使用线程池比手动创建线程好在哪里?
手动创建线程:
线程池创建线程:
使用线程池的好处
线程池的各个参数的含义?
线程池的参数
线程创建的时机
其他参数解析
线程池的四种拒绝策略
新建线程池时可以指定它的任务拒绝策略，例如:
拒绝的两种场景:
四种拒绝策略讲解:
有哪 6 种常见的线程池?
线程池常用的阻塞队列有哪些?
LinkedBlockingQueue
SynchronousQueue
DelayedWorkQueue
为什么不应该自动创建线程池?
FixedThreadPool
SingleThreadExecutor
CachedThreadPool
ScheduledThreadPool 和 SingleThreadScheduledExecutor
合适的线程数量是多少? CPU 核心数和线程数的关系?
线程数量根据任务区分:
如何根据实际需要，定制自己的线程池?
如何正确关闭线程池? shutdown 和 shutdownNow 的区别?
关闭方法:
关闭判断:
线程池实现"线程复用"的原理?
面试题:
线程池的状态
线程池创建多少线程合适?
线程池分段导出:
有序性（不推荐有序，性能不好）:
保证全部导出:
线程池实现线程复用代码:
线程池怎么把处理失败的线程?
ThreadPoolTaskExecutor和ThreadPoolExecutor的区别?
Threadpool怎么定长

使用线程池比手动创建线程好在哪里?

手动创建线程:

示例代码:

```
// for循环新建10个线程
public class TenTask {
    public static void main(String[] args) {
```

```
        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(new Task());
            thread.start();
        }
    }
    static class Task implements Runnable {
        public void run() {
            System.out.println("Thread Name: " + Thread.currentThread().getName())
;
        }
    }
}
```

执行结果：

```
Thread Name: Thread-1
Thread Name: Thread-4
Thread Name: Thread-3
Thread Name: Thread-2
Thread Name: Thread-0
Thread Name: Thread-5
Thread Name: Thread-6
Thread Name: Thread-7
Thread Name: Thread-8
Thread Name: Thread-9
```

打印出来的顺序是错乱的，因为运行的顺序取决于线程调度器，有很大的随机性，这是需要我们注意的地方。

手动创建线程的缺点：创建线程时会产生系统开销，并且每个线程还会占用一定的内存等资源，更重要的是我们创建如此多的线程也会给稳定性带来危害，因为每个系统中，可创建线程的数量是有一个上限的，不可能无限的创建。线程执行完需要被回收，大量的线程又会给垃圾回收带来压力。

线程池创建线程：

线程池思路：用一些固定的线程一直保持工作状态并反复执行任务；固定数量的线程池，假设线程池有 5 个线程，但此时的任务大于 5 个，线程池会让余下的任务进行排队，而不是无限制的扩张线程数量，保障资源不会被过度消耗。

示例代码：

```
//用固定线程数的线程池执行10000个任务
public class ThreadPoolDemo {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10000; i++) {
            service.execute(new Task());
        }
    System.out.println(Thread.currentThread().getName());
    }
    static class Task implements Runnable {
        public void run() {
            System.out.println("Thread Name: " + Thread.currentThread().getName())
;
```

```
        }
    }
}
```

执行结果：

```
Thread Name: pool-1-thread-1
Thread Name: pool-1-thread-2
Thread Name: pool-1-thread-3
Thread Name: pool-1-thread-4
Thread Name: pool-1-thread-5
Thread Name: pool-1-thread-5
Thread Name: pool-1-thread-5
Thread Name: pool-1-thread-5
Thread Name: pool-1-thread-5
...
```

如打印结果所示，打印的线程名始终在 Thread Name: pool-1-thread-1~5 之间变化，并没有超过这个范围，也就证明了线程池不会无限制地扩张线程的数量，始终是这5个线程在工作。

使用线程池的好处

使用线程池比手动创建线程主要有三点好处。

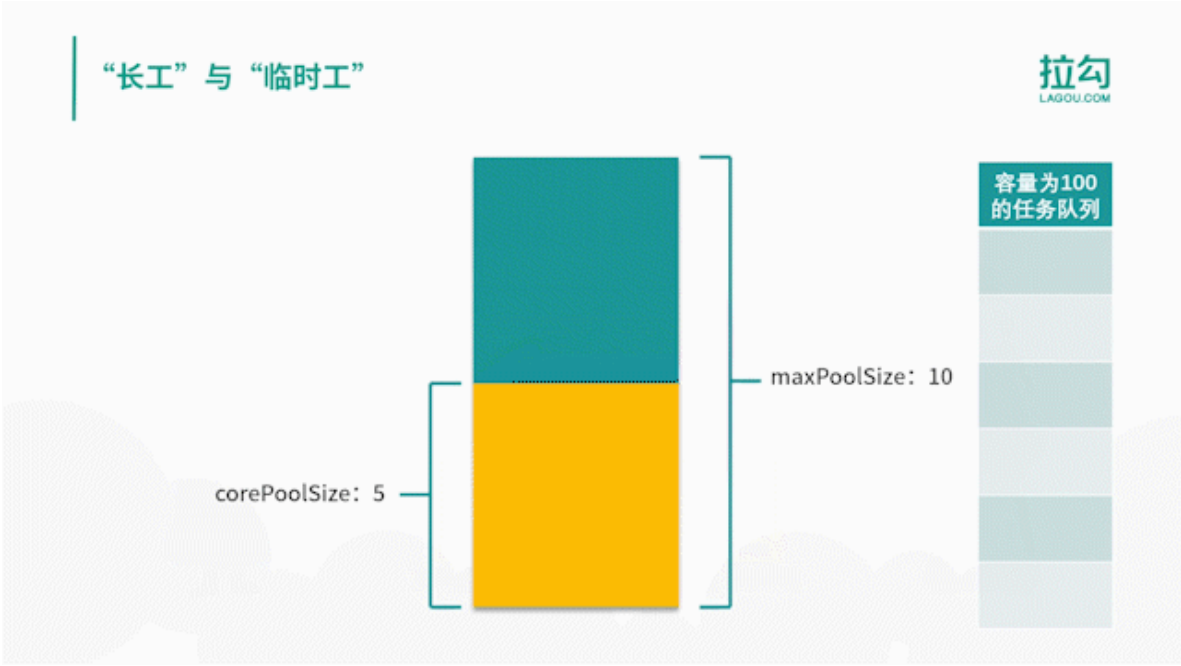
1. 第一点，线程池可以解决线程生命周期的系统开销问题，同时还可以加快响应速度。因为线程池中的线程是可以复用的，我们只用少量的线程去执行大量的任务，这就大大减小了线程生命周期的开销。而且线程通常不是等接到任务后再临时创建，而是已经创建好时刻准备执行任务，这样就消除了线程创建所带来的延迟，提升了响应速度，增强了用户体验。
2. 第二点，线程池可以统筹内存和 CPU 的使用，避免资源使用不当。线程池会根据配置和任务数量灵活地控制线程数量，不够的时候就创建，太多的时候就回收，避免线程过多导致内存溢出，或线程太少导致 CPU 资源浪费，达到了一个完美的平衡。
3. 第三点，线程池可以统一管理资源。比如线程池可以统一管理任务队列和线程，可以统一开始或结束任务，比单个线程逐一处理任务要更方便、更易于管理，同时也有利于数据统计，比如我们可以很方便地统计出已经执行过的任务的数量。

线程池的各个参数的含义？

线程池的参数

参数名	含义
corePoolSize	核心线程数
maxPoolSize	最大线程数
keepAliveTime+时间单位	空闲线程的存活时间
ThreadFactory	线程工厂、用来创建新线程
workQueue	用于存放任务的队列
Handler	处理被拒绝的任务

线程创建的时机



如上图所示，当线程达到corePoolSize（核心线程数）=5时，会将任务加入workQueue（任务队列）中，等待核心线程执行完当前任务后重新从 workQueue 中提取正在等待被执行的任务。如果队列满了则创建新的线程（也就是创建大于核心线程数的线程），如果未来线程有空闲，大于 corePoolSize 的线程会被合理回收。如果达到maximumPoolSize依然不能满足需求，则会拒绝任务。

其他参数解析

keepAliveTime+时间单位：

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。如果后期任务又多了起来，线程池也会根据规则重新创建线程，所以这是一个可伸缩的过程，比较灵活，我们也可以用 setKeepAliveTime 方法动态改变 keepAliveTime 的参数值。

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

workQueue 和 Handler：

分别对应阻塞队列和任务拒绝策略，在后面的课时会对它们进行详细展开讲解。

线程池的四种拒绝策略

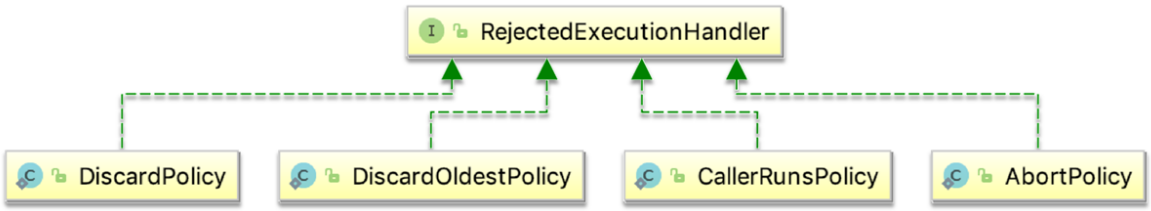
新建线程池时可以指定它的任务拒绝策略，例如：

```
ThreadPoolExecutor(5, 10, 5, TimeUnit.SECONDS, new LinkedBlockingQueue<>(),
   new ThreadPoolExecutor.DiscardOldestPolicy());
```

拒绝的两种场景：

- 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（已经达到maximumPoolSize的值了）。

ThreadPoolExecutor 类中为我们提供了 4 种默认的拒绝策略来应对不同的场景，都实现了 RejectedExecutionHandler 接口，如图所示：



四种拒绝策略讲解：

1. **DiscardPolicy**：当新任务被提交后直接被丢弃掉，也不会给你任何的通知，相对而言存在一定的风险，因为我们提交的时候根本不知道这个任务会被丢弃，可能造成数据丢失。
2. **DiscardOldestPolicy**：如果线程池没被关闭且没有能力执行，则会丢弃任务队列中的头结点，通常是存活时间最长的任务，这种策略与第二种不同之处在于它丢弃的不是最新提交的，而是队列中存活时间最长的，这样就可以腾出空间给新提交的任务，但同理它也存在一定的数据丢失风险。（删除队列中头结点的任务来腾空间给新任务）
3. **CallerRunsPolicy**：当有新任务提交后，如果线程池没被关闭且没有能力执行，则把这个任务交于提交任务的线程执行，也就是谁提交任务，谁就负责执行任务。这样做主要有两点好处：
   - 第一点新提交的任务不会被丢弃，这样也就不会造成业务损失。
   - 第二点好处是，由于谁提交任务谁就要负责执行任务，这样提交任务的线程就得负责执行任务，而执行任务又是比较耗时的，在这段期间，提交任务的线程被占用，也就不会再提交新的任务，减缓了任务提交的速度，相当于是一个负反馈。在此期间，线程池中的线程也可以充分利用这段时间来执行掉一部分任务，腾出一定的空间，相当于是给了线程池一定的缓冲期。
4. **AbortPolicy**：抛出一个类型为 RejectedExecutionException 的 RuntimeException，捕获异常之后可以根据业务逻辑选择重试或提交等策略；

有哪 6 种常见的线程池?

6 种常见的线程池如下：

- **FixedThreadPool**：它的最大线程数和核心线程数一样的，它的特点是线程池中的线程数除了初始阶段需要从 0 开始增加外，之后的线程数量就是固定的，就算超过线程数，也不会在增加线程，而是放入队列中进行等待，就算任务队列满了，也不会增加线程。
- **CachedThreadPool**：称作可缓存线程池，核心线程数是 0，它的最大线程数是 Integer 的最大值（实际最大可以达到 Integer.MAX_VALUE，为 2^31-1，这个数非常大，所以基本不可能达到），而当线程闲置时还可以对线程进行回收。也就是说该线程池的线程数量不是固定不变的，当然它也有一个用于存储提交任务的队列，但这个队列是 SynchronousQueue，队列的容量为0，实际不存储任何任务，它只负责对任务进行中转和传递，所以效率比较高。

  当我们提交一个任务后，线程池会判断已创建的线程中是否有空闲线程，如果有空闲线程则将任务直接指派给空闲线程，如果没有空闲线程，则新建线程去执行任务，这样就做到了动态地新增线程。让我们举个例子，如下方代码所示。

```
ExecutorService service = Executors.newCachedThreadPool();
    for (int i = 0; i < 1000; i++) {
        service.execute(new Task() {
    });
 }
```

  使用 for 循环提交 1000 个任务给 CachedThreadPool，假设这些任务处理的时间非常长，会发生什么情况呢？因为 for 循环提交任务的操作是非常快的，但执行任务却比较耗时，就可能导致 1000 个任务都提交完了但第一个任务还没有被执行完，所以此时 CachedThreadPool 就可以动态的伸缩线程数量，随着任务的提交，不停地创建 1000 个线程来执行任务，而当任务执行完之后，假设没有新的任务了，那么大量的闲置线程又会造成内存资源的浪费，这时线程池就会检测线程在 60 秒内有没有可执行任务，如果没有就会被销毁，最终线程数量会减为 0。

- **ScheduledThreadPool**：它支持定时或周期性执行任务。比如每隔 10 秒钟执行一次任务，而实现这种功能的方法主要有 3 种：

```
ScheduledExecutorService service = Executors.newScheduledThreadPool(10);
//延迟指定时间后执行一次任务，10秒后执行一次任务后就结束
service.schedule(new Task(), 10, TimeUnit.SECONDS);
//以固定的频率执行任务，第二个参数 initialDelay 表示第一次延时时间，第三个参数 period 表示周期，也就是第一次延时后每次延时多长时间执行一次任务。
service.scheduleAtFixedRate(new Task(), 10, 10, TimeUnit.SECONDS);
//也是周期执行任务，区别在于对周期的定义，以任务结束的时间为下一次循环的时间起点开始计时。
service.scheduleWithFixedDelay(new Task(), 10, 10, TimeUnit.SECONDS);
```

  scheduleAtFixedRate是以任务开始时间为时间起点开始计时的：

  - 00:00: 开始喝咖啡
  - 00:10: 喝完了
  - 01:00: 开始喝咖啡
  - 01:10: 喝完了
  - 02:00: 开始喝咖啡
  - 02:10: 喝完了

  scheduleWithFixedDelay是以任务完成的时间为时间起点开始计时的：

- 00:00: 开始喝咖啡
- 00:10: 喝完了
- 01:10: 开始喝咖啡
- 01:20: 喝完了
- 02:20: 开始喝咖啡
- 02:30: 喝完了

- **SingleThreadExecutor**：线程只有一个，如果线程在执行任务的过程中发生异常，线程池也会重新创建一个线程来执行后续的任务。这种线程池由于只有一个线程，所以非常适合用于所有任务都需要按被提交的顺序依次执行的场景，而前几种线程池不一定能够保障任务的执行顺序等于被提交的顺序，因为它们是多线程并行执行的。
- **SingleThreadScheduledExecutor**：和第三种 ScheduledThreadPool 线程池非常相似，它只是 ScheduledThreadPool 的一个特例，内部只有一个线程，它只是将 ScheduledThreadPool 的核心线程数设置为了 1。
- **ForkJoinPool**：这个线程池是在 JDK 7 加入的，非常适合可拆分执行的任务，最后合到一起成最终结果，之前的线程池所有的线程共用一个队列,ForkJoinPool 线程池中每个线程都有自己独立的任务双端队列比如斐波那契数列：

```
class Fibonacci extends RecursiveTask<Integer> {
    int n;
    public Fibonacci(int n) {
        this.n = n;
    }
    @Override
    public Integer compute() {
        if (n <= 1) {
            return n;
        }
    Fibonacci f1 = new Fibonacci(n - 1);
    f1.fork();
    Fibonacci f2 = new Fibonacci(n - 2);
    f2.fork();
    return f1.join() + f2.join();
    }
 }
public static void main(String[] args) throws ExecutionException, InterruptedException {
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    for (int i = 0; i < 10; i++) {
        ForkJoinTask task = forkJoinPool.submit(new Fibonacci(i));
        System.out.println(task.get());
    }
 }
```

先继承了 RecursiveTask，RecursiveTask 类是对ForkJoinTask 的一个简单的包装，fork() 方法分裂任务并分别执行，最后在 return 的时候，使用 join() 方法把结果汇总，这样就实现了任务的分裂和汇总。

执行结果：

```
0
1
1
2
3
5
8
13
21
34
```

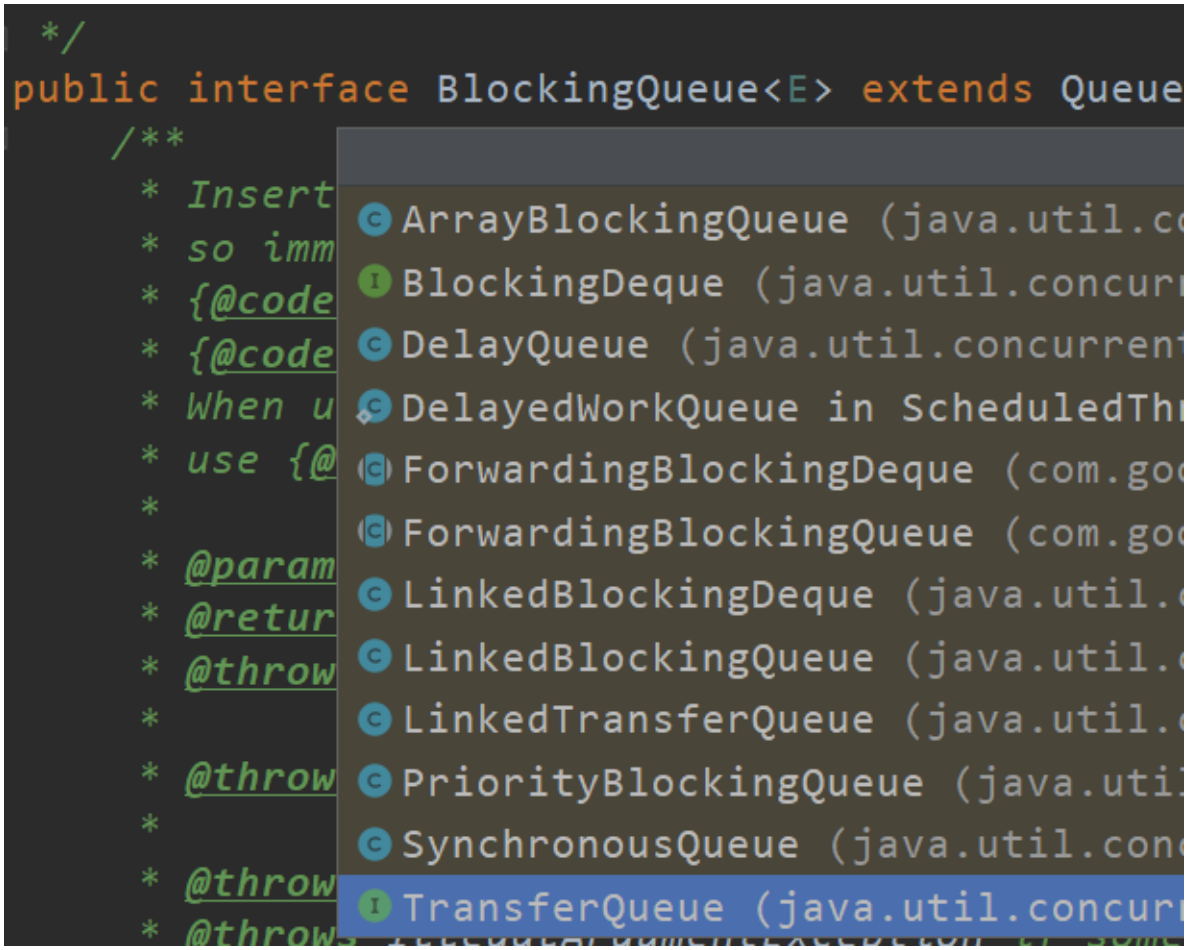
ForkJoinPool 非常适合用于递归的场景，例如树的遍历、最优路径搜索等场景。

线程池常用的阻塞队列有哪些?

线程池的内部结构主要由四部分组成，如图所示。

- 第一部分是线程池管理器，它主要负责管理线程池的创建、销毁、添加任务等管理操作，它是整个线程池的管家。
- 第二部分是工作线程，也就是图中的线程 t0~t9，这些线程勤勤恳恳地从任务队列中获取任务并执行。
- 第三部分是任务队列，作为一种缓冲机制，线程池会把当下没有处理的任务放入任务队列中，由于多线程同时从任务队列中获取任务是并发场景，此时就需要任务队列满足线程安全的要求，所以线程池中任务队列采用 BlockingQueue 来保障线程安全。
- 第四部分是任务，任务要求实现统一的接口，以便工作线程可以处理和执行。

线程池对应的堵塞队列：



FixedThreadPool	LinkedBlockingQueue
SingleThreadExecutor	LinkedBlockingQueue
CachedThreadPool	SynchronousQueue
ScheduledThreadPool	DelayedWorkQueue
SingleThreadScheduledExecutor	DelayedWorkQueue

LinkedBlockingQueue

阻塞队列是容量为 Integer.MAX_VALUE ，可以认为是无界队列。由于线程池的任务队列永远不会放满，所以线程池只会创建核心线程数量的线程，所以此时的最大线程数对线程池来说没有意义，因为并不会触发生成多于核心线程数的线程。

SynchronousQueue

SynchronousQueue队列的容量为0，实际不存储任何任务，对任务直接进行转发。

DelayedWorkQueue

不是按照放入的时间排序，而是会按照延迟的时间长短对任务进行排序，方便任务的执行，内部采用的是“堆”的数据结构。

为什么不应该自动创建线程池?

在本课时我们主要学习为什么不应该自动创建线程池，所谓的自动创建线程池就是直接调用 Executors 的各种方法来生成前面学过的常见的线程池，例如 Executors.newCachedThreadPool()。但这样做是有一定风险的，接下来我们就来逐一分析自动创建线程池可能带来哪些问题。

FixedThreadPool

首先我们来看第一种线程池 FixedThreadPool， 它是线程数量固定的线程池，如源码所示，newFixedThreadPool 内部实际还是调用了 ThreadPoolExecutor 构造函数。

```
1 public static ExecutorService newFixedThreadPool(int nThreads) {
2     return new ThreadPoolExecutor(nThreads, nThreads,0L, TimeUnit.MILLISECONDS,new
  LinkedBlockingQueue<Runnable>());
3 }
```

通过往构造函数中传参，创建了一个核心线程数和最大线程数相等的线程池，它们的数量也就是我们传入的参数，这里的重点是使用的队列是容量没有上限的 LinkedBlockingQueue，如果我们对任务的处理速度比较慢，那么随着请求的增多，队列中堆积的任务也会越来越多，最终大量堆积的任务会占用大量内存，并发生 OOM ，也就是OutOfMemoryError，这几乎会影响到整个程序，会造成很严重的后果。

SingleThreadExecutor

第二种线程池是 SingleThreadExecutor，我们来分析下创建它的源码。

```
1 public static ExecutorService newSingleThreadExecutor() {
2     return new FinalizableDelegatedExecutorService (new ThreadPoolExecutor(1, 1,0L,
  TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>()));
3 }
```

你可以看出，newSingleThreadExecutor 和 newFixedThreadPool 的原理是一样的，只不过把核心线程数和最大线程数都直接设置成了 1，但是任务队列仍是无界的 LinkedBlockingQueue，所以也会导致同样的问题，也就是当任务堆积时，可能会占用大量的内存并导致 OOM。

CachedThreadPool

第三种线程池是 CachedThreadPool，创建它的源码下所示。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,60L, TimeUnit.SECONDS,new SynchronousQueue<Runnable>());
}
```

这里的 CachedThreadPool 和前面两种线程池不一样的地方在于任务队列使用的是 SynchronousQueue，SynchronousQueue 本身并不存储任务，而是对任务直接进行转发，这本身是没有问题的，但你会发现构造函数的第二个参数被设置成了 Integer.MAX_VALUE，这个参数的含义是最大线程数，所以由于 CachedThreadPool 并不限制线程的数量，当任务数量特别多的时候，就可能会导致创建非常多的线程，最终超过了操作系统的上限而无法创建新线程，或者导致内存不足。

ScheduledThreadPool 和 SingleThreadScheduledExecutor

第四种线程池 ScheduledThreadPool 和第五种线程池 SingleThreadScheduledExecutor 的原理是一样的，创建 ScheduledThreadPool 的源码如下所示。

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
```

而这里的 ScheduledThreadPoolExecutor 是 ThreadPoolExecutor 的子类，调用的它的构造方法如下所示。

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,new DelayedWorkQueue());
}
```

我们通过源码可以看出，它采用的任务队列是 DelayedWorkQueue，这是一个延迟队列，同时也是一个无界队列，所以和 LinkedBlockingQueue 一样，如果队列中存放过多的任务，就可能导致 OOM。

你可以看到，这几种自动创建的线程池都存在风险，相比较而言，我们自己手动创建会更好，因为我们可以更加明确线程池的运行规则，不仅可以选择适合自己的线程数量，更可以在必要的时候拒绝新任务的提交，避免资源耗尽的风险。

总结：自动创建线程池容易出现接受的任务过多导致内存溢出（因为即使他们核心线程数只有一个，他的队列是无限大的，也会导致内存溢出）。

合适的线程数量是多少？CPU 核心数和线程数的关系？

线程数量根据任务区分:

- **CPU 密集型任务：**

  最佳的线程数为 CPU 核心数的 1~2 倍，最好还要同时考虑在同一台机器上还有哪些其他会占用过多 CPU 资源的程序在运行，然后对资源使用做整体的平衡。

- **耗时 IO 型任务：**

数据库、文件的读写，网络通信等任务，这种任务的特点是并不会特别消耗 CPU 资源，但是 IO 操作很耗时，总体会占用比较多的时间，这种任务可以遵从以下计算方法：

```
线程数 = CPU 核心数 *（1+平均等待时间/平均工作时间）
```

通过这个公式，我们可以计算出一个合理的线程数量，如果任务的平均等待时间长，线程数就随之增加，而如果平均工作时间长，也就是对于我们上面的 CPU 密集型任务，线程数就随之减少。

可能一段时间是 CPU 密集型，另一段时间是 IO 密集型，或是同时有两种任务相互混搭。那么在这种情况下，我们可以把最大线程数设置成核心线程数的几倍，以便应对任务突发情况。

如果想要更准确的话，可以进行压测，监控 JVM 的线程情况以及 CPU 的负载情况，根据实际情况衡量应该创建的线程数，合理并充分利用资源。

如何根据实际需要，定制自己的线程池？

根据实际需要，比如说并发量、内存大小、是否接受任务被拒绝等一系列因素去定制一个非常适合自己业务的线程池，这样既不会导致内存不足，同时又可以用合适数量的线程来保障任务执行的效率，并在拒绝任务时有所记录方便日后进行追溯。

如何正确关闭线程池？shutdown 和 shutdownNow 的区别？

> - void shutdown;
> - boolean isShutdown;
> - boolean isTerminated;
> - boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;
> - List shutdownNow;

关闭方法：

- **shutdown()：**

  等待任务执行完后关闭，还有新任务提交的话根据拒绝策略直接拒绝后续新提交的任务。

- **shutdownNow()：**

  5 种方法里功能最强大的，

关闭判断：

- **isShutdown()：**

返回 true 或者 false 来判断线程池是否已经开始了关闭工作。（就是线程池是否开启关闭的判断）

- **isTerminated()：**

检测线程池是否真正“终结”了，不仅代表线程池已关闭，同时代表线程池中的所有任务都已经都执行完毕了。（就是线程池完全关闭的判断）

- **awaitTermination()：**

awaitTermination 方法传入的参数是 10 秒，那么它就会陷入 10 秒钟的等待，这个期间会进行线程池的终结：

> 等待期间，如果线程池完全关闭，返回true；
>
> 等待时间到后，还未终结，返回false；
>
> 等待期间线程被中断，方法会抛出 InterruptedException 异常。

线程池实现“线程复用”的原理?

先查看excute源码

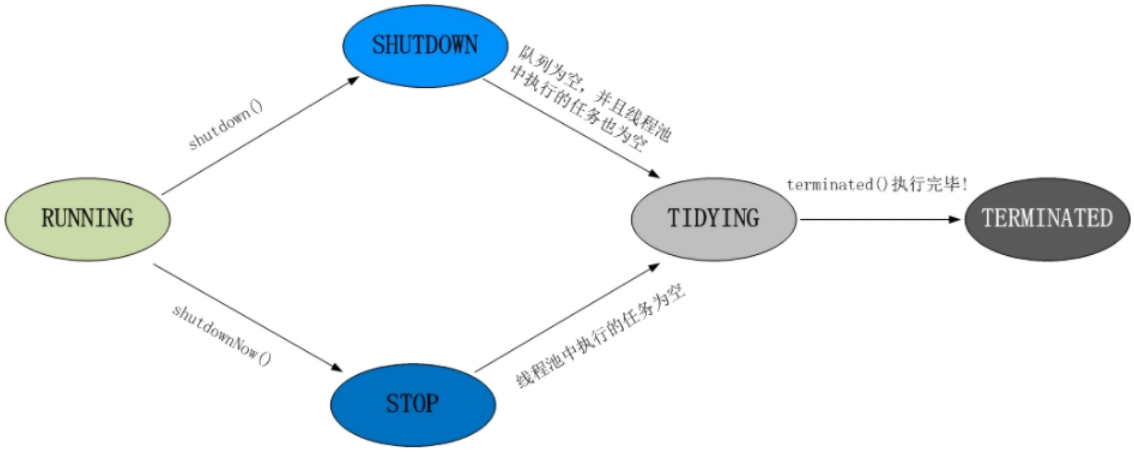
```
public void execute(Runnable command) {
    //如果传入的Runnable的空，就抛出异常
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();
    //判断当前线程数是否小于核心线程数，如果小于核心线程数就调用 addWorker() 方法增加一个 Worker，这里的 Worker 就可以理解为一个线程
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, 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，那么就需要通过 if (isRunning(c) && workQueue.offer(command)) 检查线程池状态是否为Running,如果线程池状态是 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reject(command);
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    //进入到该方法里说明线程池状态为 Running，如果检查当前线程数为 0，那就执行addWorker() 方法新建线程。
    else if (!addWorker(command, false))
        reject(command);
}
```

Worker 类中的 run 方法里执行的 runWorker 方法（这里是实现线程复用的源码，线程通过不停的循环获取待执行的任务）：

```
runWorker(Worker w) {
    Runnable task = w.firstTask;
    //1.通过取 Worker 的 firstTask 或者通过 getTask 方法从 workQueue 中获取待执行的任务。
    //2.直接调用 task 的 run 方法来执行具体的任务（而不是新建线程）。
    while (task != null || (task = getTask()) != null) {
        try {
            task.run();
        } finally {
            task = null;
       }
    }
}
```

面试题：

线程池的状态



如上图：

- **RUNNING**：线程池处在RUNNING状态时，能够接收新任务，以及对已添加的任务进行处理。
- **SHUTDOWN**：线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务。
- **STOP**：线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。
- **TIDYING**：当所有的任务已终止，ctl记录的”任务数量”为0，线程池会变为TIDYING状态。当线程池变为TIDYING状态时，会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。
- **TERMINATED**：线程池彻底终止，就变成TERMINATED状态。

线程池创建多少线程合适?

根据哪个服务来决定，区分 CPU 密集型还是 IO 密集型：CPU 密集型任务应配置尽可能少的线程，如配置 CPU核心数 + 1 个线程的线程池。而对于 IO 密集型任务，它的线程并不是一直在执行任务，则应配置尽可能多的线程，如 CPU核心数 * 2 ，rpc和数据库读写属于IO密集型。

线程池也可以使用临时线程

线程池分段导出:

> 万一有一个导出失败、以及怎么保证全部导出、有序性

有序性（不推荐有序，性能不好）：

- 使用newSingleThreadExecutor线程池，由于核心线程数只有一个，所以能够顺序执行。
- 使用Thread原生方法join
- 使用线程间通信的等待/通知机制
- 使用Conditon
- 使用线程的CountDownLatch
- 使用cyclicbarrier （多个线程互相等待，直到到达同一个同步点，再继续一起执行）
- 使用信号量 Semaphore

保证全部导出：

1. 使用线程的CountDownLatch：

   优点：

   - 操作相对简便，可以把等待线程池中任务完成后的后续工作做成任务，同样放到线程池中运行，简单来说，就是可以控制线程池中任务执行的顺序。

   缺点：

   - 需要提前知道任务的数量。
   - 不可以重复使用
2. 使用线程的CyclicBarrier

3. 使用线程池的原生函数isTerminated()：

   优点：操作简便；

   缺点：需要主线程阻塞；

   executor提供一个原生函数isTerminated()来判断线程池中的任务是否全部完成。全部完成返回true，否则返回false。

4. Future判断任务执行状态：

项目中实现线程池：

```
public class ThreadPoolExecutorEventQueue implements EventQueue {
    private int corePoolSize = 5;  //核心线程
    private int maximumPoolSize = 10; //最大线程
    private long keepAliveTime = 0L;
    private int queueSize = 128; //最大队列
    private int warningQueueDepth = 2;
    private Processor<Event> processor;
    protected Log log = LogFactory.getLog(this.getClass());
    private Interceptor<Event> interceptor;
    //使用引用类型原子类保证原子性
    private AtomicReference<ThreadPoolExecutor> esReference = new
AtomicReference();
    private String queueName;

    public void setInterceptor(Interceptor<Event> interceptor) {
        this.interceptor = interceptor;
    }

    public ThreadPoolExecutorEventQueue() {
    }

    public ThreadPoolExecutorEventQueue(String queueName, int corePoolSize, int
maximumPoolSize, long keepAliveTime, int queueSize, int warningQueueDepth,
Processor<Event> processor, Interceptor<Event> interceptor) {
        this.queueName = queueName;
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
```

```
        this.keepAliveTime = keepAliveTime;
        this.queueSize = queueSize;
        this.warningQueueDepth = warningQueueDepth;
        this.processor = processor;
        this.interceptor = interceptor;
    }

    public ThreadPoolExecutor getThreadPoolExecutor() {
        return (ThreadPoolExecutor)this.esReference.get();
    }

    public void put(final Event event) {
        ThreadPoolExecutor es = (ThreadPoolExecutor)this.esReference.get();
        if (es == null) {
            //设置LinkedBlockingQueue类型的消息队列
            ThreadPoolExecutor te = new ThreadPoolExecutor(this.corePoolSize,
this.maximumPoolSize, this.keepAliveTime, TimeUnit.MILLISECONDS, new
LinkedBlockingQueue(this.queueSize), new SimpleThreadFactory(this.queueName,
true));
            //设置AbortPolicy的拒绝策略（满了就抛错然后重试）
            te.setRejectedExecutionHandler(new AbortPolicy());
            //通过cas判断引用类型是否为空，为空则停止线程
            if (!this.esReference.compareAndSet((Object)null, te)) {
                te.shutdown();
            }

            es = (ThreadPoolExecutor)this.esReference.get();
        }

        if (this.log.isInfoEnabled() && this.warningQueueDepth > 0) {
            int size = es.getQueue().size();
            if (size > this.warningQueueDepth && size != this.queueSize) {
                this.log.info(String.format("WARN#%s#%s#%s#%d", this.queueName,
event.getTarget().getId(), event.getId(), size));
            }
        }

        try {
            es.submit(new Runnable() {
                public void run() {
                    try {
                        if (ThreadPoolExecutorEventQueue.this.interceptor !=
null) {

 ThreadPoolExecutorEventQueue.this.interceptor.insertParams(event);
                        }

                        if
(ThreadPoolExecutorEventQueue.this.log.isDebugEnabled()) {

 ThreadPoolExecutorEventQueue.this.log.debug(String.format("START#%s#%s#%s",
ThreadPoolExecutorEventQueue.this.queueName, event.getTarget().getId(),
event.getId()));
                        }

                        RpcContext rpcContext = RpcContext.getRpcContext();
                        rpcContext.setSource(event.getSource());
                        rpcContext.setSubject(event.getSubject());
                        rpcContext.setTarget(event.getTarget());
                        rpcContext.setLocale(event.getLocale());
```

```
75
                ThreadPoolExecutorEventQueue.this.processor.process(event);
76                          if
(ThreadPoolExecutorEventQueue.this.log.isDebugEnabled()) {
77
                ThreadPoolExecutorEventQueue.this.log.debug(String.format("STOP#%s#%s#%s",
ThreadPoolExecutorEventQueue.this.queueName, event.getTarget().getId(),
event.getId()));
78                          }
79                      } catch (Throwable var5) {
80
                ThreadPoolExecutorEventQueue.this.log.warn(String.format("ERROR#%s#%s#%s",
ThreadPoolExecutorEventQueue.this.queueName, event.getTarget().getId(),
event.getId()), var5);
81                      } finally {
82                          RpcContext.reset();
83                          if (ThreadPoolExecutorEventQueue.this.interceptor !=
null) {
84
                ThreadPoolExecutorEventQueue.this.interceptor.destroy();
85                          }
86
87                      }
88
89                  }
90              });
91          } catch (RejectedExecutionException var4) {
92              throw new RejectedExecutionException(this.queueName);
93          }
94      }
95
96      public Event get(long timeout) {
97          throw new RuntimeException("unsupported");
98      }
99
100     public void close() throws IOException {
101         ThreadPoolExecutor te =
(ThreadPoolExecutor)this.esReference.getAndSet((Object)null);
102         if (te != null) {
103             te.shutdown();
104         }
105
106     }
107 }
```

线程池实现线程复用代码：

Worker 类中的 run 方法里执行的 runWorker 方法（这里是实现线程复用的源码，线程通过不停的循环获取待执行的任务）：

```
runWorker(Worker w) {
    Runnable task = w.firstTask;
    //1.通过取 Worker 的 firstTask 或者通过 getTask 方法从 workQueue 中获取待执行的任务。
    //2.直接调用 task 的 run 方法来执行具体的任务（而不是新建线程）。
    while (task != null || (task = getTask()) != null) {
        try {
            task.run();
        } finally {
            task = null;
       }
    }
}
```

线程池怎么把处理失败的线程?

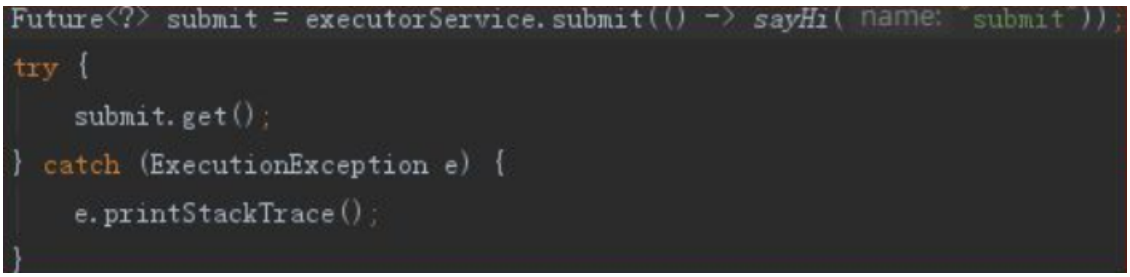
参考博文：https://www.cnblogs.com/fanguangdexiaoyuer/p/12332082.html

```
public class Test {

    public static void main(String[] args) throws Exception {
        ThreadPoolTaskExecutor executor = init();
        executor.execute(() -> sayHi("execute"));
        Thread.sleep(1000);
        executor.submit(() -> sayHi("submit"));
    }

    public static void sayHi(String name) {
        String printStr = "thread-name:" + Thread.currentThread().getName() + ",执行方式：" + name;
        System.out.println(printStr);
        throw new RuntimeException(printStr + " error!!!");
    }

    private static ThreadPoolTaskExecutor init() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setThreadNamePrefix("thread_");
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(1000);
        executor.setKeepAliveSeconds(30);
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
        executor.initialize();;
        return executor;
    }
}
```

以上代码的结果：

```
thread-name:thread_1,执行方式：execute
Exception in thread "thread_1" java.lang.RuntimeException: thread-name:thread_1,执行方式：execute error!!!
    at com.tgb.spring.aop.Test.sayHi(Test.java:28)
    at com.tgb.spring.aop.Test.lambda$main$0(Test.java:20) <2 internal calls>
    at java.lang.Thread.run(Thread.java:748)
thread-name:thread_3,执行方式：submit
```

execute()方式会抛异常，submit()方式不会抛异常，要使得submit()方式抛异常的话需要以下改造



总结：

1、当执行方式是execute时,可以看到堆栈异常的输出

原因：ThreadPoolExecutor.runWorker()方法中，task.run()，即执行我们的方法，如果异常的话会throw x;所以可以看到异常。

2、当执行方式是submit时,堆栈异常没有输出。但是调用Future.get()方法时，可以捕获到异常

原因：ThreadPoolExecutor.runWorker()方法中，task.run()，其实还会继续执行FutureTask.run()方法，再在此方法中c.call()调用我们的方法，
如果报错是setException()，并没有抛出异常。当我们去get()时，会将异常抛出。

3、不会影响线程池里面其他线程的正常执行

4、线程池会把这个线程移除掉，并创建一个新的线程放到线程池中

当线程异常，会调用ThreadPoolExecutor.runWorker()方法最后面的finally中的processWorkerExit()，会将此线程remove，并重新addworker()一个线程。

execute源码执行流程

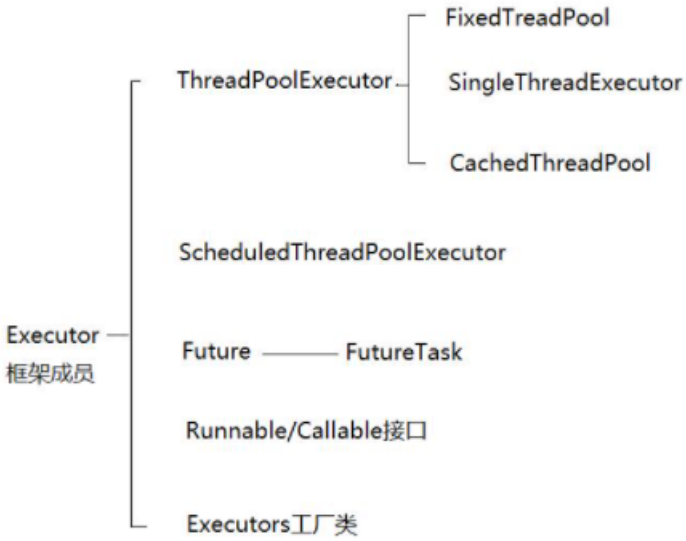
1、开始执行任务，新增或者获取一个线程去执行任务(比如刚开始是新增coreThread去执行任务)。执行到task.run()时会去执行提交的任务。
如果任务执行失败，或throw x抛出异常。
2、之后会到finally中的afterExecute()扩展方法，我们可以扩展该方法对异常做些什么。
3、之后因为线程执行异常会跳出runWorker的外层循环，进入到processWorkerExit()方法，此方法会将执行任务失败的线程删除，并新增一个线程。
4、之后会到ThreadGroup#uncaughtException方法，进行异常处理。
如果没有通过setUncaughtExceptionHandler()方法设置默认的UncaughtExceptionHandler，就会在uncaughtException()方法中打印出异常信息。

submit源码执行流程

1、将传进来的任务封装成FutureTask，同样走execute的方法调用，然后直接返回FutureTask。
2、开始执行任务，新增或者获取一个线程去执行任务(比如刚开始是新增coreThread去执行任务)。
3、执行到task.run()时，因为是FutureTask，所以会去调用FutureTask.run()。
4、在FutureTask.run()中，c.call()执行提交的任务。如果抛出异常，并不会throw x，而是setException()保存异常。
5、当我们阻塞获取submit()方法结果时get()，才会将异常信息抛出。当然因为runWorker()没有抛出异常，所以并不会删除线程。

ThreadPoolTaskExecutor和ThreadPoolExecutor的区别?

ThreadPoolTaskExecutor本质依然是ThreadPoolExecutor来实现基本的线程池工作，不同的是前者更关注自己实现的增强扩展部分，让线程池具有更多特性可供使用



Threadpool怎么定长

https://www.cnblogs.com/sxkgeek/p/9343519.html