参考：

https://mp.weixin.qq.com/s/-IKMuPNE6M0e5B0QsUkIyQ

# 前言

ThreadLocal这个概念比较冷门，但也算是并发家族中一个重要的成员。为啥用的不多呢，它的作用是在并发中将成员变量变为线程局部变量（相当于每个线程独享一个版本），实际上，很少会有机会能遇到这样的场景。

# Demo

当然我们前提先明白我们什么样的场景下要用ThreadLocal，而不是无脑的关于线程安全去了，ThreadLocal正如前言所说是想在一个对象的情况下并发保证每个线程的变量值都是自己独享的。

OK，先看看我们并发正常情况下是什么样的情况。看代码。

```
public class TestThreadLocal {

    private Integer i = 0;

    // private static ThreadLocal<Integer> local = new ThreadLocal<Integer>();
    //
    // public void setLocal() {
    // local.set(0);
    // }

    public void count() {
        i++;

        // Integer i = local.get();
        // i++;
        // System.out.println(i);
    }

    public static void main(String[] args) {

        TestThreadLocal local = new TestThreadLocal();
```

```
        for (int i = 0; i < 100; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    // local.setLocal();
                    local.count();
                }
            }).start();
        }

        // 停顿半秒保证所有线程都结束再输出结果
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("结果：" + local.i);
    }

}
```

结果不确定（由于线程安全性问题，这里不讨论这个，有关线程安全性问题找相关笔记进行查阅）。

```
结果：98
```

很显然，并发100个线程操作一个对象，对象里面的i值肯定是不断累加的，由于线程安全性问题不会每次都100，可能99，98，97等等。显然达不到我们要100个线程的i值最后都是1这个要求，要简单，可以，每操作一个线程new一个新的对象，但是性能消耗随着线程数量的扩大那是量级别的扩大。这时候我们引入ThreadLocal的使用（其实代码里已经完成了，我们把现有关于i的操作注释，并放开原本注释的代码），看结果。

```
<terminated> TestThreadLocal [Java Application] E:\Program Files
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
```

这结果就符合我们这边的需求了。

## ThreadLocal原理分析

我们先来自己分析下这个需求，并发情况下每个线程具有一个原本应该共享的变量的独享值，那除了每个线程新new好像没有法子了啊。

这就狭隘了，我们先不要想ThreadLocal，我们自己来分析下现在有啥要啥，我们有100个线程，有一个变量i，我们只有一个对象，但我们想要100个线程都有自己的i值。

那我们为啥不把i值的管理权交给第三方呢，线程是有自己的标识的，相当于100个可以根据身份证唯一标识的客户，那我们第三方就根据每个线程存一份i值，要用时由第三方找到对应线程的i值给他就是了。

ThreadLocal就是利用了这种原理，便扮演了这样的一个第三方。基于储值方式通俗的说，它内部通过一个基于WeakReference的键值对存储的方式将对应线程的该变量值进行设值和获取操作，保证每个线程有自己的值，这样实现了线程局部变量（每个线程独享自己的变量值）。

## 基于源码分析ThreadLocal

首先我们看这个。

```
private static ThreadLocal<Integer> local = new ThreadLocal<Integer>();
```

这一步就干了俩事，实例化一个ThreadLocal对象并由于它是泛型类所以指定泛型实例类（这个类和原本变量的类型对应即可）。

然后我们看关键两步骤的第一步，设值，

```
public void setLocal() {
    local.set(0);
}
```

这一步相当于比原始的用法多的，并且不能在构造方法中用，我们实例化对象时不涉及到我们100个线程，那样设值只会设值一个并且时我们的启动主线程，我们这边肯定不能放在count方法中直接用，不然每次都是从0开始，所以单独写了一个方法（具体情况具体使用）。

直接进去看源码。

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

可以看到果然是根据当前线程去操作的，值存在ThreadLocalMap这个看起来像是个map的类里（这个不是传统的map，是基于WeakReference去操作的，后面讲）。这边会先根据当前线程取值，若有直接覆盖，若没有会调用createMap方法创建，我们先看创建的方法。

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

简单粗暴，直接调用ThreadLocalMap的构造方法将实例赋给线程的threadLocals（先提一下吧，ThreadLocalMap是ThreadLocal的内部类，然后是静态的，但是主要存放在Thread中跟随线程走动，这个用法的原因等后面再解读）。

```
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);
}
```

构造方法我们来研究下，重点在画红线处，通过Entry这个类貌似创建了键值对的存储方式。

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

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

这个话题我们暂时到这，至少明白了，最后存值的是Entry。

我们看看怎么取值。

```
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();
}
```

首先获取相关线程的ThreadLocalMap，有就继续处理，无则设值初始化值，我们先看看这个setInitialValue。

```
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;
}
```

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

emmm，绕了半天最后就是返回个null。（我想了好几种说法来圆这边的绕弯，然而都说不太通，先放着吧）（解答见下文）

再看看有怎么处理的，获取ThreadLocalMap的Entry，所以说，这边的Entry为啥要放在一个数组里我又不懂了。（因为该线程又不是只处理A类，B、C甚至更多，都有ThreadLocal，都归属这个线程管理）

```
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);
}
```

无奈，虽说有几处看不懂的，总体来说，这就是ThreadLocal的set，get的方式，通过绑定线程的ThreadLocal去获取唯一值。

# 针对ThreadLocal产生OOM（Out Of Memory）的原因分析

首先来个例子模拟下出现这个问题。

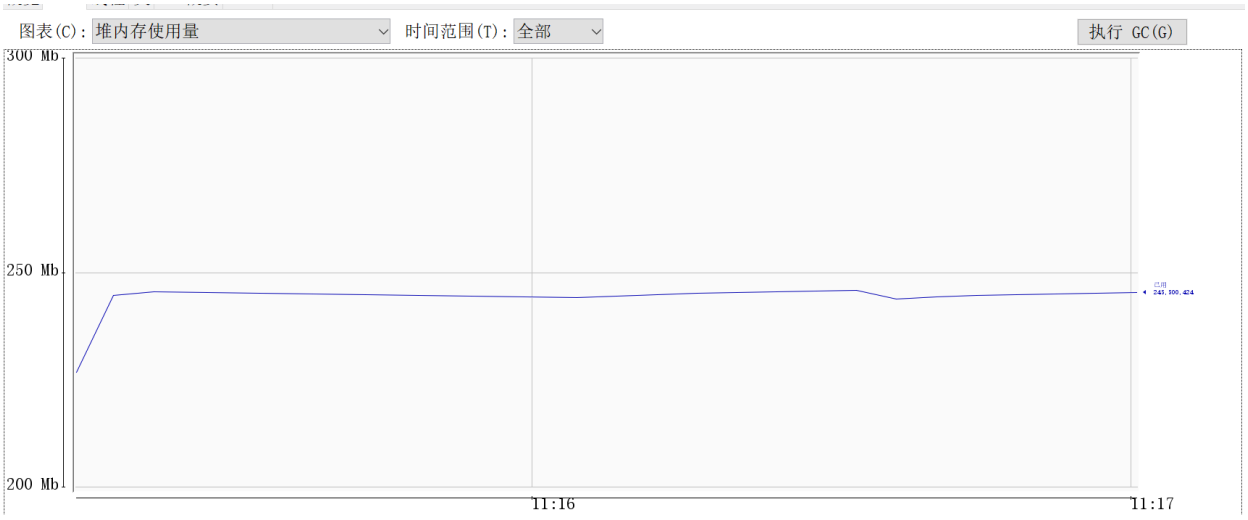
```
public class ThreadLocalOOM {

    private static ThreadLocal<List<User>> threadLocal = new ThreadLocal<>();

    private List<User> addUsers() {
        List<User> users = new ArrayList<>(100000);
        for (int i = 0; i < 100000; i++) {
            users.add(new User(1L, "admin", "123456", 12));
        }
        return users;
    }

    public static void main(String[] args) {
        Executor executor = Executors.newFixedThreadPool(500);
```

```
            for (int i = 0; i < 500; i++) {
                executor.execute(new Runnable() {
                    @Override
                    public void run() {
                        threadLocal.set(new ThreadLocalOOM().addUsers());
                        System.out.println(new Thread().currentThread().getName());
                    }
                });
                try {
                    Thread.sleep(1000l);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
}

class User {
        private Long id;
        private String username;
        private String password;
        private Integer age;

        public User(Long id, String username, String password, Integer age) {
            super();
            this.id = id;
            this.username = username;
            this.password = password;
            this.age = age;
        }

}
```

然后执行时设值启动参数-Xmx256m，限定JVM最高256m内存，然后我们执行，最终就是报错了。

```
kernalBoot - ThreadLocalOOM [Spring Boot App] E:\Program Files\Java\jdk1.8.0_151\bin\javaw.exe (2019年1月23日 上午11:14:12)
pool-1-thread-62
pool-1-thread-63
pool-1-thread-64
pool-1-thread-65
pool-1-thread-66
pool-1-thread-67
Exception in thread "pool-1-thread-70" Exception in thread "main" Exception in thread "pool-1-thread-72" java.lang.OutOfMemoryError: Java heap space
	at java.util.ArrayList.<init>(ArrayList.java:153)
	at com.xyz.thread.threadlocal.ThreadLocalOOM.addUsers(ThreadLocalOOM.java:13)
	at com.xyz.thread.threadlocal.ThreadLocalOOM.access$1(ThreadLocalOOM.java:12)
	at com.xyz.thread.threadlocal.ThreadLocalOOM$1.run(ThreadLocalOOM.java:27)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
	at java.lang.Thread.run(Thread.java:748)
Exception in thread "pool-1-thread-71" java.lang.OutOfMemoryError: Java heap space
	at java.util.ArrayList.<init>(ArrayList.java:153)
	at com.xyz.thread.threadlocal.ThreadLocalOOM.addUsers(ThreadLocalOOM.java:13)
	at com.xyz.thread.threadlocal.ThreadLocalOOM.access$1(ThreadLocalOOM.java:12)
	at com.xyz.thread.threadlocal.ThreadLocalOOM$1.run(ThreadLocalOOM.java:27)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
	at java.lang.Thread.run(Thread.java:748)
java.lang.OutOfMemoryError: Java heap space
	at java.util.ArrayList.<init>(ArrayList.java:153)
	at com.xyz.thread.threadlocal.ThreadLocalOOM.addUsers(ThreadLocalOOM.java:13)
	at com.xyz.thread.threadlocal.ThreadLocalOOM.access$1(ThreadLocalOOM.java:12)
	at com.xyz.thread.threadlocal.ThreadLocalOOM$1.run(ThreadLocalOOM.java:27)
	at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
	at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
	at java.lang.Thread.run(Thread.java:748)
Exception in thread "pool-1-thread-68" Exception in thread "pool-1-thread-77" java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: GC overhead limit exceeded
java.lang.OutOfMemoryError: GC overhead limit exceeded
Exception in thread "pool-1-thread-69" java.lang.OutOfMemoryError: GC overhead limit exceeded
```

我们打开JDK提供的工具jconsole，电脑配置过jdk的直接cmd中输入jconsole即可。




果然早就达到JVM的上限了，那么给我们个OOM报错也很正常。

案例就到这了，关键点在于这是如何产生的，并且我们该怎么解决。

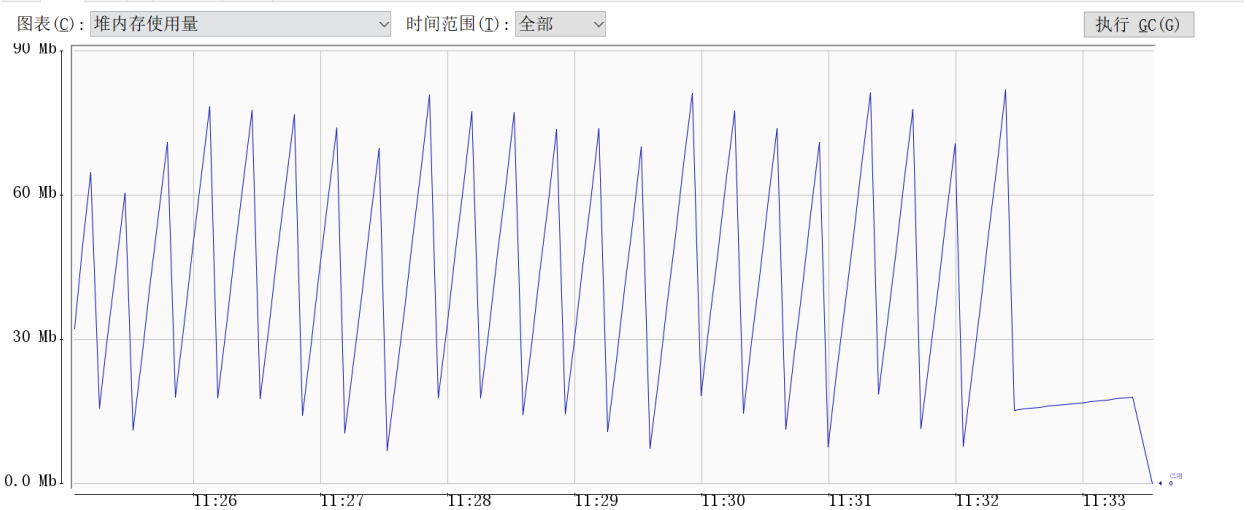
上面的源码分析中我们也明确了，ThreadLocal存值的方式是通过ThreadLocalMap实现，ThreadLocal作为存储方式的key存在，而ThreadLocalMap对象是绑定线程的，在Thread中。因此，在这个过程中，如果ThreadLocal很长时间不再有调用，就会在GC时被回收，然

而，绑定的Thread的ThreadLocalMap还会依旧存在，只不过key变为了null，但是value依旧存在，将持续到当前线程结束为止。而线程一旦累加了，并不及时结束，那显然很容易造成OOM。

那么，为了避免这种问题，我们应该养成良好的习惯，在必要时刻调用ThreadLocal的remove方法，类似这样。

```
threadLocal.set(new ThreadLocalOOM().addUsers());
System.out.println(new Thread().currentThread().getName());
threadLocal.remove();
```

我们再看jconsole。




从图中就可以看粗来，gc在不断的清理数据。当然这边强行用有点牵强人意的意思，具体的案例还得思考具体案例的解决方案。同时，也最好确保数据量的问题，明明只能装10个苹果，硬要塞100个苹果，这从需求上也是存在合理性问题的。