Reducing serde_json allocations in REST API app

help

smarnach May 20, 2022, 3:52pm 1

I'm working on reimplementing a relatively simple Python REST API proxy server in Rust. The aim of the rewrite is to save compute resources – the workload is completely CPU bound at the moment. The proxy server translates between two different APIs. It takes JSON requests from clients, transforms the JSON structure, and passes the transformed request on to the backend. The reply from the backend is again transformed and returned to the client. So the main work this proxy is doing is deserializing, restructuring and serializing JSON.

The JSON is deeply nested and contains a lot of strings of vastly different lengths, but usually the same field in each struct will contain a string of similar length for each request. I believe that the proxy server will spend a significant amount of its time doing memory allocations (I haven't measured yet, but I'm prematurely thinking about solutions already anyway).

My main idea to reduce the number of heap allocations is approximately this:

- Create thread-local pools for the incoming JSON structs, so we can reuse them across requests.
- Use `deserialize_in_place()` to deserialize the incoming JSON (both from client and backend).
- Use string slices referencing the original JSON for the transformed JSON.

While `deserialize_in_place()` works great for `String` fields, and will indeed reuse the allocations, I don't think it will help much with `Vec<SomeSubStruct>` or even just `Option<String>`, which we have a lot of. One top-level structure for JSON replies from the server for example looks like this:

```
#[derive(Deserialize)]
pub struct DecisionResponse {
   pub decisions: HashMap<String, Option<Vec<Decision>>>,
}
```

The hash map will contain exactly one key most of the time, but the value for that key will sometimes be `null`, and sometimes a list of many `Decisions`, which themsleves are rather nested objects and contain a lot of `Strings` and `Option<String>s`. Whenever the `DecisionResponse` gets overwritten with a resonse having `null` as the value, the vector and all `Decisions` in it will be deallocated, which means if it gets overwritten again with a long list of `Decisions`, all of them and the strings in them will need to be allocated again.

One idea to solve this is to implement my own data structures that keep all allocations around, e.g.

```
struct KVec<T> {
   data: Vec<T>,
   length: usize,
}

impl<T> Deref for KVec<T> {
   type Target = [T];

   fn deref(&self) -> &Self::Target {
       &self.data[..self.length]
   }
}
```

Then I could implement `deserialize_in_place()` for this in a way to only reduce `length` if there are few items then last time, but without actually dropping the unneeded elements. I'd need similar data structures for `Option` and `HashMap`, and all of them would need custom `Serialize` implementations as well.

Overall, I believe the approach I outlined would work, but it's kind of cumbersome to implement. Is there any easier solution to reduce the number of allocations in this scenario? Or is there an easier way to make the `deserialize_in_place()` approach work?

1 Like

cliff May 21, 2022, 11:47am 2

What about taking a lazy approach to dynamically sized lists of values? Store a cursor into the Json that deserializes one object at a time. maybe that's just as cumbersome as your current approach.

FWIW, I implemented a zero copy BSON deserializer, which doesn't entirely work for JSON because of string escaping, but might give you some inspiration for other approaches to your problem.

1 Like

smarnach May 22, 2022, 9:02pm 3

Hi Cliff! Thanks a lot for your suggestions! It's not easy to serialize one object at a time, since some of the transformations require some global view of the data. I already thought about some zero-copy approach – for most strings, I don't really care about escape sequences, since the strings are passed on without modification. So as long as the serializer is aware that the string is already escaped, this would actually save work at both ends, in addition to making the allocation for the string unnecessary. I considered using serde_json_core, which almost works for my use case, except for a few issues. Maybe adapting that crate is the way to go, but it isn't significantly easier than my original plan.

Anyway, it looks like I didn't miss anything obvious – otherwise I guess someone would have pointed it out by now, so I'll just pick one of the two approaches.

1 Like

cliff May 23, 2022, 11:36am 4

oh yeah, this isn't really a general purpose solution, but make any closed-set string values as you can into enums. that will help significantly.

simonbuchan May 23, 2022, 12:09pm 5

A couple of notes:

- Unless you can skip some expensive init, no allocation caching you're doing is going to noticeably outperform the allocator, and is likely to make it worse. A simple case of this working is keeping a short-term "high-water mark" allocation, where you can skip continuously resizing up to the average size of the list/string, but the keyword there is short-term. Get it working, then start measuring.
- JSON strings can be escaped, but normally can be shared straight from the source. Use `Cow<'a, str>` to represent both these cases efficiently. Though note *usage* is slightly slower, as it's just `enum Cow<'a, T: ToOwned> { Borrowed(&'a T), Owned(T::Owned) }`, so code has to branch each time. For normal cases, this is an easy win.
- Coming from Python, you're going to get a *lot* faster just out of the box, for a lot of reasons. See if the "dumb" implementation is good enough first, before spending time melting your brain for dubious wins! On the plus side, you get to see it working earlier, then get to see it speed up if you can find and squash hot spots after.

simonbuchan May 23, 2022, 12:13pm 6

smarnach:

> for most strings, I don't really care about escape sequences, since the strings are passed on without modification

For this it sounds like what you want is `RawValue`, which just skips over the whole thing and doesn't bother parsing at all.

1 Like

smarnach May 23, 2022, 1:03pm 7

@simonbuchan Thanks a lot for your thoughts! I'm aware this is premature optimization, and so far I'm not spending a lot of time on it. We've got working code already, by the way, we just haven't load-tested it yet. I expect what we have to be at least ten times faster than the Python code already.

Regarding outperforming the allocator, we are not talking about replacing a single allocation with a look up in some kind of cache. We are rather talking about replacing a couple hundred calls to the allocator with a single lookup in the allocation cache, because the structure is so deeply nested and contains so many strings. I expect I'd be able to easily outperform the allocator in this scenario.

However, your `RawValue` suggestion makes all of that moot. Using that, I can probably get rid of 95% of the allocations with very little effort, and in addition speed up deserialization and serialization signifcantly as well (in addition to some other gains I won't detail here). If I end up implementing this optimization, I'll report back how much performance I could gain that way.

smarnach May 23, 2022, 1:13pm 8

Another reason for trying to avoid allocations is that the allocator is the main source for resource contention between threads in this codebase, so even if I can't outperform the allocator, avoiding allocations might still be a performance win.

simonbuchan May 23, 2022, 1:32pm 9

Interesting, I wouldn't expect that much contention unless there was a lot of memory moving threads (freed in a different thread to the allocation), but perhaps that's not worth optimizing for in common cases.

system Closed August 21, 2022, 1:33pm 10

This topic was automatically closed 90 days after the last reply. We invite you to open a new topic if you have further questions or comments.

Related topics

Topic	Replies	Views	Activity
Reuse memory in parsing ndjson with a known (at runtime) schema help	1	311	November 18, 2021
Cow + serde_json help	3	1730	May 28, 2022

Minimal Rocket app generates probably enormous amount of heap allocations help	5	526	August 31, 2023
Serde json Deserializer stream help	7	1669	March 2, 2021
Zero copy deserialize arbitrary json data with serde help	10	3277	October 4, 2023

- Home
- Categories
- Guidelines
- Terms of Service

Powered by Discourse, best viewed with JavaScript enabled