西部开源

多线程详解

Java.Thread

讲师：秦疆

西安西部开源教育科技有限公司

目录

CONTENTS

西部开源

01 线程简介

任务，进程，线程，多线程

西部开源

多任务

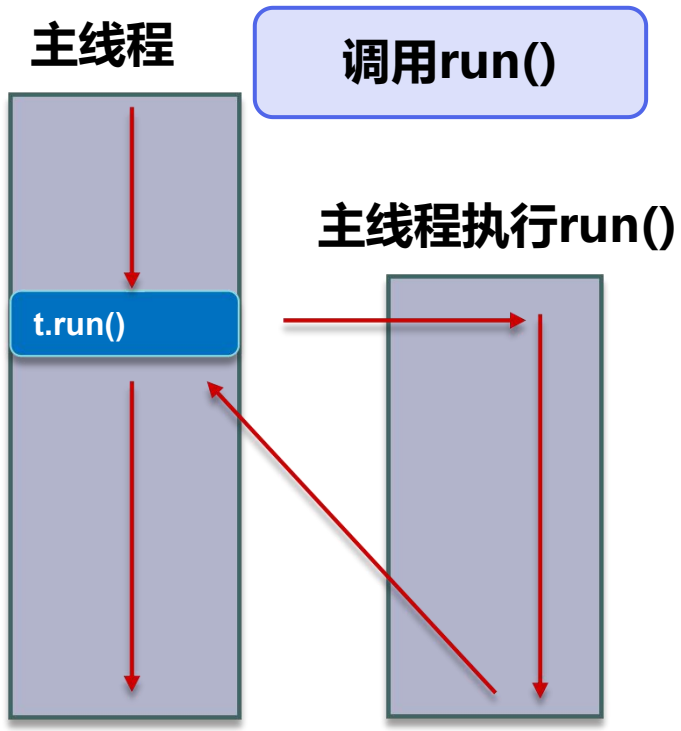
- 现实中太多这样同时做多件事情的例子了，看起来是多个任务都在做，其实本质上我们的大脑在同一时间依旧只做了一件事情。

多线程

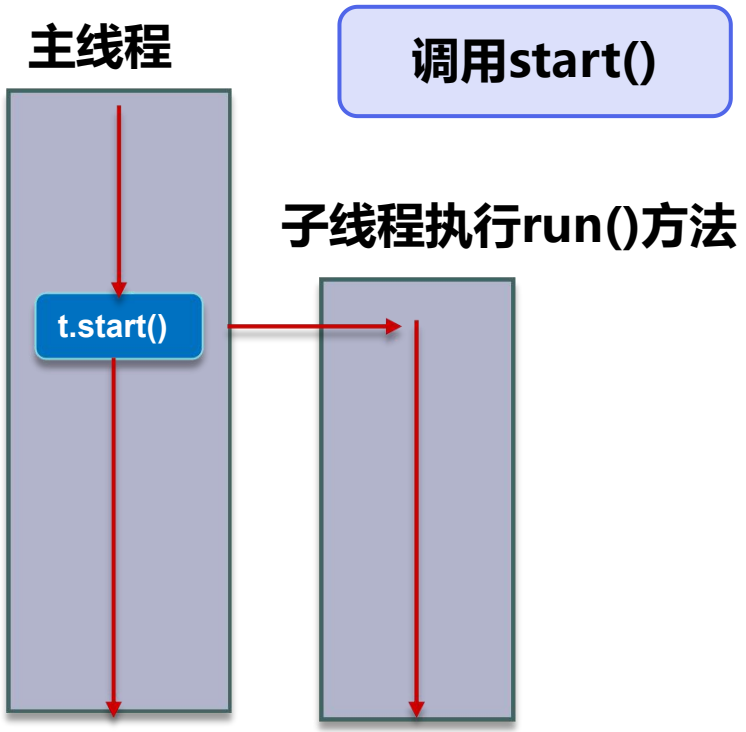
原来是一条路，慢慢因为车太多了，道路堵塞，效率极低。
为了提高使用的效率，能够充分利用道路，于是加了多个车道。
从此，妈妈再也不用担心道路堵塞了。

说说你们的多线程例子（生活，游戏，编程）

普通方法调用和多线程

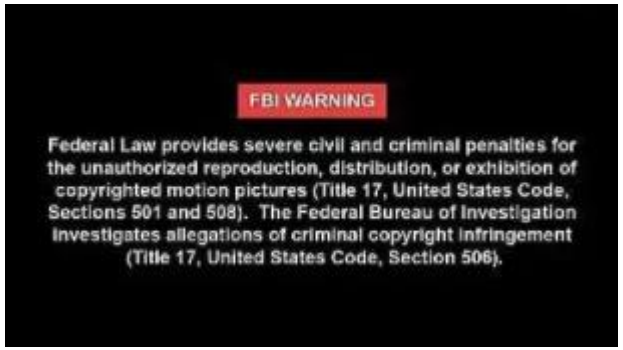


只有主线程一条执行路径

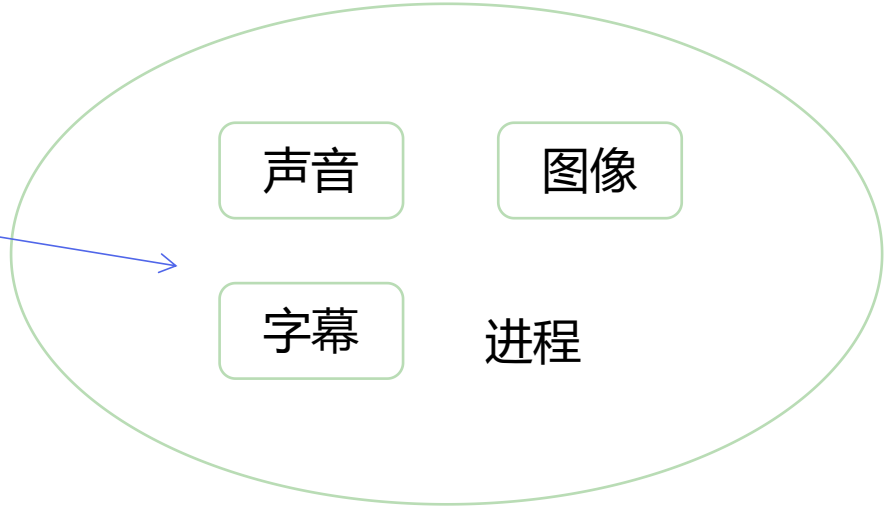


多条执行路径，主线程和子线程并行交替执行

程序.进程.线程



在操作系统中运行的程序就是进程，比如你的QQ，播放器，游戏，IDE等等。。。。



一个进程可以有多个线程，如视频中同时听声音，看图像，看弹幕，等等

Process与Thread

- 说起进程，就不得不说下**程序**。程序是指令和数据的有序集合，其本身没有任何运行的含义，是一个静态的概念。
- 而**进程**则是执行程序的一次执行过程，它是一个动态的概念。是系统资源分配的单位
- 通常在一个进程中可以包含若干个**线程**，当然一个进程中至少有一个线程，不然没有存在的意义。线程是CPU调度和执行的的单位。

注意：很多多线程是模拟出来的，真正的多线程是指有多个cpu，即多核，如服务器。如果是模拟出来的多线程，即在一个cpu的情况下，在同一个时间点，cpu只能执行一个代码，因为切换的很快，所以就有同时执行的错局。

本章核心概念

西部开源

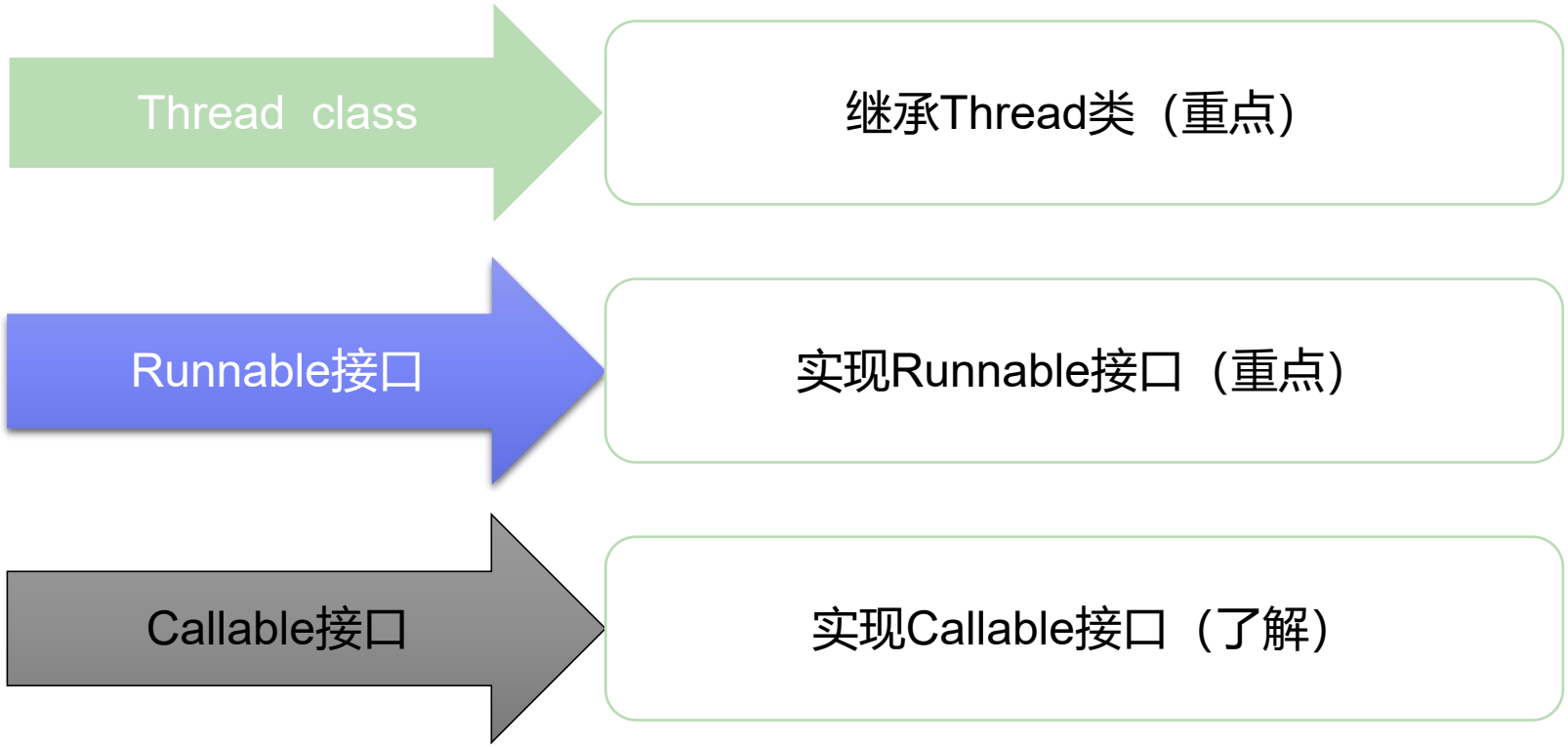
- 线程就是独立的执行路径；
- 在程序运行时，即使没有自己创建线程，后台也会有多个线程，如主线程，gc线程；
- main() 称之为主线程，为系统的入口，用于执行整个程序；
- 在一个进程中，如果开辟了多个线程，线程的运行由调度器安排调度，调度器是与操作系统紧密相关的，先后顺序是不能认为的干预的。
- 对同一份资源操作时，会存在资源抢夺的问题，需要加入并发控制；
- 线程会带来额外的开销，如cpu调度时间，并发控制开销。
- 每个线程在自己的工作内存交互，内存控制不当会造成数据不一致

02 线程创建

Thread 、Runnable、Callable

西部开源

三种创建方式



Thread

学习提示：查看JDK帮助文档

西部开源

- 自定义线程类继承Thread类
- 重写run()方法，编写线程执行体
- 创建线程对象，调用start()方法启动线程

```java
public class StartThread1 extends Thread {

    //线程入口点
    @Override
    public void run() {
        //线程体
        for (int i = 0; i < 20; i++) {
            System.out.println("我在听课=====");
        }
    }
}
```

```java
public static void main(String[] args) {

    //创建线程对象
    StartThread1 t = new StartThread1();
    t.start();
```

线程不一定立即执行，CPU安排调度

案例：下载图片

西部开源

第一步

```
//文件下载工具类
class WebDownloader{
    //远程路径，存储名字
    public void downloader(String url,String name){
        try {
            //Commons IO是针对开发IO流功能的工具类库。
            //FileUtils文件工具，复制url到文件
            FileUtils.copyURLToFile(new URL(url),new File(name));
        } catch (IOException e) {
            e.printStackTrace();
            System.out.println("文件下载失败");
        }
    }
}
```

第二步

```
@Override
public void run() {
    WebDownloader webDownloader = new WebDownloader();
    webDownloader.downloader(url,name);
    System.out.println(name);
}
```

第三步

```
public static void main(String[] args)
    StartThread2 t1 = new StartThread2
    StartThread2 t2 = new StartThread2
    StartThread2 t3 = new StartThread2

    //启动三个线程
    t1.start();
    t2.start();
    t3.start();
```

实现Runnable

学习提示：查看JDK帮助文档

西部开源

- 定义MyRunnable类实现Runnable接口
- 实现run()方法，编写线程执行体
- 创建线程对象，调用start()方法启动线程

```java
public class StartThread3 implements Runnable{
    @Override
    public void run() {
        //线程体
        for (int i = 0; i < 20; i++) {
            System.out.println("我在听课=====");
        }
    }
}
```

```java
//创建实现类对象
StartThread3 st = new StartThread3();
//创建代理类对象
Thread thread = new Thread(st);
//启动
thread.start();
```

推荐使用Runnable对象，因为Java单继承的局限性

小结

西部开源

- 继承Thread类
  - 子类继承Thread类具备多线程能力
  - 启动线程：子类对象. start()
  - 不建议使用：避免OOP单继承局限性

```
//一份资源
StartThread4 station = new StartThread4();

//多个代理
new Thread(station, name: "小明").start();
new Thread(station, name: "老师").start();
new Thread(station, name: "小红").start();
```

- 实现Runnable接口
  - 实现接口Runnable具有多线程能力
  - 启动线程：传入目标对象+Thread对象.start()
  - 推荐使用：避免单继承局限性，灵活方便，方便同一个对象被多个线程使用

案例：龟兔赛跑-Race

1. 首先来个赛道距离，然后要离终点越来越近
2. 判断比赛是否结束
3. 打印出胜利者
4. 龟兔赛跑开始
5. 故事中是乌龟赢的，兔子需要睡觉，所以我们来模拟兔子睡觉
6. 终于，乌龟赢得比赛

实现Callable接口（了解即可）

西部开源

1. 实现Callable接口，需要返回值类型
2. 重写call方法，需要抛出异常
3. 创建目标对象
4. 创建执行服务：ExecutorService ser = Executors.newFixedThreadPool(1);
5. 提交执行：Future<Boolean> result1 = ser.submit(t1);
6. 获取结果：boolean r1 = result1.get()
7. 关闭服务：ser.shutdownNow();

演示：利用callable改造下载图片案例

静态代理

- 你：真实角色
- 婚庆公司：代理你，帮你处理结婚的事
- 结婚：实现都实现结婚接口即可

演示：实现静态代理对比Thread

Lamda表达式

- λ 希腊字母表中排序第十一位的字母，英语名称为Lambda
- 避免匿名内部类定义过多
- 其实质属于函数式编程的概念

（params）-> expression [表达式]
（params）-> statement [语句]
（params）-> { statements }


```
a-> System.out.println("i like lambda-->"+a);
```

new Thread（()->System.out.println(“多线程学习。。。。”)）.start();

Lamda表达式

- 为什么要使用lambda表达式
  - 避免匿名内部类定义过多
  - 可以让你的代码看起来很简洁
  - 去掉了一堆没有意义的代码，只留下核心的逻辑。

- 也许你会说，我看了Lambda表达式，不但不觉得简洁，反而觉得更乱，看不懂了。那是因为我们还没有习惯，用的多了，看习惯了，就好了。

Lamda表达式

西部开源

- 理解Functional Interface（函数式接口）是学习Java8 lambda表达式的关键所在。
- 函数式接口的定义：
  - 任何接口，如果只包含唯一一个抽象方法，那么它就是一个函数式接口。

```
public interface Runnable {
        public abstract void run();
}
```

  - 对于函数式接口，我们可以通过lambda表达式来创建该接口的对象。

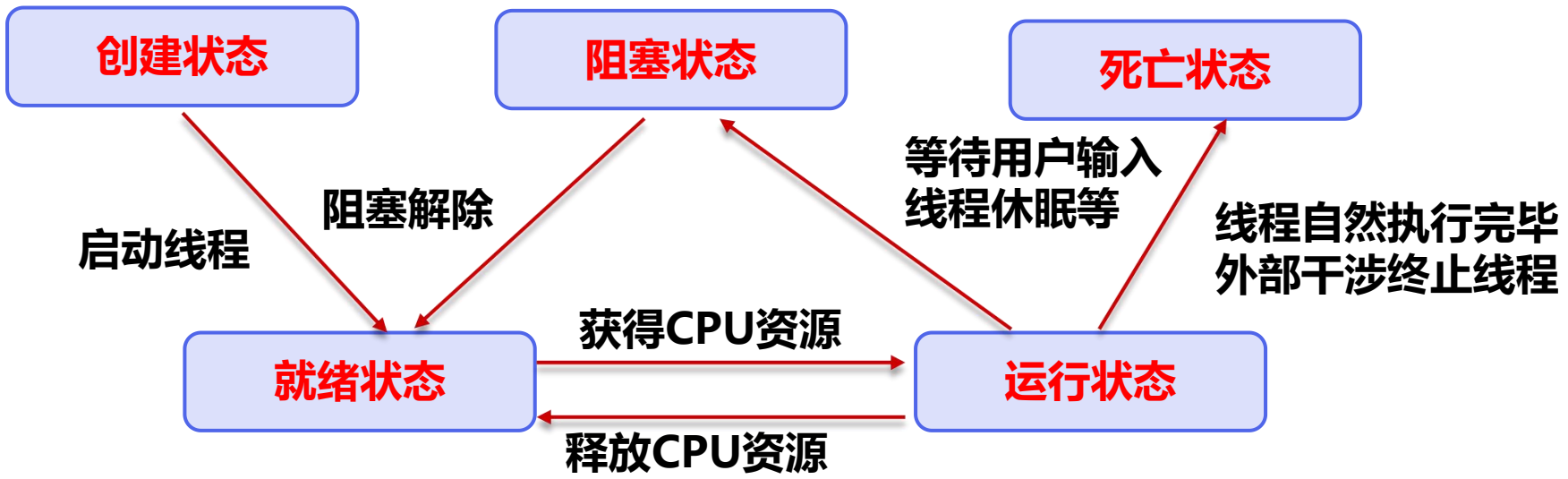
演示：代码推导lambda表达式

03 线程状态

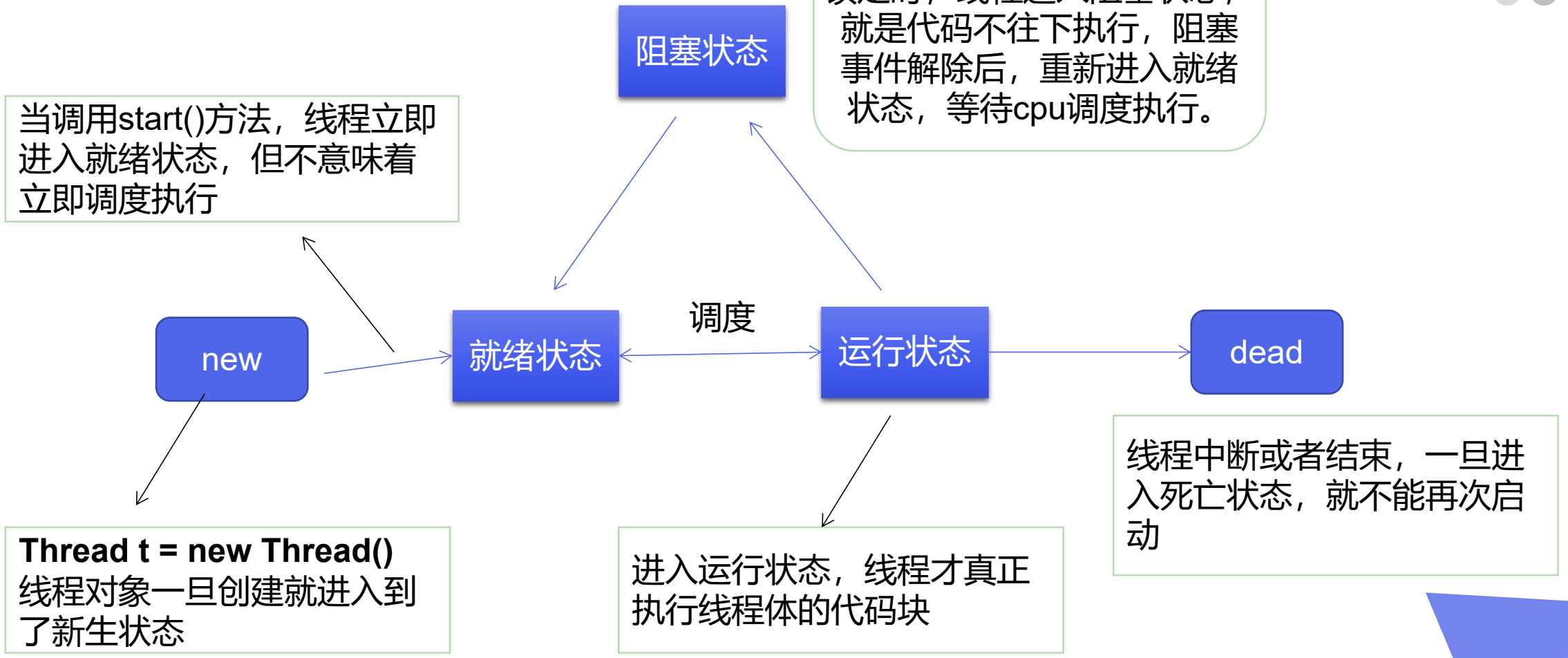
五大状态

西部开源

线程状态



线程状态



线程方法

西部开源

方　　法	说　　明
setPriority(int newPriority)	更改线程的优先级
static void sleep(long millis)	在指定的毫秒数内让当前正在执行的线程休眠
void join()	等待该线程终止
static void yield()	暂停当前正在执行的线程对象，并执行其他线程
void interrupt()	中断线程，别用这个方式
boolean isAlive()	测试线程是否处于活动状态

停止线程

- 不推荐使用JDK提供的 stop()、destroy()方法。【已废弃】
- 推荐线程自己停止下来
- 建议使用一个标志位进行终止变量

  当flag=false，则终止线程运行。

```
public class TestStop implements Runnable {
    //1.线程中定义线程体使用的标识
    private boolean flag = true;

    @Override
    public void run() {
        //2.线程体使用该标识
        while (flag){
            System.out.println("run... Thread");
        }
    }

    //3.对外提供方法改变标识
    public void stop(){
        this.flag = false;
    }

}
```

线程休眠

- sleep (时间) 指定当前线程阻塞的毫秒数；
- sleep存在异常InterruptedException；
- sleep时间达到后线程进入就绪状态；
- sleep可以模拟网络延时，倒计时等。
- 每一个对象都有一个锁，sleep不会释放锁；

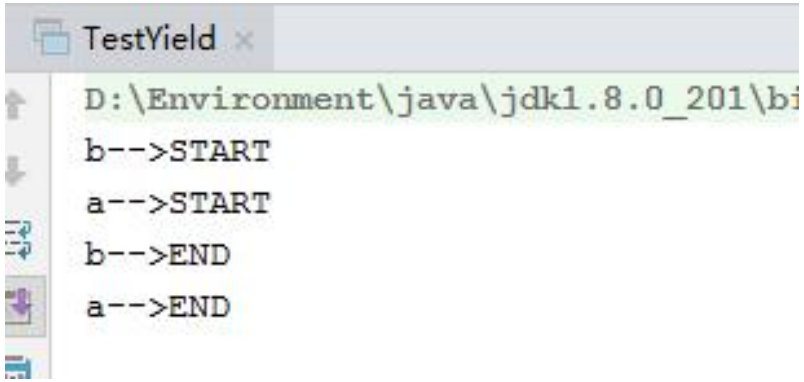
演示：计时

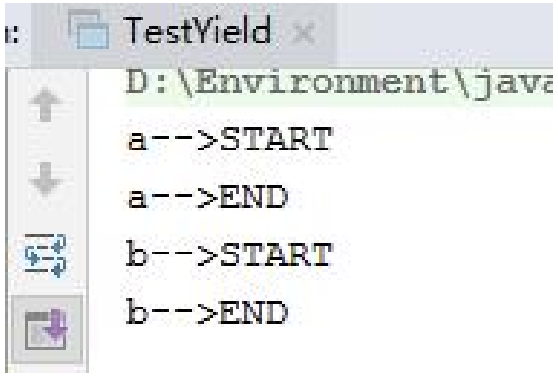
```
//实时获取车位信息
public void GetOnlineInfo()
{
    HttpBrowserCapabilities bc = Request.Browser;
    int hbcWidth = bc.ScreenPixelsWidth;
    //string hbcHeight = bc.ScreenPixelsHeight.ToString();

    //项目经理要求这里运行缓慢,好让客户给钱优化，并得到明显的速度提升
    Thread.Sleep(2000);
```

线程礼让

- 礼让线程，让当前正在执行的线程暂停，但不阻塞
- 将线程从运行状态转为就绪状态
- **让cpu重新调度，礼让不一定成功！看CPU心情**





Join

- Join合并线程，待此线程执行完成后，再执行其他线程，其他线程阻塞
- 可以想象成插队

```java
public class TestJoin implements Runnable {
    public static void main(String[] args) throws InterruptedException {
        TestJoin testJoin = new TestJoin();
        Thread thread = new Thread(testJoin);
        thread.start();

        for (int i = 0; i < 100; i++) {
            if (i==50){
                thread.join(); //main线程阻塞
            }
            System.out.println("main..."+i);
        }
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("join..."+i);
        }
    }
}
```

线程状态观测

西部开源

- Thread.State

查看JDK帮助文档

线程状态。 线程可以处于以下状态之一：

- NEW
  尚未启动的线程处于此状态。
- RUNNABLE
  在Java虚拟机中执行的线程处于此状态。
- BLOCKED
  被阻塞等待监视器锁定的线程处于此状态。
- WAITING
  正在等待另一个线程执行特定动作的线程处于此状态。
- TIMED_WAITING
  正在等待另一个线程执行动作达到指定等待时间的线程处于此状态。
- TERMINATED
  已退出的线程处于此状态。

一个线程可以在给定时间点处于一个状态。 这些状态是不反映任何操作系统线程状态的虚拟机状态。

线程优先级

西部开源

- 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，线程调度器按照优先级决定应该调度哪个线程来执行。
- 线程的优先级用数字表示，范围从1~10.
  - Thread.MIN_PRIORITY = 1;
  - Thread.MAX_PRIORITY = 10;
  - Thread.NORM_PRIORITY = 5;
- 使用以下方式改变或获取优先级
  - getPriority() . setPriority(int xxx)

优先级的设定建议在start()调度前

优先级低只是意味着获得调度的概率低.并不是优先级低就不会被调用了.这都是看CPU的调度

守护(daemon)线程

西部开源

- 线程分为**用户线程**和**守护线程**
- 虚拟机必须确保用户线程执行完毕
- 虚拟机不用等待守护线程执行完毕
- 如,后台记录操作日志,监控内存,垃圾回收等待..

人生不过三万天

03 线程同步

多个线程操作同一个资源

西部开源

并发

西部开源

- 并发：**同一个对象**被**多个线程**同时操作

上万人同时抢100张票

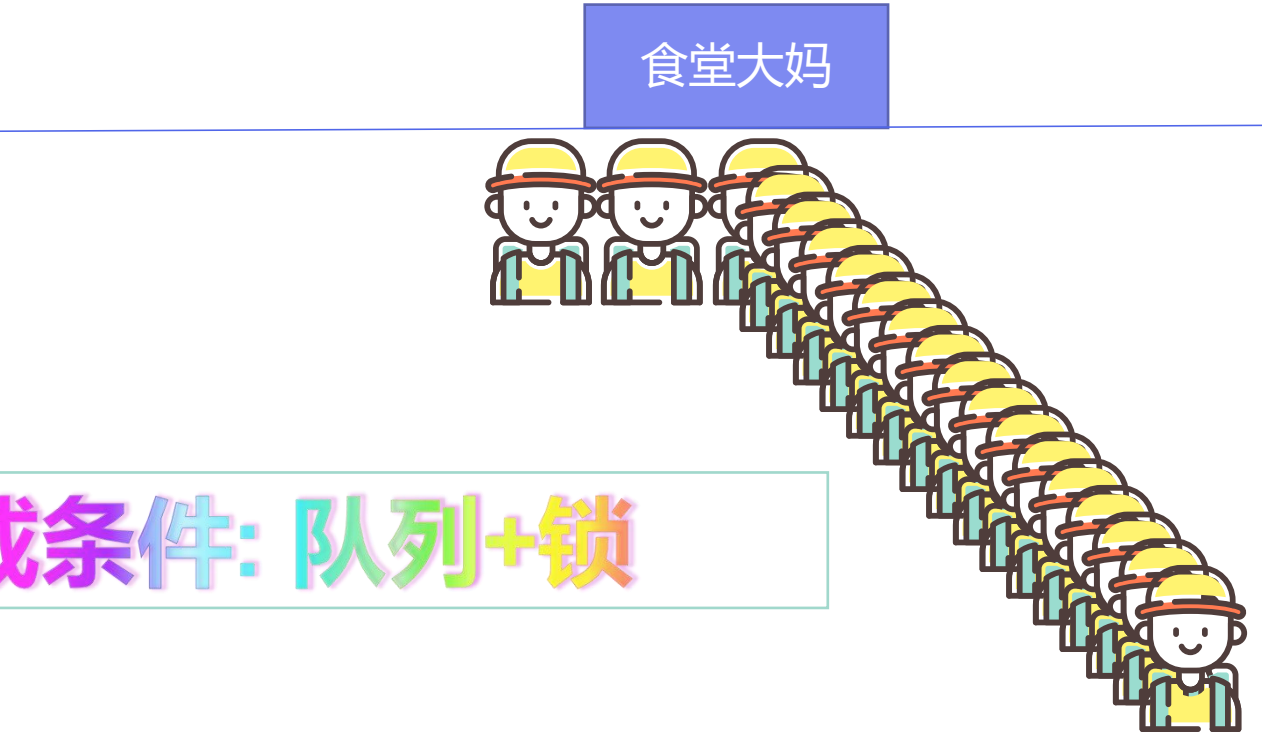
两个银行同时取钱

线程同步

- 现实生活中,我们会遇到”同一个资源，多个人都想使用”的问题， 比如,食堂排队打饭，每个人都想吃饭，最天然的解决办法就是，排队．一个个来.

- 处理多线程问题时，多个线程访问同一个对象，并且某些线程还想修改这个对象．这时候我们就需要线程同步．线程同步其实就是一种等待机制，多个需要同时访问此对象的线程进入这个**对象的等待池** 形成队列, 等待前面线程使用完毕，下一个线程再使用

队列 和 锁



形成条件: 队列+锁

线程同步

- 由于同一进程的多个线程共享同一块存储空间，在带来方便的同时,也带来了访问冲突问题，为了保证数据在方法中被访问时的正确性，在访问时加入 **锁机制** **synchronized**，当一个线程获得对象的排它锁，独占资源，其他线程必须等待，使用后释放锁即可．存在以下问题：
  - 一个线程持有锁会导致其他所有需要此锁的线程挂起；
  - 在多线程竞争下，加锁，释放锁会导致比较多的上下文切换 和 调度延时,引起性能问题；
  - 如果一个优先级高的线程等待一个优先级低的线程释放锁 会导致优先级倒置，引起性能问题．

同步方法

西部开源

- 由于我们可以通过 private 关键字来保证数据对象只能被方法访问，所以我们只需要针对方法提出一套机制，这套机制就是 synchronized 关键字，它包括两种用法：synchronized 方法 和synchronized 块 .

  同步方法： public synchronized void method(int args) {}

- synchronized方法控制对 “对象” 的访问，每个对象对应一把锁，每个synchronized方法都必须获得调用该方法的对象的锁才能执行，否则线程会阻塞，方法一旦执行，就独占该锁，直到该方法返回才释放锁，后面被阻塞的线程才能获得这个锁，继续执行

  缺陷：**若将一个大的方法申明为synchronized 将会影响效率**

同步方法弊端

- **方法里面需要修改的内容才需要锁，锁的太多，浪费资源**

同步块

西部开源

- 同步块 : synchronized **(Obj) { }**
- **Obj** 称之为 **同步监视器**
  - **Obj** 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就是this，就是这个对象本身，或者是 class [反射中讲解]
- 同步监视器的执行过程
  1. 第一个线程访问，锁定同步监视器，执行其中代码 .
  2. 第二个线程访问，发现同步监视器被锁定，无法访问 .
  3. 第一个线程访问完毕，解锁同步监视器 .
  4. 第二个线程访问, 发现同步监视器没有锁，然后锁定并访问

死锁

西部开源

- 多个线程各自占有一些共享资源，并且互相等待其他线程占有的资源才能运行，而导致两个或者多个线程都在等待对方释放资源，都停止执行的情形．某一个同步块同时拥有“**两个以上对象的锁**”时，就可能会发生“死锁”的问题．

死锁避免方法

西部开源

- 产生死锁的四个必要条件：
  1. 互斥条件：一个资源每次只能被一个进程使用。
  2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
  3. 不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。
  4. 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

上面列出了死锁的四个必要条件，我们只要想办法破其中的任意一个或多个条件就可以避免死锁发生

Lock(锁)

西部开源

- 从JDK 5.0开始，Java提供了更强大的线程同步机制——通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当
- java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象
- ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁。

Lock(锁)

西部开源

```
class A{
        private final ReentrantLock lock = new ReenTrantLock();
        public void m(){
                lock.lock();
                try{
                        //保证线程安全的代码;
                }
        finally{
                lock.unlock();
                //如果同步代码有异常，要将unlock()写入finally语句块
            }
        }
}
```

synchronized 与 Lock 的对比

西部开源

- Lock是显式锁（手动开启和关闭锁，别忘记关闭锁）synchronized是隐式锁，出了作用域自动释放
- Lock只有代码块锁，synchronized有代码块锁和方法锁
- 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）

04 线程协作

生产者消费者模式

西部开源

线程通信

西部开源

- 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 .

线程通信-分析

西部开源

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，要通知生产者已经结束消费，需要生产新的产品以供消费.
- 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)

线程通信

西部开源

- Java提供了几个方法解决线程之间的通信问题

方法名	作用
wait()	表示线程一直等待，直到其他线程通知，与sleep不同，会释放锁
wait(long timeout)	指定等待的毫秒数
notify()	唤醒一个处于等待状态的线程
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调度

注意：均是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IllegalMonitorStateException

解决方式1

并发协作模型“ 生产者 / 消费者模式 ”--->管程法

- 生产者 : 负责生产数据的模块 (可能是方法 , 对象 , 线程 , 进程) ;
- 消费者 : 负责处理数据的模块 (可能是方法 , 对象 , 线程 , 进程) ;
- 缓冲区 : 消费者不能直接使用生产者的数据 , 他们之间有个“ 缓冲区

生产者将生产好的数据放入缓冲区 , 消费者从缓冲区拿出数据

解决方式2

西部开源

- 并发协作模型“ 生产者 / 消费者模式 ”--->信号灯法

使用线程池

西部开源

- 背景：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：提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。
- 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）
  - 便于线程管理(....)
    - corePoolSize：核心池的大小
    - maximumPoolSize：最大线程数
    - keepAliveTime：线程没有任务时最多保持多长时间后会终止

使用线程池

西部开源

- JDK 5.0起提供了线程池相关API：**ExecutorService** 和 **Executors**
- ExecutorService：真正的线程池接口。常见子类ThreadPoolExecutor
  - 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
- 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