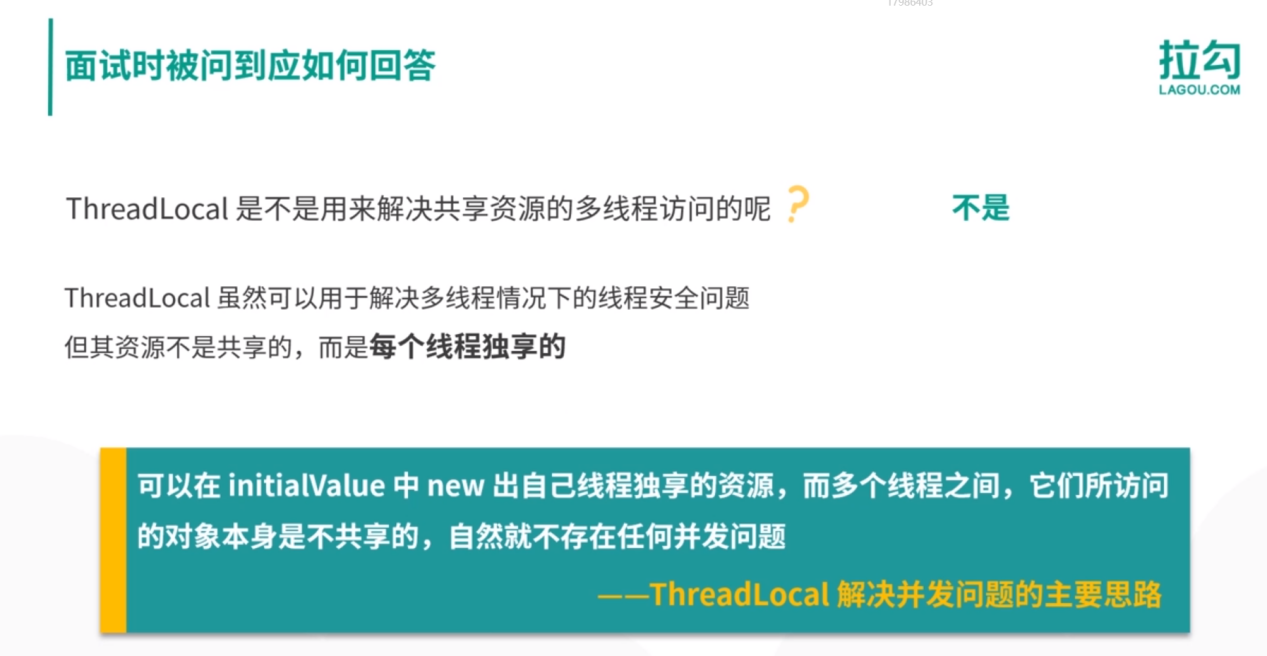
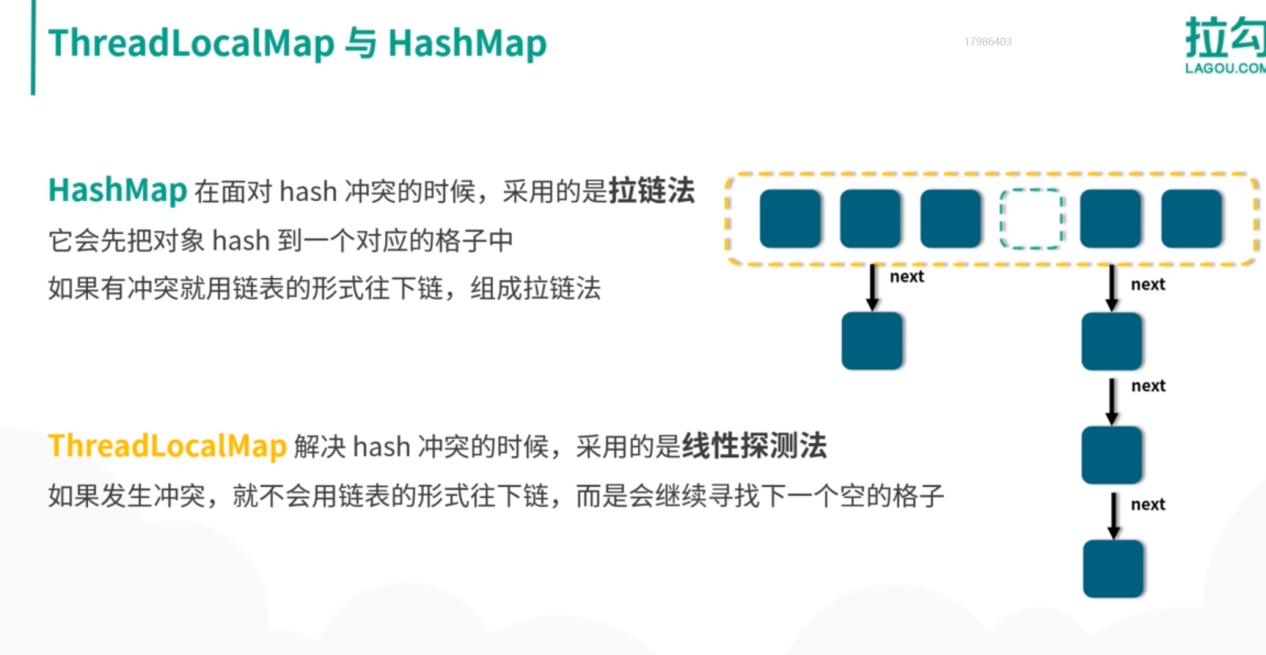
1. **ThreadLocal 适用场景**
2. ThreadLocal 用作保存每个线程独享的对象，为每个线程都创建一个副本，每个线程都只能修改自己所拥有的副本, 而不会影响其他线程的副本，这样就让原本在并发情况下，线程不安全的情况变成了线程安全的情况。
3. ThreadLocal 用作每个线程内需要独立保存信息的场景，供其他方法更方便得获取该信息，每个线程获取到的信息都可能是不一样的，前面执行的方法设置了信息后，后续方法可以通过 ThreadLocal 直接获取到，避免了传参。
4. **ThreadLocal 和 synchronized 是什么关系**

* ThreadLocal 是通过让每个线程独享自己的副本，避免了资源的竞争。
* synchronized 主要用于临界资源的分配，在同一时刻限制最多只有一个线程能访问该资源。

1. **ThreadLocal 解决共享资源的多线程访问？**

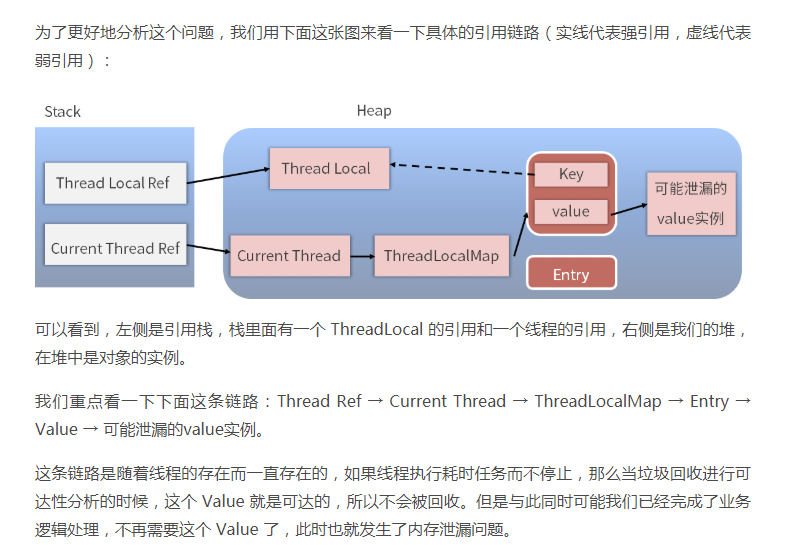


1. **ThreadLocalMap**

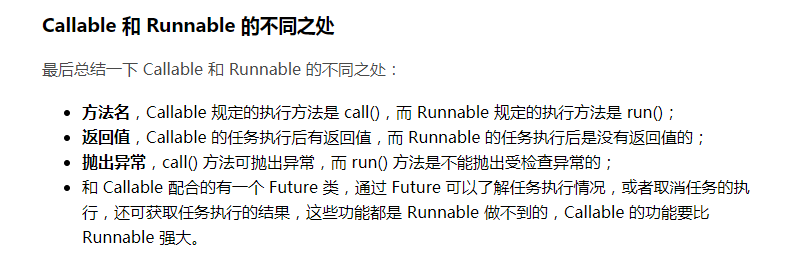


1. **为何每次用完 ThreadLocal 都要调用 remove()？**

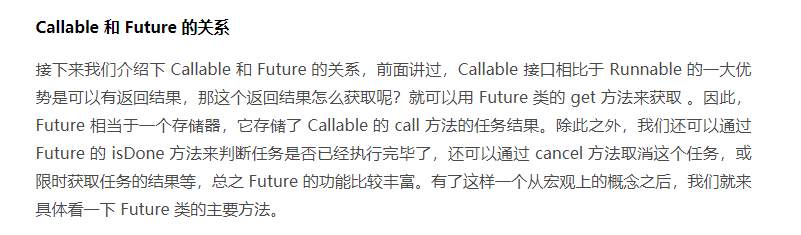
内存泄漏：



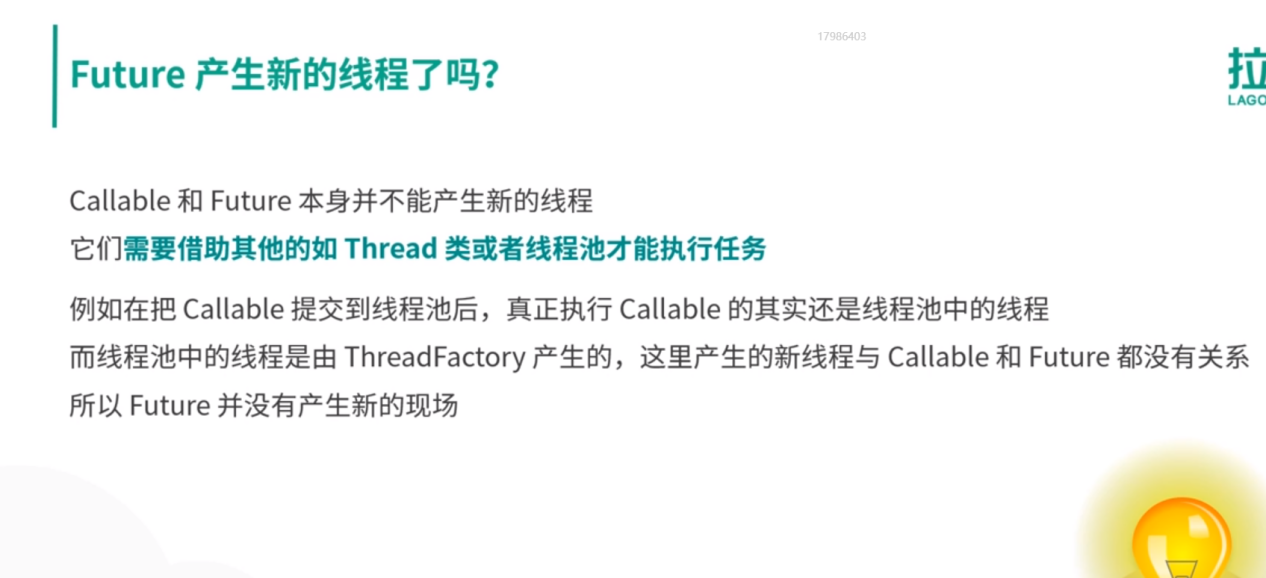
1. **Callable 和 Runnable 的不同之处**



1. **Callable 和 Future 的关系**



1. **Future产生新的线程了吗？**



1. **线程并行协作**

1、CountDownLatch

如果有一个请求特别慢，相当于有一个线程没有执行 countDown 方法，来不及在 3 秒钟之内执行完毕，

那么这个带超时参数的 await 方法也会在 3 秒钟到了以后，及时地放弃这一次等待，于是就把 prices 给返回了。

所以这样一来，我们就利用 CountDownLatch 实现了这个需求，也就是说我们最多等 3 秒钟，

但如果在 3 秒之内全都返回了，我们也可以快速地去返回，不会傻等，提高了效率。

代码：

public class CountDownLatchDemo {

ExecutorService threadPool = Executors.newFixedThreadPool(3);

public static void main(String[] args) throws InterruptedException {

CountDownLatchDemo countDownLatchDemo = new CountDownLatchDemo();

System.out.println(countDownLatchDemo.getPrices());

}

private Set<Integer> getPrices() throws InterruptedException {

Set<Integer> prices = Collections.synchronizedSet(new HashSet<Integer>());

CountDownLatch countDownLatch = new CountDownLatch(3);

threadPool.submit(new Task(123, prices, countDownLatch));

threadPool.submit(new Task(456, prices, countDownLatch));

threadPool.submit(new Task(789, prices, countDownLatch));

countDownLatch.await(3, TimeUnit.SECONDS);

return prices;

}

private class Task implements Runnable {

Integer productId;

Set<Integer> prices;

CountDownLatch countDownLatch;

public Task(Integer productId, Set<Integer> prices,

CountDownLatch countDownLatch) {

this.productId = productId;

this.prices = prices;

this.countDownLatch = countDownLatch;

}

@Override

public void run() {

int price = 0;

try {

Thread.sleep((long) (Math.random() \* 4000));

price = (int) (Math.random() \* 4000);

} catch (InterruptedException e) {

e.printStackTrace();

}

prices.add(price);

countDownLatch.countDown();//相当于把计数减 1

}

}

}

2、CompletableFuture

如果在 3 秒钟之内这 3 个任务都可以顺利返回，也就是这个任务包括的那三个任务，

每一个都执行完毕的话，则这个 get 方法就可以及时正常返回，并且往下执行，

相当于执行到 return prices。在下面的这个 Task 的 run 方法中，该方法如果执行完毕的话，

对于 CompletableFuture 而言就意味着这个任务结束，它是以这个作为标记来判断任务是不是执行完毕的。

但是如果有某一个任务没能来得及在 3 秒钟之内返回，

那么这个带超时参数的 get 方法便会抛出 TimeoutException 异常，同样会被我们给 catch 住。

这样一来它就实现了这样的效果：会尝试等待所有的任务完成，但是最多只会等 3 秒钟，

在此之间，如及时完成则及时返回。

代码：

public class CompletableFutureDemo {

private final CompletableFuture<Void> completableFuture = new CompletableFuture<Void>();

public static void main(String[] args)

throws Exception {

CompletableFutureDemo completableFutureDemo = new CompletableFutureDemo();

System.out.println(completableFutureDemo.getPrices());

}

private Set<Integer> getPrices() {

Set<Integer> prices = Collections.synchronizedSet(new HashSet<Integer>());

CompletableFuture<Void> task1 = CompletableFuture.runAsync(new Task(123, prices));

CompletableFuture<Void> task2 = CompletableFuture.runAsync(new Task(456, prices));

CompletableFuture<Void> task3 = CompletableFuture.runAsync(new Task(789, prices));

CompletableFuture<Void> allTasks = CompletableFuture.allOf(task1, task2, task3);

try {

allTasks.get(3, TimeUnit.SECONDS);

} catch (InterruptedException e) {

} catch (ExecutionException e) {

} catch (TimeoutException e) {

}

return prices;

}

private class Task implements Runnable {

Integer productId;

Set<Integer> prices;

public Task(Integer productId, Set<Integer> prices) {

this.productId = productId;

this.prices = prices;

}

@Override

public void run() {

int price = 0;

try {

Thread.sleep((long) (Math.random() \* 4000));

price = (int) (Math.random() \* 4000);

} catch (InterruptedException e) {

e.printStackTrace();

}

prices.add(price);

}

}

}