CS 6422

Table of Contents

Introduction

The assignment consists of a implementation of your own request types using an open source Cassandra Java Driver. For this exercise we will use DataStax register a custom processor to use it. The DataStax Java driver is available from Maven central, and you can browse its sources in this GitHub repository.

Background: Request processors in the driver

The top-level Session interface defines a very generic execution method:

```
<RequestT extends Request, ResultT> ResultT
    execute( RequestT request, GenericType<ResultT>
    resultType);
```

`resultType` is a *type token* that tells the driver how to execute the request. In general, we use sub- interfaces to hide that behind methods with a more user-friendly signature. For example in CqlSession:

```
default ResultSet execute(Statement<?>
  statement) { return execute(statement,
  Statement.SYNC);
}
default ResultSet execute(String query) {
  return execute(SimpleStatement.newInstance(query));
}
```

At runtime, we use the tuple `<RequestT, ResultT>` to select a RequestProcessor that will execute the request. Processors are stored in a registry that gets passed to the session during initialization.

Assignment

To avoid the overhead of bootstrapping a project, installing and configuring Cassandra, we provide a sample Maven project with a few classes and an integration test. You should use it as a starting point for the assignment.

Overview

The assignment project contains two interfaces that define a producer-consumer API:

```
public interface Producer<T> {
  void register(Consumer<? super T>
  consumer); void produce(long n);
  void cancel();
}
public interface
  Consumer<T> { void
  consume(T item);
  void operationComplete();
  void operationAborted(Throwable error);
}
```

Read carefully the javadocs of these interfaces to understand the contracts behind each method. Also, pay attention to the fact that this API establishes a hybrid "push-pull" model between producer and consumer: the consumer notifies the producer when it is ready to accept more items, and how many items it is ready to accept; the producer must take that information into account so that it will never produce more items than requested by the consumer. This creates a flow regulation between producer and consumer that is sometimes referred to as *backpressure*.

The goal of this assignment is to support the above-mentioned producer-consumer API when querying Cassandra. In other words, we would like to be able to submit a CQL query and get back a `Producer<Row>` that would emit rows to a registered `Consumer`.

The assignment project has been built and tested with Java 8. Feel free to use a higher version of Java if you like, but please be aware that the integration tests start a Cassandra instance in the background, and most Cassandra versions do not work well with more recent Java versions. Also, if you have trouble running the integration tests, make sure you set your global environment variable `JAVA_HOME` to a valid Java 8 installation root.

Also, the project contains all the dependencies you need to complete the assignment, but feel free to add more dependencies as you see fit.

Step 1: Session and request processor

To achieve this goal, you will first need to complete the `ProducerConsumerSession` interface, and its companion builder, `ProducerConsumerSessionBuilder`.

Then you will be required to write a request processor that "knows" how to generate results of type

`Producer<Row>`; you can write it from scratch, but a better idea would be to leverage an existing request processor and "wrap" it with a delegate pattern.

Tip: there is a similar example in integration tests suite under the example/guava package: take inspiration from it and you won't need to start from scratch. We even have an integration test that covers all this: RequestProcessorIT. Study that code carefully: it shows you how to wire the processor registry in the session, and wrap everything nicely with a custom session interface and a builder.

Step 2: `Producer<Row>`

Your request processor needs to create instances of `Producer<Row>` for each request; implement that interface now. Please follow the implementation guidelines below:

1. Your implementation *must* be thread-safe.
2. Your implementation *must* abide by all the rules defined in the javadocs of the `Producer` interface.
3. Your implementation *should* avoid blocking calls and *should* make extensive use of synchronization primitives such as locks, volatile fields and atomic variables.
4. Your implementation should store temporary items in a queue, and drain it whenever the consumer requests more items (via `Producer.produce()`), or when more results are made available by the server.

Step 3: Unit tests

Once your row producer is ready, write unit tests for it. We usually write the unit tests *before* writing the system under test, but we want to see how you write and organize your unit tests.

Tip: The Maven project already contains a few test dependencies that you should use in your tests: JUnit 4, AssertJ and Mockito. Here is a typical unit test:

```
import static
org.assertj.core.api.Assertions.*; import
static org.mockito.ArgumentMatchers.*;
import static org.mockito.Mockito.*;

@Test
public void should_consume_all_rows() {
  // given
  Consumer<Row> mockConsumer =
  ... Producer<Row> producer =
  ...;
  // when
  producer.register(consume
  r); producer.produce(10);
  // then
  // verify consumer's behavior
  // assert the contents of each row
}
```

Step 4: Integration tests

And finally, there is an integration test present in the project: `ProducerConsumerIT`. Obviously, this test must pass without any modification.

Note: The driver integration tests use CCM (Cassandra Cluster Manager). You must install this software prior to running any integration test.

Bonus points

If you have extra time, you can try one of these tasks:

- Your row producer implementation has blocking calls or locks? Try writing a fully non-blocking, lock-free version of it. There are sophisticated patterns that allow to drain a queue in a non- blocking way. See for example RxJava's original queue-drain pattern, and a few variants thereof.
- Implement the extra methods commented out in the `Producer` interface.