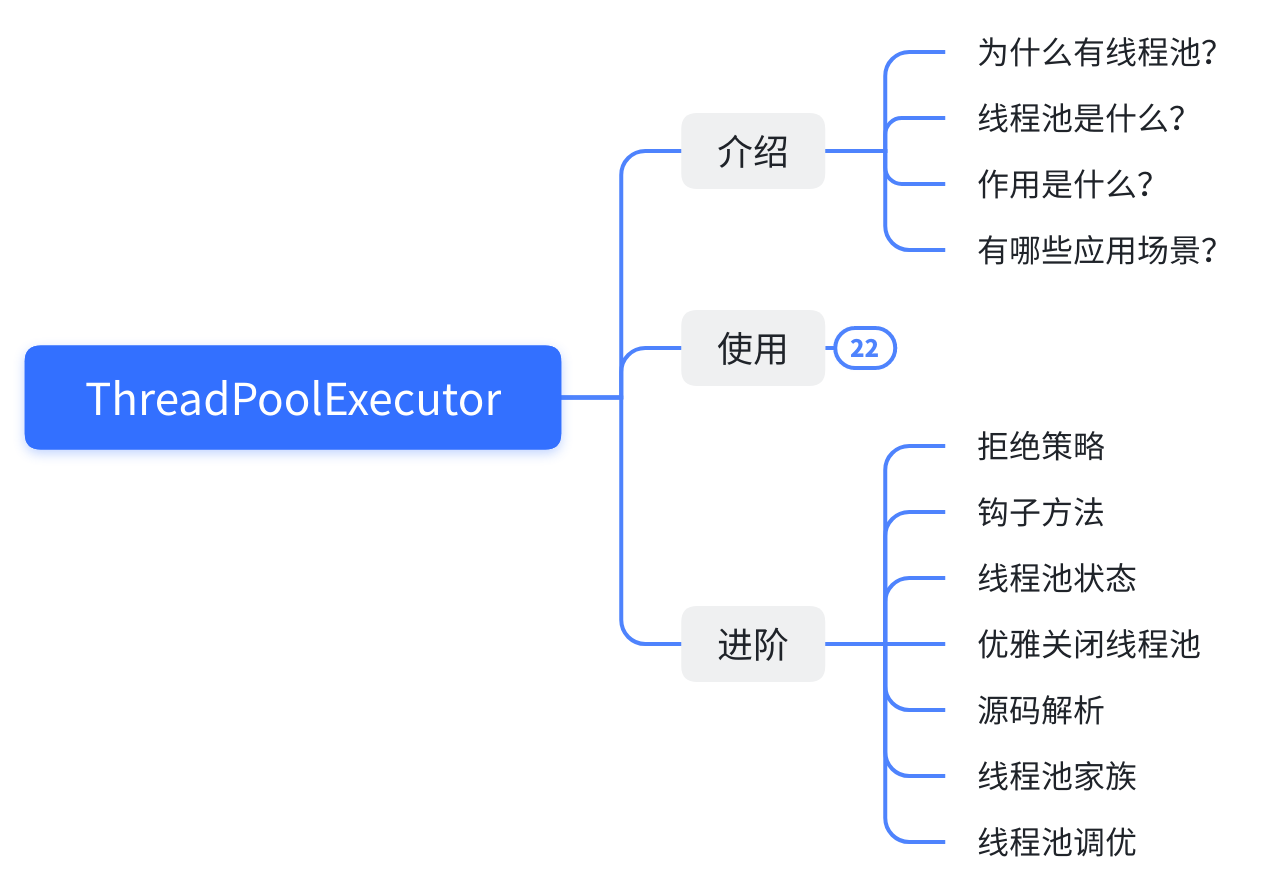
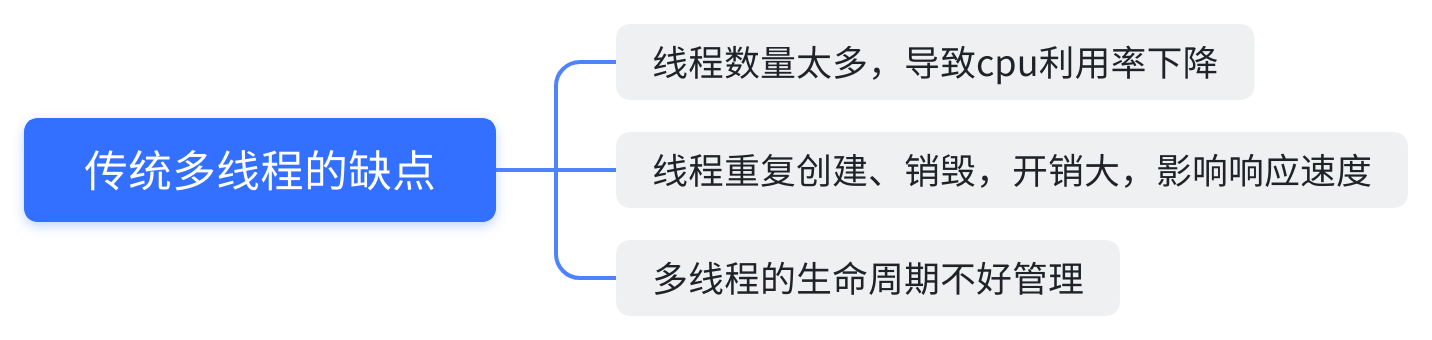
**ThreadPoolExecutor**

官方文档：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：

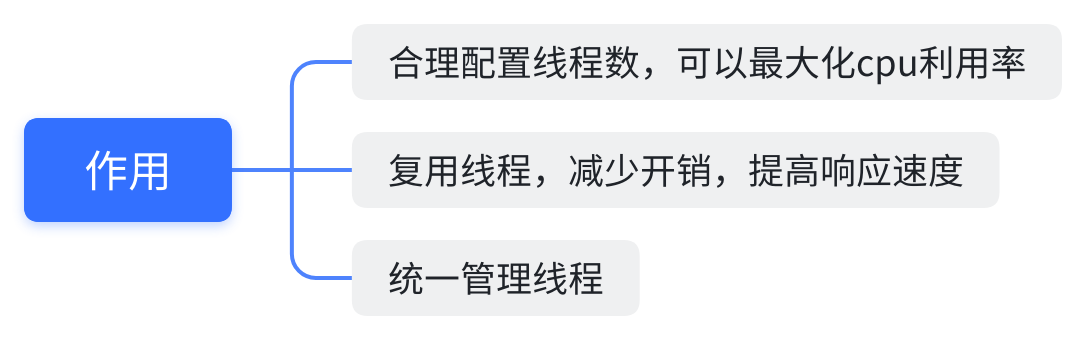
如果每到达一个请求，就去创建一个线程，这样线程数量是特别多的。线程上下文频繁切换，会导致cpu利用率下降。而且每个线程都需要不停的创建和销毁，这样开销是特别大的，会影响响应速度。

**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

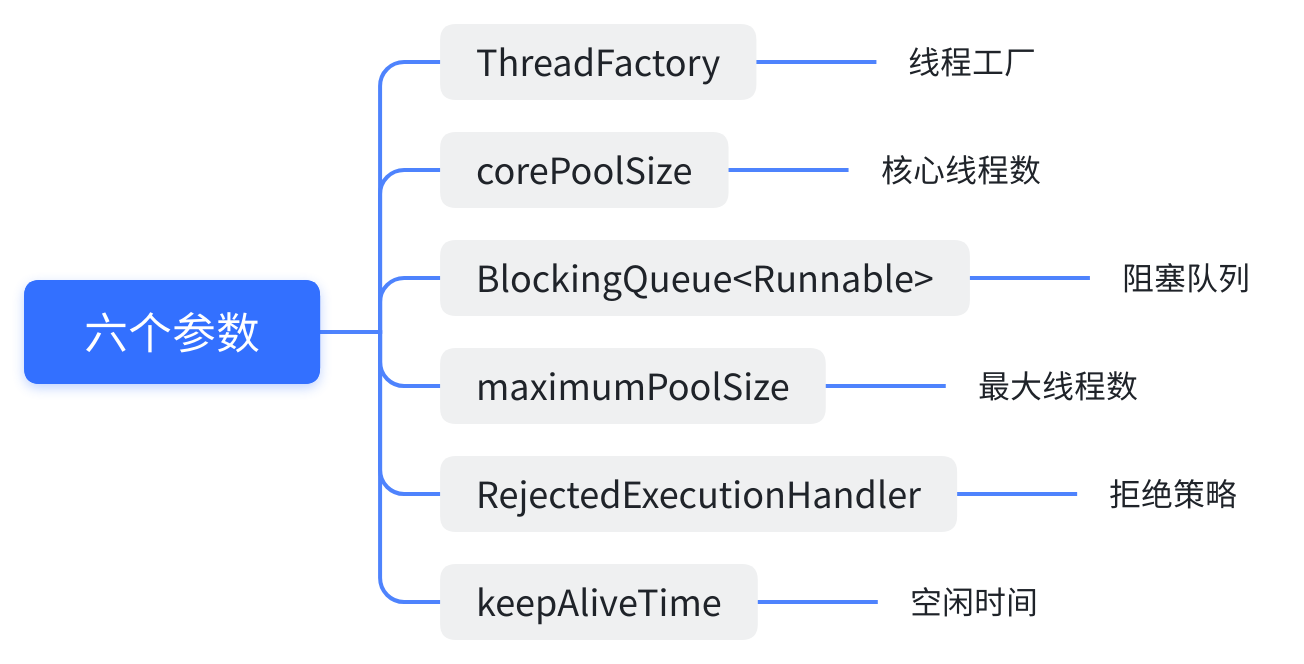


**应用场景**

根据不同的线程池类型，有不同的应用场景，后面会介绍

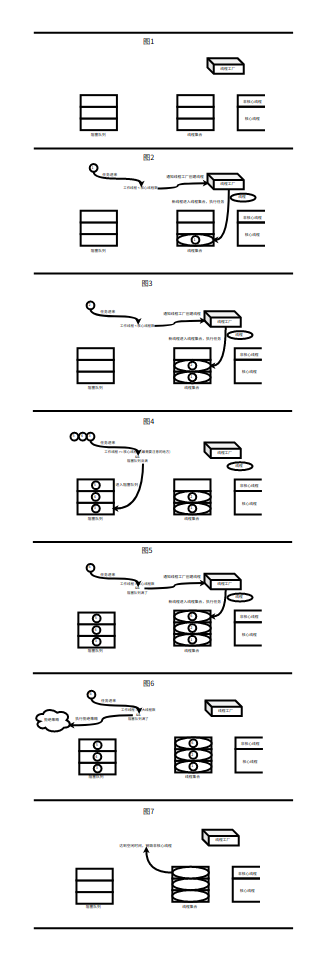
**使用**

**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**

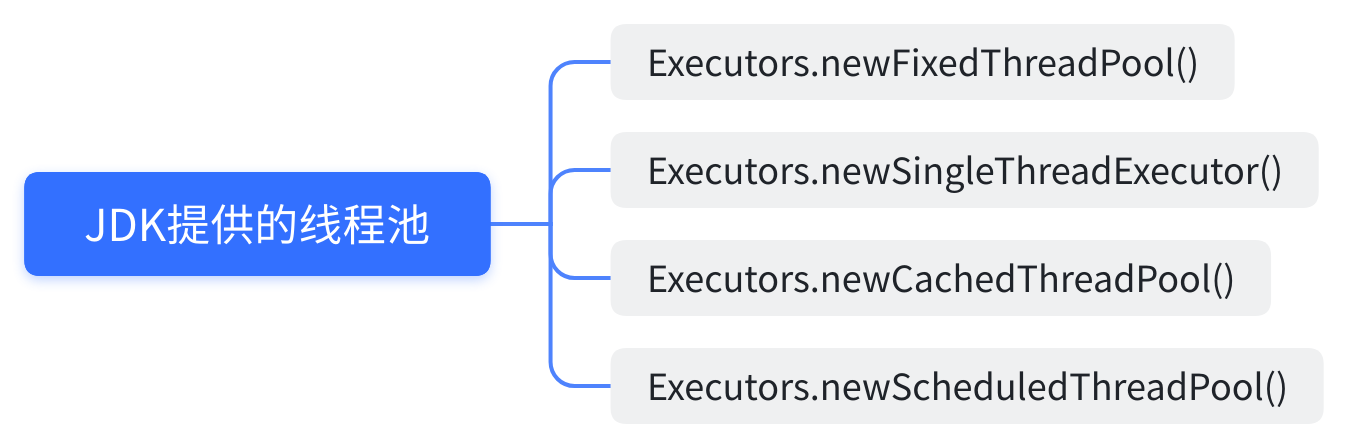


在使用线程池之前，先介绍一下线程池的参数

|  |  |
| --- | --- |
| 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 |
| **SingleThreadExecutor** | 单线程 | 按入队顺序执行任务 | 1 | 1 | 0 | LinkedBlockingQueue |
| **CachedThreadPool** | 可缓存 | 适合任务执行速度快的 | 0 | Integer.MAX\_VALUE | 60s | SynchronousQueue |
| **ScheduledThreadPool** | 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 | DelayedWorkQueue |

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

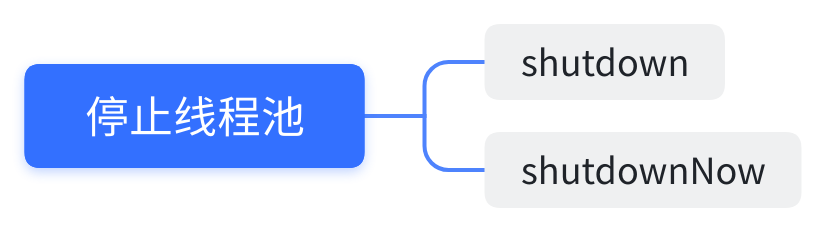
**启动线程池**

**execute(runnable)**

会发现JVM不会退出，因为线程池还在运行

|  |
| --- |
| Java  public static void executeDemo() {  Runnable runnable = () -> {  System.*out*.println(Thread.*currentThread*().getName() + "执行任务");  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  };    ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5,  10,  2L,  TimeUnit.*MILLISECONDS*,  new ArrayBlockingQueue<Runnable>(10),  Executors.*defaultThreadFactory*(),  new ThreadPoolExecutor.AbortPolicy());  threadPoolExecutor.allowCoreThreadTimeOut(false);  *// 证明不是线程池创建的时候就马上创建线程的*  System.*out*.println(threadPoolExecutor.toString());  *// 演示核心线程数*  *// for (int i = 0; i < 15; i++) {*  *// threadPoolExecutor.execute(runnable);*  *// }*  *// 演示最大线程数*  *// for (int i = 0; i < 16; i++) {*  *// threadPoolExecutor.execute(runnable);*  *// }*    *// 演示拒绝策略*  for (int i = 0; i < 21; i++) {  threadPoolExecutor.execute(runnable);  }  *//证明线程数达到最大线程*  System.*out*.println(threadPoolExecutor.toString());  try {  Thread.*sleep*(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  *//证明线程数减少到核心线程*  System.*out*.println(threadPoolExecutor.toString());  } |

**停止线程池**



|  |  |
| --- | --- |
| 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  public static void shutdownDemo() throws InterruptedException {  *// 创建任务*  Runnable runnable = () -> {  System.*out*.println(Thread.*currentThread*().getName() + "执行任务");  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  System.*out*.println(Thread.*currentThread*().getName() + "被中断唤醒");  }  };  *// 创建线程池*  ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5,  10,  2L,  TimeUnit.*MILLISECONDS*,  new ArrayBlockingQueue<Runnable>(10),  Executors.*defaultThreadFactory*(),  new MyRejectedExecutionHandler());  threadPoolExecutor.allowCoreThreadTimeOut(false);  *// 打印线程池初始状态*  System.*out*.println(threadPoolExecutor.toString());  for (int i = 0; i < 20; i++) {  threadPoolExecutor.execute(runnable);  }  *// 打印线程池正常工作情况*  System.*out*.println(threadPoolExecutor.toString());  *// 把所有任务都完成*  *threadPoolExecutor.shutdown();*  *// 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*  *//* List<Runnable> runnables = threadPoolExecutor.shutdownNow();  *// 线程停止后，再有任务进来，抛出拒绝异常*  threadPoolExecutor.execute(runnable);  *// 证明停止后不会再接收任务*  try {  threadPoolExecutor.execute(runnable);  } catch (Exception e) {  System.*out*.println("异常被catch住了");  }  *// 让任务都执行完*  Thread.*sleep*(5000);  *// 打印线程池停止后的情况*  System.*out*.println(threadPoolExecutor.toString());  *// System.out.println( threadPoolExecutor.isShutdown());*  *// System.out.println( threadPoolExecutor.isTerminated());*  *// System.out.println( threadPoolExecutor.isTerminating());*    } |

**shutdownNow()立即关闭**

|  |
| --- |
| D  public static void shutdownDemo() throws InterruptedException {  *// 创建任务*  Runnable runnable = () -> {  System.*out*.println(Thread.*currentThread*().getName() + "执行任务");  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  System.*out*.println(Thread.*currentThread*().getName() + "被中断唤醒");  }  };  *// 创建线程池*  ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(5,  10,  2L,  TimeUnit.*MILLISECONDS*,  new ArrayBlockingQueue<Runnable>(10),  Executors.*defaultThreadFactory*(),  new MyRejectedExecutionHandler());  threadPoolExecutor.allowCoreThreadTimeOut(false);  *// 打印线程池初始状态*  System.*out*.println(threadPoolExecutor.toString());  for (int i = 0; i < 20; i++) {  threadPoolExecutor.execute(runnable);  }  *// 打印线程池正常工作情况*  System.*out*.println(threadPoolExecutor.toString());  *// 把所有任务都完成*  *// threadPoolExecutor.shutdown();*  *// 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*  List<Runnable> runnables = threadPoolExecutor.shutdownNow();  *// 线程停止后，再有任务进来，抛出拒绝异常*  threadPoolExecutor.execute(runnable);  *// 证明停止后不会再接收任务*  try {  threadPoolExecutor.execute(runnable);  } catch (Exception e) {  System.*out*.println("异常被catch住了");  }  *// 让任务都执行完*  Thread.*sleep*(5000);  *// 打印线程池停止后的情况*  System.*out*.println(threadPoolExecutor.toString());  *// System.out.println( threadPoolExecutor.isShutdown());*  *// System.out.println( threadPoolExecutor.isTerminated());*  *// System.out.println( threadPoolExecutor.isTerminating());*    } |

**自动关闭线程池**

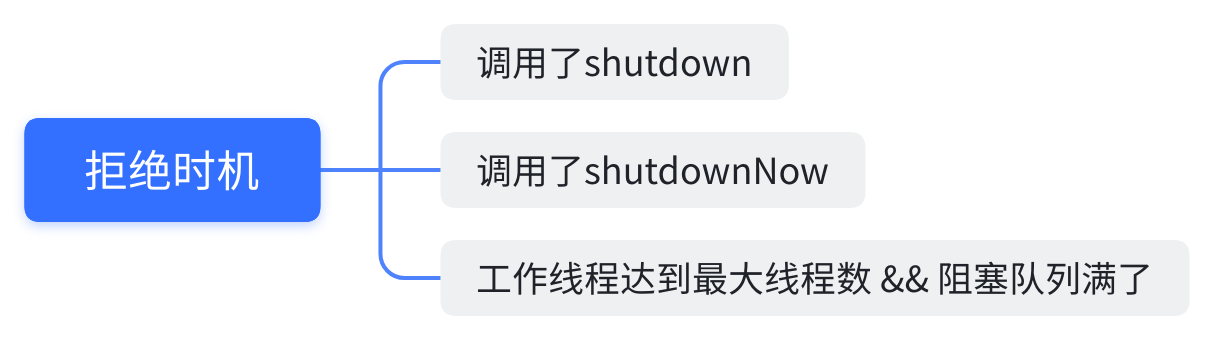
被gc回收的时候

|  |
| --- |
| Java  protected void finalize() {  SecurityManager sm = System.*getSecurityManager*();  if (sm == null || acc == null) {  // 停止线程池  shutdown();  } else {  PrivilegedAction<Void> pa = () -> { shutdown(); return null; };  AccessController.*doPrivileged*(pa, acc);  }  } |

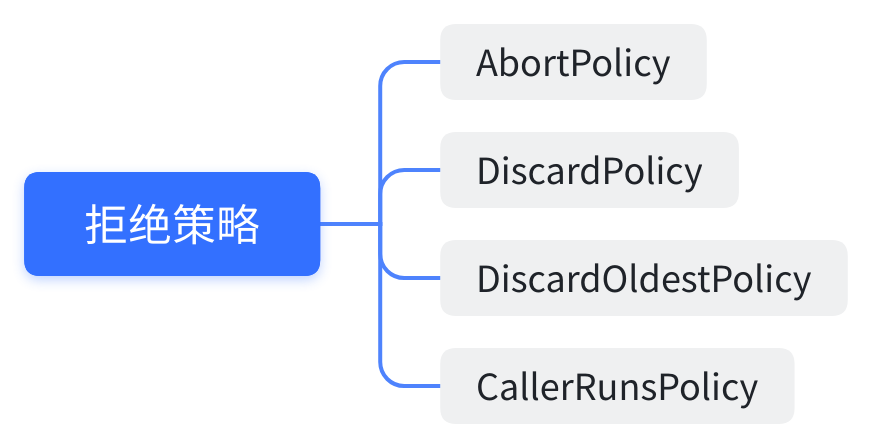
**进阶**

**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 |
| DiscardOldestPolicy | 丢弃旧的任务 |  |
| CallerRunsPolicy | 使用提交任务的线程执行任务 |  |

|  |
| --- |
| Java  if (!e.isShutdown()) {  e.getQueue().poll();//出队列  e.execute(r);//出完队列重新执行excute方法，通常是被放到阻塞队列中  } |

|  |
| --- |
| Java  if (!e.isShutdown()) {  r.run();//当前线程直接调用runnable的方法  } |

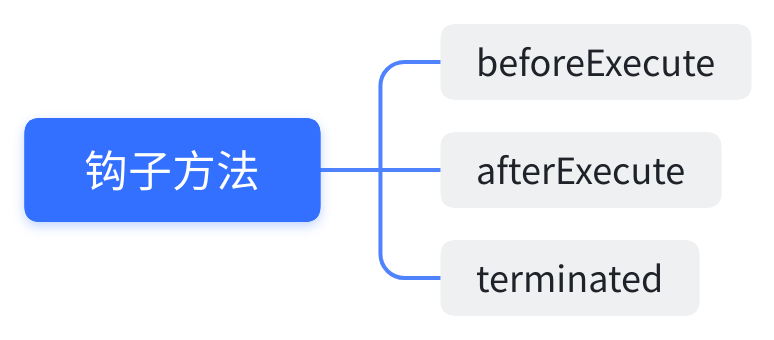
**自定义拒绝策略**

实现RejectedExecutionHandler接口，自定义策略

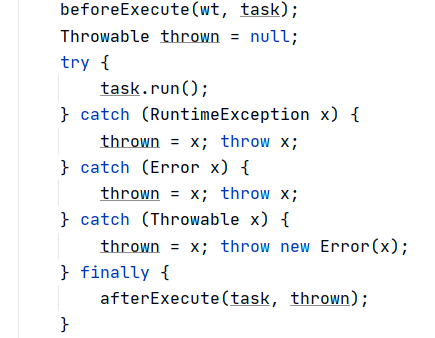
|  |
| --- |
| Java  static class MyRejectedExecutionHandler implements RejectedExecutionHandler {  @Override  public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {  System.*out*.println("这个任务我不干！");  }  } |

**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

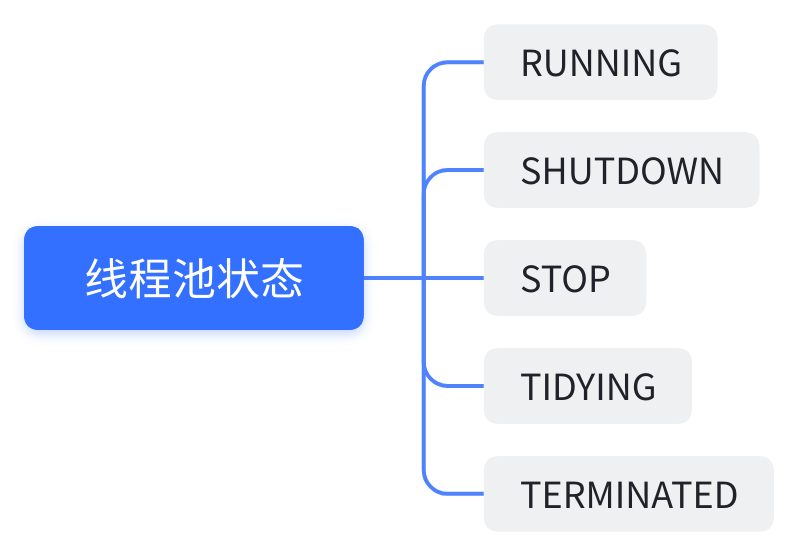


|  |  |
| --- | --- |
| beforeExecute | 任务执行前 |
| afterExecute | 任务执行后 |
| terminated | 线程池结束后 |

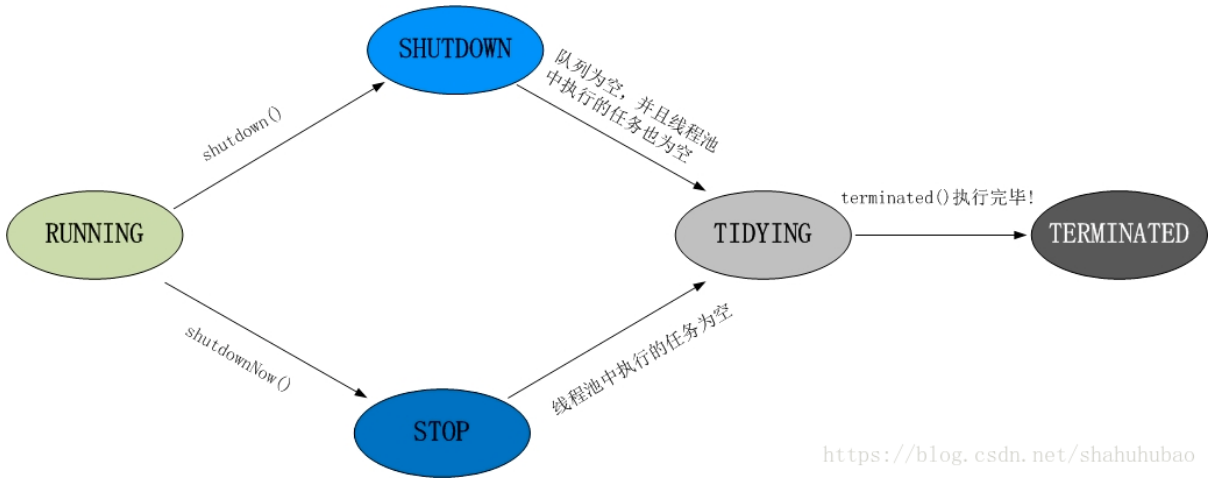


**线程池状态**

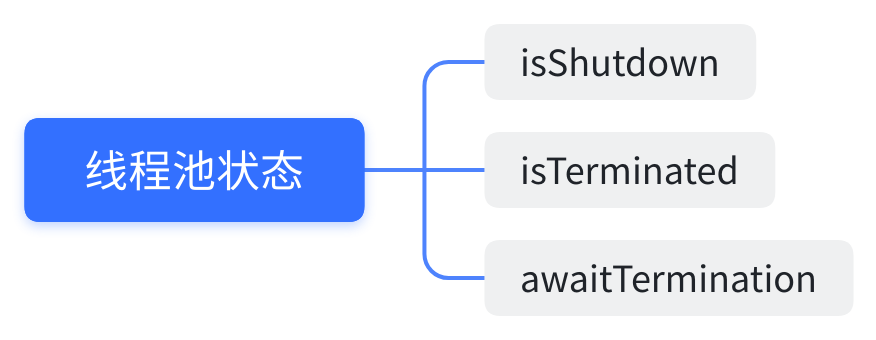
**五种线程池状态**



|  |  |  |
| --- | --- | --- |
| 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 | 不接收新任务，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 |



**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  public static void shutdownAndAwaitTermination(ExecutorService executor, long timeout, TimeUnit unit) {  executor.shutdown();  try {  *// 指定时间还没关闭*  if (!executor.awaitTermination(timeout, unit)) {  *//立即关闭*  executor.shutdownNow();  *// 如果有工作线程正在执行非常耗时的任务，且无法中断*  if (!executor.awaitTermination(timeout, unit)) {  System.*err*.println("Pool did not terminate");  }  }  }catch (InterruptedException e){  *// 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*  executor.shutdownNow();  *// 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*  Thread.*currentThread*().interrupt();  }  } |

**源码解析**

**核心属性和相关方法**

|  |
| --- |
| Java  //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）  private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));  //有效线程数所占位数（29）  private static final int COUNT\_BITS = Integer.SIZE - 3;  //理论上有效线程数的最大值  private static final int CAPACITY = (1 << COUNT\_BITS) - 1;    /\*\*线程池运行状态\*/  //线程池能够处理新任务，并且处理队列任务  private static final int RUNNING = -1 << COUNT\_BITS;  //线程池不接受新任务，但是会处理队列任务  private static final int SHUTDOWN = 0 << COUNT\_BITS;  //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  private static final int STOP = 1 << COUNT\_BITS;  //线程池中所有任务结束，有效线程数为0  private static final int TIDYING = 2 << COUNT\_BITS;  //线程池完成状态  private static final int TERMINATED = 3 << COUNT\_BITS;    //从ctl中解析出线程池运行状态的方法  private static int runStateOf(int c) { return c & ~CAPACITY; }  //从ctl中解析出有效线程数的方法  private static int workerCountOf(int c) { return c & CAPACITY; }  //ctl的初始化方法  private static int ctlOf(int rs, int wc) { return rs | wc; } |

**execute(runnable)**

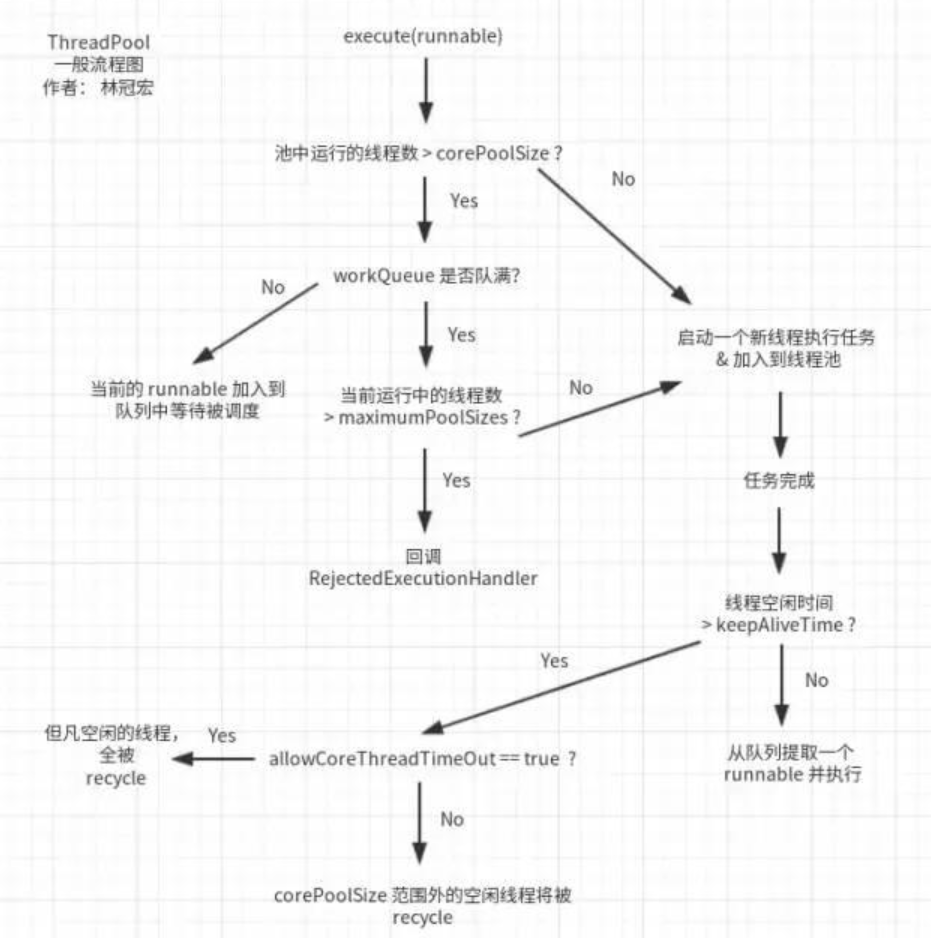
|  |
| --- |
| Java  public void execute(Runnable command) {  // 传入的任务为null，抛出空指针异常，不允许Runnable为null  if (command == null)  throw new NullPointerException();  // 获取线程池状态  int c = ctl.get();  // 核心逻辑1：当前正在运行的线程数 < 核心线程数  if (*workerCountOf*(c) < corePoolSize) {  // 创建线程，并且使用新线程执行任务  if (addWorker(command, true))  return;  // 创建线程失败，重新获取线程池状态  c = ctl.get();  }  // 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，就会先往阻塞队列中添加任务，而不是去增加非核心线程数）  // 如果线程池是否处于RUNNABLE状态 && 添加任务阻塞队列中成功  // 核心逻辑2：（当前正在运行的线程数 >= 核心线程数）  if (*isRunning*(c) && workQueue.offer(command)) {  //添加阻塞队列成功，进入方法  // 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？？？  int recheck = ctl.get();  // 线程池处于非运行状态 && 从队列中移除任务成功  if (! *isRunning*(recheck) && remove(command))  // 执行拒绝策略  reject(command);  // 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？  else if (*workerCountOf*(recheck) == 0)  addWorker(null, false);  }  // 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，说明已经达到最大线程数了，就执行拒绝策略  // 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  else if (!addWorker(command, false))  // 添加线程失败，进入方法  // 执行拒绝策略  reject(command);  } |

**addWorker**

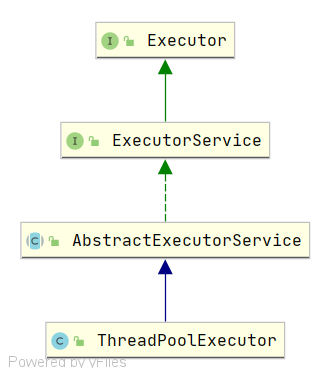
|  |
| --- |
| Java  // core  private boolean addWorker(Runnable firstTask, boolean core) {  retry: //类似goto，名字可以随便定义  for (;;) {  // 获取线程池状态  int c = ctl.get();  int rs = *runStateOf*(c);    *// Check if queue empty only if necessary.*  if (rs >= *SHUTDOWN* && ! (rs == *SHUTDOWN* && firstTask == null && !workQueue.isEmpty()))  return false;    for (;;) {  int wc = *workerCountOf*(c);  if (wc >= *CAPACITY* ||  wc >= (core ? corePoolSize : maximumPoolSize))  return false;  if (compareAndIncrementWorkerCount(c))  break retry;  c = ctl.get(); *// Re-read ctl*  if (*runStateOf*(c) != rs)  continue retry;  *// else CAS failed due to workerCount change; retry inner loop*  }  }    boolean workerStarted = false;  boolean workerAdded = false;  Worker w = null;  try {  // 对象内存放新进来的任务firstTask，还有用线程工程去新建线程  w = new Worker(firstTask);  final Thread t = w.thread;  if (t != null) {  final ReentrantLock mainLock = this.mainLock;  mainLock.lock();  try {  *// Recheck while holding lock.*  *// Back out on ThreadFactory failure or if*  *// shut down before lock acquired.*  int rs = *runStateOf*(ctl.get());    if (rs < *SHUTDOWN* ||  (rs == *SHUTDOWN* && firstTask == null)) {  if (t.isAlive()) *// precheck that t is startable*  throw new IllegalThreadStateException();  workers.add(w);  int s = workers.size();  if (s > largestPoolSize)  largestPoolSize = s;  workerAdded = true;  }  } finally {  mainLock.unlock();  }  if (workerAdded) {  t.start();  workerStarted = true;  }  }  } finally {  if (! workerStarted)  addWorkerFailed(w);  }  return workerStarted;  } |

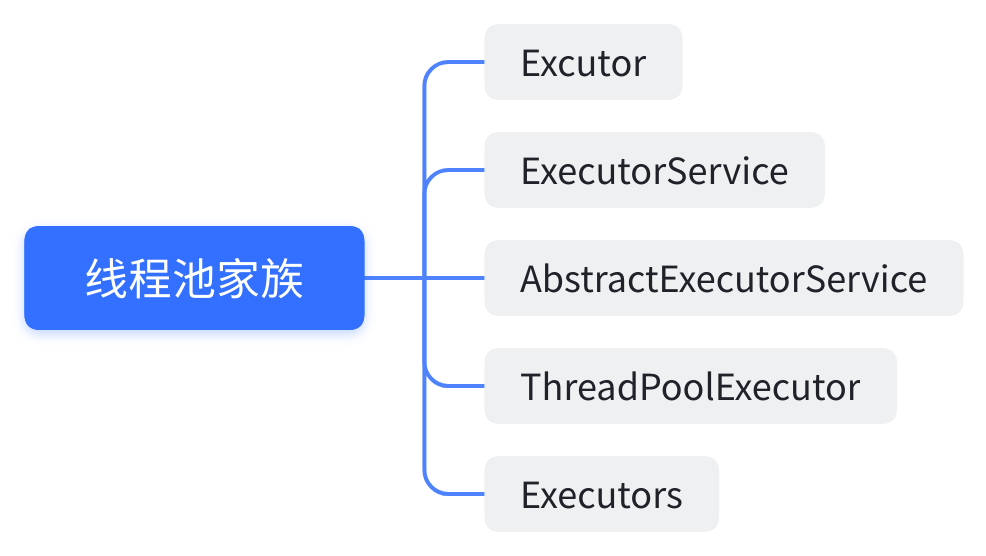
**Worker**

|  |
| --- |
| Java  private final class Worker  extends AbstractQueuedSynchronizer  implements Runnable  {  */\*\**  *\* This class will never be serialized, but we provide a*  *\* serialVersionUID to suppress a javac warning.*  *\*/*  private static final long *serialVersionUID* = 6138294804551838833L;    */\*\* Thread this worker is running in. Null if factory fails. \*/*  final Thread thread;  */\*\* Initial task to run. Possibly null. \*/*  Runnable firstTask;  */\*\* Per-thread task counter \*/*  volatile long completedTasks;    */\*\**  *\* Creates with given first task and thread from ThreadFactory.*  *\* @param firstTask the first task (null if none)*  *\*/*  Worker(Runnable firstTask) {  setState(-1); *// inhibit interrupts until runWorker*  this.firstTask = firstTask;  this.thread = getThreadFactory().newThread(this);  }    */\*\* Delegates main run loop to outer runWorker \*/*  public void run() {  runWorker(this);  }    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(); }    void interruptIfStarted() {  Thread t;  if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {  try {  t.interrupt();  } catch (SecurityException ignore) {  }  }  }  } |



**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、shutdowNow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 |

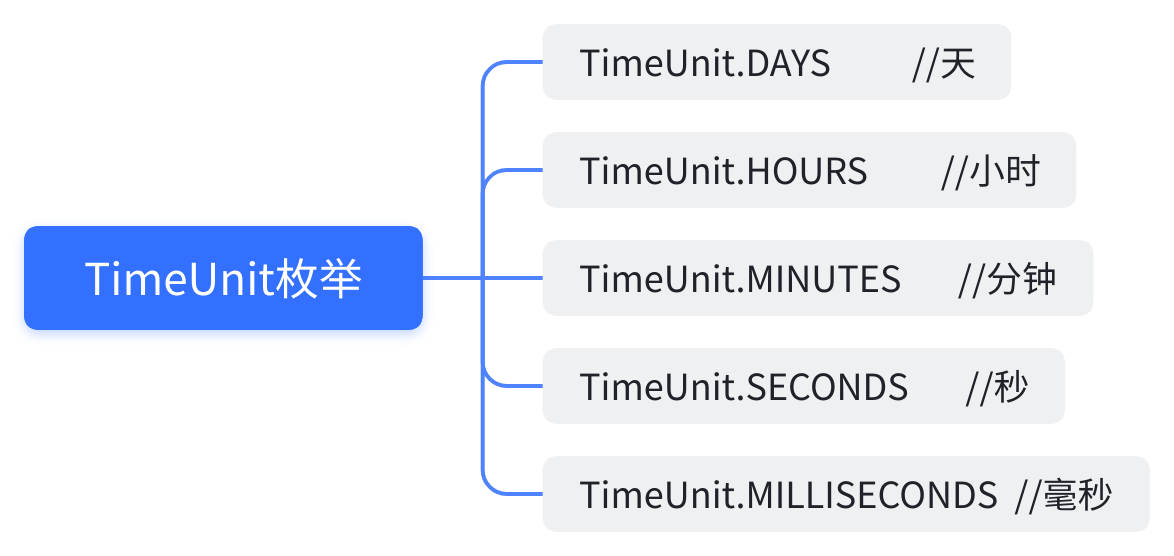
**线程池调优**

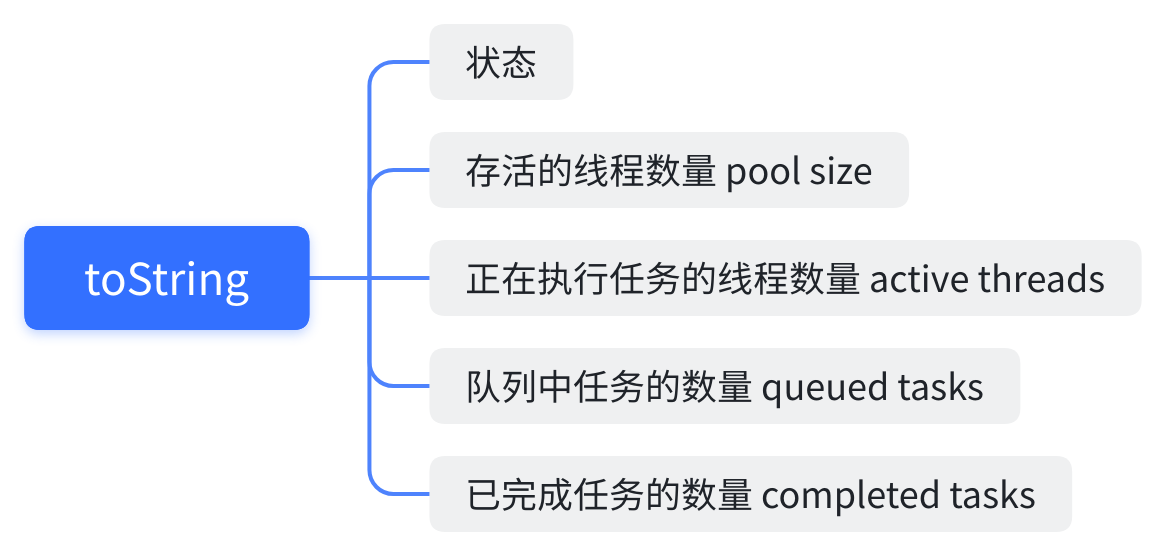
**合理配置线程池数量**

|  |  |
| --- | --- |
| 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

1. submit和executor的区别
2. 线程池是如何实现线程复用的？
3. *workerCountOf*(c) //好像是获取某个状态的线程数量
4. 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shutdowmnow返回了？？？

1. 状态验证Demo

最大的线程数是合理的线程数