##### 1. 线程池基础知识

线程池就是管理一系列线程的资源池，其提供了一种限制和管理线程资源的方式。

好处：

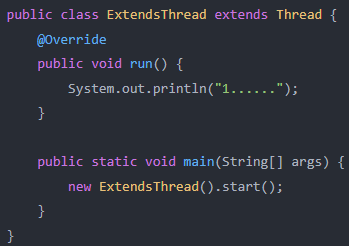
① **降低资源消耗**。重复利用已创建的线程降低线程创建和销毁造成的消耗。

② **提高响应速度**。当任务到达时，不需要等到线程创建就能立即执行。

③ **提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性。线程池可以进行统一的分配，调优和监控。

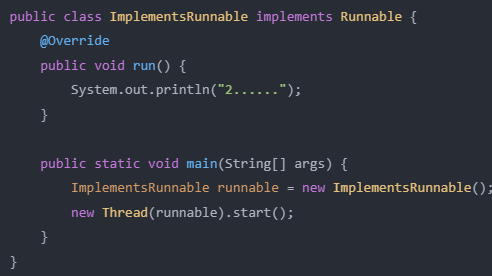
Java创建线程最简单的三种方法：

①继承Thread类



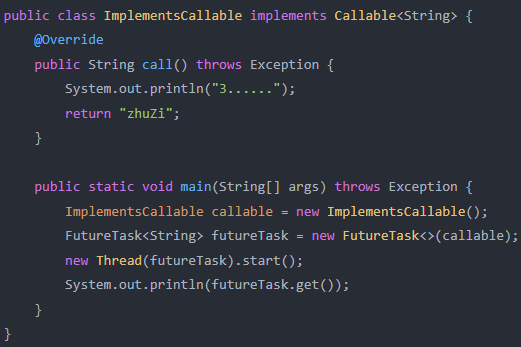
②实现Runnable接口

借助Thread并调用start方法实现，如果直接run不会创建新的线程，还是用原来的。



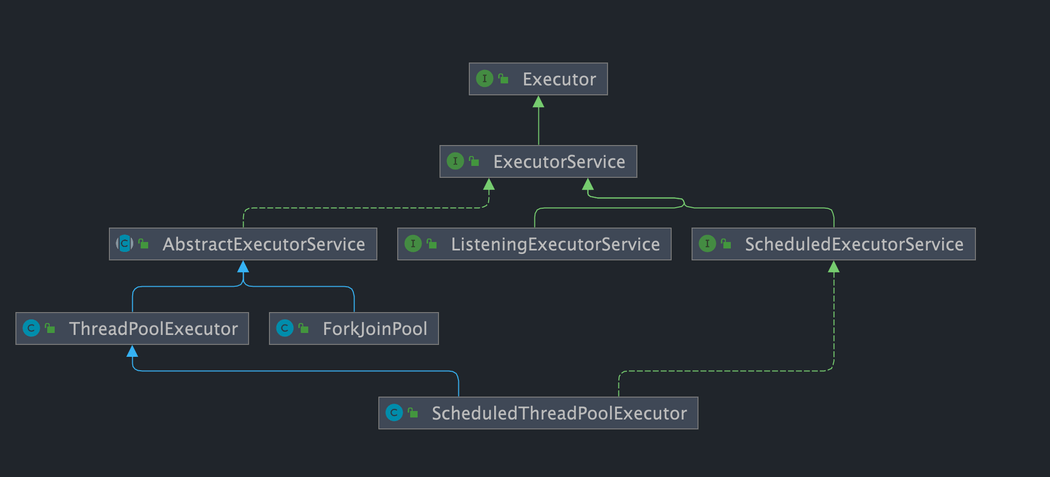
③实现Callable接口

借助Thread并调用start方法实现，如果直接run不会创建新的线程，还是用原来的。此外，借助FutureTask获取返回值。



##### 2. Executor框架

Executor框架是一个强大的工具，用于管理和控制线程的执行。它提供了一种高层次的线程管理机制，使得开发者可以更方便地处理线程池、任务调度和并发编程。



Executor 框架结构主要由三大部分组成：

① 任务（Runnable / Callable）

执行任务需要实现的 Runnable 接口 或 Callable接口。Runnable 接口或 Callable 接口 实现类都可以被 ThreadPoolExecutor 或 ScheduledThreadPoolExecutor 执行。

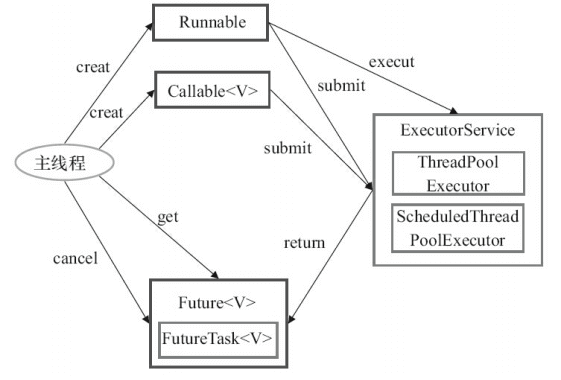
② 执行器（Executor）

包括任务执行机制的核心接口 Executor ，以及继承自 Executor 接口的 ExecutorService 接口。ThreadPoolExecutor 和 ScheduledThreadPoolExecutor 这两个关键类实现了 ExecutorService 接口。

③ 异步计算的结果

Future 接口以及 Future 接口的实现类 FutureTask 类都可以代表异步计算的结果。当我们把Runnable接口或Callable接口的实现类提交给ThreadPoolExecutor或 ScheduledThreadPoolExecutor 执行。（调用 submit() 方法时会返回一个 FutureTask 对象）

框架使用流程：



① 主线程首先要创建实现 Runnable 或者 Callable 接口的任务对象。

② 把创建完成的实现 Runnable接口的对象直接交给 ExecutorService 执行: ExecutorService.execute（Runnable command）。或者也可以把 Runnable 对象或Callable 对象提交给 ExecutorService 执行（ExecutorService.submit（Runnable task）或 ExecutorService.submit（Callable <T> task））。

③ 如果执行 ExecutorService.submit（…），ExecutorService 将返回一个实现Future接口的对象（我们刚刚也提到过了执行 execute()方法和 submit()方法的区别，submit()会返回一个 FutureTask 对象）。由于 FutureTask 实现了 Runnable，我们也可以创建 FutureTask，然后直接交给 ExecutorService 执行。

④ 最后，主线程可以执行 FutureTask.get()方法来等待任务执行完成。主线程也可以执行 FutureTask.cancel（boolean mayInterruptIfRunning）来取消此任务的执行。

##### 3. ThreadPoolExecutor类



三个重要参数：

① corePoolSize : 任务队列未达到队列容量时，最大可以同时运行的线程数量。

② maximumPoolSize : 任务队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。

③ workQueue: 新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。

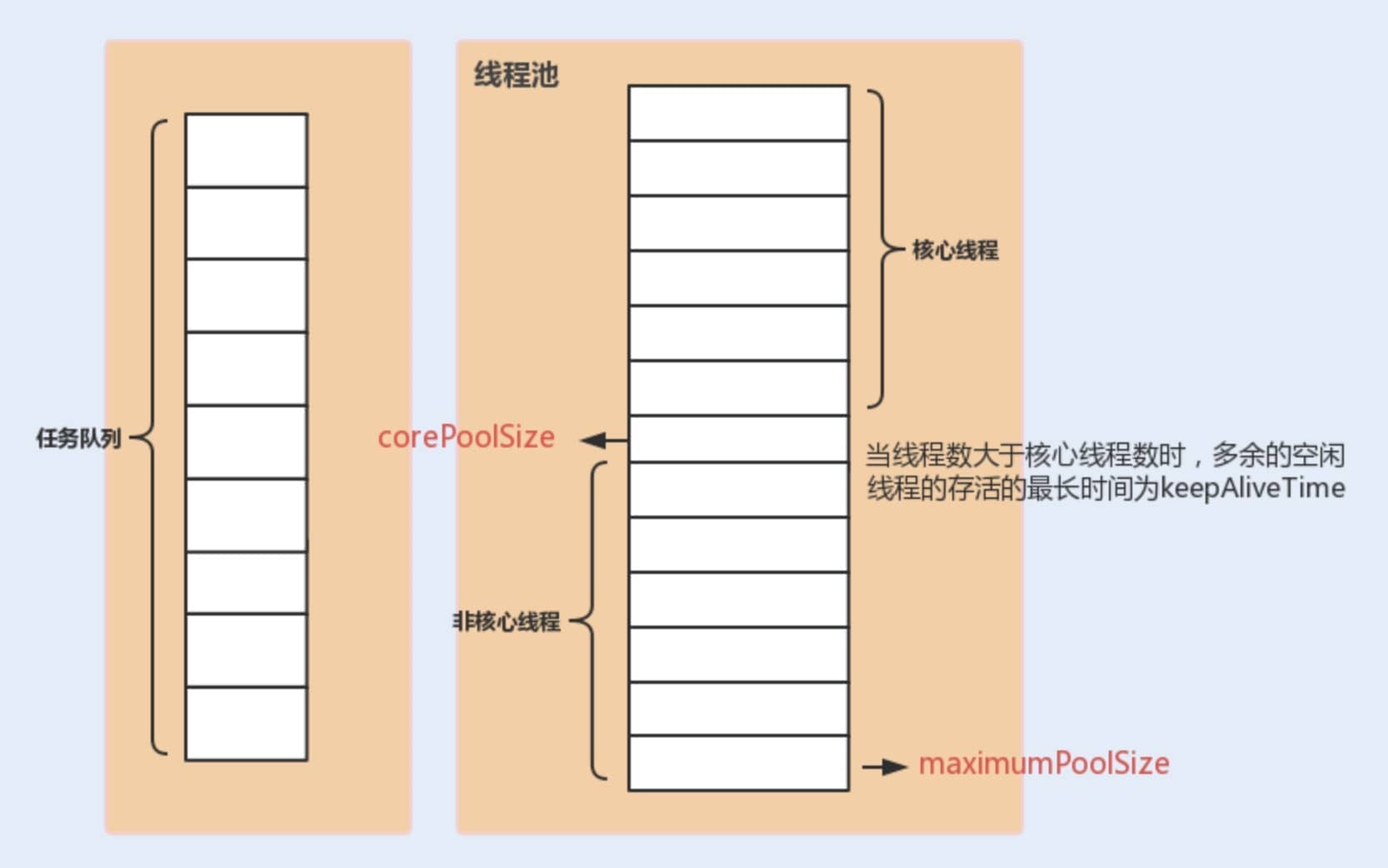
其他参数：

④ keepAliveTime:线程池中的线程数量大于 corePoolSize 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 keepAliveTime才会被回收销毁。

⑤ unit : keepAliveTime 参数的时间单位。

⑥ threadFactory :executor 创建新线程的时候会用到。

⑦ handler :拒绝策略。



**拒绝策略**定义：

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任务时，ThreadPoolExecutor 定义一些策略:

① ThreadPoolExecutor.AbortPolicy：抛出 RejectedExecutionException来拒绝新任务的处理。

② ThreadPoolExecutor.CallerRunsPolicy：调用执行自己的线程运行任务，也就是**直接在调用execute方法的线程中运行(run)被拒绝的任务**，如果执行程序已关闭，则会丢弃该任务。因此这种策略会降低对于新任务提交速度，影响程序的整体性能。如果您的应用程序可以承受此延迟并且你要求任何一个任务请求都要被执行的话，你可以选择这个策略。

③ ThreadPoolExecutor.DiscardPolicy：不处理新任务，直接丢弃掉。

④ ThreadPoolExecutor.DiscardOldestPolicy：此策略将丢弃最早的未处理的任务请求。

##### 4. 线程池创建的两种方式

① 通过ThreadPoolExecutor构造函数来创建（推荐）。

② 通过 Executor 框架的工具类 Executors 来创建。

《阿里巴巴 Java 开发手册》强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 构造函数的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

通过Executors工具类可以创建多种类型的线程池，包括：

**FixedThreadPool**

固定线程数量的线程池。该线程池中的线程数量始终不变（因为corePoolSize= maximumPoolSize）。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。

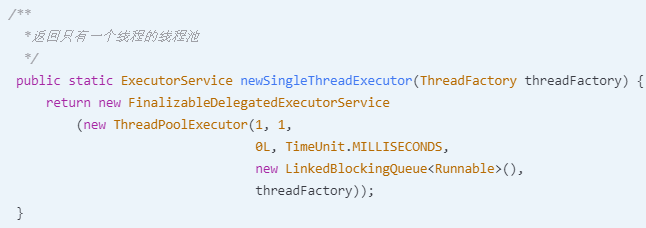


缺点：

FixedThreadPool 使用**无界队列** LinkedBlockingQueue（队列的容量为 Integer.MAX\_VALUE）作为线程池的工作队列。运行中的 FixedThreadPool（未执行 shutdown()或 shutdownNow()）**不会拒绝任务**，在任务比较多的时候会导致 OOM（内存溢出）。

**SingleThreadExecutor**

只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。



缺点：

同FixedThreadPool，使用**无界队列**，且只有一个线程

**CachedThreadPool**

可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。使用**同步队列**SynchronousQueue，没有容量，不存储元素。



缺点：

corePoolSize 被设置为空（0），maximumPoolSize被设置为 Integer.MAX.VALUE，即它是无界的，这也就意味着如果主线程提交任务的速度高于 maximumPool 中线程处理任务的速度时，CachedThreadPool 会不断创建新的线程。极端情况下，这样会导致耗尽 cpu 和内存资源。

**ScheduledThreadPool**

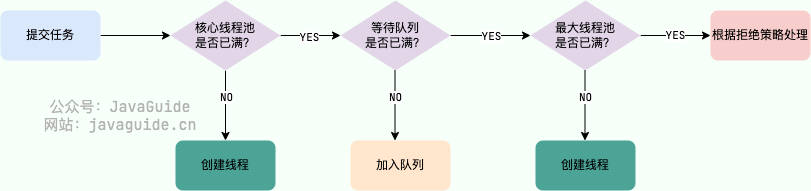
给定的延迟后运行任务或者定期执行任务的线程池。



（注意super就是ThreadPoolExecutor，而上面三个线程池类并没有继承，而是直接new）

使用DelayedWorkQueue（**延迟阻塞队列**）作为线程池的任务队列。DelayedWorkQueue 的内部元素并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**，内部采用的是“堆”的数据结构，可以保证每次出队的任务都是当前队列中执行时间最靠前的。**DelayedWorkQueue 添加元素满了之后会自动扩容原来容量的 1/2，即永远不会阻塞**，最大扩容可达 Integer.MAX\_VALUE，所以最多只能创建核心线程数的线程。

##### 5. 线程池原理分析



流程：

① 如果当前运行的线程数小于核心线程数，那么就会新建一个线程来执行任务。

② 如果当前运行的线程数等于或大于核心线程数，但是小于最大线程数，那么就把该任务放入到任务队列里等待执行。

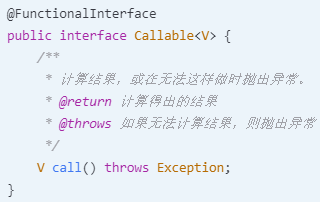
③ 如果向任务队列投放任务失败（任务队列已经满了），但是当前运行的线程数是小于最大线程数的，就新建一个线程来执行任务。

④ 如果当前运行的线程数已经等同于最大线程数了，新建线程将会使当前运行的线程超出最大线程数，那么当前任务会被拒绝，拒绝策略会调用RejectedExecutionHandler.rejectedExecution()方法。

##### 6. 几个常见对比

**① Runnable vs Callable**

Runnable自 Java 1.0 以来一直存在，但Callable仅在 Java 1.5 中引入,目的就是为了来处理Runnable不支持的用例。**Runnable 接口不会返回结果或抛出检查异常，但是 Callable 接口可以。**所以，如果任务不需要返回结果或抛出异常推荐使用 Runnable 接口，这样代码看起来会更加简洁。



**② execute() vs submit()**

两种提交任务到线程池的方法。区别：

返回值：

execute() 方法用于提交不需要返回值的任务。通常用于执行 Runnable 任务，无法判断任务是否被线程池成功执行。

submit() 方法用于提交需要返回值的任务。可以提交 Runnable 或 Callable 任务。submit() 方法返回一个 Future 对象，通过这个 Future 对象可以判断任务是否执行成功，并获取任务的返回值（get()方法会阻塞当前线程直到任务完成， get（long timeout，TimeUnit unit）多了一个超时时间，如果在 timeout 时间内任务还没有执行完，就会抛出 java.util.concurrent.TimeoutException）。

异常处理：

execute()：而在使用 execute() 方法时，异常处理需要通过自定义的 ThreadFactory （在线程工厂创建线程的时候设置UncaughtExceptionHandler对象来 处理异常）或 ThreadPoolExecutor 的 afterExecute() 方法来处理

submit()：在使用 submit() 方法时，可以通过 Future 对象处理任务执行过程中抛出的异常

**③ shutdown()VSshutdownNow()**

shutdown() :关闭线程池，线程池的状态变为 SHUTDOWN。线程池不再接受新任务了，但是队列里的任务得执行完毕。

shutdownNow() :关闭线程池，线程池的状态变为 STOP。线程池会终止当前正在运行的任务，并停止处理排队的任务并返回正在等待执行的 List。

**④ isTerminated() VS isShutdown()**

isShutDown 当调用 shutdown() 方法后返回为 true。

isTerminated 当调用 shutdown() 方法后，并且所有提交的任务完成后返回为 true