



# 第7章 多线程

主讲人：丛小茗



# 涉及到课本章节：

- 第7章 多线程



# 提纲：

- **7.0 CPU及进程**
- **7.1 多线程**
- **7.2 Java线程使用**
- **7.3 线程同步**
- **7.4 线程池**

# 核酸检测多线程场景

- 单队检测
- 多队检测
- 忘带证件回家取





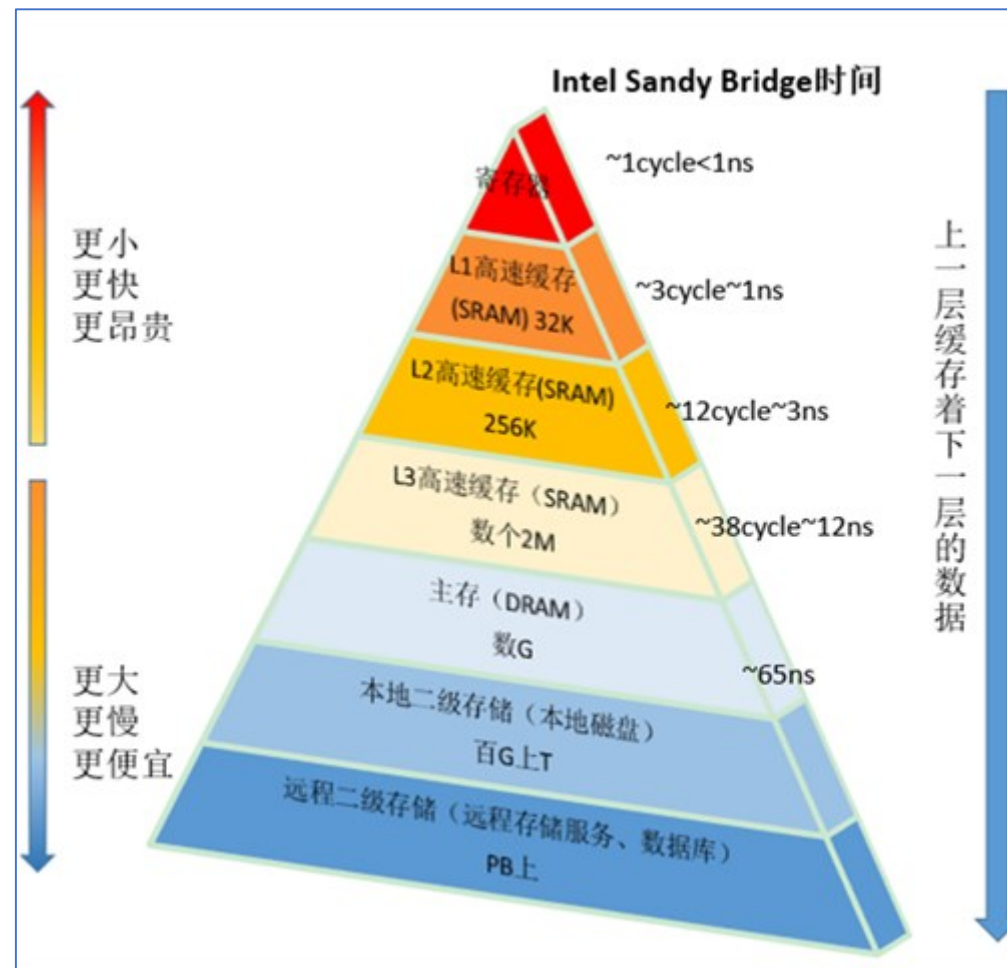
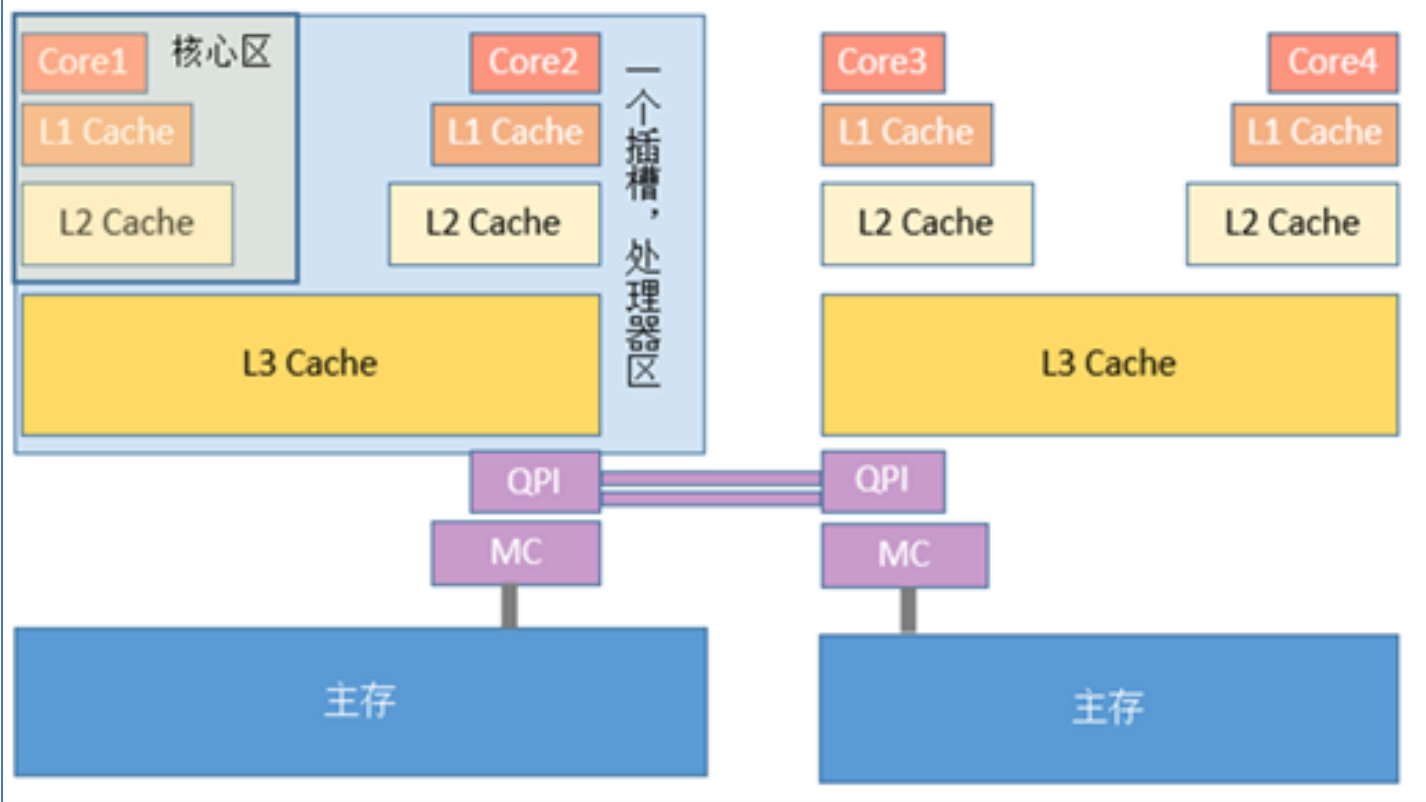
# 进程的产生及CPU空闲

- 单道系统，纸带
  - 单道批处理系统，磁带
  - 多道批处理系统
  - 分时系统
  - 多任务操作系统
- 
- 进程是60年代初首先由麻省理工学院的MULTICS系统和IBM公司的CTSS/360系统引入的。



# 两处理器四核心CPU结构、数据读取时间

Intel Sandy Bridge 两个处理器，四个核心的CPU结构





# CPU速度——快出天际

多进程的基本条件

CPU眼中的其他设备



其他设备眼中的CPU

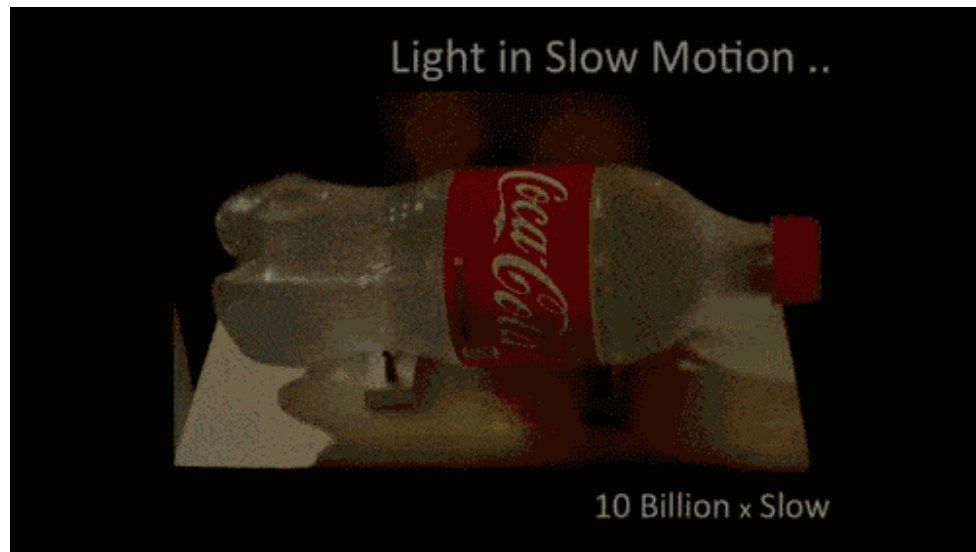
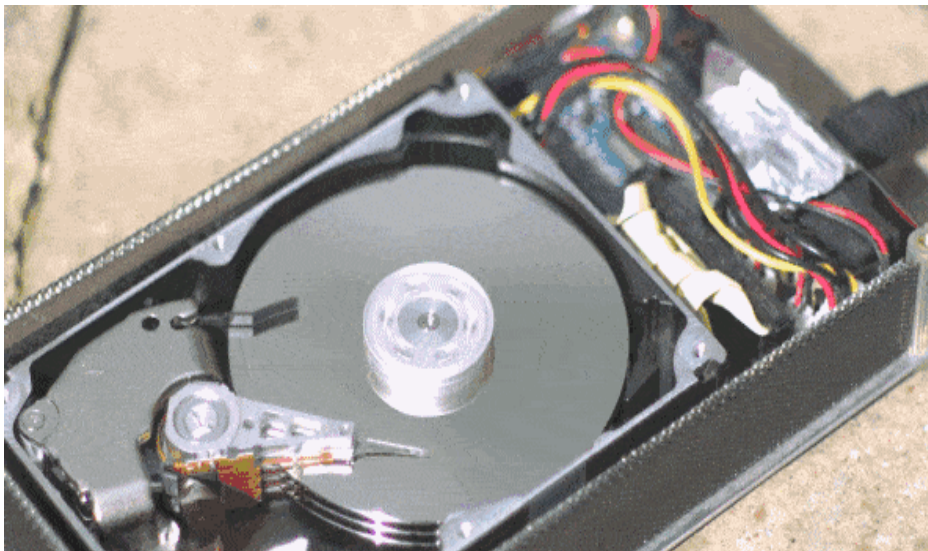
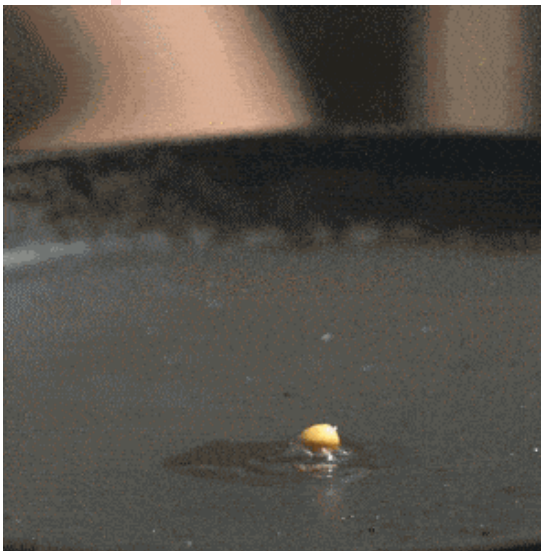


快到人类无法感知



# 放慢镜头去感知和思考

借助高速摄像机开启调试模式





# CPU定位——快者多劳

快速切换容纳更多进程

- 管理
- 报表
- 核算
- 沟通
- 进度
- 商务
- 出差



# CPU分心——半途不费

多进程轮换运行



© 东方IC

## 分心而不偏心

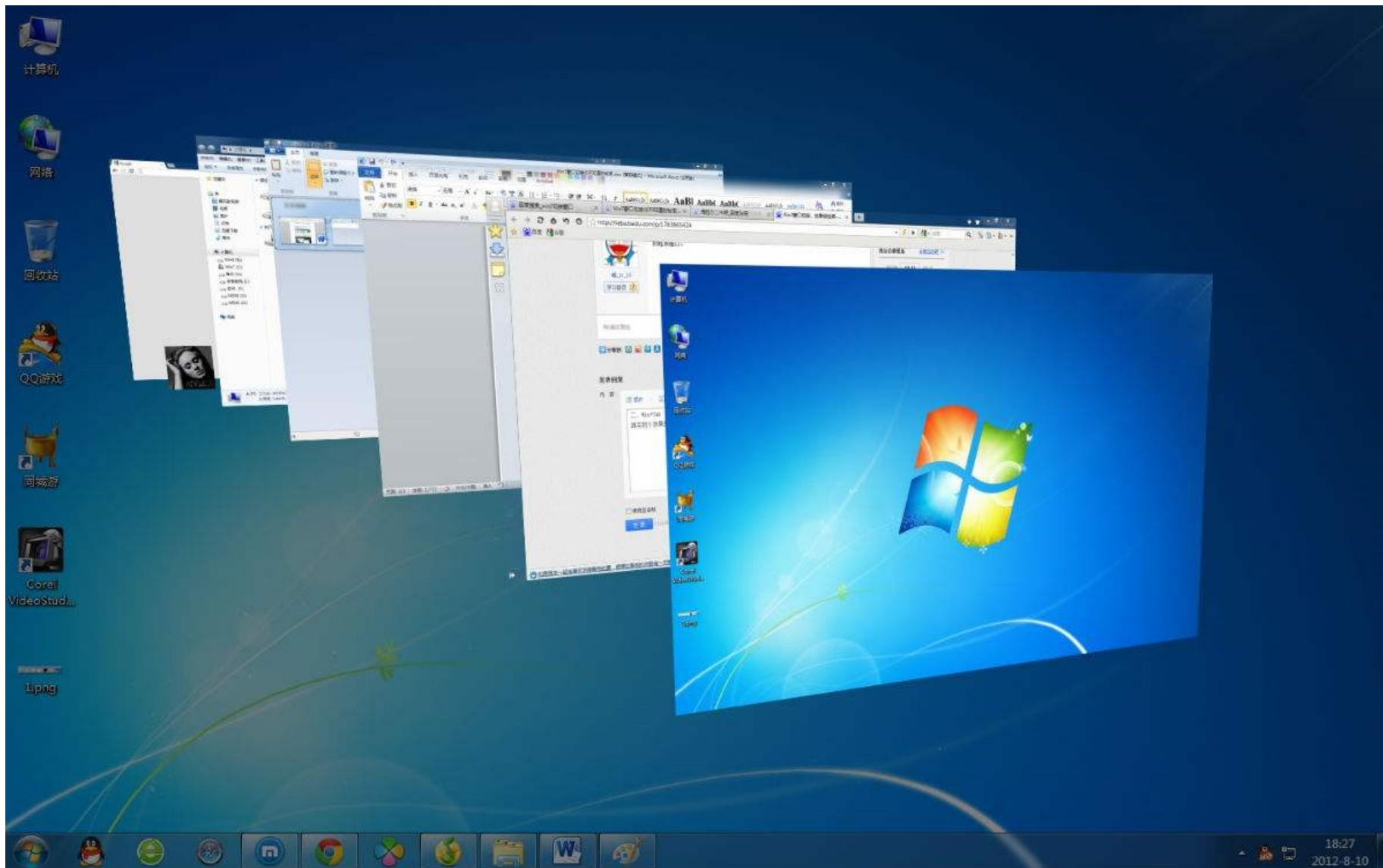
- 先进先出算法FIFO
- 短进程优先算法SCBF
- 时间片轮转调度算法RR
- 优先级算法PSA

# 进程调度超出人类感知

纳秒级↔毫秒级30ms

- 音乐
- 图片
- 文档
- 网页
- 文件

感觉同时运行  
实际快速切换







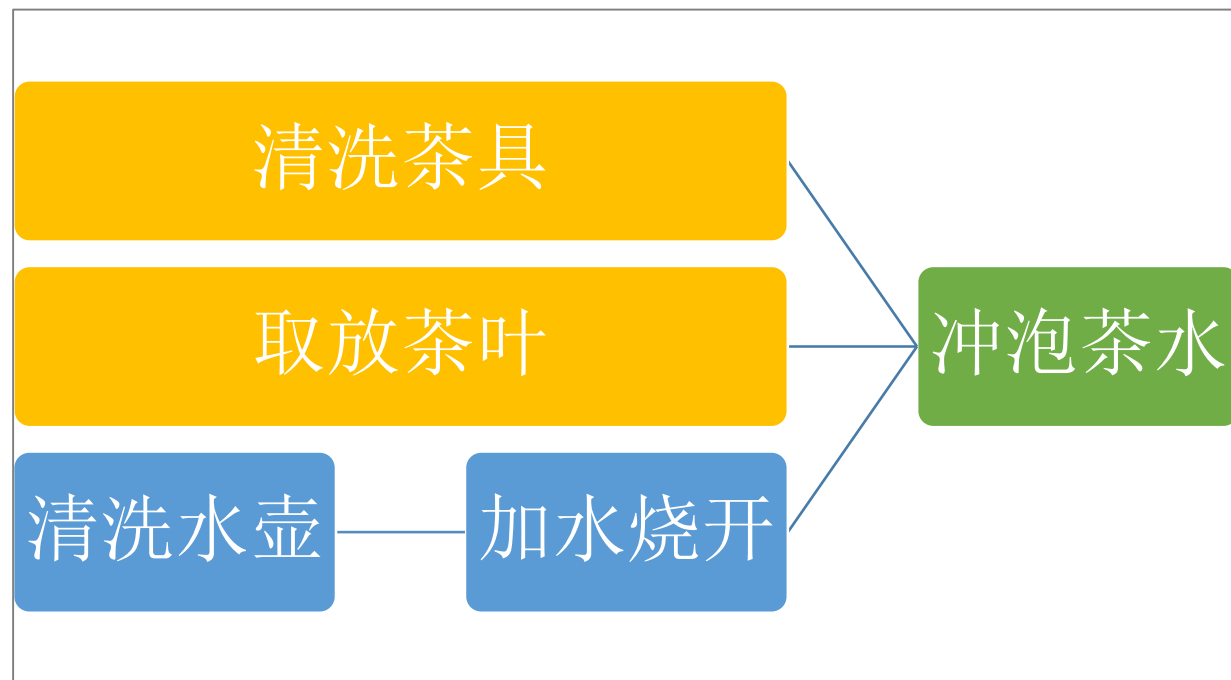
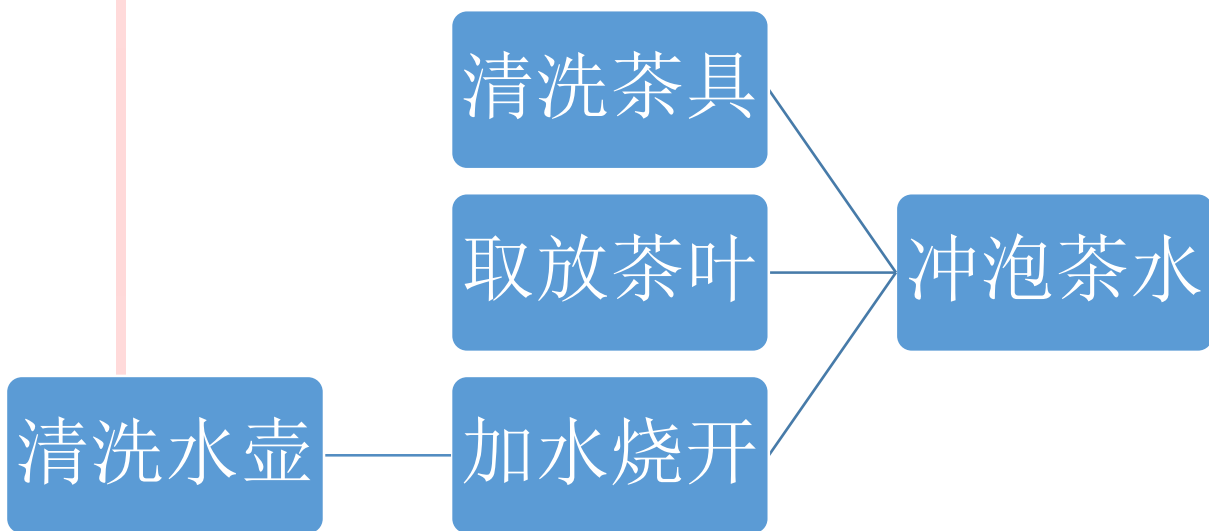
## 7.1 多线程

进程与线程：

- 进程（**process**）：是一个程序关于某个数据集合的一次执行过程，是操作系统进行资源分配和保护的基本单位
- 线程（**thread**）：是进程中能够独立执行的实体(控制流)，是处理器调度和分派的基本单位。

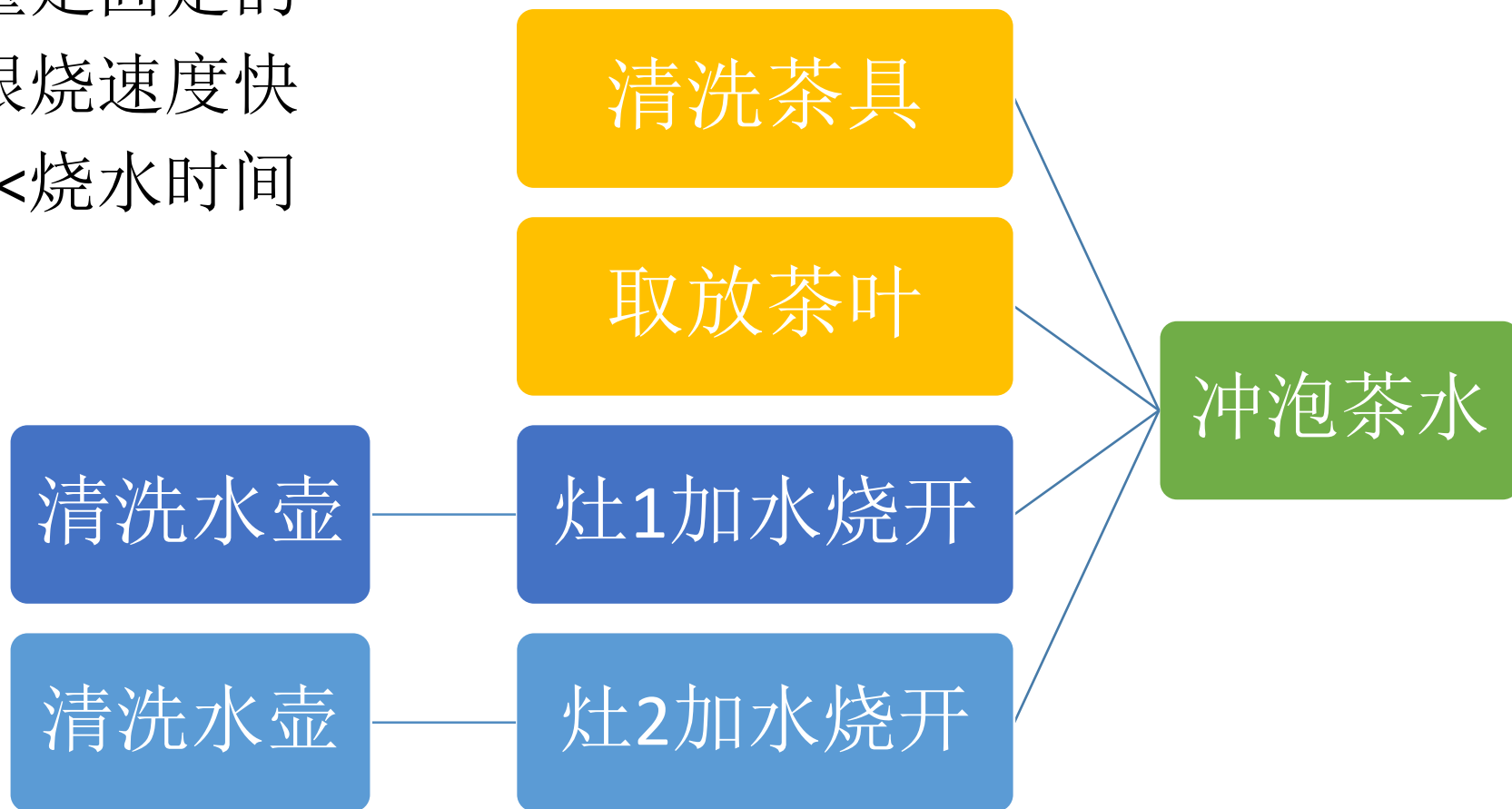
# 人类分配时间的方法——统筹方法

- 清洗水壶、加水烧开、清洗茶具、取放茶叶、冲泡茶水



# 多进程烧水

- 泡茶需要的水量是固定的
- 分两壶两个灶眼烧速度快
- 清洗水壶时间 $\ll$ 烧水时间



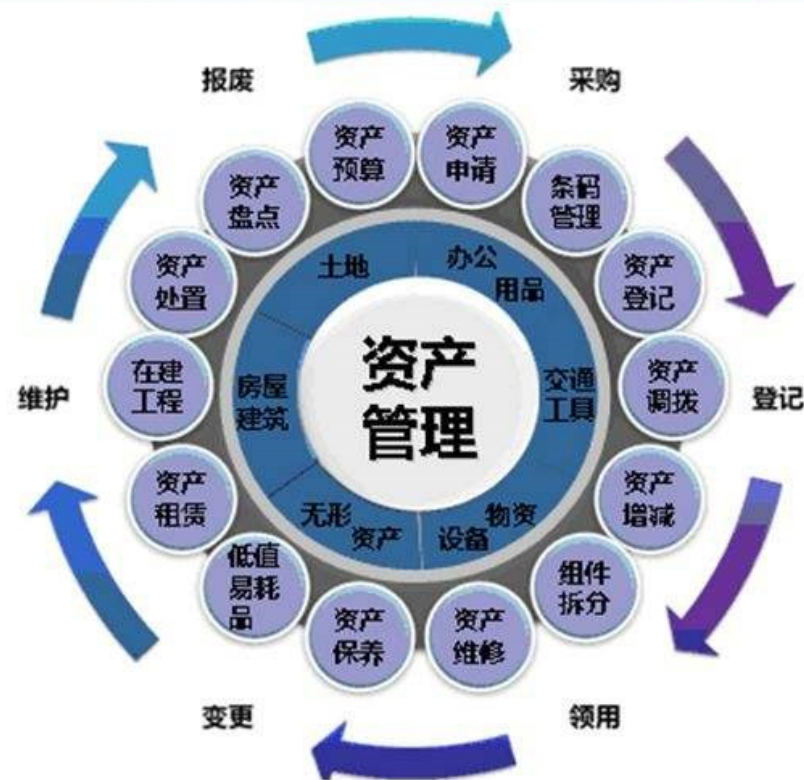
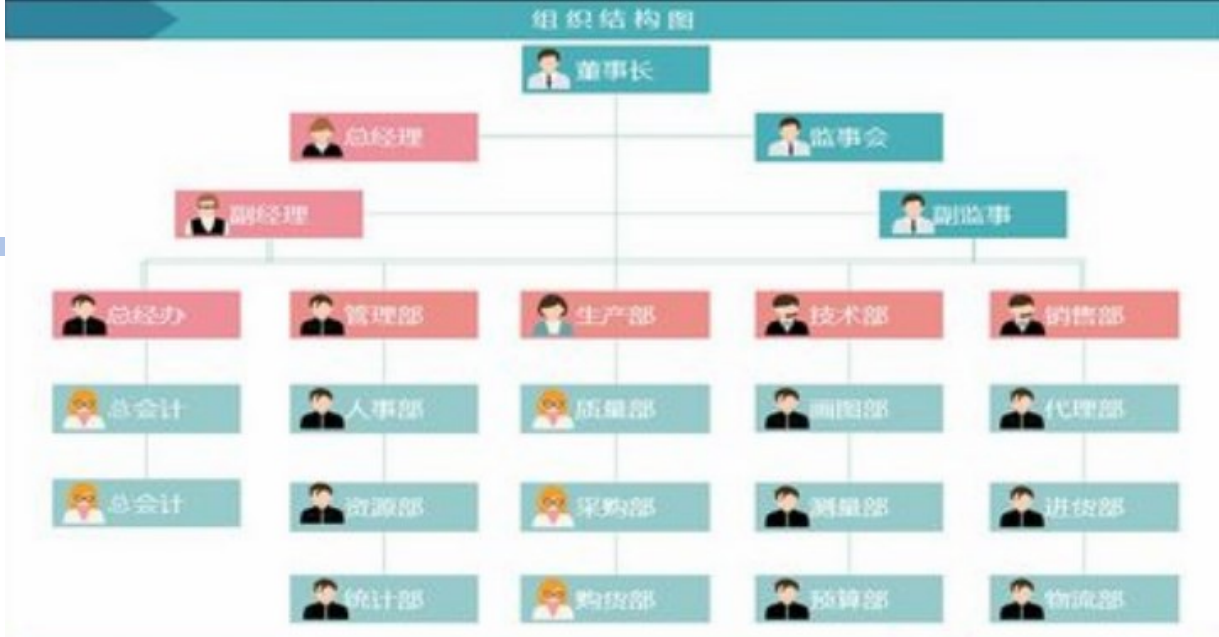


# 进程与线程

线程是轻量级的进程。



线程是进程的组成部分，每一个进程中允许包含多个并发执行的线程。

同一个进程中的所有线程共享进程获得的内存空间和资源。所以这些线程都能访问相同的变量并在同一个堆上分配对象。





# 进程、线程与CPU时间片

时间片      运行状态:       就绪状态: 



任务1




任务2



任务3

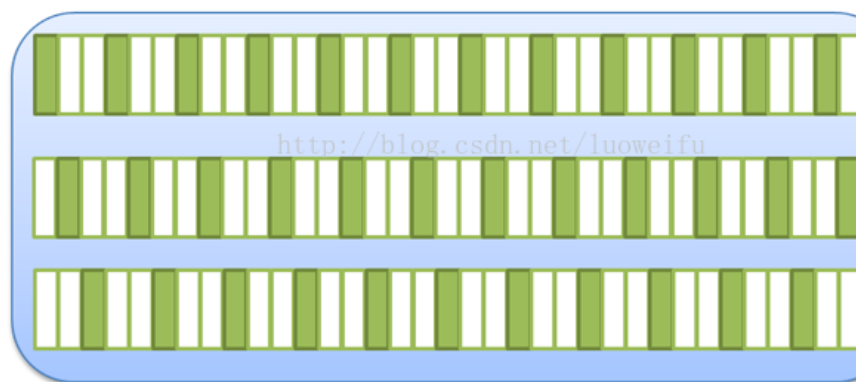
时间  时间的方向

时间片      运行状态:       就绪状态: 

单线程



多线程



线程1

线程2

线程3

时间  时间的方向



# 进程、线程→火车、车厢

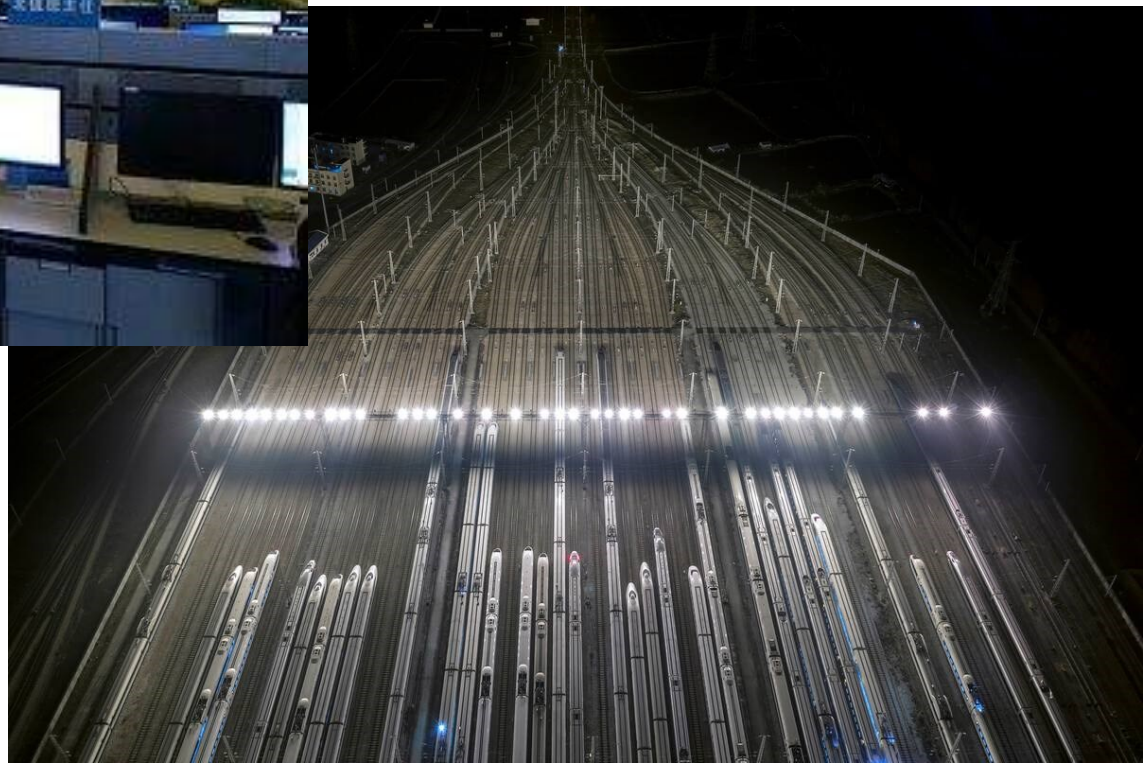


进程、线程	火车、车厢
线程在进程下行进	单纯的车厢无法运行
一个进程可以包含多个线程	一辆火车可以有多个车厢
不同进程间数据很难共享	一辆火车上的乘客很难换到另外一辆火车，比如站点换乘
同一进程下不同线程间数据很易共享	A车厢换到B车厢很容易
进程要比线程消耗更多的计算机资源	采用多列火车相比多个车厢更耗资源
进程间不会相互影响，一个线程挂掉将导致整个进程挂掉	一列火车不会影响到另外一列火车，但是如果一列火车上中间的一节车厢着火了，将影响到所有车厢
进程可以拓展到多机，进程最多适合多核	不同火车可以开在多个轨道上，同一火车的车厢不能在行进的不同的轨道上
线程可以对进程使用的内存地址上锁	比如火车上的洗手间





# 进程隔离、争抢资源→火车分道、铁路排线





# 线程的优势与风险

## 优势

- GUI界面中，线程可以提高GUI的响应灵敏度
- 服务器应用中，可以提升资源利用率和系统吞吐率
- 可以简化JVM的实现，垃圾收集器通常在一个或者多个专门的线程中运行
- 发挥多处理器的强大能力



# 线程的风险

风险点	期望值
线程的安全性	永远不发生糟糕的事情
线程的活跃性	某件正确的事情最终会发生
线程的性能	正确的事情尽快发生





# Java应用与线程

- JVM启动时:

- JVM的内部任务（垃圾回收等）创建后台线程

- 创建主线程运行main

- AWT和Swing将创建线程管理用户界面事件



## 7.2 Java线程使用

- 7.2.1 Runnable 接口
- 7.2.2 Thread类



# 1 实现Runnable接口

```
public class RunnableClass implements Runnable {  
  
    public void run(){ // 覆写run()方法，作为线程 的操作主体  
        for(int i=0;i<10;i++){  
            System.out.println(name + "运行, i = " + i) ;  
        }  
    }  
}
```

```
public interface Runnable{  
    //必须实现，线程对象的线程体  
    public abstract void run();  
}
```



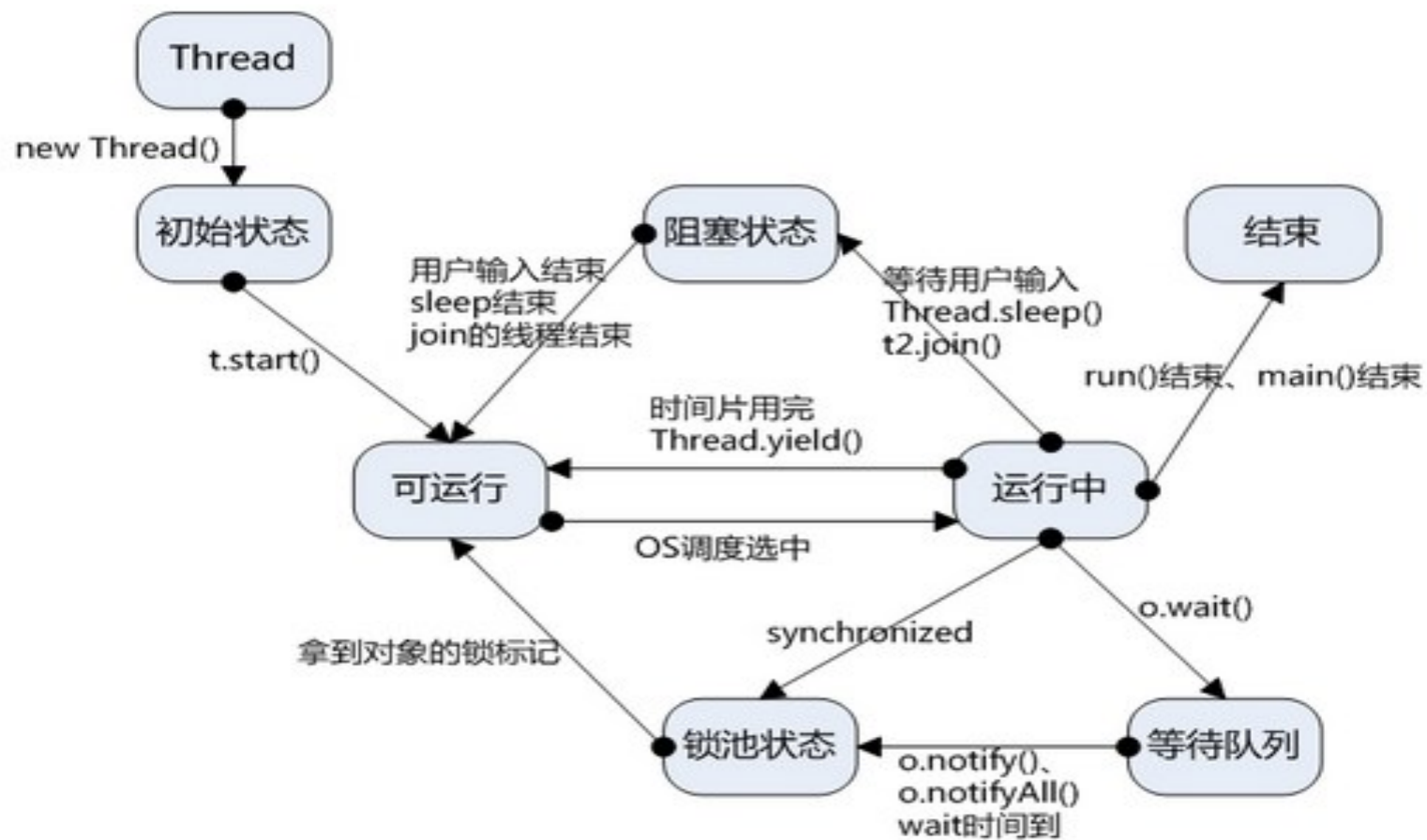
## 2 继承Thread类

```
public class ThreadClass extends Thread {  
  
    public void run(){ // 覆写run()方法，作为线程 的操作主体  
        for(int i=0;i<10;i++){  
            System.out.println(name + "运行, i = " + i) ;  
        }  
    }  
}
```

```
public class Thread implements Runnable {  
    public void run(){ ..... }  
}
```



# 线程的状态及状态转化图





# 线程的状态转化

- **可运行→运行**: 线程执行了 `start` 方法后, 线程处于可运行的状态, 最终由OS的线程调度来决定哪个可运行状态下的线程被执行。
- **运行→可运行**: CPU的时间片用完但线程还没有结束时, 线程变为可运行状态, 等待OS的再次调度; 运行的线程里执行 `Thread.yield()` 方法同样可以使当前线程变为可运行状态。
- **运行→阻塞**: 一个运行中的线程等待用户输入、调用 `Thread.sleep()`、调用了其他线程的 `join()` 方法, 则当前线程变为阻塞状态。阻塞状态的线程用户输入完毕、`sleep` 时间到、`join` 的线程结束, 则当前线程由阻塞状态变为可运行状态
- **运行→等待队列**: 运行中的线程调用 `wait` 方法, 此线程进入等待队列。
- **锁池状态**: 运行中的线程遇到 `synchronized` 同时没有拿到对象的锁标记、等待队列的线程 `wait` 时间到、等待队列的线程被 `notify` 方法唤醒、有其他线程调用 `notifyAll` 方法, 则线程变成锁池状态
- **锁池→可运行**: 锁池状态的线程获得对象锁标记, 则线程变成可运行状态。
- **运行中→结束**: 运行中的线程 `run` 方法执行完毕或 `main` 线程结束, 则线程运行结束。



# Thread和Runnable示例

eclipse-workspace - HelloJava/src/cn/sdu/java/Thread/ThreadClassTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

employee2.txt JTableDemo.java RunnableClass.java ThreadClass.java

```
3 public class RunnableClass implements Runnable {
4
5     private String name ;           // 表示线程的名称
6     public RunnableClass(String name){
7         this.name = name ;         // 通过构造方法配置名称
8     }
9     public void run(){ // 覆写run()方法, 作为线程的操作
10         for(int i=0;i<10;i++){
11             System.out.println(name + "运行, i = " + i);
12         }
13     }
14 }
```

```
3 public class ThreadClass extends Thread {
4
5     private String name ;           // 表示线程的名称
6     public ThreadClass(String name){
7         this.name = name ;         // 通过构造方法配置名称
8     }
9     public void run(){ // 覆写run()方法, 作为线程的操作
10         for(int i=0;i<10;i++){
11             System.out.println(name + "运行, i = " + i);
12         }
13     }
14 }
```

```
4
5 public static void main(String args[]) {
6     RunnableClass rc1 = new RunnableClass("线程一");
7     RunnableClass rc2 = new RunnableClass("线程二");
8     Thread t1 = new Thread(rc1); // 实例化Thread
9     Thread t2 = new Thread(rc2); // 实例化Thread
10     t1.start(); // 启动多线程
11     t2.start(); // 启动多线程
12 }
13 }
```

```
2
3 public class ThreadClassTest {
4     public static void main(String args[]){
5         ThreadClass tc1 = new ThreadClass("线程一");
6         ThreadClass tc2 = new ThreadClass("线程二");
7         tc1.start(); // 调用线程主体
8         tc2.start(); // 调用线程主体
9     }
10 }
11 }
```



# Thread和Runnable区别

- 实现Runnable接口方式可以避免继承Thread方式由于Java单继承特性带来的缺陷。
- 实现Runnable的代码可以被多个线程（Thread实例）共享  
详见火车售票示例



# 火车票余票共享示例

- 【条件】
- 火车票余票10张
- 售票窗口2个
- 对比Thread和Runnable两种形式

## 【建模】

- 每个线程代表一个窗口
- 在run()中进行卖票操作
- 窗口每卖出一张就减扣一张





# 火车票余票共享示例——结论

- Runnable模式的多线程可以进行数据共享  
Thread其实也可以，实例化的时候传入继承了Thread的子类  
详见ThreadWindowTest\_createFromThread.java
- 共享数据就需要考虑数据同步问题

# 线程用不好会出现差错

- 两人一起包饺子

A对馅料加盐（未告知B）

B又对馅料加盐（未询问A）

最后饭菜太咸没法吃



- 高频交易银行卡进行交易

存款线程收款前读取余额为500，+100元，余额变600（尚未回写）

扣款线程出款前读取余额为500，- 100元，余额变400

存款线程回写余额为600

扣款线程回写余额为400



## 7.3 线程同步

- 并发执行的线程之间需要共享资源或交换数据，则这一组线程称为交互线程。
- 交互线程并发执行时相互之间会干扰或者影响其他线程的执行结果。因此交互线程之间需要同步机制。





# 线程同步

- Java 使用synchronized同步锁实现线程同步：
- 使用方法：

## (1)同步语句

```
synchronized(对象)  
    语句
```

## (2)同步方法

```
synchronized 方法声明; //锁定方法  
synchronized(this){    //锁定语句块  
    //在方法体开头写与锁定方法等价  
    语句块  
}
```



# synchronized使用方法：

1. 修饰一个代码块，被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码；锁的是当前的实例对象。

```
class SyncThread implements Runnable {  
    private static int count;  
  
    public SyncThread() {  
        count = 0;  
    }  
  
    public void run() {  
        synchronized(this) {  
            for (int i = 0; i < 5; i++) {  
                try {  
                    System.out.println(Thread.currentThread().getName() + ":" + (count++));  
                    Thread.sleep(100);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

一个线程访问一个对象中的synchronized(this)同步代码块时，其他试图访问该对象的线程将被阻塞



# synchronized使用方法：

- 2. 修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象；

```
public synchronized void run() {  
    for (int i = 0; i < 5; i++) {  
        try {  
            System.out.println(Thread.currentThread().getName() + ":" + (count++));  
            Thread.sleep(100);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



# synchronized使用方法：

- 3. 修饰一个静态的方法，其作用的范围是整个静态方法，作用的对象是这个类的**所有对象**；

```
public synchronized static void method() {  
    for (int i = 0; i < 5; i++) {  
        try {  
            System.out.println(Thread.currentThread().getName() + ":" + (count++));  
            Thread.sleep(100);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}  
  
public synchronized void run() {  
    method();  
}  
}
```





# synchronized使用方法：

- 4. 修饰一个类，其作用的范围是synchronized后面括号括起来的部分，作用的对象是这个类的所有对象。

```
class SyncThread implements Runnable {  
    private static int count;  
  
    public SyncThread() {  
        count = 0;  
    }  
  
    public static void method() {  
        synchronized(SyncThread.class) {  
            for (int i = 0; i < 5; i++) {  
                try {  
                    System.out.println(Thread.currentThread().getName() + ":" + (count++));  
                    Thread.sleep(100);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

# 火车票余票共享示例（数据同步）

- 【条件】
- 火车票余票10张
- 售票窗口2个
- 对比Thread和Runnable两种形式



## 【建模】

- 每个线程代表一个窗口
- 在run()中进行卖票操作（对代码增加同步）
- 窗口每卖出一张就减扣一张



## 7.4 线程池

- 线程池：管理一组同构工作线程的资源池
- 由专门的工作线程从工作队列（等待执行的任务）中获取一个任务，执行任务，然后返回线程池并等待下一个任务。
- 任务是一组逻辑工作单元，线程是使得任务异步执行的机制



# 线程池优势

“在线程池中执行任务”比“为每一个任务分配一个线程”要更有优势

通过重用现有线程而不是创建新线程，可以在处理多个请求时分摊在线程创建和销毁过程中产生的巨大开销。



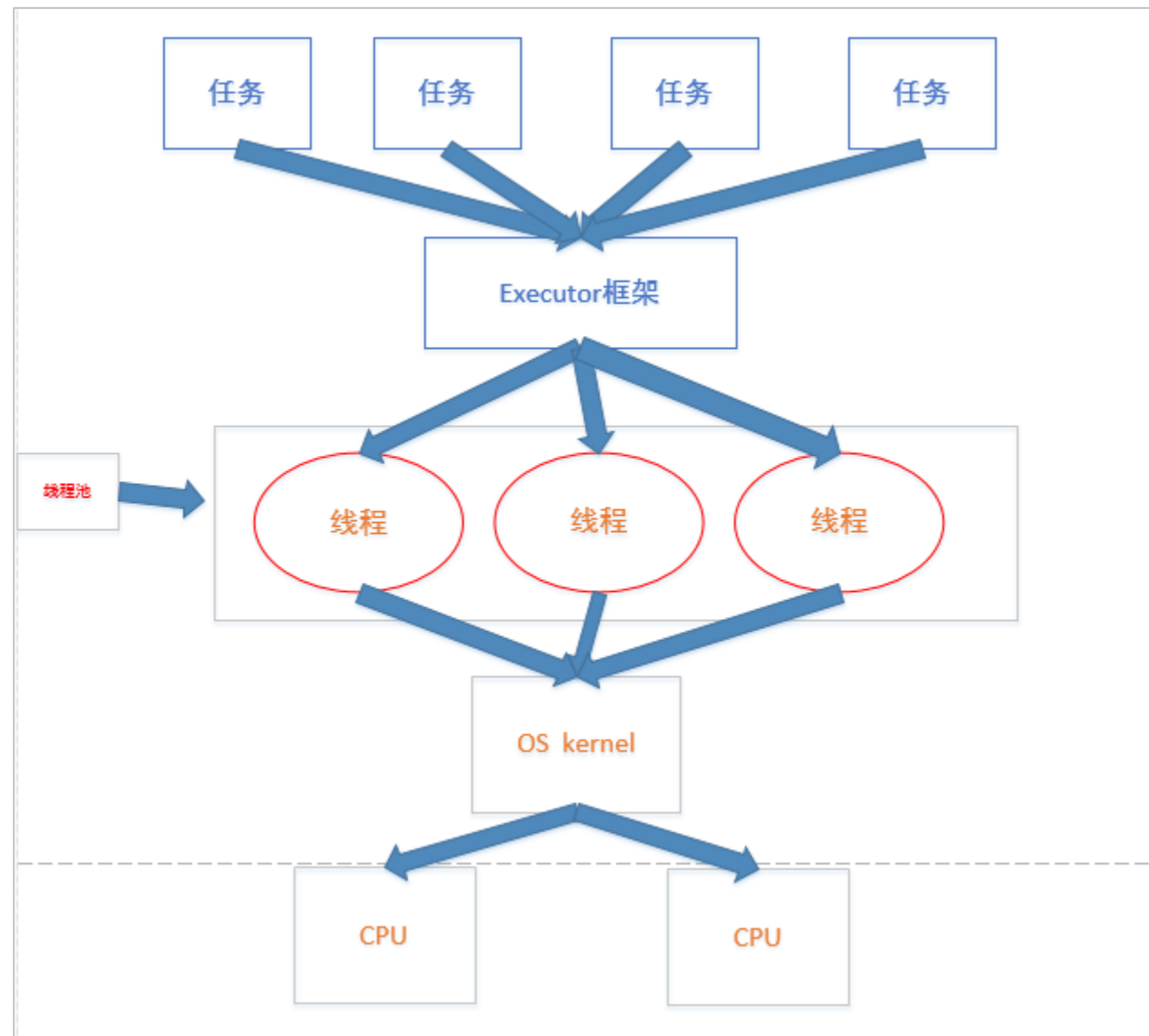


# 线程池分析

- **【本质】** 线程租赁模式
- **【目标】** 提高计算效率
- **【特征】** 资源标准化可相互替代  
并非一直占用（多个用户可轮流使用）  
创建成本>>承租成本
- **【类比】** 共享充电宝  
共享单车（单车池）      单车出列 → 出行任务 → 单车入列

# Executor框架

- Executor框架就是线程池的实现。
- Java线程被一对一映射为本地操作系统线程。Java线程启动时会创建一个本地操作系统线程；当Java线程终止时，这个操作系统线程也会被回收。操作系统会调用所有线程并将他们分配给可用的CPU。
- 可以将此种模式分为两层，在上层，Java多线程程序通常把应用程序分解为若干任务，然后使用用户级的调度器（Executor框架）将这些任务映射为固定数量的线程；在底层，操作系统内核将这些线程映射到硬件处理器上。





# Executor框架 (java.util.concurrent)

```
public interface Executor{
    void execute(Runnable command);
    //Runnable 表示任务
}
public interface ExecutorService extends Executor{
    void shutdown(); //扩展Executor的关闭功能
    List<Runnable> shutdown();
    .....
}
```



# 线程池创建

Executors类里面提供了一些静态工厂，生成一些常用的线程池。

- **newSingleThreadExecutor()**

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

- **newFixedThreadPool(int nThreads)**

创建固定大小的线程池。每提交一个任务就创建一个线程，直到达到线程池的最大数量。线程池的大小一旦达到最大值就会保持不变，再提交新任务，任务将会进入等待队列中等待。

- **newCachedThreadPool()**

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒处于等待任务到来）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。

- **newScheduledThreadPool(int corePoolSize)** 创建一个大小无限的线程池。