多线程

> Author：zhangzhang
>
> Version：1.0.0

一、进程和线程

1.1 进程

> 进程：程序是静止的，只有真正运行时的程序，才被称为进程。

> 特点：
>
> - 单核CPU在任何时间点上。
> - 只能运行一个进程。
> - 宏观并行、微观串行。

进程

1.2 线程

> 线程：又称轻量级进程（Light Weight Process）。
>
> - 程序中的一个顺序控制流程，同时也是CPU的基本调度单位。
> - 进程由多个线程组成，彼此间完成不同的工作，交替执行，称为多线程。
>
> 比如：
>
> - 迅雷是一个进程，当中的多个下载任务即是多个线程。
> - Java虚拟机是一个进程，默认包含主线程（main），通过代码创建多个独立线程，与main并发执行。

1.3 进程和线程区别

> - 进程是操作系统资源分配的基本单位，而线程是CPU的基本调度单位。
> - 一个程序运行后至少有一个进程。
> - 一个进程可以包含多个线程，但是至少需要有一个线程。
> - 进程间不能共享数据段地址，但同进程的线程之间可以。

1.4 线程组成

> 任何一个线程都具有基本的组成部分：
>
> - CPU时间片：操作系统（OS）会为每个线程分配执行时间。
> - 运行数据：
>   堆空间：存储线程需使用的对象，多个线程可以共享堆中的对象。
>   栈空间：存储线程需使用的局部变量，每个线程都拥有独立的栈。
> - 线程的逻辑代码。

二、创建线程【 重点 】

> Java中创建线程主要有两种方式：
>
> - 继承Thread类。
> - 实现Runnable接口。

2.1 继承Thread类

> 步骤：
>
> - 编写类、继承Thread。
> - 重写run方法。
> - 创建线程对象。
> - 调用start方法启动线程。

案例演示：

MyThread类:

```
public class MyThread extends Thread {

  public MyThread() {
    // TODO Auto-generated constructor stub
  }
  public MyThread(String name) {
    super(name);
  }

  @Override
  public void run() {
    for(int i=0;i<100;i++) {
      System.out.println("子线程:"+i);
    }
  }
}
```

TestMyThread类：

```
public class TestThread {
  public static void main(String[] args) {
    //1创建线程对象
    MyThread myThread=new MyThread();
    myThread.start();//myThread.run()
    //创建第二个线程对象
    MyThread myThread2=new MyThread();
    myThread2.start();
    //主线程执行
    for(int i=0;i<50;i++) {
      System.out.println("主线程======================"+i);
    }
  }
}
```

获取线程名称：

- getName()。
- Thread.currentThread().getName()。

```
  public void run() {
    for(int i=0;i<100;i++) {
      //this.getId获取线程Id
      //this.getName获取线程名称
      //第一种方式 this.getId和this.getName();
      //System.out.println("线程id:"+this.getId()+" 线程名称:"+this.getName()+" 子线程............."+i);
      //第二种方式 Thread.currentThread() 获取当前线程
      System.out.println("线程id:"+Thread.currentThread().getId()+" 线程名称:"+Thread.currentThread().getName()+" 子线程。。。。。。。"+i);
    }
  }
```

```
public static void main(String[] args) {
    //1创建线程对象
    MyThread myThread=new MyThread("我的子线程1");
    //2启动线程,不能使用run方法
    //修改线程名称
    //myThread.setName("我的子线程1");
    myThread.start();//myThread.run()

    //创建第二个线程对象
    MyThread myThread2=new MyThread("我的子线程2");

    //myThread2.setName("我的子线程2");
    myThread2.start();

    //主线程执行
    for(int i=0;i<50;i++) {
      System.out.println("主线程======================"+i);
    }
  }
```

2.2 课堂案例

实现四个窗口各卖100张票。

TicketWin类

```java
public class TicketWin extends Thread{

  public TicketWin() {
    // TODO Auto-generated constructor stub
  }
  public TicketWin(String name) {
    super(name);
  }
  private int ticket=100;//票
  @Override
  public void run() {
    //卖票功能
    while(true) {
      if(ticket<=0) {
        break;
      }
      System.out.println(Thread.currentThread().getName()+"卖了第"+ticket+"张票");
      ticket--;
    }
  }
}
```

TestWin类：

```java
public class TestWin {
  public static void main(String[] args) {
    //创建四个窗口
    TicketWin w1=new TicketWin("窗口1");
    TicketWin w2=new TicketWin("窗口2");
    TicketWin w3=new TicketWin("窗口3");
    TicketWin w4=new TicketWin("窗口4");
    //启动线程
    w1.start();
    w2.start();
    w3.start();
    w4.start();
  }
}
```

2.3 实现Runnable接口

步骤：

- 编写类实现Runnable接口、并实现run方法。
- 创建Runnable实现类对象。
- 创建线程对象，传递实现类对象。
- 启动线程。

案例演示：

MyRunnable类：

```java
public class MyRunnable implements Runnable{

  @Override
  public void run() {
    for(int i=0;i<100;i++) {
      System.out.println(Thread.currentThread().getName()+" ........."+i);
    }
  }

}
```

TestMyRunnable类：

```java
public class TestRunnable {
  public static void main(String[] args) {
    //1创建MyRunnable对象,表示线程要执行的功能
    MyRunnable runnable=new MyRunnable();
    //2创建线程对象
    Thread thread=new Thread(runnable, "我的线程1");
    //3启动
    thread.start();

    for(int i=0;i<50;i++) {
      System.out.println("main............"+i);
    }
  }
}
```

2.4 课堂案例

> 实现四个窗口共卖100张票。

> Ticket类：

```
public class Ticket implements Runnable {
  private int ticket=100;//100张票


  @Override
  public void run() {
    while(true) {
      if(ticket<=0) {
        break;
      }
      System.out.println(Thread.currentThread().getName()+" 卖了第"+ticket+"张票");
      ticket--;
    }

  }
}
```

> TestTicket类：

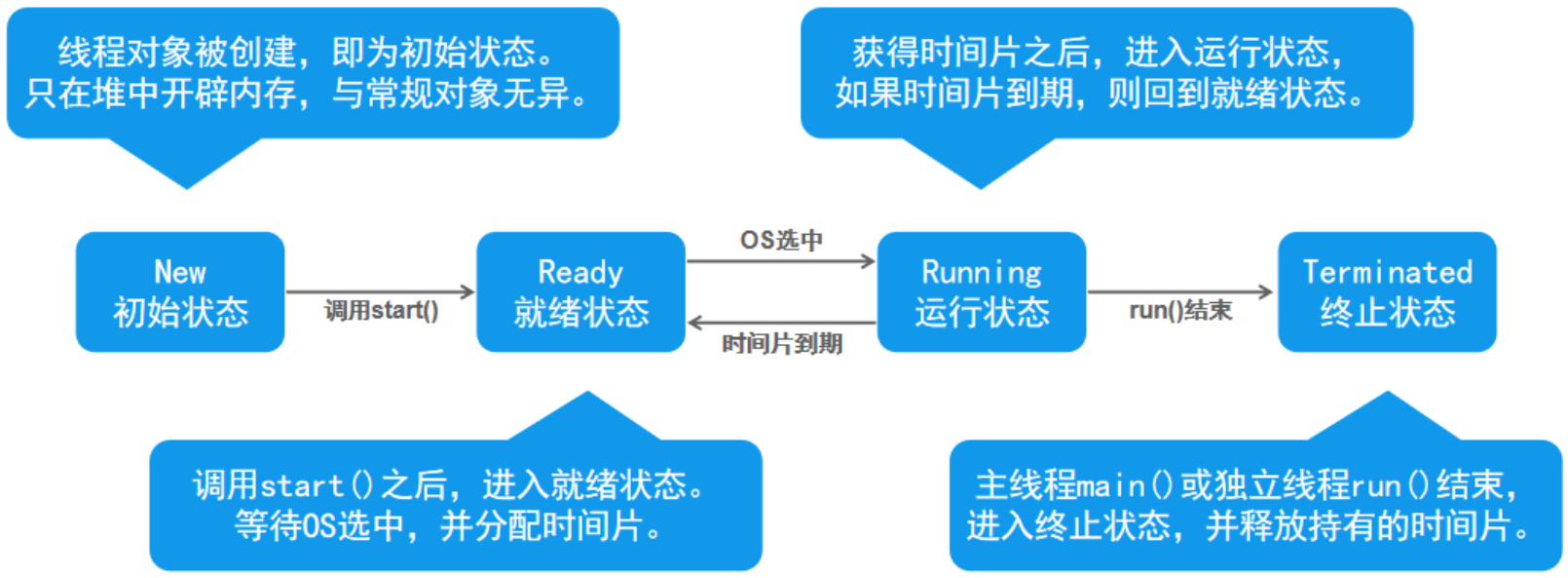
```
public class TestTicket {
  public static void main(String[] args) {
    //1创建票对象
    Ticket ticket=new Ticket();
    //2创建线程对象
    Thread w1=new Thread(ticket, "窗口1");
    Thread w2=new Thread(ticket, "窗口2");
    Thread w3=new Thread(ticket, "窗口3");
    Thread w4=new Thread(ticket, "窗口4");
    //3启动线程
    w1.start();
    w2.start();
    w3.start();
    w4.start();
  }
}
```

三、线程状态

3.1 线程状态（基本）

> 线程状态：新建、就绪、运行、终止。

线程基本状态



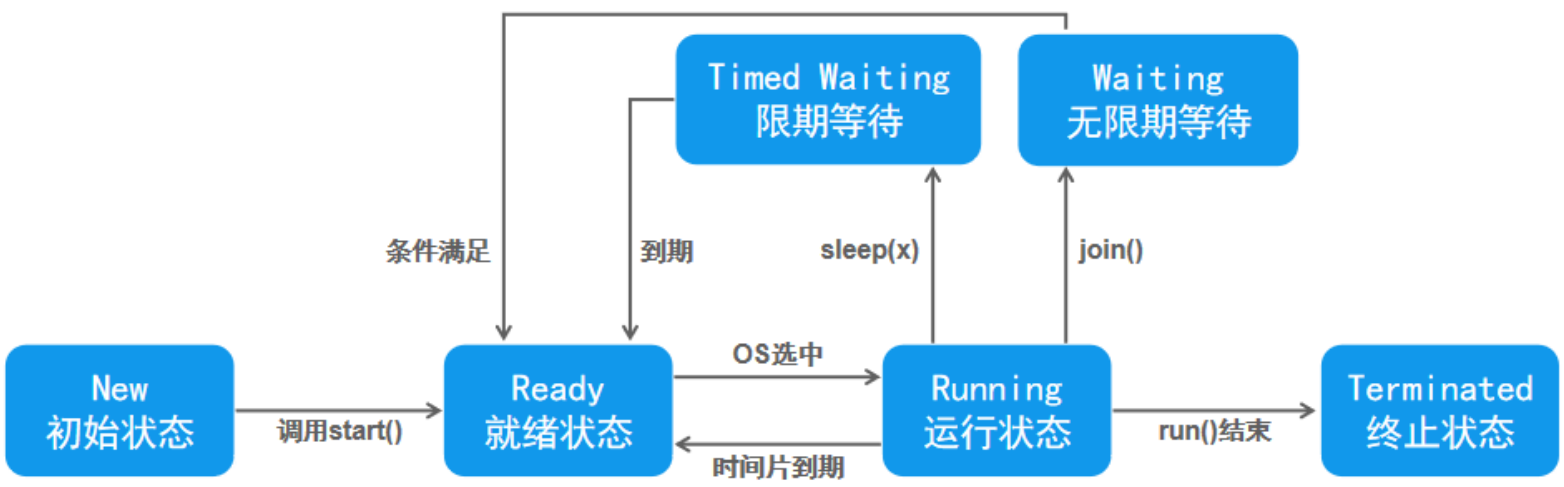
3.2 常见方法

方法名	说明
public static void sleep(long millis)	当前线程主动休眠 millis 毫秒。
public static void yield()	当前线程主动放弃时间片，回到就绪状态，竞争下一次时间片。
public final void join()	允许其他线程加入到当前线程中。
public void setPriority(int)	线程优先级为1-10，默认为5,优先级越高，表示获取CPU机会越多。
public void setDaemon(boolean)	设置为守护线程线程有两类：用户线程（前台线程）、守护线程（后台线程）

3.3 线程状态（等待）

> 线程状态：新建、就绪、运行、等待、终止。

线程状态



四、线程安全【重点】

> **为什么会出现线程安全问题?**
>
> - 需求：A线程将“Hello”存入数组；B线程将“World”存入数组。
> - 线程不安全：
>   - 当多线程并发访问临界资源时，如果破坏原子操作，可能会造成数据不一致。
>   - 临界资源：共享资源（同一对象），一次仅允许一个线程使用，才可保证其正确性。
>   - 原子操作：不可分割的多步操作，被视作一个整体，其顺序和步骤不可打乱或缺省。

> 案例演示：

```
public class ThreadSafe {
  private static int index=0;
  public static void main(String[] args)  throws Exception{
    //创建数组
    String[] s=new String[5];
    //创建两个操作
    Runnable runnableA=new Runnable() {

      @Override
      public void run() {
        //同步代码块
        synchronized (s) {
          s[index]="hello";
          index++;
        }

      }
    };
    Runnable runnableB=new Runnable() {

      @Override
      public void run() {
        synchronized (s) {
          s[index]="world";
          index++;
        }

      }
    };
    //创建两个线程对象
    Thread a=new Thread(runnableA,"A");
    Thread b=new Thread(runnableB,"B");
    a.start();
    b.start();

    a.join();//加入线程
    b.join();//加入线程

    System.out.println(Arrays.toString(s));

  }
}
```

4.1 同步代码块

> **语法:**

```
synchronized(临界资源对象){ //对临界资源对象加锁
  //代码（原子操作）
}
```

注意：

- **每个对象都有一个互斥锁标记，用来分配给线程的。**
- **只有拥有对象互斥锁标记的线程，才能进入对该对象加锁的同步代码块。**
- **线程退出同步代码块时，会释放相应的互斥锁标记。**

演示案例：

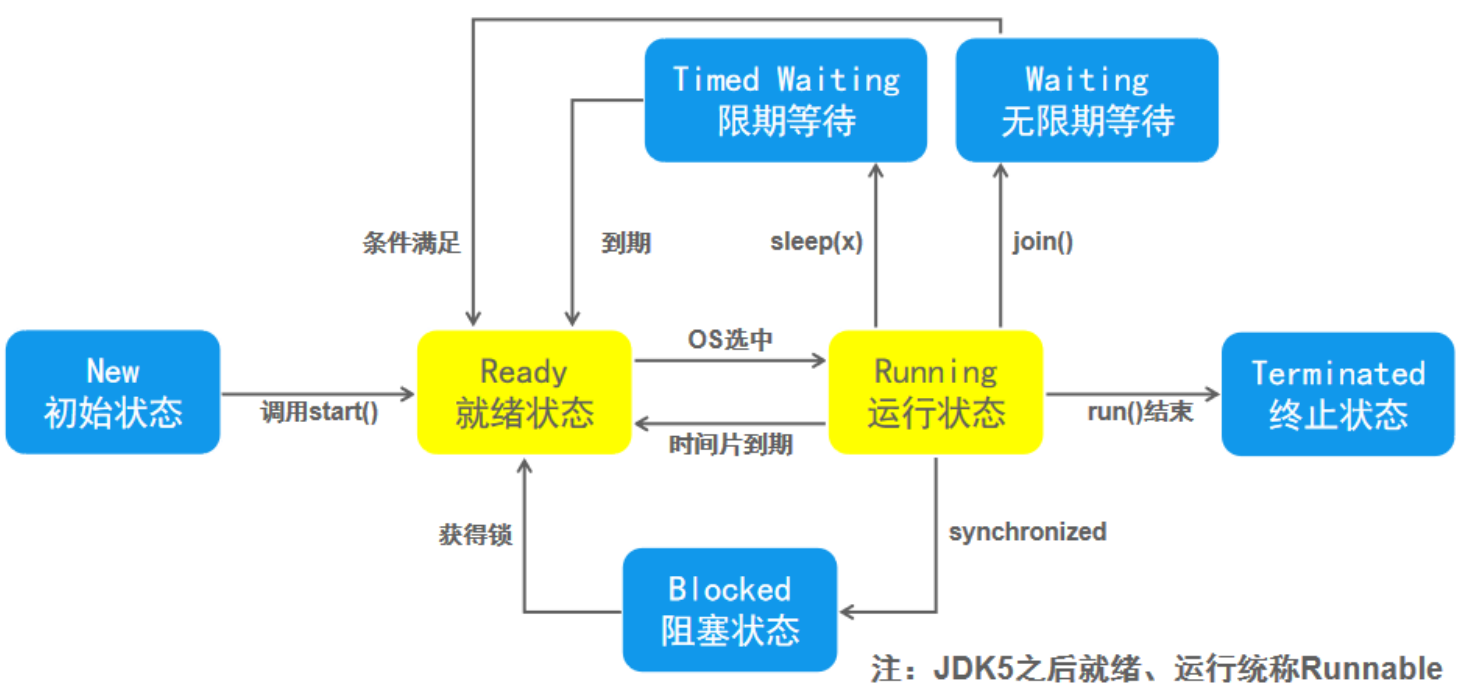
Ticket类：

```
public class Ticket implements Runnable{

  private int ticket=100;
  //创建锁
  //private Object obj=new Object();

  @Override
  public void run() {

    while(true) {
      synchronized (this) {//this ---当前对象
        if(ticket<=0) {
          break;
        }
        System.out.println(Thread.currentThread().getName()+"卖了第"+ticket+"票");
        ticket--;
      }

    }
  }

}
```

4.2 线程状态（阻塞）

线程状态：新建、就绪、运行、阻塞、终止。

线程状态



4.3 同步方法

语法：

```
synchronized 返回值类型 方法名称(形参列表){ //对当前对象（this）加锁
  // 代码（原子操作）
}
```

注意：

- **只有拥有对象互斥锁标记的线程，才能进入该对象加锁的同步方法中。**
- **线程退出同步方法时，会释放相应的互斥锁标记。**
- **如果方式是静态，锁是类名.class。**

4.4 同步规则

- 只有在调用包含同步代码块的方法，或者同步方法时，才需要对象的锁标记。
- 如调用不包含同步代码块的方法，或普通方法时，则不需要锁标记，可直接调用。

> JDK中线程安全的类：
>
> - StringBuffer
> - Vector
> - Hashtable
>
> 以上类中的公开方法，均为synchonized修饰的同步方法。

五、死锁

5.1 什么是死锁?

> - 当第一个线程拥有A对象锁标记，并等待B对象锁标记，同时第二个线程拥有B对象锁标记，并等待A对象锁标记时，产生死锁。
> - 一个线程可以同时拥有多个对象的锁标记，当线程阻塞时，不会释放已经拥有的锁标记，由此可能造成死锁。

5.2 死锁案例

> MyLock类：

```
public class MyLock {
  //两个锁(两个筷子)
  public static Object a=new Object();
  public static Object b=new Object();
}
```

> BoyThread类：

```
public class Boy extends Thread{
  @Override
  public void run() {
    synchronized (MyLock.a) {
      System.out.println("男孩拿到了a");
      synchronized (MyLock.b) {
        System.out.println("男孩拿到了b");
        System.out.println("男孩可以吃东西了...");
      }
    }
  }
}
```

> GirlThread类：

```
public class Girl extends Thread {
  @Override
  public void run() {
    synchronized (MyLock.b) {
      System.out.println("女孩拿到了b");
      synchronized (MyLock.a) {
        System.out.println("女孩拿到了a");
        System.out.println("女孩可以吃东西了...");
      }
    }
  }
}
```

> TestDeadLock类：

```
public class TestDeadLock {
  public static void main(String[] args) {
    Boy boy=new Boy();
    Girl girl=new Girl();
    girl.start();
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }

    boy.start();
  }
}
```

六、线程通信

6.1 线程通信方法

方法	说明
public final void wait()	释放锁，进入等待队列
public final void wait(long timeout)	在超过指定的时间前，释放锁，进入等待队列
public final void notify()	随机唤醒、通知一个线程
public final void notifyAll()	唤醒、通知所有线程

注意：所有的等待、通知方法必须在对加锁的同步代码块中。

6.2 生产者消费者

> 若干个生产者在生产产品，这些产品将提供给若干个消费者去消费，为了使生产者和消费者能并发执行，在两者之间设置一个能存储多个产品的缓冲区，生产者将生产的产品放入缓冲区中，消费者从缓冲区中取走产品进行消费，显然生产者和消费者之间必须保持同步，即不允许消费者到一个空的缓冲区中取产品，也不允许生产者向一个满的缓冲区中放入产品。

> Bread类：

```
public class Bread {
  private int id;
  private String productName;
  public Bread() {
    // TODO Auto-generated constructor stub
  }
  public Bread(int id, String productName) {
    super();
    this.id = id;
    this.productName = productName;
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getProductName() {
    return productName;
  }
  public void setProductName(String productName) {
    this.productName = productName;
  }
  @Override
  public String toString() {
    return "Bread [id=" + id + ", productName=" + productName + "]";
  }
}
```

> BreadCon类：

```
public class BreadCon {
  //存放面包的数组
  private Bread[] cons=new Bread[6];
  //存放面包的位置
  private int index=0;

  //存放面包
  public synchronized void input(Bread b) { //锁this
    //判断容器有没有满
    while(index>=6) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }

    cons[index]=b;
    System.out.println(Thread.currentThread().getName()+"生产了"+b.getId()+"");
    index++;
    //唤醒
    this.notifyAll();



}
//取出面包
public synchronized void output() {//锁this
  while(index<=0) {
    try {
      this.wait();
    } catch (InterruptedException e) {
```

```
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
    index--;
    Bread b=cons[index];
    System.out.println(Thread.currentThread().getName()+"消费了"+b.getId()+" 生产者:"+b.getProductName());
    cons[index]=null;
    //唤醒生产者
    this.notifyAll();
  }
}
```

> Consume类：

```
public class Consume implements Runnable{

  private BreadCon con;

  public Consume(BreadCon con) {
    super();
    this.con = con;
  }

  @Override
  public void run() {
    for(int i=0;i<30;i++) {
      con.output();
    }
  }

}
```

> Produce类：

```
public class Prodcut implements Runnable {

  private BreadCon con;

  public Prodcut(BreadCon con) {
    super();
    this.con = con;
  }

  @Override
  public void run() {
    for(int i=0;i<30;i++) {
      con.input(new Bread(i, Thread.currentThread().getName()));
    }
  }

}
```

> Test类：

```
public class Test {
  public static void main(String[] args) {
    //容器
    BreadCon con=new BreadCon();
    //生产和消费
    Prodcut prodcut=new Prodcut(con);
    Consume consume=new Consume(con);
    //创建线程对象
    Thread chenchen=new Thread(prodcut, "晨晨");
    Thread bingbing=new Thread(consume, "消费");
    Thread mingming=new Thread(prodcut, "明明");
    Thread lili=new Thread(consume, "莉莉");
    //启动线程
    chenchen.start();
    bingbing.start();
    mingming.start();
    lili.start();
  }
}
```

七、线程池【 重点 】

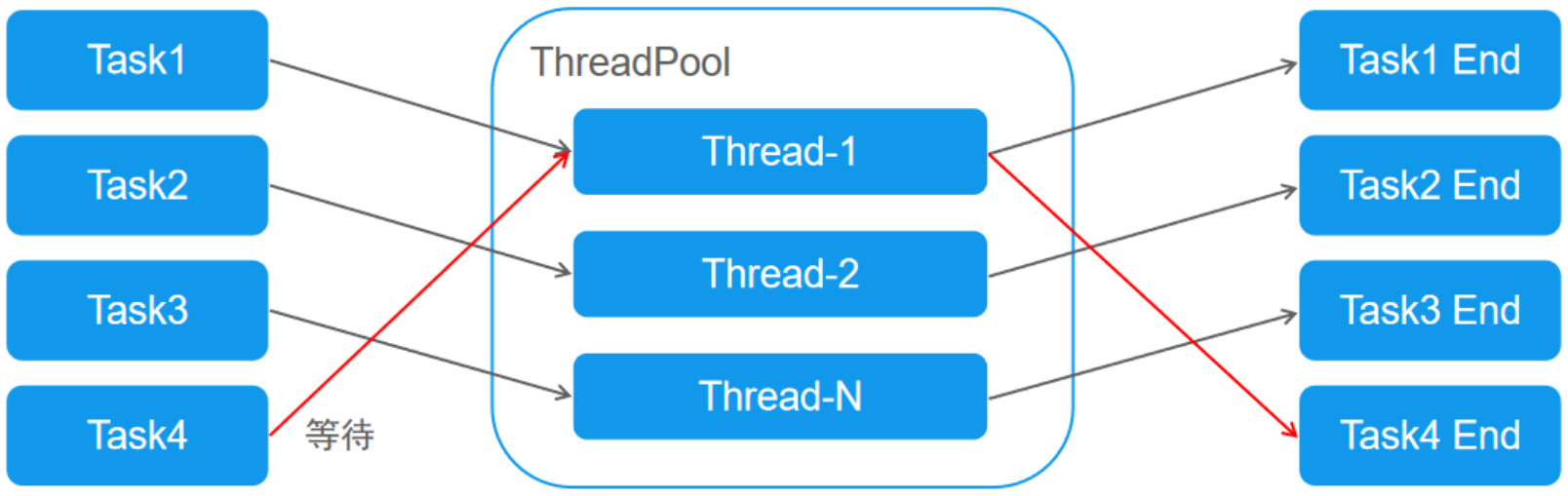
7.1 为什么需要线程池?

- 如果有非常的多的任务需要多线程来完成，且每个线程执行时间不会太长，这样频繁的创建和销毁线程。
- 频繁创建和销毁线程会比较耗性能。有了线程池就不要创建更多的线程来完成任务，因为线程可以重用。

7.2 线程池原理

> 线程池用维护者一个队列，队列中保存着处于等待（空闲）状态的线程。不用每次都创建新的线程。

线程池



7.3 线程池API

> 常用的线程池接口和类(所在包java.util.concurrent)。

> Executor：线程池的顶级接口。
>
> ExecutorService：线程池接口，可通过submit(Runnable task) 提交任务代码。
>
> Executors工厂类：通过此类可以获得一个线程池。

方法名	描述
newFixedThreadPool(int nThreads)	获取固定数量的线程池。参数：指定线程池中线程的数量。
newCachedThreadPool()	获得动态数量的线程池，如不够则创建新的，无上限。
newSingleThreadExecutor()	创建单个线程的线程池，只有一个线程。
newScheduledThreadPool()	创建固定大小的线程池，可以延迟或定时执行任务。

> 案例演示：测试线程池。

```
public class TestThreadPool {
  public static void main(String[] args) {
    //1.1创建固定线程个数的线程池
    //ExecutorService es=Executors.newFixedThreadPool(4);
    //1.2创建缓存线程池，线程个数由任务个数决定
    ExecutorService es=Executors.newCachedThreadPool();
    //1.3创建单线程线程池
    //Executors.newSingleThreadExecutor();
    //1.4创建调度线程池  调度:周期、定时执行
    //Executors.newScheduledThreadPool(corePoolSize)
    Executors.newScheduledThreadPool(3);
    //2创建任务
    Runnable runnable=new Runnable() {
      private int ticket=100;
      @Override
      public void run() {
        while(true) {
          if(ticket<=0) {
            break;
          }
          System.out.println(Thread.currentThread().getName()+"买了第"+ticket+"张票");
          ticket--;
        }
      }
    };
    //3提交任务
    for(int i=0;i<5;i++) {
      es.submit(runnable);
    }
    //4关闭线程池
    es.shutdown();//等待所有任务执行完毕 然后关闭线程池，不接受新任务。
  }
}
```

7.4 Callable接口

```
public interface Callable< V >{
    public V call() throws Exception;
}
```

注意：

- **JDK5加入，与Runnable接口类似，实现之后代表一个线程任务。**
- **Callable具有泛型返回值、可以声明异常。**

案例演示：Callable接口的使用。

```
public class TestCallable {
  public static void main(String[] args) throws Exception{
    //功能需求：使用Callable实现1-100和
    //1创建Callable对象
    Callable<Integer> callable=new Callable<Integer>() {
      @Override
      public Integer call() throws Exception {
        System.out.println(Thread.currentThread().getName()+"开始计算");
        int sum=0;
        for(int i=1;i<=100;i++) {
          sum+=i;
          Thread.sleep(100);
        }
        return sum;
      }
    };
    //2把Callable对象 转成可执行任务
    FutureTask<Integer> task=new FutureTask<>(callable);

    //3创建线程
    Thread thread=new Thread(task);

    //4启动线程
    thread.start();

    //5获取结果(等待call执行完毕，才会返回)
    Integer sum=task.get();
    System.out.println("结果是:"+sum);
  }
}
```

Runnable接口和Callable接口的区别：

- Callable接口中call方法有返回值,Runnable接口中run方法没有返回值。
- Callable接口中call方法有声明异常，Runnable接口中run方法没有异常。

7.5 Future接口

- Future接口表示将要执行完任务的结果。
- get()以阻塞形式等待Future中的异步处理结果（call()的返回值）。

案例演示：计算1-100的和。

```
public class TestFuture {
  public static void main(String[] args) throws Exception{
    //1创建线程池
    ExecutorService es=Executors.newFixedThreadPool(1);
    //2提交任务 Future:表示将要执行完任务的结果
    Future<Integer> future=es.submit(new Callable<Integer>() {

      @Override
      public Integer call() throws Exception {
        System.out.println(Thread.currentThread().getName()+"开始计算");
        int sum=0;
        for(int i=1;i<=100;i++) {
          sum+=i;
          Thread.sleep(10);
        }
        return sum;
      }
    });

    //3获取任务结果,等待任务执行完毕才会返回.

    System.out.println(future.get());

    //4关闭线程池
    es.shutdown();
  }
}
```

7.6 课堂案例

> 需求：使用两个线程，并发计算1~50、51~100的和，再进行汇总统计。

```
public class TestFuture2 {
  public static void main(String[] args) throws Exception{
    //1创建线程池
    ExecutorService es=Executors.newFixedThreadPool(2);
    //2提交任务
    Future<Integer> future1=es.submit(new Callable<Integer>() {

      @Override
      public Integer call() throws Exception {
        int sum=0;
        for(int i=1;i<=50;i++) {
          sum+=i;
        }
        System.out.println("1-50计算完毕");
        return sum;
      }
    });
    Future<Integer> future2=es.submit(new Callable<Integer>() {

      @Override
      public Integer call() throws Exception {
        int sum=0;
        for(int i=51;i<=100;i++) {
          sum+=i;
        }
        System.out.println("51-100计算完毕");
        return sum;
      }
    });
    //3获取结果
    int sum=future1.get()+future2.get();
    System.out.println("结果是:"+sum);
    //4关闭线程池
    es.shutdown();
  }
}
```

八、Lock接口

8.1 Lock

> - JDK5加入，与synchronized比较，显示定义，结构更灵活。
> - 提供更多实用性方法，功能更强大、性能更优越。

> 常用方法：

方法名	描述
void lock()	获取锁，如锁被占用，则等待。
boolean tryLock()	尝试获取锁（成功返回true。失败返回false，不阻塞）。
void unlock()	释放锁。

8.2 重入锁

> ReentrantLock：
>
> - Lock接口的实现类，与synchronized一样具有互斥锁功能。

```
public class MyList {
  //创建锁
  private Lock lock=new ReentrantLock();
  private String[] str= {"A","B","","","",""};
  private int count=2;

  public void add(String value) {
    lock.lock();
    try {
      str[count]=value;
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
      count++;
      System.out.println(Thread.currentThread().getName()+"添加了"+value);
    } finally {
```

```
      lock.unlock();
    }
  }

  public String[] getStr() {
    return str;
  }
}
```

8.3 读写锁

> ReentrantReadWriteLock：
>
> - 一种支持一写多读的同步锁，读写分离，可分别分配读锁、写锁。
> - 支持多次分配读锁，使多个读操作可以并发执行。
>
> 互斥规则：
>
> - 写-写：互斥，阻塞。
> - 读-写：互斥，读阻塞写、写阻塞读。
> - 读-读：不互斥、不阻塞。
> - 在读操作远远高于写操作的环境中，可在保障线程安全的情况下，提高运行效率。

> ReadWriteDemo类：

```
public class ReadWriteDemo {
  //创建读写锁
  private ReentrantReadWriteLock rrl=new ReentrantReadWriteLock();
  //获取读锁
  private ReadLock readLock=rrl.readLock();
  //获取写锁
  private WriteLock writeLock=rrl.writeLock();

  //互斥锁
  private ReentrantLock lock=new ReentrantLock();

  private String value;

  //读取
  public String getValue() {
    //使用读锁上锁
    lock.lock();
    try {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
      System.out.println("读取:"+this.value);
      return this.value;
    }finally {
      lock.unlock();
    }
  }
  //写入
  public void setValue(String value) {
    lock.lock();
    try {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
      System.out.println("写入:"+value);
      this.value=value;
    }finally {
      lock.unlock();
    }
  }

}
```

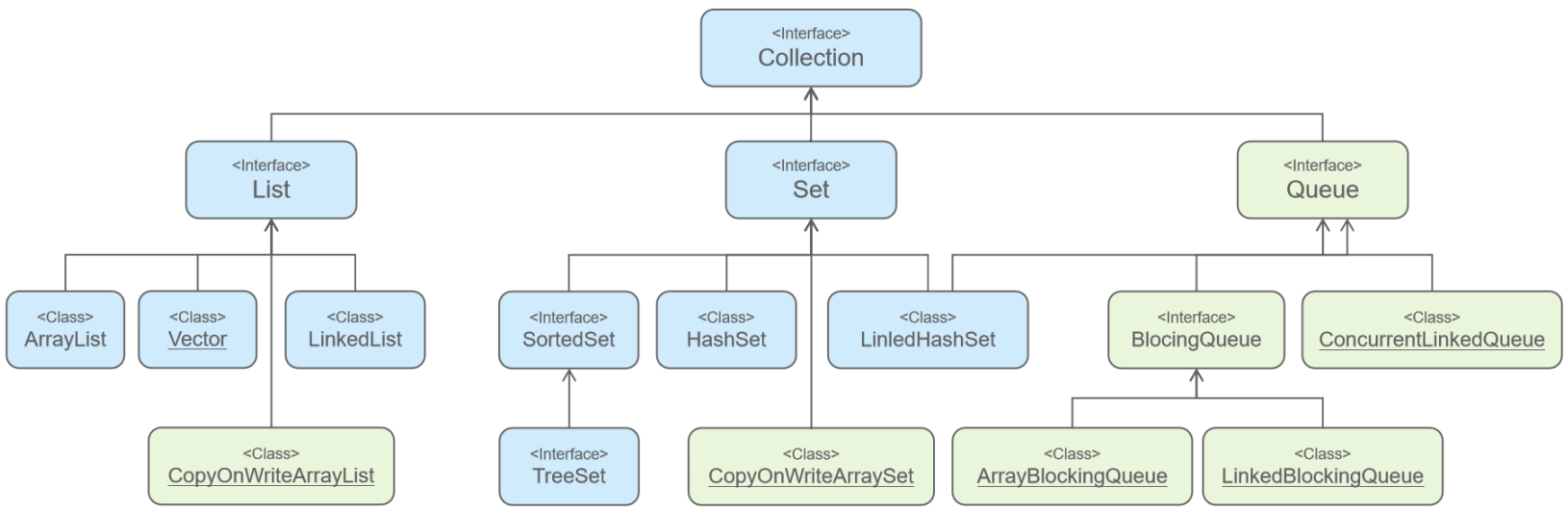
> TestReadWriteLock类：

```
public class TestReadWriteLock {
  public static void main(String[] args) {
    ReadWriteDemo readWriteDemo=new ReadWriteDemo();
    //创建线程池
    ExecutorService es=Executors.newFixedThreadPool(20);

    Runnable read=new Runnable() {

      @Override
```

```
      public void run() {
        readWriteDemo.getValue();
      }
    };
    Runnable write=new Runnable() {

      @Override
      public void run() {
        readWriteDemo.setValue("张三:"+new Random().nextInt(100));
      }
    };
    long start=System.currentTimeMillis();
    //分配2个写的任务
    for(int i=0;i<2;i++) {
      es.submit(write);
    }

    //分配18读取任务
    for(int i=0;i<18;i++) {
      es.submit(read);
    }


    es.shutdown();//关闭
    while(!es.isTerminated()) {//空转

    }
    long end=System.currentTimeMillis();
    System.out.println("用时:"+(end-start));


  }
}
```

九、线程安全集合

线程安全集合



注：绿色代表新增知识，下划线代表线程安全集合。

> Collections工具类中提供了多个可以获得线程安全集合的方法。

方法名
public static Collection synchronizedCollection(Collection c)
public static List synchronizedList(List list)
public static Set synchronizedSet(Set s)
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
public static SortedSet synchronizedSortedSet(SortedSet s)
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)

注：JDK1.2提供，接口统一、维护性高，但性能没有提升，均以synchonized实现。

9.1 CopyOnWriteArrayList

> - 线程安全的ArrayList，加强版读写分离。
> - 写有锁，读无锁，读写之间不阻塞，优于读写锁。
> - 写入时，先copy一个容器副本、再添加新元素，最后替换引用。
> - 使用方式与ArrayList无异。

```
public class TestCopyOnWriteArrayList {
  public static void main(String[] args) {
    //1创建集合
    CopyOnWriteArrayList<String> list=new CopyOnWriteArrayList<>();
    //2使用多线程操作
    ExecutorService es=Executors.newFixedThreadPool(5);
```

```
    //3提交任务
    for(int i=0;i<5;i++) {
      es.submit(new Runnable() {

        @Override
        public void run() {
          for(int j=0;j<10;j++) {
            list.add(Thread.currentThread().getName()+"...."+new Random().nextInt(1000));
          }
        }
      });
    }
    //4关闭线程池
    es.shutdown();
    while(!es.isTerminated()) {}
    //5打印结果
    System.out.println("元素个数:"+list.size());
    for (String string : list) {
      System.out.println(string);
    }
  }
}
```

9.2 CopyOnWriteArraySet

> - 线程安全的Set，底层使用CopyOnWriteArrayList实现。
> - 唯一不同在于，使用addIfAbsent()添加元素，会遍历数组。
> - 如存在元素，则不添加（扔掉副本）。

```
public class TestCopyOnWriteArraySet {
  public static void main(String[] args) {
    //1创建集合
    CopyOnWriteArraySet<String> set=new CopyOnWriteArraySet<>();
    //2添加元素
    set.add("pingguo");
    set.add("huawei");
    set.add("xiaomi");
    set.add("lianxiang");
    set.add("pingguo");
    //3打印
    System.out.println("元素个数:"+set.size());
    System.out.println(set.toString());
  }
}
```

9.3 ConcurrentHashMap

> - 初始容量默认为16段（Segment），使用分段锁设计。
> - 不对整个Map加锁，而是为每个Segment加锁。
> - 当多个对象存入同一个Segment时，才需要互斥。
> - 最理想状态为16个对象分别存入16个Segment，并行数量16。
> - 使用方式与HashMap无异。

```
public class TestConcurrentHashMap {
  public static void main(String[] args) {
    //1创建集合
    ConcurrentHashMap<String, String> hashMap=new ConcurrentHashMap<String, String>();
    //2使用多线程添加数据
    for(int i=0;i<5;i++) {
      new Thread(new Runnable() {

        @Override
        public void run() {
          for(int k=0;k<10;k++) {
            hashMap.put(Thread.currentThread().getName()+"--"+k, k+"");
            System.out.println(hashMap);
          }
        }
      }).start();
    }
  }

}
```

9.4 Queue

> - Collection的子接口，表示队列FIFO（First In First Out）。

> **推荐方法：**

方法名	描述
boolean offer(E e)	顺序添加一个元素 （到达上限后，再添加则会返回false）。
E poll()	获得第一个元素并移除 （如果队列没有元素时，则返回null）。
E keep()	获得第一个元素但不移除 （如果队列没有元素时，则返回null）。

```
public class TestQueue {
  public static void main(String[] args) {
    //1创建队列
    Queue<String> queue=new LinkedList<>();
    //2入队
    queue.offer("苹果");
    queue.offer("橘子");
    queue.offer("葡萄");
    queue.offer("西瓜");
    queue.offer("榴莲");
    //3出队
    System.out.println(queue.peek());
    System.out.println("----------------");
    System.out.println("元素个数:"+queue.size());
    int size=queue.size();
    for(int i=0;i<size;i++) {
      System.out.println(queue.poll());
    }
    System.out.println("出队完毕:"+queue.size());

  }
}
```

9.5 ConcurrentLinkedQueue

> - 线程安全、可高效读写的队列，高并发下性能最好的队列。
> - 无锁、CAS比较交换算法，修改的方法包含三个核心参数（V,E,N）。
> - V：要更新的变量、E：预期值、N：新值。
> - 只有当V==E时，V=N；否则表示已被更新过，则取消当前操作。

```
public class TestConcsurrentLinkedQueue {
  public static void main(String[] args) throws Exception {
    //1创建安全队列
    ConcurrentLinkedQueue<Integer> queue=new ConcurrentLinkedQueue<>();
    //2入队操作
    Thread t1=new Thread(new Runnable() {

      @Override
      public void run() {
        for(int i=1;i<=5;i++) {
          queue.offer(i);
        }
      }
    });
    Thread t2=new Thread(new Runnable() {

      @Override
      public void run() {
        for(int i=6;i<=10;i++) {
          queue.offer(i);
        }
      }
    });
    //3启动线程
    t1.start();
    t2.start();

    t1.join();
    t2.join();
    System.out.println("-------------出队-------------");
    //4出队操作
    int size=queue.size();
    for(int i=0;i<size;i++) {
      System.out.println(queue.poll());
    }

  }
}
```

9.6 BlockingQueue

> - Queue的子接口，阻塞的队列，增加了两个线程状态为无限期等待的方法。
> - 可用于解决生产生、消费者问题。

> **推荐方法：**

方法名	描述
void put(E e)	将指定元素插入此队列中，如果没有可用空间，则等待。
E take()	获取并移除此队列头部元素，如果没有可用元素，则等待。

9.6.1 ArrayBlockingQueue

> - 数组结构实现，有界队列。
> - 手工固定上限。

```
public class TestArrayBlockingQueue {
  public static void main(String[] args) throws Exception{
    //创建一个有界队列，添加数据
    ArrayBlockingQueue<String> queue=new ArrayBlockingQueue<>(5);
    //添加元素
    queue.put("aaa");
    queue.put("bbb");
    queue.put("ccc");
    queue.put("ddd");
    queue.put("eee");
    //删除元素
    queue.take();
    System.out.println("已经添加了5个元素");
    queue.put("xyz");
    System.out.println("已经添加了6个元素");
    System.out.println(queue.toString());
  }
}
```

9.6.2 LinkedBlockingQueue

> - 链表结构实现，无界队列。
> - 默认上限Integer.MAX_VALUE。
> - 使用方法和ArrayBlockingQueue相同。

9.6.3 课堂案例

> 使用阻塞队列实现生产者和消费者。

```
public class Demo7 {
  public static void main(String[] args) {
    //1创建队列
    ArrayBlockingQueue<Integer> queue=new ArrayBlockingQueue<>(6);
    //2创建两个线程
    Thread t1=new Thread(new Runnable() {

      @Override
      public void run() {
        for(int i=0;i<30;i++) {
          try {
            queue.put(i);
            System.out.println(Thread.currentThread().getName()+"生产了第"+i+"号面包");
          } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
          }

        }
      }
    }, "晨晨");

    Thread t2=new Thread(new Runnable() {

      @Override
      public void run() {
        for(int i=0;i<30;i++) {
          try {
            Integer num=queue.take();
            System.out.println(Thread.currentThread().getName()+"消费了第"+i+"号面包");
          } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
          }

        }
      }
    }, "冰冰");

    //启动线程
    t1.start();
    t2.start();
  }
}
```