Concurrent интерфейсы Callable, Future

При работе многопоточного приложения часто необходимо получение от потока результата его деятельности в виде некоторого объекта. Эту задачу можно решить с использованием интерфейсов Callable<V> и Future<V>. Совместное использование двух реализаций данных интерфейсов позволяет получить результат в виде некоторого объекта.

Интерфейс Callable<V>

Интерфейс **Callable**<V> очень похож на интерфейс Runnable. Объекты, реализующие данные интерфейсы, исполняются другим потоком. Однако, в отличие от Runnable, интерфейс **Callable** использует Generic'и для определения типа возвращаемого объекта. Runnable содержит метод run(), описывающий действие потока во время выполнения, а Callable – метод call().

Интерфейс Future<V>

Интерфейс **Future** также, как и интерфейс *Callable*, использует Generic'и. Методы интерфейса можно использовать для проверки завершения работы потока, ожидания завершения и получения результата. Результат выполнения может быть получен методом *get*, если поток завершил работу. Прервать выполнения задачи можно методом *cancel*. Дополнительные методы позволяют определить завершение задачи: нормальное или прерванное. Если задача завершена, то прервать ее уже невозможно.

Интерфейс java.util.concurrent.Future описывает API для работы с задачами, результат которых мы планируем получить в будущем: методы получения результата, методы проверки статуса. Для Future нас интересует его реализация java.util.concurrent.FutureTask. То есть это Task, который будет выполнен во Future. Чем эта реализация ещё интересна, так это тем, что она реализует и Runnable.

Методы интерфейса Future

Метод	**Описание**
cancel (boolean mayInterruptIfRunning)	попытка завершения задачи
V get()	ожидание (при необходимости) завершения задачи, после чего можно будет получить результат
V get(long timeout, TimeUnit unit)	ожидание (при необходимости) завершения задачи в течение определенного времени, после чего можно будет получить результат
isCancelled()	вернет true, если выполнение задачи будет прервано прежде завершения
isDone()	вернет true, если задача завершена

Пример:

```
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;

public class HelloWorld {

    public static void main(String []args) throws Exception {
        Callable task = () -> {
```

```
            return "Hello, World!";
        };
        FutureTask<String> future = new FutureTask<>(task);
        new Thread(future).start();
        System.out.println(future.get());
    }
}
```

ExecutorService

В многопоточный пакет concurrent для управления потоками включено средство, называемое сервисом исполнения **ExecutorService**. Данное средство служит альтернативой классу Thread, предназначенному для управления потоками. В основу сервиса исполнения положен интерфейс **Executor**, в котором определен один метод:

```
void execute(Runnable thread);
```

При вызове метода execute исполняется поток thread. То есть, метод execute запускает указанный поток на исполнение. Следующий код показывает, как вместо обычного старта потока Thread.start() можно запустить поток с использованием сервиса исполнения :

```
// Вместо следующего кода
new Thread(new RunnableTask()).start();

// можно использовать
ExecutorService executor;
. . .
executor.execute(new CallableSample1());
Future<String> f1 = executor.submit(new CallableSample2());
```

При запуске задач с помощью Executor пакета java.util.concurrent не требуется прибегать к низкоуровневой поточной функциональности класса Thread, достаточно создать объект типа ExecutorService с нужными свойствами и передать ему на исполнение задачу типа Callable. Впоследствии можно легко просмотреть результат выполнения этой задачи с помощью объекта Future.

Интерфейс **ExecutorService** расширяет свойства *Executor*, дополняя его методами управления исполнением и контроля. Так в интерфейс ExecutorService включен метод shutdown(), позволяющий останавливать все потоки исполнения, находящиеся под управлением экземпляра ExecutorService.

Методы интерфейса ExecutorService

Метод	**Описание**
boolean **awaitTermination**(long timeout, TimeUnit unit)	Блокировка до тех пор, пока все задачи не завершат выполнение после запроса на завершение работы или пока не наступит тайм-аут или не будет прерван текущий поток, в зависимости от того, что произойдет раньше
List<Future<T>> **invokeAll** (Collection<? extends Callable<T>> tasks)	Выполнение задач с возвращением списка задач с их статусом и результатами завершения
List<Future<T>> **invokeAll** (Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Выполнение задач с возвращением списка задач с их статусом и результатами завершения в течение заданного времени

T **invokeAny**(Collection<? extends Callable<T>> tasks)	Выполнение задач с возвращением результата успешно выполненной задачи (т. е. без создания исключения), если таковые имеются
T **invokeAny**(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Выполнение задач в течение заданного времени с возвращением результата успешно выполненной задачи (т. е. без создания исключения), если таковые имеются
boolean **isShutdown()**	Возвращает true, если исполнитель сервиса остановлен (shutdown)
boolean **isTerminated()**	Возвращает true, если все задачи исполнителя сервиса завершены по команде остановки (shutdown)
void **shutdown()**	Упорядоченное завершение работы, при котором ранее отправленные задачи выполняются, а новые задачи не принимаются
List<Runnable> **shutdownNow()**	Остановка всех активно выполняемых задач, остановка обработки ожидающих задач, возвращение списка задач, ожидающих выполнения
Future<T> **submit**(Callable<T> task)	Завершение выполнения задачи, возвращающей результат в виде объекта Future
Future<?> **submit**(Runnable task)	Завершение выполнения задачи, возвращающей объект Future, представляющий данную задачу
Future<T> **submit**(Runnable task, T result)	Завершение выполнения задачи, возвращающей объект Future, представляющий данную задачу

Наибольший интерес в интерфейсе **ExecutorService** представляет метод *submit()*, который ставит задачу в очередь на выполнение. В качестве входного параметра данный метод принимает объект типа *Callable* или *Runnable*, а возвращает параметризованный объект типа *Future*, который можно использовать для доступа к результату выполнения задачи. Метод *call* соответствующего Callable-объекта возвращает объект *Future*. С использованием объекта Future можно определить завершение выполнения задачи (метод isDone()) и получить доступ к результату (метод get) или исключительной ситуации, если в процессе выполнения задачи произошла ошибка.

Стоит обратить внимание на метод *shutdown()*, который выполняет остановку объекта *ExecutorService*. Поскольку потоки в объекте *ExecutorService* не останавливаются сами, как обычно, поэтому их необходимо явно остановить с помощью данного метода; при этом, если в ExecutorService находятся невыполненные задачи, то потоки будут остановлены только, когда завершится последняя задача.

ExecutorService представляет собой суб-интерфейс Executor, который добавляет функциональность для управления жизненным циклом потоков. Он также включает в себя метод submit(), который аналогичен методу execute(), но более универсален. Перегруженные версии метода submit() могут принимать как выполняемый (Runnable), так и вызываемый (Callable) объект. Вызываемые объекты аналогичны выполняемым, за тем исключением, что задача, определенная вызываемым объектом, также может возвращать значение. Поэтому, если мы передаем объект Callable методу submit(), он возвращает объект Future. Этот объект можно

использовать для получения возвращаемого значения Callable и управления статусом как Callable, так и Runnable задач.

Помимо трех вышеупомянутых интерфейсов, Executor Framework также содержит класс Executors, который по умолчанию включает в себя методы для создания различных типов служб-исполнителей. С помощью этого класса и интерфейсов можно создавать пулы потоков. Что же это такое?

Пулы потоков

Пул потоков — это набор объектов Runnable и постоянно работающих потоков. Коллекция объектов Runnable называется рабочей очередью. Постоянно запущенные потоки проверяют рабочий запрос на наличие новой работы, и если новая работа должна быть выполнена, то из рабочей очереди будет запущен объект Runnable. Чтобы использовать фреймворк Executor, нам нужно создать пул потоков и отправить туда задачу для выполнения. В классе Executors есть четыре основных метода, которые используются для создания пулов потоков. Рассмотрим каждый из них на примере.

newSingleThreadExecutor()

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class MyRunnable implements Runnable {
    private final String task;

    MyRunnable(String task) {
        this.task = task;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
           System.out.println("Executing "+ task +" with
"+Thread.currentThread().getName());
        }
        System.out.println();
    }
}

public class Exec1 {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        for (int i = 1; i <= 5; i++) {
            Runnable worker = new MyRunnable("Task" + i);
            executor.execute(worker);
        }
        executor.shutdown();    /* После этого исполнитель перестанет
принимать какие-либо новые потоки и завершит все существующие в очереди */
    }
}
```

В данном примере мы отправляем на исполнение пять задач. Но так как применяется метод newSingleThreadExecutor(), будет создан только один новый поток и одновременно будет выполняться только одна задача. Остальные четыре задачи находятся в очереди ожидания. Как только задача выполнится потоком, этот поток тут же выберет и выполнит следующую. Метод shutdown() ожидает завершения выполнения задач, в настоящий момент переданных исполнителю, чтобы завершить его работу. Однако, если вам хочется завершить работу исполнителя без ожидания, используйте вместо этого метод shutdownNow().

newFixedThreadPool()

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class MyRunnable implements Runnable {
    private final String task;

    MyRunnable(String task) {
        this.task = task;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
           System.out.println("Executing "+ task +" with
"+Thread.currentThread().getName());
        }
        System.out.println();
    }
}

public class Exec2 {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 1; i <= 5; i++) {
            Runnable worker = new MyRunnable("Task" + i);
            executor.execute(worker);
        }
        executor.shutdown();
    }
}
```

Использован тот же пример, что и в предыдущем случае, только на этот раз — с методом newFixedThreadPool(). Этот метод позволяет создать пул с фиксированным количеством потоков. Таким образом, когда мы отправим пять задач, в коде будет создано три новых потока и будут выполнены три задачи. Остальные две задачи находятся в очереди ожидания. Как только какая-либо задача выполнится потоком, этим же потоком будет выбрана и выполнена следующая задача.

newCachedThreadPool()

Когда мы создаем пул потоков с помощью этого метода, максимальный размер пула потоков устанавливается на максимальное целочисленное значение в Java. Этот метод создает новые потоки по запросу и разрушает потоки, которые простаивают больше минуты, если запрос отсутствует.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class MyRunnable implements Runnable {
    private final String task;

    MyRunnable(String task) {
        this.task = task;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
           System.out.println("Executing "+ task +" with
"+Thread.currentThread().getName());
        }
        System.out.println();
    }
}

public class Exec3 {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 1; i <= 5; i++) {
            Runnable worker = new MyRunnable("Task" + i);
            executor.execute(worker);
        }
        executor.shutdown();
    }
}
```

В данном примере метод `newCachedThreadPool()` изначально создаст пять новых потоков и обработает пять задач. Никакой очереди ожидания здесь не будет. Если поток остается в бездействии более минуты, метод устраняет его. Таким образом, этот метод — хороший выбор, если вам хочется добиться большей производительности очереди, чем это возможно с методом `newFixedThreadPool()`. Но если вы хотите ограничить количество параллельно выполняемых задач во имя управления ресурсами, лучше использовать `newFixedThreadPool()`.