Java 应用与开发
线程编程

王晓东
wangxiaodong@ouc.edu.cn

中国海洋大学

November 6, 2018

学习目标

1. 线程基础：理解任务调度、进程和线程，掌握其联系和区别；掌握 Java 的线程模型，以及如何创建线程；理解后台线程。
2. 线程控制：理解线程的生命周期，明白各阶段的含义；掌握线程控制方法，理解各线程控制方法对线程状态切换的作用。
3. 线程的同步：理解临界资源问题，进一步明白线程安全的意义；了解关键字 synchronized 的用法；了解死锁的概念；通过生产者—消费者问题分析理解线程同步。

大纲

接下来…

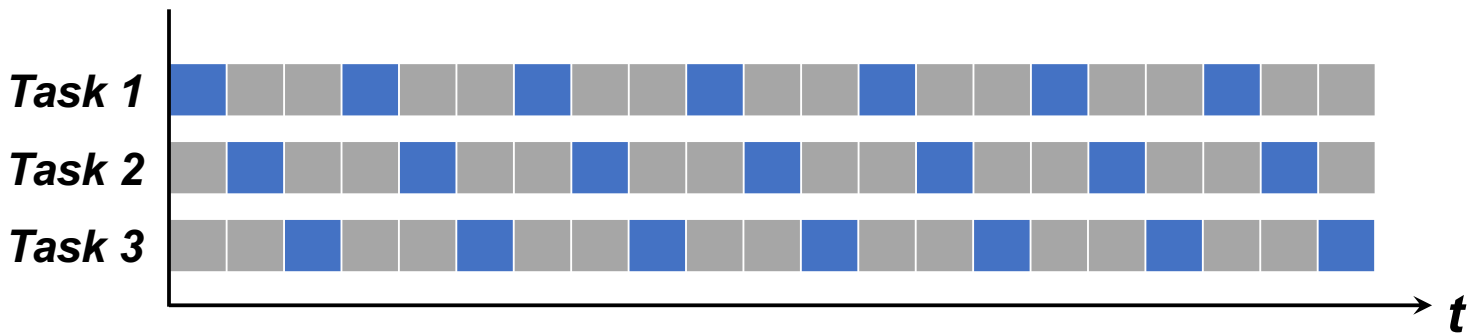
- 线程基础
  - 相关知识回顾
  - 线程的概念模型
  - 创建线程
  - 后台线程
- 线程控制
  - 线程生命的周期
  - 线程优先级
  - 线程串行化
  - 线程休眠
  - 线程让步
  - 线程挂起与恢复
  - 线程等待与通知
- 线程的同步

接下来…

概念回顾

❖ 任务调度

- 大部分操作系统的任务调度是采用**时间片轮转的抢占式调度方式**，一个任务执行一小段时间后强制暂停去执行下一个任务，每个任务轮流执行。
- CPU 的执行效率非常高，时间片非常短，在各个任务之间快速地切换，让人感觉像是多个任务在“同时进行”，这也就是我们所说的**并发**。



概念回顾

❖ 进程

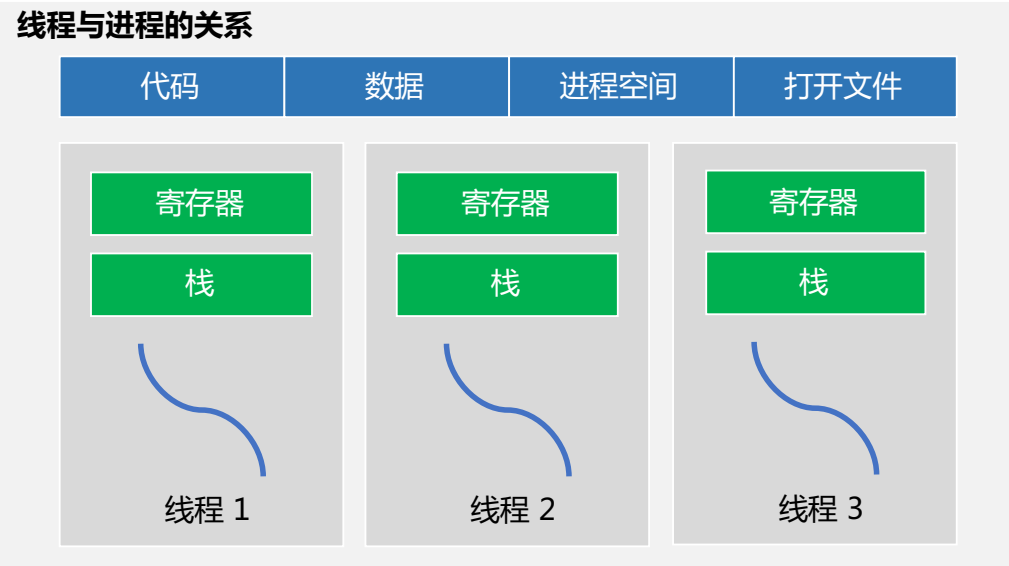
- 进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用程序运行的载体。
  (展示类 UNIX 系统的进程树)
- 进程一般由**程序段、数据段和进程控制块**三部分构成进程实体。

什么是线程

根据多任务原理，在一个程序内部也可以实现多个任务（顺序控制流）的并发执行，其中每个任务被称为**线程（Thread）**。更专业的表述为：

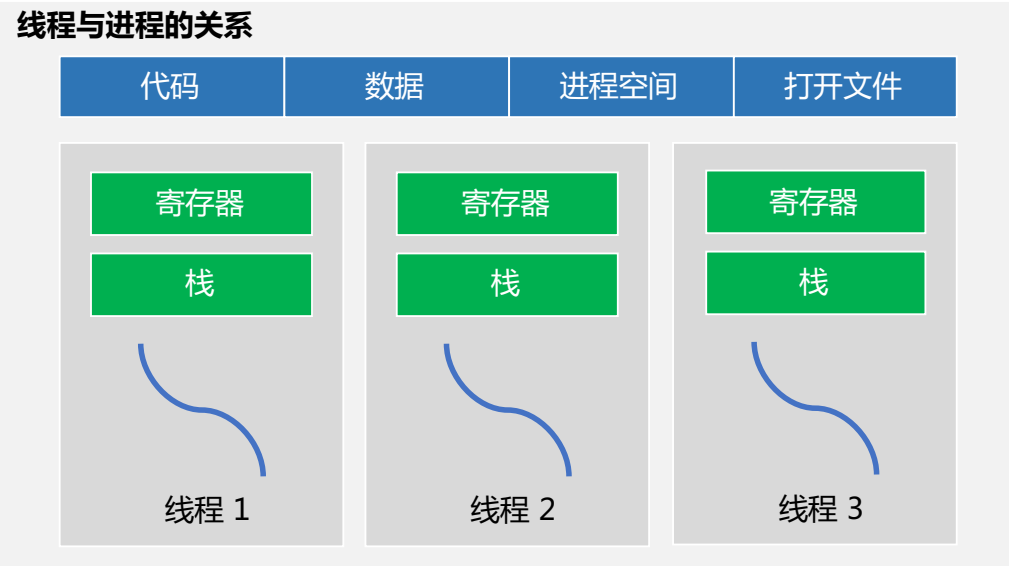
线程是程序内部的顺序控制流。

线程和进程的区别和联系



1. 每个进程都有独立的代码和数据空间（进程上下文），进程切换的开销大。
2. 线程作为“轻量的进程”，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。
3. 多进程——在操作系统中能同时运行多个任务（程序）。
4. 多线程——在同一应用程序中有多个顺序流同时执行。

线程和进程的区别和联系



1. 每个进程都有独立的代码和数据空间（进程上下文），进程切换的开销大。
2. 线程作为“轻量的进程”，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。
3. 多进程——在操作系统中能同时运行多个任务（程序）。
4. 多线程——在同一应用程序中有多个顺序流同时执行。

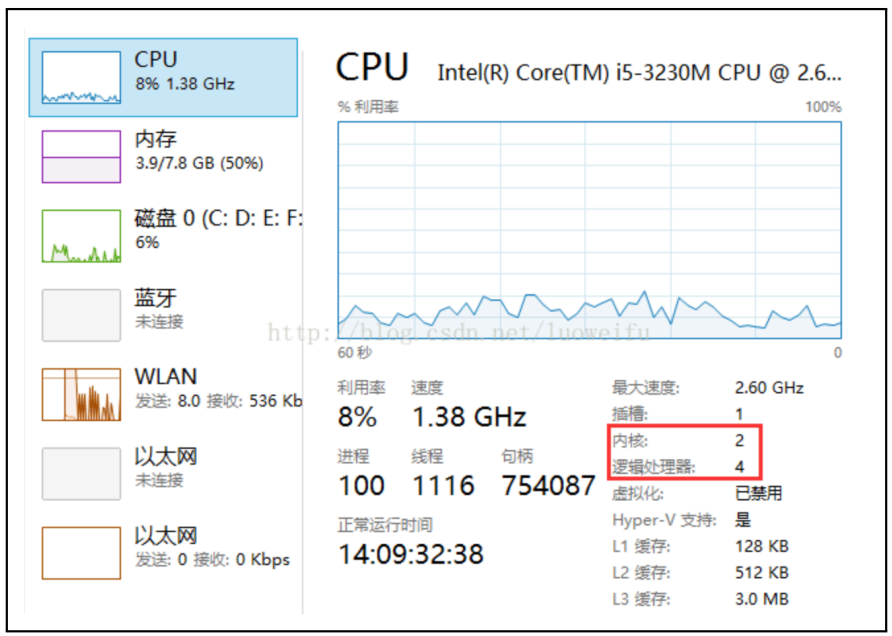
多核与多线程

- 多核处理器是指在一个处理器上集成多个运算核心以提高并行计算能力，每一个处理核心对应一个内核线程（Kernel Thread，KLT）。
- 内核线程是直接由操作系统内核支持的线程，由内核来完成线程切换，内核通过操作调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。

多核与多线程

一般一个处理核心对应一个内核线程，比如单核处理器对应一个内核线程，双核处理器对应两个内核线程。

而现代计算机采用超线程技术将一个物理处理核心模拟成两个逻辑处理核心对应两个内核线程，一般是双核四线程、四核八线程。[1]



[1]课后自行搜索了解超线程的概念，内核线程与用户线程的映射

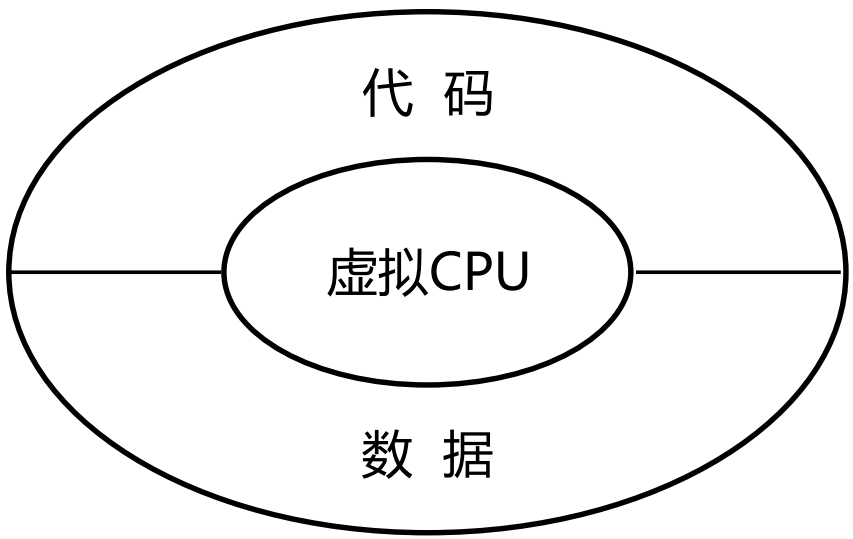
接下来…

- 线程基础
  - 相关知识回顾
  - 线程的概念模型
  - 创建线程
  - 后台线程
- 线程控制
  - 线程生命的周期
  - 线程优先级
  - 线程串行化
  - 线程休眠
  - 线程让步
  - 线程挂起与恢复
  - 线程等待与通知
- 线程的同步

Java 线程的概念模型

在 Java 语言中，多线程的机制通过**虚拟** CPU 来实现。

1. 虚拟的 CPU，由 java.lang.Thread 类封装和虚拟；
2. CPU 所执行的代码和数据，传递给 Thread 类对象。



接下来…

创建和启动线程的一般步骤

每个线程都是通过某个特定 Thread 对象所对应的方法 run() 来完成其操作，方法 run() 称为**线程体**。

1. 定义一个类实现 Runable 接口，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建 Runable 接口实现类的对象；
3. 创建 Thread 类的对象（封装 Runable 接口实现类型对象）；
4. 调用 Thread 对象的 start() 方法，启动线程。

课程配套代码 ▸ sample.thread.FirstThreadSample.java

创建和启动线程的一般步骤

每个线程都是通过某个特定 Thread 对象所对应的方法 run() 来完成其操作，方法 run() 称为**线程体**。

1. 定义一个类实现 Runable 接口，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建 Runable 接口实现类的对象；
3. 创建 Thread 类的对象（封装 Runable 接口实现类型对象）；
4. 调用 Thread 对象的 start() 方法，启动线程。

课程配套代码 ▸ sample.thread.FirstThreadSample.java

创建和启动线程的一般步骤

每个线程都是通过某个特定 Thread 对象所对应的方法 run() 来完成其操作，方法 run() 称为**线程体**。

1. 定义一个类实现 Runable 接口，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建 Runable 接口实现类的对象；
3. 创建 Thread 类的对象（封装 Runable 接口实现类型对象）；
4. 调用 Thread 对象的 start() 方法，启动线程。

课程配套代码 ▸ sample.thread.FirstThreadSample.java

创建和启动线程的一般步骤

每个线程都是通过某个特定 Thread 对象所对应的方法 run() 来完成其操作，方法 run() 称为**线程体**。

1. 定义一个类实现 Runable 接口，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建 Runable 接口实现类的对象；
3. 创建 Thread 类的对象（封装 Runable 接口实现类型对象）；
4. 调用 Thread 对象的 start() 方法，启动线程。

课程配套代码 ▸ sample.thread.FirstThreadSample.java

创建线程的第二种方式

❖ 直接继承 Thread 类创建线程

```
1   public class TestThread3 {
2     public static void main(String args[]) {
3       Thread t = new Runner3();
4       t.start();
5     }
6   }
7   class Runner3 extends Thread {
8     public void run() {
9       for(int i=0; i<30; i++) {
10        System.out.println("No.␣" + i);
11      }
12    }
13  }
```

1. 定义一个类继承 Thread 类，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建该 Thread 类的对象；
3. 调用该对象的 start() 方法。

两种创建线程的方式比较

❖ 使用 Runnable 接口创建线程

- 可以将虚拟 CPU、代码和数据分开，形成清晰的模型；
- 线程体 run() 方法所在的类还可以从其他类继承一些有用的属性或方法；
- 有利于保持程序风格的一致性。

❖ 直接继承 Thread 类创建线程

- Thread 子类无法再从其他类继承；
- 编写简单，run() 方法的当前对象就是线程对象，可直接操纵。

两种创建线程的方式比较

❖ 使用 Runnable 接口创建线程

- 可以将虚拟 CPU、代码和数据分开，形成清晰的模型；
- 线程体 run() 方法所在的类还可以从其他类继承一些有用的属性或方法；
- 有利于保持程序风格的一致性。

❖ 直接继承 Thread 类创建线程

- Thread 子类无法再从其他类继承；
- 编写简单，run() 方法的当前对象就是线程对象，可直接操纵。

多线程

Java 中引入线程机制的目的在于实现多线程（Multi-Thread）并发执行，以及实现多任务之间的协同。

❖ 使用多线程

课程配套代码 ➧ sample.thread.MultiThreadsSample.java

多线程

❖ 多线程之间可以共享代码和数据

```
1    Runner2 r = new Runner2();
2    Thread t1 = new Thread(r);
3    Thread t2 = new Thread(r);
```

线程	虚拟 CPU	代码
t1	Thread 类对象	Runner2 类中的 run() 方法
t2	Thread 类对象	Runner2 类中的 run() 方法

多线程间数据共享

- 当多个线程的执行代码都是同一个类的 run() 方法时，称这个多线程共享相同的**代码**。
- 当多个线程共享访问相同的对象时，则称它们共享相同的**数据**。

两种线程的创建方法主要区别在于数据的共享。
只要用同一个实现了 Runnable 接口的类的对象作为参数创建多个线程即可以实现多个线程共享相同的数据。

课程配套代码 ▸ sample.thread.ShareDataWithinThreadsSample.java

多线程间数据共享

- 当多个线程的执行代码都是同一个类的 run() 方法时，称这个多线程共享相同的**代码**。
- 当多个线程共享访问相同的对象时，则称它们共享相同的**数据**。

两种线程的创建方法主要区别在于**数据的共享**。
只要用同一个实现了 Runnable 接口的类的对象作为参数创建多个线程即可以实现多个线程共享相同的数据。

课程配套代码 ▸ sample.thread.ShareDataWithinThreadsSample.java

接下来…

- 线程基础
  - 相关知识回顾
  - 线程的概念模型
  - 创建线程
  - 后台线程
- 线程控制
  - 线程生命的周期
  - 线程优先级
  - 线程串行化
  - 线程休眠
  - 线程让步
  - 线程挂起与恢复
  - 线程等待与通知
- 线程的同步

后台线程

❖ 相关概念

后台处理 是指在分时处理或多任务系统中，当实时、会话式、高优先级或需迅速响应的计算机程序不再使用系统资源时，计算机去执行较低优先级程序的过程。批量处理、文件打印通常采取后台处理的形式。

后台线程 是指那些在后台运行的，为其他线程提供服务的功能，如 JVM 的垃圾回收线程等，后台线程也称为守护线程（Daemon Thread）。

用户线程 和后台线程相对应，其他完成用户任务的线程可称为“用户线程”。

后台线程

❖ 相关概念

后台处理 是指在分时处理或多任务系统中，当实时、会话式、高优先级或需迅速响应的计算机程序不再使用系统资源时，计算机去执行较低优先级程序的过程。批量处理、文件打印通常采取后台处理的形式。

后台线程 是指那些在后台运行的，为其他线程提供服务的功能，如 JVM 的垃圾回收线程等，后台线程也称为守护线程（Daemon Thread）。

用户线程 和后台线程相对应，其他完成用户任务的线程可称为“用户线程”。

后台线程

❖ 相关概念

后台处理 是指在分时处理或多任务系统中，当实时、会话式、高优先级或需迅速响应的计算机程序不再使用系统资源时，计算机去执行较低优先级程序的过程。批量处理、文件打印通常采取后台处理的形式。

后台线程 是指那些在后台运行的，为其他线程提供服务的功能，如 JVM 的垃圾回收线程等，后台线程也称为守护线程（Daemon Thread）。

用户线程 和后台线程相对应，其他完成用户任务的线程可称为“用户线程”。

后台线程

❖ Thread 类与后台线程相关的方法

1. 测试当前线程是否为守护线程，如果是则返回 true，否则返回 false

```
1    public final boolean isDaemon()
```

2. 将当前线程标记为守护线程或用户线程，本方法必须在启动线程前调用

```
1    public final void setDaemon(Boolean on)
```

课程配套代码 ➧ sample.thread.DaemonThreadSample.java

GUI 线程

GUI 程序运行过程中，系统会自动创建若干 GUI 线程，以提供 GUI 程序运行所需的功能，主要包括：

1. 窗体显示和重绘；
2. GUI 事件处理；
3. 关闭抽象窗口工具集等。

课程配套代码 ➧ sample.thread.GUIThreadSample.java

GUI 自动创建的线程

- AWT-Windows 线程
  负责从操作系统获取底层事件通知，并将之发送到系统事件队列（EventQueue）等待处理。在其他平台上运行时，此线程的名字也会作相应变化，例如在 Unix 系统则为“AWT-Unix”。
- AWT-EventQueue-n 线程
- AWT-Shutdown 线程
- DestroyJavaVM 线程

GUI 自动创建的线程

- AWT-Windows 线程
- AWT-EventQueue-n 线程
  也称事件分派线程，该线程负责从事件队列中获取事件，将之分派到相应的 GUI 组件（事件源）上，进而触发各种 GUI 事件处理对象，并将之传递给相应的事件监听器进行处理。
- AWT-Shutdown 线程
- DestroyJavaVM 线程

GUI 自动创建的线程

- AWT-Windows 线程
- AWT-EventQueue-n 线程
- AWT-Shutdown 线程
  负责关闭已启用的抽象窗口工具，释放其所占用的资源，该线程将等到其他 GUI 线程均退出后才开始其清理工作。
- DestroyJavaVM 线程

GUI 自动创建的线程

- AWT-Windows 线程
- AWT-EventQueue-n 线程
- AWT-Shutdown 线程
- DestroyJavaVM 线程
  在所有其他用户线程退出后，负责释放任意线程所占用系统资源并卸载 Java 虚拟机。该线程在主线程运行结束时由系统自动启动，但要等到所有其他用户线程均退出后才开始其卸载工作。

接下来…

- 线程基础
  - 相关知识回顾
  - 线程的概念模型
  - 创建线程
  - 后台线程
- 线程控制
  - 线程生命的周期
  - 线程优先级
  - 线程串行化
  - 线程休眠
  - 线程让步
  - 线程挂起与恢复
  - 线程等待与通知
- 线程的同步

接下来…

线程的生命周期

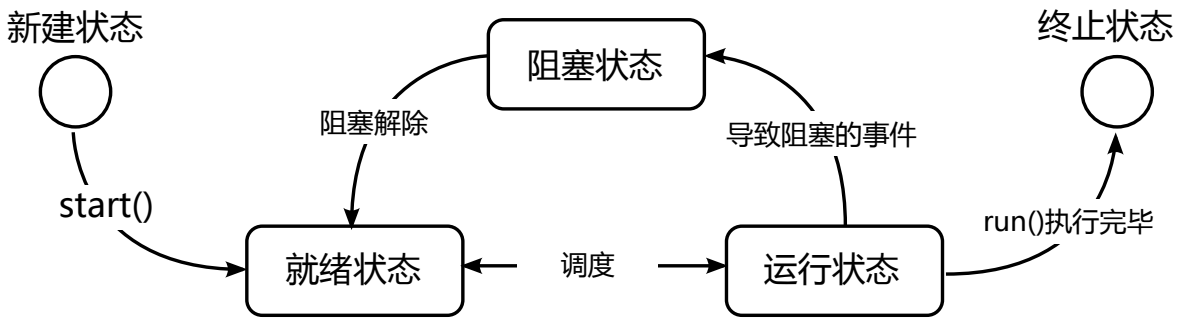
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程的生命周期

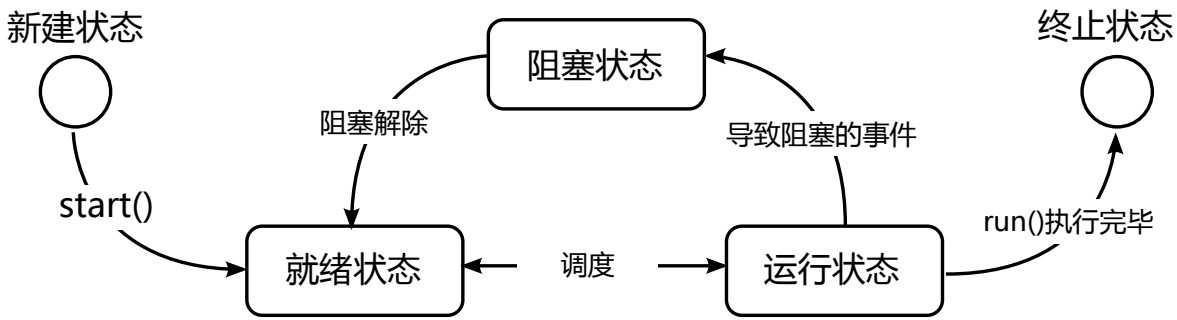
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程的生命周期

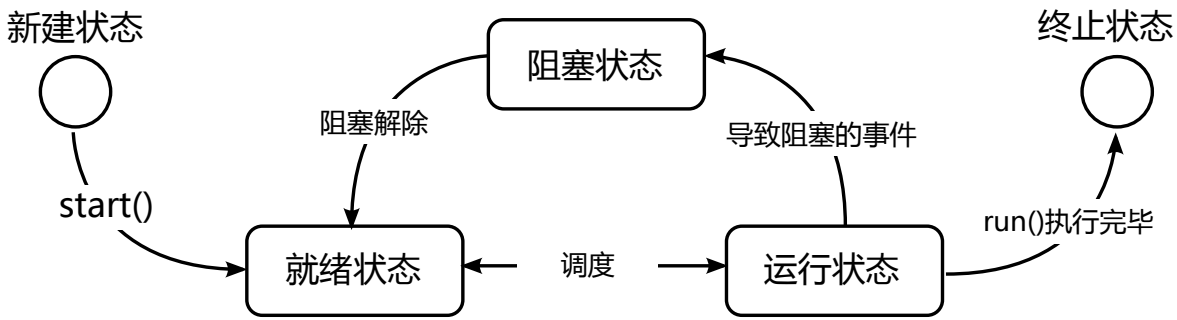
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程的生命周期

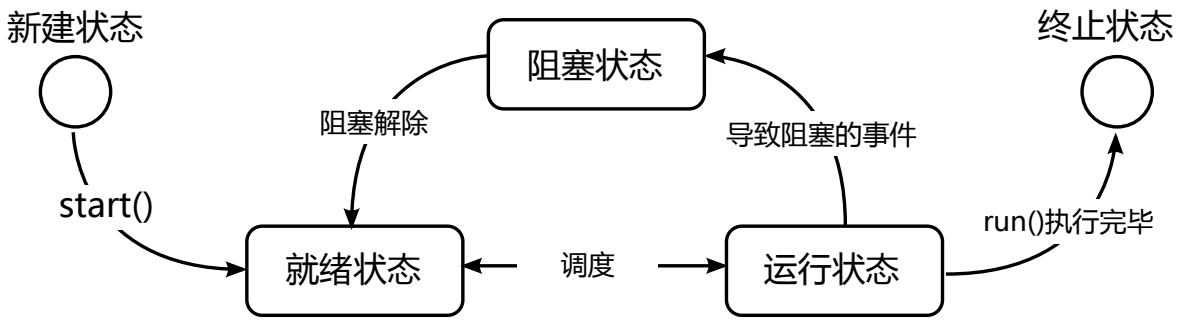
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程的生命周期

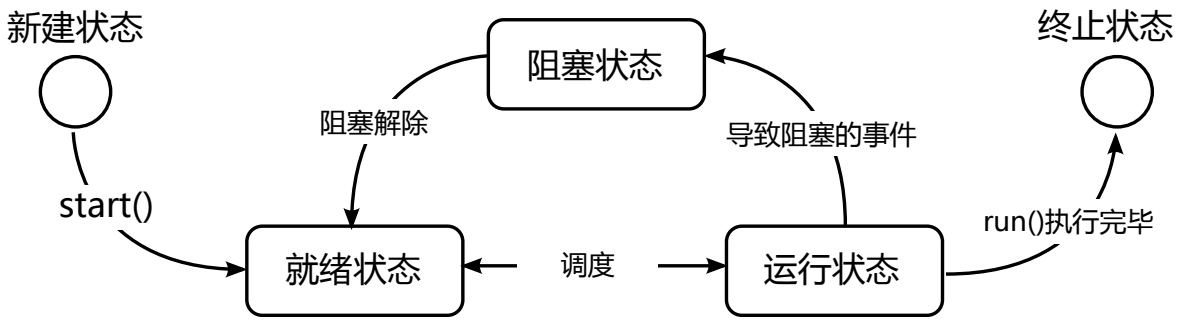
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



接下来…

线程优先级

线程的优先级用数字来表示，范围从 1 到 10。主线程的缺省优先级是 5，子线程的优先级默认与其父线程相同。可以使用 Thread 类提供的方法获得和设置线程的优先级：

- 获取当前线程优先级

```
1    public final int getPriority();
```

- 设定当前线程优先级

```
1    public final void setPriority(int newPriority);
```

相关静态整型常量：

```
1    Thread.MIN_PRIORITY = 1
2    Thread.MAX_PRIORITY = 10
3    Thread.NORM_PRIORITY = 5
```

练习 请自行编写线程优先级测试代码。

接下来…

线程串行化

在多线程程序中，如果在一个线程运行的过程中要用到另一个线程的运行结果，则可进行线程的串型化处理。
Thread 类提供的相关方法：

```
1 public final void join()
2 public final void join(long millis)
3 public final void join(long millis, int nanos)
```

课程配套代码 ➧ sample.thread.ThreadJoinSample.java

线程串行化

❖ 线程串行化程序说明

- 主线程在执行过程中调用了线程 t 的 join() 方法，该方法导致当前**线程阻塞**（主线程）。
- 直到线程 t 运行终止后，主线程才会获得继续执行的机会，相当于将线程 t 串行加入到主线程中。

接下来…

- 线程基础
  - 相关知识回顾
  - 线程的概念模型
  - 创建线程
  - 后台线程
- 线程控制
  - 线程生命的周期
  - 线程优先级
  - 线程串行化
  - 线程休眠
  - 线程让步
  - 线程挂起与恢复
  - 线程等待与通知
- 线程的同步

线程休眠

线程休眠，即暂停执行当前运行中的线程，使之进入阻塞状态，待经过指定的“延迟时间”后再醒来并转入到就绪状态。

Thread 类提供的相关方法：

```
1 public static void sleep(long millis)
2 public static void sleep(long millis, int nanos)
```

课程配套代码 ➧ sample.thread.DigitaltimerByThreadSleep.java

接下来…

线程让步

线程让步，让运行中的线程主动放弃当前获得的 CPU 处理机会，但不是使该线程阻塞，而是使之转入**就绪状态**。
Thread 类提供的相关方法：

```
1    public static void yield()
```

☛ 注意

从代码执行结果来看，由于设置了线程让步，thread-1（第二个线程）明显执行时间长。调用 yield() 方法只是令当前线程主动在时间片到期前使其他线程获得运行机会。

课程配套代码 ➧ sample.thread.ThreadYieldSample.java

接下来…

线程挂起与恢复

线程挂起 暂时停止当前运行中的线程，使之转入阻塞状态，并且不会自动恢复运行。

线程恢复 使得一个已挂起的线程恢复运行。

Thread 类提供的相关方法：

```
1    public final void suspend()
2    public final void resume()
```

注意

suspend() 和 resume() 方法已不提倡使用，原因是 suspend() 方法挂起线程时并不释放其锁定的资源，这可能会影响到其他线程的执行，且容易导致线程死锁。

接下来…

- 线程基础
  - 相关知识回顾
  - 线程的概念模型
  - 创建线程
  - 后台线程
- 线程控制
  - 线程生命的周期
  - 线程优先级
  - 线程串行化
  - 线程休眠
  - 线程让步
  - 线程挂起与恢复
  - 线程等待与通知
- 线程的同步

线程等待与通知

❖ 将运行中的线程转为阻塞状态的另外一种途径

调用该线程中被锁定资源（Java 对象）的 wait() 方法，该方法在 Object 类中定义，其功能是让当前线程等待（进入阻塞状态），直到有其他线程调用了同一个对象的 notify() 或 notifyAll() 方法通知其结束等待，或是经历了约定的等待时间后等待线程才会醒来，重新进入可执行状态（就绪状态）。

❖ 等待线程与 suspend() 方法导致的线程挂起比较

- 线程挂起时不会释放所占用的资源；
- 线程等待时则会释放资源，以使其他线程获得运行机会。

线程等待与通知

❖ 将运行中的线程转为阻塞状态的另外一种途径

调用该线程中被锁定资源（Java 对象）的 wait() 方法，该方法在 Object 类中定义，其功能是让当前线程等待（进入阻塞状态），直到有其他线程调用了同一个对象的 notify() 或 notifyAll() 方法通知其结束等待，或是经历了约定的等待时间后等待线程才会醒来，重新进入可执行状态（就绪状态）。

❖ 等待线程与 suspend() 方法导致的线程挂起比较

- 线程挂起时不会释放所占用的资源；
- 线程等待时则会释放资源，以使其他线程获得运行机会。

接下来…

- 线程基础
  - 相关知识回顾
  - 线程的概念模型
  - 创建线程
  - 后台线程
- 线程控制
  - 线程生命的周期
  - 线程优先级
  - 线程串行化
  - 线程休眠
  - 线程让步
  - 线程挂起与恢复
  - 线程等待与通知
- 线程的同步

临界资源问题

两个线程 A 和 B 在同时操纵 Stack 类的同一个实例 (栈)，A 向栈里 push 一个数据，B 要从堆栈中 pop 一个数据。

❖ 代码

```
1  public class Stack {
2    int idx = 0;
3    char[ ] data = new char[6];
4
5    public void push(char c) {
6      data[idx] = c;
7      idx++;
8    }
9    public char pop() {
10     idx--;
11     return data[idx];
12   }
13 }
```

临界资源问题

❖ 问题分析

1. 操作之前，假设 data = |a|b| | | |，idx = 2；
2. 线程 A 执行 push 中的第一个语句，将 c 推入堆栈; data = |a|b|c| | |，idx = 2；
3. 线程 A 还未执行 idx++ 语句，A 的执行被线程 B 中断，B 执行 pop 方法，data = |a|b|c| | | idx = 1；
4. 线程 A 继续执行 push 的第二个语句: data = |a|b|c| | |，idx = 2；
5. 最后的结果相当于 c 没有入栈，产生这种问题的原因在于对共享数据访问的操作的不完整性。

临界资源问题

❖ 问题分析

1. 操作之前，假设 data = |a|b| | | |，idx = 2；
2. 线程 A 执行 push 中的第一个语句，将 c 推入堆栈; data = |a|b|c| | |，idx = 2；
3. 线程 A 还未执行 idx++ 语句，A 的执行被线程 B 中断，B 执行 pop 方法，data = |a|b|c| | | idx = 1；
4. 线程 A 继续执行 push 的第二个语句: data = |a|b|c| | |，idx = 2；
5. 最后的结果相当于 c 没有入栈，产生这种问题的原因在于对共享数据访问的操作的不完整性。

临界资源问题

❖ 问题分析

1. 操作之前，假设 data = |a|b| | | |，idx = 2；
2. 线程 A 执行 push 中的第一个语句，将 c 推入堆栈; data = |a|b|c| | |，idx = 2；
3. 线程 A 还未执行 idx++ 语句，A 的执行被线程 B 中断，B 执行 pop 方法，data = |a|b|c| | | idx = 1；
4. 线程 A 继续执行 push 的第二个语句: data = |a|b|c| | |，idx = 2；
5. 最后的结果相当于 c 没有入栈，产生这种问题的原因在于对共享数据访问的操作的不完整性。

临界资源问题

❖ 问题分析

1. 操作之前，假设 data = |a|b| | | |，idx = 2；
2. 线程 A 执行 push 中的第一个语句，将 c 推入堆栈; data = |a|b|c| | |，idx = 2；
3. 线程 A 还未执行 idx++ 语句，A 的执行被线程 B 中断，B 执行 pop 方法，data = |a|b|c| | | idx = 1；
4. 线程 A 继续执行 push 的第二个语句: data = |a|b|c| | |，idx = 2；
5. 最后的结果相当于 c 没有入栈，产生这种问题的原因在于对共享数据访问的操作的不完整性。

临界资源问题

❖ 问题分析

1. 操作之前，假设 data = |a|b| | | |，idx = 2；
2. 线程 A 执行 push 中的第一个语句，将 c 推入堆栈; data = |a|b|c| | |，idx = 2；
3. 线程 A 还未执行 idx++ 语句，A 的执行被线程 B 中断，B 执行 pop 方法，data = |a|b|c| | | idx = 1；
4. 线程 A 继续执行 push 的第二个语句: data = |a|b|c| | |，idx = 2；
5. 最后的结果相当于 c 没有入栈，产生这种问题的原因在于对共享数据访问的操作的不完整性。

什么是临界资源

在并发程序设计中，对多线程共享的资源或数据称为**临界资源**（或同步资源），而把每个线程中访问临界资源的那一段代码称为**临界代码**（或临界区）。

- 在一个时刻只能被一个线程访问的资源就是临界资源。
- 访问临界资源的那段代码就是临界区，临界区必须互斥地使用。

互斥锁

- Java 引入对象**互斥锁**机制来实现线程的互斥操作，保证共享数据操作的完整性。
- Java 中每个对象都有一个互斥锁与之相连，用来保证在任一时刻，只能有一个线程访问该对象。
- 多线程对临界资源的并发访问是通过竞争互斥锁实现的。

synchronized 的用法

为了保证互斥，Java 语言使用**synchronized**关键字标识同步的资源，包括：

- 对象

```
synchronized(对象) {
  临界代码段
}
```

- 方法

```
public synchronized 返回值类型 方法名 {
  方法体
}
```

等效方式：

```
public 返回值类型 方法名 {
  synchronized(this) {
    方法体
  }
}
```

- 语句块（一段代码），用法同方法的等效方式

synchronized 的用法示例

- 用于方法声明中，标明整个方法为同步方法

```
1    public synchronized void push(char c) {
2      data[idx] = c;
3      idx++;
4    }
```

- 用于修饰语句块，标明整个语句块为同步块

```
1    // Other code
2    public char pop() {
3      synchronized(this) {
4        idx--;
5        return data[idx];
6      }
7      // Other code
8    }
```

synchronized 的功能

- 首先判断对象或者方法的**互斥锁**是或否存在，若存在就获得互斥锁，然后执行紧随其后的临界代码段或方法体；
- 如果对象或方法的互斥锁不在（已经被其他线程拿走），就进入线程**等待状态**，直到获得互斥锁。

课程配套代码 ◗ sample.thread.syn.WithdrawMoneyFromBankSample.java

线程死锁

并发运行的多个线程间彼此等待、都无法运行的状态称为**线程死锁**。

为避免死锁，在线程进入阻塞状态时应尽量释放其锁定的资源，以为其他的线程提供运行的机会。

❖ 相关方法

- public final void wait()
- public final void notify()
- public final void notifyAll()

线程间通信

通过线程间的对话来解决线程间的同步问题。

❖ 线程间通信的有效手段

wait() 如果一个正在执行同步代码（synchronized）的线程 A 执行了 wait() 调用（在对象 x 上），该线程暂停执行而进入对象 x 的等待队列，并释放已获得的对象 x 的互斥锁。线程 A 要一直等到其他线程在对象 x 上调用 notify() 或 notifyAll() 方法，才能重新获得对象 x 的互斥锁后继续执行（从 wait() 语句后继续执行）。

notify() 唤醒正在等待该对象互斥锁的第一个线程。

notifyAll() 唤醒正在等待该对象互斥锁的所有线程，具有最高优先级的线程首先被唤醒并执行。

线程间通信

通过线程间的对话来解决线程间的同步问题。

❖ 线程间通信的有效手段

wait() 如果一个正在执行同步代码（synchronized）的线程 A 执行了 wait() 调用（在对象 x 上），该线程暂停执行而进入对象 x 的等待队列，并释放已获得的对象 x 的互斥锁。线程 A 要一直等到其他线程在对象 x 上调用 notify() 或 notifyAll() 方法，才能重新获得对象 x 的互斥锁后继续执行（从 wait() 语句后继续执行）。

notify() 唤醒正在等待该对象互斥锁的第一个线程。

notifyAll() 唤醒正在等待该对象互斥锁的所有线程，具有最高优先级的线程首先被唤醒并执行。

线程间通信

通过线程间的对话来解决线程间的同步问题。

❖ 线程间通信的有效手段

wait() 如果一个正在执行同步代码（synchronized）的线程 A 执行了 wait() 调用（在对象 x 上），该线程暂停执行而进入对象 x 的等待队列，并释放已获得的对象 x 的互斥锁。线程 A 要一直等到其他线程在对象 x 上调用 notify() 或 notifyAll() 方法，才能重新获得对象 x 的互斥锁后继续执行（从 wait() 语句后继续执行）。

notify() 唤醒正在等待该对象互斥锁的第一个线程。

notifyAll() 唤醒正在等待该对象互斥锁的所有线程，具有最高优先级的线程首先被唤醒并执行。

线程间通信

通过线程间的对话来解决线程间的同步问题。

❖ 线程间通信的有效手段

wait() 如果一个正在执行同步代码（synchronized）的线程 A 执行了 wait() 调用（在对象 x 上），该线程暂停执行而进入对象 x 的等待队列，并释放已获得的对象 x 的互斥锁。线程 A 要一直等到其他线程在对象 x 上调用 notify() 或 notifyAll() 方法，才能重新获得对象 x 的互斥锁后继续执行（从 wait() 语句后继续执行）。

notify() 唤醒正在等待该对象互斥锁的第一个线程。

notifyAll() 唤醒正在等待该对象互斥锁的所有线程，具有最高优先级的线程首先被唤醒并执行。

Object.wait() 和 notify()

- wait() 和 notify() 只能在同步代码块中调用。
- wait() 在放弃 CPU 资源的同时交出了对资源的控制权。

Thread.sleep() 与 Object.wait()、notify() 的区别

- 所属对象不同。sleep() 是 Thread 类的方法，而 wait()，notify()，notifyAll() 是 Object 类中定义的方法，都会影响线程的执行行为。
- 锁行为不同。
- 线程恢复方式不同。

Thread.sleep() 与 Object.wait()、notify() 的区别

- 所属对象不同。
- 锁行为不同。Thread.sleep() 不会导致锁行为的改变，如果当前线程是拥有锁的，那么 Thread.sleep() 不会让线程释放锁。可以简单认为和锁相关的方法都定义在 Object 类中，因此调用 Thread.sleep() 不会影响锁的相关行为。
- 线程恢复方式不同。

Thread.sleep() 与 Object.wait()、notify() 的区别

- 所属对象不同。
- 锁行为不同。
- 线程恢复方式不同。Thread.sleep 和 Object.wait 都会暂停当前的线程，即表示它暂时不再需要 CPU 的执行时间。区别是调用 wait 后，需要别的线程执行 notify/notifyAll 才能够重新获得 CPU 执行时间。

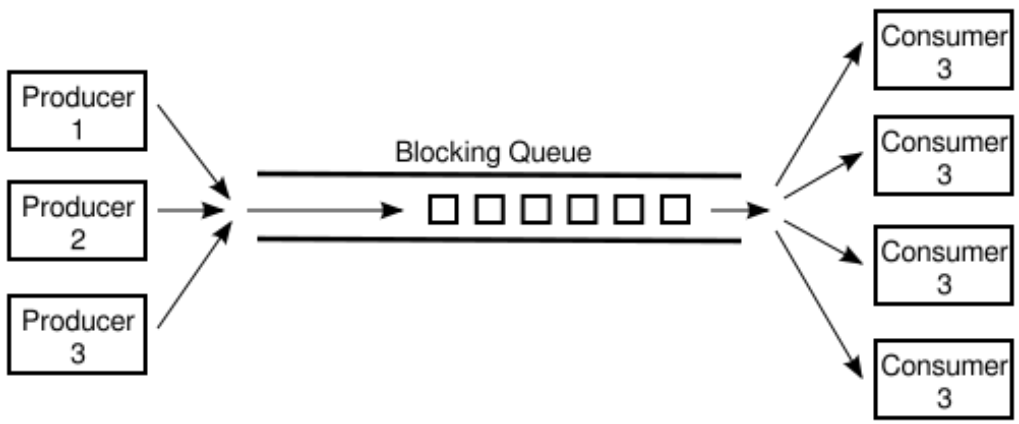
生产者—消费者问题

生产者消费者模型就是在一个系统中存在生产者和消费者两种角色，他们之间通过内存缓冲区进行通信，生产者生产消费者需要的资料，消费者把资料做成产品。

生产者—消费者问题

❖ 生产者消费者问题是线程模型中的经典问题

- 生产者和消费者在同一时间段内共用同一存储空间，生产者向空间里生产数据，而消费者取走数据。
- 阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费者解耦的。



课程配套代码 ➧ sample.thread.syn.ProducerAndConsumer.java

本节习题

❖ 简答题

1. 简述线程的基本概念。程序、进程、线程的关系是什么？
2. 线程的生命周期包括哪些基本状态？这些状态的关系如何？状态间的切换控制如何进行？（可以通过思维导图、文字描述等方式梳理线程状态与控制转换方法之间的关系）

❖ 小编程

1. 编程实践生产者—消费者模式，并在理解的基础上对代码给出比较完整的注释。

THE END

wangxiaodong@ouc.edu.cn