Java's `CompletableFuture` API

Java's CompletableFuture implements the popular "Promise Pipeline" concept.

Promises use a monadic approach that allows us to define the computation that should be applied to input data, and to send the result to another processing stage, but allows the computation to be performed in another thread, often taken from a pool.

In particular, long running processes that support a "callback" style completion can be supported directly without the need for hard-to-maintain nested blocks.

Promise pipelines differ from normal monads in that they provide two data paths, one for success results, and another for exception or failure propagation, along with recovery mechanisms to get back onto the "happy path".

© 2020 Simon Roberts

Starting the pipeline

Unlike a Stream, a `CompletableFuture` processes a single data item, it's also a single use structure.

Pipeline might start with an API operation that returns a `CompletableFuture`, or with a "supply" method:

```
CompletableFuture.supplyAsync(Supplier<U> supplier)
```

Methods with the suffix "`async`" cause the work to be performed by a thread from a pool. Those without are executed in the thread that completes the `CompletableFuture`.

By default, the pool is the `ForkJoinPool.commonPool()`, but can be a caller specified pool.

© 2020 Simon Roberts

Operations on the `CompletableFuture`

An operation equivalent to a monadic "map" as `thenApply` / `thenApplyAsync`

There is no equivalent of "filter"

A "forEach" method is available as `thenAccept` / `thenAcceptAsync` (remember only one data item runs down a single `CompletableFuture`)

The "flatMap" method is called `thenCompose` / `thenComposeAsync`. Remember that the function provided to a flatMap must return a monad of the same type. This method is how we chain functions that complete asynchronously after immediately returning a `CompletableFuture`.

© 2020 Simon Roberts

Teeing pipelines

CompletableFuture pipelines can have as many subsequent pipelines as we desire, and the data produced will be propagated down each.

```
CompletableFuture<String> cfs =
    CompletableFuture.supplyAsync(()->"Hello");
cfs.thenAcceptAsync(
       m -> System.out.println("Message is " + m));
cfs
    .thenApplyAsync(m -> m.toUpperCase())
    .thenAcceptAsync(System.out::println);
```

© 2020 Simon Roberts

Rejoining pipelines

Multiple CompletableFutures can be collected together allowing combining of their output data. It's also possible to select the first to complete successfully:

```
cf1.acceptEither(cf2, consumer)
cf1.thenAcceptBoth(cf2, biConsumer)
cf1.applyToEither(cf2, function)
cf1.thenCombine(cf2, biFunction)
cf1.runAfterEither(cf2, runnable)
```

© 2020 Simon Roberts

Handling exceptions

Java's functional interfaces do not permit throwing checked exceptions, but unchecked exceptions can be thrown and cause the pipeline to jump to the second pipeline.

Normal pipeline elements will be skipped over by an exception, but the first handler will be invoked.

```
cf.handle(BiFunction<E, Throwable, R> fn)
```

fn is called with either a value, or an exception. The "missing" element is a null value. The result continues down the pipeline.

© 2020 Simon Roberts

Using `thenCompose`

If a utility method returns a CompletableFuture, it runs asynchronously, likely using a callback type approach internally. Include it in the CompletableFuture pipeline with the method thenCompose:

```
cf1.thenCompose(x -> functionReturningCompletableFuture(x))
```

© 2020 Simon Roberts

Wrapping callback functions

Given a callback-style asynchronous behavior, it can be wrapped to work with a CompletableFuture pipeline.

Assuming an asynchronous method makes a callback to some function fn, we can wrap this in a function that returns a CompletableFuture by a wrapper function that immediately returns a new CompletableFuture, and then calls the async operation with a callback that completes the CompletableFuture, either normally, or with an exception.

© 2020 Simon Roberts

Wrapping a callback to a `CompletableFuture`

Simulated callback function:

```
public static void callback(String s, Consumer<String> op) {
   new Thread(() -> {
      try {
         Thread.sleep(1000 + (int)(Math.random() * 1000));
         op.accept(s.toUpperCase());
      } catch (InterruptedException ie) {}
   }).start();
}
```

© 2020 Simon Roberts

Wrapping a callback to a `CompletableFuture`

Wrapping function:

```
public static CompletableFuture<String> doAsync(String x) {
   CompletableFuture<String> cfs = new CompletableFuture<>();
   callback(x, s -> cfs.complete(s));
   return cfs;
}
```

The callback lambda can, if needed, handle exceptions by calling `cfs.completeExceptionally(exception)`

© 2020 Simon Roberts

Wrapping a callback to a `CompletableFuture`

Calling the function that returns a CompletableFuture in a pipeline:

```
CompletableFuture.supplyAsync(() -> "hello")
    .thenApplyAsync(x -> x + " world!")
    .thenComposeAsync(x -> doAsync(x))
    .thenAcceptAsync(System.out::println)
    .join();
```

The `thenComposeAsync` method can be called with any method that returns a `CompletableFuture`, so when APIs are available that are built around `CompletableFuture`, this will be a key way to build code in those APIs.

© 2020 Simon Roberts