链滴

# 一文看懂 ThreadLocal

作者：BigBigBigPeach

原文链接：https://ld246.com/article/1564929656266

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是什么?

ThreadLocal的JavaDoc上的描述是这样的...

This class provides thread-local variables. These variables differ from their normal counterpats in that each thread that accesses one (via its {@code get} or {@code set} method) has its on, independently initialized copy of the variable. {@code ThreadLocal} instances are typically rivate static fields in classes that wish to associate state with a thread (e.g.,a user ID or Transacion ID).

这个类提供了线程本地变量。这个变量和正常的变量不同。通过get & set 方法，每个线程可以获取自己独立的变量。这个变量实例通常是私有且静态的，可以存储与线程相关的信息，如员工id、事务i等。

# ThreadLocal能解决什么问题?

线程并发问题。因为ThreadLocal解决了变量共享的问题。ThreadLocal还有一个隐含的好处，那就不需要将参数在方法中进行传递，可以直接从线程中获取。

举个栗子：

- SimpleDateFormat是线程不安全的，所以很多项目中在使用SimpleDateFormat对象的时候都是其放入ThreadLocal中。
- Shiro中的Subject就是采用ThreadLocal实现的~SecurityUtils.getSubject();
- Spring中事务信息就是通过ThreadLocal进行传递的

```
@Test
public void testSimpleDateFormat() {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    for (int i = 0; i < 10; ++i) {
```

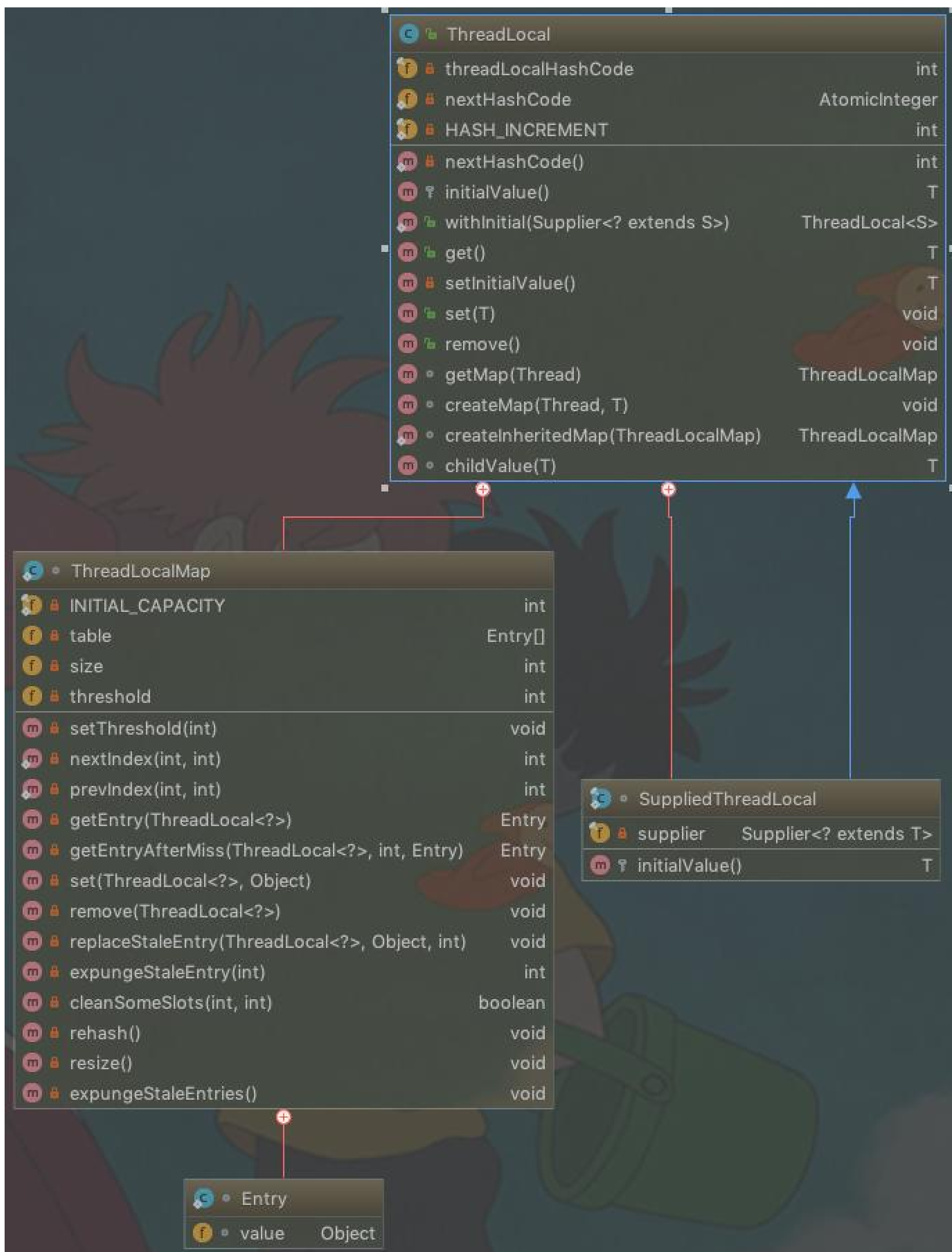
```
        new Thread(() -> {
            try {
                System.out.println(sdf.parse("2017-12-13 15:17:27"));
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

上述代码运行，就会大概率出现java.lang.NumberFormatException（如果运气好没出现，可以尝将线程数量调大一点儿）

SimpleDateFormat之所以是非线程安全的是因为其内部Calendar是线程不安全的。归根结底是因为alendar存放日期数据的变量。

# ThreadLocal是怎么设计的?

## ThreadLocal的类结构图




ThreadLocal类主要是用了一个内部ThreadLocalMap，这个map中存储的键值对是Entry<ThreadLocal,Object>，Entry类中只有value成员变量，key是ThreadLocal对象，Entry继承了WeakReference。

为什么要使用弱引用呢，起初的设计估计是为了让GC自动去清理已经挂掉的线程的相关value。但是有个问题，我们待会儿分析。

ThreadLocal本身并没有持有这个Map对象，而是让Thread对象持有Map对象，大家可以想想这个为什么。

## ThreadLocal & ThreadLocalMap & Thread & Entry的关系

- Thread只能拥有一个ThreadLocalMap对象。
- 一个ThreadLocalMap对象存储多个Entry对象。
- Entry对象的key弱引用指向一个ThreadLocal对象。

- 一个ThreadLocal对象可以被多个线程共享。
- ThreadLocal对象不持有value对象，value由Entry对象持有。

### ThreadLocal的灵魂

- set() 如果不设置值，那么容易引起脏数据问题。（比如上次这个线程被线程池回收，但是没有调用remove，下文我们会提到这个问题）
- get() 如果没有get()，那么我们找不到使用ThreadLocal的意义...
- remove() remove方法一个是保证，接下来使用的时候不会出现脏数据，另外就是保证弱引用的Enty的value能被GC回收，不然会出现内存泄漏...下面有代码演示

# ThreadLocal应该怎么用?

我们一般用ThreadLocal的时候都会在一个类中，声明一个静态对象，让类持有ThreadLocal。

再调用其set()、get()方法，来实现赋值和取值。具体可以看如下代码

```
@Test
public void testThreadLocal() throws InterruptedException {
    for (int i = 0; i < 10; i++) {
        new Thread(Task::new).start();
    }
    for (int i = 0; i < 2; i++) {
        Thread.sleep(2000);
    }
}

static class Task implements Runnable {

    @Override
    public void run() {
        setName(Thread.currentThread().getName());
        try {
            TimeUnit.MILLISECONDS.sleep(500);
        } catch (InterruptedException e) {
            log.error(e.getMessage());
        }
        printName(Thread.currentThread().getName());
     NameThreadLocal.remove();
    }

    /**
     * threadLocal保存一下线程相关名称
     *
     * @param name
     */
    static void setName(String name) {
        NameThreadLocal.setName(name);
    }

    /**
     * 打印一下 取出ThreadLocal的名称，看是否和当前线程一致
```

```
     *
     * @param threadName
     */
    static void printName(String threadName) {
        log.info(threadName + "======" + NameThreadLocal.getName());
    }
}

/**
 * 此类持有ThreadLocal静态对象
 * 此处的Name可以是登录状态也可以是分布式的请求的traceId等等
 */
static class NameThreadLocal {
    private static ThreadLocal<String> threadLocal = new ThreadLocal<>();

    static void setName(String name) {
        threadLocal.set(name);
    }

    static String getName() {
        return threadLocal.get();
    }

    static void remove() {
       threadLocal.remove();
    }
}
// 结果如下
//Thread-3======Thread-3
//Thread-8======Thread-8
//Thread-5======Thread-5
//Thread-11======Thread-11
//Thread-7======Thread-7
//Thread-9======Thread-9
//Thread-12======Thread-12
//Thread-10======Thread-10
//Thread-6======Thread-6
//Thread-4======Thread-4
```

# ThreadLocal使用时有什么坑?

## ThreadLocal的value不能放共享变量

假设有一个共享变量Object。

线程A设置Object的成员变量property为x，放入了ThreadLocal。

线程B设置Object的同一个成员变量property为y，放入了ThreadLocal。

假设两个线程同时执行，我们无法保证线程B再get的时候拿到的property的值是x。

我们对上面的Task类代码进行改造，如下

```
static class Task implements Runnable {
        // 一个共享变量
```

```
    static StringBuilder sb = new StringBuilder("start");

    static AtomicInteger errorNum = new AtomicInteger(0);

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            try {
                TimeUnit.MILLISECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            sb.append(i);
        }
        setName(Thread.currentThread().getName() + sb.toString());
        printName(Thread.currentThread().getName() + sb.toString());
    }

    /**
     * threadLocal保存一下线程相关名称
     *
     * @param name
     */
    static void setName(String name) {
        NameThreadLocal.setName(name);
    }

    /**
     * 打印一下 取出ThreadLocal的名称，看是否和当前线程一致
     *
     * @param threadName
     */
    static void printName(String threadName) {
        if (!threadName.equals(NameThreadLocal.getName())) {
            errorNum.getAndIncrement();
            log.error(errorNum.toString());
        }
    }
  }
```

经过测试，errorNum基本上都是大于0。大家可以自己试一下~

## ThreadLocal遇上线程池

ThreadLocal遇上线程池的问题，就是容易发生内存泄漏。

我们还是对Task类进行改造，再增加一个测试方法。

```
static class Task implements Runnable {
    // 一个共享变量
    static AtomicInteger errorNum = new AtomicInteger(0);

    @Override
    public void run() {
```

```
        // 我们将前两个请求设置名称，后面的不设置
        if (errorNum.getAndIncrement() > 2) {
            setName(Thread.currentThread().getName());
        }
        printName(Thread.currentThread().getName());
    }

    /**
     * threadLocal保存一下线程相关名称
     *
     * @param name
     */
    static void setName(String name) {
        NameThreadLocal.setName(name);
    }

    /**
     * 打印一下 取出ThreadLocal的名称，看是否和当前线程一致
     *
     * @param threadName
     */
    static void printName(String threadName) {
        log.info(threadName + "=======" + NameThreadLocal.getName());
    }
}
// 增加一个测试方法
@Test
public void testMemoryLeak() {
    ExecutorService executorService = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 10; i++) {
        executorService.submit(new Task());
    }
}
// 结果如下
//pool-1-thread-2=======null
//pool-1-thread-3=======null
//pool-1-thread-1=======pool-1-thread-1
//pool-1-thread-2=======pool-1-thread-2
//pool-1-thread-1=======pool-1-thread-1
//pool-1-thread-3=======pool-1-thread-3
//pool-1-thread-2=======pool-1-thread-2
//pool-1-thread-1=======pool-1-thread-1
//pool-1-thread-3=======pool-1-thread-3
```

通过上面的结果我们看到，即使一个线程执行完成了任务，在没有主动清空ThreadLocal的时候，再之前线程池中的线程还会带有之前的ThreadLocal对应的value.因为线程并没有被回收，ThreadLocal键值对并不会被清空，所以也就解答了上文我们提到的问题。这种问题一旦出现，会比较隐蔽。

解决办法就是我们在使用完成之后，需要主动进行释放

`ThreadLocal.remove()`

## Thread创建了子线程，ThreadLocal怎么办?

我们可以看到Thread类中有一个成员变量是用来处理此情况的。

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

我们可以直接使用ThreadLocal的子类**InheritableThreadLocal。**但是继承等操作在线程池中的话可能会因为动态的创建线程而变得非常混乱。所以不是很建议在线程池用InheritableThreadLocal。**
*在看代码的时候，可以看一下此方法的调用。

java.lang.Thread#init(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String, long, java.security.AccessControlContext, boolean)

# 总结

ThreadLocal虽然有一些问题，但是我们不能因噎废食，否认其优秀。希望大家在使用的时候，注意面的坑。