梳 理

2013-3-24 森语星园

1. 深入理解JVM虚拟机
   1. JVM内存结构

（1）方法区：存储编译之后的源文件是

（2）栈区：保存基本类型和对象的引用

* 基本类型：字节类型byte、字符类型：char、短整形shot、整形int、长整形long、浮点型float、双精度类型double、布尔型boolean
* 1char=2byte=16bit
* 类图关系：



说明：Byte、Short、Integer类等都是final类型，final类是不可以被继承的。Final方法是不可以被重载的。Final类所有方法都是final的。

（3）堆区：保存对象

* 堆区，按照年代划分，新生代、老年代和永久代。新生代分伊甸区和生存区，比例一般是8:1:1；
* 新生代一般采用复制算法，收集垃圾。老年代，采用标记整理算法，收集对象。基于效率上的考虑。
* 判断对象是否包含引用，1是计数法，2是根搜索法。
* 一次GC，是建立在新生代和老年代都已经用完的情况之下。

（4）计数器：保存线程执行到的代码位置

* 1. JVM编译方式

1. JVM采用二种编译方式：解释执行和编译执行

* 解释执行：源文件->目标文件；例如c，javascript；优点：快速，直接，省去编译中间码的过程。缺点：处理循环等上面有致命的缺点。
* 编译执行：源文件->中间码->目标文件；例如java；javac、eclipse的增量编译，是将源文件编译成中间码class文件；jvm（hotspot vm）在采用c1（client compile）和c2（server compile），将中间文件，编译成目标文件。C1是侧重效率（将热点代码迅速编译成本地代码），c2是侧重优化（相对优化会比较耗时）。
* 利用解释执行，达到快速启动应用的目的，利用编译执行，实现运行时的编译优化。

1. 编译过程：源文件-词法分析-语法分析-中间码-目标码；
2. 文件加载：

* 启动类加载-扩展类加载-应用类加载-自定义类加载
* 加载-连接（准备、验证、解析）-初始化-使用-卸载
* 例如：jdbc驱动加载

|  |
| --- |
| Class.*forName*("com.mysql.jdbc.Driver");  Connection con = DriverManager.*getConnection*("jdbc:mysql://127.0.0.1/test?useUnicode=true&characterEncoding=utf-8", "root", "root");  Statement state = con.createStatement();  … … |

补充：java.sql类图



* 1. JVM线程模型



1. Java并行编程实践

2.1、线程类图



* 构造线线程二种方式：

（1）XXX implements Runnable

（2）XXX extends Thread

* Join、yield、sleep、wait的区别：

1. Wait是object方法、其余是Thread中方法；wait会释放所占用的锁，其他线程可以执行方法，其余不会释放锁。
2. Join调用wait实现，等待当前线程执行结束，程序才往下继续执行。
3. Sleep暂停当前线程，yield也是，只不过后者不能指定暂停时间。

2.2、线程状态图



2.3、Concurrent并发库

（1）接口设计



（2）同步工具类

* 闭锁：相当于一扇门，在闭锁到达结束状态之前，这扇门是关闭的，没有任何线程可以通过；当到达结束状态时，这扇门打开，并允许所有线程通过。

例如：基于二元闭锁（包括二个状态，开启门和结束门）

|  |
| --- |
| **private** **void** timeTasks(**int** nThreads, **final** Runnable command) **throws** InterruptedException {    //开启门  **final** CountDownLatch startGate = **new** CountDownLatch(1);  //结束门  **final** CountDownLatch endGate = **new** CountDownLatch(nThreads);    **for** (**int** i=0;i<nThreads;i++ ) {  Thread t = **new** Thread() {  @Override  **public** **void** run() {  **try** {  // 等待所有线程初始化完毕  startGate.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **try**{  command.run();  } **finally** {  endGate.countDown();  }  }  };  t.start();  }  **long** start = System.*nanoTime*();  startGate.countDown();  // 等待所有线程执行结束  endGate.await();  **long** end = System.*nanoTime*();  System.*out*.println(start - end);  } |

* FutureTask：SwingWork的实现；定时资源池的实现；
* 信号量：用于控制同时访问某个特定资源的操作数量，或者用于执行某个操作的数量。可用于实现资源池；

|  |
| --- |
| **public** **class** SemaphoreTest<T> {  **private** **final** Set<T> set;  **private** **final** Semaphore sem;    **public** SemaphoreTest(**int** bond) {  **this**.set = Collections.*synchronizedSet*(**new** HashSet<T>());  **this**.sem = **new** Semaphore(10);  }    **public** **boolean** add(T o) **throws** InterruptedException {  // 先获取准入许可，否则一直阻塞  sem.acquire();  **boolean** wasAdd = **true**;  **try** {  wasAdd = set.add(o);  } **finally** {  // 加载失败的时候，也释放资源，提供在此添加的机会  **if** (wasAdd == **false**) sem.release();  }  **return** wasAdd;  }    **public** **boolean** remove(Object o) {  **boolean** wasRemoved = set.remove(o);  **if** (wasRemoved) {  // 删除成功，释放资源  sem.release();  }  **return** wasRemoved;  }  } |

* 栅栏：闭锁是一次性对象，一旦进入终止状态，就不能重复设置。类似于闭锁，其能阻止一组线程直到某个状态发生。

2.4、显式锁Lock

（1）API



说明：reentrantLock提供公平策略来构造锁，默认采用竞争方式。

（2）概念

* 锁分类1：内置锁和显式锁

内置锁：每一个java对象都可以作为实现同步的锁，这些锁就称为内置锁或监视器锁。使用synchronized关键字。

显式锁：Lock提供一种无条件、可轮询（tryLock）、定时以及可中断的锁操作机制。所有加锁解锁的动作，都是显式的。

注意：一定要在finally中释放lock，方式代码块出现异常抛出，锁未解除。显式锁并不是要替换内置锁，二是其辅助作用。Synchronized本身在编译时会优化，显式锁不一定会。所以效率上不会比synchronized要高。

* 锁分类2：独占锁（排他锁、互斥锁）和可重入锁

独占锁：只有一个线程可以拥有这个锁。

可重入锁：并发读、独占写。例如ReadWriteLock

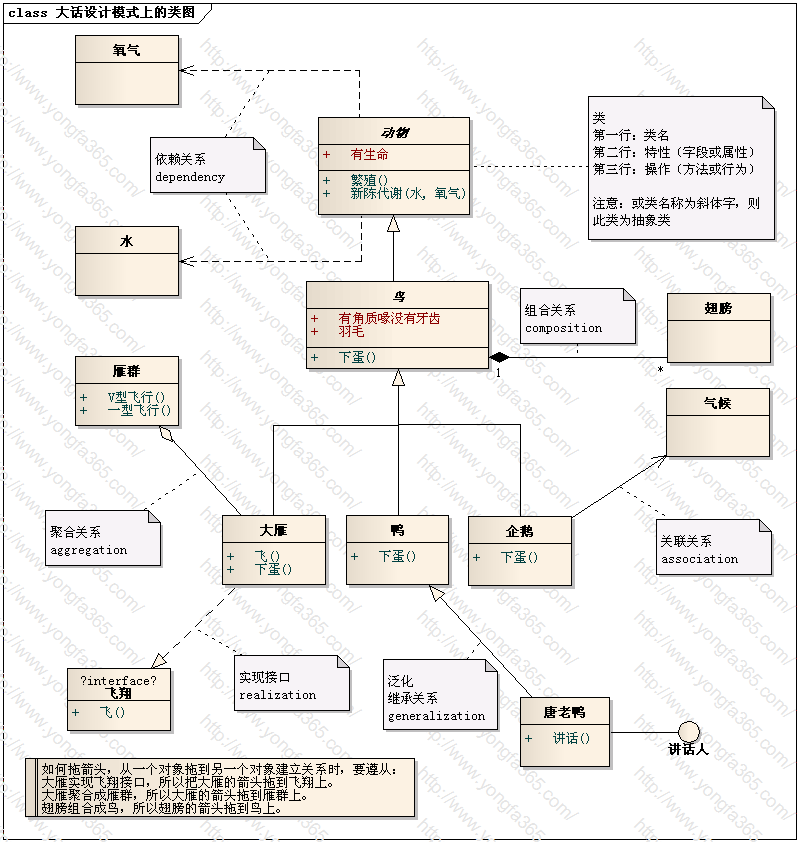
* 显式锁：无条件锁lock、可轮询锁tryLock、定时tryLock(timeout,unit)以及可中断锁lockInterruptibly；
* 减少锁竞争：第一缩小锁的范围（开放式调用，快进快出）；第二减少锁的粒度（锁分解和锁分段）。

（3）例子：

基于轮询锁，必须在finally中显式解锁。

|  |
| --- |
| **private** Lock lock1 = **new** ReentrantLock();  **private** Lock lock2 = **new** ReentrantLock();  **private** **void** method1() {  **try**{  lock1.tryLock();  System.*out*.println(Thread.*currentThread*().getName() + ":进入方法1");  Thread.*sleep*(1000);  System.*out*.println(Thread.*currentThread*().getName() + ":准备进入方法2...");  method2();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  lock1.unlock();  }  }    **private** **void** method2() {  **try**{  lock2.tryLock();  System.*out*.println(Thread.*currentThread*().getName() + ":进入方法2");  Thread.*sleep*(1000);  System.*out*.println(Thread.*currentThread*().getName() + ":准备进入方法1...");  method1();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  lock2.unlock();  }  } |

（4）补充：接口、类之间关系：



**明确依赖和关联（组合和聚合）的区别**

2.5、广义内置条件队列Condition

（1）API



（2）Demo

* 基于Condition实现自定义阻塞式队列：

|  |
| --- |
| **private** **final** **int** MAX\_SIZE = 10;    **private** **final** List<String> vector = **new** ArrayList<String>();    **private** Lock lock = **new** ReentrantLock();    **private** Condition notEmptySignal = lock.newCondition();    **private** Condition notFullSignal = lock.newCondition();    **public** **void** put(String value) {  **try** {  lock.lock();    **if** (vector.size() == MAX\_SIZE) {  notFullSignal.await();  }    vector.add(value);    notEmptySignal.signalAll();  } **catch** (InterruptedException e) {  e.printStackTrace();    **throw** **new** RuntimeException("写入发生异常");  } **finally** {  lock.unlock();  }  }    **public** String get() {  **try** {  lock.lock();    **if** (vector.size() == 0) {  notEmptySignal.await();  }    notFullSignal.signalAll();    **return** vector.remove(0);  } **catch** (InterruptedException e) {  e.printStackTrace();    **throw** **new** RuntimeException("读取发生异常");  } **finally** {  lock.unlock();  }  } |

* 异步请求转化为同步请求：

|  |
| --- |
| **public** CommonMsg syncRequestRemoteService(String fid, CommonMsg msg, **int** timeout) {  **final** Lock syncRequestLock = **new** ReentrantLock();  **try** {  **final** Condition condition = syncRequestLock.newCondition();  **final** CommonMsg[] tempCommonMsg = **new** CommonMsg[1];  syncRequestLock.lock();  requestRemoteService(fid, msg, timeout, **new** CommonMsgCallback() {  **public** **void** onMessage(CommonMsg msg) {  **try** {  syncRequestLock.lock();  tempCommonMsg[0] = msg;  condition.signalAll();  } **finally** {  syncRequestLock.unlock();  }  }  });  condition.awaitUninterruptibly();  **if** (tempCommonMsg[0] == **null**) {//tempCommonMsg[0]不应该等于null  CommonMsg errorMsg = **new** CommonMsg(CommonMsg.*Receive*);  errorMsg.setFID(fid);  errorMsg.setErrorMsg("内部错误");  **return** errorMsg;  } **else** {  **return** tempCommonMsg[0];  }  } **finally** {  syncRequestLock.unlock();  }  } |

2.6、补充：获取线程结果的方式

第一：线程体注入接口，采用回调接口的方式

第二：异步转同步，如使用join，强制等待当前线程执行结束，程序才继续往下执行。

1. Thinking In Java（Collection）

3.1、API



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **List** | **Set** | **Map** | **Queue** |
| 线性 | 数组：**ArrayList**  链表：LinkedList  CopyOnWriteArrayList | 链表：LinkedHashSet  CopyOnWriteArraySet |  | 队列:  ArrayBlockingQueue  LinkedBlockingQueue |
| 非线性 |  | 哈希：**HashSet**  树：TreeSet | **HashMap**  TreeMap  ConcurrentHashMap |  |
| 关于集合中的并、交、差集：在ADT中定义了访问接口，  差集：boolean removeAll(Collection)  交集：boolean retainAll(Collection)  并集：boolean addAll(Collection)  包含重复的并集：先差在并，例如list1. removeAll(list2);list1.addAll(list2)  上述操作都是值传递！ | | | | |

3.2、说明

（1）Collections

同步接口：SynchronizedCollection、SynchronizedMap、SynchronizedList、SynchronizedSet

排序接口：sort(List)、sort(List,Comparator)

其他：反转接口（reverse(list)）、替换fill(List,object)、最大值、最小值、交换等

（2）常用接口

排序接口：java.lang.Comparable compareTo(T)、java.util.Comparator compare(T,T)

迭代接口：java.lang.Iterable 、java.util.Iterator、 ListIterator

克隆接口：java.lang.Cloneable 标记性接口（调用对象clone方法时，对象必须实现此接口，否则调用时，抛出异常）

可序列化接口：java.io. Serializable 标记性接口

1. 编写高质量java代码的151建议

4.1、java语法

（1）java开发中的统通用准则

* + 防止自增的陷阱

|  |
| --- |
| **int** count = 0;  **for** (**int** index =0;index < 10;index++) {  count = count ++;  }  System.*out*.println(count); // 结果是0 |

（2）基本类型

* + 基本类型

字节类型byte、字符类型char、短整型shot、整形int、长整形long、单精度float、双精度double、布尔boolean

* + 用整形类型处理货币，或是BigDecimal类型

补充：

十进制转二进制：整数部分**除2倒取余**，小数部分**乘2取整**

例如：10.2 = (1010.00110)B

10/2=5 ….0

5/2=2….1

2/2=1….0

1/2=0….1

0.2\*2=0.4…0

0.4\*2=0.8…0

0.8\*2=1.6…1

0.6\*2=1.2…1

0.2\*2=0.4…0

…

所以：10-9.6=0.4000000036（0.4在二进制中是无线循环小数）

解决方法：第一是使用BigDecimal；第二是扩大变整数在缩小。

（3）类、对象和方法

* **覆写equal方法时，一定要覆写hashcode**

基于hash码的数据结构（hashmap、hashset、hashtable）：先通过hashcode找到元素的存储地址，取出值，在调用equal方法判断是否相等。

在没有重写hashcode的方法下，每个对象的哈希码都不相同。

其他结构：直接调用equal方法。

Object.equal方法基于地址。默认地址也是基于hashcode。

|  |
| --- |
| Map<Person, String> map = **new** HashMap<Person, String>();  map.put(**new** Person("测试"), "测试");  List<Person> list = **new** ArrayList<Person>();  list.add(**new** Person("测试"));    /\*\*  \* false:  \* 因为先通过对象hashcode，找到对象存放地址；在取出地址下的值  \* 因为没有覆写haosecode，所以二个对象生成不同的hashcode  \*/  System.*out*.println(map.containsKey(**new** Person("测试")));  // true  System.*out*.println(list.contains(**new** Person("测试")));  ------------------------------------------------------------  **class** Person{  String name = "";  **public** Person(String name) {  **this**.name = name;  }    @Override  **public** **boolean** equals(Object obj) {  **if** (obj **instanceof** Person){  **return** ((Person)obj).name.equals(name);  }  **return** **false**;  }  } |

* 建议使用getClass替换instanceof

4.2、JDK介绍

（4）字符串

* String的不可变性

第一：String是final的，**不可能产生子类**。

第二：String类中的方法，如果返回字符串，就会**产生一个新的String对象**，不会修改源对象；

|  |  |  |
| --- | --- | --- |
| **String** | **StringBuffer** | **StringBuilder** |
| 不变 | 可变 | 可变 |
|  | 线程安全 | 线程不安全 |

（5）数组和集合

* 不同列表选择不同遍历方法

|  |  |
| --- | --- |
| **数组类型（表）数据结构** | **链表类型（表）数据结构** |
| 采用下标遍历（说明前后关系没有依赖，可以随机存取）eg：ArrayList | 采用foreach方式（说明前后元素有关系，相互依赖）eg：LinkedList |

* subList产生的列表，只是原列表的一个试图，所有的修改动作，直接作用于原列表。
* 推荐使用subList处理局部列表。例如，删除集合下标XX开始，XX结束的元素。
* 集合运算更加优雅的方式

|  |  |  |  |
| --- | --- | --- | --- |
| **交集** | **差集** | **并集** | **不重复并集** |
| retainAll | removeAll | addAll | 先差，在并  list1.removeAll(list2).addAll(list2); |

内部实现：遍历集合，判断是否相等

（6）枚举和注解

* 推荐使用枚举定义常量

（7）泛型和反射

（8）异常

（9）多线程和并发

4.3、框架、性能和调优

（10）性能和效率

（11）开源世界

（12）思想为源

算法分析与数据结构（Java）

PowerDesigner15.1

附录：情境领导：保罗.赫塞

1. **教练模式：引导和指示；引导模式：解释和劝导；参与模式：激励和解决问题；授权模式：考核和监督**
2. 教练模式：新员工、引导模式：抱怨者、参与模式：老员工、授权模式：明星员工；

新员工：低能力、低意愿R1；教练模式：高工作、低关系S1

抱怨者：低能力、高意愿R2；引导模式：高工作、高关系S2

老员工：高能力、低意愿R3；参与模式：低工作、高关系S3

明星员工：高能力、高意愿R4；授权模式：低工作、低关系S4

【R：准备度模型、S：领导模型】

1. 三大心法：

领导是尝试影响他人的一种行为，而有效的领导是针对被领导者或团队绩效的需求，然后适当的调整自己的行为。

* 1. 第一、工作不成功在于领导（此领导非直接上司，而是指的自己）；
  2. 第二、达到互利双赢
  3. 第三、好领导，可以实现员工自治

1. 准备度包括：包括能力和意愿

能力包括：信心：能做、承诺：将会做、动机：想做

意愿包括：知识：知道如何做、经验：曾经做过、技能：正在执行

1. 自我介绍：【明确任务、工作分解、降低难度】

评价一个人，不可以简单的贴标签，应该立体和全面。

**第一、职位、角色或职能**

**第二、工作、目标或目的**

**第三、活动、行为或环节**

例1：

核心开发工程师

设计、编码和测试

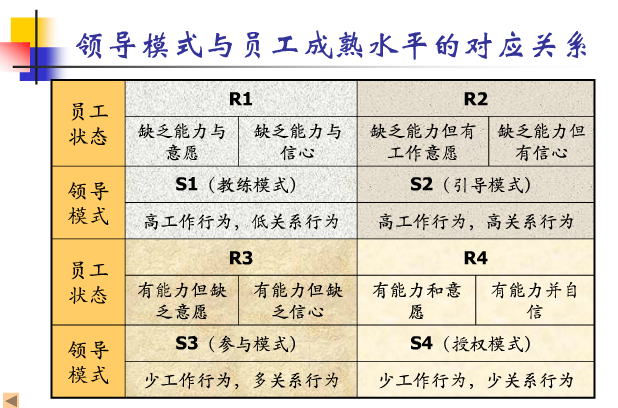
编写概要设计文档、原始界面设计、数据表设计；编写代码，例如ssh；单元测试、功能测试等；

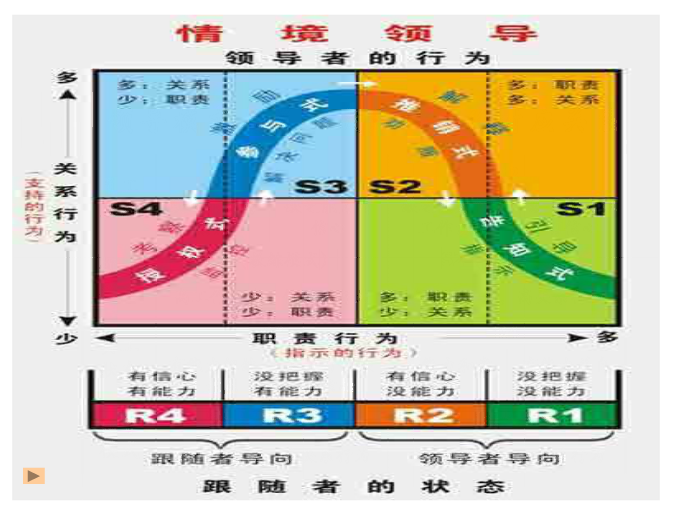
例2：

秘书

服务领导、会议安排、培训

预算、审批和考核(会议安排细化)





说明：

工作行为：讲方法、方向是单向的、工作行为需要监督

关系行为：讲感情、方向是多向的、适当参与提供反馈

授课老师（中国电信讲师）

杨杨：13301160103；QQ:28144153