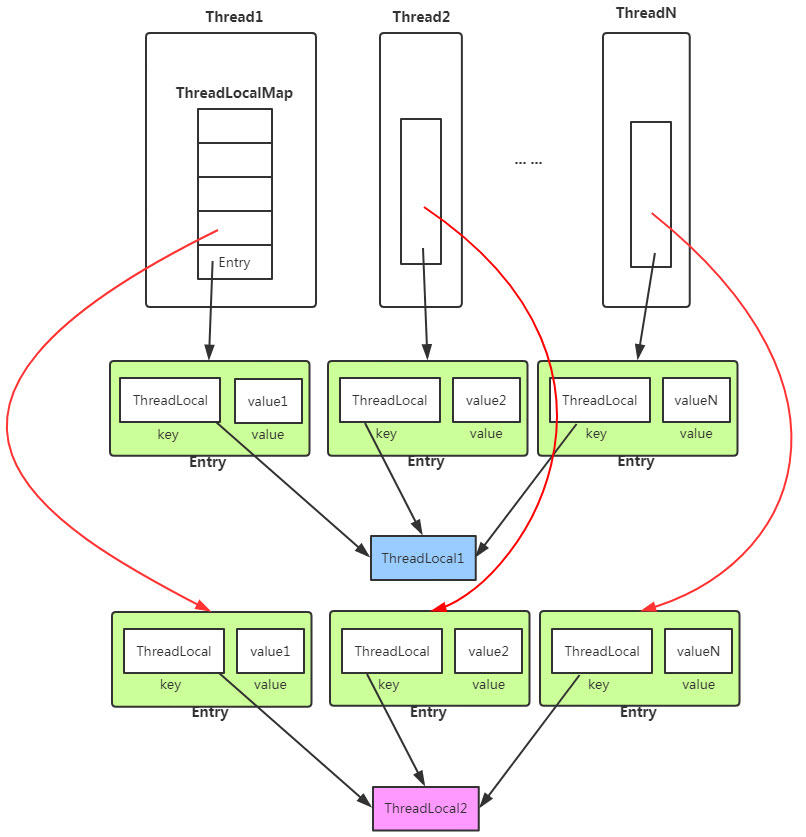
**ThreadLocal是什么**

ThreadLocal是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，在高并发场景下，可以实现无状态的调用，特别适用于各个线程依赖不通的变量值完成操作的场景。

**从数据结构入手**

**下图为ThreadLocal的内部结构图**



ThreadLocal结构内部

从上面的结构图，我们已经窥见ThreadLocal的核心机制：

* 每个Thread线程内部都有一个Map。
* Map里面存储线程本地对象（key）和线程的变量副本（value）
* 但是，Thread内部的Map是由ThreadLocal维护的，由ThreadLocal负责向map获取和设置线程的变量值。

所以对于不同的线程，每次获取副本值时，别的线程并不能获取到当前线程的副本值，形成了副本的隔离，互不干扰。

Thread线程内部的Map在类中描述如下：

public class Thread implements Runnable {

/\* ThreadLocal values pertaining to this thread. This map is maintained

\* by the ThreadLocal class. \*/

ThreadLocal.ThreadLocalMap threadLocals = null;

}

**深入解析ThreadLocal**

ThreadLocal类提供如下几个核心方法：

public T get()

public void set(T value)

public void remove()

* get()方法用于获取当前线程的副本变量值。
* set()方法用于保存当前线程的副本变量值。
* initialValue()为当前线程初始副本变量值。
* remove()方法移除当前前程的副本变量值。

**get()方法**

/\*\*

\* Returns the value in the current thread's copy of this

\* thread-local variable. If the variable has no value for the

\* current thread, it is first initialized to the value returned

\* by an invocation of the {@link #initialValue} method.

\*

\* @return the current thread's value of this thread-local

\*/

public T get() {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null) {

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

if (e != null)

return (T)e.value;

}

return setInitialValue();

}

ThreadLocalMap getMap(Thread t) {

return t.threadLocals;

}

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;

}

步骤：  
1.获取当前线程的ThreadLocalMap对象threadLocals  
2.从map中获取线程存储的K-V Entry节点。  
3.从Entry节点获取存储的Value副本值返回。  
4.map为空的话返回初始值null，即线程变量副本为null，在使用时需要注意判断NullPointerException。

**set()方法**

/\*\*

\* Sets the current thread's copy of this thread-local variable

\* to the specified value. Most subclasses will have no need to

\* override this method, relying solely on the {@link #initialValue}

\* method to set the values of thread-locals.

\*

\* @param value the value to be stored in the current thread's copy of

\* this thread-local.

\*/

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);

}

步骤：  
1.获取当前线程的成员变量map  
2.map非空，则重新将ThreadLocal和新的value副本放入到map中。  
3.map空，则对线程的成员变量ThreadLocalMap进行初始化创建，并将ThreadLocal和value副本放入map中。

**remove()方法**

/\*\*

\* Removes the current thread's value for this thread-local

\* variable. If this thread-local variable is subsequently

\* {@linkplain #get read} by the current thread, its value will be

\* reinitialized by invoking its {@link #initialValue} method,

\* unless its value is {@linkplain #set set} by the current thread

\* in the interim. This may result in multiple invocations of the

\* <tt>initialValue</tt> method in the current thread.

\*

\* @since 1.5

\*/

public void remove() {

ThreadLocalMap m = getMap(Thread.currentThread());

if (m != null)

m.remove(this);

}

ThreadLocalMap getMap(Thread t) {

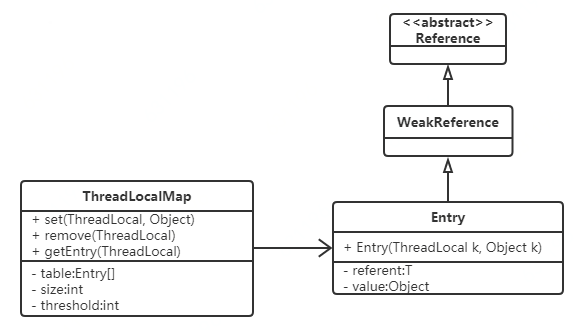
return t.threadLocals;

}

remove方法比较简单，不做赘述。

**ThreadLocalMap**

ThreadLocalMap是ThreadLocal的内部类，没有实现Map接口，用独立的方式实现了Map的功能，其内部的Entry也独立实现。



ThreadLocalMap类图

在ThreadLocalMap中，也是用Entry来保存K-V结构数据的。但是Entry中key只能是ThreadLocal对象，这点被Entry的构造方法已经限定死了。

static class Entry extends WeakReference<ThreadLocal> {

/\*\* The value associated with this ThreadLocal. \*/

Object value;

Entry(ThreadLocal k, Object v) {

super(k);

value = v;

}

}

Entry继承自WeakReference（弱引用，生命周期只能存活到下次GC前），但只有Key是弱引用类型的，Value并非弱引用。

ThreadLocalMap的成员变量：

static class ThreadLocalMap {

/\*\*

\* 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;

/\*\*

\* The number of entries in the table.

\*/

private int size = 0;

/\*\*

\* The next size value at which to resize.

\*/

private int threshold; // Default to 0

}

**Hash冲突怎么解决**

和HashMap的最大的不同在于，ThreadLocalMap结构非常简单，没有next引用，也就是说ThreadLocalMap中解决Hash冲突的方式并非链表的方式，而是采用线性探测的方式，所谓线性探测，就是根据初始key的hashcode值确定元素在table数组中的位置，如果发现这个位置上已经有其他key值的元素被占用，则利用固定的算法寻找一定步长的下个位置，依次判断，直至找到能够存放的位置。

ThreadLocalMap解决Hash冲突的方式就是简单的步长加1或减1，寻找下一个相邻的位置。

/\*\*

\* Increment i modulo len.

\*/

private static int nextIndex(int i, int len) {

return ((i + 1 < len) ? i + 1 : 0);

}

/\*\*

\* Decrement i modulo len.

\*/

private static int prevIndex(int i, int len) {

return ((i - 1 >= 0) ? i - 1 : len - 1);

}

显然ThreadLocalMap采用线性探测的方式解决Hash冲突的效率很低，如果有大量不同的ThreadLocal对象放入map中时发送冲突，或者发生二次冲突，则效率很低。

**所以这里引出的良好建议是：每个线程只存一个变量，这样的话所有的线程存放到map中的Key都是相同的ThreadLocal，如果一个线程要保存多个变量，就需要创建多个ThreadLocal，多个ThreadLocal放入Map中时会极大的增加Hash冲突的可能。**

**ThreadLocalMap的问题**

由于ThreadLocalMap的key是弱引用，而Value是强引用。这就导致了一个问题，ThreadLocal在没有外部对象强引用时，发生GC时弱引用Key会被回收，而Value不会回收，如果创建ThreadLocal的线程一直持续运行，那么这个Entry对象中的value就有可能一直得不到回收，发生内存泄露。

**如何避免泄漏**  
既然Key是弱引用，那么我们要做的事，就是在调用ThreadLocal的get()、set()方法时完成后再调用remove方法，将Entry节点和Map的引用关系移除，这样整个Entry对象在GC Roots分析后就变成不可达了，下次GC的时候就可以被回收。

如果使用ThreadLocal的set方法之后，没有显示的调用remove方法，就有可能发生内存泄露，所以养成良好的编程习惯十分重要，使用完ThreadLocal之后，记得调用remove方法。

ThreadLocal<Session> threadLocal = new ThreadLocal<Session>();

try {

threadLocal.set(new Session(1, "Misout的博客"));

// 其它业务逻辑

} finally {

threadLocal.remove();

}

**应用场景**

还记得Hibernate的session获取场景吗？

private static final ThreadLocal<Session> threadLocal = new ThreadLocal<Session>();

//获取Session

public static Session getCurrentSession(){

Session session = threadLocal.get();

//判断Session是否为空，如果为空，将创建一个session，并设置到本地线程变量中

try {

if(session ==null&&!session.isOpen()){

if(sessionFactory==null){

rbuildSessionFactory();// 创建Hibernate的SessionFactory

}else{

session = sessionFactory.openSession();

}

}

threadLocal.set(session);

} catch (Exception e) {

// TODO: handle exception

}

return session;

}

为什么？每个线程访问数据库都应当是一个独立的Session会话，如果多个线程共享同一个Session会话，有可能其他线程关闭连接了，当前线程再执行提交时就会出现会话已关闭的异常，导致系统异常。此方式能避免线程争抢Session，提高并发下的安全性。

使用ThreadLocal的典型场景正如上面的数据库连接管理，线程会话管理等场景，只适用于独立变量副本的情况，如果变量为全局共享的，则不适用在高并发下使用。

### 总结

* 每个ThreadLocal只能保存一个变量副本，如果想要上线一个线程能够保存多个副本以上，就需要创建多个ThreadLocal。
* ThreadLocal内部的ThreadLocalMap键为弱引用，会有内存泄漏的风险。
* 适用于无状态，副本变量独立后不影响业务逻辑的高并发场景。如果如果业务逻辑强依赖于副本变量，则不适合用ThreadLocal解决，需要另寻解决方案。