并发与并行 并发：同一时刻只有一条指令执行，但多进程指令快速轮行，宏观上多进程同时进行； 并行：在同一时刻，有多条指令在多个处理器上同时执行。

线程

1. 线程是进程的执行单元；一个线程必须要有一个父进程；线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不拥有系统资源；
2. 线程都是独立运行，**抢占式的**；可以创建和撤销另一个线程。
3. 优点：线程之间共享内存容易；分配资源代价少，效率高；

线程的创建与启动

1. 继承Thread类

> 定义时，线程类继承Thread类，实现run()方法。 使用时，创建该线程类，调用其start()方法来启动线程。

Thread.currentThread()静态方法，用于获取当前线程对象； getName()和setName()：设置线程对象名；

2. 实现Runnable接口

> 定义时，实现Runnable接口，重写run()方法； 使用时，创建runnable实现类的实例，**将该实例作为Thread的target（即构造器参数）来创建thread对象**，new Thread(target); 调用该**Thread对象的start()方法启动线程**； 一般可以直接写成：new Thread(target).start();来启动线程。 多个线程（**使用同一个runnable实现类的实例创建的**）可以共享同一个对象的变量。

以上两种方式，如果需要**在run()方法中**获取当前线程对象的话。第一种方法可以直接使用this作为当前线程对象，第二种则需要使用Thread.currentThread()来获取了。

3. Callable接口和Future接口

> 解决的问题：run()方法没有返回值，如果希望线程执行有返回值则使用这种方式； Callable接口是一个**函数式接口**，内含call()方法； 这种方法还是**需要创建Thread对象，使用start()方法**，但Callable接口不是Runnable接口的子接口，所以不能直接使用构造器创建Thread对象，需要使用Future接口； Future接口有一个**FutureTask**的实现类，其也实现了Runnable接口； 使用流程：
>
> - 创建FutureTask对象，使用其FutureTask(Callable callable)的构造器，把Callable接口的实现类传入；
> - 再将FutureTask对象作为Thread的参数创建Thread对象后启动线程；
> - 使用FutureTask对象的get()方法获取线程返回值； get()方法会阻塞程序；

一般推荐使用第二第三种创建线程，因为可以继承别的类。

如果希望**调用start()方法后，子线程立即执行**，则可以使用Thread.sleep(1);让当前线程睡眠1毫秒，1毫秒后就能让CPU启动别的线程。

线程需要经历：新建->就绪->运行->死亡四个阶段，还有一个特殊阶段是**阻塞**，运行->阻塞->就绪；**阻塞后不能直接运行，必须经历就绪状态**； 调用yield()方法可以让运行进入就绪状态。 进入阻塞的情况：

> 1. 调用sleep();
> 2. 调用了阻塞式IO方法；

> 3. 试图获取了一个同步监视器（即试图进入同步代码块，获取同步资源），而该同步监视器在被别的线程使用；
> 4. 等待通知；
> 5. 调用suspend()方法；对应解除的是：被调用了resume()恢复方法。

线程死亡 情况：执行结束；抛出未处理的异常；调用了stop()方法。 可以调用**线程对象的**isAlive()方法（实例方法），来检查线程是否死亡；

控制线程

> join()方法：如果在某一个线程A中调用别的线程B的join()方法，则调用线程A会被阻塞，直到B线程执行完毕；

后台线程

> 任务是为其他线程服务，**如果所有前台线程死亡，后台线程也会自动死亡**； 通过调用Thread对象的setDaemon(true)方法，把该线程对象设置为后台线程。 **必须在start()方法前设置好！**

sleep()方法：参数是毫秒 yield()方法

> 静态方法；将某个线程暂停，进入就绪状态；**只有优先级相同或者高于当前线程的**才会获得执行机会，低的是没有的！

setPriority(int i)设置线程优先级 整数范围在1到10之间，推荐使用

> - MIN_PRIORITY = 1
> - NORM_PRIORITY = 5
> - MAX_PRIORITY = 10 三个Thread类的静态变量来设置，方便跨平台，因为有些平台没有10个线程优先级；

线程同步

1. 同步代码块

```
synchronized (obj){
    代码块
}
```

obj就是同步监视器； 2. 同步方法 使用synchronized关键修饰的方法，同步监视器就是this 以下情况下**不会**释放同步监视器：

> 1. 调用sleep()，yield()方法暂停当前进程；
> 2. 调用suspend()方法； 当同步代码中：
> 3. 执行结束；
> 4. 抛出异常；
> 5. 调用wait()方法；

同步锁：Lock对象与condition

线程通信 概念：控制线程的轮换执行；

1. 调用**Object类（不是Thread类）**的

> wait():暂停当前线程，并**释放同步监视器，直到使用该同步监视器的别的线程调用notify()方法**；
> notify():任意唤醒一个**等待该同步监视器的**线程，但只有当前线程放弃同步监视器才执行别的线程。
> notifyAll():唤醒所有等待线程，但只有当前线程放弃同步监视器才执行别的线程。 **只有同步监视器对象才能调用这几个方法，即同步代码块中调用这些方法！**

2. 使用Condition类

3. 使用阻塞队列BlockingQueue**接口**来控制 具体控制思路如下： 在两种线程类中加入BlockingQueue实现类的对象，执行run()代码时，加入对BlockingQueue队列情况的判断，从而达到控制线程执行的效果。BlockingQueue有两个阻塞式方法：

> 1. put(E e)：尝试放入元素，如果队列元素已满，则阻塞线程；
> 2. take(E e)：尝试取出从队头元素，如果队列元素已空，则阻塞线程。 通过上面的两个方法，让**两种线程代表的行为进行轮流执行**。

线程组

1. 使用ThreadGroup来表示线程组，允许程序直接对线程组进行控制；
2. 一旦某个线程加入某个线程组则直到死亡都不可改变线程组；
3. Thread(ThreadGroup g, Runnable a);还有一个多加一个参数为线程名的构造器，指定某个线程为某个线程组；（通过实现接口创建线程的）
4. Thread(ThreadGroup g, String s)；直接创建一个新线程，且线程名和线程组都指定好。（通过继承创建线程的）
5. getThreadGroup()返回所属线程组；
6. 线程组能对其内的线程进行：返回线程数；中断；是否为后台线程组；设置为后台线程组；设优先级；

线程的异常处理： Thread类提供了**两个方法**设置异常处理器（Thread.UncaughtExceptionHandler对象），而ThreadGroup线程组自动实现了异常处理的接口Thread.UncaughtExceptionHandler； jvm在线程执行时遇到抛出的未处理的异常时，会在线程结束前自动查找是否有对应的Thread.UncaughtExceptionHandler对象，会调用该对象的void uncaughtException(Thread t, Throwable e)处理异常；参数t是代表出现异常的线程；参数e代表抛出的异常；

线程池

1. 概念：系统启动时创建大量空闲线程（指定数量），通过反复使用线程来完成不同的Runnable对象的run()方法；
2. 优势：降低资源消耗（线程反复使用）；提高响应速度（任务不再需要等待线程创建就能执行）；提高线程的可管理性。
3. 用Executors工具类来创建各种线程池对象（不推荐）

> 原因：Executors工具类的创建线程方法实际上都是调用ThreadPoolExecutor创建，有几个方法创建的线程池对象由于设置参数的原因，可能会耗费非常大的内存。

ThreadPoolExecutor

1. 源码解析

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

- corePoolSize 线程池核心池的大小
- maximumPoolSize 线程池的最大线程数
- keepAliveTime 当线程数大于核心时，此为终止前多余的空闲线程 等待新任务的最长时间
- unit keepAliveTime 的时间单位
- workQueue 用来储存等待执行任务的队列
- threadFactory 线程工厂
- handler 拒绝策略 corePoolSize：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程(**即在线程数没达到该数值时，有新任务就会创建新线程**)，等到需要执行的任务数大于线程池基本大小时就不再创建。
  maximumPoolSize：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。(如果使用了无界队列，这个参数无效) **上面两个参数的区别：**

> 创建线程池后，线程池不会自动创建线程，**只有有任务才会创建新线程**； 当线程数达到corePoolSize后，**新的任务到来后会加入阻塞队列**，当队列满了，才会继续开始创建线程，直至最大值。

keepAliveTime：线程池的工作线程空闲后，保持存活的时间。如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率

TimeUnit：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)

runnableTaskQueue：用于保存等待执行的任务的阻塞队列。有5种可选择：

> 1. ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。
> 2. LinkedBlockingQueue：一个基于链表结构的无界阻塞队列，此队列按FIFO排序元素，吞吐量通常高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。
> 3. SynchronousQueue：**默认**。一个不存储元素的阻塞队列。每个线程的插入必须等另一个线程的移除，否则插入操作一直处于阻塞状态，吞吐量通常要高于 LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。
> 4. PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。
> 5. DelayQueue： 一个使用优先级队列实现的无界阻塞队列。 **2和3**队列容易导致内存不足。(？)

ThreadFactory：通过线程工厂给每个创建出来的线程设置名字，帮助Debug和定位问题(起名用)
RejectedExecutionHandler：**当队列和线程池都满了，说明线程池处于饱和状态**，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。jdk提供4种策略

> ThreadPoolExecutor.AbortPolicy: 丢弃任务并抛出RejectedExecutionException异常(默认)
> ThreadPoolExecutor.DiscardPolicy：丢弃任务，但是不抛出异常

> ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务(重复此过程)
> ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

ExecutorService接口和ScheduledExecutorService

1. 使用ThreadPoolExecutor构造器创建一个ExecutorService（**ThreadPoolExecutor实现了ExecutorService接口**）
2. 使用submit(Runnable a)来提交任务可以获得一个Future对象，用来提交有返回值的任务；
3. 使用execute()方法用于提交不需要返回值的任务；
4. 使用shutdown()方法关闭线程池。