目录

43 ThreadLocal 源码解析

更新时间：2019-11-27 10:55:55

> 辛苦是获得一切的定律。
>
> ——牛顿

引导语

ThreadLocal 提供了一种方式，让在多线程环境下，每个线程都可以拥有自己独特的数据，并且可以在整个线程执行过程中，从上而下的传递。

1 用法演示

可能很多同学没有使用过 ThreadLocal，我们先来演示下 ThreadLocal 的用法，demo 如下：

```
/**
 * ThreadLocal 中保存的数据是 Map
 */
static final ThreadLocal<Map<String, String>> context = new ThreadLocal<>();

@Test
public void testThread() {
  // 从上下文中拿出 Map
  Map<String, String> contextMap = context.get();
  if (CollectionUtils.isEmpty(contextMap)) {
    contextMap = Maps.newHashMap();
  }

  contextMap.put("key1", "value1");
  context.set(contextMap);
  log.info("key1， value1被放到上下文中");
        // 从上下文中拿出刚才放进去的数据
  getFromComtext();
}
```

```
    return value1;
}
//运行结果:
demo.ninth.ThreadLocalDemo - key1，value1被放到上下文中
demo.ninth.ThreadLocalDemo - 从 ThreadLocal 中取出上下文，key1 对应的值为：value1
```

从运行结果中可以看到，key1 对应的值已经从上下文中拿到了。

getFromComtext 方法是没有接受任何入参的，通过 context.get().get(“key1”) 这行代码就从上下文中拿到了 key1 的值，接下来我们一起来看下 ThreadLocal 底层是如何实现上下文的传递的。

2 类结构

2.1 类泛型

ThreadLocal 定义类时带有泛型，说明 ThreadLocal 可以储存任意格式的数据，源码如下：

```
public class ThreadLocal<T> {}
```

2.2 关键属性

ThreadLocal 有几个关键属性，我们一一看下：

```
// threadLocalHashCode 表示当前 ThreadLocal 的 hashCode，用于计算当前 ThreadLocal 在 Th
private final int threadLocalHashCode = nextHashCode();
// 计算 ThreadLocal 的 hashCode 值(就是递增)
private static int nextHashCode() {
    return nextHashCode.getAndAdd(HASH_INCREMENT);
}
// static + AtomicInteger 保证了在一台机器中每个 ThreadLocal 的 threadLocalHashCode 是唯一
// 被 static 修饰非常关键，因为一个线程在处理业务的过程中，ThreadLocalMap 是会被 set 多个 T
private static AtomicInteger nextHashCode = new AtomicInteger();
```

还有一个重要属性：ThreadLocalMap，当一个线程有多个 ThreadLocal 时，需要一个容器来管理多个 ThreadLocal，ThreadLocalMap 的作用就是这个，管理线程中多个 ThreadLocal。

2.2.1 ThreadLocalMap

ThreadLocalMap 本身就是一个简单的 Map 结构，key 是 ThreadLocal，value 是 ThreadLocal 保存的值，底层是数组的数据结构，源码如下：

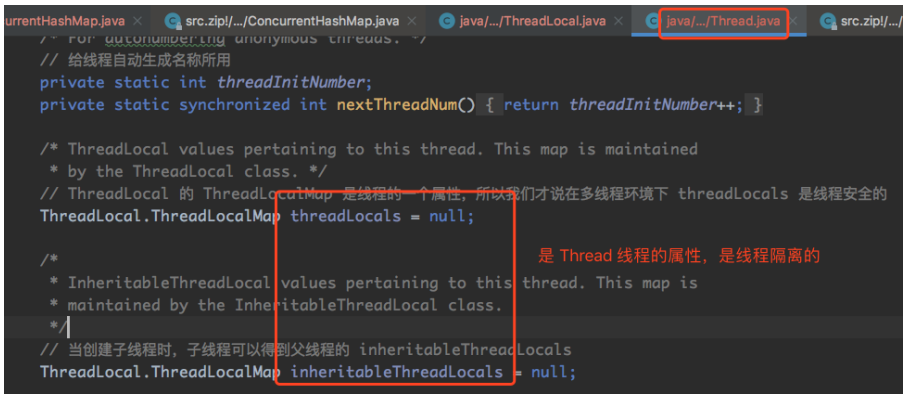
```
static class ThreadLocalMap {
        // 数组中的每个节点值，WeakReference 是弱引用，当没有引用指向时，会直接被回收
        static class Entry extends WeakReference<ThreadLocal<?>> {
            // 当前 ThreadLocal 关联的值
            Object value;
            // WeakReference 的引用 referent 就是 ThreadLocal
            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }
```

```
    private Entry[] table;
    // 扩容的阈值，默认是数组大小的三分之二
    private int threshold;
}
```

从源码中看到 ThreadLocalMap 其实就是一个简单的 Map 结构，底层是数组，有初始化大小，也有扩容阈值大小，数组的元素是 Entry，Entry 的 key 就是 ThreadLocal 的引用，value 是 ThreadLocal 的值。

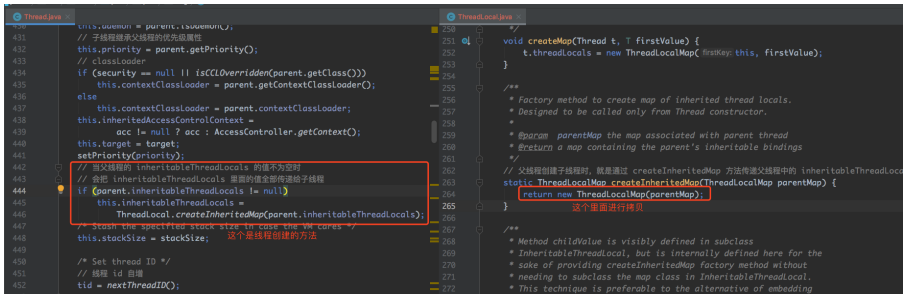
3 ThreadLocal 是如何做到线程之间数据隔离的

ThreadLocal 是线程安全的，我们可以放心使用，主要因为是 ThreadLocalMap 是线程的属性，我们看下线程 Thread 的源码，如下：



从上图中，我们可以看到 ThreadLocals.ThreadLocalMap 和 InheritableThreadLocals.ThreadLocalMap 分别是线程的属性，所以每个线程的 ThreadLocals 都是隔离独享的。

父线程在创建子线程的情况下，会拷贝 inheritableThreadLocals 的值，但不会拷贝 threadLocals 的值，源码如下：



从上图中我们可以看到，在线程创建时，会把父线程的 inheritableThreadLocals 属性值进行拷贝。

4 set 方法

set 方法的主要作用是往当前 ThreadLocal 里面 set 值，假如当前 ThreadLocal 的泛型是 Map，那么就是往当前 ThreadLocal 里面 set map，源码如下：

```
// set 操作每个线程都是串行的，不会有线程安全的问题
public void set(T value) {
    Thread t = Thread.currentThread();
```

```
        map.set(this, value);
    // 初始化ThreadLocalMap
    else
        createMap(t, value);
}
```

代码逻辑比较清晰，我们在一起来看下 ThreadLocalMap.set 的源码，如下：

```
private void set(ThreadLocal<?> key, Object value) {
    Entry[] tab = table;
    int len = tab.length;
    // 计算 key 在数组中的下标，其实就是 ThreadLocal 的 hashCode 和数组大小-1取余
    int i = key.threadLocalHashCode & (len-1);

    // 整体策略：查看 i 索引位置有没有值，有值的话，索引位置 + 1，直到找到没有值的位置
    // 这种解决 hash 冲突的策略，也导致了其在 get 时查找策略有所不同，体现在 getEntryAfterMi
    for (Entry e = tab[i];
         e != null;
         // nextIndex 就是让在不超过数组长度的基础上，把数组的索引位置 + 1
         e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get();
        // 找到内存地址一样的 ThreadLocal，直接替换
        if (k == key) {
            e.value = value;
            return;
        }
        // 当前 key 是 null，说明 ThreadLocal 被清理了，直接替换掉
        if (k == null) {
            replaceStaleEntry(key, value, i);
            return;
        }
    }
    // 当前 i 位置是无值的，可以被当前 thradLocal 使用
    tab[i] = new Entry(key, value);
    int sz = ++size;
    // 当数组大小大于等于扩容阈值(数组大小的三分之二)时，进行扩容
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash();
}
```

上面源码我们注意几点：

1. 是通过递增的 AtomicInteger 作为 ThreadLocal 的 hashCode 的；
2. 计算数组索引位置的公式是：hashCode 取模数组大小，由于 hashCode 不断自增，所以不同的 hashCode 大概率上会计算到同一个数组的索引位置（但这个不用担心，在实际项目中，ThreadLocal 都很少，基本上不会冲突）；
3. 通过 hashCode 计算的索引位置 i 处如果已经有值了，会从 i 开始，通过 +1 不断的往后寻找，直到找到索引位置为空的地方，把当前 ThreadLocal 作为 key 放进去。

好在日常工作中使用 ThreadLocal 时，常常只使用 1~2 个 ThreadLocal，通过 hash 计算出重复的数组的概率并不是很大。

set 时的解决数组元素位置冲突的策略，也对 get 方法产生了影响，接着我们一起来看一下 get 方法。

get 方法主要是从 ThreadLocalMap 中拿到当前 ThreadLocal 储存的值，源码如下：

```
public T get() {
    // 因为 threadLocal 属于线程的属性，所以需要先把当前线程拿出来
    Thread t = Thread.currentThread();
    // 从线程中拿到 ThreadLocalMap
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        // 从 map 中拿到 entry，由于 ThreadLocalMap 在 set 时的 hash 冲突的策略不同，导致拿的
        ThreadLocalMap.Entry e = map.getEntry(this);
        // 如果不为空，读取当前 ThreadLocal 中保存的值
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    // 否则给当前线程的 ThreadLocal 初始化，并返回初始值 null
    return setInitialValue();
}
```

接着我们来看下 ThreadLocalMap 的 getEntry 方法，源码如下：

```
// 得到当前 thradLocal 对应的值，值的类型是由 thradLocal 的泛型决定的
// 由于 thradLocalMap set 时解决数组索引位置冲突的逻辑，导致 thradLocalMap get 时的逻辑也
// 首先尝试根据 hashcode 取模数组大小-1 = 索引位置 i 寻找，找不到的话，自旋把 i+1，直到找到
private Entry getEntry(ThreadLocal<?> key) {
    // 计算索引位置：ThreadLocal 的 hashCode 取模数组大小-1
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];
    // e 不为空，并且 e 的 ThreadLocal 的内存地址和 key 相同，直接返回，否则就是没有找到，继
    if (e != null && e.get() == key)
        return e;
    else
    // 这个取数据的逻辑，是因为 set 时数组索引位置冲突造成的
        return getEntryAfterMiss(key, i, e);
}
```

```
// 自旋 i+1，直到找到为止
private Entry getEntryAfterMiss(ThreadLocal<?> key, int i, Entry e) {
    Entry[] tab = table;
    int len = tab.length;
    // 在大量使用不同 key 的 ThreadLocal 时，其实还蛮耗性能的
    while (e != null) {
        ThreadLocal<?> k = e.get();
        // 内存地址一样，表示找到了
        if (k == key)
            return e;
        // 删除没用的 key
        if (k == null)
            expungeStaleEntry(i);
        // 继续使索引位置 + 1
        else
            i = nextIndex(i, len);
        e = tab[i];
    }
    return null;
```

get 逻辑源码中注释已经写的很清楚了，我们就不重复说了。

6 扩容

ThreadLocalMap 中的 ThreadLocal 的个数超过阈值时，ThreadLocalMap 就要开始扩容了，我们一起来看下扩容的逻辑：

```
//扩容
private void resize() {
    // 拿出旧的数组
    Entry[] oldTab = table;
    int oldLen = oldTab.length;
    // 新数组的大小为老数组的两倍
    int newLen = oldLen * 2;
    // 初始化新数组
    Entry[] newTab = new Entry[newLen];
    int count = 0;
    // 老数组的值拷贝到新数组上
    for (int j = 0; j < oldLen; ++j) {
        Entry e = oldTab[j];
        if (e != null) {
            ThreadLocal<?> k = e.get();
            if (k == null) {
                e.value = null; // Help the GC
            } else {
                // 计算 ThreadLocal 在新数组中的位置
                int h = k.threadLocalHashCode & (newLen - 1);
                // 如果索引 h 的位置值不为空，往后+1，直到找到值为空的索引位置
                while (newTab[h] != null)
                    h = nextIndex(h, newLen);
                // 给新数组赋值
                newTab[h] = e;
                count++;
            }
        }
    }
    // 给新数组初始化下次扩容阈值，为数组长度的三分之二
    setThreshold(newLen);
    size = count;
    table = newTab;
}
```

源码注解也比较清晰，我们注意两点：

1. 扩容后数组大小是原来数组的两倍；
2. 扩容时是绝对没有线程安全问题的，因为 ThreadLocalMap 是线程的一个属性，一个线程同一时刻只能对 ThreadLocalMap 进行操作，因为同一个线程执行业务逻辑必然是串行的，那么操作 ThreadLocalMap 必然也是串行的。

7 总结

ThreadLocal 是非常重要的 API，我们在写一个中间件的时候经常会用到，比如说流程引擎中上下文的传递，调用链ID的传递等等，非常好用，但坑也很多。

目录

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论

千学不如一看，千看不如一练