

第十二章 线程基础

天行健，君子以自强不息 地势坤 君子以厚德载物 -- 《周易》

12.1 线程示例

```

1
2 public class Test {
3
4     public static void main (String[] args) {
5
6         while(true) {
7             System.out.println("天行健, 君子以自强不息");
8         }
9
10        |
11        while(true) {
12            System.out.println("地势坤, 君子以厚德载物");
13        }
14
15    }
16 }
17

```

我们已经知道，在Java中无法执行到的代码，是无法通过编译的！

所以这里报错！ 因为上面的循环是一个死循环！！！！

因为以上的代码是一个“单线程”程序，单线程程序的特点就是，必须等待上面的代码执行完，下面的代码才能执行。如果上面的代码是一个死循环，则下面的代码永无执行之日！

那么多线程程序可以绕过这个限制，也就是说多线程程序，可以不用等待上面的代码执行完毕，下面的代码就可以先执行了。如下：

```

1  Test.java
2  public class Test {
3
4      public static void main (String[] args) {
5
6          // 创建一个线程，同时把线程要执行的任务也传进去
7          // 线程要执行的任务，就是以下run方法中的代码。
8          Thread t = new Thread(new Runnable() {
9              public void run() {
10
11                  while(true) {
12                      System.out.println("天行健，君子以自强不息。");
13                  }
14
15              }
16          });
17          // 启动线程，就会让线程去执行run方法中的任务
18          t.start();
19
20
21          while(true) {
22              System.out.println("地势坤 君子以厚德载物");
23          }
24      }
25  }
26
27

```

该例子告诉我们，多线程程序，就是可以有多个执行线索，这多个执行线索可以同时执行！

第一个while是由t线程执行
第二个while是由main线程执行的。
这是两个线程同时执行！

[illegible]

12.2 线程相关概念

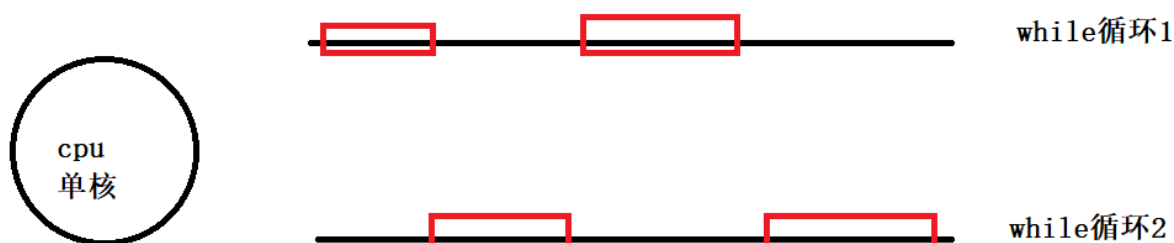
进程：一个正在运行当中的程序，该程序有独立的内存空间。进程和进程之间不会有共享的内存。

线程：一个进程中可以有多个线程，每一个线程都是一个执行线索（都有自己的执行任务）。多个线程可以同时执行。

并发和并行：

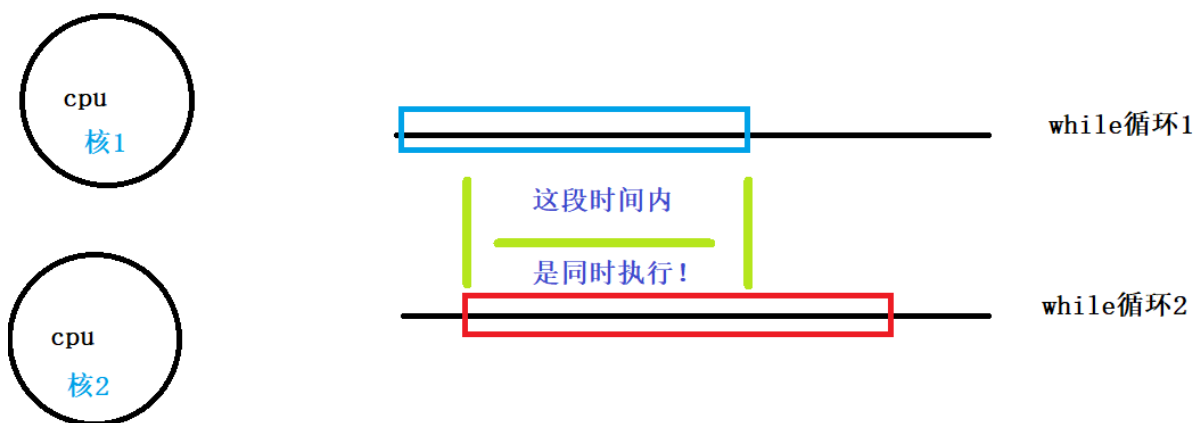
哪怕是单核计算机，也支持多线程。这就是并发

并发



针对于多核CPU，那才是真正意义上的同时执行，这就是并行

并行



CPU时间片

一个核在执行多个线程时，会快速在多个线程之间切换，那么一个核执行一个线程时，到底要执行多久才切换呢？这取决于CPU为该线程分配了多少个CPU时间片，一个CPU时间片就是几十毫秒。也就是说，在一个核即将要执行某一个线程之前，会先规划好要执行多久。执行多久是如何规则的呢？是随机的！这样的执行算法术语上叫：时间片轮法。

注意，后续为了好理解，我们经常会说，线程在抢占CPU，哪个线程抢到了CPU，就能执行代码。实际上不是这样。

12.3 使用线程的优点

使用多线程是有优点的：

1. 更快的执行速度。针对于多核cpu而言，多行是真正意义上的同时执行，那当然变快了。
2. 更好的编程模型。利用线程可以制作仿真程序。用每一个线程可以代表一个生活中的物体。
3. 更快的响应速度。（学到web编程，才知道啥叫响应速度）。比如，在网站上做一些操作，这个操作完成以后，会给你发个邮件或者短信。比如淘宝上买个商品。把下订单核发短信的操作分别放在两个线程中执行，这样就算发短信的时间比较久，也不会影响响应速度。

12.4 创建线程

创建线程的方法一共有4种：

1. 创建一个类，继承java.lang.Thread类，并且重写run方法。
2. 创建一个类，实现java.lang.Runnable接口，并且实现run方法。
3. 使用线程池
4. 使用Future和Callable

现在只能讲解前两种。

方法一：创建一个类，继承java.lang.Thread类，并且重写run方法。

```
Test.java
1
2 // A类就是一个线程类，那么A类的对象，就是一个线程。
3 class A extends Thread {
4     // run方法中的代码，叫做：“任务”
5     public void run() {
6         while(true) {
7             System.out.println("A");
8             System.out.println("B");
9             System.out.println("C");
10            System.out.println("D");
11        }
12    }
13 }
14 class B extends Thread {
15     public void run() {
16         while(true) {
17             System.out.println("1111");
18             System.out.println("2222");
19             System.out.println("3333");
20             System.out.println("4444");
21        }
22    }
23 }
24
25 public class Test {
26     public static void main (String[] args) {
27         A a = new A();
28         B b = new B();
29         a.start();
30         b.start();
31     }
32 }
33
```

继承Thread类并重写run方法

Console

<terminated> Test (61) [Java Application] D:\Java\jdk1

4444
1111
2222
3333
4444
1111
2222
3333
4444
C
D
A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
D
A

注意：

1. 万万不能调用线程的run方法，因为调用run方法，仍然是一个单线程程序，run方法不会开启线程。只有start方法才会开启线程（也就是开启一个新的执行线索）
2. 我们调用了start方法，这个start方法会自动调动线程对象中的run方法。这种现象称之为“回调”。

```
public class Test {
    public static void main (String[] args) {
        A a = new A();
        B b = new B();
        a.start();
        b.start();
    }
}
```

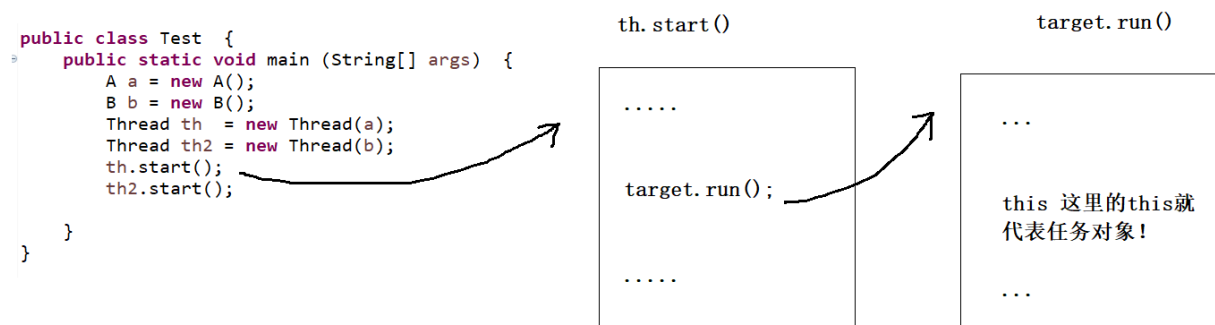
这个模型说明，子线程在执行的时候，主线程不会等待它执行完，而是直接向下执行。

方法二：创建一个类，实现java.lang.Runnable接口，并且实现run方法。

```
Test.java
1 // A不是线程类，只有继承了Thread的类，才是线程了
2 // A仅仅只是实现了Runnable接口而已，A类中只是封装了任务
3 class A implements Runnable {
4     public void run() {
5         while (true) {
6             System.out.println("AAAA");
7             System.out.println("BBBB");
8             System.out.println("CCCC");
9             System.out.println("DDDD");
10        }
11    }
12 }
13
14 class B implements Runnable {
15     public void run() {
16         while (true) {
17             System.out.println("11111111");
18             System.out.println("22222222");
19             System.out.println("33333333");
20             System.out.println("44444444");
21        }
22    }
23 }
24
25 public class Test {
26     public static void main (String[] args) {
27         A a = new A();
28         B b = new B(); 任务无法调用start方法
29         Thread th = new Thread(a); 必须把任务传给一个
30         Thread th2 = new Thread(b); 线程，再由线程来调
31         th.start(); 用start方法
32         th2.start();
33     }
34 }
35
36
37
Console
<terminated> Test (61) [Java Application] D:\Java\jdk1.8.0_121\bin\javaw.e
44444444
11111111
22222222
33333333
44444444
11111111
22222222
33333333
44444444
CCCC
DDDD
AAAA
BBBB
CCCC
DDDD
AAAA
BBBB
CCCC
DDDD
AAAA
11111111
22222222
33333333
44444444
11111111
22222222
33333333
44444444
```

注意：

1. 当我们调用th.start()方法的时候，在该方法中，会自动调用a.run方法的，所以在run方法中的this代表的是任务对象，而不是线程对象。



目前学习完了两种创建线程的方式，那么问题是到底用哪种方式？

答：只是第二种。第一种几乎不用，很少使用，如果用，那也是讲课的时候，随便写一写，在生产环境中很少见。因为第一种方式受到了“单继承”的限制。在Java中一个类只能有一个父类。所以第二种比第一种常用。

12.5 线程的状态

线程的状态有6种：

1. 新建状态 ✓
2. 可运行状态 ✓
3. 等待状态
4. wait等待状态
5. sleep等待状态 ✓
6. 终结状态 ✓

新建状态：

在一个线程刚刚被new出来以后，在start方法调用之前，这一段时间内，线程处于新建状态。

```
public class Test {  
    public static void main (String[] args) {  
        A a = new A();  
  
        Thread th = new Thread(a);  
  
        th.start();  
    }  
}
```

—— 这段时间内，线程一直处于新建状态

处于新建状态的线程，可以设置一些线程的属性：线程优先级、线程是否为守护线程、给线程起名字等等这些操作，只能在新建状态来操作，一旦线程启动了，再做这些操作就来不及了！

```

1 // A不是线程类，只有继承了Thread的类，才是线程了
2 // A仅仅只是实现了Runnable接口而已，A类中只是封装了任务
3 class A implements Runnable {
4     public void run() {
5         while (true) {
6             System.out.println("AAAA");
7             System.out.println("BBBB");
8             System.out.println("CCCC");
9             System.out.println("DDDD");
10        }
11    }
12 }
13 class B implements Runnable {
14     public void run() {
15         while (true) {
16             System.out.println("11111111");
17             System.out.println("22222222");
18             System.out.println("33333333");
19             System.out.println("44444444");
20        }
21    }
22 }
23 public class Test {
24     public static void main (String[] args) {
25         A a = new A();
26         B b = new B();
27         Thread th = new Thread(a);
28         Thread th2 = new Thread(b);
29
30         th.setPriority(1);
31         th2.setPriority(10);
32
33         th.start();
34         th2.start();
35     }
}

```

以上 `th.setPriority`就是用来设置线程优先级的（1-10）

可运行状态：

处于新建状态的线程，调用了`start`方法之后，就会进入可运行状态。

```

public class Test {
    public static void main (String[] args) {
        A a = new A();
        Thread th = new Thread(a);

        th.start();
    }
}

```

—— 可运行状态

注意，处于可运行状态的线程，就有资格去抢占CPU。并不意味着，处于可运行状态的线程就一定正在执行中！处于可运行状态的线程只是有机会被CPU执行，到底能不能执行还取决于某一个时刻该线程是否抢到cpu！！

Test.java

```

1 // A不是线程类，只有继承了Thread的类，才是线程了
2 // A仅仅只是实现了Runnable接口而已，A类中只是封装了任务
3 class A implements Runnable {
4     public void run() {
5         while (true) {
6             System.out.println("AAAA");
7             System.out.println("BBBB");
8             System.out.println("CCCC");
9             System.out.println("DDDD");
10            try {
11                Thread.sleep(5000);
12            } catch (InterruptedException e) {
13                e.printStackTrace();
14            }
15        }
16    }
17 }
18 public class Test {
19     public static void main (String[] args) {
20         A a = new A();
21         Thread th = new Thread(a);
22         th.start();
23     }
24 }
25

```

调用Thread类的静态方法sleep
就可以让当前线程阻塞指定的时间

在阻塞期间，线程就处于 sleep阻塞状态

一个线程，


```
1 import java.util.Scanner;
2
3 // A不是线程类，只有继承了Thread的类，才是线程了
4 // A仅仅是实现了Runnable接口而已，A类中只是封装了任务
5 class A implements Runnable {
6     public void run() {
7         // 吃面
8         while (true) {
9             System.out.println("拿起筷子");
10            System.out.println("夹起面条");
11            System.out.println("吸溜的嘴里");
12            System.out.println("吸溜完了，咀嚼咽下");
13        }
14    }
15 }
16
17 public class Test {
18     public static void main(String[] args) throws Exception {
19         A a = new A();
20         Thread th = new Thread(a);
21         th.start();
22
23         System.in.read();
24
25         th.stop();
26     }
27 }
28
29 }
```

当我们键入回车，th线程被中断了，
这恰恰是在什么时候键入回车，就在什么
时候中断线程，此时的线程很有可能还没有
把手头的事情处理到一个节点上！

既然stop禁止使用了，那么一定有一个代替的方案：interrupt（英文意思是打扰一下）

我们让interrupt能代替stop:

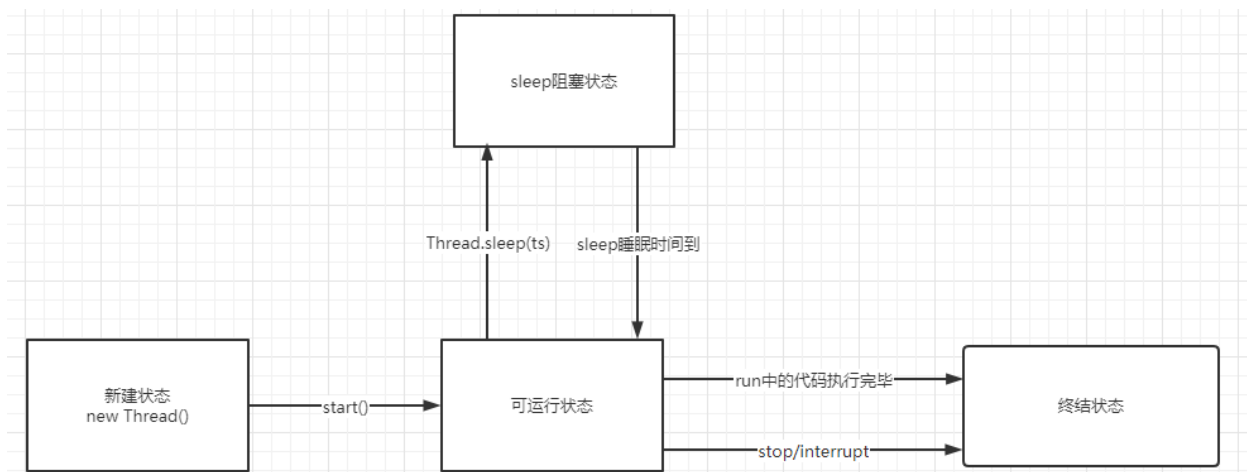
```

1 // A不是线程类，只有继承了Thread的类，才是线程了
2 // A仅仅只是实现了Runnable接口而已，A类中只是封装了任务
3 class A implements Runnable {
4     public void run() {
5         // Thread.interrupted() 也是检测线程是否被中断
6         // 平时会一直返回false，只有在调用了线程的interrupt()方法之后
7         // Thread.interrupted() 才会返回true，而且只要调用了Thread.interrupted()
8         // 方法，那么Thread.interrupted()方法还会把线程被中断的状态置为false
9         // Thread.interrupted()用在哪个线程中，就用来判断哪个线程是否被中断
10        while (!Thread.interrupted()) {
11            System.out.println("AAAA");
12            System.out.println("BBBB");
13            System.out.println("CCCC");
14            System.out.println("DDDD");
15        }
16        System.out.println("~~~: " + Thread.interrupted());
17    }
18 }
19
20 public class Test {
21     public static void main(String[] args) throws Exception {
22         A a = new A();
23         Thread th = new Thread(a);
24         th.start();
25
26         System.in.read();
27         th.interrupt();
28     }
29 }
30
31 }
32

```

关键之处，这两个地方要搭配使用

以上程序，无论何时键入Enter，线程一定会在DDDD输出完毕以后，结束的！这就叫做优雅地关闭线程。



至此线程的4种状态讲解完毕。

12.6 线程的调度

设置线程的优先级，优先级高的线程，被分配时间片就会多一些，优先级低的线程，被分配的时间片就少一些。所以我们一般把重要的任务对应的线程优先级调高一点。

```

1 class A implements Runnable {
2     public void run() {
3         while (!Thread.interrupted()) {
4             System.out.println("AAAA");
5             System.out.println("BBBB");
6             System.out.println("CCCC");
7             System.out.println("DDDD");
8         }
9     }
10 }
11 class B implements Runnable {
12     public void run() {
13         while (!Thread.interrupted()) {
14             System.out.println("11111111");
15             System.out.println("22222222");
16             System.out.println("33333333");
17             System.out.println("44444444");
18         }
19     }
20 }
21 public class Test {
22     public static void main(String[] args) throws Exception {
23         A a = new A();
24         B b = new B();
25         Thread th = new Thread(a);
26         Thread th2 = new Thread(b);
27
28         th.setPriority(Thread.MIN_PRIORITY); 最低优先级: 其实就是1
29         th2.setPriority(Thread.MAX_PRIORITY); 最高优先级: 其实就是10
30
31         th.start();
32         th2.start();
33
34         我们建议在程序中写有名字的常量: final int AGE = 10
35         不建议使用没有名字的常量!!
36     }
37 }

```

yield(), 英文意思是“让步”的意思, 让一个正在执行当中的线程, 让出cpu, 然后该线程继续抢占cpu。调用这个方法, 可以增大线程之间的切换几率。为了模拟仿真。如下,


```

class A implements Runnable {
    private Thread tt;

    public void run() {
        for (int i = 1; i <= 100; i++) {
            System.out.println("A:" + i);
            if(i == 5) {
                try {
                    tt.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public void setTt(Thread tt) {
        this.tt = tt;
    }
}

class B implements Runnable {
    public void run() {
        for (int i = 200; i <= 300; i++) {
            System.out.println("B:" + i);
        }
    }
}

```

下面定义了2个线程:

```

public class Test {
    public static void main(String[] args) throws Exception {
        A a = new A();
        B b = new B();
        Thread th = new Thread(a);
        Thread th2 = new Thread(b);

        a.setTt(th2);

        th.setPriority(Thread.MAX_PRIORITY);
        th2.setPriority(Thread.MIN_PRIORITY);

        th.start();
        th2.start();
    }
}

```

结果如下:

```

Console
<terminated> Test (61) [Java Application] D:\Java\jdk1.8.0_121\bin\javaw.

A:1
B:200
A:2
A:3
A:4
A:5
B:201
B:202
B:203
B:204
B:205
B:206
B:207
B:208
B:209
B:210
B:211
B:212
B:213
B:214

```

A任务在1到5循环直接，是与B任务
抢占cpu的，而A循环到5，就绝对不
抢占了，让B任务对应的线程执行完

如何给tt传值的：

```

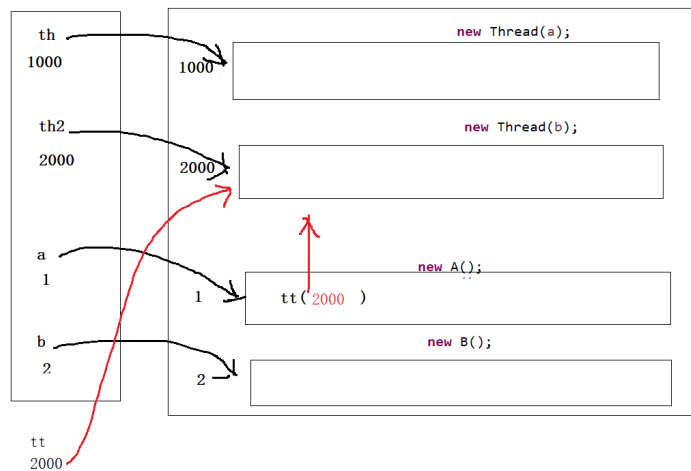
class A implements Runnable {
    private Thread tt;
    public void run() {
        for (int i = 1; i <= 100; i++) {
            System.out.println("A:" + i);
            if(i == 5) {
                try {
                    tt.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    public void setTt(Thread tt) {
        this.tt = tt;
    }
}

class B implements Runnable {
    public void run() {
        for (int i = 200; i <= 300; i++) {
            System.out.println("B:" + i);
        }
    }
}

public class Test {
    public static void main(String[] args) throws Exception {
        A a = new A();
        B b = new B();
        Thread th = new Thread(a);
        Thread th2 = new Thread(b);
        a.setTt(th2);
        th.setPriority(Thread.MAX_PRIORITY);
        th2.setPriority(Thread.MIN_PRIORITY);
        th.start();
        th2.start();
    }
}

```

建议看视频的演示过程！！



看懂以下代码，重点是，th线程会让th2线程执行完，可是th3仍然会与th2抢占cpu

```

1 class A implements Runnable {
2
3     private Thread tt;
4
5     public void run() {
6         for (int i = 1; i <= 100; i++) {
7             System.out.println("A:" + i);
8             if(i == 5) {

```

```
9  try {
10  tt.join();
11  } catch (InterruptedException e) {
12  e.printStackTrace();
13  }
14  }
15  }
16  }
17  public void setTt(Thread tt) {
18  this.tt = tt;
19  }
20  }
21  class B implements Runnable {
22  public void run() {
23  for (int i = 200; i <= 300; i++) {
24  System.out.println("B:" + i);
25  }
26  }
27  }
28  class C implements Runnable {
29
30  @Override
31  public void run() {
32  for (int i = 10000; i < 10100; i++) {
33  System.out.println("C:" + i);
34  }
35  }
36  }
37
38  public class Test {
39  public static void main(String[] args) throws Exception {
40  A a = new A();
41  B b = new B();
42  C c = new C();
43  Thread th = new Thread(a);
44  Thread th2 = new Thread(b);
45  Thread th3 = new Thread(c);
46
47  a.setTt(th2);
48
```



```

49  th.setPriority(Thread.MAX_PRIORITY);
50  th2.setPriority(Thread.MIN_PRIORITY);
51  th3.setPriority(Thread.MIN_PRIORITY);
52
53  th.start();
54  th2.start();
55  th3.start();
56  }
57  }
58

```

12.7 线程的上下文切换

即使是单核处理器也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程是同时执行的，时间片一般是几十毫秒（ms）。

CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

这就像我们同时读两本书，当我们在读一本英文的技术书时，发现某个单词不认识，于是便打开中英文字典，但是在放下英文技术书之前，大脑必须先记住这本书读到了多少页的第多少行，等查完单词之后，能够继续读这本书。这样的切换是会影响读书效率的，同样上下文切换也会影响多线程的执行速度。

```

1  public class ConcurrencyTest {
2
3      /**
4       * 当count取值不超过百万次时，并发会比串行执行累加操作要慢，为什么会这样呢？
5       * 这是因为线程有创建和上下文切换的开销
6       */
7      private static final long count = 100000000;
8
9      public static void main(String[] args) throws InterruptedException {
10         concurrency();
11         serial();
12     }
13
14     private static void concurrency() throws InterruptedException {
15         long start = System.currentTimeMillis();
16         Thread thread = new Thread(new Runnable() {

```

```

17  @Override
18  public void run() {
19      int a = 0;
20      for (long i = 0; i < count; i++) {
21          a += 5;
22      }
23  }
24  });
25  thread.start();
26  int b = 0;
27  for (long i = 0; i < count; i++) {
28      b--;
29  }
30  thread.join();
31  long time = System.currentTimeMillis() - start;
32  System.out.println("concurrency: " + time + "ms, b = " + b);
33  }
34
35  private static void serial() {
36      long start = System.currentTimeMillis();
37      int a = 0;
38      for (long i = 0; i < count; i++) {
39          a += 5;
40      }
41      int b = 0;
42      for (long i = 0; i < count; i++) {
43          b--;
44      }
45      long time = System.currentTimeMillis() - start;
46      System.out.println("serial:\t\t" + time + "ms, b = " + b + ", a = " +
47          a);
48  }

```

可以发现，当并发自行累加操作不超过一定次数时，速度会比串行执行累加操作要慢。那么，为什么并发执行的速度会比串行慢呢？这是因为线程有创建和上下文切换的开销。

12.8 同步（重点、难点）

步骤一：

```
App.java 12
1 package xian.woniuxy.test;
2
3 class A implements Runnable {
4     public void run() {
5         while(true) {
6             System.out.println("AAAA");
7             System.out.println("BBBB");
8             System.out.println("CCCC");
9             System.out.println("DDDD");
10        }
11    }
12 }
13 class B implements Runnable {
14     public void run() {
15         while(true) {
16             System.out.println("11111111");
17             System.out.println("22222222");
18             System.out.println("33333333");
19             System.out.println("44444444");
20        }
21    }
22 }
23
24 public class App {
25
26     public static void main(String[] args) {
27         Runnable a = new A();
28         Runnable b = new B();
29         Thread th = new Thread(a);
30         Thread th2 = new Thread(b);
31
32         th.start();
33         th2.start();
34
35     }
36 }
```

这个例子告诉我们，
两个进入可运行状态的线程会抢占cpu。

这两个线程抢占cpu的
时候，没有任何限制，
说抢走就抢走，往往互
相打断对象

也就是ABCD会打断
1234，或者1234会打
断ABCD

```
Console 12
<terminated> - App (37) [Java Application] D:\Java\j
22222222
33333333
44444444
11111111
22222222
33333333
44444444
11111111
BBBB
CCCC
DDDD
AAAA
BBBB
CCCC
DDDD
AAAA
BBBB
CCCC
DDDD
AAAA
BBBB
CCCC
DDDD
AAAA
BBBB
CCCC
DDDD
AAAA
```

步骤二：我们希望，ABCD和1234之间，不要互相打断对方，就算要打断，也必须在一节点上打断，也就是说要让ABCD和1234，不从中断打断对方，但可以在D和4的后面打断。也就是同步！同步也就是保证一段段，在某个时间内，不会被其他线程打断。为了达到以上目的，我们需要使用“锁”！

关于锁，有几句慢慢说：

1、java中的每一个对象，都有一把锁。

```
public class App {

    public static void main(String[] args) {
        String s = new String("abc");
        App app = new App();
        Object obj = new Object();
        List list = new ArrayList();
        String s2 = "xyz";
    }
}
```

本例中的这些对象，都有锁！

每个对象各自都有各自的锁！

2. 对象的锁，是可以被线程占有的（对比，线程除了可以抢占cpu，还可以抢占了）。如果一个锁，已经被某一个线程占有了，那么另一个线程就无法获取这个锁，当一个线程想要获取某个锁而不可得的时候，该线程会阻塞（进入了等锁状态）。

3. 一个已经占有锁的线程，还可以释放这个锁，每当释放自己占有的锁时，还会唤醒那些等待这个锁的其他线程。

以下代码加上了同步块，保证ABCD和1234不会从中间打断对方：

```

3 class A implements Runnable {
4     private Object obj;
5     public A(Object obj) {
6         this.obj = obj;
7     }
8     public void run() {
9         while(true) {
10             synchronized(obj) { // 获取obj的锁
11                 System.out.println("AAAA");
12                 System.out.println("BBBB");
13                 System.out.println("CCCC");
14                 System.out.println("DDDD");
15             } // 释放obj的锁，同时唤醒其他争抢该锁的线程
16         }
17     }
18 }
19
20 class B implements Runnable {
21     private Object obj;
22     public B(Object obj) {
23         this.obj = obj;
24     }
25     public void run() {
26         while(true) {
27             synchronized(obj) { // 获取obj的锁
28                 System.out.println("11111111");
29                 System.out.println("22222222");
30                 System.out.println("33333333");
31                 System.out.println("44444444");
32             } // 释放obj的锁，同时唤醒其他争抢该锁的线程
33         }
34     }
35 }

```

```

public class App {

    public static void main(String[] args) {
        // obj对象有一把锁。
        Object obj = new Object();

        Runnable a = new A(obj);
        Runnable b = new B(obj);
        Thread th = new Thread(a);
        Thread th2 = new Thread(b);

        th.start();
        th2.start();
    }
}

```

注意

- 1、为了让2个线程不会互相打断对方的关键代码，则务必保证它们争抢同一把锁！！
- 2、为某段代码添加上synchronized，就是保证了该段代码的安全性，所以我们又把加上了synchronized的代码段称之为：“线程安全”的代码。暂时这么理解：线程安全就是保证某段代码不会被从中间打断（不完整的解释）。

为什么要保证线程安全呢？或者说为什么要保证某段代码不能被打断？在多个线程共用某一个共享资源时，该共享资源应该在某一时刻只能被一个线程使用，不能让多个线程一起使

用。比如：洗澡。（下午有实际的代码演示）

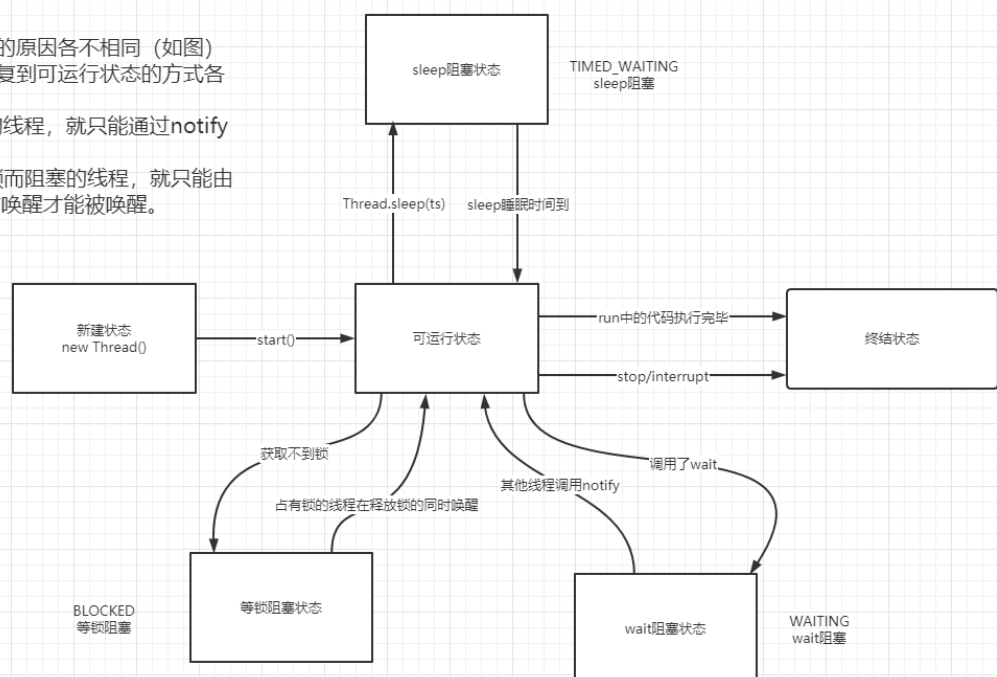
12.9 线程间的通信

运行中的线程，如果仅仅是孤立地运行，那么没有一点儿价值，或者说价值很少，如果多个线程能够相互配合完成工作，这将带来巨大的价值。

目前，我们的1234和ABCD两个线程，已经保证了不会从中间打断。接下来我们需要这样的效果：让1234和ABCD交替起来：1234ABCD1234ABCD1234ABCD.....

阻塞状态有3种：

1. 进入这3种阻塞状态的原因各不相同（如图）
2. 从这3种阻塞状态恢复到可运行状态的方式各不相同。
3. 比如由于wait阻塞的线程，就只能通过notify的方式唤醒。
4. 比如由于获取不到锁而阻塞的线程，就只能由于别的线程释放锁同时唤醒才能被唤醒。



ABCD和1234交替执行：

```
class A implements Runnable {
    private Object obj;
    public A(Object obj) {
        this.obj = obj;
    }
    public void run() {
        while(true) {
            synchronized(obj) { // 获取obj的锁
                System.out.println("AAAA");
                System.out.println("BBBB");
                System.out.println("CCCC");
                System.out.println("DDDD");
                try {
                    obj.notify();
                    obj.wait(); // wait的作用有2个：1.释放锁，同时唤醒其他争抢该锁的线程 2.让线程直接进入阻塞状态（释放cpu）
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } // 释放obj的锁，同时唤醒其他争抢该锁的线程
        }
    }
}
```

```

class B implements Runnable {
    private Object obj;
    public B(Object obj) {
        this.obj = obj;
    }
    public void run() {
        while(true) {
            synchronized(obj) { // 获取obj的锁
                System.out.println("11111111");
                System.out.println("22222222");
                System.out.println("33333333");
                System.out.println("44444444");
                try {
                    obj.notify();
                    obj.wait(); // wait的作用有2个：1.释放锁，同时唤醒其他争抢该锁的线程 2.让线程直接进入阻塞状态（释放cpu）
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } // 释放obj的锁，同时唤醒其他争抢该锁的线程
        }
    }
}

public class App {
    public static void main(String[] args) {
        // obj对象有一把锁。
        Object obj = new Object();
        Runnable a = new A(obj);
        Runnable b = new B(obj);
        Thread th = new Thread(a);
        Thread th2 = new Thread(b);

        th.start();
        th2.start();
    }
}

```

生产者消费者模型（重点，难点）

有一个饭店，其中有2个打工仔，一个负责洗碗，另一个负责使用，洗好的碗，会放在一个碗架上，使用的时候就从碗架上取下来。

分析：我们可以定义两个任务，一个洗碗，一个取碗，这两个线程有没有共享对象？有，就是碗架。如果洗碗洗得特别快，则碗架会放满，放满以后就不能再放入了。如果取得特别快，取空以后就不能再取了。这两个线程需要协调，就是通过碗架来协调的。

让我们先从碗架做起，别考虑线程：

```
App.java
1 package xian.woniuxy.test;
2 import java.util.Arrays;
3 // 货架
4 class Shelf {
5     private String[] strs = "o o o o o".split(" ");
6     private int i;
7     // 放
8     public void put() {
9         this.strs[i++] = "●";
10        System.out.println("放入, " + this);
11    }
12    // 取
13    public void get() {
14        this.strs[--i] = "o";
15        System.out.println("取出, " + this);
16    }
17    // 判断货架是否放满
18    public boolean isFull() {
19        return i == 6;
20    }
21    // 判断货架是否取空
22    public boolean isEmpty() {
23        return i == 0;
24    }
25
26    public String toString() {
27        return Arrays.toString(strs);
28    }
29 }
30 public class App {
31     public static void main(String[] args) {
32         Shelf sh = new Shelf();
33         sh.put();
34         sh.put();
35         sh.put();
36         sh.get();
37     }
38 }
39 }
```

```
Console
<terminated> App (37) [Java Application] D:\Java
放入: [●, o, o, o, o, o]
放入: [●, ●, o, o, o, o]
放入: [●, ●, ●, o, o, o]
取出: [●, ●, o, o, o, o]
```

加入线程：

```
Shelf.java
1 package xian.woniuxy.test;
2 import java.util.Arrays;
3
4 public class Shelf {
5     private String[] strs = "o o o o o".split(" ");
6     private int i;
7     // 放
8     public void put() {
9         synchronized (strs) {
10             this.strs[i++] = "●";
11             System.out.println("放入, " + this);
12         }
13     }
14     // 取
15     public void get() {
16         synchronized (strs) {
17             this.strs[--i] = "o";
18             System.out.println("取出, " + this);
19         }
20     }
21     // 判断货架是否放满
22     public boolean isFull() {
23         return i == 6;
24     }
25     // 判断货架是否取空
26     public boolean isEmpty() {
27         return i == 0;
28     }
29     public String toString() {
30         return Arrays.toString(strs);
31     }
32 }
33 }
```

```
App.java
4     private Shelf sh;
5     public T(Shelf sh) {
6         this.sh = sh;
7     }
8     public void run() {
9         while(true) {
10             sh.put();
11         }
12     }
13
14 class S implements Runnable {
15     private Shelf sh;
16
17     public S(Shelf sh) {
18         this.sh = sh;
19     }
20     @Override
21     public void run() {
22         while(true) {
23             sh.get();
24         }
25     }
26 }
27
28 public class App {
29     public static void main(String[] args) {
30         Shelf sh = new Shelf();
31         T t = new T(sh);
32         S s = new S(sh);
33         Thread th = new Thread(t);
34         Thread th2 = new Thread(s);
35         th.start();
36         th2.start();
37     }
38 }
39 }
```

```
Console
<terminated> App (37) [Java Application] D:\Java
放入: [●, o, o, o, o, o]
取出: [o, o, o, o, o, o]
Exception in thread "Thr
    at xian.woniuxy.
    at xian.woniuxy.
    at java.lang.Thr
java.lang.ArrayIndexOutC
    at xian.woniuxy.
    at xian.woniuxy.
    at java.lang.Thr
```

以上代码没有解决线程之间的通信问题，会造成数组下标越界的问题。

此处我们完善一下这个概念：线程安全，所谓的线程安全，就是保证一个被多个线程共享的资源，在任何时刻只能被一个线程操作。只有当出走它的线程完成了一个操作后，其他线程才能对它继续操作。总之不能让多个线程同时操作一个共享对象。

继续完善以上例子，仅仅修改了放和取得方法，其他没变。

```

private int i,
// 放
public void put() {
    synchronized (strs) {
        // 在放满以后, 就不应该再放了
        // 所以这里一定要判断碗架是不是满了
        if(isFull()) {
            try {
                strs.notify();
                strs.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.strs[i++] = "●";
        System.out.println("放入: " + this);
    }
}
// 取
public void get() {
    synchronized (strs) {
        if(isEmpty()) {
            try {
                strs.notify();
                strs.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.strs[--i] = "○";
        System.out.println("取出: " + this);
    }
}
}

```

如此：生产者就会放完，再唤醒消费者，消费者取完就会唤醒生产者，如此往复。

放入:	[●, ○, ○, ○, ○, ○]
放入:	[●, ●, ○, ○, ○, ○]
放入:	[●, ●, ●, ○, ○, ○]
放入:	[●, ●, ●, ●, ○, ○]
放入:	[●, ●, ●, ●, ●, ○]
放入:	[●, ●, ●, ●, ●, ●]
取出:	[●, ●, ●, ●, ●, ○]
取出:	[●, ●, ●, ●, ○, ○]
取出:	[●, ●, ●, ○, ○, ○]
取出:	[●, ●, ○, ○, ○, ○]
取出:	[●, ○, ○, ○, ○, ○]
取出:	[○, ○, ○, ○, ○, ○]
放入:	[●, ○, ○, ○, ○, ○]
放入:	[●, ●, ○, ○, ○, ○]
放入:	[●, ●, ●, ○, ○, ○]
放入:	[●, ●, ●, ●, ○, ○]
放入:	[●, ●, ●, ●, ●, ○]
放入:	[●, ●, ●, ●, ●, ●]
取出:	[●, ●, ●, ●, ●, ○]
取出:	[●, ●, ●, ●, ○, ○]
取出:	[●, ●, ●, ○, ○, ○]
取出:	[●, ●, ○, ○, ○, ○]
取出:	[●, ○, ○, ○, ○, ○]
取出:	[○, ○, ○, ○, ○, ○]

问题是，洗碗和取碗，用得着非要等到对方放满或者取空才唤醒吗？只要碗架有哪怕一个空曹，我们就应该让洗碗的醒着，只要碗架哪怕只有一个碗，我们就应该让取碗的醒着。

```

// 放
public void put() {
    synchronized (strs) {
        // 在放满以后，就不应该再放了
        // 所以这里一定要判断碗架是不是满了
        if(isFull()) {
            try {
                strs.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            this.strs[i++] = "●";
            System.out.println("放入: " + this);
            strs.notify();
        }
    }
}

// 取
public void get() {
    synchronized (strs) {
        if(isEmpty()) {
            try {
                strs.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.strs[--i] = "○";
        System.out.println("取出: " + this);
        strs.notify();
    }
}

```

多生产者和多消费者的例子：

```

// 放
public void put() {
    synchronized (strs) {
        // 在放满以后，就不应该再放了
        // 所以这里一定要判断碗架是不是满了
        while(isFull()) {
            try {
                strs.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.strs[i++] = "●";
        System.out.println(Thread.currentThread().getName() + "放入: " + this);
        strs.notify();
    }
}

// 取
public void get() {
    synchronized (strs) {
        while(isEmpty()) {
            try {
                strs.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.strs[--i] = "○";
        System.out.println(Thread.currentThread().getName() + "取出: " + this);
        strs.notify();
    }
}

```

注意以下代码是如何给一个线程指定名字的：

```

public class App {
    public static void main(String[] args) {
        Shelf sh = new Shelf(10);
        T t = new T(sh);
        S s = new S(sh);
        Thread th = new Thread(t, "郭靖");
        Thread th2 = new Thread(t, "杨康");
        Thread th3 = new Thread(s, "黄蓉");
        Thread th4 = new Thread(s, "穆念慈");
        th.start();
        th2.start();
        th3.start();
        th4.start();
    }
}

```

当，一个方法中，使用了`synchronized(this) {}`时，且，该同步块扩住了该方法中的所有代码时，我们就可以直接把`synchronized`关键字写在方法签名（signature）上

以下使用的锁，术语上叫做 对象锁

```

// 放
public void put() {
    synchronized (this) {
        // 在放满以后，就不应该再放了
        // 所以这里一定要判断碗架是不是满了
        while(isFull()) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.strs[i++] = "●";
        System.out.println(Thread.currentThread().getName() + "放入");
        this.notifyAll();
    }
}

// 取
public void get() {
    synchronized (this) {
        while(isEmpty()) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.strs[--i] = "○";
        System.out.println(Thread.currentThread().getName() + "取出");
        this.notifyAll();
    }
}

```

以上代码等价于以下代码， 以下代码使用的所术语上叫 方法锁：

```

// 放
public synchronized void put() {
    // 在放满以后, 就不应该再放了
    // 所以这里一定要判断碗架是不是满了
    while(isFull()) {
        try {
            this.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    this.strs[i++] = "●";
    System.out.println(Thread.currentThread().getName() + "放入: " + this);
    this.notifyAll();
}

// 取
public synchronized void get() {
    while(isEmpty()) {
        try {
            this.wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    this.strs[--i] = "○";
    System.out.println(Thread.currentThread().getName() + "取出: " + this);
    this.notifyAll();
}

```

问题是：静态方法呢？我们已经静态方法中只能访问静态字段或其他静态方法，不能访问实例字段，或者实例方法。静态方法中也不能访问this和super。

以下左右两端代码等价！！	
<pre> public class App { public static void f2() { synchronized(App.class) { } } } </pre>	<pre> public class App { public static synchronized void f2() { } } </pre>

1. 在静态方法前加上synchronized，会等价于在静态方法中写了一个同步块，而这个同步块会把该静态方法中的所有代码都括起来！
2. 在静态方法前加上synchronized，其实获取的就是静态方法所在的类的那个字节码对象的锁。

以上的锁术语上叫做：静态方法锁。

以下例子证明了，当一个线程通过共享对象的notify()方法，唤醒竞争同一把锁的另外一个线程时，

另一个线程只有获取锁之后，才能从wait方法返回。并不是调用了notify之后，wait阻塞的线程就能立即执行，线程从wait阻塞状态中恢复到可运行状态后，仍然要去抢锁，抢到锁才能执行。

```
1 class T implements Runnable {
2     private Object obj;
3     public T(Object obj) {
4         this.obj = obj;
5     }
6     @Override
7     public void run() {
8         synchronized (obj) {
9             System.out.println("A");
10            System.out.println("B");
11            System.out.println("C");
12            try {
13                obj.wait();
14            } catch (InterruptedException e) {
15                e.printStackTrace();
16            }
17            System.out.println("D");
18            System.out.println("E");
19            System.out.println("F");
20        }
21    }
22 }
23 class S implements Runnable {
24     private Object obj;
25     public S(Object obj) {
26         this.obj = obj;
27     }
28     @Override
29     public void run() {
30         synchronized (obj) {
31             System.out.println("1");
32             System.out.println("2");
33             System.out.println("3");
34             obj.notify();
35             for (int i = 100; i < 200; i++) {
36                 System.out.println(i);
```

```
37  try {
38  Thread.sleep(100);
39  } catch(Exception e) {
40  e.printStackTrace();
41  }
42  }
43  }
44  }
45  }
46  public class Test {
47  public static void main(String[] args) throws Exception {
48  Object obj = new Object();
49  T t = new T(obj);
50  S s = new S(obj);
51  Runnable target;
52  Thread th = new Thread(t);
53  Thread th2 = new Thread(s);
54  th.start();
55  try {
56  Thread.sleep(1000);
57  } catch(Exception e) {
58  e.printStackTrace();
59  }
60  th2.start();
61  }
62  }
```

12.10 jps和jstack

在dos界面下，键入tasklist就可以看到所有正在运行的进程。

```
C:\WINDOWS\system32\cmd.exe

C:\Users\Administrator>tasklist

映像名称 PID 会话名 会话# 内存使用
=====
System Idle Process 0 Services 0 8 K
System 4 Services 0 28 K
Registry 96 Services 0 6,052 K
smss.exe 412 Services 0 600 K
csrss.exe 612 Services 0 2,336 K
wininit.exe 716 Services 0 3,436 K
services.exe 792 Services 0 7,680 K
lsass.exe 800 Services 0 11,892 K
svchost.exe 932 Services 0 748 K
svchost.exe 960 Services 0 20,872 K
```

如果你安装了jdk，并且也配置好了path环境变量，那么你就可以使用这一个命令：jps，该命令专门用于打印出java进程。

```
4
5 public class App {
6     public static void main(String[] args) throws Exception {
7         while(true)
8             ;
9     }
10 }
11
```

```
C:\WINDOWS\system32\cmd.exe

C:\Users\Administrator>jps
1472 Jps
13156 App
11016
```

在知道了Java进程的id以后，我们就可以使用jstack命令来查看一个Java进程中的所有线程的相关信息：比如该Java进程中有哪些线程，处于什么状态....

```
App.java
1 package xian.woniuxy.test4;
2
3 class A implements Runnable {
4     public void run() {
5         while(true) {
6             ;
7         }
8     }
9 }
10
11
12 public class App {
13     public static void main(String[] args) throws Exception {
14         A a = new A();
15         Thread th = new Thread(a, "步惊云");
16         th.start();
17     }
18 }
19
```

```
C:\WINDOWS\system32\cmd.exe

11016
15368 App
7292 Jps

C:\Users\Administrator>jstack 15368
2021-09-17 11:19:23
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.121-b13 mixed mode):

"DestroyJavaVM" #11 prio=5 os_prio=0 tid=0x0000000002fa0800 nid=0x30c8 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"步惊云?" #10 prio=5 os_prio=0 tid=0x0000000001da6a00 nid=0xc6c runnable [0x0000000001e5ce000]
java.lang.Thread.State: RUNNABLE
  at xian.woniuxy.test4.A.run(App.java:5)
  at java.lang.Thread.run(Thread.java:745)
```

```

3 class A implements Runnable {
4     public void run() {
5         while(true) {
6             try {
7                 Thread.sleep(10000);
8             } catch (InterruptedException e) {
9                 e.printStackTrace();
10            }
11        }
12    }
13 }
14
15 public class App {
16     public static void main(String[] args) {
17         A a = new A();
18         Thread th = new Thread(a, "步惊云");
19         th.start();
20     }
21 }

```

```

C:\WINDOWS\system32\cmd.exe
C:\Users\Administrator>jps
11016
15992 App
13244 Jps

C:\Users\Administrator>jstack 15992
2021-09-17 11:21:34
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.121-b1
"DestroyJavaVM" #11 prio=5 os_prio=0 tid=0x0000000002b10800 n
java.lang.Thread.State: RUNNABLE

"步惊云" #10 prio=5 os_prio=0 tid=0x0000000001d64a800 nid=0x34
java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at xian.woniuxy.test4.A.run(App.java:7)
    at java.lang.Thread.run(Thread.java:745)

```

```

class A implements Runnable {
    public void run() {
        while(true) {
            synchronized (this) {
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

public class App {
    public static void main(String[] args) {
        A a = new A();
        Thread th = new Thread(a, "步惊云");
        th.start();
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
11016
4068 App

C:\Users\Administrator>jstack 4068
2021-09-17 11:22:58
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.121-b13 mixed m
"DestroyJavaVM" #11 prio=5 os_prio=0 tid=0x00000000027d0800 nid=0x3fb0
java.lang.Thread.State: RUNNABLE

"步惊??" #10 prio=5 os_prio=0 tid=0x0000000001d37d000 nid=0x3ff0 in Obj
java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x0000000076b4b0e58> (a xian.woniuxy.test4.A)
    at java.lang.Object.wait(Object.java:502)
    at xian.woniuxy.test4.A.run(App.java:8)
    - locked <0x0000000076b4b0e58> (a xian.woniuxy.test4.A)
    at java.lang.Thread.run(Thread.java:745)

"Service Thread" #9 daemon prio=9 os_prio=0 tid=0x0000000001d320800 nid
java.lang.Thread.State: RUNNABLE

```



```
App.java
3 class A implements Runnable {
4     private Object obj;
5     public A(Object obj) {
6         this.obj = obj;
7     }
8     public void run() {
9         synchronized (obj) {
10             while(true) {
11                 System.out.println("AAA");
12             }
13         }
14     }
15 }
16 class B implements Runnable {
17     private Object obj;
18     public B(Object obj) {
19         this.obj = obj;
20     }
21     public void run() {
22         synchronized (obj) {
23             while(true) {
24                 System.out.println("BBB");
25             }
26         }
27     }
28 }
29 public class App {
```

```
Console
App (38) [Java Application] D:\Java\jdk1.8.
AAA
AAA
AAA
AAA
AAA
AAA
AAA
AAA
AAA
AAA
```

```
C:\WINDOWS\system32\cmd.exe
2021-09-17 11:26:53
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.121-b13 mixed mode):
"DestroyJavaVM" #12 prio=5 os_prio=0 tid=0x0000000003430800 nid=0x12ec waiting c
  java.lang.Thread.State: RUNNABLE
"暴风" #11 prio=5 os_prio=0 tid=0x0000000001df2d000 nid=0x3048 waiting for monito
  java.lang.Thread.State: BLOCKED (on object monitor)
    at xian.woniuxy.test4.B.run(App.java:24)
    - waiting to lock <0x0000000076f40a80> (a java.lang.Object)
    at java.lang.Thread.run(Thread.java:745)
"步惊云" #10 prio=5 os_prio=0 tid=0x0000000001df2c000 nid=0x1eb8 runnable [0x0000
  java.lang.Thread.State: RUNNABLE
    at java.io.FileOutputStream.writeBytes(Native Method)
    at java.io.FileOutputStream.write(FileOutputStream.java:326)
    at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:82)
    at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:140)
    - locked <0x0000000076f482e30> (a java.io.BufferedOutputStream)
    at java.io.PrintStream.write(PrintStream.java:432)
    - locked <0x0000000076f404ca8> (a java.io.PrintStream)
    at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:221)
    at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:291)
```

12.11 死锁

死锁的情况要避免，我们学习死锁不是为了写它，而是为了避免。什么叫死锁呢？一个线程已经占有A锁，还想占有B锁，另一个线程已经占有B锁，还想要A锁。以下代码有可能造成死锁！循环一段时间后有几率出现

```

package xian.woniuxy.test4;
class A implements Runnable {
    private Object obj, obj2;

    public A(Object obj, Object obj2) {
        this.obj = obj;
        this.obj2 = obj2;
    }

    public void run() {
        while (true) {
            synchronized (obj) {
                synchronized (obj2) {
                    System.out.println("AAA");
                }
            }
        }
    }
}

class B implements Runnable {
    private Object obj, obj2;
    public B(Object obj, Object obj2) {
        this.obj = obj;
        this.obj2 = obj2;
    }
    public void run() {
        while (true) {
            synchronized (obj2) {
                synchronized (obj) {
                    System.out.println("BBB");
                }
            }
        }
    }
}

public class App {
    public static void main(String[] args) throws Exception {
        Object obj = new Object();
        Object obj2 = new Object();
        A a = new A(obj, obj2);
        B b = new B(obj, obj2);
        Thread th = new Thread(a, "步惊云");
        Thread th2 = new Thread(b, "聂风");
        th.start();
        th2.start();
    }
}

```

如何避免死锁，你就避免多个线程同时争抢多个资源。

12.12 守护线程

一个处于新建状态的线程，才可以被设置为守护线程。守护线程也是一个线程，我们之前学习过的线程的所有那些基础语法，守护线程也都有，但是守护线程的特点是：当一个程序中只剩下守护线程的时候，程序直接结束。比如：西游记，圣斗士。

```

1 package xian.woniuxy.test4;
2 class A implements Runnable {
3
4     public void run() {
5         while (true) {
6             System.out.println("AAAA");
7         }
8     }
9 }
10
11 public class App {
12     public static void main(String[] args) throws Exception {
13         A a = new A();
14         Thread th = new Thread(a);
15
16         // 设置th为守护线程
17         th.setDaemon(true);
18         th.start();
19
20         System.in.read();
21         System.out.println("over");
22     }
23 }
24

```

当我们键入Enter后，main线程就结束了

程序中此时就只有守护线程，则程序直接结束！

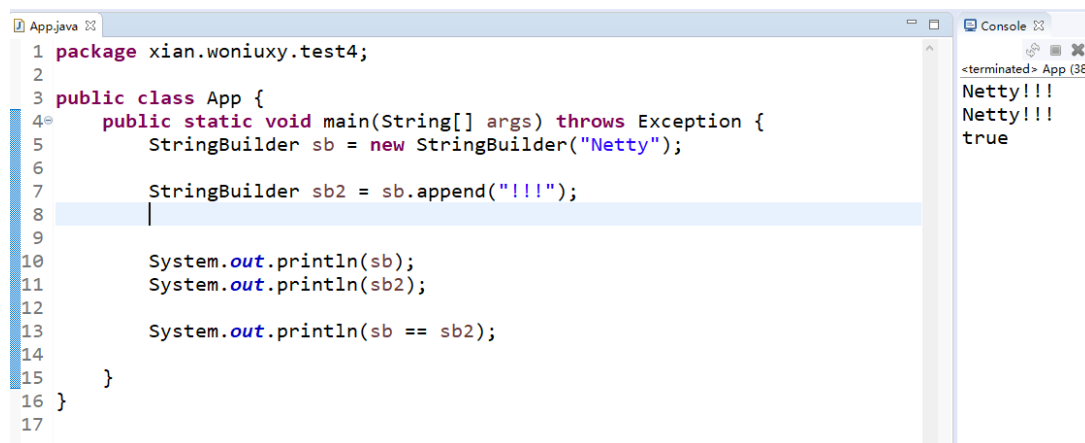
守护线程的价值何在？守护线程，顾名思义，就是守护普通线程，比如GC就是一个守护线程，负责把那些不可达的对象干掉！如果正常的线程都结束了，GC也就没必要存在了。

[illegible]

12.13 StringBuilder和StringBuffer的区别

我们已经知道，String和StringBuffer的区别了。 我们现在要学习的使用StringBuilder和StringBuffer的区别。

1. StringBuilder和StringBuffer都是可变字符串，对它们任何一个类的实例进行操作，都是在原有对象的空间上直接修改的，不会产生新空间。



The screenshot shows an IDE with a Java file named App.java. The code defines a package xian.woniuxy.test4 and a public class App. Inside the class, there is a main method that takes a String array args and throws an Exception. The main method creates a StringBuilder sb with the initial value "Netty". It then creates a second StringBuilder sb2 by appending "!!!" to sb. Finally, it prints sb, sb2, and checks if sb equals sb2. The console output on the right shows the program terminated, followed by the printed values "Netty!!!" and "Netty!!!" and the boolean result "true".

```
1 package xian.woniuxy.test4;
2
3 public class App {
4     public static void main(String[] args) throws Exception {
5         StringBuilder sb = new StringBuilder("Netty");
6
7         StringBuilder sb2 = sb.append("!!!");
8
9
10        System.out.println(sb);
11        System.out.println(sb2);
12
13        System.out.println(sb == sb2);
14
15    }
16 }
17
```

Console Output:

```
<terminated> App (38)
Netty!!!
Netty!!!
true
```

2. StringBuilder和StringBuffer的区别在于，StringBuilder是线程不安全的，StringBuffer是线程安全的。

3. StringBuilder的优点是执行速度快，缺点是不安全。 StringBuffer优点是线程安全的，缺点是加锁和放锁是有性能损耗的。

所以如何抉择？ 你说呢？

12.14 线程池

先讲解一半线程池。另外一半4阶段再见

池，但凡在编程领域中，听到“池”这个字眼，池就是容器的意思，池中是可以存放一些东西的，而在池中的这些东西，就是为了重用。当需要用的时候，从池中拿出来一个现成的资源，在用完以后不销毁，直接换回池中。

这里要讲解的线程池，当然也具备池的特点，线程池顾名思义，就是为了重用线程。为什么要重用线程呢？ 因为系统创建一个线程的性能代价较高，而且销毁一个线程的代价也很高。所以我们应该避免每次需要使用线程的时候，都new一个新的线程。 所以我们就使用线程池。

线程池我们可以自定义，系统也有内置好的线程池。我们这里就只学习4种系统内置好的线程。

固定线程池:

```
Testjava
2
3 import java.util.concurrent.ExecutorService;
4 import java.util.concurrent.Executors;
5 class Task implements Runnable {
6     private static int seed = 1;
7     private final int id = seed++;
8
9     public void run() {
10         String name = Thread.currentThread().getName();
11         System.out.println(name + "开始执行" + this.id + "任务");
12         sleep();
13         System.out.println(name + "执行完毕" + this.id + "任务");
14     }
15
16     private void sleep() {
17         try {
18             Thread.sleep(5000);
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22     }
23 }
24 public class Test {
25     public static void main(String[] args) {
26         ExecutorService es = Executors.newFixedThreadPool(3);
27         for(int i = 1; i <= 10; i++) {
28             Task t = new Task();
29             es.execute(t);
30         }
31         es.shutdown();
32     }
33 }
34
```

```
<terminated> Test (64) [Java Application] D:\Java\jdk1.8
pool-1-thread-1开始执行1任务
pool-1-thread-2开始执行2任务
pool-1-thread-3开始执行3任务
pool-1-thread-1执行完毕1任务
pool-1-thread-2执行完毕2任务
pool-1-thread-3执行完毕3任务
pool-1-thread-2开始执行4任务
pool-1-thread-3开始执行5任务
pool-1-thread-1开始执行6任务
pool-1-thread-3执行完毕5任务
pool-1-thread-1执行完毕6任务
pool-1-thread-2执行完毕4任务
pool-1-thread-1开始执行8任务
pool-1-thread-2开始执行9任务
pool-1-thread-3开始执行7任务
pool-1-thread-3执行完毕7任务
pool-1-thread-3开始执行10任务
pool-1-thread-2执行完毕9任务
pool-1-thread-1执行完毕8任务
pool-1-thread-3执行完毕10任务
```

缓存线程池

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class Task implements Runnable {
    private static int seed = 1;
    private final int id = seed++;

    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name + "开始执行" + this.id + "任务");
        sleep();
        System.out.println(name + "执行完毕" + this.id + "任务");
    }

    private void sleep() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class Test {
    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        for(int i = 1; i <= 10; i++) {
            Task t = new Task();
            es.execute(t);
        }
        es.shutdown();
    }
}
```

```
<terminated> Test (64) [Java Application] D:\Java\jdk1.8.0_121\bin\jav
pool-1-thread-1开始执行1任务
pool-1-thread-3开始执行3任务
pool-1-thread-2开始执行2任务
pool-1-thread-4开始执行4任务
pool-1-thread-5开始执行5任务
pool-1-thread-6开始执行6任务
pool-1-thread-8开始执行8任务
pool-1-thread-7开始执行7任务
pool-1-thread-9开始执行9任务
pool-1-thread-10开始执行10任务
pool-1-thread-5执行完毕5任务
pool-1-thread-2执行完毕2任务
pool-1-thread-1执行完毕1任务
pool-1-thread-4执行完毕4任务
pool-1-thread-3执行完毕3任务
pool-1-thread-8执行完毕8任务
pool-1-thread-7执行完毕7任务
pool-1-thread-6执行完毕6任务
pool-1-thread-9执行完毕9任务
pool-1-thread-10执行完毕10任务
```

缓存线程池，特点是，有多少个任务交给线程池，该线程池内部就会创建多少个线程，来者不拒!!! 统统立即执行。这样确实伤内存，伤cpu。

它的优点是，一旦该线程池new出一个线程以后，等该线程把任务执行完毕以后，不会立即销毁线程，而是让该线程在池中闲置一段时间，当闲置时间一过，才销毁该线程。

单一线程池，顾名思义，就是一个线程池中只有一个线程：


```
Test.java
1 package xian.woniuxy.test3;
2 import java.util.concurrent.ExecutorService;
3 import java.util.concurrent.Executors;
4 class Task implements Runnable {
5     private static int seed = 1;
6     private final int id = seed++;
7
8     public void run() {
9         String name = Thread.currentThread().getName();
10        System.out.println(name + "开始执行" + this.id + "任务");
11        sleep();
12        System.out.println(name + "执行完毕" + this.id + "任务");
13    }
14
15    private void sleep() {
16        try {
17            Thread.sleep(2000);
18        } catch (InterruptedException e) {
19            e.printStackTrace();
20        }
21    }
22 }
23 public class Test {
24     public static void main(String[] args) {
25         ExecutorService es = Executors.newFixedThreadPool(3);
26         for(int i = 1; i <= 10; i++) {
27             Task t = new Task();
28             es.execute(t);
29         }
30         es.shutdownNow(); 说关闭就立即关闭，不会等待线程把任务执行完
31         System.out.println("over");
32     }
33 }
34

Console
<terminated> Test (64) [Java Application] D:\Java\jdk1.8.0_121\bin\javaw.exe (2021年9月17日 下午3:45:16)
over
pool-1-thread-1开始执行1任务
pool-1-thread-2开始执行2任务
pool-1-thread-3开始执行3任务
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at xian.woniuxy.test3.Task.sleep(Test.java:17)
    at xian.woniuxy.test3.Task.run(Test.java:11)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(Thread.java:745)
pool-1-thread-2执行完毕2任务
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at xian.woniuxy.test3.Task.sleep(Test.java:17)
    at xian.woniuxy.test3.Task.run(Test.java:11)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(Thread.java:745)
java.lang.InterruptedException: sleep interruptedpool
    at java.lang.Thread.sleep(Native Method)
    at xian.woniuxy.test3.Task.sleep(Test.java:17)
    at xian.woniuxy.test3.Task.run(Test.java:11)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(Thread.java:745)
pool-1-thread-1执行完毕1任务
```

12.15 单例设计模式

单例模式：一个类只能有一个实例，并且只有一种获取该实例的方法。

以上单例模式是：饱汉式

```
App.java
1 package xian.woniuxy.test5;
2
3 class Singleton {
4     private static Singleton s = new Singleton();
5     private Singleton() {
6     }
7     public static Singleton getInstance() {
8         return s;
9     }
10 }
11
12 public class App {
13     public static void main(String[] args) {
14         Singleton s = Singleton.getInstance();
15         Singleton s2 = Singleton.getInstance();
16
17         System.out.println(s == s2);
18
19     }
20 }
21 }
22

Console
<terminated> App (3)
true
```

懒汉式

```
App.java
1 package xian.woniuxy.test5;
2
3 class Singleton {
4     private static Singleton s = null;
5     private Singleton() {
6         System.out.println("实例化");
7     }
8     public static Singleton getInstance() {
9         if(s == null) {
10             s = new Singleton();
11         }
12         return s;
13     }
14 }
15
16 public class App {
17     public static void main(String[] args) {
18         Singleton s = Singleton.getInstance();
19         Singleton s2 = Singleton.getInstance();
20
21         System.out.println(s == s2);
22     }
23 }
24
25 }
26
```

Console

<terminated> App (39) [Java Appli

实例化

true

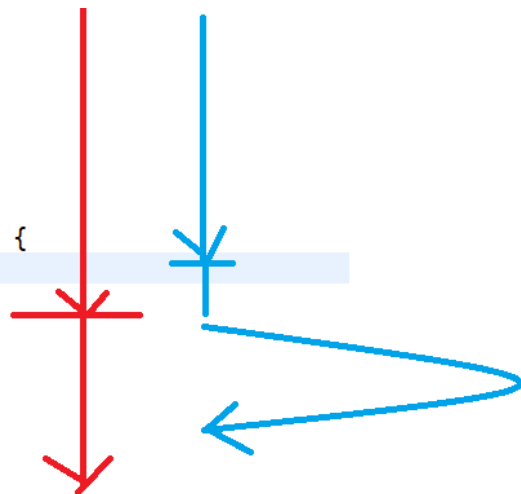
```
App.java
1 package xian.woniuxy.test5;
2
3 class Singleton {
4     private static Singleton s = null;
5     private Singleton() {
6         System.out.println("实例化");
7     }
8     public static Singleton getInstance() {
9         if(s == null) {
10             s = new Singleton();
11         }
12         return s;
13     }
14 }
15
```

在多线程下，单例模式不再试单例了：


```
App.java
1 package xian.woniuxy.test5;
2
3 class Singleton {
4     private static Singleton s = null;
5     private Singleton() {
6         System.out.println("实例化");
7     }
8     public static Singleton getInstance() {
9         if(s == null) {
10             s = new Singleton();
11         }
12         return s;
13     }
14 }
15
16 public class App {
17     public static void main(String[] args) {
18         for(int i = 1; i <= 10; i++) {
19             Thread th = new Thread(new Runnable() {
20                 public void run() {
21                     Singleton s = Singleton.getInstance();
22                     System.out.println(s);
23                 }
24             });
25             th.start();
26         }
27     }
28 }
29
30 }
31
```

```
Console
<terminated> App (39) [Java Application] D:\Java\jdk1.8.0_121\bin\javaw.exe (2021年9月1
实例化
xian.woniuxy.test5.Singleton@5ef7355a
实例化
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
xian.woniuxy.test5.Singleton@41873c1
```

```
class Singleton {
    private static Singleton s = null;
    private Singleton() {
        System.out.println("实例化");
    }
    public static Singleton getInstance() {
        synchronized (Singleton.class) {
            if (s == null) {
                s = new Singleton();
            }
        }
        return s;
    }
}
```



12.16 ThreadLocal (选讲)