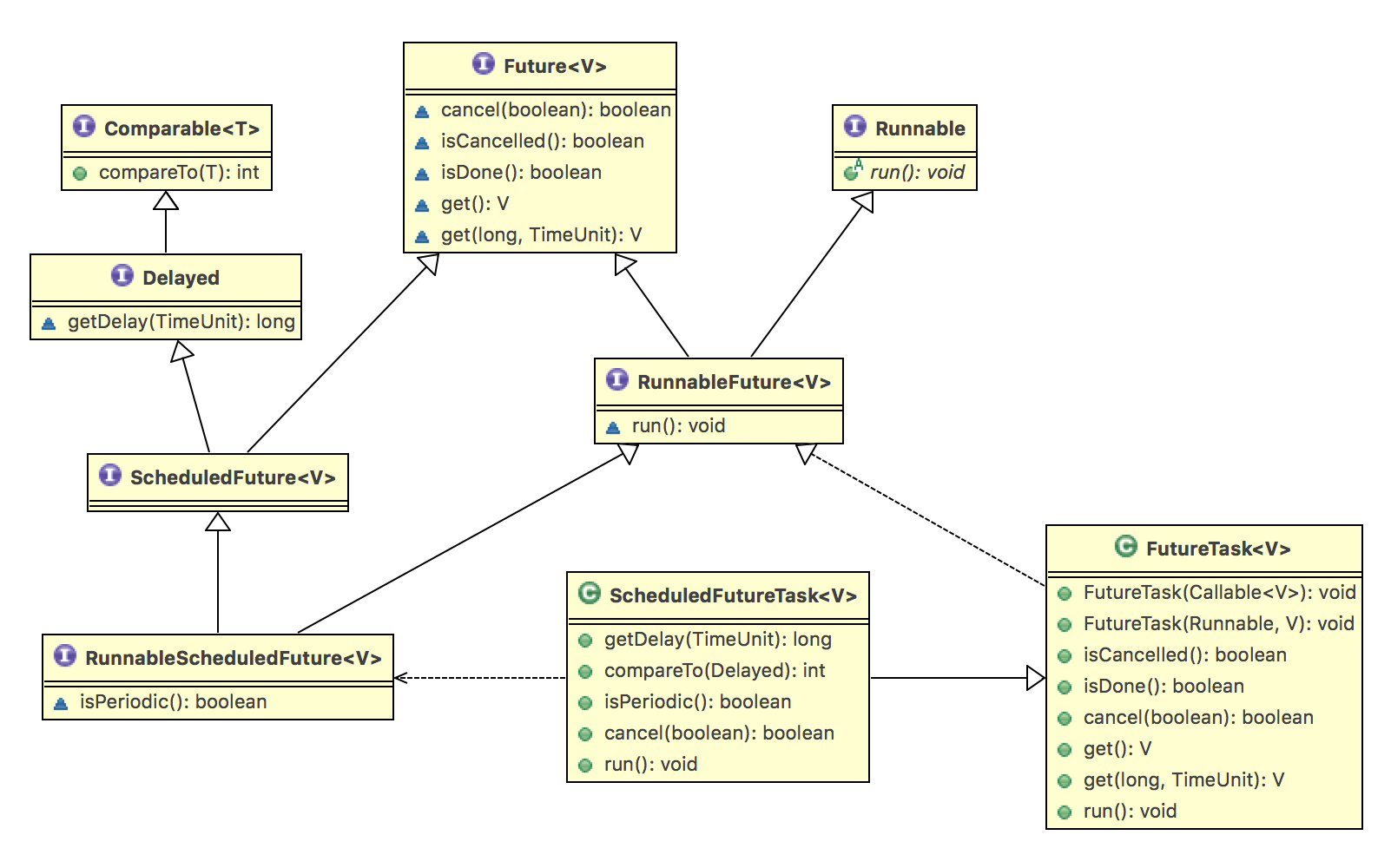
线程池之ScheduledThreadPoolExecutor调度原理

ScheduledThreadPoolExecutor的调度原理主要基于两个内部类，ScheduledFutureTask和DelayedWorkQueue：

1. ScheduledFutureTask是对任务的一层封装，将我们提交的Runnable或Callable封装成具有时间周期的任务；
2. DelayedWorkQueue实现了对ScheduledFutureTask的延迟出队管理；

ScheduledFutureTask



ScheduledFutureTask有以下几种构造方法：

ScheduledFutureTask(Runnable r, V result, long ns) {

super(r, result);

this.time = ns;

this.period = 0;

this.sequenceNumber = sequencer.getAndIncrement();

}

ScheduledFutureTask(Runnable r, V result, long ns, long period) {

super(r, result);

this.time = ns;

this.period = period;

this.sequenceNumber = sequencer.getAndIncrement();

}

ScheduledFutureTask(Callable<V> callable, long ns) {

super(callable);

this.time = ns;

this.period = 0;

this.sequenceNumber = sequencer.getAndIncrement();

}

super中调用FutureTask的构造方法，可以参考《FutureTask实现原理》。ScheduledFutureTask主要配置参数如下：

|  |  |
| --- | --- |
| 名称 | 含义 |
| time | 任务能够执行的时间点（单位: nanoTime） |
| period | 正值表示固定时间周期执行。  负值表示固定延迟周期执行。  0表示非重复任务。 |
| sequenceNumber | FIFO调度序列值（用AtomicLong实现） |

注意：period大于0或小于0时，都是周期性执行的，只是执行时间规律不一样。

ScheduledFutureTask的主要调度辅助方法如下：

public long getDelay(TimeUnit unit) {// 任务的延迟执行时间

return unit.convert(time - now(), NANOSECONDS);

}

public int compareTo(Delayed other) {//实现任务的排序，执行时间越小越靠前，相同则按照队列FIFO顺序

if (other == this) return 0; // compare zero if same object

if (other instanceof ScheduledFutureTask) {

ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;

long diff = time - x.time;

if (diff < 0) return -1;

else if (diff > 0) return 1;

else if (sequenceNumber < x.sequenceNumber) return -1; // 时间一样时，按照FIFO的顺序

else return 1;

}

long diff = getDelay(NANOSECONDS) - other.getDelay(NANOSECONDS);

return (diff < 0) ? -1 : (diff > 0) ? 1 : 0;

}

public boolean isPeriodic() { return period != 0;// 是否是周期性任务 }

private void setNextRunTime() { // 设置下一次运行时间

long p = period;

if (p > 0) time += p; // 按固定时间周期，下次执行时间为上次执行时间 + 周期时间

else time = triggerTime(-p); // 按固定延时周期，下次执行时间为当前时间 + 延时时间

}

核心run方法

public void run() {

boolean periodic = isPeriodic();

if (!canRunInCurrentRunState(periodic)) // 判断是否可以运行任务

cancel(false); // 取消任务，移除队列

else if (!periodic) // 非周期性任务 直接调用父类 FutureTask 的 run 方法

ScheduledFutureTask.super.run();

else if (ScheduledFutureTask.super.runAndReset()) { // 周期性任务，调用父类 runAndReset 方法，返回是否执行成功

// 执行成功后继续设置下一次运行时间

setNextRunTime();

// 重新执行周期性任务（可能因为线程池运行状态的改变而被拒绝）

reExecutePeriodic(outerTask);

}

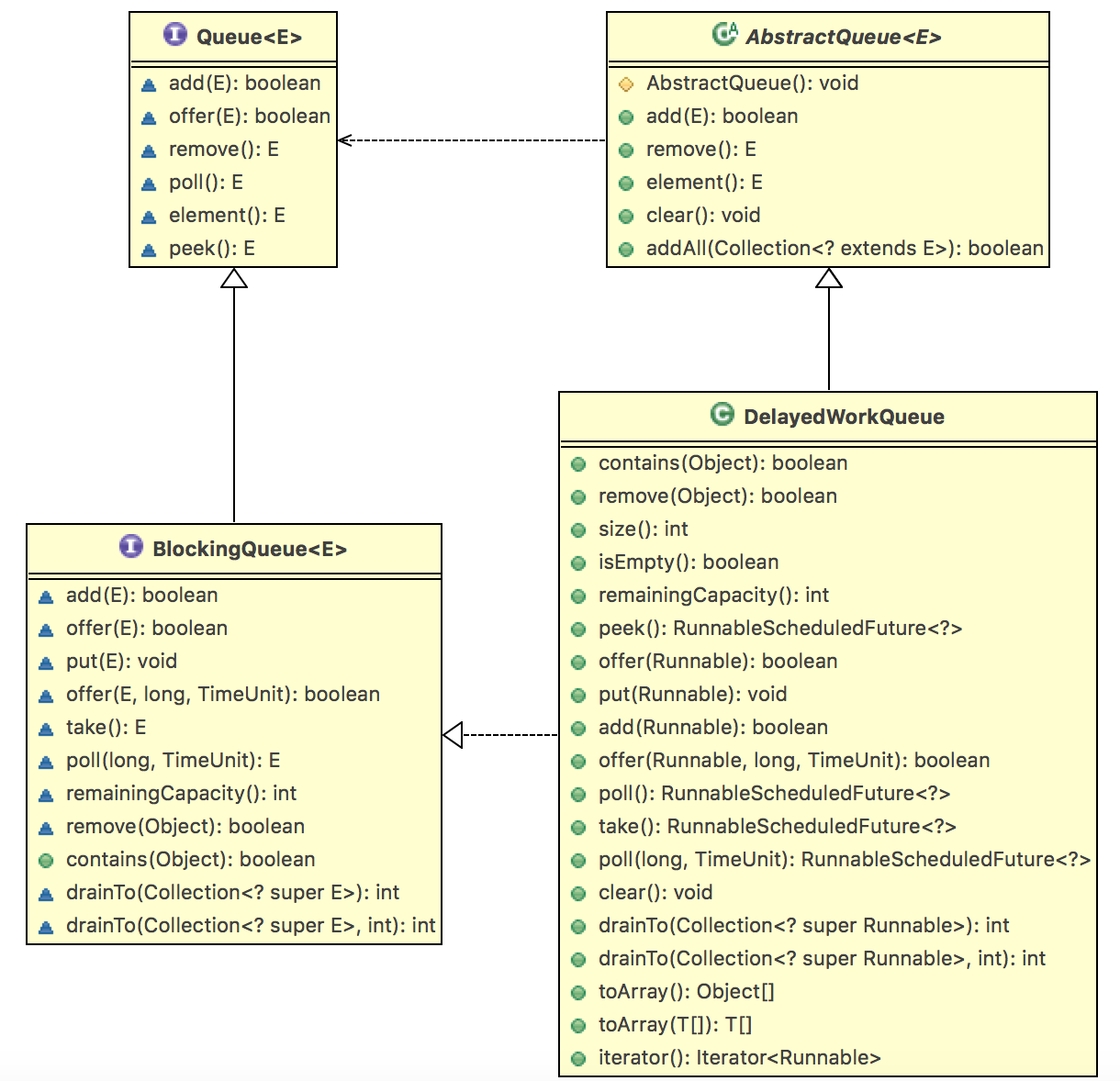
}

对于周期性任务，在run()方法中执行成功后会继续设置下一次执行时间，并把任务加入延时队列。但需注意，如果任务执行失败，将不会再被周期性调用。所以在可能执行失败的周期性任务中，必须做好异常处理。

DelayedWorkQueue

DelayedWorkQueue是一个延时有序队列，内部采用数组维护队列元素，采用堆排序的思想维护队列顺序，并在队列元素（ScheduledFutureTask）建立索引，支持快速删除。

注意：DelayedWorkQueue的整个队列不是完全有序的，只保证元素有序出队。



下面详细讲解DelayedWorkQueue的实现：

核心入队方法：

public boolean add(Runnable e) { return offer(e); }

public boolean offer(Runnable x) {

if (x == null) throw new NullPointerException();

RunnableScheduledFuture<?> e = (RunnableScheduledFuture<?>)x;

final ReentrantLock lock = this.lock;

lock.lock();

try {

int i = size;

if (i >= queue.length) grow(); // 队列扩容 类似 ArrayList 扩容

size = i + 1;

if (i == 0) { // 队列为空，直接加入

queue[0] = e;

setIndex(e, 0); // 设置元素在队列的索引，即告诉元素自己在队列的第几位

} else { siftUp(i, e); // 放入适当的位置 }

if (queue[0] == e) {

leader = null; // 等待队列头的线程

available.signal(); // 通知

}

} finally { lock.unlock(); }

return true;

}

入队方法中最重要的是siftUp方法，sift在英文单词中是”筛”的意思，这里可将siftUp理解为向前筛，找到合适的堆排序点加进去。

private void siftUp(int k, RunnableScheduledFuture<?> key) {

while (k > 0) {

int parent = (k - 1) >>> 1; // (k-1)/2

RunnableScheduledFuture<?> e = queue[parent];

if (key.compareTo(e) >= 0)

break;

queue[k] = e;

setIndex(e, k);

k = parent;

}

queue[k] = key;

setIndex(key, k);

}

siftUp主要思想是将新增的任务与前(k-1)/2的位置比较，如果任务执行时间较近者替换位置(k-1)/2。依次往前比较，直到无替换发生。每次新增元素调用siftUp仅能保证第一个元素是最小的。整个队列不一定有序：

例将：5 10 9 3 依次入队,队列变化如下

[5]

[5,10]

[5,9,10]

[3,5,10,9]

如果对上述的入队方式不了解，可用下面的排序代码进行断点调试：

public class SortArray {// DelayedWorkQueue 的入队、出队排序模拟

Integer[] queue = new Integer[16];

int size = 0;

public static void main(String[] args) {

SortArray array = new SortArray();

array.add(5);

array.add(9);

array.add(10);

array.add(3);

System.out.println(array.take());

System.out.println(array.take());

System.out.println(array.take());

System.out.println(array.take());

}

boolean add(Integer e) {

if (e == null) throw new NullPointerException();

int i = size;

size = i + 1;

if (i == 0) { queue[0] = e;

} else { siftUp(i, e); }

return true;

}

Integer take() {

Integer i = queue[0];

int s = --size;

Integer k = queue[s];

if (size != 0) siftDown(0, k);

return i;

}

private void siftUp(int k, Integer key) {

while (k > 0) {

int parent = (k - 1) >>> 1;

Integer e = queue[parent];

if (key.compareTo(e) >= 0) break;

queue[k] = e;

k = parent;

}

queue[k] = key;

}

private void siftDown(int k, Integer key) {

int half = size >>> 1;

while (k < half) {

int child = (k << 1) + 1;

Integer c = queue[child];

int right = child + 1;

if (right < size && c.compareTo(queue[right]) > 0) c = queue[child = right];

if (key.compareTo(c) <= 0) break;

queue[k] = c;

k = child;

}

queue[k] = key;

}

}

核心出队方法：

public RunnableScheduledFuture<?> take() throws InterruptedException {

final ReentrantLock lock = this.lock;

lock.lockInterruptibly();

try {

for (;;) {

// 直接获取队首任务

RunnableScheduledFuture<?> first = queue[0];

if (first == null) // 空 则等待

available.await();

else {

long delay = first.getDelay(NANOSECONDS); // 看任务是否可以执行

if (delay <= 0)

return finishPoll(first); // 可执行，则进行出队操作

// 可不执行，还需等待，则往下走

first = null;

// 看是否有正在等待的leader线程

if (leader != null)

available.await();

else {

Thread thisThread = Thread.currentThread();

leader = thisThread;

try {

available.awaitNanos(delay); // 延时等待

} finally {

if (leader == thisThread)

leader = null;

}

}

}

}

} finally {

if (leader == null && queue[0] != null)

available.signal();

lock.unlock();

}

}

代码中的available是一个信号量，会在队列的头部由新任务变为可用或者新线程，可能需要称为领导者时，发出信号。

private final Condition available = lock.newCondition();

take()方法中重要的方法是finishPoll( first )，主要进行出队时维护队列顺序：

private RunnableScheduledFuture<?> finishPoll(RunnableScheduledFuture<?> f) {

int s = --size;

RunnableScheduledFuture<?> x = queue[s];

queue[s] = null;

if (s != 0)

siftDown(0, x);

setIndex(f, -1);

return f;

}

private void siftDown(int k, RunnableScheduledFuture<?> key) {

int half = size >>> 1;

while (k < half) {

int child = (k << 1) + 1;

RunnableScheduledFuture<?> c = queue[child];

int right = child + 1;

if (right < size && c.compareTo(queue[right]) > 0)

c = queue[child = right];

if (key.compareTo(c) <= 0)

break;

queue[k] = c;

setIndex(c, k);

k = child;

}

queue[k] = key;

setIndex(key, k);

}

siftDown跟前面的siftUp很像，它也只能保证出队后下一个仍为最近的任务。并不会移动和清理整个队列。

还是用上面列出的SortArray这个类为例：

public static void main(String[] args) {

SortArray array = new SortArray();

array.add(5);

array.add(9);

array.add(10);

array.add(3);

System.out.println(Arrays.toString(array.queue));

System.out.println(array.take());

System.out.println(array.take());

System.out.println(array.take());

System.out.println(array.take());

System.out.println(Arrays.toString(array.queue));

array.add(20);

array.add(4);

System.out.println(Arrays.toString(array.queue));

}

我们先将5,9,10,3依次入队，然后全部出队，再入队20,4，我们看下最后的队列里面的数据是什么样子：

[3, 5, 10, 9, null, null, null, null, null, null, null, null, null, null, null, null]

3

5

9

10

[10, 10, 10, 9, null, null, null, null, null, null, null, null, null, null, null, null]

[4, 20, 10, 9, null, null, null, null, null, null, null, null, null, null, null, null]

看了这个结果你可能有点奇怪，已经出队了的元素居然还在队列里面。这是一种lazy策略，DelayedWorkQueue并不会真正直接清理掉队列里出队的元素，用size来控制队列的逻辑大小，并发物理实际大小，后来的元素会根据size来覆盖原有的元素。

关于DelayedWorkQueue的出队和入队还有疑问的，可以自己调试SortArray的代码，看看不同的情况的不同处理结果。DelayedWorkQueue的siftUp、siftDown这种排序策略非常高效，并非维护整个队列实时有序，只保证第一个出队元素的正确性。

元素删除

上文提到ScheduledFutureTask的索引，DelayedWorkQueue运用索引可以快速定位删除元素：

public boolean remove(Object x) {

final ReentrantLock lock = this.lock;

lock.lock();

try {

int i = indexOf(x);

if (i < 0)

return false;

setIndex(queue[i], -1);

int s = --size;

RunnableScheduledFuture<?> replacement = queue[s];

queue[s] = null;

if (s != i) {

siftDown(i, replacement); // 顺序调整

if (queue[i] == replacement)

siftUp(i, replacement);

}

return true;

} finally {

lock.unlock();

}

}

// 使用索引获取下标

private int indexOf(Object x) {

if (x != null) {

if (x instanceof ScheduledFutureTask) {

int i = ((ScheduledFutureTask) x).heapIndex; // 索引

if (i >= 0 && i < size && queue[i] == x)

return i;

} else {

for (int i = 0; i < size; i++)

if (x.equals(queue[i]))

return i;

}

}

return -1;

}

remove方法里面首先利用indexOf调用索引获取下标，然后使用siftDown、siftUp来调整队列顺序。这里索引的使用能够极大提高元素定位的效率，尤其是在队列比较长的时候。

最后思考一个问题：为什么DelayedWorkQueue使用数组而不是链表结构？

个人认为，因为使用数据结构，利用下标快速访问，可以发挥基于siftDown、siftUp的高效排序算法，而链表的下标访问效率低，因此选择使用数组。