# 1.线程池ThreadPoolExecutor.

参考：<https://www.cnblogs.com/dolphin0520/p/3932921.html>

https://blog.csdn.net/l\_kanglin/article/details/57411851

## 继承关系：

ThreadPoolExecutor继承自抽象类AbstractExecutorService.

AbstractExcutorService实现了ExecutorService接口

ExecutorService接口继承了Executor接口

## 构造函数：

**public** ThreadPoolExecutor(**int** corePoolSize,

**int** maximumPoolSize,

**long** keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {

**if** (corePoolSize < 0 ||

maximumPoolSize <= 0 ||

maximumPoolSize < corePoolSize ||

keepAliveTime < 0)

**throw** **new** IllegalArgumentException();

**if** (workQueue == **null** || threadFactory == **null** || handler == **null**)

**throw** **new** NullPointerException();

**this**.acc = System.*getSecurityManager*() == **null** ?

**null** :

AccessController.*getContext*();

**this**.corePoolSize = corePoolSize;

**this**.maximumPoolSize = maximumPoolSize;

**this**.workQueue = workQueue;

**this**.keepAliveTime = unit.toNanos(keepAliveTime);

**this**.threadFactory = threadFactory;

**this**.handler = handler;

}

共有四个构造函数，但是其它三个构造函数都是调用这个构造函数来完成对象的实例化的。

各参数的意义：

**corePoolSize：**表示线程池的大小，一旦创建，它就是这么大。

**maximumPoolSize:**表示线程池的最大的大小。当任务过多时，可能会创建新的线程来执行任务，但是总的线程池的大小不能超过这个maximumPoolSize，否则就会抛出异常，或者拒绝任务，这个根据拒绝策略来定。

**KeepAliveTime:** 当线程池中线程的数量大于corePoolSize的时候，如果某个线程空闲时间超过keepAliveTime，就会销毁这个线程。

**Unit：**keppAliveTime的时间单位。

**workQueue**:任务缓存队列，当线程数量大于等于corePoolSize的时候，新来的任务就会被缓存到workQueue中。

**ThreadFactory**:用来创建新的线程的线程工厂。

**RejectedExecutionHandler** :拒绝处理任务的策略。

## 线程池的五种状态

// runState is stored in the high-order bits

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

**private** **static** **final** **int** ***SHUTDOWN*** = 0 << ***COUNT\_BITS***;

**private** **static** **final** **int** ***STOP*** = 1 << ***COUNT\_BITS***;

**private** **static** **final** **int** ***TIDYING*** = 2 << ***COUNT\_BITS***;

**private** **static** **final** **int** ***TERMINATED*** = 3 << ***COUNT\_BITS***;

当线程池刚刚创建的时候处于***RUNNING***状态，当调用shutdown()方法的时候，线程池会处于***SHUTDOWN***状态，此时线程池不再接收新的任务，但是会继续把正在执行的任务和缓存队列中的任务都执行完了以后再销毁。当调用线程池的shutdownNow()方法的时候,线程池会进行***STOP***状态，此时线程池不再接收新的任务，并且会尝试去终止正在执行的任务和清空缓存对列。当线程池为空的时候，即线程池中没有执行的线程了的时候。就会进入***TIDYING*** 状态。此时线程池就会执行另一个钩子函数terminate(),执行完这个函数，线程池就会从***TIDYING*** 状态变为***TERMINATED***状态。

如下图：



## 线程池原理：

线程池创建之初，会设定**corePoolSize**和**maximumPoolSize**。并且不会创建线程，除非通过预创建的方法prestartAllCoreThreads()或者prestartCoreThread()。在没有预创建的情况下，就是来一个任务创建一个线程。当线程数到达**corePoolSize**的时候，就会把新来的任务放到缓存队列中。当缓存队列满了以后，就会创建新的线程。当线程数到达**maximumPoolSize**。再有新的任务过来，就会抛出异常，拒绝任务，并关闭线程池。但是已经在缓存队列中的线程和正在执行的线程会执行完。

当线程的数量大于**corePoolSize的时候，**如果某个线程的空闲时间超过**KeepAliveTime，**就会被销毁。

## 使用示例

**import** java.util.concurrent.ArrayBlockingQueue;

**import** java.util.concurrent.ThreadPoolExecutor;

**import** java.util.concurrent.TimeUnit;

**public** **class** TreadPoolTest {

**public** **static** **void** main(String[] args) {

ThreadPoolExecutor executor=**new** ThreadPoolExecutor(5, 10, 200, TimeUnit.***MILLISECONDS***, **new** ArrayBlockingQueue<Runnable>(5));

**for**(**int** i=0;i<15;i++) {

MyTask myTask=**new** MyTask(i);

executor.execute(myTask);

System.***out***.println("线程池数目"+executor.getPoolSize()+",队列中正在等待执行的任务数目："+executor.getQueue().size()+

",已执行完别的任务数目："+executor.getCompletedTaskCount());

}

executor.shutdown();

}

}

**class** MyTask **implements** Runnable{

**private** **int** taskNum;

**public** MyTask(**int** i) {

**this**.taskNum=i;

}

@Override

**public** **void** run() {

System.***out***.println("正在执行task"+taskNum);

**try** {

Thread.*currentThread*().*sleep*(4000);

} **catch** (Exception e) {

e.printStackTrace();

}

System.***out***.println("task"+taskNum+"执行完毕");

}

}

**输出结果：**

正在执行task0

线程池数目1,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目2,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task1

线程池数目3,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task2

线程池数目4,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task3

线程池数目5,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：1,已执行完别的任务数目：0

正在执行task4

线程池数目5,队列中正在等待执行的任务数目：2,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：3,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：4,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

线程池数目6,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task10

线程池数目7,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task11

线程池数目8,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task12

线程池数目9,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task13

线程池数目10,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task14

task0执行完毕

task1执行完毕

task2执行完毕

正在执行task6

正在执行task5

正在执行task7

task4执行完毕

task3执行完毕

正在执行task9

task10执行完毕

正在执行task8

task14执行完毕

task12执行完毕

task13执行完毕

task11执行完毕

从输出结果中可以看出，当缓冲队列满的时候，才会创建新的线程。如果把任务数改为20的时候，输出接口如下：

正在执行task0

线程池数目1,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目2,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目3,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task1

正在执行task2

线程池数目4,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

正在执行task3

线程池数目5,队列中正在等待执行的任务数目：0,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：1,已执行完别的任务数目：0

正在执行task4

线程池数目5,队列中正在等待执行的任务数目：2,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：3,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：4,已执行完别的任务数目：0

线程池数目5,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

线程池数目6,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task10

线程池数目7,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task11

线程池数目8,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task12

线程池数目9,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task13

线程池数目10,队列中正在等待执行的任务数目：5,已执行完别的任务数目：0

正在执行task14

Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task MyTask@7d4991ad rejected from java.util.concurrent.ThreadPoolExecutor@28d93b30[Running, pool size = 10, active threads = 10, queued tasks = 5, completed tasks = 0]

at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:2063)

at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:830)

at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1379)

at TreadPoolTest.main(TreadPoolTest.java:11)

task4执行完毕

task0执行完毕

task1执行完毕

正在执行task7

task2执行完毕

task11执行完毕

task3执行完毕

正在执行task9

task10执行完毕

正在执行task8

task12执行完毕

正在执行task6

正在执行task5

task13执行完毕

task14执行完毕

从上面可以看出，当缓冲队列满的时候，而且线程池中的线程已经为maximumPoolSize的时候，就会报出异常，但是报出异常之后并不会终止线程池中的线程，而是会把正在执行的线程和缓存队列中的线程都执行完了以后，再终止线程池。

# 2.Synchronized的使用方法

**public** **class** Test5 {

**class** Inner{

//synchronized规则：

//1.两个线程并发访问同一个object的同一同步代码块，当一个线程执行这个代码块时，另一个线程就会阻塞

//2.两个线程并发访问同一个对象，访问不同的同步代码块时，当一个线程执行一个同步代码块时，另一个线程执行另一个同步代码块的访问也会被阻塞。

//3.两个线程并发访问同一个对象，一个线程访问同步代码块时，另一个线程可以访问非同步代码块，不会被阻塞。

//synchronized的两个用法：1.修饰方法。2.synchronized(object) ,加对象锁

**private** **void** mt1() {

**int** i=5;

**while**(i-->0) {

System.***out***.println(Thread.*currentThread*().getName()+":Inner.mt1()="+i);

**try** {

Thread.*sleep*(500);

} **catch** (Exception e) {

// **TODO**: handle exception

}

}

}

**private** **void** mt2() {

**int** i=5;

**while**(i-->0) {

System.***out***.println(Thread.*currentThread*().getName()+":Inner.mt2()="+i);

**try** {

Thread.*sleep*(500);

} **catch** (Exception e) {

// **TODO**: handle exception

}

}

}

}

**private** **void** mt1(Inner inner) {

**synchronized** (inner){//使用对象锁

inner.mt1();

}

}

**private** **void** mt2(Inner inner) {

// synchronized (inner) {

inner.mt2();

// }

}

**public** **static** **void** main(String[] args) {

**final** Test5 myt3=**new** Test5();

**final** Inner inner=myt3.**new** Inner();

Thread t1=**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

myt3.mt1(inner);

}

},"t1");

Thread t2=**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

myt3.mt2(inner);

}

},"t2");

t1.start();

t2.start();

}

}

# 3.线程和原子性

## 进程和线程

进程是资源调度和分配的基本单位，线程是CPU分配的基本单位。多个线程共享进程的资源（堆栈，上下文，寄存器）。进程结束后它拥有的所有线程都将销毁，而线程的结束不会影响同个进程中的其他线程的结束

## 原子性

原子性，所谓原子性就是不可分割性，是指某组操作是一个不可分割的部分。

Java提供了一种内置锁来支持原子性，叫做同步代码块（Synchronized block）.

若父类中的某个方法，和子类重写的该方法都是Synchronized方法，那么调用这个两个方法时，都会加上父类的锁，所以如果内置锁不可重入，则可能会发生死锁。

## 可见性

其实就是执行读操作的线程能够看到其他线程写入的值。即多个线程对内存的写入都是可见性。可见性需要使用同步机制来实现。

## 有序性

就是让程序按照代码的顺序串行执行，不进行重新排序。

## volatail变量

Volatail变量是一种削弱的同步机制，用来确保变量的更新操作通知到其他线程，也就是说编译器会把volatail变量标记为共享变量，不会让对volatail变量的操作和其他内存操作一起重排序。Volatail变量总是可见的，不会被缓存到如寄存器之类的处理器不可见的地方，所以每次读都是读取最新的值。

Volatial变量的优势：

（1）volatail变量的访问不会加锁，所以它不会造成线程阻塞，这就是一种比Syncronized更加轻量级的同步机制。

(2) volatail变量提供了变量状态的可见性。

Volatail变量在什么情况下使用？

仅当volatail变量能够简化代码的实现以及对同步策略的验证时，使用它：

1. 确保他们自身状态的可见性。
2. 确保他们所引用的对象的状态的可见性。
3. 标识一些重要的程序生命周期事件的发生。

（注：当在验证正确性的时候需要进行复杂的可见性判断时，就不要使用volatail变量）

使用volatail变量的条件：

1. 对变量的写入操作不依赖变量的当前值（反例：i++），或者你能确保只有单线程来访问这个变量。
2. 该变量不会和其它变量一起纳入不变性条件种
3. 在访问变量时不需要枷锁

典型用法：判断时否推出循环的某个标记变量。

# 4.jdk1.8中的锁

截止JDK1.8，java中的锁包括：参考 <http://ifeve.com/java_lock_see/>