**Java 与 C++ 的区别**

* Java 是纯粹的面向对象语言，所有的对象(除基本类型)都继承自 java.lang.Object，C++ 为了兼容 C 即支持面向对象也支持面向过程。但是因此写算法的时候写起来没有C++方便。
* Java 通过虚拟机从而实现跨平台特性，但是 C++ 依赖于特定的平台，不能很好地屏蔽硬件差异。
* Java 没有指针，它的引用可以理解为安全指针，而 C++ 具有和 C 一样的指针。
* Java 支持自动垃圾回收，而 C++ 需要手动回收。
* Java 不支持多重继承，只能通过实现多个接口来达到相同目的（但是没有提供对接口中出现相同名字函数、变量的解法），而 C++ 支持多重继承。
* Java 不支持操作符重载，虽然可以对两个 String 对象支持加法运算，但是这是语言内置支持的操作，不属于操作符重载，而 C++ 可以。
* Java库里面很多时候函数都会抛出异常，然后要自己写try语句来处理，C++基本很少见。
* Java 的static不同，Java中的static关键字不会影响到变量或者方法的作用域。
* Java没有goto

java四大名著：

java 编程思想

Effective java： <https://www.cnblogs.com/IcanFixIt/p/8082510.html>

深入理解java虚拟机

Java核心技术 卷I、II

很好的博客：<https://blog.csdn.net/javazejian/article/details/72772461>

Oracle 勇于通过代价高昂的法律诉讼维护其知识产权——它与 [Google](http://www.yidianzixun.com/channel/w/google) 围绕 Android 的官司就是证明。那次官司的结果是法律不支持 Java API 被复制或分支（copy/fork），也不支持通过封装或重命名的方式移为他用。Java Community Process 是目前唯一可以改变该语言核心或标准 API 的方式。第三方供应商若想开发 Java 工具并大量发售，必须获得（大多是以购买）Oracle 的许可。Google Android系统借鉴了Oracle JDK的几个Java API，估计是为了统一吧，但是这个导致一直陷入官司。谷歌从 Android N（7.0） 版本开始就不再依赖私有的 Java API 了，而是以开源的 OpenJDK 取代之。2019年开始，Oracle JDK会收费。

java10之前的jdk命名规则java7对应jdk1.7

java10 对应 jdk10

java中建议数据成员命名的时候不要加上m\_，类的构造函数中通过this.value=value来进行赋值。

不推荐将树的节点类中构造方法通过数组直接构造完全二叉树，虽然这么做没有问题（只要一开始将this存放到队列中就好），但是构建树不是节点的任务。不过这样写起来是方便了很多。

idea java gradle项目中获取resources文件夹下文件：

xxxx.class.getClassLoader().getResourceAsStream("dura\_dist.txt"); 获取内容

xxxx.class.getClassLoader().getResource("dura\_dist.txt").getPath(); 获取绝对路径

idea打开gradle项目，要么open-》xxx.gradle不能是项目文件夹，要么import project

0.Java applet：Java在web里的小程序

Android会使用到Java里面常用的库，但是只是这样。所以我就没有怎么学习JavaFX，这个时下流行的java窗口。说C++：用过JNI的人都知道本地的C/C++库，至少要编译多次。不同的框架就代表者不同的编译环境。而且直接用C++编译运行的话。C++并不能很好的屏蔽掉硬件差异。所以对于开发来说调试来说，都是一个比较艰巨的任务(说到底还是编译组的锅)。而且每一个应用一个虚拟机是很好的想法。所以C++没有被选为Android的开发语言。

Java SE 是客户端。Java EE 是服务器端。Java ME是微型版，嵌入式开发。

JavaFX 界面的制作比Java Swing好

Java9 引入jShell，这种做法方便初学者或者来复习一些东西。

0.5当下在航空电子应用中Java程序多数应用于不太重要的系统（认为不安全），我们需要定义Java的标准子集，以适合于在最高安全重要级别下使用：

* 安全至上的Java支持只是标准库的一个很小的子集;
* 安全至上的Java提供了更大范围的实时优先级和更精确地限制了优先级调度程序的行为;

安全至上Java标准草案还增加了直接访问设备的I / O寄存器和实现在Java语言中的第一级中断处理程序的能力。

都是为了避免与更好地处理重要的进程出错

1.

8种基本类型：

* boolean/8
* byte/8
* char/16
* short/16
* int/32
* float/32 1.1f，向下转型是一件危险的事情
* long/64
* double/64 小数常量默认类型

boolean 布尔类型，输出就是True或者Flase。

//以上做法纠正了C里面多余的一种类型，去掉了long long

float 4 小数点后有效位数7-8 后缀是f F

double 8 小数点后有效位数15-17 后缀是d D

/\*没有long double\*/

范围大的类型转换为范围小的类型必须要使用显示强转。

float a=0.2f;

short a =2; //这个是例外

以下代码对于基本类型返回的是对应的包装类：

public static String getType(Object o){ //获取变量类型方法  
 return o.getClass().toString(); //使用int类型的getClass()方法  
}

数组常量不能作为参数传递，因为数组常量根本不能直接被识别为一个对象，就连上面判断类型这个函数都不能使用，需要走一步变量声明来构造数组类型对象。function(new int[]{1,2,3,4,5})

java enum变量： 适合实现单例模式

public enum Size{ SMALL, MEDIUM, LARGE, EXTRA\_LARGE };

如果要自定义自己的enum，就必须要自己编写好构造函数并且实例化好序列:

public enum WeekDay {

Mon("Monday"), Tue("Tuesday");

private final String day;

private WeekDay(String day) {

this.day = day;

}

}

反编译相当于

class WeekDay extends Enum{

public static final WeekDay Mon;

public static final WeekDay Tue;

}

对照C语言中的enum变量：enum Test {a,b}test=a;

package com.company;  
  
/\*  
\* 八数码问题  
\* \*/  
  
public class Main{  
 public enum Week {  
 //定义枚举类对象  
 *Monday*("MON","星期一"),  
 *Tuesday*("TUE","星期二"),  
 *Wednesday*("WED","星期三"),  
 *Thursday*("THU","星期四"),  
 *Friday*("FRI","星期五"),  
 *Saturday*("SAT","星期六"){  
 @Override  
 public boolean isRest(){  
 return true;  
 }  
 },  
 *Sunday*("SUN","星期日"){  
 @Override  
 public boolean isRest(){  
 return true;  
 }  
 };  
  
 private String abbreviation = "";//缩写  
 private String chineseName = "";//中文名字  
  
  
  
 //定义自己的构造器 非仅仅一个int属性，构造函数是必须的，注意是private  
 private Week(String abbreviation,String chineseName){  
 this.abbreviation = abbreviation;  
 this.chineseName = chineseName;  
 }  
  
 public String abbreviation(){  
 return abbreviation;  
 }  
  
 public String getChineseName(){  
 return chineseName;  
 }  
  
 //周六和周日应该返回true，此方法在周六和周日的值中被重载  
 public boolean isRest(){  
 return false;  
 }  
  
 //重载，对它进行稍稍的改动  
 @Override  
 public String toString(){  
 return this.getClass().getName()+"."+this.name();  
 }  
   
  
 //你会发现main函数并不是直接在Main类中，而是在Enum类中，也是可以的  
 public static void main(String[] args) {  
 for(Week week:Week.*values*()){  
 System.*out*.println("-----------------------------------------");  
 System.*out*.println("ordinal():"+week.ordinal());  
 System.*out*.println("name():"+week.name());  
 System.*out*.println("getChineseName():"+week.getChineseName());  
 System.*out*.println("abbreviation():"+week.abbreviation());  
 System.*out*.println("isRest():"+week.isRest());  
 System.*out*.println("toString():"+week.toString());//如果此方法没有被重载，与name()功能相同.  
 System.*out*.println("getDeclaringClass():"+week.getDeclaringClass());  
 }  
 }  
}

enum Type{

A{

public String getType() {

return "I will not tell you";

}

},B,C,D;

static int value;

public static int getValue() {

return value;

}

String type;

public String getType() {

return type;

}

}

EnumSet基于bitmap，Enum的Set。

模板实例化不能适应基本类型

通过byte[]的封装实现C语言里面的Union。

>> << 有符号位移

>>> 无符号位移（不存在对应的有符号左移指令）

switch中case后变量支持的类型为int,String（jdk1.7开始支持）,enum型的。所以连long类型也是不支持的。对照关系：C语言里面的case也不支持long类型。

[PO BO VO DTO POJO DAO DO这些Java中的概念分别指一些什么？](https://www.zhihu.com/question/39651928" \t "/home/vega/Documents\\x/_blank)

<https://blog.csdn.net/uestcyms/article/details/80244407>

PO(Persistant Object) 用于持久化数据，不包含数据操作

BO(Business Object) 主要作用是把业务逻辑封装为一个对象。这个对象可以包括一个或多个其它的类对象。

VO(Value Object) 应用界面显示需要的对象，对于Android而言即是activity或view中的数据元素。

DTO(Data Transfer Object)  用于数据库操作的对象

POJO(Plain ordinary java object) 简单JavaBeans对象，带有部分属性的getter/setter 方法的标准java类 一个POJO持久化以后就是PO；直接用它传递、传递过程中就是DTO；直接用来对应表示层就是VO。

JSP中的JavaBeans的设计原则：

　　（1）公有类

　　（2）无参的公有构造方法

　　（3）属性私有

（4）getter和setter方法

在J2EE里，Enterprise Java Beans(EJB)称为Java 企业Bean，是Java的核心代码

不过一般Beans指所有的类

1.5. UTF-16be 中的 be 指的是 Big Endian，也就是大端。相应地也有 UTF-16le，le 指的是 Little Endian，也就是小端。Java 使用双字节编码 UTF-16be。

 UTF-8 编码中，中文字符占 3 个字节，英文字符占 1 个字节；

 UTF-16be 编码中，中文字符和英文字符都占 2 个字节。

1.6. try 后面不一定跟着catch，见过try-finally组合（估计写的是伪代码）？但是没有catch，IDEA会报错

2.java的import java.xxxx导入类就好比C++里面的#inlcude<头文件> using xxxxx一样，之后的使用可以简化名字，否则要使用全名。import 使用通配符就好比C++ using了命名空间一样。java包的思想比C++命名空间好多了，前者import只需要一步就好，就可以满足解析的需求。

java会默认导入java.lang和当前java所在文件夹的包

3.java从键盘缓冲区读取输入：

/\*在idea中，在主函数中写一个Scanner，然后就会自动加上对应的import\*/

import java.util.Scanner; //这是一个用于扫描输入文本的类

public class ComputerAverage { //类名的命名规则

public static void main(String[] args) { //args.length 基本遇到的都是length函数，这里却是成员

//Integer.parseInt(args[0])+1

//void main通过System.exit(flag ? 1:0);返回不同的结果值，不能通过return

Scanner input =new Scanner(System.in);//不要同时声明多个这个，又没有锁。扫描输入流

System.out.print("Enter three number");//和System.in一样，估计是个对象

double number1 =input.nextDouble();//方法是从第二个单词开始大写

double number2 =input.nextDouble();

double number3 =input.nextDouble();

double average =(number1+number2+number3)/3;

System.out.println("The average of three number is"+average);//默认就是String类型，而不是char\*，所以可以直接+。

if(args.length!=2) { //不是4

System.out.println("Usage: java Exercise12\_11 John filename");

System.exit(1);

}

}

}

对应java基本类型，有以下方法：

.nextByte()

.nextShort()

.nextInt()

.nextLong()

.nextFloat()

.nextDouble()

.nextBoolean() 输入true、false

.nextLine() ACM有的题目输入是按照行为单位的

nextint只获取数字，后门还有一个换行符。再执行到nextline时，换行符给了nextline。所以nextline里是一个换行符，没等你输入他就已经有内容了(换行符)。

4.使用final来修饰这是一个变量设定为常量！如果修饰一个类表示这个类不能被继承。修饰方法表示该方法不会被覆盖。即做出不会被改写的。

final 变量不同于C++ const变量，在声明的时候赋值，或者在构造函数中赋值一次。

final 内存语义：1）在构造函数内对一个final域（一般变量或者引用变量）的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。2）初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。毕竟不是static，但是这种不可变常量通常会被设置为static。



性质一保证了 线程A的构建与线程B的使用之间是顺序的，性质二保证reader()中两次读取是顺序的。在引用变量为任意线程可见之前，该引用变量指向的对象的final域已经在构造函数中被正确初始化过了。其实，要得到这个效果，还需要一个保证：在构造函数内部，不能让这个被构造对象的引用为其他线程所见，也就是对象引用不能在构造函数中“逸出”：

public class FinalReferenceEscapeExample {

　　final int i;

　　static FinalReferenceEscapeExample obj;

　　public FinalReferenceEscapeExample () {

　　　　i = 1; // 1写final域

　　　　obj = this; // 2 this引用在此"逸出"

　　}

　　public static void writer() {

　　　　new FinalReferenceEscapeExample ();

　　}

　　public static void reader() {

　　　　if (obj != null) { // 3

　　　　　　int temp = obj.i; // 4

　　　　}

　　}

}

编译器会在构造函数的返回之前，插入一个StoreStore屏障（我认为会对所有变量生效），从而禁止将final域的写重排序到构造函数之外。这同时说明一件事情：一般的变量赋值可能会被重排序到构造函数的外面。如果另外的线程引用对象的话，final变量可以保证可见性，但是一般变量不一定（需要volatile修饰）。

所以加上一个final变量，可以解决多线程中的构造函数问题。

但是在X86处理器中，这些屏障都会被屏蔽掉，因为X86处理器仅支持写-读操作。

4.1Math.PI 对pi的较长的一个static final double 定义。

4.2对于数字常量，进制的设置在前面：

这倒是和C格式上一样，但是不同汇编和一般写法。

0B11111111

0377

0x(0X)EE

6.标识符（命名规则）：

由字母、数字、下划线(\_)和美元符号($)构成的字符序列，而且不以数字开头，书上写着是任意长度。

6.5返回系统分隔符：

System.getProperty("line.separator"); 底层猜测使用map实现

7.伪随机数：

java.lang.Random;

UML

+Random()

+Random(seed: long) //使用前记得构造对象

+nextInt(): int

+nextInt(n: int): int //返回0与n之间（不包含n）的随机数 //n为负数，抛出无效参数异常java.lang.llegalArgumentException

+nextLong():long

+nextFloat(): float

+nextDouble():double

+nextBoolean(): boolean

+nextBytes(Byte[] ) //居然可以直接创建一个随机数组

Math.random()返回一个[0,1)小数。所以random\*101加上向下转换为int类型 就是得到一个[0,100]之间的整数。'a'+Math.random\*('z'-'a'+1)得到一个小写字母随机数，联想到自己产生随机验证码的写法。

System.currentTimeMillis() //话说形容词在time后面，java中的随机数不需要初始化种子

9.Math类在java.lang包里面，而这个包是自动导入的。

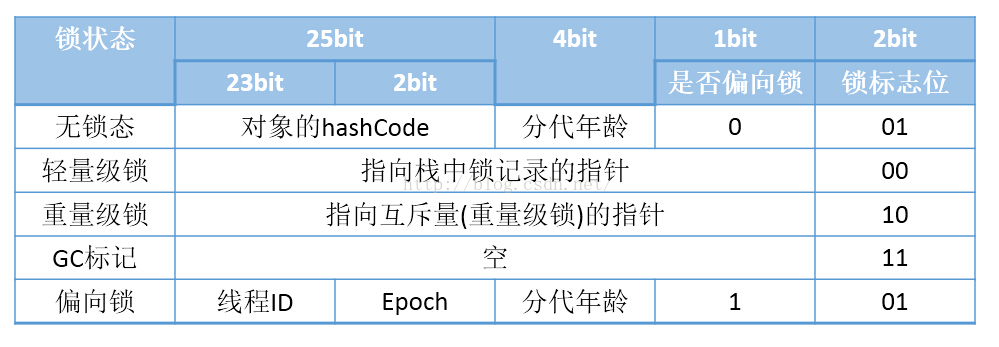
三角函数(静态方法)：

输入弧度sin() cos() tan() asin()

acos() atan() toDegrees()

输入角度转换为弧度toRadians()

10.5. 对象头（Mark Word）本身是8的倍数，当对象的实例变量数据不是8的倍数，便需要填充数据来保证8字节的对齐。下图中显示了5种类型32bits的Mark Word。



hashCode的存储长度只有25bits

偏向锁，估计就是因为太轻量级了，所以对象头中还有“分代年龄”。

锁是存在Java对象头里的。每一个对象都可以使用Sychronized，所以每一个对象都有一个对应的monitor 对象，有人说对象头里面存储了指向的引用。monitor 监视一段代码 ，防止多次进入。

11.Character类的静态方法： char是基本类型而不是类，所以按照java的思想就需要一个新的类来装载这些方法。

java.lang.Character 所以可以直接使用

isDight()

isLetter()

isLetterOrDight()

isLowerCase()

isUpperCase()

使用举例：

System.out.println(""+isUpperCase('a'));

12.静态方法也可以使用对象来调用（Java里面声明一个对象就可以了）。Java里面直接调用是类名.静态方法，C++中间是::。

13.String类： 计算引用指向的值大小的时候注意每一个字符都是两字节

"xxx"包括的就是String类的一个对象

""是空串

通过构造函数将一个char[]转为String，那么这个char[]有多大，转换出来的String就会有多长。

信息获取：

.length()

.charAt(index) //代替索引 并且char可以和string相加 不存在[]

判断：

.equals(s1)

.equalsIgnoreCase(s1)

.compareTo(s1) //String类型，内容的比较要用equals、compareTo函数（返回Boolean），不能用==。Byte.equals方法结果为true的前提是比较双方均为Byte类型。compare 比较的才是值。对于大部分对象，equals方法其实就会返回==的结果，不过String重写了equals

.compareToIgnoreCase(s1)

.startWith(s1)

.endsWith(s1)

.contains(s1) //字串检测

转换和结合：

.concat(s1)

.toUpperCase()

.toLowerCase()

.trim() //返回一个新字符，去掉两边的空白字符。

子串获取： //这里面第二个s真的是小写

.substring(beginIndex)

.substring(beginIndex, endIndex)

搜索：

.indexOf(ch) //没找到返回-1。这些返回值都是int，所以不要是使用short去接受、

.indexOf(ch, fromIndex) //fromIndex之后出现的第一个

.indexOf(str)

.indexOf(str, fromIndex)

.lastIndexOf(ch)

.lastIndexOf(ch, fromIndex) //fromIndex之前的最后一个

.lastIndexOf(str)

.lastIndexOf(str, fromIndex)

替换： 创建一个新的字符串对象

replace()

replaceAll() 正则表达式 替换所有

replaceFirst() 正则表达式 但是只替换一个

划分：

split(string)

正则匹配：

.matches("Java")

.matches(".\*")

\\d表示数字，\\d{3}表示3个连续数字（看来会单独对字符串先进行一次转义）

.匹配除“\r\n”之外的任何单个字符。要匹配包括“\r\n”在内的任何字符，请使用像“[\s\S]”的模式。

"abc def".replaceFirst("(\\w+)\\s+(\\w+)", "$2 $1"); //结果为 def abc

"abc def aaa bbb".replaceAll("(\\w+)\\s+(\\w+)", "$2 $1"); //结果是 def abc bbb aaa

一般的email,形如zhangshan@163.com,abc@sina.com.cn这样一些常用的形式就行了,但是在我们公司的一些客户中邮箱却有一些zhangshna.Mr@163.com,abc\_Wang.dd@sian.com,abc\_Wang.dd.cc@sian.com这 种类似的形式,在@符号之前还有点.,原来是拿来就用,可是现在就不行,得自己研究正则的用法了

原来的正则表达式/^[a-zA-Z0-9\_-]+@[a-zA-Z0-9\_-]+(\.[a-zA-Z0-9\_-]+)+$/; 可以改为

/^(\w)+(\.\w+)\*@(\w)+((\.\w{2,3}){1,3})$/;或者/^(\w)+(\.\w+)\*@(\w)+((\.\w+)+)$/;

\w就是[a-zA-Z0-9\_]还不是很对，少了一个字符“-”

与数据：

.toCharArray()

静态方法初始化：(使用数组构造String类)

valueOf(new char[] {'J', 'a', 'v', 'a'}) 好像可以直接通过构造函数借助字符数组构造

数据类型转化为String：

基本类型使用""+数值

包装类使用toString()

还可以使用更加进一步的格式化数据类型方法：format() 。似乎C++的stringstream有这种，在c里面是sprintf函数。

15.三种print输出函数：都是System类的方法：

.print()

.println() //ln是line，也就是输出以后换行

.printf() //f是format，格式化输出

System.out.printf("输出一个浮点数：%f，一个整数：%d，一个字符串：%s",d,i,s);

System.out.printf("字符串：%2$s，%1$d的十六进制数：%1$#x",i,s);

17.被重载的方法必须具有不同的参数列表。不能基于不同修饰符或返回值类型来重载方法。其实能够类型推断的话，返回值也是可以的，虽然不明白为什么不这么做。

18.编程格式：

public class RandomCharacter {

private char getRandomCharacter(char ch1, char ch2) {

if( ch1 > ch2 )

ch1^=ch2^=ch1^=ch2;

return (char)(ch1+Math.random\*(ch2-ch1+1));

}

public char getRandomLowerCaseLetter() {

return getRandomCharacter('a', 'z');

}

public char getRandomUpperCaseLetter() {

return getRandomCharacter('A', 'Z');

}

}

//但是ch1会变成0，ch2能获取ch1的值

估计是因为jvm的优化，使用“java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly Main” 查看汇编代码，我查xor指令只查到对自己的置零异或。

14.数组的声明：elementType[] arrayRefer;

二维数组的声明：elementType[][] arrayRefer;不建议使用elementType arrayRefer[][];

使用new elementType[arraySize]来构建数组，然后传过去给arrayRefer。数组的名字在Java里面是种引用。

只能int[][] test =new int[2][2]; 不能int[2][2] test;

15.Java中默认数据类型初始化为0，char类型初始化为'\u0000'，boolean初始化为flase。引用类型会被默认初始化为null（String和数组）。在if中都能被转化为false。 16.Java里面的foreach： for(elementType element:xxxx) {...} 同C++

或者使用forEach方法

17.复制数组有四种方法：

1.使用循环对元素进行复制

2.System.arraycopy 静态方法 arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length)

这个方法的命名目前唯一违反了Java规则

通过这种复制就不需要多走一步引用过：

//下面代码考虑到了两个数组重合在一起的情况，经过下面这样考虑，之后可以保证to数组是含有完整的数据，但是from可能会被破坏

void \_Copy\_conjoint\_jints\_atomic(jint\* from, jint\* to, size\_t count) {

if (from > to) {

jint \*end = from + count;

while (from < end)

\*(to++) = \*(from++);

}

else if (from < to) {

jint \*end = from;

from += count - 1;

to += count - 1;

while (from >= end)

\*(to--) = \*(from--);

}

}

3..clone方法（类的克隆）

4.序列化与反序列

20.Java可变参数列表：类型名...变量名

要求：只能表达同类型，而且必须要是最后一个参数（所以一个函数形参里面只能有一个这个）

public static void printMax(double...numbers) {

if(number.length == 0) {

System.out.println("No argument passed");

return;

}

xxxxx;//当做一个数组一样使用

}

20.5 比较器

java.util.Comparable 接口中只提供了一个方法： compareTo(Object obj) ，该方法的返回值是 int 。sort函数比较的如果是对象的话，会调用compareTo。

java.util.Comparator 是比较器，sort函数通常支持传递一个比较器对象作为第二个参数。Comparator 接口中定义了两个方法： compare(Object o1, Object o2) 和 equals 方法，由于 equals 方法所有对象都有的方法，因此当我们实现 Comparator 接口时，我们只需重写 compare 方法，而不需重写 equals 方法。Comparator 接口中对重写 equals 方法的描述是：“注意，不重写 Object.equals(Object) 方法总是安全的。然而，在某些情况下，重写此方法可以允许程序确定两个不同的 Comparator 是否强行实施了相同的排序，从而提高性能。”。

21.java.util.Arrays类包含一些对数组进行操作的静态方法：

.sort(arrayRefer) //两个参数的话，第二个参数传入比较器对象

.sort(arrayRefer, start, one\_after\_end) 后面两个参数是索引号

.equals(arrayRefer1, arrayRefer2)

.fill(arrayRefer, 5)

.fill(arrayRefer, start, one\_after\_end, fill\_element)

.asList(arrayRefer)



.length 就可以知道长度，是一个public成员变量

int[] getClass().toString()输出是[I

Integer[] arrayInt ={1,2,3,4,5}; //如果是int类型数组，下面一行代码会出问题

ArrayList<Integer> list =new ArrayList<>(Arrays.asList(arrayInt) ); //<>内不能使用基本类型。int是基本类型，Integer是整数类

打印输出一个ArrayList，按照数组形式输出每一个元素的hashcode函数返回值： [Person@16acdd1, Person@ee6681, Person@18bbc5a]

24.java.util.Date类 推荐joda.DataTime

UML

+Date() //给当前时间创建一个对象

+Date(elapseTime: long) //为指定时间创建一个data对象。

+toString() :String

+getTime() : long

+setTime(elaplseTime: long):void //给对象设置一个新的时间

时间的设置都是1970年1月1号开始的毫秒数。

这个类既有setTime又有getTime，已经可以直接给出数据域了。

使用java.util.Date date =new java.util.Date();

Date初始化中第二个参数月份从0开始

日期： Calendar抽象类

Calendar c = Calendar.getInstance(); //创建子类GregorianCalendar对象

c.add(c.DATE,1);

int day = c.get(Calendar.DAY\_OF\_MONTH);

使用Calendar之后还继续封装成自己想要的时间类，一般名字叫做DateUtils。org.apache.commons.lang.time.DateUtils这个是很不错的一种实现。

GregorianCalendar是 Calendar 的一个具体子类，提供了标准阳历。Calendar的getInstance( )方法返回用默认的地区和时区的当前日期和当前时间所初始化的GregorianCalendar（标准日历）相当于new GregorianCalendar()；

Calendar c = new GregorianCalendar();相当于使用了Calendar.getInstance() System.out.println(c); 将一个类的数据成员名字和值全部显示出来了。

GregorianCalendar firstFlight = new GregorianCalendar(1903, Calendar.DECEMBER, 17);

GregorianCalendar firstFlight = new GregorianCalendar(1903, 11, 17);

最好用的是JodaTIme

25.UML中静态方法使用下划线。如果是-减号开头就表示是private。

26.5 最简单的jfx程序

import javafx.application.Application;

import javafx.stage.Stage;

public class Test extends Application {

public void start(Stage primaryStage) {

primaryStage.show(); //舞台会展示一个空窗口界面

}

public static void main(String[] args) {

// TODO Auto-generated method stub

Application.launch(args);

}

}

http://www.java2s.com/Tutorials/Java/JavaFX/0340\_\_JavaFX\_GridPane.htm

BurpSuite 使用java开发

26.5 Pane类型可以add Pane类型上：

下面的程序会画四个风扇

import javafx.application.Application;

import javafx.stage.Stage;

import javafx.scene.Scene;

import javafx.scene.layout.GridPane;

import javafx.scene.layout.StackPane;

import javafx.scene.layout.Pane;

import javafx.scene.shape.Circle;

import javafx.scene.shape.Arc;

import javafx.scene.shape.ArcType;

import javafx.scene.paint.Color;

import javafx.geometry.Insets;

public class Test extends Application {

@Override // Override the start method in the Application class

public void start(Stage primaryStage) {

// Create a GridPane and set its properties

GridPane gridPane = new GridPane();

gridPane.setPadding(new Insets(10, 10, 10, 10));

gridPane.setHgap(10);

gridPane.setVgap(10);

// Place nodes in the pane

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

// Create a stack pane

StackPane stackPane = new StackPane();

// Add circle to stack pane

Circle circle = getCircle();

stackPane.getChildren().add(circle);

// Add Arcs to stack pane

getArcs(stackPane);

gridPane.add(stackPane, i, j);

}

}

// Create a scene and place in in the stage

Scene scene = new Scene(gridPane);

primaryStage.setTitle("Exercise\_14\_09"); // Set the stage title

primaryStage.setScene(scene); // Place the scene in the stage

primaryStage.show(); // Display the stage

}

/\*\* Add four arcs to a pane and place them in a stack pane \*/

private void getArcs(StackPane stackPane) {

double angle = 30; // Start angle

for (int i = 0; i < 4; i++) {

Pane pane = new Pane();

Arc arc = new Arc(100, 100, 80, 80, angle + 90, 35);

arc.setFill(Color.BLACK);

arc.setType(ArcType.ROUND);

pane.getChildren().add(arc);

stackPane.getChildren().add(pane);

angle += 90;

}

}

/\*\* Return a Circle \*/

private Circle getCircle() {

Circle c = new Circle();

c.setRadius(100);

c.setStroke(Color.BLACK);

c.setFill(Color.WHITE);

return c;

}

}

27.javafx的geometry包中有一个point2D类（这名字取得真好）

UML

+point2D(x:double y:double)

+distance(x:double y:double):double

+distance(p:point2D ):double

+getX(): double

+getY(): double

+toString(): String//返回改点的字符串表示

import javafx.geometry.Point2D

30.

package p1;

class C1 { //不具备公共性

}

package p2;

class C2 {

}

31.java安装了jdk，然后就会把jre的安装文件放在安装目录下叫做jre.msi，安装以后就会将jre安装到和安装目录同一个父目录下。不过jre其实不需要安装了，因为jdk里面其实有jre的。

具备开发功能的jdk自己的jre下才会同时有client性质的jvm和server性质的 jvm，而仅仅作为运行环境的jre下只需要client性质的jvm.dll就够了。

32.不可变类是指没有提供更改方法用于对数据域进行更改的类。

33.javac是将.java-》.class, 然后java class 类名就好。

C:\Users\Administrator>javac main.java

main.java:1: 错误: 类TestInitialize是公共的, 应在名为 TestInitialize.java 的文件

中声明

public class TestInitialize {

^

1 个错误

以上错误说明java的源程序文件名是有要求的，就是类名。类名和文件名大小写无所谓？不行，类名要与文件名完全一样。不过windows的识别是无视大小写的，所以可以继续javac 全小写.java。小写的，cmd会处理了以后正常的文件名传给java。

36.将数值类型转换为包装类对象的过程称为装箱（boxing），相反的转换过程称为开箱（unboxing）。比较、赋值等都是可以自动完成装箱与开箱：

Integer i = 10; //装箱

int n = i; //拆箱

//import java.lang.Double; .lang与当前文件夹中类是自动包含

Boolean、Character、Double、Float、Byte、Short、Integer和Long这些都是包装类名字，即使名字的全称。

初始化：

可以采用字符串初始化（例如 使用"5"初始化Interger），也可以使用对应的数值类型初始化

转换：类似数值类型的自动转换，返回数值类型

doubleValue()

floatVlaue()

longValue()

intValue()

shortValue()

boolValue()

toString()

compareTo()

静态方法：

初始化

valueOf(str)//这个函数其实没有什么用

valueOf(str, radix) 比如Double.valueOf("1.1111", 2) 。当然这是错误的，进制转换只有在整数下。因为小数可能是不尽的。

valueOf(1) 等等

以下静态方法适用于将字符串转化为对应的数值类型：；

Interger 是parseInt(str) 或者 parseInt(str, radix) 这个返回int

Java里面不使用静态方法没法构建临时变量，因为构造函数前面需要使用new。

37.大数类（不可变imutable，没有这个关键词）：

import java.math.\*;

BigInteger和BigDecimal

可以使用字符串来进行初始化

方法：

add()

subtract()

multiple()

divide()

remainder() //取余

由于没有对BigDemical类对象精度进行限制，所以如果结果不能终止，除法方法就会抛出ArithmeticException异常。所以使用devide()没住需要传递三个参数使用重载方法，第二个参数指定精度第三个参数指定舍入方式。比如所a.divide(b, 20, BigDemical.ROUND\_UP); a和b都是BigDemical对象。

Integer 缓存池：

在 Java 8 中，Integer 缓存池的大小默认为 -128~127

new Integer(123) 每次都会新建一个对象

Integer.valueOf(123) 会使用缓存池中的对象，多次调用会取得同一个对象的引用。编译器会在自动装箱过程调用 valueOf() 方法，比如Integer m = 123;

38.Unsafe中存在java提供的对内存操作函数，包括内存分配函数。但是这个库会在之后的版本中被删除，认为不安全。

39.次类 子类 扩展类 派生类

超类 父类 基类

40.java类只能继承一个父类，但是可以实现多个接口。

41.所有的类都继承自java.lang.Object，包括java.lang.Class。基于Object实现的多态是很强的，Object数组，列表。

Object对象默认提供的10种方法：除了clone与finalize是protected其余都public

public final native Class<?> getClass()

public native int hashCode()

jdk根据对象的地址或者字符串或者数字算出来的int类型的数值，map的话是key与value的hashCode异或。

public boolean equals(Object obj)

protected native Object clone() throws CloneNotSupportedException

使用protected，一个类没有显示重写这个方法其他类就不能够调用。如果要通知目标自己的clone可以被调用，应该implements Cloneable（作为一个标识接口不提供任何方法）。

clone的时候只是创建的对象地址不同，但是对象里面的属性引用到的还是同一块内存，所以深度克隆是指在clone函数中通过super.clone创建一个对象之后，还通过new等重新创建内部属性。

除了使用clone函数，还可通过序列化复制对象。

public String toString()

public String toString() {

return getClass().getName() + "@" +  Integer.toHexString(hashCode());

}

以下方法需要在synchronized 同步关键字所限定的作用域中调用，否则会报错java.lang.IllegalMonitorStateException

public final native void notify()

public final native void notifyAll()

public final native void wait(long timeout) throws InterruptedException

public final void wait(long timeout, int nanos) throws InterruptedException

public final void wait() throws InterruptedException

wait()方法释放该对象的monitor（管程）并进入休眠状态，直到其他线程调用该对象的notify()或者notifyAll()使之再次获取该对象的monitor。所以使用这些方法需要在synchronized  下，不然得不到monitor对象。



notify()或notifyAll()方法调用后，等待线程依旧不会从wait()返回，需要调用notify()或notifyAll()的线程释放锁之后，等待线程才有机会从wait()返回。notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中（使用monitor所以不是随机唤醒，但是进入等待队列的先后说不准），而notifyAll()方法则是将等待队列中所有的线程全部移到同步队列。

与条件变量的关系：

而条件变量是ReentrantLock的实现基于内部一个State变量以及两种队列，是单独的。

protected void finalize() throws Throwable {}

流程：当对象变成(GC Roots)不可达时，GC会判断该对象是否覆盖了finalize方法，若未覆盖，则直接将其回收。否则，若对象未执行过finalize方法，将其放入F-Queue队列，由一低优先级线程执行该队列中对象的finalize方法。执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”

子类可以覆盖该方法以实现资源清理工作，GC在回收对象之前调用该方法。C++中的析构函数调用的时机是确定的（对象离开作用域或delete掉），但Java中的finalize的调用具有不确定性。

finalize方法被低优先级线程执行，于是可能导致性能问题。

对象再生问题：finalize方法中，可将待回收对象赋值给GC Roots可达的对象引用，从而达到对象再生的目的

System.gc()与System.runFinalization()方法增加了finalize方法执行的机会，但不可盲目依赖它们

41.1. 返回一个对象的“地址”

System.identityHashCode(Object) 系统方法

Object.hashcode() // 不能说作为地址，但是在hash的时候可能会使用这个作为hash中的key

所以有的人建议在重写equals函数之后，也应该重写hashCode函数，因为如果在hashmap等中使用，会优先比较hashcode的值，再调用equals函数。

42. 因为使用的是public class声明，所以类名称应该与文件名称完全一致，即应该使用"HelloDemo.java"表示类的名称。

一个源文件中只能有一个public类。

一个源文件可以有多个非public类。

源文件的名称应该和public类的类名保持一致。例如：源文件中public类的类名是Employee，那么源文件应该命名为Employee.java。

42.5 类中东西的顺序： static（父static->子static）语句块->父其余->子其余

父类的 static 语句块和 static 成员变量

子类的 static 语句块和 static 成员变量

父类的非static 语句块和 非 static 成员变量

父类的构造方法

子类的 非 static 语句块和 非 static 成员变量

子类的构造方法

43.

public boolean equals(Object o) {

if(o instanceof Myclass)

return num == ((Myclass)o).num;

}

会被检测出来缺少return，毕竟万一条件不满足呢？！

43.

public class Myclass extends Farther {

Myclass(int a) { super(a); }

public boolean equals(Object o) {

if(o instanceof Myclass) //传入farther对象，这里判断是错误。所以不同于形参是Farther类的对象了

return num == ((Myclass)o).num;

else

return true;

}

}

伪代码实现instanceof:

//尝试类型转换，如果不对jvm会抛出异常，捕获异常，然后将result变量作为结果返回

boolean result;

if (obj == null) {

result = false;

} else {

try {

T temp = (T) obj; // checkcast

result = true;

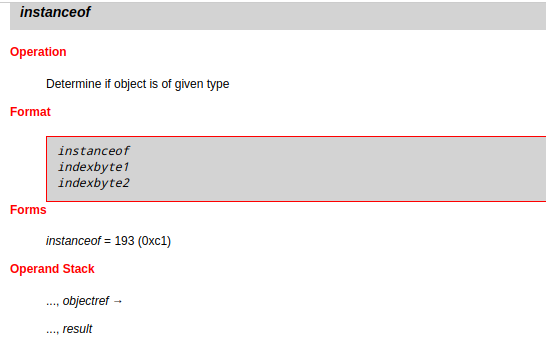
} catch (ClassCastException e) {

result = false;

}

}

JVM有一条名为 instanceof 的指令，而Java源码编译到Class文件时会把Java语言中的 instanceof 运算符映射到JVM的 instanceof 指令上



对indexbyte1和indexbyte2构造的常量池索引进行解析，然后根据java规范判断解析的类是不是objectref的一个实例，最后在栈顶写入结果。（或许应该将这个当做函数调用）

44.java.util.ArrayList<E>

ArrayList是种泛型类，就是C++中的模板类。不传入大小，默认初始空间是10。

ArrayList 基于动态数组实现（连续存储），序列化的时候会将数组的size以及capacity都转为二进制数据。添加元素时使用 ensureCapacityInternal() 方法来保证容量足够，如果不够时，需要使用 grow() 方法进行扩容，新容量的大小为 oldCapacity + (oldCapacity >> 1)，也就是旧容量的大约 1.5 倍。

因为底层是数组，所以向中间插入/删除元素的代价都是很高的（主要调用 System.arraycopy() ）。如果使用数组链表，这时候即便是插入或者删除一个元素的代价都是O(1) 。

存储String等这种长度不定的对象（虽然String是不可变类），很明显存储的是引用。

modCount（不少类内部都实现了这个变量，用于fast-fail） 用来记录 ArrayList 结构发生变化的次数。结构发生变化是指添加或者删除至少一个元素的所有操作，在进行序列化或者迭代等操作时，需要比较操作前后 modCount 是否改变，如果改变了需要抛出 ConcurrentModificationException。其他类中也有。

private transient Object[] elementData; 因为这个是一个动态扩展的数组，所以实现了自己的writeObject()、readObject()，使得可以只序列化数组中有内容的那部分数据。

ArrayList<elementType> list =new ArrayList<elementType>(); =>JDK1.7开始 ArrayList<elementType> list =new ArrayList<>();

UML

+ArrayList()

+add(o: E):void

+add(index: int, o: E):void 没想到不是叫做insert

+clear():void

+contain(o: E): boolean

+get(index: int):E

+indexOf(o: Object): int

+isEmpty():boolean

+lastIndexOf(o: object):int

+remove(o:Object):boolean

+remove(index:int):boolean

+set(index:int o:E) :E

线程安全的ArrayList:

（1）Vector 是线程安全版ArrayList：

* Vector 是同步的，因此开销就比 ArrayList 要大，访问速度更慢。最好使用 ArrayList 而不是 Vector，因为同步操作完全可以由程序员自己来控制；
* Vector 每次扩容请求其大小的 2 倍空间，而 ArrayList 是 1.5 倍。

（2）

List<String> list = new ArrayList<>();

List<String> synList = Collections.synchronizedList(list);

（3）

concurrent 并发包下的 CopyOnWriteArrayList 类**，**先copy出一个容器(可以简称副本)，再往新的容器里添加这个新的数据，最后把新的容器的引用地址赋值给了之前那个旧的的容器地址，但是在添加这个数据的期间，其他线程如果要去读取数据，仍然是读取到旧的容器里的数据。**可见性，是指当一条线程修改了共享变量的值，新值对于其他线程来说是可以立即得知的。很显然，上述的例子中是没有办法做到内存可见性的。**

Android 使用的Java 库是Apache的Harmony, 与官方Java库接口相同，里面实现不一样。Android写方法里面使用**synchronized**修饰，Java 使用**ReentrantLock** 来构建写时锁，因为复制了多份，所以在一个线程写的时候不影响其他线程的读。

CopyOnWriteArrayList 在大量写的情况下，因为多次使用System.arraycopy，反而性能不如（1）（2）。在少量写的时候，因为通过各自复制一份的手段允许写的时候读取内容，所以性能三者最好。也不适用于内存紧张的时候。

45.泛型类

命名类型参数

泛型类的类型要求使用对象，但是基本类型不是对象，所以不能以基本类型作为泛型类的元素类型。这也是包装类的一个意义。但是有的类型里面存放null，比如那些存储位置和具体值有关的类型。

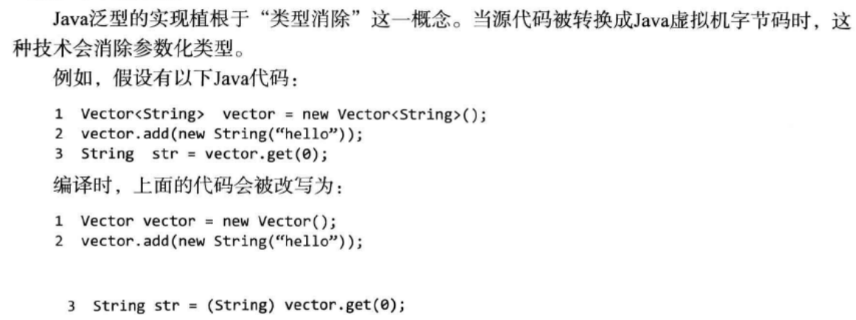
推荐的命名约定是使用大写的单个字母名称作为类型参数。这与 C++ 约定有所不同，并反映了大多数泛型类将具有少量类型参数的假定。对于常见的泛型模式，推荐的名称是：

K —— 键，比如映射的键。   
 V —— 值类型，比如 List 和 Set 的内容，或者 Map 中的值。   
 E —— Element。   
 T —— java类型。 ？ —— 不确定类型，所以代码中用Object。

public interface Map<K, V> {  
 public void put(K key, V value);  
 public V get(K key);  
}

public class Lhist<V> {   
 public Lhist(int capacity) { ... }  
 public int size() { ... }  
 public void add(V value) { ... }  
 public void remove(V value) { ... }  
 public V get(int index) { ... }  
}

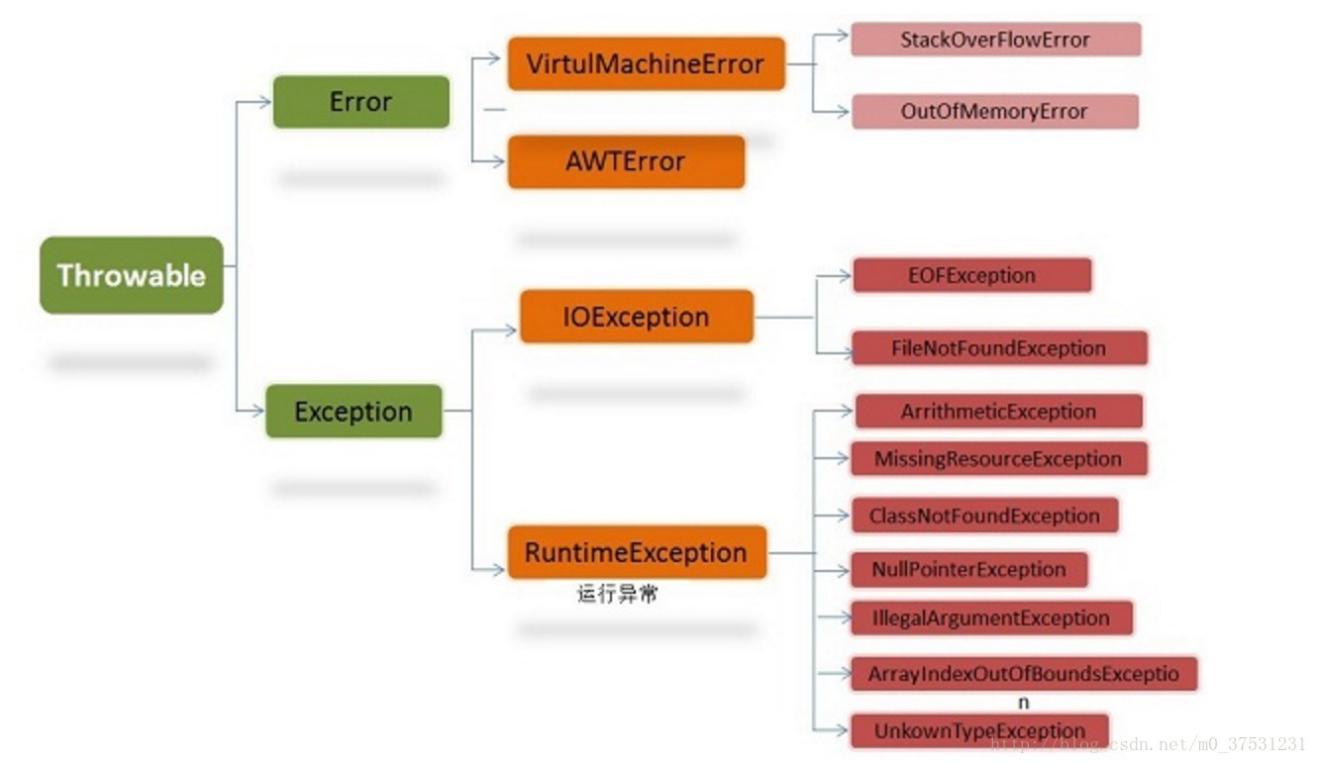
Lhist<String> stringList = new Lhist<String>(10);



47. 抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

47.5. AWT 是Abstract Window ToolKit (抽象窗口工具包)的缩写，这个工具包提供了一套与本地[图形界面](https://www.baidu.com/s?wd=%E5%9B%BE%E5%BD%A2%E7%95%8C%E9%9D%A2&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "/home/vega/Documents\\x/_blank)进行交互的接口。

48.异常



C++与Java中的异常都是对操作系统中SEH的封装。throw的时候会将变量压入栈中，同时调用call指令，跳转到之后的能够参数匹配的catch处理部分（SEH链）。参考： <https://my.oschina.net/ybusad/blog/186545>

同C++如果异常没有被处理就会导致程序退出。它的使用和C++里面的使用一样，抛出未命名异常然后命名。

数组越界会触发异常：ArrayIndexOutOfBoundsException

如果要求用户输入一个整数但是输入double：InputMismatchException C/C++就会读到点号为止

C++中Exception分为logic\_error和runtime\_error。这种基类，程序一定会用到，所以他们在java.lang里面，而不是在java.util。

Java库里面的每一个异常类都提供两种初始化：无参数初始化与String参数初始化。

异常处理是很慢的事情：涉及新对象的创建，调用栈的返回，寻找异常处理块catch

如果异常处理器不能处理一个异常，或者只是简单地希望调用者可以注意到异常，可以对异常进行重新的抛出。没有什么用，只是告诉开发者，这个异常需要其他部分额catch处理。

栈大小申请异常：

当线程请求的栈深度超过最大值，会抛出 StackOverflowError 异常；

栈进行动态扩展时如果无法申请到足够内存，会抛出 OutOfMemoryError 异常。

50.文本IO涉及到自动的解码和编码（字符流），而二进制IO不需要（二进制流），所以这是最本质的效率问题。块传输和块处理对两者都是可以适用的，这只是一种提速技术。java 的二进制IO文件与主机无关而与JVM有关。

51.

JAR（Java archive）是将一系列文件合并到单个压缩文件里，就象tar那样，多了一个META-INF/MANIFEST.MF 文件。就是别人已经写好的一些类，然后将这些类进行打包，你可以将这些jar包引入你的项目中，然后就可以直接使用这些jar包中的类和属性以及方法。jar包可能也是个可执行文件。

还可以作为一个可被java运行的应用程序，比如burpsuite

涉及因特网应用时，JAR文件显得特别有用（这时候或许叫WAR）。在JAR文件之前，Web浏览器必须重复多次请求Web服务器，以便下载完构成一个“程序片”（Applet）的所有文件。除此以外，每个文件都是未经压缩的。但在将所有这些文件合并到一个JAR文件里以后，只需向远程服务器发出一次请求即可。同时，由于采用了压缩技术，所以可在更短的时间里获得全部数据。

52.为了确保Java技术不会被邪恶目的所利用，SUN公司在设计Java的时候，设计了一套精密的安全模型；即安全管理器（Security Manager）将检查有权使用的所有系统资源。在默认的情况下，只允许那些无害的操作，要想允许执行这些操作，代码需得到数字签名，用户必须得到数字认证。

54.java生成可执行文件：

生成jar：（只需要安装jdk即可）

在同一目录下生成.java对应的.class。然后jar -cvf xxx.jar 目录名字

上述命令就会将目录打包成一个jar

要使用的话：set classpath=.;d:\javapro\my.jar，估计这是设置临时路径

之后可以直接java 类名执行了，不需要再生成.class文件。

生成exe：

对.jar使用exe4j这个工具进行转化为.exe

55.编译的时候：

注: 输出警告、错误的一些某些消息已经过简化; 请使用 -Xdiags:verbose 重新编译以获得完整输出

56.

JAR

WAR

EAR

RAR （源文件打包）

英文

Java Archive file

Web Archive file

Enterprise Archive file

包含内容

class、properties文件，是文件封装的最小单元；包含Java类的普通库、资源（resources）、辅助文件（auxiliary files）等

Servlet、JSP页面、JSP标记库、JAR库文件、HTML/XML文档和其他公用资源文件，如图片、音频文件等

除了包含JAR、WAR以外，还包括EJB组件

部署文件

application-client.xml

web.xml

application.xml

容器

应用服务器（application servers）

小型服务程序容器（servlet containers）

EJB容器（EJB containers）

级别

小（库）

中（项目）

大

57.String是非基础变量类型，所以==会出现。这时候是一种引用，存的是地址，但是用的时候就想直接对对象的使用。

Integer a = 1000, b = 1000;

System.out.println(a == b);//false，说明指向的不是同一个内存对象

Integer c = 100, d = 100;

System.out.println(c == d);//true

如果你看去看 Integer.java 类，你会发现有一个内部私有类，IntegerCache.java，它缓存了从-128到127之间的所有的整数对象。如果值的范围在-128到127之间，它就从高速缓存返回实例。

不可变对象默认是线程安全的，可以通过pool的思想实现优化：

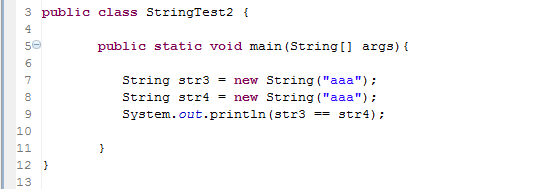
有两种创建字符串对象的方式：1）采用字面值的方式赋值  2）采用new关键字新建一个字符串对象。

字符串池在String类中，且不会被GC回收机制回收，在 Java 7 之前，字符串常量池被放在运行时常量池中，它属于永久代。而在 Java 7，字符串常量池被移到 Native Method 中。这是因为永久代的空间有限，在大量使用字符串的场景下会导致 OutOfMemoryError 错误。

调用String对象的intern()，如果池中已经有字符串了就会返回池中的字符串，不过这时候自己也已经是创建了一个变量了；关键是如果这时候池中没有这个字符串，就会加入字符串的池。

采用字面值的方式创建一个字符串时，JVM首先会去字符串池中查找是否存在"aaa"这个对象，如果不存在，则在字符串池中创建"aaa"这个对象，然后将池中"aaa"这个对象的引用地址返回给字符串常量str，这样str会指向池中"aaa"这个字符串对象；如果存在，则不创建任何对象，直接将池中"aaa"这个对象的地址返回，赋给字符串常量。

采用new关键字新建一个字符串对象时，一定会创建一个单独的变量，而不是使用缓存池。



如果要判断字符串的内容是否相同应该使用equals(Object)

StringBuffer功能上与StringBuilder一样，但是内部使用 synchronized ，即是线程安全的。

运行速度上：StringBuilder > StringBuffer（线程安全） > String

为什么 HashMap 常用 String 对象作 key

因为String的hashCode() 值是根据 String 对象中的每一个字符计算的，而Integer是直接return value作为对象的hashcode ()。冲突的可能性下降了很多。

String是一种不可变对象

String被设计为不可变（final char 数组），是为了继承、使用（线程安全；Hash的时候作为kvalue，会存在被破坏唯一性的可能性）的时候出现问题，设计为不可变类之后还可以使用常量池。

java 输出两位小数：

/\*\*

\* DecimalFormat转换最简便

\*/

public void m2() {

DecimalFormat df = new DecimalFormat("#.00");

System.out.println(df.format(f));

}

/\*\*

\* String.format打印最简便

\*/

public void m3() {

System.out.println(String.format("%.2f", f));

}

58.class not found这错误遇到的次数太多了...记得的原因一些，比如编译器版本不对，导致源文件没有被编译，还有把自己的class名成了和系统某个class一样的名，还有maven下载包的损坏，损坏的原因可能只是断网了一下，即使能手动找到包也没用，只能重新删除库重来

62.Change your project JSDK to 64-bit version, you are building and running with 32-bit JDK right now which cannot allocate more than 2GB RAM on Windows。应用程序分配内存也不是真的就会占用这么大，除非是vmware这种。

63.java.net

String mytext = java.net.URLEncoder.encode("中国", "utf-8");

String mytext2 = java.net.URLDecoder.decode(mytext, "utf-8");

63.5 如果函数使用Void作为返回值，必须要有一个return null;

64.Lambda表达式：

Java8的特性，倒着支持到android2.3

本质只是一个"语法糖",由编译器推断并帮你转换包装为常规的代码。函数式接口是指只有一个函数的接口（编译器会自动判断），lambda顶替函数式接口对象，lambda的函数类型是上下文推导过来的。

当 lambda 表达式作为方法参数时，重载解析就会影响到 lambda 表达式的目标类型。编译器会通过它所得之的信息来做出决定。如果 lambda 表达式具有 显式类型（参数类型被显式指定），编译器就可以直接 使用lambda 表达式的返回类型；如果lambda表达式具有 隐式类型（参数类型被推导而知），重载解析则会忽略 lambda 表达式函数体而只依赖 lambda 表达式参数的数量。示例：<https://blog.csdn.net/lastsweetop/article/details/82807255>

lambda 直接写一个在IDEA中，可以看到不能直接作为一个语句运行，提示不是一个表达式。

能够直接捕获外界有效只读（Effectively final）的变量（初始化之后没有改变数值，当然对于类中final变量虽然是在初始化的时候赋值，但是也支持）。不少时候，lambda都会暗中捕获this变量。除非使用数组、this->等间接方式，否则不能改变捕获的外界变量。

方法引用，ClassName::Method、object::Method（System.out::println）、this::Method（this不能省略）、super::Method、Class::new（构造方法引用）、TypeName[]::new（数组构造方法引用），不需要方法有一个static。lambda可以传递给Callable<xxx>对象，但是不能传递一个方法引用。

Function<Void,Void> callable = this::p;

对应这种写法：

public Void p(Void a) {  
 return null;  
}//第一个Void不能是void，第二个Void不能为空

基本语法:

(parameters) -> expression

或

(parameters) ->{ statements; }

// 1. 不需要参数,返回值为 5

() -> 5

// 2. 接收一个参数(数字类型),返回其2倍的值

当 lambda 的参数只有一个而且它的类型可以被推导得知时，该参数列表外面的括号可以被省略 x -> 2 \* x

// 3. 接受2个参数(数字),并返回他们的差值

(x, y) -> x – y //会推导x、y的类型

// 4. 接收2个int型整数,返回他们的和

(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)

(String s) -> System.out.print(s)

x -> 2 \* x 是有返回值的，不能用在list.forEach，而x -> x \*= 2 没有返回值

List<Person> people = ...

Collections.sort(people, new Comparator<Person>() {

public int compare(Person x, Person y) {

return x.getLastName().compareTo(y.getLastName());

}

})

=》

Collections.sort(people, (Person x, Person y) -> x.getLastName().compareTo(y.getLastName()));

=》

Collections.sort(people, Comparator.comparing((Person p) -> p.getLastName()));

=》

在类型推导和静态导入的帮助下，我们可以进一步简化上面的代码：

Collections.sort(people, comparing(p -> p.getLastName()));

=》

Collections.sort(people, comparing(Person::getLastName));

=》

由于 Collections.sort 方法不属于 List 接口，所以不推荐上面这么使用，应该使用list的sort方法

//配合java8中的forEach使用，或者配合其它函数进行使用

String[] atp = {"Rafael Nadal", "Novak Djokovic",

"Stanislas Wawrinka",

"David Ferrer","Roger Federer",

"Andy Murray","Tomas Berdych",

"Juan Martin Del Potro"};

List<String> players = Arrays.asList(atp);

// 以前的循环方式

for (String player : players) {

System.out.print(player + "; ");

}

// 使用 lambda 表达式以及函数操作(functional operation)

players.forEach((player) -> System.out.print(player + "; ")); //forEach方法会将每一个元素传参到lambda表达式

players.forEach(System.out::println); //这时候传递的是out对象的println方法引用

类似的函数还有sort、find、filter等

方法引用是用来直接访问类或者实例的已经存在的方法或者构造方法。方法引用提供了一种引用而不执行方法的方式，它需要由兼容的函数式接口构成的目标类型上下文。计算时，方法引用会创建函数式接口的一个实例。

流求值是惰性的，并因此能够实现无限数据源

并行流基于Fork/Join框架

phpProgrammers.stream()

.filter((p) -> (p.getSalary() > 1400))

.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

javaProgrammers.stream()

.filter(genderFilter)

.limit(3) //限制了结果集的大小为3

.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

//代替内部匿名类

// 使用匿名内部类

btn.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent event) {

System.out.println("Hello World!");

}

});

// 或者使用 lambda expression

btn.setOnAction(event -> System.out.println("Hello World!"));

//比如实现Runnable接口

// 1.1使用匿名内部类

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("Hello world !");

}

}).start();

// 1.2使用 lambda expression

new Thread(() -> System.out.println("Hello world !")).start();

65.使用迭代器或 for-each 循环是遍历 ArrayList 最有效的方式，性能比采用索引值的传统 for 循环方式好两倍。

66.我们先创建了一个 ArrayList，并插入一个 100000 个随机整数，并通过5 种不同的方式遍历所有的值来查找最大值

让我们快速看一下每个方法，按照运行速度由快到慢（尤其是foreach和迭代器）：

命令式风格

iteratorMaxInteger()——使用迭代器遍历列表：

public int iteratorMaxInteger() {

int max = Integer.MIN\_VALUE;

for (Iterator it = integers.iterator(); it.hasNext(); ) {

max = Integer.max(max, it.next());

}

return max;

}

forEachLoopMaxInteger()——不使用迭代器，使用 For-Each 循环遍历列表（不要误用 Java 8 的 forEach）

public int forEachLoopMaxInteger() {

int max = Integer.MIN\_VALUE;

for (Integer n : integers) {

max = Integer.max(max, n);

}

return max;

}

forMaxInteger()——使用简单的 for 循环和索引遍历列表：

public int forMaxInteger() {

int max = Integer.MIN\_VALUE;

for (int i = 0; i < size; i++) {

max = Integer.max(max, integers.get(i));

}

return max;

}

1

然后就是优雅但是很慢的函数式风格

parallelStreamMaxInteger()——使用 Java 8 并行流遍历列表：

public int parallelStreamMaxInteger() {

Optional max = integers.parallelStream().reduce(Integer.max);

return max.get();

}

lambdaMaxInteger()——使用 lambda 表达式及流遍历列表。优雅的一行代码：

public int lambdaMaxInteger() {

return integers.stream().reduce(Integer.MIN\_VALUE, (a, b) -> Integer.max(a, b));

}

68.Java8中的Stream特性

Stream 不是集合元素，它不是数据结构并不保存数据，它是有关算法和计算的，它更像一个高级版本的 Iterator。原始版本的 Iterator，用户只能显式地一个一个遍历元素并对其执行某些操作；高级版本的 Stream，用户只要给出需要对其包含的元素执行什么操作，比如 “过滤掉长度大于 10 的字符串”、“获取每个字符串的首字母”等，Stream 会隐式地在内部进行遍历，做出相应的数据转换。Stream 就如同一个迭代器（Iterator），单向，不可往复，数据只能遍历一次，遍历过一次后即用尽了。

集合类持有的所有元素都是存储在内存中的，非常巨大的集合类会占用大量的内存，而Stream的元素却是在访问的时候才被计算出来，这种“延迟计算”的特性有点类似Clojure的lazy-seq，占用内存很少。所以stream可以表示无限的集合，比如说自然数。

sortedJavaProgrammers = javaProgrammers

.stream()

.sorted( (p, p2) -> (p.getSalary() - p2.getSalary()) )

.collect( toList() );

平方数

List<Integer> nums = Arrays.asList(1, 2, 3, 4);

List<Integer> squareNums = nums.stream().

map(n -> n \* n).

collect(Collectors.toList());

如果是偶数就会平方输出

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Stream<Integer> stream = numbers.stream();

stream.filter((x) -> {

return x % 2 == 0;

}).map((x) -> {

return x \* x;

}).forEach(System.out::println);

}

自然数集合的规则非常简单，每个元素都是前一个元素的值+1，因此，自然数发生器用代码实现如下：

class NaturalSupplier implements Supplier<Long> {

long value = 0;

public Long get() {

this.value = this.value + 1;

return this.value;

}

}

Stream<Long> natural = Stream.generate(new NaturalSupplier()); //反复调用get()，将得到一个无穷数列，利用这个Supplier，可以创建一个无穷的代表自然数的Stream

natural.limit(10).forEach(System.out::println);

Java8中增加了流(stream)的概念，为数据的处理带来了很大的方便。并行流将数据分割成不同的块，并且用不同的线程处理不同的块。

现在要根据输入`n`，求从1到n的和，这里我们不使用`n\*(1+n)/2`的方式（可能溢出），而是使用累加。我们分别写出迭代、顺序流、并行流的写法。

public class ParallelStreams {

public static long sequentialSum(long n) {

return Stream.iterate(1L, i->i+1)

.limit(n)

.reduce(0L, Long::sum);

}

public static long iterativeSum(long n) {

long result = 0;

for (long i = 1L; i <= n; i++) {

result += i;

}

return result;

}

public static long parallelSum(long n) {

return Stream.iterate(1L, i->i+1)

.limit(n)

.parallel()//我们可以看到，将顺序流转换为并行流只需要`parallel()`方法即可。

.reduce(0L, Long::sum);

}

}

耗时如下：

顺序流:173 毫秒

循环:4 毫秒

并行流:697 毫秒

为什么会这样？

`iterate`生成的是`Stream<Long>`对象，需要拆箱才能求和；

`iterate`很难分割成独立的小块，因为每次应用这个函数都需要前一次应用的结果，也就是说它其实是顺序执行的。这样反而在并行时增加了分配线程的开销。

这里我们使用`LongStream.rangeClosed`方法来优化，它的特点有：

`LongStream.rangeClosed`直接产生原始类型的`long`数字，没有拆箱与装箱的开销。

`LongStream.rangeClosed`产生一个数字范围，很容易拆分成多个小块。

我们再用`LongStream.rangeClosed`写出顺序和并行计算的版本：

public static long parallelRangedSum(long n) {

return LongStream.rangeClosed(1, n)

.parallel()

.reduce(0, Long::sum);

}

使用了LongStream以后耗时如下：

顺序Range流:7 毫秒

并行Range流:3 毫秒

69.类就是对接口的实现，java因为多重继承太复杂了就只允许进行单一继承，只能有一个父类，不过可以有多个继承接口。

70.org.apache.http.HttpResponse这种包，那么在环境变量中就要设置能找到org文件夹的路径。

72.idea可以直接打开eclipse的项目，我打开的时候（打开项目中的.propject文件）遇到了项目使用的是Java SE1.6，然后就提醒我没有。这不就是我已经有了的jdk1.8的旧版么。idea可以选择一个文件来进行run，或者选择项目来进行run。我一开始打算直接用javac和java来解决问题，但是这个项目我虽然引入了需要的基本类文件，却不能将依赖的jar引入进去（其实需要指定jar包classPath）。无奈之下只好使用IDE来直接打开了。

73.产生4位随机验证码：

public static void main(String[] args) {

String codeTable ="1234567890abcdefghijkmnpqrstuvwxyz";

String messageCode ="";

Random random =new Random();

for(int i1=0; i1 < 4; ++i1)

messageCode +=codeTable.charAt( random.nextInt(codeTable.length()) );

System.out.println(messageCode);

}

reverse word <=>keyword

Unix epoch时间戳

pre increment decrement

post

vild card 通配符

escape 转义

nested 嵌套的

imutable 不可变的

74.反射 ：

相关网址：<https://blog.csdn.net/javazejian/article/details/70768369>

主要是指程序可以访问，检测和修改它本身状态或行为的一种能力，能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。“加载数据库驱动”，“Spring的IOC加载XML中指定内容”使用了反射。

其作用是使用一定性能来换取灵活，但是性能牺牲还是很大的，后来几种框架ButterKnife、Dagger2通过apt代码自动生成技术，其注解是停留在编译时处理，完全不影响性能

Class.forName方法的好处是无须持有该类的实例对象引用而却获取Class对象。

有以下几个作用：

使用不同的字符串类名来加载不同的驱动，如果将这些字符串独立写一个配置文件，通过配置文件中的类名，通过对配置文件中的类名的修改，然后在某一个函数中使用配置文件的行数中的字符串来实例化一个对象

使用别人的库的时候，有时候会不满意功能，想要访问一个私有变量的时候只能使用反射

通过反射还能够修改final关键字修饰的变量，只要不会被编译器内联优化的 final 属性就可以通过反射有效的进行修改

class Person {

public final String name = "Mike"; //简单的字符串类型会被直接优化，直接认为是一个常量，所有用到的地方都被直接替换为字符串常量

public final String name2 = new String("Mike"); //

public final String city; //可被有效修改

}

java.lang.reflect.Constructor;

java.lang.reflect.Field;

java.lang.reflect.Method;

java.lang.reflect.Modifier;

获取类：

//第一种方式：

java.lang.Class;

Classc1 = Class.forName("Employee");

//第二种方式：

//java中每个类型都有class 属性.

Classc2 = Employee.class;

//第三种方式：

//java语言中任何一个java对象都有getClass 方法

Employee e = new Employee();

Classc3 = e.getClass(); //c3是运行时类 (e的运行时类是Employee)

创建对象：

Object =Classc1.newInstance(); 就会调用无参数构造函数

获取类的属性：

//获取整个类 ，像定义一样输出类

Class c = Class.forName("java.lang.Integer");

//获取所有的属性

Field[] fs = c.getDeclaredFields();

//定义可变长的字符串，用来存储属性

StringBuffer sb = new StringBuffer();

//通过追加的方法，将每个属性拼接到此字符串中

//最外边的public定义

sb.append(Modifier.toString(c.getModifiers()) + " class " + c.getSimpleName() +"{\n");

//里边的每一个属性

for(Field field:fs){

sb.append("\t");//空格

sb.append(Modifier.toString(field.getModifiers())+" ");//获得属性的修饰符，例如public，static等等

sb.append(field.getType().getSimpleName() + " ");//属性的类型的名字

sb.append(field.getName()+";\n");//属性的名字+回车

}

sb.append("}");

System.out.println(sb);

//获取指定属性

Field idF = c.getDeclaredField("id");

idF.setAccessible(true); //使用反射机制打破了封装性

Object o = c.newInstance();

idF.set(o, "110");

通过类名来实例化一个对象： object\_factory\_map内容来自于配置文件

Any GetInstanceByName(const string& name) {

return Class.forName(name).newInstance(); //使用newInstance函数来实例化一个对象只能没有参数

}//Any 是java idl 里面的一个类，继承自Object实现了IDLEntity

不带参数：

A a=(A)Class.forName("reflect.A").newInstance();

带参数：

Class c=Class.forName("A");

Constructor c1=c.getDeclaredConstructor(new Class[]{int.class,int.class});

A a1=(A)c1.newInstance(new Object[]{5,6}); //需要强制类型转换

通过反射机制能够修改一个对象的私有成员

import java.lang.reflect.Field;

public class Demo1 {

public static void main(String[] args) throws ClassNotFoundException, IllegalArgumentException, Exception {

Class c=Class.forName("com.jr.reflect.Dep");

Object c1=c.newInstance();//创建对象后应赋值给另一个对象

Field [] fields=c.getDeclaredFields();

fields[0].setAccessible(true); //修改访问限制

System.out.println(fields[0].getName());

fields[0].set(c1, 1);//第一个参数为:被创建的对象，二：要修改的值

System.out.println(fields[0].get(c1));

}

}

通过修改访问限制来实现python中“删除”成员的效果

带参数的构造：

//取得指定带int和String参数构造函数,该方法是私有构造

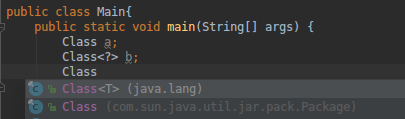
Constructor cs2=clazz.getDeclaredConstructor(int.class,String.class);

//由于是private必须设置可访问

cs2.setAccessible(true);

//创建user对象

User user2= (User) cs2.newInstance(25,"lidakang");



Constructor<?> 相当于 Constructor

75.java 注解

注解的作用：

生成文档，用于JavaDoc

减少代码编写的复杂性，Android中的ButterKnife、Spring框架等

在编译的时候进行严格的检查，比如@Override

java内置一部分注解，例如：当方法或是类上面有@Deprecated注解时，说明该方法或是类都已经过期不建议再用，@SuppressWarnings 则表示忽略指定警告，@SuppressWarnings(“uncheck”)是最简单的使用

自定义注解：

//声明 @Test(name=”xx”) 注解

@Target(ElementType.METHOD) //指明注解目标是类、方法、参数、方法

//复杂点 @Target(value={CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE}) 或者 @Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE}) 说明这实际上是一个数组类型，虽然最简写法是单元素

@Retention(RetentionPolicy.RUNTIME) //注解生存期是运行时

public @interface Test {

String name() default "";

//判断是否作为主键约束

boolean primaryKey() default false;

}

注解不支持继承，自定义注解之后在编译的时候自动继承 java.lang.annotation.Annotation接口，虽然java支持继承自多个接口，但是这里做不到。

**自定义注解的四个元注解修饰**：

注解类型： @Target 必须

CONSTRUCTOR：构造器的声明  
 FIELD：域声明（包括enum实例）  
 LOCAL\_VARIABLE：局部变量声明  
 METHOD：方法声明  
 PACKAGE：包声明  
 PARAMETER：参数声明  
 TYPE：类、接口（包括注解类型）或enum声明

注解生存期： @Retention 必须

SOURCE：注解将被编译器丢弃  
 CLASS：注解在class文件中可用，但会被VM丢弃（只在编译期间有效的注解）  
 RUNTIME：VM将在运行期间保留注解，因此可以通过反射机制读取注解的信息

@Documented 将注解包含在JavaDoc中

@inherited 允许子类继承父类的注解

注解一开始使用反射机制去读取信息

将注解配合APT技术开发？

嵌套注解：



下面的描述中使用POJO不是很好，因为POJO不包含JavaBean，实际上应该是“Java类”

Annotation Processing （后面简称 AP ）发生在编译时，可以利用 AP 在编译时生成一些代码来解决一些通用的问题，从某种程度上讲，可以替代一些完全依赖反射实现的库从而取得更好的性能。annotation processing tools(APT)，这也是Android ButterKnife依赖的东西。我们在运行时用反射的场景一般是为了通用，比如一些解析 JSON 的库，作者在写库的时候不可能知道你所使用的 JSON 格式是什么样子的，不知道你的 POJO 是什么样子，所以用反射在运行时获取 POJO 有哪些字段，从而动态的完成 JSON 与 POJO 的映射。假如我们在运行之前就知道了 JSON 的格式，POJO 有哪些字段，知道了 JSON 与 POJO 的映射关系，那我们就可以不用反射。我们在写完代码，编译运行的时候，我们的代码就固定了，有哪些 POJO 就固定了。AP 可以在编译的时候获取所有 POJO 对象，然后通过反射（这里的反射发生在编译时，不在运行时）获取 POJO 的字段，在编译期生成 JSON 与 POJO 映射的相关代码，最后这些代码会和你写的代码一起编译打包运行。相当于你自己手写的转换代码现在是 AP 帮你写了，在这种场景下，AP 生成代码的方式肯定是优于完全基于运行时反射的 JSON 解析库，因为就和你自己手写的代码一样。

这里有一个小问题，那就是 AP 生成的代码怎么引用到，像 butterknife、 dagger 在运行时也是有反射的，因为他们要少量反射去寻找 AP 生成的那些 class，性能优于完全基于反射的库。而 dagger2 去掉了所有的运行时反射，因为它不需要运行时去寻找 AP 生成的 class，你要在代码中直接引用生成的 class，这时候就能感觉到是 AP 再帮你写代码，因为你引用了一个不是自己写的 class。

75. JavaBean and JavaBeans

JavaBeans 从狭义来说，指的是 JavaBeans 规范也就是位于 java.beans 包中的一组 API。从广义上来说，JavaBeans 指的是 API 集合，比如 Enterprise JavaBeans。

JavaBean 指的是按照某种规范实现的Java类，对所有变量实现了自己的read、write函数。

76.POJO是Plain Ordinary Java Object的缩写不错，但是它通指没有使用Entity Beans的普通java对象，可以把POJO作为支持业务逻辑的协助类。POJO实际上指的即普通Java类，不具有任何特殊角色和不继承或不实现任何其它Java框架的类或接口，可能会对部分变量实现read、write函数。

77.邮件的处理有Java Mail API

厂商所提供的Java Mail服务程序可以有选择地实现某些邮件协议，常见的邮件协议包括：

（1）SMTP：简单邮件传输协议，用于发送电子邮件的传输协议；

（2）POP3：用于接收电子邮件的标准协议；

（3）IMAP：互联网消息协议，是POP3的替代协议。

这三种协议都有对应SSL加密传输的协议，分别是SMTPS，POP3S和IMAPS。除Java Mail服务提供程序之外，Java Mail还需要JAF(JavaBeans Activation Framework)来处理不是纯文本的邮件内容，这包括MIME（多用途互联网邮件扩展）。对附件的处理和HTTP发送文件一样：如果一封邮件中含有附件，那邮件的Content-Type域中必须定义multipart/mixed类型，邮件通过multipart/mixed类型中定义的boundary标识将附件内容同邮件其它内容分成不同的段。基本格式：Content-Type： multipart/mixed;boundary="{分段标识}"

mail. jar：此JAR文件包含Java Mail API和Sun提供的SMTP、IMAP和POP3服务提供程序；

activation. jar：此JAR文件包含JAF API和Sun的实现。

Java Mail包含两部分内容，一部分是Java Mail API，定义了一组平台无关、独立于通讯协议的邮件程序框架，该部分称为应用级接口，也就是供我们调用的部分，另一部分是service provider，该部分使用特定的协议来实现第一部分定义的抽象类和接口，这些协议包括：SMTP、NNTP、POP3、IMAP，如果让Java Mail与邮件服务器通信，就需要相应的协议支持，该部分称为服务提供者接口，也就是Java Mail自身需要的协议支持。在使用Java Mail时，通常我们只需将mail. jar放在class path下使用，它包含了Java Mail API部分和SUN自己实现的service provider部分。可能也有特殊的时候，我们应用程序中需要自己实现service provider部分，那我们只需要mailapi. jar。

79.javadoc：

在注释中出现以@开头东东被称之为Javadoc文档标记，是JDK定义好的，如@author、@version、@since、@see、@link、@code、@param、@return、@exception、@throws等，通常使用以下这种注释，可能还会配合一点HTML标签：

/\*\*

\*@param context

\*@param attrs

\*/

Public DotView(Context context, Attribute attrs) {

}

使用 javadoc -d 文档存放目录 -author -version 源文件名.java  导出成为javadoc

79.实际上，Java中存在四种引用，它们由强到弱依次是：强引用、软引用、弱引用、虚引用：

强引用（Strong Reference）：通常我们通过new来创建一个新对象时返回的引用就是一个强引用，若一个对象通过一系列强引用可到达，它就是强可达的(strongly reachable)，那么它就不被回收

软引用（Soft Reference）：软引用和弱引用的区别在于，若一个对象是弱引用可达，无论当前内存是否充足它都会被回收，而软引用可达的对象在内存不充足时才会被回收，因此软引用要比弱引用“强”一些

弱引用对象（WeakReference类 ）的存在不会阻止它所指向的对象被垃圾回收器回收。弱引用最常见的用途是实现规范映射(canonicalizing mappings，比如哈希表）。弱可达的(weakly reachable)（也就是说当前指向它的全都是弱引用）。弱引用 的目标只有 弱引用自己的时候就会在GC的回收。举例：ThreadLocalMap 通过Entry extends WeakReference<ThreadLocal>来表示value。

虚引用（Phantom Reference）：虚引用是Java中最弱的引用，那么它弱到什么程度呢？它是如此脆弱以至于我们通过虚引用甚至无法获取到被引用的对象，虚引用存在的唯一作用就是当它指向的对象被回收后，虚引用本身会被加入到引用队列中（通过查看队列可以知道对象是否被回收），用作记录它指向的对象已被回收。

80.调用exe并获取其标准输出作为返回值

最好不要使用

Process p =Runtime.getRuntime().exec("xxxx.exe")

p.getInputStream(), p.getOutputStream()

而是使用一个回调，exe每输出一行字符就进行一次回调。不过实时显示最好的办法是类似于linux中popen函数，建立程序之间的管道。

84.类的编写： 通过设置interface实现CallBack

public class CalendarItem extends Fragment implements NoteCalendar2.ChangeDateListener{

CallBack mCallBack;

public void setCalendarCallBack(CallBack cb){

mCallBack = cb;

}

public interface CallBack{

abstract void onChangeSelectedDate(int year,int month ,int day);

}

}在其他方法中通过mCallBack.onChangeSelectedDate(year, month, day); 调用。

85.

void关键字表示函数没有返回结果，是java中的一个关键字。

java.lang.Void是一种类型（也就有Void.TYPE），Void作为函数返回值表示函数返回null。

87.

// 错误: 无法从静态上下文中引用非静态变量 this

public class Test {

public static void main(String args[]) {

Test test=new Test();

Test2 test2=new Test2();

test2.print();

}

public int function() {

int a=1,b=4;

int c=5;

return (a+b)\*c;

}

public class Test2 {

private int a=function();

public void print() {

System.out.println(a);

}

}

}

java静态内部类的使用：

class OuterClass{

private static String msg = "GeeksForGeeks";

// 静态内部类

public static class NestedStaticClass{

// 静态内部类只能访问外部类的静态成员

public void printMessage() { //没有static

// 试着将msg改成非静态的，这将导致编译错误

System.out.println("Message from nested static class: " + msg);

}

}

}

88.Java中闭包的实现是不完整的：

Java8编译器实现的只是capture-by-value（除了数组），暂时没有实现capture-by-reference.但Java又不肯明说，只能粗暴地一刀切，就说既然内外不能同步，那就不许大家改外围的局部变量。这也就导致了,匿名类里面使用外部的变量的话需要使用final修饰.

同时匿名类已经不是自己类中了，所以不能在其中new自己类中定义的private对象。

但是Java只是不允许改变被lambda表达式捕获的变量，并没有限制这些变量所指向的对象的状态能不能变。要从lambda表达式向外传值的常见workaround之一就是用长度为1的数组：

String[] a = new String[1];

... ( () -> a[0] = "a" );

return a[0];

90.CodeCache

关于启动后负载高的原因，网上很多文章都说是由于启动后随着代码的执行，jvm的jit编译器将部分热点代码编译为目标机器代码，由于编译线程占用了大量的cpu导致系统负载高。jit编译器需要代码执行超过一定频率才会将其编译并保存，系统刚启动的时候大部分代码都是处于解释执行阶段

Jvm提供了一个参数-Xcomp，可以使jvm运行在纯编译模式下，所有方法在第一次被调用的时候就会被编译成机器代码。如果因为解释导致代码的效率一直不是很好，可以开启这个从而提升效率。

编译后的代码被缓存起来就是CodeCache，如果这个缓冲区满了就满了，没有清理机制。很自然，会导致不能后面的代码摆脱解释执行的命运。

92.没有1E，E后面必须要加上数字：

int i1=2;

System.out.println(""+1E2+i1);

93.文件流操作：Java中的基本输入输出流 以及 文件相关输入输出流

基本上函数的参数都是byte[]，而不是char[]

需要处理的异常：IOException（java.io.IOException）

try{

System.out.println(test.getInputMessage());

}catch(IOException ex) {

ex.printStackTrace();

}

都有close函数、flush()函数，有点似乎flush函数需要传如一个true值

1.通过int count=System.in.read(buffer) 去读到缓冲区，读取到回车为止

byte buffer[]=new byte[1024];

int count=System.in.read(buffer);

缺点：下面的函数有一个不支持多字符文字的bug，如果直接将return new String(buffer);就能够显示出来，但是这时候后面会跟着一堆不可见字符，因为java里面的字符串也是要借助一些事物来判断结尾。

public String getInputMessage() throws IOException{

System.out.println("请输入您的命令∶");

byte buffer[]=new byte[1024];

int count=System.in.read(buffer);

char[] ch=new char[count-2];//最后两位为结束符\r\n，删去不要

for(int i=0;i<count-2;i++)

ch[i]=(char)buffer[i]; //直接限制了长度，从而得结尾位置，但是这里还是存在一个byte直接转一个char会失效的问题

String str=new String(ch);

return str;

}

通过上面的函数可以知道通过char转为String的时候,每一个字节都作为单独的字符显示。但是将byte[] 转换为String的时候能够正常进行多字符编码的解析。最终修复的函数如下：

public String getInputMessage() throws IOException{

System.out.println("请输入您的命令∶");

byte buffer[]=new byte[1024];

int count=System.in.read(buffer);

byte[] ch=new byte[count-2]; //最后两位为结束符，删去不要

for(int i=0;i<count-2;i++)

ch[i]=buffer[i]; // 不能直接转，必须选择有数据的部分，不然会带上乱码

String str=new String(ch);

return str;

}

byte[] 、 char[] 以及 String 之间的区别与转换：

java中的char是Unicode字符，也就是一个字符会占有两个字节的整数位置这样。但是byte这种类型的存储单位就是一个字节。两者之间本质还是一个编码与解码的关系。所以转换函数就是通过编码以及解码。

private byte[] getBytes (char[] chars) {

Charset cs = Charset.forName ("UTF-8");

CharBuffer cb = CharBuffer.allocate (chars.length); //缓冲区分配大小

cb.put (chars);

cb.flip ();

ByteBuffer bb = cs.encode (cb); //java.net.URLEncoder.encode

return bb.array();

}

private char[] getChars (byte[] bytes) {

Charset cs = Charset.forName ("UTF-8");

ByteBuffer bb = ByteBuffer.allocate (bytes.length);

bb.put (bytes);

bb.flip ();

CharBuffer cb = cs.decode (bb);

return cb.array();

}

或者使用类InputStreamReader OutputStreamWriter

InputStreamReader 将字节流转换为字符流。是字节流通向字符流的桥梁。 如果不指定字符集编码，该解码过程将使用平台默认的字符编码。

int read();//读取单个字符。

int read(char []cbuf);//将读取到的字符存到数组中。返回读取的字符数。

OutputStreamWriter 将字符流转换为字节流，操作系统显示的时候会自动将字节流按照当前的编码设置转换为字符流显示

void write(int c);//将单个字符写入。

viod write(String str,int off,int len);//将字符串某部分写入。

void flush();//将该流中的缓冲数据刷到目的地中去。

//String 与它们之间的转换

String str ="123";

byte[] bt=str.getBytes();

2.FileInputStream 、FileOutputStream 支持操作何形式的文件，FileDescriptor 文件描述符指定的流，如果是操作文本文件据说FileInputReader和FileOutputWriter效率更高

学习网址：http://blog.csdn.net/caixiexin/article/details/6719407 关注read以及write函数。读入支持skip函数

很明显是从java.io.InputStream、java.io.OutputStream继承过来，所以也会自带缓冲区，因为也有flush函数

文件指针的seek操作，由文件对象提供与维护

public void copyFile(String src,String dest) throws IOException{

FileInputStream in=new FileInputStream(src);//构造函数接受文件路径、File对象

File file=new File(dest);

if(!file.exists())

file.createNewFile();

FileOutputStream out=new FileOutputStream(file);

int c;

byte buffer[]=new byte[1024];

while((c=in.read(buffer))!=-1) {

for(int i=0;i<c;i++)

out.write(buffer[i]);

}

in.close();

out.close();

}

FileChannel类 Java NIO FileChannel是连接文件的通道。Java NIO FileChannel类是NIO用于替代使用标准Java IO API读取文件的方法，在速度上较传统的文件读写API快。使用FileChannel，您可以从文件中读取数据和将数据写入文件，以及操作（比如说对文件的裁剪）。虽然是NIO中的一个类，但是缺点就是FileChannel无法设置为非阻塞模，因为继承的父类没有提供 configureBlocking方法调成非阻塞。不过IO中有提供 Selector类，类似于C中的select函数，可以配合阻塞操作。

可以通过FileInputStream 、FileOutputStream对象中的getChannel()返回一个FileChannel对象。这个类的对象需要通过InputStream，OutputStream或RandomAccessFile来获取。同样是最后要close的。如果要说的话，获取与linux中的管道有点接近。

try {

fileInputStream = new FileInputStream(fromFile);

fileOutputStream = new FileOutputStream(toFile);

//得到fileInputStream的文件通道

fileChannelInput = fileInputStream.getChannel();

//得到fileOutputStream的文件通道

fileChannelOutput = fileOutputStream.getChannel();

//将fileChannelInput通道的数据，写入到fileChannelOutput通道

fileChannelInput.transferTo(0, fileChannelInput.size(), fileChannelOutput);

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (fileInputStream != null)

fileInputStream.close();

if (fileChannelInput != null)

fileChannelInput.close();

if (fileOutputStream != null)

fileOutputStream.close();

if (fileChannelOutput != null)

fileChannelOutput.close();

} catch (IOException e) {

e.printStackTrace();

}

}

返回通道连接的文件的大小： 不如从文件属性中获取文件大小来得快

channel.size（）;

读写数据：通过read以及write函数

ByteBuffer buf = ByteBuffer.allocate（48）;

FileInputStream is = new FileInputStream(in);

FileOutputStream os = new FileOutputStream(out);

FileChannel fis = is.getChannel();

FileChannel fos = os.getChannel();

while(fis.read(bytedata)!= -1){

bytedata.flip();

fos.write(bytedata);

bytedata.clear();

}

截断文件：

channel.truncate（1024）;//从当前位置开始的1024字节截断文件，相当于删除之后的内容

设置文件指针位置： 从而进行特定位置上的读写

long pos channel.position（）;

channel.position（pos +123）;

强制写入磁盘：

channel.force(true) 将通道里尚未写入磁盘的数据强制写到磁盘上。出于性能方面的考虑，操作系统会将数据缓存在内存中。操作系统的行为会影响部分函数API的具体执行效果。尤其是这种调用操作系统API并且对性能有不小影响的。

4.FileReader FileWriter 对文本文件的读写

FileReader：

通过这种方式读到的是\r\n回车，如果仅仅消除了\n，那么\r的效果就会有点爽

int read(); // 读取单个字符。返回作为整数读取的字符，如果已达到流末尾，则返回 -1。

int read(char []cbuf);//将字符读入数组。返回读取的字符数。

void close();//关闭此流对象。释放与之关联的所有资源。

FileWriter:

void write(String str) //写入字符串。当执行完此方法后，字符数据还并没有写入到目的文件中去。此时字符数据会保存在缓冲区中。

viod flush() //刷新该流中的缓冲。尝试将缓冲区中的字符数据保存到目的文件中去。

viod close() //关闭此流。在关闭前会先刷新此流的缓冲区。在关闭后，再写入或者刷新的话，会抛IOException异常。

write函数的参数类型与read函数的参数类型 不是一样的诶

1. Scanner PrintWriter PrintStream 格式化的输入以及输出。

Scanner：//支持任意类型的读取到对应的类型对象

Scanner s = new Scanner(System.in);

PrintWriter：//支持任意类型的格式化输出

学习网址：http://www.cnblogs.com/xiaotiaosi/p/6394147.html

比较奇特的事情（这种不是格式化输出）是提供了一个append函数允许直接向其缓冲区中提供字符序列（比如说字符串）

PrintStream 为其他输出流添加了功能，使它们能够方便地打印各种数据值表示形式。与其他输出流不同，PrintStream 永远不会抛出IOException，应该是在函数中自己处理好了！

同时提供一个写入自动刷新功能，即在write函数之后自动调用flush函数。

PrintStream 打印的所有字符都使用平台的默认字符编码转换为字节。

创建对象提供一个OutStream：

FileOutputStream out=new FileOutputStream("D:/test.txt");

PrintStream p=new PrintStream(out);

6.BufferedReader BufferedWriter

int read();//读取单个字符。

int read(char[] cbuf,int off,int len);//将字符读入到数组的某一部分。返回读取的字符数。达到尾部 ，返回-1。

String readLine(); //读取一个文本行。

void close(); //关闭该流。并释放与该流相关的所有资源。

BufferedReader inFromClient = new BufferedReader(new InputStreamReader(socket.getInputStream())); 调用了close方法以后socket也会被关闭

使用了上面这行命令以后就会通过getInputStream读取到自己设定的缓冲区的时候就就会出现一些错误的字符，建议使用DataOutpuStream，这种流支持的接口比较多。

BufferedWriter：

void write(char ch);//写入单个字符。

void write(char []cbuf,int off,int len)//写入字符数据的某一部分。

void write(String s,int off,int len)//写入字符串的某一部分。

void newLine()//写入一个行分隔符。

void flush();//刷新该流中的缓冲。将缓冲数据写到目的文件中去。

void close();//关闭此流，再关闭前会先刷新他。

1. JDK8 提供的特性： 两个readAllxxx 函数

最开头介绍的getResourceStream也是获取所有内容

System.out.println(new String(Files.readAllBytes(Paths.get("D:\\jd.txt"))));

List<String> lines = Files.readAllLines(Paths.get("D:\\jd.txt"), StandardCharsets.UTF\_8);

import java.nio.file.Files;

import java.io.IOException;

import java.nio.file.Paths;

8.读取特殊文件，比如通过dom//特定将两种斜杠都试了一下

System.out.println(new String(Files.readAllBytes(Paths.get("C:/Users\\tmp\\jd.txt"))));

}catch(IOEx4j读取xml文件

装饰流：DataInputStream DataOutputStream Scanner PrintStream 建立在其他流之上，提供额外的功能（FileChannel也是装饰类）

DataInputStream DataOutputStream

DataOutputStream数据输出流 将java基本数据类型写入数据输出流中。并可以通过数据输入流DataInputStream将数据读入

DataInputStream:

int read(byte[] b);//从输入流中读取一定的字节，存放到缓冲数组b中。返回缓冲区中的总字节数。

int read(byte[] buf,int off,int len);//从输入流中一次读入len个字节存放在字节数组中的偏移off个字节及后面位置。

使用DataInputStream来读取字节流中的数据：

ByteArrayInputStream bin=new ByteArrayInputStream(byte[] buf);

DataInputStream dis=new DataInputStream(bin);

String name=dis.readUTF();//从字节数组中读取字符串（编码）

序列化有很多种协议，Java默认的序列化协议在空间与时间上的开销都不好。https://blog.csdn.net/baiye\_xing/article/details/73249819

Java序列化： 存的不仅仅是数据还有类型。Android还额外提供了Parcelable序列化接口，但是这种序列化不提供永久存储功能。

序列化输出与输入： 不需要考虑异常抛出（不需要自己处理）

输出到文件：

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("student.txt"));

oos.writeObject(new Student(10090, "小明", 20));

oos.close();

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("student.txt"));

Student s = (Student)ois.readObject();

System.out.println(s.id);

产生的文件内容是这样子的“id:10090 name:小明 age:20”

如果要将序列化的内容输入/输出到字节数组，那么应：

ByteArrayOutputStream baos = new ByteArrayOutputStream(); //ByteArrayOutputStream捕捉内存缓冲区的数据转换为字节流。构造函数支持传递一个int值作为内部byte缓冲区的大小。

ObjectOutputStream oos2 = new ObjectOutputStream(baos);

oos2.writeObject("Save another object:\n");

oos2.writeObject(new Person("Phil", "654321")); //将数据写入到内部的byte缓冲区中

oos2.close();

ObjectInputStream ois2 = new ObjectInputStream(new ByteArrayInputStream(baos.toByteArray()));

s = (String)ois2.readObject();

p = (Person)ois2.readObject();

System.out.println(s + p);

在public等域作用关键字后面加上transient，就能指示不希望被序列化的数据成员。一个静态变量不管是否被transient修饰，均不能被序列化。但是其实还是可以让被transient修饰的成员被序列化，需要自己编写序列化的输出输入函数：

public class Person implements Serializable {

private String userName;

private transient String password;

public Person(String userName, String password) {

this.userName = userName;

this.password = password;

}

public String toString() {

return "userName:" + userName + " password:" + password;

}

private void writeObject(ObjectOutputStream out) throws IOException {

out.defaultWriteObject(); //序列化所有非transient字段,必须是该方法的第一个操作

out.writeObject(password); //序列化transient字段

}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {

in.defaultReadObject(); //反序列化所有非transient字段,必须是该方法的第一个操作

password = (String)in.readObject(); //反序列化transient字段

}

}

ArrayList本身实现了Serializable接口

读取整个文件的时候可以通过配合文件操作类File，获取File.length 来动态new一个数组。之后直接读取到这个数组中。如果文件过大就分块读取。

Java的输出可以到一个文件，也可以到字节数组（如果视为缓冲区，那么和输出到标准输出流中一样，都是输出到内存上的缓冲区）。

94.文件操作： File 类

学习网址：http://www.runoob.com/java/java-file.html

File file=new File(dest);

if(!file.exists())

file.createNewFile(); //可以是写文件的时候创建

提供了文件内指针的操作seek函数

95.缓冲区 ByteBuffer 、CharBuffer、StringBuffer 缓冲区的作用就是将多次的读写通过缓冲变成一次

ByteBuffer buffer=ByteBuffer.allocate(256); //分配指定大小的空间最为缓冲区

ByteBuffer buffer=ByteBuffer.wrap(byteArray); //直接将数组包含进来，从而缓冲区中就会拥有与数组一样的数据

buffer.flip();//人称"反转此缓冲区"。因为就像C语言中的文件指针一样到了整个缓冲区的最后，需要将它挪到最前面。

buffer.position() 得到最后的位置

buffer.clear();

int bytesReaded=socketChannel.read(buffer); //从套接字通道读取数据到缓冲区

socketChannel.write(buffer); //从缓冲区写入数据到套接字缓冲区

提供的这些方法就是ByteBuffer以及byte[]数组之间的区别

StringBuffer 类中的方法主要偏重于对于字符串的变化，例如追加、插入和删除等，这个也是StringBuffer和String类的主要区别。

常用函数：

append、insert、reverse、setCharAt、deleteCharAt、trimToSize

96.

char[] cChar=new char[]{'a','b'};

char[] c = new char[2];

//char[] cChar=new char[2]{'a','b'}; 是错误的，不能讲初始化与指定大小结合起来

98.使用get()和set()而不使用public的成员变量，是因为虽然两者看起来是效果上等价，但是在后期升级上面是不同的，比如加上限制判断，或者因为上面原因吧需要加上异常处理。特别是类的编写与使用部门不是同一个的时候。

99.StringBuilder

因为String是不可修改的类，所以每一次改动都会创建一个新的对象。这时不如使用同等地位的StringBuilder，其中的具体原理与C++中的vector一样， 会按照预定义的内存扩容方式，控制好内存缓冲区。（ArrayList数组同样也是实现了自动扩容）

sb.deleteCharAt(sb.length()-1); //删除最后一个字符

101.异或运算

a ^= b ^= a ^=b;

不等价于

a ^=b;

b ^= a;

a ^= b;

b ^= a ^=b;

a^=b; 这样子的两条式子却又是对的

这是因为编译器的优化，导致式子里面的两个a是同一个数值。于是经过了a ^= b ^= a ^=b; a一定会变成0。

102. 支持类里的声明时候的初始化：

public class Test {

int[] id =new int[120];

public static void main(String[] args) {

}

}

103. 在一个文件中使用另一个文件中定义的类，如果没有设置包名就可直接使用，如果设置了包名并且不是同一个包就需要import 或者使用的时候类名签名都加上包名。

104.ATM.java:10: 错误: 不兼容的类型: int无法转换为boolean

while(1) {

105. 字符串常量多行字符：

System.out.println("\nMain menu\n"+

"1:check balance\n"+

"2:withdraw\n"+

"3:deposit\n"+

"4.exit\n"+

"Enter a choice:");

可惜不能使用C、C++中在字符串后面使用一个\来包含下一行为本行扩展。那个功能是编译器特殊支持的。不知道编译器会不会帮你优化一下，最坏的情况是要运行时处理。

105.5. Java 中的网络支持：

InetAddress：用于表示网络上的硬件资源，即 IP 地址；

URL：统一资源定位符；

Sockets：使用 TCP 协议实现网络通信；

Datagram：使用 UDP 协议实现网络通信

net 比如URL编码与解码

106.Socket 需要使用io与net包中的相关类

TcpClient.java

import java.io.\*;

import java.net.\*;

public class TcpClient {

public static void main(String[] args) throws Exception{

String sentence;

String modifieddSentence;

BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

Socket clientSocket = new Socket("127.0.0.1",5678); //指定目标ip与端口

DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());

BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '\n');

modifieddSentence = inFromServer.readLine();

System.out.println("From Server:" + modifieddSentence);

clientSocket.close();

}

}

TcpServer.java

import java.io.\*;

import java.net.\*;

public class TcpServer {

public static void main(String argv[]) throws Exception{

String clientSentence;

String capitalizedSentence;

ServerSocket welcomeSocket = new ServerSocket(5678);

while(true) {

Socket connectionSocket = welcomeSocket.accept();

BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));

DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes(capitalizedSentence);

}

}

}

与C语言的步骤相比隐藏了listen、connect

UdpClient.java

import java.io.\*;

import java.net.\*;

public class UdpClient {

public static void main(String args[]) throws Exception{

BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

DatagramSocket clientSocket = new DatagramSocket();

InetAddress IPAdress = InetAddress.getByName("127.0.0.1");//因为不是绑定，所以生成一个对象，作为发送时候的单独的参数

byte[] sendData = new byte[1024];

byte[] receiveData = new byte[1024];

String sentence = inFromUser.readLine();

sendData = sentence.getBytes();

DatagramPacket sendPacket = new DatagramPacket(sendData,sendData.length,IPAdress,987);

clientSocket.send(sendPacket);

DatagramPacket receivePacket = new DatagramPacket(receiveData,receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence = new String(receivePacket.getData());

System.out.println("From Server:" + modifiedSentence);

clientSocket.close();

}

}

UdpServer.java

import java.io.\*;

import java.net.\*;

public class UdpServer {

public static void main(String args[]) throws Exception{

DatagramSocket serverSocket = new DatagramSocket(987); //作为服务器监听端口，所以就需要绑定，这一点与C不一样，C估计为了代码的对称性

byte[] receiveData = new byte[1024];

byte[] sendData = new byte[1024];

while(true) {

DatagramPacket receivePacket = new DatagramPacket(receiveData,receiveData.length);

serverSocket.receive(receivePacket);

String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket = new DatagramPacket(sendData,sendData.length,IPAddress,port);

serverSocket.send(sendPacket);

}

}

}

TcpClient先于TcpServer运行会抛出异常，UdpClient却可以先于UdpServer执行。

NIO技术：

IoAcceptor acceptor = new NioSocketAcceptor();//创建一个非阻塞的server端的Socket

107.java中访问修饰符 默认default

| **权限** | **类内** | **同包** | **不同包子类** | **不同包非子类** |
| --- | --- | --- | --- | --- |
| private | √ | × | × | × |
| default | √ | √ | × | × |
| protected | √ | √ | √ | × |
| public | √ | √ | √ | √ |

108.不同的机子上面class文件不一定可以运行，所以需要java重新javac。

110.Android开发中

Bitmap是一种类

Bitmap.Config也是一个公开类，通过这个类来设置Bitmap，引入构造函数

111.List与ArrayList的关系就像是Calendar与GeometricCalendar之间的关系。前者是抽象类，所以不能创建对象；后者是具体实现的类，所以一般都是通过前者来使用后者类的实例。Calendar的使用中存在一种使用方法：Calendar c =Calendar.getInstance()；实际上就是new GeometricCalendar()，构造了这个类的实例。

List list =new ArrayList<类名>();基本上不能直接在构造List的时候传入参数，目前只见过一种看起来有点像的方式：new ArrayList<类名>(Arrays.asList("123","123"));通过这种方式转为List之后传入进去。

113.接口：

[修饰符] interface 接口名 [extends 父接口名列表]{

[public] [static] [final] 常量;

[public] [abstract] 方法;

...比如static方法块

}

114.

在java语言中，注释有三种:

1) 单行注释： 用 "//" 开头，后跟注释内容。

2) 多行注释： 用 "/\*" 开头，"\*/" 结尾，其中包含多行注释内容。

3) 文档注释：　用 "/\*\*"开头，"\*/" 结尾，其中包含多行注释内容。

115.Java只能继承自一个类，要想多重继承，可以通过内部类（组合）来实现。https://www.cnblogs.com/haodawang/p/5967249.html

116.assert：https://www.cnblogs.com/wardensky/p/4307848.html

117.在Java虚拟机规范中试图定义一种Java内存模型（Java Memory Model，JMM）来屏蔽各个硬件平台和操作系统的内存访问差异,它定义了程序中变量的访问规则，往大一点说是定义了程序执行的次序。注意，为了获得较好的执行性能，Java内存模型并没有限制执行引擎使用处理器的寄存器或者高速缓存来提升指令执行速度，也没有限制编译器对指令进行重排序。也就是说，在java内存模型中，也会存在缓存一致性问题和指令重排序(编译器想要优化程序而按照数据依赖性对指令重排，可是多线程代码似乎会出错)的问题。

（原子性）在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行：

x = 10; //语句1

y = x; //语句2

x++; //语句3

x = x + 1; //语句4

上面四条语句中只有语句1是原子操作。语句2实际上包含2个操作，它先要去读取x的值，再将x的值写入工作内存，虽然读取x的值以及 将x的值写入工作内存 这2个操作都是原子性操作，但是合起来就不是原子性操作了。同样的，x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。由这些语句可以知道volatile不能保证原子性。

（可见性）Java提供了volatile关键字来保证可见性。当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

（指令有序性）在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。（其实就是可见性）。

volatile关键字禁止指令重排序有两层意思：

1) 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；（写操作都需要前面的变量，同时会对后面的变量有影响；但是volatile读不会受影响）相当于一个顺序点

java的分号（语句结束符号）不是一个顺序点

2) 在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

//x、y为非volatile变量

//flag为volatile变量

x = 2; //语句1

y = 0; //语句2

flag = true; //语句3

x = 4; //语句4

y = -1; //语句5

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。volatile关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”，volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

C++ 没有使用像jvm一样分散成jvm内存与各线程内。所以volatile，只需要避免编译器将从内存读取优化成为从寄存器读取（就是前面指令的结果，或者读取的数据）。

synchronized 加到 static 方法前面是给class 加锁，即类锁；而synchronized 加到非静态方法前面是给对象上锁。synchronized(this)是对象锁，如果有多个对象就有相对应的多个锁，synchronized(类的名.class)是类锁，不管有几个对象就公用一把锁。

synchronized是可重入锁。两段synchronized嵌套就能够验证。

可重入函数是指对外界没有变化（比如不会改变全局变量的值）。

119. .class（属性成员） 返回的类型是java.lang.Class

120. 如果return在try里面，return会自动关闭try的监视。而且应该是在return发生之前。

121.类中非静态类可以调用外边类的“类名.this”（this其实也是一个属性成员）。就像C结构体或者类里可以创建与本类类型相关的对象，Java也可以。

122.虽然号称Java的&&也是有结合性，但是 前面已经判断出false了，它还是会继续去判断后面， 不同C语言。可以用一个对象去测试：

if(false && obj.xxx)

xxx

else

obj =new Classxxx();

而且java里面的&&并不是一个顺序点，所以不要相信i && i++的作用。结合性体现在编译器或者解释器是如何解析表达式，但并不意味这对内部表达式的计算顺序。

123.多线程中的中断（结束）函数：

首先，一个线程不应该由其他线程来强制中断或停止，而是应该由线程自己自行停止。所以，Thread.stop, Thread.suspend, Thread.resume 都已经被废弃了。而 Thread.interrupt 的作用其实也不是中断线程，而是「通知线程应该中断了」，具体到底中断还是继续运行，应该由被通知的线程自己处理。具体来说，当对一个线程，调用 interrupt() 时，① 如果线程处于被阻塞状态（例如处于sleep, wait, join 等状态），那么线程将立即退出被阻塞状态，并抛出一个InterruptedException异常并将中断标志位设置true，这时候调用thread.interrupted()会清除线程中断标志位。② 如果线程处于正常活动状态，那么会将该线程的中断标志设置为 true，仅此而已。被设置中断标志的线程将继续正常运行，不受影响。interrupt() 并不能真正的中断线程，需要被调用的线程自己进行配合才行。也就是说，一个线程如果有被中断的需求，那么就可以这样做。① 在正常运行任务时，经常检查本线程的中断标志位，如果被设置了中断标志就自行停止线程。② 在调用阻塞方法时正确处理InterruptedException异常。（例如，catch异常后就结束线程。）

支持中断的线程编写：

Thread thread = new Thread(() -> {

while (!Thread.interrupted()) {

// do more work.

}

});

thread.start();

// 一段时间以后

thread.interrupt();

thread.interrupted() isInterrupted()+清除中断标志位《=》isInterrupted(true)

thread.isInterrupted() 根据线程的中断标志位，返回当前中断信息《=》isInterrupted(false)

124.DecimalFormat 可以限制float、double等的位数。

DecimalFormat fnum = new DecimalFormat("##0.00");

fnum.format(Float.parseFloat("12.3456"))

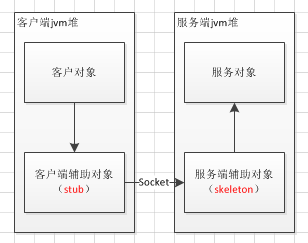
或者使用String对象的format函数

126.Java 远程方法调用RMI(Remote Method Invocation)：

<https://blog.csdn.net/a19881029/article/details/9465663>

RMI与RPC的关系： 两者都可以理解为进程间的Socket通信  
  RMI只用于Java；RPC是网络服务协议，与操作系统和语言无关。

**RMI中是通过在客户端的Stub对象（客户端辅助对象）作为远程接口进行远程方法的调用**。**每个远程方法都具有方法签名**。如果一个方法在服务器上执行，但是**没有相匹配的签名被添加到这个远程接口(stub)上，那么这个新方法就不能被RMI客户方所调用**。 能够让在某个java虚拟机上的对象像调用本地对象一样调用另一个java 虚拟机中的对象上的方法，使用socket实现。



**RPC中是通过网络服务协议向远程主机发送请求**，请求包含了一个参数集和一个文本值，通常形成“classname.methodname(参数集)”的形式。RPC远程主机就去搜索与之相匹配的类和方法，找到后就执行方法并把结果编码，通过网络协议发回。与RMI相比少了两个辅助对象。

远程对象RPC补充：

 首先创建远程接口并声明远程方法，需要继承自java.rmi.Remote（Client和Server需要共享这个接口）；

 创建远程接口的实现类，这个类必须继承自java.rmi.server.UnicastRemoteObject（只有Server需要这个实现类，Client可以没有）；

 编写Server，绑定端口，注册对象；

 编写Client，监听端口并查找对象。

129. Java单例:https://blog.csdn.net/goodlixueyong/article/details/51935526、https://blog.csdn.net/mnb65482/article/details/80458571， 推荐使用文章中介绍的方法3、4、5,尤其是最后一种通过枚举来实现。补充懒汉模式可使用static synchronized。使用枚举除了线程安全和防止反射调用构造器之外，还提供了自动序列化机制，防止反序列化的时候创建新的对象。

单例中构造函数都需要设置为private修饰符；饿汉模式意为有没有使用都会被创建（其实暗中使用了类的初始化锁来保证不会多个线程创建多个实例）；懒汉模式意为只有被使用的时候才会创建，所以这时候还要考虑多个线程之间的锁关系；内部静态类模式是对饿汉进行了一层静态方法的封装，只有调用方法的时候才会创建变量。

单例中使用null判断静态变量是否创建完成，比如双重校验方式，由于指令重排优化的存在，导致初始化Singleton和将对象地址赋给instance字段的顺序是不确定的，所以需要加上volatile修饰。

Java scope 分为单例与多例：

singleton作用域：当把一个Bean定义设置为singleton作用域是，Spring IoC容器中只会存在一个共享的Bean实例，并且所有对Bean的请求，只要id与该Bean定义相匹配，则只会返回该Bean的同一实例。值得强调的是singleton作用域是Spring中的缺省作用域。

prototype作用域：prototype作用域的Bean会导致在每次对该Bean请求（将其注入到另一个Bean中，或者以程序的方式调用容器的getBean()方法）时都会创建一个新的Bean实例。根据经验，对有状态的Bean应使用prototype作用域，而对无状态的Bean则应该使用singleton作用域。

Dagger2 Scope分为：

Dagger 2默认并不提供@ActivityScope 、 @ApplicationScope 这些注解。这些只是最常用的自定义Scope。只有@Singleton scope是默认提供的（由Java自己提供）

@Singleton可以保持类的单例。

@ApplicationScope注解的Component类与Applicaiton对象的生命周期一致。

@ActivityScope注解的Component类与Activity的生命周期一致

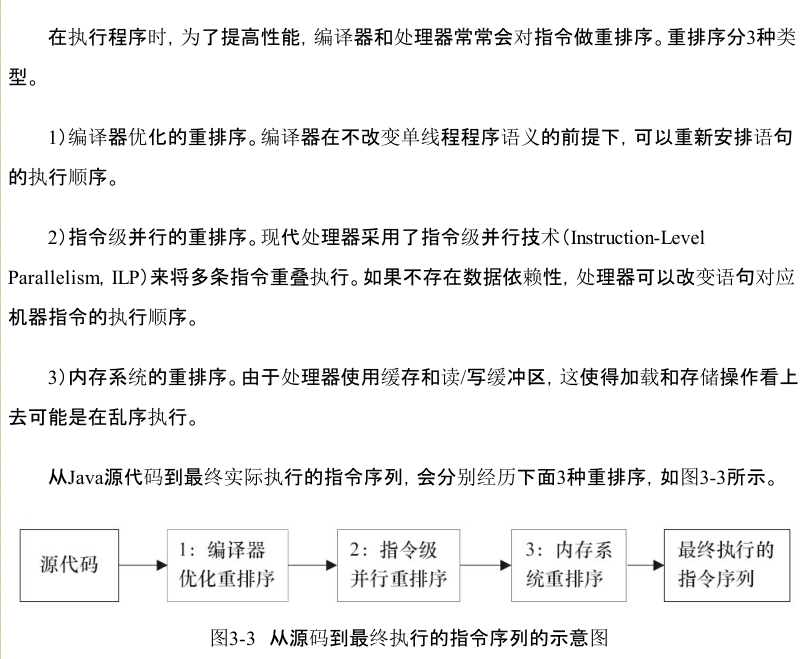
130.JVM

java 有着商业虚拟机：sun系列的HotSpot（使用最广的虚拟机）（有一个永久代）、JRockit虚拟机、IBM J9虚拟机

《Java虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。在HotSpot上**使用永久代来实现方法区**。永久代（JDK 1.8 后HotSpots消失），因为这个代不会被回收。元空间是JDK 1.8 之后HotSpots才有的，功能和永久代类似。唯一到的区别是，永久代使用的是JVM的堆内存空间，而元空间使用的是物理内存，数据直接受到本机的物理内存限制，元空间存储类的元信息，静态变量和常量池等并入堆中。

HotSpot VM的自动内存管理要求对象起始地址必须是8字节的整数倍。对象头（Mark Word）本身是8的倍数，当对象的实例变量数据不是8的倍数，便需要填充数据来保证8字节的对齐。

1. Java协程： kilim框架（过时）、Quasar框架
2. happens-before



指令级重排序、内存系统重排序 可以合并称为 处理器重排序。在处理器层次 决定 内存模型的实现程度

happens-before（as-if-serial） **执行结果可见**并不能代表两者是同一个线程。

主要通过**内存屏障**实现

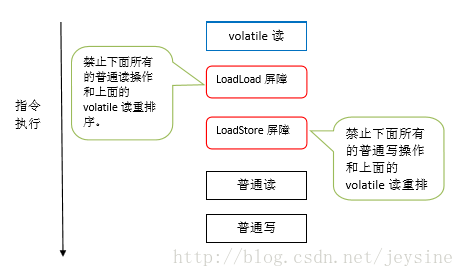
**监视器锁规则**： 对一个锁的解锁，happens-before 于随后对这个锁的加锁（没有加锁就解锁，或者通过共享对象，在一个线程中加锁另外一个线程中解锁，都会触发java.lang.IllegalMonitorStateException）

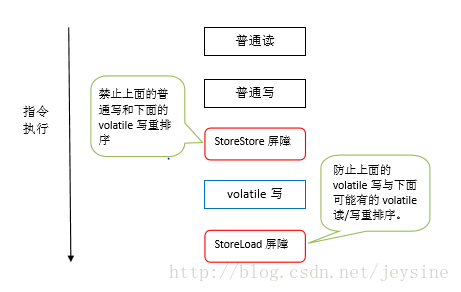
**volatile变量规则**：

普通变量可以当做线程内环境变量；这些规则对普通变量和volatile变量进行了区分，但是在屏障的时候只看读写操作

小结：

1. 当第一个操作是Volatile读时，不管第二个操作是什么，都不能重排序；
2. 当第一个操作是Volatile写时，第二个操作是Volatile读或写，不能重排序；
3. 当第一个操作是普通读写，第二个操作是Volatile写时，不能重排序





StoreLoad包含了所有屏障的效果

**final对重排序的影响**

**CAS （相当于同时实现了volatile读写内存语义，不过不是从内存角度解决问题，而是从CPU的调度方面）** intel X86处理器，程序会根据当前处理器的类型来决定是否为cmpxchg指令添加lock前缀。如果程序是在多处理器上运行，就为cmpxchg指令加上lock前缀（Lock Cmpxchg）。反之，如果程序是在单处理器上运行，就省略lock前缀（单处理器自身会维护单处理器内的顺序一致性，不需要lock前缀提供的内存屏障效果）。Lock 前缀的指令会在执行期间锁住总线。

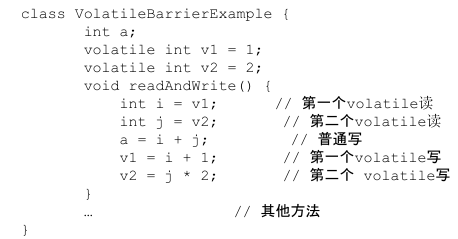
JMM 提供了四种内存屏障： 实际上通过一系列指令实现

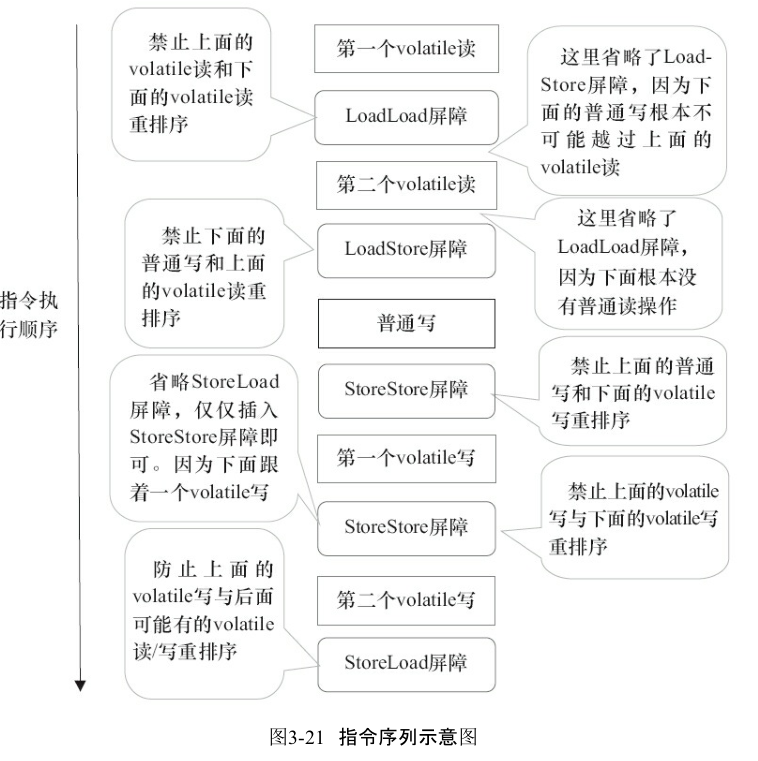
LoadLoad 、StoreStore 、StoreLoad（包含其他三个效果） 、LoadStore

四种功能对应的名字，比如storestore：

序列：Store1，StoreStore，Store2 确保Store1的数据在Store2以及后续Store指令操作相关数据之前对其它处理器可见（例如向主存刷新数据）。

JMM默认是采用保守策略，但是指定了一部分的优化情况，从而去掉不必要的内存屏障：





最后的StoreLoad 不能省略，因为完成写之后方法立即return，编译器并不能判断之后是否还存在volatile读或者写（return语句中的表达式）。上面的优化针对任意处理器平台，由于不同的处理器有不同“松紧度”的处理器内存模型。

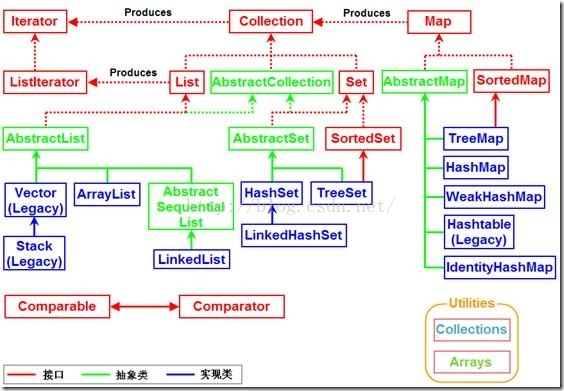
X86处理器支持两种指令：

x86上的”ifence“指令，强制所有在load屏障指令之后的load指令

x86上的”mfence“指令，复合了load和save屏障的功能

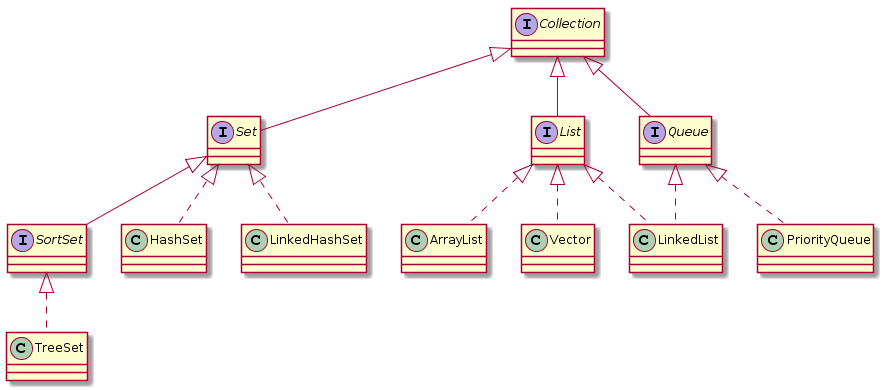
为了满足标准定义的内存语义，估计缺失的几种内存屏障会全用mfence实现。

134.

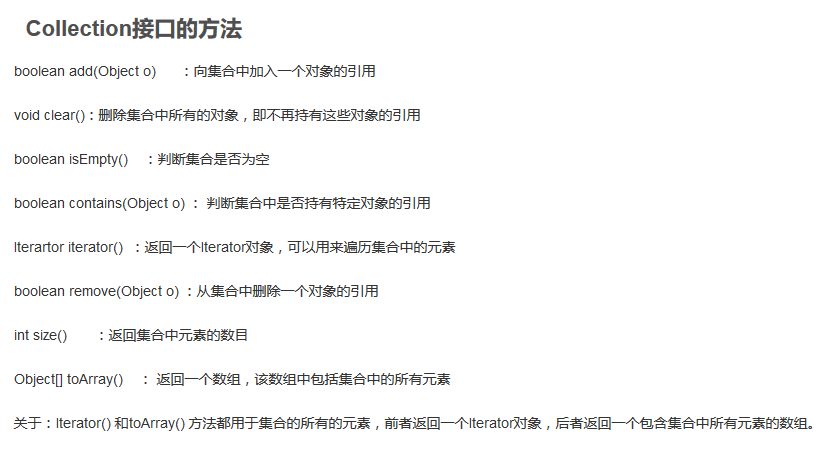


Map 并不是从Collection那里实现过来的，两个是平级的接口。





上图的Collection接口声明了Map、List、Queue通用的方法：



Collections.sort(object,cmp\_function)

Collections.synchronizedMap()方法来获取一个线程安全的Collections（Collections.synchronizedMap()实现原理是Collections定义了一个SynchronizedMap的内部类，这个类实现了Map接口，会在调用方法时使用synchronized来保证线程同步

**fail-fast 机制是java Collection中的一种错误机制，基于modCount变量实现。**当多个线程对同一个集合的内容进行操作时，就可能（尽最大努力）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。暗中做了很多事情呀。

所以之后Set接口只是再此基础上定义了Set独有的方法，具体实现对应的Set类的时候会加上对应的Map类，所以内部元素存储的顺序不能够被保证，同时元素不能传入空值，但是可以存入一个null作为元素。如果HashMap使用put函数，存入key一样的内容，会是覆盖效果。

对照C++的Set实现：

set，不像map那样是key-value对，它的key与value是相同的。关于set有两种说法，第一个是STL中的set，用的是红黑树；第二个是hash\_set（或者unordered\_set），底层用得是hash table。红黑树与hash table最大的不同是，红黑树是有序结构（支持排序输出），而hash table不是。但不是说set就不能用hash，如果只是判断set中的元素是否存在，那么hash显然更合适，因为set 的访问操作时间复杂度是log(N)的，而使用hash底层实现的hash\_set是近似O(1)的。然而，set应该更加被强调理解为“集合”，而集合所涉及的操作并、交、差等，即STL提供的如交集set\_intersection()、并集set\_union()、差集set\_difference()和对称差集set\_symmetric\_difference()，都需要进行大量的比较工作，能够直接使用基于hash的set一定是最快的。

如果要实现自己的比较函数传入set等：

class CTest {

public:

        CTest() { num = 0; str = ""; }

        CTest(int \_num, string \_str) { num = \_num; str = \_str; }

        string str;

        int num;

};

class CTestCmp {

public:

        bool operator() (const CTest& lc, const CTest& rc) {

                //return !!(lc.num < rc.num);

                return !!(lc.str < rc.str);

        }

};

typedef set<CTest, CTestCmp> CTestSet;

TreeSet因为TreeMap是通过红黑树（方便插入，又有二叉搜索的时间复杂度）实现，虽然查找效率上没有另外几个Map效果好，但是提供一个按照排序输出的功能，默认是升序（字母序）。排序的顺序与构造时候传入的函数有关系：

Map< Integer, String> map=new TreeMap<>(new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

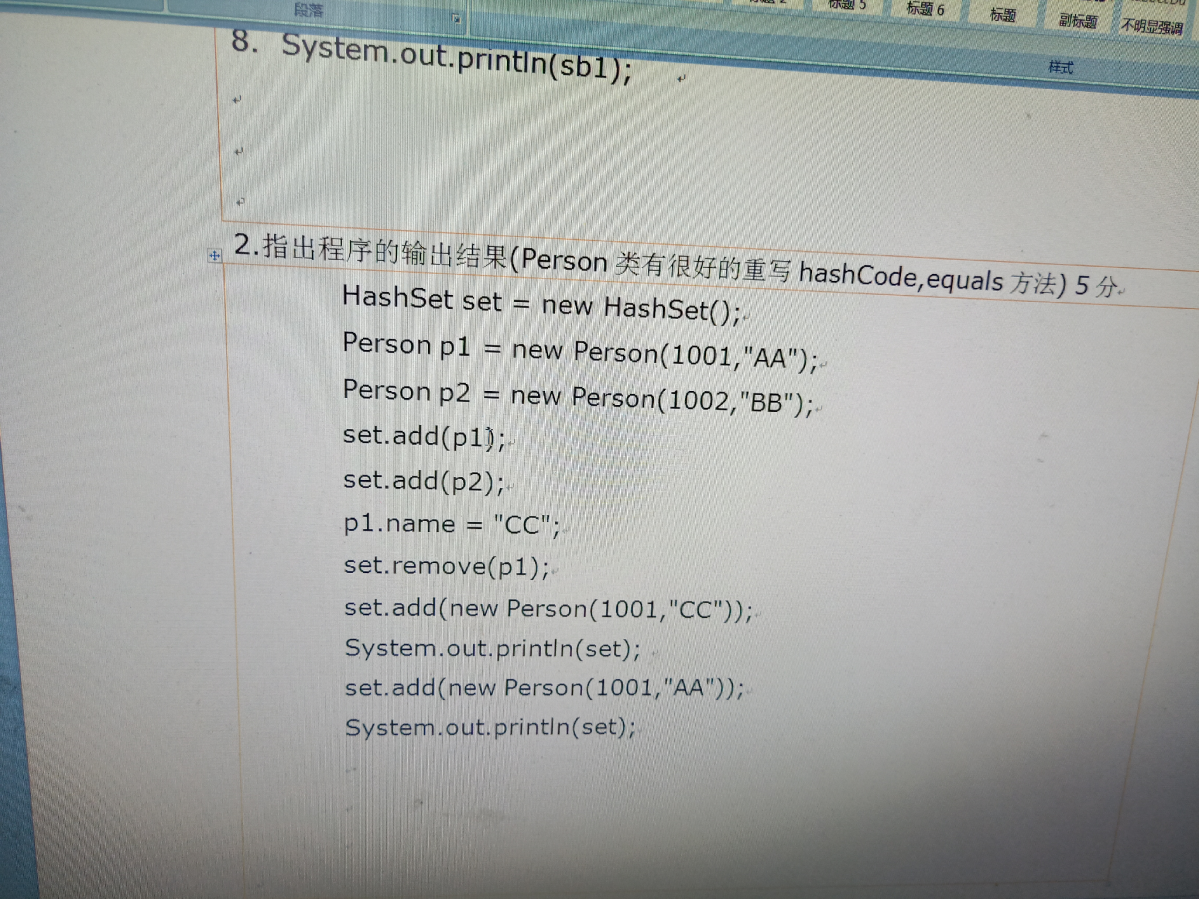
// return o1.compareTo(o2); //按照key值大小升序排列 -o1.compareTo(o2)即o2.compareTo(o1)按照key值大小降序排列

return -1; //按put反顺序排序 1则为 put顺序排列

}

});

HashSet（不存在重复key的元素，因为是Set）的时候，存储的位置就是通过对象的hashCode函数决定。存储的结构，可以类比HashMap（其实内部通过HashMap实现），所以下面这个问题的答案是四个对象都存在在里面：



Hash判断两个对象是否相等有两个准则，首先看hashCode的返回值，其次看equals函数。首先判断hashCode的返回值，估计是为了效率问题，比如String对象的返回值专门有一个成员变量存储，但是同一个hashCode不一定就是同一个对象，因为存在碰撞的问题，所以还要第二条准则。

LinkedHashSet 继承自HashSet，对象内部元素存储的时候使用双向链表实现。构造的时候可以传入一个容量参数，否则默认容量是16 ，典型的构造函数：

new LinkedHashMap<String, Bitmap>(0, 0.75f, true);

第一个参数是初始容量，如果实参小于1，那么程序内部实现的时候就是1；如果大于，就会不断x2翻倍直到大于人为设置的大小**：**

while (capacity < initialCapacity)

          capacity <<= 1;  // 所以最后的capacity一定会是2的倍数

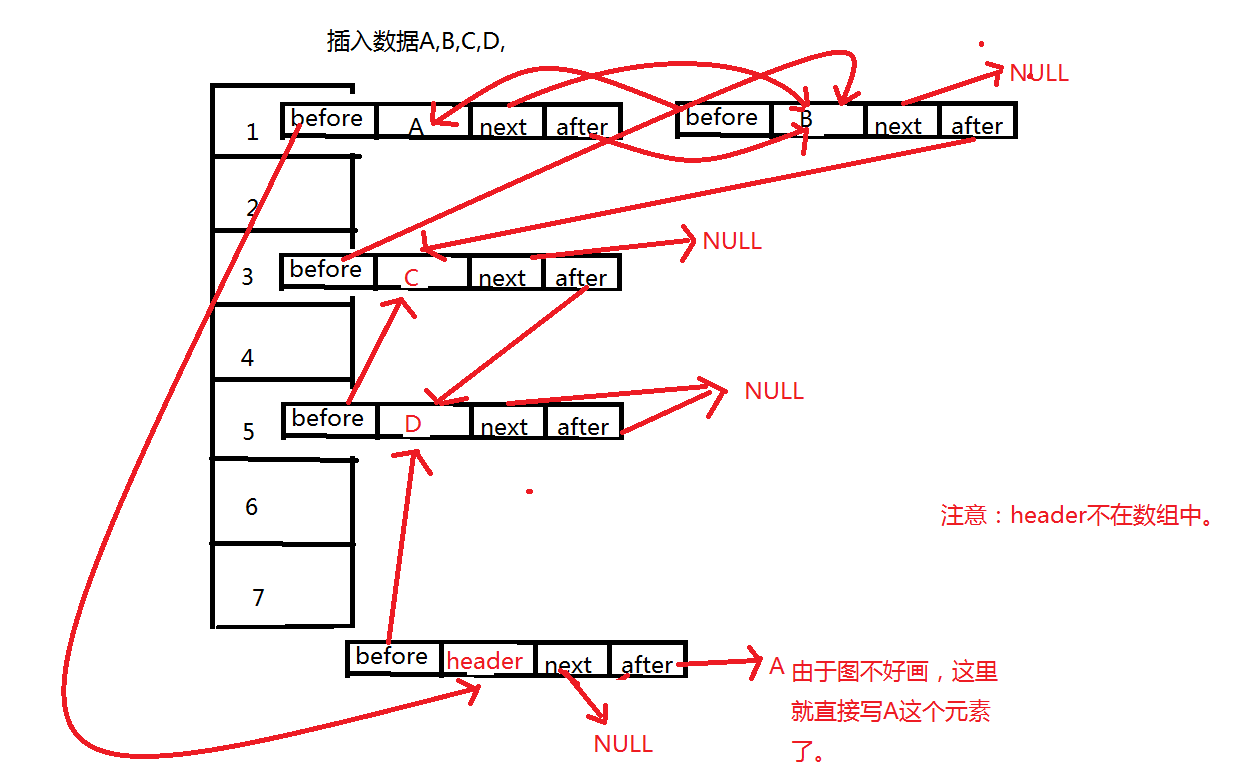
第二个参数 负载因子（threshold=(int)(capacity \* loadFactor)）

第三个参数false： 基于插入顺序     true：  基于访问顺序

在insert模式下，虽然是链表实现，但是因为是Set所以每一次插入都是末尾插入，于是遍历顺序就是插入顺序。

在access模式下实现的是LRU算法， 链表的insert和search因为HashMap使用基于地址的hashCode函数，于是都是O(1)代价的方案。在access模式中，每一次访问的元素会被移到队尾，removeEldestEntry要删除的元素是队首，所以为了效率存在记录队首与队尾的变量。

是不是很神奇，HashMap可以和有序链表结合在一起：



实现LRU缓存：

Java里面实现LRU缓存通常有两种选择，一种是使用LinkedHashMap，一种是自己设计数据结构，使用链表+HashSet。

class LRUCache<K, V> extends LinkedHashMap<K, V> {

private static final int MAX\_ENTRIES = 3;

@override

protected boolean removeEldestEntry(Map.Entry eldest) {

return size() > MAX\_ENTRIES;

//因为我们的重写改变了size上限，但是size的最大值并不是3哦，到底能够多大与容量有关

}

LRUCache() {

super(MAX\_ENTRIES, 0.75f, true);

}

}//默认实现的get会记录访问次数，并且按照模式看是否调用removeEldestEntry删除最老的元素，所以只需要重写removeEldestEntry。LRU为最近最少使用算法，将超过范围的时候，符合要求的元素移出去。

Android、Linux中经常使用LruCache 、DiskLruCache，比如glide使用这几个分别作为内存、磁盘缓存算法。

DiskLruCache算法原理：

journal文件是DiskLruCache的一个日志文件，程序对每张图片的操作记录都存放在这个文件中，基本上看到journal这个文件就标志着该程序使用DiskLruCache技术了。其中内容：

libcore.io.DiskLruCache

1

100

2

CLEAN 3400330d1dfc7f3f7f4b8d4d803dfcf6 832 21054

DIRTY 335c4c6028171cfddfbaae1a9c313c52

CLEAN 335c4c6028171cfddfbaae1a9c313c52 3934 2342

REMOVE 335c4c6028171cfddfbaae1a9c313c52

DIRTY 1ab96a171faeeee38496d8b330771a7a

CLEAN 1ab96a171faeeee38496d8b330771a7a 1600 234

READ 335c4c6028171cfddfbaae1a9c313c52

READ 3400330d1dfc7f3f7f4b8d4d803dfcf6

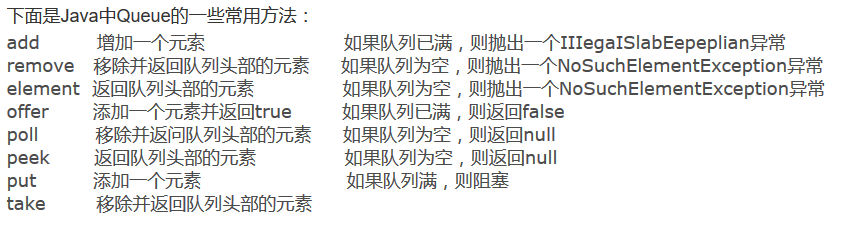
第一行固定为libcore.io.DiskLruCache；第二行是DiskLruCache的版本，目前固定为1；第三行表示所属应用的版本号；第四行valueCount表示一个缓存key可能对应多少个缓存文件，它决定了后面一个CLEAN状态记录最多可以size大小数据；第五行是空行。此后记录的就是DiskLruCache针对磁盘缓存的操作记录了。其中几个状态表示如下：

* CLEAN 表示缓存处于一个稳定状态，即当前没有对该缓存数据进行写操作，在该状态下，对缓存文件的读写都是安全的。
* DIRTY 表示当前该key对应的缓存文件正在被修改，该状态下对缓存文件的读写都是不安全的，需要阻塞到对文件的修改完成，使该key对应的状态转变成CLEAN为止。
* REMOVE 表示该key对应的缓存文件被删除了，在缓存整理的过程中可能会出现多条这样的记录。
* READ 表示一个对key对应的缓存文件进行读取的操作记录。

Glide中一个key可能对应多个文件，因为一张图片可能存在多个不同的分辨率与大小。

通过阅读日志文件回到之前程序退出之前的缓存状态。

Queue接口实现：



可以看到添加元素的offer、add分别在失败的时候会返回不同的值或者抛出异常。

Deque 是java内部提供的双向队列，不推荐向其中插入null，因为很多方法使用null判断这个队列是否为空，比如peek、poll。

Queue<Integer> result = new PriorityQueue<>();

....

Integer i;  
while( (i=result.poll())!=null ) {  
 System.*out*.print(i+" ");  
}

HashTable 是HashMap的线程安全版本，通过synchronized实现线程安全，这意味着所有的线程竞争的是一把锁，于是效率低下。它是遗留类，不应该去使用它。

HashTable的键或者值不允许null，HashMap允许。

Queue<String> queue = new LinkedList<String>();

上面将一个List传递给一个Queue，不过一般使用应该是ArrayDuque

PriorityQueue 优先队列默认通过小顶堆实现，如果要实现大顶堆，需要在构造的时候传入一个自己定义的compare函数（C++一样的原理）：

PriorityQueue <Integer> maxHeap = new PriorityQueue<Integer>(11, new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

// TODO Auto-generated method stub

return o2.compareTo(o1);

}

}); //推荐传入lambda函数

因为这种奇特的实现方式，所以不能能够直接从末尾选中元素。

J.U.C

ConcurrentSkipListMap 支持并发跳跃表

Semaphore 信号量

ConcurrentLinkedQueue 使用非阻塞的方式实现了线程安全（CAS）

**入队**：

如果有一个线程正在入队，那么它必须先获取尾节点，然后设置尾节点的下一个节点为入队节点，但这时可能有另外一个线程插队了，那么队列的尾节点就会发生变化，这时当前线程要暂停入队操作，然后重新获取尾节点。源码中死循环加CAS一直到成功入队。

tail节点并不总是尾节点，所以每次入队都必须先通过tail节点来找到尾节点。尾节点可能是tail节点，也可能是tail节点的next节点。通过CAS操作与死循环操作：

for (;;) {

Node<E> t = tail;

if (t.casNext(null, n) && casTail(t, n)) { //前者是判断是否t还是尾节点，后者是判断是否能够将尾节点设置为指向一个新的对象

return true; //所以不能按照返回值为true判断是否入队成功

}

}

为了能够提高入队效率，减少CAS操作，引入一个hops变量：

private static final int HOPS = 1;

当tail节点和真实尾节点的距离大于等于常量HOPS的值（默认等于1）时才更新tail节点（取消了casTail操作），tail和尾节点的距离越长，使用CAS更新tail节点的次数就会越少，但是距离越长带来的负面效果就是每次入队时定位尾节点的时间就越长。增加对volatile变量的读操作来减少对volatile变量的写操作。

**出队**：  
 需要更新head节点的指向，同时移除原先的节点。

BlockingQueue 可以阻塞操作的有界队列



抛出异常：队列满如果还是插入元素就会抛出异常；空的时候取元素也会抛出异常。

返回特殊值：当往队列插入元素时，会返回元素是否插入成功，成功返回true。如果是移除方法，则是从队列里取出一个元素，如果没有则返回null。

下面几种阻塞队列都是线程安全的： 使用volatile、CAS、ReentrantLock等实现

·ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。基本都是无界队列，因为无界其实可以自己控制为有界队列，ArrayBlockingQueue估计底层实现是直接分配一个数组，所以有界，且不打算不断重新调整大小。

·LinkedBlockingQueue：一个由链表结构组成的无界阻塞队列。

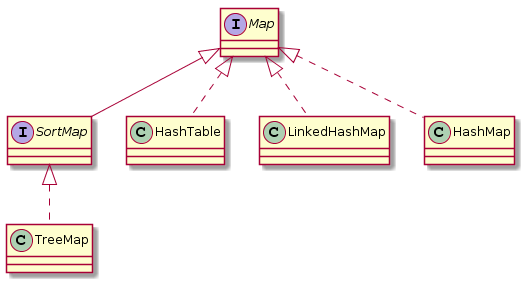
·PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列，可以自定义类实现compareTo()方法来指定元素排序规则，或者初始化PriorityBlockingQueue时，指定构造参数Comparator来对元素进行排序。

·DelayQueue：一个支持延时获取元素的无界阻塞队列。在创建元素时可以指定多久才能从队列中获取当前元素，只有在延迟期满时才能从队列中提取元素。使用这种性质可以实现定时任务调度。内部按照优先队列，按照时间进行排序，队头的到期时间最长（最需要被处理），所以只需要一个变量来计时就好。用于放置实现了Delayed接口的对象。

·SynchronousQueue：一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。可以同时触发多个put操作，这时候每一个都会进入阻塞。take 从队头获取数据。

·LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。相对于其他阻塞队列，LinkedTransferQueue多了tryTransfer和transfer方法。如果没有消费者等待会将元素放入队列，元素的非公平性。这种算法都需要一个tryTransfer 来测试。

·LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。无界队列，但是可以设置容量。



HashEntry 用来封装散列映射表中的键值对。

Hashmap采用桶链表（从 JDK 1.8 开始，一个桶存储的链表长度大于 8 时会将链表转换为红黑树，小于6会重新转成链表存储，不是小于等于8留下两个空间作为缓冲来减少变换次数）来解决hash冲突，在多线程的情况下容易导致链表出现环路（几个线程同时进行HashMap的rehash操作就容易出现），于是触发get函数时候的死循环，导致CPU一直被占用。

HashMap内部的几个常量：

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4;//默认的桶数组大小

static final int MAXIMUM\_CAPACITY = 1 << 30;//极限值

static final int tableSizeFor(int cap) { //threshold更新函数，判断最高位的1到哪里了，还不如直接用与运算

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;//负载因子（加载因子）

0.75×16 = 12 当元素数量达到这个值的时候就会启动扩容函数，调整容量为当前的两倍，同时rehash。因为担心hash碰撞所以负载因子不能够过大。capacity、size、threshold、loadFactor都是针对于table。

rehash优化：

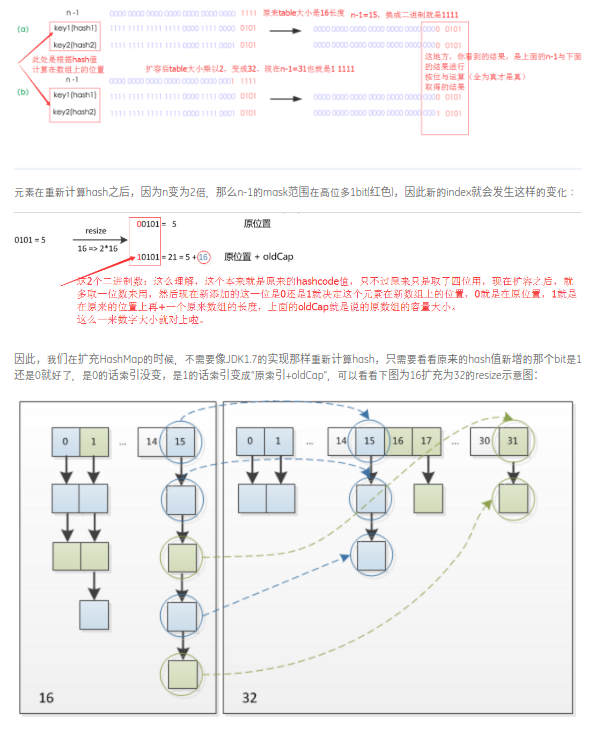
假设原数组长度 capacity 为 16，扩容之后 new capacity 为 32：

capacity : 00010000

new capacity : 00100000

对于一个 Key，

* 它的哈希值如果在第 5 位上为 0，那么取模得到的结果和之前一样；
* 如果为 1，那么得到的结果为原来的结果 +16，部分元素直接移动16即可



内部有一个变量transient Entry[] table;桶链表是从这个数组上引出来，

HashMap实现了Serializable接口，但是这个关键的成员变量却是transient，是因为HashMap已经实现了自己的readObject、writeObject函数，再调用的时候不会再用系统的函数defaultWriteObject()和defaultReadObject()，所以加上这个关键字让别人更加明白点。之所以需要自己进行处理，是因为在不同的JVM中，得到的hash值不一定相同，意思就是在windows下的虚拟机将key=‘1’计算出来的hash值可能是存在table的第0个位置的，但是在linux环境下的虚拟机计算出来的key=‘1’的hash值可能是放在table的第1个位置，当我们去读table中的值的时候未必能拿到key=’1’的值。在反序列化的时候，会严格按照index存入value。

int hash = hash(key);

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}//我认为^，并不是一个顺序点，所以上面的程序有问题

int i = indexFor(hash, table.length); //通过indexFor确定桶的数组下标

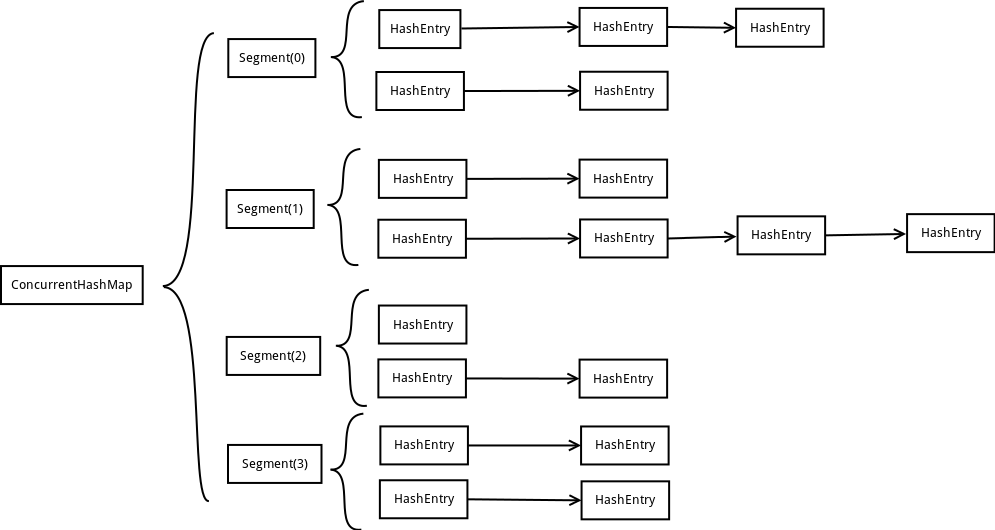
static int indexFor(int h, int length) { return h & (length-1); } //并不算取余操作，是截取操作，因为hashmap的长度一定是2的整数倍。当n为2的次幂时候有(n - 1) & hash == hash % n，通过位运算加快取模，所以容量一定会是2的幂次。

HashMap来说，我们都知道只要key的值相同（严谨说法是：key.equals(k)，这个方法中有时候也是==实现的） 那么我们认为他们是同一个可以Entry。对于IdentityHashMap则不同，只承认key==e.key（对象的地址）的结果为true时，才认为是相同的Entry，可能hash的时候将地址也考虑进去了（支持存储一样的值）。

hashmap以数组为value：

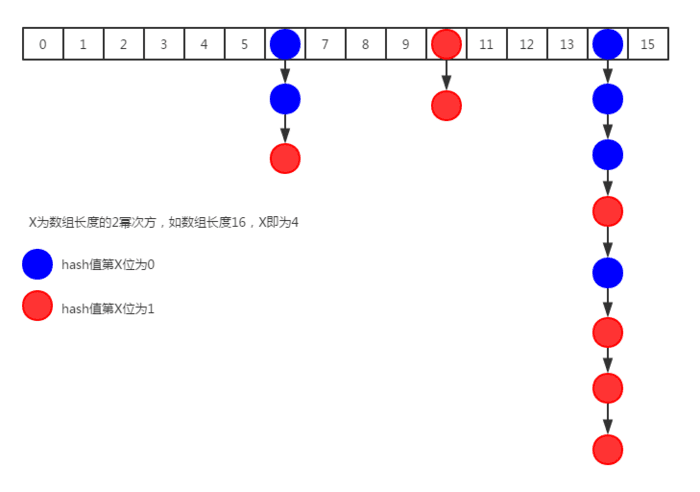
public class Main{  
 public static void main(String[] args) {  
 HashMap<Integer,int[]> map= new HashMap<Integer, int[]>();  
 int[] array = {1,2};  
 map.put(Integer.*valueOf*(2),array); // array部分不能使用数组常量代替  
   
 }  
}

ConcurrentHashMap JDK1.7之前使用segment（继承自ReentrantLock）实现线程安全（默认16个Segment锁，每一段4个HashEntry，将目标hash到这个二维数组上），这时候的结构是下面这样： ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护着一个HashEntry数组里的元素（每一个数组元素都是一个链表，碰撞的HashEntry链接而成）,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。



每个 Segment 维护了一个 count 变量来统计该 Segment 中的键值对个数(用于变换红黑树与链表结构)，执行size的时候并不是返回一个成员变量，而是遍历所有的segment，将count累加。统计ConcurrentHashMap 整体大小的时候（将所有的segment中的count相加），但是多线程可能存在count即时变化的问题，先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化（使用modCount 判断），则再采用加锁的方式来统计所有Segment的大小。

插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置，然后将其放在HashEntry数组里。 所以第一步如果因为插入的元素已经存在的，就会导致无效扩容。为了高效（不要锁住整个对象），ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment的HashEntry数组进行扩容：



上图中红色节点会因为扩容被移动，帮助扩容是指帮助节点的移动（链表或者红黑树的重建）。一个程序扩容的时候，会锁住一个HashEntry，这时候如果有其他线程来取这个HashEntry下的节点，会发现处于扩容状态，于是帮助扩容（不然相当于没有事情做，阻塞）（1.8后加入）。

1.8之后使用volatile、CAS等实现线程安全，如果CAS操作失败会使用synchronized，但是我不确定。去掉了分段锁，将锁的级别控制在了更细粒度的HashEntry首元素。

因为不是使用分段，所以扩容的时候，如果其他线程读写操作会被阻塞。优化是让闲着的线程一起参与扩容，于是在JDK8的源码里面就引入了一个ForwardingNode类，在一个线程发起扩容的时候，就会改变sizeCtl这个值，其含义如下：

sizeCtl ：默认为0，用来控制table的初始化和扩容操作，具体应用在后续会体现出来。

-1 代表table正在初始化

-N 表示有N-1个线程正在进行扩容操作

其余情况：

1、如果table未初始化，表示table需要初始化的大小。

2、如果table初始化完成，表示table的容量，默认是table大小的0.75倍

在此期间如果其他线程的有读写操作都会判断head节点是否为forwardNode节点，如果是就帮助扩容。

WeakHashMap 的 Entry 继承自 WeakReference，被 WeakReference 关联的对象在下一次垃圾回收时会被回收。主要用来实现缓存，通过使用 WeakHashMap 来引用缓存对象（Tomcat 中的 ConcurrentCache），由 JVM 对这部分缓存进行回收。private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V>

Tomcat的ConcurrentCache 采取的是分代缓存：

* 经常使用的对象放入 eden 中，eden 使用 ConcurrentHashMap 实现，不用担心会被回收（伊甸园）；
* 不常用的对象放入 longterm，longterm 使用 WeakHashMap 实现，这些老对象如果没有其余强引用指向会被垃圾收集器回收。
* 当调用 get() 方法时，会先从 eden 区获取，如果没有找到的话再到 longterm 获取，当从 longterm 获取到就把对象放入 eden 中，从而保证经常被访问的节点不容易被回收。
* 当调用 put() 方法时，如果 eden 的大小超过了 size，那么就将 eden 中的所有对象都放入 longterm 中。

Map.entrySet() 这个方法返回的是一个Set<Map.Entry<K,V>>，Map.Entry 是Map中的一个接口，他的用途是表示一个映射项（里面有Key和Value），而Set<Map.Entry<K,V>>表示一个映射项的Set。Map.Entry里有相应的getKey和getValue方法，即JavaBean，让我们能够从一个项中取出Key和Value。

135.AQS 框架 java.util.current

一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch... AQS提供了大量的模板方法来实现同步，主要是分为三类：独占式获取和释放同步状态、共享式获取和释放同步状态、查询同步队列中的等待线程情况。

/\*\* Marker to indicate a node is waiting in exclusive mode \*/

static final Node EXCLUSIVE = null;

/\*\* waitStatus value to indicate thread has cancelled \*/

static final int CANCELLED = 1;

/\*\* waitStatus value to indicate successor's thread needs unparking \*/

static final int SIGNAL = -1;

/\*\* waitStatus value to indicate thread is waiting on condition \*/

static final int CONDITION = -2;

static final int PROPAGATE = -3;

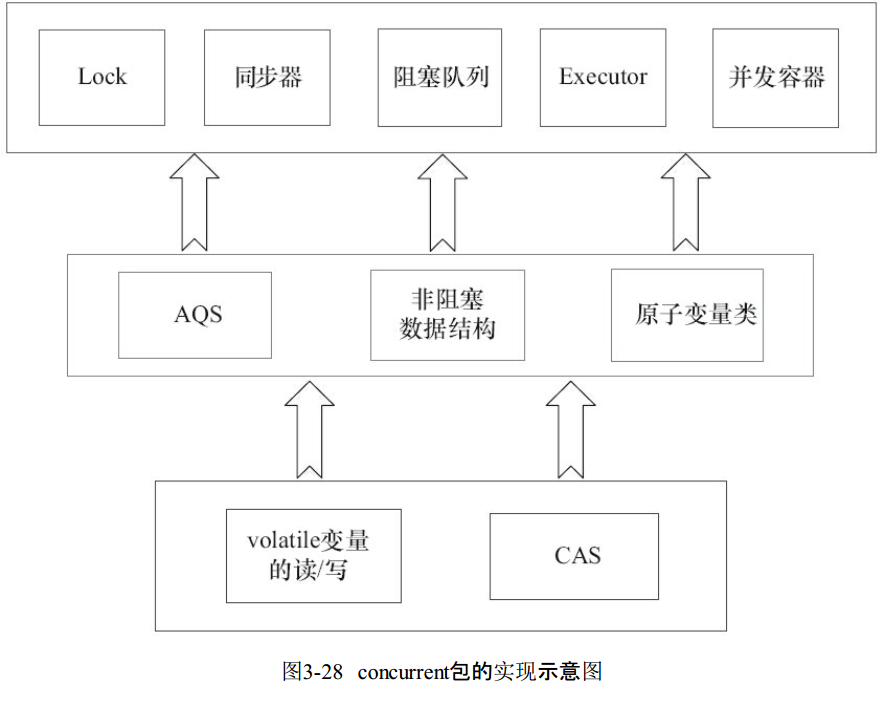
volatile int waitStatus; // 同步状态变量

volatile Node prev;

volatile Node next;

volatile Thread thread;

Exclusive（独占，只有一个线程能执行，如ReentrantLock，一个时刻仅一个线程持有同步状态）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。



队列同步器AbstractQueuedSynchronizer ，实现了一个FIFO的队列，其他上层类通过集成这个类可以快速实现，主要方法：

java.util.concurrent.locks.AbstractQueuedSynchronizer.getState()

java.util.concurrent.locks.AbstractQueuedSynchronizer.setState(int)

java.util.concurrent.locks.AbstractQueuedSynchronizer.compareAndSetState(int, int)

定义了以下的接口方法：





为了方便从开头、结尾取值，每一个队列都有一个head、tail变量。

使用这个框架可以很方便地自定义一个同步器，因为只需要实现state的获取与释放，其余的比如说等待队列已经在顶层自动实现。

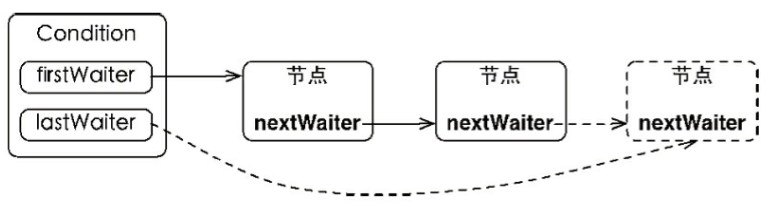
继承AQS 制定一个Mutex：



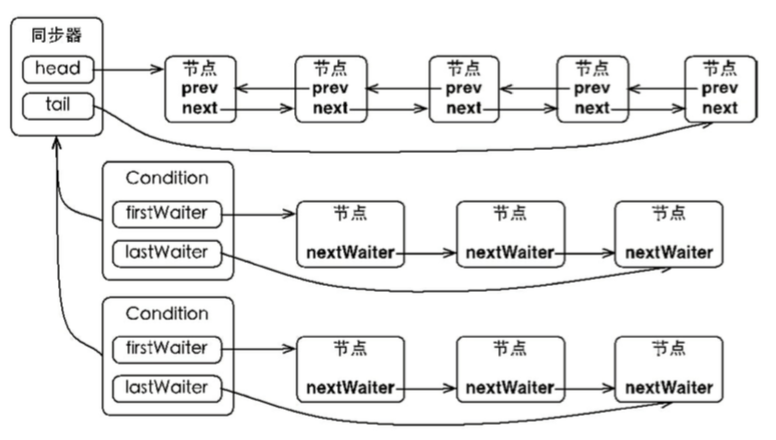
setExclusiveOwnerThread 设置占有状态的线程

继承了Lock接口，但是相关的Lock函数通过内部的AQS类实现。注意，释放锁的时候不需要CAS操作。

Condition 条件变量类，这个对象需要Lock对象创建（一个Lock对象可以关联多个Condition），之后通过调用函数如果发生需要阻塞的情况就直接使用这个关联的锁。



等待对列



同步队列与等待对列

同步队列存放着竞争同步资源的线程的引用（不是存放线程），而等待队列存放着待唤醒的线程的引用。当调用Condition的signal()后unlock() 就会将等待队列中的一个节点移动到 同步队列；调用await() 会将同步队列的队头 移到 Condition的等待队列。

Condition接口也提供了类似Object监视器的方法，对象由Lock对象创建，一个Lock对象可以创建多个Condition对象，实现有界队列的时候应该使用Condition（上限、下限）。

Lock lock = new ReentrantLock();

Condition condition = lock.newCondition();

public void conditionWait() throws InterruptedException {

lock.lock(); //加入到同步队列

try {

condition.await(); // 从同步队列移到等待队列，同步节点中下一个节点会获得锁（因为成为首节点，虽然没有执行unlock()），由此实现了共享锁

} finally {

lock.unlock();

}

}

public void conditionSignal() throws InterruptedException {

lock.lock();

try {

condition.signal();

} finally {

lock.unlock();

}

}

await 函数通过LockSupport.park（this）来实现

文件读写的过程中，写操作要求对资源的独占式访问，而读操作可以是共享式访问。共享与独占之间也要相互排斥。使用ReentrantLock可以实现一个简单的读写锁思想的锁，我想的是每一次读操作完成之后就调用条见变量的await，写操作完成之后调用signalAll唤醒所有的读线程。



实现三个线程交替打印ABC：

import java.util.concurrent.locks.Condition;import java.util.concurrent.locks.ReentrantLock;

public class ThreeThreadPrintABC {

private static ReentrantLock lock = new ReentrantLock();

private static Condition wait = lock.newCondition();

// 用来控制该打印的线程

private static int count = 0;

public static void main(String[] args) {

Thread printA = new Thread(new PrintA());

Thread printB = new Thread(new PrintB());

Thread printC = new Thread(new PrintC());

printA.start();

printB.start();

printC.start();

}

static class PrintA implements Runnable {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

lock.lock();

try {

while ((count % 3) != 0) {

wait.await();

}

System.out.println(Thread.currentThread().getName() + " A");

count++;

wait.signalAll();// 不能只是唤醒一个，因为可能唤醒的是Thread C，然后它又一次进入await，但是因为lock的存在即便是signalAll也是一个个逐步唤醒的

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

}

}

static class PrintB implements Runnable {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

lock.lock();

try {

while ((count % 3) != 1) {

wait.await();

}

System.out.println(Thread.currentThread().getName() + " B");

count++;

wait.signalAll();

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

}

}

static class PrintC implements Runnable {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

lock.lock();

try {

while ((count % 3) != 2) {

wait.await();

}

System.out.println(Thread.currentThread().getName() + " C");

count++;

wait.signalAll();

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

}

}

}

或者

import java.util.concurrent.locks.Condition;import java.util.concurrent.locks.ReentrantLock;

public class ThreeThreadPrintABC {

private static ReentrantLock lock = new ReentrantLock();

private static Condition waitA = lock.newCondition();

private static Condition waitB = lock.newCondition();

private static Condition waitC = lock.newCondition();

// 用来控制该打印的线程

private static int count = 0;

public static void main(String[] args) {

Thread printA = new Thread(new PrintA());

Thread printB = new Thread(new PrintB());

Thread printC = new Thread(new PrintC());

printA.start();

printB.start();

printC.start();

}

static class PrintA implements Runnable {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

lock.lock();

try {

while ((count % 3) != 0) {

waitA.await();

}

System.out.println(Thread.currentThread().getName() + " A");

count++;

waitB.signal();

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

}

}

static class PrintB implements Runnable {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

lock.lock();

try {

while ((count % 3) != 1) {

waitB.await();

}

System.out.println(Thread.currentThread().getName() + " B");

count++;

waitC.signal();

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

}

}

static class PrintC implements Runnable {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

lock.lock();

try {

while ((count % 3) != 2) {

waitC.await();

}

System.out.println(Thread.currentThread().getName() + " C");

count++;

waitA.signal();

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

lock.unlock();

}

}

}

}

}

await就可以看到线程对同步状态的释放与重新获取：

await会阻塞队列，也要阻塞住其他一样的线程，而lock只有一个

要保证一些变量的更新在锁的持有下



独占式: addWaiter(Node.EXCLUSIVE) 互斥量类

一个时刻只能一个线程成功获取同步状态

public final void acquire(int arg) {

if (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

selfInterrupt();

}//如果获取同步状态失败，使用addWaiter向同步队列队尾加入一个独占式的节点，最后调用acquireQueued(Node node,int arg)方法，使得该节点以“死循环”的方式获取同步状态。如果获取不到则阻塞节点中的线程，而被阻塞线程的唤醒主要依靠前驱节点的出队或阻塞线程被中断来实现

addWaiter函数 内部使用了一个死循环配合compareAndSetTail保证节点的正确添加 ，每一次循环都会重新获取tail节点。虽然感觉很奇怪，但是CAS操作本身就是一种乐观锁。

acquireQueued 函数 线程会进入自旋的状态，并检查属于线程的node是否是头节点的后继节点（即下一个被唤醒的节点），是的话，获取同步状态并退出死循环。

public final boolean release(int arg) {

if (tryRelease(arg)) {

Node h = head;

if (h != null && h.waitStatus != 0)

unparkSuccessor(h);

return true;

}

return false;

}//该方法执行时，tryRelease释放同步状态,会唤醒头节点的后继节点线程，unparkSuccessor(Node node)方法使用LockSupport（在后面的章节会专门介绍）来唤醒处于等待状态的线程。

共享式： addWaiter(Node.SHARED) 信号量类

允许同一时刻能否有多个线程同时获取到同步状态

调用同步器的acquireShared(int arg)方法可以共享式地获取同步状态

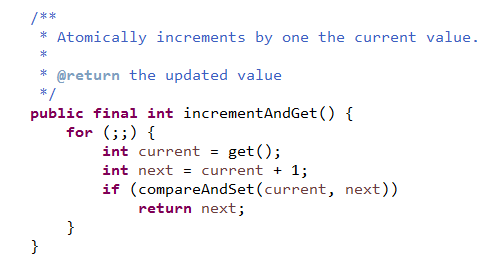
doAcquireNanos(int arg,long nanosTimeout)方法可以超时获取同步状态. 在Java 5之前，当一个线程获取不到锁而被阻塞在synchronized之外时，对该线程进行中断操作，此时该线程的中断标志位会被修改，但线程依旧会阻塞在synchronized上，等待着获取锁。在Java 5中，同步器提供了acquireInterruptibly(int arg)方法，这个方法在等待获取同步状态时，如果当前线程被中断，会立刻返回，并抛出InterruptedException。

调用releaseShared(int arg)方法可以释放同步状态

Unsafe只提供了3种CAS方法：compareAndSwapObject、compareAndSwapInt和compareAndSwapLong

cas的乐观锁是一种非阻塞的形式，Synchronized 悲观锁是一种阻塞的形式。

Compare and Swap，比较并操作，CPU指令，在大多数处理器架构，包括IA32、Space中采用的都是CAS指令，CAS的语义是“我认为V的值应该为A，如果是，那么将V的值更新为B，否则不修改并告诉V的值实际为多少”。java中使用一个函数：



Synchronized(xx) {} xx是锁对象

  \* 1. 如果用了类里面的一个属性object作为锁,由于这个属性每一个实例化对象都有一份,所以运行发现3个线程都没有同步

  \* 2. 如果用了类对象的class对象或者是类的静态属性作为锁, 或者其他的线程关联对象

一般见到是在方法面前修饰，修饰方法的时候相当于是锁对象。

Synchronized是非公平锁。从JVM规范中可以看到Synchonized在JVM里的实现原理，JVM基于进入和退出Monitor对象（锁定/释放这个对象）来实现方法同步和代码块同步，但两者的实现细节不一样。代码块同步是使用monitorenter和monitorexit指令实现的，而方法同步是使用另外一种方式实现的，细节在JVM规范里并没有详细说明。但是，方法的同步同样可以使用这两个指令来实现。monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处，JVM要保证每个monitorenter必须有对应的monitorexit与之配对。

**synchronized 采用的优化：**

**锁消除：**

对于一些看起来没有加锁的代码，其实隐式的加了很多锁。例如下面的字符串拼接代码就隐式加了锁：

public static String concatString(String s1, String s2, String s3) {

return s1 + s2 + s3;

}

String 是一个不可变的类，编译器会对 String 的拼接自动优化。在 JDK 1.5 之前，会转化为 StringBuffer 对象的连续 append() 操作：

public static String concatString(String s1, String s2, String s3) {

StringBuffer sb = new StringBuffer();

sb.append(s1);

sb.append(s2);

sb.append(s3);

return sb.toString();

}

在 JDK 1.5 之前，会转化为 StringBuffer 对象的连续 append() 操作，这时候编译器加上一些选项就会自动将这个锁消除掉，在 JDK 1.5 及以后的版本中，会转化为 StringBuilder 对象的连续 append() 操作

**锁粗化**：

如果一系列的连续操作都对同一个对象反复加锁和解锁，频繁的加锁操作就会导致性能损耗。所以如果被检测到这种行为，就会自动将锁的范围扩展，从而减少锁的获取。数据库高并发需要锁细化。比如不断对StringBuffer对象进行操控。

**自适应的自旋锁**

JDK1.6中引入了自适应的自旋锁。 **自适应意味着自旋的时间不再是固定的**， 而是由前一次在同一个锁上的自旋时间以及锁拥有者的状态来决定。如果在同一个锁对象上, **自旋等待刚好成功获得锁， 并且在持有锁的线程在运行中**， 那么虚拟机就会认为这次自旋也是很有可能获得锁， 进而它将允许**自旋等待相对更长的时间**。

**轻量级锁**：

本质上是大量使用CAS，因为对于绝大部分的锁，在整个同步周期内都是不存在竞争的，因此也就不需要都使用互斥量进行同步，可以先采用 CAS 操作进行同步，如果 CAS 失败了再改用互斥量进行同步，升级为重量锁。

**偏向锁**：

偏向锁的思想是偏向于让第一个获取锁对象的线程，这个线程在之后获取该锁就不再需要进行同步操作，甚至连 CAS 操作也不再需要。

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径。因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。上面说过，轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

ReentrantLock是一个可重入的互斥锁，又被称为“独占锁”，默认是非公平锁（效率比公平锁高）。

顾名思义，ReentrantLock锁在同一个时间点只能被一个线程锁持有；而可重入的意思是锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次成功获取。不过也是锁对象。判断是否阻塞之前需要判断线程号，同时记录本线程重入的次数。

ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。性能比较好，但是耗费CPU。ReenTrantLock 是java中可以自己控制的一种锁，对链表内元素加锁的时候可以使用。

ReentrantLock分为公平锁和非公平锁的内存语义：

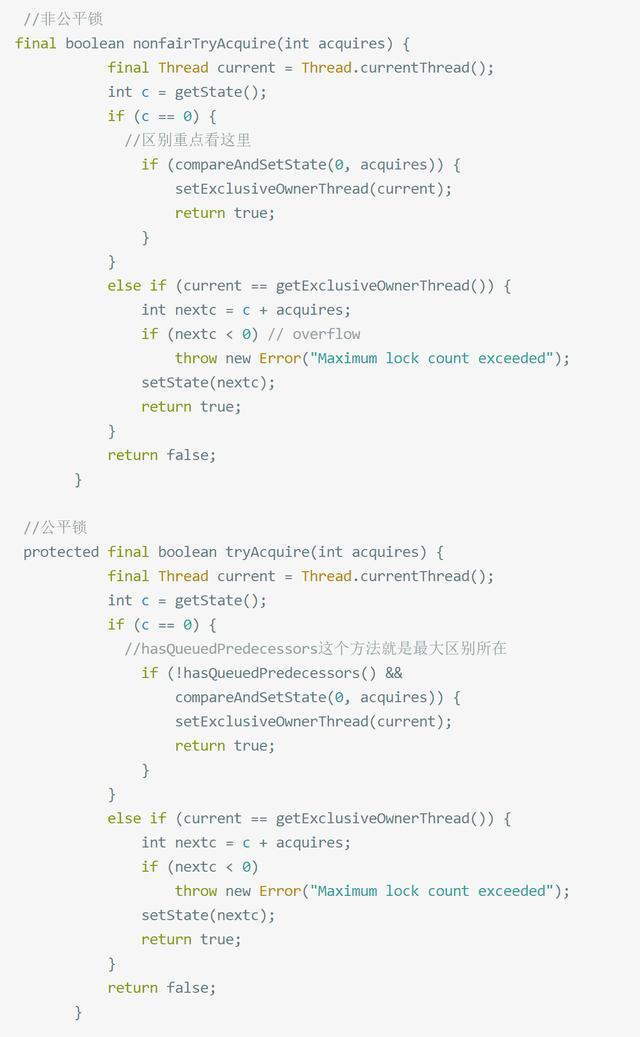
公平锁和非公平锁释放时，最后都要写一个volatile变量state。

下面的话我不是很懂，除了括号内的补充，都来自书上原话：

公平锁获取时，首先会去读volatile变量。

非公平锁获取时，首先会用CAS更新volatile变量，这个操作同时具有volatile读和volatile写的内存语义。（即如果能够获取到的话就就不会进入等待队列）

反正我看源码都是先读volatie变量，两者只是差一个是不是队头的判断。



!hasQueuedPredecessors() 判断是否是队头

**ReenTrantLock相比较Synchronized：**

**便利性**：很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放

**锁的细粒度和灵活度**：很明显ReenTrantLock优于Synchronized。ReenTrantLock需要指定使用的是公平锁还是非公平锁，手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。超时等待获取锁ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。tryLock() 非阻塞地尝试获取锁，如果获取返回true，失败返回false。可以引申出条件变量的使用。

**共享锁**：ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们

**ReentrantReadWriteLock：**

　ReentrantReadWriteLock lock = new ReentrantReadWriteLock()

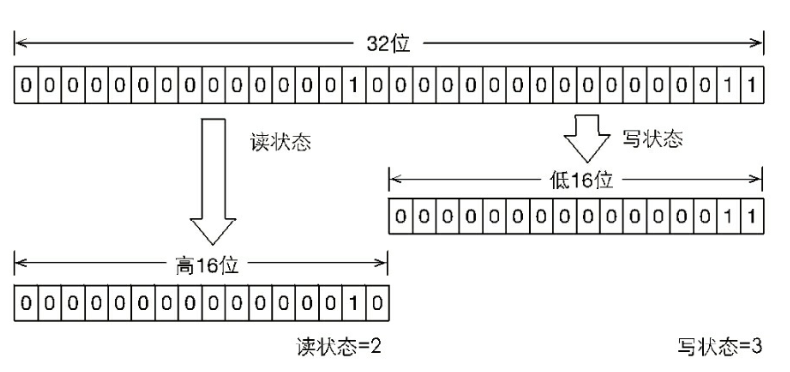
　　ReadLock r = lock.readLock();

　　WriteLock w = lock.writeLock();

两者都有lock,unlock方法。写写，写读互斥；读读不互斥（相当于多个读程序一起占用一个写锁，具体实现与可重入锁的实现差不多，然后写锁之间互斥）。可以实现并发读的高效线程安全代码

可以实现锁降级，通过写锁转换为一个读锁

读写锁需要在一个变量上维护多个读线程和一个写线程的状态： 大致如下



每个线程各自获取读锁的次数只能选择保存在ThreadLocal中，例如getReadHoldCount()方法，作用是返回当前线程获取读锁的次数。

java.util.concurrent.atomic

**基本类：**AtomicInteger、AtomicLong、AtomicBoolean、....；

**数组类型：**AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray、...

通过"volatile"和"Unsafe提供的CAS函数实现"原子操作：  
(01) value是volatile类型。这保证了：当某线程修改value的值时，其他线程看到的value值都是最新的value值，即修改之后的volatile的值。  
(02) 通过CAS设置value。这保证了：当某线程池通过CAS函数(如compareAndSet函数)设置value时，它的操作是原子的，即线程在操作value时不会被中断。

**AtomicInteger**:

public class AtomicExample {

private AtomicInteger cnt = new AtomicInteger();

public void add() {

cnt.incrementAndGet();

}

public int get() {

return cnt.get();

}

}

public static void main(String[] args) throws InterruptedException {

final int threadSize = 1000;

AtomicExample example = new AtomicExample(); // 只修改这条语句

final CountDownLatch countDownLatch = new CountDownLatch(threadSize);

ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < threadSize; i++) {

executorService.execute(() -> {

example.add();

countDownLatch.countDown();

});

}

countDownLatch.await(); // await会阻塞主线程,直到N变成零

executorService.shutdown();

System.out.println(example.get());

}

内部变量：private volatile int value;

自增等同时需要读写操作：CAS+自旋

public final int incrementAndGet() {

for (;;) {

int current = get();

int next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

/\*\*

\* 最终设为指定值，但其它线程不能马上看到变化，会延时一会

\*/

public final void lazySet(int newValue) {

unsafe.putOrderedInt(this, valueOffset, newValue);

}

从理论上讲lazySet能比一个标准的volatile变量的写性能更好，源码不能说明，但是jvm对这个unsafe库进行了加强。

**AtomicReference**：

一种原子类，所有方法保证原子操作。使用这个能够保证一个对象的原子性操作，多个变量能够存放到一个对象中，即对多个变量的CAS操控。

虽然提供了compareAndSet原子操作，但是可能会导致ABA问题（对象的值从A变成B，之后又从B变成A，虽然这对值的使用没有多大影响，但是这可能意味着环境的变化）<https://blog.csdn.net/youling_lh/article/details/9403115> 改用AtomicStampedReference，每一次变化都会更新一个stamp码值

**AtomicStampedReference:** 维护对象的同时，维护了一个版本号。每一次对对象进行操作的时候，还会对版本号进行修改。

这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

**AtomicMarkableReference**：

AtomicStampedReference是使用int stamp作为计数器使用，AtomicMarkableReference使用的是boolean mark。 AtomicStampedReference可能关心的是动过几次，AtomicMarkableReference关心的是有没有被人动过，方法都比较简单。

**AtomicReferenceFieldUpdater**：

一个基于反射的工具类，它能对指定类的指定的volatile引用字段进行原子更新。AtomicReferenceFieldUpdater updater=AtomicReferenceFieldUpdater.newUpdater(Dog.**class**,String.**class**,"name")

Dog dog1=**new** Dog();

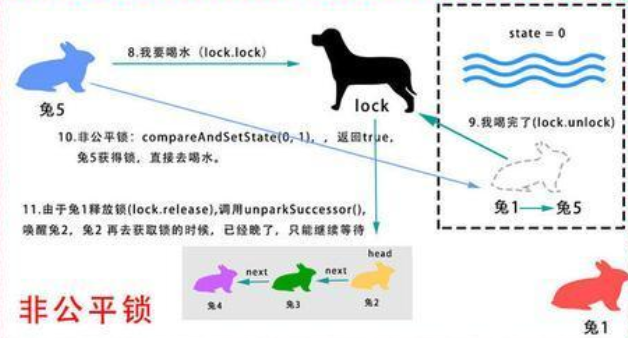
updater.compareAndSet(dog1,dog1.name,"test") ;

·AtomicIntegerFieldUpdater：原子更新整型的字段的更新器。

·AtomicLongFieldUpdater：原子更新长整型字段的更新器。

上面三个对字段的更新都是抽象类，所以创建的时候使用静态方法newUpdater()

java AQS中的非公平锁并不是随机唤醒一个等待线程，也有一个等待队列的实现，当有线程占用锁的时候，申请锁的线程会进入等待队列，当线程释放锁且等待队列队头获取锁之前，如果这时候恰好有线程申请就会直接得到资源，然后队头获取锁失败。减少了线程挂起的几率，所以效率比公平锁高。



非公平锁的弊端：可能产生饥饿现象

锁降级指的是写锁降级为读锁，把持住(当前拥有的)写锁，再获取到读锁，随后释放(先前有用的)写锁的过程。即换了一个占有的锁。避免在直接释放写的时候，被另外的线程获取到了写锁，因为读写排斥，导致之后读锁获取失败。

1. ThreadLocal

将变量弄成一个线程的本地变量，比如方法中的临时变量，因为每一个线程都有自己的栈，所以线程之间不会因为这个变量出问题。或者使用java.lang.ThreadLocal 类来实现线程本地存储功能。每个 Thread 都有一个 ThreadLocal.ThreadLocalMap 对象(因为要支持存储多种对象，所以值的类型是Object，不能存储基本类型)，ThreadLocal提供了操作这个Map的方法。

138.java中private变量之后，如果要访问变量应该创建getter、setter方法，这个看似与public一样，但是很多框架会需要这样的方法来配合。

他们这么做的原因号称是为了防止客户端的随意修改，但是两函数都提供了之后效果上是一样的。对于包级私有的类或者私有的嵌套类，那么直接暴露成员不会有特别大的影响。使用getter、setter方法可以将内部成员变量进一步封装，比如通过这两个方法进行格式转换、权限控制等额外的功能。

1. 序列化记录的不仅仅是数据，还有一些相关的信息。

static、transient 不能参与默认提供的序列化，序列化不受访问修饰符的限制。

序列化与反序列化可以实现对象的复制，所以单例设计模式不能实现序列化接口。

140. static除了可以修饰变量、方法、类，还可以用来形成静态代码块以优化程序性能，修饰import（不推荐）。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。

// 如何让一个对象打印static变量：

public class Test {

static int value = 33;

public static void main(String[] args) throws Exception{

new Test().printValue();

}

private void printValue(){

int value = 3;

System.out.println(this.value);

}

}

**静态导包：**在使用静态变量和方法时不用再指明 ClassName，从而简化代码，但可读性大大降低。import static com.xxx.ClassName.\*

接口与抽象类在“是否所有方法都是抽象方法”、“要求的修饰符”等情况不同

142.java中的null，一个不属于任何类型却又可以被转换为除了基本类型之外任何一个类型的对象（转换不会报错，但是它还是一个null），是所有引用对象的默认值。

public class Null{  
public static void nihao(){  
　　System.out.println("你好陌生人");  
  }  
public static void main(String[] args){  
　　((Null)null).nihao();  
  }  
}因为上面使用的是static方法，所以可以执行。



public class Test {

public static void main(String[] args) {

System.out.println((Integer)null);

}

}//输出null

(Integer)null+Integer.valueOf(0) 编译的时候不会报错，运行的时候出现异常Exception in thread "main" java.lang.NullPointerException

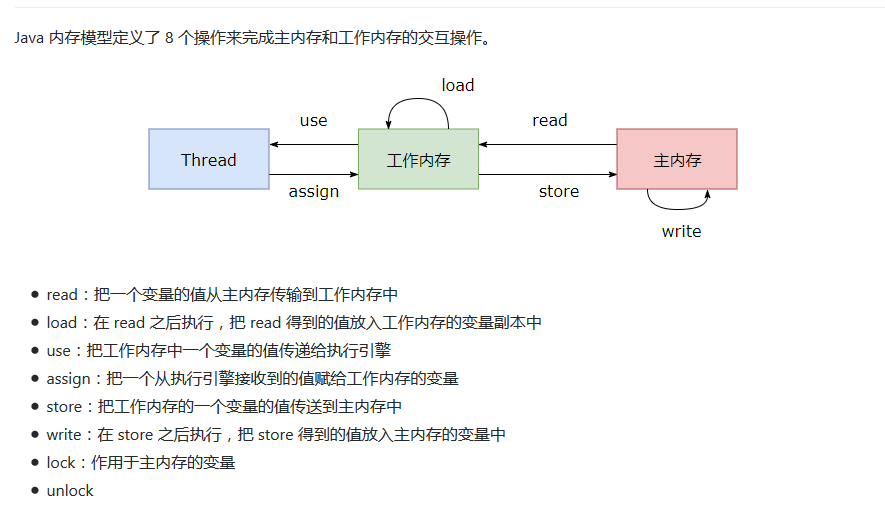
(StringBuilder)null.append("123") 因为append不是静态方法所以不能被执行

143.jvm

jvm单进程多线程

jvm支持的语言不少（包括动态类型的语言），只要最终变成可以识别的字节码

jvm 并不是唯一使用了GC算法的东东



上面8种操作是jvm中内存模型的原子操作，但是这几种原子操作安装一定顺序执行的时候，就会出现问题：

Java内存区域（注意不是Java内存模型JMM）的划分？

线程独有：程序计数器、虚拟机栈、本地方法栈

线程共享：Java堆、方法区（其中永久代分离为元空间，元空间已经脱离了JMM）

本地方法栈和虚拟机栈类似，不过虚拟机栈针对Java方法，而本地方法栈针对Native方法。

方法区存放被虚拟机加载的**类信息、常量（final）、静态变量（static）、即时编译期编译后的代码**。

栈：存放基本类型的变量数据和对象的引用（都是局部变量），但对象本身不存放在栈中，而是存放在堆（new 出来的对象）或者常量池中（字符串常量对象存放在常量池中） 。

即便是ThreadLocalMap都是存储在堆，可见堆内的变量存储分线程的。

堆内内存 = 新生代+老年代+持久代（持久代已经被彻底删除了，取代它的是堆外内存——元空间）其中存储的是栈内对象的引用

堆外内存就是把内存对象分配在Java虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机），这样做的结果就是能够在一定程度上减少垃圾回收对应用程序造成的影响。比如java.nio.DirectByteBuffer

DirectByteBuffer(int cap) {

super(-1, 0, cap, cap);

//内存是否按页分配对齐

boolean pa = VM.isDirectMemoryPageAligned();

//获取每页内存大小

int ps = Bits.pageSize();

//分配内存的大小，如果是按页对齐方式，需要再加一页内存的容量

long size = Math.max(1L, (long)cap + (pa ? ps : 0));

//用Bits类保存总分配内存(按页分配)的大小和实际内存的大小

Bits.reserveMemory(size, cap);

long base = 0;

try {

//在堆外内存的基地址，指定内存大小

base = Unsafe.allocateMemory(size);

} catch (OutOfMemoryError x) {

Bits.unreserveMemory(size, cap);

throw x;

}

Unsafe.setMemory(base, size, (byte) 0);

//计算堆外内存的基地址

if (pa && (base % ps != 0)) {

// Round up to page boundary

address = base + ps - (base & (ps - 1));

} else {

address = base;

}

cleaner = Cleaner.create(this, new Deallocator(base, size, cap));

att = null;

}

使用堆外内存的好处：

1. 空间更大（64bits没有了内存大小限制）
2. 减少GC压力（需自己管理内存）
3. 有一些操作需先将堆内内存转移到堆外

tomcat使用的是堆外缓存

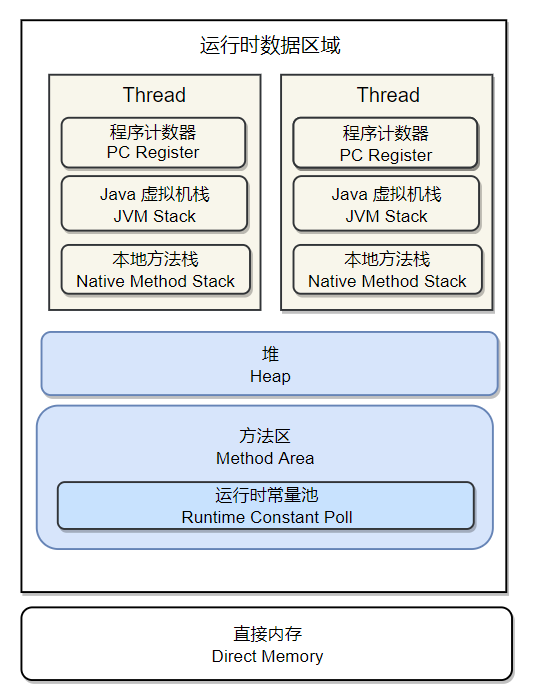
静态成员变量（类变量）----->方法区的静态部分

静态方法                   ----->方法区的静态部分

静态代码块               ----->方法区的静态部分

非静态方法（包括构造函数）----->方法区的非静态部分

构造代码块 (类内直接定义且没有static修饰的代码块)              ----->方法区的静态部分



上图中堆、方法区的内存不需要连续的内存（物理内存不连续，但是虚拟内存连续），并且可以动态扩展，垃圾回收算法主要针对这部分区域。

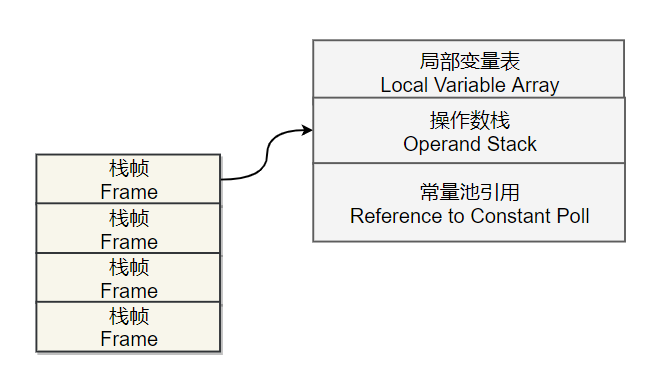
线程资源在线程结束之后自己会被回收，所以不需要垃圾回收机制处理。

**本地方法栈：**与 Java 虚拟机栈类似，它们之间的区别只不过是本地方法栈为本地方法服务。本地方法一般是用其它语言（C、C++ 或汇编语言等）编写的，并且被编译为基于本机硬件和操作系统的程序，对待这些方法需要特别处理。

**常量池**主要用于存放两大类常量：字面量(Literal)和符号引用量(Symbolic References)，字面量相当于Java语言层面常量的概念，如文本字符串，声明为final的常量值等，符号引用则属于编译原理方面的概念，包括了如下三种类型的常量：

* 类和接口的全限定名
* 字段名称和描述符
* 方法名称和描述符

java方法在执行的时候，每一个方法对应堆栈里面的一个**栈帧（Frame）**，栈帧指向局部变量表（基本类型、对象引用）、操作数栈、常量池引用。方法执行前后就是栈帧的入栈与出栈

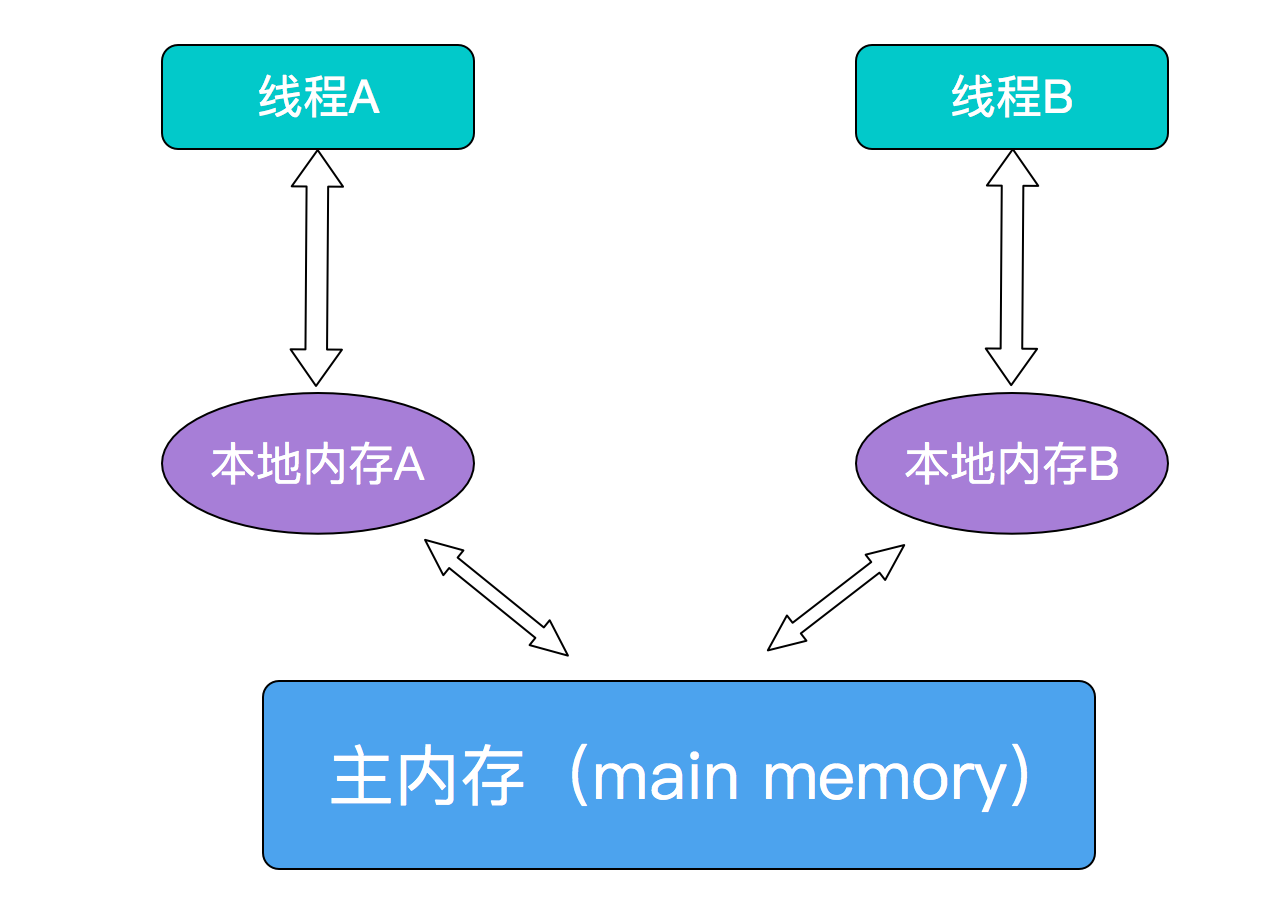


不同于程序计数器，Java虚拟机没有寄存器，程序计数器也无法被程序指令直接访问。Java虚拟机的指令是从操作数栈中而不是从寄存器中取得操作数的，因此它的运行方式是基于栈的而不是基于寄存器的。虽然指令也可以从其他地方取得操作数，比如从字节码流中跟随在操作码（代表指令的字节）之后的字节中或从常量池中，但是主要还是从操作数栈中获得操作数。虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈，记得instanceof就会将结果协会栈中。

java虚拟机有自己的内存模型（Java Memory Model，JMM），JMM可以屏蔽掉各种硬件和操作系统的内存访问差异，以实现让java程序在各种平台下都能达到一致的内存访问效果。

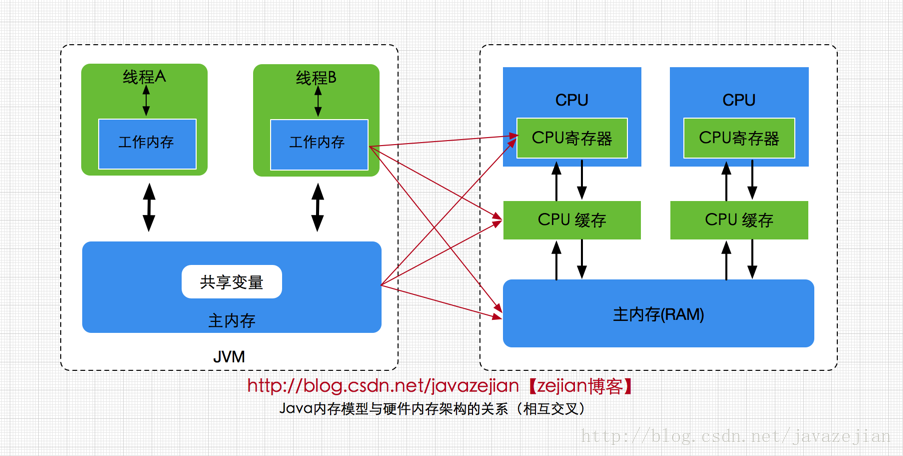
32 位 JMM不保证对64位的long型和double型变量的写操作具有原子性，因为总线工作机制。https://blog.csdn.net/qq\_25343557/article/details/78010832 通过一个程序，大量循环+if判断+多线程。

当写一个volatile变量时，JMM会把该线程对应的本地内存中的变量强制刷新到主内存中去（保证变量的可见性、但是不能保证操作的 原子性）（通过线程之间的通信），同时会禁止jvm进行指令重排序，同时导致其他线程本地内存中的变量缓存（包括其他CPU核心中的缓存）立即无效（对应汇编Lock指令）。

****

上图说明java采用是共享内存模型，主内存中是所有的变量，本地内存中是变量的缓存

volatile并不是一种线程同步的方式，因为不能提供阻塞操作，只是提供了一种数据同步方式



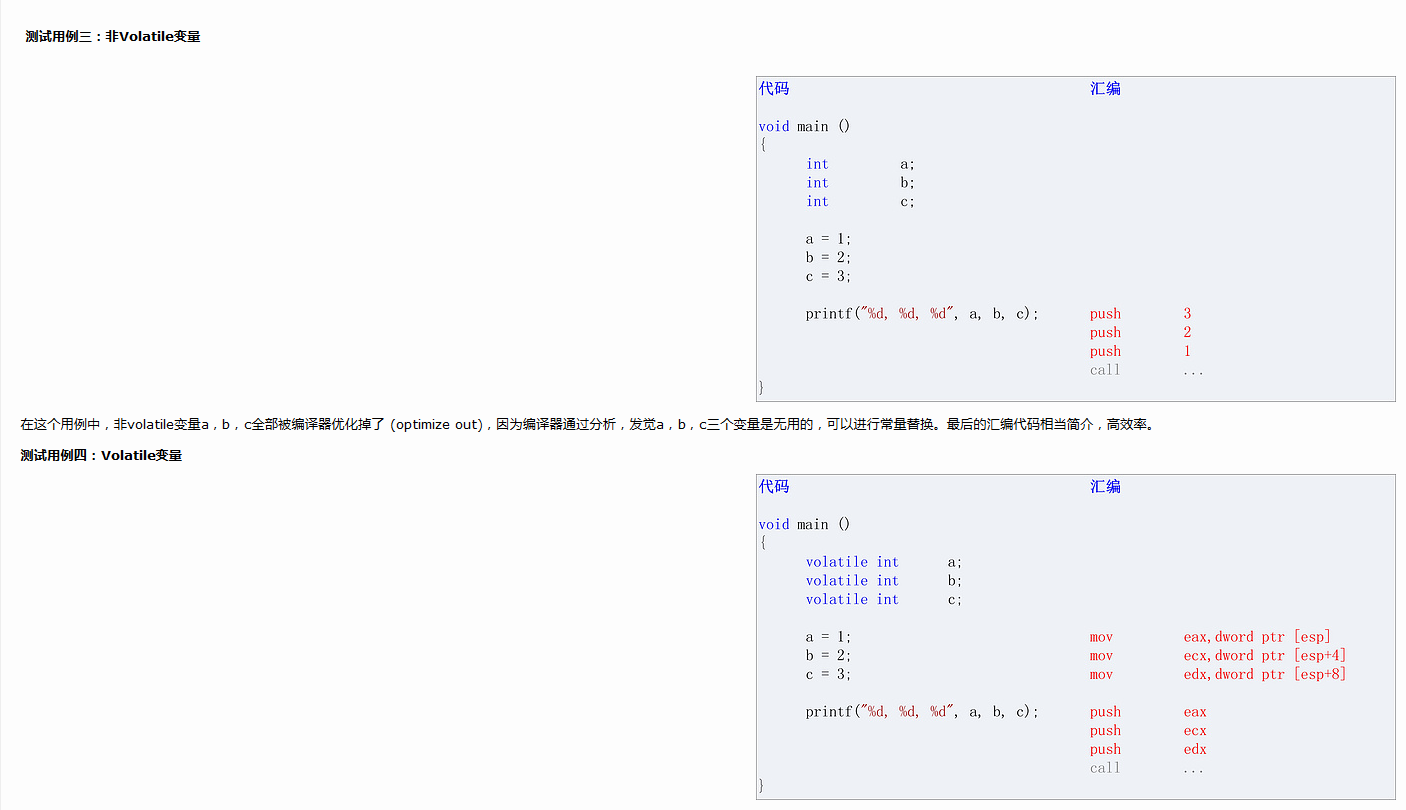
上图其实说明了JVM中没有寄存器，都是被映射为内存。



因为num++不是个原子操作，实际上分为3步骤，所以一个线程因为写入之后刷新其余线程本地内存的时候，其余线程可能已经跑完一次循环，这也就意味着一次白跑了。

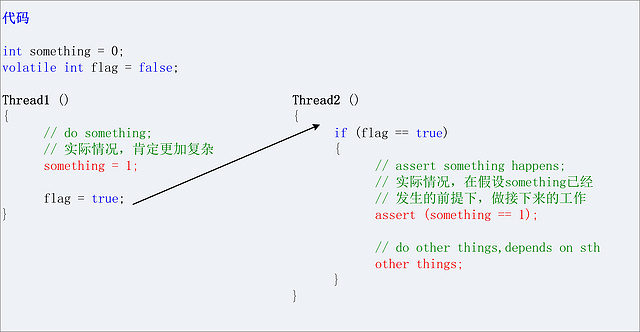
对照C++中的volatile：

1. 表示“易变性”，所以在汇编层面反映出来，就是两条语句，下一条语句不会直接使用上一条语句对应的volatile变量的寄存器内容，而是重新从内存中读取。
2. 禁止相关重排序。



通常会在”全局”变量配合多线程的时候使用，但是和Java一样容易产生问题：

经过编译器的优化之后，something内容可能在flag=true之后执行，对于下图中的例子要想保证执行顺序可以将flag、something都设置为volatile，于是编译器就不会优化。



jvm开启的时候会创建5个线程，之后运行的程序有几个线程就会里面也会多上几个线程。Attach Listener ：该线程是负责接收到外部的命令，执行该命令，并且把结果返回给发送者。通常我们会用一些命令去要求jvm给我们一些反馈信息，如：java -version、jmap、jstack等等。如果该线程在jvm启动的时候没有初始化，那么，则会在用户第一次执行jvm命令时，得到启动。

signal dispather： 前面我们提到第一个Attach Listener线程的职责是接收外部jvm命令，当命令接收成功后，会交给signal dispather线程去进行分发到各个不同的模块处理命令，并且返回处理结果。signal dispather线程也是在第一次接收外部jvm命令时，进行初始化工作。

Finalizer： JVM在垃圾收集时会将失去引用的对象包装成Finalizer对象（Reference的实现），并放入ReferenceQueue，由Finalizer线程来处理；最后将该Finalizer对象的引用置为null，由垃圾收集器来回收。

Reference Handler ：它主要用于处理引用对象本身（软引用、弱引用、虚引用）的垃圾回收问题。

main：主线程，用于执行我们编写的java程序的main方法。

-Xms堆初始值

-Xmx堆最大值

-Xss 每一个线程栈大小

-Xmn 年轻代大小

-XX:PermSize ， -XX:MaxPermSize 永久区分配大小调整

与下面这种查看方式相比，我肯定喜欢界面化的：

-XX:+printGC参数来使能JVM的GC日志打印

[GC (Allocation Failure) 4416K->716K(15872K), 0.0018384 secs]

-XX:+PrintGCDetails

[GC (Allocation Failure) [DefNew: 4480K->0K(4992K), 0.0001689 secs] 5209K->729K(15936K), 0.0001916 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

输出更加详细的打印信息，user是指用户态的cpu时间..

并且会在退出的时候输出jvm堆的使用情况

-XX:+PrintClassHistogram：监控类的使用情况

遇到Ctrl-Break后打印类实例的信息，与jmap -histo功能相同

jps 可以查看jvm关联的进程以及pid

jstat 可以显示本地或远程虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据，在没有GUI图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。不同参数查看到的信息不同。

jinfo 实时地查看和调整虚拟机的各项参数。

jmap命令：jmap的用途是为了展示java进程的内存映射信息，还可以用于生成堆转储快照(一般称为heapdump或dump文件)，查询finalize执行队列。如果不使用jmap命令，要想获取Java堆转储快照还有一些比较“暴力”的手段：譬如在第2章中用过的-XX:+HeapDumpOnOutOfMemoryError参数，可以让虚拟机在OOM异常出现之后自动生成dump文件，通过-XX:+HeapDumpOnCtrlBreak参数则可以使用【Ctrl】+【Break】键让虚拟机生成dump文件，又或者在Linux系统下通过Kill -3命令发送进程退出信号“恐吓”一下虚拟机，也能拿到dump文件。

jhat 分析jmap导出的堆文件，并通过html的方式展示

jstack 打印栈，并可以得知那些线程处于阻塞或者等待状态，从而找出死锁等问题。当线程占用CPU的时候，因为生产环境不能debug，打印栈找到问题代码。（dump出来的进程id是十六进制）

打印线程信息

javap是 Java class文件分解器,可以反编译(即对javac编译的文件进行反编译),也可以查看java编译器生成的字节码。用于分解class文件。

内存分配策略：

1. 对象在新生代 Eden 区分配，当 Eden 区空间不够时，发起 Minor GC。
2. 需要连续空间的大对象（很长的字符串以及数组）存放到老年代，因为大对象会占用大量内存空间，所以不能再放在Eden，同时为了方便老年代的整理，应该是连续的大对象。
3. 长期存活的对象进入老年代，从而腾出Eden内存。

为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。有点奇怪的是这种做法没有考虑LRU

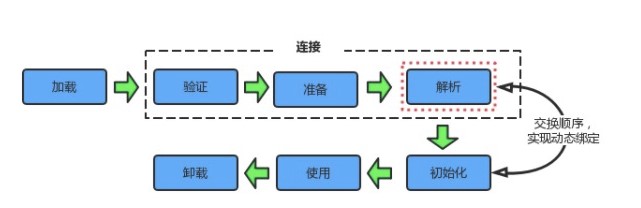
虚拟机并不是永远地要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄。

1. 因为3的存在，所以每一次Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间来判断这次Minor GC是否是安全的，如果不成立的话虚拟机会查看 HandlePromotionFailure 设置的值（是否允许担保分配内存失败，即整个年老代空间不足，而整个新生代中Eden和Survivor对象都存活的极端情况），如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 设置不允许担保即担保失败，那么就要进行一次 Full GC。

从上面可以知道要想避免出现老年代内存不足的情况，应该通过-Xmn调大Eden（将对象控制在新生代，这比扩大老年区内存好），同时尽量避免创建连续的大对象，-XX:MaxTenuringThreshold 调大对象进入老年代的年龄。

**类的加载：**

类的加载是动态的，这个动态链接库出现的原因一样。



动态绑定指多态

**加载**

* 通过一个类的全限定名获取定义该类的二进制字节流
* 将字节流表示的静态存储结构转化为方法区的运行时数据结构
* 在内存中生成这个类的Class对象，作为方法区这个类的各种数据的访问入口（存储在方法区）

其中二进制字节流可以从以下方式中获取：

* 从 ZIP 包读取，成为 JAR、EAR、WAR 格式的基础。
* 从网络中获取，最典型的应用是 Applet。
* 运行时计算生成，例如动态代理技术，在 java.lang.reflect.Proxy 使用 ProxyGenerator.generateProxyClass 的代理类的二进制字节流。
* 由其他文件生成，例如由 JSP 文件生成对应的 Class 类。

**验证**

* 文件格式验证：比如检查是否以魔数0xCAFEBABE开头
* 元数据验证：对类的元数据信息进行语义校验，保证不存在不符合Java语言规范的元数据信息。比如检查该类是否继承了被final修饰的类。元数据就是存储在方法区里面的那些数据。
* 字节码验证，通过数据流和控制流的分析，验证程序语义是合法的、符合逻辑的。
* 安全审计（一些危险代码不允许）

**准备**。 为类变量（static）分配内存并设置默认值。比如static int a = 123在准备阶段的默认值是0，但是如果有final修饰，在准备阶段就会被赋值为123了。使用方法区内存，估计还有final变量也会被分配内存并设置默认值。

**解析**。

解析阶段，将常量池中的符号引用（xx.xxx.xx）改为直接的地址引用，因为准备阶段已经创建好内存了。其中解析过程在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 的动态绑定（多态性）。

**初始化**。

按照程序员的计划初始化类变量。如static int a = 123，在准备阶段a的值被设置为默认的0，而到了初始化阶段其值被设置为123。

初始化阶段即虚拟机执行类构造器 <clinit>() 方法（编译器收集的静态类变量创建与静态语句块构成，所以不一定会有）的过程。虚拟机会保证一个类的 <clinit>() 方法在多线程环境下被正确的加锁和同步，如果多个线程同时初始化一个类，只会有一个线程执行这个类的 <clinit>() 方法，其它线程都会阻塞等待，直到活动线程执行 <clinit>() 方法完毕。如果在一个类的 <clinit>() 方法中有耗时的操作，就可能造成多个线程阻塞，在实际过程中此种阻塞很隐蔽。

public class Test {

static {

i = 0; // 静态语句块只能访问到定义在它之前的类变量

System.out.print(i); // 这句编译器会提示“非法向前引用”

}

static int i = 1;

}

类加载的时机： 标准只规定了必须进行类加载的时机

主动引用：

* 遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则必须先触发其初始化。最常见的生成这 4 条指令的场景是：使用 new 关键字实例化对象的时候；读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候；以及调用一个类的静态方法的时候。
* 使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行初始化，则需要先触发其初始化。
* 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
* 当虚拟机启动时，用户需要指定一个要执行的主类（包含 main() 方法的那个类），虚拟机会先初始化这个主类；
* 当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果为 REF\_getStatic, REF\_putStatic, REF\_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化；

被动引用： 不会触发初始化，因为还没有这个必要

* 通过子类引用父类的静态字段，不会导致子类初始化等。单例的构造函数方式依靠了这个原理。
* 通过数组定义来引用类，不会触发此类的初始化。该过程会对数组类进行初始化，数组类是一个由虚拟机自动生成的、直接继承自 Object 的子类，其中包含了数组的属性和方法。

在首次发生下列任意一种情况时，一个类或接口类型T将被立即**初始化**。

1）T是一个类，而且一个T类型的实例被创建。

2）T是一个类，且T中声明的一个静态方法被调用。

3）T中声明的一个静态字段被赋值。

4）T中声明的一个静态字段被外部使用，而且这个字段不是一个常量字段。

想到了static final 变量，在类的构造函数中设置自己的值。

如果一个变量要大量被循环使用的话，可以为多线程维护一个变量池。

连接池的定义:通过构造函数初始化连接的最大上限，通过一个双向队列来维护连接，调用方需要先调用fetchConnection(long)方法来指定在多少毫秒内超时获取连接，当连接使用完成后，需要调用releaseConnection(Connection)方法将连接放回线程池。线程取过来的连接会用ThreadLocal保存。

因为重排序的关系，构造函数（类加载）实际上分为3步（分配地址、对地址进行初始化、返回地址给目标），但如果将“对地址的初始化”与“返回地址给目标”重排序就会导致：取到instance不为null时，instance引用的对象有可能还没有完成初始化。多线程的时候需要初始化不能被其他线程看到就可以解决。类的初始化的时候是加锁的。

**类加载器**：

* 启动类加载器（Bootstrap ClassLoader），C++实现，此类加载器负责将存放在 <JRE\_HOME>\lib 目录中的，或者被 -Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给启动类加载器，直接使用 null 代替即可。
* 扩展类加载器（Extension ClassLoader）这个类加载器是由 ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将 <JAVA\_HOME>/lib/ext 或者被 java.ext.dir 系统变量所指定路径中的所有类库加载到内存中，开发者可以直接使用扩展类加载器。
* 应用程序类加载器（Application ClassLoader）这个类加载器是由 AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，因此一般称为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

因为每一个类加载器都有自己的类命名空间，不同空间中两个类时不可见的（会导致类的 Class 对象的 equals() 方法、isAssignableFrom() 方法、isInstance() 方法的返回结果为 false，也包括使用 instanceof 关键字做对象所属关系判定结果为 false，同一种类会被认为是不同的类，说明类的类型判断信息里面包含类加载器），但是只要得到类对象的引用《=》class.inInstance(obj)，还是可以使用另外一个空间中的类：

import java.net.\*;

import java.lang.reflect.\*;

public class LoaderSample2 {

    public static void main(String[] args) {

        try {

            String path = System.getProperty("user.dir");

            URL[] us = {new URL("file://" + path + "/sub/")};

            ClassLoader loader = new URLClassLoader(us); //传路径作为参数

            Class c = loader.loadClass("LoaderSample3");

            Object o = c.newInstance();

            Field f = c.getField("age");

            int age = f.getInt(o);

            System.out.println("age is " + age);

        } catch (Exception e) {

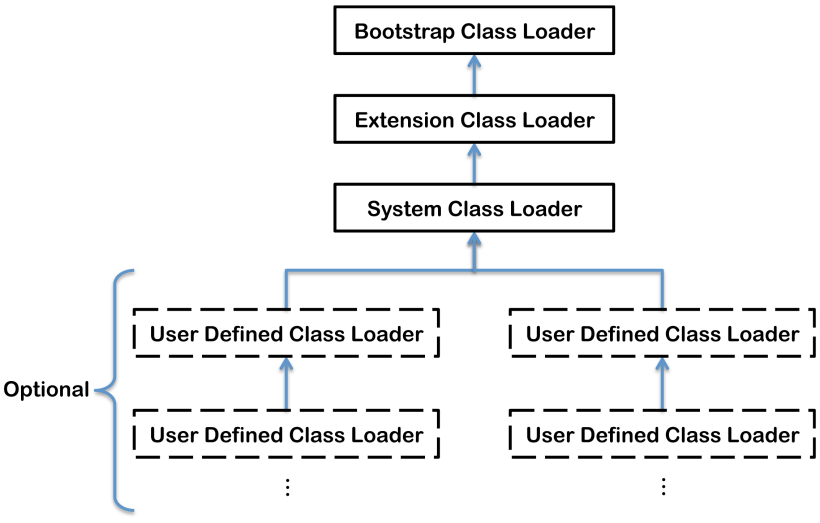
            e.printStackTrace();

        }

    }

}

**双亲委派模型**（Parents Delegation Model）：该安全模型要求除了顶层的启动类加载器外，其余的类加载器都应有自己的父类加载器，一个类加载器首先将类加载请求传送到父类加载器，只有当父类加载器无法完成类加载请求时抛出ClassNotFoundException才尝试自己加载：



使用了该模型之后，如果自己编写一个类在ClassPath中，但是因为类加载器的选择，系统的默认Object的优先级高于自己实现的Object，还是会加载系统实现。

**如何打破双亲委派模型？**使用OSGi可以打破。OSGI(Open Services Gateway Initiative)，或者通俗点说JAVA动态模块系统。可以实现代码热替换、模块热部署。在OSGi环境下，类加载器不再是双亲委派模型中的树状结构，而是进一步发展为更加复杂的网状结构。

**自定义类加载器实现**

FileSystemClassLoader 是自定义类加载器，继承自 java.lang.ClassLoader，用于加载文件系统上的类。它首先根据类的全名在文件系统上查找类的字节代码文件（.class 文件），然后读取该文件内容，最后通过 defineClass() 方法来把这些字节代码转换成 java.lang.Class 类的实例。

java.lang.ClassLoader 的 loadClass() 实现了双亲委派模型的逻辑，因此自定义类加载器一般不去重写它，但是需要重写 findClass() 方法。

public class FileSystemClassLoader extends ClassLoader {

private String rootDir;

public FileSystemClassLoader(String rootDir) {

this.rootDir = rootDir;

}

protected Class<?> findClass(String name) throws ClassNotFoundException {

byte[] classData = getClassData(name);

if (classData == null) {

throw new ClassNotFoundException();

} else {

return defineClass(name, classData, 0, classData.length);

}

}

private byte[] getClassData(String className) {

String path = classNameToPath(className);

try {

InputStream ins = new FileInputStream(path);

ByteArrayOutputStream baos = new ByteArrayOutputStream();

int bufferSize = 4096;

byte[] buffer = new byte[bufferSize];

int bytesNumRead;

while ((bytesNumRead = ins.read(buffer)) != -1) {

baos.write(buffer, 0, bytesNumRead);

}

return baos.toByteArray();

} catch (IOException e) {

e.printStackTrace();

}

return null;

}

private String classNameToPath(String className) {

return rootDir + File.separatorChar

+ className.replace('.', File.separatorChar) + ".class";

}

}

Java10中通过完全并行GC来改善G1最坏情况。

你能不能谈谈，java GC是在什么时候，对什么东西，做了什么事情？

**哪些对象可以作为GC Roots？ （栈中引用）**

虚拟机栈中引用的对象

方法区中类静态属性引用的对象（static）

方法区中常量引用的对象（final）

本地方法栈中引用的对象

JNI handles，包括global handles和local handles

**Safepoint**： GC的时间点

指一些特定的位置，具体要看jvm的选择，当线程运行到这些位置时，线程的一些状态可以被确定，使JVM可以安全的进行一些操作，比如开始GC。

safepoint指的特定位置主要有:

1. 循环的末尾 (防止大循环的时候一直不进入safepoint，而其他线程在等待它进入safepoint)

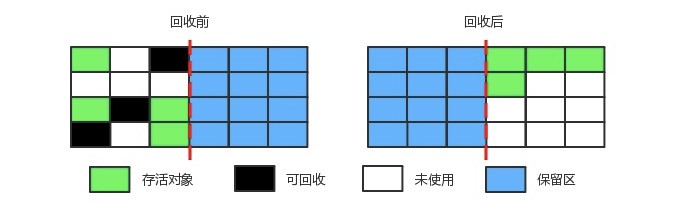
2. 方法返回前

3. 调用方法的call之后

4. 抛出异常的位置

之所以选择这些位置作为safepoint的插入点，主要的考虑是“避免程序长时间运行而不进入safepoint”，比如GC的时候必须要等到Java线程都进入到safepoint的时候VMThread才能开始执行GC，如果程序长时间运行而没有进入safepoint，那么GC也无法开始，JVM可能进入到Freezen假死状态。

新生代算法MinorGC（中文叫做复制算法）：内存回收的时候，标记-》复制，现在的算法整理的时候，会将内存分为两部分。一开始只是使用一部分内存进行分配，之后在清除之后通过copy到另外一个备份的内存区，没有复制的对象相当于被清除。这种算法（1）不需要暂停用户程序，达到并行的目的 （2）不会产生内存碎片。



**Survivor Space**幸存者区，用于保存在eden space内存区域中经过垃圾回收后没有被回收的对象。Survivor有两个，分别为To Survivor、 From Survivor，就是上面看到的两个保留区的思想。执行垃圾回收的时候Eden区域不能被回收的对象被放入到空的survivor（也就是To Survivor，同时Eden区域的内存会在垃圾回收的过程中全部释放），另一个survivor（即From Survivor）里不能被回收的对象也会被放入这个survivor（即To Survivor），然后To Survivor 和 From Survivor的标记会互换，始终保证一个survivor是空的。Eden专门留了一部分内存作为幸存者内存空间。

在新生代又被划分为Eden区，From Sruvivor和To Survivor区，比例是**8:1:1**，所以新生代可用空间其实只有其容量的90%（达到了节省内存的目的）。对象优先被分配在Eden区：

算法过程：

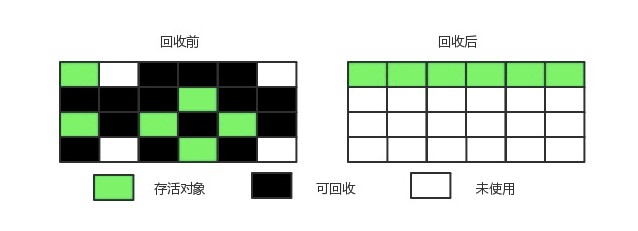
1. Eden+S0可分配新生对象；

2. 对Eden+S0进行垃圾收集，存活对象复制到S1。清理Eden+S0。一次新生代GC结束。

3. Eden+S1可分配新生对象；

4. 对Eden+S1进行垃圾收集，存活对象复制到S0。清理Eden+S1。二次新生代GC结束。

5. goto 1。

老年代MajorGC（FullGC）：只是简单的整理，或者没有整理（如果不进行整理，就会导致碎片化于是不能为大对象分配足够的内存），并没有分成两部分内存。  


在进行MajorGC前一般都先进行了一次MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。MajorGC采用标记‘—清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象，同时容易产生内存碎片，所以还可能采用标记——清除——整理。执行速度比MinorGC慢很多，因为是全内存段的扫描。

System.gc()会建议jvm执行一次Full GC。大量的Full GC会导致CPU占用飙升

垃圾收集器 很老很老的是单线程，有的是多线程。

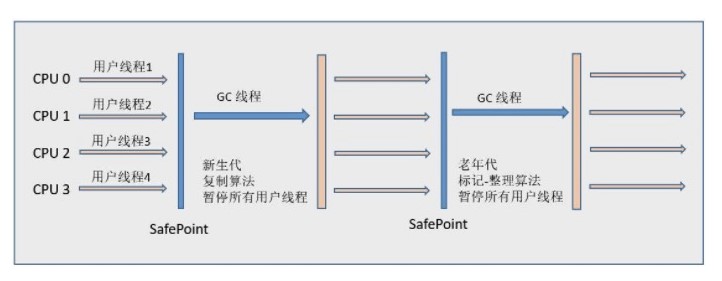
垃圾回收算法执行的时候并行与串行是指能不能在回收的同时，执行用户应用程序代码。除了 CMS 和 G1 之外，其它垃圾收集器都是以串行的方式执行。

垃圾回收算法关注的点只有两个，要么是用户应用程序的停顿时间（客户端需求），要么是CPU时间的吞吐量（服务器需求）。这两者是矛盾的是因为，缩短停顿时间是基于：新生代空间变小，垃圾回收变得频繁。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验。而高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，适合在后台运算而不需要太多交互的任务。可以通过一个开关参数打开 GC 自适应的调节策略（GC Ergonomics），虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。

**浮动垃圾**是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收。

**Serial 收集器**：



它是 Client 模式下的默认新生代收集器，因为在该应用场景下，分配给虚拟机管理的内存一般来说不会很大。Serial 收集器收集几十兆甚至一两百兆的新生代停顿时间可以控制在一百多毫秒以内。所以推荐使用这种单线程的收集器。

Serial Old 是这个收集器的老年代版本。

**ParNew 收集器**：

它是 Serial 收集器的多线程版本。是Server模式下的首选收集器。除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合工作。

默认开启的线程数目与CPU核心数量一样。

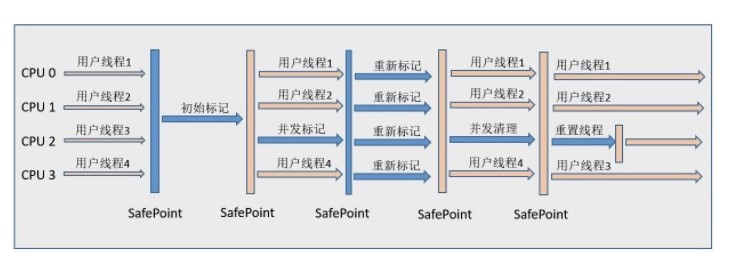


**Parallel Scavenge 收集器**：

一种多线程收集器，关注的目标不是用户线程停顿的时间，而是可控制的吞吐量（CPU 用于运行用户代码的时间占总时间的比值）。

Parallel Old 是这个收集器的老年代版本。

**CMS（Concurrent Mark Sweep）收集器**：



* 初始标记：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿。
* 并发标记：进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。
* 重新标记：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，需要停顿。
* 并发清除：不需要停顿。

缺点：

* 吞吐量低
* 标记——清除算法存在空间碎片
* 不能处理浮动垃圾
* 对于老年代空间没整理，最后会导致一次Full GC

优点

* 响应时间短，适合用户交互场景

**G1（Garbage-First） 收集器**：

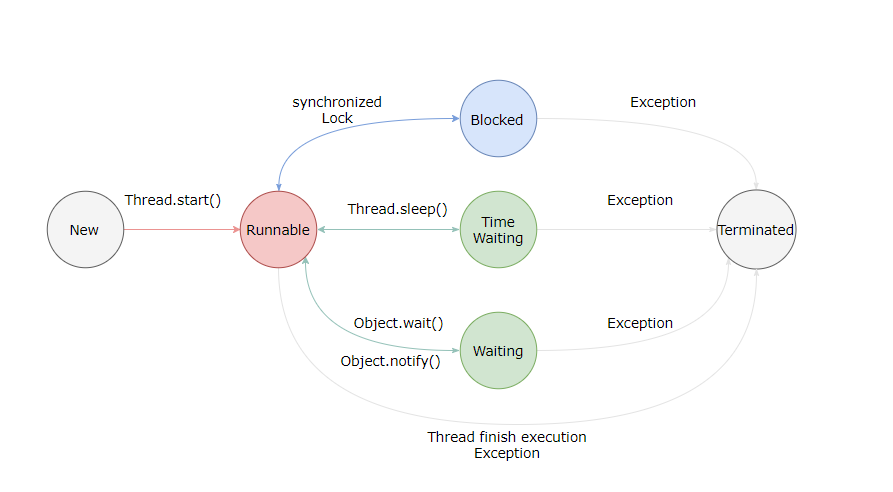
使命是可以替换掉CMS收集器，面向服务器端（不过现在台式机的硬件已经够好了，笔记本的也不错）。目标是维持高内存回收率。Parallel gc目前是最快的。CMS响应时间短。

G1堆的内存布局和其他垃圾收集器不同，它将整个Java堆划分成多个大小相等的独立区域(Region)。G1依然保留了分代收集，但是新生代和老年代不再是物理隔离的，它们都属于一部分Region的集合，因此仅使用G1就可以管理整个堆。

G1能实现可预测的停顿，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。有计划地避免进行全区域地回收，优先回收价值最大的Region。

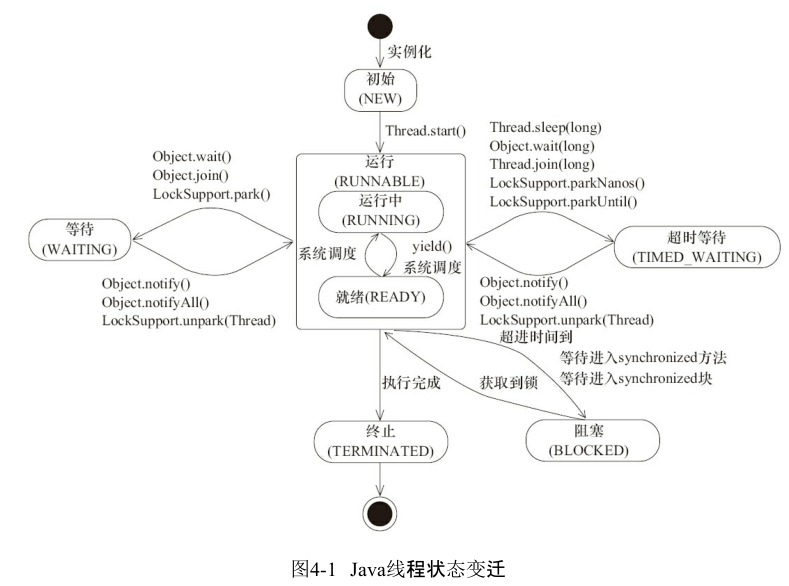
标记——整理算法不会产生空间碎片。G1会将某个区域存活的对象拷贝的其他区域，然后将整个区域整个回收。流程与CMS一致。

144.Java中的线程状态：



上图中Runnable包含了操作系统线程状态中的 Running 和 Ready。

还需要补充一种CAS乐观锁方式。





#### Java中线程的创建方式有哪些？

* 继承Thread并重写run方法
* 实现Runnable并重写run方法，然后作为参数传入Thread
* 实现Callable，并重写call()，call方法有返回值。使用FutureTask包装Callable实现类，其中FutureTask实现了Runnable和Future接口(两者合并后的接口名字RunnableFuture)，最后将FutureTask作为参数传入Thread中
* 由线程池创建并管理线程。

实现 Runnable 和 Callable 接口的类只能当做一个可以在线程中运行的任务，不是真正意义上的线程，因此最后还需要通过 Thread 来调用。可以说任务是通过线程驱动从而执行的。

new Thread(() -> System.out.println("Hello world !")).start();



FutureTask的使用：

public class MyCallable implements Callable<Integer> {

public Integer call() {

return 123;

}

}

public static void main(String[] args) throws ExecutionException, InterruptedException {

MyCallable mc = new MyCallable();

FutureTask<Integer> ft = new FutureTask<>(mc);

Thread thread = new Thread(ft);

thread.start();

System.out.println(ft.get());

}

线程操作函数 suspend()、resume()、stop()因为存在带来的副作用，所以这些方法被标记为不建议使用的过期方法。如果需要，暂停和恢复操作可以使用等待/通知机制来替代。

start() 只要线程规划器空闲，应立即启动线程。并不是保障线程一定会启动。为了方便jstack分析，建议每一个线程起一个独立的名字。

interrupt() 来中断该线程，如果该线程处于阻塞、限期等待或者无限期等待状态，那么就会抛出 InterruptedException（例如Thread.sleep方法）（方法在抛出InterruptedException之前，Java虚拟机会先将该线程的中断标识位清除，然后抛出InterruptedException，此时调用isInterrupted()方法将会返回false），从而提前结束该线程，但是不能中断 I/O 阻塞和 synchronized 锁阻塞；如果是处于一个正在运行的状态，就会设置线程的中断标记，此时调用 interrupted () 方法会返回 true，因此可以在线程执行代码的循环体中使用 interrupted() 方法来判断线程是否处于中断状态，从而提前结束线程。

public static boolean interrupted () {

return currentThread().isInterrupted(true);

}

public boolean isInterrupted () {

return isInterrupted( false);

}

isInterrupted如果传参，这个参数为true，说明返回线程的状态位后，要清掉原来的状态位（恢复成原来情况）。这个参数为false，就是直接返回线程的状态位。只有当前线程能够清楚自己的中断位，interrupted 是作用于当前线程，isInterrupted 是为了得到当前状态。（线程对象对应的线程不一定是当前运行的线程。例如我们可以在A线程中去调用B线程对象的isInterrupted方法。）

setDaemon(true); 一步就设置为jvm中的守护线程（主线程结束的时候，守护线程也会结束），但是不能对已经运行的线程设置为守护线程。在Daemon线程中产生的新线程也是Daemon的（守护进程fork出来的子进程不是守护进程）。举例：servlet的调度线程、jvm一开始的5个线程

join： 先通过等待到差不多的时间，之后自旋判断线程还是否存活，如果存活继续等待wait(0)

public final synchronized void join(long millis) throws InterruptedException {

long base = System.currentTimeMillis();

long now = 0;

if (millis < 0) {

throw new IllegalArgumentException("timeout value is negative");

}

if (millis == 0) {

while (isAlive()) {

wait(0);

}

} else {

while (isAlive()) {

long delay = millis - now;

if (delay <= 0) {

break;

}

wait(delay);

now = System.currentTimeMillis() - base;

}

}

}

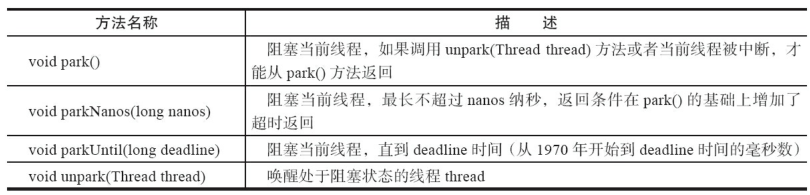
【静态】Thread.yield() 的调用声明了当前线程已经完成了生命周期中最重要的部分，可以切换给其它线程来执行。该方法只是对线程调度器的一个建议，而且也只是建议具有相同优先级的其它线程可以运行。

继承Thread实现多线程对类的功能来讲不是很好，同时因为java不支持多重继承，于是这会导致很多不必要的麻烦。

thread.setPriority(priority); 设置线程的优先级 常量 Thread.MIN\_PRIORITY Thread.MAX\_PRIORITY 优先级的范围从1~10（越小越优先） 默认优先级是5 。一般 设置线程优先级时，针对频繁阻塞（休眠或者I/O操

作）的线程需要设置较高优先级，而偏重计算（需要较多CPU时间或者偏运算）的线程则设置较低的优先级，确保处理器不会被独占。

LockSupport类提供一组静态方法进行线程的阻塞与唤醒：



底层不是使用wait、notify等实现，自己重新看了一个队列，不然notifyAll等会影响到

public static void park(Object blocker) {

Thread t = Thread.currentThread();

setBlocker(t, blocker); // 对应的blcoker会记录在Thread的一个parkBlocker属性中,通过jstack命令可以非常方便的监控具体的阻塞对象

Unsafe.park(false, 0L); // native API，第二个参数指定等待的时间

setBlocker(t, null); // 清除Thread.parkBlocker属性的值

}

Unsafe.park Linux对应pthread\_cond\_wait Windows中对应WaitForSingleObject

park这个方法会阻塞当前线程，只有以下3种情况中的一种发生时，该方法才会返回：

与park对应的unpark执行或已经执行时。“已经执行”是指unpark先执行，然后再执行park的情况。

·线程被中断时。

·等待完time参数指定的毫秒数时。

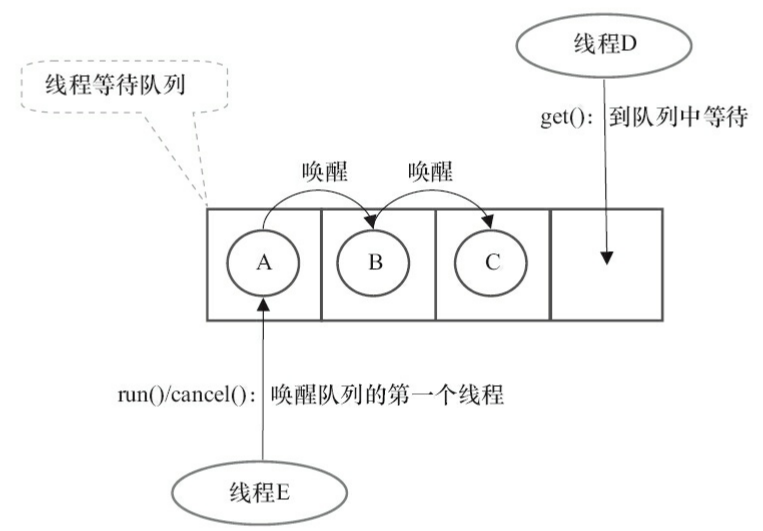
·异常现象发生时，这个异常现象没有任何原因。

**Future接口**：

Future接口定义了任务的运行状态（是否取消以及是否结束）、对运行的操作（取消）、运行结果（一个阻塞等待函数，还有一个是设置了超时的函数）

Sync（ReentrankLock也用于记录状态）是FutureTask的内部私有类，它继承自AQS。用不同的state来表示任务不同的运行状态（1:表示正在运行；2:表示运行完毕；4:表示任务取消）

当执行FutureTask.get()方法时，如果FutureTask不是处于执行完成状态RAN或已取消状态CANCELLED，当前执行线程将到AQS的线程等待队列中等待（见下图的线程A、B、C和D）。当某个线程执行FutureTask.run()方法或FutureTask.cancel（...）方法时，会唤醒线程等待队列的第一个线程（见图10-16所示的线程E唤醒线程A）。最终级联唤醒所有的线程。



**sleep、wait、yield的区别和联系？**

sleep() 允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到CPU 时间，指定的时间一过，线程重新进入可执行状态。调用sleep后不会释放锁。

yield() 使得线程放弃CPU执行时间，但是不使线程阻塞，线程从运行状态进入就绪状态，随时可能再次分得 CPU 时间。有可能当某个线程调用了yield()方法暂停之后进入就绪状态，它又马上抢占了CPU的执行权，继续执行。

wait()是Object的方法，会使线程进入阻塞状态，和sleep不同，wait会同时释放锁。wait/notify在调用之前必须先获得对象的锁。

sleep和wait以及其他锁或者条件变量，都是是线程进入阻塞状态，不过等待的队列不同。

当一个线程处于被阻塞状态或者试图执行一个阻塞操作时（sleep操作），使用Thread.interrupt()方式中断该线程，注意此时将会抛出一个InterruptedException的异常，同时中断状态将会被复位(由中断状态改为非中断状态)

线程中断对于synchronized、LockSupport没有任何响应

异常是一个对象，这个对象不能跨线程传播

**TimerTask**是一个抽象类，实现了Runnable接口。作为一个可以配合Timer类（线程安全）实现定时的任务。联想到一个和时间有关的阻塞队列。

**CountDownLatch**用来控制一个线程等待多个线程（等待这几个线程的执行到某一个步骤）。本质上通过计数器实现，每一个线程结束的时候会调用countDown()导致计数器减 1。当减到0的时候就会唤醒等待的线程。对象不支持被循环使用，因为不支持重新初始化或者设置计数器。

**CyclicBarrier** 用来控制多个线程互相等待，只有当多个线程都到达时，这些线程才会继续执行。本质上也是通过计数器实现，内部的计数器通过调用reset方法可以重置，于是对象支持被循环使用。还提供其他有用的方法，比如getNumberWaiting方法可以获得Cyclic-Barrier阻塞的线程数量。isBroken()方法用来了解阻塞的线程是否被中断。

上面两者不同在于（1）计数器的减一操作进行的时候，一个是线程调用countDown()（完成任务的时候），另外一个是线程调用这个对象的await()（到等待区域的时候） （2）对象能不能支持重用的问题

**Exchanger**用于进行线程间的数据交换（同步点原理）。如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange方法（可以设置超时时间），当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。用于数据校对、遗传算法。

**线程池**：

核心线程池数量：大于这个数量之后，如果有线程空闲时间超时了会将线程回收

好处：降低资源消耗、提高对线程的管理性、提高响应速度（任务到达的时候就可以响应）

线程池执行器将会根据核心线程池数量和最大线程池数量自动地调整线程池大小。如果当前线程池任务线程数量小于核心线程池数量，执行器总是优先创建一个任务线程；如果当前线程池任务线程数量大于核心线程池数量，任务加入阻塞队列，执行器总是优先从线程队列中取一个空闲线程。如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法，后面的具体效果看你设置的策略：

·AbortPolicy：直接抛出异常。

·CallerRunsPolicy：只用调用者所在线程来运行任务。

·DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。

·DiscardPolicy：不处理，丢弃掉。

提供了四种Executor类静态方法：

**newCachedThreadPool** 可以缓存的线程池，如果创建的线程池空间多了，就会自动回收一部分线程；当需求增加的时候，就会自动增加线程，线程池不存在上限。因为使用了SynchronousQueue作为线程池的工作队列。因为没有上限，所以适合处理大量短期的异步任务。

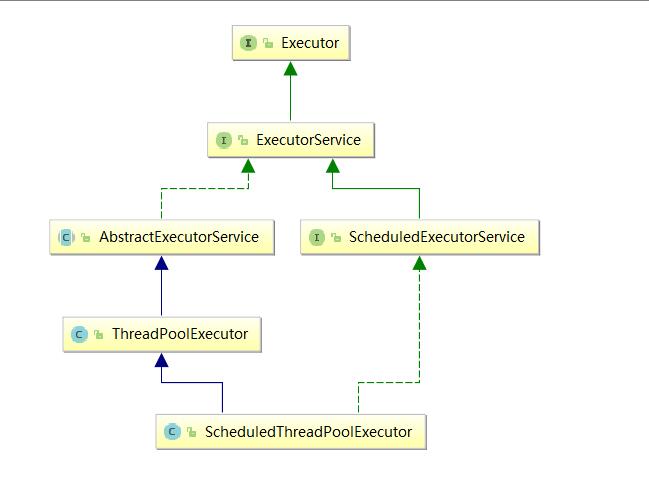
**newFixedThreadPool** 固定数量的线程池

**newScheduledThreadPool 固定数量的线程池，并且支持定时、周期执行任务**

**newSingleThreadExecutor** 单线程的Executor

或者自己通过ThreadPoolExecutor 构造函数实现自定义的线程池

一个Executor的生命周期有三种状态，运行 ，关闭，终止 。Executor创建时处于运行状态。当调用ExecutorService.shutdown()后，处于关闭状态，isShutdown()方法返回true。这时，Executor会拒绝新任务，同时中断正在运行的线程，所有已添加的任务执行完毕后，Executor处于终止状态，isTerminated()返回true。



ExecutorService 开始就是线程池。

执行任务需要实现的**Runnable接口**或**Callable接口**。  
**Runnable接口**或**Callable接口**实现类都可以被**ThreadPoolExecutor**或**ScheduledThreadPoolExecutor**执行。

execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功。

如果只想中断 Executor 中的一个线程，可以通过使用 submit() 方法来提交一个线程，它会返回一个 Future<?> 对象，通过调用该对象的 cancel(true) 方法就可以中断线程。

调用 Executor 的 shutdown() 方法会等待线程都执行完毕之后再关闭并interrupt没有在执行的线程，但是如果调用的是 shutdownNow() 方法，首先将线程池的状态设置成STOP，然后尝试interrupt()所有的正在执行或暂停任务的线程，并返回等待执行任务的列表：  
public static void main(String[] args) {

ExecutorService executorService = Executors.newCachedThreadPool();

executorService.execute(() -> {

try {

Thread.sleep(2000);

System.out.println("Thread run");

} catch (InterruptedException e) {

e.printStackTrace();

}

});

executorService.shutdownNow();

System.out.println("Main run");

}//这段代码会抛出线程被中断后的异常

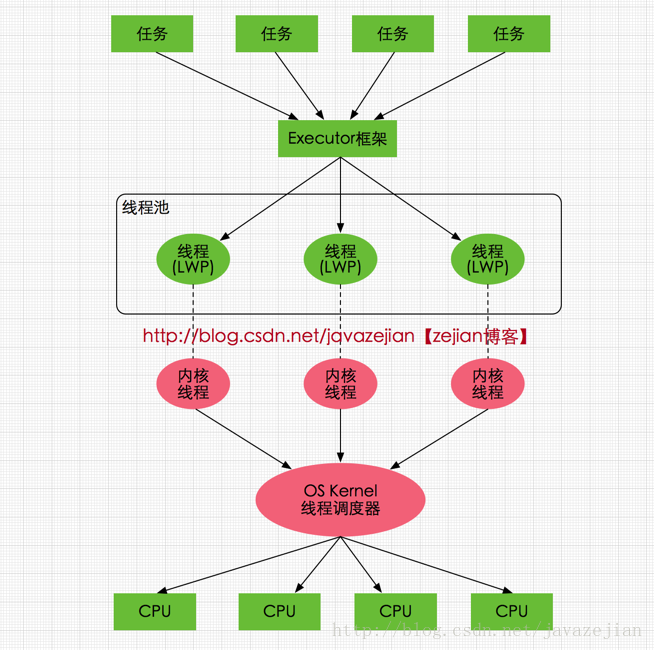
只要调用了这两个关闭方法中的任意一个，isShutdown方法就会返回true。当所有的任务都已关闭后，才表示线程池关闭成功，这时调用isTerminaed方法会返回true。

**Executor 框架：**

在上层，Java多线程程序通常把应用分解为若干个任务，然后使用用户级的调度器

（Executor框架）将这些任务映射为固定数量的线程。

Executor是一个接口，它是Executor框架的基础，它将任务的提交与任务的执行分离开来。ThreadPoolExecutor、ScheduledThreadPoolExecutor（比Timer更灵活）是线程池的核心实现类（实现了ExecutorService接口），用来执行被提交的任务。Future接口和实现Future接口的FutureTask类，代表异步计算的结果。Runnable接口和Callable接口的实现类 作为任务类。工具类Executors可以把一个Runnable对象封装为一个Callable对象（Executors.callable（Runnable task）或Executors.callable（Runnable task，Object resule），可以有返回值）。



java中的线程（LWP，轻量级线程，和内核线程一对一关系，不对啊，多对多关系更好呀，通过系统调用连接用户态与内核态）最后都会映射成为一个内核线程，由操作系统的内核来负责线程的切换（调度）。

用户线程指不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。曾经Java也使用过用户线程，不需要用户态/核心态切换，速度快，操作系统内核不知道多线程的存在，因此一个线程阻塞将使得整个进程（包括它的所有线程）阻塞，即不能很好地利用多核CPU的优势。

获取FutureTask的结果，如果任务还没有完成则当前线程会阻塞，如果我们希望任务完成后就把其结果加到result中，而不用依次等待每个任务完成，可以使CompletionService。生产者submit()执行的任务。使用者take()已完成的任务，并按照完成这些任务的顺序处理它们的结果 。也就是调用CompletionService的take方法是，会返回按完成顺序放回任务的结果，CompletionService内部维护了一个阻塞队列BlockingQueue，如果没有任务完成，take()方法也会阻塞。

import java.util.concurrent.Callable;

import java.util.concurrent.CompletionService;

import java.util.concurrent.ExecutorCompletionService;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class FutureTest {

public static void main(String[] args) throws Exception {

ExecutorService service = Executors.newFixedThreadPool(3);

CompletionService<Integer> completionService = new ExecutorCompletionService<Integer>(service);

for (int i = 0; i < 10; i++) {

final Integer seq = i + 1;

completionService.submit(new Callable<Integer>() {

public Integer call() throws Exception {

Thread.sleep((long) (Math.random() \* 1000));

return seq;

}

});

}

for (int i = 0; i < 10; i++) {

Future<Integer> f = completionService.take();

System.out.println(f.get());

}

service.shutdown();

}

}

[ExecutorService](https://blog.csdn.net/zmx729618/article/details/51441202" \t "_blank)的submit([Callable](https://blog.csdn.net/zmx729618/article/details/51441202" \t "_blank)<[T](https://blog.csdn.net/zmx729618/article/details/51441202" \t "_blank)> task)参数是一个Callable对象,submit(Runnable task)也可以接受Runnable对象；e[xecutor](https://blog.csdn.net/zmx729618/article/details/51441202" \t "_blank)Service.execute([Runnable](https://blog.csdn.net/zmx729618/article/details/51441202" \t "_blank) command)的参数是一个Runnable对象。

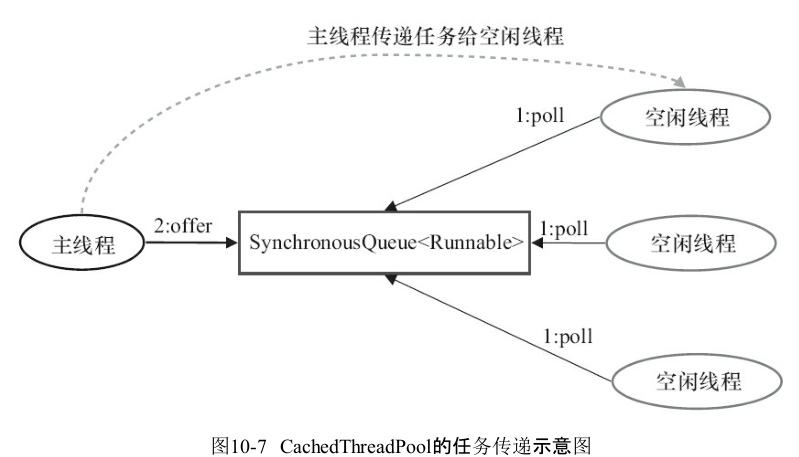
·ScheduledThreadPoolExecutor。包含若干个线程的ScheduledThreadPoolExecutor。

·SingleThreadScheduledExecutor。只包含一个线程的ScheduledThreadPoolExecutor。

线程池与Executor框架一般使用LinkedBlockingQueue作为线程池的工作队列

CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列，所以线程数量没有上限。我心里有点奇怪，为什么不使用PriorityBlockingQueue这种无界队列实现按照任务的先后顺序作为优先级??

首先执行SynchronousQueue.offer（Runnable task）（相当于put），如果这时候有空闲线程正在执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS）（请求任务，最多等待60s，超时之后说明现在没有这么大需求，会终止空闲线程），就会配对成功，主线程将任务交给空闲线程执行。



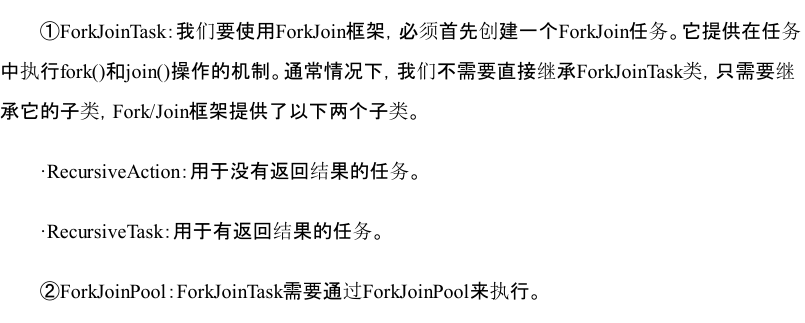
ScheduledThreadPoolExecutor 为了能够支持超时操作，使用DelayQueue 作为无界队列。并不一定是到期了马上取出，DelayQueue 只是按照时间顺序实现了一个优先级队列。take() 需要 ReentrantLock。队列空与满使用条件变量控制。

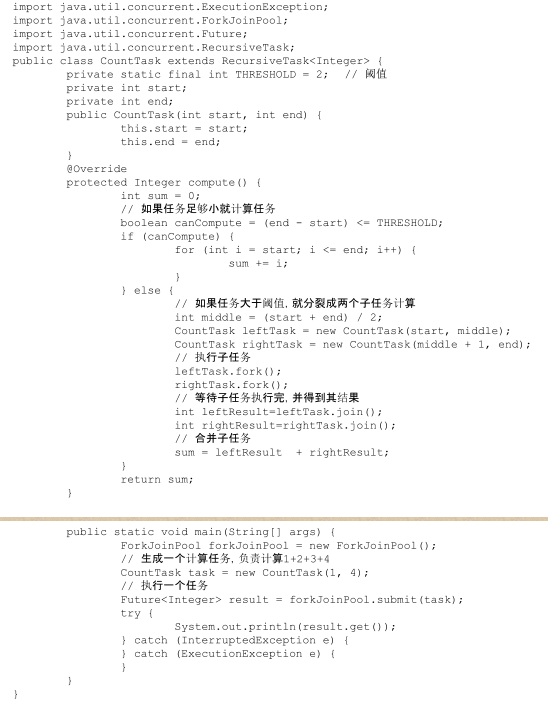
**ForkJoin框架**：

ForkJoin 使用 ForkJoinPool 来启动，它是一个特殊的线程池，线程数量取决于 CPU 核数。

实现了工作窃取（work-stealing）算法来提高 CPU 的利用率。每个线程都维护了一个双端队列，用来存储需要执行的任务。工作窃取算法允许空闲的线程从其它线程的双端队列中窃取一个任务来执行。窃取的任务必须是最晚的任务，避免和队列所属线程发生竞争。线程会执行头节点代表的任务。

算是线程版本的Mapreduce，但是任务的切割需自己编写。





首先一个任务进入队列，被切分之后，变成两个任务重新进入队列，之后原本的任务进入join等待子任务，子任务因为是两个，使用工作窃取。

在ForkJoin框架中声明的compute()方法不允许在运行是抛出异常，因为这个方法的实现没有包含任何throws申明。但是任务抛出异常的时候会被取消，所以ForkJoinTask提供了isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过ForkJoinTask的getException方法获取异常（如果任务被取消了则返回CancellationException；如果任务没有完成或者没有抛出异常则返回null）。

ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成，ForkJoinTask数组负责将存放程序提交给ForkJoinPool的任务，而ForkJoinWorkerThread数组负责提交任务、执行这些任务。

当我们调用ForkJoinTask的fork方法时，程序会调用ForkJoinWorkerThread的pushTask方法异步地把当前任务存放在ForkJoinTask数组队列里。然后再调用ForkJoinPool的signalWork()方法唤醒或创建一个工作线程来执行任务。

join如果任务正常完成会直接返回任务结果，被取消则直接抛出CancellationException，抛出异常则直接抛出对应的异常。

生产者把任务丢给线程池，线程池创建线程并处理任务，如果将要运行的任务数大于线程池的基本线程数就把任务扔到阻塞队列里

任务池的主要处理流程是，每台机器会启动一个任务池，每个任务池里有多个线程池，当

某台机器将一个任务交给任务池后，任务池会先将这个任务保存到数据中，然后某台机器上

的任务池会从数据库中获取待执行的任务，再执行这个任务。任务必须无状态：任务不能在执行任务的机器中保存数据，比如某个任务是处理上传的文件，任务的属性里有文件的上传路径，如果文件上传到机器1，机器2获取到了任务则会处理失败，所以上传的文件必须存在其他的集群里，比如OSS或SFTP。

jvm里面有一个同步队列也有一个等待队列，阻塞是被动的，等待是主动

同步队列（blocking）是为了synchronized、lock

等待队列（waiting）是为了wait、notify、join、LockSupport.park

等待队列会先进入同步队列，之后再变成running

java 管道通信:

面向字节: PipedOutputStream、PipedInputStream

面向字符: PipedReader和PipedWriter

创建输入、输出对象之后，一定要进行连接构成一个管道。使用Runnable等构造函数将其中一个对象传递过去，之后对这个对象进行读写，但是java没有C中做到不同线程应该关闭不相关的端头。

子线程会继承父线程的属性：Daemon、优先级和加载资源的contextClassLoader以及可继承的ThreadLocal （<https://blog.csdn.net/a837199685/article/details/52712547> 实现可继承的ThreadLocal）

java.util.concurrent.BlockingQueue 接口有以下阻塞队列的实现：

* **FIFO 队列** ：LinkedBlockingQueue、ArrayBlockingQueue（固定长度）
* **优先级队列** ：PriorityBlockingQueue

使用方式接近信号量，指定一个长度，然后通过take() 和 put() 方法操控队列中的元素，如果队列为空的时候就阻塞线程：

public class ProducerConsumer {

private static BlockingQueue<String> queue = new ArrayBlockingQueue<>(5);

private static class Producer extends Thread {

@Override

public void run() {

try {

queue.put("product");

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.print("produce..");

}

}

private static class Consumer extends Thread {

@Override

public void run() {

try {

String product = queue.take();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.print("consume..");

}

}

}// 通过阻塞队列实现生产者与消费者

145.等待超时 ：

调用一个方法时等待一段时间（一般来说是给 定一个时间段），如果该方法能够在给定的时间段之内得到结果，那么将结果立刻返回，反之，超时返回默认结果。和锁一样的思想，资源不够的时候阻塞线程，但是这种不能像条件变量一样控制进入临界区的数量。

// 对当前对象加锁

public synchronized Object get(long mills) throws InterruptedException {

long future = System.currentTimeMillis() + mills;

long remaining = mills;

// 当超时大于0并且result返回值不满足要求

while ((result == null) && remaining > 0) {

wait(remaining);

remaining = future - System.currentTimeMillis();

}

return result;

}

146.不能在构造函数中使用Thread，构造函数没有执行完，所以这个引用是有问题。

148.实现生产者-消费者模型？

wait()/notify() //使用的时候配合synchronized

await()/signal()

BlockingQueue （线程安全）

1. 应用编程层次上提升数据库性能：

使用数据库连接池

使用ThreadLocal对象来保存Connection对象有一个好处，它可以保证当前线程中任何方的Connection数据库连接都是相同的。类似的还有Session。

1. java 代理：



**静态代理**：代理类通过实现与目标对象相同的接口，并在类中维护一个代理对象。通过构造器塞入目标对象，赋值给代理对象，进而执行代理对象实现的接口方法，并实现前拦截，后拦截等所需的业务功能。

**动态代理**：基于java反射技术，和静态代理一样要传入一个接口  
1. 通过getProxyClass()生成代理类。JDK生成的最终真正的代理类，它继承自Proxy并实现了我们定义的接口.

2. 通过Proxy.newProxyInstance()生成代理类的实例对象，创建对象时传入InvocationHandler类型的实例。

3. 调用新实例的方法，即此例中的save()，即原InvocationHandler类中的invoke()方法。

invoke函数原型如：public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {...} 传入了对象，方法名，方法参数。

public class StuInvocationHandler<T> implements InvocationHandler {

//invocationHandler持有的被代理对象

T target;

public StuInvocationHandler(T target) {

this.target = target;

}

/\*\*

\* proxy:代表动态代理对象

\* method：代表正在执行的方法

\* args：代表调用目标方法时传入的实参

\*/

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("代理执行" +method.getName() + "方法");

\*/

//代理过程中插入监测方法,计算该方法耗时

MonitorUtil.start();

Object result = method.invoke(target, args);

MonitorUtil.finish(method.getName());

return result;

}

}

public class ProxyTest {

public static void main(String[] args) {

//创建一个实例对象，这个对象是被代理的对象

Person zhangsan = new Student("张三");

//创建一个与代理对象相关联的InvocationHandler

InvocationHandler stuHandler = new StuInvocationHandler<Person>(zhangsan);

//创建一个代理对象stuProxy来代理zhangsan，代理对象的每个执行方法都会替换执行Invocation中的invoke方法。看传入的参数，没想到居然要从类加载器开始创建对象。

Person stuProxy = (Person) Proxy.newProxyInstance(Person.class.getClassLoader(), new Class<?>[]{Person.class}, stuHandler)；

//代理执行上交班费的方法

stuProxy.giveMoney();

}

}

另外一种奇特的写法，android开发的时候也会遇到两种写法：

public Object getProxyInstance(){

return Proxy.newProxyInstance(

target.getClass().getClassLoader(),

target.getClass().getInterfaces(),

new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("开始事务111");

//执行目标对象方法

Object returnValue = method.invoke(target, args);

System.out.println("提交事务111");

return returnValue;

}

}

);

}

**Cglib代理**：

原理是自动生成的代理类继承自被代理类/目标类，所以对final、static方法不会拦截，同时被代理类不能是final修饰。代理类要集成自MethodInterceptor接口。

/\*\*

\* Cglib子类代理工厂

\* 对UserDao在内存中动态构建一个子类对象

\*/

public class ProxyFactory implements MethodInterceptor{

//维护目标对象

private Object target;

public ProxyFactory(Object target) {

this.target = target;

}

//给目标对象创建一个代理对象

public Object getProxyInstance(){

//1.工具类

Enhancer en = new Enhancer();

//2.设置父类

en.setSuperclass(target.getClass());

//3.设置回调函数

en.setCallback(this);

//4.创建子类(代理对象)

return en.create();

}

@Override

public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {

System.out.println("开始事务...");

//执行目标对象的方法

Object returnValue = method.invoke(target, args);

System.out.println("提交事务...");

return returnValue;

}

}

151. 静态函数不能存放在inner class中，除非这个inner class也是static

152 java写交换基本类型的数组两个元素的静态函数，至少需要传递三个函数，因为基本类型是值传递。

153. 打印输出函数们，如果打印的是对象会自动执行toString重载。如果打印非字符串基本类型，我估计是先变成包装类再toString打印。

154. 我们本来写Java接口的时候，是不能有方法体的函数，就类似于C++中的虚函数，default关键字在接口中修饰方法时，方法可以有方法体，来看一个例子。

public interface Default {

default public void method(){

System.out.println("method in interface");

}

}

默认方法在一般的继承规则上，加上“被其它类型所覆盖的方法会被忽略。这条规则适用于超类型共享一个公共祖先的情况”。

使用默认方法，为接口添加方法的时候，就不需要改动所有implents于接口的类。

Spring中最重要的两个思想AOP、IOC

AOP、IOC（DI）举例介绍：<https://www.cnblogs.com/zss-xjx/p/5949215.html>



#### **Spring的生命周期？**

Spring创建、管理对象。Spring容器负责创建对象，装配它们，配置它们并管理它们的整个生命周期。

* 实例化：Spring对bean进行实例化
* 填充属性：Spring将值和bean的引用注入到bean对应的属性中
* 调用BeanNameAware的setBeanName()方法：若bean实现了BeanNameAware接口，Spring将bean的id传递给setBeanName方法
* 调用BeanFactoryAware的setBeanFactory()方法：若bean实现了BeanFactoryAware接口，Spring调用setBeanFactory方法将BeanFactory容器实例传入
* 调用ApplicationContextAware的setApplicationContext方法：如果bean实现了ApplicationContextAware接口，Spring将调用setApplicationContext方法将bean所在的应用上下文传入
* 调用BeanPostProcessor的预初始化方法：如果bean实现了BeanPostProcessor，Spring将调用它们的叛postProcessBeforeInitialization方法
* 调用InitalizingBean的afterPropertiesSet方法：如果bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet方法
* 如果bean实现了BeanPostProcessor接口，Spring将调用它们的postProcessAfterInitialzation方法
* 此时bean已经准备就绪，可以被应用程序使用，它们将一直驻留在应用杀死那个下文中，直到该应用的上下文被销毁。
* 如果bean实现了DisposableBean接口，Spring将调用它的destroy方法。

#### **Spring的配置方式，如何装配bean？bean的注入方法有哪些？**

* XML配置，如<bean id="">
* Java配置即JavaConfig，使用@Bean注解
* 自动装配，组件扫描（component scanning）和自动装配（autowiring），@ComponentScan和@AutoWired注解

bean的注入方式有：

* 构造器注入
* 属性的setter方法注入

推荐对于强依赖使用构造器注入，对于弱依赖使用属性注入。

#### **bean的作用域？**

* 单例（Singleton）：在整个应用中，只创建bean一个实例。
* 原型（Prototype）：每次注入或通过Spring应用上下文获取时，都会创建一个新的bean实例。
* 会话（Session）：在Web应用中，为每个会话创建一个bean实例。
* 请求（Request）：在Web应用中，为每个请求创建一个bean实例。

默认情况下Spring中的bean都是单例的。

#### **Spring中涉及到哪些设计模式？**

* 工厂方法模式。在各种BeanFactory以及ApplicationContext创建中都用到了；
* 单例模式。在创建bean时用到，Spring默认创建的bean是单例的；
* 代理模式。在AOP中使用Java的动态代理；
* 策略模式。比如有关资源访问的Resource类
* 模板方法。比如使用JDBC访问数据库，JdbcTemplate。
* 观察者模式。Spring中的各种Listener，如ApplicationListener
* 装饰者模式。在Spring中的各种Wrapper和Decorator
* 适配器模式。Spring中的各种Adapter，如在AOP中的通知适配器AdvisorAdapter

#### **MyBatis和Hibernate的区别和应用场景？**

Hibernate :是一个标准的ORM(对象关系映射) 框架； SQL语句是自己生成的，程序员不用自己写SQL语句。因此要对SQL语句进行优化和修改比较困难。适用于中小型项目。

MyBatis： 程序员自己编写SQL， SQL修改和优化比较自由。 MyBatis更容易掌握，上手更容易。主要应用于需求变化较多的项目，如互联网项目等。

1. JNI:

**为什么要使用JNI ?**

* 复用很多优秀的c/c++代码
  + ffmpeg 多媒体播放器
  + opencv 图形识别引擎
  + 7-zip 压缩
  + opencore 视频播放
* 效率问题
  + java代码跨平台，不直接操作硬件（调用native库），虚拟机解释执行（jvm基于native）。垃圾回收机制。
  + art 在安装apk应用程序的时候，把apk里面的dex翻译成机器码（apk的体积会变大）
  + c代码直接操作硬件，程序员可以手工的释放内存
* 应用场景问题
  + c代码可以直接操作硬件。
  + java代码只能操作虚拟机(一次编译到处执行)
  + 智能家居操作硬件
  + 车载电脑（obd: onboard debug system)
* 特殊业务逻辑
  + c代码反编译困难，调试困难。
  + java代码反编译容易，很容易看懂。

推荐学习： <https://www.zybuluo.com/cxm-2016/note/563686>