JDK早期：
ThreadLocal
  ThreadLocalMap
    KEY VALUE
    Thread value

JDK8:
Thread
  private ThreadLocalMap threadLocals
    KEY VALUE
    ThreadLocal value

Thread中属性
  private ThreadLocal.ThreadLocalMap threadLocals 存放KEY VALUE
  KEY是ThreadLocal对象本身，value是真实设置的值。

```
public class ThreadLocal {
    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);
    }

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

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

    private T setInitialValue() {
        T value = initialValue();
```

```
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
        return value;
    }

    protected T initialValue() {
        return null;
    }

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

总结代码：

set()方法：

先获取当前线程的ThreadLocalMap对象，如果没有就创建一个，将本ThreadLocal作为KEY, 将value作为值存进去。

get()方法：

先获取当前线程的ThreadLocalMap对象，根据本ThreadLocal对象当作Key进行查找，如果存在则返回值，如果不存在则创建并返回初始值。

remove()方法：

先获取当前线程的ThreadLocalMap对象，根据本ThreadLocal对象当作Key删除这个KEY对应的Entry。

protected T initialValue()方法，默认返回null：

如果从来没set过直接先执行get(), 会调用setInitialValue中initialValue方法获取默认值，并进行初始化，所以对于每个线程中的一个ThreadLocal, 这个initialValue指挥被调用一次。

可以继承ThreadLocal复写这个方法，来改变默认值。

JDK8设计相对于之前的好处：

1.ThreadLocalMap存储的Entry数量变少。

原来的设计中ThreadLocalMap在ThreadLocal中，意味着有多少个线程，就有多少个entry(MAP中的元素)。

JDK8设计中ThreadLocalMap在Thread中，KEY是ThreadLocal对象本身，value是真实设置的值。意味着有几个ThreadLocal变量，ThreadLocalMap就有几个Entry.

开发中ThreadLocal的数量要远远少于Thread的数量的。

2.JDK8中，因为ThreadLocalMap是Thread中的一个成员变量，当Thread销毁时，ThreadLocalMap也会随之销毁。减少内存使用。

而早期ThreadLocalMap是ThreadLocal的成员变量，而Thread只是作为ThreadLocalMap的KEY存在。当Thread销毁后，ThreadLocalMap仍然存在。

ThreadLocalMap：

ThreadLocalMap中，Entry来存放K-V结构数据，KEY只能时ThreadLocal对象，在构造方法中已经限定死了。

Entry继承WeakReference, Entry的key也就是Threadlocal对象是WeakReference类中的reference成员变量，也就是说KEY对应的Threadlocal对象是一个弱引用，其目的是将Threadlocal对象的生命周期与线程生命周期解绑。

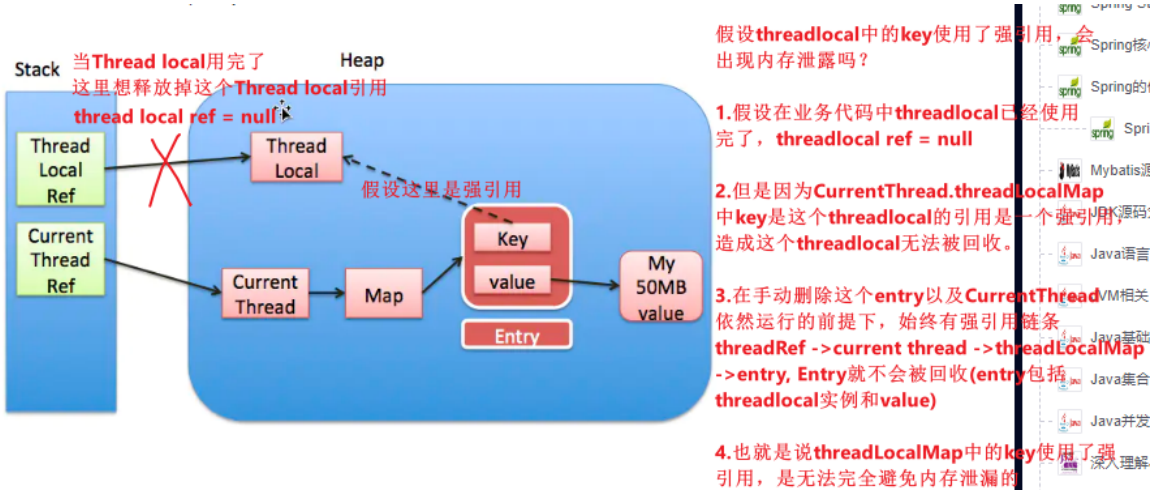
static class ThreadLocalMap {

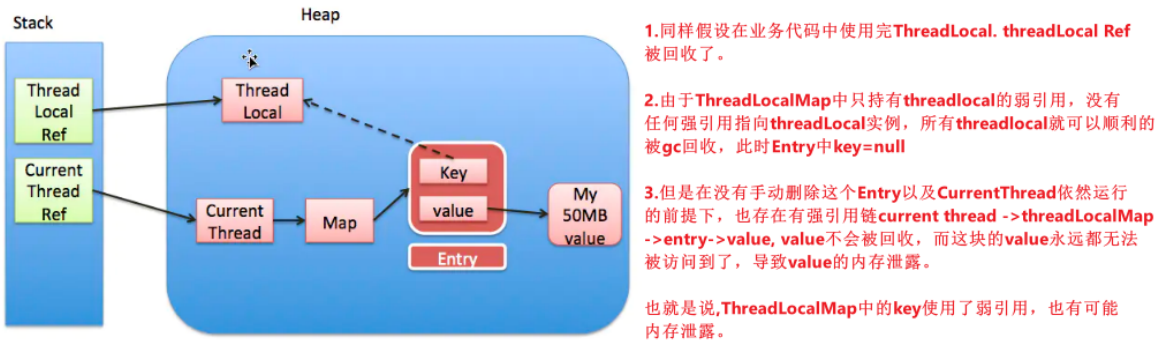
```
static class Entry extends WeakReference<ThreadLocal<?>> {
    /**
     * The value associated with this ThreadLocal.
     */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k); //k是WeakReference中的成员变量reference
        value = v;
    }
}
```

}

弱引用：垃圾回收期一旦发现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。如果弱引用的对象，被其他强引用所引用，那么他依然不会被回收掉。





因此，无论是threadLocalMap中的key使用强引用还是弱引用都会导致内存泄露。而threadLocalMap是当前线程的成员，所以在没有手动删除

entry的前提下，他的生命周期和当前线程一样长，当前线程销毁了，他就销毁了。还有一种方法就是在使用完这个threadLocal后及时调用删除他对应的entry.

综上：ThreadLocal内存泄露的根本原因是：由于ThreadLocalMap的生命周期和Thread一样长，如果没有手动删除对应key就会导致内存泄漏。

当使用到了线程池，线程不会及时销毁，更需要注意及时调用threadLocal的remove方法，及时手动删除对应的key以防止内存泄露。

既然ThreadLocalMap中的Entry中的key无论是强引用还是弱引用都不能避免内存泄露，为什么要定义为弱引用呢?

当线程长时间存活，且在使用完后一直没有手动删除ThreadLocalMap中的ThreadLocal类型的Key，因为ThreadLocalMap中Entry的key是弱引用，当gc垃圾回收时，会将弱引用清除掉，变为null。

而ThreadLocalMap中的set和getEntry方法，会对KEY为Null(即ThreadLocal为null) 进行判断，如果为null的话，那么会将value也置为null, 这样就释放掉了value。

这就意味着使用完ThreadLocal, CurrentThread依然在运行的前提下，就算忘记调用remove方法，弱引用比强引用可以多一层保障：弱引用的threadlocal会被gc垃圾回收，对应的value在下一次ThreadLocalMap调用set,get,remove中任意一个方法的时候会被清除为null, 从而避免内存泄露。