在Java语言中，虽然创建并启动一个线程非常方便，但是由于创建线程需要占用一定的操作系统资源，在高并发的情况下，频繁的创建和销毁线程会大量消耗CPU和内存资源，对程序性能造成很大的影响。为了避免这一问题，Java给我们提供了线程池。

线程池是一种基于池化技术思想来管理线程的工具。在线程池中维护了多个线程，由线程池统一的管理调配线程来执行任务。通过线程复用，减少了频繁创建和销毁线程的开销。

1.线程池的使用

线程池的使用和创建可以说非常的简单，这得益于JDK提供给我们良好封装的API。线程池的实现被封装到了ThreadPoolExecutor中，我们可以通过ThreadPoolExecutor的构造方法来实例化出一个线程池，代码如下：

// 实例化一个线程池

ThreadPoolExecutor executor = new ThreadPoolExecutor(3, 10, 60,

TimeUnit.SECONDS, new ArrayBlockingQueue<>(20));

// 使用线程池执行一个任务

executor.execute(() -> {

// Do something

});

// 关闭线程池,会阻止新任务提交，但不影响已提交的任务

executor.shutdown();

// 关闭线程池，阻止新任务提交，并且中断当前正在运行的线程

executor.showdownNow();

创建好线程池后直接调用execute方法并传入一个Runnable参数即可将任务交给线程池执行，通过shutdown/shutdownNow方法可以关闭线程池。

ThreadPoolExecutor的构造方法中参数众多，对于初学者而言在没有了解各个参数的作用的情况下很难去配置合适的线程池。因此Java还为我们提供了一个线程池工具类Executors来快捷的创建线程池。Executors提供了很多简便的创建线程池的方法，举两个例子，代码如下：

// 实例化一个单线程的线程池

ExecutorService singleExecutor = Executors.newSingleThreadExecutor();

// 创建固定线程个数的线程池

ExecutorService fixedExecutor = Executors.newFixedThreadPool(10);

// 创建一个可重用固定线程数的线程池

ExecutorService executorService2 = Executors.newCachedThreadPool();

2.线程池的生命周期

线程池从诞生到死亡，中间会经历RUNNING、SHUTDOWN、STOP、TIDYING、TERMINATED五个生命周期状态。

RUNNING 表示线程池处于运行状态，能够接受新提交的任务且能对已添加的任务进行处理。RUNNING状态是线程池的初始化状态，线程池一旦被创建就处于RUNNING状态。

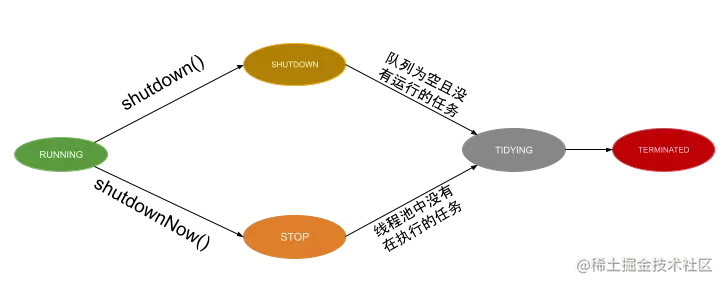
SHUTDOWN 线程处于关闭状态，不接受新任务，但可以处理已添加的任务。RUNNING状态的线程池调用shutdown后会进入SHUTDOWN状态。

STOP 线程池处于停止状态，不接收任务，不处理已添加的任务，且会中断正在执行任务的线程。RUNNING状态的线程池调用了shutdownNow后会进入STOP状态。

TIDYING 当所有任务已终止，且任务数量为0时，线程池会进入TIDYING。当线程池处于SHUTDOWN状态时，阻塞队列中的任务被执行完了，且线程池中没有正在执行的任务了，状态会由SHUTDOWN变为TIDYING。当线程处于STOP状态时，线程池中没有正在执行的任务时则会由STOP变为TIDYING。

TERMINATED 线程终止状态。处于TIDYING状态的线程执行terminated()后进入TERMINATED状态。

根据上述线程池生命周期状态的描述，可以画出如下所示的线程池生命周期状态流程示意图。



1.ThreadPoolExecutor中的参数

上一小节中，我们使用ThreadPoolExecutor的构造方法来创建了一个线程池。其实在ThreadPoolExecutor中有多个构造方法，但是最终都调用到了下边代码中的这一个构造方法：

public class ThreadPoolExecutor extends AbstractExecutorService {

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {

// ...省略校验相关代码

this.corePoolSize = corePoolSize;

this.maximumPoolSize = maximumPoolSize;

this.workQueue = workQueue;

this.keepAliveTime = unit.toNanos(keepAliveTime);

this.threadFactory = threadFactory;

this.handler = handler;

}

// ...

}

这个构造方法中有7个参数之多，我们逐个来看每个参数所代表的含义：

corePoolSize 表示线程池的核心线程数。当有任务提交到线程池时，如果线程池中的线程数小于corePoolSize,那么则直接创建新的线程来执行任务。

workQueue 任务队列，它是一个阻塞队列，用于存储来不及执行的任务的队列。当有任务提交到线程池的时候，如果线程池中的线程数大于等于corePoolSize，那么这个任务则会先被放到这个队列中，等待执行。

maximumPoolSize 表示线程池支持的最大线程数量。当一个任务提交到线程池时，线程池中的线程数大于corePoolSize,并且workQueue已满，那么则会创建新的线程执行任务，但是线程数要小于等于maximumPoolSize。

keepAliveTime 非核心线程空闲时保持存活的时间。非核心线程即workQueue满了之后，再提交任务时创建的线程，因为这些线程不是核心线程，所以它空闲时间超过keepAliveTime后则会被回收。

unit 非核心线程空闲时保持存活的时间的单位

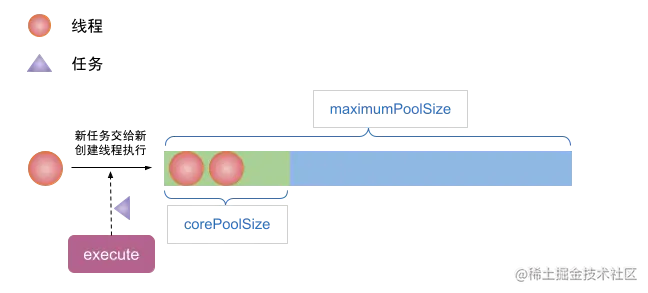
threadFactory 创建线程的工厂，可以在这里统一处理创建线程的属性

handler 拒绝策略，当线程池中的线程达到maximumPoolSize线程数后且workQueue已满的情况下，再向线程池提交任务则执行对应的拒绝策略

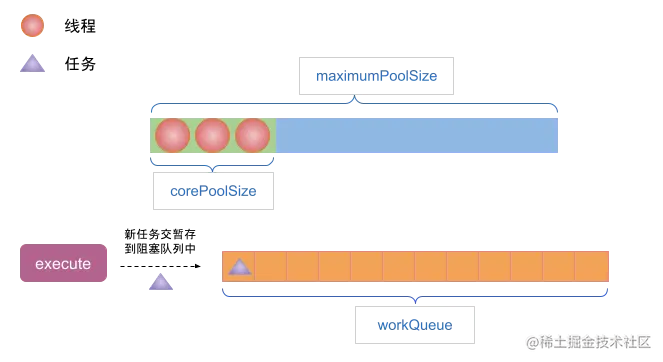
2.线程池工作流程

线程池提交任务是从execute方法开始的，我们可以从execute方法来分析线程池的工作流程。

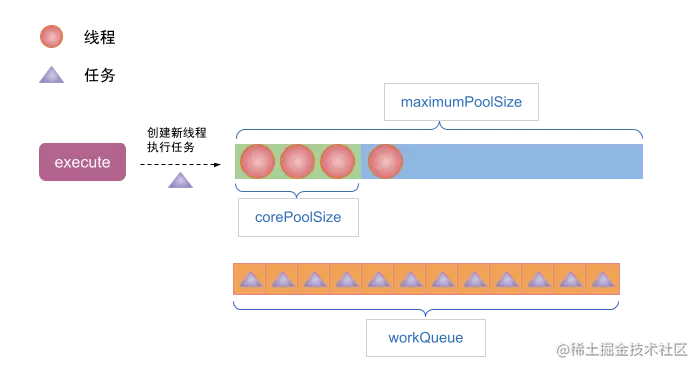
（1）当execute方法提交一个任务时，如果线程池中线程数小于corePoolSize,那么不管线程池中是否有空闲的线程，都会创建一个新的线程来执行任务。



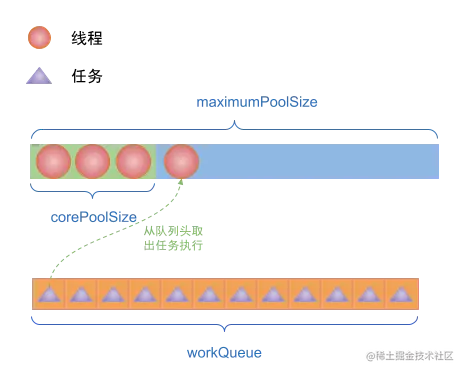
（2）当execute方法提交一个任务时，线程池中的线程数已经达到了corePoolSize,且此时没有空闲的线程，那么则会将任务存储到workQueue中。



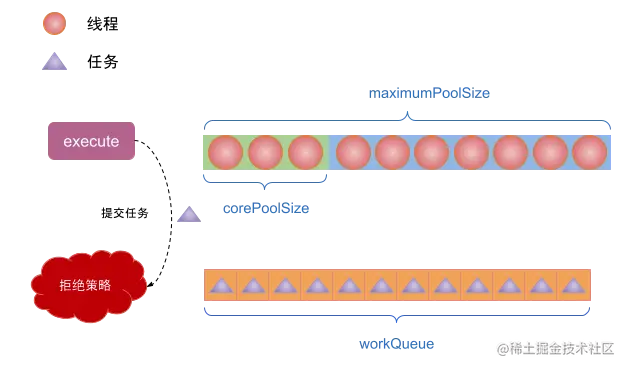
execute提交任务时线程池中的线程数已经到达了corePoolSize,并且workQueue已满，那么则会创建新的线程来执行任务，但总线程数应该小于maximumPoolSize。



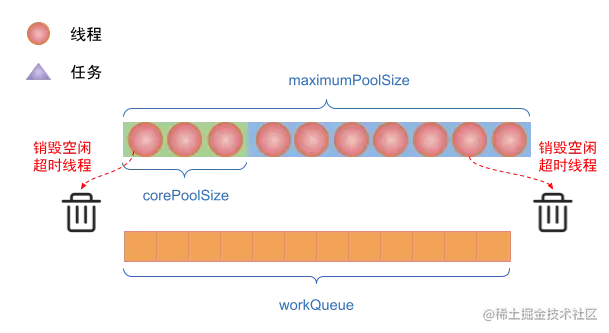
（4）如果线程池中的线程执行完了当前的任务，则会尝试从workQueue中取出第一个任务来执行。如果workQueue为空则会阻塞线程。



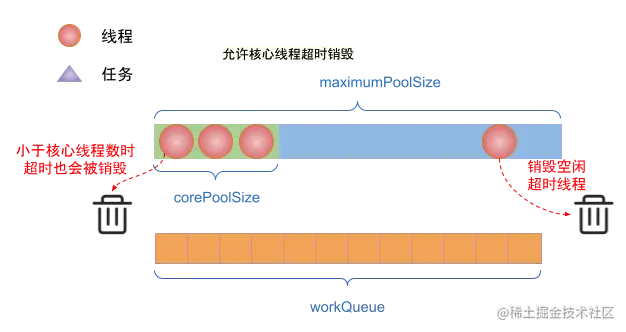
（5）如果execute提交任务时，线程池中的线程数达到了maximumPoolSize，且workQueue已满，此时会执行拒绝策略来拒绝接受任务。



（6）如果线程池中的线程数超过了corePoolSize，那么空闲时间超过keepAliveTime的线程会被销毁，但程池中线程个数会保持为corePoolSize。



（7）如果线程池存在空闲的线程，并且设置了allowCoreThreadTimeOut为true。那么空闲时间超过keepAliveTime的线程都会被销毁。



### 3.线程池的拒绝策略

如果线程池中的线程数达到了maximumPoolSize，并且workQueue队列存储满的情况下，线程池会执行对应的拒绝策略。在JDK中提供了RejectedExecutionHandler接口来执行拒绝操作。实现RejectedExecutionHandler的类有四个，对应了四种拒绝策略。分别如下：

DiscardPolicy 当提交任务到线程池中被拒绝时，线程池会丢弃这个被拒绝的任务

DiscardOldestPolicy 当提交任务到线程池中被拒绝时，线程池会丢弃等待队列中最老的任务。

CallerRunsPolicy 当提交任务到线程池中被拒绝时，会在线程池当前正在运行的Thread线程中处理被拒绝额任务。即哪个线程提交的任务哪个线程去执行。

AbortPolicy 当提交任务到线程池中被拒绝时，直接抛出RejectedExecutionException异常。

1.线程池中的位运算（了解内容）

在向线程池提交任务时有两个比较中要的参数会决定任务的去向，这两个参数分别是线程池的状态和线程池中的线程数。在ThreadPoolExecutor内部使用了一个AtomicInteger类型的整数ctl来表示这两个参数，代码如下：

public class ThreadPoolExecutor extends AbstractExecutorService {

// Integer.SIZE = 32.所以 COUNT\_BITS= 29

private static final int COUNT\_BITS = Integer.SIZE - 3;

// 00011111 11111111 11111111 11111111 这个值可以表示线程池的最大线程容量

private static final int COUNT\_MASK = (1 << COUNT\_BITS) - 1;

// 将-1左移29位得到RUNNING状态的值

private static final int RUNNING = -1 << COUNT\_BITS;

// 线程池运行状态和线程数

private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

private static int ctlOf(int rs, int wc) { return rs | wc; }

// ...

}

因为涉及多线程的操作，这里为了保证原子性，ctl参数使用了AtomicInteger类型，并且通过ctlOf方法来计算出了ctl的初始值。如果你不了解位运算大概很难理解上述代码的用意。

我们知道，int类型在Java中占用4byte的内存,一个byte占用8bit,所以Java中的int类型共占用32bit。对于这个32bit，我们可以进行高低位的拆分。做Android开发的同学应该都了解View测量流程中的MeasureSpec参数，这个参数将32bit的int拆分成了高2位和低30位，分别表示View的测量模式和测量值。而这里的ctl与MeasureSpec类似，ctl将32位的int拆分成了高3位和低29位，分别表示线程池的运行状态和线程池中的线程个数。

可以看到上述代码中RUNNING的值为-1左移29位，我们知道在计算机中\*\*负数是以其绝对值的补码来表示的，而补码是由反码加1得到。\*\*因此-1在计算机中存储形式为1的反码+1

1的原码：00000000 00000000 00000000 00000001

+

1的反码：11111111 11111111 11111111 11111110

---------------------------------------

-1存储： 11111111 11111111 11111111 11111111

接下来对-1左移29位可以得到RUNNING的值为：

// 高三位表示线程状态，即高三位为111表示RUNNING

11100000 00000000 00000000 00000000

而AtomicInteger初始线程数量是0，因此ctlOf方法中的“|”运算如下：

RUNNING： 11100000 00000000 00000000 00000000

|

线程数为0: 00000000 00000000 00000000 00000000

---------------------------------------

得到ctl： 11100000 00000000 00000000 00000000

通过RUNNING|0(线程数)即可得到ctl的初始值。同时还可以通过以下方法将ctl拆解成运行状态和线程数：

// 00011111 11111111 11111111 11111111

private static final int COUNT\_MASK = (1 << COUNT\_BITS) - 1;

// 获取线程池运行状态

private static int runStateOf(int c) { return c & ~COUNT\_MASK; }

// 获取线程池中的线程数

private static int workerCountOf(int c) { return c & COUNT\_MASK; }

假设此时线程池为RUNNING状态，且线程数为0，验证一下runStateOf是如何得到线程池的运行状态的：

COUNT\_MASK: 00011111 11111111 11111111 11111111

~COUNT\_MASK: 11100000 00000000 00000000 00000000

&

ctl: 11100000 00000000 00000000 00000000

----------------------------------------

RUNNING: 11100000 00000000 00000000 00000000

如果不理解上边的验证流程没有关系，只要知道通过runStateOf方法可以得到线程池的运行状态，通过workerCountOf可以得到线程池中的线程数即可。

2.ThreadPoolExecutor的execute

向线程池提交任务的方法是execute方法，execute方法是ThreadPoolExecutor的核心方法，以此方法为入口来进行剖析，execute方法的代码如下：

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

// 获取ctl的值

int c = ctl.get();

// 1.线程数小于corePoolSize

if (workerCountOf(c) < corePoolSize) {

// 线程池中线程数小于核心线程数，则尝试创建核心线程执行任务

if (addWorker(command, true))

return;

c = ctl.get();

}

// 2.到此处说明线程池中线程数大于核心线程数或者创建线程失败

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

// 如果线程是运行状态并且可以使用offer将任务加入阻塞队列未满，

// offer是非阻塞操作。

int recheck = ctl.get();

// 重新检查线程池状态，因为上次检测后线程池状态可能发生改变，

// 如果非运行状态就移除任务并执行拒绝策略

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

reject(command);

// 如果是运行状态，并且线程数是0，则创建线程

else if (workerCountOf(recheck) == 0)

// 线程数是0，则创建非核心线程，且不指定首次执行任务，这里的第二个参数其实没有实际意义

addWorker(null, false);

}

// 3.阻塞队列已满，创建非核心线程执行任务

else if (!addWorker(command, false))

// 如果失败，则执行拒绝策略

reject(command);

}

execute方法中的逻辑可以分为三部分：

1.如果线程池中的线程数小于核心线程，则直接调用addWorker方法创建新线程来执行任务。

2.如果线程池中的线程数大于核心线程数，则将任务添加到阻塞队列中，接着再次检验线程池的运行状态，因为上次检测过之后线程池状态有可能发生了变化，如果线程池关闭了，那么移除任务，执行拒绝策略。如果线程依然是运行状态，但是线程池中没有线程，那么就调用addWorker方法创建线程，注意此时传入任务参数是null，即不指定执行任务，因为任务已经加入了阻塞队列。创建完线程后从阻塞队列中取出任务执行。

3.如果第2步将任务添加到阻塞队列失败了，说明阻塞队列任务已满，那么则会执行第三步，即创建非核心线程来执行任务，如果非核心线程创建失败那么就执行拒绝策略。

可以看到，代码的执行逻辑和我们在第二章中分析的线程池的工作流程是一样的。

接下来看下execute方法中创建线程的方法addWoker，addWoker方法承担了核心线程和非核心线程的创建，通过一个boolean参数core来区分是创建核心线程还是非核心线程。先来看addWorker方法前半部分的代码：

// 返回值表示是否成功创建了线程

private boolean addWorker(Runnable firstTask, boolean core) {

// 这里做了一个retry标记，相当于goto.

retry:

for (int c = ctl.get();;) {

// Check if queue empty only if necessary.

if (runStateAtLeast(c, SHUTDOWN)

&& (runStateAtLeast(c, STOP)

|| firstTask != null

|| workQueue.isEmpty()))

return false;

for (;;) {

// 根据core来确定创建最大线程数，超过最大值则创建线程失败，

// 注意这里的最大值可能有三个corePoolSize、maximumPoolSize和线程池线程的最大容量

if (workerCountOf(c)

>= ((core ? corePoolSize : maximumPoolSize) & COUNT\_MASK))

return false;

// 通过CAS来将线程数+1，如果成功则跳出循环，执行下边逻辑

if (compareAndIncrementWorkerCount(c))

break retry;

c = ctl.get(); // Re-read ctl

// 线程池的状态发生了改变，退回retry重新执行

if (runStateAtLeast(c, SHUTDOWN))

continue retry;

}

}

// ...省略后半部分

return workerStarted;

}

这部分代码会通过是否创建核心线程来确定线程池中线程数的值，如果是创建核心线程，那么最大值不能超过corePoolSize,如果是创建非核心线程那么线程数不能超过maximumPoolSize，另外无论是创建核心线程还是非核心线程，最大线程数都不能超过线程池允许的最大线程数COUNT\_MASK(有可能设置的maximumPoolSize大于COUNT\_MASK)。如果线程数大于最大值就返回false，创建线程失败。

接下来通过CAS将线程数加1，如果成功那么就break retry结束无限循环，如果CAS失败了则就continue retry从新开始for循环，注意这里的retry不是Java的关键字，是一个可以任意命名的字符。

接下来，如果能继续向下执行则开始执行创建线程并执行任务的工作了，看下addWorker方法的后半部分代码：

private boolean addWorker(Runnable firstTask, boolean core) {

// ...省略前半部分

boolean workerStarted = false;

boolean workerAdded = false;

Worker w = null;

try {

// 实例化一个Worker,内部封装了线程

w = new Worker(firstTask);

// 取出新建的线程

final Thread t = w.thread;

if (t != null) {

// 这里使用ReentrantLock加锁保证线程安全

final ReentrantLock mainLock = this.mainLock;

mainLock.lock();

try {

int c = ctl.get();

// 拿到锁湖重新检查线程池状态，只有处于RUNNING状态或者

// 处于SHUTDOWN并且firstTask==null时候才会创建线程

if (isRunning(c) ||

(runStateLessThan(c, STOP) && firstTask == null)) {

// 线程不是处于NEW状态，说明线程已经启动，抛出异常

if (t.getState() != Thread.State.NEW)

throw new IllegalThreadStateException();

// 将线程加入线程队列，这里的worker是一个HashSet

workers.add(w);

workerAdded = true;

int s = workers.size();

if (s > largestPoolSize)

largestPoolSize = s;

}

} finally {

mainLock.unlock();

}

if (workerAdded) {

// 开启线程执行任务

t.start();

workerStarted = true;

}

}

} finally {

if (! workerStarted)

addWorkerFailed(w);

}

return workerStarted;

}

这部分逻辑其实比较容易理解，就是创建Worker并开启线程执行任务的过程，Worker是对线程的封装，创建的worker会被添加到ThreadPoolExecutor中的HashSet中。也就是线程池中的线程都维护在这个名为workers的HashSet中并被ThreadPoolExecutor所管理，HashSet中的线程可能处于正在工作的状态，也可能处于空闲状态，一旦达到指定的空闲时间，则会根据条件进行回收线程。

我们知道，线程调用start后就会开始执行线程的逻辑代码，执行完后线程的生命周期就结束了，那么线程池是如何保证Worker执行完任务后仍然不结束的呢？当线程空闲超时或者关闭线程池又是怎样进行线程回收的呢？这个实现逻辑其实就在Worker中。看下Worker的代码：

private final class Worker

extends AbstractQueuedSynchronizer

implements Runnable

{

// 执行任务的线程

final Thread thread;

// 初始化Worker时传进来的任务，可能为null，如果不空，

// 则创建和立即执行这个task，对应核心线程创建的情况

Runnable firstTask;

Worker(Runnable firstTask) {

// 初始化时设置setate为-1

setState(-1); // inhibit interrupts until runWorker

this.firstTask = firstTask;

// 通过线程工程创建线程

this.thread = getThreadFactory().newThread(this);

}

// 线程的真正执行逻辑

public void run() {

runWorker(this);

}

// 判断线程是否是独占状态，如果不是意味着线程处于空闲状态

protected boolean isHeldExclusively() {

return getState() != 0;

}

// 获取锁

protected boolean tryAcquire(int unused) {

if (compareAndSetState(0, 1)) {

setExclusiveOwnerThread(Thread.currentThread());

return true;

}

return false;

}

// 释放锁

protected boolean tryRelease(int unused) {

setExclusiveOwnerThread(null);

setState(0);

return true;

}

// ...

}

Worker是位于ThreadPoolExecutor中的一个内部类，它继承了AQS，使用AQS来实现了独占锁的功能，但是并没支持可重入。这里使用不可重入的特性来表示线程的执行状态，即可以通过isHeldExclusively方法来判断，如果是独占状态，说明线程正在执行任务，如果非独占状态，说明线程处于空闲状态。关于AQS我们前边文章中已经详细分析过了，不了解AQS的可以翻看前边ReentrantLock的文章。

另外，Worker还实现了Runnable接口，因此它的执行逻辑就是在run方法中，run方法调用的是线程池中的runWorker(this)方法。任务的执行逻辑就在runWorker方法中，它的代码如下：

final void runWorker(Worker w) {

Thread wt = Thread.currentThread();

// 取出Worker中的任务，可能为空

Runnable task = w.firstTask;

w.firstTask = null;

w.unlock(); // allow interrupts

boolean completedAbruptly = true;

try {

// task不为null或者阻塞队列中有任务，通过循环不断的从阻塞队列中取出任务执行

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

w.lock();

// ...

try {

// 任务执行前的hook点

beforeExecute(wt, task);

try {

// 执行任务

task.run();

// 任务执行后的hook点

afterExecute(task, null);

} catch (Throwable ex) {

afterExecute(task, ex);

throw ex;

}

} finally {

task = null;

w.completedTasks++;

w.unlock();

}

}

completedAbruptly = false;

} finally {

// 超时没有取到任务，则回收空闲超时的线程

processWorkerExit(w, completedAbruptly);

}

}

可以看到，runWorker的核心逻辑就是不断通过getTask方法从阻塞队列中获取任务并执行.通过这样的方式实现了线程的复用，避免了创建线程。这里要注意的是这里是一个“生产者-消费者”模式，getTask是从阻塞队列中取任务，所以如果阻塞队列中没有任务的时候就会处于阻塞状态。getTask中通过判断是否要回收线程而设置了等待超时时间，如果阻塞队列中一直没有任务，那么在等待keepAliveTime时间后会返回一个null。最终会走到上述代码的finally方法中，意味着有线程空闲时间超过了keepAliveTime时间，那么调用processWorkerExit方法移除Worker。processWorkerExit方法中没有复杂难以理解的逻辑，这里就不再贴代码了。我们重点看下getTask中是如何处理的，代码如下：

private Runnable getTask() {

boolean timedOut = false; // Did the last poll() time out?

for (;;) {

int c = ctl.get();

// ...

// Flag1. 如果配置了allowCoreThreadTimeOut==true或者线程池中的

// 线程数大于核心线程数，则timed为true，表示开启指定线程超时后被回收

boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

// ...

try {

// Flag2. 取出阻塞队列中的任务,注意如果timed为true，则会调用阻塞队列的poll方法，

// 并设置超时时间为keepAliveTime，如果超时没有取到任务则会返回null。

Runnable r = timed ?

workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :

workQueue.take();

if (r != null)

return r;

timedOut = true;

} catch (InterruptedException retry) {

timedOut = false;

}

}

}

重点看getTask是如何处理空闲超时的逻辑的。我们知道，回收线程的条件是线程大于核心线程数或者配置了allowCoreThreadTimeOut为true,当线程空闲超时的情况下就会回收线程。上述代码在Flag1处先判断了如果线程池中的线程数大于核心线程数，或者开启了allowCoreThreadTimeOut，那么就需要开启线程空闲超时回收。所有在Flag2处，timed为true的情况下调用了阻塞队列的poll方法，并传入了超时时间为keepAliveTime，poll方法是一个阻塞方法，在没有任务时候回进行阻塞。如果在keepAliveTime时间内，没有获取到任务，那么poll方法就会返回null，结束runWorker的循环。进而执行runWorker方法中回收线程的操作。

这里需要我们理解阻塞队列poll方法的使用，poll方法接受一个时间参数，是一个阻塞操作，在给定的时间内没有获取到数据就返回null。poll方法的核心代码如下：

while (count == 0) {

if (nanos <= 0L)

return null;

nanos = notEmpty.awaitNanos(nanos);

}

其实说白了，阻塞队列就是一个使用ReentrantLock实现的“生产者-消费者”模式，我们在[深入理解Java线程的等待与唤醒机制（二）](https://juejin.cn/post/6980655421497278495/" \o "https://juejin.cn/post/6980655421497278495/" \t "/Users/zpw/Documents\\x/_blank)这篇文章中使用ReentrantLock实现“生产者-消费者”模型其实就是一个简单的阻塞队列，与JDK中的BlockingQueue实现机制类似。

3.ThreadPoolExecutor的拒绝策略

上一小节中我们多次提到线程池的拒绝策略，它是在reject方法中实现的。实现代码也非常简单,代码如下：

final void reject(Runnable command) {

handler.rejectedExecution(command, this);

}

通过调用handler的rejectedExecution方法实现。这里其实就是运用了策略模式，handler是一个RejectedExecutionHandler类型的成员变量，RejectedExecutionHandler是一个接口，只有一个rejectedExecution方法。在实例化线程池时构造方法中传入对应的拒绝策略实例即可。前文已经提到了Java提供的几种默认实现分别为DiscardPolicy、DiscardOldestPolicy、CallerRunsPolicy以及AbortPolicy。

以AbortPolicy直接抛出异常为例，来看下代码实现：

public static class AbortPolicy implements RejectedExecutionHandler {

public AbortPolicy() { }

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

throw new RejectedExecutionException("Task " + r.toString() +

" rejected from " +

e.toString());

}

}

可以看到直接在rejectedExecution方法中抛出了RejectedExecutionException来拒绝任务。其他的几个策略实现也都比较简单，有兴趣可以自己查阅代码。

4.ThreadPoolExecutor的shutdown

调用shutdown方法后，会将线程池标记为SHUTDOWN状态，上边execute的源码可以看出，只有线程池是RUNNING状态才接受任务，因此被标记位SHUTDOWN后，再提交任务会被线程池拒绝。shutdown的代码如下:

public void shutdown() {

final ReentrantLock mainLock = this.mainLock;

mainLock.lock();

try {

//检查是否可以关闭线程

checkShutdownAccess();

// 将线程池状态置为SHUTDOWN状态

advanceRunState(SHUTDOWN);

// 尝试中断空闲线程

interruptIdleWorkers();

// 空方法，线程池关闭的hook点

onShutdown();

} finally {

mainLock.unlock();

}

tryTerminate();

}

private void interruptIdleWorkers() {

interruptIdleWorkers(false);

}

修改线程池为SHUTDOWN状态后，会调用interruptIdleWorkers去中断空闲线程线程，具体实现逻辑是在interruptIdleWorkers(boolean onlyOne)方法中，如下：

private void interruptIdleWorkers(boolean onlyOne) {

final ReentrantLock mainLock = this.mainLock;

mainLock.lock();

try {

for (Worker w : workers) {

Thread t = w.thread;

// 尝试tryLock获取锁，如果拿锁成功说明线程是空闲状态

if (!t.isInterrupted() && w.tryLock()) {

try {

// 中断线程

t.interrupt();

} catch (SecurityException ignore) {

} finally {

w.unlock();

}

}

if (onlyOne)

break;

}

} finally {

mainLock.unlock();

}

}

shutdown的逻辑比较简单，里边做了两件比较重要的事情，即先将线程池状态修改为SHUTDOWN，接着遍历所有Worker，将空闲的Worker进行中断。