一个 ThreadLocal 和面试官大战 30 个回合

小林coding *5月11日*

开场

杭州某商务楼里，正发生着一起求职者和面试官的battle。

面试官：你先自我介绍一下。

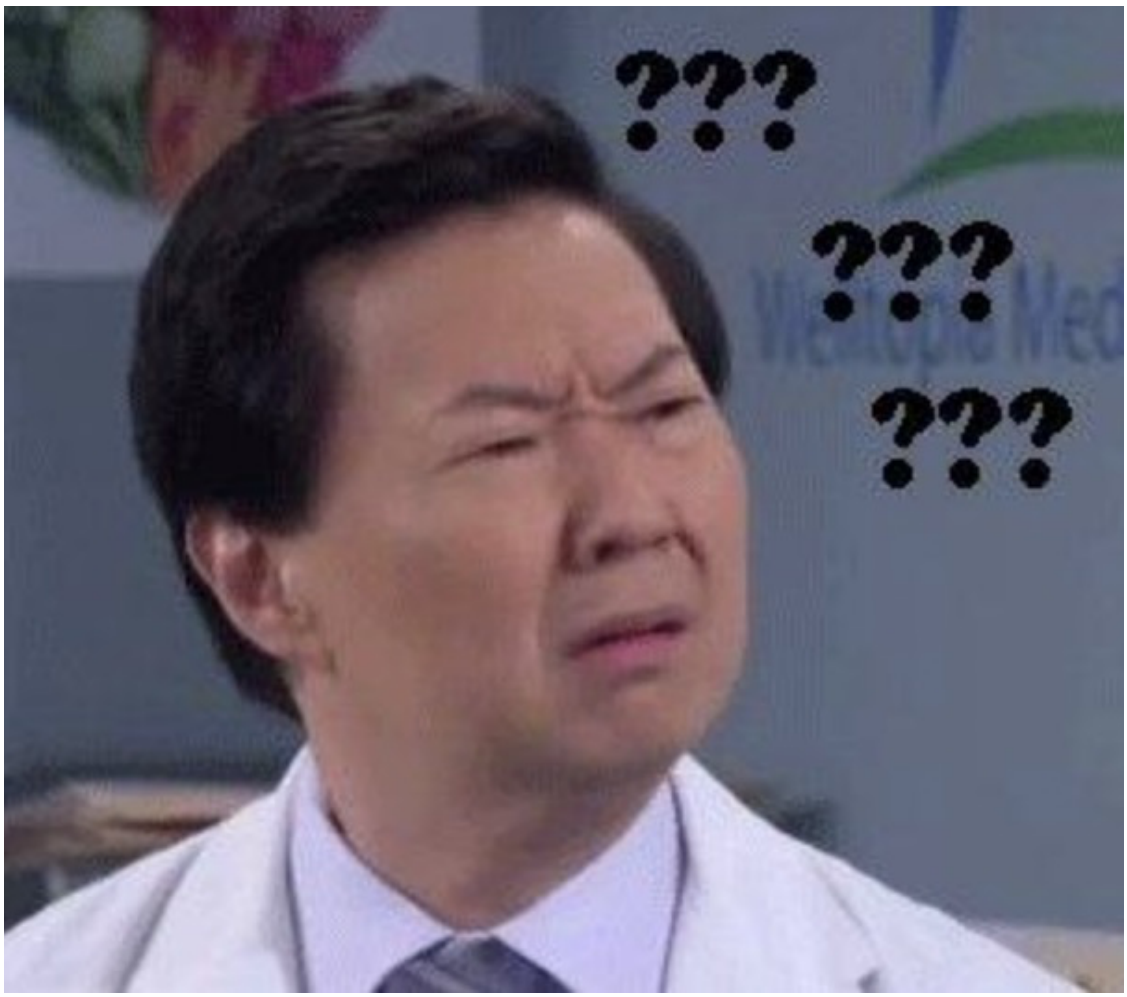
安琪拉：面试官你好，我是草丛三婊，最强中单（妲己不服），草地摩托车车手，第21套广播体操推广者，火的传人安琪拉，这是我的简历，请过目。

面试官：看你简历上写熟悉多线程编程，熟悉到什么程度？

安琪拉：精通。

对。。。，你没看错，问就是“精通”，把666打在评论区。

面试官：



[心想] 莫不是个憨批，上来就说自己精通，谁把精通挂嘴上，莫不是个愣头青嘞！

面试官：那我们开始吧。用过Threadlocal 吧？

安琪拉：用过。

面试官：那你跟我讲讲 ThreadLocal 在你们项目中的用法吧。

安琪拉：我们项目属于保密项目，无可奉告，你还是换个问题吧！

面试官：那说个不保密的项目，或者你直接告诉我Threadlocal 的实现原理吧。

正题

安琪拉：show time。。。

安琪拉：举个栗子，我们支付宝每秒钟同时会有很多用户请求，那每个请求都带有用户信息，我们知道通常都是一个线程处理一个用户请求，我们可以把用户信息丢到Threadlocal里面，让每个线程处理自己的用户信息，线程之间互不干扰。

面试官：等等，问你个私人问题，为什么从支付宝跑出来面试，受不了PUA了吗？

安琪拉：PUA我，不存在的，能PUA我的人还没出生呢！公司食堂吃腻了，想换换口味。

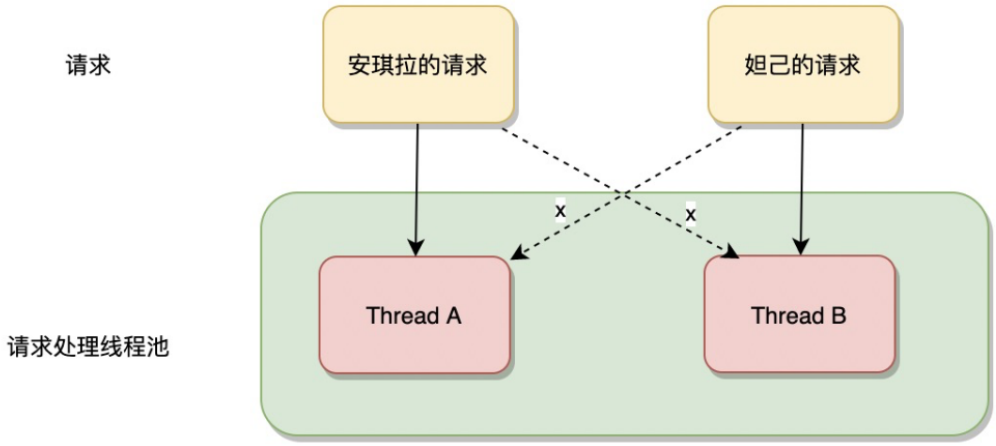
img

面试官：那你来给我讲讲`Threadlocal`是干什么的?

安琪拉：Threadlocal 主要用来做**线程变量的隔离**，这么说可能不是很直观。

还是说前面提到的例子，我们程序在处理用户请求的时候，通常后端服务器是有一个线程池，来一个请求就交给一个线程来处理，那为了防止多线程并发处理请求的时候发生串数据，比如AB线程分别处理安琪拉和妲己的请求，A线程本来处理安琪拉的请求，结果访问到妲己的数据上了，把妲己支付宝的钱转走了。

所以就可以把安琪拉的数据跟A线程绑定，线程处理完之后解除绑定。



面试官：那把你刚才说的场景用伪代码实现一下，来笔给你!

安琪拉：ok

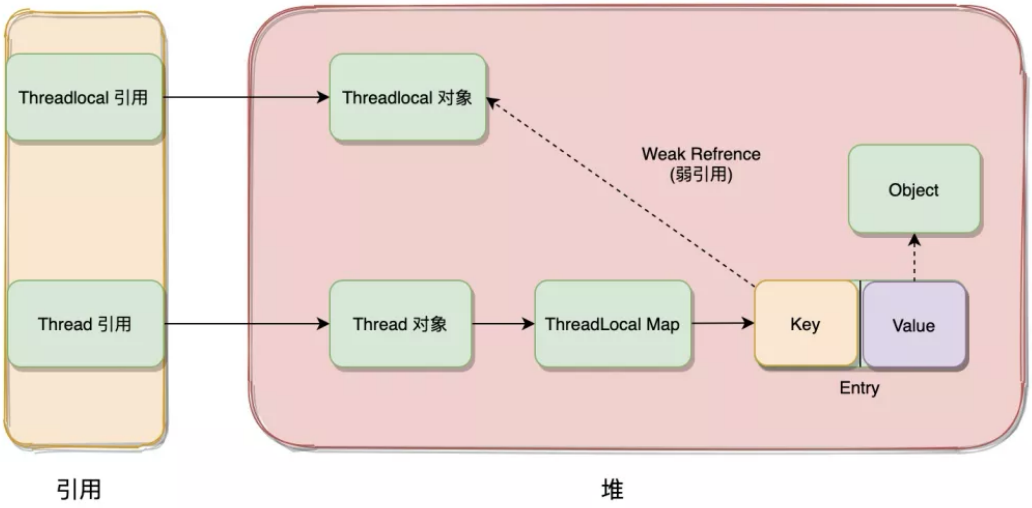
```
//存放用户信息的ThreadLocal
private static final ThreadLocal<UserInfo> userInfoThreadLocal = new
ThreadLocal<>();

public Response handleRequest(UserInfo userInfo) {
  Response response = new Response();
  try {
    // 1.用户信息set到线程局部变量中
    userInfoThreadLocal.set(userInfo);
    doHandle();
  } finally {
    // 3.使用完移除掉
    userInfoThreadLocal.remove();
```

```
  }

  return response;
}

//业务逻辑处理
private void doHandle () {
  // 2.实际用的时候取出来
  UserInfo userInfo = userInfoThreadLocal.get();
  //查询用户资产
  queryUserAsset(userInfo);
}
```

1.2.3 步骤很清楚了。

面试官：那你跟我说说`Threadlocal`怎么实现线程变量的隔离的?

安琪拉：Oh, 这么快进入正题，我先给你画个图，如下



面试官：图我看了，那你对着前面你写的代码讲一下对应图中流程。

安琪拉：没问题

- 首先我们通过`ThreadLocal<UserInfo> userInfoThreadLocal = new ThreadLocal()`初始化了一个Threadlocal 对象，就是上图中说的Threadlocal 引用，这个引用指向堆中的ThreadLocal 对象；
- 然后我们调用`userInfoThreadLocal.set(userInfo);`这里做了什么事呢?

  我们把源代码拿出来，看一看就清晰了。

  我们知道 Thread 类有个 ThreadLocalMap 成员变量，这个Map key是Threadlocal 对象，value是你要存放的线程局部变量。

  ```
  # Threadlocal类 Threadlocal.class
  public void set(T value) {
    //获取当前线程Thread，就是上图画的Thread 引用
    Thread t = Thread.currentThread();
    //Thread类有个成员变量ThreadlocalMap，拿到这个Map
    ThreadLocalMap map = getMap(t);
    if (map != null)
      //this指的就是Threadlocal对象
      map.set(this, value);
    else
      createMap(t, value);
  ```

```
}

ThreadLocalMap getMap(Thread t) {
  //获取线程的ThreadLocalMap
  return t.threadLocals;
}

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

```
# Thread类 Thread.class
public class Thread implements Runnable {
    //每个线程都有自己的ThreadLocalMap 成员变量
    ThreadLocal.ThreadLocalMap threadLocals = null;
}
```

这里是在当前线程对象的ThreadlocalMap中put了一个元素(Entry)，key是**Threadlocal对象**，value是userInfo。

理解二件事就都清楚了：

ThreadLocalMap 类的定义在 Threadlocal中。

- 第一，Thread 对象是Java语言中线程运行的载体，每个线程都有对应的Thread 对象，存放线程相关的一些信息，
- 第二，Thread类中有个成员变量ThreadlocalMap，你就把他当成普通的Map，key存放的是Threadlocal对象，value是你要跟线程绑定的值（线程隔离的变量），比如这里是用户信息对象（UserInfo）。

面试官：你刚才说Thread 类有个 ThreadlocalMap 属性的成员变量，但是ThreadlocalMap 的定义却在Threadlocal 中，为什么这么做?

安琪拉：我们看下ThreadlocalMap的说明

> ```
> class ThreadLocalMap
> * ThreadLocalMap is a customized hash map suitable only for
> * maintaining thread local values. No operations are exported
> * outside of the ThreadLocal class. The class is package private to
> * allow declaration of fields in class Thread.  To help deal with
> * very large and long-lived usages, the hash table entries use
> * WeakReferences for keys. However, since reference queues are not
> * used, stale entries are guaranteed to be removed only when
> * the table starts running out of space.
> ```

大概意思是ThreadLocalMap 就是为维护线程本地变量而设计的，只做这一件事情。

这个也是为什么 ThreadLocalMap 是Thread的成员变量，但是却是Threadlocal 的内部类（非public，只有包访问权限，Thread和Threadlocal都在java.lang 包下），就是让使用者知道ThreadLocalMap就只做保存线程局部变量这一件事的。

面试官：既然是线程局部变量，那为什么不用线程对象（Thread对象）作为key，这样不是更清晰，直接用线程作为key获取线程变量?

安琪拉：这样设计会有个问题，比如: 我已经把用户信息存在线程变量里了，这个时候需要新增加一个线程变量，比方说新增用户地理位置信息，我们ThreadlocalMap 的key用的是线程，再存一个地理位置信息，key都是同一个线程（key一样），不就把原来的用户信息覆盖了嘛。Map.put(key,value) 操作熟悉吧，所以网上有些文章说ThreadlocalMap使用线程作为key是瞎扯

的。

面试官：那新增地理位置信息应该怎么做?

安琪拉：新创建一个Threadlocal对象就好了，因为ThreadLocalMap的key是Threadlocal 对象，比如新增地理位置，我就再 Threadlocal < Geo> geo = new Threadlocal（），存放地理位置信息，这样线程的ThreadlocalMap里面会有二个元素，一个是用户信息，一个是地理位置。

面试官：ThreadlocalMap 是什么数据结构实现的?

安琪拉：跟HashMap 一样，也是数组实现的。

代码如下：

```
class ThreadLocalMap {
 //初始容量
 private static final int INITIAL_CAPACITY = 16;
 //存放元素的数组
 private Entry[] table;
 //元素个数
 private int size = 0;
}
```

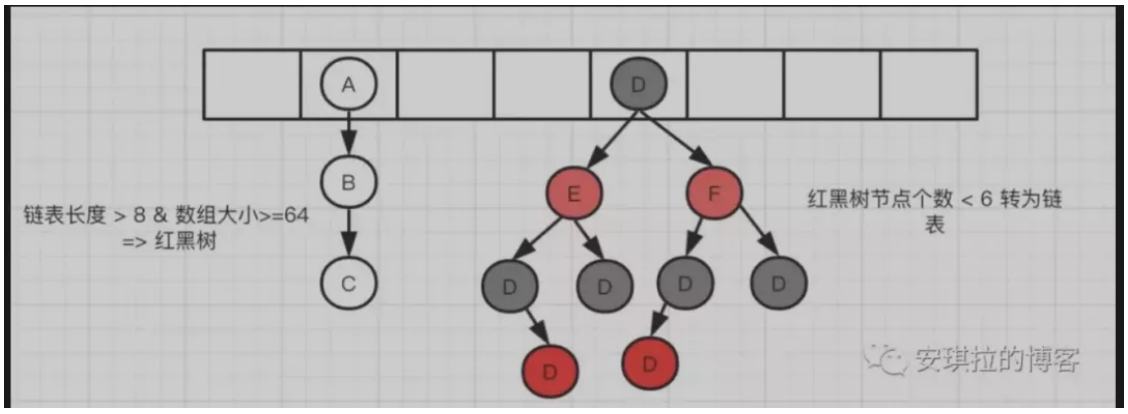
table 就是存储线程局部变量的数组，数组元素是Entry类，Entry由key和value组成，key是Threadlocal对象，value是存放的对应线程变量

我们前面举得例子，数组存储结构如下图:

面试官：ThreadlocalMap 发生hash冲突怎么办？跟HashMap 有什么区别?

安琪拉：【心想】第一次碰到有问ThreadlocalMap哈希冲突的，这个面试越来越有意思了。

说道：有区别的，对待哈希冲突，HashMap采用的链表 + 红黑树的形式，如下图，链表长度过长(>8) 就会转成红黑树:

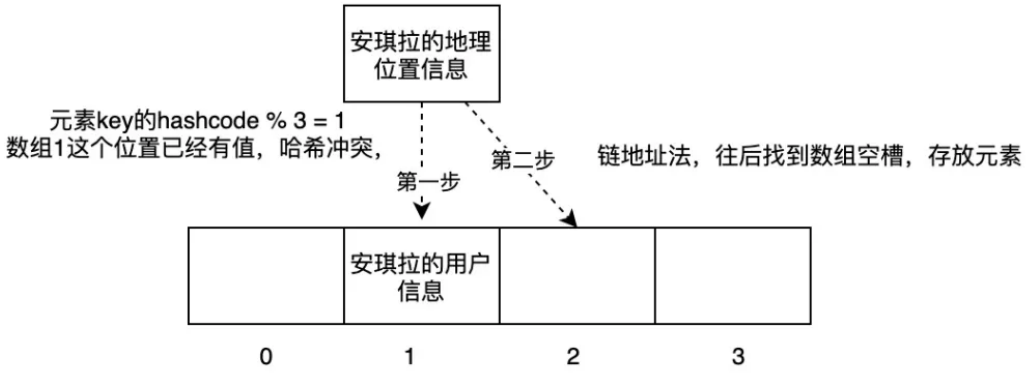


HashMap详解：

> 参考
>
> 安琪拉，公众号：安琪拉的博客[一个HashMap跟面试官扯了半个小时]

ThreadlocalMap既没有链表，也没有红黑树，采用的是开放定址法 ，是这样，是如果发生冲突，ThreadlocalMap直接往后找**相邻**的下一个节点，如果相邻节点为空，直接存进去，如果不为空，继续往后找，直到找到空的，把元素放进去，或者元素个数超过数组长度阈值，进行扩容。

如下图：还是以之前的例子讲解，ThreadlocalMap 数组长度是4，现在存地理位置的时候发生hash冲突（位置1已经有数据），那就把往后找，发现2 这个位置为空，就直接存放在2这个位置。



源代码（如果阅读起来困难，可以看完后文回过头来阅读）：

```
private void set(ThreadLocal<?> key, Object value) {
  Entry[] tab = table;
  int len = tab.length;
  // hashcode & 操作其实就是 %数组长度取余数，例如：数组长度是4，hashCode % (4-1) 就找到要存放元素的数组下标
  int i = key.threadLocalHashCode & (len-1);

  //找到数组的空槽（=null），一般ThreadlocalMap存放元素不会很多
  for (Entry e = tab[i];
       e != null; //找到数组的空槽（=null）
       e = tab[i = nextIndex(i, len)]) {
    ThreadLocal<?> k = e.get();

    //如果key值一样，算是更新操作，直接替换
    if (k == key) {
      e.value = value;
      return;
    }
  //key为空，做替换清理动作，这个后面聊WeakReference的时候讲
    if (k == null) {
      replaceStaleEntry(key, value, i);
      return;
    }
  }
 //新new一个Entry
  tab[i] = new Entry(key, value);
  //数组元素个数+1
  int sz = ++size;
  //如果没清理掉元素或者存放元素个数超过数组阈值，进行扩容
  if (!cleanSomeSlots(i, sz) && sz >= threshold)
    rehash();
}

//顺序遍历 +1 到了数组尾部，又回到数组头部（0这个位置）
private static int nextIndex(int i, int len) {
  return ((i + 1 < len) ? i + 1 : 0);
```

```
}

// get()方法，根据ThreadLocal key获取线程变量
private Entry getEntry(ThreadLocal<?> key) {
  //计算hash值 & 操作其实就是 %数组长度取余数，例如：数组长度是4，hashCode % (4-1) 就找到要查询的数组地址
  int i = key.threadLocalHashCode & (table.length - 1);
  Entry e = table[i];
  //快速判断 如果这个位置有值，key相等表示找到了，直接返回
  if (e != null && e.get() == key)
    return e;
  else
    return getEntryAfterMiss(key, i, e); //miss之后顺序往后找（链地址法，这个后面再介绍）
}
```

面试官：我看你最前面图中画的ThreadlocalMap 中key是 WeakReference类型，能讲讲Java中有几种类似的引用，什么区别吗？

安琪拉：可以

- **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，当**内存空间不足**时，`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。
- 如果一个对象只具有**软引用**，则**内存空间充足**时，**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。
- **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内存区域时，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。
- **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。

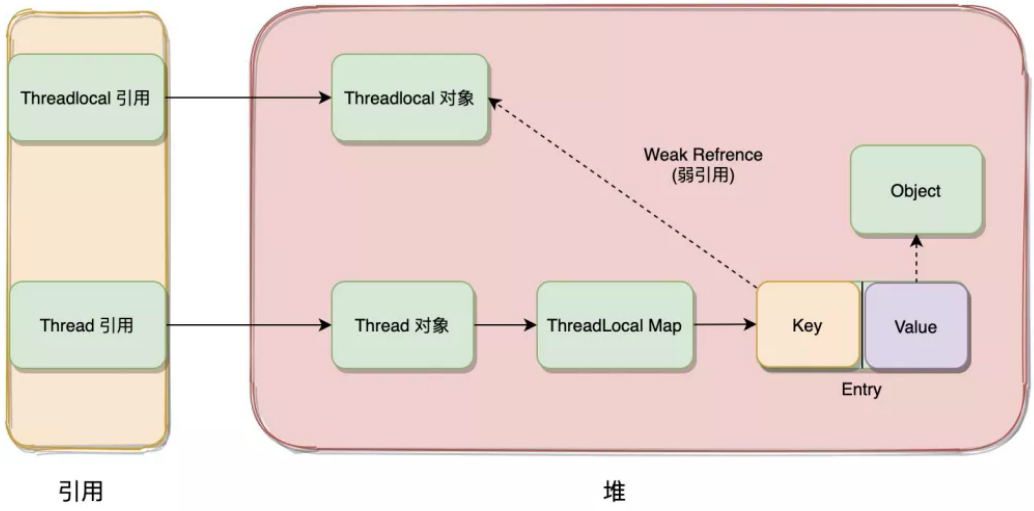
妥妥的八股文啊！尴尬(ー.ー|||。

面试官：那你能讲讲为什么ThreadlocalMap 中key 设计成 WeakReference（弱引用）类型吗？

安琪拉：可以的，为了尽最大努力避免内存泄漏。

面试官：能详细讲讲吗？为什么是尽最大努力，你前面也讲被WeakReference 引用的对象会直接被GC（内存回收器） 回收，为什么不是直接避免了内存泄漏呢？

安琪拉：我们还是看下下面这张图



```
private static final ThreadLocal<UserInfo> userInfoThreadLocal = new
ThreadLocal<>();
userInfoThreadLocal.set(userInfo);
```

这里的引用关系是userInfoThreadLocal 引用了ThreadLocal对象，这是个强引用，ThreadLocal对象同时也被ThreadlocalMap的key引用，这是个WeakReference引用，我们前面说GC要回收ThreadLocal对象的前提是它只被WeakReference引用，没有任何强引用。

为了方便大家理解弱引用，我写了段Demo程序

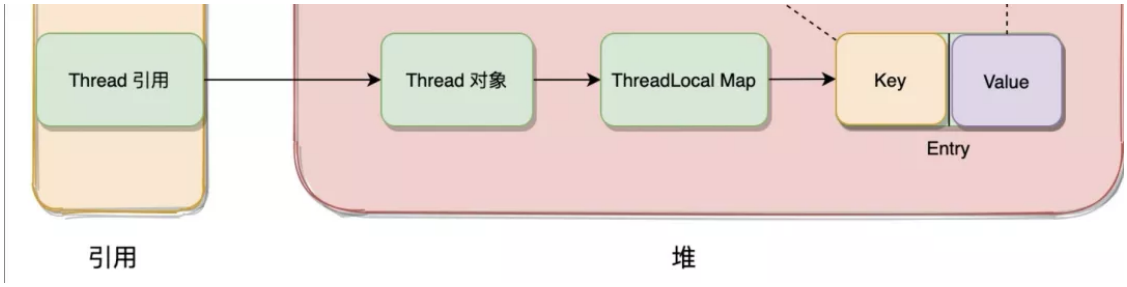
```
public static void main(String[] args) {
  Object angela = new Object();
  //弱引用
  WeakReference<Object> weakReference = new WeakReference<>(angela);
  //angela和弱引用指向同一个对象
  System.out.println(angela);//java.lang.Object@4550017c
  System.out.println(weakReference.get());//java.lang.Object@4550017c
  //将强引用angela置为null，这个对象就只剩下弱引用了,内存够用，弱引用也会被回收
  angela = null;
  System.gc();//内存够用不会自动gc，手动唤醒gc
  System.out.println(angela);//null
  System.out.println(weakReference.get());//null
}
```

可以看到一旦一个对象只被弱引用引用，GC的时候就会回收这个对象。

所以只要ThreadLocal对象如果还被 userInfoThreadLocal（强引用） 引用着，GC是不会回收被WeakReference引用的对象的。

面试官：那既然ThreadLocal对象有强引用，回收不掉，干嘛还要设计成WeakReference类型呢?

安琪拉：ThreadLocal的设计者考虑到线程往往生命周期很长，比如经常会用到线程池，线程一直存活着，根据JVM **根搜索算法**，一直存在 Thread -> ThreadLocalMap -> Entry（元素）这样一条引用链路, 如下图，如果key不设计成WeakReference类型，是强引用的话，就一直不会被GC回收，key就一直不会是null，不为null Entry元素就不会被清理（ThreadLocalMap是根据key是否为null来判断是否清理Entry）



所以ThreadLocal的设计者认为只要ThreadLocal 所在的作用域结束了工作被清理了，GC回收的时候就会把key引用对象回收，key置为null，ThreadLocal会尽力保证Entry清理掉来最大可能避免内存泄漏。

来看下代码

```
//元素类
static class Entry extends WeakReference<ThreadLocal<?>> {
  /** The value associated with this ThreadLocal. */
  Object value; //key是从父类继承的，所以这里只有value

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

```
    value = v;
  }
}

//WeakReference 继承了Reference，key是继承了范型的referent
public abstract class Reference<T> {
  //这个就是被继承的key
  private T referent;
  Reference(T referent) {
    this(referent, null);
  }
}
```

Entry 继承了WeakReference类，Entry 中的 key 是WeakReference类型的，在Java 中当对象只被 WeakReference 引用，没有其他对象引用时，被WeakReference 引用的对象发生GC 时会直接被回收掉。

面试官：那如果Threadlocal 对象一直有强引用，那怎么办？岂不是有内存泄漏风险。

安琪拉：最佳实践是用完手动调用remove函数。

我们看下源码：

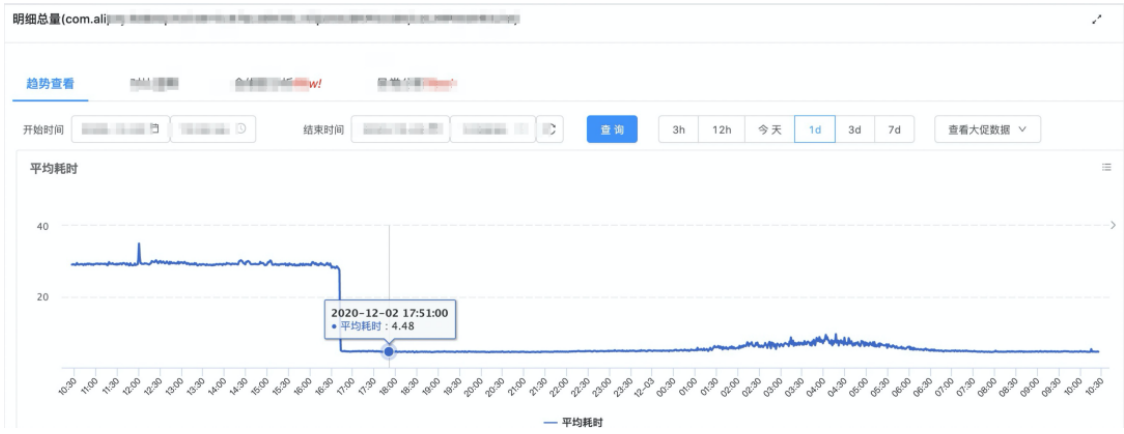
```
class Threadlocal {
  public void remove() {
      //这个是拿到线程的ThreadLocalMap
      ThreadLocalMap m = getMap(Thread.currentThread());
      if (m != null)
        m.remove(this); //this就是ThreadLocal对象，移除，方法在下面
  }
}

class ThreadlocalMap {
  private void remove(ThreadLocal<?> key) {
    Entry[] tab = table;
    int len = tab.length;
    //计算位置
    int i = key.threadLocalHashCode & (len-1);
    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
      //清理
      if (e.get() == key) {
        e.clear();
        expungeStaleEntry(i); //清理空槽
        return;
      }
   }
 }
}

//这个方法就是做元素清理
private int expungeStaleEntry(int staleSlot) {
  Entry[] tab = table;
  int len = tab.length;

  //把staleSlot的value置为空，然后数组元素置为空
  tab[staleSlot].value = null;
```

```
    tab[staleSlot] = null;
    size--; //元素个数-1

    // Rehash until we encounter null
    Entry e;
    int i;
    for (i = nextIndex(staleSlot, len);
         (e = tab[i]) != null;
         i = nextIndex(i, len)) {
      ThreadLocal<?> k = e.get();
      //k 为null代表引用对象被GC回收掉了
      if (k == null) {
        e.value = null;
        tab[i] = null;
        size--;
      } else {
        //因为元素个数减少了，就把后面的元素重新hash
        int h = k.threadLocalHashCode & (len - 1);
        //hash地址不相等，就代表这个元素之前发生过hash冲突(本来应该放在这没放在这)，
        //现在因为有元素被移除了，很有可能原来冲突的位置空出来了，重试一次
        if (h != i) {
          tab[i] = null;

          //继续采用链地址法存放元素
          while (tab[h] != null)
            h = nextIndex(h, len);
          tab[h] = e;
        }
      }
    }
    return i;
  }
```

面试官：你有没有用Threadlocal的工程实际经历，给我讲讲。

安琪拉：有啊！

之前我跟你们一面面试官聊过，我是怎么把支付宝后台负责的系统四十几个核心rpc接口性能大幅度提升的，下面这个就是其中一个接口切流之后的效果，其中就用到了`Threadlocal`。



面试官：嗯，说说。

安琪拉：我刚才说有四十多个接口要做技改优化，那风险是很高的，我需要保证接口切换后业务不受影响，也叫等效切换。

流程是这样的：

- 把这四十多个接口按照业务含义定义了接口常量名称，比如接口名`alipay.quickquick.follow.angela`；
- 按照接口的流量从低到高开始切流，提前配置中心配置好每个接口的切流比例和用户白名单；
- 切流也有讲究，先按照userId白名单切，再按照userId尾号切百分比，完全没问题再完整切；
- 在顶层抽象模版方法的入口通过ThreadLocal Set 接口名，把接口名塞进去；
- 然后我在切流的地方通过ThreadLocal 获取接口名，用于接口切流判断切流；

面试官：最后一个问题，如果我有很多变量都要塞到ThreadlocalMap中，那岂不是要申明很多个Threadlocal 对象？有没有好的解决办法。

安琪拉：我们的最佳实践是搞个再封装一下，把ThreadLocalMap 的value 弄成Map就好了，这样只要一个Threadlocal 对象就好了。

面试官：能详细讲讲吗？

安琪拉：讲不动了，太累了。

面试官：讲讲。

安琪拉：真不想讲了。

面试官：那今天先到这，您出了这个门右拐，回去等通知吧！