JUC多线程 (三)

学习目标:

- 掌握CyclicBarrier同步屏障的使用
- 掌握CountDownLatch的使用
- 掌握Semaphore信号量的使用
- 掌握ConcurrentHashMap同步容器的使用
- 掌握四种BlockingQueue阻塞队列的使用
- 掌握线程池的使用，了解内置的四种线程池

11 J.U.C之并发工具类

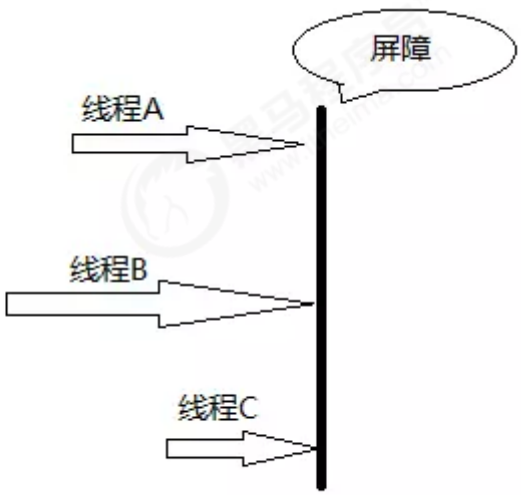
11.1 CyclicBarrier

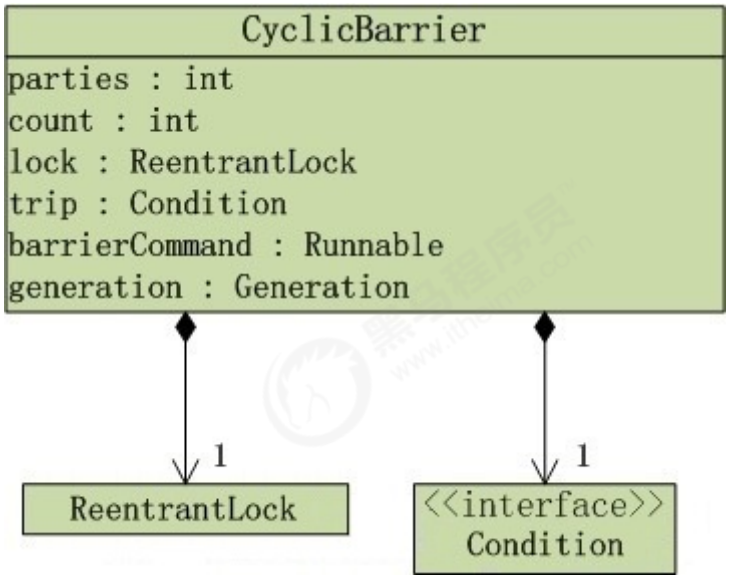
11.1.1 介绍

CyclicBarrier也叫同步屏障，在JDK1.5被引入的一个同步辅助类，在API中是这么介绍的:

允许一组线程全部等待彼此达到共同屏障点的同步辅助。 循环阻塞在涉及固定大小的线程方的程序中很有用，这些线程必须偶尔等待彼此。 屏障被称为循环，因为它可以在等待的线程被释放之后重新使用。

CyclicBarrier好比一扇门，默认情况下关闭状态，堵住了线程执行的道路，直到所有线程都就位，门才打开，让所有线程一起通过。





通过上图我们可以看到CyclicBarrier的内部是使用重入锁ReentrantLock和Condition。它有两个构造方法：

- CyclicBarrier(int parties)：它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动屏障时执行预定义的操作。parties表示拦截线程的数量。
- CyclicBarrier(int parties, Runnable barrierAction)：创建一个新的CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，并在启动屏障时执行给定的屏障操作，该操作由最后一个进入屏障的线程执行。

构造方法如下：

```
public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    this.parties = parties;
    this.count = parties;
    this.barrierCommand = barrierAction;
}

public CyclicBarrier(int parties) {
    this(parties, null);
}
```

在CyclicBarrier中最重要的方法莫过于await()方法，每个线程调用await方法告诉CyclicBarrier已经到达屏障位置，线程被阻塞。源码如下：

```
BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
```

await()方法的逻辑：如果该线程不是到达的最后一个线程，则他会一直处于等待状态，除非发生以下情况：

1. 最后一个线程到达，即index == 0
2. 超出了指定时间（超时等待）
3. 其他的某个线程中断当前线程
4. 其他的某个线程中断另一个等待的线程
5. 其他的某个线程在等待屏障超时
6. 其他的某个线程在此屏障调用reset()方法。reset()方法用于将屏障重置为初始状态。

11.1.3 案例

田径比赛，所有运动员准备好了之后，大家一起跑，代码如下

```
public class Demo1CyclicBarrier {

    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

        List<Thread> threadList = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new Athlete(cyclicBarrier, "运动
员" + i));
            threadList.add(t);
        }

        for (Thread t : threadList) {
            t.start();
        }
    }
```

```
        private CyclicBarrier cyclicBarrier;
        private String name;

        public Athlete(CyclicBarrier cyclicBarrier, String name)
{
            this.cyclicBarrier = cyclicBarrier;
            this.name = name;
        }

        @Override
        public void run() {
            System.out.println(name + "就位");
            try {
                cyclicBarrier.await();
                System.out.println(name + "跑到终点。");
            } catch (Exception e) {
            }
        }
    }
}
```

11.2 CountDownLatch

11.2.1 介绍

CountDownLatch是一个计数的闭锁，作用与CyclicBarrier有点儿相似。

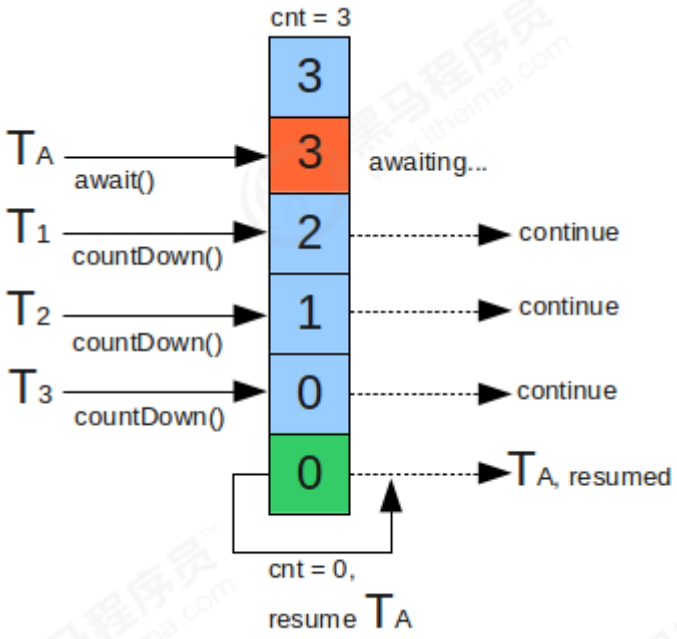
在API中是这样描述的：

> 用给定的计数 初始化 CountDownLatch。由于调用了 countDown() 方法，所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。
>
> 这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。

- CountDownLatch：一个或者多个线程，等待其他多个线程完成某件事情之后才能执行；
- CyclicBarrier：多个线程互相等待，直到到达同一个同步点，再继续一起执行。

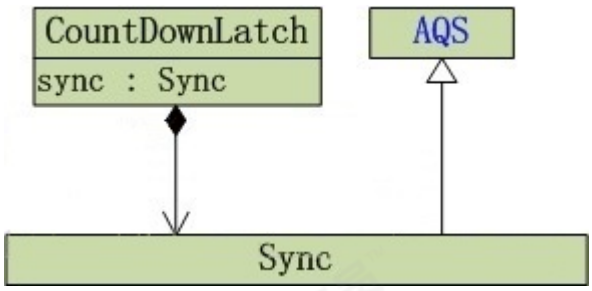
线程任完成 某件事情 之后，可以终止，也可以等待。而对于CyclicBarrier，重点是多个线程，在任意一个线程没有完成，所有的线程都必须等待。

CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。每当一个线程完成了自己的任务后，计数器的值就会减1。当计数器值到达0时，它表示所有的线程已经完成了任务，然后就可以恢复等待的线程继续执行了。如下图



11.2.2 实现分析

CountDownLatch结构如下



通过上面的结构图我们可以看到，CountDownLatch内部依赖Sync实现，而Sync继承AQS。CountDownLatch仅提供了一个构造方法：

CountDownLatch(int count)： 构造一个用给定计数初始化的 CountDownLatch

```
public CountDownLatch(int count) {
    if (count < 0) throw new IllegalArgumentException("count <
0");
    this.sync = new Sync(count);
}
```

CountDownLatch是采用共享锁来实现的。最长用的两个方法是await()和countDown()：

- CountDownLatch提供await()方法来使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。内部使用AQS的getState方法获取计数器，如果计数器值不等于0，则会以自旋方式会尝试一直去获取同步状态。
- CountDownLatch提供countDown() 方法递减锁存器的计数，如果计数到达零，则释放所有等待的线程。内部调用AQS的releaseShared(int arg)方法来释放共享锁同步状态。

11.2.3 案例

在CyclicBarrier应用场景之上进行修改，添加接力运动员。

起点运动员应该等其他起点运动员准备好才可以起跑（CyclicBarrier）。

接力运动员不需要关心其他人，只需和自己有关的起点运动员到接力点即可开跑（CountDownLatch）。

```
public class Demo2CountDownLatch {

    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

        List<Thread> threadList = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            CountDownLatch countDownLatch = new
CountDownLatch(1);
            //起点运动员
            Thread t1 = new Thread(new Athlete(cyclicBarrier,
countDownLatch, "起点运动员" + i));

            //接力运动员
            Thread t2 = new Thread(new Athlete(countDownLatch,
"接力运动员" + i));

            threadList.add(t1);
            threadList.add(t2);
        }

        for (Thread t : threadList) {
```

```
    }

    static class Athlete implements Runnable {

        private CyclicBarrier cyclicBarrier;
        private String name;

        CountDownLatch countDownLatch;

        //起点运动员
        public Athlete(CyclicBarrier cyclicBarrier,
CountDownLatch countDownLatch, String name) {
            this.cyclicBarrier = cyclicBarrier;
            this.countDownLatch = countDownLatch;
            this.name = name;
        }

        //接力运动员
        public Athlete(CountDownLatch countDownLatch, String
name) {
            this.countDownLatch = countDownLatch;
            this.name = name;
        }

        @Override
        public void run() {
            //判断是否是起点运动员
            if (cyclicBarrier != null) {

                System.out.println(name + "就位");
                try {
                    cyclicBarrier.await();
                    System.out.println(name + "到达交接点。");

                    //已经到达交接点
                    countDownLatch.countDown();
                } catch (Exception e) {
                }
            }

            //判断是否是接力运动员
            if (cyclicBarrier == null) {
                System.out.println(name + "就位");
```

```
                    System.out.println(name + "到达终点。");
                } catch (Exception e) {
                }
            }
        }
    }
}
```

11.3 Semaphore

11.3.1 介绍

Semaphore是一个控制访问多个共享资源的计数器，和CountDownLatch一样，其本质上是一个“共享锁”。

Semaphore维护了一个信号量许可集。线程可以获取信号量的许可；当信号量中有可用的许可时，线程能获取该许可；否则线程必须等待，直到有可用的许可为止。 线程可以释放它所持有的信号量许可，被释放的许可归还到许可集中，可以被其他线程再次获取。

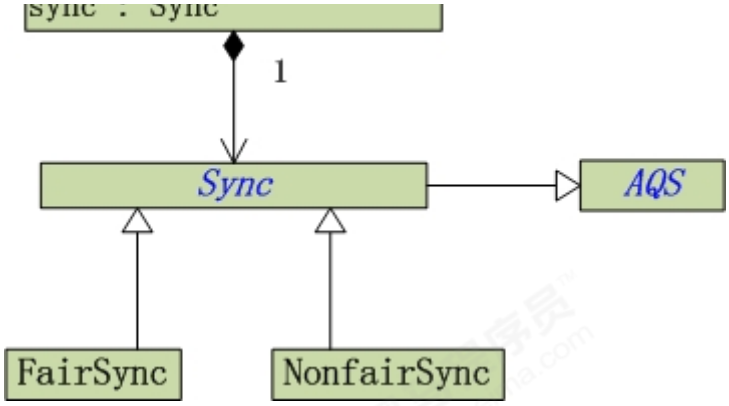
举个例子：

假设停车场仅有5个停车位，一开始停车场没有车辆所有车位全部空着，然后先后到来三辆车，停车场车位够，安排进去停车，然后又来三辆，这个时候由于只剩两个停车位，所有只能停两辆，其余一辆必须在外面候着，直到停车场有空车位，当然以后每来一辆都需要在外面等着。当停车场有车开出去，里面有空位了，则安排一辆车进去（至于是哪辆车 要看选择的机制是公平还是非公平）。

Semaphore常用于约束访问一些（物理或逻辑）资源的线程数量。

当信号量初始化为 1 时，可以当作互斥锁使用，因为它只有两个状态：有一个许可能使用，或没有许可能使用。当以这种方式使用时，“锁”可以被其他线程控制和释放，而不是主线程控制释放。

11.3.2 实现分析



从上图可以看出Semaphore内部包含公平锁（FairSync）和非公平锁（NonfairSync），继承内部类Sync，其中Sync继承AQS（再一次阐述AQS的重要性）。

Semaphore提供了两个构造函数：

1. Semaphore(int permits)： 创建具有给定的许可数和非公平的 Semaphore。
2. Semaphore(int permits, boolean fair)： 创建具有给定的许可数和给定的公平设置的 Semaphore。

实现如下： （Semaphore默认选择非公平锁）

```
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new
NonfairSync(permits);
}
```

信号量获取

Semaphore提供了acquire()方法来获取一个许可。

```
public void acquire() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}
```

内部使用AQS以共享模式获取同步状态，核心源码：

```
    for (;;) {
        //判断该线程是否位于CLH队列的列头
        if (hasQueuedPredecessors())
            return -1;
        //获取当前的信号量许可
        int available = getState();

        //设置“获得acquires个信号量许可之后，剩余的信号量许可数”
        int remaining = available - acquires;

        //CAS设置信号量
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

```
//非公平 对于非公平而言，因为它不需要判断当前线程是否位于CLH同步队列列头。
protected int tryAcquireShared(int acquires) {
    return nonfairTryAcquireShared(acquires);
}

final int nonfairTryAcquireShared(int acquires) {
    for (;;) {
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

信号量释放

获取了许可，当用完之后就需要释放，Semaphore提供release()来释放许可。

```
public void release() {
    sync.releaseShared(1);
}
```

```
protected final boolean tryReleaseShared(int releases) {
    for (;;) {
        int current = getState();
        //信号量的许可数 = 当前信号许可数 + 待释放的信号许可数
        int next = current + releases;
        if (next < current) // overflow
            throw new Error("Maximum permit count exceeded");
        //设置可获取的信号许可数为next
        if (compareAndSetState(current, next))
            return true;
    }
}
```

11.3.3 案例

停车为示例：

```
public class Demo3Semaphore {

    public static void main(String[] args) {
        Parking parking = new Parking(3);
        for (int i = 0; i < 5; i++) {
            new Car(parking).start();
        }
    }

    static class Parking {
        //信号量
        private Semaphore semaphore;

        Parking(int count) {
            semaphore = new Semaphore(count);
        }

        public void park() {
            try {
                //获取信号量
                semaphore.acquire();
                long time = (long) (Math.random() * 10);
```

```
场，停车" + time + "秒...");
                Thread.sleep(time);

 System.out.println(Thread.currentThread().getName() + "开出停车
场...");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                //释放信号量
                semaphore.release();
            }
        }
    }

    static class Car extends Thread {
        Parking parking;

        Car(Parking parking) {
            this.parking = parking;
        }

        @Override
        public void run() {
            //进入停车场
            parking.park();
        }
    }
}
```

12 J.U.C之并发容器ConcurrentHashMap

12.1 介绍

HashMap是我们用得非常频繁的一个集合，但是它是线程不安全的。并且在多线程环境下，put操作是有可能产生死循环，不过在JDK1.8的版本中更换了数据插入的顺序，已经解决了这个问题。

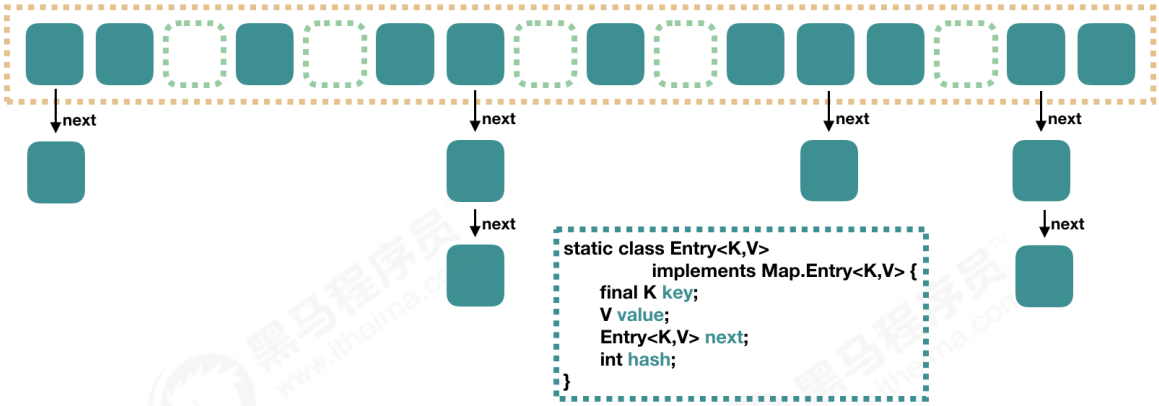
两种解决方案，但是这两种方案都是对读写加锁，独占式。一个线程在读的其他线程必须等待，吞吐量较低，性能较为低下。而J.U.C给我们提供了高性能的线程安全HashMap：ConcurrentHashMap。

在1.8版本以前，ConcurrentHashMap采用分段锁的概念，使锁更加细化，但是1.8已经改变了这种思路，而是利用CAS+Synchronized来保证并发更新的安全，当然底层采用数组+链表+红黑树的存储结构。

12.2 JDK7 HashMap

HashMap 是最简单的，它不支持并发操作，下面这张图是 HashMap 的结构：

Java7 HashMap 结构



HashMap 里面是一个数组，然后数组中每个元素是一个单向链表。每个绿色的实体是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next。

`public HashMap(int initialCapacity, float loadFactor)`初始化方法的参数说明：

> capacity：当前数组容量，始终保持 2^n，可以扩容，扩容后数组大小为当前的 2 倍。
> loadFactor：负载因子，默认为 0.75。
> threshold：扩容的阈值，等于 capacity * loadFactor

put 过程

- 数组初始化，在第一个元素插入 HashMap 的时候做一次数组的初始化，先确定初始的数组大小，并计算数组扩容的阈值。

数组中。

- 找到数组下标后，会先进行 key 判断是否重复，如果没有重复，就准备将新值放入到链表的表头（在多线程操作中，这种操作会造成死循环，在jdk1.8已解决）。
- 数组扩容，在插入新值的时候，如果当前的 size 已经达到了阈值，并且要插入的数组位置上已经有元素，那么就会触发扩容，扩容后，数组大小为原来的 2 倍。扩容就是用一个新的大数组替换原来的小数组，并将原来数组中的值迁移到新的数组中。

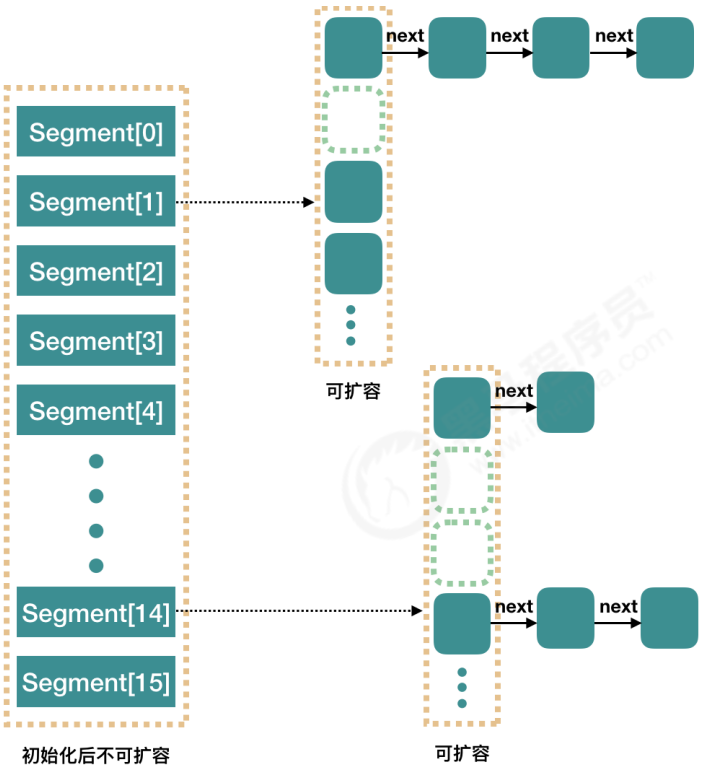
get过程

- 根据 key 计算 hash 值。
- 根据hash值找到相应的数组下标。
- 遍历该数组位置处的链表，直到找到相等的 key。

12.3 JDK7 ConcurrentHashMap

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。

整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，所以很多人都会将其描述为分段锁。简单的说，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的。



再具体到每个 Segment 内部，其实每个 Segment 很像之前介绍的 HashMap，每次操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的。

初始化

`public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)` 初始化方法

- initialCapacity：整个 ConcurrentHashMap 的初始容量，实际操作的时候需要平均分给每个 Segment。
- concurrencyLevel：并发数(或者Segment 数，有很多叫法，重要的是如何理解)。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。
- loadFactor：负载因子，Segment 数组不可以扩容，所以这个负载因子是给每个 Segment 内部使用的。

举个简单的例子：

用 new ConcurrentHashMap() 无参构造函数进行初始化的，那么初始化完成后：

- Segment[]的默认大小为2，负载因子是0.75，得出初始阈值为1.5，也就是以后插入第一个元素不会触发扩容，插入第二个会进行第一次扩容
- 这里初始化了 segment[0]，其他位置还是 null，

put过程

- 根据 hash 值能找到相应的 Segment，之后就是 Segment 内部的 put 操作了。
- Segment 内部是由 数组+链表 组成的，由于有独占锁的保护，所以 segment 内部的操作并不复杂。保证多线程安全的，就是做了一件事，那就是获取该 segment 的独占锁。
- Segment 数组不能扩容，**rehash方法**扩容是 segment 数组某个位置内部的数组 HashEntry[] 进行扩容，扩容后，容量为原来的 2 倍。

get过程

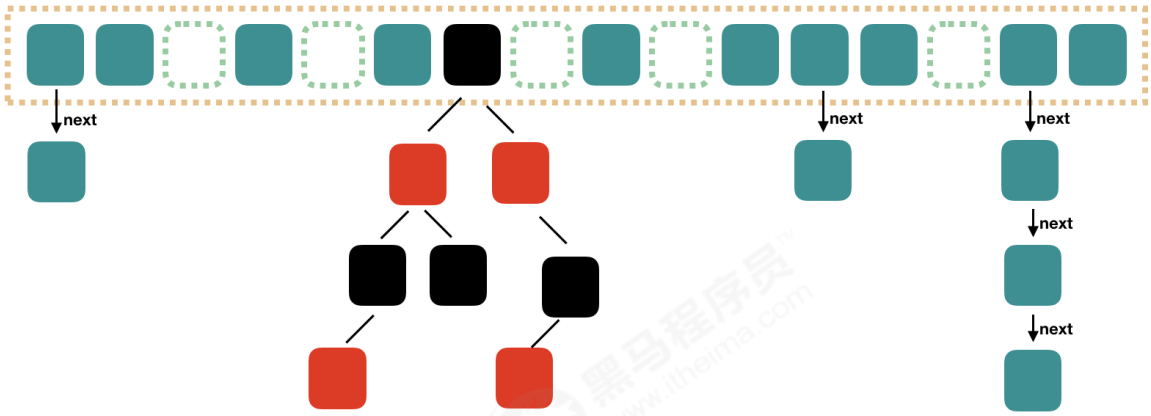
- 计算 hash 值，找到 segment 数组中的具体位置
- segment 数组中也是数组，再根据 hash 找到数组中具体值的位置
- 到这里是链表了，顺着链表进行查找即可

12.4 JDK8 HashMap

Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链表+红黑树 组成。

根据 Java7 HashMap 的介绍，我们知道，查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后的话，需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决于链表的长度。

为了降低这部分的开销，在 Java8 中，当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度。



jdk7 中使用 Entry 来代表每个 HashMap 中的数据节点，jdk8 中使用 Node，基本没有区别，都是 key，value，hash 和 next 这四个属性，不过，Node 只能用于链表的情况，红黑树的情况需要使用 TreeNode。

我们根据数组元素中，第一个节点数据类型是 Node 还是 TreeNode 来判断该位置下是链表还是红黑树的。

put过程

和jdk7的put差不多

- 和 Jdk7 不一样的地方就是，jdk7是先扩容后插入新值的，jdk8 先插值再扩容
- 先使用链表进行存放数据，当数量超过8个的时候，将链表转为红黑树

get 过程分析

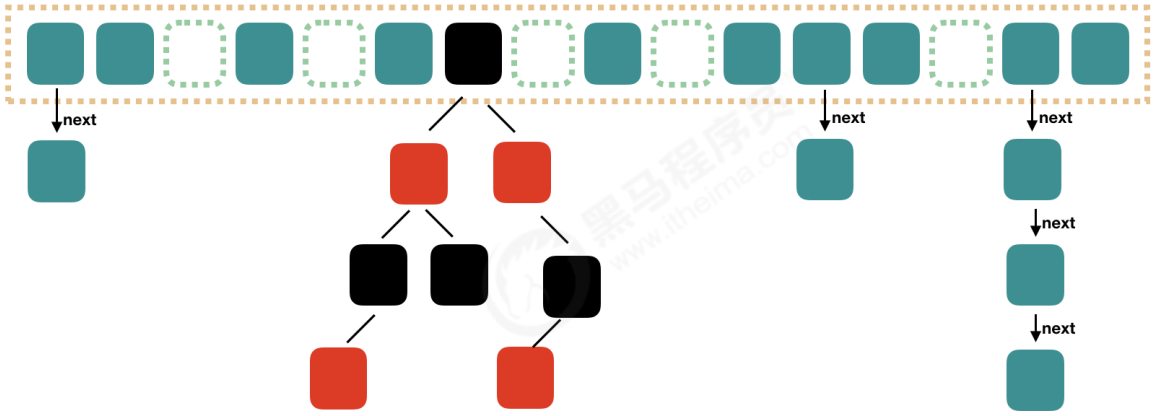
1. 计算 key 的 hash 值，根据 hash 值找到对应数组下标。
2. 判断数组该位置处的元素是否刚好就是我们要找的，如果不是，走第三步。
3. 判断该元素类型是否是 TreeNode，如果是，用红黑树的方法取数据，如果不是，走第四步。
4. 遍历链表，直到找到相等(==或equals)的 key。

12.5 JDK8 ConcurrentHashMap

Java7 中实现的 ConcurrentHashMap 还是比较复杂的，Java8 对 ConcurrentHashMap 进行了比较大的改动。可以参考 Java8 中 HashMap 相对于 Java7 HashMap 的改动，对于 ConcurrentHashMap，Java8 也引入了红黑树。

1.8已经改变了这种思路，而是利用CAS+Synchronized来保证并发更新的安全，底层采用数组+链表+红黑树的存储结构。

Java8 ConcurrentHashMap 结构



12.6 使用场景

ConcurrentHashMap通常只被看做并发效率更高的Map，用来替换其他线程安全的Map容器，比如Hashtable和Collections.synchronizedMap。线程安全的容器，特别是Map，很多情况下一个业务中涉及容器的操作有多个，即复合操作，而在并发执行时，线程安全的容器只能保证自身的数据不被破坏，和数据在多个线程间是可见的，但无法保证业务的行为是否正确。

ConcurrentHashMap总结：

- HashMap是线程不安全的，ConcurrentHashMap是线程安全的，但是线程安全仅仅指的是对容器操作的时候是线程安全的
- ConcurrentHashMap的public V get(Object key)不涉及到锁，也就是说获得对象时没有使用锁
- put、remove方法，在jdk7使用锁，但多线程中并不一定有锁争用，原因在于ConcurrentHashMap将缓存的变量分到多个Segment，每个Segment上有一个锁，只要多个线程访问的不是一个Segment就没有锁争用，就没有堵塞，各线程用各自的锁，ConcurrentHashMap缺省情况下生成16个Segment，也就是允许16个线程并发的更新而尽量没有锁争用。而在jdk8中使用的CAS+Synchronized来保证线程安全，比加锁的性能更高
- ConcurrentHashMap线程安全的，允许一边更新、一边遍历，也就是说在对象遍历的时候，也可以进行remove,put操作，且遍历的数据会随着remove,put操作产出变化，

```
public class Demo4ConcurrentHashMap1 {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap();
        //Map<String, Integer> map = new ConcurrentHashMap<>();
        //Map<String, Integer> map = new Hashtable<>();

        map.put("a", 1);
        map.put("b", 1);
        map.put("c", 1);
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            map.remove(entry.getKey());
        }

        System.out.println(map.size());
    }
}
```

案例2：业务操作的线程安全不能保证

```
public class Demo4ConcurrentHashMap2 {
    public static void main(String[] args) {
        final Map<String, Integer> count = new HashMap<>();
        //final Map<String, Integer> count = new
ConcurrentHashMap<>();
        //final Hashtable<String, Integer> count = new
Hashtable<>();
        count.put("count", 0);

        Runnable task = new Runnable() {
            @Override
            public void run() {
                //synchronized (count) {
                    int value;
                    for (int i = 0; i < 2000; i++) {
                        value = count.get("count");
                        count.put("count", value + 1);
                    //}
                }
            }
        };
        new Thread(task).start();
```

```
        try {
            Thread.sleep(1000l);
            System.out.println(count);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

案例3：多线程删除

```
public class Demo4ConcurrentHashMap3 {
    public static void main(String[] args) {
        final Map<String, Integer> count = new HashMap<>();
        //final Map<String, Integer> count = new
ConcurrentHashMap<>();
        //final Hashtable<String, Integer> count = new
Hashtable<>();

        for (int i = 0; i < 2000; i++) {
            count.put("count" + i, 1);
        }

        Runnable task1 = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 500; i++) {
                    count.remove("count" + i);
                }
            }
        };
        Runnable task2 = new Runnable() {
            @Override
            public void run() {
                for (int i = 1000; i < 1500; i++) {
                    count.remove("count" + i);
                }
            }
        };
```

```
            new Thread(task2).start();

            try {
                Thread.sleep(10001);
                System.out.println(count.size());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

12.7 对比Hashtable

Hashtable和ConcurrentHashMap的不同点：

- Hashtable对get,put,remove都使用了同步操作，它的同步级别是正对Hashtable来进行同步的，也就是说如果有线程正在遍历集合，其他的线程就暂时不能使用该集合了，这样无疑就很容易对性能和吞吐量造成影响，从而形成单点。而ConcurrentHashMap则不同，它只对put,remove操作使用了同步操作，get操作并不影响。
- Hashtable在遍历的时候，如果其他线程，包括本线程对Hashtable进行了put，remove等更新操作的话，就会抛出ConcurrentModificationException异常，但如果使用ConcurrentHashMap的话，就不用考虑这方面的问题了

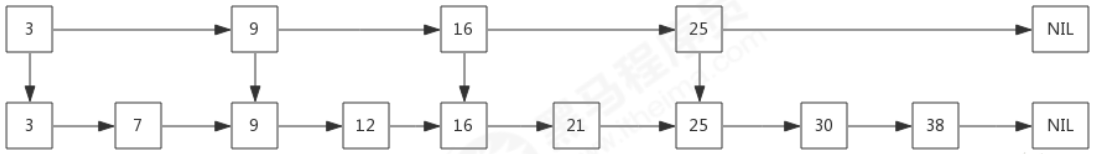
12.8 了解ConcurrentSkipListMap

通过对前面ConcurrentHashMap的学习，我们了解到Map存放数据的两种数据结构：链表和红黑树，这两种数据结构各自都有着优缺点。而ConcurrentSkipListMap使用的是第三种数据结构：SkipList。SkipList有着不低于红黑树的效率。

Skip List，称之为跳表，它是一种可以替代平衡树的数据结构，其数据元素默认按照key值升序，天然有序。Skip list让已排序的数据分布在多层链表中，以0-1随机数决定一个数据的向上攀升与否，通过“空间来换取时间”的一个算法，在每个节点中增加了向前的指针，在插入、删除、查找时可以忽略一些不可能涉及到的结点，从而提高了效率。

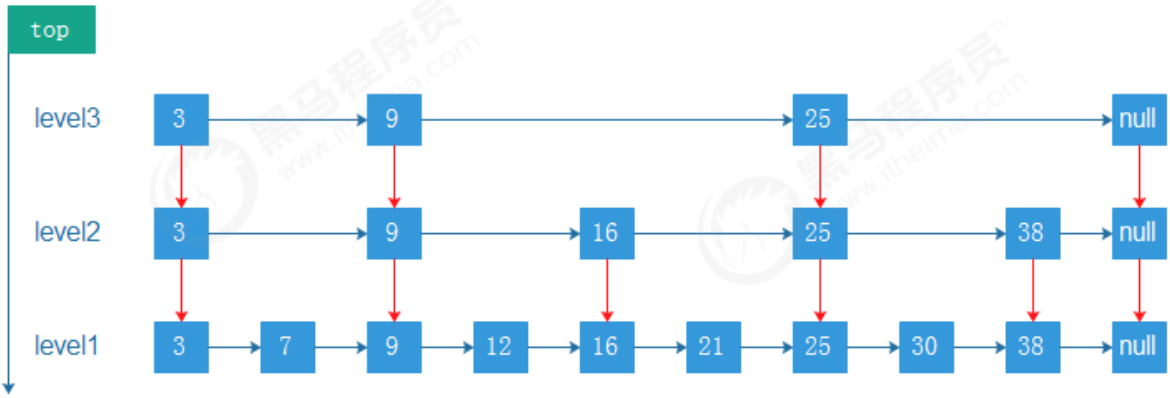
我们先看一个简单的链表，如下：

如果我们需要查询9、21、30，则需要比较次数为3 + 6 + 8 = 17 次，那么有没有优化方案呢？有！我们将该链表中的某些元素提炼出来作为一个比较“索引”，如下：



我们先与这些索引进行比较来决定下一个元素是往右还是下走，由于存在“索引”的缘故，导致在检索的时候会大大减少比较的次数。当然元素不是很多，很难体现出优势，当元素足够多的时候，这种索引结构就会大显身手。

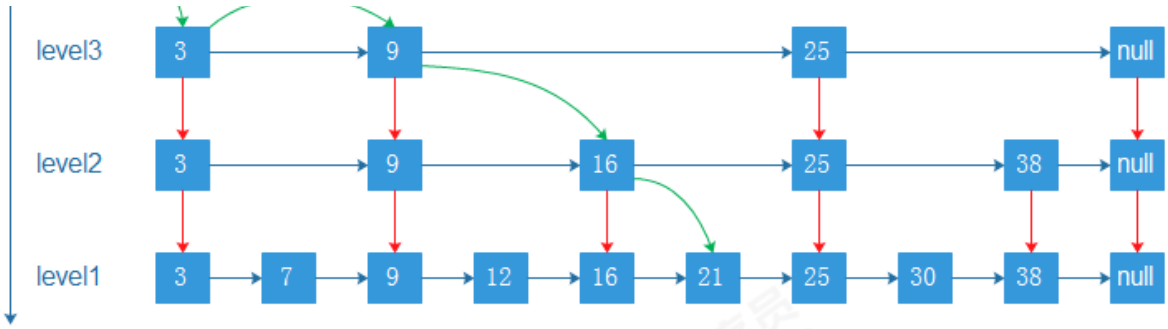
我们将上图再做一些扩展就可以变成一个典型的SkipList结构：



SkipListd的查找

对于上面我们我们要查找元素21，其过程如下：

1. 比较3，大于，往后找（9），
2. 比9大，继续往后找（25），但是比25小，则从9的下一层开始找（16）
3. 16的后面节点依然为25，则继续从16的下一层找
4. 找到21



绿线代表查询路径

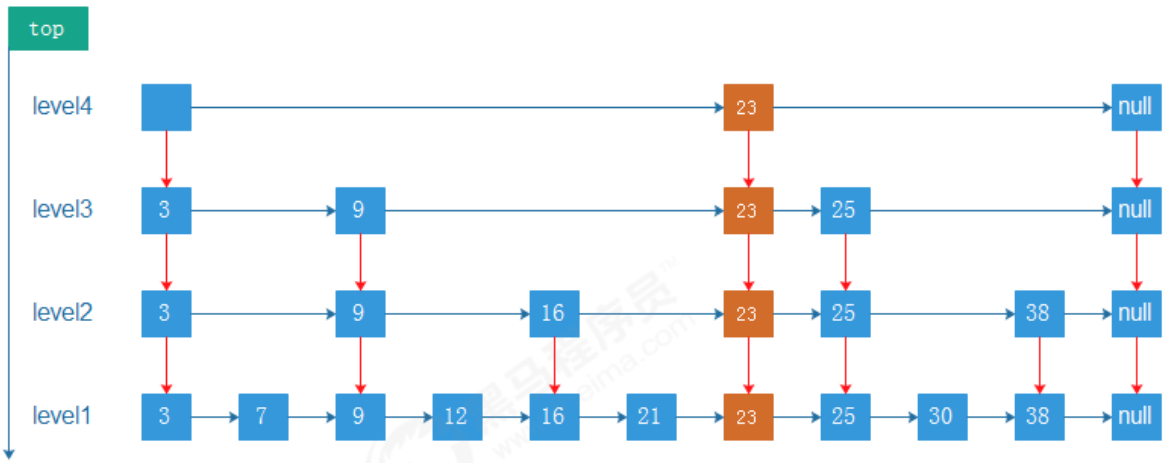
SkipList的插入

SkipList的插入操作主要包括：

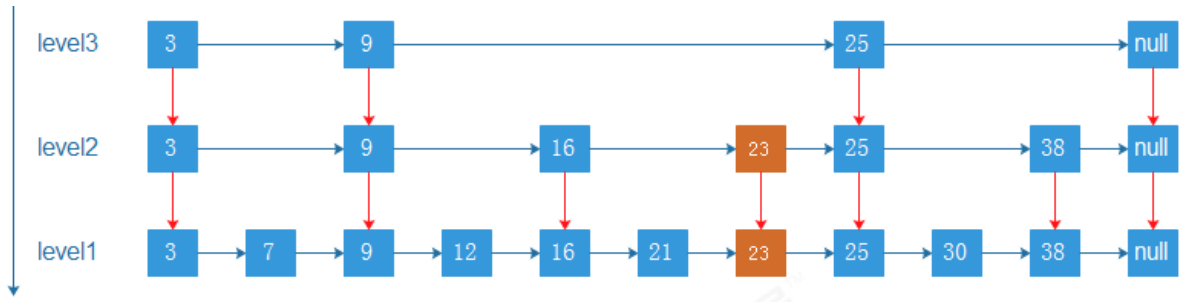
1. 查找合适的位置。这里需要明确一点就是在确认新节点要占据的层次K时，采用丢硬币的方式，完全随机。如果占据的层次K大于链表的层次，则重新申请新的层，否则插入指定层次
2. 申请新的节点
3. 调整指针

假定我们要插入的元素为23，经过查找可以确认她是位于25后，9、16、21前。当然需要考虑申请的层次K。

如果层次K > 3，需要申请新层次（Level 4）



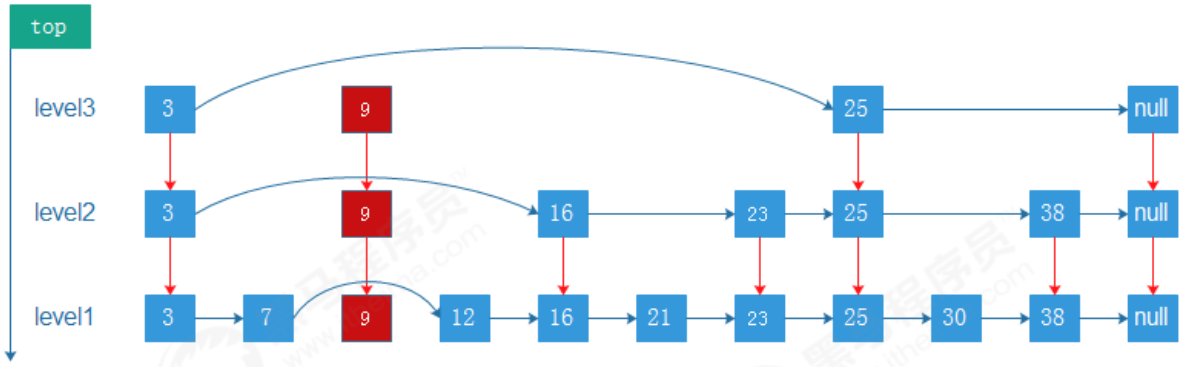
如果层次 K = 2，直接在Level 2 层插入即可



SkipList的删除

删除节点和插入节点思路基本一致：找到节点，删除节点，调整指针。

比如删除节点9，如下：



13 J.U.C队列

要实现一个线程安全的队列有两种方式：阻塞和非阻塞:

queue	阻塞与否	是否有界	线程安全保障	适用场景	注意事项
ConcurrentLinkedQueue	非阻塞	无界	CAS	对全局的集合进行操作的场景	size() 是要遍历一遍集合，慎用
ArrayBlockingQueue	阻塞	有界	一把全局锁	生产消费模型，平衡两边处理速度	--
LinkedBlockingQueue	阻	可	存取采用	生产消费模	无界的时候

queue	阻塞与否	是否有界	锁是否安全保障	适用场景	注意事项
PriorityBlockingQueue	阻塞	无界	全局锁	支持优先级排序	
SynchronousQueue	阻塞	无界	CAS	不存储元素的阻塞队列	

13.1 非阻塞队列ConcurrentLinkedQueue

在单线程编程中我们会经常用到一些集合类，比如ArrayList,HashMap等，但是这些类都不是线程安全的类。在面试中也经常会有一些考点，比如ArrayList不是线程安全的，Vector是线程安全。而保障Vector线程安全的方式，是非常粗暴的在方法上用synchronized独占锁，将多线程执行变成串行化。要想将ArrayList变成线程安全的也可以使用`Collections.synchronizedList(List<T> list)`方法ArrayList转换成线程安全的，但这种转换方式依然是通过synchronized修饰方法实现的，很显然这不是一种高效的方式，同时，队列也是我们常用的一种数据结构。

为了解决线程安全的问题，J.U.C为我们准备了ConcurrentLinkedQueue这个线程安全的队列。从类名就可以看的出来实现队列的数据结构是链式。ConcurrentLinkedQueue是一个基于链接节点的无边界的线程安全队列，遵循队列的FIFO原则，队尾入队，队首出队。采用CAS算法来实现的。

使用案例：

```
public class ConcurrentLinkedQueueDemo {
    public static void main(String[] args) throws Exception {
        Queue<String> queue = new ConcurrentLinkedQueue<String>
();
        for (int i = 0; i < 10000; i++) {
            //队列中添加元素
            queue.add(String.valueOf(i));
        }

        QueueDemo1 demo1 = new QueueDemo1(queue);

        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(demo1);
            t.start();
```

```
}
class QueueDemo1 implements Runnable {
    Queue<String> queue;

    public QueueDemo1(Queue<String> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            long start = new Date().getTime();
            //检索并移除此队列的头，如果此队列为空，则返回 null
            while (queue.poll() != null) {
                //if (queue.size() == 0) {
                //}

                if (queue.isEmpty()) {
                }
            }
            System.out.println(System.currentTimeMillis() -
start);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

注意：

1. ConcurrentLinkedQueue的.size() 是要遍历一遍集合的，很慢的，所以尽量要避免用size
2. 使用了这个ConcurrentLinkedQueue 类之后还是需要自己进行同步或加锁操作。例如queue.isEmpty()后再进行队列操作queue.add()是不能保证安全的，因为可能queue.isEmpty()执行完成后，别的线程开始操作队列。

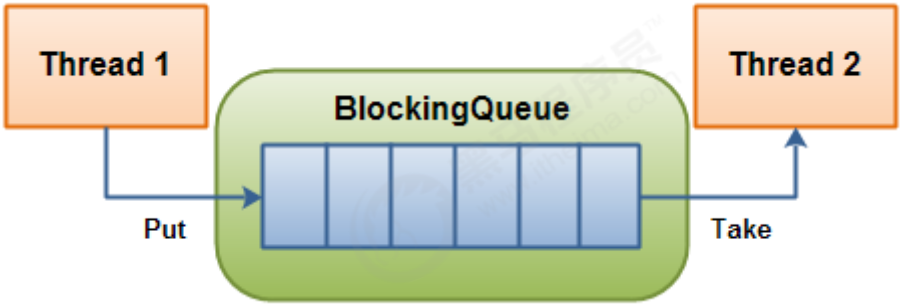
13.2 阻塞队列BlockingQueue

13.2.1 BlockingQueue介绍

BlockingQueue即阻塞队列，从阻塞这个词可以看出，在某些情况下对阻塞队列的访问可能会造成阻塞。被阻塞的情况主要有如下两种：

2. 当队列空了的时候进行出队列操作

因此，当一个线程试图对一个已经满了的队列进行入队列操作时，它将会被阻塞，除非有另一个线程做了出队列操作；同样，当一个线程试图对一个空队列进行出队列操作时，它将会被阻塞，除非有另一个线程进行了入队列操作。



BlockingQueue 对插入操作、移除操作、获取元素操作提供了四种不同的方法用于不同的场景中使用：

1. 抛出异常
2. 返回特殊值（null 或 true/false，取决于具体的操作）
3. 阻塞等待此操作，直到这个操作成功
4. 阻塞等待此操作，直到成功或者超时指定时间。总结如下：

操作	抛出异常	特殊值	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

接下来我们介绍这个接口的几个实现类。

13.2.1 ArrayBlockingQueue

ArrayBlockingQueue是一个由数组实现的有界阻塞队列。该队列采用FIFO的原则对元素进行排序添加的。

ArrayBlockingQueue为有界且固定，其大小在构造时由构造函数来决定，确认之后就不能再改变了。

策略，但是在默认情况下不保证线程公平的访问，在构造时可以选择公平策略（fair = true）。公平性通常会降低吞吐量，但是减少了可变性和避免了“不平衡性”。

ArrayBlockingQueue继承AbstractQueue，实现BlockingQueue接口。java.util.AbstractQueue，在Queue接口中扮演着非常重要的作用，该类提供了对queue操作的骨干实现。BlockingQueue继承java.util.Queue为阻塞队列的核心接口，提供了在多线程环境下的出列、入列操作，作为使用者，则不需要关心队列在什么时候阻塞线程，什么时候唤醒线程，所有一切均由BlockingQueue来完成。

ArrayBlockingQueue内部使用可重入锁ReentrantLock + Condition来完成多线程环境的并发操作。

- items，一个定长数组，维护ArrayBlockingQueue的元素
- takeIndex，int，为ArrayBlockingQueue队首位置
- putIndex，int，ArrayBlockingQueue队尾位置
- count，元素个数
- lock，锁，ArrayBlockingQueue出列入列都必须获取该锁，两个步骤公用一个锁

```
public class ArrayBlockingQueue<E> extends AbstractQueue<E>
implements BlockingQueue<E>, Serializable {
    private static final long serialVersionUID =
-8179116326528984 26L;
    final Object[] items;
    int takeIndex;
    int putIndex;
    int count;
    // 重入锁
    final ReentrantLock lock;
    // notEmpty condition
    private final Condition notEmpty;
    // notFull condition
    private final Condition notFull;
    transient ArrayBlockingQueue.Itrs itrs;
}
```

使用示例：

```
public class Demo6BlockingQueueTest {
    //最大容量为5的数组阻塞队列
```

```
    //private static LinkedBlockingQueue<Integer> queue = new
LinkedBlockingQueue<Integer>(5);

    public static void main(String[] args) {

        Thread t1 = new Thread(new ProducerTask());
        Thread t2 = new Thread(new ConsumerTask());

        //启动线程
        t1.start();
        t2.start();

    }

    //生产者
    static class ProducerTask implements Runnable {
        private Random rnd = new Random();

        @Override
        public void run() {
            try {
                while (true) {
                    int value = rnd.nextInt(100);
                    //如果queue容量已满，则当前线程会堵塞，直到有空间再继
续
                    queue.put(value);

                    System.out.println("生产者：" + value);

                    TimeUnit.MILLISECONDS.sleep(100); //线程休眠
                }
            } catch (Exception e) {
            }
        }
    }

    //消费者
    static class ConsumerTask implements Runnable {
        @Override
        public void run() {
            try {
                while (true) {
                    //如果queue为空，则当前线程会堵塞，直到有新数据加入
```

```
                    System.out.println("消费者:" + value);

                    TimeUnit.MILLISECONDS.sleep(15); //线程休眠
                }
            } catch (Exception e) {
            }
        }
    }
}
```

13.2.3 LinkedBlockingQueue

LinkedBlockingQueue和ArrayBlockingQueue的使用方式基本一样，但还是有一定的区别：

1. **队列的数据结构不同**

   ArrayBlockingQueue是一个由数组支持的有界阻塞队列

   LinkedBlockingQueue是一个基于链表的有界（可设置）阻塞队列

2. **队列中锁的实现不同**

   ArrayBlockingQueue实现的队列中的锁是没有分离的，即生产和消费用的是同一个锁；

   LinkedBlockingQueue实现的队列中的锁是分离的，即生产用的是putLock，消费是takeLock

3. **在生产或消费时操作不同**

   ArrayBlockingQueue实现的队列中在生产和消费的时候，是直接将枚举对象插入或移除的；

   LinkedBlockingQueue实现的队列中在生产和消费的时候，需要把枚举对象转换为Node进行插入或移除，会影响性能

4. **队列大小初始化方式不同**

   ArrayBlockingQueue实现的队列中必须指定队列的大小；

Integer.MAX_VALUE

13.2.4 PriorityBlockingQueue

PriorityBlockingQueue类似于ArrayBlockingQueue内部使用一个独占锁来控制，同时只有一个线程可以进行入队和出队。

PriorityBlockingQueue是一个优先级队列，它在java.util.PriorityQueue的基础上提供了可阻塞的读取操作。它是无界的，就是说向Queue里面增加元素没有数量限制，但可能会导致内存溢出而失败。

PriorityBlockingQueue始终保证出队的元素是优先级最高的元素，并且可以定制优先级的规则，内部使用二叉堆，通过使用一个二叉树最小堆算法来维护内部数组，这个数组是可扩容的，当当前元素个数>=最大容量时候会通过算法扩容。值得注意的是为了避免在扩容操作时候其他线程不能进行出队操作，实现上使用了先释放锁，然后通过CAS保证同时只有一个线程可以扩容成功。

小结：

1、优先队列不允许空值，而且不支持non-comparable（不可比较）的对象，比如用户自定义的类。优先队列要求使用Java Comparable和Comparator接口给对象排序，并且在排序时会按照优先级处理其中的元素。

2、优先队列的头是基于自然排序或者Comparator排序的最小元素。如果有多个对象拥有同样的排序，那么就可能随机地取其中任意一个。也可以通过提供的Comparator（比较器）在队列实现自定的排序。当我们获取队列时，返回队列的头对象。

3、优先队列的大小是不受限制的，但在创建时可以指定初始大小，当我们向优先队列增加元素的时候，队列大小会自动增加。

4、PriorityQueue是非线程安全的，所以Java提供了PriorityBlockingQueue（实现BlockingQueue接口）用于Java多线程环境。

使用案例：

```
public class Demo7PriorityBlockQueue {

    public static void main(String[] args) throws
InterruptedException {
        PriorityBlockingQueue<User> queue = new
PriorityBlockingQueue<User>();
```

```
        for (int i = 0; i < 5; i++) {
            new Thread(demo).start();
        }

        Thread.sleep(100);

        User u = queue.poll();
        while (u != null) {
            System.out.println("优先级是: " + u.getPriority() + ","
+ u.getUsername());
            u = queue.poll();
        }
    }

    static class PriorityDemo implements Runnable {

        PriorityBlockingQueue queue;
        Random r = new Random();

        public PriorityDemo(PriorityBlockingQueue queue) {
            this.queue = queue;
        }

        @Override
        public void run() {
            for (int i = 0; i < 3; i++) {
                User user = new User();
                user.setPriority(r.nextInt(100));
                user.setUsername("张三" + i);

                queue.add(user);
            }
        }
    }

    static class User implements Comparable<User> {

        private Integer priority;
        private String username;

        @Override
        public int compareTo(User user) {
```

```
            return this.priority.compareTo(user.getPriority());
        }

        public Integer getPriority() {
            return priority;
        }

        public void setPriority(Integer priority) {
            this.priority = priority;
        }

        public String getUsername() {
            return username;
        }

        public void setUsername(String username) {
            this.username = username;
        }
    }
}
```

13.2.5 SynchronousQueue

SynchronousQueue，实际上它不是一个真正的队列，因为它不会为队列中元素维护存储空间。与其他队列不同的是，它维护一组线程，这些线程在等待着把元素加入或移出队列。SynchronousQueue没有存储功能，因此put和take会一直阻塞，直到有另一个线程已经准备好参与到交付过程中。

仅当有足够多的消费者，并且总是有一个消费者准备好获取交付的工作时，才适合使用同步队列。这种实现队列的方式看似很奇怪，但由于可以直接交付工作，从而降低了将数据从生产者移动到消费者的延迟。

直接交付方式还会将更多关于任务状态的信息反馈给生产者。当交付被接受时，它就知道消费者已经得到了任务，而不是简单地把任务放入一个队列——这种区别就好比将文件直接交给同事，还是将文件放到她的邮箱中并希望她能尽快拿到文件。

SynchronousQueue对于正在等待的生产者和使用者线程而言，默认是非公平排序，也可以选择公平排序策略。但是，使用公平所构造的队列可保证线程以 FIFO 的顺序进行访问。 公平通常会降低吞吐量，但是可以减小可变性并避免得不到服务。

- 是一种阻塞队列，其中每个 put 必须等待一个 take，反之亦然。同步队列没有任何内部容量，甚至连一个队列的容量都没有。
- 是线程安全的，是阻塞的。
- 不允许使用 null 元素。
- 公平排序策略是指调用put的线程之间，或take的线程之间的线程以 FIFO 的顺序进行访问。
- SynchronousQueue的方法：
  - iterator()： 永远返回空，因为里面没东西。
  - peek()： 永远返回null。
  - put()： 往queue放进去一个element以后就一直wait直到有其他thread进来把这个element取走。
  - offer()： 往queue里放一个element后立即返回，如果碰巧这个element被另一个thread取走了，offer方法返回true，认为offer成功；否则返回false。
  - offer(2000, TimeUnit.SECONDS)： 往queue里放一个element但等待时间后才返回，和offer()方法一样。
  - take()： 取出并且remove掉queue里的element，取不到东西他会一直等。
  - poll()： 取出并且remove掉queue里的element，方法立即能取到东西返回。否则立即返回null。
  - poll(2000, TimeUnit.SECONDS)： 等待时间后再取，并且remove掉queue里的element,
  - isEmpty()：永远是true。
  - remainingCapacity()： 永远是0。
  - remove()和removeAll()： 永远是false。

使用案例：

```
public class Demo8SynchronousQueue {
    public static void main(String[] args) throws
InterruptedException {
        SynchronousQueue<Integer> queue = new
SynchronousQueue<Integer>();

        new Thread(new Product(queue)).start();
        new Thread(new Customer(queue)).start();
    }
```

```
        Random r = new Random();

        public Product(SynchronousQueue<Integer> queue) {
            this.queue = queue;
        }

        @Override
        public void run() {
            while (true) {
                int number = r.nextInt(1000);
                System.out.println("等待1秒后运送" + number);
                try {
                    TimeUnit.SECONDS.sleep(1);

                    queue.put(number);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

    static class Customer implements Runnable {
        SynchronousQueue<Integer> queue;

        public Customer(SynchronousQueue<Integer> queue) {
            this.queue = queue;
        }

        @Override
        public void run() {
            while (true) {
                try {
                    System.out.println("收到了:" + queue.take());
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

线程是一个程序员一定会涉及到的概念，但是线程的创建和切换都是代价比较大的。所以，我们需要有一个好的方案能做到线程的复用，这就涉及到一个概念——线程池。合理的使用线程池能够带来3个很明显的好处：

1. 降低资源消耗：通过重用已经创建的线程来降低线程创建和销毁的消耗
2. 提高响应速度：任务到达时不需要等待线程创建就可以立即执行。
3. 提高线程的可管理性：线程池可以统一管理、分配、调优和监控。

java的线程池支持主要通过ThreadPoolExecutor来实现，我们使用的ExecutorService的各种线程池策略都是基于ThreadPoolExecutor实现的，所以ThreadPoolExecutor十分重要。要弄明白各种线程池策略，必须先弄明白ThreadPoolExecutor。

14.1 线程池状态

线程池同样有五种状态：Running, SHUTDOWN, STOP, TIDYING, TERMINATED。

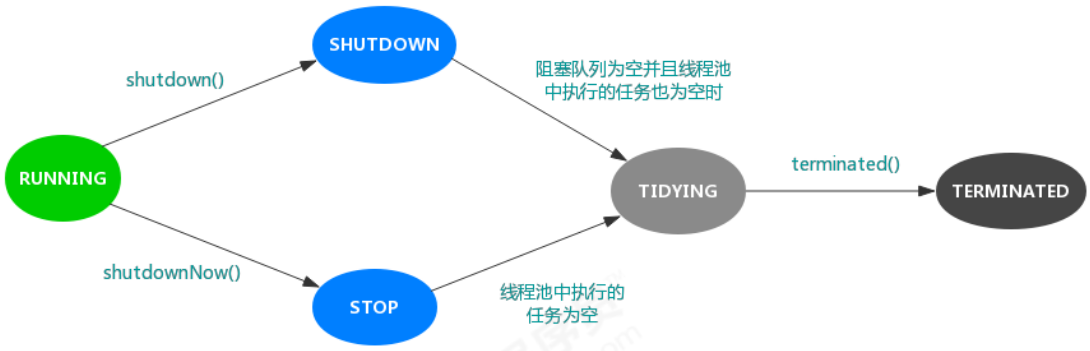
```
    private final AtomicInteger ctl = new
AtomicInteger(ctlOf(RUNNING, 0));
    private static final int COUNT_BITS = Integer.SIZE - 3;
    private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

    // runState is stored in the high-order bits
    private static final int RUNNING    = -1 << COUNT_BITS;//对应
的高3位值是111
    private static final int SHUTDOWN   =  0 << COUNT_BITS;//对应
的高3位值是000
    private static final int STOP       =  1 << COUNT_BITS;//对应
的高3位值是001
    private static final int TIDYING    =  2 << COUNT_BITS;//对应
的高3位值是010
    private static final int TERMINATED =  3 << COUNT_BITS;//对应
的高3位值是011

    // Packing and unpacking ctl
    private static int runStateOf(int c)     { return c &
~CAPACITY; }
```

```
    private static int ctlOf(int rs, int wc) { return rs | wc; }
```

变量**ctl**定义为AtomicInteger，记录了"线程池中的任务数量"和"线程池的状态"两个信息。共32位，其中高3位表示"线程池状态"，低29位表示"线程池中的任务数量"。

- RUNNING：处于RUNNING状态的线程池能够接受新任务，以及对新添加的任务进行处理。
- SHUTDOWN：处于SHUTDOWN状态的线程池不可以接受新任务，但是可以对已添加的任务进行处理。
- STOP：处于STOP状态的线程池不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。
- TIDYING：当所有的任务已终止，ctl记录的"任务数量"为0，线程池会变为TIDYING状态。当线程池变为TIDYING状态时，会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。
- TERMINATED：线程池彻底终止的状态。

各个状态的转换如下：



14.2 构造方法

我现在分析线程池参数最全的构造方法，了解其内部的参数意义

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
```

```
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler ==
null)
            throw new NullPointerException();
        this.acc = System.getSecurityManager() == null ?
            null :
        AccessController.getContext();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

共有七个参数，每个参数含义如下：

- corePoolSize

  线程池中核心线程的数量（也称为线程池的基本大小）。当提交一个任务时，线程池会新建一个线程来执行任务，直到当前线程数等于corePoolSize。如果调用了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有基本线程。

- maximumPoolSize

  线程池中允许的最大线程数。线程池的阻塞队列满了之后，如果还有任务提交，如果当前的线程数小于maximumPoolSize，则会新建线程来执行任务。注意，如果使用的是无界队列，该参数也就没有什么效果了。

- keepAliveTime

  线程空闲的时间。线程的创建和销毁是需要代价的。线程执行完任务后不会立即销毁，而是继续存活一段时间：keepAliveTime。默认情况下，该参数只有在线程数大于corePoolSize时才会生效。

- unit

  keepAliveTime的单位。TimeUnit

- workQueue

  用来保存等待执行的任务的BlockQueue阻塞队列，等待的任务必须实现Runnable接口。选择如下：

  > ArrayBlockingQueue：基于数组结构的有界阻塞队列，FIFO。
  > LinkedBlockingQueue：基于链表结构的有界阻塞队列，FIFO。
  > PriorityBlockingQueue：具有优先级别的阻塞队列。
  > SynchronousQueue：不存储元素的阻塞队列，每个插入操作都必须等待一个移出操作。

- threadFactory

  用于设置创建线程的工厂。ThreadFactory的作用就是提供创建线程的功能的线程工厂。他是通过newThread()方法提供创建线程的功能，newThread()方法创建的线程都是“非守护线程”而且“线程优先级都是默认优先级”。

- handler

  RejectedExecutionHandler，线程池的拒绝策略。所谓拒绝策略，是指将任务添加到线程池中时，线程池拒绝该任务所采取的相应策略。当向线程池中提交任务时，如果此时线程池中的线程已经饱和了，而且阻塞队列也已经满了，则线程池会选择一种拒绝策略来处理该任务。

  线程池提供了四种拒绝策略：

  > AbortPolicy：直接抛出异常，默认策略；
  > CallerRunsPolicy：用调用者所在的线程来执行任务；
  > DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
  > DiscardPolicy：直接丢弃任务；
  > 当然我们也可以实现自己的拒绝策略，例如记录日志等等，实现RejectedExecutionHandler接口即可。

14.3 四种线程池

我们除了可以使用ThreadPoolExecutor自己根据实际情况创建线程池以外，Executor框架也提供了三种线程池，他们都可以通过工具类Executors来创建。

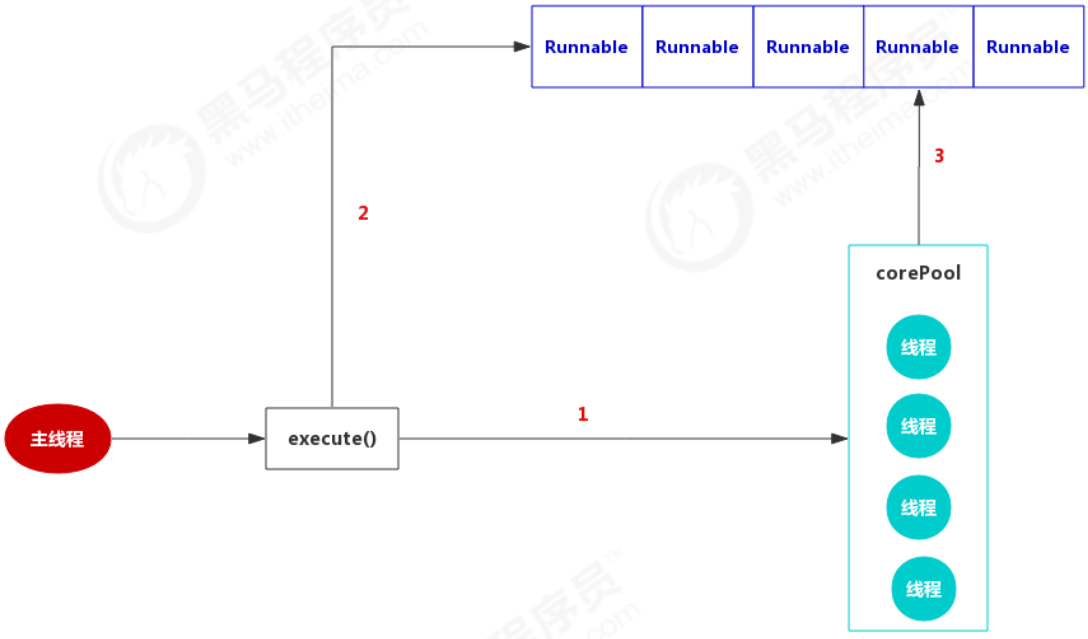
扩展了功能的ThreadPoolExecutor

14.3.1 FixedThreadPool

FixedThreadPool是复用**固定数量的线程**处理一个**共享的无边界队列**，其定义如下：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new
LinkedBlockingQueue<Runnable>());
}
```

corePoolSize 和 maximumPoolSize都设置为创建FixedThreadPool时指定的参数nThreads，由于该线程池是固定线程数的线程池，当线程池中的线程数量等于corePoolSize 时，如果继续提交任务，该任务会被添加到阻塞队列workQueue中，而workQueue使用的是LinkedBlockingQueue，但没有设置范围，那么则是最大值（Integer.MAX_VALUE），这基本就相当于一个无界队列了。



案例：

```
public class Demo9FixedThreadPoolCase {

    public static void main(String[] args) throws
InterruptedException {
        ExecutorService exec = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 5; i++) {
            exec.execute(new Demo());
            Thread.sleep(10);
```

```
    }

    static class Demo implements Runnable {
        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            for (int i = 0; i < 2; i++) {
                System.out.println(name + ":" + i);
            }
        }
    }
}
```

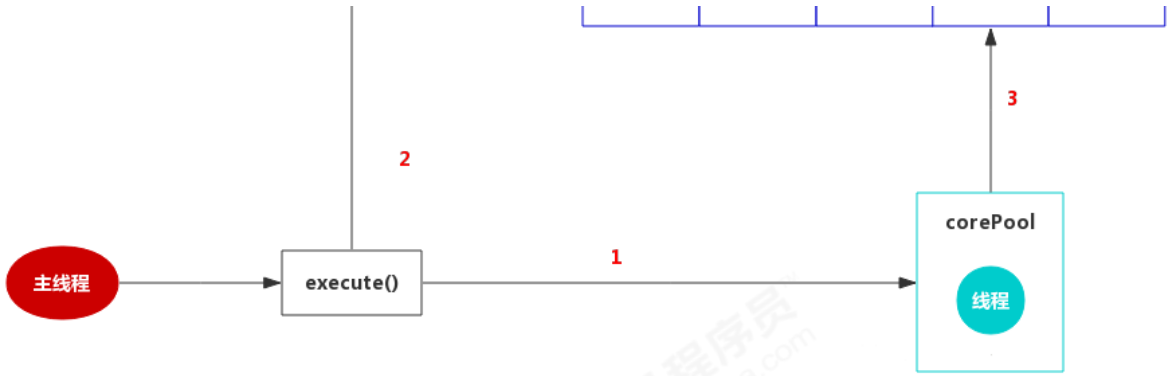
14.3.2 SingleThreadExecutor

SingleThreadExecutor只会使用单个工作线程来执行一个无边界的队列。

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>
()));
}
```

作为单一worker线程的线程池，它把corePool和maximumPoolSize均被设置为1，和FixedThreadPool一样使用的是无界队列LinkedBlockingQueue,所以带来的影响和FixedThreadPool一样。

SingleThreadExecutor只会使用单个工作线程，它可以保证认为是按顺序执行的，任何时候都不会有多于一个的任务处于活动状态。注意，如果单个线程在执行过程中因为某些错误中止，新的线程会替代它执行后续线程。



案例：

```
public class Demo9SingleThreadPoolCase {
    static int count = 0;

    public static void main(String[] args) throws
InterruptedException {
        ExecutorService exec =
Executors.newSingleThreadExecutor();
        for (int i = 0; i < 10; i++) {
            exec.execute(new Demo());
            Thread.sleep(5);
        }
        exec.shutdown();
    }

    static class Demo implements Runnable {
        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            for (int i = 0; i < 2; i++) {
                count++;
                System.out.println(name + ":" + count);
            }
        }
    }
}
```

14.3.3 CachedThreadPool

CachedThreadPool会根据需要，在线程可用时，重用之前构造好的池中线程，否则创建新线程：

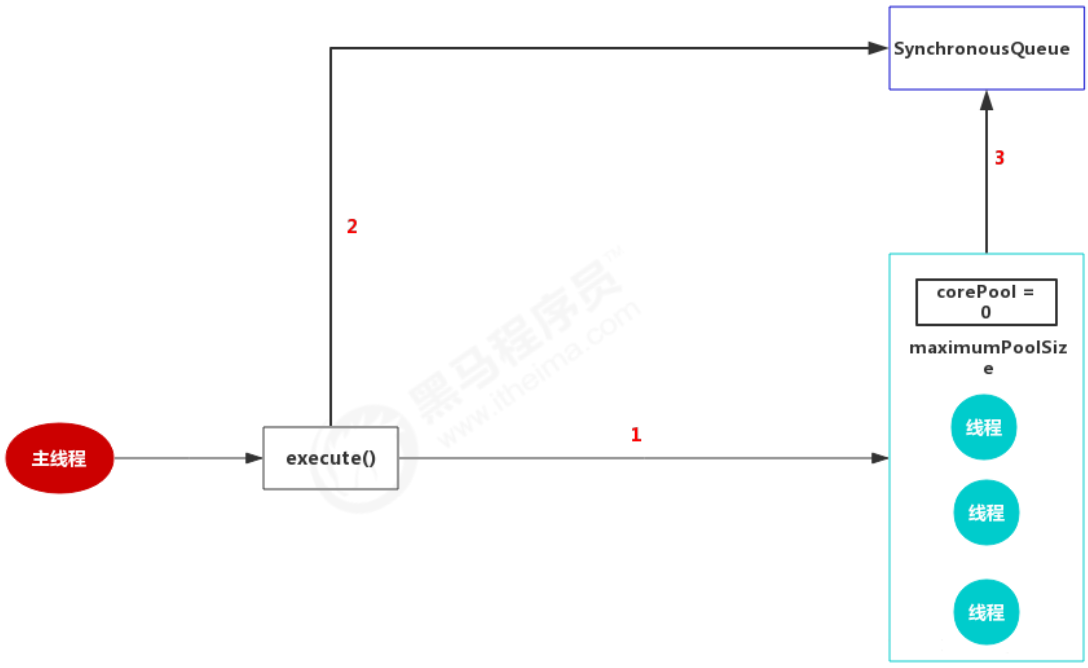
```
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>
());
}
```

它把corePool为0，maximumPoolSize为Integer.MAX_VALUE，这就意味着所有的任务一提交就会加入到阻塞队列中。因为线程池的基本大小设置为0，一般情况下线程池中没有程池，用的时候再创建。

但是keepAliveTime设置60，unit设置为秒，意味着空闲线程等待新任务的最长时间为60秒，空闲线程超过60秒后将会被终止。阻塞队列采用的SynchronousQueue，这是是一个没有元素的阻塞队列。

这个线程池在执行 大量短生命周期的异步任务时，可以显著提高程序性能。调用 execute 时，可以重用之前已构造的可用线程，如果不存在可用线程，那么会重新创建一个新的线程并将其加入到线程池中。如果线程超过 60 秒还未被使用，就会被中止并从缓存中移除。因此，线程池在长时间空闲后不会消耗任何资源。

但是这样就处理线程池会存在一个问题，如果主线程提交任务的速度远远大于CachedThreadPool的处理速度，则CachedThreadPool会不断地创建新线程来执行任务，这样有可能会导致系统耗尽CPU和内存资源，所以在使用该线程池是，一定要注意控制并发的任务数，否则创建大量的线程可能导致严重的性能问题。



案例：

```
InterruptedException {
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
            exec.execute(new Demo());
            Thread.sleep(1);
        }
        exec.shutdown();
    }

    static class Demo implements Runnable {
        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            try {
                //修改睡眠时间，模拟线程执行需要花费的时间
                Thread.sleep(1);

                System.out.println(name + "执行完了");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

14.3.4 ScheduledThreadPool

Timer与TimerTask虽然可以实现线程的周期和延迟调度，但是Timer与TimerTask存在一些问题：

- Timer在执行定时任务时只会创建一个线程，所以如果存在多个任务，且任务时间过长，超过了两个任务的间隔时间，会发生一些缺陷。
- 如果TimerTask抛出RuntimeException，Timer会停止所有任务的运行。
- Timer执行周期任务时依赖系统时间，如果当前系统时间发生变化会出现一些执行上的变化

为了解决这些问题，我们一般都是推荐ScheduledThreadPoolExecutor来实现。

ScheduledExecutorService接口，它就相当于提供了“延迟”和“周期执行”功能的ThreadPoolExecutor。

ScheduledThreadPoolExecutor，它可另行安排在给定的延迟后运行命令，或者定期执行命令。需要多个辅助线程时，或者要求 ThreadPoolExecutor 具有额外的灵活性或功能时，此类要优于Timer。

提供了四种构造方法：

```
    public ScheduledThreadPoolExecutor(int corePoolSize) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue());
    }

    public ScheduledThreadPoolExecutor(int corePoolSize,
                                       ThreadFactory
threadFactory) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), threadFactory);
    }

    public ScheduledThreadPoolExecutor(int corePoolSize,
                                       RejectedExecutionHandler
handler) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), handler);
    }


    public ScheduledThreadPoolExecutor(int corePoolSize,
                                       ThreadFactory
threadFactory,
                                       RejectedExecutionHandler
handler) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), threadFactory, handler);
    }
```

在ScheduledThreadPoolExecutor的构造函数中，我们发现它都是利用ThreadLocalExecutor来构造的，唯一变动的地方就在于它所使用的阻塞队列变成了DelayedWorkQueue。

队列和优先级队列。在执行定的任务的时候，每个任务的执行的间都不同，所以DelayedWorkQueue的工作就是按照执行时间的升序来排列，执行时间距离当前时间越近的任务在队列的前面，这样就可以保证每次出队的任务都是当前队列中执行时间最靠前的。

案例：

```
public class Demo9ScheduledThreadPool {

    public static void main(String[] args) throws
InterruptedException {
        ScheduledExecutorService scheduledThreadPool =
Executors.newScheduledThreadPool(2);
        System.out.println("程序开始：" + new Date());
        // 第二个参数是延迟多久执行
        scheduledThreadPool.schedule(new Task(), 0,
TimeUnit.SECONDS);
        scheduledThreadPool.schedule(new Task(), 1,
TimeUnit.SECONDS);
        scheduledThreadPool.schedule(new Task(), 5,
TimeUnit.SECONDS);

        Thread.sleep(5000);

        // 关闭线程池
        scheduledThreadPool.shutdown();
    }

    static class Task implements Runnable {
        @Override
        public void run() {
            try {
                String name = Thread.currentThread().getName();

                System.out.println(name + ", 开始：" + new
Date());
                Thread.sleep(1000);
                System.out.println(name + ", 结束：" + new
Date());

            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
        }
    }
}
```