

Investigación sobre ThreadLocal

Asignatura: Programación de servicios y procesos

Alumnos: Tingmei Huang (Sophia), Óscar Gómez, Gema García

Indice

| | |
|--|-----------|
| Introducción..... | Página 3 |
| 1.Explicamos el concepto..... | Página 3 |
| 2.Casos de usos..... | Página 7 |
| 3.Ventajas e inconvenientes..... | Página 9 |
| 4. Método ThreadLocal..... | Página 11 |
| 5.Realizamos una pequeña prueba de concepto..... | Página 13 |
| Referencias..... | Página 18 |

Uso de ThreadLocal en Java

Introducción

Para esta investigación, se ha llevado a cabo una comparativa entre varias inteligencias artificiales, como *ChatGPT* y *Bard*, al igual que se ha consultado *Baeldung* para completar la información.

1.Explicamos el concepto

CHATGPT

Concepto:

ThreadLocal es una clase en Java que proporciona almacenamiento de datos separado para cada hilo. En otras palabras, te permite mantener variables específicas para cada hilo de ejecución en un programa. Cuando usas **ThreadLocal**, cada hilo tiene su propia copia independiente de la variable almacenada, lo que significa que cada hilo puede acceder y modificar su propia versión de la variable sin afectar a las copias almacenadas en otros hilos.

Este mecanismo es útil cuando tienes datos que son específicos para cada hilo y no deseas compartirlos directamente o no quieres preocuparte por problemas de concurrencia al acceder a esos datos desde diferentes hilos. Un ejemplo común es el uso de **ThreadLocal** en aplicaciones web para almacenar información de solicitud HTTP, como el usuario actual o la conexión de base de datos, de manera que cada hilo que maneja una solicitud tenga su propia copia de estos datos sin interferir con otras solicitudes que se están manejando al mismo tiempo.

¿Viene incorporado en el framework, necesitamos una librería externa?

ThreadLocal es una clase incorporada en el paquete **java.lang** en Java, por lo que no se necesita una librería externa para utilizarla. Aquí tienes un ejemplo simple que prueba su funcionalidad:

```
MainPruebas.java x
1 package Pruebas1;
2
3 public class MainPruebas {
4
5     2 usages
6     private static ThreadLocal<Integer> threadLocal = new ThreadLocal<>();
7
8     public static void main(String[] args) {
9         // Establecemos un valor en el ThreadLocal para el hilo principal
10        threadLocal.set(42);
11
12        // Creamos e iniciamos un nuevo hilo
13        Thread thread = new Thread(() -> {
14        });
15        thread.start();
16
17        // Obtenemos y mostramos el valor del ThreadLocal del hilo principal
18        int mainThreadValue = threadLocal.get();
19        System.out.println("Valor en el ThreadLocal del hilo principal: " + mainThreadValue);
20    }
21 }
```

MainPruebas x

C:\Users\oscar\.jdk\openjdk-20.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Com
Valor en el ThreadLocal del hilo principal: 42

En este código se demuestra que ThreadLocal no necesita de librerías externas, se establece un método Set para el threadLocal, luego se crea el hilo mediante Lambda y finalmente mediante el método get se muestra en pantalla el valor incluido anteriormente.

BARD:

Concepto:

En Java, un ThreadLocal es una variable que está asociada a un hilo específico. Esto significa que cada hilo tiene su propia copia de la variable, y las copias de las variables ThreadLocal no son accesibles desde otros hilos.

La clase ThreadLocal proporciona los métodos set() y get() para establecer y obtener el valor de una variable ThreadLocal. El método set() establece el valor de la variable para el hilo actual, y el método get() devuelve el valor de la variable para el hilo actual.

Objetivo de ThreadLocal:

- Permite almacenar datos de forma que cada hilo tenga su propia copia independiente, sin afectar a otras copias de otros hilos.

¿Viene incorporado en el framework, necesitamos una librería externa?

Sí, la clase ThreadLocal viene incorporada en el framework de Java. No es necesario utilizar una librería externa para utilizar ThreadLocals.

La clase ThreadLocal se encuentra en el paquete java.lang. Se puede utilizar directamente en cualquier aplicación Java.



```
Java

import java.lang.ThreadLocal;

public class ThreadLocalExample {

    public static void main(String[] args) {
        // Comprobar si la clase ThreadLocal existe
        if (ThreadLocal.class.isPresent()) {
            System.out.println("La clase ThreadLocal está presente");
        } else {
            System.out.println("La clase ThreadLocal no está presente");
        }
    }
}

Utiliza el código con precaución. Más información
```

Como puede verse en este código , que comprueba si ThreadLocal está dentro del framework, se encuentra ya implementado dentro de java.

BAELDUNG

Concepto:

ThreadLocal sirve para manejar información que solo va a ser accesible para un mismo THREAD y convierte en proceso en *Thread safe*. Esto significa que mantiene la información guardada de manera más segura. Complementada con una interfaz, *ThreadPool*, puede ser utilizada para guardar los threads al terminar su trabajo, hasta que se les llame de nuevo. Se debe limpiar bien el Pool de la información almacenada para que no haya problemas al reutilizar dicho thread.

La manera de implementarlo es la siguiente: primero, se toma prestado un thread del Pool. Segundo, se almacena la información que va a utilizar en un *ThreadLocal*. Al terminar su

ejecución, el thread vuelve al Pool hasta que la aplicación lo llame de nuevo para realizar otro proceso.

2.Casos de usos:

CHATGPT

- **Gestión de Sesiones en Aplicaciones Web:**
 - En aplicaciones web, se puede utilizar ThreadLocal para almacenar información de sesión del usuario de manera segura. Cada hilo que maneja una solicitud web tiene su propia copia de los datos de sesión, evitando problemas de concurrencia.
- **Transacciones en Bases de Datos:**
 - En entornos de bases de datos, ThreadLocal puede utilizarse para almacenar objetos de conexión a la base de datos o transacciones. Cada hilo puede tener su propia conexión, evitando así problemas de concurrencia en el acceso a la base de datos.
- **Manejo de Contexto en Hilos:**
 - Algunas aplicaciones requieren que ciertos datos estén disponibles en todos los componentes de la aplicación sin pasar explícitamente esos datos como parámetros. ThreadLocal puede utilizarse para mantener un contexto específico de un hilo, como la información del usuario o la configuración de la aplicación.
- **Evitar Problemas de Concurrencia:**
 - Al usar ThreadLocal, se evita la necesidad de sincronización explícita en algunas situaciones. Cada hilo tiene su propia instancia de la variable, por lo que no hay interferencia entre los hilos.
- **Optimización de Recursos:**
 - En situaciones en las que se necesita una instancia costosa de un objeto, ThreadLocal puede ayudar a optimizar el rendimiento al garantizar que cada hilo tenga su propia instancia y no sea necesario recrearla cada vez que se accede.

BARD

Los ThreadLocals son una herramienta poderosa en Java que permiten asociar datos a un hilo específico. Esto puede ser útil en una variedad de casos, como:

- **Autenticación y autorización:** Los ThreadLocals se pueden usar para almacenar información de autenticación y autorización, como el nombre de usuario y el token de acceso. Esto permite que cada hilo tenga su propio estado de autenticación, lo que es importante para la seguridad.
- **Estado de la aplicación:** Los ThreadLocals se pueden usar para almacenar el estado de la aplicación, como la configuración o el estado de una transacción. Esto permite

que cada hilo tenga su propio estado de la aplicación, lo que puede ser útil para la eficiencia y la consistencia.

- **Contexto de ejecución:** Los ThreadLocals se pueden usar para almacenar el contexto de ejecución, como el nombre del hilo o el ID de la transacción. Esto permite que cada hilo tenga su propio contexto de ejecución, lo que puede ser útil para el seguimiento y la depuración.

Aquí hay algunos ejemplos específicos de cómo se pueden usar los ThreadLocals:

- Un servidor web puede usar ThreadLocals para almacenar la información de sesión para cada usuario. Esto permite que cada usuario tenga su propia sesión, lo que es importante para la seguridad y la privacidad.
- Un sistema de bases de datos puede usar ThreadLocals para almacenar la conexión a la base de datos para cada hilo. Esto permite que cada hilo tenga su propia conexión a la base de datos, lo que puede mejorar el rendimiento y la escalabilidad.
- Un framework de desarrollo web puede usar ThreadLocals para almacenar el estado de la solicitud para cada solicitud. Esto permite que cada solicitud tenga su propio estado, lo que puede facilitar el desarrollo de aplicaciones web.

Es importante tener en cuenta que los ThreadLocals pueden tener un impacto en el rendimiento, ya que requieren que el hilo mantenga una referencia a la variable ThreadLocal. Por lo tanto, es importante usar ThreadLocals solo cuando sea necesario.

Aquí hay algunos consejos para usar ThreadLocals de manera efectiva:

- **Evite usar ThreadLocals para almacenar datos grandes o complejos.**
- **Use ThreadLocals solo cuando sea necesario.**
- **Limpie las variables ThreadLocal cuando ya no sean necesarias.**

3.Ventajas e inconvenientes:

VENTAJAS

CHATGPT

ThreadLocal en Java es una clase que proporciona almacenamiento de variables con ámbito de hilo. Cada hilo que accede a un objeto **ThreadLocal** tiene su propia copia independiente de la variable almacenada. Aquí hay algunas ventajas e inconvenientes de usar **ThreadLocal**:

Ventajas:

1. **Separación de contexto por hilo:** Permite almacenar datos específicos de un hilo, lo que puede ser útil en aplicaciones con múltiples hilos, evitando la necesidad de sincronización explícita.
2. **Eficiencia en operaciones de hilos:** Al evitar la sincronización en el acceso a los datos almacenados, puede mejorar el rendimiento en comparación con el uso de estructuras de datos compartidas entre hilos.
3. **Sencillez de uso:** Su API es relativamente simple, lo que facilita su implementación para almacenar y recuperar datos específicos del hilo.

Inconvenientes:

1. **Posible pérdida de memoria:** Si no se gestiona correctamente, el uso de **ThreadLocal** puede llevar a fugas de memoria debido a que los datos almacenados están asociados con cada hilo y pueden persistir más tiempo del esperado.
2. **Dificultad para rastrear y depurar:** La utilización de variables **ThreadLocal** puede complicar la comprensión del flujo de datos en aplicaciones, especialmente en aquellas donde se utilizan extensivamente y en contextos complejos.
3. **Potencial para mal uso:** Su facilidad de uso puede llevar a un mal uso, como almacenar datos globales de manera innecesaria, lo que puede conducir a problemas de mantenimiento y legibilidad del código.

En resumen, **ThreadLocal** es una herramienta poderosa para manejar datos específicos del hilo en aplicaciones multihilo, pero su uso debe ser cuidadoso y considerado para evitar problemas de rendimiento, mantenibilidad y posibles fugas de memoria.

BARD

VENTAJAS:

- Permite almacenar datos privados de cada hilo. Esto puede ser útil para almacenar datos que deben ser accesibles solo por un hilo específico, como la información de sesión de un usuario o el estado de un proceso.
- Mejora el rendimiento. El acceso a los datos almacenados en ThreadLocal es más eficiente que el acceso a los datos globales, ya que no requiere que el hilo los busque en la memoria compartida.
- Simplifica el código. El uso de ThreadLocal elimina la necesidad de pasar datos entre hilos, lo que puede simplificar el código.

INCONVENIENTES:

- Puede causar fugas de memoria. Si un hilo no elimina los datos almacenados en ThreadLocal, estos datos se quedarán en la memoria y pueden causar fugas de memoria.
- Puede ser difícil de depurar. Si un hilo no puede acceder a los datos almacenados en ThreadLocal, puede ser difícil depurar el problema.

4. Método ThreadLocal:

Method Summary

| All Methods | Static Methods | Instance Methods | Concrete Methods |
|---------------------------|----------------|--|--|
| Modifier and Type | | Method and Description | |
| T | | <code>get()</code> | Returns the value in the current thread's copy of this thread-local variable. |
| protected T | | <code>initialValue()</code> | Returns the current thread's "initial value" for this thread-local variable. |
| void | | <code>remove()</code> | Removes the current thread's value for this thread-local variable. |
| void | | <code>set(T value)</code> | Sets the current thread's copy of this thread-local variable to the specified value. |
| static <S> ThreadLocal<S> | | <code>withInitial(Supplier<? extends S> supplier)</code> | Creates a thread local variable. |

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

1. `get()`:

- **Descripción:** Este método retorna el valor en la copia actual del hilo de esta variable local del hilo.
- **Uso común:** Se utiliza para obtener el valor asociado con la variable local del hilo actual.
- **Ejemplo:**

java


 Copy code

```
ThreadLocal<String> threadLocalVariable = new ThreadLocal<>();  
String valor = threadLocalVariable.get();
```

2. `initialValue()`:

- **Descripción:** Este método devuelve el "valor inicial" actual del hilo para esta variable local del hilo.
- **Uso común:** Puede ser sobrescrito para proporcionar un valor predeterminado al inicializar la variable local del hilo.
- **Ejemplo:**

java

 Copy code

```
ThreadLocal<String> threadLocalVariable = new ThreadLocal<String>() {  
    @Override  
    protected String initialValue() {  
        return "Valor predeterminado";  
    }  
};
```

3. `remove()`:

- **Descripción:** Elimina el valor asociado con la variable local del hilo actual.
- **Uso común:** Se utiliza para limpiar la referencia de la variable local del hilo, generalmente después de que la tarea asociada al hilo ha terminado.
- **Ejemplo:**

java


 Copy code

```
threadLocalVariable.remove();
```

4. `set(T value)`:

- **Descripción:** Establece la copia actual de la variable local del hilo actual con el valor especificado.
- **Uso común:** Se utiliza para asignar un valor a la variable local del hilo actual.
- **Ejemplo:**

java

 Copy code

```
threadLocalVariable.set("Nuevo valor");
```

5. `static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier)`:

- **Descripción:** Crea una variable local del hilo utilizando un proveedor (Supplier) para especificar el "valor inicial" del hilo actual.
- **Uso común:** Proporciona una forma de inicializar la variable local del hilo con un valor específico.
- **Ejemplo:**

```
ThreadLocal<String> threadLocalVariable = ThreadLocal.withInitial(() → "Valor inicial");
```

5.Realizamos una pequeña prueba de concepto:

Sin usar Thread safe

```
public class EjemploThreadLocal {

    // Variable global compartida por todos los hilos
    private static int variableGlobal = 0;

    public static void main(String[] args) {
        // Hilo 1
        Thread hilo1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                variableGlobal++; // Incrementar la variable global
                System.out.println("Hilo 1 - Valor: " + variableGlobal);
            }
        });

        // Hilo 2
        Thread hilo2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                variableGlobal--; // Decrementar la variable global
                System.out.println("Hilo 2 - Valor: " + variableGlobal);
            }
        });

        // Iniciar hilos
        hilo1.start();
        hilo2.start();
    }
}
```

```

// Declara una instancia de ThreadLocal para almacenar la variable local
2 usages
private static ThreadLocal<Integer> threadLocalVariable = new ThreadLocal<>();

public static void main(String[] args) {
    // Crear varios hilos
    for (int i = 0; i < 5; i++) {
        new Thread(() -> {
            // Generar un valor aleatorio para cada hilo
            int randomValue = (int) (Math.random() * 100);

            // Establecer la variable local para el hilo actual
            threadLocalVariable.set(randomValue);

            // Imprimir el valor local asociado al hilo actual
            System.out.println("Hilo " + Thread.currentThread().getId() + ": Variable Local = " + threadLocalVariable.get());

            // No es necesario limpiar la variable local, ya que cada hilo tiene su propia instancia
        }).start();
    }
}

```

En este ejemplo:

Declaramos una instancia de **ThreadLocal** llamada **threadLocalVariable** para almacenar una variable local por cada hilo.

En el bucle que crea varios hilos, cada hilo genera un valor aleatorio y lo establece en su propia variable local a través del método **set()** de **ThreadLocal**.

Dentro del hilo, podemos acceder a su variable local en cualquier momento a través del método **get()** de **ThreadLocal**.

No es necesario limpiar la variable local al final del hilo, ya que cada hilo tiene su propia instancia de **ThreadLocal** y, por lo tanto, su propia variable local.

Este ejemplo ilustra cómo cada hilo tiene su propia copia de la variable local almacenada en **ThreadLocal**, y los cambios en una copia no afectan las copias de otras hebras.

ThreadLock

```
public class EjemploThreadLock {  
  
    private static int contador = 0;  
    private static Lock lock = new ReentrantLock();  
  
    public static void main(String[] args) {  
        // Crear varios hilos que incrementan el contador  
        for (int i = 0; i < 5; i++) {  
            Thread hilo = new Thread(() -> {  
                for (int j = 0; j < 10000; j++) {  
                    incrementarContador();  
                }  
            });  
            hilo.start();  
        }  
  
        // Esperar a que todos los hilos terminen  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
  
        // Imprimir el valor final del contador  
        System.out.println("Valor final del contador: " + contador);  
    }  
  
    private static void incrementarContador() {  
        // Adquirir el bloqueo antes de realizar la operación crítica  
        lock.lock();  
        try {  
            contador++;  
        } finally {  
            // Asegurarse de liberar el bloqueo, incluso si ocurre una excepción  
            lock.unlock();  
        }  
    }  
}
```

Explicación detallada:

Se importa la clase ReentrantLock que implementa la interfaz Lock.

Se crea un objeto ReentrantLock llamado lock.

En el método incrementarContador(),

lock.lock() adquiere el bloqueo antes de incrementar el contador.

try y finally aseguran que el bloqueo se libere con lock.unlock() incluso si ocurre una excepción durante la ejecución del código dentro del bloque try.

Incrementamos el contador dentro del bloque crítico, asegurando que solo un hilo puede realizar esta operación a la vez.

En resumen, ReentrantLock es utilizado para proporcionar exclusión mutua en secciones críticas del código, asegurando que solo un hilo a la vez pueda ejecutar esa sección crítica.

Método que utiliza hilos con ThreadLocal y ThreadPool

```
public class EjemploThreadPoolConThreadLocal {  
  
    2 usages  
    private static ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 0);  
  
    public static void main(String[] args) {  
        // Crear un ThreadPool con un tamaño inicial de 2 hilos, máximo 5 hilos,  
        // tiempo de retención de 60 segundos y unidad de tiempo en segundos  
        ThreadPoolExecutor executorService = new ThreadPoolExecutor(  
            corePoolSize: 2,           // corePoolSize  
            5,                         // maximumPoolSize  
            60,                       // keepAliveTime  
            TimeUnit.SECONDS,         // unit  
            new LinkedBlockingQueue<>() // workQueue  
        );  
  
        // Lanza tareas en el ThreadPool  
        for (int i = 0; i < 10; i++) {  
            executorService.execute(() -> {  
                // Cada hilo obtiene su propio valor único a través de ThreadLocal  
                int valor = threadLocalValue.get();  
                System.out.println("Hilo: " + Thread.currentThread().getName() +  
                    ", Valor ThreadLocal: " + valor);  
  
                // Incrementa el valor para la próxima vez que este hilo acceda a ThreadLocal  
                threadLocalValue.set(valor + 1);  
            });  
        }  
  
        // Mostrar información sobre el estado del ThreadPool y la cola de tareas  
        System.out.println("Tamaño actual del ThreadPool: " + executorService.getPoolSize());  
        System.out.println("Número de hilos en ejecución: " + executorService.getActiveCount());  
        System.out.println("Tamaño de la cola de tareas: " + executorService.getQueue().size());  
  
        // Apagar el ThreadPool después de que todas las tareas han sido procesadas  
        executorService.shutdown();  
    }  
}
```

Ahora, explicaremos cada parte del código:

ThreadLocal<Integer> threadLocalValue:

- Se declara un objeto ThreadLocal para almacenar valores específicos de cada hilo. En este caso, se usa para almacenar un entero que representa un valor único para cada hilo

withInitial(() -> 0): Inicializa el valor de la variable para cada hilo. La expresión lambda () -> 0 indica que el valor inicial será 0. Esto significa que cada hilo obtendrá su propia copia de threadLocalValue con un valor inicial de 0.

Creación del ThreadPoolExecutor:

- `ThreadPoolExecutor` es una implementación de `ExecutorService` que proporciona un `ThreadPool`. Se configura con:
 - `corePoolSize`: 2, es el tamaño inicial del `ThreadPool`.
 - `maximumPoolSize`: 5, es el límite máximo de hilos.
 - `keepAliveTime`: 60, es el tiempo de retención para hilos inactivos.
 - `TimeUnit.SECONDS`: Indica que el tiempo de retención se mide en segundos.
 - `new LinkedBlockingQueue<>`: Es una cola de tareas (colocada entre el productor y el consumidor) que almacena tareas pendientes.

Lanzamiento de Tareas:

- Un bucle `for` lanza 10 tareas en el `ThreadPool` mediante el método `execute`. Cada tarea es representada por una `lambda` que utiliza `ThreadLocal` para obtener y actualizar un valor específico de cada hilo.

Mostrar Información del ThreadPool:

- Después de lanzar las tareas, se imprime información sobre el estado del `ThreadPool` y la cola de tareas:
 - `executorService.getPoolSize()`: Tamaño actual del `ThreadPool`.
 - `executorService.getActiveCount()`: Número de hilos en ejecución.
 - `executorService.getQueue().size()`: Tamaño de la cola de tareas.

Apagar el ThreadPool:

- Se utiliza `executorService.shutdown()` para apagar el `ThreadPool` después de que todas las tareas han sido procesadas.

Este código simula la ejecución de tareas concurrentemente en un `ThreadPool` y muestra información sobre su estado. Además, utiliza `ThreadLocal` para asignar valores específicos a cada hilo en el contexto del `ThreadPool`.

Un `ThreadPool` es una colección de hilos reutilizables que ejecutan tareas concurrentemente. Sus ventajas incluyen la gestión automática de hilos, control del número de hilos, manejo de tareas excedentes, optimización de rendimiento y simplificación de la programación concurrente. Mejora la eficiencia al evitar la creación constante de hilos y facilita el desarrollo de aplicaciones concurrentes.

Referencias

Baeldung, & Baeldung. (2023, 6 mayo). *An introduction to ThreadLocal in Java* | Baeldung. Baeldung. <https://www.baeldung.com/java-threadlocal>

ChatGPT | <https://chat.openai.com/>

Dónde puedes utilizar Bard - Ayuda de Bard. (s. f.)https://support.google.com/bard/answer/13575153?hl=es&ref_topic=13194540&sjid=8751149329986779881-EU

Api java:

<https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html#afterExecute-java.lang Runnable-java.lang Throwable->