Dashboards em Shiny I

Reatividade (parte 1)

Setembro de 2023

Atividade

Vamos fazer um Shiny app que mostre o resultado do sorteio de n números entre 1 e 10, sendo n um valor escolhido na UI.

Ao RStudio: 05-amostra-sem-reactive.R

Reatividade: outro paradigma

Quando escrevemos código R, dois paradigmas estão sempre presentes:

- podemos avaliar uma linha de código assim que a escrevermos; e
- se decidirmos rodar todo o script de uma vez, as linhas de código serão avaliadas sequencialmente.

Isso faz com que as nossas tarefas de análise de dados geralmente virem scripts sequenciais, cujo código não pode ser executado fora de ordem.

No Shiny, utilizamos um outro paradigma de programação.

Exemplo de script sequencial

O código abaixo executa a corriqueira tarefa de importar, manipular e visualizar uma base. Se o código for rodado fora de ordem, nada vai funcionar. Esse paradigma é conhecido como **programação imperativa**.

```
tab_starwars <- dplyr::starwars

tab_grafico <- tab_starwars |>
  tidyr::unnest(films) |>
  tidyr::drop_na(species) |>
  dplyr::group_by(films) |>
  dplyr::summarise(total_especies = dplyr::n_distinct(species))
  dplyr::mutate(
    films = forcats::fct_reorder(films, total_especies)
  )

tab_grafico |>
  ggplot2::ggplot(ggplot2::aes(y = films, x = total_especies))
  ggplot2::geom_col() +
  ggplot2::theme_minimal() +
  ggplot2::labs(x = "Total de espécies", y = "Filme")
```

Programação declarativa

Na **programação declarativa**, os comandos não são executados imediatamente. Nosso código funciona como uma receita, que só será utilizada quando necessário.

No contexto do Shiny, a nossa receita será um **um diagrama de reatividade**, isto é, um conjunto de dependências que decide quais outputs devem ser recalculados quando um input muda.

O diagrama de reatividade de um shiny app possui 3 estruturas principais:

- **valores reativos**
- **funções observadoras**
- **expressões reativas**

Valores reativos

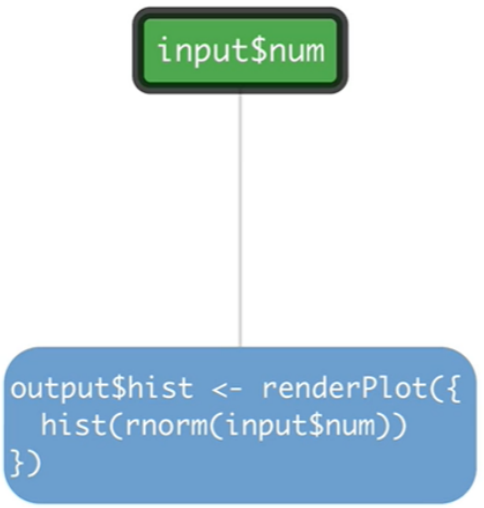
Os valores reativos são **a origem do diagrama de reatividade**. Eles guardam as informações que vêm da UI (a partir dos inputs) e disparam *sinais de alerta* sempre que essas informações mudam.

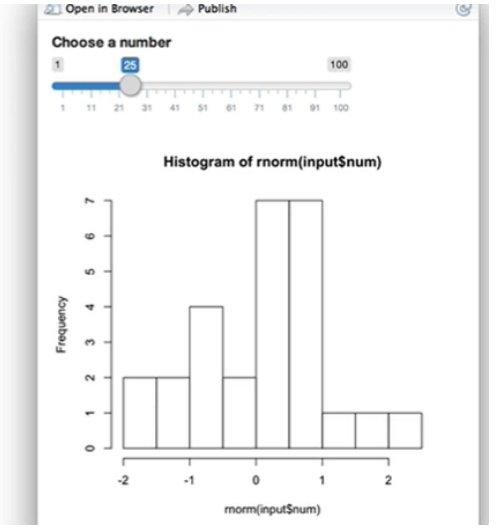
Os valores reativos mais comuns são aqueles dentro da lista `input`.

Funções observadoras

Os *sinais de alerta* disparados por um valor reativo são um aviso dizendo que todos os outputs que dependem dele precisam ser recalculados. O destino desses sinais são as **funções observadoras**. Elas guardam o código de cada output e são o **ponto final do fluxo de reatividade**.

As **funções observadoras** mais comuns são as funções `render*()`.





Expressões reativas

Muitas vezes, um aplicativo shiny precisa de passos intermediários, entre o input de origem e o output final. Isto é, precisamos de uma estrutura que receba um valor reativo, faça alguma conta e devolva um resultado, um valor também reativo que será utilizado posteriormente em uma função observadora.

Essas estruturas são as **expressões reativas**. Para criá-las, podemos utilizar as funções `reactive()` e `eventReactive()`.

A função reactive()

A função `reactive()` pode ser utilizada para criar **expressões reativas**, cujos valores funcionam como um valor reativo. Essa função é muito utilizada para a construção de expressões intermediárias, que podem ser usadas na construção de outputs diferentes.

```
# server
amostra <- reactive({
  sample(1:10, input$tamanho, replace = TRUE)
})
```

Para acessar o valor dessa expressão reativa, devemos usar parênteses após o nome, como se fosse uma função sem argumentos.

```
# server
output$summary <- renderPrint({
  amostra() |>
    table() |>
    barplot()
})
```

Atividade

Vamos reconstruir nosso app anterior utilizando a função `reactive()` para resolver o problema da amostra.

Ao RStudio: 06-amostra-com-reactive.R

Contexto reativo

Valores reativos e **expressões reativas** só podem ser utilizadas dentro de um **contexto reativo**.

No exemplo abaixo, a função `renderPlot()` cria um contexto reativo e, por isso, podemos utilizar o valor reativo `input$num` dentro dela.

Certo

```
# server
output$hist <- renderPlot({hist(rnorm(input$num))})
```

Errado

```
# server
output$hist <- hist(rnorm(input$num))
#> Error : Can't access reactive value 'tamanho' outside of
# reactive consumer.
```

Disparo de reatividade

O envio de sinais por parte dos valores reativos, avisando que seus valores estão desatualizados para que as expressões reativas e funções observadoras recalculem seus resultados, só acontece se o **diagrama de reatividade estiver completo**.

Podemos chamar esse envio de **disparo de reatividade**. Ele só acontece para os caminhos do diagrama de reatividade que começam com um valor reativo e terminam com uma função observadora. **Caminhos que terminam com expressões reativas não causam o disparo de reatividade**.

Atividade

Vamos olhar mais de perto o contexto reativo e o disparo de reatividade.

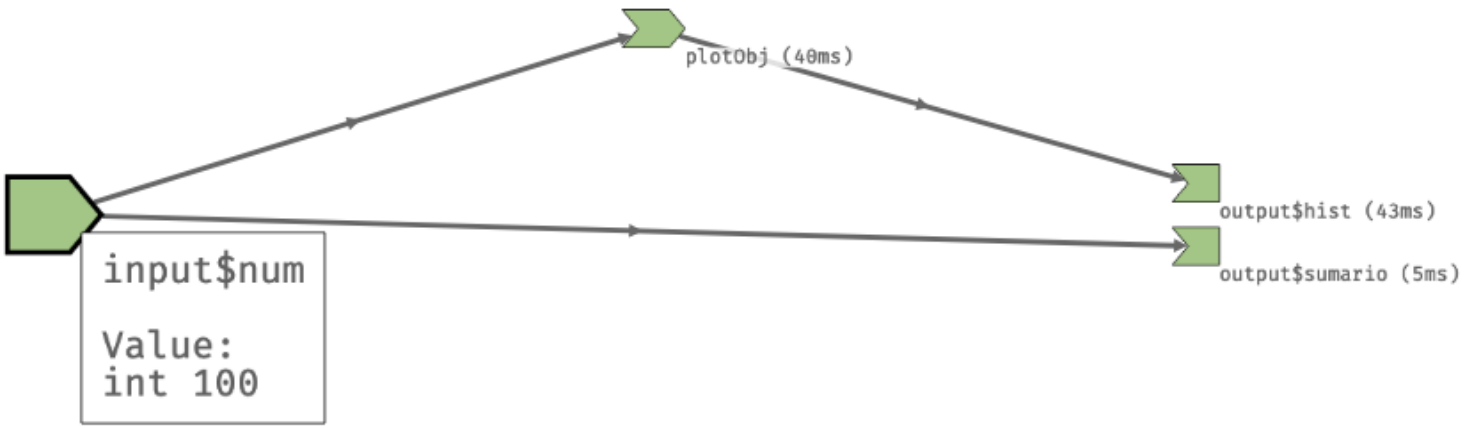
Ao RStudio: 07-disparo-reatividade.R

Gráfico de reatividade

Você pode gerar um gráfico de reatividade do seu aplicativo utilizando o pacote `reactlog`. Com o pacote instalado, basta rodar o seguinte comando no Console

```
options(shiny.reactlog = TRUE)
```

e então, dentro do app, utilizar o comando `CTRL + F3` (no Mac, `command + F3`).



Atividade

Vamos olhar o gráfico de reatividade do nosso app `06-amostra-com-reactive.R`.

Ao RStudio: 06-amostra-com-reactive.R

A função eventReactive()

Enquanto a função `reactive()` observa mudanças em todos os valores reativos presentes no código dela, a função `eventReactive()` observa mudanças em apenas um valor reativo, especificado na chamada da própria função.

```
# ui
actionButton("atualizar", "Gerar gráfico")
```

```
# server
titulo <- eventReactive(input$atualizar, {
  input$titulo
})

output$hist <- renderPlot({
  hist(amostra(), main = titulo())
})
```

Atividade

Vamos criar um botão para atualizar o título do gráfico apenas quando clicarmos nele.

Ao RStudio: 08-eventReactive.R

Corrigindo erros (debug)

A seguir, listamos erros muito comuns que cometemos quando estamos programando em Shiny.

- Erros comuns de programação em R, como chamar objetos ou funções inexistentes, operações não permitidas ou má utilização de funções costumam devolver mensagens de erro informativas. O primeiro passo para resolver qualquer problema de programação de Shiny (ou de R em geral) é ler a mensagem de erro e tentar interpretá-la.

- Erros de sintaxe no Shiny. Em geral, o app não roda e receberemos a mensagem de erro `unexpected symbol`. Causados principalmente por falta ou excesso de vírgulas, parênteses ou chaves.

- Usar uma expressão reativa fora de uma função reativa. O app não vai rodar e você verá a seguinte mensagem de erro `Can't access reactive value 'tamanho' outside of reactive consumer.`.

Corrigindo erros (debug)

- Você só pode *ler* valores da lista `input`. Se você tentar gravar um valor diretamente, será retornado um erro. A lista `input` será sempre uma *cópia* das ações do usuário no navegador.

```
ui <- fluidPage(shiny::numericInput("valor", "Um número", valu

server <- function(input, output, session) {
  input$valor <- 10
}
# Error in `$<-.reactivevalues`(`*tmp*`, valor, value = 10) :
#   Attempted to assign value to a read-only reactivevalues ob
```

Corrigindo erros (debug)

- Você só pode *escrever* valores na lista `output`. Se você tentar ler um valor, será retornado um erro.

```
ui <- fluidPage(shiny::textOutput("valor"))

server <- function(input, output, session) {
  print("O valor do output é ", output$valor)
}
# Error in `$.shinyoutput`(output, valor.) :
#   Reading from shinyoutput object is not allowed.
```

Corrigindo erros (debug)

- Não fazer a correspondência certa entre as funções `_Output()` e `render_()`. O app vai rodar, mas a visualização não ser mostrada. Em algumas situações, uma mensagem de erro será retornada. Em outras, o erro será silencioso.

- Errar o nome de um input (usar um input que não existe). O app vai rodar e, geralmente, retornar um erro relacionado a uma função receber um valor que não deveria ser NULL.

- Errar o nome de um output. O app vai rodar e não vai retornar erro. O output não será gerado.

Corrigindo erros (debug)

- Usar o mesmo `Id` para dois outputs. O app vai rodar e não vai retornar erro. Os outputs não serão gerados.

- Esquecer os parênteses ao chamar uma expressão reativa (objeto criado pelas funções `reactive()` e `eventReactive()`). Normalmente receberemos uma mensagem indicando que a classe de algum objeto está errada, como `'x' must be numeric` ou a mensagem `cannot coerce type 'closure' to vector of type ...`.

A função browser()

Quando as mensagens de erro não nos ajudarem, podemos usar a função `browser()` para espiarmos o que está acontecendo dentro do server.

Basta colocar essa função onde você suspeita que o erro está acontecendo e, ao rodar o app, você poderá utilizar o Console para avaliar os objetos que estiverem no Environment.

Enquanto a função `browser()` estiver ativa, o app ficará congelado.

A função browser()

```
# server
valor_reativo <- reactive({
  sample(1:10, 1)
})

output$plot <- renderPlot({
  browser()
  hist(rnorm(100, valor_reativo, 1))
})
```

```
# No console

# Browse[1]> valor_reativo()
# [1] 4
```

A função browser()

Embora você possa colocar a função browser em qualquer lugar no server, é melhor colocá-la dentro de uma função reativa. Caso contrário, o `browser()` só será chamado na inicialização do app e você não conseguirá avaliar valores reativos.

Veja o exemplo no slide a seguir.

A função browser()

```
# server
amostra <- reactive({
  sample(1:10, input$n)
})

browser()

output$plot <- renderPlot({
  amostra() |>
    table() |>
    barplot()
})
```

```
# No console

# Browse[1]> amostra()
#Error : Can't access reactive value 'amostra' outside of reac
```

Atividade

Vamos tentar corrigir os erros de um script e fazer o app rodar.

Ao RStudio: 09-debug.R

Carregando bases de dados

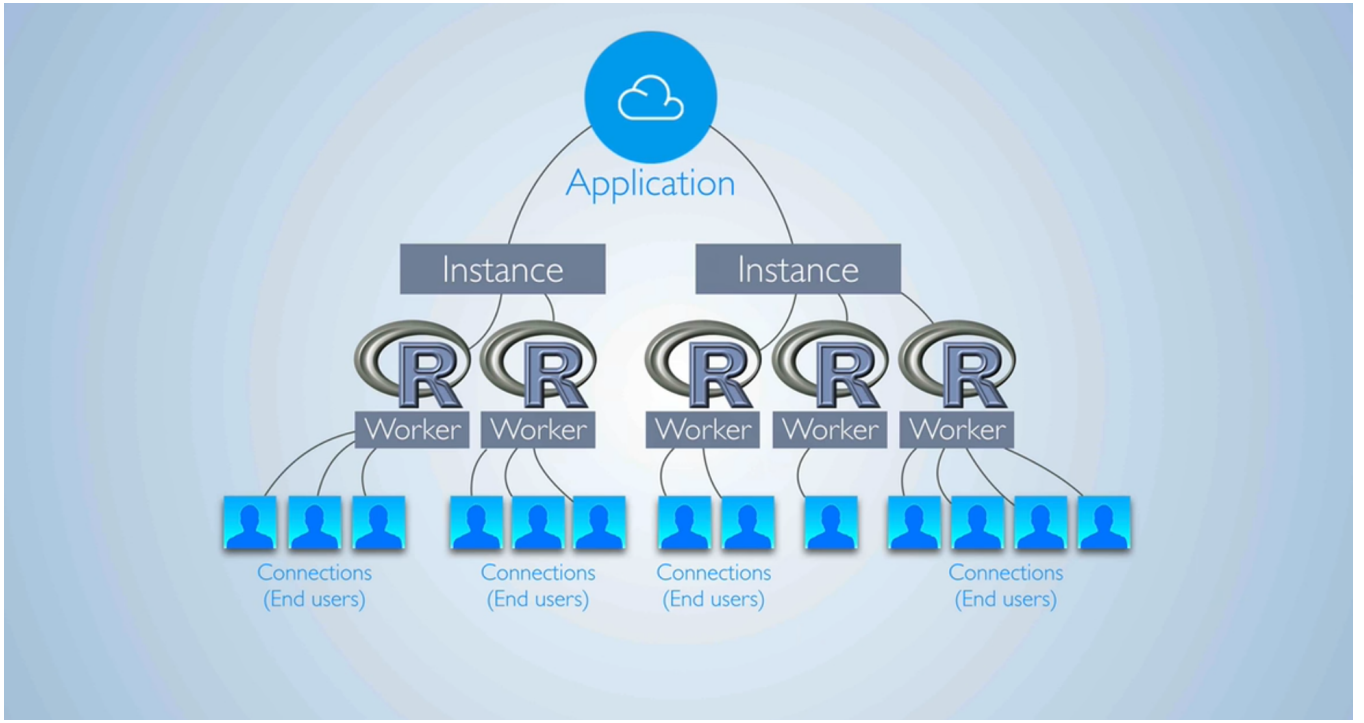
Você pode importar uma base de dados normalmente dentro da função server. Em geral, importar bases não envolve um contexto reativo, portanto essa operação pode ser feita fora de expressões reativas ou funções observadoras.

```
# server
imdb <- read_rds("dados/imdb.rds")

output$table <- renderTable({
  imdb |>
    top_n(20, receita) |>
    select(titulo, ano, diretor)
})
```

Arquitetura de um servidor shiny

A única mudança na tarefa de importar bases de dados que temos dentro do Shiny é decidir onde isso será feito (dentro ou fora do `server`). Para entender melhor, precisamos olhar a estrutura de um servidor shiny.



Fonte: rstudio.com/shiny/

1. O primeiro nível (*Application*) representa o servidor onde o app está hospedado.

2. O segundo nível (*Instance*) representa uma máquina virtual (instância), gerada pelo servidor principal, que tem como única missão rodar o seu app. Cada servidor pode abrir várias instâncias.

3. O terceiro nível (*Worker*) é uma sessão de R, responsável por servir o app. Cada instância pode abrir várias sessões de R.

4. O último nível (*Connections*) representa o usuário final. Cada sessão de R pode abrir várias sessões independentes do app, servindo vários usuários ao mesmo tempo.

Novas instâncias e sessões são abertas para administrar eficientemente fluxos maiores de usuários.

Para saber mais, veja a página no Shiny Server.

Onde carregar a nossa base de dados?

Essa arquitetura é importante para entendermos onde devemos importar os nossos dados. Mas, se quisermos abstrair todas essa informações, basta lembrarmos das seguintes regras:

Carregando fora do server, a base será importada apenas uma vez, assim que cada sessão de R for criada.

```
library(shiny)
dados <- funcao_que_le_os_dados()

ui <- fluidPage(
  # o objeto dados pode ser utilizado aqui
)

server <- function(input, output, session) {
  # o objeto dados pode ser utilizado aqui
}

shinyApp(ui, server)
```

Onde carregar a nossa base de dados?

Carregando dentro do server, mas fora de um contexto reativo, a base será importada sempre que alguém abrir o app. No entanto, não importa qual ação a pessoa fizer, a base não será importada novamente enquanto ela estiver usando app.

```
library(shiny)

ui <- fluidPage(
  # o objeto dados NÃO pode ser utilizado aqui
)

server <- function(input, output, session) {
  dados <- funcao_que_le_os_dados()
}

shinyApp(ui, server)
```

Onde carregar a nossa base de dados?

Carregando dentro do server, mas dentro de um contexto reativo, a base será importada sempre que alguém fizer a ação gatilho desse contexto. Se o objeto `dados` for uma expressão reativa, ele só poderá ser utilizado dentro de um contexto reativo.

Veja o exemplo no próximo slide.

Onde carregar a nossa base de dados?

```
library(shiny)

ui <- fluidPage(
  actionButton(
    inputId = "ler_base",
    label = "Atualizar a base"
  )
  # o objeto dados NÃO pode ser utilizado aqui
)

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

  dados <- eventReactive(input$ler_base{
    importar_versao_mais_recente_dos_dados()
  })

  # o objeto dados pode ser utilizado aqui, mas apenas dentro
  # de um contexto reativo

}

shinyApp(ui, server)
```

Atividade

Vamos criar um novo Shiny app que importa uma base de dados.

Ao RStudio: 10-dados.R

Exercícios

1) Faça os exercícios do Capítulo 3 do livro Programando em Shiny.

2) Faça os exercícios do Capítulo 4 do livro Programando em Shiny.

Referências e material extra

Tutoriais

- Tutorial de Shiny do Garrett Grolemund
- O pacote reaclog

Material avançado

- Mais sobre reatividade
- Mais sobre debug
- Shiny Server