Generate global option function

Zuguang Gu <z.gu@dkfz.de>

December 24, 2014

Global option function such as `options` and `par` can provide a way to control global settings. Here the *GlobalOptions* package can generate a function which takes care of such global settings.

1 General usage

The most simple use is to generate an option function with default values by callling `setGlobalOptions`:

```
library(GlobalOptions)
foo.options = setGlobalOptions(
    a = 1,
    b = "text"
)
```

The returned value `foo.options` is an option function which can be used to get or set options:

```
foo.options()
```

```
## $a
## [1] 1
##
## $b
## [1] "text"
```

```
foo.options("a")
```

```
## [1] 1
```

```
op = foo.options()
op
```

```
## $a
## [1] 1
##
## $b
## [1] "text"
```

```
foo.options(a = 2, b = "new text")
foo.options()
```

```
## $a
## [1] 2
##
## $b
## [1] "new text"
```

```
foo.options(op)
foo.options()
```

```
## $a
## [1] 1
##
## $b
## [1] "text"
```

`foo.options` generated by `setGlobalOptions` will contain an argument `RESET` which is used to reset the options to the default:

```
foo.options(a = 2, b = "new text")
foo.options(RESET = TRUE)
foo.options()

## $a
## [1] 1
##
## $b
## [1] "text"
```

2 Advanced usage

If option values are set as lists, more close controls can be customized. There are two basic fields that are used to check the input option values:

```
foo.options = setGlobalOptions(
    a = list(.value = 1,
             .length = c(1, 3),
             .class = "numeric")
)
```

In above code, `.value` is the default value for the option `a`, the length of the value is controlled by `.length` and the length should be either 1 or 3, and the class of the value should be `numeric`. If the input value does not fit the criterion, there will be an error. The value of `.length` or `.class` is a vector and the checking will be passed if one of the value fits user's input.

```
foo.options(a = 1:2)

## Error in foo.options(a = 1:2):  Length of 'a' should be one of 1, 3.

foo.options(a = "text")

## Error in foo.options(a = "text"):  Class of 'a' should be one of 'numeric'.
```

User can set the value as read-only and modifying such option will cause an error.

```
foo.options = setGlobalOptions(
    a = list(.value = 1,
             .read.only = TRUE)
)
foo.options(a = 2)

## Error in foo.options(a = 2):  'a' is a read-only option.
```

There is also a pre-defined argument `READ.ONLY` exported with `foo.options` which controls whether to return only the read-only options or not.

```
foo.options = setGlobalOptions(
    a = list(.value = 1,
             .read.only = TRUE),
    b = 2
)
foo.options()
```

```
## $a
## [1] 1
##
## $b
## [1] 2

foo.options(READ.ONLY = TRUE)

## $a
## [1] 1

foo.options(READ.ONLY = FALSE)

## $b
## [1] 2
```

Validation of the option values can be controlled by `.validate` field. The value of `.validate` should be a function. The input of the validation function is the input option value and the function should only return a logical value.

```
foo.options = setGlobalOptions(
    a = list(.value = 1,
             .validate = function(x) x > 0 && x < 10)
)
foo.options(a = 20)

## Error in tryCatchList(expr, classes, parentenv, handlers):  Your option is invalid.
```

Filtering on the option values can be controlled by `.filter` field. This is useful when the input option value is not valid but it is not necessary to throw errors. More proper way is to modify the value silently. For example, there is an option to control whether to print messages or not and it should be set to `TRUE` or `FALSE`. However, users may set some other type of values such as `NULL` or `NA`. In this case, non-`TRUE` values can be converted to logical values by `.filter`. Similar as `.validate`, the input value for filter function is the input option value, and it should return a filtered option value.

```
foo.options = setGlobalOptions(
    verbose =
        list(.value = TRUE,
             .filter = function(x) {
                 if(is.null(x)) {
                     return(FALSE)
                 } else if(is.na(x)) {
                     return(FALSE)
                 } else {
                     return(x)
                 }
             })
)
foo.options(verbose = FALSE); foo.options("verbose")

## [1] FALSE

foo.options(verbose = NA); foo.options("verbose")

## [1] FALSE

foo.options(verbose = NULL); foo.options("verbose")

## [1] FALSE
```

The input option value can be set as dynamic by setting it as a function. When the option value is set as a function, it will be executed when querying the option. In the following example, the `prefix` option corresponds to the prefix of log messages. The returned option value is the string after the execution of the input function and the function itself is attached as an attribute of the returned option value.

```
foo.options = setGlobalOptions(
    prefix = ""
)
foo.options(prefix = function() paste("[", Sys.time(), "] ", sep = " "))
foo.options("prefix")

## [1] "[ 2014-12-24 00:28:53 ] "
## attr(,"FUN")
## function ()
## paste("[", Sys.time(), "] ", sep = " ")

Sys.sleep(2)
foo.options("prefix")

## [1] "[ 2014-12-24 00:28:55 ] "
## attr(,"FUN")
## function ()
## paste("[", Sys.time(), "] ", sep = " ")
```

If the value of the option is really a function and users don't want to execute it, just set `.class` to contain `function`, then the function will be treated as a simple value.

```
foo.options = setGlobalOptions(
    test = list(.value = function(x1, x2) t.test(x1, x2)$p.value,
                .class = "function")
)
foo.options(test = function(x1, x2) cor.test(x1, x2)$p.value)
foo.options("test")

## function(x1, x2) cor.test(x1, x2)$p.value
```

The self-defined function (*i.e.* value function, validation function or filter function) is applied per-option. But if it relies on other option values, there is pre-defined variable `OPT` which is a list containing values for all options. In following example, default value of option `b` is two times of the value of option `a`.

```
foo.options = setGlobalOptions(
    a = list(.value = 1),
    b = list(.value = function() 2 * OPT$a)
)
foo.options("b")

## [1] 2
## attr(,"FUN")
## function ()
## 2 * OPT$a

foo.options(a = 2)
foo.options("b")

## [1] 4
## attr(,"FUN")
## function ()
## 2 * OPT$a
```

And in the second example, sign of `b` should be as same as sign of `a`.

```
foo.options = setGlobalOptions(
    a = list(.value = 1),
    b = list(.value = 0,
             .validate = function(x) {
                 if(OPT$a > 0) x > 0
                 else x < 0
             },
             .filter = function(x) {
                 x + OPT$a
             })
)
foo.options(b = 1)
foo.options("b")

## [1] 3

foo.options(a = 1, b = -1)

## Error in tryCatchList(expr, classes, parentenv, handlers):  Your option is invalid.
```

3 Features for package development

Two additional fields may be helpful when developing packages. `.visible` controls whether options are visible to users. If options names are specified as empty or a vector with length more than one when querying options, the invisible options will be hidden. The invisible option can only be queried or modified by specifying its single option name (just like you can only open the door with the correct unique key). This would be helpful if users want to put some secret options while do not want others to access. Is this case, they can assign names with complex strings like `.__MY_PRIVATE_KEY__.` as their secret options and afterwards they can access it with this special key.

```
foo.options = setGlobalOptions(
        a = list(.value = 1,
                 .visible = FALSE),
        b = 2
)
foo.options()

## $b
## [1] 2

foo.options("a")

## [1] 1

foo.options(a = 2)
foo.options("a")

## [1] 2
```

Another field `.private` controls whether the option is only private to the package. If it is set to `TRUE`, the option can only be modified in the same namespace (or top environment) where the option function is generated. E.g, if you are writing a package named `foo` and generating an option function `foo.options`, by setting the option with `.private` to `TRUE`, the value for such options can only be modified inside `foo` package while it is not permitted outside `foo`. At the same time, private options become read-only options if querying outside `foo` package.

The option function generated by `setGlobalOptions` contains four arguments: `...`, `RESET`, `READ.ONLY`. If you want to put the option function into a package, remember to document all the three arguments:

```
args(foo.options)

## function (..., RESET = FALSE, READ.ONLY = NULL)
## NULL
```

4 Misc

The order of checking when setting an option value is `.read.only`, `.private`, `.length`, `.class`, `.validate`, `.filter`, `.length`, `.class`. Note checking on length and class of the option values will be applied again after filtering.

Global options are stored in a private environment. Each time when generating a option function, there will be a new environment created. Thus global options will not conflict if they come from different option functions.

```
foo.options1 = setGlobalOptions(
    a = list(.value = 1)
)
foo.options2 = setGlobalOptions(
    a = list(.value = 1)
)
foo.options1(a = 2)
foo.options1("a")

## [1] 2

foo.options2("a")

## [1] 1
```

Note the option values can also be set as a list, so for the list containing configurations, names of the field is started with a dot `.` to be distinguished from the normal list.

The final and the most important thing is the check by `.class`, `.length`, `.validate`, `.filter` will not be applied on default values when calling `setGlobalOptions` because users who design their option functions by `setGlobalOptions` should know whether the default values are valid or not.