ThreadLocal

1. 两大使用场景——ThreadLocal的用途

- 典型场景1：

  每个线程需要一个**独享**的对象（**通常是工具类**，由于工具类本身不是线程安全的，如果多个线程共享同一个静态工具类，会存在线程安全问题。典型需要使用的类有SimpleDateFormat和Random，这两个类都不是线程安全的。）

- 典型场景2：

  每个线程内需要保存类似于全局变量的信息（例如在拦截器中获取用户信息，该信息在本线程执行的各方法中保持不变），可以让不同方法直接使用，却不想被多线程共享（因为不同线程获取到的用户信息不一样），**避免参数传递的麻烦**

场景一：每个Thread内有自己的实例副本,不可以共享

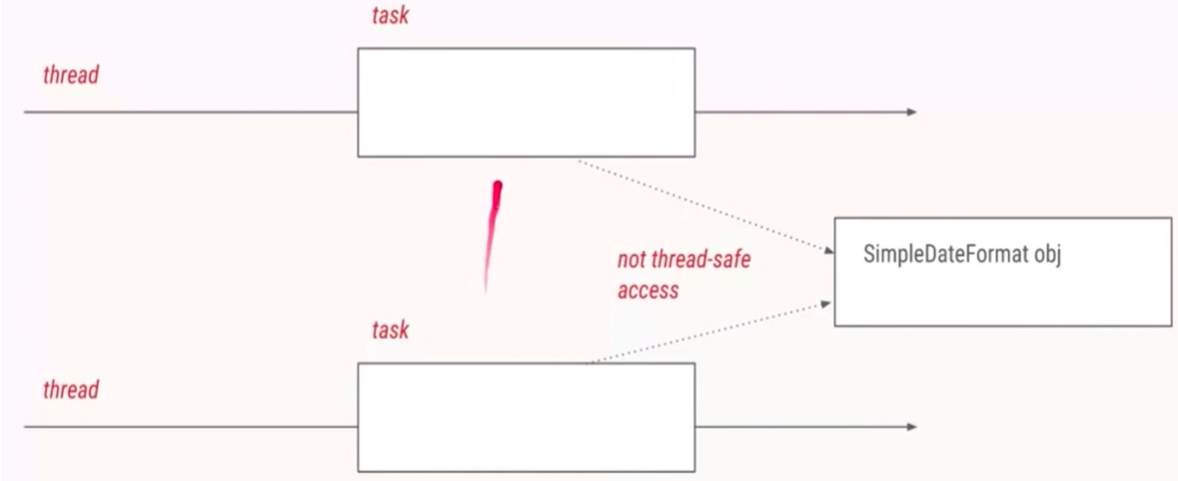
使用SimpleDateFormat演示场景一：多个线程共用SimpleDateFormat打印日期出现的问题

使用static SimpleDateFormat会打印出有相同时间

```
package threadlocal;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 描述： 1000个打印日期的任务，用线程池来执行,使用static SimpleDateFormat发现打印出有相同的 时间 发生了线程安全问题
 */
public class ThreadLocalNormalUsage03 {

    public static ExecutorService threadPool = Executors.newFixedThreadPool(10);
    static SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            int finalI = i;
            threadPool.submit(new Runnable() {
                @Override
                public void run() {
                    String date = new ThreadLocalNormalUsage03().date(finalI);
                    System.out.println(date);
                }
            });
        }
        threadPool.shutdown();
    }
```

```
29
30      public String date(int seconds) {
31          //参数的单位是毫秒，从1970.1.1 00:00:00 GMT计时
32          Date date = new Date(1000 * seconds);
33          return dateFormat.format(date);
34      }
35  }
```

这个问题是由于多个线程指向同一个SimpleDateFormate对象造成的



解决方案：

1. 给format方法加锁

```
1  synchronized (ThreadLocalNormalUsage04.class) {
2      s = dateFormat.format(date);
3  }
```

这种方式会导致后面线程排队等待进入这个方法，十分影响性能，不推荐

2. 使用ThreadLocal，给每个线程分配自己的dateFormat对象，保证了线程安全，高效利用内存

```
1  package threadlocal;
2
3  import java.text.SimpleDateFormat;
4  import java.util.Date;
5  import java.util.concurrent.ExecutorService;
6  import java.util.concurrent.Executors;
7
8  /**
9   * 描述：     利用ThreadLocal，给每个线程分配自己的dateFormat对象，保证了线程安全，高效利用内存
10  */
11 public class ThreadLocalNormalUsage05 extends ThreadLocal<SimpleDateFormat>{
12
13      // 这样没法设置为static的,无法在静态方法中使用
14      @Override
15      protected SimpleDateFormat initialValue() {
16          return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
17      }
18
```

```
19      public static ExecutorService threadPool =
        Executors.newFixedThreadPool(10);
20
21      public static void main(String[] args) throws InterruptedException
        {
22          for (int i = 0; i < 1000; i++) {
23              int finalI = i;
24              threadPool.submit(new Runnable() {
25                  @Override
26                  public void run() {
27                      String date = new
                        ThreadLocalNormalUsage05().date(finalI);
28                      System.out.println(date);
29                  }
30              });
31          }
32          threadPool.shutdown();
33      }
34
35      public String date(int seconds) {
36          //参数的单位是毫秒，从1970.1.1 00:00:00 GMT计时
37          Date date = new Date(1000 * seconds);
38  //          SimpleDateFormat dateFormat1 = new SimpleDateFormat("yyyy-MM-
        dd HH:mm:ss");
39          SimpleDateFormat dateFormat2 =
            ThreadSafeFormatter.dateFormatThreadLocal2.get();
40  //          SimpleDateFormat dateFormat3 = this.get();
41          return dateFormat2.format(date);
42      }
43  }
44
45  class ThreadSafeFormatter {
46
47      public static ThreadLocal<SimpleDateFormat> dateFormatThreadLocal =
        new ThreadLocal<>() {
48          @Override
49          protected SimpleDateFormat initialValue() {
50              return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
51          }
52      };
53
54      public static ThreadLocal<SimpleDateFormat> dateFormatThreadLocal2
        = ThreadLocal
55              .withInitial(() -> new SimpleDateFormat("yyyy-MM-dd
                HH:mm:ss"));
56  }
```

3. 使用DateTimeFormatter

使用旧的`Date`对象时，我们用`SimpleDateFormat`进行格式化显示。使用新的`LocalDateTime`或`ZonedLocalDateTime`时，我们要进行格式化显示，就要使用`DateTimeFormatter`。

和`SimpleDateFormat`不同的是，`DateTimeFormatter`不但是不变对象，它还是线程安全的。现在我们只需要记住：因为`SimpleDateFormat`不是线程安全的，使用的时候，只能在方法内部创建新的局部变量。而`DateTimeFormatter`可以只创建一个实例，到处引用。

创建`DateTimeFormatter`时，我们仍然通过传入格式化字符串实现：

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
```

格式化字符串的使用方式与`SimpleDateFormat`完全一致。

另一种创建`DateTimeFormatter`的方法是：传入格式化字符串时，同时指定`Locale`：

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("E, yyyy-MMMM-dd HH:mm", Locale.US);
```

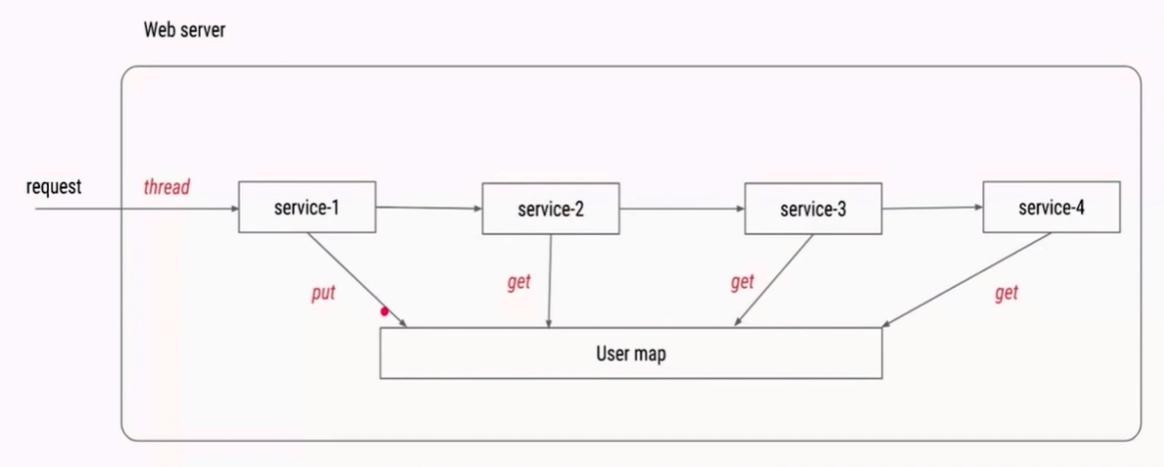
这种方式可以按照`Locale`默认习惯格式化。我们来看实际效果：

```
import java.time.*; import java.time.format.*; import java.util.Locale;
```

场景二：当前用户信息需要被线程内所有方法共享

*比如用户下订单，扣款，使用优惠券... 等方法：一个比较繁琐的解决方案是把user作为参数**层层传递**,从service-1（）传到service-2（）,再从 service-2（）传到service-3（）,以此类推,但是这样做会导致代码冗余且不易维护*

解决方案一：使用userMap



但要注意普通的map也是线程不安全的，我们必须要使用concurrentHashMap，或者加锁：这样一定是消耗性能的。

可以用 static 的 Concurrenthashmap,把当前线程的ID作为key,把user作为 value来保存，这样可以做到线程间的隔离,但是依然有性能影响。

方案二：ThreadLocal

更好的が法是使用 ThreadLocal,这样无需 synchronized,可以在不影响性能的情况下,也无需层层传递参数,就可达到保存当前线程对应 的用户信息的目的

- 强调的是同一个请求内(**同个线程**内)**不同方法间的共享**
- 不需重写 `initialvalue()`方法,但是必须手动调用`set()`方法

```
package threadlocal;

/**
 * 描述：       演示ThreadLocal用法2：避免传递参数的麻烦
 */
public class ThreadLocalNormalUsage06 {
```

```
7
8       public static void main(String[] args) {
9           new Service1().process("");
10
11      }
12  }
13
14  // 在service1中放入user对象
15  class Service1 {
16
17      public void process(String name) {
18          User user = new User("超哥");
19          UserContextHolder.holder.set(user);
20          new Service2().process();
21      }
22  }
23
24  // 在service2中可以取到threadLocal中的user对象
25  class Service2 {
26
27      public void process() {
28          User user = UserContextHolder.holder.get();
29          ThreadSafeFormatter.dateFormatThreadLocal.get();
30          System.out.println("Service2拿到用户名：" + user.name);
31          new Service3().process();
32      }
33  }
34
35  // 在service3中可以取到threadLocal中的user对象
36  class Service3 {
37
38      public void process() {
39          User user = UserContextHolder.holder.get();
40          System.out.println("Service3拿到用户名：" + user.name);
41          UserContextHolder.holder.remove();
42      }
43  }
44
45  class UserContextHolder {
46
47      // 这里我们只是生产一个ThreadLocal对象，并不像前面一个场景一样，对其进行初始化，而
    是在需要的位置为其赋值。
48      public static ThreadLocal<User> holder = new ThreadLocal<>();
49
50  }
51
52  class User {
53
54      String name;
55
56      public User(String name) {
57          this.name = name;
58      }
59  }
```

2. 总结ThreadLocal的两个作用

1. 让某个需要用到的对象在**线程间隔离**(每个线程都有自己的独立的对象)
2. 在任何方法中都可以**轻松获取**到该对象

3. 使用ThreadLocal带来的好处

1. 达到线程安全
2. 不需要加锁，提高执行效率
3. 更高效地利用内存、节省开销：相比于每个任务都新建一个SimpleDateFormat，显然用ThreadLocal可以节省内存和开销。
4. 免去传参的繁琐：无论是场景一的工具类，还是场景二的用户名，都可以在任何地方直接通过ThreadLocal拿到，再也不需要每次都传同样的参数。ThreadLocal使得代码耦合度更低，更优雅。

4. ThreadLocal主要方法介绍

T initialValue( )： 返回ThreadLocal传入的泛型。

1. 该方法会返回当前线程对应的“初始值”，这是一个延迟加载的方法，只有在调用get的时候，才会触发。

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry( key: this);
        if (e != null) {
            /unchecked/
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

进入`setInitialValue()中可以看到

```
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);
    }
    if (this instanceof TerminatingThreadLocal) {
        TerminatingThreadLocal.register((TerminatingThreadLocal<?>) this);
    }
    return value;
}
```

这里调用了initialValue()方法

、

2. 当线程第一次使用get方法访问变量时，将调用此方法，除非线程先前调用了set方法，在这种情况下，不会为线程调用本initialValue方法。这正对应了ThreadLocal的两种典型用法。

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry( key: this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

如果有过向ThreadLocal中set值，那么这两层判断必然都不为空，返回ThreadLocalMap中实体的value

3. 通常，每个线程最多调用一次此方法，但如果已经调用了remove()后，再调用get()，则可以再次调用此方法。
4. 如果不重写本方法，这个方法会返回null。**一般使用匿名内部类的方法来重写initialize()方法**，以便在后续使用中可以初始化副本对象。

void set(T t)

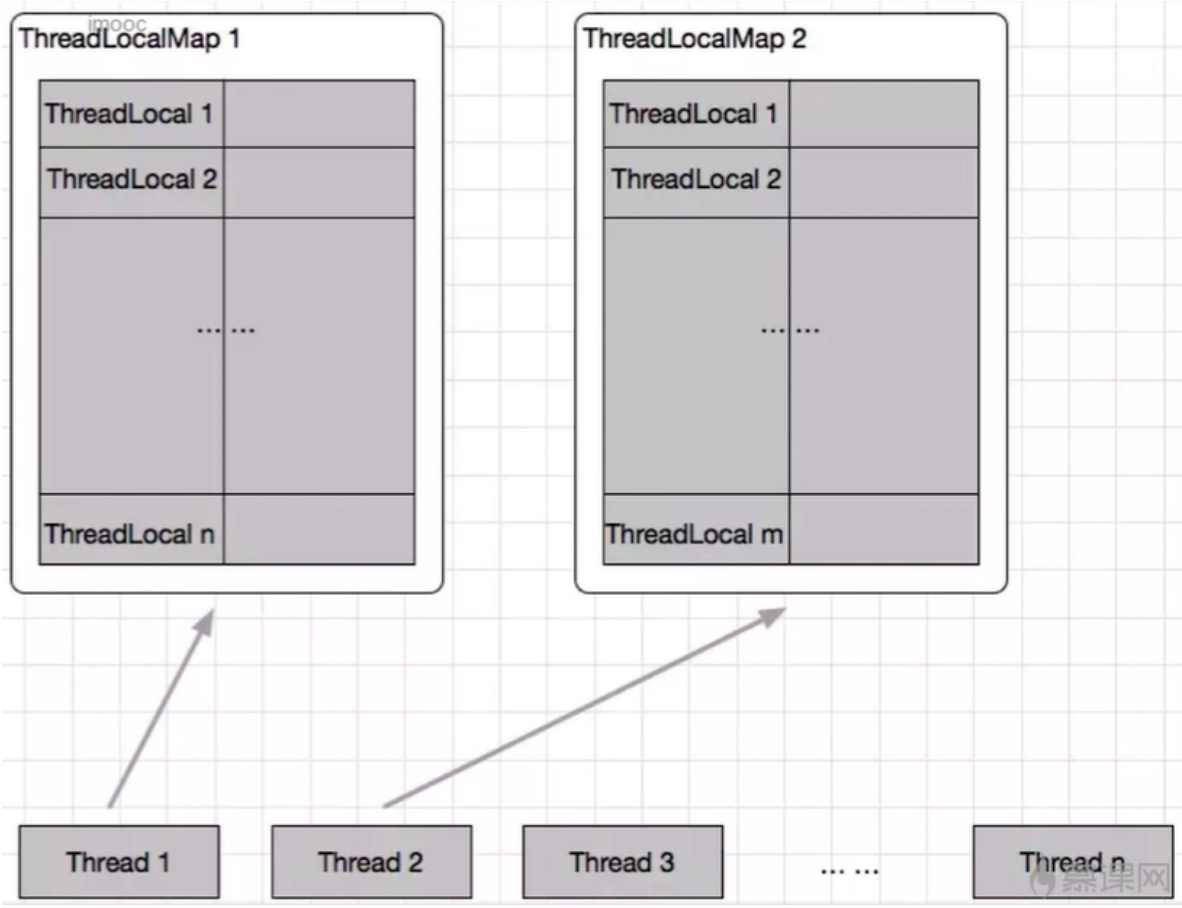
T get( )

void remove( )

5. ThreadLocal源码分析

- 搞清楚 Thread、ThreadLocal以及ThreadLocalMap三者之间的关系
- 每个 Thread对象中都持有一个ThreadLocalMap成员变量
- 每个ThreadLocalMap都是以当前ThreadLocal对象作为key，set进去的值作为value

  存储在ThreadLocalMap内的就是一个以Entry为元素的table数组，Entry就是一个key-value结构，key为ThreadLocal，value为存储的值。类比HashMap的实现，其实就是每个线程借助于一个哈希表，存储线程独立的值



进入Thread类，可以看到有ThreadLocalMap成员变量

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

ThreadLocalMap 类，也就是Thread.threadLocals

HashMap解决冲突的方式是拉链法，Threadlocalmap这里采用的是**线性探测法**，也就是如果发生冲突，就**继续找下一个空位置**,而不是用链表拉链，所以不存在链表和红黑树这样的数据结构。

一个ThreadLocalMap会存储很多的threadLocals，因为一个线程可能拥有多个ThreadLocal对象，我们来仔细看一下Thread类中的ThreadLocalMap类

```
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;
    }

    /**
     * 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);
    }

    // 各种方法 ...
}
```

ThreadLocalMap被定义为一个静态类，上面是包含的主要成员：

1. 首先是Entry的定义，前面已经说过；
2. 初始的容量为`INITIAL_CAPACITY = 16`；
3. 主要数据结构就是一个Entry的数组**table**；
4. size用于记录Map中实际存在的entry个数；
5. threshold是扩容上限，当size到达threashold时，需要resize整个Map，threshold的初始值为`len * 2 / 3`相当于HashMap中的负载因子。；
6. nextIndex和prevIndex则是为了安全的移动索引，后面的函数里经常用到。

ThreadLocal.get方法

```
public T get() {
    // 获取到当前线程
    Thread t = Thread.currentThread();、
    // 获取到该线程的ThreadLocalMap对象，每个线程（Thread类）都有一个ThreadLocalMap成员变量，没有set值前为null
    ThreadLocalMap map = getMap(t);
    // 如果有调用threadLocal的set方法，为其设值，则map不为空
    // 而这个map的key就是当前ThreadLocal对象
    if (map != null) {
        // this: 当前ThreadLocal对象,获取以当前ThreadLocal对象为key的map entity
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            // 获取set到该ThreadLocal（this）的值
            T result = (T)e.value;
            // 进行返回
            return result;
        }
    }
    // 没有set过值，调用初始化value的方法
    return setInitialValue();
}
```

ThreadLocal.setInitialValue:

setInitialValue在Map不存在的时候调用

```
private T setInitialValue() {
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        // 调用下面的set方法
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

1. 首先是调用initialValue生成一个初始的value值，深入initialValue函数，我们可知它就是返回一个null;
2. 然后还是在get以下Map，如果map存在，则直接map.set, 看下面的set方法
3. 如果不存在则会调用createMap创建ThreadLocalMap

ThreadLocal.set方法（setInitialValue方法很类似）

```
public void set(T value) {
    // 获取到当前线程
    Thread t = Thread.currentThread();
    //传入当前线程，获取到该线程的ThreadLocalMap对象
    ThreadLocalMap map = getMap(t);
    // 如果已经存在map
    if (map != null) {
```

```
        // 把当前ThreadLocal对象作为key，传入的参数作为value放入ThreadLocalMap中
        map.set(this, value);
    } else {
        // 如果不存在map，创建当前线程的ThreadLocalMap
        createMap(t, value);
    }
}
```

进入ThreadLocal类的`createMap`方法：

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

进入ThreadLocal类的`new ThreadLocalMap(this, firstValue);`

就是使用firstKey和firstValue创建一个Entry，计算好索引i，然后把创建好的Entry插入table中的i位置，再设置好size和threshold。

```
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    // 创建一个默认大小为16的数组
    table = new Entry[INITIAL_CAPACITY];
    // 计算i作为数组下标
    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
    // firstKey是上面create方法中传入的this：当前ThreadLocal对象
    table[i] = new Entry(firstKey, firstValue);
    size = 1;
    setThreshold(INITIAL_CAPACITY);
}
```

通过源码分析可以看出, setInitialValue( )和直接set**最后都是利用map.set()方法来设置值**也就是说,最后都会对应到 ThreadLocalMap的一个 Entry,**只不过是起点和入口不一样**,前者是通过get的时候发现ThreadLocalMap为空而调用的

ThreadLocal.initialValue方法

initialvalue方法:是没有默认实现的,如果我们要用initialvalue方法;需要自己实现,通常是匿名内部类的方式(回顾代码)

ThreadLocal.ThreadLocalMap.remove方法

最后一个需要探究的就是remove函数，它用于在map中移除一个不用的Entry。也是先计算出hash值，若是第一次没有命中，就循环直到null，在此过程中也会调用expungeStaleEntry清除空key节点。代码如下：

```
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;
        }
    }
}
```

ThreadLocalMap 类，也就是Thread.threadLocals

HashMap解决冲突的方式是拉链法，Threadlocalmap这里采用的是**线性探测法**，也就是如果发生冲突,就**继续找下一个空位置**,而不是用链表拉链，所以不存在链表和红黑树这样的数据结构。

6. 使用ThreadLocal注意点

1. 内存泄漏

什么是内存泄漏

内存泄漏:某个对象不再有用,但是占用的内存却不能被回收

这会导致这部分内存始终被占有，如果程序有很多内存泄漏的情况，就有可能导致OOM的发生。

Key的泄漏

我们在看entry的数据结构：

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

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

我们可以看到，Entry是继承了WeakReference弱引用的，而Entry中key的设值是通过super调用父类方法的，我们点进入看super（k）

```
public WeakReference(T referent) {
    super(referent);
}
```

弱引用的特点是，如果这个对象**只**被弱引用关联(**没有任何强引用关联**，我们通常的赋值都是强引用),那么这个对象就**可以被GC回收**

Value的泄漏

Thread Localmap的每个 Entry都是一个**对key的弱引用**,同时每个 Entry都包含了一个**对 valuel的强引用**

正常情况下,当线程终止,保存在 Thread Local里的 value会被垃圾回收,因为没有任何强引用了

但是,**如果线程不终止**(比如线程需要保持很久)，那么key对应的value就不能被回收，因为有以下的调用链：

因为 value和 Thread之间还存在这个强引用链路,所以导致 value无法回收,就可能会出现OOM

JDK的设计已经考虑到了这个问题，所以在set()、remove()、resize()方法中会扫描到key为null的Entry，并且把对应的value设置为null，这样value对象就可以被回收。

但是如果一个 Thread Local不被使用,那么实际上set, remove,rehash方法也不会被调用,如果同时线程又不停止,那么调用链就一直存在,那么就导致了 valuel的内存泄漏

如何避免内存泄露（阿里规约）

调用remove()方法，就会删除对应的Entry对象，可以避免内存泄漏，所以使用完ThreadLocal后，要调用remove()方法。

2. ThreadLocal的空指针异常问题

```
/**
 * ThreadLocal的空指针异常问题
 */
public class ThreadLocalNPE {

    ThreadLocal<Long> longThreadLocal = new ThreadLocal<>();

    public void set() {
        longThreadLocal.set(Thread.currentThread().getId());
    }

    public Long get() {
        return longThreadLocal.get();
    }

    public static void main(String[] args) {

        ThreadLocalNPE threadLocalNPE = new ThreadLocalNPE();

        //如果get方法返回值为基本类型，则会报空指针异常，如果是包装类型就不会出错
        System.out.println(threadLocalNPE.get());

        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                threadLocalNPE.set();
                System.out.println(threadLocalNPE.get());
            }
        });
        thread1.start();
    }
}
```

空指针异常问题的解决

如果get方法返回值为基本类型，则会报空指针异常，如果是包装类型就不会出错。这是因为基本类型和包装类型存在装箱和拆箱的关系，造成空指针问题的原因在于使用者。

共享对象问题

如果在每个线程中ThreadLocal.set()进去的东西本来就是多个线程共享的同一对象，比如static对象，那么多个线程调用ThreadLocal.get()获取的内容还是同一个对象，还是会发生线程安全问题。

可以不使用ThreadLocal就不要强行使用

如果在任务数很少的时候，在局部方法中创建对象就可以解决问题，这样就不需要使用ThreadLocal。

优先使用框架的支持，而不是自己创造

例如在Spring框架中，如果可以使用RequestContextHolder，那么就不需要自己维护ThreadLocal，因为自己可能会忘记调用remove()方法等，造成内存泄漏。

3. 共享对象

如果在每个线程中 Threadloca.et(0进去的东西本来就是多线程共享的同一个对象,比如 static对象,那么多个线程的 Thread Local.get0取得的还是这个共享对象本身,还是有并发访问问题。

4. 如果可以不使用ThreadLocal就解决问题，那么不要强行使用

5. 优先使用框架的支持，而不是自己创造

例如在 Spring中,如果可以使用 RequestContextHolder,那么就不需要自己维护 Thread Local,因为自己可能会忘记调用remove方法等,造成内存泄漏

7. 实际应用场景——在Spring中的实例分析

- Datetime Contextholdera类,看到里面用了 Thread Local
- 每次HTTP请求都对应一个线程,线程之间相互隔离,这就是Thread Locale的典型应用场景

本文仅为自己学习时记下的笔记，参考自慕课：https://coding.imooc.com/class/409.html