目录

[Java执行命令 2](#_Toc4982)

[配置环境变量 2](#_Toc22270)

[Java编译及运行过程 3](#_Toc29340)

[JDK, JRE, JVM关系 3](#_Toc25742)

[int和Integer 3](#_Toc6663)

[静态变量、实例变量 3](#_Toc2716)

[静态方法不能发出对非静态方法的调用 3](#_Toc11535)

[对象、类 3](#_Toc13205)

[面向过程、面向对象 4](#_Toc14096)

[面向对象四大特征 4](#_Toc1379)

[静态绑定、动态绑定 4](#_Toc30287)

[JVM加载class文件的原理机制 4](#_Toc7503)

[主要类加载器 5](#_Toc21176)

[创建对象的方法 5](#_Toc11313)

[堆、栈 5](#_Toc2011)

[垃圾回收(Garbage Collection) 5](#_Toc32012)

[内存泄漏、内存溢出 6](#_Toc923)

[Overload和Override的区别 6](#_Toc380)

[接口、抽象类 6](#_Toc18269)

[内部类 6](#_Toc6819)

[常用包 6](#_Toc18370)

[Object的方法 7](#_Toc19069)

[Equals()和HashCode() 7](#_Toc26423)

["=="和"equals" 7](#_Toc6330)

[String常用方法 7](#_Toc18898)

[String, int和Integer之间的转换 7](#_Toc3988)

[String和StringBuffer 8](#_Toc30046)

[常量池 8](#_Toc6209)

[Queue, Deque和Stack 8](#_Toc1667)

[List, Set, Map三个接口,存取元素时,各有什么特点 9](#_Toc31720)

[Vector, ArrayList, LinkedList的区别 9](#_Toc19563)

[HashMap, Hashtable 9](#_Toc13632)

[Comparable和Comparator 9](#_Toc26196)

[IO 9](#_Toc24654)

[BIO, NIO, AIO 10](#_Toc6895)

[序列化和反序列化 10](#_Toc16595)

[Error和Exception 10](#_Toc8062)

[常见的运行时异常 10](#_Toc23916)

[常见CheckedException 11](#_Toc29959)

[throw和throws 11](#_Toc23666)

[有return的情况下try， catch， finally的执行顺序 11](#_Toc29734)

[final， finally， finalize的区别  11](#_Toc31006)

[抛出异常的好处 11](#_Toc9265)

[进程和线程 11](#_Toc23158)

[创建线程的方式 11](#_Toc8406)

[线程的生命周期 12](#_Toc8984)

[同步、异步、并行、并发 12](#_Toc10590)

[线程安全、线程同步 12](#_Toc2009)

[实现线程安全的方式 12](#_Toc3886)

[ReadWriteLock 12](#_Toc27853)

[线程不安全、线程安全归类 12](#_Toc9545)

[锁对象 13](#_Toc17214)

[sleep()和wait()的区别 13](#_Toc8407)

[锁池、等待池 13](#_Toc30864)

[线程池 13](#_Toc6918)

[死锁 13](#_Toc1329)

**正则表达式**

|  |  |  |  |
| --- | --- | --- | --- |
| 正则表达式 | 说明 | 正则表达式 | 说明 |
| [abc] | a，b，c中任意一个 | . | 任意一个 |
| [^abc] | 除了a，b，c | \d | 任意一个数字，即[0-9] |
| [a-zA-Z0-9] | a~z，A~Z，0~9中任意一个 | \w | 单词字符，即[a-zA-Z0-9\_] |
| [a-z&&[^bc]] | a~z中除了b和c以外的任意一个 | \s | 空白字符，即[\t\n\xOB\f\r] |
| X? | 0个或1个X，即X{0，1} | \D | 非数字字符 |
| X\* | ≥0个X | \W | 非单词字符 |
| X+ | ≥1个X | \S | 非空白字符 |
| X{n} | n个X | ^ | 字符串开始 |
| X{n，} | ≥n个X | $ | 字符串结束 |
| X{n，m} | m≥X≥n | \b | 单词边界（单词和符号的边界） |
| (…|…|…) | 分组 | \B | 非单词边界 |
| 1.要匹配的字符串 \ 2.正则 \\ 3.java表示的正则 \\\\ | | | |
| $ () \* + . [] ? \ ^ {} , | 特殊字符 | | | |
| X?、X\*、X+、X{n，}都是贪婪型； X??、X\*?、X+?、X{n，}?都是勉强型； X?+、X\*+、X++、X{n，}+都是占有型 ??? | | | |

**日期时间转换**

|  |  |
| --- | --- |
| Date → long | long time = date.getTime(); |
| long → Date | date.setTime(time); |
| Date → String | String str = sdf.format(date); |
| String → Date | date = sdf.parse(str); |
| Date → Calendar | calendar.setTime(date); |
| Calendar → Date | date = calendar.getTime(); |

# Queue, Deque和Stack

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 尾进 | 头进 | 尾出 | 头出 | 获取 |
| Queue | offer |  |  | poll | peek |
| Deque  Stack | offer, offerLast | offerFirst, push | pollLast | poll, pollFirst, pop | peek,peekfirst/Last |
| List | add, addAll | | remove, removeAll, clear | | get, set, subList |
| Map | put, putAll | | remove, clear | | get |

# List, Set, Map三个接口,存取元素时,各有什么特点

|  |  |  |
| --- | --- | --- |
| List | Set | Map |
| 实现Collection接口,是单列元素的集合 | | 双列集合 |
| 允许重复 | 不允许有重复的元素(equals)  重复元素会覆盖掉 | 不允许存储重复的Key(equals)  重复的key会覆盖，返回替换掉的value |
| 按放入顺序排序 |  |  |
| Vector, ArrayList, LinkedList | HashSet, LinkedHashSet | HashMap, Hashtable, LinkedHashMap |
| 可以用for等 | 只能以Iterator取得所有元素，再逐一遍历； |  |

# Vector, ArrayList, LinkedList的区别

|  |  |  |
| --- | --- | --- |
| Vector | ArrayList | LinkedList |
| 以数组的形式存储，适合查找 | | 以双向链表的形式存储，适合插入、删除 |
| 线程同步 | 线程不同步，可以通过Collections.synchronizedList()实现线程同步 | |
| 元素填满容器时会自动扩容100% | 元素填满容器时会自动扩容50% | 实现了Deque接口,提供了更多的方法,包括offer(),peek(),poll()等,多与一些线程池一起使用 |

# HashMap, Hashtable

|  |  |
| --- | --- |
| HashMap | Hashtable |
| 都实现了Map接口 | |
| 非线程安全  加锁或使用Collections.synchronizedMap() | 线程安全 |
| key, value均可以空 | key, value不允许为空 |
| 直接继承的父类是AbstractMap | 直接继承的父类是Dictionary |
| containsValue(Object value)  containsKey(Object key) | contains(Object value) |
| 重新计算hash值 | 直接使用对象的hashCode |
|  |  |
| ConcurrentHashMap是HashMap线程安全的实现，并且逐渐取代Hashtable的使用，因为Hashtable锁的机制是对整个象加锁，而ConcurrentHashMap使用的是局部锁技术，实际上就是把Map分成了分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中，而这里的每个segment都相当于一个小的Hashtable，性能将高于Hashtable | |

# Comparable和Comparator

|  |  |
| --- | --- |
| *Comparable* | *Comparator* |
| 在类的设计中使用 | 用于类设计已经完成后 |
| compareTo(Object o) | compare(To1, To2) |

# IO

|  |  |  |
| --- | --- | --- |
| 字节流 | OutputStream / InputStream | 字节流的父类 |
| FileOutputStream / FileInputStream | 文件流 FOS(file, true):写到文件末尾 |
| BufferedOutputStream  BufferedInputStream | 缓冲流 BOS/BIS(os/is, sz):指定缓冲大小 |
| ObjectOutputStream / ObjectInputStream | 对象流 |
| 字符流 | Writer / Reader | 字符流的父类 |
| OutputStreamWriter  InputStreamReader | 转换流 ???  OSW/ISR(os, cs/enc):指定字符集/字符集编码器 |
| FileWriter / FileReader | 字符文件流 FW(file, true):写到文件末尾 |
| PrintWriter / BufferedReader | 打印写 PW(os, true):自动刷新输出缓冲区  PW(os, cs): 指定字符集 pw.println  字符缓冲流 BR(read, sz):指定缓冲大小 br.readLine |
| RandomAccessFile | | write, writeInt, read, seek, skipBytes |

# BIO, NIO, AIO

|  |  |  |
| --- | --- | --- |
| BIO：同步并阻塞 | NIO：同步非阻塞 | AIO(NIO.2)： 异步非阻塞 |
| 一个连接一个线程，如果这个连接不做任何事情会造成不必要的线程开销，可以通过线程池机制改善 | 一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理 | 一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理 |
| 适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解 | 适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持 | 适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持 |

# 序列化和反序列化

|  |  |  |
| --- | --- | --- |
| 定义 | 把对象转换为字节序列的过程称为对象的序列化 | 把字节序列恢复为对象的过程称为对象的反序列化。 |
| 作用 | 比如远程接口调用，对象网络传输；深度克隆时，复杂对象的的复制；缓存对象的硬盘存储；Session的硬盘序列化等 | 就是把序列化的对象重新加载到内存中使用 |
| 序列化的实现 | 首先必须实现Serializable或者Externalizable，而Serializable是超级接口，Externalizable是其实现。Serializable接口中什么也没有定义，它只是对对象生成一个唯一的ID标识，告诉JVM该对象是可以序列化的，如果不实现序列化接口，在序列化的过程中会抛出异常 | |
| JavaAPI提供的序列化方式 | ObjectOutputStream对象输出流，它的writeObject (Object obj)方法可对参数指定的obj对象进行序列化，把得到的字节序列写到一个目标输出流中。ObjectInputStream对象输入流，它的readObject()方法从一个源输入流中读取字节序列，再把它们反序列化为一个对象，并将其返回 | |

# Error和Exception

|  |  |
| --- | --- |
| Error | Exception |
| 一般指与虚拟机相关的问题。对于这类错误，Java编译器不去检查它们。对于这类错误导致的应用程序中断，仅靠程序本身无法恢复和预防，遇到这样的错误，建议让程序终止。 | 表示程序可以处理的异常，可以捕获且可以恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。 |
| InternalError: JVM中出现一些意外的内部错误 | CheckedException:  JVM强制让我们处理的异常，用try{}catch(){}捕捉 |
| OutOfMemoryError: 内存溢出错误 |
| NoClassDefFoundError: JVM或ClassLoader…（隐式） | RuntimeException:  JVM不强制我们去捕捉，异常一旦发生，会由JVM接管，一直往上抛，直到遇到异常处理的代码… |
| StackOverflowError: 程序递归太深而发生堆栈溢出 |
| IOError: 严重的 I/O 错误 |

# 常见的运行时异常

|  |  |
| --- | --- |
| ArithmeticException | 算术运算异常：2/0 |
| NullPointerException | 空指针异常 |
| ClassCastException | 类型转换异常（将对象强制转换为不是实例的子类） |
| NumberFormatException | 数字格式异常：Integer.parseInt("abc") |
| IndexOutOfBoundException | 下标越界异常 |
| NegativeArraySizeException | 数组大小为负异常（创建大小为负数的数组） |
| UnsupportedOperationException | 不支持的操作异常：使用Arrays.asList()后调用add，remove等 |
| IllegalArgumentException | 传递非法参数异常（???） |
| ArrayStoreException | 数组存储异常（向数组中存放与声明类型不兼容对象） |

# 常见CheckedException

|  |  |
| --- | --- |
| ClassNotFoundException | 类文件找不到（显式）Class.forName(…), loadClass(…) |
| NoSuchMethod/NoFieldException | 无法找到某一特定方法/字段 |
| CloneNotSupportedException | 在没有实现Cloneable接口的实例上调用Object的clone() ??? |
| IOException | 发生某种I/O异常 |

# throw和throws

|  |  |
| --- | --- |
| throw | throws |
| 语句真正的抛出一个异常，语法：throw e | 该方法可能要抛出异常，调用者必须捕获或继续抛出 |
| 和try-catch或throws配合使用 | 可单独使用 |

# [有return的情况下try， catch， finally的执行顺序](http://blog.csdn.net/kavensu/article/details/8067850)

|  |
| --- |
| 当try和catch中有return时，finally仍然会执行 |
| finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，不管finally中的代码怎么样，返回的值都不会改变，仍然是之前保存的值），所以函数返回值是在finally执行前确定的 |
| finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值 |

# final， finally， finalize的区别

|  |  |
| --- | --- |
| final | 属性不可变，方法不可覆盖，类不可继承 |
| finally | 与try…catch结合使用，finally语句块中的语句一定会被执行(比return和break语句后执行)，通常用来进行资源的关闭操作，如数据连接关闭等 |
| finalize | Object类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等 |

# 抛出异常的好处

|  |
| --- |
| 让你把精力集中在那关键的代码上，不用加各种错误判断，所有的错误在后面统一处理。这样处理实现了错误代码和关键代码的基本分离，增强代码可读性，也方便维护。 |

# 进程和线程

|  |  |
| --- | --- |
| 进程 | 具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统分配资源的最小单元 |
| 线程 | 是进程的一个实体，是操作系统调度的最小单元，有自己程序计数器、一组寄存器和栈 ??? |
| 区别 | 1. 一个程序至少有一个进程，一个进程至少有一个线程 2. 线程的划分尺度小于进程，使得多线程程序的并发性高 3. 多个线程共享进程的内存，从而极大提高了程序的运行效率 |
| 原子操作 | 不会被线程调度机制打断的操作，这种操作一旦开始，就一直运行到结束，中间不会有任何context switch切换到另一个线程(如：对除long和double型的基本类型进行赋值(或返回)) |

# 创建线程的方式

|  |  |
| --- | --- |
| 继承Thread | new Thread(){public void run()…}.start() |
| 实现Runnable | new Thread(new Runnable(){public void run()…}).start() |
| 实现Callable | new Thread(new FutureTask<>(new Callable<>(){public object call()…return…})).start() |
| 线程池 | ExecutorService pool = Executors.newFixedThreadPool(3);  for (int i = 0; i < 10; i++) {  pool.exe**c**ute(new Runnable() {  public void run() {…}});} |

# 线程的生命周期

|  |  |  |
| --- | --- | --- |
| New（初始） | new一个线程 | |
| Runnable（可运行/就绪） | 线程对象调用了start()，或者调用了yield() | |
| Running（运行） | 线程启动后，线程体run()执行 | |
| Blocked（阻塞） | 线程因为某些原因放弃cpu，线程会重新进入Runable | |
| 原因 | 1)sleep 2)join 3)waite 4)获取对象同步锁被占用 5)发出了I/O请求 |
| Dead（死亡） | 线程执行完run()中的代码，或调用了stop()，或遇到未捕获的异常 | |

# 同步、异步、并行、并发

|  |  |
| --- | --- |
| 同步 | 发出一个同步调用时，在没有得到结果之前，该调用就不返回。也就是必须一件一件事做，等前一件做完了才能做下一件事 |
| 异步 | 发出一个异步调用时，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。或通过回调函数处理这个调用 ??? |
| 并发 | 指两个或多个指令在同一时间间隔内执行，是逻辑层面上的同时工作 |
| 并行 | 指两个或多个指令在同一时刻执行，是物理层面上的同时工作。并行是并发的一个“子集”，这些指令可能执行于同一CPU的多核上，或者多个CPU上，或者多个物理主机甚至多个网络上 |

# 线程安全、线程同步

|  |  |
| --- | --- |
| 线程安全 | 多个线程在执行同一段代码的时候，每次的执行结果和单线程执行的结果都是一样的，不存在执行结果的二义性，就可以称作是线程安全的。线程安全问题多是由全局变量和静态变量等共享资源引起的。当多个线程对共享数据只执行读操作，不执行写操作时，一般是线程安全的。 |
| 线程同步 | 将操作共享数据的代码行作为一个整体，同一时间只允许一个线程执行，执行过程中其他线程不能参与执行。 |

# 实现线程安全的方式

|  |
| --- |
| 1. 不使用单例模式，而是使用多实例 2. 线程同步：使用Synchronized和Lock，可以锁方法和代码块；使用Lock，并发高的情况下，Lock性能比较好 3. 使用java.util.concurrent下面的类库 |

# ReadWriteLock

|  |  |
| --- | --- |
| ReadWriteLock rwl = new ReentrantReadWriteLock() | |
| rwl.writeLock().lock(); // 取到写锁 | rwl.readLock().lock(); // 取到读锁 |
| rwl.writeLock().unlock(); // 释放写锁 | rwl.readLock().unlock(); // 释放读锁 |
| 自动实现，能读和写互斥，写和写互斥，读和读不互斥 | |

# 线程不安全、线程安全归类

|  |  |
| --- | --- |
| 线程不安全 | 线程安全 |
| String, StringBuilder | StringBuffer |
| ArrayList, LinkedList | Vector, Stack, CopyOnWriteArrayList |
| HashSet | CopyOnWriteArraySet |
| HashMap | Hashtable, ConcurrentHashMap |
|  | AtomicLong, AtomicInteger |
|  | BlockingQueue 和 BlockingDeque 的实现类 |

# 锁对象

|  |
| --- |
| 多个需要同步的线程在访问该同步块时，看到的应该是同一个锁对象引用。通常我们会使用this来作为锁对象  在方法上使用synchronized，那么同步监视器对象就是当前方法所属对象，即this  当我们对一个静态方法加锁，那么该方法锁的对象是类对象（类名.class）。即使用不同的对象调用该方法，也会同步。 |

# sleep()和wait()的区别

|  |  |
| --- | --- |
| sleep() | wait() |
| 是线程类Thread中的一个静态方法 | 是Object类中的方法，每个类中都可以被调用 |
| 不会导致线程释放对象锁；其它线程必须等待其执行结束，也可以调用interrupt()强行打断；在sleep时间结束后重新参与cpu时间抢夺，不一定会立刻被执行 | 会导致线程释放对象锁，进入等待此对象的等待池，只有针对此对象调用notify/notifyAll()后，本程序才进入对象的锁池准备获取对象锁进入运行状态 |

# 锁池、等待池

|  |  |  |
| --- | --- | --- |
| 对象锁 | 锁池 | 等待池 |
|  | notify/notifyAll() | wait() |

# 线程池

|  |  |  |
| --- | --- | --- |
| 好处 | 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗 | |
| 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行 | |
| 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控 | |
| [Java](http://lib.csdn.net/base/java)自带线程池 | newFixedThreadPool | 定长 |
| newCachedThreadPool | 根据需要创建新线程，最大Interger.MAX\_VALUE |
| newSingleThreadExecutor | 返回一个只有一个线程的线程池，即每次只能执行一个线程任务，多余的任务按FIFO方式顺序执行。如果这个线程因为异常结束，那么会有一个新的线程执行剩下的任务。 |
| newScheduleThreadPool | 定长，定时的以及周期性的任务执行 |

# 死锁

|  |  |
| --- | --- |
| 定义 | 指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去 |
| 四个必要条件 | 1. 互斥条件：进程对其所要求的资源进行独占性控制； 2. 请求保持条件：进程已经保持了至少一个资源，但又提出新的资源申请，而该资源又得不到满足，于是该进程阻塞，但其原先获得的资源保持不放； 3. 不可剥夺条件：进程已获得的资源在未使用完之前，不能被剥夺，只能在使用完后由自己释放； 4. 循环等待条件：当发生死锁时，在进程资源有向图中存在环路。 |
| 避免死锁 | 1. 加锁顺序：当多个线程需要相同的一些锁，确保所有的线程都是按照相同的顺序获得锁 2. 加锁限时：线程尝试获取锁的时候加上时间限制，超过时限则放弃对该锁的请求，并释放已占有的资源 3. 死锁检测： ??? |