<https://cloud.tencent.com/developer/article/1362835>

java创建线程有哪几种方法

1. 继承 Thread 类，然后调用 start 方法。

class SampleThread extends Thread {

//重写run方法，线程运行后，跑的就是run方法

public void run(){

//System.out.println("");

}

public static void main(String[] args){

Thread t1 = new SampleThread();

Thread t2 = new SampleThread();

t1.start(); //线程运行,调用的 run()方法.

t2.start(); //线程运行,调用的 run()方法..

}

}

2. 实现 Runnable 接口的 run 方法, 然后再用 Thread 类包裹后，调用 start 方法。

class A implements Runnable{

@Override

public void run() {

// implement run method here

}

public static void main() {

final A obj = new A();

Thread t1 = new Thread(new A());

t1.start();

}

}

当传入一个Runnable target参数给Thread后，Thread的run()方法就会调用target.run()，参考JDK源代码：

public void run() {

　　if (target != null) {

　　 target.run();

　　}

}

3. 实现 Callable 接口的 call 方法，用 FutureTask 类包裹 Callable 对象。然后再用 Thread 类包裹 FutureTask 类，并调用 start 方法。call() 方法可以有返回值。

class MyCallable implements Callable {

@Override

public Integer call() throws Exception {

int sum = 0;

for (int i = 1; i <= 100; i++) {

sum += i;

}

return sum;

}

public static void main(String[] args) throws Exception {

MyCallable mc = new MyCallable(); //实例化 callable

FutureTask oneTask = new FutureTask(mc); //用FutureTask包裹

Thread oneThread = new Thread(oneTask); //用Thread包裹

oneThread.start();

System.out.print(oneTask.get()); //获取返回值

}

}

Callable 方法在 Java 8 后，支持拉姆达表达式的写法，可以创建一个 FutureTask 类，语句上不是太罗嗦。 Callable 方式有以下几个优点：

* 可以捕获线程上的异常。
* 可以通过 get 方法得到返回值。
* get 方法阻塞当前线程，直到调用的线程运行结束。
* 可以取消线程的运行。

Java 6 之后，还可以通过创建线程池来创建线程，使用 ExecutorService 的 execute 方法:

ExecutorService es = Executors.newCachedThreadPool();

Runnable r = <your runnable here>;

es.execute(r);

方式间的对比：

1.直接继承Thread和实现Runnable无法获取执行结果。

2.从Java 1.5开始，就提供了Callable和Future，通过它们可以在任务执行完毕之后得到任务执行结果。

3.Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

4.FutureTask类实现了RunnableFuture接口，RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。FutureTask是Future接口的一个唯一实现类。  
5.　如果为了可取消性而使用 Future 但又不提供可用的结果，则可以声明 Future<?> 形式类型、并返回 null 作为底层任务的结果。

# 线程池用过吗？都有什么参数？底层如何实现的？？

https://blog.csdn.net/z285692001/article/details/81916470

1.corePoolSize：核心池的大小，在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务， 除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。

默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

2.maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize； 当阻塞队列是无界队列, 则maximumPoolSize则不起作用, 因为无法提交至核心线程池的线程会一直持续地放入workQueue.

3.keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，

如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用， 直到线程池中的线程数为0；

4.unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

5.workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue;

LinkedBlockingQueue;

SynchronousQueue;

　　ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

(1) ArrayBlockingQueue：基于数组结构的有界阻塞队列，按FIFO排序任务；

(2) LinkedBlockingQueue：基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于ArrayBlockingQuene； 是无界的，可以不指定队列的大小，但是默认是Integer.MAX\_VALUE。当然也可以指定队列大小，从而成为有界的。

(3) SynchronousQueue：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene； 如果应用程序确实需要比较大的工作队列容量，

而又想避免无界工作队列可能导致的问题，不妨考虑SynchronousQueue。SynchronousQueue实现上并不使用缓存空间。

使用SynchronousQueue的目的就是保证“对于提交的任务，如果有空闲线程，则使用空闲线程来处理；否则新建一个线程来处理任务”。https://segmentfault.com/a/1190000011207824

(4) priorityBlockingQueue：具有优先级的无界阻塞队列；

6.threadFactory：线程工厂，主要用来创建线程；创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名。默认为DefaultThreadFactory

7.handler：表示当拒绝处理任务时的策略，有以下四种取值：

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

AbortPolicy：直接抛出异常，默认策略；

CallerRunsPolicy：用调用者所在的线程来执行任务；

DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

**Java通过Executors提供四种线程池，分别为：**

1.newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

2.newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。定长线程池的大小最好根据系统资源进行设置。

如Runtime.getRuntime().availableProcessors()。

3.newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

4.newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

  对于 java.util.concurrent.BlockingQueue 类有有三种方法将线程添加到线程队列里面，然而如何区别三种方法的不同呢，其实在队列未满的情况下结果相同，都是将线程添加到线程队列里面，区分就在于当线程队列已经满的时候，此时

public boolean add(E e) 方法将抛出IllegalStateException异常，说明队列已满。

public boolean offer(E e) 方法则不会抛异常，只会返回boolean值，告诉你添加成功与否，队列已满，当然返回false。

public void put(E e) throws InterruptedException 方法则一直阻塞（即等待，直到线程池中有线程运行完毕，可以加入队列为止）。

add，offer，put三种添加线程到队列的方法只在队列满的时候有区别，add为抛异常，offer返回boolean值，put直到添加成功为止。

5.同理remove，poll， take三种移除队列中线程的方法只在队列为空的时候有区别， remove为抛异常，poll为返回boolean值， take等待直到有线程可以被移除。

看看下面这张图就清楚了：



关闭线程池，可以通过shutdown和shutdownNow这两个方法。它们的原理都是遍历线程池中所有的线程，然后依次中断线程。shutdown和shutdownNow还是有不一样的地方：

1. shutdownNow首先将线程池的状态设置为STOP,然后尝试停止所有的正在执行和未执行任务的线程，并返回等待执行任务的列表；
2. shutdown只是将线程池的状态设置为SHUTDOWN状态，然后中断所有没有正在执行任务的线程

可以看出shutdown方法会将正在执行的任务继续执行完，而shutdownNow会直接中断正在执行的任务。调用了这两个方法的任意一个，isShutdown方法都会返回true，当所有的线程都关闭成功，才表示线程池成功关闭，这时调用isTerminated方法才会返回true。

为了避免重复的创建线程，线程池的出现可以让线程进行复用。通俗点讲，当有工作来，就会向线程池拿一个线程，当工作完成后，并不是直接关闭线程，而是将这个线程归还给线程池供其他任务使用。