La API de simultaneidad de Java

Cristián Mateos1,2

[1] ISISTAN-UNICEN-CONICET

2Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Computación Paralela y Distribuida

Agenda

- La API de concurrencia de Java: por qué
- API de simultaneidad de Java: conceptos básicos
  - Ejecutores: grupos de subprocesos y programación
  - Futuros
  - Sincronizadores
  - Variables atómicas y colecciones concurrentes
- El marco ForkJoin

¿Por qué las utilidades de concurrencia?

- Recuerde que Java tiene primitivas de concurrencia incorporadas: esperar (), notificar () y sincronizado

... pero todos tienen defectos - Difícil de usar correctamente; fácil de usar incorrectamente - Nivel demasiado bajo para muchas aplicaciones - Puede conducir a un rendimiento deficiente si se usa incorrectamente - Omita muchas construcciones útiles de concurrencia

¿Por qué las utilidades de concurrencia?

- Recuerde que Java tiene primitivas de concurrencia incorporadas: esperar (), notificar () y sincronizado

... pero todos tienen defectos - Difícil de

usar correctamente; fácil de usar incorrectamente - Nivel demasiado bajo para muchas aplicaciones - Puede conducir a un rendimiento deficiente si se usa incorrectamente - Omita muchas construcciones útiles de concurrencia

Utilidades de concurrencia: Objetivos

Por un lado, proporcionar bloques de construcción de concurrencia eficientes, correctos y reutilizables. Por otro lado, mejore la escalabilidad, el rendimiento, la legibilidad, el mantenimiento y la seguridad de subprocesos de las aplicaciones Java simultáneas.

Bloques de construcción

- Ejecutor, ThreadPool y futuro
- Sincronizadores: Semáforos y Barreras
- Variables atómicas
  - Operaciones de comparación y configuración de bajo nivel
- colecciones concurrentes
  - BlockingQueue, SynchronousQueue, ConcurrentHashMap
- Lectura recomendada: “Java Concurrency in Practice” de Brian Goetz et al.

Ejecutor

- Estandariza la invocación asíncrona
- Separa el envío del trabajo de la política de ejecución
  - anExecutor.execute(aRunnable) no nuevo
  - Thread(aRunnable).start()
- Se admiten dos estilos de código:
  - Acciones: Ejecutables
  - Funciones: Llamables
  - También tiene gestión del ciclo de vida: p. ej., cancelación, apagado
- Ejecutor generalmente creado a través de la clase de fábrica Executors
  - Implícitamente asociado a un ThreadPoolExecutor
  - A través de ExecutorService, personaliza los métodos de apagado, ganchos antes/después, políticas de saturación, colas

Ejecutor y ExecutorService

- ExecutorService agrega la gestión del ciclo de vida a Executor

```
interfaz pública Ejecutor r void          {
  ejecutar (comando ejecutable);
}
public interface E xecutorSe r vi ce extends Executo r void shutdown      {
  ( );
   L ist <Ejecutable> shutdownNow ( ) ;
  booleano isShutdown ();
  booleano i sTe rminado ( ) ;
  boolean aw a itT e rm in at ion ( tiempo de espera        ,   unidad TimeUnit);
   prolongado / / otros métodos convenientes para enviar tareas
}
```

Métodos de fábrica de ejecutores

Ejecutores **de clase pública** {

static E xe cuto rSe r vi ce newSingleThreadedExecutor ( ); **static** E xe cuto rSe r vi ce newFixedThreadPool (int n); // ¡Cuidado con los picos de demanda y los subprocesos de larga duración! **static** E xe cuto rSe rvice newCachedThreadPool (int n); Servicio de ejecución programado **estático**

newScheduledThreadPool (int n); // versiones adicionales y métodos de utilidad }

Ejemplo sin Ejecutores: Servidor Web

- 1 hilo por mensaje recibido (sin límite en la creación de hilos)

```
clase servidor web {
  public static void principal ( String [ ] a rgs ) {
    Se rve rSocket socket = nuevo Se rve rSocket ( 8 0 ) ; while
    ( verdadero ) conexión de socket final = socket. aceptar ( ) ;
      Ejecutable r = nuevo Ejecutable ( ) {

          public void run ( ) { handleRequest ( conexión ) ; } } ; hilo nuevo
      ( r ) . comienzo ( ) ; } } public static void handleRequest (Socket
      so c ket) {. . . } }
```

Ejemplo con Ejecutores: Servidor Web

- Servidor web basado en grupos de subprocesos... ¡mejor gestión de recursos!

```
class WebServer { Executor
  pool = Executo rs . newFixedThreadPool ( 7 ) ; public static void principal
  ( String [ ] a rgs ) {
    Se rve rSocket socket = nuevo Se rve rSocket ( 8 0 ) ; while
    ( verdadero ) conexión{ de socket final = socket. aceptar ( ) ; Ejecutable
      r = new Ejecutable ( ) public void run ( ) { handleRequest (conne
      ction); } } ; piscina . ejecutar (r); } } public static void handleRequest
        (Socket so c ket) {. . . } }
```

Llamable y Futuro

- Callable es funcionalmente análogo a Runnable

interfaz llamable <V> {
  V call ( ) **lanza** E xception ;
}

- El futuro representa el resultado del cálculo asíncrono

interfaz Futuro <V> {
  V get ( ) **lanza** I nt erru pt e dE xce pt ion , E xecuciónE xcepción ;
  V get (tiempo **de** espera prolongado , unidad TimeUnit);
  booleano cancel (booleano puede yI nt erru ptIf R unning);
  booleano i sC cancelado ( ) ;
  booleano isDone ( ) ;
}

Uso de futuros

- El cliente inicia el cálculo asíncrono a través de un mensaje unidireccional
- El cliente recibe un "mango" para el resultado: un futuro
- El cliente hace otro trabajo mientras espera el resultado
- Cuando esté listo, el cliente solicita el resultado de Future, bloqueándolo si es necesario hasta que el resultado esté disponible
- El cliente usa el resultado
- **Ejemplo:** buscar un archivo grande para una cadena determinada

buscador de archivos

```
// Estoy conscientemente ignorando el manejo de iones excepto. . .

public class F ut ur eF il eSearcher { E xe cuto
  rSe r vi ce e xe cuto r = Executo rs . newFixedThreadPool (1); S tringsearch F ile (S tring
  palabra clave, S tringfile P ath) {Futu re <String > futu re = e xecu to r. su bmit (() -> {
    ch (palabra clave , } } ) ; // Haga otras cosas aquí

                                                                    ruta de archivo ) ;

    // Verifique cada 0.5 segundos //
    (¡solo por el bien de la ejemplificación!) while (! future. isDone
    ())
      Hilo . dormir (5 0 0); volver
    a girar futuro. conseguir
  ( ) ; } }
```

FuturoTarea

- Un cálculo asíncrono cancelable
- Una implementación base de Future
- Puede envolver un Callable o Runnable
  - Permite que FutureTask se envíe a un Ejecutor

```
. . .
FutureTask <Cadena > f ut ur eT ask = new FutureTask <Cadena > (
 new C allable <String > ( )
   { public Stringcall ( ) { re tu rn
     doFileSea r ch ( palabra clave , } } ) ;        ruta de archivo ) ;
   eje cutor . ejecutar (futura tarea);


. . .
```

FutureTask y subprocesos convencionales

```
FutureTask <I nt eger > tarea = new FutureTask <I nt eger >(()ÿ> {
    Sistema . afuera . println("Se calcula algo ␣complicado");  ␣  ␣
    Hilo . dormir ( 1 0 0 0 );
    vuelta vuelta 42;
});
Subproceso t 1 = nuevo Subproceso ( ( ) ÿ > {
    prueba {
        int r = tarea. conseguir ( ) ;
        Sistema . afuera . println(" El resultado␣es ␣  "  +r);
    } catch ( I nt erru pt e dE xce pt ion | E xecutionE xception e ) } );  { }

Sistema . afuera . println("t 1 wil␣a it h␣asta qu␣e el cóm␣puto esté listo"); ␣  ␣
t 1 . comienzo ( ) ;
tarea . correr ( ) ;
```

Ejemplo de Future y FutureTask: caché

```
public class Cache<K, V>
 { Map<K, Future <V>> map = new ConcurrentHashMap<K, Future <V> > ( );
  Executo re xe cuto r = Ejecutores . newFixedThreadPool (8); public V get ( clave K
  final ) { Futu re <V> f = mapa. obtener la clave ) ; // clave nullif no encontrada if ( f
    == n ull ) { C allable <V> c = new C allable <V> ( ) { public V call ( ) { // calcule el
    valor asociado con h key } } ; f = nueva TareaFutura <V> ( c ) ; // si no se
      encuentra la clave, ponga (clave, f) y devuelva nulo Futuro antiguo = mapa. p
      utIfA bse nt (key, f) if (antiguo == n ull ) { // de lo contrario, devuelve get (key)
                                                                                ;
        eje cutor . ejecutar (f); }
      más { f = viejo ; } } volver
    f . conseguir ( ) ; } }
```

ScheduledExecutorService

- Para tareas diferidas y recurrentes, puede programar
  - Llamable o Ejecutable para ejecutarse una vez con un retraso fijo después del envío
  - Programe un Runnable para que se ejecute periódicamente a una tasa fija
  - Programe un Runnable para que se ejecute periódicamente con un retraso fijo entre ejecuciones
- El envío devuelve un identificador ScheduledFutureTask que se puede utilizado para cancelar la tarea
- Como Timer, pero admite más subprocesos, agrupación y control de subprocesos más fino, por lo que es más versátil

Sincronizadores

- Semáforo: semáforo de conteo de Dijkstra, gestión de cierto número de permisos
- CountDownLatch: permite que uno o más subprocesos esperen un conjunto de hilos para completar una acción
- CyclicBarrier: permite que un conjunto de subprocesos espere hasta que todos alcancen un punto de barrera específico
- Intercambiador: permite que dos subprocesos se reúnan e intercambien datos, como intercambiar un búfer vacío por uno lleno

Semáforos

- Mantener un conjunto lógico de permisos
- adquirir () bloquea hasta que un permiso está libre, luego lo toma
- release () agrega un permiso, liberando un adquirente de bloqueo
- A menudo se usa para restringir la cantidad de subprocesos que pueden acceder algún recurso
    - Pero se puede usar para implementar muchos núcleos de sincronización, patrones y primitivos.

pestillo de cuenta regresiva

- Las variables de enganche son condiciones que una vez establecidas nunca cambian
- A menudo se usa para iniciar varios subprocesos, mientras se les pide que esperen una señal antes de continuar.

Principal :

CountDownLatch start S ignal = **new** CountDownLatch (1) ; CountDownLatch st o
pS ignal = **new** CountDownLatch (COUNT); Se r vi ce r e x e cuto r e xe cuto r =
Ejecutores . newFixedThreadPool (COUNT); **for** (int i = 0; i < COUNT; i ++) { e xecutor . ejecutar
(new Worker (start S ignal, stopSignal } e xecutor . shutdown ( ) ; // Hacer otras cosas aquí . . .
, i)) ;

Señal de inicio. cuenta regresiva ( ) ;
intente {detener la señal pS. awa it
( ) ;
} **catch** (I nt erru pt e dE xce pt ion ex) { ej . impri ntSt
ack Traza ( ) ; }

CountDownLatch (continuación)

Trabajador :

```
public void ejecutar ( )          {
  probar {
    // esperar hasta que el pestillo haya contado hasta cero
    Señal de inicio. awa it ( ) ;
  } catch ( I nt erru pt e dExce pt ion ex ) ex . impri ntSt       {
    ack Traza ( ) ;
  }
  Sistema . afuera . println(" En ejecución: señal ␣"     + nombre);
  de detención. cuenta regresiva ();
}
```

Barrera cíclica

- Permite que los subprocesos esperen en un punto de barrera común
- Útil cuando un grupo de subprocesos de tamaño fijo debe esperar ocasionalmente el uno al otro
- Se puede reutilizar después de que se liberen los subprocesos.
- Puede ejecutar un Runnable una vez por punto de barrera
  - Después de que llegue el último subproceso, pero antes de que se publique alguno
  - Útil para actualizar el estado compartido antes de que continúen los hilos

Barrera cíclica: Ejemplo

barrera cíclica final = **nueva barrera** cíclica (3) ; Se r vi ce r e x e cuto r
e xe cuto r = Ejecutores . newFixedThreadPool (3); **for** (int i = 0; i < 3; i ++) { e
xecutor . ejecutar (**nuevo** R unnable ( ) { **public void** run ( ) { try { barrera . esperar ( ) ; log (" Al
esperar ( ) ; log (" **Esperar** a que finalice ") ; } ejecutor . apagar ( ) ;

{ mi . impri ntSt ack Traza ( ) ; }

intercambiador

- Punto de sincronización donde dos hilos intercambian objetos
- SynchronousQueue bidireccional
- Cada subproceso presenta algún objeto al ingresar al método exchange() y recibe el objeto presentado por el otro subproceso en devolver
- Ejemplo: dos subprocesos que llenan y vacían dos búfer diferentes

Intercambiador (continuación)

- intercambio (V x): espera que otro hilo llegue al punto de intercambio e intercambie objetos con ese hilo. intercambio (V x, tiempo de espera largo,
- unidad de unidad de tiempo): espera otro subproceso durante el intervalo de tiempo específico proporcionado en el método e intercambia el objeto con ese subproceso.