1. **用法**

**连接地址：https://www.jianshu.com/p/*1e4eaedbac19***

JDK 1.2的版本中就提供java.lang.ThreadLocal。ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序，ThreadLocal并不是一个Thread，而是Thread的局部变量。

ThreadLocal用于保存某个线程共享变量：对于同一个static ThreadLocal，其为每个使用该变量的线程提供独立的变量副本，不同线程只能从中get，set，remove自己的变量，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。从线程的角度看，目标变量就象是线程的本地变量，这也是类名中“Local”所要表达的意思。

ThreadLocal类接口很简单，只有4个方法：

1. void set(Object value)设置当前线程的线程局部变量的值。
2. public Object get()该方法返回当前线程所对应的线程局部变量。
3. public void remove()将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。
4. protected Object initialValue()返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

值得一提的是，在JDK5.0中，ThreadLocal已经支持泛型，该类的类名已经变为ThreadLocal<T>。API方法也相应进行了调整，新版本的API方法分别是void set(T value)、T get()以及T initialValue()。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单：在ThreadLocal类中有一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值对应线程的变量副本。

/\*\*

\* ThreadLocal用法

\*

\* @author xxp

\*/

public class MyThreadLocal {

private static final ThreadLocal<Object> threadLocal = new ThreadLocal<Object>() {

/\*\*

\* ThreadLocal没有被当前线程赋值时或当前线程刚调用remove方法后调用get方法，返回此方法值

\*/

@Override

protected Object initialValue() {

System.out.println("调用get方法时，当前线程共享变量没有设置，调用initialValue获取默认值！");

return null;

}

};

public static void main(String[] args) {

new Thread(new MyIntegerTask("IntegerTask1")).start();

new Thread(new MyStringTask("StringTask1")).start();

new Thread(new MyIntegerTask("IntegerTask2")).start();

new Thread(new MyStringTask("StringTask2")).start();

}

public static class MyIntegerTask implements Runnable {

private String name;

MyIntegerTask(String name) {

this.name = name;

}

@Override

public void run() {

for (int i = 0; i < 5; i++) {

// ThreadLocal.get方法获取线程变量

if (null == MyThreadLocal.threadLocal.get()) {

// ThreadLocal.et方法设置线程变量

MyThreadLocal.threadLocal.set(0);

System.out.println("线程" + name + ": 0");

} else {

int num = (Integer) MyThreadLocal.threadLocal.get();

MyThreadLocal.threadLocal.set(num + 1);

System.out.println("线程" + name + ": " + MyThreadLocal.threadLocal.get());

if (i == 3) {

MyThreadLocal.threadLocal.remove();

}

}

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public static class MyStringTask implements Runnable {

private String name;

MyStringTask(String name) {

this.name = name;

}

@Override

public void run() {

for (int i = 0; i < 5; i++) {

if (null == MyThreadLocal.threadLocal.get()) {

MyThreadLocal.threadLocal.set("a");

System.out.println("线程" + name + ": a");

} else {

String str = (String) MyThreadLocal.threadLocal.get();

MyThreadLocal.threadLocal.set(str + "a");

System.out.println("线程" + name + ": " + MyThreadLocal.threadLocal.get());

}

try {

Thread.sleep(800);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

结果如下：

调用get方法时，当前线程共享变量没有设置，调用initialValue获取默认值！

线程IntegerTask1: 0

调用get方法时，当前线程共享变量没有设置，调用initialValue获取默认值！

调用get方法时，当前线程共享变量没有设置，调用initialValue获取默认值！

线程IntegerTask2: 0

线程StringTask1: a

调用get方法时，当前线程共享变量没有设置，调用initialValue获取默认值！

线程StringTask2: a

线程StringTask1: aa

线程StringTask2: aa

线程IntegerTask1: 1

线程IntegerTask2: 1

线程StringTask1: aaa

线程StringTask2: aaa

线程IntegerTask1: 2

线程IntegerTask2: 2

线程StringTask2: aaaa

线程StringTask1: aaaa

线程IntegerTask1: 3

线程IntegerTask2: 3

线程StringTask1: aaaaa

线程StringTask2: aaaaa

调用get方法时，当前线程共享变量没有设置，调用initialValue获取默认值！

线程IntegerTask1: 0

调用get方法时，当前线程共享变量没有设置，调用initialValue获取默认值！

线程IntegerTask2: 0

**二、原理**

**线程共享变量缓存如下：**

Thread.ThreadLocalMap<ThreadLocal, Object>;

1. Thread: 当前线程，可以通过Thread.currentThread()获取。
2. ThreadLocal：我们的static ThreadLocal变量。
3. Object: 当前线程共享变量。

我们调用ThreadLocal.get方法时，实际上是从当前线程中获取ThreadLocalMap<ThreadLocal, Object>，然后根据当前ThreadLocal获取当前线程共享变量Object。

ThreadLocal.set，ThreadLocal.remove实际上是同样的道理。

**这种存储结构的好处：**

1. 线程死去的时候，线程共享变量ThreadLocalMap则销毁。
2. ThreadLocalMap<ThreadLocal,Object>键值对数量为ThreadLocal的数量，一般来说ThreadLocal数量很少，相比在ThreadLocal中用Map<Thread, Object>键值对存储线程共享变量（Thread数量一般来说比ThreadLocal数量多），性能提高很多。

**关于ThreadLocalMap<ThreadLocal, Object>弱引用问题：**

当线程没有结束，但是ThreadLocal已经被回收，则可能导致线程中存在ThreadLocalMap<null, Object>的键值对，造成内存泄露。（ThreadLocal被回收，ThreadLocal关联的线程共享变量还存在）。

虽然ThreadLocal的get，set方法可以清除ThreadLocalMap中key为null的value，但是get，set方法在内存泄露后并不会必然调用，所以为了防止此类情况的出现，我们有两种手段。

1. 使用完线程共享变量后，显示调用ThreadLocalMap.remove方法清除线程共享变量；
2. JDK建议ThreadLocal定义为private static，这样ThreadLocal的弱引用问题则不存在了