简述

线程的创建与销毁都需要销毁资源，为了避免频繁的创建与销毁线程，可以让创建的线程进行复用。类似数据库连接池的概念

线程池的分类

Java通过Executors提供四种线程池，分别为：

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

缓存线程池(CachedThreadPool)

当一个线程处理的过来的时候

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExecutorTest {
    public static void main(String[] args) {
        ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
```

```
            final int index = i;
            try {
                Thread.sleep(index * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            cachedThreadPool.execute(new Runnable() {
                public void run() {
                    System.out.println(Thread.currentThread().getId() + " : " + index);
                }
            });
        }
        cachedThreadPool.shutdown();
    }
}

打印结果发现 执行上一个线程处理的过来打印，所以都是同一个线程完成任务
/* 输出
9 : 0
9 : 1
9 : 2
9 : 3
9 : 4
9 : 5
9 : 6
9 : 7
9 : 8
9 : 9
*/
```

当一个线程处理不过来的时候

```

public class ThreadPoolExecutorTest {
    public static void main(String[] args) {
        ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
            final int index = i;
            cachedThreadPool.execute(new Runnable() {
                public void run() {
                    System.out.println(Thread.currentThread().getId() + " : " + index);
                }
            });
        }
        cachedThreadPool.shutdown();
    }
}

/* 输出
9 : 0
12 : 3
10 : 1
11 : 2
14 : 5
```

```
15 : 6
13 : 4
16 : 7
14 : 8
16 : 9
*/
```

newFixedThreadPool定长线程池

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExecutorTest {
    public static void main(String[] args) {
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 10; i++) {
            final int index = i;
            fixedThreadPool.execute(new Runnable() {
                public void run() {
                    try {
                        System.out.println(Thread.currentThread().getId() + " : " + index);
                        Thread.sleep(2000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }
        fixedThreadPool.shutdown();
    }
}

打印结果 始终只有三个线程处理，没有来得及处理的排队等待
/*
9 : 0
10 : 1
11 : 2
10 : 3
11 : 5
9 : 4
11 : 6
9 : 8
10 : 7
10 : 9
*/
```

调度计划线程池(ScheduledThreadPool)

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExecutorTest {
```

```
    public static void main(String[] args) {
        ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(10);
        scheduledThreadPool.scheduleAtFixedRate(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getId() + " : delay 3 seconds, and excute
            }
        }, 3, 2, TimeUnit.SECONDS);
    }
}
打印结果：  三秒后开始执行打印，之后没两秒执行一次，不断轮询，每次执行的线程不固定
/*
9 : delay 3 seconds, and excute every 2 seconds
9 : delay 3 seconds, and excute every 2 seconds
11 : delay 3 seconds, and excute every 2 seconds
9 : delay 3 seconds, and excute every 2 seconds
12 : delay 3 seconds, and excute every 2 seconds
11 : delay 3 seconds, and excute every 2 seconds
13 : delay 3 seconds, and excute every 2 seconds
9 : delay 3 seconds, and excute every 2 seconds
*/
```

单线程线程池(SingleThreadExecutor)

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExecutorTest {
    public static void main(String[] args) {
        ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
        for (int i = 0; i < 20; i++) {
            final int index = i;
            singleThreadExecutor.execute(new Runnable() {
                public void run() {
                    try {
                        System.out.println(Thread.currentThread().getId() + " : " + index);
                        Thread.sleep(20);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }
    }
}
```

自定义线程池（ThreadPoolExecutor）

上面四种线程池除了（ScheduledThreadPool）外都继承与ThreadPoolExecutor类，只不过传递的参数不同而已.

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

```
}

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

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

线程池的构造

```
ExecutorService es = new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, unit, worl
```

- corePoolSize：指定线程池中核心线程数量
  核心线程会一直存活，及时没有任务需要执行
  当线程数小于核心线程数时，即使有线程空闲，线程池也会优先创建新线程处理
  设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭
- maximumPoolSize : 指定线程池中最大线程数
  当线程数>=corePoolSize，且任务队列已满时。线程池会创建新线程来处理任务
  当线程数=maxPoolSize，且任务队列已满时，线程池会拒绝处理任务而抛出异常
- keepAliveTime : 当线程池线程数量超过corePoolSize时，多余的空闲线程存活时间
  当线程空闲时间达到keepAliveTime时，线程会退出，直到线程数量=corePoolSize
  如果allowCoreThreadTimeout=true，则会直到线程数量=0
- unit : keepAliveTime 参数的单位 时、分、秒等
- workQueue : 任务队列
- threadFactory : 线程工厂用于创建线程池中的线程
- handler : 任务拒绝处理器
  两种情况会拒绝处理任务：
  - 当线程数已经达到maxPoolSize，切队列已满，会拒绝新任务
  - 当线程池被调用shutdown()后，会等待线程池里的任务执行完毕，再shutdown。如果在调用shutdown()和线程池真正shutdow

  线程池会调用rejectedExecutionHandler来处理这个任务。如果没有设置默认是AbortPolicy，会抛出异常
  ThreadPoolExecutor类有几个内部实现类来处理这类情况：
  - AbortPolicy 丢弃任务，抛运行时异常
  - CallerRunsPolicy 执行任务
  - DiscardPolicy 忽视，什么都不会发生
  - DiscardOldestPolicy 从队列中踢出最先进入队列（最后一个执行）的任务

  实现RejectedExecutionHandler接口，可自定义处理器

参数详解

- 当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。
- 当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行
- 当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务
- 当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理
- 当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程
- 当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭

举例说明参数

公司要设立一个项目组来处理某些任务，hr部门给的人员编制是10个人（corePoolSize）。同时给他们专门设置了一间有15个座位（maxim
来了一个任务，就招聘一个人。就这样，一个一个的招聘，招满了十个人，不断有新的任务安排给这个项目组，每个人也在不停的接任务干
法处理完了。其他的任务就只能在走廊外面(workQueue)排队了。后来任务越来越多，走廊的排队(workQueue)队伍也挤不下。然后只好找
公室只有15个座位，所以它们最多也就只能找5个临时工。可是任务依旧越来越多，根本处理不完，那没办法，这个项目组只好拒绝再接新
最后任务渐渐的少了，大家都比较清闲了。所以就决定看大家表现，谁表现不好，谁就被清理出这个办公室（空闲时间超过 keepAliveTime
（corePoolSize），维持固定的人员编制为止

自定义线程池

```
//自定义线程工厂
ExecutorService singleThreadExecutor = new ThreadPoolExecutor(10, 10, 0, TimeUnit.DAYS,
                new LinkedBlockingQueue<Runnable>(),new ThreadFactory() {

            @Override
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r);
                return t;
            }
        });

//使用默认线程工厂
ExecutorService singleThreadExecutor = new ThreadPoolExecutor(10, 10, 0, TimeUnit.DAYS, new LinkedB
```

扩展线程池

每个线程执行之前，每个线程结束之后，线程池终止调用的回调方法

```
ExecutorService es = new ThreadPoolExecutor(10, 10, 0, TimeUnit.DAYS, new LinkedBlockingQueue<Runnab

    @Override
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
    }

    @Override
    protected void afterExecute(Runnable r, Throwable t) {
         super.afterExecute(r, t);
     }

     @Override
     protected void terminated() {
         super.terminated();
     }

};
```

示例

```
public static void main(String[] args) {
             ExecutorService es = new ThreadPoolExecutor(10, 10, 0, TimeUnit.DAYS, new LinkedBlockin
             @Override
             protected void beforeExecute(Thread t, Runnable r) {
```

```
                System.out.println(((Task)r).getName() + " before");
            }

            @Override
            protected void afterExecute(Runnable r, Throwable t) {
                    System.out.println(((Task)r).getName() + " end");
             }

             @Override
             protected void terminated() {
                 System.out.println("线程池shutdown");
             }
        };

        for (int i = 0 ; i < 5  ; i++){
            Task t = new Task();
            t.setName("t : " + i + " ");
             es.execute(t);
        }

        es.shutdown();
    }


    static class Task implements Runnable{

            private String name;

            public String getName() {
                    return name;
            }

            public void setName(String name) {
                    this.name = name;
            }

            @Override
            public void run() {
                    System.out.println(this.name + " : run");
            }

    }
```

线程池-拒绝策略（RejectedExecutioHandler）

ThreadPoolExecutor的构造函数中指定了拒绝策略，及当任务数量超过了系统实际承载能力时该如何处理。 JDK内置了四种拒绝策略：

- AbortPolicy 该策略直接抛出异常，阻止系统工作
- CallerRunsPolicy 只要线程池未关闭，该策略直接在调用者线程中运行当前被丢弃的任务。显然这样不会真的丢弃任务，但是，调用
- DiscardOledestPolicy丢弃最老的一个请求任务，也就是丢弃一个即将被执行的任务，并尝试再次提交当前任务。
- DiscardPolicy 默默的丢弃无法处理的任务，不予任何处理。

```
public class TestRejectHandler {

    static class MyTask implements Runnable {
```

```
4               @Override
5               public void run() {
6                       System.out.println("Thread ID: " + Thread.currentThread().getId());
7                       try {
8                               Thread.sleep(1000);
9                       } catch (InterruptedException e) {
10                              e.printStackTrace();
11                      }
12              }
13      }
14
15      public static void main(String[] args) {
16              ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 5, 0L, TimeUnit.MILLISECONDS,
17                              new LinkedBlockingQueue<Runnable>(5), Executors.defaultThreadFactory(),
18                                      @Override
19                                      public void rejectedExecution(Runnable r, ThreadPoolExecutor ex
20                                              System.out.println(r.toString() + " 被抛弃了");
21                                      }
22                              });
23              MyTask task = new MyTask();
24              for (int i = 0; i < 20; i++) {
25                      executor.submit(task);
26              }
27              executor.shutdown();
28      }
29  }
```