Voll daneben!

Nebenläufigkeit und Parallelität unter Java

Michael Wiedeking · Java Forum Stuttgart · 13. Juli 2023

- Ofen vorheizen
- Zutaten mischen
- Formen
- Auf’s Backblech legen
- Backen
- Deko anrühren
- Deko aufstreichen
- Ruhen lassen

- Ofen vorheizen
- Backen
- Deko aufstreichen
- Ruhen lassen

- Zutaten mischen
- Formen
- Auf’s Backblech legen
- Deko anrühren

```
public class Task extends Thread {

    public Task(…) {
        …
    }

    @Override
    public void run() {
        // Die zu erledigende Aufgabe.
    }

}

var task = new Task(…);
task.start()
```

```
Thread thread = new Thread(

    new Runnable() {

        @Override
        public void run() {
            // Die zu erledigende Aufgabe.
        }

    }

);

thread.start();
```

```
var thread = new Thread(
    () –> {
        // Die zu erledigende Aufgabe.
    }
);
thread.start();
```

```
var builder = Thread.ofPlatform();

var thread = builder.unstarted(

    () -> {

        // Die zu erledigende Aufgabe.

    }

);

thread.start();
```

```
var builder = Thread.ofPlatform();

var thread = builder.start(

    () -> {

        // Die zu erledigende Aufgabe.

    }

);
```

var *t1* = **new** Task1();

var *t2* = **new** Task2();

t1.start();

t2.start();

t1.join();

t2.join();

// Die Ergebnisse von sowohl *t1* als auch *t2* stehen nun zur Verfügung.

…

```
class BlinkingLabel extends JLabel {

    public BlinkingLabel(String label, Duration blink) {

        super(label);

        installBlinker(blink);

    }

    private void toggleColor() {
        …
    }

    …

}
```

```
private void installBlinker(Duration blink) {
    new Thread(() –> {
        while (true) {
            Thread.sleep(blink);
            toggleColor();
        }
    }).start();
}
```

```
private void installBlinker(Duration blink) {

    new Thread(() -> {

        while (true) {

            try {
                Thread.sleep(blink);
            } catch (InterruptedException e) {
                break;
            }

            toggleColor();

        }

    }).start();

}
```

t.interrupt();

assert *t*.isInterrupted();

assert Thread.interrupted();

assert !Thread.interrupted(); // Wenn nicht …

```
private void installBlinker(Duration blink) {

    new Thread(() -> {

        while (true) {

            try {
                Thread.sleep(blink);
            } catch (InterruptedException e) {
                break;
            }

            toggleColor();

        }

    }).start();

}
```

```
private void toggleColor() {

    var c = nextColor();

    setBackground(c);

}
```

```
private void toggleColor() {
    var color = nextColor();
    SwingUtilities.invokeLater(() -> setBackground(color));
}
```

var *t1* = **new** Task1();

var *t2* = **new** Task2();

t1.start();

t2.start();

t1.join();

t2.join();

// Die Ergebnisse von sowohl *t1* als auch *t2* stehen nun zur Verfügung.

…

```
var t1 = new WeatherForecastTask(location, date);

var t2 = new TrafficForecastTask(route, date);

t1.start();

t2.start();

t1.join();

t2.join();

// Die Vorhersage sowohl vom Wetter als auch vom Verkehr stehen nun zur Verfügung.

…
```

```
ExecutorService executor = Executors.newXxx(…);

Future<WeatherForecast> weather =  executor.submit(
    () –> getWeatherForecast(location, date)
);

Future<TrafficForecast> traffic = executor.submit(
    () –> getTrafficForecast(route, date)
);

… planVacation(weather.get(), traffic.get()) …
```

```
ExecutorService executor = Executors.newCachedThreadPool();

Future<WeatherForecast> weather =  executor.submit(
    () –> getWeatherForecast(location, date)
);

Future<TrafficForecast> traffic = executor.submit(
    () –> getTrafficForecast(route, date)
);

… planVacation(weather.get(), traffic.get()) …
```

```
ExecutorService executor = Executors.newFixedThreadPool(2);

Future<WeatherForecast> weather =  executor.submit(
    () –> getWeatherForecast(location, date)
);

Future<TrafficForecast> traffic = executor.submit(
    () –> getTrafficForecast(route, date)
);

… planVacation(weather.get(), traffic.get()) …
```

```
ExecutorService executor = Executors.newSingleThreadExecutor();

Future<WeatherForecast> weather =  executor.submit(
    () –> getWeatherForecast(location, date)
);

Future<TrafficForecast> traffic = executor.submit(
    () –> getTrafficForecast(route, date)
);

… planVacation(weather.get(), traffic.get()) …
```

```
ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();

Future<WeatherForecast> weather =  executor.submit(
    () –> getWeatherForecast(location, date)
);

Future<TrafficForecast> traffic = executor.submit(
    () –> getTrafficForecast(route, date)
);

… planVacation(weather.get(), traffic.get()) …
```

```
ExecutorService executor = Executors.newThreadPerTaskExecutor(threadFactory);

Future<WeatherForecast> weather =  executor.submit(
    () -> getWeatherForecast(location, date)
);

Future<TrafficForecast> traffic = executor.submit(
    () -> getTrafficForecast(route, date)
);

… planVacation(weather.get(), traffic.get()) …
```

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks);

<T> List<Future<T>> invokeAll(
    Collection<? extends Callable<T>> tasks,
    long timeout,
    TimeUnit unit
);

<T> List<Future<T>> invokeAny(Collection<? extends Callable<T>> tasks);

<T> List<Future<T>> invokeAny(
    Collection<? extends Callable<T>> tasks,
    long timeout,
    TimeUnit unit
);
```

Future<V>

boolean cancel(boolean *mayInterruptIfRunning*)

default Throwable exceptionNow()

V get()

V get(long *timeout*, TimeUnit *unit*)

boolean isCancelled()

boolean isDone()

default V resultNow()

default Future.State state()

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

    Supplier<WeatherForecast> weather = scope.fork(
        () -> getWeatherForecast(location, date)
    );
    Supplier<TrafficForecast> traffic = scope.fork(
        () -> getTrafficForecast(route, date)
    );

    scope.join();                // Beide Tasks sind fertig …
    scope.throwIfFailed();   // … und etwaige Fehler werden weitergereicht.

    … planVacation(weather.get(), traffic.get()) …

}
```

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

    Supplier<WeatherForecast> weather = scope.fork(
        () -> getWeatherForecast(location, date)
    );
    Supplier<TrafficForecast> traffic = scope.fork(
        () -> getTrafficForecast(route, date)
    );

    scope.join();                  // Beide Tasks sind fertig …
    scope.throwIfFailed(e -> new VacationException(e));

    … planVacation(weather.get(), traffic.get()) …

}
```

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

    Future<WeatherForecast> weather = scope.fork(
        () -> getWeatherForecast(location, date)
    );
    Future<TrafficForecast> traffic = scope.fork(
        () -> getTrafficForecast(route, date)
    );

    scope.join();                // Beide Tasks sind fertig …
    scope.throwIfFailed();       // … und etwaige Fehler werden weitergereicht.

    … planVacation(weather.get(), traffic.get()) …

}
```

```
private WeatherForecast getWeatherForecast() throws ExecutionException {

    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<WeatherForecast>()) {

        Supplier<WeatherForecast> weather1 = scope.fork(
            () -> getWeatherForecast1(location, date)
        );

        Supplier<WeatherForecast> weather2 = scope.fork(
            () -> getWeatherForecast2(location, date)
        );


        scope.join();
        return scope.result();

    }

}
```

```
private WeatherForecast getWeatherForecast() throws ExecutionException {

    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<WeatherForecast>()) {

        scope.fork(
            () -> getWeatherForecast1(location, date)
        );

        scope.fork(
            () -> getWeatherForecast2(location, date)
        );


        scope.join();
        return scope.result();

    }

}
```

```
private WeatherForecast getWeatherForecast() throws ExecutionException {
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<WeatherForecast>()) {
        List<Callable<WeatherForecast>> services = getAvailableWeatherServices();
        services.stream().forEach(scope::fork);
        scope.join();
        return scope.result();
    }
}
```

```
private WeatherForecast getWeatherForecast() throws ExecutionException {

    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<WeatherForecast>()) {

        List<Callable<WeatherForecast>> services = getAvailableWeatherServices();

        for (var s : services) {
            scope.fork(s);
        }

        scope.join();

        return scope.result();

    }

}
```

```
public class StructuredTaskScope<T> implements AutoCloseable

    <U extends T> Subtask<U> fork(Callable<? extends U> task)

    void shutdown()

    StructuredTaskScope<T> join() throws InterruptedException

    StructuredTaskScope<T> joinUntil(Instant deadline)
                                                    throws InterruptedException, TimeoutException

    void close();

    void handleComplete(Subtask<? extends T> handle)

    final void ensureOwnerAndJoined()
```

```
public sealed interface Subtask<T> extends Supplier<T>

    enum State {
        SUCCESS, FAILED, UNAVAILABLE
    }



    public State state();

    public Callable<? extends T> task();

    @Override
    public T get();

    public Throwable exception();
```

```
class MaxResultsScope<T> extends StructuredTaskScope<T> {

    private final List<T> results;
    private final int n;

    …

    @Override
    public MaxResultsScope<T> join() throws InterruptedException {
        super.join();
        return this;
    }

    public List<T> results() {
        super.ensureOwnerAndJoined();
        return results;
    }

}
```

```
public MaxResultsScope(int n) {
    this.results = new ArrayList<>(n);
    this.n = n;
}

@Override
protected void handleComplete(Subtask<? extends T> subtask) {
    if (subtask.state() == Subtask.State.SUCCESS) {
        synchronized (results) {
            if (results.size() < n) {
                results.add(subtask.get());
                if (results.size() == n) {
                    super.shutdown();
                }
            }
        }
    }
}
```

```
public void serve(ServerSocket serverSocket) throws IOException, InterruptedException {
    try (var scope = new StructuredTaskScope<Void>()) {

        try {
            while (true) {
                var socket = serverSocket.accept();
                scope.fork(() -> handle(socket));
            }
        } finally {
            scope.shutdown();
            scope.join();
        }

    }
}
```

```
var builder = Thread.ofVirtual();

var thread = builder.start(

    () -> {

        // Die zu erledigende Aufgabe.

    }

);
```

Thread.Builder

Thread.Builder.OfVirtual　Thread.ofVirtual()

Thread.Builder.OfPlatform　Thread.ofPlatform()

```
Thread thread = Thread.ofVirtual().start(runnable);

Thread thread = Thread.ofVirtual().name("duke").unstarted(runnable);

Thread thread = Thread.startVirtualThread(runnable);
```

```
class Server {

    final static ThreadLocal<Principal> PRINCIPAL = new ThreadLocal<>();

    void serve(Request request, Response response) {

        var level = (request.isAuthorized() ? ADMIN : GUEST);

        var principal = new Principal(level);

        PRINCIPAL.set(principal);

        Application.handle(request, response);

    }

}
```

```
class DBAccess {

    DBConnection open() {

        var principal = Server.PRINCIPAL.get();

        if (!principal.canOpen()) {

            throw new InvalidPrincipalException();

        }

        return newConnection(…);

    }

}
```

```
final static ScopedValue<...> V = ScopedValue.newInstance();


// In einer Methode

ScopedValue.where(V, <value>).run(() -> {

    ... V.get() ...

});
```

```
class Server {
    final static ScopedValue<Principal> PRINCIPAL = ScopedValue.newInstance();

    void serve(Request request, Response response) {

        var level = (request.isAdmin() ? ADMIN : GUEST);

        var principal = new Principal(level);

        ScopedValue.where(PRINCIPAL, principal).run(() ->
            Application.handle(request, response)
        );

    }
}
```

```
class DBAccess {

    DBConnection open() {

        var principal = Server.PRINCIPAL.get();

        if (!principal.canOpen()) {

            throw new InvalidPrincipalException();

        }

        return newConnection(…);

    }

}
```

Fragen!?

Vielen Dank!