
ORACLE®


Java™
ORACLE®

Completable Future

Srinivasan Raghavan
Senior Member of Technical Staff
Java Platform Group

Program Agenda

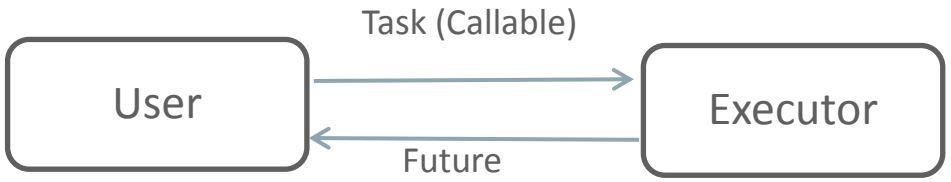
1. java.util.Future Introduction
2. Cloud Services Design and the fight for Performance
3. CompletableFuture and power of parallelism
4. Building powerful libraries with Completable Future

java.util.Future Introduction

- Before jdk1.5 , no java.util.concurrent.*, only threads ,synchronized primitives
- “Write once , run anywhere ?” – great success , “But run it a million times , work the same ?“ big question
- In came java.util.concurrent.* with executor service and future tasks and many other concurrency constructs
- A java.util.Future is a construct which holds a result available at a later time
- Future is asynchronous , but its not non-blocking

The Executor Service and Future Task

- Executors contains pool of threads which accepts tasks and supplies to the Future
- When a task is submitted to the executor it returns a future and future.get() would block the computation until it ends




```
ExecutorService executorService =
Executors.newFixedThreadPool(20);

Future<Integer> future =
executorService.submit(new Callable<Integer>()
{
@Override
public Integer call() throws Exception {
return 42;
}
});

System.out.println(future.get());

executorService.shutdown();
```

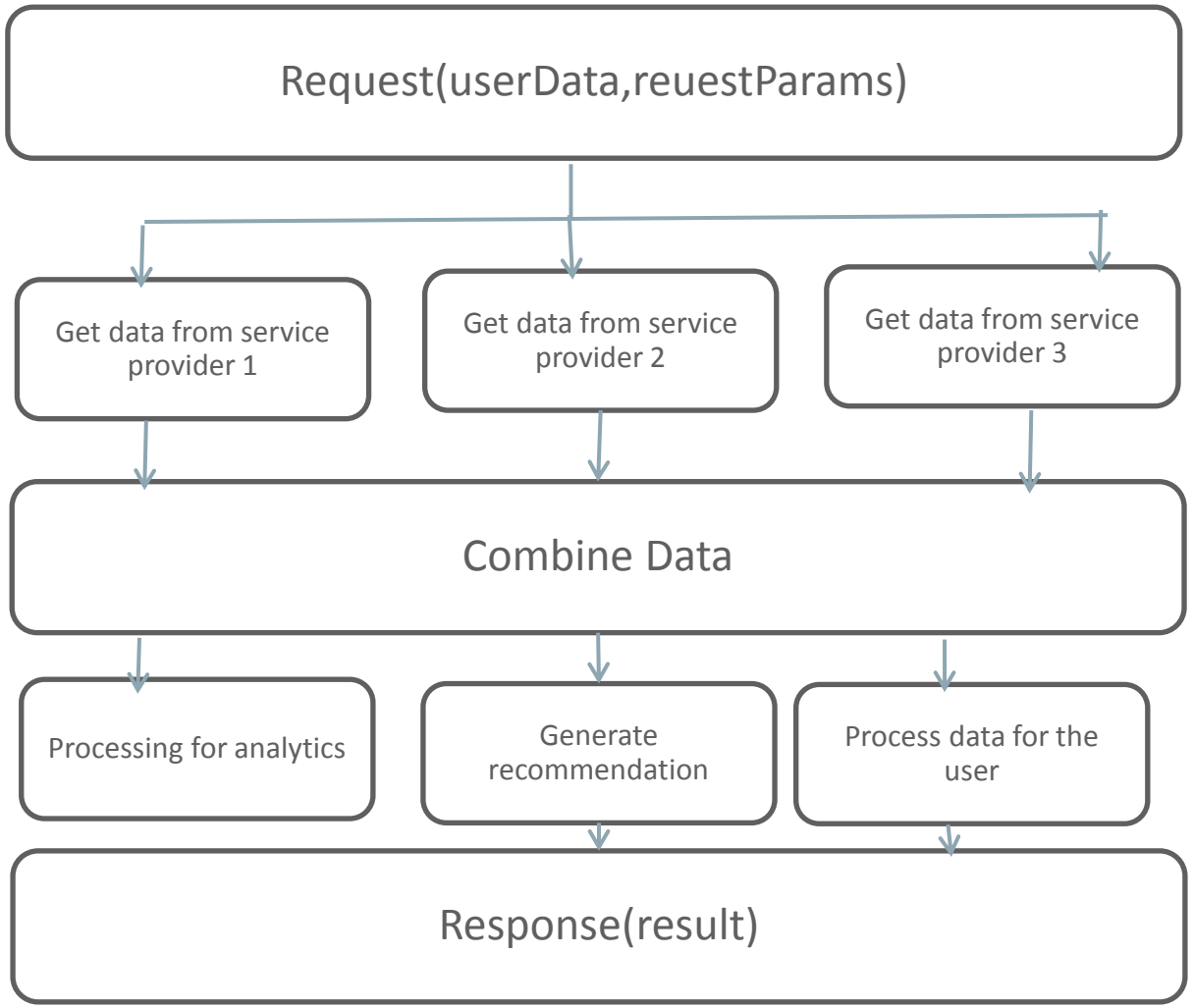
What can be done ?

- Future can allow computation in the background so improving performance
- Future.get() would block the thread and wait till the computation is complete and get the result
- Can get an exception if there is a failure
- future.cancel() cancels the computation
- future.isDone() checks the computation is complete
- And that's it /////

```
ExecutorService executorService =
Executors.newFixedThreadPool(20);

Future<Integer> future =
executorService.submit(new Callable<Integer>()
{
@Override
public Integer call() throws Exception {
//Some complex work
return 42;
}
});

System.out.println(future.get());

executorService.shutdown();
```

Distributed Cloud Services Design and the fight for Performance

A Typical distributed cloud service



Performance Bottlenecks and impacts

- Java.util.Future can help in parallelism
- But it cannot help with pipeline the tasks and managing the thread pool for you
- Future does not supports call backs and chaining of operations
- Building libraries with future can be complictaed
- Performing in a serial way can impact the latency big time
- It can destroy all benefits of having distributed services
- No amount of horizontal and vertical scaling can help
- Without dynamic services offerings business can be affected

CompletableFuture and power of parallelism

What is Completable future ?

- Its new addition to java 8
- Asynchronous, allows registering asyc callbacks just like java scripts, event-driven programming model
- Support for depending functions to be triggered on completion
- Each stage can be executed with different executor pool
- Also comes with a built in thread pool for executing tasks
- Built in lambda support ,super flexible and scalable api

Basics

```
 CompletableFuture<String> future =
CompletableFuture.supplyAsync(new Supplier<String>()
{
        @Override
        public String get() {
        // ...long running...
           return "42";
        }
        }, executor1);

future.get();
```

```
//Come on ...  we are in JDK 8 !!!

 CompletableFuture<String> future =
         CompletableFuture
             .supplyAsync(() -> "42",executor1);
future.get();
```

Lambdas , Crash Recap

Lambdas

- Function definition that is not bound to an identifier

```
/**
 *
 * This is a piece of code that illustrate how lambdas work in
Java 8
 *
 * Given an example
 *
 * f(x) = 2x+5;
 *
 * given x= 2 ; f(x) = 9 ;
 *
 * z(x)= f(g(x)) where g(x) =3x+5
 *
 * z(x) = 2(3x+5) +5 = 6x+15 12+15 = 27
 * *
 */
```

```
public class LambdaUnderStanding2 {
	@FunctionalInterface
	static interface Funct {

		int apply(int x);

		default Funct compose(Funct before) {
			return (x) -> apply(before.apply(x));
		}
	}

	public static void main(String[] args) {

		Funct annoymous = new Funct() {

			@Override
			public int apply(int x) {
				return 2 * x + 5;;
			}
		};


		Funct funct = (x) -> 2 * x + 5;

		System.out.println(funct.apply(2));


		Funct composed = funct.compose((x) -> 3 * x + 5);

		System.out.println(composed.apply(2));
	}
}
```

Java 8 Functional Interface

```
Predicate<Integer> test = (x) -> x> 10;

Function<Integer, Integer> function = (x) -> 3 * x + 5;

Consumer<Integer> print = (x) -> System.out.println(x);

BiFunction<Integer, Integer, Integer> biFunction = (x, y) -> 3 * x + 4* y + 2;

Supplier<Integer> supplier = () -> Integer.MAX_VALUE;
```

thenApply() -transformations

```
 /**
 *  Classic call back present in javascript or scala .
 *  theApply is like run this function when the result
*arrives  from previous stages
 */

final CompletableFuture<Integer> future1 =
CompletableFuture
            .supplyAsync(() -> "42", executor1).thenApply(
                        (x) -> Integer.parseInt(x));
```

```
/**
 * thenApply is trasformative changing
CompletableFuture<Integer> to
 * CompletableFuture<Double>
 */
CompletableFuture<Double> future2 = CompletableFuture
            .supplyAsync(() -> "42", executor1)
            .thenApply((x) -> Integer.parseInt(x))
            .thenApply(r -> r * r * Math.PI);
```

```
/**
 * U can supply a differnt  executor pool for
thenApply
 */

CompletableFuture<Double> future = CompletableFuture
.supplyAsync(() -> "42", executor1)
.thenApplyAsync((x) -> Integer.parseInt(x),
executor2)
.thenApplyAsync(r -> r * r * Math.PI, executor2);
```

thenCombine() , whenComplete() –completion

```
final CompletableFuture<Integer> future = CompletableFuture
.supplyAsync(() -> "32", executor1).thenApply(
(x) -> Integer.parseInt(x));

CompletableFuture.supplyAsync(() -> "42", executor2)
.thenApply((x) -> Integer.parseInt(x))
.thenCombine(future, (x, y) -> x + y)
.thenAccept((result) -> System.out.println(result));
```

```
/**
 * When complete is final stage where it can check
the execpetions
 * propagated and pass results through it
 */
final CompletableFuture<Integer> future =
CompletableFuture
.supplyAsync(() -> "42", executor1)
.thenApply((x) -> Integer.parseInt(x))
.whenComplete(
(x, throwable) -> {
if (throwable != null) {
Logger.getAnonymousLogger().log(Level.SEVERE,
"Logging" + throwable);

} else {
Logger.getAnonymousLogger().log(Level.FINE,
" Passed " + x);
}
});
```

thenCombine() allOf() - combining futures

```
/**
 * Combining two dependent futures
 */
final CompletableFuture<Integer> future =
CompletableFuture
.supplyAsync(() -> "32", executor1).thenApply(
(x) -> Integer.parseInt(x));

CompletableFuture.supplyAsync(() -> "42", executor2)
.thenApply((x) -> Integer.parseInt(x))
.thenCombine(future, (x, y) -> x + y)
.thenAccept((result) -> System.out.println(result));
```

```
/**
 * Combining n  futures unrelated
 */

CompletableFuture<Void> future2 = CompletableFuture
.supplyAsync(() -> "42", executor1)
.thenApplyAsync((x) -> Integer.parseInt(x),
executor2)
.thenAcceptAsync(
(x) -> Logger.getAnonymousLogger().log(Level.FINE,
"Logging" + x), executor2);

CompletableFuture<Void> future1 = CompletableFuture
.supplyAsync(() -> "32", executor1)
.thenApplyAsync((x) -> Integer.parseInt(x),
executor2)
.thenAcceptAsync(
(x) -> Logger.getAnonymousLogger().log(Level.FINE,
"Logging" + x), executor2);

CompletableFuture.allOf(future1,future2).join();
```

Building powerful libraries with Completable Future

Building powerful libraries with completable futures

- Building scalable service orchestrator
- Building dynamic http client framework
- Improve parallelism in existing services with are done serial
- Building libraries tuned for vertical scalability

References

- https://docs.oracle.com/javase/tutorial/
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html
- http://cs.oswego.edu/mailman/listinfo/concurrency-interest
- https://github.com/srinivasanraghavan/functional

Questions ?


Java™
ORACLE®