LESS IS MORE

深入理解 Java 之 ThreadLocal 工作原理

09-04, 2017

ThreadLocal这个概念很重要，面试也是经常问，由此可见大多数人不容易掌握这个知识点。其实其实现原理非常简单，简单理解“Thread”即线程，“Local”即本地。连续起来理解就是 **每个线程本地独有的**。强烈你推荐看下面的概念引入，我将叙述为什么会有这个概念的出现。

概念引入

现在的软件开发过程中，并发是很重要的手段。由此而带来的语言层面的切入点就是线程了，引入多线程开发之后，自然要考虑好同步、互斥、安全等内容。而因为这些需求就出现了以下三种来实现线程安全的手段。

- 互斥同步

简单点理解就是通过加锁来实现对临界资源的访问限制。加锁方式有Synchorized和Lock。

- 非阻塞同步

前面提到的互斥同步属于一种悲观锁机制，非阻塞同步属于乐观锁机制。典型的实现方式就是CAS操 作。

- 无同步方案

要保证线程安全，并不是一定就需要同步，两者没有因果关系，同步只是保证共享数据征用时正确性的手段，如果一个方法本来就不涉及共享数据，那它就不需要任何同步措施去保证正确性。ThreadLocal的概念就是从这里引申出来的。

示例用法

先通过下面这个实例来理解ThreadLocal的用法。先声明一个ThreadLocal对象，存储布尔类型的数值。然后分别在主线程中、Thread1、Thread2中为ThreadLocal对象设置不同的数值：

```
public class ThreadLocalDemo {
    public static void main(String[] args) {

        // 声明 ThreadLocal对象
        ThreadLocal<Boolean> mThreadLocal = new ThreadLocal<Boolean>();

        // 在主线程、子线程1、子线程2中去设置访问它的值
        mThreadLocal.set(true);

        System.out.println("Main " + mThreadLocal.get());

        new Thread("Thread#1"){
            @Override
            public void run() {
                mThreadLocal.set(false);
                System.out.println("Thread#1 " + mThreadLocal.get());
            }
        }.start();

        new Thread("Thread#2"){
            @Override
            public void run() {
                System.out.println("Thread#2 " + mThreadLocal.get());
            }
        }.start();
    }
}
```

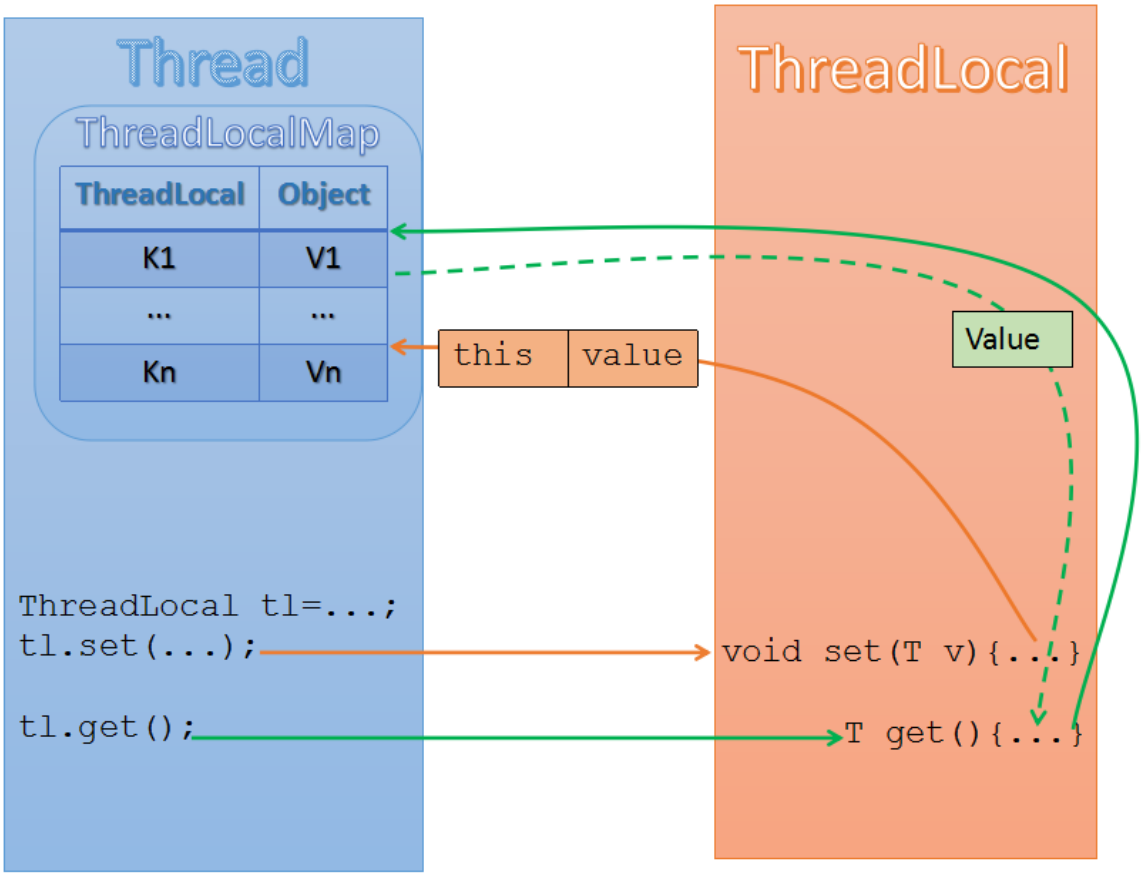
打印的结果输出如下所示：

```
MainThread true
Thread#1 false
Thread#2 null
```

可以看见，在不同线程对同一个ThreadLocal对象设置数值，在不同的线程中取出来的值不一样。接下来就分析一下源码，看看其内部结构。

结构概览

这张图我是看人家博客记住的，当时就保存在笔记本上，由于时间久远不记得出处了，在此声明仅作学习。如有不适，请告知。



清晰的看到一个线程Thread中存在一个ThreadLocalMap，ThreadLocalMap中的key对应ThreadLocal，在此处可见Map可以存储多个key即(ThreadLocal)。另外Value就对应着在ThreadLocal中存储的Value。因此总结出：每个Thread中都具备一个ThreadLocalMap，而ThreadLocalMap可以存储以ThreadLocal为key的键值对。这里解释了为什么每个线程访问同一个ThreadLocal，得到的确是不同的数值。如果此处你觉得有点突兀，接下来看源码分析吧！！！ let us go

源码分析

ThreadLocal#set

```
public void set(T value) {
        // 获取当前线程对象
        Thread t = Thread.currentThread();
        // 根据当前线程的对象获取其内部Map
        ThreadLocalMap map = getMap(t);
        // 注释1
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
    }
```

如上所示，大部分解释已经在代码中做出，注意注释1处，得到map对象之后，用的 this 作为key，this在这里代表的是当前线程的ThreadLocal对象。 另外就是第二句根据getMap获取一个ThreadLocalMap，其中getMap中传入了参数t(当前线程对象)，这样就能够获取每个线程的 ThreadLocal 了。 继续跟进到ThreadLocalMap中查看set方法：

ThreadLocalMap

ThreadLocalMap是ThreadLocal的一个内部类，在分析其set方法之前，查看一下其类结构和成员变量。

```
static class ThreadLocalMap {
        // Entry类继承了WeakReference<ThreadLocal<?>>，即每个Entry对象都有一个
  //（作为key），这是为了防止内存泄露。一旦线程结束，key变为一个不可达的对象，这
        static class Entry extends WeakReference<ThreadLocal<?>> {
            /** The value associated with this ThreadLocal. */
            Object value;
            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }
        // ThreadLocalMap 的初始容量，必须为2的倍数
        private static final int INITIAL_CAPACITY = 16;

        // resized时候需要的table
        private Entry[] table;

        // table中的entry个数
        private int size = 0;

        // 扩容数值
        private int threshold; // Default to 0
```

一起看一下其常用的构造函数：

```
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
            table = new Entry[INITIAL_CAPACITY];
            int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
            table[i] = new Entry(firstKey, firstValue);
            size = 1;
            setThreshold(INITIAL_CAPACITY);
        }
```

构造函数的第一个参数就是本ThreadLocal实例(this)，第二个参数就是要保存的线程本地变量。构造函数首先创建一个长度为16的Entry数组，然后计算出firstKey对应的哈希值，然后存储到table中，并设置size和threshold。

注意一个细节，计算hash的时候里面采用了hashCode & (size - 1)的算法，这相当于取模运算hashCode % size的一个更高效的实现（和HashMap中的思路相同）。正是因为这种算法，我们要求size必须是2的指数，因为这可以使得hash发生冲突的次数减小。

ThreadLocalMap#set

ThreadLocal中put函数最终调用了ThreadLocalMap中的set函数，跟进去看一看：

```
   private void set(ThreadLocal<?> key, Object value) {
            Entry[] tab = table;
            int len = tab.length;
            int i = key.threadLocalHashCode & (len-1);

            for (Entry e = tab[i];
                 e != null;
                 // 冲突了
                 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();
        }
```

在上述代码中如果Entry在存放过程中冲突了，调用nextIndex来处理，如下所示。是否还记得hashmap中对待冲突的处理？这里好像是另一种套路：只要i的数值小于len，就加1取值，官方术语称为：线性探测法。

```
private static int nextIndex(int i, int len) {
            return ((i + 1 < len) ? i + 1 : 0);
}
```

以上步骤ok了之后，再次关注一下源码中的cleanSomeSlots，该函数主要的作用就是清理无用的entry，具体细节就不扣了：

```
private boolean cleanSomeSlots(int i, int n) {
            boolean removed = false;
            Entry[] tab = table;
            int len = tab.length;
            do {
                i = nextIndex(i, len);
                Entry e = tab[i];
                if (e != null && e.get() == null) {
                    n = len;
                    removed = true;
                    i = expungeStaleEntry(i);
                }
            } while ( (n >>>= 1) != 0);
            return removed;
        }
```

ThreadLocal#get

看完了set函数，肯定是要关注Get的，源码如下所示：

```
public T get() {
        // 获取Thread对象t
        Thread t = Thread.currentThread();
        // 获取t中的map
        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();
    }
```

如果map为null，就返回setInitialValue()这个方法，跟进这个方法看一下：

```
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;
    }
```

最后返回的是value，而value来自 `initialValue()` ,进入这个源码中查看：

```
protected T initialValue() {
        return null;
    }
```

原来如此，如果不设置ThreadLocal的数值，默认就是null，来自于此。

Ok，整体上关于的ThreadLocal内容就这么多了，还有一些细节没有讲述到，慢慢补充和优化。