Java 并发 - ThreadLocal详解

ThreadLocal是通过线程隔离的方式防止任务在共享资源上产生冲突, 线程本地存储是一种自动化机制，可以为使用相同变量的每个不同线程都创建不同的存储。

面试问题去理解

- 什么是ThreadLocal? 用来解决什么问题的?
- 说说你对ThreadLocal的理解
- ThreadLocal是如何实现线程隔离的?
- 为什么ThreadLocal会造成内存泄露? 如何解决
- 还有哪些使用ThreadLocal的应用场景?

ThreadLocal简介

线程安全(是指广义上的共享资源访问安全性，因为线程隔离是通过副本保证本线程访问资源安全性，它不保证线程之间还存在共享关系的狭义上的安全性)的解决思路：

- 互斥同步: synchronized 和 ReentrantLock
- 非阻塞同步: CAS, AtomicXXXX
- 无同步方案: 栈封闭，本地存储(Thread Local)，可重入代码

本地存储(Thread Local)。官网的解释是这样的：

> This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its {@code get} or {@code set} method) has its own, independently initialized copy of the variable. {@code ThreadLocal} instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID) 该类提供了线程局部 (thread-local) 变量。这些变量不同于它们的普通对应物，因为访问某个变量(通过其 get 或 set 方法)的每个线程都有自己的局部变量，它独立于变量的初始化副本。ThreadLocal 实例通常是类中的 private static 字段，它们希望将状态与某一个线程(例如，用户 ID 或事务 ID)相关联。

总结而言：ThreadLocal是一个将在多线程中为每一个线程创建单独的变量副本的类; 当使用ThreadLocal来维护变量时, ThreadLocal会为每个线程创建单独的变量副本, 避免因多线程操作共享变量而导致的数据不一致的情况。

ThreadLocal理解

> 提到ThreadLocal被提到应用最多的是session管理和数据库链接管理，这里以数据访问为例帮助你理解ThreadLocal：

- 如下数据库管理类在单线程使用是没有任何问题的

```
class ConnectionManager {
```

```
    private static Connection connect = null;

    public static Connection openConnection() {
        if (connect == null) {
            connect = DriverManager.getConnection();
        }
        return connect;
    }

    public static void closeConnection() {
        if (connect != null)
            connect.close();
    }
}
```

很显然，在多线程中使用会存在线程安全问题：第一，这里面的2个方法都没有进行同步，很可能在openConnection方法中会多次创建connect；第二，由于connect是共享变量，那么必然在调用connect的地方需要使用到同步来保障线程安全，因为很可能一个线程在使用connect进行数据库操作，而另外一个线程调用closeConnection关闭链接。

- 为了解决上述线程安全的问题，第一考虑：互斥同步

你可能会说，将这段代码的两个方法进行同步处理，并且在调用connect的地方需要进行同步处理，比如用Synchronized或者ReentrantLock互斥锁。

- 这里再抛出一个问题：这地方到底需不需要将connect变量进行共享?

事实上，是不需要的。假如每个线程中都有一个connect变量，各个线程之间对connect变量的访问实际上是没有依赖关系的，即一个线程不需要关心其他线程是否对这个connect进行了修改的。即改后的代码可以这样：

```
class ConnectionManager {
    private Connection connect = null;

    public Connection openConnection() {
        if (connect == null) {
            connect = DriverManager.getConnection();
        }
        return connect;
    }

    public void closeConnection() {
        if (connect != null)
            connect.close();
    }
}

class Dao {
    public void insert() {
        ConnectionManager connectionManager = new ConnectionManager();
        Connection connection = connectionManager.openConnection();

        // 使用connection进行操作

        connectionManager.closeConnection();
    }
}
```

这样处理确实也没有任何问题，由于每次都是在方法内部创建的连接，那么线程之间自然不存在线程安全问题。但是这样会有一个致命的影响：导致服务器压力非常大，并且严重影响程序执行性能。由于在方法中需要频繁地开启和关闭数据库连接，这样不仅严重影响程序执行效率，还可能导致服务器压力巨大。

- 这时候ThreadLocal登场了

那么这种情况下使用ThreadLocal是再适合不过的了，因为ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。下面就是网上出现最多的例子：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionManager {

    private static final ThreadLocal<Connection> dbConnectionLocal = new ThreadLocal<Connection>
() {
        @Override
        protected Connection initialValue() {
            try {
                return DriverManager.getConnection("", "", "");
            } catch (SQLException e) {
                e.printStackTrace();
            }
            return null;
        }
    };

    public Connection getConnection() {
        return dbConnectionLocal.get();
    }
}
```

- 再注意下ThreadLocal的修饰符

ThreaLocal的JDK文档中说明：ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread。如果我们希望通过某个类将状态(例如用户ID、事务ID)与线程关联起来，那么通常在这个类中定义private static类型的ThreadLocal 实例。

> 但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。

ThreadLocal原理

如何实现线程隔离

主要是用到了Thread对象中的一个ThreadLocalMap类型的变量threadLocals, 负责存储当前线程的关于Connection的对象, dbConnectionLocal(以上述例子中为例) 这个变量为Key, 以新建的Connection对象为Value; 这样的话, 线程第一次读取的时候如果不存在就会调用ThreadLocal的initialValue方法创建一个Connection对象并且返回;

具体关于为线程分配变量副本的代码如下:

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

- 首先获取当前线程对象t, 然后从线程t中获取到ThreadLocalMap的成员属性threadLocals
- 如果当前线程的threadLocals已经初始化(即不为null) 并且存在以当前ThreadLocal对象为Key的值, 则直接返回当前线程要获取的对象(本例中为Connection);
- 如果当前线程的threadLocals已经初始化(即不为null)但是不存在以当前ThreadLocal对象为Key的的对象, 那么重新创建一个Connection对象, 并且添加到当前线程的threadLocals Map中,并返回
- 如果当前线程的threadLocals属性还没有被初始化, 则重新创建一个ThreadLocalMap对象, 并且创建一个Connection对象并添加到ThreadLocalMap对象中并返回。

如果存在则直接返回很好理解, 那么对于如何初始化的代码又是怎样的呢?

```
private T setInitialValue() {
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

- 首先调用我们上面写的重载过后的initialValue方法, 产生一个Connection对象
- 继续查看当前线程的threadLocals是不是空的, 如果ThreadLocalMap已被初始化, 那么直接将产生的对象添加到ThreadLocalMap中, 如果没有初始化, 则创建并添加对象到其中;

同时, ThreadLocal还提供了直接操作Thread对象中的threadLocals的方法

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

这样我们也可以不实现initialValue, 将初始化工作放到DBConnectionFactory的getConnection方法中:

```
public Connection getConnection() {
    Connection connection = dbConnectionLocal.get();
    if (connection == null) {
        try {
            connection = DriverManager.getConnection("", "", "");
            dbConnectionLocal.set(connection);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    return connection;
}
```

那么我们看过代码之后就很清晰的知道了为什么ThreadLocal能够实现变量的多线程隔离了; 其实就是用了Map的数据结构给当前线程缓存了, 要使用的时候就从本线程的threadLocals对象中获取就可以了, key就是当前线程;

当然了在当前线程下获取当前线程里面的Map里面的对象并操作肯定没有线程并发问题了, 当然能做到变量的线程间隔离了;

现在我们知道了ThreadLocal到底是什么了, 又知道了如何使用ThreadLocal以及其基本实现原理了是不是就可以结束了呢? 其实还有一个问题就是ThreadLocalMap是个什么对象, 为什么要用这个对象呢?

*ThreadLocalMap*对象是什么

本质上来讲, 它就是一个Map, 但是这个ThreadLocalMap与我们平时见到的Map有点不一样

- 它没有实现Map接口;
- 它没有public的方法, 最多有一个default的构造方法, 因为这个ThreadLocalMap的方法仅仅在ThreadLocal类中调用, 属于静态内部类
- ThreadLocalMap的Entry实现继承了WeakReference<ThreadLocal<?>>
- 该方法仅仅用了一个Entry数组来存储Key, Value; Entry并不是链表形式, 而是每个bucket里面仅仅放一个Entry;

要了解ThreadLocalMap的实现, 我们先从入口开始, 就是往该Map中添加一个值:

```
private void set(ThreadLocal<?> key, Object value) {

    // We don't use a fast path as with get() because it is at
    // least as common to use set() to create new entries as
    // it is to replace existing ones, in which case, a fast
    // path would fail more often than not.

    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);

    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get();

        if (k == key) {
            e.value = value;
            return;
        }

        if (k == null) {
```

```
            replaceStaleEntry(key, value, i);
            return;
        }
    }

    tab[i] = new Entry(key, value);
    int sz = ++size;
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash();
}
```

先进行简单的分析, 对该代码表层意思进行解读:

- 看下当前threadLocal的在数组中的索引位置 比如: `i = 2`, 看 `i = 2` 位置上面的元素(Entry)的Key是否等于threadLocal 这个 Key, 如果等于就很好说了, 直接将该位置上面的Entry的Value替换成最新的就可以了;
- 如果当前位置上面的 Entry 的 Key为空, 说明ThreadLocal对象已经被回收了, 那么就调用replaceStaleEntry
- 如果清理完无用条目(ThreadLocal被回收的条目)、并且数组中的数据大小 > 阈值的时候对当前的Table进行重新哈希 所以, 该HashMap是处理冲突检测的机制是向后移位, 清除过期条目 最终找到合适的位置;

了解完Set方法, 后面就是Get方法了:

```
private Entry getEntry(ThreadLocal<?> key) {
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];
    if (e != null && e.get() == key)
        return e;
    else
        return getEntryAfterMiss(key, i, e);
}
```

先找到ThreadLocal的索引位置, 如果索引位置处的entry不为空并且键与threadLocal是同一个对象, 则直接返回; 否则去后面的索引位置继续查找。

ThreadLocal造成内存泄露的问题

网上有这样一个例子:

```
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadLocalDemo {
    static class LocalVariable {
        private Long[] a = new Long[1024 * 1024];
    }

    // (1)
    final static ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(5, 5, 1,
TimeUnit.MINUTES,
            new LinkedBlockingQueue<>());
    // (2)
    final static ThreadLocal<LocalVariable> localVariable = new ThreadLocal<LocalVariable>();

    public static void main(String[] args) throws InterruptedException {
        // (3)
        Thread.sleep(5000 * 4);
```

```
        for (int i = 0; i < 50; ++i) {
            poolExecutor.execute(new Runnable() {
                public void run() {
                    // (4)
                    localVariable.set(new LocalVariable());
                    // (5)
                    System.out.println("use local varaible" + localVariable.get());
                    localVariable.remove();
                }
            });
        }
        // (6)
        System.out.println("pool execute over");
    }
}
```

如果用线程池来操作ThreadLocal 对象确实会造成内存泄露, 因为对于线程池里面不会销毁的线程, 里面总会存在着<ThreadLocal, LocalVariable>的强引用, 因为final static 修饰的 ThreadLocal 并不会释放, 而ThreadLocalMap 对于Key 虽然是弱引用, 但是强引用不会释放, 弱引用当然也会一直有值, 同时创建的LocalVariable对象也不会释放, 就造成了内存泄露; 如果LocalVariable对象不是一个大对象的话, 其实泄露的并不严重, 泄露的内存 = 核心线程数 * LocalVariable对象的大小;

所以, 为了避免出现内存泄露的情况, ThreadLocal提供了一个清除线程中对象的方法, 即 remove, 其实内部实现就是调用 ThreadLocalMap 的remove方法:

```
private void remove(ThreadLocal<?> key) {
    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);
    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        if (e.get() == key) {
            e.clear();
            expungeStaleEntry(i);
            return;
        }
    }
}
```

找到Key对应的Entry, 并且清除Entry的Key(ThreadLocal)置空, 随后清除过期的Entry即可避免内存泄露。

再看ThreadLocal应用场景

每个线程维护了一个“序列号”

> 再回想上文说的，如果我们希望通过某个类将状态(例如用户ID、事务ID)与线程关联起来，那么通常在这个类中定义private static类型的ThreadLocal 实例。

每个线程维护了一个“序列号”

```
public class SerialNum {
    // The next serial number to be assigned
    private static int nextSerialNum = 0;
```

```
    private static ThreadLocal serialNum = new ThreadLocal() {
        protected synchronized Object initialValue() {
            return new Integer(nextSerialNum++);
        }
    };

    public static int get() {
        return ((Integer) (serialNum.get())).intValue();
    }
}
```

*Session*的管理

经典的另外一个例子：

```
private static final ThreadLocal threadSession = new ThreadLocal();

public static Session getSession() throws InfrastructureException {
    Session s = (Session) threadSession.get();
    try {
        if (s == null) {
            s = getSessionFactory().openSession();
            threadSession.set(s);
        }
    } catch (HibernateException ex) {
        throw new InfrastructureException(ex);
    }
    return s;
}
```

在线程内部创建*ThreadLocal*

还有一种用法是在线程类内部创建ThreadLocal，基本步骤如下：

- 在多线程的类(如ThreadDemo类)中，创建一个ThreadLocal对象threadXxx，用来保存线程间需要隔离处理的对象xxx。
- 在ThreadDemo类中，创建一个获取要隔离访问的数据的方法getXxx()，在方法中判断，若ThreadLocal对象为null时候，应该new()一个隔离访问类型的对象，并强制转换为要应用的类型。
- 在ThreadDemo类的run()方法中，通过调用getXxx()方法获取要操作的数据，这样可以保证每个线程对应一个数据对象，在任何时刻都操作的是这个对象。

```
public class ThreadLocalTest implements Runnable{

    ThreadLocal<Student> StudentThreadLocal = new ThreadLocal<Student>();

    @Override
    public void run() {
        String currentThreadName = Thread.currentThread().getName();
        System.out.println(currentThreadName + " is running...");
        Random random = new Random();
        int age = random.nextInt(100);
        System.out.println(currentThreadName + " is set age: "  + age);
```

```
        Student Student = getStudentt(); //通过这个方法，为每个线程都独立的new一个Studentt对象，每个线程的的Studentt对象都可以设置不同的值
        Student.setAge(age);
        System.out.println(currentThreadName + " is first get age: " + Student.getAge());
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println( currentThreadName + " is second get age: " + Student.getAge());

    }

    private Student getStudentt() {
        Student Student = StudentThreadLocal.get();
        if (null == Student) {
            Student = new Student();
            StudentThreadLocal.set(Student);
        }
        return Student;
    }

    public static void main(String[] args) {
        ThreadLocalTest t = new ThreadLocalTest();
        Thread t1 = new Thread(t,"Thread A");
        Thread t2 = new Thread(t,"Thread B");
        t1.start();
        t2.start();
    }

}

class Student{
    int age;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

}
```

java 开发手册中推荐的 *ThreadLocal*

看看阿里巴巴 java 开发手册中推荐的 ThreadLocal 的用法:

```
import java.text.DateFormat;
import java.text.SimpleDateFormat;

public class DateUtils {
    public static final ThreadLocal<DateFormat> threadLocal = new ThreadLocal<DateFormat>(){
        @Override
        protected DateFormat initialValue() {
            return new SimpleDateFormat("yyyy-MM-dd");
        }
    };
}
```

然后我们再要用到 DateFormat 对象的地方，这样调用：

```
DateUtils.df.get().format(new Date());
```