

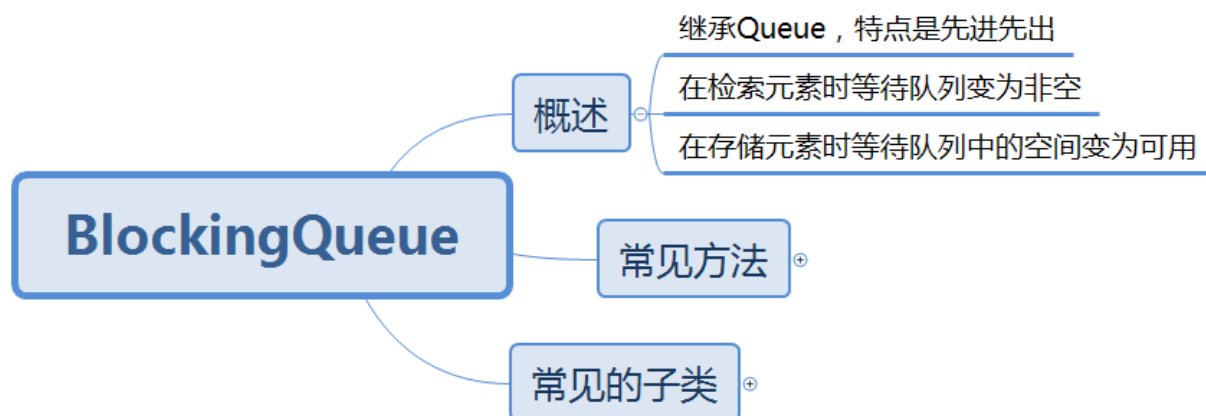
* 学习目标

* 能够掌握BlockingQueue相关阻塞队列

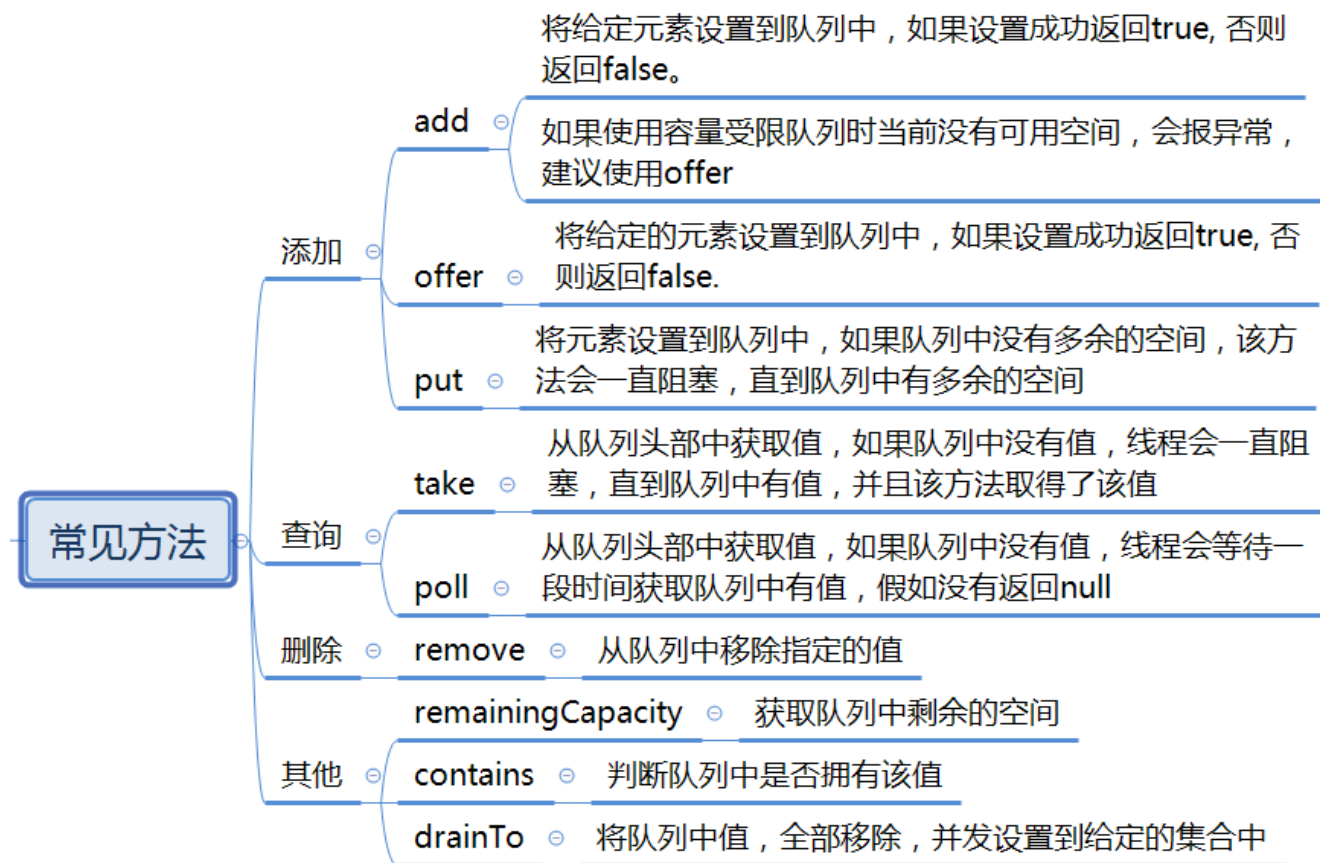
* 能够掌握线程池

* 能够掌握BlockingQueue相关阻塞队列

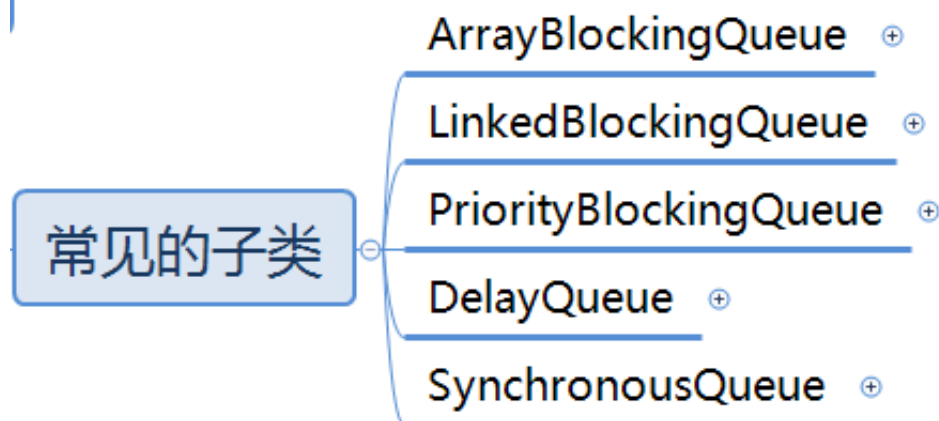
* 概述



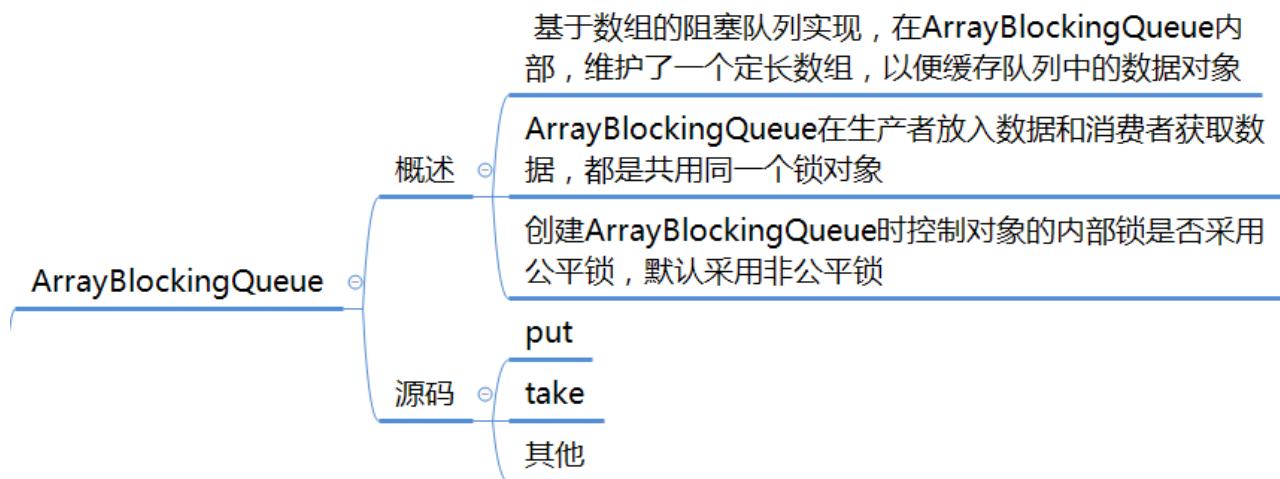
* 常见方法



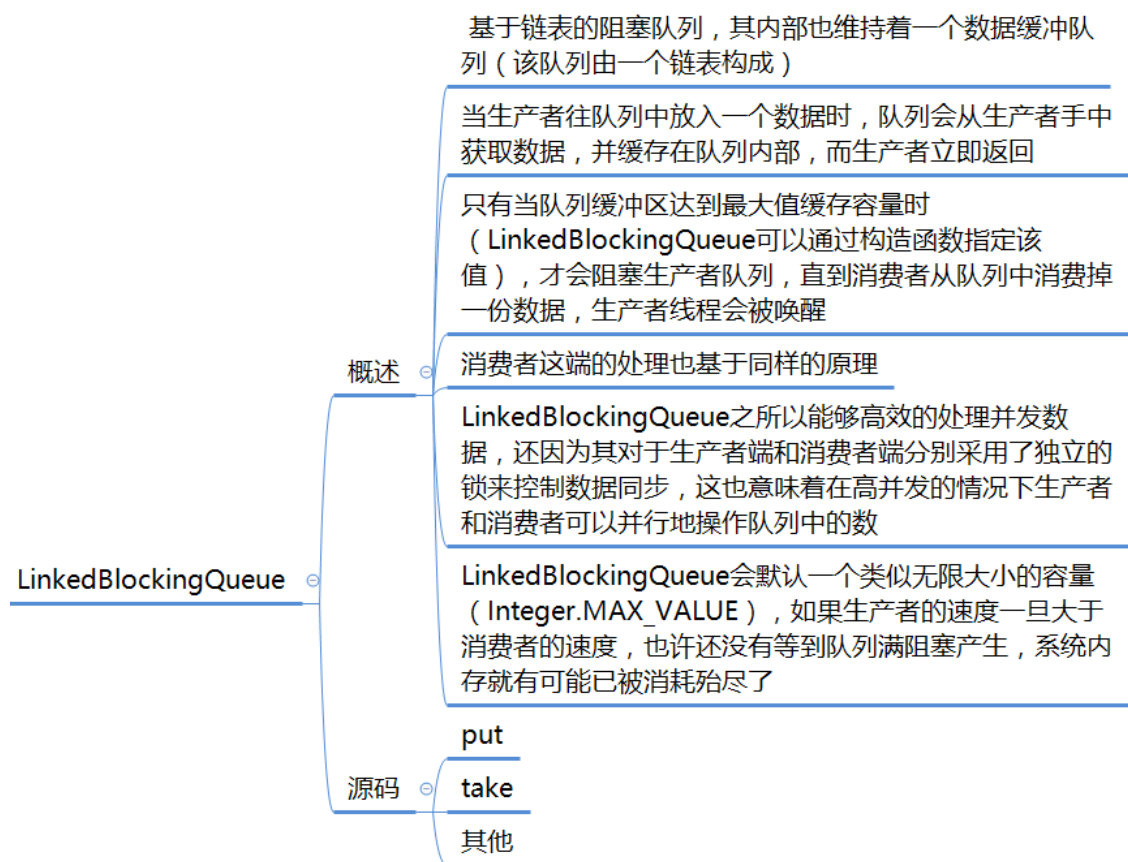
* 常见的子类



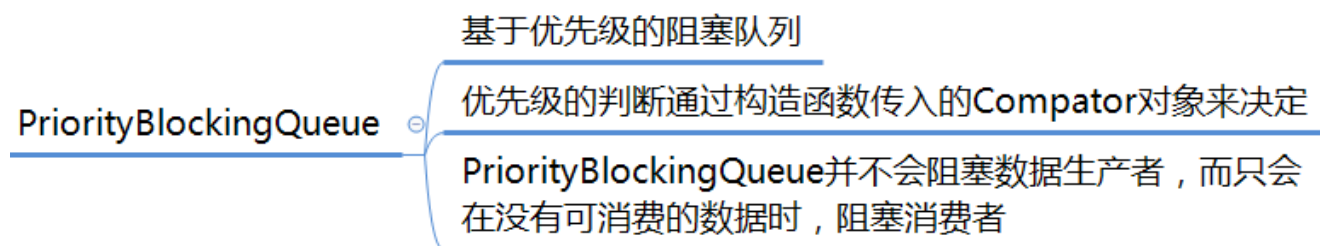
* ArrayBlockingQueue



* LinkedBlockingQueue



* PriorityBlockingQueue



* DelayQueue

DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素

DelayQueue

DelayQueue是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞

* SynchronousQueue

SynchronousQueue

无缓冲的等待队列

并不是真正的队列

是一种管理直接在生产者和消费者之间交流的机制

```
1 * 案例: ArrayBlockingQueue
2 * 生产者
3 public class Producer1 implements Runnable {
4     private BlockingQueue mQueue;
5     public Producer1(BlockingQueue queue){
6         this.mQueue=queue;
7     }
8     @Override
9     public void run() {
10         // 一直在生产
11         Random r=new Random();
12         while (true){
13             // 没有数据，就生产
14             while (mQueue.size()<=0) {
15                 try {
16                     Thread.sleep(2000);
17                     int num = r.nextInt(50);
18                     mQueue.put(num);
19                     System.out.println(Thread.currentThread().getName() + "线程
20 } catch (InterruptedException e) {
21     e.printStackTrace();
22 }
23 }
24 }
```

```

25     }
26 }
27 * 消费者
28 public class Consumer1 implements Runnable {
29     private BlockingQueue mQueue;
30     public Consumer1(BlockingQueue queue){
31         this.mQueue=queue;
32     }
33     @Override
34     public void run() {
35         while (true){
36             while (mQueue.size() > 0) {
37                 try {
38                     Thread.sleep(1000);
39                     System.err.println(Thread.currentThread().getName()+"线
40                 } catch (InterruptedException e) {
41                     e.printStackTrace();
42                 }
43             }
44         }
45     }
46 }
47 * Main
48 public class Main {
49     public static void main(String[] args) {
50         BlockingQueue blockingQueue=new ArrayBlockingQueue(16);
51         for (int i = 0; i < 5; i++) {
52             // 生产者
53             new Thread(new Producer1(blockingQueue),"生产者"+i){}.start();
54             // 消费者
55             new Thread(new Consumer1(blockingQueue),"消费者"+i){}.start();
56         }
57     }
58 }
59 }
60
61 * 案例二: LinkedBlockedQueue
62 * ArrayBlockingQueue 替换成LinkedBlockedQueue
63
64 * 案例三: PriorityBlockingQueue

```

```
65 @Data
66 @AllArgsConstructor
67 public class Task implements Comparable<Task> {
68     private int id;
69     private String name;
70     @Override
71     public int compareTo(Task o) {
72         return this.getId()-o.id ;
73     }
74 }
75
76 public class Main {
77     public static void main(String[] args) throws InterruptedException {
78         BlockingQueue<Task> blockingQueue=new PriorityBlockingQueue<Task> (16);
79         Task task1=new Task(1,"xiaohei");
80         Task task3=new Task(3,"xiaobai");
81         Task task2=new Task(2,"xiaoming");
82         blockingQueue.put(task1);
83         blockingQueue.put(task2);
84         blockingQueue.put(task3);
85         for (Task task:blockingQueue){
86             System.out.println(task);
87         }
88         new Thread(new Runnable() {
89             @Override
90             public void run() {
91                 try {
92                     Thread.sleep(3000);
93                     blockingQueue.put(new Task(5,"xiaohong"));
94                 } catch (InterruptedException e) {
95                     e.printStackTrace();
96                 }
97             }
98         }).start();
99         List<Task> tasks=new ArrayList();
100         blockingQueue.drainTo(tasks);
101         Task task = blockingQueue.take();
102         System.out.println("task = " + task);
103         System.out.println("main over");
104     }
```

```
105
106     }
107 }
108
109 * 案例网吧上网 :DelayQueue
110 * 实现 Delayed接口
111 @Data
112 public class Customer implements Delayed {
113     private int id;
114     private String name;
115     private long endTime;
116
117     @Override
118     public long getDelay(TimeUnit unit) {
119         return this.endTime-System.currentTimeMillis();
120     }
121
122     public Customer(int id, String name, long endTime) {
123         this.id = id;
124         this.name = name;
125         // 加上当前时间
126         this.endTime = endTime+System.currentTimeMillis();
127     }
128
129     @Override
130     public int compareTo(Delayed o) {
131         return (int) (this.endTime-((Customer) o).endTime);
132     }
133 }
134 * DownTask
135 public class DownTask implements Runnable {
136     private BlockingQueue<Customer> mQueue;
137     public DownTask(BlockingQueue<Customer> blockingQueue){
138         this.mQueue=blockingQueue;
139     }
140     @Override
141     public void run() {
142         while (true) {
143             try {
144                 Customer customer = mQueue.take();
```

```

145         System.out.println("编号: " + customer.getId() + " 姓名: " + cus
146     } catch (InterruptedException e) {
147         e.printStackTrace();
148     }
149 }
150 }
151 }
152
153 * Main
154 public class Main {
155     public static void main(String[] args) {
156         BlockingQueue<Customer> queue=new DelayQueue<Customer>();
157         Customer c1=new Customer(1001,"刘备",1000*5);
158         Customer c2=new Customer(1002,"关羽",1000*10);
159         Customer c3=new Customer(1003,"张飞",1000*20);
160         queue.add(c1);
161         queue.add(c2);
162         queue.add(c3);
163         new Thread(new DownTask(queue)).start();
164         SimpleDateFormat sdf =new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
165         while (true){
166             try {
167                 Thread.sleep(1000);
168             } catch (InterruptedException e) {
169                 e.printStackTrace();
170             }
171             System.out.println(sdf.format(new Date()));
172         }
173     }
174 }
175
176 * 案例: SynchronousQueue
177 * 生产者
178 public class Producer1 implements Runnable {
179     private BlockingQueue mQueue;
180     public Producer1(BlockingQueue queue){
181         this.mQueue=queue;
182     }
183     @Override
184     public void run() {

```



```

185 // 一直在生产
186 Random r=new Random();
187 while (true){
188     try {
189         Thread.sleep(2000);
190         int num = r.nextInt(50);
191         mQueue.put(num);
192         System.out.println(Thread.currentThread().getName() + "线程
193     } catch (InterruptedException e) {
194         e.printStackTrace();
195     }
196
197 }
198 }
199 }
200
201 * 消费者
202 public class Consumer1 implements Runnable {
203     private BlockingQueue mQueue;
204     public Consumer1(BlockingQueue queue){
205         this.mQueue=queue;
206     }
207     @Override
208     public void run() {
209         while (true){
210             try {
211                 Thread.sleep(1000);
212                 System.err.println(Thread.currentThread().getName()+"线
213             } catch (InterruptedException e) {
214                 e.printStackTrace();
215             }
216         }
217     }
218 }
219
220 * Main
221 public class Main {
222     public static void main(String[] args) {
223         BlockingQueue blockingQueue=new SynchronousQueue();
224         // 生产者

```

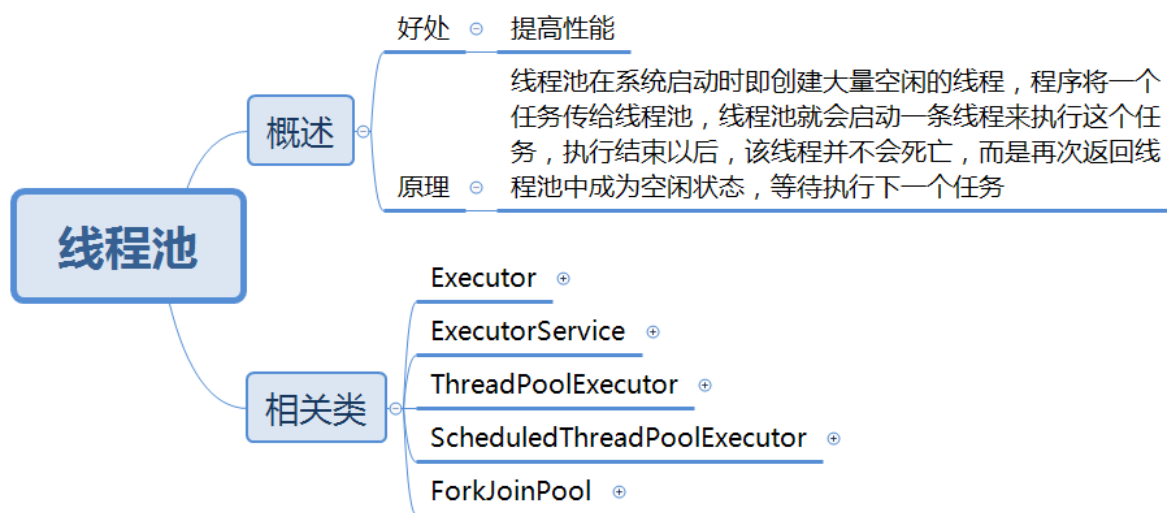
```

225         new Thread(new Producer1(blockingQueue,"生产者")).start();
226         // 消费者
227         new Thread(new Consumer1(blockingQueue,"消费者")).start();
228     }
229 }
230

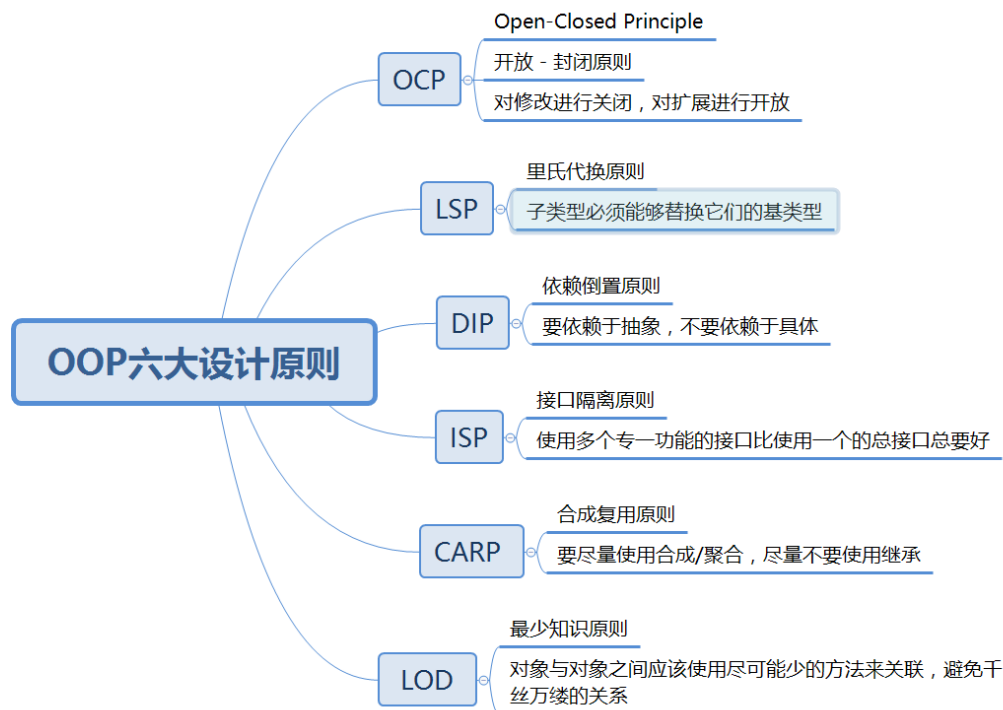
```

* 能够掌握线程池

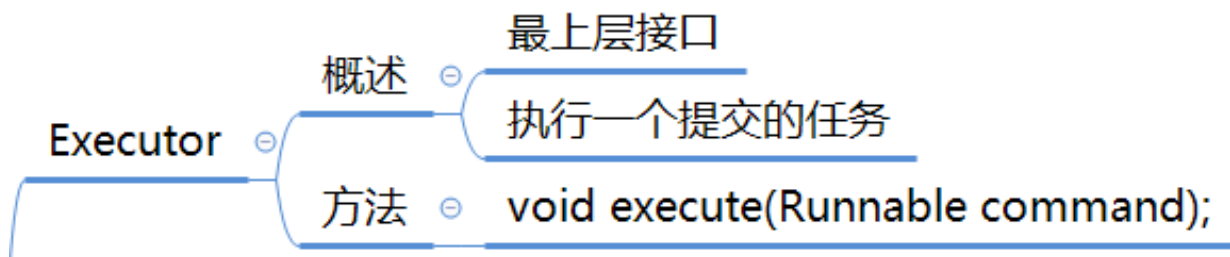
* 线程池概述



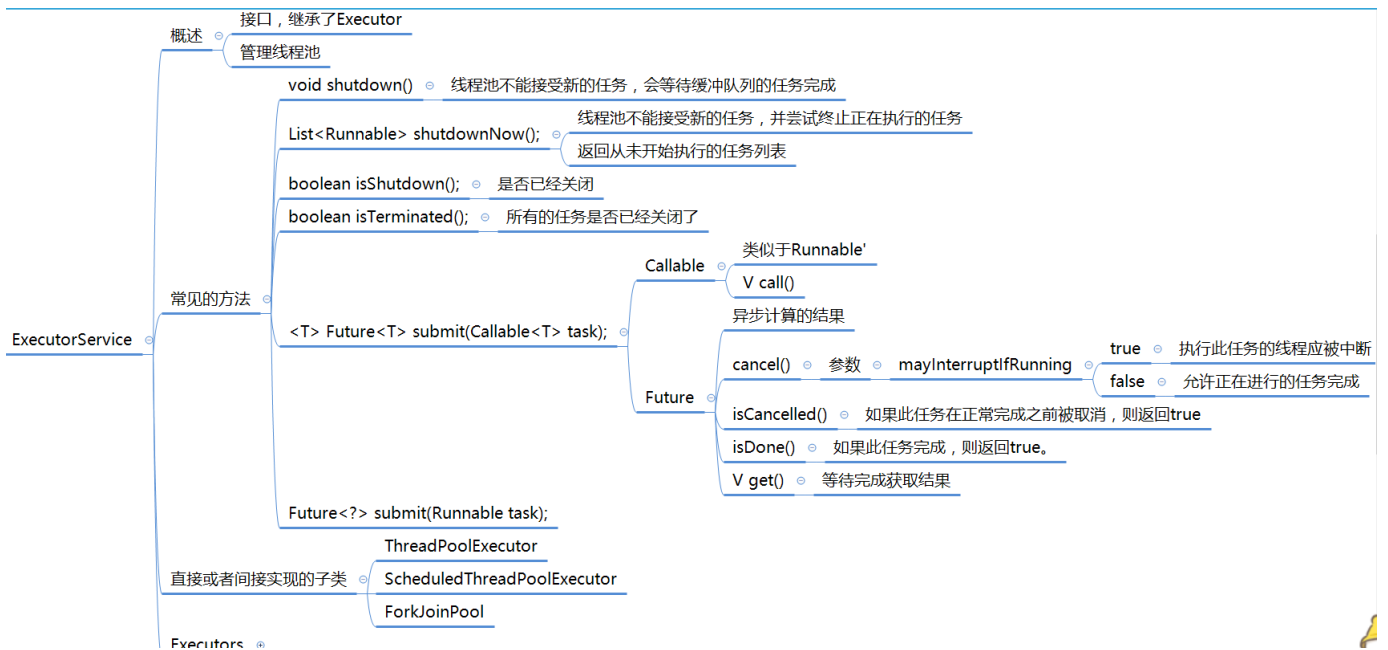
* 面向对象六大设计原则



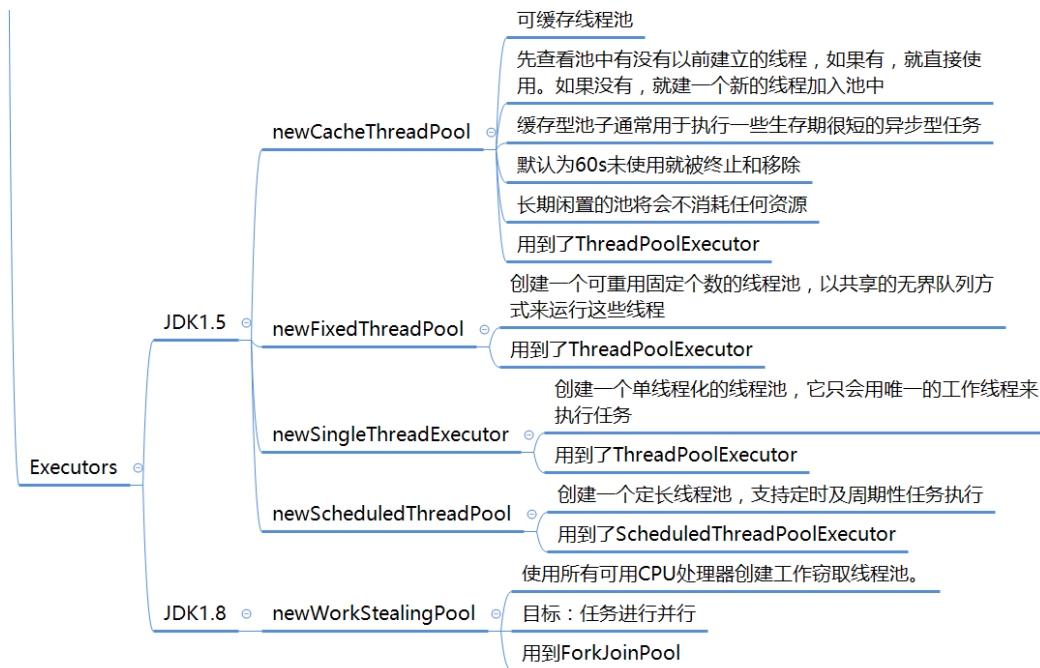
* Executor



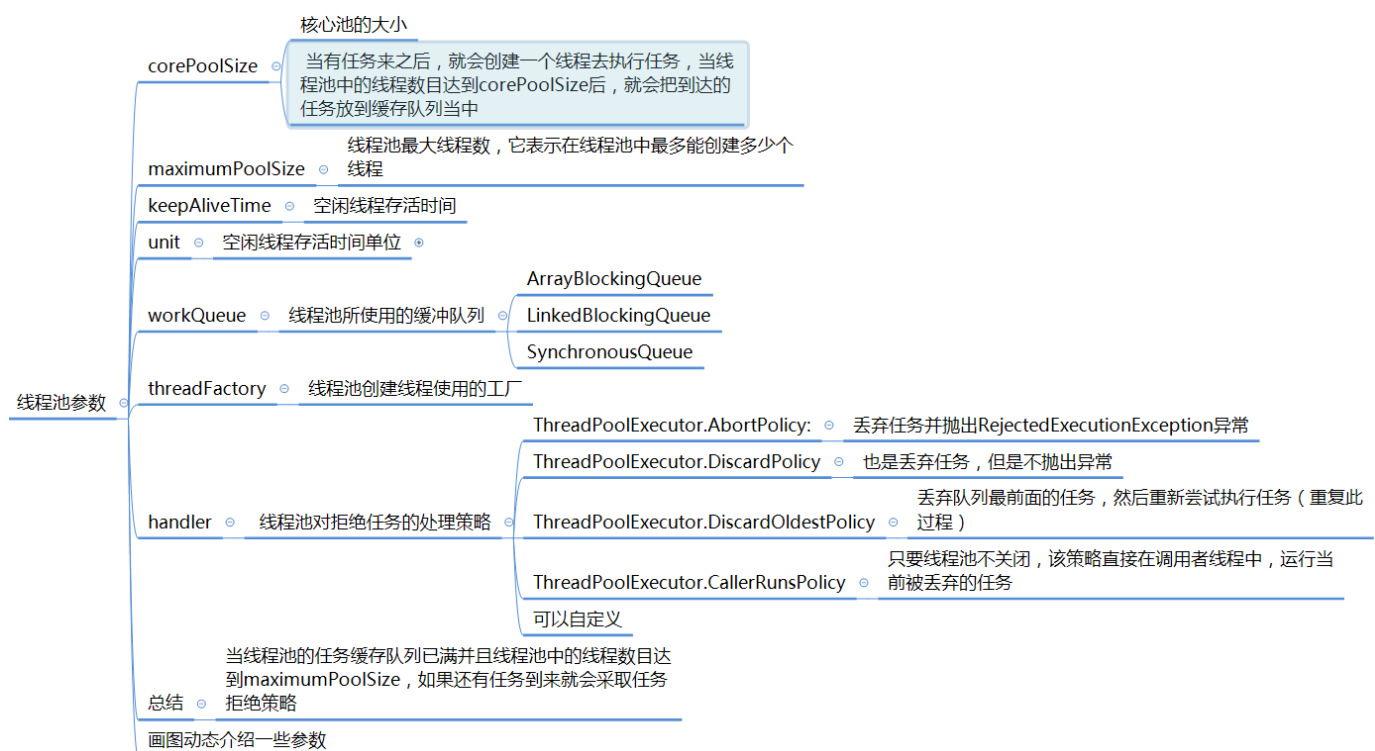
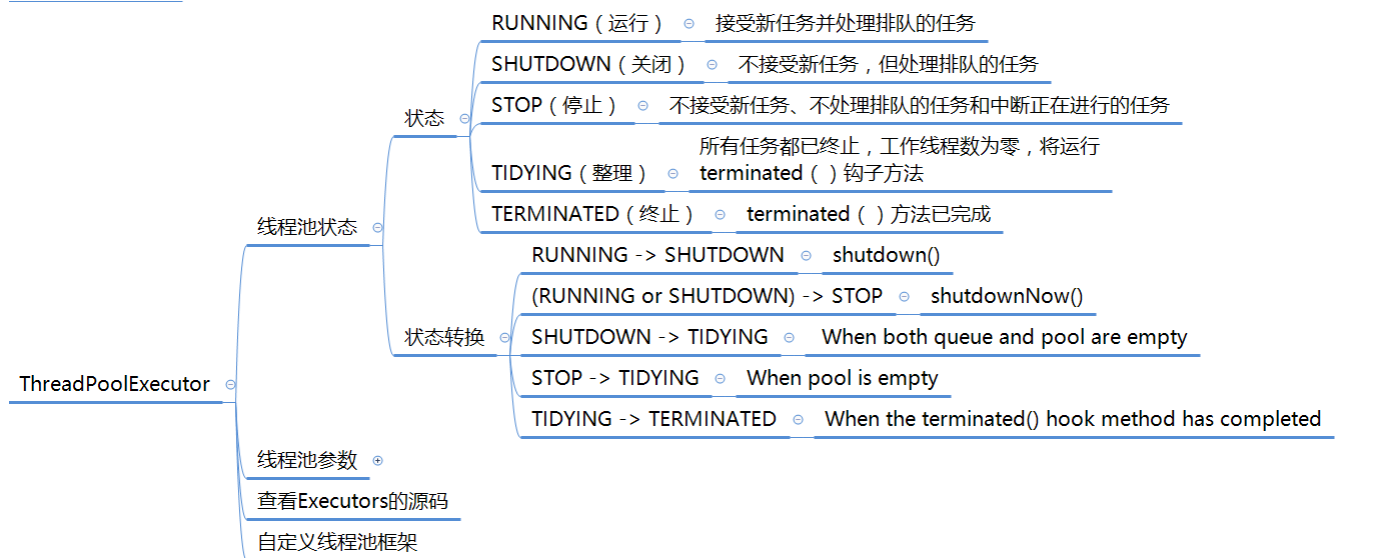
* ExecutorService



* Executors

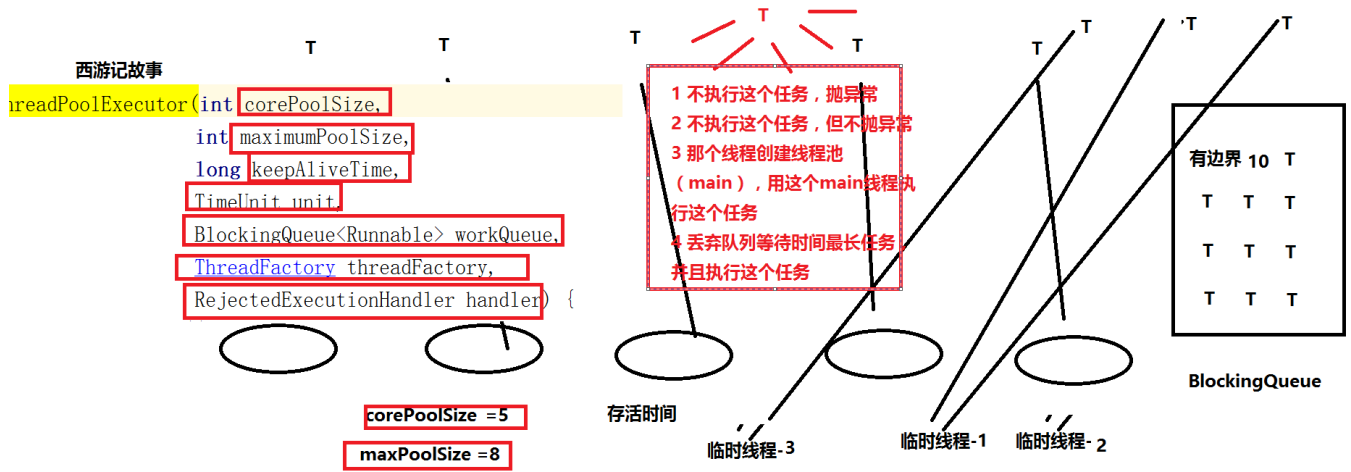


* ThreadPoolExecutor

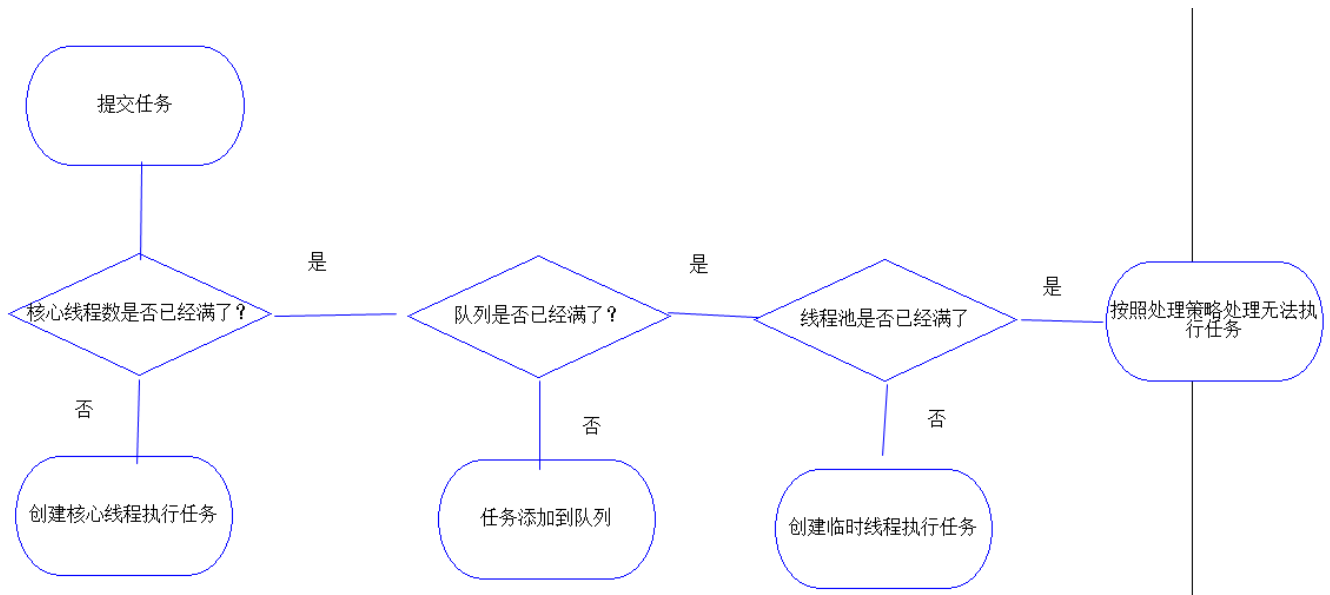


* 线程池执行任务逻辑和线程池参数的关系

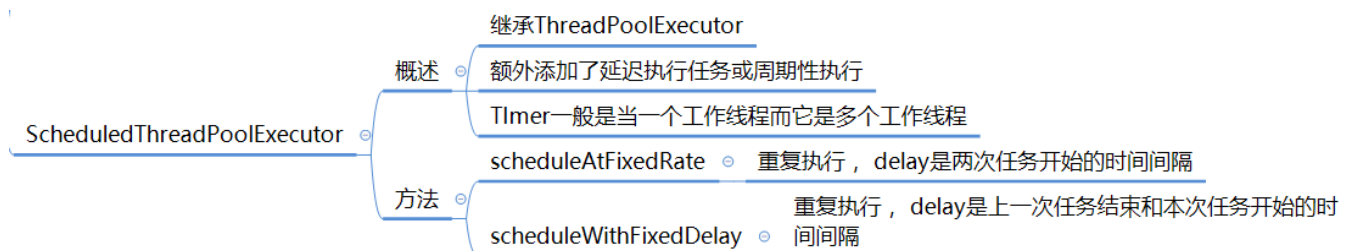
* 西游记的故事



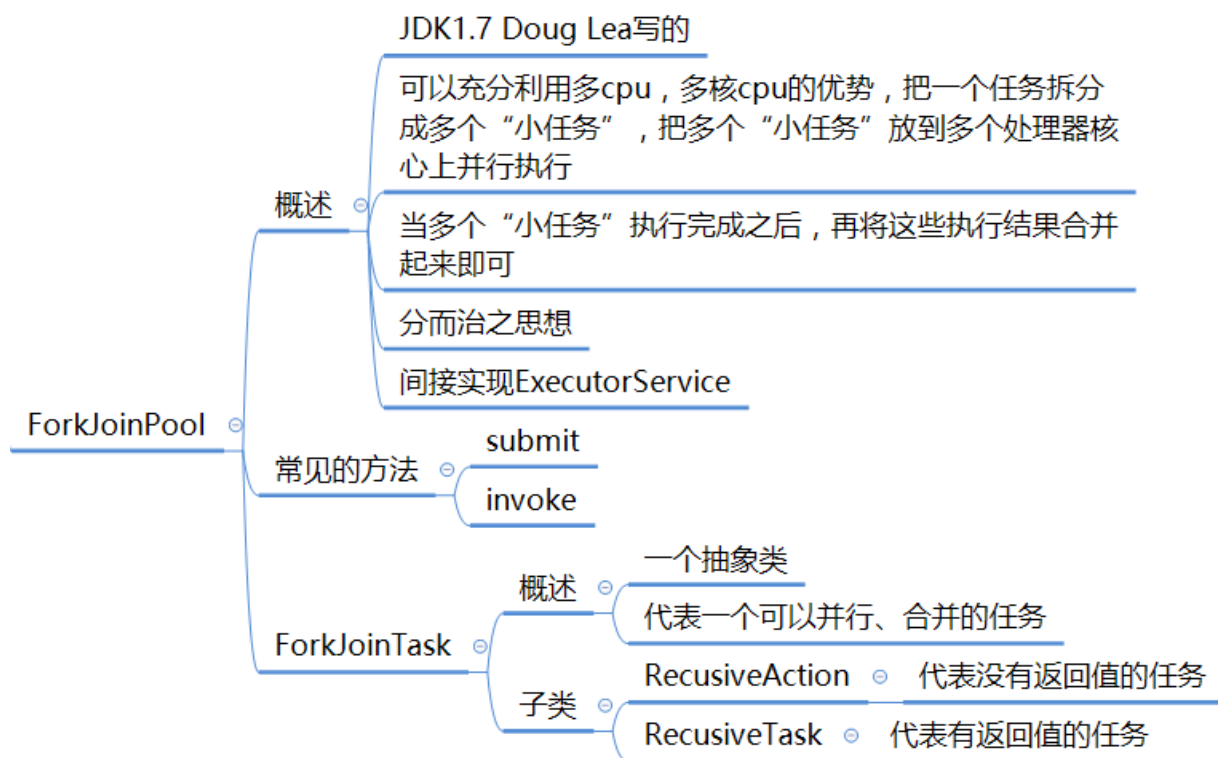
* 流程图



* ScheduledThreadPoolExecutor



* ForkJoinPool



```
1 案例一： Executors
2  public class Main {
3      public static void main(String[] args) {
4          // 阿里建议手动创建线程池，后面讲解
5          ExecutorService executorService = Executors.newCachedThreadPool();
6          //Executors.newSingleThreadExecutor();
7          //Executors.newFixedThreadPool(5);
8          for (int i = 0; i < 100; i++) {
9              int temp = i;
10             executorService.execute(new Runnable() {
11                 @Override
12                 public void run() {
13                     try {
14                         Thread.sleep(1000);
15                     } catch (InterruptedException e) {
16                         e.printStackTrace();
17                     }
18                     System.out.println(Thread.currentThread().getName()+"线程被
19                 }
20             });
21         }
22     }
23 }
```

```

24 }
25 * 查看效果体会
26 * 案例二: ScheduledThreadPoolExecutor: 定时的效果
27     Timer
28 public class Main {
29     public static void main(String[] args) {
30         // 阿里建议手动创建线程池, 后面讲解
31         ScheduledThreadPoolExecutor schedule =
32             (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(
33             // 1 秒后执行此任务
34             schedule.schedule(new Runnable() {
35                 @Override
36                 public void run() {
37                     System.out.println(Thread.currentThread().getName()+"线程被执行")
38                 }
39             },1,TimeUnit.SECONDS);
40         // 是固定的频率来执行某项计划, 它不受计划执行时间的影响
41         schedule.scheduleAtFixedRate(new Runnable() {
42             @Override
43             public void run() {
44                 try {
45                     Thread.sleep(2000);
46                 } catch (InterruptedException e) {
47                     e.printStackTrace();
48                 }
49                 System.err.println(Thread.currentThread().getName()+"线程被执行")
50             }
51             },1000,5000, TimeUnit.MILLISECONDS);
52         //无论某个任务执行多长时间, 等执行完了, 我再延迟指定的时间
53         schedule.scheduleWithFixedDelay(new Runnable() {
54             @Override
55             public void run() {
56                 try {
57                     Thread.sleep(2000);
58                 } catch (InterruptedException e) {
59                     e.printStackTrace();
60                 }
61                 System.out.println(Thread.currentThread().getName()+"线程被执行")
62             }
63             },1000,5000, TimeUnit.MILLISECONDS);

```



```

64     }
65 }
66
67 public class Main {
68     public static void main(String[] args) {
69         // 阿里建议是ScheduledThreadPoolExecutor代替Timer
70         Timer timer=new Timer();
71         timer.schedule(new TimerTask() {
72             @Override
73             public void run() {
74                 System.err.println(Thread.currentThread().getName()+"线程被执行");
75             }
76         },1000,2000);
77     }
78 }
79
80 * 案例三: ExecutorService 常见的方法
81 public class Main {
82     public static void main(String[] args) throws Exception {
83         ExecutorService pool = Executors.newFixedThreadPool(5);
84         for (int i = 0; i < 10; i++) {
85             Future<String> future = pool.submit(new Callable<String>() {
86                 @Override
87                 public String call() throws Exception {
88                     Thread.sleep(3000);
89                     System.out.println(Thread.currentThread().getName()+"线程被
90                         return "success";
91                 }
92             });
93             System.out.println(future.get());
94         }
95         pool.shutdown();
96         // pool.shutdownNow();
97     }
98 }
99
100 * 案例四: ForkJoinPool
101 public class Main {
102     public static void main(String[] args) throws Exception {
103         ForkJoinPool pool=new ForkJoinPool();

```

```

104 //创建实例，并执行分割任务
105 for (int i = 0; i < 10; i++) {
106     ForkJoinTask<Void> submit = pool.submit(new RecursiveAction() {
107         @Override
108         protected void compute() {
109             System.out.println(Thread.currentThread().getName()+"线程被
110             try {
111                 Thread.sleep(2000);
112             } catch (InterruptedException e) {
113                 e.printStackTrace();
114             }
115         }
116     });
117 }
118 //线程阻塞，等待所有任务完成
119 pool.awaitTermination(10,TimeUnit.SECONDS);
120 pool.shutdown();
121 }
122 }
123
124 * 案例五:
125 public class Main {
126     public static void main(String[] args) throws Exception {
127         int corePoolSize=5;
128         int maximumPoolSize=8;
129         long keepAliveTime=5;
130         TimeUnit unit= TimeUnit.SECONDS;
131         BlockingQueue<Runnable> workQueue=new LinkedBlockingDeque<>(10);
132         ThreadFactory threadFactory=Executors.defaultThreadFactory();
133         RejectedExecutionHandler handler=new ThreadPoolExecutor.AbortPolicy();
134         ThreadPoolExecutor pool=new ThreadPoolExecutor(corePoolSize,maximumPool
135             keepAliveTime,unit,workQueue,threadFactory,handler);
136         // 5 测试
137         // 6 测试
138         //16 测试
139         // 19 测试
140         // 更换拒绝策略，自定义策略
141         for (int i = 0; i < 5 ; i++) {
142             int temp=i;
143             pool.submit(new Callable<String>() {

```

```

144         @Override
145         public String call() throws Exception {
146             Thread.sleep(1000);
147             System.out.println(Thread.currentThread().getName()+"执行了");
148             return "Success";
149         }
150     });
151 }
152 pool.shutdown();
153 }
154 }
155
156 * 案例六：自定义ThreadPool
157 public class ThreadPoolUtils {
158     private static int corePoolSize;
159     private static int maximumPoolSize;
160     private static long keepAliveTime;
161     private static String capacity;
162     private static RejectedExecutionHandler handler;
163     private static BlockingQueue<Runnable> workQueue;
164     private static TimeUnit unit= TimeUnit.MILLISECONDS;
165     private static ThreadFactory threadFactory= Executors.defaultThreadFactory();
166     private static ThreadPoolExecutor pool;
167     private ThreadPoolUtils(){
168
169     }
170     static {
171         Properties prop=new Properties();
172         try {
173             prop.load(ThreadPoolUtils.class.getClassLoader().getResourceAsStream("threadPool.properties"));
174         } catch (IOException e) {
175             e.printStackTrace();
176         }
177         corePoolSize = Integer.parseInt(prop.getProperty("corePoolSize", "5"));
178         maximumPoolSize= Integer.parseInt(prop.getProperty("maximumPoolSize", "8"));
179         keepAliveTime =Long.parseLong(prop.getProperty("keepAliveTime", "1000"));
180         capacity = prop.getProperty("capacity","0");
181         if(capacity.equals("0")){
182             workQueue=new LinkedBlockingDeque<>();
183         }else{

```

```

184         workQueue=new LinkedBlockingDeque<>(Integer.parseInt(capcity));
185     }
186     String handler1 = prop.getProperty("handler","1");
187     switch (handler1){
188         case "1":
189             handler=new ThreadPoolExecutor.AbortPolicy();
190             break;
191         case "2":
192             handler=new ThreadPoolExecutor.DiscardPolicy();
193             break;
194         case "3":
195             handler=new ThreadPoolExecutor.DiscardOldestPolicy();
196             break;
197         case "4":
198             handler=new ThreadPoolExecutor.CallerRunsPolicy();
199             break;
200         default:
201             handler=new ThreadPoolExecutor.AbortPolicy();
202             break;
203     }
204     pool=new ThreadPoolExecutor(corePoolSize,maximumPoolSize,keepAliveTime,
205 }
206
207     public static ThreadPoolExecutor getPool(){
208         return pool;
209     }
210 }
211 * 配置文件
212 corePoolSize=5
213 maximumPoolSize=8
214 keepAliveTime=1000
215 capcity=10
216 -- 1:AbortPolicy|DiscardPolicy|DiscardOldestPolicy|CallerRunsPolicy
217 handler=1
218
219 * 测试代码
220 public class Main {
221     public static void main(String[] args) throws Exception {
222         // 5 测试
223         // 6 测试

```

```
224 //16 测试
225 // 19 测试
226 // 更换拒绝策略，自定义策略
227 for (int i = 0; i <19 ; i++) {
228     int temp=i;
229     ThreadPoolUtils.getPool().submit(new Callable<String>() {
230         @Override
231         public String call() throws Exception {
232             Thread.sleep(1000);
233             System.out.println(Thread.currentThread().getName()+"执行了
234                 return "Success";
235         }
236     });
237 }
238 ThreadPoolUtils.getPool().shutdown();
239 }
240 }
241
242
```