5. 最长回文子串.ts

> 题目策略：多维动态规划
>
> leetcode 地址：https://leetcode.cn/problems/longest-palindromic-substring/description/

问题描述

1. 问题：给你一个字符串 s，找到 s 中最长的回文子串。s 仅由数字和英文字母组成。如果字符串向前和向后读都相同，则它满足回文性。
2. 示例
   - Input s = "babad" Output "bab"
   - Input s = "cbbd" Output = "bb"

问题解决 - 1 brute force

1. 思路

```
/**
 * - 暴力解法：遍历字符串的所有可能子串，检查每个子串是否回文，并记录最长的回文子串
 * - 该解法的时间复杂度为 O(n^3)，不推荐
 */
```

2. 时间复杂度 `O(n^3)`
3. 代码实现

```
function longestPalindrome_bruteForce(s: string): string {
  /* 检查是否回文，时间复杂度 O(n) */
  // const isPalindrome = (str: string): boolean => {
  //   let i = 0;
  //   let j = str.length - 1;

  //   while (i <= j) {
  //     if (str.charAt(i) !== str.charAt(j)) return false;
  //     i++;
  //     j--;
```

```
  //   }

  //   return true;
  // };
  /* 检查是否回文，时间复杂度 O(n) - 优化 1，减少了不必要的字符串截取 */
  const isPalindrome = (s: string, left: number, right: number): boolean =>
{
    while (left <= right) {
      if (s[left] !== s[right]) return false;
      left++;
      right--;
    }
    return true;
  };

  let longestPalindrome = ""; // 最长的回文子串
  /* 暴力检查所有子串是不是回文子串，时间复杂度 O(n^3) */
  for (let i = 0; i < s.length; i++) {
    /* 剪枝：发如果剩余的子串长度小于 longestPalindrome，则直接跳过 - 优化 2，通过剪枝
避免不必要的检查 */
    for (let j = i; j < s.length; j++) {
      const curStrLen = j - i + 1; // 当前子串的长度
      if (curStrLen <= longestPalindrome.length) continue;
      if (isPalindrome(s, i, j)) longestPalindrome = s.slice(i, j + 1);
    }
  }
  return longestPalindrome;
}
```

问题解决 - 2 center extension

1. 思路

```
/**
 * - 中心扩展法：回文字符具有对称性，因此可以以每个字符为中心，向两边扩展以寻找最长的回文子
串
 * - 该解法的实现步骤为：
 *    1. 遍历字符串 s 的每个字符，将其作为回文中心
 *    2. 对于每个中心，向左右扩展，判断左右两侧的字符是否相同。相同则继续扩展，否则停止。
 *    3. 这里需要注意回文中心可以是单个字符，也可是两个连续字符。
 *    4. 在扩展过程中，记录最长的回文子串
 * - 该解法的时间复杂度为 O(n^2)
 */
```

2. 时间复杂度 `O(n^2)`

3. 代码实现

```
function longestPalindrome_centerExtension(s: string): string {
  if (s.length === 0) return ""; // 优化 1，处理空字符串的情况

  /* 根据提供的回文中心，扩展出最长的回文子串在 s 中的左右索引。
    当 left === right 时，回文子串为奇数长度；
    当 left + 1 === right 时，回文子串为偶数长度；
    其他情况报错。 */
  /* 扩展回文子串并返回左右边姐 */
  const extensionPalindrome = (
    left: number,
    right: number
  ): [number, number] => {
    if (left !== right && left !== right - 1) throw new Error("非法的索引");
    // 扩展回文子串
    while (left >= 0 && right < s.length && s[left] === s[right]) {
      left--;
      right++;
    }
    return [left + 1, right - 1]; // 返回有效的回文子串的左右边界
  };

  // let longestPalindrome = ""; // 记录最长的回文子串
  let [start, maxLen] = [0, 1]; // 记录最长的回文子串在 s 中的起始索引及长度，优化
2，避免了对字符串 s 的重复截取
  for (let center = 0; center < s.length; center++) {
    // 1. 以 center 为回文中心进行扩展
    // 奇数长度的回文扩展
    const [left1, right1] = extensionPalindrome(center, center);
    // const curStrLen1 = right1 - left1 + 1;
    // if (curStrLen1 > longestPalindrome.length)
    //   longestPalindrome = s.slice(left1, right1 + 1);
    const len1 = right1 - left1 + 1;
    if (len1 > maxLen) {
      start = left1;
      maxLen = len1;
    }

    // 2. 以 center 和 center + 1 为回文中心进行扩展
    // 偶数长度的回文扩展
    if (center >= s.length - 1) continue;
    const [left2, right2] = extensionPalindrome(center, center + 1);
    // const curStrLen2 = right2 - left2 + 1;
```

```
      // if (curStrLen2 > longestPalindrome.length)
      //   longestPalindrome = s.slice(left2, right2 + 1);
      const len2 = right2 - left2 + 1;
      if (len2 > maxLen) {
        start = left2;
        maxLen = len2;
      }
    }
    return s.slice(start, start + maxLen);
}
```

问题解决 - 3 dynamic programming

1. 思路

```
/**
 * - 动态规划法：使用动态规划记录每个子串是否是回文，从较短的子串开始，逐渐向较长的子串扩展
 * - 该解法的实现步骤为：
 *    1. 创建二维布尔数组 dp 用于表示 s 的子串是不是回文，如 dp[i][j] 表示 s[i:j] 是不是回文
 *    2. 初始化布尔数组 dp：(1) 所有长度为 1 的子串都是回文，即 s[k, k] = true (2) 所有长度为 2 的子串，如果两字符相同也是回文
 *    3. 遍历所有可能的子串长度，从 3 开始，如果 s[i] === s[j] && dp[i + 1][j - 1] === true ⟹ dp[i][j] = true，即字符串 s[i:j] 是回文
 *    4. 记录最长的回文子串的起始位置和长度，即 i 和 j - i + 1
 *    5. 注意：数组 dp 的规模为 n * n，其中 dp[i][j] = dp[j][i]，一般确保 j ⩾ i，即可以只填充数组的右上三角部分
 * - 该解法的时间复杂度为 O(n^2)
 */
```

2. 时间复杂度 `O(n^2)`
3. 代码实现

```
function longestPalindrome_dynamicProgramming(s: string): string {
  if (s.length === 0) return "";
  let [start, maxLen] = [0, 1]; // 记录最长的回文子串的起始位置和长度

  // 创建 dp 数组，dp[i, j] 表示 s[i, j] 是否回文
  const dp: Array<boolean[]> = new Array(s.length);
  for (let i = 0; i < s.length; i++) {
    dp[i] = new Array(s.length).fill(false);
  }
```

```
  // 遍历所有可能的字串长度，包含长度为 1 和 2 的情况，优化 1，将初始化逻辑与主循环合并，从而避免冗余操作
  for (let len = 1; len ⩽ s.length; len++) {
    for (let i = 0; i ⩽ s.length - len; i++) {
      const j = i + len - 1;
      /*
       * 子串回文性判断
       * - 如果 len = 1，则所有子串必定回文
       * - 如果 len = 2，则两个字符相同子串的必定回文
       * - 如果 len > 3，则首尾字符相同且去除首尾的子串回文的子串必定回文
       */
      if (s[i] === s[j] && (len ⩽ 2 || dp[i + 1][j - 1])) {
        dp[i][j] = true;
        if (len > maxLen) [start, maxLen] = [i, len];
      }
    }
  }

  // // 初始化 dp 数组
  // for (let i = 0; i < s.length; i++) {
  //   dp[i][i] = true;
  //   if (i > 0 && s[i - 1] === s[i]) {
  //     dp[i - 1][i] = true;
  //     if (maxLen < 2) [start, maxLen] = [i - 1, 2];
  //   }
  // }

  // // 遍历所有可能的子串长度，从 3 开始
  // for (let curLen = 3; curLen ⩽ s.length; curLen++) {
  //   // 遍历指定长度 len 下的所有子串
  //   for (let curStart = 0; curStart ⩽ s.length - curLen; curStart++) {
  //     if (
  //       s[curStart] === s[curStart + curLen - 1] &&
  //       dp[curStart + 1][curStart + curLen - 2]
  //     ) {
  //       dp[curStart][curStart + curLen - 1] = true;
  //       if (curLen > maxLen) [start, maxLen] = [curStart, curLen];
  //     }
  //   }
  // }

  // for (let len = 1; len ⩽ s.length; len++) {
  //   for (let i = 0; i ⩽ s.length - len; i++) {
  //     const j = i + len - 1; // 计算指定长度下的子串的最后一个字符的索引
  //
```

```
    //   }
    // }
    return s.slice(start, start + maxLen);
  }
```

性能测试

```
export const performanceTest = () => {
  console.time("最长回文子串 - 暴力解法");
  console.log(longestPalindrome_bruteForce("babad"));
  console.log(longestPalindrome_bruteForce("cbbd"));
  console.log(longestPalindrome_bruteForce("a"));
  console.timeEnd("最长回文子串 - 暴力解法"); // 1.208ms
  console.time("最长回文子串 - 中心扩展法");
  console.log(longestPalindrome_centerExtension("babad"));
  console.log(longestPalindrome_centerExtension("cbbd"));
  console.log(longestPalindrome_centerExtension("a"));
  console.timeEnd("最长回文子串 - 中心扩展法"); // 0.31ms
  console.time("最长回文子串 - 动态规划法");
  console.log(longestPalindrome_dynamicProgramming("babad"));
  console.log(longestPalindrome_dynamicProgramming("cbbd"));
  console.log(longestPalindrome_dynamicProgramming("a"));
  console.timeEnd("最长回文子串 - 动态规划法"); // 0.436ms
};
```