ThreadLocal是一个泛型类，泛型表示ThreadLocal可以存储的类型，它的使用非常简单。举个例子，在子线程中用ThreadLcoal存储一个数字，然后分别在子线程和主线程将中来获取这个值，代码如下：

public static void main(String[] args) {

ThreadLocal<Integer> threadLocal = new ThreadLocal<>();

new Thread(() -> {

threadLocal.set(10);

try {

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() + " value = " + threadLocal.get());

} catch (InterruptedException e) {

e.printStackTrace();

}

}).start();

try {

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() + " value = " + threadLocal.get());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

上述代码的打印结果如下：

Thread-0 value = 10

main value = null

可以看到，我们在子线程中通过ThreadLocal存储了一个10，则子线程中可以取到这个值。而主线程中取到的却是null。这意味着通过某个线程通过ThreadLocal存储的数据，只有在这个线程中才能访问的到。

除此之外，ThreadLocal可以设置全局的初始值，代码如下：

ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 10);

通过ThreadLocal的withInitial方法指定初始值为10，接着分别从子线程和主线程中取值，打印结果如下：

Thread-0 value = 10

main value = 10

除了set和get方法之外，ThreadLocal还提供了remove方法，使用很简单这里就不再列举代码了。

1. ThreadLocal的set过程

set方法的源码比较简单，如下：

public void set(T value) {

// 获取当前线程

Thread t = Thread.currentThread();

// 获取线程中的ThreadLocalMap

ThreadLocalMap map = getMap(t);

if (map != null) {

// 将值存储到ThreadLocalMap中

map.set(this, value);

} else {

// 创建ThreadLocalMap，并存储值

createMap(t, value);

}

void createMap(Thread t, T firstValue) {

// 实例化当前线程中的ThreadLocalMap

t.threadLocals = new ThreadLocalMap(this, firstValue);

}

上述代码首先获取到了当前线程，然后从当前线程中获取ThreadLocalMap，ThreadLocalMap是一个存储K-V的集合，我们后边分析。如果此时ThreadLocalMap不为空，那么就通过ThreadLocalMap的set方法将值存储到当前线程对应的ThreadLocalMap中。如果ThreadLocalMap为空，那么就创建ThreadLcoalMap，然后将值存储到ThreadLocalMap中。并且，这里我们注意到ThreadLocalMap的key是当前的ThreadLocal。

1. ThreadLocal的get过程

接下来我们看如何从ThreadLocal中取出数据，get方法的代码如下：

public T get() {

// 获取当前线程

Thread t = Thread.currentThread();

// 获取当前线程对应的ThreadLocalMap

ThreadLocalMap map = getMap(t);

if (map != null) {

// 从ThreadLocalMap中取出值

ThreadLocalMap.Entry e = map.getEntry(this);

if (e != null) {

@SuppressWarnings("unchecked")

T result = (T)e.value;

return result;

}

}

// 如果值为空则返回初始值

return setInitialValue();

}

// 为ThreadLocal设置初始值

private T setInitialValue() {

// 初始值为null

T value = initialValue();

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null) {

map.set(this, value);

} else {

// 创建ThreadLocalMap

createMap(t, value);

}

if (this instanceof TerminatingThreadLocal) {

TerminatingThreadLocal.register((TerminatingThreadLocal<?>) this);

}

return value;

}

// 初始值为空

protected T initialValue() {

return null;

}

可以看到，get方法依然是先获取到当前线程，然后拿到当前线程的ThreadLocalMap，并通过ThreadLocalMap的getEntry方法将这个ThreadLocal作为key来取值。如果ThreadLocalMap为null，则会通过setInitialValue方法返回了一个null值。

总的来看，set方法将value放到了当前线程的ThreadLocalMap中，而key是当前的这个ThreadLocal。而get方法则是获取当前线程中的ThreadLcoalMap，然后将这个ThreadLocal作为key来取出value。到这里其实我们已经能够解答为什么ThreadLocal中的值只能被设置这个值的线程可见了。

1. ThreadLocalMap

从前两小节其实我们已经知道，ThreadLocalMap是一个存储K-V类型的数据结构，并且Thread类中维护了一个ThreadLocalMap的成员变量。代码如下：

public class Thread implements Runnable {

ThreadLocal.ThreadLocalMap threadLocals = null;

...

}

可以看到ThreadLocalMap是ThreadLocal的内部类,ThreadLocalMap的类结构如下：

static class ThreadLocalMap {

private Entry[] table;

private int size = 0;

}

ThreadLocalMap内部维护了一个Entry数组，和一个int类型的size。Entry是ThreadLocalMap的内部类，它就是对我们设置的value的封装，代码如下：

static class Entry extends WeakReference<ThreadLocal<?>> {

Object value;

Entry(ThreadLocal<?> k, Object v) {

super(k);

value = v;

}

}

可以看到Entry类的结构很简单，它继承了WeakReference，并且内部维护了一个Object类型的value。而WeakReference中维护了一个referent的成员，在Entry中就是指ThreadLocal。也就是说Entry中维护了一个ThreadLocal作为key和一个Object的value作为value。

接下来看ThreadLocalMap的set方法。

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

Entry[] tab = table;

int len = tab.length;

// 获取key哈希值，作为在Entry数组中的位置

int i = key.threadLocalHashCode & (len-1);

// 出现哈希冲突，这里使用的是线性探测再散列方法来处理

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;

}

}

// 将key和value封装到Entry中，并放入Entry数组

tab[i] = new Entry(key, value);

int sz = ++size;

if (!cleanSomeSlots(i, sz) && sz >= threshold)

rehash();

}

通过ThreadLocalMap的set方法可以看出，ThreadLocalMap是一个哈希表结构。set方法是将value插入到哈希表中的操作。我们知道哈希表是会出现哈希冲突的，因此，上述代码首先使用线性探测再散列法进行哈希冲突的处理，然后再将value封装成Entry，插入到Entry数组中。

接下来看ThreadLocalMap的getEntry方法，这里不用想也应该知道getEntry方法一定是从哈希表中取数据的。它的代码如下：

private Entry getEntry(ThreadLocal<?> key) {

int i = key.threadLocalHashCode & (table.length - 1);

Entry e = table[i];

// 不存在哈希冲突的情况，取到了值

if (e != null && e.get() == key)

return e;

else

// 存在哈希冲突的情况，则通过线性探测法来查找值

return getEntryAfterMiss(key, i, e);

}

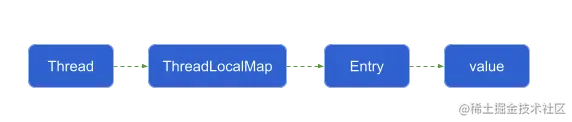
由于是从哈希表中取值，所有这个方法中一定存在两种情况，即存在哈希冲突和不存在哈希冲突。首先，如果不存在哈希冲突，那么直接从Entry数组中取出第i个元素即可。而如果存在哈希冲突，那么则需要继续线性探测来查找key的位置。getEntryAfterMiss就是线性探测的实现，无非就是循环遍历然后比较，这里就不再贴这个方法的代码了。

如果了解哈希表的话，看懂ThreadLocalMap的代码其实并不难。但是这里有一个问题，为什么ThreadLocalMap中的Entry要继承WeakReference呢？

我们知道，弱引用在发生GC时这个对象一定会被回收。通常来说使用弱引用是为了避免内存泄漏。这里也不例外，ThreadLocal使用弱引用可以避免内存泄漏问题的发生。

试想，如果将ThreadLocal声明为强引用，一旦ThreadLocal不再使用，就需要被回收。但是此时由于ThreadLocalMap中的Entry数组持有了ThreadLocal。导致ThreadLocal不能够被回收而出现内存泄漏。那么，如果将ThreadLocal声明为弱引用就可以避免这一问题的出现。

那么，是否意味着将Entry中的ThreadLocal声明为弱引用，我们就可以肆无忌惮的使用ThreadLocal也不会出现内存泄漏了？事实并非如此。



如上图所示，在ThreadLocal中存在一个这样的引用连。如果Thread一直在运行，那么此时由于强引用的value不能被回收，故此种情况下也可能出现内存泄漏的问题。因此，通常来说，在不需要使用这个ThreadLocal变量的使用，需要调用remove方法来避免内存泄漏的问题。

从源码角度来看ThreadLocal,在了解哈希表的情况下，弄懂它的实现原理其实并不难。ThreadLocal的set方法会将自身作为key，连带value封装到Entry中。然后将这个Entry插入到当前线程的ThreadLocalMap中。这个ThreadLocalMap是一个哈希表结构，内部使用线性探测再散列来存储Entry。

当然，由于可能存在多个ThreadLocal的情况，如下代码：

ThreadLocal<Integer> threadLocal = new ThreadLocal<>();

ThreadLocal<Integer> threadLocal2 = new ThreadLocal<>();

threadLocal.set(1);

threadLocal2.set(2);

因此，可以给出ThreadLocalMap的结构图如下：

