Package: pool (via r-universe)

September 20, 2024

Type Package

Title Object Pooling

Version 1.0.3.9000

Description Enables the creation of object pools, which make it less computationally expensive to fetch a new object. Currently the only supported pooled objects are 'DBI' connections.

License MIT + file LICENSE

URL https://github.com/rstudio/pool, https://rstudio.github.io/pool/

BugReports https://github.com/rstudio/pool/issues

Depends methods, R (>= 3.6.0)

Imports DBI (>= 1.2.1), later (>= 1.0.0), R6, rlang (>= 1.0.0)

Suggests covr, dbplyr (>= 2.4.0), dplyr, knitr, rmarkdown, RMySQL, RSQLite, shiny, testthat (>= 3.0.0), tibble

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Repository https://posit-dev-shinycoreci.r-universe.dev

RemoteUrl https://github.com/rstudio/pool

RemoteRef HEAD

RemoteSha 0fa16fa29fd6439aa13f383ce89c8934427fd6bf

Contents

`DBI-custom` *Unsupported DBI methods*

Description

Most pool methods for DBI generics check out a connection, perform the operation, and the return the connection to the pool, as described in DBI-wrap.

This page describes the exceptions:

- `DBI::dbSendQuery()` and `DBI::dbSendStatement()` can't work with pool because they return result sets that are bound to a specific connection. Instead use `DBI::dbGetQuery()`, `DBI::dbExecute()`, or `localCheckout()`.
- `DBI::dbBegin()`, `DBI::dbRollback()`, `DBI::dbCommit()`, and `DBI::dbWithTransaction()` can't work with pool because transactions are bound to a connection. Instead use `poolWithTransaction()`.
- `DBI::dbDisconnect()` can't work because pool handles disconnection.
- `DBI::dbGetInfo()` returns information about the pool, not the database connection.
- `DBI::dbIsValid()` returns whether or not the entire pool is valid (i.e. not closed).

Usage

```
## S4 method for signature 'Pool'
dbSendQuery(conn, statement, ...)

## S4 method for signature 'Pool,ANY'
dbSendStatement(conn, statement, ...)

## S4 method for signature 'Pool'
dbDisconnect(conn, ...)

## S4 method for signature 'Pool'
dbGetInfo(dbObj, ...)

## S4 method for signature 'Pool'
dbIsValid(dbObj, ...)

## S4 method for signature 'Pool'
dbBegin(conn, ...)

## S4 method for signature 'Pool'
dbCommit(conn, ...)
```

```
## S4 method for signature 'Pool'
dbRollback(conn, ...)

## S4 method for signature 'Pool'
dbWithTransaction(conn, code)
```

Arguments

conn, dbObj	A Pool object, as returned from dbPool().
statement, code, ...	See DBI documentation.

---

dbPool *Create a pool of database connections*

Description

dbPool() is a drop-in replacement for DBI::dbConnect() that provides a shared pool of connections that can automatically reconnect to the database if needed.

Usage

```
dbPool(
  drv,
  ...,
  minSize = 1,
  maxSize = Inf,
  onCreate = NULL,
  idleTimeout = 60,
  validationInterval = 60,
  validateQuery = NULL
)
```

Arguments

drv	A DBI Driver, e.g. RSQLite::SQLite(), RPostgres::Postgres(), odbc::odbc() etc.
...	Arguments passed on to DBI::dbConnect(). These are used to identify the database and provide needed authentication.
minSize, maxSize	The minimum and maximum number of objects in the pool.
onCreate	A function that takes a single argument, a connection, and is called when the connection is created. Use this with DBI::dbExecute() to set default options on every connection created by the pool.
idleTimeout	Number of seconds to wait before destroying idle objects (i.e. objects available for checkout over and above minSize).

validationInterval
: Number of seconds to wait between validating objects that are available for checkout. These objects are validated in the background to keep them alive.

  To force objects to be validated on every checkout, set `validationInterval = 0`.

validateQuery
: A simple query that can be used to verify that the connetction is valid. If not provided, `dbPool()` will try a few common options, but these don't work for all databases.

Examples

```
# You use a dbPool in the same way as a standard DBI connection
pool <- dbPool(RSQLite::SQLite())
pool

DBI::dbWriteTable(pool, "mtcars", mtcars)
dbGetQuery(pool, "SELECT * FROM mtcars LIMIT 4")

# Always close a pool when you're done using it
poolClose(pool)

# Using the RMySQL package
if (requireNamespace("RMySQL", quietly = TRUE)) {
  pool <- dbPool(
    drv = RMySQL::MySQL(),
    dbname = "shinydemo",
    host = "shiny-demo.csa7qlmguqrf.us-east-1.rds.amazonaws.com",
    username = "guest",
    password = "guest"
  )

  dbGetQuery(pool, "SELECT * from City LIMIT 5;")

  poolClose(pool)
}
```

Pool-class *Create a pool of reusable objects*

Description

A generic pool class that holds objects. These can be fetched from the pool and released back to it at will, with very little computational cost. The pool should be created only once and closed when it is no longer needed, to prevent leaks.

Every usage of `poolCreate()` should always be paired with a call to `poolClose()` to avoid "leaking" resources. In shiny app, you should create the pool outside of the server function and close it on stop, i.e. `onStop(function() pool::poolClose(pool))`.

See `dbPool()` for an example of object pooling applied to DBI database connections.

Usage

```
poolCreate(
  factory,
  minSize = 1,
  maxSize = Inf,
  idleTimeout = 60,
  validationInterval = 60,
  state = NULL
)

poolClose(pool)

## S4 method for signature 'Pool'
poolClose(pool)
```

Arguments

`factory`	A zero-argument function called to create the objects that the pool will hold (e.g. for DBI database connections, `dbPool()` uses a wrapper around `DBI::dbConnect()`).
`minSize, maxSize`	The minimum and maximum number of objects in the pool.
`idleTimeout`	Number of seconds to wait before destroying idle objects (i.e. objects available for checkout over and above `minSize`).
`validationInterval`	Number of seconds to wait between validating objects that are available for checkout. These objects are validated in the background to keep them alive. To force objects to be validated on every checkout, set `validationInterval = 0`.
`state`	A `pool` public variable to be used by backend authors.
`pool`	A Pool object previously created with `poolCreate`

`poolCheckout` *Check out and return object from the pool*

Description

Use `poolCheckout()` to check out an object from the pool and `poolReturn()` to return it. You will receive a warning if all objects aren't returned before the pool is closed.

`localCheckout()` is a convenience function that can be used inside functions (and other function-scoped operations like `shiny::reactive()` and `local()`). It checks out an object and automatically returns it when the function exits

Note that validation is only performed when the object is checked out, so you generally want to keep the checked out around for as little time as possible.

When pooling DBI database connections, you normally would not use `poolCheckout()`. Instead, for single-shot queries, treat the pool object itself as the DBI connection object and it will perform checkout/return for you. And for transactions, use `poolWithTransaction()`.

Usage

```
poolCheckout(pool)

## S4 method for signature 'Pool'
poolCheckout(pool)

poolReturn(object)

## S4 method for signature 'ANY'
poolReturn(object)

localCheckout(pool, env = parent.frame())
```

Arguments

pool	The pool to get the object from.
object	Object to return
env	Environment corresponding to the execution frame. For expert use only.

Examples

```
pool <- dbPool(RSQLite::SQLite())
# For illustration only. You normally would not explicitly use
# poolCheckout with a DBI connection pool (see Description).
con <- poolCheckout(pool)
con
poolReturn(con)

f <- function() {
  con <- localCheckout(pool)
  # do something ...
}
f()

poolClose(pool)
```

---

poolWithTransaction *Self-contained database transactions using pool*

Description

This function allows you to use a pool object directly to execute a transaction on a database connection, without ever having to actually check out a connection from the pool and then return it. Using this function instead of the direct transaction methods will guarantee that you don't leak connections or forget to commit/rollback a transaction.

Usage

```
poolWithTransaction(pool, func)
```

Arguments

pool	The pool object to fetch the connection from.
func	A function that has one argument, conn (a database connection checked out from pool).

Details

This function is similar to DBI::dbWithTransaction(), but its arguments work a little differently. First, it takes in a pool object, instead of a connection. Second, instead of taking an arbitrary chunk of code to execute as a transaction (i.e. either run all the commands successfully or not run any of them), it takes in a function. This function (the func argument) gives you an argument to use in its body, a database connection. So, you can use connection methods without ever having to check out a connection. But you can also use arbitrary R code inside the func's body. This function will be called once we fetch a connection from the pool. Once the function returns, we release the connection back to the pool.

Like its DBI sister DBI::dbWithTransaction(), this function calls dbBegin() before executing the code, and dbCommit() after successful completion, or dbRollback() in case of an error. This means that calling poolWithTransaction always has side effects, namely to commit or roll back the code executed when func is called. In addition, if you modify the local R environment from within func (e.g. setting global variables, writing to disk), these changes will persist after the function has returned.

Also, like DBI::dbWithTransaction(), there is also a special function called dbBreak() that allows for an early, silent exit with rollback. It can be called only from inside poolWithTransaction.

Value

func's return value.

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
  pool <- dbPool(RSQLite::SQLite(), dbname = ":memory:")

  dbWriteTable(pool, "cars", head(cars, 3))
  dbReadTable(pool, "cars")   # there are 3 rows

  ## successful transaction
  poolWithTransaction(pool, function(conn) {
    dbExecute(conn, "INSERT INTO cars (speed, dist) VALUES (1, 1);")
    dbExecute(conn, "INSERT INTO cars (speed, dist) VALUES (2, 2);")
    dbExecute(conn, "INSERT INTO cars (speed, dist) VALUES (3, 3);")
  })
  dbReadTable(pool, "cars")   # there are now 6 rows

  ## failed transaction -- note the missing comma
```

```
  tryCatch(
    poolWithTransaction(pool, function(conn) {
      dbExecute(conn, "INSERT INTO cars (speed, dist) VALUES (1, 1);")
      dbExecute(conn, "INSERT INTO cars (speed dist) VALUES (2, 2);")
      dbExecute(conn, "INSERT INTO cars (speed, dist) VALUES (3, 3);")
    }),
    error = identity
  )
  dbReadTable(pool, "cars")   # still 6 rows

  ## early exit, silently
  poolWithTransaction(pool, function(conn) {
    dbExecute(conn, "INSERT INTO cars (speed, dist) VALUES (1, 1);")
    dbExecute(conn, "INSERT INTO cars (speed, dist) VALUES (2, 2);")
    if (nrow(dbReadTable(conn, "cars")) > 7) dbBreak()
    dbExecute(conn, "INSERT INTO cars (speed, dist) VALUES (3, 3);")
  })
  dbReadTable(pool, "cars")   # still 6 rows

  poolClose(pool)

} else {
  message("Please install the 'RSQLite' package to run this example")
}
```

`tbl.Pool` *Use pool with dbplyr*

Description

Wrappers for key dplyr (and dbplyr) methods so that pool works seemlessly with dbplyr.

Usage

```
tbl.Pool(src, from, ..., vars = NULL)

copy_to.Pool(dest, df, name = NULL, overwrite = FALSE, temporary = TRUE, ...)
```

Arguments

`src, dest`	A dbPool.
`from`	Name table or `dbplyr::sql()` string.
`...`	Other arguments passed on to the individual methods
`vars`	A character vector of variable names in `src`. For expert use only.
`df`	A local data frame, a `tbl_sql` from same source, or a `tbl_sql` from another source. If from another source, all data must transition through R in one pass, so it is only suitable for transferring small amounts of data.

name	Name for remote table. Defaults to the name of df, if it's an identifier, otherwise uses a random name.
overwrite	If TRUE, will overwrite an existing table with name name. If FALSE, will throw an error if name already exists.
temporary	if TRUE, will create a temporary table that is local to this connection and will be automatically deleted when the connection expires

Examples

```
library(dplyr)

pool <- dbPool(RSQLite::SQLite())
# copy a table into the database
copy_to(pool, mtcars, "mtcars", temporary = FALSE)

# retrieve a table
mtcars_db <- tbl(pool, "mtcars")
mtcars_db
mtcars_db %>% select(mpg, cyl, disp)
mtcars_db %>% filter(cyl == 6) %>% collect()

poolClose(pool)
```

Index