Java Callable (Future) Understanding

# Java callable understanding

In Java Multithreading program we extensively use Java Callable and Future. I believe all of you have basic understanding in thread. In briefly thread is a separate path of execution so if you have to do a repetitive task you can break the work into multiple chunk(tasks) and assign them to threads. Multiple Threads will execute tasks in parallel to get the result quickly.

From Java 5 **java.util.concurrent.Callable** interface has been introduced in concurrency package, which is similar to Runnable interface but it can return any **Object and able to throw Exception.**

Java Callable interface use Generic <T> so it can return any type of Object. Executor Framework provide a submit() method to execute Callable implementations in a thread pool. Actually Java Executor Framework follows WorkerThread pattern where in a thread pool you can initiate threads by Executors.*newFixedThreadPool*(10); method. Then you can submit task into it ,as you can remember in java ,runnable act as target of a thread and in runnable interface a method ***public void run() method*** has to be implemented where you define the task which will executed by Threads in thread pool. Executor framework assign a work (runnable target) to threads only if there is an available thread in pool. If all threads are in use work has to wait. Once a task is completed by a Thread same returns to pool as available thread. Callable is same as Runnable but it can return any type of Object, if we want to get a result/ status from a work (callable) You can return it.

## Java Future

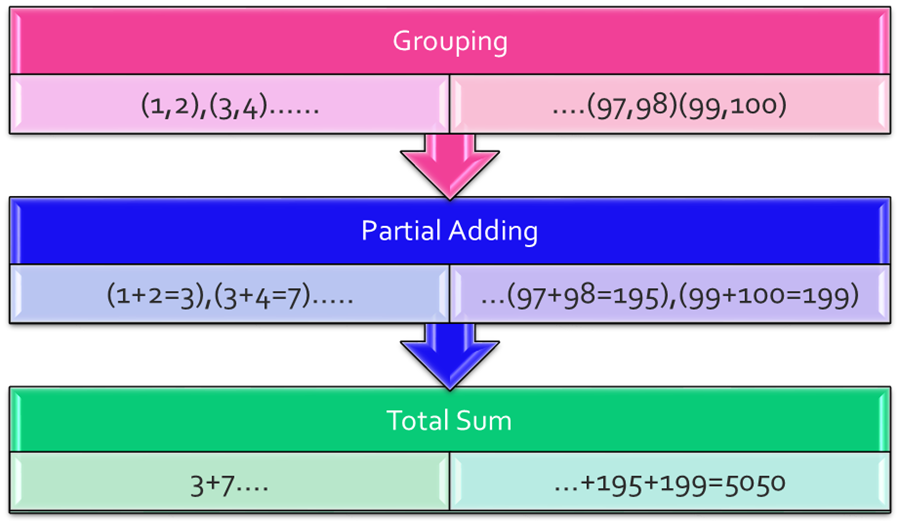
Java Callable tasks return **java.util.concurrent.Future** object. Java Future provides **cancel()** method to cancel the associated Callable task. There is an overloaded version of get() method where we can specify the time to wait for the result, it’s useful to avoid current thread getting blocked for longer time. Please note that get method is synchronous method, until callable finishes it’s task and returns a value it will wait for callable. There are **isDone()** and **isCancelled()** methods to find out the current status of associated Callable task.

Example : Suppose the problem is to find sum 1 to 100. We can do it by looping 1 to 100 sequentially and adding them.

Another way we can achieve it by *divide and Conquer* rule. Group the numbers in a way so each group has exact two elements. Then Assign those group to a pool of threads

So each thread returns a partial sum parallely. Then collect those partial sum and add them in order to get whole sum.

Step 1 :



Grouping

|  |  |  |  |
| --- | --- | --- | --- |
| 1,2 | 3,4 | …. | 99,100 |

Partial Adding

|  |  |  |  |
| --- | --- | --- | --- |
| 1+2=3 | 3+4=7 | ….. | 99+100=199 |

Collect and Total Sum

|  |
| --- |
| 3 + 7+……..199 =5050 |

Coding :

Step : create a Adder class implement callable to do the Partial sum on group

**package** com.example.thread.callable;

**import** java.util.concurrent.Callable;

**public** **class** CallableAdder **implements** Callable<Integer> {

Integer operand1;

Integer operand2;

CallableAdder(Integer operand1,Integer operand2)

{

**this**.operand1=operand1;

**this**.operand2=operand2;

}

**public** Integer call() **throws** Exception {

// **TODO** Auto-generated method stub

System.***out***.println(Thread.*currentThread*().getName()+" says : partial Sum for " + operand1 + " and "+ operand2+ " is " +(operand1+operand2));

**return** operand1+operand2;

}

}

Step 2: Create a manager class which responsible for grouping integers and submit group to Executor framework for partial add then collect the partial sum,

Wait till all partial sum returns and add them

package com.example.thread.callable;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class ParallelAdder {

public Integer parallelSum()

{

//long t1 = System.currentTimeMillis();

ExecutorService executor = Executors.newFixedThreadPool(10);

List <Future<Integer>> list = new ArrayList<Future<Integer>>();

int count=1;

int prev=0;

for(int i=1;i<=100;i++)

{

if(count%2==0)//grouping

{

System.out.println("Prev :" + prev + " current: " + i);

Future<Integer> future = executor.submit(new CallableAdder(prev,i));

list.add(future);

count=1;

continue;

}

prev=i;

count++;

}

int totsum=0;

for(Future<Integer> fut : list)

{

try {

totsum = totsum+ fut.get();

} catch (InterruptedException e) {

e.printStackTrace();

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ExecutionException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

System.out.println("Total Sum is " + totsum);

//long t2 = System.currentTimeMillis();

// System.out.println("Time taken by parallelSum " + (t2-t1));

return totsum;

}

public int sequentialSum()

{

//long t1 = System.currentTimeMillis();

Integer totsum=0;

for(int i=0;i<=100;i++)

{

totsum=totsum+i;

}

//long t2 = System.currentTimeMillis();

System.out.println("sequentialSum Total Sum is " + totsum);

//System.out.println("Time taken by sequentialSum " + (t2-t1));

return totsum;

}

public static void main(String[] args) {

ParallelAdder adder = new ParallelAdder();

int pSum= adder.parallelSum();

int sSum= adder.sequentialSum();

System.out.println("parallel Sum equals to Sequential Sum ? " );

System.out.println("Answer is :: " + (pSum==sSum));

}

}