1. **线程池的参数**



1. **corePoolSize（长工） 与 maxPoolSize（临时工）**

当corePoolSize已经满，线程则会存放再workQueue，workQueue满后，会创建新的线程 ，直到达到maxPoolSize后，不再提交到线程池任务队列

总结：



1. **keepAliveTime+时间单位**

当线程池中线程数量多于核心线程数时，而此时又没有任务可做，线程池就会检测线程的 keepAliveTime，如果超过规定的时间，无事可做的线程就会被销毁，以便减少内存的占用和资源消耗。

1. **ThreadFactory**

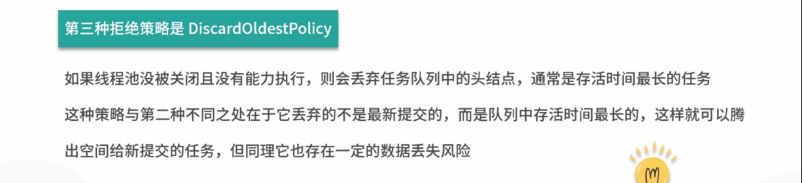
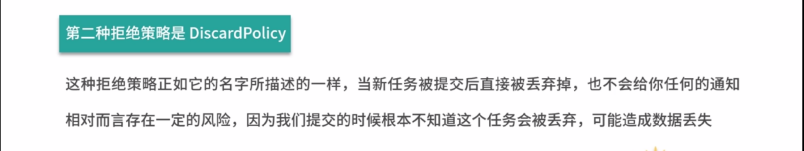
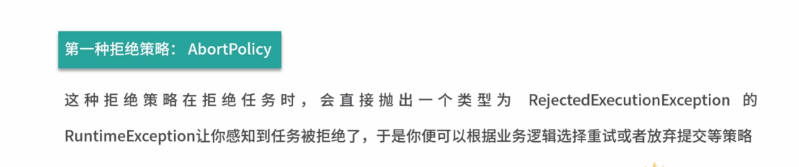
ThreadFactory 实际上是一个线程工厂，它的作用是生产线程以便执行任务。我们可以选择使用默认的线程工厂，创建的线程都会在同一个线程组，并拥有一样的优先级，且都不是守护线程，我们也可以选择自己定制线程工厂，以方便给线程自定义命名，不同的线程池内的线程通常会根据具体业务来定制不同的线程名。

1. **四种拒绝策略**

线程池会在以下两种情况下会拒绝新提交的任务：

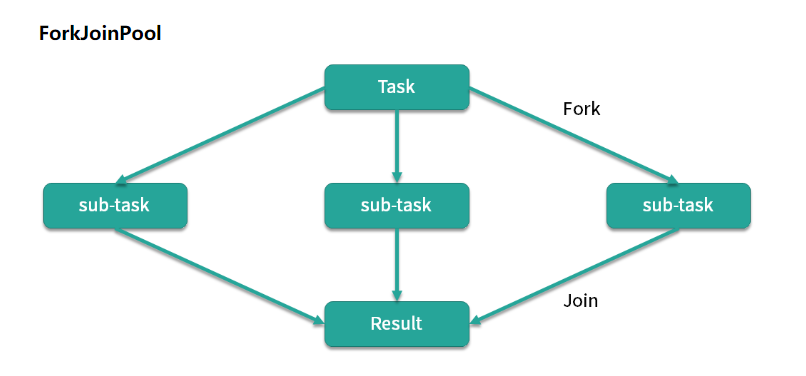
* 第一种情况是当我们调用 shutdown 等方法关闭线程池后，即便此时可能线程池内部依然有没执行完的任务正在执行，但是由于线程池已经关闭，此时如果再向线程池内提交任务，就会遭到拒绝。
* 第二种情况是线程池没有能力继续处理新提交的任务，也就是工作已经非常饱和的时候。

如图：



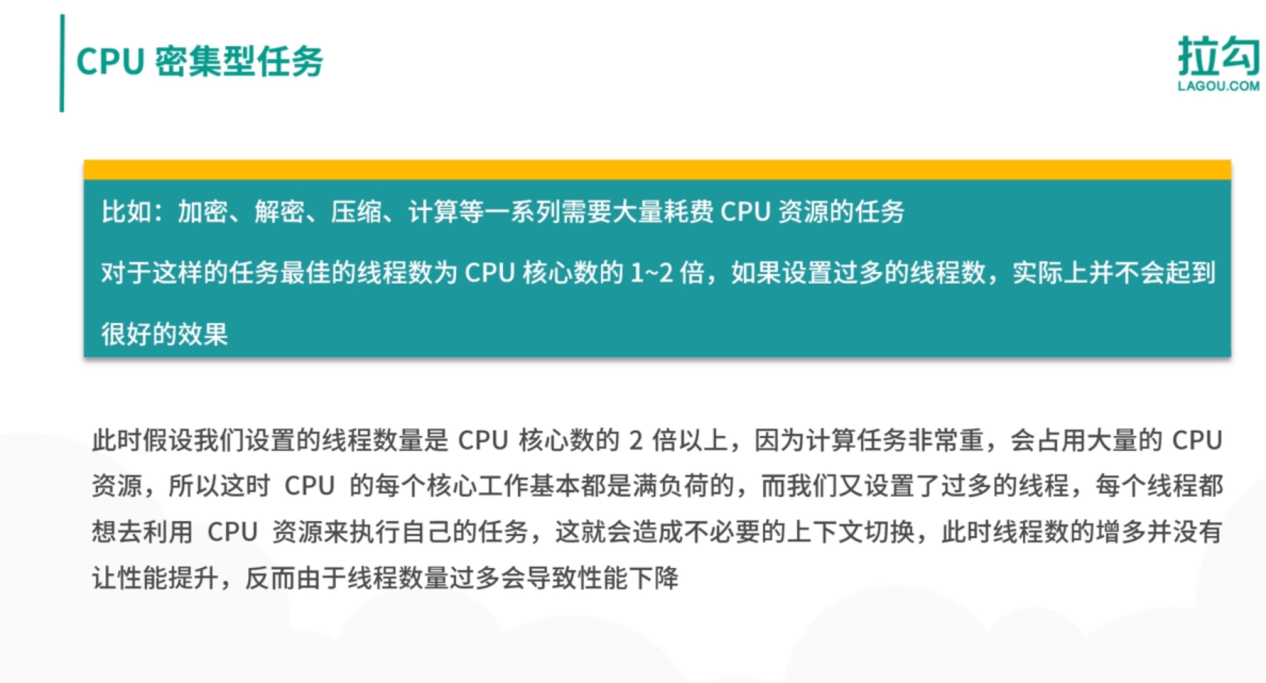
1. **Java8 的 ForkJoinPool？**

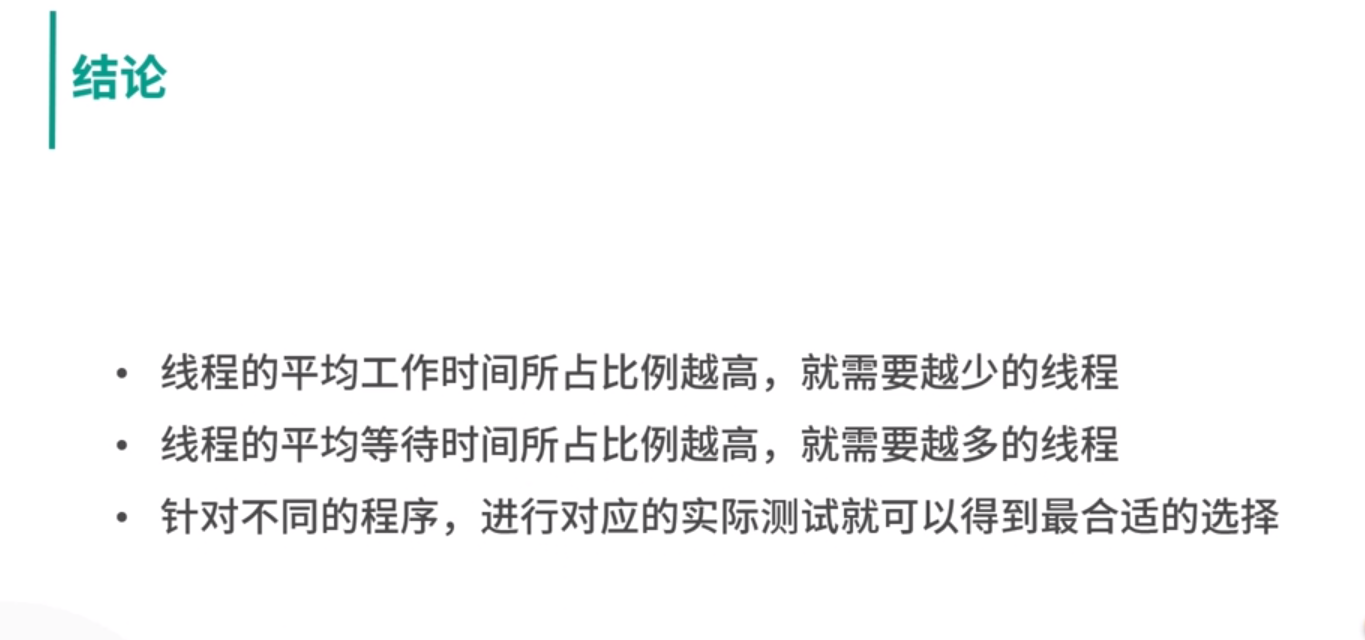
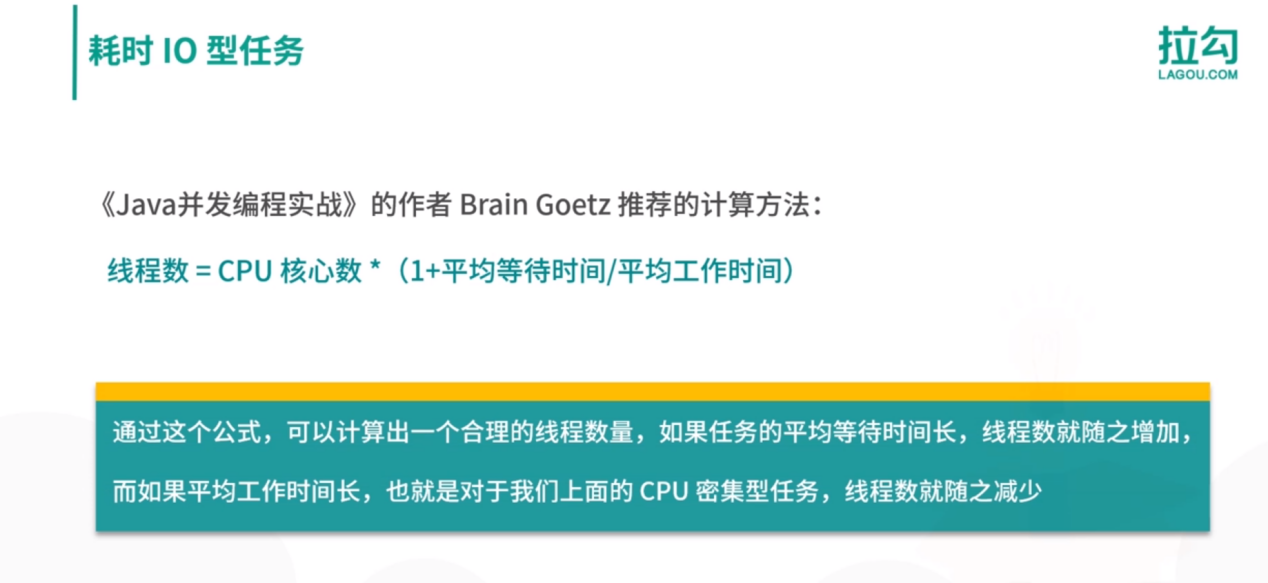
可以执行多个子任务，并最后汇总到主任务中，如图：



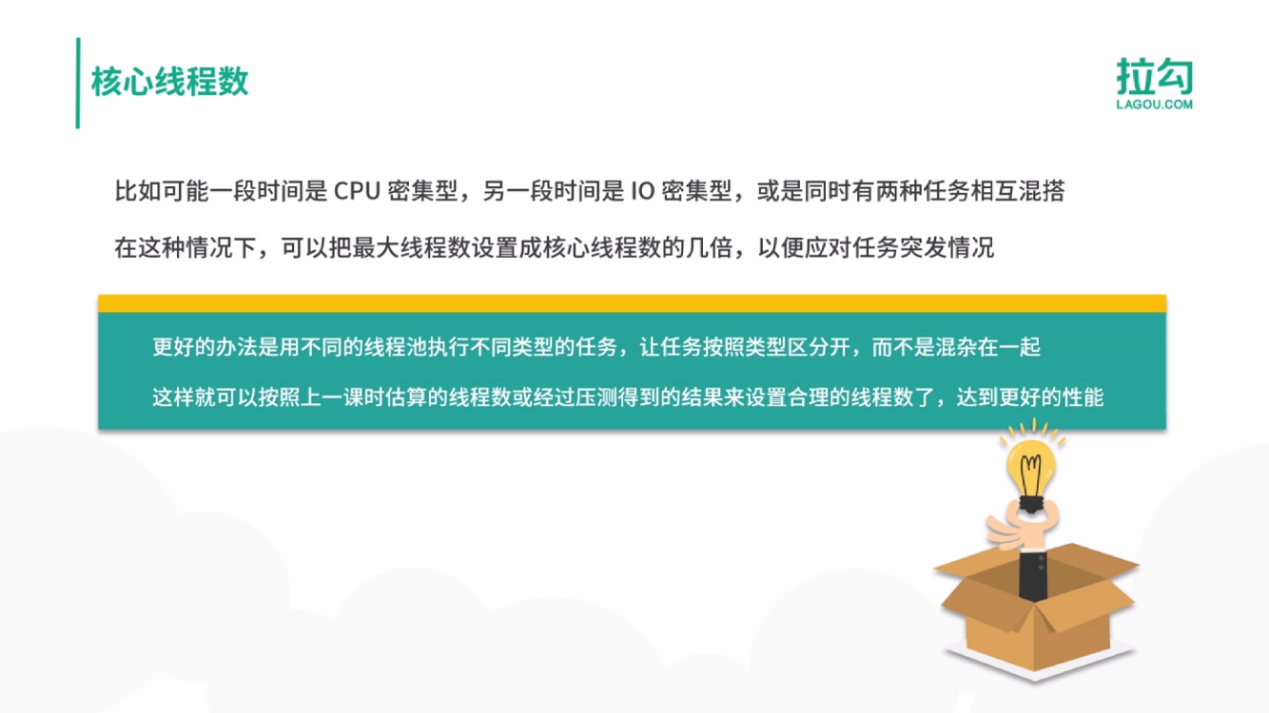
ForkJoinPool 线程池内部除了有一个共用的任务队列之外，每个线程还有一个对应的双端队列 deque，这时一旦线程中的任务被 Fork 分裂了，分裂出来的子任务放入线程自己的 deque 里，而不是放入公共的任务队列中。

1. **合适的线程数量是多少？CPU 核心数和线程数的关系？**

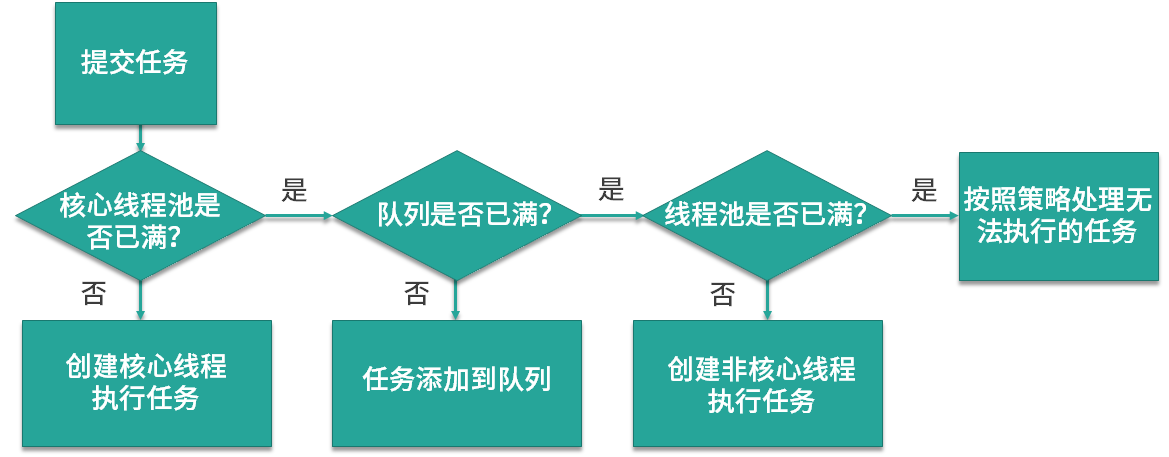




1. **如何根据实际需要，定制自己的线程池？**



1. **线程池创建新线程的时机和规则：**



1. **线程池实现“线程复用”的原理**

execute源码分析：

public void execute(Runnable command) {

//如果传入的Runnable的空，就抛出异常

    if (command == null)

        throw new NullPointerException();

int c = ctl.get();

//判断当前线程数是否小于核心线程数

if (workerCountOf(c) < corePoolSize) {

//如果小于核心线程数就调用 addWorker() 方法增加一个 Worker，这里的 Worker 就可以理解为一个线程

        if (addWorker(command, true))

            return;

        c = ctl.get();

}

//如果代码执行到这里，说明当前线程数大于或等于核心线程数或者 addWorker 失败了

//如果线程池状态Running 就把任务放入任务队列中，也就是 workQueue.offer(command)。如果线程池已经不处于 Running 状态，说明线程池被关闭，那么就移除刚刚添加到任务队列中的任务，并执行拒绝策略

    if (isRunning(c) && workQueue.offer(command)) {

        int recheck = ctl.get();

        if (! isRunning(recheck) && remove(command))

            reject(command);

//防止没有可执行线程的情况发生（比如之前的线程被回收了或意外终止了）

        else if (workerCountOf(recheck) == 0)

            addWorker(null, false);

}

//执行到这里，说明线程池不是 Running 状态或线程数大于或等于核心线程数并且任务队列已经满了

    else if (!addWorker(command, false))

        reject(command);

}

addWorker 方法又是做什么用的呢？

addWorker 方法的主要作用是在线程池中创建一个线程并执行第一个参数传入的任务，它的第二个参数是个布尔值，如果布尔值传入 true 代表增加线程时判断当前线程是否少于 corePoolSize，小于则增加新线程，大于等于则不增加；同理，如果传入 false 代表增加线程时判断当前线程是否少于 maxPoolSize，小于则增加新线程，大于等于则不增加。

所以这里的布尔值的含义是以核心线程数为界限还是以最大线程数为界限进行是否新增线程的判断。addWorker() 方法如果返回 true 代表添加成功，如果返回 false 代表添加失败。

可以看出，在 execute 方法中，多次调用 addWorker 方法把任务传入，addWorker 方法会添加并启动一个 Worker，这里的 Worker 可以理解为是对 Thread 的包装，Worker 内部有一个 Thread 对象，它正是最终真正执行任务的线程，所以一个 Worker 就对应线程池中的一个线程，addWorker 就代表增加线程。线程复用的逻辑实现主要在 Worker 类中的 run 方法里执行的 runWorker 方法中，简化后的 runWorker 方法代码如下所示。

runWorker(Worker w) {

    Runnable task = w.firstTask;

    while (task != null || (task = getTask()) != null) {

        try {

            task.run();

        } finally {

            task = null;

        }

    }

}

可以看出，实现线程复用的逻辑主要在一个不停循环的 while 循环体中。

\*通过取 Worker 的 firstTask 或者通过 getTask 方法从 workQueue 中获取待执行的任务。

\*直接调用 task 的 run 方法来执行具体的任务（而不是新建线程）。