



java.thread

目录

- 线程简介
- 线程实现
- 线程状态
- 线程同步
- 生产消费者
- 高级主题



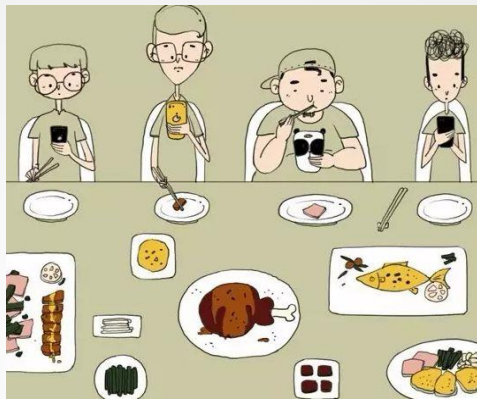
介绍

INTRODUCTION

线程简介

PART ONE

多任务



现实中太多的同时进行例子了，看似是多任务，其实在一个时间点我们的大脑还是只做了一件事情



多线程



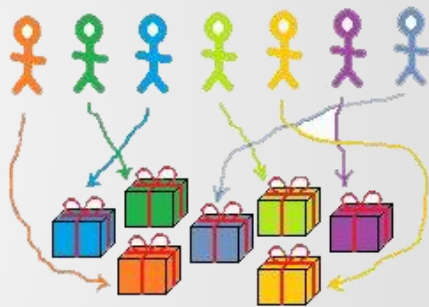
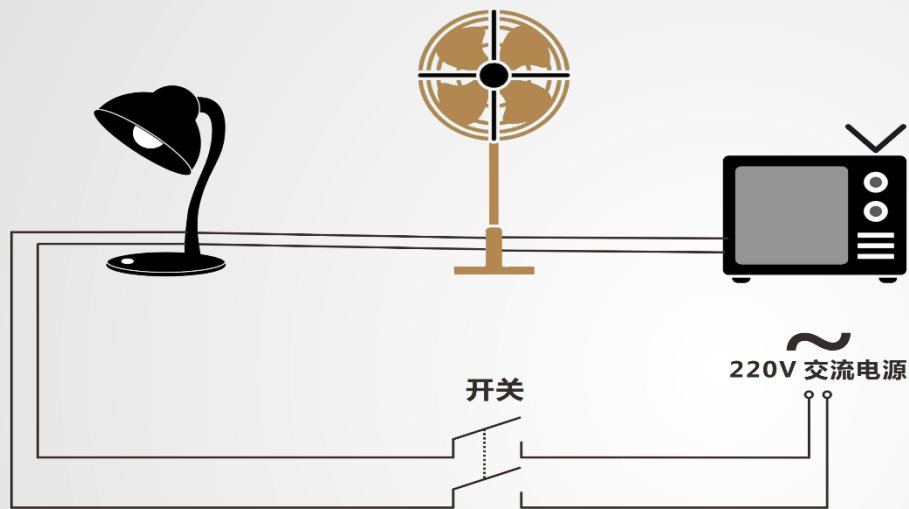
- 原来是一条路，因为车多了，为提高使用效率，充分使用这条道路，中间加了个栅栏，变成了多条车道，所有的车共享这条路。



- 游戏的多线程应用最广了，在同一个界面中操作多个角色和不同动作。

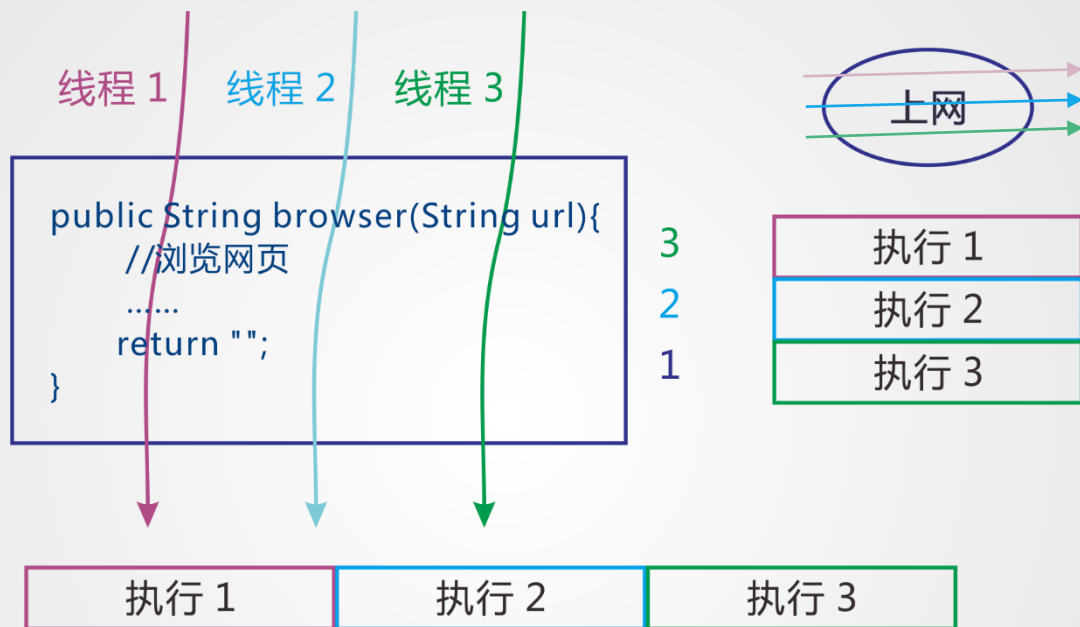


多线程



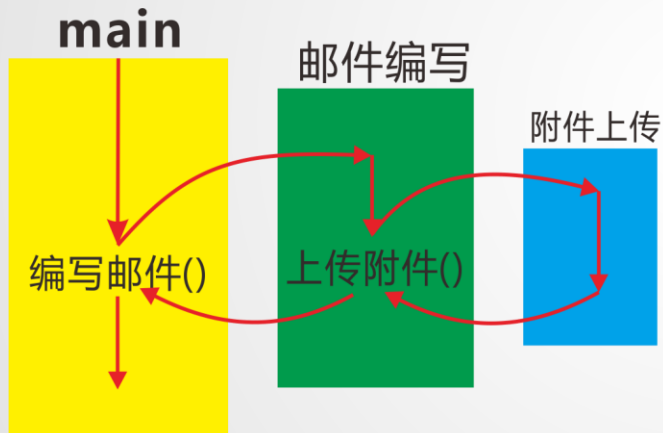
想象下现实中，三个电器接在同一个电线上，打开任何的电源，都能够同时亮起，同时工作。

多线程

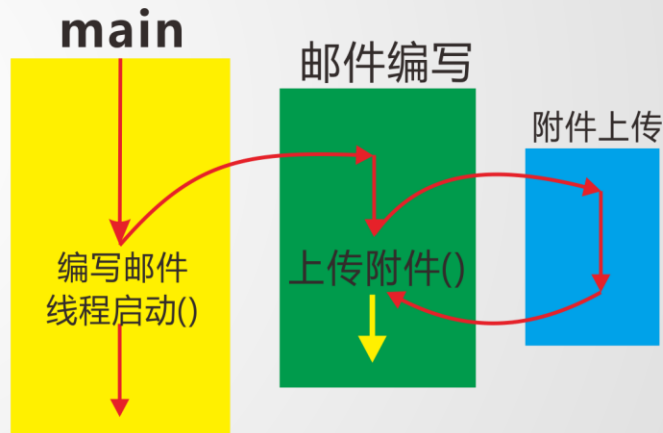


11.11
购物狂欢节

- 方法间调用：普通方法调用，从哪里来到哪里去，闭合的一条路径
- 多线程使用：开辟了多条路径



一条路径

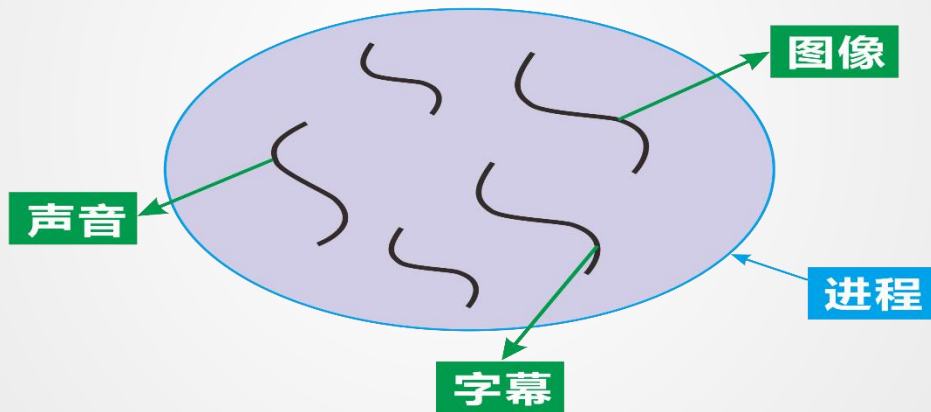


并行的两条路径

程序、进程与线程



在操作系统中运行中的程序就是进程，如看视频



一个进程可以有多个线程，如视频中同时听声音、看图像、显示字幕

Process与Thread

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。

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觉。

核心概念

- 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，后台也会存在多个线程，如gc线程、主线程；
- `main()` 称之为主线程，为系统的入口点，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，先后顺序是不能人为的干预的；
- 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，加载和存储主内存控制不当会造成数据不一致。



线程创建

CREATE

三种方式

PART TWO



Thread

Threading



构造器

Thread()

Thread(String name)

Thread(Runnable target)

Thread(Runnable target, String name)

Thread(ThreadGroup group, Runnable target)

Runnable
target

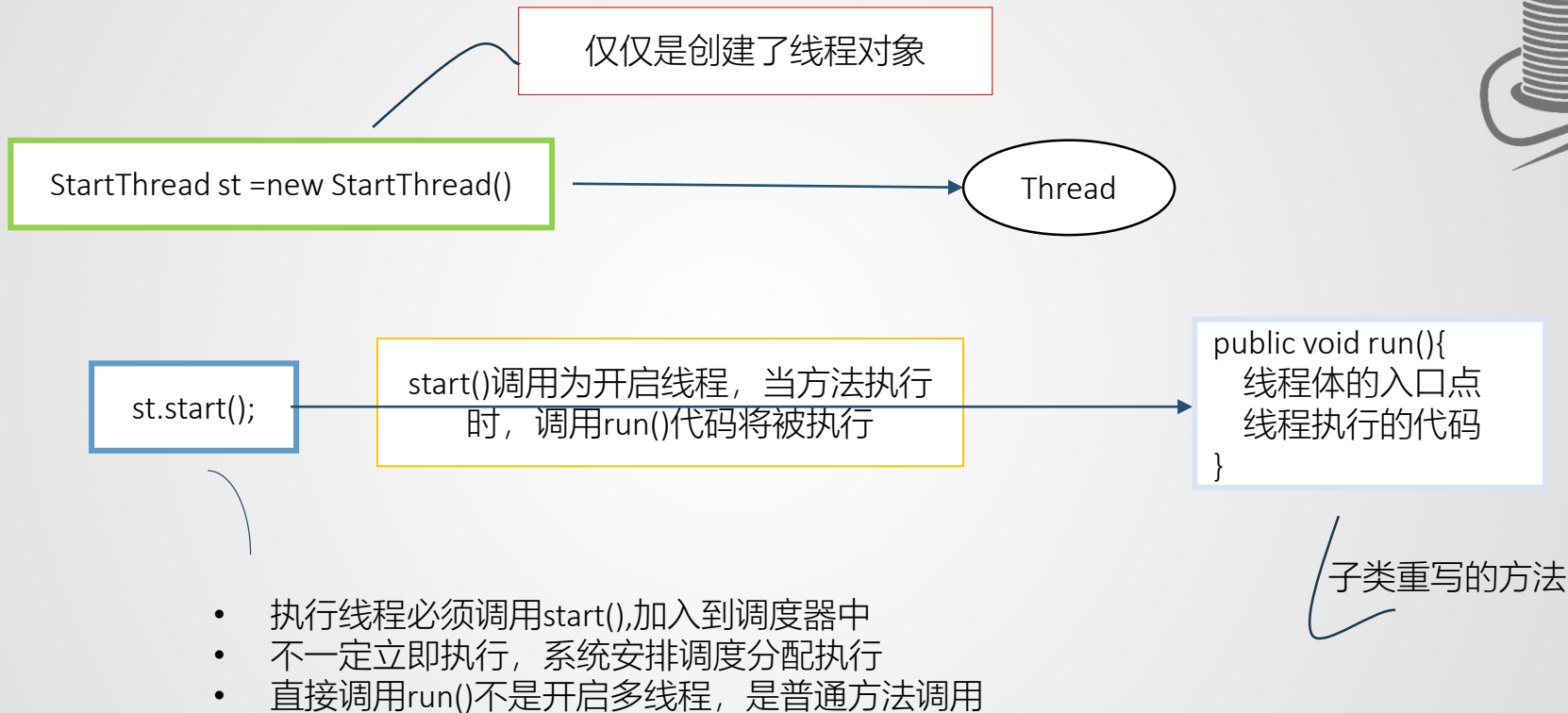
具有线程能力的
目标对象

ThreadGroup
group

此线程所在的组

String name

一个线程的命名



继承Thread

Threading



继承Thread

```
class TDownloader extends Thread{  
    private String url;  
    private String fname;  
    public TDownloader(String url,String fname) {  
        this.url = url;  
        this.fname = fname;  
    }  
    public void run() {  
        WebDownloader downloader = new WebDownloader();  
        downloader.download(url, fname);  
    }  
}
```

```
class WebDownloader {  
    public void download(String url,String name) {  
        try {  
            FileUtils.copyURLToFile(new URL(url), new File(name));  
        } catch (MalformedURLException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
TDownloader td1 = new TDownloader("x", "x.jpg");
```

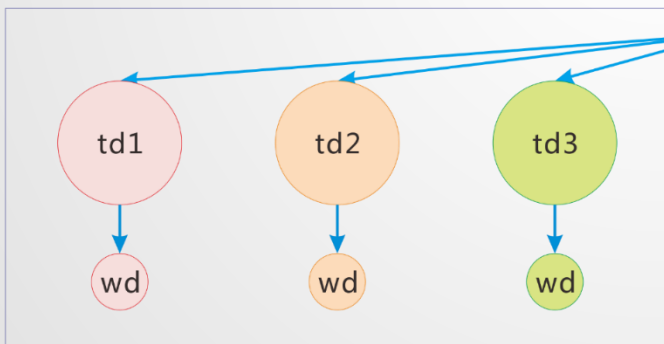
```
TDownloader td2 = new TDownloader("m", "m.jpg");
```

```
TDownloader td3 = new TDownloader("b", "baidu.png");
```

```
td1.start();
```

```
td2.start();
```

```
td3.start();
```





实现Runnable

```
class IDownloader implements Runnable{
    private String url;
    private String fname;
    public IDownloader(String url,String fname) {
        this.url = url;
        this.fname = fname;
    }
    public void run() {
        WebDownloader downloader =new WebDownloader();
        downloader.download(url, fname);
    }
}
```

使用

1. 创建目标对象: `IDownloader id =new IDownloader ("图片地址", "baidu.png");`
2. 创建线程对象+关联目标对象: `Thread t =new Thread(id);`
3. 启动线程: `t.start()`

实现Callable

实现Callable

```
class CDownloader implements Callable{
    private String url;
    private String fname;
    public CDownloader(String url,String fname) {
        this.url = url;
        this.fname = fname;
    }
    public Object call() throws Exception {
        WebDownloader downloader =new WebDownloader();
        downloader.download(url, fname);
        return true;
    }
}
```

使用

1. 创建目标对象: `CDownloader cd =new CDownloader("图片地址","baidu.png");`
2. 创建执行服务: `ExecutorService ser=Executors.newFixedThreadPool(1);`
3. 提交执行: `Future<Boolean> result1 =ser.submit(cd1) ;`
4. 获取结果: `boolean r1 =result1.get();`
5. 关闭服务: `ser.shutdownNow();`

对比

```
class TDownloader extends Thread{
    private String url;
    private String fname;
    public TDownloader(String url,String fname) {
        this.url = url;
        this.fname = fname;
    }
    public void run() {
        WebDownloader downloader
=new WebDownloader();
        downloader.download(url, fname);
    }
}
```

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

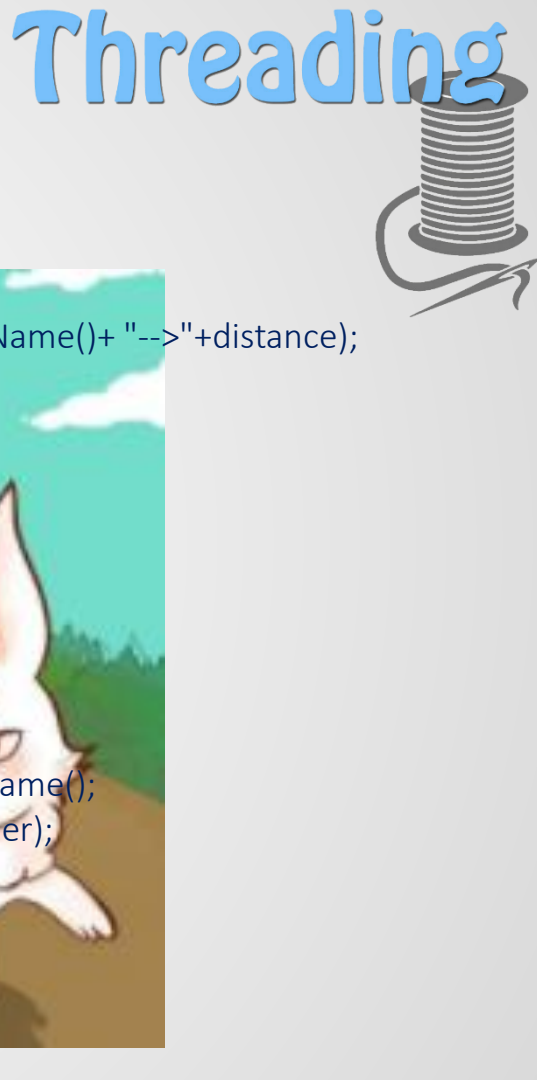
Threading



```
class IDownloader implements Runnable{
    private String url;
    private String fname;
    public IDownloader(String url,String fname) {
        this.url = url;
        this.fname = fname;
    }
    public void run() {
        WebDownloader downloader =new
WebDownloader();
        downloader.download(url, fname);
    }
}
```

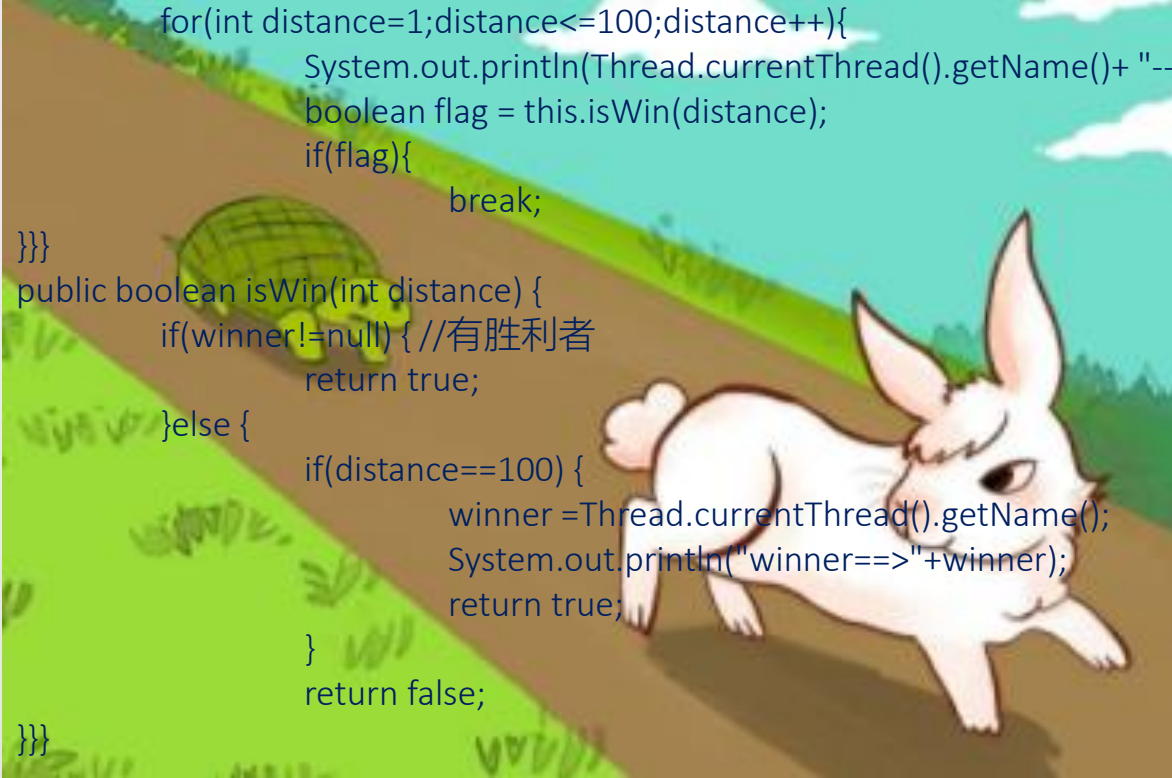
- 实现接口Runnable具有多线程能力
- 启动线程: 传入目标对象+Thread对象.start()
- 推荐使用: OOP多实现, 灵活方便,方便同一份对象的代理。

龟兔赛跑



```
public class Racer implements Runnable{  
    public static String winner;  
    public void run() {
```

```
        for(int distance=1;distance<=100;distance++){  
            System.out.println(Thread.currentThread().getName()+ "-->" +distance);  
            boolean flag = this.isWin(distance);  
            if(flag){  
                break;  
            }  
        }  
        public boolean isWin(int distance) {  
            if(winner!=null) { //有胜利者  
                return true;  
            }else {  
                if(distance==100) {  
                    winner=Thread.currentThread().getName();  
                    System.out.println("winner==>" +winner);  
                    return true;  
                }  
                return false;  
            }  
        }  
    }  
}
```





- Happy你 :真实角色
- 婚庆公司:代理角色,帮你搞婚庆
- 结婚礼仪:实现相同的接口

lamda

- 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  
(params) -> statement  
(params) -> { statements }
```

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





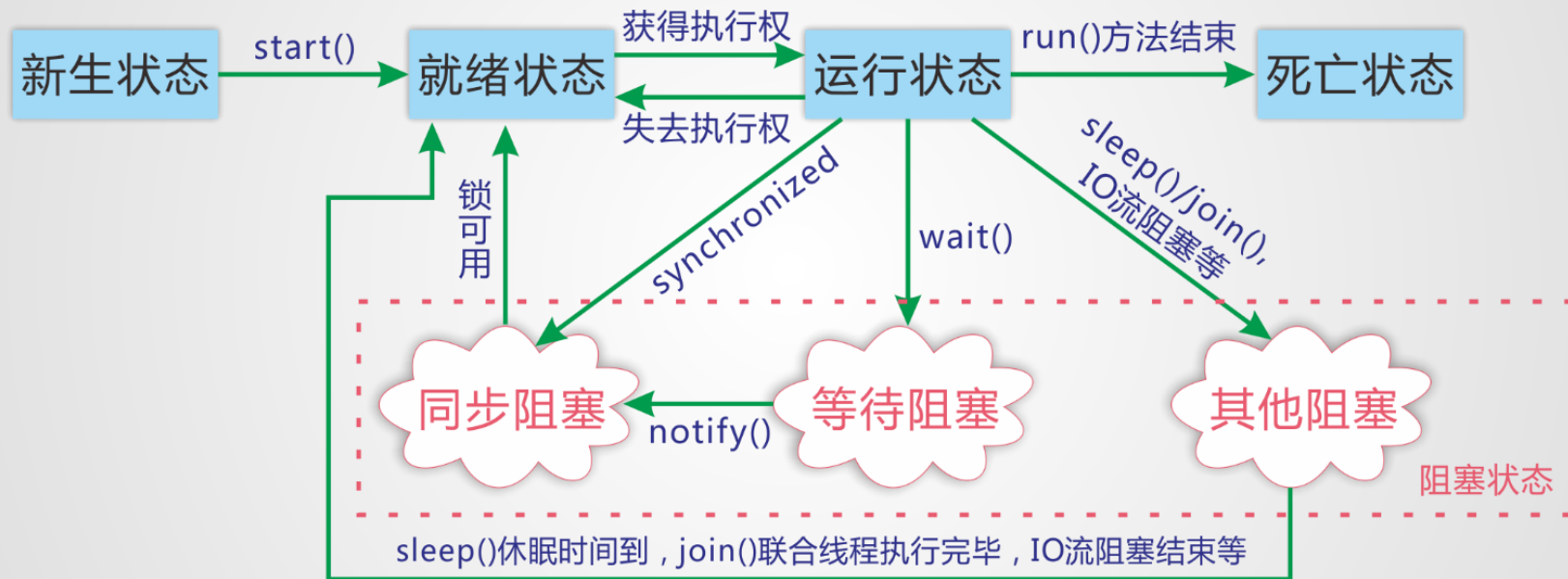
线程状态

STATUS

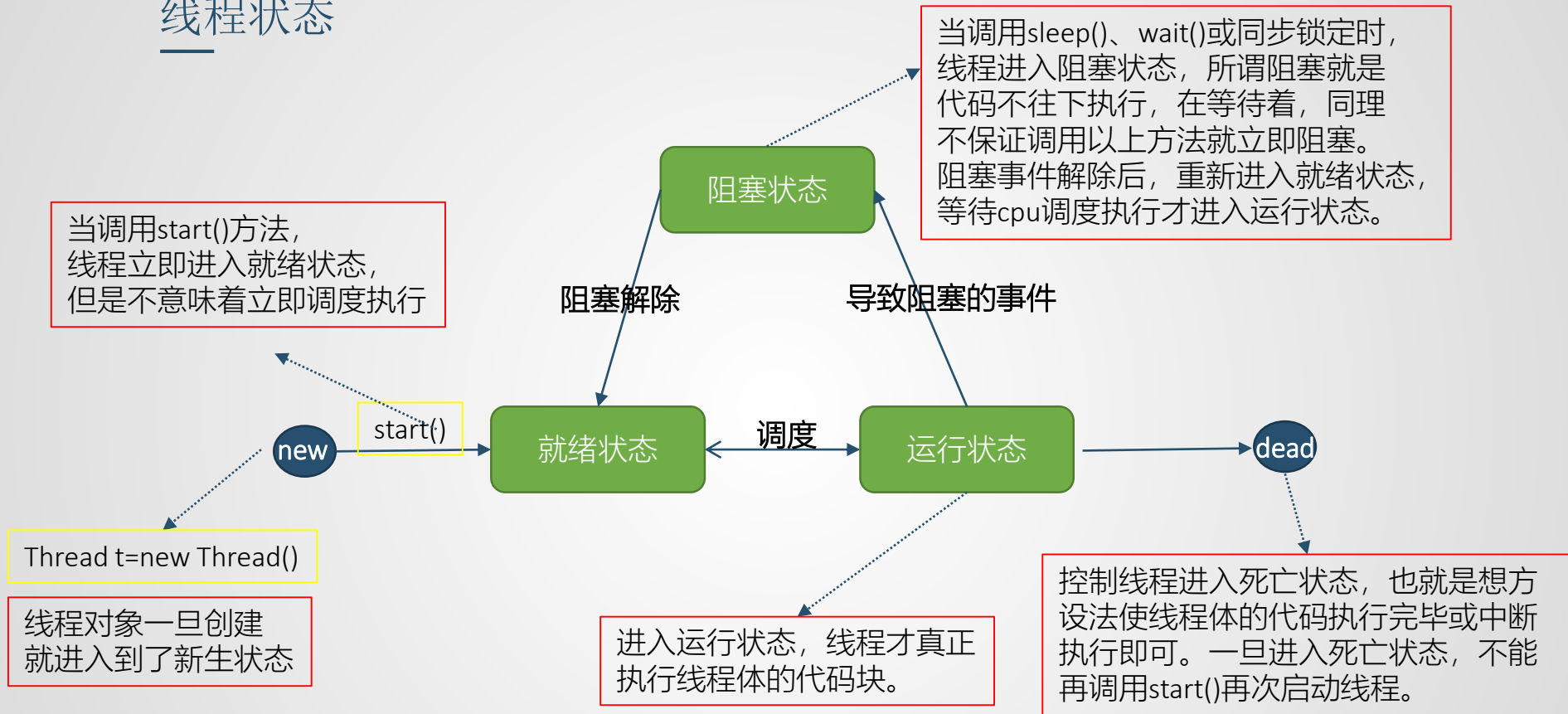
五大状态

PART THREE

线程状态



线程状态



线程方法

- 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！
- join ()
 - **阻塞**指定线程等到另一个线程完成以后再继续执行。
- 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**；
 - 调用了yield方法之后，如果没有其他等待执行的线程，**此时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，**守护线程**；
 - 创建用户线程的线程结束时，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
- setPriority(int newPriority) getPriority()
 - 线程的优先级代表的是概率
 - 范围从1到10，默认为5
- stop() 停止线程
 - **不推荐使用**

线程停止

- 不使用JDK提供的stop()/destroy()方法(它们本身也被JDK废弃了)。
- 提供一个boolean型的终止变量，当这个变量置为false，则终止线程的运行。

```
class Study implements Runnable{  
    //1)、线程类中 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("study thread...");  
        }  
    }  
    //3)、对外提供方法改变标识  
    public void stop() {  
        this.flag =false;  
    }  
}
```



sleep

- 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;



```
class Web12306 implements Runnable {  
    private int num =50;  
    public void run() {  
        while(true){  
            if(num<=0){break; }  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
System.out.println(Thread.currentThread().getName()+"抢到了"+num--);
```

```
}}}
```



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**



```
public class JoinDemo01 extends Thread {  
    public static void main(String[] args) {  
        JoinDemo01 demo = new JoinDemo01();  
        Thread t = new Thread(demo);  
        t.start();  
        for(int i=0;i<1000;i++) {  
            if(50==i) {  
                t.join(); //main阻塞...  
            }  
            System.out.println("main...."+i)  
        }  
    }  
    public void run() {  
        for(int i=0;i<1000;i++) {  
            System.out.println("join...."+i)  
        }  
    }  
}
```

yield

- 礼让线程, 让当前正在执行线程暂停
- 不是阻塞线程, 而是将线程从**运行状态**转入**就绪状态**
- 让cpu调度器重新调度

```
public class YieldDemo01 extends Thread {
    public static void main(String[] args) {
        YieldDemo01 demo = new YieldDemo01();
        Thread t = new Thread(demo);
        t.start();
        for(int i=0;i<1000;i++) {
            if(i%20==0) {
                //暂停本线程 main
                Thread.yield();
            }
            System.out.println("main...."+i);
        }
    }
    public void run() {
        for(int i=0;i<1000;i++) {
            System.out.println("yield...."+i);
        }
    }
}
```

priority

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);`



优先级的设定建议在`start()`调用前

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。

daemon



- 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、监控内存使用等。

常用其他方法

方法	功能
isAlive()	判断线程是否还活着,即线程是否还未终止
setName()	给线程起一个名字
getName()	获取线程的名字
currentThread()	取得当前正在运行的线程对象,也就是获取自己本身



线程同步

synchronized

并发控制

PART FOUR

并发

并发: 同一个对象多个线程同时操作



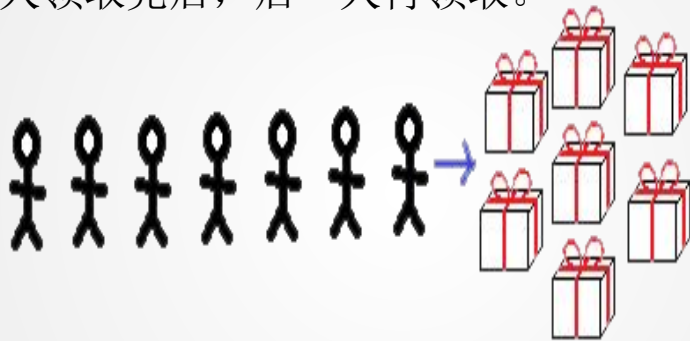
同时操作统一账户

同时购买同一车次的票



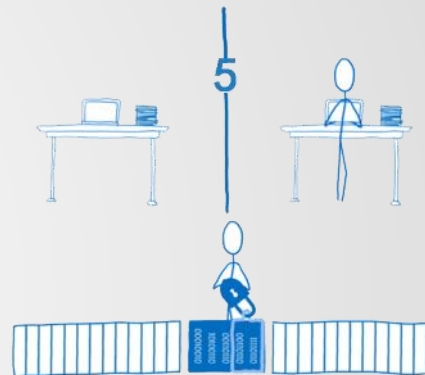
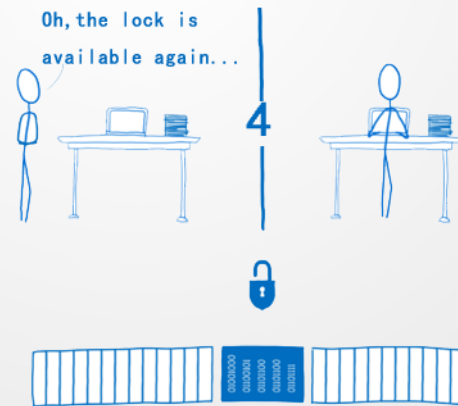
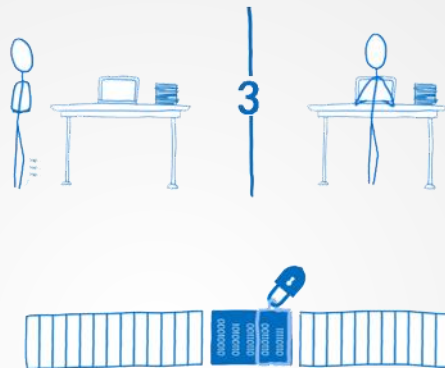
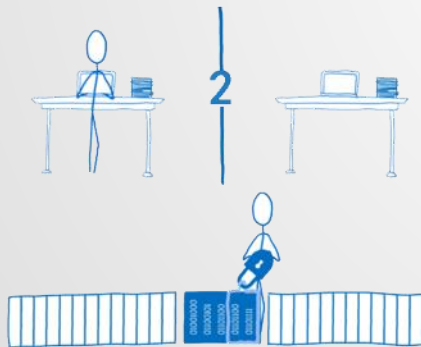
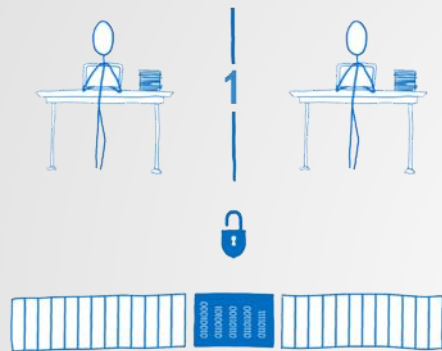
线程同步

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：派发礼品，多个人都想获得。天然的解决办法就是，在礼品前，大家排队。前一人领取完后，后一人再领取。



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，我们就需要用到“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**，等待前面的线程使用完毕后，下一个线程再使用。

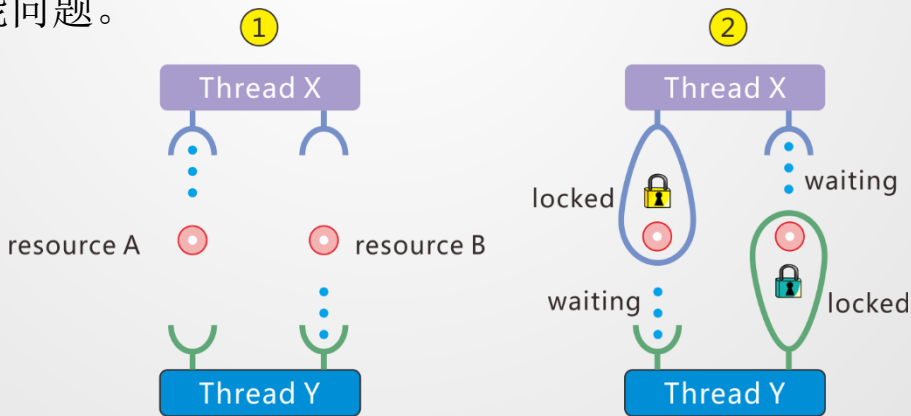
线程同步



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，也带来了访问冲突的问题。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它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。



线程同步

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“成员变量|类变量”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方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。

缺陷：若将一个大的方法声明为 `synchronized` 将会大大影响效率。

线程同步

- **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是this即该对象本身，或class即类的模子
- 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

死锁

死锁: 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进行，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。



小丫占镜子大丫拿口红





线程协作

cooperation

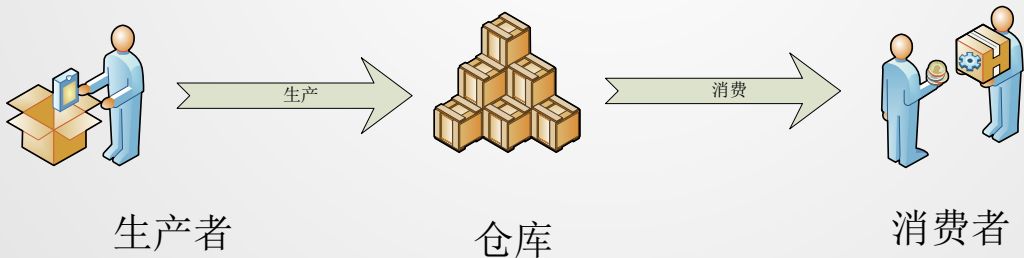
生产者消费者模式

PART FIVE

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。



线程通信

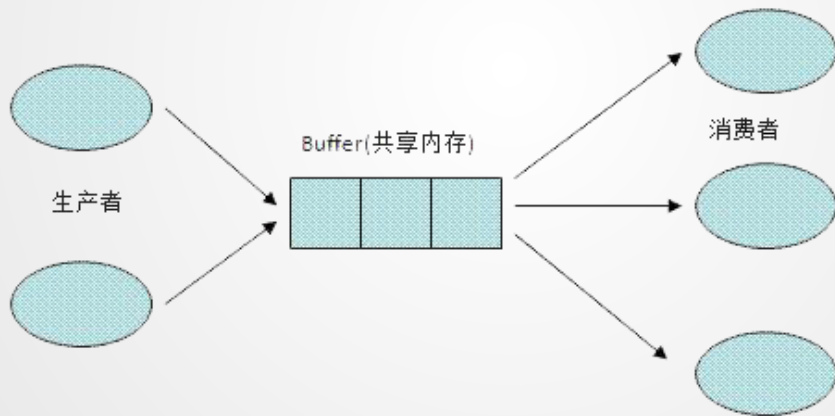
分析:这是一个线程同步问题，生产者和消费者共享同一个资源，并且生产者和消费者之间相互依赖，互为条件

- 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）

线程通信

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(这里模块可能是:方法、对象、线程、进程);
 - 消费者:负责处理数据的模块(这里模块可能是: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,它们之间有个“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”拿要处理的数据。



解耦、提高效率

线程通信

解决方式2: 并发协作模型 “生产者/消费者模式” → 信号灯法



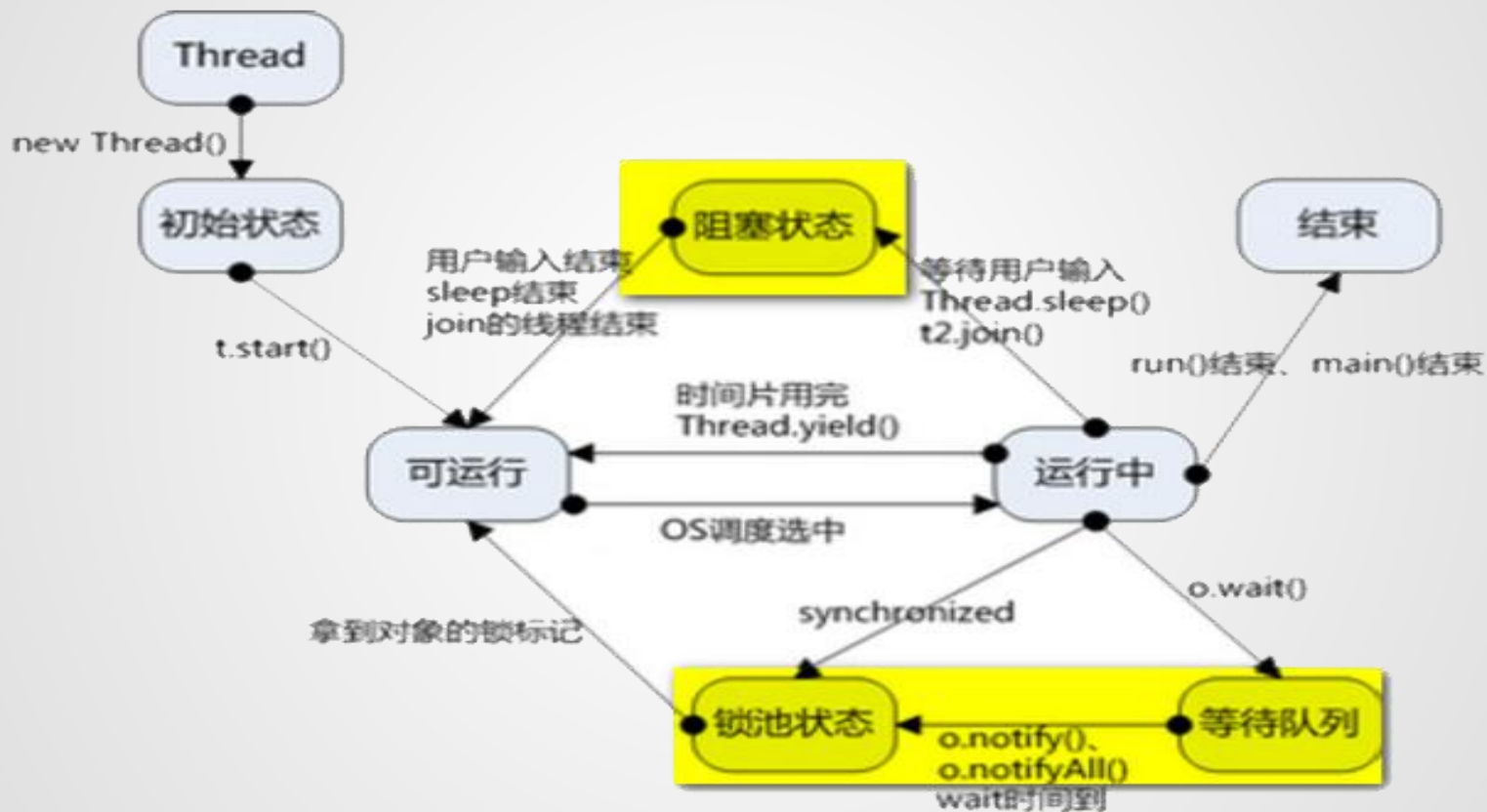
线程通信

Java提供了3个方法解决线程之间的通信问题

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

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

生命周期





高级主题

more

更上一层楼

PART SIX

任务定时调度

某一个有规律的时间点干某件事：

- 每天早上8点，闹钟响起
- 每年4月1日自己给当年暗恋女神发一封匿名贺卡
- 想每隔1小时，上传一下自己的学习笔记到云盘

通过Timer和Timetask，我们可以实现定时启动某个线程。

- **java.util.Timer**:类似闹钟的功能，本身实现的就是一个线程
- **java.util.TimerTask**:一个抽象类，该类实现了Runnable接口，所以该类具备多线程的能力。

```
class TestTimer {
    public static void main(String[] args) {
        Timer t1 = new Timer(); // 定义计时器;
        MyTask task1 = new MyTask(); // 定义任务;
        t1.schedule(task1, 3000); // 3秒后执行;
        // t1.schedule(task1, 5000, 1000); // 5秒以后每隔1
        秒执行一次!
        //GregorianCalendar calendar1 = new
        GregorianCalendar(2010, 0, 5, 14, 36, 57);
        //t1.schedule(task1, calendar1.getTime()); //
        指定时间定时执行;
    }
}

class MyTask extends TimerTask { // 自定义线程类继承
    TimerTask类;
    public void run() {
        for(int i=0; i<10; i++) {
            System.out.println("任务1:" + i);
        }
    }
}
```

任务定时调度



Scheduler - 调度器，控制所有的调度

Trigger - 触发条件，采用DSL模式

JobDetail - 需要处理的JOB

Job - 执行逻辑

DSL: Domain-specific language 领域特定语言，针对一个特定的领域，具有受限表达性的一种计算机程序语言，即领域专用语言，声明式编程：

- 1. Method Chaining **方法链** Fluent Style 流畅风格，builder 模式构建器
- 2. Nested Functions 嵌套函数
- 3. Lambda Expressions/Closures
- 4. Functional Sequence

简洁 连贯

HappenBefore

- 你写的代码很可能根本没按你期望的顺序执行，因为编译器和 CPU 会尝试重排指令使得代码更快地运行

```
subTotal = price + fee;  
total += subTotal;  
isDone = true;
```



```
ADD R1, R2 → R3  
ADD R4, R3 → R4  
MOV 1 → R5
```



HappenBefore

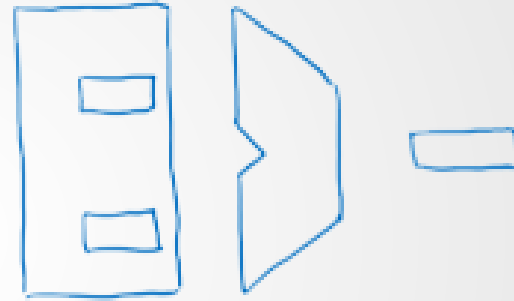
Instructions	
1	ADD R1, R2 → R3
2	ADD R4, R3 → R4
3	MOV 1 → R5

Registers	
price	
fee	
subTotal	
total	
isDone	

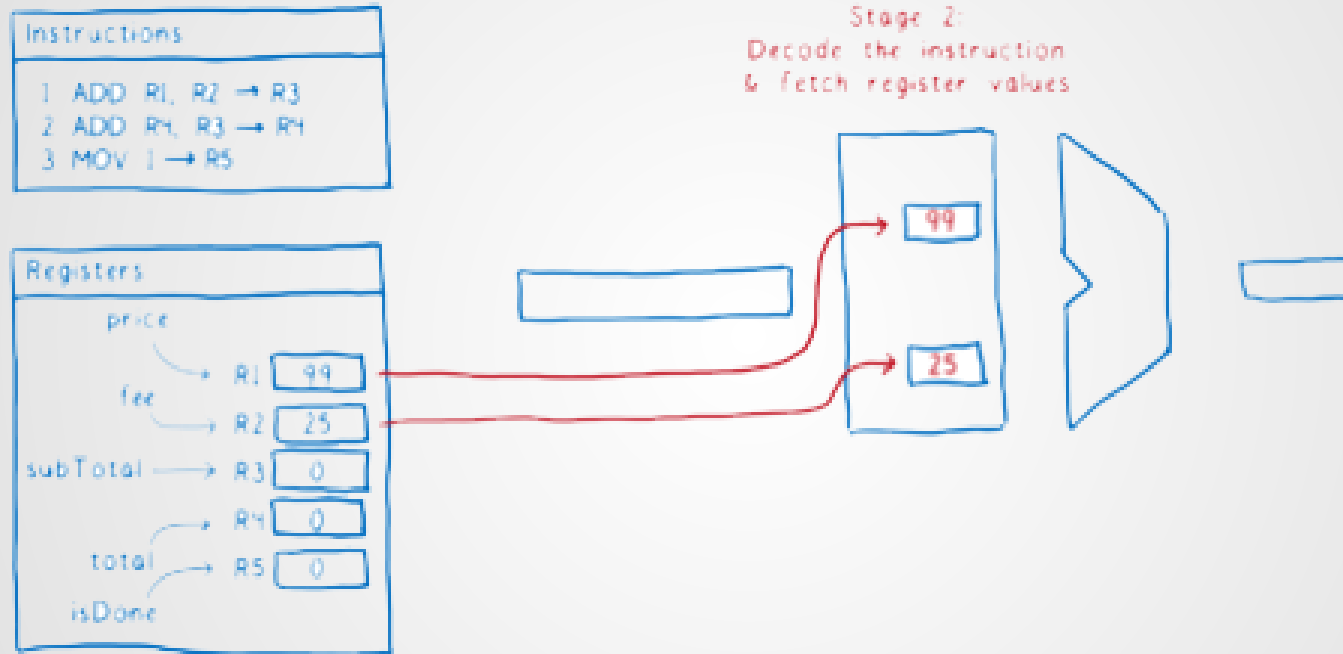
R1	95
R2	25
R3	0
R4	0
R5	0

Stage 1:
Fetch the instruction

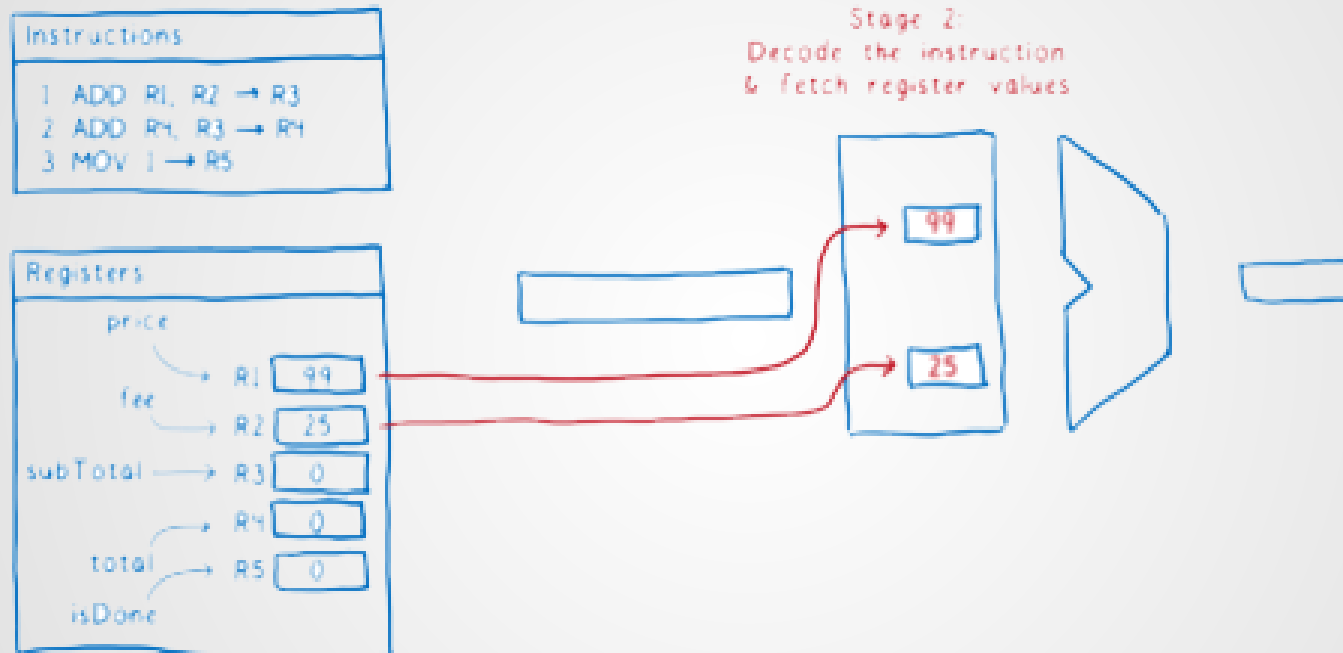
ADD R1, R2 → R3



HappenBefore



HappenBefore

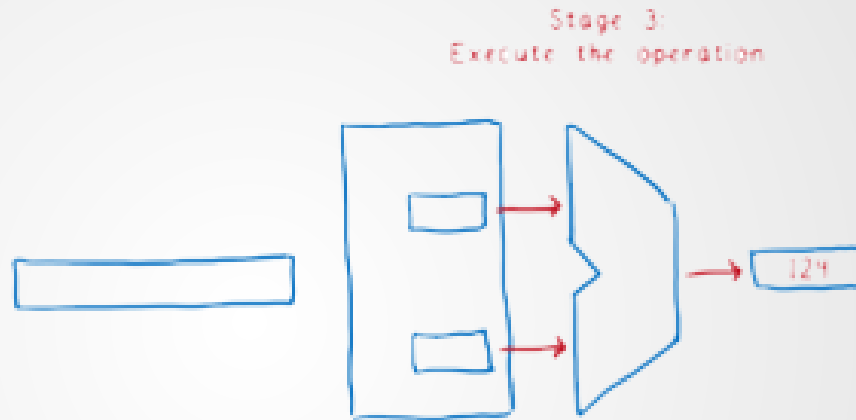


HappenBefore

Instructions	
1	ADD R1, R2 → R3
2	ADD R4, R3 → R4
3	MOV 1 → R5

Registers	
price	
fee	
subTotal	
total	
isDone	

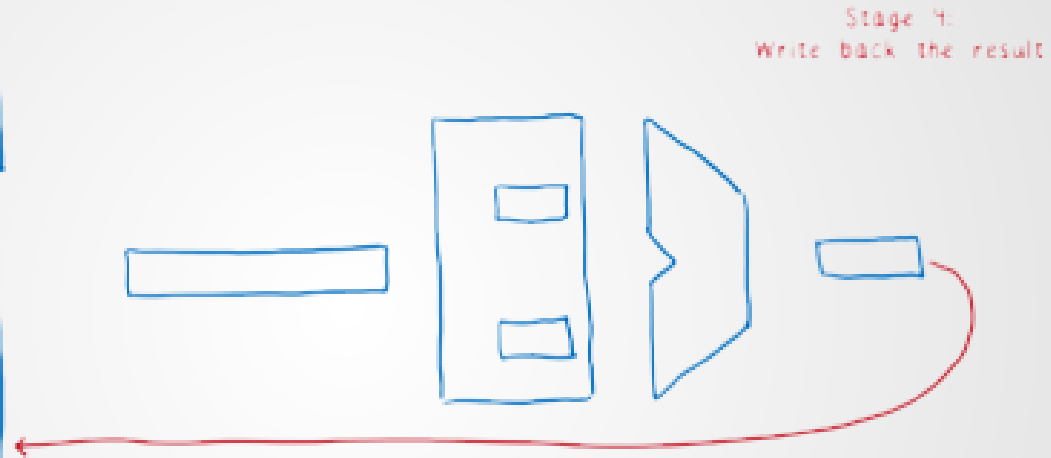
R1	99
R2	25
R3	0
R4	0
R5	0



HappenBefore

Instructions	
1	ADD R1, R2 → R3
2	ADD R1, R3 → R1
3	MOV 1 → R5

Registers	
price	
fee	→ R1 99
	→ R2 25
subTotal	→ R3 124
	→ R4 0
total	→ R5 0
isDone	



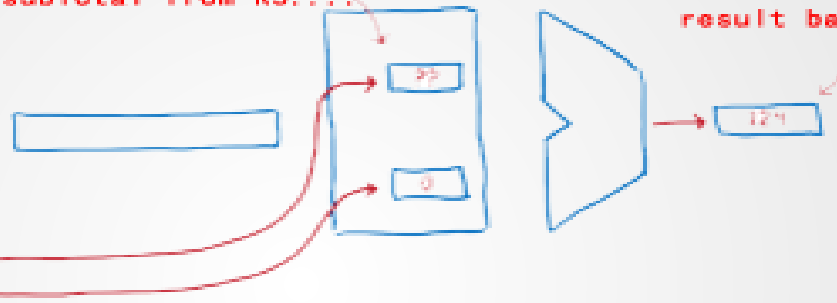
HappenBefore

Instructions	
1	ADD R1, R2 → R3
2	ADD R4, R3 → R4
3	MOV 1 → R5

Registers	
price	→ R1
fee	→ R2
subTotal	→ R3
	→ R4
total	→ R5
isDone	

Instruction 2 needs to
fetch the value for
subTotal from R3....

...but instruction 1
has not written its
result back yet...



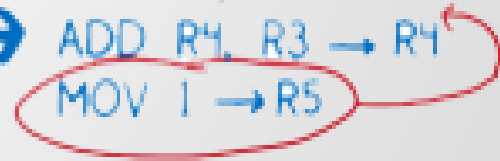
subTotal = price + fee;
total += subTotal;
isDone = true;



ADD R1, R2 → R3
ADD R4, R3 → R4
MOV 1 → R5



ADD R1, R2 → R3
ADD R4, R3 → R4
MOV 1 → R5



HappenBefore

执行代码的顺序可能与编写代码不一致，即虚拟机优化代码顺序，则为指令重排 **happen-before**即:编译器或运行时环境为了**优化程序性能**而采取的对指令进行重新排序执行的一种手段。

- 在虚拟机层面，为了尽可能减少内存操作速度远慢于CPU运行速度所带来的CPU空置的影响，虚拟机会按照自己的一些规则(这规则后面再叙述)将程序编写顺序打乱——即写在后面的代码在时间顺序上可能会先执行，而写在前面的代码会后执行——以尽可能充分地利用CPU。拿上面的例子来说：假如不是a=1的操作，而是a=new byte[1024*1024](分配1M空间)，那么它会运行地很慢，此时CPU是等待其执行结束呢，还是先执行下面那句flag=true呢？显然，先执行flag=true可以提前使用CPU，加快整体效率，当然这样的前提是不会产生错误(什么样的错误后面再说)。虽然这里有两种情况：后面的代码先于前面的代码开始执行；前面的代码先开始执行，但当效率较慢的时候，后面的代码开始执行并先于前面的代码执行结束。不管谁先开始，总之后面的代码在一些情况下存在先结束的可能。
- 在硬件层面，CPU会将接收到的一批指令按照其规则重排序，同样是基于CPU速度比缓存速度快的原因，和上一点的目的类似，只是硬件处理的话，每次只能在接收到的有限指令范围内重排序，而虚拟机可以在更大层面、更多指令范围内重排序。

数据依赖

如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖。数据依赖分下列三种类型：

名称	代码示例	说明
写后读	<code>a = 1; b = a;</code>	写一个变量之后，再读这个位置。
写后写	<code>a = 1; a = 2;</code>	写一个变量之后，再写这个变量。
读后写	<code>a = b; b = 1;</code>	读一个变量之后，再写这个变量。

上面三种情况，只要重排序两个操作的执行顺序，程序的执行结果将会被改变。所以，编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。

volatile

volatile保证线程间变量的**可见性**，简单地说就是当线程A对变量X进行了修改后，在线程A后面执行的其他线程能看到变量X的变动，更详细地说是要符合以下两个规则：

- 线程对变量进行修改之后，要立刻回写到主内存。
- 线程对变量读取的时候，要从主内存中读，而不是缓存。



各线程的工作内存间彼此独立、互不可见，在线程启动的时候，虚拟机为每个内存分配一块工作内存，不仅包含了线程内部定义的局部变量，也包含了线程所需要使用的共享变量（非线程内构造的对象）的副本，即为了提高执行效率。

volatile是不错的机制，但是**volatile**不能保证原子性。

单例模式

- double-checking
- volatile



ThreadLocal

- 在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程。
- ThreadLocal能够放一个线程级别的变量，其本身能够被多个线程共享使用，并且又能够达到线程安全的目的。说白了，ThreadLocal就是想在多线程环境下去保证成员变量的安全，常用的方法，就是 `get/set/initialValue` 方法。
- JDK建议ThreadLocal定义为 `private static`
- ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的方法都可以非常方便地访问这些资源。
 - Hibernate的Session 工具类HibernateUtil
 - 通过不同的线程对象设置Bean属性，保证各个线程Bean对象的独立性。

可重入锁

锁作为并发共享数据保证一致性的工具，大多数内置锁都是可重入的，也就是说，如果某个线程试图获取一个已经由它自己持有的锁时，那么这个请求会立刻成功，并且会将这个锁的计数值加1，而当线程退出同步代码块时，计数器将会递减，当计数值等于0时，锁释放。如果没有可重入锁的支持，在第二次企图获得锁时将会进入死锁状态。可重入锁随处可见：

```
// 第一次获得锁
synchronized(this) {
    while(true) {
        // 第二次获得同样的锁
        synchronized(this) {
            System.out.println("ReentrantLock!");
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
class ReentrantLockTest {
    public synchronized void a() {
    }
    public synchronized void b() {
    }
    /**
     * 很明显的可重入锁用法
     */
    public synchronized void all() {
        this.a(); //此时对象的锁计数值已经达到2了
        this.b();
    }
}
```


CAS

锁分为两类:

- 悲观锁:synchronized是独占锁即悲观锁, 会导致其它所有需要锁的线程挂起, 等待持有锁的线程释放锁。
- 乐观锁:每次不加锁而是假设没有冲突而去完成某项操作, 如果因为冲突失败就重试, 直到成功为止。

Compare and Swap 比较并交换:

- 乐观锁的实现;
- 有三个值:一个当前内存值V、旧的预期值A、将更新的值B。先获取到内存当中当前的内存值V, 再将内存值V和原值A作比较, 要是相等就修改为要修改的值B并返回true, 否则什么都不做, 并返回false;
- CAS是一组原子操作, 不会被外部打断;
- 属于硬件级别的操作(利用CPU的CAS指令, 同时借助JNI来完成的非阻塞算法), 效率比加锁操作高。
- ABA问题:如果变量V初次读取的时候是A, 并且在准备赋值的时候检查到它仍然是A, 那能说明它的值没有被其他线程修改过了吗? 如果在这段期间曾经被改成B, 然后又改回A, 那CAS操作就会误认为它从来没有被修改过。

更多高级主题

java.util.concurrent

总结

感谢您的支持与信任

THANK YOU FOR WATCHING