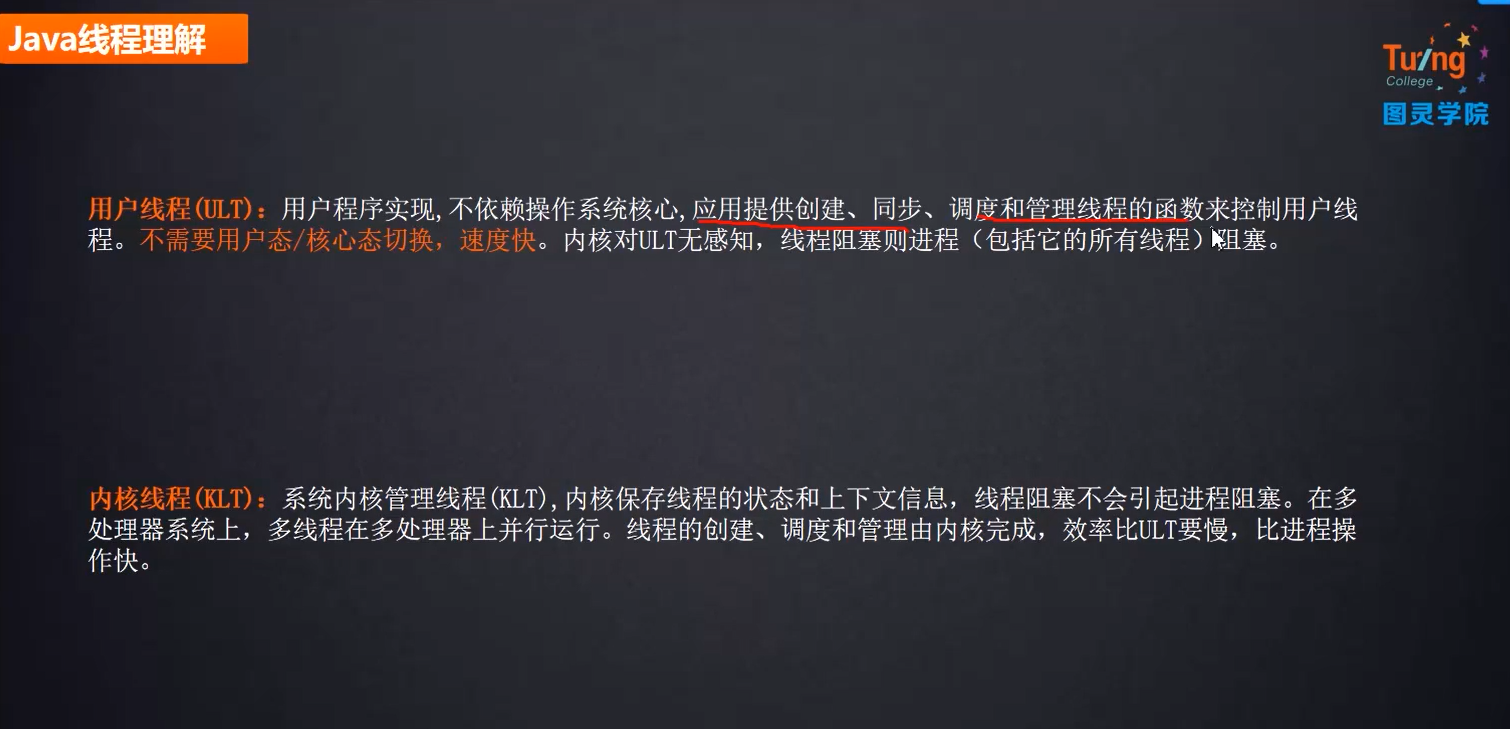
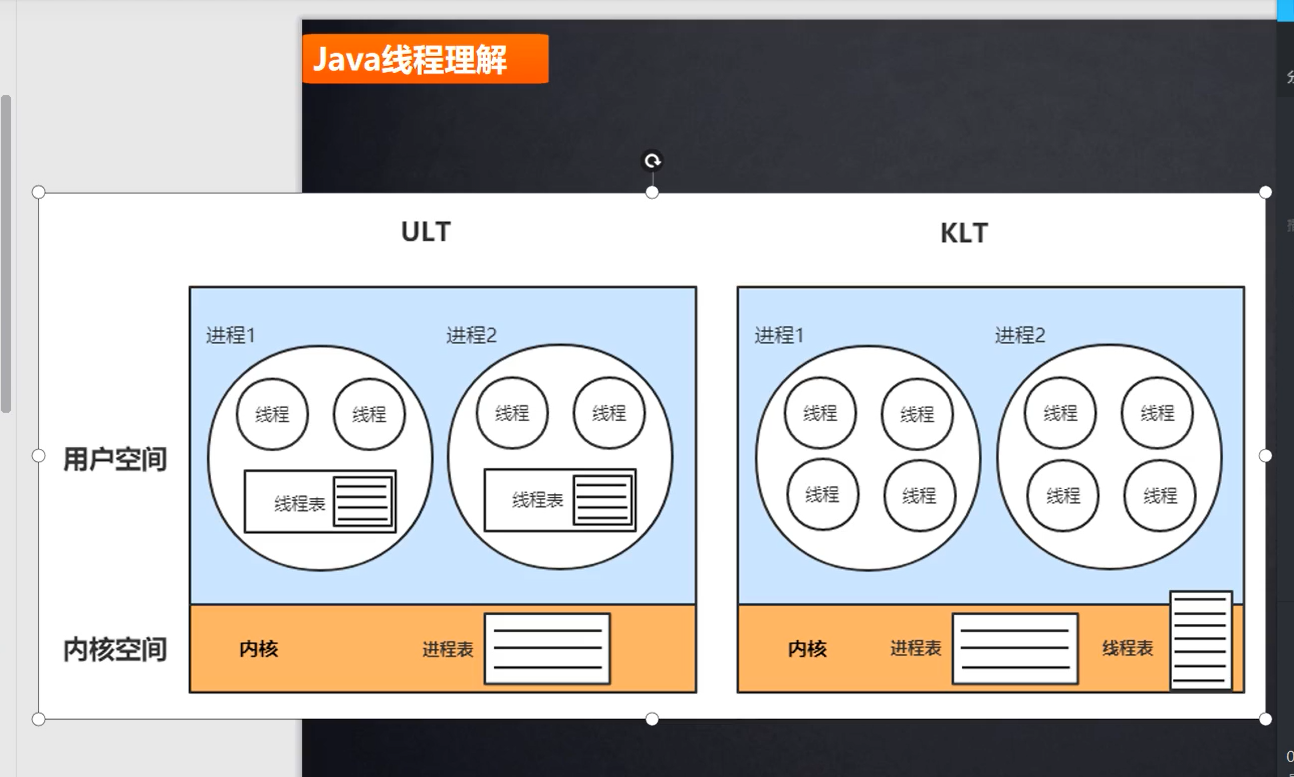
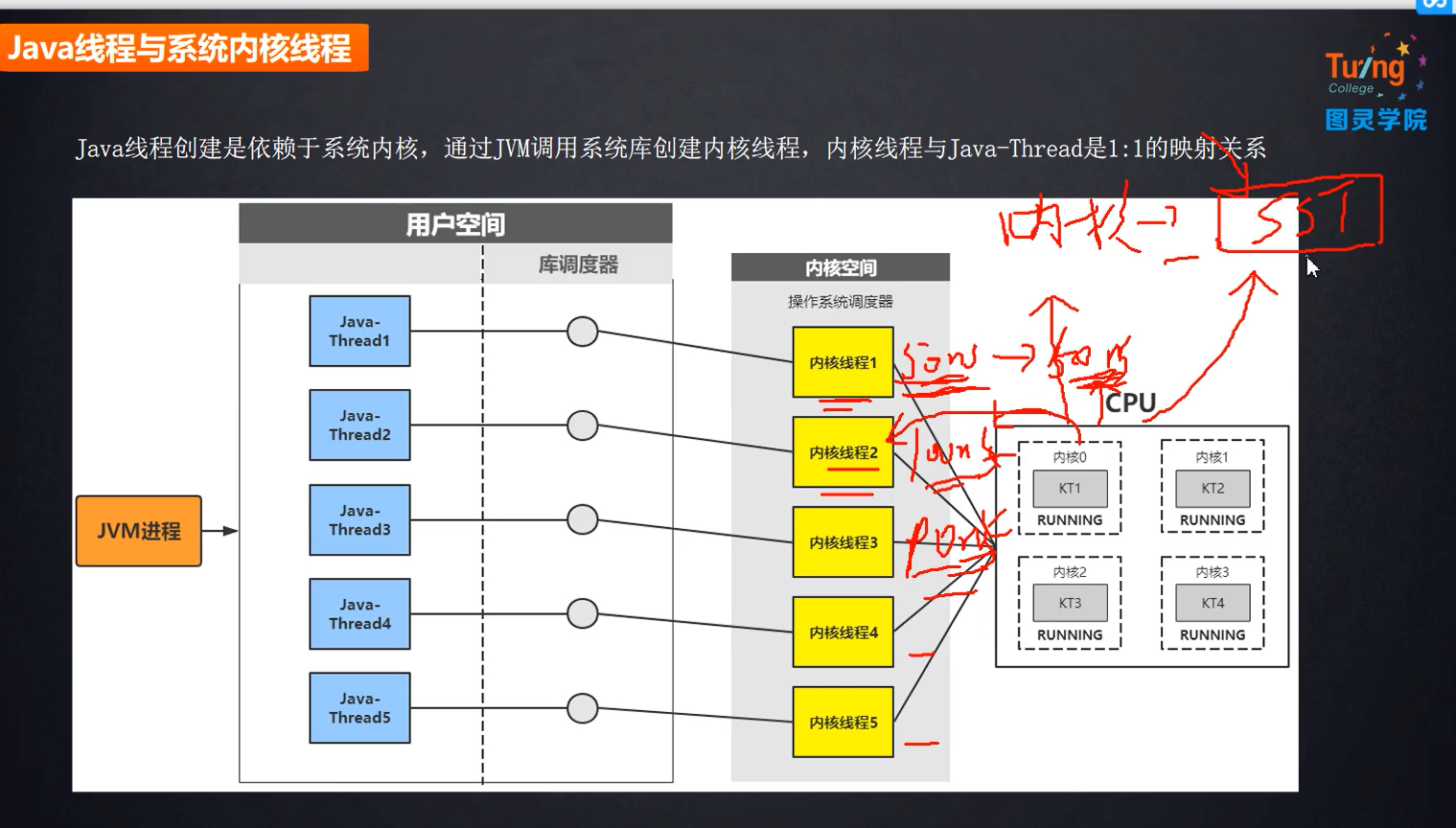
# 线程和线程池

#### 1,线程分类

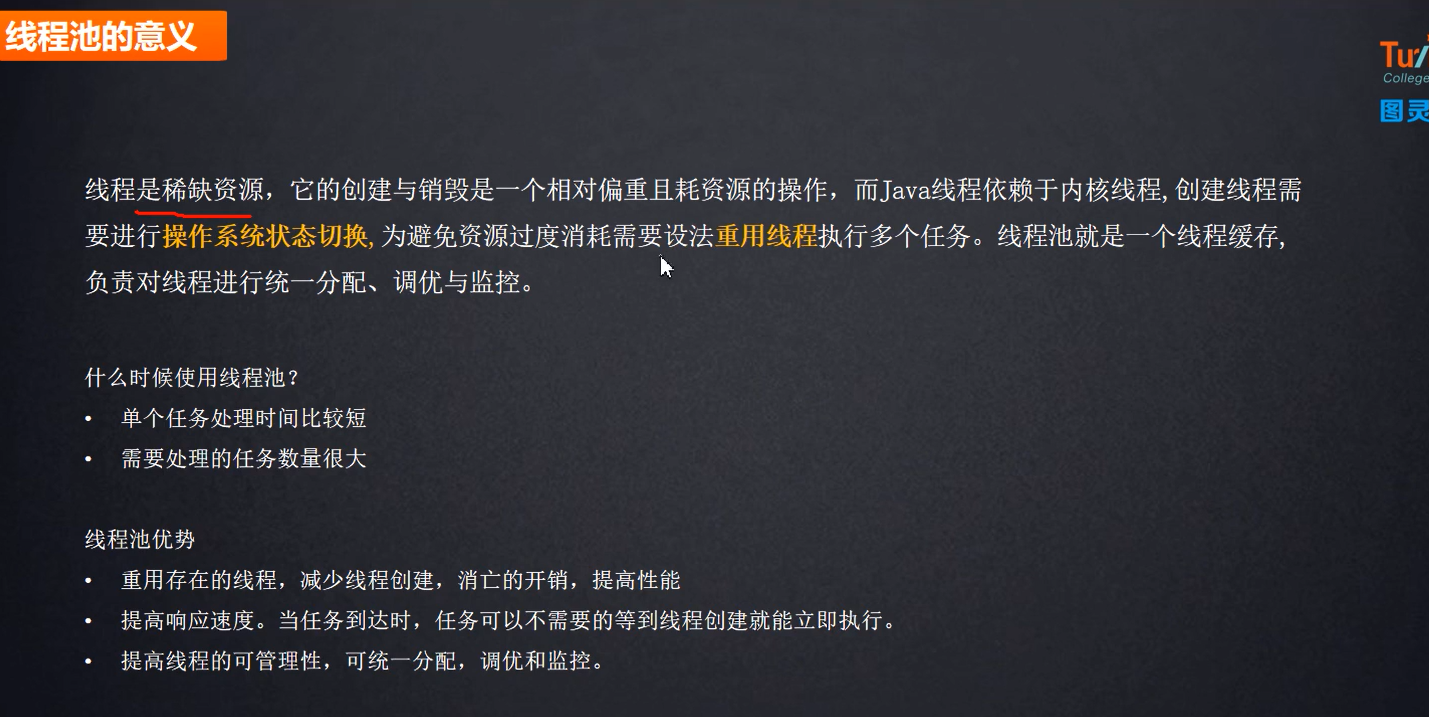


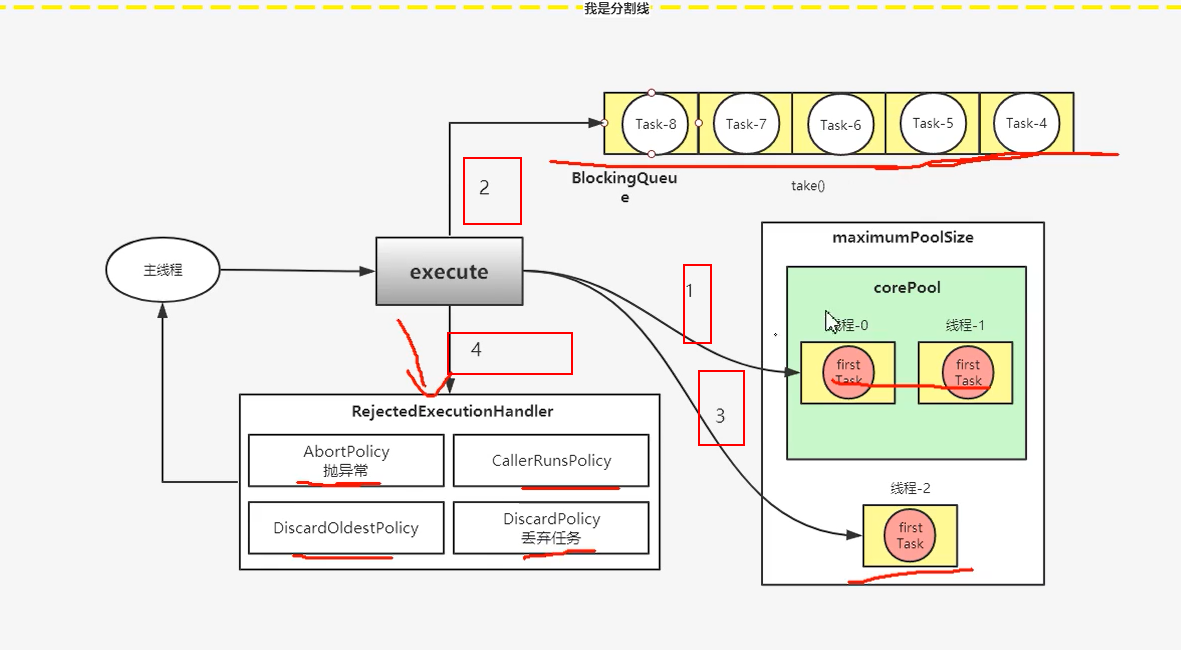
#### 2. JVM：大部分用的是内核线程（KLT）

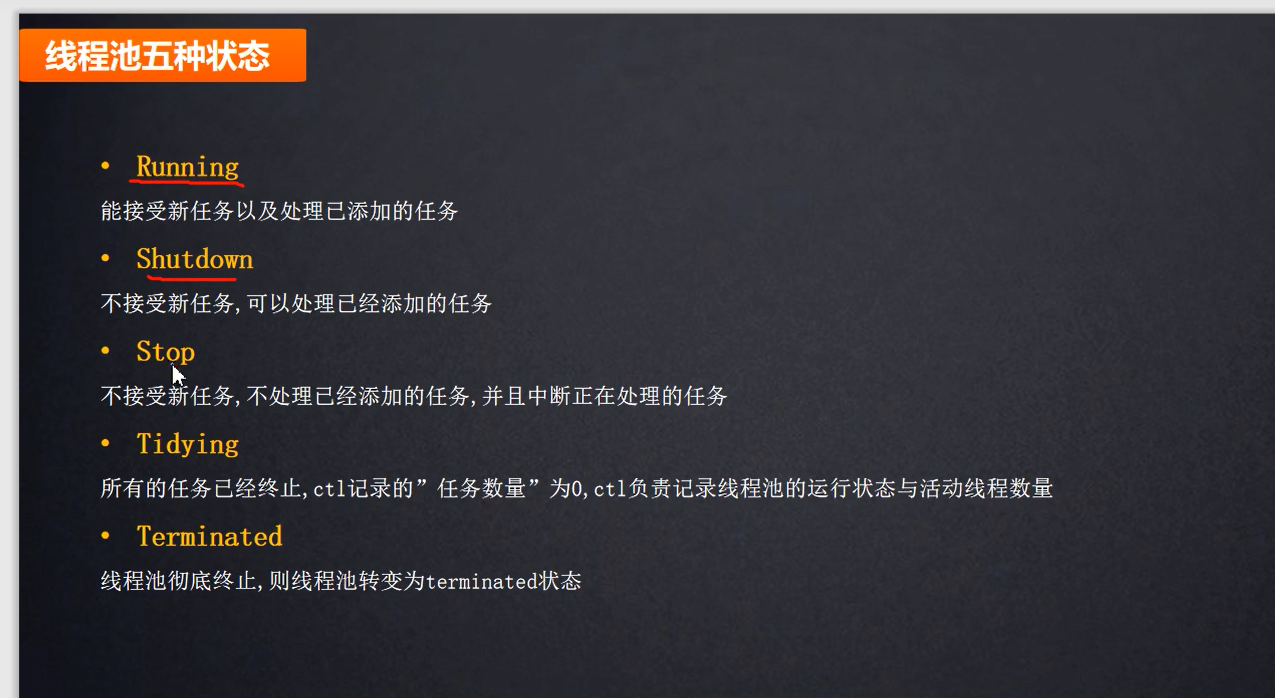


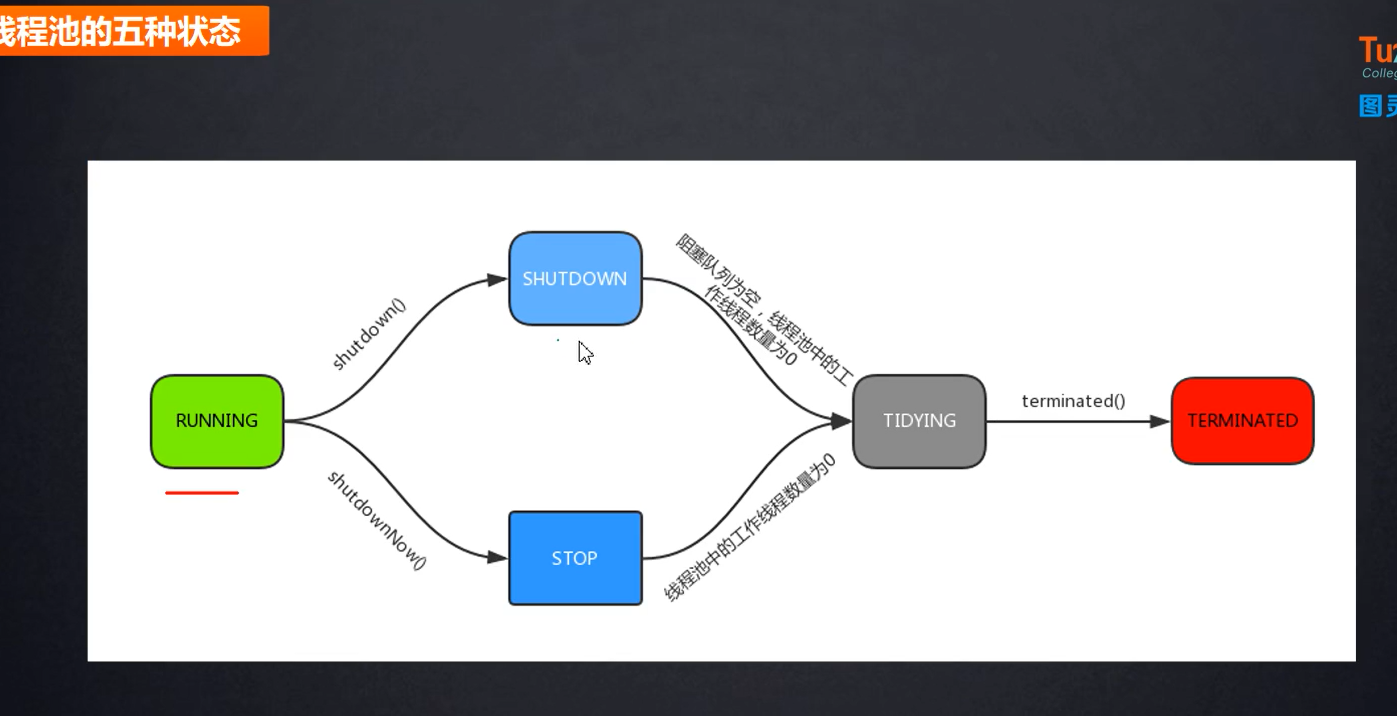


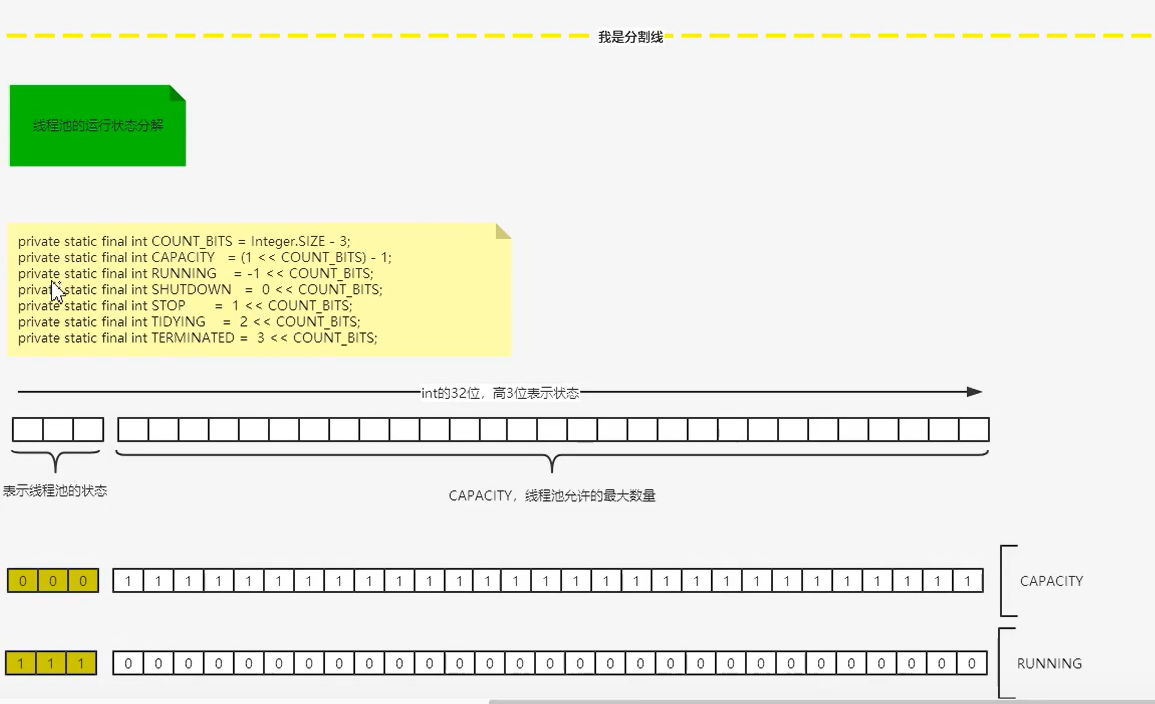
#### 3. 线程池的好处意义

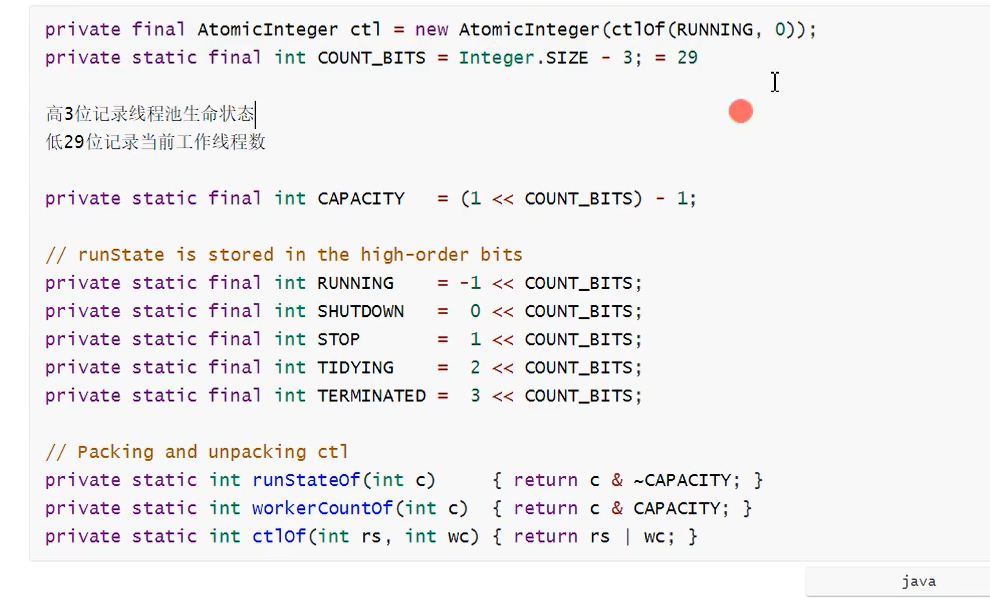


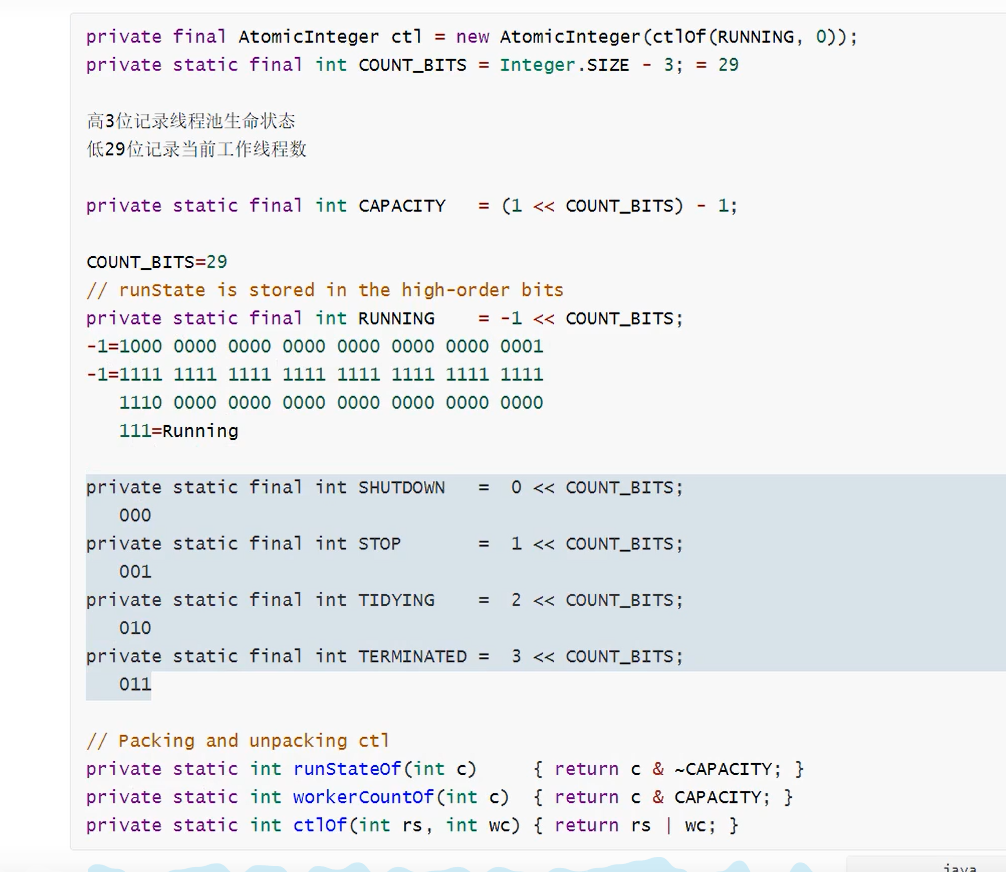


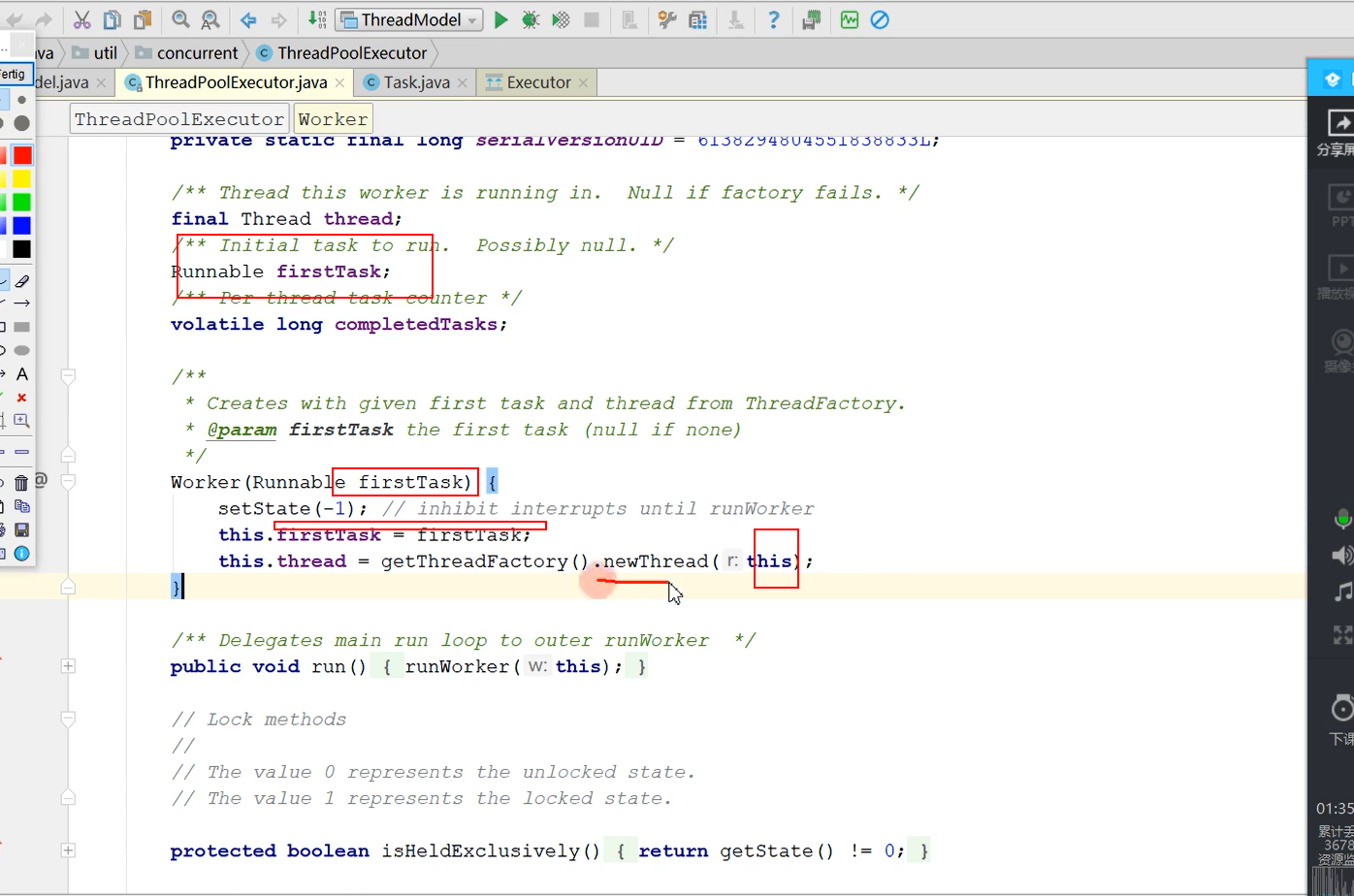


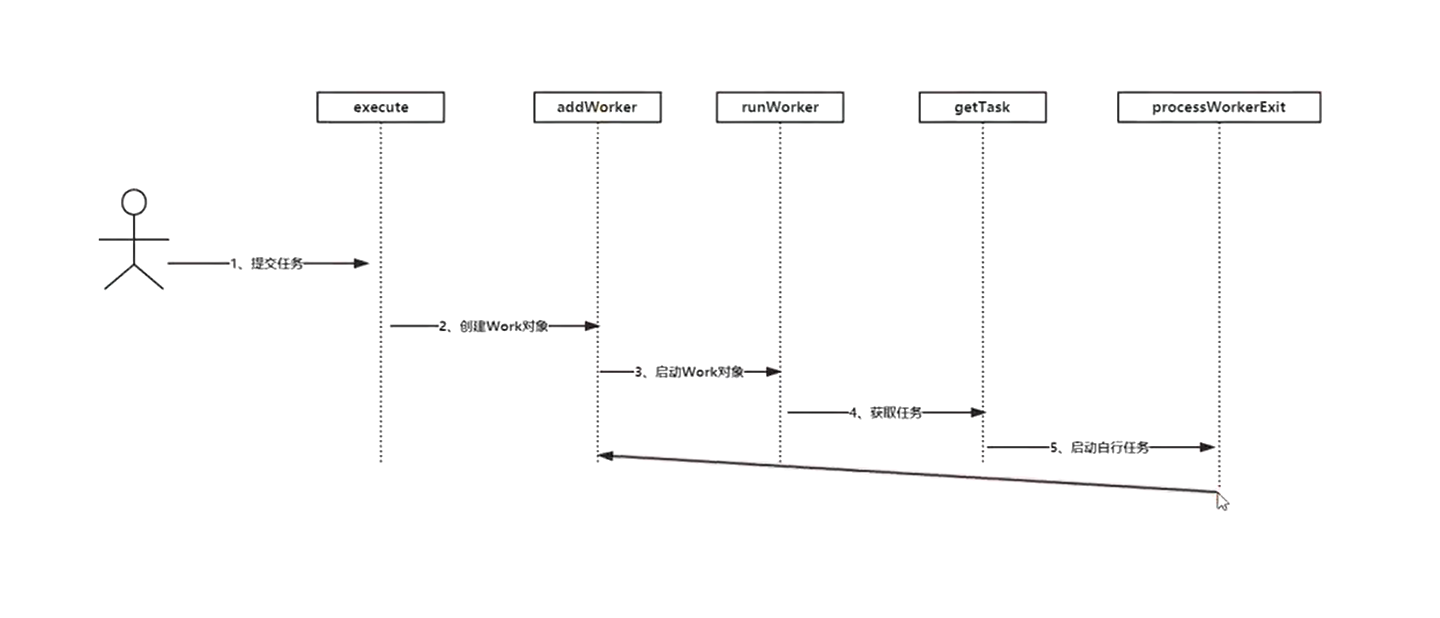














核心线程复用保活两个点：

1，加回去

/\*\*  
 \* Performs cleanup and bookkeeping for a dying worker. Called  
 \* only from worker threads. Unless completedAbruptly is set,  
 \* assumes that workerCount has already been adjusted to account  
 \* for exit. This method removes thread from worker set, and  
 \* possibly terminates the pool or replaces the worker if either  
 \* it exited due to user task exception or if fewer than  
 \* corePoolSize workers are running or queue is non-empty but  
 \* there are no workers.  
 \*  
 \* **@param w** the worker  
 \* **@param completedAbruptly** if the worker died due to user exception  
 \*/  
**private void processWorkerExit**(**Worker w**, **boolean completedAbruptly**) {  
 **if** (**completedAbruptly**) **// If abrupt, then workerCount wasn't adjusted** decrementWorkerCount();  
  
 **final ReentrantLock** mainLock = **this**.mainLock;  
 mainLock.lock();  
 **try** {  
 completedTaskCount += **w**.completedTasks;  
 workers.remove(**w**);  
 } **finally** {  
 mainLock.unlock();  
 }  
  
 tryTerminate();  
  
 **int** c = ctl.get();  
 **if** (**runStateLessThan**(c, ***STOP***)) {  
 **if** (!**completedAbruptly**) {  
 **int** min = allowCoreThreadTimeOut ? **0** : corePoolSize;  
 **if** (min == **0** && ! workQueue.**isEmpty**())  
 min = **1**;  
 **if** (**workerCountOf**(c) >= min)  
 **return**; **// replacement not needed** }  
 **addWorker(null, false);重新加回去** }  
}

1. 队列阻塞等待任务
2. /\*\*  
    \* Performs blocking or timed wait for a task, depending on  
    \* current configuration settings, or returns null if this worker  
    \* must exit because of any of:  
    \* 1. There are more than maximumPoolSize workers (due to  
    \* a call to setMaximumPoolSize).  
    \* 2. The pool is stopped.  
    \* 3. The pool is shutdown and the queue is empty.  
    \* 4. This worker timed out waiting for a task, and timed-out  
    \* workers are subject to termination (that is,  
    \* {**@code** allowCoreThreadTimeOut || workerCount > corePoolSize})  
    \* both before and after the timed wait, and if the queue is  
    \* non-empty, this worker is not the last thread in the pool.  
    \*  
    \* **@return** task, or null if the worker must exit, in which case  
    \* workerCount is decremented  
    \*/  
   **private Runnable getTask**() {  
    **boolean** timedOut = **false**; **// Did the last poll() time out?  
     
    for** (;;) {  
    **int** c = ctl.get();  
    **int** rs = **runStateOf**(c);  
     
    **// Check if queue empty only if necessary.  
    if** (rs >= ***SHUTDOWN*** && (rs >= ***STOP*** || workQueue.**isEmpty**())) {  
    decrementWorkerCount();  
    **return null**;  
    }  
     
    **int** wc = **workerCountOf**(c);  
     
    **// Are workers subject to culling?  
    boolean** timed = allowCoreThreadTimeOut || wc > corePoolSize;  
     
    **if** ((wc > maximumPoolSize || (timed && timedOut))  
    && (wc > **1** || workQueue.**isEmpty**())) {  
    **if** (compareAndDecrementWorkerCount(c))  
    **return null**;  
    **continue**;  
    }  
     
    **try** {  
    **Runnable** r = timed ?  
    workQueue.**poll**(keepAliveTime, **TimeUnit**.***NANOSECONDS***) :  
    **workQueue.take();**  
    **if** (r != **null**)  
    **return** r;  
    timedOut = **true**;  
    } **catch** (**InterruptedException retry**) {  
    timedOut = **false**;  
    }  
    }  
   }

<https://blog.csdn.net/MingHuang2017/article/details/79571529>

#### 面试题

#### 总结

1.5种状态

2.7个参数

3.4中默认拒绝策略

4. 4中默认线程池

5.执行流程

6.核心线程两种保活策略

7.自定义线程策略

N+1 2n+1

线程池合理的线程数你是如何考虑的？：

1.先看下机器的CPU核数，然后在设定具体参数：

System.out.println(Runtime.getRuntime().availableProcessors());

即CPU核数 = Runtime.getRuntime().availableProcessors()

2.分析下线程池处理的程序是CPU密集型，还是IO密集型

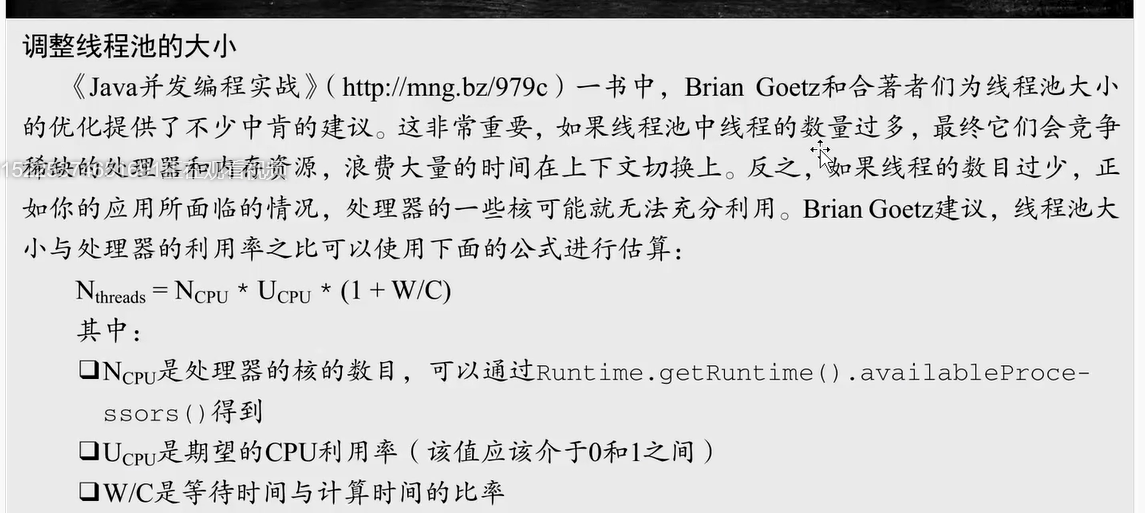
CPU密集型：核心线程数 = CPU核数 + 1

IO密集型：核心线程数 = CPU核数 \* 2

注：IO密集型（某大厂实践经验）

       核心线程数 = CPU核数 / （1-阻塞系数）     例如阻塞系数 0.8，CPU核数为4

       则核心线程数为20



#### TODO 线程池源码

#### Java8的线程池优化点

1，addWorker()方法里，获取锁放到了if里面

if (t != null) {

final ReentrantLock mainLock = this.mainLock;

2. private Runnable getTask() 中由jdk7的双层for循环优化为单层for循环，提高效率

3.finalize()由简单shutdown();增加了安全检查，不过从9不建议使用了。

@Deprecated(since="9")

protected void finalize() {

SecurityManager sm = System.getSecurityManager();

if (sm == null || acc == null) {

shutdown();

} else {

PrivilegedAction<Void> pa = () -> { shutdown(); return null; };

AccessController.doPrivileged(pa, acc);

}

}

4.增加了安全检查

this.acc = System.getSecurityManager() == null ?

null :

AccessController.getContext();

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {

5.修改核心线程时候，增加了检查判断 maximumPoolSize < corePoolSize。

public void setCorePoolSize(int corePoolSize) {

if (corePoolSize < 0 || maximumPoolSize < corePoolSize)

6. Java8 中，默认创建线程池的方法多了一个——Executors.newWorkStealingPool()。本质还是ForkjoinPool.

\* **@since** 1.8  
 \*/  
**public static ExecutorService newWorkStealingPool**(**int parallelism**) {  
 **return new** ForkJoinPool  
 (**parallelism**,  
 **ForkJoinPool**.***defaultForkJoinWorkerThreadFactory***,  
 **null**, **true**);  
}

newWorkStealingPool 的文档描述：

*“Creates a work-stealing thread pool using all available processors as its target parallelism level.”*

newWorkStealingPool 会创建一个含有足够多线程的线程池，来维持相应的并行级别，它会通过工作窃取的方式，使得多核的 CPU 不会闲置，总会有活着的线程让 CPU 去运行。

工作窃取概念（Work stealing）

工作窃取不是什么 Java 独有的东西，.NET 的 TPL 库早就存在好几年了。所谓工作窃取，指的是闲置的线程去处理本不属于它的任务。

每个处理器核，都有一个队列存储着需要完成的任务。对于多核的机器来说，当一个核对应的任务处理完毕后，就可以去帮助其他的核处理任务。