目录

实战JAVA虚拟机

1.1. hp-note

 updated 2017-11-27 22:25:04

1.2. 高端JAVA基础

小数和计算问题，二进制不能精确的展示小数，所以会有 double a = 0.1

实际上 可能是0.1000000000000000000000000511100 这样的值， 所以为了更精确就有了BigIntger和BigDecmeg。 他们使用的不同的方式来实现的

 updated 2017-11-28 22:20:59

1.2.1. 基本数据类型

> 官网教程点这里

按类型分：

1. 字符类型：char
2. 布尔类似：boolean
3. 数值类型：byte、short、int、long、float、double

按长度分：

1. 未明确定义：boolean
2. 8位（1byte）：byte
3. 16位：char、short
4. 32位：int、float 5：64位：long，double

上面的8中类型，除去boolean和char外，其他的都是有符号的，范围大小是负数~正数。char数据类型是一个16位Unicode字符。它的最小值为'\u0000'（或0），最大值为'\uffff'（或65,535）。

1.2.1.1. 位运算

位运算常用在 网络通信、协议解析、高性能编程中。

左移操作 `<<`

丢弃最高位，用*0*补位

左移操作，数字没有溢出的前提下，对于正数和负数，左移一位都相当于乘以2^1

右移操作 `>>`

符号位不变，左边补上符号位

正数右移，左边补0，负数右移，左边补1

无符号右移操作 `>>>`

无符号始终补*0*

按位与 `&`

只有 *1 & 1 = 1*

```
0000 0011  // 十进制 3
0000 0010  // 十进制 2
--------- 按位与
0000 0010  // 结果十进制为 2
```

按位或 `|`

只有 *0 | 0 = 0*

按位取反 `~`

对二进制按位取反，即 *0* 变成 *1*，*1* 变成 *0*

按位异或 `^`

只有 *1 ^ 0 = 1* 或则 *0 ^ 1 = 1*

1.2.1.1.1. 位运算总结

1. 位运算常用在 网络通信、协议解析、高性能编程中比较常见。
2. 位运算是针对整型的，进行位操作时，除long外，其他的类型会自动转成int

   ```
   如：byte ba   127 ba    2
   这个运算，ba变量其实扩展成了int
   ```

3. 如果移动的位数超过了32位(long是64位)，那么编译器会对移动的位数取模

   如：对int移动33位，33 % 32 = 1，实际上只移动了一位

1.2.1.2. 位运算实际应用举例

1.2.1.2.1. 获取数值的16进制变形形式

下面这个例子：把变量b打印出它的16进制表现形式

```
    char hex
            '0'  '1'  '2'  '3'  '4'  '5'  '6'  '7'  '8'  '9'
            'a'  'b'  'c'  'd'  'e'  'f'
```

```
int b   0xf1
// b = 0xf1
System out println "b = 0x"   hex  b    4    0x0f    hex b   0x0f
```

以上程序有1个固定的16进制常量：0xF1 转换为二进制是：1111 0001

以下有几个比较常用的16进制常量：

- 0xFF : 一个F对应4个1，即：1111 1111
- 0x0F ： 0000 1111

有关进制之间的转换，详看 进制 章节

以上程序计算分解步骤如下：

在上面说到过，在位计算的时候其实会扩展成int来计算，一个int其实是32位， 这里为了方便观看，使用8位来表示

`hex[(b >> 4) & 0x0f]`

```
* b 的二进制形式为：1111 0001
* 右移4位(高位补0)：0000 1111
* 按位与(1 & 1 = 1)：0000 0001
* 结果转为十进制为 15，对于hex数组中的f
```

`hex[b & 0x0f])` 和上面类似。

作用就是在用位运算，取出每一个4位，得到这个4位对应的16进制表现字符

技巧：利用了 位运算 按位与(1 & 1 = 1) 和 16进制 换算方便的常量，把不需要的位变成了0，再配合右移操作，得到目标数值

上面的程序可以使用一个循环来得到：

```
char hex
        '0'  '1'  '2'  '3'  '4'  '5'  '6'  '7'  '8'  '9'
        'a'  'b'  'c'  'd'  'e'  'f'

byte b    byte  0xf1
StringBuffer sb   new StringBuffer "0x"
for  int i   1  i    2  i
    int num    2   i    4
    char t   hex  b    num    0x0f
    sb append t

System out println sb
```

1.2.1.2.2. 把有符号数字转为无符号

```
    /**
     * 把data转换为无符号数字；
     * @param data
     * @return 返回 0 - 255 的数字
     */
    public static int getUnsignedByte byte data
        // data & 1111 1111;
        // 只有 1 & 1 = 1;
        /**
         * 1000 0010  -2的源码
         * 1111 1101  -2的反码
         * 11111111111111111111111111 1111 1110  -2的补码
         * 00000000000000000000000000 1111 1111
         * --------------------------------
         * 00000000000000000000000000 1111 1110
         */
        // 要记住一点，而且是非常重要：位运算是针对整型的，进行位操作时，除long外，其他的类型
会自动转成int
        // 所以这里为什么 & 运算之后，一个负数就变成了正数。
        // 这里我按照8位来算，始终都没有搞明白为什么8位的开头是1，却是一个正数
        // java 中的类似是有符号的
//        return Byte.toUnsignedInt(data);
        return data   0xFF
```

1.2.1.2.3. 网络协议中的位运算

```
0xf0   Msg Bytes 消息体
```

0xf0 表示报文类型，1字节，无符号；十进制是240

那么Java中的Byte是有符号的，如果传递的是1字节过来，那么在Java中会解析为一个负数。

用上一小结的 位运算方法能获取到无符号的int数字。

```
01101000   Msg bytes
```

在前面使用无符号的byte来充当报文类型，感觉还是太浪费空间了，那么这里用1byte来标识更多的信息可以这样做：使用二进制，1byte有8个bit位。

比如：规定第三位到第5位(不包含5，索引从1开始) 也就是 01 [10] 1000 中括号括起来的位置来表示消息的紧急程度；

我们该如何正确的获取到这个中括号中标识的十进制呢？

使用按位与和位移操作能获取到；如下代码

```
// 01101000
byte head    byte  104
// 使用按位与 把第三和第4位提取出来
int x   head   0b0011_0000
// 把第5位开始的都移走，让目标数字变成最低位
// 由于上面使用了按位与，所以把不相关的位都变成了0
// 这里再使用位移操作就得到了我们想要的值
System out println x    4   // 十进制2
```

1.2.1.3. 自动装箱的秘密

`Integer i = 100` 相当于编译器自动为您作以下的语法编译： `Integeri= Integer.valueOf(100);`

读过源码的都知道，该方法默认-128 ~ 127 之间的数字使用同一个缓存Integer对象，超过该值的就new一个新的对象返回。

> 在Java8的class中我没有看到编译器的转换Integer.valueOf(100)。在8以前我的确是看到有这样的转换。所以难道是java8的机制变了？

那么为什么要这样设计呢？就涉及到了一个空间的问题；

1.2.1.3.1. 自动装箱的代价

在Java中存储一个对象大概是下面这样

```
对象头 8字节    对象里的各种属性值    padding填充位
```

请容许我哭一会儿，在百度找了半天没有找到更深入的资料；

就这样大概来看吧。基本类型没有这些额外的数据来记录信息，而包装类有，所以 就很明白了为啥会设置一个缓存机制；

这里看来，HashMap的key只允许包装类，得浪费多少内存

1.2.1.4. Java小数的问题

 updated 2018-02-05 23:42:58

1.2.2. 进制

1.2.2.1. 十六进制转二进制

十进制与十六进制对应关系

```
0    1    2    3    4    5    6    7    8
---------------------------------------
0    1    2    3    4    5    6    7    8

9    10    11    10    13    14    15
---------------------------------------
9    A     B     C     D     E     F
```

16进制与二进制对应关系

```
 0     1     2     3      4    5      6    7
-----------------------------------------------
0000  0001  0010  0011  0100  0101  0110  0111

 8     9     A      B     C     D     E     F
-----------------------------------------------
1000  1001  1010   1011  1100  1101  1110  1111
```

二进制转成十六进制方法是：取四合一法，既从二进制的小数点为分界点，向左或向右每4位取成一位

```
101110011011.1001
------------------
1011  1001  1011 . 1001

每4位表示一个16位
```

组分好以后，转换成16进制，小数点不变

```
1011  1001  1011 . 1001
----------------------
 B     9     B    .   9
```

在Java中 0xB 表示十进制的11，其他的也有用H作为后缀的，

还需要注意，位数不足的时候，用0补位。(注意是小数点为分界)

```
10111.011
---------------
0001  0111 . 0110
----------------
 1      7  .   6              对应的十进制

可以看到从小数点往左开始分组，然后最左边被补位了

但是我没有明白的是 负数也用0补位吗？
```

16进制转二进制

```
 B      F    4   .    B     5
------------------------------
1011  1111  0100 .  1011   0101
```

 updated 2018-01-29 10:08:26

1.2.3. 源码、反码、补码

> 摘抄至 http://www.linuxidc.com/Linux/2015-02/113863.htm

以下例子

Demo	7	-7
原码	00000111	10000111
反码	00000111	11111000
补码	00000111	11111001

可以看出来，正数的码都一样，负数的反码：原码中除去符号位，其他的数值位取反，0变1，1变0。负数的补码：其反码+1.

本篇文章讲解了计算机的原码, 反码和补码. 并且进行了深入探求了为何要使用反码和补码, 以及更进一步的论证了为何可以用反码, 补码的加法计算原码的减法.

1.2.3.1. 机器数和真值

在学习原码, 反码和补码之前, 需要先了解机器数和真值的概念.

1.2.3.1.1. 机器数

一个数在计算机中的二进制表示形式, 叫做这个数的机器数。机器数是带符号的，在计算机用一个数的最高位存放符号, 正数为0, 负数为1.

比如，十进制中的数 +3 ，计算机字长为8位，转换成二进制就是00000011。如果是 -3 ，就是10000011 。

那么，这里的 00000011 和 10000011 就是机器数。

1.2.3.1.2. 真值

因为第一位是符号位，所以机器数的形式值就不等于真正的数值。例如上面的有符号数 10000011，其最高位1代表负，其真正数值是 -3 而不是形式值131（10000011转换成十进制等于131）。所以，为区别起见，将带符号位的机器数对应的真正数值称为机器数的真值。

例：0000 0001的真值 = +000 0001 = +1，1000 0001的真值 = –000 0001 = –1

1.2.3.2. 原码, 反码, 补码的基础概念和计算方法.

在探求为何机器要使用补码之前, 让我们先了解原码, 反码和补码的概念.对于一个数, 计算机要使用一定的编码方式进行存储. 原码, 反码, 补码是机器存储一个具体数字的编码方式.

1.2.3.2.1. 原码

原码就是符号位加上真值的绝对值, 即用第一位表示符号, 其余位表示值. 比如如果是8位二进制:

```
原 0000 0001   1

原 1000 0001   -1
```

第一位是符号位. 因为第一位是符号位, 所以8位二进制数的取值范围就是:

```
 1111 1111 , 0111 1111

即

 -127 , 127
```

原码是人脑最容易理解和计算的表示方式.

1.2.3.2.2. 反码

反码的表示方法是:

- 正数的反码是其本身
- 负数的反码是在其原码的基础上, 符号位不变，其余各个位取反.

```
 +1     00000001 原    00000001 反

 -1     10000001 原    11111110 反
```

可见如果一个反码表示的是负数, 人脑无法直观的看出来它的数值. 通常要将其转换成原码再计算.

1.2.3.2.3. 补码

补码的表示方法是:

- 正数的补码就是其本身
- 负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1. (即在反码的基础上+1)

```
+1    00000001 原    00000001 反    00000001 补

-1    10000001 原    11111110 反    11111111 补
```

对于负数, 补码表示方式也是人脑无法直观看出其数值的. 通常也需要转换成原码在计算其数值.

1.2.3.3. 为何要使用原码, 反码和补码

在开始深入学习前, 我的学习建议是先"死记硬背"上面的原码, 反码和补码的表示方式以及计算方法.

现在我们知道了计算机可以有三种编码方式表示一个数. 对于正数因为三种编码方式的结果都相同:

```
+1    00000001 原    00000001 反    00000001 补

所以不需要过多解释，但是对于负数：

-1    10000001 原    11111110 反    11111111 补
```

可见原码, 反码和补码是完全不同的. 既然原码才是被人脑直接识别并用于计算表示方式, 为何还会有反码和补码呢?

首先, 因为人脑可以知道第一位是符号位, 在计算的时候我们会根据符号位, 选择对真值区域的加减. (真值的概念在本文最开头). 但是对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单. 计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法. 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: 1-1 = 1 + (-1) = 0 , 所以机器可以只有加法而没有减法, 这样计算机运算的设计就更简单了.于是人们开始探索 将符号位参与运算, 并且只保留加法的方法. 首先来看原码:

计算十进制的表达式: 1-1=0

```
1 - 1   1 +  -1     00000001 原 +  10000001 原    10000010 原   -2
```

如果用原码表示, 让符号位也参与计算, 显然对于减法来说, 结果是不正确的.这也就是为何计算机内部不使用原码表示一个数.

为了解决原码做减法的问题, 出现了反码:

计算十进制的表达式: 1-1=0

```
1 - 1   1 +  -1
   0000 0001 原 +  1000 0001 原
   0000 0001 反 +  1111 1110 反
   1111 1111 反
   1000 0000 原
  -0
```

发现用反码计算减法, 结果的真值部分是正确的. 而唯一的问题其实就出现在"0"这个特殊的数值上. 虽然人们理解上+0和-0是一样的, 但是0带符号是没有任何意义的. 而且会有[0000 0000]原和[1000 0000]原两个编码表示0.

于是补码的出现, 解决了0的符号以及两个编码的问题:

```
1-1   1 +  -1
   0000 0001 原 +  1000 0001 原
   0000 0001 补 +  1111 1111 补
   0000 0000 补
   0000 0000 原
```

这样0用[0000 0000]表示, 而以前出现问题的-0则不存在了.而且可以用[1000 0000]表示-128:

```
 -1  +  -127
   1000 0001 原 +  1111 1111 原
   1111 1111 补 +  1000 0001 补
   1000 0000 补
```

疑问：这个相加不是超过了8位数了？溢出了？没搞明白

-1-127的结果应该是-128, 在用补码运算的结果中, [1000 0000]补 就是-128. 但是注意因为实际上是使用以前的-0的补码来表示-128, 所以-128并没有原码和反码表示.(对-128的补码表示[1000 0000]补算出来的原码是[0000 0000]原, 这是不正确的)

使用补码, 不仅仅修复了0的符号以及存在两个编码的问题, 而且还能够多表示一个最低数. 这就是为什么8位二进制, 使用原码或反码表示的范围为[-127, +127], 而使用补码表示的范围为[-128, 127].

因为机器使用补码, 所以对于编程中常用到的32位int类型, 可以表示范围是: [-231, 231-1] 因为第一位表示的是符号位.而使用补码表示时又可以多保存一个最小值.

1.2.3.4. 原码, 反码, 补码 再深入

计算机巧妙地把符号位参与运算, 并且将减法变成了加法, 背后蕴含了怎样的数学原理呢?

将钟表想象成是一个1位的12进制数. 如果当前时间是6点, 我希望将时间设置成4点, 需要怎么做呢? 我们可以:

1. 往回拨2个小时: 6 - 2 = 4
2. 往前拨10个小时: (6 + 10) mod 12 = 4
3. 往前拨10+12=22个小时: (6+22) mod 12 =4

2,3方法中的mod是指取模操作, 16 mod 12 =4 即用16除以12后的余数是4.所以钟表往回拨(减法)的结果可以用往前拨(加法)替代!

现在的焦点就落在了如何用一个正数, 来替代一个负数. 上面的例子我们能感觉出来一些端倪, 发现一些规律. 但是数学是严谨的. 不能靠感觉.

首先介绍一个数学中相关的概念: 同余

1.2.3.4.1. 同余的概念

两个整数a，b，若它们除以整数m所得的余数相等，则称a，b对于模m同余

记作 a ≡ b (mod m) 读作 a 与 b 关于模 m 同余。

举例说明:

```
4 mod 12   4
16 mod 12   4
28 mod 12   4
```

所以4, 16, 28关于模 12 同余.

1.2.3.4.2. 负数取模

正数进行mod运算是很简单的. 但是负数呢?

下面是关于mod运算的数学定义: $x \bmod y = x - y\lfloor x/y \rfloor, \quad \text{for } y \neq 0.$

上面是截图, "取下界"符号找不到如何输入(word中粘贴过来后乱码). 下面是使用"L"和"J"替换上图的"取下界"符号:

```
x mod y   x - y L x / y J
```

x mod y等于 x 减去 y 乘上 x与y的商的下界.

以 -3 mod 2 举例:

```
-3 mod 2
```

```
  -3 - 2 x L -3/2 J
  -3 - 2 x L-1.5J
  -3 - 2 x  -2
  -3 + 4   1

所以:

 -2  mod 12   12-2 10

 -4  mod 12   12-4   8

 -5  mod 12   12 - 5   7
```

1.2.3.4.3. 开始证明

再回到时钟的问题上:

```
回拨2小时 ≡ 前拨10小时
回拨4小时 ≡ 前拨8小时
回拨5小时 ≡ 前拨7小时
```

注意, 这里发现的规律!

结合上面学到的同余的概念.实际上:

```
 -2  mod 12   10
10   mod 12   10
```

-2与10是同余的.

```
 -4  mod 12   8
8    mod 12   8
```

距离成功越来越近了. 要实现用正数替代负数, 只需要运用同余数的两个定理:

反身性: `a ≡ a (mod m)` ,这个定理是很显而易见的.

线性运算定理:

```
如果
 a ≡ b  mod m , c ≡ d  mod m
那么:
 1 a ± c ≡ b ± d  mod m

 2 a * c ≡ b * d  mod m
```

如果想看这个定理的证明, 请看:http://baike.baidu.com/view/79282.htm

所以:

```
7 ≡ 7  mod 12
 -2  ≡ 10  mod 12
7 -2 ≡ 7 + 10  mod 12
```

现在我们为一个负数, 找到了它的正数同余数. 但是并不是7-2 = 7+10, 而是 `7 -2 ≡ 7 + 10 (mod 12)` , 即计算结果的余数相等.

接下来回到二进制的问题上, 看一下: 2-1=1的问题.

```
2-1   2+ -1
   0000 0010 原 +  1000 0001 原
   0000 0010 反 +  1111 1110 反
```

先到这一步, -1的反码表示是1111 1110. 如果这里将[1111 1110]认为是原码, 则 [1111 1110]原 = -126, 这里将符号位除去, 即认为是126.

发现有如下规律:

```
 -1  mod 127   126
126  mod 127   126
即
 -1  ≡ 126  mod 127
2-1 ≡ 2+126  mod 127
```

> !我发现还是理解不了负数取模。怎么变成126的 2-1 与 2+126的余数结果是相同的! 而这个余数, 正式我们的期望的计算结果: 2-1=1;

所以说一个数的反码, 实际上是这个数对于一个膜的同余数. 而这个膜并不是我们的二进制, 而是所能表示的最大值! 这就和钟表一样, 转了一圈后总能找到在可表示范围内的一个正确的数值!

而2+126很显然相当于钟表转过了一轮, 而因为符号位是参与计算的, 正好和溢出的最高位形成正确的运算结果.

既然反码可以将减法变成加法, 那么现在计算机使用的补码呢? 为什么在反码的基础上加1, 还能得到正确的结果?

```
2-1 2+ -1
   0000 0010 原 +  1000 0001 原
   0000 0010 补 +  1111 1111 补
```

如果把[1111 1111]当成原码, 去除符号位, 则:

```
 0111 1111 原   127
```

其实, 在反码的基础上+1, 只是相当于增加了膜的值:

```
 -1  mod 128   127
127 mod 128   127
2-1 ≡ 2+127  mod 128
```

此时, 表盘相当于每128个刻度转一轮. 所以用补码表示的运算结果最小值和最大值应该是[-128, 128].

但是由于0的特殊情况, 没有办法表示128, 所以补码的取值范围是[-128, 127]

1.2.3.5. int强制转换为byte用补码原码的知识得到正确的值

考虑以下强转

```
byte a    byte 508
```

```
0000 0001 1111 1100   // 508
1111 1100  // 强制转换后，取最低8位，因为是一个负数，所以这个是补码,十进制 -124
1111 1011  // 根据规则，符号位不变，+1，十进制 -123
1000 0100  //
```

这个减1，不用全部换算成十进制来算。右边数第3位 `1111 1100` 等于1，换算成十进制等于4，4-1等于3，那么当前位变成0，右边开数1，2位变成1，就组合成了3

 updated 2017-12-01 16:22:40

1.2.4. Java常用排序算法

> 本文摘抄至:
>
> - http://blog.csdn.net/daogepiqian/article/details/50770404
> - http://blog.csdn.net/wuqilianga/article/details/52798728

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数						

排序大的分类可以分为两种：内排序和外排序。在排序过程中，全部记录存放在内存，则称为内排序，如果排序过程中需要使用外存，则称为外排序。下面讲的排序都是属于内排序。　　内排序有可以分为以下几类：　　(1)、插入排序：直接插入排序、二分法插入排序、希尔排序。(2)、选择排序：简单选择排序、堆排序。　　(3)、交换排序：冒泡排序、快速排序。　　(4)、归并排序　　(5)、基数排序

1.2.4.1. 插入排序

思想：每步将一个待排序的记录，按其顺序码大小插入到前面已经排序的字序列的合适位置，直到全部插入排序完为止。

关键问题：在前面已经排好序的序列中找到合适的插入位置。

又分为几种实现方式

- 直接插入排序
- 二分插入排序
- 希尔排序

1.2.4.1.1. 直接插入排序

基本思想：每步将一个待排序的记录，按其顺序码大小插入到前面已经排序的字序列的合适位置（从后向前找到合适位置），直到全部插入排序完为止。

算法步骤：

1）将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。

2）从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置(从后向前找到合适位置)。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

```
int   a    65  49  38  97  76  13  27  49  78  34  12  64  1
System out println Arrays toString a

for  int i   1  i   a length  i
    int temp   a i   // 待插入元素
    int k
    for  k   i   1  k    0  k
        if  a k    temp
            // 把大于目标值的元素往后移动一位，直到没有比目标元素大的时候
            // 这里最重要的不是交换位置，而是直接往后移动，覆盖掉了目标元素，而目标元素存储在
第3个变量中
            a k   1    a k
          else
            break


    a k   1    temp
```

比如这里的详细流程分析

```
第一次：
    i 1  temp   49
    比较有序列表：
        第一次：k 0，a k  65，65   49？
                移动前：65，49，38，97，76
                移动后：65，65，38，97，76
                内部循环退出后，K--   -1 a k+1    a 0    temp
                把待插入的值插入后：49,65,38,97,76
第三次：
    i 2 temp   38
    比较有序列表：
        第一次：k 1,a 1   65，65   38？
            移动前：49,65,38,97,76
            移动后：49,65,65,97,76
            k--   0；符合条件再次循环
        第二次：k 0,a 0  49，49   38？
            移动前：49,65,65,97,76
            移动后：49,49,65,97,76
            k--   -1,不满足条件，退出内部循环。a k+1    a 0    38
            插入后：38,49,65,97,76
```

1.2.4.2. 快速排序

```
    public void q7
//        int[] arr = {6, 8, 1, 3, 2, 6, 4};
        int   arr    49  38  65  97  76  13  27  49  78  34  12  64  1  8
        System out println "排序之前："   Arrays toString arr
        quick arr
        System out println "排序之后："   Arrays toString arr


    private static void quick int   a
        if  a length   0
            quickSort a  0  a length   1


    private static void quickSort int   arr  int low  int hight
        if  low   hight
            int middle   getMiddle arr  low  hight
            quickSort arr  low  middle   1
            quickSort arr  middle   1  hight


    private static int getMiddle int   a  int low  int hight
```

```
        // 分5步,核心分4步
        /**
         * 1.从低位找一个基准值
         * 2.从高位开始降低对比找到比基准值还小的数值，然后把这个找到的放到当前的低位
         * 3.从低位开始增高对比找到比基准值还大的数值，然后把这个找到的放到当前的高位
         * 4.把当前的低位放入基准值，这个就是一个分界点，左边的小于基准值，右边的大于等于基准
值
         * 5.把左右两拨分开递归调用重复上面的1，2，3，4步骤。
         */
        int base   a low   // 基准值
        while  low   hight
            System out println "----------------------------"
            System out println "s:"   Arrays toString a
            // 找到比基准值小的数
            while  low   hight    base    a hight
                hight

            //找到比基准值小的，则把基准所在的位置 设置上这个小的值
            a low    a hight
            System out println "m:"   Arrays toString a
            // 找到比基准值大的数
            while  low   hight    base    a low
                // 如果从左边开始，比基准值小的话，那么要把索引增加，相当于遍历
                low

            a hight    a low
            System out println "e:"   Arrays toString a

        a low    base
        return low  // 返回这个中间值
```

```
基准值 6
---------------------------- 第一轮
s: 6, 8, 1, 3, 2, 6, 4     // 在高位找到比基准值小的值
m: 4, 8, 1, 3, 2, 6, 4     // 找到后，直接把当前的低位（最开始是拿第一个值作为低位的）换成这
个小值
e: 4, 8, 1, 3, 2, 6, 8     // 在低位找到比基准值大的值，
                           // 上一步已经把高位中的一个值存放到基准上面了，所以空缺,就把这个
高位插入找到的大值
---------------------------- 第二论
s: 4, 8, 1, 3, 2, 6, 8
m: 4, 2, 1, 3, 2, 6, 8
e: 4, 2, 1, 3, 2, 6, 8

------------------------- 从两头往中间对比，最后到达中间位置没有可比较的了
 4, 2, 1, 3, 6, 6, 8          // 把当前的低位也就是中间位赋值上基准值，就如同这里，基准值左边
的始终比基准值小，右边的始终比基准值大,或则相等。

// 然后分成两拨，继续上面的步骤，直到最后没有可比了，则整个列表就都是有序的了
```

 updated 2017-12-02 23:28:46

1.3. 高端基础作业

> GIBOOK在线地址
>
> GITHUb在线地址
>
> - 未翻墙打不开Gitbook的话可以使用git上的查看，也可以下载该项目，在本地使用gitbook server启动服务
> - gitbook在线或则本地启动的服务浏览效果比浏览git源文件要好很多很多
>
> GITHUB练习题项目地址

git目录规范说明：

```
|- _01              # 第N课
 |- q01             # 第N题包
  |- Practice       # 第N题测试入口
 |- q02
```

笔记规范：

```
===第 N 课
  ===第 N 题
    如果是有代码练习的答案，则放上git上的测试入口；

    ===题目
     原题题目
    ===答案
     练习代码的解题思路，或则程序输出；
     如果代码篇幅很小就几行，就直接贴在这里，很大就只能跳转到git上观看
```

以上目录规范，从第02课严格执行，第01课中后面部分执行

1.3.1. 笔记模版

```
## 第 07 题
> [习题答案测试入口请右键新窗口查看GIT](链接)
> 这里未体现出来的回答都在该测试里面能看到

### 题目
xxxxx

### 答案
```

```
先讲解思路;
程序输出
```

 updated 2018-03-04 11:37:27

1.3.1. 第01课

1.3.1.1. 本课知识点：

- 基本数据类型
- 原码、反码、补码 - JVM ；原码、反码、补码

1.3.1.2. 第01题

以下程序为什么会出错？给出解释，并且纠正错误

```
byte ba 127

byte bb ba  2

System out println bb
```

答：

语法规则：在进行位运算的时候，除long外都会提升成int， 计算完成后变成了int，赋值给一个byte，精度降低， 有可能不再是原来的值了，因为byte的范围是-127~128.

```
byte ba 127

int bb ba  2   // 修改为int接收即可
```

```
System out println bb
```

1.3.1.3. 第02题

给出 **b** 和 **c** 的结果，并且用位移方式图示解释

```
int a  1024
int b   a  1
int c   a   1
```

答案：

```
1111 1111 1111 1111 1111 1100 0000 0000   -1024的补码
-------------------------------------      1
1111 1111 1111 1111 1111 1110 0000 0000   -512

1111 1111 1111 1111 1111 1100 0000 0000   -1024的补码
-------------------------------------      1
0111 1111 1111 1111 1111 1110 0000 0000   2147483136
```

下面用程序来验证

```
public void _02
    int a    1024
    int b   a    1
    int c   a     1
    // 11111111111111111111110000000000
    System out println Integer toBinaryString a
    System out println Integer toBinaryString b
    System out println Integer toBinaryString c
    System out println b
    System out println c


                输出如下

11111111111111111111110000000000
11111111111111111111111000000000
1111111111111111111111000000000    // 注意这里少了一位
 512
2147483136
```

> 本题知识点：
>
> 1. 有符号和无符号位移的区别：无符号始终用0补高位，有符号则补符号位

> 2. 负数在java中以补码形式呈现

1.3.1.4. 第03题

题目：

定义一个10240*10240的byte数组，分别采用行优先与列优先的循环方式来计算 这些单元格的总和，看看性能的差距，并解释原因 行优先的做法，每次遍历一行，然后到下一行。

这里要明白行优先和列优先,看下面的列子就明白了

```
public void _03
    int     arr   new int 5  5
    for  int i   0  i   arr length  i
        int length   arr i  length
        for  int j   0  j   length  j
            arr i  j    j


    System out println "------------ 行优先 -------------"
    for  int i   0  i   arr length  i
        int length   arr i  length
        for  int j   0  j   length  j
            if  j   1    length
                System out print arr i  j
              else
                System out print arr i  j    " , "


        System out println ""


    System out println "------------ 列优先 -------------"
    for  int i   0  i   arr length  i
        int length   arr i  length
        for  int j   0  j   length  j
            if  j   1    length
                System out print arr j  i
              else
                System out print arr j  i    " , "


        System out println ""
```

这是输出

```
------------ 行优先 --------------
0 , 1 , 2 , 3 , 4
0 , 1 , 2 , 3 , 4
0 , 1 , 2 , 3 , 4
0 , 1 , 2 , 3 , 4
0 , 1 , 2 , 3 , 4
------------ 列优先 --------------
0 , 0 , 0 , 0 , 0
1 , 1 , 1 , 1 , 1
2 , 2 , 2 , 2 , 2
3 , 3 , 3 , 3 , 3
4 , 4 , 4 , 4 , 4

可以看出：
行优先的读取顺序是：0,0、0,1、0,2
列优先的读取顺序是：0,0、1,0、2,0
```

答案：

```
    public void _03_01
        byte     arr   new byte 10240  10240
        for  int i   0  i   arr length  i
            int length   arr i  length
            for  int j   0  j   length  j
                arr i  j    1


        _03_01Row arr
        _03_01Column arr


    private void _03_01Row byte     arr
        Instant start   Instant now
        int total   0
        for  int i   0  i   arr length  i
            int length   arr i  length
            for  int j   0  j   length  j
                total    arr i  j


        Instant end   Instant now
        System out println "结果"   total
                  ";耗时："   Duration between start  end  toMillis


    private void _03_01Column byte     arr
        Instant start   Instant now
        int total   0
        for  int i   0  i   arr length  i
```

```
            int length   arr i  length
            for  int j   0  j   length  j
                total    arr j  i


        Instant end   Instant now
        System out println "结果"   total
                  ";耗时："   Duration between start  end  toMillis
```

这是输出

```
结果104857600 耗时：54
结果104857600 耗时：3163
```

由于数组的特性 缓存友好性，造成了此差异；在程序启动后就立即生成了数组中的值；

这里可能有一个误区，我以为是初始化的方式问题，结果按照列优先的方式去初始化，结果还是行优先快。然后就去百度了，得到的答案是：c和java的数组都是按 行优先的方式存储的，即一维存储完再继续下一维

1.3.1.5. 第 04 题目

题目

```
定义Java类Salary {String name, int baseSalary, int bonus  },随机产生1万个实例，
属性也随机产生（baseSalary范围是5-100万，bonus为（0-10万），其中name长度为5，随机字符串，
然后进行排序，排序方式为收入总和（baseSalary*13+bonus），输出收入最高的10个人的名单
```

答案：

```
public class Salary
    private String name
    private int baseSalary
    private int bonus


public class Practice_04
    private static Random random   new Random

    public static void main String   args
        Salary   salaries   mockData
        Arrays sort salaries   s1  s2
            int s1Total   s1 getBaseSalary     13   s1 getBonus
            int s2Total   s2 getBaseSalary     13   s2 getBonus
```

```
            if  s1Total   s2Total  return 1
            if  s1Total    s2Total  return 0
            return  1

        Salary   top10   Arrays copyOf salaries  10
        for  Salary salary   top10
            System out println salary getName     " : "    salary getBaseSalary     13
salary getBonus



    private static Salary   mockData
        Salary   salaries   new Salary 10000
        for  int i   0  i   salaries length  i
            Salary salary   new Salary
            salary setName buildName
            salary setBaseSalary build 50000  100_0000
            salary setBonus build 0  10_0000
            salaries i    salary

        return salaries


    private static int build int min  int max
        return random nextInt max   min   1    min



    private static String buildName
        // ASCII 码, A = 65;Z=90
        StringBuffer sb   new StringBuffer
        for  int i   0  i   5  i
            int c   random nextInt 90   65   1    65
            sb append  char  c

        return sb toString
```

程序输出

```
ZJBVN : 13098270
QODRR : 13088847
EOOJS : 13080197
EUQXO : 13076913
PTQOG : 13076635
MWAFJ : 13073621
BIGBK : 13072453
BQTUF : 13070849
```

```
SABNM : 13069242
BVOGJ : 13065665
```

1.3.1.6. 第 05 题

1.3.1.6.1. 题目

编码实现下面的要求

现有对象 MyItem {byte type,byte color,byte price} ，
要求将其内容存放在一个扁平的byte[]数组存储数据的ByteStore {byte[] storeByteArry}对象里，
即每个MyItem占用3个字节，第一个MyItem占用`storeByteArry[0]-storeByteArry[2]` 3个连续字节，
以此类推，最多能存放1000个MyItem对象

ByteStore提供如下方法
- putMyItem(int index,MyItem item)
  在指定的Index上存放MyItem的属性，这里的Index是0-999，而不是storeByteArry的Index
- getMyItem(int index)
  从指定的Index上查找MyItem的属性，并返回对应的MyItem对象。

要求放入3个MyItem对象（index为0-2）并比较getMyItem方法返回的这些对象是否与之前放入的对象equal。

加分功能如下：
放入1000个MyItem对象到ByteStore中，采用某种算法对storeByteArry做排序，排序以price为基准，
排序完成后，输出前100个结果

1.3.1.6.2. 答案

> 详细代码请右键新窗口查看GIT

以下是程序输出：

```
MyItem{type=1, color=-128, price=3}
MyItem{type=1, color=-128, price=3}
true
--------------------
MyItem{type=2, color=3, price=4}
MyItem{type=2, color=3, price=4}
true
--------------------
MyItem{type=5, color=-9, price=-4}
MyItem{type=5, color=-9, price=-4}
true
--------------------
```

加分功能输出：

```
MyItem{type=95, color=29, price=127}
MyItem{type=63, color=119, price=127}
MyItem{type=69, color=52, price=127}
MyItem{type=27, color=11, price=127}
MyItem{type=110, color=60, price=127}
...
```

1.3.1.7. 第 06 题

1.3.1.7.1. 题目

```
Arrays.parallelSort在数组超过多少时候才开启并行排序？采用位运算，给出推导过程
```

1.3.1.7.2. 答案

从源码上可以看出来，数量大于 MIN_ARRAY_SORT_GRAN（8192）时，开启并行排序

```
    private static final int MIN_ARRAY_SORT_GRAN   1    13
    public static void parallelSort int   a
        int n   a length  p  g
        if  n    MIN_ARRAY_SORT_GRAN
             p   ForkJoinPool getCommonPoolParallelism       1
            DualPivotQuicksort sort a  0  n   1  null  0  0
        else
            new ArraysParallelSortHelpers FJInt Sorter
                 null  a  new int n   0  n  0
                   g   n    p    2      MIN_ARRAY_SORT_GRAN
                 MIN_ARRAY_SORT_GRAN   g  invoke
```

1.3.1.8. 第 07 题

> 习题答案测试入口请右键新窗口查看GIT

1.3.1.8.1. 题目

```
DualPivotQuicksort算法与普通冒泡算法相比，有哪些改进?
对比常见的几种基于数组的排序算法，说说为什么Java选择了快排?

以下是加分题目
 第一：
```

```
写出标准冒泡排序与快速排序的算法，排序对象为上面说的 Salary {name, baseSalary, bonus},
收入总和为baseSalary*13+bonus,以收入总和为排序标准。
排序输出 年薪最高的100个人，输出结果为 xxxx:yyyy万

第三：
第五题中的 storeByteArry改为int[]数组，采用Java位操作方式来实现1个Int 拆解为4个Byte，存放My Item对象的属性。
```

1.3.1.8.2. 主题答案

主题答案不会；

1.3.1.8.3. 加分题 01 答案

快速排序的思路:

```
* 1. 从低位找一个基准值
* 2. 从高位开始降低对比找到比基准值还小的数。然后把这个数放到当前的低位
* 3. 从低位开始增高对比找到比基准值还大的数。然后把这个数放到当前的高位
* 4. 把当前的低位放入基准值。 这个就是一个分界点，左边的小于基准值。右边的大于基准值
* 5. 把左右两拨分开递归重复上述1，2，3，4步骤
```

程序输出

```
排序前：[6, 8, 1, 3, 2, 6, 4]
------------------- 基准：6
找数之前：[6, 8, 1, 3, 2, 6, 4]
找到小数：[4, 8, 1, 3, 2, 6, 4]
找到大数：[4, 8, 1, 3, 2, 6, 8]
------------------- 基准：6
找数之前：[4, 8, 1, 3, 2, 6, 8]
找到小数：[4, 2, 1, 3, 2, 6, 8]
找到大数：[4, 2, 1, 3, 2, 6, 8]
填回枢轴：[4, 2, 1, 3, 6, 6, 8]
------------------- 基准：4
找数之前：[4, 2, 1, 3, 6, 6, 8]
找到小数：[3, 2, 1, 3, 6, 6, 8]
找到大数：[3, 2, 1, 3, 6, 6, 8]
填回枢轴：[3, 2, 1, 4, 6, 6, 8]
------------------- 基准：3
找数之前：[3, 2, 1, 4, 6, 6, 8]
找到小数：[1, 2, 1, 4, 6, 6, 8]
找到大数：[1, 2, 1, 4, 6, 6, 8]
填回枢轴：[1, 2, 3, 4, 6, 6, 8]
------------------- 基准：1
找数之前：[1, 2, 3, 4, 6, 6, 8]
找到小数：[1, 2, 3, 4, 6, 6, 8]
找到大数：[1, 2, 3, 4, 6, 6, 8]
```

```
填回枢轴：[1, 2, 3, 4, 6, 6, 8]
------------------- 基准：6
找数之前：[1, 2, 3, 4, 6, 6, 8]
找到小数：[1, 2, 3, 4, 6, 6, 8]
找到大数：[1, 2, 3, 4, 6, 6, 8]
填回枢轴：[1, 2, 3, 4, 6, 6, 8]
排序后：[1, 2, 3, 4, 6, 6, 8]
```

从这里看，和结合代码看，在每一次找分界点的时候，他每次定位一个基准值，开始处理的时候，每一轮其实都只是设置了一个小值和一个大值， 类似于交换位置 ；基准值拿出去了，那么就相当于第三方变量。每一轮都要设置最低值和最高值；

最后当高位和低位相等的时候就找到了中间值索引，把基准值填会这个中间值。就按这个基准值把左右两边的数分开了；

冒泡排序:相邻的两个元素对比，大/小的 互相交换位置，一轮下来，最大/小的元素一定会再一端的开始

1.3.1.8.4. 加分题 02 答案

> 代码在题目处进入git查看

本题思路和知识点：

```
用int来存储三个byte，示意图如下
00000000 00000000 00000000 00000000 一个int4个8位，32bit
          type    color   price
使用位移方式把byte移动到指定位置

获取元素则相反，需要使用到 与操作和位移操作，逐个拿到需要的位置上的值

1. 需要通过与操作将其他不需要的位置重置为0：
    e & 0x00FF0000 ；  使用 与 的操作特性，1 & 1 才得1，
    FF 二进制表现形式是8个1;
2. 然后通过位移操作把值移动到最（低位）右端
    无符号右移始终补0
必须注意位运算中的符号优先级问题
```

 updated 2018-03-04 11:16:13

1.4. JAVA内存模型

1.4.1. volatile

- 特性：

  1. 可见性：对一个volatile变量的读，总是能看到(任意线程)对这个volatile变量最后的写入
  2. 原子性：对任意单个volatile变量的读/写具有原子性，但类似与volatile++这种复合操作不具有原子性

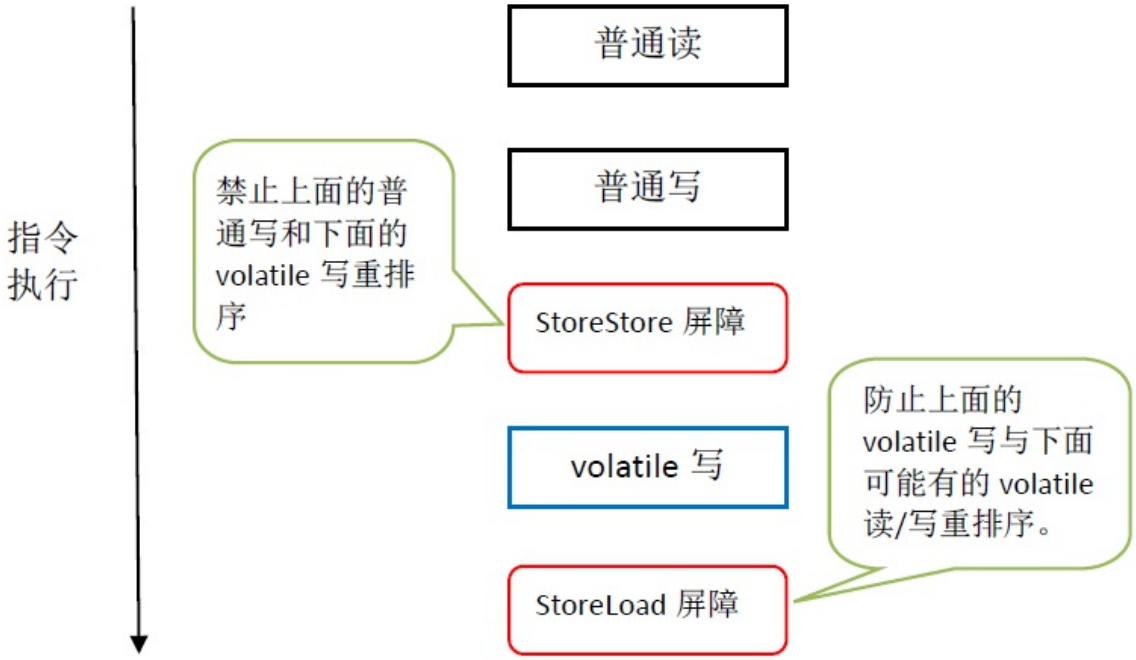
- 与读/写建立的happens before关系

- 写/读的内存语义

  当写一个volatile变量时，该操作之前的操作都将会刷新到主内存中去，包括自身

  当读一个volatile变量时，JMM把本地内存置为无效，直接从主内存中读取

- 内存语义的实现 在每个volatile写操作的前面插入一个StoteStore屏障 在每个volatile写操作的后面插入一个StoreLoad屏障 在每个volatile读操作的后面插入一个LoadLoad屏障 在每个volatile读操作的后面插入一个LoadStore屏障



这里也能看出为什么会在一个volatile前后都插入屏障了，一个防止与前面发生重排序，一个防止与后面发生重排序

 updated 2017-12-22 13:46:55

1.4.1. 基础

1.4.1.1. 并发编程模型的分类

在并发编程中，需要处理两个问题：

1. 线程之间如何通信
2. 线程直接如何同步

线程之间的通信机制有两种：

1. 共享内存：有公共状态
2. 消息传递：无公共状态，必须通过明确的发送消息进行通信

Java的并发采用的是共享内存模型；

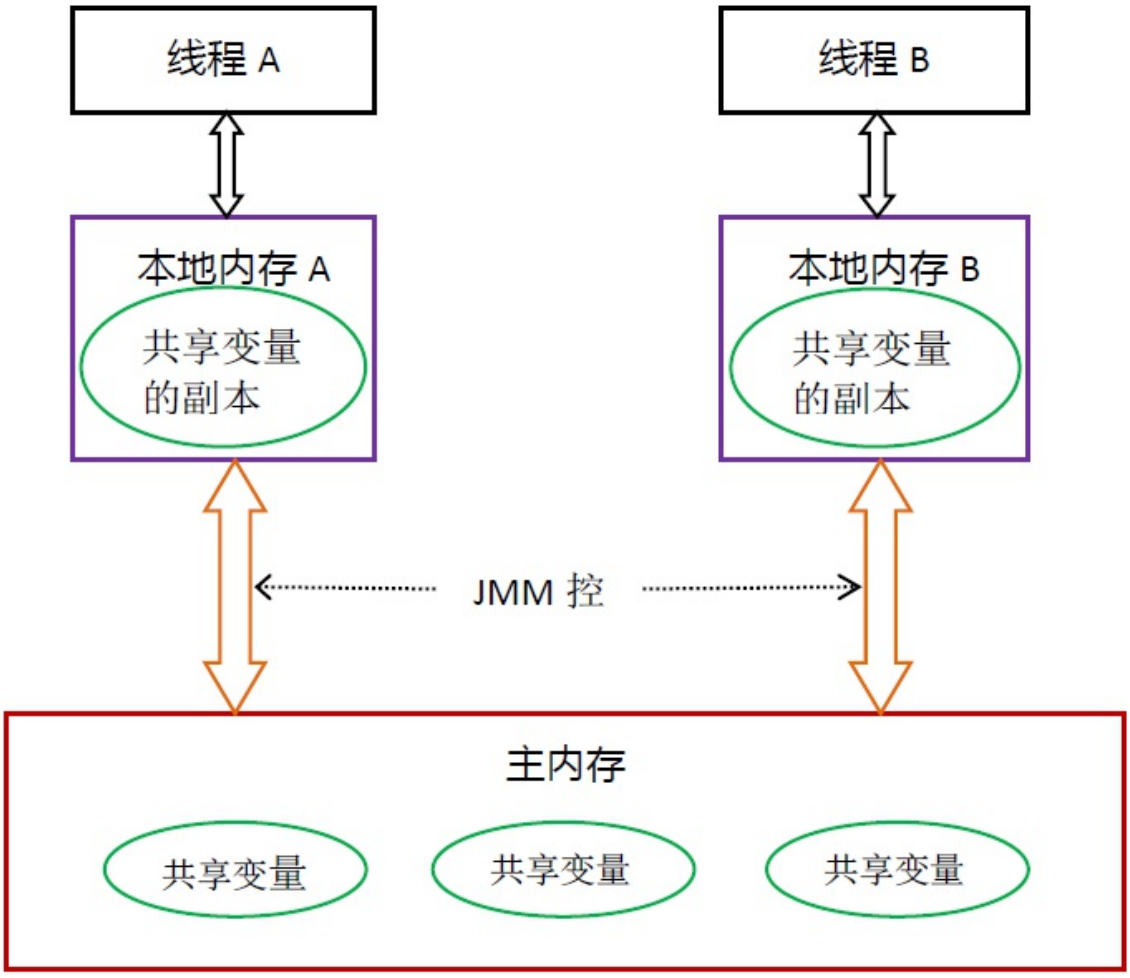
1.4.1.2. JAVA内存模型的抽象

堆内存：所有实例域、静态域、和数组元素存储在堆内存中，堆内存在线程直接共享（可称为共享变量）

局部变量（Local variables），方法定义参数，异常处理器参数不会在线程之间共享，他们不会有内存可见性的问题，也不受内存模型的影响。

Java内存模型（JMM）定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存中（main memory），每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，不真实存在。包含了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。示意图如下

Java内存模型（JMM）定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存中（main memory），每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，不真实存在。包含了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。示意图如下



1.4.1.3. volatile

 updated 2017-12-22 13:46:55

 updated 2018-01-29 11:39:41

2.1.1. 原码、反码、补码 - 虚拟机

本文章的知识是在葛一鸣的《实战JAVA虚拟机》一书看到的知识点，貌似也终于解惑了我在应用上的一些问题

整数在计算机中用补码表示，Java的在虚拟机中也一样；

2.1.1.1. 什么是源码？

原码 = 符号位 + 数字的二进制

比如：

```
10  的源码：0000 0000 0000 0000 0000 0000 0000 1010
-10 的源码：1000 0000 0000 0000 0000 0000 0000 1010
```

对于源码来说，绝对值相同的正负数只有符号位不同；

2.1.1.2. 什么是反码 ？

反码 = 原码的基础上，符号位不变，其余位取反，（正数的反码是本身）

```
10  的反码：0000 0000 0000 0000 0000 0000 0000 1010
-10 的反码：1111 1111 1111 1111 1111 1111 1111 0101
```

2.1.1.3. 什么是补码？

补码 = 反码 + 1,(正数的反码是本身)

```
10  的补码：0000 0000 0000 0000 0000 0000 0000 1010
-10 的补码：1111 1111 1111 1111 1111 1111 1111 0110
```

在JAVA中不使用API来查看负数在虚拟机中的表现形式如下：

```
System out println Integer toHexString a
System out println Integer toBinaryString a
```

```
输出如下：可以看出在JAVA中负数使用补码表示的
fffffff6
11111111111111111111111111110110
```

那么不使用API这么高大上的方式，要怎么获取到反码表现形式呢？

```
int a    10
for  int i   0  i   32  i
    // 0x8000 0000 是一个符号位为1，其余都为0的数
    int t    a    0x80000000     i        31   i
    System out println t
```

这段程序的基本思路是：循环32次，依次从左到右取到每一位数字 取对应位置数值原理：& 上一个你想要的位置上为1，其他位置上全部为0的数，然后右移动到最低位 这段程序的实现思路是：

1. 先用 `0x80000000 >>> i` 得到当前要获取的位置
2. 然后 a & 第一步的位
3. `>>> (31 - i)` 把对应的位置移动到最后一位

比如程序中：

```
第一次：i = 0
0x80000000 >>> i :
1000 0000 0000 0000 0000 0000 0000 0000
------------------------------------ 无符号右移动0位，还是原来的数
1111 1111 1111 1111 1111 1111 1111 0110
------------------------------------ 再&上-10，这里要特别注意，在计算机中负数用反码表示
1000 0000 0000 0000 0000 0000 0000 0000
------------------------------------ 最后>>> (31 - i)=31
0000 0000 0000 0000 0000 0000 0000 0001 变成了1

第三次:i = 1
0x80000000 >>> i :
1000 0000 0000 0000 0000 0000 0000 0000
------------------------------------ 无符号右移动1位
0100 0000 0000 0000 0000 0000 0000 0000
------------------------------------ 再&上-10
1111 1111 1111 1111 1111 1111 1111 0110
------------------------------------
0100 0000 0000 0000 0000 0000 0000 0000
------------------------------------最后>>> (31 - i)=30
0000 0000 0000 0000 0000 0000 0000 0001 拿到了正数 第2位，依次类推
```

2.1.1.4. 使用补码能解决什么？

1. 能解决 0 的存储问题；如下：

```
0  的补码：  0000 0000 0000 0000 0000 0000 0000 0000

-0 的原码：  1000 0000 0000 0000 0000 0000 0000 0000
-0 的反码：  1111 1111 1111 1111 1111 1111 1111 1111
------------------------------------------------- 反码 + 1
-0 的补码：  0000 0000 0000 0000 0000 0000 0000 0000
```

2. 使用补码可以简化整数的加减法计算，将减法视为加法计算

   比如：计算-6 + 5 的过程如下

```
-6的补码：1111 1010
5 的补码：0000 0101
------------------ 相加
       ：1111 1111  这是补码，要换算成原码
------------------ 补码-1 等于反码
       : 1111 1110
------------------ 反码取反等于原码
       : 1000 0001
```

 updated 2018-01-21 17:52:54