链滴

ThreadLocal 在 session 管理中的应用与原理

作者：zhengliwei

原文链接：https://ld246.com/article/1604753236492

来源网站：链滴

hello，大家好，欢迎来到银之庭。我是Z，一个普通的程序员。今天我们来看下Java里ThreadLocal关API在web应用场景中的应用：用来管理用户session，以及ThreadLocal的底层实现原理。

1. 背景

在我维护的一个Javaweb应用中，我看到前人写了个拦截器，在请求开始时校验用户登录态，如果没录就返回错误（前端会校验这个错误，引导用户登录），如果登录了，会去公司统一的session服务获取用户信息（session服务内部应该就是把用户登录信息存到redis里了），构造一个session对象，存到本地的ThreadLocal中，然后在业务处理完成后清空ThreadLocal。而在业务处理过程中，就可随时从这个session里取用用户信息了，算是个方便开发的小手段。抽象的代码逻辑大概如下，大家可以从我的git仓库里下载完整项目代码。

注意：以下代码有大量模拟的逻辑，只是为了演示，线上可千万别这么写！！！

拦截器逻辑：

```
package top.zhengliwei.threadLocalTest.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;
import top.zhengliwei.threadLocalTest.model.RequestContext;
import top.zhengliwei.threadLocalTest.model.UserSession;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Obj
ct handler) {
        // 线上是从cookie中取的，这个只是演示
        String token = request.getParameter("token");
        if (!hasLogin(token)) {
            // 线上会用response直接写返回值，这里省略
            return false;
        }

        // 调用户服务拿到用户信息，写入requestContext的userSession里，这里的userSession实际
是个threadLocal变量
        RequestContext.setUserSession(getUserInfo(token, request));
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler,
                           ModelAndView modelAndView) {
        // 处理完成后删除userSession
        RequestContext.deleteUserSession();
    }

    // 根据token校验是否登录，也是调用公司统一的用户登录服务校验的，这里直接返回true
```

```
    public boolean hasLogin(String token) {
        return true;
    }

    // 线上是根据token查询用户服务，这里为了方便直接从请求参数里读取用户信息
    private UserSession getUserInfo(String token, HttpServletRequest request) {
        String userId = request.getParameter("userId");
        String userName = request.getParameter("userName");
        return UserSession.builder().userId(Integer.valueOf(userId)).userName(userName).build();
    }
}
```

注册拦截器：

```
package top.zhengliwei.threadLocalTest.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import top.zhengliwei.threadLocalTest.interceptor.LoginInterceptor;

@Configuration
public class InterceptorConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        InterceptorRegistration registration = registry.addInterceptor(new LoginInterceptor());
        registration.addPathPatterns("/**");
    }
}
```

UserSession类

```
package top.zhengliwei.threadLocalTest.model;

import lombok.Builder;
import lombok.Data;

@Data
@Builder
public class UserSession {
    private Integer userId;
    private String userName;
}
```

RequestContext类

```
package top.zhengliwei.threadLocalTest.model;

public class RequestContext {
    private static final ThreadLocal<UserSession> USER_SESSION = new ThreadLocal<>();
```

```
    public static void setUserSession(UserSession userSession) {
        USER_SESSION.set(userSession);
    }

    public static UserSession getUserSession() {
        return USER_SESSION.get();
    }

    public static void deleteUserSession() {
        USER_SESSION.remove();
    }
}
```

示例接口，在接口中使用userSession:

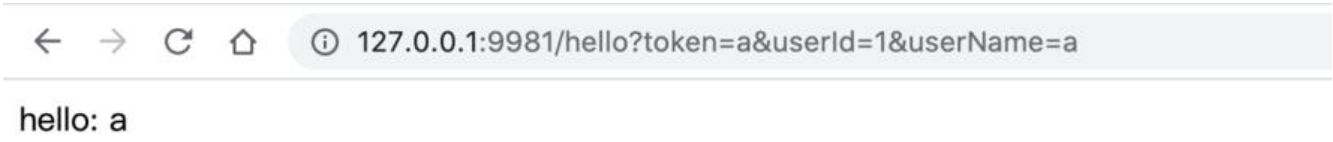
```
package top.zhengliwei.threadLocalTest.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import top.zhengliwei.threadLocalTest.model.RequestContext;

@RestController
@RequestMapping
public class DemoController {

    @RequestMapping("/hello")
    public String sayHello() {
        return "hello: " + RequestContext.getUserSession().getUserName();
    }
}
```

项目跑起来后，访问本地的http://127.0.0.1:9981/hello?token=a&userId=1&userName=a接口正常的话就会调用上面的hello接口，返回hello: a了，如下：



那么问题来了，**为什么要用ThreadLocal对象保存session对象呢？能不能直接用个全局变量？**

2. ThreadLocal的作用

上面的问题答案，其实是不能用全局变量的，threadLocal还是有作用的。先从原理上分析一下，SpringBoot启动时，会启动tomcat或netty作为应用服务器进程，而不管是tomcat还是netty接收客户端求都是用的IO多路复用模型，也就是会有个工作线程池，当同时有多个请求过来时，是可能有多个工线程在运行的，而这些工作线程是共用jvm里的堆内存的（但实际上，在我们这个例子里，RequestContext并没有实例化，我们是直接用它的类变量的，而类变量是和类信息一起存在方法区里的，当然方法区也是线程间共享的），自然会访问到同一个全局变量，这时，如果两个线程并发执行顺序不合，就可能导致线程A读到了线程B写入全局变量的用户信息，造成业务逻辑出错。下面，我们来验证一

，先写个不用ThreadLocal的RequestContext实现：

```
package top.zhengliwei.threadLocalTest.model;

public class BadRequestContext {
    private static UserSession USER_SESSION;

    public static void setUserSession(UserSession userSession) {
        USER_SESSION = userSession;
    }

    public static UserSession getUserSession() {
        return USER_SESSION;
    }

    public static void deleteUserSession() {
        USER_SESSION = null;
    }
}
```

在拦截器里，把用户信息写入这个新的RequestContext实现里：

```
        //RequestContext.setUserSession(getUserInfo(token, request));
        BadRequestContext.setUserSession(getUserInfo(token, request));
```

顺便可以把postHandle方法里的删除userSession的代码注释掉，防止影响我们的验证：

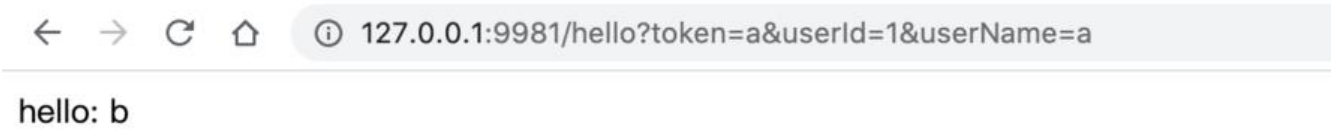
```
@Override
public void postHandle(HttpServletRequest request, HttpServletResponse response, Object h
ndler,
                       ModelAndView modelAndView) {
        // 处理完成后删除userSession
//        RequestContext.deleteUserSession();
    }
```

最后，为了方便验证，在controller里休眠5s，模拟处理请求时间比较长的情况：

```
@RestController
@RequestMapping
public class DemoController {

    @RequestMapping("/hello")
    public String sayHello() throws InterruptedException {
//        return "hello: " + RequestContext.getUserSession().getUserName();
        Thread.sleep(5000);
        return "hello: " + BadRequestContext.getUserSession().getUserName();
    }
}
```

重新启动项目，在浏览器里开两个页面，先访问http://127.0.0.1:9981/hello?token=a&userId=1&userName=a，再访问http://127.0.0.1:9981/hello?token=a&userId=2&userName=b，只要两次问时间不超过5s，就会发现，第一次请求返回了hello: b，这就证明，出现了两个线程并发操作全局量导致的并发问题了，如下：



而ThreadLocal维护的对象保证的语义是：该对象和当前线程绑定，只有本线程可以读写该对象，其线程要读写ThreadLocal类型的对象，实际上是新建了个对象来和这个线程绑定。这样，就防止了多程并发操作同一个对象的问题了。

下面，我们来看下ThreadLocal内部的实现原理。

3. ThreadLocal的实现

点击RequestContext代码里的USER_SESSION.set()方法即可进到ThreadLocal的源代码里。我把set，et和remove方法的代码贴上来方便大家查看：

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}

public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}

 public void remove() {
     ThreadLocalMap m = getMap(Thread.currentThread());
     if (m != null)
         m.remove(this);
 }
```

可以看到，这三个方法都是操作当前thread对象的一个ThreadLocalMap类型的map，而ThreadLoca对象本身是不存储任何外部可修改变量的，这也解释了为什么我们可以安全地多线程操作ThreadLoca对象，因为它只负责操作数据，不负责存储数据，数据是在每个thread对象里存储的，自然没有并发全问题。

下面我们来看看ThreadLocalMap类，这个类是ThreadLocal的静态内部类，可以看作是个简化版的hshmap，它内部定义了Entry类，并维护了Entry的数组，entry的key的hash值经过映射后作为数组

下标，如果有hash冲突，直接向后取第一个为空的位置插入，算是线性探测再散列的解决方案。

4. 为什么是弱引用

如果我们注意观察Entry类，会发现它继承了WeakReference类，来持有一个ThreadLocal对象的弱用作为key，为什么这里要用弱引用呢？我们来思考一下，假如这里用强引用的话，如果我们的业务码已经释放了ThreadLocal对象的引用，那么这个ThreadLocal对象能否被GC回收呢？答案是不能的因为只要当前线程还存活，那么thread对象就会存在，thread对象会持有ThreadLocalMap的引用，ThreadLocalMap又持有内部每个Entry的引用，而Entry又持有前面的ThreadLocal对象的引用（作为ey），有这样一条引用链存在，GC自然不会回收ThreadLocal对象，但从业务代码来看，我们已经没任何方法可以再次访问到前面的ThreadLocal对象了（我们能不能直接通过Thread的API操作线程的TheadLocalMap对象呢？实际上是不能的，因为线程的threadLocalMap属性是protected的，只有JD内部代码能直接操作它，那么有什么办法能间接操作它吗？还是有的，比如我们再定义一个ThreadLoal对象出来，通过这个ThreadLocal对象间接操作当前线程的threadLocalMap对象，这个操作下面介绍使用场景），它不能被回收，就可以看作是个内存泄露的bug。而使用弱引用的话，只要业务代里释放了ThreadLocal对象的引用，那么就只存在Entry来的一个弱引用，这时GC就会选择回收这个象。

5. 关于内存泄露

网上有很多关于ThreadLocal内存泄露的文章，但我看完后还是有点疑问，经过自己仔细地思考才大想清楚。下面我把我的思考过程记录下来，供大家参考。首先要明确的一点是，所谓内存泄露，是指个对象，我们已经无法访问它了，但它不能被GC回收，因为存在某种强引用关系在引用它。而在ThredLocal里，如果存在这种对象，一定是Entry的value对象，因为key对象是弱引用，是可以被回收的那么，什么情况下value会成为被泄露的对象呢？假设，我们业务代码里清除了对ThreadLocal对象的用，但在清除之前没有调用remove方法，这时，entry的key（也就是ThreadLocal对象）只有一个引用存在，可以被清除，但value对象有来自entry的强引用，因此不能被清除。这时就有可能出现内泄露了，之所以说可能，是因为JDK开发者考虑到了这一点，在ThreadLocal的set, get, remove方法检查了当前thread对象的ThreadLocalMap中是否有key为null的entry存在，如果有，则清理这些enty。我认为，这是JDK开发者提供的一个默认兜底措施，如果业务中先后用了两个ThreadLocal对象的，假如第一个没有执行remove直接清除引用，只要后续调用了第二个ThreaadLocal对象的get或set或emove方法，还是有机会修复第一个错误操作导致的内存泄露问题的（JDK开发者真是为业务开发者碎了心啊）。到这，我认为，要用ThreadLocal构造出一个内存泄露的场景其实挺难的，需要：

1. 创建ThreadLocal对象，并set一个值；

2. 直接清除这个ThreadLocal对象的引用，在这之前不调用remove方法；

3. 确保线程不被销毁，因为销毁线程时，从线程出发引用的ThreadLocalMap，entry和value自然都被GC回收；

4. 确保后续不会再使用ThreadLocal对象，因为，如果使用的话，在get，set等方法内，会尝试检查前线程的ThreadLocalMap里，key为null的entry，清理掉（这里需要注意的是，由于我们已经没有上一个ThreadLocal对象的引用，所以不再能调用它的get，set等方法了，所以，假如我们不小心没调用remove方法，直接清除了ThreadLocal对象的引用，可能需要再建个ThreadLocal对象，调用一get，set方法，来清除上一个ThreadLocal对象关联的value对象。。。开个玩笑）。

综上，我觉得，ThreadLocal的内存泄露问题，更像是个理论上的问题，实际出现的概率不大，除非序员特别不走心。。。

6. 使用建议

虽然我觉得ThreadLocal使用不当，造成内存泄露的情况不多见，不过，还是给大家提供两个ThreadLcal使用上的建议吧，这样使用ThreadLocal会更正规：

1. 在确定不再使用ThreadLocal对象后，显式地调用remove方法。

2. 把ThreadLocal对象定义成 static的，即定义成类变量，这样，我们会一直持有ThreadLocal对象从根源上断绝了内存泄露的情况（不知道大家有没有注意到，上面讨论的弱引用也好，内存泄露也好都是在ThreadLocal对象会被业务代码释放引用的前提下进行讨论的）。

以上，就是我对ThreadLocal的全部认识了，如果有讲的不对的地方，欢迎大家讨论～

吾生也有涯，而知也无涯。庄子诚不我欺。