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)

# 

# Java执行命令

|  |  |  |
| --- | --- | --- |
| javac | 将java源文件编译为class字节码文件 | javac HelloWorld.java |
| java | 运行java程序 | java HelloWorld |

# 配置环境变量

|  |  |  |
| --- | --- | --- |
| JAVA\_HOME | JDK安装路径 | E:\tts9\jdk1.8.0\_45 |
| PATH | JDK工具命令路径 | %JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin; |
| CLASSPATH | JDK工具命令所对应的工具包路径 | .;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar; |

# Java编译及运行过程

|  |  |
| --- | --- |
| 编译 | Java源文件（\*.java)经过Java编译器(javac)编译成JVM能够识别的Java二进制字节码文件(\*.class) |
| 运行 | 1. 代码装入：类装载器读取和装载程序所需的类（需要用到该类的时候才加载） 2. 代码校验(?)：1)class文件结构检查 2)类型数据的语义检查 3)字节码校验 4)符号引用的校验 3. 代码执行：Java字节码经过JVM，解析为具体平台的机器码，并运行 |
| 跨平台 | 不同平台有不同的JVM，主流平台都提供了相应的JVM |

# JDK, JRE, JVM关系

|  |  |  |
| --- | --- | --- |
| JVM: Java虚拟机 | JRE: Java运行环境 | JDK: Java开发工具包 |
| Java Virtual Machine | Java Runtime Environment | Java Development Kit |
| 是一个虚拟的计算机，通过在实际的计算机上仿真模拟各种计算机的功能来实现的；有完善的硬体架构，如处理器，堆栈，寄存器(?)等，还有相应的指令系统；屏蔽了与具体操纵系统平台相关的信息(?)，使得Java程序只需生成JVM上运行的字节码，就可以在多平台上不加修改地运行 | 运行Java程序所必须的软件环境，包含JVM标准实现和Java核心类库 | 是整个Java的核心，包含JRE，Java开发工具和Java标准类库(?)  开发工具如:   1. javac:编译Java程序 2. java:启动JVM运行java程序 3. javadoc:生成文档 4. jar:打开包(?) |

# int和Integer

|  |  |
| --- | --- |
| int：种基本数据类型之一 | Integer：int的封装类 |
| 默认值为0，即无法表达出未赋值的情况。  在JSP开发中，int的默认值为0，所以用el表达式在文本框中显示时，显示0。所以int不适合作为web层的表单数据的类型 | 默认值为null，即可以区分出未赋值和值为0的情况。  在JSP开发中，Integer的默认为null，所以用el表达式在文本框中显示时，显示空白字符串 |
| 在Hibernate中，如果将OID定义为了int类型，还需要在hbm映射(?)文件中设置其unsaved-value属性为0 | 在Hibernate中，如果将OID定义为Integer类型，那么Hibernate就可以根据其值是否为null而判断一个对象是否是临时的 |
|  | parseInt, |
|  | MAX\_VALUE, MIN\_VALUE |

# 静态变量、实例变量

|  |  |
| --- | --- |
| 静态变量 | 实例变量 |
| 属于类  注：所以没有继承不继承的说法，如果子类也有同名的静态属性或静态方法，则会被隐藏。（即便在子类中重用了超类中的静态方法的名称，也不会给你带来任何新的东西，但是却会丧失很多东西——《java解惑》）。 | 属于某个对象的属性 |
| 只要程序加载了类的字节码，静态变量就会被分配空间，静态变量就可以被使用了，通过类名来调用 | 必须创建了实例对象，其中的实例变量才会被分配空间，才能使用这个实例变量 |

# 静态方法不能发出对非静态方法的调用

|  |
| --- |
| 因为非静态方法是要与对象关联在一起的，必须创建一个对象后，才可以在该对象上进行方法调用，而静态方法调用时不需要创建对象，可以直接调用。当一个静态方法被调用时，可能还没有创建任何实例对象，如果从一个静态方法中发出对非静态方法的调用，那这个非静态方法是关联到哪个对象上的呢？这个逻辑无法成立。 |

# 对象、类

|  |  |
| --- | --- |
| 对象 | 人们要进行研究的任何事物，它能代表具体的事物，还能表示抽象的规则，计划或事件；对象具有状态，用数据值来描述对象的状态；对象还有操作，用于改变对象的状态，对象及其操作就是对象的行为（?）；对象实现了数据和操作的结合，使数据和操作封装于对象的统一体中 |
| 类 | 具有相同特征（数据元素）和行为（功能）的对象的抽象就是类；类的实例是对象，类实际上就是一种数据类型；类具有属性，它是对象的状态的抽象，用数据结构来描述类的属性；类具有操作，它是对象的行为的抽象，用操作名和实现该操作的方法来描述 |
| 区别 | 类是一个抽象的概念，不存在于现实中的时间和空间里，只为对象定义抽象的属性和行为；对象是类的一个具体，它是实实在在存在的东西。  类是一个静态的概念，类本身不携带数据，当没有为类创建对象时，类本身不存在于内存中；对象是一个动态的概念，每一个对象都存在着有别于其它对象的独特的属性和行为，对象的属性可以随着它自己的行为而发生改变 |
| 简答 | 对象：对象是类的一个实例，有状态和行为。  类：类是一个模板，它描述一类对象的行为和状态。 |

# 面向过程、面向对象

|  |  |
| --- | --- |
| 面相过程 | 面相对象 |
| 分析出解决问题的步骤，使用方法将这些步骤一步步实现，使用时在一个个调用 | 把构成问题的事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描述某个事物在整个解决问题的步骤中的行为 |
| 一件事"该怎么做" | 一件事"该让谁来做"，然后那个"谁"就是对象，他要怎么做是他自己的事，反正最后一群对象合力能把事做好就行了 |
| 性能比面向对象高；因为类调用时需要实例化，开销比较大，比较消耗资源 | 易维护，易复用，易扩展；因为面向对象有封装，继承，多态的特性，可以设计出低耦合的系统 |
| 单片机，嵌入式开发，Linux/Unix等  （性能是最重要因素） |  |

# 面向对象四大特征

|  |  |
| --- | --- |
| 抽象 | 找出一些事物的相似和共性之处，然后将这些事物归为一个类。  这个类会忽略与当前主题和目标无关的那些方面，将注意力集中在与当前目标有关的方面。 |
| 封装 | 从字面上理解就是包装的意思，专业点就是信息隐藏，是指利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分割的独立实体，数据被保护在抽象数据类型的内部，尽可能地隐藏内部的细节，只保留一些对外接口使之与外部发生联系。 |
| 继承 | 在定义和实现一个类的时候，可以在一个已经存在的类的基础之上来进行，把这个已经存在的类所定义的内容作为自己的内容，并可以加入若干新的内容，或修改原来的方法使之更适合特殊的需要，这就是继承。继承是子类自动共享父类数据和方法的机制，这是类之间的一种关系，提高了软件的可重用性和可扩展性。 |
| 多态 | 静态绑定：重载（根据参数实现多态）  动态绑定：1)继承 2)重写（根据对象实现多态） 3)向上造型（）  同一类型的引用指向不同的对象时，有不同的实现------行为的多态：医生、爸爸←我  同一对象被造型为不同的类型时，有不同的功能-----对象的多态：动物←猫、鸟 |

# 静态绑定、动态绑定

|  |  |
| --- | --- |
| 静态绑定 | 动态绑定 |
| 1. 成员变量 2. final、static、private修饰的方法 3. 构造方法 | 其它方法 |
| 静态绑定看引用的类型 | 动态绑定看对象的类型 |

# JVM加载class文件的原理机制

|  |  |  |
| --- | --- | --- |
| JVM中类的装载是由ClassLoader和它的子类来实现的，Java ClassLoader是一个重要的Java运行时系统组件。它负责在运行时查找和装入类文件的类。 | | |
| 类加载到卸载的生命周期 | 1)加载(Loading) 2)验证(Verification) 3)准备(Preparation) 4)解析(Resolution)  5)初始化(Initialization) 6)使用(Using) 7)卸载(Unloading) | |
| 获得ClassLoader的途径 | 使用当前类 | this.getClass().getClassLoader() |
| 使用当前线程 | Thread.currentThread().getContextClassLoader() |
| 使用系统ClassLoader | ClassLoader.getSystemClassLoader() |
| JVM加载类的途径 | Dog dog = new Dog();如果ClassLoader无法找到Dog，则抛出NoClassDefFoundError | |
| Class clazz = Class.forName("Dog"); 会初始化，即会执行被加载类的static块  Object dog = clazz.newInstance();  如果无法找到Dog，则抛出ClassNotFoundException | |
| Class clazz = classLoader.loadClass("Dog"); 不会初始化  Object dog = clazz.newInstance();  如果无法找到Dog，则抛出ClassNotFoundException | |

# 主要类加载器

|  |  |
| --- | --- |
| bootstrap classloader | 负责加载JAVA核心类( jre 下lib和class目录中的内容) |
| extension classloader | 负责加载JAVA扩展类(jre 下lib/ext 目录中的内容) |
| system classloader | 负责加载应用指定的类 (环境变量classpath中配置的内容) |

自定义类加载器

# 创建对象的方法

|  |
| --- |
| 使用new创建对象 |
| 利用反射创建对象，调用java.lang.Class或java.lang.reflect.Constructor类的newInstance()方法 |
| 调用对象的clone()方法创建对象 |
| 运用反序列化创建对象，调用java.io.ObjectInputStream对象的readObject()方法 |

# 堆、栈

|  |  |
| --- | --- |
| 堆 | 栈 |
| 都是在Ram里存放数据的地方。[Java](http://lib.csdn.net/base/java" \t "_blank" \o "Java 知识库)自动管理 | |
| 一个运行时数据区，类的对象从中分配空间。通过比如：new等指令建立，不需要代码显式的释放，由垃圾回收来负责 | 其数据项的插入和删除都只能在称为栈顶的一端完成，后进先出。栈中存放一些基本类型的变量和对象句柄 |
| 优点：可以动态地分配内存大小，垃圾收集器会自动回收垃圾数据。  缺点：由于其优点，所以存取速度较慢。数据不共享 | 优点：读取数度比堆要快，仅次于寄存器。数据可以共享。  缺点：存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。 ? |

# 垃圾回收(Garbage Collection)

|  |  |
| --- | --- |
| Java的一个重要特征，JVM自带的一个线程，在系统认为需要时自动启动一个线程处理 | |
| 作用 | 用于回收没有任何引用所指向的对象，释放不再被使用的内存 |
| 机制 | GC通过有向图的进行可达性分析（?），不可达的对象就被视为是垃圾对象 |
| 触发条件 | GC在优先级最低的线程中运行，一般在应用程序空闲即没有应用线程在运行时被调用，  当创建新对象申请内存时，没有足够的内存空间时，JVM会强制调用GC线程。 若GC一次后仍不能满足内存分配，JVM会再进行两次GC， 若仍无法满足要求，则JVM将报"out of memory"的错误，Java应用停止 |
| 减少GC开销的措施 | 1. 不要显式调用System.gc() 2. 尽量减少临时对象的使用 3. 对象不用时最好显式置为null：有利于GC收集器判定垃圾，从而提高GC的效率 4. 尽量使用StringBuffer，而不是String累加字符串 5. 能用基本类型，就不用包装类：基本类型变量占用的内存资源比相应对象占用的少得多 6. 尽量少用静态对象变量：属于全局变量，不会被GC回收 7. 分散对象创建，删除的时间 |

# 内存泄漏、内存溢出

|  |  |
| --- | --- |
| 内存泄漏 memory leak | 内存溢出 out of memory |
| 指程序在申请内存后，无法释放已申请的内存空间；内存对象已经不需要的时候，还仍然保留着这块内存和它的访问方式（引用）；内存泄漏会最终导致内存溢出 | 指程序在申请内存时，没有足够的内存空间供其使用 |

# Overload和Override的区别

|  |  |
| --- | --- |
| 重载 Overload | 重写 Override |
| 发生在一个类中 | 发生在父子类之间 |
| 方法名相同，参数列表不同（与返回值类型无关） | 方法签名相同，返回值类型相同（或是父类返回值的子类）；  覆盖的方法抛出的异常<=父类的，访问权限>=父类的（如果父类为private，相当于子类中增加了一个全新的方法） |

# 接口、抽象类

|  |  |
| --- | --- |
| 抽象类 | abstract修饰的类即为抽象类，抽象类不能创建实例对象，含有abstract修饰的方法的类必须定义为抽象类。抽象类中的抽象方法必须在子类中实现，所以不能有抽象的构造方法或抽象的静态方法。如果子类没有实现抽象父类中的所有方法，那么子类也必须定义为抽象类 |
| 接口 | 是抽象类的特例，只包含常量(默认public static final)和抽象方法。接口中的方法只能为public，且默认即为public abstract |

# 常用包

|  |  |
| --- | --- |
| lang | 系统默认导入的包  (Object, Number, Math, String, StringBuffedr, StringBuilder, System, ClassLoader, Throwable, Thread, 封装类, *Comparable*, *Runnable*) |
| util | 提供了各种实用功能的类和接口  (Date, Calendar, Random, Scanner, Arrays, [Properties](https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html" \o "class in java.util), Timer, HashMap, HashSet, Hashtable, Collections, *Comparator*, *Iterator*, *Collection*, *Deque*, *Queue*, *List*, *Map*, *Set*) |
| io | 提供了系统输入输出类和接口 (File, 各种流, *FileFilter*, *Serializable*) |
| net | 提供了实现网络应用程序的类 (Socket, URI, URL, URLDecoder, URLEncoder) |
| sql | 提供了使用Java访问并处理存储在数据源中的数据的API  (DriverManager, *[PreparedStatement](https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html" \o "interface in java.sql)*, *Statement, Connection, ResultSet*) |
| math | 提供用于执行任意精度整数（BigInteger）、小数（BigDecimal）算法的类 |
| text | 提供以与自然语言无关的方式来处理文本、日期、数字和消息的类和接口 ([SimpleDateFormat](mk:@MSITStore:C:\\Users\\Administrator.ppp-PC\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/text/SimpleDateFormat.html" \o "java.text 中的类)) |

# Object的方法

|  |
| --- |
| 1)getClass 2)hashCode 3)equals 4)clone 5)toString 6)notify 7)notifyAll 8)wait 9)finalize |

# Equals()和HashCode()

|  |  |
| --- | --- |
| equals | 用来判断两个对象是否相等 |
| hashCode | 是一个散列码，用来在散列存储结构中(HashMap，HashSet)确定对象的存储地址 |
| 在每个覆盖了equals方法的类中，也必须覆盖hashCode方法 | |
| 如果两个对象相等，它们的hashcode必须相等;如果两个对象的hashcode相等，这两个对象不一定相等 | |

# "=="和"equals"

|  |  |
| --- | --- |
| == | 对于基本数据类型用来比较值，对于对象用来比较对象的地址 |
| equals | 此方法其实是交给开发者去重写的，让开发者自己去定义在什么情况下两个对象相等  默认情况下：public boolean equals(Object obj) { return (this == obj); } |

# String常用方法

|  |  |
| --- | --- |
| equals[IgnoreCase] | 将此字符串与指定的对象比较 |
| charAt | 返回指定索引处的char值 |
| compareTO[IgnoreCase] | 按字典顺序比较两个字符串 |
| concat | 将指定字符串连接到此字符串的结尾 |
| contains | 当且仅当此字符串包含指定的char值序列时，返回true |
| endsWith / startWith | 测试此字符串是否以指定的后缀结束/前缀开始 |
| indexOf / lastIndexOf | 返回指定子字符/字符串在此字符串中第一次出现处的索引 |
| isEmpty | 当且仅当[length()](mk:@MSITStore:C:\\Users\\Administrator.ppp-PC\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \l "length())为0时返回true |
| length | 返回此字符串的长度 |
| matches (String regex) | 告知此字符串是否匹配给定的[正则表达式](mk:@MSITStore:C:\\Users\\Administrator.ppp-PC\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html" \l "sum)，默认包含^ $边界检查 |
| replace (char oldChar, char newChar) | 使用指定的字面值替换此字符串所有匹配给定的字面值的子字符串 |
| replaceAll | …replacement…[正则表达式](mk:@MSITStore:C:\\Users\\Administrator.ppp-PC\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html" \l "sum)… |
| split | 根据给定[正则表达式](mk:@MSITStore:C:\\Users\\Administrator.ppp-PC\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html" \l "sum)的匹配拆分此字符串 |
| substring | 返回一个新字符串，它是此字符串的一个子字符串 |
| toCharArray | 将此字符串转换为一个新的字符数组 |
| toLowerCase / toUpperCase | 将String中的所有字符都转换为小写/大写 |
| trim | 返回字符串的副本，忽略前导空白和尾部空白 |
| valueOf | 返回xxx参数的字符串表示形式 |

# String, int和Integer之间的转换

|  |  |
| --- | --- |
| String→int | ①int i = Integer.parseInt("2"); |
| int→String | ①String str = String.valueOf(2); ②String str = Integer.toString(2);  ③String str = "" + 2; |
| String→Integer | ①Integer it = Integer.valueOf("2"); ②Integer it = new Integer("2"); |
| Integer→String | ①String str = String.valueOf(it); ②String str = Integer.toString(it);  ③String str = "" + it; |
| int→Integer | ①Integer it = Integer.valueOf(2); ②Integer it = new Integer(2);  ③Integer it = 2; |
| Integer→int | ①int i = it.intValue(); ②int i = it; |

# String和StringBuffer

|  |  |  |
| --- | --- | --- |
| String | StringBuffer | StringBuilder |
| 被final修饰，不能被继承使用，一旦声明不能被改变 | 长度可变 | 长度可变 |
|  | 线程安全，性能下降 |  |
| 重写了equals | 没有重写equals | 没有重写equals |

# 常量池

|  |  |  |
| --- | --- | --- |
| 在编译期被确定,并被保存在已编译的.class文件中的一些数据 | | |
| 1.6在方法区 | 1.7在堆内存 | 1.8在元空间 |

**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" \t "_blank" \o "Java 知识库)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： ??? |