

## JDK源码解析——序章

- 1、为什么要学习JDK源码?
- 2、如何学习JDK源码?

## JDK源码解析(1)——java.lang.Object类

- 1、Object 类的结构图
- 2、为什么java.lang包下的类不需要手动导入?
- 3、类构造器
- 4、equals 方法
- 5、getClass 方法
- 6、hashCode 方法
- 7、toString 方法
- 8、notify()/notifyAll()/wait()
- 10、registerNatives 方法

## JDK源码解析(2)——java.lang.Integer 类

- 1、Integer 的声明
- 2、Integer 的主要属性
- 3、构造方法 Integer(int) Integer(String)
- 4、toString() toString(int i) toString(int i, int radix)
- 5、自动拆箱和装箱
- 6、equals(Object obj)方法
- 7、hashCode() 方法
- 8、parseInt(String s) 和 parseInt(String s, int radix) 方法
- 9、compareTo(Integer anotherInteger) 和 compare(int x, int y) 方法

## JDK源码解析(3)——java.lang.String 类

- 1、String 类的定义
- 2、字段属性
- 3、构造方法
- 4、equals(Object anObject) 方法
- 5、hashCode() 方法
- 6、charAt(int index) 方法
- 7、compareTo(String anotherString) 和 compareToIgnoreCase(String str) 方法
- 8、concat(String str) 方法
- 9、indexOf(int ch) 和 indexOf(int ch, int fromIndex) 方法
- 10、split(String regex) 和 split(String regex, int limit) 方法
- 11、replace(char oldChar, char newChar) 和 String replaceAll(String regex, String replacement) 方法
- 12、substring(int beginIndex) 和 substring(int beginIndex, int endIndex) 方法
- 13、常量池
- 14、intern() 方法
- 15、String 真的不可变吗?

## JDK源码解析(4)——java.util.Arrays 类

- 1、asList
- 2、sort
- 3、binarySearch
- 4、copyOf
- 5、equals 和 deepEquals

6、fill

7、toString 和 deepToString

JDK源码解析(5)——java.util.ArrayList 类

1、ArrayList 定义

2、字段属性

3、构造函数

4、添加元素

5、删除元素

6、修改元素

7、查找元素

8、遍历集合

9、SubList

10、size()

11、isEmpty()

12、trimToSize()

JDK源码解析(6)——java.util.LinkedList 类

1、LinkedList 定义

2、字段属性

3、构造函数

4、添加元素

5、删除元素

6、修改元素

7、查找元素

8、遍历集合

9、迭代器和for循环效率差异

JDK源码解析(7)——java.util.HashMap 类

1、哈希表

2、什么是 HashMap?

3、HashMap定义

4、字段属性

5、构造函数

6、确定哈希桶数组索引位置

7、添加元素

8、扩容机制

9、删除元素

10、查找元素

11、遍历元素

12、总结

JDK源码解析(8)——java.util.HashSet 类

1、HashSet 定义

2、字段属性

3、构造函数

4、添加元素

5、删除元素

6、查找元素

7、遍历元素

JDK源码解析(9)——java.util.LinkedHashMap 类

1、LinkedHashMap 定义

2、字段属性

- 3、构造函数
- 4、添加元素
- 5、删除元素
- 6、查找元素
- 7、遍历元素
- 8、迭代器
- 9、总结

JDK源码解析(10)——java.util.LinkedHashSet类

- 1、LinkedHashSet 定义
- 2、构造函数
- 3、添加元素
- 4、删除元素
- 5、查找元素
- 6、遍历元素



微信搜一搜

Q IT可乐

## JDK源码解析——序章

曾经小时候的我，以为自己长大后可以拯救整个世界，可等长大后才发现整个世界都拯救不了我。直到我看到了这本书，给我的编码人生指明了道路。



哈哈，说笑了，这本书 可乐 将给大家介绍 JDK 中的各种常用类的源码解析，从浅入深，保证让你学完之后，彻底了解 JDK 源码的底层逻辑，如果你学完之后，还是没有理解，那当我没有保证（手动狗头）。



## 1、为什么要学习JDK源码？

### ①、学习优秀的代码

不可否认，目前Java是全球使用人数最多的一种语言，而且其历史悠久，从1996年第一个版本诞生，到现在2021年JDK16，25年的历史，其源码经过各路大神不断迭代，虽然有些历史原因造成的代码瑕疵，但总体来说，其源码设计是极其优秀的。

想学习代码规范？看JDK源码！

想学习代码设计模式？看JDK源码！

想学习优秀的算法逻辑？看JDK源码！

所以，学完JDK源码之后，你的代码能力会有很大的提高。

## ②、弄懂原理，快速解决问题

作为一个有经验的程序员，每次代码出现bug之后，我们习惯性的会去追溯源码，只有从根源上弄懂源码实现，我们才能快速的定位问题。

同理，JDK源码是基础，很多上层框架都脱离不了JDK源码，我们弄清楚JDK源码的底层实现，对我们快速的排查问题很有帮助。

## ③、面试所需

相信很多面试招聘上，都写着一条：对Java有比较深刻的理解。如果你弄懂JDK源码，不失为你面试的一大亮点。而且面试官通常喜欢问JDK中集合类源码、并发类源码的实现逻辑，比如HashMap的扩容机制，底层数据结构实现；ConcurrentHashMap 怎么保证线程安全，是如何实现的，JDK1.7和JDK1.8实现原理又有什么区别？等等，这些问题，有的可能临时抱佛脚，突击一下面试答案，但是如果你深入学习了JDK源码，我相信这些对你而言，就可以和面试官侃侃而谈了。

## ④、催眠必备

额 ..... 不过的确有效（滑稽）。

# 2、如何学习JDK源码？

## ①、按需阅读

切记，切记，不要一开始就想把整个源码吃透，通常来说，源码是比较晦涩难懂的，想一口气吃成一个胖子，只会打击你的自信心。

所以我们要根据需求去阅读相应的源码，比如，我想学习如何使用 ArrayList 这个类，这是一个集合，首先我们可能是要用到其增删改查的功能吧，那么我们可以先去看其增删改查的源码；看完之后，我们可能又想看到其为什么不声明长度，可以存放任意长度的数据，于是我们去了解其扩容机制；等等诸如此类，根据需求去拆解其源码，慢慢消化。

## ②、学会调试

一段代码，其执行过程通常不会按照书写顺序来，所以为了追溯代码的实现逻辑，我们可以通过不断的调试debug，逐步拆解。日常用的开发工具IntelliJ IDEA 以及eclipse等，都可以很好的进行调试。

## ③、多画流程图

光说不练，光看不练，都是不行的，如果想彻底了解源码，你得学会画流程图，类结构图，类运行逻辑图等等。

talk is cheap, show me the image

## ④、必要的知识储备

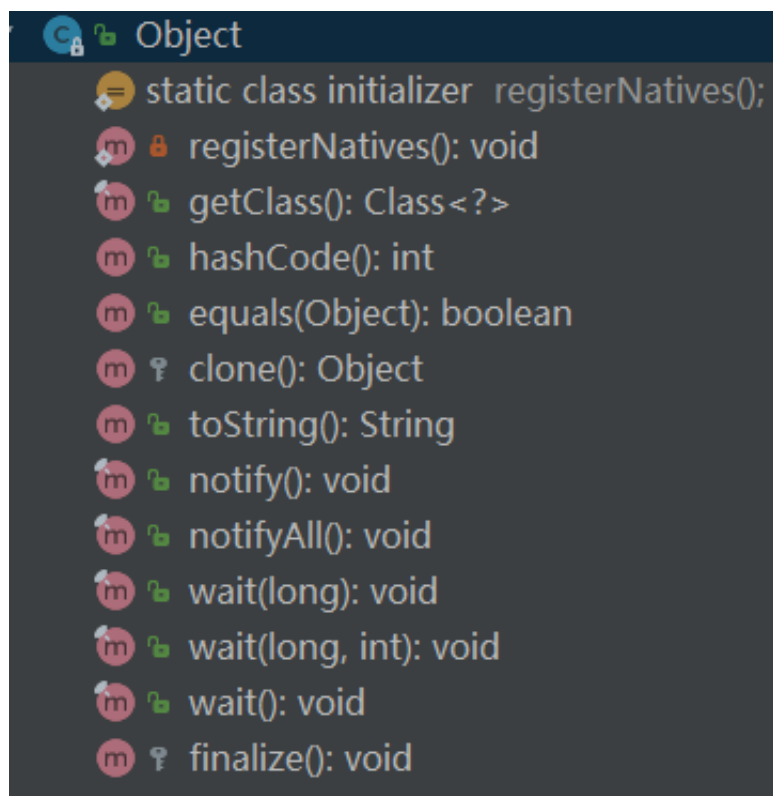
因为源码中通常都会涉及到一些常用的设计模式，数据结构和算法等，如果你能提前了解这些方面的资料，那么读起源码，将会事半功倍，否则会吃力很多。

# JDK源码解析(1)——java.lang.Object类

对于Java源码，而且是第一篇文章，可乐当然要为大伙介绍JDK中使用最频繁的类，那就是所有类的基类——java.lang.Object。

Object 类属于 java.lang 包，此包下的所有类在使用时无需手动导入，系统会在程序编译期间自动导入。Object 类是所有类的基类，当一个类没有直接继承某个类时，默认继承Object类，也就是说任何类都直接或间接继承此类，Object 类中能访问的方法在所有类中都可以调用，下面我们会分别介绍Object 类中的所有方法。

## 1、Object 类的结构图



Object.class类

```
1  /*
2   * Copyright (c) 1994, 2012, Oracle and/or its affiliates. All rights
   reserved.
3   * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
4   *
5   */
6
7  package java.lang;
8
9  /**
10   * Class {@code Object} is the root of the class hierarchy.
11   * Every class has {@code Object} as a superclass. All objects,
12   * including arrays, implement the methods of this class.
13   *
14   * @author unascribed
```

```
15  * @see      java.lang.Class
16  * @since    JDK1.0
17  */
18  public class Object {
19
20      private static native void registerNatives();
21      static {
22          registerNatives();
23      }
24
25      public final native Class<?> getClass();
26
27      public native int hashCode();
28
29      public boolean equals(Object obj) {
30          return (this == obj);
31      }
32
33      protected native Object clone() throws CloneNotSupportedException;
34
35      public String toString() {
36          return getClass().getName() + "@" +
Integer.toHexString(hashCode());
37      }
38
39      public final native void notify();
40
41      public final native void notifyAll();
42
43      public final native void wait(long timeout) throws
InterruptedException;
44
45      public final void wait(long timeout, int nanos) throws
InterruptedException {
46          if (timeout < 0) {
47              throw new IllegalArgumentException("timeout value is
negative");
48          }
49          if (nanos < 0 || nanos > 999999) {
50              throw new IllegalArgumentException(
51                  "nanosecond timeout value out of range");
52          }
53          if (nanos > 0) {
54              timeout++;
55          }
56          wait(timeout);
57      }
58
59      public final void wait() throws InterruptedException {
```

```
60         wait(0);
61     }
62
63     protected void finalize() throws Throwable { }
64 }
```

## 2、为什么java.lang包下的类不需要手动导入？

不知道大家注意到没，我们在使用诸如Date类时，需要手动导入import java.util.Date，再比如使用File类时，也需要手动导入import java.io.File。但是我们在使用Object类，String 类，Integer类等不需要手动导入，而能直接使用，这是为什么呢？

这里先告诉大家一个结论：使用 java.lang 包下的所有类，都不需要手动导入。

另外我们介绍一下Java中的两种导包形式，导包有两种方法：

①、单类型导入（single-type-import），例如import java.util.Date

②、按需类型导入(type-import-on-demand)，例如import java.util.\*

单类型导入比较好理解，我们编程所使用的各种工具默认都是按照单类型导包的，需要什么类便导入什么类，这种方式是导入指定的public类或者接口；

按需类型导入，比如 import java.util.\*，可能看到后面的\*，大家会以为是导入java.util包下的所有类，其实并不是这样，我们根据名字按需导入要知道他是按照需求导入，并不是导入整个包下的所有类。

Java编译器会从启动目录(bootstrap)，扩展目录(extension)和用户类路径下去定位需要导入的类，而这些目录进仅仅是给出了类的顶层目录，编译器的类文件定位方法大致可以理解为如下公式：

顶层路径名 \ 包名 \ 文件名.class = 绝对路径

单类型导入我们知道包名和文件名，所以编译器可以一次性查找定位到所要的类文件。按需类型导入则比较复杂，编译器会把包名和文件名进行排列组合，然后对所有的可能性进行类文件查找定位。例如：

```
1 package com;
2
3 import java.io.*;
4
5 import java.util.*;
```

如果我们文件中使用到了 File 类，那么编译器会根据如下几个步骤来进行查找 File 类：

①、File // File类属于无名包，就是说File类没有package语句，编译器会首先搜索无名包

②、com.File // File类属于当前包，就是我们当前编译类的包路径

③、java.lang.File //由于编译器会自动导入java.lang包，所以也会从该包下查找

④、java.io.File

⑤、java.util.File



.....

需要注意的地方就是，编译器找到java.io.File类之后并不会停止下一步的寻找，而要把所有的可能性都查找完以确定是否有类导入冲突。假设此时的顶层路径有三个，那么编译器就会进行 $3 \times 5 = 15$ 次查找。

如果在查找完成后，编译器发现了两个同名的类，那么就会报错。要删除你不用那个类，然后再编译。

所以我们可以得出这样的结论：按需类型导入是绝对不会降低Java代码的执行效率的，但会影响到Java代码的编译速度。所以我们在编码时最好是使用单类型导入，这样不仅能提高编译速度，也能避免命名冲突。

讲清楚Java的两种导包类型了，我们在回到为什么可以直接使用 Object 类，看到上面查找类文件的第③步，编译器会自动导入 java.lang 包，那么当然我们能直接使用了。至于原因，因为用的多，提前加载了，省资源。

### 3、类构造器

我们知道类构造器是创建Java对象的途径之一，通过new 关键字调用构造器完成对象的实例化，还能通过构造器对对象进行相应的初始化。一个类必须要有一个构造器的存在，如果没有显示声明，那么系统会默认创建一个无参构造器，在JDK的Object类源码中，是看不到构造器的，系统会自动添加一个无参构造器。我们可以通过：

Object obj = new Object(); 构造一个Object类的对象。

### 4、equals 方法

通常很多面试题都会问 equals() 方法和 == 运算符的区别，== 运算符用于比较基本类型的值是否相同，或者比较两个对象的引用是否相等，而 equals 用于比较两个对象是否相等，这样说可能比较宽泛，两个对象如何才是相等的呢？这个标尺该如何定？

我们可以看看 Object 类中的equals 方法：

```
1 public boolean equals(Object obj) {  
2     return (this == obj);  
3 }
```

可以看到，在 Object 类中，== 运算符和 equals 方法是等价的，都是比较两个对象的引用是否相等，从另一方面来讲，如果两个对象的引用相等，那么这两个对象一定是相等的。对于我们自定义的一个对象，如果不重写 equals 方法，那么在比较对象的时候就是调用 Object 类的 equals 方法，也就是用 == 运算符比较两个对象。我们可以看看 String 类中的重写的 equals 方法：

```
1 public boolean equals(Object anObject) {  
2     if (this == anObject) {  
3         return true;  
4     }  
5     if (anObject instanceof String) {  
6         String anotherString = (String)anObject;  
7         int n = value.length;  
8         if (n == anotherString.value.length) {
```

```

9         char v1[] = value;
10        char v2[] = anotherString.value;
11        int i = 0;
12        while (n-- != 0) {
13            if (v1[i] != v2[i])
14                return false;
15            i++;
16        }
17        return true;
18    }
19 }
20 return false;
21 }

```

String 是引用类型，比较时不能比较引用是否相等，重点是字符串的内容是否相等。所以 String 类定义两个对象相等的标准是字符串内容都相同。

在Java规范中，对 equals 方法的使用必须遵循以下几个原则：

- ①、自反性：对于任何非空引用值 x，x.equals(x) 都应返回 true。
- ②、对称性：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。
- ③、传递性：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true，并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。
- ④、一致性：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改
- ⑤、对于任何非空引用值 x，x.equals(null) 都应返回 false。

下面我们自定义一个 Person 类，然后重写其 equals 方法，比较两个 Person 对象：

```

1 package com.js.bean;
2 /**
3  * Create by vae
4  */
5 public class Person {
6     private String pname;
7     private int page;
8
9     public Person(){}
10
11     public Person(String pname,int page){
12         this.pname = pname;
13         this.page = page;
14     }
15     public int getPage() {
16         return page;
17     }

```

```

18
19     public void setPage(int page) {
20         this.page = page;
21     }
22
23     public String getPname() {
24         return pname;
25     }
26
27     public void setName(String pname) {
28         this.pname = pname;
29     }
30     @Override
31     public boolean equals(Object obj) {
32         if(this == obj){//引用相等那么两个对象当然相等
33             return true;
34         }
35         if(obj == null || !(obj instanceof Person)){//对象为空或者不是Person
类的实例
36             return false;
37         }
38         Person otherPerson = (Person)obj;
39         if(otherPerson.getPname().equals(this.getPname()) &&
otherPerson.getPage()==this.getPage()){
40             return true;
41         }
42         return false;
43     }
44
45     public static void main(String[] args) {
46         Person p1 = new Person("Tom",21);
47         Person p2 = new Person("Marry",20);
48         System.out.println(p1==p2);//false
49         System.out.println(p1.equals(p2));//false
50
51         Person p3 = new Person("Tom",21);
52         System.out.println(p1.equals(p3));//true
53     }
54
55 }

```

通过重写 equals 方法，我们自定义两个对象相等的标尺为Person对象的两个属性都相等，则对象相等，否则不相等。如果不重写 equals 方法，那么始终是调用 Object 类的equals 方法，也就是用 == 比较两个对象在栈内存中的引用地址是否相等。

这时候有个Person 类的子类 Man，也重写了 equals 方法：

```

1 package com.js.bean;
2 /**

```

```

3  * Create by vae
4  */
5  public class Man extends Person{
6      private String sex;
7
8      public Man(String pname,int page,String sex){
9          super(pname,page);
10         this.sex = sex;
11     }
12     @Override
13     public boolean equals(Object obj) {
14         if(!super.equals(obj)){
15             return false;
16         }
17         if(obj == null || !(obj instanceof Man)){//对象为空或者不是Person类的
实例
18             return false;
19         }
20         Man man = (Man) obj;
21         return sex.equals(man.sex);
22     }
23
24     public static void main(String[] args) {
25         Person p = new Person("Tom",22);
26         Man m = new Man("Tom",22,"男");
27
28         System.out.println(p.equals(m));//true
29         System.out.println(m.equals(p));//false
30     }
31 }

```

通过打印结果我们发现 `person.equals(man)` 得到的结果是 `true`，而 `man.equals(person)` 得到的结果却是 `false`，这显然是不正确的。

问题出现在 `instanceof` 关键字上，关于 `instanceof` 关键字的用法，可以参考我的这篇文章：<http://www.cnblogs.com/ysocean/p/8486500.html>

`Man` 是 `Person` 的子类，`person instanceof Man` 结果当然是 `false`。这违反了我们上面说的对称性。

实际上用 `instanceof` 关键字是做不到对称性的要求的。这里推荐做法是用 `getClass()` 方法取代 `instanceof` 运算符。`getClass()` 关键字也是 `Object` 类中的一个方法，作用是返回一个对象的运行时类，下面我们会详细讲解。

那么 `Person` 类中的 `equals` 方法为

```

1 public boolean equals(Object obj) {
2     if(this == obj){//引用相等那么两个对象当然相等
3         return true;
4     }
5     if(obj == null || (getClass() != obj.getClass())){//对象为空或者不是
        Person类的实例
6         return false;
7     }
8     Person otherPerson = (Person)obj;
9     if(otherPerson.getName().equals(this.getName()) &&
        otherPerson.getPage()==this.getPage()){
10         return true;
11     }
12     return false;
13 }

```

打印结果 person.equals(man)得到的结果是 false，man.equals(person)得到的结果也是false，满足对称性。

注意：使用 getClass 不是绝对的，要根据情况而定，毕竟定义对象是否相等的标准是由程序员自己定义的。而且使用 getClass 不符合多态的定义，比如 AbstractSet 抽象类，它有两个子类 TreeSet 和 HashSet,他们分别使用不同的算法实现查找集合的操作，但无论集合采用哪种方式实现，都需要拥有对两个集合进行比较的功能，如果使用 getClass 实现equals方法的重写，那么就不能在两个不同子类的对象进行相等的比较。而且集合类比较特殊，其子类是不需要自定义相等的概念的。

所以什么时候使用 instanceof 运算符，什么时候使用 getClass() 有如下建议：

①、如果子类能够拥有自己的相等概念，则对称性需求将强制采用 getClass 进行检测。

②、如果有超类决定相等的概念，那么就可以使用 instanceof 进行检测，这样可以在不同的子类的对象之间进行相等的比较。

下面给出一个完美的 equals 方法的建议：

1、显示参数命名为 otherObject，稍后会将它转换成另一个叫做 other 的变量。

2、判断比较的两个对象引用是否相等，如果引用相等那么表示是同一个对象，那么当然相等

3、如果 otherObject 为 null，直接返回false，表示不相等

4、比较 this 和 otherObject 是否是同一个类：如果 equals 的语义在每个子类中有所改变，就使用 getClass 检测；如果所有的子类都有统一的定义，那么使用 instanceof 检测

5、将 otherObject 转换成对应的类类型变量

6、最后对对象的属性进行比较。使用 == 比较基本类型，使用 equals 比较对象。如果都相等则返回true，否则返回false。注意如果是在子类中定义equals，则要包含 super.equals(other)

下面我们给出 Person 类中完整的 equals 方法的书写：

```

1 @Override
2 public boolean equals(Object otherObject) {

```

```

3      //1、判断比较的两个对象引用是否相等，如果引用相等那么表示是同一个对象，那么当然
相等
4      if(this == otherObject){
5          return true;
6      }
7      //2、如果 otherObject 为 null，直接返回false，表示不相等
8      if(otherObject == null ){//对象为空或者不是Person类的实例
9          return false;
10     }
11     //3、比较 this 和 otherObject 是否是同一个类（注意下面两个只能使用一种）
12     //3.1：如果 equals 的语义在每个子类中所有改变，就使用 getClass 检测
13     if(this.getClass() != otherObject.getClass()){
14         return false;
15     }
16     //3.2：如果所有的子类都有统一的定义，那么使用 instanceof 检测
17     if(!(otherObject instanceof Person)){
18         return false;
19     }
20
21     //4、将 otherObject 转换成对应的类类型变量
22     Person other = (Person) otherObject;
23
24     //5、最后对对象的属性进行比较。使用 == 比较基本类型，使用 equals 比较对象。如
果都相等则返回true，否则返回false
25     // 使用 Objects 工具类的 equals 方法防止比较的两个对象有一个为 null而报
错，因为 null.equals() 是会抛异常的
26     return Objects.equals(this.pname,other.pname) && this.page ==
other.page;
27
28     //6、注意如果是在子类中定义equals，则要包含 super.equals(other)
29     //return super.equals(other) &&
Objects.equals(this.pname,other.pname) && this.page == other.page;
30
31     }

```

请注意，无论何时重写此方法，通常都必须重写hashCode方法，以维护hashCode方法的一般约定，该方法声明相等对象必须具有相同的哈希代码。hashCode 也是 Object 类中的方法，后面会详细讲解。

## 5、getClass 方法

上面我们在介绍 equals 方法时，介绍如果 equals 的语义在每个子类中有所改变，那么使用 getClass 检测，为什么这样说呢？

getClass()在 Object 类中如下，作用是返回对象的运行时类。

```

1  public final native Class<?> getClass();

```

这是一个用 native 关键字修饰的方法，关于 native 关键字的详细介绍如下：<http://www.cnblogs.com/ysocean/p/8476933.html>

这里我们要知道用 native 修饰的方法我们不用考虑，由操作系统帮我们实现，该方法的作用是返回一个对象的运行时类，通过这个类对象我们可以获取该运行时类的相关属性和方法。也就是Java中的反射，各种通用的框架都是利用反射来实现的，这里我们不做详细的描述。

这里详细的介绍 getClass 方法返回的是一个对象的运行时类对象，这该怎么理解呢？Java中还有一种这样的用法，通过 类名.class 获取这个类的类对象，这两种用法有什么区别呢？

父类：Parent.class

```
1 public class Parent {}
```

子类：Son.class

```
1 public class Son extends Parent{}
```

测试：

```
1 @Test
2 public void testClass(){
3     Parent p = new Son();
4     System.out.println(p.getClass());
5     System.out.println(Parent.class);
6 }
```

打印结果：

```
class com.js.test.Son
class com.js.test.Parent
```

结论：class 是一个类的属性，能获取该类编译时的类对象，而 getClass() 是一个类的方法，它是获取该类运行时的类对象。

还有一个需要大家注意的是，虽然Object类中getClass() 方法声明是：public final native Class getClass();返回的是一个 Class，但是如下是能通过编译的：

```
1 Class<? extends String> c = "".getClass();
```

也就是说类型为T的变量getClass方法的返回值类型其实是Class而非getClass方法声明中的Class。

这在官方文档中也有说明：<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass-->

## 6、hashCode 方法

hashCode 在 Object 类中定义如下：

```
1 public native int hashCode();
```

这也是一个用 native 声明的本地方法，作用是返回对象的散列码，是 int 类型的数值。

那么这个方法存在的意义是什么呢？

我们知道在Java 中有几种集合类，比如 List,Set，还有 Map，List集合一般是存放的元素是有序可重复的，Set 存放的元素则是无序不可重复的，而 Map 集合存放的是键值对。

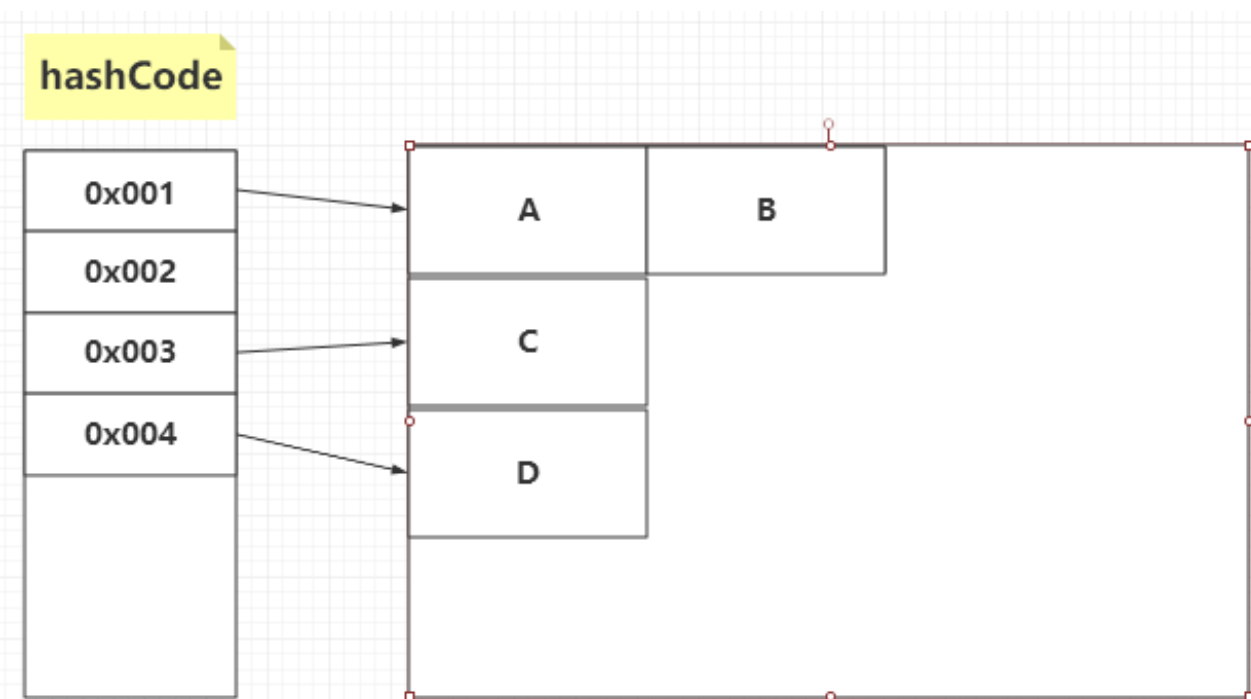
前面我们说过判断一个元素是否相等可以通过 equals 方法，没增加一个元素，那么我们就通过 equals 方法判断集合中的每一个元素是否重复，但是如果集合中有10000个元素了，但我们新加入一个元素时，那就需要进行10000次equals方法的调用，这显然效率很低。

于是，Java 的集合设计者就采用了 哈希表 来实现。关于哈希表的数据结构我有过介绍。哈希算法也称为散列算法，是将数据依特定算法产生的结果直接指定到一个地址上。这个结果就是由 hashCode 方法产生。这样一来，当集合要添加新的元素时，先调用这个元素的 hashCode 方法，就一下子能定位到它应该放置的物理位置上。

①、如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；

②、如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了；

③、不相同的话，也就是发生了Hash key相同导致冲突的情况，那么就在这个Hash key的地方产生一个链表，将所有产生相同HashCode的对象放到这个单链表上去，串在一起（很少出现）。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。



这里有 A,B,C,D四个对象，分别通过 hashCode 方法产生了三个值，注意 A 和 B 对象调用 hashCode 产生的值是相同的，即 A.hashCode() = B.hashCode() = 0x001,发生了哈希冲突，这时候由



于最先是插入了 A，在插入的B的时候，我们发现 B 是要插入到 A 所在的位置，而 A 已经插入了，这时候就通过调用 equals 方法判断 A 和 B 是否相同，如果相同就不插入 B，如果不同则将 B 插入到 A 后面的位置。所以对于 equals 方法和 hashCode 方法有如下要求：

#### 一、hashCode 要求

①、在程序运行时期，只要对象的（字段的）变化不会影响equals方法的决策结果，那么，在这个期间，无论调用多少次hashCode，都必须返回同一个散列码。

②、通过equals调用返回true 的2个对象的hashCode一定一样。

③、通过equals返回false 的2个对象的散列码不需要不同，也就是他们的hashCode方法的返回值允许出现相同的情况。

因此我们可以得到如下推论：

两个对象相等，其 hashCode 一定相同；

两个对象不相等，其 hashCode 有可能相同；

hashCode 相同的两个对象，不一定相等；

hashCode 不相同的两个对象，一定不相等；

这四个推论通过上图可以更好的理解。

可能会有人疑问，对于不能重复的集合，为什么不直接通过 hashCode 对于每个元素都产生唯一的值，如果重复就是相同的值，这样不就不需要调用 equals 方法来判断是否相同了吗？

实际上对于元素不是很多的情况下，直接通过 hashCode 产生唯一的索引值，通过这个索引值能直接找到元素，而且还能判断是否相同。比如数据库存储的数据，ID 是有序排列的，我们能通过 ID 直接找到某个元素，如果新插入的元素 ID 已经有了，那就表示是重复数据，这是很完美的办法。但现实是存储的元素很难有这样的 ID 关键字，也就很难这种实现 hashCode 的唯一算法，再者就算能实现，但是产生的 hashCode 码是非常大的，这会大的超过 Java 所能表示的范围，很占内存空间，所以也是不予考虑的。

#### 二、hashCode 编写指导：

①、不同对象的hash码应该尽量不同，避免hash冲突，也就是算法获得的元素要尽量均匀分布。

②、hash 值是一个 int 类型，在Java中占用 4 个字节，也就是 2<sup>32</sup> 次方，要避免溢出。

在 JDK 的 Integer类，Float 类，String 类等都重写了 hashCode 方法，我们自定义对象也可以参考这些类来写。

下面是 JDK String 类的hashCode 源码：

```

1 public int hashCode() {
2     int h = hash;
3     if (h == 0 && value.length > 0) {
4         char val[] = value;
5
6         for (int i = 0; i < value.length; i++) {
7             h = 31 * h + val[i];
8         }
9         hash = h;
10    }
11    return h;
12 }

```

再次提醒大家，对于 Map 集合，我们可以选取Java中的基本类型，还有引用类型 String 作为 key，因为它们都按照规范重写了 equals 方法和 hashCode 方法。但是如果你用自定义对象作为 key，那么一定要覆写 equals 方法和 hashCode 方法，不然会有意想不到的错误产生。

## 7、toString 方法

该方法在 JDK 的源码如下：

```

1 public String toString() {
2     return getClass().getName() + "@" + Integer.toHexString(hashCode());
3 }

```

getClass().getName()是返回对象的全类名（包含包名），Integer.toHexString(hashCode()) 是以16进制无符号整数形式返回此哈希码的字符串表示形式。

打印某个对象时，默认是调用 toString 方法，比如 System.out.println(person),等价于 System.out.println(person.toString())

## 8、notify()/notifyAll()/wait()

这是用于多线程之间的通信方法，在后面讲解多线程会详细描述，这里就不做讲解了。

```

1 protected void finalize() throws Throwable { }

```

该方法用于垃圾回收，一般由 JVM 自动调用，一般不需要程序员去手动调用该方法。后面再讲解 JVM 的时候会详细展开描述。

## 10、registerNatives 方法

该方法在 Object 类中定义如下：

```

1 private static native void registerNatives();

```

这是一个本地方法，在 native 介绍 中我们知道一个类定义了本地方法后，想要调用操作系统的实现，必须还要装载本地库，但是我们发现在 `Object.class` 类中具有很多本地方法，但是却没有看到本地库的载入代码。而且这是用 `private` 关键字声明的，在类外面根本调用不了，我们接着往下看关于这个方法的类似源码：

```
1      static {  
2          registerNatives();  
3      }
```

看到上面的代码，这就明白了吧。静态代码块就是一个类在初始化过程中必定会执行的内容，所以在类加载的时候是会执行该方法的，通过该方法来注册本地方法。

参考文档：<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别刷选的书籍资料



微信搜一搜

IT可乐

## JDK源码解析(2)——`java.lang.Integer` 类

上一篇文章我们介绍了 `java.lang` 包下的 `Object` 类，那么本篇文章接着介绍该包下的另一个也很常用的类 `Integer`。

### 1、`Integer` 的声明

```
1 public final class Integer extends Number implements Comparable<Integer>{}
```

`Integer` 是用 `final` 声明的常量类，不能被任何类所继承。并且 `Integer` 类继承了 `Number` 类并实现了 `Comparable` 接口。`Number` 类是一个抽象类，8中基本数据类型的包装类除了 `Character` 和 `Boolean` 没有继承该类外，剩下的都继承了 `Number` 类，该类的方法用于各种数据类型的转换。`Comparable` 接口就一个 `compareTo` 方法，用于元素之间的大小比较，下面会对这些方法详细展开介绍。

## 2、Integer 的主要属性

```
MIN_VALUE: int = -2147483648
MAX_VALUE: int = 2147483647
TYPE: Class<Integer> = Class.getPrimitiveClass(...)
  digits: char[] = new char[]{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', ...
  DigitTens: char[] = new char[]{'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '1', '1', '1', '1', '1', '1', '1', ...
  DigitOnes: char[] = new char[]{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '1', '2', '3', '4', '5', '6', '7', ...
  sizeTable: int[] = new int[]{9, 99, 999, 9999, 99999, 999999, 9999999, 99999999, 999999999, 9999999999, 2147483647}
  value: int
  SIZE: int = 32
  BYTES: int = 4
  serialVersionUID: long = 1360826667806852920L
```

### 字段

修饰符和类型	字段和说明
static int	<b>BYTES</b> 用于以int二进制补码形式表示值的字节数。
static int	<b>MAX_VALUE</b> 保持最大值的常数int可以有 $2^{31}-1$ 。
static int	<b>MIN_VALUE</b> 持有最小值的常数int可以有 $-2^{31}$ 。
static int	<b>SIZE</b> 用于表示int二进制补码形式的值的位数。
static Class<Integer>	<b>TYPE</b> Class表示原始类型的实例 int。

int 类型在 Java 中是占据 4 个字节，所以其可以表示大小的范围是  $-2^{31}$ —— $2^{31}-1$  即 -2147483648——2147483647，我们在用 int 表示数值时一定不要超出这个范围了。

## 3、构造方法 Integer(int) Integer(String)

```
1 public Integer(int var1) {
2     this.value = var1;
3 }
```

对于第二个构造方法 Integer(String) 就是我们将我们输入的字符串数据转换成整型数据。

首先我们必须要知道能转换成整数的字符串必须分为两个部分：第一位必须是"+"或者"-", 剩下的必须是 0-9 和 a-z 字符

```
1 public Integer(String s) throws NumberFormatException {
2     this.value = parseInt(s, 10); // 首先调用parseInt(s, 10)方法，其中s表示我们需要转换的字符串，10表示以十进制输出，默认也是10进制
3 }
4
5 public static int parseInt(String s, int radix) throws
    NumberFormatException{
```

```

6      //如果转换的字符串如果为null, 直接抛出空指针异常
7      if (s == null) {
8          throw new NumberFormatException("null");
9      }
10     //如果转换的radix(默认是10)<2 则抛出数字格式异常, 因为进制最小是 2 进制
11     if (radix < Character.MIN_RADIX) {
12         throw new NumberFormatException("radix " + radix +
13             " less than Character.MIN_RADIX");
14     }
15     //如果转换的radix(默认是10)>36 则抛出数字格式异常, 因为0到9一共10位, a到z一共26
    位, 所以一共36位
16     //也就是最高只能有36进制数
17     if (radix > Character.MAX_RADIX) {
18         throw new NumberFormatException("radix " + radix +
19             " greater than
    Character.MAX_RADIX");
20     }
21     int result = 0;
22     boolean negative = false;
23     int i = 0, len = s.length();//len是待转换字符串的长度
24     int limit = -Integer.MAX_VALUE;//limit = -2147483647
25     int multmin;
26     int digit;
27     //如果待转换字符串长度大于 0
28     if (len > 0) {
29         char firstChar = s.charAt(0);//获取待转换字符串的第一个字符
30         //这里主要用来判断第一个字符是"+"或者"-", 因为这两个字符的 ASCII码都小于字
    符'0'
31         if (firstChar < '0') {
32             if (firstChar == '-') {//如果第一个字符是 '-'
33                 negative = true;
34                 limit = Integer.MIN_VALUE;
35             } else if (firstChar != '+')//如果第一个字符是不是 '+', 直接抛出异常
36                 throw NumberFormatException.forInputString(s);
37
38             if (len == 1) //待转换字符长度是1, 不能是单独的"+"或者"-", 否则抛出异
    常
39                 throw NumberFormatException.forInputString(s);
40             i++;
41         }
42         multmin = limit / radix;
43         //通过不断循环, 将字符串除掉第一个字符之后, 根据进制不断相乘在相加得到一个正整
    数
44         //比如 parseInt("2abc",16) = 2*16的3次方+10*16的2次方+11*16+12*1
45         //parseInt("123",10) = 1*10的2次方+2*10+3*1
46         while (i < len) {
47             digit = Character.digit(s.charAt(i++),radix);
48             if (digit < 0) {
49                 throw NumberFormatException.forInputString(s);

```

```

50         }
51         if (result < multmin) {
52             throw NumberFormatException.forInputString(s);
53         }
54         result *= radix;
55         if (result < limit + digit) {
56             throw NumberFormatException.forInputString(s);
57         }
58         result -= digit;
59     }
60 } else { //如果待转换字符串长度小于等于0, 直接抛出异常
61     throw NumberFormatException.forInputString(s);
62 }
63 //根据第一个字符得到的正负号, 在结果前面加上符号
64 return negative ? result : -result;
65 }

```

## 4、toString() toString(int i) toString(int i, int radix)

这三个方法重载，能返回一个整型数据所表示的字符串形式，其中最后一个方法 toString(int,int) 第二个参数是表示的进制数。

```

1  public String toString() {
2      return toString(value);
3  }
4
5  public static String toString(int i) {
6      if (i == Integer.MIN_VALUE)
7          return "-2147483648";
8      int size = (i < 0) ? stringSize(-i) + 1 : stringSize(i);
9      char[] buf = new char[size];
10     getChars(i, size, buf);
11     return new String(buf, true);
12 }

```

toString(int) 方法内部调用了 stringSize() 和 getChars() 方法，stringSize() 它是用来计算参数 i 的位数也就是转成字符串之后的字符串的长度，内部结合一个已经初始化好的int类型的数组sizeTable来完成这个计算。

```

1  final static int [] sizeTable = { 9, 99, 999, 9999, 99999, 999999, 9999999,
2                                     99999999, 999999999, Integer.MAX_VALUE
3  };
4
5  // Requires positive x
6  static int stringSize(int x) {
7      for (int i=0; ; i++)
8          if (x <= sizeTable[i])
9              return i+1;
10 }

```

实现的形式很巧妙。注意负数包含符号位，所以对于负数的位数是 `stringSize(-i) + 1`。

再看 `getChars` 方法：

```

1  static void getChars(int i, int index, char[] buf) {
2      int q, r;
3      int charPos = index;
4      char sign = 0;
5
6      if (i < 0) {
7          sign = '-';
8          i = -i;
9      }
10
11     // Generate two digits per iteration
12     while (i >= 65536) {
13         q = i / 100;
14         // really: r = i - (q * 100);
15         r = i - ((q << 6) + (q << 5) + (q << 2));
16         i = q;
17         buf [--charPos] = DigitOnes[r];
18         buf [--charPos] = DigitTens[r];
19     }
20
21     // Fall thru to fast mode for smaller numbers
22     // assert(i <= 65536, i);
23     for (;;) {
24         q = (i * 52429) >>> (16+3);
25         r = i - ((q << 3) + (q << 1)); // r = i-(q*10) ...
26         buf [--charPos] = digits[r];
27         i = q;
28         if (i == 0) break;
29     }
30     if (sign != 0) {
31         buf [--charPos] = sign;
32     }
33 }

```

i:被初始化的数字,

index:这个数字的长度(包含了负数的符号"-"),

buf:字符串的容器-一个char型数组。

第一个if判断, 如果i<0,sign记下它的符号"-", 同时将i转成整数。下面所有的操作也就只针对整数了, 最后在判断sign如果不等于零将 sign 你的值放在char数组的首位buf[--charPos] = sign;。

## 5、自动拆箱和装箱

自动拆箱和自动装箱是JDK1.5以后才有的功能, 也就是java当中众多的语法糖之一, 它的执行是在编译期, 会根据代码的语法, 在生成class文件的时候, 决定是否进行拆箱和装箱动作。

### ①、自动装箱

我们知道一般创建一个类的对象需要通过 new 关键字, 比如:

```
Object obj = new Object();
```

但是实际上, 对于 Integer 类, 我们却可以直接这样使用:

```
Integer a = 128;
```

为什么可以这样, 通过反编译工具, 我们可以看到, 生成的class文件是:

```
Integer a = Integer.valueOf(128);
```

我们可以看看 valueOf() 方法

```
1 public static Integer valueOf(int i) {
2     assert IntegerCache.high >= 127;
3     if (i >= IntegerCache.low && i <= IntegerCache.high)
4         return IntegerCache.cache[i + (-IntegerCache.low)];
5     return new Integer(i);
6 }
```

其实最后返回的也是通过new Integer() 产生的对象, 但是这里要注意前面的一段代码, 当i的值 -128 <= i <= 127 返回的是缓存类中的对象, 并没有重新创建一个新的对象, 这在通过 equals 进行比较的时候我们要注意。

这就是基本数据类型的自动装箱, 128是基本数据类型, 然后被解析成Integer类。

### ②、自动拆箱

我们将 Integer 类表示的数据赋值给基本数据类型int, 就执行了自动拆箱。

```
1 Integer a = new Integer(128);
2 int m = a;
```

反编译生成的class文件:

```
1 Integer a = new Integer(128);
2 int m = a.intValue();
```



简单来讲：自动装箱就是 Integer.valueOf(int i); 自动拆箱就是 i.intValue();

## 6、equals(Object obj) 方法

```
1 public boolean equals(Object obj) {
2     if (obj instanceof Integer) {
3         return value == ((Integer)obj).intValue();
4     }
5     return false;
6 }
```

这个方法很简单，先通过 instanceof 关键字判断两个比较对象的关系，然后将对象强转为 Integer，在通过自动拆箱，转换成两个基本数据类 int，然后通过 == 比较。

## 7、hashCode() 方法

```
1 public int hashCode() {
2     return value;
3 }
```

Integer 类的 hashCode 方法也比较简单，直接返回其 int 类型的数据。

## 8、parseInt(String s) 和 parseInt(String s, int radix) 方法

前面通过 toString(int i) 可以将整型数据转换成字符串类型输出，这里通过 parseInt(String s) 能将字符串转换成整型输出。

这两个方法我们在介绍 构造函数 Integer(String s) 时已经详细讲解了。

## 9、compareTo(Integer anotherInteger) 和 compare(int x, int y) 方法

```
1 public int compareTo(Integer anotherInteger) {
2     return compare(this.value, anotherInteger.value);
3 }
```

compareTo 方法内部直接调用 compare 方法：

```
1 public static int compare(int x, int y) {
2     return (x < y) ? -1 : ((x == y) ? 0 : 1);
3 }
```

如果  $x < y$  返回 -1

如果  $x == y$  返回 0

如果  $x > y$  返回 1

```
1 System.out.println(Integer.compare(1, 2)); //-1
2 System.out.println(Integer.compare(1, 1)); //0
3 System.out.println(Integer.compare(1, 0)); //1
```

那么基本上 Integer 类的主要方法就介绍这么多了，后面如果有比较重要的，会再进行更新。

参考文档：

<https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html>

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别刷选的书籍资料



微信搜一搜

IT可乐

## JDK源码解析(3)——java.lang.String 类

String 类也是java.lang 包下的一个类，算是日常编码中最常用的一个类了，那么本篇博客就来详细的介绍 String 类。

### 1、String 类的定义

```
1 public final class String
2     implements java.io.Serializable, Comparable<String>, CharSequence {}
```

和上一篇博客所讲的 Integer 类一样，这也是一个用 final 声明的常量类，不能被任何类所继承，而且一旦一个String对象被创建，包含在这个对象中的字符序列是不可改变的，包括该类后续的所有方法都是不能修改该对象的，直至该对象被销毁，这是我们需要特别注意的（该类的一些方法看似改变了字符串，其实内部都是创建一个新的字符串，下面讲解方法时会介绍）。接着实现了 Serializable接口，这是一个序列化标志接口，还实现了 Comparable 接口，用于比较两个字符串的大小（按顺序比较单个字符的ASCII码），后面会有具体方法实现；最后实现了 CharSequence 接口，表示是一个有序字符的集合，相应的方法后面也会介绍。

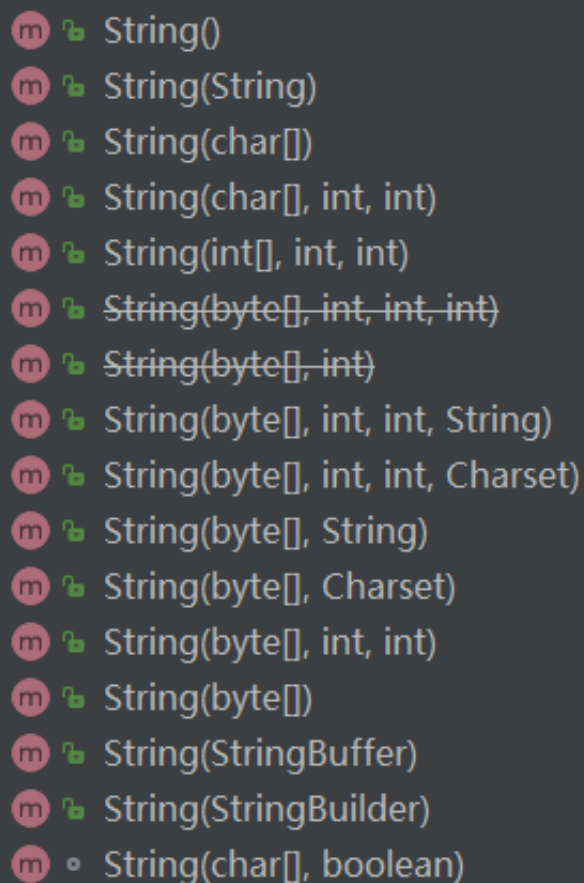
## 2、字段属性

```
1  /**用来存储字符串 */
2  private final char value[];
3
4  /** 缓存字符串的哈希码 */
5  private int hash; // Default to 0
6
7  /** 实现序列化的标识 */
8  private static final long serialVersionUID = -6849794470754667710L;
```

一个 String 字符串实际上是一个 char 数组。

## 3、构造方法

String 类的构造方法很多。可以通过初始化一个字符串，或者字符数组，或者字节数组等等来创建一个 String 对象。



- String()
- String(String)
- String(char[])
- String(char[], int, int)
- String(int[], int, int)
- String(byte[], int, int, int)
- String(byte[], int)
- String(byte[], int, int, String)
- String(byte[], int, int, Charset)
- String(byte[], String)
- String(byte[], Charset)
- String(byte[], int, int)
- String(byte[])
- String(StringBuffer)
- String(StringBuilder)
- String(char[], boolean)

```
1  String str1 = "abc"; //注意这种字面量声明的区别，文末会详细介绍
2  String str2 = new String("abc");
3  String str3 = new String(new char[]{'a', 'b', 'c'});
```

## 4、equals(Object anObject) 方法

```
1 public boolean equals(Object anObject) {
2     if (this == anObject) {
3         return true;
4     }
5     if (anObject instanceof String) {
6         String anotherString = (String)anObject;
7         int n = value.length;
8         if (n == anotherString.value.length) {
9             char v1[] = value;
10            char v2[] = anotherString.value;
11            int i = 0;
12            while (n-- != 0) {
13                if (v1[i] != v2[i])
14                    return false;
15                i++;
16            }
17            return true;
18        }
19    }
20    return false;
21 }
```

String 类重写了 equals 方法，比较的是组成字符串的每一个字符是否相同，如果都相同则返回 true，否则返回 false。

## 5、hashCode() 方法

```
1 public int hashCode() {
2     int h = hash;
3     if (h == 0 && value.length > 0) {
4         char val[] = value;
5
6         for (int i = 0; i < value.length; i++) {
7             h = 31 * h + val[i];
8         }
9         hash = h;
10    }
11    return h;
12 }
```

String 类的 hashCode 算法很简单，主要就是中间的 for 循环，计算公式如下：

$$s[0]31^{(n-1)} + s[1]31^{(n-2)} + \dots + s[n-1]$$

s 数组即源码中的 val 数组，也就是构成字符串的字符数组。这里有个数字 31，为什么选择 31 作为乘积因子，而且没有用一个常量来声明？主要原因有两个：

- ①、31是一个不大不小的质数，是作为 hashCode 乘子的优选质数之一。
- ②、31可以被JVM优化， $31 * i = (i < 5) - i$ 。因为移位运算比乘法运行更快更省性能。

## 6、charAt(int index) 方法

```
1 public char charAt(int index) {
2     //如果传入的索引大于字符串的长度或者小于0，直接抛出索引越界异常
3     if ((index < 0) || (index >= value.length)) {
4         throw new StringIndexOutOfBoundsException(index);
5     }
6     return value[index]; //返回指定索引的单个字符
7 }
```

我们知道一个字符串是由一个字符数组组成，这个方法是通过传入的索引（数组下标），返回指定索引的单个字符。

## 7、compareTo(String anotherString) 和 compareToIgnoreCase(String str) 方法

我们先看看 compareTo 方法：

```
1 public int compareTo(String anotherString) {
2     int len1 = value.length;
3     int len2 = anotherString.value.length;
4     int lim = Math.min(len1, len2);
5     char v1[] = value;
6     char v2[] = anotherString.value;
7
8     int k = 0;
9     while (k < lim) {
10         char c1 = v1[k];
11         char c2 = v2[k];
12         if (c1 != c2) {
13             return c1 - c2;
14         }
15         k++;
16     }
17     return len1 - len2;
18 }
```

源码也很好理解，该方法是按字母顺序比较两个字符串，是基于字符串中每个字符的 Unicode 值。当两个字符串某个位置的字符不同时，返回的是这一位置的字符 Unicode 值之差，当两个字符串都相同时，返回两个字符串长度之差。

compareToIgnoreCase() 方法在 compareTo 方法的基础上忽略大小写，我们知道大写字母是比小写字母的Unicode值小32的，底层实现是先都转换成大写比较，然后都转换成小写进行比较。

## 8、concat(String str) 方法

该方法是将指定的字符串连接到此字符串的末尾。

```
1 public String concat(String str) {
2     int otherLen = str.length();
3     if (otherLen == 0) {
4         return this;
5     }
6     int len = value.length;
7     char buf[] = Arrays.copyOf(value, len + otherLen);
8     str.getChars(buf, len);
9     return new String(buf, true);
10 }
```

首先判断要拼接的字符串长度是否为0，如果为0，则直接返回原字符串。如果不为0，则通过 Arrays 工具类（后面会详细介绍这个工具类）的 copyOf 方法创建一个新的字符数组，长度为原字符串和要拼接的字符串之和，前面填充原字符串，后面为空。接着在通过 getChars 方法将要拼接的字符串放入新字符串后面为空的位置。

注意：返回值是 new String(buf, true)，也就是重新通过 new 关键字创建了一个新的字符串，原字符串是不变的。这也是前面我们说的一旦一个String对象被创建，包含在这个对象中的字符序列是不可改变的。

## 9、indexOf(int ch) 和 indexOf(int ch, int fromIndex) 方法

indexOf(int ch)，参数 ch 其实是字符的 Unicode 值，这里也可以放单个字符（默认转成int），作用是返回指定字符第一次出现的此字符串中的索引。其内部是调用 indexOf(int ch, int fromIndex)，只不过这里的 fromIndex = 0，因为是从 0 开始搜索；而 indexOf(int ch, int fromIndex) 作用也是返回首次出现的此字符串内的索引，但是从指定索引处开始搜索。

```
1 public int indexOf(int ch) {
2     return indexOf(ch, 0); //从第一个字符开始搜索
3 }
```

```
1 public int indexOf(int ch, int fromIndex) {
2     final int max = value.length; //max等于字符的长度
3     if (fromIndex < 0) { //指定索引的位置如果小于0，默认从 0 开始搜索
4         fromIndex = 0;
5     } else if (fromIndex >= max) {
6         //如果指定索引值大于等于字符的长度（因为是数组，下标最多只能是max-1），直接返回-1
7         return -1;
8     }
9
10    if (ch < Character.MIN_SUPPLEMENTARY_CODE_POINT) { //一个char占用两个字节，如果ch小于2的16次方（65536），绝大多数字符都在此范围内
```

```

11         final char[] value = this.value;
12         for (int i = fromIndex; i < max; i++) {//for循环依次判断字符串每个字符
是否和指定字符相等
13             if (value[i] == ch) {
14                 return i;//存在相等的字符，返回第一次出现该字符的索引位置，并终止循
环
15             }
16         }
17         return -1;//不存在相等的字符，则返回 -1
18     } else {//当字符大于 65536时，处理的少数情况，该方法会首先判断是否是有效字符，然
后依次进行比较
19         return indexOfSupplementary(ch, fromIndex);
20     }
21 }

```

## 10、split(String regex) 和 split(String regex, int limit) 方法

split(String regex) 将该字符串拆分为给定正则表达式的匹配。split(String regex, int limit) 也是一样，不过对于 limit 的取值有三种情况：

①、limit > 0，则pattern（模式）应用n - 1 次

```

1 String str = "a,b,c";
2 String[] c1 = str.split(",", 2);
3 System.out.println(c1.length);//2
4 System.out.println(Arrays.toString(c1));//{"a","b,c"}

```

②、limit = 0，则pattern（模式）应用无限次并且省略末尾的空字符串

```

1 String str2 = "a,b,c,,";
2 String[] c2 = str2.split(",", 0);
3 System.out.println(c2.length);//3
4 System.out.println(Arrays.toString(c2));//{"a","b","c"}

```

③、limit < 0，则pattern（模式）应用无限次

```

1 String str2 = "a,b,c,,";
2 String[] c2 = str2.split(",", -1);
3 System.out.println(c2.length);//5
4 System.out.println(Arrays.toString(c2));//{"a","b","c","",""}

```

下面我们看看底层的源码实现。对于 split(String regex) 没什么好说的，内部调用 split(regex, 0) 方法：

```

1 public String[] split(String regex) {
2     return split(regex, 0);
3 }

```

重点看 split(String regex, int limit) 的方法实现：

```
1 public String[] split(String regex, int limit) {
2     /* 1、单个字符，且不是".$|()[]{^?*\+\\\"其中一个
3      * 2、两个字符，第一个是"\"，第二个大小写字母或者数字
4      */
5     char ch = 0;
6     if (((regex.value.length == 1 &&
7         ".$|()[]{^?*\+\\\".indexOf(ch = regex.charAt(0)) == -1) ||
8         (regex.length() == 2 &&
9         regex.charAt(0) == '\\\" &&
10        ((ch = regex.charAt(1))-'0')|('9'-ch)) < 0 &&
11        ((ch-'a')|('z'-ch)) < 0 &&
12        ((ch-'A')|('Z'-ch)) < 0)) &&
13        (ch < Character.MIN_HIGH_SURROGATE ||
14         ch > Character.MAX_LOW_SURROGATE))
15     {
16         int off = 0;
17         int next = 0;
18         boolean limited = limit > 0; //大于0, limited==true,反之
limited==false
19         ArrayList<String> list = new ArrayList<>();
20         while ((next = indexOf(ch, off)) != -1) {
21             //当参数limit<=0 或者 集合list的长度小于 limit-1
22             if (!limited || list.size() < limit - 1) {
23                 list.add(substring(off, next));
24                 off = next + 1;
25             } else { //判断最后一个list.size() == limit - 1
26                 list.add(substring(off, value.length));
27                 off = value.length;
28                 break;
29             }
30         }
31         //如果没有一个能匹配的，返回一个新的字符串，内容和原来的一样
32         if (off == 0)
33             return new String[]{this};
34
35         // 当 limit<=0 时, limited==false,或者集合的长度 小于 limit是，截取添加剩
下的字符串
36         if (!limited || list.size() < limit)
37             list.add(substring(off, value.length));
38
39         // 当 limit == 0 时，如果末尾添加的元素为空（长度为0），则集合长度不断减1，
直到末尾不为空
40         int resultSize = list.size();
41         if (limit == 0) {
42             while (resultSize > 0 && list.get(resultSize - 1).length() ==
0) {
43                 resultSize--;
```



```

44         }
45     }
46     String[] result = new String[resultSize];
47     return list.subList(0, resultSize).toArray(result);
48 }
49 return Pattern.compile(regex).split(this, limit);
50 }

```

## 11、replace(char oldChar, char newChar) 和 String replaceAll(String regex, String replacement) 方法

①、replace(char oldChar, char newChar)：将原字符串中所有的oldChar字符都替换成newChar字符，返回一个新的字符串。

②、String replaceAll(String regex, String replacement)：将匹配正则表达式regex的匹配项都替换成replacement字符串，返回一个新的字符串。

## 12、substring(int beginIndex) 和 substring(int beginIndex, int endIndex) 方法

①、substring(int beginIndex)：返回一个从索引 beginIndex 开始一直到结尾的子字符串。

```

1 public String substring(int beginIndex) {
2     if (beginIndex < 0) { //如果索引小于0，直接抛出异常
3         throw new StringIndexOutOfBoundsException(beginIndex);
4     }
5     int subLen = value.length - beginIndex; //subLen等于字符串长度减去索引
6     if (subLen < 0) { //如果subLen小于0，也是直接抛出异常
7         throw new StringIndexOutOfBoundsException(subLen);
8     }
9     //1、如果索引值beginIndex == 0，直接返回原字符串
10    //2、如果不等于0，则返回从beginIndex开始，一直到结尾
11    return (beginIndex == 0) ? this : new String(value, beginIndex,
12        subLen);
13 }

```

②、substring(int beginIndex, int endIndex)：返回一个从索引 beginIndex 开始，到 endIndex 结尾的子字符串。

## 13、常量池

在前面讲解构造函数的时候，我们知道最常见的两种声明一个字符串对象的形式有两种：

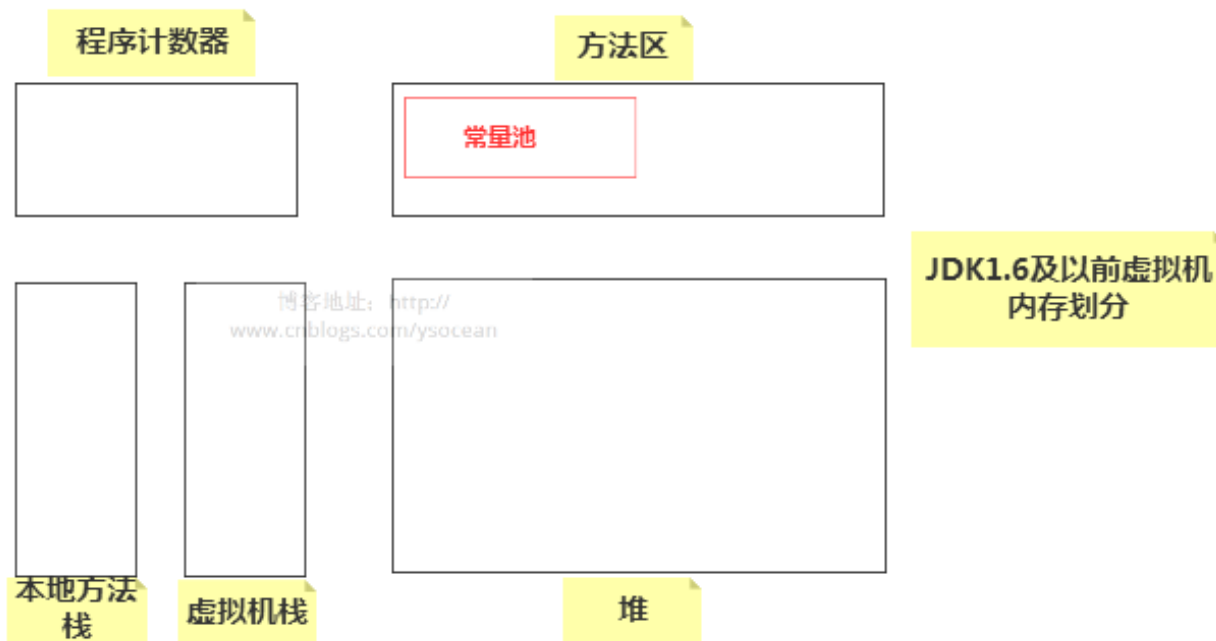
①、通过“字面量”的形式直接赋值

```
String str = "hello";
```

②、通过 new 关键字调用构造函数创建对象

```
String str = new String("hello");
```

那么这两种声明方式有什么区别呢？在讲解之前，我们先介绍JDK1.7（不包括1.7）以前的JVM的内存分布：



①、程序计数器：也称为 PC 寄存器，保存的是程序当前执行的指令的地址（也可以说保存下一条指令的所在存储单元的地址），当CPU需要执行指令时，需要从程序计数器中得到当前需要执行的指令所在存储单元的地址，然后根据得到的地址获取到指令，在得到指令之后，程序计数器便自动加1或者根据转移指针得到下一条指令的地址，如此循环，直至执行完所有的指令。线程私有。

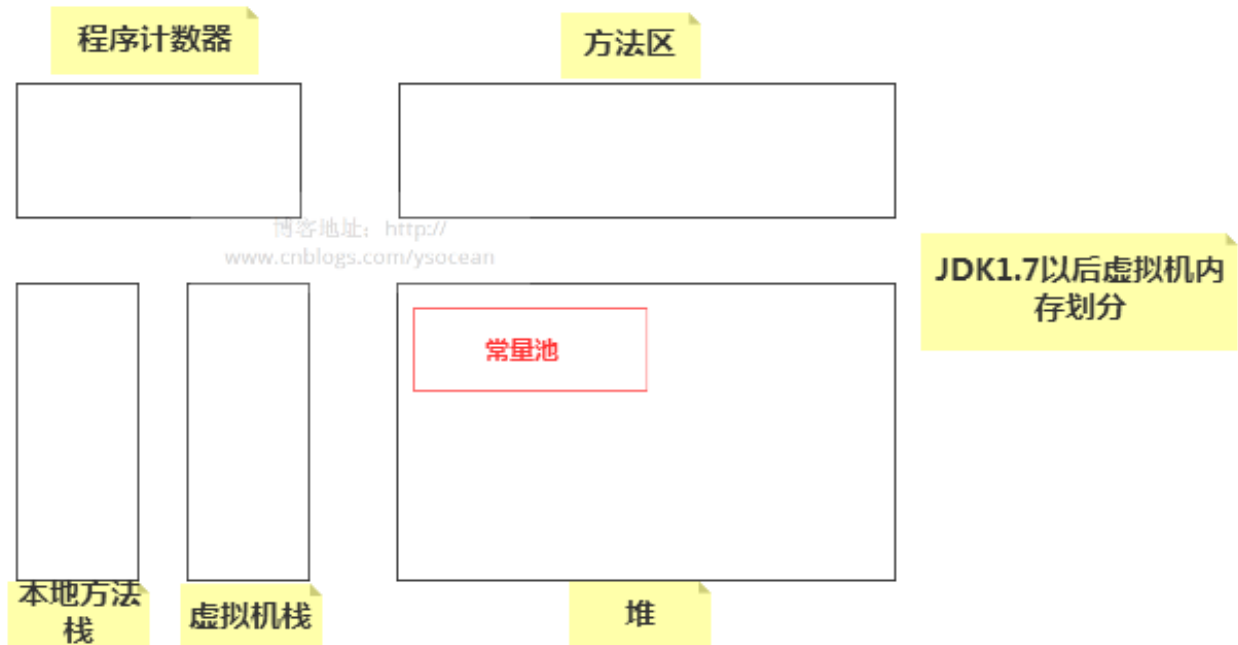
②、虚拟机栈：基本数据类型、对象的引用都存放在这。线程私有。

③、本地方法栈：虚拟机栈是为执行Java方法服务的，而本地方法栈则是为执行本地方法（Native Method）服务的。在JVM规范中，并没有对本地方法栈的具体实现方法以及数据结构作强制规定，虚拟机可以自由实现它。在HotSopt虚拟机中直接就把本地方法栈和虚拟机栈合二为一。

④、方法区：存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。注意：在Class文件中除了类的字段、方法、接口等描述信息外，还有一项信息是常量池，用来存储编译期间生成的字面量和符号引用。

⑤、堆：用来存储对象本身的以及数组（当然，数组引用是存放在Java栈中的）。

在JDK1.7以后，方法区的常量池被移除放到堆中了，如下：



常量池：Java运行时维护一个String Pool（String池），也叫“字符串缓冲区”。String池用来存放运行时中产生的各种字符串，并且池中的字符串的内容不重复。

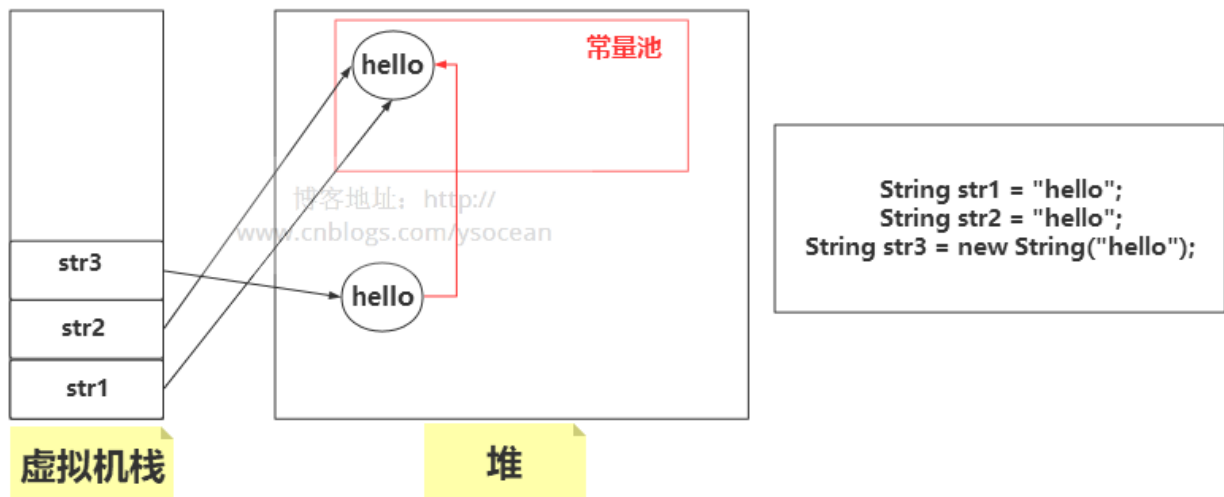
①、字面量创建字符串或者纯字符串（常量）拼接字符串会先在字符串池中找，看是否有相等的对象，没有的话就在字符串池创建该对象；有的话则直接用池中的引用，避免重复创建对象。

②、new关键字创建时，直接在堆中创建一个新对象，变量所引用的都是这个新对象的地址，但是如果通过new关键字创建的字符串内容在常量池中存在了，那么会由堆指向常量池的对应字符；但是反过来，如果通过new关键字创建的字符串对象在常量池中不存在，那么通过new关键词创建的字符串对象是不会额外在常量池中维护的。

③、使用包含变量表达式来创建String对象，则不仅会检查维护字符串池，还会在堆区创建这个对象，最后是指向堆内存的对象。

```
1 String str1 = "hello";
2 String str2 = "hello";
3 String str3 = new String("hello");
4 System.out.println(str1==str2);//true
5 System.out.println(str1==str3);//false
6 System.out.println(str2==str3);//false
7 System.out.println(str1.equals(str2));//true
8 System.out.println(str1.equals(str3));//true
9 System.out.println(str2.equals(str3));//true
```

对于上面的情况，首先 `String str1 = "hello"`，会先到常量池中检查是否有“hello”的存在，发现是没有的，于是在常量池中创建“hello”对象，并将常量池中的引用赋值给str1；第二个字面量 `String str2 = "hello"`，在常量池中检测到该对象了，直接将引用赋值给str2；第三个是通过new关键字创建的对象，常量池中有了该对象了，不用在常量池中创建，然后在堆中创建该对象后，将堆中对象的引用赋值给str3，再将该对象指向常量池。如下图所示：



注意：看上图红色的箭头，通过 new 关键字创建的字符串对象，如果常量池中存在了，会将堆中创建的对象指向常量池的引用。我们可以通过文章末尾介绍的intern()方法来验证。

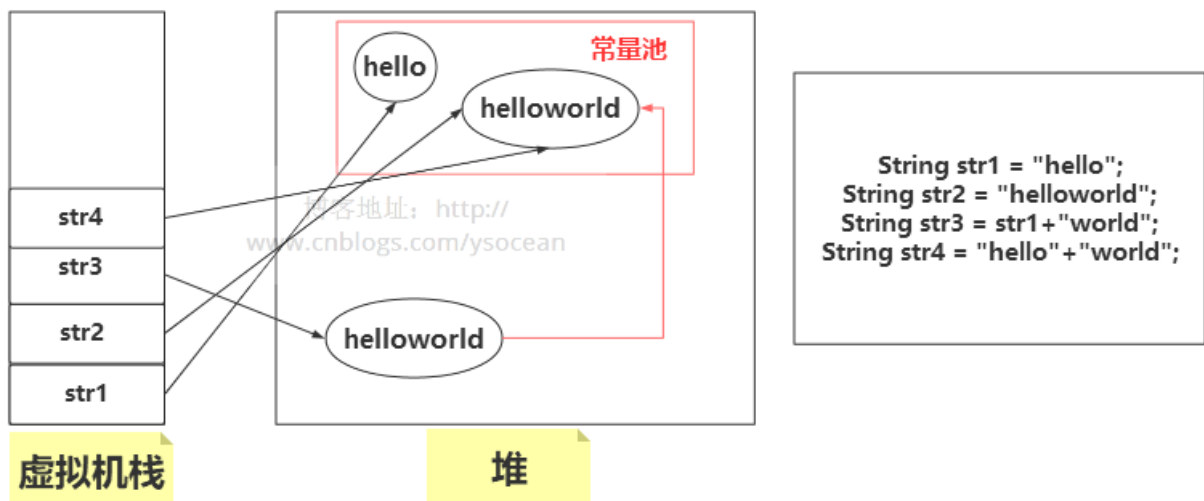
使用包含变量表达式创建对象：

```

1 String str1 = "hello";
2 String str2 = "helloworld";
3 String str3 = str1+"world";//编译器不能确定为常量(会在堆区创建一个String对象)
4 String str4 = "hello"+"world";//编译器确定为常量，直接到常量池中引用
5
6 System.out.println(str2==str3);//false
7 System.out.println(str2==str4);//true
8 System.out.println(str3==str4);//false

```

str3 由于含有变量str1，编译器不能确定是常量，会在堆区中创建一个String对象。而str4是两个常量相加，直接引用常量池中的对象即可。



## 14、intern() 方法

这是一个本地方法：

```
public native String intern();
```

当调用intern方法时，如果池中已经包含一个与该String确定的字符串相同equals(Object)的字符串，则返回该字符串。否则，将此String对象添加到池中，并返回此对象的引用。

这句话什么意思呢？就是说调用一个String对象的intern()方法，如果常量池中有该对象了，直接返回该字符串的引用（存在堆中就返回堆中，存在池中就返回池中），如果没有，则将该对象添加到池中，并返回池中的引用。

```
1 String str1 = "hello";//字面量 只会在常量池中创建对象
2 String str2 = str1.intern();
3 System.out.println(str1==str2);//true
4
5 String str3 = new String("world");//new 关键字只会在堆中创建对象
6 String str4 = str3.intern();
7 System.out.println(str3 == str4);//false
8
9 String str5 = str1 + str2;//变量拼接的字符串，会在常量池中和堆中都创建对象
10 String str6 = str5.intern();//这里由于池中已经有对象了，直接返回的是对象本身，也就是堆中的对象
11 System.out.println(str5 == str6);//true
12
13 String str7 = "hello1" + "world1";//常量拼接的字符串，只会在常量池中创建对象
14 String str8 = str7.intern();
15 System.out.println(str7 == str8);//true
```

## 15、String 真的不可变吗？

前面我们介绍了，String 类是用 final 关键字修饰的，所以我们认为其是不可变对象。但是真的不可变吗？

每个字符串都是由许多单个字符组成的，我们知道其源码是由 char[] value 字符数组构成。

```
1 public final class String
2     implements java.io.Serializable, Comparable<String>, CharSequence {
3     /** The value is used for character storage. */
4     private final char value[];
5
6     /** Cache the hash code for the string */
7     private int hash; // Default to 0
```

value 被 final 修饰，只能保证引用不被改变，但是 value 所指向的堆中的数组，才是真实的数据，只要能够操作堆中的数组，依旧能改变数据。而且 value 是基本类型构成，那么一定是可变的，即使被声明为 private，我们也可以通过反射来改变。

```
1 String str = "vae";
2 //打印原字符串
3 System.out.println(str);//vae
4 //获取String类中的value字段
5 Field fieldStr = String.class.getDeclaredField("value");
6 //因为value是private声明的，这里修改其访问权限
7 fieldStr.setAccessible(true);
8 //获取str对象上的value属性的值
9 char[] value = (char[]) fieldStr.get(str);
10 //将第一个字符修改为 v(小写改大写)
11 value[0] = 'V';
12 //打印修改之后的字符串
13 System.out.println(str);//Vae
```

通过前后两次打印的结果，我们可以看到 String 被改变了，但是在代码里，几乎不会使用反射的机制去操作 String 字符串，所以，我们会认为 String 类型是不可变的。

那么，String 类为什么要这样设计成不可变呢？我们可以从性能以及安全方面来考虑：

## 安全

引发安全问题，譬如，数据库的用户名、密码都是以字符串的形式传入来获得数据库的连接，或者在 socket 编程中，主机名和端口都是以字符串的形式传入。因为字符串是不可变的，所以它的值是不可改变的，否则黑客们可以钻到空子，改变字符串指向的对象的值，造成安全漏洞。

保证线程安全，在并发场景下，多个线程同时读写资源时，会引竞态条件，由于 String 是不可变的，不会引发线程的问题而保证了线程。

HashCode，当 String 被创建出来的时候，hashcode 也会随之被缓存，hashcode 的计算与 value 有关，若 String 可变，那么 hashcode 也会随之变化，针对于 Map、Set 等容器，他们的键值需要保证唯一性和一致性，因此，String 的不可变性使其比其他对象更适合当容器的键值。

## 性能

当字符串是不可变时，字符串常量池才有意义。字符串常量池的出现，可以减少创建相同字面量的字符串，让不同的引用指向池中同一个字符串，为运行时节约很多的堆内存。若字符串可变，字符串常量池失去意义，基于常量池的 String.intern() 方法也失效，每次创建新的 String 将在堆内开辟出新的空间，占据更多的内存

参考文档：

<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

<https://segmentfault.com/a/1190000009914328>

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



微信搜一搜

IT可乐

## JDK源码解析(4)——java.util.Arrays 类

java.util.Arrays 类是 JDK 提供的一个工具类，用来处理数组的各种方法，而且每个方法基本上都是静态方法，能直接通过类名 Arrays 调用。

### 1、asList

```
1 public static <T> List<T> asList(T... a) {  
2     return new ArrayList<>(a);  
3 }
```

作用是返回由指定数组支持的固定大小列表。

**注意：**这个方法返回的 ArrayList 不是我们常用的集合类 java.util.ArrayList。这里的 ArrayList 是 Arrays 的一个内部类 java.util.Arrays.ArrayList。这个内部类有如下属性和方法：

```
ArrayList<E>  
  serialVersionUID : long  
  a : E[]  
  ArrayList(E[])  
  size() : int  
  toArray() : Object[]  
  toArray(T[]) <T> : T[]  
  get(int) : E  
  set(int, E) : E  
  indexOf(Object) : int  
  contains(Object) : boolean
```

```
1 private static class ArrayList<E> extends AbstractList<E>  
2     implements RandomAccess, java.io.Serializable  
3 {
```

```
4         private static final long serialVersionUID =
-2764017481108945198L;
5         private final E[] a;
6
7         ArrayList(E[] array) {
8             if (array==null)
9                 throw new NullPointerException();
10            a = array;
11        }
12
13        public int size() {
14            return a.length;
15        }
16
17        public Object[] toArray() {
18            return a.clone();
19        }
20
21        public <T> T[] toArray(T[] a) {
22            int size = size();
23            if (a.length < size)
24                return Arrays.copyOf(this.a, size,
25                                     (Class<? extends T[]>) a.getClass());
26            System.arraycopy(this.a, 0, a, 0, size);
27            if (a.length > size)
28                a[size] = null;
29            return a;
30        }
31
32        public E get(int index) {
33            return a[index];
34        }
35
36        public E set(int index, E element) {
37            E oldValue = a[index];
38            a[index] = element;
39            return oldValue;
40        }
41
42        public int indexOf(Object o) {
43            if (o==null) {
44                for (int i=0; i<a.length; i++)
45                    if (a[i]==null)
46                        return i;
47            } else {
48                for (int i=0; i<a.length; i++)
49                    if (o.equals(a[i]))
50                        return i;
51            }
```



```

52         return -1;
53     }
54
55     public boolean contains(Object o) {
56         return indexOf(o) != -1;
57     }
58 }

```

①、返回的 **ArrayList** 数组是一个定长列表，我们只能对其进行查看或者修改，但是不能进行添加或者删除操作

通过源码我们发现该类是没有 `add()` 或者 `remove()` 这样的方法的，如果对其进行增加或者删除操作，都会调用其父类 `AbstractList` 对应的方法，而追溯父类的方法最终会抛出 `UnsupportedOperationException` 异常。如下：

```

1  String[] str = {"a", "b", "c"};
2  List<String> listStr = Arrays.asList(str);
3  listStr.set(1, "e");//可以进行修改
4  System.out.println(listStr.toString());//[a, e, c]
5  listStr.add("a");//添加元素会报错 java.lang.UnsupportedOperationException

```

Failures: 0

Failure Trace

```

java.lang.UnsupportedOperationException
at java.util.AbstractList.add(AbstractList.java:148)
at java.util.AbstractList.add(AbstractList.java:108)
at com.js.test.JDKTest.testArrays(JDKTest.java:55)

```

②、引用类型的数组和基本类型的数组区别

```

1  String[] str = {"a", "b", "c"};
2  List listStr = Arrays.asList(str);
3  System.out.println(listStr.size());//3
4
5  int[] i = {1,2,3};
6  List listI = Arrays.asList(i);
7  System.out.println(listI.size());//1

```

上面的结果第一个 `listStr.size()==3`，而第二个 `listI.size()==1`。这是为什么呢？

我们看源码，在 `Arrays.asList` 中，方法声明为 `List asList(T... a)`。该方法接收一个可变参数，并且这个可变参数类型是作为泛型的参数。我们知道基本数据类型是不能作为泛型的参数的，但是数组是引用类型，所以数组是可以泛型化的，于是 `int[]` 作为了整个参数类型，而不是 `int` 作为参数类型。

所以将上面的方法泛型化补全应该是：

```

1 String[] str = {"a", "b", "c"};
2 List<String> listStr = Arrays.asList(str);
3 System.out.println(listStr.size()); //3
4
5 int[] i = {1, 2, 3};
6 List<int[]> listI = Arrays.asList(i); //注意这里List参数为 int[] , 而不是 int
7 System.out.println(listI.size()); //1
8
9 Integer[] in = {1, 2, 3};
10 List<Integer> listIn = Arrays.asList(in); //这里参数为int的包装类Integer, 所以集合长度为3
11 System.out.println(listIn.size()); //3

```

### ③、返回的列表ArrayList里面的元素都是引用，不是独立出来的对象

```

1 String[] str = {"a", "b", "c"};
2 List<String> listStr = Arrays.asList(str);
3 //执行更新操作前
4 System.out.println(Arrays.toString(str)); // [a, b, c]
5 listStr.set(0, "d"); //将第一个元素a改为d
6 //执行更新操作后
7 System.out.println(Arrays.toString(str)); // [d, b, c]

```

这里的Arrays.toString()方法就是打印数组的内容，后面会介绍。我们看修改集合的内容，原数组的内容也变化了，所以这里传入的是引用类型。

### ④、已知数组数据，如何快速获取一个可进行增删改查的列表List?

```

1 String[] str = {"a", "b", "c"};
2 List<String> listStr = new ArrayList<>(Arrays.asList(str));
3 listStr.add("d");
4 System.out.println(listStr.size()); //4

```

这里的ArrayList 集合类后面我们会详细讲解，大家目前只需要知道有这种用法即可。

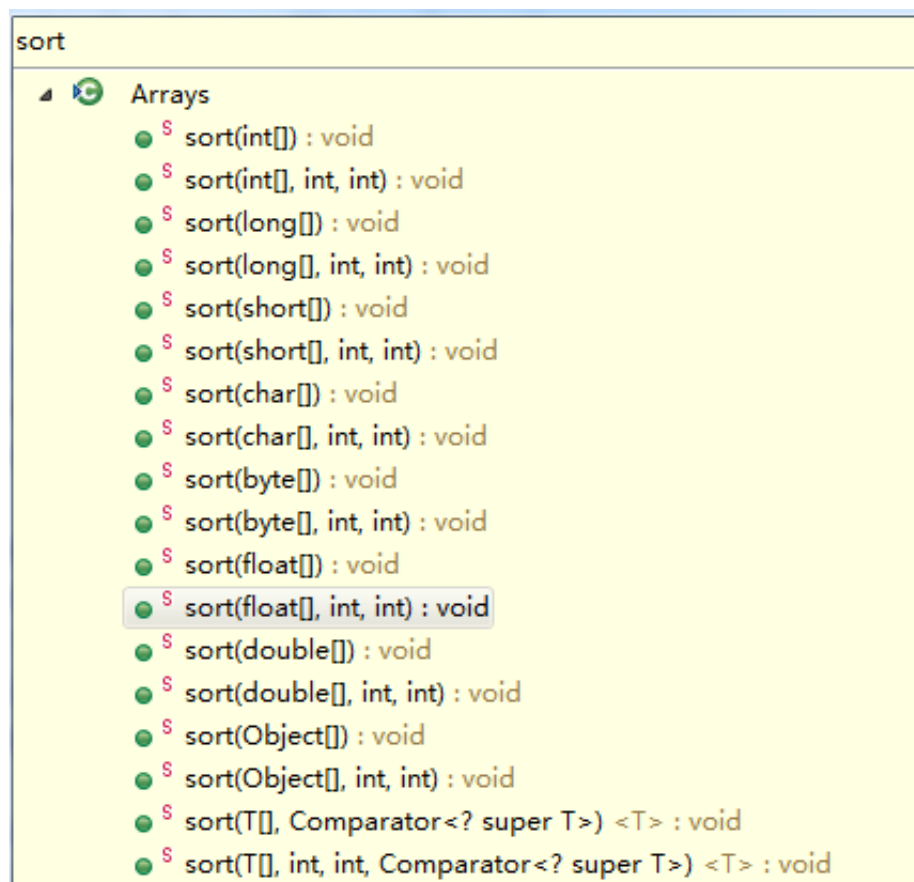
### ⑤、Arrays.asList() 方法使用场景

Arrays工具类提供了一个方法asList, 使用该方法可以将一个变长参数或者数组转换成List。但是，生成的List的长度是固定的；能够进行修改操作（比如，修改某个位置的元素）；不能执行影响长度的操作（如add、remove等操作），否则会抛出UnsupportedOperationException异常。

所以 Arrays.asList 比较适合那些已经有数组数据或者一些元素，而需要快速构建一个List，只用于读取操作，而不进行添加或删除操作的场景。

## 2、sort

该方法用于数组排序，在 Arrays 类中有该方法的一系列重载方法，能对7种基本数据类型，包括 byte,char,double,float,int,long,short 等都能进行排序，还有 Object 类型（实现了 Comparable 接口），以及比较器 Comparator。



### ①、基本类型的数组

这里我们以 int[] 为例看看：

```
1  int[] num = {1,3,8,5,2,4,6,7};
2  Arrays.sort(num);
3  System.out.println(Arrays.toString(num));//[1, 2, 3, 4, 5, 6, 7, 8]
```

通过调用 sort(int[] a) 方法，将原数组按照升序的顺序排列。下面我们通过源码看看是如何实现排序的：

```
1  public static void sort(int[] a) {
2      DualPivotQuicksort.sort(a, 0, a.length - 1, null, 0, 0);
3  }
```

在 Arrays.sort 方法内部调用 DualPivotQuicksort.sort 方法，这个方法的源码很长，分别对于数组的长度进行了各种算法的划分，包括快速排序，插入排序，冒泡排序都有使用。详细源码可以参考这篇博客。

### ②、对象类型数组

该类型的数组进行排序可以实现 Comparable 接口，重写 compareTo 方法进行排序。

```
1 String[] str = {"a","f","c","d"};
2 Arrays.sort(str);
3 System.out.println(Arrays.toString(str));//[a, c, d, f]
```

String 类型实现了 Comparable 接口，内部的 compareTo 方法是按照字典码进行比较的。

### ③、没有实现Comparable接口的，可以通过Comparator实现排序

```
1 Person[] p = new Person[] {new Person("zhangsan",22),new
  Person("wangwu",11),new Person("lisi",33)};
2 Arrays.sort(p,new Comparator<Person>() {
3     @Override
4     public int compare(Person o1, Person o2) {
5         if(o1 == null || o2 == null){
6             return 0;
7         }
8         return o1.getPage()-o2.getPage();
9     }
10 });
11 System.out.println(Arrays.toString(p));
```

## 3、binarySearch

用二分法查找数组中的某个元素。该方法和 sort 方法一样，适用于各种基本数据类型以及对象。

注意：二分法是对以及有序的数组进行查找（比如先用Arrays.sort()进行排序，然后调用此方法进行查找）。找到元素返回下标，没有则返回 -1

实例：

```
1 int[] num = {1,3,8,5,2,4,6,7};
2 Arrays.sort(num);
3 System.out.println(Arrays.toString(num));//[1, 2, 3, 4, 5, 6, 7, 8]
4 System.out.println(Arrays.binarySearch(num, 2));//返回元素的下标 1
```

具体源码实现：

```
1 public static int binarySearch(int[] a, int key) {
2     return binarySearch0(a, 0, a.length, key);
3 }
4 private static int binarySearch0(int[] a, int fromIndex, int
  toIndex,int key) {
5     int low = fromIndex;
6     int high = toIndex - 1;
7
8     while (low <= high) {
```

```

9         int mid = (low + high) >>> 1; //取中间值下标
10        int midVal = a[mid]; //取中间值
11
12        if (midVal < key)
13            low = mid + 1;
14        else if (midVal > key)
15            high = mid - 1;
16        else
17            return mid;
18    }
19    return -(low + 1);
20 }

```

## 4、copyOf

拷贝数组元素。底层采用 System.arraycopy() 实现，这是一个native方法。

```

1    public static native void arraycopy(Object src,  int  srcPos,
2                                         Object dest, int destPos,
3                                         int length);

```

src:源数组

srcPos:源数组要复制的起始位置

dest:目的数组

destPos:目的数组放置的起始位置

length:复制的长度

注意：src 和 dest都必须是同类型或者可以进行转换类型的数组。

```

1    int[] num1 = {1,2,3};
2    int[] num2 = new int[3];
3    System.arraycopy(num1, 0, num2, 0, num1.length);
4    System.out.println(Arrays.toString(num2)); //[1, 2, 3]

```

```

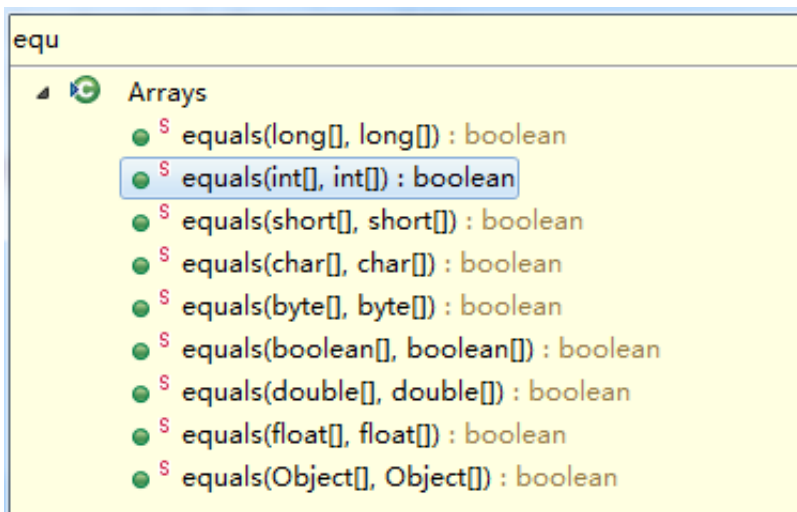
1    /**
2     * @param original 源数组
3     * @param newLength //返回新数组的长度
4     * @return
5     */
6    public static int[] copyOf(int[] original, int newLength) {
7        int[] copy = new int[newLength];
8        System.arraycopy(original, 0, copy, 0,
9                          Math.min(original.length, newLength));
10       return copy;
11    }

```

## 5、equals 和 deepEquals

### ①、equals

equals 用来比较两个数组中对应位置的每个元素是否相等。



八种基本数据类型以及对象都能进行比较。

我们先看看 int 类型的数组比较源码实现：

```
1 public static boolean equals(int[] a, int[] a2) {
2     if (a==a2)//数组引用相等，则里面的元素一定相等
3         return true;
4     if (a==null || a2==null)//两个数组其中一个为null，都返回false
5         return false;
6
7     int length = a.length;
8     if (a2.length != length)//两个数组长度不等，返回false
9         return false;
10
11     for (int i=0; i<length; i++)//通过for循环依次比较数组中每个元素是否相等
12         if (a[i] != a2[i])
13             return false;
14
15     return true;
16 }
```

在看对象数组的比较：

```
1 public static boolean equals(Object[] a, Object[] a2) {
2     if (a==a2)
3         return true;
4     if (a==null || a2==null)
5         return false;
6
7     int length = a.length;
```

```

8         if (a2.length != length)
9             return false;
10
11        for (int i=0; i<length; i++) {
12            Object o1 = a[i];
13            Object o2 = a2[i];
14            if (!(o1==null ? o2==null : o1.equals(o2)))
15                return false;
16        }
17
18        return true;
19    }

```

基本上也是通过 equals 来判断。

## ②、deepEquals

也是用来比较两个数组的元素是否相等，不过 deepEquals 能够进行比较多维数组，而且是任意层次的嵌套数组。

```

1        String[][] name1 = {{ "G","a","o" },{ "H","u","a","n"},{
    "j","i","e"}};
2        String[][] name2 = {{ "G","a","o" },{ "H","u","a","n"},{
    "j","i","e"}};
3        System.out.println(Arrays.equals(name1,name2)); // false
4        System.out.println(Arrays.deepEquals(name1,name2)); // true

```

## 6、fill

该系列方法用于给数组赋值，并能指定某个范围赋值。

```

1        //给a数组所有元素赋值 val
2        public static void fill(int[] a, int val) {
3            for (int i = 0, len = a.length; i < len; i++)
4                a[i] = val;
5        }
6
7        //给从 fromIndex 开始的下标, toIndex-1结尾的下标都赋值 val,左闭右开
8        public static void fill(int[] a, int fromIndex, int toIndex, int val)
9        {
10            rangeCheck(a.length, fromIndex, toIndex); //判断范围是否合理
11            for (int i = fromIndex; i < toIndex; i++)
12                a[i] = val;
13        }

```

## 7、toString 和 deepToString

toString 用来打印一维数组的元素，而 deepToString 用来打印多层次嵌套的数组元素。

```
1 public static String toString(int[] a) {
2     if (a == null)
3         return "null";
4     int iMax = a.length - 1;
5     if (iMax == -1)
6         return "[]";
7
8     StringBuilder b = new StringBuilder();
9     b.append('[');
10    for (int i = 0; ; i++) {
11        b.append(a[i]);
12        if (i == iMax)
13            return b.append(']').toString();
14        b.append(", ");
15    }
16 }
```

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



微信搜一搜

IT可乐

## JDK源码解析(5)——java.util.ArrayList 类

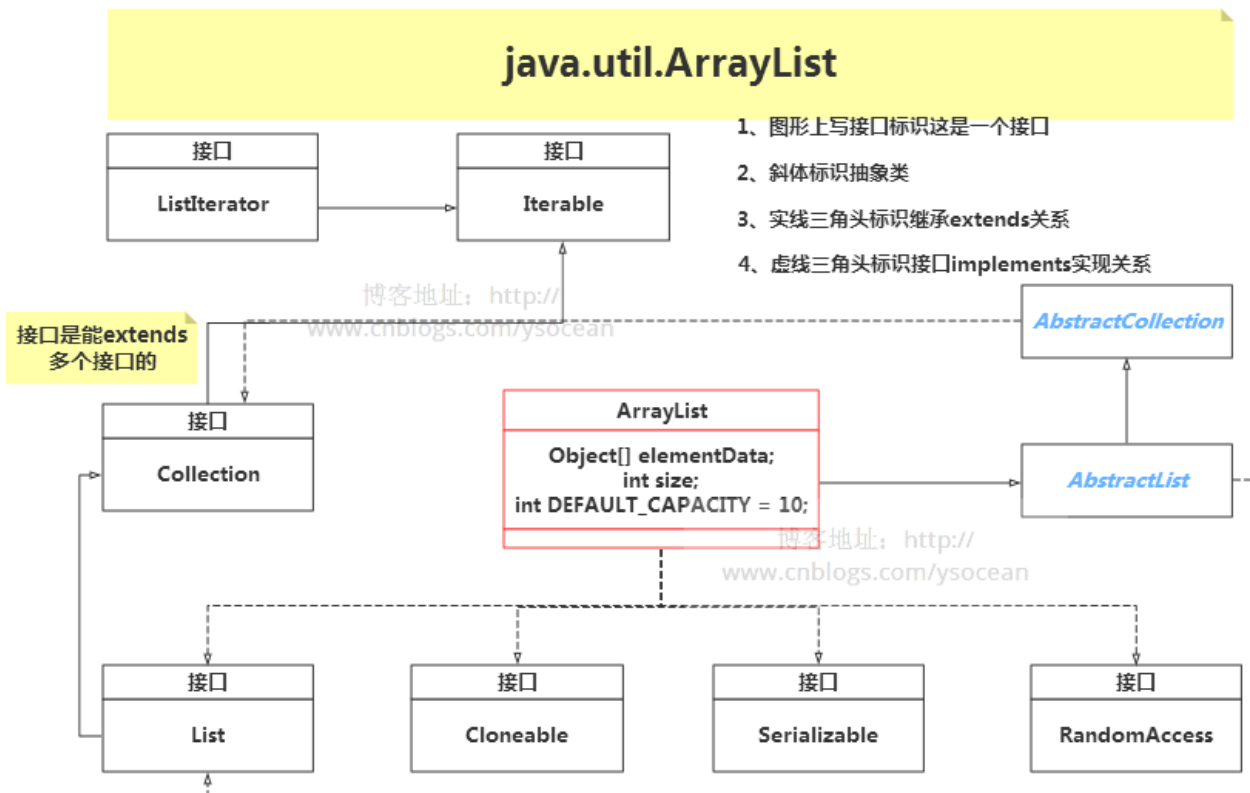
搞Java的，对于集合框架应该再熟悉不过了，前面几篇文章我们介绍了java.lang包下的几种基本数据类型，接下来我们将介绍java.util包下的几个集合类，首先介绍的是 ArrayList 类。



# 1、ArrayList 定义

ArrayList 是一个用数组实现的集合，支持随机访问，元素有序且可以重复。

```
1 public class ArrayList<E> extends AbstractList<E>
2     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```



## ①、实现 RandomAccess 接口

这是一个标记接口，一般此标记接口用于 List 实现，以表明它们支持快速（通常是恒定时间）的随机访问。该接口的主要目的是允许通用算法改变其行为，以便在应用于随机或顺序访问列表时提供良好的性能。

比如在工具类 Collections(这个工具类后面会详细讲解)中，应用二分查找方法时判断是否实现了 RandomAccess 接口：

```
1 int binarySearch(List<? extends Comparable<? super T>> list, T key) {
2     if (list instanceof RandomAccess || list.size()
3         < BINARYSEARCH_THRESHOLD)
4         return Collections.indexedBinarySearch(list, key);
5     else
6         return Collections.iteratorBinarySearch(list, key);
7 }
```

## ②、实现 Cloneable 接口

这个类是 `java.lang.Cloneable`，前面我们讲解深拷贝和浅拷贝的原理时，我们介绍了浅拷贝可以通过调用 `Object.clone()` 方法来实现，但是调用该方法的对象必须要实现 `Cloneable` 接口，否则会抛出 `CloneNotSupportedException` 异常。

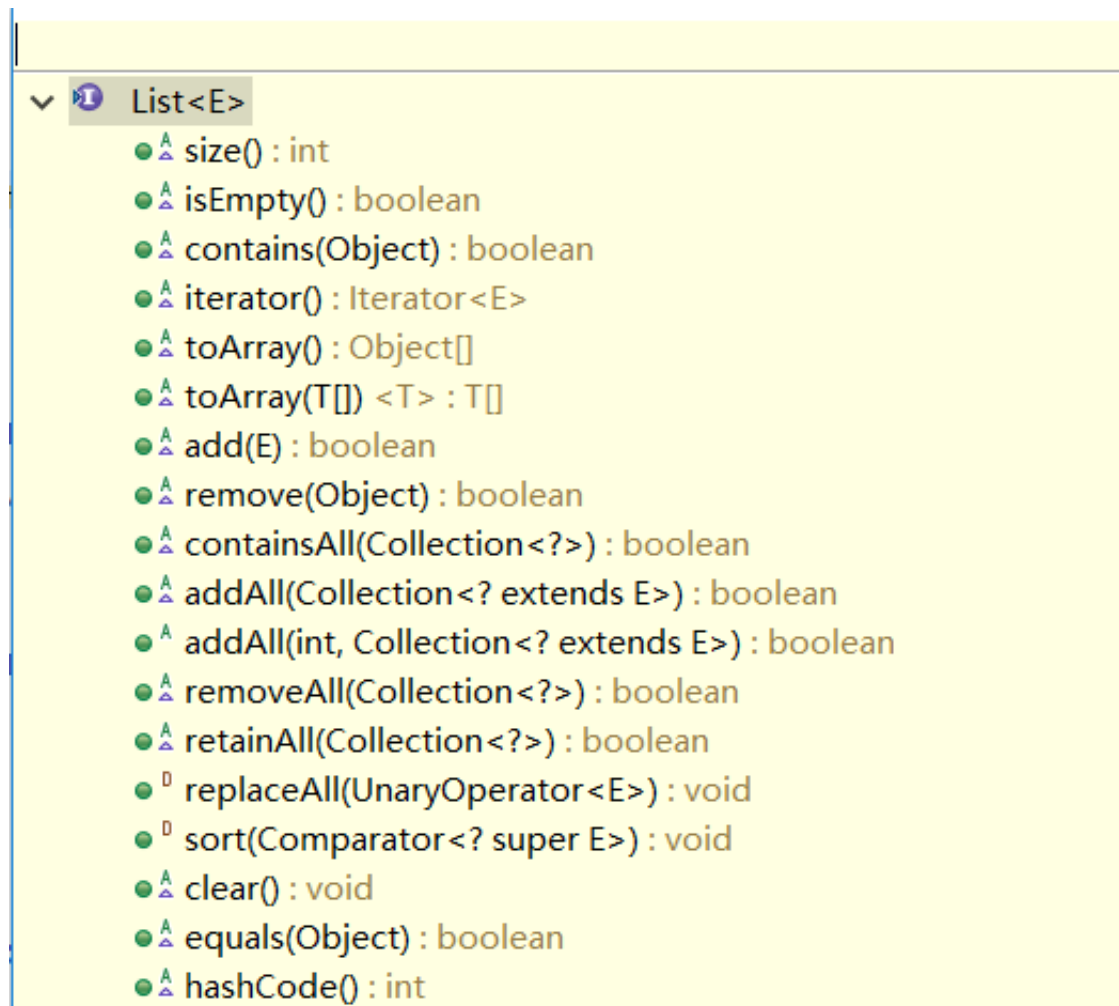
Cloneable 和 RandomAccess 接口一样也是一个标记接口，接口内无任何方法体和常量的声明，也就是说如果想克隆对象，必须要实现 Cloneable 接口，表明该类是可以被克隆的。

### ③、实现 Serializable 接口

这个没什么好说的，也是标记接口，表示能被序列化。

### ④、实现 List 接口

这个接口是 List 类集合的上层接口，定义了实现该接口的类都必须要实现的一组方法，如下所示，下面我们会对这一系列方法的实现做详细介绍。



## 2、字段属性

```

1 //集合的默认大小
2 private static final int DEFAULT_CAPACITY = 10;
3 //空的数组实例
4 private static final Object[] EMPTY_ELEMENTDATA = {};
5 //这也是一个空的数组实例，和EMPTY_ELEMENTDATA空数组相比是用于了解添加元素时数组膨胀多少
6 private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
7 //存储 ArrayList集合的元素，集合的长度即这个数组的长度
8 //1、当 elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA 时将会清空 ArrayList
9 //2、当添加第一个元素时，elementData 长度会扩展为 DEFAULT_CAPACITY=10
10 transient Object[] elementData;
11 //表示集合的长度
12 private int size;

```

### 3、构造函数

```

1 public ArrayList() {
2     this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
3 }

```

此无参构造函数将创建一个 DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA 声明的数组，注意此时初始容量是0，而不是大家以为的 10。

**注意：**根据默认构造函数创建的集合，ArrayList list = new ArrayList();此时集合长度是0。

```

1 public ArrayList(int initialCapacity) {
2     if (initialCapacity > 0) {
3         this.elementData = new Object[initialCapacity];
4     } else if (initialCapacity == 0) {
5         this.elementData = EMPTY_ELEMENTDATA;
6     } else {
7         throw new IllegalArgumentException("Illegal Capacity: "+
8                                         initialCapacity);
9     }
10 }

```

初始化集合大小创建 ArrayList 集合。当大于0时，给定多少那就创建多大的数组；当等于0时，创建一个空数组；当小于0时，抛出异常。

```

1 public ArrayList(Collection<? extends E> c) {
2     elementData = c.toArray();
3     if ((size = elementData.length) != 0) {
4         // c.toArray might (incorrectly) not return Object[] (see
6260652)
5         if (elementData.getClass() != Object[].class)
6             elementData = Arrays.copyOf(elementData, size,
Object[].class);
7     } else {
8         // replace with empty array.
9         this.elementData = EMPTY_ELEMENTDATA;
10    }
11 }

```

这是将已有的集合复制到 ArrayList 集合中去。

## 4、添加元素

通过前面的字段属性和构造函数，我们知道 ArrayList 集合是由数组构成的，那么向 ArrayList 中添加元素，也就是向数组赋值。我们知道一个数组的声明是能确定大小的，而使用 ArrayList 时，好像是能添加任意多个元素，这就涉及到数组的扩容。

扩容的核心方法就是调用前面我们讲过的 Arrays.copyOf 方法，创建一个更大的数组，然后将原数组元素拷贝过去即可。下面我们看看具体实现：

```

1 public boolean add(E e) {
2     ensureCapacityInternal(size + 1); //添加元素之前，首先要确定集合的大小
3     elementData[size++] = e;
4     return true;
5 }

```

如上所示，在通过调用 add 方法添加元素之前，我们要首先调用 ensureCapacityInternal 方法来确定集合的大小，如果集合满了，则要进行扩容操作。

```

1 private void ensureCapacityInternal(int minCapacity) { //这里的minCapacity 是
集合当前大小+1
2     //elementData 是实际用来存储元素的数组，注意数组的大小和集合的大小不是相等
的，前面的size是指集合大小
3     ensureExplicitCapacity(calculateCapacity(elementData,
minCapacity));
4 }
5 private static int calculateCapacity(Object[] elementData, int
minCapacity) {
6     if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) { //如果数组为
空，则从size+1的值和默认值10中取最大的
7         return Math.max(DEFAULT_CAPACITY, minCapacity);
8     }
9     return minCapacity; //不为空，则返回size+1

```

```

10     }
11     private void ensureExplicitCapacity(int minCapacity) {
12         modCount++;
13
14         // overflow-conscious code
15         if (minCapacity - elementData.length > 0)
16             grow(minCapacity);
17     }

```

在 ensureExplicitCapacity 方法中，首先对修改次数 modCount 加一，这里的 modCount 给 ArrayList 的迭代器使用的，在并发操作被修改时，提供快速失败行为（保证 modCount 在迭代期间不变，否则抛出 ConcurrentModificationException 异常，可以查看源码 865 行），接着判断 minCapacity 是否大于当前 ArrayList 内部数组长度，大于的话调用 grow 方法对内部数组 elementData 扩容，grow 方法代码如下：

```

1 private void grow(int minCapacity) {
2     int oldCapacity = elementData.length; // 得到原始数组的长度
3     int newCapacity = oldCapacity + (oldCapacity >> 1); // 新数组的长度等于
    原数组长度的1.5倍
4     if (newCapacity - minCapacity < 0) // 当新数组长度仍然比 minCapacity 小，
    则为保证最小长度，新数组等于 minCapacity
5         newCapacity = minCapacity;
6     // MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8 = 2147483639
7     if (newCapacity - MAX_ARRAY_SIZE > 0) // 当得到的新数组长度比
    MAX_ARRAY_SIZE 大时，调用 hugeCapacity 处理大数组
8         newCapacity = hugeCapacity(minCapacity);
9     // 调用 Arrays.copyOf 将原数组拷贝到一个大小为 newCapacity 的新数组（注意是拷
    贝引用）
10    elementData = Arrays.copyOf(elementData, newCapacity);
11 }
12
13 private static int hugeCapacity(int minCapacity) {
14     if (minCapacity < 0) //
15         throw new OutOfMemoryError();
16     return (minCapacity > MAX_ARRAY_SIZE) ? // minCapacity >
    MAX_ARRAY_SIZE, 则新数组大小为 Integer.MAX_VALUE
17         Integer.MAX_VALUE :
18         MAX_ARRAY_SIZE;
19 }

```

对于 ArrayList 集合添加元素，我们总结一下：

①、当通过 ArrayList() 构造一个空集合，初始长度是为 0 的，第 1 次添加元素，会创建一个长度为 10 的数组，并将该元素赋值到数组的第一个位置。

②、第 2 次添加元素，集合不为空，而且由于集合的长度 size+1 是小于数组的长度 10，所以直接添加元素到数组的第二个位置，不用扩容。

③、第 11 次添加元素，此时  $\text{size}+1 = 11$ ，而数组长度是 10，这时候创建一个长度为  $10+10*0.5 = 15$  的数组（扩容 1.5 倍），然后将原数组元素引用拷贝到新数组。并将第 11 次添加的元素赋值到新数组下标为 10 的位置。

④、第  $\text{Integer.MAX\_VALUE} - 8 = 2147483639$ ，然后  $2147483639*1.5=1431655759$ （这个数是要进行扩容）次添加元素，为了防止溢出，此时会直接创建一个  $1431655759+1$  大小的数组，这样一直，每次添加一个元素，都只扩大一个范围。

⑤、第  $\text{Integer.MAX\_VALUE} - 7$  次添加元素时，创建一个大小为  $\text{Integer.MAX\_VALUE}$  的数组，在进行元素添加。

⑥、第  $\text{Integer.MAX\_VALUE} + 1$  次添加元素时，抛出 `OutOfMemoryError` 异常。

注意：能向集合中添加 `null` 的，因为数组可以有 `null` 值存在。

```
1 Object[] obj = {null,1};
2
3 ArrayList list = new ArrayList();
4 list.add(null);
5 list.add(1);
6 System.out.println(list.size()); //2
```

## 5、删除元素

### ①、根据索引删除元素

```
1 public E remove(int index) {
2     rangeCheck(index); //判断给定索引的范围，超过集合大小则抛出异常
3
4     modCount++;
5     E oldValue = elementData(index); //得到索引处的删除元素
6
7     int numMoved = size - index - 1;
8     if (numMoved > 0) //size-index-1 > 0 表示 0<= index < (size-1),即索引
        不是最后一个元素
9         //通过 System.arraycopy()将数组elementData 的下标index+1之后长度为
        numMoved的元素拷贝到从index开始的位置
10        System.arraycopy(elementData, index+1, elementData, index,
11                           numMoved);
12        elementData[--size] = null; //将数组最后一个元素置为 null，便于垃圾回收
13
14        return oldValue;
15    }
```

`remove(int index)` 方法表示删除索引 `index` 处的元素，首先通过 `rangeCheck(index)` 方法判断给定索引的范围，超过集合大小则抛出异常；接着通过 `System.arraycopy` 方法对数组进行自身拷贝。关于这个方法的用法可以参考这篇博客。

### ②、直接删除指定元素

```

1  public boolean remove(Object o) {
2      if (o == null) { //如果删除的元素为null
3          for (int index = 0; index < size; index++)
4              if (elementData[index] == null) {
5                  fastRemove(index);
6                  return true;
7              }
8      } else { //不为null, 通过equals方法判断对象是否相等
9          for (int index = 0; index < size; index++)
10             if (o.equals(elementData[index])) {
11                 fastRemove(index);
12                 return true;
13             }
14     }
15     return false;
16 }
17
18
19 private void fastRemove(int index) {
20     modCount++;
21     int numMoved = size - index - 1;
22     if (numMoved > 0)
23         System.arraycopy(elementData, index+1, elementData, index,
24                             numMoved);
25     elementData[--size] = null; //
26 }

```

remove(Object o)方法是删除第一次出现的该元素。然后通过System.arraycopy进行数组自身拷贝。

## 6、修改元素

通过调用 set(int index, E element) 方法在指定索引 index 处的元素替换为 element。并返回原数组的元素。

```

1  public E set(int index, E element) {
2      rangeCheck(index); //判断索引合法性
3
4      E oldValue = elementData(index); //获得原数组指定索引的元素
5      elementData[index] = element; //将指定索引处的元素替换为 element
6      return oldValue; //返回原数组索引元素
7  }

```

通过调用 rangeCheck(index) 来检查索引合法性。

```

1     private void rangeCheck(int index) {
2         if (index >= size)
3             throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
4     }

```

当索引为负数时，会抛出 `java.lang.ArrayIndexOutOfBoundsException` 异常。当索引大于集合长度时，会抛出 `IndexOutOfBoundsException` 异常。

## 7、查找元素

### ①、根据索引查找元素

```

1     public E get(int index) {
2         rangeCheck(index);
3
4         return elementData(index);
5     }

```

同理，首先还是判断给定索引的合理性，然后直接返回处于该下标位置的数组元素。

### ②、根据元素查找索引

```

1     public int indexOf(Object o) {
2         if (o == null) {
3             for (int i = 0; i < size; i++)
4                 if (elementData[i]==null)
5                     return i;
6         } else {
7             for (int i = 0; i < size; i++)
8                 if (o.equals(elementData[i]))
9                     return i;
10        }
11        return -1;
12    }

```

**注意：**`indexOf(Object o)` 方法是返回第一次出现该元素的下标，如果没有则返回 -1。

还有 `lastIndexOf(Object o)` 方法是返回最后一次出现该元素的下标。

## 8、遍历集合

### ①、普通 for 循环遍历

前面我们介绍查找元素时，知道可以通过 `get(int index)` 方法，根据索引查找元素，那么遍历同理：



```

1 ArrayList list = new ArrayList();
2 list.add("a");
3 list.add("b");
4 list.add("c");
5 for(int i = 0 ; i < list.size() ; i++){
6     System.out.print(list.get(i)+" ");
7 }

```

## ②、迭代器 iterator

先看看具体用法：

```

1 ArrayList<String> list = new ArrayList<>();
2 list.add("a");
3 list.add("b");
4 list.add("c");
5 Iterator<String> it = list.iterator();
6 while(it.hasNext()){
7     String str = it.next();
8     System.out.print(str+" ");
9 }

```

在介绍 ArrayList 时，我们知道该类实现了 List 接口，而 List 接口又继承了 Collection 接口，Collection 接口又继承了 Iterable 接口，该接口有个 Iterator iterator() 方法，能获取 Iterator 对象，能用该对象进行集合遍历，为什么能用该对象进行集合遍历？我们再看看 ArrayList 类中的该方法实现：

```

1     public Iterator<E> iterator() {
2         return new Itr();
3     }

```

该方法是返回一个 Itr 对象，这个类是 ArrayList 的内部类。

```

1     private class Itr implements Iterator<E> {
2         int cursor;           //游标， 下一个要返回的元素的索引
3         int lastRet = -1; // 返回最后一个元素的索引；如果没有这样的话返回-1.
4         int expectedModCount = modCount;
5
6         //通过 cursor != size 判断是否还有下一个元素
7         public boolean hasNext() {
8             return cursor != size;
9         }
10
11         @SuppressWarnings("unchecked")
12         public E next() {
13             checkForComodification(); //迭代器进行元素迭代时同时进行增加和删除操作，会抛出异常
14             int i = cursor;
15             if (i >= size)

```

```

16         throw new NoSuchElementException();
17         Object[] elementData = ArrayList.this.elementData;
18         if (i >= elementData.length)
19             throw new ConcurrentModificationException();
20         cursor = i + 1; //游标向后移动一位
21         return (E) elementData[lastRet = i]; //返回索引为i处的元素，并将
lastRet赋值为i
22     }
23
24     public void remove() {
25         if (lastRet < 0)
26             throw new IllegalStateException();
27         checkForComodification();
28
29         try {
30             ArrayList.this.remove(lastRet); //调用ArrayList的remove方法删
除元素
31             cursor = lastRet; //游标指向删除元素的位置，本来是lastRet+1的，这
里删除一个元素，然后游标就不变了
32             lastRet = -1; //lastRet恢复默认值-1
33             expectedModCount = modCount; //expectedModCount值和modCount
同步，因为进行add和remove操作，modCount会加1
34         } catch (IndexOutOfBoundsException ex) {
35             throw new ConcurrentModificationException();
36         }
37     }
38
39     @Override
40     @SuppressWarnings("unchecked")
41     public void forEachRemaining(Consumer<? super E> consumer) { //便于
进行forEach循环
42         Objects.requireNonNull(consumer);
43         final int size = ArrayList.this.size;
44         int i = cursor;
45         if (i >= size) {
46             return;
47         }
48         final Object[] elementData = ArrayList.this.elementData;
49         if (i >= elementData.length) {
50             throw new ConcurrentModificationException();
51         }
52         while (i != size && modCount == expectedModCount) {
53             consumer.accept((E) elementData[i++]);
54         }
55         // update once at end of iteration to reduce heap write
traffic
56         cursor = i;
57         lastRet = i - 1;
58         checkForComodification();

```

```

59     }
60
61     //前面在新增元素add() 和 删除元素 remove() 时，我们可以看到 modCount++。修
    改set() 是没有的
62     //也就是说不能在迭代器进行元素迭代时进行增加和删除操作，否则抛出异常
63     final void checkForComodification() {
64         if (modCount != expectedModCount)
65             throw new ConcurrentModificationException();
66     }
67 }

```

注意在进行 next() 方法调用的时候，会进行 checkForComodification() 调用，该方法表示迭代器进行元素迭代时，如果同时进行增加和删除操作，会抛出 ConcurrentModificationException 异常。比如：

```

1  ArrayList<String> list = new ArrayList<>();
2  list.add("a");
3  list.add("b");
4  list.add("c");
5  Iterator<String> it = list.iterator();
6  while(it.hasNext()){
7      String str = it.next();
8      System.out.print(str+" ");
9      list.remove(str);//集合遍历时进行删除或者新增操作，都会抛出
    ConcurrentModificationException 异常
10     //list.add(str);
11     list.set(0, str);//修改操作不会造成异常
12 }

```

Failures: 0

#### Failure Trace

```

java.util.ConcurrentModificationException
at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
at java.util.ArrayList$Itr.next(ArrayList.java:851)
at com.js.test.JDKTest.testArrayList(JDKTest.java:144)

```

解决办法是不调用 ArrayList.remove() 方法，转而调用 迭代器的 remove() 方法：

```

1  Iterator<String> it = list.iterator();
2  while(it.hasNext()){
3      String str = it.next();
4      System.out.print(str+" ");
5      //list.remove(str);//集合遍历时进行删除或者新增操作，都会抛出
    ConcurrentModificationException 异常
6      it.remove();
7  }

```

注意：迭代器只能向后遍历，不能向前遍历，能够删除元素，但是不能新增元素。

### ③、迭代器的变种 forEach

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("a");
3 list.add("b");
4 list.add("c");
5 for(String str : list){
6     System.out.print(str + " ");
7 }
```

这种语法可以看成是 JDK 的一种语法糖，通过反编译 class 文件，我们可以看到生成的 java 文件，其具体实现还是通过调用 Iterator 迭代器进行遍历的。如下：

```
1 ArrayList list = new ArrayList();
2     list.add("a");
3     list.add("b");
4     list.add("c");
5     String str;
6     for (Iterator iterator1 = list.iterator(); iterator1.hasNext();
7         System.out.print((new StringBuilder(String.valueOf(str))).append("
8         ").toString()))
9         str = (String)iterator1.next();
```

### ④、迭代器 ListIterator

还是先看看具体用法：

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("a");
3 list.add("b");
4 list.add("c");
5 ListIterator<String> listIt = list.listIterator();
6
7 //向后遍历
8 while(listIt.hasNext()){
9     System.out.print(listIt.next()+" "); //a b c
10 }
11
12 //向前遍历,此时由于上面进行了向后遍历，游标已经指向了最后一个元素，所以此处向前遍历能
13 //有值
14 while(listIt.hasPrevious()){
15     System.out.print(listIt.previous()+" "); //c b a
16 }
```

还能一边遍历，一边进行新增或者删除操作：

```

1  ArrayList<String> list = new ArrayList<>();
2  list.add("a");
3  list.add("b");
4  list.add("c");
5  ListIterator<String> listIt = list.listIterator();
6
7  //向后遍历
8  while(listIt.hasNext()){
9      System.out.print(listIt.next()+" "); //a b c
10     listIt.add("1");//在每一个元素后面增加一个元素 "1"
11 }
12
13 //向后前遍历,此时由于上面进行了向后遍历, 游标已经指向了最后一个元素, 所以此处向前遍历能有
  值
14 while(listIt.hasPrevious()){
15     System.out.print(listIt.previous()+" "); //1 c 1 b 1 a
16 }

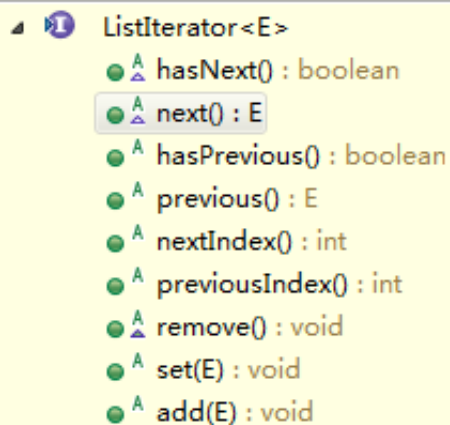
```

也就是说相比于 Iterator 迭代器, 这里的 ListIterator 多出了能向前迭代, 以及能够新增元素。下面我们看看具体实现:

对于 Iterator 迭代器, 我们查看 JDK 源码, 发现还有 ListIterator 接口继承了 Iterator:

```
public interface ListIterator extends Iterator
```

该接口有如下方法:



```

ListIterator<E>
  hasNext() : boolean
  next() : E
  hasPrevious() : boolean
  previous() : E
  nextIndex() : int
  previousIndex() : int
  remove() : void
  set(E) : void
  add(E) : void

```

我们看 ArrayList 类中, 有如下方法可以获得 ListIterator 接口:

```

1  public ListIterator<E> listIterator() {
2      return new ListItr(0);
3  }

```

这里的 ListItr 也是一个内部类。

```

1  //注意 内部类 ListItr 继承了另一个内部类 Itr
2  private class ListItr extends Itr implements ListIterator<E> {

```

```

3      ListItr(int index) { //构造函数，进行游标初始化
4          super();
5          cursor = index;
6      }
7
8      public boolean hasPrevious() { //判断是否有上一个元素
9          return cursor != 0;
10     }
11
12     public int nextIndex() { //返回下一个元素的索引
13         return cursor;
14     }
15
16     public int previousIndex() { //返回上一个元素的索引
17         return cursor - 1;
18     }
19
20     //该方法获取当前索引的上一个元素
21     @SuppressWarnings("unchecked")
22     public E previous() {
23         checkForComodification(); //迭代器进行元素迭代时同时进行增加和删除操作，会抛出异常
24         int i = cursor - 1;
25         if (i < 0)
26             throw new NoSuchElementException();
27         Object[] elementData = ArrayList.this.elementData;
28         if (i >= elementData.length)
29             throw new ConcurrentModificationException();
30         cursor = i; //游标指向上一个元素
31         return (E) elementData[lastRet = i]; //返回上一个元素的值
32     }
33
34
35     public void set(E e) {
36         if (lastRet < 0)
37             throw new IllegalStateException();
38         checkForComodification();
39
40         try {
41             ArrayList.this.set(lastRet, e);
42         } catch (IndexOutOfBoundsException ex) {
43             throw new ConcurrentModificationException();
44         }
45     }
46
47     //相比于迭代器 Iterator，这里多了一个新增操作
48     public void add(E e) {
49         checkForComodification();
50

```

```

51         try {
52             int i = cursor;
53             ArrayList.this.add(i, e);
54             cursor = i + 1;
55             lastRet = -1;
56             expectedModCount = modCount;
57         } catch (IndexOutOfBoundsException ex) {
58             throw new ConcurrentModificationException();
59         }
60     }
61 }

```

## 9、SubList

在 ArrayList 中有这样一个方法：

```

1     public List<E> subList(int fromIndex, int toIndex) {
2         subListRangeCheck(fromIndex, toIndex, size);
3         return new SubList(this, 0, fromIndex, toIndex);
4     }

```

作用是返回从 fromIndex(包括) 开始的下标，到 toIndex(不包括) 结束的下标之间的元素视图。如下：

```

1 ArrayList<String> list = new ArrayList<>();
2 list.add("a");
3 list.add("b");
4 list.add("c");
5
6 List<String> subList = list.subList(0, 1);
7 for(String str : subList){
8     System.out.print(str + " "); //a
9 }

```

这里出现了 SubList 类，这也是 ArrayList 中的一个内部类。

注意：返回的是原集合的视图，也就是说，如果对 subList 出来的集合进行修改或新增操作，那么原始集合也会发生同样的操作。

```

1 ArrayList<String> list = new ArrayList<>();
2 list.add("a");
3 list.add("b");
4 list.add("c");
5
6 List<String> subList = list.subList(0, 1);
7 for(String str : subList){
8     System.out.print(str + " "); //a
9 }
10 subList.add("d");
11 System.out.println(subList.size()); //2
12 System.out.println(list.size()); //4, 原始集合长度也增加了

```

想要独立出来一个集合，解决办法如下：

```
List subList = new ArrayList<>(list.subList(0, 1));
```

## 10、size()

```

1 public int size() {
2     return size;
3 }

```

注意：返回集合的长度，而不是数组的长度，这里的 size 就是定义的全局变量。

## 11、isEmpty()

```

1 public boolean isEmpty() {
2     return size == 0;
3 }

```

返回 size == 0 的结果。

## 12、trimToSize()

```

1 public void trimToSize() {
2     modCount++;
3     if (size < elementData.length) {
4         elementData = Arrays.copyOf(elementData, size);
5     }
6 }

```

该方法用于回收多余的内存。也就是说一旦我们确定集合不在添加多余的元素之后，调用 trimToSize() 方法会将实现集合的数组大小刚好调整为集合元素的大小。

**注意：**该方法会花时间来复制数组元素，所以应该在确定不会添加元素之后在调用。



参考文档：

<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#>

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



微信搜一搜

IT可乐

## JDK源码解析(6)——java.util.LinkedList 类

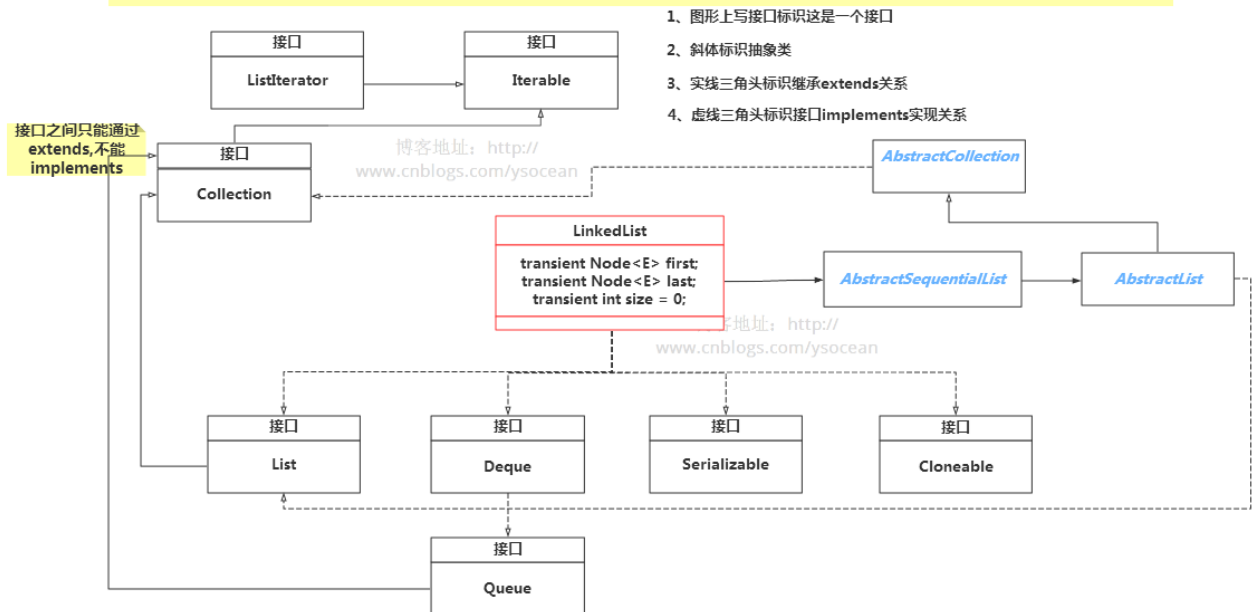
上一篇博客我们介绍了List集合的一种典型实现 [ArrayList](#)，我们知道 ArrayList 是由数组构成的，本篇博客我们介绍 List 集合的另一种典型实现 LinkedList。

### 1、LinkedList 定义

LinkedList 是一个用链表实现的集合，元素有序且可以重复。

```
1 public class LinkedList<E>
2     extends AbstractSequentialList<E>
3     implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

## java.util.LinkedList



和 `ArrayList` 集合一样, `LinkedList` 集合也实现了 `Cloneable` 接口和 `Serializable` 接口, 分别用来支持克隆以及支持序列化。 `List` 接口也不用多说, 定义了一套 `List` 集合类型的方法规范。

注意, 相对于 `ArrayList` 集合, `LinkedList` 集合多实现了一个 `Deque` 接口, 这是一个双向队列接口, 双向队列就是两端都可以进行增加和删除操作。

## 2、字段属性

```
1 //链表元素（节点）的个数
2 transient int size = 0;
3
4 /**
5  *指向第一个节点的指针
6  */
7 transient Node<E> first;
8
9 /**
10  *指向最后一个节点的指针
11  */
12 transient Node<E> last;
```

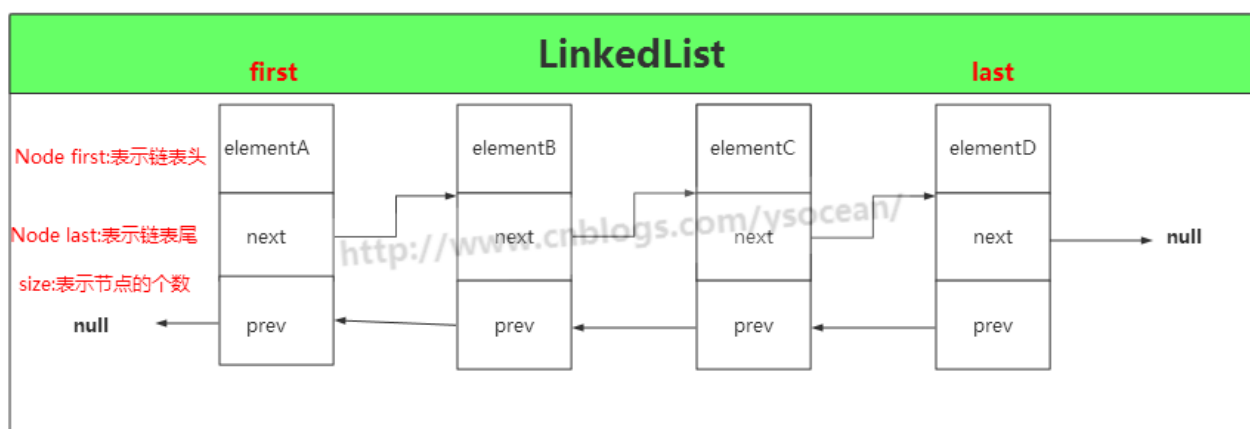
注意这里出现了一个 `Node` 类, 这是 `LinkedList` 类中的一个内部类, 其中每一个元素就代表一个 `Node` 类对象, `LinkedList` 集合就是由许多个 `Node` 对象类似于手拉着手构成。

```

1 private static class Node<E> {
2     E item; //实际存储的元素
3     Node<E> next; //指向上一个节点的引用
4     Node<E> prev; //指向下一个节点的引用
5
6     //构造函数
7     Node(Node<E> prev, E element, Node<E> next) {
8         this.item = element;
9         this.next = next;
10        this.prev = prev;
11    }
12 }

```

如下图所示：



上图的 LinkedList 是有四个元素，也就是由 4 个 Node 对象组成，size=4，head 指向第一个 elementA, tail 指向最后一个节点 elementD。

### 3、构造函数

```

1 public LinkedList() {
2 }
3 public LinkedList(Collection<? extends E> c) {
4     this();
5     addAll(c);
6 }

```

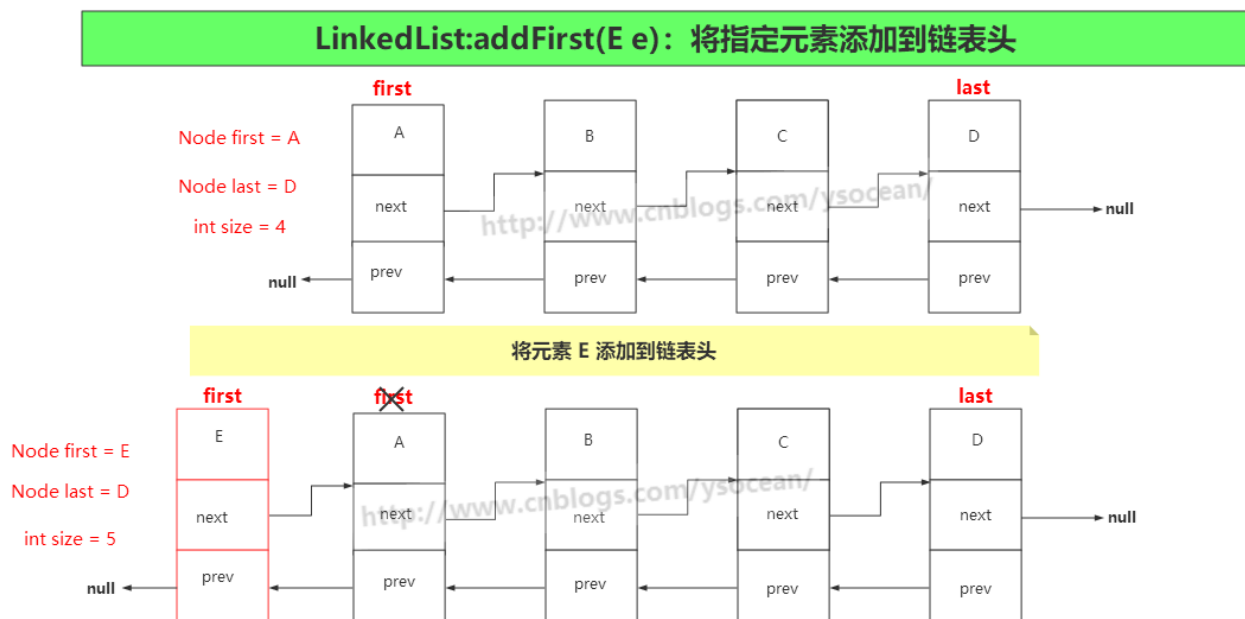
LinkedList 有两个构造函数，第一个是默认的空的构造函数，第二个是将已有元素的集合 Collection 的实例添加到 LinkedList 中，调用的是 addAll() 方法，这个方法下面我们会介绍。

注意：LinkedList 是没有初始化链表大小的构造函数，因为链表不像数组，一个定义好的数组是必须要有确定的大小，然后去分配内存空间，而链表不一样，它没有确定的大小，通过指针的移动来指向下一个内存地址的分配。

## 4、添加元素

### ①、addFirst(E e)

将指定元素添加到链表头

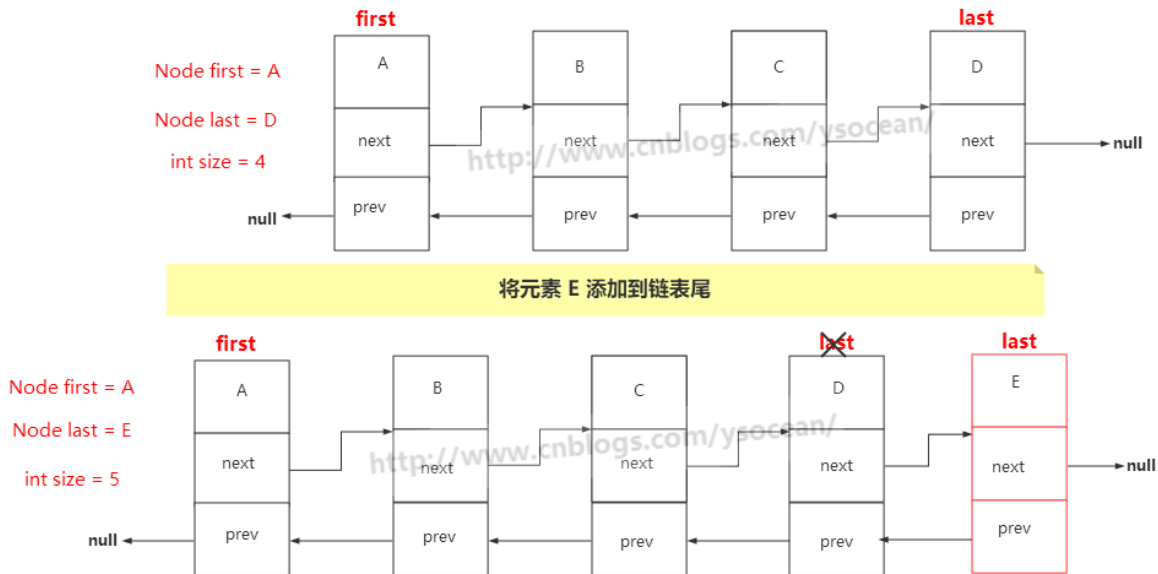


```
1 //将指定的元素附加到链表头节点
2 public void addFirst(E e) {
3     linkFirst(e);
4 }
5 private void linkFirst(E e) {
6     final Node<E> f = first;//将头节点赋值给 f
7     final Node<E> newNode = new Node<>(null, e, f);//将指定元素构造成一个
//新节点，此节点的指向下一个节点的引用为头节点
8     first = newNode;//将新节点设为头节点，那么原先的头节点 f 变为第二个节点
9     if (f == null)//如果第二个节点为空，也就是原先链表是空
10         last = newNode;//将这个新节点也设为尾节点（前面已经设为头节点了）
11     else
12         f.prev = newNode;//将原先的头节点的上一个节点指向新节点
13     size++;//节点数加1
14     modCount++;//和ArrayList中一样，iterator和listIterator方法返回的迭代器
//和列表迭代器实现使用。
15 }
```

### ②、addLast(E e)和add(E e)

将指定元素添加到链表尾

## LinkedList:addLast(E e)和add(E e)：将指定元素添加到链表尾



```

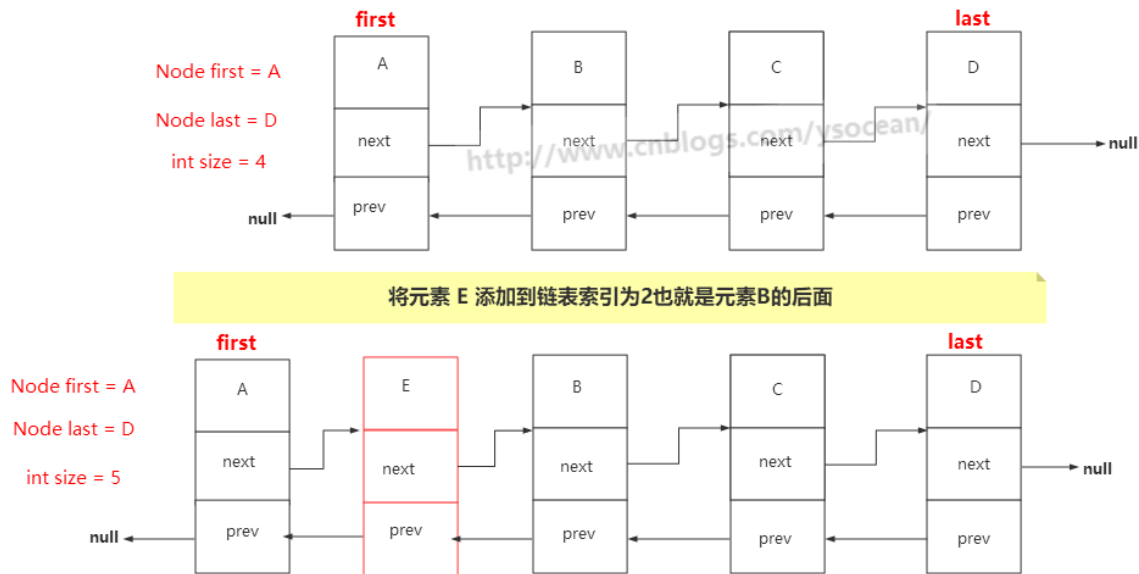
1  //将元素添加到链表末尾
2  public void addLast(E e) {
3      linkLast(e);
4  }
5  //将元素添加到链表末尾
6  public boolean add(E e) {
7      linkLast(e);
8      return true;
9  }
10 void linkLast(E e) {
11     final Node<E> l = last; //将l设为尾节点
12     final Node<E> newNode = new Node<>(l, e, null); //构造一个新节点，节点
    上一个节点引用指向尾节点l
13     last = newNode; //将尾节点设为创建的新节点
14     if (l == null) //如果尾节点为空，表示原先链表为空
15         first = newNode; //将头节点设为新创建的节点（尾节点也是新创建的节点）
16     else
17         l.next = newNode; //将原来尾节点下一个节点的引用指向新节点
18     size++; //节点数加1
19     modCount++; //和ArrayList中一样，iterator和listIterator方法返回的迭代器
    和列表迭代器实现使用。
20 }

```

### ③、add(int index, E element)

将指定的元素插入此列表中的指定位置

## LinkedList.add(int index, E element): 将指定元素添加到指定位置



```
1 //将指定的元素插入此列表中的指定位置
2 public void add(int index, E element) {
3     //判断索引 index >= 0 && index <= size中时抛出
4     //IndexOutOfBoundsException异常
5     checkPositionIndex(index);
6
7     if (index == size) //如果索引值等于链表大小
8         linkLast(element); //将节点插入到尾节点
9     else
10        linkBefore(element, node(index));
11 }
12 void linkLast(E e) {
13     final Node<E> l = last; //将l设为尾节点
14     final Node<E> newNode = new Node<>(l, e, null); //构造一个新节点, 节点
15     //上一个节点引用指向尾节点l
16     last = newNode; //将尾节点设为创建的新节点
17     if (l == null) //如果尾节点为空, 表示原先链表为空
18         first = newNode; //将头节点设为新创建的节点 (尾节点也是新创建的节点)
19     else
20         l.next = newNode; //将原来尾节点下一个节点的引用指向新节点
21     size++; //节点数加1
22     modCount++; //和ArrayList中一样, iterator和listIterator方法返回的迭代器
23     //和列表迭代器实现使用。
24 }
25 Node<E> node(int index) {
26     if (index < (size >> 1)) { //如果插入的索引在前半部分
27         Node<E> x = first; //设x为头节点
28         for (int i = 0; i < index; i++) //从开始节点到插入节点索引之间的所有
29             //节点向后移动一位
30             x = x.next;
31         return x;
32     }
33     // ... (rest of the method implementation)
34 }
```

```

28         } else { //如果插入节点位置在后半部分
29             Node<E> x = last; //将x设为最后一个节点
30             for (int i = size - 1; i > index; i--) //从最后节点到插入节点的索引
位置之间的所有节点向前移动一位
31                 x = x.prev;
32             return x;
33         }
34     }
35     void linkBefore(E e, Node<E> succ) {
36         final Node<E> pred = succ.prev; //将pred设为插入节点的上一个节点
37         final Node<E> newNode = new Node<>(pred, e, succ); //将新节点的上引用
设为pred, 下引用设为succ
38         succ.prev = newNode; //succ的上一个节点的引用设为新节点
39         if (pred == null) //如果插入节点的上一个节点引用为空
40             first = newNode; //新节点就是头节点
41         else
42             pred.next = newNode; //插入节点的下一个节点引用设为新节点
43         size++;
44         modCount++;
45     }

```

#### ④、addAll(Collection<? extends E> c)

按照指定集合的迭代器返回的顺序，将指定集合中的所有元素追加到此列表的末尾

此方法还有一个 addAll(int index, Collection<? extends E> c)，将集合 c 中所有元素插入到指定索引的位置。其实

addAll(Collection<? extends E> c) == addAll(size, Collection<? extends E> c)

源码如下：

```

1 //按照指定集合的迭代器返回的顺序，将指定集合中的所有元素追加到此列表的末尾。
2 public boolean addAll(Collection<? extends E> c) {
3     return addAll(size, c);
4 }
5 //将集合 c 中所有元素插入到指定索引的位置。
6 public boolean addAll(int index, Collection<? extends E> c) {
7     //判断索引 index >= 0 && index <= size中时抛出
IndexOutOfBoundsException异常
8     checkPositionIndex(index);
9
10    Object[] a = c.toArray(); //将集合转换成一个 Object 类型的数组
11    int numNew = a.length;
12    if (numNew == 0) //如果添加的集合为空，直接返回false
13        return false;
14
15    Node<E> pred, succ;
16    if (index == size) { //如果插入的位置等于链表的长度，就是将原集合元素附加到
链表的末尾
17        succ = null;

```

```

18         pred = last;
19     } else {
20         succ = node(index);
21         pred = succ.prev;
22     }
23
24     for (Object o : a) { //遍历要插入的元素
25         @SuppressWarnings("unchecked") E e = (E) o;
26         Node<E> newNode = new Node<>(pred, e, null);
27         if (pred == null)
28             first = newNode;
29         else
30             pred.next = newNode;
31         pred = newNode;
32     }
33
34     if (succ == null) {
35         last = pred;
36     } else {
37         pred.next = succ;
38         succ.prev = pred;
39     }
40
41     size += numNew;
42     modCount++;
43     return true;
44 }

```

看到上面向 LinkedList 集合中添加元素的各种方式，我们发现 LinkedList 每次添加元素只是改变元素的上一个指针引用和下一个指针引用，而且没有扩容。，对比于 ArrayList，需要扩容，而且在中间插入元素时，后面的所有元素都要移动一位，两者插入元素时的效率差异很大，下一篇博客会对这两者的效率，以及何种情况选择何种集合进行分析。

还有，每次进行添加操作，都有 modCount++ 的操作。

## 5、删除元素

删除元素和添加元素一样，也是通过更改指向上一个节点和指向下一个节点的引用即可，这里就不作图形展示了。

### ①、remove()和removeFirst()

从此列表中移除并返回第一个元素

```

1 //从此列表中移除并返回第一个元素
2 public E remove() {
3     return removeFirst();
4 }
5 //从此列表中移除并返回第一个元素
6 public E removeFirst() {

```



```

7         final Node<E> f = first; //f设为头结点
8         if (f == null)
9             throw new NoSuchElementException(); //如果头结点为空，则抛出异常
10        return unlinkFirst(f);
11    }
12    private E unlinkFirst(Node<E> f) {
13        // assert f == first && f != null;
14        final E element = f.item;
15        final Node<E> next = f.next; //next 为头结点的下一个节点
16        f.item = null;
17        f.next = null; // 将节点的元素以及引用都设为 null，便于垃圾回收
18        first = next; //修改头结点为第二个节点
19        if (next == null) //如果第二个节点为空（当前链表只存在第一个元素）
20            last = null; //那么尾节点也置为 null
21        else
22            next.prev = null; //如果第二个节点不为空，那么将第二个节点的上一个引用置
为 null
23        size--;
24        modCount++;
25        return element;
26    }

```

## ②、removeLast()

从该列表中删除并返回最后一个元素

```

1    //从该列表中删除并返回最后一个元素
2    public E removeLast() {
3        final Node<E> l = last;
4        if (l == null) //如果尾节点为空，表示当前集合为空，抛出异常
5            throw new NoSuchElementException();
6        return unlinkLast(l);
7    }
8
9    private E unlinkLast(Node<E> l) {
10        // assert l == last && l != null;
11        final E element = l.item;
12        final Node<E> prev = l.prev;
13        l.item = null;
14        l.prev = null; //将节点的元素以及引用都设为 null，便于垃圾回收
15        last = prev; //尾节点为倒数第二个节点
16        if (prev == null) //如果倒数第二个节点为null
17            first = null; //那么将节点也置为 null
18        else
19            prev.next = null; //如果倒数第二个节点不为空，那么将倒数第二个节点的下一
个引用置为 null
20        size--;
21        modCount++;
22        return element;

```

### ③、remove(int index)

删除此列表中指定位置的元素

```

1 //删除此列表中指定位置的元素
2     public E remove(int index) {
3         //判断索引 index >= 0 && index <= size中时抛出
IndexOutOfBoundsException异常
4         checkElementIndex(index);
5         return unlink(node(index));
6     }
7     E unlink(Node<E> x) {
8         // assert x != null;
9         final E element = x.item;
10        final Node<E> next = x.next;
11        final Node<E> prev = x.prev;
12
13        if (prev == null) { //如果删除节点位置的上一个节点引用为null (表示删除第一个
元素)
14            first = next; //将头结点置为第一个元素的下一个节点
15        } else { //如果删除节点位置的上一个节点引用不为null
16            prev.next = next; //将删除节点的上一个节点的下一个节点引用指向删除节点的
下一个节点 (去掉删除节点)
17            x.prev = null; //删除节点的上一个节点引用置为null
18        }
19
20        if (next == null) { //如果删除节点的下一个节点引用为null (表示删除最后一个节
点)
21            last = prev; //将尾节点置为删除节点的上一个节点
22        } else { //不是删除尾节点
23            next.prev = prev; //将删除节点的下一个节点的上一个节点的引用指向删除节点
的上一个节点
24            x.next = null; //将删除节点的下一个节点引用置为null
25        }
26
27        x.item = null; //删除节点内容置为null, 便于垃圾回收
28        size--;
29        modCount++;
30        return element;
31    }

```

### ④、remove(Object o)

如果存在, 则从该列表中删除指定元素的第一次出现

此方法本质上和 remove(int index) 没多大区别, 通过循环判断元素进行删除, 需要注意的是, 是删除第一次出现的元素, 不是所有的。

```

1 public boolean remove(Object o) {
2     if (o == null) {
3         for (Node<E> x = first; x != null; x = x.next) {
4             if (x.item == null) {
5                 unlink(x);
6                 return true;
7             }
8         }
9     } else {
10        for (Node<E> x = first; x != null; x = x.next) {
11            if (o.equals(x.item)) {
12                unlink(x);
13                return true;
14            }
15        }
16    }
17    return false;
18 }

```

## 6、修改元素

通过调用 `set(int index, E element)` 方法，用指定的元素替换此列表中指定位置的元素。

```

1 public E set(int index, E element) {
2     //判断索引 index >= 0 && index <= size中时抛出
    IndexOutOfBoundsException异常
3     checkElementIndex(index);
4     Node<E> x = node(index); //获取指定索引处的元素
5     E oldVal = x.item;
6     x.item = element; //将指定位置的元素替换成要修改的元素
7     return oldVal; //返回指定索引位置原来的元素
8 }

```

这里主要是通过 `node(index)` 方法获取指定索引位置的节点，然后修改此节点位置的元素即可。

## 7、查找元素

### ①、getFirst()

返回此列表中的第一个元素

```

1 public E getFirst() {
2     final Node<E> f = first;
3     if (f == null)
4         throw new NoSuchElementException();
5     return f.item;
6 }

```

## ②、getLast()

返回此列表中的最后一个元素

```
1 public E getLast() {
2     final Node<E> l = last;
3     if (l == null)
4         throw new NoSuchElementException();
5     return l.item;
6 }
```

## ③、get(int index)

返回指定索引处的元素

```
1 public E get(int index) {
2     checkElementIndex(index);
3     return node(index).item;
4 }
```

## ④、indexOf(Object o)

返回此列表中指定元素第一次出现的索引，如果此列表不包含元素，则返回-1。

```
1 //返回此列表中指定元素第一次出现的索引，如果此列表不包含元素，则返回-1。
2 public int indexOf(Object o) {
3     int index = 0;
4     if (o == null) { //如果查找的元素为null(LinkedList允许null值)
5         for (Node<E> x = first; x != null; x = x.next) { //从头结点开始不
断向下一个节点进行遍历
6             if (x.item == null)
7                 return index;
8             index++;
9         }
10    } else { //如果查找的元素不为null
11        for (Node<E> x = first; x != null; x = x.next) {
12            if (o.equals(x.item))
13                return index;
14            index++;
15        }
16    }
17    return -1; //找不到返回-1
18 }
```

# 8、遍历集合

## ①、普通 for 循环

```

1  LinkedList<String> linkedList = new LinkedList<>();
2  linkedList.add("A");
3  linkedList.add("B");
4  linkedList.add("C");
5  linkedList.add("D");
6  for(int i = 0 ; i < linkedList.size() ; i++){
7      System.out.print(linkedList.get(i)+" "); //A B C D
8  }

```

代码很简单，我们就利用 LinkedList 的 get(int index) 方法，遍历出所有的元素。

但是需要注意的是，get(int index) 方法每次都要遍历该索引之前的所有元素，这句话这么理解：

比如上面的一个 LinkedList 集合，我放入了 A,B,C,D 是个元素。总共需要四次遍历：

第一次遍历打印 A：只需遍历一次。

第二次遍历打印 B：需要先找到 A，然后再找到 B 打印。

第三次遍历打印 C：需要先找到 A，然后找到 B，最后找到 C 打印。

第四次遍历打印 D：需要先找到 A，然后找到 B，然后找到 C，最后找到 D。

这样如果集合元素很多，越查找到后面（当然此处的 get 方法进行了优化，查找前半部分从前面开始遍历，查找后半部分从后面开始遍历，但是需要的时间还是很多）花费的时间越多。那么如何改进呢？

## ②、迭代器

```

1  LinkedList<String> linkedList = new LinkedList<>();
2  linkedList.add("A");
3  linkedList.add("B");
4  linkedList.add("C");
5  linkedList.add("D");
6
7
8  Iterator<String> listIt = linkedList.listIterator();
9  while(listIt.hasNext()){
10     System.out.print(listIt.next()+" "); //A B C D
11 }
12
13 //通过适配器模式实现的接口，作用是倒叙打印链表
14 Iterator<String> it = linkedList.descendingIterator();
15 while(it.hasNext()){
16     System.out.print(it.next()+" "); //D C B A
17 }

```

在 LinkedList 集合中也有一个内部类 ListItr，方法实现大体上也差不多，通过移动游标指向每一次要遍历的元素，不用在遍历某个元素之前都要从头开始。其方法实现也比较简单：

```

1  public ListIterator<E> listIterator(int index) {
2      checkPositionIndex(index);

```

```

3         return new ListItr(index);
4     }
5
6     private class ListItr implements ListIterator<E> {
7         private Node<E> lastReturned;
8         private Node<E> next;
9         private int nextIndex;
10        private int expectedModCount = modCount;
11
12        ListItr(int index) {
13            // assert isPositionIndex(index);
14            next = (index == size) ? null : node(index);
15            nextIndex = index;
16        }
17
18        public boolean hasNext() {
19            return nextIndex < size;
20        }
21
22        public E next() {
23            checkForComodification();
24            if (!hasNext())
25                throw new NoSuchElementException();
26
27            lastReturned = next;
28            next = next.next;
29            nextIndex++;
30            return lastReturned.item;
31        }
32
33        public boolean hasPrevious() {
34            return nextIndex > 0;
35        }
36
37        public E previous() {
38            checkForComodification();
39            if (!hasPrevious())
40                throw new NoSuchElementException();
41
42            lastReturned = next = (next == null) ? last : next.prev;
43            nextIndex--;
44            return lastReturned.item;
45        }
46
47        public int nextIndex() {
48            return nextIndex;
49        }
50
51        public int previousIndex() {

```

```
52         return nextIndex - 1;
53     }
54
55     public void remove() {
56         checkForComodification();
57         if (lastReturned == null)
58             throw new IllegalStateException();
59
60         Node<E> lastNext = lastReturned.next;
61         unlink(lastReturned);
62         if (next == lastReturned)
63             next = lastNext;
64         else
65             nextIndex--;
66         lastReturned = null;
67         expectedModCount++;
68     }
69
70     public void set(E e) {
71         if (lastReturned == null)
72             throw new IllegalStateException();
73         checkForComodification();
74         lastReturned.item = e;
75     }
76
77     public void add(E e) {
78         checkForComodification();
79         lastReturned = null;
80         if (next == null)
81             linkLast(e);
82         else
83             linkBefore(e, next);
84         nextIndex++;
85         expectedModCount++;
86     }
87
88     public void forEachRemaining(Consumer<? super E> action) {
89         Objects.requireNonNull(action);
90         while (modCount == expectedModCount && nextIndex < size) {
91             action.accept(next.item);
92             lastReturned = next;
93             next = next.next;
94             nextIndex++;
95         }
96         checkForComodification();
97     }
98
99     final void checkForComodification() {
100         if (modCount != expectedModCount)
```

```

101         throw new ConcurrentModificationException();
102     }
103 }

```

这里需要重点注意的是 modCount 字段，前面我们在增加和删除元素的时候，都会进行自增操作 modCount，这是因为如果想一边迭代，一边用集合自带的方法进行删除或者新增操作，都会抛出异常。（使用迭代器的增删方法不会抛异常）

```

1 final void checkForComodification() {
2     if (modCount != expectedModCount)
3         throw new ConcurrentModificationException();
4 }

```

比如：

```

1 LinkedList<String> linkedList = new LinkedList<>();
2 linkedList.add("A");
3 linkedList.add("B");
4 linkedList.add("C");
5 linkedList.add("D");
6
7
8 Iterator<String> listIt = linkedList.listIterator();
9 while(listIt.hasNext()){
10     System.out.print(listIt.next()+" "); //A B C D
11     //linkedList.remove();//此处会抛出异常
12     listIt.remove();//这样可以进行删除操作
13 }

```

迭代器的另一种形式就是使用 foreach 循环，底层实现也是使用的迭代器，这里我们就不做介绍了。

```

1 LinkedList<String> linkedList = new LinkedList<>();
2 linkedList.add("A");
3 linkedList.add("B");
4 linkedList.add("C");
5 linkedList.add("D");
6 for(String str : linkedList){
7     System.out.print(str + " ");
8 }

```



## 9、迭代器和for循环效率差异

```
1  LinkedList<Integer> linkedList = new LinkedList<>();
2  for(int i = 0 ; i < 10000 ; i++){//向链表中添加一万个元素
3      linkedList.add(i);
4  }
5  long beginTimeFor = System.currentTimeMillis();
6  for(int i = 0 ; i < 10000 ; i++){
7      System.out.print(linkedList.get(i));
8  }
9  long endTimeFor = System.currentTimeMillis();
10 System.out.println("使用普通for循环遍历10000个元素需要的时间: "+ (endTimeFor -
    beginTimeFor));
11
12
13 long beginTimeIte = System.currentTimeMillis();
14 Iterator<Integer> it = linkedList.listIterator();
15 while(it.hasNext()){
16     System.out.print(it.next()+" ");
17 }
18 long endTimeIte = System.currentTimeMillis();
19 System.out.println("使用迭代器遍历10000个元素需要的时间: "+ (endTimeIte -
    beginTimeIte));
```

打印结果为：

使用普通for循环遍历10000个元素需要的时间：94

使用迭代器遍历10000个元素需要的时间：39

一万个元素两者之间都相差一倍多的时间，如果是十万，百万个元素，那么两者之间相差的速度会越来越大。下面通过图形来解释：

普通for循环：每次遍历一个索引的元素之前，都要访问之间所有的索引。

### 普通for循环遍历元素



迭代器：每次访问一个元素后，都会用游标记录当前访问元素的位置，遍历一个元素，记录一个位置。

### 迭代器遍历元素



参考文档：

<https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html#>



微信搜一搜

IT可乐

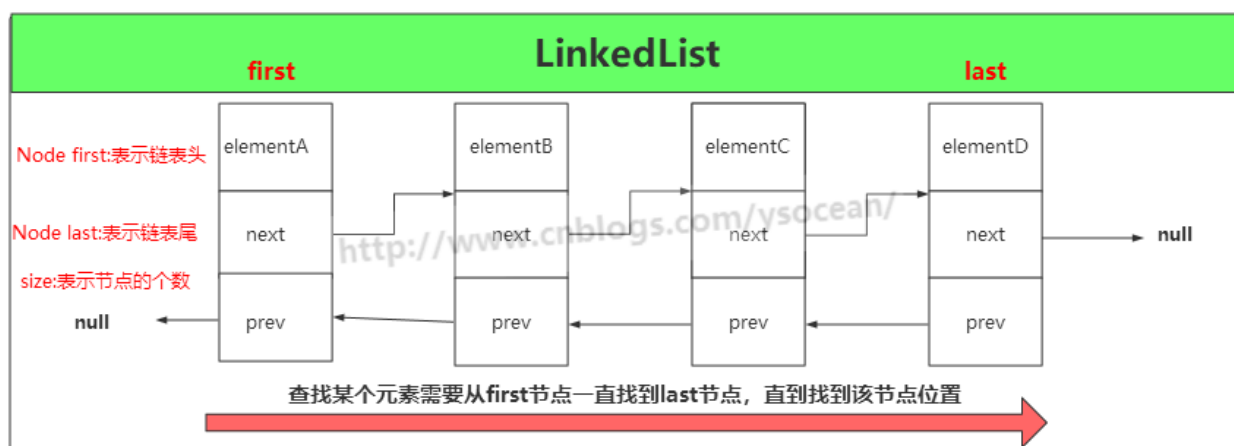
## JDK源码解析(7)——java.util.HashMap 类

本篇博客我们来介绍在JDK1.8中HashMap的源码实现，这也是最常用的一个集合。但是在介绍HashMap之前，我们先介绍什么是Hash表。

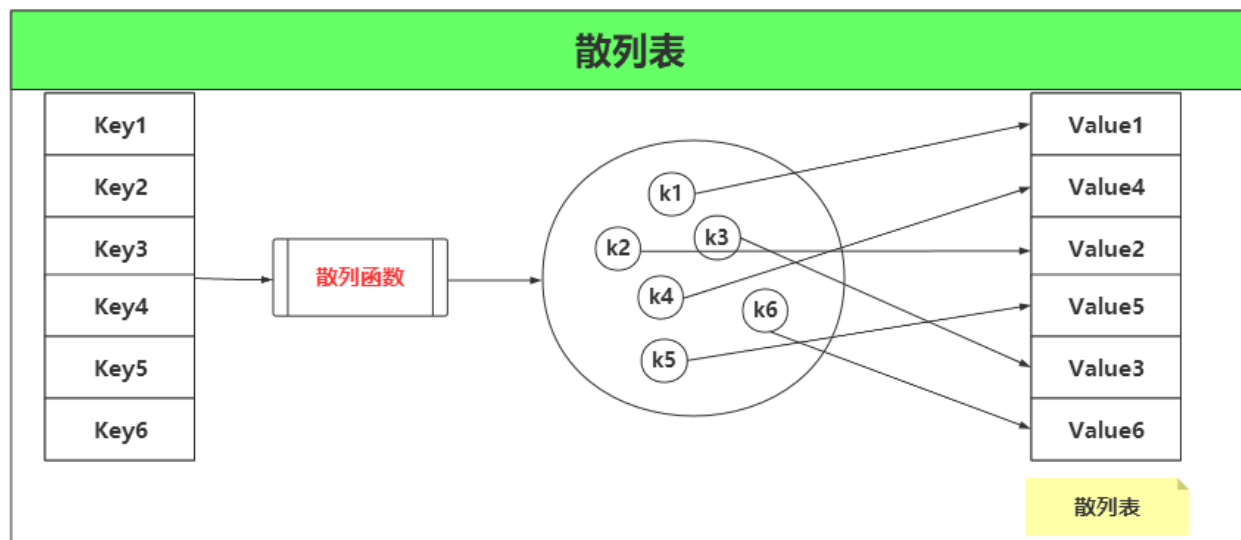
### 1、哈希表

Hash表也称为散列表，也有直接译作哈希表，Hash表是一种根据关键字值（key - value）而直接进行访问的数据结构。也就是说它通过把关键码值映射到表中的一个位置来访问记录，以此来加快查找的速度。在链表、数组等数据结构中，查找某个关键字，通常要遍历整个数据结构，也就是 $O(N)$ 的时间级，但是对于哈希表来说，只是 $O(1)$ 的时间级。

比如对于前面我们讲解的ArrayList集合和LinkedList，如果我们要查找这两个集合中的某个元素，通常是通过遍历整个集合，需要 $O(N)$ 的时间级。



如果是哈希表，它是通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表，只需要 $O(1)$ 的时间级。



①、存放在哈希表中的数据是key-value 键值对，比如存放哈希表的数据为：

{Key1-Value1,Key2-Value2,Key3-Value3,Key4-Value4,Key5-Value5,Key6-Value6}

如果我们想查找是否存在键值对 Key3-Value3，首先通过 Key3 经过散列函数，得到值 k3，然后通过 k3 和散列表对应的值找到是 Value3。

②、当然也有可能存放哈希表的值只是 Value1,Value2,Value3这种类型：

{Value1,Value2,Value3,Value4,Value5,Value6}

这时候我们可以假设 Value1 是等于 Key1的，也就是{Value1-Value1,Value2-Value2,Value3-Value3,Value4-Value4,Value5-Value5,Value6-Value6}可以将 Value1经过散列函数转换成与散列表对应的值。

大家都用过汉语字典吧，汉语字典的优点是我们可以通过前面的拼音目录快速定位到所要查找的汉字。当给定我们某个汉字时，大脑会自动将汉字转换成拼音（如果我们认识，不认识可以通过偏旁部首），这个转换的过程我们可以看成是一个散列函数，之后在根据转换得到的拼音找到该字所在的页码，从而找到该汉字。

汉语字典是哈希表的典型实现，但是我们仔细思考，会发现这样几个问题？

①、为什么要有散列函数？

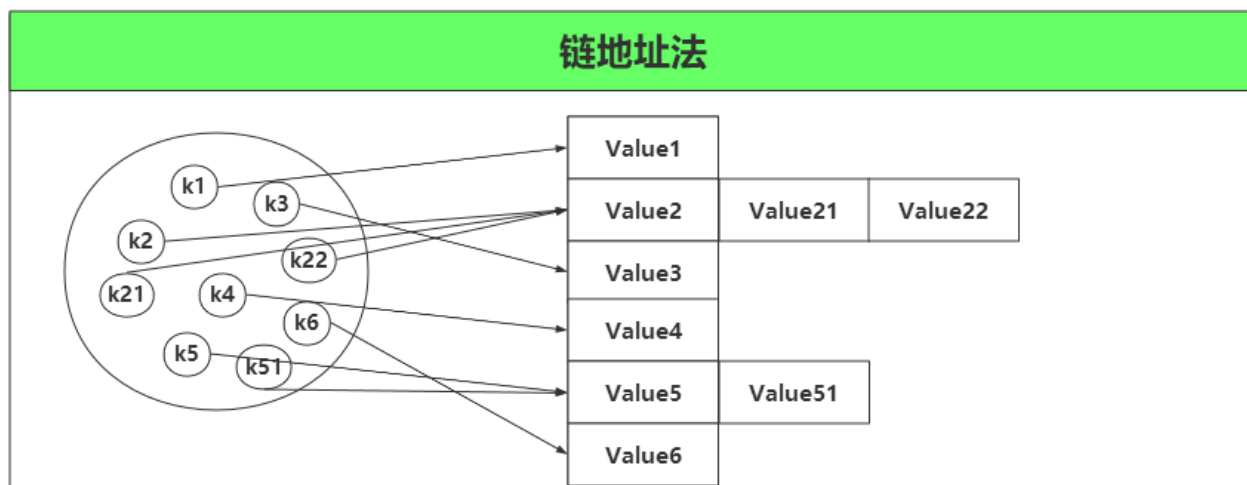
②、多个 key 通过散列函数会得到相同的值，这时候怎么办？

对于第一个问题，散列函数的存在能够帮助我们更快的确定key和value的映射关系，试想一下，如果没有汉字和拼音的转换规则（或者汉字和偏旁部首的），给你一个汉字，你该如何从字典中找到该汉字？我想除了遍历整部字典，你没有什么更好的办法。

对于第二个问题，多个 key 通过散列函数得到相同的值，这其实也是哈希表最大的问题——冲突。比如同音字汉字，我们得到的拼音就会是相同的，那么我们该如何在字典中存放同音字汉字呢？有两种做法：

第一种是开放地址法，当我们遇到冲突了，这时候通过另一种函数再计算一遍，得到相应的映射关系。比如对于汉语字典，一个字“余”，拼音是“yu”，我们将其放在页码为567(假设在该位置)，这时候又来了一个汉字“于”，拼音也是“yu”，那么这时候我们要是按照转换规则，也得将其放在页码为567的位置，但是我们发现这个页码已经被占用了，这时候怎么办？我们可以在通过另一种函数，得到的值加1。那么汉字“于”就会被放在567+1=568的位置。

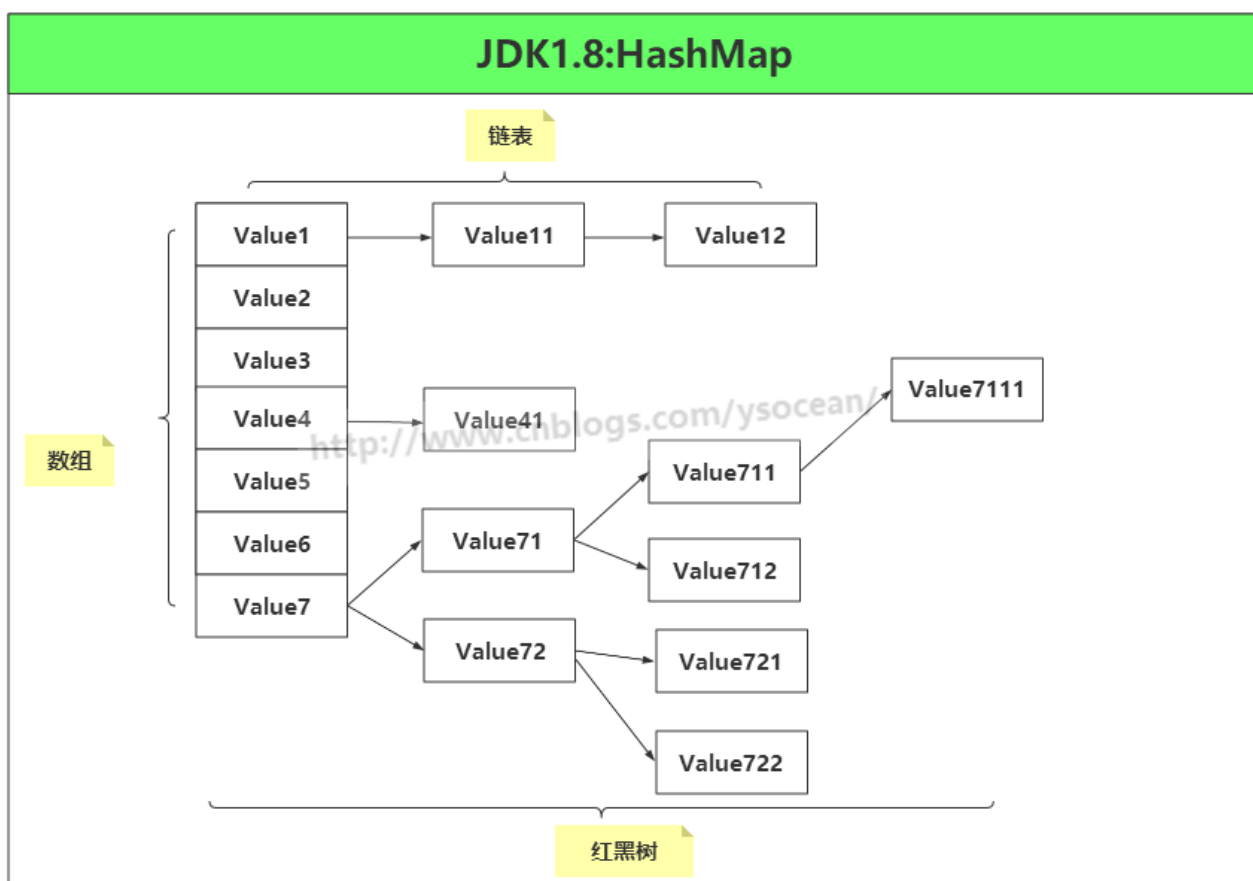
第二种是链地址法，我们可以将字典的每一页都看成是一个子数组或者子链表，当遇到冲突了，直接往当前页码的子数组或者子链表里面填充即可。那么我们进行同音字查找的时候，可能需要遍历其子数组或者子链表。如下图所示：



对于开放地址法，可能会遇到二次冲突，三次冲突，所以需要良好的散列函数，分布的越均匀越好。对于链地址法，虽然不会造成二次冲突，但是如果一次冲突很多，那么会造成子数组或者子链表很长，那么我们查找所需遍历的时间也会很长。

## 2、什么是 HashMap?

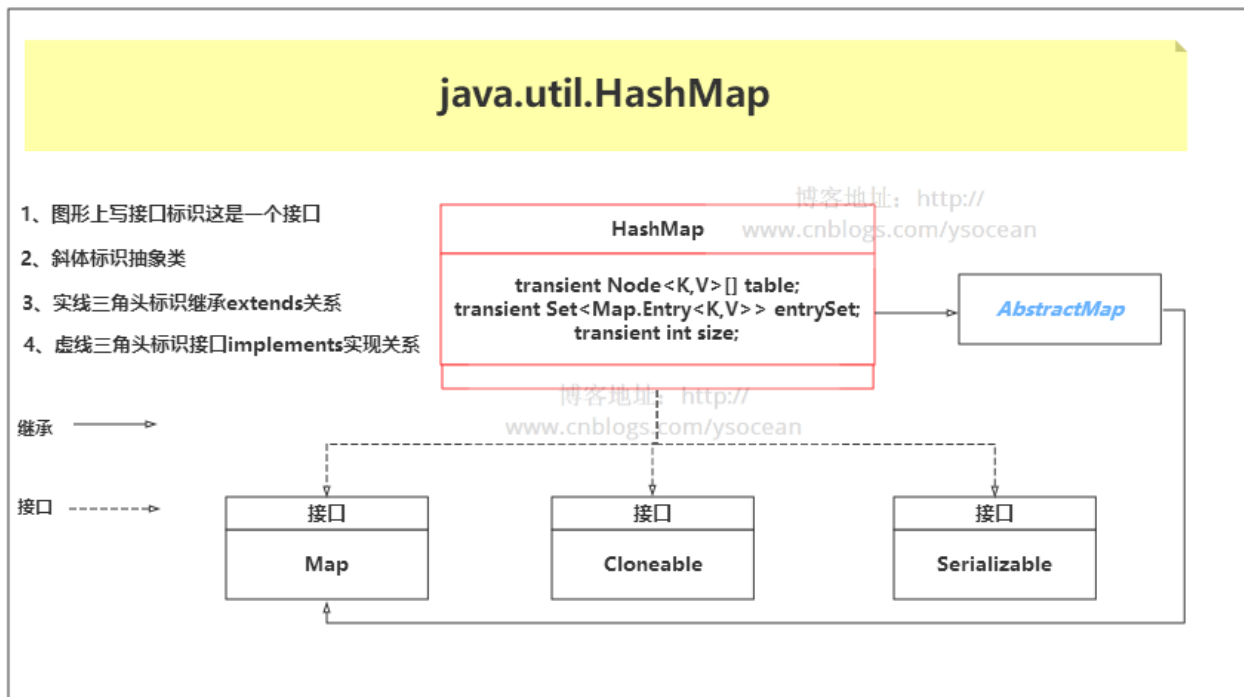
听名字就知道，HashMap 是一个利用哈希表原理来存储元素的集合。遇到冲突时，HashMap 是采用链地址法来解决，在 JDK1.7 中，HashMap 是由 数组+链表构成的。但是在 JDK1.8 中，HashMap 是由 数组+链表+红黑树构成，新增了红黑树作为底层数据结构，结构变得复杂了，但是效率也变的更高效。下面我们来具体介绍在 JDK1.8 中 HashMap 是如何实现的。



### 3、HashMap定义

HashMap 是一个散列表，它存储的内容是键值对(key-value)映射，而且 key 和 value 都可以为 null。

```
1 public class HashMap<K,V> extends AbstractMap<K,V>
2     implements Map<K,V>, Cloneable, Serializable {
```



首先该类实现了一个 Map 接口，该接口定义了一组键值对映射通用的操作。储存一组成对的键-值对象，提供key（键）到value（值）的映射，Map中的key不要求有序，不允许重复。value同样不要求有序，但可以重复。但是我们发现该接口方法有很多，我们设计某个键值对的集合有时候并不像实现那么多方法，那该怎么办？

JDK 还为我们提供了一个抽象类 `AbstractMap`，该抽象类继承 Map 接口，所以如果我们不想实现所有的 Map 接口方法，就可以选择继承抽象类 `AbstractMap`。

但是我们发现 `HashMap` 类即继承了 `AbstractMap` 接口，也实现了 Map 接口，这样做难道不是多此一举？后面我们会讲的 `LinkedHashSet` 集合也有这样的写法。

毕竟 JDK 经过这么多年的发展维护，博主起初也是认为这样是有具体的作用的，后来找了很多资料，发现这其实完全没有任何作用

据 java 集合框架的创始人 Josh Bloch 描述，这样的写法是一个失误。在 java 集合框架中，类似这样的写法很多，最开始写 java 集合框架的时候，他认为这样写，在某些地方可能是有价值的，直到他意识到错了。显然的，JDK 的维护者，后来不认为这个小小的失误值得去修改，所以就存在下来了。

`HashMap` 集合还实现了 `Cloneable` 接口以及 `Serializable` 接口，分别用来进行对象克隆以及将对象进行序列化。

## 4、字段属性

```
1 //序列化和反序列化时，通过该字段进行版本一致性验证
2 private static final long serialVersionUID = 362498820763181265L;
3 //默认 HashMap 集合初始容量为16 (必须是 2 的倍数)
4 static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
5 //集合的最大容量，如果通过带参构造指定的最大容量超过此数，默认还是使用此数
6 static final int MAXIMUM_CAPACITY = 1 << 30;
7 //默认的填充因子
8 static final float DEFAULT_LOAD_FACTOR = 0.75f;
9 //当桶(bucket)上的结点数大于这个值时会转成红黑树(JDK1.8新增)
10 static final int TREEIFY_THRESHOLD = 8;
11 //当桶(bucket)上的节点数小于这个值时会转成链表(JDK1.8新增)
12 static final int UNTREEIFY_THRESHOLD = 6;
13 /**(JDK1.8新增)
14  * 当集合中的容量大于这个值时，表中的桶才能进行树形化，否则桶内元素太多时会扩容，
15  * 而不是树形化 为了避免进行扩容、树形化选择的冲突，这个值不能小于 4 *
16 TREEIFY_THRESHOLD
17  */
18 static final int MIN_TREEIFY_CAPACITY = 64;
```

**注意：**后面三个字段是JDK1.8 新增的，主要是用来进行红黑树和链表的互相转换。

```
1 /**
2  * 初始化使用，长度总是 2的幂
3  */
4 transient Node<K,V>[] table;
5
6 /**
7  * 保存缓存的entrySet ()
8  */
9 transient Set<Map.Entry<K,V>> entrySet;
10
11 /**
12  * 此映射中包含的键值映射的数量。（集合存储键值对的数量）
13  */
14 transient int size;
15
16 /**
17  * 跟前面ArrayList和LinkedList集合中的字段modCount一样，记录集合被修改的次数
18  * 主要用于迭代器中的快速失败
19  */
20 transient int modCount;
21
22 /**
23  * 调整大小的下一个大小值（容量*加载因子）。capacity * load factor
24  */
25 int threshold;
```

```

26
27     /**
28      * 散列表的加载因子。
29      */
30     final float loadFactor;

```

下面我们重点介绍上面几个字段：

### ①、Node<K,V>[] table

我们说 HashMap 是由数组+链表+红黑树组成，这里的数组就是 table 字段。后面对其进行初始化长度默认是 DEFAULT\_INITIAL\_CAPACITY= 16。而且 JDK 声明数组的长度总是 2 的 n 次方(一定是合数)，为什么这里要求是合数，一般我们知道哈希算法为了避免冲突都要求长度是质数，这里要求是合数，下面在介绍 HashMap 的 hashCode() 方法(散列函数)，我们再进行讲解。

### ②、size

集合中存放 key-value 的实时对数。

### ③、loadFactor

装载因子，是用来衡量 HashMap 满的程度，计算 HashMap 的实时装载因子的方法为：  
size/capacity，而不是占用桶的数量去除以 capacity。capacity 是桶的数量，也就是 table 的长度 length。

默认的负载因子 0.75 是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下，如果内存空间很多而又对时间效率要求很高，可以降低负载因子 loadFactor 的值；相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子 loadFactor 的值，这个值可以大于 1。

### ④、threshold

计算公式：capacity \* loadFactor。这个值是当前已占用数组长度的最大值。过这个数目就重新 resize(扩容)，扩容后的 HashMap 容量是之前容量的两倍

## 5、构造函数

### ①、默认无参构造函数

```

1     /**
2      * 默认构造函数，初始化加载因子 loadFactor = 0.75
3      */
4     public HashMap() {
5         this.loadFactor = DEFAULT_LOAD_FACTOR;
6     }

```

无参构造器，初始化散列表的加载因子为 0.75

### ②、指定初始容量的构造函数

```

1     /**
2      *

```



```

3      * @param initialCapacity 指定初始化容量
4      * @param loadFactor 加载因子 0.75
5      */
6      public HashMap(int initialCapacity, float loadFactor) {
7          //初始化容量不能小于 0 , 否则抛出异常
8          if (initialCapacity < 0)
9              throw new IllegalArgumentException("Illegal initial capacity:
" +
10
11                                     initialCapacity);
12          //如果初始化容量大于2的30次方, 则初始化容量都为2的30次方
13          if (initialCapacity > MAXIMUM_CAPACITY)
14              initialCapacity = MAXIMUM_CAPACITY;
15          //如果加载因子小于0, 或者加载因子是一个非数值, 抛出异常
16          if (loadFactor <= 0 || Float.isNaN(loadFactor))
17              throw new IllegalArgumentException("Illegal load factor: " +
18
19                                     loadFactor);
20          this.loadFactor = loadFactor;
21          this.threshold = tableSizeFor(initialCapacity);
22      }
23      // 返回大于等于initialCapacity的最小的二次幂数值。
24      // >>> 操作符表示无符号右移, 高位取0。
25      // | 按位或运算
26      static final int tableSizeFor(int cap) {
27          int n = cap - 1;
28          n |= n >>> 1;
29          n |= n >>> 2;
30          n |= n >>> 4;
31          n |= n >>> 8;
32          n |= n >>> 16;
33          return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY :
n + 1;
34      }

```

## 6、确定哈希桶数组索引位置

前面我们讲解哈希表的时候, 我们知道是用散列函数来确定索引的位置。散列函数设计的越好, 使得元素分布的越均匀。HashMap 是数组+链表+红黑树的组合, 我们希望在有限个数组位置时, 尽量每个位置的元素只有一个, 那么当我们用散列函数求得索引位置的时候, 我们能马上知道对应位置的元素是不是我们想要的, 而不是要进行链表的遍历或者红黑树的遍历, 这会大大优化我们的查询效率。我们看 HashMap 中的哈希算法:

```

1      static final int hash(Object key) {
2          int h;
3          return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
4      }
5
6      i = (table.length - 1) & hash; //这一步是在后面添加元素putVal()方法中进行位置的
      确定

```

主要分为三步：

- ①、取 hashCode 值：key.hashCode()
- ②、高位参与运算：h>>>16
- ③、取模运算：(n-1) & hash

这里获取 hashCode() 方法的值是变量，但是我们知道，对于任意给定的对象，只要它的 hashCode() 返回值相同，那么程序调用 hash(Object key) 所计算得到的 hash 码 值总是相同的。

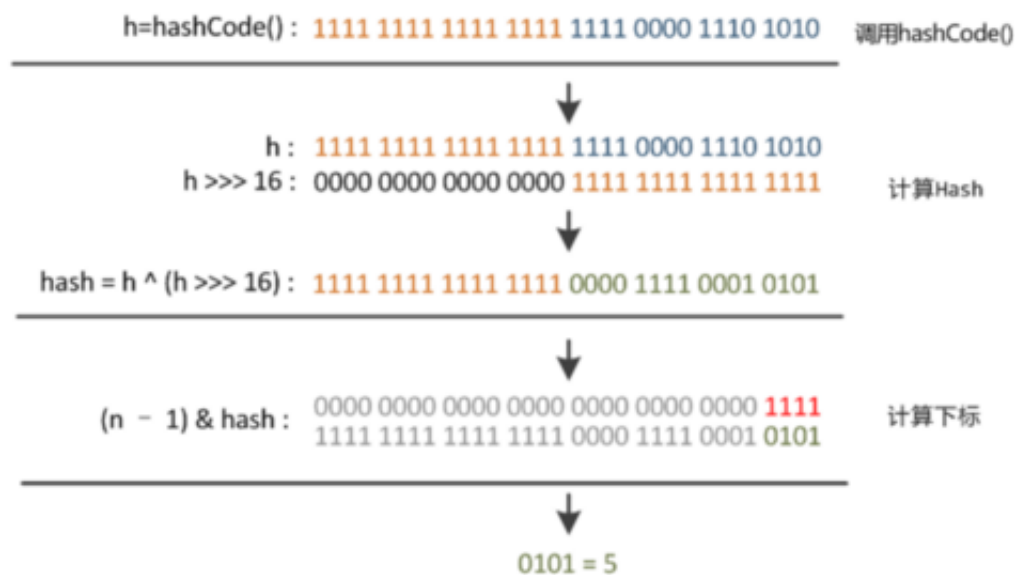
为了让数组元素分布均匀，我们首先想到的是把获得的 hash 码对数组长度取模运算(hash%length)，但是计算机都是二进制进行操作，取模运算相对开销还是很大的，那该如何优化呢？

HashMap 使用的方法很巧妙，它通过 hash & (table.length -1)来得到该对象的保存位，前面说过 HashMap 底层数组的长度总是2的n次方，这是HashMap在速度上的优化。当 length 总是2的n次方时，hash & (length-1)运算等价于对 length 取模，也就是 hash%length，但是&比%具有更高的效率。比如  $n \% 32 = n \& (32 - 1)$

这也解释了为什么要保证数组的长度总是2的n次方。

再就是在JDK1.8 中还有个高位参与运算，hashCode() 得到的是一个32位 int 类型的值，通过 hashCode()的高16位 异或 低16位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

下面举例说明下，n为table的长度：



## 7、添加元素

```
1 //hash(key)就是上面讲的hash方法，对其进行了第一步和第二步处理
2 public V put(K key, V value) {
3     return putVal(hash(key), key, value, false, true);
4 }
5 /**
6     *
```

```

7      * @param hash 索引的位置
8      * @param key 键
9      * @param value 值
10     * @param onlyIfAbsent true 表示不要更改现有值
11     * @param evict false表示table处于创建模式
12     * @return
13     */
14     final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
15                   boolean evict) {
16         Node<K,V>[] tab; Node<K,V> p; int n, i;
17         //如果table为null或者长度为0, 则进行初始化
18         //resize()方法本来是用于扩容, 由于初始化没有实际分配空间, 这里用该方法进行空
间分配, 后面会详细讲解该方法
19         if ((tab = table) == null || (n = tab.length) == 0)
20             n = (tab = resize()).length;
21         //注意: 这里用到了前面讲解获得key的hash码的第三步, 取模运算, 下面的if-else
分别是 tab[i] 为null和不为null
22         if ((p = tab[i = (n - 1) & hash]) == null)
23             tab[i] = newNode(hash, key, value, null); //tab[i] 为null, 直接
将新的key-value插入到计算的索引i位置
24         else { //tab[i] 不为null, 表示该位置已经有值了
25             Node<K,V> e; K k;
26             if (p.hash == hash &&
27                 ((k = p.key) == key || (key != null && key.equals(k))))
28                 e = p; //节点key已经有值了, 直接用新值覆盖
29             //该链是红黑树
30             else if (p instanceof TreeNode)
31                 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
value);
32             //该链是链表
33             else {
34                 for (int binCount = 0; ; ++binCount) {
35                     if ((e = p.next) == null) {
36                         p.next = newNode(hash, key, value, null);
37                         //链表长度大于8, 转换成红黑树
38                         if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for
1st
39                             treeifyBin(tab, hash);
40                         break;
41                     }
42                     //key已经存在直接覆盖value
43                     if (e.hash == hash &&
44                         ((k = e.key) == key || (key != null &&
key.equals(k))))
45                         break;
46                     p = e;
47                 }
48             }
49             if (e != null) { // existing mapping for key

```

```

50         V oldValue = e.value;
51         if (!onlyIfAbsent || oldValue == null)
52             e.value = value;
53         afterNodeAccess(e);
54         return oldValue;
55     }
56 }
57 ++modCount; //用作修改和新增快速失败
58 if (++size > threshold) //超过最大容量，进行扩容
59     resize();
60 afterNodeInsertion(evict);
61 return null;
62 }

```

①、判断键值对数组 table 是否为空或为null，否则执行resize()进行扩容；

②、根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③、判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；

④、判断table[i] 是否为TreeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤、遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥、插入成功后，判断实际存在的键值对数量size是否超过了最大容量threshold，如果超过，进行扩容。

⑦、如果新插入的key不存在，则返回null，如果新插入的key存在，则返回原key对应的value值（注意新插入的value会覆盖原value值）

注意1：看第 58,59 行代码：

```

1  if (++size > threshold) //超过最大容量，进行扩容
2      resize();

```

这里有个考点，我们知道 HashMap 是由数组+链表+红黑树（JDK1.8）组成，如果在添加元素时，发生冲突，会将冲突的数放在链表上，当链表长度超过8时，会自动转换成红黑树。

那么有如下问题：数组上有5个元素，而某个链表上有3个元素，问此HashMap的 size 是多大？

我们分析第58,59 行代码，很容易知道，只要是调用put() 方法添加元素，那么就会调用 ++size(这里有个例外是插入重复key的键值对，不会调用，但是重复key元素不会影响size),所以，上面的答案是 7。

注意2：看第 53 、 60 行代码：

```

1  afterNodeAccess(e);
2  afterNodeInsertion(evict);

```

这里调用的该方法，其实是调用了如下实现方法：

```
1 void afterNodeAccess(Node<K,V> p) { }
2 void afterNodeInsertion(boolean evict) { }
```

这都是一个空的方法实现，我们在这里可以不用管，但是在后面介绍 LinkedHashMap 会用到，LinkedHashMap 是继承的 HashMap，并且重写了该方法，后面我们会详细介绍。

## 8、扩容机制

扩容 (resize)，我们知道集合是由数组+链表+红黑树构成，向 HashMap 中插入元素时，如果 HashMap 集合的元素已经大于了最大承载容量 threshold ( $\text{capacity} * \text{loadFactor}$ )，这里的 threshold 不是数组的最大长度。那么必须扩大数组的长度，Java 中数组是无法自动扩容的，我们采用的方法是用一个更大的数组代替这个小的数组，就好比以前是用小桶装水，现在小桶装不下了，我们使用一个更大的桶。

JDK1.8 融入了红黑树的机制，比较复杂，这里我们先介绍 JDK1.7 的扩容源码，便于理解，然后在介绍 JDK1.8 的源码。

```
1 //参数 newCapacity 为新数组的大小
2 void resize(int newCapacity) {
3     Entry[] oldTable = table; //引用扩容前的 Entry 数组
4     int oldCapacity = oldTable.length;
5     if (oldCapacity == MAXIMUM_CAPACITY) { //扩容前的数组大小如果已经达到最大
6         // (2^30) 了
7         threshold = Integer.MAX_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了
8         return;
9     }
10    Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组
11    transfer(newTable, initHashSeedAsNeeded(newCapacity)); //将数组元素转移到新数组里面
12    table = newTable;
13    threshold = (int) Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1); //修改阈值
14 }
15 void transfer(Entry[] newTable, boolean rehash) {
16     int newCapacity = newTable.length;
17     for (Entry<K,V> e : table) { //遍历数组
18         while (null != e) {
19             Entry<K,V> next = e.next;
20             if (rehash) {
21                 e.hash = null == e.key ? 0 : hash(e.key);
22             }
23             int i = indexFor(e.hash, newCapacity); //重新计算每个元素在数组中的索引位置
24             e.next = newTable[i]; //标记下一个元素，添加是链表头添加
```

```

25         newTable[i] = e; //将元素放在链上
26         e = next; //访问下一个 Entry 链上的元素
27     }
28 }
29 }

```

通过方法我们可以看到，JDK1.7中首先是创建一个新的大容量数组，然后依次重新计算原集合所有元素的索引，然后重新赋值。如果数组某个位置发生了hash冲突，使用的是单链表的头插入方法，同一位置的新元素总是放在链表的头部，这样与原集合链表对比，扩容之后的可能就是倒序的链表了。

下面我们在看看JDK1.8的。

```

1  final Node<K,V>[] resize() {
2      Node<K,V>[] oldTab = table;
3      int oldCap = (oldTab == null) ? 0 : oldTab.length; //原数组如果为
null, 则长度赋值0
4      int oldThr = threshold;
5      int newCap, newThr = 0;
6      if (oldCap > 0) { //如果原数组长度大于0
7          if (oldCap >= MAXIMUM_CAPACITY) { //数组大小如果已经大于等于最大值
(2^30)
8              threshold = Integer.MAX_VALUE; //修改阈值为int的最大值(2^31-
1), 这样以后就不会扩容了
9              return oldTab;
10         }
11         //原数组长度大于等于初始化长度16, 并且原数组长度扩大1倍也小于2^30次方
12         else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
13             oldCap >= DEFAULT_INITIAL_CAPACITY)
14             newThr = oldThr << 1; // 阈值扩大1倍
15     }
16     else if (oldThr > 0) //旧阈值大于0, 则将新容量直接等于就阈值
17         newCap = oldThr;
18     else { //阈值等于0, oldCap也等于0 (集合未进行初始化)
19         newCap = DEFAULT_INITIAL_CAPACITY; //数组长度初始化为16
20         newThr = (int)(DEFAULT_LOAD_FACTOR *
DEFAULT_INITIAL_CAPACITY); //阈值等于16*0.75=12
21     }
22     //计算新的阈值上限
23     if (newThr == 0) {
24         float ft = (float)newCap * loadFactor;
25         newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
26             (int)ft : Integer.MAX_VALUE);
27     }
28     threshold = newThr;
29     @SuppressWarnings({"rawtypes","unchecked"})
30     Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
31     table = newTab;
32     if (oldTab != null) {

```

```

33         //把每个bucket都移动到新的buckets中
34         for (int j = 0; j < oldCap; ++j) {
35             Node<K,V> e;
36             if ((e = oldTab[j]) != null) {
37                 oldTab[j] = null; //元数据j位置置为null, 便于垃圾回收
38                 if (e.next == null) //数组没有下一个引用 (不是链表)
39                     newTab[e.hash & (newCap - 1)] = e;
40                 else if (e instanceof TreeNode) //红黑树
41                     ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
42                 else { // preserve order
43                     Node<K,V> loHead = null, loTail = null;
44                     Node<K,V> hiHead = null, hiTail = null;
45                     Node<K,V> next;
46                     do {
47                         next = e.next;
48                         //原索引
49                         if ((e.hash & oldCap) == 0) {
50                             if (loTail == null)
51                                 loHead = e;
52                             else
53                                 loTail.next = e;
54                             loTail = e;
55                         }
56                         //原索引+oldCap
57                         else {
58                             if (hiTail == null)
59                                 hiHead = e;
60                             else
61                                 hiTail.next = e;
62                             hiTail = e;
63                         }
64                     } while ((e = next) != null);
65                     //原索引放到bucket里
66                     if (loTail != null) {
67                         loTail.next = null;
68                         newTab[j] = loHead;
69                     }
70                     //原索引+oldCap放到bucket里
71                     if (hiTail != null) {
72                         hiTail.next = null;
73                         newTab[j + oldCap] = hiHead;
74                     }
75                 }
76             }
77         }
78     }
79     return newTab;
80 }

```

该方法分为两部分，首先是计算新桶数组的容量 newCap 和新阈值 newThr，然后将原集合的元素重新映射到新集合中。

第一个 if--else if--else 语句		
	<code>oldCap &gt;= 2<sup>30</sup></code>	修改阈值为int的最大值(2 <sup>31</sup> -1)，数组长度不变，不在进行扩容
<code>oldCap &gt; 0</code>		
	<code>newCap &lt; 2<sup>30</sup> &amp;&amp; oldCap &gt; 16</code>	新阈值 <code>newThr = oldThr &lt;&lt; 1</code> (扩大一倍)， <code>newCap = oldCap &lt;&lt; 1</code> ，移位可能会导致溢出
<code>oldThr &gt; 0</code>	threshold > 0，且桶数组未被初始化	调用 <code>HashMap(int)</code> 和 <code>HashMap(int, float)</code> 构造方法时会产生这种情况，此种情况下 <code>newCap = oldThr</code> ， <code>newThr</code> 在下面分支中算出
<code>oldCap == 0 &amp;&amp; oldThr == 0</code>	桶数组未被初始化，且 threshold 为 0	调用 <code>HashMap()</code> 构造方法会产生这种情况。 <code>newCap=16, newThr=0.75*16=12</code>
第二个 if 语句		
<code>newThr == 0</code>	第一个条件分支未计算 newThr 或嵌套分支在计算过程中导致 newThr 溢出归零	<code>newCap&lt;2<sup>30</sup>&amp;&amp;(newCap * 0.75)&lt;2<sup>30</sup></code> 成立则 <code>newThr=(newCap * 0.75)</code> 不成立则 <code>newThr=2<sup>31</sup>-1</code>
第三个if语句：扩容后重新分配元素		
桶数组某个位置元素.next=null	表示既不是链表，也不是红黑树，直接插入即可	<code>newTab[e.hash &amp; (newCap - 1)] = e;</code>
该位置为红黑树	<code>e instanceof TreeNode</code>	<code>split()</code> 方法重新分配
该位置为链表		不需要像JDK1.7重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”

相比于JDK1.7，1.8使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位，要么是在原位置再移动2次幂的位置。我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”。

## 9、删除元素

HashMap 删除元素首先是要找到 桶的位置，然后如果是链表，则进行链表遍历，找到需要删除的元素后，进行删除；如果是红黑树，也是进行树的遍历，找到元素删除后，进行平衡调节，注意，当红黑树的节点数小于 6 时，会转化成链表。

```
1 public V remove(Object key) {
2     Node<K,V> e;
3     return (e = removeNode(hash(key), key, null, false, true)) == null
4         ?
5         null : e.value;
6
7     }
8
9     final Node<K,V> removeNode(int hash, Object key, Object value,
10         boolean matchValue, boolean movable) {
```



```

9      Node<K,V>[] tab; Node<K,V> p; int n, index;
10     //(n - 1) & hash找到桶的位置
11     if ((tab = table) != null && (n = tab.length) > 0 &&
12         (p = tab[index = (n - 1) & hash]) != null) {
13         Node<K,V> node = null, e; K k; V v;
14         //如果键的值与链表第一个节点相等, 则将 node 指向该节点
15         if (p.hash == hash &&
16             ((k = p.key) == key || (key != null && key.equals(k))))
17             node = p;
18         //如果桶节点存在下一个节点
19         else if ((e = p.next) != null) {
20             //节点为红黑树
21             if (p instanceof TreeNode)
22                 node = ((TreeNode<K,V>)p).getTreeNode(hash, key); //找到需要删除的红
黑树节点
23         else {
24             do { //遍历链表, 找到待删除的节点
25                 if (e.hash == hash &&
26                     ((k = e.key) == key ||
27                     (key != null && key.equals(k)))) {
28                     node = e;
29                     break;
30                 }
31                 p = e;
32             } while ((e = e.next) != null);
33         }
34     }
35     //删除节点, 并进行调节红黑树平衡
36     if (node != null && (!matchValue || (v = node.value) == value ||
37         (value != null && value.equals(v)))) {
38         if (node instanceof TreeNode)
39             ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);
40         else if (node == p)
41             tab[index] = node.next;
42         else
43             p.next = node.next;
44         ++modCount;
45         --size;
46         afterNodeRemoval(node);
47         return node;
48     }
49 }
50 return null;
51 }

```

注意第 46 行代码

afterNodeRemoval(node);

这也是为实现 LinkedHashMap 做准备的，在这里和上面一样，是一个空方法实现，可以不用管。而在 LinkedHashMap 中进行了重写，用来维护删除节点后，链表的前后关系。

## 10、查找元素

### ①、通过 key 查找 value

首先通过 key 找到计算索引，找到桶位置，先检查第一个节点，如果是则返回，如果不是，则遍历其后面的链表或者红黑树。其余情况全部返回 null。

```
1 public V get(Object key) {
2     Node<K,V> e;
3     return (e = getNode(hash(key), key)) == null ? null : e.value;
4 }
5
6 final Node<K,V> getNode(int hash, Object key) {
7     Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
8     if ((tab = table) != null && (n = tab.length) > 0 &&
9         (first = tab[(n - 1) & hash]) != null) {
10        //根据key计算的索引检查第一个索引
11        if (first.hash == hash && // always check first node
12            ((k = first.key) == key || (key != null &&
13            key.equals(k))))
14            return first;
15        //不是第一个节点
16        if ((e = first.next) != null) {
17            if (first instanceof TreeNode) //遍历树查找元素
18                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
19            do {
20                //遍历链表查找元素
21                if (e.hash == hash &&
22                    ((k = e.key) == key || (key != null &&
23                    key.equals(k))))
24                    return e;
25            } while ((e = e.next) != null);
26        }
27        return null;
28 }
```

### ②、判断是否存在给定的 key 或者 value

```
1 public boolean containsKey(Object key) {
2     return getNode(hash(key), key) != null;
3 }
4 public boolean containsValue(Object value) {
5     Node<K,V>[] tab; V v;
6     if ((tab = table) != null && size > 0) {
7         //遍历桶
```

```

8         for (int i = 0; i < tab.length; ++i) {
9             //遍历桶中的每个节点元素
10            for (Node<K,V> e = tab[i]; e != null; e = e.next) {
11                if ((v = e.value) == value ||
12                    (value != null && value.equals(v)))
13                    return true;
14            }
15        }
16    }
17    return false;
18 }

```

## 11、遍历元素

首先构造一个 HashMap 集合：

```

1    HashMap<String,Object> map = new HashMap<>();
2    map.put("A", "1");
3    map.put("B", "2");
4    map.put("C", "3");

```

①、分别获取 key 集合和 value 集合。

```

1    //1、分别获取key和value的集合
2    for(String key : map.keySet()){
3        System.out.println(key);
4    }
5    for(Object value : map.values()){
6        System.out.println(value);
7    }

```

②、获取 key 集合，然后遍历key集合，根据key分别得到相应value

```

1    //2、获取key集合，然后遍历key，根据key得到 value
2    Set<String> keySet = map.keySet();
3    for(String str : keySet){
4        System.out.println(str+"-"+map.get(str));
5    }

```

③、得到 Entry 集合，然后遍历 Entry

```

1    //3、得到 Entry 集合，然后遍历 Entry
2    Set<Map.Entry<String,Object>> entrySet = map.entrySet();
3    for(Map.Entry<String,Object> entry : entrySet){
4        System.out.println(entry.getKey()+"-"+entry.getValue());
5    }

```

#### ④、迭代

```
1 //4、迭代
2 Iterator<Map.Entry<String,Object>> iterator = map.entrySet().iterator();
3 while(iterator.hasNext()){
4     Map.Entry<String,Object> mapEntry = iterator.next();
5     System.out.println(mapEntry.getKey()+"-"+mapEntry.getValue());
6 }
```

基本上使用第三种方法是性能最好的,

第一种遍历方法在我们只需要 key 集合或者只需要 value 集合时使用;

第二种方法效率很低, 不推荐使用;

第四种方法效率也挺好, 关键是在遍历的过程中我们可以对集合中的元素进行删除。

## 12、总结

①、基于JDK1.8的HashMap是由数组+链表+红黑树组成, 当链表长度超过 8 时会自动转换成红黑树, 当红黑树节点个数小于 6 时, 又会转化成链表。相对于早期版本的JDK HashMap 实现, 新增了红黑树作为底层数据结构, 在数据量较大且哈希碰撞较多时, 能够极大的增加检索的效率。

②、允许 key 和 value 都为 null。key 重复会被覆盖, value 允许重复。

③、非线程安全

④、无序 (遍历HashMap得到元素的顺序不是按照插入的顺序)

参考文档:

<https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html#>

<http://www.importnew.com/20386.html>

<https://www.cnblogs.com/nullllun/p/8327664.html>

本系列教程持续更新, 可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



微信搜一搜

IT可乐

## JDK源码解析(8)——java.util.HashSet 类

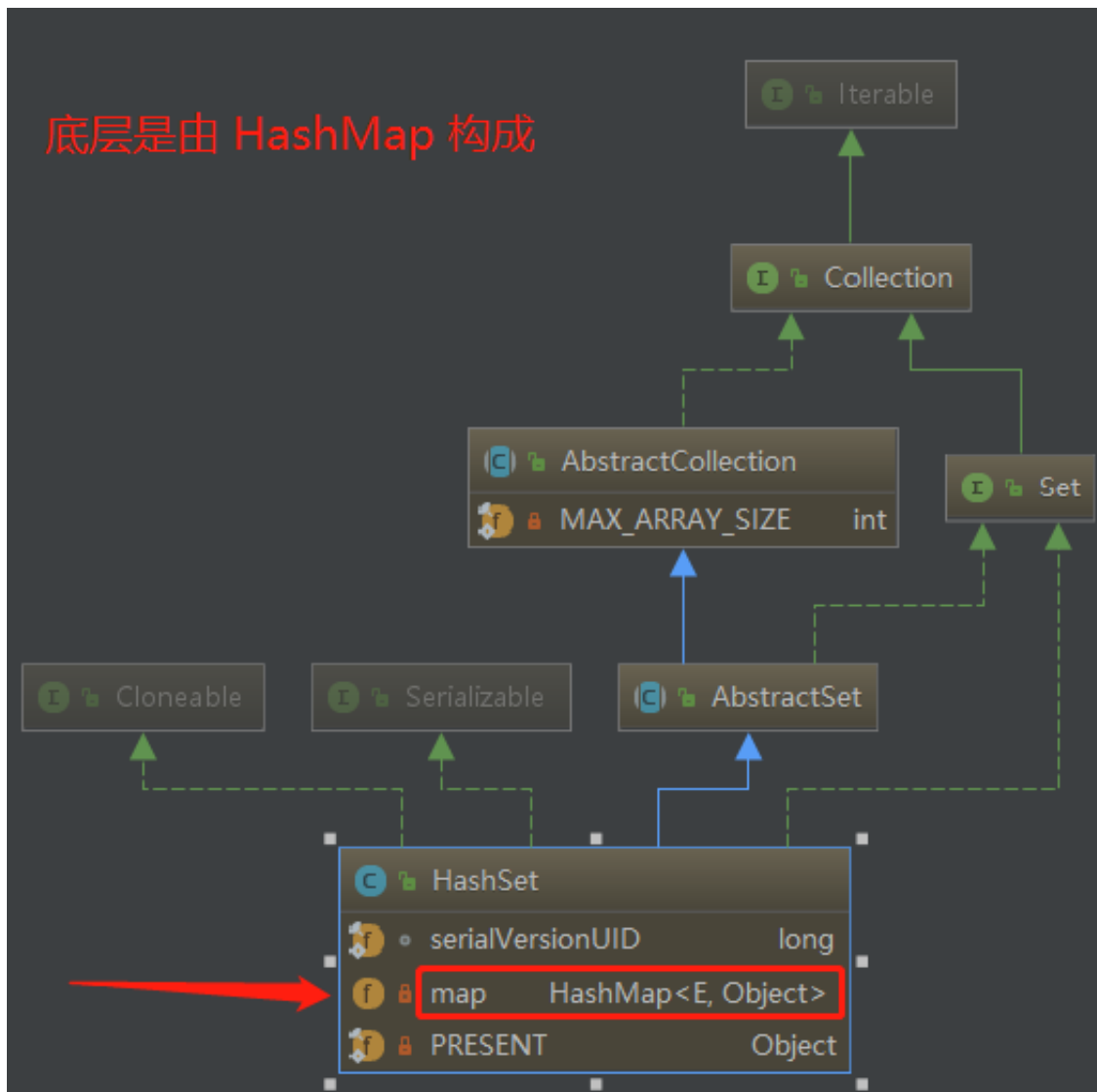
在上一篇博客，我们介绍了 Map 集合的一种典型实现 [HashMap](#)，在 JDK1.8 中，HashMap 是由数组+链表+红黑树构成，相对于早期版本的 JDK HashMap 实现，新增了红黑树作为底层数据结构，在数据量较大且哈希碰撞较多时，能够极大的增加检索的效率。了解 HashMap 的具体实现后，我们再来介绍由 HashMap 作为底层数据结构实现的一种数据结构——HashSet。（如果不了解 HashMap 的实现原理，建议先看看 HashMap，不然直接看 HashSet 是很难看懂的）。

### 1、HashSet 定义

HashSet 是一个由 HashMap 实现的集合。元素无序且不能重复。

```
1 public class HashSet<E>
2     extends AbstractSet<E>
3     implements Set<E>, Cloneable, java.io.Serializable
```

底层是由 HashMap 构成



和前面介绍的大多数集合一样，HashSet 也实现了 Cloneable 接口和 Serializable 接口，分别用来支持克隆以及支持序列化。还实现了 Set 接口，该接口定义了 Set 集合类型的一套规范。

## 2、字段属性

```
1 //HashSet集合中的内容是通过 HashMap 数据结构来存储的
2 private transient HashMap<E, Object> map;
3 //向HashSet中添加数据，数据在上面的 map 结构是作为 key 存在的，而value统一都是
  PRESENT
4 private static final Object PRESENT = new Object();
```

第一个定义一个 HashMap，作为实现 HashSet 的数据结构；第二个 PRESENT 对象，因为前面讲过 HashMap 是作为键值对 key-value 进行存储的，而 HashSet 不是键值对，那么选择 HashMap 作为实现，其原理就是存储在 HashSet 中的数据 作为 Map 的 key，而 Map 的 value 统一为 PRESENT（下面介绍具体实现时会了解）。

## 3、构造函数

### ①、无参构造

```
1 public HashSet() {  
2     map = new HashMap<>();  
3 }
```

直接 new 一个 HashSet 对象出来，采用无参的 HashSet 构造函数，具有默认初始容量（16）和加载因子（0.75）。

### ②、指定初始容量

```
1 public HashSet(int initialCapacity) {  
2     map = new HashMap<>(initialCapacity);  
3 }
```

### ③、指定初始容量和加载因子

```
1 public HashSet(int initialCapacity, float loadFactor) {  
2     map = new HashMap<>(initialCapacity, loadFactor);  
3 }
```

### ④、构造包含指定集合中的元素

```
1 public HashSet(Collection<? extends E> c) {  
2     map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));  
3     addAll(c);  
4 }
```

集合容量很好理解，这里我介绍一下什么是加载因子。在 HashSet 中，能够存储元素的数量就是：总的容量\*加载因子，新增一个元素时，如果HashSet集合中的元素大于前面公式计算的结果了，那么就必须要进行扩容操作，从时间和空间考虑，加载因子一般都选默认的0.75。

## 4、添加元素

```
1 public boolean add(E e) {  
2     return map.put(e, PRESENT)!=null;  
3 }
```

通过 map.put() 方法来添加元素，在上一篇博客介绍该方法时，说明了该方法如果新插入的key不存在，则返回null，如果新插入的key存在，则返回原key对应的value值（注意新插入的value会覆盖原value值）。

也就是说 HashSet 的 add(E e) 方法，会将 e 作为 key，PRESENT 作为 value 插入到 map 集合中，如果 e 不存在，则插入成功返回 true;如果存在，则返回false。

## 5、删除元素

```
1 public boolean remove(Object o) {
2     return map.remove(o) == PRESENT;
3 }
```

调用 HashMap 的 remove(Object o) 方法，该方法会首先查找 map 集合中是否存在 o，如果存在则删除，并返回该值，如果不存在则返回 null。

也就是说 HashSet 的 remove(Object o) 方法，删除成功返回 true，删除的元素不存在会返回 false。

## 6、查找元素

```
1 public boolean contains(Object o) {
2     return map.containsKey(o);
3 }
```

调用 HashMap 的 containsKey(Object o) 方法，找到了返回 true，找不到返回 false。

## 7、遍历元素

```
1 HashSet<Integer> set = new HashSet<>();
2 set.add(1);
3 set.add(2);
4 //增强for循环
5 for(Integer i : set){
6     System.out.println(i);
7 }
8 //普通for循环
9 Iterator<Integer> iterator = set.iterator();
10 while (iterator.hasNext()){
11     System.out.println(iterator.next());
12 }
```

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料





微信搜一搜

IT可乐

## JDK源码解析(9)——java.util.LinkedHashMap 类

前面我们介绍了 Map 集合的一种典型实现 HashMap，关于 HashMap 的特性，我们再来复习一遍：

①、基于JDK1.8的HashMap是由数组+链表+红黑树组成，相对于早期版本的JDK HashMap 实现，新增了红黑树作为底层数据结构，在数据量较大且哈希碰撞较多时，能够极大的增加检索的效率。

②、允许 key 和 value 都为 null。key 重复会被覆盖，value 允许重复。

③、非线程安全

④、无序（遍历HashMap得到元素的顺序不是按照插入的顺序）

HashMap 集合可以说是最重要的集合之一，上篇博客介绍的 HashSet 集合就是继承 HashMap 来实现的。而本篇博客我们介绍 Map 集合的另一种实现——LinkedHashMap，其实也是继承 HashMap 集合来实现的，而且我们在介绍 HashMap 集合的 put 方法时，也指出了 put 方法中调用的部分方法在 HashMap 都是空实现，而在 LinkedHashMap 中进行了重写。所以想要彻底了解 LinkedHashMap 的实现原理，HashMap 的实现原理一定不能不懂。

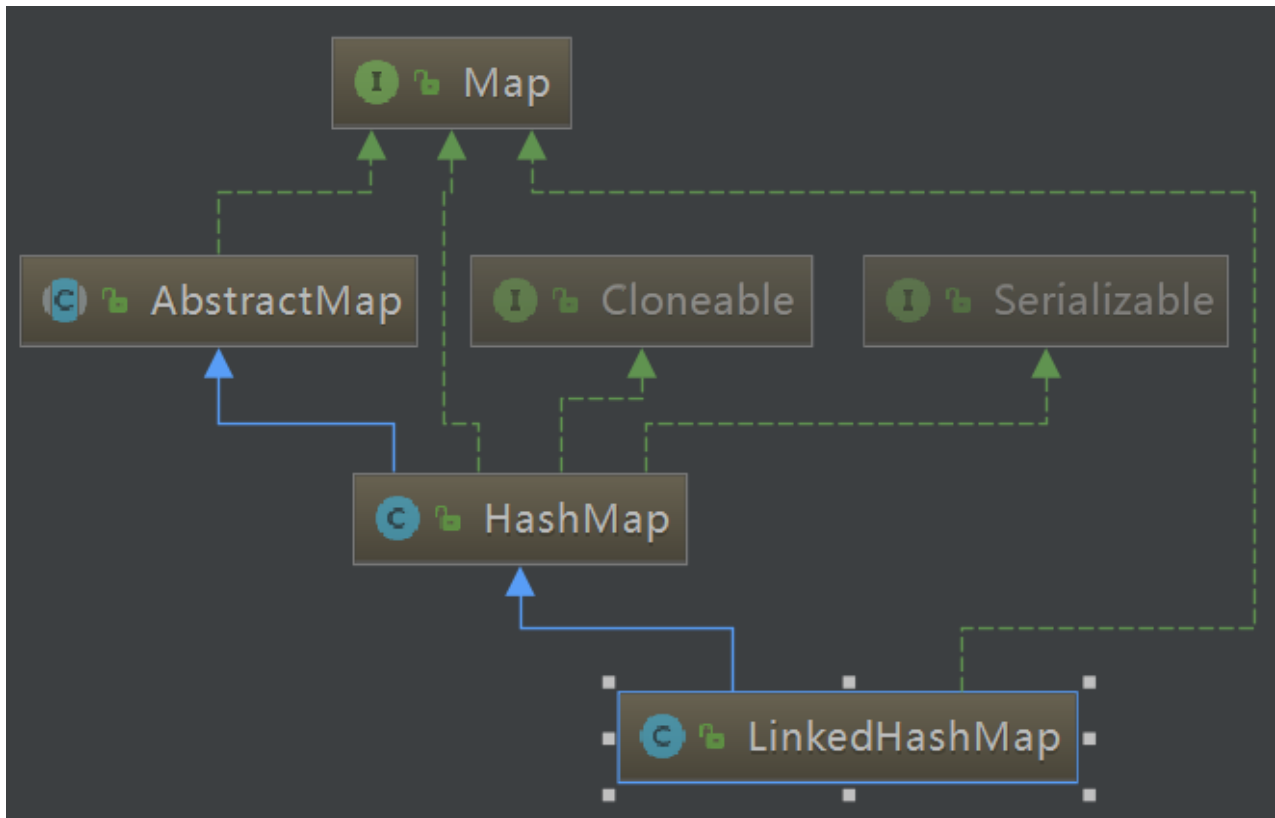
### 1、LinkedHashMap 定义

LinkedHashMap 是基于 HashMap 实现的一种集合，具有 HashMap 集合上面所说的所有特点，除了 HashMap 无序的特点，LinkedHashMap 是有序的，因为 LinkedHashMap 在 HashMap 的基础上单独维护了一个具有所有数据的双向链表，该链表保证了元素迭代的顺序。

所以我们可以直接这样说：LinkedHashMap = HashMap + LinkedList。LinkedHashMap 就是在 HashMap 的基础上多维护了一个双向链表，用来保证元素迭代顺序。

更形象化的图形展示可以直接移到文章末尾。

```
1 public class LinkedHashMap<K,V>
2     extends HashMap<K,V>
3     implements Map<K,V>
```



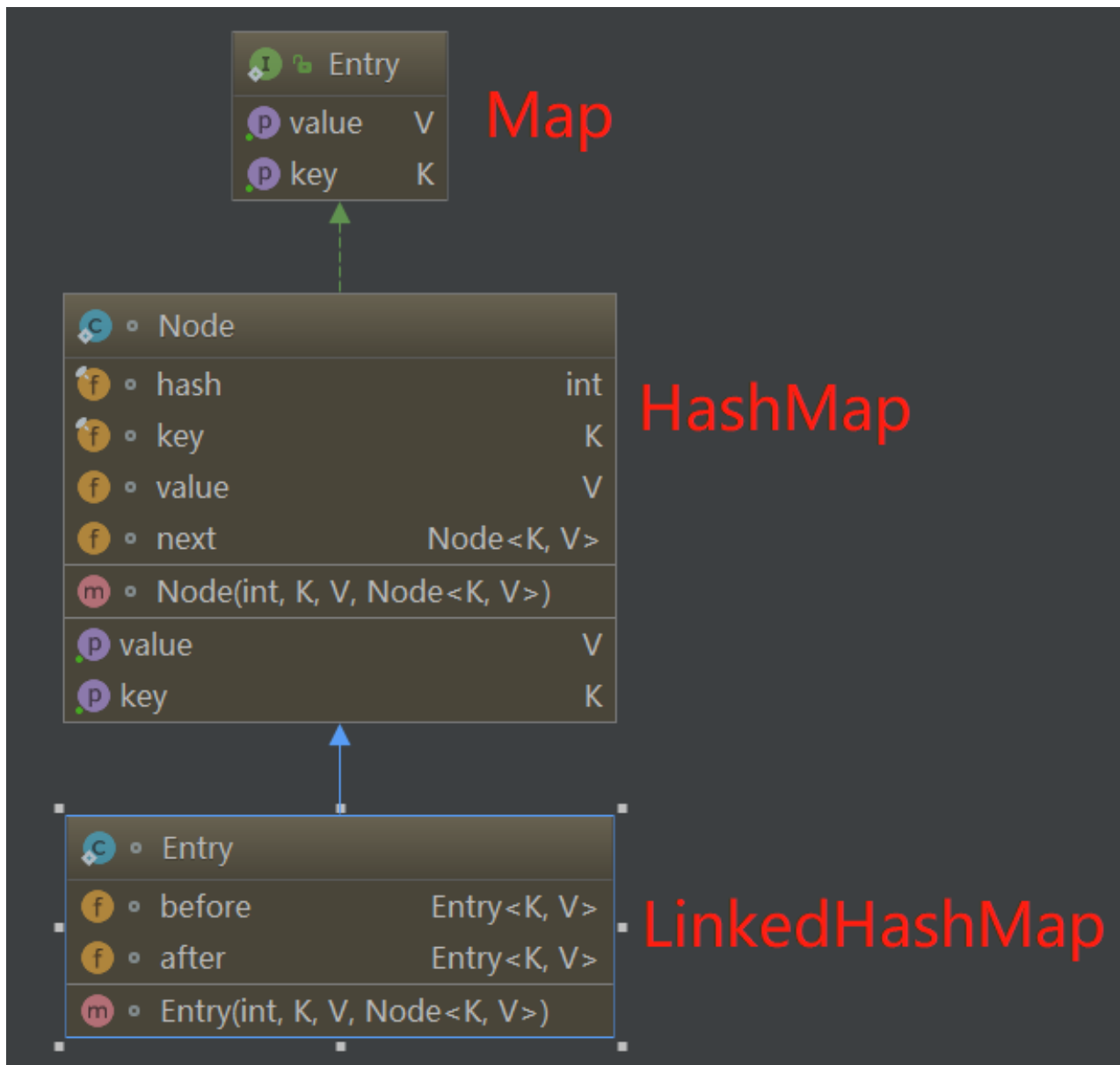
## 2、字段属性

### ①、Entry<K,V>

```
1 static class Entry<K,V> extends HashMap.Node<K,V> {
2     Entry<K,V> before, after;
3     Entry(int hash, K key, V value, Node<K,V> next) {
4         super(hash, key, value, next);
5     }
6 }
```

LinkedHashMap 的每个元素都是一个 Entry，我们看到对于 Entry 继承自 HashMap 的 Node 结构，相对于 Node 结构，LinkedHashMap 多了 before 和 after 结构。

下面是Map类集合基本元素的实现演变。



LinkedHashMap 中 Entry 相对于 HashMap 多出的 before 和 after 便是用来维护 LinkedHashMap 插入 Entry 的先后顺序的。

## ②、其它属性

```
1 //用来指向双向链表的头节点
2 transient LinkedHashMap.Entry<K,V> head;
3 //用来指向双向链表的尾节点
4 transient LinkedHashMap.Entry<K,V> tail;
5 //用来指定LinkedHashMap的迭代顺序
6 //true 表示按照访问顺序，会把访问过的元素放在链表后面，放置顺序是访问的顺序
7 //false 表示按照插入顺序遍历
8 final boolean accessOrder;
```

**注意：**这里有五个属性别搞混淆的，对于 Node next 属性，是用来维护整个集合中 Entry 的顺序。对于 Entry before, Entry after, 以及 Entry head, Entry tail, 这四个属性都是用来维护保证集合顺序的链表，其中前两个 before 和 after 表示某个节点的上一个节点和下一个节点，这是一个双向链表。后两个属性 head 和 tail 分别表示这个链表的头节点和尾节点。

### 3、构造函数

#### ①、无参构造

```
1 public LinkedHashMap() {  
2     super();  
3     accessOrder = false;  
4 }
```

调用无参的 HashMap 构造函数，具有默认初始容量（16）和加载因子（0.75）。并且设定了 accessOrder = false，表示默认按照插入顺序进行遍历。

#### ②、指定初始容量

```
1 public LinkedHashMap(int initialCapacity) {  
2     super(initialCapacity);  
3     accessOrder = false;  
4 }
```

#### ③、指定初始容量和加载因子

```
1 public LinkedHashMap(int initialCapacity, float loadFactor) {  
2     super(initialCapacity, loadFactor);  
3     accessOrder = false;  
4 }
```

#### ④、指定初始容量和加载因子，以及迭代规则

```
1 public LinkedHashMap(int initialCapacity,  
2                       float loadFactor,  
3                       boolean accessOrder) {  
4     super(initialCapacity, loadFactor);  
5     this.accessOrder = accessOrder;  
6 }
```

#### ⑤、构造包含指定集合中的元素

```
1 public LinkedHashMap(Map<? extends K, ? extends V> m) {  
2     super();  
3     accessOrder = false;  
4     putMapEntries(m, false);  
5 }
```

上面所有的构造函数默认 accessOrder = false，除了第四个构造函数能够指定 accessOrder 的值。

## 4、添加元素

LinkedHashMap 中是没有 put 方法的，直接调用父类 HashMap 的 put 方法。关于 HashMap 的 put 方法，可以参看我对于 HashMap 的介绍。

我将方法介绍复制到下面：

```
1 //hash(key)就是上面讲的hash方法，对其进行了第一步和第二步处理
2 public V put(K key, V value) {
3     return putVal(hash(key), key, value, false, true);
4 }
5 /**
6  *
7  * @param hash 索引的位置
8  * @param key 键
9  * @param value 值
10  * @param onlyIfAbsent true 表示不要更改现有值
11  * @param evict false表示table处于创建模式
12  * @return
13  */
14 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
15               boolean evict) {
16     Node<K,V>[] tab; Node<K,V> p; int n, i;
17     //如果table为null或者长度为0，则进行初始化
18     //resize()方法本来是用于扩容，由于初始化没有实际分配空间，这里用该方法进行空
    间分配，后面会详细讲解该方法
19     if ((tab = table) == null || (n = tab.length) == 0)
20         n = (tab = resize()).length;
21     //注意：这里用到了前面讲解获得key的hash码的第三步，取模运算，下面的if-else
    分别是 tab[i] 为null和不为null
22     if ((p = tab[i = (n - 1) & hash]) == null)
23         tab[i] = newNode(hash, key, value, null); //tab[i] 为null，直接
    将新的key-value插入到计算的索引i位置
24     else { //tab[i] 不为null，表示该位置已经有值了
25         Node<K,V> e; K k;
26         if (p.hash == hash &&
27             ((k = p.key) == key || (key != null && key.equals(k))))
28             e = p; //节点key已经有值了，直接用新值覆盖
29         //该链是红黑树
30         else if (p instanceof TreeNode)
31             e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
    value);
32         //该链是链表
33         else {
34             for (int binCount = 0; ; ++binCount) {
35                 if ((e = p.next) == null) {
36                     p.next = newNode(hash, key, value, null);
37                     //链表长度大于8，转换成红黑树
```

```

38         if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for
1st
39             treeifyBin(tab, hash);
40             break;
41         }
42         //key已经存在直接覆盖value
43         if (e.hash == hash &&
44             ((k = e.key) == key || (key != null &&
key.equals(k))))
45             break;
46             p = e;
47         }
48     }
49     if (e != null) { // existing mapping for key
50         V oldValue = e.value;
51         if (!onlyIfAbsent || oldValue == null)
52             e.value = value;
53         afterNodeAccess(e);
54         return oldValue;
55     }
56 }
57 ++modCount; //用作修改和新增快速失败
58 if (++size > threshold) //超过最大容量，进行扩容
59     resize();
60 afterNodeInsertion(evict);
61 return null;
62 }

```

这里主要介绍上面方法中，为了保证 LinkedHashMap 的迭代顺序，在添加元素时重写了的4个方法，分别是第23行、31行以及53、60行代码：

```

1  newNode(hash, key, value, null);
2  putTreeVal(this, tab, hash, key, value) //newTreeNode(h, k, v, xpn)
3  afterNodeAccess(e);
4  afterNodeInsertion(evict);

```

#### ①、对于 newNode(hash,key,value,null) 方法

```

1  HashMap.Node<K,V> newNode(int hash, K key, V value, HashMap.Node<K,V> e) {
2      LinkedHashMap.Entry<K,V> p =
3          new LinkedHashMap.Entry<K,V>(hash, key, value, e);
4      linkNodeLast(p);
5      return p;
6  }
7
8  private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {
9      //用临时变量last记录尾节点tail
10     LinkedHashMap.Entry<K,V> last = tail;

```

```

11      //将尾节点设为当前插入的节点p
12      tail = p;
13      //如果原先尾节点为null，表示当前链表为空
14      if (last == null)
15          //头结点也为当前插入节点
16          head = p;
17      else {
18          //原始链表不为空，那么将当前节点的上节点指向原始尾节点
19          p.before = last;
20          //原始尾节点的下一个节点指向当前插入节点
21          last.after = p;
22      }
23  }

```

也就是说将当前添加的元素设为原始链表的尾节点。

## ②、对于 putTreeVal 方法

是在添加红黑树节点时的操作，LinkedHashMap 也重写了该方法的 newTreeNode 方法：

```

1      TreeNode<K,V> newTreeNode(int hash, K key, V value, Node<K,V> next) {
2          TreeNode<K,V> p = new TreeNode<K,V>(hash, key, value, next);
3          linkNodeLast(p);
4          return p;
5      }

```

也就是说上面两个方法都是在将新添加的元素放置到链表的尾端，并维护链表节点之间的关系。

③、对于 afterNodeAccess(e) 方法，在 putVal 方法中，是当添加数据键值对的 key 存在时，会对 value 进行替换。然后调用 afterNodeAccess(e) 方法：

```

1  //把当前节点放到双向链表的尾部
2  void afterNodeAccess(HashMap.Node<K,V> e) { // move node to last
3      LinkedHashMap.Entry<K,V> last;
4      //当 accessOrder = true 并且当前节点不等于尾节点tail。这里将last节点赋值为
      tail节点
5      if (accessOrder && (last = tail) != e) {
6          //记录当前节点的上一个节点b和下一个节点a
7          LinkedHashMap.Entry<K,V> p =
8              (LinkedHashMap.Entry<K,V>)e, b = p.before, a =
p.after;
9          //释放当前节点和后一个节点的关系
10         p.after = null;
11         //如果当前节点的前一个节点为null
12         if (b == null)
13             //头节点=当前节点的下一个节点
14             head = a;
15         else
16             //否则b的后节点指向a

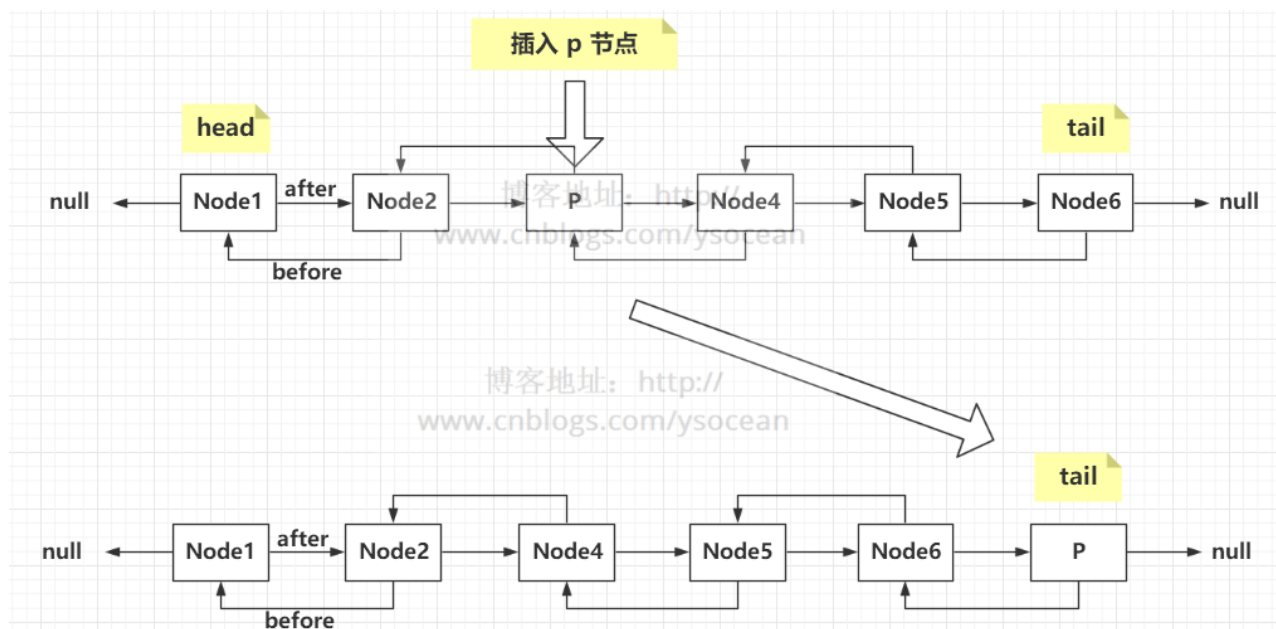
```

```

17         b.after = a;
18         //如果a != null
19         if (a != null)
20             //a的前一个节点指向b
21             a.before = b;
22         else
23             //b设为尾节点
24             last = b;
25         //如果尾节点为null
26         if (last == null)
27             //头节点设为p
28             head = p;
29         else {
30             //否则将p放到双向链表的最后
31             p.before = last;
32             last.after = p;
33         }
34         //将尾节点设为p
35         tail = p;
36         //LinkedHashMap对象操作次数+1，用于快速失败校验
37         ++modCount;
38     }
39 }

```

该方法是在 `accessOrder = true` 并且 插入的当前节点不等于尾节点时，该方法才会生效。并且该方法的作用是将插入的节点变为尾节点，后面在`get`方法中也会调用。代码实现可能有点绕，可以借助下图来理解：



④、在看 `afterNodeInsertion(evict)` 方法



```

1 void afterNodeInsertion(boolean evict) { // possibly remove eldest
2     LinkedHashMap.Entry<K,V> first;
3     if (evict && (first = head) != null && removeEldestEntry(first)) {
4         K key = first.key;
5         removeNode(hash(key), key, null, false, true);
6     }
7 }

```

该方法用来移除最老的首节点，首先方法要能执行到if语句里面，必须 evict = true，并且 头节点不为null，并且 removeEldestEntry(first) 返回true，这三个条件必须同时满足，前面两个好理解，我们看最后这个方法条件：

```

1     protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
2         return false;
3     }

```

这就奇怪了，该方法直接返回的是 false，也就是说怎么都不会进入到 if 方法体内了，那这是怎么回事呢？

这其实是用来实现 LRU（Least Recently Used，最近最少使用）Cache 时，重写的一个方法。比如在 mybatis-connector 包中，有这样一个类：

```

1 package com.mysql.jdbc.util;
2
3 import java.util.LinkedHashMap;
4 import java.util.Map.Entry;
5
6 public class LRUCache<K, V> extends LinkedHashMap<K, V> {
7     private static final long serialVersionUID = 1L;
8     protected int maxElements;
9
10    public LRUCache(int maxSize) {
11        super(maxSize, 0.75F, true);
12        this.maxElements = maxSize;
13    }
14
15    protected boolean removeEldestEntry(Entry<K, V> eldest) {
16        return this.size() > this.maxElements;
17    }
18 }

```

可以看到，它重写了 removeEldestEntry(Entry<K,V> eldest) 方法，当元素的个数大于设定的最大个数，便移除首元素。

## 5、删除元素

同理也是调用 HashMap 的 remove 方法，这里我不作过多的讲解，着重看 LinkedHashMap 重写的第 46 行方法。

```
1 public V remove(Object key) {
2     Node<K,V> e;
3     return (e = removeNode(hash(key), key, null, false, true)) == null
4     ?
5         null : e.value;
6 }
7
8 final Node<K,V> removeNode(int hash, Object key, Object value,
9     boolean matchValue, boolean movable) {
10     Node<K,V>[] tab; Node<K,V> p; int n, index;
11     //(n - 1) & hash找到桶的位置
12     if ((tab = table) != null && (n = tab.length) > 0 &&
13         (p = tab[index = (n - 1) & hash]) != null) {
14         Node<K,V> node = null, e; K k; V v;
15         //如果键的值与链表第一个节点相等，则将 node 指向该节点
16         if (p.hash == hash &&
17             ((k = p.key) == key || (key != null && key.equals(k))))
18             node = p;
19         //如果桶节点存在下一个节点
20         else if ((e = p.next) != null) {
21             //节点为红黑树
22             if (p instanceof TreeNode)
23                 node = ((TreeNode<K,V>)p).getTreeNode(hash, key); //找到需要删除的红
24                 黑树节点
25             else {
26                 do { //遍历链表，找到待删除的节点
27                     if (e.hash == hash &&
28                         ((k = e.key) == key ||
29                             (key != null && key.equals(k)))) {
30                         node = e;
31                         break;
32                     }
33                     p = e;
34                 } while ((e = e.next) != null);
35             }
36             //删除节点，并进行调节红黑树平衡
37             if (node != null && (!matchValue || (v = node.value) == value ||
38                 (value != null && value.equals(v)))) {
39                 if (node instanceof TreeNode)
40                     ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);
41                 else if (node == p)
42                     tab[index] = node.next;
```

```

42         else
43             p.next = node.next;
44             ++modCount;
45             --size;
46             afterNodeRemoval(node);
47             return node;
48         }
49     }
50     return null;
51 }

```

我们看第 46 行代码实现：

```

1  void afterNodeRemoval(HashMap.Node<K,V> e) { // unlink
2      LinkedHashMap.Entry<K,V> p =
3          (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
4      p.before = p.after = null;
5      if (b == null)
6          head = a;
7      else
8          b.after = a;
9      if (a == null)
10         tail = b;
11     else
12         a.before = b;
13 }

```

该方法其实很好理解，就是当我们删除某个节点时，为了保证链表还是有序的，那么必须维护其前后节点。而该方法的作用就是维护删除节点的前后节点关系。

## 6、查找元素

```

1  public V get(Object key) {
2      Node<K,V> e;
3      if ((e = getNode(hash(key), key)) == null)
4          return null;
5      if (accessOrder)
6          afterNodeAccess(e);
7      return e.value;
8  }

```

相比于 HashMap 的 get 方法，这里多出了第 5,6 行代码，当 accessOrder = true 时，即表示按照最近访问的迭代顺序，会将访问过的元素放在链表后面。

对于 afterNodeAccess(e) 方法，在前面第 4 小节 添加元素已经介绍过了，这就不在介绍。

## 7、遍历元素

在介绍 HashMap 时，我们介绍了 4 中遍历方式，同理，对于 LinkedHashMap 也有 4 种，这里我们介绍效率较高的两种遍历方式：

①、得到 Entry 集合，然后遍历 Entry

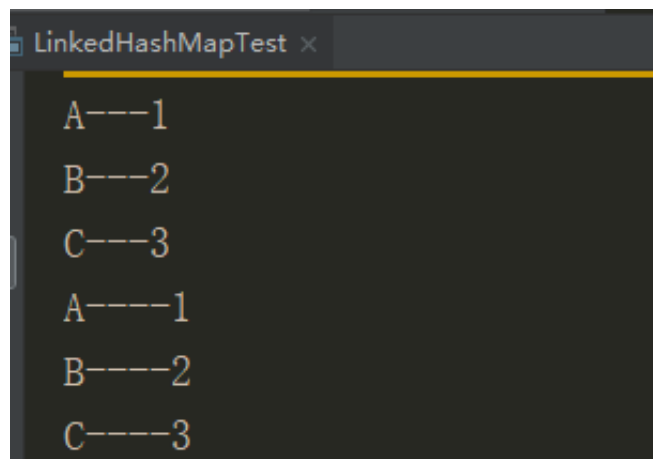
```
1 LinkedHashMap<String,String> map = new LinkedHashMap<>();
2     map.put("A","1");
3     map.put("B","2");
4     map.put("C","3");
5     map.get("B");
6     Set<Map.Entry<String,String>> entrySet = map.entrySet();
7     for(Map.Entry<String,String> entry : entrySet ){
8         System.out.println(entry.getKey()+"---"+entry.getValue());
9     }
```

②、迭代

```
1     Iterator<Map.Entry<String,String>> iterator =
map.entrySet().iterator();
2     while(iterator.hasNext()){
3         Map.Entry<String,String> entry = iterator.next();
4         System.out.println(entry.getKey()+"----"+entry.getValue());
5     }
```

这两种效率都还不错，通过迭代的方式可以对一边遍历一边删除元素，而第一种删除元素会报错。

打印结果：



```
LinkedHashMapTest x
A---1
B---2
C---3
A----1
B----2
C----3
```

## 8、迭代器

我们把上面遍历的 LinkedHashMap 构造函数改成下面的：

```
LinkedHashMap<String,String> map = new LinkedHashMap<>(16,0.75F,true);
```

也就是说将 `accessOrder = true`，表示按照访问顺序来遍历，注意看上面的 第 5 行代码：`map.get("B")`。也就是说设置 `accessOrder = true` 之后，那么 B---2 应该是最后输出，我们看一下打印结果：

```
"C:\Program Files\Java\jdk
A----1
C----3
B----2
```

结果跟预期一致。那么在遍历的过程中，`LinkedHashMap` 是如何进行的呢？

我们追溯源码：首先进入到 `map.entrySet()` 方法里面：

```
public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> es;
    return (es = entrySet) == null ? (entrySet = new LinkedEntrySet()) : es;
}
```

发现 `entrySet = new LinkedEntrySet()`，接下来我们查看 `LinkedEntrySet` 类。

```
final class LinkedEntrySet extends AbstractSet<Map.Entry<K,V>> {
    public final int size() { return size; }
    public final void clear() { LinkedHashMap.this.clear(); }
    public final Iterator<Map.Entry<K,V>> iterator() {
        return new LinkedEntryIterator();
    }
}
```

这是一个内部类，我们查看其 `iterator()` 方法，发现又 `new` 了一个新对象 `LinkedEntryIterator`，接着看这个类：

```
final class LinkedEntryIterator extends LinkedHashMapIterator
    implements Iterator<Map.Entry<K,V>> {
    public final Map.Entry<K,V> next() { return nextNode(); }
}
```

这个类继承 `LinkedHashMapIterator`。

```
1  abstract class LinkedHashMapIterator {
2      LinkedHashMap.Entry<K,V> next;
3      LinkedHashMap.Entry<K,V> current;
4      int expectedModCount;
5
6      LinkedHashMapIterator() {
7          next = head;
8          expectedModCount = modCount;
9          current = null;
10     }
```

```

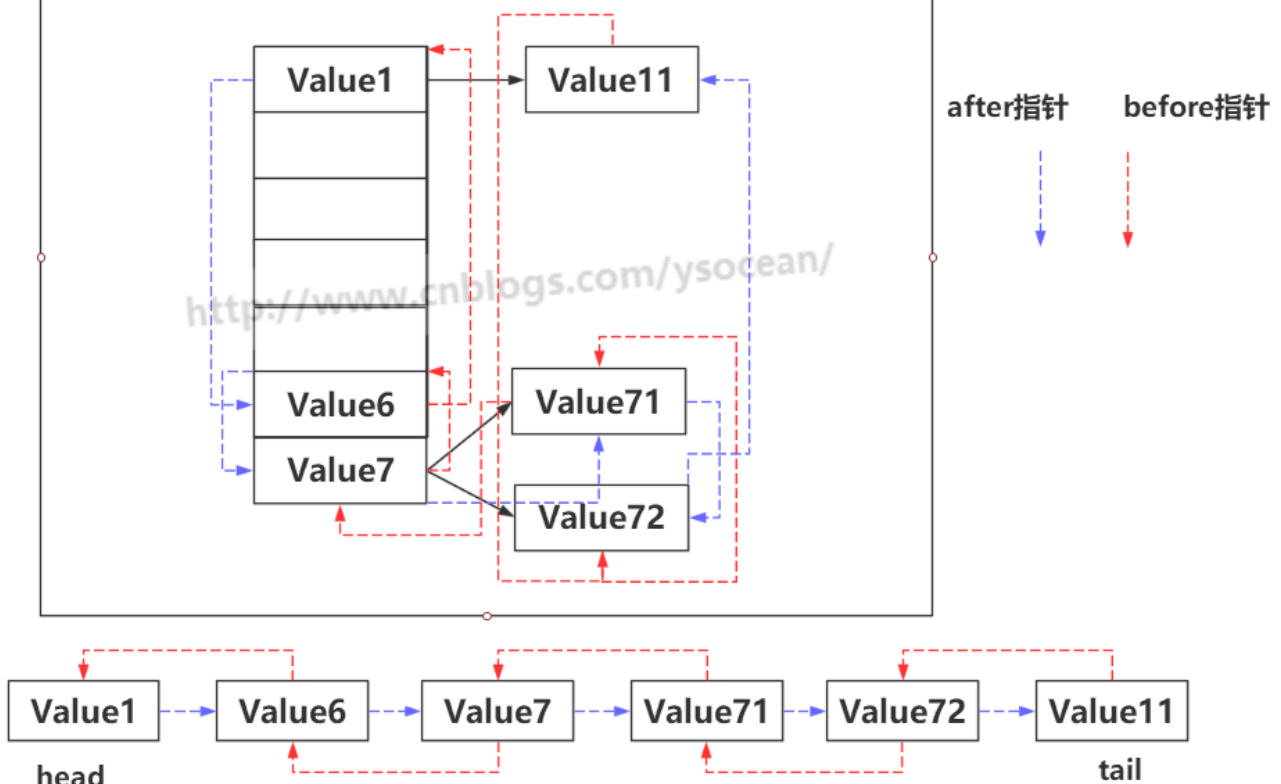
11
12     public final boolean hasNext() {
13         return next != null;
14     }
15
16     final LinkedHashMap.Entry<K,V> nextNode() {
17         LinkedHashMap.Entry<K,V> e = next;
18         if (modCount != expectedModCount)
19             throw new ConcurrentModificationException();
20         if (e == null)
21             throw new NoSuchElementException();
22         current = e;
23         next = e.after;
24         return e;
25     }
26
27     public final void remove() {
28         HashMap.Node<K,V> p = current;
29         if (p == null)
30             throw new IllegalStateException();
31         if (modCount != expectedModCount)
32             throw new ConcurrentModificationException();
33         current = null;
34         K key = p.key;
35         removeNode(hash(key), key, null, false, false);
36         expectedModCount = modCount;
37     }
38 }

```

## 9、总结

通过上面的介绍，关于 LinkedHashMap，我想直接用下面一幅图来解释：

## JDK1.8:LinkedHashMap



去掉红色和蓝色的虚线指针，其实就是一个HashMap。

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



微信搜一搜

IT可乐

## JDK源码解析(10)——java.util.LinkedHashSet类

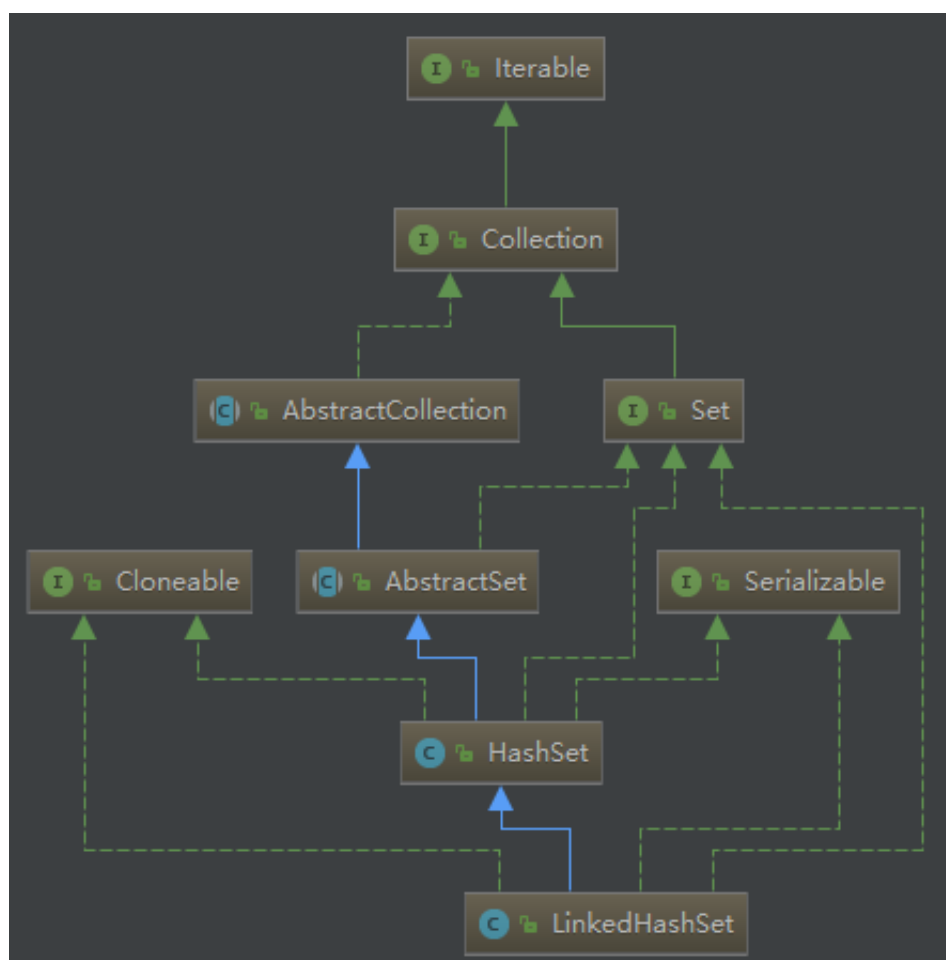
同 HashSet 与 HashMap 的关系一样，本篇博客所介绍的 LinkedHashSet 和 LinkedHashMap 也是一致的。在 JDK 集合框架中，类似 Set 集合通常都是由对应的 Map 类集合来实现的（TreeSet 和 TreeMap 同理），这里很重要的一个理论就是：Set 类集合是不允许重复的，而 Map 类集合的 key 也是不允许重复的，所以通常很容易就用 Map 类集合实现了 Set 类集合。

本篇博客之前，我们已经详细介绍了 HashSet、HashMap、LinkedHashMap。在此基础上，再来理解 LinkedHashSet 就很容易了。

## 1、LinkedHashSet 定义

LinkedHashSet 是由 LinkedHashMap 实现的集合。元素有序且不能重复。

```
1 public class LinkedHashSet<E>
2     extends HashSet<E>
3     implements Set<E>, Cloneable, java.io.Serializable {
```



看上图类定义，LinkedHashSet 是由 HashSet 来实现的，其实底层是通过 LinkedHashMap 来实现的。

## 2、构造函数

在 LinkedHashSet 中，有如下几个构造方法：

- ①、指定初始容量和加载因子



```

1      public LinkedHashSet(int initialCapacity, float loadFactor) {
2          super(initialCapacity, loadFactor, true);
3      }

```

## ②、指定初始容量

```

1      public LinkedHashSet(int initialCapacity) {
2          super(initialCapacity, .75f, true);
3      }

```

## ③、默认无参构造函数

```

1      public LinkedHashSet() {
2          super(16, .75f, true);
3      }

```

## ④、构造包含指定集合的元素

```

1      public LinkedHashSet(Collection<? extends E> c) {
2          super(Math.max(2*c.size(), 11), .75f, true);
3          addAll(c);
4      }

```

上面所有的构造方法，都调用父类，也就是 HashSet 的 `super(initialCapacity, loadFactor, true);`

```

1      HashSet(int initialCapacity, float loadFactor, boolean dummy) {
2          map = new LinkedHashMap<>(initialCapacity, loadFactor);
3      }

```

前面两个参数分别设置HashMap 的初始容量和加载因子。dummy 可以忽略掉，这个参数只是为了区分 HashSet 别的构造方法。

## 3、添加元素

```

1      public boolean add(E e) {
2          return map.put(e, PRESENT)!=null;
3      }

```

通过 `map.put()` 方法来添加元素，说明了该方法如果新插入的key不存在，则返回null，如果新插入的key存在，则返回原key对应的value值（注意新插入的value会覆盖原value值）。

也就是说 `add(E e)` 方法，会将 `e` 作为 key，`PRESENT` 作为 value 插入到 `map` 集合中，如果 `e` 不存在，则插入成功返回 `true`；如果存在，则返回 `false`。

## 4、删除元素

```
1 public boolean remove(Object o) {  
2     return map.remove(o) == PRESENT;  
3 }
```

调用 HashMap 的 remove(Object o) 方法，该方法会首先查找 map 集合中是否存在 o，如果存在则删除，并返回该值，如果不存在则返回 null。

也就是 remove(Object o) 方法，删除成功返回 true，删除的元素不存在会返回 false。

## 5、查找元素

```
1 public boolean contains(Object o) {  
2     return map.containsKey(o);  
3 }
```

调用 HashMap 的 containsKey(Object o) 方法，找到了返回 true，找不到返回 false。

## 6、遍历元素

```
1 LinkedHashSet<String> hashSet = new LinkedHashSet<>();  
2 hashSet.add("A");  
3 hashSet.add("B");  
4 hashSet.add("C");  
5 //1、增强for循环  
6 for(String str : hashSet){  
7     System.out.println(str);  
8 }  
9 //2、迭代器  
10 Iterator<String> iterator = hashSet.iterator();  
11 while(iterator.hasNext()){  
12     System.out.println(iterator.next());  
13 }
```

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



微信搜一搜

Q IT可乐