以下是一些常用的 Java `String` 方法：

1. **length()**: 返回字符串的长度。

```
String str = "Hello, World!";
int length = str.length(); // 13
```

2. **charAt(int index)**: 返回指定索引处的字符。

```
String str = "Hello, World!";
char ch = str.charAt(0); // 'H'
```

3. **substring(int beginIndex, int endIndex)**: 返回从 `beginIndex` 到 `endIndex` 之间的子字符串。

```
String str = "Hello, World!";
String sub = str.substring(0, 5); //
"Hello"
```

4. **contains(CharSequence s)**: 判断字符串是否包含指定的字符序列。

```
String str = "Hello, World!";
boolean contains = str.contains("World");
// true
```

5. **indexOf(String str)**: 返回指定子字符串在此字符串中第一次出现的索引。

```
String str = "Hello, World!";
int index = str.indexOf("World"); // 7
```

6. **toUpperCase()**: 将字符串转换为大写。

```
String str = "Hello, World!";
String upper = str.toUpperCase(); // "HELLO, WORLD!"
```

7. **toLowerCase()**: 将字符串转换为小写。

```
String str = "Hello, World!";
String lower = str.toLowerCase(); // "hello, world!"
```

8. **trim()**: 去除字符串两端的空白字符。

```
String str = "Hello, World!";
String trimmed = str.trim(); // "Hello, World!" (如果原字符串有前后空格)
```

9. **replace(CharSequence target, CharSequence replacement)**: 将字符串中的所有目标字符序列替换为新的字符序列。

```
String str = "Hello, World!";
String replaced = str.replace("World", "Java"); // "Hello, Java!"
```

10. **split(String regex)**: 根据给定的正则表达式分割字符串。

```
String str = "Hello, World!";
String[] parts = str.split(","); // ["Hello", "World!"]
```

以下是 StringBuilder 类的一些常用方法：

1. **append()**：将指定的字符串追加到此字符序列。

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
// sb 现在包含 "Hello World"
```

2. **insert()**：在指定位置插入字符串。

```
StringBuilder sb = new StringBuilder("Hello");
sb.insert(5, " World");
// sb 现在包含 "Hello World"
```

3. **delete()**：删除从 start 到 end-1 位置的字符。

```
StringBuilder sb = new StringBuilder("Hello World");
sb.delete(5, 11);
// sb 现在包含 "Hello"
```

4. **deleteCharAt()**：删除指定位置的字符。

```
StringBuilder sb = new StringBuilder("Hello");
sb.deleteCharAt(4);
// sb 现在包含 "Hell"
```

5. **replace()**：用指定的字符串替换从 start 到 end-1 位置的字符。

```
StringBuilder sb = new StringBuilder("Hello World");
sb.replace(6, 11, "Java");
// sb 现在包含 "Hello Java"
```

6. **reverse()**：将此字符序列用其反转形式取代。

```
StringBuilder sb = new StringBuilder("Hello");
sb.reverse();
// sb 现在包含 "olleH"
```

7. **toString()**：将此字符序列转换为一个字符串。

```
StringBuilder sb = new StringBuilder("Hello");
String str = sb.toString();
// str 现在包含 "Hello"
```

8. **setCharAt()**：将指定位置的字符设置为 ch。

```
StringBuilder sb = new StringBuilder("Hello");
sb.setCharAt(1, 'a');
// sb 现在包含 "Hallo"
```

9. **length()**：返回字符序列的长度。

```
StringBuilder sb = new StringBuilder("Hello");
int len = sb.length();
// len 现在是 5
```

10. **capacity()**：返回当前容量。

```java
StringBuilder sb = new StringBuilder("Hello");
int cap = sb.capacity();
// cap 现在是 21（默认容量为 16，加上 "Hello" 的长度 5）
```

数字	^[0-9]+$
m-n位的数字	^\d{m,n}$
字母	^[a-zA-Z]+$
空格	^\s*$
非空格	^\S*$

str.length()	
str1.concat(str2)	
str.indexOf("a")	
str.lastIndexOf("a")	
str.charAt(3)	

str.subString(3)	[3, str.length()]
str.subString(3, 6)	[3, 6)
str.trim()	
str.replace("ab", "cd")	
str.replaceAll("[\d*]", "")	
str.startsWith("ab")	
str.startsWith("bc", 1)	
str.endsWith("de")	
str1.equalsIgnoreCase(str2)	
str1.compareTo(str2)	
Integer.parseInt(str)	
String.valueOf(num)	
str.split(" ")	
str.split(" ", 2)	
str.split("\\.")	
str.split("and|or")	
char[] myChar = str.toCharArray()	
byte[] myByte = str.getByte()	
StringBuffer str = new StringBuffer("hello")	

str.append(" world")	
str.delete(0, 5) -> world	[0, 5]
str.deleteCharAt(str.length() - 1)	
str.reverse()	
String s = "1"; str.replace(2, 4, s)	
str.insert(1, 2)	
str.setCharAt(3, "g")	

1. 模拟实现trim()方法

创建一个函数，遍历字符串，记录第一个非空格字符的位置作为左边界，同时记录最后一个非空格字符的位置作为右边界，然后截取左右边界的子串。

```
public String customTrim(String str) {
    int start = 0, end = str.length() - 1;
    while (start < str.length() &&
str.charAt(start) == ' ') start++;
    while (end >= 0 && str.charAt(end) == '
') end--;
    return str.subString(start, end + 1);
}
```

2. 字符串反转

整体反转：将字符串转换为字符数组，使用双指针交换首尾字符直至中间，然后重新构建字符串。
指定部分反转：同样将子串转换为字符数组，使用双指针交换指定范围内的首尾字符。

```
public String reverseWholeString(String s) {
    return reverseSubstring(s, 0, str.length() - 1)
}

// 反转指定部分字符串
public String reverseSubstring(String s, int ,start int end) {
    char[] chars = s.toCharArray();
    while (start < end) {
        char temp = chars[start];
        chars[start] = chars[end];
        chars[end] = temp;
        start++;
        end--;
    }
    return new String(chars);
}
```

3. 查找子串出现次数

使用indexOf()方法，从头开始查找子串，每当找到一个就更新计数，并将下次查找的起始位置设为本次查找结束位置+1，直到找不到为止。

```java
public int countSubStr(String str, String subStr) {
    int count = 0, index = 0;
    while ((index = str.indexOf(subStr, index)) != -1) {
        count++;
        index += subStr.length();
    }
    return count;
}
```

4. 判断字符串是否为回文

可以通过比较字符串与其反转字符串是否相等来判断，或者利用双指针从两端向中间移动比较字符是否一致。

```java
public boolean isPalindrome(String str) {
    return new StringBuffer(str).reverse().toString().equals(str);
}

public boolean isPalindrome(String str) {
    str = str.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();
    int start = 0, end = str.length() - 1;
    while (start < end) {
        if (str.charAt(start) != str.charAt(end)) {
            return false;
        }
        start++;
```

```
            end--;
        }
        return true;
    }
```

5. 连接字符串不使用+或concat()

创建一个StringBuilder或StringBuffer对象，调用append()方法逐个添加字符串，最后调用toString()方法返回结果。

```
public String concatenateStrings(String... strings) {
    StringBuilder sb = new StringBuilder();
    for (String str : strings) {
        sb.append(str);
    }
    return sb.toString();
}
```

6. 替换子串

使用StringBuilder的replace()方法，或者手动遍历字符串并在发现匹配子串时插入新子串。

```
public String replaceSubstring(String original, String oldSubstring, String newSubstring) {
    return original.replace(oldSubstring, newSubstring);
}
```

7. 检查字符串是否包含连续重复字符

遍历字符串，比较当前字符与下一个字符是否相等，若相等则表示存在连续重复字符。

```
public boolean hasConsecutiveRepeatingChars(String str) {
    for (int i = 0; i < str.length() - 1; i++) {
        if (str.charAt(i) == str.charAt(i + 1)) {
            return true;
        }
    }
    return false;
}
```

8. 删除字符串中连续的重复字符

创建一个新的StringBuilder，遍历输入字符串，在连续字符重复时只将首个字符添加至新字符串中。

```java
public String
removeConsecutiveDuplicates(String str) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < str.length() - 1;) {
        char currentStr = str.charAt(i);
        sb.append(currentStr);
        while (i < str.length() - 1 &&
str.charAt(i) == str.charAt(i + 1)) {
            i++;
        }
        i++;
    }
    return sb.toString();
}
```

9. 翻转单词顺序

分割字符串为单词数组，然后逆序遍历数组并将单词之间加入空格，最后合并成新的字符串。

```java
public String reverseWordsInString(String str) {
    String[] words = str.split("\\s+");
    StringBuilder reversed = new StringBuilder();
    for (int i = str.length() - 1; i >= 0; i--) {
        reversed.append(word[i]);
        if (i > 0) {
            reversed.append(' ');
        }
    }
    return reversed.toString().trim();
}
```

10. 字符串中最长的公共前后缀

从前向后遍历字符串，同时从后向前遍历字符串，对比两者的字符，找到最长的公共前缀和后缀。

```java
public String longestCommonPrefixSuffix(String str) {
    if (str.isEmpty()) {
        return "";
    }
    int n = str.length();
    String reversedStr = new StringBuilder().reverse().toString();
    for (int i = 0; i < str.length() - 1; i++) {
        if (str.charAt(i) != reversedStr.charAt(i)) {
            break;
        }
    }
    return str.subString(0, i);
}
```

11. KMP算法实现

构建next数组，用于存储模式串的部分匹配值，然后在文本串中搜索，失败时利用next数组快速回溯。

```java
private static int[] computeLps(String pattern) {
    int[] lps = new int[pattern.length()];
    int len = 0;
    for (int i = 1; i < pattern.length(); i++) {
        while (len > 0 && pattern.charAt(len) != pattern.charAt(i)) {
            len = lps[len - 1];
```

```
        }
        if (pattern.charAt(len) == pattern.charAt(i)) {
            len++;
        }
        lps[i] = len;
    }
    return lps;
}

public static int kmpSearch(String text, String pattern) {
    if (text == null || pattern == null) return -1;
    int[] lps = computeLps(pattern);
    int j = 0;
    for (int i = 0; i < text.length() - 1; i++) {
        while (j > 0 && text.charAt(i) != pattern.charAt(j)) {
            j = lps[j - 1];
        }
        if (text.charAt(i) == pattern.charAt(j)) {
            j++;
        }
        if (j == pattern.length()) {
            return i - pattern.length() + 1;
        }
    }
    return -1;
}
```

12. Z字形变换

创建一个二维数组模拟Z字形路径，根据行数和列数的奇偶性决定字符的填充方向。

```
1
```

13. 判断两个字符串是否互为变形词（异位词）

将两个字符串转换为字符数组并排序，比较排序后的数组是否相等。

```
1 public static boolean areAnagrams(String s,
  String t) {
2     if (s.length() != t.length()) return
  false;
3     char[] sChars = s.toCharArray(), tChars =
  t.toCharArray();
4     Arrays.sort(sChars);
5     Arrays.sort(tChars);
6     return Arrays.equals(sChars, tChars);
7 }
```

14. 计算字符串的最长递增子序列

使用动态规划思想，dp[i]表示以第i个字符结尾的最长递增子序列长度，遍历字符串更新dp数组。

```
1 public static int
  findLongestIncreasingSubsequenceLength(Strin
  g s) {
2     if (s.isEmpty() || s.length() == 0) {
3         return 0;
```

```
    }
    int[] dp = new int[s.length()];
    for (int i = 0; i < s.length(); i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++) {
            if (s.charAt(i) > s.charAt(j)) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }
    }
    int maxLength = 0;
    for (int length : dp) {
        maxLength = Math.max(maxLength, length);
    }
    return maxLength;
}
```

15. 字符串分割问题

使用Java的split()方法传入分隔符，或者手动遍历字符串并根据分隔符切割成子串放入数组。

```
public static List<String> splitStringByDelimiter(String s, String delimiter) {
    return Arrays.stream(s.split(delimiter))
        .map(String::trim)
        .collect(Collectors.toList());
}
```

16. 最小窗口子串

使用滑动窗口方法，维持一个满足条件的窗口，不断尝试收缩和扩展窗口，记录满足条件的最小区间。

```
public String minWindow(String s, String t) {

}
```

17. 验证括号的有效性

使用栈数据结构，遍历字符串，遇到左括号入栈，遇到右括号检查栈顶元素是否与之匹配，匹配则出栈，遍历结束后栈为空则有效。

```
public boolean isValid(String s) {
    Deque<String> deque = new LinkedList<>();
    for (char c : s.toCharArray()) {
        if (c == '(' || c == '[' || c == '{') {
            deque.push(c);
        } else if (c == ')' && !deque.isEmpty() && deque.peek() == '(') {
            deque.pop();
        } else if (c == ']' && !deque.isEmpty() && deque.peek() == '[') {
            deque.pop();
        } else if (c == '}' && !deque.isEmpty() && deque.peek() == '{') {
            deque.pop();
        } else {
            return false;
```

```
14            }
15        }
16        return deque.isEmpty();
17    }
```

18. 字符替换求解最小子串

计算字符映射后的字符串与目标字符串的编辑距离，编辑距离即为最少替换次数。

```
1  public String minReplaceToMakeGood(String s,
   Map<Character, Character> mapping) {
2
3  }
```