**Concurenta**

In Java SE5 s-a introdus pachetul java.util.concurrent ce contine clase utile in programarea concurenta. Acesta cuprinde:

* Colectii concurente
* Sincronizare si blocare alternativa
* Pool-uri de fire

In pachetul java.util.concurrent.atomic avem clase ce suporta programare thread-safe fara blocare, pe variabile singulare.

**public** **class** AtomicExample {

**public** **static** **void** main(String[] args) {

AtomicInteger ai = **new** AtomicInteger(5);

**if**(ai.compareAndSet(5, 42)) {

System.*out*.println("Replaced 5 with 42");

}

}

}

In exemplul anterior operatia atomica asigura ca valoarea curenta este 5 si apoi este setata la 42. Pentru arhitecturile procesoarelor ce suporta compararea nativa si operatia de setare nu este nevoie de blocare cand se executa exemplul anterior. Alte arhitecturi pot necesita unele forme de blocare interna.

Pachetul java.util.concurrent.locks este un framework pentru blocare si asteptarea unor conditii ce este folosit pentru sincronizare predefinita si monitoare. Una dintre facilitatile acestui pachet este implementarea multi-reader si blocarea single writer. Astfel, un fir nu va obtine blocarea la citire atunci cand o actiune de scriere este in derulare. Mai multe fire pot cere concurent pot dobandi blocarea la citire, dar numai unul blocarea la scriere. Mai mult, un fir ce a obtinut blocarea pentru scriere poate apela alte metode ce obtin blocarea la scriere fara frica de blocare a executiei.

**public** **class** ShoppingCart {

**private** **final** ReentrantReadWriteLock rwl =

**new** ReentrantReadWriteLock();

**public** **void** addItem(Object o) {

rwl.writeLock().lock();

// modify shopping cart

rwl.writeLock().unlock();

}

**public** String getSummary() {

String s = "";

rwl.readLock().lock();

// read cart, modify s

rwl.readLock().unlock();

**return** s;

}

**public** **double** getTotal() {

**double** total = 0;

rwl.readLock().lock();

// read cart, add everything to total

rwl.readLock().unlock();

**return** total;

}

}

In exemplul anterior toate metodele ce sunt destinate citirii permit executia concurenta atat a citirii singulare cat si multiple.

**Colectii thread-safe**

Colectiile din java.util nu sunt thread-safe. Pentru a le face thread-safe vom folosi blocuri de cod sincronizate pentru accesul la o colectie atunci cand facem scriere, creem un wrapper sincronizat folosind metode precum java.util.Collections.synchronizedList(List<T>) sau utilizam colectiile din java.util.concurrent.

Observatie: facand o colectie sincronizata aceasta nu inseamna ca elementele sale sunt sincronizate.

Clasa ConcurrentLinkedQueue furnizeaza o coada FIFO eficient scalabila, non-blocabila si thread-safe. Cinci implementari din java.util.concurrent suporta interfata BlockingQueue, ce definesc versiuni de blocare pentru a pune si a lua: LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue si DelayQueue.

Pe linga cozi acest pachet furnizeaza implementari de colectii destinate a fi utilizate in contexte multifir: ConcurrentHashMap, ConcurrentSkipListSet, CopyOnWriteArrayList si CopyOnWriteArraySet. Cand se asteapta ca mai multe fire sa acceseze o colectie data un ConcurrentHashMap este preferat unui HashMap sincronizat, un ConcurrentSkipListMap este prefeta unui TreeMap sincronizat. Un CopyOnWriteArraySet este preferabil unui ArrayList sincronizat atunci cand numarul asteptat de citiri si traversari este mai mare decat numarul update-urilor.

In pachetul java.util.concurrent avem cinci clase ce furnizeaza actiuni relative sincronizarii. Acestea permit firelor blocarea pina cand o anumita stare este atinsa:

* Semaphore, reprezinta un tool clasic de concurenta. Acesta gestioneaza o multime de permisiuni. Firele incearca sa obtina permisiuni si se pot bloca pina cand alte fire elibereaza permisiunile.
* CountDownLatch, un utilitar simplu pentru blocare, cat timp se pastreaza un numar de semnale, evenimete sau conditii. Unul sau mai multe fire se blocheaza pina cand un cronometru se incheie. Dupa incheierea cronometrului toate firele continua. CountDownLatch nu poate fi refolosit.
* CyclicBarrier, un punct resetabil ce permite sincronizarea pe mai multe cai si este folositoare in programarea paralela. Creaza o numaratoare de fire. Dupa ce un numar de fire au apelat await() pe CyclicBarrier, ele vor fi eliberate. CyclicBarrier sunt reutilizabile
* Phaser, furnizeaza o forma mai flexibila de barrier ce poate fi folosita pentru controlul calculelor de-a lungul firelor multiple. Firele se pot inregistra si deinregistra afectand astfel numarul de fire din barrier
* Exchanger, permit ca doua fire sa schimbe obiecte intr-un punct de intalnire si este folositor in diverse proiectari pipeline. Firele sunt blocate pina cand are loc schimbarea. Este bidirectional, eficient cu memoria si o alternativa la SynchronousQueue.

Exemplu:

**public** **class** CyclicBarrierExample {

**public** **static** **void** main(String[] args) {

**final** CyclicBarrier barrier = **new** CyclicBarrier(2);

**new** Thread() {

**public** **void** run() {

**try** {

System.*out*.println("before await - thread 1");

barrier.await();

System.*out*.println("after await - thread 1");

} **catch** (BrokenBarrierException|InterruptedException ex) {

}

}

}.start();

**new** Thread() {

**public** **void** run() {

**try** {

System.*out*.println("before await - thread 2");

barrier.await();

System.*out*.println("after await - thread 2");

} **catch** (BrokenBarrierException|InterruptedException ex) {

}

}

}.start();

}

}

In exemplul anterior am creat un CyclicBarrier care va determina ca doua fire sa apeleze await() pina cand vor fi deblocate. Daca doar un fir apeleaza await(), firul va fi blocat permanent. Clasa CyclicBarrier contine si metoda await(long timeout, TimeUnit unit), care va bloca firul pentru o durata de timp si va arunca o TimeoutException daca acea durata a fost atinsa.

**Alternative high-level ale firelor**

API-urile traditionale pentru fire sunt dificil de utilizat corect. Ca alternativa avem:

* java.util.concurrent.ExecutorService, un mecanism de nivel inalt utilizat pentru executarea task-urilor: poate crea si reutiliza obiecte Thread
* framework-ul Fork-Join, un ExecutorService nou in Java 7

ExecutorService este folosit pentru a executa task-uri eliminand nevoia de creare si gestionare manuala a firelor. Task-urile pot fi executate in paralel dependent de implementarea executorului. Task-urile pot fi: java.lang.Runnable sau java.util.concurrent.Callable.

Putem obtine instante ale ExecutorService prin Executors: ExecutorService es = Executors.newCachedThreadPool();.

Un ExecutorService pool cahed creaza fire noi la nevoie, reutilizeaza firele (firele nu mor dupa utilizare) si termina firele ce au fost inactive (idle) timp de 60 de secunde.

Un alt tip de ExecutorService este ilustrat in exemplu:

**int** count = Runtime.*getRuntime*().availableProcessors();

ExecutorService ex = Executors.*newFixedThreadPool*(count);

Acesta creaza un fixed pool, ce contine un numar fix de fire, reutilizeaza firele, stocheaza in coada task-urile pina cand un fir devine disponibil si poate fi utilizat pentru a evita supraincarcarea unui sistem.

Interfata Callable defineste un task trimis unui ExecutorService. Este similar lui Runnable, dar poate returna un rezultat folosind generice si arunca o checked exception.

public interface Callable<V>{

V call() throws Exception;

}

Interfata Future este folosita pentru a obtine rezultate din apelul metodei call() a interfetei Callable, asa dupa cum ilustreaza urmatoarea secventa de cod:

Future<V> future = es.submit(callable);// es este un ExecutorService, care controleaza activitatea firului

**try** {

V result = future.get();

} **catch** (ExecutionException | InterruptedException ex) {}

Pentru ca apelul lui get() va determina blocarea va trebui sa facem una dintre urmatoarele actiuni:

* sa trimitem toata treaba unui ExecutorService inainte de apelul lui get()
* trebuie sa pregatim asteptarea pentru un obiect Future care va obtine rezultatul
* folosim o metoda non-blocking precum isDone() inaintea apelului lui get() sau get(long timeout, TimeUnit unit), care va arunca o TimeoutException daca rezultatul nu este disponibil intr-o anumita perioada de timp

Oprirea unui ExecutorService este importanta pentru ca firele sale sunt nondaemon si va opri JVM de la inchidere.

//Stop accepting new Callables

es.shutdown();

**try** {

//Block until all Callables have a chance to finish

es.awaitTermination(5, TimeUnit.*SECONDS*);

} **catch** (InterruptedException ex) {

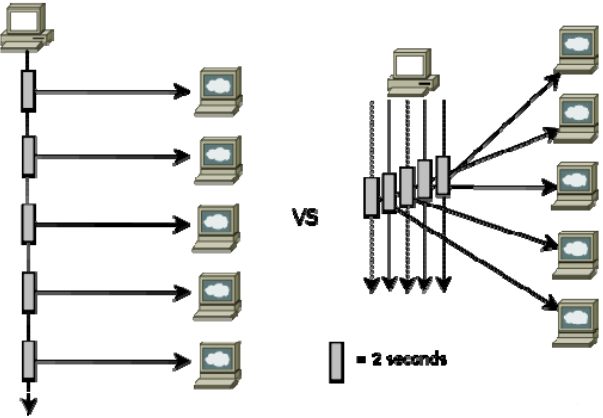
System.*out*.println("Stopped waiting early");

}

Este recomandabil ca dupa inchiderea executorului sa mai aspetam un pic dand sanse ca toate Callable-urile sa se termine.

**Concurenta in I/O**

Exista diferite modalitati de a masura timpul. Spre exemplu, o secventa de 5 apeluri in serie catre servere va dura aproximativ 10 secunde daca fiecare apel dureaza 2 secunde. 5 apeluri concurente catre servere va dura un pic peste 2 secunde daca fiecare apel dureaza 2 secunde. Graficul ilustrativ al procesului este dat mai jos:



Fie urmatorul exemplu al unui client intr-o retea cu un singur fir:

**public** **class** SingleThreadClientMain {

**public** **static** **void** main(String[] args) {

String host = "localhost";

**for** (**int** port = 10000; port < 10010; port++) {

RequestResponse lookup = **new** RequestResponse(host, port);

**try** (Socket sock = **new** Socket(lookup.host, lookup.port);

Scanner scanner = **new** Scanner(sock.getInputStream());) {

lookup.response = scanner.next();

System.*out*.println(lookup.host + ":" + lookup.port + " "

+ lookup.response);

} **catch** (NoSuchElementException | IOException ex) {

System.*out*.println("Error talking to " + host + ":" + port);

}

}

}

}

Exemplul prezentat incearca sa descopere care vendor ofera pretul minim pentru un articol. Clientul comunica cu 10 servere diferite, fiecarui server trebuindu-i 2 secunde pentru a cauta datele cerute si a le returna. Pot interveni si alte intarzieri introduse de retea. Acest client intr-un singur fir trebuie sa astepte dupa fiecare server raspunsul inainte de a trece la urmatorul. Timpul total va fi de aproape 20 de secunde.

In urmatorul exemplu vom da varianta de invocare asincrona:

**public** **class** MultiThreadedClientMain {

**public** **static** **void** main(String[] args) {

//ThreadPool used to execute Callables

ExecutorService es = Executors.*newCachedThreadPool*();

//A Map used to connect the the request data with the potential result

Map<RequestResponse, Future<RequestResponse>> callables = **new** HashMap<>();

String host = "localhost";

//loop to create and submit a bunch of Callable instances

**for** (**int** port = 10000; port < 10010; port++) {

RequestResponse lookup = **new** RequestResponse(host, port);

NetworkClientCallable callable = **new** NetworkClientCallable(lookup);

Future<RequestResponse> future = es.submit(callable);

callables.put(lookup, future);

}

//Stop accepting new Callables

es.shutdown();

**try** {

//Block until all Callables have a chance to finish

es.awaitTermination(5, TimeUnit.*SECONDS*);

} **catch** (InterruptedException ex) {

System.*out*.println("Stopped waiting early");

}

**for** (RequestResponse lookup : callables.keySet()) {

Future<RequestResponse> future = callables.get(lookup);

**try** {

lookup = future.get();

System.*out*.println(lookup.host + ":" + lookup.port + " " + lookup.response);

} **catch** (ExecutionException | InterruptedException ex) {

//This is why the callables Map exists

//future.get() fails if the task failed

System.*out*.println("Error talking to " + lookup.host + ":" + lookup.port);

}

}

}

}

In acelasi scenariu de la exemplul precedent, clientul nu asteapta ca fiecare server sa raspunda inainte de a trece la a comunica cu urmatorul server. In acest caz timpul de executie este de aproximativ 2 secunde.

Clasele auxiliare necesare rularii sunt date ami jos, inclusiv cele de pe partea de server:

**public** **class** RequestResponse {

**public** String host; //request

**public** **int** port; //request

**public** String response; //response

**public** RequestResponse(String host, **int** port) {

**this**.host = host;

**this**.port = port;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (obj **instanceof** RequestResponse) {

RequestResponse lookup = (RequestResponse) obj;

**if** (host.equals(lookup.host) && port == lookup.port) {

**return** **true**;

}

}

**return** **false**;

}

**public** **int** hashCode() {

**int** hash = 7;

hash = 97 \* hash + Objects.*hashCode*(**this**.host);

hash = 97 \* hash + **this**.port;

**return** hash;

}

}

**public** **class** NetworkClientCallable **implements** Callable<RequestResponse> {

**private** RequestResponse lookup;

**public** NetworkClientCallable(RequestResponse lookup) {

**this**.lookup = lookup;

}

@Override

**public** RequestResponse call() **throws** IOException {

**try** (Socket sock = **new** Socket(lookup.host, lookup.port);

Scanner scanner = **new** Scanner(sock.getInputStream())) {

lookup.response = scanner.next();

**return** lookup;

}

}

}

**public** **class** PriceRangeServer **implements** Runnable {

**private** String price;

**private** ServerSocket ss;

//low inclusive

//high exclusive

**public** PriceRangeServer(**int** port, **int** low, **int** high) **throws** IOException {

ss = **new** ServerSocket(port);

ss.setSoTimeout(250);

**double** d = Math.*random*() \* (high - low) + low;

price = String.*format*("%.2f", d);

}

**public** **void** accept() **throws** IOException {

System.*out*.println("Accepting connections on port " + ss.getLocalPort());

**while** (!Thread.*interrupted*()) {

**try** (Socket sock = ss.accept();

BufferedWriter bw = **new** BufferedWriter(**new** OutputStreamWriter(sock.getOutputStream()))) {

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException ex) {

**return**;

}

bw.write(price);

} **catch** (SocketTimeoutException ste) {

//timeout every .25 seconds to see if interrupted

}

}

System.*out*.println("Done accepting");

}

@Override

**public** **void** run() {

**try** {

accept();

} **catch** (IOException ex) {

ex.printStackTrace();

}

}

}

**public** **class** NetworkServerMain {

**public** **static** **void** main(String[] args) {

ExecutorService exSrv = Executors.*newCachedThreadPool*();

List<Runnable> runners = **new** ArrayList<>();

**for** (**int** port = 10000; port < 10010; port++) {

Runnable r;

**try** {

r = **new** PriceRangeServer(port, 20, 110);

runners.add(r);

} **catch** (IOException ex) {

System.*out*.println("Port " + port + " already in use");

}

}

**for** (Runnable r : runners) {

exSrv.execute(r);

}

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException ex) {

}

System.*out*.println("Press enter to quit...");

**try** {

System.*in*.read();

} **catch** (IOException ex) {

}

System.*out*.println("Quiting...");

exSrv.shutdownNow();

}

}

Sistemele moderne contin multiple procesoare. Pentru a folosi avantajul puterii de procesare va trebui sa executam task-uri in paralel pe aceste procesoare. Situatiile aparute sunt:

* imparte si stapaneste: un task se divide in subtask-uri. Va trebui doar sa identificam acele subtask-uri ce se pot executa in paralel
* unele probleme sunt dificil a fi executate in task-uri paralele
* serverele ce suporta clienti multipli pot folosiun task separat pentru a manipula fiecare client
* programarea a prea multor task-uri in paralel pot avea un impact negativ de performanta

Daca task-urile sunt orientate pe calcule mai degraba decat pe I/O, numarul task-urilor paralele nu trebuie sa fie mai mare decat numarul de procesoare din sistem. Putem afla numarul procesoarelor, asa dupa cum am vazut intr-un exemplu anterior.

Daca nu folosim fire de executie, doar o portiune din puterea de procesare a sistemului va fi folosita. Spre exemplu, daca avem o cantitate mare de date de procesat si doar un fir care proceseaza aceste date, doar un procesor va fi utilizat. Intr-un sistem cu 4 procesoare, 3 dintre ele stau in asteptare pe timpul prelucrarii datelor.

O solutie simpla de folosire a paralelismului imparte datele ce urmeaza a fi procesate in mai multe multimi, distribuind o multime unui procesor si un fir pentru procesarea fiecarui set. Dupa procesare submultimile trebuie combinate intr-un mod potrivit.

Avem mai multe moduri de a imparti datele de dimensiuni mari. Spre exemplu, in cazul unui array crearea unui nou array pe fiecare fir ce contine o copie a unei portiuni din firul original este prea consumatoare de memorie. Solutia mai eficienta este ca fiecare array sa imparta o referinta dintr-un singur array mare, dar accesul sa fie doar la o submultime intr-un mod non-blocking thread-safe.

In cazul unei baze de date, impartirea pentru fiecare fir a unei submultimi de dimensiune egala de inregistrari, poate genera probleme.

Totusi, ideal ar fi ca toate procesoarele sa fie utilizate complet pina cand task-ul este terminat, dar:

* procesoarele pot rula cu viteza diferita
* task-uri non Java vor necesita procesorul si vor reduce timpul alocat unui fir Java pe respectivul procesor
* datele ce urmeaza a fi analizate pot necesita variate perioade de timp pentru a fi procesate

Pentru a tine mai multe fire ocupate:

* impartim datele de procesat intr-un numar mare de submultimi
* asignam submultimile de date unei cozi de fire ce urmeaza a fi procesate
* fiecare fir va avea mai multe submultimi asignate

Daca un fir termina toate submultimile mai repede, poate „fura” submultimile dintr-un alt fir.

Prin impartirea datelor ce urmeaza a fi procesate pina cand avem mai multe submultimi decat fire, facilitam „work-stealing”. In work-stealing un fir ce ruleaza si isi termina treaba poate fura o submultime de date din coada de procesare a unui alt fir. Trebuie doar sa decidem cantitatea optima de adaugat fiecarei cozi de procesare. O supramaruntire a datelor poate duce la o ineficienta incarcare, in timp ce o submaruntire poate conduce la o subutilizare a procesoarelor.

In exemplul urmator avem doua task-uri separate ce pot fi executate in paralel. Initializarea sirului cu valori aleatoare si cautarea in sir a valorii celei mai mari pot fi facute in paralel.

**public** **class** Main {

//We use a lot of memory

**public** **static** **void** main(String[] args) {

**int**[] data = **new** **int**[1024 \* 1024 \* 32]; //32MB

**for** (**int** i = 0; i < data.length; i++) {

data[i] = ThreadLocalRandom.*current*().nextInt();

}

**int** max = Integer.*MIN\_VALUE*;

**for** (**int** value : data) {

**if** (value > max) {

max = value;

}

}

System.*out*.println("Max value found:" + max);

}

}

Obiectul ForkJoinTask reprezinta un task ce urmeaza a fi executat:

* un task contine cod si date de procesat, similar lui Runnable sau Callable
* un numar mare de teask-uri sunt create si procesate de un numar mic de fire dintr-un pool Fork-Join. Un ForkJoinTask creaza mai multe instante ForkJoinTask pina cand datele de procesat s-au impartit in mod adecvat
* dezvoltatorii folosesc subclasele: RecursiveAction, cand un task nu necesita returnarea unui rezultat sau RecursiveTask altfel

Prin aceasta metoda acelasi sir este trimis fiecarui task, dar cu diferite valori de start si sfarsit.

Pentru utilizarea framework-ului Fork-Join trebuie sa tinem cont de urmatoarele recomandari:

* sa evitam I/O sau operatiile de blocare. Doar un fir este creat implicit pe procesor. Operatiile de blocare ne vor tine de la a utiliza toate resursele procesorului
* solutia Fork-Join va avea performante mai mici pe un singur procesor, decat solutia standard secventiala
* multe probleme conduc la alte probleme daca sunt executate in paralel

**public** **class** FindMaxTask **extends** RecursiveTask<Integer> {

**private** **static** **final** **long** *serialVersionUID* = -9018695924326599662L;

**private** **final** **int** threshold;

**private** **final** **int**[] myArray;

**private** **int** start;

**private** **int** end;

**public** FindMaxTask(**int**[] myArray, **int** start, **int** end, **int** threshold) {

**this**.threshold = threshold;

**this**.myArray = myArray;

**this**.start = start;

**this**.end = end;

}

**protected** Integer compute() {

**if** (end - start < threshold) {

**int** max = Integer.*MIN\_VALUE*;

**for** (**int** i = start; i <= end; i++) {

**int** n = myArray[i];

**if** (n > max) {

max = n;

}

}

**return** max;

} **else** {

**int** midway = (end - start) / 2 + start;

FindMaxTask a1 = **new** FindMaxTask(myArray, start, midway, threshold);

a1.fork();

FindMaxTask a2 = **new** FindMaxTask(myArray, midway + 1, end, threshold);

**return** Math.*max*(a2.compute(), a1.join());

}

}

}

**public** **class** RandomArrayAction **extends** RecursiveAction {

**private** **static** **final** **long** *serialVersionUID* = 1L;

**private** **final** **int** threshold;

**private** **final** **int**[] myArray;

**private** **int** start;

**private** **int** end;

**public** RandomArrayAction(**int**[] myArray, **int** start, **int** end, **int** threshold) {

**this**.threshold = threshold;

**this**.myArray = myArray;

**this**.start = start;

**this**.end = end;

}

**protected** **void** compute() {

**if** (end - start < threshold) {

**for** (**int** i = start; i <= end; i++) {

myArray[i] = ThreadLocalRandom.*current*().nextInt();

}

} **else** {

**int** midway = (end - start) / 2 + start;

RandomArrayAction r1 = **new** RandomArrayAction(myArray, start, midway, threshold);

RandomArrayAction r2 = **new** RandomArrayAction(myArray, midway + 1, end, threshold);

*invokeAll*(r1, r2);

}

}

}

**public** **class** Main {

//We use a lot of memory

**public** **static** **void** main(String[] args) {

**int**[] data = **new** **int**[1024 \* 1024 \* 32]; //32MB

ForkJoinPool pool = **new** ForkJoinPool();

RandomArrayAction action = **new** RandomArrayAction(data, 0, data.length-1, data.length/16);

pool.invoke(action);

FindMaxTask task = **new** FindMaxTask(data, 0, data.length-1, data.length/16);

Integer result = pool.invoke(task);

System.*out*.println("Max value found:" + result);

}

}