Effective Asynchronous Computations in Java 8

Claudiu Cosar
Programmer @SAP

How to achieve a correct concurrent model

- Immutability
- Thread-confinement
- Explicit synchronization

Previous asynchronous forms: Future

- Java 1.5
- Referenceable computation
- get() → ?

Alternative for Future-based programming model

CompletionService

- ExecutorCompletionService
- Producer-Consumer

Alternative for Future-based programming model 2

Building your own asynchronous abstraction on top of Future:

- SimplifiedCompletionFuture

  .**from**:: Callable<T> -> Wrapper<T>

  .**thenApply**::

  Wrapper<T>, Function<T, R> -> Wrapped<R>

Introducing CompletionStage

"A stage of a possibly ***asynchronous*** computation, that performs an action or computes a value ***when*** another CompletionStage completes. A stage completes upon termination of its computation, but this may in turn trigger other dependent stages"

- Java 1.8

Characteristics

- Callback Pattern without nesting
- Declarative Programming Model - Recipe
- Asynchronous(non-blocking), Event Driven
- Composable

Create from factory functions

- Factory constructor-functions
  - supplyAsync
  - runAsync
- ForkJoinPool.commonPool()

Asynchronous transformations

- Don’t block the main thread
- Timeout with a default “value”
- Differences between:
  - thenApply()
  - thenApplyAsync()

Mental mapping the API

Operation	Takes a:
supplyAsync	Supplier
runAsync	Runnable
thenApply	Function
thenAccept	Consumer
thenRun	Runnable
thenCompose	Function
thenCombine	Function

Attaching completion callbacks

- CompletableFuture
  - exceptionally(Function)
  - whenComplete(BiConsumer)
  - handle(BiFunction)

CompletionStage as “promise”

- Promises can control the execution flow(synchronization)
- Promises can only be completed once
- Dependent parties “inherit” “delivered”- errors
- …can …lead to deadlocks when misused
- Java mixes the concepts

Composing Stages

- Composing completion-stages
- Async variations are also available
- Stage-failings stops propagation

Combining Stages

- Combines two stages via a function
- Stages are processed in parallel asynchronously
- Used when two tasks can be divided and computed independently
- Async variations are available

Fastest & Barrier Semantic 1

- acceptEither:: CF<T>, CF<T>, Consumer<T>
- thenAcceptBoth:: CF<T>, CF<R>, BiConsumer<T, R>
- Async variations also available

Fastest & Barrier Semantic Constructors

- Exposes factory functions for
  - waiting for all – barrier-semantic: allOf()
  - waiting for first fastest compution: anyOf()
- Lack a “consistent” parameterized returned Type

Cancelling computations

- Interrupting used to work with plain ol' Future
- Interrupting is changed for CompletionStage (consider it a "feature")
- "[…] interrupts should not control processing"