Package 'RcppSimdJson'

October 7, 2020

Type Package

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

Version 0.1.2

Date 2020-10-07

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 2020-10-07 15:10:03 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 = NULL,
  empty_array = NULL,
  empty_object = NULL,
  single_null = NULL,
```

```
  parse_error_ok = FALSE,
  on_parse_error = NULL,
  query_error_ok = FALSE,
  on_query_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 = NULL,
  empty_array = NULL,
  empty_object = NULL,
  single_null = NULL,
  parse_error_ok = FALSE,
  on_parse_error = NULL,
  query_error_ok = FALSE,
  on_query_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,
  compressed_download = FALSE
)
```

Arguments

`json`	JSON strings, file paths, or raw vectors. • `fparse()` – `character`: One or more JSON strings. – `raw`: `json` is interpreted as the bytes of a single JSON string. – `list` Every element must be of type `"raw"` and each is individually interpreted as the bytes of a single JSON string. • `fload()` – `character`: One or more paths to files (local or remote) containing JSON.
`query`	If not `NULL`, JSON Pointer(s) used to identify and extract specific elements within `json`. See Details and Examples. `NULL`, `character()`, or `list()` of `character()`. default: `NULL`
`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`.
`parse_error_ok`	Whether to allow parsing errors. default: `FALSE`.

on_parse_error If parse_error_ok is TRUE, on_parse_error is any R object to return when query errors occur. default: NULL.

query_error_ok Whether to allow parsing errors. default: FALSE.

on_query_error If query_error_ok is TRUE, on_query_error is any R object to return when query 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 doubles
- "string" or 1L: big integers become characters
- "integer64" or 2L: big integers become bit64::integer64s

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

compressed_download
Whether to request server-side compression on the downloaded document, default: FALSE

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().
- query's goal is to minimize te amount of data that must be materialized as R objects (the main performance bottleneck) as well as facilitate any post-parse processing.

- To maximize flexibility, there are two approaches to consider when designing `query` arguments.
  - `character` vectors are interpreted as containing queries that meant to be applied to all elements of `json=`.
    - If `json=` contains 3 strings and `query=` contains 3 strings, the returned object will be a list of 3 elements (1 for each element of `json=`), which themselves each contain 3 lists (1 for each element of `query=`).
  - `list`s of `character` vectors are interpreted as containing queries meant to be applied to `json` in a zip-like fashion.

Author(s)

Brendan Knapp

Examples

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

raw_json <- as.raw(
    c(0x22, 0x72, 0x61, 0x77, 0x20, 0x62, 0x79, 0x74, 0x65, 0x73, 0x20, 0x63,
      0x61, 0x6e, 0x20, 0x62, 0x65, 0x63, 0x6f, 0x6d, 0x65, 0x20, 0x4a, 0x53,
      0x4f, 0x4e, 0x20, 0x74, 0x6f, 0x6f, 0x21, 0x22)
)
fparse(raw_json)

# 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)
```

```
fparse(
    list(
        raw_json1 = as.raw(c(0x74, 0x72, 0x75, 0x65)),
        raw_json2 = as.raw(c(0x66, 0x61, 0x6c, 0x73, 0x65))
    )
)

# 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", parse_error_ok = TRUE)
fparse("junk JSON", parse_error_ok = TRUE,
       on_parse_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"'),
    parse_error_ok = TRUE,
    on_parse_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]],
           "d": [1,2,3,4]
        }
```

```
    }
]')
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")

# handling invalid queries ================================================
fparse(json_to_query, query = "/1/b/d",
       query_error_ok = TRUE,
       on_query_error = "d isn't a key here!")

# multiple queries applied to EVERY element ===============================
fparse(json_to_query, query = c(query1 = "/1/b/c/1/0",
                                query2 = "/1/b/c/1/1",
                                query3 = "/1/b/c/1/2"))

# multiple queries applied to EACH element ================================
fparse(json_to_query,
       query = list(queries_for_json1 = c(c1 = "/1/b/c/1/0",
                                          c2 = "/1/b/c/1/1"),
                    queries_for_json2 = c(d1 = "/1/b/d/1",
                                          d2 = "/1/b/d/2")))

# 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)


## Not run:

# load remote JSON ========================================================
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)

# download compressed (faster) JSON =======================================
fload(multiple_urls, query = "name", verbose = TRUE,
      compressed_download = 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