一，ThreadLocal的基本使用与原理

为线程创建独一份的副本数据。

1，基本使用

```
/**
 * @author yhd
 * @createtime 2021/2/17 11:31
 */
public class TheadLocalDemo {

    private static final AtomicInteger nextId = new AtomicInteger(0);

    private static final ThreadLocal<Integer> threadId=
ThreadLocal.withInitial(() -> nextId.getAndIncrement());

    public static int get(){
        return threadId.get();
    }

    public static void main(String[] args)throws Exception {
        RunnableTask task = new RunnableTask();

        new Thread(task,"A").start();
        TimeUnit.SECONDS.sleep(1);

        new Thread(task,"B").start();
        TimeUnit.SECONDS.sleep(1);

        new Thread(task,"C").start();
        TimeUnit.SECONDS.sleep(1);
    }


    static class RunnableTask implements Runnable{
        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + "-----"+
threadId.get());
            }finally {
                threadId.remove();
            }
        }
    }
}
```

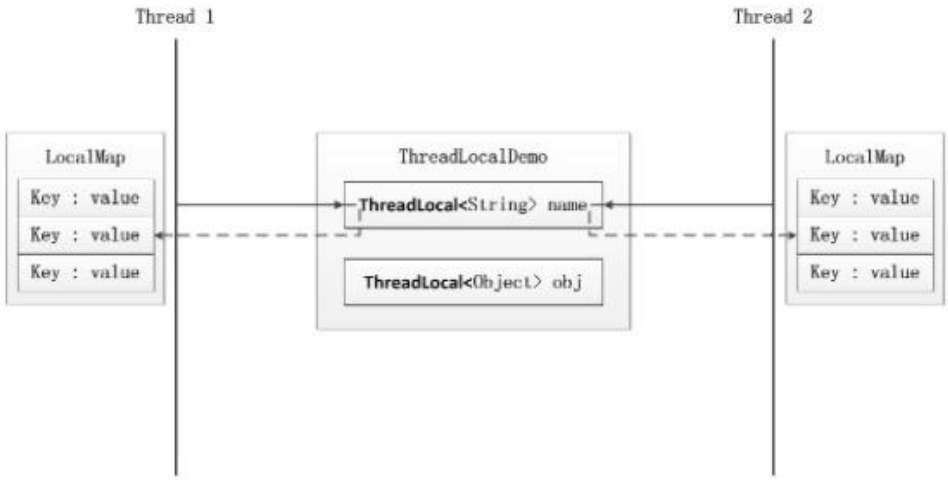
2，原理分析

里面维护一个ThreadLocalMap结构，每一个元素对应一个桶位。

使用ThreadLocal定义的变量，将指向当前线程本地的一个LocalMap空间。

ThreadLocal变量作为key，其内容作为value，保存在本地。

多线程对ThreadLocal对象进行操作，实际上是对各自的本地变量进行操作，不存在线程安全问题



二，源码分析

1，属性

```
    /**
     * 线程获取ThreadLocal.get()时，如果是第一次在某个threadLocal对象上get时，会给当前线程分配一个value
     * 这个value和当前的threadLocal对象被包装成一个entry ，其中key=threadLocal对象，value=threadLocal
     * 对象给当前线程生成的value。这个entry存放到那个位置与这个value有关。
     */
    private final int threadLocalHashCode = nextHashCode();

    //创建threadLocal对象时会使用到，每创建一个threadLocal对象就会使用他分配一个hash值给对象。
    private static AtomicInteger nextHashCode =
            new AtomicInteger();

    //没创建一个threadLocal对象，这个ThreadLocal.nextHashCode就会增长0x61c88647。
    private static final int HASH_INCREMENT = 0x61c88647;

    //创建新的threadLocal对象时，给当前对象分配hash时用到
    private static int nextHashCode() {
        return nextHashCode.getAndAdd(HASH_INCREMENT);
    }

    //需要重写的
    protected T initialValue() {
        return null;
    }
```

2，get（）

```
    /**
     * 返回当前线程与当前threadLocal对象相关联的线程局部变量，这个变量只有当前线程能访问到。
```

```
     * 如果当前线程未分配，则给当前线程分配。
     *
     * @return
     */
    public T get() {
        Thread t = Thread.currentThread();//获取当前线程
        ThreadLocalMap map = getMap(t);//获取当前线程thread对象的threadLocals map引
用
        if (map != null) { //已经初始化过
            //根据key（当前对象threadLocal）去拿到entry
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {//说明当前线程初始化过
                @SuppressWarnings("unchecked")
                T result = (T) e.value;
                return result;
            }
        }
        //执行到这里要么当前线程对应额threadLocals==null，要么当前线程没有生成相关联的局部
变量。
        //初始化当前线程与threadLocal对象相关联的value，且如果threadLocals没初始化还会初
始化threadLocals
        return setInitialValue();
    }
```

1）setInitialValue()

```
//初始化当前线程与threadLocal对象相关联的value，且如果threadLocals没初始化还会初始化
threadLocals
private T setInitialValue() {
    //调用当前对象的的initialValue()
    //value就是当前ThreadLocal对象与当前线程相关联的线程局部变量
    T value = initialValue();
    Thread t = Thread.currentThread();
    //获取当前线程内部的threadLocals
    ThreadLocalMap map = getMap(t);
    if (map != null) //如果threadLocals已经初始化过，直接将值放进去
        map.set(this, value);
    else  //没初始化过，执行初始化，并将值设置进去。
        createMap(t, value);
    return value;
}
```

2）getMap（）

```
//返回当前线程的threadLocals
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

3）createMap（）

```
//利用构造器初始化threadLocals并将当前线程和线程对应的value设置进去
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

3，set()

```
//将当前线程与对应的value设置进threadLocals
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)//threadLocals已经初始化
        map.set(this, value);
    else//threadLocals未初始化
        createMap(t, value);
}
```

4,remove()

```
public void remove() {
    //从threadLocals获取到当前线程对应的threadLocals
    ThreadLocalMap m = getMap(Thread.currentThread());
    if (m != null)//如果threadLocals不为空
        m.remove(this);//移除当前线程对应的key-val
}
```

5,内部类ThreadLocalMap

1）Entry

```
/**
 * key是弱引用保留，key保存的是threadLocal对象
 * value使用的是强引用，value保存的是threadLocal对象与当前线程关联的value
 * 这样设计的好处？
 * help GC 防止内存泄漏
 */
static class Entry extends WeakReference<ThreadLocal<?>> {
    /**
     * The value associated with this ThreadLocal.
     */
    Object value;

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

2）属性

```
//初始化当前threadLocals的长度
private static final int INITIAL_CAPACITY = 16;

//散列表的引用，必须是2的次方数
private Entry[] table;

//当前撒列表占用情况
private int size = 0;

//扩容阈值  当前数组长度的2/3
//先rehash全量检查过期数据，把所有过期的Entry移除
//移除之后在判断如果达到整个threadLocals的四分之三，就resize()
private int threshold; // Default to 0

//将阈值设置为当前数组长度的2/3
private void setThreshold(int len) {
    threshold = len * 2 / 3;
}

//获取下一个位置
private static int nextIndex(int i, int len) {
    return ((i + 1 < len) ? i + 1 : 0);
}

//获取上一个位置
private static int prevIndex(int i, int len) {
    return ((i - 1 >= 0) ? i - 1 : len - 1);
}
```

3）构造器

```
        //延迟初始化，只有现成第一次存储或者获取的时候，才会创建threadLocals并初始化
        ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
            table = new Entry[INITIAL_CAPACITY];
            int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
            table[i] = new Entry(firstKey, firstValue);
            size = 1;
            setThreshold(INITIAL_CAPACITY);
        }
```

4）getEntry（）

```
//threadLocal.get()-->this
//key是threadLocal对象
private Entry getEntry(ThreadLocal<?> key) {
    //寻址
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];//获取元素
    if (e != null && e.get() == key) //检验
        return e;
    else
        //继续向当前桶位后面搜索，直到找到，如果没找到，说明被GC了，会做一次探测式过期检查
        //  因为threadLocals不支持hash冲突，冲突了就放在下一位
        return getEntryAfterMiss(key, i, e);
}
```

expungeStaleEntry()

```
        //探测式过期检查
        private int expungeStaleEntry(int staleSlot) {
            Entry[] tab = table;
            int len = tab.length;
            //staleSlot的位置就是一个过期数据，所以从这里开始往后检查，如果过期了就置空
            tab[staleSlot].value = null;
            tab[staleSlot] = null;
            size--;
            Entry e;
            int i;
            //循环从staleSlot+1开始搜索过期数据，直到碰到slot=null结束
            for (i = nextIndex(staleSlot, len);
                 (e = tab[i]) != null;
                 i = nextIndex(i, len)) {
                ThreadLocal<?> k = e.get();
                //说明key表示的threadLocal对象已经被回收了，当前的entry属于脏数据，置空
                if (k == null) {
                    e.value = null;
                    tab[i] = null;
                    size--;
                } else {
                    //执行到这里说明当前遍历的slot中的entry是非过期数据，因为前面有可能清理
掉几个过期数据
                    //且有可能当前这个位置存储的时候可能碰到hash冲突，因此做位置优化，重新计
算。
                    //这样查询的效率会更高，更准确。
                    int h = k.threadLocalHashCode & (len - 1);
                    if (h != i) {
                        tab[i] = null;
                        while (tab[h] != null)
                            h = nextIndex(h, len);
                        tab[h] = e;
                    }
                }
            }
            return i;
        }
```

5）set()

```
private void set(ThreadLocal<?> key, Object value) {
    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len - 1);//计算hash值
    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get();

        if (k == key) { //如果已经有值，覆盖
            e.value = value;
            return;
        }

        if (k == null) { //碰到过期数据，走替换逻辑
            replaceStaleEntry(key, value, i);
            return;
        }
    }
    //真正的新数据，创建一个新的Entry
    tab[i] = new Entry(key, value);
    int sz = ++size;
    if (!cleanSomeSlots(i, sz) && sz >= threshold) //如果达到扩容条件，扩容
        rehash();
}
```

replaceStaleEntry()

```
        //替换过期entry
        private void replaceStaleEntry(ThreadLocal<?> key, Object value,
                                       int staleSlot) {
            Entry[] tab = table;
            int len = tab.length;
            Entry e;

            //表示开始探测式清理过期数据的下标
            //从当前位置往前迭代，查找过期数据，直到碰到null
            int slotToExpunge = staleSlot;
            for (int i = prevIndex(staleSlot, len);
                 (e = tab[i]) != null;
                 i = prevIndex(i, len))
                if (e.get() == null)//如果找到了，下标-1
                    slotToExpunge = i;

            //从当前位置往后迭代，知道碰到null为止
            for (int i = nextIndex(staleSlot, len);
                 (e = tab[i]) != null;
                 i = nextIndex(i, len)) {
                ThreadLocal<?> k = e.get();


                if (k == key) { //说明是一个替换逻辑
                    e.value = value;//替换新数据
```

```
                tab[i] = tab[staleSlot]; //将过期数据放到当前循环到的table[i]
                tab[staleSlot] = e; //位置优化，其实就是


                if (slotToExpunge == staleSlot)//说明往前找并没有过期数据
                    slotToExpunge = i;//吧探测的开始位置改成当前位置
                cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
                return;
            }

            //当前遍历entry是一个过期数据 && 往前找过期数据没找到
            if (k == null && slotToExpunge == staleSlot)
                //向后查询过程中查找到了一个过期数据，更新探测位置为当前位置
                slotToExpunge = i;
        }


        tab[staleSlot].value = null;
        tab[staleSlot] = new Entry(key, value);


        if (slotToExpunge != staleSlot)
            cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
    }
```

cleanSomeSlots()

```
    //启发式清理工作 i 开始清理位置  n 结束条件，数组长度
    private boolean cleanSomeSlots(int i, int n) {
        boolean removed = false;
        Entry[] tab = table;
        int len = tab.length;
        do {
            //获取当前i的下一个下标
            i = nextIndex(i, len);
            Entry e = tab[i]; //获取当前下标为I的元素
            if (e != null && e.get() == null) {  //断定为过期元素
                n = len; //更新数组长度
                removed = true;
                i = expungeStaleEntry(i); //从当前过期位置开始一次谈测试清理工作
            }
        } while ((n >>>= 1) != 0); //假设table.length=16 ，
        return removed;
    }
```

6)rehash()

```
    //扩容相关
    private void rehash() {
        //遍历，探测式清理，干掉所有过期数据
        expungeStaleEntries();
        //仍然达到扩容条件
        if (size >= threshold - threshold / 4)
            resize();
    }
```

resize()

```
        private void resize() {
            Entry[] oldTab = table;
            int oldLen = oldTab.length;
            int newLen = oldLen * 2; //扩容为原来的2倍
            Entry[] newTab = new Entry[newLen];
            int count = 0;

            for (int j = 0; j < oldLen; ++j) {
                Entry e = oldTab[j]; //访问old表指定位置的data
                if (e != null) { //data存在
                    ThreadLocal<?> k = e.get();
                    if (k == null) { //过期数据
                        e.value = null; // Help the GC
                    } else {
                        int h = k.threadLocalHashCode & (newLen - 1);//重新计算
hash值
                        while (newTab[h] != null) //获取到一个最近的，可以使用的位置
                            h = nextIndex(h, newLen);
                        newTab[h] = e; //数据迁移
                        count++;
                    }
                }
            }

            setThreshold(newLen);//设置下一次扩容的指标
            size = count;
            table = newTab;
        }
```