

作者: JavaGieGie

微信公众号: Java开发零到壹

前言

蹲坑多线程系列 前面已经更新过六个章节了，强烈建议小伙伴按照顺序学习：

我：狗剩子，早上好啊，两天不见，还以为你跳槽进了阿三的公司呢！

狗剩子：你这是吃辣子放屁，带刺激性的，花Gie教程不看完，我跳什么槽。

我：哟呼，那你这两天去哪了？

狗剩子：我这不是闭关修炼，刚刚出关嘛！

我：.... 分析完毕，是条傻狗。



正文

我：狗哥狗哥，你可以和我说说 ThreadLocal是什么吗？上次阿香问我，我没回答上来，很尴尬啊？

不要怕，狗哥来教你。

ThreadLocal是一个数据结构，有点类似HashMap，主要用来存储线程局部变量，该变量只对当前线程可见。简单说就是在每个线程中为变量都创建了一个副本，当线程需要用到该变量时，可以通过 `set()` 和 `get()` 来对该副本进行操作，从而实现线程之间的数据隔离。

我：听起来也不是很难嘛！那我们平时都是在哪用到ThreadLocal 呢？

听起来虽然很简单，但是要想真正的掌握，也不是那么简单的事哦。倒杯茶，点个赞，且听老衲细细道来，下面举几个常用的案例。



1) SimpleDateFormat时间格式转换：

狗哥敢保证，100%的小伙伴都用过 `SimpleDateFormat` 来格式化时间，一般我们会封装一个工具类，下面这个例子，用来打印 1 到 1000 秒的格式化时间。

```
public class ThreadLocalDemo {
    //创建一个固定大小的线程池
    public static ExecutorService executorService =
        Executors.newFixedThreadPool(5);

    public static void main(String[] args) {

        for (int i = 0; i <= 1000; i++) {
            int time = i ;
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    //调用时间类静态方法
                    String formatTime = TimeUtil.getSecond(time);
                    System.out.println(formatTime);
                }
            });
        }
        executorService.shutdown();
    }
}


//时间工具类
class TimeUtil {
    static SimpleDateFormat sdf = new SimpleDateFormat("YYYY-MM-DD HH:mm:ss");

    public static String getSecond(int count){
        Date date = new Date(1000*count);
        return sdf.format(date);
    }
}
```

submit: 添加线程任务; shutdown: 停止线程池;

线程池这里先进行简单理解，就是始终维护几个线程保持运行，等到有任务到来时会立即处理，若任务太多来不及处理时，就会将其塞入队列，等到正在执行的任务结束后，再继续执行。

这时候我们会发现诡异的一幕，`time` 变量明明是 0-1000 的自然数，并且不会出现重复数字，这里为什么会打印出相同的格式化结果呢。是不是想起以前写过的那些方法，顿感脊背发凉呢。



```
1970-01-01 08:15:11
1970-01-01 08:15:09
1970-01-01 08:15:08
1970-01-01 08:15:08
1970-01-01 08:15:05
```

出现这种情况的原因，是因为所有的线程都共用同一个 `SimpleDateFormat` 对象，发生了线程安全问题，聪明的小伙伴肯定可以想到，我们可以用 `synchronized` 来锁住关键代码，不就可以保证结果线程安全了嘛。

```

public static String getSecond(int count){
    Date date = new Date(1000*count);
    String result = null;
    synchronized(ThreadLocalDemo.class){
        result = sdf.format(date);
    }
    return result;
}

```



这样是能够实现我们的目的，结果中也不会出现意外了，但是这样做会导致同一时刻只能有一个线程执行时间格式化（执行串行化），严重影响程序的性能，在高并发情况下坚决不能忍。

这时候我们的主角 `ThreadLocal` 就要闪亮登场了，这里直接看代码，`main`方法不变，只修改工具类。

```

class TimeUtil {
    public static String getSecond(int count){
        Date date = new Date(1000*count);
        //使用get获取 SimpleDateFormat
        SimpleDateFormat sdf =
        ThreadLocalUtils.simpleDateFormatThreadLocal.get();
        return sdf.format(date);
    }
}

//创建 ThreadLocal
class ThreadLocalUtils {
    public static ThreadLocal<SimpleDateFormat> simpleDateFormatThreadLocal = new
    ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue() {
            return new SimpleDateFormat("YYYY-MM-DD HH:mm:ss");
        }
    };
}

```

这时候无论怎么执行，都不会出现相同的结果，说明用 `ThreadLocal` 创建的对象能够保证线程线程。

避免一些参数传递

昨天618不知道大家有没有剁手，反正我是无手可剁了，只想好好活着。这里我们思考一个问题，在你下单时，后台程序是怎么处理的呢？事实上，后台程序需要处理包括用户信息查询、优惠券查询、收货地址查询、消息通知等等在内的很多流程。因为每一个步骤都可能会用到用户信息，如果把用户信息当做参数层层传递，这样会导致代码耦合性较高且十分臃肿，非常不利于维护。

有些小伙伴可能会想到，我就不用你的 `ThreadLocal`，我写一个静态的 `map`集合存储，不就可以保存了吗？



我可真skr小机灵鬼儿

©掘金技术社区

在多线程访问同一个变量的情况下，我们知道这会出现线程安全问题，如果使用线程安全类型的集合（如 `ConcurrentHashMap`）或者 直接加锁，都会影响程序的执行性能，和上面使用 `synchronized` 代码块 修饰导致的性能问题是一样的。

因此我们可以总结一下：**在线程的生命周期内，使用 `ThreadLocal` 的 `set()` 方法可以存储该线程的私有变量，并且在需要该变量时通过 `get()` 进行获取。该变量在不同线程中内容是独立的，这样在不损耗性能的情况下，避免了参数多级传递的麻烦。**

```
public class ThreadLocalDemo2 {
    public static ExecutorService executorService =
        Executors.newFixedThreadPool(10);

    public static void main(String[] args) {
        User user = new User("花Gie");
        ThreadLocalInfo.userThreadLocal.set(user);
        //1. 调用获取地址方法
        new AddressService().getAddress();
    }
}

class AddressService{
    public void getAddress(){
        User user = ThreadLocalInfo.userThreadLocal.get();
        System.out.println("根据用户信息"+user.getUserName()+"获取用户地址");
        //2. 调用优惠券方法
    }
}
```

```

        new TicketService().getTicket();
    }
}

class TicketService{
    public void getTicket(){
        User user = ThreadLocalInfo.userThreadLocal.get();
        System.out.println("根据用户信息"+user.getUserName()+"获取用户优惠券");
        //3.调用发送消息
        new MessageService().sendMessage();
    }
}

class MessageService{
    public void sendMessage(){
        User user = ThreadLocalInfo.userThreadLocal.get();
        System.out.println("根据用户"+user.getUserName()+"发送消息");
    }
}

class User {
    String userName;

    public User(String userName) {
        this.userName = userName;
    }

    public String getUserName() {
        return userName;
    }
}

//创建ThreadLocal变量
class ThreadLocalInfo {
    public static ThreadLocal<User> userThreadLocal = new ThreadLocal<>();
}

```

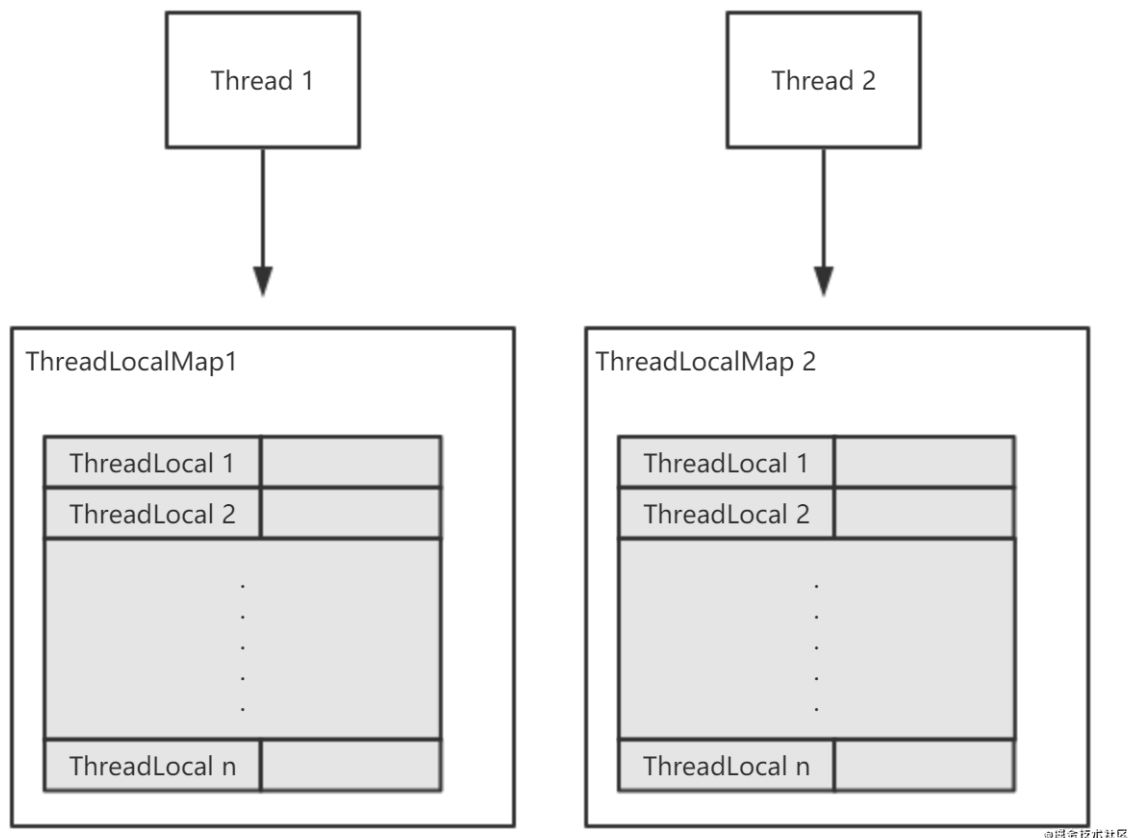
代码的讲解方式真的香，清晰明了，上述代码在获取地址、获取优惠券、发送消息的三个步骤中，我们并没有一层层的传递user对象，但是都能获取到user的对象内容，即使后续需求或流程有调整，我们都可以轻松应对。

根据用户信息花Gie获取用户地址
 根据用户信息花Gie获取用户优惠券
 根据用户花Gie发送消息

@掘金技术社区

我：狗哥狗哥，我现在知道用法和作用了，那他的实现原理可以说一下吗？

好的呀，先看下这张图，大致了解一下这 Thread、ThreadLocal、ThreadLocalMap 三者的关系。



从图中可以清楚的看到，每一个 Thread 都会拥有一个 ThreadLocalMap 对象，而每一个 ThreadLocalMap 又包含多个 ThreadLocal。

我：这是你自己画的图...我怎么知道是不是胡扯的呢!

花Gie几天不见学聪明了啊，那我们就看一下源码呗。ThreadLocal有四个比较重要的方法：

```
public T get() {}  
public void set(T value) {}  
protected T initialValue() {}
```

首先看一下第一种场景中我们重写过的 initialValue()方法：

```
protected T initialValue() {  
    return null;  
}
```

非常明显，如果我们不主动重写 initialValue() 方法的话，它会返回一个null值。

接下来看下set方法：

```

public void set(T value) {
    //1. 获取当前线程
    Thread t = Thread.currentThread();
    //2. 获取ThreadLocalMap对象
    ThreadLocalMap map = getMap(t);
    //3. 如果map存在，就会将当前ThreadLocal对象作为key存储到map中
    //this:就是当前的ThreadLocal
    if (map != null)
        map.set(this, value);
    else
        //4. map不存在时，则创建
        createMap(t, value);
}

```

上面第二步有一个ThreadLocalMap，这个是什么呢，其实在Thread类中我们可以找到结果，他是Thread中的一个内部类。这里看下关键代码，ThreadLocalMap是用键值数组Entry[] table，来存储数据，Entry就可以类比为是一个map，其中键是指ThreadLocal；值则是需要保存的内容，如SimpleDateFormat、user。

ThreadLocalMap就是上图中灰色部分

```

static class ThreadLocalMap {
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }
    private Entry[] table;
    ....省略
}

```

然后看下getMap的内容,他会返回当前线程的threadLocals对象，因此在没有重写initialValue()进行初始化时，首次调用set()方法getMap会返回一个null，然后对其进行createMap操作。

```

ThreadLocalMap getMap(Thread t) {    return t.threadLocals;}// Thread.java //
threadLocals默认为nullThreadLocal.ThreadLocalMap threadLocals = null;

```

createMap操作也是比较简单的，就是用来新建一个ThreadLocalMap对象赋值给当前线程的threadLocals变量。

```

void createMap(Thread t, T firstValue) {    t.threadLocals = new
ThreadLocalMap(this, firstValue);}

```

看完上面的源码，那get()方法就非常容易理解了。首先看第3步代码，如果之前已经调用过set()方法（threadLocals对象没有被初始化），就会将当前ThreadLocal对象作为key，获取threadLocals中事先保存的value值。

```
public T get() {    //1.获取当前线程    Thread t = Thread.currentThread();    //2.
    获取当前线程的threadLocalMap    ThreadLocalMap map = getMap(t);    //3.如果
    ThreadLocal已经初始化    if (map != null) {        ThreadLocalMap.Entry e =
    map.getEntry(this);        if (e != null) {
    @SuppressWarnings("unchecked")        T result = (T)e.value;
    return result;        }    }    //4.ThreadLocal没有初始化, 则调用initialValue()进行初
    始化操作    return setInitialValue();}
```

但是最后一句是做什么的呢，其实这里是用到了延迟加载的方式，我们在重写initialValue()方法时，它并没有立即进行初始化，而是等到第一次查询的时候，才执行setInitialValue()方法进行初始化。

```
private T setInitialValue() {    //重写的initialValue方法    T value =
    initialValue();    Thread t = Thread.currentThread();    ThreadLocalMap map =
    getMap(t);    if (map != null)        map.set(this, value);    else
    createMap(t, value);    return value;}
```

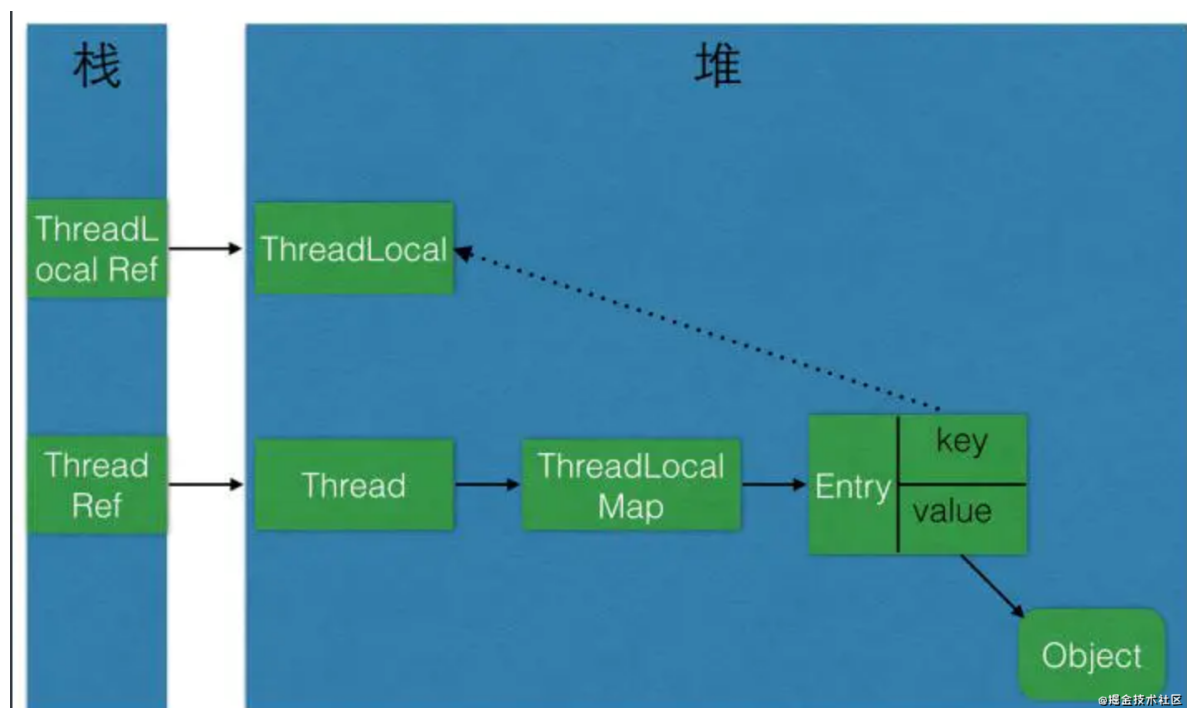
看到这里，是不是基本搞清了图中Thread、ThreadLocal、ThreadLocalMap三者的关系及流程呢。

我：虽然很啰嗦，但是真的很详细呀！

这里想说一下，每次说这么多，我也是手酸、嘴也累啊，但是又十分担心如果忽略了某些细节，很多小伙伴可能就听不懂了，所以比较流弊的小伙伴可以跳着看，忽略一些细节，如果是新手小伙伴，看起来也是更加轻松，更加容易理解。

我：狗子用心了，花Gie在此给你点个赞。那ThreadLocal既然这么多好处，它有什么缺陷吗？

凡是都有两面性，ThreadLocal肯定也有些问题需要注意，就是内存泄漏问题



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

如上述代码，**value = v**表示我们设置到ThreadLocal中的值和ThreadLocalMap是强引用，再结合上图我们可以得到内存泄露的原因：是由于ThreadLocalMap的生命周期跟Thread一样长，如果线程不销毁，那ThreadLocal就会一直存活，如果没有手动删除对应key就会导致内存泄漏。

比如在线程池这种场景下，因为线程始终存活，所以此时使用ThreadLocal的话，就会导致内存泄漏。想要避免内存泄露就要**手动remove()掉**！

我：狗儿你这修炼两天的效果，我都惊呆了，给小伙伴们总结一下吧。

ThreadLocal总结起来有以下几点：

- 在该线程的生命周期中，可以轻松获取该对象；
- 1. 作用：
 - 每个线程可以拥有自己独立的对象，与其他线程实现隔离；
 - 在该线程的生命周期中，可以轻松获取该对象；
- 2. 优点：
 - 不需要加锁就能实现线程安全
 - 高效利用内存，节省开销
 - 无需层层传参，实现代码解耦
- 3. 场景选择：
 - 对于工具类这种，所有线程共用一个实例时，在创建ThreadLocal变量时就将其初始化；
 - 对于不同线程拥有不同对象时，灵活使用set()方法进行设置。
- 4. 原理总结：
 - ThreadLocalMap是Thread的内部类，并且每个Thread维护一个ThreadLocalMap的引用
 - initialValue()和set()设置键值对本质相同，本质都是调用map.set(this,value)
 - ThreadLocal本身并不存储值，它只是作为一个key从ThreadLocalMap获取value。

总结

用ThreadLocal的目的并不是为了解决并发或者共享变量的问题，而是为了能够在当前线程中有属于自己的变量，实现线程的数据隔离。

下期预告

今天我们用到了线程池，可能有些小伙伴并不是很了解，下一章花Gie会介绍线程池的使用。**希望大家持续关注，为了大厂梦，我们继续肝。**

点关注，防走丢

以上就是本期全部内容，**如有纰漏之处，请留言指教，非常感谢。**我是花GieGie，有问题大家随时留言讨论，我们下期见👋。

文章持续更新，可以微信搜一搜 **Java开发零到壹** 第一时间阅读，并且可以获取**面试资料学习视频**等，有兴趣的小伙伴欢迎关注，一起学习，一起哈哈😄。