﻿Thread group : Group of thread

problems with traditional synchronize

1.We don't have facility to get lock without waiting.

2 that may create performance problems, may cause deadlock.

3.if a thread released a lock which waiting thread will get a lock we don't have any control on this.

4.there is no api to list out all waiting thread for a lock.

5. the synchronize we have to use either at method level or block level. With in the method its possible to use accross multiple methods.

to avoid these problems. Lock concept came.

java.util.concurrnt.lock(1.5)

Lock -

fairness policy : - longest thread will get chance.

it also provide several enhance to the programmer. like to provide more control on concurrency.

Lock (Interface :)

lock object is similar to implicit lock.

Acquired by a thread synch blok or method.

lock implementation more extensive operations than traditional implicit locks.

important method of lock.

void lock() : we can use this method, to get a lock if lock is already available then immediately current will get a lock.

if the lock is not available. then it will wait until getting the lock

It is exactly same behavior of traditional synchronized key .

boolean tryLock():-- to acquire a lock without waiting.

if we get a lock(return true), then perform safe operation else

perform normal operation.(retun false). in this case thread wount go for waiting state.

if(l.trylock()){

perform safe operation.

}else{

perform alernate operation.

}

3. boolean tryLock(long time,TimeUnit unit) : if the lock is not available the thread will wait specified amount of time.

still if the lock is not available. then thread can continue its execution.

ex tryLock(1,TimeUnit.HOUR)

Time Unit : Is an enum , java.util.concocurrent.TimeUnit

time unit is an ENUM .

enum TimeUnit{

NANOSECOND

MICROSECOND

MILLIDECOND

SECONDS

MINITES

HOURS

DAYS

}

if(l.tryLock(1000,TimeUnit.MILISECONDS)){

......

}

void lockIntruptibly(): get the lock if its available,and returns immediately,

if the lock is not available then it will wait,while waiting if the thread is interrupted, then thread wouldn't get the lock.

void unLock(): to release the lock.

to call this method compulsory current thread should be owner of the lock.

Otherwise we will get Runtime EXception IllegalMonirtStateException

ReentrantLock(C) : implementation of Lock interface.

and direct child class of Object.

Reentrant : again an again

a thread can take same lock multiple times without any issue.

Internally, ReentrantLock increment threads personal count. whenever we call lock method. and Decrement count value.

Whenever thread call unlock method.

and lock will be released whenever count reaches 0

ReentrantLock re = new Reentrant()

l.lock()

Constructor

ReentrantLock re = new Reentrant() : created an instance of ReentrantLock lock.

ReentrantLock re = new Reentrant(boolean fairness)

created an instance of ReentrantLock.with the given fairness policy.

if the fairness is true. then longest waiting thread can acquire the lock if it available, that is it follows

FCFS policy.

If fairness is false then. which waiting thread will get the chance we can't expect

Note : Default value for fairness is false.

int getHouldCount() : -return number of holds on this lock by current thread

boolean idHeldByCurrentThread(): -returns true iff lock is hold by current thread.

int getQueueLength(): returns number of thread waiting for lock.

Collection getQueuedThreads():

boolean hasQueuedThreads()

boolean isLocked()

Thread getOwner()

class Display{

public void synchronized wish(String name){

for(int i=0;i<10;i++){

System.out.print("Good Morning");

try{

Thread.sleep(2000);

}catch(Exception e){

}

System.out.print(name);

}

}

}

with ReentrantLock

class Display{

ReentrantLock re = new Reentrant();

public void wish(String name){

re.lock(); --- line 1

for(int i=0;i<10;i++){

System.out.print("Good Morning");

try{

Thread.sleep(2000);

}catch(Exception e){

}

System.out.print(name);

}

re.unlock(); --- line 2

}

}

Thread Pools(Executor Framework) : Introduces java 1.5 version

creating a new thread for every job may create performance and memory problem, to overcome this we should go for thread pool.

thread pool, is a pool of already created thread , ready to do our jobs.

Thread pool framework also knows as executor framework.

Advantage of Thread pool

1. Thread creation

2. Thread Management

3. Task submission and execution

we can create a thread pool as follows

Types of way to create thread-pool:

1 Fixed thread pool : fixed number of thread are allowed and for other thread have to wait

ExecutorService service = Excecutors.newFixedThreadPool(9)

2 Cached thread pool : depends on number of thread. no wait (all the task start concurrently)

ExecutorService service = Excecutors.newCachedThreadPool()

3 Single thread executor

ExecutorService service = Excecutors.newSingleThreadExwcutor()

at a time only 1 thread is allowed to perform the task.

internally implemented synchronization.

Got result 1 by 1 for multiple task submitted.

\*\*\*\*\*\*\*\*\*\*\*\*\* Tas1 start -------------

\*\*\*\*\*\*\*\*\*\*\*\*\* Tas1 End -------------

\*\*\*\*\*\*\*\*\*\*\*\*\* Tas2 start -------------

\*\*\*\*\*\*\*\*\*\*\*\*\* Tas2 End -------------

ExecutorService service = Excecutors.newFixedThreadPool(3)

service.execute(new task());

.....

service.shutdown();

service.submit(job);

only 3 thread are allowed.

we can submit, a runnable job by using submit(j)

we can shutdown : service.shutdown();

ScheduledExecutorService :

ScheduledExecutorService is used to execute a task either periodically or after a specified delay.

Example :

ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);

Runnable task = () -> {

System.out.println("Executing Task At " + System.nanoTime());

};

System.out.println("Submitting task at " + System.nanoTime() + " to be executed after 5 seconds.");

scheduledExecutorService.schedule(task, 5, TimeUnit.SECONDS);

scheduledExecutorService.shutdown();

Callable and Future:

In the case of runnable job, thread woun't return any thing after completing job.

If a thread is required to return some result, after execution,then we should goo for callable.

callable interface contain only 1 method.

public Object call() throws Exception{

...........

}

If we submit a callable object to executor then , after completing the job,

thread returns a Future Object.

Future<String> future = executorService.submit(callable);

that is Future object can be used to retrieve the result from callable job

String result = future.get();

System.out.println(result);

executorService.shutdown();

Note that, the get() method blocks until the task is completed.

The Future API also provides an isDone() method to check whether the task is completed or not.

Difference b/w runnable and callable

Runnable : if a thread is not required to return anything , after job done then we should go for runnable.

Callable : if a thread is required to return something , after job done then we should go for callable.

Runnable : runnable interface contains only 1 method run()

Callable : callable interface contains only 1 method call()

Runnable : runnable job not required to return anything.hence return type of run method is void

Callable : callable job required to return anything.hence return type of run method is Object.

Runnable : with in the run method. is there any chance of rasing checked exception . compulsory we should handle using try catch.

because. we can't use throws keyword for run method.

Callable : with in the call method. is there any chance of rasing checked exception . Not required by using try catch.

because. we call method throws exception.

Runnable :runnable is java.lang package

Callable :callable present in java.util.concurrent package

Introduced in 1.0 version

introduced in 1.5 version

ThreadLocal

ThreadLocal class provide thread local variables.

ThreadLocal class maintains values per thread bases. Each thread local object maintains a separate value like user id, transaction id etc.

For each thread that access as Object.

thread can access its local value, manipulate its value. and even can remove its value.

in every part of the code which is executed by the thread.

we can access its local variable.

Ex

consider a servlet, which invokes some business methods, we have a requirement

to generate a unique tx id for each and every req and we have to pass this tx id to the business methods

for this requirement we can use thread-local.

to maintain a separate tx id for every req. ie for every thread.

Note : ThreadLocal class introduced in 1.2 version. And enhanced in 1.5 version.

ThreadLocal can be associated with thread scope.

Total code which is executed has access to the corresponding ThreadLocal variable.

a thread can access its own ThreadLocal variable and can't access other local variable.

once thread entered into dead state . all its local variable by default available for GC

ThreadLocal tl = new ThreadLocal();

methods:

Object get() : returns the value of thread-local variable associated with current thread.

Object initialValue() : returns initial value of ThreadLocal variable associated with current thread. (default value null)

void set(Object newValue) :set new value of ThreadLocal variable associated with current thread.

void remove(): remove the value.associated with current thread.

Semaphore :

Is used to controls access to shared resources by using permits.

If permits is greater than 0,

then semaphore allow access to shared resources.

If permits is 0 or less than 0,

then semaphore does not access to shared resouces.

These permits are sort of counters, which allow access to the shared resouces.

Semaphore(int permit) : number of permits available.

Semaphore(int permit, boolean fair)

CounterDownLatch : Its allowed one or more thread to wait for a given set of operation to complete.

Its initialized with given count.

This count is decreased by calls to countDown() method.

Threads waiting for this count reach 0 can call await() method.

Call await() blocks the thread until the count reaches zero.

CyclicBarrier :

Its a barrier that all the threads must wait, untill all the threads reach, before ant of the

threads can continue.

CounterDownLatch vs CyclicBarrier

Both CountDownLatch and CyclicBarrier are used to implement a scenario where one Thread waits for one or more Thread to complete there job before

starts processing. But There is one Difference between CountDownLatch and CyclicBarrier ,

You can not reuse same CountDownLatch instance once count reaches to zero and latch is open,

on the other hand CyclicBarrier can be reused by resetting Barrier, Once barrier is broken.