Java 源代码分析

1. Object 类——一切类的父类

（1）Object 类的声明

```
package java.lang;     //Object 类定义在 java.lang 包下
public class Object
```

（2）hashCode 方法

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

（3）（重要）equals( )方法——比较对象的引用

Object 类的 equals( )方法比较的是对象的引用而不是内容，这与实际生活的情况不符，实际生活中人们要求比较的是内容，比较引用没有意义，所以**其他类继承 Object 类之后，往往需要重写 equals( )方法**

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

（4）（重要）toString( )方法——打印对象的信息

打印一个对象就是调用 toString( )方法，该方法打印这个对象的信息，**其他类在继承 Object 类后，一般要重写此方法**，设置打印关于对象的信息的操作。

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

（5）线程的等待

```
public final void wait() throws InterruptedException {
    wait(0);
}
```

```
public final native void wait(long timeout)
    throws InterruptedException;
```

还可以指定等待的最长毫秒和纳秒

```
public final void wait(long timeout, int nanos) throws
InterruptedException {
        if (timeout < 0) {
            throw new IllegalArgumentException("timeout value is
negative");
        }
```

```
        if (nanos < 0 || nanos > 999999) {
            throw new IllegalArgumentException(
                                "nanosecond timeout value out of range");
        }
        if (nanos >= 500000 || (nanos != 0 && timeout == 0)) {
            timeout++;
        }
        wait(timeout);
    }
```

（6）唤醒等待的线程

6.1 唤醒一个线程

```
public final native void notify();
```

6.2 唤醒全部线程

唤醒全部线程，哪个线程的优先级高，他就有可能先执行

```
public final native void notifyAll();
```

（7）finalize()方法——对象的回收

当确定一个对象不会被其他方法再使用时，该对象就没有存在的意义了，就只能等待 JVM 的垃圾回收线程来回收了。垃圾回收是以占用一定内存资源为代价的。System.gc();就是启动垃圾回收线程的语句。当用户认为需要回收时，可以使用 Runtime.getRuntime( ).gc( );或者 System.gc();来回收内存。（System.gc();调用的就是 Runtime 类的 gc( )方法）

当一个对象在回收前想要执行一些操作，就要覆写 Object 类中的 finalize( )方法。

```
protected void finalize() throws Throwable { }
```

注意到抛出的是 Throwable，说明除了常规的异常 Exceprion 外，还有可能是 JVM 错误。说明调用该方法不一定只会在程序中产生异常，还有可能产生 JVM 错误。

（8）clone( )方法——用于对象克隆

Object 类的直接或间接子类通过在重写该方法并直接调用本类的该方法完成对象克隆。

```
protected native Object clone() throws CloneNotSupportedException;
```

native 关键字表示要调用本机操作系统的函数。

2. String 类

（1）String 类声明

String 类实现了序列化、比较器两个接口，
用 final 修饰说明该类不能被其他类继承，其他类不能重写该类已有的方法

```
package java.lang;   //定义在java.lang包下
```

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
```

（2）字符串的本质是 char[ ]

```
/** The value is used for character storage. */ //该数组用于字符存储
    private final char value[];  // String 类底层是 char 数组
```

（3）String 类的无参构造方法

```
public String() {
        this.value = new char[0];  //构造一个空串
}
```

（4）使用 char[]构造一个字符串

调用 Arrays 类字符串拷贝函数复制一份字符完成构造

```
public String(char value[]) {
        this.value = Arrays.copyOf(value, value.length);
}
```

```
public String(char value[], int offset, int count) {
        //处理非法参数
        if (offset < 0) {
            throw new StringIndexOutOfBoundsException(offset);
        }
        if (count < 0) {
            throw new StringIndexOutOfBoundsException(count);
        }
        // Note: offset or count might be near -1>>>1.
        if (offset > value.length - count) {
            throw new StringIndexOutOfBoundsException(offset + count);
        }
        //参数合法，调用字符串复制函数复制目标char[]到value[]
        this.value = Arrays.copyOfRange(value, offset, offset+count);
}
```

（5）检查边界的函数

如果越界会抛出异常 StringIndexOutOfBoundsException

```
private static void checkBounds(byte[] bytes,int offset,int length){
        if (length < 0)
            throw new StringIndexOutOfBoundsException(length);
        if (offset < 0)
            throw new StringIndexOutOfBoundsException(offset);
        if (offset > bytes.length - length)
            throw new StringIndexOutOfBoundsException(offset + length);
}
```

（6）使用 bytes[ ]初始化字符串对象

```
public String(byte bytes[]) {
        this(bytes, 0, bytes.length);
}
```

```
public String(byte bytes[], int offset, int length) {
        //offset是起始位置，length是从起始位置计要构造字符串的长度
        checkBounds(bytes, offset, length);
        this.value = StringCoding.decode(bytes, offset, length);
}
```

```
public String(byte bytes[]) {
        this(bytes, 0, bytes.length);
}
```

（7）获取字符串的长度

字符串的长度也就是 value 数组的长度，即有效字符的个数。

```
public int length() {
        return value.length;
}
```

（8）判断字符串是否是空串

```
public boolean isEmpty() {
        return value.length == 0;  //检测其有效字符个数是否为0
}
```

（9）获取某一位置的字符

如果传入的索引合法，就返回该位置的字符

```
public char charAt(int index) {
        if ((index < 0) || (index >= value.length)) {
            throw new StringIndexOutOfBoundsException(index);
        }
```

```
        return value[index];
}
```

（10）将字符串转换为 byte[ ]数组

```
public byte[] getBytes() {
        return StringCoding.encode(value, 0, value.length);
}
```

```
static byte[] encode(char[] ca, int off, int len) {
        String csn = Charset.defaultCharset().name();
        try {
        // use charset name encode() variant which provides caching.
            return encode(csn, ca, off, len);
        } catch (UnsupportedEncodingException x) {
            warnUnsupportedCharset(csn);
        }
        try {  //默认转换为ISO-8859-1编码
            return encode("ISO-8859-1", ca, off, len);
        } catch (UnsupportedEncodingException x) {
/* If this code is hit during VM initialization, MessageUtils is the only
way we will be able to get any kind of error message.*/
            MessageUtils.err("ISO-8859-1 charset not available: "
                             + x.toString()); //如果转换不成功，输出信息
/* If we can not find ISO-8859-1 (a required encoding) then things are
seriously wrong with the installation.*/
            System.exit(1);
            return null;
        }
}
```

（11）（重点）字符串比较函数 equals()

不同于”= =”比较，该函数比较的是字符串的内容而非引用，前者比较的是引用（地址）

```
public boolean equals(Object anObject) {  //使用Object类接收，向上转型
        if (this == anObject) {
            return true;
        }
        if (anObject instanceof String) {  //确定是否是String类的对象，
            //向下转型一定要通过instanceof进行检测，确保操作安全
            String anotherString = (String) anObject;//强制转换为String
            int n = value.length;  //获取待比较字符串的长度
            //先比较两者长度，若长度相等再比较内容
            if (n == anotherString.value.length) {  //如果两串等长
                char v1[] = value;                  //获取本串内容
                char v2[] = anotherString.value;    //获取待比较串内容
```

```
                int i = 0;  //设置一个指示器，指示两串中同一位置
                while (n-- != 0) {  //字符串未比较完
                    if (v1[i] != v2[i])   //若相同位置的字符不同
                            return false;  //比较失败，返回false
                    i++;  //否则继续比较，直到比较不成功或者全部比较完成
                }
                return true; //若全部比较成功，返回true表示成功
            }
        }
        return false;  //如果传入的引用不是String类型的，则返回false
}
```

（12）比较两字符串内容，忽略大小写

```
public boolean equalsIgnoreCase(String anotherString) {
        //this指代当前对象，当前字符串
        return (this == anotherString) ? true: (anotherString != null)
        && (anotherString.value.length == value.length)
        && regionMatches(true, 0, anotherString, 0, value.length);
        // regionMatches()是忽略大小写具体比较方法
}
```

```
public boolean regionMatches(boolean ignoreCase, int toffset,
            String other, int ooffset, int len) {
        char ta[] = value;          //本字符串
        int to = toffset;
        char pa[] = other.value;   //待比较串
        int po = ooffset;
        // Note: toffset, ooffset, or len might be near -1>>>1.
        if ((ooffset < 0) || (toffset < 0)   //参数的值不合法
                || (toffset > (long)value.length - len)
                || (ooffset > (long)other.value.length - len)) {
            return false;  //返回false并退出
        }
        //若参数的值合法
        while (len-- > 0) {    //转换没有结束
            char c1 = ta[to++];  //取本串某一位置的一个字符
            char c2 = pa[po++];  //取待比较串同一位置的一个字符
            if (c1 == c2) {  //先看两字符不转换大小写能否比较成功
                continue;
            }
            if (ignoreCase) {  //如果传递的参数要求忽略大小写比较
/* If characters don't match but case may be ignored, try converting both
characters to uppercase. If the results match, then the comparison scan
should continue.
   如果两字符不匹配，但是这种情况可以被忽略，尝试将两字符都转换为大写形式后比
```

```
较，如果结果匹配则可以跳过后续比较，这是由于个别语言的字母大写转换后仍然比较不成功的问题决定的，绝大多数情况下两字符都转换成大写比较成功后就应该结束算法而不是像下面一样转换成小写再比较。*/
                char u1 = Character.toUpperCase(c1);  //将c1转换成大写
                char u2 = Character.toUpperCase(c2);  //将c2转换成大写
                if (u1 == u2) { //如果两字符都转换为大写后比较成功了
                    continue;  //跳过该字符的后续比较，比较下一个位置的字符
                }
/* Unfortunately, conversion to uppercase does not work properly for the Georgian alphabet, which has strange rules about case conversion.  So we need to make one last check before exiting. 不幸的是，Georgian 字母表大小写转换语法比较怪异，所以在退出前我们需要进行最后一次必要的检测*/
                if (Character.toLowerCase(u1) ==
Character.toLowerCase(u2)) {  //如果两字符都转换成小写比较相等了
                    continue;   //跳过本次比较，进行下一次比较
                }
            }
            return false;  //比较失败
        }
        return true;       //比较成功
    }
```

（13）判断字符串是否以特定串开头

```
public boolean startsWith(String prefix) {
        return startsWith(prefix, 0);  //调用下面的函数，从头开始查找
}
```

```
public boolean startsWith(String prefix, int toffset) {
        char ta[] = value;           //获取当前串
        int to = toffset;            //toffset是开始查找的位置
        char pa[] = prefix.value;    //perfix是前缀，即以…开头
        int po = 0;
        int pc = prefix.value.length;  //获取当前串长度
        // Note: toffset might be near -1>>>1.
        if ((toffset < 0) || (toffset > value.length - pc)) {
            return false;   //参数不合法，结束方法，返回false
        }
        while (--pc >= 0) {  //从头向后查找
            if (ta[to++] != pa[po++]) { //比较两串同一位置字符是或否相等
                //在比较过程中出现了不相等的情况，说明前缀不是perfix
                return false;
            }
        }
        return true;
}
```

（14）判断字符串是否以指定后缀结尾

```
public boolean endsWith(String suffix) {
        return startsWith(suffix, value.length - suffix.value.length);
}
```

其实：endsWith(String suffix)是间接调用startsWith(String prefix, int toffset)完成操作的，只不过比较的是靠近结尾的若干字符

（15）查找字符串的位置

从头开始查找特定字符串的位置，找不到返回-1

```
public int indexOf(String str) {
        return indexOf(str, 0);
}
```

```
public int indexOf(String str, int fromIndex) {
        //str - 要搜索的子字符串。   fromIndex - 开始搜索的索引位置
        return indexOf(value , 0 , value.length , str.value , 0 ,
str.value.length , fromIndex);
}
```

```
static int indexOf(char[] source, int sourceOffset, int sourceCount,
char[] target, int targetOffset, int targetCount, int fromIndex) {
        // source是总串，即当前串，sourceOffset是总串的起始查找位置，
        // sourceCount是总串的长度
        // target是目标串（子串），targetOffset是子串的起始查找位置
        // targetCount是子串的长度  fromIndex - 开始搜索的索引位置
        //先判断参数是否合法，处理不合法参数
        if (fromIndex >= sourceCount) { //搜索起始位置大于等于总串的长度
            return (targetCount == 0 ? sourceCount : -1);
            //返回-1表示查找失败
        }
        if (fromIndex < 0) { //起始查找位置小于0，参数非法
            fromIndex = 0;
        }
        if (targetCount == 0) { //子串长度为0，返回起始查找位置
            return fromIndex;
        }
        char first = target[targetOffset]; //获取子串起始查找位置的字符
        int max = sourceOffset + (sourceCount - targetCount);
        //最大查找位置是总串起始查找位置+开始查找索引位置-子串长
        for (int i = sourceOffset + fromIndex; i <= max; i++) {
        //设置i的初值为起始搜素索引位置，搜索终止条件是到达最大查找位置
            if (source[i] != first) {
                //总串的搜索起始字符不是子串的第一个字符，
                //则向后查找该字符，到达待比较位置
                while (++i <= max && source[i] != first); //向后查找
```

```
            }
            //总串中不一定能查找到子串的第一个字符，因此要进行判断
            if (i <= max) { //如果i没有超过最大查找位置，即找到了first
                int j = i + 1;
                //j指示总串中查找到子串的第一个字符的位置后面的字符
                int end = j + targetCount - 1;  //查找的终止位置
                for (int k = targetOffset + 1; j < end && source[j]
                        == target[k]; j++, k++);
                //设置一个指示器K，并从j位置开始向后比较，
                //直到出现比较失败（失败）或到达终止位置（成功）
                if (j == end) { //j和终止位置相等，说明比较成功了
                    return i - sourceOffset;
                }
            }
        }
        return -1;
}
```

（16）截取子串

```
public String substring(int beginIndex) {
        //处理非法参数
        if (beginIndex < 0) {
            throw new StringIndexOutOfBoundsException(beginIndex);
        }
        int subLen = value.length - beginIndex;
        if (subLen < 0) {
            throw new StringIndexOutOfBoundsException(subLen);
        }
        //参数合法后根据开始索引返回对应的操作结果
        return (beginIndex == 0)?this : new String(value, beginIndex ,
subLen);
}
```

```
public String substring(int beginIndex, int endIndex) {
        //处理非法参数
        if (beginIndex < 0) {
            throw new StringIndexOutOfBoundsException(beginIndex);
        }
        if (endIndex > value.length) {
            throw new StringIndexOutOfBoundsException(endIndex);
        }
        int subLen = endIndex - beginIndex;
        if (subLen < 0) {
            throw new StringIndexOutOfBoundsException(subLen);
        }
```

```
        return ((beginIndex == 0) && (endIndex == value.length)) ? this:
new String(value, beginIndex, subLen);  //返回要求的字符串
}
```

取子串就是用主串的一部分构造一个新串

（17）串连接

```
public String concat(String str) {
        int otherLen = str.length();
        if (otherLen == 0) {
            return this;
        }
        int len = value.length;
        char buf[] = Arrays.copyOf(value, len + otherLen);
        str.getChars(buf, len);
        return new String(buf, true);
}
```

```
String(char[] value, boolean share) {
        this.value = value;
}
```

```
void getChars(char dst[], int dstBegin) {
        System.arraycopy(value, 0, dst, dstBegin, value.length);
}
```

（18）替换全部

替换主串中的全部的指定串

```
public String replaceAll(String regex, String replacement) {
        return Pattern.compile(regex).matcher(this).replaceAll
(replacement);
}
```

（19）字符串的拆分

```
public String[] split(String regex) {
        return split(regex, 0);
}
```

```
public String[] split(String regex, int limit) {
/* fastpath if the regex is a
   (1)one-char String and this character is not one of the RegEx's meta
characters ".$|()[{^?*+\\", or
   (2)two-char String and the first char is the backslash and the second
is not the ascii digit or ascii letter.
如果该字符串是如下情况可以走捷径：
   （1）单字符字符串并且第一个字符不是以下字符串中的任何一个字符：
".$|()[{^?*+\\" 或者，
```

```
    （2）双字符字符串，第一个字符不是反斜杠且第二个字符不是ASCII数字或者ASCII
字母
*/
        char ch = 0;
        if (((regex.value.length == 1 &&
              ".$|()[{^?*+\\".indexOf(ch = regex.charAt(0)) == -1)
              || (regex.length() == 2
              && regex.charAt(0) == '\\'
              && (((ch = regex.charAt(1))-'0')|('9'-ch)) < 0
              && ((ch-'a')| ('z'-ch)) < 0
              && ((ch-'A')|('Z'-ch)) < 0))
              &&(ch < Character .MIN_HIGH_SURROGATE ||ch > Character.
MAX_LOW_SURROGATE)){
            int off = 0;
            int next = 0;
            boolean limited = limit > 0;
            ArrayList<String> list = new ArrayList<>();
            while ((next = indexOf(ch, off)) != -1) {
                if (!limited || list.size() < limit - 1) {
                    list.add(substring(off, next));
                    off = next + 1;
                } else {    // last one
                    list.add(substring(off, value.length));
                    off = value.length;
                    break;
                }
            }
            // If no match was found, return this
            if (off == 0)
                return new String[]{this};

            // Add remaining segment
            if (!limited || list.size() < limit)
                list.add(substring(off, value.length));

            // Construct result
            int resultSize = list.size();
            if (limit == 0)
                while (resultSize > 0 && list.get(resultSize - 1).length()
== 0)
                    resultSize--;
            String[] result = new String[resultSize];
            return list.subList(0, resultSize).toArray(result);
        }
```

```
        return Pattern.compile(regex).split(this, limit);
}
```

（20）将一个字符串全部变为小写字母

```
public String toLowerCase() {
        return toLowerCase(Locale.getDefault());
}
```

```
public String toLowerCase(Locale locale) {
        if (locale == null) {
            throw new NullPointerException();
        }
        int firstUpper;
        final int len = value.length;
        /* Now check if there are any characters that need to be changed.
*/
        scan: {
            for (firstUpper = 0 ; firstUpper < len; ) {
                char c = value[firstUpper];
                if ((c >= Character.MIN_HIGH_SURROGATE)
                        && (c <= Character.MAX_HIGH_SURROGATE)) {
                    int supplChar = codePointAt(firstUpper);
                    if (supplChar != Character.toLowerCase(supplChar)) {
                        break scan;
                    }
                    firstUpper += Character.charCount(supplChar);
                } else {
                    if (c != Character.toLowerCase(c)) {
                        break scan;
                    }
                    firstUpper++;
                }
            }
            return this;
        }

        char[] result = new char[len];
        int resultOffset = 0;  /* result may grow, so i+resultOffset
                                * is the write location in result */

        /* Just copy the first few lowerCase characters. */
        System.arraycopy(value, 0, result, 0, firstUpper);

        String lang = locale.getLanguage();
        boolean localeDependent =
```

```
                (lang == "tr" || lang == "az" || lang == "lt");
        char[] lowerCharArray;
        int lowerChar;
        int srcChar;
        int srcCount;
        for (int i = firstUpper; i < len; i += srcCount) {
            srcChar = (int)value[i];
            if ((char)srcChar >= Character.MIN_HIGH_SURROGATE
                    && (char)srcChar <= Character.MAX_HIGH_SURROGATE) {
                srcChar = codePointAt(i);
                srcCount = Character.charCount(srcChar);
            } else {
                srcCount = 1;
            }
            if (localeDependent || srcChar == '\u03A3') { // GREEK CAPITAL
LETTER SIGMA
                lowerChar = ConditionalSpecialCasing.toLowerCaseEx(this,
i, locale);
            } else if (srcChar == '\u0130') { // LATIN CAPITAL LETTER I DOT
                lowerChar = Character.ERROR;
            } else {
                lowerChar = Character.toLowerCase(srcChar);
            }
            if ((lowerChar == Character.ERROR)
                    || (lowerChar >=
Character.MIN_SUPPLEMENTARY_CODE_POINT)) {
                if (lowerChar == Character.ERROR) {
                    if (!localeDependent && srcChar == '\u0130') {
                        lowerCharArray =

ConditionalSpecialCasing.toLowerCaseCharArray(this, i, Locale.ENGLISH);
                    } else {
                        lowerCharArray =

ConditionalSpecialCasing.toLowerCaseCharArray(this, i, locale);
                    }
                } else if (srcCount == 2) {
                    resultOffset += Character.toChars(lowerChar, result,
i + resultOffset) - srcCount;
                    continue;
                } else {
                    lowerCharArray = Character.toChars(lowerChar);
                }
```

```
                /* Grow result if needed */
                int mapLen = lowerCharArray.length;
                if (mapLen > srcCount) {
                    char[] result2 = new char[result.length + mapLen -
srcCount];
                    System.arraycopy(result, 0, result2, 0, i +
resultOffset);
                    result = result2;
                }
                for (int x = 0; x < mapLen; ++x) {
                    result[i + resultOffset + x] = lowerCharArray[x];
                }
                resultOffset += (mapLen - srcCount);
            } else {
                result[i + resultOffset] = (char)lowerChar;
            }
        }
        return new String(result, 0, len + resultOffset);
    }
```

（21）将一个字符串全部变为大写字母

```
public String toUpperCase() {
        return toUpperCase(Locale.getDefault());
}
```

```
public String toUpperCase(Locale locale) {
        if (locale == null) {
            throw new NullPointerException();
        }

        int firstLower;
        final int len = value.length;

        /* Now check if there are any characters that need to be changed.
*/
        scan: {
           for (firstLower = 0 ; firstLower < len; ) {
                int c = (int)value[firstLower];
                int srcCount;
                if ((c >= Character.MIN_HIGH_SURROGATE)
                        && (c <= Character.MAX_HIGH_SURROGATE)) {
                    c = codePointAt(firstLower);
                    srcCount = Character.charCount(c);
                } else {
                    srcCount = 1;
```

```
                }
                int upperCaseChar = Character.toUpperCaseEx(c);
                if ((upperCaseChar == Character.ERROR)
                        || (c != upperCaseChar)) {
                    break scan;
                }
                firstLower += srcCount;
            }
            return this;
        }

        char[] result = new char[len]; /* may grow */
        int resultOffset = 0;  /* result may grow, so i+resultOffset
         * is the write location in result */

        /* Just copy the first few upperCase characters. */
        System.arraycopy(value, 0, result, 0, firstLower);

        String lang = locale.getLanguage();
        boolean localeDependent =
                (lang == "tr" || lang == "az" || lang == "lt");
        char[] upperCharArray;
        int upperChar;
        int srcChar;
        int srcCount;
        for (int i = firstLower; i < len; i += srcCount) {
            srcChar = (int)value[i];
            if ((char)srcChar >= Character.MIN_HIGH_SURROGATE &&
                (char)srcChar <= Character.MAX_HIGH_SURROGATE) {
                srcChar = codePointAt(i);
                srcCount = Character.charCount(srcChar);
            } else {
                srcCount = 1;
            }
            if (localeDependent) {
                upperChar = ConditionalSpecialCasing.toUpperCaseEx(this,
i, locale);
            } else {
                upperChar = Character.toUpperCaseEx(srcChar);
            }
            if ((upperChar == Character.ERROR)
                    || (upperChar >=
Character.MIN_SUPPLEMENTARY_CODE_POINT)) {
                if (upperChar == Character.ERROR) {
```

```
                    if (localeDependent) {
                        upperCharArray =

ConditionalSpecialCasing.toUpperCaseCharArray(this, i, locale);
                    } else {
                        upperCharArray =
Character.toUpperCaseCharArray(srcChar);
                    }
                } else if (srcCount == 2) {
                    resultOffset += Character.toChars(upperChar, result,
i + resultOffset) - srcCount;
                    continue;
                } else {
                    upperCharArray = Character.toChars(upperChar);
                }

                /* Grow result if needed */
                int mapLen = upperCharArray.length;
                if (mapLen > srcCount) {
                    char[] result2 = new char[result.length + mapLen -
srcCount];
                    System.arraycopy(result, 0, result2, 0, i +
resultOffset);
                    result = result2;
                }
                for (int x = 0; x < mapLen; ++x) {
                    result[i + resultOffset + x] = upperCharArray[x];
                }
                resultOffset += (mapLen - srcCount);
            } else {
                result[i + resultOffset] = (char)upperChar;
            }
        }
        return new String(result, 0, len + resultOffset);
    }
```

（22）去掉左右两端的空格

```
public String trim() {
        int len = value.length;
        int st = 0;
        char[] val = value;    /* avoid getfield opcode */
        while ((st < len) && (val[st] <= ' ')) {
            st++;
        }
```

```
        while ((st < len) && (val[len - 1] <= ' ')) {
            len--;
        }
        return ((st > 0) || (len < value.length)) ? substring(st, len) :
this;
}
```

记录左端第一个不是空格字符的索引，记下从末尾逆向计算第一个不是空格的字符的索引，使用这两个索引取子串即可。

（23）toString()方法

一个对象调用 toString()方法相当于打印该对象的信息。

```
public String toString() {
        return this;
}
```

（24）将一个字符串转换为 char[ ]

```
public char[] toCharArray() {
        // Cannot use Arrays.copyOf because of class initialization order
issues
        char result[] = new char[value.length];
        System.arraycopy(value, 0, result, 0, value.length);
        return result;
}
```

3. StringBuffer 类

（1）StringBuffer 类声明

该类被声明成 final 类型，说明该类无法被其他类继承，实现了 Serializable 接口说明该类是可序列化的。该类继承了 AbstractStringBuilder 这一抽象类。

```
package java.lang;  //定义在java.lang包下
```

```
public final class StringBuffer extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
```

```
abstract class AbstractStringBuilder implements Appendable, CharSequence
```

（2）无参构造函数

StringBuffer 类的无参构造函数调用父类的构造函数

```
public StringBuffer() {
        super(16);   //调用父类的构造方法，传递参数16
}
```

```
char[] value;  //StringBuffer底层是char数组，该数组是父类中的成员
int count;     //count是实际存储字符的个数或者说可用字符的个数，继承自父类
```

```
AbstractStringBuilder(int capacity) {
    value = new char[capacity];  //生成指定空间capacity的char数组
}
```

（3）获得字符串的长度

```
public synchronized int length() { //synchronized是线程同步的关键字
        return count;  //count是其父类中的成员变量,有效字符个数
}
```

（4）获得存储容量

```
public synchronized int capacity() {
        return value.length;   //容量指char数组能够存储字符的个数
}
```

注意容量和字符串长度的区别，字符串长度是指有效字符的个数，而容量指char数组能够存储字符的个数，两者区别在于：存储在char数组中的字符不一定都是有效字符，开辟存储空间和扩容总是以一定单位进行的，有些存储空间可能不会被用到，所以字符串长度一定小于或等于容量。

（5）（重点）字符串追加

```
public synchronized StringBuffer append(StringBuffer sb) {
        super.append(sb);  //调用父类的同名方法
        return this;
}
```

```
public AbstractStringBuilder append(StringBuffer sb) {
```

```
        if (sb == null)    //首先判断是否为空
            return append("null");  //追加字符串"null"
        int len = sb.length();  //若不为空，获取其长度
        ensureCapacityInternal(count + len);  //确认是否需要扩容
        sb.getChars(0, len, value, count);  //复制追加内容到字符串末尾
        count += len;
        return this;
}
```

```
private void ensureCapacityInternal(int minimumCapacity) {
        if (minimumCapacity - value.length > 0)
            expandCapacity(minimumCapacity);  //扩容
}
```

```
public synchronized void getChars(int srcBegin, int srcEnd, char[] dst,
int dstBegin){
/* 将字符从此序列复制到目标字符数组 dst。要复制的第一个字符在索引 srcBegin
处；要复制的最后一个字符在索引 srcEnd-1 处。要复制的字符总数为
srcEnd-srcBegin。要复制到 dst 子数组的字符从索引 dstBegin 处开始，结束于以
下索引：
 dstbegin + (srcEnd-srcBegin) - 1
    srcBegin - 从此偏移量处开始复制。   srcEnd - 在此偏移量处停止复制。
    dst - 用来保存复制数据的数组。       dstBegin - dst 中的偏移量。   */
        super.getChars(srcBegin, srcEnd, dst, dstBegin); //调用父类方法
}
```

```
public void getChars(int srcBegin, int srcEnd, char[] dst, int
dstBegin){   //该方法是AbstractStringBuilder类中的同名方法
/* 将字符从此序列复制到目标字符数组 dst。要复制的第一个字符在索引 srcBegin
处；要复制的最后一个字符在索引 srcEnd-1 处。要复制的字符总数为
srcEnd-srcBegin。要复制到 dst 子数组的字符从索引 dstBegin 处开始，结束于以
下索引：
 dstbegin + (srcEnd-srcBegin) - 1
    srcBegin - 从此偏移量处开始复制。   srcEnd - 在此偏移量处停止复制。
    dst - 用来保存复制数据的数组。       dstBegin - dst 中的偏移量。   */
        if (srcBegin < 0)
            throw new StringIndexOutOfBoundsException(srcBegin);
        if ((srcEnd < 0) || (srcEnd > count))
            throw new StringIndexOutOfBoundsException(srcEnd);
        if (srcBegin > srcEnd)
            throw new StringIndexOutOfBoundsException("srcBegin >
srcEnd");
        System.arraycopy(value, srcBegin, dst, dstBegin, srcEnd -
srcBegin);
}
```

```
public synchronized StringBuffer append(String str) {
```

```
    super.append(str);
    return this;
}
```

```
public AbstractStringBuilder append(String str) {
        if (str == null) str = "null";  //判断是否为空
        int len = str.length();          //获取追加串长度
        ensureCapacityInternal(count + len);  //确定是否需要扩容
        str.getChars(0, len, value, count);   //复制追加串到当前串尾部
        count += len;   //当前串长度相应增加
        return this;   //返回操作结果
}
```

```
public synchronized StringBuffer append(char c) {
        super.append(c);  //调用父类的方法
        return this;      //返回操作结果
}
```

```
public AbstractStringBuilder append(char c) {
        ensureCapacityInternal(count + 1);   //确认是否需要扩容
        value[count++] = c;                   //将c增加到当前串末尾
        return this;   //返回操作结果
}
```

（6）指定内容替换

```
public synchronized StringBuffer replace(int start, int end, String str){
        // start是开始位置，end是结束位置，str是替换串
        super.replace(start, end, str);  //调用父类的方法
        return this;
}
```

```
public AbstractStringBuilder replace(int start, int end, String str){
        //如果存在不合法参数，先处理
        if (start < 0)
            throw new StringIndexOutOfBoundsException(start);
        if (start > count)
            throw new StringIndexOutOfBoundsException("start >
length()");
        if (start > end)
            throw new StringIndexOutOfBoundsException("start > end");
        if (end > count)
            end = count;
        //各参数合法，进行处理
        int len = str.length();  //获取替换串串长
        //替换操作有可能换成更长的串，如果存储空间不够可能会扩容，
        //所以要先计算替换后的新串长
        int newCount = count + len - (end - start);
```

```
        ensureCapacityInternal(newCount);   //确认是否需要扩容
        //将替换串拷贝到被替换位置
        System.arraycopy(value, end, value, start + len, count - end);
        str.getChars(value, start);
        count = newCount;  //串长增加
        return this;
}
```

（7）删除指定范围的字符串

```
public synchronized StringBuffer delete(int start, int end) {
        super.delete(start, end);  //调用父类的方法完成操作
        return this;
}
```

```
public AbstractStringBuilder delete(int start, int end) {
        //先处理不合法参数
        if (start < 0)
            throw new StringIndexOutOfBoundsException(start);
        if (end > count)
            end = count;
        if (start > end)
            throw new StringIndexOutOfBoundsException();
        int len = end - start;   //求要删除字符串的长度
        if (len > 0) {  //长度合法
            System.arraycopy(value, start+len, value, start, count-end);
//将end后面的内容拷贝到start位置，覆盖掉待删除内容
            count -= len;  //串长相应减少
        }
        return this;  //返回操作结果
}
```

（8）截取子串

```
public synchronized String substring(int start) {
        return substring(start, count);
}
```

```
public synchronized String substring(int start, int end) {
        return super.substring(start, end);  //调用父类的方法
}
```

```
public String substring(int start, int end) {
        //先处理不合法参数
        if (start < 0)
            throw new StringIndexOutOfBoundsException(start);
```

```
        if (end > count)
            throw new StringIndexOutOfBoundsException(end);
        if (start > end)
            throw new StringIndexOutOfBoundsException(end - start);
        return new String(value, start, end - start); //生成新串
}
```

截取子串就是将当前串的某一部分或全部构造一个新串

（9）在指定位置插入串

```
public synchronized StringBuffer insert(int offset, String str) {
        super.insert(offset, str);  //调用父类的方法
        return this;
}
```

```
public AbstractStringBuilder insert(int offset, String str) {
        //先处理非法参数
        if ((offset < 0) || (offset > length()))
            throw new StringIndexOutOfBoundsException(offset);
        if (str == null)
            str = "null";
        int len = str.length();  //获取待插入串长度
        ensureCapacityInternal(count + len);  //确认是否需要扩容
        System.arraycopy(value, offset, value, offset + len, count -
offset);   //将插入位置之后的字符复制到后面，腾出插入字符串的空间
        str.getChars(value, offset);  //将插入串复制到指定位置上
        count += len;  //串长相应增加
        return this;   //返回操作结果
}
```

（10）查找指定字符串第一次出现的位置

```
public int indexOf(String str) {
        return indexOf(str, 0);
}
```

```
public synchronized int indexOf(String str, int fromIndex) {
        return String.indexOf(value, 0, count,str.toCharArray(), 0,
str.length(), fromIndex);  //调用String类的静态方法
}
```

```
static int indexOf(char[] source, int sourceOffset, int sourceCount,
char[] target, int targetOffset, int targetCount, int fromIndex) {
        // source是总串，即当前串，sourceOffset是总串的起始查找位置，
        // sourceCount是总串的长度
        // target是目标串（子串），targetOffset是子串的起始查找位置
        // targetCount是子串的长度  fromIndex - 开始搜索的索引位置
        //先判断参数是否合法，处理不合法参数
```

```
        if (fromIndex >= sourceCount) { //搜索起始位置大于等于总串的长度
            return (targetCount == 0 ? sourceCount : -1);
            //返回-1表示查找失败
        }
        if (fromIndex < 0) { //起始查找位置小于0，参数非法
            fromIndex = 0;
        }
        if (targetCount == 0) { //子串长度为0，返回起始查找位置
            return fromIndex;
        }
        char first = target[targetOffset]; //获取子串起始查找位置的字符
        int max = sourceOffset + (sourceCount - targetCount);
        //最大查找位置是总串起始查找位置+开始查找索引位置-子串长
        for (int i = sourceOffset + fromIndex; i <= max; i++) {
        //设置i的初值为起始搜素索引位置，搜索终止条件是到达最大查找位置
            if (source[i] != first) {
                //总串的搜索起始字符不是子串的第一个字符，
                //则向后查找该字符，到达待比较位置
                while (++i <= max && source[i] != first); //向后查找
            }
            //总串中不一定能查找到子串的第一个字符，因此要进行判断
            if (i <= max) { //如果i没有超过最大查找位置，即找到了first
                int j = i + 1;
                //j指示总串中查找到子串的第一个字符的位置后面的字符
                int end = j + targetCount - 1;  //查找的终止位置
                for (int k = targetOffset + 1; j < end && source[j]
                        == target[k]; j++, k++);
                //设置一个指示器K，并从j位置开始向后比较，
                //直到出现比较失败（失败）或到达终止位置（成功）
                if (j == end) { //j和终止位置相等，说明比较成功了
                    return i - sourceOffset;
                }
            }
        }
        return -1;
}
```

（11）字符串逆置（翻转）

```
public synchronized StringBuffer reverse() {
        super.reverse();  //调用父类中的方法
        return this;
}
```

```
public AbstractStringBuilder reverse() {
        boolean hasSurrogate = false;
```

```
        int n = count - 1;
        for (int j = (n-1) >> 1; j >= 0; --j) {
            char temp = value[j];
            char temp2 = value[n - j];
            if (!hasSurrogate) {
                hasSurrogate = (temp >= Character.MIN_SURROGATE && temp <=
Character.MAX_SURROGATE)
                    || (temp2 >= Character.MIN_SURROGATE && temp2 <=
Character.MAX_SURROGATE);
            }
            value[j] = temp2;
            value[n - j] = temp;
        }
        if (hasSurrogate) {
            // Reverse back all valid surrogate pairs
            for (int i = 0; i < count - 1; i++) {
                char c2 = value[i];
                if (Character.isLowSurrogate(c2)) {
                    char c1 = value[i + 1];
                    if (Character.isHighSurrogate(c1)) {
                        value[i++] = c1;
                        value[i] = c2;
                    }
                }
            }
        }
        return this;
}
```

关于字符串翻转算法中hasSurrogate的一点说明：

从方法的内容来看，源代码中的基本思路是采用遍历一半字符串，然后将每个字符与其对应的字符进行交换。但是和字符串传统翻转算法的不同之处就是要判断每个字符是否在Character.MIN_SURROGATE(\ud800)和Character.MAX_SURROGATE(\udfff)之间。如果发现整个字符串中含有这种情况，则再次从头至尾遍历一次，同时判断value[i]是否满足Character.isLowSurrogate()，如果满足的情况下，继续判断value[i+1]是否满足Character.isHighSurrogate()，如果也满足这种情况，则将第i位和第i+1位的字符互换。可能有的人会疑惑，为什么要这么做，因为Java中的字符已经采用Unicode代码，每个字符可以放下一个汉字。为什么还要这么做？

一个完整的Unicode字符叫代码点CodePoint，而一个Java char叫代码单元code unit。**String 对象以UTF-16保存 Unicode 字符，需要用2个字符表示一个超大字符集的汉字，这这种表示方式称之为 Surrogate**，第一个字符叫 Surrogate High，第二个就是 Surrogate Low。具体需要注意的事宜如下：

判断一个char是否是Surrogate区的字符，用Character的 isHighSurrogate()/ isLowSurrogate()方法即可判断。从两个Surrogate High/Low 字符，返回一个完整的 Unicode CodePoint 用 Character.toCodePoint()/codePointAt()方法。

一个Code Point，可能需要一个也可能需要两个char表示，因此不能直接使用CharSequence.length()方法直接返回一个字符串到底有多少个汉字，而需要用String.codePointCount() / Character.codePointCount()。要定位字符串中的第N个字符，不能直接将N作为偏移量，而需要从字符串头部依次遍历得到，需要用String/ Character. offsetByCodePoints() 方法。

从字符串的当前字符，找到上一个字符，也不能直接用offset-- 实现，而需要用String.codePointBefore()/Character.codePointBefore()，或用 String/ Character. offsetBy Code Points()

从当前字符，找下一个字符，不能直接用 offset++实现，需要判断当前 CodePoint 的长度后，再计算得到，或用String/Character.offsetByCodePoints()。

（12）toString( )方法

打印一个对象相当于调用他的 toString( )方法

```
public synchronized String toString() {
        return new String(value, 0, count);//使用底层的value数组产生新串
}
```

```
public String(char value[], int offset, int count) {
        if (offset < 0) {
            throw new StringIndexOutOfBoundsException(offset);
        }
        if (count < 0) {
            throw new StringIndexOutOfBoundsException(count);
        }
        if (offset > value.length - count) {
            throw new StringIndexOutOfBoundsException(offset + count);
        }
        this.value = Arrays.copyOfRange(value, offset, offset+count);
}
```

小结：StringBuffer 类处处可见 synchronized 关键字，StringBuffer 是线程安全的。

StringBuffer 类善于处理含有大量字符串拼接的操作，占用内存少，性能较好，比 String 类性能好很多。

4. Thread 类

（1）Thread 类和 Runnable 接口的声明

Thread 类实现了 Runnable 接口，可以看做 Runnable 接口的子类

```
package java.lang;  //定义在java.lang包下
public class Thread extends Object implements Runnable
```

Runnable 接口的完整声明，可以看出该接口只有一个抽象方法 run( )方法。

实现该接口的类必须重写 run( )方法，否则会变成抽象类。

```
public interface Runnable {
    public abstract void run();
}
```

（2）Thread 类的构造方法

2.1 接收 Runnable 接口子类对象，实例化 Thread 对象

```
public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum(), 0);
}
```

```
private void init(ThreadGroup g, Runnable target, String name, long
stackSize) {
        init(g, target, name, stackSize, null);
}
```

```
public Thread(ThreadGroup group, String name) {
        init(group, null, name, 0);
}
```

```
public Thread(String name) {
        init(null, null, name, 0);
}
```

（3）Thread 类中的 run( )方法

```
public void run() {
        if (target != null) {
            target.run();
        }
}
```

```
private Runnable target;  //target是一个实现了Runnable接口的类创建的对象
//target调用的是被覆写（重写）的run()方法
```

（4）start( )方法

start( )方法调用 start0( )方法，start0( )方法会调用本机操作系统函数协调 CPU 资源

```
private volatile int threadStatus = 0;
//threadStatus表示线程的一种状态：0表示还未启动的新线程
```

```
public synchronized void start() {
        if (threadStatus != 0)  //如果线程启动过了就不能再启动了
            throw new IllegalThreadStateException();
        group.add(this);  //若线程未启动过，将其加入线程组
        boolean started = false;  //表示线程是否成功启动
        try {
            start0();        //真正调用的是start0()方法
            started = true; //执行到这里没有抛出异常说明线程启动成功
        } finally {
            try {
                if (!started) {  //线程没有成功启动
                    group.threadStartFailed(this);  //执行此方法
                }
            } catch (Throwable ignore) {}
        }
}
```

```
//该方法位于ThreadGroup类中
void threadStartFailed(Thread t) {  //从线程组中移除未成功启动的线程
        synchronized(this) { //声明同步块
            remove(t);
            nUnstartedThreads++;
        }
}
```

```
private native void start0();//用native声明说明该方法会调用本机操作系统函数
```

（5）线程优先级相关的常量

```
    /*  The minimum priority that a thread can have.          */
    public final static int MIN_PRIORITY = 1;    //最低权限
   /*   The default priority that is assigned to a thread.   */
    public final static int NORM_PRIORITY = 5;   //默认权限
   /*   The maximum priority that a thread can have.          */
    public final static int MAX_PRIORITY = 10;   //最高权限
```

（6）获得当前线程

```
public static native Thread currentThread();
```

（7）线程礼让

```
public static native void yield();
```

（8）线程的休眠

线程休眠的过程中可能会出现异常，需要 try-catch 捕获

```
public static native void sleep(long millis)
      throws InterruptedException;
```

（9）判断线程是否被中断

```
public boolean isInterrupted() {
        return isInterrupted(false); //线程被中断返回true，否则返回false
}
```

```
private native boolean isInterrupted(boolean ClearInterrupted);
```

（10）判断线程是否在活动

如果是返回 true，否则返回 false

```
public final native boolean isAlive();
```

（11）取得线程优先级

```
private int priority;  //Thread类中的属性，代表优先级
```

```
public final int getPriority() {
        return priority;
}
```

（12）设置线程的优先级

```
public final void setPriority(int newPriority) {
        ThreadGroup g;
        checkAccess();
        //检查设置的优先级是否在1-10之间
        if (newPriority>MAX_PRIORITY || newPriority<MIN_PRIORITY) {
            throw new IllegalArgumentException();
        }
        //设置线程优先级
        if((g = getThreadGroup()) != null) {
            if (newPriority > g.getMaxPriority()) {
                newPriority = g.getMaxPriority();
            }
            setPriority0(priority = newPriority);
        }
}
```

```
private native void setPriority0(int newPriority);
```

```
//该方法位于ThreadGroup类中
public final int getMaxPriority() {
        return maxPriority;
}
```

（13）获得线程名称

```
private char name[];  //该数组存放线程名称，是 Thread 类的属性
```

```
public final String getName() {
        return String.valueOf(name);
}
```

（14）设置线程名称

```
public final void setName(String name) {
        checkAccess();      //检查通过
        this.name = name.toCharArray();   //转换为char[]并设置为线程名称
}
```

（15）线程的强制运行

```
public final void join() throws InterruptedException {
        join(0);  //要求线程立刻运行
}
```

```
public final synchronized void join(long millis) throws
InterruptedException {
        long base = System.currentTimeMillis();  //得到当前时间（毫秒）
        long now = 0;
        //处理非法参数
        if (millis < 0) {
            throw new IllegalArgumentException("timeout value is
negative");
        }
        if (millis == 0) {  //如果线程要求马上强制运行
            while (isAlive()) {  //线程处于活动状态
                wait(0);  //停止等待，立即执行
            }
        } else {  //如果线程要求一段时间后强制运行
            while (isAlive()) {  //线程处于活动状态
                long delay = millis - now;  //计算延迟时间
                if (delay <= 0) {
                    break;
                }
                wait(delay);  //线程等待，直到delay变为0，线程执行
                now = System.currentTimeMillis() - base; //计算当前时间
            }
        }
}
```

```
public final native void wait(long timeout)
     throws InterruptedException;
```

（16）toString( )方法

```
public String toString() {      //打印线程对象的相关信息
        ThreadGroup group = getThreadGroup();
        if (group != null) {
            return "Thread[" + getName() + "," + getPriority() + "," +
                           group.getName() + "]";
        } else {
            return "Thread[" + getName() + "," + getPriority() + "," +
                            "" + "]";
        }
}
```

（17）设置线程到后台执行

```
private boolean  daemon = false;  //Thread类成员变量，标志线程是否后台
public final void setDaemon(boolean on) {
        checkAccess();
        if (isAlive()) {  //线程处于活动状态
            throw new IllegalThreadStateException();
        }
        daemon = on;  //设置为后台线程
}
```

（18）（废弃）暂时挂起线程

```
@Deprecated
public final void suspend() {
    checkAccess();
    suspend0();
}
private native void suspend0();
```

（19）（废弃）恢复挂起的线程

```
@Deprecated
public final void resume() {
    checkAccess();
    resume0();
}
private native void resume0();
```

（20）（废弃）停止线程

```
@Deprecated
    public final void stop() {
        stop(new ThreadDeath());
}
```

```
public class ThreadDeath extends Error {
    private static final long serialVersionUID=-4417128565033088268L;
}
```

5. Number 类

（1）Number 类的声明

Number 类是一个抽象类，作为模版产生其他数字类型。

```
package java.lang;   //定义在 java.lang 包下
public abstract class Number implements java.io.Serializable
```

（2）获取 int 值

抽象方法，等待被子类重写实现具体操作。

```
public abstract int intValue();
```

（3）获取 long 值

抽象方法，等待被子类重写实现具体操作。

```
public abstract long longValue();
```

（4）获取 float 值

抽象方法，等待被子类重写实现具体操作。

```
public abstract float floatValue();
```

（5）获取 double 值

抽象方法，等待被子类重写实现具体操作。

```
public abstract double doubleValue();
```

（6）获取 byte 值

```
public byte byteValue() {
    return (byte)intValue();   //通过转换int值为byte值实现
}
```

（7）获取 short 值

```
public short shortValue() {
    return (short)intValue(); //通过转换int值为short值实现
}
```

说明：

1. 抽象类 Number 是以下类的父类：
   BigDecimal、BigInteger、Byte、Double、Float、Integer、Long、Short。
2. Number 类的子类必须覆写 Number 类中的抽象方法并提供代表数字的以下值：
   byte、double、float、int、long、short。

6. Byte 类

（1）Byte 类的声明

可以看出，Byte 是 Number 的子类，并且实现了比较器接口。

由于被 final 修饰，Byte 无法被其他类继承。

```
package java.lang;
public final class Byte extends Number implements Comparable<Byte>
```

（2）Byte 类中存储 byte 值的成员——value

此成员存储了对象拆箱后的 byte 值

```
private final byte value;
```

（3）Byte 的最大值和最小值

Byte 的最大值和最小值是作为常量出现在 Byte 类中的。

通过计算机组成原理关于定点有符号整数的表示范围的相关知识可以知道：

字长为 8 位的定点有符号整数用补码表示，范围是-2^7~（2^7-1）

```
public static final int SIZE = 8;                    //8位二进制数
public static final byte   MIN_VALUE = -128;
public static final byte   MAX_VALUE = 127;
```

（4）toString( )方法

一个对象调用 toString( )方法相当于打印该对象的信息。

```
public String toString() {
    return Integer.toString((int)value);
}
```

该方法调用Integer类的toString方法，该方法将会在Integer类中作介绍。
详细内容请参考第 37 页

（5）Byte 中的内部类 ByteCache

这个类声明为 private，仅提供给外部类使用，不提供给用户使用。

这个类中含有一个常量数组，该数组初始化时调用 Byte 的构造方法，存储了从-128 到 127 的所有整数，也就是 Byte 所能表示的全部整数。

数组的内容经初始化后无法更改。

```
private static class ByteCache {
    private ByteCache(){}
    static final Byte cache[] = new Byte[-(-128) + 127 + 1];
    static {
        for(int i = 0; i < cache.length; i++)
            cache[i] = new Byte((byte)(i - 128));
    }
}
```

（6）Byte 类拆箱

Byte 类是基本数据类型 byte 的包装类，将包装类变为基本数据类型的操作称为拆箱。在 JDK1.5 以前，装箱和拆箱是需要手动完成的，并且在集合中添加的元素不能是基本数据类型的，这会导致大量的拆装箱操作。JDK1.5 之后，提供了自动装箱和自动拆箱。

```
public static Byte valueOf(byte b) {  //拆箱的方法
        final int offset = 128;
        return ByteCache.cache[(int)b + offset];
}
```

（7）Byte 类装箱

调用 Byte 的构造方法完成装箱

```
public Byte(byte value) {
        this.value = value;
}
```

（8）将数字字符串转换为 Byte 类型

此方法很常用，但是要考虑到传入参数有可能出现的错误，当传入的字符串

①含有非数字字符

②字符串引用为 null

③含有数字的字符串表示的数字超出了 byte 表示的范围

会抛出 NumberFormatException 异常。

```
public static byte parseByte(String s) throws NumberFormatException {
    return parseByte(s, 10);
}
```

```
public static byte parseByte(String s, int radix)
    throws NumberFormatException{
    int i = Integer.parseInt(s, radix);  //调用Integer类的parseInt方法
    if (i < MIN_VALUE || i > MAX_VALUE)  //传入的数字超出了byte表示范围
        throw new NumberFormatException( //抛出异常，提示超范围
          "Value 串out of range. Value:\"" + s + "\" Radix:" + radix);
    return (byte)i;  //将操作结果强制类型转换成byte
}
```

（9）Byte 类的两个构造方法

分别用 byte 数值和字符串初始化 Byte 对象。

```
public Byte(byte value) {
    this.value = value;
}
```

```
public Byte(String s) throws NumberFormatException {
    this.value = parseByte(s, 10);
}
```

（10）获取 byte 值

覆写 Number 类中的抽象方法，返回用 byte 表示的该数的值。

```
public byte byteValue() {
    return value;
}
```

（11）获取 short 值

覆写 Number 类中的抽象方法，返回用 short 表示的该数的值。

```
public short shortValue() {
    return (short)value;
}
```

（12）获取 int 值

覆写 Number 类中的抽象方法，返回用 int 表示的该数的值。

```
public int intValue() {
    return (int)value;
}
```

（13）获取 long 值

覆写 Number 类中的抽象方法，返回用 long 表示的该数的值。

```
public long longValue() {
    return (long)value;
}
```

（14）获取 float 值

覆写 Number 类中的抽象方法，返回用 float 表示的该数的值。

```
public float floatValue() {
    return (float)value;
}
```

（15）获取 double 值

覆写 Number 类中的抽象方法，返回用 double 表示的该数的值。

```
public double doubleValue() {
    return (double)value;
}
```

（16）获取对象的哈希值

覆写 Object 类中的 hashCode( ) 方法，返回对象的哈希值，用于对象比较。

```
public int hashCode() {
    return (int)value;
}
```

（17）equals( )方法

覆写 Object 类的 equals( )方法，用于比较两个 Byte 对象的内容——即 byte 值。

```
public boolean equals(Object obj) {
    //先检测传入对象obj是否是Byte类型，防止类型不匹配引发的异常和错误
    if (obj instanceof Byte) {  //如果传入的是Byte对象
        return value == ((Byte)obj).byteValue();  //返回判断结果
    }
    return false;  //传入对象不是Byte类型，返回false
}
```

（18）compareTo( )方法

Byte 类实现了比较器接口，所以要重写 compareTo( )方法。该方法用于比较两个 Byte 对象存储的 byte 值。

```
public int compareTo(Byte anotherByte) {
    return compare(this.value, anotherByte.value);
}
```

```
public static int compare(byte x, byte y) {
    return x - y;
}
```

7. Integer 类

（1）Integer 类的声明

Integer 类同样是 Number 类的子类，并且实现了比较器的接口。由于被 final 修饰，该类不能被其他类继承。

```
package java.lang;
public final class Integer extends Number
      implements Comparable<Integer>
```

（2）Integer 类中存储整数值的成员变量 value

```
private final int value;
```

（3）Integer 类构造方法

构造方法有两种，一个是用 int 值初始化，另一个用 String 初始化。

```
public Integer(int value) {
    this.value = value;
}
public Integer(String s) throws NumberFormatException {
    this.value = parseInt(s, 10);  //后面会介绍parseInt方法
}
```

（4）int 型数值的最大值和最小值

int 的最大值和最小值是作为常量出现在 Integer 类中的。

通过计算机组成原理关于定点有符号整数的表示范围的相关知识可以知道：

字长为 32 位的定点有符号整数用补码表示，范围是-2^{31}～（$2^{31}-1$）

```
public static final int SIZE = 32;  //int 型数值用 32 位二进制数表示
public static final int  MIN_VALUE = 0x80000000; //16进制
//二进制：1000,0000,0000,0000,0000,0000,0000,0000 10进制：-2147483648
public static final int  MAX_VALUE = 0x7fffffff; //16 进制
//二进制：0111,1111,1111,1111,1111,1111,1111,1111 10 进制：2147483647
```

（5）toString( )方法

一个对象调用 toString( )方法相当于打印该对象的信息。

```
public String toString() {
    return toString(value);
}
public static String toString(int i) {
        if (i == Integer.MIN_VALUE)
            return "-2147483648";  //表示最小值的字符串
        //计算存储数字所需的字符位数，主要是数字的位数和符号位
        int size = (i < 0) ? stringSize(-i) + 1 : stringSize(i);
```

```
        char[] buf = new char[size];  //分配存储相应位数数字的空间
        getChars(i, size, buf);   //将数字对应的char序列buf[]
        return new String(buf, true);  //创建对应该字符串
}
```

```
static int stringSize(int x) {
    for (int i=0; ; i++)
        if (x <= sizeTable[i]) //找到存储x需要的位数
            return i+1; //该位数恰巧等于数组下标+1
}
```

```
final static int [] sizeTable = { 9, 99, 999, 9999, 99999, 999999, 9999999,
99999999, 999999999, Integer.MAX_VALUE };
//分别表示0~9位数的最大值以及int型数值的最大值
```

```
static void getChars(int i, int index, char[] buf) {
        int q, r;
        int charPos = index;
        char sign = 0;      //表示符号位，正数省略，负数'-'
        if (i < 0) {  //如果是负数，设置符号位
            sign = '-';
            i = -i;  //设置符号位后需要转换成i的绝对值
        }

        // Generate two digits per iteration  每次循环生成两个数字
        while (i >= 65536) {
            q = i / 100;  //每次循环可以操作两位
        // really: r = i - (q * 100);
            r = i - ((q << 6) + (q << 5) + (q << 2));
            i = q;
            buf [--charPos] = DigitOnes[r];
            buf [--charPos] = DigitTens[r];
        }
        // Fall thru to fast mode for smaller numbers
        // assert(i <= 65536, i);
        for (;;) {
            q = (i * 52429) >>> (16+3);
            r = i - ((q << 3) + (q << 1));  // r = i-(q*10) ...
            buf [--charPos] = digits [r];
            i = q;
            if (i == 0) break;
        }

        if (sign != 0) {
            buf [--charPos] = sign;
        }
}
```

```
final static char [] DigitOnes = {
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
} ;
```

```
final static char [] DigitTens = {
        '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
        '1', '1', '1', '1', '1', '1', '1', '1', '1', '1',
        '2', '2', '2', '2', '2', '2', '2', '2', '2', '2',
        '3', '3', '3', '3', '3', '3', '3', '3', '3', '3',
        '4', '4', '4', '4', '4', '4', '4', '4', '4', '4',
        '5', '5', '5', '5', '5', '5', '5', '5', '5', '5',
        '6', '6', '6', '6', '6', '6', '6', '6', '6', '6',
        '7', '7', '7', '7', '7', '7', '7', '7', '7', '7',
        '8', '8', '8', '8', '8', '8', '8', '8', '8', '8',
        '9', '9', '9', '9', '9', '9', '9', '9', '9', '9',
} ;
```

（6）String 转换 int——parseInt( )方法

```
public static int parseInt(String s) throws NumberFormatException {
    return parseInt(s,10);  //采用10进制转换
}
```

以下方法中用到Character类中的两个常量

```
public static final int MIN_RADIX = 2;  //最小进制——2进制
public static final int MAX_RADIX = 36; //最大进制——36进制
```

```
public static int parseInt(String s, int radix)  //radix是进制
      throws NumberFormatException{
        //先处理非法参数
        if (s == null) {   //先判断字符串是否为空，如果是，抛出异常
            throw new NumberFormatException("null");
        }
        if (radix < Character.MIN_RADIX) { //进制非法，小于最小进制2
            throw new NumberFormatException("radix " + radix + " less than
Character.MIN_RADIX");
        }
        if (radix > Character.MAX_RADIX) { //进制非法，大于最大进制36
            throw new NumberFormatException("radix " +radix+ " greater
```

```
than Character.MAX_RADIX");
        }
        int result = 0;
        boolean negative = false;     //数字是否是负数
        int i = 0, len = s.length();  //len数字字符串长度（可能含有负号）
        //i是循环控制变量，指示正在分析的字符，恰好是字符串value数组的下标
        int limit = -Integer.MAX_VALUE;
        int multmin;
        int digit;
        //进行字符串的分析、鉴别和处理
        if (len > 0) {  //如果字符串（含符号）不是空串
            char firstChar = s.charAt(0);  //获取字符串第一个字符
 /*先判断第一个字符是不是数字，借此鉴别是正数、负数还是非法串。如果第一个字符
不是数字，那么它要么是负号，要么是非数字非’-’（同时也不是’+’）的其他字符，是
负号说明字符串表示的是一个负数，如果是即非’-’又非’+’的非数字字符，说明是非法
字符串（含有非数字的不表示符号位字符的字符串）。*/
            if (firstChar < '0') {  //第一个字符不是数字
                if (firstChar == '-') {  //第一个字符是负号
                    negative = true;  //标志置为true，表示该数是负数
                    limit = Integer.MIN_VALUE;//负数最小不能超过MIN_VALUE
                } else if (firstChar != '+')  //不是’+’号，是非法串
                    throw NumberFormatException.forInputString(s);
                //如果字符串含负号的长度为1，说明后面没有数值位，
                //这样只含有符号位而无数值位的字符串也是非法串
                if (len == 1) // Cannot have lone "+" or "-"
                    throw NumberFormatException.forInputString(s);
                i++;
            }
            multmin = limit / radix;
            while (i < len) {
                // Accumulating negatively avoids surprises near MAX_VALUE
                digit = Character.digit(s.charAt(i++),radix);
                //将指定位置的数字字符变成10进制数字
                if (digit < 0) {  //转换出的数值应该是非负的，否则不合法
                    throw NumberFormatException.forInputString(s);
                }
                if (result < multmin) {
                    throw NumberFormatException.forInputString(s);
                }
                result *= radix;
                if (result < limit + digit) {
                    throw NumberFormatException.forInputString(s);
                }
                result -= digit;
```

```
            }
        } else {
            throw NumberFormatException.forInputString(s);
        }  //最后判断是正数还是负数，添加符号位
        return negative ? result : -result;
    }
```

（7）valueOf( )——返回存储的 int 值

```
public static Integer valueOf(String s) throws NumberFormatException{
    return Integer.valueOf(parseInt(s, 10));
}
```

```
public static Integer valueOf(String s, int radix)
        throws NumberFormatException {
    return Integer.valueOf(parseInt(s,radix));
}
```

```
public static Integer valueOf(int i) {
        assert IntegerCache.high >= 127;  //断言int值最大值为127
        if (i >= IntegerCache.low && i <= IntegerCache.high) //合法
            return IntegerCache.cache[i + (-IntegerCache.low)];
            //返回操作结果
        return new Integer(i);  //否则重新构建Integer对象
}
```

（8）获得 byte 值

覆写 Number 类中的抽象方法，返回用 byte 表示的该数的值。

```
public byte byteValue() {
    return (byte)value;
}
```

（9）获得 short 值

覆写 Number 类中的抽象方法，返回用 short 表示的该数的值。

```
public short shortValue() {
    return (short)value;
}
```

（10）获得 int 值

覆写 Number 类中的抽象方法，返回用 int 表示的该数的值。

```
public int intValue() {
    return value;
}
```

（11）获得 long 值

覆写 Number 类中的抽象方法，返回用 long 表示的该数的值。

```
public long longValue() {
    return (long)value;
}
```

（12）获得 float 值

覆写 Number 类中的抽象方法，返回用 float 表示的该数的值。

```
public float floatValue() {
    return (float)value;
}
```

（13）获得 double 值

覆写 Number 类中的抽象方法，返回用 double 表示的该数的值。

```
public double doubleValue() {
    return (double)value;
}
```

（14）hashCode( )方法

```
public int hashCode() {
    return value;
}
```

（15）equals( )方法

```
public boolean equals(Object obj) {  //用Object类接收比较安全
        if (obj instanceof Integer) { //先判断是否是Integer对象
            return value == ((Integer)obj).intValue(); //返回比较结果
        }
        return false; //传入的obj不是Integer类对象，返回false
}
```

（16）compareTo( )方法

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

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

8. Runtime 类

（1）Runtime 类的声明

```
package java.lang;
public class Runtime
```

（2）Runtime 实例对象的获得——单态模式的应用

Runtime 类不像其他类，由于其构造方法被私有化，不能通过调用构造方法实例化，而是通过调用静态方法间接调用构造方法来完成，这是单态模式的典型应用。

```
private static Runtime currentRuntime = new Runtime();
private Runtime() {}   //构造方法被私有化
public static Runtime getRuntime() {
    return currentRuntime;
}
```

（3）获得 JVM 的最大内存量

```
public native long maxMemory();
```

（4）获得 Java 虚拟机中的空闲内存量

```
public native long freeMemory();
```

（5）运行垃圾回收器，释放空间

垃圾回收器可以回收垃圾空间，释放空间。

```
public native void gc();
```

（6）执行本机命令

```
public Process exec(String command) throws IOException {
    return exec(command, null, null);
}
public Process exec(String command, String[] envp, File dir)
        throws IOException {
        if (command.length() == 0)
            throw new IllegalArgumentException("Empty command");
        //提取命令的名称
        StringTokenizer st = new StringTokenizer(command);
        String[] cmdarray = new String[st.countTokens()];
        for (int i = 0; st.hasMoreTokens(); i++)
            cmdarray[i] = st.nextToken();
        return exec(cmdarray, envp, dir);  //返回一个Process对象
        //Process对象表示一个操作系统的进程类
}
```

```
public Process exec(String[] cmdarray, String[] envp, File dir)
        throws IOException {
        return new ProcessBuilder(cmdarray).environment(envp)
.directory(dir).start();   //创建并启动相关进程
}
```

```
// ProcessBuilder类的构造方法
public ProcessBuilder(String... command) {
        this.command = new ArrayList<>(command.length);
        for (String arg : command)
            this.command.add(arg);
}
```

9. Locale 类

Locale 类用于表示一个国家语言类，是实现国际化程序的重要类。

（1）Locale 类的声明

该类被 final 修饰，没有子类。该类可以被序列化，可以进行对象克隆操作。

```
package java.util;
public final class Locale implements Cloneable, Serializable
```

（2）一些表示国家和地区的重要的常量

```
static public final Locale
     SIMPLIFIED_CHINESE = createConstant("zh", "CN");
//中文-中国
static public final Locale FRANCE = createConstant("fr", "FR");
//法国-法语
static public final Locale GERMANY = createConstant("de", "DE");
//德国-德语
static public final Locale JAPAN = createConstant("ja", "JP");
//日本-日语
static public final Locale KOREA = createConstant("ko", "KR");
//韩国-韩语
static public final Locale CHINA = SIMPLIFIED_CHINESE;
//简体中文
static public final Locale US = createConstant("en", "US");
//英文-美国
```

（3）Locale 类的构造方法

```
public Locale(String language) {
       this(language, "", "");
}
public Locale(String language, String country) {
       this(language, country, "");
}
public Locale(String language, String country, String variant) {
    if (language== null || country == null || variant == null) {
        throw new NullPointerException();
    }
    baseLocale = BaseLocale.getInstance(convertOldISOCodes(language)
, "", country, variant);
    localeExtensions = getCompatibilityExtensions(language, "",
country , variant);
}
```

10. ResourceBundle 类

ResourceBundle 类的作用是读取属性文件，获取其中的内容。

（1）ResourceBundle 类的声明

可以看出，ResourceBundle 类是一个抽象类。

```
package java.util;
public abstract class ResourceBundle
```

（2）获取 ResourceBundle 类实例

虽然该类有构造方法，但因为是抽象类，无法使用构造方法获得类的实例，要使用 getBundle 方法获得实例。

```
public ResourceBundle(){}
public static final ResourceBundle getBundle(String baseName){
        return getBundleImpl(baseName, Locale.getDefault(),
getLoader(Reflection.getCallerClass()),Control.INSTANCE);
}
public static final ResourceBundle getBundle(String baseName,
      Locale locale){
        return getBundleImpl(baseName, locale,
getLoader(Reflection.getCallerClass()),Control.INSTANCE);
}
```

（3）根据 key 从对应的资源文件中取出 value

```
public final String getString(String key) {
    return (String) getObject(key);
}
public final Object getObject(String key) {
        Object obj = handleGetObject(key);
        if (obj == null) {
            if (parent != null) {
                obj = parent.getObject(key);
            }
            if (obj == null)
                throw new MissingResourceException("Can't find resource
for bundle "+this.getClass().getName()+", key "+key,
this.getClass().getName(),key);
        }
        return obj;
}
```

11. Serializable 接口

（1）Sreializable 接口的声明和定义

Serializable 接口是一个标识接口，表示一个类具备了被序列化的能力，该接口本身不含有任何抽象方法。实现该接口后声明对象的内容将会被自动序列化，无需用户手动干预。

```
package java.io;
public interface Serializable {}
```

（2）transient 关键字的使用

如果有内容不想被序列化，可以使用 transient 关键字修饰，该关键字修饰的属性不会被序列化。

例如：java.util.ArrayList 类就实现了这一接口。

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

在 java.util.ArrayList 类中，对象数组 elementData 被 transient 关键字修饰，表示该数组不会被序列化。

```
private transient Object[] elementData;
```

使用 Serializable 接口 + transient 关键字可以代替 Externalizable 接口，使用较为简便。

12. Externalizable 接口

如果用户想手动指定要被序列化的内容，要用 Externalizable 接口。但其使用较复杂，建议使用 Serializable 接口 + transient 关键字。

（1）Externalizable 接口声明定义

从继承关系看出 Externalizable 接口是 Serializable 接口的子接口。
该接口中只有两个抽象方法。

```
package java.io;
public interface Externalizable extends java.io.Serializable
```

（2）手动指定要保存的属性

该方法指定要保存的属性信息，对象序列化时调用。

```
void writeExternal(ObjectOutput out) throws IOException;
public interface ObjectOutput extends DataOutput, AutoCloseable
```

（3）读取被保存的信息

该方法中读取被保存的信息，对象反序列化调用。

```
void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException;
public interface ObjectInput extends DataInput, AutoCloseable
```

以上两个接口分别继承了 DataOutput 和 DataInput，这样在这两个方法中就可以像 DataOutputStream 和 DataInputStream 那样直接输出和读取各种类型的数据。

13. Cloneable 接口

（1）Cloneable 接口的声明

一个类实现了 Cloneable 接口则该类声明的对象可以被克隆。

Cloneable 接口和 Serializable 接口一样，都是标识接口。Cloneable 接口中仍然没有任何方法定义。

```
package java.lang;
public interface Cloneable {}
```

（2）clone( )方法——对象克隆

如果一个对象想要得到它的克隆对象，必须重写 Object 类中的 clone( )方法，并在其中调用从 Object 类中继承而来的 clone( )方法。

Object 类中关于 clone( )方法的声明。native 关键字表示调用本机操作系统的函数。

```
protected native Object clone() throws CloneNotSupportedException;
```

例如：（ArrayList 类中的 clone( )方法）

```
public Object clone() {
        try {
            @SuppressWarnings("unchecked")
                ArrayList<E> v = (ArrayList<E>) super.clone();
            v.elementData = Arrays.copyOf(elementData, size);
            v.modCount = 0;
            return v;
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError();
        }
}
```

Format 类中的 clone( )方法

```
public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {  // will never happen
        return null;
    }
}
```

14. Format 类

（1）Format 类的声明

从声明可以看出，Format 类是一个抽象类，并且实现了 Serializable 和 Cloneable 接口。

```
package java.text;
public abstract class Format implements Serializable, Cloneable
```

（2）clone( )方法

```
public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {  // will never happen
            return null;
        }
}
```

（3）format( )方法

```
public final String format (Object obj) {
   return format(obj, new StringBuffer(),new FieldPosition(0)).toString();
}
public abstract StringBuffer format(Object obj,StringBuffer toAppendTo,
      FieldPosition pos);
```

15. MessageFormat 类

（1）MessageFormat 类的声明

可以看出，MessageFormat 类是 Format 类的子类。体现了抽象类的模板设计思想。

```
package java.text;
public class MessageFormat extends Format
```

（2）format 方法

```
public static String format(String pattern, Object ... arguments) {
    MessageFormat temp = new MessageFormat(pattern);
    return temp.format(arguments);  //调用父类中的format()方法
}
```

16. ListResourceBundle 类

（1）ListResourceBundle 类声明

可以看出，ListResourceBundle 类是 ResourceBundle 类的子类，而且它还是一个抽象类。如果想要使用类替代资源文件，可以继承该类并覆写 getContents( )方法。

```
package java.util;
public abstract class ListResourceBundle extends ResourceBundle
```

（2）getContents( )方法

可以看出，该方法是抽象方法。等待被继承的子类覆写。

```
abstract protected Object[][] getContents();
```

17. Comparable 接口 （比较器）

（1）Comparable 接口的声明

Comparable 接口用于指定排序时类的比较规则。如果待比较类未实现这个接口，将会出现 ClassCastException（类型转换异常），因为所有对象比较时都向 Comparable 进行转换。

```
package java.lang;
public interface Comparable<T>
```

（2）接口中唯一的抽象方法——compareTo( )方法

实现比较器的类通过重写 compareTo( )方法制定比较规则，该比较规则将被用于排序时的对象比较。排序由 Arrays.sort( )方法完成。

该类的返回值为 int，表示比较结果，规定如下：1 表示大于　-1 表示小于　0 表示等于

```
public int compareTo(T o);
```

18. Comparator 接口（另一种比较器）

如果一些类已经开发完成，但是此类建立初期未实现 Comparable 接口，肯定是不能进行对象比较的，为了解决这样的问题，Java 又定义了另一个比较器操作接口 Comparator。

可以看出，Comparator 接口明显是一种补救措施，但由于其使用不如 Comparable 接口方便，所以不推荐使用。

（1）Comparator 接口声明

Comparator 接口中只有两个抽象方法。

```
package java.util;  //说明该类定义在 java.util 包中
public interface Comparator<T>
```

（2）compareTo( )方法

此方法用于制定比较规则，但与之前不同的是，该方法接收两个对象。

```
int compare(T o1, T o2);
```

（3）equals( )方法

此方法用于判断两个对象是否相等。

```
boolean equals(Object obj);
```

19. Date 类

Date 类是一个操作较简单的类，调用其构造方法可以得到完整的日期。

（1）Date 类声明

```
package java.util;
```

```
public class Date
    implements java.io.Serializable, Cloneable, Comparable<Date>
```

（2）Date 类构造方法

```
public Date() {
    this(System.currentTimeMillis());  //调用重载的构造方法实现
}
```

```
public Date(long date) {
    fastTime = date;
}
```

```
private transient long fastTime;  //fastTime保存Date类中的时间
//由transient关键字知该对象不会被序列化
```

（3）toString()方法

打印一个对象就是调用其 toString()方法

```
public String toString() {
        // "EEE MMM dd HH:mm:ss zzz yyyy";
        BaseCalendar.Date date = normalize();
        StringBuilder sb = new StringBuilder(28);
        int index = date.getDayOfWeek();
        if (index == gcal.SUNDAY) {
            index = 8;
        }
        convertToAbbr(sb, wtb[index]).append(' ');      // EEE
        convertToAbbr(sb, wtb[date.getMonth() - 1 + 2 + 7]).append(' ');
// MMM
        CalendarUtils.sprintf0d(sb, date.getDayOfMonth(), 2).append('
```

```
'); // dd
        CalendarUtils.sprintf0d(sb, date.getHours(), 2).append(':');
// HH
        CalendarUtils.sprintf0d(sb, date.getMinutes(), 2).append(':');
// mm
        CalendarUtils.sprintf0d(sb, date.getSeconds(), 2).append(' ');
// ss
        TimeZone zi = date.getZone();
        if (zi != null) {
            sb.append(zi.getDisplayName(date.isDaylightTime(),
zi.SHORT, Locale.US)); // zzz
        } else {
            sb.append("GMT");
        }
        sb.append(' ').append(date.getYear());  // yyyy
        return sb.toString();
}
```

（4）getTime( )

getTime 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。

```
public long getTime() {
        return getTimeImpl();
}
```

```
private final long getTimeImpl() {
        if (cdate != null && !cdate.isNormalized()) {
            normalize();
        }
        return fastTime;
}
```

（5）equals（）方法

该方法用于比较日期是否相等。

```
public boolean equals(Object obj) {
    return obj instanceof Date && getTime() == ((Date) obj).getTime();
}
```

（6）compareTo( )方法

```
public int compareTo(Date anotherDate) {
        long thisTime = getMillisOf(this);
        long anotherTime = getMillisOf(anotherDate);
        return (thisTime<anotherTime ?
                -1 : (thisTime==anotherTime ? 0 : 1));
}
```

20. System 类

System 类是一些与系统相关的属性和方法的集合。System 类中的属性和方法都是静态的，通过类名就可以直接调用。

（1）System 类的声明

System 类被 final 修饰说明该类不能被继承，System 类没有子类。

```
public final class System
```

（2）System 类没有实例

System 类的构造方法被私有化，并且不是单态模式。System 类源码中的一句注释告诉我们不允许任何人得到该类的实例化对象。因此我们只能通过类名引用该类的属性和调用该类的方法。

```
/** Don't let anyone instantiate this class */
private System() {}
```

（3）exit( )方法——系统退出

如果 status 为非 0 表示异常退出。

```
public static void exit(int status) {   //System类中的exit()方法
        Runtime.getRuntime().exit(status);
}
```

```
public void exit(int status) {   //Runtime类中的exit()方法
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkExit(status);
        }
        Shutdown.exit(status);
}
```

```
static void exit(int status) {  //Shutdown类中的exit()方法
        boolean runMoreFinalizers = false;
        synchronized (lock) {
            if (status != 0)  //status非0表示异常退出
                runFinalizersOnExit = false;
            switch (state) {
            case RUNNING:       /* Initiate shutdown */
                state = HOOKS;
                break;
            case HOOKS:         /* Stall and halt */
                break;
            case FINALIZERS:
                if (status != 0) {
                    /* Halt immediately on nonzero status */
```

```
                    halt(status);
                } else {
                    /* Compatibility with old behavior:
                        Run more finalizers and then halt */
                    runMoreFinalizers = runFinalizersOnExit;
                }
                break;
            }
        }
        if (runMoreFinalizers) {
            runAllFinalizers();
            halt(status);
        }
        synchronized (Shutdown.class) {
         /* Synchronize on the class object, causing any other thread
          * that attempts to initiate shutdown to stall indefinitely*/
            sequence();
            halt(status);
        }
}
```

（4）gc( )方法——垃圾回收

System.gc();就是启动垃圾回收线程的语句。当用户认为需要回收时，可以使用 Runtime.getRuntime( ).gc( );或者 System.gc();来回收内存。

System.gc();调用的就是 Runtime 类的 gc( )方法。

```
public static void gc() {
        Runtime.getRuntime().gc();  //调用Runtime类的gc()方法
}
```

```
public native void gc();   //Runtime类中的gc()方法
//native表示调用本机操作系统函数
```

（5）获得当前系统时间

currentTimeMillis( )方法返回以毫秒为单位的当前时间。

native 表示调用本机操作系统的函数。

```
public static native long currentTimeMillis();
```

（6）数组复制

arrayCopy( )是 System 类提供的数组复制方法，该方法广泛应用于 Java 的很多类中，比如 String 类的字符串处理函数复制底层 char 数组时就是调用这一方法完成的。

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

native 表示调用本机操作系统的函数。

（7）取得当前系统的全部属性

System 类有一个静态成员变量 props，保存了当前系统的全部属性

```
private static Properties props;
```

通过调用 System 类中的 getProperties( )方法可以获得上述的静态成员变量。

```
public static Properties getProperties() {
        SecurityManager sm = getSecurityManager();
        if (sm != null) {
            sm.checkPropertiesAccess();
        }
        return props;
}
```

（8）根据键值取得属性的具体内容

使用 getProperty(String key)方法可以根据键值 key 取得属性的具体内容。

```
public static String getProperty(String key) {
        checkKey(key);
        SecurityManager sm = getSecurityManager();
        if (sm != null) {
            sm.checkPropertyAccess(key);
        }
        return props.getProperty(key);  //调用Properties类同名方法
}
```

```
public String getProperty(String key) {
        Object oval = super.get(key);
        String sval = (oval instanceof String) ? (String)oval : null;
        return ((sval == null) && (defaults != null)) ?
                defaults.getProperty(key) : sval;
}
```

21. Calendar 类

Calendar 类可以把取得的时间精确到毫秒。

（1）Calendar 类的声明

Calendar 类是一个抽象类，必须依靠对象多态性，通过其子类对父类进行实例化操作。Calendar 的子类是 GregorianCalendar 类。

```
package java.util;
```

```
public abstract class Calendar
     implements Serializable, Cloneable, Comparable<Calendar>
```

```
public class GregorianCalendar extends Calendar
```

（2）Calendar 类中的常量

`public final static int YEAR = 1;`	取得年
`public final static int MONTH = 2;`	取得月，从 0 开始计，获得结果要+1
`public final static int DAY_OF_MONTH = 5;`	取得日
`public final static int HOUR_OF_DAY = 11;`	取得小时，24 小时制
`public final static int MINUTE = 12;`	取得分
`public final static int SECOND = 13;`	取得秒
`public final static int MILLISECOND = 14;`	取得毫秒

（3）根据默认的时区实例化对象

注意，这里获得的不是 Calendar 类的实例，因为 Calendar 类是抽象类，不能有实例化对象，所以这里获取的是其子类（GregorianCalendar 类）的对象。

```
public static Calendar getInstance(){
    Calendar cal = createCalendar(TimeZone.getDefaultRef(),
          Locale.getDefault(Locale.Category.FORMAT));
    cal.sharedZone = true;
    return cal;
}
```

```
private static Calendar createCalendar(TimeZone zone,Locale aLocale){
        Calendar cal = null;
        String caltype = aLocale.getUnicodeLocaleType("ca");
        if (caltype == null) {
            // Calendar type is not specified.
            // If the specified locale is a Thai locale,
            // returns a BuddhistCalendar instance.
            if ("th".equals(aLocale.getLanguage())
                    && ("TH".equals(aLocale.getCountry()))) {
                cal = new BuddhistCalendar(zone, aLocale);
            } else {
                cal = new GregorianCalendar(zone, aLocale);
            }
        } else if (caltype.equals("japanese")) {
            cal = new JapaneseImperialCalendar(zone, aLocale);
        } else if (caltype.equals("buddhist")) {
            cal = new BuddhistCalendar(zone, aLocale);
        } else {
            // Unsupported calendar type.
            // Use Gregorian calendar as a fallback.
            cal = new GregorianCalendar(zone, aLocale);
        }
        return cal;
    }
```

（4）compareTo()方法

该方法实际比较的其实是两个日期转换成的毫秒数。

```
public int compareTo(Calendar anotherCalendar) {
    return compareTo(getMillisOf(anotherCalendar));
}
```

```
private static final long getMillisOf(Calendar calendar) {
    //获得日期的毫秒表示
    if (calendar.isTimeSet) {
        return calendar.time;
    }
    Calendar cal = (Calendar) calendar.clone();
    cal.setLenient(true);
    return cal.getTimeInMillis();
}
```

```
private int compareTo(long t) {
    long thisTime = getMillisOf(this);
    //真正比较的其实是毫秒数
    return (thisTime > t) ? 1 : (thisTime == t) ? 0 : -1;
}
```

（5）判断一个日期是否在另一个日期之前

```
public boolean before(Object when) {
    return when instanceof Calendar //首先确保必须是Calendar类型的实例
         && compareTo((Calendar)when) < 0; //再进行比较
}
```

（6）判断一个日期是否在另一个日期之后

```
public boolean after(Object when) {
    return when instanceof Calendar //首先确保必须是Calendar类型的实例
         && compareTo((Calendar)when) > 0; //再进行比较
}
```

（7）返回给定日历字段的值

```
public int get(int field){
    complete();
    return internalGet(field);
}
```

22. DateFormat 类

（1）DateFormat 类的声明

DateFormat 类也是一个抽象类，它继承了 Format 类。该类无法通过构造方法直接获得实例化对象，但是在此抽象类中提供了静态方法，可以直接取得本类的实例。

```
package java.text;
public abstract class DateFormat extends Format
```

（2）获取实例化对象

可以通过以下方法获得该类的实例化对象

```
public final static DateFormat getDateInstance(){ //得到默认对象
    return get(0, DEFAULT, 2,
          Locale.getDefault(Locale.Category.FORMAT));
}
```

```
public final static DateFormat getDateInstance(int style,
    Locale aLocale){  //根据Locale得到对象
    return get(0, style, 2, aLocale);
}
```

```
public final static DateFormat getDateTimeInstance(){//得到日期时间对象
    return get(DEFAULT, DEFAULT, 3,
          Locale.getDefault(Locale.Category.FORMAT));
}
```

```
public final static DateFormat getDateTimeInstance(int dateStyle, int
timeStyle){  //根据Locale得到日期时间对象
    return get(timeStyle, dateStyle, 3,
          Locale.getDefault(Locale.Category.FORMAT));
}
```

（3）Date 类型转换 String 类型

```
public final String format(Date date){
       return format(date, new StringBuffer(),
                     DontCareFieldPosition.INSTANCE).toString();
}
```

```
public abstract StringBuffer format(Date date,
       StringBuffer toAppendTo, FieldPosition fieldPosition);
```

23. SimpleDateFormat 类

SimpleDateFormat 类可以实现比 DateFormat 类更方便的操作，并且可以把日期转换成自己想要的类型。

（1）SimpleDateFormat 类声明

SimpleDateFormat 类是 DateFormat 类的子类，可以实现比 DateFormat 类更方便的操作。

```
package java.text;
public class SimpleDateFormat extends DateFormat
```

（2）SimpleDateFormat 类的构造方法

```
private String pattern;   //日期格式化的模板
public SimpleDateFormat(String pattern){  //通过指定模板构造对象
    this(pattern, Locale.getDefault(Locale.Category.FORMAT));
}
public SimpleDateFormat(String pattern, Locale locale){
    if (pattern == null || locale == null) {
        throw new NullPointerException();
    }
    initializeCalendar(locale);
    this.pattern = pattern;
    this.formatData = DateFormatSymbols.getInstanceRef(locale);
    this.locale = locale;
    initialize(locale);
}
```

（3）将包含日期格式的字符串转换为 Date 类型

```
public Date parse(String text, ParsePosition pos){
    checkNegativeNumberExpression();
    int start = pos.index;
    int oldStart = start;
    int textLength = text.length();
    boolean[] ambiguousYear = {false};
    CalendarBuilder calb = new CalendarBuilder();

    for (int i = 0; i < compiledPattern.length; ) {
        int tag = compiledPattern[i] >>> 8;
        int count = compiledPattern[i++] & 0xff;
        if (count == 255) {
            count = compiledPattern[i++] << 16;
            count |= compiledPattern[i++];
        }
```

```
        switch (tag) {
        case TAG_QUOTE_ASCII_CHAR:
            if (start >= textLength ||
                    text.charAt(start) != (char)count) {
                pos.index = oldStart;
                pos.errorIndex = start;
                return null;
            }
            start++;
            break;

        case TAG_QUOTE_CHARS:
            while (count-- > 0) {
                if (start >= textLength ||
                        text.charAt(start) != compiledPattern[i++]) {
                    pos.index = oldStart;
                    pos.errorIndex = start;
                    return null;
                }
                start++;
            }
            break;

        default:
            boolean obeyCount = false;
            boolean useFollowingMinusSignAsDelimiter = false;
            if (i < compiledPattern.length) {
                int nextTag = compiledPattern[i] >>> 8;
                if (!(nextTag == TAG_QUOTE_ASCII_CHAR ||
                      nextTag == TAG_QUOTE_CHARS)) {
                    obeyCount = true;
                }
                if (hasFollowingMinusSign &&
                    (nextTag == TAG_QUOTE_ASCII_CHAR ||
                     nextTag == TAG_QUOTE_CHARS)) {
                    int c;
                    if (nextTag == TAG_QUOTE_ASCII_CHAR) {
                        c = compiledPattern[i] & 0xff;
                    } else {
                        c = compiledPattern[i+1];
                    }

                    if (c == minusSign) {
                        useFollowingMinusSignAsDelimiter = true;
```

```
                    }
                }
            }
            start = subParse(text, start, tag, count, obeyCount,
                             ambiguousYear, pos,
                             useFollowingMinusSignAsDelimiter, calb);
            if (start < 0) {
                pos.index = oldStart;
                return null;
            }
        }
    }
    pos.index = start;
    Date parsedDate;
    try {
        parsedDate = calb.establish(calendar).getTime();
        if (ambiguousYear[0]) {
            if (parsedDate.before(defaultCenturyStart)) {
                parsedDate =
calb.addYear(100).establish(calendar).getTime();
            }
        }
    }
    catch (IllegalArgumentException e) {
        pos.errorIndex = start;
        pos.index = oldStart;
        return null;
    }
    return parsedDate;
}
```

（4）将一个 Date 类型按照指定格式变为 String 类型

DateFormat 类中有一个抽象方法:

```
public abstract StringBuffer format(Date date,
    StringBuffer toAppendTo, FieldPosition fieldPosition);
```

SimpleDateFormat 类实现了这个方法，可以将一个 Date 类型按照指定格式变为 String 类型。

```
public StringBuffer format(Date date, StringBuffer toAppendTo,
    FieldPosition pos) {
    pos.beginIndex = pos.endIndex = 0;
    return format(date, toAppendTo, pos.getFieldDelegate());
}
```

24. Math 类

Math 类是一个数学操作类，提供了一系列数学操作方法，包括求绝对值、三角函数等。Math 类和 System 类一样，所有的方法都是静态方法，直接由类名调用即可。

（1）Math 类的声明

Math 类被 final 修饰，不能有子类。

```
public final class Math
```

（2）Math 类没有实例

和 System 类一样，Math 类的构造方法被私有化，并且不是单态模式。Math 类源码中的一句注释告诉我们不允许任何人得到该类的实例化对象。因此我们只能通过类名引用该类的属性和调用该类的方法。

```
/*** Don't let anyone instantiate this class.*/
private Math() {}
```

（3）圆周率的定义

Math 类中有一些重要的数学常量的定义，圆周率就是其中之一。

```
public static final double PI = 3.14159265358979323846;
```

（4）求 sin(x)的值

Math 类调用 StrictMath 类的 sin()方法求得 sin 值。

```
public static double sin(double a) {
    return StrictMath.sin(a);
}
```

```
public static native double sin(double a);  //StrictMath类的sin()方法
```

（5）求 cos(x)的值

Math 类调用 StrictMath 类的 cos()方法求得 cos 值。

```
public static double cos(double a) {
    return StrictMath.cos(a);
}
```

```
public static native double cos(double a);  //StrictMath类的cos()方法
```

（6）求 tan(x)的值

Math 类调用 StrictMath 类的 tan()方法求得 tan 值。

```
public static double tan(double a) {
    return StrictMath.tan(a);
}
```

```
public static native double tan(double a); //StrictMath类的tan()方法
```

（7）求 e 的 x 次幂

```
public static double exp(double a) {
        return StrictMath.exp(a);
}
```

```
public static native double exp(double a); //StrictMath类的exp()方法
```

（8）求 ln(x)的值

log 以 e 为底 x 的对数。

```
public static double log(double a) {
    return StrictMath.log(a);
}
```

```
public static native double log(double a); //StrictMath类的log()方法
```

（9）求 lg(x)的值

log 以 10 为底 x 的值。

```
public static double log10(double a) {
    return StrictMath.log10(a);
}
```

```
public static native double log10(double a); //StrictMath类的log10()方法
```

（10）求$\sqrt{x}$的值

```
public static double sqrt(double a) {
    return StrictMath.sqrt(a);
}
```

```
public static native double sqrt(double a); //StrictMath类的sqrt()方法
```

（11）对一个小数向上取整（入）

```
public static double ceil(double a) {
    return StrictMath.ceil(a);
}
```

```
public static double ceil(double a) {
    return floorOrCeil(a, -0.0, 1.0, 1.0); //StrictMath类的ceil()方法
}
```

（12）对一个小数向上取整（舍）

```
public static double floor(double a) {
    return StrictMath.floor(a);
}
```

```
public static double floor(double a) {
    return floorOrCeil(a, -1.0, 0.0, -1.0); //StrictMath类的floor()方法
}
```

（13）返回 x 的 y 次幂

返回第一个参数的第二个参数次幂的值。

```
public static double pow(double a, double b) {
    return StrictMath.pow(a, b);
}
```

```
public static native double pow(double a, double b);
    //StrictMath类的pow()方法
```

（14）求 x 的绝对值

该方法被重载多次，用于求各种类型数字的绝对值

```
public static int abs(int a) {
    return (a < 0) ? -a : a;
}
```

```
public static long abs(long a) {
    return (a < 0) ? -a : a;
}
```

```
public static float abs(float a) {
    return (a <= 0.0F) ? 0.0F - a : a;
}
```

```
public static double abs(double a) {
    return (a <= 0.0D) ? 0.0D - a : a;
}
```

（15）进行四舍五入操作

此方法操作时会将小数点后面的全部数字忽略掉，如果想精确到指定位数，必须用 BidDecimal 类完成。

```
public static int round(float a) {
    if (a != 0x1.fffffep-2f)
        return (int)floor(a + 0.5f);
    else
        return 0;
}
```

（16）获得随机数

返回带正号的 double 值，该值大于等于 0.0 且小于 1.0

```
public static double random() {
    Random rnd = randomNumberGenerator;
    if (rnd == null)
        rnd = initRNG();
    return rnd.nextDouble();
}
```

（17）求两数中的最大值

该方法被重载多次，用于求各种类型数字的最大值。

```
public static int max(int a, int b) {
    return (a >= b) ? a : b;
}
```

```
public static long max(long a, long b) {
    return (a >= b) ? a : b;
}
```

```
public static float max(float a, float b) {
    if (a != a) return a;   // a is NaN（非数）
    if ((a == 0.0f) && (b == 0.0f)
        && (Float.floatToIntBits(a) == negativeZeroFloatBits)) {
        return b;
    }
    return (a >= b) ? a : b;
}
```

```
public static double max(double a, double b) {
    if (a != a) return a;   // a is NaN（非数）
    if ((a == 0.0d) && (b == 0.0d)
        && (Double.doubleToLongBits(a) == negativeZeroDoubleBits)) {
        return b;
    }
    return (a >= b) ? a : b;
}
```

（18）求两数中的最小值

该方法被重载多次，用于求各种类型数字的最小值。

```
public static int min(int a, int b) {
    return (a <= b) ? a : b;
}
```

```
public static long min(long a, long b) {
    return (a <= b) ? a : b;
}
```

```
public static float min(float a, float b) {
    if (a != a) return a;   // a is NaN（非数）
    if ((a == 0.0f) && (b == 0.0f)
        && (Float.floatToIntBits(b) == negativeZeroFloatBits)) {
        return b;
    }
    return (a <= b) ? a : b;
}
```

```
public static double min(double a, double b) {
    if (a != a) return a;   // a is NaN（非数）
```

```
    if ((a == 0.0d) && (b == 0.0d)
        && (Double.doubleToLongBits(b) == negativeZeroDoubleBits)) {
        return b;
    }
    return (a <= b) ? a : b;
}
```

25. Arrays 类

（1）Arrays 类声明

```
package java.util;
public class Arrays
```

（2）Arrays 类没有实例

同 System 类和 Math 类一样

```
// Suppresses default constructor, ensuring non-instantiability.
// 抑制默认的构造器（无参构造），确保该类无法被实例化
/* Java语法中，如果用户不声明构造方法，系统将会默认生成一个什么都不做的无参构造方法，而如果用户声明一个及以上的构造方法则不会生成默认的构造方法。
为了保证该类没有实例化对象，就要封堵实例化的必经之路——构造器。
首先要保证系统不会生成默认的构造方法，因为默认的构造方法是用public修饰的，用户可以访问，可以自己写一个构造方法，这样系统就不会自动生成；然后要保证自己写的构造方方法用户无法访问到，这就是要用private修饰该构造方法，将其私有化，使其无法被外部调用，造成虽然形式上有构造方法但实际不可用的情况，这样就达到了使这个类无法被实例化的效果。 */
private Arrays() {}  //重要思想：构造器私有化
```

（3）数组排序

```
public static void sort(int[] a) {
    DualPivotQuicksort.sort(a);
}
```

阅读算法可以知道主要用的是插入排序和一种改进的快速排序，默认是插入排序。

```
private static void sort(int[] a, int left, int right, boolean leftmost){
    int length = right - left + 1; // Use insertion sort on tiny arrays
    if (length < INSERTION_SORT_THRESHOLD) {
        if (leftmost) {
            /* Traditional (without sentinel) insertion sort,
             * optimized for server VM, is used in case of
             * the leftmost part. */
            for (int i = left, j = i; i < right; j = ++i) {
                int ai = a[i + 1];
                while (ai < a[j]) {
```

```
                    a[j + 1] = a[j];
                    if (j-- == left) {
                        break;
                    }
                }
                a[j + 1] = ai;
            }
        } else {
            //Skip the longest ascending sequence.
            do {
                if (left >= right) {
                    return;
                }
            } while (a[++left] >= a[left - 1]);
            /* Every element from adjoining part plays the role
             * of sentinel, therefore this allows us to avoid the
             * left range check on each iteration. Moreover, we use
             * the more optimized algorithm, so called pair insertion
             * sort, which is faster (in the context of Quicksort)
             * than traditional implementation of insertion sort.*/
            for (int k = left; ++left <= right; k = ++left) {
                int a1 = a[k], a2 = a[left];
                if (a1 < a2) {
                    a2 = a1; a1 = a[left];
                }
                while (a1 < a[--k]) {
                    a[k + 2] = a[k];
                }
                a[++k + 1] = a1;
                while (a2 < a[--k]) {
                    a[k + 1] = a[k];
                }
                a[k + 1] = a2;
            }
            int last = a[right];

            while (last < a[--right]) {
                a[right + 1] = a[right];
            }
            a[right + 1] = last;
        }
        return;
    }
    // Inexpensive approximation of length / 7
```

```
    int seventh = (length >> 3) + (length >> 6) + 1;
    /* Sort five evenly spaced elements around (and including) the
     * center element in the range. These elements will be used for
     * pivot selection as described below. The choice for spacing
     * these elements was empirically determined to work well on
     * a wide variety of inputs.*/
    int e3 = (left + right) >>> 1; // The midpoint
    int e2 = e3 - seventh;
    int e1 = e2 - seventh;
    int e4 = e3 + seventh;
    int e5 = e4 + seventh;
    // Sort these elements using insertion sort
    if (a[e2] < a[e1]) { int t = a[e2]; a[e2] = a[e1]; a[e1] = t; }
    if (a[e3] < a[e2]) { int t = a[e3]; a[e3] = a[e2]; a[e2] = t;
        if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
    }
    if (a[e4] < a[e3]) { int t = a[e4]; a[e4] = a[e3]; a[e3] = t;
        if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
            if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
        }
    }
    if (a[e5] < a[e4]) { int t = a[e5]; a[e5] = a[e4]; a[e4] = t;
        if (t < a[e3]) { a[e4] = a[e3]; a[e3] = t;
            if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
                if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
            }
        }
    }
    // Pointers
    int less  = left;  // The index of the first element of center part
    int great = right;// The index before the first element of right part
    if (a[e1] != a[e2] && a[e2] != a[e3] && a[e3] != a[e4]
        && a[e4] != a[e5]) {
        /*Use the second and fourth of the five sorted elements as pivots.
         * These values are inexpensive approximations of the first and
         * second terciles of the array. Note that pivot1 <= pivot2.*/
        int pivot1 = a[e2];
        int pivot2 = a[e4];
        /* The first and the last elements to be sorted are moved to the
         * locations formerly occupied by the pivots. When partitioning
         * is complete, the pivots are swapped back into their final
         * positions, and excluded from subsequent sorting. */
        a[e2] = a[left];
        a[e4] = a[right];
```

```
        /* Skip elements, which are less or greater than pivot values.*/
        while (a[++less] < pivot1);
        while (a[--great] > pivot2);
       /* Partitioning:
        *
        *   left part          center part                  right part
        * +--------------------------------------------------------------+
        * |  < pivot1  |  pivot1 <= && <= pivot2  |    ?    |  > pivot2  |
        * +--------------------------------------------------------------+
        *              ^                          ^       ^
        *              |                          |       |
        *            less                         k     great
        * Invariants:
        *              all in (left, less)   < pivot1
        *    pivot1 <= all in [less, k)     <= pivot2
        *              all in (great, right) > pivot2
        * Pointer k is the first index of ?-part.          */
        outer:
        for (int k = less - 1; ++k <= great; ) {
            int ak = a[k];
            if (ak < pivot1) { // Move a[k] to left part
                a[k] = a[less];
                /* Here and below we use "a[i] = b; i++;" instead
                 * of "a[i++] = b;" due to performance issue.  */
                a[less] = ak;
                ++less;
            } else if (ak > pivot2) { // Move a[k] to right part
                while (a[great] > pivot2) {
                    if (great-- == k) {
                        break outer;
                    }
                }
                if (a[great] < pivot1) { // a[great] <= pivot2
                    a[k] = a[less];
                    a[less] = a[great];
                    ++less;
                } else { // pivot1 <= a[great] <= pivot2
                    a[k] = a[great];
                }
                /* Here and below we use "a[i] = b; i--;" instead
                 * of "a[i--] = b;" due to performance issue.     */
                a[great] = ak;
                --great;
            }
```

```
        }
        // Swap pivots into their final positions
        a[left]  = a[less  - 1]; a[less  - 1] = pivot1;
        a[right] = a[great + 1]; a[great + 1] = pivot2;
        // Sort left and right parts recursively, excluding known pivots
        sort(a, left, less - 2, leftmost);
        sort(a, great + 2, right, false);
        /* If center part is too large (comprises > 4/7 of the array),
         * swap internal pivot values to ends. */
        if (less < e1 && e5 < great) {
            /* Skip elements, which are equal to pivot values. */
            while (a[less] == pivot1) {
                ++less;
            }
            while (a[great] == pivot2) {
                --great;
            }
           /* Partitioning:
            *   left part         center part                  right part
            * +--------------------------------------------------------------+
            * | == pivot1 |  pivot1 < && < pivot2  |    ?    | == pivot2 |
            * +--------------------------------------------------------------+
            *             ^                       ^       ^
            *             |                       |       |
            *           less                      k     great
            * Invariants:
            *              all in (*,  less) == pivot1
            *     pivot1 < all in [less,  k)  < pivot2
            *              all in (great, *) == pivot2
            *
            * Pointer k is the first index of ?-part.  */
            outer:
            for (int k = less - 1; ++k <= great; ) {
                int ak = a[k];
                if (ak == pivot1) { // Move a[k] to left part
                    a[k] = a[less];
                    a[less] = ak;
                    ++less;
                } else if (ak == pivot2) { // Move a[k] to right part
                    while (a[great] == pivot2) {
                        if (great-- == k) {
                            break outer;
                        }
                    }
```

```
                if (a[great] == pivot1) { // a[great] < pivot2
                    a[k] = a[less];
                    /* Even though a[great] equals to pivot1, the
                     * assignment a[less] = pivot1 may be incorrect,
                     * if a[great] and pivot1 are floating-point zeros
                     * of different signs. Therefore in float and
                     * double sorting methods we have to use more
                     * accurate assignment a[less] = a[great].
                     */
                    a[less] = pivot1;
                    ++less;
                } else { // pivot1 < a[great] < pivot2
                    a[k] = a[great];
                }
                a[great] = ak;
                --great;
            }
        }
    }
    // Sort center part recursively
    sort(a, less, great, false);
} else { // Partitioning with one pivot
    /* Use the third of the five sorted elements as pivot.
     * This value is inexpensive approximation of the median. */
    int pivot = a[e3];
    /* Partitioning degenerates to the traditional 3-way
     * (or "Dutch National Flag") schema:
     *
     *   left part    center part             right part
     * +-------------------------------------------------+
     * |  < pivot  |   == pivot   |     ?    |  > pivot  |
     * +-------------------------------------------------+
     *             ^              ^        ^
     *             |              |        |
     *           less             k      great
     * Invariants:
     *   all in (left, less)   < pivot
     *   all in [less, k)     == pivot
     *   all in (great, right) > pivot
     * Pointer k is the first index of ?-part.  */
    for (int k = less; k <= great; ++k) {
        if (a[k] == pivot) {
            continue;
        }
```

```
            int ak = a[k];
            if (ak < pivot) { // Move a[k] to left part
                a[k] = a[less];
                a[less] = ak;
                ++less;
            } else { // a[k] > pivot - Move a[k] to right part
                while (a[great] > pivot) {
                    --great;
                }
                if (a[great] < pivot) { // a[great] <= pivot
                    a[k] = a[less];
                    a[less] = a[great];
                    ++less;
                } else { // a[great] == pivot
                    /* Even though a[great] equals to pivot, the
                     * assignment a[k] = pivot may be incorrect,
                     * if a[great] and pivot are floating-point
                     * zeros of different signs. Therefore in float
                     * and double sorting methods we have to use
                     * more accurate assignment a[k] = a[great].  */
                    a[k] = pivot;
                }
                a[great] = ak;
                --great;
            }
        }
        /* Sort left and right parts recursively.
         * All elements from center part are equal
         * and, therefore, already sorted. */
        sort(a, left, less - 1, leftmost);
        sort(a, great + 1, right, false);
    }
}
```

（4）二分法查找

使用二分法查找是有前提的，二分法查找的数组必须经过排序（顺序必须是升序）。

```
public static int binarySearch(int[] a, int key) {
    return binarySearch0(a, 0, a.length, key);
}
```

```
private static int binarySearch0(int[] a, int fromIndex, int toIndex,
    int key) {
    int low = fromIndex;
    int high = toIndex - 1;
    while (low <= high) {
```

```
        int mid = (low + high) >>> 1;  //mid=(low + high)/2
        int midVal = a[mid];
        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

（5）比较两数组是否相等

```
public static boolean equals(Object[] a, Object[] a2) {
    if (a==a2)   //如果两数组引用相同，说明引用对应的实体也相同，数组相同
        return true;
    //如果两数组引用有一个为空，那么比较就失去了意义，
    //两者都为空，比较两个空值是否相等没有意义，
    //要么比较结果一定为假，因为空引用没有实体，
    //一个有实体的数组和空引用没有可比性。
    //还有一点要注意：引用一个没有实体的数组会引发空指针异常
    if (a==null || a2==null)
        return false;
    //排除了以上两种情况后，接下来就是比较两者的长度
    int length = a.length;
    //如果两数组长度不等，就没必要继续比下去了，直接可以得到两者不相等的结论
    if (a2.length != length)
        return false;
    //如果上述情况都不存在，就只好逐一比较两数组同一位置每个元素了
    for (int i=0; i<length; i++) {
        Object o1 = a[i];
        Object o2 = a2[i];
        //元素比较时调用了该元素类中定义的equals方法
        //如果元素类中没有覆写equals方法，则默认调用Object类的equals方法
        //Object类的equals方法比较的是两对象的引用，不比较对象的属性
        if (!(o1==null ? o2==null : o1.equals(o2)))
            return false;  //如果某一位置元素不等，可直接得出结论
    }
    return true;  //若顺利通过比较，则可以得到两者相等的结论
}
```

（6）数组填充

将数组中所有位置填充成 val 值

```
public static void fill(int[] a, int val) {
    for (int i = 0, len = a.length; i < len; i++)
        a[i] = val;
}
```

（7）toString( )方法

对象调用 toString 相当于打印自身的信息。Arrays 类调用该方法会打印数组的全部元素。

```
public static String toString(long[] a) {  //其他类型的数组方法相同
    if (a == null)  //没有实体的数组
        return "null";
    int iMax = a.length - 1;  //数组最大下标
    if (iMax == -1) //不含任何元素的的数组
        return "[]";
    StringBuilder b = new StringBuilder();
    //打印数组的全部元素需要做大量字符串连接操作，String类做字符串连接操作
    //性能较差，所以使用StringBuilder类，该类擅长做字符串连接操作，性能较好
    b.append('[');
    for (int i = 0; ; i++) {
        b.append(a[i]); //在字符串尾部追加每个元素
        if (i == iMax)
            return b.append(']').toString();
        b.append(", ");
    }
}
```

（8）数组拷贝

```
public static <T> T[] copyOf(T[] original, int newLength) {
    return (T[]) copyOf(original, newLength, original.getClass());
}
```

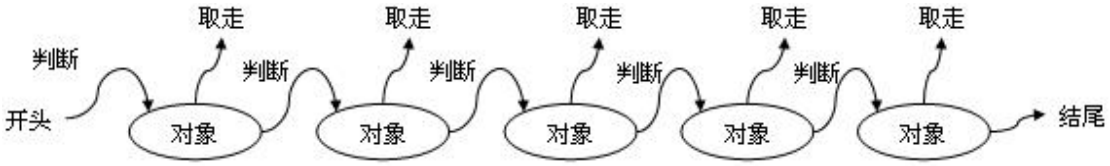
```
public static <T,U> T[] copyOf(U[] original, int newLength,
    Class<? extends T[]> newType) {
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(),
        newLength);
    System.arraycopy(original, 0, copy, 0,
        Math.min(original.length, newLength));
    //copyOf是在调用System类的arraycopy方法完成数组拷贝的
    //只是将该功能重新封装而已
    return copy;
}
```

```
public static <T> T[] copyOfRange(T[] original, int from, int to) {
    return copyOfRange(original, from, to,
          (Class<T[]>) original.getClass());
}
```

```
public static <T,U> T[] copyOfRange(U[] original, int from, int to,
    Class<? extends T[]> newType) {
    int newLength = to - from;
    if (newLength < 0)
        throw new IllegalArgumentException(from + " > " + to);
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(),
newLength);
    System.arraycopy(original, from, copy, 0,
         Math.min(original.length - from, newLength));
    return copy;
}
```

26. Iterator 接口（迭代器）

Iterator 是专门的迭代输出接口，所谓迭代输出就是将元素一个个进行判断，判断其是否有内容，如果有内容则把内容取出。只要碰到了集合输出操作，就一定使用 Iterator 接口。



（1）Iterator 接口声明

```
package java.util;
public interface Iterator<E>
```

（2）判断集合中是否还存在元素

```
boolean hasNext();
```

（3）取得当前元素

```
E next();
```

（4）移除当前元素

```
void remove();
```

27. Iterable 接口

实现该接口的集合类具备迭代输出的能力。

（1）Iterable 接口的声明

```
package java.lang;
public interface Iterable<T>
```

（2）接口中唯一的方法——iterator( )

子类通过覆写该方法获得迭代输出的能力。

```
Iterator<T> iterator();
```

28. Collection 接口

Collection 表示一组对象，这些对象也称为 collection 的元素。一些 collection 允许有重复的元素（List），而另一些则不允许（set）。一些 collection 是有序的，而另一些则是无序的。JDK 不提供此接口的任何直接 实现：它提供更具体的子接口（如 Set 和 List）实现。此接口通常用来传递 collection，并在需要最大普遍性的地方操作这些 collection。

（1）Collection 接口的定义

可以看出，该接口是 Iterable 的子接口。

```
package java.util;
public interface Collection<E> extends Iterable<E>
```

（2）获得集合中有效元素的个数

返回此 collection 中的元素数。如果此 collection 包含的元素大于 Integer.MAX_VALUE，则返回 Integer.MAX_VALUE。

```
int size();
```

（3）判断集合是否为空

如果此 collection 不包含元素，则返回 true。

```
boolean isEmpty();
```

（4）判断集合中是否包含某个元素

如果此 collection 包含指定的元素，则返回 true。

```
boolean contains(Object o);
```

（5）迭代输出的能力——实现了 Iterable 接口

由于实现了 Iterable 接口中的同名方法，Collection 接口具备了迭代输出的能力。

```
Iterator<E> iterator();
```

（6）将一个集合变为对象数组

```
Object[] toArray();
```

（7）将一个集合变为指定类型的对象数组

```
<T> T[] toArray(T[] a);
```

（8）向集合中添加元素

```
boolean add(E e);
```

（9）从集合中移除元素

```
boolean remove(Object o);
```

（10）判断一组对象是否在集合中存在

```
boolean containsAll(Collection<?> c);
```

（11）向集合中添加一组对象

```
boolean addAll(Collection<? extends E> c);
```

（12）从集合中移除一组对象

```
boolean removeAll(Collection<?> c);
```

（13）保留两集合的交集，删除其余元素

仅保留此 collection 中那些也包含在指定 collection 的元素

```
boolean retainAll(Collection<?> c);
```

（14）清空集合

```
void clear();
```

（15）对象比较——equals 方法

```
boolean equals(Object o);
```

（16）取得哈希码——hashCode 方法

```
int hashCode();
```

29. List 接口

（1）List 接口的声明

```
package java.util;
public interface List<E> extends Collection<E>
```

（2）获得集合中的元素

```
int size();
```

（3）判断集合是否为空

```
boolean isEmpty();
```

（4）判断集合中是否包含某个元素

判断是否包含某个元素时，是否包含取决于元素是否相等，也就取决于元素所在类定义的 equals 方法判断的结果。

```
boolean contains(Object o);
```

（5）迭代输出功能——通过子类实例化 Iterator

想要获得 Iterator 的实例只能通过其子类为其实例化。

```
Iterator<E> iterator();
```

（6）获得对象数组

集合的是动态的对象数组。

```
Object[] toArray();
```

（7）获得指定类型对象的数组

返回的类型由外部指定。

```
<T> T[] toArray(T[] a);
```

（8）向集合中添加一个元素

```
boolean add(E e);
```

（9）移除集合中的一个元素

```
boolean remove(Object o);
```

（10）判断集合中是否包含一组元素

```
boolean containsAll(Collection<?> c);
```

（11）向集合中添加一组元素

```
boolean addAll(Collection<? extends E> c);
```

（12）向集合指定位置添加（插入）一组元素

```
boolean addAll(int index, Collection<? extends E> c);
```

（13）移除集合中的指定元素

```
boolean removeAll(Collection<?> c);
```

（14）保留集合中的指定内容，删除其余内容

```
boolean retainAll(Collection<?> c);
```

（15）删除集合中的全部元素——清空集合

```
void clear();
```

（16）equals 方法

```
boolean equals(Object o);
```

（17）hashCode 方法

```
int hashCode();
```

（18）get 方法

这是 List 接口及其子类独有的方法，Set、Map 及其子类以及 Collection 接口中都没有此方法。

```
E get(int index);
```

（19）修改某一位置元素的值

```
E set(int index, E element);
```

（20）在某一位置插入值

```
void add(int index, E element);
```

（21）移除某一位置的元素

元素移除后，其余元素依次前移。

```
E remove(int index);
```

（22）得到某一元素在列表中的位置

```
int indexOf(Object o);
```

（23）从后向前查找指定元素的位置

```
int lastIndexOf(Object o);
```

（24）为 ListIterator 接口实例化

```
ListIterator<E> listIterator();
```

（25）截取列表

```
List<E> subList(int fromIndex, int toIndex);
```

30. AbstractCollection 类

（1）AbstractCollection 类声明

可以看出，该类是一个抽象类，并且实现了 Collection 接口，其作用是给继承它的子类提供一个模板，规定作为一个 List 应该有的功能。

```
package java.util;
public abstract class AbstractCollection<E> implements Collection<E>
```

（2）构造方法

```
protected AbstractCollection() {}
```

（3）迭代输出——为 Iterator 接口实例化

```
public abstract Iterator<E> iterator();
```

（4）获得集合中元素的个数

```
public abstract int size();
```

（5）判断集合是否为空

```
public boolean isEmpty() {
    return size() == 0;
}
```

（6）判断集合中是否包含某一元素

通过观察可以发现，决定集合中元素是否重复与否的关键是 equals 方法，这里的 equals 方法是元素所在类中的 equals 方法而不是集合中的 equals 方法。

```
public boolean contains(Object o) {
    Iterator<E> it = iterator();
    if (o==null) {
        while (it.hasNext())
            if (it.next()==null)
                return true;
```

```
    } else {
        while (it.hasNext())
            if (o.equals(it.next()))
                return true;
    }
    return false;
}
```

（9）获得对象数组

```
public Object[] toArray() {
    Object[] r = new Object[size()];  //根据集合大小初始化对象数组
    Iterator<E> it = iterator();      //获得Iterator实例，准备迭代
    for (int i = 0; i < r.length; i++) {
        if (! it.hasNext())           //迭代复制，直到集合中没有元素
            return Arrays.copyOf(r, i);
        r[i] = it.next();
    }
    return it.hasNext() ? finishToArray(r, it) : r;
}
```

（10）获得指定类型的对象数组

基本思路同上。

```
public <T> T[] toArray(T[] a) {
    int size = size();     //获得集合的大小
    //如果空间不足就通过反射机制取得新数组的实例
    T[] r = a.length >= size ? a :(T[])java.lang.reflect.Array
        .newInstance(a.getClass().getComponentType(), size);
    Iterator<E> it = iterator();   //实例化Iterator
    for (int i = 0; i < r.length; i++) {
        if (! it.hasNext()) { // 迭代器中没有元素了
            if (a == r) {  //没有开辟新空间，说明所有元素拷贝完成
                r[i] = null; // 其余元素设空值
            } else if (a.length < i) {
                return Arrays.copyOf(r, i);
            } else {
                System.arraycopy(r, 0, a, 0, i); //数组拷贝
                if (a.length > i) {
                    a[i] = null;
                }
            }
            return a;
        }
        r[i] = (T)it.next();  //迭代取出元素
```

```
    }
    return it.hasNext() ? finishToArray(r, it) : r;
}
```

```
private static <T> T[] finishToArray(T[] r, Iterator<?> it) {
    int i = r.length;
    while (it.hasNext()) {
        int cap = r.length;
        if (i == cap) {
            int newCap = cap + (cap >> 1) + 1;
            // overflow-conscious code
            if (newCap - MAX_ARRAY_SIZE > 0)
                newCap = hugeCapacity(cap + 1);
            r = Arrays.copyOf(r, newCap);
        }
        r[i++] = (T)it.next();
    }
    // trim if overallocated
    return (i == r.length) ? r : Arrays.copyOf(r, i);
}
```

（11）向集合中添加一个元素

```
public boolean add(E e) {
    throw new UnsupportedOperationException();
}
```

（12）移除集合中指定位置的元素

```
public boolean remove(Object o) {
    Iterator<E> it = iterator();
    if (o==null) {
        while (it.hasNext()) {
            if (it.next()==null) {
                it.remove();
                return true;
            }
        }
    } else {
        while (it.hasNext()) {
            if (o.equals(it.next())) {
                it.remove();
                return true;
            }
        }
    }
```

```
    return false;
}
```

（13）判断集合中是否包含一组元素

```
public boolean containsAll(Collection<?> c) {
    for (Object e : c)
        if (!contains(e))
            return false;
    return true;
}
```

（14）向集合中添加一组元素

```
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

（15）删除集合中指定的一组元素

```
public boolean removeAll(Collection<?> c) {
    boolean modified = false;
    Iterator<?> it = iterator();
    while (it.hasNext()) {
        if (c.contains(it.next())) {
            it.remove();
            modified = true;
        }
    }
    return modified;
}
```

（16）保留一组指定的元素，删除其余元素

```
public boolean retainAll(Collection<?> c) {
    boolean modified = false;
    Iterator<E> it = iterator();
    while (it.hasNext()) {
        if (!c.contains(it.next())) {
            it.remove();
            modified = true;
        }
    }
```

```
    return modified;
}
```

（17）删除所有元素，清空集合

```
public void clear() {
    Iterator<E> it = iterator();
    while (it.hasNext()) {
        it.next();
        it.remove();
    }
}
```

（18）toString 方法

调用该方法可以打印集合中所有的元素的相关信息。如果元素所在类未覆写 toString 方法，则会调用 Object 类继承来的 toString 方法，打印对象所在类的类名及其哈希值。

```
public String toString() {
    Iterator<E> it = iterator();
    if (! it.hasNext())
        return "[]";
    //打印过程中字符串连接操作较多，出于性能考虑，使用StringBuilder类
    StringBuilder sb = new StringBuilder();
    sb.append('[');
    for (;;) {
        E e = it.next();
        sb.append(e == this ? "(this Collection)" : e);
        if (! it.hasNext())
            //迭代每个元素并调用元素所在类的toString方法打印每个元素的信息
            return sb.append(']').toString();
        sb.append(',').append(' ');
    }
}
```

30. AbstractList 类

（1）AbstractList 类声明

```
package java.util;
public abstract class AbstractList<E> extends AbstractCollection<E>
      implements List<E>
```

（2）构造方法

```
protected AbstractList() {}
```

（3）向 List 中添加一个元素

```
public boolean add(E e) {
    add(size(), e);
    return true;
}
public void add(int index, E element) {
    throw new UnsupportedOperationException();  //该方法会被子类覆写
}
```

（4）get 方法——List 独有的方法

```
abstract public E get(int index);
```

（5）修改集合中某一位置的元素

```
public E set(int index, E element) {
    throw new UnsupportedOperationException();  //该方法会被子类覆写
}
```

（6）移除指定位置的元素

```
public E remove(int index) {
    throw new UnsupportedOperationException();  //该方法会被子类覆写
}
```

（7）查找指定元素的位置

如果找到会返回索引，找不到返回-1

```
public int indexOf(Object o) {
    ListIterator<E> it = listIterator();
    if (o==null) {
        while (it.hasNext())
            if (it.next()==null)
                return it.previousIndex();
```

```
    } else {
        while (it.hasNext())
            if (o.equals(it.next()))
                return it.previousIndex();
    }
    return -1;
}
```

（8）从后向前查找指定元素的位置

如果找到会返回索引，找不到返回-1

```
public int lastIndexOf(Object o) {
    ListIterator<E> it = listIterator(size());
    if (o==null) {
        while (it.hasPrevious())
            if (it.previous()==null)
                return it.nextIndex();
    } else {
        while (it.hasPrevious())
            if (o.equals(it.previous()))
                return it.nextIndex();
    }
    return -1;
}
```

（9）清空 List

```
public void clear() {
    removeRange(0, size());
}
```

```
protected void removeRange(int fromIndex, int toIndex) {
    ListIterator<E> it = listIterator(fromIndex);
    for (int i=0, n=toIndex-fromIndex; i<n; i++) {
        it.next();
        it.remove();
    }
}
```

（10）向集合中添加一组元素

```
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);
    boolean modified = false;
    for (E e : c) {
        add(index++, e);
        modified = true;
```

```
    }
    return modified;
}
```

```
public void add(int index, E element) {
    throw new UnsupportedOperationException();
}
```

（11）实例化 Iterator 接口

```
public Iterator<E> iterator() {
    return new Itr();
}
```

（12）实例化 ListIterator 接口

```
public ListIterator<E> listIterator() {
    return listIterator(0);
}
```

```
public ListIterator<E> listIterator(final int index) {
    rangeCheckForAdd(index);
    return new ListItr(index);
}
```

（13）截取列表的一部分

```
public List<E> subList(int fromIndex, int toIndex) {
    return (this instanceof RandomAccess ?
            new RandomAccessSubList<>(this, fromIndex, toIndex) :
            new SubList<>(this, fromIndex, toIndex));
}
```

（14）equals 方法

```
public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof List))
            return false;
        ListIterator<E> e1 = listIterator();
        ListIterator e2 = ((List) o).listIterator();
        while (e1.hasNext() && e2.hasNext()) {
            E o1 = e1.next();
            Object o2 = e2.next();
            if (!(o1==null ? o2==null : o1.equals(o2)))
                return false;
        }
        return !(e1.hasNext() || e2.hasNext());
```

```
}
```

（15）hashCode 方法

```
public int hashCode() {
    int hashCode = 1;
    for (E e : this)
        hashCode = 31*hashCode + (e==null ? 0 : e.hashCode());
    return hashCode;
}
```

31. ArrayList 类 （数组列表）

（1）ArrayList 声明

```
package java.util;
public class ArrayList<E> extends AbstractList<E>
     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

（2）ArrayList 底层是对象数组

```
private transient Object[] elementData;
```

（3）有效元素个数—size

```
private int size;
```

（4）构造方法

```
private static final Object[] EMPTY_ELEMENTDATA = {}; //空对象数组
public ArrayList() {
    super();
    this.elementData = EMPTY_ELEMENTDATA;  //初始化为空对象数组
}
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException(
              "Illegal Capacity: "+initialCapacity);
    //初始化指定大小的对象数组
    this.elementData = new Object[initialCapacity];
}
```

（5）扩容方法

```
private void grow(int minCapacity) {
    // overflow-conscious code
```

```
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)     //小于最小容量
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)  //大于最大容量
        newCapacity = hugeCapacity(minCapacity);
    //在确认newCapacity没有问题后开辟相应空间并复制旧数组内容到新数组
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

```
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
// Integer.MAX_VALUE - 8 = 2147483647 - 8 = 2147483639
```

（6）获得元素个数

该方法常常用于作为循环次数的控制条件。

```
public int size() {
    return size;
}
```

（7）判断集合是否为空

```
public boolean isEmpty() {
    return size == 0;
}
```

（8）判断集合中是否包含某一元素

如果此列表中包含指定的元素，则返回 true。

```
public boolean contains(Object o) {
    return indexOf(o) >= 0;  //只要不返回-1，就表示找到了该元素
}
```

（9）查找某一元素在集合中的位置

返回此列表中首次出现的指定元素的索引，或如果此列表不包含元素，则返回 -1。

```
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

（10）从后向前查找元素第一次出现的位置

返回此列表中最后一次出现的指定元素的索引，或如果此列表不包含索引，则返回 -1。

```
public int lastIndexOf(Object o) {
    if (o == null) {
        for (int i = size-1; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = size-1; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

（11）对象克隆

由于实现了 Cloneable 接口，所以要覆写 clone 方法。

```
public Object clone() {
    try {
        @SuppressWarnings("unchecked")
            ArrayList<E> v = (ArrayList<E>) super.clone();
        v.elementData = Arrays.copyOf(elementData, size);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError();
    }
}
```

（12）得到底层的对象数组

```
public Object[] toArray() {
    return Arrays.copyOf(elementData, size);  //数组复制
}
```

（13）得到指定类型的对象数组

对象数组的类型由外部指定。

```
@SuppressWarnings("unchecked")
public <T> T[] toArray(T[] a) {
    if (a.length < size)
         return (T[]) Arrays.copyOf(elementData, size, a.getClass());
    System.arraycopy(elementData, 0, a, 0, size); //数组拷贝
    if (a.length > size)
```

```
        a[size] = null;
    return a;
}
```

（14）get 方法—获取某一位置的元素

```
public E get(int index) {
    rangeCheck(index);
    return elementData(index);
}
```

（15）修改某一位置元素的值

```
public E set(int index, E element) {
    rangeCheck(index);
    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
```

（16）向集合中添加元素的值

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

```
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    ensureExplicitCapacity(minCapacity);
}
```

```
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

（17）在列表指定位置插入元素

```
public void add(int index, E element) {
    rangeCheckForAdd(index);
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
          size - index);
```

```
    elementData[index] = element;
    size++;
}
```

（18）索引检测——判断是否越界

可以看出，这里抛出的都是索引越界异常（IndexOutOfBoundsException）。

```
private void rangeCheck(int index) {
    if (index >= size)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

```
private void rangeCheckForAdd(int index) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

```
private String outOfBoundsMsg(int index) {
    return "Index: "+index+", Size: "+size;
}
```

（19）从列表中移除一组元素

```
public boolean removeAll(Collection<?> c) {
    return batchRemove(c, false);  //调用批量移除方法
}
```

```
private boolean batchRemove(Collection<?> c, boolean complement) {
    final Object[] elementData = this.elementData;
    int r = 0, w = 0;
    //r是循环控制变量，最大值为底层存储对象的elementData数组长度-1
    //w表示删除完元素后列表中的有效元素个数
    boolean modified = false;
    try {
        //对原列表的每个元素，逐一判断是否在c中，若不在，则放到列表中
        for (; r < size; r++)
            if (c.contains(elementData[r]) == complement) //不在c中
                //元素不再c中，就会被保留在原集合中，列表有效元素个数w + 1
                elementData[w++] = elementData[r];
    } finally {
        if (r != size) {
            System.arraycopy(elementData, r,elementData, w,size - r);
            w += size - r;
        }
        if (w != size) {
            // 告诉垃圾回收器，可以回收这些内存了
            for (int i = w; i < size; i++)
                elementData[i] = null;
```

```
            modCount += size - w;
            size = w;
            modified = true;
        }
    }
    return modified;
}
```

（20）迭代输出——实例化 Iterator 接口

```
public Iterator<E> iterator() {
    return new Itr(); //Itr是ArrrayList的私有内部类
}
```

由于篇幅限制，内部类只给出方法声明，不再给出方法实现

```
private class Itr implements Iterator<E> {
    public boolean hasNext() {... }
    public E next() {...}
    public void remove() {... }
    final void checkForComodification() {...}
}
```

（21）截取列表

```
public List<E> subList(int fromIndex, int toIndex) {
    subListRangeCheck(fromIndex, toIndex, size);
    return new SubList(this, 0, fromIndex, toIndex);
    //SubList是ArrayList的私有内部类
}
```

```
static void subListRangeCheck(int fromIndex, int toIndex, int size) {
    if (fromIndex < 0)
        throw new IndexOutOfBoundsException("fromIndex = "+fromIndex);
    if (toIndex > size)
        throw new IndexOutOfBoundsException("toIndex = " + toIndex);
    if (fromIndex > toIndex)
        throw new IllegalArgumentException("fromIndex(" + fromIndex
             +") > toIndex(" + toIndex + ")");
}
```

由于篇幅限制，内部类只给出方法声明，不再给出方法实现

```
private class SubList extends AbstractList<E>
    implements RandomAccess{
    int size;
    SubList(AbstractList<E> parent,
        int offset, int fromIndex, int toIndex) {...}
    public E set(int index, E e) {...}
    public E get(int index) {...}
```

```
    public int size() {...}
    public void add(int index, E e) {...}
    public E remove(int index) {...}
    protected void removeRange(int fromIndex, int toIndex) {...}
    public boolean addAll(Collection<? extends E> c) {...}
    public boolean addAll(int index, Collection<? extends E> c) {...}
    public Iterator<E> iterator() {...}
    public ListIterator<E> listIterator(final int index) {...}
    public List<E> subList(int fromIndex, int toIndex) {...}
    private void rangeCheck(int index) {...}
    private void rangeCheckForAdd(int index) {...}
    private String outOfBoundsMsg(int index) {...}
    private void checkForComodification() {...}
}
```

32. Vector 类

该类是一个元老级的类，从 JDK1.0 就开始有了，JDK1.2 后重点强调了集合框架的概念，考虑到很多人的使用习惯，Java 设计者让 Vector 类多实现了一个 List 接口，Vector 类才得以被保存下来。这个类是一个旧类，使用不是很多了。

（1）Vector 类声明

```
package java.util;
public class Vector<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

（2）Vector 类底层是对象数组

```
protected Object[] elementData;
```

（3）有效元素个数

```
protected int elementCount;
```

（4）扩容的增量

```
protected int capacityIncrement;
```

（5）构造方法

```
public Vector(int initialCapacity, int capacityIncrement) {...}
public Vector(int initialCapacity) {...}
public Vector() {...}
public Vector(Collection<? extends E> c) {...}
```

（6）数组的最大容纳量

```
private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
// Integer.MAX_VALUE - 8 = 2147483647 - 8 = 2147483639
```

（7）获得有效元素的个数

从这里看出 Vector 的很多方法都用 synchornized 声明，说明该类采用同步的操作方式，是线程安全的。

```
public synchronized int size() {
    return elementCount;
}
```

（8）判断集合是否为空

```
public synchronized boolean isEmpty() {
    return elementCount == 0;
}
```

（9）废弃的接口 Enumeration

Enumeration 是一种古老的集合输出方式，现在已经很少用了，基本都用 Iterator。

```
public Enumeration<E> elements() {
    return new Enumeration<E>() {  //匿名内部类
        public boolean hasMoreElements() {...}
        public E nextElement() {synchronized (Vector.this) {...}...}
    };
}
```

```
package java.util;
/** 省略中间部分注释
 * @since   JDK1.0  该接口有着古老的历史
 */
public interface Enumeration<E> {
    boolean hasMoreElements();
    E nextElement();
}
```

（10）判断集合中是否包含某个元素

```
public boolean contains(Object o) {
    return indexOf(o, 0) >= 0;
}
```

（11）查找某个元素在列表中的位置

```
public int indexOf(Object o) {
    return indexOf(o, 0);
}
```

```
public synchronized int indexOf(Object o, int index) {
    if (o == null) {
        for (int i = index ; i < elementCount ; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = index ; i < elementCount ; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

（12）从后向前查找某一元素在列表中的位置

```
public synchronized int lastIndexOf(Object o) {
    return lastIndexOf(o, elementCount-1);
}
```

```
public synchronized int lastIndexOf(Object o, int index) {
    if (index >= elementCount)
        throw new IndexOutOfBoundsException(
              index+" >= " + elementCount);
    if (o == null) {
        for (int i = index; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = index; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

其余的方法基本上和 ArrayList 差不多，就不再一一列举了。

33. Queue 接口

Queue 接口指明了队列操作应该有的方法，实现该接口的类可以进行队列操作。

（1）Queue 接口的声明

```
package java.util;
```

```
public interface Queue<E> extends Collection<E>
```

（2）入队操作—队列上溢会抛出异常而不会改变队长

上溢：队列空间已满，而继续往队列中插入元素，就会使数组越界而导致程序代码被破坏，称为“上溢”。

执行该方法出现上述情况出现后，将不会增加队长而是抛出异常提示用户队列上溢。

参数： e - 要添加的元素

返回： true（根据 Collection.add(E) 的规定）

抛出：IllegalStateException - 如果由于容量的限制此时不能添加该元素

ClassCastException - 如果指定元素的类不允许将其添加到此队列

NullPointerException - 如果指定元素为 null 并且此队列不允许 null 元素

IllegalArgumentException - 如果此元素的某些属性不允许将其添加到此队列

```
boolean add(E e);
```

（3）入队操作—队列上溢会增加队长以插入元素

执行该方法出现上述情况出现后，将会增加队长以执行插入操作。

参数： e - 要添加的元素

返回： 如果该元素已添加到此队列，则返回 true；否则返回 false

抛出： ClassCastException - 如果指定元素的类不允许将其添加到此队列

NullPointerException - 如果指定元素为 null 并且此队列不允许 null 元素

IllegalArgumentException - 如果此元素的某些属性不允许将其添加到此队列

```
boolean offer(E e);
```

（4）出队操作—获取并移除队头，发生下溢抛出异常

返回：队列的头

抛出：NoSuchElementException - 如果此队列为空

```
E remove();
```

（5）出队操作—获取并移除队头，发生下溢返回 null

获取并移除此队列的头，如果此队列为空，则返回 null。

返回：队列的头，如果此队列为空，则返回 null

```
E poll();
```

（6）获取但不移除队头—队列为空抛出异常

获取但不移除此队列的头，队列为空时将抛出一个异常。

返回：队列的头

抛出： NoSuchElementException - 如果此队列为空

```
E element();
```

（7）获取但不移除队头—队列为空返回 null

获取但不移除此队列的头；如果此队列为空，则返回 null。

```
E peek();
```

34. LinkedList 类 （链表|链队列）

（1）LinkedList 类声明

LinkedList 实现了数据结构中的单链表操作，除了实现 List 接口、Cloneable 和 Serializable 接口外，还实现了 Deque 接口，而 Deque 接口又是 Queue 接口的子接口，所以 LinkedList 不仅仅是线性单链表，还是一个链队列，可以实现队列的各种操作。

```
package java.util;
public class LinkedList<E> extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
public interface Deque<E> extends Queue<E>
```

（2）链表的长度—链表中元素的个数

```
transient int size = 0;
```

（3）链表的头结点和尾结点——对应队列中的队头和队尾

```
transient Node<E> first;
transient Node<E> last;
private static class Node<E> {
    E item;         //元素
    Node<E> next;  //后继结点的引用
    Node<E> prev;  //前驱结点的引用
    Node(Node<E> prev, E element, Node<E> next) {  //构造方法
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

（4）构造方法

```
public LinkedList() {}
public LinkedList(Collection<? extends E> c) {
    this();
    addAll(c);
}
```

（5）获取链表中的第一个元素

```
public E getFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return f.item;
```

```
}
```

（6）获得链表中的最后一个元素

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

（7）获取并移除链表的头

```
public E removeFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return unlinkFirst(f);
}
```

（8）得到表中的最后一个元素并移除

```
public E removeLast() {
    final Node<E> l = last;
    if (l == null)
        throw new NoSuchElementException();
    return unlinkLast(l);
}
```

（9）在表头插入元素

```
public void addFirst(E e) {
    linkFirst(e);
}
```

```
private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
    size++;
    modCount++;
}
```

（10）在表尾插入元素

```
public void addLast(E e) {
    linkLast(e);
}
```

```
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

（11）判断单链表中是否包含某一元素

```
public boolean contains(Object o) {
    return indexOf(o) != -1;
}
```

（12）获取某一元素在链表中的位置

```
public int indexOf(Object o) {
    int index = 0;
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null)
                return index;
            index++;
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
    }
    return -1;  //未找到该元素
}
```

（13）获取单链表中有效元素的个数

```
public int size() {
```

```
    return size;
}
```

（14）向链表中增加一个元素——默认添加到表尾

```
public boolean add(E e) {
    linkLast(e);
    return true;
}
```

（15）获取并移除表头

```
public boolean remove(Object o) {
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x);
                return true;
            }
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item)) {
                unlink(x);
                return true;
            }
        }
    }
    return false;
}
```

```
E unlink(Node<E> x) {
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;
    if (prev == null) {
        first = next;
    } else {
        prev.next = next;
        x.prev = null;
    }
    if (next == null) {
        last = prev;
    } else {
        next.prev = prev;
        x.next = null;
```

```
    }
    x.item = null;
    size--;
    modCount++;
    return element;
}
```

（16）在单链表尾部添加一组元素

```
public boolean addAll(Collection<? extends E> c) {
    return addAll(size, c);
}
```

```
public boolean addAll(int index, Collection<? extends E> c) {
    checkPositionIndex(index);
    Object[] a = c.toArray();
    int numNew = a.length;
    if (numNew == 0)
        return false;
    Node<E> pred, succ;
    if (index == size) {
        succ = null;
        pred = last;
    } else {
        succ = node(index);
        pred = succ.prev;
    }
    for (Object o : a) {
        @SuppressWarnings("unchecked") E e = (E) o;
        Node<E> newNode = new Node<>(pred, e, null);
        if (pred == null)
            first = newNode;
        else
            pred.next = newNode;
        pred = newNode;
    }
    if (succ == null) {
        last = pred;
    } else {
        pred.next = succ;
        succ.prev = pred;
    }
    size += numNew;
    modCount++;
    return true;
}
```

（17）删除单链表中的所有元素——清空链表

```
public void clear() {
    for (Node<E> x = first; x != null; ) {  //从前到后对每个结点进行操作
        Node<E> next = x.next;  //获取后继结点
        x.item = null;  //让元素断开引用，成为垃圾空间，等待被回收
        x.next = null;  //断开与后继结点的连接
        x.prev = null;  //断开与前驱结点的连接
        x = next;
    }
    first = last = null;
    size = 0;
    modCount++;
}
```

（18）获取单链表某一位置的元素

```
public E get(int index) {
    checkElementIndex(index);  //检查索引index是否合法
    return node(index).item;
}
```

```
private void checkElementIndex(int index) {
    if (!isElementIndex(index))  //如果索引不合法，则抛出异常
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

```
private String outOfBoundsMsg(int index) {
    return "Index: "+index+", Size: "+size;
}
```

```
private boolean isElementIndex(int index) {
    return index >= 0 && index < size;  //合法index的取值是[0,size)
}
```

```
Node<E> node(int index) {  //根据索引找到结点
    if (index < (size >> 1)) {  //若index小于元素个数的一半，正向查找
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {  //若index大于元素个数的一半，反向查找 这样有利于提高查找效率
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

（19）更新某一位置元素的值

该方法会返回修改前的值。

```
public E set(int index, E element) {
    checkElementIndex(index);
    Node<E> x = node(index);
    E oldVal = x.item;
    x.item = element;
    return oldVal;
}
```

（20）在单链表的某一位置插入值

```
public void add(int index, E element) {
    checkPositionIndex(index);
    if (index == size)
        linkLast(element);  //链接到表尾
    else
        linkBefore(element, node(index));
}
```

```
private void checkPositionIndex(int index) {
    if (!isPositionIndex(index))
        throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}
```

```
private boolean isPositionIndex(int index) {
    return index >= 0 && index <= size; //合法index的取值是[0,size]
}
```

```
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

```
void linkBefore(E e, Node<E> succ) {
    final Node<E> pred = succ.prev;
    final Node<E> newNode = new Node<>(pred, e, succ);
    succ.prev = newNode;
    if (pred == null)
        first = newNode;
    else
```

```
        pred.next = newNode;
    size++;
    modCount++;
}
```

（21）获取并移除链表的头

```
public E remove(int index) {
    checkElementIndex(index);
    return unlink(node(index));
}
```

（22）从后向前查找某一元素在链表中的位置

```
public int lastIndexOf(Object o) {
    int index = size;
    if (o == null) {
        for (Node<E> x = last; x != null; x = x.prev) {
            index--;
            if (x.item == null)
                return index;
        }
    } else {
        for (Node<E> x = last; x != null; x = x.prev) {
            index--;
            if (o.equals(x.item))
                return index;
        }
    }
    return -1;
}
```

（23）获取链表的第一个元素（不抛异常）

该方法不会抛出异常，但第一个元素为空时返回 null。

```
public E peek() {
    final Node<E> f = first;
    return (f == null) ? null : f.item;
}
```

（24）获取链表的第一个元素（抛异常）

第一个元素为空时抛出异常。

```
public E element() {
    return getFirst();
}
```

（25）获取链表的第一个元素并移除（不抛异常）

```
public E poll() {
    final Node<E> f = first;
    return (f == null) ? null : unlinkFirst(f);
}
```

```
private E unlinkFirst(Node<E> f) {
    final E element = f.item;
    final Node<E> next = f.next;
    f.item = null;
    f.next = null; // 断开与后继结点的连接，以便垃圾回收器能够回收
    first = next;
    if (next == null)  //链表中只有一个元素
        last = null;
    else
        next.prev = null;
    size--;
    modCount++;
    return element;
}
```

（26）移除并获取链表的头（抛异常）

```
public E remove() {
    return removeFirst();
}
```

（27）在链表尾部添加元素

```
public boolean offer(E e) {
    return add(e);
}
```

（28）压栈方法

```
public void push(E e) {
    addFirst(e);
}
```

（29）弹栈方法

```
public E pop() {
    return removeFirst();
}
```

（30）获得对象数组

```
public Object[] toArray() {
```

```
    Object[] result = new Object[size];
    int i = 0;
    for (Node<E> x = first; x != null; x = x.next)
        result[i++] = x.item;
    return result;
}
```

（31）获得指定类型的对象数组

```
@SuppressWarnings("unchecked")
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        a = (T[])java.lang.reflect.Array.newInstance(
                    a.getClass().getComponentType(), size);
    int i = 0;
    Object[] result = a;
    for (Node<E> x = first; x != null; x = x.next)
        result[i++] = x.item;
    if (a.length > size)
        a[size] = null;
    return a;
}
```

35. Set 接口

Set 接口也是 Collection 接口的子接口，但是与 Collection 或 List 接口不同的是，Set 接口中不能加入重复的元素，其主要方法与 Collection 是一致的。Set 接口的实例无法像 List 接口那样进行双向输出。

（1）Set 接口的定义

```
package java.util;
public interface Set<E> extends Collection<E>
```

（2）返回 Set 中的元素数量

```
int size();
```

（3）判断 Set 是否为空

```
boolean isEmpty();
```

（4）判断 Set 中是否包含某一元素

如果 set 包含指定的元素，则返回 true。

```
boolean contains(Object o);
```

（5）实例化 Iterator 接口

```
Iterator<E> iterator();
```

（6）获得对象数组

```
Object[] toArray();
```

（7）获得指定类型的对象数组

```
<T> T[] toArray(T[] a);
```

（8）向 Set 中添加一个元素

```
boolean add(E e);
```

（9）从 Set 中移除一个元素

```
boolean remove(Object o);
```

（10）判断 Set 中是否包含一组元素

如果此 set 包含指定 collection 的所有元素，则返回 true。

```
boolean containsAll(Collection<?> c);
```

（11）向 Set 中添加一组元素

如果 set 中没有指定 collection 中的所有元素，则将其添加到此 set 中（可选操作）。

```
boolean addAll(Collection<? extends E> c);
```

（12）保留 Set 中的指定元素，并删除其余元素

```
boolean retainAll(Collection<?> c);
```

（13）从 Set 中移除一组元素

```
boolean removeAll(Collection<?> c);
```

（14）删除 Set 中的全部元素——清空 Set

```
void clear();
```

（15）equals 方法

比较指定对象与此 set 的相等性。

```
boolean equals(Object o);
```

（16）hashCode 方法

返回 set 的哈希码值。

```
int hashCode();
```

36. HashSet 类 （散列存放）

HashSet 是 Set 接口的一个子类，主要的特点是：里面不能存放重复元素，而且是采用散列的存储方式，所以是没有顺序的。

（1）HashSet 类的声明

```
package java.util;
public class HashSet<E> extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
```

（2）HashSet 的底层是 HashMap

```
private transient HashMap<E,Object> map;
```

（3）HashSet 的构造方法

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

（4）迭代输出——Iterator 接口的实例化

```
public Iterator<E> iterator() {
    return map.keySet().iterator();
}
```

（5）获取有效元素的个数

```
public int size() {
    return map.size();
}
```

（6）判断 HashSet 是否为空

```
public boolean isEmpty() {
    return map.isEmpty();
}
```

（7）判断 HashSet 中是否包含某个元素

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

（8）向 HashSet 中添加元素

```
private static final Object PRESENT = new Object();
```

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

（9）从 HashSet 中移除一个元素

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

（10）清空 HashSet

```
public void clear() {
    map.clear();
}
```

（11）clone 方法

```
public Object clone() {
    try {
        HashSet<E> newSet = (HashSet<E>) super.clone();
        newSet.map = (HashMap<E, Object>) map.clone();
        return newSet;
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}
```

37. TreeSet 类

如果想对输入的数据进行有序排列，则要使用 TreeSet 类。

（1）TreeSet 类的声明

```
package java.util;
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
public interface NavigableSet<E> extends SortedSet<E>
public interface SortedSet<E> extends Set<E>
```

（2）TreeSet 的底层是 TreeMap

```
private transient NavigableMap<E,Object> m;
```

（3）构造方法

```
public TreeSet() {
```

```
    this(new TreeMap<E,Object>());
}
```

（4）迭代输出——为 Iterator 接口实例化

```
public Iterator<E> iterator() {
    return m.navigableKeySet().iterator();
}
```

（5）获得 TreeSet 中的元素个数

```
public int size() {
    return m.size();
}
```

（6）判断 TreeSet 是否为空

```
public boolean isEmpty() {
    return m.isEmpty();
}
```

（7）判断 TreeSet 中是否包含某个元素

```
public boolean contains(Object o) {
    return m.containsKey(o);
}
```

（8）向 TreeSet 中添加一个元素

```
public boolean add(E e) {
    return m.put(e, PRESENT)==null;
}
```

（9）从 TreeSet 中移除一个元素

```
public boolean remove(Object o) {
    return m.remove(o)==PRESENT;
}
```

（10）清空 TreeSet

```
public void clear() {
    m.clear();
}
```

（11）向 TreeSet 中添加一组元素

```
public boolean addAll(Collection<? extends E> c) {
    if (m.size()==0 && c.size() > 0 && c instanceof SortedSet
```

```
          && m instanceof TreeMap) {
        SortedSet<? extends E> set = (SortedSet<? extends E>) c;
        TreeMap<E,Object> map = (TreeMap<E, Object>) m;
        Comparator<? super E> cc =
             (Comparator<? super E>) set.comparator();
        Comparator<? super E> mc = map.comparator();
        if (cc==mc || (cc != null && cc.equals(mc))) {
            map.addAllForTreeSet(set, PRESENT);
            return true;
        }
    }
    return super.addAll(c);
}
```

（12）获得 Set 中的第一个元素

```
public E first() {
    return m.firstKey();
}
```

（13）获得 Set 中的最后一个元素

```
public E last() {
    return m.lastKey();
}
```

38. SortedSet 接口

从 TreeSet 类的定义中可以发现，TreeSet 里实现了 SortedSet 接口，此接口主要是用于排序操作的，即：实现此接口的子类都属于排序的子类。

（1）SortedSet 接口定义

```
package java.util;
public interface SortedSet<E> extends Set<E>
```

（2）排序操作依赖比较器

```
Comparator<? super E> comparator();
```

（3）获取指定元素之间的对象

```
SortedSet<E> subSet(E fromElement, E toElement);
```

（4）返回从开始到指定元素的集合

```
SortedSet<E> headSet(E toElement);
```

（5）返回从指定元素到最后一个元素的集合

```
SortedSet<E> tailSet(E fromElement);
```

（6）返回集合的第一个元素

```
E first();
```

（7）返回集合的最后一个元素

```
E last();
```

39. ListIterator 接口

ListIterator 接口是专门用于输出 List 内容的。

（1）ListIterator 接口定义

```
package java.util;
public interface ListIterator<E> extends Iterator<E>
```

（2）判断是否有下一个值

以正向遍历列表时，如果列表迭代器有多个元素，则返回 true（换句话说，如果 next 返回一个元素而不是抛出异常，则返回 true）。

```
boolean hasNext();
```

（3）获取当前元素

```
E next();
```

（4）判断是否还有前一个值（前驱）

```
boolean hasPrevious();
```

（5）获得前一个值（前驱）

```
E previous();
```

（6）下一个值的索引

返回对 next 的后续调用所返回元素的索引。

```
int nextIndex();
```

（7）上一个值的索引

返回对 previous 的后续调用所返回元素的索引。

```
int previousIndex();
```

（8）移除元素

从列表中移除由 next 或 previous 返回的最后一个元素（可选操作）。

```
void remove();
```

（9）修改某一位置元素的值

用指定元素替换 next 或 previous 返回的最后一个元素（可选操作）。

```
void set(E e);
```

（10）向集合中添加一个元素

将指定的元素插入列表（可选操作）。

```
void add(E e);
```

40. Map 接口

Collection、Set、List 接口都属于单值的操作，即：每次只能操作一个对象，而 Map 与它们不同的是，每次操作的是一对对象，即二元偶对象，Map 中的每个元素都使用 key → value 的形式存储在集合之中

（1）Map 接口的声明

```
package java.util;
public interface Map<K,V>
```

（2）获得 key->value 对数

```
int size();
```

（3）判断 Map 中是否为空

```
boolean isEmpty();
```

（4）判断指定的 key 是否存在

```
boolean containsKey(Object key);
```

（5）判断指定的 value 是否存在

```
boolean containsValue(Object value);
```

（6）根据 key 取得 value

```
V get(Object key);
```

（7）向 Map 中加入 key->value 对

```
V put(K key, V value);
```

（8）根据 key 删除 value

```
V remove(Object key);
```

（9）将一个 Map 加入另一个 Map

```
void putAll(Map<? extends K, ? extends V> m);
```

（10）清空 Map 集合

```
void clear();
```

（11）取得全部的 value

```
Collection<V> values();
```

（12）将 Map 变为 Set

```
Set<Map.Entry<K, V>> entrySet();
```

（13）equals 方法

```
boolean equals(Object o);
```

（14）hashCode 方法

```
int hashCode();
```

（15）内部接口 Entry

Map.Entry 接口是 Map 内部定义的一个接口，专门用来保存 key->value 的内容。

```
interface Entry<K,V> {
    K getKey();
    V getValue();
    V setValue(V value);
    boolean equals(Object o);
    int hashCode();
}
```

41. HashMap 类

HashMap 是典型的“数组+链表”的数据结构。

（1）HashMap 类的声明

```
package java.util;
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
```

（2）HashMap 的构造方法

```
public HashMap() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
}
```

```
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}
```

```
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)    //初始化空间大小<0
        throw new IllegalArgumentException(
             "Illegal initial capacity: " + initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException(
             "Illegal load factor: " + loadFactor);
    this.loadFactor = loadFactor;
    threshold = initialCapacity;
    init();
}
```

```
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;   // aka 16
//默认初始化大小 永远是 2 的次方数
```

（3）成员变量 size——保存 HashMap 有效 key->value 对的个数

```
transient int size;
```

（4）HashMap 底层是一个数组

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

```
static final Entry<?,?>[] EMPTY_TABLE = {};
```

（5）HashMap 的内部类——Entry——数组+链表结构的由来

从 Entry 的四个成员中可以看出，Entry 实际上是典型的链表结构。

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;   //链接后继结点的引用
    int hash;

    Entry(int h, K k, V v, Entry<K,V> n) {
        value = v;
        next = n;
```

```
        key = k;
        hash = h;
    }

    public final K getKey() {
        return key;
    }

    public final V getValue() {
        return value;
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    public final boolean equals(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry e = (Map.Entry)o;
        Object k1 = getKey();
        Object k2 = e.getKey();
        if (k1 == k2 || (k1 != null && k1.equals(k2))) {
            Object v1 = getValue();
            Object v2 = e.getValue();
            if (v1 == v2 || (v1 != null && v1.equals(v2)))
                return true;
        }
        return false;
    }

    public final int hashCode() {
        return Objects.hashCode(getKey())
                ^ Objects.hashCode(getValue());
    }

    public final String toString() {
        return getKey() + "=" + getValue();
    }
    void recordAccess(HashMap<K,V> m) {}
    void recordRemoval(HashMap<K,V> m) {}
}
```

（5）向 HashMap 中添加 key->value 对

```
public V put(K key, V value) {
    //处理空值的情况，具体细节暂不考虑
    if (table == EMPTY_TABLE) {
        inflateTable(threshold);
    }
    if (key == null)
        return putForNullKey(value);
    //1.根据传入的key求得哈希值
    int hash = hash(key);
    //2.根据hash值确定在数组中的位置
    int i = indexFor(hash, table.length);
    //3.根据e是否为null分情况向table数组中添加Entry(也就是key->value对)
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(hash, key, value, i);
    return null;
}
```

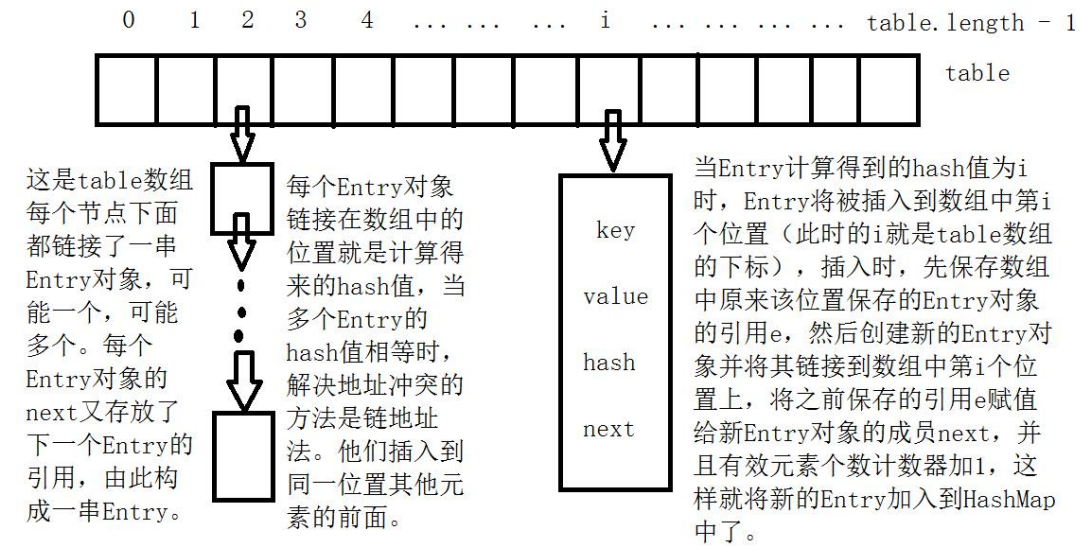
```
static int indexFor(int h, int length) {
    return h & (length-1);  //得到一个[0,length-1]区间内的数值
}
//说明：hash & table.length-1 <=> hash % table.length
//注意：以上公式只在table.length是2的n次幂时才成立
```

```
void addEntry(int hash, K key, V value, int bucketIndex) {
    //如需扩容，则进行扩容处理（具体扩容细节暂不考虑）
    if ((size >= threshold) && (null != table[bucketIndex])) {
        resize(2 * table.length);  //每次扩容都是之前容量的2倍
        hash = (null != key) ? hash(key) : 0;
        bucketIndex = indexFor(hash, table.length);
    }
    createEntry(hash, key, value, bucketIndex);  //真正添加Entry的方法
}
```

```
void createEntry(int hash, K key, V value, int bucketIndex) {
    //保存table数组相应位置已经链接好的成员的引用
    Entry<K,V> e = table[bucketIndex];
```

```
    //将创建的新的Entry实体链接到Table已经计算好的相应位置上，
    //并且将刚刚保存的原来table数组对应位置的一串对象链接到该对象尾部
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    size++;  //保存有有效元素个数的计数器size+1
}
```

示意图：



（6）根据传入的 key 求得哈希值的算法

```
final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }
    h ^= k.hashCode();  //根据key的哈希值再次哈希
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);   //得到最终的hash值
}
```

```
transient int hashSeed = 0;
```