例如，您可以指定应用程序所依赖的CSV文件的路径，或您需要的API的URL。在配置文件中，枚举这些可以为您提供一个可以跟踪和管理这些依赖项的单一位置。更好的是，它使您能够为不同的环境创建不同的配置。例如，如果您的应用程序分析包含大量数据的数据库，您可以选择配置几个不同的环境：

- 在生产环境中，您将应用程序连接到真正的“生产”数据库。
- 在测试环境中，您可以将应用程序配置为使用测试数据库，以便在测试中正确行使数据库连接，但如果您意外进行更改，损坏数据，则不会有损坏生产数据库的风险。
- 在开发中，您可以将应用程序配置为使用带有数据子集的小型CSV，以允许更快的更多的再处理。

最后，要注意对本地文件系统进行假设。例如，如果您的代码引用了 `C:\data\cars.csv` 或 `~/my-projects/genes.rds` 中的数据，您需要意识到这些文件不太可能存在于另一台计算机上。相反，可以使用相对于应用程序目录的路径（例如 `data/cars.csv` 或 `genes.rds`），也可以使用config包，使外部路径明确且可配置。

## 17.5 Source code management

任何长期编程的人都不可避免地处于一种状态，他们意外损坏了应用程序，并希望回滚到之前的工作状态。当手动完成时，这是令人难以置信的艰巨。然而，幸运的是，您可以依靠“版本控制系统”，该系统可以轻松跟踪原子变化，回滚到以前的工作，并集成多个贡献者的工作。

R社区中最受欢迎的版本控制系统是Git。Git通常与GitHub配对，GitHub是一个可以轻松与他人共享您的git repos的网站。熟练使用Git和GitHub肯定需要工作，但任何经验丰富的开发人员都会确认这种努力是值得的。如果您是git的新手，我强烈建议从Jenny Bryan的[Happy Git and Github for the user](#)开始使用。

## 17.6 Continuous integration/deployment(CI, CD)

一旦您使用版本控制系统并拥有一套强大的自动化测试，您可能会从持续集成（CI）中受益。CI是一种永久验证您对应用程序所做的更改没有破坏任何东西的方法。您可以追溯使用它（如果您刚刚所做的更改破坏了应用程序，则通知您）或主动使用（如果提议的更改会破坏您的应用程序，则通知您）。

有各种服务可以连接到Git repo，并在您推送新提交或提出更改时自动运行测试。根据您的代码托管位置，您可以考虑[GitHub操作](#)、[Travis CI](#)、[Azure Pipelines](#)、[AppVeyor](#)、[Jenkins](#)或[GitLab CI/CD](#)等。

## All checks have passed

5 successful checks





✓		continuous-integration/appveyor/branch — AppV...	<a href="#">Details</a>
✓		continuous-integration/appveyor/pr — AppVeyor ...	<a href="#">Details</a>
✓		continuous-integration/travis-ci/pr — The Travis ...	<a href="#">Details</a>
✓		continuous-integration/travis-ci/push — The Tra...	<a href="#">Details</a>

图17.1：CI运行示例，显示四个独立测试环境的成功结果

图17.1显示了当CI系统连接到GitHub以测试拉取请求时的样子。如您所见，所有CI测试都显示绿色检查，这意味着每个自动测试环境都是成功的。如果任何测试失败，在将更改合并到应用程序中之前，您会收到失败的提醒。拥有CI流程不仅可以防止经验丰富的开发人员犯意外错误，还可以帮助新贡献者对他们的更改充满信心。

## 17.7 Code reviews

许多软件公司已经发现了让别人在代码正式纳入代码库之前审查代码的好处。这个“代码审查”过程有很多好处：

- 在错误被纳入应用程序之前捕获错误，使其修复成本低得多。
- 提供教学机会——各级程序员通常通过审查他人的代码来学习新东西。
- 促进团队之间的交叉授粉和知识共享，以消除只有一个人了解该应用程序。
- 由此产生的对话通常会提高代码的可读性。

通常，代码审查涉及您以外的其他人，但即使只有您，您仍然可以受益。大多数经验丰富的开发人员都会同意，花点时间查看自己的代码通常会发现一些小缺陷，特别是如果您可以在写作和审查之间至少停留几个小时。

以下是您在审查代码时需要考虑的一些问题：

- 新函数有简洁但令人回味的名字吗？
- 你觉得代码中有什么地方令人困惑吗？
- 未来哪些领域可能会发生变化，并且会特别受益于自动化测试？
- 代码的样式与应用程序的其余部分匹配吗？（或者更好的是，制定文档代码样式。）

如果您嵌入到一个具有强大工程文化的组织中，为数据科学代码设置代码审查应该相对简单，并且您将可以利用现有的工具和经验。如果你在一个几乎没有其他软件工程师的组织中，你可能需要做更令人信服的事情。

我推荐两个资源：

- <https://github.com/thoughtbot/guides/tree/master/code-review>
- <https://google.github.io/eng-practices/review/>

## 17.8 Summary

现在您已经了解了一点软件工程师的心态，接下来的章节将深入探讨适用于Shiny应用程序的功能编写、测试、安全性和性能的细节。你需要在其他章节之前阅读第18章，否则你可以跳过。

## Chapter 18. Functions

随着应用程序的扩大，越来越难将所有碎片都掌握在脑海中，也会越来越难理解。反过来，这使得添加新功能变得更加困难，当出现问题时也更难找到解决方案（即更难调试）。如果您不采取深思熟虑的步骤，应用程序的开发速度将放缓，并且工作起来会越来越不愉快。

在本章中，您将了解编写函数是如何为您提供帮助。这对UI和服务器组件来说往往略有不同：

- 在用户界面中（UI），您有多个地方重复的组件，但有细微的变化。将重复的代码拉出到函数中可以减少重复（使从一个地方更新许多控件更容易），并且可以与函数式编程技术相结合，以一次生成许多控件。
- 在服务器中（Server），复杂的反应很难调试，因为您需要处于应用程序中间。将反应拉出到一个单独的函数中，即使该函数仅在一个地方调用，也使调试变得容易得多，因为您可以独立于反应性进行计算实验。

函数在Shiny应用程序中具有另一个重要作用：它们允许您将应用程序代码分散到多个文件中。虽然您当然可以拥有一个巨大的 `app.R` 文件，但当分散在多个文件上时，管理起来要容易得多。

我想你已经熟悉了函数的基础知识。本章的目标是激活您现有的技能，向您展示一些使用函数可以大大提高应用程序清晰度的具体情况。一旦您掌握了本章中的想法，下一步就是学习如何编写需要跨UI和服务器协调的代码。这需要**模块**，您将在第19章中了解这些模块。

```
library(shiny)
```

### 18.1 File organization

在我们继续讨论如何在应用程序中使用函数之前，我想从一个直接的好处开始：函数可以生活在 `app.R` 之外。根据它们有多大，你可以把它们放在两个地方：

- 我建议将大型函数（以及他们需要的任何较小的辅助函数）放入他们自己的 `R/{function-name}.R` 文件。
- 您可能想把更小、更简单的功能收集到一个地方。我经常为此使用 `R/utils.R`，但如果它们主要在您的UI中使用，您可能会使用 `R/ui.R`。

如果您以前制作过R包，您可能会注意到Shiny使用相同的约定来存储包含函数的文件。事实上，如果您正在制作一个复杂的应用程序，特别是如果有多个作者，那么制作一个完整的软件包有很大的优势。如果您想这样做，我建议您阅读“[Engineering Shiny](#)”书，并使用随附的[golem](#)包。当我们更多地谈论测试时，我们将再次触及软件包。

### 18.2 UI functions

函数是减少UI代码重复的强大工具。让我们从一些重复代码的具体示例开始。想象一下，您正在创建一堆滑块，每个滑块需要从0到1不等，从0.5开始，步骤为0.1。您可以进行一堆复制和粘贴来生成所有滑块：



```
ui <- fluidRow(
  sliderInput("alpha", "alpha", min = 0, max = 1, value = 0.5, step = 0.1),
  sliderInput("beta", "beta", min = 0, max = 1, value = 0.5, step = 0.1),
  sliderInput("gamma", "gamma", min = 0, max = 1, value = 0.5, step = 0.1),
  sliderInput("delta", "delta", min = 0, max = 1, value = 0.5, step = 0.1)
)
```

但我认为识别重复的模式并提取一个函数是值得的。这使得UI代码大大简化：

```
sliderInput01 <- function(id) {
  sliderInput(id, label = id, min = 0, max = 1, value = 0.5, step = 0.1)
}

ui <- fluidRow(
  sliderInput01("alpha"),
  sliderInput01("beta"),
  sliderInput01("gamma"),
  sliderInput01("delta")
)
```

在这里，一个函数在两个方面有帮助：

- 我们可以给该函数一个令人回味的名称，以便将来重新阅读代码时更容易理解发生了什么。
- 如果我们需要改变行为，我们只需要在一个地方做。例如，如果我们决定这些步骤需要更精细的分辨率，我们只需要在一个地方写 `step = 0.01`，而不是四个地方。

## 18.2.1 Other applications

函数在许多其他地方都很有用。以下是一些让你的创意源源不断的想法：

- 如果您正在为您的国家使用自定义的 `dateInput()`，请将其拉到一个地方，以便您可以使用一致的参数。例如，想象一下，你想要一个日期控制，供美国人用来选择工作日：

```
usWeekDateInput <- function(inputId, ...) {
  dateInput(inputId, ..., format = "dd M, yy", daysofweekdisabled = c(0, 6))
}
```

注意 `...` 的使用；这意味着您仍然可以传递任何其他参数到 `dateInput()`

- 或者，也许您想要一个单选按钮，以便更轻松地提供图标：

```
iconRadioButtons <- function(inputId, label, choices, selected = NULL) {
  names <- lapply(choices, icon)
  values <- if (is.null(names(choices))) names(choices) else choices
  radioButtons(inputId,
    label = label,
    choiceNames = names, choiceValues = values, selected = selected
  )
}
```

- 或者，如果有多个选择，您在多个地方重复使用：

```
stateSelectInput <- function(inputId, ...) {  
  selectInput(inputId, ..., choices = state.name)  
}
```

如果您在组织内开发许多闪亮的应用程序，您可以通过将这样的功能放入共享软件包来帮助提高跨应用程序的一致性。

## 18.2.2 Functional programming

回到我们的激励示例，如果您对函数式编程感到满意，您可以进一步减少代码。

```
library(purrr)  
  
vars <- c("alpha", "beta", "gamma", "delta")  
sliders <- map(vars, sliderInput01)  
ui <- fluidRow(sliders)
```

这里有两个大想法：

- `map()` 为存储在 `vars` 中的每个字符串调用 `sliderInput01()` 一次。它返回一个滑块列表。
- 当您将列表传递到 `fluidRow()`（或任何html容器）时，它会自动解压列表，使元素成为容器的子元素。

如果您想了解有关 `map()`（或其基本等价物 `lapply()`）的更多信息，您可能会喜欢Advanced R的[函数章节](#)。

## 18.2.3 UI as data

如果控件有多个不同的输入，则可以进一步概括这个想法。首先，我们使用 `tibble::tribble()` 创建一个内联数据框，定义每个控件的参数。我们正在将UI结构变成一个显式的数据结构。

```
vars <- tibble::tribble(  
  ~ id, ~ min, ~ max,  
  "alpha", 0, 1,  
  "beta", 0, 10,  
  "gamma", -1, 1,  
  "delta", 0, 1,  
)
```

然后我们创建一个函数，其中参数名称与列名匹配：

```
mySliderInput <- function(id, label = id, min = 0, max = 1) {  
  sliderInput(id, label, min = min, max = max, value = 0.5, step = 0.1)  
}
```

最后，我们使用 `purrr::pmap()` 为每行 `vars` 调用 `mySliderInput()` 一次：

```
sliders <- pmap(vars, mySliderInput)
```

如果这个代码对你来说看起来像胡言乱语，不要担心：你可以继续使用复制和粘贴。但从长远来看，我建议你学习更多关于函数式编程的知识，因为它给你提供了简洁表达其他冗长概念的绝佳能力。您可以查看其他示例，请参阅第[10.3节](#)，以了解使用这些技术响应用户操作生成动态UI的更多示例。

## 18.3 Server functions

每当您有一个长反应（例如>10行）时，您应该考虑将其拉出一个不使用任何反应性的单独函数。这有两个优点：

- 如果您可以对代码进行分区，以便反应性存在于 `server()` 中，并且复杂的计算存在于您的函数中，那么调试和测试代码就容易得多。
- 当查看反应式表达式或输出时，除了仔细阅读代码块外，没有办法轻松判断它所依赖的确切值。然而，函数定义会确切地告诉你输入是什么。

UI中函数的主要好处往往是减少重复。server中函数的关键优势往往与隔离（isolation）和测试（testing）有关。

### 18.3.1 Reading uploaded data

从第[9.1.3节](#)中取出此server。它包含一个中等复杂的 `reactive()`

```
server <- function(input, output, session) {  
  data <- reactive({  
    req(input$file)  
  
    ext <- tools::file_ext(input$file$name)  
    switch(ext,  
      csv = vroom::vroom(input$file$datapath, delim = ","),  
      tsv = vroom::vroom(input$file$datapath, delim = "\t"),  
      validate("Invalid file; Please upload a .csv or .tsv file")  
    )  
  })  
  
  output$head <- renderTable({  
    head(data(), input$n)  
  })  
}
```

如果这是一个真正的应用程序，我会认真考虑提取一个专门用于读取上传文件的功能：

```
load_file <- function(name, path) {  
  ext <- tools::file_ext(name)  
  switch(ext,  
    csv = vroom::vroom(path, delim = ","),  
    tsv = vroom::vroom(path, delim = "\t"),  
    validate("Invalid file; Please upload a .csv or .tsv file")  
  )  
}
```



在提取此类功能时，避免将反应（reactive）作为输入或返回输出。相反，将值传递到参数中，并假设调用者将根据需要，将结果转换为反应。这不是一个硬性规则；有时你的函数输入或输出反应是有意义的。但一般来说，我认为最好将应用程序的反应和非反应部分尽可能分开。在这种情况下，我仍然在使用 `validate()`，因为在Shiny的 `validate()` 之外的工作原理与 `stop()` 类似。但我把 `req()` 保存在server中，因为文件解析代码不应该负责知道它何时运行。

由于现在是一个独立的函数，它可以存在于自己的文件（比如 `R/load_file.R`）中，从而保持 `server()` 的瘦身。这有助于保持server函数专注于反应性的大局，而不是每个组件下面的小细节。

```
server <- function(input, output, session) {  
  data <- reactive({  
    req(input$file)  
    load_file(input$file$name, input$file$datapath)  
  })  
  
  output$head <- renderTable({  
    head(data(), input$n)  
  })  
}
```

另一个大优势是，您可以在Shiny应用程序之外的控制台上执行 `load_file()`。如果您转向应用程序的正式测试（见第21章），这也使该代码更容易测试。

### 18.3.2 Internal functions

大多数时候，您希望使该函数完全独立于server函数，以便您可以将其放入一个单独的文件中。但是，如果函数需要使用 `input`、`output` 或 `session`，则该函数在server函数中存在，可能是有意义的：

```
server <- function(input, output, session) {  
  switch_page <- function(i) {  
    updateTabsetPanel(input = "wizard", selected = paste0("page_", i))  
  }  
  
  observeEvent(input$page_12, switch_page(2))  
  observeEvent(input$page_21, switch_page(1))  
  observeEvent(input$page_23, switch_page(3))  
  observeEvent(input$page_32, switch_page(2))  
}
```

这并没有使测试或调试变得更容易，但它确实减少了重复的代码。

我们当然可以将 `session` 添加到函数的参数中：

```
switch_page <- function(i) {
  updateTabsetPanel(input = "wizard", selected = paste0("page_", i))
}

server <- function(input, output, session) {
  observeEvent(input$page_12, switch_page(2))
  observeEvent(input$page_21, switch_page(1))
  observeEvent(input$page_23, switch_page(3))
  observeEvent(input$page_32, switch_page(2))
}
```

但这感觉很奇怪，因为该函数仍然从根本上与该应用程序耦合，因为它只影响具有一组非常特定选项卡的名“向导（wizard）”的控件。

## 18.4 Summary

随着您的应用程序越来越大，从应用程序的流程中提取非反应性函数将使您的生活大大简化。函数允许您编写反应性和非反应性代码，并将代码分散到多个文件中。这通常使您更容易看到应用程序的反应图形状，并且通过将复杂的逻辑从应用程序中移动到常规的R代码中，可以更容易进行实验、重现和测试。当你开始提取函数时，可能会感到有点缓慢和沮丧，但随着时间的推移，你会越来越快，很快它就会成为你工具箱中的关键工具。

本章中的这些功能有一个重要的缺点——它们只能生成UI或服务器组件，而不是两者。在下一章中，您将学习如何创建Shiny模块，这些模块将UI和服务器代码协调为单个对象。

# Chapter 19. Shiny modules

在最后一章中，我们使用函数将Shiny应用程序的部分分解为独立的部分。函数适用于完全在服务器端或完全在客户端的代码。对于跨越两者的代码，即服务器代码是否依赖于UI中的特定结构，您将需要一种新技术：模块。

在最简单的层面上，模块是一对UI和server函数。模块的魔力之所以出现，是因为这些函数是以一种特殊方式构造的，从而创建一个“命名空间（namespace）”。到目前为止，在编写应用程序时，控件的名称（`id`）是全局的：server函数的所有部分都可以看到用户界面的所有部分。模块使您能够创建只能从模块内部看到的控件。这被称为 `anamespace`，因为它创建了与应用程序其余部分隔离的“名称（`names`）”的“空间（`spaces`）”。

Shiny模块有两个很大的优势。首先，命名空间可以更轻松地了解您的应用程序的工作原理，因为您可以单独编写、分析和测试单个组件。其次，由于模块是函数，它们可以帮助您重用代码；任何你能用函数做的事情，你都可以用模块来做。

```
library(shiny)
```

## 19.1 Motivation

在我们深入研究创建模块的细节之前，了解它们如何改变应用程序的“形状（shape）”是有用的。我将借用[Eric Nantz](https://youtu.be/yLLVo2VL50)的例子，他在rstudio::conf(2019)上谈到了模块：<https://youtu.be/yLLVo2VL50>。Eric有动力使用模块，因为他有一个大型复杂的应用程序，如图19.1所示。您不知道这个应用程序的细节，但由于许多相互连接的组件，您可以了解其复杂性。

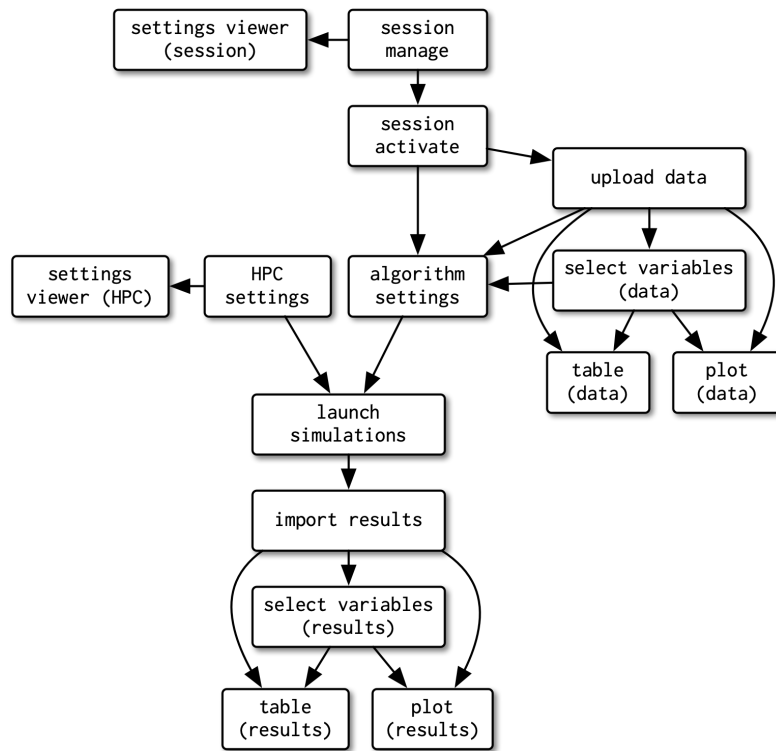


图19.1：一个复杂应用程序的粗略草图。我已尽最大努力将其简单地显示在图表中，但仍然很难理解所有部分是什么

图19.2显示了使用模块重写后应用程序现在的样子：

- 该应用程序被分成几块，每块都有一个名称。命名片段意味着控件的名称可以更简单。例如，以前该应用程序有“会话管理（session manage）”和“会话激活（session activate）”，但现在我们只需要“管理（manage）”和“激活（activate）”，因为这些控件嵌套在会话模块中。这是命名空间！
- 模块是一个包含定义输入和输出的黑盒。其他模块只能通过模块的接口（外部）进行通信，它们无法到达内部并直接检查或修改内部控制和反应。这为整个应用程序强制执行更简单的结构。
- 模块是可重复使用的，因此我们可以编写函数来生成黄色和蓝色组件。这可以显著减少应用程序中的代码总量。

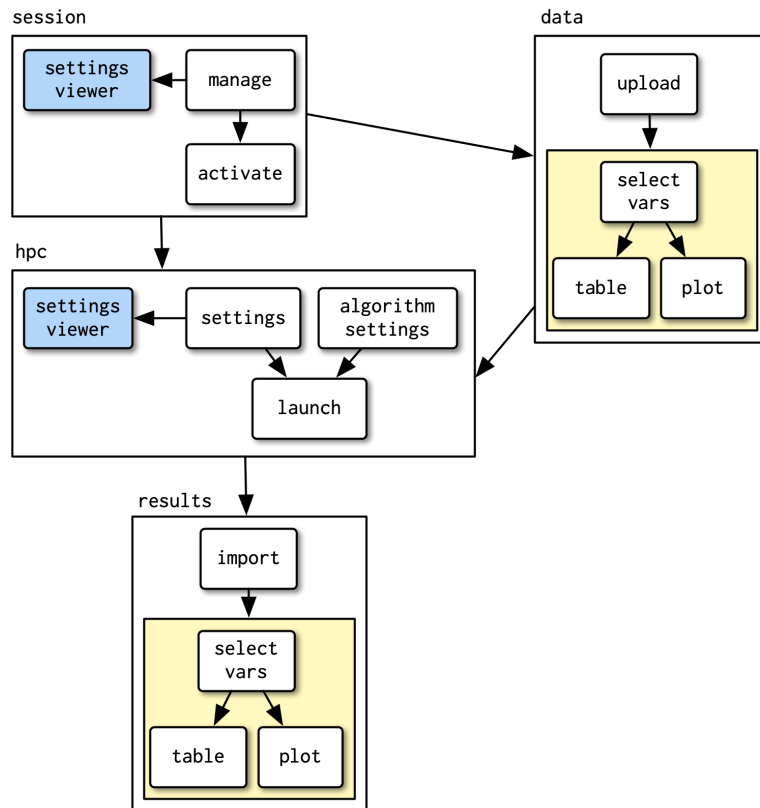


图19.2：将应用程序转换为使用模块后，更容易看到应用程序的大组件，并查看在多个地方重复使用的内容（蓝色和黄色组件）。

## 19.2 Module basics

要创建您的第一个模块，我们将从一个非常简单的应用程序中提取一个模块，该应用程序可以绘制直方图：

```

ui <- fluidPage(
  selectInput("var", "Variable", names(mtcars)),
  numericInput("bins", "bins", 10, min = 1),
  plotOutput("hist")
)
server <- function(input, output, session) {
  data <- reactive(mtcars[[input$var]])
  output$hist <- renderPlot({
    hist(data(), breaks = input$bins, main = input$var)
  }, res = 96)
}
  
```

这个应用程序非常简单，提取一个模块没有真正的好处，但在我们深入研究更现实、更复杂的用例之前，它将说明基本机制。

模块与应用程序非常相似。就像应用程序一样，它由两部分组成：

- 生成 `ui` 规范的**模块UI (module UI)** 函数。
- 在 `server` 函数内运行代码的**模块服务器 (module server)** 函数。

这两个函数有标准形式。他们都接受一个 `id` 参数，并用它来命名模块。要创建一个模块，我们需要从应用程序UI和server中提取代码，并将其放入模块UI和server中。

## 19.2.1 Module UI

我们将从模块UI开始。有两个步骤：

- 将UI代码放在具有 `id` 参数的函数中。
- 在对 `NS()` 的调用中包装每个现有ID，以便（例如）`"var"` 变成 `NS(id, "var")`

这产生了以下函数：

```
histogramUI <- function(id) {  
  tagList(  
    selectInput(NS(id, "var"), "Variable", choices = names(mtcars)),  
    numericInput(NS(id, "bins"), "bins", value = 10, min = 1),  
    plotOutput(NS(id, "hist"))  
  )  
}
```

在这里，我通过 `tagList()` 形式，返回了UI组件，这是一种特殊类型的布局函数，允许您将多个组件捆绑在一起。调用 `histogramUI()` 的人有责任根据他们的需要将结果包装在 `column()` 或 `fluidRow()` 等布局函数中。

## 19.2.2 Module server

接下来，我们解决server函数。这被包装在另一个函数中，该函数必须有一个 `id` 参数。此函数使用 `id` 调用 `moduleServer()`，以及一个看起来像常规server函数的函数：

```
histogramServer <- function(id) {  
  moduleServer(id, function(input, output, session) {  
    data <- reactive(mtcars[[input$var]])  
    output$hist <- renderPlot({  
      hist(data(), breaks = input$bins, main = input$var)  
    }, res = 96)  
  })  
}
```

这两个层次的函数，在这里很重要。我们稍后会回到他们身上，但简而言之，它们有助于区分模块的参数和server函数的参数。如果这看起来非常复杂，请不要担心；它基本上是模版，您可以为您创建的每个新模块复制和粘贴。

请注意，`moduleServer()` 自动处理命名空间：在 `moduleServer(id)` 内部，`input$var` 和 `input$bins` 指的是带有名称 `NS(id, "var")` 和 `NS(id, "bins")` 的输入。

## 19.2.3 Updated app

现在我们有ui和server函数，最好编写一个函数，使用它们来生成一个应用程序，我们可以用它来进行实验和测试：

```

histogramApp <- function() {
  ui <- fluidPage(
    histogramUI("hist1")
  )
  server <- function(input, output, session) {
    histogramServer("hist1")
  }
  shinyApp(ui, server)
}

```

请注意，像所有Shiny控件一样，您需要在UI和server中使用相同的 `id`，否则这两部分将无法连接（其实，`id` 定义了一个命名空间）。

模块在Shiny 0.13（2016年1月）中引入了 `callModule()`，并在Shiny 1.5.0（2020年6月）中引入了 `moduleServer()`，并进行了大修。如果您不久前学习过模块，您可能已经学习了 `callModule()`，并想知道 `moduleServer()` 是怎么回事。这两个函数是相同的，只是前两个参数被翻转了。这是一个简单的变化，导致整个应用程序的结构相当不同：

```

histogramServerOld <- function(input, output, session) {
  data <- reactive(mtcars[[input$var]])
  output$hist <- renderPlot({
    hist(data(), breaks = input$bins, main = input$var)
  }, res = 96)
}
server <- function(input, output, session) {
  callModule(histogramServerOld, "hist1")
}

```

对于这个简单的应用程序来说，差异在很大程度上是肤浅的，但 `moduleServer()` 使带有参数的更复杂的模块更容易理解。

## 19.2.4 Namespacing

现在我们有了一个完整的应用程序，让我们回过头来再谈谈命名空间。使模块工作的关键思想是，每个控件（即它的 `id`）的名称现在由两部分决定：

- 第一部分来自模块**用户**，即调用 `histogramServer()` 的开发人员。
- 第二部分来自模块**作者**，即编写 `histogramServer()` 的开发人员。

这个由两部分组成的规范意味着您，模块作者，无需担心与用户创建的其他UI组件发生冲突。您拥有自己的“空间”名称，并且可以安排以最好地满足您自己的需求。

命名空间将模块变成黑匣子。从模块外部，您看不到其中的任何输入、输出或反应。例如，以下面的应用程序为例。文本输出 `output$out` 永远不会更新，因为没有 `input$bins`；`bins` 输入只能在 `hist1` 模块内看到。



```

ui <- fluidPage(
  histogramUI("hist1"),
  textOutput("out")
)
server <- function(input, output, session) {
  histogramServer("hist1")
  output$out <- renderText(paste0("Bins: ", input$bins))
}

```

如果您想从应用程序中其他地方的反应中获取输入，您需要将它们明确地传递给模块函数；我们很快就会回到这一点。

请注意，模块UI和server在名称空间的表达方式上有所不同：

- 在模块UI中，命名空间是*明确的*：每次创建输入或输出时，您都必须调用 `NS(id, "name")`。
- 在模块server中，命名空间是*隐式的*。您只需在调用 `moduleServer()` 使用 `id`，然后Shiny自动命名空间 `input` 和 `output`，以便在您的模块代码中，`input$name` 表示带有 `nameNS NS(id, "name")` 的输入。

## 19.2.5 Naming conventions

在这个例子中，我为模块的所有组件使用了特殊的命名方案，我建议您也将其用于自己的模块。在这里，模块绘制了直方图，所以我称它为 `histogram` 模块。然后，这个基本名称在各个地方使用：

- `R/histogram.R` 保存模块的所有代码。
- `histogramUI()` 是模块UI。如果它主要用于输入或输出，我会调用 `histogramInput()` 或 `histogramOutput()`。
- `histogramServer()` 是模块server。
- `histogramApp()` 创建一个完整的应用程序，用于交互式实验和更正式的测试。

## 19.2.6 Exercises

1. 为什么将模块放在 `R/` 目录中自己的文件中是良好做法？您需要做什么来确保它由您的Shiny应用程序加载？
2. 以下模块UI包含一个关键错误。它是什么，为什么会引起问题？

```

histogramUI <- function(id) {
  tagList(
    selectInput("var", "Variable", choices = names(mtcars)),
    numericInput("bins", "bins", value = 10, min = 1),
    plotOutput("hist")
  )
}

```

3. 每次单击go时，以下模块都会生成一个新的随机数：

```

randomUI <- function(id) {
  tagList(
    textOutput(NS(id, "val")),
    actionButton(NS(id, "go"), "Go!")
  )
}

randomServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    rand <- eventReactive(input$go, sample(100, 1))
    output$val <- renderText(rand())
  })
}

```

创建一个应用程序，在单个页面上显示此模块的四个副本。验证每个模块是否独立。您如何更改 `randomUI()` 的返回值，使显示更具吸引力？

4. 你已经厌倦了打字模块样板了吗？阅读有关 [RStudio snippets](#) 并将以下片段添加到您的RStudio配置中，以便更轻松地创建新模块。

```

${1}UI <- function(id) {
  tagList(
    ${2}
  )
}

${1}Server <- function(id) {
  moduleServer(id, function(input, output, session) {
    ${3}
  })
}

```

## 19.3 Inputs and outputs

有时，只有模块UI和server的 `id` 参数的模块很有用，因为它允许您在自己的文件中隔离复杂的代码。这对于聚合独立组件的应用程序特别有用，例如企业仪表板，每个选项卡都显示每个业务线的定制报告。在这里，模块允许您在自己的文件中开发每个部分，而不必担心ID在组件之间发生冲突。

然而，很多时候，您的模块UI和server将需要额外的参数。向模块UI添加参数可以更好地控制模块外观，允许您在应用程序中的更多地方使用相同的模块。但模块UI只是一个常规的R函数，因此需要学习的Shiny特有的知识相对较少，其中大部分内容已经在第18章中涵盖了。

因此，在接下来的章节中，我将重点关注模块server，并讨论您的模块如何接受额外的反应性输入并返回一个或多个反应性输出。与常规的Shiny代码不同，将模块连接在一起需要您明确输入和输出。起初，这会让人感到厌烦。这当然比Shiny通常的自由形式更复杂。但模块强制执行特定的通信线路是有原因的：它们需要更多的工作，但更容易理解，并允许您构建更复杂的应用程序。

您可能会看到使用 `session$userData` 或其他技术来突破模块straitjacket的建议。警惕这样的建议：它向您展示了如何绕过命名空间强加的规则，使您很容易重新引入应用程序的复杂性，并首先大大减少了使用模块的好处。

### 19.3.1 Getting started: UI input + server output

为了了解输入和输出的工作原理，我们将从一个模块开始，该模块允许用户从数据集包提供的内置数据中选择数据集。这本身并不非常有用，但它说明了一些基本原则，是更复杂模块的有用构建块，您之前在第1.4节中见过这个想法。

我们将从模块UI开始。在这里，我使用一个额外的参数，以便您可以将选项限制为内置数据集，这些数据集要么是数据框（`filter = is.data.frame`）或矩阵（`filter = is.matrix`）。我使用此参数可选地过滤数据集中找到的对象，然后创建一个 `selectInput()`

```
datasetInput <- function(id, filter = NULL) {  
  names <- ls("package:datasets")  
  if (!is.null(filter)) {  
    data <- lapply(names, get, "package:datasets")  
    names <- names[vapply(data, filter, logical(1))]  
  }  
  
  selectInput(NS(id, "dataset"), "Pick a dataset", choices = names)  
}
```

模块server也很简单：我们只需使用 `get()` 来检索具有其名称的数据集。这里有一个新想法：像函数一样，与常规 `server()` 不同，这个模块server返回一个值。在这里，我们利用了通常的规则，即函数中处理的最后一个表达式成为返回值。这个值应该始终是反应性值。

```
datasetServer <- function(id) {  
  moduleServer(id, function(input, output, session) {  
    reactive(get(input$dataset, "package:datasets"))  
  })  
}
```

要使用返回某值的模块server，您只需用 `<-` 捕获其返回值。下面的模块应用程序演示了这一点，在那里我捕获数据集，然后将其显示在 `tableOutput()`

```
datasetApp <- function(filter = NULL) {  
  ui <- fluidPage(  
    datasetInput("dataset", filter = filter),  
    tableOutput("data")  
  )  
  server <- function(input, output, session) {  
    data <- datasetServer("dataset")  
    output$data <- renderTable(head(data()))  
  }  
  shinyApp(ui, server)  
}
```

我在设计这个功能时做出了一些行政决定：

- 它需要一个传递给模块UI的 `filter` 参数，因此很容易尝试该输入参数。

- 我使用表格输出来显示所有数据。你在这里使用什么其实并不重要，但你的用户界面越有表现力，就越容易检查模块是否达到你的预期。

### 19.3.2 Case study: selecting a numeric variable

接下来，我们将创建一个控件，允许用户从给定的被动数据集中选择指定类型的变量。因为我们希望数据集是反应性的，所以我们无法在启动应用程序时填写选项。这使得模块UI非常简单：

```
selectVarInput <- function(id) {  
  selectInput(NS(id, "var"), "Variable", choices = NULL)  
}
```

Server函数将有两个参数：

- 从中选择变量 `data`。我希望这是反应性的值，这样它就可以与我上面创建 `dataset` 模块一起工作。
- 用于选择要列出的变量的 `filter`。这将由模块的调用者设置，因此不需要设置反应性。为了保持模块server的简单性，我已将关键想法提取到一个辅助函数中：

```
find_vars <- function(data, filter) {  
  names(data)[vapply(data, filter, logical(1))]  
}
```

然后，模块server使用 `observeEvent()` 在数据更改时更新 `inputSelect` 选项，并返回一个反应值，提供所选变量的值。

```
selectVarServer <- function(id, data, filter = is.numeric) {  
  moduleServer(id, function(input, output, session) {  
    observeEvent(data(), {  
      updateSelectInput(session, "var", choices = find_vars(data(), filter))  
    })  
  
    reactive(data()[[input$var]])  
  })  
}
```

为了制作我们的应用程序，我们再次捕获模块server的结果，并将其连接到用户界面中的输出。我想确保所有反应式管道都是正确的，所以我使用 `dataset` 模块作为反应式数据框的来源。

```
selectVarApp <- function(filter = is.numeric) {  
  ui <- fluidPage(  
    datasetInput("data", is.data.frame),  
    selectVarInput("var"),  
    verbatimTextOutput("out")  
  )  
  server <- function(input, output, session) {  
    data <- datasetServer("data")  
    var <- selectVarServer("var", data, filter = filter)  
    output$out <- renderPrint(var())  
  }  
}
```

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

### 19.3.3 Server inputs

在设计模块server时，您需要考虑谁将为每个参数提供值：是调用模块的R程序员，还是使用该应用程序的人？另一种思考方式是值何时可以改变：它是在应用程序的生命周期内固定和恒定的，还是随着用户与应用程序的交互而被动的变化？这是一个重要的设计决策，它决定了论点是否应该是被动的。

一旦你做出这个决定，我认为最好的做法是检查模块的每个输入是被动的还是恒定的。如果您不这样做，并且用户提供了错误的类型，他们将收到一条神秘的错误消息。通过快速的调用 `stopifnot()`，您可以使模块用户使用的更轻松。例如，`selectVarServer()` 可以检查 `data` 是否是反应性的，并且 `filter` 是否不是反应性的。使用以下代码：

```
selectVarServer <- function(id, data, filter = is.numeric) {
  stopifnot(is.reactive(data))
  stopifnot(!is.reactive(filter))

  moduleServer(id, function(input, output, session) {
    observeEvent(data(), {
      updateSelectInput(session, "var", choices = find_vars(data(), filter))
    })

    reactive(data()[[input$var]])
  })
}
```

如果您希望该模块被许多人多次使用，您也可以考虑使用 `if` 语句和调用 `stop()`，手动制作错误消息。

检查模块输入是否是反应性的（或非），有助于您在将模块与其他输入控件混合时避免常见问题。`input$var` 不是反应性的，因此每当您将输入值传递到模块时，您需要将其包装在 `reactive()` 里（例如 `selectVarServer("var", reactive(input$x))`）。如果你像我推荐的那样检查输入，你会得到一个明显的错误；如果你不这样做，你会得到一些神秘的东西，比如 `could not find function "data"`。

您也可以将此策略应用于 `find_vars()`，这里没有那么重要，但因为调试Shiny应用程序比调试常规R代码要难一点，我认为花更多时间检查输入确实是有意义的，这样当出现问题时，您就会收到更清晰的错误消息。

```
find_vars <- function(data, filter) {
  stopifnot(is.data.frame(data))
  stopifnot(is.function(filter))
  names(data)[vapply(data, filter, logical(1))]
}
```

这抓住了我在处理本章时犯的几个错误。

### 19.3.4 Modules inside of modules

在我们继续讨论server函数的输出之前，我想强调模块是可组合的，创建一个本身包含模块的模块可能是有意义的。例如，我们可以结合 `dataset` 和 `selectVar` 模块，制作一个模块，允许用户从内置数据集中选择变量：

```
selectDataVarUI <- function(id) {
  tagList(
    datasetInput(NS(id, "data"), filter = is.data.frame),
    selectVarInput(NS(id, "var"))
  )
}

selectDataVarServer <- function(id, filter = is.numeric) {
  moduleServer(id, function(input, output, session) {
    data <- datasetServer("data")
    var <- selectVarServer("var", data, filter = filter)
    var
  })
}

selectDataVarApp <- function(filter = is.numeric) {
  ui <- fluidPage(
    sidebarLayout(
      sidebarPanel(selectDataVarUI("var")),
      mainPanel(verbatimTextOutput("out"))
    )
  )
  server <- function(input, output, session) {
    var <- selectDataVarServer("var", filter)
    output$out <- renderPrint(var(), width = 40)
  }
  shinyApp(ui, server)
}
```

### 19.3.5 Case study: histogram

现在让我们回到原始直方图模块，并将其重构为更好组合的东西。创建模块的关键挑战是创建足够灵活的函数，可以在多个地方使用，但又足够简单，易于理解。弄清楚如何编写好的构建模块的函数是一生的旅程；预计在做对之前，你将不得不做错很多次。（我希望我能在这里提供更具体的建议，但目前这是一项你必须通过练习和有意识的反思来完善的技能。）

我还将把它视为一个输出控制，因为它确实使用一个输入（箱数），该输入仅用于调整显示，而不需要由模块返回。

```
histogramOutput <- function(id) {
  tagList(
    numericInput(NS(id, "bins"), "bins", 10, min = 1, step = 1),
    plotOutput(NS(id, "hist"))
  )
}
```



我决定给这个模块两个输入：`x`，要绘制的变量和直方图的 `title`。两者都是反应性的，这样它们就可以随着时间的推移而改变。（标题有点轻浮，但它很快就会激发一项重要技术）。注意 `title` 的默认值：它必须是反应性的，因此我们需要在 `reactive()` 中包装一个常量值。

```
histogramServer <- function(id, x, title = reactive("Histogram")) {
  stopifnot(is.reactive(x))
  stopifnot(is.reactive(title))

  moduleServer(id, function(input, output, session) {
    output$hist <- renderPlot({
      req(is.numeric(x()))
      main <- paste0(title(), " [", input$bins, "]")
      hist(x(), breaks = input$bins, main = main)
    }, res = 96)
  })
}

histogramApp <- function() {
  ui <- fluidPage(
    sidebarLayout(
      sidebarPanel(
        datasetInput("data", is.data.frame),
        selectVarInput("var"),
      ),
      mainPanel(
        histogramOutput("hist")
      )
    )
  )

  server <- function(input, output, session) {
    data <- datasetServer("data")
    x <- selectVarServer("var", data)
    histogramServer("hist", x)
  }

  shinyApp(ui, server)
}

# histogramApp()
```

请注意，如果您想允许模块用户在应用程序的不同位置放置中断控制和直方图，您可以使用多个UI功能。这里没有大用，但看到基本方法很有用。

```
histogramOutputBins <- function(id) {
  numericInput(NS(id, "bins"), "bins", 10, min = 1, step = 1)
}

histogramOutputPlot <- function(id) {
  plotOutput(NS(id, "hist"))
}

ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
```

```

    datasetInput("data", is.data.frame),
    selectVarInput("var"),
    histogramOutputBins("hist")
  ),
  mainPanel(
    histogramOutputPlot("hist")
  )
)
)

```

### 19.3.6 Multiple outputs

如果我们能将所选变量的名称包含在直方图的标题中，那就太好了。目前没有办法做到这一点，因为 `selectVarServer()` 只返回变量的值，而不是其名称。我们当然可以重写 `selectVarServer()` 来返回名称，但随后模块用户将不得不执行子设置。更好的方法是 `selectVarServer()` 同时返回名称和值。

`Sever`函数可以返回多个值，就像任何R函数可以返回多个值一样：通过返回一个列表。下面我们修改 `selectVarServer()` 以返回名称和值，作为反应值。

```

selectVarServer <- function(id, data, filter = is.numeric) {
  stopifnot(is.reactive(data))
  stopifnot(!is.reactive(filter))

  moduleServer(id, function(input, output, session) {
    observeEvent(data(), {
      updateSelectInput(session, "var", choices = find_vars(data(), filter))
    })

    list(
      name = reactive(input$var),
      value = reactive(data()[[input$var]])
    )
  })
}

```

现在我们可以更新我们的 `histogramApp()` 来利用这一点。UI保持不变；但现在我们传递所选变量的值及其名称到 `histogramServer()`

```

histogramApp <- function() {
  ui <- fluidPage(...)

  server <- function(input, output, session) {
    data <- datasetServer("data")
    x <- selectVarServer("var", data)
    histogramServer("hist", x$value, x$name)
  }
  shinyApp(ui, server)
}

```

此类代码的主要挑战是记住何时使用反应式表达式（例如 `x$value`）与何时使用反应值（例如 `x$value()`）。请记住，当将参数传递给模块时，您希望模块对值变化做出反应，这意味着您必须传递反应值，而不是当前值。

如果您发现自己经常从模块的反应性表达式返回多个值，您也可以考虑使用 [zeallot](#) 包。zeallot 提供 `%<-%` 运算符，允许您分配到多个变量（有时称为多个变量、解包或解构赋值）。这在返回多个值时很有用，因为您可以避免一层间接。

```
library(zeallot)

histogramApp <- function() {
  ui <- fluidPage(...)

  server <- function(input, output, session) {
    data <- datasetServer("data")
    c(value, name) %<-% selectVarServer("var", data)
    histogramServer("hist", value, name)
  }
  shinyApp(ui, server)
}
```

## 19.3.7 Exercises

1. 重写 `selectVarServer()`，以便 `data` 和 `filter` 都是反应性的。然后将其与应用程序功能一起使用，该功能允许用户使用 `inputSelect()`，借助 `dataset` 模块和过滤函数选择数据集。让用户能够过滤数字、字符或因子变量。
2. 以下代码定义了模块的输出和服务组件，该模块接受数字输入，并生成三个汇总统计信息的项目符号列表。创建一个应用程序功能，允许您进行实验。应用程序函数应将数据框作为输入，并使用 `numericVarSelectInput()` 选择要汇总的变量。

```
summaryOutput <- function(id) {
  tags$ul(
    tags$li("Min: ", textOutput(NS(id, "min"), inline = TRUE)),
    tags$li("Max: ", textOutput(NS(id, "max"), inline = TRUE)),
    tags$li("Missing: ", textOutput(NS(id, "n_na"), inline = TRUE))
  )
}

summaryServer <- function(id, var) {
  moduleServer(id, function(input, output, session) {
    rng <- reactive({
      req(var())
      range(var(), na.rm = TRUE)
    })

    output$min <- renderText(rng()[[1]])
    output$max <- renderText(rng()[[2]])
    output$n_na <- renderText(sum(is.na(var())))
  })
}
```

3. 以下模块输入提供了一个文本控件，允许您以ISO8601格式（yyyy-mm-dd）键入日期。通过提供server函数来完成模块，如果输入的值不是有效日期，则使用 `output$error` 显示消息。模块应该返回一个有效日期 `Date` 对象。（提示：使用 `strptime(x, "%Y-%m-%d")` 来解析字符串；如果值不是有效日期，它将返回 `NA`。）

```
ymdDateUI <- function(id, label) {  
  label <- paste0(label, " (yyyy-mm-dd)")  
  
  fluidRow(  
    textInput(NS(id, "date"), label),  
    textOutput(NS(id, "error"))  
  )  
}
```

## 19.4 Case studies

总结一下你到目前为止学到的东西：

- 模块输入（即模块server的附加参数）可以是反应或常量。选择是您根据谁设置论点以及何时改变做出的设计决定。您应该始终检查参数是否属于预期类型，以避免无益的错误消息。
- 与应用程序服务器不同，但与常规函数一样，模块server可以返回值。模块的返回值应始终是反应值，或者，如果您想返回多个值，则应为反应列表。

为了帮助这些想法融入其中，我将介绍一些案例研究，展示一些使用模块的更多示例。不幸的是，我没有空间展示你可能使用模块来帮助简化应用程序的所有可能方式，但希望这些示例能给你一点味道，说明你可以做什么，并建议未来要考虑的方向。

### 19.4.1 Limited selection + other

模块的另一个重要用途是给复杂的UI元素一个更简单的用户界面。在这里，我将创建一个Shiny默认不提供的有用控件：显示的一小部分选项，带有单选按钮和“其他”字段。该模块的内部使用多个输入元素，但从外部看，它作为一个组合对象工作。

我将用标签、选项和所选内容来参数化UI端，这些内容将直接传递给 `radioButtons()`。我还创建了一个 `textInput()`，其中包含一个占位符，默认为“其他”。为了组合文本框和单选按钮，我利用了这样一个事实，即 `choiceNames` 可以是HTML元素的列表，包括其他输入小部件。图19.3让您了解它的外观。

```
radioExtraUI <- function(id, label, choices, selected = NULL, placeholder = "Other") {  
  other <- textInput(NS(id, "other"), label = NULL, placeholder = placeholder)  
  
  names <- if (is.null(names(choices))) choices else names(choices)  
  values <- unname(choices)  
  
  radioButtons(NS(id, "primary"),  
    label = label,  
    choiceValues = c(names, "other"),  
    choiceNames = c(as.list(values), list(other)),  
    selected = selected  
  )  
}
```

How do you usually read csv files?

- ☒ read.csv()
- ☐ readr::read\_csv()
- ☐ data.table::fread()
- ☐ Other

图19.3: 使用 `radioExtraUI()` 了解您通常如何阅读CSV文件的示例

在服务器上，如果您修改占位符值，我想自动选择“其他”单选按钮。您还可以想象，如果选择了其他文本，则使用验证来确保存在一些文本。

```
radioExtraServer <- function(id) {  
  moduleServer(id, function(input, output, session) {  
    observeEvent(input$other, ignoreInit = TRUE, {  
      updateRadioButtons(session, "primary", selected = "other")  
    })  
  
    reactive({  
      if (input$primary == "other") {  
        input$other  
      } else {  
        input$primary  
      }  
    })  
  })  
}
```

然后，我把这两部分都包装在一个应用程序函数中，这样我就可以测试它了。在这里，我使用...将任意数量的参数传递到我的 `radioExtraUI()` 中。图19.4让您了解它的行为方式。

```
radioExtraApp <- function(...) {  
  ui <- fluidPage(  
    radioExtraUI("extra", ...),  
    textOutput("value")  
  )  
  server <- function(input, output, server) {  
    extra <- radioExtraServer("extra")  
    output$value <- renderText(paste0("Selected: ", extra()))  
  }  
  
  shinyApp(ui, server)  
}
```

How do you usually read csv files?

- ☐ read.csv()
- ☐ readr::read\_csv()
- ☐ data.table::fread()
- ☒ vroom::vroom()

Selected: vroom::vroom()

图19.4: 用关于如何阅读CSV的相同问题测试 `radioExtraApp()`。现在, 如果您在另一个字段中键入内容, 则会自动选择相应的单选按钮。

您可以继续总结此模块, 以达到更具体的目的。例如, 一个需要一点关注的变量是性别, 因为人们有许多不同的方式来表达他们的性别。

```
genderUI <- function(id, label = "Gender") {  
  radioExtraUI(id,  
    label = label,  
    choices = c(  
      male = "Male",  
      female = "Female",  
      na = "Prefer not to say"  
    ),  
    placeholder = "Self-described",  
    selected = "na"  
  )  
}
```

在这里, 重要的是提供最常见的选择, 男性和女性, 一个不提供这些数据的选项, 然后是一个写入选项, 人们可以使用他们最熟悉的任何术语。在这里不使用“其他”的占位符是考虑周到的。

## 19.4.2 Wizard

接下来, 我们将处理两个案例研究, 深入研究命名空间的一些微妙之处, 其中UI是由不同的人在不同的时间生成的。这些情况很复杂, 因为您需要记住命名空间的工作原理的细节。

我们将从一个包装向导界面的模块开始, 这是一种UI风格, 您可以将复杂的流程分解为一系列简单的页面, 用户逐一工作。我在第10.2.2节中展示了如何创建基本向导。现在, 我们将自动执行该过程, 以便在创建向导时, 您可以专注于每个页面的内容, 而不是它们如何连接在一起形成一个整体。

为了解释这个模块, 我将从底部开始, 我们将向上工作。向导用户界面的主要部分是按钮。每个页面都有两个按钮: 一个是将它们带到下一页, 一个是将它们返回到上一页。我们将首先创建助手来构建这些按钮:

```
nextPage <- function(id, i) {  
  actionButton(NS(id, paste0("go_", i, "_", i + 1)), "next")  
}  
prevPage <- function(id, i) {  
  actionButton(NS(id, paste0("go_", i, "_", i - 1)), "prev")  
}
```



这里唯一真正的复杂性是 `id`：由于每个输入元素都需要有一个唯一的id，因此每个按钮的id需要包括当前页面和目标页面。

接下来，我写一个函数来生成向导的页面。这包括“标题”（未显示，但用于识别切换的页面）、页面内容（由用户提供）和两个按钮。

```
wrapPage <- function(title, page, button_left = NULL, button_right = NULL) {  
  tabPanel(  
    title = title,  
    fluidRow(  
      column(12, page)  
    ),  
    fluidRow(  
      column(6, button_left),  
      column(6, button_right)  
    )  
  )  
}
```

然后我们可以把它们放在一起生成整个向导。我们循环用户提供的页面列表，创建按钮，然后将用户提供的页面包装到一个 `tabPanel` 中，然后将所有面板合并到一个 `tabsetPanel`。请注意，按钮有两种特殊情况：

- 第一页没有上一个按钮。在这里，我使用了一个技巧，`if` 条件是 `FALSE` 并且没有其他块，则返回 `NULL`。
- 最后一页使用用户提供的输入控件。我认为这是允许用户控制向导完成后会发生什么的最简单方法。

```
wizardUI <- function(id, pages, doneButton = NULL) {  
  stopifnot(is.list(pages))  
  n <- length(pages)  
  
  wrapped <- vector("list", n)  
  for (i in seq_along(pages)) {  
    # First page only has next; last page only prev + done  
    lhs <- if (i > 1) prevPage(id, i)  
    rhs <- if (i < n) nextPage(id, i) else doneButton  
    wrapped[[i]] <- wrapPage(paste0("page_", i), pages[[i]], lhs, rhs)  
  }  
  
  # Create tabsetPanel  
  # https://github.com/rstudio/shiny/issues/2927  
  wrapped$id <- NS(id, "wizard")  
  wrapped$type <- "hidden"  
  do.call("tabsetPanel", wrapped)  
}
```

Page 1

next

Page 2

prev

next

Page 3

prev

All done!

图19.5：向导UI的简单示例。

创建选项卡集面板的代码需要一些解释：不幸的是，`tabsetPanel()` 不允许我们传递选项卡列表。因此，我们需要做一点 `do.call()` 魔术来使其工作。`do.call(function_name, list(arg1, arg2, ...))` 等价于 `function_name(arg1, arg2, ...)`，所以在这里我们正在创建一个调用像 `tabsetPanel(pages[[1]], pages[[2]], ..., id = NS(id, "wizard"), type = "hidden")` 一样的面板。希望这能在未来版本的Shiny中得到简化。

现在我们已经完成了模块UI，我们需要将注意力转向模块server。server的本质是直截了当的：我们只需要让按钮工作，这样你就可以从一个页面到另一个方向。为此，我们需要为调用 `updateTabsetPanel()` 的每个按钮设置一个 `observeEvent()`。如果我们确切知道有多少页，那就相对简单了。但我们没有，因为模块的用户可以控制它。

因此，我们需要做一些函数式编程来设置  $(n - 1) * 2$  观察器（除了第一页和最后一页外，每页有两个观察器，只需要一个）。下面的server函数从提取我们在 `changePage()` 函数中一个按钮所需的基本代码开始。它使用 `input[[ ]]`，如第10.3.2节所述，因此我们可以动态控制。然后，我们使用 `lapply()` 循环所有之前的按钮（除第一页外，每个页面都需要）和所有下一个按钮（除最后一页外，每个页面都需要）。

```
wizardServer <- function(id, n) {  
  moduleServer(id, function(input, output, session) {  
    changePage <- function(from, to) {  
      observeEvent(input[[paste0("go_", from, "_", to)]], {  
        updateTabsetPanel(session, "wizard", selected = paste0("page_", to))  
      })  
    }  
    ids <- seq_len(n)  
    lapply(ids[-1], function(i) changePage(i, i - 1))  
    lapply(ids[-n], function(i) changePage(i, i + 1))  
  })  
}
```

请注意，这里不可能使用for循环来代替 `map()` / `lapply()`。for循环通过更改同一 `i` 变量的值来工作，这样到循环完成时，每个 `changePage()` 都会使用相同的值。`map()` 和 `lapply()` 通过创建每个具有自己值 `i` 的新环境来工作。

现在我们可以构建一个应用程序和简单的示例，以确保我们正确地将所有内容组合在一起：

```
wizardApp <- function(...) {
  pages <- list(...)

  ui <- fluidPage(
    wizardUI("whiz", pages)
  )
  server <- function(input, output, session) {
    wizardServer("whiz", length(pages))
  }
  shinyApp(ui, server)
}
```

不幸的是，在使用模块时，我们需要稍微重复一下，我们需要确保 `wizardServer()` 的 `n` 参数与 `wizardUi()` 的 `pages` 参数一致。这是模块系统的原则性限制，我们将在第19.5节中更详细地讨论。

现在，让我们在一个稍微现实一点的应用程序中使用向导，该应用程序有输入和输出，并产生图19.6。需要注意的要点是，即使页面由模块显示，其ID也由模块用户控制。是开发人员创建了控件，谁控制了其名称，谁最终组装了控件以最终在网页上显示并不重要。

```
page1 <- tagList(
  textInput("name", "What's your name?")
)
page2 <- tagList(
  numericInput("age", "How old are you?", 20)
)
page3 <- tagList(
  "Is this data correct?",
  verbatimTextOutput("info")
)

ui <- fluidPage(
  wizardUI(
    id = "demographics",
    pages = list(page1, page2, page3),
    doneButton = actionButton("done", "Submit")
  )
)
server <- function(input, output, session) {
  wizardServer("demographics", 3)

  observeEvent(input$done, showModal(
    modalDialog("Thank you!", footer = NULL)
  ))

  output$info <- renderText(paste0(
    "Age: ", input$age, "\n",
    "Name: ", input$name, "\n"
  ))
}
```

What's your name?

next

How old are you?

prev

next

Is this data correct?

Age: 20  
Name:

prev

Submit

图19.6：用我们的新模块创建的简单但完整的向导。

### 19.4.3 Dynamic UI

我们将以一个使用动态用户界面的案例研究结束，将[10.3.3](#)节中的动态过滤代码的一部分转换为模块。模块中动态UI的主要挑战是，由于您将在server函数中生成UI代码，我们需要更精确的定义，何时需要显式命名空间。

像往常一样，我们将从模块UI开始。这里非常简单，因为我们只是产生了一个占位，它将由一个server函数动态填充。

```
filterUI <- function(id) {  
  uiOutput(NS(id, "controls"))  
}
```

要创建模块server，我们将首先复制[10.3.3](#)节中的辅助函数：`make_ui()`对每列进行控制，`filter_var()`帮助生成最终的逻辑向量。这里只有一个区别：`make_ui()`获得一个额外的`id`参数，这样我们就可以将控件命名为模块。

```
library(purrr)  
  
make_ui <- function(x, id, var) {  
  if (is.numeric(x)) {  
    rng <- range(x, na.rm = TRUE)  
    sliderInput(id, var, min = rng[1], max = rng[2], value = rng)  
  } else if (is.factor(x)) {  
    levs <- levels(x)  
    selectInput(id, var, choices = levs, selected = levs, multiple = TRUE)  
  } else {  
    # Not supported  
    NULL  
  }  
}  
  
filter_var <- function(x, val) {  
  if (is.numeric(x)) {  
    !is.na(x) & x >= val[1] & x <= val[2]  
  }  
}
```

```

} else if (is.factor(x)) {
  x %in% val
} else {
  # No control, so don't filter
  TRUE
}
}

```

现在我们创建模块server。有两个主要部分：

- 我们使用 `purrr::map()` 和 `make_ui()` 生成 `controls`。注意此处 `NS()` 的显式使用。这是需要的，因为即使我们在模块server内，自动命名空间也仅适用于 `input`、`output` 和 `session`。
- 我们返回逻辑过滤向量作为模块输出。

```

filterServer <- function(id, df) {
  stopifnot(is.reactive(df))

  moduleServer(id, function(input, output, session) {
    vars <- reactive(names(df()))

    output$controls <- renderUI({
      map(vars(), function(var) make_ui(df()[[var]], NS(id, var), var))
    })

    reactive({
      each_var <- map(vars(), function(var) filter_var(df()[[var]], input[[var]]))
      reduce(each_var, `&`)
    })
  })
}

```

现在，我们可以将所有内容放在一个模块应用程序中，该应用程序允许您选择内置数据集，然后过滤任何数字或分类变量。

```

filterApp <- function() {
  ui <- fluidPage(
    sidebarLayout(
      sidebarPanel(
        datasetInput("data", is.data.frame),
        textOutput("n"),
        filterUI("filter"),
      ),
      mainPanel(
        tableOutput("table")
      )
    )
  )
  server <- function(input, output, session) {
    df <- datasetServer("data")
    filter <- filterServer("filter", df)
  }
}

```

```

    output$table <- renderTable(df()[filter(), , drop = FALSE])
    output$n <- renderText(paste0(sum(filter()), " rows"))
  }
  shinyApp(ui, server)
}

```

在这里使用模块的一大优势是，它总结了一堆先进的Shiny编程技术。您可以使用过滤器模块，而无需了解使其工作的动态UI和函数式编程技术。

## 19.5 Single object modules

为了结束本章，我想以简短的讨论结束对模块的常见反应性编程。如果这不是你的反应，请随时跳过本节。当有些人（像我一样！）第一次遇到模块，他们立即尝试将模块server和模块UI组合成一个模块对象。为了说明这个问题，让我们从本章第一部分概括激励示例，这样数据框现在是一个参数：

```

histogramUI <- function(id, df) {
  tagList(
    selectInput(NS(id, "var"), "Variable", names(df)),
    numericInput(NS(id, "bins"), "bins", 10, min = 1),
    plotOutput(NS(id, "hist"))
  )
}

histogramServer <- function(id, df) {
  moduleServer(id, function(input, output, session) {
    data <- reactive(df[[input$var]])
    output$hist <- renderPlot({
      hist(data(), breaks = input$bins, main = input$var)
    }, res = 96)
  })
}

```

这导致了以下应用程序：

```

ui <- fluidPage(
  tabsetPanel(
    tabPanel("mtcars", histogramUI("mtcars", mtcars)),
    tabPanel("iris", histogramUI("iris", iris))
  )
)

server <- function(input, output, session) {
  histogramServer("mtcars", mtcars)
  histogramServer("iris", iris)
}

```

我们必须在UI和server中重复ID和数据集的名称，这似乎不可取，因此想要包装成一个同时返回UI和server的单个函数是很自然的：



```

histogramApp <- function(id, df) {
  list(
    ui = histogramUI(id, df),
    server = histogramServer(id, df)
  )
}

```

然后，我们在UI和server之外定义模块，根据需要从列表中提取元素：

```

hist1 <- histogramApp("mtcars", mtcars)
hist2 <- histogramApp("iris", iris)

ui <- fluidPage(
  tabsetPanel(
    tabPanel("mtcars", hist1$ui()),
    tabPanel("iris", hist2$ui())
  )
)
server <- function(input, output, session) {
  hist1$server()
  hist2$server()
}

```

这个代码有两个问题。首先，它不起作用，因为 `moduleServer()` 必须在server函数内调用。但想象一下，这个问题不存在，或者你以其他方式解决它。仍然有一个大问题：如果我们想允许用户选择数据集，即我们想使 `df` 参数是反应性的呢。这不起作用，因为模块在server函数之前，即在我们知道该信息之前被实例化。

在Shiny中，UI和server本质上是断开的；Shiny不知道哪个UI调用属于哪个server会话。您可以在整个Shiny中看到这种模式：例如，`plotOutput()` 和 `renderPlot()` 仅通过共享ID连接。将模块作为单独的函数来反映这一现实：它们是不同的函数，除了通过共享ID连接外，它们不会连接。

## 19.6 Summary

本章向您展示了如何使用Shiny模块，这是函数的泛化，允许您将协调的UI和server代码提取到可重用的组件中。了解模块需要一段时间，但一旦了解，您将解锁一种强大的技术来简化复杂的应用程序。

在下一章中，您将学习如何将Shiny应用程序构建成为软件包，以便您可以利用R软件包可用的测试工具。

## Chapter 20. Packages

如果您正在创建大型或长期的Shiny应用程序，我强烈建议您以与R包相同的方式组织您的应用程序。这意味着你：

- 将所有R代码放在 `R/` 目录中。
- 编写一个启动应用程序的函数（即使用UI和server调用 `shinyApp()`）。
- 在应用程序的根目录中创建一个 `DESCRIPTION` 文件。

这种结构让你的脚本进入包装开发的环境中。它离一个完整的软件包还有很长的路要走，但它仍然很有用，因为它激活了新工具，使得使用更大的应用程序更容易。当我们在第21章中谈论测试时，软件包结构将进一步得到回报，因为您会获得工具，使运行测试变得容易，并查看被测试的代码的测试。从长远来看，它还可以帮助您使用 [roxygen2](#) 记录复杂的应用程序，尽管我们不会在这本书中讨论这一点。

很容易将软件包视为巨大的复杂事物，如Shiny、ggplot2或dplyr。但软件包也可以非常简单。软件包的核心思想是，它是一组组织代码和相关组件的惯例：如果您遵循这些惯例，您将免费获得一堆工具。在本章中，我将向您展示最重要的惯例，然后提供一些后续步骤的提示。

当您开始使用应用程序软件包时，您可能会发现您喜欢软件包开发的过程，并希望了解更多信息。我建议从[R包](#)开始，以获得包开发的布局，然后继续Colin Fay、Sébastien Rochette、Vincent Guyader、Cervan Girard的[Engineering Shiny](#)，以了解更多关于R包和Shiny应用程序的交叉点。

```
library(shiny)
```

## 20.1 Convert an existing app

将应用程序转换为软件包需要一些前期工作。假设您有一个名为 `myApp` 的应用程序，并且它已经位于一个名为 `myApp/` 的目录中，您将需要执行以下操作：

- 创建一个 `R` 目录并将 `app.R` 移动到其中。
- 通过包装将您的应用程序转换为独立功能：

```
library(shiny)

myApp <- function(...) {
  ui <- fluidPage(
    ...
  )
  server <- function(input, output, session) {
    ...
  }
  shinyApp(ui, server, ...)
}
```

- 调用 `usethis::use_description()` 创建一个描述文件。在许多情况下，您永远不需要查看此文件，但您需要它来激活RStudio的“软件包开发模式”，该模式提供了我们稍后将使用的键盘快捷键。
- 如果您还没有，请通过调用 `usethis::use_rstudio()` 创建一个RStudio项目。
- 重新启动RStudio并重新打开您的项目。

您现在可以按 `Cmd/Ctrl + Shift + L` 来运行 `devtools::load_all()` 并加载所有软件包代码和数据。这意味着你现在可以：

- 删除对 `source()` 的任何调用，因为 `load_all()` 会自动获取所有调用的源代码。`R/` 中的R文件。
- 如果您正在使用 `read.csv()` 或类似内容加载数据集，则可以使用 `usethis::use_data(mydataset)` 将数据保存在 `data/` 目录中。`load_all()` 会自动为您加载数据。

为了使这个过程更加具体，我们接下来将通过一个简单的案例研究进行工作，然后再回到第20.2节中，了解这项工作的其他好处。

## 20.1.1 Single file

想象一下，我有一个相对复杂的应用程序，目前只存在于一个应用程序中-app.R。

```
library(shiny)

monthFeedbackUI <- function(id) {
  textOutput(NS(id, "feedback"))
}

monthFeedbackServer <- function(id, month) {
  stopifnot(is.reactive(month))

  moduleServer(id, function(input, output, session) {
    output$feedback <- renderText({
      if (month() == "October") {
        "You picked a great month!"
      } else {
        "Eh, you could do better."
      }
    })
  })
}

stones <- vroom::vroom("birthstones.csv")
birthstoneUI <- function(id) {
  p(
    "The birthstone for ", textOutput(NS(id, "month"), inline = TRUE),
    " is ", textOutput(NS(id, "stone"), inline = TRUE)
  )
}

birthstoneServer <- function(id, month) {
  stopifnot(is.reactive(month))

  moduleServer(id, function(input, output, session) {
    stone <- reactive(stones$stone[stones$month == month()])
    output$month <- renderText(month())
    output$stone <- renderText(stone())
  })
}

months <- c(
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
)

ui <- navbarPage(
  "Sample app",
  tabPanel("Pick a month",
    selectInput("month", "What's your favourite month?", choices = months)
  ),
  tabPanel("Feedback", monthFeedbackUI("tab1")),
  tabPanel("Birthstone", birthstoneUI("tab2"))
)
```

```
server <- function(input, output, session) {
  monthFeedbackServer("tab1", reactive(input$month))
  birthstoneServer("tab2", reactive(input$month))
}
shinyApp(ui, server)
```

此代码创建了简单的三页应用程序，使用模块来隔离页面。这是一个玩具应用程序，但它仍然是现实的——与真实应用程序相比，主要区别在于，这里的单个UI和server组件要简单得多。

## 20.1.2 Module files

在将其转换为软件包之前，我的第一步是按照第[19.2.5](#)节中的建议将两个模块拉到自己的文件中：

- `R/monthFeedback.R`：

```
monthFeedbackUI <- function(id) {
  textOutput(NS(id, "feedback"))
}
monthFeedbackServer <- function(id, month) {
  stopifnot(is.reactive(month))

  moduleServer(id, function(input, output, session) {
    output$feedback <- renderText({
      if (month() == "October") {
        "You picked a great month!"
      } else {
        "Eh, you could do better."
      }
    })
  })
}
```

- `R/birthstone.R`：

```
birthstoneUI <- function(id) {
  p(
    "The birthstone for ", textOutput(NS(id, "month"), inline = TRUE),
    " is ", textOutput(NS(id, "stone"), inline = TRUE)
  )
}
birthstoneServer <- function(id, month) {
  stopifnot(is.reactive(month))

  moduleServer(id, function(input, output, session) {
    stone <- reactive(stones$stone[stones$month == month()])
    output$month <- renderText(month())
    output$stone <- renderText(stone())
  })
}
```

这给我留下了以下 `app.R`：

```

library(shiny)

stones <- vroom::vroom("birthstones.csv")
#> Rows: 12 Columns: 2
#> — Column specification —————
#> Delimiter: ","
#> chr (2): month, stone
#>
#> i Use `spec()` to retrieve the full column specification for this data.
#> i Specify the column types or set `show_col_types = FALSE` to quiet this message.
months <- c(
  "January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"
)

ui <- navbarPage(
  "Sample app",
  tabPanel("Pick a month",
    selectInput("month", "What's your favourite month?", choices = months)
  ),
  tabPanel("Feedback", monthFeedbackUI("tab1")),
  tabPanel("Birthstone", birthstoneUI("tab2"))
)

server <- function(input, output, session) {
  monthFeedbackServer("tab1", reactive(input$month))
  birthstoneServer("tab2", reactive(input$month))
}

shinyApp(ui, server)

```

只需将模块拉出到单独的文件中是有用的，因为它可以帮助我了解应用程序的大局。如果我想深入研究细节，我可以查看模块文件。

### 20.1.3 A package

现在让我们把这个做成一个软件包。首先，我运行 `usethis::use_description()` 它创建一个 `DESCRIPTION` 文件。接下来，我将 `app.R` 移动到 `R/app.R`，并将 `shinyApp()` 包装成一个函数：

```

library(shiny)

monthApp <- function(...) {
  stones <- vroom::vroom("birthstones.csv")
  months <- c(
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December"
  )

  ui <- navbarPage(
    "Sample app",
    tabPanel("Pick a month",
      selectInput("month", "What's your favourite month?", choices = months)
    ),

```

```

    tabPanel("Feedback", monthFeedbackUI("tab1")),
    tabPanel("Birthstone", birthstoneUI("tab2"))
  )
  server <- function(input, output, session) {
    monthFeedbackServer("tab1", reactive(input$month))
    birthstoneServer("tab2", reactive(input$month))
  }
  shinyApp(ui, server, ...)
}

```

作为可选的额外内容，我通过运行 `usethis::use_data(stones)` 将 `birthstones.csv` 转换为软件包数据集。这将创建 `data/stones.rda`，当我加载软件包时，它将自动加载。我现在可以删除 `birthstones.csv`，并删除在：  
`stones <- vroom::vroom("birthstones.csv")` 中读取它的行。

您可以在<https://github.com/hadley/monthApp>上查看最终产品。

## 20.2 Benefits

为什么要费心做这些工作？最重要的好处是新的工作流程，可以更轻松地准确地重新加载所有应用程序代码并重新启动应用程序。但它也使在应用程序之间共享代码并与其他人共享应用程序变得更加容易。

### 20.2.1 Workflow

将您的应用程序代码放入软件包结构中会解锁一个新的工作流程：

- 使用 `Cmd/Ctrl + Shift + L` 重新加载应用程序中的所有代码。这调用 `devtools::load_all()`，它自动保存所有打开的文件，`source()` 每一个 R/ 中的文件，加载 `data/` 中的所有数据集，然后将光标放在控制台中。
- 使用 `myApp()` 重新运行该应用程序。

随着您的应用程序越来越大，也值得了解两个最重要的代码导航键盘快捷键：

- `Ctrl/Cmd + .` 将打开“模糊文件和函数查找器”——在要导航到的文件或函数的开头键入几个字母，用箭头键选择它，然后按回车键。这允许您快速跳过应用程序，而无需将手从键盘上拿开。
- 当您的光标位于函数名称上时，`F2` 将跳转到函数定义。

如果您做了大量软件包开发，您可能希望自动加载 `usethis`，因此您可以键入（例如）`use_description()` 而不是 `usethis::use_description()`。您可以通过将以下行添加到您的 `.Rprofile` 中，来做到这一点。此文件包含每当您启动R时都会运行的R代码，因此它是自定义交互式开发环境的好方法。

```

if (interactive()) {
  require(usethis, quietly = TRUE)
}

```

最简单的方法是找到并打开你的 `.Rprofile` 是 `runusethis::edit_r_profile()`

## 20.2.2 Sharing

由于您的应用程序现在包含在一个函数中，因此很容易在同一软件包中包含多个应用程序。由于您在同一个地方有多个应用程序，现在跨应用程序共享代码和数据要容易得多。如果您有大量用于相关任务的应用程序，这是一个巨大的好处。

软件包也是共享应用程序的好方法。[shinyapps.io](https://shinyapps.io)和[RStudio Connect](https://rstudioconnect.com)是与不熟悉R的人共享应用程序的好方法。但有时您想与确实使用R的同事共享应用程序——也许与其允许用户上传数据集，不如为他们提供一个他们用data.frame调用的功能。例如，以下非常简单的应用程序允许R用户为交互式汇总数据，提供自己的数据框：

```
dataSummaryApp <- function(df) {  
  ui <- fluidPage(  
    selectInput("var", "Variable", choices = names(df)),  
    verbatimTextOutput("summary")  
  )  
  
  server <- function(input, output, session) {  
    output$summary <- renderPrint({  
      summary(df[[input$var]])  
    })  
  }  
  
  shinyApp(ui, server)  
}
```

[RStudio Gadgets](#)基于这个想法：它们是Shiny应用程序，允许您向RStudio IDE添加新的用户界面。甚至可以编写生成代码的小工具，因此您可以执行一些易于交互完成的任务，该小工具生成相应的代码并保存回打开的文件中。

## 20.3 Extra steps

除了基础知识之外，您可能会采取两个常见的额外步骤：轻松部署应用程序包，并将其变成“真正的”软件包。

### 20.3.1 Deploying your app-package

如果您想将应用程序部署到RStudio Connect或Shiny，您需要两个额外的步骤：

- 您需要一个 `app.R`，它告诉部署服务器如何运行您的应用程序。最简单的方法是使用pkgload加载代码：

```
pkgload::load_all(".")  
myApp()
```

（您可以在<https://engineering-shiny.org/deploy.html>上查看其他技术）。

- 通常，当您部署应用程序时，rsconnect软件包会自动计算出您的代码使用的所有软件包。但现在您有一个 `DESCRIPTION` 文件，它要求您明确指定它们。最简单的方法是调用 `usethis::use_package()`，您需要从shiny和pkgload开始：

```
usethis::use_package("shiny")  
usethis::use_package("pkgload")
```

这需要更多的工作，但需要花费的时间是在一个地方明确列出您的应用程序需要的每个软件包。



现在，每当您准备与用户共享应用程序的更新版本时，您就可以运行 `rsconnect::deployApp()`。

## 20.3.2 R CMD check

最小的一个R软件包，包含一个 `R/` 目录、一个 `DESCRIPTION` 文件和一个运行应用程序的功能。正如你所看到的，这已经很有用了，因为它解锁了一些有用的工作流程来加快应用程序开发。但是什么造就了“真正的”应用程序？对我来说，它正在认真努力让 `R CMD check` 通过。`R CMD check` 是R的自动化系统，可以检查您的包裹是否存在常见问题。在RStudio中，您可以通过按 `Cmd/Ctrl + Shift + E` 来运行 `R CMD check`。

我不建议你第一次、第二次甚至第三次尝试软件包结构时这样做。相反，我建议您在采取下一步制作完全兼容的软件包之前熟悉基本结构和工作流程。这也是我通常为重要应用程序保留的东西，特别是任何将部署在其他地方的应用程序。让 `R CMD check` 通过可能需要很多工作，短期内几乎没有回报。但从长远来看，这将保护您免受一些潜在问题的影响，并且因为它确保您的应用程序符合R开发人员熟悉的标准，因此其他人更容易为您的应用程序做出贡献。

在制作第一个完整的应用程序包之前，您应该阅读R软件包的“[The whole game](#)”一章：它将让您更全面地了解软件包结构，并向您介绍其他有用的工作流程。然后使用以下提示让 `R CMD check` 干净地通过：

- 删除对 `library()` 或 `require()` 的任何调用，而是用 `DESCRIPTION` 中的声明替换它们。  
`usethis::use_package("name")` 将所需的包添加到 `DESCRIPTION` 中。然后，您需要决定是使用 `::` 显式引用每个函数，还是使用 `@importFrom packageName functionName` 在一个地方声明导入。  
至少，您需要 `usethis::use_package("shiny")`。对于Shiny应用程序，我建议使用 `@import shiny` 来使Shiny软件包中的所有功能都易于使用。（使用 `@import` 通常不被认为是最佳实践，但在这里是有意义的）。
- 选择许可证，然后使用适当的 `use_license_` 函数将其放在正确的位置。对于专有代码，您可以使用 `usethis::use_proprietary_license()`。有关更多详细信息，请参阅<https://r-pkgs.org/license.html>。
- 使用 `usethis::use_build_ignore("app.R")` 或类似的方法，将 `app.R` 添加到 `.Rbuildignore` 中。
- 如果您的应用程序包含小型参考数据集，请将它们放入 `data` 或 `inst/extdata`。我们上面讨论了 `usethis::use_data()`；或者，您可以将原始数据放置于 `inst/ext` 中，并使用 `read.csv(system.file("extdata", "mydata.csv", package = "myApp"))` 或类似方式加载。
- 您还可以更改您的 `app.R`，以使用软件包。这需要您的软件包在部署机器可以安装的地方可用。对于公共工作，这意味着CRAN或GitHub软件包；对于私人工作，这可能意味着使用[RStudio Package Manager](#)或[drat](#)等工具。

```
myApp::myApp()
```

## 20.4 Summary

在本章中，你把脚趾浸入了包装开发的水中。如果您想到ggplot2和shiny软件包，这可能会让人不知所措，但软件包可能非常简单。事实上，所有项目都需要是一个包，是一个R文件和 `DESCRIPTION` 文件的目录。软件包只是一组轻量级的约定，可以解锁有用的工具和工作流程。在本章中，您学习了如何将应用程序转换为软件包，以及您可能想要的一些原因。在下一章中，您将了解将应用程序转换为软件包的最重要原因：使其更容易测试。

## Chapter 21. Testing

对于简单的应用程序，很容易记住应用程序应该如何工作，这样当您进行更改以添加新功能时，您就不会意外破坏现有功能。然而，随着您的应用程序变得越来越复杂，不可能同时将其全部保留在脑海中。测试是捕获代码所需行为的一种方式，您可以自动验证它是否继续以您预期的方式工作。首次将现有的非正式测试转换为代码是痛苦的，因为您需要小心地将每次按键和鼠标点击都变成一行代码，但一旦完成，重新运行测试的速度就会快得多。

我们将使用[testthat](#)软件包进行自动测试。测试需要将您的应用程序变成一个软件包，但正如第20章所讨论的，这不是太多的工作，我认为出于其他原因会得到回报。

测试看起来像这样的测试：

```
test_that("as.vector() strips names", {  
  x <- c(a = 1, b = 2)  
  expect_equal(as.vector(x), c(1, 2))  
})
```

我们很快就会回到细节上来，但请注意，测试从声明意图（“`as.vector()` 删除名称”）开始，然后使用常规R代码生成一些测试数据。然后使用**期望（expectation）**将测试数据与预期结果进行比较，**期望**是一个以 `expect_` 开头的函数。第一个参数是一些要运行的代码，第二个参数描述了预期的结果：在这里，我们验证 `as.vector(x)` 输出是否等于 `c(1, 2)`。

在本章中，我们将完成四个级别的测试：

- 我们将从测试非反应性**函数**开始。这将帮助您了解基本的测试工作流程，并允许您验证从server函数或UI中提取代码的行为。如果您正在编写软件包，这与您所做的测试类型完全相同，因此您可以在R软件包的[测试章节](#)中找到更多详细信息。
- 接下来，您将学习如何测试server函数中的**反应流**。您将设置输入的值，然后验证反应和输出是否具有您期望的值。
- 然后，我们将通过在后台网络浏览器中运行应用程序来测试Shiny中使用**JavaScript**的部分（例如 `update*` 函数）。这是一个高保真度的模拟，因为它运行的是真正的浏览器，但缺点是，测试运行速度较慢，您再也无法如此轻松地在应用程序内窥视。
- 最后，我们将通过保存所选元素的屏幕截图来测试**应用程序的视觉效果**。这是测试应用程序布局、CSS、绘图和HTML小部件所必需的，但很脆弱，因为屏幕截图很容易因多种原因而更改。这意味着需要人为干预来确认每个变化是否正常，使其成为最劳动密集的测试形式。

这些测试级别形成了自然的层次结构，因为每种技术都提供了对应用程序用户体验的更全面的模拟。更好的模拟的缺点是，每个级别都更慢，因为它必须做更多事情，而更脆弱，因为更多的外部力量发挥作用。您应该始终努力在尽可能低的水平上工作，以便您的测试尽可能快速和健壮。随着时间的推移，这也会影响您编写代码的方式：知道哪种代码更容易测试，自然会将其推向更简单的设计。在不同级别的测试之间交错，我还将提供有关测试工作流程和更一般的测试理念的建议。

```
library(shiny)  
library(testthat) # >= 3.0.0  
library(shinytest)
```

## 21.1 Testing functions

应用程序中最容易测试的部分是与Shiny关系最小的部分：从UI和server代码中提取的函数，如第18章所述。我们将首先讨论如何测试这些非反应性函数，向您展示使用testthat进行单元测试的基本结构。

### 21.1.1 Basic structure

测试分为三个级别：

- **文件。**所有测试文件都位于 `tests/testthat` 中，每个测试文件应与 `R/` 中的代码文件相对应，例如 `R/module.R` 中的代码应由位于 `tests/testthat/test-module.R` 的代码进行测试。幸运的是，您不必记住这个惯例：只需使用 `usethis::use_test()` 自动创建或找到与当前打开的R文件相对应的测试文件。
- **测试。**每个文件都分解为测试，即对 `test_that()` 的调用。测试通常应该检查函数的单个属性。很难确切地描述这意味着什么，但一个好的启发式是，你可以很容易地在 `test_that()` 的第一个参数中描述测试。
- **期望。**每个测试都包含一个或多个期望值，其函数以 `expect_` 开头。这些准确地定义了您期望代码做什么，无论是返回特定值、抛出错误还是其他东西。在本章中，我将讨论对Shiny应用程序最重要的期望，但您可以在[测试网站上](#)看到完整列表。

测试的艺术是弄清楚如何编写测试，明确定义函数的预期行为，而不依赖于未来可能改变的附带细节。

### 21.1.2 Basic workflow

现在您了解了基本结构，让我们深入研究一些例子。我将从第[18.3.1](#)节中的一个简单示例开始。在这里，我从我的server函数中提取了一些代码，并称其为 `load_file()`

```
load_file <- function(name, path) {  
  ext <- tools::file_ext(name)  
  switch(ext,  
    csv = vroom::vroom(path, delim = ",", col_types = list()),  
    tsv = vroom::vroom(path, delim = "\t", col_types = list()),  
    validate("Invalid file; Please upload a .csv or .tsv file")  
  )  
}
```

为了这个例子，我将假装这个代码存在于 `R/load.R` 中，因此我的测试需要存在于 `tests/testthat/test-load.R` 中。创建该文件的最简单方法是在 `load.R` 中，运行 `usethis::use_test()`

我想为这个功能测试三件主要事情：它能加载csv文件吗，可以加载tsv文件吗，以及它是否对其他类型提供错误消息？为了测试这三件事，我需要一些示例文件，我将这些文件保存在会话临时目录中，以便在测试运行后自动清理它们。然后我写了三个期望，两个检查加载的文件是否等于原始数据，一个检查我收到错误。

```
test_that("load_file() handles all input types", {
  # Create sample data
  df <- tibble::tibble(x = 1, y = 2)
  path_csv <- tempfile()
  path_tsv <- tempfile()
  write.csv(df, path_csv, row.names = FALSE)
  write.table(df, path_tsv, sep = "\t", row.names = FALSE)

  expect_equal(load_file("test.csv", path_csv), df)
  expect_equal(load_file("test.tsv", path_tsv), df)
  expect_error(load_file("blah", path_csv), "Invalid file")
})
#> Test passed 🎉
```

有四种方法可以运行此测试：

- 当我开发它时，我在控制台上交互式运行每条线。当期望失败时，它会变成一个错误，然后我来修复它。
- 一旦我完成开发，我就会运行整个测试块。如果测试通过，我会收到一条消息，比如 `Test passed` 🎉。如果失败了，我会得到出错的细节。
- 随着我开发更多的测试，我使用 `devtools::test_file()` 运行当前file的所有测试。因为我经常这样做，所以我设置了一个特殊的键盘快捷键，使其尽可能简单。我很快就会告诉你如何自己设置。
- 时不时地，我会用 `devtools::test()` 运行整个软件包的所有测试。这确保了我没有意外损坏当前文件以外的任何东西。

### 21.1.3 Key expectations

在测试函数时，您会花很多时间进行两种预期：`expect_equal()` 和 `expect_error()`。像所有期望函数一样，第一个参数是要检查的代码，第二个参数是预期结果：在 `expect_equal()` 的情况下是预期值，在 `expect_error()` 的情况下是预期错误文本。

为了了解这些函数的工作原理，在测试之外直接调用它们是有用的。

使用 `expect_equal()` 时，请记住，您不必测试整个对象：一般来说，最好只测试您感兴趣的组件：

```
complicated_object <- list(
  x = list(mtcars, iris),
  y = 10
)
expect_equal(complicated_object$y, 10)
```

对 `expect_equal()` 的特殊情况有一些期望，可以节省您一些打字

- `expect_true(x)` 和 `expect_false(x)` 等价于 `expect_equal(x, FALSE)` 和 `expect_equal(x, TRUE)`。  
`expect_null(x)` 等价于 `expect_equal(x, NULL)`
- `expect_named(x, c("a", "b", "c"))` 等价于 `expect_equal(names(x), c("a", "b", "c"))`，但拥有可选的选项 `ignore.order` 和 `ignore.case`。`expect_length(x, 10)` 等价于 `expect_equal(length(x), 10)`。

还有一些函数为向量实现 `expect_equal()` 的轻松版本：

- `expect_setequal(x, y)` 测试 `x` 中的每个值都发生在 `y` 中，`y` 中的每个值都发生在 `x` 中。
- `expect_mapequal(x, y)` 测试 `x` 和 `y` 具有相同的名称，并且 `x[names(y)]` 等于 `y`。

测试代码生成错误通常很重要，您可以使用 `expect_error()`

```
expect_error("Hi!")
#> Error: "Hi!" did not throw the expected error.
expect_error(stop("Bye"))
```

请注意，`expect_error()` 的第二个参数是一个正则表达式——目标是找到一个与您预期的错误相匹配的简短文本片段，并且不太可能匹配您预期的错误。

```
f <- function() {
  stop("Calculation failed [location 1]")
}

expect_error(f(), "Calculation failed [location 1]")
#> Error in f(): Calculation failed [location 1]
expect_error(f(), "Calculation failed \\[location 1\\]")
```

但最好还是选择一个小片段来匹配：

```
expect_error(f(), "Calculation failed")
```

或者使用 `expect_snapshot()`，我们稍后会讨论。`expect_error()` 还附带变体 `expect_warning()` 和 `expect_message()`，以与错误相同的方式测试警告和消息。测试shiny应用程序很少需要这些，但对测试软件包非常有用。

## 21.1.4 User interface functions

您可以使用相同的基本想法来测试您从UI代码中提取的函数。但这些需要一个新的期望，因为手动键入所有HTML会很乏味，所以我们使用快照测试（snapshot test）。快照期望与其他期望的主要不同之处在于，预期结果存储在单独的快照文件中，而不是代码本身。当您设计复杂的用户界面设计系统时，快照测试最有用，这超出了大多数应用程序的范围。因此，在这里，我将简要向您展示关键想法，然后向您指出其他资源以了解更多信息。

以我们之前定义的这个UI功能为例：

```
sliderInput01 <- function(id) {
  sliderInput(id, label = id, min = 0, max = 1, value = 0.5, step = 0.1)
}

cat(as.character(sliderInput01("x")))
#> <div class="form-group shiny-input-container">
#>   <label class="control-label" id="x-label" for="x">x</label>
#>   <input class="js-range-slider" id="x" data-skin="shiny" data-min="0" data-max="1"
data-from="0.5" data-step="0.1" data-grid="true" data-grid-num="10" data-grid-snap="false"
data-prettyify-separator="," data-prettyify-enabled="true" data-keyboard="true" data-data-
type="number"/>
#> </div>
```

我们如何测试这个输出是否像我们预期的那样？我们可以使用 `expect_equal()`

```
test_that("sliderInput01() creates expected HTML", {
  expect_equal(as.character(sliderInput01("x")), "<div class=\"form-group shiny-input-
container\">\n  <label class=\"control-label\" id=\"x-label\" for=\"x\">x</label>\n
<input class=\"js-range-slider\" id=\"x\" data-skin=\"shiny\" data-min=\"0\" data-
max=\"1\" data-from=\"0.5\" data-step=\"0.1\" data-grid=\"true\" data-grid-num=\"10\"
data-grid-snap=\"false\" data-prettify-separator=\", \" data-prettify-enabled=\"true\"
data-keyboard=\"true\" data-data-type=\"number\"/>\n</div>")
})
#> Test passed 🍌
```

但引号和换行符的存在需要在字符串中进行大量的转义——这使得我们很难看到我们期望的确切内容，如果输出发生变化，就很难看到到底发生了什么。

快照测试的关键思想是将预期结果存储在一个单独的文件中：这可以防止笨重的数据出现在测试代码之外，这意味着您不必担心在字符串中逃避特殊值。在这里，我们使用 `expect_snapshot()` 来捕获控制台上显示的输出：

```
test_that("sliderInput01() creates expected HTML", {
  expect_snapshot(sliderInput01("x"))
})
```

与其他期望的主要区别在于，没有第二个论点来描述你期望看到的东西。相反，该数据保存在单独的 `.md` 文件中。如果您的代码在 `R/slider.R` 中，并且您的测试在 `tests/testthat/test-slider.R` 中，那么快照将保存在 `tests/testthat/_snaps/slider.md`。首次运行测试时，`expect_snapshot()` 将自动创建参考输出，其外观如下：

```
# sliderInput01() creates expected HTML

Code
  sliderInput01("x")
Output
<div class="form-group shiny-input-container">
  <label class="control-label" id="x-label" for="x">x</label>
  <input class="js-range-slider" id="x" data-skin="shiny" data-min="0" data-max="1"
data-from="0.5" data-step="0.1" data-grid="true" data-grid-num="10" data-grid-snap="false"
data-prettify-separator="," data-prettify-enabled="true" data-keyboard="true" data-data-
type="number"/>
</div>
```

如果输出稍后发生变化，测试将失败。您要么需要修复导致其失败的错误，要么如果是故意更改，请通过 `testthat::snapshot_accept()` 更新快照。

在将其作为测试之前，值得考虑这里的输出。你在这里到底在测试什么？如果您查看输入如何成为输出，您会注意到大多数输出是由Shiny生成的，只有非常少量是您的代码的结果。这表明这个测试不是特别有用：如果这个输出发生变化，它更可能是更改为Shiny的结果，而不是更改代码的结果。这使得测试变得脆弱；如果它失败了，那不太可能是你的错，修复失败也不太可能在你的控制范围内。

您可以在<https://testthat.r-lib.org/articles/snapshotting.html>上了解有关快照测试的更多信息。



## 21.2 Workflow

在我们谈论使用反应性或JavaScript的测试函数之前，我们将进行简短的题外话来处理您的工作流程。

### 21.2.1 Code coverage

验证您的测试是否测试了您认为它们正在测试的内容非常有用。做到这一点的一个好方法是使用“代码覆盖（code coverage）”，它运行您的测试并跟踪运行的每一行代码。然后，您可以查看结果，看看哪些代码行从未被测试触及，并让您有机会反思您是否测试了代码中最重要、风险最高或最难编程的部分。它不能代替探究你的代码——你可以有100%的测试覆盖率，但仍然有错误。但它是一个有趣和有用的工具，可以帮助你思考什么是重要的，特别是当你有复杂的嵌套代码时。

这里不会详细介绍，但我强烈建议使用 `devtools::test_coverage()` 或 `devtools::test_coverage_file()` 来尝试。需要注意的主要事情是绿线是经过测试的；红线不是。

代码覆盖支持略微不同的工作流程：

1. 使用 `test_coverage()` 或 `test_coverage_file()` 查看测试了哪些代码行。
2. 查看未经测试的线条和专门用于测试的设计测试。
3. 重复，直到测试所有重要的代码行。（通常获得100%的测试覆盖率是不值得的，但您应该检查您是否击中了应用程序的最关键部分）

代码覆盖率还适用于测试反应性和（在某种程度上）JavaScript的工具，因此它是一种有用的基础技能。

### 21.2.2 Keyboard shortcuts

如果您遵循了第20.2节中的建议，那么您已经可以通过在控制台上键入 `test()` 或 `test_file()` 来运行测试。但测试是你会经常做的事情，值得让键盘快捷键触手可及。RStudio内置了一个有用的快捷方式：`Cmd/Ctrl + Shift + T` 运行 `devtools::test()`。我建议你自己加三个来完成这个套装：

- 绑定 `Cmd/Ctrl + T` 到 `devtools::test_file()`
- 绑定 `Cmd/Ctrl + Shift + R` 到 `devtools::test_coverage()`
- 绑定 `Cmd/Ctrl + R` 到 `devtools::test_coverage_file()`

当然，您可以自由选择任何对您有意义的快捷方式，但这些都有一些共同的基本结构。使用Shift的键盘快捷键适用于整个软件包，不带Shift的键盘快捷键适用于当前文件。

图21.1显示了我的键盘快捷键在Mac上的样子。

manage visual test cases		Assign
Report test coverage for a file	Cmd+R	Addin
Report test coverage for a package	Shift+Cmd+R	Addin
Run a test file	Cmd+T	Addin
Visual test Run		Addin

图21.1：我的Mac键盘快捷键。

### 21.2.3 Workflow summary

以下是我到目前为止谈论过的所有技术的摘要：

- 从R文件中，使用 `usethis::use_test()` 创建测试文件（首次运行时）或导航到测试文件（如果已经存在）。

- 编写代码/编写测试。按 `cmd/ctrl + T` 在控制台中运行测试并查看结果。根据需要进行。
- 如果您遇到新错误，请从在测试中捕获不良行为开始。在制作最小代码的过程中，您通常会更好地了解错误的位置，并且进行测试将确保您不会自欺欺人地认为自己已经修复了错误。
- 按 `ctrl/cmd + R` 检查您是否正在测试您认为正在测试的内容
- 按 `ctrl/cmd + shift + T`，让您执行完整的测试。

## 21.3 Testing reactivity

现在您已经了解了如何测试常规、非反应性代码，是时候继续应对Shiny特有的挑战了。第一个挑战是测试反应性。正如您已经看到的，您无法以交互方式运行反应式代码：

```
x <- reactive(input$y + input$z)
x()
#> Error in `.getReactiveEnvironment()$currentContext()` :
#> ! Operation not allowed without an active reactive context.
#> • You tried to do something that can only be done from inside a reactive consumer.
```

您可能想知道如何使用 `reactiveConsole()`，就像我们在第15章中所做的那样。不幸的是，它的反应性模拟从根本上依赖于交互式控制台，因此在测试中不起作用。

当我们试图评估它时，不仅会出现反应性错误，即使它确实工作，`input$y` 和 `input$z` 也不会被定义。要了解它的工作原理，让我们从一个简单的应用程序开始，它有三个输入、一个输出和三个反应：

```
ui <- fluidPage(
  numericInput("x", "x", 0),
  numericInput("y", "y", 1),
  numericInput("z", "z", 2),
  textOutput("out")
)
server <- function(input, output, session) {
  xy <- reactive(input$x - input$y)
  yz <- reactive(input$z + input$y)
  xyz <- reactive(xy() * yz())
  output$out <- renderText(paste0("Result: ", xyz()))
}
```

要测试此代码，我们将使用 `testServer()`，该函数需要两个参数：一个server函数和一些要运行的代码。代码在server函数内的特殊环境中运行，因此您可以访问输出、反应和允许您模拟用户交互的特殊 `session` 对象。您将花费主要的时间使用 `session$setInputs()`，它允许您设置输入控件的值，就像您是在浏览器中与应用程序交互的用户一样。

```
testServer(server, {
  session$setInputs(x = 1, y = 1, z = 1)
  print(xy())
  print(output$out)
})
#> [1] 0
#> [1] "Result: 0"
```



(您可以滥用 `testServer()`，进入支持反应的交互式环境：`testServer(myApp(), browser())`)

请注意，我们只是在测试`server`函数；应用程序的`ui`组件被完全忽略。通过检查输入，您可以最清楚地看到这一点：与真正的Shiny应用程序不同，所有输入都以`NULL`开头，因为初始值记录在`ui`中。我们将回到第21.4节中的UI测试。

```
testServer(server, {
  print(input$x)
})
#> NULL
```

现在您有了在反应环境中运行代码的方法，您可以将其与您已经了解的测试代码相结合，以创建以下内容：

```
test_that("reactives and output updates", {
  testServer(server, {
    session$setInputs(x = 1, y = 1, z = 1)
    expect_equal(xy(), 0)
    expect_equal(yz(), 2)
    expect_equal(output$out, "Result: 0")
  })
})
#> Test passed 🎉
```

一旦您掌握了`testServer()`的使用，那么测试反应性代码几乎和测试非反应性代码一样容易。主要挑战是调试失败的测试：您无法像常规测试一样逐行完成它们，因此您需要在`testServer()`中添加`browser()`以便进行交互式实验来诊断问题。

## 21.3.1 Modules

您可以以类似于测试应用程序功能的方式测试模块，但在这里更清楚的是，您只是在测试模块的服务器端。让我们从一个简单的模块开始，该模块使用三个输出来显示变量的简要摘要：

```
summaryUI <- function(id) {
  tagList(
    outputText(ns(id, "min")),
    outputText(ns(id, "mean")),
    outputText(ns(id, "max")),
  )
}
summaryServer <- function(id, var) {
  stopifnot(is.reactive(var))

  moduleServer(id, function(input, output, session) {
    range_val <- reactive(range(var(), na.rm = TRUE))
    output$min <- renderText(range_val()[[1]])
    output$max <- renderText(range_val()[[2]])
    output$mean <- renderText(mean(var()))
  })
}
```

我们将如上所述使用 `testServer()`，但调用有点不同。和以前一样，第一个参数是server函数（现在是模块server），但现在我们还需要在名为 `args` 的列表提供额外的参数。这需要将参数列表带到模块server中（`id` 参数是可选的；如果省略，`testServer()` 将自动填充它）。然后我们完成要运行的代码：

```
x <- reactiveVal(1:10)
testServer(summaryServer, args = list(var = x), {
  print(range_val())
  print(output$min)
})
#> [1] 1 10
#> [1] "1"
```

同样，我们可以通过将其放入 `test_that()`，并调用一些 `expect_` 函数来将其转换为自动测试。在这里，我把所有内容都打包成一个测试，检查模块在反应性输入变化时是否正确响应：

```
test_that("output updates when reactive input changes", {
  x <- reactiveVal()
  testServer(summaryServer, args = list(var = x), {
    x(1:10)
    session$flushReact()
    expect_equal(range_val(), c(1, 10))
    expect_equal(output$mean, "5.5")

    x(10:20)
    session$flushReact()
    expect_equal(range_val(), c(10, 20))
    expect_equal(output$min, "10")
  })
})
#> Test passed 🌈
```

这里有一个重要的技巧：因为 `x` 是在 `testServer()` 之外创建的，更改 `x` 不会自动更新反应图，所以我们必须通过调用 `session$flushReact()` 手动更新。

如果您的模块具有返回值（反应值或反应性列表），您可以使用 `session$getReturned()` 捕获它。然后，您可以检查该反应的值，就像任何其他反应一样。

```
datasetServer <- function(id) {
  moduleServer(id, function(input, output, session) {
    reactive(get(input$dataset, "package:datasets"))
  })
}

test_that("can find dataset", {
  testServer(datasetServer, {
    dataset <- session$getReturned()

    session$setInputs(dataset = "mtcars")
    expect_equal(dataset(), mtcars)
  })
})
```

```
session$setInputs(dataset = "iris")
expect_equal(dataset(), iris)
})
})
#> Test passed 🐱
```

我们需要测试如果 `input$dataset` 不是数据集会发生什么吗？在这种情况下，我们没有，因为我们知道模块UI将选项限制为有效选择。仅从server函数的检查中，这一点并不明显。

## 21.3.2 Limits

`testServer()` 是您应用程序的模拟。该模拟很有用，因为它允许您快速测试反应性代码，但它并不完整。

- 与现实世界不同，时间不会自动前进。因此，如果您想测试依赖 `reactiveTimer()` 或 `invalidateLater()` 的代码，您需要通过调用 `session$elapse(millis = 300)` 来手动提前时间。
- `testServer()` 忽略UI。这意味着输入不会得到默认值，并且JavaScript不起作用。最重要的是，这意味着您无法测试 `update*` 函数，因为它们通过将JavaScript发送到浏览器来模拟用户交互来工作。您需要下一个技术来测试此类代码。

## 21.4 Testing JavaScript

`testServer()` 只是对整个Shiny应用程序的有限模拟，因此任何依赖“真实”浏览器运行的代码都无法工作。最重要的是，这意味着不会运行JavaScript。这可能看起来并不重要，因为我们在本书中没有讨论过JavaScript，但有一些重要的Shiny函数在幕后使用它：

- 所有 `update*()` 函数，第[10.1节](#)。
- `showNotification()` / `removeNotification()` 第[8.2节](#)。
- `showModal()` / `hideModal()` 第[8.4.1节](#)。
- `insertUI()` / `removeUI()` / `appendTab()` / `insertTab()` / `removeTab()`，我们将在书的后面介绍。

要测试这些功能，您需要在真正的浏览器中运行Shiny应用程序。当然，您可以使用 `runApp()` 并四处单击来完成此操作，但我们希望该过程自动化，以便您经常运行测试。我们将通过[shinytest](#)包的无标签应用做到这一点。您可以按照网站的建议使用shinytest，使用应用程序自动生成测试代码，但由于您已经熟悉testthat，我们将采取不同的方法，手动构建测试。

我们将使用shinytest软件包中的一个R6对象：`ShinyDriver`。创建新的 `ShinyDriver` 实例会启动一个新的R进程，该过程运行您的Shiny应用程序和**headless**浏览器。headless浏览器的工作原理与普通浏览器一样，但它没有可以交互的窗口；唯一的交互方式是通过代码。这种技术的主要缺点是比其他方法慢（即使是最简单的应用程序也至少需要一秒钟），而且您只能测试应用程序的外部（即更难看到反应变量的值）。

### 21.4.1 Basic operation

为了演示基本操作，我将创建一个非常简单的应用程序，用名字问候您，并提供重置按钮。

```

ui <- fluidPage(
  textInput("name", "What's your name"),
  textOutput("greeting"),
  actionButton("reset", "Reset")
)
server <- function(input, output, session) {
  output$greeting <- renderText({
    req(input$name)
    paste0("Hi ", input$name)
  })
  observeEvent(input$reset, updateTextInput(session, "name", value = ""))
}

```

要使用shinytest，您可以使用 `app <- ShinyDriver$new()` 启动应用程序，使用 `app$setInputs()`，与它交互，然后获取 `app$getValue()` 返回的值：

```

app <- shinytest::ShinyDriver$new(shinyApp(ui, server))
app$setInputs(name = "Hadley")
app$getValue("greeting")
#> [1] "Hi Hadley"
app$click("reset")
app$getValue("greeting")
#> [1] ""

```

shinytest的每次使用都从使用 `ShinyDriver$new()` 创建一个ShinyDriver对象开始，该对象采用Shiny应用程序对象或Shiny应用程序的路径。它返回一个R6对象，您与上面遇到的会话对象非常相似，使用 `app$setInputs()` 它需要一组名称-值对，更新浏览器中的控件，然后等待所有反应更新完成。

第一个区别是，您需要使用 `app$getValue(name)` 显式检索值。与 `testServer()` 不同，您无法使用ShinyDriver访问反应值，因为它只能看到应用程序用户可以看到的内容。但有一个特殊的shiny函数，叫做 `exportTestValues()` 它创造了一个特殊的输出，shiny的测试可以看到，但人类看不到。

还有另外两种方法允许您模拟其他操作：

- `app$click(name)` 单击一个名为 `name` 的按钮。
- `app$sendKeys(name, keys)` 将按键发送到名为 `name.keys` 的输入控件，通常会像 `app$sendKeys(id, "Hi!")` 一样的字符串。但您也可以使用 `webdriver::key`，`app$sendKeys(id, c(webdriver::key$control, "x"))` 发送特殊密钥。请注意，任何修饰键都将应用于所有后续按键，因此，如果您想要一些带有修饰键的按键和没有修饰符的按键，则需要多次调用。

通过 `?ShinyDriver` 了解更多详细信息，以及更深奥的方法列表。

和以前一样，一旦您以交互方式找出了适当的操作顺序，您可以通过包装 `test_that()` 并调用期望来将其转换为测试：

```
test_that("can set and reset name", {
  app <- shinytest::ShinyDriver$new(shinyApp(ui, server))
  app$setInputs(name = "Hadley")
  expect_equal(app$getValue("greeting"), "Hi Hadley")

  app$click("reset")
  expect_equal(app$getValue("greeting"), "")
})
```

当垃圾收集器删除和收集 `app` 对象时，后台shiny的应用程序和网络浏览器会自动关闭。如果您不熟悉这意味着什么，您可能会发现<https://adv-r.hadley.nz/names-values.html#gc>有帮助。

## 21.4.2 Case study

我们将以案例研究结束，探索如何测试一个更现实的示例，结合 `testServer()` 和 `shinytest`。我们将使用一个单选按钮控件，该控件还提供一个自由文本“其他”选项。这可能看起来很熟悉，因为我们之前把它作为在[19.4.1](#)中开发模块的动机。

```
ui <- fluidPage(
  radioButtons("fruit", "What's your favourite fruit?",
    choiceNames = list(
      "apple",
      "pear",
      textInput("other", label = NULL, placeholder = "Other")
    ),
    choiceValues = c("apple", "pear", "other")
  ),
  textOutput("value")
)

server <- function(input, output, session) {
  observeEvent(input$other, ignoreInit = TRUE, {
    updateRadioButtons(session, "fruit", selected = "other")
  })

  output$value <- renderText({
    if (input$fruit == "other") {
      req(input$other)
      input$other
    } else {
      input$fruit
    }
  })
}
```

实际计算非常简单。我们可以考虑将 `renderText()` 表达式拉到它自己的函数中：

```
other_value <- function(fruit, other) {
  if (fruit == "other") {
    other
  } else {
    fruit
  }
}
```

但我认为这不值得，因为这里的逻辑非常简单，不能推广到其他情况。我认为将此代码从应用程序中提取到一个单独的文件中的净效果是使代码更难阅读。

因此，我们将从测试反应性的基本流程开始：在将 `fruit` 设置为现有选项后，我们是否得到了正确的值？在将水果设置为其他并添加一些自由文本后，我们是否得到了正确的值？

```
test_that("returns other value when primary is other", {
  testServer(server, {
    session$setInputs(fruit = "apple")
    expect_equal(output$value, "apple")

    session$setInputs(fruit = "other", other = "orange")
    expect_equal(output$value, "orange")
  })
})
#> Test passed 🎉
```

当我们开始在另一个框中键入时，这不会检查其他是否自动选择。我们无法使用 `testServer()` 进行测试，因为它依赖于 `updateRadioButtons()`

```
test_that("returns other value when primary is other", {
  testServer(server, {
    session$setInputs(fruit = "apple", other = "orange")
    expect_equal(output$value, "orange")
  })
})
#> — Failure (<text>:2:3): returns other value when primary is other —————
#> output$value (`actual`) not equal to "orange" (`expected`).
#>
#> `actual`:  "apple"
#> `expected`: "orange"
#> Backtrace:
#>   1. shiny::testServer(...)
#>  22. testthat::expect_equal(output$value, "orange")
#> Error in `reporter$stop_if_needed()`:
#> ! Test failed
```

所以现在我们需要使用ShinyDriver:

```
test_that("automatically switches to other", {
  app <- ShinyDriver$new(shinyApp(ui, server))
  app$setInputs(other = "orange")
  expect_equal(app$getValue("fruit"), "other")
  expect_equal(app$getValue("value"), "orange")
})
```

一般来说，您最好尽可能多地使用 `testServer()`，并且只对需要真实浏览器的位使用 `ShinyDriver`。

## 21.5 Testing visuals

很难使用代码描述正确外观的绘图或HTML小部件等组件呢？您可以使用最终、最丰富和最脆弱的测试技术：保存受影响组件的屏幕截图。这结合了来自shinytest的屏幕截图和来自testthat的整个文件快照。它的工作原理与第21.1.4节中描述的快照类似，但它没有将文本保存到 `.md` 文件中，而是创建一个 `.png` 文件。这也意味着无法在控制台上看到差异，因此系统会提示您运行 `testthat::snapshot_review()`，它使用shiny应用程序来可视化差异。

使用屏幕截图进行测试的主要缺点是，即使是最微小的更改也需要人工确认它没问题。这是一个问题，因为很难让不同的计算机生成像素可重现的屏幕截图。操作系统、浏览器版本甚至字体版本的差异可能导致屏幕截图在人类中看起来相同，但略有不同。这通常意味着视觉测试最好由一个人在本地计算机上运行，在持续集成工具中运行它们通常不值得。绕过这些问题是可能的，但这是相当大的挑战，超出了这本书的范围。

在shinytest中截屏单个元素和在test中截屏整个文件，这都是非常新的功能，我们仍然不清楚什么是理想的界面。因此，现在您需要自己将碎片串在一起，使用以下代码：

```
path <- tempfile()
app <- ShinyDriver$new(shinyApp(ui, server))

# Save screenshot to temporary file
app$takeScreenshot(path, "plot")
#
expect_snapshot_file(path, "plot-init.png")

app$setValue(x = 2)
app$takeScreenshot(path, "plot")
expect_snapshot_file(path, "plot-update.png")
```

`expect_snapshot_file()` 的第二个参数给出了图像将保存在文件快照目录中的文件名。如果这些测试在名为 `test-app.R` 的文件中，那么这两个文件快照将保存在 `tests/testthat/_snaps/app/plot-init.png` 和 `tests/testthat/_snaps/app/plot-update.png`。您希望这些文件的名称简短，但足以提醒您，如果出现问题，您正在测试什么。

## 21.6 Philosophy

本文档主要侧重于测试的机制，当您开始测试时，这些机制是最重要的。但你很快就会了解力学，你的问题将变得更加结构化和哲学化。

我认为考虑假阳性和假阴性是有用的：有可能在应该的时候不失败，在不应该失败的时候不失败。我认为，当你开始测试时，你最大的挣扎是假阳性：你如何确保你的测试实际上抓住了不良行为。但我认为你很快就过去了。



## 21.6.1 When should you write tests?

你应该什么时候写测试？有三个基本选择

- **在你写代码之前。**这是一种称为测试驱动开发的代码风格，如果您确切地知道函数应该如何行为，那么在开始编写实现之前将该知识作为代码捕获是有意义的。
- **在你写完代码之后。**在编写代码时，您通常会建立一份关于代码的担忧的心理待办事项列表。编写完函数后，将这些转换为测试，以便您可以确信该函数以您预期的方式工作。

当你开始编写测试时，小心写得太早。如果您的功能仍在积极发展，让您的测试与所有变化保持同步会令人沮丧。这可能表明你需要再等一会儿。

- **当你发现一个错误时。**每当您发现错误时，最好将其转换为自动测试用例。这有两个优点。首先，为了制作一个好的测试用例，您需要坚持不懈地简化问题，直到您有一个可以包含在测试中的最小重复。其次，您将确保错误永远不会再次出现！

## 21.7 Summary

本章向您展示了如何将应用程序组织成一个软件包，以便您可以利用testthat软件包提供的强大工具。如果您以前从未制作过软件包，这似乎势不可挡，但正如您所看到的，软件包只是一组简单的惯例，您可以轻松适应shiny应用程序。这需要一些前期工作，但会带来巨大的回报：自动测试的能力从根本上提高了您编写复杂应用程序的能力。

在下一章中，您将学习如何找出是什么使您的应用程序变慢，以及一些使其更快的技术。

## Chapter 22. Security

大多数shiny应用程序都部署在公司防火墙中，由于您通常可以假设您的同事不会尝试入侵您的应用程序，因此您不需要考虑安全性。但是，如果您的应用程序包含只有您的一些同事才能访问的数据，或者您想向公众公开您的应用程序，您将需要花一些时间进行安全处理。在保护您的应用程序时，有两件主要需要保护：

- **您的数据：**您希望确保攻击者无法访问任何敏感数据。
- **您的计算资源：**您希望确保攻击者不能开采比特币或将您的服务器用作垃圾邮件农场的一部分。

幸运的是，你的工作变得容易了一点，因为安全是一项团队运动。无论谁部署您的应用程序，都要负责应用程序之间的安全性，确保应用程序A无法访问应用程序B中的代码或数据，并且无法窃取服务器上的所有内存和计算能力。您的责任是应用程序内的安全性，确保攻击者不能滥用您的应用程序来实现他们的目的。本章将介绍保护您的Shiny的基础知识，细分为保护您的数据和保护您的计算资源。

如果您有兴趣了解更多关于安全和R的一般知识，我强烈推荐Colin Gillespie的娱乐和教育性使用R！2019年谈话，[“R和安全”](#)。

```
library(shiny)
```



## 22.1 Data

最敏感的数据是个人身份信息（PII）、受监管数据、信用卡数据、健康数据，或者任何其他如果被公开将成为公司法律噩梦的数据。幸运的是，大多数Shiny应用程序都不处理这些类型的数据，但有一种重要的数据类型你确实需要担心：密码。您不应该在应用程序的源代码中包含密码。相反，可以将它们放在环境变量中，或者如果有很多，则使用config包。无论哪种方式，通过向.gitignore添加适当的文件，确保它们永远不会包含在源代码控制中。我还建议记录新开发人员如何获得适当的证书。

或者，您可能有特定于用户的数据。如果您需要对用户进行身份验证，即通过用户名和密码识别他们，切勿尝试自己滚动解决方案。有太多事情可能会出错。相反，您需要与您的IT团队合作来设计一个安全的访问机制。您可以在<https://solutions.rstudio.com/sys-admin/auth/kerberos/>和<https://db.rstudio.com/best-practices/deployment/>上查看一些最佳实践。请注意，`server()`中的代码是隔离的，因此一个用户会话无法查看来自另一个用户会话的数据。唯一的例外是，如果您使用缓存——有关详细信息，请参阅第23.5.5节。

最后，请注意，Shiny输入使用客户端验证，即有效输入的检查由浏览器中的JavaScript执行，而不是R执行。这意味着知识渊博的攻击者有可能发送您意想不到的值。例如，以这个简单的应用程序为例：

```
secrets <- list(
  a = "my name",
  b = "my birthday",
  c = "my social security number",
  d = "my credit card"
)

allowed <- c("a", "b")
ui <- fluidPage(
  selectInput("x", "x", choices = allowed),
  textOutput("secret")
)
server <- function(input, output, session) {
  output$secret <- renderText({
    secrets[[input$x]]
  })
}
```

您可能期望用户可以访问我的名字和生日，但不能访问我的社会保险号或信用卡详细信息。但知识渊博的攻击者可以在浏览器中打开JavaScript控制台，并运行`Shiny.setInputValue("x", "c")`来查看我的SSN。因此，为了安全起见，您需要检查R代码中的所有用户输入：

```
server <- function(input, output, session) {
  output$secret <- renderText({
    req(input$x %in% allowed)
    secrets[[input$x]]
  })
}
```

我故意没有创建用户友好的错误消息——你唯一一次看到它是当你试图破坏应用程序时，我们不需要帮助攻击者。

## 22.2 Compute resources

希望很明显，以下应用程序非常危险，因为它允许用户运行他们想要的任何R代码。他们可以删除重要文件、修改数据或将机密数据发回给应用程序的用户。

```
ui <- fluidPage(
  textInput("code", "Enter code here"),
  textOutput("results")
)
server <- function(input, output, session) {
  output$results <- renderText({
    eval(parse(text = input$code))
  })
}
```

一般来说，`parse()` 和 `eval()` 的组合是任何Shiny app的一个很大的警告标志：它们会立即使您的应用程序易受攻击。同样，您永远不应该 `source()` 上传的 `.R` 文件，或 `rmarkdown::render()` 上传 `.Rmd`。但这些案例非常明显，不太可能成为真正问题的根源。

更大的挑战之所以出现，是因为许多函数以你不知道的方式 `parse()` 和 `eval()` 或两者兼而有之。以下是最常见的：

- **模型公式 (model formulas)**。可以构造一个执行任意R代码的模型：

```
df <- data.frame(x = 1:5, y = runif(5))
mod <- lm(y ~ {print("Hi!"); x}, data = df)
#> [1] "Hi!"
```

这使得很难安全地允许用户定义自己的模型。

- **胶水标签 (Glue labels)**。glue包提供了一种从数据创建字符串的强大方法：

```
title <- "foo"
number <- 1
glue::glue("{title}-{number}")
#> foo-1
```

但 `glue()` 评估 `{}` 内部的任何东西：

```
glue::glue("{title}-{print('Hi'); number}")
#> [1] "Hi"
#> foo-1
```

如果您想允许用户提供胶水字符串来生成标签，请改用 `glue::glue_safe()`，它只查找变量名，不计算代码：

```
glue::glue_safe("{title}-{number}")
#> foo-1
glue::glue_safe("{title}-{print('Hi'); number}")
#> Error: object 'print('Hi'); number' not found
```

- **变量变换 (variable transformation)**。没有办法安全地允许使用提供代码片段来转换dplyr或ggplot2的变量。你可能期望他们会写 `log10(x)`，但他们可以写 `{print("Hi"); log10(x)}`

这也意味着您永远不应该将旧的 `ggplot2::aes_string()` 与用户提供的输入一起使用。相反，坚持第12章中的技巧。

SQL也可能出现同样的问题。例如，如果您用 `paste()` 构造SQL，例如：

```
find_student <- function(name) {
  paste0("SELECT * FROM Students WHERE name = ('", name, "');")
}
find_student("Hadley")
#> [1] "SELECT * FROM Students WHERE name = ('Hadley');"
```

攻击者可以提供恶意用户名：

```
find_student("Robert'); DROP TABLE Students; --")
#> [1] "SELECT * FROM Students WHERE name = ('Robert'); DROP TABLE Students; --';"
```

这看起来有点奇怪，但它是一个有效的SQL查询，分为三部分：

- `SELECT * FROM Students WHERE name = ('Robert');` 找到一个叫罗伯特的学生。
- `DROP TABLE Students;` 删除 `Students` 表 (! ! )。
- `--'` 是防止额外 `'` 变成语法错误所需的注释。

为了避免这个问题，永远不要用粘贴生成SQL字符串，而是使用自动转义用户输入的系统（如 [dbplyr](#)），或使用 `glue::glue_sql()`

```
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
find_student <- function(name) {
  glue::glue_sql("SELECT * FROM Students WHERE name = ({name});", .con = con)
}
find_student("Robert'); DROP TABLE Students; --")
#> <SQL> SELECT * FROM Students WHERE name = ('Robert'); DROP TABLE Students; --';
```

乍一看有点难分辨，但这是安全的，因为SQL等价于“是”，因此查询返回学生表的所有行，其中名称字面意思是“罗伯特”；`DROP TABLE Students; -`。

## Chapter 23. Performance

如果开发方式正确，Shiny应用程序可以支持数千或数万用户。但大多数Shiny应用程序会迅速组合在一起，以解决紧迫的分析需求，并且通常在性能不佳的情况下开始使用。这是Shiny的一个功能：它允许您快速制作适合您的概念验证原型，然后再弄清楚如何快速使许多人可以同时使用它。幸运的是，只需进行一些简单的调整，即可获得10-100倍的性能，这通常很简单。本章将向您展示如何操作。

我们将从一个比喻开始：思考像餐厅这样的shiny应用程序。接下来，您将学习如何对您的应用程序进行**基准测试**，使用shinyloadtest软件包来模拟许多人同时使用您的应用程序。这是开始的地方，因为它可以让你发现你有问题，并有助于衡量你所做的任何改变的影响。

然后，您将学习如何使用profvis软件包对应用程序进行**分析**，以识别R代码的缓慢部分。剖析可以让您确切地看到您的代码花在哪里，因此您可以将精力集中在它们影响最大的地方。

最后，您将学习一些有用的技术来**优化代码**，在需要时提高性能。您将学习如何缓存反应，如何将数据准备代码从应用程序中移出，以及一些应用心理学，以帮助您的应用程序尽可能快地**感觉**。

对于基准测试、剖析和优化的整个过程的演示，我建议观看Joe Cheng的rstudio::conf (2019) 主题演讲：[生产中的光泽：原则、最佳实践和工具](#)。在那次演讲（以及随附的[案例研究](#)）中，Joe用一个现实的应用程序完成了整个过程。

```
library(shiny)
```

特别感谢我的RStudio同事Joe Cheng、Sean Lopp和Alan Dipert，他们的RStudio::conf()谈话在撰写本章时特别有帮助。

## 23.1 Dining at restaurant Shiny

在考虑性能时，将Shiny应用程序视为餐厅是有用的。每个客户（用户）进入餐厅（服务器）并下订单（请求），然后由厨师准备（R流程）。这个比喻很有用，因为就像餐厅一样，一个R过程可以同时为多个用户服务，并且有类似的方法来应对日益增长的需求。

首先，您可以研究如何提高您当前厨师的效率（优化您的R代码）。要做到这一点，您首先要花一些时间观察您的厨师工作，以找到他们方法（剖析）的瓶颈，然后集思广益，帮助他们更快地工作（优化）。例如，也许你可以雇佣一个预科厨师，他可以在第一个客户之前来切一些蔬菜（准备数据），或者你可以投资一个节省时间的小工具（更快的R包）。

或者，您可以考虑在餐厅（服务器）中添加更多的厨师（流程）。幸运的是，添加更多流程比雇用训练有素的厨师要容易得多。如果您继续雇用更多的厨师，最终厨房（服务器）将变得太满，您将需要添加更多设备（核心记忆）。添加更多资源允许服务器运行更多进程，这称为扩展。

在某些时候，你会尽可能多地把厨师塞进你的餐厅，但这仍然不足以满足需求。到那时，你需要建造更多的餐厅。这被称为扩展，对于Shiny来说，这意味着使用多台服务器。扩展允许您处理任意数量的客户，只要您可以支付基础设施成本。在本章中，我不会更多地谈论扩展，因为虽然细节很简单，但它们完全取决于您的部署基础设施。

有一个主要的地方是这个比喻分解的：一个普通的厨师可以同时做多道菜，仔细地交织步骤，利用一个食谱中的停机时间来处理另一个食谱。然而，R是单线程的，这意味着不能同时做多件事。如果所有饭菜都煮得很快，这很好，但如果有人要求24小时的真空猪肚，所有后来的顾客都必须等待24小时，厨师才能开始用餐。幸运的是，您可以使用异步编程来绕过这一限制。异步是一个复杂的主题，超出了这本书的范围，但您可以在<https://rstudio.github.io/promises/>上了解更多信息。

## 23.2 Benchmark

您几乎总是从为自己开发应用程序开始：您的应用程序是一个私人厨师，一次只需要为一个客户（您！）服务。虽然您现在可能对他们的表现感到满意，但您可能也担心他们无法同时处理需要使用您应用程序的10个人。基准测试允许您通过多个用户检查应用程序的性能，而不会将真实用户暴露在潜在的缓慢的应用程序中。或者，如果您想为100或1000名用户提供服务，基准测试将帮助您确定每个进程可以处理多少用户，因此您需要使用多少服务器。

基准测试过程由[shinyloadtest](#)软件包支持，并有三个基本步骤：

1. 使用 `shinyloadtest::record_session()` 记录模拟典型用户的脚本。
2. 使用shinycannon命令行工具与多个同时用户重播脚本。
3. 使用 `shinyloadtest::report()` 分析结果。

在这里，我将概述每个步骤的工作原理；如果您需要更多详细信息，请查看shinyloadtest的文档和小插图。

### 23.2.1 Recording

如果您在笔记本电脑上进行基准测试，您需要使用两个不同的R进程——一个用于Shiny，一个用于shinyloadtest。

- 在第一个过程中，启动您的应用程序并复制它为您提供的网址：

```
runApp("myapp.R")
#> Listening on http://127.0.0.1:7716
```

- 在第二个过程中，将网址粘贴到 `record_session()` 调用中：

```
shinyloadtest::record_session("http://127.0.0.1:7716")
```

`record_session()` 将打开一个新窗口，其中包含您的应用程序版本，该版本记录您用它所做的一切。现在，您需要与应用程序进行交互，以模拟“典型”用户。我建议从书面脚本开始，以指导你的行动——如果你发现缺少一些重要部分，这将使将来更容易重复。您的基准测试只会与您的模拟一样好，因此您需要花一些时间思考如何模拟与应用程序的现实交互。例如，不要忘记添加暂停，以反映真实用户所需的思考时间。

完成后，关闭应用程序，shinyloadtest会将 `recording.log` 保存到您的工作目录中。这以易于重播的方式记录每个动作。抓住它，因为你下一步需要它。

（虽然基准测试在您的笔记本电脑上效果很好，但您可能希望尽可能密切地模拟最终部署，以获得最准确的结果。因此，如果您的公司有特殊的方式来为Shiny应用程序提供服务，请与您的IT人员讨论如何设置一个可用于负载测试的环境。）

### 23.2.2 Replay

现在您有一个代表单个用户操作的脚本，接下来我们将使用它来模拟许多人使用一个名为shinycannon的特殊工具。shinycannon需要安装一些额外的工作，因为它不是R包。它是用Java编写的，因为Java语言特别适合使用尽可能少的计算资源并行执行数十或数百个Web请求的问题。这使得您的笔记本电脑可以运行应用程序并模拟许多用户。因此，请按照<https://rstudio.github.io/shinyloadtest/#shinycannon>上的说明安装syshinycannon。

然后使用以下命令从终端运行shinycannon：

```
shinycannon recording.log http://127.0.0.1:7911 \  
--workers 10 \  
--loaded-duration-minutes 5 \  
--output-dir run1
```

`shinycannon` 有六个论点：

- 第一个参数是您在上一步中创建的记录的路径。
- 第二个参数是您的Shiny应用程序的网址（您在上一步中复制并粘贴了该应用程序）。
- `--workers` 设置要模拟的并行用户数量。上述命令将模拟您的应用程序的性能，就像10个人同时使用它一样。
- `--loaded-duration-minutes` 确定测试运行多长时间。如果这比您的脚本需要更长的时间，`shinycannon` 将从头开始再次启动脚本。
- `--output-dir` 给出要保存输出的目录的名称。在尝试性能改进时，您可能会多次运行负载测试，因此请努力为这些目录提供信息丰富的名称。

首次负载测试时，最好从少量工人开始，持续时间短，以便快速发现任何重大问题。

### 23.2.3 Analysis

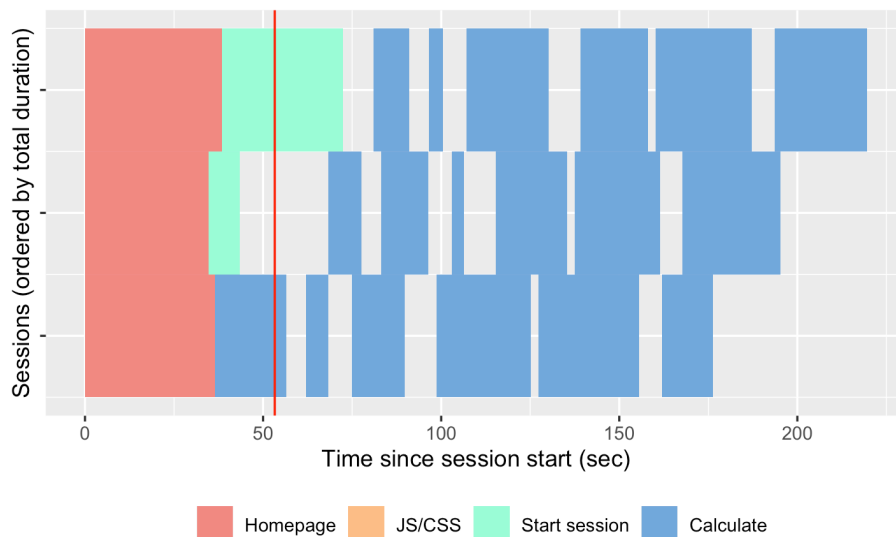
现在您已经用多个用户模拟了应用程序，是时候查看结果了。首先，使用 `load_runs()` 将数据加载到R中：

```
library(shinyloadtest)  
df <- load_runs("scaling-testing/run1")
```

这会产生一个整洁的碎屑，如果你愿意，你可以手工分析。但通常您将创建标准的shinyloadtest报告。这是一份HTML报告，包含Shiny团队发现最有用的图形摘要。

```
shinyloadtest_report(df, "report.html")
```

我不会在这里讨论报告中的所有页面。相反，我将专注于我认为最重要的情节：会话持续时间。要了解有关其他页面的更多信息，我强烈建议阅读[分析负载测试日志](#)的文章。





会话持续时间图将每个模拟用户会话显示为行。每个事件都是一个矩形，宽度与所花费的时间成正比，按事件类型着色。红线显示原始记录所花费的时间。

当看这个情景时：

- 该应用程序在负载下的性能是否与单个用户的性能相同？如果是这样，恭喜你！你的应用程序已经足够快了，你可以停止阅读这一章 😊。
- “主页”中的速度慢吗？如果是这样，您可能正在使用 `ui` 函数，并且您不小心在那里做了太多工作。
- “开始会话”慢吗？这表明您的 `server` 函数的执行速度很慢。一般来说，运行 `server` 函数应该很快，因为您所做的只是定义反应图（在下一步中运行）。如果速度慢，请将昂贵的代码移到 `server()` 之外（因此它在应用程序启动时运行一次）或进入反应（因此按需运行）。
- 否则，最典型的是，缓慢将在“计算”中，这表明您的反应中的一些计算是缓慢的，您需要使用本章其余部分中的技术来查找和修复瓶颈。

## 23.3 Profiling

如果您的应用程序花费大量时间计算，您接下来需要弄清楚哪个计算速度慢，即您需要分析代码以找到瓶颈。我们将使用 `profvis` 软件包进行剖析，该软件包为 `utils::Rprof()` 收集的剖析数据提供交互式可视化。我将首先介绍火焰图，用于分析的可视化，然后向您展示如何使用 `profvis` 来分析 R 代码和 Shiny 应用程序。

### 23.3.1 The flame graph

在编程语言中，用于可视化分析数据的最常见工具是火焰图。为了帮助您理解它，我将首先重新审视代码执行的基础知识，然后逐步构建到最终的可视化。

为了使过程更具体，我们将使用以下代码，我使用 `profvis::pause()`（稍后会详细介绍）来指示正在进行的工作：

```
library(profvis)

f <- function() {
  pause(0.2)
  g()
  h()
  10
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.3)
}
```

如果我让你在精神上运行 `f()`，然后解释什么函数被调用，你可能会这样说：

- 我们从 `f()` 开始。
- 然后 `f()` 调用 `g()`
- 然后 `g()` 调用 `h()`

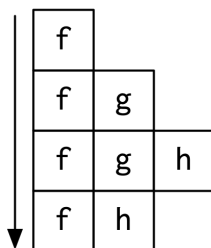
- 然后 `f()` 调用 `h()`

这有点难以理解，因为我们无法确切地看到调用是如何嵌套的，因此您可能会采用更具概念性的描述：

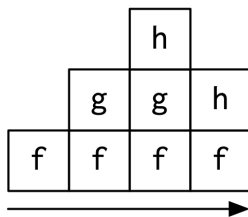
- `f`
- `f > g`
- `f > g > h`
- `f > h`

在这里，我们录制了一个调用堆栈列表，当我们谈论调试时，您可能还记得第[5.2.1节](#)。调用堆栈会描述函数的完整调用序列。

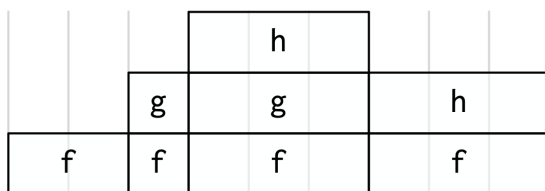
我们可以通过在每个函数名称周围绘制一个矩形来将该列表转换为图表：



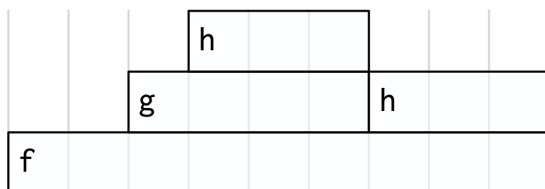
我认为最自然的是考虑时间从上到下的向下流动，就像你通常考虑代码运行一样。但按照惯例，火焰图是随着时间从左到右流动绘制的，因此我们将图表旋转90度：



我们可以通过使每个调用的宽度与所需的时间成正比来使该图表更具信息性。我还在后台添加了一些网格线，以便更容易检查我的工作：



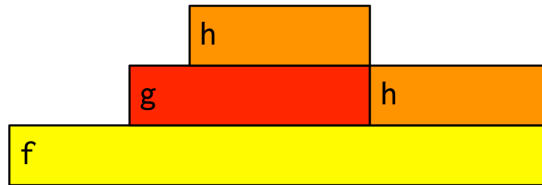
最后，我们可以通过将相邻调用合并到相同的函数来清理一下：



这是一个火焰图！很容易看出 `f()` 需要多长时间才能运行，以及为什么需要那么长时间，即它的时间花在哪里。

你可能想知道为什么它被称为火焰图。野外的大多数火焰图都是随机着色的“暖”颜色，这意味着唤起计算机运行“热”的想法。然而，由于这些颜色不会添加任何额外的信息，我们通常会省略它们，并坚持使用黑白。您可以在[“The Flame Graph”](#)中了解更多关于这种配色方案、替代方案和火焰图的历史。





## 23.3.2 Profiling R code

现在您了解了火焰图，让我们用profvis软件包将其应用于真实代码。易于使用：只需包装您想要配置的代码到 `profvis::profvis()`。

```
profvis::profvis(f())
```

代码完成后，profvis将弹出一个交互式可视化，图23.1。你会注意到，它看起来与我手绘的图表非常相似，但时间并不完全相同。这是因为R的分析器通过每10毫秒停止执行并记录调用堆栈。不幸的是，我们不能总是停留在我们想要的地方，因为R可能处于无法被打断的事情中。这意味着结果会受到少量随机变化的影响；如果您重新分析此代码，您将获得另一个略微不同的结果。

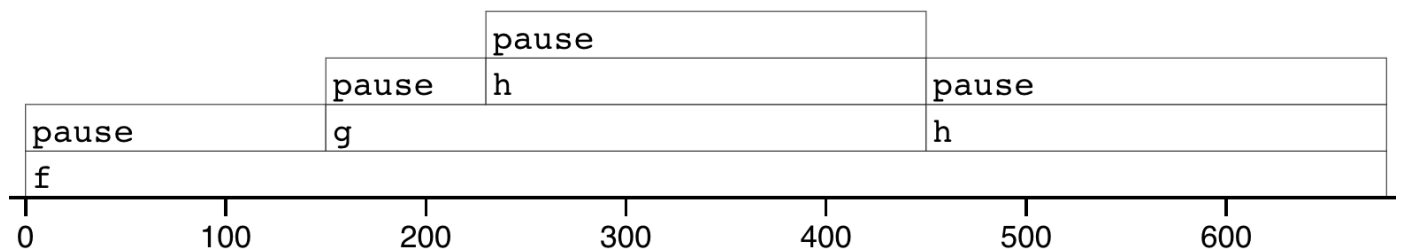


图23.1：使用profvis分析 `f()` 的结果。X轴以毫秒显示经过的时间，y轴显示调用堆栈的深度。

除了火焰图外，profvis还尽最大努力查找和显示底层源代码，以便您可以单击火焰图中的函数来准确查看运行的内容。

## 23.3.3 Profiling a Shiny app

剖析Shiny应用程序时没有太大变化。为了了解差异，我将制作一个非常简单的应用程序，包裹着 `f()` 结果，如图23.2所示。

```
ui <- fluidPage(
  actionButton("x", "Push me"),
  textOutput("y")
)
server <- function(input, output, session) {
  output$y <- eventReactive(input$x, f())
}

# Note the explicit call to runApp() here: this is important
# as otherwise the app won't actually run.
profvis::profvis(runApp(shinyApp(ui, server)))
```



## 23.4 Improve performance

提高性能的最有效方法是找到配置文件中最慢的东西，并尝试加快速度。一旦您隔离了一个缓慢的部分，请确保它是一个独立的功能片段（第18章）。然后制作一个最小的代码片段，重新创建并测试速度，重新分析它，以检查您是否正确捕获了它。当您尝试可能的改进时，您将多次重新运行此片段。我还建议写一些测试（第21章），因为根据我的经验，使代码更快的最简单方法是使其不正确😅。

Shiny代码只是R代码，所以大多数提高性能的技术都是通用的。两个好的开始是Colin Gillespie和Robin Lovelace的Advanced R和[Efficient R programming](#)的[Improving performance](#)部分。我不会在这里重复他们的建议：相反，我将专注于最有可能影响你的Shiny应用程序的技术。我还强烈推荐Alan Dipert的rstudio::conf(2018)演讲：[Making Shiny fast by doing as little as possible](#)。

首先解决现有代码运行频率高于预期的任何问题——确保您没有在多个反应中重复相同的工作，并且反应图没有比您预期更频繁地更新（第14.6节）。

接下来，我将讨论提高应用程序性能的最简单方法，使用缓存来记住和重复缓慢的计算。我将用另外两种技术来结束，这些技术可以帮助许多shiny应用程序：将昂贵的预处理拉到单独的步骤中，并仔细管理用户期望。

## 23.5 Caching

缓存是提高代码性能的非常强大的技术。基本想法是记录对函数的每次调用的输入和输出。当使用已经看到的一组输入调用缓存函数时，它可以重新释放记录的输出，而无需重新计算。像[memoise](#)这样的软件包提供了缓存常规R函数的工具。

缓存对Shiny应用程序特别有效，因为缓存可以在用户之间共享。这意味着当许多人使用同一应用程序时，只有第一个用户需要等待计算结果，然后其他人都会从缓存中获得快速结果。

Shiny提供了一个用于缓存任何反应式表达式或渲染函数的通用工具：`bindCache()`。如您所知，反应式表达式已经缓存了最近计算的值；`bindCache()`允许您缓存任意数量的值，并在用户之间共享这些值。在本节中，我将向您介绍`bindCache()`的基础知识，向您展示几个实际示例，然后讨论缓存“密钥”和范围的一些细节。如果您想了解更多，我建议从“[在Shiny中使用缓存来最大限度地提高性能](#)”和“[使用bindCache\(\)来加快应用程序的速度](#)”开始。

### 23.5.1 Basics

`bindCache()`易于使用。只需将您要缓存的`reactive()`或`render*`函数通过管道链接到`bindCache()`。

```
r <- reactive(slow_function(input$x, input$y)) %>%
  bindCache(input$x, input$y)

output$text <- renderText(slow_function2(input$z)) %>%
  bindCache(input$z)
```

额外的参数是缓存密钥—这些是用于确定之前是否见过计算的值。在展示几个实际用途后，我们将更详细地讨论缓存密钥。

## 23.5.2 Caching a reactive

使用缓存的常见地方是与Web API结合使用—即使API非常快，您仍然必须发送请求，等待服务器响应，然后解析结果。因此，缓存API结果通常会带来很大的性能提升。让我们使用与GitHub的API对话的`gh`包，以一个简单的例子来说明这一点。

想象一下，你想设计一个显示人们最近一直在工作的应用程序。在这里，我写了一个小函数，从Github的事件API中获取数据，并做一些简单的[矩形](#)，将其变成一个小石头：

```
library(purrr)

latest_events <- function(username) {
  json <- gh::gh("/users/{username}/events/public", username = username)
  tibble::tibble(
    repo = json %>% map_chr(c("repo", "name")),
    type = json %>% map_chr("type"),
  )
}

system.time(hadley <- latest_events("hadley"))
#>    user  system elapsed
#> 0.074   0.015   0.796
head(hadley)
#> # A tibble: 6 × 2
#>   repo                type
#>   <chr>              <chr>
#> 1 tidyverse/purrr IssueCommentEvent
#> 2 tidyverse/purrr IssueCommentEvent
#> 3 tidyverse/purrr IssuesEvent
#> 4 tidyverse/purrr IssueCommentEvent
#> 5 tidyverse/purrr IssueCommentEvent
#> 6 tidyverse/purrr IssueCommentEvent
```

我可以把它变成一个非常简单的应用程序：

```
ui <- fluidPage(
  textInput("username", "GitHub user name"),
  tableOutput("events")
)
server <- function(input, output, session) {
  events <- reactive({
    req(input$username)
    latest_events(input$username)
  })
  output$events <- renderTable(events())
}
```

这个应用程序会感觉有点迟钝，因为每次您输入用户名时，它都必须重新请求数据，即使您15秒前才要求它。我们可以通过使用 `bindCache()` 大幅提高性能：

```
server <- function(input, output, session) {
  events <- reactive({
    req(input$username)
    latest_events(input$username)
  }) %>% bindCache(input$username)
  output$events <- renderTable(events())
}
```

您可能已经发现了这种方法的问题—如果您明天再来为同一用户请求数据，会发生什么？即使可能有新的活动，您也会获得今天的数据。您需要明确表示对时间的隐性依赖。您可以通过将 `Sys.Date()` 添加到缓存密钥中来做到这一点，这样缓存实际上只持续一天：

```
server <- function(input, output, session) {
  events <- reactive({
    req(input$username)
    latest_events(input$username)
  }) %>% bindCache(input$username, Sys.Date())
  output$events <- renderTable(events())
}
```

您可能担心缓存会稳步积累过去几天的数据，您再也不会看这些数据，但幸运的是，缓存具有固定的总大小，并且足够聪明，可以在需要更多空间时自动删除最近使用最少的数据。

### 23.5.3 Caching plots

大多数时候，您会缓存反应，但您也可以将 `bindCache()` 与渲染函数一起使用。大多数渲染函数都非常快，但如果你有复杂的图形，有一个函数可能会很慢：`renderPlot()`

例如，以以下应用程序为例。如果您自己运行它，您会注意到第一次显示每个图时，渲染需要明显的几分之一秒，因为它必须绘制约50,000点。但下次您绘制每个情节时，它会立即出现，因为它是从缓存中检索的。

```
library(ggplot2)

ui <- fluidPage(
  selectInput("x", "X", choices = names(diamonds), selected = "carat"),
  selectInput("y", "Y", choices = names(diamonds), selected = "price"),
  plotOutput("diamonds")
)

server <- function(input, output, session) {
  output$diamonds <- renderPlot({
    ggplot(diamonds, aes(.data[[input$x]], .data[[input$y]])) +
      geom_point()
  }) %>% bindCache(input$x, input$y)
}
```

(如果您不熟悉 `.data` 语法，请参阅第12章了解详情。)

在缓存打印时，有一个特殊的考虑因素：每个打印都以各种大小绘制，因为默认打印占用了100%的可用宽度，而可用宽度随着浏览器大小的调整而变化。这种灵活性对于缓存来说效果不佳，因为即使是大小上的一个像素差异也意味着无法从缓存中检索到绘图。为了避免这个问题，`bindCache()` 缓存具有固定大小的绘图。默认值经过精心选择，在大多数情况下“仅起作用”，但如果需要，您可以使用 `sizePolicy` 参数进行控制，并通过 `sizeGrowthRatio`，了解更多信息。

## 23.5.4 Cache key

值得简要地谈论一下缓存密钥：用于确定之前是否执行过计算的值。这些值也用于确定反应依赖项，就像 `observeEvent()` 或 `eventReactive()` 的第一个参数一样。这意味着，如果您使用错误的缓存密钥，您可能会获得非常令人困惑的结果。例如，我有这个缓存反应的图像：

```
r <- reactive(input$x + input$y) %>% bindCache(input$x)
```

如果 `input$y` 更改，`r()` 将不会重新计算。如果从缓存中检索结果，它将是 `x` 的当前值和缓存该值时 `y` 碰巧具有的任何值的总和。

因此，缓存密钥应始终包含表达式中的所有反应式输入。但您可能还想包含反应中未使用的附加值。最有用的例子是添加当前日期或一些四舍五入的当前时间，以便缓存值仅用于固定时间。

除了输入，您还可以使用其他 `reactive()` 作为缓存密钥，但您需要尽可能保持它们简单（即原子向量或简单的原子向量列表）。不要使用大型数据集，因为确定是否已经看到大型数据框是昂贵的，这将减少您从缓存中看到结果的好处。

## 23.5.5 Cache scope

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

- `bindCache(..., cache = "session")` 将为每个用户会话使用单独的缓存。这确保了私人数据不会在用户之间共享，但也减少了缓存的好处。
- 使用 `shinyOptions(cache=cachem:: cache_mem())` 或 `shinyOptions`。您可以使用使缓存在多个进程之间共享，并在应用程序重新启动时持续。通过 `?bindCache` 获取更多详细信息。

也可以将多个缓存链接在一起或编写自己的自定义存储后端。您可以在cachem的文档中了解有关这些选项的更多信息，[cachem](#)是支持 `bindCache()` 的缓存包。

## 23.6 Other optimizations

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

### 23.6.1 Schedule data munging

想象一下，您的Shiny应用程序使用一个数据集，该数据集需要一些初始数据清理。数据准备相对复杂，需要花费较长的时间。您发现这是您应用程序的瓶颈，并希望做得更好。

让我们假装你已经将代码提取到一个函数中，它看起来像这样：

```
my_data_prep <- function() {
  df <- read.csv("path/to/file.csv")
  df %>%
    filter(!not_important) %>%
    group_by(my_variable) %>%
    some_slow_function()
}
```

目前，您在server函数中调用它：

```
server <- function(input, output, session) {
  df <- my_data_prep()
  # Lots more code
}
```

每次新会话启动时都会调用server函数，但数据总是相同的，因此您可以通过将数据处理移出 `server()`，来使您的应用程序更快（使用更少的内存）：

```
df <- my_data_prep()
server <- function(input, output, session) {
  # Lots more code
}
```

由于您正在关注此代码，因此也值得检查您是否使用最有效的方法来加载数据：

- 如果您有一个平面文件，请尝试 `data.table::fread()` 或 `vroom::vroom()` 而不是 `read.csv()` 或 `read.table()`
- 如果您有数据框，请尝试使用 `arrow::write_feather()` 保存，并使用 `arrow::read_feather()` 阅读。Feather是一种二进制文件格式，读写速度可以快得多。
- 如果您有不是数据框的对象，请尝试使用 `qs::qread()` / `qs::qsave()` 而不是 `readRDS()` / `saveRDS()`

如果这些更改不足以解决瓶颈，您可以考虑使用单独的cron作业或计划的RMarkdown报告来调用 `my_data_prep()` 并保存结果。然后，您的应用程序可以加载预先准备好的数据并开始工作。这就像雇佣一个预料厨师，他在凌晨3点（没有顾客的时候）来，这样在午餐高峰期，你的厨师可以尽可能高效。

## 23.6.2 Manage user expectations

最后，您可以对应用程序设计进行一些调整，使其感觉更快，并改善应用程序的整体用户体验。以下是可用于许多应用程序的四个技巧：

- 使用 `tabsetPanel()` 将您的应用程序拆分为选项卡。仅重新计算当前选项卡上的输出，因此您可以使用它来专注于用户当前正在查看的内容。
- 需要按下按钮才能开始长期运行的操作。操作开始后，使用第8.2节的技术让用户知道发生了什么。如果可能，显示增量进度条（第8.3节），因为有充分证据表明进度条使操作感觉更快：  
<https://www.nngroup.com/articles/progress-indicators/>。
- 如果应用程序在启动时需要大量工作（并且您无法通过预处理来减少它），请确保设计您的应用程序，以便UI仍然可以显示，以便您可以让用户知道他们需要等待。



- 最后，如果您想在后台进行一些昂贵的操作时保持应用程序响应，是时候了解异步编程了：  
<https://rstudio.github.io/promises/index.html>。

## 23.7 Summary

本章为您提供了精确测量和提高任何shiny应用程序性能的工具。您学习了shiny负载测试来衡量性能，使用shinycannon模拟多个用户同时使用您的应用程序。然后，您学会了如何使用profvis来找到最昂贵的操作，以及可用于改进它的技术。

这是《Mastering Shiny》的最后一章—感谢您一直坚持到最后！我希望你发现这本书有用，我给你的技能能帮助你制作许多引人注目的shiny应用程序。我很想听听你是否觉得这本书有用，或者你认为未来有什么可以改进的。联系的最佳方式是推特，[@hadleywickham](#)，或Github，<https://github.com/hadley/mastering-shiny/>。再次感谢您的阅读，并祝愿您未来的shiny应用程序越来越好用！