链滴

JDK 源码中的一些“小技巧”

作者：vvsuperman
原文链接：https://ld246.com/article/1521890633733
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

均摘选自JDK源码，都收录在《Java基础教程-手写JDK》中
大家不妨围观下：）

1 i++ vs i--

String源码的第985行，equals方法中

```
while (n--!= 0) {
    if (v1[i] != v2[i])
        return false;
    i++;
}
```

这段代码是用于判断字符串是否相等，但有个奇怪地方是用了i--!=0来做判断，我们通常不是用i++？为什么用i--呢？而且循环次数相同。 原因在于编译后会多一条指令：

i-- 操作本身会影响CPSR(当前程序状态寄存器)，CPSR常见的标志有N(结果为负), Z(结果为0)，C（进位），O（有溢出）。i > 0，可以直接通过Z标志判断出来。

i++操作也会影响CPSR(当前程序状态寄存器)，但只影响O（有溢出）标志，这对于i < n的判断没有何帮助。所以还需要一条额外的比较指令，也就是说每个循环要多执行一条指令。

简单来说，跟0比较会少一条指令。所以，循环使用i--，**高端大气上档次。**

2 成员变量 vs 局部变量

JDK源码在任何方法中几乎都会用一个局部变量来接受成员变量，比如

```
public int compareTo(String anotherString) {
  int len1 = value.length;
  int len2 = anotherString.value.length;
```

因为局部变量初始化后是在该方法线程栈中，而成员变量初始化是在堆内存中，显然前者更快，所以我们在方法中尽量避免直接使用成员变量，而是使用局部变量。

3 刻意加载到寄存器 && 将耗时操作放到锁外部

在ConcurrentHashMap中，锁segment的操作很有意思，它不是直接锁，而是类似于自旋锁，反复试获取锁，并且在获取锁的过程中，会遍历链表，从而将数据先加载到寄存器中缓存中，避免在锁的程中在便利，同时，生成新对象的操作也是放到锁的外部来做，避免在锁中的耗时操作

```
  final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    /** 在往该 segment 写入前，需要先获取该 segment 的独占锁
      不是强制lock()，而是进行尝试 */
    HashEntry<K,V> node = tryLock() ? null :
      scanAndLockForPut(key, hash, value);
```

scanAndLockForPut()源码

```
private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {
```

```
    HashEntry<K,V> first = entryForHash(this, hash);
    HashEntry<K,V> e = first;
    HashEntry<K,V> node = null;
    int retries = -1; // negative while locating node

    // 循环获取锁
    while (!tryLock()) {
        HashEntry<K,V> f; // to recheck first below
        if (retries < 0) {
            if (e == null) {
                if (node == null) // speculatively create node
                    //该hash位无值，新建对象，而不用再到put()方法的锁中再新建
                    node = new HashEntry<K,V>(hash, key, value, null);
                retries = 0;
            }
            //该hash位置key也相同，退化成自旋锁
            else if (key.equals(e.key))
                retries = 0;
            else
                // 循环链表，cpu能自动将链表读入缓存
                e = e.next;
        }
        // retries>0时就变成自旋锁。当然，如果重试次数如果超过 MAX_SCAN_RETRIES（单核1多核
4），那么不抢了，进入到阻塞队列等待锁
        //    lock() 是阻塞方法，直到获取锁后返回，否则挂起
        else if (++retries > MAX_SCAN_RETRIES) {
            lock();
            break;
        }
        else if ((retries & 1) == 0 &&
                 // 这个时候是有大问题了，那就是有新的元素进到了链表，成为了新的表头
                 //     所以这边的策略是，相当于重新走一遍这个 scanAndLockForPut 方法
                 (f = entryForHash(this, hash)) != first) {
            e = first = f; // re-traverse if entry changed
            retries = -1;
        }
    }
    return node;
}
```

4 判断对象相等可先用==

在判断对象是否相等时，可先用==，因为==直接比较地址，非常快，而equals的话会最对象值的比，相对较慢，所以有可能的话，可以用a==b || a.equals(b)来比较对象是否相等

5 关于transient

transient是用来阻止序列化的，但HashMap源码中内部数组是定义为transient的

```
/**
    * The table, resized as necessary. Length MUST Always be a power of two.
    */
```

```
transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY_TABLE;
```

那岂不里面的键值对都无法序列化了么，网络中用hashmap来传输岂不是无法传输，其实不然。

Effective Java 2nd, Item75, Joshua大神提到:

For example, consider the case of a hash table. The physical representation is a sequence of hash buckets containing key-value entries. The bucket that an entry resides in is a function of the hash code of its key, which is not, in general, guaranteed to be the same from JVM implementation to JVM implementation. In fact, it isn't even guaranteed to be the same from run to run. Therefore, accepting the default serialized form for a hash table would constitute a serious bug. Serializing and deserializing the hash table could yield an object whose invariants were seriously corrupt.

怎么理解? 看一下HashMap.get()/put()知道, 读写Map是根据Object.hashcode()来确定从哪个bucke读/写. 而Object.hashcode()是native方法, 不同的JVM里可能是不一样的.

打个比方说, 向HashMap存一个entry, key为 字符串"STRING", 在第一个java程序里, "STRING"的has code()为1, 存入第1号bucket; 在第二个java程序里, "STRING"的hashcode()有可能就是2, 存入第2号ucket. 如果用默认的串行化(Entry[] table不用transient), 那么这个HashMap从第一个java程序里通串行化导入第二个java程序后, 其内存分布是一样的, 这就不对了.

举个例子，比如向HashMap存一个键值对entry, key="方老司", 在第一个java程序里, "方老司"的has code()为1, 存入table1，好，现在传到另一个在JVM程序里, "方老司" 的hashcode()有可能就是2, 于到table[2]去取，结果值不存在。

HashMap现在的readObject和writeObject是把内容 输出/输入, 把HashMap重新生成出来.

6 不要用char

char在Java中utf-16编码，是2个字节，而2个字节是无法表示全部字符的。2个字节表示的称为 BMP 另外的作为high surrogate和 low surrogate 拼接组成由4字节表示的字符。比如String源码中的inde Of:

```
//这里用int来接受一个char，方便判断范围
public int indexOf(int ch, int fromIndex) {
    final int max = value.length;
    if (fromIndex < 0) {
        fromIndex = 0;
    } else if (fromIndex >= max) {
        // Note: fromIndex might be near -1>>>1.
        return -1;
    }
    //在Bmp范围
    if (ch < Character.MIN_SUPPLEMENTARY_CODE_POINT) {
        // handle most cases here (ch is a BMP code point or a
        // negative value (invalid code point))
```

```
        final char[] value = this.value;
        for (int i = fromIndex; i < max; i++) {
            if (value[i] == ch) {
                return i;
            }
        }
        return -1;
    } else {
        //否则转到四个字节的判断方式
        return indexOfSupplementary(ch, fromIndex);
    }
}
```

所以Java的char只能表示utf-16中的bmp部分字符。对于CJK（中日韩统一表意文字）部分扩展字符则无法表示。

例如，下图中除Ext-A部分，char均无法表示。

此外还有一种说法是要用char，密码别用String，String是常量（即创建之后就无法更改），会保存常量池中，如果有其他进程可以dump这个进程的内存，那么密码就会随着常量池被dump出去从而露，而char[]可以写入其他的信息从而改变，即是被dump了也会减少泄露密码的风险。

但个人认为你都能dump内存了难道是一个char能够防范的住的？除非是String在常量池中未被回收而被其它线程直接从常量池中读取，但恐怕也是非常罕见的吧。

1