双向文本（BiDirectional Text）处理机制

叶家炜

Oct 04, 2025

1　从乱码到清晰：揭秘阿拉伯语与希伯来语文本的渲染逻辑

想象一下，你在一个简单的文本编辑器中输入字符串 `Hello - 123 -  `，期望看到清晰的显示，但结果却可能是一片混乱：阿拉伯语部分顺序错乱，数字和标点符号位置不当。这种问题并非偶然，而是源于不同语言书写方向的冲突。英语和中文等语言遵循从左向右的书写顺序，而阿拉伯语和希伯来语等语言则采用从右向左的顺序。当这些方向混合时，如果没有适当的处理机制，渲染就会失败。这正是双向文本处理所要解决的核心问题。本文将带你深入探讨双向文本的复杂性，并亲手实现一个简化版的 Unicode 双向算法，让你从理论到实践全面掌握这一关键技术。

1.1　背景知识

双向文本指的是包含从左向右和从右向左混合书写方向的文本内容。例如，LTR 语言如英语、中文和俄语，其字符顺序从左侧开始向右延伸；而 RTL 语言如阿拉伯语和希伯来语，则从右侧开始向左书写。如果不加处理，简单地将这些文本拼接在一起，会导致显示顺序与逻辑顺序脱节，从而产生乱码。逻辑顺序指的是文本在内存中的存储序列，而显示顺序则是最终在屏幕上呈现的视觉序列。Unicode 标准通过其双向算法规范 UAX #9 来解决这一问题，它为文本渲染引擎提供了一套规则，确保混合方向文本的正确显示。

1.2　核心原理

Unicode 双向算法是处理双向文本的核心，它分为三个阶段：分解段落、解析与重置、以及重新排序与镜像。首先，在阶段一，算法将输入文本按段落分隔符如换行符拆分成独立段落，并确定每个段落的基础方向。基础方向可以通过启发式规则基于首强字符推断，或由外部协议如 HTML 的 `dir` 属性指定。强字符包括 L、R 和 AL 等类别，它们具有明确的方向性；弱字符如数字和标点符号方向性较弱；中性字符如空格和分隔符则依赖上下文确定方向。

阶段二是算法的核心，涉及解析字符方向并重置层级。层级是一个整数，0 表示 LTR 基础方向，1 表示 RTL，以此类推。算法使用栈结构处理显式嵌入字符如 RLE 和 LRE，以及重写字符如 LRO 和 RLO，这些字符可以临时改变文本方向。例如，RLE 字符会推入一个 RTL 嵌入级别到栈中，直到遇到 PDF 字符才弹出。接下来，算法解析中性字符，根据规则 N1 和 N2 确定其方向：N1 规则要求中性字符继承前一个强字符的方向，如果不存在，则继承段落基础方向；N2 规则处理数字周围的中性字符，确保它们与数字方向一致。举例来说，一个句号 `.` 在英文和阿拉伯文混合文本中，会根据相邻强字符决定其显示位置。最后，规则 L1 处理数字，确保在 RTL 段落中数字仍保持 LTR 内部顺序，避免顺序混乱。

阶段三负责重新排序和字符镜像。根据计算出的层级，算法使用反转层级方法：偶数层级从左向右显示，奇数层级从右向左显示。同时，字符镜像会将对称字符如括号 () 在 RTL 上下文中替换为镜像形式)(，以保持视觉一致性。这三个阶段共同确保文本从逻辑顺序正确映射到显示顺序。

1.3 动手实践

现在，我们来动手实现一个简化的双向文本处理算法。这个版本专注于纯文本处理，忽略显式嵌入字符和数字形状处理，输入为一个字符串和段落基础方向，输出重新排序后的字符数组。我们选择 Python 作为实现语言，因其简洁性和丰富的 Unicode 支持。首先，我们需要获取字符的双向类别，可以使用 Python 的 `unicodedata` 库来查询 `bidi_class` 属性。

在步骤一中，我们设置环境并定义字符方向性查找函数。以下代码初始化一个字典，映射常见字符到其双向类别，但为了简化，我们硬编码一些示例字符的类别。例如，英文字母归类为 L，阿拉伯字母归类为 AL，数字为 EN，标点为 ON。

```
import unicodedata

def get_bidi_class(char):
    return unicodedata.bidi_class(char)
```

这段代码使用 `unicodedata.bidi_class` 函数获取任意 Unicode 字符的双向类别。例如，字符 'A' 的类别是 'L'，表示从左向右；字符 '؟' 的类别是 'AL'，表示阿拉伯字母从右向左。通过这个函数，我们可以为每个字符分配初始方向属性，为后续解析奠定基础。

步骤二实现方向解析，遍历字符串并为每个字符分配方向，同时处理中性字符。我们根据规则 N1 和 N2 调整中性字符的方向：向前和向后查找最近的强字符，如果找到，则继承其方向；否则使用段落基础方向。

```
def resolve_neutral_chars(text, base_dir):
    chars = list(text)
    bidi_classes = [get_bidi_class(c) for c in chars]
    for i, char_class in enumerate(bidi_classes):
        if char_class == 'ON': # 中性字符
            left_strong = None
            right_strong = None
            # 向左查找强字符
            for j in range(i-1, -1, -1):
                if bidi_classes[j] in ['L', 'R', 'AL']:
                    left_strong = bidi_classes[j]
                    break
            # 向右查找强字符
            for j in range(i+1, len(bidi_classes)):
                if bidi_classes[j] in ['L', 'R', 'AL']:
                    right_strong = bidi_classes[j]
                    break
```

```
        # 应用规则 N1
        if left_strong and right_strong:
            if left_strong == right_strong:
                bidi_classes[i] = left_strong
            else:
                bidi_classes[i] = base_dir
        elif left_strong:
            bidi_classes[i] = left_strong
        elif right_strong:
            bidi_classes[i] = right_strong
        else:
            bidi_classes[i] = base_dir
    return bidi_classes
```

这段代码首先将输入字符串转换为字符列表，并获取每个字符的双向类别。然后，它遍历每个字符，如果字符是中性类别 ON，则向前和向后搜索最近的强字符 L、R 或 AL。根据规则 N1，如果左右强字符方向一致，则中性字符继承该方向；如果不一致或找不到强字符，则使用段落基础方向。这确保了中性字符如标点符号在混合文本中正确对齐。

步骤三实现重新排序，根据层级将文本分成多个运行，并对奇数层级运行进行反转。我们假设所有字符初始层级为 0 或 1，基于段落基础方向。

```
def reorder_bidi_text(text, base_dir):
    base_level = 0 if base_dir == 'L' else 1
    levels = [base_level] * len(text)
    # 简化处理：假设无显式嵌入，因此层级不变
    runs = []
    current_run = []
    current_level = base_level
    for i, char in enumerate(text):
        if levels[i] != current_level:
            if current_run:
                runs.append((current_level, ''.join(current_run)))
            current_run = [char]
            current_level = levels[i]
        else:
            current_run.append(char)
    if current_run:
        runs.append((current_level, ''.join(current_run)))
    # 反转奇数层级运行
    reordered_chars = []
    for level, run in runs:
```

```
        if level % 2 == 1: # 奇数层级，从右向左
            reordered_chars.extend(list(reversed(run)))
        else: # 偶数层级，从左向右
            reordered_chars.extend(list(run))
    return ''.join(reordered_chars)
```

这段代码首先初始化层级数组，假设所有字符具有相同的层级（基于段落基础方向）。然后，它将文本分成连续运行，每个运行包含相同层级的字符。对于奇数层级运行，代码使用 `reversed` 函数反转字符顺序，模拟从右向左显示；偶数层级运行保持原顺序。最后，将所有运行拼接成最终字符串。例如，输入字符串 `car (CAR) car` 在 RTL 基础方向下，输出应为 `rac (RAC) rac`，因为括号内的部分在奇数层级被反转。

为了测试我们的实现，我们可以运行一个示例：给定输入 `Hello - 123 -` 和基础方向 LTR，算法应正确解析中性字符并重新排序 RTL 部分。通过打印中间步骤如解析后的方向和最终输出，我们可以验证算法的正确性。

1.4 现实世界的应用

在实际应用中，双向文本处理广泛集成于现代技术中。例如，在 HTML 和 CSS 中，可以使用 `dir` 属性指定文本方向，配合 `unicode-bidi` 和 `direction` 属性实现复杂渲染。文本引擎如 HarfBuzz 和 Pango 实现了完整的 UAX #9 算法，支持多语言文本布局。终端和编辑器如 VS Code 也内置了双向文本支持，确保代码和注释在混合语言环境下的可读性。需要注意的是，我们的简化实现忽略了显式嵌入字符和数字处理，完整版本涉及更多边界情况和优化，但这些库提供了可靠的生产级解决方案。

通过本文，我们深入探讨了双向文本的核心问题，从乱码现象出发，解析了 Unicode 双向算法的三个阶段，并亲手实现了一个简化版本。关键收获在于理解了逻辑顺序与显示顺序的映射关系，以及算法如何通过层级和方向解析确保文本正确渲染。尽管我们的实现侧重于基础功能，但它为进一步探索完整规范奠定了基础。鼓励读者阅读 UAX #9 官方文档，或尝试集成成熟库如 HarfBuzz 到实际项目中，以解决更复杂的双向文本挑战。

1.5 参考资料

Unicode 官方标准 UAX #9 提供了双向算法的完整规范，可在 Unicode 官网查阅。W3C 关于双向文本的指南提供了 Web 开发中的实践建议。HarfBuzz 和 Pango 项目是开源文本布局引擎，源码可用于深入学习。此外，Unicode 联盟提供了测试字符串，可用于验证算法实现。这些资源将帮助你扩展知识并应用于实际场景。