2、导致 JVM 内存泄露的 ThreadLocal 详解

为什么要有 ThreadLocal

我们首先来看看一段最纯粹的原生 JDBC 代码

```
/*获得数据库连接*/
2 usages
private static Connection getConn() {
    String driver = "com.mysql.jdbc.Driver";
    String url = "jdbc:mysql://localhost:3306/samp_db";
    String username = "root";
    String password = "";
    Connection conn = null;
    try {
        Class.forName(driver); //classLoader,加载对应驱动
        conn = (Connection) DriverManager.getConnection(url,
```

```
/*Insert数据, Update,Delete用法与之类似*/
private static int insert(String student) {
    Connection conn = getConn();
    int i = 0;
    String sql = "insert into students (Name,Sex,Age) values(?,?,?)";
    PreparedStatement pstmt;
    try {
        pstmt = (PreparedStatement) conn.prepareStatement(sql);
        pstmt.setString( parameterIndex: 1, student);
        i = pstmt.executeUpdate();
```

```
/*查询数据*/
private static Integer getAll() {
    Connection conn = getConn();
    String sql = "select * from students";
    PreparedStatement pstmt;
    try {
        pstmt = (PreparedStatement)conn.prepareStatement(sql);
        ResultSet rs = pstmt.executeQuery();
        while (rs.next()) {
```

可以看到，在使用 JDBC 时，我们首先要配置后再拿到 JDBC 连接，然后在增删改查的业务方法中拿到这个连接，并把我们的 SQL 语句交给 JDBC 连接发送到真实的 DB 上执行。

在实际的工作中，我们不会每次执行 SQL 语句时临时去建立连接，而是会借助数据库连接池，同时因为实际业务的复杂性，为了保证数据的一致性，我们还会引入事务操作，于是上面的代码就会变成：

```java
/*数据库连接池*/
3 usages
private static DBPool dbPool = new DBPool( initialSize: 10);

/*使用事务*/
public void business(){
    /*获取连接*/
    Connection conn = null;
    try {
        conn = dbPool.fetchConnection( mills: 1000);
        conn.setAutoCommit(false);/*开启事务*/

        insert( student: "13号");
        System.out.println("其他业务工作");
        insert( student: "14号");

        conn.commit();/*提交*/
    } catch (Exception e) {
        try {
```

```java
private static int insert(String student) {
    int i = 0;
    try {
        Connection conn = dbPool.fetchConnection( mills: 1000);
        String sql = "insert into students (Name,Sex,Age) values(?,?,?)";
        PreparedStatement pstmt;
```

```java
/*查询数据*/
private static Integer getAll() {
    try {
        Connection conn = dbPool.fetchConnection( mills: 1000);
        String sql = "select * from students";
        PreparedStatement pstmt;
        pstmt = (PreparedStatement)conn.prepareStatement(sql);
```

但是上面的代码包含什么样的问题呢？分析代码我们可以发现，执行业务方法 business 时，为了启用事务，我们从数据库连接池中拿了一个连接，但是在具体的 insert 方法和 getAll 方法中，在执行具体的 SQL 语句时，我们从数据库连接池中拿一个连接，这就说执行事务和执行 SQL 语句完全是不同的数据库连接，这会导致什么问题？事务失效了！！数据库执行事务时，事务的开启和提交、语句的执行等都是必须在一个连接中的。实际上，上面的代码要保证数据的一致性，就必须要启用分布式事务。

怎么解决这个问题呢？有一个解决思路是，把数据库连接作为方法的参数，在方法之间进行传递，比如下面这样：

```java
/*Insert数据，Update,Delete用法与之类似*/
2 usages
private static int insert(String student,Connection conn) {
    int i = 0;
    try {
```

```java
/*查询数据*/
private static Integer getAll(Connection conn) {
    try {
        String sql = "select * from students";
        PreparedStatement pstmt;
```

```java
/*使用事务*/
public void business(){
    /*获取连接*/
    Connection conn = null;
    try {
        conn = dbPool.fetchConnection( mills: 1000);
        conn.setAutoCommit(false);/*开启事务*/

        insert( student: "13号", conn);
        System.out.println("其他业务工作");
        insert( student: "14号", conn);
```

但是我们分析平时我们使用 SSM 的代码会发现，我们在编写数据访问相关代码的时候从来没有把数据库连接作为方法参数进行传递。这意味着，对 Spring 来说，在帮我们进行事务托管的时候，会遇到同样的问题，那么 Spring 是如何解决这个问题的？

其实稍微分析下 Spring 的事务管理器的代码就能发现端倪，在 org.springframework.jdbc.datasource.DataSourceTransactionManager#doBegin 中，我们会看到如下代码

```java
// Bind the connection holder to the thread.
if (txObject.isNewConnectionHolder()) {
    TransactionSynchronizationManager.bindResource(getDataSource(),
}
```

上面的注释已经很清楚了说明“绑定连接到这个线程”，如何绑定的？继续深入看看

```java
public static void bindResource(Object key, Object value) th
    Object actualKey = TransactionSynchronizationUtils.unwr
    Assert.notNull(value, message: "Value must not be null
    Map<Object, Object> map = resources.get();
    // set ThreadLocal Map if none found
    if (map == null) {
        map = new HashMap<Object, Object>();
        resources.set(map);
    }
```

```java
private static final ThreadLocal<Map<Object, Object>> resources =
        new NamedThreadLocal<Map<Object, Object>>("Transactional res
```

```java
public class NamedThreadLocal<T> extends ThreadLocal<T> {
```

看来，Spring 是使用一个 ThreadLocal 来实现“绑定连接到线程”的。

现在我们可以对 ThreadLocal 下一个比较确切的定义了

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

此类提供线程局部变量。这些变量与普通对应变量的不同之处在于，访问一个变量的每个线程（通过其 get 或 set 方法）都有自己独立初始化的变量副本。ThreadLocal 实例通常是希望将状态与线程（例如，用户 ID 或事务 ID）相关联的类中的私有静态字段。

也就是说 ThreadLocal 为每个线程都提供了变量的副本，使得每个线程在某一时间访问到的并非同一个对象，这样就隔离了多个线程对数据的数据共享。

由此也可以看出 ThreadLocal 和 Synchonized 都用于解决多线程并发访问。可是 ThreadLocal 与 synchronized 有本质的差别。synchronized 是利用锁的机制，使变量或代码块在某一时该仅仅能被一个线程访问，ThreadLocal 则是副本机制。此时不论多少线程并发访问都是线程安全的。

ThreadLocal 的一大应用场景就是跨方法进行参数传递，比如 Web 容器中，每个完整的请求周期会由一个线程来处理。结合 ThreadLocal 再使用 Spring 里的 IOC 和 AOP，就可以很好的解决我们上面的事务的问题。只要将一个数据库连接放入 ThreadLocal 中，当前线程执行时只要有使用数据库连接的地方就从 ThreadLocal 获得就行了。

再比如，在微服务领域，链路跟踪中的 traceId 传递也是利用了 ThreadLocal。

ThreadLocal 的使用

ThreadLocal 类接口很简单，只有 4 个方法，我们先来了解一下：

- void set(Object value)

设置当前线程的线程局部变量的值。

- public Object get()

该方法返回当前线程所对应的线程局部变量。

- public void remove()

将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是 JDK 5.0 新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。

- protected Object initialValue()

返回该线程局部变量的初始值，该方法是一个 protected 的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第 1 次调用 get() 或 set(Object)时才执行，并且仅执行 1 次。ThreadLocal 中的缺省实现直接返回一个 null。

实现解析

实现分析

怎么实现 ThreadLocal，既然说让每个线程都拥有自己变量的副本，最容易的方式就是用一个 Map 将线程的副本存放起来，Map 里 key 就是每个线程的唯一性标识，比如线程 ID，value 就是副本值，实现起来也很简单：

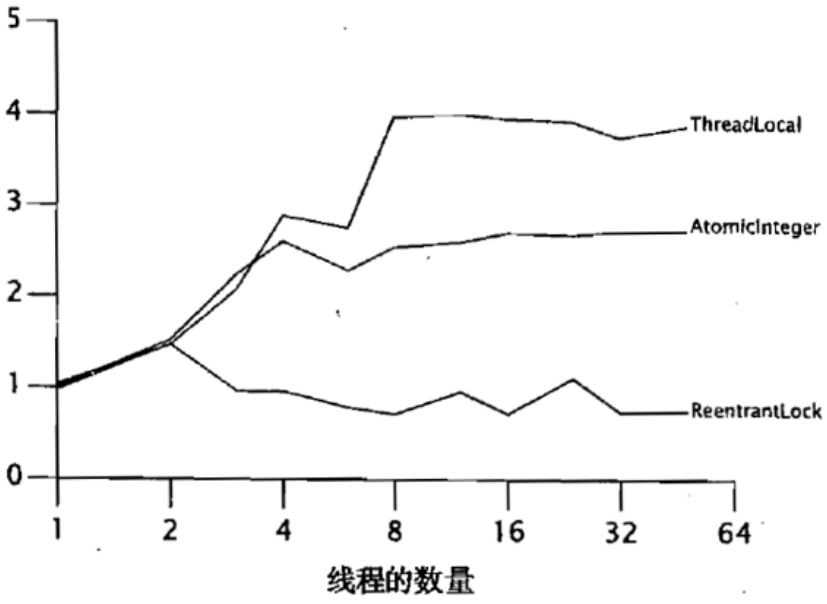
```
public class MyThreadLocal<T> {
    /*存放变量副本的map容器，以Thread为键，变量副本为value*/
    2 usages
    private Map<Thread,T> threadTMap = new HashMap<>();

    public synchronized T get(){
        return  threadTMap.get(Thread.currentThread());
    }

    public synchronized void set(T t){
        threadTMap.put(Thread.currentThread(),t);
    }

}
```

考虑到并发安全性，对数据的存取用 synchronize 关键字加锁，但是 DougLee 在《并发编程实战》中为我们做过性能测试

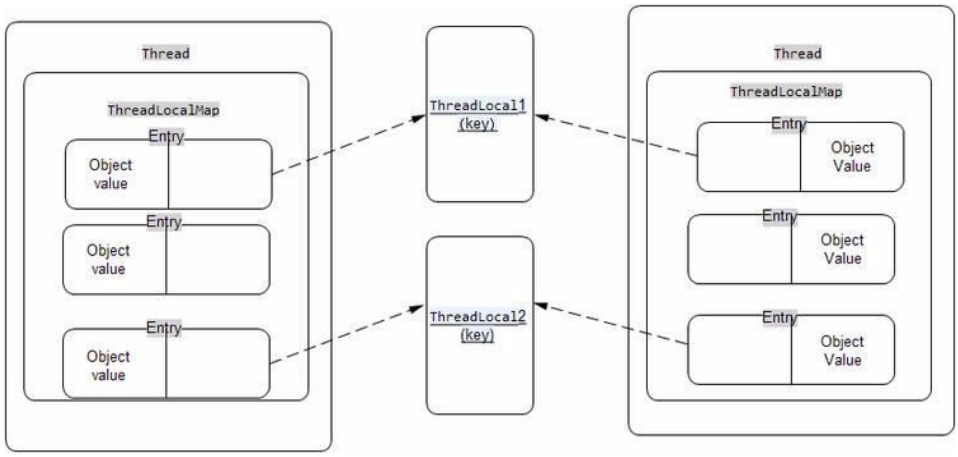


可以看到 ThreadLocal 的性能远超类似 synchronize 的锁实现 ReentrantLock，比我们后面要学的 AtomicInteger 也要快很多，即使我们把 Map 的实现更换为 Java 中专为并发设计的 ConcurrentHashMap 也不太可能达到这么高的性能。

怎么样设计可以让 ThreadLocal 达到这么高的性能呢？最好的办法则是让变量副本跟随着线程本身，而不是将变量副本放在一个地方保存，这样就可以在存取时避开线程之间的竞争。

同时，因为每个线程所拥有的变量的副本数是不定的，有些线程可能有一个，有些线程可能有 2 个甚至更多，则线程内部存放变量副本需要一个容器，而且容器要支持快速存取，所以在每个线程内部都可以持有一个 Map 来支持多个变量副本，这个 Map 被称为 ThreadLocalMap。

具体实现




```
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();
}

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

java × OnlyMain.java × ThreadPoolExecutor.java × ThreadLocal.java × Thread.java ×

```
ThreadLocal.ThreadLocalMap threadLocals = null;
```

上面先取到当前线程，然后调用 getMap 方法获取对应的 ThreadLocalMap，ThreadLocalMap 是一个声明在 ThreadLocal 的静态内部类，然后 Thread 类中有一个这样类型成员变量，也就是 ThreadLocalMap 实例化是在 Thread 内部，所以 getMap 是直接返回 Thread 的这个成员。

看下 ThreadLocal 的内部类 ThreadLocalMap 源码，这里其实是个标准的 Map 实现，内部有一个元素类型为 Entry 的数组，用以存放线程可能需要的多个副本变量。

```
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;
```

类似于map的key,value结构 key就是ThreadLocal，Value就需要隔离访问的变量

用数组保存了Entry，因为可能有多个变量需要线程隔离访问

可以看到有个 Entry 内部静态类，它继承了 WeakReference，总之它记录了两个信息，一个是 ThreadLocal<?>类型，一个是 Object 类型的值。getEntry 方法则是获取某个 ThreadLocal 对应的值，set 方法就是更新或赋值相应的 ThreadLocal 对应的值。

```
private Entry getEntry(ThreadLocal<?> key) {
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];
```

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

    // We don't use a fast path as with get() because
```

回顾我们的 get 方法，其实就是拿到**每个线程独有的 ThreadLocalMap**

然后再用 ThreadLocal 的当前实例，拿到 Map 中的相应的 Entry，然后就可以拿到相应的值返回出去。当然，如果 Map 为空，还会先进行 map 的创建，初始化等工作。

Hash 冲突的解决

什么是 Hash，就是把任意长度的输入（又叫做预映射， pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值，输入的微小变化会导致输出的巨大变化。所以 Hash 常用在消息摘要或签名上，常用 hash 消息摘要算法有：（1)MD4（2) MD5 它对输入仍以 512 位分组，其输出是 4 个 32 位字的级联（3)SHA-1 及其他。

Hash 转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。比如有 10000 个数放到 100 个桶里，不管怎么放，有个桶里数字个数一定是大于 2 的。

所以 Hash 简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。常用 HASH 函数：直接取余法、乘法取整法、平方取中法。 Java 里的 HashMap 用的就是直接取余法。

我们已经知道 Hash 属于压缩映射，一定能会产生多个实际值映射为一个 Hash 值的情况，这就产生了冲突，常见处理 Hash 冲突方法：

开放定址法：

基本思想是，出现冲突后按照一定算法查找一个空位置存放，根据算法的不同又可以分为线性探测再散列、二次探测再散列、伪随机探测再散列。

线性探测再散列即依次向后查找，二次探测再散列，即依次向前后查找，增量为 1、2、3 的二次方，伪随机，顾名思义就是随机产生一个增量位移。

ThreadLocal 里用的则是线性探测再散列

```
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;
}
```

```
private static int nextIndex(int i, int len) {
    return ((i + 1 < len) ? i + 1 : 0);
}
```

链地址法：

这种方法的基本思想是将所有哈希地址为 i 的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第 i 个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。Java

里的 HashMap 用的就是链地址法，为了避免 hash 洪水攻击，1.8 版本开始还引入了红黑树。

再哈希法：

这种方法是同时构造多个不同的哈希函数：Hi=RH1（key） i=1，2，…，k 当哈希地址 Hi=RH1（key）发生冲突时，再计算 Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

引发的内存泄漏分析

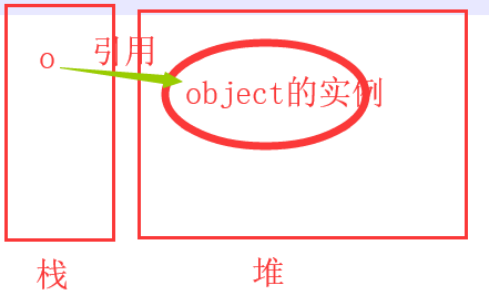
复习

引用

Object o = new Object();

这个 o，我们可以称之为对象引用，而 new Object()我们可以称之为在内存中产生了一个对象实例。

```
Object o = new Object();
```



当写下 **o=null** 时，只是表示 o 不再指向堆中 object 的对象实例，不代表这个对象实例不存在了。

强引用就是指在程序代码之中普遍存在的，类似“Object obj=new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象实例。

软引用是用来描述一些还有用但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象实例列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 SoftReference 类来实现软引用。

弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象实例只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，

无论当前内存是否足够，都会回收掉只被弱引用关联的对象实例。在 JDK 1.2 之后，提供了 WeakReference 类来实现弱引用。

虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象实例是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象实例被收集器回收时收到一个系统通知。在 JDK 1.2 之后，提供了 PhantomReference 类来实现虚引用。

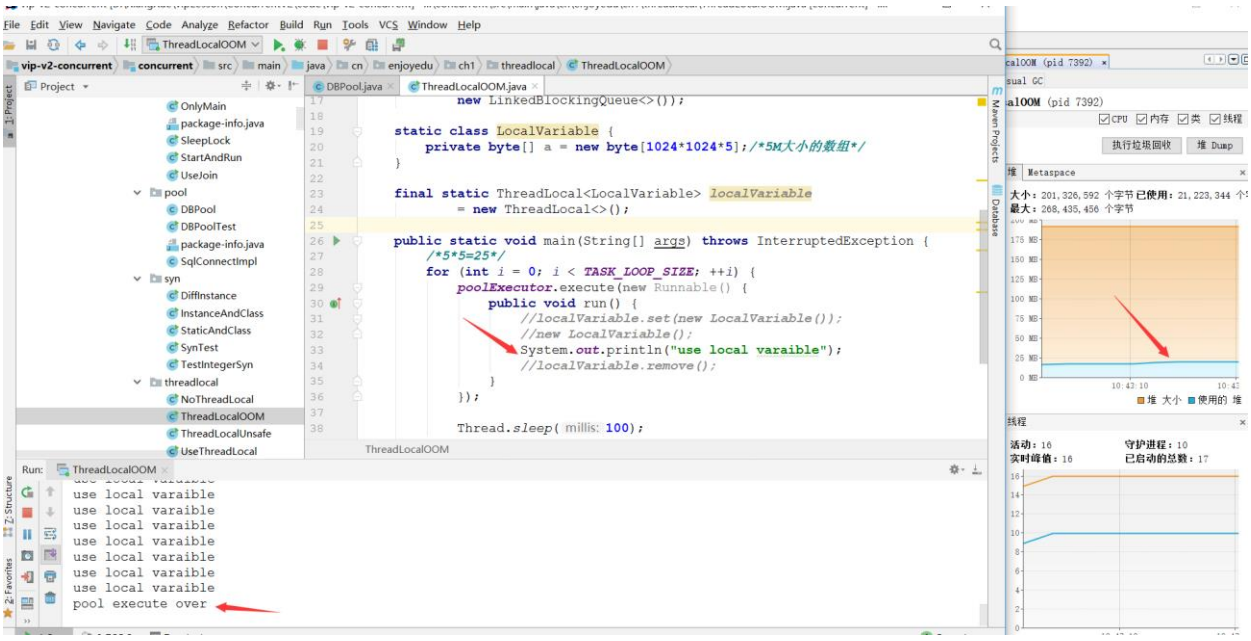
内存泄漏的现象

执行 cn.tulingxueyuan.tl.threadlocal 下的 ThreadLocalMemoryLeak，并将堆内存大小设置为-Xmx256m，

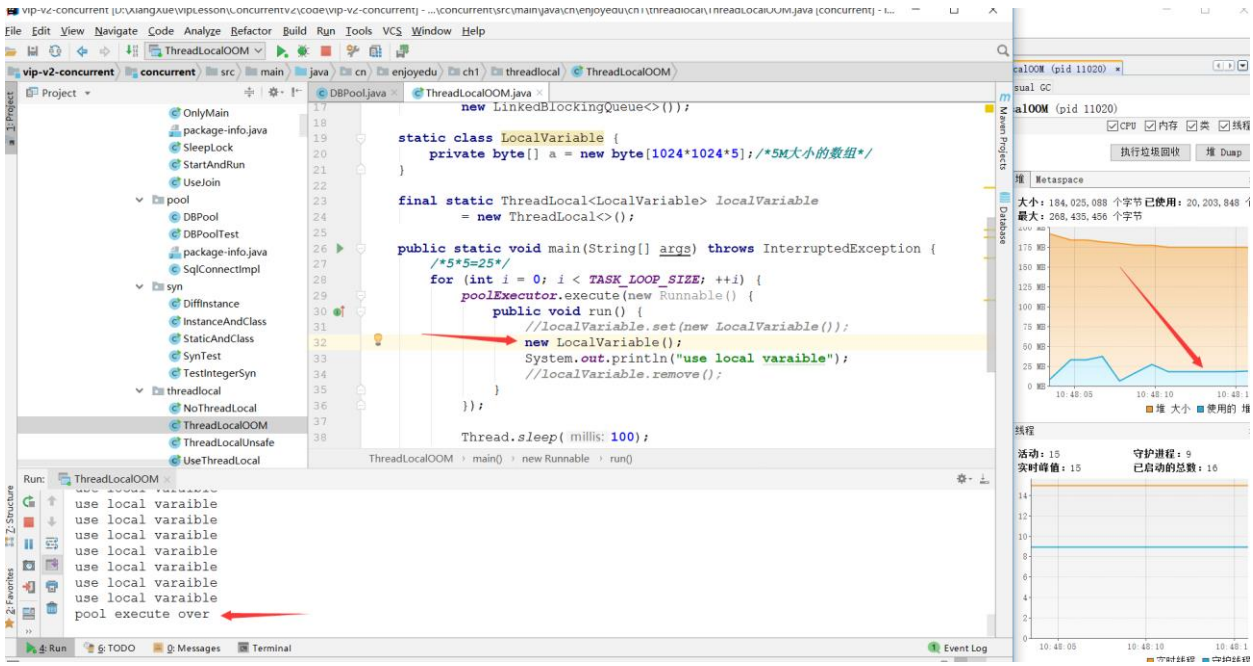
我们启用一个线程池，大小固定为 5 个线程

```
final static ThreadPoolExecutor poolExecutor
        = new ThreadPoolExecutor( corePoolSize: 5,  maximumPoolSize: 5,  keepAliveTime: 1,
        TimeUnit.MINUTES,
        new LinkedBlockingQueue<>());
```

场景 1，首先任务中不执行任何有意义的代码，当所有的任务提交执行完成后，可以看见，我们这个应用的内存占用基本上为 25M 左右



场景 2，然后我们只简单的在每个任务中 new 出一个数组，执行完成后我们可以看见，内存占用基本和场景 1 同

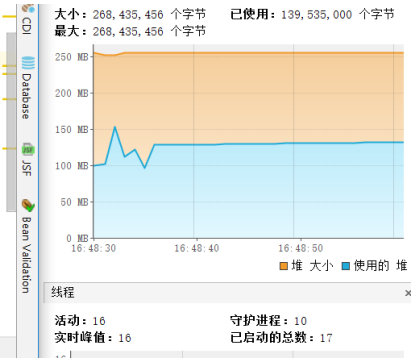


场景 3，当我们启用了 ThreadLocal 以后：

```
    private byte[] a = new byte[1024*1024*5];/*5M大小的数组*/
}

ThreadLocal<LocalVariable> localVariable;
        //= new ThreadLocal<>();

public static void main(String[] args) throws InterruptedException {
    /*5*5=25*/
    for (int i = 0; i < TASK_LOOP_SIZE; ++i) {
        poolExecutor.execute(new Runnable() {
            public void run() {
                ThreadLocalOOM oom = new ThreadLocalOOM();
                oom.localVariable = new ThreadLocal<>();
                oom.localVariable.set(new LocalVariable());
                //new LocalVariable();
                System.out.println("use local varaible");
                //oom.localVariable.remove();
```

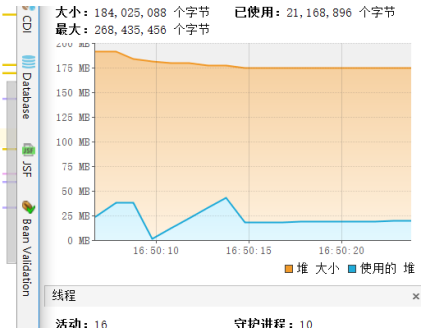


执行完成后我们可以看见，内存占用变为了 100 多 M

场景 4，于是，我们加入一行代码，再执行，看看内存情况:

```
public static void main(String[] args) throws InterruptedException {
    /*5*5=25*/
    for (int i = 0; i < TASK_LOOP_SIZE; ++i) {
        poolExecutor.execute(new Runnable() {
            public void run() {
                ThreadLocalOOM oom = new ThreadLocalOOM();
                oom.localVariable = new ThreadLocal<>();
                oom.localVariable.set(new LocalVariable());
                //new LocalVariable();
                System.out.println("use local varaible");
                oom.localVariable.remove();
            }
        });

        Thread.sleep( millis: 100);
    }
```



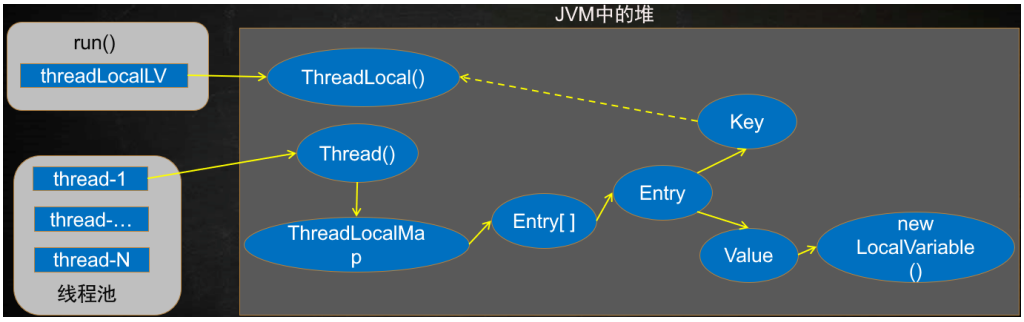
可以看见，内存占用基本和场景 1 同。

这就充分说明，场景 3，当我们启用了 ThreadLocal 以后确实发生了内存泄漏。

分析

根据我们前面对 ThreadLocal 的分析，我们可以知道每个 Thread 维护一个 ThreadLocalMap，这个映射表的 key 是 ThreadLocal 实例本身，value 是真正需要存储的 Object，也就是说 ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value。仔细观察 ThreadLocalMap，这个 map 是使用 ThreadLocal 的弱引用作为 Key 的，弱引用的对象在 GC 时会被回收。

因此使用了 ThreadLocal 后，引用链如图所示



图中的虚线表示弱引用。

这样，当把 threadlocal 变量置为 null 以后，没有任何强引用指向 threadlocal 实例，所以 threadlocal 将会被 gc 回收。这样一来，ThreadLocalMap 中就会出现 key 为 null 的 Entry，就没有办法访问这些 key 为 null 的 Entry 的 value，如果当前线程再迟迟不结束的话，这些 key 为 null 的 Entry 的 value 就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value，而这块 value 永远不会被访问到了，所以存在着内存泄露。

只有当前 thread 结束以后，current thread 就不会存在栈中，强引用断开，Current Thread、Map value 将全部被 GC 回收。最好的做法是不在需要使用 ThreadLocal 变量后，都调用它的 remove()方法，清除数据。

所以回到我们前面的实验场景，场景 3 中，虽然线程池里面的任务执行完毕了，但是线程池里面的 5 个线程会一直存在直到 JVM 退出，我们 set 了线程的 localVariable 变量后没有调用 localVariable.remove()方法，导致线程池里面的 5 个线程的 threadLocals 变量里面的 new LocalVariable()实例没有被释放。

其实考察 ThreadLocal 的实现，我们可以看见，无论是 get()、set()在某些时候，调用了 expungeStaleEntry 方法用来清除 Entry 中 Key 为 null 的 Value，但是这是不及时的，也不是每次都会执行的，所以一些情况下还是会发生内存泄露。只有 remove()方法中显式调用了 expungeStaleEntry 方法。

从表面上看内存泄漏的根源在于使用了弱引用，但是另一个问题也同样值得思考：为什么使用弱引用而不是强引用？

下面我们分两种情况讨论：

key 使用强引用：对 ThreadLocal 对象实例的引用被置为 null 了，但是 ThreadLocalMap 还持有这个 ThreadLocal 对象实例的强引用，如果没有手动删除，ThreadLocal 的对象实例不会被回收，导致 Entry 内存泄漏。

key 使用弱引用：对 ThreadLocal 对象实例的引用被被置为 null 了，由于 ThreadLocalMap 持有 ThreadLocal 的弱引用，即使没有手动删除，ThreadLocal 的对象实例也会被回收。value 在下一次 ThreadLocalMap 调用 set，get，remove 都有机会被回收。

比较两种情况，我们可以发现：由于 ThreadLocalMap 的生命周期跟 Thread 一样长，如果都没有手动删除对应 key，都会导致内存泄漏，但是使用弱引用可以多一层保障。

因此，ThreadLocal 内存泄漏的根源是：由于 ThreadLocalMap 的生命周期跟 Thread 一样长，如果没有手动删除对应 key 就会导致内存泄漏，而不是因为弱引用。

总结

JVM 利用设置 ThreadLocalMap 的 Key 为弱引用，来避免内存泄露。

JVM 利用调用 remove、get、set 方法的时候，回收弱引用。

当 ThreadLocal 存储很多 Key 为 null 的 Entry 的时候，而不再去调用 remove、get、set 方法，那么将导致内存泄漏。

使用**线程池+** ThreadLocal 时要小心，因为这种情况下，线程是一直在不断的重复运行的，从而也就造成了 value 可能造成累积的情况。

错误使用 ThreadLocal 导致线程不安全

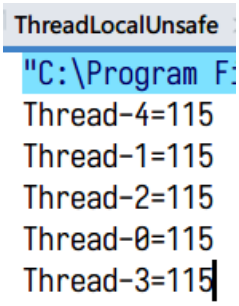
参见代码：

```
public static Number number = new Number(0);
2 usages
public static ThreadLocal<Number> value = new ThreadLocal<Number>()/*{
    @Override
    protected Number initialValue() {
        return new Number(0);
    }
}*/;

public void run() {
    Random r = new Random();
    //Number number = value.get();
    //每个线程计数加随机数
    number.setNum(number.getNum()+r.nextInt( bound: 100));
  //将其存储到ThreadLocal中
    value.set(number);
    SleepTools.ms( seconds: 2);
    //输出num值
    System.out.println(Thread.currentThread().getName()+"="+value.get().getNum());
}

public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
        new Thread(new ThreadLocalUnsafe()).start();
    }
}
```

运行后的结果为



为什么每个线程都输出 115？难道他们没有独自保存自己的 Number 副本吗？为什么其他线程还是能够修改这个值？仔细考察 ThreadLocal 和 Thead 的代码，我们发现 ThreadLocalMap 中保存的其实是对象的一个引用，这样的话，当有其他线程对这个引用指向的对象实例做修改时，其实也同时影响了所有的线程持有的对象引用所指向的同一个对象实例。这也就是为什么上面的程序为什么会输出一样的结果。

而上面的程序要正常的工作，应该的用法是让每个线程中的 ThreadLocal 都应该持有一个新的 Number 对象。

```
//    public static Number number = new Number(0);
    3 usages
    public static ThreadLocal<Number> value = new ThreadLocal<Number>(){
        @Override
        protected Number initialValue() {
            return new Number(0);
        }
    };

    public void run() {
        Random r = new Random();
        Number number = value.get();
        //每个线程计数加随机数
        number.setNum(number.getNum()+r.nextInt( bound: 100));
      //将其存储到ThreadLocal中
        value.set(number);
        SleepTools.ms( seconds: 2);
        //输出num值
        System.out.println(Thread.currentThread().getName()+"="+value.get().getNum());
    }

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            new Thread(new ThreadLocalUnsafe()).start();
        }
    }
```

分享地址

http://note.youdao.com/noteshare?id=8cfb674095580f33bf80a822d33b5361&sub=C6D5B6A8BC4A4F31BCFB94740F8146B7