
zhenchao RP 120
2018-04-11 发布

ThreadLocal 线程安全机制与小地雷

Java 多线程类库对于共享数据的读写控制主要采用锁机制保证线程安全，本文所要探究的 ThreadLocal 则采用了一种完全不同的策略。ThreadLocal 不是用来解决共享数据的并发访问问题的，它让每个线程都将目标数据复制一份作为线程私有，后续对于该数据的操作都是在各自私有的副本上进行，线程之间彼此相互隔离，也就不存在竞争问题。

下面的例子演示了 ThreadLocal 的典型应用场景，在 jdk 1.8 之前，如果我们希望对日期和时间进行格式化操作，则需要使用 SimpleDateFormat 类，而我们知道它是是非线程安全的，在多线程并发执行时会出现一些奇怪的问题，而对于该类使用的最佳实践则是采用 ThreadLocal 进行包装，以保证每个线程都有一份属于自己的 SimpleDateFormat 对象，如下所示：

```
ThreadLocal<SimpleDateFormat> sdf = new ThreadLocal<SimpleDateFormat>() {
    @Override
    protected SimpleDateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    }
};
```

一. 线程安全机制

那么 ThreadLocal 是怎么做到让修饰的对象能够在每个线程中各持有一份呢？我们先来简单的概括一下：在 ThreadLocal 中定义了一个静态内部类 ThreadLocalMap，可以将其理解为一个特有的 Map 类型，而在 Thread 类中声明了一个 ThreadLocalMap 类型的属性 threadLocals，所以针对每个 Thread 对象，也就是每个线程来说都包含了一个 ThreadLocalMap 对象，即每个线程都有一个属于自己的内存数据库，而数据库中存储的就是我们用 ThreadLocal 修饰的对象，这里的 key 就是对应的 ThreadLocal 对象，而 value 就是我们记录在 ThreadLocal 中的值。当希望获取该对象时，我们首先需要拿到当前线程对应的 Thread 对象，然后获取到该对象对应的 threadLocals 属性，也就拿到了线程私有的内存数据库，最后以 ThreadLocal 对象为 key 获取到其修饰的目标值。整个过程还是有点绕的，可以借助下面这幅图进行理解。

1.1 内存数据库 ThreadLocalMap

接下来看一下相应的源码实现，首先来看一下内部定义的 ThreadLocalMap 静态内部类：

```
        size = 1;
        setThreshold(INITIAL_CAPACITY);
    }

    // 构造方法
    private ThreadLocalMap(ThreadLocalMap parentMap) {
        Entry[] parentTable = parentMap.table;
        int len = parentTable.length;
        setThreshold(len);
        table = new Entry[len];

        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++;
```

ThreadLocalMap 是一个定制化的 Map 实现，这里可以简单将其理解为一般的 Map，用作键值存储的内存数据库，至于为什么要专门实现而不是复用已有的 HashMap，我们在后面进行说明。

1.2 ThreadLocal 方法实现

了解了 ThreadLocalMap 的定义，我们再来看一下 ThreadLocal 的实现。对于 ThreadLocal 来说，对外暴露的方法主要有 get、set，以及 remove 三个，下面逐一来看：

- **获取线程私有值：get()**

与一般的 Map 取值操作不同，这里的 `get()` 并没有要求提供查询的 key，也正如前面所说的，这里的 key 就是调用 `get()` 方法的对象自身：

```
public T get() {
    // 获取当前线程对象
    Thread t = Thread.currentThread();
    // 获取当前线程对象的 threadLocals 属性
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        // 以 ThreadLocal 对象为 key 获取目标线程私有值
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

如果当前线程对应的内存数据库 map 对象还未创建，则会调用 `setInitialValue()` 方法执行创建，如果在构造 ThreadLocal 对象时覆盖实现了 `initialValue()` 方法，则会调用该方法获取构造的初始化值并记录到创建的 map 对象中：

```
private T setInitialValue() {
    // 调用模板方法 initialValue 获取指定的初始值
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        // 以当前 ThreadLocal 对象为 key 记录初始值
        map.set(this, value);
    else
        // 创建 map 并记录初始值
        createMap(t, value);
    return value;
}
```

- **添加线程私有值：set(T value)**

再来看一下 set 方法，因为 key 就是当前 ThreadLocal 对象，所以 set 方法也不需要指定 key：

```
public void set(T value) {
    // 获取当前线程对象
    Thread t = Thread.currentThread();
    // 获取当前线程对象的 threadLocals 属性
    ThreadLocalMap map = getMap(t);
    if (map != null)
        // 以当前 ThreadLocal 对象为 key 记录线程私有值
        map.set(this, value);
    else
        createMap(t, value);
}
```

和 get 方法的流程大致一样，都是操作当前线程私有的内存数据库 ThreadLocalMap，并记录目标值。

- **删除线程私有值：remove()**

remove 方法以当前 ThreadLocal 为 key，从当前线程内存数据库 ThreadLocalMap 中删除目标值，具体逻辑比较简单：

```
public void remove() {
    ThreadLocalMap m = getMap(Thread.currentThread());
    if (m != null)
        // 以当前 ThreadLocal 对象为 key
        m.remove(this);
}
```

ThreadLocal 对外暴露的功能虽然有点小神奇，但是具体对应到内部实现并没有什么复杂的逻辑，如果我们把每个线程持有的专属 ThreadLocalMap 对象理解为当前线程的私有数据库，那么也就不难理解 ThreadLocal 的运行机制，每个线程自己维护自己的数据，彼此相互隔离，不存在竞争，也就没有线程安全问题可言。

二. 真的就高枕无忧了吗?

虽然对于每个线程来说数据是隔离的，但这也不表示任何对象丢到 ThreadLocal 中就万事大吉了，思考一下下面几种情况：

> 1. 如果记录在 ThreadLocal 中的是一个线程共享的外部对象呢?
> 2. 引入线程池，情况又会有什么变化?

> 3. 如果 ThreadLocal 被 static 关键字修饰呢?

先来看 **第一个问题** ，如果我们记录的是一个外部线程共享的对象，虽然我们以当前线程私有的 ThreadLocal 对象作为 key 对其进行了存储，但是恶魔终究是恶魔，共享的本质并不会因此而改变，这种情况下的访问还是需要进行同步控制，最好的方法就是从源头屏蔽掉这类问题。我们来举个例子：

```
public class ThreadLocalWithSharedInstance implements Runnable {

    // list 是一个事实共享的实例，即使被 ThreadLocal 修饰
    private static List<String> list = new ArrayList<>();
    private ThreadLocal<List<String>> threadLocal = ThreadLocal.withInitial(() -> l

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            List<String> li = threadLocal.get();
            li.add(Thread.currentThread().getName() + "_" + RandomUtils.nextInt(0,
            threadLocal.set(li);
        }
        System.out.println("[Thread-" + Thread.currentThread().getName() + "], list=
    }

    public static void main(String[] args) throws Exception {
        Thread ta = new Thread(new ThreadLocalWithSharedInstance(), "a");
        Thread tb = new Thread(new ThreadLocalWithSharedInstance(), "b");
        Thread tc = new Thread(new ThreadLocalWithSharedInstance(), "c");
        ta.start(); ta.join();
        tb.start(); tb.join();
        tc.start(); tc.join();
    }
```

以上程序最终的输出如下：

```
[Thread-a], list=[a_2, a_7, a_4, a_5, a_7]
[Thread-b], list=[a_2, a_7, a_4, a_5, a_7, b_3, b_3, b_4, b_7, b_7]
[Thread-c], list=[a_2, a_7, a_4, a_5, a_7, b_3, b_3, b_4, b_7, b_7, c_8, c_3, c_4,
```

可以看到虽然使用了 ThreadLocal 修饰，但是 list 还是以共享的方式在多个线程之间被访问，如果不加同步控制，则会存在线程安全问题。

再来看 **第二个问题** ，相对问题一来说引入线程池就更加可怕，因为大部分时候我们都不会意识到问题的存在，直到代码暴露出奇怪的现象，这个时候并没有违背线程私有的本质，只是一个线程被复用来处理多个业

务，而这个被线程私有的对象也会在多个业务之间被“共享”。例如：

```
public class ThreadLocalWithThreadPool implements Callable<Boolean> {

    private static final int NCPU = Runtime.getRuntime().availableProcessors();

    private ThreadLocal<List<String>> threadLocal = ThreadLocal.withInitial(() -> {
        System.out.println("thread-" + Thread.currentThread().getId() + " init threa
        return new ArrayList<>();
    });

    @Override
    public Boolean call() throws Exception {
        for (int i = 0; i < 5; i++) {
            List<String> li = threadLocal.get();
            li.add(Thread.currentThread().getId() + "_" + RandomUtils.nextInt(0, 10
            threadLocal.set(li);
        }
        System.out.println("[Thread-" + Thread.currentThread().getId() + "], list="
        return true;
    }

    public static void main(String[] args) throws Exception {
        System.out.println("cpu core size : " + NCPU);
        List<Callable<Boolean>> tasks = new ArrayList<>(NCPU * 2);
        ThreadLocalWithThreadPool tl = new ThreadLocalWithThreadPool();
```

以上程序的最终输出如下：

```
cpu core size : 8
thread-12 init thread local
thread-11 init thread local
[Thread-12], list=[12_8, 12_8, 12_4, 12_0, 12_1]
[Thread-11], list=[11_3, 11_3, 11_4, 11_8, 11_4]
[Thread-12], list=[12_8, 12_8, 12_4, 12_0, 12_1, 12_6, 12_7, 12_8, 12_8, 12_8]
[Thread-11], list=[11_3, 11_3, 11_4, 11_8, 11_4, 11_0, 11_2, 11_1, 11_7, 11_9]
[Thread-12], list=[12_8, 12_8, 12_4, 12_0, 12_1, 12_6, 12_7, 12_8, 12_8, 12_8, 12_8
[Thread-11], list=[11_3, 11_3, 11_4, 11_8, 11_4, 11_0, 11_2, 11_1, 11_7, 11_9, 11_0
[Thread-12], list=[12_8, 12_8, 12_4, 12_0, 12_1, 12_6, 12_7, 12_8, 12_8, 12_8, 12_8
[Thread-11], list=[11_3, 11_3, 11_4, 11_8, 11_4, 11_0, 11_2, 11_1, 11_7, 11_9, 11_0
[Thread-12], list=[12_8, 12_8, 12_4, 12_0, 12_1, 12_6, 12_7, 12_8, 12_8, 12_8, 12_8
[Thread-11], list=[11_3, 11_3, 11_4, 11_8, 11_4, 11_0, 11_2, 11_1, 11_7, 11_9, 11_0
[Thread-12], list=[12_8, 12_8, 12_4, 12_0, 12_1, 12_6, 12_7, 12_8, 12_8, 12_8, 12_8
[Thread-11], list=[11_3, 11_3, 11_4, 11_8, 11_4, 11_0, 11_2, 11_1, 11_7, 11_9, 11_0
[Thread-12], list=[12_8, 12_8, 12_4, 12_0, 12_1, 12_6, 12_7, 12_8, 12_8, 12_8, 12_8
[Thread-11], list=[11_3, 11_3, 11_4, 11_8, 11_4, 11_0, 11_2, 11_1, 11_7, 11_9, 11_0
```

```
[Thread-12], list=[12_8, 12_8, 12_4, 12_0, 12_1, 12_6, 12_7, 12_8, 12_8, 12_8, 12_8
[Thread-11], list=[11_3, 11_3, 11_4, 11_8, 11_4, 11_0, 11_2, 11_1, 11_7, 11_9, 11_0
```

在我的 8 核处理器上，我用一个大小为 2 的线程池进行了模拟，可以看到初始化方法被调用了两次，所有线程的操作都是复用这两个线程。回忆一下前文所说的，ThreadLocal 的本质就是每个线程维护一个线程私有的内存数据库来记录线程私有的对象，但是在线程池情况下线程是会被复用的，也就是说线程私有的内存数据库也会被复用，如果在一个线程被使用完准备回放到线程池中之前，我们没有对记录在数据库中的数据执行清理，那么这部分数据就会被下一个复用该线程的业务看到，从而间接的共享了该部分数据（哈哈，你的笔记本电脑在送人之前一定要对硬盘执行多次格式化，不然冠希哥会对你微笑哦）。

最后我们再来看一下 **第三个问题** ，我们尝试将 ThreadLocal 对象用 static 关键字进行修饰：

```
public class ThreadLocalWithStaticEmbellish implements Runnable {

    private static final int NCPU = Runtime.getRuntime().availableProcessors();

    private static ThreadLocal<List<String>> threadLocal = ThreadLocal.withInitial(
        System.out.println("thread-" + Thread.currentThread().getName() + " init th
        return new ArrayList<>();
    });

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            List<String> li = threadLocal.get();
            li.add(Thread.currentThread().getId() + "_" + RandomUtils.nextInt(0, 10
            threadLocal.set(li);
        }
        System.out.println("[Thread-" + Thread.currentThread().getName() + "], list=
    }

    public static void main(String[] args) throws Exception {
        ThreadLocalWithStaticEmbellish tl = new ThreadLocalWithStaticEmbellish();
        for (int i = 0; i < NCPU + 1; i++) {
            Thread thread = new Thread(tl, String.valueOf((char) (i + 97)));
            thread.start(): thread.ioin():
```

以上程序的最终输出如下：

```
thread-a init thread local
[Thread-a], list=[11_4, 11_4, 11_4, 11_8, 11_0]
thread-b init thread local
[Thread-b], list=[12_0, 12_9, 12_0, 12_3, 12_3]
```

```
thread-c init thread local
[Thread-c], list=[13_6, 13_7, 13_5, 13_2, 13_0]
thread-d init thread local
[Thread-d], list=[14_1, 14_5, 14_5, 14_9, 14_2]
thread-e init thread local
[Thread-e], list=[15_4, 15_2, 15_6, 15_0, 15_8]
thread-f init thread local
[Thread-f], list=[16_7, 16_3, 16_8, 16_0, 16_0]
thread-g init thread local
[Thread-g], list=[17_6, 17_3, 17_8, 17_7, 17_1]
thread-h init thread local
[Thread-h], list=[18_0, 18_4, 18_5, 18_9, 18_3]
thread-i init thread local
[Thread-i], list=[19_7, 19_3, 19_7, 19_2, 19_0]
```

由程序运行结果可以看到 static 修饰并没有引出什么问题，实际上这也是很容易理解的，ThreadLocal 采用 static 修饰仅仅是让数据库中记录的 key 是一样的，但是每个线程的内存数据库还是私有的，并没有被共享，就像不同的公司都有自己的用户信息表，即使一些公司之间的用户 ID 是一样的，但是对应的用户数据却是完全隔离的。

以上例子演示了一开始抛出的三个问题，其中问题一和问题二都是 ThreadLocal 使用过程中的小地雷。例子举的不一定恰当，实际中可能也不一定会如示例中这样去使用 ThreadLocal，主要还是为了传达一些意识。如果明白了 ThreadLocal 的内部实现细节，就能够很自然的绕过这些小地雷。

三. 真的会内存泄露吗?

关于 ThreadLocal 导致内存泄露的问题，曾经有一段时间在网上争得沸沸扬扬，那么到底会不会导致内存泄露呢？这里先给出答案：

> 如果使用不恰当，存在内存泄露的可能性。

我们来分析一下内存泄露的条件和原因，在最开始看 ThreadLocal 源码的时候，我就有一个疑问，__ThreadLocal 为什么要专门实现 ThreadLocalMap，而不是采用已有的 HashMap 代替__？后来分析具体实现时看到执行存储时的 key 为当前 ThreadLocal 对象，不需要专门指定 key 能够在一定程度上简化使用，但这并不足以为此专门去实现 ThreadLocalMap。继续阅读我发现 ThreadLocalMap 在实现 Entry 的时候有些奇怪，居然继承了 WeakReference：

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

```
        Object value;

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

从而让 key 成为一个弱引用，我们知道弱引用对象拥有非常短暂的生命周期，在垃圾收集器线程扫描其所管辖的内存区域过程中，一旦发现了弱引用对象，不管当前内存空间是否足够都会回收它的内存。也就是说这样的设计会很容易导致 ThreadLocal 对象被回收，线程所执行任务的时间长度是不固定的，这样的设计能够方便垃圾收集器回收线程私有的变量。

所以作者这样设计的目的是为了防止内存泄露，那怎么就变成了被很多文章所分析的是内存泄漏的导火索呢？这些文章的共同观点就是 key 被回收了，但是 value 是一个强引用没有被回收，这些 value 就变成了一个个的僵尸。这样的分析没有错，value 确实存在，且和线程是同生命周期的，但是如下策略可以保证尽量避免内存泄露：

> 1. ThreadLocal 在每次执行 get 和 set 操作的时候都会去清理 key 为 null 的 value 值
> 2. value 与线程同生命周期，线程死亡之时，也是 value 被 GC 之日

策略一没啥好说的，看看源码就知道，我们来举例验证一下策略二：

```
public class ThreadLocalWithMemoryLeak implements Callable<Boolean> {

    private class My50MB {

        private byte[] buffer = new byte[50 * 1024 * 1024];

        @Override
        protected void finalize() throws Throwable {
            super.finalize();
            System.out.println("gc my 50 mb");
        }
    }

    private class MyThreadLocal<T> extends ThreadLocal<T> {

        @Override
        protected void finalize() throws Throwable {
            super.finalize();
```

```
            System.out.println("gc my thread local");
        }
    }

    private MyThreadLocal<My50MB> threadLocal = new MyThreadLocal<>();
```

以上程序的最终输出如下：

```
Thread-11 is running
do gc
gc my thread local
sleep 60s
do gc
gc my 50 mb
```

可以看到 value 最终还是被 GC 了，虽然第一次 GC 的时候没有被回收，这也验证 value 和线程是同生命周期的，之所以示例中等待 60 秒是因为 `Executors.newCachedThreadPool()` 中的线程默认生命周期是 60 秒，如果生命周期内该线程没有被再次复用则会死亡，我们这里就是要等待线程死亡，一但线程死亡，value 也就被 GC 了。所以 **出现内存泄露的前提必须是持有 value 的线程一直存活** ，这在使用线程池时是很正常的，在这种情况下 value 一直不会被 GC，因为线程对象与 value 之间维护的是强引用。此外就是 **后续线程执行的业务一直没有调用 ThreadLocal 的 get 或 set 方法，导致不会主动去删除 key 为 null 的 value 对象** ，在满足这两个条件下 value 对象一直常驻内存，所以存在内存泄露的可能性。

那么我们应该怎么避免呢？前面我们分析过线程池情况下使用 ThreadLocal 存在小地雷，这里的内存泄露一般也都是发生在线程池的情况下，所以在使用 ThreadLocal 时，对于不再有效的 value 主动调用一下 remove 方法来进行清除，从而消除隐患，这也算是最佳实践吧。

本文最先发布于 " **指间数据** " 公众号，微信扫描下方二维码进行关注，第一时间获取高质量的技术类文章。

你可能感兴趣的

- **ThreadLocal** liaosilzu2007 thread java
- **ThreadLocal 内部实现和应用场景（慎用，可能有内存泄露）** defcon java
- **理解 ThreadLocal** bingchen java
- **第二章 线程安全性** william_lee java
- **java并发编程学习17--ThreadLocal** 极品公子 java
- **Java 线程相关类** 布still java 线程 线程安全
- **ThreadLocal类的简单使用** 尚学堂明辉 thread locale java
- **（基础系列）ThreadLocal的用法、原理和用途** 猿青木 java

评论

默认排序 | 时间排序

文明社会，理性评论

发布评论