目录

第1章 基础

01 开篇词：为什么学习本专栏

02 String、Long 源码解析和面 最近阅读

03 Java 常用关键字理解

04 Arrays、Collections、Objects 常用方法源码解析

第2章 集合

05 ArrayList 源码解析和设计思路

06 LinkedList 源码解析

07 List 源码会问哪些面试题

08 HashMap 源码解析

09 TreeMap 和 LinkedHashMap 核心源码解析

10 Map源码会问哪些面试题

11 HashSet、TreeSet 源码解析

12 彰显细节：看集合源码对我们实际工作的帮助和应用

13 差异对比：集合在 Java 7 和 8 有何不同和改进

14 简化工作：Guava Lists Maps 实际工作运用和源码

第3章 并发集合类

15 CopyOnWriteArrayList 源码解析和设计思路

16 ConcurrentHashMap 源码解析和设计思路

17 并发 List、Map源码面试题

18 场景集合：并发 List、Map的应用

02 String、Long 源码解析和面试题

更新时间：2019-11-26 09:44:44

> 劳动是一切知识的源泉。
>
> ——陶铸

引导语

String 和 Long 大家都很熟悉，本小节主要结合实际的工作场景，来一起看下 String 和 Long 的底层源码实现，看看平时我们使用时，有无需要注意的点，总结一下这些 API 都适用于哪些场景。

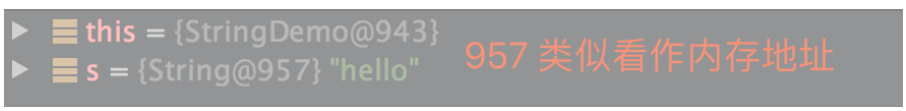
1 String

1.1 不变性

我们常常听人说，HashMap 的 key 建议使用不可变类，比如说 String 这种不可变类。这里说的不可变指的是类值一旦被初始化，就不能再被改变了，如果被修改，将会是新的类，我们写个 demo 来演示一下。

```
String s ="hello";
s ="world";
```

从代码上来看，s 的值好像被修改了，但从 debug 的日志来看，其实是 s 的内存地址已经被修改了，也就说 s =“world” 这个看似简单的赋值，其实已经把 s 的引用指向了新的 String，debug 的截图显示内存地址已经被修改，两张截图如下：



目录

我们从源码上查看一下原因：

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
}
```

我们可以看出来两点：

1. String 被 final 修饰，说明 String 类绝不可能被继承了，也就是说任何对 String 的操作方法，都不会被继承覆写；
2. String 中保存数据的是一个 char 的数组 value。我们发现 value 也是被 final 修饰的，也就是说 value 一旦被赋值，内存地址是绝对无法修改的，而且 value 的权限是 private 的，外部绝对访问不到，String 也没有开放出可以对 value 进行赋值的方法，所以说 value 一旦产生，内存地址就根本无法被修改。

以上两点就是 String 不变性的原因，充分利用了 final 关键字的特性，如果你自定义类时，希望也是不可变的，也可以模仿 String 的这两点操作。

因为 String 具有不变性，所以 String 的大多数操作方法，都会返回新的 String，如下面这种写法是不对的：

```
String str ="hello world !";
// 这种写法是替换不掉的，必须接受 replace 方法返回的参数才行，这样才行：str = str.replace("l", "dd");
str.replace("l", "dd");
```


1.2 字符串乱码

在生活中，我们经常碰到这样的场景，进行二进制转化操作时，本地测试的都没有问题，到其它环境机器上时，有时会出现字符串乱码的情况，这个主要是因为在二进制转化操作时，并没有强制规定文件编码，而不同的环境默认的文件编码不一致导致的。

我们也写了一个 demo 来模仿一下字符串乱码：

```
String str  ="nihao 你好 喬亂";
// 字符串转化成 byte 数组
byte[] bytes = str.getBytes("ISO-8859-1");
// byte 数组转化成字符串
String s2 = new String(bytes);
log.info(s2);
// 结果打印为：
nihao ?? ??
```

打印的结果为？？，这就是常见的乱码表现形式。这时候有同学说，是不是我把代码修改成 String s2 = new String(bytes,"ISO-8859-1"); 就可以了？这是不行的。主要是因为 ISO-8859-1 这种编码对中文的支持有限，导致中文会显示乱码。唯一的解决办法，就是在所有需要用到编码的地方，都统一使用 UTF-8，对于 String 来说，getBytes 和 new String 两个方法都会使用到编码，我们把这两处的编码替换成 UTF-8 后，打印出的结果就正常了。

1.3 首字母大小写

目录

景，在反射场景下面，我们也经常要使类属性的首字母小写，这时候我们一般都会这么做：

`name.substring(0, 1).toLowerCase() + name.substring(1);`，使用 substring 方法，该方法主要是为了截取字符串连续的一部分，substring 有两个方法：

1. `public String substring(int beginIndex, int endIndex)` beginIndex：开始位置，endIndex：结束位置；
2. `public String substring(int beginIndex)` beginIndex：开始位置，结束位置为文本末尾。

substring 方法的底层使用的是字符数组范围截取的方法 ： `Arrays.copyOfRange(字符数组, 开始位置, 结束位置);` 从字符数组中进行一段范围的拷贝。

相反的，如果要修改成首字母大写，只需要修改成 `name.substring(0, 1).toUpperCase() + name.substring(1)` 即可。

1.4 相等判断

我们判断相等有两种办法，equals 和 equalsIgnoreCase。后者判断相等时，会忽略大小写，近期看见一些面试题在问：如果让你写判断两个 String 相等的逻辑，应该如何写，我们来一起看下 equals 的源码，整理一下思路：

```
public boolean equals(Object anObject) {
    // 判断内存地址是否相同
    if (this == anObject) {
        return true;
    }
    // 待比较的对象是否是 String，如果不是 String，直接返回不相等
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        // 两个字符串的长度是否相等，不等则直接返回不相等
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            // 依次比较每个字符是否相等，若有一个不等，直接返回不相等
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```


从 equals 的源码可以看出，逻辑非常清晰，完全是根据 String 底层的结构来编写出相等的代码。这也提供了一种思路给我们：如果有人问如何判断两者是否相等时，我们可以从两者的底层结构出发，这样可以迅速想到一种贴合实际的思路和方法，就像 String 底层的数据结构是 char 的数组一样，判断相等时，就挨个比较 char 数组中的字符是否相等即可。

1.5 替换、删除

目录

其中在使用 replace 时需要注意，replace 有两个方法，一个入参是 char，一个入参是 String，前者表示替换所有字符，如：`name.replace('a','b')`，后者表示替换所有字符串，如：`name.replace("a","b")`，两者就是单引号和多引号的区别。

需要注意的是， replace 并不只是替换一个，是替换所有匹配到的字符或字符串哦。

写了一个 demo 演示一下三种场景：

```
public void testReplace(){
  String str ="hello word !!";
  log.info("替换之前 :{}",str);
  str = str.replace('l','d');
  log.info("替换所有字符 :{}",str);
  str = str.replaceAll("d","l");
  log.info("替换全部 :{}",str);
  str = str.replaceFirst("l","");
  log.info("替换第一个 l :{}",str);
}
//输出的结果是：
替换之前 :hello word !!
替换所有字符 :heddo word !!
替换全部 :hello worl !!
替换第一个 :helo worl !!
```

当然我们想要删除某些字符，也可以使用 replace 方法，把想删除的字符替换成“ ”即可。

1.6 拆分和合并

拆分我们使用 split 方法，该方法有两个入参数。第一个参数是我们拆分的标准字符，第二个参数是一个 int 值，叫 limit，来限制我们需要拆分成几个元素。如果 limit 比实际能拆分的个数小，按照 limit 的个数进行拆分，我们演示一个 demo：

```
String s ="boo:and:foo";
// 我们对 s 进行了各种拆分，演示的代码和结果是：
s.split(":") 结果:["boo","and","foo"]
s.split(":",2) 结果:["boo","and:foo"]
s.split(":",5) 结果:["boo","and","foo"]
s.split(":",-2) 结果:["boo","and","foo"]
s.split("o") 结果:["b","",":and:f"]
s.split("o",2) 结果:["b","o:and:foo"]
```

从演示的结果来看，limit 对拆分的结果，是具有限制作用的，还有就是拆分结果里面不会出现被拆分的字段。

那如果字符串里面有一些空值呢，拆分的结果如下：

```
String a =",a,,b,";
a.split(",") 结果:["","a","","b"]
```

从拆分结果中，我们可以看到，空值是拆分不掉的，仍然成为结果数组的一员，如果我们想删除空值，只能自己拿到结果后再做操作，但 Guava（Google 开源的技术工具）提供了一些可靠的工具类，可以帮助我们快速去掉空值，如下：

目录

```
List<String> list = Splitter.on(',')
    .trimResults()// 去掉空格
    .omitEmptyStrings()// 去掉空值
    .splitToList(a);
log.info("Guava 去掉空格的分割方法：{}",JSON.toJSONString(list));
// 打印出的结果为：
["a","b  c"]
```

从打印的结果中，可以看到去掉了空格和空值，这正是我们工作中常常期望的结果，所以推荐使用 Guava 的 API 对字符串进行分割。

合并我们使用 join 方法，此方法是静态的，我们可以直接使用。方法有两个入参，参数一是合并的分隔符，参数二是合并的数据源，数据源支持数组和 List，在使用的时候，我们发现有两个不太方便的地方：

1. 不支持依次 join 多个字符串，比如我们想依次 join 字符串 s 和 s1，如果你这么写的话 String.join(",",s).join(",",s1) 最后得到的是 s1 的值，第一次 join 的值被第二次 join 覆盖了；
2. 如果 join 的是一个 List，无法自动过滤掉 null 值。

而 Guava 正好提供了 API，解决上述问题，我们来演示一下：

```
// 依次 join 多个字符串，Joiner 是 Guava 提供的 API
Joiner joiner = Joiner.on(",").skipNulls();
String result = joiner.join("hello",null,"china");
log.info("依次 join 多个字符串:{}",result);

List<String> list = Lists.newArrayList(new String[]{"hello",null,"china"});
log.info("自动删除 list 中空值:{}",joiner.join(list));
// 输出的结果为；
依次 join 多个字符串:hello,china
自动删除 list 中空值:hello,china
```


从结果中，我们可以看到 Guava 不仅仅支持多个字符串的合并，还帮助我们去掉了 List 中的空值，这就是我们在工作中常常需要得到的结果。

2 Long

2.1 缓存

Long 最被我们关注的就是 Long 的缓存问题，Long 自己实现了一种缓存机制，缓存了从 -128 到 127 内的所有 Long 值，如果是这个范围内的 Long 值，就不会初始化，而是从缓存中拿，缓存初始化源码如下：

```
private static class LongCache {
    private LongCache(){}
    // 缓存，范围从 -128 到 127，+1 是因为有个 0
    static final Long cache[] = new Long[-(-128) + 127 + 1];

    // 容器初始化时，进行加载
    static {
        // 缓存 Long 值，注意这里是 i - 128 ，所以再拿的时候就需要 + 128
        for(int i = 0; i < cache.length; i++)
            cache[i] = new Long(i - 128);
    }
}
```

目录

3.1 为什么使用 Long 时，大家推荐多使用 valueOf 方法，少使用 parseLong 方法

答：因为 Long 本身有缓存机制，缓存了 -128 到 127 范围内的 Long，valueOf 方法会从缓存中去拿值，如果命中缓存，会减少资源的开销，parseLong 方法就没有这个机制。

3.2 如何解决 String 乱码的问题

答：乱码的问题的根源主要是两个：字符集不支持复杂汉字、二进制进行转化时字符集不匹配，所以在 String 乱码时我们可以这么做：

1. 所有可以指定字符集的地方强制指定字符集，比如 new String 和 getBytes 这两个地方；
2. 我们应该使用 UTF-8 这种能完整支持复杂汉字的字符集。

3.3 为什么大家都说 String 是不可变的

答：主要是因为 String 和保存数据的 char 数组，都被 final 关键字所修饰，所以是不可变的，具体细节描述可以参考上文。

3.4 String 一些常用操作问题，如何如何分割、合并、替换、删除、截取等等问题

答：这些都属于问 String 的基本操作题目，考察我们平时对 String 的使用熟练程度，可以参考上文。

总结

String 和 Long 在我们工作中使用频率很高，在面试的过程中，考官也喜欢问一些关于实际操作的问题，来考察我们的使用熟练度，所以本文中列举的一些 demo，大家可以试试手，完整的代码可以去 GitHub 上面去拉取。

← 01 开篇词：为什么学习本专栏　　03 Java 常用关键字理解 →

精选留言 54

欢迎在这里发表留言，作者筛选后可公开显示

走出深坑_爬出井底

1.6拆分与合并： String的join方法无法连续使用，是因为join方法处理的是参数elements中的元素吧，是将elements中的元素合并，他与调用join方法的字符串对象本身是没有任何关系的。

0 回复 2019-12-04

甜树果子二号

目录

法返回的是包装类型，只基于缓存命中这方面，这两者应该没有可比性吧，真的需要包装类型，也会自动装箱，一样的会命中缓存

0 回复 2019-11-27

qq_阿福_7 回复 **甜树果子二号**

同意你的说法。 字数你妹。

回复 2019-12-04 11:21:01

慕神9346227 回复 **甜树果子二号**

大概是如果要String->Long的转换用valueOf会多一个取缓存的优化，这是对装箱的优化

回复 9天前

Lxiaoyueyue

感觉讲重写equals方法，就只讲了String对象的实现，写的很针对。没有拓展重写equals方法应该遵循的规律，更期待的解答的是，在什么情况用getclass，在什么情况下用instanceof检测两个对象是否属于同一个类

0 回复 2019-11-25

文贺 回复 **Lxiaoyueyue**

equals 覆写的规律和文中 String 中的 equals 方法一样，我是希望通过 String 的 equals 方法给大家提供一种覆写的思路。比如说给你一个对象 DTO，如何覆写 equals 呢？我们可以使用 idea 自带生成 equals 的方法，也可以把依次判断 DTO 的各个属性。至于你说的 getClass，instanceof 属于另外的内容哈，可能不能符合每个读者的口味了。

回复 2019-11-30 13:40:59

窗下有梧桐

我觉得String的不变性不在于final修饰，因为char[]不属于基本类型，即引用不能改变，但是引用地址的值可以改变。String的不变性应该是由于它底层的源码导致的，每次给String的引用赋值其实是创建了一个新的对象。不知道我这么理解有没有问题

1 回复 2019-11-21

文贺 回复 **窗下有梧桐**

你可以用反证法想一下，如果 String 没有 final 修饰会怎么样？

回复 2019-11-23 16:37:07

---

qq_慕哥2339582

老师，我想问一下，Integer a = new Integer（12）创建了几个对象，它在内存中的存储过程是怎样的

0 回复 2019-11-02

古卷 回复 **qq_慕哥2339582**

目测是创建一个对象。

回复 2019-11-04 11:56:39

xiaobaicaisss 回复 **qq_慕哥2339582**

目测是2个，一个在堆，一个在栈

目录

qq_慕哥2339582

老师你好，string的常量池和long的缓冲池作用是一样的吗？也能起到一个存储的作用

0 回复 2019-11-01

文贺 回复 **qq_慕哥2339582**

同学你好，是的，原理是一样的，但是底层实现机制不同，都是起一个缓存的作用。

回复 2019-11-04 10:12:58

ChangleAmazing

老师，还是想问下 Long.valueOf(String) 和 Long.parseLong(String) 的问题。 源码中 valueOf 实际上也是调用了 parseLong 之后才走缓存的吧。即使是有缓存，应该会比 parseLong 更慢啊。

1 回复 2019-10-30

凉话

static final Long cache[] = new Long[-(-128) + 127 + 1]; 老师这一句话怎么就直接缓存了256个数字

0 回复 2019-10-30

文贺 回复 **凉话**

同学你好，cache.length 是 256 哈，其实下面还有代码的： static { for(int i = 0; i < cache.length; i++) cache[i] = new Long(i - 128); }

回复 2019-10-31 10:50:59

xiaobaicaisss 回复 **凉话**

下面那个静态代码块才是真正的缓存数据，类加载的时候静态代码块也运行，相当于把一个游泳池蓄上水，这里就相当于把那个数组里-128到127装满

回复 2019-11-20 19:52:09

studyHardHard

如果有StringBuffet和StringBuilder，立马下单

0 回复 2019-10-18

文贺 回复 **studyHardHard**

同学你好，没有的哈，StringBuffer 和 StringBuilder 比较简单，一个面试题就能说完，难以写成一篇文档，所以我们没有放进课程内容里面，不过有问题可以留言互相交流哈。

回复 2019-10-19 10:49:53

studyHardHard 回复 **文贺**

不简单吧，比如3者底层数组的扩容机制有什么不同，线程是否安全，各自的使用场景。还有就是关于String常量池， + 在进行String操作时底层调用的是StringBuilder。还有就是经典的求一段字符串操作创建了几个对象的问题，这些有的写的

回复 2019-10-20 09:21:21

慕仙6328494 回复 **studyHardHard**

……你这个也太基础了吧 如果真写这些，这个专栏感觉档次下降一般

目录

Blue_Fish0323

intern 这个方法的分析没有嘛？还是在后面有？

0 回复 2019-10-17

文贺 回复 **Blue_Fish0323**

同学你好，没有的哈，intern 方法几乎没有看见有同学实际用过，但有问题可以交流。

回复 2019-10-19 10:47:51

yaDONGgua

String s ="boo:and:foo"; s.split("o"); 运行结果是["b","",":and:f"] 为什么不是["b","",":and:f",""] ??

0 回复 2019-10-16

文贺 回复 **yaDONGgua**

点击展开剩余评论