在java多线程开发中ThreadLocal（线程本地变量）是必备知识点，ThreadLocal在每个线程中都创建了变量的副本，实现了多线程中变量的隔离，在多线程环境下保证了成员变量的安全。常用的应用场景有：数据库连接，session会话连接。让我们来深入了解下ThreadLocal吧。

1、ThreadLocal的简单应用

2、ThreadLocal源码

3、ThreadLocalMap的结构和存取

4、解决ThreadLocal内存泄漏

第1、2两部分内容很简单，第3部分内容是这篇文章的核心，也是ThreadLocal最核心的内容。

1、ThreadLocal简单应用

让我们通过一个简单的例子看下ThreadLocal是怎么用的吧

@Slf4j  
public class MgDemo20200628 implements Runnable {  
  
 private static ThreadLocal<String> *local*=new ThreadLocal<>();  
 @Override  
 public void run() {  
 try {  
 *local*.set(Thread.*currentThread*().getName() +"---" + System.*currentTimeMillis*());  
 Thread.*sleep*(1000\*5);  
 *log*.info(*local*.get());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 public static void main(String[] args) {  
 MgDemo20200628 demo =new MgDemo20200628();  
 for(int i=0;i<3;i++)  
 {  
 Thread task = new Thread(demo);  
 task.start();  
 }  
 }  
}

运行一下，控制台的输出结果

22:37:17.293 [Thread-1] INFO com.mg.empty.demo.thread.threadlocal.MgDemo20200628 - Thread-1---1593355032290

22:37:17.293 [Thread-2] INFO com.mg.empty.demo.thread.threadlocal.MgDemo20200628 - Thread-2---1593355032290

22:37:17.293 [Thread-0] INFO com.mg.empty.demo.thread.threadlocal.MgDemo20200628 - Thread-0---1593355032290

由执行结果可知，每个线程通过ThreadLocal获取到的String都是自己线程副本的内容，各线程有很好的隔离。

注意：ThreadLocal通常都定义为static，ThreadLocal没有存储功能，变量副本的真实存储位置是Thread对象的threadLocals这个ThreadLocal.ThreadLocalMap变量中，可以将ThreadLocal理解为一个工具类，用来保证线程本地变量的存储和存储碰撞。

让我们一块研究下ThreadLocal的具体实现，来更深入了解下。

二、ThreadLocal源码简述

ThreadLocal源码中常用的方法有get、set和remove，这几个函数的源码很简单

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();  
}

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

这几个函数中都要先调用getMap这个函数来获取ThreadLocalMap对象，然后操作这个map对象来实现变量的存取。看下getMap这个函数

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

**这是获取Thread对象中的threadLocals变量，threadLocals是ThreadLocal.ThreadLocalMap类型。ThreadLocalMap的存储位置是在Thread线程中，不是在ThreadLocal对象中, ThreadLocal只是作为ThreadLocalMap的key来存取变量。一定要记住ThreadLocalMap是存储在Thread中，这个是理解ThreadLocal原理很重要的前提条件。**

让我们一块研究下ThreadLocalMap这个内部类，来一块揭晓一下ThreadLocal的真正神秘所在。

三、ThreadLocalMap结构解析

可以通过下图了整体了解一下ThreadLocalMap的结构和使用



ThreadLocalMap是Map对象，Entry继承了WeakReference<ThreadLocal<?>>，即key是ThreadLocal对象也是弱引用(这是TreadLocal会内存泄漏的原因)，value是对应的变量。代码如下

static class Entry extends WeakReference<ThreadLocal<?>> {Object value;  
  
 Entry(ThreadLocal<?> k, Object v) {  
 super(k);  
 value = v;  
 }  
}

既然是map结构，那么就存在避免碰撞的问题，这个是如何解决的呢，在TreadLocal的解决方案如下：

private static AtomicInteger *nextHashCode* =new AtomicInteger();

private static final int *HASH\_INCREMENT* = 0x61c88647;

private static int nextHashCode() {  
 return *nextHashCode*.getAndAdd(*HASH\_INCREMENT*);  
}

private final int threadLocalHashCode = *nextHashCode*();

在创建一个新的ThreadLocal对象，都会对nextHashCode加0x61c88647（这个值与斐波那契散列法以及黄金分割有关，具体的原理没有研究，使value能均匀的分布来减少碰撞）作为hash码来减少碰撞，每一个ThreadLocal对象的threadLocalHashCode都是不一样的，并且都相差n\* 0x61c88647。

这样只是减少了碰撞的几率，不是不会碰撞了，如果发生碰撞怎么办呢，看下ThreadLocalMap的set方法

private void set(ThreadLocal<?> key, Object value) {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();  
}

set的操作过程是先用hash码与长度取余，然后找到对应数组位置index，如果index位置为空，生成entry对象，并存储；如果不为空，index++直到找到空的位置，存放entry对象。

简单讲，就是如果没有发生碰撞，就按约定位置存储，如果发生碰撞，就顺序找个空的位置存储。个人理解，ThreadLocal中不会有太大数据量的存储，所以设计ThreadLocalMap的时候，有大量方法存在的遍历数组的操作。

注意：ThreadLocalMap中存储的是该线程所有ThreadLocal对象（一个线程中定义的多个ThreadLocal数据都是存在这个Entry数组中），不是每新建一个ThreadLocal对象都会产生一个新的ThreadLocalMap。Map中的碰撞是由单个线程不同ThreadLocal存储变量数据时候引起的。

同样get方法，也是先定位index，如果找不到会遍历数组

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);  
}

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;  
}

在getEntryAfterMiss这个函数中会调用expungeStaleEntry（清空key为null的entry对象，防止内存泄漏（第四部分详细说明））这个方法。同样remove方法也是相同的遍历过程

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;  
 }  
 }  
}

有兴趣的可以看下expungeStaleEntry函数的源码， ThreadLocal中的get、set、remove方法往下跟的话，都会进这个函数。

private int expungeStaleEntry(int staleSlot) {  
 Entry[] tab = table;  
 int len = tab.length;  
  
 *// expunge entry at staleSlot* tab[staleSlot].value = null;  
 tab[staleSlot] = null;  
 size--;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;while (tab[h] != null)  
 h = *nextIndex*(h, len);  
 tab[h] = e;  
 }  
 }  
 }  
 return i;  
}

这样ThreadLocalMap就基本明白了，接下来让我们看下为什么ThreadLocal会存在内存溢出

四、ThreadLocal引起的内存泄漏

1、什么是内存泄漏

内存泄漏，即已经申请的内存空间无法释放，就会造成内存泄漏，例如线程死循环、资源不关闭等，大量的内存泄漏堆积就会造成内存不够，从而发生内存溢出。

2、TreadLocal内存泄漏的场景

TreadLocalMap的Entry继承了WeakReference<ThreadLocal<?>>，即Entry的key是弱引用（该引用只存在弱引用的情况下，下一次GC会回收这个引用对象，可以参考这篇文章了解）。让我们一起看下如下应用场景：当线程存在多个ThreadLocal时，其中的一个或几个ThreadLocal的引用在线程运行中，在没有执行remove的情况下被设置为null，即释放了强引用，ThreadLocal对象只有Entry的弱引用了。下次gc被回收，key是弱引用被回收，value是强引用一直存储在内存中。线程还在运行，value的内存就不会释放，这样发生了内存泄漏。

单个线程的情况下，线程结束的时候，内存泄漏的value会被回收，不会有什么问题。不过如果使用的线程池呢，线程是常备线程，会一直运行下去，这样value内存会被一直占用，并且可能会不断增加，这样就严重了。那需要怎么来避免和解决呢。

3、解决方法

这个没有太好的解决方案，只能通过规范代码使用来解决，上一章中说到expungeStaleEntry方法，会将这种key引用已经失效的entry对象释放。所以为了避免内存泄漏，在ThreadLocal使用结束以后，规范操作，执行下remove方法。