

R documentation

of 'RJSONIO/man/JSON_T_NULL.Rd' etc.

December 7, 2011

Bob

Symbolic constants identifying the type of a JSON value.

Description

These constants are used by handler functions that are called when a JSON value is encountered by the JSON parser. These identify the type of the JSON value. The values will already have been converted, but the start and end array and object events won't have a type.

Format

A collection of integer values.

Source

JSON_parser.h code from <http://www.json.org>.

References

<http://www.json.org>.

asJSVars

Serialize R objects as Javascript/ActionScript variables

Description

This function takes R objects and serializes them as Javascript/ActionScript values. It uses the specified names in the R call as Javascript variable names. One can also specify qualifiers ('public', 'protected', 'private') and also types. These are optional, but useful, in ActionScript.

Usage

```
asJSVars(..., .vars = list(...), qualifier = character(), types = character())
```

Arguments

...	name = value pairs where the value is an R object that is converted to JSON format and name is the name of the corresponding Javascript variable.
.vars	this is an alternative to ... as a way to specify a collection of name = value pairs that is already in a list.
qualifier	a character vector (recycled as necessary) which is used as qualifiers for the individual ActionScript variables. The values should be public, protected or private.
types	either a logical value or a character vector (which is recycled if necessary). If this is TRUE, then we compute the Javascript type for each of the R objects (using the non-exported function jsType)

Value

A character vector of length 1 giving the variable declarations and initializations.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

See Also

[toJSON](#)

Examples

```
cat(asJSVars( a = 1:10, myMatrix = matrix(1:15, 3, 5)))
cat(asJSVars( a = 1:10, myMatrix = matrix(1:15, 3, 5), types = TRUE))
cat(asJSVars( a = 1:10, myMatrix = matrix(1:15, 3, 5),
             qualifier = "protected", types = TRUE))
```

basicJSONHandler

Create handler for processing JSON elements from a parser

Description

This function creates a handler object that is used to consume tokens/elements from a JSON parser and combine them into R objects.

This is slow relative to using C code because this is done in R and also we don't know the length of each object until we have consumed all its elements.

Usage

```
basicJSONHandler(default.size = 100, simplify = FALSE)
```

Arguments

default.size	the best guess as to the sizes of the different elements. This is used for preallocating space for elements
simplify	a logical value indicating whether to simplify arrays from lists to vectors if the elements are of compatible types.

Value

update	a function called with a JSON element and used to process that element and add it to the relevant R object
value	a function to retrieve the result after processing the JSON

Author(s)

Duncan Temple Lang

See Also

[fromJSON](#) and the handler argument.

Examples

```
h = basicJSONHandler()
x = fromJSON("[1, 2, 3]", h)
x
h$value()
```

fromJSON

Convert JSON content to R objects

Description

This function and its methods read content in JSON format and de-serializes it into R objects. JSON content is made up of logicals, integers, real numbers, strings, arrays of these and associative arrays/hash tables using key: value pairs. These map very naturally to R data types (logical, integer, numeric, character, and named lists).

Usage

```
fromJSON(content, handler = NULL,
         default.size = 100, depth = 150L, allowComments = TRUE,
         asText = isContent(content), data = NULL,
         maxChar = c(0L, nchar(content)), simplify = Strict,
         nullValue = NULL, simplifyWithNames = TRUE,
         encoding = NA_character_, ...)
```

Arguments

content	the JSON content. This can be the name of a file or the content itself as a character string. We will add support for connections in the near future.
handler	an R object that is responsible for processing each individual token/element within the JSON content. By default, this is NULL and we use the fast libjson parsing approach. Unless you want to customize the processing of the nodes in the tree, use NULL. This can be an R function, a list of functions with class "JSONParserHandler" having update and value elements, or the address of a native (C) routine. In the case of the latter, the data parameter can be used to specify an object that is passed to the C routine each time it is called. This will commonly be an externalptr object.

<code>default.size</code>	a number giving the default buffer size to use for arrays and objects in an effort to avoid reallocating each time we add a new element.
<code>depth</code>	the maximum number of nested JSON levels, i.e. arrays and objects within arrays and objects.
<code>allowComments</code>	a logical value indicating whether to allow C-style comments within the JSON content or to raise an error if they are encountered.
<code>asText</code>	a logical value indicating whether the value of the content argument should be treated as the JSON content, i.e. read directly rather than considered the name of a file.
<code>data</code>	a value that is only used when the value of <code>handler</code> is a native (C) routine. In this case, the value is passed in each call to that C routine by the JSON tokenizer.
<code>maxChar</code>	an integer vector of length 2 giving the start and end offsets in the character string to be processed. This allows the caller to specify a subset of the string to process without explicitly having to make a copy of the substring.
<code>simplify</code>	either a logical value or a number, e.g. the value of the variable <code>Strict</code> (the default). This controls whether we attempt to collapse collections/arrays of homogeneous scalar elements to R vectors. If this is <code>FALSE</code> , no effort to combine scalars is made and they remain as separate list elements. If this is <code>TRUE</code> , then logicals, numbers and strings are collapsed to their common types in the same manner as <code>c</code> . The value <code>Strict</code> does attempt to collapse collections of scalars but only if they are all of the same type, i.e. all strings, all numbers or all logicals. If we want to collapse numbers, but not logicals or characters, we can use <code>StrictNumeric</code> . Similarly, to collapse logicals but not numeric or character collections, we use <code>StrictLogical</code> . And, to collapse only character collections, we use <code>StrictCharacter</code> . If we want to collapse two types but not a third, we add the two values, e.g. <code>StrictLogical + StrictNumeric</code> , or pass them as a vector <code>c(StrictLogical, StrictNumeric)</code> . <code>Strict</code> is merely the combination of all 3 of the individual strict variables. Currently this is only implemented when the caller does not provide a handler and in the C code.
<code>nullValue</code>	an R value that is used when we encounter a JSON null value in the JSON content. This can be used to map null to something more R-like such as <code>NA</code> . This can be an arbitrary R object.
<code>simplifyWithNames</code>	a logical value that controls whether we attempt to collapse collections if the elements have names in the JSON content, i.e. a dictionary/associative array. If this is <code>TRUE</code> , then we consider collapsing according to the value of <code>simplify</code> . If this is <code>FALSE</code> , if the collection has names, we do not attempt to simplify.
<code>encoding</code>	the encoding for the content. This is used to ensure the encoding of any resulting strings/character vectors have this encoding. The default for this value is to use the same encoding as the input content.
<code>...</code>	additional parameters for methods.

Value

An R object created by mapping the JSON content to its R equivalent.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu

References

<http://www.json.org>

See Also

[toJSON](#) the non-exported collector function {RJSONIO:::basicJSONHandler}.

Examples

```

fromJSON(I(toJSON(1:10)))

fromJSON(I(toJSON(1:10 + .5)))

fromJSON(I(toJSON(c(TRUE, FALSE, FALSE, TRUE))))

x = fromJSON('{"ok":true,"id":"x123","rev":"1-1794908527"}')

# Reading from a connection. It is a text connection so we could
# just read the text directly, but this could be a dynamic connection.
m = matrix(1:27, 9, 3)
txt = toJSON(m)
con = textConnection(txt)
identical(m, fromJSON(con))

# Use a connection and move the cursor ahead to skip over some lines.
f = system.file("sampleData", "obj1.json", package = "RJSONIO")
con = file(f, "r")
readLines(con, 1)
fromJSON(con)
close(con)

f = system.file("sampleData", "embedded.json", package = "RJSONIO")
con = file(f, "r")
readLines(con, 1) # eat the first line
fromJSON(con, maxNumLines = 4)
close(con)

## Not run:
if(require(rjson)) {
  # We see an approximately a factor of 3.9 speed up when we use
  # this approach that mixes C-level tokenization and an R callback
  # function to gather the results into objects.

  f = system.file("sampleData", "usaPolygons.as", package = "RJSONIO")
  t1 = system.time(a <- RJSONIO:::fromJSON(f))
  t2 = system.time(b <- fromJSON(paste(readLines(f), collapse = "\n")))
}

## End(Not run)
# Use a C routine
fromJSON(I("[1, 2, 3, 4]"),
  getNativeSymbolInfo("R_json_testNativeCallback", "RJSONIO"))

```

```

# Use a C routine that populates an R integer vector with the
# elements read from the JSON array. Note that we must ensure
# that the array is big enough.
fromJSON(I("[1, 2, 3, 4]"),
         getNativeSymbolInfo("R_json_IntegerArrayCallback", PACKAGE = "RJSONIO"),
         data = rep(1L, 5))

x = fromJSON(I("[1.1, 2.2, 3.3, 4.4]"),
            getNativeSymbolInfo("R_json_RealArrayCallback", PACKAGE = "RJSONIO"),
            data = rep(1, 5))
length(x) = 4

# This illustrates a "specialized" handler which knows what it is
# expecting and pre-allocates the answer
# This then populates the answer with the values.
# The speed improvement is 1.8 versus "infinity"!

x = rnorm(1000000)
str = toJSON(x, digits = 6)

fromJSON(I(str),
         getNativeSymbolInfo("R_json_RealArrayCallback", PACKAGE = "RJSONIO"),
         data = numeric(length(x)))

# This is another example of very fast reading of specific JSON.
x = matrix(rnorm(1000000), 1000, 1000)
str = toJSON(x, digits = 6)

v = fromJSON(I(str),
            getNativeSymbolInfo("R_json_RealArrayCallback", PACKAGE = "RJSONIO"),
            data = matrix(0, 1000, 1000))

# nulls and NAs
fromJSON("{ 'abc': 1, 'def': 23, 'xyz': null, 'ooo': 4}", nullValue = NA)
fromJSON("{ 'abc': 1, 'def': 23, 'xyz': null, 'ooo': 4}", nullValue = NULL) # default

fromJSON("[1, 2, 3, null, 4]", nullValue = NA)
fromJSON("[1, 2, 3, null, 4]", nullValue = NULL)

# we can supply a complex object for null if we ever should need to.
fromJSON('[ 1, 2, null]', nullValue = list(a = 1, b = 1:10))[[3]]

# Using StrictNumeric, etc.
x = list(sub1 = list(a = 1:10, b = 100, c = 1000),
        sub2 = list(anim1 = "ape", anim2 = "bear", anim3 = "cat"),
        sub3 = rep(c(TRUE, FALSE), 3))
js = toJSON(x)

fromJSON(js)
# leave character strings uncollapsed
fromJSON(js, simplify = StrictNumeric + StrictLogical)
fromJSON(js, simplify = c(StrictNumeric, StrictLogical))

```

```
fromJSON(js, simplifyWithNames = FALSE)
fromJSON(js, simplifyWithNames = TRUE)
```

toJSON

Convert an R object to a string in Javascript Object Notation

Description

This function and its methods convert an R object into a string that represents the object in Javascript Object Notation (JSON).

The different methods try to map R's vectors to JSON arrays and associative arrays. There is ambiguity here as an R vector of length 1 can be a JSON scalar or an array with one element. When there are names on the R vector, the decision is clearer. We have introduced the `emptyNamedList` variable to identify an empty list that has an empty names character vector and so maps to an associative array in JSON, albeit an empty one.

Objects of class `AsIs` in R, i.e. that are enclosed in a call to `I()` are treated as containers even if they are of length 1. This allows callers to indicate the desired representation of an R "scalar" as an array of length 1 in JSON

Usage

```
toJSON(x, container = .level == 1L || length(x) > 1 || length(names(x)) > 0,
       collapse = "\n", ..., .level = 1L,
       .withNames = length(x) > 0 && length(names(x)) > 0, .na = "null",
       .escapeEscapes = TRUE )
```

Arguments

<code>x</code>	the R object to be converted to JSON format
<code>...</code>	additional arguments controlling the formatting of the JSON.
<code>container</code>	a logical value indicating whether to treat the object as a vector/container or a scalar and so represent it as an array or primitive in JavaScript.
<code>collapse</code>	a string that is used as the separator when combining the individual lines of the generated JSON content
<code>.level</code>	an integer value. This is not a parameter the caller is supposed to supply. It is a value that is passed in recursive calls to identify the top-level and sub-level serialization to JSON and so help to identify when a scalar needs to be in a container and when it is legitimate to output a scalar value directly.
<code>.withNames</code>	a logical value. If we are dealing with a named vector/list, we typically generate a JSON associative array/dictionary. If there are no names, we create a simple array. This argument allows us to explicitly control whether we use a dictionary or to ignore the names and use an array.
<code>.na</code>	a value to use when we encounter an NA value in the R objects. This allows the caller to convert these to whatever makes sense to them. For example, we might specify this as "null" and then the NA values will appear as null in the JSON output. One can also specify an unusual numeric value, e.g. -9999999 to indicate a missing value!

`.escapeEscapes` a logical value that controls how new line and tab characters are serialized. If this is TRUE, we preserve them symbolically by escaping the `\`. Otherwise, we replace them with their literal value.

Value

A string containing the JSON content.

Author(s)

Duncan Temple Lang <duncan@wald.ucdavis.edu>

References

<http://www.json.org>

See Also

[fromJSON](#)

Examples

```
toJSON(1:10)
toJSON(rnorm(3))
toJSON(rnorm(3), digits = 4)

toJSON(c("Duncan", "Temple Lang"))

toJSON(c(FALSE, FALSE, TRUE))

# List of elements
toJSON(list(1L, c("a", "b"), c(FALSE, FALSE, TRUE), rnorm(3)))
# with digits controlling formatting of sub-elements
toJSON(list(1L, c("a", "b"), c(FALSE, FALSE, TRUE), rnorm(3)),
        digits = 10)

# nested lists
toJSON(list(1L, c("a", "b"), list(c(FALSE, FALSE, TRUE), rnorm(3))))

# with names
toJSON(list(a = 1L, c("a", "b"), c(FALSE, FALSE, TRUE), rnorm(3)))

setClass("TEMP", representation(a = "integer", xyz = "logical"))
setClass("TEMP1", representation(one = "integer", two = "TEMP"))

new("TEMP1", one = 1:10, two = new("TEMP", a = 4L, xyz = c(TRUE, FALSE)))

toJSON(list())
toJSON(emptyNamedList)
toJSON(I(list("hi")))
toJSON(I("hi"))

x = list(list(),
        emptyNamedList,
```

```

      I(list("hi")),
      "hi",
      I("hi"))
toJSON(x)

# examples of specifying .withNames
toJSON(structure(1:3, names = letters[1:3]))
toJSON(structure(1:3, names = letters[1:3]), .withNames = FALSE)

# Controlling NAs and mapping them to whatever we want.
toJSON(c(1L, 2L, NA), .na = "null")
toJSON(c(1L, 2L, NA), .na = -9999)

toJSON(c(1, 2, pi, NA), .na = "null")

toJSON(c(TRUE, FALSE, NA), .na = "null")

toJSON(c("A", "BCD", NA), .na = "null")

toJSON( factor(c("A", "B", "A", NA, "A")), .na = "null" )

toJSON(list(TRUE, list(1, NA), NA), .na = "null")

setClass("Foo", representation(a = "integer", b = "character"))
obj = new("Foo", a = c(1L, 2L, NA, 4L), b = c("abc", NA, "def"))
toJSON(obj)
toJSON(obj, .na = "null")

# hexmode example with .na ?

toJSON(matrix(c(1, 2, NA, 4), 2, 2), .na = "null")
toJSON(matrix(c(1, 2, NA, 4), 2, 2), .na = -9999999)

x = '"foo\tbar\n\tagain"'
cat(toJSON(x))
cat(toJSON(list(x)))

# if we want to expand the new lines and tab characters
cat(toJSON(x), .escapeEscapes = FALSE)

```

Index

*Topic **IO**

- asJSVars, 1
- basicJSONHandler, 2
- fromJSON, 3
- toJSON, 7

*Topic **datasets**

- Bob, 1

*Topic **programming**

- asJSVars, 1
- basicJSONHandler, 2
- fromJSON, 3
- toJSON, 7

asJSVars, 1

basicJSONHandler, 2

Bob, 1

emptyNamedList (toJSON), 7

fromJSON, 3, 3, 8

fromJSON, AsIs, ANY-method (fromJSON), 3

fromJSON, AsIs, function-method
(fromJSON), 3

fromJSON, AsIs, JSONParserHandler-method
(fromJSON), 3

fromJSON, AsIs, NativeSymbolInfo-method
(fromJSON), 3

fromJSON, AsIs, NULL-method (fromJSON), 3

fromJSON, character, ANY-method
(fromJSON), 3

fromJSON, connection, ANY-method
(fromJSON), 3

JSON_T_ARRAY_BEGIN (Bob), 1

JSON_T_ARRAY_END (Bob), 1

JSON_T_FALSE (Bob), 1

JSON_T_FLOAT (Bob), 1

JSON_T_INTEGER (Bob), 1

JSON_T_KEY (Bob), 1

JSON_T_MAX (Bob), 1

JSON_T_NONE (Bob), 1

JSON_T_NULL (Bob), 1

JSON_T_OBJECT_BEGIN (Bob), 1

JSON_T_OBJECT_END (Bob), 1

JSON_T_STRING (Bob), 1

JSON_T_TRUE (Bob), 1

Strict (fromJSON), 3

StrictCharacter (fromJSON), 3

StrictLogical (fromJSON), 3

StrictNumeric (fromJSON), 3

toJSON, 2, 5, 7

toJSON, ANY-method (toJSON), 7

toJSON, AsIs-method (toJSON), 7

toJSON, character-method (toJSON), 7

toJSON, factor-method (toJSON), 7

toJSON, hexmode-method (toJSON), 7

toJSON, integer, missing-method (toJSON),
7

toJSON, integer-method (toJSON), 7

toJSON, list-method (toJSON), 7

toJSON, logical-method (toJSON), 7

toJSON, matrix-method (toJSON), 7

toJSON, name-method (toJSON), 7

toJSON, NULL-method (toJSON), 7

toJSON, numeric-method (toJSON), 7