24 举一反三：队列在 Java 其它源码中的应用

更新时间：2019-10-24 12:03:24

> 世上无难事,只要肯登攀。
>
> ——毛泽东

引导语

队列除了提供 API 供开发者使用外，自身也和 Java 中其他 API 紧密结合，比如线程池和锁，线程池直接使用了队列的 API，锁借鉴了队列的思想，重新实现了队列，线程池和锁都是我们工作中经常使用的 API，也是面试官常问的 API，队列在两者的实现上发挥着至关重要的作用，接下来我们一起来看下。

1 队列和线程池的结合

1.1 队列在线程池中的作用

线程池大家应该都使用过，比如我们想新建一个固定大小的线程池，并让运行的线程打印一句话出来，我们会这么写代码：

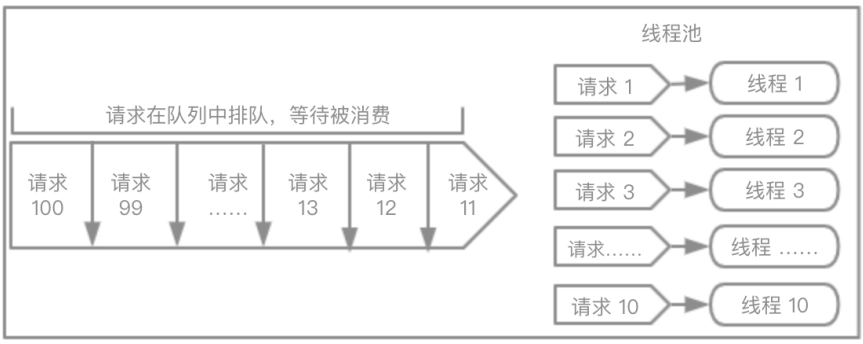
```
ExecutorService executorService = Executors.newFixedThreadPool(10);
// submit 是提交任务的意思
// Thread.currentThread() 得到当前线程
executorService.submit(() -> System.out.println(Thread.currentThread().getName() + " is run"));
// 打印结果(我们打印出了当前线程的名字)：
pool-1-thread-1 is run
```

代码中的 Executors 是并发的工具类，主要是为了帮助我们更方便的构造线程池的，其中 newFixedThreadPool 方法表示会构造出固定大小的线程池，我们给的入参是 10，代表线程池最大可以构造 10 个线程出来。

呢?

这时候就需要队列出马了，我们会把线程无法消化的数据放到队列中去，让数据在队列中排队，等线程有能力消费了，再从队列中拿出来慢慢去消费。

我们画一个图释义一下：



上图右边表示 10 个线程正在全力消费请求，左边表示剩余请求正在队列中排队，等待消费。

由此可见，队列在线程池中占有很重要的地位，当线程池中的线程忙不过来的时候，请求都可以在队列中等待，从而慢慢地消费。

接下来我们来看下，线程池到底用到了那几种队列类型，分别起的什么作用。

1.1 线程池中使用到的队列的类型

1.1.1 LinkedBlockingQueue 队列的使用

刚刚我们说的 newFixedThreadPool 是一种固定大小的线程池，意思是当线程池初始化好后，线程池里面的线程大小是不会变的了（线程池默认设置是不会回收核心线程数的），我们来看下 newFixedThreadPool 的源码：

```
// ThreadPoolExecutor 初始化时，第一个参数表示 coreSize，第二个参数是 maxSize，coreSize =
// 表示线程池初始化时，线程大小已固定，所以叫做固定(Fixed)线程池。
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

源码中可以看到初始化了 ThreadPoolExecutor，ThreadPoolExecutor 是线程池的 API，我们在线程池章节会细说，它的第五个构造参数就是队列，线程池根据场景会选择不同的队列，此处使用的是 LinkedBlockingQueue，并且是默认参数的 Queue，这说明此阻塞队列的最大容量是 Integer 的最大值，也就是说当线程池的处理能力有限时，阻塞队列中最大可以存放 Integer 最大值个任务。

但我们在实际工作中，常常不建议直接使用 newFixedThreadPool，主要是因为其使用的是 LinkedBlockingQueue 的默认构造器，队列容量太大了，在要求实时响应的请求中，队列容量太大往往危害也很大。

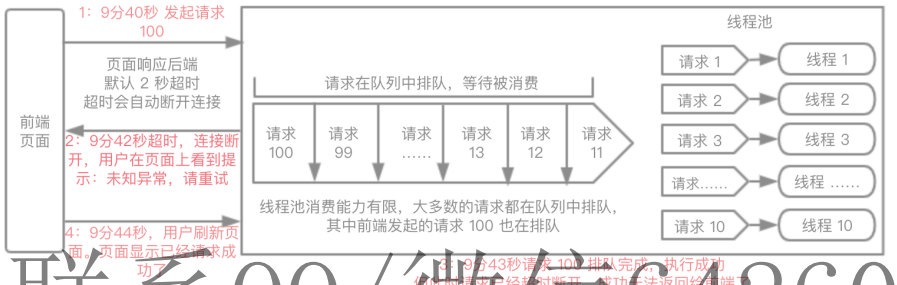
10 个线程仍然在不断地消费，但需要消费完队列中的所有数据是需要时间的，假设需要 3 秒才能全部消费完，而这些实时请求都是有超时时间的，默认超时时间是 2 秒，当时间到达 2 秒时，请求已经超时了，返回报错，可这时候队列中的任务还有很多都在等待消费呢，即使后来消费完成，也无法返回给调用方了。

以上情况就会造成，调用方看到接口是超时报错返回的，但服务端的任务其实还在排队执行，过了 3 秒后，服务端的任务可能都会执行成功，但调用方已经无法感知了，调用方再次调用时，就会发现其实这笔请求已经成功了。

如果调用方是从页面发起的，那么体验就会更差，页面上第一次调用页面报错，用户重新刷新页面时，页面显示上次的请求已经成功了，这个就是很不好的体验了。

所以我们希望队列的大小不要设置成那么大，可以根据实际的消费情况来设置队列的大小，这样就可以保证在接口超时前，队列中排队的请求可以执行完。

场景比较复杂，为了方便理解，我们画了一个图，把整个流程释义一下：



这种问题，在实际工作中已经属于非常严重的生产事故了，我们使用时一定要小心。

和 newFixedThreadPool 相同的是，newSingleThreadExecutor 方法底层使用的也是 LinkedBlockingQueue，newSingleThreadExecutor 线程池底层线程只会有一个，这代表着这个线程池一次只能处理一个请求，其余的请求都会在队列中排队等待执行，我们看下 newSingleThreadExecutor 的源码实现：

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        // 前两个参数规定了这个线程池一次只能消费一个线程
        // 第五个参数使用的是 LinkedBlockingQueue,说明当请求超过单线程消费能力时，就会排队
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

可以看到，底层使用的也是 LinkedBlockingQueue 的默认参数，也就是说排队的最大值是 Integer 的最大值。

1.1.2 SynchronousQueue 队列

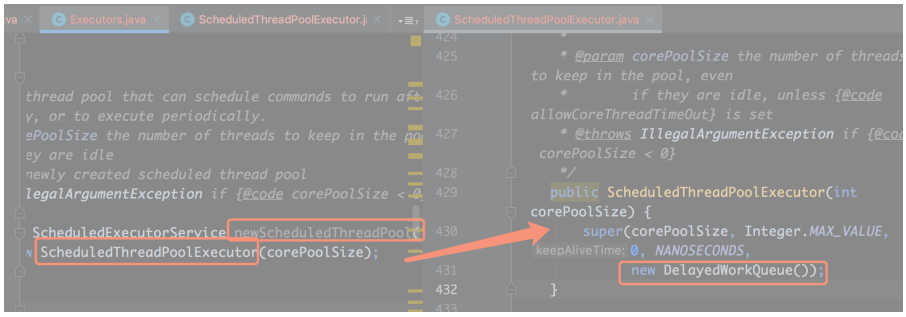
除了 newFixedThreadPool 方法，在线程池新建时，还有其他的几个方法也对应着不同的队列，我们一起来看下 newCachedThreadPool，newCachedThreadPool 底层对应的是 SynchronousQueue 队列，源码如下：

```
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

SynchronousQueue 队列是没有大小限制的，请求多少队列都能承受的住，可以说这是他的优点，缺点就是每次往队列里面 put 数据时，并不能立马返回，而是需要等待有线程 take 数据之后，才能正常返回，如果请求量大，而消费能力较差时，就会导致大量请求被 hodler 住，必须等到慢慢消费完成之后才能被释放，所以在平时工作使用中也需要慎重。

1.1.3 DelayedWorkQueue

newScheduledThreadPool 代表定时任务线程池，底层源码如下：



截图从左往右我们可以看到，底层队列使用的是 DelayedWorkQueue 延迟队列，说明线程池底层延时的功能就是 DelayedWorkQueue 队列提供的，新的延迟请求都先到队列中去，延迟时间到了，线程池自然就能从队列中拿出线程进行执行了。

newSingleThreadScheduledExecutor 方法也是和 newScheduledThreadPool 一样的，使用 DelayedWorkQueue 的延迟功能，只不过前者是单个线程执行。

1.2 小结

从线程池的源码中，我们可以看到：

1. 队列在线程池的设计中，起着缓冲数据，延迟执行数据的作用，当线程池消费能力有限时，可以让请求进行排队，让线程池可以慢慢消费。
2. 线程池根据不同的场景，选择使用了 DelayedWorkQueue、SynchronousQueue、LinkedBlockingQueue 多种队列，从而实现自己不同的功能，比如使用 DelayedWorkQueue 的延迟功能来实现定时执行线程池。

2 队列和锁的结合

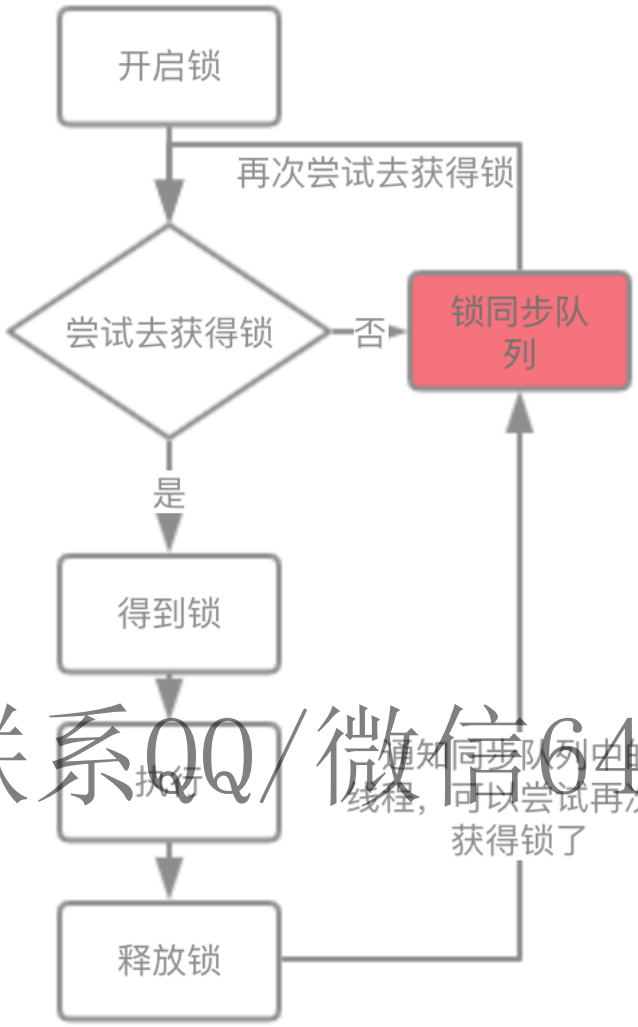
我们平时写锁代码的时候都这么写：

```
ReentrantLock lock = new ReentrantLock();
try{
    lock.lock();
    // do something
}catch(Exception e){
  //throw Exception;
}finally {
    lock.unlock();
}
```

目录

一个线程才能获得锁的，那么此时其他获取不到锁的线程该怎么办呢？

等待，其他获取不到锁的线程，都会到一个等待队列中去等待，等待锁被释放掉时，再去竞争锁，我们画一个示意图。



图中红色标识的就是同步队列，获取不到锁的线程都会到同步队列中去排队，当锁被释放后，同步队列中的线程就又开始去竞争锁。

可以看出队列在锁中起的作用之一，就是帮助管理获取不到锁的线程，让这些线程可以耐心的等待。

同步队列并没有使用现有的队列的 API 去实现，但底层的结构，思想和目前队列是一致的，所以我们学好队列章节，对理解锁的同步队列，用处非常大。

3 总结

队列的数据结构真的很重要，在线程池和锁两个重量级 API 中起着非常重要的作用，我们要非常清楚队列底层的大体的数据结构，了解数据是如何入队的，如何出队的，队列这章也是比较复杂的，建议大家多多 debug，我们 github 上也提供了一些 debug 的 demo，大家可以尝试调试起来。

目录

精选留言 1

欢迎在这里发表留言，作者筛选后可公开显示

慕粉3445147

之前面试的时候,面试官问过对于不同的场景使用何种队列,现在回想一下,答得太简单了,也是面试官给面子,里面的想法真的很多啊..

0 回复 2019-11-04

文贺 回复 **慕粉3445147**

是滴，这个问题其实比较考察我们对队列的使用经验。

回复 2019-11-05 20:31:36

千学不如一看，千看不如一练