ThreadLocal介绍

# ThreadLocal的功能

为每一个线程创建一个变量的副本。

# ThreadLocal类介绍

## 继承关系

public class **ThreadLocal**<T> extends Object

Direct Known Subclasses: **InheritableThreadLocal**



## 功能

This class provides **thread-local variables**. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

当使用**ThreadLocal维护变量**时，ThreadLocal为每个使用**该变量的线程**提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其他线程的所对应的副本。

从**线程**角度看，**目标变量就像是线程的本地变量**，这也是类名中**Local**所要表达的意思。

## 构造方法

ThreadLocal() Creates a thread local variable.

## 方法介绍

### protected T **initialValue**()：返回当前线程的thread-local变量的初始值。

该方法是一个protected方法，显然是为了让子类覆盖而设计的。

**该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，并且仅执行一次**。ThreadLocal中的缺省实现直接返回一个null。

**Returns the current thread's "initial value" for this thread-local variable**. This method will be invoked the first time a thread accesses the variable with the **get()** method, unless the thread previously invoked the **set(T) method**, in which case the initialValue method will not be invoked for the thread. Normally, this method is invoked at most once per thread, but it may be invoked again in case of subsequent invocations of **remove()** followed by **get().**

This implementation simply returns null; if the programmer desires thread-local variables to have an initial value other than null, ThreadLocal must be subclassed, and this method overridden. Typically, an **anonymous** inner class will be used.

Returns: **the initial value for this thread-local**

### **public T get()**

获取当前线程所对应的线程局部变量。

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 **initialValue()** method.

Returns: the current thread's value of this thread-local

### public void set(T value)

**设置当前线程的线程局部变量的值。**

**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 initialValue() method to set the values of thread-locals.

Parameters: value - the value to be stored in the current thread's copy of this thread-local.

### public void **remove**() 将当前线程的局部变量的值删除。

目的是：减少内存的占用。这是JDK1.5新增加的方法。

注意：**当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显示调用该方法清除线程的局部变量并不是必须的操作，但是它可以加快内存回收的速度。**

Removes the current thread's value for this thread-local variable. If this thread-local variable is subsequently read by the current thread, its value will be reinitialized by invoking its **initialValue()** method, unless its value is set by the current thread in the interim. This may result in multiple invocations of the initialValue method in the current thread.

Since:1.5

### public static <S> **ThreadLocal**<S> **withInitial**(Supplier<? extends S> supplier)

**Creates a thread local variable.** The initial value of the variable is determined by invoking the get method on the Supplier.

Type Parameters:S - the type of the thread local's value

Parameters: supplier - the supplier to be used to determine the initial value

Returns:a new thread local variable

Throws:NullPointerException - if the specified supplier is null

**Since:1.8**

## 使用示例

For example, the class below **generates unique identifiers local to** **each thread**. A thread's id is assigned the first time it invokes **ThreadId.get()** and remains unchanged on subsequent calls.

import java.util.concurrent.**atomic**.**AtomicInteger**;

**public class ThreadId** {

// Atomic integer containing the next thread ID to be assigned

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

// Thread local variable containing each thread's ID

private static final ThreadLocal<Integer> threadId =

new ThreadLocal<Integer>() {

**@Override** protected Integer **initialValue**() {//重载该方法

return nextId.**getAndIncrement**();

}

};

// Returns the current thread's unique ID, assigning it if necessary

public static int get() {

return threadId.get();

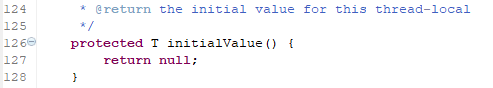
}

}

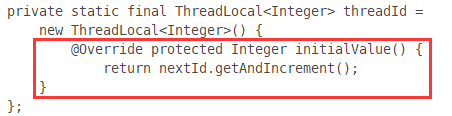
Each thread holds **an implicit reference** to its copy of a thread-local variable as long as the thread is alive and the **ThreadLocal** instance is accessible; after a thread goes away, all of its copies of thread-local instances are subject to **garbage collection** (unless other references to these copies exist).

# ThreadLocal的实现原理及源码分析

## initialValue()方法



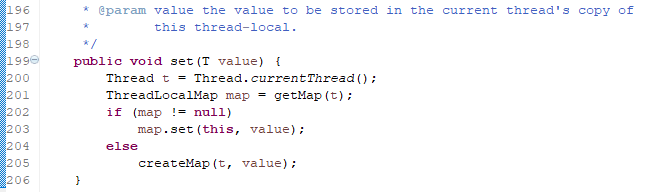
由于initialValue该方法缺省下返回**null**，所以一般创建ThreadLocal实例的时候需要覆盖initialValue该方法。

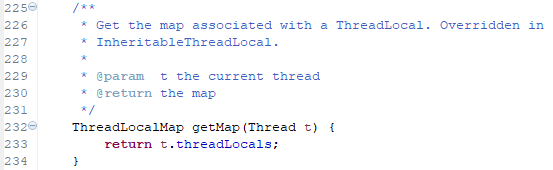


## ThreadLocal到底是如何实现这种“为每个线程提供不同的变量拷贝”的呢？

**ThreadLocal的源码：**

set方法源代码：



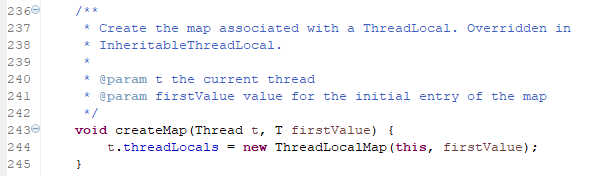


注意：Thread对象的**threadLocals属性**，只能被ThreadLocal所拥有，外界无法获取；每一个线程对应着唯一的**ThreadLocal.ThreadLocalMap对象**。在每个**ThreadLocal.ThreadLocalMap对象中，都是以ThreadLocal对象作为键，存放当前线程的变量副本。**

**区分：ThreadLocalMap类定义在ThreadLocal类中，但是是每一个Tread队拥有一个ThreadLocal对象即静态属性threadLocals。**

**ThreadLocalMap中的键：是ThreadLocal对象；值：是每个线程对应的值。**

**这样的话，编程人员只需要创建一次ThreadLocal变量，但是该变量在不同的线程内部会创建对应的部分。**



注意:ThreadLocalMap

解析：set方法内部首先通过**getMap(Thread t)**获取一个和当前线程相关的ThreadLocalMap，然后将变量的值设置到这个ThreadLocalMap对象中，若没有ThreadLocalMap对象，则利用createMap创建。

线程隔离的秘密就在于**ThreadLocalMap**这个类。

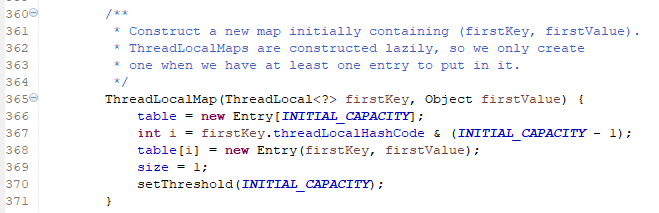
**ThreadLocalMap是ThreadLocal类的一个静态内部类，它实现了键值对的设置与获取。每个线程中都有一个独立的ThreadLocalMap副本，它所存储的值只能被当前线程读取和修改。ThreadLocal类通过操作每个线程独有的ThreadLocalMap副本，从而实现了变量在不同线程中的隔离。因为每个线程的变量都是自己特有的，完全不会有并发错误。**

**还有一点就是：ThreadLocalMap存储的键值对中的键是this对象指向的ThreadLocal对象，而值就是你所设置的对象了。**

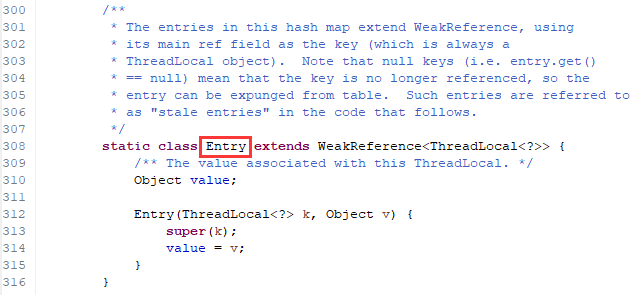
## ThreadLocalMap静态内部类

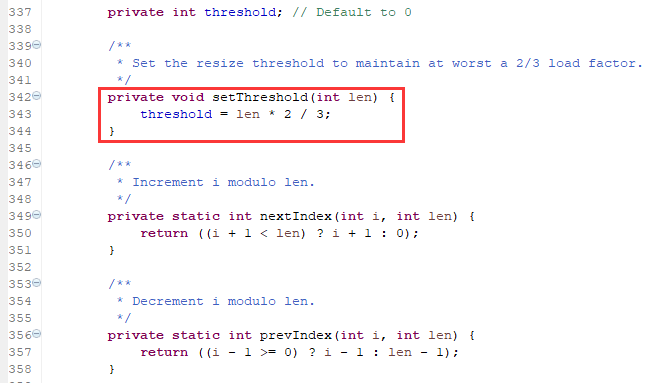
static class **ThreadLocalMap**

**构造方法:**

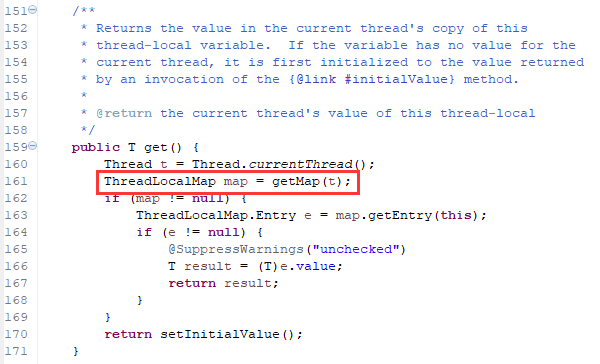


ThreadLocalMap内部还有一个内部类Entry：



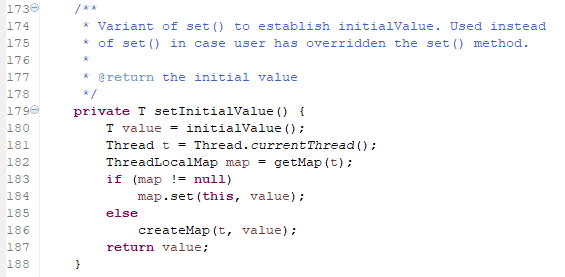


## get()源码



当没有获取到值时，通过setInitialValue方法返回Value。

获取和当前线程绑定的值时，ThreadLocalMap对象是以this指向的ThreadLocal对象为键进行查找的，和**set()方法**的代码是相呼应的。



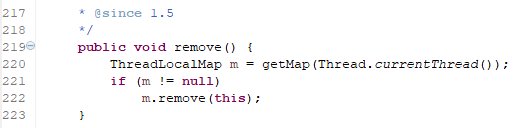
由于ThreadLocal对象肯定是在某一个线程中创建的，此时其他线程中没有对应的该ThreadLocal对象，此时可能其他线程还没有初始化threadLocals属性，即没有ThreadLocaMap对象，或有了ThreadLocalMap对象，但是从该线程的ThreadLocalMap中获取不到对应的ThreadLocal值，怎么办呢？

上面的第一点：线程没有初始化ThreadLocalMap对象，对应的方法就是**setInitialValue**方法，通过createMap创建ThreadLocalMap对象，并把ThreadLocal对象作为键，value为值，存到Map中；另外一种情况是，存在ThreadLocalMap，但是该Map中没有ThreadLocal—value的键值对，此时需要通过initialValue方法获取默认值，然后向Map中添加该对应关系。

也就是说，在一个线程创建ThreadLocal的时候,只有该线程拥有ThreadLocal---value的对应关系，其他线程中并没有；只有其他线程调用set方法或get方法的时候，才会将该对应关系添加到线程自己的ThreadLocalMap中。但是记住关键一点：所有的线程的ThreadLocal值的起点都是initialValue，完全隔离的，但是ThreadLocal的inititalValue始终不变的。

## remove()方法源码

删除也只是删除本线程中的对应关系，不影响其他线程，同时，如果再次调用get方法的时候，返回的是该ThreadLocal对象的initialValue。



## 多个变量

**可以创建不同的ThreadLocal实例来实现多个变量在不同线程间的访问隔离。**

为什么可以这么做？

因为不同的ThreadLocal对象作为不同的键，当然也可以在线程的ThreadLocalMap对象中设置不同的值了。

通过ThreadLocal对象，在多线程中共享一个值和多个值的区别，就像在一个HashMap对象中存储一个键值对和多个键值对一样，仅此而已。

每个线程对应着唯一的ThreadLocal.ThreadLocalMap对象，ThreadLocal.ThreadLocalMap中以ThreadLocal对象为键存放ThreadLocal对应的变量副本。**多个ThreadLocal就对应多个变量，在ThreadLocalMap中就对应着多个键值对。**

**如果只定义一个ThreadLocal对象，那么线程对应的ThreadLocalMap中就只有一个键值对；如果创建多个ThreadLocal对象，那么线程的ThreadLocalMap中就有多个键值对。**

## 利用ThreadLocal实现线程安全

利用ThreadLocal处理线程的局部变量的时候比后面要讲的synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

## 容易内存泄露

使用ThreadLocal一般都是声明在静态变量中，如果不断地创建ThreadLocal而且没有调用remove方法，将会导致内存泄露，特别是在高并发的Web容器当中这么做的时候。

注意：当线程结束后，对应该线程的局部变量将自动被垃圾回收，但是如果线程没有结束，**还是需要显示调用remove删除变量的**。

# ThreadLocal中的引用

//TO-DO

弱引用---

如何使用软引用、弱引用、虚引用。

<https://blog.csdn.net/xiaofengcanyuexj/article/details/45271195>

# ThreadLocal多个变量示例

public class ThreadLocalTest {

public static void main(String[] args) {

//一个ThreadLocal对应一个变量，几个变量对应几个ThreadLocal对象

//第一个变量

**ThreadLocal<Integer> tl = new ThreadLocal<Integer>(){//共享变量**

**@Override**

**protected Integer initialValue(){//重载方法，缺省返回null**

**return 0;**

**}**

**};**

//第二个变量

**ThreadLocal<String> tl2 = new ThreadLocal<String>(){//共享变量**

**@Override**

**protected String initialValue(){//重载方法，缺省返回null**

**return "A";**

**}**

**};**

Runnable target = new Runnable() {

@Override

public void run() {

for(int i = 0;i<3;i++){

tl.set(tl.get()+1);

tl2.set(tl2.get()+"--"+i); **System.out.println(Thread.currentThread().getName()+":"+tl.get()+";"+tl2.get());**

}

}

};

Thread t1 = new Thread(target);

Thread t2 = new Thread(target);

Thread t3 = new Thread(target);

Thread t4 = new Thread(target);

t1.start(); t2.start(); t3.start(); t4.start();

}

}

结果：

**Thread-2:1;A--0**

Thread-1:1;A--0

**Thread-3:1;A--0**

Thread-1:2;A--0--1

Thread-0:1;A--0

Thread-0:2;A--0--1

Thread-0:3;A--0--1--2

Thread-2:2;A--0--1

Thread-1:3;A--0--1--2

**Thread-3:2;A--0--1**

Thread-2:3;A--0--1--2

**Thread-3:3;A--0--1--2**