Concurrency in JavaFX

Tecniche di Programmazione – A.A. 2014/2015
Summary

1. The problem
2. javafx.concurrent package
The problem

Concurrency in JavaFX
UI Responsiveness

- JavaFX is single threaded
- The JavaFX application class manages the interaction
  - User <-> User interface elements
- All events are managed by a single (synchronous) queue, and processed one by one
- When an event handler takes a long time, the UI gets “stuck” until the event handler terminates
Single Threaded behavior

- Event
- User action
- Queue
- Process event

The Scene Graph

Many, asynchronous

One at a time, synchronous

System defined behavior

User defined behavior (my handlers)
The solution

- Long-running computations should not block the user interface
- Need to be run in a separate (parallel) thread

**Warning:** different threads cannot modify the same variables, otherwise “race conditions” may occur
Single Threaded behavior

The Scene Graph

Many, asynchronous

User action

Queue

Event

Process event

One at a time, synchronous

JavaFX Thread

Long-running event handler

Background Thread

Execute in parallel

Start and return immediately

Read/Modify

Finish
Interacting from the background

- The parallel thread can not interact with any object created in the foreground (JavaFX) thread
  - How can the application know about the progress of the background state (is it 10% or 98%?) Or whether it is terminated?
  - How can the background method modify some elements in the scene graph?
  - How can the background method modify some values in the Model?
Single Threaded behavior

The Scene Graph

User action

Queue

Event

Process event

Update operations

Many, asynchronous

Long-running event handler

Start and return immediately

Schedule for later

Visible

JavaFX Thread

Background Thread

Update (thread safe)

Finish

Read/Modify

Progress properties

Update operations

Finish
The solution: updating progress

- The background thread has some “shared” state variables
  - Progress (in % and absolute terms)
  - Message
- Progress variables can be updated anytime
- The JavaFX thread can
  - Read these variable in event handlers
  - “Bind” these variables directly to some UI elements
The solution: scheduling operations

- The background thread may schedule some operations to be run “later” in the JavaFX thread
- Such operations are injected in the main JavaFX event queue
- When executed (“later”), they run on the main JavaFX thread, and they can access any object (Model or Scene Graph)
The solution: reacting to thread state

- The main application may monitor the background thread, and may register to state-change events
  - E.g., be notified when the thread is finished
- The main thread may read the “value” computed by the background object
javafx.concurrent package

Concurrency in JavaFX
Main classes

- **Task**: to be executed only once
- **Service**: to be re-used many times
- `<V>` the class type of the returned value
  - `.getValue()` to read it
  - `<Void>` if no value to return
States of a Task

- READY
- SCHEDULED
- RUNNING
- CANCELLED
- FAILED
- SUCCEEDED

Transitions:
- new
- start()
- call()
- cancel()
- exceptions
- return
Implementing a Task

- **Sub-class from Task<MyValue>**
  - Anonymous class or explicit class
- **Override call() method**
  - Do computations
  - Upon completion, return an object of type MyValue
  - Constantly monitor for isCancelled() property
  - Constantly update the progress properties
- If needed, schedule UI updates
Example

```java
import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {

    @Override protected Integer call() throws Exception {
        int iterations;
        for (iterations = 0; iterations < 100000; iterations++) {
            if (isCancelled()) {
                break;
            }
            System.out.println("Iteration " + iterations);
        }
        return iterations;
    }
};
```
Progress properties

- **double progress**
  - current progress of this Worker in terms of percent complete
  - Value between 0.0 and 1.0
  - If cannot be determined, set to -1.0 (default)

- **double totalWork**
  - Amount of ‘work’ that is expected to be done
  - Value between 0.0 and MAX_DOUBLE
  - If cannot be determined, set to -1.0 (default)

- **double workDone**
  - Current amount of ‘work’ already done
  - Value between 0.0 and totalWork
  - If cannot be determined, set to -1.0 (default)
Progress properties

- **double progress**
  - current progress of this Worker in terms of percent complete
  - Value between 0.0 and 1.0
  - If cannot be determined, set to -1.0 (default)

- **double totalWork**
  - Amount of 'work' that is expected to be done
  - Value between 0.0 and MAX_DOUBLE
  - If cannot be determined, set to -1.0 (default)

- **double workDone**
  - Current amount of 'work' already done
  - Value between 0.0 and totalWork
  - If cannot be determined, set to -1.0 (default)

All three properties are updated, in a thread-safe way, by a call to

```java
protected void updateProgress(double workDone, double totalWork)
```

Tecniche di programmazione  A.A. 2014/2015
Message property

- String message
  - Any message that the task may update
  - Usually, to be displayed in some part of the UI
- Updated by means of a call to
  - protected void updateMessage(String message)
import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {

    @Override protected Integer call() throws Exception {
        int iterations;
        for (iterations = 0; iterations < 100000; iterations++) {
            if (isCancelled()) {
                break;
            }
            updateMessage("Iteration " + iterations);
            updateProgress(iterations, 1000);
        }
        return iterations;
    }
};
import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {
    @Override protected Integer call() throws Exception {
        ...
    }
};

Thread th = new Thread(task);

th.setDaemon(true);

th.start();
Running a task

import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {

    @Override protected Integer call() throws Exception {
        ...
    }

};

Thread th = new Thread(task);
th.setDaemon(true);
th.start();

If you want a background thread to prevent the VM from existing after the last stage is closed, then you would want daemon to be false.

If you want the background threads to simply terminate after all the stages are closed, then you must set daemon to true.
Properties of the Task can be “bound” to properties of the UI

- Node.property.bind(task.property)
- progressProperty
- runningProperty

Boolean, e.g., for visibility of progress bar

```java
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty());
new Thread(task).start();
```
Updating the GUI state

  - Thread-safe violations may result in run time exceptions or in unpredictable behavior of the program

- Schedule updates to be executed in the main thread
  - Platform.runLater(Runnable r)
  - The method r.run() is scheduled to be executed in the JavaFX Thread

- Try to limit the number of scheduled updates, to avoid overloading the front task
@Override protected Customer call() throws Exception {
    // pseudo-code:
    // query the database
    // read the values

    // Now update the customer
    Platform.runLater(new Runnable() {
        @Override public void run() {
            customer.setFsetFirstName(rs.getString("FirstName"));
            // etc
        }
    });

    return customer;
}
Initializing the task

- It is **strongly encouraged** that all Tasks be initialized with immutable state upon which the Task will operate.
- This should be done by providing a Task constructor which takes the parameters necessary for execution of the Task.
- For an inline implementation, we may use final variables in the enclosing context.
- For a full-class implementation, we may use a constructor that stores the parameters into final variables.
Example (inline)

```java
final int totalIterations = 9000000;

Task<Integer> task = new Task<Integer>() {
    @Override protected Integer call() throws Exception {
        int iterations;
        for (iterations = 0; iterations < totalIterations;
                iterations++) {
            if (isCancelled()) {
                updateMessage("Cancelled");
                break;
            }
            updateMessage("Iteration " + iterations);
            updateProgress(iterations, totalIterations);
        }
        return iterations;
    }
};
```
Example (full class)

```java
public class IteratingTask extends Task<Integer> {
    private final int totalIterations;

    public IteratingTask(int totalIterations) {
        this.totalIterations = totalIterations;
    }

    @Override protected Integer call() throws Exception {
        int iterations = 0;
        for (iterations = 0; iterations < totalIterations; iterations++) {
            if (isCancelled()) {
                updateMessage("Cancelled"); break;
            }
            updateMessage("Iteration " + iterations);
            updateProgress(iterations, totalIterations);
        }
        return iterations;
    }

    IteratingTask task = new IteratingTask(8000000);
```
Warning

- Do not pass mutable state to a Task and then operate on it from a background thread.
- Doing so may introduce race conditions.
- It is possible that both the Task and some other application code will be reading or modifying the state of the Model from different threads: there could be a violation of threading rules!
- For such cases, modify the Model object from the FX Application Thread rather than from the background thread (with Platform.runLater)
Get final value

- The `call()` method in `Task<V>` returns a `<V>` object.
- The object is stored in the `value` property of the task: `task.getValue()` returns the `<V>` object.
  - But only after the method has returned.
- We must set an `EventHandler` on the transition to the `SUCCEEDED` state.
  - `task.setOnSucceeded(EventHandler eh)`
Example

myTask.setOnSucceeded(
    new EventHandler<WorkerStateEvent>() {
        @Override public void handle(WorkerStateEvent t) {
            // use .getValue()
            listView.setItems(myTask.getValue());
        }
    });

Returning partial results

- Updating the Return Value before the task is finished
  - Call `updateValue(Object)`
  - This method may be called repeatedly from the background thread.
  - Updates are coalesced to prevent saturation of the FX event queue.
  - This means you can call it as frequently as you like from the background thread but only the most recent set is ultimately set.
Sharing information

- We may expose a new property on the Task which will represent the partial result
- Always modify it in a runLater() call
Resources

Concurrency in JavaFX
Resources

- Concurrency in JavaFX,
  https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm

- Worker Threading in JavaFX 2.0,

- JavaDoc for class javafx.concurrent.Task,
  https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/Task.html

Licenza d’uso

- Queste diapositive sono distribuite con licenza Creative Commons “Attribuzione - Non commerciale - Condividi allo stesso modo (CC BY-NC-SA)”

- Sei libero:
  - di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
  - di modificare quest'opera

- Alle seguenti condizioni:
  - **Attribuzione** — Devi attribuire la paternità dell'opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l'opera.
  - **Non commerciale** — Non puoi usare quest'opera per fini commerciali.
  - **Condividi allo stesso modo** — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con una licenza identica o equivalente a questa.

- [http://creativecommons.org/licenses/by-nc-sa/3.0/](http://creativecommons.org/licenses/by-nc-sa/3.0/)