Parallel Streams, CompletableFutures, and All That

...

Parallelism and Concurrency in Java

Contact Info

Ken Kousen

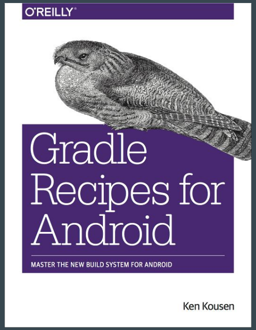
Kousen IT, Inc.

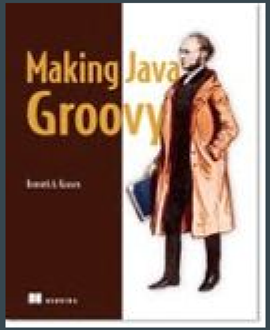
ken.kousen@kousenit.com

http://www.kousenit.com

http://kousenit.org (blog)

@kenkousen





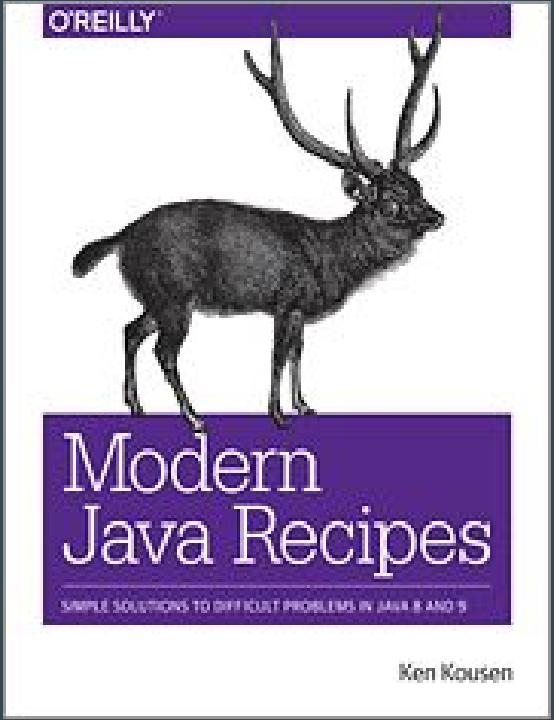
Modern Java Recipes

Examples are from the book

Source code:

https://github.com/kousen/java_8_recipes

https://github.com/kousen/cfboxscores



Videos (available on Safari)

O'Reilly video courses: See Safari Books Online for details

- Groovy Programming Fundamentals
- Practical Groovy Programming
- Mastering Groovy Programming
- Learning Android
- Practical Android
- Gradle Fundamentals
- Gradle for Android
- Spring Framework Essentials
- Advanced Java Development

Let's Get This Out Of The Way...

Concurrency:

Multiple tasks can run at the same time

You design for concurrency

Parallelism:

Task actually run simultaneously

Simple Made Easy

Keynote by Rich Hickey

- http://www.infoq.com/presentations/Simple-Made-Easy
- Converting to parallel streams is easy
- That doesn't make concurrency or parallelism simple

Going Parallel

"Parallelism is strictly an optimization"

-- Brian Goetz

Five-part series of articles on Java Streams at IBM DeveloperWorks

https://www.ibm.com/developerworks/library/j-java-streams-4-brian-goetz/index.html

Converting a stream

By default, all stream factory methods result in sequential streams

```
Collection.stream()
```

(as opposed to `Collection.parallelStream()`)

```
Stream.of(T...)

Stream.iterate(seed, UnaryOperator<T>)

Stream.generate(Supplier<T>)
```

Converting a stream

`Stream.parallel()`

`Stream.sequential()`

Intermediate operations

Return new streams, or the same if already as required

Check with `isParallel()`

Converting a stream

Note:

Can't do both sequential and parallel in same pipeline

`SequentialToParallelTest.java`

When is Parallel Worth It?

Requirements:

- Operations are independent and associative
  - a op (b op c) === (a op b) op c

When is Parallel Worth It?

Requirements:

- Operations are independent and associative
  - a op (b op c) === (a op b) op c
- Either lots of data, or long processing per element
  - N * Q > 10000

When is Parallel Worth It?

Requirements:

- Operations are independent and associative
  - a op (b op c) === (a op b) op c
- Either lots of data, or long processing per element
  - N * Q > 10000
- Data easy to partition
  - Arrays are good, linked lists are bad

JMH

Java Microbenchmark Harness

Part of the OpenJDK project

http://openjdk.java.net/projects/code-tools/jmh/

IntelliJ plugin

https://github.com/artyushov/idea-jmh-plugin

Parallel Streams

By default, uses the common `ForkJoinPool`

Note: implements `ExecutorService`

Performs work stealing

Default pool size:

```
ForkJoinPool.commonPool().getPoolSize() ==
        Runtime.getRuntime().availableProcessors() - 1
```

Parallel streams

Replace `stream()` with `parallelStream()`

- Introduces overhead
- If previous conditions apply, may help a lot

`ParallelDemo.java`

`DoublingDemo.java` (JMH)

Changing the common pool size

Use -D flag

```
java.util.concurrent.ForkJoinPool.common.parallelism
```

Equivalently,

```
System.setProperty("...above...", 16)
```

`CommonPoolSize.java`

Future

`Future<T> ExecutorService.submit(Callable<T> callable)`

Callable is a functional interface, so use a lambda expression

Method calls return immediately, but

you have to call `get()` (a blocking call) to retrieve the result

Future

Difficult to coordinate multiple futures

```
while (!future.isDone()) {
      System.out.println("Waiting...");
}
```

busy waiting

Can generate billions of calls ... not a good idea

`FutureDemo.java`

CompletableFuture

Great for coordination

But first, how do you complete a CompletableFuture?

- `complete(T value)`
- `completedFuture(U value)`
- `completeExceptionally(Throwable ex)`

Why all three?

`CompletableFutureDemos.java`

```java
private Map<Integer, Product> cache =
                    new ConcurrentHashMap<>();

private Product getLocal(int id) { return cache.get(id); }

private Product getRemote(int id) {
    try {
        Thread.sleep(100);
        if (id == 666) {
            throw new RuntimeException("Evil request");
        }
    } catch (InterruptedException ignored) { }
    return new Product(id, "name");
}
```

```java
public CompletableFuture<Product> getProduct(int id) {
    try {
        Product product = getLocal(id);
        if (product != null) {
            return CompletableFuture.completedFuture(product);
        } else {
            CompletableFuture<Product> future = new CompletableFuture<>();
            Product p = getRemote(id);  // legacy, synchronous
            cache.put(id, p);
            future.complete(p);
            return future;
        }
    } catch (Exception e) {
        CompletableFuture<Product> future = new CompletableFuture<>();
        future.completeExceptionally(e);
        return future;
    }
}
```

Running asynchronously

- `CompletableFuture<T> implements Future<T>, CompletionStage<T>`
- CompletionStage has 38 methods
    - Lots of overloads

CompletableFuture

Some mnemonics:

- apply methods take a `Function`
- accept methods take a `Consumer`
- run methods take a `Runnable`
- supply methods take a `Supplier`

CompletableFuture

```
stage.thenApply(x -> square(x))
     .thenAccept(x -> System.out.print(x))
     .thenRun(() -> System.out.println())
```

CompletableFutureTests.java

CompletableFuture

More patterns in method names:

- then
- either
- both
- combine

CompletableFuture

- Method names often have additional suffix **async**
  - Without async, in same thread as caller
  - With async, re-submitted to thread pool

CompletableFuture

Also overloaded to take an extra arg of type Executor

- Without, use common ForkJoinPool
- With, use supplied thread pool

`CompletableFutureTests.java`

```
public CompletableFuture<Product> getProduct(int id) {
    try {
        Product product = getLocal(id);
        if (product != null) {
            return CompletableFuture.completedFuture(product);
        } else {
            // async
            return CompletableFuture.supplyAsync(() -> {
                Product p = getRemote(id);
                cache.put(id, p);
                return p;
            });
        }
    } catch (Exception e) {
        CompletableFuture<Product> future = new CompletableFuture<>();
        future.completeExceptionally(e);
        return future;
    }
}
```

Completing the CompletableFuture

`get()` blocks, declares `ExecutionException, InterruptedException`

`join()` blocks, declares (unchecked) `CompletionException`

Can just wait for the pool to become "quiescent"

Await quiescence

```
ForkJoinPool.commonPool()
    .awaitQuiescence(1, TimeUnit.SECONDS);
```

AwaitQuiesenceTest.java

All of

`CompletableFuture.allOf(CompletableFuture<?>... cfs)`

- static method
- returns `CompletableFuture<Void>`

Use `join()` to wait for all to be done

Post-process using streams to extract results

`AllOfDemo.java`

Boxscores

- Bigger demo
  - Major League Baseball boxscores
    - http://gd2.mlb.com/components/game/mlb/
  - Subdirectories for year, month, day
  - Boxscores in JSON format for each game on a given day

Boxscores

1. Access site for games on a range of dates
2. Determine game links for each day
3. Download JSON boxscore for each game
4. Transform JSON data to objects
5. Save results to local files
6. Determine scores of each game
7. Determine game with highest total score
8. Print individual game scores, along with max game and max score

GitHub repo: https://github.com/kousen/cfboxscores

Summary

- Going parallel is easy, benefitting from it is hard
- Parallel streams use common ForkJoinPool
- CompletableFuture lets you coordinate futures
- Many, many methods to do the coordination