JSON as a Native Tcl Value

Cyan Ogilvie
Ruby Lane, Inc.
cyan@rubylane.com

October 24, 2016

Abstract

JSON, being a string representation of structured data, finds a natural home in Tcl since this is exactly how Tcl sees all values (EIAS). Whereas most languages (even Javascript!) require that JSON first be converted to a native representation, and later serialized back to a string, Tcl provides the means (through the Tcl_Obj mechanism) to efficiently manipulate the JSON directly. rl_json [1] does this following the pattern of the `dict` command, and uses templates to provide the missing type information when producing new JSON values. I discuss the lessons learned while optimizing rl_json to meet the performance required, and how having self-describing structured data in Tcl is valuable beyond just data exchange.

1 Types Are a Problem

JSON has emerged as the de facto standard for data interchange between systems on the Internet, replacing XML in this role largely due to its lighter syntax and simpler structure, making it easy for most languages to directly translate it to and from native data structures. Tcl was excluded from this because it lacks types (or rather, its types are not intrinsic to values but are a transient property of values when they are accessed). Of course it is possible to convert the JSON object to a dictionary, and many existing Tcl packages do this, but it is a lossy transformation. Because JSON is typed, some systems use that channel to convey information which is lost when converting to a dictionary. Many Tcl JSON packages treat at least two of the following JSON documents as equivalent:

```
{
    "foo": "is this a dict?",
    "bar": null
```

[1] https://github.com/RubyLane/rl_json

```
}

{
    "foo": ["is", "this", "a", "dict?"],
    "bar": ""
}

{
    "foo": {
        "is": "this",
        "a":  "dict?"
    },
    "bar": "null"
}
```

The null representation issue is well trodden ground, and consensus seems to be that there is no reasonable way to represent this in Tcl without fundamentally changing the language. The string / list / dictionary ambiguity is more subtle and is inescapably rooted in the fact that a valid Tcl dictionary is all three. Indeed most English sentences are valid Tcl lists and about half are valid dictionaries. For many applications in which Tcl is a consumer of JSON data this issue isn't a practical concern, since the script implicitly provides the interpretation by how it accesses the values, just as it does for all other values in Tcl. When the type information itself encodes meaning this strategy fails, but those situations are fortunately rare in practice.

When it comes to producing JSON from Tcl values though the problem is unavoidable, and there is no choice but to provide the type information explicitly in some way. This typically leads to code that obscures the structure of the JSON document produced, and requires the programmer reading the code to mentally execute it while maintaining a mental model of the document being produced. While most programmers will succeed at this it is likely that whatever mental process they were busy with will have been evicted from working memory, hugely impacting on their productivity. This is especially true for very large and complex JSON documents, such as Elasticsearch queries (our typical query is around 260 lines, 8.5 KB, and 13 levels of nesting).

2 Types Are Not Such a Problem If You Squint Just Right

rl_json resolves this difficulty by avoiding the conversion from JSON to another (lossy) representation, providing instead a command that operates on the JSON values directly, analogous to how the `dict` command works with (possibly nested) dictionary values. In this way the type information is preserved and available when required, without obfuscating the script

in the majority of cases where it is not needed. To support this and maintain an acceptable level of performance, rl_json shimmers the JSON string to a custom Tcl_ObjType that stores the parsed document, representing each contained JSON value with a type enum and a Tcl_Obj holding the value. For a JSON object the Tcl_Obj is a dictionary, for an array it is a list, and so on. This allows efficient use of the contained values by Tcl, and elegantly handles sharing the JSON value (or fragments of it) using the normal Tcl_Obj reference counting. This approach makes it straightforward to implement efficient in-place updates to JSON values, which just invalidate the string rep and store a reference to the supplied Tcl_Obj (together with the JSON type).

For larger chunks of JSON structure that are filled in with values from the script, a template approach is used, which can be combined with incremental construction as in this admittedly contrived example:

```
set someval 0xFF

set res [json template {
    {
        "foo": "~S:someval",
        "bar": "~N:someval",
        "baz": ["~B:someval", "~S:otherval", "~L:~S:someval"]
    }
}]

# setting the "end+1" element of an array appends to it
json set res baz end+1 $res    ;#YoDawg
json set res new {"entry"}
# "end" and "end-1" are unambiguous because they index into an array
json unset res baz end baz end-1

puts [json pretty $res]
puts "baz: [json get $res baz 0] is a [json get $res baz 0 ?type]"
```

Which will produce:

```
{
    "foo": "0xFF",
    "bar": 255,
    "baz": [
        true,
        null,
        "~S:someval",
        {
```

```
            "foo": "0xFF",
            "bar": 255,
            "baz": [
                true,
                "~S:someval"
            ]
        }
    ],
    "new": "entry"
}
baz: 1 is a boolean
```

`json template` replaces strings in the supplied JSON document that match the regular expression `^~[SNBJTL]:.+$` with values sourced from the named variables, and types indicated by the replacement prefix (String, Number, Boolean, JSON, Template and Literal). Since templates are themselves just JSON, they can be manipulated in the same way that any other JSON values are.

3 Performance Considerations

Although obvious in hindsight, I was initially surprised that parsing the JSON strings to build the internal representation was largely bottlenecked by Tcl_Alloc – each JSON value and object key needs to have a Tcl_Obj created to store it, resulting in a lot of calls to Tcl_NewStringObj. To address this rl_json maintains a per-interp cache of recently used strings, using a hitrate and aging scheme to quickly learn patterns in the data, while remaining small and preferring frequently requested values. Besides being substantially faster to lookup and reuse a Tcl_Obj from this cache than to allocate a fresh instance, it also consumes less memory since each instance is just a pointer to the shared Tcl_Obj.

This sharing also promotes efficient reuse of the intreps of the Tcl_Objs in the cache – for instance if some string value in the JSON document contains an account status like “active”, “closed”, “opening”, etc, and that value is later fed to Tcl_GetIndexFromObj, then the Tcl_Obj in the cache will shimmer to record the index found. Similar JSON documents processed in the near future with the same account status value will receive references to that cached Tcl_Obj, so their Tcl_GetIndexFromObj lookups will be very fast.

When parsing a large number of similar JSON documents (such as the results returned by an Elasticsearch query) the difference that deduplication makes over just calling Tcl_NewStringObj for each value is about an 8% to 15% reduction in execution time.

I used the Linux performance analysis tools to get some insight into where the time was spent when parsing large amounts of JSON. These data were obtained using using perf record to

sample a script that ran 100 iterations parsing a 4MB JSON document.

3.1 Using Tcl_NewStringObj

This version just called Tcl_NewStringObj for each key and string value in the doc, taking 56.45 ms per iteration to parse the JSON document:

```
# Overhead  Command   Shared Object       Symbol
# ........  ........  ..................  ....................................
    17.90%  tclsh8.6  libtcl8.6.so        [.] TclpAlloc
    14.08%  tclsh8.6  libtcl8.6.so        [.] TclThreadAllocObj
    13.44%  tclsh8.6  libtcl8.6.so        [.] FreeDictInternalRep
     9.96%  tclsh8.6  librl_json0.9.4.so  [.] value_type
     6.90%  tclsh8.6  libtcl8.6.so        [.] Ptr2Block
     5.23%  tclsh8.6  libtcl8.6.so        [.] CreateHashEntry
     3.56%  tclsh8.6  libpthread-2.23.so  [.] pthread_getspecific
     2.37%  tclsh8.6  libtcl8.6.so        [.] TclFreeObjEntry
     2.33%  tclsh8.6  libtcl8.6.so        [.] Tcl_DeleteHashTable
     2.16%  tclsh8.6  libtcl8.6.so        [.] TclFreeObj
     2.08%  tclsh8.6  libtcl8.6.so        [.] TclpFree
     1.74%  tclsh8.6  librl_json0.9.4.so  [.] free_internal_rep
     1.64%  tclsh8.6  libtcl8.6.so        [.] TclHashObjKey
     1.61%  tclsh8.6  librl_json0.9.4.so  [.] set_from_any
     1.35%  tclsh8.6  libtcl8.6.so        [.] PutBlocks
     1.21%  tclsh8.6  librl_json0.9.4.so  [.] skip_whitespace
```

TclpAlloc and TclThreadAllocObj dominate the overhead in this configuration.
value_type is where the bulk of the actual JSON parsing is done.

3.2 Using String Deduplication

This version enables the deduplication of string values, using a Tcl_HashTable keyed by the string itself and mapping to a Tcl_Obj containing the string. Each time the cache is hit that entry's hit count is incremented by one. When the number of entries exceeds 154 (a number chosen by tuning for our data patterns) the cache is aged: each entries' hitcount is divided by two and those that reach 0 are evicted from the cache. In this way the size of the cache remains bounded. Strings longer than 16 bytes are excluded from caching scheme. This provides a very cheap cache that tends to keep the most common values in the cache while remaining small and requiring very little housekeeping.

Using the same test setup as above, this version took 51.91 ms per iteration:

```
# Overhead  Command   Shared Object       Symbol
# ........  ........  ..................  ..............................
    19.25%  tclsh8.6  libtcl8.6.so        [.] FreeDictInternalRep
    18.48%  tclsh8.6  libtcl8.6.so        [.] TclpAlloc
     9.96%  tclsh8.6  librl_json0.9.4.so  [.] value_type
     8.62%  tclsh8.6  libtcl8.6.so        [.] TclThreadAllocObj
     8.42%  tclsh8.6  libtcl8.6.so        [.] CreateHashEntry
```

```
      5.30%  tclsh8.6  libtcl8.6.so          [.] Ptr2Block
      2.26%  tclsh8.6  libtcl8.6.so          [.] Tcl_DeleteHashTable
      2.23%  tclsh8.6  libpthread-2.23.so    [.] pthread_getspecific
      1.92%  tclsh8.6  librl_json0.9.4.so    [.] new_stringobj_dedup
      1.90%  tclsh8.6  librl_json0.9.4.so    [.] set_from_any
      1.78%  tclsh8.6  librl_json0.9.4.so    [.] free_internal_rep
      1.66%  tclsh8.6  libtcl8.6.so          [.] TclHashObjKey
      1.37%  tclsh8.6  libtcl8.6.so          [.] HashStringKey
      1.32%  tclsh8.6  libtcl8.6.so          [.] TclFreeObj
      1.26%  tclsh8.6  libtcl8.6.so          [.] TclpFree
      1.24%  tclsh8.6  librl_json0.9.4.so    [.] skip_whitespace
      1.13%  tclsh8.6  libc-2.23.so          [.] __strcmp_sse2_unaligned
```

In this run TclpAlloc and TclThreadAllocObj's share of the time has dropped in proportion to the other hot functions.

4 When Both Sides Are Tclish

Dictionaries are an effective way to represent simple structured data such as a row from a SQL result, or where the structure consists exclusively of nested dictionaries, but they start to creak when the structure becomes much more complex. This is particularly true when the structure mixes nested lists and dictionary values, as might be returned by a proc in the data access layer that returns order information containing a list of items in the order and correspondence about the order. The string representation of such a structure is daunting to parse by eye, and the string / list / dictionary ambiguity means that it isn't possible to write a generic pretty-printer.

Since the structure of a JSON document is an intrinsic property of the value itself, a generic pretty-printer can be written that allows programmers to bring their Gestalt pattern recognition abilities to bear to quickly understand the document structure, and to pick out a particular item of interest. Since programming languages are adapters between human and machine intelligences, being able make effective use of the programmer's wetware is an important design goal for these tools.

Although the impedance mismatch of a typed data structure in a typeless language would seem to make it an odd choice for purely internal use cases there are compensations that weigh in its favour. One of these is that the pattern of indexing into nested dictionaries provided by `dict get` can be extended to work with arrays, since the type being indexed into by an element in the key path is known. Consider:

```
set json {
    {
        "foo": "bar",
        "baz": ["str", 123, 123.4, true, false, null, {"inner": "target"}]
    }
```

```
}

# These two approaches pick out the value "target" from the above

# Since the types are known at each step of the path, the syntax for
# indexing into mixed objects and arrays can be very terse:
json get $json baz end inner

# If that interpretation has to be supplied implicitly by the commands
# used to access each step, the syntax becomes cumbersome:
dict get [lindex [dict get [json::json2dict $json] baz] end] inner
```

Being able to directly modify arrays with `json set` also provides a nice way to implement queues, or reorders:

```
set items {
    {
        "source":   ["a", "b", "c", "d", "e", "f"],
        "shuffled": []
    }
}

while {[json get $items source ?length]} {
    set picked [expr {int( rand() * [json get $items source ?length] )}]

    # setting element "-1" of an array prepends an element to it.
    # "end+1" would work here too
    json set items shuffled -1 [json extract $items source $picked]

    # unsetting an element in an array removes it
    # (rather than setting it to null)
    json unset items source $picked
}
```

5 Future Work

`json template` currently produces a string result, but experience with the package has shown that the result is very often further manipulated by other json commands. It would therefore be more efficient to return a native JSON Tcl_Obj and defer the serialization until it is actually needed.

It would be interesting to apply the speculative deduplication used here for short strings to the core in general (probably by incorporating it into TclNewStringObj), since it is both faster and reduces memory footprint.

`json foreach` and `json lmap` should be NRE enabled.

Appendix A: Quick Reference

- `json get json_val ?key ... ?modifier??`
  Extract the value of a portion of the *json_val*, returns the closest native Tcl type (other than JSON) for the extracted portion.
- `json get_typed json_val ?key ... ?modifier??`
  Extract the value of a portion of the *json_val*, returns a two element list: the first being the value that would be returned by `json get` and the second being the JSON type of the extracted portion.
- `json extract json_val ?key ... ?modifier??`
  Extract the value of a portion of the *json_val*, returns the JSON fragment.
- `json exists json_val ?key ... ?modifier??`
  Tests whether the supplied key path and modifier resolve to something that exists in *json_val*.
- `json set json_variable_name ?key ...? value`
  Updates the JSON value stored in the variable *json_variable_name*, replacing the value referenced by key ... with the JSON value value.
- `json unset json_variable_name ?key ...?`
  Updates the JSON value stored in the variable *json_variable_name*, removing the value referenced by key ...
- `json normalize json_val`
  Return a normalized version of the input *json_val* – all optional whitespace trimmed.
- `json template json_val ?dictionary?`
  Return a JSON value by interpolating the values from dictionary into the template, or from variables in the current scope if dictionary is not supplied, in the manner described in Appendix A.2: Templates.
- `json new type value`
  Return a JSON fragment of type type and value value.
- `json foreach varlist1 json_val1 ?varlist2 json_val2 ...? script`
  Evaluate script in a loop in a similar way to the `foreach` command. In each iteration, the values stored in the iterator variables in varlist are the JSON fragments from

json_val. Supports iterating over JSON arrays and JSON objects. In the JSON object case, varlist must be a two element list, with the first specifying the variable to hold the key and the second the value. In the JSON array case, the rules are the same as the *foreach* command.

- `json lmap varlist1 json_val1 ?varlist2 json_val2 ...?  script`
  As for `json foreach`, except that it is collecting - the result from each evaluation of `script` is added to a list and returned as the result of the `json lmap` command. If the script results in a TCL_CONTINUE code, that iteration is skipped and no element is added to the result list. If it results in TCL_BREAK the iterations are stopped and the results accumulated so far are returned.

Appendix A.1: Paths

The commands `json get`, `json get_typed`, `json extract` and `json exists` accept a path specification that names some subset of the supplied *json_val*. The rules are similar to the equivalent concept in the `dict` command, except that the paths used by `json` allow indexing into JSON arrays by the integer key (or a string matching the regex `^end(-[0-9]+)?$`), and that the last element can be a modifier:

- `?type` – Returns the type of the named fragment.
- `?length` – When the path refers to an array, this returns the length of the array. When the path refers to a string, this returns the number of characters in the string. All other types throw an error.
- `?size` – Valid only for objects, returns the number of keys defined in the object.
- `?keys` – Valid only for objects, returns a list of the keys in the object.

A literal value that would match one of the above modifiers can be used as the last element in the path by doubling the ?:

```
json get {
    {
        "foo": {
            "?size": "quite big"
        }
    }
} foo ??size
```

Returns “quite big”.

Appendix A.2: Templates

The `json template` command generates JSON documents by interpolating values into a template from a supplied dictionary or variables in the current call frame. The templates are valid JSON documents containing string values which match the regex `^~[SNBJTL]:.+$`. The second character determines what the resulting type of the substituted value will be:

- S: A string.
- N: A number.
- B: A boolean.
- J: A JSON fragment.
- T: A JSON template (substitutions are performed on the inserted fragment).
- L: A literal – the resulting string is simply everything from the forth character onward (this allows literal strings to be included in the template that would otherwise be interpreted as the substitutions above).

None of the first three characters for a template may be escaped.

The value inserted is determined by the characters following the substitution type prefix. When interpolating values from a dictionary they name keys in the dictionary which hold the values to interpolate. When interpolating from variables in the current scope, they name scalar or array variables which hold the values to interpolate. In either case if the named key or variable doesn't exist, a JSON null is interpolated in its place.