Futures and furrr

Statistical Computing & Programming

Shawn Santo

Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Additional resources

- A Future for R: A Comprehensive Overview
- Future Topologies
- furrr
- Futureverse

Futures

What is a future?

- A concept first introduced in 1977 in a paper by Henry Baker and Carl Hewitt. It is an abstraction for a value that may be available at some point in the future.
- Futures have two states: **resolved** or **unresolved**.
- Once a future is resolved, it is immediately available for use.

The `future` package

- The `future` package provides a simple way to evaluate R expressions asynchronously or in the current R session. It is similar to what we did with `mcparallel()` and `mccollect()` in the `parallel` package.
- Package provides a lot of flexibility
- Serves as the foundation for a few other packages we will work with: `furrr` and `disk.frame`

Creating futures

Futures can be created implicitly or explicitly. Both styles work well, and you may find the implicit style more natural.

```
library(future)
```

Implicit future creation

```
a %<-% {
  cat("This value came from an implicit future", "\n")
  20
}
```

```
a
```

```
#> This value came from an implicit future
```

```
#> [1] 20
```

The `%<-%` operator creates a future and a promise to its value.

Creating futures (continued)

Futures can be created implicitly or explicitly. Both styles work well, and you may find the implicit style more natural.

```
library(future)
```

Explicit future creation

```
a <- future({
  cat("This value came from an explicit future", "\n")
  20
})
```

```
value(a)
```

```
#> This value came from an explicit future
```

```
#> [1] 20
```

Function `future()` creates a future and function `value()` gets the value.

Choosing a `plan()`

Futures are useful because you can evaluate the "future" expression in a separate R process simply by changing your `plan()` of execution. The `plan()` defines how futures are resolved.

Strategy	Style	OS compatibility	Machines
sequential	sequentially	all	single
multisession	in parallel	all	single
multicore	in parallel	not on Windows	single
cluster	in parallel	all	many

The default is `sequential`. As such, the main R process is blocked until the future is resolved.

Strategy consistency

Regardless of which strategy (see previous slide) you choose, the following hold:

1. All future expression evaluation is done in a local environment.

2. When a future is constructed, global variables are identified. For asynchronous evaluation, globals are exported to the R process/session that will be evaluating the future expression.

3. Future expressions are only evaluated once.

This consistency makes it easy to go from sequential to parallel processing and be compatible in a variety of computing environments.

Exercises

Try the following examples. What do you notice?

```
plan(sequential)
ls() # show objects in current environment
Sys.getpid()

ex_1 %<-% {
  cat("The system PID is", Sys.getpid(), "\n")
  b <- sapply(mtcars, is.na)
  sum(b)
}
ls()
```

```
plan(multisession(workers = 2))
ls()
Sys.getpid()
X <- matrix(rnorm(1000 * 1000), nrow = 1000, ncol = 1000)

ex_2 <- future({
  cat("The system PID is", Sys.getpid(), "\n")
  solve(X)
})
X_inverse <- value(ex_2)
```

Exercises (continued)

```
plan(multisession(workers = 2))
library(ggplot2)
library(plotly)

ex_3_a %<-% {
  Sys.sleep(5)
  plot_ly(data = diamonds,
          x = ~price, y = ~carat, z = ~table,
          type = "scatter3d", mode = "markers", color = ~cut)
}

ex_3_b %<-% {
  Sys.sleep(5)
  ggplot(diamonds, aes(x = carat, y = sqrt(price), color = cut)) +
    geom_point(alpha = 0.2) +
    geom_smooth() +
    theme_minimal()
}

ex_3_c %<-% {
  Sys.sleep(5)
  ggplot(diamonds, aes(x = price)) +
    geom_histogram() +
    theme_minimal()
}
```

Asynchronous futures and blocking

- These futures are non-blocking by default
- Blocking can still happen if the future is not yet resolved or if all background sessions are still busy (see example on last slide)
- A future can be checked without blocking by using `resolved()` (explicit futures) or `futureOf()` (implicit futures)

Example - checking futures

```
plan(multisession(workers = 2))

ftr_1 %<-% {
  cat("Implicit future on process", Sys.getpid(), "\n")
  Sys.sleep(5)
  getwd()
}

ftr_2 <- future({
  cat("Explicit future on process", Sys.getpid(), "\n")
  Sys.sleep(5)
  getwd()
})

# convert implicit to explicit future
ftr_1_check <- futureOf(ftr_1)
resolved(ftr_1_check) # check if ftr_1 is resolved

resolved(ftr_2) # check if ftr_2 is resolved
```

Other odds and ends

- View and adjust the options associated with futures by checking out `help("future.options")`
- Futures *do* relay output from `cat()`, `print()`, `str()` regardless of the strategy
- Relayed output can be captured with `capture.output()`
- Futures can be nested, i.e. one future can create another future, which can create another future

furrr

purrr and furrr

	
`map_*()`	`future_map_*()`
`map()`	`future_map()`
`map2()`	`future_map2()`
`pmap()`	`future_pmap()`
`walk()`	`future_walk()`
`imap()`	`future_imap()`
`modify()`	`future_modify()`

`furrr` uses `future`. The default backend for future (and through it, furrr) is a sequential one. This means that the above code will run out of the box, but it **will not be in parallel**. The design of future makes it incredibly easy to change this so that your code will run in parallel.

Examples

```
library(furrr)
library(tidyverse)
```

```
map_dbl(mtcars, mean)
```

```
#>        mpg        cyl       disp         hp       drat         wt       qsec
#>  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
#>         vs         am       gear       carb
#>   0.437500   0.406250   3.687500   2.812500
```

```
future_map_dbl(mtcars, mean)
```

```
#>        mpg        cyl       disp         hp       drat         wt       qsec
#>  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
#>         vs         am       gear       carb
#>   0.437500   0.406250   3.687500   2.812500
```

```
plan(multisession, workers = 4)
future_map_dbl(mtcars, mean)
```

```
#>        mpg        cyl       disp         hp       drat         wt       qsec
#>  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
#>         vs         am       gear       carb
#>   0.437500   0.406250   3.687500   2.812500
```

Not sure we are running in parallel?

```
system.time({map_dbl(mtcars, ~ {Sys.sleep(1); mean(.x)})})
```

```
#>    user  system elapsed
#>   0.006   0.000  11.032
```

```
system.time({future_map_dbl(mtcars, ~ {Sys.sleep(1); mean(.x)})})
```

```
#>    user  system elapsed
#>   0.052   0.002   3.147
```

```
plan(sequential)
```

Examples (continued)

Recall our data from Homework 03:

```
library(jsonlite)
library(janitor)
events_json <- read_json("data/events_england.json")
```

Suppose we have lots of games of data.

```
events_json <- rep(events_json, 100)
```

Some helper functions:

```
set_start_position_names <- function(x, y) {
  names(x$positions[[1]]) <- y
  x
}

set_tag_names <- function(x) {
  if (!is_empty(x$tags)) {
    names(x$tags) <- str_c("id_", 1:length(x$tags))
  }
  x
}
```

```
system.time({
events <- events_json %>%
  modify(set_start_position_names, c("start_y", "start_x")) %>%
  modify(set_tag_names) %>%
  map_df(unlist) %>%
  clean_names() %>%
  rename(
    start_y = positions_start_y,
    start_x = positions_start_x,
    end_y   = positions_y,
    end_x   = positions_x
  )
})
```

```
   user  system elapsed
  6.713   0.111   6.912
```

```
plan(multisession, workers = 4)

system.time({
events_parallel <- events_json %>%
  modify(set_start_position_names, c("start_y", "start_x")) %>%
  modify(set_tag_names) %>%
  future_map_dfr(unlist) %>%
  clean_names() %>%
  rename(
    start_y = positions_start_y,
    start_x = positions_start_x,
    end_y   = positions_y,
    end_x   = positions_x
  )
})

plan(sequential)
```

```
   user  system elapsed
 33.098   0.655  55.894
```

```
all_equal(events, events_parallel)
```

```
TRUE
```

What happened?

Chunking input

- Chunking is the process of breaking up your vector into smaller pieces that can be sent off to different workers to be processed in parallel.
- Once a worker gets a chunk of your vector, it maps over it calling `.f` on each element.
- The results from the workers are returned to the main R session once all chunks have been processed to be combined and returned from `future_map()`.

Internal function that controls chunking:

```
make_chunks <- furrr:::make_chunks
```

Default chunking strategy

```
make_chunks(n_x = 20, n_workers = 4)
```

```
#> [[1]]
#> [1] 1 2 3 4 5
#>
#> [[2]]
#> [1]  6  7  8  9 10
#>
#> [[3]]
#> [1] 11 12 13 14 15
#>
#> [[4]]
#> [1] 16 17 18 19 20
```

```
make_chunks(n_x = 20, n_workers = 3)
```

```
#> [[1]]
#> [1] 1 2 3 4 5 6 7
#>
#> [[2]]
#> [1]  8  9 10 11 12 13
#>
#> [[3]]
#> [1] 14 15 16 17 18 19 20
```

Default chunking strategy (continued)

```
make_chunks(n_x = 20, n_workers = 1)
```

```
#> [[1]]
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Dynamic chunking strategy

```
make_chunks(n_x = 16, n_workers = 2, scheduling = 2L)
```

```
#> [[1]]
#> [1] 1 2 3 4
#>
#> [[2]]
#> [1] 5 6 7 8
#>
#> [[3]]
#> [1]  9 10 11 12
#>
#> [[4]]
#> [1] 13 14 15 16
```

- Components 1 to 4 are sent to worker 1, and components 5 to 8 are sent to worker 2, the rest wait
- The first worker that is done will get components 9 to 12, components 13 to 16 wait
- The next worker that is done will get the last components 13 to 16

These chunking strategies can be set with `furrr_options()`.

Exercise

Suppose you have decided on the below as your final model.

```
lm(mpg ~ wt + hp, data = mtcars)
```

Use target shuffling and extract the R^2 metric for 1000 shuffles to assess the relationship. Compare the performance of doing this with `map_dbl()` and `future_map_dbl()`. Use the `microbenchmark` package.

References

1. "Apply Mapping Functions In Parallel Using Futures". Furrr.Futureverse.Org, 2021, https://furrr.futureverse.org/.

2. "Unified Parallel And Distributed Processing In R For Everyone". Future.Futureverse.Org, 2021, https://future.futureverse.org/.