Java library for implementation and scheduling of tasks in a dependency DAG

Blagoj Atanasovski 139066

Overview

This whitepaper gives a brief overview of the implementation of a Java library designed for quick and easy:

- implementation of parallel tasks
- communication between the tasks
- dependency specification
- safe usage of shared resources
- and scheduling the tasks on the execution environment

Many high level programing languages offer APIs for defining tasks that can be run on separate threads, and language constructs to synchronize work between these threads. The Java programming language in particular offers APIs for defining and working with threads (java.lang.Thread), interfaces for things the threads can execute (java.lang.Runnable, and

java.lang.Callable), and executor services that provide cached thread pools, mechanisms for scheduling repeated tasks and many other abilities (java.util.concurrent.Executors). But when it comes to defining dependencies between the tasks and inter-thread communication, then developers must define and implement it themselves.

The goal of this library is to create a flexible framework for a developer to define the subtasks of an algorithm as easily as implementing Runnables, having data-flow (pass results from one task as input to a following task) ingrained in the framework, dependency definition to be trivial, hide synchronization details between tasks as much as possible by using the libraries own implementation of a shared resource, and be able to schedule these tasks using various scheduling algorithms with limited or unlimited amount of concurrency between the tasks.

Usage

Executable

In order for the library to be used, the developer must implement only the executables (subtasks of the algorithm). The executables represent the logic of the algorithm, the workforce. Each executable may be supplied with input parameters, either at the time of its initialization, or the results from a previous executable that is a dependency for the current one. Also an executable can produce output to be used as the result of the algorithm or be passed to the dependants of the executable.

```
class ReduceWithSum extends Executable {
    public ReduceWithSum(String id) {
        super(id);
    }

    @Override
    public void execute() {
        List<Object> input = get("input");
        int result = input.stream()
                .map(element -> (int) element)
                .reduce(0, (k, l) -> k + l);
        produce("output", result);
    }
}
```

Listing 1. Example implementation of executable

As shown in Listing 1, to implement a task, the developer needs to extend the abstract class Executable. The IDE prompts the developer at compile time that the *void execute()* method needs to be implemented, and a constructor that passes an id to the superclass.
In the execute method, input for the task is obtained from the *List<Object> get(String paramName)* method. The method returns a list of input objects because a task may be dependent on multiple other tasks, that work on parts of the same data. I.E. an array of 10k strings needs to be filtered for specific prefixes, 2 tasks do this in parallel, they both supply

their results to a 3 task that starts after they finish, the third task needs to have all the filtered items from both tasks.

When the task has a result that it wants to pass onto it's dependants or as a result from the algorithm, it needs to call the *void produce(String paramName, Object result)* method. Calling this method multiple times with the same parameter name does not override the previous result for that name, just adds another value to a list.

Schedule

When all the subtasks of the algorithm are defined and implemented, in order to execute them, a developer needs to define the dependency DAG. The dependencies are added to a *Schedule*. Let us say that we have implemented tasks in a class named SomeTask, and want to create a schedule.

```
void main(){
      SomeTask a = new SomeTask("a");
      SomeTask b = new SomeTask("b");
      SomeTask c = new SomeTask("c");
      Schedule someSchedule = new Schedule();
      someSchedule.add(a).add(b).add(c,a,b);
}
```

Listing 2. An example schedule with 3 tasks a,b and c. Task c depends on a and b.

The example in Listing 2, defines 3 tasks, and in order to execute task *c*, both *a* and *b* must complete first. The tasks *a* and *b* can execute in parallel. The *Schedule add(Executable task, Executable… dependencies)* of the Schedule class returns the same instance so we can chain the calls for clearer syntax. It also accepts a variable argument length for the dependencies of the task being added.

Scheduler

After the tasks have been defined and the dependencies between them, they are ready to be executed. Tasks are executed in a Scheduler. The library provides the implementation, and the developer only has to choose the scheduling algorithm used, and whether to bound the number of threads in parallel to be used for the execution of the tasks.

```
void main() {
      Task a = new Task("a");
      Task b = new Task("b");
      Task c = new Task("c");
      Schedule schedule = new Schedule();
      schedule.add(a).add(b).add(c,a,b);

      Scheduler scheduler;
      scheduler = new Scheduler(HLFETSchedulingAlgorithm());
      scheduler.execute(schedule);
      System.out.println(scheduler.getResults().toString());
}
```

Listing 3. Example of defining a scheduler and executing a schedule

The scheduler provides two constructors, both require the scheduling algorithm, the second also requires the number of threads to use. The *Map<String, List<Object>> getResults()* method returns a map with all are all the parameter names (and values) produced by subtasks of the graph that have no dependants.

Shared Resource

The framework allows the developer to easily define and safely use resources (objects) that need to be shared (and accessed from) multiple threads. Shared resources force the developer to lock them before attempting to read or modify their values, and unlock them when they are done. If a resource has not been locked by a thread that is trying to use it it will throw a runtime exception. When a thread tries to lock a resource, that thread will block until the resource is free for locking. Additionally if the developer wants to access the value stored in the shared resource, they must define an Operation Function. A shared resourceis a generic type, and to dissuade the developer from obtaining a reference to the internal object that is kept in the shared resource and then be able to use it in an unsafe manner, operations on the value must be controlled.

```
SharedResource<String> res = new SharedResource<>("test");
res.lock();
SharedResource<String>.ResourceOperation<String> operation;
operation = res.createSafeGet(String::new);
try {
   String value = operation.getResult();
   value += " test";
   res.set(value);
} catch (IllegalAccessException e) {
   e.printStackTrace();
}

try {
   res.unlock();
} catch (IllegalAccessException e) {
   e.printStackTrace();
}
```

Listing 4. Example of how to use a shared resource safely

In listing 4, a shared resource of type String is defined with a starting value “test”. The example calls the lock method, and after a lock has been obtained by the current thread execution continues. A resource operation that takes the value of the resource as input, and produces a String as output is created with the createSafeGet method, where the String::new method reference creates a clone of the input. When the getResult method of the ResourceOperation object is called, it will too check if the current thread has acquired a lock, and throw an exception if not. The ResourceOperation does not guarantee complete safety as the developer can again return the reference to the value directly. Further

development of the project should enforce that either only immutable objects can be set as shared, or that they implement the Clonable interface.

Implementation

The library is implemented in the Java 8, with dependencies on the JGraphT library and the SLF4J logging facade. JGraphT[1] is a free Java graph library that provides mathematical graph-theory objects and algorithms. It is used internally in the library for the representation of the task dependencies as a directed graph, and by the scheduling algorithms to easily navigate through the dependency graph. The Simple Logging Facade for Java (SLF4J)[2] serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time. Both of these are found in the Central Maven Repository

The project uses the gradle[3] for setting up, dependency management and building the project. The source code is hosted as open source on GitHub under the repository DAG_Task_Scheduler.

Executable

If the developer wants to create a task that will be executed, they need to extend the Executable abstract class. This abstract class implements the Runnable interface, so it can be executed by a Java executor from the java.util.concurrent package.

The runnable interface requires an implementation for the *void run()* method. The run method checks if all the fields have been initialized properly, calls the abstract *void execute()* method, after the execution is done checks if errors were produced, and either notifies the scheduler that this task completed successfully or with an error.

The Executable task contains in itself a field that contains the input parameters passed to the task before execution starts, and a field that will collect the output of the task, the scheduler will take the output and pass it on to the following tasks as input. Input parameters are accessed through a protected getter, so that only a subclass of Executable can access them, and a protected produce method, for outputting data.

Each executable has fields that are not required to be always set. ExecutionTime is a field that is supposed to hold an approximation of the time/work required to execute the task. Execution time is required by static scheduling algorithms beforehand. ExecutionWeight is a field that holds a weight value that each different scheduling algorithm can define and set by himself, later to use it as a priority measure for selecting the next task to execute.

```
public abstract class Executable implements Runnable
      // Constructors
      protected Executable(String id, int executionTimeEstimate)
      protected Executable(String id)
      // Public methods
      public abstract void execute();
```

[1] http://jgrapht.org/
[2] http://www.slf4j.org/
[3] http://gradle.org/

```
      public void setExecutionWeight(float weight)
      public final boolean equals(Object o)
      public final int hashCode()
      public Map<String, List<Object>> getOutputParameters()
      public final String getId()
      public void run()
      public Executable addInput(Map<String, List<Object>> parameters)
      public Executable addInput(String parameterName, Object... values)
      public Executable addInput(String parameterName, Collection<? extends
Object> values)
      public final List<String> getErrors()
      public void setScheduler(Scheduler s)
      public int getExecutionTime()
      public float getExecutionWeight()
      public boolean hasExecutionWeight()
      public String toString()
      // Protected methods
      protected List<? extends Object> get(String inputName)
      protected final void produce(String outputName, Object result)
      protected final void error(String error)
      //Private methods
      private final void setOutputParameters()
      private final Map<String, List<Object>> getInputParameters()
 }
```

Listing 4. The methods of the Executable abstract class

Schedule

The dependencies between tasks are recorded in a Schedule. The schedule contains a DirectedGraph[4] instance, where the vertices are Executables, and the dependencies between them are the edges. Adding new tasks is done with the *Schedule add(Executable task, Executable… dependsOn)* method that takes a variable length of Executables that the task added depends on. The method returns the same instance of the Schedule to allow method chaining and cleaner syntax.

The scheduler informs the schedule that a task has finished executing (and the schedule updates its dependency graph accordingly) with the *notifyDone* or *notifyError* methods. If the schedule is notified for an error, the scheduler stops the execution. The boolean isDone() method tells the executor if all the tasks have been scheduled and there have been no errors.

The *void setAsStarted(Executable task)* method specifies that a task has started, it is not removed from the graph, but can not be returned no more by the *Executable[] getReadyTasks()* method. Ready tasks are tasks whose dependencies have finished with execution and they haven't already been started.

```
public class Schedule
      public ConcurrentMap<String, List<Object>> getResults()
      public Schedule add(Executable task, Executable... dependsOn)
```

[4] From the jGraphT library

```
	public synchronized boolean isDone()
	public synchronized Executable[] getReadyTasks()
	public synchronized void notifyDone(Executable task)
	public synchronized void notifyError(Collection<String> error)
	public boolean hasErrors()
	public void setAsStarted(Executable task)
	public List<String> getErrors()
	public DirectedGraph<Executable, DefaultEdge> getDependencies()
}
```

Listing 5. The methods of the Schedule class

Scheduling algorithm

The scheduler uses different scheduling algorithms to pick which tasks out of a set of ready tasks to schedule next for execution. There are several scheduling algorithms implemented in the project, they implement the SchedulingAlgorithm interface. This interface has 3 methods.

The most important method is the *Executable choose(Executable… readyTasks)* method, that takes an array of tasks that can be scheduled next and choose one based on the implemented algorithm. The second method is *boolean usesPriority()*, a method that returns true if the algorithm needs to calculate some kind of priority measure for the tasks before execution can start for a schedule. The third and final method is the *void calculatePriorities(Schedule schedule)* method, this method usually stores the results in the executionWeight field of an Executable, or internally in the algorithm implementation.

```
public interface SchedulingAlgorithm {
	Executable choose(Executable... readyTasks);
	boolean usesPriority();
	void calculatePriorities(Schedule schedule);
}
```

Listing 6. The SchedulingAlgorithm interface

Scheduler

The class that binds everything together is the Scheduler. The scheduler takes a schedule, creates the thread pool, manages the life of the threads, gets the ready tasks from the schedule and passes them to the scheduling algorithm to choose the next task, and notifies the schedule when a task finishes or aborts.

```
public class Scheduler{
	// Constructors
	public Scheduler(SchedulingAlgorithm algorithm)
	public Scheduler(SchedulingAlgorithm algorithm, int maxParallelTasks)
	// Public methods
	public void execute(Schedule schedule)
	public synchronized void notifyDone(Executable task)
	public synchronized void notifyError(List<String> errors)
```

}

Listing 7. The methods of the Scheduler class

The scheduler can be bounded, or not. When unbounded the scheduler attempts to schedule all the ready tasks, the underlying JVM is responsible for actually scheduling the threads to the CPUs. If there are enough available processors, then all of the ready tasks will be scheduled. The execute method starts by checking for cyclic dependencies in the schedule, then if the scheduling algorithm needs task priorities are calculated. An executor service is then started and then the scheduler schedules the ready Executable runnables to it with a synchronization lock. After all the ready tasks are scheduled, the lock is released and the thread of the executor puts itself to sleep. When an executable is done, it calls the *notifyDone* method of the scheduler, which is used with synchronized access and wakes up the scheduler to check if new ready tasks exist. When the schedule is done, the executor service is shutdown and the results are ready.

Implemented scheduling algorithms

Dummy scheduling

Dummy scheduling or list scheduling as properly in research, is an algorithm that takes a listed ordering of all the tasks, and out of the subset of ready tasks chooses the one that has the lowest index in the listed ordering. When using unbounded scheduling this algorithm has the best performance, because it doesn't do any work choosing the next task, and all the ready tasks are scheduled.

Highest Levels First with Estimated Times (HLFET)

This algorithm assigns each task/node in the computation DAG a level, or priority for scheduling. The level of node n_i is defined as the largest sum of execution times along any directed path from n_i to an end node of the graph, over all end nodes of the graph. Since these levels remain constant for the entire scheduling duration, we refer to these levels as being static, and denote the static level of a node ni as $SL(n_i)$. The list scheduling algorithm is then invoked using *SL* as priority measure.

The SL(ni) measure is calculated before execution is started, in the calculatePriorities method of the HLFETSchedulingAlgorithm class, called at the beginning of the execute method in the Scheduler. The calculation of the static levels (*SL*) is done by running a DFS algorithm from each of the starting nodes (nodes with inDegree of 0), and setting the *SL* of n_i as a sum of the largest *SL* of the children (following neighbours) of n_i plus the execution time of n_i.

Modified Critical-Path (MCP)

The Modified Critical-Path (MCP) algorithm is designed based on an attribute called latest possible start time of a task (strand/node in the computation DAG). A task's latest possible start time is determined through the as-late-as-possible (ALAP) binding, which is

done by traversing the task graph upward from the exit nodes to the entry nodes and by pulling the nodes downwards as much as possible constrained by the length of the critical path (CP).

The MCP algorithm first computes all the latest possible start times for all nodes. Then, each node is associated with a list of latest possible start times which consists of the latest possible start time of the node itself, followed by a decreasing order of the latest possible start times of its children nodes. The MCP algorithm then constructs a list of nodes in an increasing lexicographical order of the latest possible start times lists. At each scheduling step, the first node is removed from the list and scheduled to a processor that allows for the earliest start time.

Mobility Directed (MD)

The Mobility Directed (MD) algorithm selects a node at each step for scheduling based on an attribute called the relative mobility. Mobility of a node is defined as the difference between a node's earliest start time and latest start time. Similar to the ALAP binding in MCP, the earliest possible start time is assigned to each node through the as-soon-as-possible (ASAP) binding which is done by traversing the task graph downward from the entry nodes to the exit nodes and by pulling the nodes upward as much as possible. Relative mobility is obtained by dividing the mobility with the node's computation cost.

Essentially, a node with zero mobility is a node on the CP. At each step, the MPD algorithm schedules the node with the smallest mobility to the first processor which has a large enough time slot to accommodate the node without considering the minimization of the node's start time. After a node is scheduled, all the relative mobilities are updated.

User Authentication workflow example

A hypothetical web application server receives a login/authentication request, the request contains a username and a password hash, the server must check the credentials, and if they are ok, generate an authentication token for future requests, prepare the response html, populate it with info about the user taken from the database, and return a response with the html and the token. The task dependency graph looks something like Figure 1.

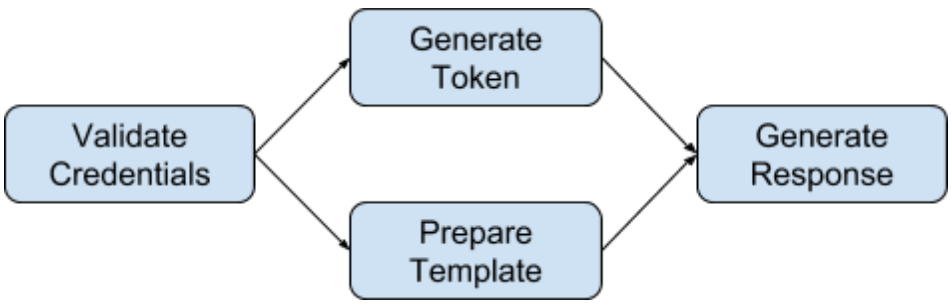

Figure 1. User authentication workflow

Implementation

```
public class ValidateCredentials extends Executable {

	public ValidateCredentials(String id) {
		super(id);
	}

	@Override
	public void execute() {
		List<? extends Object> input = this.get("credentials");
		if (input == null || input.size() != 2) {
			error("Wrong input");
			return;
		}

		//Simulate checking of credentials in database
		Thread.sleep(1000);

		// Pass username to PrepareTemplate and GenerateToken
		produce("username", input.get(0));
	}
}

public class CreateToken extends Executable {
	protected CreateToken(String id) {
		super(id);
	}

	@Override
	public void execute() {
		String userName = get("username").get(0).toString();
		int token = userName.hashCode();
		produce("token", token);
	}
}

public class PrepareTemplate extends Executable {
	public PrepareTemplate(String id) {
		super(id);
	}

	@Override
	public void execute() {
		//Simulate workload of creating html
		String userName = (String) get("username").get(0);
		Thread.sleep(500);
		produce("template",
		String.format(
			"<html>something something %s</html>", userName)
```

```
            );
      }
}

public class GenerateResponse extends Executable {

      protected GenerateResponse(String id) {
            super(id);
      }

      @Override
      public void execute() {
            String template = get("template").get(0).toString();
            int token = (int) get("token").get(0);
            Response r = new Response();
            r.header = template;
            r.body = Integer.toString(token);
            produce("response", r);
      }
}

class Response {
      String header;
      String body;
}

public class LogInSchedule extends Schedule {
      public LogInSchedule(String userName, String passhash) {
            Executable cred = new ValidateCredentials("c.credentials")
                  .addInput("credentials", userName, passhash);
            Executable prep = new PrepareTemplate("prep. template");
            Executable token = new CreateToken("cr. token");
            Executable response = new GenerateResponse("gen. response");
            this.add(cred)
                  .add(prep, cred)
                  .add(token, cred)
                  .add(response, prep, token);
      }
}

void main(String… args){
      Schedule schedule = new LogInSchedule("some user","pass hash");
      Scheduler s = new Scheduler(new DummySchedulingAlgorithm());
      s.execute(schedule);
      logger.info("Done!");
      logger.info(schedule.getResults().toString());
}
```

Source code

The complete source code of the project is available at GitHub as a public repository that anyone can clone or fork. The project is a gradle project, and all that is required to build the library in a jar archive is to invoke the gradle jar command. And then just include the built jar as dependency in another project.

Link to the project: https://github.com/delix/DAG_Task_Scheduler