Key Scheduler Operators for Project Reactor Reactive Types (Part 3)

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University

Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand the capabilities of the ParallelFlux class
- Recognize how Scheduler operators are used with ParallelFlux
  - These operators provide the context to run other operators in designated threads & thread pools
    - e.g., Schedulers.boundedElastic()

These operators also work with the Flux & Mono classes

Key Scheduler Operators for Project Reactor Reactive Types

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers

```
static Scheduler
       boundedElastic()
```

See projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#boundedElastic

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
    - Returns a new Scheduler that is suited for I/O-bound work

```
static Scheduler
       boundedElastic()
```

i.e., threads can be dynamically added or removed from the pool

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking tasks

```
public abstract class Schedulers
extends Object
```

`Schedulers` provides various `Scheduler` flavors usable by `publishOn` or `subscribeOn` :

- `parallel()`: Optimized for fast `Runnable` non-blocking executions
- `single()`: Optimized for low-latency `Runnable` one-off executions
- `boundedElastic()`: Optimized for longer executions, an alternative for blocking tasks where the number of active tasks (and threads) is capped
- `immediate()`: to immediately run submitted `Runnable` instead of scheduling them (somewhat of a no-op or "null object" `Scheduler`)
- `fromExecutorService(ExecutorService)` to create new instances around `Executors`

Factories prefixed with `new` (eg. `newBoundedElastic(`
`String)` return a new instance of their flavor of `Schedu` er
factories like `boundedElastic()` return a shared inst e
one used by operators requiring that flavor as their defa Schedu . All
instances are returned in a `initialized` state.

See projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html

Key Scheduler Operators for Project Reactor Reactive Types

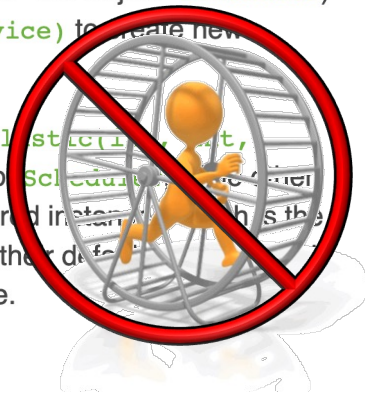
- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking tasks
        - i.e., I/O-bound tasks *not* compute-/CPU-bound tasks!

```
public abstract class Schedulers
extends Object
```

`Schedulers` provides various `Scheduler` flavors usable by `publishOn` or `subscribeOn` :

- `parallel()`: Optimized for fast `Runnable` non-blocking executions
- `single()`: Optimized for low-latency `Runnable` one-off executions
- `boundedElastic()`: Optimized for longer executions, an alternative for blocking tasks where the number of active tasks (and threads) is capped
- `immediate()`: to immediately run submitted `Runnable` instead of scheduling them (somewhat of a no-op or "null object" `Scheduler`)
- `fromExecutorService(ExecutorService)` to create new instances around `Executors`

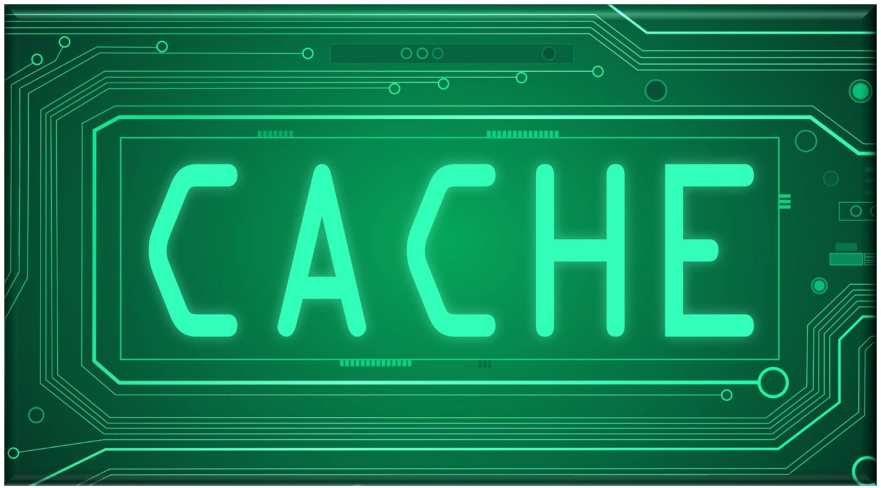
Factories prefixed with `new` (eg. `newBoundedElastic(...)`, `String)` return a new instance of their flavor of `Scheduler`, while factories like `boundedElastic()` return a shared instance, which is the one used by operators requiring that flavor as their default. All instances are returned in a `initialized` state.



I/O bound tasks can benefit from more threads, where CPU-bound tasks can't

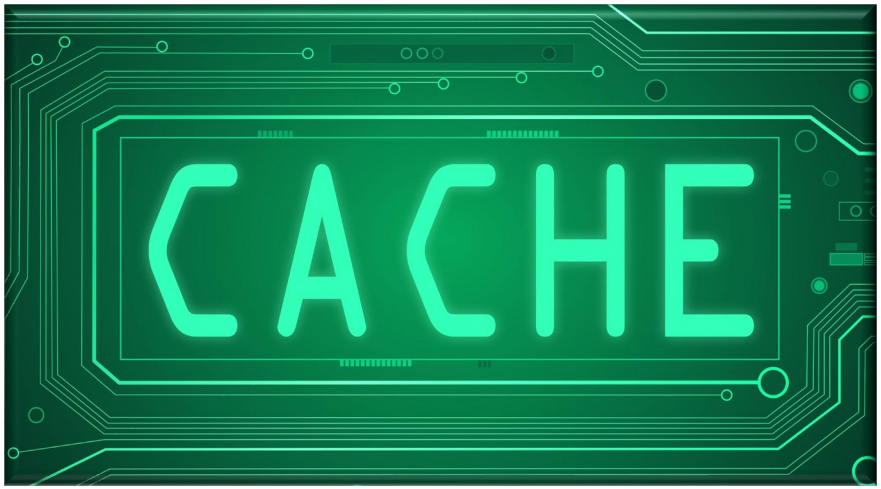
Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking tasks
      - Either starts a new thread or reuses an idle one from a cache



Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking tasks
      - Either starts a new thread or reuses an idle one from a cache



The underlying threads can be evicted if idle for more than 60 seconds

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking tasks
    - Either starts a new thread or reuses an idle one from a cache
      - The goal is to maximally utilize the CPU cores

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking tasks
      - Either starts a new thread or reuses an idle one from a cache
    - The max # of created threads is bounded by a cap
      - By default, this # is ten times the # of available CPU cores

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
    - Returns a new Scheduler that is suited for I/O-bound work
      - Optimized for blocking tasks
      - Either starts a new thread or reuses an idle one from a cache
      - The max # of created threads is bounded by a cap
      - The max # of task submissions enqueued & deferred on each of these backing threads is also bounded
        - By default, 100K additional tasks

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
  - Used for making network calls, file I/O, database operations, etc.

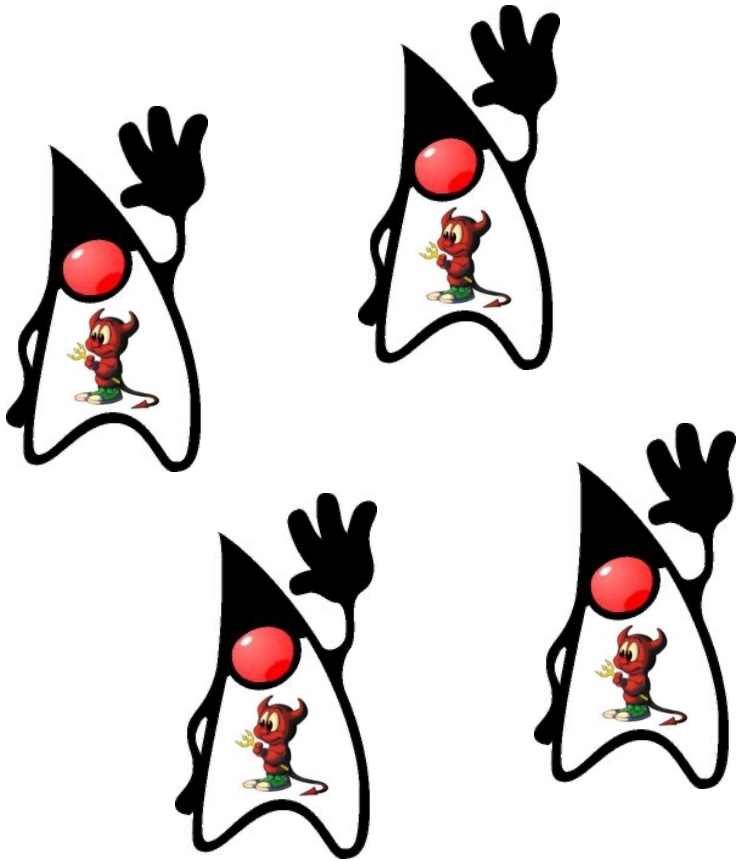
e.g., download images from remote web servers in parallel & store them on the local computer

```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/flux/ex5

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
  - Used for making network calls, file I/O, database operations, etc.
  - Implemented via "daemon threads"
    - i.e., won't prevent the app from exiting even if its work isn't done



See www.baeldung.com/java-daemon-thread

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
  - Used for making network calls, file I/O, database operations, etc.
  - Implemented via "daemon threads"
- The Schedulers.io() operator in RxJava is similar

io

```
@NonNull
public static @NonNull Scheduler io()
```

Returns a default, shared `Scheduler` instance intended for IO-bound work.

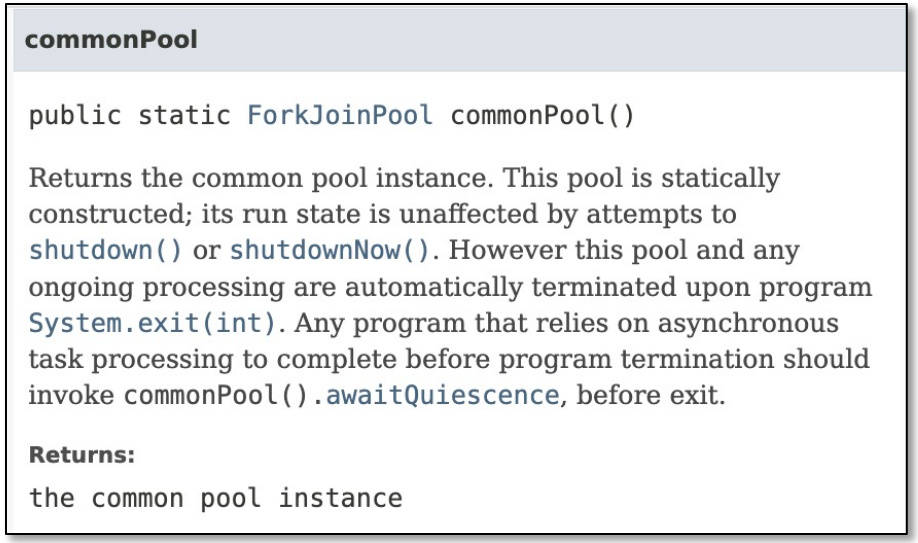
This can be used for asynchronously performing blocking IO.

The implementation is backed by a pool of single-threaded `ScheduledExecutorService` instances that will try to reuse previously started instances used by the worker returned by `Scheduler.createWorker()` but otherwise will start a new backing `ScheduledExecutorService` instance. Note that this scheduler may create an unbounded number of worker threads that can result in system slowdowns or `OutOfMemoryError`. Therefore, for casual uses or when implementing an operator, the Worker instances must be disposed via `Disposable.dispose()`.

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/schedulers/Schedulers.html#io

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
  - Used for making network calls, file I/O, database operations, etc.
  - Implemented via "daemon threads"
  - The Schedulers.io() operator in RxJava is similar
- The Java common fork-join pool is also similar



commonPool

```
public static ForkJoinPool commonPool()
```

Returns the common pool instance. This pool is statically constructed; its run state is unaffected by attempts to `shutdown()` or `shutdownNow()`. However this pool and any ongoing processing are automatically terminated upon program `System.exit(int)`. Any program that relies on asynchronous task processing to complete before program termination should invoke `commonPool().awaitQuiescence`, before exit.

Returns:

the common pool instance

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#commonPool

Key Scheduler Operators for Project Reactor Reactive Types

- The Schedulers.boundedElastic() operator
  - Dynamically creates a bounded # of ExecutorService-based workers
  - Used for making network calls, file I/O, database operations, etc.
  - Implemented via "daemon threads"
  - The Schedulers.io() operator in RxJava is similar

- The Java common fork-join pool is also similar
  - Especially when used with the ManagedBlocker mechanism..

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html

Programming with Schedulers.boundedElastic()

Programming with Schedulers.boundedElastic()

- Download images from remote web servers in parallel & store them on the local computer

```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Reactive/flux/ex5

Programming with Schedulers.boundedElastic()

- Download images from remote web servers in parallel & store them on the local computer

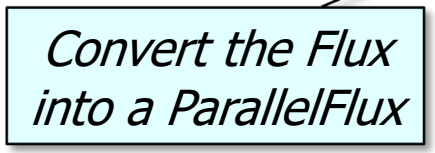
```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```

Create a Flux containing URLs to download from remote web servers

Programming with Schedulers.boundedElastic()

- Download images from remote web servers in parallel & store them on the local computer

```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```



Programming with Schedulers.boundedElastic()

- Download images from remote web servers in parallel & store them on the local computer

```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```

Designate the I/O Scheduler that will download & store each image in parallel

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/parallel/ParallelFlux.html#runOn

Programming with Schedulers.boundedElastic()

- Download images from remote web servers in parallel & store them on the local computer

```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```

Download & store images in parallel

Programming with Schedulers.boundedElastic()

- Download images from remote web servers in parallel & store them on the local computer

```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```

Merge the values from each 'rail' in a round-robin fashion & expose it as a regular Flux sequence

Programming with Schedulers.boundedElastic()

- Download images from remote web servers in parallel & store them on the local computer

```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```

Collect the Flux into a List

Programming with Schedulers.boundedElastic()

- Download images from remote web servers in parallel & store them on the local computer

```
return Options.instance()
   .getUrlFlux()

   .parallel()

   .runOn(Schedulers
          .boundedElastic())

   .map(downloadAndStoreImage)

   .sequential()

   .collectList()

   .doOnSuccess(...)
```

Handle the final 'reduced' results

End of Key Scheduler Operators for Project Reactor Reactive Types (Part 3)