ThreadLocal ，也叫线程本地变量，可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了所使用的的变量副本。使用起来都是在线程的本地工作内存中操作，并且提供了set和get方法来访问拷贝过来的变量副本。底层也是封装了ThreadLocalMap集合类来绑定当前线程和变量副本的关系，各个线程独立并且访问安全！

set操作，为线程绑定变量：

public void set(T value) {

　　 Thread t = Thread.currentThread();//1.首先获取当前线程对象

ThreadLocalMap map = getMap(t);//2.获取该线程对象的ThreadLocalMap

if (map != null)

map.set(this, value);//如果map不为空，执行set操作，以当前threadLocal对象为key，实际存储对象为value进行set操作

else

createMap(t, value);//如果map为空，则为该线程创建ThreadLocalMap

}

可以很清楚的看到，ThreadLocal只不过是个入口，真正的变量副本绑定到当前线程上的。

//Thread中的成员变量

    ThreadLocal.ThreadLocalMap threadLocals = null; //每个Thread线程中都封装了一个ThreadLocalMap对象

    //ThreadLocal类中获取Thread类中的ThreadLocalMap对象

    ThreadLocalMap getMap(Thread t) {

return t.threadLocals;

    }

    //ThreadLocal类中创建Thread类中的ThreadLocalMap成员对象

    void createMap(Thread t, T firstValue) {

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

}

现在，我们可以看出ThreadLocal的设计思想了：

(1) ThreadLocal仅仅是个变量访问的入口；

(2) 每一个Thread对象都有一个ThreadLocalMap对象，这个ThreadLocalMap持有对象的引用；

(3) ThreadLocalMap以当前的threadLocal对象为key，以真正的存储对象为value。get()方法时通过threadLocal实例就可以找到绑定在当前线程上的副本对象。

看上去有点绕。我们完全可以设计成Map<Thread,Value>这种形式，一个线程对应一个存储对象。

ThreadLocal这样设计有两个目的：

第一：可以保证当前线程结束时，相关对象可以立即被回收；第二：ThreadLocalMap元素会大大减少，因为Map过大容易造成哈希冲突而导致性能降低。

get操作，为线程获取变量：

public T get() {

　　 Thread t = Thread.currentThread();//1.首先获取当前线程

ThreadLocalMap map = getMap(t);//2.获取线程的map对象

if (map != null) {//3.如果map不为空，以threadlocal实例为key获取到对应Entry，然后从Entry中取出对象即可。

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

if (e != null)

return (T)e.value;

}

return setInitialValue();//如果map为空，也就是第一次没有调用set直接get（或者调用过set，又调用了remove）时，为其设定初始值

}

void createMap(Thread t, T firstValue) { //this指的是ThreadLocal对象

    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

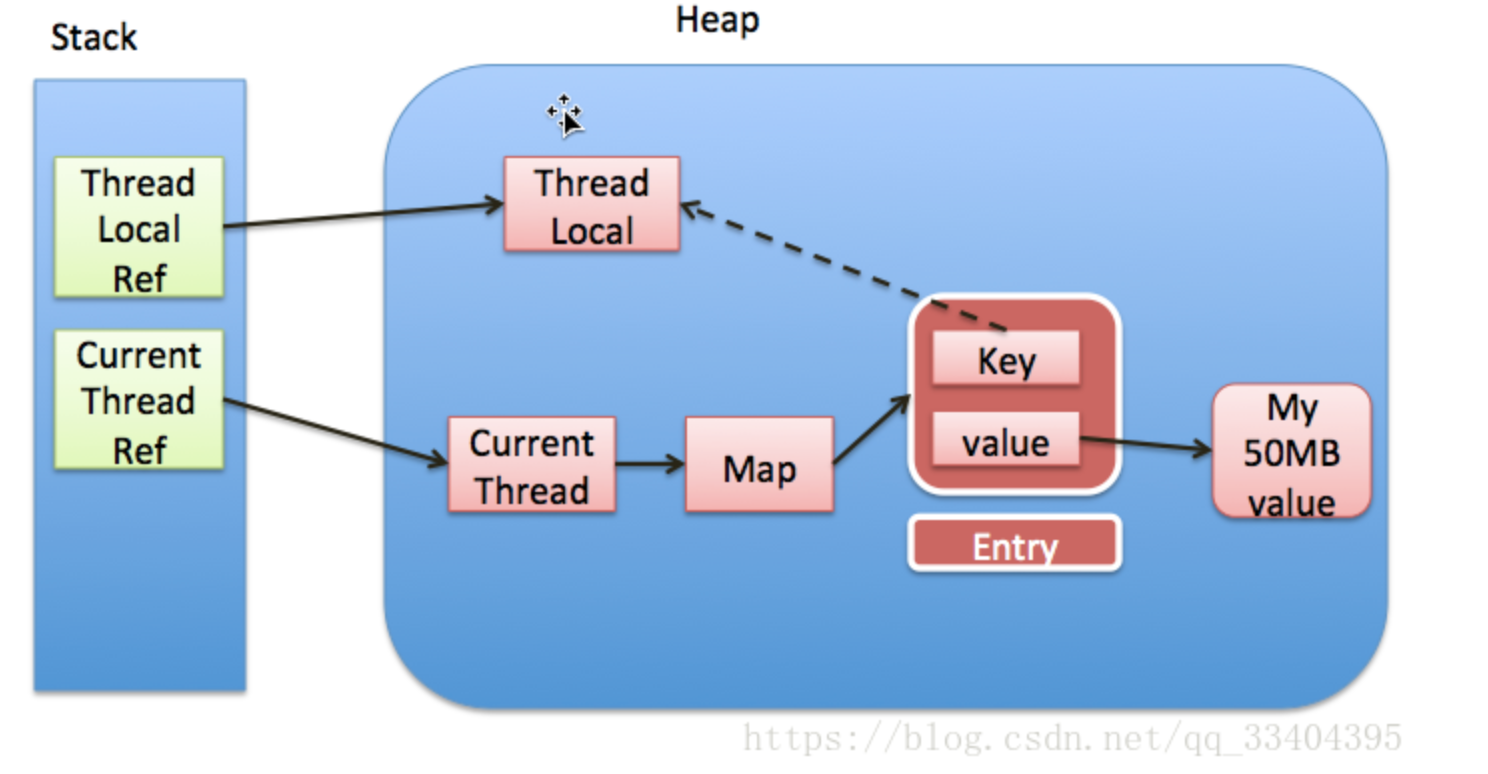
        createMap(t, value);

        return value;

}

ThreadLocal对象通常用于防止对可变的单实例变量或全局变量进行共享。例如：由于对象不是线程安全的，因此，当多个线程应用程序在没有协同的情况下，使用全局变量时，就是线程不安全的。通过将对象保存到ThreadLocal中，每个线程都会拥有自己的连接对象副本。

ThreadLocal为什么会内存泄漏？



ThreadLocal的实现是这样的：每个Thread 维护一个 ThreadLocalMap 映射表，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。

值得注意的是图中的虚线，表示 ThreadLocalMap 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。

ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统 GC 的时候，这个ThreadLocal势必会被回收，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永远无法回收，造成内存泄漏。

ThreadLocal如何防止内存泄漏？

每次使用完ThreadLocal，都调用它的remove()方法，清除数据。

在使用线程池的情况下，没有及时清理ThreadLocal，不仅是内存泄漏的问题，更严重的是可能导致业务逻辑出现问题。所以，使用ThreadLocal就跟加锁完要解锁一样，用完就需要清理。