10. 多线程

CONTENTS

本章技能点列表

技能点名称	难易程度	认知程度	重要程度
进程和线程	易	理解	**
线程的创建和启动	中	应用	***
线程生命周期	易	理解	***
线程控制	易	理解	**
线程同步	中	应用	**
同步代码块	中	应用	**
同步方法	中	应用	**
ReentrantLock锁	中	应用	**
线程通信	难	应用	**
线程组	易	了解	*
线程池	中	理解	**

1. 程序、进程与线程

- 程序Program
  - 程序是一段静态的代码，它是应用程序执行的蓝本
- 进程Process
  - 进程是指一种正在运行的程序，有自己的地址空间
- 进程的特点
  - 动态性
  - 并发性
  - 独立性
  - 并发和并行的区别
    - 多个CPU同时执行多个任务
    - 一个CPU（采用时间片）同时执行多个任务

1. 程序、进程与线程

- 线程Thread
  - 进程内部的一个执行单元，它是程序中一个单一的顺序控制流程。
  - 线程又被称为轻量级进程(lightweight process)
  - 如果在一个进程中同时运行了多个线程，用来完成不同的工作，则称之为多线程

- 线程特点
  - 轻量级进程
  - 独立调度的基本单位
  - 可并发执行
  - 共享进程资源

1. 程序、进程与线程

- 线程和进程的区别

区别	进程	线程
根本区别	作为资源分配的单位	调度和执行的单位
开　　销	每个进程都有独立的代码和数据空间(进程上下文)，进程间的切换会有较大的开销。	线程可以看成时轻量级的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换的开销小。
所处环境	在操作系统中能同时运行多个任务(程序)	在同一应用程序中有多个顺序流同时执行
分配内存	系统在运行的时候会为每个进程分配不同的内存区域	除了CPU之外，不会为线程分配内存（线程所使用的资源是它所属的进程的资源），线程组只能共享资源
包含关系	没有线程的进程是可以被看作单线程的，如果一个进程内拥有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的。	线程是进程的一部分，所以线程有的时候被称为是轻权进程或者轻量级进程。

1. 程序、进程与线程

- 班级：204
- 小组：1,2,3,4,5.....
- 完成一件事情：大扫除
  - 总负责：校长
  - 步骤1：以班级为单位领取大扫除工具，本班级的所有小组都使用该班级领取的资源
  - 步骤2：以小组为单位开始大扫除
  - 步骤3：校长亲自监督，如果发现不合格，直接要求该小组重新打扫；如果某小组打扫完毕，校长可以直接给该小组安排其他任务
- 对比
  - CPU：校长
  - 进程：班级 （一个班级可以有多个小组，班级是资源分配的单位）
  - 线程：小组 （校长直接指挥小组进行工作）

2. 线程的创建和启动

- 线程的创建
  - 方式1：继承Java.lang.Thread类，并覆盖run() 方法
  - 方式2：实现Java.lang.Runnable接口，并实现run() 方法
  - 方法run( )称为线程体。
- 线程的启动
  - 新建的线程不会自动开始运行，必须通过start( )方法启动
  - 不能直接调用run()来启动线程，这样run()将作为一个普通方法立即执行，执行完毕前其他线程无法兵法执行
  - Java程序启动时，会立刻创建主线程，main就是在这个线程上运行。当不再产生新线程时，程序是单线程的

2. 线程的创建和启动

- 两种线程创建方式的比较
  - 继承Thread类方式的多线程
    - 优势：编写简单
    - 劣势：无法继承其它父类
  - 实现Runnable接口方式的多线程
    - 优势：可以继承其它类，多线程可共享同一个Runnable对象
    - 劣势：编程方式稍微复杂，如果需要访问当前线程，需要调用Thread.currentThread()方法
  - 实现Runnable接口方式要通用一些。

2. 线程的创建和启动

- Thread类常用方法

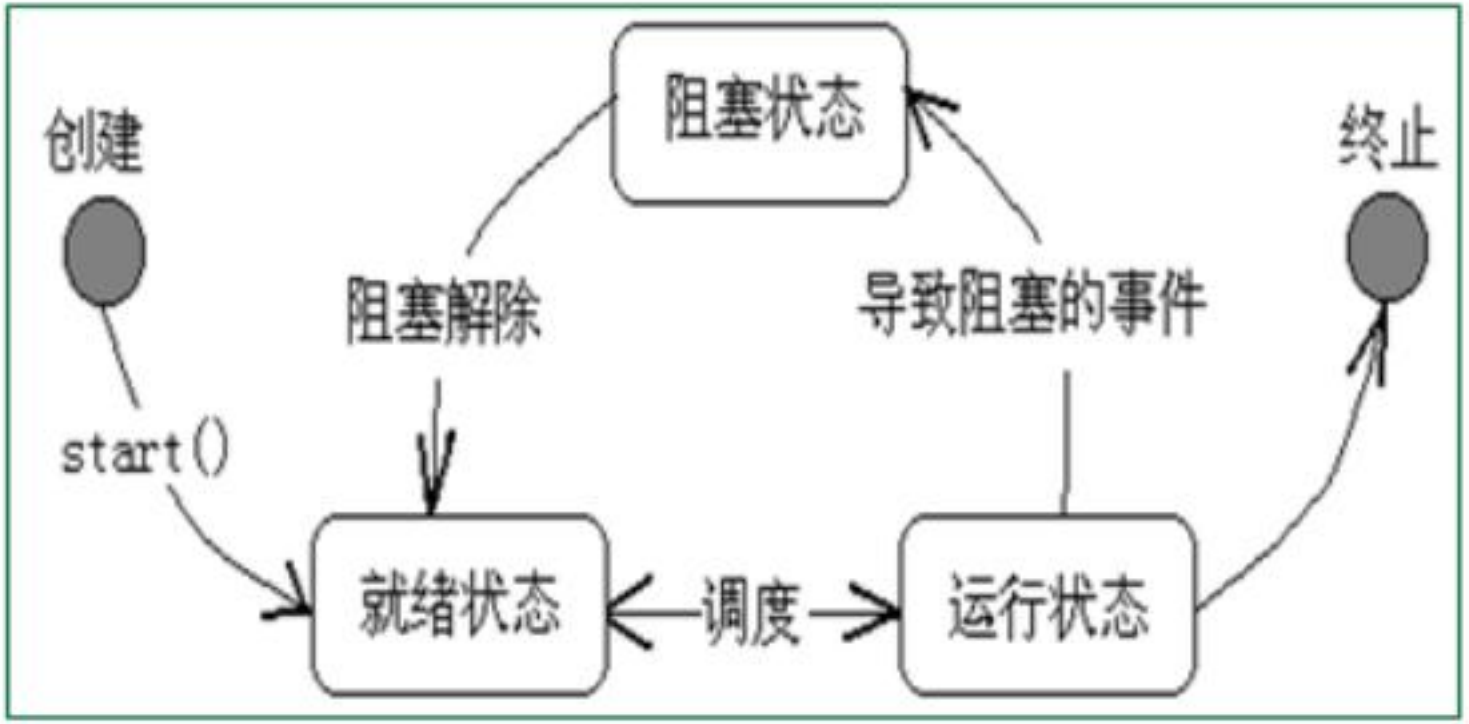
方 法	功 能
static Thread currentThread()	得到当前线程
getName( )	返回线程的名称
setName (String name)	将线程的名称设置为由name指定的名称
int getPriority()	获得线程的优先级数值
void setPriority()	设置线程的优先级数值
void start( )	调用run( )方法启动线程，开始线程的执行
void run( )	存放线程体代码
isAlive()	判断线程是否还“活”着，即线程是未终止

2. 线程的创建和启动

- 第三种方式：实现Callable接口
  - 与实行Runnable相比， Callable功能更强大些
  - 方法不同
  - 可以有返回值，支持泛型的返回值
  - 可以抛出异常
  - 需要借助FutureTask，比如获取返回结果

- Future接口
  - 可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。
  - FutrueTask是Futrue接口的唯一的实现类
  - FutureTask 同时实现了Runnable, Future接口。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

3. 线程的生命周期



3. 线程的生命周期

- 新生状态：
  - 用new关键字建立一个线程对象后，该线程对象就处于新生状态。
  - 处于新生状态的线程有自己的内存空间，通过调用start进入就绪状态
- 就绪状态：
  - 处于就绪状态线程具备了运行条件，但还没分配到CPU，处于线程就绪队列，等待系统为其分配CPU
  - 当系统选定一个等待执行的线程后，它就会从就绪状态进入执行状态，该动作称之为“cpu调度”。
- 运行状态：
  - 在运行状态的线程执行自己的run方法中代码，直到等待某资源而阻塞或完成任务而死亡。
  - 如果在给定的时间片内没有执行结束，就会被系统给换下来回到等待执行状态。
- 阻塞状态：
  - 处于运行状态的线程在某些情况下，如执行了sleep（睡眠）方法，或等待I/O设备等资源，将让出CPU并暂时停止自己的运行，进入阻塞状态。
  - 在阻塞状态的线程不能进入就绪队列。只有当引起阻塞的原因消除时，如睡眠时间已到，或等待的I/O设备空闲下来，线程便转入就绪状态，重新到就绪队列中排队等待，被系统选中后从原来停止的位置开始继续运行。
- 死亡状态：
  - 死亡状态是线程生命周期中的最后一个阶段。线程死亡的原因有三个。一个是正常运行的线程完成了它的全部工作；另一个是线程被强制性地终止，如通过执行stop方法来终止一个线程[不推荐使用】，三是线程抛出未捕获的异常

4. 线程控制方法

- Java提供一个线程调度器来监控程序中启动后进入就绪状态的所有线程。线程调度器按照线程的优先级决定应调度哪个线程来执行。
- 线程的优先级用数字表示，范围从1到10
  - Thread.MIN_PRIORITY = 1
  - Thread.MAX_PRIORITY = 10
  - Thread.NORM_PRIORITY = 5
- 使用下述方法获得或设置线程对象的优先级。
  - int getPriority();
  - void setPriority(int newPriority);
- 注意：优先级低只是意味着获得调度的概率低。并不是绝对先调用优先级高后调用优先级低的线程。

4. 线程控制方法

- join ()
  - 阻塞指定线程等到另一个线程完成以后再继续执行
- sleep ()
  - 使线程停止运行一段时间，将处于阻塞状态
  - 如果调用了sleep方法之后，没有其他等待执行的线程，这个时候当前线程不会马上恢复执行！
- yield ()
  - 让当前正在执行线程暂停，不是阻塞线程，而是将线程转入就绪状态
  - 如果调用了yield方法之后，没有其他等待执行的线程，这个时候当前线程就会马上恢复执行！
- setDaemon()
  - 可以将指定的线程设置成后台线程
  - 创建后台线程的线程结束时，后台线程也随之消亡
  - 只能在线程启动之前把它设为后台线程
- interrupt()
  - 并没有直接中断线程，而是需要被中断线程自己处理
- stop()
  - 结束线程，不推荐使用

5. 线程同步

- 应用场景：
  - 多个用户同时操作一个银行账户。每次取款100元，取款前先检查余额是否足够。如果不够，放弃取款
- 分析
  - 使用多线程解决
  - 开发一个取款线程类，每个用户对应一个线程对象
  - 因为多个线程共享同一个银行账户，使用Runnable方式解决
- 思路
  - 创建银行账户类Account
  - 创建取款线程AccountRunnable
  - 创建测试类TestAccount，让两个用户同时取款

5. 线程同步

- 当多个线程访问同一个数据时，容易出现线程安全问题。需要让线程同步，保证数据安全

- 线程同步
  - 当两个或两个以上线程访问同一资源时，需要某种方式来确保资源在某一时刻只被一个线程使用

- 线程同步的实现方案
  - 同步代码块
    - synchronized (obj){ }
  - 同步方法
    - private synchronized void makeWithdrawal(int amt) {}

5. 线程同步

- 同步监视器
  - synchronized (obj){　}中的obj称为同步监视器
  - 同步代码块中同步监视器可以是任何对象，但是推荐使用共享资源作为同步监视器
  - 同步方法中无需指定同步监视器，因为同步方法的同步监视器是this，也就是该对象本事

- 同步监视器的执行过程
  - 第一个线程访问，锁定同步监视器，执行其中代码
  - 第二个线程访问，发现同步监视器被锁定，无法访问
  - 第一个线程访问完毕，解锁同步监视器
  - 第二个线程访问，发现同步监视器未锁，锁定并访问

5. 线程同步

- Lock锁
  - JDK1.5后新增功能，与采用synchronized相比，lock可提供多种锁方案，更灵活
  - java.util.concurrent.lock 中的 Lock 框架是锁定的一个抽象，它允许把锁定的实现作为 Java 类，而不是作为语言的特性来实现。这就为 Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。
  - ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义， 但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。
  - 注意：如果同步代码有异常，要将unlock()写入finally语句块
- Lock和synchronized的区别
  - 1.Lock是显式锁（手动开启和关闭锁，别忘记关闭锁），synchronized是隐式锁
  - 2.Lock只有代码块锁，synchronized有代码块锁和方法锁
  - 3.使用Lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类）
- 优先使用顺序：
  - Lock----同步代码块（已经进入了方法体，分配了相应资源）----同步方法（在方法体之外）

5. 线程同步

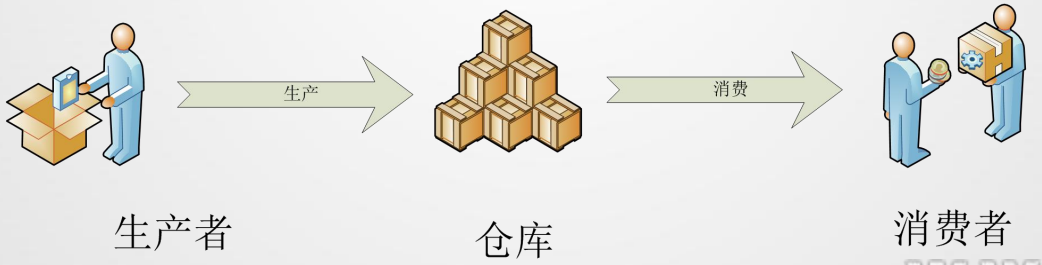
- 线程同步的好处
  - 解决了线程安全问题

- 线程同步的缺点
  - 性能下降
  - 会带来死锁

- 死锁
  - 当两个线程相互等待对方释放“锁”时就会发生死锁
  - 出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续
  - 多线程编程时应该注意避免死锁的发生

6. 线程通信

- 应用场景：生产者和消费者问题
  - 假设仓库中只能存放一件产品，生产者将生产出来的产品放入仓库，消费者将仓库中产品取走消费
  - 如果仓库中没有产品，则生产者将产品放入仓库，否则停止生产并等待，直到仓库中的产品被消费者取走为止
  - 如果仓库中放有产品，则消费者可以将产品取走消费，否则停止消费并等待，直到仓库中再次放入产品为止



6. 线程通信

- 分析
  - 这是一个线程同步问题，生产者和消费者共享同一个资源，并且生产者和消费者之间相互依赖，互为条件
  - 对于生产者，没有生产产品之前，要通知消费者等待。而生产了产品之后，又需要马上通知消费者消费
  - 对于消费者，在消费之后，要通知生产者已经消费结束，需要继续生产新产品以供消费
  - 在生产者消费者问题中，仅有synchronized是不够的
    - synchronized可阻止并发更新同一个共享资源，实现了同步
    - synchronized不能用来实现不同线程之间的消息传递（通信）

6. 线程通信

- Java提供了3个方法解决线程之间的通信问题

方法名	作　用
final void wait()	表示线程一直等待，直到其它线程通知
void wait(long timeout)	线程等待指定毫秒参数的时间
final void wait(long timeout,int nanos)	线程等待指定毫秒、微妙的时间
final void notify()	唤醒一个处于等待状态的线程
final void notifyAll()	唤醒同一个对象上所有调用wait()方法的线程，优先级别高的线程优先运行

均是java.lang.Object类的方法
都只能在同步方法或者同步代码块中使用，否则会抛出异常

6. 线程通信

- 实现思路
  - 定义产品类
  - 定义消费者线程
  - 定义生产者线程
  - 测试运行

6. 线程通信

- 产

```
//产品类
public class Product {
    private String name;//馒头 玉米饼
    private String color;//白色 黄色
    private boolean isProduce = false;//是否生产产品
    public synchronized void get(){
        //如果没有生产，等待
        if(isProduce == false){wait();   }
        System.out.println("消费者消费:"+name+" "+color); //消费产品
        isProduce = false; //修改状态：没有生产
        notify();//通知生产者生产
    }
    public synchronized void put(String name,String color){
        //如果已经生产，等待
        if(isProduce == true){    wait();//生产产品
        this.name = name;this.color = color;
        System.out.println("生产者生产："+this.name+" "+this.color);
        isProduce = true; //修改状态：已经生产
        notify(); //通知消费者消费
    }
}
```

6. 线程通信

- 定义生产者线程类和消费者线程类

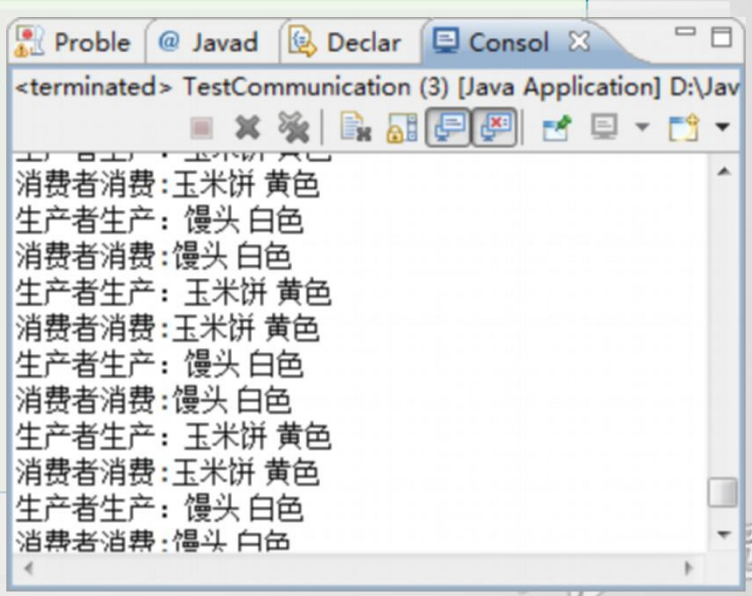
```
//生产者线程
public class Producer implements Runnable {
    private Product product;
    public Producer() {  }
    public Produce
     this.product
    }
    public void ru
        int i = 0;
        while (tru
            if (i
            produc
            } else
            produc
            }
            i++;
     }
    }
}
```

```
//消费者线程
public class Consumer implements Runnable{
    private Product product;
    public Consumer() {
     super();
    }
    public Consumer(Product product) {
        super();
        this.product = product;
    }
    public void run() {
        while(true){
        product.get();
        }
    }
}
```

6. 线程通信

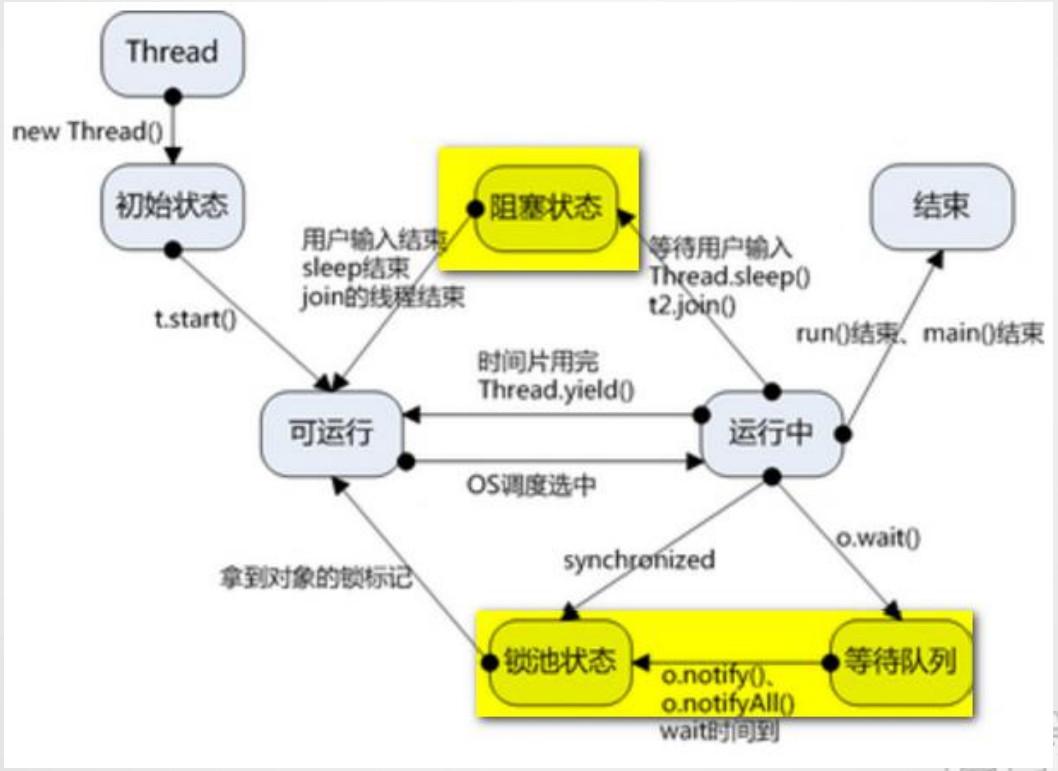
- 定义测试类

```
//测试类
public class TestCommunication {
    public static void main(String[] args) {
        //创建产品类（生产者和消费者操作的是同一个产品）
        Product product = new Product();
        //创建两个线程
        Consumer c = new Consumer(product);
        Thread t1 = new Thread(c);
        Producer p = new Producer(product);
        Thread t2 = new Thread(p);
        //启动两个线程
        t1.start();
        t2.start();
    }
}
```



6. 线程通信

- 更完整的线程生命周期



7. 线程组

- 线程组
  - 线程组表示一个线程的集合。
  - 线程组也可以包含其他线程组。线程组构成一棵树。在树中，除了初始线程组外，每个线程组都有一个父线程组。
  - 顶级线程组名system，线程的默认线程组名称是main
  - 在创建之初，线程被限制到一个组里，而且不能改变到一个不同的组

- 线程组的作用
  - 统一管理：便于对一组线程进行批量管理线程或线程组对象
  - 安全隔离：允许线程访问有关自己的线程组的信息，但是不允许它访问有关其线程组的父线程组或其他任何线程组的信息

- 查看ThreadGroup、Thread构造方法代码，观察默认线程组的情况

8. 线程池

- 什么是线程池
  - 创建和销毁对象是非常耗费时间的
  - 创建对象：需要分配内存等资源
  - 销毁对象：虽然不需要程序员操心，但是垃圾回收器会在后台一直跟踪并销毁
  - 对于经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
  - 思路：创建好多个线程，放入线程池中，使用时直接获取引用，不使用时放回池中。可以避免频繁创建销毁、实现重复利用

  - 生活案例：在尚学堂借用和归还电脑，共享单车
  - 技术案例：线程池、数据库连接池

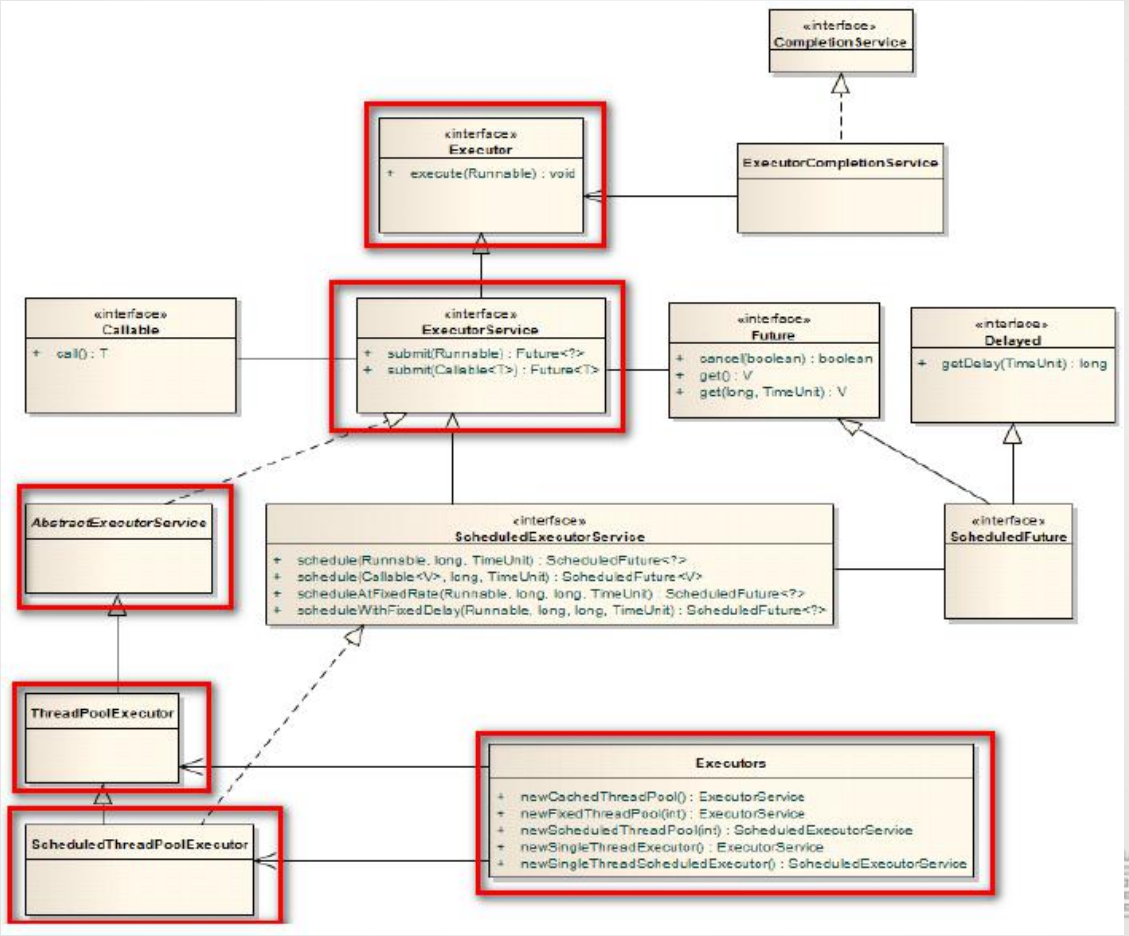
  - JDK1.5起，提供了内置线程池

8. 线程池

- 线程池的好处
  - 提高响应速度（减少了创建新线程的时间）
  - 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
  - 提高线程的可管理性：避免线程无限制创建、从而销耗系统资源，降低系统稳定性，甚至内存溢出或者CPU耗尽

- 线程池的应用场合
  - 需要大量线程，并且完成任务的时间端
  - 对性能要求苛刻
  - 接受突发性的大量请求

8. 线程池


«interface»
CompletionService
«interface»
Executor
+ execute(Runnable) : void
ExecutorCompletionService
«interface»
Callable
+ call() : T
«interface»
ExecutorService
+ submit(Runnable) : Future<?>
+ submit(Callable<T>) : Future<T>
«interface»
Future
+ cancel(boolean) : boolean
+ get() : V
+ get(long, TimeUnit) : V
«interface»
Delayed
+ getDelay(TimeUnit) : long
AbstractExecutorService
«interface»
ScheduledExecutorService
+ schedule(Runnable, long, TimeUnit) : ScheduledFuture<?>
+ schedule(Callable<V>, long, TimeUnit) : ScheduledFuture<V>
+ scheduleAtFixedRate(Runnable, long, long, TimeUnit) : ScheduledFuture<?>
+ scheduleWithFixedDelay(Runnable, long, long, TimeUnit) : ScheduledFuture<?>
«interface»
ScheduledFuture
ThreadPoolExecutor
Executors
+ newCachedThreadPool() : ExecutorService
+ newFixedThreadPool(int) : ExecutorService
+ newScheduledThreadPool(int) : ScheduledExecutorService
+ newSingleThreadExecutor() : ExecutorService
+ newSingleThreadScheduledExecutor() : ScheduledExecutorService
ScheduledThreadPoolExecutor

8. 线程池

- Executor：线程池顶级接口，只有一个方法
- ExecutorService：真正的线程池接口
  - void execute(Runnable command) ：执行任务/命令，没有返回值，一般用来执行Runnable
  - <T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行Callable
  - void shutdown() ：关闭连接池
- AbstractExecutorService：基本实现了ExecutorService的所有方法
- ThreadPoolExecutor：默认的线程池实现类
- ScheduledThreadPoolExecutor：实现周期性任务调度的线程池
- Executors：工具类、线程池的工厂类，用于创建并返回不同类型的线程池
  - Executors.newCachedThreadPool()：创建一个可根据需要创建新线程的线程池
  - Executors.newFixedThreadPool(n); 创建一个可重用固定线程数的线程池
  - Executors.newSingleThreadExecutor() ：创建一个只有一个线程的线程池
  - Executors.newScheduledThreadPool(n)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

8. 线程池

- 线程池参数
  - corePoolSize：核心池的大小
    - 默认情况下，创建了线程池后，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。
    - 但是当线程池中线程数量达到corePoolSize，就会把到达的任务放到队列中等待。
  - maximumPoolSize：最大线程数。
    - corePoolSize和maximumPoolSize之间的线程数会自动释放，小于等于corePoolSize的不会释放。当大于了这个值就会将任务由一个丢弃处理机制来处理。
  - keepAliveTime：线程没有任务时最多保持多长时间后会终止
    - 默认只限于corePoolSize和maximumPoolSize之间的线程
  - TimeUnit：
    - keepAliveTime的时间单位
  - BlockingQueue：
    - 存储等待执行的任务的阻塞队列，有多中选择，可以是顺序队列、链式队列等。

8. 线程池

- 线程池参数
  - ThreadFactory
    - 线程工厂，默认是DefaultThreadFactory，Executors的静态内部类
  - RejectedExecutionHandler：
    - 拒绝处理任务时的策略。如果线程池的线程已经饱和，并且任务队列也已满，对新的任务应该采取什么策略。
    - 比如抛出异常、直接舍弃、丢弃队列中最旧任务等，默认是直接抛出异常。
      - 1、CallerRunsPolicy：如果发现线程池还在运行，就直接运行这个线程
      - 2、DiscardOldestPolicy：在线程池的等待队列中，将头取出一个抛弃，然后将当前线程放进去。
      - 3、DiscardPolicy：什么也不做
      - 4、AbortPolicy：java默认，抛出一个异常：

8. 线程池

- 线程

```
public class TestPool2 {
        public static void main(String[] args) throws InterruptedException, ExecutionException {
                //创建线程池
                ExecutorService  pool = Executors.newCachedThreadPool();
                //使用线程池执行多个任务
                List<Future> futures = new ArrayList<Future>();
                for(int i=0;i<20;i++){
                        //创建一个任务
                        Callable<Integer> task = new RandomCallable();
                        //使用线程执行该任务
                        Future<Integer>  future = pool.submit(task);
                        //输出结果
                        //System.out.println(future.get());//需要等线程结束有了结果才执行
                        futures.add(future);
                }
                for(Future future :futures){
                        System.out.println(future.get());
                }
                //关闭线程池
                pool.shutdown();
        }
}
```

作业

- 进程和线程有什么联系和区别？
- 创建线程的两种方式分别是什么？
- 如何实现线程同步？
- Java中实现线程通信的三个方法的作用是什么？
- 线程池的原理和好处，线程池的拒绝策略