Package 'pool'

January 14, 2021

Type Package

Title Object Pooling

Version 0.1.6

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 GPL-3

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

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

Depends methods, R (>= 3.0.0)

Imports DBI, R6, later (>= 1.0.0)

Suggests testthat, tibble, dplyr, dbplyr, RSQLite, RMySQL

Encoding UTF-8

LazyData TRUE

RoxygenNote 7.1.1

Collate 'utils.R' 'scheduler.R' 'pool.R' 'pool-methods.R' 'object.R'
'DBI.R' 'DBI-connection-interpolate.R' 'DBI-connection-quote.R'
'DBI-connection-sql.R' 'DBI-connection-transaction.R'
'DBI-connection.R' 'DBI-driver.R' 'DBI-object-deprecated.R'
'DBI-object.R' 'DBI-pool.R' 'dbplyr.R' 'zzz-hooks.R'

NeedsCompilation no

Author Joe Cheng [aut, cre],
Barbara Borges [aut],
RStudio [cph]

Maintainer Joe Cheng <joe@rstudio.com>

Repository CRAN

Date/Publication 2021-01-14 21:20:03 UTC

R topics documented:

`DBI-connection` *DBIConnection methods.*

Description

As a convenience, Pool implements DBIConnection methods; calling any implemented DBI method directly on a Pool object will result in a connection being checked out (with `poolCheckout()`), the operation being performed on that connection, and the connection being returned to the pool (with `poolReturn()`).

Usage

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

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

## S4 method for signature 'Pool,character'
dbGetQuery(conn, statement, ...)

## S4 method for signature 'Pool,character'
dbExecute(conn, statement, ...)

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

## S4 method for signature 'Pool,character'
dbListFields(conn, name, ...)
```

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

## S4 method for signature 'Pool'
dbListObjects(conn, prefix = NULL, ...)

## S4 method for signature 'Pool,character'
dbReadTable(conn, name, ...)

## S4 method for signature 'Pool,ANY'
dbWriteTable(conn, name, value, ...)

## S4 method for signature 'Pool'
dbCreateTable(conn, name, fields, ..., row.names = NULL, temporary = FALSE)

## S4 method for signature 'Pool'
dbAppendTable(conn, name, value, ..., row.names = NULL)

## S4 method for signature 'Pool,ANY'
dbExistsTable(conn, name, ...)

## S4 method for signature 'Pool,ANY'
dbRemoveTable(conn, name, ...)

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

Arguments

`conn, dbObj`	A Pool object, as returned from `dbPool()`.
`statement, name, value, prefix, fields, row.names, temporary, ...`	See DBI documentation.

Details

Pool cannot implement the `DBI::dbSendQuery()` and `DBI::dbSendStatement()` methods because they both return live ResultSet objects. This is incompatible with the Pool model, because once a connection is returned to the pool, using an existing ResultSet object could give erroneous results, throw an error, or even crash the entire R process. In most cases, `DBI::dbGetQuery()` and `DBI::dbExecute()` can be used instead. If you really need the control that `dbSendQuery` gives you (for example, to process a large table in chunks) then use `poolCheckout()` to get a real connection object (and don't forget to return it to the pool using `poolReturn()` afterwards).

See Also

For the original documentation, see:

- `DBI::dbSendQuery()` (not implemented by Pool)

- DBI::dbSendStatement() (not implemented by Pool)
- DBI::dbAppendTable()
- DBI::dbCreateTable()
- DBI::dbGetQuery()
- DBI::dbExecute()
- DBI::dbIsReadOnly()
- DBI::dbListFields()
- DBI::dbListObjects()
- DBI::dbListResults()
- DBI::dbListTables()
- DBI::dbReadTable()
- DBI::dbWriteTable()
- DBI::dbExistsTable()
- DBI::dbRemoveTable()

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
  mtcars1 <- mtcars[ c(1:16), ] # first half of the mtcars dataset
  mtcars2 <- mtcars[-c(1:16), ] # second half of the mtcars dataset

  pool <- dbPool(RSQLite::SQLite(), dbname = ":memory:")

  # write the mtcars1 table into the database
  dbWriteTable(pool, "mtcars", mtcars1, row.names = TRUE)

  # list the current tables in the database
  dbListTables(pool)

  # read the "mtcars" table from the database (only 16 rows)
  dbReadTable(pool, "mtcars")

  # append mtcars2 to the "mtcars" table already in the database
  dbWriteTable(pool, "mtcars", mtcars2, row.names = TRUE, append = TRUE)

  # read the "mtcars" table from the database (all 32 rows)
  dbReadTable(pool, "mtcars")

  # get the names of the columns in the databases's table
  dbListFields(pool, "mtcars")

  # use dbExecute to change the "mpg" and "cyl" values of the 1st row
  dbExecute(pool,
    paste(
      "UPDATE mtcars",
      "SET mpg = '22.0', cyl = '10'",
      "WHERE row_names = 'Mazda RX4'"
```

```
  )
)

# read the 1st row of "mtcars" table to confirm the previous change
dbGetQuery(pool, "SELECT * FROM mtcars WHERE row_names = 'Mazda RX4'")

# drop the "mtcars" table from the database
dbRemoveTable(pool, "mtcars")

# list the current tables in the database
dbListTables(pool)

poolClose(pool)

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

DBI-connection-interpolate *Safely interpolate values into an SQL string and parse them back.*

Description

Pool object wrappers around DBIConnection methods that deal with the safe interpolation of values into an SQL string and the reverse – parsing interpolated variables from SQL. See DBI::sqlInterpolate() and DBI::sqlParseVariables() for the original documentation.

Usage

```
## S4 method for signature 'Pool'
sqlInterpolate(conn, sql, ..., .dots = list())

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

Arguments

conn, sql, ..., .dots
: See DBI::sqlInterpolate().

DBI-connection-quote *SQL quoting.*

Description

Pool object wrappers around DBIConnection methods that deal with SQL escaping needs. See `DBI::SQL()` for the original documentation.

Usage

```
## S4 method for signature 'Pool,ANY'
dbQuoteIdentifier(conn, x, ...)

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

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

## S4 method for signature 'Pool,ANY'
dbQuoteString(conn, x, ...)
```

Arguments

`conn, x, ...`	See `DBI::SQL()`.

DBI-connection-sql *Creating and manipulating SQL tables.*

Description

Pool object wrappers around DBIConnection methods that deal with the creation and manipulation of SQL tables. See `DBI::sqlData()`, `DBI::sqlCreateTable()` and `DBI::sqlAppendTable()` for the original documentation.

Usage

```
## S4 method for signature 'Pool'
sqlData(con, value, row.names = NA, ...)

## S4 method for signature 'Pool'
sqlCreateTable(con, table, fields, row.names = NA, temporary = FALSE, ...)

## S4 method for signature 'Pool'
sqlAppendTable(con, table, values, row.names = NA, ...)
```

Arguments

con, value, row.names, ...	See DBI::sqlData().
table, fields, temporary	See DBI::sqlCreateTable().
values	See DBI::sqlAppendTable().

DBI-connection-transaction

DBIConnection transaction methods are not supported for Pool objects.

Description

You cannot perform SQL transaction using a Pool object directly (because that would imply keeping a connection open and not knowing when to return it back to the pool).

Usage

```
## 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, ..., code	See DBI::transactions().

Details

If you must use these methods, fetch an actual connection first with `conn <-poolCheckout(pool)` – then call the appropriate DBI method on `conn`. Since you're fetching a connection from the pool yourself, you must also remember to return it back to the pool when you're done: `poolReturn(conn)` (otherwise, you have a leaked connection).

For simple transactions, consider using `poolWithTransaction()` instead, which is safer since it does not require you to fetch and release the connection yourself.

See `DBI::transactions()` for the original documentation.

DBI-object *DBIObject methods.*

Description

Pool object wrappers around DBIObject methods. See DBI::dbDataType(), DBI::dbGetInfo() and DBI::dbIsValid() for the original documentation.

Usage

```
## S4 method for signature 'Pool'
dbDataType(dbObj, obj, ...)

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

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

Arguments

dbObj, obj, ... See DBI::dbDataType().

Examples

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

  dbGetInfo(pool)
  dbIsValid(pool)

  dbDataType(pool, 1:5)
  dbDataType(pool, 1)
  dbDataType(pool, TRUE)
  dbDataType(pool, Sys.Date())
  dbDataType(pool, Sys.time())
  dbDataType(pool, Sys.time() - as.POSIXct(Sys.Date()))
  dbDataType(pool, c("x", "abc"))
  dbDataType(pool, list(raw(10), raw(20)))
  dbDataType(pool, I(3))
  dbDataType(pool, iris)

  poolClose(pool)

  dbIsValid(pool)

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

DBI-object-deprecated *Make R identifiers into legal SQL identifiers.*

Description

Pool object wrappers around DBIObject methods. See DBI::make.db.names() for the original documentation. Note that these methods are DEPRECATED. Please use dbQuoteIdentifier (or possibly dbQuoteString) instead, as documented in DBI-connection-quote().

Usage

```
## S4 method for signature 'Pool,character'
make.db.names(
  dbObj,
  snames,
  keywords = .SQL92Keywords,
  unique = TRUE,
  allow.keywords = TRUE,
  ...
)

## S4 method for signature 'Pool,character'
isSQLKeyword(
  dbObj,
  name,
  keywords = .SQL92Keywords,
  case = c("lower", "upper", "any")[3],
  ...
)

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

Arguments

dbObj, snames, keywords, unique, allow.keywords, name, case, ...
: see DBI::make.db.names()

dbPool *Create a DBI Database Connection Pool.*

Description

A wrapper around poolCreate to simplify the creation of a DBI database connection pool. Check the documentation of poolCreate() for a generic overview of the parent function and the Pool object. The main thing to point out is that, for dbPool, you always need to provide a DBI driver (i.e. of class DBI::DBIDriver-class()), and it should always be accompanied by the required authorization arguments (see the example below).

Usage

```
dbPool(drv, ..., validateQuery = NULL)
```

Arguments

drv	An object that inherits from DBI::DBIDriver-class(), or an existing DBI::DBIConnection-class() object (in order to clone an existing connection).
...	Arguments needed for both DBI::dbConnect() (mandatory if required by the DBIDriver you're using) and poolCreate() (optional) - all arguments should be named: 1. The required authorization arguments needed by the DBMS instance; these typically include user, password, dbname, host, port, etc. For details check the appropriate DBIDriver's documentation. 2. Optionally, override the poolCreate defaults: minSize (minimum number of connections that the pool should have at all times), maxSize (maximum number of connections that the pool may have at any time), idleTimeout (number of seconds to wait before closing a connection, if the number of connection is above minSize), and validationInterval (number of seconds to wait before validating the connection again).
validateQuery	The query to run to verify that the connection is valid (it should be as simple as possible). If this is not provided, dbPool will try a few possibilities, but these are not exhaustive.

Examples

```
# 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;")
  #>   ID           Name CountryCode      District Population
  #> 1  1          Kabul         AFG         Kabol    1780000
  #> 2  2       Qandahar         AFG      Qandahar     237500
  #> 3  3          Herat         AFG         Herat     186800
  #> 4  4 Mazar-e-Sharif         AFG         Balkh     127800
```

```
  #> 5  5      Amsterdam         NLD Noord-Holland     731200

  poolClose(pool)

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

# Using a DSN with an ODBC driver
## Not run:
pool <- dbPool(odbc::odbc(), dsn = "Data Source Name")

## End(Not run)
```

---

dplyr-db-methods *DBIConnection methods from dplyr and dbplyr*

Description

Pool object wrappers around DBIConnection methods, whose generics are defined either in dplyr or in dbplyr. For the original documentation, see dplyr's reference page and dbplyr's reference page.

Usage

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

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

db_analyze.Pool(con, table, ...)

db_begin.Pool(con, ...)

db_commit.Pool(con, ...)

db_create_index.Pool(con, table, columns, name = NULL, unique = FALSE, ...)

db_create_indexes.Pool(con, table, indexes = NULL, unique = FALSE, ...)

db_create_table.Pool(con, table, types, temporary = FALSE, ...)
```

```
db_data_type.Pool(con, fields)

db_desc.Pool(x)

db_drop_table.Pool(con, table, force = FALSE, ...)

db_explain.Pool(con, sql, ...)

db_has_table.Pool(con, table)

db_insert_into.Pool(con, table, values, ...)

db_list_tables.Pool(con)

db_query_fields.Pool(con, sql, ...)

db_query_rows.Pool(con, sql, ...)

db_rollback.Pool(con, ...)

db_save_query.Pool(con, sql, name, temporary = TRUE, ...)

db_write_table.Pool(con, table, types, values, temporary = FALSE, ...)

sql_escape_string.Pool(con, x)

sql_join.Pool(con, x, y, vars, type = "inner", by = NULL, ...)

sql_select.Pool(
  con,
  select,
  from,
  where = NULL,
  group_by = NULL,
  having = NULL,
  order_by = NULL,
  limit = NULL,
  distinct = FALSE,
  ...
)

sql_semi_join.Pool(con, x, y, anti = FALSE, by = NULL, ...)

sql_subquery.Pool(con, from, name = random_table_name(), ...)

sql_translate_env.Pool(con)

db_collect.Pool(con, sql, n = -1, warn_incomplete = TRUE, ...)
```

```
db_compute.Pool(
  con,
  table,
  sql,
  temporary = TRUE,
  unique_indexes = list(),
  indexes = list(),
  ...
)

db_sql_render.Pool(con, sql, ...)

sql_escape_logical.Pool(con, x)

sql_join_suffix.Pool(con, ...)
```

Arguments

`dest`, `df`, `name`, `overwrite`, `temporary`, `...`, `src`, `from`, `con`, `table`, `columns`, `unique`, `indexes`, `types`, `fields`, See original documentation.

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
  library(dplyr)

  db <- tempfile()
  pool <- dbPool(RSQLite::SQLite(), dbname = db)

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

Pool-class *S4 class for compatibility with DBI methods*

Description

A generic pool class that holds objects. These can be fetched

Usage

```
poolCreate(
  factory,
  minSize = 1,
  maxSize = Inf,
  idleTimeout = 60,
  validationInterval = 600,
  state = NULL
)

poolClose(pool)

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

Arguments

factory	A factory function responsible for the generation of the objects that the pool will hold (ex: for DBI database connections, this function is dbConnect). It must take no arguments.
minSize	An optional number specifying the minimum number of objects that the pool should have at all times.
maxSize	An optional number specifying the maximum number of objects that the pool may have at any time.
idleTimeout	The number of seconds that an idle object will be kept in the pool before it is destroyed (only applies if the number of objects is over the minSize). Use Inf if you want created objects never to be destroyed (there isn't a great reason for this usually).
validationInterval	The minimum number of seconds that pool will wait before running a validation check on the next checked out object. By not necessarily validating every checked out object, there can be substantial performance gains (especially if the interval between checking out new objects is very small).
state	A pool public variable to be used by backend authors as necessary.
pool	A Pool object previously created with poolCreate

---

poolCheckout *Checks out an object from the pool.*

Description

Should be called by the end user if they need a persistent object, that is not returned to the pool automatically. When you don't longer need the object, be sure to return it to the pool using poolReturn(object).

Usage

```
poolCheckout(pool)

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

Arguments

pool	The pool to get the object from.

---

poolReturn *Returns an object back to the pool.*

Description

Should be called by the end user if they previously fetched an object directly using `object <-poolCheckout(pool)` and are now done with said object.

Usage

```
poolReturn(object)

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

Arguments

object	A pooled object.

---

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)

dbBreak()
```

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")
}
```

`show,Pool-method` *Show method*

Description

Show method

Usage

```
## S4 method for signature 'Pool'
show(object)
```

Arguments

`object`	A Pool object.

Index