


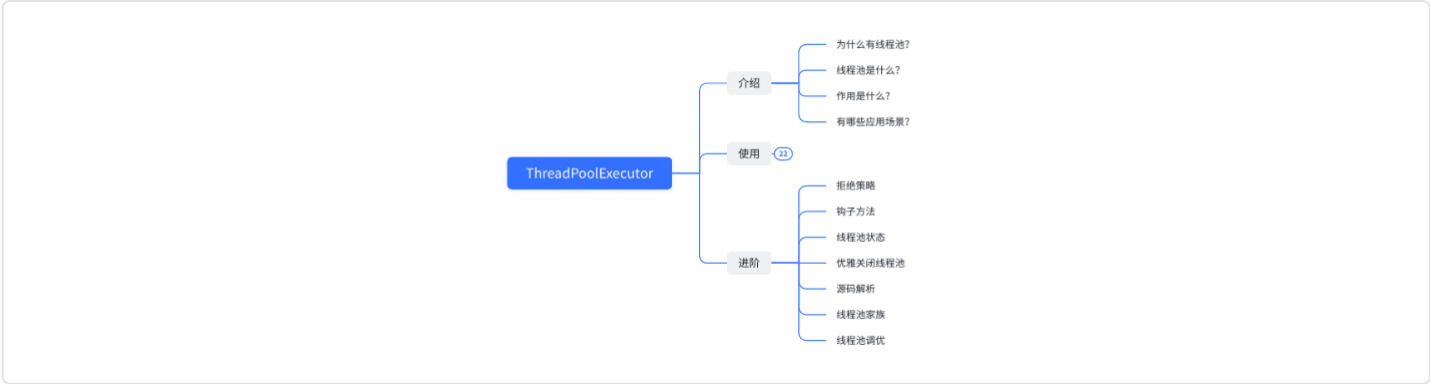
ThreadPoolExecutor

官方文档：<https://docs.oracle.com/javase/8/docs/api/index.html>

Java Platform SE 8

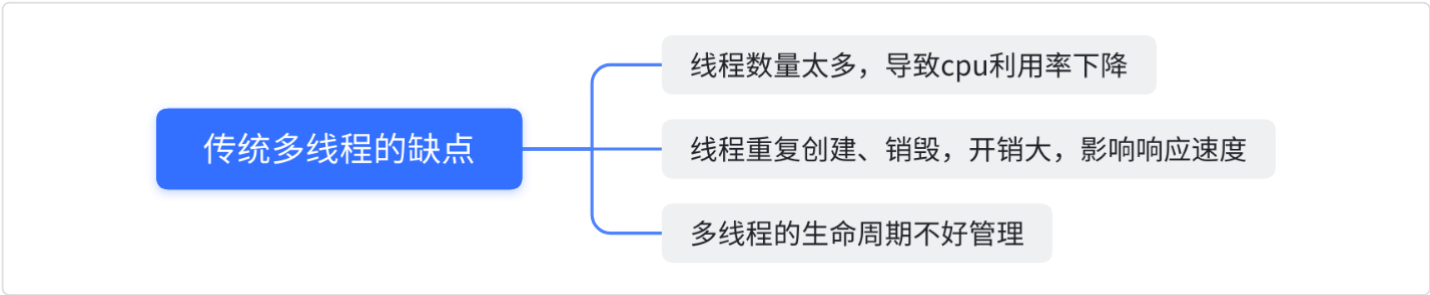
<noscript> <div>JavaScript is disabled on your browser.</div> </noscript> <h2>Frame Alert</h2> <p>This document is designed to be viewed using the frames feature. If you see this message, you are…

 <https://docs.oracle.com/javase/8/docs/api/index.html>



介绍

为什么会有线程池？



传统多线程的缺点：

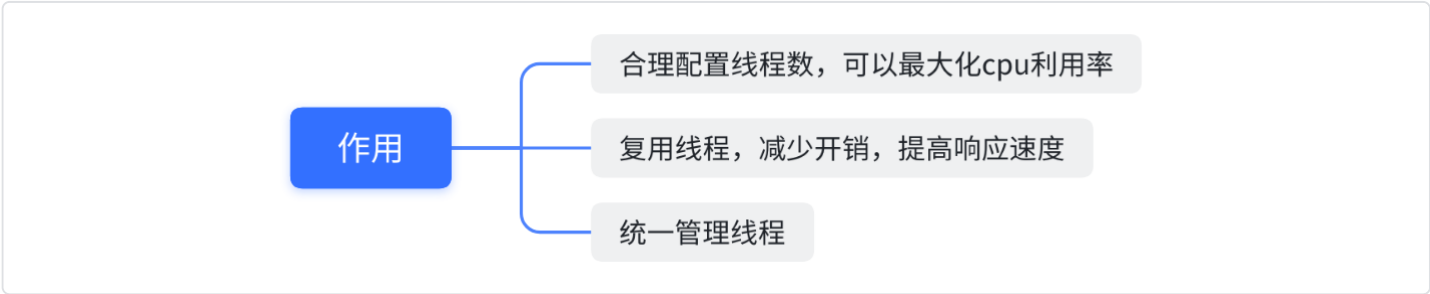
如果每到达一个请求，就去创建一个线程，这样线程数量是特别多的。线程上下文频繁切换，会导致cpu利用率下降。而且每个线程都需要不停的创建和销毁，这样开销是特别大的，会影响响应速度。

线程池是什么

线程池是java中运用最广泛的并发框架，几乎所有需要**异步**和**并发执行任务**的程序都可以使用线程池。

线程池的作用

解决了传统多线程的问题

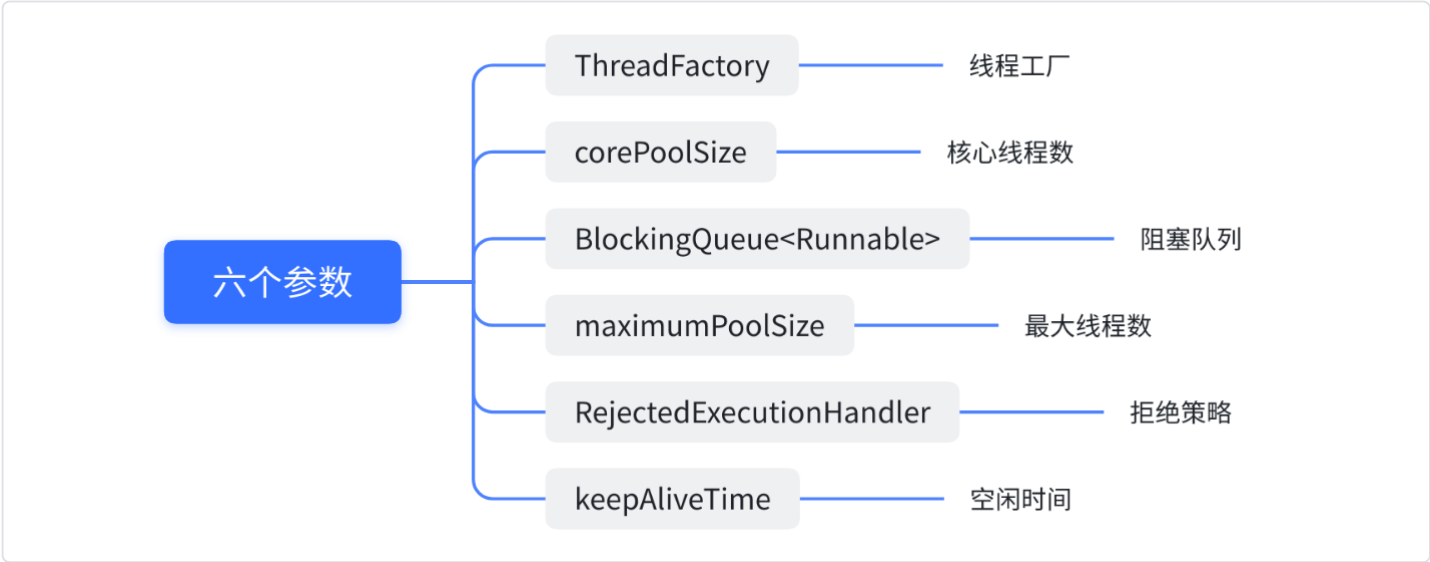


应用场景

根据不同的线程池类型，有不同的应用场景，后面会介绍

使用

六大参数介绍



因为使用和参数、原理绑定在一起，所以使用前需要简单讲下参数和原理

工作原理

图1

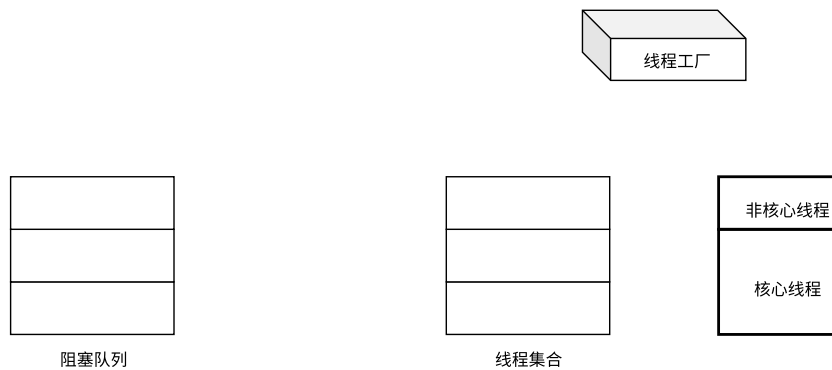


图2

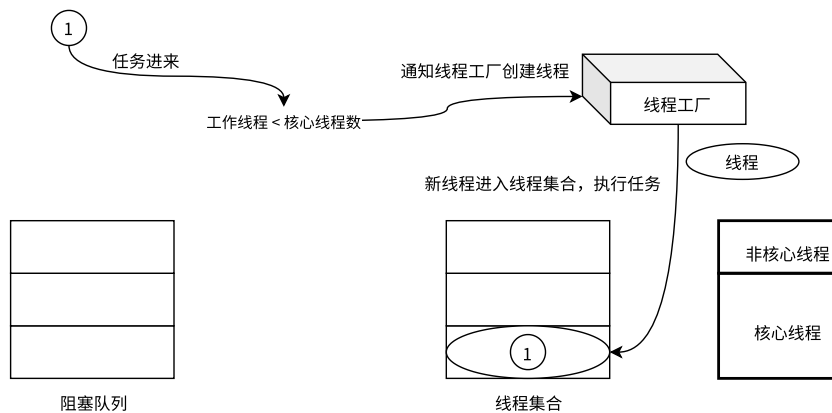


图3

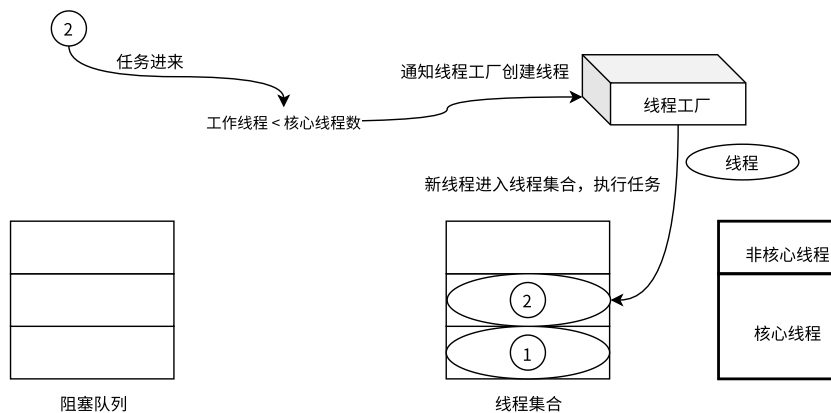
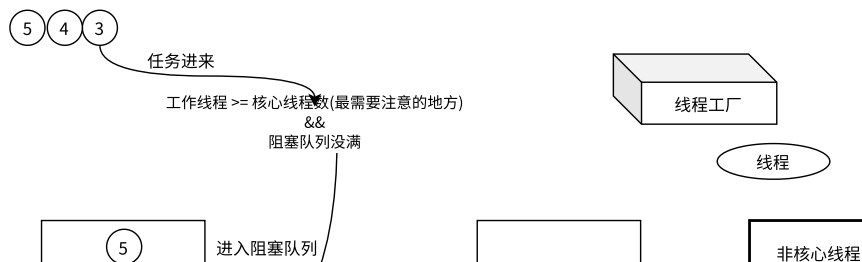


图4



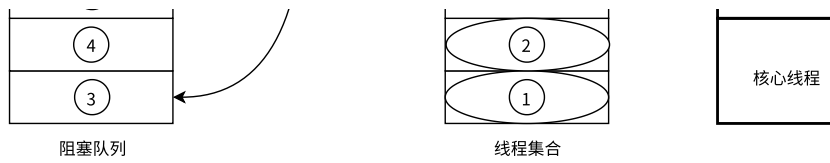


图5

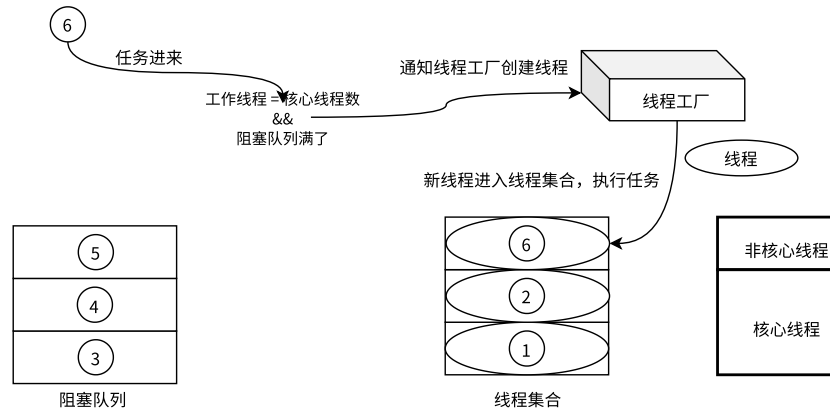


图6

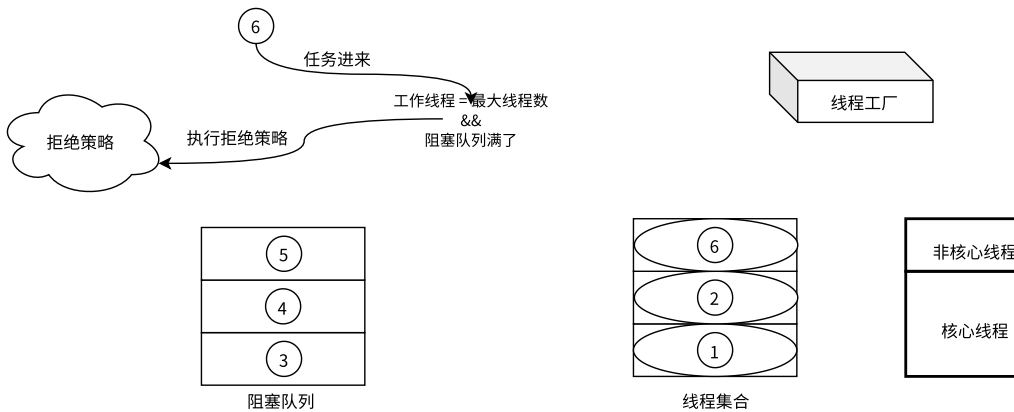
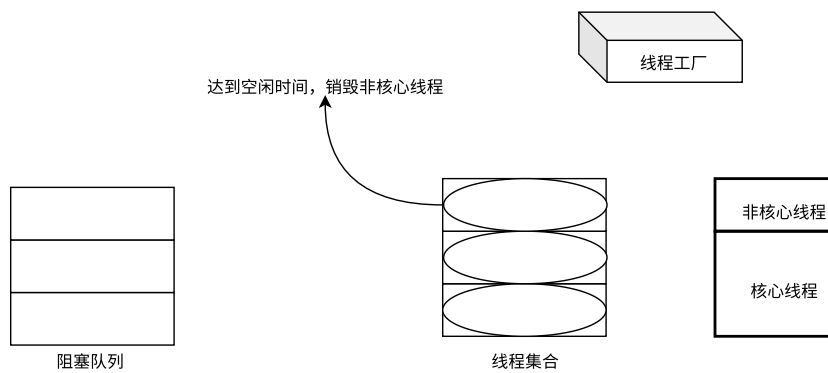


图7

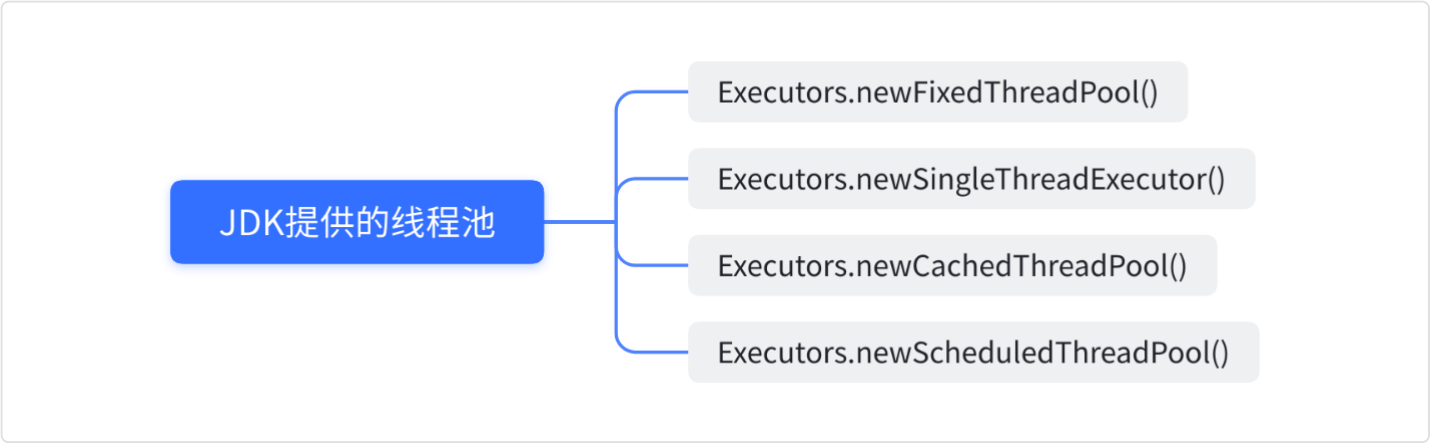


在使用线程池之前，先介绍一下线程池的参数

ThreadFactory	工厂方法设计模式，用来创建和管理线程用的，默认使用的是默认工厂，创建的是拥有相同线程组、相同优先级的用户线程
corePoolSize	线程池正常工作时的线程数
BlockingQueue<Runnable>	当正在工作线程数达到核心线程数时，再有任务进来，就会被放到阻塞队列中
maximumPoolSize	当工作线程数达到核心线程数，且阻塞队列满了，就会进一步增加线程数，最多达到最大线程数
RejectedExecutionHandler	当工作线程数达到最大线程数，且阻塞队列满了，就会启动拒绝策略，新来的任务就会被拒绝。
keepAliveTime	当非核心线程空闲了，并且达到空闲时间，那么就会进行销毁线程

创建线程池

jdk提供的线程池



	作用	应用场景	核心线程数	最大线程数	空闲时间	阻塞队列
FixedThreadPool	固定线程数	稳定执行，任务生成速度波动不大	指定	同核心线程数	0	LinkedBlockingQueue
SingleThreadEx	单线程	按入队顺序执行任务	1	1	0	LinkedBlockin

ecutor						gQueue
CachedThreadP ool	可缓存	适合任务执行速度快的	0	Integer.MAX_ VALUE	60s	SynchronousQ ueue
ScheduledThre adPool	延时、周 期执行	适合需要延时或者周期 执行的任务	指定	Integer.MAX_ VALUE	0	DelayedWorkQ ueue

自定义线程池

默认线程池的缺点

FixedThreadPool	无界队列，任务消耗速度 < 任务生成速度容易发生内存泄漏
SingleThreadExecutor	无界队列，任务消耗速度 < 任务生成速度容易发生内存泄漏
CachedThreadPool	无限线程，任务消耗速度 < 任务生成速度容易发生内存溢出
ScheduledThreadPool	无限线程，任务消耗速度 < 任务生成速度容易发生内存溢出

自己定义线程池的优点

可以根据不同的业务场景定义合适的线程池参数，合理定义线程数量可以最大化CPU利用率

Java

```
1 public class ThreadPoolExecutorDemo {
2     public static void main(String[] args) {
3         ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
4             5,      // 核心线程数
5             10,     // 最大线程数
6             2000L,  // 空闲时间
7             TimeUnit.MILLISECONDS, // 时间单位
8             new ArrayBlockingQueue<Runnable>(10), // 阻塞队列
9             Executors.defaultThreadFactory(), // 线程工厂
10            new ThreadPoolExecutor.AbortPolicy()); // 拒绝策略
11     }
12 }
```

启动线程池

execute(runnable)

会发现JVM不会退出，因为线程池还在运行

Java

```
1  public static void executeDemo() {
2      Runnable runnable = () -> {
3          System.out.println(Thread.currentThread().getName() + "执行任务");
4          try {
5              Thread.sleep(2000);
6          } catch (InterruptedException e) {
7              e.printStackTrace();
8          }
9      };
10
11     ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5,
12         10,
13         2L,
14         TimeUnit.MILLISECONDS,
15         new ArrayBlockingQueue<Runnable>(10),
16         Executors.defaultThreadFactory(),
17         new ThreadPoolExecutor.AbortPolicy());
18     threadPoolExecutor.allowCoreThreadTimeOut(false);
19     // 证明不是线程池创建的时候就马上创建线程的
20     System.out.println(threadPoolExecutor.toString());
21     // 演示核心线程数
22     //     for (int i = 0; i < 15; i++) {
23     //         threadPoolExecutor.execute(runnable);
24     //     }
25     // 演示最大线程数
26     //     for (int i = 0; i < 16; i++) {
27     //         threadPoolExecutor.execute(runnable);
28     //     }
29
30     // 演示拒绝策略
31     for (int i = 0; i < 21; i++) {
32         threadPoolExecutor.execute(runnable);
33     }
34     //证明线程数达到最大线程
35     System.out.println(threadPoolExecutor.toString());
36     try {
37         Thread.sleep(5000);
38     } catch (InterruptedException e) {
39         e.printStackTrace();
40     }
```

```

40     }
41     //证明线程数减少到核心线程
42     System.out.println(threadPoolExecutor.toString());
43 }

```

停止线程池

停止线程池

shutdown

shutdownNow

shutdown	拒绝新任务，并把等待阻塞队列中的任务全部运行结束后，销毁所有线程
shutdownNow	拒绝新任务，中断正在执行的线程，返回等待任务后，销毁所有线程

shutdown()有序关闭

Java

```

1  public static void shutdownDemo() throws InterruptedException {
2      // 创建任务
3      Runnable runnable = () -> {
4          System.out.println(Thread.currentThread().getName() + "执行任务");
5          try {
6              Thread.sleep(2000);
7          } catch (InterruptedException e) {
8              System.out.println(Thread.currentThread().getName() + "被中断唤醒");
9          }
10     };
11     // 创建线程池
12     ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5,
13         10,
14         2L,
15         TimeUnit.MILLISECONDS,
16         new ArrayBlockingQueue<Runnable>(10),
17         Executors.defaultThreadFactory(),
18         new MyRejectedExecutionHandler());
19     threadPoolExecutor.allowCoreThreadTimeOut(false);

```



```

20      // 打印线程池初始状态
21      System.out.println(threadPoolExecutor.toString());
22      for (int i = 0; i < 20; i++) {
23          threadPoolExecutor.execute(runnable);
24      }
25      // 打印线程池正常工作情况
26      System.out.println(threadPoolExecutor.toString());
27      // 把所有任务都完成
28      threadPoolExecutor.shutdown();
29      // 返回阻塞队列中的任务，后续手动处理，具体怎么处理看业务场景
30      // List<Runnable> runnables = threadPoolExecutor.shutdownNow();
31      // 线程停止后，再有任务进来，抛出拒绝异常
32      threadPoolExecutor.execute(runnable);
33      // 证明停止后不会再接收任务
34      try {
35          threadPoolExecutor.execute(runnable);
36      } catch (Exception e) {
37          System.out.println("异常被catch住了");
38      }
39      // 让任务都执行完
40      Thread.sleep(5000);
41      // 打印线程池停止后的情况
42      System.out.println(threadPoolExecutor.toString());
43      //      System.out.println( threadPoolExecutor.isShutdown());
44      //      System.out.println( threadPoolExecutor.isTerminated());
45      //      System.out.println( threadPoolExecutor.isTerminating());
46
47      }

```

shutdownNow()立即关闭

D

```

1  public static void shutdownDemo() throws InterruptedException {
2      // 创建任务
3      Runnable runnable = () -> {
4          System.out.println(Thread.currentThread().getName() + "执行任务");
5          try {
6              Thread.sleep(2000);
7          } catch (InterruptedException e) {
8              System.out.println(Thread.currentThread().getName() + "被中断唤醒");
9          }
10     }

```

```

10     };
11     // 创建线程池
12     ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5,
13         10,
14         2L,
15         TimeUnit.MILLISECONDS,
16         new ArrayBlockingQueue<Runnable>(10),
17         Executors.defaultThreadFactory(),
18         new MyRejectedExecutionHandler());
19     threadPoolExecutor.allowCoreThreadTimeOut(false);
20     // 打印线程池初始状态
21     System.out.println(threadPoolExecutor.toString());
22     for (int i = 0; i < 20; i++) {
23         threadPoolExecutor.execute(runnable);
24     }
25     // 打印线程池正常工作情况
26     System.out.println(threadPoolExecutor.toString());
27     // 把所有任务都完成
28     // threadPoolExecutor.shutdown();
29     // 返回阻塞队列中的任务，后续手动处理，具体怎么处理看业务场景
30     List<Runnable> runnables = threadPoolExecutor.shutdownNow();
31     // 线程停止后，再有任务进来，抛出拒绝异常
32     threadPoolExecutor.execute(runnable);
33     // 证明停止后不会再接收任务
34     try {
35         threadPoolExecutor.execute(runnable);
36     } catch (Exception e) {
37         System.out.println("异常被catch住了");
38     }
39     // 让任务都执行完
40     Thread.sleep(5000);
41     // 打印线程池停止后的情况
42     System.out.println(threadPoolExecutor.toString());
43     // System.out.println( threadPoolExecutor.isShutdown());
44     // System.out.println( threadPoolExecutor.isTerminated());
45     // System.out.println( threadPoolExecutor.isTerminating());
46
47 }

```

自动关闭线程池

被gc回收的时候

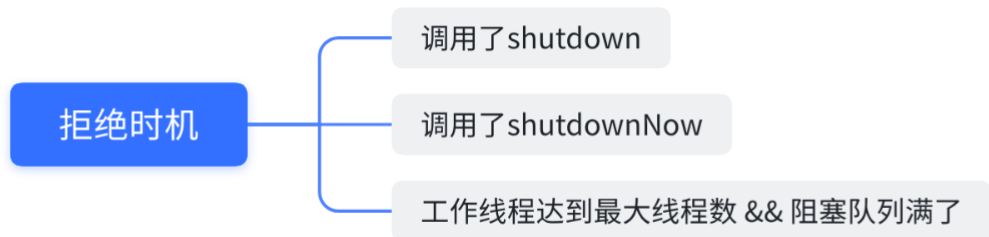
Java

```
1  protected void finalize() {  
2      SecurityManager sm = System.getSecurityManager();  
3      if (sm == null || acc == null) {  
4          // 停止线程池  
5          shutdown();  
6      } else {  
7          PrivilegedAction<Void> pa = () -> { shutdown(); return null; };  
8          AccessController.doPrivileged(pa, acc);  
9      }  
10 }
```

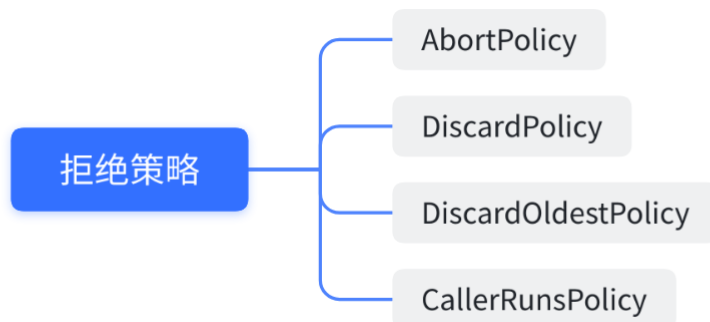
进阶

拒绝策略

拒绝时机



默认拒绝策略



AbortPolicy	无视任务，直接抛异常（默认）	方法里面直接抛出RejectedExecutionException异常
DiscardPolicy	无视任务，没有任何提示	方法里面是空的，不做任何处理
DiscardOldest Policy	丢弃旧的任务	<div>Java</div> <pre>1 if (!e.isShutdown()) { 2 e.getQueue().poll(); //出队列 3 e.execute(r); //出完队列重新执行execute方法，通常是被放到阻塞队列中 4 }</pre>
CallerRunsPolicy	使用提交任务的线程执行任务	<div>Java</div> <pre>1 if (!e.isShutdown()) { 2 r.run(); //当前线程直接调用Runnable的方法 3 }</pre>

自定义拒绝策略

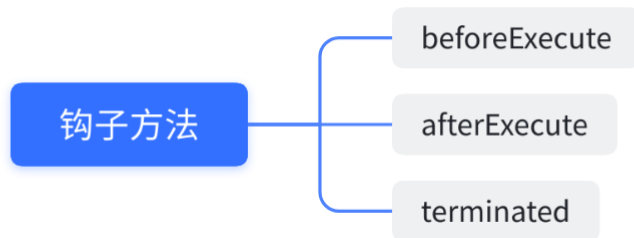
实现RejectedExecutionHandler接口，自定义策略

Java

```
1  static class MyRejectedExecutionHandler implements RejectedExecutionHandler {
2      @Override
3      public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
4          System.out.println("这个任务我不干!");
5      }
6  }
```

钩子方法

空方法，主要用来给子类重写，通常用来记录日志等，类似aop的作用

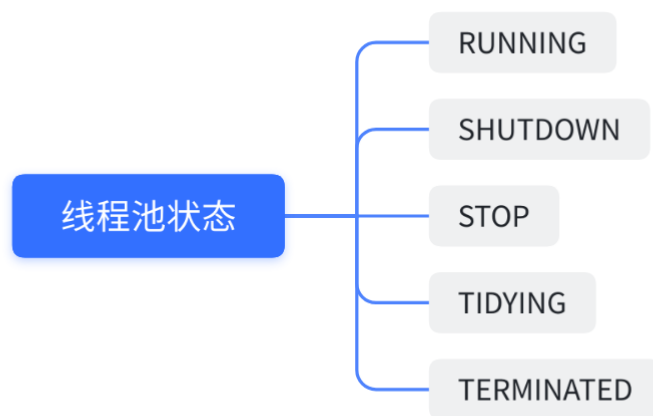


beforeExecute	任务执行前
afterExecute	任务执行后
terminated	线程池结束后

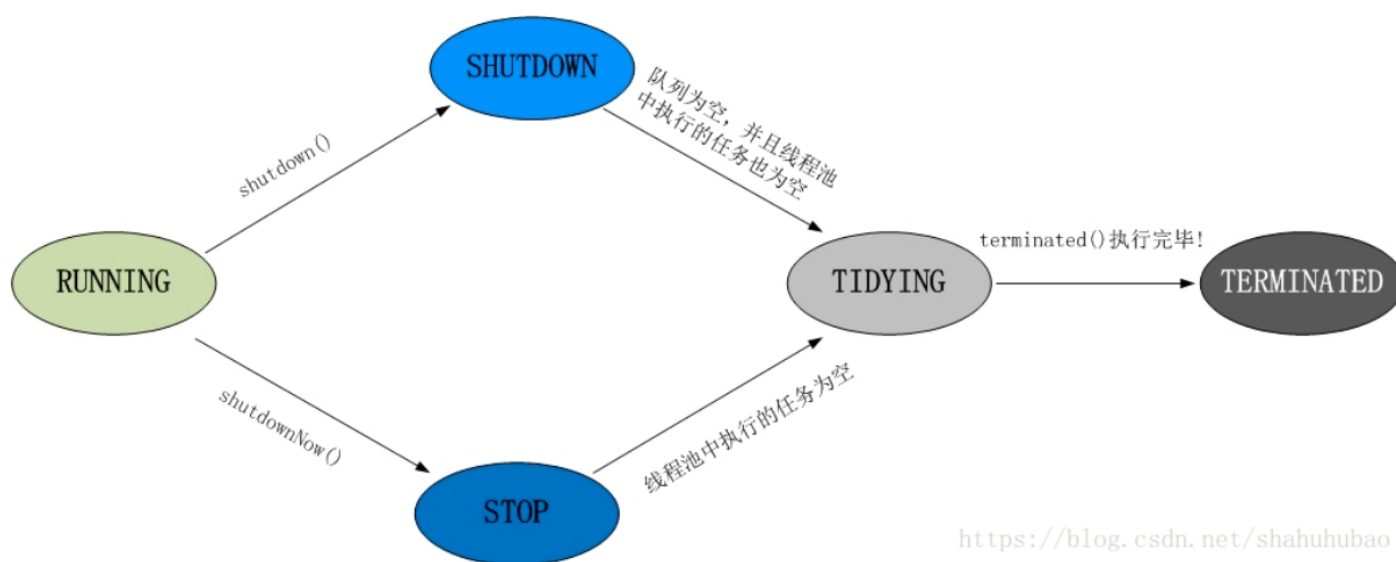
```
beforeExecute(wt, task);
Throwable thrown = null;
try {
    task.run();
} catch (RuntimeException x) {
    thrown = x; throw x;
} catch (Error x) {
    thrown = x; throw x;
} catch (Throwable x) {
    thrown = x; throw new Error(x);
} finally {
    afterExecute(task, thrown);
}
```

线程池状态

五种线程池状态

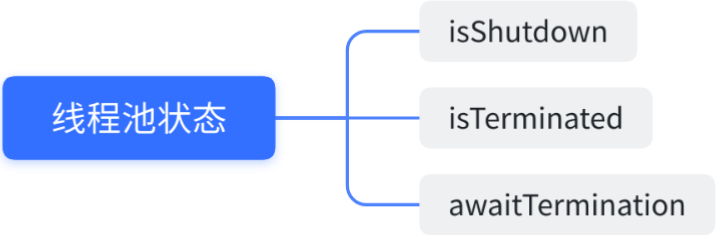


RUNNING	-1	接收新任务，处理排队任务
SHUTDOWN	0	不接受新任务，但处理排队任务
STOP	1	不接受新任务，也不执行排队任务，中断正在执行的线程
TIDYING	2	任务全部执行完毕，线程全部被销毁了，准备调用terminate钩子方法
TERMINATED	3	调用完terminate钩子方法，线程池生命周期结束



<https://blog.csdn.net/shahuhubao>

判断线程池状态



看源码才是王道，别人视频上看的不准，isShutdown说是判断是否调用了Shutdown

isShutdown	判断线程池是否正在停止或者已经停止	status >= 0
isTerminating	判断线程池是否正在停止	status >=0 && status < 3
isTerminated	判断线程池是否已经停止	status >= 3
awaitTermination	isTerminated的计时方法，判断线程池在指定时间内是否停止	

优雅关闭线程池

Java

```
1 public static void shutdownAndAwaitTermination(ExecutorService executor, long timeout, TimeUnit unit) {
2     executor.shutdown();
3     try {
4         // 指定时间还没关闭
5         if (!executor.awaitTermination(timeout, unit)) {
6             //立即关闭
7             executor.shutdownNow();
8             // 如果有工作线程正在执行非常耗时的任务，且无法中断
9             if (!executor.awaitTermination(timeout, unit)) {
10                 System.err.println("Pool did not terminate");
11             }
12         }
13     } catch (InterruptedException e){
14         // 如果当前线程被中断，并且捕获了中断信号，立即关闭线程池
15         executor.shutdownNow();
16         // 重新抛出中断信号，可能其他代码需要响应中断，所以重新抛出
17         Thread.currentThread().interrupt();
18     }
19 }
```

源码解析

核心属性和相关方法

Java

```
1 //ctl包含线程池的运行状态（高3位）和有效线程数信息（低29位）
2 private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
3 //有效线程数所占位数（29）
4 private static final int COUNT_BITS = Integer.SIZE - 3;
5 //理论上有效线程数的最大值
6 private static final int CAPACITY = (1 << COUNT_BITS) - 1;
7
8 /**线程池运行状态*/
9 //线程池能够处理新任务，并且处理队列任务
10 private static final int RUNNING = -1 << COUNT_BITS;
11 //线程池不接受新任务，但是会处理队列任务
12 private static final int SHUTDOWN = 0 << COUNT_BITS;
13 //线程池不接受新任务，也不处理队列任务，会中断进行中的任务
14 private static final int STOP = 1 << COUNT_BITS;
15 //线程池中所有任务结束，有效线程数为0
16 private static final int TIDYING = 2 << COUNT_BITS;
17 //线程池完成状态
18 private static final int TERMINATED = 3 << COUNT_BITS;
19
20 //从ctl中解析出线程池运行状态的方法
21 private static int runStateOf(int c) { return c & ~CAPACITY; }
22 //从ctl中解析出有效线程数的方法
23 private static int workerCountOf(int c) { return c & CAPACITY; }
24 //ctl的初始化方法
25 private static int ctlOf(int rs, int wc) { return rs | wc; }
```

execute(runnable)

```

1  public void execute(Runnable command) {
2      // 传入的任务为null，抛出空指针异常，不允许Runnable为null
3      if (command == null)
4          throw new NullPointerException();
5      // 获取线程池状态
6      int c = ctl.get();
7      // 核心逻辑1: 当前正在运行的线程数 < 核心线程数
8      if (workerCountOf(c) < corePoolSize) {
9          // 创建线程，并且使用新线程执行任务
10         if (addWorker(command, true))
11             return;
12         // 创建线程失败，重新获取线程池状态
13         c = ctl.get();
14     }
15     // 到这说明 正在运行的线程数 >= 核心线程数（也就是说，只要阻塞队列没满，就会先往阻塞
16     // 队列中添加任务，而不是去增加非核心线程数）
17     // 如果线程池是否处于RUNNABLE状态 && 添加任务阻塞队列中成功
18     // 核心逻辑2: （当前正在运行的线程数 >= 核心线程数）
19     if (isRunning(c) && workQueue.offer(command)) {
20         // 添加阻塞队列成功，进入方法
21         // 获取线程池状态，再次校验，防止第一次校验后，线程池关闭了??
22         int recheck = ctl.get();
23         // 线程池处于非运行状态 && 从队列中移除任务成功
24         if (! isRunning(recheck) && remove(command))
25             // 执行拒绝策略
26             reject(command);
27         // 如果正在运行的线程数=0，就创建一个线程?? 什么时候会走到这里?
28         else if (workerCountOf(recheck) == 0)
29             addWorker(null, false);
30     }
31     // 走到这里，说明阻塞队列满了，会去添加非核心线程，如果添加非核心线程失败，说明已经达
32     // 到最大线程数了，就执行拒绝策略
33     // 核心逻辑3: （当前正在运行的线程数 >= 核心线程数） && 添加阻塞队列失败
34     else if (!addWorker(command, false))
35         // 添加线程失败，进入方法
36         // 执行拒绝策略
37         reject(command);
38 }

```

Java

```
1  // core
2  private boolean addWorker(Runnable firstTask, boolean core) {
3      retry: //类似goto, 名字可以随便定义
4      for (;;) {
5          // 获取线程池状态
6          int c = ctl.get();
7          int rs = runStateOf(c);
8
9          // Check if queue empty only if necessary.
10         if (rs >= SHUTDOWN && ! (rs == SHUTDOWN && firstTask == null && !workQueue.isEmpty()))
11             return false;
12
13         for (;;) {
14             int wc = workerCountOf(c);
15             if (wc >= CAPACITY ||
16                 wc >= (core ? corePoolSize : maximumPoolSize))
17                 return false;
18             if (compareAndIncrementWorkerCount(c))
19                 break retry;
20             c = ctl.get(); // Re-read ctl
21             if (runStateOf(c) != rs)
22                 continue retry;
23             // else CAS failed due to workerCount change; retry inner loop
24         }
25     }
26
27     boolean workerStarted = false;
28     boolean workerAdded = false;
29     Worker w = null;
30     try {
31         // 对象内存放新进来的任务firstTask, 还有用线程工程去新建线程
32         w = new Worker(firstTask);
33         final Thread t = w.thread;
34         if (t != null) {
35             final ReentrantLock mainLock = this.mainLock;
36             mainLock.lock();
37             try {
38                 // Recheck while holding lock.
39                 // Back out on ThreadFactory failure or if
40                 // shut down before lock acquired.
41                 int rs = runStateOf(ctl.get());
```

```

42
43         if (rs < SHUTDOWN ||
44             (rs == SHUTDOWN && firstTask == null)) {
45             if (t.isAlive()) // precheck that t is startable
46                 throw new IllegalStateException();
47             workers.add(w);
48             int s = workers.size();
49             if (s > largestPoolSize)
50                 largestPoolSize = s;
51             workerAdded = true;
52         }
53     } finally {
54         mainLock.unlock();
55     }
56     if (workerAdded) {
57         t.start();
58         workerStarted = true;
59     }
60 }
61 } finally {
62     if (! workerStarted)
63         addWorkerFailed(w);
64 }
65 return workerStarted;
66 }

```

Worker

Java

```

1  private final class Worker
2      extends AbstractQueuedSynchronizer
3      implements Runnable
4  {
5      /**
6       * This class will never be serialized, but we provide a
7       * serialVersionUID to suppress a javac warning.
8       */
9      private static final long serialVersionUID = 6138294804551838833L;
10
11     /** Thread this worker is running in. Null if factory fails. */
12     final Thread thread;
13     /** Initial task to run. Possibly null. */
14     Runnable firstTask;

```

```

15  /** Per-thread task counter */
16  volatile long completedTasks;
17
18  /**
19   * Creates with given first task and thread from ThreadFactory.
20   * @param firstTask the first task (null if none)
21   */
22  Worker(Runnable firstTask) {
23      setState(-1); // inhibit interrupts until runWorker
24      this.firstTask = firstTask;
25      this.thread = getThreadFactory().newThread(this);
26  }
27
28  /** Delegates main run loop to outer runWorker */
29  public void run() {
30      runWorker(this);
31  }
32
33  protected boolean isHeldExclusively() {
34      return getState() != 0;
35  }
36
37  protected boolean tryAcquire(int unused) {
38      if (compareAndSetState(0, 1)) {
39          setExclusiveOwnerThread(Thread.currentThread());
40          return true;
41      }
42      return false;
43  }
44
45  protected boolean tryRelease(int unused) {
46      setExclusiveOwnerThread(null);
47      setState(0);
48      return true;
49  }
50
51  public void lock()      { acquire(1); }
52  public boolean tryLock() { return tryAcquire(1); }
53  public void unlock()    { release(1); }
54  public boolean isLocked() { return isHeldExclusively(); }
55
56  void interruptIfStarted() {
57      Thread t;
58      if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {

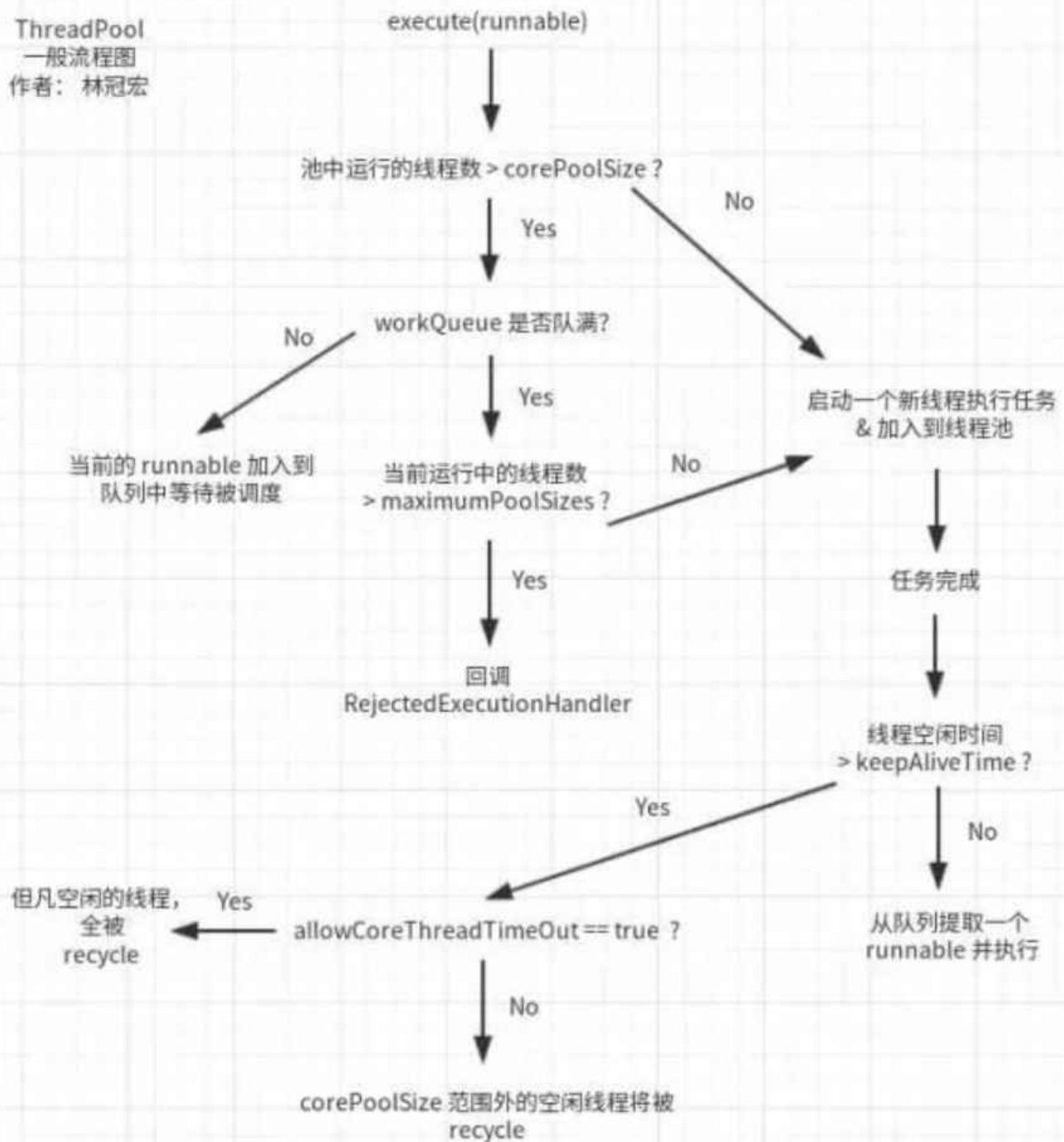
```

```

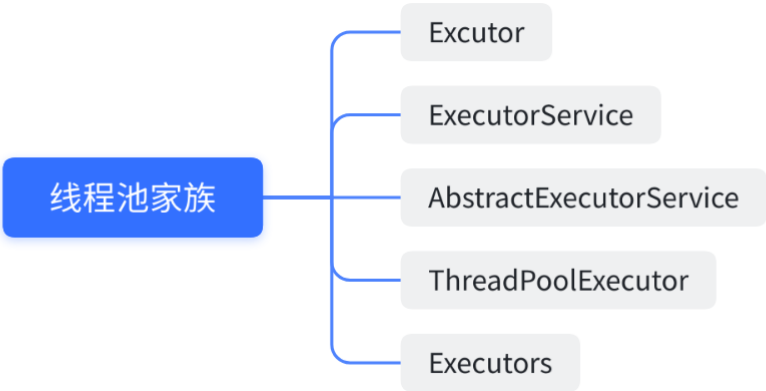
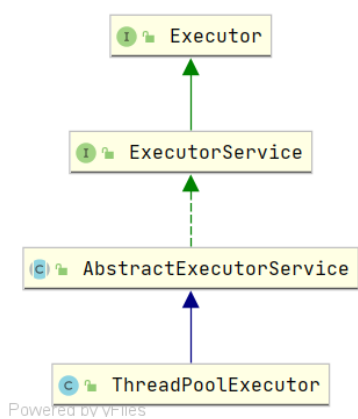
59         try {
60             t.interrupt();
61         } catch (SecurityException ignore) {
62         }
63     }
64 }
65 }

```

ThreadPool
一般流程图
作者：林冠宏



线程池家族



Excutor	顶级接口，只提供了一个execute方法
ExecutorService	扩展接口，提供了线程池核心方法，比如submit、shutdown、shutdownNow等方法
AbstractExecutorService	抽象类，主要实现了submit方法，其他抽象方法交给子类实现
ThreadPoolExecutor	线程池实现类
Executors	线程池工具类，主要提供了一些创建线程池的方法

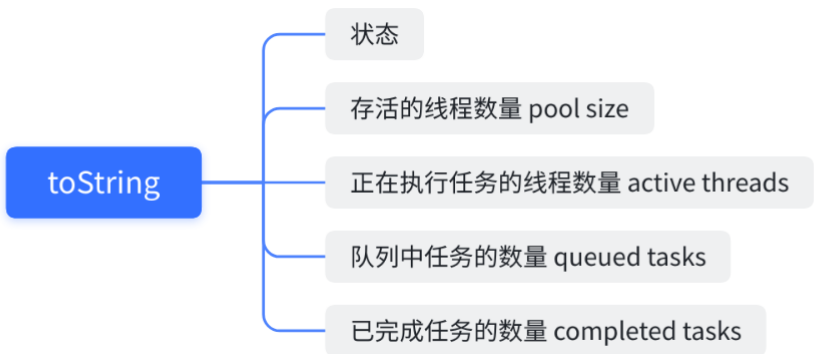
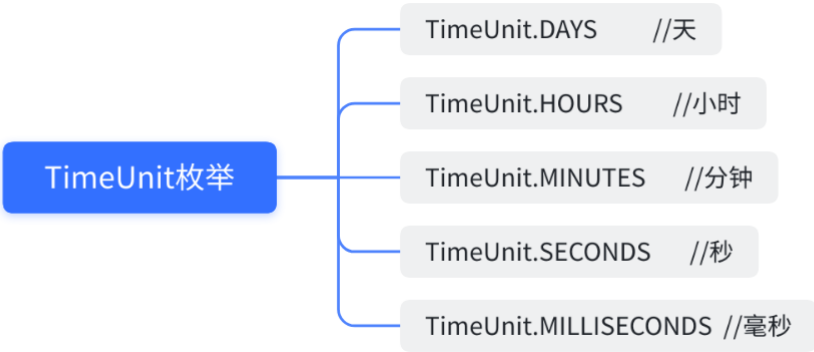
线程池调优

合理配置线程池数量

cpu密集型	大量的计算，最佳线程数为CPU核心数+1，没必要频繁去切换线程。
IO密集型	大量的IO，一般线程数是CPU核心数的很多倍，最佳线程数=cpu核心数*（1+平均等待时间（IO耗时）/平均工作时间（CPU耗时）），保证线程空闲时间可以衔接上。

实际场景举例 TODO

其他



总结

线程池东西少，个人

学习过程

- 1、博客找源码，知道哪些是核心属性和核心方法（只学习核心的）
- 2、分析核心方法在正常情况下的核心步骤
- 3、更多细节看书籍，并发编程之美、并发编程实战、并发编程详解，对比，总结

小知识

1. 线程池中只有一把锁，但是每个工作线程都有一个锁，实现了aqs，只有在run方法里面有用到这个锁
 2. HashSet<Worker> workers 使用hashset装线程
- ◦ ◦

问题

1. 线程池有哪些实际应用场景？
2. 为什么要设计最大线程数和核心线程数？
3. 如何自定义拒绝策略和线程工厂？
4. shutdownNow中断后的线程怎么处理？
内部自己处理中断信号，可能继续执行，可能停止线程
5. submit和executor的区别
6. 线程池是如何实现线程复用的？
7. `workerCountOf(c)` //好像是获取某个状态的线程数量
8. 线程池死锁问题
任务之间互相依赖，正在执行的任务，依赖未执行的任务，未执行的任务被shutdownnow返回了？？？
9. 状态验证Demo
最大的线程数是合理的线程数