The magic of strong and weak references in java

Preface

Under what circumstances might ThreadLocal have a memory leak?If you want to understand the context of this problem, it is essential to look at the source code. After looking at the source code, you find that static class Entry extends WeakReference <ThreadLocal<?> {} is actually used in ThreadLocal, and the puzzle is actually WeakReference with a weak reference.

Summary of this article

- Strong reference: Object o = new Object()
- Soft reference: new SoftReference(o);
- Weak reference: new WeakReference(o);
- Virtual reference: new PhantomReference(o);
- Use of ThreadLocal and the cause of memory leaks from improper use

Jdk 1.2 adds abstract classes Reference and SoftReference, WeakReference, PhantomReference, expanding the classification of reference types to achieve finer-grained control of memory.

For example, when we have cached data, I want the cache to free up memory or put the cache out of the heap when there is not enough memory.

But how do we differentiate which objects need to be recycled (garbage collection algorithm, accessibility analysis) so that we can be notified when recycling, so JDK 1.2 brings these types of references.

reference type	When to recycle
Strong Reference	Strongly referenced objects, as long as the GC root is reachable, will not be recycled, out of memory, will throw oom
Soft reference: SoftReference	Soft reference object, in GC root, only soft reference can reach an object a, before oom garbage collection will recycle object a
WeakReference	Weak references, in GC root, only weak references can reach an object c, where gc is recycled
Virtual Reference: PhantomReference	Virtual reference, must be used with ReferenceQueue. It is not known when to recycle, but after recycling, you can manipulate ReferenceQueue to get the recycled reference

Strong Reference

Strong references are the way we often use them: Object o = new Object().When garbage is collected, strongly referenced variables are not recycled. The garbage collector cleans up objects in the heap and releases memory only if o=null is set, jvm passes accessibility analysis, and no GC root reaches the object.When you continue to apply for memory allocation, oom will occur.

Define a class Demo, Demo instance takes up memory size of 10m, don't keep adding Demo examples to the list, program throws oom to terminate because memory allocation can't be applied for

```
// -Xmx600m
```

```
public class SoftReferenceDemo {
    // 1m
    private static int _1M = 1024 * 1024 * 1;
    public static void main(String[] args) throws InterruptedException {
        ArrayList<Object> objects = Lists.newArrayListWithCapacity(50);
        int count = 1;
        while (true) {
            Thread.sleep(100);
            // Get how much jvm free memory is m
            long meme_free = Runtime.getRuntime().freeMemory() / _1M;
            if ((meme_free - 10) >= 0) {
                Demo demo = new Demo(count);
                objects.add(demo);
                count++;
                demo = null;
            }
            System.out.println("jvm idle memory" + meme_free + " m");
            System.out.println(objects.size());
        }
    }

    @Data
    static class Demo {
        private byte[] a = new byte[_1M * 10];
        private String str;
        public Demo(int i) {
            this.str = String.valueOf(i);
        }
    }
}
```

As a result of running the above code, throw oom program to stop

```
jvm Free memory 41 m
54
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at
com.fly.blog.ref.SoftReferenceDemo$Demo.<init>(SoftReferenceDemo.java:37)
        at
com.fly.blog.ref.SoftReferenceDemo.main(SoftReferenceDemo.java:25)
```

But there are business scenarios where we need to run out of memory to free up unnecessary data.For example, the user information we store in the cache.

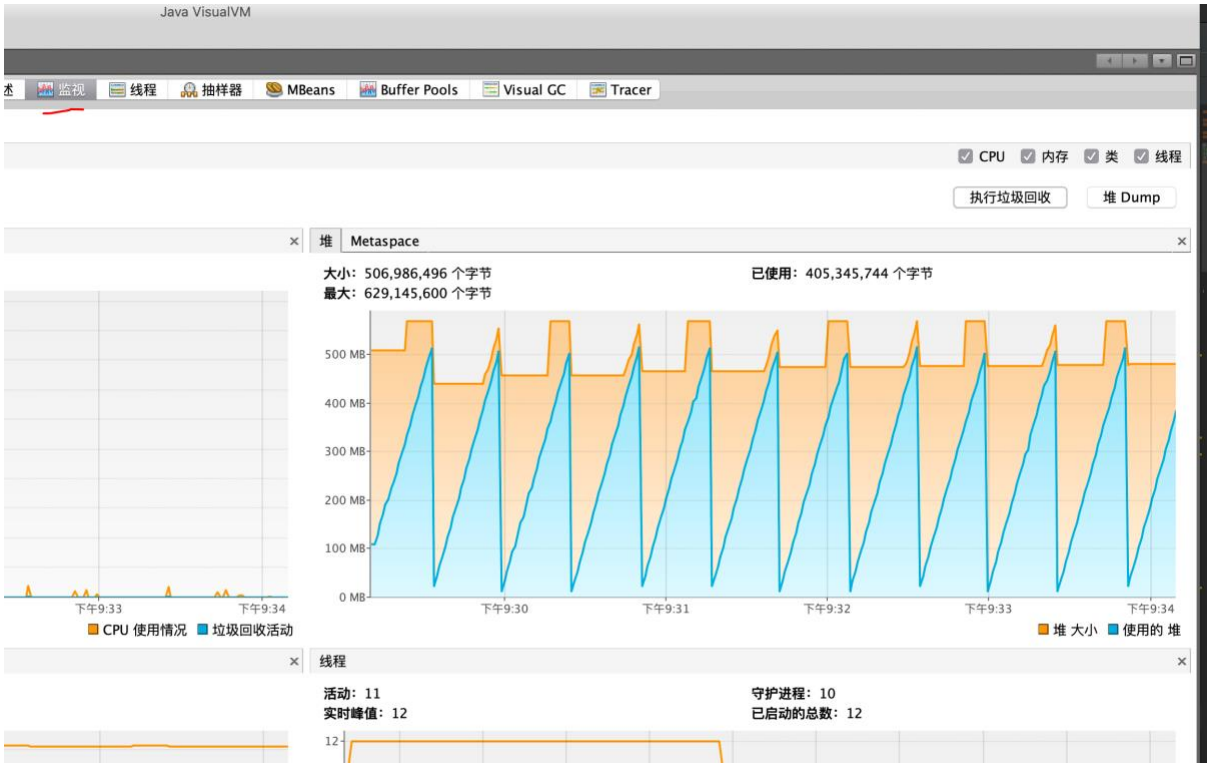
Soft reference

jdk has joined Reference since 1.2. SoftReference is one of the categories. Its purpose is to reach object a through GC root. Only SoftReference, object a will be released by jvm gc before jvm oom.

Loop indefinitely to add data about 10m in size (SoftReference) to the List and no oom appears.

```
// -Xmx600m
public class SoftReferenceDemo {
    // 1m
    private static int _1M = 1024 * 1024 * 1;
    public static void main(String[] args) throws InterruptedException {
        ArrayList<Object> objects = Lists.newArrayListWithCapacity(50);
        int count = 1;
```

```
        while (true) {
            Thread.sleep(500);
            // Get how much jvm free memory is m
            long meme_free = Runtime.getRuntime().freeMemory() / _1M;
            if ((meme_free - 10) >= 0) {
                Demo demo = new Demo(count);
                SoftReference<Demo> demoSoftReference = new
SoftReference<>(demo);
                objects.add(demoSoftReference);
                count++;
                // Demo is null, only demoSoftReference refers to an
instance that arrives at Demo, and GC will recycle Demo instances before
oom
                demo = null;
            }
            System.out.println("jvm idle memory" + meme_free + " m");
            System.out.println(objects.size());
        }
    }
    @Data
    static class Demo {
        private byte[] a = new byte[_1M * 10];
        private String str;
        public Demo(int i) {
            this.str = String.valueOf(i);
        }
    }
}
```



Looking at the use of the jvm heap through jvisualvm, you can see that the heap will be recycled when it overflows. When there is a lot of free memory, you actively perform garbage collection and the memory will not be recycled.

Weak reference

When the object demo's reference is only accessible by WeakReference, demo is recycled after gc to free up memory.

The following programs will continue to run, but at different times the memory will be freed

```
// -Xmx600m -XX:+PrintGCDetails
public class WeakReferenceDemo {
    // 1m
    private static int _1M = 1024 * 1024 * 1;

    public static void main(String[] args) throws InterruptedException {
        ArrayList<Object> objects = Lists.newArrayListWithCapacity(50);
        int count = 1;
        while (true) {
            Thread.sleep(100);
            // Get how much jvm free memory is m
            long meme_free = Runtime.getRuntime().freeMemory() / _1M;
            if ((meme_free - 10) >= 0) {
                Demo demo = new Demo(count);
                WeakReference<Demo> demoWeakReference = new
WeakReference<>(demo);
                objects.add(demoWeakReference);
                count++;
                demo = null;
            }
            System.out.println("jvm idle memory" + meme_free + " m");
            System.out.println(objects.size());
        }
    }

    @Data
    static class Demo {
        private byte[] a = new byte[_1M * 10];
        private String str;
        public Demo(int i) {
            this.str = String.valueOf(i);
        }
    }
}
```

As a result, SoftReference's available memory frees up when it's almost exhausted, while WeakReference runs out of garbage around 360m each time it's available, freeing up memory

```
[GC (Allocation Failure) [PSYoungGen: 129159K->1088K(153088K)]
129175K->1104K(502784K), 0.0007990 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
jvm Free memory 364 m
36
jvm Free memory 477 m
```

Virtual reference

It's also called a phantom reference because you don't know when it will be recycled and you have to work with a ReferenceQueue to get an instance of PhantomReference from this queue when the object is recycled.

```
// -Xmx600m -XX:+PrintGCDetails
public class PhantomReferenceDemo {
    // 1m
    private static int _1M = 1024 * 1024 * 1;

    private static ReferenceQueue referenceQueue = new ReferenceQueue();

    public static void main(String[] args) throws InterruptedException {
        ArrayList<Object> objects = Lists.newArrayListWithCapacity(50);
        int count = 1;
        new Thread(() -> {
            while (true) {
                try {
                    Reference remove = referenceQueue.remove();
                    // objects accessibility analysis, PhantomReference
<Demo>, memory is not released in time, we need to get the Demo in the
queue recycled, and then
                    // Remove this object from objects
                    if (objects.remove(remove)) {
                        System.out.println("Removing Elements");
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
        while (true) {
            Thread.sleep(500);
            // Get how much jvm free memory is m
            long meme_free = Runtime.getRuntime().freeMemory() / _1M;
            if ((meme_free - 10) > 40) {
                Demo demo = new Demo(count);
                PhantomReference<Demo> demoWeakReference = new
PhantomReference<>(demo, referenceQueue);
                objects.add(demoWeakReference);
                count++;
                demo = null;
            }
            System.out.println("jvm idle memory" + meme_free + " m");
            System.out.println(objects.size());
        }
    }

    @Data
    static class Demo {
        private byte[] a = new byte[_1M * 10];
        private String str;

        public Demo(int i) {
            this.str = String.valueOf(i);
        }
    }
}
```

ThreadLocal

ThreadLocal is still used a lot in our actual development.So what exactly is it (thread local variable)? We know local variables (variables defined within a method) and member variables (properties of a class).

Sometimes, we want the life cycle of a variable to run through a task cycle of the entire thread (threads in the thread pool can be assigned to perform different tasks), and we get this preset variable when each method is called, which is what ThreadLocal does.

For example, we want to get the current request's HttpServletRequest, which is available in all current methods. SpringBoot has been wrapped for us. After each request comes, the RequestContextFilter sets the thread's local variable through the RequestContextHolder, which operates on ThreadLocal.

ThreadLocal is only for calls in the current thread, cross-thread calls are not possible, so Jdk implements it by inheriting ThreadLocal from InheritableThreadLocal.

ThreadLocal Gets User Information for Current Request

Look at the notes to get a general idea of how TheadLocal is used

```
/**
 * @author Zhang Panqin
 * @date 2018/12/21-22:59
 */
@RestController
public class UserInfoController {
    @RequestMapping("/user/info")
    public UserInfoDTO getUserInfoDTO() {
        return UserInfoInterceptor.getCurrentRequestUserInfoDTO();
    }
}

@Slf4j
public class UserInfoInterceptor implements HandlerInterceptor {
    private static final ThreadLocal<UserInfoDTO> THREAD_LOCAL = new
ThreadLocal();
    // Request Header User Name
    private static final String USER_NAME = "userName";
    // Note that only bean s injected into ioc can be injected
    @Autowired
    private IUserInfoService userInfoService;
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        // Determine if it is an interface request
        if (handler instanceof HandlerMethod) {
            String userName = request.getHeader(USER_NAME);
            UserInfoDTO userInfoByUserName =
userInfoService.getUserInfoByUserName(userName);
            THREAD_LOCAL.set(userInfoByUserName);
            return true;
        }
        return false;
    }
    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
        // Release memory when used up
        THREAD_LOCAL.remove();
    }
    // Get user information for current thread settings
    public static UserInfoDTO getCurrentRequestUserInfoDTO() {
```

```
        return THREAD_LOCAL.get();
    }
}

@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

    /**
     * Inject UserInfoInterceptor into ioc container
     */
    @Bean
    public UserInfoInterceptor getUserInfoInterceptor() {
        return new UserInfoInterceptor();
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // Calling this method returns ioc's bean

registry.addInterceptor(getUserInfoInterceptor()).addPathPatterns("/**");
    }
}
```

InheritableThreadLocal

Sometimes, we want the life cycle of the local variable of the current thread to extend to the child thread, where the variable set by the parent thread is taken.InheritableThreadLocal provides this capability.

```
/**
 * @author Zhang Panqin
 * @date 2020-06-27-21:18
 */
public class InheritableThreadLocalDemo {
    static InheritableThreadLocal<String> INHERITABLE_THREAD_LOCAL = new
InheritableThreadLocal();
    static ThreadLocal<String> THREAD_LOCAL = new ThreadLocal<>();
    public static void main(String[] args) throws InterruptedException {
        INHERITABLE_THREAD_LOCAL.set("Use in parent thread
InheritableThreadLocal set variable");
        THREAD_LOCAL.set("Use in parent thread ThreadLocal set variable");
        Thread thread = new Thread(
                () -> {
                    // Get the set variable
                    System.out.println("from InheritableThreadLocal Take
the variable set by the parent thread: " + INHERITABLE_THREAD_LOCAL.get());
                    // Print as null
                    System.out.println("from ThreadLocal Take the variable
set by the parent thread: " + THREAD_LOCAL.get());
                }
        );
        thread.start();
        thread.join();
    }
}
```

Source Code Analysis for ThreadLocal get Method

You can understand that the Thead object has an attribute Map whose key is the ThreadLoal instance, which gets the source code for the thread local variable

```
public class ThreadLocal<T> {
    public T get() {
        // Get Run on That Thread
        Thread t = Thread.currentThread();
        // Get Map from Thread
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            // Get values from Map using the ThreadLocal instance
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        // Initialize the Map and return the initialization value, which is
null by default. You can define a method from which to load the
initialization value
        return setInitialValue();
    }
}
```

InheritableThreadLocal Gets Data Analysis of Parent Thread Settings

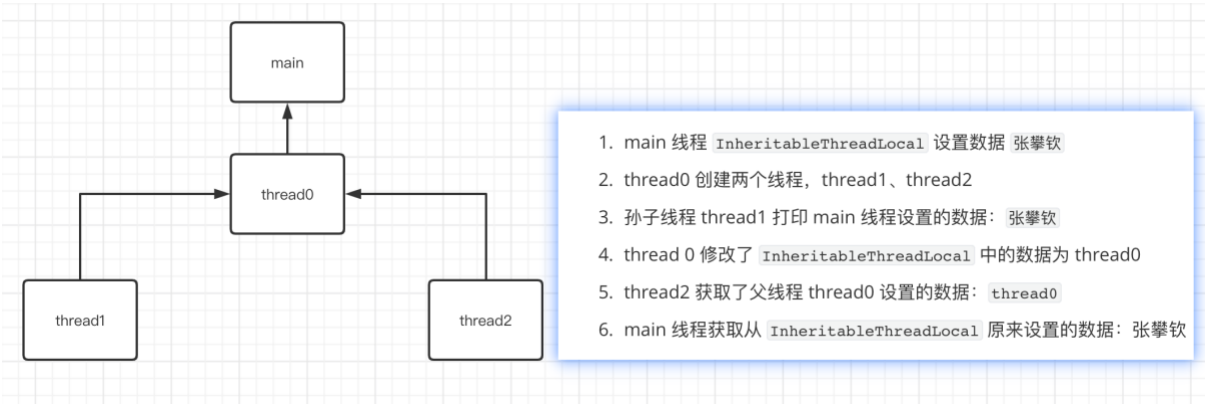
Each Thread also has a Map property, inheritableThreadLocals, that holds the value copied from the parent thread.

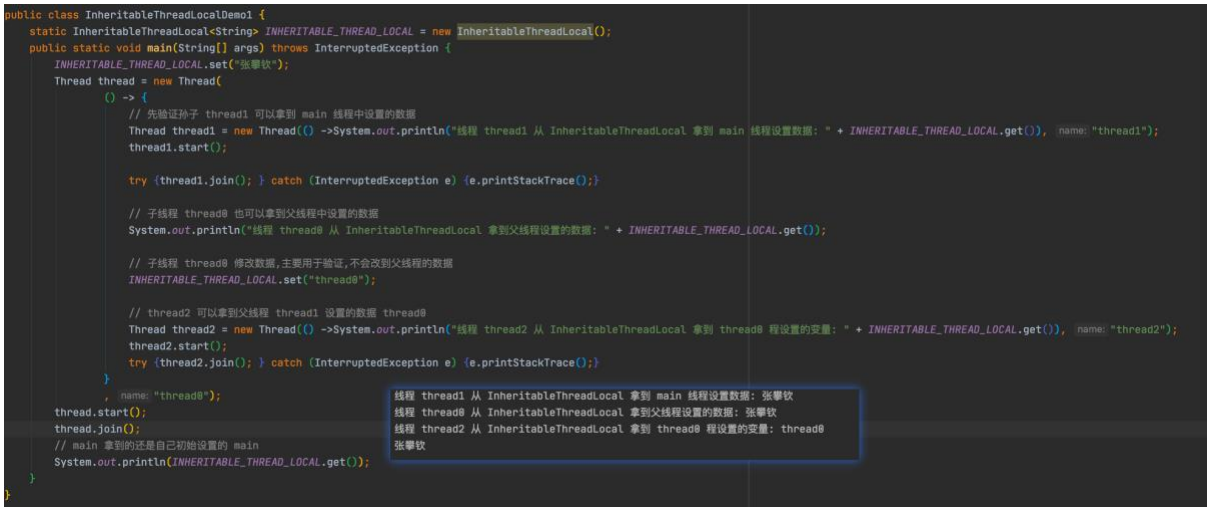
When a child thread is initialized, it copies the value of its parent's Map (inheritable ThreadLocals) to its own Thead Map (inheritable ThreadLocals). Each thread maintains its own inheritable ThreadLocals, so the child thread cannot change the data maintained by the parent thread, only the child thread can get the data set by the parent thread.

```
public class Thread{

        // Maintain thread local variables
    ThreadLocal.ThreadLocalMap threadLocals = null;

    // Maintain data for parent threads that can be inherited by child
threads
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;

   // Thread Initialization
    public Thread(ThreadGroup group, Runnable target, String name,
                  long stackSize) {
        init(group, target, name, stackSize);
    }

    private void init(ThreadGroup g, Runnable target, String name,
                      long stackSize, AccessControlContext acc,
                      boolean inheritThreadLocals) {
        if (inheritThreadLocals && parent.inheritableThreadLocals != null){
            // Copy inheritable ThreadLocals data from parent thread to
child thread
            this.inheritableThreadLocals =
ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
        }
    }
}
```

```
public class TheadLocal{
    static ThreadLocalMap createInheritedMap(ThreadLocalMap parentMap) {
        ///Create your own thread's Map, copy in the parent thread's value
        return new ThreadLocalMap(parentMap);
    }

    static class ThreadLocalMap {
        private ThreadLocalMap(ThreadLocalMap parentMap) {
            Entry[] parentTable = parentMap.table;
            int len = parentTable.length;
            setThreshold(len);
            table = new Entry[len];
            // Traverse the parent thread and copy the data over
            for (int j = 0; j < len; j++) {
                Entry e = parentTable[j];
                if (e != null) {
                    @SuppressWarnings("unchecked")
                    ThreadLocal<Object> key = (ThreadLocal<Object>)
e.get();
                    if (key != null) {
                        Object value = key.childValue(e.value);
                        Entry c = new Entry(key, value);
                        int h = key.threadLocalHashCode & (len - 1);
                        while (table[h] != null)
                            h = nextIndex(h, len);
                        table[h] = c;
                        size++;
                    }
                }
            }
        }
    }
}
```

demo validation, above analysis



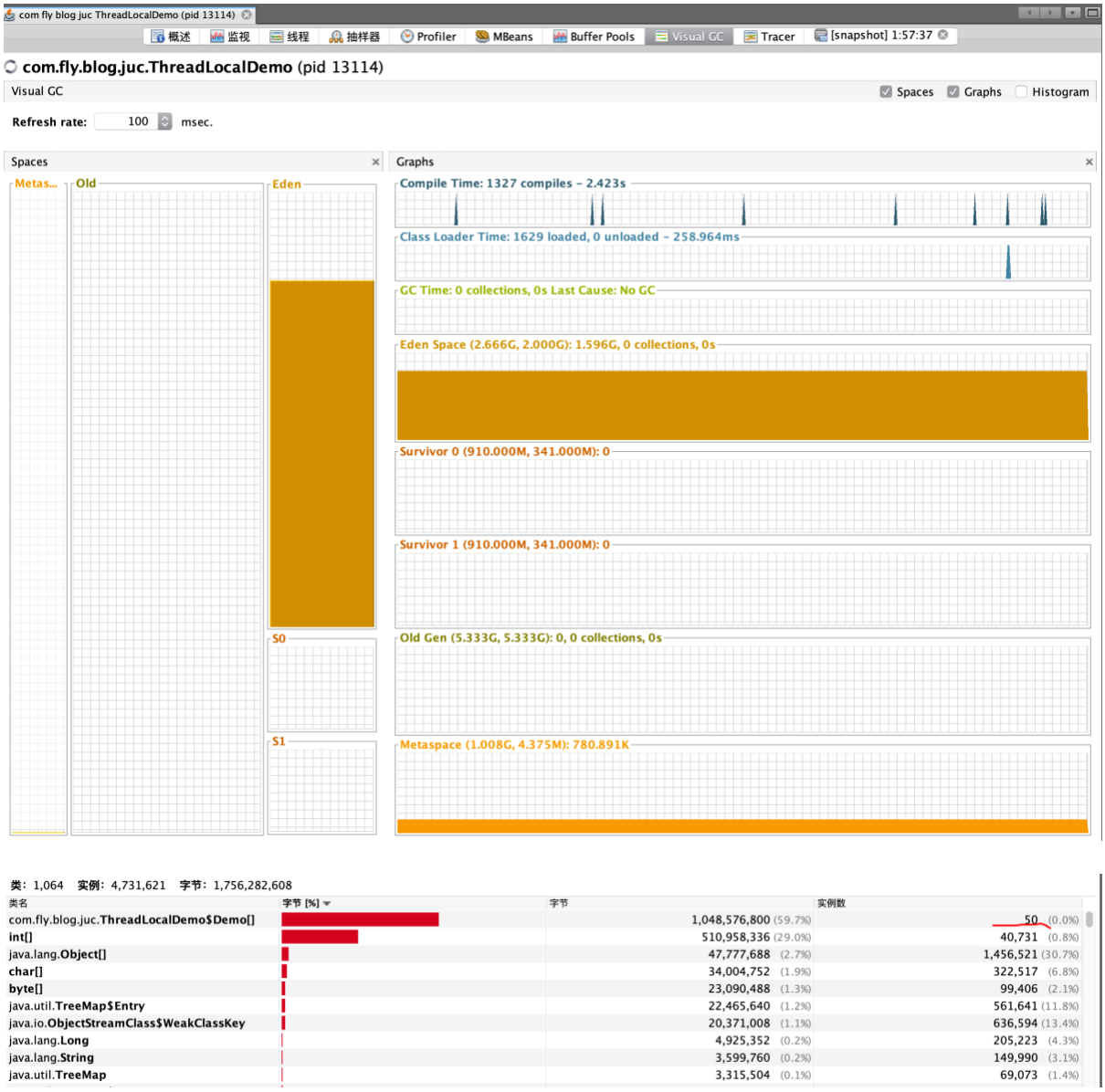


Memory leak cause

A 20-size thread pool is defined to perform 50 tasks, and after execution, threadLocal is set to null to simulate a memory leak scenario.To eliminate interference, I set the jvm parameter to -Xms8g -Xmx8g -XX:+PrintGCDetails

```
public class ThreadLocalDemo {
    private static ExecutorService executorService =
Executors.newFixedThreadPool(20);
    private static ThreadLocal threadLocal = new ThreadLocal();

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 50; i++) {
            executorService.submit(() -> {
                try {
                    threadLocal.set(new Demo());
                    Thread.sleep(50);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    if (Objects.nonNull(threadLocal)) {
                        // To prevent memory leaks, the current thread runs
out, clearing the value
//                          threadLocal.remove();
                    }
                }
            });
        }
        Thread.sleep(5000);
        threadLocal = null;
        while (true) {
            Thread.sleep(2000);
        }
    }
    @Data
    static class Demo {
        //
        private Demo[] demos = new Demo[1024 * 1024 * 5];
    }
}
```

Run the program without printing gc logs indicating no garbage collection



In the Java VisualVM we do garbage collection, the distribution of memory after collection, this 20 ThreadLocalDemo$Demo[] is not recyclable, this is a memory leak.

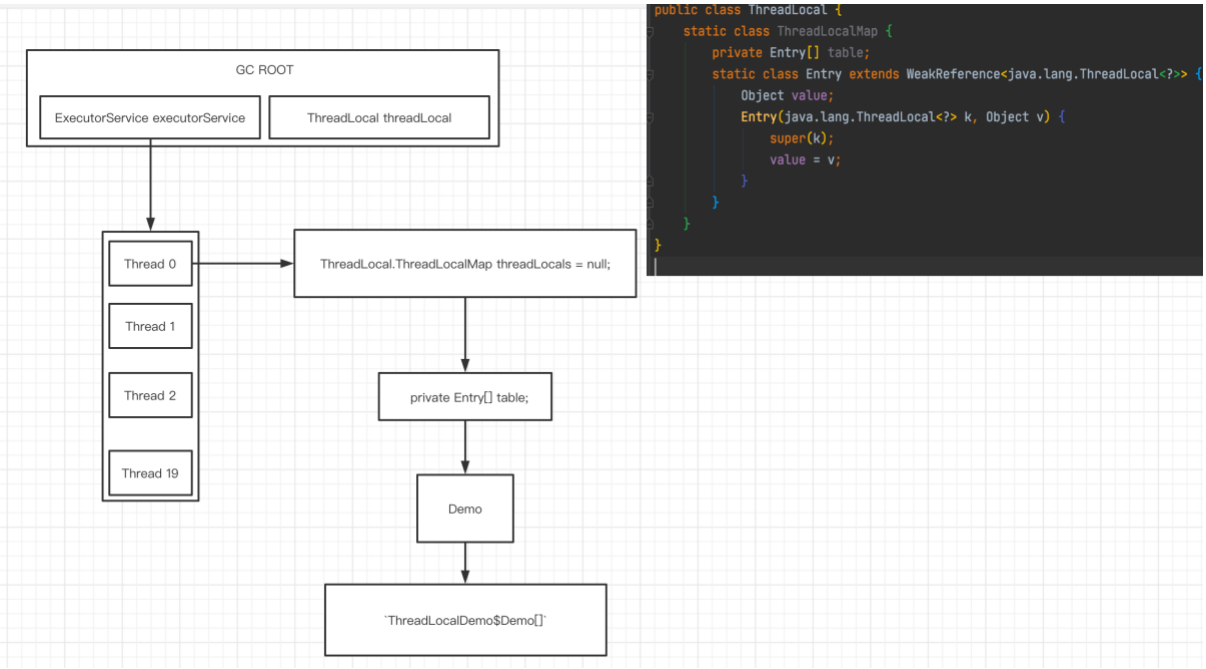
Fifty Demos were created 50 times in a program loop, and garbage collection is not triggered while the program is running (guaranteed by setting the jvm parameter), so the number of instances ThreadLocalDemo$Demo[] survives is 50.

When I trigger the GC manually, the number of instances drops to 20, which is not what we expected. This is the memory leak problem in the program.

Why did a memory leak occur?

Because each thread corresponds to a Thread, the thread pool size is 20.In ThreadThreadLocal.ThreadLocalMapThreadLocals = null;

ThreadLocalMap has Entry[] tables, k is a weak reference.When we set threadLocal to null, the reference chain from GC ROOT to ThreadLocalDemo$Demo[] still exists, but k is recycled, value still exists, tables are of constant length and will not be recycled.



ThreadLocal is optimized for the case where k is null when set and get, setting the corresponding tables[i] to null.This allows a single Entry to be recycled.But once we set ThreadLocal to null, we can't manipulate method calls.You can only wait until Thread calls another ThreadLocal to operate on ThreadLocalMap, then make a rehash of the Map and delete the Entry with K null.

The above problem solving is also convenient. When a thread has finished using thread local variables, call remove to actively clear the Entry.

This article is written by Zhang Panqin's Blog http://www.mflyyou.cn/ A literary creation.It can be freely reproduced and quoted, but with the signature of the author and the citation of the article.

If you upload it to the WeChat Public Number, add the Author Public Number 2-D at the end of the article.WeChat Public Number Name: Mflyyou

Keywords: Front-end jvm Java JDK SpringBoot

Added by **JoeZ** on *Sat, 27 Jun 2020 22:59:49 +0300*