面向对象的软件构造导论

第十章：多线程

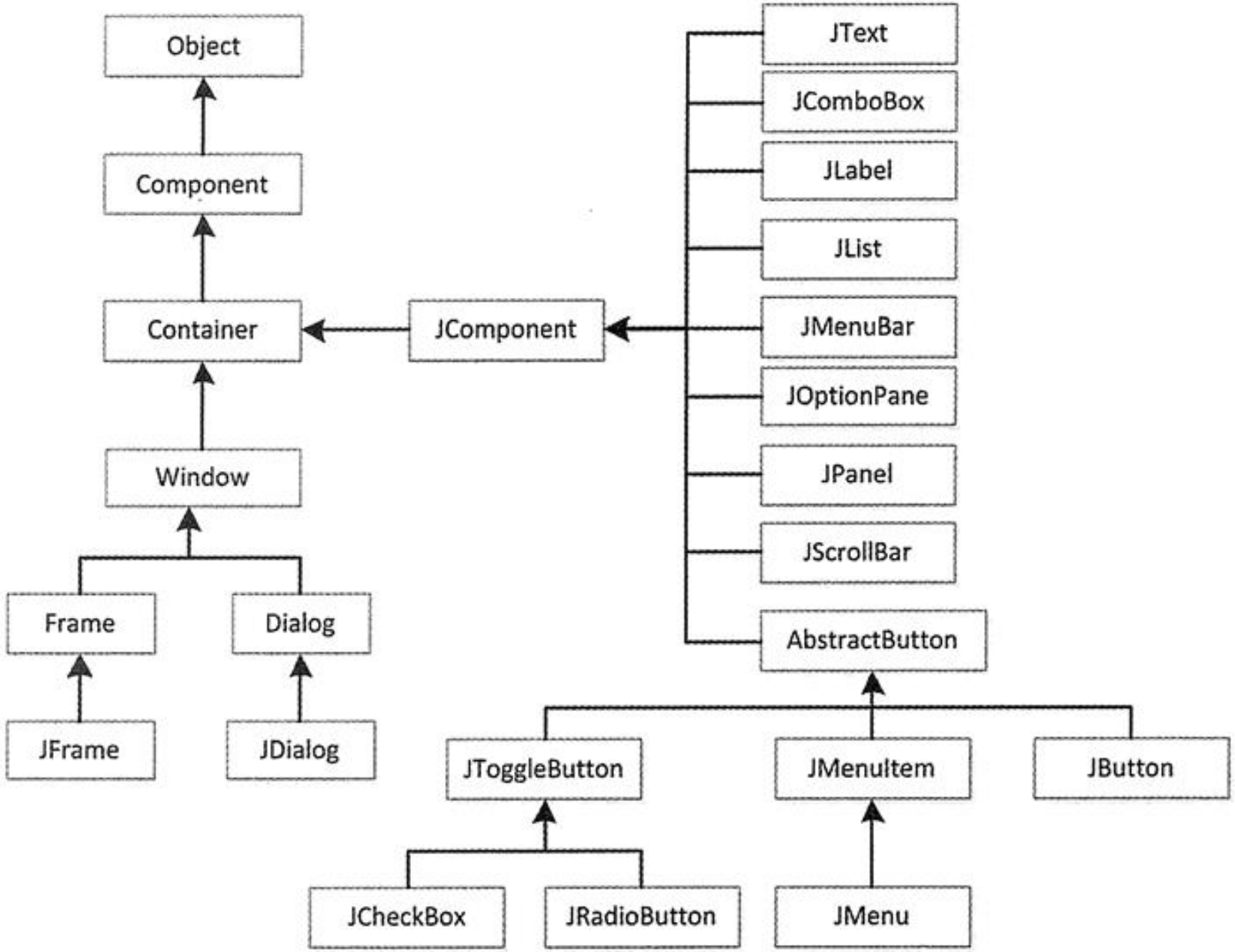
课程内容回顾（第九章）

- Swing框架
- Swing图形处理、绘制颜色的原理
- 事件机制
- Swing基本用户组件
- MVC模式

Swing框架

- Swing GUI包含了两种元素：**组件和容器**。
- 组件是单独的控制元素，例如按键或者文本编辑框。组件要放到容器中才能显示出来。
- 容器也是组件，因此容器也可放到别的容器中。
- 组件和容器构成了包含层级关系。

Swing框架


Object
Component
Container
JComponent
Window
Frame
Dialog
JFrame
JDialog
JText
JComboBox
JLabel
JList
JMenuBar
JOptionPane
JPanel
JScrollBar
AbstractButton
JToggleButton
JMenuItem
JButton
JCheckBox
JRadioButton
JMenu

Swing框架

- 容器是一种可以包含组件的特殊组件。Swing中有两大类容器。
  - 一类是**重量级容器**，或者称为顶层容器，它们不继承于Jcomponent，包括JFrame，JApplet，Jdialog. 它们的最大特点是不能被别的容器包含，只能作为界面程序的最顶层容器来包含其它组件。
  - 第二类容器是**轻量级容器**，或者称为中间层容器，它们继承于JComponent，包括JPanel，JScrollPane等。中间层容器用来将若干个相关联的组件放在一起。由于中间层容器继承于JComponent，因此它们本身也是组件，它们可以（也必须）包含在其它的容器中。

Swing框架

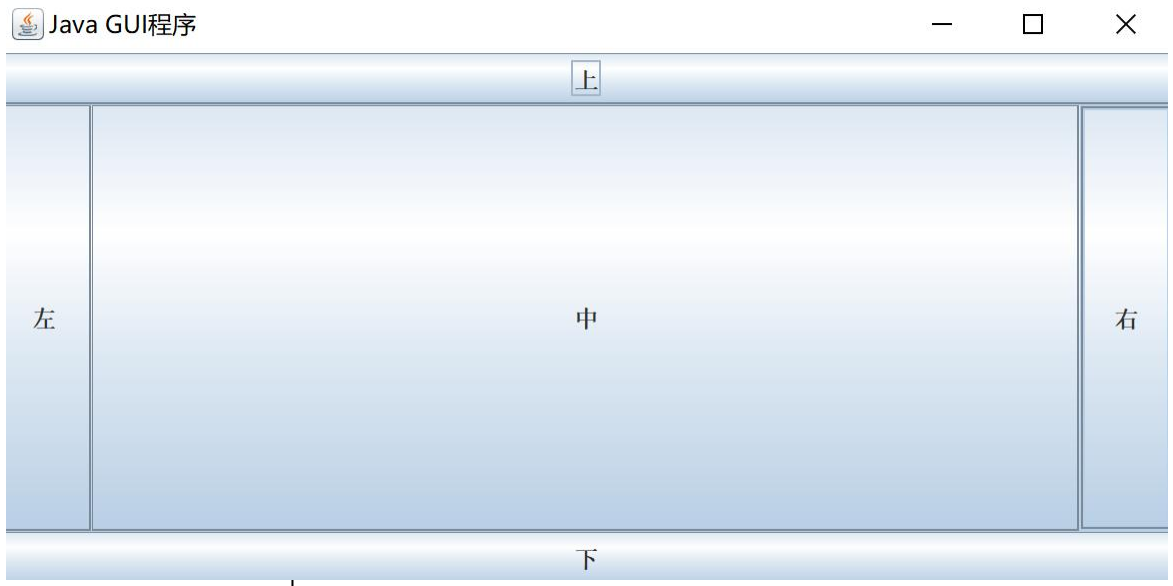
- Java提供了若干种布局管理器。

布局管理器	特性
FlowLayout	流式布局管理器，是从左到右，中间放置，一行放不下就换到另外一行。
BorderLayout	边框布局管理器分为东、南、西、北、中心五个方位。
GridLayout	网格式布局。
GridBagLayout	网格式布局，可以放置不同大小的组件。
BoxLayout	盒布局管理器，把组件水平或者竖直排在一起。
SpringLayout	按照一定的约束条件来组织组件。

Swing框架

- 例如BorderLayout

```
JFrame frame=new JFrame("Java GUI程序");    //创建Frame窗口
frame.setSize(400,200);
frame.setLayout(new BorderLayout());    //为Frame窗口设置布局为BorderLayout
JButton button1=new JButton ("上");
JButton button2=new JButton("左");
JButton button3=new JButton("中");
JButton button4=new JButton("右");
JButton button5=new JButton("下");
frame.add(button1,BorderLayout.NORTH);
frame.add(button2,BorderLayout.WEST);
frame.add(button3,BorderLayout.CENTER);
frame.add(button4,BorderLayout.EAST);
frame.add(button5,BorderLayout.SOUTH);
frame.setBounds(300,200,600,300);
frame.setVisible(true);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



显示窗体

- 顶层窗口（没有包含在其它窗口中的窗口）称为窗体(Frame)。
- Swing中用于描述顶层窗口的类名为JFrame，扩展了AWT中的Frame库。
- JFrame是极少数不绘制在画布上的Swing组件之一。它的修饰部件（按钮、标题栏、图标等）由用户的窗口系统绘制。

显示窗体

```
1.  class SimpleFrame extends JFrame{
2.      private static final int DEFAULT_WIDTH = 300;
3.      private static final int DEFAULT_HEIGHT = 200;
4.
        public SimpleFrame(){
5.          setSize(DEFAULT_WIDTH,DEFAULT_HEIGHT);
6.      }
7.  }
```

- 默认情况下，窗体大小为0×0像素。

显示窗体

```
1.  public class SimpleFrameTest {
2.      public static void main(String[] args){
3.          EventQueue.invokeLater(()->
4.          {
5.              var frame = new SimpleFrame();
6.              frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7.              frame.setVisible(true);
8.          });
9.      }
10. }
```

显示窗体中的信息

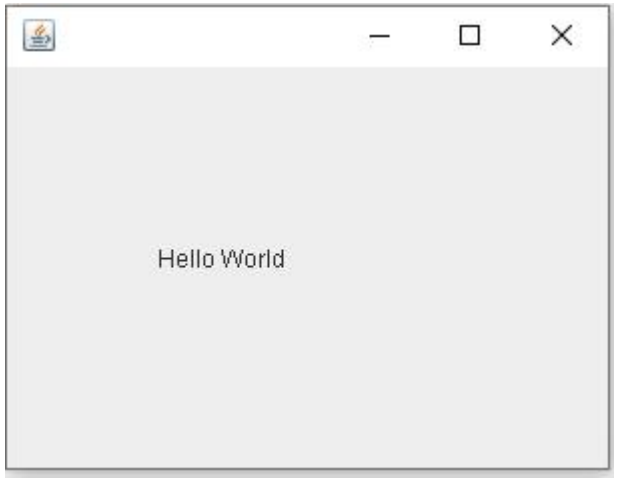
- 我们想在窗体中显示信息（如字符串“Hello World”）。
- 我们把一个组件添加到窗体中，消息将绘制在这个组件上。
- 在组件上进行绘制：
  - 定义一个扩展JComponent的类
  - 覆盖其中的paintComponent方法。
  - 在paintComponent方法中，设置Graphics对象。Graphics对象包含了绘制图案、图像和文本的方法。Java中，所有的绘制都必须通过Graphics对象完成。

显示窗体中的信息

```
1. class helloworldComponent extends JComponent{
2.     private static final int MSG_X = 75;
3.     private static final int MSG_Y = 100;
4.
       private static final int DEFAULT_WIDTH = 300;
5.     private static final int DEFAULT_HEIGHT = 200;
6.
       public void paintComponent(Graphics g){
7.         g.drawString("Hello World", MSG_X, MSG_Y);
8.     }
9.
       public Dimension getPreferredSize(){
10.        return new Dimension(DEFAULT_WIDTH,DEFAULT_HEIGHT);
11.    }
12.}
```

显示窗体中的信息

```
13. class helloworldFrame extends JFrame{
14.
        public helloworldFrame()
15.     {
16.         add(new helloworldComponent());
17.         pack();
18.     }
19. }
```



- 在Line 10.中，我们设定了组件的大小，返回一个有首选宽度和高度的Dimension类对象。
- Line 17.中，在窗体中填入组件时，用pack()方法来使用它们的首选大小。

绘制2D图形

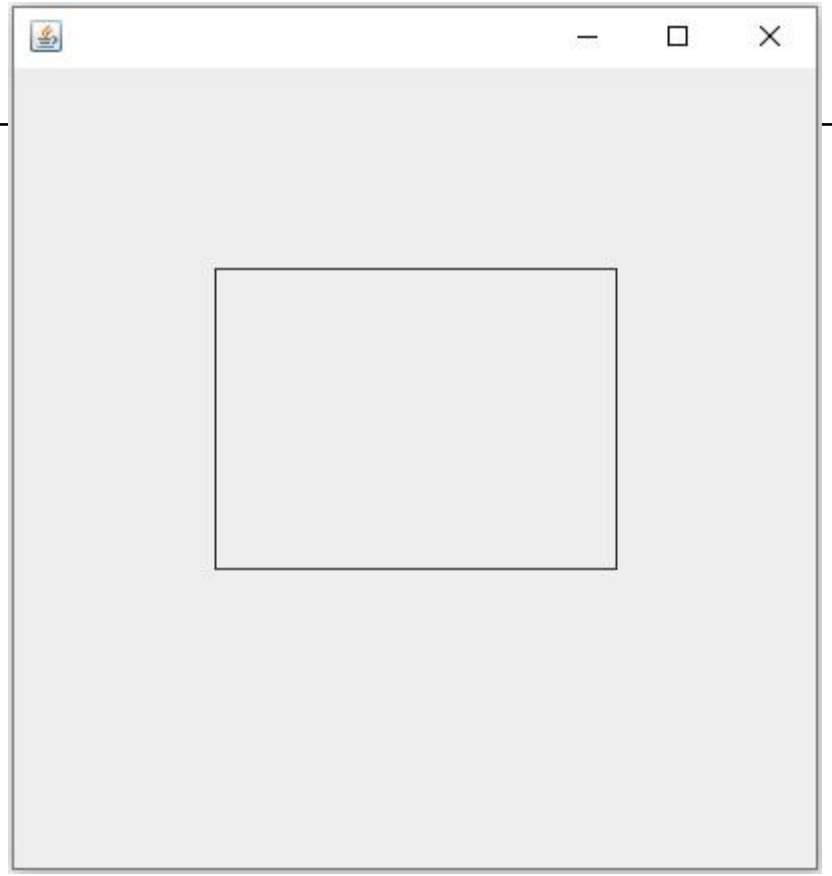
- 获得Graphics2D类的一个对象，使用Java 2D库绘制图形。
- Java 2D库采用面向对象的方式组织几何图形：
  - Line2D
  - Rectangle2D
  - Ellipse2D
- Java 2D库针对像素采用的是浮点坐标。内部计算都采用单精度float.为避免强制类型转换的处理，可以使用Double图形类。

绘制2D图形

```
1.  class DrawComponent extends JComponent{
2.      public static final int DEFAULT_WIDTH = 400;
3.      public static final int DEFAULT_HEIGHT = 400;
4.
        public void paintComponent(Graphics g){
5.          var g2 = (Graphics2D) g;
6.
            double leftX = 100;
7.          double topY = 100;
8.          double width = 200;
9.          double height = 150;
10.
11.         var rect = new Rectangle2D.Double(leftX,topY,width,height);
12.         g2.draw(rect);
13.     }
```

绘制2D图形

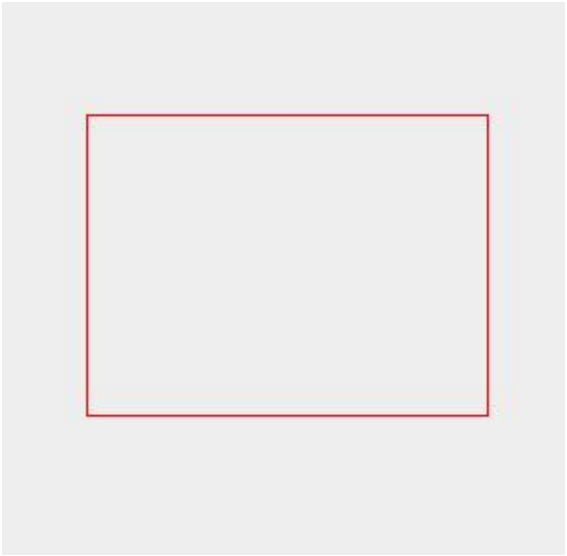
```
14.     public Dimension getPreferredSize(){
15.             return new Dimension(DEFAULT_WIDTH,DEFAULT_HEIGHT);
16.     }
17.
18. }
```



使用颜色

- 使用Graphics2D类的setPaint方法可以为图形上下文的所有后续的绘制操作选择颜色。

```
var rect = new Rectangle2D.Double(leftX,topY,width,height);
g2.setPaint(Color.RED);
g2.draw(rect);
```



使用颜色

- 可以用一种颜色填充一个封闭图形的内部。

```
var rect = new Rectangle2D.Double(leftX,topY,width,height);
g2.setPaint(Color.RED);
g2.fill(rect);
```

显示图像

- 可以使用ImageIcon类从文件读取图像。
- 变量image包含一个封装了图像数据的对象的引用。可以使用Graphics类的drawImage方法显示这个图像。

```
1.      public void paintComponent(Graphics g)
2.      {
3.          int X = 100;
4.          int Y = 100;
5.          Image image = new ImageIcon(path).getImage();
6.
            g.drawImage(image, X, Y, null);
7.
        }
```

事件机制

- 任何支持GUI的操作环境都要不断地监视按键或点击鼠标这样的事件。这些事件再报告给正在运行的程序。每个程序将决定如何对这些事件做出响应。
- 事件处理机制（三类对象）
  - 事件（Event）：用户对组件的一次操作称为一个事件
  - 事件源（ Event Source）：事件发生的场所，通常就是各个组件如按钮或滚动条。
  - 事件监听器：实现了监听器接口(listener interface)的类实例。

事件机制

```
1.ActionListener listener = , , ,;
2.var button = new Jbutton("OK");
3.button.addActionListener(listener)
```

- 只要按钮被点击（产生了“动作事件”），listener对象就会得到通知。

实例：按钮点击

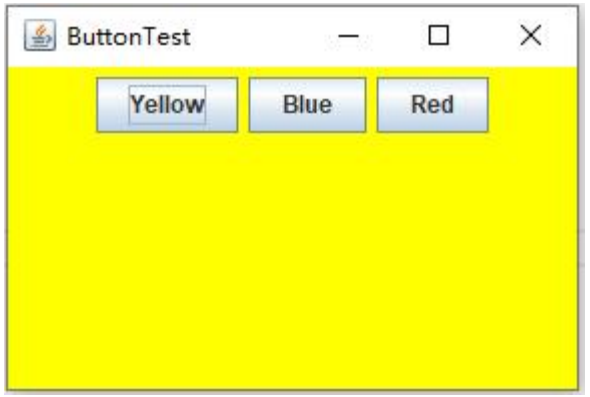
```
1. public class ButtonFrame extends JFrame
2. {
3.     private JPanel buttonPanel;
4.     private static final int DEFAULT_WIDTH = 300;
5.     private static final int DEFAULT_HEIGHT = 200;
6.
       public ButtonFrame()
7.     {
8.         setSize(DEFAULT_WIDTH,DEFAULT_HEIGHT);
9.
           var YellowButton = new JButton("Yellow");
10.        var BlueButton = new JButton("Blue");
11.        var RedButton = new JButton("Red");

12.        buttonPanel = new JPanel();
```

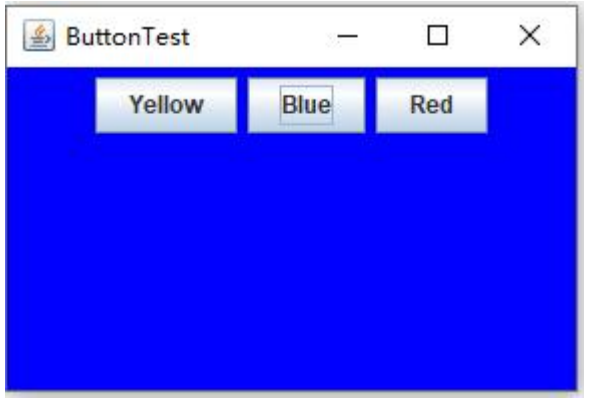
实例：按钮点击

```
13.         buttonPanel = new JPanel();
14.
            buttonPanel.add(YellowButton);
15.         buttonPanel.add(BlueButton);
16.         buttonPanel.add(RedButton);
17.
            add(buttonPanel);
18.
            var YellowAction = new ColorAction(Color.YELLOW);
19.         var BlueAction = new ColorAction(Color.BLUE);
20.         var RedAction = new ColorAction(Color.RED);
21.
            YellowButton.addActionListener(YellowAction);
22.         BlueButton.addActionListener(BlueAction);
23.         RedButton.addActionListener(RedAction);
24.     }
```

实例：按钮点击

```
25.private class ColorAction implements ActionListener{
26.     private Color backgroundColor;
27.
        public ColorAction(Color C){
28.         backgroundColor = C;
29.     }

30.     public void actionPerformed(ActionEvent event){
31.         buttonPanel.setBackground(backgroundColor);
32.     }
33.}
```





简洁地指定监听器

- 一般情况下，每个监听器执行一个单独的动作。

```
exitButton.addActionListener(event->System.exit(0));
```

- 有多个相互关联的动作，可以实现一个辅助方法（以颜色按钮为例）。

```
public void makeButton(String name, Color backgroundColor)
{
    var button = new JButton(name);
    buttonPanel.add(button);
    button.addActionListener(event->
        buttonPanel.setBackground(backgroundColor));
}
```

简洁地指定监听器

- 有多个相互关联的动作，可以实现一个辅助方法,从而改进上例。

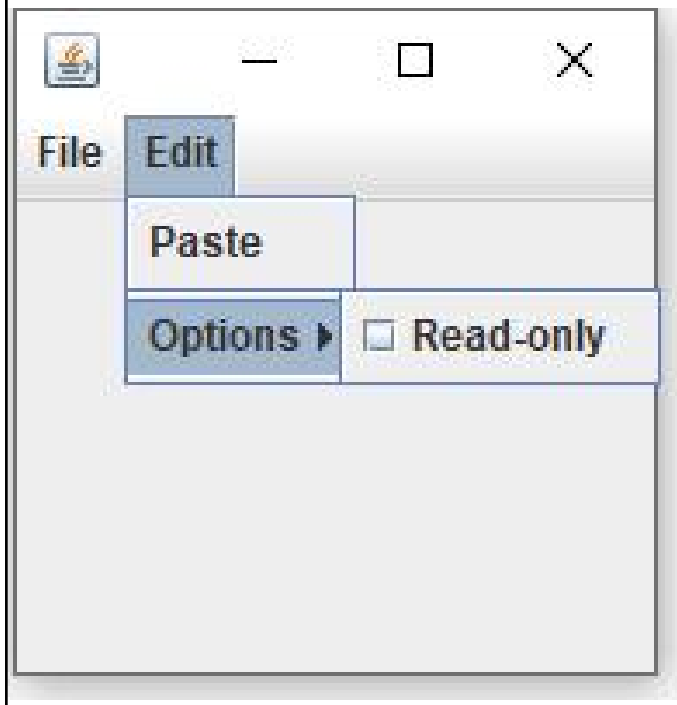
```
public BottonFrame2()
{
    setSize(DEFAULT_WIDTH,DEFAULT_HEIGHT);

    buttonPanel = new JPanel();
    add(buttonPanel);

    makeButton("yellow", Color.YELLOW);
    makeButton("blue", Color.BLUE);
    makeButton("red", Color.RED);
    makeButton(“green", Color.Green);
}
```

Swing基本用户组件　文本输入

- 三个继承自JTextComponent类（抽象类）的方法：
  - 文本域(JTextField)：接收单行文本。
  - 文本区(JTextArea)：接收多行文本。
  - 密码域(JPasswordField)：接收单行文本，且不显示文本内容。

菜单

```
1.  var menuBar = new JMenuBar();    //创建菜单栏
2.  frame.setJMenuBar(menuBar);      //将菜单栏放在窗体顶部
3.  var fileMenu = new JMenu("File");
4.  menuBar.add(fileMenu);
5.  var editMenu = new JMenu("Edit");
6.  menuBar.add(editMenu);
7.  var pasteItem = new JMenuItem("Paste");
8.  editMenu.add(pasteItem);
9.  editMenu.addSeparator();
10. var readonlyItem = new JCheckBoxMenuItem("Read-only");
11. var optionMenu = new JMenu("Options");
12. optionMenu.add(readonlyItem);
13. editMenu.add(optionMenu);
```



MVC模式

- 模型：
  - 存储完整的内容。
  - 实现改变内容和查找内容的方法。
  - 模型没有用户界面，是完全不可见的。
- 视图：
  - 一个模型可以有多个视图。
  - 每个视图可以显示全部内容的不同部分或不同方面。
  - 当模型更新时，需要通知与之关联的所有视图同步更新。

MVC模式

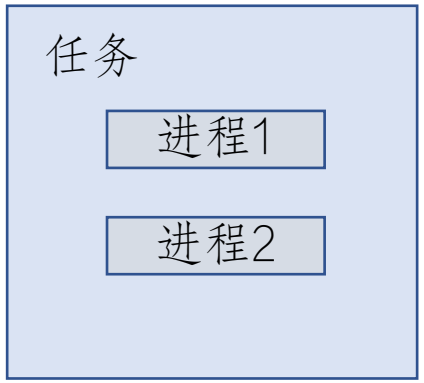
- 控制器：
  - 使视图与模型分离开
  - 负责处理用户输入事件，如点击鼠标和按键。
  - 决定是否将事件转化成对模型或视图的更改。
  - 例：用户在文本框中敲下了一个按键，控制器调用模型的“插入字符”命令，然后模型告诉视图进行更新。
  - 例：用户按下一个箭头键，控制器通知视图滚动，对底层文本不会有影响。

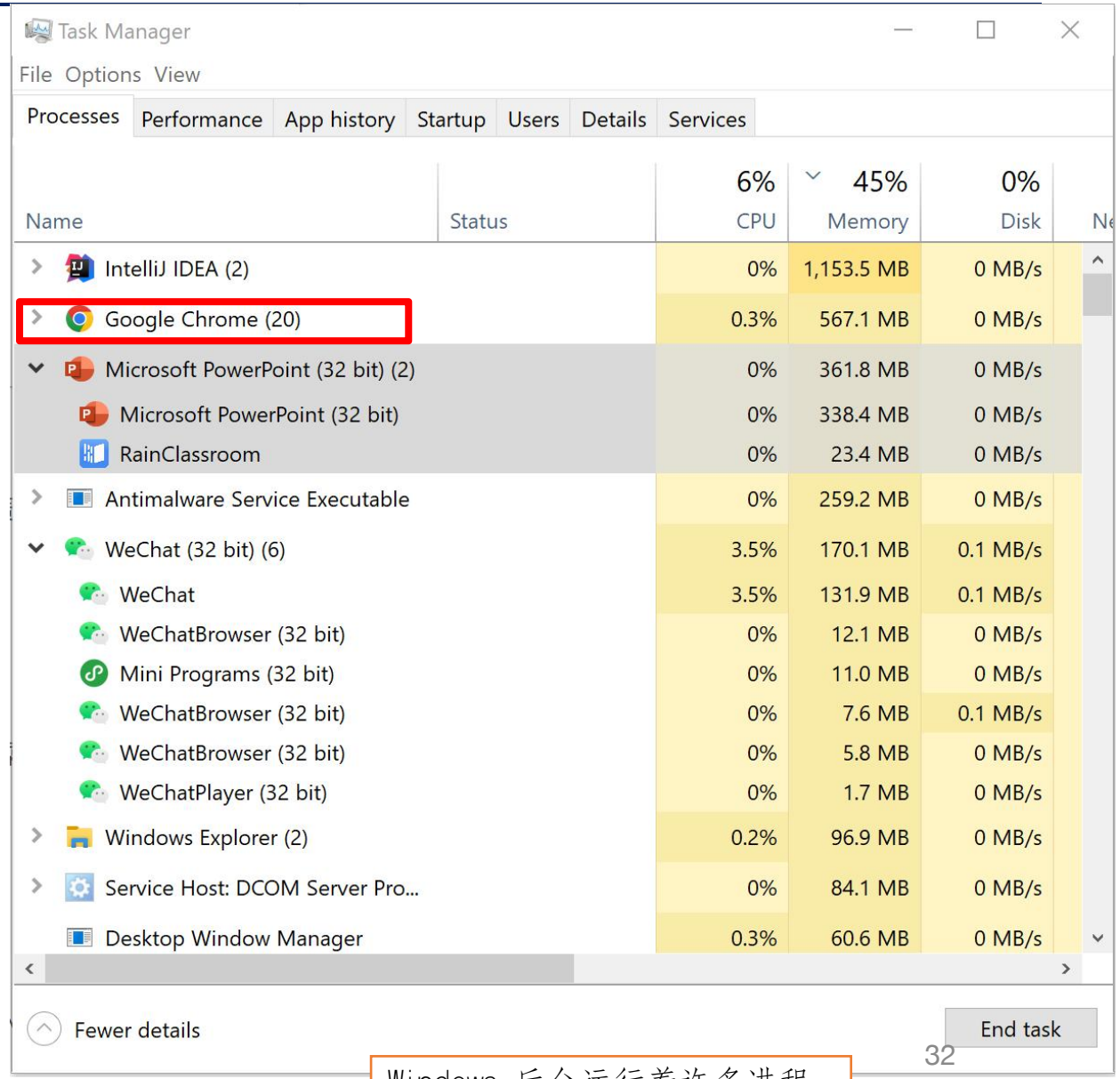
课程导航

- 进程与线程、多线程
- Java中对线程的控制
- 同步、死锁及如何避免
- 任务创建与线程池
- 多线程应用—生产者与消费者模式

什么是进程

- 进程（Process）：正在运行的程序的实例
  - 私有空间，彼此隔离
  - 是操作系统分配资源的基本单位
  - 多进程之间不共享内存
  - 进程之间通过消息传递进行协作
  - 一般来说：进程 == 程序 == 应用
    - ✓但一个应用中也可能包含多个进程

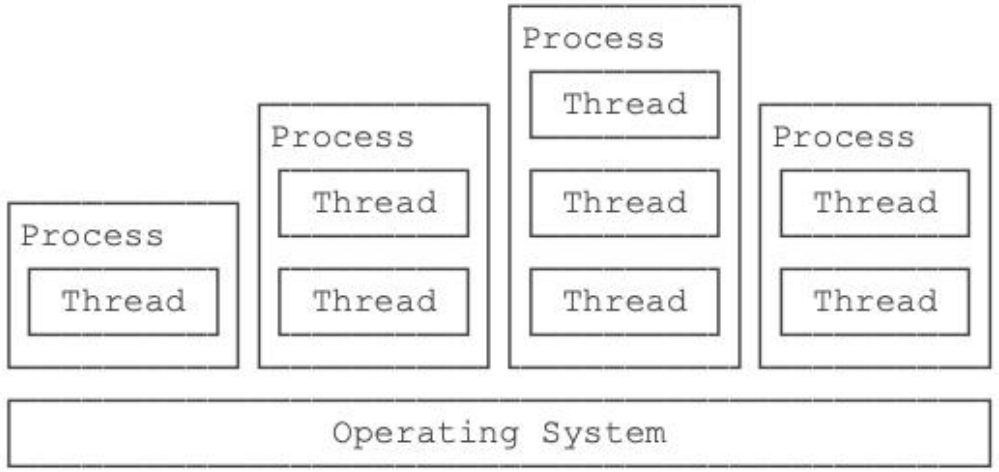


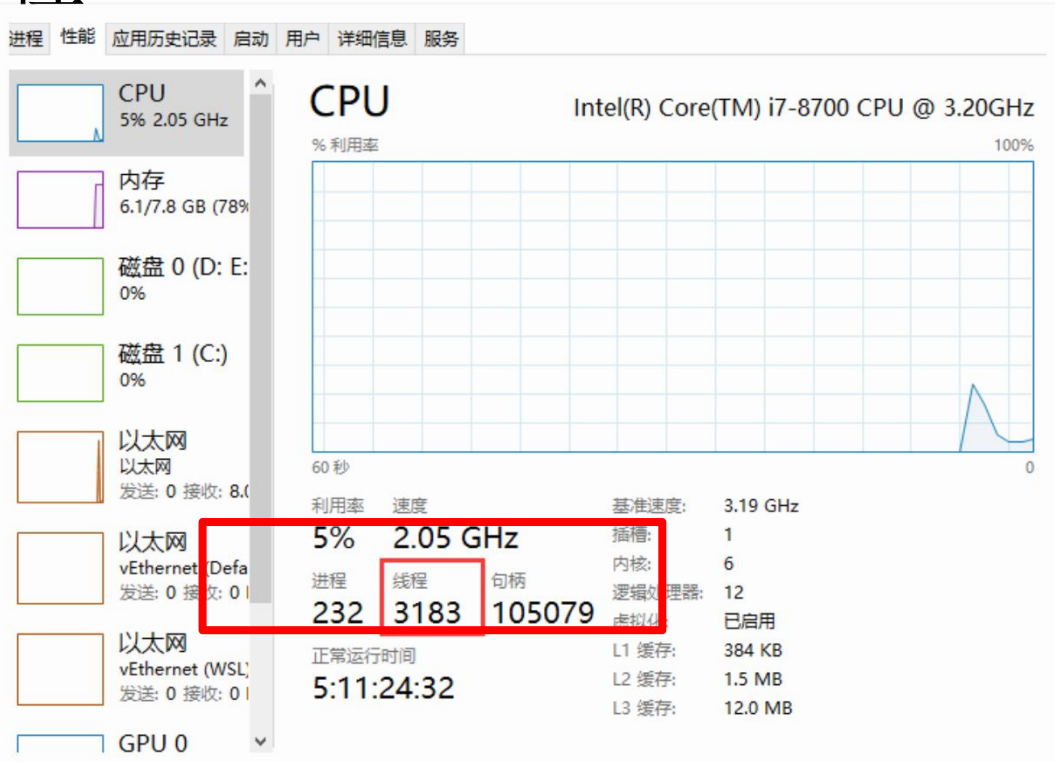


Windows 后台运行着许多进程

什么是线程

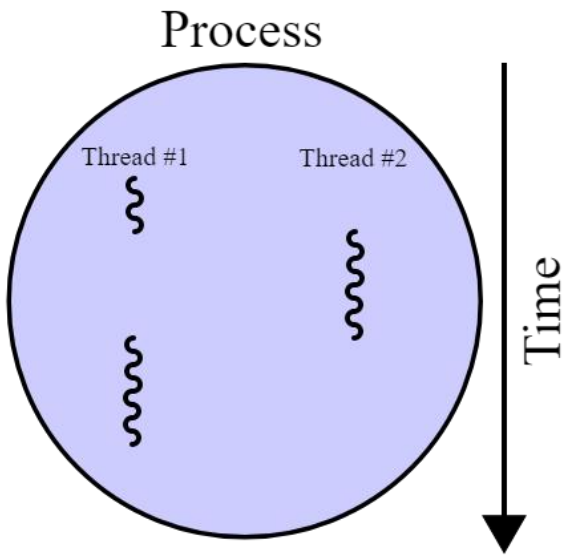
- 线程（Thread）：进程中一个单一顺序的控制流
  - 操作系统能够进行运算调度的最小单位
  - 包含在进程中，是进程的实际运作单位
  - 一个进程可以包含（并发）多个线程
  - 一个进程至少包含一个线程
  - 多个线程之间共享内存





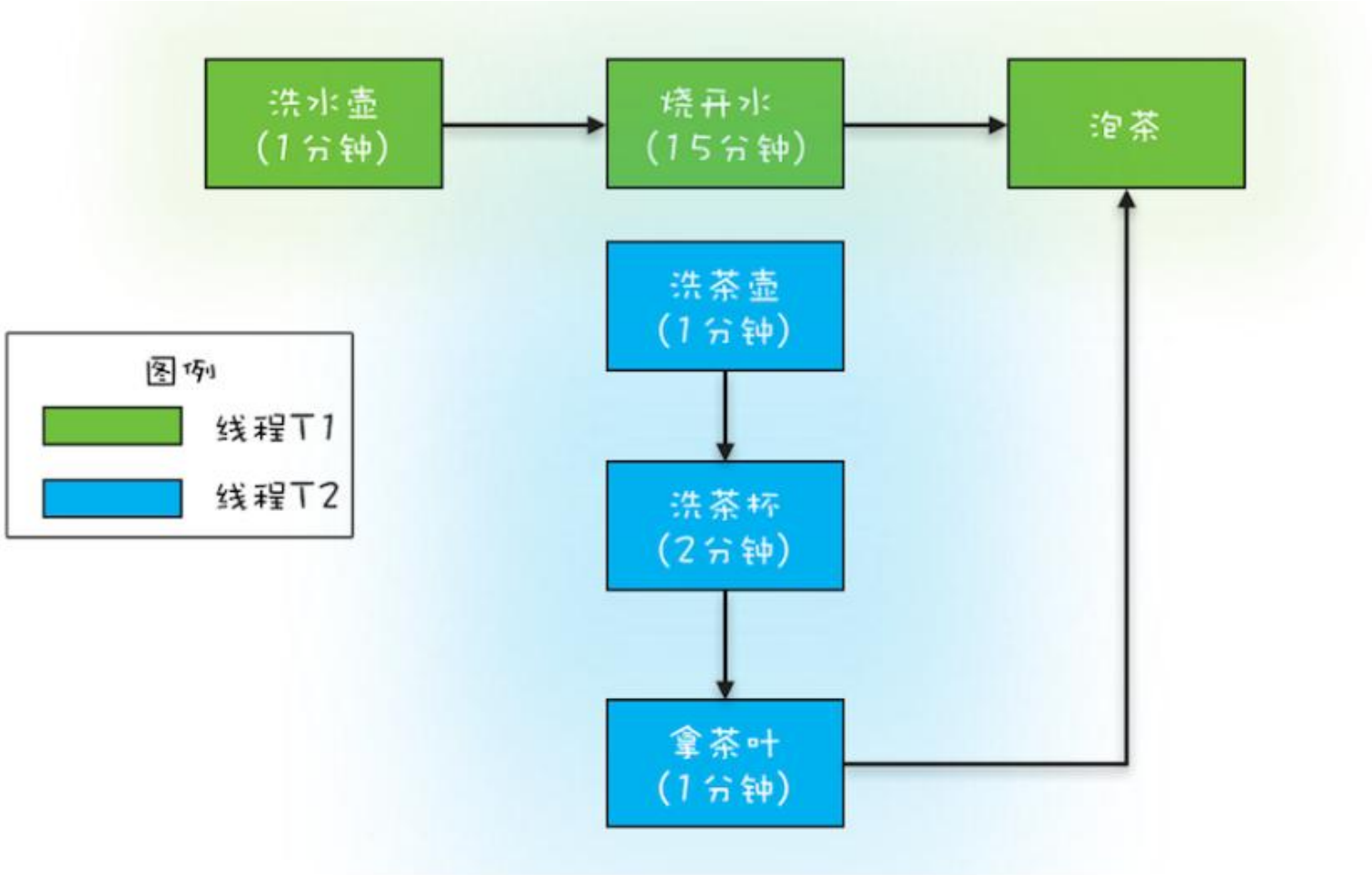
线程与进程

进程（Process）	线程（Thread）
重量级	轻量级
一个应用可以包含多个进程	一个进程可以包含多个线程
多个进程间 不共享内存	一个进程的多个线程间 共享内存
进程表现为 虚拟机	线程表现为 虚拟CPU



多线程

生活中的“多线程”



烧水泡茶最优分工方案

飞机大战中的多线程

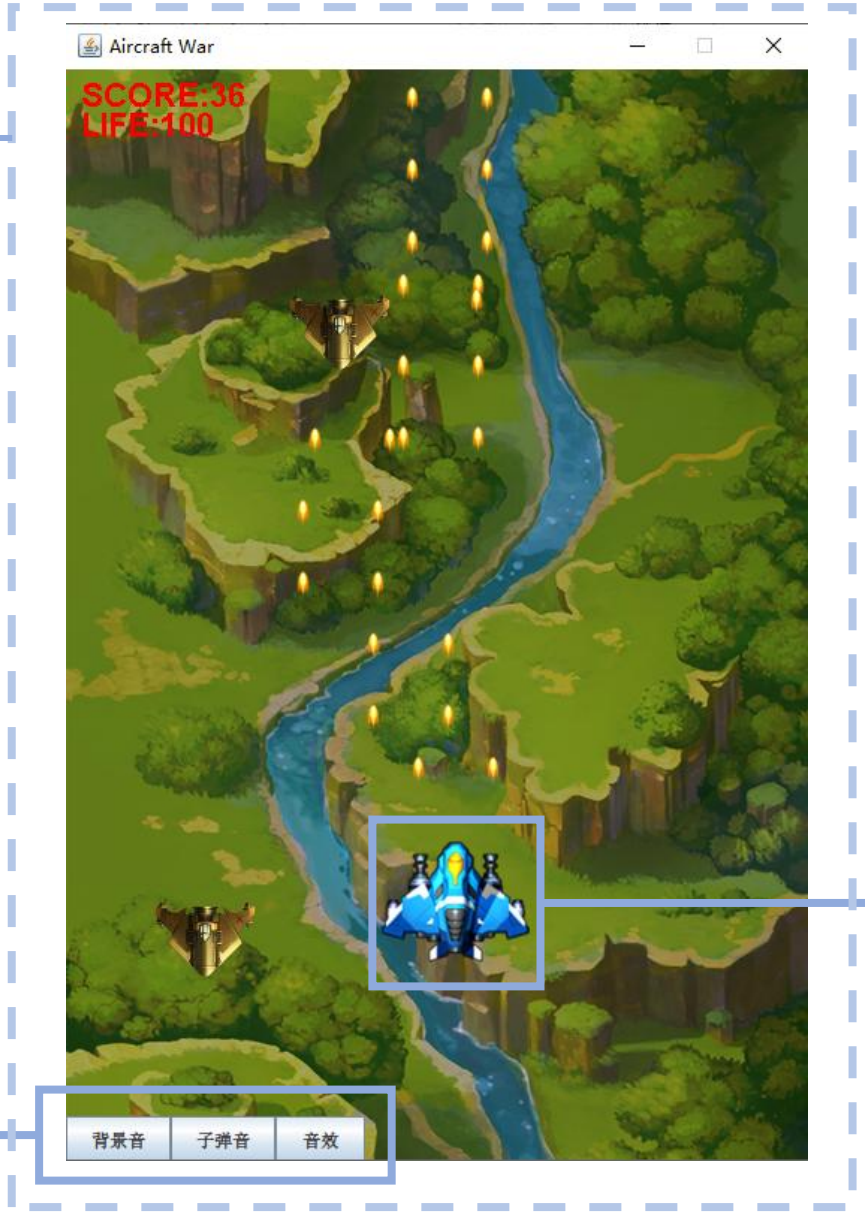
飞机大战应用 进程

游戏控制主线程（waiting）

游戏逻辑线程

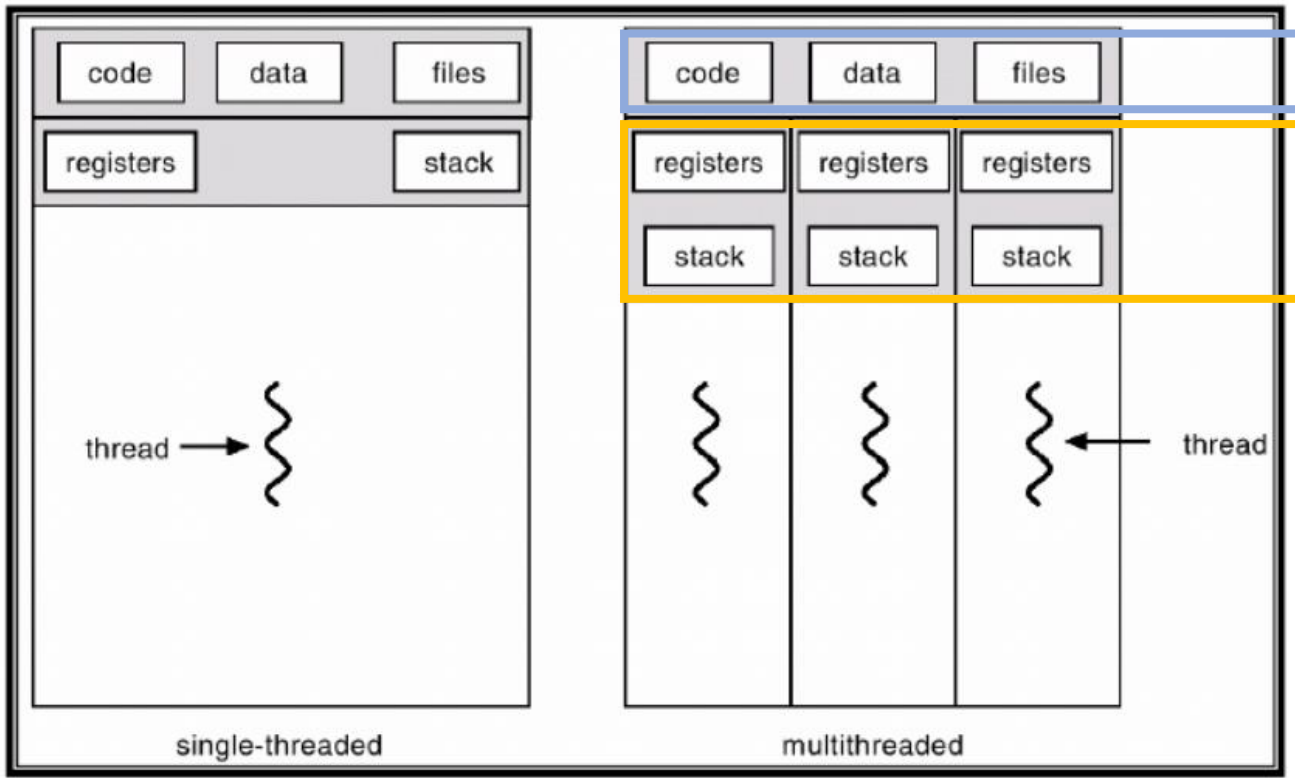
音效控制线程

鼠标/键盘监听线程



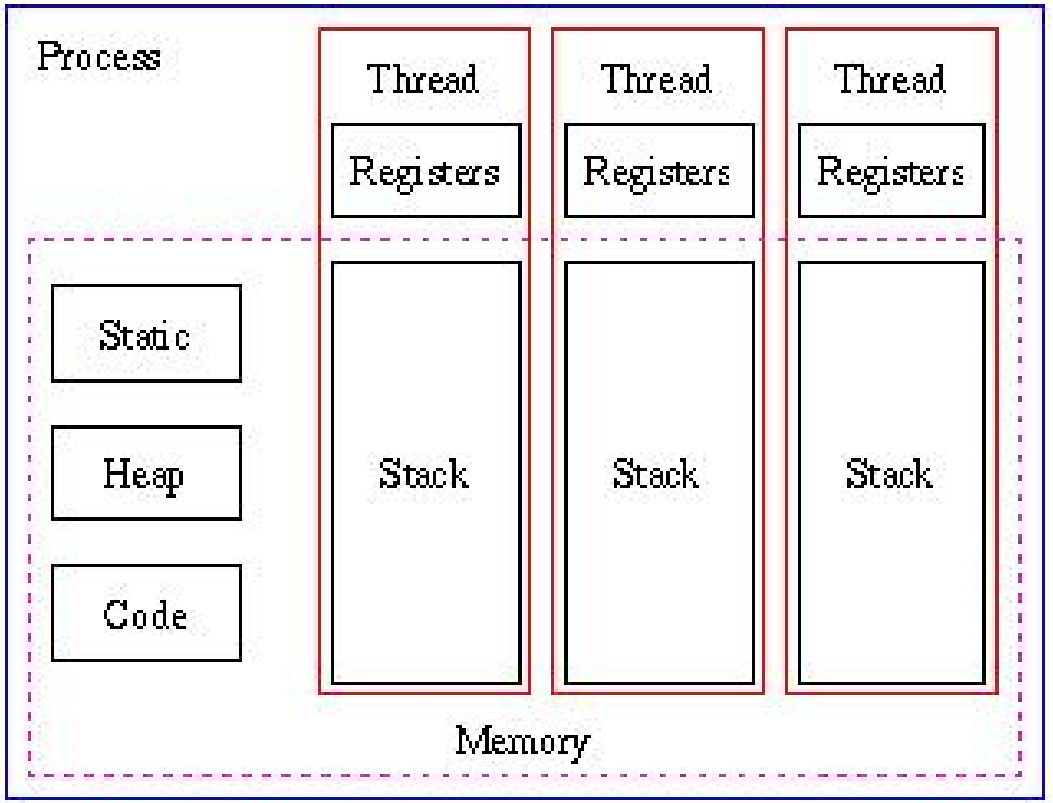
鼠标监听线程

多线程



共用同一个地址空间，
共享内存

不共享寄存器与栈



单选题 1分

设置

是否线程并发的时候都需要使用多线程?

A 是

B 否

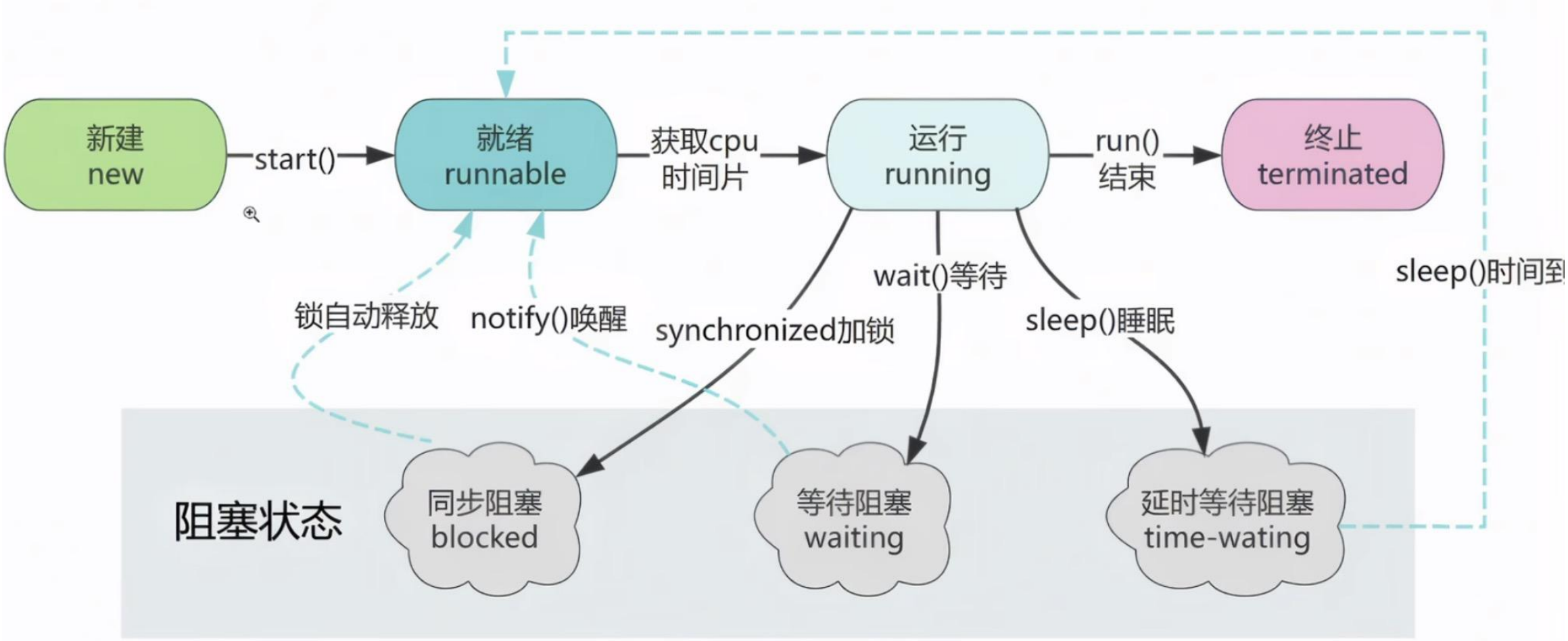
C 不确定

提交

课程导航

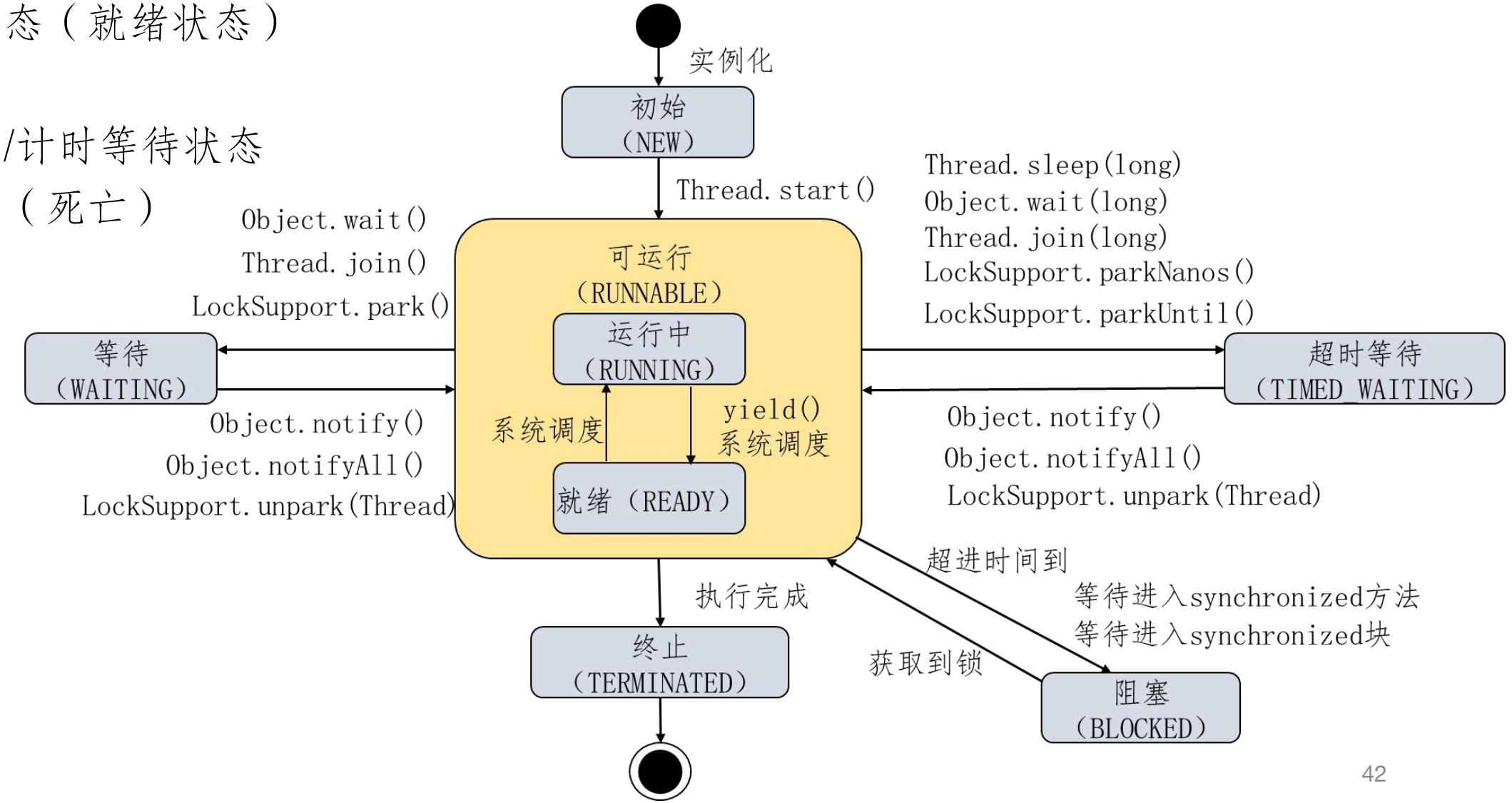
- 进程与线程、多线程
- Java中对线程的控制
- 同步、死锁及如何避免
- 任务创建与线程池
- 多线程应用—生产者与消费者模式

线程的状态（生命周期）


新建
new
start()
就绪
runnable
获取cpu
时间片
运行
running
run()
结束
终止
terminated
sleep()时间到
锁自动释放
notify()唤醒
synchronized加锁
wait()等待
sleep()睡眠
阻塞状态
同步阻塞
blocked
等待阻塞
waiting
延时等待阻塞
time-wating

线程的状态（生命周期）

- 新建状态
- 可运行状态（就绪状态）
- 阻塞状态
- 等待状态/计时等待状态
- 终止状态（死亡）



创建线程

```java
public class Thread1 extends Thread {
    @Override
    public void run() {
        System.out.println("New Thread");
    }
}
```

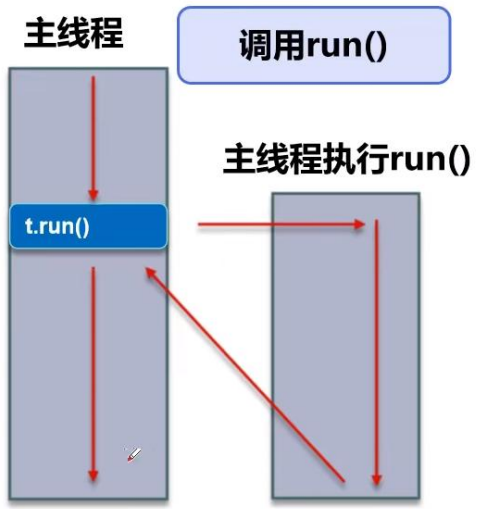
1

```java
public class Thread2 implements Runnable {
    @Override
    public void run() {
        System.out.println("New Thread");
    }
}
```

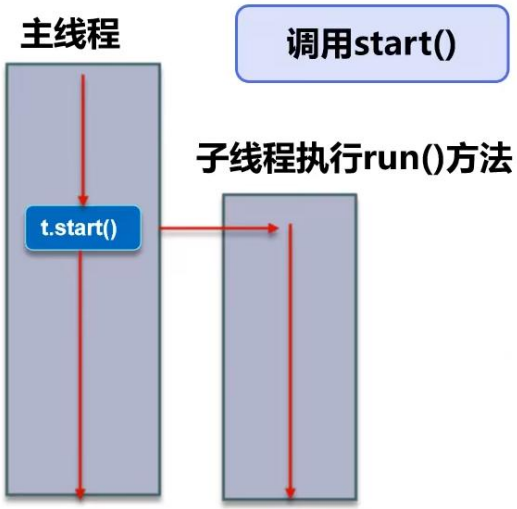
2

- 两种方式
  - 继承 Thread 类
  - 实现 Runnable 接口
- 调用 start 方法
  - 启动线程，将引发调用 run 方法；
  - start方法将立即返回；
  - 新线程将并发运行。
- 注意：不能直接调用 run 方法
  - 直接调用 run 方法只会执行同一个线程中的任务， 而不会启动新线程。

```java
public class Main {
    public static void main(String[] args){
        new Thread1().start();
        new Thread(new Thread2()).start();
    }
}
```



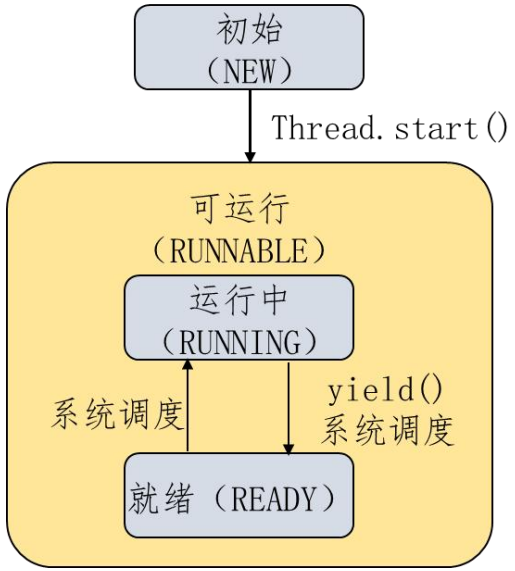
只有主线程一条执行路径



多条执行路径，主线程和子线程并行交替执行

创建线程到可运行状态

- Runnable更常用，其优势在于：
  - 任务与运行机制解耦，降低开销；
  - 更容易实现多线程资源共享；
  - 避免由于单继承局限所带来的影响。



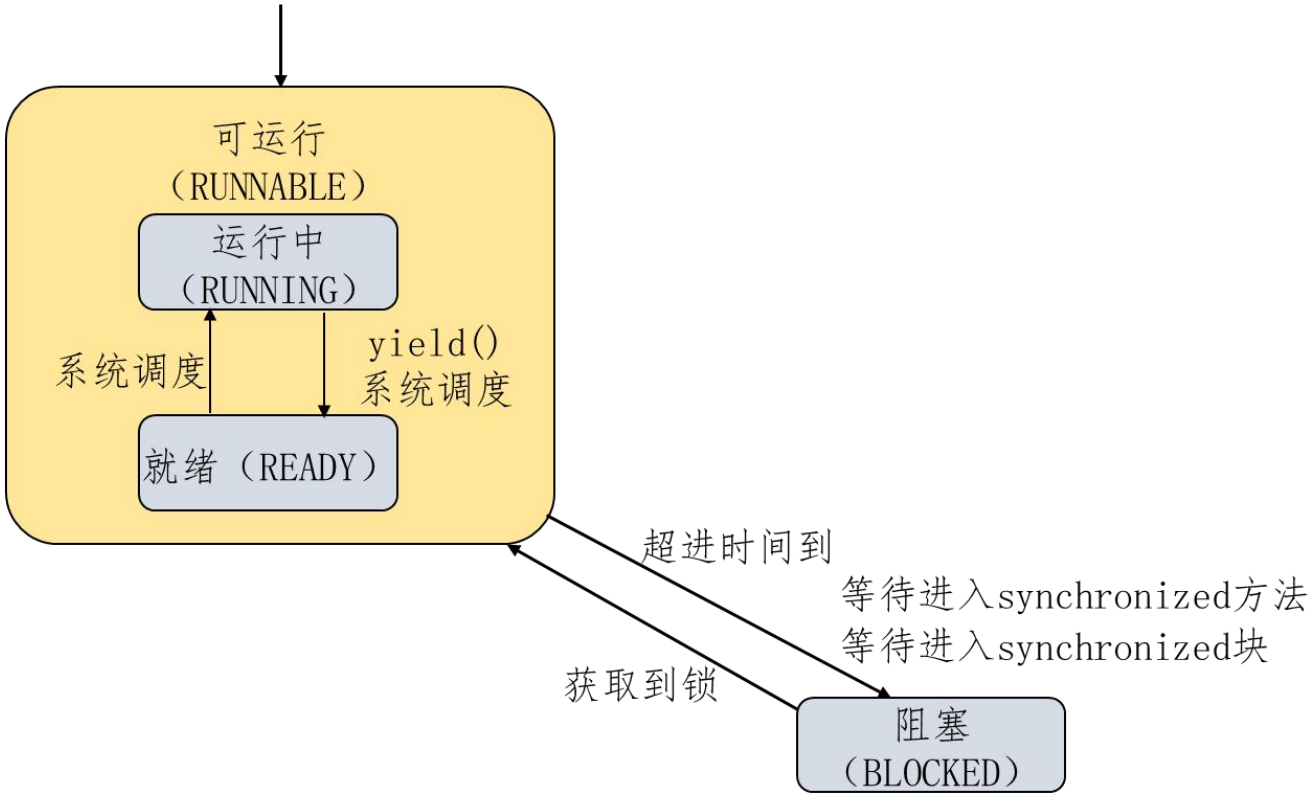
- Java8后引入的lambda语法进一步简写为：
  - 创建并启动线程

```
public static void main(String[] args) {
    //lambda 表达式
    Runnable r = () -> {
        System.out.println("Thread: Runnable with Lambda Expression");

    };
    // 启动线程
    new Thread(r).start();
}
```

线程阻塞

被动进入

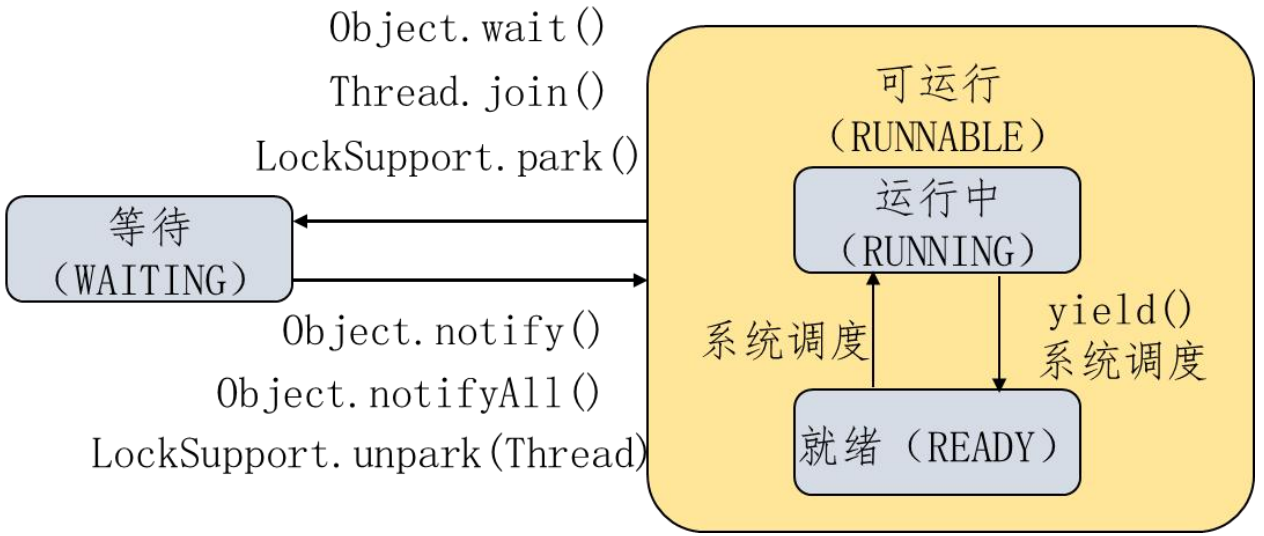
- 线程何时进入阻塞状态：
  - 当一个线程试图获取一个内部的对象锁，而该锁被其他线程持有。
- 线程何时变成非阻塞状态：
  - 当所有其他线程释放该锁，且线程调度器允许本线程持有它的时候。
- 当线程处于被阻塞或等待状态时，它暂时不活动
  - 它不运行任何代码且消耗最少的资源。直到线程调度器重新激活它。



线程等待

主动进入

- 进入该状态表示当前线程需要等待其他线程做出一些的特定的动作（通知或中断）。
- 运行→等待：
  - **情形1：**当前线程对象调用Object.wait()方法，等待被其它线程唤醒；
  - **情形2：**其他线程调用 Thread.join()方法，等其它线程结束后主线程再执行。



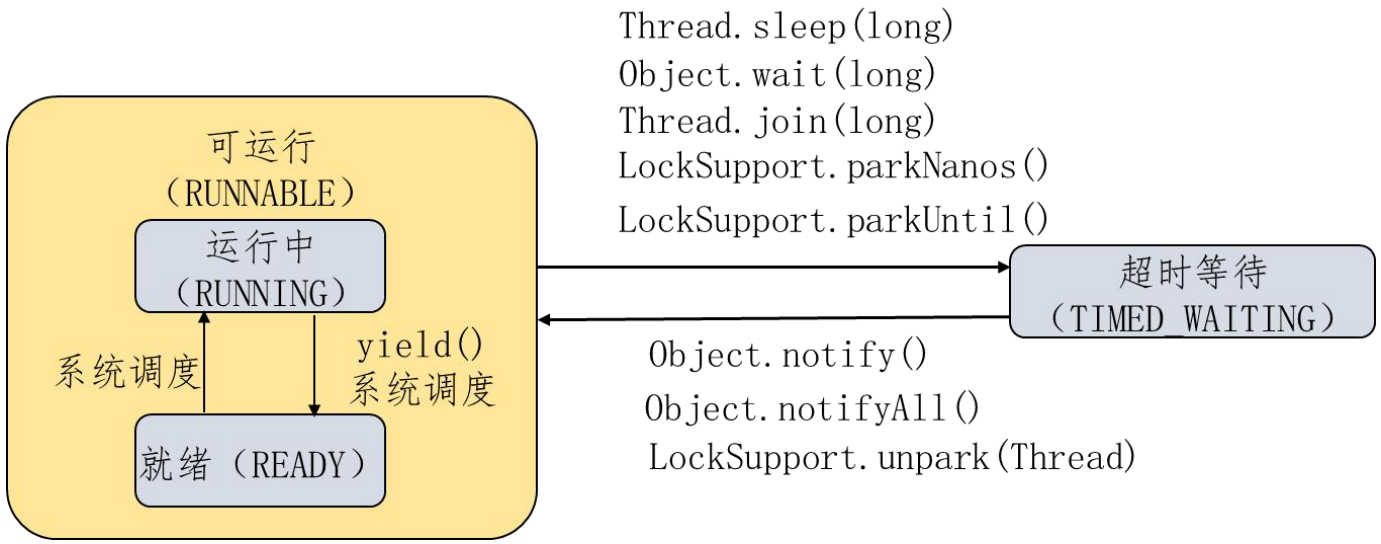
- 等待→运行：
  - **情形1：**等待的线程被其他线程对象唤醒，调用Object. notify()或者Object. notifyAll()。
  - **情形2：**调用 Thread.join()方法的线程结束。

线程计时等待

Sleep v.s. wait:
- sleep方法是当前线程休眠，让出CPU，不释放锁，这是Thread的静态方法；
- wait方法是当前线程等待，释放锁，这是Object的方法.

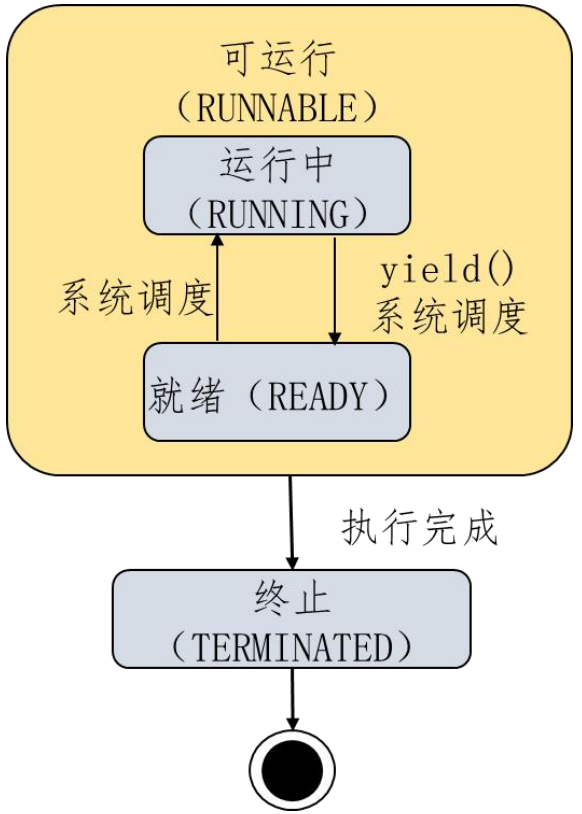
- **运行→计时等待：**
  - **情形1：**当前线程对象调用Object.wait(time)方法；
  - **情形2：**当前线程调用 Thread.sleep(time)方法。
  - **情形3：**其他线程调用 Thread.join(time)方法。

- **计时等待→运行：**
  - 区别于等待状态，它可以在指定的时间自行返回。



终止线程

- 线程会由于以下两个原因之一而终止：
  - run 方法正常退出，线程自然终止；
  - 因为一个没有捕获的异常终止了 run 方法，使线程意外终止。
- **注意：**stop 方法已被废弃
  - 该方法抛出 ThreadDeath 错误对象，由此杀死线程。
  - 调用线程的 stop 方法会终止此线程，但是由于stop方法强行终止线程，是**极端不安全的**。



中断线程

- 当对一个线程调用 interrupt 方法时，线程的中断状态将被置位
  - 这是每一个线程都具有的 boolean 标志（true：打断了；false：没有打断）。
  - 每个线程都应该不时地检查此标志，以判断线程是否被中断。
- 当在一个等待状态线程上调用 interrupt 方法
  - 等待状态（即sleep或wait调用）将会被InterruptedException 异常中断。
- 中断一个线程不过是引起它的注意
  - 被中断的线程可以决定如何响应中断。

线程优先级

- 每一个线程都有一个优先级
  - 默认情况下，一个线程继承它的父线程的优先级；
  - 可以用 setPriority(int newPriority) (1~10, 默认值5) 方法提高或降低任何一个线程的优先级。
- 每当调度器决定运行一个新线程时，会尽可能在具有高优先级的线程中进行选择，尽管这样会使低优先级的线程完全饿死
- 线程优先级是高度依赖于系统的
  - 在 Oracle 为 Linux 提供的 Java 虚拟机中，线程的优先级被忽略。

守护线程

- 使用 setDaemon 方法标识该线程为守护线程或用户线程。
  - thread.setDaemon(true); // true:守护线程
- 守护线程的唯一用途是为其他线程提供服务。
  - 例子：计时线程。它定时地发送“计时器嘀嗒” 信号给其他线程。
  - 守护线程最典型的应用就是 GC (垃圾回收器)，它就是一个很称职的守护者。
- 守护线程的结束，是在run方法运行结束后或main函数结束后。
- 必须在调用 start() 方法之前调用此方法，否则会抛出 IllegalThreadStateException 异常。

线程的相关方法总结

主要总结Thread类中的核心方法

方法名称	是否static	方法说明
start()	否	让线程启动，进入就绪状态,等待cpu分配时间片
run()	否	重写Runnable接口的方法,线程获取到cpu时间片时执行的具体逻辑
sleep(time)	是	线程休眠固定时间，进入阻塞状态，休眠时间完成后重新争抢时间片,休眠可被打断
join()/join(time)	否	调用线程对象的join方法，调用者线程进入阻塞,等待线程对象执行完或者到达指定时间才恢复，重新争抢时间片
isInterrupted()	否	获取线程的打断标记，true:被打断，false：没有被打断。调用后不会修改打断标记
interrupt()	否	打断线程，抛出InterruptedException异常的方法均可被打断，但是打断后不会修改打断标记，正常执行的线程被打断后会修改打断标记
interrupted()	否	获取线程的打断标记。调用后会清空打断标记
currentThread()	是	获取当前线程

线程的相关方法总结

Object中与线程相关方法

方法名称	方法说明
wait()/wait(long timeout)	获取到锁的线程进入阻塞状态
notify()	随机唤醒被wait()的一个线程
notifyAll();	唤醒被wait()的所有线程，重新争抢时间片

课程导航

- 进程与线程、多线程
- Java中对线程的控制
- 同步、死锁及如何避免
- 任务创建与线程池
- 多线程应用—生产者与消费者模式

线程同步

- 当多个线程同时运行时，线程的调度由操作系统决定，程序本身无法决定。因此，任何一个线程都有可能在任何指令处被操作系统暂停，然后在某个时间段后继续执行。
- 如果多个线程同时读写共享变量，会出现数据不一致的问题。

```
class Counter {
    public static int count = 0;
}
class AddThread extends Thread {
    public void run() {
        for (int i=0; i<500; i++) { Counter.count += 1; }
    }
}
```

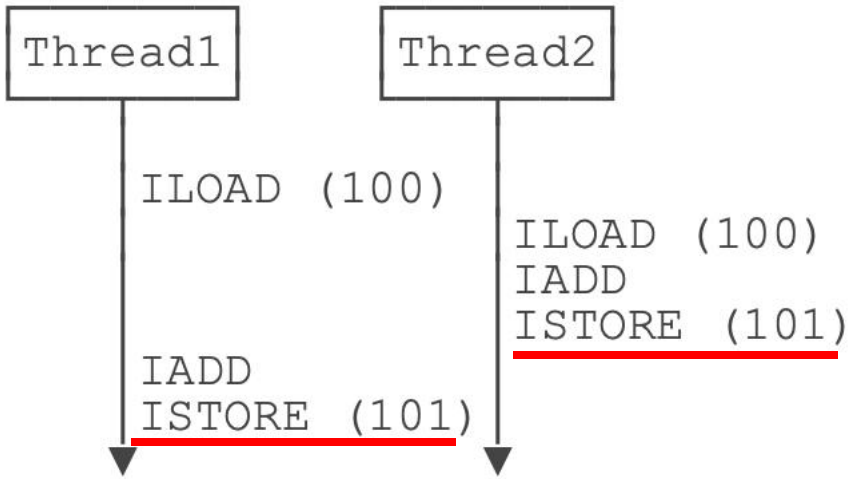
```
class DecThread extends Thread {
    public void run() {
        for (int i=0; i<500; i++) { Counter.count -= 1; }
    }
}
```

线程同步

- 原子操作：指不能被中断的一个或一系列操作。即要么完全执行，要么完全不执行，不存在执行了一半的情况。

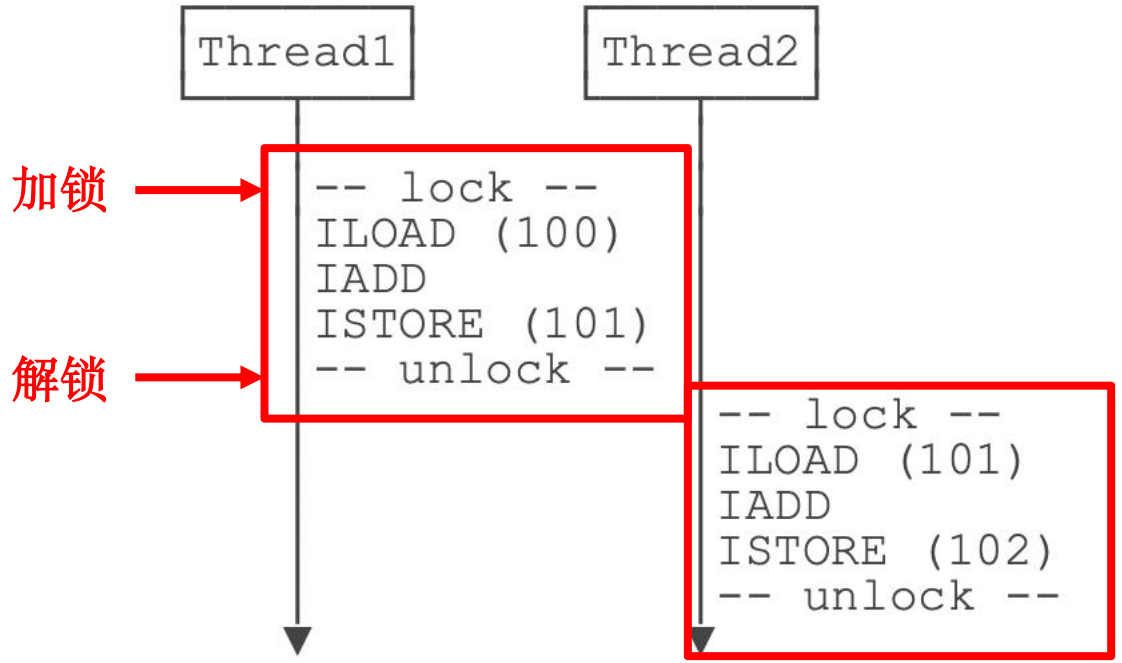
- 例子：n = n + 1; 看上去是一行语句，实际上对应了3条指令

ILOAD	取数
IADD	加法运算
ISTORE	结果保存



线程同步

- 这说明多线程模型下，要保证逻辑正确，对共享变量进行读写时，必须保证一组指令以原子方式执行：即某一个线程执行时，其他线程必须等待。



通过加锁和解锁的操作，保证3条指令总是在一个线程执行期间，不会有其他线程会进入此指令区间。

线程同步

- 代码实现：
  - Java程序使用synchronized关键字对一个对象进行加锁。

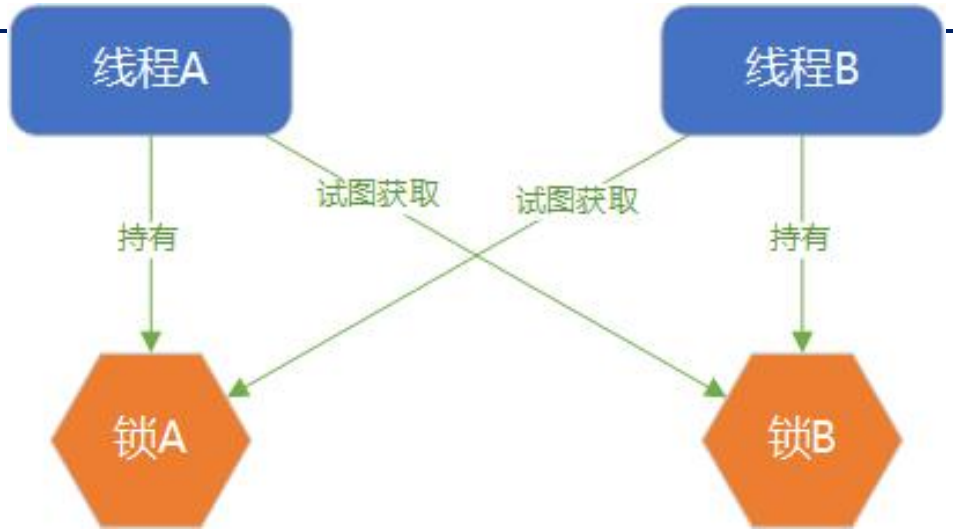
```
synchronized(lock) {
    n = n + 1;
}
```

  - 改写之前的代码

```
class AddThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {// 获取锁
                Counter.count += 1;
            } // 释放锁
}}}
```

```
class DecThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {// 获取锁
                Counter.count -= 1;
            } // 释放锁
}}}
```

线程死锁



- **死锁**:在**线程**A持有**锁**A并想获得**锁**B的同时，**线程**B持有**锁**B并尝试获得**锁**A，那么这两个线程将永远地等待下去。

- **避免死锁**：线程获取锁的顺序要一致。即严格按照先获取锁A，再获取锁B的顺序，或者先获取锁B，再获取锁A的顺序。

线程死锁

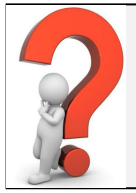
```
class PerA implements Runnable{
    public void run() {
        try {
            System.out.println( " PerA: 我有蓝色钥匙请给
我红色钥匙");
            while(true){
                synchronized (DeadLock.bluekey) {
                    System.out.println(" PerA 锁住蓝色钥匙");
                    Thread.sleep(3000); // 此处等待是给PerB机会
                    synchronized (DeadLock.redkey) {
                        System.out.println(" PerA 拿两把钥匙开锁");
                        Thread.sleep(60 * 1000); // 为测试
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
class PerB implements Runnable{
    public void run() {
        try {
            System.out.println(" PerB：我有红色钥匙请给
我蓝色钥匙");
            while(true){
                synchronized (DeadLock.redkey) {
                    System.out.println(" PerB 锁住红色钥匙");
                    Thread.sleep(3000); // 此处等待是给PerA机会
                    synchronized (DeadLock.bluekey) {
                        System.out.println(" PerB拿两把钥匙开锁");
                        Thread.sleep(60 * 1000); // 为测试
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

课程导航

- 进程与线程、多线程
- Java中对线程的控制
- 同步、死锁及如何避免
- 任务创建与线程池
- 多线程应用—生产者与消费者模式

Callable接口

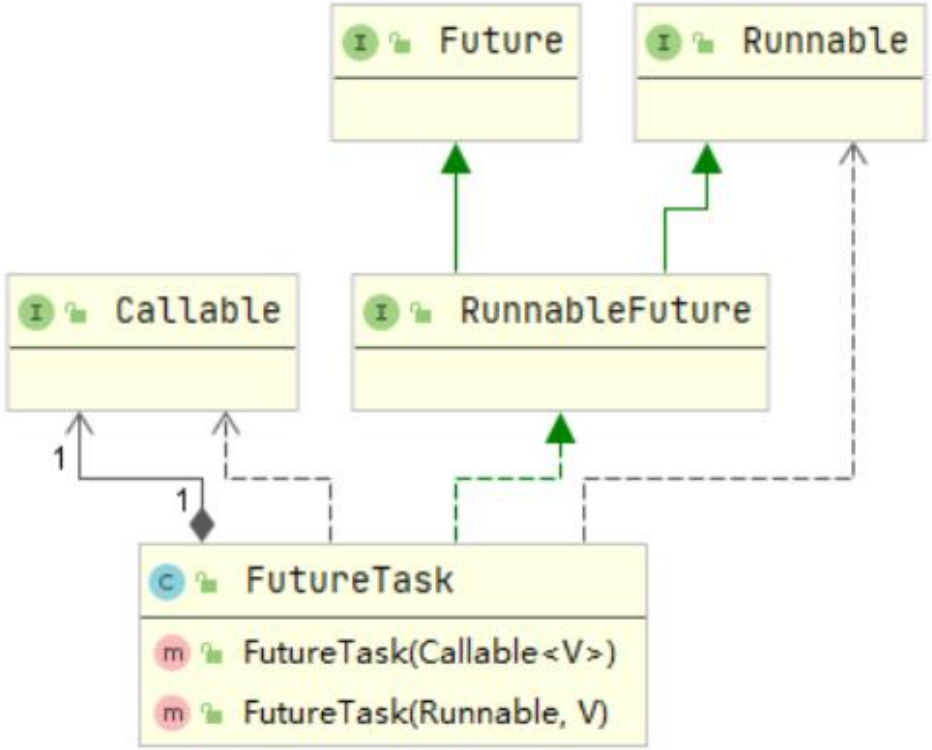


前面的多线程方法在执行完任务后无法取得执行结果，如果需要获取执行结果，就必须通过共享变量或者使用线程通信的方式来达到效果，有更简单的方式吗？

- Runnable：封装一个异步运行的任务，没有参数和返回值
- Callable：封装一个异步运行的任务，有返回值
  - 需实现 call 方法
  - Callable<Integer> 表示返回 Integer 对象的异步计算任务
  - **可以抛出异常**
  - 不能直接进行线程操作，也不能传入Thread

FutureTask类

- FutureTask：执行 Callable的一种方法，位于java.util.concurrent包中
  - 控制 Callable 任务的执行，获得其运算结果
  - 间接实现了Runnable接口，进而可以通过Thread启动线程



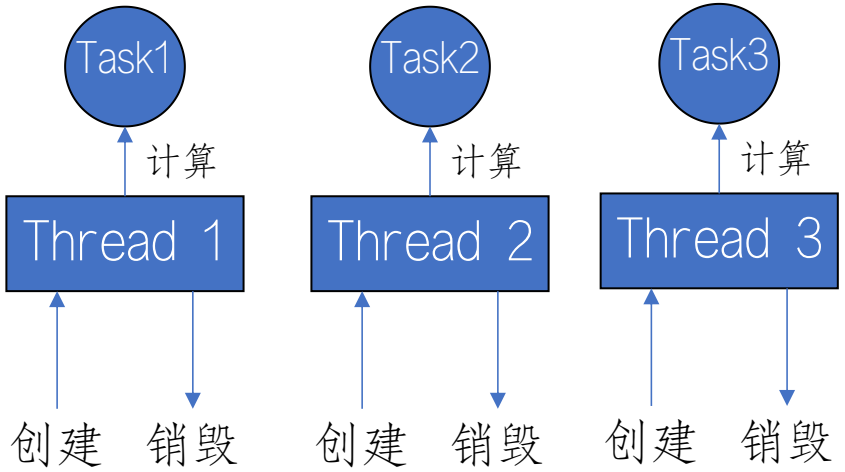

```java
// 实例化 Callable 任务，指定返回类型为 String
Callable<String> callable = new Callable<String>() {
    public String call() throws Exception {
        // balabalabala~
        return "Hello world ~";
    }
};

// 将 callable 任务委派给 FutureTask
FutureTask<String> task = new FutureTask<String>(callable);

// FutureTask实现了Runnable接口，所以可以直接传给 Thread 启动线程
new Thread(task).start();

// 通过 FutureTask 获取返回值
String call = task.get();

System.out.println(call); // print: Hello world ~
```

线程池

频繁创建/终止大量线程，会带来大量开销，怎么办？

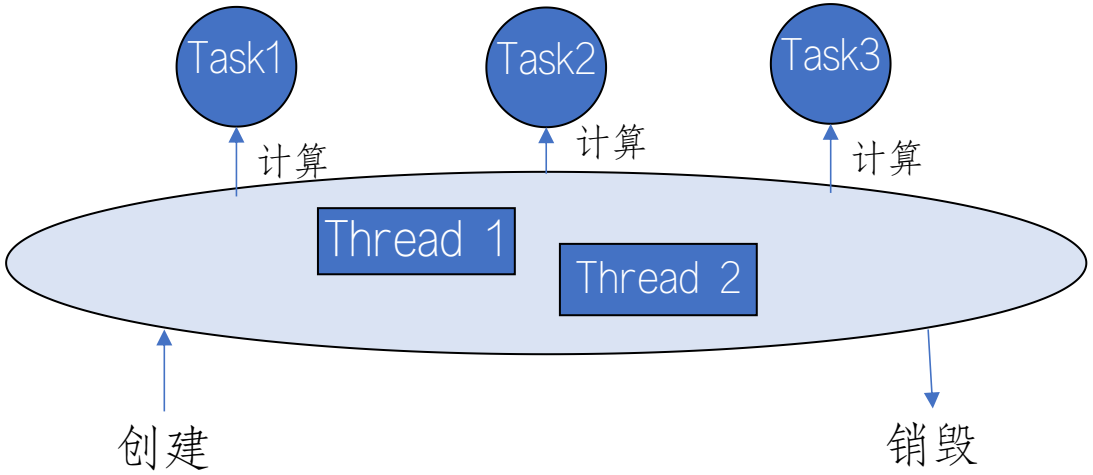
池化技术能够减少资源对象的创建次数，提高程序的性能，特别是在高并发下这种提高更加明显。

是对资源的充分利用！



为每个任务创建一个线程，任务完成后线程销毁

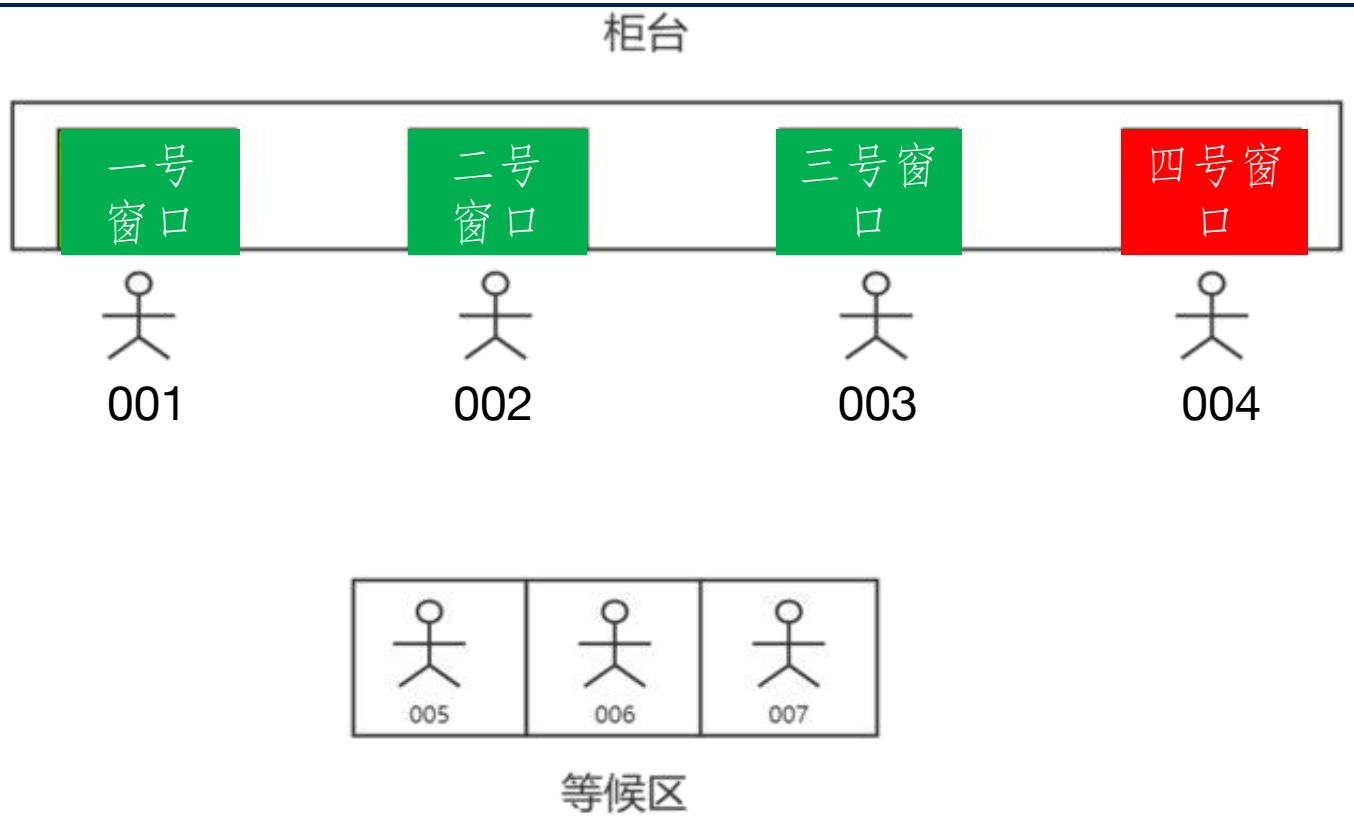
类比：每次有活时，招募工人，干完活遣散工人
再有活时，再招募工人



提前在线程池中创建线程，任务来时线程执行任务，执行完毕后线程休息，等待下一个任务

类比：包工头提前招募若干工人，有活时安排工人工作，
工人完成工作后休息，等待下一次被安排工作

线程池



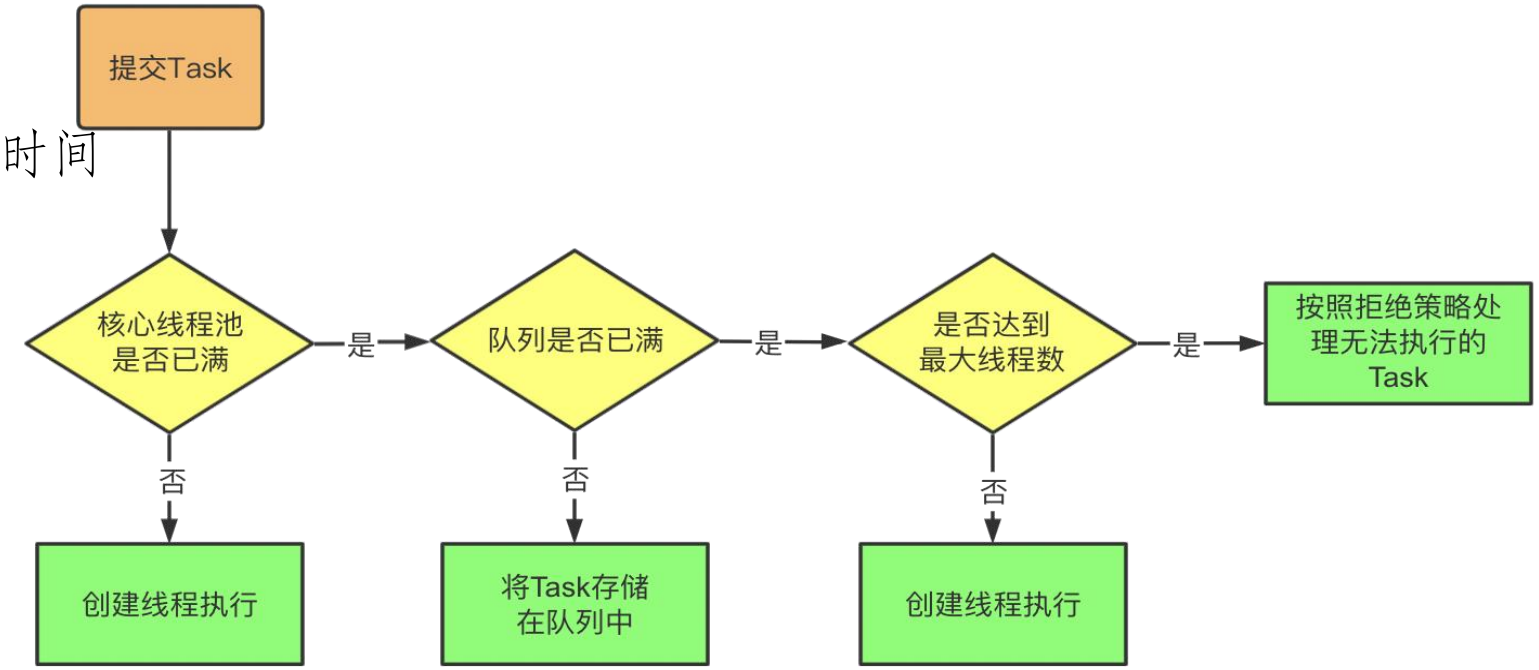
使用线程池的优点：

- **降低资源消耗。** 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。** 当任务到达时，任务可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性。** 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配、调优和监控。

线程池

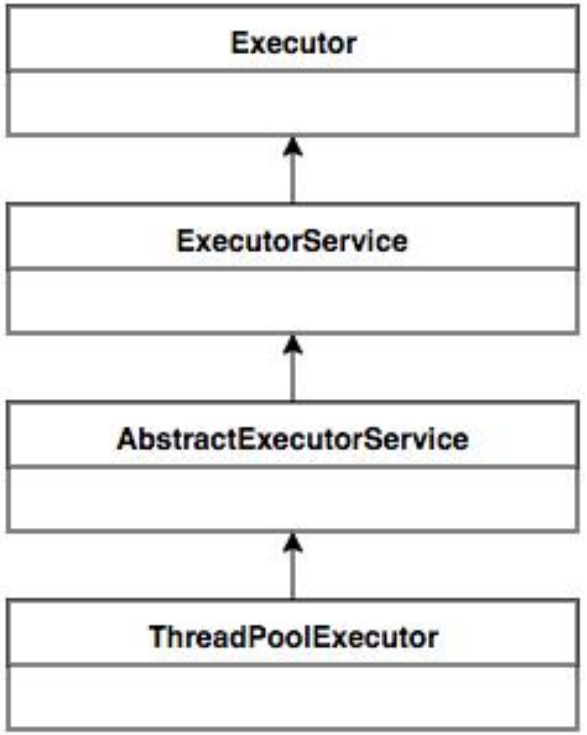
- ThreadPoolExecutor 线程池主要参数：
  - corePoolSize：
    线程池中所保存的核心线程数
  - maximumPoolSize：
    线程池中允许的最大线程数
  - keepAliveTime：
    线程池中的空闲线程所能持续的最长时间
  - workQueue：阻塞队列
  - threadFactory：
    创建线程的工厂，主要定义线程名
  - handler：拒绝策略

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



线程池

- 线程池的继承关系
  - Executor：是一个接口，将业务逻辑提交与任务执行进行分离
  - ExecutorService：接口继承了Execute，做了shutdown()等业务的扩展，可以是说真正的线程池接口
  - AbstractExecutorService：抽象类实现了ExecutorService接口的大部分方法
  - ThreadPoolExecutor：线程池的核心实现类，用来执行被提交的任务



线程池的多任务控制

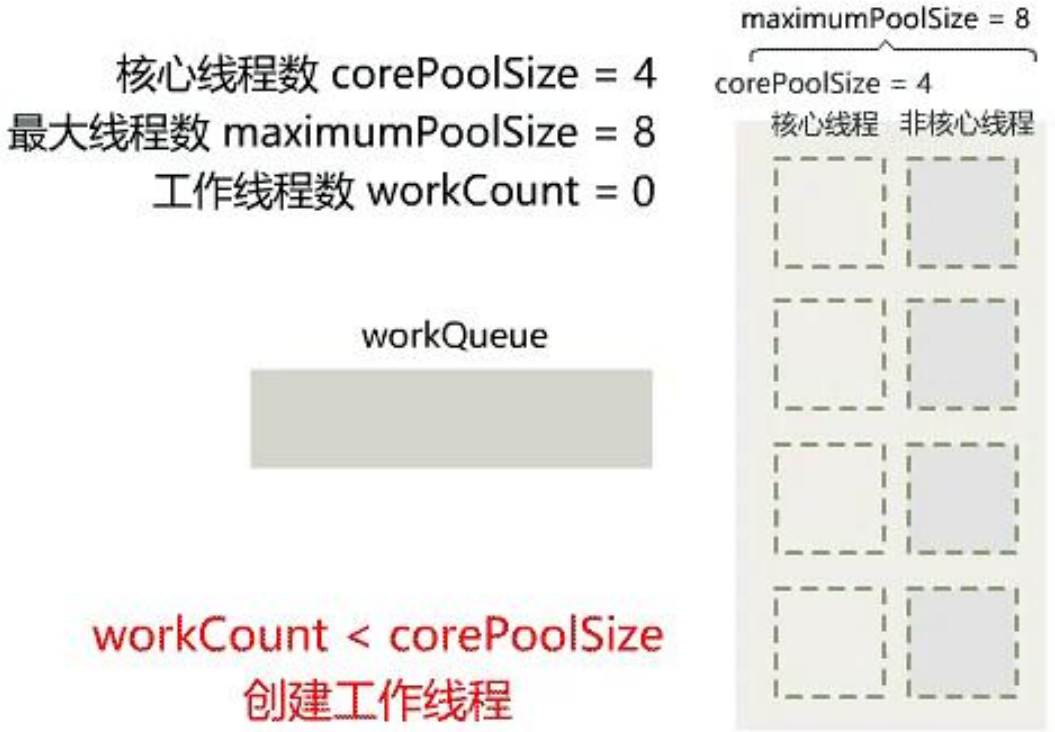
1. 首先检测线程池运行状态，如果不是RUNNING，则直接拒绝，线程池要保证在RUNNING的状态下执行任务。

2. 如果workerCount < corePoolSize，则创建并启动一个线程来执行新提交的任务。

3. 如果workerCount >= corePoolSize，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中。

4. 如果workerCount >= corePoolSize && workerCount < maximumPoolSize，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务。

5. 如果workerCount >= maximumPoolSize，并且线程池内的阻塞队列已满，则根据拒绝策略来处理该任务，默认的处理方式是直接抛异常。



课程导航

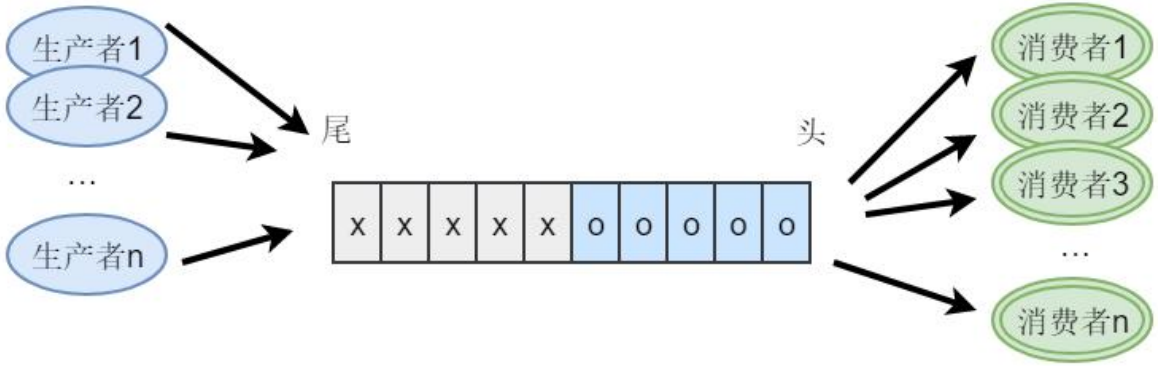
- 进程与线程、多线程
- Java中对线程的控制
- 同步、死锁及如何避免
- 任务创建与线程池
- 多线程应用—生产者与消费者模式

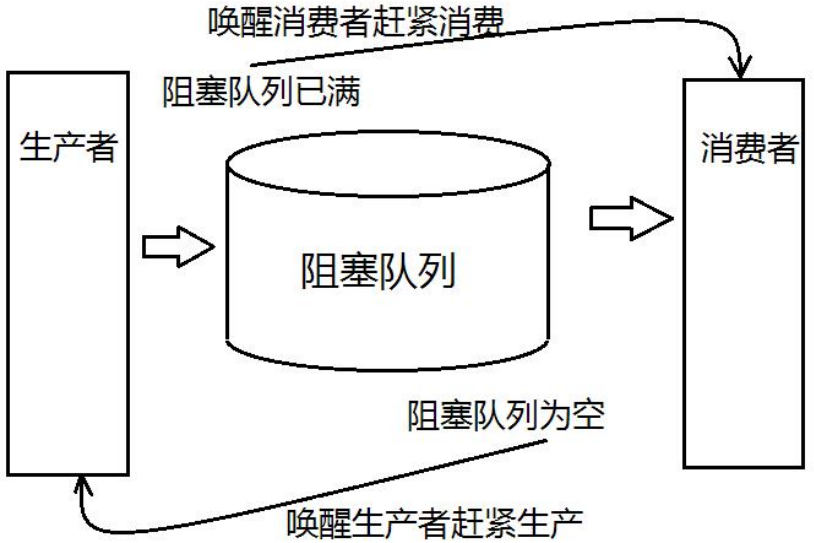
生产者-消费者设计模式

- 在实际的软件开发过程中，经常会碰到如下场景：

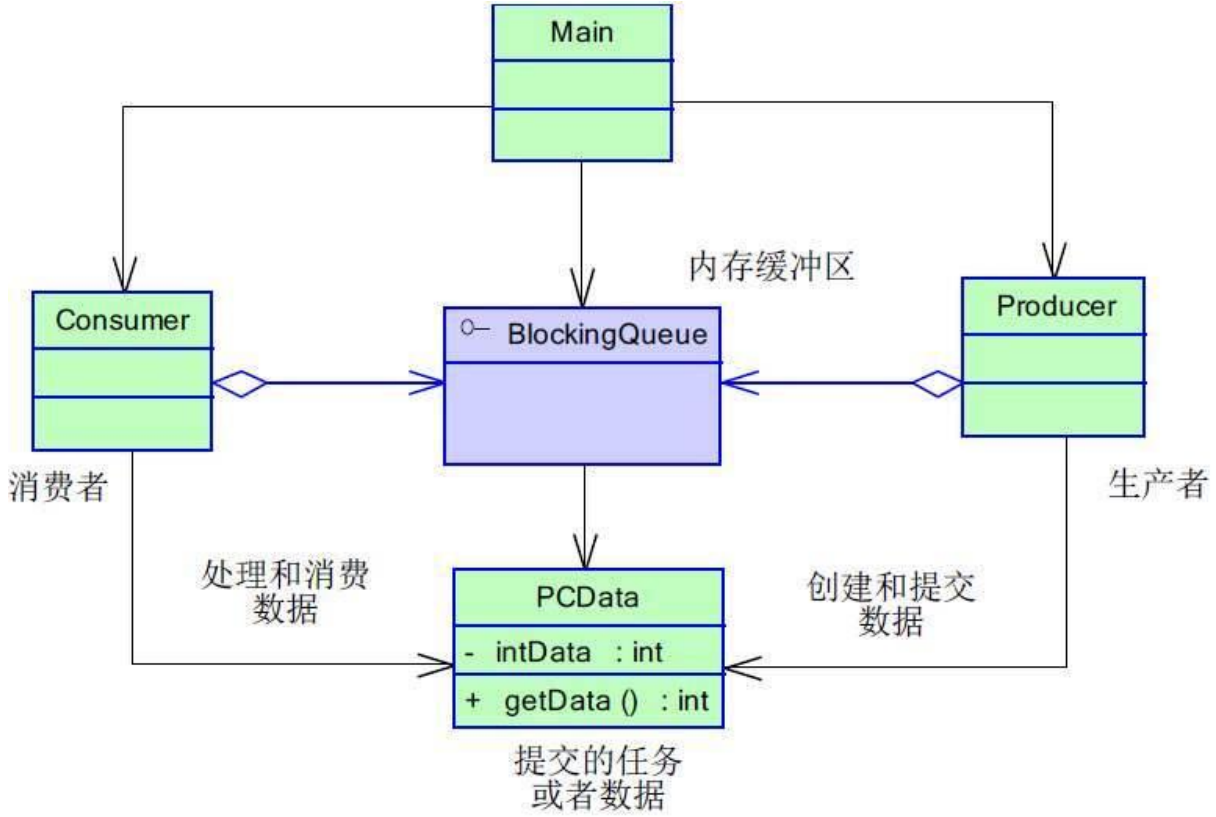
  某个模块负责产生数据，这些数据由另一个模块来负责处理（此处的模块是广义的，可以是类、函数、线程、进程等）。

- 产生数据的模块，就形象地称为生产者；而处理数据的模块，就称为消费者。
- 如果生产者生产速度过快，消费者消费的很慢，并且缓存区达到了最大时。缓存区会阻塞生产者，让生产者停止生产，等待消费者消费了数据后，再唤醒生产者
- 当消费者消费速度过快时，缓存区为空时。缓存区则会阻塞消费者，待生产者向队列添加数据后，再唤醒消费者。





生产者-消费者设计模式



- 优点：
  - 并发（异步）：生产者和消费者各司其职，生产者和消费者都只需要关心缓冲区，不需要互相关注，通过异步的方式支持高并发，将一个耗时的流程拆成生产和消费两个阶段。
  - 解耦：生产者和消费者进行解耦（通过缓冲区通讯）。

主观题 1分

设置

如何保证缓冲区数据的一致性?

正常使用主观题需2.0以上版本雨课堂

作答

生产者-消费者设计模式

- **注意：**
  - 永远在synchronized的方法或对象里使用wait、notify和notifyAll，不然Java虚拟机会生成 IllegalMonitorStateException。
  - 永远在while循环里而不是if语句下使用wait。这样，循环会在线程睡眠前后都检查wait的条件，并在条件实际上并未改变的情况下处理唤醒通知。
  - 永远在多线程间共享的对象（在生产者消费者模型里即缓冲区队列）上使用wait。

```
synchronized (monitor) {
    // 判断条件是否得到满足
    while(!locked) {
        // 等待唤醒
        monitor.wait();
    }
    // 处理其他的业务逻辑
}
```

prodconsumer

生产者-消费者设计模式

Java代码实现缓冲区

```
public class Buffer {
    private List<Integer> data = new ArrayList<>();
    private static final int MAX = 2;
    private static final int MIN = 0;
    public void put(int value){
        while (true){
            try {//模拟生产数据
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (this){
//当容器满的时候，producer处于等待状态
                while (data.size() == MAX){
                    System.out.println("buffer is full,waiting ...");
                    try {
                        wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
//没有满，则继续produce
                System.out.println("producer--"+
Thread.currentThread().getName()+"--put:" +
value);
                data.add(value);
//唤醒其他所有处于wait()的线程，包括消费者和生产者
                notifyAll();
            }
        }
    }
```

生产者-消费者设计模式

Java代码实现缓冲区

```
public Integer take(){
        Integer val = 0;
        while (true){
            try {{//模拟消费数据
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (this){
                //如果容器中没有数据，consumer处于等待状态
                while (data.size() == MIN){
                    System.out.println("buffer is
empty,waiting ...");
                    try {
                        wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                //如果有数据，继续consume
                        val = data.remove(0) ;
                        System.out.println("consumer--"+
Thread.currentThread().getName()+"--take:" +
val);

                //唤醒其他所有处于wait()的线程，包括消费者和生
产者
                        notifyAll();
                    }
                }
            }

        }
```

生产者–消费者设计模式

Java代码实现

生产者

```
public class Producer implements Runnable{
    private Buffer buffer;
    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }
    @Override
    public void run() {
        buffer.put(new Random().nextInt(100));
    }
}
```

生产者–消费者设计模式

Java代码实现

消费者

```
public class Consumer implements Runnable{
    private Buffer buffer;
    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }
    @Override
    public void run() {
        Integer val = buffer.take();
    }
}
```