Package 'async'

May 26, 2022

Title Asynchronous Code Constructs: Generators, Yield, Async, Await

Version 0.2.1

Date 2022-05-15

URL https://crowding.github.io/async/,
https://github.com/crowding/async/

BugReports https://github.com/crowding/async/issues

Description Write sequential-looking code that pauses and resumes.
gen() creates a generator, an iterator that returns a
value and pauses each time it reaches a yield() call.
async() creates a promise, which runs until it reaches
a call to await(), then resumes when information is available.
These work similarly to generator and async constructs
from 'Python' or 'JavaScript'. Objects produced are
compatible with the 'iterators' and 'promises' packages.

License GPL-2

Encoding UTF-8

Depends R (>= 3.5.0)

Imports nseval (>= 0.4), iterators, itertools, later, promises

Suggests rmarkdown, testthat (>= 3.0.0), knitr, magrittr, audio,
profvis, covr

RoxygenNote 7.1.2

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation no

Author Peter Meilstrup [aut, cre]

Maintainer Peter Meilstrup <peter.meilstrup@gmail.com>

Repository CRAN

Date/Publication 2022-05-26 08:30:02 UTC

R topics documented:

`async-package` *The async package.*

Description

The `async` package allows you to write sequential-looking code that can pause, return control to R, then pick up where it left off. Async constructs include generators and async/await blocks.

Details

A generator runs until it yields a value and then stops, returning control to R until another value is requested. An async block can pause and return control to R until some data is available, then resume. Generators implement the iterator interface, while async blocks implement the promise interface.

- `gen(...)` creates a generator (an iterator); within a generator use `yield(x)` to return a value.
- `async(...)` creates an async block (a promise); within the `async` write `await(x)` to pause on `x` (another promise).

Author(s)

Peter Meilstrup

`async` *Create an asynchronous task from sequential code.*

Description

`async({...})`, with an expression written in its argument, allows that expression to be evaluated in an asynchronous, or non-blocking manner. `async` returns an object with class `c("async", "promise")` which implements the promise interface.

Usage

```
async(expr, ..., split_pipes = TRUE, trace = trace_)

await(prom)

with_prefix(prefix)
```

Arguments

expr	An expression, to be executed asynchronously.
...	Undocumented.
split_pipes	Rewrite chained calls that use await (see below)
trace	Enable verbose logging by passing a function to trace, like async(trace=cat, {...}). trace should take a character argument. Helper with_prefix makes a function that prints a message with the given prefix. You can also say something like trace=browser for "single stepping" through an async.
prom	A promise, or something that can be converted to such by promises::as.promise().
prefix	Character prefix to print before the trace.

Details

An example Shiny app using async/await is on Github: https://github.com/crowding/cranwhales-await

When an async object is activated, it will evaluate its expression until it reaches the keyword await. The async object will return to its caller and preserve the partial state of its evaluation. When the awaited value is resolved, evaluation continues from where the async left off.

When an async block finishes (either by reaching the end, or using return()), the promise resolves with the resulting value. If the async block stops with an error, the promise is rejected with that error.

The syntax rules for an async are analogous to those for gen(); await must appear only within the arguments of functions for which there is a pausable implementation (See [pausables()]). By default split_pipes=TRUE is enabled and this will reorder some expressions to satisfy this requirement.

Async blocks and generators are conceptually related and share much of the same underlying mechanism. You can think of one as "output" and the other as "input". A generator pauses until a value is requested, runs until it has a value to output, then pauses again. An async runs until it requires an external value, pauses until it receives the value, then continues.

When split_pipes=FALSE, await() can only appear in the arguments of pausables and not ordinary R functions. This is a inconvenience as it prevents using await() in a pipeline. async by default has split_pipes=TRUE which enables some syntactic sugar: if an await() appears in the leftmost, unnamed, argument of an R function, the pipe will be "split" at that call using a temporary variable. For instance,

```
async(makeRequest() |> await() |> sort())
```

will be effectively rewritten to something like

```
async({.tmp <- await(makeRequest()); sort(.tmp)})
```

This works only so long as await appears in calls that evaluate their leftmost arguments normally. split_pipes can backfire if the outer call has other side effects; for instance suppressWarnings(await(x)) will be rewritten as {.tmp <- await(x); suppressWarnings(x)}, which would defeat the purpose.

Value

`async()` returns an object with class "promise" as described by the promises package (i.e. not the promises used in R's lazy evaluation.)

In the context of an `async`, `await(x)` returns the resolved value of a promise `x`, or stops with an error.

Examples

```
myAsync <- async(for (i in 1:4) {
  await(delay(5))
  cat(i, "\n")
}, trace=with_prefix("myAsync"))
```

`delay` *Asynchronous pause.*

Description

"delay" returns a promise which resolves only after the specified number of seconds. This uses the R event loop via later. In an `[async]` construct you can use `await(delay(secs))` to yield control, for example if you need to poll in a loop

Usage

```
delay(secs, expr = NULL)
```

Arguments

`secs`	The promise will resolve after at least this many seconds.
`expr`	The value to resolve with; will be forced after the delay.

Value

An object with class "promise".

Examples

```
# print a message after a few seconds
async({await(delay(10)); cat("Time's up!\n")})
```

gen *Create an iterator using sequential code.*

Description

`gen({...})` with an expression written in its argument, creates a generator, which acts like a block of code whose execution can pause and resume. From the "inside," a generator looks like you are writing sequential code with loops, branches and such, writing values to the outside world by calling `yield()`. From the "outside," a generator behaves like an iterator over an indefinite collection.

Usage

```
gen(expr, ..., split_pipes = FALSE, trace = trace_)

yield(expr)
```

Arguments

`expr`	An expression, to be turned into an iterator.
`...`	Undocumented.
`split_pipes`	Silently rewrite expressions where "yield" appears in chained calls. See async.
`trace`	Optional tracing function for debugging. See async.

Details

When `nextElem` is called on a generator, the generator executes its given expression until it reaches a call to `yield(...)`. `nextElem` returns argument to `yield` is returne, and the generator's execution state is preserved. The generator will resume on the next call to `nextElem()`.

The generator expression is evaluated in a local environment.

Generators are not based on forking or parallel OS processes; they run in the same thread as their caller. The control flow in a generator is interleaved with that of the R code which queries it.

A generator expression can use any R functions, but a call to `yield` may only appear in some positions. This package has several built-in pausables, equivalents to R's base control flow functions, such as `if`, `while`, `tryCatch`, `<-`, `{}`, `||` and so on. A call to `yield` may only appear in an argument of one of these pausable functions. So this random walk generator:

```
rwalk <- gen({x <- 0; repeat {x <- yield(x + rnorm(1))}})
```

is legal, because `yield` appears within arguments to `{}`, `repeat`, and `<-`, for which this package has pausable definitions. However, this:

```
rwalk <- gen({x <- rnorm(1); repeat {x <- rnorm(1) + yield(x)}})
```

is not legal, because `yield` appears in an argument to `+`, which does not have a pausable definition.

Value

An object with class "iterator".

pausables *Pausable functions.*

Description

async and gen rely on "pausable" workalikes for R functions like `if`, `while`, and so on. `pausables()` scans for and returns a list of all pausable functions visible in the present environment and in attached packages.

Usage

```
pausables(envir = caller(), packages = base::.packages())
```

Arguments

`envir`	The environment to search (defaulting to the calling environment).
`packages`	Which packages to search; defaults to currently loaded packages. You can scan all packages with `pausables(packages=base::.packages(all.available=TRUE))`

Details

It is possible for a third party package to define pausable functions. To do this:

1. Define and export a function `yourname` and an ordinary R implementation (the pausable version is only used when there is an `await` or `yield` in the arguments.)
2. Also define a function `yourname_cps` in your package namespace. (It does not need to be exported.) `yourname_cps` should have the pausable (callback based) implementation.

The API for pausable functions is not yet fixed, but it is described in source file `cps.r` along with implementations for R builtins.

Value

A list of expressions (either names or `:::` calls)

Index