How to use Java parallel Fork/Join framework - Hello world example

Sheng Wang 11:54 PM Concurrency, High Level Concurrency, Java SE 3 Comments

Since Java 7, fork/join framework has be introduced in to Java API. The main difference between fork/join and other multi-threading mechanism like executor or thread pools is that: traditional multithreading focuses on "let every task has the chance to run simultaneously", fork/join framework focuses on "saturate the CPU usage, make full use of the hardware resources".

0. Why fork/join?

So what's the problem for traditional thread pools, or why does fork/join need to be introduced since there are already many ways for working in parallel?Fork/join framework normally used with single task, which is BIG. Let's suppose the CPU has 4 cores. To make max usage of the quad-core CPU, you don't want any core idle while others are still busy with some work. But If you split the big task into 4 subtasks and give them to each core to run using the traditional thread pools, when every thread terminate, a core will become idle while the rest cores are still struggling. It's a kind of waste of CPU resources. The fork/join framework can prevent this from happening. Every core keeps busy until the whole job is done. **When a thread(CPU core) is idle or done with it's workload, it will try to help other threads instead of just sit there doing nothing.** In fork/join framework, it's called work stealing**.**

Work stealing is the key feature of the fork/join framework.

1. Working theory

In brief, the fork/join framework also use a thread pool, **java.util.concurrent.ForkJoinPool**, but unlike traditional thread pool, every thread in this pool has a queue. Every thread can access each other thread's queue. The queue of each thread can be treat as a work load buffer.

1.1 What happens when firstly a thread get a task

The fork/join framework starts in this way, suppose the first thread get the task is call **Thread A**:

step 1. When **thread A** gets a task, if it's small enough, do the real calculation; if still big, the task will be cut into 2 subtasks.

step 2. The **thread A** will keep working on one of the subtasks, and put the rest into **thread A**'s queue (its own queue).

step 3. An Idle thread, **thread B**, can take subtasks out from **thread A**'s queue, which is called work stealing. Then **on thread B**, same process repeats from step 1

After a big task submits to one thread initially, it will soon propagated to **ALL** threads of the fork/join thread pool. Something's worthy to mention is that Thread A will keep recursively cut task->queue 1/2->cut 1/2 task-> queue 1/4 -> cut 1/4-> queue 1/8...... until the task is small enough. Recursion is also a feature of fork/join framework.

By default the fork/join thread pool will has threads size exactly same as the available threading unit that you CPU can run simultaneously. For example a Quad-core CPU with Hyper-Threading(2 threads on each physical core), the pool will has 4*2 = 8 threads. So after task has been given to the fork/join pool, all threads/ all CPU will be occupied.

1.2 What happens when any threads finish a subtask

If thread X gets a task and splits it into 2 subtasks, it puts half in to its queue and starts working on the other half. When the second half is done, it will try to check if the first half is done.

- if the first half is done, then it can continue to work stealing.
- if the first half has been stolen and processed by other thread, thread X has to wait until this half finish.
- if the first half is still in the queue, thread X will start to process the first half itself recursively, which means cut the first half, queue 1/4 and work on the other 1/4.

2. Java API

In API level, when to put a subtask into the queue, call **fork()**. when to process a some pieces of work ,call **compute()**. when to wait for rest to finish call **join()**, These 3 methods are key methods of the fork/join framework, which are also where the framework's name comes from.

Always call fork() before compute() and join() so other threads can have the chance to help sharing the workload

In package java.util.concurrent, there are 4 classes key to fork/join framework.

- **ForkJoinPool** - Thread pool for fork/join framework. Implements ExecutorService interface.
- **ForkJoinTask** - Abstract class, has **fork()** and **join()** method, as parent class for the next 2 children.
- **RecursiveTask** - Abstract class extends ForkJoinTask, only abstract method is **compute()**
- **RecursiveAction** - Abstract class extends ForkJoinTask, only abstract method is **compute()**

The only difference between RecursiveTask and RecursiveAction is that RecursiveTask's compute() has return, but RecursiveAction's compute() doesn't. (Task has return, action doesn't)

3. Demo

We have big char array, 100M items. Every item in this array is one upper case letter from A-Z. The application tries to count how many letter 'A' in this big array. By using fork/join framework, the array will be divided into small area for each thread to go through. Let's first see the main class.

```java
package com.shengwang.demo;

public class ForkJoinDemo {
  private static final int ARRAY_SIZE = 100_000_000;
  private static char[] letterArray = new char[ARRAY_SIZE];

  private static int countLetterUsingForkJoin(char key) {
    int total = 0;
    ForkJoinPool pool = new ForkJoinPool(); // create thread pool for fork/join
    CountLetterTask task = new CountLetterTask(key, letterArray, 0, ARRAY_SIZE);
    total = pool.invoke(task); // submit the task to fork/join pool

    pool.shutdown();
    return total;
  }

  public static void main(String[] args) {
    char key = 'A';
    // fill the big array with A-Z randomly
    for (int i = 0; i < ARRAY_SIZE; i++) {
      letterArray[i] = (char) (Math.random() * 26 + 65); // A-Z
    }

    int count = countLetterUsingForkJoin(key);
    System.out.printf("Using ForkJoin, found %d '%c'\n", count, key);
  }
}
```

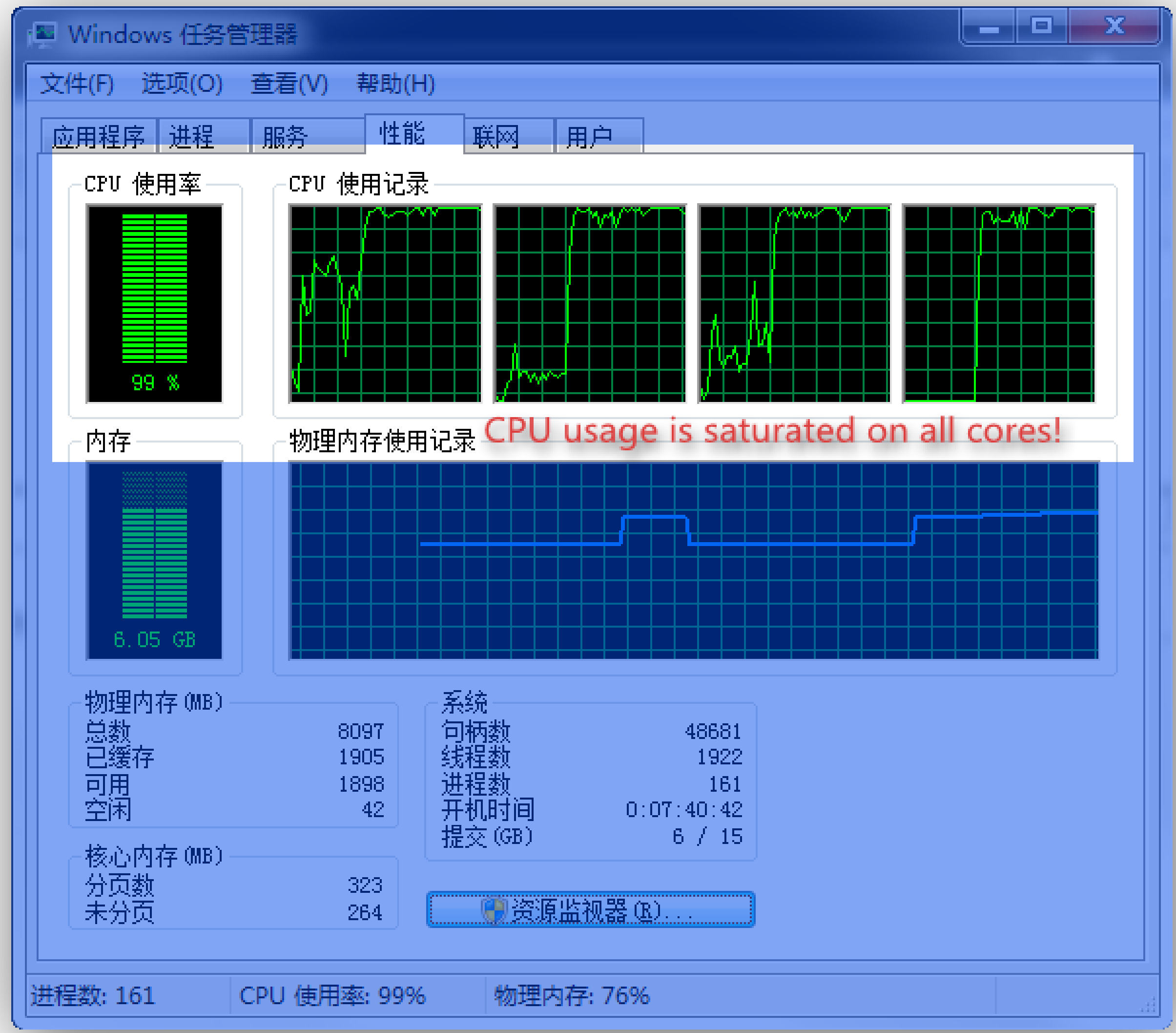
The main class is simple, main() first fill a big array with random upper case letters, then call the *countLetterUsingForkJoin()*, in which a **ForkJoinPool** is created and task submit to it. After finishing whole task and get the final result, the pool shuts down and result returned. The task class *CountLetterTask* is the kernel of this demo and it's shown below.

```java
package com.shengwang.demo;

import java.util.concurrent.RecursiveTask;

class CountLetterTask extends RecursiveTask<Integer> {

  private static final long serialVersionUID = 1L;
  private static final int ACCEPTABLE_SIZE = 10_000;
  private char[] letterArray;
  private char key;
  private int start;
  private int stop;

  public CountLetterTask(char key, char[] letterArray, int start, int stop) {
    this.key = key;
    this.letterArray = letterArray;
    this.start = start;
    this.stop = stop;
  }

  @Override
  protected Integer compute() {
    int count = 0;
    int workLoadSize = stop - start;
    if (workLoadSize < ACCEPTABLE_SIZE) {
      // String threadName = Thread.currentThread().getName();
      // System.out.printf("Calculation [%d-%d] in Thread %s\n",start,stop,threadName);
      for (int i = start; i < stop; i++) {
        if (letterArray[i] == key)
          count++;
      }
    } else {
      int mid = start + workLoadSize / 2;
      CountLetterTask left = new CountLetterTask(key, letterArray, start, mid);
      CountLetterTask right = new CountLetterTask(key, letterArray, mid, stop);

      // fork (push to queue)-> compute -> join
      left.fork();
      int rightResult = right.compute();
      int leftResult = left.join();
      count = leftResult + rightResult;
    }
    return count;
  }
}
```

Let's go through class *CountLetterTask*. It extends **RecursiveTask<Integer>** which mean final result of the task is an Integer. To avoid creating copy of the original big array, the reference of the big array will be send in as a constructor parameter. The current task size is defined by the start(inclusive) and stop(exclusive) index in the array. The criteria to say whether the current task is small enough is defined as a constant variable ACCEPTABLE_SIZE. Here when the subtask deal with part of the array less than 10k is considered as "small enough".

The most interesting part is the **compute()** method, it first checks if the current task is smaller enough, if so, do the real calculation. If not, the array range will be divided into 2 parts. One task becomes two subtasks, each is also a *CountLetterTask* instance. Put the first part into queue then call **compute()** on the second half. The task will be recursively cut small until it's "small enough". Then call the join() to make sure whole task is done. Remember **fork()** has to run before **compute()** and **join()**

4. Run


Windows 任务管理器
文件(F) 选项(O) 查看(V) 帮助(H)
应用程序 进程 服务 性能 联网 用户
CPU 使用率
99 %
CPU 使用记录
CPU usage is saturated on all cores!
内存
6.05 GB
物理内存使用记录
资源监视器(R)...

物理内存(MB)

总数	8097
已缓存	1905
可用	1898
空闲	42

核心内存(MB)

分页数	323
未分页	264

系统

句柄数	48681
线程数	1922
进程数	161
开机时间	0:07:40:42
提交(GB)	6 / 15

进程数: 161 CPU 使用率: 99% 物理内存: 76%

From the screenshot, CPU resources are fully used for the big task. (Since the task will only take less than 30ms also on my PC to finish, the screenshot actually comes from a even bigger array running in a loop for many times)

Facebook Twitter Google+ Stumble Digg

← Newer Post Home Older Post →

3 comments:

fleetwu March 3, 2016 at 8:34 PM