目录

[1.1.1 Java语言发展史 2](#_Toc15932333)

[1.1.2 Java语言平台版本 2](#_Toc15932334)

[Java Development Kit 2](#_Toc15932335)

[Java语言平台版本 3](#_Toc15932336)

[1.1.3 Java语言特点 3](#_Toc15932337)

[1.1.4 JRE与JDK 4](#_Toc15932338)

[1. 定义 4](#_Toc15932339)

[2、区别 4](#_Toc15932340)

[1.1.5 JAVA核心机制与JVM运行原理 4](#_Toc15932341)

[1.Java语言运行的过程 4](#_Toc15932342)

[2.JVM： 5](#_Toc15932343)

[1.2.1 JDK安装与配置 6](#_Toc15932344)

[1.2.4 JAVA程序运行原理 6](#_Toc15932345)

[1.3.2 常量、变量内存空间分配与原理 6](#_Toc15932346)

[1.3.5 运算符与表达式 8](#_Toc15932347)

[1.3.7 标识符与关键字 8](#_Toc15932348)

[1.4.6 特殊流程控制结构 8](#_Toc15932349)

[1.5.1 数组概述及作用（P1，P2） 9](#_Toc15932350)

[1.5.3 数组的初始化与内存分析 9](#_Toc15932351)

[1.6.3方法调用的过程分析 10](#_Toc15932352)

[1.7.2 类和对象 11](#_Toc15932353)

[1.7.4 Java内存管理与垃圾回收 11](#_Toc15932354)

[Java运行时内存区域 11](#_Toc15932355)

[垃圾回收 13](#_Toc15932356)

[创建对象及内存分配 16](#_Toc15932357)

[1.7.8 this关键字 17](#_Toc15932358)

[构造方法中的this关键字 17](#_Toc15932359)

[非构造方法中的this关键字 17](#_Toc15932360)

[继承关系下的this关键字 18](#_Toc15932361)

[1.7.12 Java代码块 18](#_Toc15932362)

[1.7.14 方法重写 Override 19](#_Toc15932363)

[方法的重写规则 19](#_Toc15932364)

[重载规则: 20](#_Toc15932365)

[1.7.15 多态 20](#_Toc15932366)

[1.8.3 String类、StringBuffer类、StringBuilder类 21](#_Toc15932367)

[基本类型包装类(Integer,Character等) 21](#_Toc15932368)

[1.8.8 Date类、DateFormat类、Calendar类 22](#_Toc15932369)

[1.9.1 Java异常处理机制 22](#_Toc15932370)

[一、Throwable类 22](#_Toc15932371)

[1.9.2 使用try..catch..finally处理异常 23](#_Toc15932372)

[1.10.2 Collection接口 23](#_Toc15932373)

[1、Collection接口 23](#_Toc15932374)

[d1.10.3 List接口及实现类（ArrayList、LinkedList） 25](#_Toc15932375)

[1.10.4 Set接口及实现类（HashSet、TreeSet） 25](#_Toc15932376)

[1.10.5 Map接口及实现类（HashMap、ConcurrentHashMap） 26](#_Toc15932377)

[一、HashMap和TreeMap区别 26](#_Toc15932378)

[二、HashMap和Hashtable的区别 26](#_Toc15932379)

[三、HashSet和HashMap的区别 27](#_Toc15932380)

[四、ConcurrentMap 27](#_Toc15932381)

[五、LinkedHashMap是HashMap的一个子类 27](#_Toc15932382)

[1.10.6 Collections工具类 28](#_Toc15932383)

[1.10.7 Arrays工具类 28](#_Toc15932384)

[1.10.8 Iterator接口 Enumeration 28](#_Toc15932385)

[1）两者的函数接口不同 28](#_Toc15932386)

[2）Iterator支持fail-fast机制，而Enumeration不支持 29](#_Toc15932387)

[1.10.9 泛型类、泛型方法、泛型接口 29](#_Toc15932388)

[定义泛型 29](#_Toc15932389)

[泛型方法 30](#_Toc15932390)

[2）泛型方法要注意的事项 30](#_Toc15932391)

[类型通配符 30](#_Toc15932392)

[链表 31](#_Toc15932393)

[单项链表 31](#_Toc15932394)

[双向链表,也叫双链表 31](#_Toc15932395)

# 1.1.1 Java语言发展史

Oak由于该名字已被注册后改名为java

# 1.1.2 Java语言平台版本

## Java Development Kit

1.0.2

1.1.1 ◊ 1.1.6 ◊ 1.1.7 ◊ 1.1.8

1.2.1

Java 2 SDK

1.2.2

1.3.0 ◊ 1.3.1

1.4.0 ◊ 1.4.1 ◊1.4.2 ◊1.5.0(5.0) ◊1.6.0(6.0) ◊1.7.0(7.0)

jdk8

## Java语言平台版本

J2SE(Java 2 Platform Standard Edition)标准版

该技术体系是其他两者的基础，可以完成一些桌面应用程序的开发。

J2EE(Java 2 Platform Enterprise Edition)企业版

是为开发企业环境下的应用程序提供的一套解决方案，该技术体系中包含的技术如 Servlet、Jsp等，主要针对于Web应用程序开发。

J2ME(Java 2 Platform Micro Edition)小型版

是为开发电子消费产品和嵌入式设备提供的解决方案。

# 1.1.3 Java语言特点

简单性、解释性、面向对象、高性能、分布式处理、多线程、健壮性、安全性

开源跨平台（因为有了JVM，所以同一个Java程序在三个不同的操作系统中都可以执行。这样就实现了Java程序的跨平台性。也称为Java具有良好的可移植性）

# 1.1.4 JRE与JDK

## 1. 定义

Jre是java运行环境，而不是开发者，包括java虚拟机和java平台核心类和支持文件，不包含开发工具，调试器、编译器等。

Jdk是java开发工具包，提供java开发环境，（提供了编译器 javac 等工具，用于将java文件编译成class文件）和运行环境（提供了jvm和runtime辅助包用于解析class文件使其运行）

Jdk包含jre，是整个java的核心

## 2、区别

JRE 主要包含：java 类库的 class 文件(都在 lib 目录下打包成了 jar)和虚拟机(jvm.dll)；JDK 主要包含：java 类库的 class文件(都在 lib 目录下打包成了 jar)并自带一个 JRE。那么为什么 JDK 要自带一个 JRE 呢？而且 jdk/jre/bin 下的 client 和 server 两个文件夹下都包含 jvm.dll(说明 JDK 自带的 JRE 有两个虚拟机)。

JavaSE 可以开发和部署在桌面、服务器、嵌入式环境和实时环境中使用的 Java 应用程序。是EE，和ME的基础。

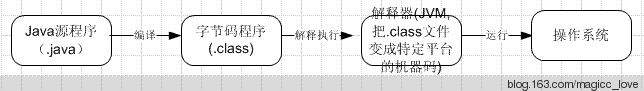
JavaEE 前端开发，使用于动态开发网页制作。

JavaME 适用于安卓开发。

# 1.1.5 JAVA核心机制与JVM运行原理

## 1.Java语言运行的过程

Java语言写的源程序通过Java编译器，编译成与平台无关的‘字节码程序’(.class文件，也就是0，1二进制程序)，然后在OS之上的Java解释器中解释执行。



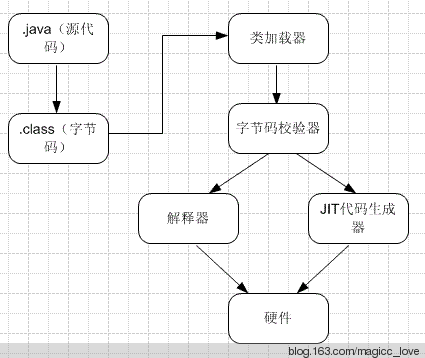
注：JVM（java虚拟机）包括解释器，不同的JDK虚拟机是相同的，解释器不同。

## 2.JVM：

Jvm是java的核心和基础，在java编译器和os平台之间的虚拟处理器。它是一种利用软件方法实现的抽象的计算机基于下层的操作系统和硬件平台，可以在上面执行java的字节码程序。

Java编译器只要面向JVM，生成JVM能理解的代码或字节码。

JVM执行程序的过程：加载.class文件->管理并分配内存->执行垃圾收集



# 1.2.1 JDK安装与配置

JAVA\_HOME：jdk的目录 E:\Java\jdk1.8.0\_161

Path后面加 %JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;

新建CLASSPATH变量 .;%JAVA\_HOME%\lib;%JAVA\_HOME%\lib\tools.jar

# 1.2.4 JAVA程序运行原理

\*.java→\*.class→机器码->java编译器 (编译) → 虚拟机(解释执行) →  解释器(翻译) → 机器码

Java中，类加载器把一个类装入JAVA虚拟机需要经过三个步骤来完成：装载、链接、初始化，其中链接又分来校验、准备、解析过程

装载：查找和导入.class文件。

链接：检查装入.class文件的正确性，然后，java虚拟机为变量分配内存，设置默认值。

初始化：把符号引用变成直接引用。

# 1.3.2 常量、变量内存空间分配与原理

1、栈区（stack）由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、堆区（heap）由程序员分配释放，若程序员不释放，程序借宿时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3、全局区（静态区）（static）全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。

4、文字常量区—常量字符串就是放在这里的，程序结束后由系统释放。

5、程序代码区—存放函数体的二进制代码。

int a = 0; 全局初始化区

char \*p1; 全局未初始化区

main()

{

int b; 栈

char s[] = "abc"; 栈

char \*p2; 栈

char \*p3 = "123456"; 123456\0在常量区，p3在栈上。

static int c =0； 全局（静态）初始化区

p1 = (char \*)malloc(10);

p2 = (char \*)malloc(20);

分配得来得10和20字节的区域就在堆区。

strcpy(p1, "123456"); 123456\0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。

}

# 1.3.5 运算符与表达式

Int x=5;

Int y=++x; y的值为6

Int y=x++; y的值为5，x的值为6

# 1.3.7 标识符与关键字

**Java中的名称命名规范**

**（1）包名：多单词组成时所有字母都小写：xxxyyyzzz**

**（2）类名、接口名：多单词组成时，所有单词的首字母大写：XxxYyyZzz**

**（3）变量名、方法名：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写：xxxYyyZzz**

**（4）常量名：所有字母都大写。多单词时每个单词用下划线连接：XXX\_YYY\_ZZZ**

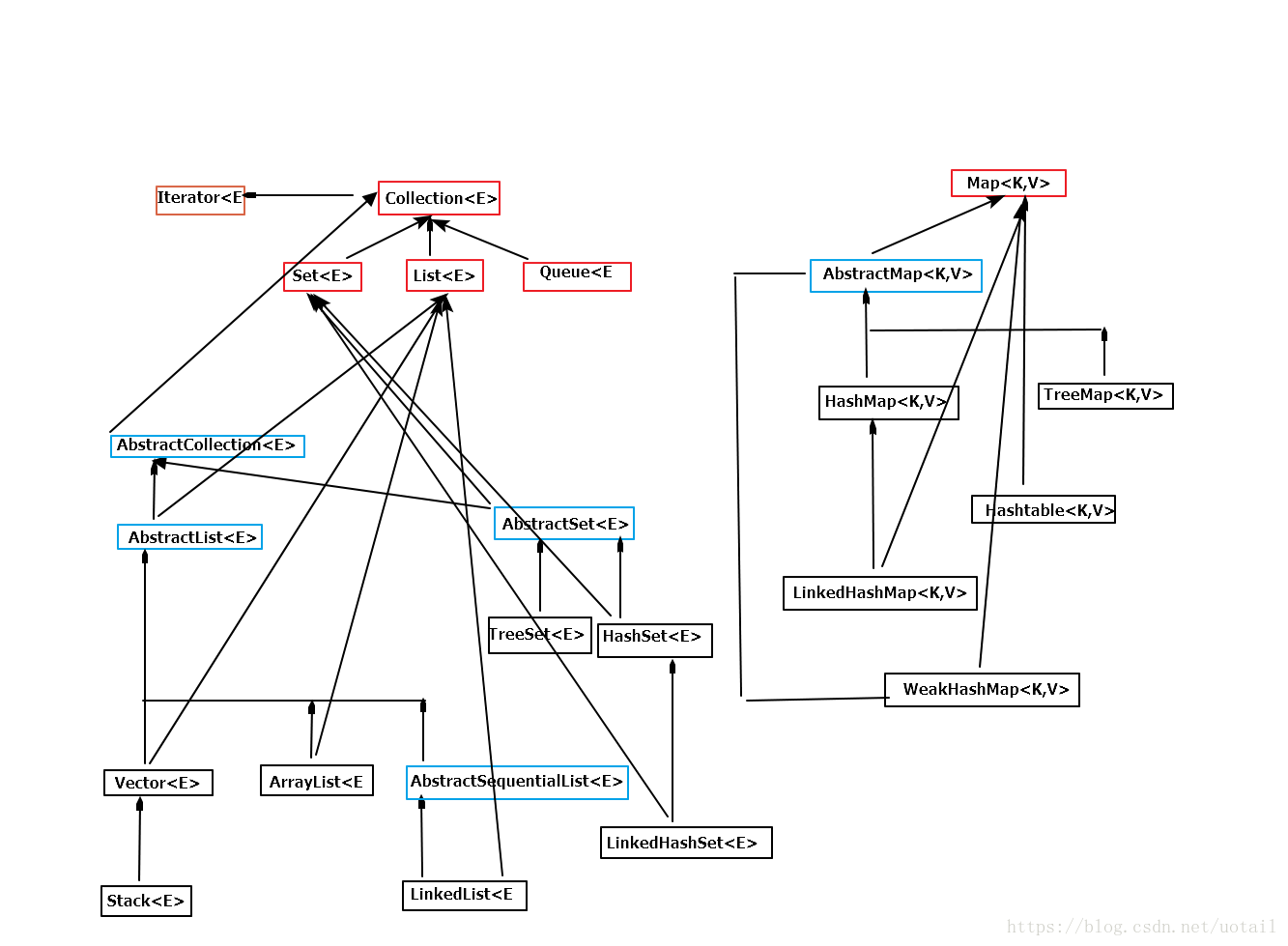
# 1.4.6 特殊流程控制结构

**Break：终止循环；   
Continue：结束本次循环，进入下一次循环；   
return：结束程序，返回结果；**注意：Break和Continue二者是适用于循环中

# 1.5.1 数组概述及作用（P1，P2）

**数组（可以存储基本数据类型）是用来存现对象的一种容器，但是数组的长度固定，不适合在对象数量未知的情况下使用。**

**集合（只能存储对象，对象类型可以不一样）的长度可变，可在多数情况下使用**。



# 1.5.3 数组的初始化与内存分析

**初始化的两种方式：**

**静态初始化：初始化时由我们自己指定每个数组元素的初始值，由系统决定需要的数组长度；**

**格式：数组名 = new 数组类型[]{元素1,元素2,元素3...元素n};   a1 = new  int{1,2,3,4,5};**

**简化语法：数组名 = {元素1,元素2,元素3...元素n};                      a1 =  {1,2,3,4,5};**

**动态初始化：初始化时由我们指定数组的长度，由系统为数组元素分配初始值；**

**格式：数组名 = new 数组类型[数组长度];                          a1 = new int[ 8];**



# 1.6.3方法调用的过程分析

**假设调用x.f(args)，隐式参数x声明为类C的一个实例对象：**

**1.编译器查看对象的声明类型和方法名。例如，可能存在方法f(int)和方法f(String)。编译器将会一一列举出所有该类中名为f的方法和其超类中访问属性为public且名为f的方法。**

**2.编译器将查看调用方法时提供的参数类型。如果在所有名为f的方法中存在一个与提供的参数类型完全匹配，就选择这个方法。这个过程称为重载解析。**

**3.如果是private方法、static方法、final方法或者构造器，那么编译器将可以准确地知道应该调用哪个方法，这种方法称为静态绑定。与此对应的，调用的方法依赖于隐式参数的实际类型，并且在运行时才动态绑定。**

**4.当程序运行时，并且采用动态绑定的方式，虚拟机一定调用与x所引用对象的实际类型最合适的那个类的方法。假设x的实际类型为D，它是C类的子类。如果D类定义了方法f(String)，就直接调用它；否则，在其父类中找到f(String)，以此例推。每次调用方法，虚拟机预先为每个类创建了一个方法表，其中列出了所有方法的签名和实际调用的方法。在真正调用方法时，虚拟机仅仅查找这个表。**

**虚拟机运行过程：**

**调用e.getSalary()的解析过程：**

**1.首先，虚拟机提取e的实际类型的方法表。**

**2.虚拟机搜索定义getSalary签名的类。**

**3.虚拟机调用方法。**

# 1.7.2 类和对象

**类是一种抽象的概念集合，是最基础的组织单位，作为对象的模板、合约或蓝图。**

**类是对象的类型，使用一个通用类可以定义同一类型的对象，类中定义对象的数据域是什么以及方法是做什么的。 对象是类的实例，一个类可以拥有多个实例，创建实例的过程叫做实例化。实例也称为对象，两者说法一致**

# 1.7.4 Java内存管理与垃圾回收

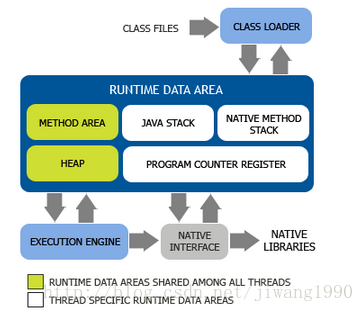
## Java运行时内存区域

**Java运行时内存区域如图2-1所示，内存区域逻辑上被划分为：程序计数器，栈，本地方法栈，堆，方法区。其中程序计数器，栈，本地方法栈都是线程私有的，堆和方法区被所有线程共享。**

**程序计数器用于指示当前线程执行的字节码的行号；栈用于描述Java方法执行的内存模型，每当进入一个新的方法，JVM都会在栈中创建一个栈帧（存放本地变量，参数，返回地址，操作数栈）；本地方法栈是本地方法执行的内存模型，HotSpot虚拟机将栈和本地方法栈合二为一。**

**在提及到栈的时候，我们需要涉及两个异常：StackOverflowError和OutOfMemoryError，这两个异常的区别在于：当线程请求的栈深度超过虚拟机允许的栈深度时，会抛出StackOverflowError；当虚拟机无法为线程扩展栈分配足够的空间时，会抛出OutOfMemoryError。**

**设置栈深度的示例：-Xss128k    ==>>   设置栈的深度为128KB**



**下面我们来看一下另一块非常重要的区域：堆，Java堆是用于存放Java对象实例的主要区域，通过new，clone，反序列化创建的对象都存放在堆中，为什么Java要把对象存放在堆中，而不是栈中呢？**

**C++由于没有垃圾回收机制，所以当定义一个变量时，其内存是在栈中分配的，只有通过new显式的创建一个对象时，对象才会从堆中分配内存，并且此时需要通过delete显式的释放对象占用的内存，否则会造成内存泄露。Java中除了基本类型变量（boolean, byte, char, short, int, long, float, double），其他类型的变量基本都是通过new来创建，所以其内存都是从堆中分配，当对象废弃时，垃圾收集器会自动回收这部分内存。由于堆是各个线程共享的内存区域，所以把对象存放在堆中有利于线程之间的通信（共享内存）。正如之前我们在描述栈时所看到的，JVM会为每个方法创建一个栈帧，所以如果对象存放在栈中，方法调用的参数将需要从调用方法的栈帧拷贝到被调用方法的栈帧，如果对象存放在堆中，只需要拷贝指针或引用（此时，两个方法将指向同一个对象）。所以我们可以认为Java之所以把对象存放在堆中，其一是Java具有非常优秀的垃圾回收机制，其二把对象存放在堆中有利于线程之间共享数据及通信，其三是可以减少不必要的对象拷贝，提升方法调用的效率，同时也节约了内存。**

**因为不同的Java对象生命周期可能不同，所以基于Java对象不同的生命周期，堆被分成了两个不同的区域：新生代和老年代，新生代中对象的生命周期短，存活率低，老年代中对象的生命周期长，存活率高。基于不同的存活率，这两个区域的垃圾收集也采用了不同的算法，新生代一般采用复制算法，老年代一般采用标记-删除或标记-整理算法。复制算法就是将存活下来的对象从一个区域复制到另一个区域，标记删除和标记整理就是将需要回收的对象标记出来，然后清除掉，标记整理算法还会对内存进行整理，这样可以避免内存碎片。将Java堆分成两个不同的年代并采用不同回收算法的垃圾收集方式被称为分代收集。下一节将详细介绍垃圾收集的机制以及常用垃圾收集器。**

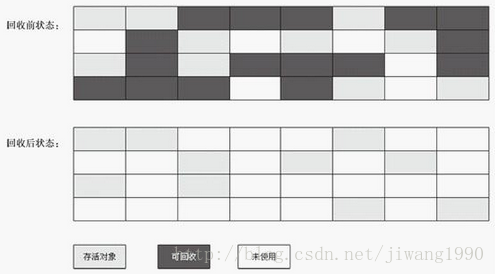
**介绍完Java堆之后，我们来看一下方法区，方法区是用于存放类信息、常量（final, static final）、静态变量(static)、即时编译器编译后的代码的地方。HotSpot虚拟机把这部分区域称为永久代，因为HotSpot虚拟机把分代收集扩展到了方法区，或者我们可以说HotSpot虚拟机通过永久代来实现方法区；同时提供了参数-XX:MaxPermSize来限制方法区的最大内存。但其实这并不是一个很好的选择，当加载的类比较多或者常量池比较大时，很容易导致内存溢出。目前HotSpot官方团队已经在逐步采用本地内存实现方法区，JDK1.7已经将常量池移出永久代。当该区域的内存无法满足要求时，也会导致内存溢出。**

## 垃圾回收

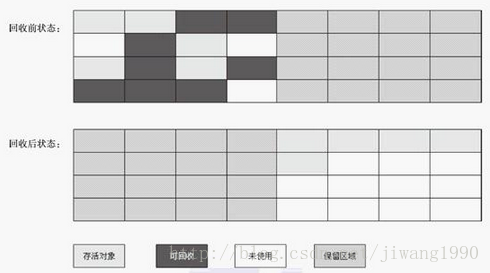
**根据前面对运行时内存区域的描述，我们知道垃圾回收主要集中在堆和方法区，方法区可以选择性实现垃圾回收，该区域的垃圾回收主要集中在回收废弃常量和类型卸载。前面我们已经提及了复制算法和标记整理算法，那在此之前我们如何知道哪些对象时废弃的，哪些对象时不能回收的呢？对于具备垃圾回收功能的语言，一般采用两种算法确定废弃对象：引用计数法（Python）和可达性分析算法（也被称为根搜索算法，C#，Lisp），Java采用可达性分析算法。引用计数法通过跟踪对象的引用计数器来确定对象是否被废弃，当一个新的引用指向该对象时，引用计数加1，当一个引用不在指向该对象时，引用计数减1，当引用计数为0时，对象被废弃。该算法在遇到堆中两个对象循环引用时（即对象A中有一个字段指向对象B，对象B中有一个字段指向A），会导致内存泄露，即这部分内存永远不会被回收，因为这两个对象的引用计数永远不为0。该算法出现问题的原因在于没有区分指向对象的引用ref的来源，假如ref位于栈或者方法区中，说明该对象没有废弃；但假如ref位于堆中，则不能确定，此时我们可以继续判断指向ref所在对象的引用ref2所在的内存区域，通过这种方法不断回溯，如果最终可以到达栈，本地方法栈或者方法区中，则说明引用链中的对象都是没有废弃的，否则都是废弃的，这就是可达性分析算法。**

**Java除了通过可达性分析算法判断哪些对象需要回收之外，还提供了不同的引用级别用于实现更加灵活的垃圾回收。Java一共提供了四种引用级别：强引用，软引用(SoftReference)，弱引用(WeakReference)，虚引用(PhantomReference)。强引用就是我们一般的引用方式，软引用指向对象在系统将要发生内存溢出时会被回收（可以用于实现缓存），弱引用指向的对象会在下一次垃圾回收时被回收，虚引用指向的对象只是会在垃圾回收时收到一个系统通知，对象的生命周期完全不会受虚引用的影响。**

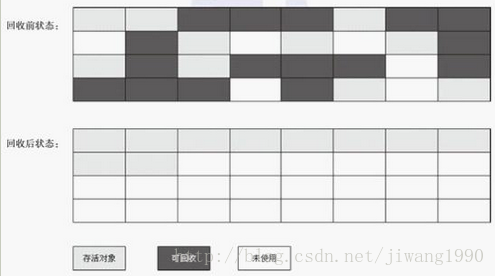
**下面我们来看一下复制算法(Copy)和标记-整理(Mark-Compact)算法的具体实现，其实这两个算法都是从标记-清除（Mark-Sweep）算法改进而来的，标记清除算法会遇到两个问题：第一个问题是效率问题，当对象存活率很低时，其实把存活对象找出来并整理到一个区域，效率会更高，这就是复制算法；第二个问题是当存活率比较高时，会出现内存碎片问题，所以出现了标记-整理算法。Java堆采用了新生代为复制算法，老年代为标记-整理或标记-清除算法的分代收集机制。**



**图2-2 标记清楚算法**



**图 2-3       复制算法**



**图2-3   标记 - 整理算法**

**复制算法将新生代分成Eden，From Survivor，To Survivor三块区域，每次垃圾收集Eden，From Survivor中存活的对象都会被复制到To Survivor中。记住，这三个区域的划分只是逻辑上的，和物理划分无关，默认Eden和Survivor的大小比例为8 : 1，比例划分这么大是为了提高内存的利用率，在这种比例下可利用的内存其实只有90%；看到这里，我想很多人可能会问，假如10%的空间不够存放生存下来的对象怎么办？JVM提供了一种被称为分配担保（Handle Promotion）的机制，由老年代为To Survivor空间提供担保，假如To Survivor没有足够的空间存放生存下来的对象，这些对象直接存放到老年代，假如老年代还不够存放，就会抛出OutOfMemoryError异常。**

**分析了垃圾收集算法的思想之后，我们来了解一下常用的垃圾收集器，新生代的垃圾收集器包括 Serial，ParNew，Parallel Scavenge，老年代的垃圾收集器包括Serial Old，CMS，ParOld。这些垃圾收集器的一个区别是单线程还是多线程，其中Serial，Serial Old是单线程的，其余是多线程的；第二个区别是垃圾收集线程和用户线程是否可以并发执行，CMS收集器可以分成初始标记，并发标记，重新标记，并发回收等过程，其中并发标记和并发回收可以与用户线程并发执行，所以它也是这些垃圾收集器中唯一真正意义上的并发收集器；Parallel Scavenge与ParOld以提高吞吐量为目的，其他收集器以减小停顿时间（Stop The World）为目的。**

## 创建对象及内存分配

**前面提到，Java创建对象的方式包括new、clone、deserialization，在虚拟机内部，这三种创建对象的方式其实是相同的。首先寻找或加载类信息，如果无法正常加载，则抛ClassNotFoundException，否则到java堆中分配内存，分配内存的方式根据内存是否规整（取决于垃圾回收算法，标记整理和复制算法的内存都是规整的，标记清除的内存不规整）有两种方式：指针碰撞和空闲列表。指针碰撞的方式中通过指针ptr将内存分成两个部分，ptr之前的部分都被使用，ptr之后的部分是空闲的，当对象需要的内存为size时，指针ptr = ptr + size。空闲列表是通过将空闲的区域通过链表连接起来，对象需要内存则遍历链表，直到遇到一个具有足够空间的元素为止。内存分配完之后就将所分配的内存初始化为0，每个对象都有一个对象头，这里保存着和对象相关的锁，对象的哈希码，对象的GC分代年龄，以及指向方法区中类型的相关引用。到此为止，对于虚拟机来说，已经成功创建了一个对象；但从Java程序来说，这才刚刚开始，接下来会执行<init>方法对所有字段进行初始化。**

**对于不同的对象，所分配的内存的区域是不同的。一般来说，优先在Eden空间中分配内存；对于大对象，优先在老年代中分配内存（size大于PrenureSizeThreshold）；当对象的年龄大于MaxTenuringThreshold时，对象也会被移动至老年代；如果Survivor空间中相同年龄的所有对象大小总和大于Survivor空间的一半，年龄大于或等于该年龄的对象直接进入老年代。**

# 1.7.8 this关键字

## 构造方法中的this关键字

1. **只能在构造方法中通过this来调用其他构造方法，普通方法中不能使用。**
2. **不能通过this递归调用构造方法，即不能在一个构造方法中通过this直接或间接调用该构造方法本身。**

**3) 通过this调用其他构造方法必须放在构造方法的第一行中执行。由于super调用父类的构造函数也必须放在构造方法的第一行中执行，因此，通过this和super调用构造方法不能同时出现一个构造方法中。也不能在一个构造方法中多次调用不同的构造方法。**

**在构造方法中也可以使用this关键字来访问本类中的成员变量和成员函数。其用法和非构造方法中的this关键字相同。**

## 非构造方法中的this关键字

**在Java中可以通过通过this关键字来调用类中的成员变量和方法。其用法是。**

**1) this.xxx; 访问类中的成员变量xxx**

**2) this.yyy(paras…); 访问类中的成员方法yyy**

**3) this; 当前类对象的引用**

**this关键字访问类的成员变量和成员函数时不受访问权限的控制，可以访问本类中所有的成员变量和方法，包括private的成员变量和方法。也可以通过this访问本类的static成员，不过由于static成员可以通过类名直接访问，如果通过this来访问会有“The static field ××× should be accessed in a static way”的警告信息。不能在类的static成员或static块中使用this。**

## 继承关系下的this关键字

**在继承关系下，父类中的this关键字并不总是表示父类中的变量和方法。this关键字的四种用法如前文所述，列举如下。**

**1) this(paras…); 访问其他的构造方法**

**2) this.xxx; 访问类中的成员变量xxx**

**3) this.yyy(paras…); 访问类中的成员方法yyy**

**4) this; 当前类对象的引用**

**对第一种，无论子类是否有相同参数的构造方法，this(paras…);访问的始终是父类中的构造方法。**

**对第二种，无论子类是否有覆盖了该成员变量，this.xxx;访问的始终是父类中的成员变量。**

**对第三种，如果子类重写了该成员方法，则this.yyy(paras…);访问的是子类的成员方法，如果子类没有重写该成员方法，则this.yyy(paras…);访问的是父类的成员方法。**

**对第四种，this始终代表的是子类的对象。**

# 1.7.12 Java代码块

**静态代码块：用staitc声明，jvm加载类时执行，仅执行一次  
构造代码块：类中直接用{}定义，每一次创建对象时执行。  
执行顺序优先级：静态块,main(),构造块,构造方法。**

**对于一个类而言，按照如下顺序执行：**

1. **执行静态代码块**
2. **执行构造代码块**
3. **执行构造函数**

**对于静态变量、静态初始化块、变量、初始化块、构造器，它们的初始化顺序依次是（静态变量、静态初始化块）>（变量、初始化块）>构造器。**

**当涉及到继承时，按照如下顺序执行：**

1. **执行父类的静态代码块，并初始化父类静态成员变量**
2. **执行子类的静态代码块，并初始化子类静态成员变量**
3. **执行父类的构造代码块，执行父类的构造函数，并初始化父类普通成员变量**
4. **执行子类的构造代码块， 执行子类的构造函数，并初始化子类普通成员变量**

# 1.7.14 方法重写 Override

## 方法的重写规则

1. **参数列表必须完全与被重写方法的相同；**
2. **返回类型与被重写方法的返回类型可以不相同，但是必须是父类返回值的派生类（java5 及更早版本返回类型要一样，java7 及更高版本可以不同）；**
3. **访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。**
4. **父类的成员方法只能被它的子类重写。**
5. **声明为final的方法不能被重写。**
6. **声明为static的方法不能被重写，但是能够被再次声明。**
7. **子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。**
8. **子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。**
9. **重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。**
10. **构造方法不能被重写。**
11. **如果不能继承一个方法，则不能重写这个方法。**

## 重载规则:

1. **被重载的方法必须改变参数列表(参数个数或类型不一样)；**
2. **被重载的方法可以改变返回类型；**
3. **被重载的方法可以改变访问修饰符；**
4. **被重载的方法可以声明新的或更广的检查异常；**
5. **方法能够在同一个类中或者在一个子类中被重载。**
6. **无法以返回值类型作为重载函数的区分标准。**

# 1.7.15 多态

**指向子类的父类引用由于向上转型了，它只能访问父类中拥有的方法和属性，而对于子类中存在而父类中不存在的方法，该引用是不能使用的，尽管是重载该方法。若子类重写了父类中的某些方法，在调用该些方法的时候，必定是使用子类中定义的这些方法（动态连接、动态调用）。**

**多态性就是相同的消息使得不同的类做出不同的响应**。

**Java实现多态有三个必要条件：继承、重写、向上转型。**

# 1.8.3 String类、StringBuffer类、StringBuilder类

**String类型声明之后不可以变。**

**StringBuffer声明之后还可以更改。**

**StringBuilder类和StringBuffer类一样，但是StringBuffer比较安全但是运行比较慢，StringBuilder运行比较块。**

# 基本类型包装类(Integer,Character等)

**基本数据类型 包装类 取值范围 占用字符**

**byte Byte -128~127 占用一个字节**

**short Short -32768~32767 占用两个字节**

**int Integer -2147483648~2147483647 占用四个字节**

**long Long 占用8个字节**

**float Float 占用4个字节**

**double Double 占用8个字节**

**char Character 占用2个字节**

**boolean Boolean**

**包装类 包装类转基本类型 基本类型转包装类**

**Byte Byte.valueOf(byte) byteInstance.byteValue()**

**Short Short.valueOf(short) shortInstance.shortValue()**

**Integer Integer.valueOf(int) integerInstance.intValue()**

**Long Long.valueOf(long) longInstance.longValue()**

**Float Float.valueOf(float) floatInstance.floatValue()**

**Double Double.valueOf(double) doubleInstance.doubleValue()**

**Character Character.valueOf(char) charInstance.charValue()**

**boolean Boolean.valueOf(booleann) booleanInstance.booleanValue()**

# 1.8.8 Date类、DateFormat类、Calendar类

**Date获取当前时间**

**DateFormat Date类获取的时间是以默认的英文格式输出日期和时间.DateDateFormat类专门用于将日期格式化为字符串或者特定格式显示的字符串转换成一个Date对象。**

**Calendar 获取时间对象**

**SimpleDateFormat 可以使用 SimpleDateFormat 来对日期时间进行格式化，如可以将日期转换为指定格式的文本，也可将文本转换为日期。**

# 1.9.1 Java异常处理机制

## 一、Throwable类

**先从Java异常类继承关系  最顶级Throwable类说起：**

**1.Throwable 可抛出的（是Object的子类）。**

**2.Throwable 类是 Java 语言中所有错误或异常的超类。只有当对象是此类（或其子类之一）的实例时，才能通过 Java 虚拟机或者 Java throw 语句抛出。类似地，只有此类或其子类之一才可以是 catch 子句中的参数类型。**

**3.两个子类的实例，Error 和 Exception，通常用于指示发生了异常情况。通常，这些实例是在异常情况的上下文中新近创建的，因此包含了相关的信息（比如堆栈跟踪数据）。**

**4.Throwable类中的常用方法**

**注意：catch关键字后面括号中的Exception类型的参数e。Exception就是try代码块传递给catch代码块的变量类型，e就是变量名。catch代码块中语句"e.getMessage();"用于输出错误性质。通常异常处理常用3个函数来获取异常的有关信息:**

**getCause()：返回抛出异常的原因。如果 cause 不存在或未知，则返回 null。**

**printStackTrace()：对象的堆栈跟踪输出至错误输出流，作为字段 System.err 的值。**

**public String getMessage() 返回此 throwable 的详细消息字符串。**

**5.Error和Exception区别**

**Error  是不可修复、不可捕获的、致命的；一般是与编译器相关的问题，如系统崩溃、虚拟机错误、动态链接库失败等。 Java虚拟机（JVM）一般会选择线程终止**

**Exception 是可以避免的；java.lang.Exception类是Java中所有异常的直接或间接父类。即Exception类是所有异常的根类。**

**注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。**

**Java中的异常分为两大类：**

**1.Checked Exception（检查型异常）**

**2.Unchecked Exception（Runtime Exception）非可检查型**

# 1.9.2 使用try..catch..finally处理异常

**在不抛出异常的情况下，程序执行完 try 里面的代码块之后，该方法并不会立即结束，而是继续试图去寻找该方法有没有 finally 的代码块，  
如果没有 finally 代码块，整个方法在执行完 try 代码块后返回相应的值来结束整个方法；  
如果有 finally 代码块，此时程序执行到 try 代码块里的 return 语句之时并不会立即执行 return，而是先去执行 finally 代码块里的代码，  
若 finally 代码块里没有 return 或没有能够终止程序的代码，程序将在执行完 finally 代码块代码之后再返回 try 代码块执行 return 语句来结束整个方法；  
若 finally 代码块里有 return 或含有能够终止程序的代码，方法将在执行完 finally 之后被结束，不再跳回 try 代码块执行 return。**

# 1.10.2 Collection接口

## 1、Collection接口

**Collection是最基本集合接口，它定义了一组允许重复的对象。Collection接口派生了两个子接口Set和List，分别定义了两种不同的存储方式，如下**

**2、 Set接口**

**Set接口继承于Collection接口，它没有提供额外的方法，但实现了Set接口的集合类中的元素是无序且不可重复。**

**特征：无序且不可重复。**

**3、  List接口**

**List接口同样也继承于Collection接口，但是与Set接口恰恰相反，List接口的集合类中的元素是对象有序且可重复。**

**特征：有序且可重复。**

**两个重要的实现类：ArrayList和LinkedList**

**1.ArrayList特点是有序可重复的**

**2.LinkedList是一个双向链表结构的。**

**4、Map接口**

**Map也是接口，但没有继承Collection接口。该接口描述了从不重复的键到值的映射。Map接口用于维护键/值对（key/value pairs）。**

**特征：它描述了从不重复的键到值的映射。**

**两个重要的实现类：HashMap和TreeMap**

**1.HashMap，中文叫散列表，基于哈希表实现，特点就是键值对的映射关系。一个key对应一个Value。HashMap中元素的排列顺序是不固定的。更加适合于对元素进行插入、删除和定位。**

**2.TreeMap，基于红黑书实现。TreeMap中的元素保持着某种固定的顺序。更加适合于对元素的顺序遍历。**

**5、Iterator接口**

**Iterator接口，在C#里有例外一种说法IEnumerator，他们都是集合访问器，用于循环访问集合中的对象。**

**所有实现了Collection接口的容器类都有iterator方法，用于返回一个实现了Iterator接口的对象。Iterator对象称作迭代器，Iterator接口方法能以迭代方式逐个访问集合中各个元素，并可以从Collection中除去适当的元素。**

**<span style="font-size:10px;">Iterator it = collection.iterator(); // 获得一个迭代子**

**while(it.hasNext())**

**{**

**Object obj = it.next(); // 得到下一个元素**

**} </span>**

**6、Comparable接口**

**Comparable可以用于比较的实现，实现了Comparable接口的类可以通过实现comparaTo方法从而确定该类对象的排序方式。**

**总结**

**Collection集合分别派生自Collection和Map接口，Collection有两个常用子接口List和Set，分别表示有序可重复，无序不可重复的集合。而Map存储的是key-value的映射。**

**上面的总结看上去很多，很繁琐，其实你只要记得一点：collection是用于处理各种数据结构的，根据各种数据结构的特点理解，一切都会变简单。**

**。这种情况通常可以用这个虚拟节点直接表示这个链表。**

# d1.10.3 List接口及实现类（ArrayList、LinkedList）

**区别如下：**

**ArrayList内部存储的数据结构是数组存储。数组的特点：元素可以快速访问。每个元素之间是紧邻的不能有间隔，缺点：数组空间不够元素存储需要扩容的时候会开辟一个新的数组把旧的数组元素拷贝过去，比较消性能。从ArrayList中间位置插入和删除元素，都需要循环移动元素的位置，因此数组特性决定了数组的特点：适合随机查找和遍历，不适合经常需要插入和删除操作。**

**Vector内部实现和ArrayList一样都是数组存储，最大的不同就是它支持线程的同步，所以访问比ArrayList慢，但是数据安全，所以对元素的操作没有并发操作的时候用ArrayList比较快。**

**LinkedList内部存储用的数据结构是链表。链表的特点：适合动态的插入和删除。访问遍历比较慢。另外不支持get，remove，insertList方法。可以当做堆栈、队列以及双向队列使用。LinkedList是线程不安全的。所以需要同步的时候需要自己手动同步，比较费事，可以使用提供的集合工具类实例化的时候同步：具体使用List<String> springokList=Collections.synchronizedCollection(new 需要同步的类)。**

**总结**

**1.内部存储结构区别：**

**ArrayList、Vector是数组存储。LinkedList是链表存储。**

**2.线程安全区别：**

**ArrayList、LinkedList线程不安全。Vector线程安全。**

**3.使用场景区别：**

**使用线程同步的时候Vector类首选或者使用Collections工具类初始化时候同步。**

**需要经常删除、增加使用LinkedList(链表结构)、经常需要查询迭代使用ArrayList(数组结构)**

# 1.10.4 Set接口及实现类（HashSet、TreeSet）

**HashSet它不保证迭代顺序，特别是它不保证该顺序恒久不变。如果我们想要元素迭代顺序与添加顺序相同，那么我们可以使用HashSet的子类java.util.LinkedHashSet<E>**

**TreeSet和hashSet用法一样**

# 1.10.5 Map接口及实现类（HashMap、ConcurrentHashMap）

## 一、HashMap和TreeMap区别

**1、HashMap是基于散列表实现的，时间复杂度平均能达到O(1)。 TreeMap基于红黑树（一种自平衡二叉查找树）实现的，时间复杂度平均能达到O(log n)。**

**2、HashMap、TreeMap都继承AbstractMap抽象类；TreeMap实现SortedMap接口，所以TreeMap是有序的！HashMap是无序的**

**3、两种常规Map性能  
          HashMap：适用于在Map中插入、删除和定位元素。  
          Treemap：适用于按自然顺序或自定义顺序遍历键(key)。**

**4.总结**

**HashMap通常比TreeMap快一点(树和哈希表的数据结构使然)，建议多使用HashMap，在需要排序的Map时候才用TreeMap。**

## 二、HashMap和Hashtable的区别

**HashMap和Hashtable都实现了Map接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。**

**HashMap几乎可以等价于Hashtable，除了HashMap是非synchronized的，并可以接受null(HashMap可以接受为null的键值(key)和值(value)，而Hashtable则不行)。  
HashMap是非synchronized，而Hashtable是synchronized，这意味着Hashtable是线程安全的，多个线程可以共享一个Hashtable；而如果没有正确的同步的话，多个线程是不能共享HashMap的。Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。  
另一个区别是HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。  
由于Hashtable是线程安全的也是synchronized，所以在单线程环境下它比HashMap要慢。如果你不需要同步，只需要单一线程，那么使用HashMap性能要好过Hashtable。  
HashMap不能保证随着时间的推移Map中的元素次序是不变的。  
  
我们能否让HashMap同步？  
HashMap可以通过下面的语句进行同步：**

**Map m = Collections.synchronizeMap(hashMap);  
  
Hashtable继承自Dictionary类，而HashMap是Java1.2引进的Map interface的一个实现.**

### 三、HashSet和HashMap的区别

**HashSet是基于HashMap实现的。**

## 四、ConcurrentMap

**ConcurrentHashMap 表现区别：不可以有null键，线程安全，原子操作。一个ConcurrentHashMap 由多个segment 组成，每个segment 包含一个Entity 的数组。这里比HashMap 多了一个segment 类。该类继承了ReentrantLock 类，所以本身是一个锁。当多线程对ConcurrentHashMap 操作时，不是完全锁住map， 而是锁住相应的segment 。这样提高了并发效率。缺点：当遍历ConcurrentMap中的元素时，需要获取所有的segment 的锁，使用遍历时慢。锁的增多，占用了系统的资源。使得对整个集合进行操作的一些方法**

## 五、LinkedHashMap是HashMap的一个子类

**LinkedHashMap保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的.也可以在构造时用带参数，按照应用次数排序。在遍历的时候会比HashMap慢，不过有种情况例外，当HashMap容量很大，实际数据较少时，遍历起来可能会比LinkedHashMap慢，因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关。**

**六、java为数据结构中的映射定义了一个接口java.util.Map;它有四个实现类,分别是HashMap Hashtable LinkedHashMap 和TreeMap.**

# 1.10.6 Collections工具类

**Collections.sort：排序。**

**Collections.shuffle：混合排序。**

**Collections.reverse：反转。**

# 1.10.7 Arrays工具类

**Arrays.asList：将数组转换成list类型，这个list的长度是固定的不可以添加或删除。**

**Arrays.sort：对数组进行排序。**

**Arrays.copyOf：数组拷贝。Arrays.copyOf(str, str.length);**

**Arrays.copyOfRange：数组拷贝，指定范围。Arrays.copyOfRange(str,1,3);**

**Arrays.fill：给数组赋值。**

**Arrays.deepToString:当数组中包含数组时，需要用deepToString。**

# 1.10.8 Iterator接口 Enumeration

## 1）两者的函数接口不同

**//Enumeration指定下面的两个方法：**

**//当仍有更多的元素可提取时返回true;当所有元素都被枚举了，则返回false**

**boolean hasMoreElements();**

**//将枚举中的下一个对象做为一个类属 Object的引用而返回**

**Object nextElement() ;**

**//Iterator接口中定义了三个方法:**

**//是否还有下一个元素**

**hasNext();**

**//返回下一个元素**

**next();**

**//删除当前元素**

**remove();**

**//需要注意的是，调用next()方法后才可以调用remove()方法**

**//而且每次调用next()后最多只能调用一次remove()方法,否则抛出IllegalStateException异常**

## 2）Iterator支持fail-fast机制，而Enumeration不支持

**Enumeration 是JDK 1.0添加的接口。使用到它的函数包括Vector、Hashtable等类， Enumeration存在的目的就是为它们提供遍历接口，Enumeration本身并没有支持同步，而在Vector、Hashtable实现Enumeration时，添加了同步。**

**Iterator 是JDK 1.2才添加的接口，它也是为了HashMap、ArrayList等集合提供遍历接口。Iterator是支持fail-fast机制的。**

**fail-fast 机制是指Java集合(Collection)中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。例如：当某一个线程A通过iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出ConcurrentModificationException异常，产生fail-fast事件。**

# 1.10.9 泛型类、泛型方法、泛型接口

## 定义泛型

**Java在接口、类或类的方法的声明中，声明一个泛型。多个参数时要用逗号隔开。**泛型的**声明需要写在接口名、类名之后，方法的返回值之前。**

定义时要注意三点：**1. 泛型的类型参数只能是引用类型，不能是基本类型。   
2. 使用尖括号 <> 声明一个泛型。   
3. <>里可以使用T、E、K、V等字母。这些对编译器来说都是一样的，可以是任意字母。只是程序员习惯在特定情况下用不同字母来区分：**

**T ： Type （类型）**

**E : Element（元素）**

**K ： Key（键）**

**V ： Value（值）**

**包含泛型声明的类型可以在定义变量、创建对象时传入一个类型实参，从而可以动态生成无数个逻辑上的子类，但这种子类在物理上并不存在。**

## 泛型方法

**1）泛型方法的定义**

**修饰符 <声明自定义泛型> 返回值类型 方法名(形参列表) {**

**...**

**}**

## 2）泛型方法要注意的事项

**A. 泛型方法的定义和普通方法定义不同的地方在于，需要在修饰符和返回类型之间加一个泛型类型参数的声明，表明在这个方法作用域中谁才是泛型类型参数。**

**B. 类型参数的作用域**

**class A<T> { ... }中T的作用域就是整个A；**

**public <T> func(...) { ... }中T的作用域就是方法func；**

**类型参数也存在作用域覆盖的问题，可以在一个泛型类、接口中继续定义泛型方法**

**C. 方法中的泛型参数无须显式传入实际类型参数，编译器会根据传入的实参类型自动推断类型参数。**

**例如：<T> void func(T t){ ... }隐式调用object.func("name")，根据”name”的类型String推断出类型参数T的类型是String。当然也可以显式指定，类型参数要写在尖括号中并放在方法名之前，例如：object.<String> func("String")**

**D. 在使用泛型方法时应避免歧义，例如：<T> void func(T t1, T t2){ ... }如果这样调用的话object.func("name", 15);会有很大隐患，T到底应该是String还是Integer存在歧义。**

## 类型通配符

**List<?>传List<int>或者List<String>都可以。**

**List<? extends Number>只能传入Number类型的子类Int或者Double。**

**List<? super Integer>只能传入Inerger的父类Number或者Double等。**

**1.方法参数带有通配符会更加通用；   
2.带有通配符类型的对象，被限制了与泛型相关方法的使用；   
3.上边界通配符：可以使用参数为泛型变量的方法。   
4.下边界通配符：可以使用返回值为泛型变量的方法；**

**5.当使用通配符时，对泛型类中的参数为泛型的方法起到了副作用，不能再使用**

# 1.11.1 流结构体系与分类

## 1.2.1.按流方向分类

**从流的方向上可分为两类(在java中是站在程序角度来区分流的方向，将数据读取到程序中就是输入流；反之，将程序中的数据写出去就是输出流)：**

**- 输入流： 从数据源中将数据读取到程序中的流。**

**- 输出流：程序将数据写入到目的地的流。**

## 1.2.2.按流的数据类型分类

**字节流： 以8位的字节形式来读写的流。他们的标志是名称以Stream结尾。InputStream与OutputStream分别是所有字节输入流与字节输出流的抽象父类。**

**字符流： 以字符形式来读写的流。它们的标志是名称以Reader或者Writer结尾。并且Reader和Writer分别是所有字符输入流与字符输出流的抽象父类。**

**1.Java IO是采用的是装饰模式，即采用处理流来包装节点流的方式，来达到代码通用性。**

**2.处理流和节点流的区分方法，节点流在新建时需要一个数据源（文件、网络）作为参数，而处理流需要一个节点流作为参数。**

**3.处理流的作用就是提高代码通用性，编写代码的便捷性，提高性能。**

**4.节点流都是对应抽象基类的实现类，它们都实现了抽象基类的基础读写方法。其中read（）方法如果返回-1，代表已经读到数据源末尾。**



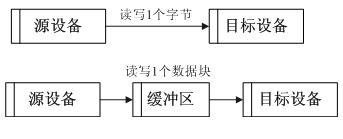
# 1.11.3 使用缓冲流提高性能

**一、缓冲输入流**

**BufferedInputStream extends FileInputStream，缓冲流的设计思想是基于装饰器设计模式的，需要在构造缓冲流的时候传入一个节点流。**

**缓冲输入流为什么能够提高效率？**

**简单一句话就是空间换时间。read方法虽然是一个字节一个字节的返回数据，但是他实际上是一次就读取了buf个字节，然后从buf中返回给read方法，如果取光了，在read一次，如此往复。也就是说，采用了缓冲技术之后，缓冲流调用本地IO的次数变为file.length/buf.length，默认buf是8192个字节，因此read方法的性能提高了约8192倍。当然这不是不需要花费代价的，花费的代价就是多消耗了8191个内存字节。**



**举个简单例子，在A地有10000本书需要搬到B地，如果一次搬1本，需要10000次。如果每次取1000本放到一个货车上，运到B地，需要10次完成。货车相当于是缓存区。同样道理，开设一个数据缓存区每次读取一数据块对于提高读取效率有显著提升。下面用一个具体代码示例来表示二者的性能差别。**

# 1.12.1 多线程概述及作用

**进程：每个进程都有独立的代码和数据空间（进程上下文），进程间的切换会有较大的开销，一个进程包含1--n个线程。（进程是资源分配的最小单位）**

**线程：同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换开销小。（线程是cpu调度的最小单位**

**线程和进程一样分为五个阶段：创建、就绪、运行、阻塞、终止。**

**多进程是指操作系统能同时运行多个任务（程序）**

**多线程是指在同一程序中有多个顺序流在执行。**

**在java中要想实现多线程，有两种手段，一种是继续Thread类，另外一种是实现Runable接口.(**其实准确来讲，应该有三种，还有一种是实现Callable接口**，并与Future、线程池结合使用**

**总结：**

**实现Runnable接口比继承Thread类所具有的优势：**

**1）：适合多个相同的程序代码的线程去处理同一个资源**

**2）：可以避免java中的单继承的限制**

**3）：增加程序的健壮性，代码可以被多个线程共享，代码和数据独立**

**4）：线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类**



**1、新建状态（New）：新创建了一个线程对象。**

**2、就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。**

**3、运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。**

**4、阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：**

**（一）、等待阻塞：运行的线程执行wait()方法，JVM会把该线程放入等待池中。(wait会释放持有的锁)**

**（二）、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。**

**（三）、其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。（注意,sleep是不会释放持有的锁）**

**Thread类的setPriority()和getPriority()方法分别用来设置和获取线程的优先级。**

**每个线程都有默认的优先级。主线程的默认优先级为Thread.NORM\_PRIORITY。**

**线程的优先级有继承关系，比如A线程中创建了B线程，那么B将和A具有相同的优先级。**

**JVM提供了10个线程优先级，但与常见的操作系统都不能很好的映射。如果希望程序能移植到各个操作系统中，应该仅仅使用Thread类有以下三个静态常量作为优先级，这样能保证同样的优先级采用了同样的调度方式。**

**2、线程睡眠：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。**

**3、线程等待：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0) 一样。**

**4、线程让步：Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。**

**5、线程加入：join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。**

**6、线程唤醒：Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。**

# 链表

## 单项链表

**包含信息域和指针域，只可向一个方向遍历，查找一个节点只能从第一个访问到最后一个。**

## 双向链表,也叫双链表

**双向链表中不仅有指向后一个节点的指针，还有指向前一个节点的指针。第一个节点的"前连接"指向NULL，最后一个节点的"后连接"指向NULL。**

**这样可以从任何一个节点访问前一个节点，也可以访问后一个节点，以至整个链表。一般是在需要大批量的另外储存数据在链表中的位置的时候用。**

**由于另外储存了指向链表内容的指针，并且可能会修改相邻的节点，有的时候第一个节点可能会被删除或者在之前添加一个新的节点。这时候就要修改指向首个节点的指针。**

**有一种方便的可以消除这种特殊情况的方法是在最后一个节点之后、第一个节点之前储存一个永远不会被删除或者移动的虚拟节点，形成一个循环链表。这个虚拟节点之后的节点就是真正的第一个节点**