Package 'RcppSimdJson'

July 7, 2020

Type Package

Title 'Rcpp' Bindings for the 'simdjson' Header-Only Library for 'JSON' Parsing

Version 0.1.0

Date 2020-07-07

Author Dirk Eddelbuettel, Brendan Knapp

Maintainer Dirk Eddelbuettel <edd@debian.org>

Description The 'JSON' format is ubiquitous for data interchange, and the 'simdjson' library written by Daniel Lemire (and many contributors) provides a high-performance parser for these files which by relying on parallel 'SIMD' instruction manages to parse these files as faster than disk speed. See the <arXiv:1902.08318> paper for more details about 'simdjson'. This package is at present still a fairly thin and not fully complete wrapper that does not aim to replace the existing and excellent 'JSON' packages for R.

License GPL (>= 2)

Imports Rcpp, utils

LinkingTo Rcpp

Suggests bit64, tinytest

RoxygenNote 7.1.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2020-07-07 18:40:02 UTC

R topics documented:

RcppSimdJson-package *'Rcpp' Bindings for the 'simdjson' Header-Only Library for 'JSON' Parsing*

Description

The 'JSON' format is ubiquitous for data interchange, and the 'simdjson' library written by Daniel Lemire (and many contributors) provides a high-performance parser for these files which by relying on parallel 'SIMD' instruction manages to parse these files as faster than disk speed. See the <arXiv:1902.08318> paper for more details about 'simdjson'. This package is at present still a fairly thin and not fully complete wrapper that does not aim to replace the existing and excellent 'JSON' packages for R.

Package Content

Index of help topics:

```
RcppSimdJson-package    'Rcpp' Bindings for the 'simdjson' Header-Only
                        Library for 'JSON' Parsing
fparse                  Fast, Friendly, and Flexible JSON Parsing
parseExample            Simple JSON Parsing Example
validateJSON            Validate a JSON file, fast
```

Maintainer

Dirk Eddelbuettel <edd@debian.org>

Author(s)

Dirk Eddelbuettel, Brendan Knapp

fparse *Fast, Friendly, and Flexible JSON Parsing*

Description

Parse JSON strings and files to R objects.

Usage

```
fparse(
  json,
  query = "",
  empty_array = NULL,
  empty_object = NULL,
  single_null = NULL,
```

```
  error_ok = FALSE,
  on_error = NULL,
  max_simplify_lvl = c("data_frame", "matrix", "vector", "list"),
  type_policy = c("anything_goes", "numbers", "strict"),
  int64_policy = c("double", "string", "integer64")
)

fload(
  json,
  query = "",
  empty_array = NULL,
  empty_object = NULL,
  single_null = NULL,
  error_ok = FALSE,
  on_error = NULL,
  max_simplify_lvl = c("data_frame", "matrix", "vector", "list"),
  type_policy = c("anything_goes", "numbers", "strict"),
  int64_policy = c("double", "string", "integer64"),
  verbose = FALSE,
  temp_dir = tempdir(),
  keep_temp_files = FALSE
)
```

Arguments

`json`	One or more `character`s of JSON or paths to files containing JSON.
`query`	String used as a JSON Pointer to identify a specific element within `json`. `character(1L)`, default: `""`
`empty_array`	Any R object to return for empty JSON arrays. default: `NULL`
`empty_object`	Any R object to return for empty JSON objects. default: `NULL`.
`single_null`	Any R object to return for single JSON nulls. default: `NULL`.
`error_ok`	Whether to allow parsing errors. default: `FALSE`.
`on_error`	If `error_ok` is `TRUE`, `on_error` is any R object to return when parsing errors occur. default: `NULL`.
`max_simplify_lvl`	Maximum simplification level. `character(1L)` or `integer(1L)`, default: `"data_frame"` • `"data_frame"` or `0L` • `"matrix"` or `1L` • `"vector"` or `2L` • `"list"` or `3L` (no simplification)
`type_policy`	Level of type strictness. `character(1L)` or `integer(1L)`, default: `"anything_goes"`. • `"anything_goes"` or `0L`: non-recursive arrays always become atomic vectors • `"numbers"` or `1L`: non-recursive arrays containing only numbers always become atomic vectors

- `"strict"` or `2L`: non-recursive arrays containing mixed types never become atomic vectors

`int64_policy` How to return big integers to R. `character(1L)` or `integer(1L)`, default: `"double"`.

- `"double"` or `0L`: big integers become `double`s
- `"string"` or `1L`: big integers become `character`s
- `"integer64"` or `2L`: big integers `bit64::integer64`s

`verbose` Whether to display status messages. `TRUE` or `FALSE`, default: `FALSE`

`temp_dir` Directory path to use for any temporary files. `character(1L)`, default: `tempdir()`

`keep_temp_files` Whether to remove any temporary files created by `fload()` from `temp_dir`. `TRUE` or `FALSE`, default: `TRUE`

Details

- Instead of using `lapply()` to parse multiple values, just use `fparse()` and `fload()` directly.
  - They are vectorized in order to leverage the underlying `simdjson::dom::parser`'s ability to reuse its internal buffers between parses.
  - Since the overwhelming majority of JSON parsed will not result in scalars, a `list()` is always returned if `json` contains more than one value.
  - If `json` contains multiple values and has `names()`, the returned object will have the same names.
  - If `json` contains multiple values and is unnamed, `fload()` names each returned element using the file's `basename()`.

Author(s)

Brendan Knapp

Examples

```
# simple parsing ============================================================
json_string <- '{"a":[[1,null,3.0],["a","b",true],[10000000000,2,3]]}'
fparse(json_string)

# controlling type-strictness ===============================================
fparse(json_string, type_policy = "numbers")
fparse(json_string, type_policy = "strict")
fparse(json_string, type_policy = "numbers", int64_policy = "string")

if (requireNamespace("bit64", quietly = TRUE)) {
    fparse(json_string, type_policy = "numbers", int64_policy = "integer64")
}

# vectorized parsing ========================================================
json_strings <- c(
    json1 = '[{"b":true,
               "c":null},
              {"b":[[1,2,3],
```

```
                    [4,5,6]],
                "c":"Q"}]',
    json2 = '[{"b":[[7, 8, 9],
                    [10,11,12]],
              "c":"Q"},
             {"b":[[13,14,15],
                    [16,17,18]],
              "c":null}]'
)
fparse(json_strings)

# controlling simplification ================================================
fparse(json_strings, max_simplify_lvl = "matrix")
fparse(json_strings, max_simplify_lvl = "vector")
fparse(json_strings, max_simplify_lvl = "list")

# customizing what `[]`, `{}`, and single `null`s return ====================
empties <- "[[],{},null]"
fparse(empties)
fparse(empties,
       empty_array = logical(),
       empty_object = `names<-`(list(), character()),
       single_null = NA_real_)

# handling invalid JSON and parsing errors ==================================
fparse("junk JSON", error_ok = TRUE)
fparse("junk JSON", error_ok = TRUE, on_error = "can't parse invalid JSON")
fparse(
    c(junk_JSON_1 = "junk JSON 1",
      valid_JSON_1 = '"this is valid JSON"',
      junk_JSON_2 = "junk JSON 2",
      valid_JSON_2 = '"this is also valid JSON"'),
    error_ok = TRUE,
    on_error = NA
)

# querying JSON w/ a JSON Pointer ===========================================
json_to_query <- c(
    json1 = '[
    "a",
    {
        "b": {
            "c": [
                [
                    1,
                    2,
                    3
                ],
                [
                    4,
                    5,
                    6
                ]
```

```
            ]
        }
    }
]',
    json2 = '[
    "a",
    {
        "b": {
            "c": [
                [
                    7,
                    8,
                    9
                ],
                [
                    10,
                    11,
                    12
                ]
            ]
        }
    }
]')
fparse(json_to_query, query = "1")
fparse(json_to_query, query = "1/b")
fparse(json_to_query, query = "1/b/c")
fparse(json_to_query, query = "1/b/c/1")
fparse(json_to_query, query = "1/b/c/1/0")

# load JSON files ============================================================
single_file <- system.file("jsonexamples/small/demo.json", package = "RcppSimdJson")
fload(single_file)

multiple_files <- c(
  single_file,
  system.file("jsonexamples/small/smalldemo.json", package = "RcppSimdJson")
)
fload(multiple_files)

# load remote JSON ===========================================================
## Not run:

a_url <- "https://api.github.com/users/lemire"
fload(a_url)

multiple_urls <- c(
  a_url,
  "https://api.github.com/users/eddelbuettel",
  "https://api.github.com/users/knapply",
  "https://api.github.com/users/dcooley"
)
fload(multiple_urls, query = "name", verbose = TRUE)
```

```
## End(Not run)
```

`parseExample` *Simple JSON Parsing Example*

Description

This example is adapted from a blogpost announcing an earlier 'simdjson' release. It is of interest mostly for the elegance and conciseness of its C++ code rather than for any functionality exported to R.

Usage

```
parseExample()
```

Details

The function takes no argument and returns nothing.

Examples

```
parseExample()
```

`validateJSON` *Validate a JSON file, fast*

Description

By relying on simd-parallel 'simdjson' header-only library JSON files can be parsed very quickly.

Usage

```
validateJSON(jsonfile)
```

Arguments

`jsonfile`	A character variable with a path and filename

Value

A boolean value indicating whether the JSON content was parsed successfully

Examples

```
if (!RcppSimdJson:::.unsupportedArchitecture()) {
    jsonfile <- system.file("jsonexamples", "twitter.json", package="RcppSimdJson")
    validateJSON(jsonfile)
}
```

Index