

ThreadLocal 续篇 ----- 父子线程如何共享数据

作者: [614756773](#)

原文链接: <https://ld246.com/article/1592292506593>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)



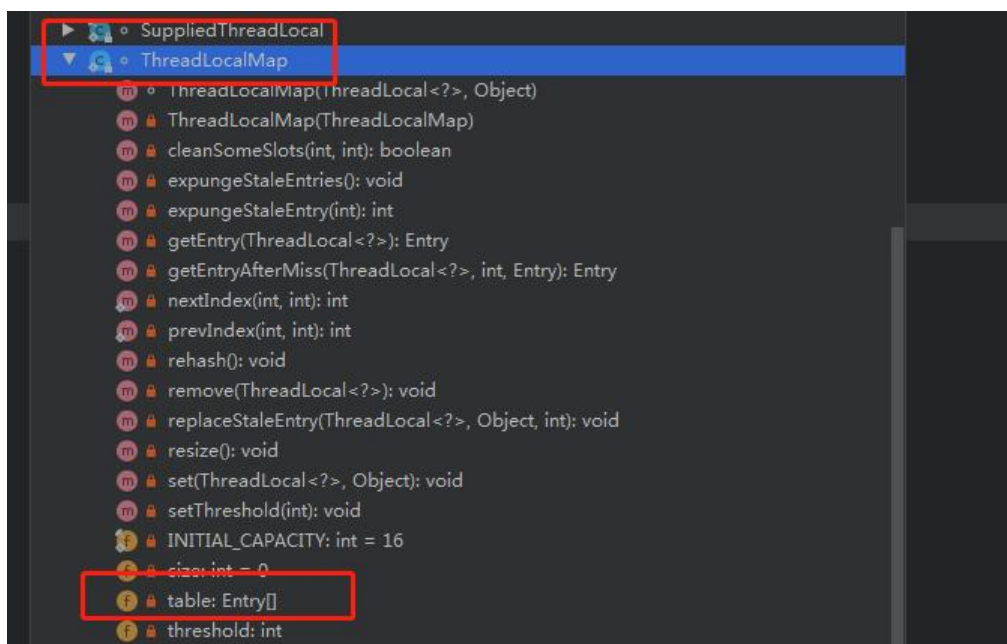
ThreadLocal 和 InheritableThreadLocal

父子线程共享数据可以通过使用 `InheritableThreadLocal` 来达成目的，`InheritableThreadLocal` 类承于 `ThreadLocal`

ThreadLocal

主要成员变量

`ThreadLocal` 最主要的成员变量是 `ThreadLocalMap`，该对象用于存储我们需要保存的数据，`ThreadLocalMap` 使用 `Entry` 数组存储数据



Entry 继承自 WeakReference，使用 ThreadLocal 实例作为键，因此如果 Entry 的键没有被强引用，那么当下次 GC 时就会被回收掉，该 Entry 实例的键就为 null 了，但是 value 依然存在。

```
/* as state entries in the code that follows.
 */
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

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

- 强引用 StrongReference

如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

- 软引用 SoftReference

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用于实现内敏感的高速缓存。

- 弱引用 WeakReference

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间是否足够，都会回收的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用对象。

- 虚引用 PhantomReference

无法通过虚引用获取与之关联的对象实例，且当对象仅被虚引用引用时，在任何发生 GC 的时候，其可被回收。

主要方法

ThreadLocal 主要的数据结构就是 ThreadLocalMap 以及 Entry，接着我们看一下最主要的方法

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

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();
}

ThreadLocalMap getMap(Thread t) {
    // ThreadLocalMap在当前线程被所有ThreadLocal共享
    return t.threadLocals;
}

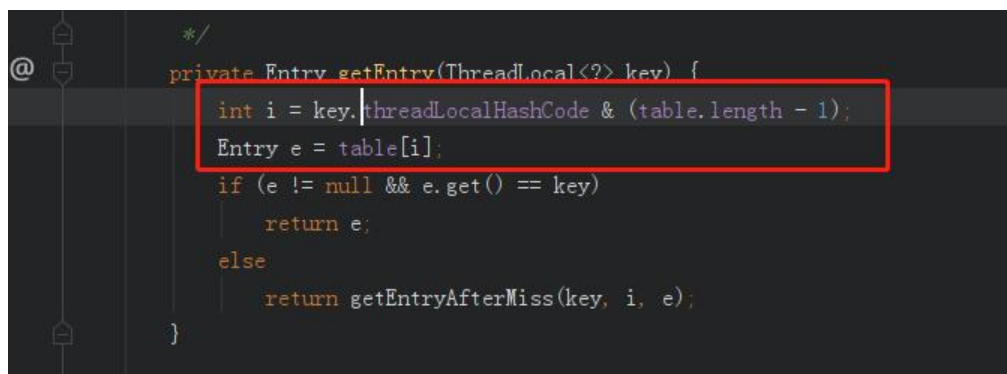
void createMap(Thread t, T firstValue) {
    // 初始化map,构建table与Entry
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}

```

ThreadLocal 在进行 set 时，把自己作为键。在 get 时首先获取当前 **Thread**，然后通过它获取到 **ThreadLocalMap** 实例，最后通过 **ThreadLocalMap** 实例获取到 value

threadLocalHashCode 和线性探测

因为 **ThreadLocalMap** 使用了 **Entry** 数组，所以我们可以知道这块设计类似于 **HashMap** 的 hash，在存取数据时肯定要先确定 hash 桶的位置，这就要使用 hashCode 来定位了



```

    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);
    }

```

getEntry 是使用的 threadLocalHashCode 定位，我们来看一看这个变量


```

85     private final int threadLocalHashCode = nextHashCode();
86
87     /**
88      * The next hash code to be given out. Updated atomically. Starts at
89      * zero.
90      */
91     private static AtomicInteger nextHashCode =
92         new AtomicInteger();
93
94     /**
95      * The difference between successively generated hash codes - turns
96      * implicit sequential thread-local IDs into near-optimally spread
97      * multiplicative hash values for power-of-two-sized tables.
98      */
99     private static final int HASH_INCREMENT = 0x61c88647;
100
101     /**
102      * Returns the next hash code.
103      */
104     private static int nextHashCode() {
105         return nextHashCode.getAndAdd(HASH_INCREMENT);
106     }

```

`threadLocalHashCode` 的生成还依赖 `HASH_INCREMENT`，这是一个写死的数，十进制为 164053127，每个 `ThreadLocal` 实例的 `hashCode` 都会根据这个值递增，并且这个递增是全局的，在整个 jvm 中不会存在重复的两个 `threadLocalHashCode` 值

`HASH_INCREMENT` 是一个很特殊的值，它能够让哈希码均匀的分布在 2 的 N 次方的数组里，也就让 `threadLocalHashCode & (table.length - 1)` 的值非常均匀，也正是因为这个原因所以 `ThreadLocalMap` 没有采用 `HashMap` 的拉链法来解决 hash 冲突，而是使用线性探测

自动清除

之前在分析数据结构时发现 `Entry` 的 `key` 是一个弱引用，所以就可能会存在 `key` 被 GC 掉了但是 `value` 还存在。那么肯定有一个机制用于清除这种 `Entry` 实例，具体的方法就是 `expungeStaleEntry`

```

private int expungeStaleEntry(int staleSlot) {
    Entry[] tab = table;
    int len = tab.length;

    // 1.直接清除指定位置的Entry对象
    tab[staleSlot].value = null;
    tab[staleSlot] = null;
    size--;

    // 2.遍历Entry数组直到某个Entry对象的value不为null，然后把这期间遇到的所有key为null
    都删除掉
    Entry e;

```

```

int i;
for (i = nextIndex(staleSlot, len); (e = tab[i]) != null; i = nextIndex(i, len)) {
    ThreadLocal<?> k = e.get();
    if (k == null) {
        e.value = null;
        tab[i] = null;
        size--;
    } else {
        int h = k.threadLocalHashCode & (len - 1);
        if (h != i) {
            tab[i] = null;

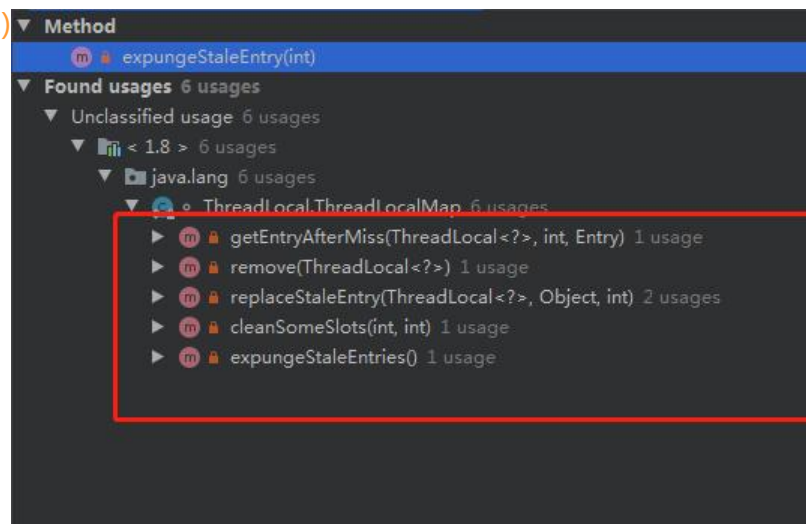
            // Unlike Knuth 6.4 Algorithm R, we must scan until
            // null because multiple entries could have been stale.
            while (tab[h] != null)
                h = nextIndex(h, len);
            tab[h] = e;
        }
    }
}
return i;
}

```

注释中写的很清楚，主要有两个删除动作：

- 1.直接清除指定位置的 Entry 对象
- 2.遍历 Entry 数组直到某个 Entry 对象的 value 不为 null，然后把这期间遇到的所有 key 为 null 的删除掉

主要是以下几个方法在调用 `expungeStaleEntry(...)`



随便举一个调用链例子：ThreadLocal 的 get 方法 -> ThreadLocalMap 的 getEntry 方法 -> getEntryAfterMiss 方法 -> expungeStaleEntry 方法

所以不难得知在调用 ThreadLocal 进行 get 或者 remove 时都会把 key 为 null 的 Entry 对象给删掉

InheritableThreadLocal

方法

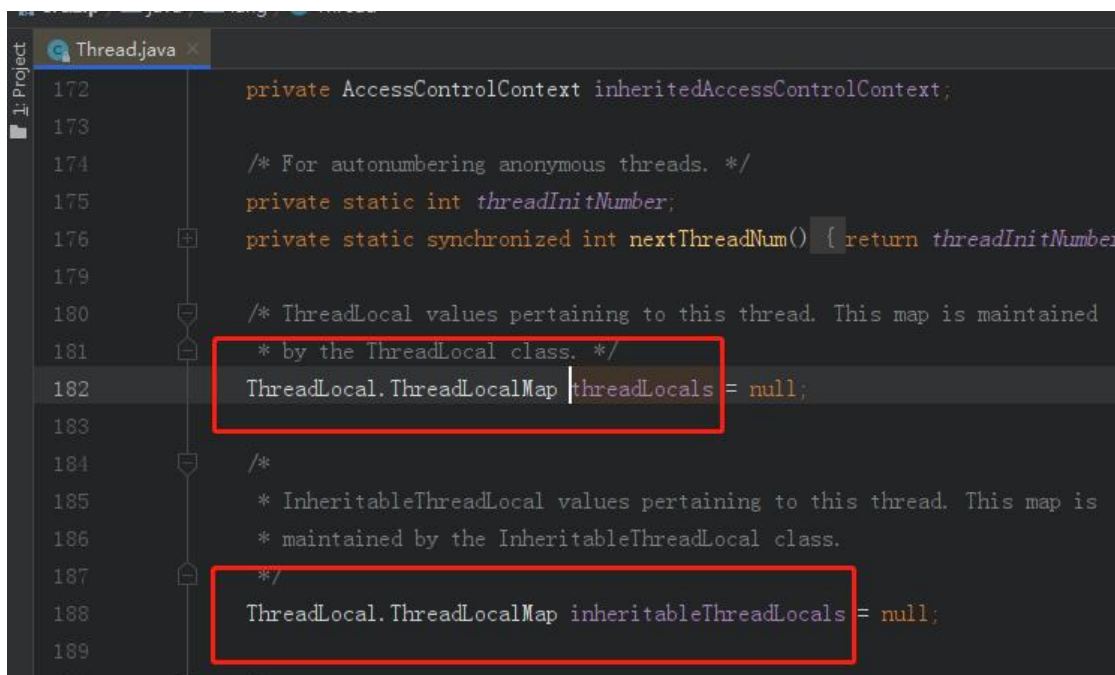
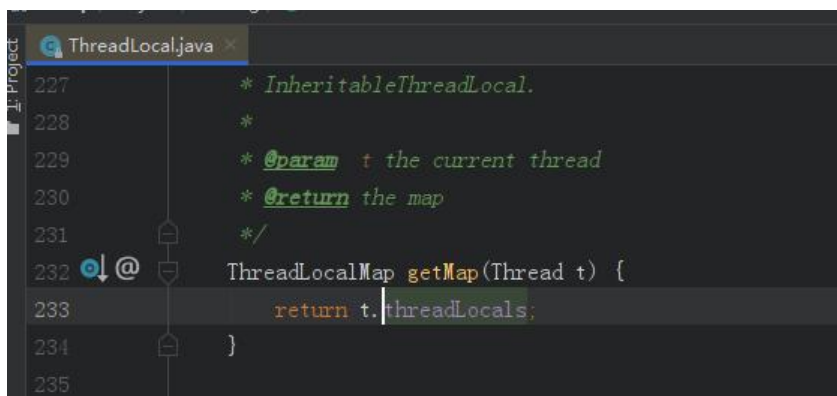
```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {
    public InheritableThreadLocal() {
    }

    protected T childValue(T var1) {
        return var1;
    }

    ThreadLocalMap getMap(Thread var1) {
        return var1.inheritableThreadLocals;
    }

    void createMap(Thread var1, T var2) {
        var1.inheritableThreadLocals = new ThreadLocalMap(this, var2);
    }
}
```

该类很简单，就重写了 `ThreadLocal` 的几个方法，在原来的 `ThreadLocal` 中 `getMap` 是调用 `Thread` 实例获取 `threadLocals` 对象，现在则是获取 `Thread` 的 `inheritableThreadLocals` 对象



同理在创建时，`ThreadLocal` 是将实例化的 `ThreadLocalMap` 赋给 `threadLocals` 变量，`InheritableThreadLocal` 则是将实例化的 `ThreadLocalMap` 赋给 `inheritableThreadLocals` 变量

父子线程共享数据

如果父线程存在 `inheritableThreadLocals` 变量（也就是 `Thread` 的实例中），那么在创建子线程时会把父线程的 `inheritableThreadLocals` 引用过来，然后就可以通过这个变量进行共享数据了，调用如下

`Thread` 类：

```
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize) {
    init(g, target, name, stackSize, null, true);
}

private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc,
                  boolean inheritThreadLocals) {
    ....
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
            ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    ....
}
```

`ThreadLocal` 类：

```
static ThreadLocalMap createInheritedMap(ThreadLocalMap parentMap) {
    return new ThreadLocalMap(parentMap);
}
```

最后关键的地方在于 `ThreadLocalMap` 的初始化，如下

```

private ThreadLocalMap(ThreadLocalMap parentMap) {
    Entry[] parentTable = parentMap.table;
    int len = parentTable.length;
    setThreshold(len);
    table = new Entry[len];

    for (int j = 0; j < len; j++) {
        Entry e = parentTable[j];
        if (e != null) {
            /unchecked/
            ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();
            if (key != null) {
                Object value = key.childValue(e.value);
                Entry c = new Entry(key, value);
                int h = key.threadLocalHashCode & (len - 1);
                while (table[h] != null)
                    h = nextIndex(h, len);
                table[h] = c;
                size++;
            }
        }
    }
}

```

代码很简单直观，就是通过父线程的 `ThreadLocalMap` 实例拿到 `Entry` 数组，然后浅拷贝一份，这样子线程就拥有了父线程通过 `InheritableThreadLocals` 设置的数据

这个 `private ThreadLocalMap(ThreadLocalMap parentMap)` 方法只有在父线程使用了 `InheritableThreadLocals` 才会用到