18 线程作用域内共享变量—深入解析ThreadLocal

更新时间：2019-10-29 11:05:24

> 一个人追求的目标越高，他的才力就发展得越快，对社会就越有益。
>
> ——高尔基

本章前面三节分别讲解了线程同步的三种方式。无论是轻量级的Atomic、volatile，还是synchronized，其实都是采用同步的方式解决了线程安全问题。本节我们将介绍另外一种解决线程安全问题的思路，线程封闭。

如果你有一个全局共享的变量，那么多线程并发的时候，对这个共享变量的访问是不安全的。方法内的局部变量是线程安全的，由于每个线程都会有自己的副本。也就是说局部变量被封闭在线程内部，其它线程无法访问（引用型有所区别）。那么有没有作用域介于两者之间，既能保证线程安全，又不至于只局限于方法内部的方式呢？答案是肯定的，我们使用ThreadLocal就可以做到这一点。ThreadLocal变量的作用域是为线程，也就是说线程内跨方法共享。例如某个对象的方法A对threadLocal变量赋值，在同一个线程中的另外一个对象的方法B能够读取到该值。因为作用域为同一个线程，那么自然就是线程安全的。但是需要注意的是，如果threadLocal存储的是共享变量的引用，那么同样会有线程安全问题。

1、ThreadLocal 的使用场景

ThreadLocal的特性决定了它的使用场景。由于ThreadLocal中存储的变量是线程隔离的，所以一般在以下情况使用ThreadLocal：

1、存储需要在线程隔离的数据。比如线程执行的上下文信息，每个线程是不同的，但是对于同一个线程来说会共享同一份数据。Spring MVC的 RequestContextHolder 的实现就是使用了ThreadLocal；

2、跨层传递参数。层次划分在软件设计中十分常见。层次划分后，体现在代码层面就是每层负责不同职责，一个完整的业务操作，会由一系列不同层的类的方法调用串起来完成。有些时候第一层获得的一个变量值可能在第三层、甚至更深层的方法中才会被使用。如果我们不借助ThreadLocal，就只能一层层地通过方法参数进行传递。使用ThreadLocal后，在第一层把变量值保存到ThreadLocal中，在使用的层次方法中直接从ThreadLocal中取出，而不用作为参数在不同方法中传来传去。不过千万不要滥用ThreadLocal，它的本意并不是用来跨方法共享变量的。结合第一种情况，我们放入ThreadLocal跨层传递的变量一般也是具有上下文属性的。比如用户的信息等。这样我们在AOP处理异常或者其他操作时可以很方便地获取当前登录用户的信息。

2、如何使用 ThreadLocal

ThreadLocal使用起来非常简单，我们先看一个简单的例子。

可以看到每个线程为同一个ThreadLocal对象set不同的值，但各个线程打印出来的依旧是自己保存进去的值，并没有被其它线程所覆盖。

一般来说，在实践中，我们会把ThreadLocal对象声名为static final，作为私有变量封装到自定义的类中。另外提供static的set和get方法。如下面的代码：

```
public final class OperationInfoRecorder {

    private static final ThreadLocal<OperationInfoDTO> THREAD_LOCAL = new ThreadLocal<>();

    private OperationInfoRecorder() {
    }

    public static OperationInfoDTO get() {
        return THREAD_LOCAL.get();
    }

    public static void set(OperationInfoDTO operationInfoDTO) {
        THREAD_LOCAL.set(operationInfoDTO);
    }

    public static void remove() {
        THREAD_LOCAL.remove();
    }
}
```

这样做的目的有二:

1、static 确保全局只有一个保存OperationInfoDTO对象的ThreadLocal实例；

2、final 确保ThreadLocal的实例不可更改。防止被意外改变，导致放入的值和取出来的不一致。另外还能防止ThreadLocal的内存泄漏，具体原因下文中会有讲解。

使用的时候可以在任何方法的任何位置调用OperationInfoRecorder的set或者get方法，保存和取出。如下面代码：

```
OperationInfoRecorder.set(operationInfoDTO)
OperationInfoRecorder.get()
```

3、ThreadLocal源代码解析

学习到这里，你一定很好奇ThreadLocal是如何做到多个线程对同一个对象set操作，但只会get出自己set进去的值呢？这个现象有点违背我们的认知。接下来我们就从set方法入手，来看看ThreadLocal的源代码：

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

一眼看过去，一下就可以看到map。没错，如果ThreadLocal能够保存多个线程的变量值，那么它一定是借助容器来实现的。

这个map不是一般的map，可以看到它是通过当前线程对象获取到的ThreadLocalMap。看到这里应该看出些端倪，这个map其实是和Thread绑定的。接下来我们看getMap方法的代码：

```
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

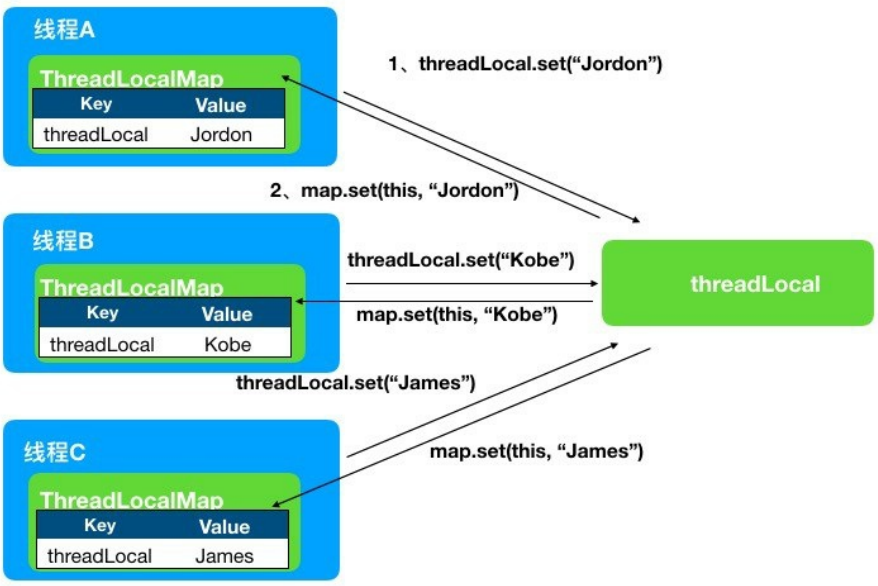
原来这个ThreadLocal就存方法Thread对象上。下面我们看看Thread中的相关代码：

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap threadLocals = null;
```

注释中写的很清楚，这个属性由ThreadLocal来维护。threadLocals的访问控制决定在包外是无法直接访问的。所以我们在使用的时候只能通过ThreadLocal对象来访问。

set时，会把当前threadLocal对象作为key，你想要保存的对象作为value，存入map。

看到这里，我们大至已经理清了ThreadLocal和Thread的关系，我们看下图：



我们接下来分析get方法，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();
}
```

get方法也是先取得当前线程对象中保存的ThreadLocalMap对象，然后使用当前threadLocal对象从map中取得相应的value。

每个Thread的ThreadMap以threadLocal作为key，保存自己线程的value副本。我们可以这么来理解ThreadLocal，其实ThreadLocal对象是你要真正保存对象的身份代表。而这个身份在每个线程中对应的值，其实是保存在每个线程中，并没有保存在ThreadLocal对象中。

这里可以举个例子，学校里要每班评选一名学习标兵，一名道德标兵。班主任会进行评选然后记录下来。学生标兵及道德标兵的身份就是两个ThreadLocal对象，而每个班主任是一个线程，记录的评选结果的小本子就是ThreadLocalMap对象。每个班主任会在自己的小本子上记录下评选结果，比如说一班班主任记录：道德标兵：小明，学习标兵：小红。二班班主任记录：道德标兵：小赵，学习标兵：小岩。通过这个例子大家应该很清楚ThreadLocal的原理了。

ThreadLocal的设计真的非常巧妙，看似自己保存了每个线程的变量副本，其实每个线程的变量副本是保存在线程对象中，那么自然就线程隔离了。如此分析起来，是不是有一种ThreadLocal没做什么事情，却抢了头功的感觉？其实不然。Thread对象中用来保存变量副本的ThreadLocalMap的定义就在ThreadLocal中。我们接下来分析ThreadLocalMap的源代码。

4、ThreadLocalMap分析

ThreadLocalMap是ThreadLocal的静态内部类，我们单独一小节来讲解它。ThreadLocalMap的功能其实是和HashMap类似的，但是为什么不直接使用HashMap呢？`在ThreadLocalMap中使用WeakReference包装后的ThreadLocal对象作为key，也就是说这里对ThreadLocal对象为弱引用。当ThreadLocal对象在ThreadLocalMap引用之外，再无其他引用的时候能够被垃圾回收。如下面代码所示：

```
static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
```

这样做会带来新的问题。如果ThreadLocal对象被回收，那么ThreadLocalMap中保存的key值就变成了null，而value会一直被Entry引用，而Entry又被threadLocalMap对象引用，threadLocalMap对象又被Thread对象所引用，那么当Thread一直不终结的话，value对象就会一直驻留在内存中，直至Thread被销毁后，才会被回收。这就是ThreadLocal引起内存泄漏问题。

而ThreadLocalMap在设计的时候也考虑到这一点，在get和set的时候，会把遇到的key为null的entry清理掉。不过这样做也不能100%保证能够清理干净。我们可以通过以下两种方式来避免这个问题：

1、把ThreadLocal对象声明为static，这样ThreadLocal成为了类变量，生命周期不是和对象绑定，而是和类绑定，延长了声明周期，避免了被回收；

2、在使用完ThreadLocal变量后，手动remove掉，防止ThreadLocalMap中Entry一直保持对value的强引用。导致value不能被回收。

4、总结

通过本节学习，我们掌握了ThreadLocal 的原理和其使用场景。绝大多数情况下，ThreadLocal用于存储和线程相关的上下文信息，也就是线程共享的信息，便于同一线程的不同方法中取值，而不用作为方法参数层层传递。

}

17 资源有限，请排队等候—Synchronized使用、原理及缺陷

19 自己动手丰衣足食—简单线程池实现