1、ThreadLocal是什么？

2、ThreadLocal怎么用？

3、ThreadLocal源码分析？

4、ThreadLocal内存泄漏问题？

1、ThreadLocal是什么？

ThreadLocal可以理解为：线程变量或者本地线程存储。在每个线程（Thread）中都创建了一个 ThreadLocalMap 对象，每个线程可以访问自己内部 ThreadLocalMap 对象内的 value。 意思是：ThreadLocal中填充的变量是属于当前线程的，该变量对于其他线程而言是隔离的。ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程都可以访问自己内部的副本变量。

使用的场景：

1、在进行对象跨层传递的时候，使用ThreadLocal可以避免多次传递，打破层次间的约束。

2、线程间数据隔离

3、进行事务操作，用于存储线程事务信息。

4、数据库连接，Session回话管理。

2、ThreadLocal怎么用？

ThreadLocal的作用主要是做数据隔离，填充的数据和值属于当前线程，变量的数据对别的线程而言是相对隔离的，在多线程环境下，如何防止自己的变量被其他线程篡改。

ThreadLocal的隔离有什么用，会用在什么场景？

Spring实现事务隔离级别的源码中有使用。

Spring采用ThreadLocal的方式，来保证单个线程中的数据库操作使用的是用一个数据库连接，同时，采用这种方式可以使业务层使用事务时不需要感知并管理connection对象，通过传播级别，巧妙地管理多个事务配置之间的切换、挂起和恢复。

（来保证对于数据的操作即事务的开启，提交或者回滚是各自对应的连接，避免出现B线程 关掉了 A线程的连接）

Spring的事务主要是ThreadLocal和AOP实现的。

基础概念：

ThreadLocal 是线程本地存储，在每个线程（Thread）中都创建了一个 ThreadLocalMap 对象，每个线程可以访问自己内部 ThreadLocalMap 对象内的 value。

ThreadLocal类允许我们创建只能被同一个线程读写的变量。

线程间数据隔离。

ThreadLocal类顾名思义可以理解为线程本地变量。也就是说如果定义了一个ThreadLocal，每个线程往这个ThreadLocal中读写是线程隔离，互相之间不会影响的。它提供了一种将可变数据通过每个线程有自己的独立副本从而实现线程封闭的机制。

经典的使用场景是为每个线程分配一个 JDBC 连接 Connection。这样就可以保证每个线程的都在各自的 Connection 上进行数据库的操作，不会出现 A 线程关了 B线程正在使用的 Connection； 还有 Session 管理 等问题。

ThreadLocal.ThreadLocalMap threadLocals = null;

在threadLocal.set(value)时，其实就是将数据put进对应的Map中

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

ThreadLocal中的get()方法，其实就是调用getMap方法,是获取Thread中的threadlocals

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

}

这里我们基本上可以找到ThreadLocal数据隔离的真相了，每个线程Thread都维护了自己的threadLocals变量，所以在每个线程创建ThreadLocal的时候，实际上数据是存在自己线程Thread的threadLocals变量里面的，别人没办法拿到，从而实现了隔离。

在JDK的实现里面这个 Map 是属于 Thread，而非属于 ThreadLocal。ThreadLocal 仅是一个代理工具类，内部并不持有任何与线程相关的数据。

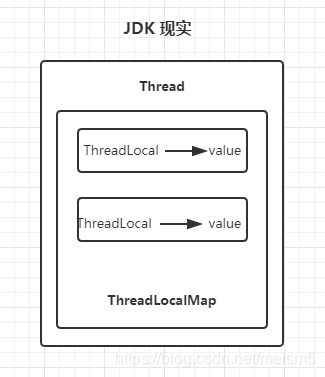
JDK 的实现中 Thread 持有 ThreadLocalMap，而且 ThreadLocalMap 里对 ThreadLocal 的引用还是弱引用（WeakReference），所以只要 Thread 对象可以被回收，那么 ThreadLocalMap 就能被回收。JDK 的这种实现方案复杂但更安全。

在ThreadLocal中有ThreadLocalMap的静态类部类：其中的Map中又有静态类部类Entry，是基于ThreadLocal的弱引用。

static class ThreadLocalMap {

static class Entry extends WeakReference<ThreadLocal<?>> {  
 /\*\* The value associated with this ThreadLocal. \*/  
 Object value;  
  
 Entry(ThreadLocal<?> k, Object v) {  
 super(k);  
 value = v;  
 }  
 }

}



问题：在线程池中使用ThreadLocal为什么会导致内存泄漏？

因为ThreadLocal中对于数据的存储存储还是通过THreadLocalMap中的Entry,Entry是THreadLocal-value的键值存储，而其中的key,是对ThreadLocal的弱引用，当发生GC的时候，该key会被回收，而value是Object类型的，是强引用，只有当对象不在被引用的时候，才会被GC回收，（为何一直会存在引用？？）所以就会导致value不会被回收，从而导致内存泄漏。

在线程池中线程的存活时间很长，往往都是和程序同生共死的，这样Thread持有的ThreadLocalMap一直都不会被回收，再加上ThreadLocalMap的Entry对ThreadLocal是弱引用（WeakReference），所以只要ThreadLocal结束了自己的生命周期后是可以被回收掉的。Entry中的value是被Entry强引用的，即便value的生命周期结束了，value也是无法被回收的，从而导致内存泄漏。

涉及到的问题：

1、弱引用

Java中的四种引用：

强引用、软引用、弱引用、虚引用

1、强引用（StrongReference）

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。

Object o=new Object(); // 强引用

   当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。如果不使用时，要通过如下方式来弱化引用，如下：

o=null; // 帮助垃圾收集器回收此对象

public void test(){

Object o=new Object();

// 省略其他操作

}

在一个方法的内部有一个强引用，这个引用保存在栈中，而真正的引用内容（Object）保存在堆中。当这个方法运行完成后就会退出方法栈，则引用内容的引用不存在，这个Object会被回收。

       但是如果这个o是全局的变量时，就需要在不用这个对象时赋值为null，因为强引用不会被垃圾回收。

强引用在实际中有非常重要的用处，举个ArrayList的实现源代码：

private transient Object[] elementData;

public void clear() {

modCount++;

// Let gc do its work

for (int i = 0; i < size; i++)

elementData[i] = null;

size = 0;

}

在ArrayList类中定义了一个私有的变量elementData数组，在调用方法清空数组时可以看到为每个数组内容赋值为null。不同于elementData=null，（此处只是将相应的引用置为了null，如果这个elementData还指向了别的引用，那么这个强引用依据是存在的）强引用仍然存在，避免在后续调用 add()等方法添加元素时进行重新的内存分配。使用如clear()方法中释放内存的方法对数组中存放的引用类型特别适用，这样就可以及时释放内存。

2、软引用（SoftReference）

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

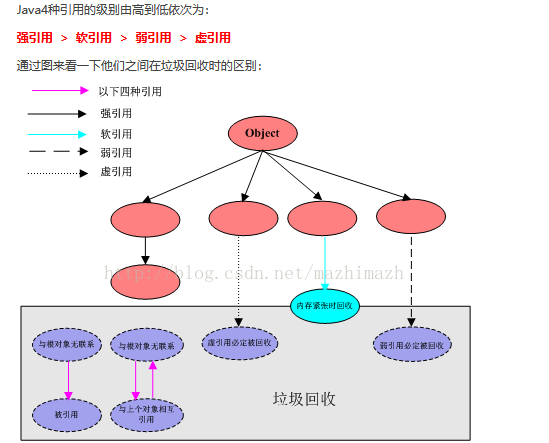
3、弱引用（WeakReference）

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

4、虚引用（PhantomReference）

虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

    虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。



2、避免内存溢出的操作

内存溢出 和 内存泄漏的区别

内存泄漏：

强引用所指向的对象不会被回收，可能导致内存泄漏，虚拟机宁愿抛出OOM(out of memory)也不会去回收他指向的对象。

当使用资源时，为该资源开辟了一段空间，但是使用完后，忘记释放资源了，这是内存还被占着，一两次没有关系，但是内存泄漏次数多了，就会导致可使用内存减少，最终导致内存溢出。

内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你Out of memory。

内存溢出：

直白的说： 系统已经不能再分配出你所需要的空间，比如你需要100M的空间，系统只剩90M了，这就叫内存溢出。

3、ThreadLocal中开放地址法解决hash冲突

关于开放地址法通常需要有三种方法：线性探测、二次探测、再哈希法，在threadLocal中使用的是线性探测法。

ThreadLocal源码解读：

//set（value）进ThreadLocal中时：

threadLocal\_01.set("threadLocal\_01");

//set的源码：

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

1、getMap(t)方法解析：获取当前线程中的ThreadLocal.ThreadLocalMap对象

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

//该变量存在于Thread类中：

ThreadLocal.ThreadLocalMap threadLocals = null;

//所以某线程第一次进行set时，相应的通过该线程获取Map肯定是null的，因为初始值为null，所以第一次set时，肯定是跑的createMap(t,value)

//但是在测试时跑main方法时，在set之前，即main方法运行时，就开始set了，当运行main方法里面的ThreadLocal.Set()时，其实已经存在相应的Map了。

//createMap(t,value):其实就是创建 ThreadLocalMap对象，并将其值赋值给t.threadLocals,所以当再次进行getMap(t)时，就能获取到相应的Map了。

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

//map.set(this, value) this:标识ThreadLocal对象

//最主要的方法之一：

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

Entry[] tab = table;  
 int len = tab.length;

//通过一系列的值获取key对应的散列值，在于16进行&运算，

//&运算：只有当都是1时，才返回1，否则返回0

//得出的i则是对应的tab 数组中的索引值。  
 int i = key.threadLocalHashCode & (len-1);

//for循环，取出数组中i位置的entry对象，判断该对象是否为null，如果不为null则进入循环体，

for (Entry e = tab[i]; e != null; e = tab[i = nextIndex(i, len)]) {

//官方注释：返回此引用引用的对象  
 ThreadLocal<?> k = e.get();

//如果该索引处的引用与当前操作传入的threadlocal引用是同一个，则直接将value替换掉  
 if (k == key) {  
 e.value = value;  
 return;  
 }

//如果该索引出的引用为null，但是Entry又不为null，所以可以判定该Entry对于Threadlocal的弱引用被GC给回收了，所以需要重新布局。  
 if (k == null) {  
 replaceStaleEntry(key, value, i);  
 return;  
 }  
 }

//如果该数组的i位置为null，则直接在该数组的索引位置插入Entry对象：key为threadlocal对象， value为threadlocal.set(value)的value  
 tab[i] = new Entry(key, value);

//size: 表示该table中的entry个数  
 int sz = ++size;

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

// replaceStaleEntry方法详解：【过期数据的交换】

//staleSlot: 是通过key计算出来的索引位置，且该索引位置的entry!=null && entry中存放的弱引用对象ThreadLocal为null， 即该索引出的Entry对象应该是要被清理的对象

private void replaceStaleEntry(ThreadLocal<?> key, Object value,  
 int staleSlot) {  
 Entry[] tab = table;  
 int len = tab.length;  
 Entry e;  
 *//*将该需要被清理的entry的索引值赋值给：slotToExpunge

// slotToExpunge第一次赋值  
int slotToExpunge = staleSlot;

//prevIndex(int I, int len)方法详解：

//return ((i - 1 >= 0) ? i - 1 : len - 1);

//依次取出table中索引i处前面的索引数据，知道对应的索引出的entry等于null时才结束循环。

for (int i = prevIndex(staleSlot, len); (e = tab[i]) != null;  
 i = prevIndex(i, len))

//如果该索引位置的entry 的弱引用threadlocal为null，【还有个前提，该entry!=null, 说明这又是一个无效的数据了，又是被GC清理了】

则将该索引值赋值给slotToExpunge

//此时该slotToExpunge已经是第二次被复制了

//所以当调用replaceStaleEntry传入索引i后，索引i前面有e.get()==null的值时，slotToExpunge记录的是最靠左的那一个。

//待定：记录这个值的作用是啥？  
 if (e.get() == null)  
 slotToExpunge = i;  
 //第二次循环，循环的是索引i右边的数据，直到碰到该索引i右边第一个null时结束for (int i = nextIndex(staleSlot, len);(e = tab[i]) != null;  
 i = nextIndex(i, len)) {  
 ThreadLocal<?> k = e.get();  
  
 //当下一个索引位置不为null，则判断该索引出的entry的key是否与传入的参数key一致，如果相等，则直接替换value，并且需要交换i 和 i+1的entryif (k == key) {  
 e.value = value;  
  
 tab[i] = tab[staleSlot];  
 tab[staleSlot] = e;  
 if (slotToExpunge == staleSlot)  
 slotToExpunge = i;  
 cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);  
 return;  
 }  
  
if (k == null && slotToExpunge == staleSlot)  
 slotToExpunge = i;  
 }  
  
 //如果当前失效的索引的下一个索引位置为null，则不再需要进行交换，可以直接将该失效索引重新赋值一个entry  
tab[staleSlot].value = null;  
 tab[staleSlot] = new Entry(key, value);  
  
if (slotToExpunge != staleSlot)  
 cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);  
}

ThreadLocalMap 和HashMap的功能类似，但是实现上却有很大的不同：

1. HashMap 的数据结构是数组+链表
2. ThreadLocalMap的数据结构仅仅是数组
3. HashMap 是通过链地址法解决hash 冲突的问题
4. ThreadLocalMap 是通过开放地址法来解决hash 冲突的问题
5. HashMap 里面的Entry 内部类的引用都是强引用
6. ThreadLocalMap里面的Entry 内部类中的key 是弱引用，value 是强引用

**问题：为什么ThreadLocalMap 采用开放地址法来解决哈希冲突?**

jdk 中大多数的类都是采用了链地址法来解决hash 冲突，为什么ThreadLocalMap 采用开放地址法来解决哈希冲突呢？

链地址法解决hash冲突，是通过数组+链表+红黑树的方式

开放地址发解决hash冲突，是通过数组+下一个空的散列地址，如果改地址存在hash冲突，则自动查找下一个位置进行存放。

所以相应的在数据清理时，一定要有换位的操作，因为相同与不同的hash值都是存储在该数组中，所以可能相同的hash值在不同的位置，且可能还是不连续的，例如hash相同，但是值存储的索引位置为：1， 5， 9.此时如果1号位的数据被清理了，则二号位肯定要前移。且二号位一定有值，以内如果二号位没有至，5号位不可能有值。如果二号位不前移，则下次再存储5号位的数据时，则能正常保存进去，则1、5两个位置的数据是一模一样的，这是不允许的。

链地址法和开放地址法的优缺点

开放地址法：（table大小的2/3，当size >= threshold时，遍历table并删除key为null的元素，如果删除后size >= threshold\*3/4时，需要对table进行扩容。）

1. 容易产生堆积问题，不适于大规模的数据存储。
2. 散列函数的设计对冲突会有很大的影响，插入时可能会出现多次冲突的现象。
3. 删除的元素是多个冲突元素中的一个，需要对后面的元素作处理，实现较复杂。

链地址法：

1. 处理冲突简单，且无堆积现象，平均查找长度短。
2. 链表中的结点是动态申请的，适合构造表不能确定长度的情况。
3. 删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。
4. 指针需要额外的空间，故当结点规模较小时，开放定址法较为节省空间。

ThreadLocalMap 采用开放地址法原因

1. ThreadLocal 中看到一个属性 HASH\_INCREMENT = 0x61c88647 ，0x61c88647 是一个神奇的数字，让哈希码能均匀的分布在2的N次方的数组里, 即 Entry[] table，关于这个神奇的数字google 有很多解析，这里就不重复说了
2. ThreadLocal 往往存放的数据量不会特别大（而且key 是弱引用又会被垃圾回收，及时让数据量更小），这个时候开放地址法简单的结构会显得更省空间，同时数组的查询效率也是非常高，加上第一点的保障，冲突概率也低

魔数0x61c88647 （魔法值）

* 生成hash code间隙为这个魔数，可以让生成出来的值或者说ThreadLocal的ID较为均匀地分布在2的幂大小的数组中。
* 可以看出，它是在上一个被构造出的ThreadLocal的ID/threadLocalHashCode的基础上加上一个魔数0x61c88647的。
* 这个魔数的选取与斐波那契散列有关，0x61c88647对应的十进制为1640531527。
* 斐波那契散列的乘数可以用(long) ((1L << 31) \* (Math.sqrt(5) - 1))可以得到2654435769，如果把这个值给转为带符号的int，则会得到-1640531527。换句话说 (1L << 32) - (long) ((1L << 31) \* (Math.sqrt(5) - 1))得到的结果就是1640531527也就是0x61c88647 。
* 通过理论与实践，当我们用0x61c88647作为魔数累加为每个ThreadLocal分配各自的ID也就是threadLocalHashCode再与2的幂取模，得到的结果分布很均匀。
* ThreadLocalMap使用的是线性探测法，均匀分布的好处在于很快就能探测到下一个临近的可用slot，从而保证效率。。为了优化效率。