

Асинхронные вычисления

Интерфейсы Callable и Future

```
public interface Callable<V>  
{  
    V call() throws Exception;  
}
```

Например

Callable <Integer> - асинхронные вычисления, результат Integer

```
public class MyCallable implements Callable<Integer> {  
    @Override  
    public Integer call() throws Exception {  
        Integer value;  
        ...  
        return value;  
    }  
}
```

Интерфейсы Callable и Future

- интерфейс Future описывает набор методов для работы с результатами асинхронных вычислений

```
public interface Future<V> {  
    V get() throws...  
    V get(long timeout, TimeUnit unit) throws...  
    boolean cancel(boolean mayInterrupt) ;  
    boolean isCancelled();  
    boolean isDone();  
  
}
```

Методы

- `V get()` throws...
- `V get(long timeout, TimeUnit unit)` throws...

Возвращает результат асинхронных вычислений, выполнение блокируется до окончания вычислений или до истечения указанного интервала времени

Методы

– `boolean cancel(boolean mayInterrupt)`

Пытается отменить выполнение задачи.

Если задача уже запущена и параметр `mayInterrupt` равен `true`, она прерывается.

Если вычисления еще не начаты, они не начнутся никогда.

Если отмена задачи прошла успешно. Метод возвращает значение `true`.

Методы

– `boolean isCancelled()`

Возвращает `true`, если задача была отменена до ее нормального завершения

– `boolean isDone()`

Возвращает `true`, если выполнение задачи завершено, если выполнение было прекращено или если в процессе ее выполнения возникло исключение

Класс FutureTask

- Реализует интерфейсы
 - Runnable
 - Future <V>

Использование FutureTask

```
Callable<Integer> myComp = . . . ;  
FutureTask <Integer> task=  
    new FutureTask<Integer>(myComp);  
Thread t = new Thread(task); //Runnable  
t.start();  
. . .  
Integer result = task.get(); //Future
```

Пример (подсчет количества файлов)

```
class MyCounter implements Callable <Integer> {
    MyCounter (File name){
    ...
    }
    ...
    public Integer call(){
        Integer count;
        List<Future<Integer>> results = new ArrayList<>(); //Список для создаваемых FutureTask
        ... // проверяем свойство файла
            // или рекурсивно создаем новый объект MyCounter
        return count;}
    }
```

Пример (подсчет количества файлов)

```
if (file.isDirectory()){
// для каталогов перебираем элементы (elem) и создаем рекурсивно
{
  MyCounter counter =new MyCounter(elem);
  FutureTask <Integer> task= new FutureTask<Integer>(counter);
  results.add(task); //добавили задание для подкаталога
  Thread t = new Thread(task);
  t.start();
}
// формируем результат для каталога
} else { //для файла проверяем условие
  if (???(file)) count++;
}
```

Пример (подсчет количества файлов)

Результаты

- Для подкаталогов в методе call

```
for (Future <Integer> result : results)
```

```
try {
```

```
    count += result.get();
```

```
}
```

```
catch (ExecutionException e) { . . . }
```

Пример (подсчет количества файлов)

```
String directory = . . . ;
MyCounter counter = new MyCounter(new File(directory));
FutureTask<Integer> task = new FutureTask<>(counter);
Thread t = new Thread(task);
t.start();
try {
    System.out.println(task.get() + " matching files.");
} catch (ExecutionException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Интерфейсы

Executor и ExecutorService

- интерфейсы для планирования и управления Runnable объектами
- определено несколько реализаций Executor, которые предлагают различные характеристики планирования. Постановка задачи в очередь к обработчику делается методом execute()

Executor

```
Executor executor = . . . ;  
executor.execute(new RunnableTask1());  
executor.execute(new RunnableTask2());  
...
```

Отдельный поток для каждого задания

```
class ThreadPerTaskExecutor  
    implements Executor {  
    public void execute  
        (Runnable r) {  
        new Thread(r).start();  
    }  
}
```

Запуск задания в том же потоке

```
class DirectExecutor
  implements Executor {
    public void execute
      (Runnable r) {
      r.run();
    }
  }
}
```

ExecutorService

- Интерфейс, наследующий Executor добавляет методы для управления завершением и методы, которые могут создавать Future для отслеживания хода выполнения одной или нескольких асинхронных задач

ExecutorService

- Методы
 - void shutdown()
 - List<Runnable> shutdownNow()
 - <T> Future<T> submit(Callable<T> task)
 - Future<?> submit (Runnable task)
 - <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
 - <T> T invokeAny(Collection<? extends Callable<T>> tasks)

ExecutorService

Интерфейс `ExecutorService` добавляет методы, позволяющие управлять выполняющимися задачами

Метод `submit()` позволяет получить объект `Future`, связанный с выполняемой задачей.

С его помощью можно опросить состояние задачи.

ExecutorService

```
ExecutorService executor = ...
```

```
Callable<String> call = new MyCallable();
```

```
Future<String> fut = executor.submit(call);
```

```
if (fut.isDone()){
```

```
    System.out.println(fut.get());
```

Executors

- Объекты `ExecutorService` создаются через статические методы класса `Executors`
- Цель – уменьшить затраты на запуск нового потока
- Для этого сразу создаются наборы потоков (пулы), которые принимают задания на выполнение
- В пул помещаются объекты `Runnable`
- Потоки из пула запускают метод `run()`
- После завершения метода `run()` поток не удаляется, а остается готовым к выполнению новой задачи

Executors

`Executors.newCachedThreadPool()`

пул потоков немедленно начинает выполнение каждой задачи

новые потоки создаются по мере необходимости

бездействующие потоки удаляются через 60 секунд

Executors

`Executors.newFixedThreadPool(n)`

создает пул потоков фиксированного размера

если число задач превышает размер пула, то новые задачи ставятся в очередь

`Executors.newSingleThreadPool()`

создает пул с одним потоком, который последовательно выполняет задачи

Executors

```
Executors.newScheduledThreadPool()
```

```
Executors.newSingleThreadScheduledExecutor()
```

создает пулы, запускающие задачи по графику

график может предполагать запуск задачи через заданный интервал времени или периодическое выполнение задачи

Executors

Все методы возвращают экземпляр класса `ThreadPoolExecutor`, реализующий интерфейс `ExecutorService`

Передача объектов `Runnable` или `Callable` для их исполнения потоками в пуле осуществляется методом `submit ()`

Для завершения работы пула используется метод `shutdown()`

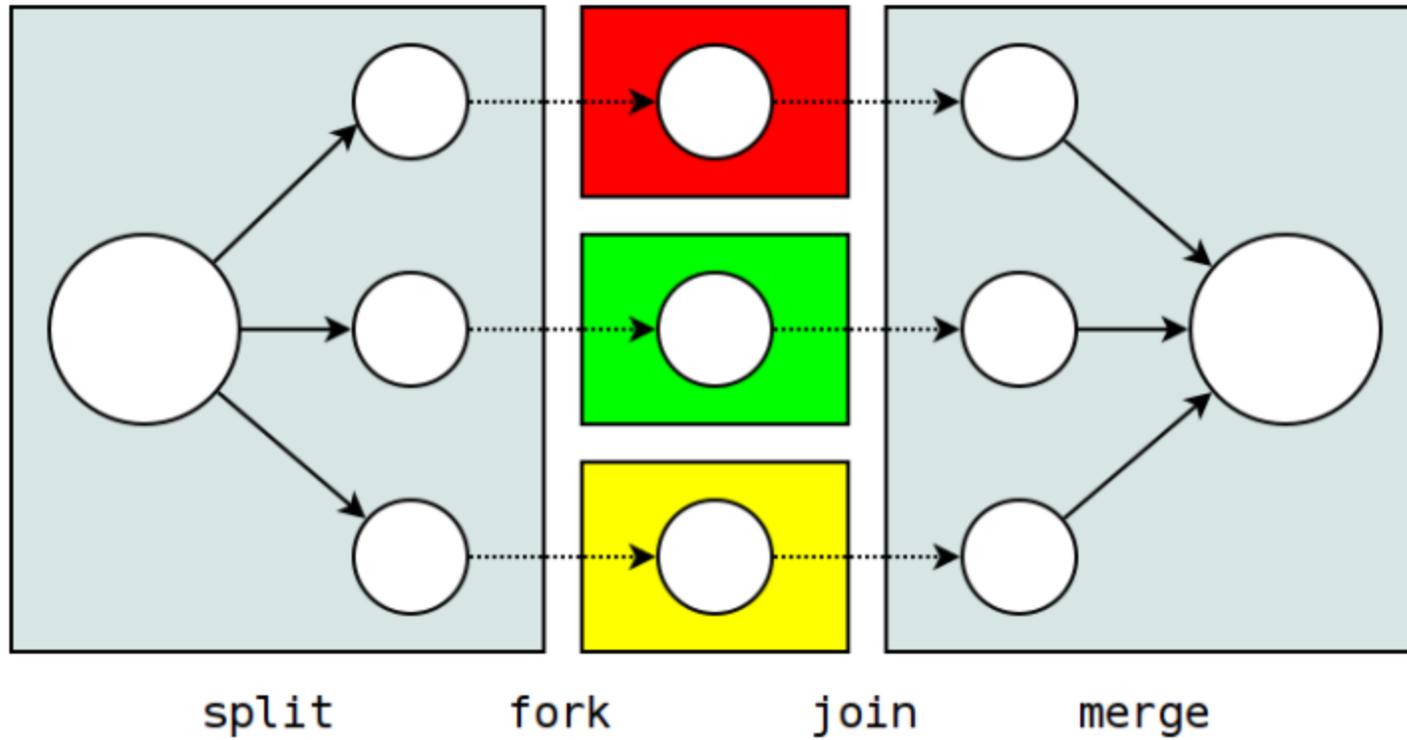
Метод `invokeAll()` позволяет передать коллекцию задач (возможно с установкой времени ожидания). По завершении всех задач получается список будущих действий с их статусом (завершена, прервана, выбросила исключение)

Метод `invokeAny()` возвращает результат первой успешно выполненной задачи в коллекции, остальные задачи отменяются

ForkJoinPool

- реализации ExecutorService для упрощения распараллеливания рекурсивных задач
- позволяет запускать ForkJoinTasks в пуле потоков
- В версии Java 8 включили fork/join framework

Классическая схема



```
ForkJoinPool common = ForkJoinPool.commonPool();  
ForkJoinPool available = new ForkJoinPool ();  
  
ForkJoinPool forkJoin2 = new ForkJoinPool(2);  
// уровень параллелизма 2
```

Выполняемые задачи

- наследники класса ForkJoinTasks
 - RecursiveAction
 - RecursiveTask

Выполняемые задачи

<https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>

if (my portion of the work is small enough)

do the work directly

else

split my work into two pieces invoke the two pieces and wait for the results

Базовая семантика методов

- `fork()`
 - Кладёт задачу в очередь, и возвращается
 - Кто-нибудь другой может эту задачу подхватить
- `join()`
 - Блокируется, пока задача не закончится
 - Но поток терять на этом нельзя!
 - FJP может дать ему что-нибудь выполнить другое

RecursiveTask

Дерево

```
public interface Node {  
    Collection<Node> getChildren();  
    Long getValue();  
}
```

```
public class ValueSumCounter extends RecursiveTask<Long>{
    private final Node node;
    @Override
    protected Long compute() {
        long sum = node.getValue();
        List<ValueSumCounter> subTasks = new LinkedList<>();
        for(Node child : node.getChildren()) {
            ValueSumCounter task = new ValueSumCounter(child);
            task.fork(); // запустим асинхронно
            subTasks.add(task);
        }
        for(ValueSumCounter task : subTasks) {
            sum += task.join();
            // дождёмся выполнения задачи и прибавим результат
        }
        return sum; }
}
```

```
public static void main(String[] args) {  
    Node root = getRootNode();  
    new ForkJoinPool().invoke(new ValueSumCounter(root));  
}
```

Параллельные операции с массивами

Java 8

`Arrays.parallelSort()`

`Arrays.parallelSetAll()`

`Arrays.parallelPrefix()`