# ThreadLocal原理分析与使用场景

## 什么是ThreadLocal变量

ThreadLoal 变量，线程局部变量，同一个 ThreadLocal 所包含的对象，在不同的 Thread 中有不同的副本。这里有几点需要注意：

因为每个 Thread 内有自己的实例副本，且该副本只能由当前 Thread 使用。这是也是 ThreadLocal 命名的由来。既然每个 Thread 有自己的实例副本，且其它 Thread 不可访问，那就不存在多线程间共享的问题。

ThreadLocal 提供了线程本地的实例。**它与普通变量的区别在于，每个使用该变量的线程都会初始化一个完全独立的实例副本。**ThreadLocal 变量通常被private static修饰。当一个线程结束时，它所使用的所有 ThreadLocal 相对的实例副本都可被回收。

总的来说，ThreadLocal 适用于每个线程需要自己独立的实例且该实例需要在多个方法中被使用，也即变量在线程间隔离而在方法或类间共享的场景。

## ThreadLocal实现原理

首先 ThreadLocal 是一个泛型类，保证可以接受任何类型的对象。

因为一个线程内可以存在多个 ThreadLocal 对象，所以其实是 ThreadLocal 内部维护了一个 Map ，这个 Map 不是直接使用的 HashMap ，而是 ThreadLocal 实现的一个叫做 **ThreadLocalMap** 的静态内部类。而我们使用的 get()、set() 方法其实都是调用了这个ThreadLocalMap类对应的 get()、set() 方法。

**set 方法：**

public void set(T value) {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

map.set(this, value);

else

createMap(t, value);

}

**get方法：**

public T get() {

Thread t = Thread.currentThread();

ThreadLocalMap map = getMap(t);

if (map != null)

return (T)map.get(this);

// Maps are constructed lazily. if the map for this thread

// doesn't exist, create it, with this ThreadLocal and its

// initial value as its only entry.

T value = initialValue();

createMap(t, value);

return value;

}

**createMap方法：**

void createMap(Thread t, T firstValue) {

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

}

ThreadLocalMap是个静态的内部类：

static class ThreadLocalMap {

........

}

最终的变量是放在了当前线程的 ThreadLocalMap 中，并不是存在 ThreadLocal 上，ThreadLocal 可以理解为只是ThreadLocalMap的封装，传递了变量值。

## 内存泄漏问题

实际上 ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用，弱引用的特点是，如果这个对象只存在弱引用，那么在下一次垃圾回收的时候必然会被清理掉。

所以如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候会被清理掉的，这样一来 ThreadLocalMap中使用这个 ThreadLocal 的 key 也会被清理掉。但是，value 是强引用，不会被清理，这样一来就会出现 key 为 null 的 value。

*ThreadLocalMap实现中已经考虑了这种情况，在调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录。*如果说会出现内存泄漏，那只有在出现了 key 为 null 的记录后，没有手动调用 remove() 方法，并且之后也不再调用 get()、set()、remove() 方法的情况下。

## 使用场景

如上文所述，ThreadLocal 适用于如下两种场景：

1、每个线程需要有自己单独的实例

2、实例需要在多个方法中共享，但不希望被多线程共享

对于第一点，每个线程拥有自己实例，实现它的方式很多。例如可以在线程内部构建一个单独的实例。ThreadLocal 可以以非常方便的形式满足该需求。

对于第二点，可以在满足第一点（每个线程有自己的实例）的条件下，通过方法间引用传递的形式实现。ThreadLocal 使得代码耦合度更低，且实现更优雅。

## 扩展：来探讨一下最近面试问的ThreadLocal问题

### 1）存储用户Session

一个简单的用ThreadLocal来存储Session的例子：

private static final ThreadLocal threadSession = new ThreadLocal();

public static Session getSession() throws InfrastructureException {

Session s = (Session) threadSession.get();

try {

if (s == null) {

s = getSessionFactory().openSession();

threadSession.set(s);

}

} catch (HibernateException ex) {

throw new InfrastructureException(ex);

}

return s;

}

### 2）解决线程安全的问题

比如Java7中的SimpleDateFormat不是线程安全的，可以用ThreadLocal来解决这个问题：

public class DateUtil {

private static ThreadLocal<SimpleDateFormat> format1 = new ThreadLocal<SimpleDateFormat>() {

@Override

protected SimpleDateFormat initialValue() {

return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

}

};

public static String formatDate(Date date) {

return format1.get().format(date);

}

}

这里的DateUtil.formatDate()就是线程安全的了。(Java8里的 java.time.format.DateTimeFormatter是线程安全的，Joda time里的DateTimeFormat也是线程安全的）。