Асинхронные вычисления

Интерфейс Callable

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

Например

Callable <Integer> - асинхронные вычисления, результат Integer

Это функциональный интерфейс, поэтому его можно использовать в качестве целевого объекта для лямбда-выражения или ссылки на метод

Интерфейс Future

java.util.concurrent.Future<V>

V get() throws…
V get(long timeout, TimeUnit unit) throws…
boolean cancel(boolean mayInterrupt)
boolean isCancelled()
boolean isDone()

Класс FutureTask

Реализует интерфейсы

- Runnable
- Future <V>

Пример

```
Callable<Integer> myComp = . . .; //Callable
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( ){. . .}
}
```

Пример (подсчет количества файлов)

```
MyCounter counter =new MyCounter(...);

FutureTask <Integer> task=
                    new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
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.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 ExecutorService добавляет методы, позволяющие управлять выполняющимися задачами

ExecutorService

- submit() позволяет получить объект Future, связанный с выполняемой задачей. С его помощью можно опросить состояние задачи.
- shutdown() завершает работу службы, оканчивая выполнение уже принятых задач, но не принимая новых
- invokeAny() запускает задачи из коллекции задач и возвращает результат одной из них
- invokeAll() выполняет задачи из коллекции и возвращает результаты их выполнения в виде списка

Пулы потоков

- 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

Класс Executors

- Статические методы создания пулов
  - newCachedThreadPool() пул потоков немедленно начинает выполнение каждой задачи, новые потоки создаются по мере необходимости, бездействующие потоки удаляются через 60 секунд
  - newFixedThreadPool() создает пул потоков фиксированного размера, если число задач превышает размер пула, то новые задачи ставятся в очередь
  - newSingleThreadPool() создает пул с одним потоком, который последовательно выполняет задачи

- newSheduledThreadPool()
- newSingleThreadSheduledExecutor()

создают пулы, запускающие задачи по графику

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

ForkJoinPool

- реализации ExecutorService для упрощения распараллеливания рекурсивных задач
- позволяет запускать ForkJoinTasks в пуле потоков

- наследники класса ForkJoinTasks
  - RecursiveAction
  - RecursiveTask

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 повыполнять

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));
}
```