ThreadLocal是否会引起内存溢出？

博客分类：

- java

javaThreadLocal

最近碰到一个使用ThreadLocal时因为未调用remove()而险些引起内存溢出的问题，所以看了下ThreadLocal的源码，结合线程池原理做一个简单的分析，确认是否最终会导致内存溢出。

既然是因为没调用remove()方法而险些导致内存溢出，那首先看下remove()方法中做了什么。

Java代码

```java
public void remove() {
         ThreadLocalMap m = getMap(Thread.currentThread());
         if (m != null)
             m.remove(this);
     }
```

从remove()的实现来看就是一个map.remove()的调用。既然不调用map.remove()可能会引起内存溢出的话，就需要看看ThreadLocalMap的实现了。

Java代码

```java
/**
 * ThreadLocalMap is a customized hash map suitable only for
 * maintaining thread local values. No operations are exported
 * outside of the ThreadLocal class. The class is package private to
 * allow declaration of fields in class Thread.  To help deal with
 * very large and long-lived usages, the hash table entries use
 * WeakReferences for keys. However, since reference queues are not
 * used, stale entries are guaranteed to be removed only when
 * the table starts running out of space.
 */
```

```java
static class ThreadLocalMap {

    /**
     * The entries in this hash map extend WeakReference, using
     * its main ref field as the key (which is always a
     * ThreadLocal object).  Note that null keys (i.e. entry.get()
     * == null) mean that the key is no longer referenced, so the
     * entry can be expunged from table.  Such entries are referred to
     * as "stale 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;
        }
    }

    /**
     * The initial capacity -- MUST be a power of two.
     */
    private static final int INITIAL_CAPACITY = 16;

    /**
     * The table, resized as necessary.
     * table.length MUST always be a power of two.
     */
    private Entry[] table;

    /**
     * The number of entries in the table.
     */
    private int size = 0;

    /**
     * The next size value at which to resize.
     */
    private int threshold; // Default to 0

    /**
     * Set the resize threshold to maintain at worst a 2/3 load factor.
     */
    private void setThreshold(int len) {
        threshold = len * 2 / 3;
    }
```

```java

    /**
     * Increment i modulo len.
     */
    private static int nextIndex(int i, int len) {
        return ((i + 1 < len) ? i + 1 : 0);
    }

    /**
     * Decrement i modulo len.
     */
    private static int prevIndex(int i, int len) {
        return ((i - 1 >= 0) ? i - 1 : len - 1);
    }

    /**
     * Construct a new map initially containing (firstKey, firstValue).
     * ThreadLocalMaps are constructed lazily, so we only create
     * one when we have at least one entry to put in it.
     */
    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);
    }

    /**
     * Construct a new map including all Inheritable ThreadLocals
     * from given parent map. Called only by createInheritedMap.
     *
     * @param parentMap the map associated with parent thread.
     */
    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) {
                ThreadLocal key = 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++;
                }
            }
        }
    }

    /**
     * Get the entry associated with key.  This method
     * itself handles only the fast path: a direct hit of existing
     * key. It otherwise relays to getEntryAfterMiss.  This is
     * designed to maximize performance for direct hits, in part
     * by making this method readily inlinable.
     *
     * @param  key the thread local object
     * @return the entry associated with key, or null if no such
     */
    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);
    }

    /**
     * Version of getEntry method for use when key is not found in
     * its direct hash slot.
     *
     * @param  key the thread local object
     * @param  i the table index for key's hash code
     * @param  e the entry at table[i]
     * @return the entry associated with key, or null if no such
     */
    private Entry getEntryAfterMiss(ThreadLocal key, int i, Entry e) {
        Entry[] tab = table;
        int len = tab.length;

        while (e != null) {
            ThreadLocal k = e.get();
            if (k == key)
                return e;
            if (k == null)
```

```
                expungeStaleEntry(i);
            else
                i = nextIndex(i, len);
            e = tab[i];
        }
        return null;
    }

    /**
     * Set the value associated with key.
     *
     * @param key the thread local object
     * @param value the value to be set
     */
    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();
    }

```

```
    /**
     * Remove the entry for key.
     */
    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;
            }
        }
    }

    /**
     * Replace a stale entry encountered during a set operation
     * with an entry for the specified key.  The value passed in
     * the value parameter is stored in the entry, whether or not
     * an entry already exists for the specified key.
     *
     * As a side effect, this method expunges all stale entries in the
     * "run" containing the stale entry.  (A run is a sequence of entries
     * between two null slots.)
     *
     * @param  key the key
     * @param  value the value to be associated with key
     * @param  staleSlot index of the first stale entry encountered while
     *         searching for key.
     */
    private void replaceStaleEntry(ThreadLocal key, Object value,
                                   int staleSlot) {
        Entry[] tab = table;
        int len = tab.length;
        Entry e;

        // Back up to check for prior stale entry in current run.
        // We clean out whole runs at a time to avoid continual
        // incremental rehashing due to garbage collector freeing
        // up refs in bunches (i.e., whenever the collector runs).
        int slotToExpunge = staleSlot;
        for (int i = prevIndex(staleSlot, len);
(e = tab[i]) != null;
             i = prevIndex(i, len))
```

```
                if (e.get() == null)
                    slotToExpunge = i;

            // Find either the key or trailing null slot of run, whichever
            // occurs first
            for (int i = nextIndex(staleSlot, len);
(e = tab[i]) != null;
                 i = nextIndex(i, len)) {
                ThreadLocal k = e.get();

                // If we find key, then we need to swap it
                // with the stale entry to maintain hash table order.
                // The newly stale slot, or any other stale slot
                // encountered above it, can then be sent to expungeStaleEntry
                // to remove or rehash all of the other entries in run.
                if (k == key) {
                    e.value = value;

                    tab[i] = tab[staleSlot];
                    tab[staleSlot] = e;

                    // Start expunge at preceding stale entry if it exists
                    if (slotToExpunge == staleSlot)
                        slotToExpunge = i;
                    cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
                    return;
                }

                // If we didn't find stale entry on backward scan, the
                // first stale entry seen while scanning for key is the
                // first still present in the run.
                if (k == null && slotToExpunge == staleSlot)
                    slotToExpunge = i;
            }

            // If key not found, put new entry in stale slot
            tab[staleSlot].value = null;
            tab[staleSlot] = new Entry(key, value);

            // If there are any other stale entries in run, expunge them
            if (slotToExpunge != staleSlot)
                cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
        }

        /**
         * Expunge a stale entry by rehashing any possibly colliding entries
         * lying between staleSlot and the next null slot.  This also expunges
```

```
     * any other stale entries encountered before the trailing null.  See
     * Knuth, Section 6.4
     *
     * @param staleSlot index of slot known to have null key
     * @return the index of the next null slot after staleSlot
     * (all between staleSlot and this slot will have been checked
     * for expunging).
     */
    private int expungeStaleEntry(int staleSlot) {
        Entry[] tab = table;
        int len = tab.length;

        // expunge entry at staleSlot
        tab[staleSlot].value = null;
        tab[staleSlot] = null;
        size--;

        // Rehash until we encounter null
        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;
    }

    /**
     * Heuristically scan some cells looking for stale entries.
     * This is invoked when either a new element is added, or
```

```
     * another stale one has been expunged. It performs a
     * logarithmic number of scans, as a balance between no
     * scanning (fast but retains garbage) and a number of scans
     * proportional to number of elements, that would find all
     * garbage but would cause some insertions to take O(n) time.
     *
     * @param i a position known NOT to hold a stale entry. The
     * scan starts at the element after i.
     *
     * @param n scan control: <tt>log2(n)</tt> cells are scanned,
     * unless a stale entry is found, in which case
     * <tt>log2(table.length)-1</tt> additional cells are scanned.
     * When called from insertions, this parameter is the number
     * of elements, but when from replaceStaleEntry, it is the
     * table length. (Note: all this could be changed to be either
     * more or less aggressive by weighting n instead of just
     * using straight log n. But this version is simple, fast, and
     * seems to work well.)
     *
     * @return true if any stale entries have been removed.
     */
    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;
    }

    /**
     * Re-pack and/or re-size the table. First scan the entire
     * table removing stale entries. If this doesn't sufficiently
     * shrink the size of the table, double the table size.
     */
    private void rehash() {
        expungeStaleEntries();

        // Use lower threshold for doubling to avoid hysteresis
        if (size >= threshold - threshold / 4)
```

```java
            resize();
    }

    /**
     * Double the capacity of the table.
     */
    private void resize() {
        Entry[] oldTab = table;
        int oldLen = oldTab.length;
        int newLen = oldLen * 2;
        Entry[] newTab = new Entry[newLen];
        int count = 0;

        for (int j = 0; j < oldLen; ++j) {
            Entry e = oldTab[j];
            if (e != null) {
                ThreadLocal k = e.get();
                if (k == null) {
                    e.value = null; // Help the GC
                } else {
                    int h = k.threadLocalHashCode & (newLen - 1);
                    while (newTab[h] != null)
                        h = nextIndex(h, newLen);
                    newTab[h] = e;
                    count++;
                }
            }
        }

        setThreshold(newLen);
        size = count;
        table = newTab;
    }

    /**
     * Expunge all stale entries in the table.
     */
    private void expungeStaleEntries() {
        Entry[] tab = table;
        int len = tab.length;
        for (int j = 0; j < len; j++) {
            Entry e = tab[j];
            if (e != null && e.get() == null)
                expungeStaleEntry(j);
        }
    }
}
```

首先从声明上来看，ThreadLocalMap并不是一个java.util.Map接口的实现，但是从Entry的实现和整个ThreadLocalMap的实现来看却实现了一个Map的功能，并且从具体的方法的实现上来看，整个ThreadLocalMap实现了一个HashMap的功能，对比HashMap的实现就能看出。

但是，值得注意的是ThreadLocalMap并没有put(K key, V value)方法，而是set(ThreadLocal key, Object value)，从这里可以看出，ThreadLocalMap并不是想象那样以Thread为key，而是以ThreadLocal为key。

了解了ThreadLocalMap的实现，也知道ThreadLocal.remove()其实就是ThreadLocalMap.remove()，那么再看看ThreadLocal的set(T value)方法，看看value是如何存储的。

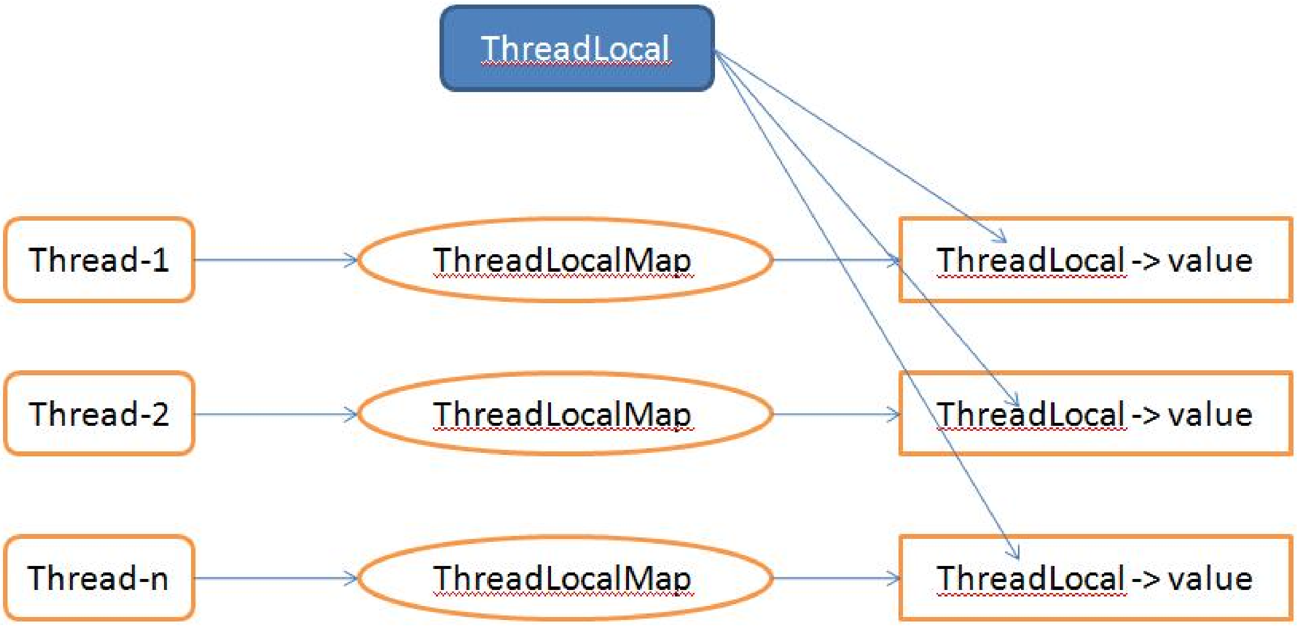
Java代码

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

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

    void createMap(Thread t, T firstValue) {
        t.threadLocals = new ThreadLocalMap(this, firstValue);
    }
```

可以看到，set(T value)方法为每个Thread对象都创建了一个ThreadLocalMap，并且将value放入ThreadLocalMap中，ThreadLocalMap作为Thread对象的成员变量保存。那么可以用下图来表示ThreadLocal在存储value时的关系。



所以当ThreadLocal作为单例时，每个Thread对应的ThreadLocalMap中只会有一个键值对。那么如果不调用remove()会怎么样呢?

假设一种场景，使用线程池，线程池中有200个线程，并且这些线程都不会释放，ThreadLocal做单例使用。那么最多也就会产生200个ThreadLocalMap，而每个ThreadLocalMap中只有一个键值对，那最多也就是200个键值对存在。

但是线程池并不是固定一个线程数不改变，下面贴一段tomcat的线程池配置
Java代码

```
<Connector executor="tomcatThreadPool"
           port="8080" protocol="HTTP/1.1"
           connectionTimeout="60000"
           keepAliveTimeout="30000"
           minProcessors="5"
           maxProcessors="75"
           maxKeepAliveRequests="150"
           redirectPort="8443" URIEncoding="UTF-8" acceptCount="1000" disableUploadTimeout="true"/>
```

可以看到线程池其实有线程最小值和最大值的，并且有超时时间，所以当线程空闲时间超时后，线程会被销毁。那么当线程销毁时，线程所持有的ThreadLocalMap也会失去引用，并且由于ThreadLocalMap中的Entry是WeakReference，所以当YGC时，被销毁的Thread所对应的value也会被回收掉，所以即使不调用remove()方法，也不会引起内存溢出。

- 查看图片附件

分享到：
lombok生成getter、setter的小陷阱

- 2013-01-23 11:56
- 浏览 5503
- [评论(1)]
- 分类:[编程语言]
- [查看更多]

相关资源推荐

- Java数据结构与算法解析(一)——表
- 关注CSDN程序人生公众号，轻松获得下载...
- 操作系统OEM DIY工具
- 一小时学会搭建网站
- 黑客基础知识大全（TXT）
- java编程语言
- 自制编程语言
- [Ruby编程语言].弗拉纳根_松本行弘.扫描版....
- 自制编程语言
- Apple_Swift编程语言入门教程
- Python Crash Course
- 微信小程序 VS 原生App
- 教你怎么免费搭建discuz论坛教程
- Wi-Fi 爆重大安全漏洞，Android、iOS、Wi...
- 编程语言的内存对齐
- C#.NET编程语言详解
- Apple Swift编程语言入门教程
- 编程语言与平台发展
- GBT 15969.3-2005 可编程序控制器 第3部...
- 西门子TDC编程语言CFC功能块详细说明中...

参考知识库

Java SE知识库 *29198* 关注 | *578* 收录

Java 知识库 *36618* 关注 | *3748* 收录

Java EE知识库 *23585* 关注 | *1416* 收录

JavaScript知识库 *17404* 关注 | *1517* 收录

jQuery知识库 *10667* 关注 | *948* 收录

AngularJS知识库 *5537* 关注 | *590* 收录

评论

1 楼 zhuyucheng123 2014-06-11
对问题的分析很精彩，但是我想问问，像后面配置文件这样解释的话，那么内存是否泄露是不是要依赖与第三方线程池呢？如果第三方线程池刚好没有所谓的“超时时间”的话，就会发生内存泄露了，对吗？

发表评论

您还没有登录,请您登录后再发表评论

liwx2000

- 浏览: 81991 次
- 性别: ♂
- 来自: 杭州
- 我现在离线

最近访客 更多访客>>

Zs123456789

xiaoqiS

zingers

a714115852

文章分类

- 全部博客 (4)
- java (1)
- 测试 (0)

社区版块

- 我的资讯 (0)
- 我的论坛 (0)

- 我的问答 (1)

存档分类

- 2013-01 (1)
- 2012-05 (3)
- 更多存档...

最新评论

- kailee： 博主分析了大半天的没用的。。。吓得我以为啥陷阱一般boolea ...
  lombok生成getter、setter的小陷阱
- maoweiwer： 这代码编译能过？ new ServletInputStream(...
  ServletRequest中getReader()和getInputStream()只能调用一次的解决办法
- xczzmn： 将字段类型boolean换成Boolean就可以了
  lombok生成getter、setter的小陷阱
- xugangwen： 和lombok没关系
  lombok生成getter、setter的小陷阱
- jacktao219： 赞一赞~~
  ServletRequest中getReader()和getInputStream()只能调用一次的解决办法
