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();

    }
```

Интерфейсы
Callable и Future

- Методы интерфейса java.util.concurrent.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> {
. . .
public Integer call(){
   . . .
  return count;}
}
```

Для файлов – проверяем вхождение ключевого слова

Для каталогов – рекурсивно создаем новый объект MyCounter и добавляем его в список FutureTask

По всем подкаталогам суммируем результаты

Пример (подсчет количества файлов)

```
MyCounter counter =new MyCounter(...);
FutureTask <Integer> task=
                    new FutureTask<Integer>(counter);
results.add(task); // для подкаталогов
Thread t = new Thread(task);
t.start();
```

Пример (подсчет количества файлов)

Результаты

- Для подкаталогов

```
for (Future <Integer> result : results)
  try {
        count += result.get();
      }
      catch (ExecutionException e) { . . .  }
```

- Для всей задачи

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

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();
    }
}
```

Большинство реализаций интерфейса Executor определяют некоторые ограничения, когда и как отдельные задачи будут запускаться

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());
```

ExecutorService

Метод shutdown() завершает работу службы, оканчивая выполнение уже принятых задач, но не принимая новых

Метод invokeAny() запускает задачи из коллекции задач и возвращает результат одной из них

Метод invokeAll() выполняет задачи из коллекции и возвращает результаты их выполнения в виде списка

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.newSheduledThreadPool()

Executors.newSingleThreadSheduledExecutor()

создает пулы, запускающие задачи по графику

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

Все методы возвращают экземпляр класса ThreadPoolExecutor, реализующий интерфейс ExecutorService

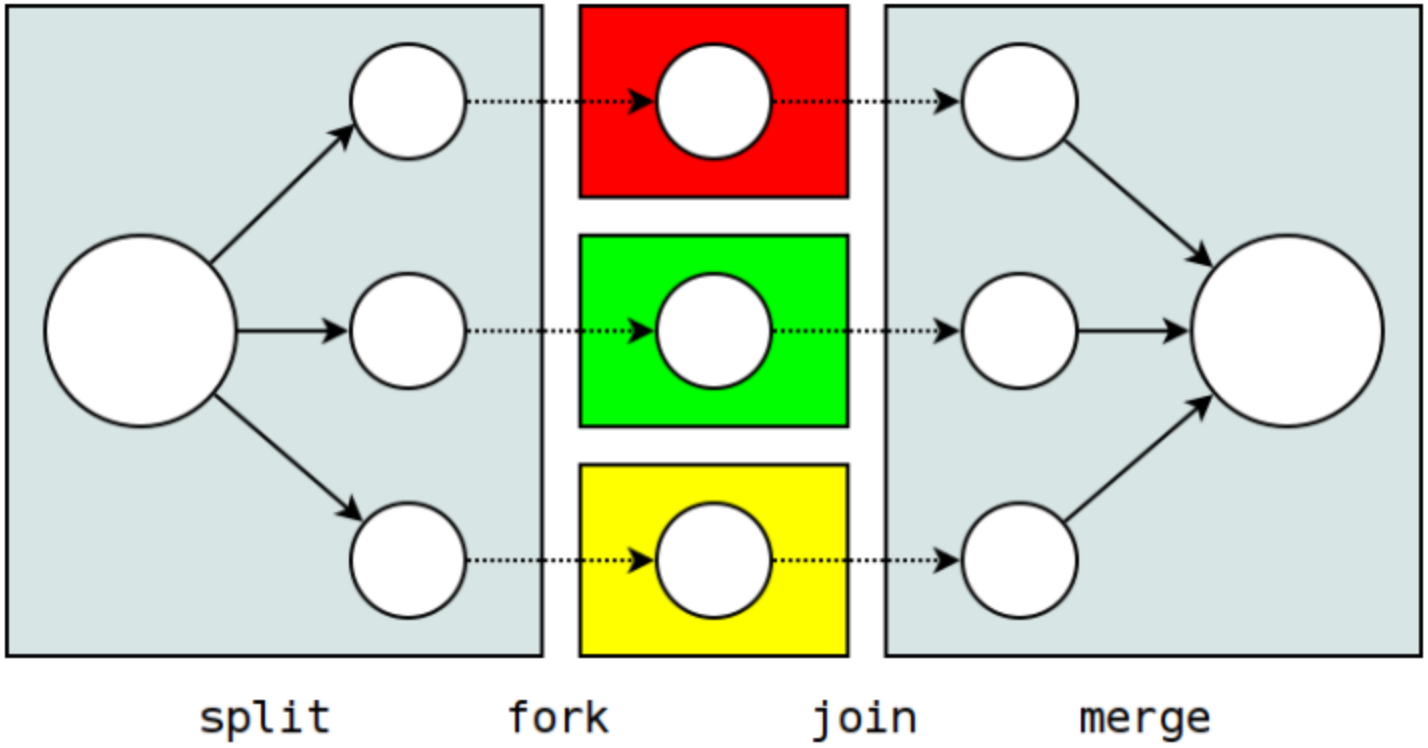
передача объектов Runnable или Callable для их исполнения потоками в пуле осуществляется методом submit ()

для завершения работы пула используется метод shutdown()

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 avialable = 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()