# JUC

`java.util.concurrent`

`java.util.concurrent.atomic`

`java.util.concurrent.locks`

# 1.进程线程回顾

- **进程**是程序的一次执行过程，是一个动态概念，是程序在执行过程中分配和管理资源的基本单位，每一个进程都有一个自己的地址空间，至少有 5 种基本状态，它们是：初始态，执行态，等待状态，就绪状态，终止状态。
- **线程**是CPU调度和分派的基本单位，它可与同属一个进程的其他的线程**共享**进程所拥有的全部资源。

**根本区别：**进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位

## 认识java的线程

### 线程的6个状态

1. `NEW`：新建的线程，还未执行
2. `RUNNABLE`：运行中的线程，正在执行`run()`方法
3. `BLOCKED`：线程阻塞
4. `WAITING`：调用`wait()`方法，死等
5. `TIMED_WAITING`：调用`sleep()`方法，计时等待
6. `TERMINATED`：线程终止

### `sleep`和`wait`的区别

1. wait是Object中的方法，而sleep则是Thread中的方法。
2. sleep可以在任何地方使用，而wait只可以在synchronized方法或synchronized块中使用。
3. sleep方法只会让出当前线程持有的时间片而不释放锁，而wait方法除了让出时间片还会让出当前线程持有的锁。
4. sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常
5. sleep是Thread类的静态方法。sleep的作用是让线程休眠指定时间，在时间到达时恢复。wait是Object的方法，任意一个对象可以调用wait方法，调用wait方法将会将调用者的线程挂起，直到其他线程调用同一个对象的notify方法才会重新激活调用者。

**相同点：哪儿睡的哪儿醒**

### 并发和并行

- 并发：强调同一时间段
- 并行：强调同时

## `synchronized`原理

sychronized 用于修饰：

- 实例方法：以当前对象为锁
- 静态方法：以当前类的类对象为锁
- 同步代码块：自定义锁（但也是个对象，不能是基本类型）

## Java对象的布局

1. 实例数据 --- 不固定，按4字节对齐
2. Java对象头 --- 固定
3. 对齐填充

> HotSpot Vm的自动内存管理系统要求对象起始地址必须为8字节的整数倍，也就是说对象的大小必须为8字节的整数倍，因为对象头部分正好是8字节的倍数，因此，当对象示例数据部分没有对齐时，就需要通过对奇填充补全对象的size，使其大小满足8字节的倍数



## java对象头

无论 `synchronized` 用于修饰哪些同步代码，它所使用的就是**对象的头**

> java对象头是synchronized锁的基础，同步锁使用的锁存储在java对象头中
>
> jvm中采用2个字存储对象头（数组对象是3个，多出来的那个保存数组长度）
>
> **详述：**
>
> > 每个对象作为synchronized的锁时，对象头中都有一个指向该对象持有的monitor对象的指针

对象头的组成：

- Mark Word：存储对象的HashCode、锁信息||分代年龄||GC标志等、
  默认情况下存储

> 由于对象头信息是与对象自身定义的数据没有关系的额外存储成本，因此考虑到JVM的空间效率，Mark Word被设计为一个非固定的数据结构，一边存储更多有效的数据，它会根据对象的状态复用自己的存储空间，在32位JVM下，其可能的变化结构如下：

| 锁状态 | 25 bit | | 4bit | 1bit | 2bit |
|---|---|---|---|---|---|
| | 23bit | 2bit | | 是否是偏向锁 | 锁标志位 |
| 轻量级锁 | 指向栈中锁记录的指针 | | | | 00 |
| 重量级锁 | 指向互斥量（重量级锁）的指针 | | | | 10 |
| GC标记 | 空 | | | | 11 |
| 偏向锁 | 线程ID | Epoch | 对象分代年龄 | 1 | 01 |



64位JVM下（未启用指针压缩），其结构为：


synchronized是重量级锁

- Class Metadata Address：类型指针指向对象的元数据，JVM通过此指针确定该对象是那个类的实例

> 该字又称为klass pointer，JVM在堆内存小于32GB开启指针压缩，也就是说java对象头在未开启指针压缩时大小为**16字节**，开启指针压缩后为**12字节**

**所以synchronized以某个对象为锁就是改变了这个对象的对象头。**

# 2.LOCK接口

## 多线程编程模板上

1. 线程操作资源类

> 多线程就是多个线程抢占资源

2. 高内聚低耦合

> 对资源的操作方法要集合在资源类中，也就是资源类提供什么方法，线程就用什么方法。线程和资源类之间是低耦合的

## Lock是什么

使用锁比使用`synchronized`更加广泛和灵活

`Lock`是`java.util.concurrent.locks`包里面的一个接口，它的实现类有：

- `ReentrantLock`：可重用锁
- `ReentrantReadWriteLock.ReadLock`：可重用读写锁，读锁

- `ReentrantReadWriteLock.WriteLock`：可重用读写锁，写锁

# 3.线程间通信

## 多线程编程模板中

1. 判断
2. 干活
3. 唤醒

**面试题**：两个线程，一个打印1~52，另一个打印12A34B...4950Y5152Z, 使用线程间通信

```
//用synchronized关键字+wait+notify实现，生产者消费者模型，不符合面试题题意
public class PrintNumLetter {
    public static void main(String[] args) {
        Resource resource = new Resource();

        new Thread(() -> {
            try {
                for (int i = 0; i < 26; i++) {

                    resource.printNumLetter();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "打印数字字母线程").start();
        new Thread(() -> {
            try {
                for (int i = 0; i < 26; i++) {
                    resource.printNum();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "打印数字线程").start();
    }
}

//定义资源
class Resource {
    private int num = 1;
    private int count = 0;

    public synchronized void printNum() throws InterruptedException {
        //判断
        if (num % 2 == 0 && num > 0) {
            this.wait();
        }
        //干活
        System.out.printf(Thread.currentThread().getName() + " ");
        for (int i = num; i < num + 2; i++) {
            System.out.printf("%d", i);
        }
        System.out.println();
```

```
            num += 1;
            //唤醒
            this.notifyAll();
        }

        public synchronized void printNumLetter() throws InterruptedException {
            //判断,这里应该把if改为while，因为存在虚假唤醒
            if (num % 2 != 0 && num > 0) {
                this.wait();
            }
            //干活
            System.out.printf(Thread.currentThread().getName() + " ");
            for (int i = num - 1; i < num + 1; i++) {
                System.out.printf("%d", i);
            }
            System.out.printf("%c\n", (65 + count));
            count++;
            num++;
            //唤醒
            this.notifyAll();
        }
    }

    //使用lock+await+signal，这个是生产者消费者模型，不符合面试题要求
    public class PrintNumLetter {
        public static void main(String[] args) {
            Resource resource = new Resource();

            new Thread(() -> {
                try {
                    for (int i = 0; i < 26; i++) {
                        resource.printNumLetter();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }, "打印数字字母线程").start();
            new Thread(() -> {
                try {
                    for (int i = 0; i < 26; i++) {

                        resource.printNum();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }, "打印数字线程").start();
        }
    }

    //定义资源
    class Resource {
        private int num = 1;
        private int count = 0;

        private Lock lock = new ReentrantLock();
        private Condition cd = lock.newCondition();
```

```
    public void printNum() throws InterruptedException {
        lock.lock();
        try {
            //判断
            while (num % 2 == 0 && num > 0) {
//                this.wait();
                cd.await();
            }
            //干活
            System.out.printf(Thread.currentThread().getName() + " ");
            for (int i = num; i < num + 2; i++) {
                System.out.printf("%d", i);
            }
            System.out.println();
            num += 1;
            //唤醒
//            this.notifyAll();
            cd.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void printNumLetter() throws InterruptedException {
        lock.lock();
        try {
            //判断，while避免虚假唤醒
            while (num % 2 != 0 && num > 0) {
//                this.wait();
                cd.await();
            }
            //干活
            System.out.printf(Thread.currentThread().getName() + " ");
            for (int i = num - 1; i < num + 1; i++) {
                System.out.printf("%d", i);
            }
            System.out.printf("%c\n", (65 + count));
            count++;
            num++;
            //唤醒
//            this.notifyAll();
            cd.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

**上面两种实现的对比**

## 多线程编程模板下

注意多线程之间的虚假唤醒

## 为什么存在虚假唤醒呢

见上述面试题中synchronized的实现

# 4.线程间定制化调用通信

即需要一个标志位，用来决定哪个线程可以执行，

需要多个钥匙给多个线程使用，可以用cd.signal()唤醒持有这个钥匙的线程

上面那个面试题的另一个实现

```
//lock+await+signal的另一个实现，可以做到一个线程执行完毕后另一个再执行，可以决定哪个线程最先执行
public class PrintNumLetter {
    public static void main(String[] args) {
        Resource resource = new Resource();

        new Thread(() -> {
            try {
                resource.printNumLetter();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "打印数字字母线程").start();
        new Thread(() -> {
            try {
                resource.printNum();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "打印数字线程").start();
    }
}

//定义资源
class Resource {
    private int flag=1;//标志位，决定了哪个线程执行

    //创建锁和钥匙
    private Lock lock = new ReentrantLock();
    private Condition cd1 = lock.newCondition();
    private Condition cd2 = lock.newCondition();

    public void printNum() throws InterruptedException {
        lock.lock();
        try {
            //判断
            while (flag!=1){
                cd1.await();
            }
            //干活
            System.out.printf(Thread.currentThread().getName() + " ");
            for (int i = 0; i <52 ; i++) {
                System.out.println(i+1);
```

```
            }
            flag++;
            cd2.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void printNumLetter() throws InterruptedException {
        lock.lock();

        try {
            //判断
            while (flag!=2){
                cd2.await();
            }
            //干活
            int ini = 1;
            System.out.printf(Thread.currentThread().getName() + " ");
            for (int i = 0; i < 26; i++) {
                for (int j = ini; j <ini+2 ; j++) {
                    System.out.printf("%d", j);
                }
                System.out.printf("%c\n", (65 + i));
                ini+=2;
            }
            //唤醒
            flag--;
            cd1.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

# 5.线程不安全的集合类

对线程不安全的集合类进行并发修改时，在线程较多的情况下会抛出`java.util.CocurrentModificationException`的异常

**那么该怎么把线程不安全的集合类变为线程安全的呢?**

1. `Collections`工具类中有各个集合类的线程安全的实现类，但是是使用`synchronized`实现，效率低
2. `ConcurrentHashMap`是`HashMap`的线程安全的实现
3. 引入一个概念：**写时拷贝**技术

> 就是等到修改数据时才真正分配内存空间，在此之前的拷贝只是增加一个指向资源的指针，这是对程序性能的优化，可以延迟甚至是避免内存拷贝，当然目的就是避免不必要的内存拷贝。

在juc中提供一两个类`CopyOnWriteArrayList`和`CopyOnWriteArraySet`，这两个类用到了写时拷贝技术，效率更高，其底层原理是发生`set`、`add`等修改操作时在原对象上生成一个副本，所有的修改操作都是在副本上完成的，原来的对象可以被**共享读**，但是副本是**独占写**，写操作完成后，原对象的指针指向副本，原对象作为垃圾被回收

# 6.Callable接口

```
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}

@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

通过上面的对比可以看到`Runnable`接口和`Callable`接口的异同：

- 区别：
    1. 前者无返回值，后者有返回值
    2. 前者无泛型，后者有泛型
    3. 前者的`run`方法无法向外抛异常，后者的`call`方法可以向外抛异常
- 相同：都是函数式接口

## `FutureTask`类

`FutureTask`是`Runnable`接口的一个实现类，它的构造方法可以接受`Callable`类型的对象，因此在构造线程的方法如下：

```
FutureTask<V> ft = new FutureTask<V>(()->{
  //todo,需要一个返回值
});

//运行call方法中的代码
new Thread(ft).start();

//获取结果
ft.get();
```

## 作用

从`FutureTask`这个类的名字中可以看出，这是一个“未来的任务”，也就是**回调**

**主要用在需要长时间计算的任务中，即为长时间计算的任务开辟一个线程，让这个任务在后台完成，达到不阻塞主线程的目的，当任务完成后通过`get`方法调用计算的结果，可以通过`ft.isDone()`方法判断任务是否完成，如果任务还未完成就调用`get`方法，将产生阻塞，同理，如果在主线程中执行该代码，也会阻塞，直至代码运行完成**

同一个`FutureTask`对象只会执行一次

> 这是因为一般交给`FutureTask`执行的任务都比较复杂，多次执行比较耗费计算资源

# 7.JUC工具类

## CountDownLatch

```
CountDownLatch cdl = new CountDownLatch(int steps);
cdl.await();
cdl.countDown();
```

它是一个允许一个或多个线程去等待直到其他一个或多个线程完成其操作后的同步工具

也就是说某些任务需要前面若干个任务全部完成后才能进行

它主要有两个方法：

- `await()`：调用该方法的线程将被阻塞，知道计数器为零被唤醒
- `countDown()`：每调用一次计数器减少1，直至为零，这是唤醒调用了`await`方法的线程，调用此方法的线程不会阻塞

## CyclicBarrier 循环栅栏

```
//第一个参数是到达屏障的次数，第二个是到达屏障的动作
CyclicBarrier cb = new CyclicBarrier(int, Runnabel);
//调用该方法阻塞，
cb.await();
```

它允许一组线程互相等待，直到到达某个公共屏障点。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。比如将计数器设置为10，那么凑齐第一批10个线程后，计数器就会归0，然后接着凑齐下一批10个线程，这就是循环栅栏内在的含义。

主要的方法：

- `await()`：调用该方法的线程将被阻塞直至最后一个线程到达屏障
- `getParties()`：获取通过该屏障的参与线程个数
- `getNumberWaiting()`：获取在屏障处等待的线程个数
- `reset()`：重置该屏障

### CyclicBarrier和CountDownLatch的区别

1. CountDownLatch的作用是允许1或N个线程等待其他线程完成执行；而CyclicBarrier则是允许N个线程相互等待。
2. CountDownLatch的计数器无法被重置；CyclicBarrier的计数器可以被重置后使用。

## Semaphore

```
Semaphore sp = new Semaphore(int, boolean);
sp.acquire();
sp.release();
```

信号灯的作用主要有两个:

- 用于多个资源的互斥使用
- 并发线程数的控制

主要方法：

- `acquire()`: 当一个线程调用该方法时，要么成功，获得信号量（信号量减1），要么等下去直到有线程释放信号量，或超时
- `release()`: 将信号量加1，并唤醒等待的线程

# 8.阻塞队列与线程池

## 阻塞队列初步

`BlockingQueue`是一个接口，它有三个常用的实现类：

1. `ArrayBlockingQueue`： 创建对象需要赋予队列的大小
2. `LinkedBlockingQueue`： 默认大小为Integer.MAX_VALUE
3. `SynchronousQueue`： 单个元素的队列

以下是阻塞队列常用的方法及其特性：

| 方法类型 | 抛出异常 | 特殊值 | 阻塞 | 超时 |
| --- | --- | --- | --- | --- |
| 插入 | add(e) | offer(e) | put(e) | offer(e,time,unit) |
| 移除 | remove() | poll() | take() | poll(time,unit) |
| 检查 | element() | peek() | 不可用 | 不可用 |

| | |
| --- | --- |
| 抛出异常 | 当阻塞队列满时，再往队列里add插入元素会抛IllegalStateException:Queue full<br>当阻塞队列空时，再往队列里remove移除元素会抛NoSuchElementException |
| 特殊值 | 插入方法，成功ture失败false<br>移除方法，成功返回出队列的元素，队列里没有就返回null |
| 一直阻塞 | 当阻塞队列满时，生产者线程继续往队列里put元素，队列会一直阻塞生产者线程直到put数据or响应中断退出<br>当阻塞队列空时，消费者线程试图从队列里take元素，队列会一直阻塞消费者线程直到队列可用 |

# 线程池

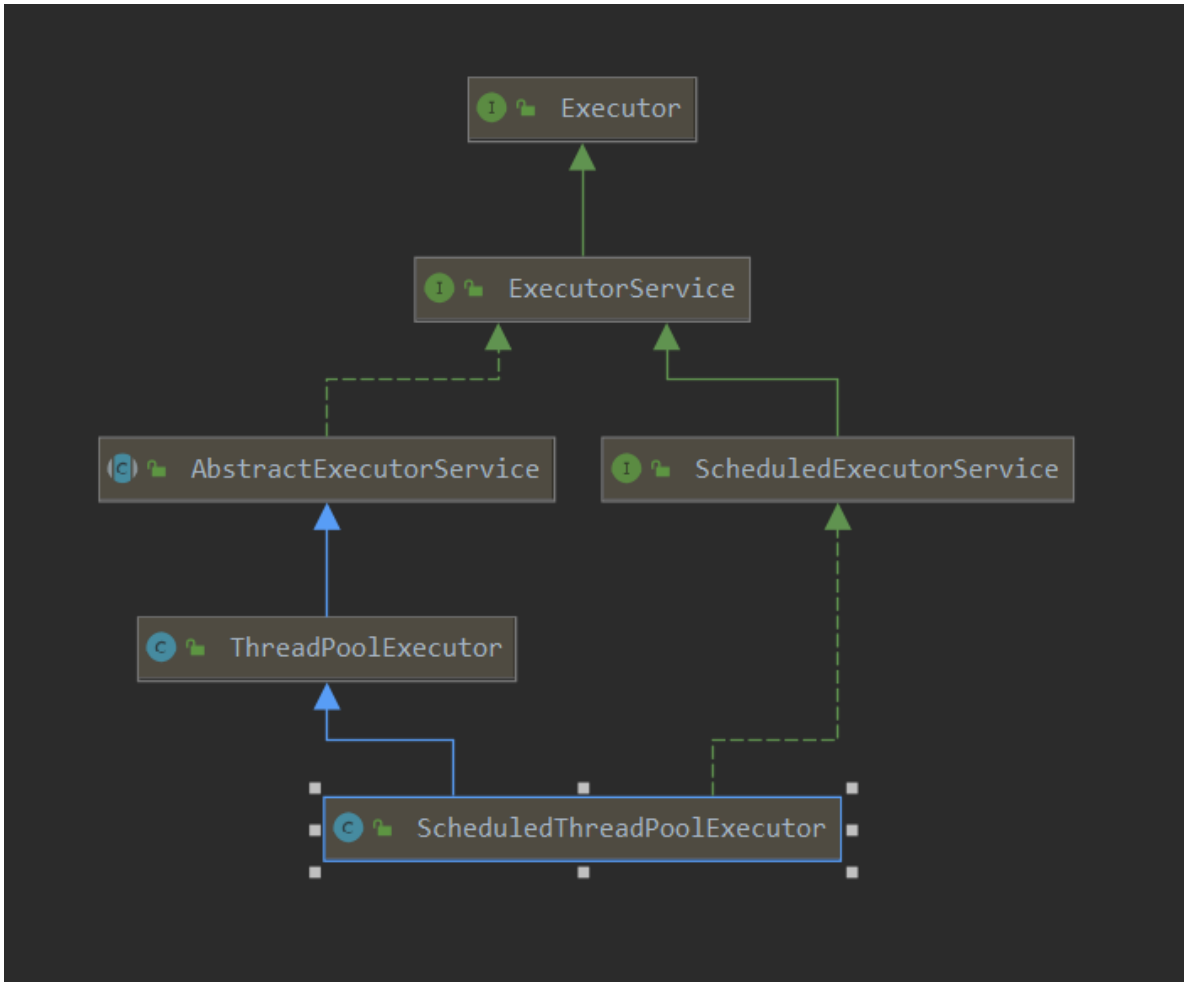
## 为啥用线程池

线程池的作用主要是控制线程的运行数量，处理过程中将任务放入队列，然后在线程中启动这些任务，如果运行的线程数量超过了最大值，超出的线程等待，等其他线程执行完毕，再从队列中取出线程运行。

- 线程复用：降低线程创建和销毁造成的损耗。
- 提高响应速度
- 控制最大并发数，可以起到削峰的作用
- 管理线程：线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源而且会增加系统的不稳定性，使用线程池可以实现统一分配、调优、监控。

## 咋用线程池

### 架构




- Executor（接口）
- ExecutorService（接口）
- **ThreadPoolExecutor**（类）
- ScheduledThreadPoolExecutor（类）
- Executors（工具类）

**创建线程池**

以下方法不可用

- ```
  Executors.newFixedThreadPool(int)//执行长期任务性能好，有固定的大小
  ```

- ```
  Executors.newSingleThreadExecutor()//一个任务一个任务地执行，一池一线程
  ```

- ```
  Executors.newCachedThreadPool()//执行很多短期异步任务，线程池根据需要创建线程，但是先前构建的线程在可用时使用它们。特点时可扩容，遇强则强
  ```

  **创建线程池应该使用**

  ```
  ExcutorService pool = new ThreadPoolExecutor(参数...)
  ```

  **使用`Executors`返回的线程池对象的弊端如下：**

  1. **FixedThreadPool和SingleThreadPool**

     **允许的阻塞队列长度为Integer.MAX_VALUE，可能会堆积大量请求，导致OOM**
  ```

2. **CachedThreadPool和ScheduledThreadPool**

   **允许的线程数量为Integer.MAX_VALUE，可能会创建大量线程，导致OOM**

**正确创建线程池**

```
ExecutorService pool = new ThreadPoolExecutor(
                3,
                5,
                3L,
                TimeUnit.SECONDS,
                new ArrayBlockingQueue<Runnable>(3),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy()
        );
```

**任务提交的两种方式**

1. `execute`只能提交`Runnable`类型的任务，无返回值；

   `submit`既可以提交`Runnable`类型的任务，也可以提交`Callable`类型的任务，并且返回一个`Future`类型的返回值，如果提交的是`Runnable`类型的任务，返回值是null;

- `excute`执行任务时，遇到异常直接抛出；

  `submit`不会直接抛出，只有使用Future的`get`方法时才会抛出异常

## 线程池底层原理

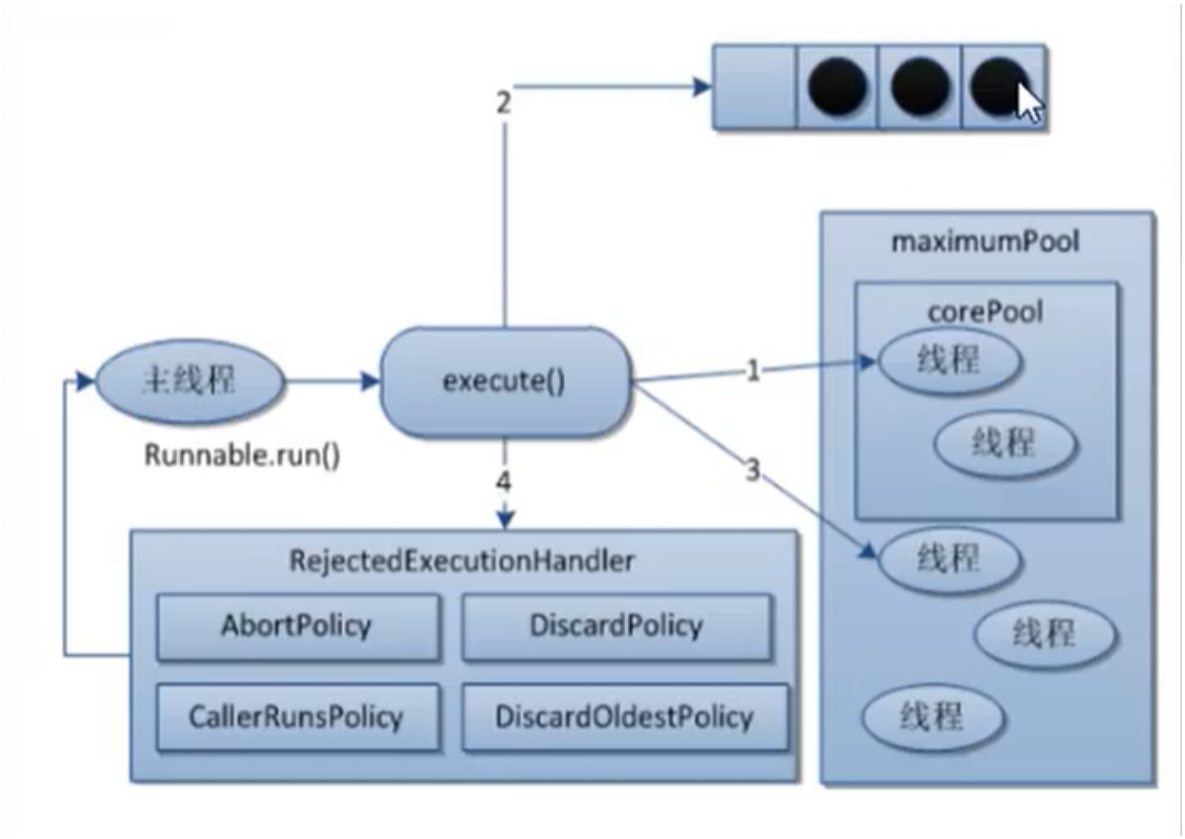
**`ThreadPoolExecutor`的参数**

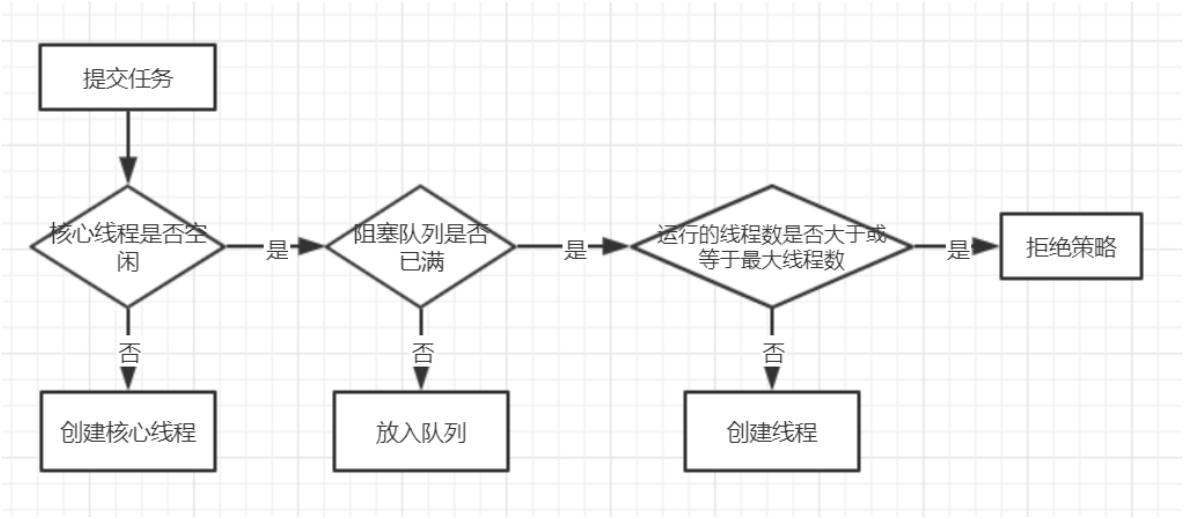
1. `int corePoolSize`:线程池中常驻核心线程数

   > 核心线程数是惰性加载的。在创建线程池的时候不会创建，只有在提交任务的时候才会判断是否创建

2. `int maximumPoolSize`：线程池中能容纳同时最大的执行线程数
3. `long keepAliveTime`：空闲线程（超过corePoolSize的部分）的存活时间
4. `TimeUnit unit`：存活时间单位
5. `BlockingQueue<Runnabel> workQueue`：任务队列，被提交但还未被执行
6. `ThreadFactory threadFactory`：生成线程池中线程的工厂，用于创建线程
7. `RejectedExecutionHandler handler`：拒绝策略，表示队列满了，并且工作线程大于等于线程池的最大线程数，此时如何拒绝请求执行的runnable的策略

**线程池底层逻辑图**




**JDK内置的拒绝策略**

- AbortPolicy(默认)：直接抛出`RejectedExecuptionException`阻止系统正常运行
- CallerRunsPolicy:“调用者运行一种调节机制”，该策略不会抛弃任何，也不会抛异常，而是将任务退回到调用者，从而降低新任务的流量
- DisgardOldestPolicy：抛弃等待队列中等待最久的任务，然会把当前任务加入队列，尝试再次提交当前任务
- DisgardPolicy：该策略默默丢弃无法处理的任务，不予处理也不抛异常。在允许任务丢失的情况下，这是最好的策略

## 线程池总结

1. 线程池什么时候创建线程?

   > 任务提交的时候

2. 核心线程占用后的其他任务是先放在阻塞队列中还是创建coreSize和maxSize之间的线程?

> 先放在阻塞队列中

3. 阻塞对列中的任务什么时候取出?

> worker中的runWorker()一个任务完成后取出下一个任务

4. 什么时候出发拒绝策略?

> 阻塞队列满，且正在运行的线程为maxSize，如果此时还有新任务，则出发阻塞队列

5. coreSize和maxSize之间的线程什么时候会die?

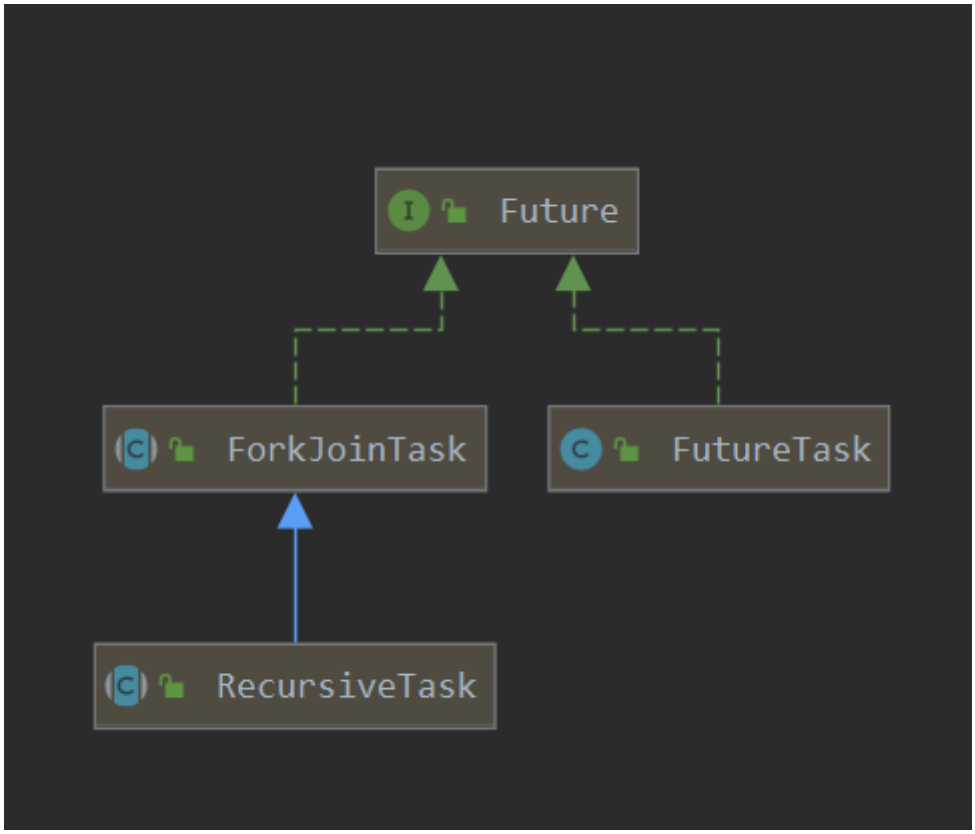
> 空闲时间超时或者抛出异常

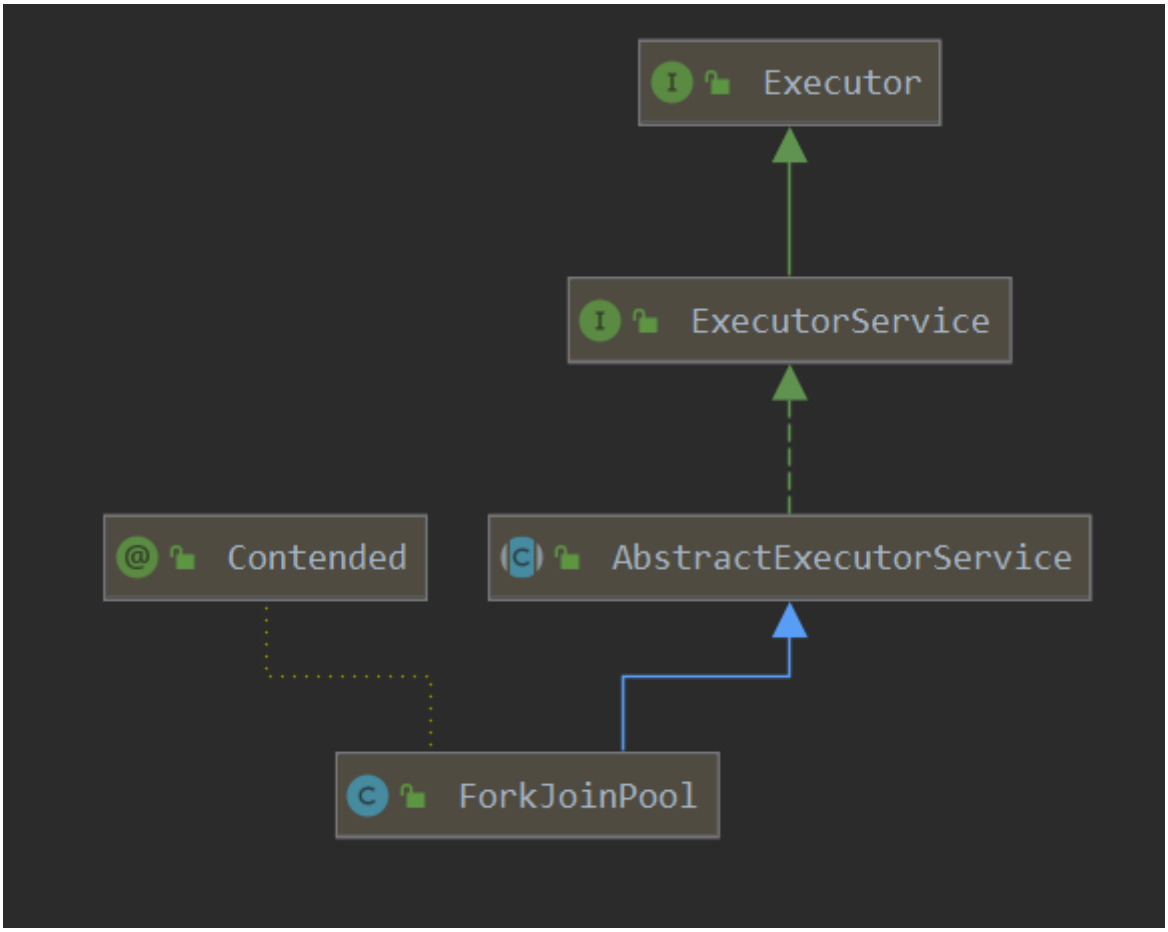
6. task抛出异常，线程池中的这个线程还能运行其他任务么?

> 不能。但是会创建其他线程来运行。

# 9.分支合并框架

## 架构






```
package com.heresun;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.RecursiveTask;

public class ForkJoinDemo {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        MyTask myTask = new MyTask(1, 1000000000);
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        ForkJoinTask<Integer> forkJoinTask = forkJoinPool.submit(myTask);
        Integer o = forkJoinTask.get();
        System.out.println(o);


    }
}

class MyTask extends RecursiveTask<Integer> {

    public static final int CONDITION = 10;
    private int begin;
    private int end;

    public MyTask(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }
```

```
    private int res;
    @Override
    protected Integer compute() {
        if ((end-begin)<=10){
            for (int i = begin; i <= end; i++) {
                res += i;
            }
        }else{
            int mid = begin + (end-begin)/2;
            MyTask myTask = new MyTask(begin, mid);
            MyTask myTask1 = new MyTask(mid + 1, end);

            myTask.fork();
            myTask1.fork();

            res = myTask.join() + myTask1.join();
        }


        return res;
    }
}
```

## 10.异步回调

没啥说的