Project: Asynchrony

Task: Create a multi-threaded, persistent key/value store server and client with *asynchronous* networking over a custom protocol.

Goals:

- Understand the patterns used when writing Rust futures
- Understand error handling with futures
- Learn to debug the type system
- Perform asynchronous networking with the tokio runtime
- Use boxed futures to handle difficult type-system problems
- Use `impl Trait` to create anonymous `Future` types

Topics: asynchrony, futures, tokio, `impl Trait`.

Extensions: tokio-fs.

Introduction

Note: this project is only outlined, not written. If you are at this point in the course email brian@pingcap.com and let me know and I will finish writing it ASAP.

In this project you will create a simple key/value server and client that communicate over a custom protocol. The server will use asynchronous networking, built on the tokio runtime. The key/value engine that reads and writes to files will remain synchronous, scheduling work on an underlying thread pool, while presenting an asynchronous interface. Along the way you will experiment with multiple ways of defining and working with future types.

Because learning to program with Rust futures is especially challenging, and existing documentation on the subject is limited, the scope of this project is relatively modest, and it contains more direct explanation than past projects.

Be sure to read the background readings on this project. And if you get frustrated, then forgive yourself, take a break, and try again with a fresh mind. Writing asynchronous Rust is difficult for everybody.

Project spec

The cargo project, `kvs`, builds a command-line key-value store client called `kvs-client`, and a key-value store server called `kvs-server`, both of which in turn call into a library called `kvs`. The client speaks to the server over a custom protocol.

The interface to the CLI is the same as in the [previous project]. The engine implementation is largely the same, distributing synchronous file I/O over a thread pool.

The difference this time is that all the networking is performed asynchronously.

As part of the conversion to asynchrony, the `KvsClient` will present a futures-based API, and the `KvsEngine` trait will also present a futures-based API, even while it is implemented with blocking (synchronous) I/O via a thread pool.

Your `KvsServer` will be based on the tokio runtime, which handles the distribution of asynchronous work to multiple threads on its own (tokio itself contains a thread pool). This means that your architecture will actually have two layers of thread pools: the first handling with the networking, asynchronously, one thread per core; the second handling the file I/O, synchronously, with enough threads to keep the networking threads as busy as possible.

As a result if this architectural change, where your jobs will be spawned into your thread pool from multiple threads, your `ThreadPool` trait and its implementations will become shared types implementing `Clone + Send + 'sync`, as your `KvsEngine` is.

Because you will be experimenting with multiple definitions of the futures returned by these types, they aren't fully specified here, and instead will be specified as they are called for.

More specifically, you will work with function signatures like all the following:

- `Client::get(&mut self, key: String) -> Box<Future<Item = Option<String>, Error = Error>`
- `Client::get(&mut self, key: String) -> future::SomeExplicitCombinator<...>`
- `Client::get(&mut self, key: String) -> impl Future<Item = Option<String>, Error = Error>`
- `Client::get(&mut self, key: String) -> ClientGetFuture`

Project setup

Continuing from your previous project, delete your privous `tests` directory and copy this project's `tests` directory into its place. This project should contain a library named `kvs`, and two executables, `kvs-server` and `kvs-client`.

You need the following dev-dependencies in your `Cargo.toml`:

```toml
[dev-dependencies]
assert_cmd = "0.11"
criterion = "0.2.11"
crossbeam-utils = "0.6.5"
predicates = "1.0.0"
rand = "0.6.5"
tempfile = "3.0.7"
walkdir = "2.2.7"
panic-control = "0.1.4"
```

Unlike with previous projects, don't bother to fill in enough type definitions to make the test suite compile. Doing so would require jumping a number of steps ahead at once. The text will indicate when to working with the test suite.

Background: Thinking in futures, in Rust

- why futures? networking vs file/io, blocking vs non-blocking, sync vs async
- futures from a user persective (not a poll-centric implementation perspective)
- don't think too hard about executors and runtimes
- method chaining and how it transforms the future type
- debugging Rust types
- Result vs Future vs FutureResult
- error handling with futures
- concrete futures vs boxed futures vs anonymous futures
- note about futures 0.1 and futures 0.3 (we'll use futures 0.1)
- note about async / await

Part 1: Introducing tokio to the client

Ultimately we're going to convert both the client and server to futures, but since the client is so simple, that's where we'll start. And we're going to introduce the tokio runtime first, while using your existing synchronous `KvsClient`.

The

for the client we're going to introduce the async runtime while keeping the sync `KvsClient`, then convert the `KvsClient`. The `connect` method of `KvsClient`. note that as a library `KvsClient` can offer the most efficiency based on futures, but our kvs-client bin doesn't take advantage of it, so the bin is going to look a bit silly running a single future and exiting.

TODO @sticnarf - see if you can write test cases that are agnostic to the concrete future types, so they work with all the below strategies.

Part 2: Converting `KvsClient` to boxed futures

the path of least resistence for future types

Part 3: `KvsClient` with explicit future types

just to have the experience of seeing how untenable it is

Part 4: `KvsClient` with anonymous future types

the final solution

Part 5: Making `ThreadPool` sharable

Part 6: Converting `KvsEngine` to futures

for the server we're going to do the opposite of what we did in the client, and give `KvsEngine` an async interface. this will show that futures and the underlying runtime are independent, and just general provide a spectrum of experience.

Part 7: Driving `KvsEngine` with tokio

note that even though we have ourselves have written very little asynchronous code, that tokio itself is distributing asynchronous work across num_cpus threads. think about the tradeoffs of putting cpu-intensive work directly on the network threads or the file threads, e.g. where does the serialization go?

TODO

Nice coding, friend. Enjoy a nice break.

Extension 1: Converting to tokio-fs

not sure if this should be required or an extension

<!--

TODO:

- can we find an excuse to write a future by hand?
- background readings
  - something on associated types

via @sticnarf:

> As there is only the outline of project 5, I write the code mostly according to my own thoughts. Hope this will be a reference while you're writing the text. @brson

> I change the concurrent_get/set tests to use async. Students should change their SledKvsEngine and KvStore to adapt to the KvsEngine trait with new async APIs. The engines have a ThreadPool type parameter and the constructor has a concurrency argument (maybe we should remove it). Students need to follow the design so that the test will work.

> I don't test the client. Implementors can choose the API design of the client themselves (unless we work out a perfect design so we can just give instructions to students).

-->