链滴

jdk 源码：Integer.getChars(int i, int index , char[] buf)

作者：Maggie

原文链接：https://ld246.com/article/1538041031423

来源网站：链滴

1. 应用：将整形数字转换成对应的十进制字符串

```
public static String toString(int i) {
    if (i == Integer.MIN_VALUE)
        return "-2147483648";
    int size = (i < 0) ? stringSize(-i) + 1 : stringSize(i);
    char[] buf = new char[size];
    getChars(i, size, buf);
    return new String(buf, true);
}
```

2. 测试代码

```
public static void main(String[] args) {
    System.out.println(Integer.toString(-2147483647));
    System.out.println(Integer.toString(2147483647));
    System.out.println(Integer.toString(66580));
}
```

执行结果：

```
-2147483647
2147483647
66580
```

3. 源码分析： getChars(int i, int index, char[] buf)

- 源码

```
/**
 * Places characters representing the integer i into the
 * character array buf. The characters are placed into
 * the buffer backwards starting with the least significant
 * digit at the specified index (exclusive), and working
 * backwards from there.
 *
 * Will fail if i == Integer.MIN_VALUE
 */
static void getChars(int i, int index, char[] buf) {
    // 代码段1
    int q, r;
    int charPos = index;
    char sign = 0;

    if (i < 0) {
        sign = '-';
        i = -i;
    }
    // 代码段2
    // Generate two digits per iteration
    while (i >= 65536) {
        q = i / 100;
```

```
        // really: r = i - (q * 100);
            r = i - ((q << 6) + (q << 5) + (q << 2));
            i = q;
            buf [--charPos] = DigitOnes[r];
            buf [--charPos] = DigitTens[r];
        }

        // 代码段3
        // Fall thru to fast mode for smaller numbers
        // assert(i <= 65536, i);
        for (;;) {
            q = (i * 52429) >>> (16+3);
            r = i - ((q << 3) + (q << 1));  // r = i-(q*10) ...
            buf [--charPos] = digits [r];
            i = q;
            if (i == 0) break;
        }
        if (sign != 0) {
            buf [--charPos] = sign;
        }
    }
```

1. 此方法作用将int数字转换放进一个字符数组

2. 在数字为int最小值时，此方法会失败

以上是此方法注释的内容，源码主要是讲数字i转换为字符串，方法有三段组成。

3. 代码段1主要判断i的正负，并对负数取反，方便处理。 **这里也解释了为什么当i为最小数字时方失败，因为int范围是-2147483648~2147483647，当其取反时超过了int的范围。**

4. 代码段2是处理i>=65536时，每次取数字i的最后两位转为字符

```
while (i >= 65536) {
        q = i / 100;
        // really: r = i - (q * 100);
        r = i - ((q << 6) + (q << 5) + (q << 2));
        i = q;
        //取DigitOnes[r]的目的其实取数字r%10的结果
        buf [--charPos] = DigitOnes[r];
        //取DigitTens[r]的目的其实是取数字r/10的结果
        buf [--charPos] = DigitTens[r];
    }
```

这个代码很简单，我们看懂两个地方就行了。

- a. r = i - ((q << 6) + (q << 5) + (q << 2)); 实际上去 i - q*100，得到r是i的最后两位
- b. buf [--charPos] = DigitOnes[r]; buf [--charPos] = DigitTens[r];巧妙的运用两个数组查找，避除法等计算。

5. 代码段3是处理i<65536时每次取一位转为字符

```
for (;;) {
        //这里其实就是除以10。取数52429和16+3的原因在后文分析。
        q = (i * 52429) >>> (16+3);
        r = i - ((q << 3) + (q << 1));  // r = i-(q*10) ...
```

```
        //将数字i的最后一位存入字符数组，
        buf [--charPos] = digits [r];
        i = q;
        //for循环结束后，buf内容为“12345678”；
        if (i == 0) break;
    }
```

这个代码我们需要知道

- a. q = (i * 52429) >>> (16+3) 约等于 i/10
- b. r = i - ((q << 3) + (q << 1))，其实是r = i - q*10, 所以r其实是i的最后一位，然后放进字符数组

但是这里我们肯定会有一些疑惑：

问题1：为什么是65536，为什么要在大于等于65536时和小于65536时采用不同的处理方法。

问题2：代码段3中 q = (i * 52429) >>> (16+3);中52439,16+3是怎么确定出来的

num1 = 65536, num2 = 52429, num3 = 19

在了解这num1,num2,num3为什么是这三个数字之前，我们有几个需要了解的前提：

1.移位效率高于乘除

2.乘法效率高于除法

这时我们注意到，代码段2里采用了除法和移位，代码段3里采用了乘法和移位。理论上我们如果都是代码段3进行处理效率会更高，但是注意到代码段3有这样一个操作是需要我们学习的。

```
q = (i * 52429) >>> (16+3);  // 约等于i/10，这里巧妙的运用了乘法和移位避免使用除法来提高效
。
```

这里巧妙的运用了乘法和移位避免使用除法来提高效率，但是同时也要求(i * 52429)不能超过整形的围。有符号整形是-2147483648至2147483647，无符号是0-4,294,967,296（2^32），由于后面的算是采用无符号右移，所以我们只需要使得(i * 52429)不超过2^32就可以了。所以说这里的i理论上不能超过2^32除以52429，这就解释了为什么要分i>=num1和i<num1来处理。

好，这样也就解释了num1,num2,num3这三个数的来历。那怎么确定这三个数的值，我们需要进行些计算。由于q = (i * 52429) >>> (16+3); // 约等于i/10，所以我们得出num2/2^num3=0.1 => um2=2^num3/10

最终num2=2^num3/10+1，**最后的+1应该是未了再计算的时候避免被向下取整了吧**。这里我一个测试代码证明了这个猜想

```
public static void main(String[] args) {
        System.out.println((1020 * 52428) >>> (16+3));  // 理论上精确的除以10，在计算机计算过
中可能会产生向下取整导致结果不准确的问题
        System.out.println((1020 * 52429) >>> (16+3));  // 改进版
    }
```

运行结果

```
101
102
```

回到主题，根据以上关系我们可以得出多组num2,num3

2^10=1024, 103/1024=0.1005859375

2^11=2048, 205/2048=0.10009765625
2^12=4096, 410/4096=0.10009765625
2^13=8192, 820/8192=0.10009765625
2^14=16384, 1639/16384=0.10003662109375
2^15=32768, 3277/32768=0.100006103515625
2^16=65536, 6554/65536=0.100006103515625
2^17=131072, 13108/131072=0.100006103515625
2^18=262144, 26215/262144=0.10000228881835938
2^19=524288, 52429/524288=0.10000038146972656
2^20=1048576, 104858/1048576=0.1000003815
2^21=2097152, 209716/2097152 = 0.1000003815
2^22= 4194304, 419431/4194304= 0.1000001431
2^23= 8388608, 838861/8388608= 0.1000000238
...

接下来其实有一个临界值((num1-1)*num2) >>> num3,需要保证((num1-1)*num2) 不能超过2^32 所以就是num1越大，num2就越小；num2越大，num1就越小。但同时这里我们有两个诉求：

1. 由于上面是约等于i/10,所以我们得对精度有要求。精度必须保证，可以得出num2和num3越大，度就越高

2. 因为乘法小于高于除法，我们偏向于num1越大越好，这样尽量多使用代码段3

从上面两个诉求来看，我们的num1,num2,num3都是越大越好。但是((num1-1)*num2) 不能超过2^2也告诉了我们num1和num2一个大了另一个就得小。所以采取折中的方式，精度足够了就好，乘除效率上我们也妥协一些。

我们发现，当num3=19时，精度达到了0.10000038146972656，0.1后面是5个0，我们称精度达到5，这个精度可以说符合要求了已经。并且当num3=20,21,22时精度都保持在5，此时提升num3和nm2对精度并没有提升。所以num3我们选择了19, num2也顺理成章为52429， 在2^15 和 2^16之。同时((num1-1)*num2) 不能超过2^32，所以num1选择了2^16。

这里笔者要吐槽一下网上对于这三个值的一些解释，自己都不能说服自己

1. 下面这种说法真真看不懂，可能是我没理解到位，下一个精度较高的组合是419431和22。当num为19,20,21,22时的精度不是一样高的吗，为什么下一个是22。 另外** “2^31/2^22 = 2^9 = 512。12这个数字实在是太小了” **实在是看不懂啊。

52429/524288=0.10000038146972656精度足够高
下一个精度较高的m和n的组合是419431和22。2^31/2^22 = 2^9 = 512。512这个数字实在是太了。65536正好是2^16，一个整数占4个字节。65536正好占了2个字节，选定这样一个数字有利于CU访问数据。

2. 下面这种说法更是可笑，你说不超出整形范围内值得是((num1-1)*num2)不超过吧。你这么说应是已经认定了num1=65536，可是num1,num2,num3这三个值的确定本就是互相影响，互相确定的 你这么说是在num1=65536的大前提下的，可是num1的值时怎么确定的呢，对吧?

至于为什么选择2的19次方524288，是因为52429/524288得到的数字精度在不超出整形范围内，精是最高的。

4. 对两个数组的解释

100以下的数，DigitTens十位数上的数字，DigitOnes为个位数的数字，如86 = DigitTens[86]+ DigiOnes[86] = '8' + '6' = 86

比较巧妙的设计

```
//100以内的数字除以10的结果（取整）
final static char [] DigitTens = {
    '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
    '1', '1', '1', '1', '1', '1', '1', '1', '1', '1',
    '2', '2', '2', '2', '2', '2', '2', '2', '2', '2',
    '3', '3', '3', '3', '3', '3', '3', '3', '3', '3',
    '4', '4', '4', '4', '4', '4', '4', '4', '4', '4',
    '5', '5', '5', '5', '5', '5', '5', '5', '5', '5',
    '6', '6', '6', '6', '6', '6', '6', '6', '6', '6',
    '7', '7', '7', '7', '7', '7', '7', '7', '7', '7',
    '8', '8', '8', '8', '8', '8', '8', '8', '8', '8',
    '9', '9', '9', '9', '9', '9', '9', '9', '9', '9',
    } ;

//100以内的数字对10取模的结果
final static char [] DigitOnes = {
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    } ;
```

5. 从这个方法我们学到的

- 乘法比除法高效：q = (i * 52429) >>> (16+3); => 约等于q0.1,但i52429是整数乘法器，结合位避免除法
- 充分利用计算结果:在获取r(i%100)时，充分利用了除法的结果，结合位移避免重复计算
- 位移比乘法高效:r = i – ((q << 6) + (q << 5) + (q << 2)); = >等价于r = i – (q * 100)
- 局部性原理之空间局部性

```
(1).buf[–charPos] =DigitOnes[r];buf[–charPos] =DigitTens[r];通过查找数组，实现快速访问,避免法计算
(2).buf [–charPos ] = digits [ r];
```

作者 @没有故事的老大爷

奋斗，但是该不该知足，该不该休息呢?

1