**20180606**

**22 多线程**

* 22.1 进程与线程

进程(Process)是资源(CPU、内存等)分配的基本单位，是程序执行时的一个实例。程序运行时系统会创建一个进程，为它分配资源，把该进程放入进程就绪队列，进程调度器选中它时就会为它分配CPU时间，程序开始真正运行。

单进程的特定的同一时间段只允许一个程序运行。多进程一个时间段可以运行多个程序，这些程序进行资源的轮流抢占(单核CPU)，同一个时间点只有一个进程运行。

线程(Thread)是程序执行流的最小单位，一个进程可以由多个线程组成，线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。

线程的启动速度比进程快许多，多线程进行并发处理性能高于多进程。

* 22.2 继承Thread类实现多线程

一个类继承了java.lang.Thread表示此类是线程的主体类，还需要覆写run()方法，run()方法属于线程的主方法。多线程要执行的内容都在run()方法内定义。

run()方法不能直接调用，因为牵扯到操作系统资源调度问题，使用start()方法启动多线程。

|  |
| --- |
| **class** MyThread **extends** Thread {  **private** String name;  **private** **int** x;  **public** MyThread(String name) {  **this**.name = name;  }  @Override  **public** **void** run() {  **while** (**this**.x < 5) {  System.***out***.println(**this**.name + ": " + **this**.x);  **this**.x++;  }  }  }  **new** MyThread("maki").start();  **new** MyThread("rin").start(); |

结果：

|  |
| --- |
| maki: 0  rin: 0  maki: 1  rin: 1  maki: 2  rin: 2  rin: 3  rin: 4  maki: 3  maki: 4 |

实例化对象调用start()方法，但是执行的是run()方法内容，所有线程交替执行，执行顺序不可控，打印结果随机。

为什么要使用start()方法启动多线程呢?

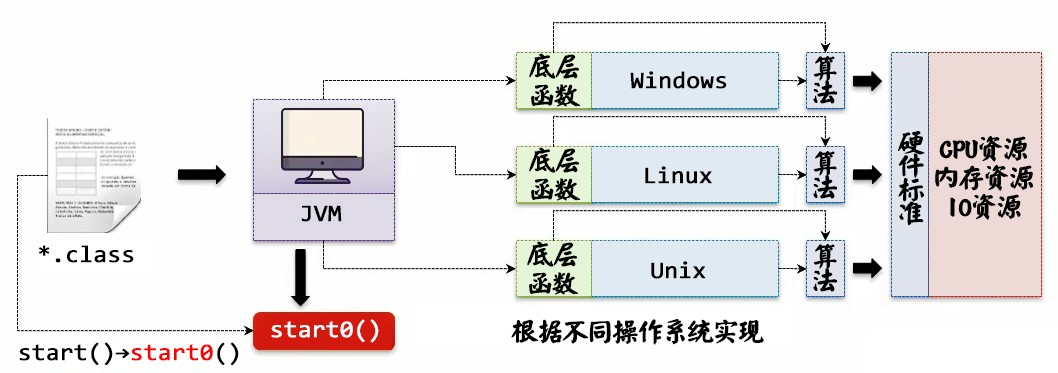
start()方法源代码：

|  |
| --- |
| public synchronized *void* start() {  if (threadStatus != 0) // 线程的状态0表示线程未开始  throw new IllegalThreadStateException();  group.add(this);  *boolean* started = false;  try {  start0();  started = true;  } finally {  try {  if (!started) {  group.threadStartFailed(this);  }  } catch (*Throwable* *ignore*) {}  }  }  private native *void* start0(); // 该方法没有方法体,没有实现 |

一个线程只能被启动一次，如果重复启动抛出IllegalThreadStateException异常。但没有throws声明或try-catch 处理，说明该异常是RuntimeException的子类

start()方法中又调用了start0()方法，此方法使用**native**关键字修饰。

Java支持本地操作系统函数调用，称为JNI (Java Native Interface)技术，但是Java开发中不推荐这样使用，利用JNI可以使用操作系统提供的底层函数。Thread类的start0()表示此方法依赖于不同的操作系统实现。



* 22.3 基于Runnable接口实现多线程

Java继承存在单继承的局限，实现java.lang.Runnable接口也可以实现多线程。

Runnable接口的定义：

|  |
| --- |
| *@FunctionalInterface* // 函数式接口  public interface Runnable {  public abstract *void* run();  } |

将MyThread改为实现Runnable接口：

|  |
| --- |
| **class** MyThread **implements** Runnable {  // 与之前一模一样...  } |

但是此时MyThread没有继承Thread，不能使用start()方法。

Thread类有一个构造方法可以接收Runnable对象作为参数：

|  |
| --- |
| public Thread(*Runnable* target) {  init(null, target, "Thread-" + nextThreadNum(), 0);  } |

启动多线程：

|  |
| --- |
| **new** Thread(**new** MyThread("maki")).start();  **new** Thread(**new** MyThread("rin")).start(); |

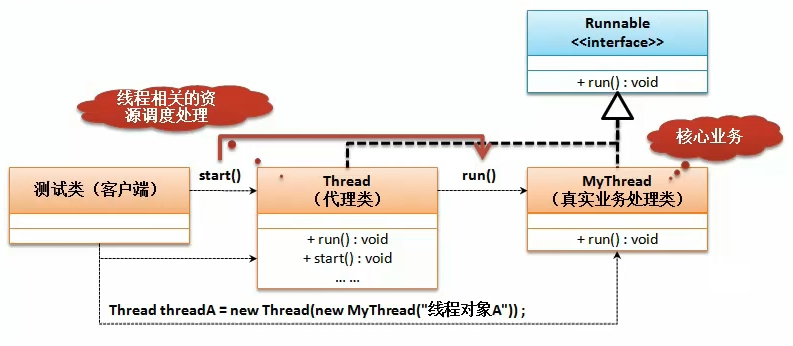
JDK 1.8开始Runnable使用了函数式接口定义，可以使用Lambda表达式定义多线程类。

|  |
| --- |
| **for** (**int** i = 0; i < 3; i++) { // 3个线程  String name = "线程-" + i;  Runnable run = () -> { // Runnable对象  **for** (**int** j = 0; j < 5; j++) {  System.***out***.println(name + ": " + j);  }  };  // 始终使用Thread对象start()方法启动多线程  **new** Thread(run).start();  } |

也可以不定义run变量，直接将其右边传入Thread的构造方法。

* 22.4 Thread类和Runnable接口关系

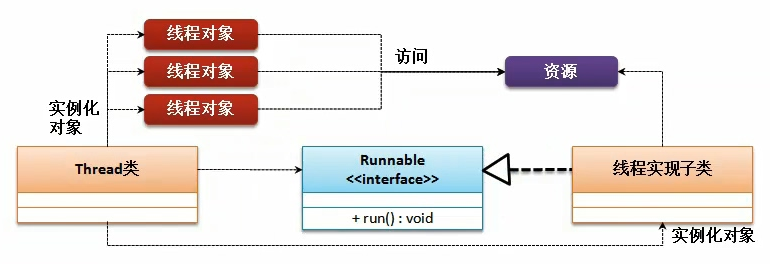
Thread类是Runnable接口的子类



多线程设计使用了代理设计模式的结构，用户自定义的线程主体只是负责核心业务的实现，所有的辅助实现都由Thread类处理。

通过Thread类的构造方法传递Runnable对象时，该对象被Thread的target属性保存。Thread启动多线程调用start()方法，start()调用run()方法，此Thread类的run()方法又去调用Runnable对象的run()方法。

多线程开发本质的多个线程可以进行同一资源的抢占。Thread主要描述线程，Runnable主要描述资源，因为*n*个Thread对象的target属性都指向了同一个Runnable对象。



* 22.5 Callable实现多线程

Runnable接口的缺点是当线程执行完毕无法获取返回值。JDK 1.5提出新的线程实现接口java.util.concurrent.Callable：

|  |
| --- |
| *@FunctionalInterface*  public interface Callable<*V*> {  *V* call() throws Exception;  } |

Callable对象可以作为FutureTask构造方法的参数保存为callable属性。

FutureTask是RunnableFuture接口的子类。

|  |
| --- |
| public class FutureTask<*V*> implements *RunnableFuture*<*V*> {  // ...  public FutureTask(*Callable*<*V*> *callable*) {  if (callable == null)  throw new NullPointerException();  this.callable = callable;  this.state = NEW; // ensure visibility of callable  }  // ...  } |

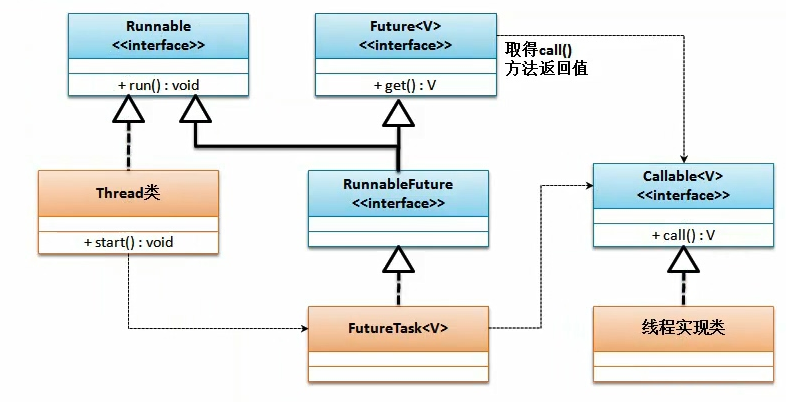
RunnableFuture接口继承于Runnable接口和Future接口。

|  |
| --- |
| public interface Runnable Future<*V*> extends *Runnable*, *Future*<*V*> {  *void* run();  } |

FutureTask类覆写了Future接口的get()方法，可以获取callable属性调用call()方法的返回值。

FutureTask类也是Runnable接口的子类，可以作为Thread构造方法的参数。

关系有点复杂：



示例：用Callable实现龟兔赛跑

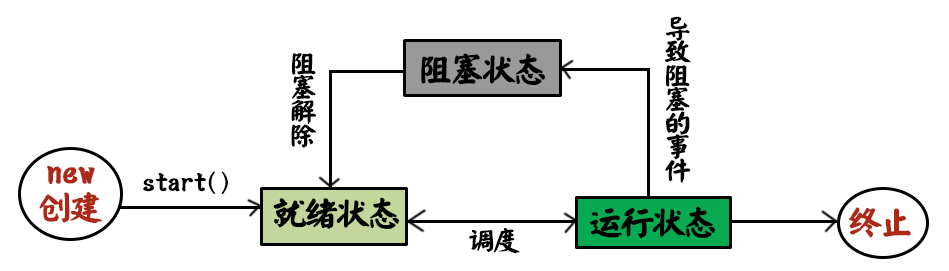
|  |
| --- |
| **class** Race **implements** Callable<Integer> {  **private** String name;  **private** **long** time; // 多少毫秒走一步  **private** **int** step; // 步数  **private** **boolean** flag = **true**; // 设为false结束线程  **public** Race(String name, **long** time) {  **this**.name = name;  **this**.time = time;  }  **public** **void** setFlag(**boolean** flag) {  **this**.flag = flag;  }  @Override  **public** Integer call() **throws** Exception {  **while** (flag) {  Thread.*sleep*(**this**.time);  **this**.step++;  System.***out***.println(Thread.*currentThread*().getName() + " " + **this**.name + ": " + **this**.step);  }  **return** **this**.step;  }  }  // 客户端代码太多了...  Race tortoise = **new** Race("乌龟", 2000);  Race rabbit = **new** Race("兔子", 500);  FutureTask<Integer> task1 = **new** FutureTask<>(tortoise);  FutureTask<Integer> task2 = **new** FutureTask<>(rabbit);  **new** Thread(task1).start();  **new** Thread(task2).start();  Thread.*sleep*(10000); // 跑10秒  tortoise.setFlag(**false**);  rabbit.setFlag(**false**);  System.***out***.println("10秒后, 乌龟:" + task1.get());  System.***out***.println("10秒后, 兔子:" + task2.get()); |

结果：

|  |
| --- |
| 10秒后, 乌龟:5  10秒后, 兔子:20 |

* 22.6 多线程运行状态

定义线程主体类，通过Thread类start()方法启动线程，但并不是调用start()方法线程就开始运行。



1) 任何一个线程对象需要使用Thread类封装，线程启动使用start()方法；但是启动时所有线程进入就绪状态，并没有执行；

2) 等待资源调度，某个线程调度成功则进入运行状态(run()方法)；但是一个线程不可能一直执行下去，执行一段时间之后就会让出资源进入阻塞状态，随后重新回到就绪状态；

3) run()方法执行完毕，线程任务结束，此时进入停止状态。

**20180607**

* 22.7 Thread类常用方法

① 线程的命名与取得

多线程运行状态不确定，所有线程的名字是个重要的属性。

|  |
| --- |
| public Thread(*Runnable* target, *String* name) // 构造方法可以自定义线程名  public final *void* setName(*String* name) // 设置线程名字  public final *String* getName() // 获取线程名字 |

线程对象的获取不可能只靠this完成，因为线程状态不可控，但是所有线程一定会执行run()方法，则可以考虑获取当前线程。

|  |
| --- |
| public static *Thread* currentThread() // 返回当前正在执行线程的引用 |

示例：自定义线程名字和获取当前线程名

|  |
| --- |
| **class** MyThread **implements** Runnable{  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName());  }  }  MyThread mt=**new** MyThread();  **new** Thread(mt,"线程1").start(); // 自定义线程名字  **new** Thread(mt).start();  **new** Thread(mt).start();  **new** Thread(mt,"线程2").start(); // 自定义线程名字  mt.run(); // main |

结果：

|  |
| --- |
| 线程1  Thread-0  线程2  Thread-1  main |

如果没有设置线程名字，会自动生成一个不重复的名字。

|  |
| --- |
| // 匿名线程使用类静态属性自动编号  **private** **static** **int** *threadInitNumber*;  **private** **static** **synchronized** **int** nextThreadNum() {  **return** *threadInitNumber*++;  } |

直接执行mt.run()就是在主方法中调用线程对象的run()方法，获得线程名字为main，所以主方法也是一个线程。

每当使用java命令执行程序时就启动了一个JVM的进程，一台电脑可以同时启动多个JVM进程，一个JVM进程都有各自的线程。

主线程可以创建若干子线程，主要将一些复杂逻辑或耗时操作交给子线程处理。

|  |
| --- |
| System.***out***.println("吃饭");  **new** Thread(() -> {  // 模拟耗时操作,耗时操作交给子线程完成  **double** pi = 0;  **double** flag = 1;  **for** (**int** i = 1; i < 1e9; i += 2) {  pi += flag / i;  flag = -flag;  }  pi \*= 4;  System.***out***.println("pi=" + pi);  }).start();  System.***out***.println("睡觉"); |

主线程复杂整体流程，子线程负责处理耗时操作。

② 线程的休眠 (sleep)

|  |
| --- |
| public static *void* sleep​(*long* millis) throws *InterruptedException*  public static *void* sleep​(*long* millis, *int* nanos) throws *InterruptedException* |

休眠时可能产生中断异常InterruptedException，是Exception的子类，说明该异常必须被处理。

多线程休眠有先后顺序，一个线程休眠会释放执行权，其他线程抢占资源。

|  |
| --- |
| Runnable run = () -> {  **for** (**int** i = 0; i < 10; i++) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + ", i=" + i);  }  };  // 5个子线程几乎同时休眠同时唤醒, 因为执行太快了; 差别是每轮打印顺序都不一样  **for** (**int** n = 0; n < 5; n++) {  **new** Thread(run, "hikari - " + n).start();  } |

③ 线程中断 (interrupt)

线程休眠可能会产生中断异常，也就是线程休眠可能被其他线程打断。

|  |
| --- |
| public *boolean* isInterrupted() // 判断线程是否被中断  public *void* interrupt() // 中断该线程 |

示例：main线程中止子线程

|  |
| --- |
| Thread t = **new** Thread(() -> {  System.***out***.println("该睡觉了...");  **try** {  Thread.*sleep*(10000); // 预计休眠10s  System.***out***.println("醒来...");  } **catch** (InterruptedException e) {  System.***out***.println("草泥马, 不要打扰劳资睡觉!");  }  });  t.start();  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {}  **if** (!t.isInterrupted()) {  // 如果线程没被中止, 则中止它; main线程中止子线程  System.***out***.println("小伙子, 该醒了!");  t.interrupt();  } |

④ 线程强制执行 (join)

当满足某些条件后，某个线程对象一直独占资源，直到该线程执行结束。

|  |
| --- |
| public final *void* join() throws *InterruptedException* |

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  // sleep()和join()都会抛出异常, 直接在方法上声明吧...  Thread t = **new** Thread(() -> {  **for** (**int** x = 0; x < 100; x++) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + ", x=" + x);  }  }, "子线程");  t.start();  **for** (**int** x = 0; x < 100; x++) {  **if** (x == 10) {  t.join(); // 主线程等待主线程t执行完再执行  }  Thread.*sleep*(100);  System.***out***.println("【main线程】, x=" + x);  }  } |

⑤ 线程的礼让 (yield)

线程的礼让是将资源让给其他线程先执行。

|  |
| --- |
| public static *void* yield() |

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  Thread t = **new** Thread(() -> {  **for** (**int** x = 0; x < 100; x++) {  **if** (x%5==0) {  // 每次x为5的倍数时,子线程让出执行权  Thread.*yield*(); // 静态方法  System.***out***.println("\*\*\*\*\*子线程礼让\*\*\*\*\*");  }  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + ", x=" + x);  }  }, "子线程");  t.start();  **for** (**int** x = 0; x < 100; x++) {  Thread.*sleep*(100);  System.***out***.println("【main线程】, x=" + x);  }  } |

礼让执行每次调用yield()方法只会礼让一次。该方法很少使用。

⑥ 线程优先级

理论上线程的优先级越高越可以先执行(抢占到资源)。

|  |
| --- |
| public final *void* setPriority(*int* newPriority)  public final *int* getPriority() |

优先级定义使用int，Thread类定义三个与优先级的int常量：

|  |
| --- |
| public static final *int* MAX\_PRIORITY = 10; // 最高优先级  public static final *int* NORM\_PRIORITY = 5; // 中等优先级  public static final *int* MIN\_PRIORITY = 1; // 最低优先级 |

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  System.***out***.println("主线程优先级: " + Thread.*currentThread*().getPriority());  Runnable run = () -> {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + " " + i);  }  };  Thread t1 = **new** Thread(run, "子线程01");  Thread t2 = **new** Thread(run, "子线程02");  Thread t3 = **new** Thread(run, "子线程03");  System.***out***.println("新创建的线程对象的优先级: " + t3.getPriority());  t1.setPriority(Thread.***MAX\_PRIORITY***); // 将子线程01优先级设为最高  t1.start();  t2.start();  t3.start();  }  } |

子线程和默认创建的子线程优先级都是5。

**20180608**

* 22.8 线程同步

多线程中可以利用Runnable描述多线程操作的资源，Thread描述每个线程对象；当多个线程访问同一资源时，就会产生数据操作错误。

|  |
| --- |
| **public** **class** Ticket **implements** Runnable {  **private** **int** ticket = 5;  @Override  **public** **void** run() {  **while** (**true**) {  **if** (**this**.ticket > 0) {  **try** {  Thread.*sleep*(100); // 模拟网络延时  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + ": ticket num=" + **this**.ticket--);  } **else** {  System.***out***.println("\*\*\*\*\*票卖完了\*\*\*\*\*");  **break**;  }  }  }  }  Runnable run=**new** Ticket();  **new** Thread(run,"票贩子A").start();  **new** Thread(run,"票贩子B").start();  **new** Thread(run,"票贩子C").start(); |

比较牛B的结果：

|  |
| --- |
| 票贩子C: ticket num=4  票贩子B: ticket num=5  票贩子A: ticket num=3  票贩子A: ticket num=2  票贩子B: ticket num=1  \*\*\*\*\*票卖完了\*\*\*\*\*  票贩子C: ticket num=0  \*\*\*\*\*票卖完了\*\*\*\*\*  票贩子A: ticket num=-1  \*\*\*\*\*票卖完了\*\*\*\*\* |

当ticket=1时，票贩子B进入if判断进行休眠，因为ticket还是1，随后票贩子C和A也进入if 判断进行休眠；票贩子B休眠结束打印1，ticket变为0；票贩子C休眠结束打印0，ticket变为-1；票贩子A休眠结束打印-1，ticket变为-2。

出现问题的原因是线程休眠前通过if判断，但唤醒时已经不符合条件了。需要将一次卖票的动作同步处理，一个线程卖票时，其他线程需要外面等待。

同步是指多个操作在同一时间只能有一个线程进行，其他线程等待此线程完成后才能执行。同步的关键是锁，使用**synchronized**关键字可以定义同步方法和同步代码块。

同步代码块：

|  |
| --- |
| @Override  **public** **void** run() {  **while** (**true**) {  **synchronized** (**this**) { // 同步对象一般使用this  **if** (**this**.ticket > 0) {// ...} **else** {// ...}  }  }  } |

**synchronized**不要放在run()方法上或把while放进同步代码块，否则只会有一个线程执行完成后，另外两个再开始已经不满足if条件，打印后退出。

同步方法：

|  |
| --- |
| **public** **class** Ticket **implements** Runnable {  **private** **int** ticket = 100;  **private** **boolean** flag = **true**;  **private** **synchronized** **void** sell() { // 同步方法  **if** (**this**.ticket > 0) {  **try** {  Thread.*sleep*(100); // 模拟网络延时  } **catch** (InterruptedException e) {}  System.***out***.println(Thread.*currentThread*().getName() + ": ticket num=" + **this**.ticket--);  } **else** {  **this**.flag = **false**;  System.***out***.println("\*\*\*\*\*票卖完了\*\*\*\*\*");  }  }  @Override  **public** **void** run() {  **while** (**this**.flag) {  **this**.sell();  }  }  } |

系统许多类同步处理都使用同步方法。使用同步会造成程序性能降低。

* 22.9 死锁

死锁是多线程同步处理时可能产生的问题，死锁是几个线程互相等待的状态。

无意义的死锁示例：

|  |
| --- |
| **public** **class** DeadLock **implements** Runnable {  **private** **static** **final** Robber ***r*** = **new** Robber();  **private** **static** **final** Person ***p*** = **new** Person();  **private** **boolean** flag = **true**;  **public** DeadLock() {}  **public** DeadLock(**boolean** flag) {  **this**.flag = flag;  }  @Override  **public** **void** run() {  **if** (**this**.flag) {  ***r***.say(***p***); // 一个线程抢占了r, 继续执行需要p;  } **else** {  ***p***.say(***r***); // 另一个线程抢占了p, 继续执行需要r, 双方一直僵持, 互不相让  }  }  }  **class** Robber {  **public** **synchronized** **void** say(Person p) {  System.***out***.println("此山是我开, 此树是我栽, 要想打此过, 留下买路财!");  p.after();  }  **public** **synchronized** **void** after() {  System.***out***.println("喽啰们让路!");  }  }  **class** Person {  **public** **synchronized** **void** say(Robber r) {  System.***out***.println("先让我走, 再给钱!");  r.after();  }  **public** **synchronized** **void** after() {  System.***out***.println("别打了,我给钱...");  }  }  **new** Thread(**new** DeadLock()).start();  **new** Thread(**new** DeadLock(**false**)).start(); |

造成死锁主要原因是线程互相等待对方先让出资源。死锁是开发中出现的不确定状态，如果代码处理不当会不定期出现死锁，属于正常开发的调试问题。而示例是强行死锁，不具备参考性。

* 22.10 综合案例：生产者与消费者

多线程开发最著名的案例就是生产者与消费者：生产者负责信息内容的生产；每当生产者生产一项完整的信息，消费者取走信息。

如果生产者没有生产完成，消费者需要等待后再消费；反之，如果消费者没有处理完成，生产者也需要等待后再生产。

生产者和消费者定义为两个独立的线程对象，定义Resource类实现两个线程的数据保存。

基本模型：

|  |
| --- |
| **public** **class** Resource {  **private** String name;  **private** String group;  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **void** setGroup(String group) {  **this**.group = group;  }  **public** String toString() {  **return** name + ">>>" + group;  }  }  **public** **class** Producer **implements** Runnable {  **private** Resource src;  **public** Producer(Resource src) {  **this**.src = src;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  **if** (i % 2 == 0) {  **this**.src.setName("星空凛");  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {}  **this**.src.setGroup("lily white");  } **else** {  **this**.src.setName("西木野真姫");  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {}  **this**.src.setGroup("BiBi");  }  }  }  }  **public** **class** Consumer **implements** Runnable {  **private** Resource src;  **public** Consumer(Resource src) {  **this**.src = src;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {}  System.***out***.println(**this**.src);  }  }  }  Resource src = **new** Resource();  **new** Thread(**new** Producer(src)).start();  **new** Thread(**new** Consumer(src)).start(); |

不使用同步打印错乱：

|  |
| --- |
| 西木野真姫>>>lily white  星空凛>>>BiBi  西木野真姫>>>lily white  星空凛>>>BiBi  ... |

生产者线程开始设置name为rin，休眠；消费者线程也休眠；生产者线程醒来，group设为lily white，此时消费者线程仍在休眠，生产者线程进入下一轮循环，设置name为maki，休眠；消费者线程醒来，打印maki>>>lily white...

使用同步：

|  |
| --- |
| **public** **class** Resource {  **private** String name;  **private** String group;  **private** **int** num; // 计数  **public** **synchronized** **void** set(String name, String group) {  **this**.name = name;  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {}  **this**.group = group;  num++;  }  @Override  **public** **synchronized** String toString() { // 给toString()添加同步没问题?  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {}  **return** name + ">>>" + group + ">>>" + num;  }  }  **public** **class** Producer **implements** Runnable {  // ...  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  **if** (i % 2 == 0) {  src.set("星空凛", "lily white");  } **else** {  src.set("西木野真姫", "BiBi");  }  }  }  }  **public** **class** Consumer **implements** Runnable {  // ...  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(**this**.src);  }  }  } |

打印结果：

|  |
| --- |
| 星空凛>>>lily white>>>3  西木野真姫>>>BiBi>>>12  西木野真姫>>>BiBi>>>14  ...  星空凛>>>lily white>>>89  西木野真姫>>>BiBi>>>100  西木野真姫>>>BiBi>>>100  ... |

因为生产者生产得太快了，消费者来不及消费，后期生产者已经完成生产，消费者每次消费都使用最后生产的数据。

* 22.11 线程等待与唤醒

解决生产者和消费者问题最好的方法是使用等待与唤醒机制，主要使用Object类几个的方法：

|  |
| --- |
| public final *void* wait() throws InterruptedException // 死等  public final *void* wait(*long* timeout) throws InterruptedException // 设置等待时间  public final *void* notify() // 唤醒任意一个等待线程  public final *void* notifyAll() // 唤醒全部等待线程 |

示例：使用等待唤醒机制

|  |
| --- |
| **public** **class** Resource {  **private** String name;  **private** String group;  **private** **int** num;  **private** **boolean** flag = **true**; // true表示允许生产不允许消费  **public** **synchronized** **void** set(String name, String group) {  **if** (!**this**.flag) { // 无法生产, 等待  **try** {  **super**.wait();  } **catch** (InterruptedException e) {}  }  **this**.name = name;  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {}  **this**.group = group;  num++;  **this**.flag = **false**; // 生产完成  **super**.notify();  }  @Override  **public** **synchronized** String toString() {  **if** (**this**.flag) { // 正在生产,不能消费  **try** {  **super**.wait();  } **catch** (InterruptedException e) {}  }  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {}  **try** {  **return** name + ">>>" + group + ">>>" + num;  } **finally** {  **this**.flag = **true**; // 消费完,可以生成产了  **super**.notify();  }  }  } |

结果：

|  |
| --- |
| 星空凛>>>lily white>>>1  西木野真姫>>>BiBi>>>2  ...  星空凛>>>lily white>>>99  西木野真姫>>>BiBi>>>100 |

这是多线程最原始处理方案，整个等待、同步、唤醒机制都是开发者自行通过原生代码实现控制。

* 22.12 优雅地停止线程

Thread类原本提供停止线程的stop()方法，但从JDK 1.2开始就已经废除了，直到现在也不推荐使用。

以下方法已全部废除，原因是可能会导致死锁。

|  |
| --- |
| 1) stop()：停止线程  2) destroy()：销毁线程  3) suspend()：挂起线程，暂停执行  4) resume()：恢复挂起线程，继续执行 |

示例：优雅地停止线程

|  |
| --- |
| **public** **class** ElegantStop **implements** Runnable {  **private** **boolean** flag = **true**;  **private** **int** num = 0;  **public** **void** setFlag(**boolean** flag) {  **this**.flag = flag;  }  @Override  **public** **void** run() {  **while** (flag) {  **try** {  Thread.*sleep*(200);  } **catch** (InterruptedException e) {}  System.***out***.println(Thread.*currentThread*().getName() + ", num=" + **this**.num++);  }  }  **public** **void** stop() { // flag设为false, run()方法结束循环, 线程结束  **this**.setFlag(**false**);  }  }  ElegantStop run = **new** ElegantStop();  **new** Thread(run).start();  **try** {  Thread.*sleep*(2000); // 主线程休眠2s后  } **catch** (InterruptedException e) {}  run.stop(); // 将子线程停止 |

设置flag字段，true表示线程执行，其他线程修改flag为false，线程停止。

* 22.13 守护线程

主线程或其他线程在执行时，守护线程将一直存在，并后台运行。如果程序执行完毕，守护线程也就消失。JVM最大的守护线程是GC线程。

|  |
| --- |
| public final *void* setDaemon(*boolean* on) // 设置为守护线程  public final *boolean* isDaemon() // 判断是否为守护线程 |

示例：

|  |
| --- |
| ElegantStop run = **new** ElegantStop();  ElegantStop daemon = **new** ElegantStop();  Thread userThread = **new** Thread(run, "用户线程");  Thread daemonThread = **new** Thread(daemon, "守护线程");  userThread.start();  daemonThread.setDaemon(**true**); // 启动之前设置为守护线程  daemonThread.start(); |

如果不停止用户线程，用户线程和守护线程将比翼双飞；如果优雅地停止用户线程，守护线程乘风而去。

* 22.14 volatile关键字

多线程中**volatile**关键字主要定义属性，表示该属性为直接数据操作，而不进行副本的拷贝处理。一些书错误地理解为~~同步属性~~。

正常进行变量处理步骤：

1) 获取变量原有的数据内容副本；

2) 利用副本为变量进行数学计算；

3) 将计算后的变量保存到原始空间中。

如果一个属性定义了**volatile**关键字，表示不使用副本，直接操作原始变量，相当于节约了拷贝副本、重新保存的步骤。

面试题：**volatile**和**synchronized**的区别

1) **volatile**主要在属性上使用；**synchronized**在代码块和方法上使用；

2) **volatile**无法描述同步，只是一种直接内存处理，避免副本的操作；**synchronized**实现同步。两者没什么联系。

* 22.15 多线程综合案例：数字加减

设4个线程对象，两个执行加操作，两个执行减操作，加减交替执行。

|  |
| --- |
| **public** **class** Resource { // 资源类  **private** **int** num = 0;  **private** **boolean** flag = **true**; // true为只执行加法,false为只执行减法  **public** **synchronized** **void** add() {  **while** (!**this**.flag) { // 此处需要while而不是if  **try** {  **super**.wait(); // 执行减法,加法休眠  } **catch** (InterruptedException e) {}  }  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {}  System.***out***.println(Thread.*currentThread*().getName() + ", num=" + ++**this**.num);  **this**.flag = !**this**.flag; // 加法执行完毕,需要执行减法  **super**.notifyAll(); // 唤醒全部等待线程  }  **public** **synchronized** **void** sub() {  **while** (**this**.flag) {  **try** {  **super**.wait();  } **catch** (InterruptedException e) {}  }  **try** {  Thread.*sleep*(200);  } **catch** (InterruptedException e) {}  System.***out***.println(Thread.*currentThread*().getName() + ", num=" + --**this**.num);  **this**.flag = !**this**.flag;  **super**.notifyAll();  }  }  // 加法线程  **public** **class** AddThread **implements** Runnable {  **private** Resource src;  **public** AddThread(Resource src) {  **this**.src = src;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 50; i++) {  src.add();  }  }  }  // 减法线程  **public** **class** SubThread **implements** Runnable {  **private** Resource src;  **public** SubThread(Resource src) {  **this**.src = src;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 50; i++) {  src.sub();  }  }  }  // 测试  Resource src = **new** Resource();  AddThread add = **new** AddThread(src);  SubThread sub = **new** SubThread(src);  **new** Thread(add, "加法线程1").start();  **new** Thread(add, "加法线程2").start();  **new** Thread(sub, "减法线程1").start();  **new** Thread(sub, "减法线程2").start(); |

结果：加减法交替执行，num的值为1或0

**注**：此处每种线程有多个，判断flag要使用while而不是if。否则可能会出现加法之后还是加法或减法之后还是减法。如：

|  |
| --- |
| t1 --> num=1, flag=false --> notifyAll()  t2 --> wait()  t3 --> num=0, flag=true --> notifyAll()  t1 --> num=1, flag=false --> notifyAll()  t2 --> num=2, flag=false --> notifyAll()  ... |

t2醒来的时候虽然flag=false，但之前已经过了if判断，还是做加法，num=2，这样就有问题了；所以线程唤醒的时候也需要判断flag，故用while。

* 22.16 Lock接口

**synchronized**对锁的操作的隐式的。JDK 1.5出现java.util.concurrent.locks包，其中的Lock接口将**synchronized**的隐式操作变为显式的锁操作，使用更加灵活，一个锁可以有多组监视器。

|  |
| --- |
| *void* lock() // 获取锁  *void* unlock() //释放锁, 通常定义在finally代码块中 |

ReentrantLock类是Lock接口的常用实现。

Lock实例的newCondition()返回此Lock实例绑定的Condition实例：

|  |
| --- |
| *Condition* newCondition() |

Condition接口将对象监视器方法(wait、notify、notifyAll)分解为不同的对象，与任意Lock实例组合，使每个对象具有多个等待集的效果。

当Lock代替同步方法或同步代码块，要使用Condition代替对象监视器方法：

|  |
| --- |
| *void* await() throws InterruptedException // 当前线程休眠  *void* signal() // 唤醒一个等待的线程  *void* signalAll() // 唤醒所有等待的线程 |

示例：竞拍抢答

设置3个抢答者(线程)，同时发出抢答指令，抢答成功显示成功，反之显示失败。

|  |
| --- |
| // 线程要返回结果实现Callable  **class** MyThread **implements** Callable<String> {  **private** **boolean** flag = **true**;  **private** Lock lock = **new** ReentrantLock(); // 可重入锁  @Override  **public** String call() **throws** Exception {  **this**.lock.lock(); // 获取锁  **try** {  **if** (**this**.flag) {  **this**.flag = **false**;  **return** Thread.*currentThread*().getName() + ": 抢答成功!";  }  **return** Thread.*currentThread*().getName() + ": 抢答失败!";  } **finally** {  **this**.lock.unlock(); // 释放锁  }  }  }  MyThread mt = **new** MyThread();  FutureTask<String> task1 = **new** FutureTask<>(mt);  FutureTask<String> task2 = **new** FutureTask<>(mt);  FutureTask<String> task3 = **new** FutureTask<>(mt);  **new** Thread(task1, "maki").start();  **new** Thread(task2, "rin").start();  **new** Thread(task3, "nozomi").start();  System.***out***.println(task1.get());  System.***out***.println(task2.get());  System.***out***.println(task3.get()); |

每次结果不一样：

|  |
| --- |
| maki: 抢答失败!  rin: 抢答成功!  nozomi: 抢答失败! |

**20180609**

**♊** のーぞーみーちゃんっ！誕生日おめでとうっ♪



希パワー注入♪

**23 Java常用类库**

* 23.1 StringBuffer类

String类是开发中一定会用到的类，其特点：

1) 每个字符串常量都是一个String类的匿名对象，内容不可变；

2) String有两个常量池：静态常量池和运行时常量池；

3) String对象实例化建议直接赋值而不是new。

String类最大的缺点是对象内容不可修改。StringBuffer类解决了此问题，可以实现字符串内容的修改。

常用方法：

|  |
| --- |
| // 1. 构造方法  public StringBuffer()  public StringBuffer(*String* str) // 接收初始化字符串内容  // 2. 插入数据  public java.lang.AbstractStringBuilder append(*DataType* data) // 不能是byte或short  public *StringBuffer* insert(*int* offset, *DataType* data) // 指定位置插入  // 3. 删除数据  public *StringBuffer* delete(*int* start, *int* end)  // 4. 字符串反转  public *StringBuffer* reverse() |

示例：

|  |
| --- |
| StringBuffer sb = **new** StringBuffer();  sb.append("hello ").append("world").append("!");  System.***out***.println(sb.toString()); // hello world!  sb.delete(6, 11).insert(6, "hikari");  System.***out***.println(sb.toString()); // hello hikari!  System.***out***.println(sb.reverse().toString()); // !irakih olleh  System.***out***.println(sb.toString()); // !irakih olleh |

JDK 1.5提供的StringBuilder类和StringBuffer类功能相同，但是StringBuffer的全部使用**synchronized**关键字，是线程安全的。

JDK 1.4提供的CharSequence接口是描述字符串结构的接口。 String、StringBuffer、StringBuilder是其常用的3个子类。

* 23.2 AutoCloseable接口

JDK 1.7提供的AutoCloseable接口用于实现资源的自动关闭(释放)，如文件、网络、数据库资源的关闭。

该接口只有一个方法：

|  |
| --- |
| *void* close() throws *Exception* |

实现自动关闭，除了要实现AutoCloseable接口，还需要结合异常处理语句完成。

|  |
| --- |
| **interface** IMessage **extends** AutoCloseable {  **void** send(String msg);  }  **class** Message **implements** IMessage {  **public** **boolean** connect() {  System.***out***.println("【connect】成功连接!");  **return** **true**;  }  @Override  **public** **void** send(String msg) {  **if** (**this**.connect()) {  System.***out***.println("发送消息: " + msg);  }  }  @Override  **public** **void** close() **throws** Exception {  System.***out***.println("【close】关闭连接!");  }  }  // 需要结合异常处理语句完成自动关闭  **try** (Message m = **new** Message()) {  m.send("hello!");  } **catch** (Exception e) {} |

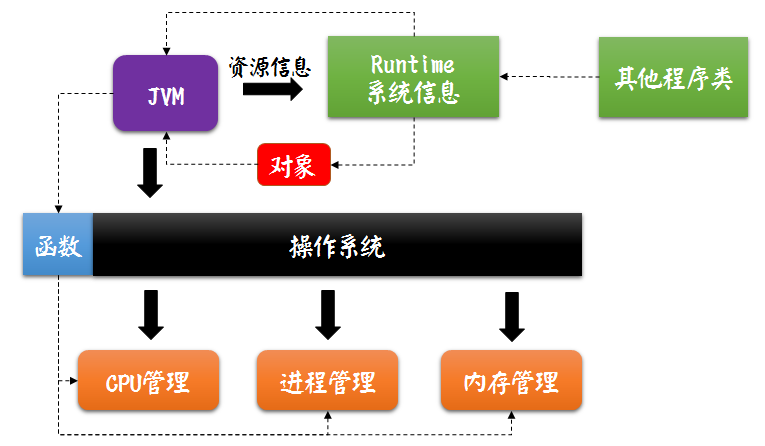
结果：

|  |
| --- |
| 【connect】成功连接!  发送消息: hello!  【close】关闭连接! |

* 23.3 Runtime类

Runtime类是唯一一个描述JVM运行状态的类，默认提供一个实例化对象。

每个JVM进程只允许提供一个Runtime对象，所以其构造方法private，使用单例设计模式，使用getRuntime()静态方法获取其实例化对象。



常用方法：

|  |
| --- |
| public *int* availableProcessors() // 返回可用JVM处理器数量，CPU内核数  public *long* maxMemory() // 返回JVM可以使用最大内存量  public *long* totalMemory() // 返回JVM中总内存量，值可能随时间变化，默认本机内存1/4  public *long* freeMemory() // 返回JVM可用内存量，默认本机内存1/64  public *void* gc()：// 运行垃圾回收器，手动调用GC回收垃圾 |

示例：

|  |
| --- |
| **int** MB = 1024 \* 1024;  Runtime run = Runtime.*getRuntime*();  System.***out***.println("CPU内核数: " + run.availableProcessors()); // 4, CPU内核数  System.***out***.println(run.maxMemory() / MB + "MB"); // 获取最大可用内存  System.***out***.println(run.totalMemory() / MB + "MB"); // 总可用内存  System.***out***.println(run.freeMemory() / MB + "MB"); // 空闲内存空间  String s = "";  // 制造大量垃圾  **for** (**int** i = 0; i < 10000; i++) {  s += i;  }  System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.***out***.println(run.maxMemory() / MB + "MB");  System.***out***.println(run.totalMemory() / MB + "MB");  System.***out***.println(run.freeMemory() / MB + "MB");  run.gc(); // 手动垃圾回收  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {}  System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.***out***.println(run.maxMemory() / MB + "MB");  System.***out***.println(run.totalMemory() / MB + "MB");  System.***out***.println(run.freeMemory() / MB + "MB"); |

结果：

|  |
| --- |
| CPU内核数: 4  996MB  64MB  62MB  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  996MB  124MB  119MB  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  996MB  8MB  7MB |

* 23.4 System类

① 数组拷贝

|  |
| --- |
| public static *void* arraycopy(*Object* src, *int* srcPos, *Object* dest, *int* destPos, *int* length) |

示例：

|  |
| --- |
| **int**[] arr = **new** **int**[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };  **int**[] lst = **new** **int**[] { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };  // arr从下标1开始复制到lst的下标2, 复制长度为5  System.*arraycopy*(arr, 1, lst, 2, 5);  hikari.ArrayUtil.*print*(lst); // [9, 8, 1, 2, 3, 4, 5, 2, 1, 0] |

② 返回当前UTC时间戳

|  |
| --- |
| public static *long* currentTimeMillis() // 毫秒  public static *long* nanoTime() // 纳秒 |

示例：计算程序运行时间

|  |
| --- |
| **private** **static** **void** test() {  String s = "";  **for** (**int** i = 0; i < 100000; i++) {  s += i;  }  }  **long** start = System.*currentTimeMillis*();  *test*();  **long** end = System.*currentTimeMillis*();  System.***out***.println(String.*format*("test()方法用时: %.2fs", (end-start)/1000.0)); // test()方法用时: 10.21s |

③ 垃圾回收

|  |
| --- |
| public static *void* gc()  // System.gc()就是调用Runtime.getRuntime().gc() |

* 23.5 Cleaner类

java.lang.ref包的Cleaner类是JDK 1.9提供的一个对象清理操作，主要功能是finalize()方法的替代。

C++有两种特殊函数：构造函数、析构函数(对象手动回收)。

Java所有垃圾都是GC自动回收，所以很多时候不需要析构函数。

但是Java提供了给对象收尾的操作(临终遗言)

Object类的finalize()方法：

|  |
| --- |
| *@Deprecated*(since="9")  protected *void* finalize() throws *Throwable* |

最大特点是抛出Throwable类型，它是Error和Exception的直接父类。

以前finalize的使用：

|  |
| --- |
| **class** Person {  **public** Person() {  System.***out***.println("【构造】在一个雷电交加的夜晚诞生了...");  }  @Override  **protected** **void** finalize() **throws** Throwable {  **try** {  System.***out***.println("【析构】人终有一死, 或重于泰山, 或重于另一座山...");  **throw** **new** Exception("我真的还想再活500年!");  }**finally** {  **super**.~~finalize~~();  }  }  }  Person p = **new** Person();  p = **null**; // 成为垃圾  System.*gc*(); // 手动回收打印遗言 |

结果：

|  |
| --- |
| 【构造】在一个雷电交加的夜晚诞生了...  【析构】人终有一死, 或重于泰山, 或重于另一座山... |

Finalization可能导致性能问题、死锁、和挂起，finalizers的错误可能导致资源泄露。占用非堆资源的实例的类需要提供一个方法来显式释放资源，按需实现AutoCloseable接口。当一个对象不可访问时，Cleaner和PhantomReference提供更加灵活有效的方法释放资源。

|  |
| --- |
| **import** java.lang.ref.Cleaner;  **public** **class** CleaningExample **implements** AutoCloseable {  // 最好是库中共享的cleaner  **private** **static** **final** Cleaner ***cleaner*** = Cleaner.*create*();  **static** **class** State **implements** Runnable { // 内部类  State() {  // 初始化清理动作所需的状态  System.***out***.println("【构造】在一个雷电交加的夜晚诞生了...");  }  **public** **void** run() { // 清理使用多线程  // 清除操作访问状态, 最多执行一次  System.***out***.println("【析构】人终有一死, 或重于泰山, 或重于另一座山...");  }  }  **private** **final** State state;  **private** **final** Cleaner.Cleanable cleanable;  **public** CleaningExample() {  **this**.state = **new** State();  **this**.cleanable = ***cleaner***.register(**this**, state);  }  **public** **void** close() {  cleanable.clean(); // 启动多线程  }  }  **try** (CleaningExample clean = **new** CleaningExample()) {  // ...  } **catch** (Exception e) {} |

新一代的清除回收更多考虑的是多线程的使用，为了防止有可能造成的延迟，许多对象回收前的处理都是单独通过一个线程完成，提供性能。

* 23.6 对象克隆

Object类的clone()方法

|  |
| --- |
| protected *Object* clone() throws *CloneNotSupportedException* |

所有的类都继承Object，所有的类一定有clone()方法，但不是所有类都希望被克隆。如果要实现对象克隆，对象的类需要实现Cloneable接口，否则抛出异常。此接口没有一个方法，其主要描述的是一种能力。

|  |
| --- |
| **class** Person **implements** Cloneable {  **private** String name;  **private** **int** age;  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  @Override  **public** String toString() {  **return** "name=" + name + ", age=" + age;  }  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  **return** **super**.clone();  }  }  Person p1 = **new** Person("hikari", 25);  **try** {  Person p2 = (Person) p1.clone();  // 克隆后的对象内容一样, 内存地址不一样  System.***out***.println(p1.toString().equals(p2.toString())); // true  System.***out***.println(Integer.*toHexString*(p1.hashCode())); // 299a06ac  System.***out***.println(Integer.*toHexString*(p2.hashCode())); // 383534aa  } **catch** (CloneNotSupportedException e) {} |

实际开发很少需要对象的克隆。

* 23.7 Math类

Math类构造方法私有化，所有方法都是static，静态导入比较合适。

|  |
| --- |
| System.***out***.println(Math.***PI***); // 3.141592653589793  System.***out***.println(Math.*pow*(2, 3)); // 8.0  System.***out***.println(Math.*round*(-1.5)); // -1, +0.5后向下取整 |

示例：自定义四舍五入：

|  |
| --- |
| **class** MyMath {  **private** MyMath() {}  **public** **static** **double** round(**double** n, **int** scale) {  **double** x = Math.*pow*(10, scale);  **return** Math.*round*(n \* x) / x;  }  }  // -1.235\*100结果是-123.50000000000001, 四舍五入为-124...  System.***out***.println(MyMath.*round*(-1.235, 2)); // -1.24 |

* 23.8 Random类 (java.util包)

一个重要方法是获取不超过某个值的随机非负整数：

|  |
| --- |
| public *int* nextInt(*int* bound) // 产生[0, bound)的随机整数 |

示例：模拟36选7

|  |
| --- |
| **private** **static** ArrayList<Integer> select(**int** num, **int** total) {  ArrayList<Integer> arr = **new** ArrayList<>();  Random r = **new** Random();  **for** (**int** i = 0; i < num;) {  **int** n = r.nextInt(total);  **if** (!arr.contains(n) && n != 0) { // 不重复,不包含0  arr.add(n);  i++;  }  }  Collections.*sort*(arr); // 排序  **return** arr;  }  System.***out***.println(*select*(7, 36));// [9, 13, 14, 23, 28, 32, 33] |

**20180610**

* 23.9 BigInteger和BigDecimal

实现海量数字的基础计算，比如超过double范围的数字计算。最早只能使用String类保存，但是计算需要逐位拆分，十分麻烦。java.math包的BigInteger和BigDecimal是大数字类，是抽象类Number的子类。

示例：BigInteger的运算

|  |
| --- |
| String sa = "9876543210";  String sb = "1234567890";  // 构造大数字的字符串形式  **for** (**int** i = 0; i < 6; i++) {  sa += sa;  sb += sb;  }  System.***out***.println("len(sa)=" + sa.length() + ", len(sb)=" + sb.length());  BigInteger a = **new** BigInteger(sa);  BigInteger b = **new** BigInteger(sb);  // 四则运算  System.***out***.println("a+b=" + a.add(b));  System.***out***.println("a-b=" + a.subtract(b));  System.***out***.println("a\*b=" + a.multiply(b));  System.***out***.println("a/b=" + a.divide(b));  // 求商和余数, 结果是两个BigInteger的数组  BigInteger[] ret = a.divideAndRemainder(b);  System.***out***.println("a/b的商: " + ret[0] + ", 余数: " + ret[1]);  // 计算指数会非常慢  System.***out***.println(a.pow(Integer.***MAX\_VALUE***)); |

如果数据没有超过基本数据类型，不要使用大数字类型，计算性能太差。

BigDecimal使用和BigInteger类似，但其除法有个数据进位的问题。

|  |
| --- |
| public *BigDecimal* divide(*BigDecimal* divisor, *int* scale, *RoundingMode* roundingMode) |

示例：使用BigDecimal四舍五入：

|  |
| --- |
| **private** **static** **double** round(**double** n, **int** scale) {  // n转为BigDecimal除以BigDecimal的1, 四舍五入, 转为double  **return** **new** BigDecimal(n).divide(BigDecimal.***ONE***, scale, RoundingMode.***HALF\_UP***).doubleValue();  }  System.***out***.println(*round*(-1.235, 2)); // -1.24 |

* 23.10 Date类

简单Java类的设计来源于数据表的结构，日期是数据表常用的类型。

Java提供java.util.Date类处理日期。

|  |
| --- |
| Date date = **new** Date();  System.***out***.println(date); // Sun Jun 10 22:44:35 JST 2018 |

其构造方法：

|  |
| --- |
| public Date() { // 无参构造默认使用当前毫秒  this(System.currentTimeMillis());  }  public Date(*long* date) {  fastTime = date;  } |

Date类只是对long的一种包装，Date类提供日期与long之间的转换。

|  |
| --- |
| public Date(*long* date) // long毫秒值转为Date对象  public *long* getTime() // Date对象转为UTC时间戳(long类型的毫秒值) |

示例：

|  |
| --- |
| **long** now = **new** Date().getTime();  **long** afterTenDays = now + 8 \* 24 \* 60 \* 60 \* 1000;  System.***out***.println(**new** Date(afterTenDays)); // Mon Jun 18 22:49:27 JST 2018 |

* 23.11 SimpleDateFormat类

Date默认打印的结构并不十分友好，需要对日期格式化处理。

java.text包的SimpleDateFormat类用于以本地敏感的方式格式化和解析日期。

SimpleDateFormat继承于抽象类DateFormat，后者继承于抽象类Format。

常用方法：

|  |
| --- |
| public SimpleDateFormat(*String* pattern) // 构造方法, 定义日期格式  public final *String* format(*Date* date) // DateFormat提供, Date对象转为字符串  public *Date* parse(*String* source) throws ParseException // DateFormat提供, 字符串转为Date对象 |

示例：

|  |
| --- |
| SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  // 1. Date对象转为字符串  Date date = **new** Date();  System.***out***.println(sdf.format(date)); // 2018-06-10 23:12:58  // 2. 字符串转为Date对象  String someday = "2020-02-02 20:02:20";  **try** {  Date day = sdf.parse(someday);  System.***out***.println(day); // Sun Feb 02 20:02:20 JST 2020  } **catch** (ParseException e) {}  // 附赠: 数字格式化  NumberFormat nf = NumberFormat.*getInstance*();  **long** n = 1234567890;  System.***out***.println(nf.format(n)); // 1,234,567,890 |

如果字符串使用的日期超过了合理范围，则会自动进位。

**24 正则表达式**

* 24.1 正则表达式简介

String是非常万能的类型，因为支持向各种数据类型转换。开发中用户输入的信息基本都是String。String向其他数据类型转换时，可能需要进行复杂的验证。

比如验证字符串是否全是数字组成，之前的练习将字符串转为字符数组，逐个比较，代码略麻烦。使用正则表达式验证则可简化代码。

|  |
| --- |
| String s = "123";  **if** (s.matches("\\d+")) {  **int** n = Integer.*parseInt*(s);  System.***out***.println(n + "\*" + n + "=" + n \* n); // 123\*123=15129  } |

正则表达式最早从Perl语言里发展而来，JDK 1.4后支持正则表达式，提供java.util.regex包，同时String也支持正则表达式。

正则表达式主要特点是方便验证处理，复杂字符串修改或获取指定数据。

* 24.2 常用正则表达式结构

① 字符类

|  |
| --- |
| [abc] a, b, or c  [^abc] 除了a, b, c的任何字符(否定)  [a-zA-Z] a 到z或A到Z(范围) |

② 预定义的字符类

|  |
| --- |
| . 任意字符 (行结束符可能(不)匹配)  \d 数字: [0-9]  \D 非数字: [^0-9]  \s 空白字符: [ \t\n\x0B\f\r]  \S 非空白字符: [^\s]  \w 单词字符: [a-zA-Z\_0-9]  \W 非单词字符: [^\w] |

③ 匹配边界

|  |
| --- |
| ^ 一行开始  $ 一行结束  \b 单词边界  \B 非单词边界 |

④ 数量词(贪婪模式)

|  |
| --- |
| X? X, 一次或零次  X\* X, 零次或多次  X+ X, 一次或多次  X{n} X, 恰好n次  X{n,} X, 至少n次  X{n,m} X, 至少n次，至多m次 |

数量词后面加?表示非贪婪模式

⑤ 逻辑运算符

|  |
| --- |
| XY XY紧随  X|Y X或Y  (X) X, 作为一个捕获组 |

* 24.3 String对正则表达式的支持

正则表达式有单独的包java.util.regex包，有两个主要的类Matcher和Pattern，如果不是必要，此包很少用，因为String也支持正则表达式，可以直接使用。

String类关于正则表达式的常用方法：

|  |
| --- |
| public *boolean* matches(*String* regex)  public *String* replaceFirst(*String* regex, *String* replacement)  public *String* replaceAll(*String* regex, *String* replacement)  public *String*[] split(*String* regex)  public *String*[] split(*String* regex, *int* limit) |

练习：验证email格式

用户名由字母、数字、\_组成，不能\_开头；域名由字母、数字组成；域名后缀必须是：.cn、.com、.net、.com.cn、.org

|  |
| --- |
| String email = "hikari@qq.com";  String re = "[a-zA-Z0-9]\\w+@[a-zA-Z0-9]+\\.(cn|com|net|com\\.cn|org)";  System.***out***.println(email.matches(re) ? "匹配成功!" : "匹配失败!"); // 匹配成功! |

* 24.4 java.util.regex包

有两个主要的类Matcher和Pattern。

① Pattern

构造方法私有，提供正则表达式编译处理。

public static *Pattern* compile(*String* regex)

提供split()方法，功能与字符串的split()一样

public *String*[] split(*CharSequence* input)

② Matcher

实现正则表达式匹配，实例化依靠Pattern类的matcher()方法

public *Matcher* matcher(*CharSequence* input)

Matcher对象主要方法：

|  |
| --- |
| public *boolean* matches()  public *String* replaceFirst(*String* replacement)  public *String* replaceAll(*String* replacement) |

如果只是split、replace、matches根本不需要使用java.util.regex包，直接使用String的方法即可。

Matcher类提供分组功能，是String类不具备的。

|  |
| --- |
| public *boolean* find() // 试图找到匹配的下一个子序列  public *String* group() // 返回前一个匹配的子序列  public *String* group(*int* group) // 返回上次匹配操作中被给定组捕获的子序列 |

示例：

|  |
| --- |
| String html = "<ul><li>rin</li><li>maki</li><li>nozomi</li></ul>";  String re = "<li>([a-z]+)</li>";  Pattern p = Pattern.*compile*(re);  Matcher m = p.matcher(html);  **while** (m.find()) { // 是否有匹配内容  System.***out***.println(m.group(1));  } |