

【讲义】第3讲：并发容器和线程池

一、并发容器

1.1> 概述

1.2> ConcurrentHashMap

1.3> ConcurrentLinkedQueue

1.3.1> add

1.3.2> remove

1.3.3> 哨兵

1.4> CopyOnWriteArrayList

1.5> BlockingQueue

1.5.1> ArrayBlockingQueue

1.5.2> LinkedBlockingQueue

1.5.3> SynchronousQueue

二、线程池

2.1> ThreadPoolExecutor

2.2> ForkJoinPool

2.2.1> 概述

2.2.2> RecursiveTask执行有返回值任务

2.2.1> RecursiveAction执行无返回值任务

三、Future

3.1> FutureTask

3.2> CompletableFuture

3.2.1> 执行通知

3.2.2> 执行异步任务

一、并发容器

1.1> 概述

- JDK提供的这些容器大部分在java.util.concurrent包中。我们挑选出一些比较有代表性的并发容器

类，来感受一下JDK自带的并发集合带来的“快感”。

并发集合	说明	对应普通集合
ConcurrentHashMap	线程安全且高效的HashMap	HashMap
CopyOnWriteArrayList	在 读多写少 的场合，这个List的性能远远好于Vector	ArrayList
ConcurrentLinkedQueue	高效的并发队列，使用 链表 实现，线程安全的LinkedList	LinkedList
BlockingQueue	这是一个接口，JDK内部通过链表、数组等方式实现了这个接口。作为 数据共享通道	阻塞队列
ConcurrentSkipListMap	使用 跳表 的数据结构进行快速查找	Map

1.2> ConcurrentHashMap

- 详情请见：[【源码解析】ConcurrentHashMap.pdf](#)

1.3> ConcurrentLinkedQueue

- ConcurrentLinkedQueue是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。
- ConcurrentLinkedQueue算是在高并发环境中性能最好的队列。底层由单向链表组成，每个节点结构如下所示：

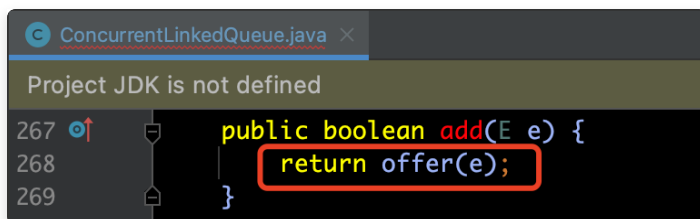
Node
volatile E item; volatile Node<E> next;

- 构造函数中，创建了一个空节点作为链表中的第一个Node节点

```
public ConcurrentLinkedQueue() {  
    head = tail = new Node<E>(null);  
}
```

1.3.1> add

- 向容器中添加元素，源码如下所示：



```
ConcurrentLinkedQueue.java x  
Project JDK is not defined  
267 public boolean add(E e) {  
268     return offer(e);  
269 }  
270
```

```
ConcurrentLinkedQueue.java x
Project JDK is not defined Setup S

297 public boolean offer(E e) {
298     checkNotNull(e);
299     final Node<E> newNode = new Node<E>(e);
300
301     for (Node<E> t = tail, p = t;;) {
302         Node<E> q = p.next;
303
304         /** case1: p是最后一个节点 */
305         if (q == null) {
306             if (p.casNext(null, newNode)) {
307                 // 每2次, 更新tail
308                 if (p != t) {
309                     casTail(t, newNode);
310                 }
311                 return true;
312             }
313         }
314
315         /** case2: 遇到哨兵节点 (即: next指向自己节点), 从head开始遍历。但如果tail被修改, 则使用tail (因为可能被修改正确了) */
316         else if (p == q) {
317             p = (t != (t = tail)) ? t : head;
318         }
319
320         /** case3: 取tail节点 (t), 还是tail的next节点 (q) */
321         else {
322             p = (p != t && t != (t = tail)) ? t : q;
323         }
324     }
325 }
```

- 关于 `t != (t = tail)` 的判断, 首先, “`!=`” 并不是原子操作, 它是可以被中断的。也就是说, 在执行 “`!=`” 时, 会先取得 `t` 的值, 再执行 `t=tail`, 并取得新的 `t` 值。然后比较这两个值是否相等。下面例子演示了这种情况:

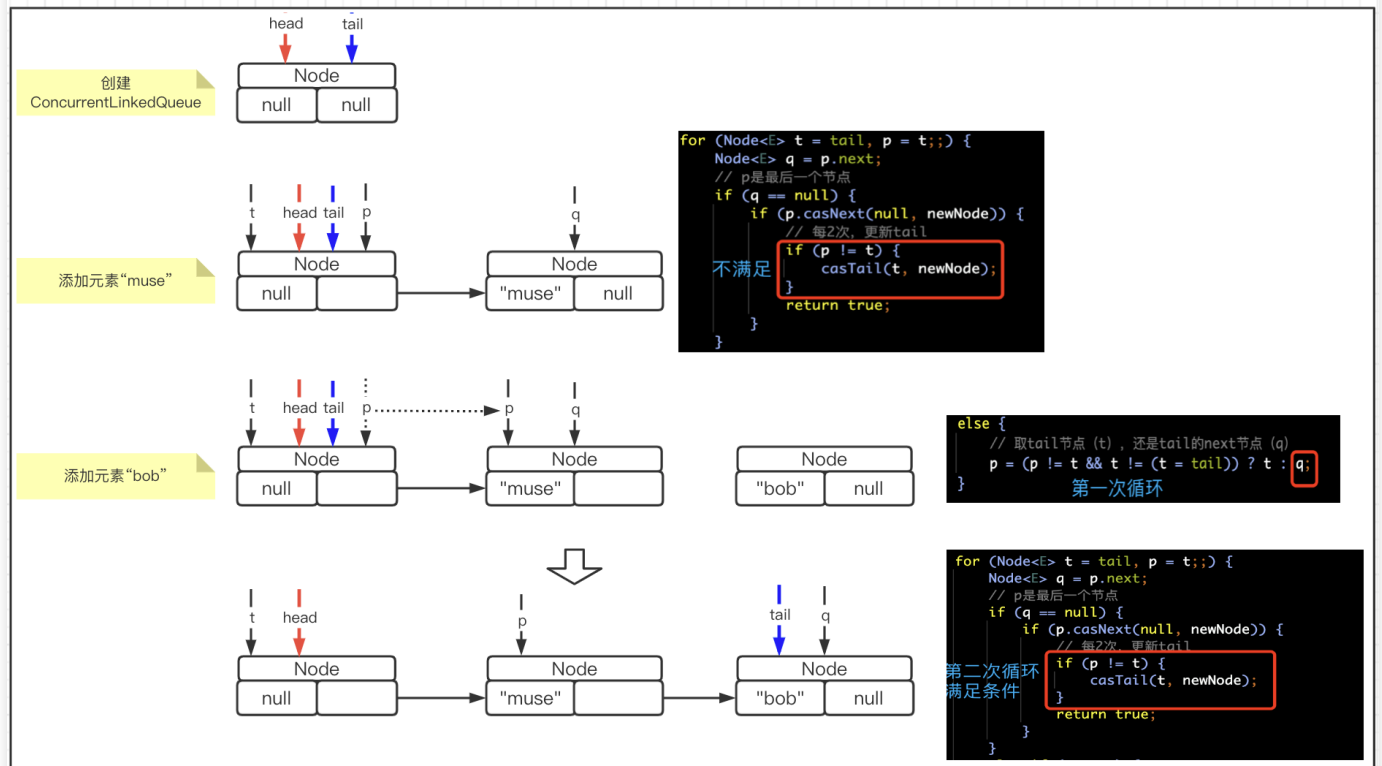
```
ConcurrentLinkedQueueDemo.java x
5 public class ConcurrentLinkedQueueDemo {
6     private static volatile int tail = 0;
7
8     public static void main(String[] args) {
9         int num = 0;
10        while (num < 2) {
11            new TaskThread().start();
12            num++;
13        }
14    }
15
16    public static class TaskThread extends Thread {
17        @SneakyThrows
18        @Override
19        public void run() {
20            int t = tail++;
21            System.out.println(getName() + " t=" + t);
22            Thread.sleep(10);
23            /**
24             * “!=”并不是原子操作，它可以被中断的。
25             * 也就是说，在执行“!=”时，会先取得t的值，再执行t=tail，并取得新的t值。然后比较这两个值是否相等。
26             */
27            if (t != (t = tail)) {
28                System.out.println(getName() + " t!=t, t=" + t);
29            }
30        }
31    }
32 }
```

Run: ConcurrentLinkedQueueDemo x

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
Thread-0 t=0
Thread-1 t=1
Thread-0 t!=t, t=2
Thread-1 t!=t, t=2
Process finished with exit code 0
```

- 添加节点如下所示：

场景1: 添加元素



【解释】

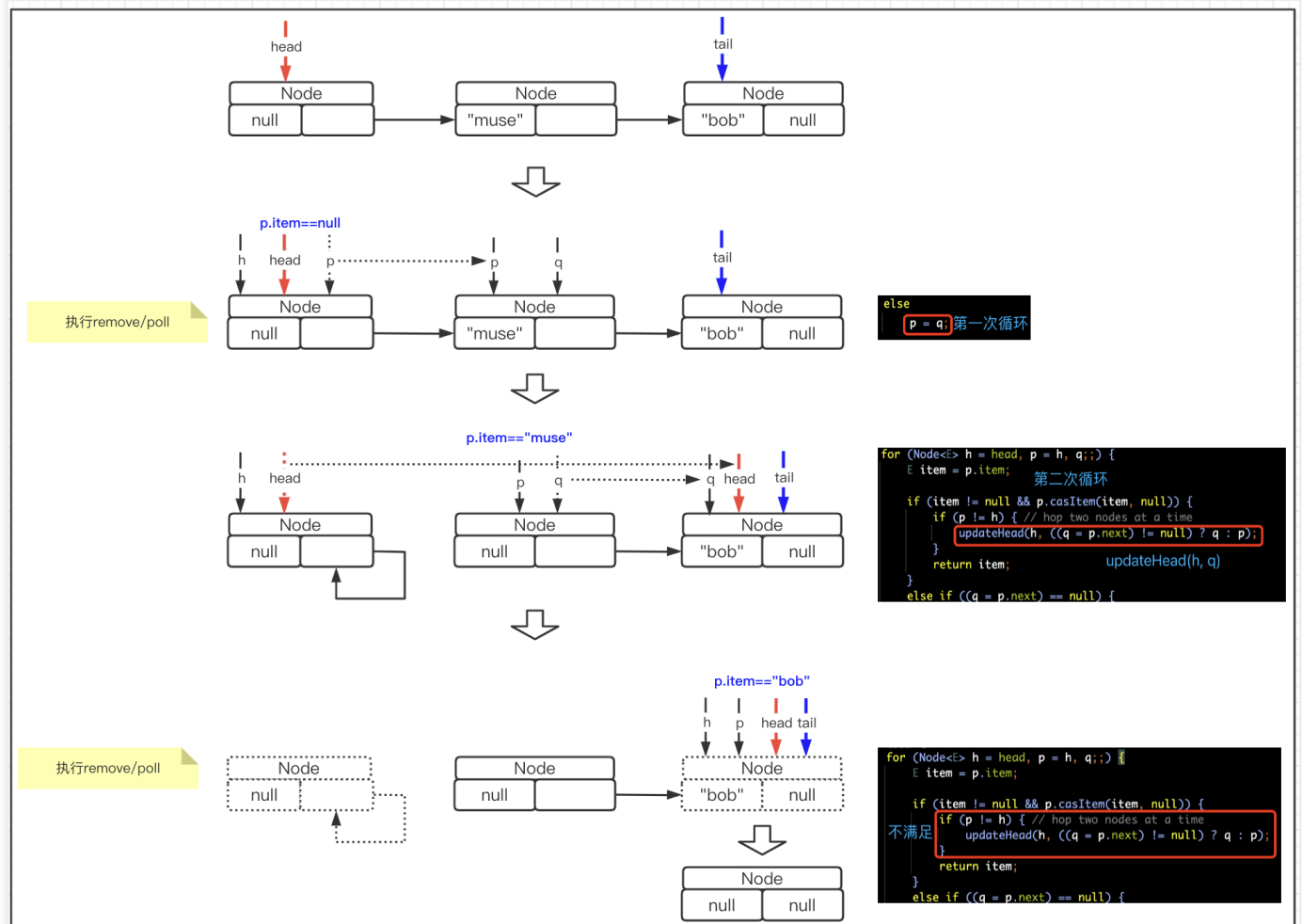
- 我们从上面的图可以看到, 对tail的更新是会产生滞后的, 也就是每次更新都会跳跃两个元素。这么做的目的, 就是为了减少cas操作的次数。例如, 我们完全可以在上述代码中通过 `if(p.casNext(null, newNode) && casTail(t, newNode))` 这种方式, 保证最新节点拼接到链表末尾, 并且tail指针永远指向末尾, 但是, 由于CAS一般都用在无限自旋的场景中, 那么对于效率的损耗就比较大了。而通过两次操作才更新一次tail, 可以有效减少性能消耗。

1.3.2> remove

- 删除元素操作, 源码如下所示:

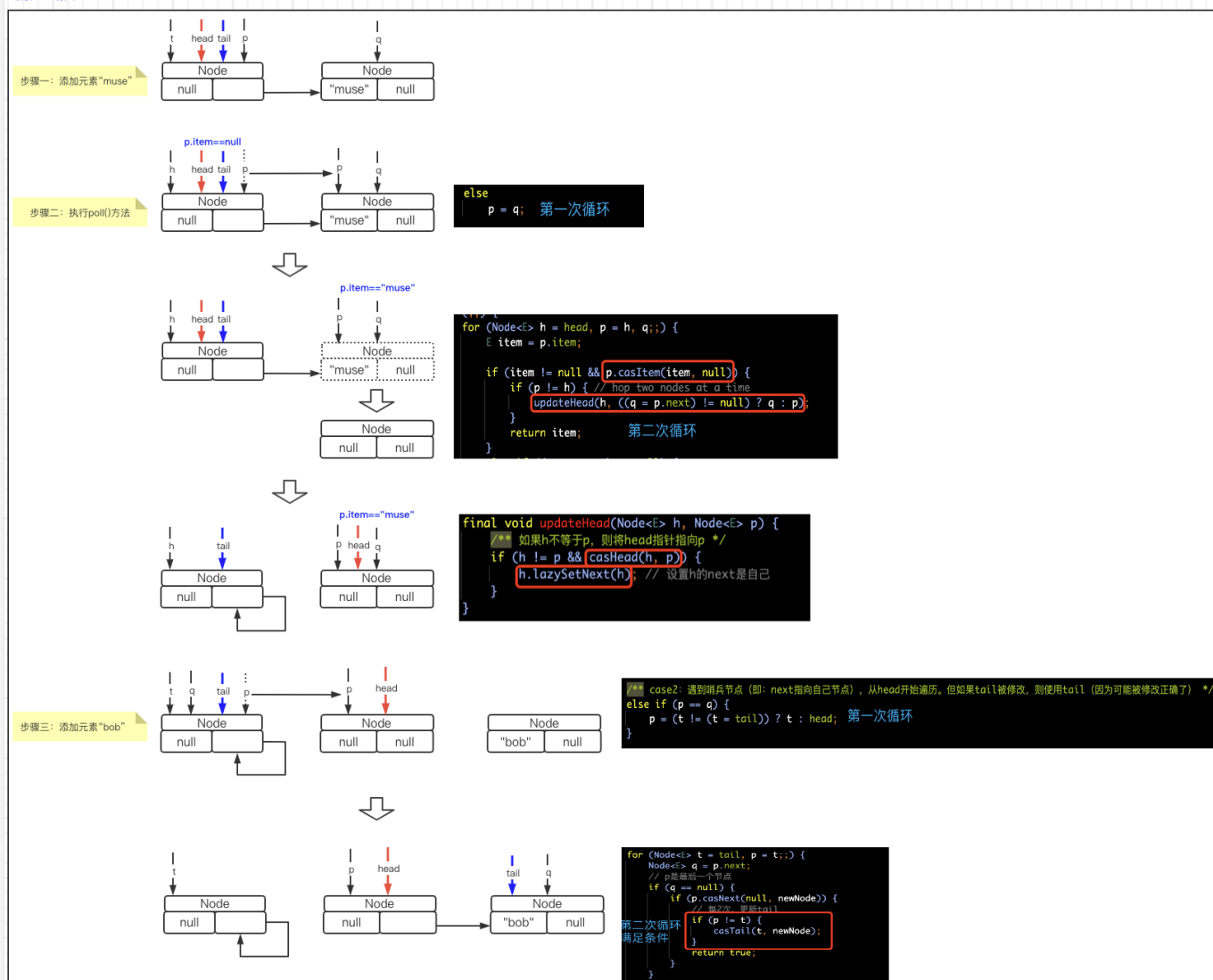
```
AbstractQueue.java
Project JDK is not defined
112 public E remove() {
113     E x = poll();
114     if (x != null)
115         return x;
116     else
117         throw new NoSuchElementException();
118 }
```

- 删除节点对链表的操作, 如下图所示:



1.3.3> 哨兵

- 我们来演示一下哨兵节点产生的原因：



1.4> CopyOnWriteArrayList

- 在大多数的应用场景中, 读操作的比例远远大于写操作。那么, 当执行读操作的时候, 对数据是没有修改的, 所以, 无须对数据进行加锁操作。而针对于写操作的场景中, 则需要加锁来保证数据的正确性。
- 而CopyOnWriteArrayList就可以满足上面所说的场景, 即: 读操作是不加锁的。而**写操作也不会阻塞读的操作**, 它采用了CopyOnWrite方式来解决写操作的问题, 即: 写入操作时, 进行一次自我复制产生一个副本, 写操作就在副本中执行, 写完之后, 再将副本替换原来的数据。这样, 就可以在写数据的同时不影响读数据操作。
- 读操作源码如下所示:

```

public E get(int index) {
    return get(getArray(), index);
}

```

```
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

【解释】

读操作比较简单，就是从数组中获取对应下标为index的元素，而由于读操作并不需要加锁，所以，get方法就是一个普通的不加锁的方法。

- 写操作源码如下所示：

```
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        /** 产生一个原数组长度加1的副本数组 */
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e; // 在副本数组中进行修改操作
        setArray(newElements); // 将副本数组替换旧数组
        return true;
    } finally {
        lock.unlock();
    }
}
```

```
/**
 * 由于array是volatile的，所以多线程之间保持可见性
 */
final void setArray(Object[] a) {
    array = a;
}
```

【解释】

- 执行写操作时，首先进行lock加锁，然后复制原数组创建一个长度加1的新数组，即：副本数组。执行新增操作时，都是针对副本数组进行操作的。当操作新增操作完毕后，将副本数组替换旧的数组。由于array是volatile的，所以当替换后，在多线程之间是可见的。
- 这样做的特点，就是，当执行写操作的时候，针对的是副本数组；而读操作，一直是针对着原数组；所以，写操作是不会阻塞读操作的。

1.5> BlockingQueue

- 阻塞队列常用方法

方法类型	抛出异常	立刻返回	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e, time, unit)
移除	remove()	poll()	take()	poll(time, unit)
检查	element()	peek()	不可用	不可用

- 由于使用offer方法时，如果队列已经满了，那么则无法插入成功，会立即返回false；同样的，当我们调用poll方法的时候，如果队列中是空的，则也会立即返回false；而比起立即返回的情况，我们更关注于put和take这种插入或移除失败，在当前阻塞的情况是如何实现的。下面我们就以ArrayBlockingQueue和LinkedBlockingQueue为例。

1.5.1> ArrayBlockingQueue

- 构造函数，默认采用非公平锁

```
public ArrayBlockingQueue(int capacity) {
    this(capacity, false); // 默认采用非公平锁
}

public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0) {
        throw new IllegalArgumentException();
    }
    this.items = new Object[capacity]; 底层使用对象数组保存元素
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition(); 判断队列是空 or 满
    notFull = lock.newCondition();
}
```

【解释】

- 由构造函数入参capacity来指定底层存储元素数组长度的大小。
 - 并且初始化了需要加锁使用的ReentrantLock实例，默认采用的是非公平锁。
 - notEmpty用于执行take时进行await()等待操作，put时进行signal()唤醒操作。
 - notFull用于执行take时进行signal()唤醒操作，put时进行await()等待操作。
- put方法

```

public void put(E e) throws InterruptedException {
    checkNotNull(e);
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly(); // 添加可被中断锁
    try {
        while (count == items.length) {
            notFull.await(); // 如果队列满了，则阻塞等待，同时释放持有的锁
        }
        // 入队操作
        enqueue(e);
    } finally {
        lock.unlock();
    }
}

```

```

private void enqueue(E x) {
    final Object[] items = this.items;
    items[putIndex] = x;
    // 如果插入到了数组末尾，则重置putIndex，从0开始继续插入
    if (++putIndex == items.length) {
        putIndex = 0;
    }
    count++;
    notEmpty.signal(); // 通知线程解除阻塞
}

```

【解释】

- 在执行put方法逻辑之前，首先尝试获得可中断锁——即：lock.lockInterruptibly()，当执行interrupt操作时，该锁可以被中断。
- 如果数组中元素的个数（count）等于数组的长度了，也就说明队列满了，那么就在该线程上执行等待操作——notFull.await()；
- 如果队列没有满，则调用enqueue(e)方法执行入列操作。
- 入列操作首先会将待插入值x放入数组下标为putIndex的位置上，然后再将putIndex加1，来指向下一次插入的下标位置。此处需要注意的是，如果加1后的putIndex等于了数组的长度，那么说明已经越界了（因为putIndex是从0开始的），那么此时将putIndex置为0，即：待插入的指针指向了数组的头部。做循环式插入。
- 最后，执行count++来计算元素总个数，并且调用notEmpty.signal()方法来解除阻塞（即：当队列为空的时候，执行take方法会被notEmpty.await()阻塞）
- take方法

```

public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0) {
            notEmpty.await(); // 如果队列为空，则执行等待，释放持有的锁
        }
        return dequeue(); // 执行出队操作
    } finally {
        lock.unlock();
    }
}

```

```

private E dequeue() {
    final Object[] items = this.items;
    @SuppressWarnings("unchecked")
    E x = (E) items[takeIndex]; // 获得对应下标的元素
    items[takeIndex] = null; // 将对应下标下的元素置为null，便于后面GC回收
    // 如果出队列操作到了数组的末尾，则重置takeIndex，从0开始继续插入
    if (++takeIndex == items.length) {
        takeIndex = 0;
    }
    count--; // 数组中总元素个数减1
    if (itrs != null) { // 默认itrs等于null，不满足判断
        itrs.elementDequeued();
    }
    notFull.signal(); // 唤醒待插入元素的线程
    return x;
}

```

【解释】

- take方法跟上面我们看的put方法类似，区别是出队的指针是takeIndex。如果队列中为空，那么当调用take方法执行出队操作时，就会执行notEmpty.await()方法执行等待操作，并释放锁资源。当调用put方法向队列中放入元素之后，会调用notEmpty.signal方法对等待的线程执行唤醒操作。那么线程继续执行出队操作，执行完毕后，会调用notFull.signal方法来唤醒在notFull上面await的线程。

1.5.2> LinkedBlockingQueue

- 构造函数，默认长度为 2^{31} ，大概21亿多

```

/**
 * A constant holding the maximum value an {@code int} can
 * have,  $2^{31}-1$ .
 */
// 01111111 11111111 11111111 11111111 = 2147483647
@Native public static final int MAX_VALUE = 0x7fffffff;

public LinkedBlockingQueue() {
    this(Integer.MAX_VALUE);
}

```

```

public LinkedBlockingQueue(int capacity) {
    if (capacity <= 0) {
        throw new IllegalArgumentException();
    }
    this.capacity = capacity; 2^31
    last = head = new Node<E>(null);
}

```

底层使用链表维护元素

【解释】

- 在构造函数中，创建一个空的节点，作为整个链表的头节点。
- put源码和注释如下所示：

```

public void put(E e) throws InterruptedException {
    if (e == null) {
        throw new NullPointerException();
    }
    int c = -1;
    Node<E> node = new Node<>(e);
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count; // count默认为0,与take操作共享全局变量
    putLock.lockInterruptibly();
    try {
        // 如果长度等于capacity,则表示队列已满,插入等待
        while (count.get() == capacity) {
            notFull.await(); 入队操作就是将 NewNode赋值为尾节点的 nextNode
        }
        enqueue(node); // 入队操作
        c = count.getAndIncrement(); // 先获取count值,然后队列中总元素数量加1
        // 注意: getAndIncrement() 方法是先获取值, 再加1
        // 如果队列未满,则唤醒其他put操作导致阻塞的线程
        if (c + 1 < capacity) {
            notFull.signal();
        }
    } finally {
        putLock.unlock();
    }

    // 当c等于0时,表明队列中已经有1个元素了,则执行notEmpty.signal()的唤醒take操作
    if (c == 0) {
        signalNotEmpty(); c初始值是-1
    }
}

```

- take源码和注释如下所示：

```

public E take() throws InterruptedException {
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try {
        // 如果队列空了,则无法获取元素,执行await()方法来执行等待操作,此时释放之前持有的锁
        while (count.get() == 0) {
            notEmpty.await();
        }
        x = dequeue(); // 执行出队操作
        c = count.getAndDecrement(); // 先获得count值,然后将队列中总节点数减1

        // 如果执行完出队操作后,队列中依然有节点,则唤醒那些曾经执行take而被阻塞的线程 (注意: c=-1才表示队列中没有元素)
        if (c > 1) {
            notEmpty.signal();
        }
    } finally {
        takeLock.unlock();
    }
    // c表示未执行出队操作之前,节点的个数,所以,实际节点个数为c-1,因为有1个空位了,那么就唤醒曾经执行put操作而被阻塞的线程,通知他们可以执行put操作了
    if (c == capacity) {
        signalNotFull();
    }
    return x;
}

private E dequeue() {
    Node<E> h = head;
    Node<E> first = h.next; // 由于默认head节点为空节点,所以真实的数据节点在head节点的next节点处
    h.next = h; // help GC (哨兵)
    head = first;
    E x = first.item; // 返回next节点的值
    first.item = null;
    return x;
}

```

1.5.3> SynchronousQueue

- 详情请见: [【源码解析】SynchronousQueue.pdf](#)

二、线程池

2.1> ThreadPoolExecutor

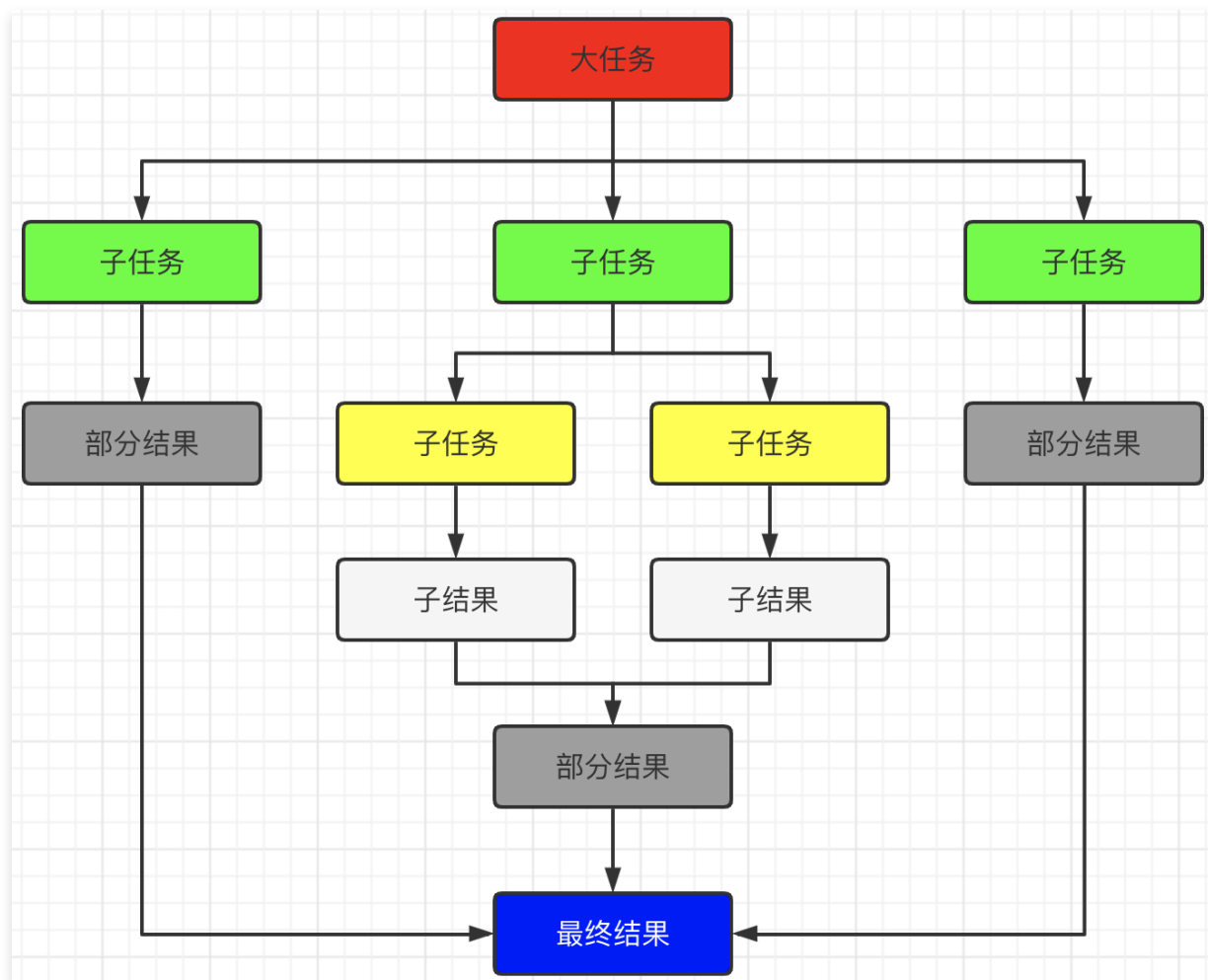
- 详情请见: [【源码解析】ThreadPoolExecutor.pdf](#)

2.2> ForkJoinPool

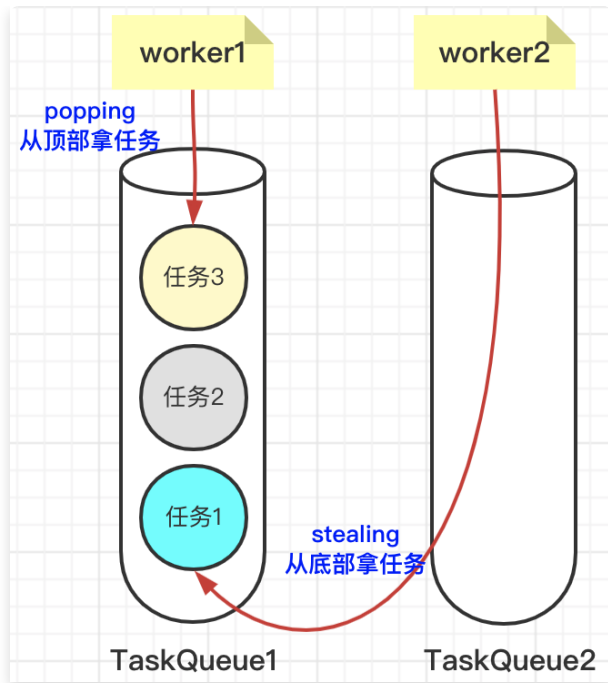
2.2.1> 概述

- ForkJoinPool可以给我们提供[分而治之](#)的功能,当我们有大量任务需要处理的时候,我们可以将其分为N个批次,然后每个批次开启子线程去并发执行,当子线程都执行完毕后,再对结果进行汇总计算。这种思想就类似于Hadoop的MapReduce方式。其中, fork表示开启一个执行分支,即:创建子线程去执行某些任务。而join我们在前面也介绍过,它具有等待的含义,也就是使用fork()后

系统多了一个执行分支或执行线程，所以需要等待这个分支执行完毕，才能进行最后结果汇总的计算。如下图所示：



- 如果我们随意的去fork线程，那么就会导致系统开启了很多子线程而造成系统开销过大，从而影响系统的性能。所以，JDK为我们提供了ForkJoinPool线程池用来解决这个问题。它采用对于fork()方法并不着急开启线程，而是提交给ForkJoinPool线程池去进行处理，从而节省系统开支。由于线程池的优化，提交的任务和线程数量并不是一对一的关系。在绝大多数情况下，一个物理线程实际上是需要处理多个逻辑任务的。因此，每个线程必然需要拥有一个任务队列。因此，在实际执行过程中，可能遇到这么一种情况：线程A已经把自己的任务都执行完成了，而线程B还有一堆任务等着处理，此时，线程A就会“帮助”线程B，从线程B的任务队列中拿一个任务过来处理，尽可能地达到平衡。从而显示了这种互相帮助的精神。但是，其中一个值得注意的地方是，当线程试图帮助别人时，总是从任务队列的底部开始拿数据，而线程试图执行自己的任务时，则是从相反的顶部开始拿。因此这种行为也十分有利于避免数据竞争。如下图所示：



2.2.2> RecursiveTask执行有返回值任务

- 通过RecursiveTask的子类，实现带返回值的计算

```

public class ForkJoinPoolDemo {
    @SneakyThrows
    public static void main(String[] args) {
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        ForkJoinTask<Long> forkJoinTask = forkJoinPool.submit(new ComputeTask(1, 2000));
        System.out.println("sum = " + forkJoinTask.get()); 获得 compute 方法计算后返回的结果
        forkJoinPool.shutdown();
    }
}

class ComputeTask extends RecursiveTask<Long> {
    private static final int THRESHOLD = 1000;
    private int start;
    private int end;

    public ComputeTask(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        long sum = 0;
        if (end - start < THRESHOLD) { // 满足一个批次数量
            for (long i = start; i <= end; i++) {
                sum++;
            }
        } else { // 超出一个批次数量, 需要拆包计算
            List<ComputeTask> computeTasks = Lists.newArrayList();
            int batchNum = (end - start) / THRESHOLD; 判断需要拆分成多少个批次
            int startNum;
            int endNum;
            ComputeTask computeTask;
            for (int i = 0; i <= batchNum; i++) {
                startNum = start + i * THRESHOLD;
                endNum = (THRESHOLD + i * THRESHOLD) > end ? end : (THRESHOLD + i * THRESHOLD);
                computeTask = new ComputeTask(startNum, endNum);
                computeTasks.add(computeTask);
                computeTask.fork(); // 分支计算 开启分支线程计算
            }

            for (ComputeTask task : computeTasks) {
                System.out.println(task.toString() + ": " + task.join());
                sum += task.join(); // 合并结果 获得每个分支的结果进行聚合计算
            }
        }
        return sum;
    }
}

```

```

Run: ForkJoinPoolDemo x
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
com.muse.thread.day3.ComputeTask@6c49510a: 1000
com.muse.thread.day3.ComputeTask@54098ce7: 1000 fork 两个子线程分别进行计算
sum = 2000 最终统计值
Process finished with exit code 0

```

2.2.1> RecursiveAction执行无返回值任务

- 通过RecursiveAction的子类, 实现不带返回值的计算

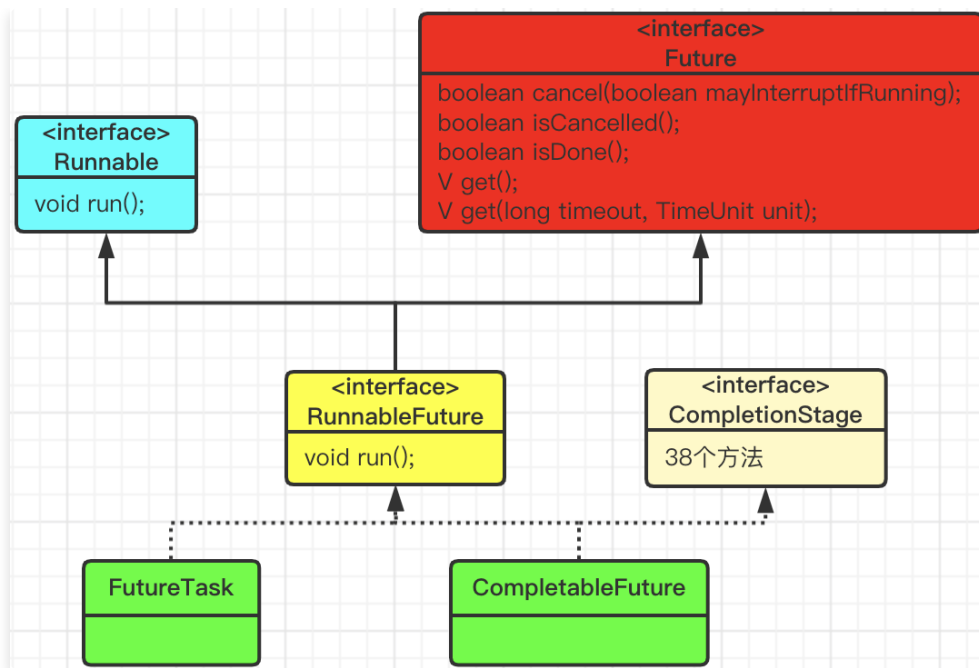
```
ForkJoinPoolDemo1.java
10 public class ForkJoinPoolDemo1 {
11     public static void main(String[] args) throws Throwable {
12         ForkJoinPool forkJoinPool = new ForkJoinPool();
13         forkJoinPool.submit(new ActionTask(1, 2000));
14         TimeUnit.SECONDS.sleep(1); // 等待子线程执行完毕，否则主线程关闭，子线程也就结束了
15         forkJoinPool.shutdown(); // 由于默认是Daemon线程，所以，主线程需要睡眠1秒钟等待子线程执
16     } // 行完毕，否则主线程执行完毕退出，子线程也会被终结。
17 }
18
19 class ActionTask extends RecursiveAction { 适用于执行无返回值的业务逻辑
20     private static final int THRESHOLD = 1000;
21     private int start;
22     private int end;
23
24     public ActionTask(int start, int end) {
25         this.start = start;
26         this.end = end;
27     }
28
29     @Override
30     protected void compute() {
31         if (end - start < THRESHOLD) { // 满足一个批次数量
32             System.out.println(Thread.currentThread().getName() + " is done!");
33         } else { // 超出一个批次数量，需要拆包计算
34             int batchNum = (end - start) / THRESHOLD; // 计算需要多少个批次
35             int startNum;
36             int endNum;
37             ActionTask actionTask;
38             for (int i = 0; i <= batchNum; i++) {
39                 startNum = start + i * THRESHOLD;
40                 endNum = (THRESHOLD + i * THRESHOLD) > end ? end : (THRESHOLD + i * THRESHOLD);
41                 actionTask = new ActionTask(startNum, endNum); // 拆包，并开启分支线程进行计算
42                 actionTask.fork(); // 分支计算
43             }
44         }
45     }
46 }
```

Run: ForkJoinPoolDemo1

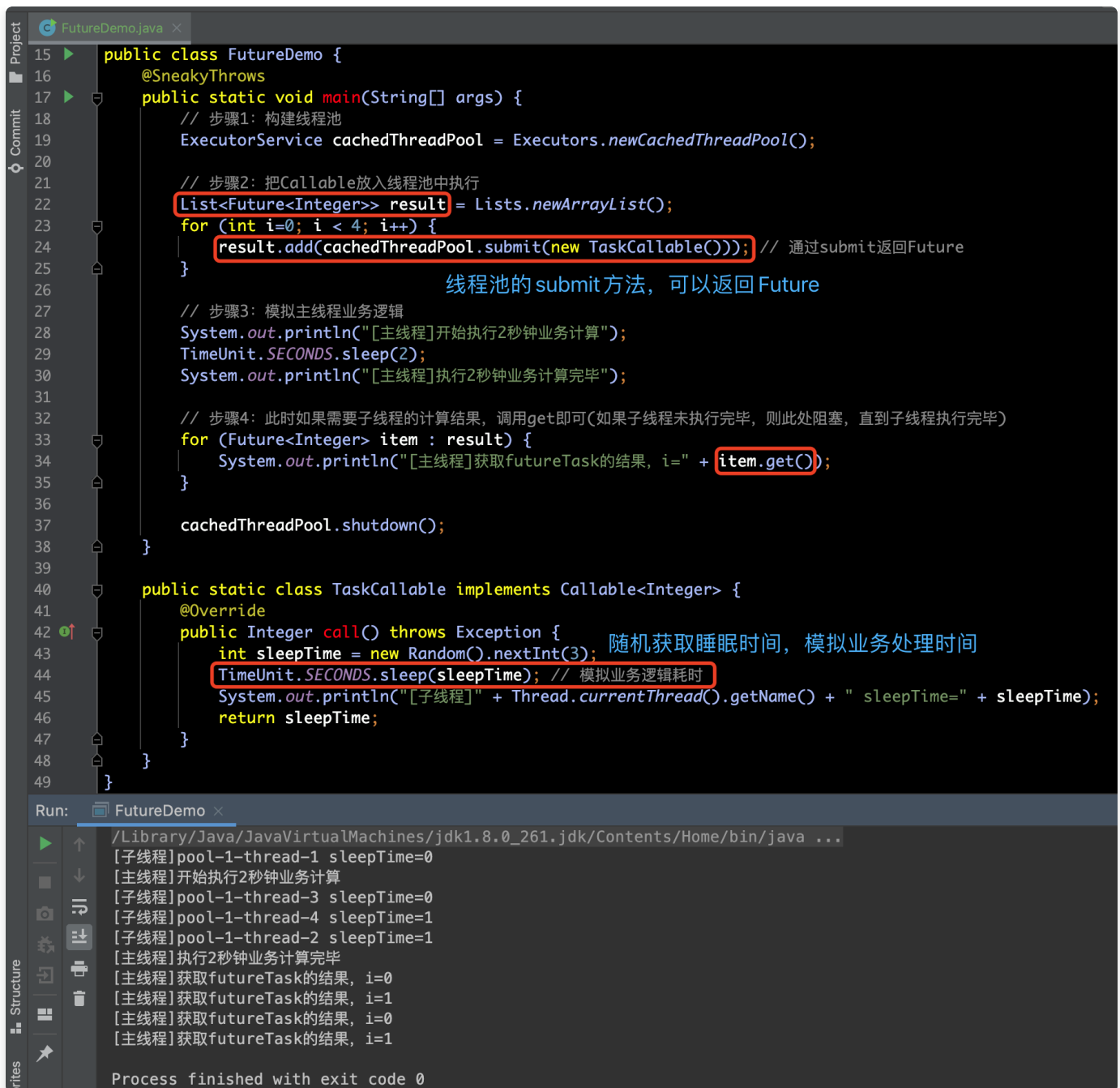
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
ForkJoinPool-1-worker-1 is done!
ForkJoinPool-1-worker-2 is done!
Process finished with exit code 0
```

三、Future

- JDK内置的Future模式



- 可以通过调用线程池的submit方法，返回Future，然后调用get方法来获得子线程计算的结果值，如下所示：



```
15 public class FutureDemo {
16     @SneakyThrows
17     public static void main(String[] args) {
18         // 步骤1: 构建线程池
19         ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
20
21         // 步骤2: 把Callable放入线程池中执行
22         List<Future<Integer>> result = Lists.newArrayList();
23         for (int i=0; i < 4; i++) {
24             result.add(cachedThreadPool.submit(new TaskCallable())); // 通过submit返回Future
25         }
26         // 线程池的submit方法, 可以返回Future
27
28         // 步骤3: 模拟主线程业务逻辑
29         System.out.println("[主线程]开始执行2秒钟业务计算");
30         TimeUnit.SECONDS.sleep(2);
31         System.out.println("[主线程]执行2秒钟业务计算完毕");
32
33         // 步骤4: 此时如果需要子线程的计算结果, 调用get即可(如果子线程未执行完毕, 则此处阻塞, 直到子线程执行完毕)
34         for (Future<Integer> item : result) {
35             System.out.println("[主线程]获取futureTask的结果, i=" + item.get());
36         }
37
38         cachedThreadPool.shutdown();
39     }
40
41     public static class TaskCallable implements Callable<Integer> {
42         @Override
43         public Integer call() throws Exception {
44             int sleepTime = new Random().nextInt(3); // 随机获取睡眠时间, 模拟业务处理时间
45             TimeUnit.SECONDS.sleep(sleepTime); // 模拟业务逻辑耗时
46             System.out.println("[子线程]" + Thread.currentThread().getName() + " sleepTime=" + sleepTime);
47             return sleepTime;
48         }
49     }
50 }
```

Run: FutureDemo

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
[子线程]pool-1-thread-1 sleepTime=0
[主线程]开始执行2秒钟业务计算
[子线程]pool-1-thread-3 sleepTime=0
[子线程]pool-1-thread-4 sleepTime=1
[子线程]pool-1-thread-2 sleepTime=1
[主线程]执行2秒钟业务计算完毕
[主线程]获取futureTask的结果, i=0
[主线程]获取futureTask的结果, i=1
[主线程]获取futureTask的结果, i=0
[主线程]获取futureTask的结果, i=1
Process finished with exit code 0
```

3.1> FutureTask

- FutureTask是RunnableFuture的一个具体实现, 它内部有一个内部类Sync, 它赋值内部逻辑的实现。而Sync会最终调用Callable接口, 完成实际数据的组装工作。
- Callable接口有一个call()方法, 通过方法内部的计算, 可以将结果返回出来。这个Callable接口也是这个Future框架和应用程序直接的重要桥梁。我们可以将需要实现的逻辑在call()方法中实现。通常, 我们会使用Callable实例构造一个FutureTask实例, 并将它提交给线程池。
- 下面是具体实现的例子:

```
FutureTaskDemo.java x
11 public class FutureTaskDemo {
12     @SneakyThrows
13     public static void main(String[] args) {
14         // 步骤1: 构造FutureTask
15         FutureTask futureTask = new FutureTask(new Callable());
16
17         // 步骤2: 把FutureTask放入线程池中执行
18         ExecutorService threadPool = Executors.newFixedThreadPool(1);
19         threadPool.submit(futureTask);
20         System.out.println("[主线程]开启子线程");
21
22         // 步骤3: 模拟主线程业务逻辑
23         System.out.println("[主线程]开始执行2秒钟业务计算");
24         TimeUnit.SECONDS.sleep(2);
25         System.out.println("[主线程]执行2秒钟业务计算完毕");
26
27         // 步骤4: 此时如果需要子线程的计算结果, 调用get即可(如果子线程未执行完毕, 则此处阻塞, 直到子线程执行完毕)
28         System.out.println("[主线程]获取futureTask的结果, i=" + futureTask.get());
29         threadPool.shutdown();
30     }
31
32     public static class TaskCallable implements Callable<Integer> {
33         @Override
34         public Integer call() throws Exception {
35             int i = 0;
36             while (true) {
37                 if (i > 2) {
38                     break;
39                 }
40                 TimeUnit.MILLISECONDS.sleep(100); // 模拟业务逻辑耗时
41                 // TimeUnit.MILLISECONDS.sleep(2000);
42                 System.out.println("[子线程]" + Thread.currentThread().getName() + " i=" + i++);
43             }
44             System.out.println("[子线程]" + Thread.currentThread().getName() + " is done!");
45             return i;
46         }
47     }
48 }
```

如果调用get方法时, 子线程已经计算完毕, 那么则直接返回计算结果;

如果调用get方法时, 子线程依然在计算中, 那么会等待, 直到子线程计算出结果再进行结果返回

如果子线程未执行完毕, 则此处阻塞, 直到子线程执行完毕)

Run: FutureTaskDemo x

/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...

[主线程]开启子线程

[主线程]开始执行2秒钟业务计算

[子线程]pool-1-thread-1 i=0

[子线程]pool-1-thread-1 i=1

[子线程]pool-1-thread-1 i=2

[子线程]pool-1-thread-1 is done!

[主线程]执行2秒钟业务计算完毕

[主线程]获取futureTask的结果, i=3

Process finished with exit code 0

【解释】

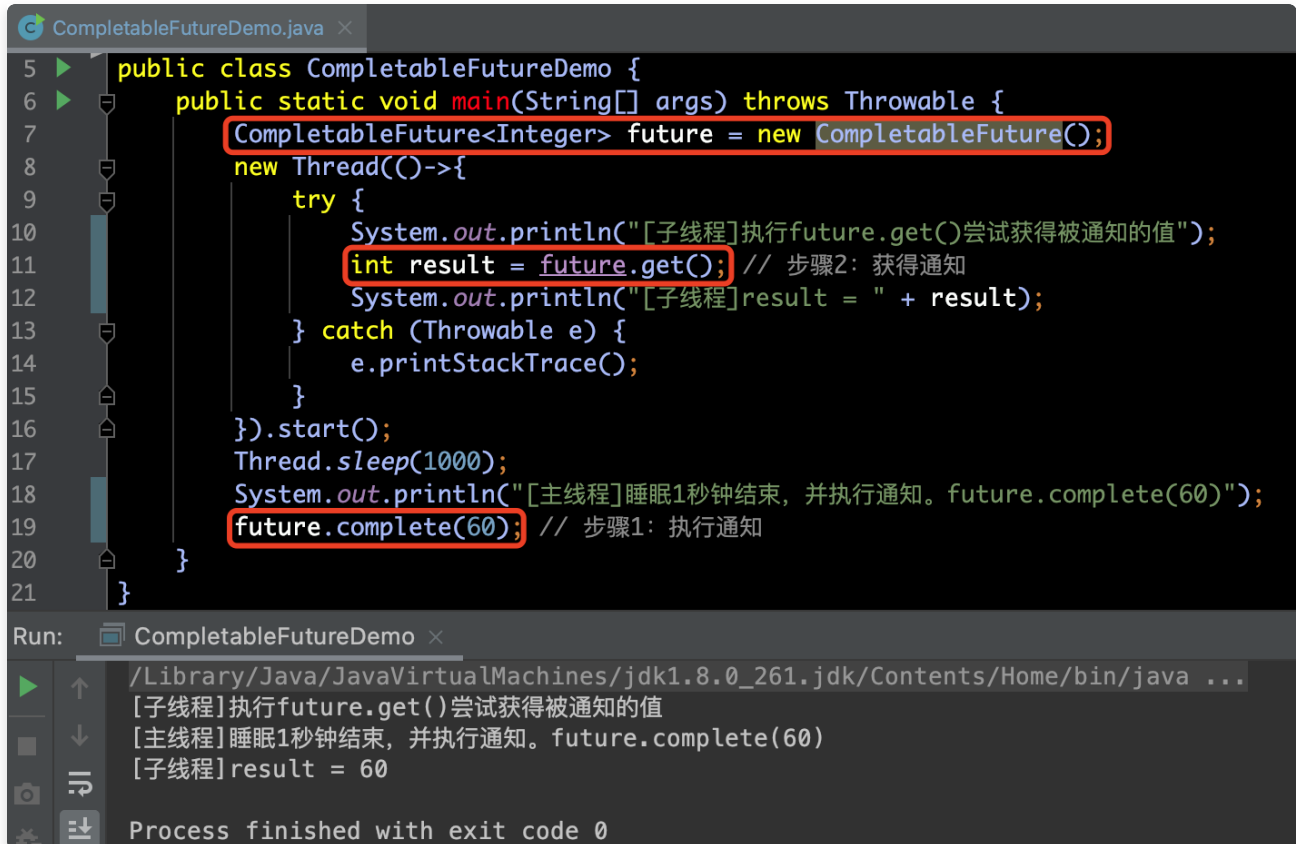
- 我们把需要实现的逻辑在Callable接口的call方法中实现。
- 当构造FutureTask时, 将Callable实例传给它, 告诉FutureTask去做什么事情可以有返回值。
- 然后, 我们将FutureTask提交 (submit) 给线程池。显然, 作为一个简单的任务提交, 这里必然是立即返回的, 因此程序不会阻塞。
- 接下来, 我们不用关系数据是如何计算和产生的, 我们放手去做其他事情 (例如: 上面例子中Sleep了2秒钟), 然后, 当我们需要计算的结果时, 调用FutureTask的get()方法获得计算结果。

3.2> CompletableFuture

- 在Java 8中，新增了CompletableFuture类作为Future的增强类。它实现了CompletionStage接口，该接口有38个方法，是为了函数式编程中的流式调用准备的。

3.2.1> 执行通知

- 如果需要向CompletableFuture请求一个数据，但是需要数据准备好才能发起这个请求，那么此时，我们就可以利用手动设置CompletableFuture的完成状态。下面例子中，我们获取并打印被通知的值：



```
5 public class CompletableFutureDemo {
6     public static void main(String[] args) throws Throwable {
7         CompletableFuture<Integer> future = new CompletableFuture();
8         new Thread(()->{
9             try {
10                 System.out.println("[子线程]执行future.get()尝试获得被通知的值");
11                 int result = future.get(); // 步骤2: 获得通知
12                 System.out.println("[子线程]result = " + result);
13             } catch (Throwable e) {
14                 e.printStackTrace();
15             }
16         }).start();
17         Thread.sleep(1000);
18         System.out.println("[主线程]睡眠1秒钟结束，并执行通知。future.complete(60)");
19         future.complete(60); // 步骤1: 执行通知
20     }
21 }
```

Run: CompletableFutureDemo

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
[子线程]执行future.get()尝试获得被通知的值
[主线程]睡眠1秒钟结束，并执行通知。future.complete(60)
[子线程]result = 60
Process finished with exit code 0
```

3.2.2> 执行异步任务

- 可以通过supplyAsync和runAsync来执行异步任务，具体方法如下所示：



```
▼ CompletableFuture 适用于有返回值的场景
  supplyAsync(Supplier<U>): CompletableFuture<U>
  supplyAsync(Supplier<U>, Executor): CompletableFuture<U>

  runAsync(Runnable): CompletableFuture<Void> 适用于无返回值的场景
  runAsync(Runnable, Executor): CompletableFuture<Void>
```

【解释】

- supplyAsync()方法用于那些需要有返回值的场景；
- runAsync()方法用于没有返回值的场景；
- 在这两个方法中，都分别有一个方法可以接收一个Executor参数。这就使我们可以让

Supplier<U>或者Runnable在指定的线程池中工作。如果不确定，则在默认的系统公共的ForkJoinPool.common线程池中执行。

- 在Java 8中，新增了ForkJoinPool.commonPool()方法。它可以获得一个公共的ForkJoin线程池。这个公共线程池中的所有线程都是Daemon线程。这意味着如果主线程退出，这些线程无论是否执行完毕，都会退出系统。

```
ForkJoinPool.java x
3395
3396     common = java.security.AccessController.doPrivileged
3397         (new java.security.PrivilegedAction<ForkJoinPool>() {
3398             public ForkJoinPool run() { return makeCommonPool(); });
3399     int par = common.config & SMASK; // report 1 even if threads disabled
3400     commonParallelism = par > 0 ? par : 1;
3401 }
```

```
ForkJoinPool.java x
1531 @ final WorkQueue registerWorker(ForkJoinWorkerThread wt) {
1532     UncaughtExceptionHandler handler;
1533     wt.setDaemon(true); 被定义为守护进程 // configure thread
1534     if ((handler = ueh) != null)
1535         wt.setUncaughtExceptionHandler(handler);
1536     WorkQueue w = new WorkQueue(this, wt);
1537     int i = 0; // assign a pool index
1538     int mode = config & MODE_MASK;
1539     int rs = lockRunState();
1540     try {
```

- 下面是异步任务的例子：

```
CompletableFutureDemo1.java x
11 ▶ public class CompletableFutureDemo1 {
12
13     @SneakyThrows
14 ▶     public static void main(String[] args) {
15         System.out.println("[主线程]开始执行CompletableFuture异步任务");
16
17         // 开启异步任务
18         CompletableFuture<Integer> future = CompletableFuture.supplyAsync() -> {
19             System.out.println("[子线程]开始执行任务");
20             try {
21                 TimeUnit.SECONDS.sleep(2); // 模拟业务处理耗时
22             } catch (InterruptedException e) {
23                 e.printStackTrace();
24             }
25             System.out.println("[子线程]任务执行完毕");
26             return 1;
27         });
28
29         // 等待异步任务执行完毕，采用通过future.get()获得值
30         System.out.println("[主线程]执行future.get()=" + future.get());
31     }
32 }
```

Run: CompletableFutureDemo1 x

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_261.jdk/Contents/Home/bin/java ...
[主线程]开始执行CompletableFuture异步任务
[子线程]开始执行任务
[子线程]任务执行完毕
[主线程]执行future.get()=1
Process finished with exit code 0
```