Механика SwingWorker

Данис Тазеев

Описание можно не читать, а сразу переходить к выводам.

Описание

Поле `doSubmit` — это экземпляр внутреннего для `SwingWorker.DoSubmitAccumulativeRunnable`. Единственный экземпляр этого класса разделяется между всеми экземплярами класса `SwingWorker`.

Конструктор `SwingWorker` создаёт экзкмпляр, унаследованный от `FutureTask`, и сохраняет его в `final` поле `future`.

Все экземпляры класса `SwingWorker` используют единственный, разделяемый между всеми ими, экземпляр `ExecutorService`. Это `ThreadPoolExecutor` с 10 `core` потоками-демонами и с очередью заданий неограниченного размера.

`SwingWorker.execute()` помещает себя в `ExecutorService`. Это возможно, так как `SwingWorker` реализует `RunnableFuture` (`RunnableFuture` — интерйес; `FutureTask` — класс, реализующий `RunnableFuture`).

Далее `ExecutorService` вызывает `SwingWorker.run()`, который просто делегирует выполнение в `future` — `future.run()`, и больше ничего не делает. Как мы помним, `future` — это `final` поле с созданным в конструкторе `SwingWorker` наследником `FutureTask`.

`SwingWorker.cancel()` просто вызывает `future.cancel()`.
`SwingWorker.isCancelled()` просто вызывает `future.isCancelled()`.
`SwingWorker.get()` просто вызывает `future.get()`.

Как извесно, когда `FutureTask` переходит в финальное состояние (`NORMAL`, `EXCEPTIONAL` или `CANCELLED`), `FutureTask` вызывает свой `protected` метод `done()` в том же потоке, в котором `FutureTask` перешёл в финальное состояние:

- если финальное состояние `NORMAL` или `EXCEPTIONAL`, то `done()` вызывается в том же потоке, в котром выполнялся `run()`;
- если финальное состояние `CANCELLED`, то `done()` вызывается в том же потоке, в ктором был вызван `cancel()`.

В моей программе `cancel()` всегда вызывается из `EDT`.

Наследник `FutureTask`, созданный в конструкторе `SwingWorker` и сохранённый в `final` поле `future`, переопределяет метод `done()`. В нём он вызывает `private` метод `SwingWorker.doneEDT()`. `doneEDT()` делает следующее:

- если `doneEDT()` вызван в `EDT` (через `cancel()`), то `doneEDT()` просто вызывает `protected` метод `SwingWorker.done()`;
- если метод `doneEDT()` вызван из одного из потокв `ExecutorService`, выполнивший `protected` метод `SwingWorker.doInBackground()`, то `doneEDT()` добавляет

отложенный вызов `SwingWorker.done()` в `DoSubmitAccumulativeRunnable` вызовом `add()`.

Класс `DoSubmitAccumulativeRunnable` — наследник абстрактого класса `AccumulativeRunnable`, реализующего `Runnable`.

`AccumulativeRunnable` работает так:

- при добавлении первого задания `AccumulativeRunnable` помещает себя в `EventQueue`, а затем, уже находясь в `EventQueue`, продолжает накапливать задания, поступающие через метод `add()`;
- когда `EDT` добирается до `AccumulativeRunnable`, находящемся в `EventQueue`, то все накопленные задания передаются в абстрактный `protected` метод `run(List)`;
- одновременно с передачей заданий в `run(List)` их список, накапливавшийся, пока `AccumulativeRunnable` находился в `EventQueue`, очищается;
- теперь тот же самый экземпляр `AccumulativeRunnable` снова поместит себя в `EventQueue`, как только в него будет добавлено следующее задание, и цикл начнётся заново.

Другими словами, `AccumulativeRunnable` накапливает задания, пока находится в `EventQueue`. А когда `EDT` извлекает его из `EventQueue`, все накопленные задания пачкой передаются на исполнение в `run(List)`.

`DoSubmitAccumulativeRunnable`, являясь наследником `AccunulativeRunnable`, реализует обработку пачки накопленных заданий (метод `run(List)`) и переопределяет действие, предпринимаемое при добавлении первого задания.

Если `AccumulativeRunnable` при добавлении первого задания помещает себя в `EventQueue`, то `DoSumbitAccumulativeRunnable` запускает таймер, накапливающий задания в течение 1/30 секунды (как минимум), после чего накопленные задания из `AccumulativeRunnable` извлекаются и передаются на пакетную обработку. И цикл накопления повторяется заново.

Другими словами, если в момент добавления первого задания в `AccumulativeRunnable` `EventQueue` пуста, то оно тут же будет выполнено. `DoSubmitAccumulativeRunnable` явно длает так, что задания накапливаются в течение как минимум 1/30 секунды, и только потом попадают в `EventQueue`.

Пакетная обработка заданий в `DoSubmitAccumulativeRunnable` — просто последовательное их выполнение. В том же порядке, в котором они были добавлены.

Выводы

Если вид (view) запускает `SwingWorker` и позволяет не дожидаться его результатов, то есть отбрасывать их до того, как `SwingWorker` отработал, то код может выглядеть так:

```
class View extends JPanel {
  private final JButton doWork = …;
  private SwingWorker worker;

  View() {
```

```
        doWork.addActionListener(e -> launchWorker());
    }

    private void launchWorker() {
        if (worker != null)
            worker.cancel(false);
        worker = new SwingWorker() {
            @Override
            protected Object doInBackground() throws Exception {
                // выполняем фоновую работу; возвращаем результат
            }

            @Override
            protected void done() {
                if (worker == this) {
                    try {
                        if (!isCancelled())
                            showWorkResult(get());
                    } catch (Throwable err) {
                        // обрабатываем исключение,
                        // случившееся в doInBackground()
                    } finally {
                        worker = null;
                    }
                }
            }
        };
        worker.execute();
    }
}
```

Наибольший интерес представляет метод `done()`. Контракт гарантирует:

- что `done()` вызывается только после того, как `SwingWorker` перешёл в финальное состояние;
- `done()` вызывается исключительно в EDT.

Если `SwingWorker`, выполнив `doInBackground()`, перешёл в финальное состояние, то в `EventQueue` помещается вызыватель `done()`. Это «штатный режим». Контракт соблюдён

Если `SwingWorker` перешёл в финальное состояние из-за успешного `cancel()`, то `done()` будет вызван **изнутри** этого `cancel()`, так как `cancel()` вызывается в EDT. Контракт соблюдён.

Как работает `launchWorker()`? Сначала предпринимается попытка `cancel()`. Если она переводит `worker` в финальное состояние, то `done()` вызывается изнутри `cancel()`, условие `worker == this` выполняется, и в конце `worker` обнуляется. Если `worker` на момент вызова `cancel()` успел перейти в финальном состояние, то `cancel()` не сработает.

Может возникнуть мысль: если `cancel()` не сработал, так как `worker` — в финальном состоянии, значит, `done()` уже выполнен (ведь `SwingWorker` уже перешёл в финальное состояние). Эта мысль ошибочная, так как вызыватель `done()` ещё находится в `EventQueue`, а EDT занят выполнением `launchWorker()`.

Итак, `cancel()` не сработал, `done()` тоже пока не выполнен. Создаётся и запускается новый `worker`. После этого EDT извлекает из `EventQueue` запускателя `done()`. И вот тут нужна проверка внутри `done()`: `if (worker == this)`. Условие внутри неё окажется ложным, так как EDT перед тем, как выполнять `done()`, выполнил `launchWorker()`, создавший новый `worker`.

Зачем нужно обнулять `worker` в `finally`? Чтобы не хранить отработавший `worker`.