线程池

一、什么是线程池

为什么要使用线程池

在开发中，为了提升效率的操作，我们需要将一些业务采用多线程的方式去执行。

比如有一个比较大的任务，可以将任务分成几块，分别交给几个线程去执行，最终做一个汇总就可以了。

比如做业务操作时，需要发送短信或者是发送邮件，这种操作也可以基于异步的方式完成，这种异步的方式，其实就是再构建一个线程去执行。

但是，如果每次异步操作或者多线程操作都需要新创建一个线程，使用完毕后，线程再被销毁，这样的话，对系统造成一些额外的开销。在处理过程中到底由多线程处理了多少个任务，以及每个线程的开销无法统计和管理。

所以咱们需要一个线程池机制来管理这些内容。线程池的概念和连接池类似，都是在一个Java的集合中存储大量的线程对象，每次需要执行异步操作或者多线程操作时，不需要重新创建线程，直接从集合中拿到线程对象直接执行方法就可以了。

JDK中就提供了线程池的类。

在线程池构建初期，可以将任务提交到线程池中。会根据一定的机制来异步执行这个任务。

- 可能任务直接被执行
- 任务可以暂时被存储起来了。等到有空闲线程再来处理。
- 任务也可能被拒绝，无法被执行。

JDK提供的线程池中记录了每个线程处理了多少个任务，以及整个线程池处理了多少个任务。同时还可以针对任务执行前后做一些勾子函数的实现。可以在任务执行前后做一些日志信息，这样可以多记录信息方便后面统计线程池执行任务时的一些内容参数等等.....

二、JDK自带的构建线程池的方式

JDK中基于Executors提供了很多种线程池

2.1 newFixedThreadPool

这个线程池的特别是线程数是固定的。

在Executors中第一个方法就是构建newFixedThreadPool

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

构建时，需要给newFixedThreadPool方法提供一个nThreads的属性，而这个属性其实就是当前线程池中线程的个数。当前线程池的本质其实就是使用ThreadPoolExecutor。

构建好当前线程池后，线程个数已经固定好（**线程是懒加载，在构建之初，线程并没有构建出来，而是随着人任务的提交才会将线程在线程池中国构建出来**）。如果线程没构建，线程会待着任务执行被创建和执行。如果线程都已经构建好了，此时任务会被放到LinkedBlockingQueue无界队列中存放，等待线程从LinkedBlockingQueue中去take出任务，然后执行。

测试功能效果

```
public static void main(String[] args) throws Exception {
    ExecutorService threadPool = Executors.newFixedThreadPool(3);
    threadPool.execute(() -> {
        System.out.println("1号任务：" + Thread.currentThread().getName() +
System.currentTimeMillis());
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    threadPool.execute(() -> {
        System.out.println("2号任务：" + Thread.currentThread().getName() +
System.currentTimeMillis());
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    threadPool.execute(() -> {
        System.out.println("3号任务：" + Thread.currentThread().getName() +
System.currentTimeMillis());
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
}
```

2.2 newSingleThreadExecutor

这个线程池看名字就知道是单例线程池，线程池中只有一个工作线程在处理任务

如果业务涉及到顺序消费，可以采用newSingleThreadExecutor

```
// 当前这里就是构建单例线程池的方式
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        // 在内部依然是构建了ThreadPoolExecutor，设置的线程个数为1
        // 当任务投递过来后，第一个任务会被工作线程处理，后续的任务会被扔到阻塞队列中
        // 投递到阻塞队列中任务的顺序，就是工作线程处理的顺序
        // 当前这种线程池可以用作顺序处理的一些业务中
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

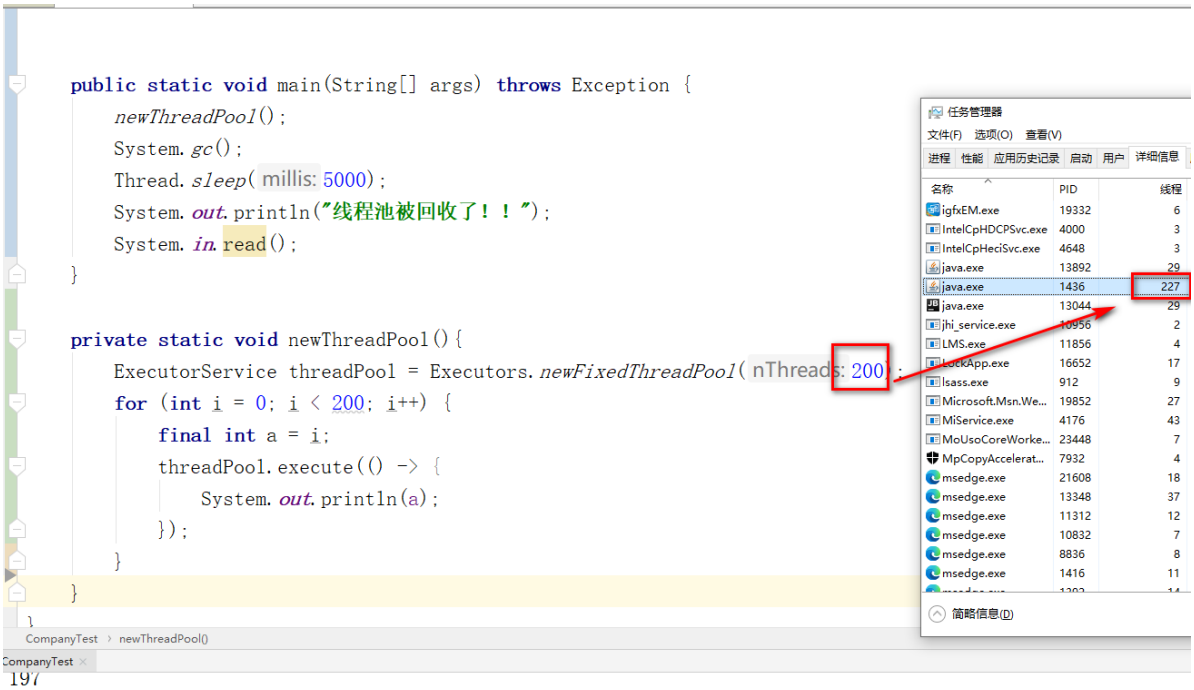
```
static class FinalizableDelegatedExecutorService extends
DelegatedExecutorService {
    // 线程池的使用没有区别，跟正常的ThreadPoolExecutor没区别
    FinalizableDelegatedExecutorService(ExecutorService executor) {
        super(executor);
    }
    // finalize是当前对象被GC干掉之前要执行的方法
    // 当前FinalizableDelegatedExecutorService的目的是为了在当前线程池被GC回收之前
    // 可以执行shutdown，shutdown方法是将当前线程池停止，并且干掉工作线程
    // 但是不能基于这种方式保证线程池一定会执行shutdown
    // finalize在执行时，是守护线程，这种线程无法保证一定可以执行完毕。
    // 在使用线程池时，如果线程池是基于一个业务构建的，在使用完毕之后，一定要手动执行
shutdown，
    // 否则会造成JVM中一堆线程
    protected void finalize() {
        super.shutdown();
    }
}
```

测试单例线程池效果：

```
public static void main(String[] args) throws Exception {
    ExecutorService threadPool = Executors.newSingleThreadExecutor();

    threadPool.execute(() -> {
        System.out.println(Thread.currentThread().getName() + "," + "111");
    });
    threadPool.execute(() -> {
        System.out.println(Thread.currentThread().getName() + "," + "222");
    });
    threadPool.execute(() -> {
        System.out.println(Thread.currentThread().getName() + "," + "333");
    });
    threadPool.execute(() -> {
        System.out.println(Thread.currentThread().getName() + "," + "444");
    });
}
```

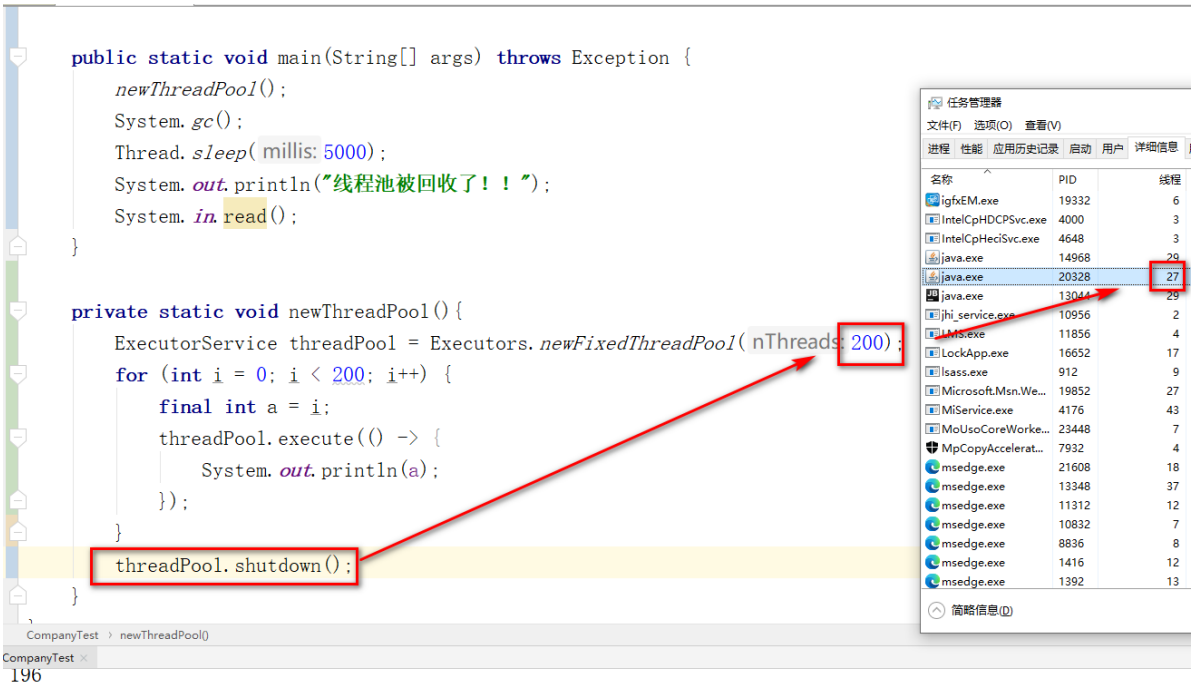
测试线程池使用完毕后，不执行shutdown的后果：

如果是局部变量仅限当前线程池使用的线程池，在使用完毕之后要记得执行shutdown，避免线程无法结束




```
197
198
199
```

线程池被回收了！！




```
196
197
198
199
```

如果是全局的线程池，很多业务都会到，使用完毕后不要shutdown，因为其他业务也要执行当前线程池

```
Exception in thread "main" java.util.concurrent.RejectedExecutionException
    at com.mashibing.CompanyTest.newThreadPool(CompanyTest.java:31)
    at com.mashibing.CompanyTest.main(CompanyTest.java:14)
```

```
static ExecutorService threadPool = Executors.newFixedThreadPool(200);

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

```
    newThreadPool();
    System.gc();
    Thread.sleep(5000);
    System.out.println("线程池被回收了！！");
    System.in.read();
}

private static void newThreadPool(){
    for (int i = 0; i < 200; i++) {
        final int a = i;
        threadPool.execute(() -> {
            System.out.println(a);
        });
    }
    threadPool.shutdown();
    for (int i = 0; i < 200; i++) {
        final int a = i;
        threadPool.execute(() -> {
            System.out.println(a);
        });
    }
}
```

2.3 newCachedThreadPool

看名字好像是一个缓存的线程池，查看一下构建的方式

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

当第一次提交任务到线程池时，会直接构建一个工作线程

这个工作线程带执行完人后，60秒没有任务可以执行后，会结束

如果在等待60秒期间有任务进来，他会再次拿到这个任务去执行

如果后续提升任务时，没有线程是空闲的，那么就构建工作线程去执行。

最大的一个特点，**任务只要提交到当前的newCachedThreadPool中，就必然有工作线程可以处理**

代码测试效果

```
public static void main(String[] args) throws Exception {
    ExecutorService executorService = Executors.newCachedThreadPool();
    for (int i = 1; i <= 200; i++) {
        final int j = i;
        executorService.execute(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + ":" + j);
        });
    }
```

```
}
```

2.4 newScheduleThreadPool

首先看到名字就可以猜到当前线程池是一个定时任务的线程池，而这个线程池就是可以以一定周期去执行一个任务，或者是延迟多久执行一个任务一次

查看一下如何构建的。

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
{
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
```

基于这个方法可以看到，构建的是ScheduledThreadPoolExecutor线程池

```
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor{
    //....
}
```

所以本质上还是正常线程池，只不过在原来的线程池基础上实现了定时任务的功能

原理是基于DelayQueue实现的延迟执行。周期性执行是任务执行完毕后，再次扔回到阻塞队列。

代码查看使用的方式和效果

```
public static void main(String[] args) throws Exception {
    ScheduledExecutorService pool = Executors.newScheduledThreadPool(10);

    // 正常执行
//        pool.execute(() -> {
//            System.out.println(Thread.currentThread().getName() + ": 1");
//        });

    // 延迟执行，执行当前任务延迟5s后再执行
//        pool.schedule(() -> {
//            System.out.println(Thread.currentThread().getName() + ": 2");
//        },5,TimeUnit.SECONDS);

    // 周期执行，当前任务第一次延迟5s执行，然后没3s执行一次
    // 这个方法在计算下次执行时间时，是从任务刚刚开始时就计算。
//        pool.scheduleAtFixedRate(() -> {
//            try {
//                Thread.sleep(3000);
//            } catch (InterruptedException e) {
//                e.printStackTrace();
//            }
//            System.out.println(System.currentTimeMillis() + ": 3");
//        },2,1,TimeUnit.SECONDS);

    // 周期执行，当前任务第一次延迟5s执行，然后没3s执行一次
    // 这个方法在计算下次执行时间时，会等待任务结束后，再计算时间
    pool.scheduleWithFixedDelay(() -> {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        System.out.println(System.currentTimeMillis() + ": 3");
    },2,1,TimeUnit.SECONDS);
}
```

至于Executors提供的newSingleThreadScheduledExecutor单例的定时任务线程池就不说了。

一个线程的线程池可以延迟或者以一定的周期执行一个任务。

2.5 newWorkStealingPool

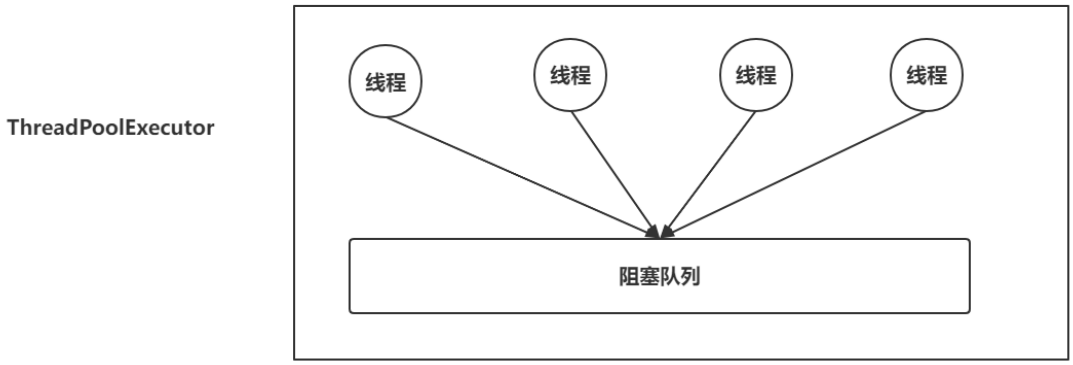
当前JDK提供构建线程池的方式newWorkStealingPool和之前的线程池很非常大的区别

之前定长，单例，缓存，定时任务都基于ThreadPoolExecutor去实现的。

newWorkStealingPool是基于ForkJoinPool构建出来的

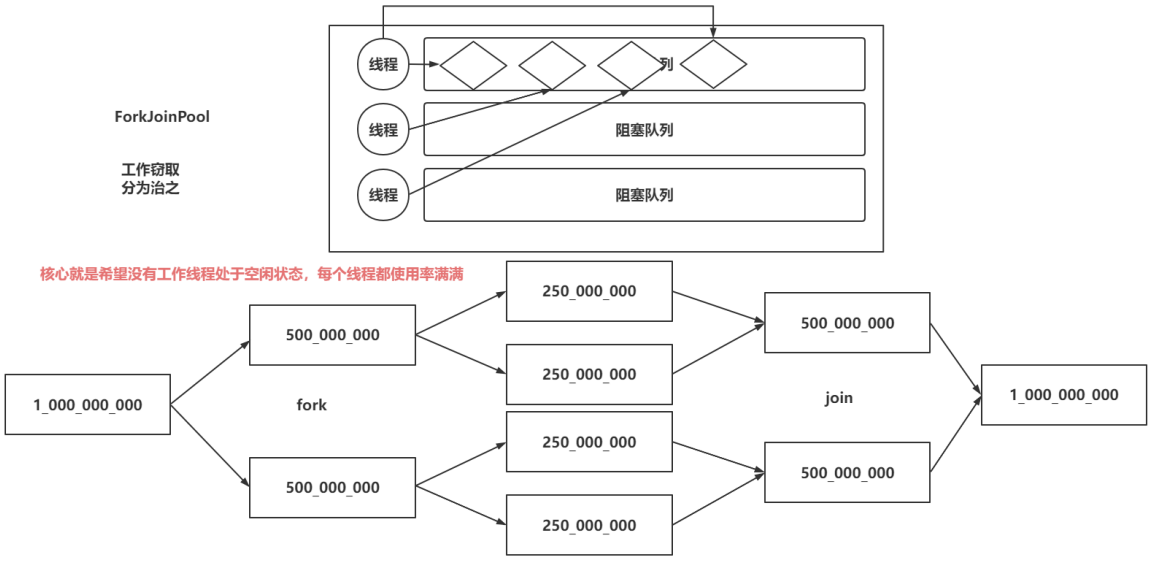
ThreadPoolExecutor的核心点：

在ThreadPoolExecutor中只有一个阻塞队列存放当前任务



ForkJoinPool的核心特点：

ForkJoinPool从名字上就能看出一些东西。当有一个特别大的任务时，如果采用上述方式，这个大任务只能会某一个线程去执行。ForkJoin第一个特点是可以将一个大任务拆分成多个小任务，放到当前线程的阻塞队列中。其他的空闲线程就可以去处理有任务的线程的阻塞队列中的任务



来一个比较大的数组，里面存满值，计算总和

单线程处理一个任务：

```
/** 非常大的数组 */
static int[] nums = new int[1_000_000_000];
// 填充值
static{
    for (int i = 0; i < nums.length; i++) {
        nums[i] = (int) ((Math.random()) * 1000);
    }
}
public static void main(String[] args) {
    // ====================单线程累加10亿数据================================
    System.out.println("单线程计算数组总和！");
    long start = System.nanoTime();
    int sum = 0;
    for (int num : nums) {
        sum += num;
    }
    long end = System.nanoTime();
    System.out.println("单线程运算结果为：" + sum + "，计算时间为：" + (end  -
start));
}
```

多线程分而治之的方式处理：

```
/** 非常大的数组 */
static int[] nums = new int[1_000_000_000];
// 填充值
static{
    for (int i = 0; i < nums.length; i++) {
        nums[i] = (int) ((Math.random()) * 1000);
    }
}
public static void main(String[] args) {
    // ====================单线程累加10亿数据================================
    System.out.println("单线程计算数组总和！");
    long start = System.nanoTime();
    int sum = 0;
    for (int num : nums) {
        sum += num;
    }
    long end = System.nanoTime();
    System.out.println("单线程运算结果为：" + sum + "，计算时间为：" + (end  -
start));

    // ====================多线程分而治之累加10亿数据================================
    // 在使用forkJoinPool时，不推荐使用Runnable和Callable
    // 可以使用提供的另外两种任务的描述方式
    // Runnable(没有返回结果) ->   RecursiveAction
    // Callable(有返回结果)   ->   RecursiveTask
    ForkJoinPool forkJoinPool = (ForkJoinPool) Executors.newWorkStealingPool();
    System.out.println("分而治之计算数组总和！");
    long forkJoinStart = System.nanoTime();
    ForkJoinTask<Integer> task = forkJoinPool.submit(new SumRecursiveTask(0,
nums.length - 1));
    Integer result = task.join();
    long forkJoinEnd = System.nanoTime();
```

```
        System.out.println("分而治之运算结果为：" + result + "，计算时间为：" +
(forkJoinEnd  - forkJoinStart));
    }

    private static class SumRecursiveTask extends RecursiveTask<Integer>{
        /** 指定一个线程处理哪个位置的数据 */
        private int start,end;
        private final int MAX_STRIDE = 100_000_000;
        //  200_000_000：  147964900
        //  100_000_000：  145942100

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

        @Override
        protected Integer compute() {
            // 在这个方法中，需要设置好任务拆分的逻辑以及聚合的逻辑
            int sum = 0;
            int stride = end - start;
            if(stride <= MAX_STRIDE){
                // 可以处理任务
                for (int i = start; i <= end; i++) {
                    sum += nums[i];
                }
            }else{
                // 将任务拆分，分而治之。
                int middle = (start + end) / 2;
                // 声明为2个任务
                SumRecursiveTask left = new SumRecursiveTask(start, middle);
                SumRecursiveTask right = new SumRecursiveTask(middle + 1, end);
                // 分别执行两个任务
                left.fork();
                right.fork();
                // 等待结果，并且获取sum
                sum = left.join() + right.join();
            }
            return sum;
        }
    }
```

最终可以发现，这种累加的操作中，采用分而治之的方式效率提升了2倍多。

但是也不是所有任务都能拆分提升效率，首先任务得大，耗时要长。

三、ThreadPoolExecutor应用&源码剖析

前面讲到的Executors中的构建线程池的方式，大多数还是基于ThreadPoolExecutor去new出来的。

3.1 为什么要自定义线程池

首先ThreadPoolExecutor中，一共提供了7个参数，每个参数都是非常核心的属性，在线程池去执行任务时，每个参数都有决定性的作用。

但是如果直接采用JDK提供的方式去构建，可以设置的核心参数最多就两个，这样就会导致对线程池的控制粒度很粗。所以在阿里规范中也推荐自己去自定义线程池。手动的去new ThreadPoolExecutor设置他的一些核心属性。

自定义构建线程池，可以细粒度的控制线程池，去管理内存的属性，并且针对一些参数的设置可能更好的在后期排查问题。

查看一下ThreadPoolExecutor提供的七个核心参数

```
public ThreadPoolExecutor(
    int corePoolSize,           // 核心工作线程（当前任务执行结束后，不会被销毁）
    int maximumPoolSize,        // 最大工作线程（代表当前线程池中，一共可以有多少个工作线程）
    long keepAliveTime,         // 非核心工作线程在阻塞队列位置等待的时间
    TimeUnit unit,              // 非核心工作线程在阻塞队列位置等待时间的单位
    BlockingQueue<Runnable> workQueue,   // 任务在没有核心工作线程处理时，任务先扔到阻塞队列中
    ThreadFactory threadFactory,         // 构建线程的线程工作，可以设置thread的一些信息
    RejectedExecutionHandler handler) {  // 当线程池无法处理投递过来的任务时，执行当前的拒绝策略
    // 初始化线程池的操作
}
```

3.2 ThreadPoolExecutor应用

手动new一下，处理的方式还是执行execute或者submit方法。

JDK提供的几种拒绝策略：

- AbortPolicy：当前拒绝策略会在无法处理任务时，直接抛出一个异常

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    throw new RejectedExecutionException("Task " + r.toString() +
                                         " rejected from " +
                                         e.toString());
}
```

- CallerRunsPolicy：当前拒绝策略会在线程池无法处理任务时，将任务交给调用者处理

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    if (!e.isShutdown()) {
        r.run();
    }
}
```

- DiscardPolicy：当前拒绝策略会在线程池无法处理任务时，直接将任务丢弃掉

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
}
```

- DiscardOldestPolicy：当前拒绝策略会在线程池无法处理任务时，将队列中最早的任务丢弃掉，将当前任务再次尝试交给线程池处理

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
    if (!e.isShutdown()) {
        e.getQueue().poll();
        e.execute(r);
    }
}
```

- 自定义Policy：根据自己的业务，可以将任务扔到数据库，也可以做其他操作。

```
private static class MyRejectedExecution implements
RejectedExecutionHandler{
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        System.out.println("根据自己的业务情况，决定编写的代码！");
    }
}
```

代码构建线程池，并处理有无返回结果的任务

```
public static void main(String[] args) throws ExecutionException,
InterruptedException {
    //1. 构建线程池
    ThreadPoolExecutor threadPool = new ThreadPoolExecutor(
            2,
            5,
            10,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(5),
            new ThreadFactory() {
                @Override
                public Thread newThread(Runnable r) {
                    Thread thread = new Thread(r);
                    thread.setName("test-ThreadPoolExecutor");
                    return thread;
                }
            },
            new MyRejectedExecution()
    );

    //2. 让线程池处理任务,没返回结果
    threadPool.execute(() -> {
        System.out.println("没有返回结果的任务");
    });

    //3. 让线程池处理有返回结果的任务
    Future<Object> future = threadPool.submit(new Callable<Object>() {
        @Override
        public Object call() throws Exception {
            System.out.println("我有返回结果！");
            return "返回结果";
        }
    });
    Object result = future.get();
    System.out.println(result);

    //4. 如果是局部变量的线程池，记得用完要shutdown
```

```
    threadPool.shutdown();
}



private static class MyRejectedExecution implements RejectedExecutionHandler{
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        System.out.println("根据自己的业务情况，决定编写的代码！");
    }
}
```

3.3 ThreadPoolExecutor源码剖析

线程池的源码内容会比较多一点，需要一点一点的去查看，内部比较多。

3.3.1 ThreadPoolExecutor的核心属性

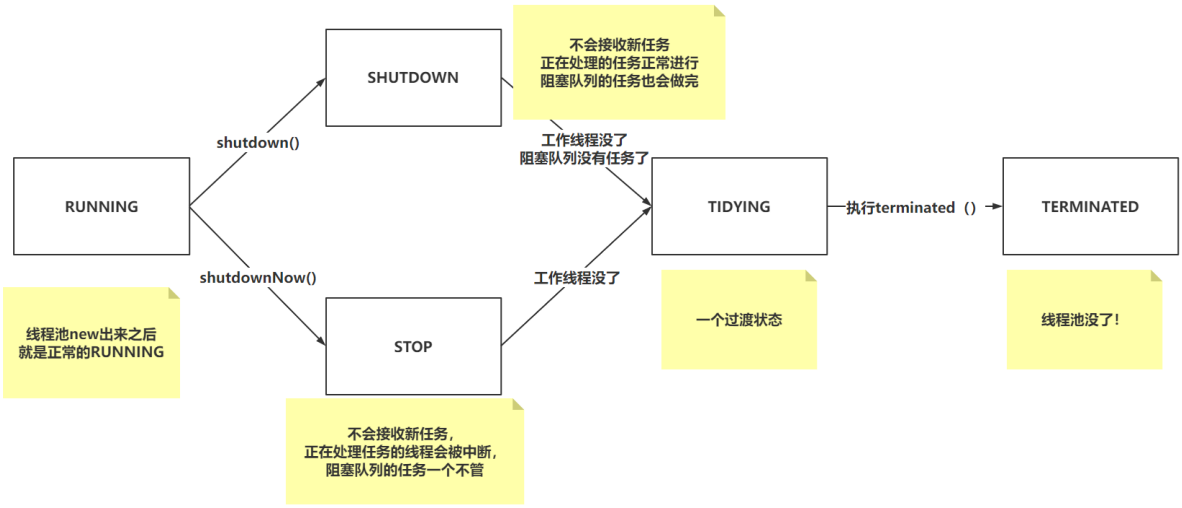
核心属性主要就是ctl，基于ctl拿到线程池的状态以及工作线程个数

在整个线程池的执行流程中，会基于ctl判断上述两个内容

```
// 当前是线程池的核心属性
// 当前的ctl其实就是一个int类型的数值，内部是基于AtomicInteger套了一层，进行运算时，是原子性的。
// ctl表示着线程池中的2个核心状态：
// 线程池的状态：ctl的高3位，表示线程池状态
// 工作线程的数量：ctl的低29位，表示工作线程的个数
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

// Integer.SIZE：在获取Integer的bit位个数
// 声明了一个常量：COUNT_BITS = 29
private static final int COUNT_BITS = Integer.SIZE - 3;
00000000 00000000 00000000 00000001
00100000 00000000 00000000 00000000
00011111 11111111 11111111 11111111
// CAPACITY就是当前工作线程能记录的工作线程的最大个数
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;


// 线程池状态的表示
// 当前五个状态中，只有RUNNING状态代表线程池没问题，可以正常接收任务处理
// 111：代表RUNNING状态，RUNNING可以处理任务，并且处理阻塞队列中的任务。
private static final int RUNNING    = -1 << COUNT_BITS;
// 000：代表SHUTDOWN状态，不会接收新任务，正在处理的任务正常进行，阻塞队列的任务也会做完。
private static final int SHUTDOWN   =  0 << COUNT_BITS;
// 001：代表STOP状态，不会接收新任务，正在处理任务的线程会被中断，阻塞队列的任务一个不管。
private static final int STOP       =  1 << COUNT_BITS;
// 010：代表TIDYING状态，这个状态是否SHUTDOWN或者STOP转换过来的，代表当前线程池马上关闭，就是过渡状态。
private static final int TIDYING    =  2 << COUNT_BITS;
// 011：代表TERMINATED状态，这个状态是TIDYING状态转换过来的，转换过来只需要执行一个terminated方法。
private static final int TERMINATED =  3 << COUNT_BITS;

// 在使用下面这几个方法时，需要传递ctl进来
```

```
// 基于&运算的特点，保证只会拿到ctl高三位的值。
private static int runStateOf(int c)     { return c & ~CAPACITY; }
// 基于&运算的特点，保证只会拿到ctl低29位的值。
private static int workerCountOf(int c)  { return c & CAPACITY; }
```

线程池状态的特点以及转换的方式



3.3.2 ThreadPoolExecutor的有参构造

有参构造没啥说的，记住核心线程个数是允许为0的。

```
// 有参构造。无论调用哪个有参构造，最终都会执行当前的有参构造
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    // 健壮性校验
    // 核心线程个数是允许为0个的。
    // 最大线程数必须大于0，最大线程数要大于等于核心线程数
    // 非核心线程的最大空闲时间，可以等于0
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        // 不满足要求就抛出参数异常
        throw new IllegalArgumentException();
    // 阻塞队列，线程工厂，拒绝策略都不允许为null，为null就扔空指针异常
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    // 不要关注当前内容，系统资源访问决策，和线程池核心业务关系不大。
    this.acc = System.getSecurityManager() == null ? null :
AccessController.getContext();
    // 各种赋值，JUC包下，几乎所有涉及到线程挂起的操作，单位都用纳秒。
    // 有参构造的值，都赋值给成员变量。
    // Doug Lea的习惯就是将成员变量作为局部变量单独操作。
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
```

```
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

3.3.3 ThreadPoolExecutor的execute方法

execute方法是提交任务到线程池的核心方法，很重要

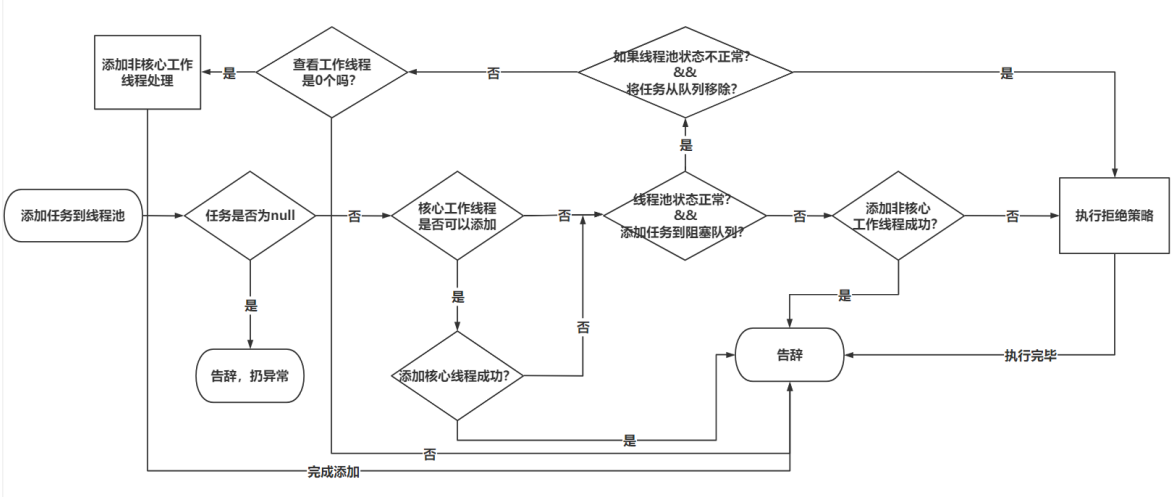
线程池的执行流程其实就是在说execute方法内部做了哪些判断

execute源码的分析

```
// 提交任务到线程池的核心方法
// command就是提交过来的任务
public void execute(Runnable command) {
    // 提交的任务不能为null
    if (command == null)
        throw new NullPointerException();
    // 获取核心属性ctl，用于后面的判断
    int c = ctl.get();
    // 如果工作线程个数，小于核心线程数。
    // 满足要求，添加核心工作线程
    if (workerCountOf(c) < corePoolSize) {
        // addWorker(任务,是核心线程吗)
        // addWorker返回true：代表添加工作线程成功
        // addWorker返回false：代表添加工作线程失败
        // addWorker中会基于线程池状态，以及工作线程个数做判断，查看能否添加工作线程
        if (addWorker(command, true))
            // 工作线程构建出来了，任务也交给command去处理了。
            return;
        // 说明线程池状态或者是工作线程个数发生了变化，导致添加失败，重新获取一次ctl
        c = ctl.get();
    }
    // 添加核心工作线程失败，往这走
    // 判断线程池状态是否是RUNNING，如果是，正常基于阻塞队列的offer方法，将任务添加到阻塞队列
    if (isRunning(c) && workQueue.offer(command)) {
        // 如果任务添加到阻塞队列成功，走if内部
        // 如果任务在扔到阻塞队列之前，线程池状态突然改变了。
        // 重新获取ctl
        int recheck = ctl.get();
        // 如果线程池的状态不是RUNNING，将任务从阻塞队列移除，
        if (!isRunning(recheck) && remove(command))
            // 并且直接拒绝策略
            reject(command);
        // 在这，说明阻塞队列有我刚刚放进去的任务
        // 查看一下工作线程数是不是0个
        // 如果工作线程为0个，需要添加一个非核心工作线程去处理阻塞队列中的任务
        // 发生这种情况有两种：
        // 1. 构建线程池时，核心线程数是0个。
        // 2. 即便有核心线程，可以设置核心线程也允许超时，设置allowCoreThreadTimeOut为
true，代表核心线程也可以超时
        else if (workerCountOf(recheck) == 0)
            // 为了避免阻塞队列中的任务饥饿，添加一个非核心工作线程去处理
            addWorker(null, false);
    }
    // 任务添加到阻塞队列失败
    // 构建一个非核心工作线程
```

```
    // 如果添加非核心工作线程成功，直接完事，告辞
    else if (!addWorker(command, false))
        // 添加失败，执行决绝策略
        reject(command);
}
```

execute方法的完整执行流程图



3.3.4 ThreadPoolExecutor的addWorker方法

addWorker中主要分成两大块去看

- 第一块：校验线程池的状态以及工作线程个数
- 第二块：添加工作线程并且启动工作线程

校验线程池的状态以及工作线程个数

```
// 添加工作线程之校验源码
private boolean addWorker(Runnable firstTask, boolean core) {
    // 外层for循环在校验线程池的状态
    // 内层for循环是在校验工作线程的个数

    // retry是给外层for循环添加一个标记，是为了方便在内层for循环跳出外层for循环
    retry:
    for (;;) {
        // 获取ctl
        int c = ctl.get();
        // 拿到ctl的高3位的值
        int rs = runStateOf(c);
//==========================线程池状态判断
==================================================
        // 如果线程池状态是SHUTDOWN，并且此时阻塞队列有任务，工作线程个数为0，添加一个工作线
程去处理阻塞队列的任务

        // 判断线程池的状态是否大于等于SHUTDOWN，如果满足，说明线程池不是RUNNING
        if (rs >= SHUTDOWN &&
            // 如果这三个条件都满足，就代表是要添加非核心工作线程去处理阻塞队列任务
            // 如果三个条件有一个没满足，返回false，配合!，就代表不需要添加
            !(rs == SHUTDOWN && firstTask == null && ! workQueue.isEmpty()))
            // 不需要添加工作线程
            return false;

        for (;;) {
```

```
//==========================工作线程个数判断
==================================================
            // 基于ctl拿到低29位的值，代表当前工作线程个数
            int wc = workerCountOf(c);
            // 如果工作线程个数大于最大值了，不可以添加了，返回false
            if (wc >= CAPACITY ||
                // 基于core来判断添加的是否是核心工作线程
                // 如果是核心：基于corePoolSize去判断
                // 如果是非核心：基于maximumPoolSize去判断
                wc >= (core ? corePoolSize : maximumPoolSize))
                // 代表不能添加，工作线程个数不满足要求
                return false;
            // 针对ctl进行 + 1，采用CAS的方式
            if (compareAndIncrementWorkerCount(c))
                // CAS成功后，直接退出外层循环，代表可以执行添加工作线程操作了。
                break retry;
            // 重新获取一次ctl的值
            c = ctl.get();
            // 判断重新获取到的ctl中，表示的线程池状态跟之前的是否有区别
            // 如果状态不一样，说明有变化，重新的去判断线程池状态
            if (runStateOf(c) != rs)
                // 跳出一次外层for循环
                continue retry;
        }
    }
    // 省略添加工作线程以及启动的过程
}
```

添加工作线程并且启动工作线程

```
private boolean addWorker(Runnable firstTask, boolean core) {
    // 省略校验部分的代码

    // 添加工作线程以及启动工作线程~~~
    // 声明了三个变量
    // 工作线程启动了没，默认false
    boolean workerStarted = false;
    // 工作线程添加了没，默认false
    boolean workerAdded = false;
    // 工作线程，默认为null
    Worker w = null;

    try {
        // 构建工作线程，并且将任务传递进去
        w = new Worker(firstTask);
        // 获取了Worker中的Thread对象
        final Thread t = w.thread;
        // 判断Thread是否不为null，在new Worker时，内部会通过给予的ThreadFactory去构建
Thread交给Worker
        // 一般如果为null，代表ThreadFactory有问题。
        if (t != null) {
            // 加锁，保证使用workers成员变量以及对largestPoolSize赋值时，保证线程安全
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // 再次获取线程池状态。
                int rs = runStateOf(ctl.get());
```

```
                // 再次判断
                // 如果满足  rs < SHUTDOWN  说明线程池是RUNNING，状态正常，执行if代码块
                // 如果线程池状态为SHUTDOWN，并且firstTask为null，添加非核心工作处理阻塞
队列任务
                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    // 到这，可以添加工作线程。
                    // 校验ThreadFactory构建线程后，不能自己启动线程，如果启动了，抛出异常
                    if (t.isAlive())
                        throw new IllegalThreadStateException();
                    // private final HashSet<Worker> workers = new
HashSet<Worker>();
                    // 将new好的Worker添加到HashSet中。
                    workers.add(w);
                    // 获取了HashSet的size，拿到工作线程个数
                    int s = workers.size();
                    // largestPoolSize在记录最大线程个数的记录
                    // 如果当前工作线程个数，大于最大线程个数的记录，就赋值
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    // 添加工作线程成功
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            // 如果工作线程添加成功，
            if (workerAdded) {
                // 直接启动Worker中的线程
                t.start();
                // 启动工作线程成功
                workerStarted = true;
            }
        }
    } finally {
        // 做补偿的操作，如果工作线程启动失败，将这个添加失败的工作线程处理掉
        if (!workerStarted)
            addWorkerFailed(w);
    }
    // 返回工作线程是否启动成功
    return workerStarted;
}
// 工作线程启动失败，需要不的步长操作
private void addWorkerFailed(Worker w) {
    // 因为操作了workers，需要加锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 如果w不为null，之前Worker已经new出来了。
        if (w != null)
            // 从HashSet中移除
            workers.remove(w);
        // 同时对ctl进行 - 1，代表去掉了一个工作线程个数
        decrementWorkerCount();
        // 因为工作线程启动失败，判断一下状态的问题，是不是可以走TIDYING状态最终到
TERMINATED状态了。
        tryTerminate();
    } finally {
```

```
            // 释放锁
            mainLock.unlock();
        }
    }
```

3.3.5 ThreadPoolExecutor的Worker工作线程

Worker对象主要包含了两个内容

- 工作线程要执行任务
- 工作线程可能会被中断，控制中断

```
// Worker继承了AQS，目的就是为了控制工作线程的中断。
// Worker实现了Runnable，内部的Thread对象，在执行start时，必然要执行Worker中断额一些操作
private final class Worker extends AbstractQueuedSynchronizer implements
Runnable{

// ======================Worker管理任务=================================
    // 线程工厂构建的线程
    final Thread thread;

    // 当前Worker要执行的任务
    Runnable firstTask;

    // 记录当前工作线程处理了多少个任务。
    volatile long completedTasks;

    // 有参构造
    Worker(Runnable firstTask) {
        // 将State设置为-1，代表当前不允许中断线程
        setState(-1);
        // 任务赋值
        this.firstTask = firstTask;
        // 基于线程工作构建Thread，并且传入的Runnable是Worker
        this.thread = getThreadFactory().newThread(this);
    }

    // 当thread执行start方法时，调用的是Worker的run方法，
    public void run() {
        // 任务执行时，执行的是runWorker方法
        runWorker(this);
    }


// ======================Worker管理中断=================================
    // 当前方法是中断工作线程时，执行的方法
    void interruptIfStarted() {
        Thread t;
        // 只有Worker中的state >= 0的时候，可以中断工作线程
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                // 如果状态正常，并且线程未中断，这边就中断线程
                t.interrupt();
            } catch (SecurityException ignore) {
            }
        }
    }
```

```
    protected boolean isHeldExclusively() {
        return getState() != 0;
    }
    protected boolean tryAcquire(int unused) {
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }
    protected boolean tryRelease(int unused) {
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }
    public void lock()        { acquire(1); }
    public boolean tryLock()  { return tryAcquire(1); }
    public void unlock()      { release(1); }
    public boolean isLocked() { return isHeldExclusively(); }


}
```

3.3.6 ThreadPoolExecutor的runWorker方法

runWorker就是让工作线程拿到任务去执行即可。

并且在内部也处理了在工作线程正常结束和异常结束时的处理方案

```
// 工作线程启动后执行的任务。
final void runWorker(Worker w) {
    // 拿到当前线程
    Thread wt = Thread.currentThread();
    // 从worker对象中拿到任务
    Runnable task = w.firstTask;
    // 将Worker中的firstTask置位空
    w.firstTask = null;
    // 将Worker中的state置位0，代表当前线程可以中断的
    w.unlock(); // allow interrupts
    // 判断工作线程是否是异常结束，默认就是异常结束
    boolean completedAbruptly = true;
    try {
        // 获取任务
        // 直接拿到第一个任务去执行
        // 如果第一个任务为null，去阻塞队列中获取任务
        while (task != null || (task = getTask()) != null) {
            // 执行了Worker的lock方法，当前在lock时，shutdown操作不能中断当前线程，因为当
前线程正在处理任务
            w.lock();
            // 比较ctl >= STOP,如果满足找个状态，说明线程池已经到了STOP状态甚至已经要凉凉了
            // 线程池到STOP状态，并且当前线程还没有中断，确保线程是中断的，进到if内部执行中断
方法
            // if(runStateAtLeast(ctl.get(), STOP) && !wt.isInterrupted()) {中断线
程}
            // 如果线程池状态不是STOP，确保线程不是中断的。
            // 如果发现线程中断标记位是true了，再次查看线程池状态是大于STOP了，再次中断线程
```

```
            // 这里其实就是做了一个事情，如果线程池状态 >= STOP，确保线程中断了。
            if (
                (
                    runStateAtLeast(ctl.get(), STOP) ||
                    (     Thread.interrupted() && runStateAtLeast(ctl.get(),
STOP)   )
                )
                && !wt.isInterrupted())
                wt.interrupt();
            try {
                // 勾子函数在线程池中没有做任何的实现，如果需要在线程池执行任务前后做一些额外
的处理，可以重写勾子函数
                // 前置勾子函数
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    // 执行任务。
                    task.run();
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    // 前后置勾子函数
                    afterExecute(task, thrown);
                }
            } finally {
                // 任务执行完，丢掉任务
                task = null;
                // 当前工作线程处理的任务数+1
                w.completedTasks++;
                // 执行unlock方法，此时shutdown方法才可以中断当前线程
                w.unlock();
            }
        }
        // 如果while循环结束，正常走到这，说明是正常结束
        // 正常结束的话，在getTask中就会做一个额外的处理，将ctl - 1，代表工作线程没一个。
        completedAbruptly = false;
    } finally {
        // 考虑干掉工作线程
        processWorkerExit(w, completedAbruptly);
    }
}
// 工作线程结束前，要执行当前方法
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    // 如果是异常结束
    if (completedAbruptly)
        // 将ctl - 1，扣掉一个工作线程
        decrementWorkerCount();

    // 操作Worker，为了线程安全，加锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 当前工作线程处理的任务个数累加到线程池处理任务的个数属性中
        completedTaskCount += w.completedTasks;
```

```
            // 将工作线程从hashSet中移除
            workers.remove(w);
        } finally {
            // 释放锁
            mainLock.unlock();
        }

        // 只要工作线程凉了，查看是不是线程池状态改变了。
        tryTerminate();

        // 获取ctl
        int c = ctl.get();
        // 判断线程池状态，当前线程池要么是RUNNING，要么是SHUTDOWN
        if (runStateLessThan(c, STOP)) {
            // 如果正常结束工作线程
            if (!completedAbruptly) {
                // 如果核心线程允许超时，min = 0，否则就是核心线程个数
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                // 如果min == 0，可能会出现没有工作线程，并且阻塞队列有任务没有线程处理
                if (min == 0 && ! workQueue.isEmpty())
                    // 至少要有一个工作线程处理阻塞队列任务
                    min = 1;
                // 如果工作线程个数 大于等于1，不怕没线程处理，正常return
                if (workerCountOf(c) >= min)
                    return;
            }
            // 异常结束，为了避免出现问题，添加一个空任务的非核心线程来填补上刚刚异常结束的工作线程
            addWorker(null, false);
        }
    }
```

3.3.7 ThreadPoolExecutor的getTask方法

工作线程在去阻塞队列获取任务前，要先查看线程池状态

如果状态没问题，去阻塞队列take或者是poll任务

第二个循环时，不但要判断线程池状态，还要判断当前工作线程是否可以被干掉

```
// 当前方法就在阻塞队列中获取任务
// 前面半部分是判断当前工作线程是否可以返回null，结束。
// 后半部分就是从阻塞队列中拿任务
private Runnable getTask() {
    // timeOut默认值是false。
    boolean timedOut = false;

    // 死循环
    for (;;) {
        // 拿到ctl
        int c = ctl.get();
        // 拿到线程池的状态
        int rs = runStateOf(c);

        // 如果线程池状态是STOP，没有必要处理阻塞队列任务，直接返回null
        // 如果线程池状态是SHUTDOWN，并且阻塞队列是空的，直接返回null
        if (rs >= SHUTDOWN &&
                (rs >= STOP || workQueue.isEmpty())) {
            // 如果可以返回null，先扣减工作线程个数
```

```
                decrementWorkerCount();
                // 返回null，结束runWorker的while循环
                return null;
            }

            // 基于ctl拿到工作线程个数
            int wc = workerCountOf(c);

            // 核心线程允许超时，timed为true
            // 工作线程个数大于核心线程数，timed为true
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

            if (
                // 如果工作线程个数，大于最大线程数。（一般情况不会满足），把他看成false
                // 第二个判断代表，只要工作线程数小于等于核心线程数，必然为false
                // 即便工作线程个数大于核心线程数了，此时第一次循环也不会为true，因为timedOut默认值是false
                // 考虑第二次循环了，因为循环内部必然有修改timeOut的位置
                (wc > maximumPoolSize || (timed && timedOut))
                &&
                // 要么工作线程还有，要么阻塞队列为空，并且满足上述条件后，工作线程才会走到if内部，结束工作线程
                (wc > 1 || workQueue.isEmpty())
               ) {
                // 第二次循环才有可能到这。
                // 正常结束，工作线程 - 1，因为是CAS操作，如果失败了，重新走for循环
                if (compareAndDecrementWorkerCount(c))
                    return null;
                continue;
            }

            // 工作线程从阻塞队列拿任务
            try {
                // 如果是核心线程，timed是false，如果是非核心线程，timed就是true
                Runnable r = timed ?
                    // 如果是非核心，走poll方法，拿任务，等待一会
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    // 如果是核心，走take方法，死等。
                    workQueue.take();
                // 从阻塞队列拿到的任务不为null，这边就正常返回任务，去执行
                if (r != null)
                    return r;
                // 说明当前线程没拿到任务，将timeOut设置为true，在上面就可以返回null退出了。
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
    }
```

3.3.8 ThreadPoolExecutor的关闭方法

首先查看shutdownNow方法，可以从RUNNING状态转变为STOP

```
// shutDownNow方法，shutdownNow不会处理阻塞队列的任务，将任务全部给你返回了。
public List<Runnable> shutdownNow() {
    // 声明返回结果
```

```
    List<Runnable> tasks;
    // 加锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 不关注这个方法……
        checkShutdownAccess();
        // 将线程池状态修改为STOP
        advanceRunState(STOP);
        // 无论怎么，直接中断工作线程。
        interruptWorkers();
        // 将阻塞队列的任务全部扔到List集合中。
        tasks = drainQueue();
    } finally {
        // 释放锁
        mainLock.unlock();
    }
    tryTerminate();
    return tasks;
}

// 将线程池状态修改为STOP
private void advanceRunState(int STOP) {
    // 死循环。
    for (;;) {
        // 获取ctl属性的值
        int c = ctl.get();
        // 第一个判断：如果当前线程池状态已经大于等于STOP了，不管了，告辞。
        if (runStateAtLeast(c, STOP) ||
            // 基于CAS，将ctl从c修改为STOP状态，不修改工作线程个数，但是状态变为了STOP
            // 如果修改成功结束
            ctl.compareAndSet(c, ctlOf(STOP, workerCountOf(c))))
            break;
    }
}
// 无论怎么，直接中断工作线程。
private void interruptWorkers() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 遍历HashSet，拿到所有的工作线程，直接中断。
        for (Worker w : workers)
            w.interruptIfStarted();
    } finally {
        mainLock.unlock();
    }
}
// 移除阻塞队列，内容全部扔到List集合中
private List<Runnable> drainQueue() {
    BlockingQueue<Runnable> q = workQueue;
    ArrayList<Runnable> taskList = new ArrayList<Runnable>();
    // 阻塞队列自带的，直接清空阻塞队列，内容扔到List集合
    q.drainTo(taskList);
    // 为了避免任务丢失，重新判断，是否需要编辑阻塞队列，重新扔到List
    if (!q.isEmpty()) {
        for (Runnable r : q.toArray(new Runnable[0])) {
            if (q.remove(r))
                taskList.add(r);
```

```
        }
    }
    return taskList;
}

// 查看当前线程池是否可以变为TERMINATED状态
final void tryTerminate() {
    // 死循环。
    for (;;) {
        // 拿到ctl
        int c = ctl.get();
        // 如果是RUNNING，直接告辞。
        // 如果状态已经大于等于TIDYING，马上就要凉凉，直接告辞。
        // 如果状态是SHUTDOWN，但是阻塞队列还有任务，直接告辞。
        if (isRunning(c) ||
            runStateAtLeast(c, TIDYING) ||
            (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
            return;
        // 如果还有工作线程
        if (workerCountOf(c) != 0) {
            // 再次中断工作线程
            interruptIdleWorkers(ONLY_ONE);
            // 告辞，等你工作线程全完事，我这再尝试进入到TERMINATED状态
            return;
        }

        // 加锁，为了可以执行Condition的释放操作
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // 将线程池状态修改为TIDYING状态，如果成功，继续往下走
            if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                try {
                    // 这个方法是空的，如果你需要在线程池关闭后做一些额外操作，这里你可以自
行实现
                    terminated();
                } finally {
                    // 最终修改为TERMINATED状态
                    ctl.set(ctlOf(TERMINATED, 0));
                    // 线程池提供了一个方法，主线程在提交任务到线程池后，是可以继续做其他操
作的。
                    // 咱们也可以让主线程提交任务后，等待线程池处理完毕，再做后续操作
                    // 这里线程池凉凉后，要唤醒哪些调用了awaitTermination方法的线程
                    termination.signalAll();
                }
                return;
            }
        } finally {
            mainLock.unlock();
        }
        // else retry on failed CAS
    }
}
```

再次shutdown方法，可以从RUNNING状态转变为SHUTDOWN

shutdown状态下，不会中断正在干活的线程，而且会处理阻塞队列中的任务

```
public void shutdown() {
    // 加锁。。
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 不看。
        checkShutdownAccess();
        // 里面是一个死循环，将线程池状态修改为SHUTDOWN
        advanceRunState(SHUTDOWN);
        // 中断空闲线程
        interruptIdleWorkers();
        // 说了，这个是为了ScheduleThreadPoolExecutor准备的，不管
        onShutdown();
    } finally {
        mainLock.unlock();
    }
    // 尝试结束线程
    tryTerminate();
}

// 中断空闲线程
private void interruptIdleWorkers(boolean onlyOne) {
    // 加锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (Worker w : workers) {
            Thread t = w.thread;
            // 如果线程没有中断，那么就去获取Worker的锁，基于tryLock可知，不会中断正在干活
的线程
            if (!t.isInterrupted() && w.tryLock()) {
                try {
                    // 会中断空闲线程
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
    }
}
```

3.4 线程池的核心参数设计规则

线程池的使用难度不大，难度在于线程池的参数并不好配置。

主要难点在于任务类型无法控制，比如任务有CPU密集型，还有IO密集型，甚至还有混合型的。

因为IO咱们无法直接控制，所以很多时间按照一些书上提供的一些方法，是无法解决问题的。

《Java并发编程实践》

想调试出一个符合当前任务情况的核心参数，最好的方式就是测试。

需要将项目部署到测试环境或者是沙箱环境中，结果各种压测得到一个相对符合的参数。

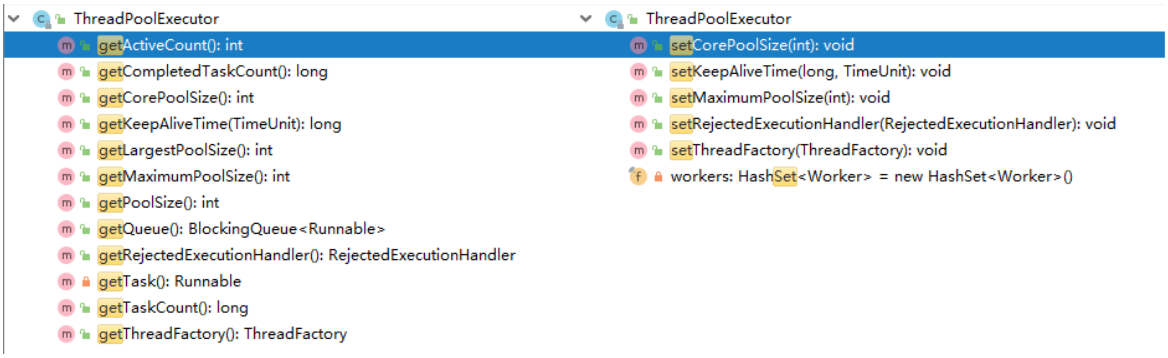
如果每次修改项目都需要重新部署，成本太高了。

此时咱们可以实现一个动态监控以及修改线程池的方案。

因为线程池的核心参数无非就是：

- corePoolSize：核心线程数
- maximumPoolSize：最大线程数
- workQueue：工作队列

线程池中提供了获取核心信息的get方法，同时也提供了动态修改核心属性的set方法。



也可以采用一些开源项目提供的方式去做监控和修改

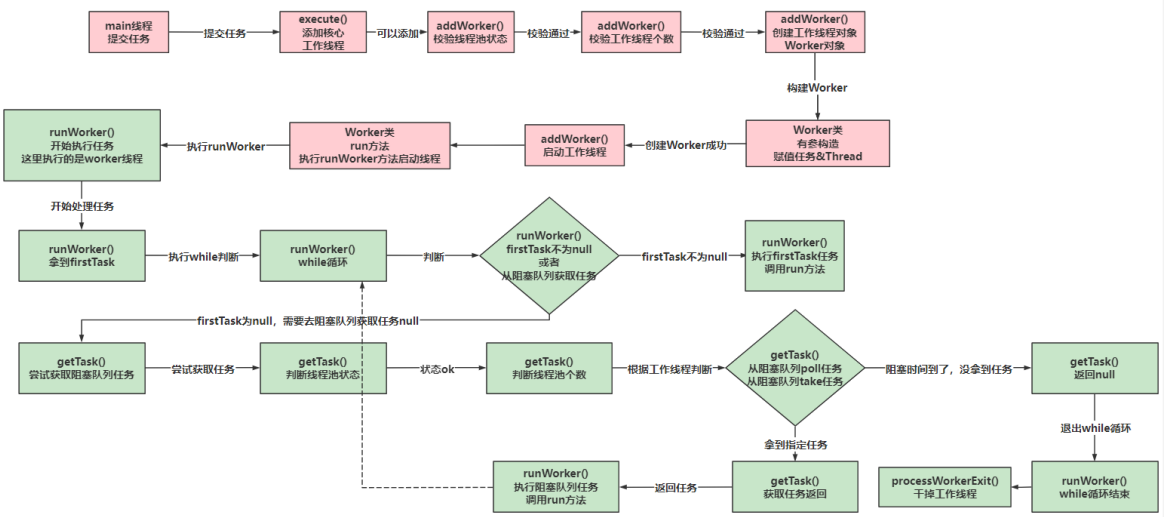
比如hippo4j就可以对线程池进行监控，而且可以和SpringBoot整合。

Github地址：https://github.com/opengoofy/hippo4j

官方文档：https://hippo4j.cn/docs/user_docs/intro

3.5 线程池处理任务的核心流程

基于addWorker添加工作线程的流程切入到整体处理任务的位置



四、ScheduleThreadPoolExecutor应用&源码

4.1 ScheduleThreadPoolExecutor介绍

从名字上就可以看出，当前线程池是用于执行定时任务的线程池。

Java比较早的定时任务工具是Timer类。但是Timer问题很多，串行的，不靠谱，会影响到其他的任务执行。

其实除了Timer以及ScheduleThreadPoolExecutor之外，正常在企业中一般会采用Quartz或者是SpringBoot提供的Schedule的方式去实现定时任务的功能。

ScheduleThreadPoolExecutor支持延迟执行以及周期性执行的功能。

4.2 ScheduleThreadPoolExecutor应用

定时任务线程池的有参构造

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

发现ScheduleThreadPoolExecutor在构建时，直接调用了父类的构造方法

ScheduleThreadPoolExecutor的父类就是ThreadPoolExecutor

首先ScheduleThreadPoolExecutor最多允许设置3个参数：

- 核心线程数
- 线程工厂
- 拒绝策略

首先没有设置阻塞队列，以及最大线程数和空闲时间以及单位

阻塞队列设置的是DelayedWorkQueue，其实本质就是DelayQueue，一个延迟队列。DelayQueue是一个无界队列。所以最大线程数以及非核心线程的空闲时间是不需要设置的。

代码落地使用

```
public static void main(String[] args) {
        //1. 构建定时任务线程池
        ScheduledThreadPoolExecutor pool = new ScheduledThreadPoolExecutor(
                5,
                new ThreadFactory() {
                    @Override
                    public Thread newThread(Runnable r) {
                        Thread t = new Thread(r);
                        return t;
                    }
                },
                new ThreadPoolExecutor.AbortPolicy()
        );

        //2. 应用ScheduledThreadPoolExecutor
        // 跟直接执行线程池的execute没啥区别
        pool.execute(() -> {
            System.out.println("execute");
        });

        // 指定延迟时间执行
        System.out.println(System.currentTimeMillis());
        pool.schedule(() -> {
            System.out.println("schedule");
            System.out.println(System.currentTimeMillis());
```

```
        },2, TimeUnit.SECONDS);

        // 指定第一次的延迟时间，并且确认后期的周期执行时间，周期时间是在任务开始时就计算
        // 周期性执行就是将执行完毕的任务再次社会好延迟时间，并且重新扔到阻塞队列
        // 计算的周期执行，也是在原有的时间上做累加，不关注任务的执行时长。
        System.out.println(System.currentTimeMillis());
        pool.scheduleAtFixedRate(() -> {
            System.out.println("scheduleAtFixedRate");
            System.out.println(System.currentTimeMillis());
        },2,3,TimeUnit.SECONDS);


    //         // 指定第一次的延迟时间，并且确认后期的周期执行时间，周期时间是在任务结束后再计
算下次的延迟时间
        System.out.println(System.currentTimeMillis());
        pool.scheduleWithFixedDelay(() -> {
            System.out.println("scheduleWithFixedDelay");
            System.out.println(System.currentTimeMillis());
            try {
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        },2,3,TimeUnit.SECONDS);
    }
```

4.3 ScheduleThreadPoolExecutor源码剖析

4.3.1 核心属性

后面的方法业务流程会涉及到这些属性。

```
// 这里是针对任务取消时的一些业务判断会用到的标记
private volatile boolean continueExistingPeriodicTasksAfterShutdown;
private volatile boolean executeExistingDelayedTasksAfterShutdown = true;
private volatile boolean removeOnCancel = false;

// 计数器，如果两个任务的执行时间节点一模一样，根据这个序列来判断谁先执行
private static final AtomicLong sequencer = new AtomicLong();

// 这个方法是获取当前系统时间的毫秒值
final long now() {
    return System.nanoTime();
}

// 内部类。核心类之一。
private class ScheduledFutureTask<V>
        extends FutureTask<V> implements RunnableScheduledFuture<V> {

    // 全局唯一的序列，如果两个任务时间一直，基于当前属性判断
    private final long sequenceNumber;

    // 任务执行的时间，单位纳秒
    private long time;

    /**
     *  period == 0：执行一次的延迟任务
```

```
     *  period > 0：代表是At
     *  period < 0：代表是With
     */
    private final long period;

    // 周期性执行时，需要将任务重新扔回阻塞队列，基础当前属性拿到任务，方便扔回阻塞队列
    RunnableScheduledFuture<V> outerTask = this;


    /**
     * 构建schedule方法的任务
     */
    ScheduledFutureTask(Runnable r, V result, long ns) {
        super(r, result);
        this.time = ns;
        this.period = 0;
        this.sequenceNumber = sequencer.getAndIncrement();
    }

    /**
     * 构建At和With任务的有参构造
     */
    ScheduledFutureTask(Runnable r, V result, long ns, long period) {
        super(r, result);
        this.time = ns;
        this.period = period;
        this.sequenceNumber = sequencer.getAndIncrement();
    }

}

// 内部类。核心类之一。
static class DelayedWorkQueue extends AbstractQueue<Runnable> implements
BlockingQueue<Runnable> {
// 这个类就是DelayQueue，不用过分关注，如果没看过，看阻塞队列中的优先级队列和延迟队列
```

4.3.2 schedule方法

execute方法也是调用的schedule方法，只不过传入的延迟时间是0纳秒

schedule方法就是将任务和延迟时间封装到一起，并且将任务扔到阻塞队列中，再去创建工作线程去take阻塞队列。

```
// 延迟任务执行的方法。
// command：任务
// delay：延迟时间
// unit：延迟时间的单位
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)
{
    // 健壮性校验。
    if (command == null || unit == null)
        throw new NullPointerException();

    // 将任务和延迟时间封装到一起，最终组成ScheduledFutureTask
    // 要分成三个方法去看
    // triggerTime：计算延迟时间。最终返回的是当前系统时间 + 延迟时间
    // triggerTime就是将延迟时间转换为纳秒，并且+当前系统时间，再做一些健壮性校验
```

```
    // ScheduledFutureTask有参构造：将任务以及延迟时间封装到一起，并且设置任务执行的方式

    // decorateTask：当前方式是让用户基于自身情况可以动态修改任务的一个扩展口
    RunnableScheduledFuture<?> t = decorateTask(command,
                                    new ScheduledFutureTask<Void>(command, null,
                                    triggerTime(delay, unit)));
    // 任务封装好，执行delayedExecute方法，去执行任务
    delayedExecute(t);

    // 返回FutureTask
    return t;
}

// triggerTime做的事情
// 外部方法，对延迟时间做校验，如果小于0，就直接设置为0
// 并且转换为纳秒单位
private long triggerTime(long delay, TimeUnit unit) {
    return triggerTime(unit.toNanos((delay < 0) ? 0 : delay));
}
// 将延迟时间+当前系统时间
// 后面的校验是为了避免延迟时间超过Long的取值范围
long triggerTime(long delay) {
    return now() + ((delay < (Long.MAX_VALUE >> 1)) ? delay :
overflowFree(delay));
}

// ScheduledFutureTask有参构造
ScheduledFutureTask(Runnable r, V result, long ns) {
    super(r, result);
    // time就是任务要执行的时间
    this.time = ns;
    // period,为0，代表任务是延迟执行，不是周期执行
    this.period = 0;
    // 基于AtmoicLong生成的序列
    this.sequenceNumber = sequencer.getAndIncrement();
}


// delayedExecute 执行延迟任务的操作
private void delayedExecute(RunnableScheduledFuture<?> task) {
    // 查看当前线程池是否还是RUNNING状态，如果不是RUNNING，进到if
    if (isShutdown())
        // 不是RUNNING。
        // 执行拒绝策略。
        reject(task);
    else {
        // 线程池状态是RUNNING
        // 直接让任务扔到延迟的阻塞队列中
        super.getQueue().add(task);
        // DCL的操作，再次查看线程池状态
        // 如果线程池在添加任务到阻塞队列后，状态不是RUNNING
        if (isShutdown() &&
            // task.isPeriodic()：现在反回的是false，因为任务是延迟执行，不是周期执行
            // 默认情况，延迟队列中的延迟任务，可以执行
            !canRunInCurrentRunState(task.isPeriodic()) &&
            // 从阻塞队列中移除任务。
            remove(task))
```

```
            task.cancel(false);
        else
            // 线程池状态正常，任务可以执行
            ensurePrestart();
    }
}

// 线程池状态不为RUNNING，查看任务是否可以执行
// 延迟执行：periodic==false
// 周期执行：periodic==true
// continueExistingPeriodicTasksAfterShutdown：周期执行任务，默认为false
// executeExistingDelayedTasksAfterShutdown：延迟执行任务，默认为true
boolean canRunInCurrentRunState(boolean periodic) {
    return isRunningOrShutdown(periodic ?
                               continueExistingPeriodicTasksAfterShutdown :
                               executeExistingDelayedTasksAfterShutdown);
}
// 当前情况，shutdownOK为true
final boolean isRunningOrShutdown(boolean shutdownOK) {
    int rs = runStateOf(ctl.get());
    // 如果状态是RUNNING，正常可以执行，返回true
    // 如果状态是SHUTDOWN，根据shutdownOK来决定
    return rs == RUNNING || (rs == SHUTDOWN && shutdownOK);
}


// 任务可以正常执行后，做的操作
void ensurePrestart() {
    // 拿到工作线程个数
    int wc = workerCountOf(ctl.get());
    // 如果工作线程个数小于核心线程数
    if (wc < corePoolSize)
        // 添加核心线程去处理阻塞队列中的任务
        addWorker(null, true);
    else if (wc == 0)
        // 如果工作线程数为0，核心线程数也为0，这是添加一个非核心线程去处理阻塞队列任务
        addWorker(null, false);
}
```

4.3.3 At和With方法&任务的run方法

这两个方法在源码层面上的第一个区别，就是在计算周期时间时，需要将这个值传递给period，基于正负数在区别At和With

所以查看一个方法就ok，查看At方法

```
// At方法，
// command：任务
// initialDelay：第一次执行的延迟时间
// period：任务的周期执行时间
// unit：上面两个时间的单位
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                              long initialDelay,
                                              long period,
                                              TimeUnit unit) {
    // 健壮性校验
    if (command == null || unit == null)
```

```
            throw new NullPointerException();
        // 周期时间不能小于等于0.
        if (period <= 0)
            throw new IllegalArgumentException();
        // 将任务以及第一次的延迟时间，和后续的周期时间封装好。
        ScheduledFutureTask<Void> sft =
            new ScheduledFutureTask<Void>(command,
                                          null,
                                          triggerTime(initialDelay, unit),
                                          unit.toNanos(period));
        // 扩展口，可以对任务做修改。
        RunnableScheduledFuture<Void> t = decorateTask(command, sft);

        // 周期性任务，需要在任务执行完毕后，重新扔会到阻塞队列，为了方便拿任务，将任务设置到
outerTask成员变量中
        sft.outerTask = t;
        // 和schedule方法一样的方式
        // 如果任务刚刚扔到阻塞队列，线程池状态变为SHUTDOWN，默认情况，当前任务不执行
        delayedExecute(t);
        return t;
    }

    // 延迟任务以及周期任务在执行时，都会调用当前任务的run方法。
    public void run() {
        // periodic == false：一次性延迟任务
        // periodic == true：周期任务
        boolean periodic = isPeriodic();
        // 任务执行前，会再次判断状态，能否执行任务
        if (!canRunInCurrentRunState(periodic))
            cancel(false);
        // 判断是周期执行还是一次性任务
        else if (!periodic)
            // 一次性任务，让工作线程直接执行command的逻辑
            ScheduledFutureTask.super.run();
        // 到这个else if，说明任务是周期执行
        else if (ScheduledFutureTask.super.runAndReset()) {
            // 设置下次任务执行的时间
            setNextRunTime();
            // 将任务重新扔回线程池做处理
            reExecutePeriodic(outerTask);
        }
    }
    // 设置下次任务执行的时间
    private void setNextRunTime() {
        // 拿到period值，正数：At，负数：With
        long p = period;
        if (p > 0)
            // 拿着之前的执行时间，直接追加上周期时间
            time += p;
        else
            // 如果走到else，代表任务是With方式，这种方式要重新计算延迟时间
            // 拿到当前系统时间，追加上延迟时间，
            time = triggerTime(-p);
    }
     // 将任务重新扔回线程池做处理
    void reExecutePeriodic(RunnableScheduledFuture<?> task) {
        // 如果状态ok，可以执行
        if (canRunInCurrentRunState(true)) {
```

```
            // 将任务扔到延迟队列
            super.getQueue().add(task);
            // DCL，判断线程池状态
            if (!canRunInCurrentRunState(true) && remove(task))
                task.cancel(false);
            else
                // 添加工作线程
                ensurePrestart();
        }
    }
```

并发集合

一、ConcurrentHashMap

1.1 存储结构

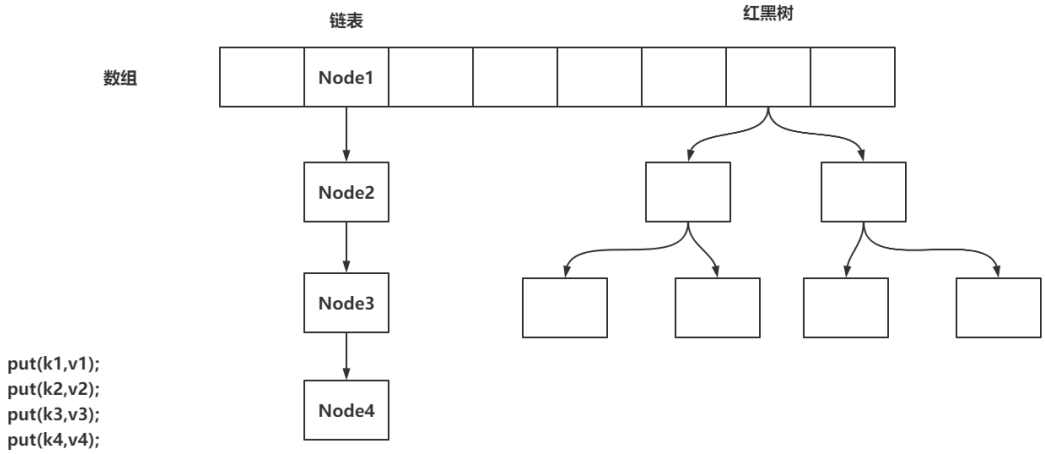
ConcurrentHashMap是线程安全的HashMap

ConcurrentHashMap在JDK1.8中是以CAS+synchronized实现的线程安全

CAS：在没有hash冲突时（Node要放在数组上时）

synchronized：在出现hash冲突时（Node存放的位置已经有数据了）

存储的结构：数组+链表+红黑树



1.2 存储操作

1.2.1 put方法

```
public V put(K key, V value) {
    // 在调用put方法时，会调用putVal，第三个参数默认传递为false
    // 在调用putIfAbsent时，会调用putVal方法，第三个参数传递的为true
    // 如果传递为false，代表key一致时，直接覆盖数据
    // 如果传递为true，代表key一致时，什么都不做，key不存在，正常添加（Redis，setnx）
    return putVal(key, value, false);
}
```

1.2.2 putVal方法-散列算法

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    // ConcurrentHashMap不允许key或者value出现为null的值，跟HashMap的区别
    if (key == null || value == null) throw new NullPointerException();
    // 根据key的hashCode计算出一个hash值，后期得出当前key-value要存储在哪个数组索引位置
    int hash = spread(key.hashCode());
    // 一个标识，在后面有用！
    int binCount = 0;
    // 省略大量的代码.....
}

// 计算当前Node的hash值的方法
static final int spread(int h) {
    // 将key的hashCode值的高低16位进行^运算，最终又与HASH_BITS进行了&运算
    // 将高位的hash也参与到计算索引位置的运算当中
    // 为什么HashMap、ConcurrentHashMap，都要求数组长度为2^n
    // HASH_BITS让hash值的最高位符号位肯定为0，代表当前hash值默认情况下一定是正数，因为hash
值为负数时，有特殊的含义
    // static final int MOVED     = -1; // 代表当前hash位置的数据正在扩容！
    // static final int TREEBIN   = -2; // 代表当前hash位置下挂载的是一个红黑树
    // static final int RESERVED  = -3; // 预留当前索引位置.....
    return (h ^ (h >>> 16)) & HASH_BITS;
    // 计算数组放到哪个索引位置的方法    (f = tabAt(tab, i = (n - 1) & hash)
    // n：是数组的长度
}
00001101 00001101 00101111 10001111  - h = key.hashCode

运算方式
00000000 00000000 00000000 00001111  - 15 (n - 1)
&
(
(
00001101 00001101 00101111 10001111  - h
^
00000000 00000000 00001101 00001101  - h >>> 16
)
&
01111111 11111111 11111111 11111111  - HASH_BITS
)
```

1.2.3 putVal方法-添加数据到数组&初始化数组

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    // 省略部分代码.....
    // 将Map的数组赋值给tab，死循环
    for (Node<K,V>[] tab = table;;) {
        // 声明了一堆变量~~
        // n:数组长度
        // i:当前Node需要存放的索引位置
        // f: 当前数组i索引位置的Node对象
        // fn：当前数组i索引位置上数据的hash值
        Node<K,V> f; int n, i, fh;
        // 判断当前数组是否还没有初始化
        if (tab == null || (n = tab.length) == 0)
            // 将数组进行初始化。
```

```
            tab = initTable();
        // 基于 (n - 1) & hash 计算出当前Node需要存放在哪个索引位置
        // 基于tabAt获取到i位置的数据
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            // 现在数组的i位置上没有数据，基于CAS的方式将数据存在i位置上
            if (casTabAt(tab, i, null,new Node<K,V>(hash, key, value, null)))
                // 如果成功，执行break跳出循环，插入数据成功
                break;
        }
        // 判断当前位置数据是否正在扩容.....
        else if ((fh = f.hash) == MOVED)
            // 让当前插入数据的线程协助扩容
            tab = helpTransfer(tab, f);
        // 省略部分代码.....
    }
    // 省略部分代码.....
}
```

sizeCtl：是数组在初始化和扩容操作时的一个控制变量
-1：代表当前数组正在初始化
小于-1：低16位代表当前数组正在扩容的线程个数（如果1个线程扩容，值为-2，如果2个线程扩容，值为-3）
0：代表数组还没初始化
大于0：代表当前数组的扩容阈值，或者是当前数组的初始化大小

```
// 初始化数组方法
private final Node<K,V>[] initTable() {
    // 声明标识
    Node<K,V>[] tab; int sc;
    // 再次判断数组没有初始化，并且完成tab的赋值
    while ((tab = table) == null || tab.length == 0) {
        // 将sizeCtl赋值给sc变量，并判断是否小于0
        if ((sc = sizeCtl) < 0)
            Thread.yield();
        // 可以尝试初始化数组，线程会以CAS的方式，将sizeCtl修改为-1，代表当前线程可以初始化数组
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            // 尝试初始化!
            try {
                // 再次判断当前数组是否已经初始化完毕。
                if ((tab = table) == null || tab.length == 0) {
                    // 开始初始化，
                    // 如果sizeCtl > 0，就初始化sizeCtl长度的数组
                    // 如果sizeCtl == 0，就初始化默认的长度
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    // 初始化数组!
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    // 将初始化的数组nt，赋值给tab和table
                    table = tab = nt;
                    // sc赋值为了数组长度 - 数组长度 右移 2位     16 - 4 = 12
                    // 将sc赋值为下次扩容的阈值
                    sc = n - (n >>> 2);
                }
            } finally {
                // 将赋值好的sc，设置给sizeCtl
                sizeCtl = sc;
            }
            break;
```

```
        }
    }
    return tab;
}
```

1.2.4 putVal方法-添加数据到链表

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    // 省略部分代码.....
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // n:数组长度
        // i:当前Node需要存放的索引位置
        // f: 当前数组i索引位置的Node对象
        // fn：当前数组i索引位置上数据的hash值
        // 省略部分代码.....
        else {
            // 声明变量为oldVal
            V oldVal = null;
            // 基于当前索引位置的Node，作为锁对象.....
            synchronized (f) {
                // 判断当前位置的数据还是之前的f么.....（避免并发操作的安全问题）
                if (tabAt(tab, i) == f) {
                    // 再次判断hash值是否大于0（不是树）
                    if (fh >= 0) {
                        // binCount设置为1（在链表情况下，记录链表长度的一个标识）
                        binCount = 1;
                        // 死循环，每循环一次，对binCount
                        for (Node<K,V> e = f;; ++binCount) {
                            // 声明标识ek
                            K ek;
                            // 当前i索引位置的数据，是否和当前put的key的hash值一致
                            if (e.hash == hash &&
                                // 如果当前i索引位置数据的key和put的key == 返回为true
                                // 或者equals相等
                                ((ek = e.key) == key || (ek != null &&
key.equals(ek)))) {
                                // key一致，可能需要覆盖数据!
                                // 当前i索引位置数据的value复制给oldVal
                                oldVal = e.val;
                                // 如果传入的是false，代表key一致，覆盖value
                                // 如果传入的是true，代表key一致，什么都不做!
                                if (!onlyIfAbsent)
                                    // 覆盖value
                                    e.val = value;
                                break;
                            }
                            // 拿到当前指定的Node对象
                            Node<K,V> pred = e;
                            // 将e指向下一个Node对象,如果next指向的是一个null，可以挂在
当前Node下面
                            if ((e = e.next) == null) {
                                // 将hash，key，value封装为Node对象，挂在pred的next上
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
```

```
                            }
                        }
                    }
                    // 省略部分代码.....
                }
            }
            // binCount长度不为0
            if (binCount != 0) {
                // binCount是否大于8（链表长度是否 >= 8）
                if (binCount >= TREEIFY_THRESHOLD)
                    // 尝试转为红黑树或者扩容
                    // 基于treeifyBin方法和上面的if判断，可以得知链表想要转为红黑树，必须保证数组长度大于等于64，并且链表长度大于等于8
                    // 如果数组长度没有达到64的话，会首先将数组扩容
                    treeifyBin(tab, i);
                // 如果出现了数据覆盖的情况，
                if (oldVal != null)
                    // 返回之前的值
                    return oldVal;
                break;
            }
        }
    }
    // 省略部分代码.....
}

// 为什么链表长度为8转换为红黑树，不是能其他数值嘛？
// 因为布松分布
 The main disadvantage of per-bin locks is that other update
 * operations on other nodes in a bin list protected by the same
 * lock can stall, for example when user equals() or mapping
 * functions take a long time.  However, statistically, under
 * random hash codes, this is not a common problem.  Ideally, the
 * frequency of nodes in bins follows a Poisson distribution
 * (http://en.wikipedia.org/wiki/Poisson_distribution) with a
 * parameter of about 0.5 on average, given the resizing threshold
 * of 0.75, although with a large variance because of resizing
 * granularity. Ignoring variance, the expected occurrences of
 * list size k are (exp(-0.5) * pow(0.5, k) / factorial(k)). The
 * first values are:
 *
 * 0:    0.60653066
 * 1:    0.30326533
 * 2:    0.07581633
 * 3:    0.01263606
 * 4:    0.00157952
 * 5:    0.00015795
 * 6:    0.00001316
 * 7:    0.00000094
 * 8:    0.00000006
 * more: less than 1 in ten million
```

1.3 扩容操作

1.3.1 treeifyBin方法触发扩容

```
// 在链表长度大于等于8时，尝试将链表转为红黑树
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    // 数组不能为空
    if (tab != null) {
        // 数组的长度n，是否小于64
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            // 如果数组长度小于64，不能将链表转为红黑树，先尝试扩容操作
            tryPresize(n << 1);
        // 省略部分代码……
    }
}
```

1.3.2 tryPreSize方法-针对putAll的初始化操作

```
// size是将之前的数组长度 左移 1位得到的结果
private final void tryPresize(int size) {
    // 如果扩容的长度达到了最大值，就使用最大值
    // 否则需要保证数组的长度为2的n次幂
    // 这块的操作，是为了初始化操作准备的，因为调用putAll方法时，也会触发tryPresize方法
    // 如果刚刚new的ConcurrentHashMap直接调用了putAll方法的话，会通过tryPresize方法进行初始化
    int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
        tableSizeFor(size + (size >>> 1) + 1);
    // 这些代码和initTable一模一样
    // 声明sc
    int sc;
    // 将sizeCtl的值赋值给sc，并判断是否大于0，这里代表没有初始化操作，也没有扩容操作
    while ((sc = sizeCtl) >= 0) {
        // 将ConcurrentHashMap的table赋值给tab，并声明数组长度n
        Node<K,V>[] tab = table; int n;
        // 数组是否需要初始化
        if (tab == null || (n = tab.length) == 0) {
            // 进来执行初始化
            // sc是初始化长度，初始化长度如果比计算出来的c要大的话，直接使用sc，如果没有sc大，
            // 说明sc无法容纳下putAll中传入的map，使用更大的数组长度
            n = (sc > c) ? sc : c;
            // 设置sizeCtl为-1，代表初始化操作
            if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
                try {
                    // 再次判断数组的引用有没有变化
                    if (table == tab) {
                        // 初始化数组
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        // 数组赋值
                        table = nt;
                        // 计算扩容阈值
                        sc = n - (n >>> 2);
                    }
                } finally {
                    // 最终赋值给sizeCtl
```

```
                    sizeCtl = sc;
                }
            }
        }
        // 如果计算出来的长度c如果小于等于sc，直接退出循环结束方法
        // 数组长度大于等于最大长度了，直接退出循环结束方法
        else if (c <= sc || n >= MAXIMUM_CAPACITY)
            break;
        // 省略部分代码
    }
}

// 将c这个长度设置到最近的2的n次幂的值，    15 - 16     17 - 32
// c == size + (size >>> 1) + 1
// size = 17
00000000 00000000 00000000 00010001
+
00000000 00000000 00000000 00001000
+
00000000 00000000 00000000 00000001
// c = 26
00000000 00000000 00000000 00011010
private static final int tableSizeFor(int c) {
    // 00000000 00000000 00000000 00011001
    int n = c - 1;
    // 00000000 00000000 00000000 00011001
    // 00000000 00000000 00000000 00001100
    // 00000000 00000000 00000000 00011101
    n |= n >>> 1;
    // 00000000 00000000 00000000 00011101
    // 00000000 00000000 00000000 00000111
    // 00000000 00000000 00000000 00011111
    n |= n >>> 2;
    // 00000000 00000000 00000000 00011111
    // 00000000 00000000 00000000 00000001
    // 00000000 00000000 00000000 00011111
    n |= n >>> 4;
    // 00000000 00000000 00000000 00011111
    // 00000000 00000000 00000000 00000000
    // 00000000 00000000 00000000 00011111
    n |= n >>> 8;
    // 00000000 00000000 00000000 00011111
    n |= n >>> 16;
    // 00000000 00000000 00000000 00100000
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;

}
```

1.3.3 tryPreSize方法-计算扩容戳并且查看BUG

```
private final void tryPresize(int size) {
    // n：数组长度
    while ((sc = sizeCtl) >= 0) {
        // 判断当前的tab是否和table一致，
        else if (tab == table) {
            // 计算扩容表示戳，根据当前数组的长度计算一个16位的扩容戳
            // 第一个作用是为了保证后面的sizeCtl赋值时，保证sizeCtl为小于-1的负数
```

```
            // 第二个作用用来记录当前是从什么长度开始扩容的
            int rs = resizeStamp(n);
            // BUG --- sc < 0，永远进不去~
            // 如果sc小于0，代表有线程正在扩容。
            if (sc < 0) {
                // 省略部分代码……协助扩容的代码（进不来~~~~）
            }
            // 代表没有线程正在扩容，我是第一个扩容的。
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                // 省略部分代码……第一个扩容的线程……
        }
    }
}
// 计算扩容表示戳
// 32 =  00000000 00000000 00000000 00100000
// Integer.numberOfLeadingZeros(32) = 26
// 1 << (RESIZE_STAMP_BITS - 1)
// 00000000 00000000 10000000 00000000
// 00000000 00000000 00000000 00011010
// 00000000 00000000 10000000 00011010
static final int resizeStamp(int n) {
    return Integer.numberOfLeadingZeros(n) | (1 << (RESIZE_STAMP_BITS - 1));
}
```

1.3.4 tryPreSize方法-对sizeCtl的修改以及条件判断的BUG

```
private final void tryPresize(int size) {
    // sc默认为sizeCtl
    while ((sc = sizeCtl) >= 0) {
        else if (tab == table) {
            // rs：扩容戳  00000000 00000000 10000000 00011010
            int rs = resizeStamp(n);
            if (sc < 0) {
                // 说明有线程正在扩容，过来帮助扩容
                Node<K,V>[] nt;
                // 依然有BUG
                // 当前线程扩容时，老数组长度是否和我当前线程扩容时的老数组长度一致
                // 00000000 00000000 10000000 00011010
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs
                    // 10000000 00011010 00000000 00000010
                    // 00000000 00000000 10000000 00011010
                    // 这两个判断都是有问题的，核心问题就应该先将rs左移16位，再追加当前值。
                    // 这两个判断是BUG
                    // 判断当前扩容是否已经即将结束
                    || sc == rs + 1   // sc == rs << 16 + 1 BUG
                    // 判断当前扩容的线程是否达到了最大限度
                    || sc == rs + MAX_RESIZERS   // sc == rs << 16 +
MAX_RESIZERS BUG
                    // 扩容已经结束了。
                    || (nt = nextTable) == null
                    // 记录迁移的索引位置，从高位往低位迁移，也代表扩容即将结束。
                    || transferIndex <= 0)
                    break;
                // 如果线程需要协助扩容，首先就是对sizeCtl进行+1操作，代表当前要进来一个线
程协助扩容
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
```

```
                    // 上面的判断没进去的话，nt就代表新数组
                    transfer(tab, nt);
                }
                // 是第一个来扩容的线程
                // 基于CAS将sizeCtl修改为  10000000 00011010 00000000 00000010
                // 将扩容戳左移16位之后，符号位是1，就代码这个值为负数
                // 低16位在表示当前正在扩容的线程有多少个，
                // 为什么低位值为2时，代表有一个线程正在扩容
                // 每一个线程扩容完毕后，会对低16位进行-1操作，当最后一个线程扩容完毕后，减1的结果还是-1，
                // 当值为-1时，要对老数组进行一波扫描，查看是否有遗漏的数据没有迁移到新数组
                else if (U.compareAndSwapInt(this, SIZECTL, sc,(rs << RESIZE_STAMP_SHIFT) + 2))
                    // 调用transfer方法，并且将第二个参数设置为null，就代表是第一次来扩容!
                    transfer(tab, null);
            }
        }
    }
```

1.3.5 transfer方法-计算每个线程迁移的长度

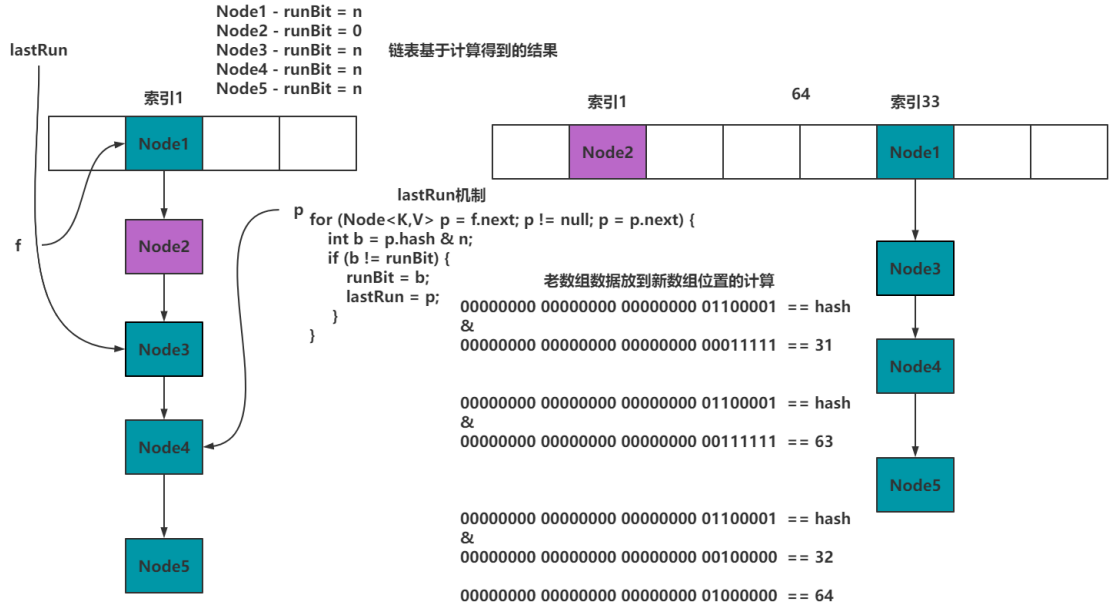
```
// 开始扩容    tab=oldTable
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    // n = 数组长度
    // stride = 每个线程一次性迁移多少数据到新数组
    int n = tab.length, stride;
    // 基于CPU的内核数量来计算，每个线程一次性迁移多少长度的数据最合理
    // NCPU = 4
    // 举个栗子：数组长度为1024 - 512 - 256 - 128 / 4 = 32
    // MIN_TRANSFER_STRIDE = 16,为每个线程迁移数据的最小长度
    if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
        stride = MIN_TRANSFER_STRIDE;
    // 根据CPU计算每个线程一次迁移多长的数据到新数组，如果结果大于16，使用计算结果。 如果结果小于16，就使用最小长度16
}
```

1.3.6 transfer方法-构建新数组并查看标识属性

```
// 以32长度数组扩容到64位例子
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    // n = 老数组长度    32
    // stride = 步长    16
    // 第一个进来扩容的线程需要把新数组构建出来
    if (nextTab == null) {
        try {
            // 将原数组长度左移一位，构建新数组长度
            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];
            // 赋值操作
            nextTab = nt;
        } catch (Throwable ex) {
            // 到这说明已经达到数组长度的最大取值范围
            sizeCtl = Integer.MAX_VALUE;
            // 设置sizeCtl后直接结束
            return;
        }
        // 将成员变量的新数组赋值
```

```
            nextTable = nextTab;
            // 迁移数据时，用到的标识，默认值为老数组长度
            transferIndex = n;   // 32
        }
        // 新数组长度
        int nextn = nextTab.length;  // 64
        // 在老数组迁移完数据后，做的标识
        ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
        // 迁移数据时，需要用到的标识
        boolean advance = true;
        boolean finishing = false;
        // 省略部分代码
    }
```

1.3.7 transfer方法-线程领取迁移任务




```
    // 以32长度扩容到64位为例子
    private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
        // n：32
        // stride：16
        int n = tab.length, stride;
        if (nextTab == null) {
            // 省略部分代码.....
            // nextTable：新数组
            nextTable = nextTab;
            // transferIndex：0
            transferIndex = n;
        }
        // nextn：64
        int nextn = nextTab.length;
        ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
        // advance：true，代表当前线程需要接收任务，然后再执行迁移，   如果为false，代表已经接收完任务
        boolean advance = true;
        // finishing：false，是否迁移结束！
        boolean finishing = false;
        // 循环.....
        // i = 15      代表当前线程迁移数据的索引值！！
        // bound = 0
```

```
    for (int i = 0, bound = 0;;) {
        // f = null
        // fh = 0
        Node<K,V> f; int fh;
        // 当前线程要接收任务
        while (advance) {
            // nextIndex = 16
            // nextBound = 16
            int nextIndex, nextBound;
            // 第一次进来，这两个判断肯定进不去。
            // 对i进行--，并且判断当前任务是否处理完毕！
            if (--i >= bound || finishing)
                advance = false;
            // 判断transferIndex是否小于等于0，代表没有任务可领取，结束了。
            // 在线程领取任务会，会对transferIndex进行修改，修改为transferIndex -
stride
            // 在任务都领取完之后，transferIndex肯定是小于等于0的，代表没有迁移数据的任务可
以领取
            else if ((nextIndex = transferIndex) <= 0) {
                i = -1;
                advance = false;
            }
            // 当前线程尝试领取任务
            else if (U.compareAndSwapInt
                     (this, TRANSFERINDEX, nextIndex,
                      nextBound = (nextIndex > stride ? nextIndex - stride :
0))) {
                // 对bound赋值
                bound = nextBound;
                // 对i赋值
                i = nextIndex - 1;
                // 设置advance设置为false，代表当前线程领取到任务了。
                advance = false;
            }
        }
        // 开始迁移数据，并且在迁移完毕后，会将advance设置为true

    }
}
```

1.3.8 transfer方法-迁移结束操作

```
// 以32长度扩容到64位为例
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    for (int i = 0, bound = 0;;) {
        while (advance) {
        // 判断扩容是否已经结束！
        // i < 0：当前线程没有接收到任务！
        // i >= n：迁移的索引位置，不可能大于数组的长度，不会成立
        // i + n >= nextn：因为i最大值就是数组索引的最大值，不会成立
        if (i < 0 || i >= n || i + n >= nextn) {
            // 如果进来，代表当前线程没有接收到任务
            int sc;
            // finishing为true，代表扩容结束
            if (finishing) {
                // 将nextTable新数组设置为null
                nextTable = null;
```

```
                    // 将当前数组的引用指向了新数组~
                    table = nextTab;
                    // 重新计算扩容阈值     64 - 16 = 48
                    sizeCtl = (n << 1) - (n >>> 1);
                    // 结束扩容
                    return;
                }
                // 当前线程没有接收到任务，让当前线程结束扩容操作。
                // 采用CAS的方式，将sizeCtl - 1，代表当前并发扩容的线程数 - 1
                if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                    // sizeCtl的高16位是基于数组长度计算的扩容戳，低16位是当前正在扩容的线程个
数
                    if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                        // 代表当前线程并不是最后一个退出扩容的线程，直接结束当前线程扩容
                        return;
                    // 如果是最后一个退出扩容的线程，将finishing和advance设置为true
                    finishing = advance = true;
                    // 将i设置为老数组长度，让最后一个线程再从尾到头再次检查一下，是否数据全部迁
移完毕。
                    i = n;
                }
            }
            // 开始迁移数据，并且在迁移完毕后，会将advance设置为true
        }
    }
```

1.3.9 transfer方法-迁移数据（链表）

```
// 以32长度扩容到64位为例
private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
    // 省略部分代码.....
    for (int i = 0, bound = 0;;) {
        // 省略部分代码.....
        if (i < 0 || i >= n || i + n >= nextn) {
             // 省略部分代码.....
        }
        // 开始迁移数据，并且在迁移完毕后，会将advance设置为true
        // 获取指定i位置的Node对象，并且判断是否为null
        else if ((f = tabAt(tab, i)) == null)
            // 当前桶位置没有数据，无需迁移，直接将当前桶位置设置为fwd
            advance = casTabAt(tab, i, null, fwd);
        // 拿到当前i位置的hash值，如果为MOVED，证明数据已经迁移过了。
        else if ((fh = f.hash) == MOVED)
            // 一般是给最后扫描时，使用的判断，如果迁移完毕，直接跳过当前位置。
            advance = true; // already processed
        else {
            // 当前桶位置有数据，先锁住当前桶位置。
            synchronized (f) {
                // 判断之前取出的数据是否为当前的数据。
                if (tabAt(tab, i) == f) {
                    // ln：null  - lowNode
                    // hn：null  - highNode
                    Node<K,V> ln, hn;
                    // hash大于0，代表当前Node属于正常情况，不是红黑树，使用链表方式迁移数
据
                    if (fh >= 0) {
                        // lastRun机制
```

```
                        //   000000000010000
                        // 这种运算结果只有两种，要么是0，要么是n
                        int runBit = fh & n;
                        // 将f赋值给lastRun
                        Node<K,V> lastRun = f;
                        // 循环的目的就是为了得到链表下经过hash & n结算，结果一致的最后一
些数据
                        // 在迁移数据时，值需要迁移到lastRun即可，剩下的指针不需要变换。
                        for (Node<K,V> p = f.next; p != null; p = p.next) {
                            int b = p.hash & n;
                            if (b != runBit) {
                                runBit = b;
                                lastRun = p;
                            }
                        }
                        // runBit == 0，赋值给ln
                        if (runBit == 0) {
                            ln = lastRun;
                            hn = null;
                        }
                        // rubBit == n,赋值给hn
                        else {
                            hn = lastRun;
                            ln = null;
                        }
                        // 循环到lastRun指向的数据即可，后续不需要再遍历
                        for (Node<K,V> p = f; p != lastRun; p = p.next) {
                            // 获取当前Node的hash值，key值，value值。
                            int ph = p.hash; K pk = p.key; V pv = p.val;
                            // 如果hash&n为0，挂到lowNode上
                            if ((ph & n) == 0)
                                ln = new Node<K,V>(ph, pk, pv, ln);
                            // 如果hash&n为n，挂到highNode上
                            else
                                hn = new Node<K,V>(ph, pk, pv, hn);
                        }
                        // 采用CAS的方式，将ln挂到新数组的原位置
                        setTabAt(nextTab, i, ln);
                        // 采用CAS的方式，将hn挂到新数组的原位置 + 老数组长度
                        setTabAt(nextTab, i + n, hn);
                        // 采用CAS的方式，将当前桶位置设置为fwd
                        setTabAt(tab, i, fwd);
                        // advance设置为true，保证可以进入到while循环，对i进行--操作
                        advance = true;
                    }
                    // 省略迁移红黑树的操作
                }
            }
        }
    }
}
```

1.3.10 helpTransfer方法-协助扩容

```
// 在添加数据时，如果插入节点的位置的数据，hash值为-1，代表当前索引位置数据已经被迁移到了新数组
// tab：老数组
// f：数组上的Node节点
final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
    // nextTab：新数组
    // sc：给sizeCtl做临时变量
    Node<K,V>[] nextTab; int sc;
    // 第一个判断：老数组不为null
    // 第二个判断：新数组不为null  （将新数组赋值给nextTab）
    if (tab != null &&
        (f instanceof ForwardingNode) && (nextTab =
((ForwardingNode<K,V>)f).nextTable) != null) {
        // ConcurrentHashMap正在扩容
        // 基于老数组长度计算扩容戳
        int rs = resizeStamp(tab.length);
        // 第一个判断：fwd中的新数组，和当前正在扩容的新数组是否相等。     相等：可以协助扩容。不相等：要么扩容结束，要么开启了新的扩容
        // 第二个判断：老数组是否改变了。      相等：可以协助扩容。不相等：扩容结束了
        // 第三个判断：如果正在扩容，sizeCtl肯定为负数，并且给sc赋值
        while (nextTab == nextTable && table == tab && (sc = sizeCtl) < 0) {
            // 第一个判断：将sc右移16位，判断是否与扩容戳一致。 如果不一致，说明扩容长度不一样，退出协助扩容
            // 第二个、三个判断是BUG：
            /*
                sc == rs << 16 + 1 ||       如果+1和当前sc一致，说明扩容已经到了最后检查的阶段
                sc == rs << 16 + MAX_RESIZERS ||     判断协助扩容的线程是否已经达到了最大值
            */
            // 第四个判断：transferIndex是从高索引位置到低索引位置领取数据的一个核心属性，如果满足 小于等于0，说明任务被领光了。
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs ||
                sc == rs + 1 ||
                sc == rs + MAX_RESIZERS ||
                transferIndex <= 0)
                // 不需要协助扩容
                break;
            // 将sizeCtl + 1，进来协助扩容
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                // 协助扩容
                transfer(tab, nextTab);
                break;
            }
        }
        return nextTab;
    }
    return table;
}
```

1.4 红黑树操作

在前面搞定了关于数据+链表的添加和扩容操作，现在要搞定红黑树。因为红黑树的操作有点乱，先对红黑树结构有一定了解。

1.4.1 什么是红黑树

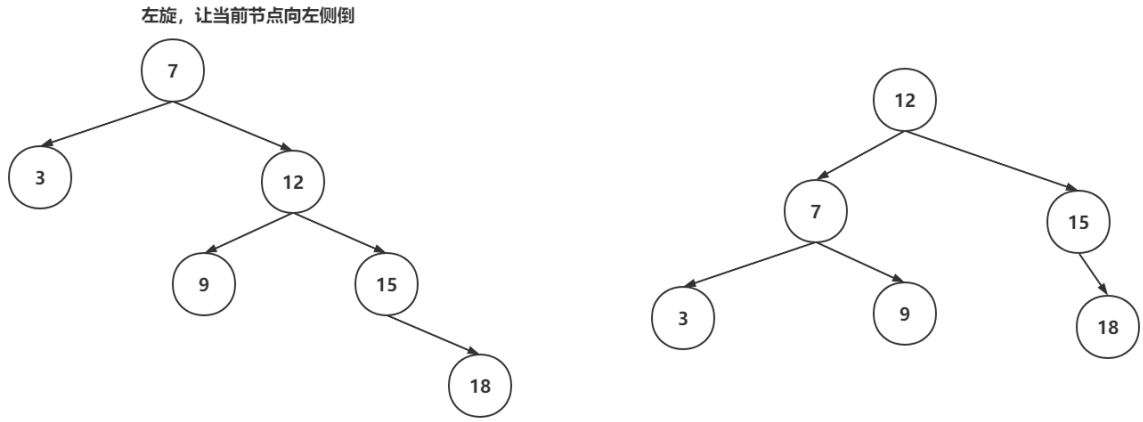
红黑树是一种特殊的平衡二叉树，首选具备了平衡二叉树的特点：左子树和右子数的高度差不会超过1，如果超过了，平衡二叉树就会基于左旋和右旋的操作，实现自平衡。

红黑树在保证自平衡的前提下，还保证了自己的几个特性：

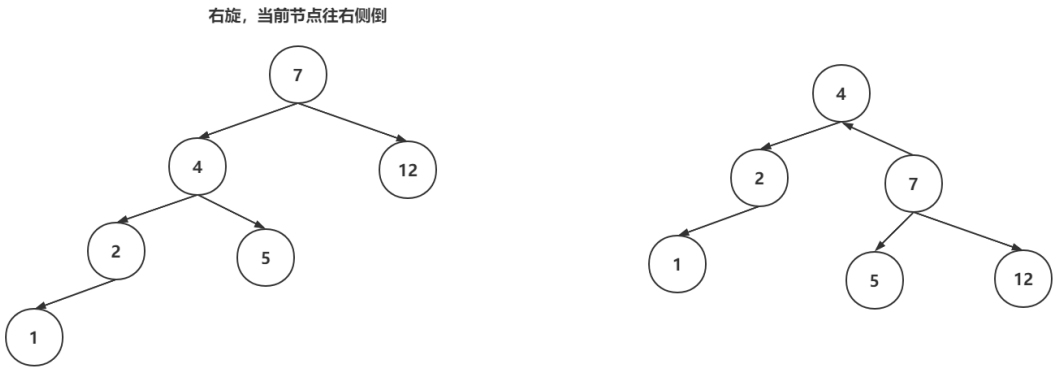
- 每个节点必须是红色或者黑色。
- 根节点必须是黑色。
- 如果当前节点是红色，子节点必须是黑色
- 所有叶子节点都是黑色。
- 从任意节点到每个叶子节点的路径中，黑色节点的数量是相同的。

当对红黑树进行增删操作时，可能会破坏平衡或者是特性，这是红黑树就需要基于左旋、右旋、变色来保证平衡和特性。

左旋操作：



右旋操作：



变色操作：节点的颜色从黑色变为红色，或者从红色变为黑色，就成为变色。变色操作是在增删数据之后，可能出现的操作。插入数据时，插入节点的颜色一般是红色，因为插入红色节点的破坏红黑树结构的可能性比较低的。如果破坏了红黑树特性，会通过变色来调整

红黑树相对比较复杂，完整的红黑树代码400~500行内容，没有必要全部记下来，或者首先红黑树。

如果向细粒度掌握红黑树的结构：https://www.mashibing.com/subject/21?courseNo=339

1.4.2 TreeifyBin方法-封装TreeNode和双向链表

```
// 将链表转为红黑树的准备操作
private final void treeifyBin(Node<K,V>[] tab, int index) {
    // b：当前索引位置的Node
    Node<K,V> b; int sc;
    if (tab != null) {
        // 省略部分代码
        // 开启链表转红黑树操作
        // 当前桶内有数据，并且是链表结构
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
            // 加锁，保证线程安全
            synchronized (b) {
                // 再次判断数据是否有变化，DCL
                if (tabAt(tab, index) == b) {
                    // 开启准备操作，将之前的链表中的每一个Node，封装为TreeNode，作为双向
链表
                    // hd：是整个双向链表的第一个节点。
                    // tl：是单向链表转换双向链表的临时存储变量
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next) {
                        TreeNode<K,V> p = new TreeNode<K,V>(e.hash, e.key,
e.val, null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    // hd就是整个双向链表
                    // TreeBin的有参构建，将双向链表转为了红黑树。
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

1.4.3 TreeBin有参构造-双向链表转为红黑树

TreeBin中不但保存了红黑树结构，同时还保存在一套双向链表

```
// 将双向链表转为红黑树的操作。 b：双向链表的第一个节点
// TreeBin继承自Node，root：代表树的根节点，first：双向链表的头节点
TreeBin(TreeNode<K,V> b) {
    // 构建Node，并且将hash值设置为-2
    super(TREEBIN, null, null, null);
    // 将双向链表的头节点赋值给first
    this.first = b;
    // 声明r的TreeNode，最后会被赋值为根节点
    TreeNode<K,V> r = null;
    // 遍历之前封装好的双向链表
    for (TreeNode<K,V> x = b, next; x != null; x = next) {
        next = (TreeNode<K,V>)x.next;

        // 先将左右子节点清空
```

```
        x.left = x.right = null;
        // 如果根节点为null，第一次循环
        if (r == null) {
            // 将第一个节点设置为当前红黑树的根节点
            x.parent = null;  // 根节点没父节点
            x.red = false;  // 不是红色，是黑色
            r = x; // 将当前节点设置为r
        }
        // 已经有根节点，当前插入的节点要作为父节点的左子树或者右子树
        else {
            // 拿到了当前节点key和hash值。
            K k = x.key;
            int h = x.hash;
            Class<?> kc = null;
            // 循环？
            for (TreeNode<K,V> p = r;;) {
                // dir：如果为-1，代表要插入到父节点的左边，如果为1，代表要插入的父节点的右
边
                // ph：是父节点的hash值
                int dir, ph;
                // pk：是父节点的key
                K pk = p.key;
                // 父节点的hash值，大于当前节点的hash值，就设置为-1，代表要插入到父节点的左
边
                if ((ph = p.hash) > h)
                    dir = -1;
                // 父节点的hash值，小于当前节点的hash值，就设置为1，代表要插入到父节点的右
边
                else if (ph < h)
                    dir = 1;
                // 父节点的hash值和当前节点hash值一致，基于compare方式判断到底放在左子树还
是右子树
                else if ((kc == null &&
                          (kc = comparableClassFor(k)) == null) ||
                         (dir = compareComparables(kc, k, pk)) == 0)
                    dir = tieBreakOrder(k, pk);
                // 拿到当前父节点。
                TreeNode<K,V> xp = p;
                // 将p指向p的left、right，并且判断是否为null
                // 如果为null，代表可以插入到这位置。
                if ((p = (dir <= 0) ? p.left : p.right) == null) {
                    // 进来就说明找到要存放当前节点的位置了
                    // 将当前节点的parent指向父节点
                    x.parent = xp;
                    // 根据dir的值，将父节点的left、right指向当前节点
                    if (dir <= 0)
                        xp.left = x;
                    else
                        xp.right = x;
                    // 插入一个节点后，做一波平衡操作
                    r = balanceInsertion(r, x);
                    break;
                }
            }
        }
    }
    // 将根节点复制给root
    this.root = r;
```

```
    // 检查红黑树结构
    assert checkInvariants(root);
}
```

1.4.4 balanceInsertion方法-保证红黑树平衡以及特性

```
// 红黑树的插入动画：https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
// 红黑树做自平衡以及保证特性的操作。   root：根节点，   x：当前节点
static <K,V> TreeNode<K,V> balanceInsertion(TreeNode<K,V> root, TreeNode<K,V> x)
{
    // 先将节点置位红色
    x.red = true;
    // xp：父节点
    // xpp：爷爷节点
    // xppl：爷爷节点的左子树
    // xxpr：爷爷节点的右子树
    for (TreeNode<K,V> xp, xpp, xppl, xppr;;) {
        // 拿到父节点，并且父节点为红
        if ((xp = x.parent) == null) {
            // 当前节点为根节点，置位黑色
            x.red = false;
            return x;
        }
        // 父节点不是红色，爷爷节点为null
        else if (!xp.red || (xpp = xp.parent) == null)
            // 什么都不做，直接返回
            return root;

        // ====================================
        // 左子树的操作
        if (xp == (xppl = xpp.left)) {
            // 通过变色满足红黑树特性
            if ((xppr = xpp.right) != null && xppr.red) {
                // 叔叔节点和父节点变为黑色
                xppr.red = false;
                xp.red = false;
                // 爷爷节点置位红色
                xpp.red = true;
                // 让爷爷节点作为当前节点，再走一次循环
                x = xpp;
            }

            else {
                // 如果当前节点是右子树，通过父节点的左旋，变为左子树的结构
                if (x == xp.right) {、
                    // 父节点做左旋操作
                    root = rotateLeft(root, x = xp);
                    xpp = (xp = x.parent) == null ? null : xp.parent;
                }
                if (xp != null) {
                    // 父节点变为黑色
                    xp.red = false;
                    if (xpp != null) {
                        // 爷爷节点变为红色
                        xpp.red = true;
                        // 爷爷节点做右旋操作
                        root = rotateRight(root, xpp);
```

```
                    }
                }
            }
        }

        // 右子树（只讲左子树就足够了，因为业务都是一样的）
        else {
            if (xppl != null && xppl.red) {
                xppl.red = false;
                xp.red = false;
                xpp.red = true;
                x = xpp;
            }
            else {
                if (x == xp.left) {
                    root = rotateRight(root, x = xp);
                    xpp = (xp = x.parent) == null ? null : xp.parent;
                }
                if (xp != null) {
                    xp.red = false;
                    if (xpp != null) {
                        xpp.red = true;
                        root = rotateLeft(root, xpp);
                    }
                }
            }
        }
    }
}
```

1.4.5 putTreeVal方法-添加节点

整体操作就是判断当前节点要插入到左子树，还是右子数，还是覆盖操作。

确定左子树和右子数之后，直接维护双向链表和红黑树结构，并且再判断是否需要自平衡。

TreeBin的双向链表用的头插法。

```
// 添加节点到红黑树内部
final TreeNode<K,V> putTreeVal(int h, K k, V v) {
    // Class对象
    Class<?> kc = null;
    // 搜索节点
    boolean searched = false;
    // 死循环，p节点是根节点的临时引用
    for (TreeNode<K,V> p = root;;) {
        // dir：确定节点是插入到左子树还是右子数
        // ph：父节点的hash值
        // pk：父节点的key
        int dir, ph; K pk;
        // 根节点是否为诶null，把当前节点置位根节点
        if (p == null) {
            first = root = new TreeNode<K,V>(h, k, v, null, null);
            break;
        }
        // 判断当前节点要放在左子树还是右子数
        else if ((ph = p.hash) > h)
            dir = -1;
```

```
            else if (ph < h)
                dir = 1;
            // 如果key一致，直接返回p，由putVal去修改数据
            else if ((pk = p.key) == k || (pk != null && k.equals(pk)))
                return p;
            // hash值一致，但是key的==和equals都不一样，基于Compare去判断
            else if ((kc == null &&
                      (kc = comparableClassFor(k)) == null) ||
                     // 基于compare判断也是一致，就进到if判断
                     (dir = compareComparables(kc, k, pk)) == 0) {
                // 开启搜索，查看是否有相同的key，只有第一次循环会执行。
                if (!searched) {
                    TreeNode<K,V> q, ch;
                    searched = true;
                    if (((ch = p.left) != null &&
                         (q = ch.findTreeNode(h, k, kc)) != null) ||
                        ((ch = p.right) != null &&
                         (q = ch.findTreeNode(h, k, kc)) != null))
                        // 如果找到直接返回
                        return q;
                }
                // 再次判断hash大小，如果小于等于，返回-1
                dir = tieBreakOrder(k, pk);
            }

            // xp是父节点的临时引用
            TreeNode<K,V> xp = p;
            // 基于dir判断是插入左子树还有右子数，并且给p重新赋值
            if ((p = (dir <= 0) ? p.left : p.right) == null) {
                // first引用拿到
                TreeNode<K,V> x, f = first;
                // 将当前节点构建出来
                first = x = new TreeNode<K,V>(h, k, v, f, xp);
                // 因为当前的TreeBin除了红黑树还维护这一个双向链表，维护双向链表的操作
                if (f != null)
                    f.prev = x;
                // 维护红黑树操作
                if (dir <= 0)
                    xp.left = x;
                else
                    xp.right = x;
                // 如果如节点是黑色的，当前节点红色即可，说明现在插入的节点没有影响红黑树的平衡
                if (!xp.red)
                    x.red = true;
                else {
                    // 说明插入的节点是黑色的
                    // 加锁操作
                    lockRoot();
                    try {
                        // 自平衡一波。
                        root = balanceInsertion(root, x);
                    } finally {
                        // 释放锁操作
                        unlockRoot();
                    }
                }
                break;
            }
```

```
    }
    // 检查一波红黑树结构
    assert checkInvariants(root);
    // 代表插入了新节点
    return null;
}
```

1.4.6 TreeBin的锁操作

TreeBin的锁操作，没有基于AQS，仅仅是对一个变量的CAS操作和一些业务判断实现的。

每次读线程操作，对lockState+4。

写线程操作，对lockState + 1，如果读操作占用着线程，就先+2，waiter是当前线程，并挂起当前线程

```
// TreeBin的锁操作
// 如果说有读线程在读取红黑树的数据，这时，写线程要阻塞（做平衡前）
// 如果有写线程正在操作红黑树（做平衡），读线程不会阻塞，会读取双向链表
// 读读不会阻塞！
static final class TreeBin<K,V> extends Node<K,V> {

    // waiter：等待获取写锁的线程
    volatile Thread waiter;
    // lockState：当前TreeBin的锁状态
    volatile int lockState;

    // 对锁状态进行运算的值
    // 有线程拿着写锁
    static final int WRITER = 1;
    // 有写线程，再等待获取写锁
    static final int WAITER = 2;
    // 读线程，在红黑树中检索时，需要先对lockState + READER
    // 这个只会在读操作中遇到
    static final int READER = 4;

    // 加锁-写锁
    private final void lockRoot() {
        // 将lockState从0设置为1，代表拿到写锁成功
        if (!U.compareAndSwapInt(this, LOCKSTATE, 0, WRITER))
            // 如果写锁没拿到，执行contendedLock
            contendedLock();
    }

    // 释放写锁
    private final void unlockRoot() {
        lockState = 0;
    }


    // 写线程没有拿到写锁，执行当前方法
    private final void contendedLock() {
        // 是否有线程正在等待
        boolean waiting = false;
        // 死循环，s是lockState的临时变量
        for (int s;;) {
            //
            // lockState & 11111101 ,只要结果为0，说明当前写锁，和读锁都没线程获取
            if (((s = lockState) & ~WAITER) == 0) {
```

```
                // CAS一波，尝试将lockState再次修改为1，
                if (U.compareAndSwapInt(this, LOCKSTATE, s, WRITER)) {
                    // 成功拿到锁资源，并判断是否在waiting
                    if (waiting)
                        // 如果当前线程挂起过，直接将之前等待的线程资源设置为null
                        waiter = null;
                    return;
                }
            }
            // 有读操作在占用资源
            // lockState &  00000010,代表当前没有写操作挂起等待。
            else if ((s & WAITER) == 0) {
                // 基于CAS，将LOCKSTATE的第二位设置为1
                if (U.compareAndSwapInt(this, LOCKSTATE, s, s | WAITER)) {
                    // 如果成功，代表当前线程可以waiting等待了
                    waiting = true;
                    waiter = Thread.currentThread();
                }
            }
            else if (waiting)
                // 挂起当前线程！会由写操作唤醒
                LockSupport.park(this);
        }
    }
}
```

1.4.7 transfer迁移数据

首先红黑结构的数据迁移是基于双向链表封装的数据。

如果高低位的长度小于等于6，封装为链表迁移到新数组

如果高低位的长度大于6，依然封装为红黑树迁移到新数组

```
// 红黑树的迁移操作单独拿出来，TreeBin中不但有红黑树，还有双向链表，迁移的过程是基于双向链表迁移
TreeBin<K,V> t = (TreeBin<K,V>)f;
// lo，hi扩容后要放到新数组的高低位的链表
TreeNode<K,V> lo = null, loTail = null;
TreeNode<K,V> hi = null, hiTail = null;
// lc，hc在记录高低位数据的长度
int lc = 0, hc = 0;
// 遍历TreeBin中的双向链表
for (Node<K,V> e = t.first; e != null; e = e.next) {
    int h = e.hash;
    TreeNode<K,V> p = new TreeNode<K,V>(h, e.key, e.val, null, null);
    // 与老数组长度做&运算，基于结果确定需要存放到低位还是高位
    if ((h & n) == 0) {
        if ((p.prev = loTail) == null)
            lo = p;
        else
            loTail.next = p;
        loTail = p;
        // 低位长度++
        ++lc;
    }
    else {
        if ((p.prev = hiTail) == null)
```

```
                hi = p;
            else
                hiTail.next = p;
            hiTail = p;
            // 高位长度++
            ++hc;
        }
    }
    // 封装低位节点，如果低位节点的长度小于等于6，转回成链表。 如果长度大于6，需要重新封装红黑树
    ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) : (hc != 0) ? new TreeBin<K,V>
    (lo) : t;
    // 封装高位节点
    hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) : (lc != 0) ? new TreeBin<K,V>
    (hi) : t;
    // 低位数据设置到新数组
    setTabAt(nextTab, i, ln);
    // 高位数据设置到新数组
    setTabAt(nextTab, i + n, hn);
    // 当前位置数据迁移完毕，设置上fwd
    setTabAt(tab, i, fwd);
    // 开启前一个节点的数据迁移
    advance = true;
```

1.5 查询数据

1.5.1 get方法-查询数据的入口

在查询数据时，会先判断当前key对应的value，是否在数组上。

其次会判断当前位置是否属于特殊情况：数据被迁移、位置被占用、红黑树结构

最后判断链表上是否有对应的数据。

找到返回指定的value，找不到返回null即可

```
// 基于key查询value
public V get(Object key) {
    // tab：数组，  e：查询指定位置的节点  n：数组长度
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    // 基于传入的key，计算hash值
    int h = spread(key.hashCode());
    // 数组不为null，数组上得有数据，拿到指定位置的数组上的数据
    if ((tab = table) != null && (n = tab.length) > 0 && (e = tabAt(tab, (n - 1)
& h)) != null) {
        // 数组上数据恩地hash值，是否和查询条件key的hash一样
        if ((eh = e.hash) == h) {
            // key的==或者equals是否一致，如果一致，数组上就是要查询的数据
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        // 如果数组上的数据的hash为负数，有特殊情况，
        else if (eh < 0)
            // 三种情况，数据迁移走了，节点位置被占，红黑树
            return (p = e.find(h, key)) != null ? p.val : null;
        // 肯定走链表操作
        while ((e = e.next) != null) {
            // 如果hash值一致，并且key的==或者equals一致，返回当前链表位置的数据
```

```
            if (e.hash == h && ((ek = e.key) == key || (ek != null &&
key.equals(ek))))
                return e.val;
        }
    }
    // 如果上述三个流程都没有知道指定key对应的value，那就是key不存在，返回null即可
    return null;
}
```

1.5.2 ForwardingNode的find方法

在查询数据时，如果发现已经扩容了，去新数组上查询数据

在数组和链表上正常找key对应的value

可能依然存在特殊情况：

- 再次是fwd，说明当前线程可能没有获取到CPU时间片，导致CHM再次触发扩容，重新走当前方法
- 可能是被占用或者是红黑树，再次走另外两种find方法的逻辑

```
// 在查询数据时，发现当前桶位置已经放置了fwd，代表已经被迁移到了新数组
Node<K,V> find(int h, Object k) {
    // key：get(key)  h：key的hash   tab：新数组
    outer: for (Node<K,V>[] tab = nextTable;;) {
        // n：新数组长度，   e：新数组上定位的位置上的数组
        Node<K,V> e; int n;
        if (k == null || tab == null || (n = tab.length) == 0 || (e = tabAt(tab,
(n - 1) & h)) == null)
            return null;
        // 开始在新数组中走逻辑
        for (;;) {
            // eh：新数组位置的数据的hash
            int eh; K ek;
            // 判断hash是否一致，如果一致，再判断==或者equals。
            if ((eh = e.hash) == h && ((ek = e.key) == k || (ek != null &&
k.equals(ek))))
                // 在新数组找到了数据
                return e;
            // 发现到了新数组，hash值又小于0
            if (eh < 0) {
                // 套娃，发现刚刚在扩容，到了新数组，发现又扩容
                if (e instanceof ForwardingNode) {
                    // 再次重新走最外层循环，拿到最新的nextTable
                    tab = ((ForwardingNode<K,V>)e).nextTable;
                    continue outer;
                }
                else
                    // 占了，红黑树
                    return e.find(h, k);
            }
            // 说明不在数组上，往下走链表
            if ((e = e.next) == null)
                // 进来说明链表没找到，返回null
                return null;
        }
    }
}
```

1.5.3 ReservationNode的find方法

没什么说的，直接返回null

因为当前桶位置被占用的话，说明数据还没放到当前位置，当前位置可以理解为就是null

```
Node<K,V> find(int h, Object k) {
    return null;
}
```

1.5.4 TreeBin的find方法

在红黑树中执行find方法后，会有两个情况

- 如果有线程在持有写锁或者等待获取写锁，当前查询就要在双向链表中锁检索
- 如果没有线程持有写锁或者等待获取写锁，完全可以对lockState + 4，然后去红黑树中检索，并且在检索完毕后，需要对lockState - 4，再判断是否需要唤醒等待写锁的线程

```
// 在红黑树中检索数据
final Node<K,V> find(int h, Object k) {
    // 非空判断
    if (k != null) {
        // e：Treebin中的双向链表，
        for (Node<K,V> e = first; e != null; ) {
            int s; K ek;
            // s：TreeBin的锁状态
            // 00000010
            // 00000001
            if (((s = lockState) & (WAITER|WRITER)) != 0) {
                // 如果进来if，说明要么有写线程在等待获取写锁，要么是由写线程持有者写锁
                // 如果出现这个情况，他会去双向链表查询数据
                if (e.hash == h && ((ek = e.key) == k || (ek != null &&
k.equals(ek))))
                    return e;
                e = e.next;
            }
            // 说明没有线程等待写锁或者持有写锁，将lockState + 4，代表当前读线程可以去红黑
树中检索数据
            else if (U.compareAndSwapInt(this, LOCKSTATE, s, s + READER)) {
                TreeNode<K,V> r, p;
                try {
                    // 基于findTreeNode在红黑树中检索数据
                    p = ((r = root) == null ? null : r.findTreeNode(h, k,
null));
                } finally {
                    Thread w;
                    // 会对lockState - 4，读线程拿到数据了，释放读锁
                    // 可以确认，如果-完4，等于WAITER，说明有写线程可能在等待，判断waiter
是否为null
                    if (U.getAndAddInt(this, LOCKSTATE, -READER) ==
(READER|WAITER) && (w = waiter) != null)
                        // 当前我是最后一个在红黑树中检索的线程，同时有线程在等待持有写锁，
唤醒等待的写线程
                        LockSupport.unpark(w);
                }
                return p;
            }
```

```
        }
    }
    return null;
}
```

1.5.6 TreeNode的findTreeNode方法

红黑树的检索方式，套路很简单，及时基于hash值，来决定去找左子树还有右子数。

如果hash值一致，判断是否 == 、equals，满足就说明找到数据

如果hash值一致，并不是找的数据，基于compare方式，再次决定找左子树还是右子数，知道找到当前节点的子节点为null，停住。

```
// 红黑树中的检索方法
final TreeNode<K,V> findTreeNode(int h, Object k, Class<?> kc) {
    if (k != null) {
        TreeNode<K,V> p = this;
        do  {
            int ph, dir; K pk; TreeNode<K,V> q;
            // 声明左子树和右子数
            TreeNode<K,V> pl = p.left, pr = p.right;
            // 直接比较hash值，来决决定走左子树还是右子数
            if ((ph = p.hash) > h)
                p = pl;
            else if (ph < h)
                p = pr;
            // 判断当前的子树是否和查询的k == 或者equals，直接返回
            else if ((pk = p.key) == k || (pk != null && k.equals(pk)))
                return p;
            else if (pl == null)
                p = pr;
            else if (pr == null)
                p = pl;
            else if ((kc != null ||
                      (kc = comparableClassFor(k)) != null) &&
                     (dir = compareComparables(kc, k, pk)) != 0)
                p = (dir < 0) ? pl : pr;
            // 递归继续往底层找
            else if ((q = pr.findTreeNode(h, k, kc)) != null)
                return q;
            else
                p = pl;
        } while (p != null);
    }
    return null;
}
```

1.6 ConcurrentHashMap其他方法

1.6.1 compute方法

修改ConcurrentHashMap中指定key的value时，一般会选择先get出来，然后再拿到原value值，基于原value值做一些修改，最后再存放到咱们ConcurrentHashMap

```
public static void main(String[] args) {
    ConcurrentHashMap<String,Integer> map = new ConcurrentHashMap();
```

```
    map.put("key",1);
    // 修改key对应的value，追加上1

    // 之前的操作方式
    Integer oldValue = (Integer) map.get("key");
    Integer newValue = oldValue + 1;
    map.put("key",newValue);
    System.out.println(map);

    // 现在的操作方式
    map.compute("key",(key,computeOldValue) -> {
        if(computeOldValue == null){
            computeOldValue = 0;
        }
        return computeOldValue + 1;
    });
    System.out.println(map);
}
```

1.6.2 compute方法源码分析

整个流程和putVal方法很类似，但是内部涉及到了占位的情况RESERVED

整个compute方法和putVal的区别就是，compute方法的value需要计算，如果key存在，基于oldValue计算出新结果，如果key不存在，直接基于oldValue为null的情况，去计算新的value。

```
// compute 方法
public V compute(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction) {
    if (key == null || remappingFunction == null)
        throw new NullPointerException();
    // 计算key的hash
    int h = spread(key.hashCode());
    V val = null;
    int delta = 0;
    int binCount = 0;
    // 初始化，桶上赋值，链表插入值，红黑树插入值
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        //  初始化
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 桶上赋值
        else if ((f = tabAt(tab, i = (n - 1) & h)) == null) {
            // 数组指定的索引位置是没有数据，当前数据必然要放到数组上。
            // 因为value需要计算得到，计算的时间不可估计，所以这里并没有通过CAS的方式处理并
发操作，直接添加临时占用节点，
            // 并占用当前临时节点的锁资源。
            Node<K,V> r = new ReservationNode<K,V>();
            synchronized (r) {
                // 以CAS的方式将数据放上去
                if (casTabAt(tab, i, null, r)) {
                    binCount = 1;
                    Node<K,V> node = null;
                    try {
                        // 如果ReservationNode临时Node存放成功，直接开始计算value
```

```
                        if ((val = remappingFunction.apply(key, null)) != null)
{
                            delta = 1;
                            // 将计算的value和传入的key封装成一个新Node，通过CAS存储到当
前数组上
                            node = new Node<K,V>(h, key, val, null);
                        }
                    } finally {
                        setTabAt(tab, i, node);
                    }
                }
            }
            if (binCount != 0)
                break;
        }
        else {
            // 省略部分代码。主要是针对在链表上的替换、添加，以及在红黑树上的替换、添加
        }
    }
    if (delta != 0)
        addCount((long)delta, binCount);
    return val;
}
```

1.6.3 computeIfPresent、computeIfAbsent、compute区别

compute的BUG，如果在计算结果的函数中，又涉及到了当前的key，会造成死锁问题。

```
public static void main(String[] args) {
    ConcurrentHashMap<String,Integer> map = new ConcurrentHashMap();

    map.compute("key",(k,v) -> {
        return map.compute("key",(key,value) -> {
            return 1111;
        });
    });
    System.out.println(map);
}
```

computeIfPresent和computeIfAbsent其实就是将compute方法拆开成了两个方法

compute会在key不存在时，正常存放结果，如果key存在，就基于oldValue计算newValue

computeIfPresent：要求key在map中必须存在，需要基于oldValue计算newValue

computeIfAbsent：要求key在map中不能存在，必须为null，才会基于函数得到value存储进去

computeIfPresent：

```
// 如果key存在，才执行修改操作
public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction) {
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 如果key不存在，什么事都不做~
        else if ((f = tabAt(tab, i = (n - 1) & h)) == null)
```

```
            break;
        else {
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f, pred = null;; ++binCount) {
                            K ek;
                            // 如果查看到有 == 或者equals的key，就直接修改即可
                            if (e.hash == h &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                val = remappingFunction.apply(key, e.val);
                                if (val != null)
                                    e.val = val;
                                else {
                                    delta = -1;
                                    Node<K,V> en = e.next;
                                    if (pred != null)
                                        pred.next = en;
                                    else
                                        setTabAt(tab, i, en);
                                }
                                break;
                            }
                            pred = e;
                            // 走完链表，还是没找到指定数据，直接break;
                            if ((e = e.next) == null)
                                break;
                        }
                    }
        // 省略部分代码
    return val;
}
```

computeIfAbsent核心位置源码：

```
// key必须不存在才会执行添加操作
public V computeIfAbsent(K key, Function<? super K, ? extends V>
mappingFunction) {
    for (Node<K,V>[] tab = table;;) {
        else if ((f = tabAt(tab, i = (n - 1) & h)) == null) {
            // 如果key不存在，正常添加;
            Node<K,V> r = new ReservationNode<K,V>();
            synchronized (r) {
                if (casTabAt(tab, i, null, r)) {
                    binCount = 1;
                    Node<K,V> node = null;
                    try {
                        if ((val = mappingFunction.apply(key)) != null)
                            node = new Node<K,V>(h, key, val, null);
                    } finally {
                        setTabAt(tab, i, node);
                    }
                }
            }
        }
```

```
        else {
            boolean added = false;
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek; V ev;
                            // 如果key存在，直接break;
                            if (e.hash == h &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                val = e.val;
                                break;
                            }
                            // 如果没有找到一样的key，计算value结果接口
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                if ((val = mappingFunction.apply(key)) != null)
{
                                    added = true;
                                    pred.next = new Node<K,V>(h, key, val,
null);
                                }
                                break;
                            }
                        }
                    }
        // 省略部分代码
    return val;
}
```

1.6.4 replace方法详解

涉及到类似CAS的操作，需要将ConcurrentHashMap的value从val1改为val2的场景就可以使用replace实现。

replace内部要求key必须存在，替换value值之前，要先比较oldValue，只有oldValue一致时，才会完成替换操作。

```
// replace方法调用的replaceNode方法， value：newValue， cv：oldValue
final V replaceNode(Object key, V value, Object cv) {
    int hash = spread(key.hashCode());
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 在数组没有初始化时，或者key不存在时，什么都不干。
        if (tab == null || (n = tab.length) == 0 ||
            (f = tabAt(tab, i = (n - 1) & hash)) == null)
            break;
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            boolean validated = false;
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
```

```
                        validated = true;
                        for (Node<K,V> e = f, pred = null;;) {
                            K ek;
                            // 找到key一致的Node了。
                            if (e.hash == hash && ((ek = e.key) == key || (ek !=
null && key.equals(ek)))) {
                                // 拿到当前节点的原值。
                                V ev = e.val;
                                // 拿oldValue和原值做比较，如果一致，
                                if (cv == null || cv == ev || (ev != null &&
cv.equals(ev))) {
                                    // 可以开始替换
                                    oldVal = ev;
                                    if (value != null)
                                        e.val = value;
                                    else if (pred != null)
                                        pred.next = e.next;
                                    else
                                        setTabAt(tab, i, e.next);
                                }
                                break;
                            }
                            pred = e;
                            if ((e = e.next) == null)
                                break;
                        }
                    }
                    else if (f instanceof TreeBin) {
                        validated = true;
                        TreeBin<K,V> t = (TreeBin<K,V>)f;
                        TreeNode<K,V> r, p;
                        if ((r = t.root) != null &&
                            (p = r.findTreeNode(hash, key, null)) != null) {
                            V pv = p.val;
                            if (cv == null || cv == pv ||
                                (pv != null && cv.equals(pv))) {
                                oldVal = pv;
                                if (value != null)
                                    p.val = value;
                                else if (t.removeTreeNode(p))
                                    setTabAt(tab, i, untreeify(t.first));
                            }
                        }
                    }
                }
            }
            if (validated) {
                if (oldVal != null) {
                    if (value == null)
                        addCount(-1L, -1);
                    return oldVal;
                }
                break;
            }
        }
    }
    return null;
}
```

1.6.5 merge方法详解

merge(key,value,Function<oldValue,value>);

在使用merge时，有三种情况可能发生：

- 如果key不存在，就跟put(key,value);
- 如果key存在，就可以基于Function计算，得到最终结果
  - 结果不为null，将key对应的value，替换为Function的结果
  - 结果为null，删除当前key

分析merge源码

```
public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>
remappingFunction) {
    if (key == null || value == null || remappingFunction == null) throw new
NullPointerException();
    int h = spread(key.hashCode());
    V val = null;
    int delta = 0;
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // key不存在，直接执行正常的添加操作，将value作为值，添加到hashMap
        else if ((f = tabAt(tab, i = (n - 1) & h)) == null) {
            if (casTabAt(tab, i, null, new Node<K,V>(h, key, value, null))) {
                delta = 1;
                val = value;
                break;
            }
        }
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f, pred = null;; ++binCount) {
                            K ek;
                            // 判断链表中，有当前的key
                            if (e.hash == h && ((ek = e.key) == key || (ek !=
null && key.equals(ek)))) {
                                // 基于函数，计算value
                                val = remappingFunction.apply(e.val, value);
                                // 如果计算的value不为null，正常替换
                                if (val != null)
                                    e.val = val;
                                // 计算的value是null，直接让上一个指针指向我的next，绕
过当前节点
                                else {
                                    delta = -1;
                                    Node<K,V> en = e.next;
                                    if (pred != null)
```

```
                                    pred.next = en;
                                else
                                    setTabAt(tab, i, en);
                            }
                            break;
                        }
                        pred = e;
                        if ((e = e.next) == null) {
                            delta = 1;
                            val = value;
                            pred.next =
                                new Node<K,V>(h, key, val, null);
                            break;
                        }
                    }
                }
                else if (f instanceof TreeBin) {
                    binCount = 2;
                    TreeBin<K,V> t = (TreeBin<K,V>)f;
                    TreeNode<K,V> r = t.root;
                    TreeNode<K,V> p = (r == null) ? null :
                        r.findTreeNode(h, key, null);
                    val = (p == null) ? value :
                        remappingFunction.apply(p.val, value);
                    if (val != null) {
                        if (p != null)
                            p.val = val;
                        else {
                            delta = 1;
                            t.putTreeVal(h, key, val);
                        }
                    }
                    else if (p != null) {
                        delta = -1;
                        if (t.removeTreeNode(p))
                            setTabAt(tab, i, untreeify(t.first));
                    }
                }
            }
        }
        if (binCount != 0) {
            if (binCount >= TREEIFY_THRESHOLD)
                treeifyBin(tab, i);
            break;
        }
    }
}
if (delta != 0)
    addCount((long)delta, binCount);
return val;
}
```

1.7 ConcurrentHashMap计数器

1.7.1 addCount方法分析

addCount方法本身就是为了记录ConcurrentHashMap中元素的个数。

两个方向组成：

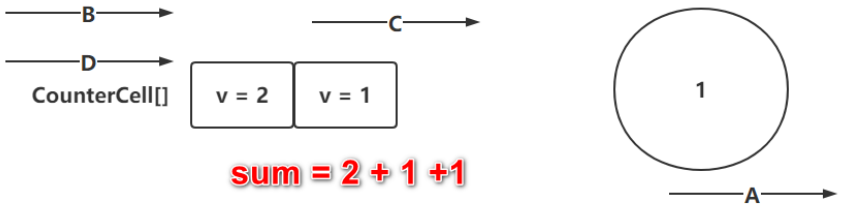
- 计数器，如果添加元素成功，对计数器 + 1
- 检验当前ConcurrentHashMap是否需要扩容

计数器选择的不是AtomicLong，而是类似LongAdder的一个功能

ConcurrentHashMap在计数时，需要保证线程安全，同时还要保证效率

AtomicLong：基于CAS操作实现的计数器，可以保证线程安全
并发比较高的情况下，多个线程同时CAS，只会有一个成功，没有成功的线程会一直执行CAS直到成功为止。

LongAdder：如果并发执行自增操作时，CAS失败了，会将数据单独的存储到一个数组中计数。



addCount源码分析

```
private final void addCount(long x, int check) {
    // ================================计数=====================================
    // as:  CounterCell[]
    // s：是自增后的元素个数
    // b：原来的baseCount
    CounterCell[] as; long b, s;
    // 判断CounterCell不为null，代表之前有冲突问题，有冲突直接进到if中
    // 如果CounterCell[]为null，直接执行||后面的CAS操作，直接修改baseCount
    if ((as = counterCells) != null ||
        // 如果对baseCount++成功。直接告辞。 如果CAS失败，直接进到if中
        !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
        // 导致，说明有并发问题。
        // 进来的方式有两种：
        // 1. counterCell[] 有值。
        // 2. counterCell[] 无值，但是CAS失败。
        // m：数组长度 - 1
        // a：当前线程基于随机数，获得到的数组上的某一个CounterCell
        CounterCell a; long v; int m;
        // 是否有冲突，默认为true，代表没有冲突
        boolean uncontended = true;
        // 判断CounterCell[]没有初始化，执行fullAddCount方法，初始化数组
        if (as == null || (m = as.length - 1) < 0 ||
            // CounterCell[]已经初始化了，基于随机数拿到数组上的一个CounterCell，如果为
null，执行fullAddCount方法，初始化CounterCell
            (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
            // CounterCell[]已经初始化了，并且指定索引位置上有CounterCell
            // 直接CAS修改指定的CounterCell上的value即可。
            // CAS成功，直接告辞！
            // CAS失败，代表有冲突，uncontended = false，执行fullAddCount方法
```

```
            !(uncontended = U.compareAndSwapLong(a, CELLVALUE, v = a.value, v +
x))) {
            fullAddCount(x, uncontended);
            return;
        }
        // 如果链表长度小于等于1，不去判断扩容
        if (check <= 1)
            return;
        // 将所有CounterCell中记录的信累加，得到最终的元素个数
        s = sumCount();
    }

    // ==================================判断扩容
=======================================
    // 判断check大于等于，remove的操作就是小于0的。 因为添加时，才需要去判断是否需要扩容
    if (check >= 0) {
        // 一堆小变量
        Node<K,V>[] tab, nt; int n, sc;
        // 当前元素个数是否大于扩容阈值，并且数组不为null，数组长度没有达到最大值。
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            // 扩容表示戳
            int rs = resizeStamp(n);
            // 正在扩容
            if (sc < 0) {
                // 判断是否可以协助扩容
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
                    transferIndex <= 0)
                    break;
                // 协助扩容
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                    transfer(tab, nt);
            }
            // 没有线程执行扩容，我来扩容
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHIFT) + 2))
                transfer(tab, null);
            // 重新计数。
            s = sumCount();
        }
    }
}


//  CounterCell的类，就类似于LongAdder的Cell
@sun.misc.Contended static final class CounterCell {
    // volatile修饰的value，并且外部基于CAS的方式修改
    volatile long value;
    CounterCell(long x) { value = x; }
}
```

@sun.misc.Contended（JDK1.8）：

这个注解是为了解决伪共享的问题（解决缓存行同步带来的性能问题）。

CPU在操作主内存变量前，会将主内存数据缓存到CPU缓存（L1,L2,L3）中，

CPU缓存L1，是以缓存行为单位存储数据的，一般默认的大小为64字节。

缓存行同步操作，影响CPU一定的性能。

@Contented注解，会将当前类中的属性，会独占一个缓存行，从而避免缓存行失效造成的性能问题。

@Contented注解，就是将一个缓存行的后面7个位置，填充上7个没有意义的数据。

```
long value;    long l1,l2,l3,l4,l5,l6,l7;


// 整体CounterCell数组数据到baseCount
final long sumCount() {
    // 拿到CounterCell[]
    CounterCell[] as = counterCells; CounterCell a;
    // 拿到baseCount
    long sum = baseCount;
    // 循环走你，遍历CounterCell[]，将值累加到sum中，最终返回sum
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}


// CounterCell数组没有初始化
// CounterCell对象没有构建
// 什么都有，但是有并发问题，导致CAS失败
private final void fullAddCount(long x, boolean wasUncontended) {
    // h：当前线程的随机数
    int h;
    // 判断当前线程的Probe是否初始化。
    if ((h = ThreadLocalRandom.getProbe()) == 0) {
        // 初始化一波
        ThreadLocalRandom.localInit();
        // 生成随机数。
        h = ThreadLocalRandom.getProbe();
        // 标记，没有冲突
        wasUncontended = true;
    }
    // 阿巴阿巴
    boolean collide = false;
    // 死循环.....
    for (;;) {
        // as：CounterCell[]
        // a：CounterCell对 null
        // n：数组长度
        // v：value值
        CounterCell[] as; CounterCell a; int n; long v;
        // CounterCell[]不为null时，做CAS操作
        if ((as = counterCells) != null && (n = as.length) > 0) {
            // 拿到当前线程随机数对应的CounterCell对象，为null
            // 第一个if：当前数组已经初始化，但是指定索引位置没有CounterCell对象，构建
CounterCell对象放到数组上
            if ((a = as[h & (n - 1)]) == null) {
                // 判断cellsBusy是否为0，
                if (cellsBusy == 0) {
                    // 构建CounterCell对象
                    CounterCell r = new CounterCell(x);
                    // 在此判断cellsBusy为0，CAS从0修改为1，代表可以操作当前数组上的指定
索引，构建CounterCell，赋值进去
                    if (cellsBusy == 0 &&
                        U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
```

```
                        // 构建未完成
                        boolean created = false;
                        try {
                            // 阿巴阿巴
                            CounterCell[] rs; int m, j;
                            // DCL，还包含复制
                            if ((rs = counterCells) != null && (m = rs.length) >
0 &&
                                // 再次拿到指定索引位置的值，如果为null，正常将前面构建的
CounterCell对象，赋值给数组
                                rs[j = (m - 1) & h] == null) {
                                // 将CounterCell对象赋值到数组
                                rs[j] = r;
                                // 构建完成
                                created = true;
                            }
                        } finally {
                            // 归位
                            cellsBusy = 0;
                        }
                        if (created)
                            // 跳出循环，告辞
                            break;
                        continue;           // Slot is now non-empty
                    }
                }
                collide = false;
            }
            // 指定索引位置上有CounterCell对象，有冲突，修改冲突标识
            else if (!wasUncontended)
                wasUncontended = true;
            // CAS，将数组上存在的CounterCell对象的value进行 + 1操作
            else if (U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))
                // 成功，告辞。
                break;
            // 之前拿到的数组引用和成员变量的引用值不一样了，
            // CounterCell数组的长度是都大于CPU内核数，不让CounterCell数组长度大于CPU内
核数。
            else if (counterCells != as || n >= NCPU)
                // 当前线程的循环失败，不进行扩容
                collide = false;
            // 如果没并发问题，并且可以扩容，设置标示位，下次扩容
            else if (!collide)
                collide = true;
            // 扩容操作
            // 先判断cellsBusy为0，再基于CAS将cellsBusy从0修改为1。
            else if (cellsBusy == 0 &&
                     U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
                try {
                    // DCL!
                    if (counterCells == as) {
                        // 构建一个原来长度2倍的数组
                        CounterCell[] rs = new CounterCell[n << 1];
                        // 将老数组数据迁移到新数组
                        for (int i = 0; i < n; ++i)
                            rs[i] = as[i];
                        // 新数组复制给成员变量
                        counterCells = rs;
```

```
                    }
                } finally {
                    // 归位
                    cellsBusy = 0;
                }
                // 归位
                collide = false;
                // 开启下次循环
                continue;
            }
            // 重新设置当前线程的随机数，争取下次循环成功!
            h = ThreadLocalRandom.advanceProbe(h);
        }
        // CounterCell[]没有初始化
        // 判断cellsBusy为0.代表没有其他线程在初始化或者扩容当前CounterCell[]
        // 判断counterCells还是之前赋值的as，代表没有并发问题
        else if (cellsBusy == 0 && counterCells == as &&
            // 修改cellsBusy，从0改为1，代表当前线程要开始初始化了
            U.compareAndSwapInt(this, CELLSBUSY, 0, 1)) {
            // 标识，init未成功
            boolean init = false;
            try {
                // DCL!
                if (counterCells == as) {
                    // 构建CounterCell[]，默认长度为2
                    CounterCell[] rs = new CounterCell[2];
                    // 用当前线程的随机数，和数组长度 - 1，进行&运算，将这个位置上构建一个
CounterCell对象，赋值value为1
                    rs[h & 1] = new CounterCell(x);
                    // 将声明好的rs，赋值给成员变量
                    counterCells = rs;
                    // init成功
                    init = true;
                }
            } finally {
                // cellsBusy归位。
                cellsBusy = 0;
            }
            if (init)
                // 退出循环
                break;
        }
        // 到这就直接在此操作baseCount。
        else if (U.compareAndSwapLong(this, BASECOUNT, v = baseCount, v + x))
            break;                          // Fall back on using base
    }
}
```

1.7.2 size方法方法分析

size获取ConcurrentHashMap中的元素个数

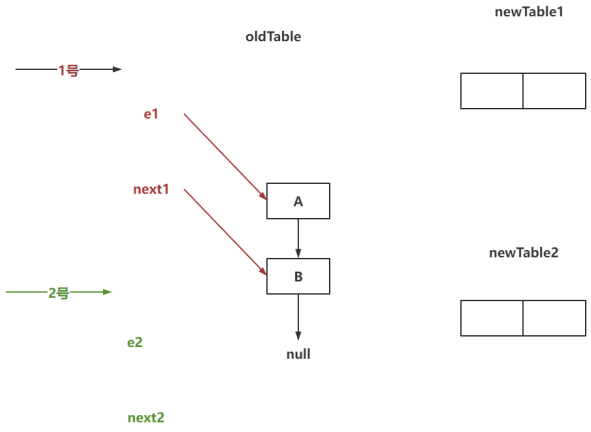
```
public int size() {
    // 基于sumCount方法获取元素个数
    long n = sumCount();
    // 做了一些简单的健壮性判断
    return ((n < 0L) ? 0 :
```

```
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}

// 整体CounterCell数组数据到baseCount
final long sumCount() {
    // 拿到CounterCell[]
    CounterCell[] as = counterCells; CounterCell a;
    // 拿到baseCount
    long sum = baseCount;
    // 循环走你，遍历CounterCell[]，将值累加到sum中，最终返回sum
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

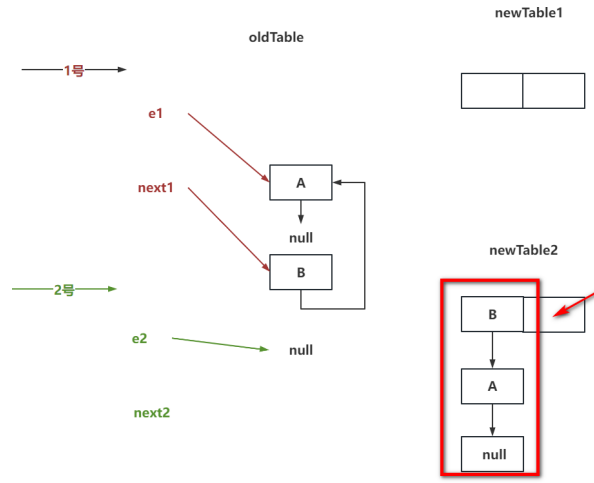
JDK1.7的HashMap的环形链表

分析扩容的核心代码

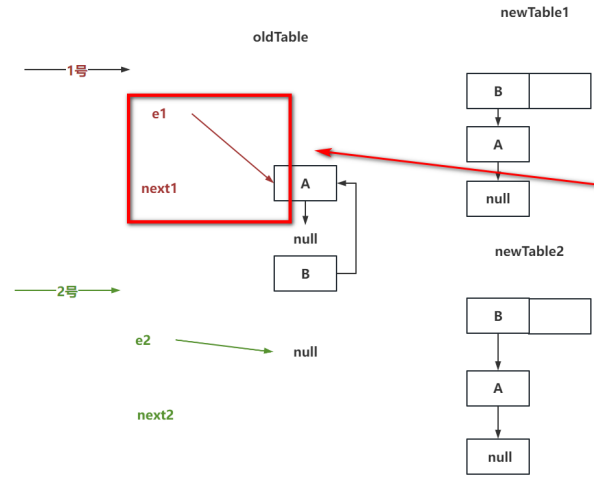
```
// 构建新数组
Entry[] newTable = new Entry[newCapacity];
// 迁移老数组数据到新数组
transfer(newTable, initHashSeedAsNeeded(newCapacity));
// 迁移完毕后，替换老数组
table = newTable;

// 迁移数据的过程
void transfer(Entry[] newTable, boolean rehash) {
    // 省略部分代码
    // 外层遍历数组
    for (Entry<K,V> e : table) {
        // 遍历链表
        // 2号线程走完第二次循环，完成迁移数据（步骤二图）
        // 1号线程走完第二次循环，发现指向的是A（步骤三图）
        // 1号线程走完第三次循环，完成迁移数据（步骤四图）
        while(null != e) {
            Entry<K,V> next = e.next;
            // 1号线程执行到这，停止（步骤一图）
            // 省略部分代码
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
// 唤醒链表的发生，是因为并发扩容，加上头插法导致的。
// 在JDK1.8中，头插法被替代，换成了尾插法
```



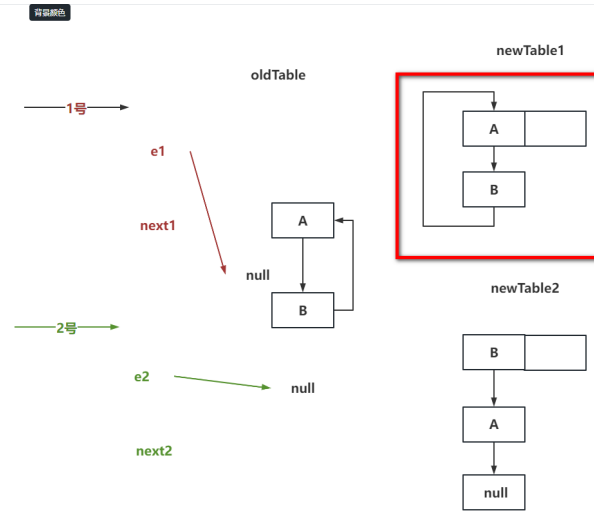

```java
// 迁移完毕后，替换老数组
table = newTable;

// 迁移数据的过程
void transfer(Entry[] newTable, boolean rehash) {
    // 省略部分代码
    // 外层遍历数组
    for (Entry<K,V> e : table) {
        // 遍历链表
        while(null != e) {
            Entry<K,V> next = e.next;
            // 1号线程执行到这，停止
            // 省略部分代码
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```




```java
transfer(newTable, initHashSeedAsNeeded(newCapaci
// 迁移完毕后，替换老数组
table = newTable;

// 迁移数据的过程
void transfer(Entry[] newTable, boolean rehash) {
    // 省略部分代码
    // 外层遍历数组
    for (Entry<K,V> e : table) {
        // 遍历链表
        // 2号线程走完第二次循环，完成迁移数据
        while(null != e) {
            Entry<K,V> next = e.next;
            // 1号线程执行到这，停止
            // 省略部分代码
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```




```java
transfer(newTable, initHashSeedAsNeeded(newCapac
// 迁移完毕后，替换老数组
table = newTable;

// 迁移数据的过程
void transfer(Entry[] newTable, boolean rehash)
    // 省略部分代码
    // 外层遍历数组
    for (Entry<K,V> e : table) {
        // 遍历链表
        // 2号线程走完第二次循环，完成迁移数据
        // 1号线程走完第二次循环，发现指向的是A
        while(null != e) {
            Entry<K,V> next = e.next;
            // 1号线程执行到这，停止
            // 省略部分代码
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```




```java
transfer(newTable, initHashSeedAsNeeded(newCapaci
// 迁移完毕后，替换老数组
table = newTable;

// 迁移数据的过程
void transfer(Entry[] newTable, boolean rehash) {
    // 省略部分代码
    // 外层遍历数组
    for (Entry<K,V> e : table) {
        // 遍历链表
        // 2号线程走完第二次循环，完成迁移数据
        // 1号线程走完第二次循环，发现指向的是A
        // 1号线程走完第三次循环，完成迁移数据
        while(null != e) {
            Entry<K,V> next = e.next;
            // 1号线程执行到这，停止
            // 省略部分代码
            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```

如果面试被问到了：

因为JDK1.7中的HashMap是线程不安全的，可能会出现并发扩容的操作。

同时JDK1.7中的HashMap在迁移数据时，采用的是头插法，导致节点的next指针会有变化。

先迁移完的线程，可能会导致其他线程在扩容时，扩容到最后，将最开始的节点重新的插入到了头节点的位置，导致指针再次变化，从而形成了一个环形链表。

二、CopyOnWriteArrayList

2.1 CopyOnWriteArrayList介绍

CopyOnWriteArrayList是一个线程安全的ArrayList。

CopyOnWriteArrayList是基于lock锁和数组副本的形式去保证线程安全。

在写数据时，需要先获取lock锁，需要复制一个副本数组，将数据插入到副本数组中，将副本数组赋值给CopyOnWriteArrayList中的array。

因为CopyOnWriteArrayList每次写数据都要构建一个副本，如果你的业务是写多，并且数组中的数据量比较大，尽量避免去使用CopyOnWriteArrayList，因为这里会构建大量的数组副本，比较占用内存资源。

CopyOnWriteArrayList是弱一致性的，写操作先执行，但是副本还有落到CopyOnWriteArrayList的array属性中，此时读操作是无法查询到的。

2.2 核心属性&方法

主要查看2个核心属性，以及2个核心方法，还有无参构造

```
/** 写操作时，需要先获取到的锁资源，CopyOnWriteArrayList全局唯一的。 */
final transient ReentrantLock lock = new ReentrantLock();

/** CopyOnWriteArrayList真实存放数据的位置，查询也是查询当前array */
private transient volatile Object[] array;

// 获取array属性
final Object[] getArray() {
    return array;
}

// 替换array属性
final void setArray(Object[] a) {
    array = a;
}

/**
 *  默认new的CopyOnWriteArrayList数组长度为0。
 *  不像ArrayList，初始长度是10，每次扩容1/2，CopyOnWriteArrayList不存在这个概念
 *  每次写的时候都会构建一个新的数组
 */
public CopyOnWriteArrayList() {
    setArray(new Object[0]);
}
```

2.3 读操作

CopyOnWriteArrayList的读操作就是get方法，基于数组索引位置获取数据。

方法之所以要差分成两个，是因为CopyOnWriteArrayList中在获取数据时，不单单只有一个array的数组需要获取值，还有副本中数据的值。

```
// 查询数据时，只能通过get方法查询CopyOnWriteArrayList中的数据
public E get(int index) {
    // getArray拿到array数组，调用get方法的重载
    return get(getArray(), index);
}
// 执行get(int)时，内部调用的方法
private E get(Object[] a, int index) {
    // 直接拿到数组上指定索引位置的值
    return (E) a[index];
}
```

2.4 写操作

CopyOnWriteArrayList是基于lock锁和副本数组的形式保证线程安全。

```
// 写入元素，不指定索引位置，直接放到最后的位置
public boolean add(E e) {
    // 获取全局锁，并执行lock
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        // 获取原数组，还获取了原数组的长度
        Object[] elements = getArray();
        int len = elements.length;
        // 基于原数组复制一份副本数组，并且长度比原来多了一个
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        // 将添加的数据放到副本数组最后一个位置
        newElements[len] = e;
        // 将副本数组，赋值给CopyOnWriteArrayList的原数组
        setArray(newElements);
        // 添加成功，返回true
        return true;
    } finally {
        // 释放锁~
        lock.unlock();
    }
}

// 写入元素，指定索引位置。（不会覆盖数据）
public void add(int index, E element) {
    // 拿锁，加锁~
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        // 获取原数组，还获取了原数组的长度
        Object[] elements = getArray();
        int len = elements.length;
        // 如果索引位置大于原数组的长度，或者索引位置是小于0的。
        if (index > len || index < 0)
```

```
            throw new IndexOutOfBoundsException("Index: "+index+
                                                ", Size: "+len);
        // 声明了副本数组
        Object[] newElements;
        // 原数组长度 - 索引位置等到numMoved
        int numMoved = len - index;
        // 如果numMoved为0，说明数据要放到最后面的位置
        if (numMoved == 0)
            // 直接走了原生态的方式，正常复制一份副本数组
            newElements = Arrays.copyOf(elements, len + 1);
        else {
            // 数组要插入的位置不是最后一个位置
            // 副本数组长度依然是原长度 + 1
            newElements = new Object[len + 1];
            // 将原数组从0索引位置开始复制，复制到副本数组中的前置位置
            System.arraycopy(elements, 0, newElements, 0, index);
            // 将原数组从index位置开始复制，复制到副本数组的index + 1往后放。
            // 这时，index就空缺出来了。
            System.arraycopy(elements, index, newElements, index + 1,
                             numMoved);
        }
        // 数据正常放到指定的索引位置
        newElements[index] = element;
        // 将副本数组，赋值给CopyOnWriteArrayList的原数组
        setArray(newElements);
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

2.5 移除数据

关于remove操作，要分析两个方法

- 基于索引位置移除指定数据
- 基于具体元素删除数组中最靠前的数据
    - 当前这种方式，嵌套了一层，导致如果元素存在话，成本是比较高的。
    - 如果元素不存在，这种设计不需要加锁，提升写的效率

```
// 删除指定索引位置的数据
public E remove(int index) {
    // 拿锁，加锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        // 获取原数组和原数组长度
        Object[] elements = getArray();
        int len = elements.length;
        // 通过get方法拿到index位置的数据
        E oldValue = get(elements, index);
        // 声明numMoved
        int numMoved = len - index - 1;
        // 如果numMoved为0，说明删除的元素是最后的位置
        if (numMoved == 0)
            // Arrays.copyOf复制一份新的副本数组，并且将最后一个数据不要了
```

```
                // 基于setArray将副本数组赋值给array原数组
                setArray(Arrays.copyOf(elements, len - 1));
            else {
                // 删除的元素不在最后面的位置
                // 声明副本数组，长度是原数组长度 - 1
                Object[] newElements = new Object[len - 1];
                // 从0开始复制的index前面
                System.arraycopy(elements, 0, newElements, 0, index);
                // 从index后面复制到最后
                System.arraycopy(elements, index + 1, newElements, index,
                                 numMoved);
                setArray(newElements);
            }
            // 返回被干掉的数据
            return oldValue;
        } finally {
            // 释放锁
            lock.unlock();
        }
    }

    // 删除元素（最前面的）
    public boolean remove(Object o) {
        // 没加锁！！！！
        // 获取原数组
        Object[] snapshot = getArray();
        // 用indexOf获取元素在数组的哪个索引位置
        // 没找到的话，返回-1
        int index = indexOf(o, snapshot, 0, snapshot.length);
        // 如果index < 0,说明元素没找到，直接返回false，告辞
        // 如果找到了元素的位置，直接执行remove方法的重载
        return (index < 0) ? false : remove(o, snapshot, index);
    }
    // 执行remove(Object o)，找到元素位置时，执行当前方法
    private boolean remove(Object o, Object[] snapshot, int index) {
        // 拿锁，加锁
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            // 拿到原数组和长度
            Object[] current = getArray();
            int len = current.length;
            // findIndex: 是给if起标识，break 标识的时候，直接跳出if的代码块~~
            if (snapshot != current) findIndex: {
                // 如果没进到if，说明数组没变化，按照原来的index位置删除即可
                // 进到这，说明数组有变化，之前的索引位置不一定对
                // 拿到index位置和原数组长度的值
                int prefix = Math.min(index, len);
                // 循环判断，数组变更后，是否影响到了要删除元素的位置
                for (int i = 0; i < prefix; i++) {
                    // 如果不相等，说明元素变化了。
                    // 同时判断变化的元素是否是我要删除的元素o
                    if (current[i] != snapshot[i] && eq(o, current[i])) {
                        // 如果满足条件，说明当前位置就是我要删除的元素
                        index = i;
                        break findIndex;
                    }
                }
```

```
                // 如果for循环结束，没有退出if，说明元素可能变化了，总之没找到要删除的元素
                // 如果删除元素的位置，已经大于等于数组长度了。
                if (index >= len)
                    // 超过索引范围了，没法删除了。
                    return false;
                // 索引还在范围内，判断是否是还是原位置，如果是，直接跳出if代码块
                if (current[index] == o)
                    break findIndex;
                // 重新找元素在数组中的位置
                index = indexOf(o, current, index, len);
                // 找到直接跳出if代码块
                // 没找到。执行下面的return false
                if (index < 0)
                    return false;
            }
            // 删除套路，构建新数组，复制index前的，复制index后的
            Object[] newElements = new Object[len - 1];
            System.arraycopy(current, 0, newElements, 0, index);
            System.arraycopy(current, index + 1,
                             newElements, index,
                             len - index - 1);
            // 复制到array
            setArray(newElements);
            // 返回true，删除成功
            return true;
        } finally {
            lock.unlock();
        }
    }
```

2.6 覆盖数据&清空集合

覆盖数据就是set方法，可以将指定位置的数据替换

```
    // 覆盖数据
    public E set(int index, E element) {
        // 拿锁，加锁
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            // 获取原数组
            Object[] elements = getArray();
            // 获取原数组的原位置数据
            E oldValue = get(elements, index);

            // 原数据和新数据不一样
            if (oldValue != element) {
                // 拿到原数据的长度，复制一份副本。
                int len = elements.length;
                Object[] newElements = Arrays.copyOf(elements, len);
                // 将element替换掉副本数组中的数据
                newElements[index] = element;
                // 写回原数组
                setArray(newElements);
            } else {
                // 原数据和新数据一样，啥不干，把拿出来的数组再写回去
                setArray(elements);
```

```
            }
            // 返回原值
            return oldValue;
        } finally {
            // 释放锁
            lock.unlock();
        }
    }
```

清空就是清空了~~~

```
public void clear() {
    // 加锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        // 扔一个长度为0的数组
        setArray(new Object[0]);
    } finally {
        lock.unlock();
    }
}
```

2.7 迭代器

用ArrayList时，如果想在遍历的过程中去移除或者修改元素，必须使用迭代器才可以。

但是CopyOnWriteArrayList这哥们即便用了迭代器也不让做写操作

不让在迭代时做写操作是因为不希望迭代操作时，会影响到写操作，还有，不希望迭代时，还需要加锁。

```
// 获取遍历CopyOnWriteArrayList的iterator。
public Iterator<E> iterator() {
    // 其实就是new了一个COWIterator对象，并且获取了array，指定从0开始遍历
    return new COWIterator<E>(getArray(), 0);
}

static final class COWIterator<E> implements ListIterator<E> {
    /** 遍历的快照 */
    private final Object[] snapshot;
    /** 游标，索引~~~ */
    private int cursor;

    // 有参构造
    private COWIterator(Object[] elements, int initialCursor) {
        cursor = initialCursor;
        snapshot = elements;
    }

    // 有没有下一个元素，基于遍历的索引位置和数组长度查看
    public boolean hasNext() {
        return cursor < snapshot.length;
    }
    // 有没有上一个元素
    public boolean hasPrevious() {
        return cursor > 0;
```

```
    }

    // 获取下一个值，游标动一下
    public E next() {
        // 确保下个位置有数据
        if (! hasNext())
            throw new NoSuchElementException();
        return (E) snapshot[cursor++];
    }

    // 获取上一个值，游标往上移动
    public E previous() {
        if (! hasPrevious())
            throw new NoSuchElementException();
        return (E) snapshot[--cursor];
    }

    // 拿到下一个值的索引，返回游标
    public int nextIndex() {
        return cursor;
    }

    // 拿到上一个值的索引，返回游标
    public int previousIndex() {
        return cursor-1;
    }

    // 写操作全面禁止！！
    public void remove() {
        throw new UnsupportedOperationException();
    }


    public void set(E e) {
        throw new UnsupportedOperationException();
    }


    public void add(E e) {
        throw new UnsupportedOperationException();
    }

    // 兼容函数式编程
    @Override
    public void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        Object[] elements = snapshot;
        final int size = elements.length;
        for (int i = cursor; i < size; i++) {
            @SuppressWarnings("unchecked") E e = (E) elements[i];
            action.accept(e);
        }
        cursor = size;
    }
}
```