Orez 的设计与实现

李延瑞 (lyr.m2@live.cn)

2025 年 1 月

前言

Donald Knuth 于 20 世纪 80 年代提出文学编程（Literate Programming）的概念并毕生践行。文学编程强调程序源代码的编写首先应便于人类阅读和理解，其次才是被计算机正确编译和执行。

程序的文学化源码通常由某种自然语言（人类语言）和源码片段混合而成。自然语言主要用于阐明程序设计思路以及解释源码片段，其形式通常为某种可用于文档排版的标记语言，如 TEX、Markdown 甚至 HTML 等，这意味着文学编程较传统的代码注释甚至所谓的自明性代码更为全面、准确、直观地阐释程序源码。

目前有一些文学编程或系统可用，其中最广为人知的系统是 Knuth 的 WEB 系统，该系统本身便是以自举的方式开发的，亦即 WEB 系统本身便是文学编程的结果，它以 TEX 语言排版文档，以 Pascal 语言编写程序。TEX 排版系统也是 WEB 自举过程中的一部分。WEB 后来演进为 CWEB，其文档语言依然是 TEX，编程语言则换为 C 语言。

为了追求排版和编程语言的自由，Norman Ramsey 设计并实现了 noweb 程序。除了不依赖具体的排版和编程语言，noweb 的优点还在于它的管线体系是开放性的，能够支持用户编写后端程序，从而让 noweb 支持更多的排版语言。

我曾经很喜欢 noweb，用它编写了一些小程序，但不久之后我发现它并不能很好地满足我的需要，例如在同名源码片段的合并方面，noweb 仅支持向后合并，而无法将一个源码片段插入到另一个同名源码片段之前。在源码片段抽取方面，noweb 不支持批量抽取。noweb 的文学编程标记不能由用户定制，Knuth 的 WEB 系统也是如此。

对我而言，noweb 最大的问题是不能直接支持我喜欢的 ConTEXt[1] 和 Markdown 语言，除非实现相应的后端程序，但是这个过程对很多人而言，甚为困难，原因是 noweb 的作者将程序的文学化源码解析为他自定义的结构化的标记语言，要实现后端程序，需要理解这种结构化文本并自行实现相应的解析过程。现代已经有了许多结构化的标记语言，诸如 XML、JSON、YAML 等皆被主流编程语言以库的形式支持，我们没有必要再纠缠于 noweb 管线所使用的这种小众且缺乏解析器的标记语言。

针对 WEB 和 noweb 的不足，我编写了一个新的工具，取名为 Orez[2]，它在支持用户自定义文学化源码标记，能够实现源码片段的双向合并，能够从程序的文学化源码中批量抽取程序代码，也能将文学化源码转化为 YAML 格式的结构化文本以便于后端程序的实现。这份文档描述了 Orez 的设计和实现，它本身由 Orez 和 ConTEXt 生成。

---

[1] 与 LATEX 同等层次和规模的 TEX 格式。

[2] 英文单词“zero”的反写。

Orez 程序用 C 语言实现。除了 C 标准库外，Orez 依赖 GLib 和 LibCYAML 库。GLib 库为 Orez 的实现除了提供字符串、列表、Hash 表、N 叉树等容器以及命令行参数处理等功能。LibCYAML 则为 Orez 提供了 YAML 格式的配置文件支持。

1 标记

1.1 基本符号

由于文学化的程序源码由文档片段和源码片段组成[3]，Orez 需要定义一些符号以区分它们。换言之，如果一处文档片段毗邻一个源码片段，应当存在一些符号用于表示二者的界限，意味着文档片段的终止或源码片段的开始。我使用符号‘@’隔离两个相邻片段，无论它们是否为同类，即该符号不仅可作为文档片段和源码片段之间的界限，也可以作为两处相邻的文档片段或两个相邻的源码片段之间的界限。例如

```
计算 x 与 y 的点积：

@

sum = 0;
for (i = 0; i < n; i++) sum += x[i] * y[i];
```

仅有‘@’并不足用。‘@’仅能帮助 Orez 意识到存在两部分文本片段，但 Orez 并不知道它们的类型——是文档片段还是源码片段。更重要的是，源码片段需要拥有名字以备其他源码片段引用。一个源码片段的名字应当置于何处呢？为了解决这些问题，必须引入新的符号用于标记源码片段的名字。我选择符号‘#’来达成这一目的。例如

```
@ 计算 x 与 y 的点积 #
sum = 0;
for (i = 0; i < n; i++) sum += x[i] * y[i];
```

上述是名字为“计算 x 与 y 的点积”的源码片段，亦即之前的文档片段现在变成了源码片段的名字。

现在，Orez 能够基于“@ ... #”这样的文本结构区分文档片段与源码片段了，只需禁止许前者包含这样的文本结构即可。为了便于描述，称“@ ... #”为源码片段的首部。

源码片段的首部可能会过长而需要分为多行。为了增强源码片段首部的终结符‘#’的语义，采用续行符对源码片段首部进行分行，默认的续行符为反斜线‘\’，其用法见以下示例：

```
@ 计算 x 与 y 的点积，\
  将所得结果存于变量 sum #
```

[3] 在 Orez 的程序实现中，文档片段和源码片段分别称为无名片段和有名片段。

```
sum = 0;
for (i = 0; i < n; i++) sum += x[i] * y[i];
```

若在另一个源码片段中引用上述源码片段，只需颠倒目标源码片段首部的‘@’和‘#’的位置，例如

```
@ 点积 #
int dot_product(double x[], double y[], int n)
{
        int i;
        double sum;
        # 计算 x 与 y 的点积 @; /* 源码片段引用 */
        return sum;
}
```

在源码片段中，形如“# ... @”的文本结构称为源码片段的引用。引用终结符之后的同一行文本会被忽略，例如上例中源码引用之后的分号以及注释文本“/* 源码片段引用 */”皆会被忽略。

这些符号几乎是 Orez 所需要的全部符号。后续还会引入一些其他符号，但对于 Orez 的基本功能而言，它们并非必须。

1.2 源码片段累加符

在文学化源码中，源码片段可以同名。同名的源码片段意味着一段复杂的源码被打碎，按照易于阅读和理解的原则散布于各处。Orez 从文学化源码中抽取程序源码时，可将这些同名源码片段按照它们在文学化源码中出现的先后次序合并起来。

例如，假设此处有名为“foobar”的源码片段：

```
@ foobar #
void foobar(void)
{
        printf("foo ");
```

下面是同样名为“foobar”的源码片段：

```
@ foobar #
        printf("bar\n");
}
```

这两个源码片段累加后，结果应为

```
@ foobar #
void foobar(void)
```

```
{
        printf("foo ");
        printf("bar\n");
}
```

现在有一个问题，尽管 Orez 能够累加两个同名的源码片段，但是在阅读文学化源码时，我们却很难快速确定这两个源码片段的先后顺序。当我们在审视这两个源码片段中的一个时，并不知道另一个位于它之前还是之后。文学编程必须让源码易于他人阅读，因此必须继续引入一个符号用于标记两个同名源码片段的先后次序，为此我使用符号 ‘+’。

例如

```
@ foobar # +
        printf("bar");
}
```

‘+’ 意味着当前源码片段是追加到另一个源码片段的尾部的。当我们看到源码片段首部后面出现 ‘+’ 时，便会确定在该源码片段之前必定存在至少一个与之同名的源码片段。

如果我们想将一个源码片段在累加时放在另一个源码片段之前，还需要引入一个符号用于修饰 ‘+’，表示向前追加。我使用 ‘^’ 作为该修饰符，其用法见下面示例：

```
@ foobar #
        printf("bar\n");

@ foobar # ^+
void foobar(void)
{
        printf("foo ");
```

对于上述示例，Orez 对同名源码片段的累加后应当给出以下结果：

```
@ foobar #
void foobar(void)
{
        printf("foo ");
        printf("bar\n");
}
```

1.3 标签

假设我们正在编写一个源码片段，它前面至少有两个源码片段与它同名，我们将它追加到其中一个之前。对于这种问题，‘+’ 与 ‘^+’ 都无法解决，前者只能将一个源码片段累加到与它同名的所有源码片段之后，而后者则反之，只能将一个源码片段累加到与它同名的所有

源码片段之前。若将一个源码片段累加到指定的同名源码片段的之前或之后，必须为后者提供一个标记，用于区分与之同名的源码片段。我的方法是为目标源码片段提供标签。例如

```
@ foobar #
        printf("bar\n");
}

@ foobar # ^+
<tag>
void foobar(void)
{
        printf("foo ");

@ foobar # <lable> ^+
#include <stdio.h>
```

Orez 对上述三个源码片段的累加结果应当为

```
#include <stdio.h>
void foobar(void)
{
        printf("foo ");
        printf("bar\n");
}
```

上述示例中，第二个源码片段首部之后的“<tag>”便是该源码片段的标签。第三个源码片段首部之后的“^+”符号之前引用了“<tag>”，表示该源码片段需要追加到带有该标签的同名源码片段之前。

注意，不允许两个同名的源码片段使用相同的标签。

1.4 编程语言

尽管 Orez 所处理的文学化源码独立于任何编程语言，但是在文档排版过程中为了实现源码片段的高亮渲染，需要知道源码片段所用的编程语言。为了解决该问题，在源码片段的首部的特定位置放置一个标记用于告知 Orez 该源码片段对应的编程语言。例如，下面是 C 语言片段，需要使用标记“[C]”：

```
@ foobar # [C]
void foobar(void)
{
        puts("foobar");
}
```

编程语言标记位于源码片段首部之后，可在引用标签之前，也可在其之后，但必须位于累加运算符之前。以下源码片段示例给出了 Orez 所需的全部符号和标记：

```
@ 源码片段名 # [编程语言] <标签引用> + 或 ^+
<标签>
void foobar(void)
{
        puts("foobar");
}
```

每一个源码片段都需要编程语言标记吗？当然不是。假设一个带有语言标记的源码片段 X，Orez 能够将该语言标记自动传播给 1) 与 X 同名的源码片段，2) 被 X 引用的源码片段，3) 引用 X 的源码片段。

语言标记是可选的。对于那些未带有语言标记的源码片段，Orez 只是会给出像下面这样的警告：

```
WARNING **: Programing Language in <foobar> is unknown.
```

2 词法分析

我从未修习过编译原理这门课程，对其认知也颇为粗浅，故而我不知道这一节内容是否属于编译原理中的词法分析阶段，只是我觉得它应该是，因为它基于文本文件中的信息构建了一个记号列表。

2.1 记号

以下结构体类型表示记号：

```
@ 记号 #
typedef struct {
        OrezTokenType type; /* 记号类型标识 */
        gsize line_number;  /* 记号对应文本在全文中的行号 */
        GString *content;   /* 记号对应的文本内容 */
} OrezToken;
```


=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>

=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>

其中 OrezTokenType 为记号的类型标识，为文学化源码中的文档片段和源码片段所需要的所有标记定义了数字标识，其定义为

```
@ 记号 # ^+
typedef enum  {
        OREZ_SNIPPET_DELIMITER, /* 文档或源码片段界限符 */
        OREZ_SNIPPET_NAME_DELIMITER, /* 片段名字界限符 */
        OREZ_LANGUAGE_BEGINNING_MARK, /* 语言标记开始符 */
        OREZ_LANGUAGE_END_MARK, /* 语言标记终结符 */
        OREZ_SNIPPET_APPENDING_MARK, /* 后向合并运算符 */
        OREZ_SNIPPET_PREPENDING_MARK, /*前向合并运算符 */
        OREZ_TAG_BEGINNING_MARK, /* 代码片段标签开始符 */
        OREZ_TAG_END_MARK, /* 代码片段标签终结符 */
        OREZ_SNIPPET_REFERENCE_BEGINNING_MARK, /* 代码片段引用开始符 */
        OREZ_SNIPPET_REFERENCE_END_MARK, /* 代码片段引用终结符 */
        OREZ_TEXT /* 普通文本 */
} OrezTokenType;
```

=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>

```
=> orez.c <96>
```

虽然在 **1** 节中给出了 Orez 所需要的所有标记，但这些标记所用的皆为默认符号。Orez 用户可通过 YAML 格式的配置文件定义符合自己喜好的符号。以下结构体用于存储默认的或用户自定义的符号：

```
@ 记号 # +
typedef struct {
        char *snippet_delimiter;
        char *snippet_name_delimiter;
        char *snippet_name_continuation;
        char *language_beginning_mark;
        char *language_end_mark;
        char *snippet_appending_mark;
        char *snippet_prepending_mark;
        char *tag_beginning_mark;
        char *tag_end_mark;
        char *snippet_reference_beginning_mark;
        char *snippet_reference_end_mark;
} OrezConfig;
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

定义 `OrezConfig` 的实例 `default_config` 用于存储 Orez 的默认符号：

```
@ 符号表初始化 #
OrezConfig config = {
        .snippet_delimiter = "@",
```

```
        .snippet_name_delimiter = "#",
        .snippet_name_continuation = "\\",
        .language_beginning_mark = "[",
        .language_end_mark = "]",
        .snippet_appending_mark = "+",
        .snippet_prepending_mark = "^+",
        .tag_beginning_mark = "<",
        .tag_end_mark = ">",
        .snippet_reference_beginning_mark = "#",
        .snippet_reference_end_mark = "@"
};
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> Orez 辅助函数 <94>
```

2.2 配置文件

Orez 允许用户通过配置文件自定义一些符号取代相应的默认符号，配置文件为 YAML 格式，其内容为一组映射（键值对），键名与 `OrezSymbols` 的成员变量名相同，例如

```
snippet_reference_beginning_mark: '「'
snippet_reference_end_mark: '」'
```

Orez 基于 LibCYAML 库解析 YAML 格式的配置文件，相关代码如下：

```
@ YAML 配置文件的模式 #
const cyaml_schema_field_t top_mapping_schema[] = {
        CYAML_FIELD_STRING_PTR("snippet_delimiter",
                               CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
```

```
                        OrezSymbols, snippet_delimiter,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("snippet_name_delimiter",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, snippet_name_delimiter,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("snippet_name_continuation",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, snippet_name_continuation,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("language_beginning_mark",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, language_beginning_mark,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("language_end_mark",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, language_end_mark,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("snippet_appending_mark",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, snippet_appending_mark,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("snippet_prepending_mark",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, snippet_prepending_mark,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("tag_beginning_mark",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, tag_beginning_mark,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("tag_end_mark",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, tag_end_mark,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("snippet_reference_beginning_mark",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, snippet_reference_beginning_mark,
                        0, CYAML_UNLIMITED),
CYAML_FIELD_STRING_PTR("snippet_reference_end_mark",
                        CYAML_FLAG_POINTER|CYAML_FLAG_OPTIONAL,
                        OrezSymbols, snippet_reference_end_mark,
```

```
                                0, CYAML_UNLIMITED),
        CYAML_FIELD_END
};
const cyaml_schema_value_t top_schema = {
        CYAML_VALUE_MAPPING(CYAML_FLAG_POINTER, OrezConfig, top_mapping_schema),
};
const cyaml_config_t cyaml_config = {
        .log_fn = cyaml_log,            /* Use the default logging function. */
        .mem_fn = cyaml_mem,            /* Use the default memory allocator. */
        .log_level = CYAML_LOG_WARNING, /* Logging errors and warnings only. */
};
=> YAML 配置文件解析 <11>
=> Orez 辅助函数 <94>
```

基于 YAML 模式，对指定的 YAML 格式的配置文件 `cofig_file` 进行解析的过程如下：

```
@ YAML 配置文件解析 #
# YAML 配置文件的模式 @ <9>
OrezConfig *user_config;
cyaml_err_t err = cyaml_load_file(config_file,
                                  &cyaml_config,
                                  &top_schema,
                                  (cyaml_data_t **)&user_config,
                                  NULL);
if (err != CYAML_OK) {
        fprintf(stderr, "Failed to open config file!\n");
        exit(EXIT_FAILURE);
}
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
```

```
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
```

config 指向的结构体，其成员指针指向由配置文件解析得来的符号；对于配置文件未定义的符号，成员指针的值为 NULL。需要注意的是，该结构体所占用的存储空间由 LibCYAML 分配，在 Orez 进程退出前，需使用以下代码进行释放：

```
@ 释放配置文件占用的资源 #
cyaml_free(&cyaml_config, &top_schema, user_config, 0);
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
```

用 config 指向的结构体中的数据替换 default_config 中相应的成员变量：

```
@ 配置用户定义的符号 #
if (user_config) {
        if (user_config->snippet_delimiter) {
                config.snippet_delimiter
                        = user_config->snippet_delimiter;
        }
        if (user_config->snippet_name_delimiter) {
                config.snippet_name_delimiter
                        = user_config->snippet_name_delimiter;
        }
        if (user_config->snippet_name_continuation) {
                config.snippet_name_continuation
                        = user_config->snippet_name_continuation;
        }
```

```
        if (user_config->language_beginning_mark) {
                config.language_beginning_mark
                        = user_config->language_beginning_mark;
        }
        if (user_config->language_end_mark) {
                config.language_end_mark
                        = user_config->language_end_mark;
        }
        if (user_config->snippet_appending_mark) {
                config.snippet_appending_mark
                        = user_config->snippet_appending_mark;
        }
        if (user_config->snippet_prepending_mark) {
                config.snippet_prepending_mark
                        = user_config->snippet_prepending_mark;
        }
        if (user_config->tag_beginning_mark) {
                config.tag_beginning_mark
                        = user_config->tag_beginning_mark;
        }
        if (user_config->tag_end_mark) {
                config.tag_end_mark
                        = user_config->tag_end_mark;
        }
        if (user_config->snippet_reference_beginning_mark) {
                config.snippet_reference_beginning_mark
                        = user_config->snippet_reference_beginning_mark;
        }
        if (user_config->snippet_reference_end_mark) {
                config.snippet_reference_end_mark
                        = user_config->snippet_reference_end_mark;
        }
}
```


=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>

=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> Orez 辅助函数 <94>

对于所获得的全部符号数据，为了后续便于使用，需将其提升为 GString 类型：

```
@ 记号 # +
typedef struct {
        GString *snippet_delimiter;
        GString *snippet_name_delimiter;
        GString *snippet_name_continuation;
        GString *language_beginning_mark;
        GString *language_end_mark;
        GString *snippet_appending_mark;
        GString *snippet_prepending_mark;
        GString *tag_beginning_mark;
        GString *tag_end_mark;
        GString *snippet_reference_beginning_mark;
        GString *snippet_reference_end_mark;
} OrezSymbols;
```

=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>

提升符号数据：

```
@ 构造符号集 -> symbols #
OrezSymbols *symbols = malloc(sizeof(OrezSymbols));
*symbols = (OrezSymbols) {
        g_string_new(config.snippet_delimiter),
        g_string_new(config.snippet_name_delimiter),
        g_string_new(config.snippet_name_continuation),
        g_string_new(config.language_beginning_mark),
        g_string_new(config.language_end_mark),
        g_string_new(config.snippet_appending_mark),
        g_string_new(config.snippet_prepending_mark),
        g_string_new(config.tag_beginning_mark),
        g_string_new(config.tag_end_mark),
        g_string_new(config.snippet_reference_beginning_mark),
        g_string_new(config.snippet_reference_end_mark)
};
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> Orez 辅助函数 <94>
=> Orez 辅助函数 <94>
```

在 Orez 进程退出前，`symbols` 以及配置文件占用的内存由以下代码释放：

```
@ 释放符号集 # +
g_string_free(symbols->snippet_delimiter, TRUE);
g_string_free(symbols->snippet_name_delimiter, TRUE);
g_string_free(symbols->snippet_name_continuation, TRUE);
g_string_free(symbols->language_beginning_mark, TRUE);
```

```
g_string_free(symbols->language_end_mark, TRUE);
g_string_free(symbols->snippet_appending_mark, TRUE);
g_string_free(symbols->snippet_prepending_mark, TRUE);
g_string_free(symbols->tag_beginning_mark, TRUE);
g_string_free(symbols->tag_end_mark, TRUE);
g_string_free(symbols->snippet_reference_beginning_mark, TRUE);
g_string_free(symbols->snippet_reference_end_mark, TRUE);
free(symbols);
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

2.3 分段

为了降低解析难度——因为我不熟悉编译原理——，将词法分析分为五个阶段。第一阶段将文学化源码文件读入内存，并析取文档片段和代码片段的定界符。假设文学化源码文件名为 `input_file_name`，以下代码打开该文件：

```
@ 打开文学化程序源码文件 -> input #
FILE *input = fopen(input_file_name, "r");
if (!input) {
        g_error("Failed to open input file!\n");
}
=> 分段 -> tokens <20>
```

在读取文件的过程中，需要一个缓冲区——基于 GLib 库的 GString 类型实现——用于临时存储读入的字符。辅助函数 `read_char_from_input` 可从文件中读取当前字符（一个字节），将其存入缓冲区：

```
@ 分段过程所需的辅助函数 #
static int read_char_from_input(FILE *input, GString *cache)
{
        int c = fgetc(input);
        if (c != EOF) {
                g_string_append_c(cache, (gchar)c);
                return c;
        } else return EOF;
}
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

为了达成第一阶段的目标，在读取文件的过程中，每读取一个字节，需检测缓冲区的尾端是否为〈换行符 + 空白字符 + OREZ_SNIPPET_DELIMITER〉，亦即文档片段与源码片段的分割符必须位于一行的开头，它的前面只允许出现空白字符（空格、全角空格或制表符）。辅助函数 tail_is_snippet_delimiter 用于判定缓冲区尾端字符是否满足上述该模式：

```
@ 分段过程所需的辅助函数 # +
# 文本处理辅助函数 @ <112> <112>
static bool tail_is_snippet_delimiter(GString *cache,
                                      GString *snippet_delimiter)
{
        char *p = NULL;
        /* 若 cache 末尾不存在片段界限符 */
        if (snippet_delimiter->len > cache->len) return false;
        p = cache->str + cache->len - snippet_delimiter->len;
        if (strcmp(p, snippet_delimiter->str) != 0) return false;
        /* 否则 */
        enum {IDLE, SPACE, SUCCESS, FAILURE} state = IDLE;
        if (p == cache->str) return true;
        else p--;
        /* 构造字符串区间 [a, b] */
        char *a = cache->str, *b = p;
        /* 从 b 开始逆序遍历 [a, b] */
        while (1) {
                switch (state) {
                case IDLE:
```

```
                        if (*p == ' ') state = SPACE;
                        else if (*p == '\t') state = SPACE;
                        else if (am_i_here(a, b, &p, "  ", -1)) {
                                state = SPACE;
                        }
                        else if (*p == '\n') state = SUCCESS;
                        else state = FAILURE;
                        break;
                case SPACE:
                        if (*p == ' ') state = SPACE;
                        else if (*p == '\t') state = SPACE;
                        else if (am_i_here(a, b, &p, "  ", -1)) state = SPACE;
                        else if (*p == '\n') state = SUCCESS;
                        else state = FAILURE;
                        break;
                default:
                        g_error("Illegal state in <<< %s >>>", cache->str);
                }
                if (p == a) {
                        /* 字符串逆序遍历到头，此时匹配过程既未成功，
                           亦未失败，则特命其成功 */
                        state = SUCCESS;
                }
                if (state == FAILURE || state == SUCCESS) break;
                else p--;
        }
        return (state == SUCCESS) ? true : false;
}
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

上述代码调用的辅助函数 `am_i_here` 的定义和说明见本文档的附录部分。

若 `tail_is_snippet_delimiter` 返回 `true`，表示缓冲区尾部是片段界限符，需将其剪除：

```
@ 剪除缓冲区尾部的片段界限符 #
g_string_erase(cache, cache->len - symbols->snippet_delimiter->len, -1);
```

```
=> 分段 -> tokens <20>
```

缓冲区尾部剪除界限符后，剩余字符作为片段内容，以类型 OREZ_TEXT 存于记号列表：

```
@ 分段过程所需的辅助函数 # +
static GList *orez_snippet(GList *tokens, GString *cache)
{
        OrezToken *snippet = malloc(sizeof(OrezToken));
        snippet->type = OREZ_TEXT;
        # 获取当前记号对应的行号 -> n @ <19>
        snippet->line_number = n;
        snippet->content = cache;
        tokens = g_list_append(tokens, snippet);
        return tokens;
}
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

上述代码中，缓冲区内的文本（当前记号）在文件中的行号需基于记号列表 tokens 的尾结点对应的行号以及该结点存储的文本行数予以确定：

```
@ 获取当前记号对应的行号 -> n #
size_t n;
GList *last = g_list_last(tokens);
if (last) {
        OrezToken *last_token = last->data;
        n = last_token->line_number;
        for (char *p = last_token->content->str; *p != '\0'; p++) {
                if (*p == '\n') {
                        n++;
                }
        }
} else {
        n = 1;
}
=> 分段过程所需的辅助函数 <19>
=> 分段过程所需的辅助函数 <20>
```

在第一阶段扫描过程中，每次向记号列表中插入类型为 OREZ_TEXT 的记号后，需要向记号列表插入片段界限符：

```
@ 分段过程所需的辅助函数 # +
static GList *orez_snippet_delimiter(GList *tokens, GString *snippet_delimiter)
{
        OrezToken *snippet = malloc(sizeof(OrezToken));
        snippet->type = OREZ_SNIPPET_DELIMITER;
        # 获取当前记号对应的行号 -> n @ <19>
        snippet->line_number = n;
        snippet->content = g_string_new(snippet_delimiter->str);
        tokens = g_list_append(tokens, snippet);
        return tokens;
}
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

基于上述过程及辅助函数，第一阶段扫描的完整过程如下：

```
@ 分段 -> tokens #
# 打开文学化程序源码文件 -> input @ <16>
GList *tokens = NULL;
GString *cache = g_string_new(NULL);
while (1) {
        int status = read_char_from_input(input, cache);
        if (status == EOF) break;
        else {
                bool t = tail_is_snippet_delimiter(cache, symbols->snippet_delimiter);
                if (t) {
                        # 剪除缓冲区尾部的片段界限符 @ <18>
                        tokens = orez_snippet(tokens, cache);
                        tokens = orez_snippet_delimiter(tokens,
                                                        symbols->snippet_delimiter);
                        cache = g_string_new(NULL); /* 刷新 cache */
                }
        }
}
if (cache->len > 0) {
```

```
        tokens = orez_snippet(tokens, cache);
} else g_string_free(cache, TRUE);
fclose(input);
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

以下代码为第一阶段扫描过程的测试代码：

```
@ 分段测试 #
# 分段过程的外境 @ <21>
# 记号 @ <6> <7> <8> <14>
# 分段过程所需的辅助函数 @ <17> <17> <19> <20> <22> <23>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        # 分段 -> tokens @ <20>
        print_tokens(tokens);
        delete_tokens(tokens);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

测试程序用到的头文件如下：

```
@ 分段过程的外境 #
#include <stdio.h>
#include <stdbool.h>
#include <locale.h>
#include <cyaml/cyaml.h>
#include <glib.h>
=> 分段测试 <21>
```

=> 片段名界限符析取过程的外境 <27>

上述测试过程中所用的打印记号列表的函数定义如下

```
@ 分段过程所需的辅助函数 # +
static void print_tokens(GList *tokens)
{
        for (GList *p = g_list_first(tokens);
             p != NULL;
             p = p->next) {
                OrezToken *t = p->data;
                switch (t->type) {
                case OREZ_SNIPPET_DELIMITER:
                        printf("snippet delimiter");
                        break;
                case OREZ_SNIPPET_NAME_DELIMITER:
                        printf("snippet name delimiter");
                        break;
                case OREZ_LANGUAGE_BEGINNING_MARK:
                        printf("language beginning mark");
                        break;
                case OREZ_LANGUAGE_END_MARK:
                        printf("language end mark");
                        break;
                case OREZ_SNIPPET_APPENDING_MARK:
                        printf("snippet appending mark");
                        break;
                case OREZ_SNIPPET_PREPENDING_MARK:
                        printf("snippet prepending mark");
                        break;
                case OREZ_TAG_BEGINNING_MARK:
                        printf("tag beginning mark");
                        break;
                case OREZ_TAG_END_MARK:
                        printf("tag end mark");
                        break;
                case OREZ_SNIPPET_REFERENCE_BEGINNING_MARK:
                        printf("snippet reference beginning mark");
                        break;
                case OREZ_SNIPPET_REFERENCE_END_MARK:
                        printf("snippet reference end mark");
```

```
                break;
        case OREZ_TEXT:
                printf("text");
                break;
        default:
                printf("Illegal token!\n");
        }
        printf(" | line %lu: %s\n", t->line_number, t->content->str);
    }
}
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

delete_tokens 用于释放记号列表占用的内存，其定义如下：

```
@ 分段过程所需的辅助函数 # +
static void delete_tokens(GList *tokens)
{
    for (GList *p = g_list_first(tokens);
         p != NULL;
         p = p->next) {
        OrezToken *t = p->data;
        g_string_free(t->content, TRUE);
        free(t);
    }
    g_list_free(tokens);
}
=> 分段测试 <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

2.4 带名字的片段

第二阶段扫描基于第一阶段扫描产生的记号列表 tokens，对其中 OREZ_TEXT 类型的结点——片段结点进行扫描，目标依然非常简单，仅析取片段名字界限符，每个含有该符号的

文本结点被拆分为三个结点，其类型依序为 OREZ_TEXT，OREZ_SNIPPET_NAME_DELIMITER 和 OREZ_TEXT。

假设片段结点 t，对其内容进行扫描，遇到片段名字界限符，则回溯检测前文是否出现换行符。若前文存在换行符，则将遇到的片段名字界限符是为普通字符。在回溯检测前文是否出现换行符的过程中，需要排查是否为续行符引导的换行情况，将符合该情况的换行符视为普通字符，使其不影响片段名字界限符的判定。上述过程由以下辅助函数实现：

```
@ 片段名界限符析取过程所需的辅助函数 #
<find_name_delimiter>
static char *find_name_delimiter(OrezToken *t,
                                 GString *delimiter,
                                 GString *continuation)
{
        /* 构造字符串区间 [a, b] */
        char *a = t->content->str;
        if (*a == '\n') return NULL;
        /* b 为 NULL 或指向片段名字界限符首字节 */
        char *b = strstr(a, delimiter->str);
        if (!b || b == a) return NULL;
        /* 检测 source 是否含有合法的片段名字界限符 */
        char *p = --b;
        enum {IDLE, LINEBREAK, SUCCESS, FAILURE} state = IDLE;
        while (1) { /* 逆序遍历 [a, b] */
                switch (state) {
                case IDLE:
                        if (*p == '\n') state = LINEBREAK;
                        break;
                case LINEBREAK:
                        if (*p == ' '
                            || *p == '\t'
                            || am_i_here(a, b, &p, "  ", -1)) state = LINEBREAK;
                        else if (am_i_here(a, b, &p, continuation->str, -1)) {
                                state = IDLE;
                        } else state = FAILURE;
                        break;
                default:
                        g_error("Illegal state in line %lu.", t->line_number);
                }
                if (p == a) {
                        /* 字符串逆序遍历到头，此时匹配过程既未成功，
                           亦未失败，则特命其成功 */
```

```
                        state = SUCCESS;
                }
                if (state == FAILURE || state == SUCCESS) break;
                else p--;
        }
        return (state == SUCCESS) ? b + 1 : NULL;
}
```
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>

find_name_delimiter 若发现符合规则的片段名字界限符则返回界限符首字符所在地址，否则返回 NULL。需要注意的是，若界限符之前皆为空白字符，find_name_delimiter 依然会认定为界限符合乎规则，且片段名字为空白字串。

对含有片段名字界限符的记号进行分割，使之裂变为三个记号：

```
@ 析取片段名界限符 -> tokens #
GList *it = g_list_first(tokens); /* 该指针在之后遍历 tokens 的过程中依然会被使用 */
while (1) {
        if (!it) break;
        OrezToken *t = it->data;
        if (t->type == OREZ_TEXT) {
                char *p = find_name_delimiter(t,
                                              symbols->snippet_name_delimiter,
                                              symbols->snippet_name_continuation);
                if (p) {
                        # 基于 p 分割 t，得到
                          snippet_name, snippet_name_delimiter 和 snippet @ <26>
                        /* 删除 it */
                        GList *next = g_list_next(it);
                        tokens = g_list_remove_link(tokens, it);
                        g_string_free(t->content, TRUE);
                        free(t);
                        g_list_free(it);
                        /* 在 prev 之后插入上述新结点 */
                        /* next 为 NULL 时，会向链表尾部插入结点 */
                        tokens = g_list_insert_before(tokens, next, snippet_name);
                        tokens = g_list_insert_before(tokens, next, snippet_name_delimiter);
                        tokens = g_list_insert_before(tokens, next, snippet);
```

```
                        /* 调整迭代指针 */
                        it = next;
                }
        }
        it = g_list_next(it);
}
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

上述对记号 t 的分割过程为

```
@ 基于 p 分割 t, 得到
  snippet_name, snippet_name_delimiter 和 snippet #
/* 提取片段名字并消除（可能存在的）续行符 */
cache = g_string_new(NULL); /* 在第一阶段扫描过程中便已出现 */
for (char *q = t->content->str; q != p; q++) {
        g_string_append_c(cache, *q);
}
g_string_replace(cache, symbols->snippet_name_continuation->str, "", 0);
/* 构建片段名字记号 */
OrezToken *snippet_name = malloc(sizeof(OrezToken));
snippet_name->type = OREZ_TEXT;
snippet_name->line_number = t->line_number;
snippet_name->content = cache;
/* 构建片段名字界限符记号 */
OrezToken *snippet_name_delimiter = malloc(sizeof(OrezToken));
snippet_name_delimiter->type = OREZ_SNIPPET_NAME_DELIMITER;
snippet_name_delimiter->line_number = snippet_name->line_number;
for (char *q = snippet_name->content->str; *q != '\0'; q++) {
        if (*q == '\n') snippet_name_delimiter->line_number++;
}
snippet_name_delimiter->content =
        g_string_new(symbols->snippet_name_delimiter->str);
/* 构建片段内容记号 */
OrezToken *snippet = malloc(sizeof(OrezToken));
snippet->type = OREZ_TEXT;
snippet->line_number = snippet_name_delimiter->line_number;
cache = g_string_new(NULL);
```

```
for (char *q = p + symbols->snippet_name_delimiter->len;
     *q != '\0';
     q++) {
        g_string_append_c(cache, *q);
}
snippet->content = cache;
=> 析取片段名界限符 -> tokens <25>
```

以下代码测试了片段名界限符析取过程：

```
@ 片段名界限符析取测试 #
# 片段名界限符析取过程的外境 @ <27>
# 记号 @ <6> <7> <8> <14>
# 分段过程所需的辅助函数 @ <17> <17> <19> <20> <22> <23>
# 片段名界限符析取过程所需的辅助函数 @ <24>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        # 分段 -> tokens @ <20>
        # 析取片段名界限符 -> tokens @ <25>
        print_tokens(tokens);
        delete_tokens(tokens);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

上述测试过程所需的头文件为

```
@ 片段名界限符析取过程的外境 #
#include <assert.h>
# 分段过程的外境 @ <21>
=> 片段名界限符析取测试 <27>
=> 语言标记和标签引用析取过程的外境 <33>
```

2.5 语言标记和标签引用

经过了前两个阶段的扫描过程的实现，现在对编写词法分析器不再有太多畏惧。第三阶段的扫描略为复杂。

假设记号 t，其类型为 OREZ_TEXT 且其前一个记号类型为 OREZ_SNIPPET_NAME_DELIMITER，则该记号内容的首部可能依序存在编程语言标记、片段运算符、标签引用以及标签，对它们的析取方法相同，皆可通过以下函数提取：

```
@ 语言标记和标签引用析取过程所需的辅助函数 #
static GString *extract_small_block_at_head(OrezToken *t,
                                            GString *beginning_mark,
                                            GString *end_mark)
{
        GString *result = NULL;
        enum {IDLE, MAYBE_MARK, FAILURE, SUCCESS} state = IDLE;
        char *block_beginning = NULL;
        char *block_end = NULL;
        size_t new_line_number = t->line_number;
        /* 构造字符串区间 [a, b] */
        char *a = t->content->str, *b = a + t->content->len - 1;
        if (a == b) return NULL;
        char *p = a;
        while (1) { /* 正序遍历 [a, b] */
                switch (state) {
                case IDLE:
                        if (*p == ' ' || *p == '\t'
                            || am_i_here(a, b, &p, "  ", 1)) {
                                state = IDLE;
                        } else if (*p == '\n') {
                                new_line_number++;
                                state = IDLE;
                        } else if (am_i_here(a, b, &p, beginning_mark->str, 1)) {
                                block_beginning = p - beginning_mark->len + 1;
                                state = MAYBE_MARK;
                        } else state = FAILURE;
                        break;
                case MAYBE_MARK:
                        if (am_i_here(a, b, &p, beginning_mark->str, 1)) {
                                state = FAILURE;
                        } else if (am_i_here(a, b, &p, end_mark->str, 1)) {
                                block_end = p + 1;
```

```
                        state = SUCCESS;
                } else state = MAYBE_MARK;
                break;
        default:
                g_error("Illegal state in line %lu.",
                        t->line_number);
        }
        if (state == SUCCESS) {
                result = g_string_new(NULL);
                for (char *q = block_beginning; q != block_end; q++) {
                        g_string_append_c(result, *q);
                }
                /* 从 t 的内容中删除语言标记 */
                GString *new_content = g_string_new(NULL);
                for (char *q = block_end; *q != '\0'; q++) {
                        g_string_append_c(new_content, *q);
                }
                g_string_free(t->content, TRUE);
                t->content = new_content;
                t->line_number = new_line_number;
                break;
        } else if (state == FAILURE || *p == '\0') break;
        else p++;
    }
    return result;
}
```

=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>

extract_small_block_at_head 返回记号 t 位于首部的块标记，即左闭右开区间 [block_beginning, block_end)，若 t 的首部未出现块标记，则返回 NULL。函数 extract_block_content 可从块标记中提取位于块开始与终止标记之间的内容，其定义为

```
@ 语言标记和标签引用析取过程所需的辅助函数 # +
static GString *extract_block_content(GString *block,
                                      GString *beginning_mark,
                                      GString *end_mark)
{
    GString *content = g_string_new(NULL);
```

```
        size_t a = beginning_mark->len;
        size_t b = block->len - end_mark->len;
        for (size_t i = a; i < b; i++) {
                g_string_append_c(content, *(block->str + i));
        }
        return content;
}
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

设 t 在记号列表 tokens 中对应的结点为 x，提取位于记号内容首部的编程语言标记并在记号列表中为其构建记号的过程如下：

```
@ 语言标记和标签引用析取过程所需的辅助函数 # +
static GList *create_block_token(GList *tokens,
                                 GList *x,
                                 GString *block,
                                 GString *beginning_mark,
                                 OrezTokenType beginning_mark_type,
                                 GString *end_mark,
                                 OrezTokenType end_mark_type)
{
        OrezToken *t = x->data;
        /* 构建块起始记号 */
        OrezToken *beginning = malloc(sizeof(OrezToken));
        beginning->type = beginning_mark_type;
        beginning->line_number = t->line_number;
        beginning->content = g_string_new(beginning_mark->str);
        tokens = g_list_insert_before(tokens, x, beginning);
        /* 构建块内容记号 */
        OrezToken *body = malloc(sizeof(OrezToken));
        body->type = OREZ_TEXT;
        body->line_number = t->line_number;
        body->content = extract_block_content(block, beginning_mark, end_mark);
        tokens = g_list_insert_before(tokens, x, body);
        /* 构建块终止记号 */
        OrezToken *end = malloc(sizeof(OrezToken));
        end->type = end_mark_type;
        end->line_number = t->line_number;
```

```
        end->content = g_string_new(end_mark->str);
        tokens = g_list_insert_before(tokens, x, end);
        return tokens;
}
```
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>

基于上述辅助函数，对于记号 t 及其所属记号结点 it，实现语言标记的提取与记号构建过程如下：

```
@ 语言标记提取与记号构建 #
GString *language = extract_small_block_at_head(t,
                                                symbols->language_beginning_mark,
                                                symbols->language_end_mark);
if (language) {
        language = g_string_ascii_down(language);
        tokens = create_block_token(tokens,
                                    it,
                                    language,
                                    symbols->language_beginning_mark,
                                    OREZ_LANGUAGE_BEGINNING_MARK,
                                    symbols->language_end_mark,
                                    OREZ_LANGUAGE_END_MARK);
        g_string_free(language, TRUE);
}
```
=> 语言标记与标签引用标记的提取与记号构建 <32>
=> 语言标记与标签引用标记的提取与记号构建 <32>

对于记号 t 及其所属记号结点 it，实现标签引用提取与记号构建过程如下：

```
@ 标签引用提取与记号构建 #
GString *tag_reference = extract_small_block_at_head(t,
                                                     symbols->tag_beginning_mark,
                                                     symbols->tag_end_mark);
if (tag_reference) {
        tokens = create_block_token(tokens,
                                    it,
                                    tag_reference,
                                    symbols->tag_beginning_mark,
                                    OREZ_TAG_BEGINNING_MARK,
```

```
                                        symbols->tag_end_mark,
                                        OREZ_TAG_END_MARK);
        g_string_free(tag_reference, TRUE);
}
=> 语言标记与标签引用标记的提取与记号构建 <32>
=> 语言标记与标签引用标记的提取与记号构建 <32>
```

由于 Orez 的语法仅要求语言标记与标签引用标记仅须相邻，但对次序没有要求，故而语言标记和标签引用标记的提取与记号构建过程需要重复执行两次方能满足该语法，即

```
@ 语言标记与标签引用标记的提取与记号构建 #
do {
        # 语言标记提取与记号构建 @ <31>
        # 标签引用提取与记号构建 @ <31>
} while (0);
do {
        # 语言标记提取与记号构建 @ <31>
        # 标签引用提取与记号构建 @ <31>
} while (0);
=> 析取语言标记和标签引用 -> tokens <32>
```

语言标记和标签引用的析取过程的实现如下：

```
@ 析取语言标记和标签引用 -> tokens #
it = g_list_first(tokens); /* 该指针在之后遍历 tokens 的过程中依然会被使用 */
while (1) {
        if (!it) break;
        OrezToken *t = it->data;
        if (t->type == OREZ_TEXT) {
                GList *prev = g_list_previous(it);
                if (prev) {
                        OrezToken *a = prev->data;
                        if (a->type == OREZ_SNIPPET_NAME_DELIMITER) {
                                # 语言标记与标签引用标记的提取与记号构建 @ <32>
                        }
                }
        }
        it = g_list_next(it);
}
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
```

```
=> 词法分析器 <44>
```

以下代码用于测试语言标记和标签引用析取过程：

```
@ 语言标记和标签引用析取测试 #
# 语言标记和标签引用析取过程的外境 @ <33>
# 记号 @ <6> <7> <8> <14>
# 分段过程所需的辅助函数 @ <17> <17> <19> <20> <22> <23>
# 片段名界限符析取过程所需的辅助函数 @ <24>
# 语言标记和标签引用析取过程所需的辅助函数 @ <28> <29> <30>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        # 分段 -> tokens @ <20>
        # 析取片段名界限符 -> tokens @ <25>
        # 析取语言标记和标签引用 -> tokens @ <32>
        print_tokens(tokens);
        delete_tokens(tokens);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

上述测试过程所需的头文件为

```
@ 语言标记和标签引用析取过程的外境 #
# 片段名界限符析取过程的外境 @ <27>
=> 语言标记和标签引用析取测试 <33>
=> 片段运算符和标签的析取过程的外境 <38>
```

2.6 片段运算符和标签

在扫描过程的第四阶段，析取片段运算符和标签。假设记号 t，若其类型为 OREZ_TEXT 且其前一个记号的类型为以下记号之一：

- OREZ_SNIPPET_NAME_DELIMITER
- OREZ_LANGUAGE_END_MARK
- OREZ_TAG_END_MARK

则析取 t 首部可能出现的片段追加运算符，过程如下：

```
@ 片段运算符和标签析取过程所需的辅助函数 #
static GString *extract_operator(OrezToken *t, GString *operator)
{
        GString *result = NULL;
        /* 构造字符串区间 [a, b) */
        char *a = t->content->str;
        char *b = strstr(a, operator->str);
        if (!b) return result;
        bool is_legal = true;
        if (b == t->content->str) ;
        else {
                enum {IDLE, FAILURE} state = IDLE;
                char *p = b - 1;
                /* 以 p 逆序遍历 [a, b) */
                while (1) {
                        switch (state) {
                        case IDLE:
                                if (*p == ' '
                                    || *p == '\t'
                                    || *p == '\n'
                                    || am_i_here(a, b - 1, &p, "  ", -1)) ;
                                else state = FAILURE;
                                break;
                        default:
                                g_error("Illegal state in line %lu.",
                                        t->line_number);
                        }
                        if (p == a) break;
                        else if (state == FAILURE) {
                                is_legal = false;
                                break;
                        } else p--;
                }
        }
        if (is_legal) {
```

```
                /* 从 t 中删除片段追加运算符及其之前的空白字符 */
                GString *new_content = g_string_new(NULL);
                size_t new_line_number = t->line_number;
                for (char *p = a; p != b; p++) {
                        if (*p == '\n') new_line_number++;
                }
                for (char *p = b + operator->len; *p != '\0'; p++) {
                        g_string_append_c(new_content, *p);
                }
                g_string_free(t->content, TRUE);
                t->content = new_content;
                t->line_number = new_line_number;
                /* 构造片段追加运算符副本 */
                result = g_string_new(operator->str);
        }
        return result;
}
```
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>

设记号 t 在记号列表 tokens 中所属结点为 x，以下代码可为析取的片段追加运算符构建记号：

```
@ 片段运算符和标签析取过程所需的辅助函数 # +
static GList *create_operator_token(GList *tokens,
                                    GList *x,
                                    GString *operator,
                                    OrezTokenType operator_type)
{
        OrezToken *t = x->data;
        OrezToken *a = malloc(sizeof(OrezToken));
        a->type = operator_type;
        a->line_number = t->line_number;
        a->content = operator;
        tokens = g_list_insert_before(tokens, x, a);
        return tokens;
}
```
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>

对记号列表 tokens 中的结点 it 所包含的记号 t 中的片段追加运算符的析取及其记号构建过程如下：

```
@ 析取片段追加运算符并为其构建记号 #
GString *appending_mark = extract_operator(t, symbols->snippet_appending_mark);
if (appending_mark) {
        tokens = create_operator_token(tokens,
                                       it,
                                       appending_mark,
                                       OREZ_SNIPPET_APPENDING_MARK);
} else {
        GString *prepending_mark
                = extract_operator(t, symbols->snippet_prepending_mark);
        if (prepending_mark) {
                tokens = create_operator_token(tokens,
                                               it,
                                               prepending_mark,
                                               OREZ_SNIPPET_PREPENDING_MARK);
        }
}
=> 析取片段运算符和标签 -> tokens <37>
```

片段追加运算符析取过程结束后，对可能存在的标签进行析取，其过程与上一节中标签引用的析取过程相同。对记号列表 tokens 中的结点 it 所包含的记号 t 中的标签析取及其记号构建过程如下：

```
@ 析取标签并为其构建记号 #
GString *tag_mark
        = extract_small_block_at_head(t,
                                      symbols->tag_beginning_mark,
                                      symbols->tag_end_mark);
if (tag_mark) {
        tokens = create_block_token(tokens,
                                    it,
                                    tag_mark,
                                    symbols->tag_beginning_mark,
                                    OREZ_TAG_BEGINNING_MARK,
                                    symbols->tag_end_mark,
                                    OREZ_TAG_END_MARK);
        g_string_free(tag_mark, TRUE);
}
=> 析取片段运算符和标签 -> tokens <37>
```

综上，第四阶段扫描过程实现如下：

```
@ 析取片段运算符和标签 -> tokens #
it = g_list_first(tokens);
while (1) {
        if (!it) break;
        OrezToken *t = it->data;
        if (t->type == OREZ_TEXT) {
                GList *prev = g_list_previous(it);
                if (prev) {
                        OrezToken *a = prev->data;
                        if (a->type == OREZ_SNIPPET_NAME_DELIMITER
                            || a->type == OREZ_LANGUAGE_END_MARK
                            || a->type == OREZ_TAG_END_MARK) {
                                # 析取片段追加运算符并为其构建记号 @ <36>
                                # 析取标签并为其构建记号 @ <36>
                        }
                }
        }
        it = g_list_next(it);
}
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

以下代码用于测试片段运算符和标签的析取过程：

```
@ 片段运算符和标签析取测试 #
# 片段运算符和标签的析取过程的外境 @ <38>
# 记号 @ <6> <7> <8> <14>
# 分段过程所需的辅助函数 @ <17> <17> <19> <20> <22> <23>
# 片段名界限符析取过程所需的辅助函数 @ <24>
# 语言标记和标签引用析取过程所需的辅助函数 @ <28> <29> <30>
# 片段运算符和标签析取过程所需的辅助函数 @ <34> <35>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
```

```
        # 构造符号集 -> symbols @ <15>
        # 分段 -> tokens @ <20>
        # 析取片段名界限符 -> tokens @ <25>
        # 析取语言标记和标签引用 -> tokens @ <32>
        # 析取片段运算符和标签 -> tokens @ <37>
        print_tokens(tokens);
        delete_tokens(tokens);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

上述代码所需的头文件为

```
@ 片段运算符和标签的析取过程的外境 #
# 语言标记和标签引用析取过程的外境 @ <33>
=> 片段运算符和标签析取测试 <37>
=> 片段引用析取过程的外境 <44>
```

2.7 片段引用

设记号 t 类型为 OREZ_TEXT，在记号列表中属于结点 x，若其前结点所含记号为以下类型之一：

- OREZ_SNIPPET_NAME_DELIMITER
- OREZ_LANGUAGE_END_MARK
- OREZ_SNIPPET_APPENDING_MARK
- OREZ_SNIPPET_PREPENDING_MARK
- OREZ_TAG_END_MARK
- OREZ_LANGUAGE_END_MARK

则 t 中可能含有片段引用，需对其予以析取。

片段引用析取过程的关键之处在于从 t 的内容中逐一析取片段引用标记。辅助函数 find_snippet_reference 可返回 t 中第一个片段引用标记（假如它存在），其定义如下：

```
@ 片段引用析取过程所需的辅助函数 #
<find_snippet_reference>
typedef struct {
        void *a;
        void *b;
} OrezPair;
static OrezPair *find_snippet_reference(GString *content,
```

```
                                        GString *reference_beginning_mark,
                                        GString *reference_end_mark,
                                        GString *name_continuation)
{
        char *a = content->str, *begin = NULL, *end = NULL;
        enum {IDLE, MAYBE_LINEBREAK, FAILURE} state;
        while (1) {
                begin = strstr(a, reference_beginning_mark->str);
                if (!begin) return NULL;
                end = strstr(a, reference_end_mark->str);
                if (!end || begin >= end) return NULL;
                /*区间 [q, p] 是片段名 */
                char *p = end - 1;
                char *q = begin + reference_beginning_mark->len;
                /* 逆序遍历 [q, end), 校验片段名是否合法 */
                bool legal = TRUE;
                state = IDLE;
                while (1) {
                        switch (state) {
                        case IDLE:
                                if (*p == '\n') state = MAYBE_LINEBREAK;
                                break;
                        case MAYBE_LINEBREAK:
                                if (*p == ' '
                                    || am_i_here(q, end - 1, &p, "  ", -1)) ;
                                else if (am_i_here(q, end - 1, &p,
                                                   name_continuation->str, -1)) {
                                        state = IDLE;
                                } else state = FAILURE;
                                break;
                        default:
                                g_error("Illegal state in <<< %s >>>.", a);
                        }
                        if (state == FAILURE) {
                                legal = FALSE;
                                break;
                        } else if (p == q) break;
                        else p--;
                }
                if (legal) break;
                else a = q;
```

```
        }
        if (begin && end) {
                /* 前闭后开区间 */
                OrezPair *result = malloc(sizeof(OrezPair));
                result->a = begin;
                result->b = end + reference_end_mark->len;
                return result;
        } else return NULL;
}
```

=> 片段引用析取测试 <43>

=> 词法分析器 <44>

在能够析取 t∈x 的内容中片段引用标记后，便可基于析取结果，将 t 的内容分割为五个部分——片段引用之前的内容、片段引用起始标记、片段名称、片段引用结束标记以及片段引用之后的内容，在记号列表中取代 t。该过程由以下辅助函数实现：

```
@ 片段引用析取过程所需的辅助函数 # +
static GList *split_snippet(GList *tokens,
                            GList *x,
                            OrezPair *snippet_reference,
                            GString *snippet_name_continuation,
                            GString *snippet_reference_beginning_mark,
                            GString *snippet_reference_end_mark)
{
        OrezToken *t = x->data;
        size_t line_number = t->line_number;
        /* 片段引用之前的内容 */
        GString *a = g_string_new(NULL);
        for (char *p = t->content->str; p != snippet_reference->a; p++) {
                g_string_append_c(a, *p);
                if (*p == '\n') line_number++;
        }
        OrezToken *t_a = malloc(sizeof(OrezToken));
        t_a->type = OREZ_TEXT;
        t_a->line_number = t->line_number;
        t_a->content = a;
        /* 片段引用起始标记 */
        GString *b = g_string_new(snippet_reference_beginning_mark->str);
        OrezToken *t_b = malloc(sizeof(OrezToken));
        t_b->type = OREZ_SNIPPET_REFERENCE_BEGINNING_MARK;
        t_b->line_number = line_number;
```

```
t_b->content = b;
/* 片段名称 */
GString *c = g_string_new(NULL);
char *left = (char *)(snippet_reference->a)
        + snippet_reference_beginning_mark->len;
char *right = (char *)(snippet_reference->b)
        - snippet_reference_end_mark->len;
for (char *p = left; p != right; p++) {
        g_string_append_c(c, *p);
        if (*p == '\n') line_number++;
}
g_string_replace(c, snippet_name_continuation->str, "", 0);
OrezToken *t_c = malloc(sizeof(OrezToken));
t_c->type = OREZ_TEXT;
t_c->line_number = t_b->line_number;
t_c->content = c;
/* 片段引用结束标记 */
GString *d = g_string_new(snippet_reference_end_mark->str);
OrezToken *t_d = malloc(sizeof(OrezToken));
t_d->type = OREZ_SNIPPET_REFERENCE_END_MARK;
t_d->line_number = line_number;
t_d->content = d;
/* 片段引用之后的内容 */
GString *e = g_string_new(NULL);
bool after_reference = TRUE;
for (char *p = snippet_reference->b; *p != '\0'; p++) {
        if (after_reference) {
                 /* 忽略与片段引用之后与片段引用终止符同一行的文本 */
                if (*p == '\n') {
                        after_reference = FALSE;
                        g_string_append_c(e, *p);
                }
        } else g_string_append_c(e, *p);
}
/* 更新记号列表 */
tokens = g_list_insert_before(tokens, x, t_a);
tokens = g_list_insert_before(tokens, x, t_b);
tokens = g_list_insert_before(tokens, x, t_c);
tokens = g_list_insert_before(tokens, x, t_d);
g_string_free(t->content, TRUE);
t->content = e;
```

```
        t->line_number = line_number;
        return tokens;
}
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

现在基于上述辅助函数便可实现第五阶段扫描：

```
@ 析取片段引用 -> tokens #
it = g_list_first(tokens);
while (1) {
        if (!it) break;
        OrezToken *t = it->data;
        if (t->type == OREZ_TEXT) {
                GList *prev = g_list_previous(it);
                if (prev) {
                        OrezToken *t_prev = prev->data;
                        if (t_prev->type == OREZ_SNIPPET_NAME_DELIMITER
                            || t_prev->type == OREZ_LANGUAGE_END_MARK
                            || t_prev->type == OREZ_SNIPPET_APPENDING_MARK
                            || t_prev->type == OREZ_SNIPPET_PREPENDING_MARK
                            || t_prev->type == OREZ_TAG_END_MARK) {
                                while (1) {
                                        # 构造片段引用 -> tokens @ <42>
                                }
                        }
                }
        }
        it = g_list_next(it);
}
/* 为记号列表尾部追加一个片段界限符，以便于后续程序处理 */
GList *last_it = g_list_last(tokens);
OrezToken *last_token = last_it->data;
if (last_token->type != OREZ_SNIPPET_DELIMITER) {
        tokens = orez_snippet_delimiter(tokens, symbols->snippet_delimiter);
}
=> 片段引用析取测试 <43>
=> 词法分析器 <44>
```

对 t 中可能存在的片段引用析取过程如下：

```
@ 构造片段引用 -> tokens #
```

```
OrezPair *snippet_reference
        = find_snippet_reference(t->content,
                                 symbols->snippet_reference_beginning_mark,
                                 symbols->snippet_reference_end_mark,
                                 symbols->snippet_name_continuation);
if (snippet_reference) {
        tokens = split_snippet(tokens,
                               it,
                               snippet_reference,
                               symbols->snippet_name_continuation,
                               symbols->snippet_reference_beginning_mark,
                               symbols->snippet_reference_end_mark);
        free(snippet_reference);
} else break;
=> 析取片段引用 -> tokens <42>
```

以下为片段引用析取过程的测试代码：

```
@ 片段引用析取测试 #
# 片段引用析取过程的外境 @ <44>
# 记号 @ <6> <7> <8> <14>
# 分段过程所需的辅助函数 @ <17> <17> <19> <20> <22> <23>
# 片段名界限符析取过程所需的辅助函数 @ <24>
# 语言标记和标签引用析取过程所需的辅助函数 @ <28> <29> <30>
# 片段运算符和标签析取过程所需的辅助函数 @ <34> <35>
# 片段引用析取过程所需的辅助函数 @ <38> <40>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        # 分段 -> tokens @ <20>
        # 析取片段名界限符 -> tokens @ <25>
        # 析取语言标记和标签引用 -> tokens @ <32>
        # 析取片段运算符和标签 -> tokens @ <37>
        # 析取片段引用 -> tokens @ <42>
        print_tokens(tokens);
```

```
        delete_tokens(tokens);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

上述测试过程所需的头文件为

```
@ 片段引用析取过程的外境 #
# 片段运算符和标签的析取过程的外境 @ <38>
=> 片段引用析取测试 <43>
=> 词法分析过程的外境 <45>
```

2.8 总结

现在，将词法分析的各阶段合并为一个完整的过程：

```
@ 词法分析器 #
# 分段过程所需的辅助函数 @ <17> <17> <19> <20> <22> <23>
# 片段名界限符析取过程所需的辅助函数 @ <24>
# 语言标记和标签引用析取过程所需的辅助函数 @ <28> <29> <30>
# 片段运算符和标签析取过程所需的辅助函数 @ <34> <35>
# 片段引用析取过程所需的辅助函数 @ <38> <40>
static GList *orez_lexer(const char *input_file_name, OrezSymbols *symbols)
{
        # 分段 -> tokens @ <20>
        # 析取片段名界限符 -> tokens @ <25>
        # 析取语言标记和标签引用 -> tokens @ <32>
        # 析取片段运算符和标签 -> tokens @ <37>
        # 析取片段引用 -> tokens @ <42>
        return tokens;
}
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
```

```
=> orez.c <96>
```

orez_lexer 的用法示例如下：

```
@ 词法分析示例 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        print_tokens(tokens);
        delete_tokens(tokens);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}

@ 词法分析过程的外境 #
# 片段引用析取过程的外境 @ <44>
=> 词法分析示例 <45>
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

3 语法分析

基于词法分析得到的记号列表构造语法树。语法结点的数据结构如下：

```
@ 语法结点 #
typedef struct {
        OrezSyntaxType type;
        GList *tokens;
} OrezSyntax;
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

语法结点的类型如下

```
@ 语法结点 # ^+
typedef enum {
        OREZ_SNIPPET,
        OREZ_SNIPPET_WITH_NAME,
        OREZ_SNIPPET_NAME,
        OREZ_SNIPPET_LANGUAGE,
        OREZ_SNIPPET_APPENDING_OPERATOR,
        OREZ_SNIPPET_PREPENDING_OPERATOR,
        OREZ_SNIPPET_TAG_REFERENCE,
        OREZ_SNIPPET_TAG,
        OREZ_SNIPPET_CONTENT,
        OREZ_SNIPPET_REFERENCE,
        OREZ_SNIPPET_TEXT
} OrezSyntaxType;
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
```

=> orez.c <96>

语法树的构造过程分为三个阶段。第一阶段，构造无名片段和有名片段结点。第二阶段，对有名片段进行结构化处理。第三阶段，对有名片段的内容进行结构化处理。语法树的根结点是无数据的。

3.1 片段

根据记号列表中的片段界限符以及片段名称界限符便可区分两类片段形式，并在语法树中为其构造结点。遍历记号列表 tokens 构造无名片段和有名片段结点的过程如下：

```
@ 语法分析过程的辅助函数 #
static GNode *syntax_tree_init(GList *tokens)
{
        GNode *root = g_node_new(NULL);
        for (GList *it = g_list_first(tokens);
             it != NULL;
             it = g_list_next(it)) {
                OrezToken *t = it->data;
                if (t->type == OREZ_SNIPPET_DELIMITER) {
                        GList *prev = g_list_previous(it);
                        if (prev) {
                                OrezSyntax *snippet = malloc(sizeof(OrezSyntax));
                                /* 默认类型，后续会确定它是否为有名片段 */
                                snippet->type = OREZ_SNIPPET;
                                snippet->tokens = NULL;
                                # 为片段收集记号并确定片段类型 @ <48>
                                /* 之前在记号列表尾部强行插入了一个片段界限记号，
                                   它可能会导致语法结点的内容为空，故而需要予以忽略 */
                                if (snippet->tokens) {
                                        g_node_append_data(root, snippet);
                                } else free(snippet);
                        }
                }
        }
        /* 释放片段界限记号 */
        for (GList *it = g_list_first(tokens);
             it != NULL;
             it = g_list_next(it)) {
                OrezToken *t = it->data;
                if (t->type == OREZ_SNIPPET_DELIMITER) {
```

```
                        g_string_free(t->content, TRUE);
                        free(t);
                }
        }
        /* 销毁记号列表，它已完成使命 */
        g_list_free(tokens);
        return root;
}
```
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器 <59>

构造片段所包含的记号子集以及片段类型的代码如下：

```
@ 为片段收集记号并确定片段类型 #
GList *a = prev;
while (1)
{
        if (a) {
                OrezToken *t_a = a->data;
                if (t_a->type == OREZ_SNIPPET_DELIMITER) break;
                else {
                        if (t_a->type == OREZ_SNIPPET_NAME_DELIMITER) {
                                snippet->type = OREZ_SNIPPET_WITH_NAME;
                        }
                        snippet->tokens = g_list_prepend(snippet->tokens, t_a);
                }
        } else break;
        a = g_list_previous(a);
}
```
=> 语法分析过程的辅助函数 <47>

以下为语法树片段结点构造过程测试代码：

```
@ 片段结点构造测试 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析过程的辅助函数 @ <47> <49> <51> <51> <53> <57>
int main(void)
```

```
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = syntax_tree_init(tokens);
        print_syntax_tree(root);
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

辅助函数 print_syntax_tree 以深度优先的方式遍历语法树，并打印各结点的信息，其定义如下：

```
@ 语法分析过程的辅助函数 # +
static size_t first_token_line_number(GNode *x)
{
        size_t line_number = 0;
        OrezSyntax *a = x->data;
        if (a->tokens) {
                OrezToken *t = g_list_first(a->tokens)->data;
                line_number = t->line_number;
        } else {
                line_number = first_token_line_number(x->children);
        }
        return line_number;
}
static void print_syntax_tree(GNode *root)
{
        if (root) {
                if (root->data) {
                        OrezSyntax *a = root->data;
                        switch (a->type) {
                        case OREZ_SNIPPET:
                                printf("<snippet>\n");
```

```
                        break;
                case OREZ_SNIPPET_WITH_NAME:
                        printf("<snippet with name>\n");
                        break;
                case OREZ_SNIPPET_NAME:
                        printf("<name>\n");
                        break;
                case OREZ_SNIPPET_LANGUAGE:
                        printf("<language>\n");
                        break;
                case OREZ_SNIPPET_APPENDING_OPERATOR:
                        printf("<snippet appending operator>\n");
                        break;
                case OREZ_SNIPPET_PREPENDING_OPERATOR:
                        printf("<snippet prepending operator>\n");
                        break;
                case OREZ_SNIPPET_TAG_REFERENCE:
                        printf("<snippet tag reference>\n");
                        break;
                case OREZ_SNIPPET_TAG:
                        printf("<snippet tag>\n");
                        break;
                case OREZ_SNIPPET_CONTENT:
                        printf("<snippet content>\n");
                        break;
                case OREZ_SNIPPET_REFERENCE:
                        printf("<snippet reference>\n");
                        break;
                case OREZ_SNIPPET_TEXT:
                        printf("<snippet text>\n");
                        break;
                default:
                        g_error("Illegal syntax in %lu.\n",
                                first_token_line_number(root));
                }
                if (a->tokens) print_tokens(a->tokens);
        }
        for (GNode *p = root->children; p != NULL; p = p->next) {
                print_syntax_tree(p);
        }
}
```

```
}
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器 <59>
```

delete_syntax_tree 用于释放语法树占用的内存，其定义如下：

```
@ 语法分析过程的辅助函数 # +
static void delete_syntax_tree_body(GNode *root)
{
        if (root) {
                if (root->data) {
                        OrezSyntax *a = root->data;
                        if (a->tokens) delete_tokens(a->tokens);
                        free(a);
                }
                for (GNode *p = root->children; p != NULL; p = p->next) {
                        delete_syntax_tree_body(p);
                }
        }
}
static void delete_syntax_tree(GNode *root)
{
        delete_syntax_tree_body(root);
        g_node_destroy(root);
}
=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器 <59>
```

3.2 有名片段的子结点集

假设语法树的第二层结点中，结点 x 包含的片段数据 snippet 为有名片段，则需基于 snippet 的记号序列构造语法结点，作为 x 的子结点，且在该过程完成后，释放 snippet 的记号序列。设语法树结点 x，若其类型为 OREZ_SNIPPET_WITH_NAME，则 create_snippet_body 函数可为其构建子结点集：

```
@ 语法分析过程的辅助函数 # +
<create_child_nodes_of_snippet_with_name>
static void create_child_nodes_of_snippet_with_name(GNode *x)
```

```
{
        OrezSyntax *snippet = x->data;
        GList *it = g_list_first(snippet->tokens);
        OrezToken *token = it->data;
        /* 构造片段名字结点 */
        OrezSyntax *snippet_name = malloc(sizeof(OrezSyntax));
        snippet_name->type = OREZ_SNIPPET_NAME;
        snippet_name->tokens = g_list_append(NULL, token);
        g_node_append_data(x, snippet_name);
        /* 跳过片段名字界限符 */
        it = g_list_next(it);
        token = it->data;
        if (token->type != OREZ_SNIPPET_NAME_DELIMITER) {
                g_error("Line %lu: Illegal snippet name.", token->line_number);
        }
        it = g_list_next(it);
        /* 构造可能存在的语言和标签引用结点,
           之所以重复两次,是因 Orez 语法未区分语言和标签引用的先后 */
        it = create_language(x, it);
        it = create_tag_reference(x, it);
        it = create_language(x, it);
        it = create_tag_reference(x, it);
        /* 构造可能存在的运算符结点 */
        token = it->data;
        if (token->type == OREZ_SNIPPET_APPENDING_MARK
            || token->type == OREZ_SNIPPET_PREPENDING_MARK) {
                OrezSyntax *operator = malloc(sizeof(OrezSyntax));
                if (token->type == OREZ_SNIPPET_APPENDING_MARK) {
                        operator->type = OREZ_SNIPPET_APPENDING_OPERATOR;
                } else {
                        operator->type = OREZ_SNIPPET_PREPENDING_OPERATOR;
                }
                operator->tokens = g_list_append(NULL, token);
                g_node_append_data(x, operator);
                it = g_list_next(it);
        }
        /* 构造可能存在的标签结点 */
        it = create_tag(x, it);
        /* 构造有名片段的内容结点 */
        OrezSyntax *content = malloc(sizeof(OrezSyntax));
        content->type = OREZ_SNIPPET_CONTENT;
```

```
        content->tokens = NULL;
        for (GList *it_a = it; it_a != NULL; it_a = g_list_next(it_a)) {
                token = it_a->data;
                if (token->type == OREZ_SNIPPET_REFERENCE_BEGINNING_MARK
                    || token->type == OREZ_SNIPPET_REFERENCE_END_MARK
                    || token->type == OREZ_TEXT) {
                        content->tokens = g_list_append(content->tokens, token);
                } else {
                        g_error("Line %lu: Illegal snippet.",
                                token->line_number);
                }
        }
        g_node_append_data(x, content);
        /* 释放一些未进入语法树的记号 */
        for (it = g_list_first(snippet->tokens);
             it != NULL;
             it = g_list_next(it)) {
                token = it->data;
                if (token->type == OREZ_SNIPPET_NAME_DELIMITER
                    || token->type == OREZ_LANGUAGE_BEGINNING_MARK
                    || token->type == OREZ_LANGUAGE_END_MARK
                    || token->type == OREZ_TAG_BEGINNING_MARK
                    || token->type == OREZ_TAG_END_MARK) {
                        g_string_free(token->content, TRUE);
                        free(token);
                }
        }
        /* 释放 snippet 的词法表，因为其内容已全部转移至其子结点中了 */
        g_list_free(snippet->tokens);
        snippet->tokens = NULL;
}
```

=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器 <59>

create_children_of_snippet_with_name 函数所需的辅助函数如下：

```
@ 语法分析过程的辅助函数 # <create_child_nodes_of_snippet_with_name> ^+
static GList *create_language(GNode *x, GList *it)
{
```

```
        OrezToken *token = it->data;
        if (token->type == OREZ_LANGUAGE_BEGINNING_MARK) {
                OrezSyntax *language = malloc(sizeof(OrezSyntax));
                it = g_list_next(it);
                token = it->data;
                language->type = OREZ_SNIPPET_LANGUAGE;
                language->tokens = g_list_append(NULL, token);
                g_node_append_data(x, language);
                it = g_list_next(it);
                token = it->data;
                if (token->type != OREZ_LANGUAGE_END_MARK) {
                        g_error("Line %lu: Illegal language mark!",
                                token->line_number);
                }
                it = g_list_next(it);
        }
        return it;
}
static GList *create_tag_reference(GNode *x, GList *it)
{
        OrezToken *token = it->data;
        if (token->type == OREZ_TAG_BEGINNING_MARK) {
                bool is_tag_ref = false;
                for (GList *it_a = it;
                     it_a != NULL;
                     it_a = g_list_next(it_a)) {
                   OrezToken *t = it_a->data;
                   if (t->type == OREZ_SNIPPET_APPENDING_MARK
                       || t->type == OREZ_SNIPPET_PREPENDING_MARK) {
                           is_tag_ref = true;
                           break;
                   }
                }
                if (is_tag_ref) {
                        OrezSyntax *tag_ref = malloc(sizeof(OrezSyntax));
                        it = g_list_next(it);
                        token = it->data;
                        if (token->type != OREZ_TEXT) {
                                g_error("Line %lu: Illegal tag reference.",
                                        token->line_number);
                        }
```

```
                        tag_ref->type = OREZ_SNIPPET_TAG_REFERENCE;
                        tag_ref->tokens = g_list_append(NULL, token);
                        g_node_append_data(x, tag_ref);
                        it = g_list_next(it);
                        token = it->data;
                        if (token->type != OREZ_TAG_END_MARK) {
                                g_error("Line %lu: Illegal tag reference.",
                                        token->line_number);
                        }
                        it = g_list_next(it);
                }
        }
        return it;
}
static GList *create_tag(GNode *x, GList *it)
{
        OrezToken *token = it->data;
        if (token->type == OREZ_TAG_BEGINNING_MARK) {
                OrezSyntax *tag = malloc(sizeof(OrezSyntax));
                it = g_list_next(it);
                token = it->data;
                if (token->type != OREZ_TEXT) {
                        g_error("Line %lu: Illegal tag.",
                                token->line_number);
                }
                tag->type = OREZ_SNIPPET_TAG;
                tag->tokens = g_list_append(NULL, token);
                g_node_append_data(x, tag);
                it = g_list_next(it);
                token = it->data;
                if (token->type != OREZ_TAG_END_MARK) {
                        g_error("Line %lu: Illegal tag reference.",
                                token->line_number);
                }
                it = g_list_next(it);
        }
        return it;
}
```

=> 片段结点构造测试 <48>

=> 有名片段的子结点集构造过程测试 <56>

=> 有名片段内容结点的子结点集构造过程测试 <59>

```
=> 语法分析器 <59>
```

设语法树根结点微 root，为语法树中第二层所有类型为 OREZ_SNIPPET_WITH_NAME 的片段结点构造子结点集的代码如下：

```
@ 为有名片段构造子结点集 #
for (GNode *p = root->children; p != NULL; p = p->next) {
        OrezSyntax *snippet = p->data;
        if (snippet->type == OREZ_SNIPPET_WITH_NAME) {
                create_child_nodes_of_snippet_with_name(p);
        }
}
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器 <59>
```

以下是有名片段的子结点集构造过程测试代码：

```
@ 有名片段的子结点集构造过程测试 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析过程的辅助函数 @ <47> <49> <51> <51> <53> <57>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = syntax_tree_init(tokens);
        # 为有名片段构造子结点集 @ <56>
        print_syntax_tree(root);
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

3.3 有名片段的内容

有名片段的内容只包含两种语法结点类型：

- OREZ_SNIPPET_REFERENCE
- OREZ_SNIPPET_TEXT

假设语法树的第三层结点[4]中，结点 x 的类型为 OREZ_SNIPPET_CONTENT，需为其构造子结点，过程如下：

```
@ 语法分析过程的辅助函数 # +
static void create_child_nodes_of_snippet_content(GNode *x)
{
        OrezSyntax *content = x->data;
        GList *it = g_list_first(content->tokens);
        while (1) {
                if (!it) break;
                OrezToken *token = it->data;
                if (token->type == OREZ_SNIPPET_REFERENCE_BEGINNING_MARK) {
                        /* 构造片段引用 */
                        it = g_list_next(it);
                        token = it->data;
                        if (token->type != OREZ_TEXT) {
                                g_error("Line %lu: Illegal snippet reference.",
                                        token->line_number);
                        }
                        OrezSyntax *reference = malloc(sizeof(OrezSyntax));
                        reference->type = OREZ_SNIPPET_REFERENCE;
                        reference->tokens = g_list_append(NULL, token);
                        g_node_append_data(x, reference);
                        /* 验证片段引用的语法合法性 */
                        it = g_list_next(it);
                        token = it->data;
                        if (token->type != OREZ_SNIPPET_REFERENCE_END_MARK) {
                                g_error("Line %lu: Illegal snippet reference.",
                                        token->line_number);
                        }
                } else if (token->type == OREZ_TEXT) {
                        OrezSyntax *text = malloc(sizeof(OrezSyntax));
                        text->type = OREZ_SNIPPET_TEXT;
```

[4] 语法树的根结点为第 1 层。

```
                        text->tokens = g_list_append(NULL, token);
                        g_node_append_data(x, text);
                } else {
                        g_error("Line %lu: Illegal snippet.",
                                token->line_number);
                }
                it = g_list_next(it);
        }
        /* 释放 content->tokens */
        for (it = g_list_first(content->tokens);
             it != NULL;
             it = g_list_next(it)) {
                OrezToken *token = it->data;
                if (token->type == OREZ_SNIPPET_REFERENCE_BEGINNING_MARK
                    || token->type == OREZ_SNIPPET_REFERENCE_END_MARK) {
                        g_string_free(token->content, TRUE);
                        free(token);
                }
        }
        g_list_free(content->tokens);
        content->tokens = NULL;
}
```

=> 片段结点构造测试 <48>
=> 有名片段的子结点集构造过程测试 <56>
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器 <59>

设语法树根结点微 root，为语法树中第三层所有类型为 OREZ_SNIPPET_CONTENT 的结点构造子结点集的代码如下：

```
@ 为有名片段的内容结点构造子结点集 #
for (GNode *p = root->children; p != NULL; p = p->next) {
        OrezSyntax *a = p->data;
        if (a->type == OREZ_SNIPPET_WITH_NAME) {
                for (GNode *q = p->children; q != NULL; q = q->next) {
                        OrezSyntax *b = q->data;
                        if (b->type == OREZ_SNIPPET_CONTENT) {
                                create_child_nodes_of_snippet_content(q);
                        }
                }
        }
```

```
}
=> 有名片段内容结点的子结点集构造过程测试 <59>
=> 语法分析器 <59>
```

以下是有名片段内容结点的子结点集构造过程测试代码：

```
@ 有名片段内容结点的子结点集构造过程测试 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析过程的辅助函数 @ <47> <49> <51> <51> <53> <57>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = syntax_tree_init(tokens);
        # 为有名片段构造子结点集 @ <56>
        # 为有名片段的内容结点构造子结点集 @ <58>
        print_syntax_tree(root);
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

3.4 总结

```
@ 语法分析器 #
# 语法分析过程的辅助函数 @ <47> <49> <51> <51> <53> <57>
static GNode *orez_parser(GList *tokens)
{
        GNode *root = syntax_tree_init(tokens);
        # 为有名片段构造子结点集 @ <56>
        # 为有名片段的内容结点构造子结点集 @ <58>
```

```
        return root;
}
=> 语法分析器用法示例 <60>
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>

@ 语法分析器用法示例 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析器 @ <59>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = orez_parser(tokens);
        print_syntax_tree(root);
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

4 建立有名片段的联系

片段有了名字，就有了最基本的纠缠——一个片段可以引用另一个片段。现在需要为有名片段建立一个关系表，呈现它们之间的引用关系。关系表的数据结构基于 GLib 库的 Hash 表结构实现。

关系表里存储的数据，其结构为

```
@ 有名片段的联系 #
typedef struct {
        GPtrArray *time_order;
        GPtrArray *spatial_order;
        GPtrArray *emissions;
} OrezTie;
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

每一个不同名字的片段都对应一个 OrezTie 类型的对象（示例），其自身会被分别存储在 time_order 和 spatial_order。同名的片段会被以它们在文学化源码中出现次序与逻辑次序（由片段运算符决定）分别存储在 spatial_order 和 time_order 中。emissions 存储片段的引用者。

4.1 键

将 OrezTie 对象加入有名片段关系表中时，所用的键是片段的名字，但是去除了所有空白字符（空格，缩进，换行符以及续行符）和一些特殊符号，辅助函数 compact_text 可完成此事，其定义为

```
@ 建立有名片段关系表所需的辅助函数 #
static GString *compact_text(GString *a,
                             GPtrArray *useless_characters)
{
        GString *b = g_string_new(a->str);
        for (size_t i = 0; i < useless_characters->len; i++) {
                GString *c = g_ptr_array_index(useless_characters, i);
                g_string_replace(b, c->str, "", 0);
        }
        /* 去除引号 */
        g_string_replace(b, "\'", "", 0);
        g_string_replace(b, "\"", "", 0);
        /* 如有需要，可继续添加 */
        /* ... ... ... */
        return b;
}
=> 有名片段关系表的构造过程测试 <67>
```

=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>

4.2 关系表初始化

对于语法树中任一有名片段结点 x，将其存入表 relations 的过程如下

```
@ 建立有名片段关系表所需的辅助函数 # +
static void init_tie(GHashTable *relations,
                     GNode *x,
                     GPtrArray *useless_characters)
{
        GString *name = NULL;
        # 从 x 的子结点获取片段名并将其存于 name @ <63>
        GString *key = compact_text(name, useless_characters);
        OrezTie *tie = g_hash_table_lookup(relations, key);
        if (tie) {
                bool has_appending_operator = FALSE;
                bool has_prepending_operator = FALSE;
                GString *tag_reference = NULL;
                # 检测 x 是否含有运算符和标签引用 @ <63>
                if (tag_reference) {
                        GString *t = compact_text(tag_reference, useless_characters);
                        int id = -1;
                        # 从时间（程序逻辑）序列中寻找与标签引用相同的标签 -> id @ <63>
                        # 根据运算符，在时间序列中，
                          将 x 插入到所引用的标签对应的同名结点之前或之后 @ <64>
                        g_string_free(t, TRUE);
                } else {
                        # 根据运算符，在时间序列中，
                          将 x 插入到同名结点之前或之后 @ <64>
                }
                g_ptr_array_add(tie->spatial_order, x);
                g_string_free(key, TRUE);
        } else {
                # 将 x 包含在新的键值对，添加至 relations @ <65>
        }
}
```

```
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

上述代码引用的一些片段，定义如下：

```
@ 从 x 的子结点获取片段名并将其存于 name #
for (GNode *p = x->children; p != NULL; p = p->next) {
        OrezSyntax *a = p->data;
        if (a->type == OREZ_SNIPPET_NAME) {
                OrezToken *t = g_list_first(a->tokens)->data;
                name = t->content;
                break;
        }
}
if (!name) {
        g_error("Line %lu: Illegal snippet.", first_token_line_number(x));
}
=> 建立有名片段关系表所需的辅助函数 <62>

@ 检测 x 是否含有运算符和标签引用 #
for (GNode *p = x->children; p != NULL; p = p->next) {
        OrezSyntax *a = p->data;
        if (a->type == OREZ_SNIPPET_TAG_REFERENCE) {
                OrezToken *t_a = g_list_first(a->tokens)->data;
                tag_reference = t_a->content;
        }
        if (a->type == OREZ_SNIPPET_APPENDING_OPERATOR) {
                has_appending_operator = TRUE;
                break;
        }
        if (a->type == OREZ_SNIPPET_PREPENDING_OPERATOR) {
                has_prepending_operator = TRUE;
                break;
        }
}
=> 建立有名片段关系表所需的辅助函数 <62>

@ 从时间（程序逻辑）序列中寻找与标签引用相同的标签 -> id #
```

```
for (int i = 0; i < tie->time_order->len; i++) {
        GNode *y = g_ptr_array_index(tie->time_order, i);
        for (GNode *it = y->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type == OREZ_SNIPPET_TAG) {
                        OrezToken *tag_token = g_list_first(a->tokens)->data;
                        GString *s = compact_text(tag_token->content,
                                                  useless_characters);
                        if (g_string_equal(s, t)) {
                                id = i;
                                g_string_free(s, TRUE);
                                break;
                        }
                        g_string_free(s, TRUE);
                }
        }
        if (id >= 0) break;
}
```

=> 建立有名片段关系表所需的辅助函数 <62>

```
@ 根据运算符，在时间序列中，
  将 x 插入到所引用的标签对应的同名结点之前或之后 #
if (id < 0) {
        g_error("Line %lu: The referenced tag does not exist.",
                first_token_line_number(x));
} else {
        if (has_appending_operator) {
                g_ptr_array_insert(tie->time_order, id + 1, x);
        } else if (has_prepending_operator) {
                g_ptr_array_insert(tie->time_order, id, x);
        } else {
                g_error("Line %lu: The snippet needs an operator.",
                         first_token_line_number(x));
        }
}
```

=> 建立有名片段关系表所需的辅助函数 <62>

```
@ 根据运算符，在时间序列中，将 x 插入到同名结点之前或之后 #
if (has_appending_operator) {
        g_ptr_array_add(tie->time_order, x);
} else if (has_prepending_operator) {
```

```
        g_ptr_array_insert(tie->time_order, 0, x);
} else {
        g_error("Line %lu: The snippet needs an operator.",
                first_token_line_number(x));
}
=> 建立有名片段关系表所需的辅助函数 <62>

@ 将 x 包含在新的键值对，添加至 relations #
tie = malloc(sizeof(OrezTie));
tie->time_order = g_ptr_array_new();
tie->spatial_order = g_ptr_array_new();
tie->emissions = g_ptr_array_new();
g_ptr_array_add(tie->time_order, x);
g_ptr_array_add(tie->spatial_order, x);
if (!g_hash_table_insert(relations, key, tie)) {
        g_error("Line %lu: Failed to insert snippet named <%s>"
                " into the relation table.",
                first_token_line_number(x), key->str);
}
=> 建立有名片段关系表所需的辅助函数 <62>
```

4.3 建立有名片段的引用关系

假设任一有名片段结点 u，其中含有对其他片段 a 的引用，则以下过程可在有名片段关系表中为二者建立联系：

```
@ 建立有名片段关系表所需的辅助函数 # +
static void create_relation(GHashTable *relations,
                            GNode *u,
                            OrezSyntax *a,
                            GPtrArray *useless_characters)
{
        OrezToken *t = g_list_first(a->tokens)->data;
        GString *s = compact_text(t->content, useless_characters);
        OrezTie *tie = g_hash_table_lookup(relations, s);
        if (tie) {
                g_ptr_array_add(tie->emissions, u);
        } else {
                g_error("Line %lu: The snippet named <%s> never existed.",
                        t->line_number, t->content->str);
        }
```

```
        g_string_free(s, TRUE);
}
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

基于上述两个函数，构造有名片段关系的完整过程由以下函数实现：

```
@ 建立有名片段关系表所需的辅助函数 # +
static GHashTable *orez_create_relations(GNode *root, GPtrArray *useless_characters)
{
        GHashTable *relations = g_hash_table_new((GHashFunc)g_string_hash,
                                                 (GEqualFunc)g_string_equal);
        for (GNode *it = root->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type != OREZ_SNIPPET_WITH_NAME) continue;
                init_tie(relations, it, useless_characters);
        }
        for (GNode *it = root->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type != OREZ_SNIPPET_WITH_NAME) continue;
                GNode *body = g_node_last_child(it);
                for (GNode *o_it = body->children; o_it != NULL; o_it = o_it->next){
                        OrezSyntax *b = o_it->data;
                        if (b->type != OREZ_SNIPPET_REFERENCE) continue;
                        create_relation(relations, it, b, useless_characters);
                }
        }
        return relations;
}
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

4.4 销毁关系表

关系表不再使用时，可通过以下函数予以销毁：

```
@ 建立有名片段关系表所需的辅助函数 # +
static void orez_destroy_relations(GHashTable *relations)
{
        GList *keys = g_hash_table_get_keys(relations);
        GList *it = keys;
        while (it) {
                GString *key = it->data;
                OrezTie *tie = g_hash_table_lookup(relations, key);
                g_ptr_array_free(tie->spatial_order, TRUE);
                g_ptr_array_free(tie->time_order, TRUE);
                g_ptr_array_free(tie->emissions, TRUE);
                free(tie);
                g_string_free(key, TRUE);
                it = it->next;
        }
        g_list_free(keys);
        g_hash_table_destroy(relations);
}
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

4.5 测试

```
@ 有名片段关系表的构造过程测试 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析器 @ <59>
# 有名片段的联系 @ <61>
# 建立有名片段关系表所需的辅助函数 @ <61> <62> <65> <66> <67>
static void print_snippet_name(GNode *x)
{
```

```
        for (GNode *it = x->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type == OREZ_SNIPPET_NAME) {
                        OrezToken *t = g_list_first(a->tokens)->data;
                        printf("<snippet-name>%s %zu</snippet-name>\n",
                               t->content->str,
                               t->line_number);
                        break;
                }
        }
}
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = orez_parser(tokens);
        /* 创建联系 */
        # 构造冗白字符集 -> useless_characters @ <69>
        GHashTable *relations = orez_create_relations(root, useless_characters);
        GList *keys = g_hash_table_get_keys(relations);
        for (GList *it = keys; it != NULL; it = it->next) {
                GString *key = it->data;
                OrezTie *tie = g_hash_table_lookup(relations, key);
                printf("<key>%s</key>\n", key->str);
                printf("<value>\n"
                       "<spatial_order>\n");
                for (size_t i = 0; i < tie->spatial_order->len; i++) {
                        print_snippet_name(g_ptr_array_index(tie->spatial_order, i));
                }
                printf("</spatial_order>\n");
                printf("<time_order>\n");
                for (size_t i = 0; i < tie->time_order->len; i++) {
                        print_snippet_name(g_ptr_array_index(tie->time_order, i));
                }
                printf("</time_order>\n");
```

```
                printf("<emissions>");
                for (size_t i = 0; i < tie->emissions->len; i++) {
                        print_snippet_name(g_ptr_array_index(tie->emissions, i));
                }
                printf("</emmisions>\n</value>\n");
        }
        g_list_free(keys);
        orez_destroy_relations(relations);
        # 释放冗白字符集 @ <69>
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

上述代码引用的一些片段，定义如下：

```
@ 构造冗白字符集 -> useless_characters #
GPtrArray *useless_characters = g_ptr_array_new();
g_ptr_array_add(useless_characters, g_string_new(" "));
g_ptr_array_add(useless_characters, g_string_new("  "));
g_ptr_array_add(useless_characters, g_string_new("\t"));
g_ptr_array_add(useless_characters, g_string_new("\n"));
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>

@ 释放冗白字符集 #
for (size_t i = 0; i < useless_characters->len; i++) {
        GString *s = g_ptr_array_index(useless_characters, i);
        g_string_free(s, TRUE);
}
g_ptr_array_free(useless_characters, TRUE);
=> 有名片段关系表的构造过程测试 <67>
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

5 抽取

对于文学化程序源码文件 input_file_name，根据有名片段关系表 relations，以给定的入口 entrance——某个有名片段的名字，抽取片段所包含的所有内容——片段及其所引用的其他片段的内容。由于被某个片段引用的片段又可能引用其他片段，因此片段内容抽取过程是递归的。

```
@ 有名片段内容抽取所需的辅助函数 #
static void orez_tangle(const char *input_file_name,
                        GHashTable *relations,
                        GString *entrance,
                        GPtrArray *useless_characters,
                        GList **indents, /* 用于记录所引用的片段的缩进层次 */
                        bool show_line_number, /* 用于控制输出的内容是否包含行号 */
                        FILE *output)
{
        GString *key = compact_text(entrance, useless_characters);
        OrezTie *tie = g_hash_table_lookup(relations, key);
        if (!tie) {
                g_error("Snippet <%s> never existed!", entrance->str);
        }
        for (size_t i = 0; i < tie->time_order->len; i++) {
                GNode *snippet = g_ptr_array_index(tie->time_order, i);
                GNode *body = g_node_last_child(snippet);
                for (GNode *it = body->children; it != NULL; it = it->next) {
                        OrezSyntax *a = it->data;
                        OrezToken *ta = g_list_first(a->tokens)->data;
                        GString *text = ta->content;
                        if (a->type == OREZ_SNIPPET_TEXT) {
                                # 输出片段内普通文本 @ <71>
                        } else if (a->type == OREZ_SNIPPET_REFERENCE) {
                                # 输出引用的片段内容 @ <74>
                        } else {
                                g_error("Line %lu: There is an undefined type!",
                                        first_token_line_number(it));
                        }
                }
        }
        g_string_free(key, TRUE);
}
=> 片段内容抽取过程测试 <74>
```

```
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

向文件写入片段内容时，使用一个字符串作为缓冲区，收集待写入的内容，然后将其一次性写入文件。在收集待写入文件的内容之前，由于片段可能是被其他片段引用，在被引用处可能存在缩进，故而需要构造缩进文本，将其添加到片段内容的每一行的行首，从而形成整个片段的缩进。该过程对于含有诸如 Python、YAML 之类代码的片段是必须存在的，其实现如下：

```
@ 输出片段内普通文本 #
GString *cache = g_string_new(NULL);
GString *current_indent = cascade_indents(*indents); /* 将各层次的缩进串联起来 */
if (show_line_number) {
        GString *line_number = g_string_new(NULL);
        g_string_printf(line_number,
                        "#line %zu \"%s\"\n",
                        first_token_line_number(it), input_file_name);
        g_string_append(cache, line_number->str);
        g_string_free(line_number, TRUE);

}
char *body_head = squeeze_head(text);
char *body_tail = squeeze_tail(text);
if (body_head < body_tail) {/* 忽略空白段 */
        g_string_append(cache, current_indent->str);
        for (char *p = body_head; p != body_tail; p++) {
                g_string_append_c(cache, *p);
                if (*p == '\n') {
                        g_string_append(cache, current_indent->str);
                }
        }
        fputs(cache->str, output);
        fputs("\n", output);
}
g_string_free(current_indent, TRUE);
g_string_free(cache, TRUE);
=> 有名片段内容抽取所需的辅助函数 <70>
```

在文学化程序源码中，有名片段的首位皆存在空行，它们的作用是在视觉上让有名片段的内容更加明显突出，在抽取片段内容时，可通过函数 `squeez_text` 予以去除。

@ 有名片段内容抽取所需的辅助函数 # ^+

<squeeze head and tail>

```
static char *squeeze_head(GString *text)
{
        char *p = text->str;
        while (1) {
                if (*p == ' ' || *p == '\t') p++;
                else if ((unsigned char)*p == 0xE3) {
                        char *q = p + 1;
                        if ((unsigned char)*q == 0x80) {
                                char *r = q + 1;
                                if ((unsigned char)*r == 0x80) {
                                        p = r + 1;
                                } else break;
                        } else break;
                } else if (*p == '\n') {
                        p++;
                        break;
                } else break;
        }
        return p;
}
static char *squeeze_tail(GString *text)
{
        char *p = text->str + text->len - 1;
        while (1) {
                if (p == text->str) break;
                else if (*p == ' ' || *p == '\t') p--;
                else if ((unsigned char)*p == 0x80) {
                        char *q = p - 1;
                        if ((unsigned char)*q == 0x80) {
                                char *r = q - 1;
                                if ((unsigned char)*r == 0xE3) {
                                        p = r - 1;
                                } else break;
                        } else break;
                } else if (*p == '\n') {
                        break;
                } else break;
        }
        return p;
```

```
}
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

串联各层次缩进的函数 cascade_indents 的定义如下：

```
@ 有名片段内容抽取所需的辅助函数 # ^+
static GString *cascade_indents(GList *indents)
{
        GString *current_indent = g_string_new(NULL);
        for (GList *it = indents; it != NULL; it = it->next) {
                GString *a = it->data;
                g_string_append(current_indent, a->str);
        }
        return current_indent;
}
=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

输出引用的片段内容，本质上是递归调用上述的片段内容的输出过程，但是需要为该过程构造正确的缩进层次，否则抽取的内容会出现缩进混乱的情况。引用的片段，其缩进即其前一个 OREZ_SNIPPET_TEXT 类型的结点最后一行空白文本，需提取该段文本，以添加到引用的片段内容除首行之外的每行行首，提取过程如下：

```
@ 有名片段内容抽取所需的辅助函数 # <squeeze head and tail> +
static GString *take_last_blank_line(OrezSyntax *a)
{
        assert(a);
        assert(a->type == OREZ_SNIPPET_TEXT);
        GString *blank_line = g_string_new(NULL);
        OrezToken *t = g_list_first(a->tokens)->data;
        if (t->content->len <= 1) return blank_line;
        else {
                char *p = squeeze_tail(t->content);
                if (p < t->content->str + t->content->len) {
                        for (char *q = p + 1; *q != '\0'; q++) {
```

```
                                g_string_append_c(blank_line, *q);
                        }
                }
        }
        return blank_line;
}
```

=> 片段内容抽取过程测试 <74>
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>

假设引用的片段为 a，其对应的语法结点为 it，则为其准备当前层次的缩进文本并递归调用 orez_tangle 实现引用的片段内容的提取过程为：

```
@ 输出引用的片段内容 #
GNode *prev_it = g_node_prev_sibling(it);
if (prev_it) {
        OrezSyntax *prev_a = prev_it->data;
        if (prev_a->type == OREZ_SNIPPET_TEXT) {
                GString *blank_line = take_last_blank_line(prev_a);
                *indents = g_list_prepend(*indents, blank_line);
        }
        OrezToken *ta = g_list_first(a->tokens)->data;
        orez_tangle(input_file_name, relations,
                    ta->content, useless_characters,
                    indents, show_line_number, output);
        /* 删除当前层次的缩进 */
        g_string_free((*indents)->data, TRUE);
        *indents = g_list_delete_link(*indents, *indents);
}
```

=> 有名片段内容抽取所需的辅助函数 <70>

以下为片段内容抽取过程的测试代码：

```
@ 片段内容抽取过程测试 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析器 @ <59>
# 有名片段的联系 @ <61>
```

```
# 建立有名片段关系表所需的辅助函数 @ <61> <62> <65> <66> <67>
# 有名片段内容抽取所需的辅助函数 @ <70> <72> <73> <73>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = orez_parser(tokens);
        /* 创建联系 */
        # 构造冗白字符集 -> useless_characters @ <69>
        GHashTable *relations = orez_create_relations(root, useless_characters);
        bool show_line_number = FALSE;
        GString *entrance = g_string_new("hello world");
        GList *indents = NULL;
        FILE *output = fopen("foo.c", "w");
        orez_tangle(input_file_name, relations,
                    entrance, useless_characters,
                    &indents, show_line_number, output);
        g_string_free(entrance, TRUE);
        fclose(output);
        orez_destroy_relations(relations);
        # 释放冗白字符集 @ <69>
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

6 织成

所谓“织成”，是为文学化程序源码的文档排版过程提供充分信息的过程，例如需要将部分有名片段的语言标记传递到与它们相关的片段，需要针对某种以标记语言（如 LaTeX、ConTeXt、Markdown、HTML 等）为基础的排版软件建立有名片段的引用关系。这些信息能够更为直观地呈现文学化程序的内容，使之易于阅读。

6.1 语言标记传播

有名片段通常用于存放程序源码。在文学化程序的文档排版过程中为了实现程序源码的高亮渲染，需要知道程序原码所属语言，因此每个有名片段皆需要带有语言标记，但是为了方便，Orez 只需为属于同一线索的所有的有名片段中的任意一个提供语言标记，然后在织成阶段，将该标记传播至该线索中的其他片段。

辅助函数 `get_thread` 可从有名片段关系表中抽取指定线索：

```
@ 语言标记传播过程所需的辅助函数 #
static GList *get_thread(GHashTable *relations,
                         GString *entrance,
                         GPtrArray *useless_characters,
                         GList *thread)
{
        GString *key = compact_text(entrance, useless_characters);
        OrezTie *tie = g_hash_table_lookup(relations, key);
        for (size_t i = 0; i < tie->spatial_order->len; i++) {
                GNode *t = g_ptr_array_index(tie->spatial_order, i);
                GNode *x = g_node_last_child(t);
                thread = g_list_prepend(thread, t);
                for (GNode *it = x->children; it != NULL; it = it->next) {
                        OrezSyntax *a = it->data;
                        if (a->type == OREZ_SNIPPET_REFERENCE) {
                                OrezToken *ta = g_list_first(a->tokens)->data;
                                thread = get_thread(relations,
                                                    ta->content,
                                                    useless_characters,
                                                    thread);
                        }
                }
        }
        g_string_free(key, TRUE);
        return thread;
}
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

辅助函数 `spread_language_mark_in_thread` 实现任意一条线索中的有名片段的语言标记传播：

@ 语言标记传播过程所需的辅助函数 # +

```
static OrezToken *find_language_token(GNode *x)
{
        OrezToken *language_token = NULL;
        for (GNode *p = x->children; p != NULL; p = p->next) {
                OrezSyntax *a = p->data;
                if (a->type == OREZ_SNIPPET_LANGUAGE) {
                        language_token = g_list_first(a->tokens)->data;
                        break;
                }
        }
        return language_token;
}
static void spread_language_mark_in_thread(GList *thread)
{
        OrezToken *language_token = NULL;
        for (GList *it = thread; it != NULL; it = it->next) {
                language_token = find_language_token(it->data);
                if (language_token) break;
        }
        if (!language_token) return;
        for (GList *it = thread; it != NULL; it = it->next) {
                GNode *t = it->data;
                /* 检测 t 是否含有语言标记 */
                OrezToken *language_token_a = find_language_token(t);;
                if (language_token_a) {
                        if (!g_string_equal(language_token->content,
                                            language_token_a->content)) {
                                g_error("Line %lu and %lu: "
                                        "there are two different language marks.",
                                        language_token->line_number,
                                        language_token_a->line_number);
                        }
                } else {
                        # 构造语言标记结点 @ <78>
                }
        }
}
```

=> 语言标记传播过程测试 <79>

=> 无名片段输出过程测试 <85>

=> 有名片段输出过程测试 <91>

```
=> orez.c <96>

@ 构造语言标记结点 #
OrezToken *new_language_token = malloc(sizeof(OrezToken));
new_language_token->type = OREZ_TEXT;
new_language_token->content = g_string_new(language_token->content->str);
/* 确定语言标记的行号 */
OrezSyntax *name = g_node_first_child(t)->data;
OrezToken *name_token = g_list_first(name->tokens)->data;
size_t line_number = name_token->line_number;
for (char *p = name_token->content->str; *p != '\0'; p++) {
        if (*p == '\n') line_number++;
}
new_language_token->line_number = line_number;
/* 构造语法结点 */
OrezSyntax *new_language = malloc(sizeof(OrezSyntax));
new_language->type = OREZ_SNIPPET_LANGUAGE;
new_language->tokens = g_list_append(NULL, new_language_token);
g_node_insert_data_after(t, g_node_first_child(t), new_language);
=> 语言标记传播过程所需的辅助函数 <77>
```

在语法树中所有的有名结点中传播语言标记的过程由辅助函数 spread_language_mark 实现：

```
@ 语言标记传播过程所需的辅助函数 # +
static void spread_language_mark(GNode *root,
                                 GHashTable *relations,
                                 GPtrArray *useless_characters)
{
        for (GNode *it = root->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type == OREZ_SNIPPET_WITH_NAME) {
                        OrezSyntax *b = g_node_first_child(it)->data;
                        OrezToken *token_b = g_list_first(b->tokens)->data;
                        GList *thread = get_thread(relations,
                                                   token_b->content,
                                                   useless_characters,
                                                   NULL);
                        spread_language_mark_in_thread(thread);
                        g_list_free(thread);
                }
        }
```

```
        /* 检测 */
        for (GNode *it = root->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type == OREZ_SNIPPET_WITH_NAME) {
                        OrezToken *language_token = find_language_token(it);
                        if (!language_token){
                                OrezSyntax *name = g_node_first_child(it)->data;
                                OrezToken *name_token = g_list_first(name->tokens)->data;
                                g_warning("Line %lu: "
                                          "the language of the snippet <%s> is unknown.",
                                          name_token->line_number,
                                          name_token->content->str);
                        }
                }
        }
}
=> 语言标记传播过程测试 <79>
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

以下是语言标记传播过程的测试代码：

```
@ 语言标记传播过程测试 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析器 @ <59>
# 有名片段的联系 @ <61>
# 建立有名片段关系表所需的辅助函数 @ <61> <62> <65> <66> <67>
# 有名片段内容抽取所需的辅助函数 @ <70> <72> <73> <73>
# 语言标记传播过程所需的辅助函数 @ <76> <77> <78>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
```

```
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = orez_parser(tokens);
        /* 创建联系 */
        # 构造冗白字符集 -> useless_characters @ <69>
        GHashTable *relations = orez_create_relations(root, useless_characters);
        spread_language_mark(root, relations, useless_characters);
        orez_destroy_relations(relations);
        # 释放冗白字符集 @ <69>
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

6.2 清理冗白

无名片段前后的空白字符或空行皆可予以清除，若后续输出排版格式文档，可根据需要另行添加。清楚这些冗白信息，需要以下辅助函数：

```
@ 清理冗白信息所需的辅助函数 #
/* 消除 text 的前冗白。注意: text 会被释放。*/
static GString *orez_string_chug(GString *text, GPtrArray *useless_characters)
{
        if (text->len == 0) return text;
        GString *new_text = g_string_new(NULL);
        /* 构造字符串闭区间 [a, b] */
        char *a = text->str, *b = text->str + text->len - 1;
        char *p = a;
        while (1) {
                char *t = p;
                for (size_t i = 0; i < useless_characters->len; i++) {
                        GString *c = g_ptr_array_index(useless_characters, i);
                        char *q = p;
                        if (am_i_here(p, b, &q, c->str, 1)) p = q + 1;
                }
                if (t == p) break;
        }
        for (; *p != '\0'; p++) g_string_append_c(new_text, *p);
        g_string_free(text, TRUE);
        return new_text;
```

```
}
/* 消除 text 的后冗白。注意: text 会被释放。*/
static GString *orez_string_chomp(GString *text, GPtrArray *useless_characters)
{
        if (text->len == 0) return text;
        /* 构造字符串闭区间 [a, b] */
        char *a = text->str, *b = text->str + text->len - 1;
        GString *new_text = g_string_new(NULL);
        char *p = b;
        while (1) {
                char *t = p;
                for (size_t i = 0; i < useless_characters->len; i++) {
                        GString *c = g_ptr_array_index(useless_characters, i);
                        char *q = p;
                        if (am_i_here(a, p, &q, c->str, -1)) {
                                p = q - 1;
                        }
                }
                if (t == p /* 走不动了 */
                    || p < a /* 走过头了 */ ) break;
        }
        p++;
        for (char *q = text->str; q != p; q++) {
                g_string_append_c(new_text, *q);
        }
        g_string_free(text, TRUE);
        return new_text;
}
/* 消除 text 的前后冗白。注意: text 会被释放。*/
static GString *orez_string_strip(GString *text, GPtrArray *useless_characters)
{
        GString *new_text = orez_string_chomp(orez_string_chug(text,
                                                               useless_characters),
                                              useless_characters);
        return new_text;
}
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

基于上述辅助函数，无名片段的冗白清理过程如下：

```
@ 清理冗白信息所需的辅助函数 # +
static void strip_snippet(GNode *x, GPtrArray *useless_characters)
{
        OrezSyntax *a = x->data;
        OrezToken *t_a = g_list_first(a->tokens)->data;
        t_a->content = orez_string_strip(t_a->content, useless_characters);
}
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

有名片段的内容部分，其前导冗白有时出于排版的目的而存在，可对其予以保留，仅去除尾部冗白；至于片段名字、语言标记、标签、片段引用等内容，前后冗白皆需去除。有名片段的冗白清除过程如下：

```
@ 清理冗白信息所需的辅助函数 # +
static void strip_snippet_with_name(GNode *x, GPtrArray *useless_characters)
{
        for (GNode *it = x->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type == OREZ_SNIPPET_CONTENT) {
                        OrezSyntax *b = NULL;
                        OrezToken *t_b = NULL;
                        for (GNode *p = it->children; p != NULL; p = p->next) {
                                b = p->data;
                                # 清理片段引用前后的冗白 @ <83>
                        }
                        # 此时，b 指向最后的单元，仅去除其尾部冗白 @ <83>
                } else {
                        OrezSyntax *b = it->data;
                        OrezToken *t_b = g_list_first(b->tokens)->data;
                        t_b->content = orez_string_strip(t_b->content,
                                                         useless_characters);
                }
        }
}
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

上述代码引用的代码片段定义如下：

```
@ 清理片段引用前后的冗白 #
t_b = g_list_first(b->tokens)->data;
if (b->type == OREZ_SNIPPET_REFERENCE) {
        t_b->content = orez_string_strip(t_b->content, useless_characters);
}
=> 清理冗白信息所需的辅助函数 <82>

@ 此时，b 指向最后的单元，仅去除其尾部冗白 #
if (b->type == OREZ_SNIPPET_TEXT) {
        t_b->content = orez_string_chomp(t_b->content, useless_characters);
} else {
        g_warning("Line %lu: Illegal snippet.", t_b->line_number);
}
=> 清理冗白信息所需的辅助函数 <82>
```

运用上述函数，对语法树中所有片段结点予以处理：

```
@ 清理冗白信息所需的辅助函数 # +
static void strip_all_snippets(GNode *root,
                               GPtrArray *useless_characters)
{
        for (GNode *it = root->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type == OREZ_SNIPPET) {
                        strip_snippet(it, useless_characters);
                } else {
                        strip_snippet_with_name(it, useless_characters);
                }
        }
}
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

6.3 格式

织成过程输出的内容反映的是文学化程序源码的语法树的结构以及有名片段关系表的内容，为了便于后续工作的扩展，织成过程采用能够被现代多数编程语言（或它们的库）支持的 YAML 格式作为输出内容的格式。

对于无名片段，Orez 将其输出为以下格式：

```
- type: 0
```

```
  content: 片段内容
```

对于有名片段，Orez 将其输出为以下格式：

```
- type: 1
  name: 片段名
  hash: 片段名，去除了冗白和特殊字符
  id: 在同名片段序列中的 ID。若该片段无同名片段，则该项不存在。
  language: 语言标记
  content:
    - text: 普通文本
    - reference:
        name: 被引用片段的名字
        hash: 被引用片段的名字，去除了冗白和特殊字符
        ids:  # 若被引用片段存在多个同名片段，该项为其 ID 序列，否则该项不存在
          - 1
          - 2
          - ...
    - text: 普通文本
    - refernece:
        ... ... ...
    - ...
  emissions: # 若存在片段的引用者，该项为引用者序列，否则该项不存在
    - emission:
        name: 引用者的名字
        hash: 引用者的名字，去除了冗白和特殊字符
        id: 引用者在同名片段序列中的 ID。若引用者无其他同名片段，则此项不存在。
    - emission:
        ... ... ...
    - ...
```

6.4 输出无名片段

采用一个存储字符串对象的列表收集所有输出内容，待收集过程结束后，再将该列表的内容写入文件。对于语法树中任一无名片段结点，其内容的输出过程如下：

```
@ 输出无名片段所需的辅助函数 #
<output_snippet>
static GList *output_snippet(GNode *x, GList *yaml)
{
        OrezSyntax *snippet = x->data;
        OrezToken *snippet_token = g_list_first(snippet->tokens)->data;
```

```
        GString *cache = g_string_new("- type: 0\n");
        g_string_append(cache, "  content: |-\n");
        /* 缩进 4 个空格，增加左引号 */
        g_string_append(cache, "    '");
         /* 缩进 4 个空格 */
        append_yaml_string(cache, snippet_token->content->str, "    ", NULL);
        g_string_append(cache, "'\n"); /* 增加右引号 */
        return g_list_append(yaml, cache);
}
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

上述代码中使用的 append_yaml_string 函数，定义如下：

```
@ 输出无名片段所需的辅助函数 # <output_snippet> ^+
static void append_yaml_string(GString *cache, char *text, const char *indent, char *padding)
{
        /* 处理多行字串的缩进 */
        for (char *p = text; *p != '\0'; p++) {
                g_string_append_c(cache, *p);
                if (*p == '\n') {
                        g_string_append(cache, indent);
                        if (padding) g_string_append(cache, padding);
                }
        }
}
=> 无名片段输出过程测试 <85>
=> 有名片段输出过程测试 <91>
=> orez.c <96>
```

以下代码用于测试无名片段输出过程：

```
@ 无名片段输出过程测试 #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析器 @ <59>
# 有名片段的联系 @ <61>
# 建立有名片段关系表所需的辅助函数 @ <61> <62> <65> <66> <67>
# 有名片段内容抽取所需的辅助函数 @ <70> <72> <73> <73>
```

```
# 语言标记传播过程所需的辅助函数 @ <76> <77> <78>
# 清理冗白信息所需的辅助函数 @ <80> <82> <82> <83>
# 输出无名片段所需的辅助函数 @ <84> <85>
int main(void)
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = orez_parser(tokens);
        /* 创建联系 */
        # 构造冗白字符集 -> useless_characters @ <69>
        GHashTable *relations = orez_create_relations(root, useless_characters);
        spread_language_mark(root, relations, useless_characters);
        strip_all_snippets(root, useless_characters);
        GList *yaml = NULL;
        for (GNode *it = root->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type == OREZ_SNIPPET) yaml = output_snippet(it, yaml);
        }
        for (GList *it = yaml; it != NULL; it = it->next) {
                GString *a = it->data;
                printf("%s", a->str);
                g_string_free(a, TRUE);
        }
        g_list_free(yaml);
        orez_destroy_relations(relations);
        # 释放冗白字符集 @ <69>
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

6.5 输出有名片段

假设有名片段结点 x，将其内容输出为 YAML 的映射对象，过程如下：

@ 输出有名片段所需的辅助函数 #

```
static GList *output_snippet_with_name(GNode *x,
                                       GHashTable *relations,
                                       GPtrArray *useless_characters,
                                       char *snippet_reference_padding,
                                       GList *yaml)
{
        OrezSyntax *name = g_node_first_child(x)->data;
        OrezToken *name_token = g_list_first(name->tokens)->data;
        GString *x_tie_key = compact_text(name_token->content, useless_characters);
        OrezTie *x_tie = g_hash_table_lookup(relations, x_tie_key);
        GString *cache = g_string_new("- type: 1\n");
        for (GNode *it = x->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                switch (a->type) {
                case OREZ_SNIPPET_NAME:
                        do {
                                # 输出片段的名字和 ID @ <88>
                        } while (0);
                        break;
                case OREZ_SNIPPET_LANGUAGE:
                        do {
                                # 输出片段的语言标记 @ <89>
                        } while (0);
                        break;
                case OREZ_SNIPPET_TAG_REFERENCE:
                        do {
                                # 输出引用的片段标签 @ <89>
                        } while (0);
                        break;
                case OREZ_SNIPPET_PREPENDING_OPERATOR:
                        do {
                                # 输出片段运算符 @ <89>
                        } while (0);
                        break;
                case OREZ_SNIPPET_APPENDING_OPERATOR:
                        do {
                                # 输出片段运算符 @ <89>
                        } while (0);
                        break;
                case OREZ_SNIPPET_TAG:
```

```
                        do {
                                # 输出片段标签 @ <89>
                        } while (0);
                        break;
                case OREZ_SNIPPET_CONTENT:
                        do {
                                # 输出片段内容 @ <89>
                        } while (0);
                        break;
                default:
                        printf("Line %lu: Unknown syntax in snippet <%s>.\n",
                               first_token_line_number(g_node_first_child(x)),
                               name_token->content->str);
                }
        }
        if (x_tie->emissions->len > 0) {
                # 输出该片段的引用者 @ <90>
        }
        g_string_free(x_tie_key, TRUE);
        return g_list_append(yaml, cache);
}
=> 有名片段输出过程测试 <91>
=> orez.c <96>

@ 输出片段的名字和 ID #
/* 片段名 */
OrezToken *t = g_list_first(a->tokens)->data;
g_string_append(cache, "  name: |-\n");
g_string_append(cache, "    '");
append_yaml_string(cache, t->content->str, "    ", NULL);
g_string_append(cache, "'\n");
/* 片段名的 hash 值 */
g_string_append_printf(cache, "  hash: '%s'\n", x_tie_key->str);
/* 确定 id */
if (x_tie->spatial_order->len > 1) {
        size_t id = 0;
        for (size_t i = 0; i < x_tie->spatial_order->len; i++) {
                if (x == g_ptr_array_index(x_tie->spatial_order, i)) {
                        id = i;
                        break;
                }
```

```
        }
        g_string_append_printf(cache, "  id: %lu\n", id + 1);
}
=> 输出有名片段所需的辅助函数 <87>

@ 输出片段的语言标记 #
OrezToken *t = g_list_first(a->tokens)->data;
g_string_append_printf(cache, "  language: '%s'\n", t->content->str);
=> 输出有名片段所需的辅助函数 <87>

@ 输出引用的片段标签 #
OrezToken *t = g_list_first(a->tokens)->data;
g_string_append(cache, "  tag_reference: \n");
g_string_append(cache, "    name: |-\n");
g_string_append(cache, "      '");
append_yaml_string(cache, t->content->str, "       ", NULL);
g_string_append(cache, "'\n");
/* 输出 hash */
GString *tag_reference_hash = compact_text(t->content, useless_characters);
g_string_append_printf(cache, "    hash: '%s'\n", tag_reference_hash->str);
g_string_free(tag_reference_hash, TRUE);
=> 输出有名片段所需的辅助函数 <87>

@ 输出片段运算符 #
OrezToken *t = g_list_first(a->tokens)->data;
g_string_append_printf(cache, "  operator: '%s'\n", t->content->str);
=> 输出有名片段所需的辅助函数 <87>
=> 输出有名片段所需的辅助函数 <87>

@ 输出片段标签 #
OrezToken *t = g_list_first(a->tokens)->data;
g_string_append(cache, "  tag: \n");
g_string_append(cache, "    name: |-\n");
g_string_append(cache, "      '");
append_yaml_string(cache, t->content->str, "       ", NULL);
g_string_append(cache, "'\n");
/* 输出 hash */
GString *tag_hash = compact_text(t->content, useless_characters);
g_string_append_printf(cache, "    hash: '%s'\n", tag_hash->str);
g_string_free(tag_hash, TRUE);
=> 输出有名片段所需的辅助函数 <87>

@ 输出片段内容 #
```

```
g_string_append(cache, "  content: \n");
for (GNode *p = it->children; p != NULL; p = p->next) {
        OrezSyntax *b = p->data;
        OrezToken *t_b = g_list_first(b->tokens)->data;
        if (b->type == OREZ_TEXT) {
                if (t_b->content->len > 0) {
                        g_string_append(cache, "    - text: |-\n");
                        g_string_append(cache, "        '");
                        append_yaml_string(cache,
                                           t_b->content->str,
                                           "        ", NULL);
                        g_string_append(cache, "'\n");
                }
        } else { /* 片段引用 */
                GString *y_tie_key = compact_text(t_b->content, useless_characters);
                OrezTie *y_tie = g_hash_table_lookup(relations, y_tie_key);
                g_string_append(cache, "    - reference: \n");
                g_string_append(cache, "        name: |-\n");
                g_string_append(cache, "          '");
                append_yaml_string(cache,
                                   t_b->content->str,
                                   "          ",
                                   snippet_reference_padding);
                g_string_append(cache, "'\n");
                g_string_append_printf(cache, "        hash: '%s'\n", y_tie_key->str);
                /* 输出引用片段的所有 ID */
                if (y_tie->spatial_order->len > 1) {
                        g_string_append(cache, "        ids: \n");
                        for (size_t i = 0; i < y_tie->spatial_order->len; i++) {
                                g_string_append_printf(cache,
                                                       "          - %lu\n",
                                                       i + 1);
                        }
                }
                g_string_free(y_tie_key, TRUE);
        }
}
```

=> 输出有名片段所需的辅助函数 <87>

@ 输出该片段的引用者 #

```
g_string_append(cache, "  emissions: \n");
```

```
for (size_t i = 0; i < x_tie->emissions->len; i++) {
        GNode *e = g_ptr_array_index(x_tie->emissions, i);
        OrezSyntax *b = g_node_first_child(e)->data;
        OrezToken *t_b = g_list_first(b->tokens)->data;
        GString *y_tie_key = compact_text(t_b->content, useless_characters);
        OrezTie *y_tie = g_hash_table_lookup(relations, y_tie_key);
        g_string_append(cache, "    - emission: \n");
        g_string_append(cache, "        name: |-\n");
        g_string_append(cache, "          '");
        append_yaml_string(cache, t_b->content->str, "          ", NULL);
        g_string_append(cache, "'\n");
        g_string_append_printf(cache, "        hash: '%s'\n", y_tie_key->str);
        /* 确定 id */
        if (y_tie->spatial_order->len > 1) {
                for (size_t j = 0; j < y_tie->spatial_order->len; j++) {
                        if (e == g_ptr_array_index(y_tie->spatial_order, j)) {
                                g_string_append_printf(cache,
                                                       "        id: %lu\n",
                                                       j + 1);
                                break;
                        }
                }
        }
        g_string_free(y_tie_key, TRUE);
}
```

=> 输出有名片段所需的辅助函数 <87>

@ 有名片段输出过程测试 #

词法分析过程的外境 @ <45>

记号 @ <6> <7> <8> <14>

词法分析器 @ <44>

语法结点 @ <46> <46>

语法分析器 @ <59>

有名片段的联系 @ <61>

建立有名片段关系表所需的辅助函数 @ <61> <62> <65> <66> <67>

有名片段内容抽取所需的辅助函数 @ <70> <72> <73> <73>

语言标记传播过程所需的辅助函数 @ <76> <77> <78>

清理冗白信息所需的辅助函数 @ <80> <82> <82> <83>

输出无名片段所需的辅助函数 @ <84> <85>

输出有名片段所需的辅助函数 @ <87>

```
int main(void)
```

```
{
        setlocale(LC_ALL, "");
        char *config_file = "orez.conf";
        char *input_file_name = "foo.orz";
        # 符号表初始化 @ <8>
        # YAML 配置文件解析 @ <11>
        # 配置用户定义的符号 @ <12>
        # 构造符号集 -> symbols @ <15>
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = orez_parser(tokens);
        /* 创建联系 */
        # 构造冗白字符集 -> useless_characters @ <69>
        GHashTable *relations = orez_create_relations(root, useless_characters);
        spread_language_mark(root, relations, useless_characters);
        strip_all_snippets(root, useless_characters);
        /* 输出 */
        GString *snippet_reference_padding
                = make_padding(symbols->snippet_reference_beginning_mark->len);
        GList *yaml = NULL;
        for (GNode *it = root->children; it != NULL; it = it->next) {
                OrezSyntax *a = it->data;
                if (a->type == OREZ_SNIPPET) {
                        yaml = output_snippet(it, yaml);
                } else {
                        yaml = output_snippet_with_name(it,
                                                        relations,
                                                        useless_characters,
                                                        snippet_reference_padding->str,
                                                        yaml);
                }
        }
        for (GList *it = yaml; it != NULL; it = it->next) {
                GString *a = it->data;
                printf("%s", a->str);
                g_string_free(a, TRUE);
        }
        g_list_free(yaml);
        g_string_free(snippet_reference_padding, TRUE);
        orez_destroy_relations(relations);
        # 释放冗白字符集 @ <69>
        delete_syntax_tree(root);
```

```
        # 释放符号集 @ <15>
        # 释放配置文件占用的资源 @ <12>
        return 0;
}
```

7 命令行界面

首先，需要定义一些全局变量，用于接收命令行参数：

```
@ 命令行界面所需的全局变量 #
gboolean orez_tangle_mode = FALSE;
gboolean orez_weave_mode = FALSE;
gboolean orez_show_line_number = FALSE;
char *orez_configure = NULL;
char *orez_entrance = NULL;
char *orez_output = NULL;
char *orez_separator = NULL;
static GOptionEntry orez_entries[] = {
        {"config", 'c', 0, G_OPTION_ARG_STRING, &orez_configure,
         "Read configure file.", "<file name>"},
        {"tangle", 't', 0, G_OPTION_ARG_NONE, &orez_tangle_mode,
         "use tangle.", NULL},
        {"line", 'l', 0, G_OPTION_ARG_NONE, &orez_show_line_number,
         "Show line number when using tangle.", NULL},
        {"weave", 'w', 0, G_OPTION_ARG_NONE, &orez_weave_mode, "use weave.", NULL},
        {"entrance", 'e', 0, G_OPTION_ARG_STRING, &orez_entrance,
         "Set <snippet name> as the entrance for tangle.", "<snippet name>"},
        {"output", 'o', 0, G_OPTION_ARG_STRING, &orez_output,
         "Send output into <file>.", "<file name>"},
        {"separator", 's', 0, G_OPTION_ARG_STRING, &orez_separator,
         "Set the separator for multi-entrances and outputs.", "<character>"},
        {NULL}
};
=> orez.c <96>
```

以下代码可从 main 函数的参数 argv 中截获相应参数并将其赋予上述全局变量：

```
@ 获取命令行参数 -> option_context #
GOptionContext *option_context = g_option_context_new("FILE");
g_option_context_add_main_entries(option_context, orez_entries, NULL);
if (!g_option_context_parse(option_context, &argc, &argv, NULL)) return -1;
```

```
if (argv[1] == NULL) g_error("You should provide an orez file!\n");
if (orez_tangle_mode && orez_weave_mode) {
        g_error("The tangle and weave modes can not be turned on simultaneously!");
}
if (orez_tangle_mode) {
        if (!orez_entrance) {
                g_error("You should provided the start of thread to be tangled");
        }
}
char *input_file_name = argv[1]; /* main 函数的参数列表经上述过程截取后剩下的部分 */
```
=> orez.c <96>

Orez 符号集构造过程如下：

```
@ Orez 辅助函数 #
static OrezSymbols *orez_create_symbols(const char *configure_file_name)
{
        # 符号表初始化 @ <8>
        if (configure_file_name) {
                # YAML 配置文件的模式 @ <9>
                OrezConfig *user_config;
                cyaml_err_t err = cyaml_load_file(configure_file_name,
                                                  &cyaml_config,
                                                  &top_schema,
                                                  (cyaml_data_t **)&user_config,
                                                  NULL);
                if (err != CYAML_OK) {
                        fprintf(stderr, "Failed to open config file!\n");
                        exit(EXIT_FAILURE);
                }
                # 配置用户定义的符号 @ <12>
                # 构造符号集 -> symbols @ <15>
                cyaml_free(&cyaml_config, &top_schema, user_config, 0);
                return symbols;
        } else {
                # 构造符号集 -> symbols @ <15>
                return symbols;
        }
}
```
=> orez.c <96>

在 tangle 模式下，需要处理多个入口的线索抽取情况，该过程的实现如下

```
@ tangle 模式 #
GPtrArray *entrances = g_ptr_array_new();
GPtrArray *file_names = g_ptr_array_new();
char *u = orez_output ? orez_output : orez_entrance;
if (!orez_separator) {
        GString *t = g_string_new(u);
        g_ptr_array_add(entrances, g_string_new(orez_entrance));
        g_ptr_array_add(file_names, compact_text(t, useless_characters));
        g_string_free(t, TRUE);
} else {
        gchar **cf_names = g_strsplit(orez_entrance, orez_separator, 0);
        gchar **v = g_strsplit(u, orez_separator, 0);
        guint m = 0; for (gchar **s = cf_names; *s != NULL; s++) m++;
        guint n = 0; for (gchar **s = v; *s != NULL; s++) n++;
        assert(m == n);
        for (guint i = 0; i < m; i++) {
                GString *t = g_string_new(v[i]);
                g_ptr_array_add(entrances, g_string_new(cf_names[i]));
                g_ptr_array_add(file_names, compact_text(t, useless_characters));
                g_string_free(t, TRUE);
        }
        g_strfreev(v);
        g_strfreev(cf_names);
}
for (guint i = 0; i < file_names->len; i++) {
        GString *entrance = g_ptr_array_index(entrances, i);
        GString *file_name = g_ptr_array_index(file_names, i);
        FILE *output = fopen(file_name->str, "w");
        GList *indents = NULL;
        orez_tangle(input_file_name,
                    relations,
                    entrance,
                    useless_characters,
                    &indents,
                    orez_show_line_number,
                    output);
        fclose(output);
        g_string_free(file_name, TRUE);
        g_string_free(entrance, TRUE);
}
g_ptr_array_free(file_names, TRUE);
```

```
g_ptr_array_free(entrances, TRUE);
=> orez.c <96>
```

weave 的过程如下：

```
@ weave 模式 #
spread_language_mark(root, relations, useless_characters);
strip_all_snippets(root, useless_characters);
GString *snippet_reference_padding
        = make_padding(symbols->snippet_delimiter->len);
GList *yaml = NULL;
for (GNode *it = root->children; it != NULL; it = it->next) {
        OrezSyntax *a = it->data;
        if (a->type == OREZ_SNIPPET) {
                yaml = output_snippet(it, yaml);
        } else {
                yaml = output_snippet_with_name(it,
                                                relations,
                                                useless_characters,
                                                snippet_reference_padding->str,
                                                yaml);
        }
}
FILE *output = NULL;
if (orez_output) output = fopen(orez_output, "w");
for (GList *it = yaml; it != NULL; it = it->next) {
        GString *a = it->data;
        if (output) fprintf(output, "%s", a->str);
        else printf("%s", a->str);
        g_string_free(a, TRUE);
}
g_list_free(yaml);
if (output) fclose(output);
g_string_free(snippet_reference_padding, TRUE);
=> orez.c <96>

@ orez.c #
# 词法分析过程的外境 @ <45>
# 记号 @ <6> <7> <8> <14>
# 词法分析器 @ <44>
# 语法结点 @ <46> <46>
# 语法分析器 @ <59>
```

```
# 有名片段的联系 @ <61>
# 建立有名片段关系表所需的辅助函数 @ <61> <62> <65> <66> <67>
# 有名片段内容抽取所需的辅助函数 @ <70> <72> <73> <73>
# 语言标记传播过程所需的辅助函数 @ <76> <77> <78>
# 清理冗白信息所需的辅助函数 @ <80> <82> <82> <83>
# 输出无名片段所需的辅助函数 @ <84> <85>
# 输出有名片段所需的辅助函数 @ <87>
# Orez 辅助函数 @ <94>
# 命令行界面所需的全局变量 @ <93>
int main(int argc, char **argv)
{
        setlocale(LC_ALL, "");
        # 获取命令行参数 -> option_context @ <93>
        OrezSymbols *symbols = orez_create_symbols(orez_configure);
        GList *tokens = orez_lexer(input_file_name, symbols);
        GNode *root = orez_parser(tokens);
        # 构造冗白字符集 -> useless_characters @ <69>
        GHashTable *relations = orez_create_relations(root, useless_characters);
        if (orez_tangle_mode) {
                # tangle 模式 @ <95>
        }
        if (orez_weave_mode) {
                # weave 模式 @ <96>
        }
        orez_destroy_relations(relations);
        # 释放冗白字符集 @ <69>
        delete_syntax_tree(root);
        # 释放符号集 @ <15>
        g_option_context_free(option_context);
        return 0;
}
```

8 后端：面向排版

Orez 的织成过程输出的是以 YAML 格式表达的语法树结构。若文学化程序源码是基于某种排版语言写成，需要将 YAML 格式的语法树结构转化为使用该排版语言表述的源文档，继而由该排版语言的编译器，或解释器，或浏览器构造文档的排版结果。本节假设文学化程序源码以 Markdown 标记语言写就，用 Python 3 编写面向 Markdown 标记语言的 Orez 后端，以此作为 Orez 后端开发的示例。

8.1 PyYAML

据我所知，在 Ubuntu 这样的 Linux 发行版上，可以用以下命令为 Python3 安装 YAML 解析库：

```
$ sudo apt install python3-yaml
```

其他 Linux 发行版的软件仓库里应该也能找到面向 Python 3 的 YAML 库。倘若熟悉 Python 3 的包管理器 pip，应该也能通过它安装 YAML 解析库。

现在假设系统中已存在 Python3 和 YAML 库，以下 Python 3 代码可读取并解析 orez 输出的 YAML 文件，并将解析结果打印出来：

```
import sys
import yaml

if __name__=="__main__":
    f = open(sys.argv[1])
    x = yaml.full_load(f)
    print(x)
    f.close()
```

假设将上述代码保存为 orez-md-demo.py 文件，为了验证它是否工作，先准备一份简单的文学化程序源码文件 foo.orz：

```
有名片段 foo:

@ foo # [text]
我是 foo。
    # bar @
@

有名片段 bar:

@ bar #
我是 bar。
@

bar 的一个同名片段:

@ bar # +
    我也是 bar。
@
```

使用以下命令可将 foo.orz 转化为 foo.yaml：

```
$ orez -w foo.orz -o foo.yaml
```

foo.yaml 内容如下：

```
- type: 0
  content: |-
    '有名片段 foo:'
- type: 1
  name: |-
    'foo'
  hash: 'foo'
  language: 'text'
  content:
    - text: |-
        '
        我是 foo。
            '
    - reference:
        name: |-
          'bar'
        hash: 'bar'
        ids:
          - 1
          - 2
- type: 0
  content: |-
    '有名片段 bar:'
- type: 1
  name: |-
    'bar'
  hash: 'bar'
  id: 1
  language: 'text'
  content:
    - text: |-
        '
        我是 bar。'
  emissions:
    - emission:
        name: |-
```

```
          'foo'
        hash: 'foo'
- type: 0
  content: |-
    'bar 的一个同名片段:'
- type: 1
  name: |-
    'bar'
  hash: 'bar'
  id: 2
  language: 'text'
  operator: '+'
  content:
    - text: |-
        '
           我也是 bar。'
  emissions:
    - emission:
        name: |-
          'foo'
        hash: 'foo'
```

YAML 标记语言很简单，若对其不熟悉，可阅读我温习它时所写的一份文档：

- https://liyanrui.github.io/output/libyaml-tutorial/yaml-intro.html

再结合 **6.3** 节的 Orez 输出格式说明，足以理解上述 foo.yaml 的全部内容。

使用以下命令执行上述的 orez-md-demo.py 脚本，便可解析 foo.yaml 并输出解析结果：

```
$ python3 orez-md-demo.py foo.yaml
```

```
[{'type': 0, 'content': "'有名片段 foo:'"},
 {'type': 1, 'name': "'foo'", 'hash': 'foo', 'language': 'text',
  'content': [{'text': "'\n我是 foo。\n    '"},
 {'reference': {'name': "'bar'", 'hash': 'bar', 'ids': [1, 2]}}]},
 {'type': 0, 'content': "'有名片段 bar:'"},
 {'type': 1, 'name': "'bar'", 'hash': 'bar', 'id': 1, 'language': 'text',
  'content': [{'text': "'\n我是 bar。'"}],
  'emissions': [{'emission': {'name': "'foo'", 'hash': 'foo'}}]},
 {'type': 0, 'content': "'有名片段 bar 的一个同名片段:'"},
 {'type': 1, 'name': "'bar'", 'hash': 'bar', 'id': 2, 'language': 'text',
  'operator': '+', 'content': [{'text': "'\n    我也是 bar。'"}],
```

```
  'emissions': [{'emission': {'name': "'foo'", 'hash': 'foo'}}]}]
```

orez-md-demo.py 输出的数据是 Python 字典对象构成的序列（数组）。

8.2 管道

现在，基于 Orez 的文学编程方式，对 orez-md-demo.py 略加改动，便可使之能够从标准输入获得 YAML 数据：

```
@ PyYAML 用法示例 #
import sys
import yaml

if __name__=="__main__":
    x = None
    # 解析 YAML 数据 -> x @ <101>
    print(x)
```

被上述代码片段引用的片段定义如下：

```
@ 解析 YAML 数据 -> x #
if len(sys.argv) > 1:
    f = open(sys.argv[1])
    x = yaml.full_load(f)
    f.close()
else:
    x = yaml.safe_load(sys.stdin)
=> PyYAML 用法示例 <101>
=> 访问片段序列 <102>
=> 访问片段序列：改进 <103>
=> Markdown 后端雏形 <104>
=> Markdown 后端的半成品 <107>
=> 简陋的 Markdown 后端脚本 <110>
```

这份文档的源文件为 orez.orz，使用以下命令便可从中抽取上述 Python 代码，将其保存为 orez-md-demo.py：

```
$ orez -t orez.orz -e "PyYAML 用法示例" -o orez-md-demo.py
```

由于 orez 在省略 -o 参数时，能够将 YAML 数据发送到标准输出，于是可使之与具备从标准输入获得 YAML 数据的 orez-md-demo.py 脚本实现管道衔接：

```
$ orez -w foo.orz | python3 orez-md-demo.py
```

即 orez 写入标准输出端的 YAML 数据，通过管道传送至 orez-md-demo.py 的标准输入端，从而避免 YAML 格式的中间文件的出现。以上命令粗略说明了 orez 程序的基本用法及其在文档排版方面与后端脚本的关系。

8.3 片段序列

PyYAML 对 orez 生成的 YAML 文档解析所得结果是一个序列（数组），其中各项皆为字典对象。以下代码可遍历该序列，访问所有的无名片段和有名片段：

```
@ 访问片段序列 #
import sys
import yaml

if __name__=="__main__":
    x = None
    # 解析 YAML 数据 -> x @ <101>
    for e in x:
        if e["type"] == 0: # 访问无名片段
            print(e["content"])
        elif e["type"] == 1: # 访问有名片段
            if "id" in e:
                print("%s %d" % (e["name"], e["id"]))
            else:
                print(e["name"])
        else:
                sys.stderr.write("Unknown snippet type!\n")
```

使用 orez 程序抽取上述代码，并将其作用于上一节的 foo.yaml，可输出以下内容：

```
'有名片段 foo: '
'foo'
'有名片段 bar: '
'bar' 1
'有名片段 bar 的一个同名片段: '
'bar' 2
```

任一有名片段，只有存在多个与之同名的片段时，才拥有 `id` 项，上述 Python 代码表明了这一点。同名片段通过 `id` 的值予以区分。对于命令 `orez -w` 而言，多个同名片段在文档中依序出现，`id` 即它们的序号，从 1 开始。

需要注意的是，上述代码输出的无名片段的内容以及有名片段的名字皆被单引号包裹，这种情况仅在 YAML 数据中对应的字符串允许换行时出现，例如

```
content: |-
    '有名片段 bar 的一个同名片段: '
```

content 的值便是允许换行的字符串，因为标识“|-”中的“|”表示允许字符串换行，“-”表示消除字符串末尾的换行符（假设它存在）。对于该类型的字符串，PyYAML 将其解析为单引号包裹的形式，但单引号对我们使用 PyYAML 解析 orez 输出的 YAML 数据没有意义，需使用 Python 字符串类型的 strip 方法予以消除：

```
@ 访问片段序列: 改进 #
import sys
import yaml

if __name__=="__main__":
    x = None
    # 解析 YAML 数据 -> x @ <101>
    for e in x:
        if e["type"] == 0: # 访问无名片段
            print(e["content"].strip("'"))
        elif e["type"] == 1: # 访问有名片段
            if "id" in e:
                print("%s %d" % (e["name"].strip("'"), e["id"]))
            else:
                print(e["name"].strip("'"))
        else:
            sys.stderr.write("Unknown snippet type!\n")
```

使用 orez 抽取上述代码，将其作用于 foo.yaml，可得

```
有名片段 foo:
foo
有名片段 bar:
bar 1
有名片段 bar 的一个同名片段:
bar 2
```

可对上述 Python 代码继续进行扩展，访问 orez 生成的 YAML 数据中有名片段的其他项，例如 language、hash 以及 content 等。需要注意，有名片段的 content 的值也像上述代码中的 x，是由一组字典对象构成的序列。

8.4 雏形

Orez 实现的文学编程，有名片段的内容仅由程序源码和片段引用两种成分构成。对于程序源码，Markdown 排版标记是

```编程语言
源码内容
```

例如

```C
int main(void) {
        printf("Hello world!\n");
        return 0;
}
```

对于有名片段中可能存在的片段引用，通常需要一些特别的排版处理，Markdown 的源码排版标记会抑制这些处理，因而并不适于排版 Orez 的有名片段内容。鉴于在 Markdown 文本中可直接使用 HTML 标记，可以尝试基于 HTML 的 pre 和 span 标记排版有名片段。

以下代码在遍历片段序列过程中，原样输出无名片段的内容，以 typepre 块的形式输出了有名片段的名字和内容，并使用 span 标记对片段名字加以渲染：

```
@ Markdown 后端雏形 #
import sys
import yaml

if __name__=="__main__":
    x = None
    # 解析 YAML 数据 -> x @ <101>
    for e in x:
        if e["type"] == 0:
            # 排版无名片段 @ <105>
        elif e["type"] == 1:
            print("<pre>")
            # 排版片段名字
            print('<span style="color:darkred">@ %s #</span>'
                    % (e["name"].strip("'")), end = "")
            for h in e["content"]:
                if "text" in h: # 排版程序源码
                    print(h["text"].strip("'"), end = "")
                else: # 排版片段引用
                    print('<span style="color:darkblue"># %s @</span>'
                            % (h["reference"]["name"].strip("'")), end = "")
            print("\n</pre>\n")
        else:
```

```
            sys.stderr.write("Unknown snippet type!\n")
```

上述代码中，无名片段的排版代码现在可以固定不变，如下：

```
@ 排版无名片段 #
print(e["content"].strip("'"))
print("") # 增加一个空行。在Markdown 语法中，空行表示分段。
=> Markdown 后端雏形 <104>
=> Markdown 后端的半成品 <107>
=> 简陋的 Markdown 后端脚本 <110>
```

还需要注意的是，上述代码中 `print` 函数的 `end = ""` 参数，其用途是抑制 `print` 在其输出的内容最后添加换行符。

用 orez 程序抽取上述 Markdown 后端雏形代码，将其作用于前文的 foo.yaml，结果为以下嵌入了一些 HTML 标记的 Markdown 文本：

```
有名片段 foo:

<pre>
<span style="color:darkred">@ foo #</span>
我是 foo。
    <span style="color:darkblue"># bar @</span>
</pre>

有名片段 bar:

<pre>
<span style="color:darkred">@ bar #</span>
我是 bar。
</pre>

有名片段 bar 的一个同名片段:

<pre>
<span style="color:darkred">@ bar #</span>
    我也是 bar。
</pre>
```

若将上述 Markdown 文本保存为 foo.md 文件，使用 pandoc 程序将其转化为 HTML 文件，即网页：

```
$ pandoc foo.md -o foo.html
```

在网页浏览器中，foo.html 可被渲染为图 **1** 所示的样式。

图 1　Markdown 后端雏形构造的排版结果

你的系统没有 pandoc 程序？它是开源软件，很容易获得并安装。倘若是 Linux 系统，例如 Ubuntu，只需

```
$ sudo apt install pandoc
```

便可得到 pandoc。

8.5 引用

由于一个有名片段可以通过片段的名字引用某个片段，若排版语言支持链接和锚点，通过它们可构造有名片段的交叉引用，从而实现片段之间的跳转。由于在 Markdown 文本中可直接使用 HTML 标记，基于 HTML 的链接标记和锚点可实现有名片段的引用排版。

HTML 标记的 `id` 属性可作为锚点。在文档的其他片段里，通过将链接指向某个锚点，便可实现片段之间的跳转。例如，在文档的某个片段设置一个带 `id` 的 `pre` 块：

```
<pre id="foo">
... ... ...
</pre>
```

在文档的另一处 `pre` 块中，使用链接标记 `a` 便可将连接指向上述 `pre` 块：

```
<pre id="bar">
... ... ...
<a href="#foo">该链接可跳到 id 为 foo 的 pre 块</a>
</pre>
```

要使得 Markdown 后端脚本支持上述形式的交叉引用，首先需要为每个有名片段转换成的 `<pre>...</pre>` 块增加 `id` 属性，使其具有唯一的标识。Orez 输出的 YAML 数据中，有

名片段的 hash 值和 id 值可用于构造该标识，并保证其唯一性。假设 e 为有名片段，以下代码可将其输出为带锚点和链接的 HTML 文本：

```
@ 输出带有锚点和链接的有名片段 #
hash = e["hash"]
if "id" in e: # 存在同名结点，需要将用 hash 值和序号构造锚点
    print('<pre id="%s-%d">' % (hash, e["id"]))
else: # 没有同名结点，直接用 hash 值锚点
    print('<pre id="%s">' % (hash))
# 输出片段名字及其内容 @ <107>
print("\n</pre>\n")
=> Markdown 后端的半成品 <107>
```

片段名字及其内容的排版代码如下：

```
@ 输出片段名字及其内容 #
print('<span style="color:darkred">@ %s #</span>'
        % (e["name"].strip("'")), end = "")
for h in e["content"]:
    if "text" in h:
        print(h["text"].strip("'"), end = "")
    else: # 将引用的片段构造为链接
        r = h["reference"]
        name = r["name"].strip("'")
        hash = r["hash"]
        if "ids" in r:
            print('<span># %s @</span>' % name, end = "")
            for id in r["ids"]:
                print(' <a href="#%s-%d">[%d]</a>' % (hash, id, id), end = "")
        else:
            print('<a href="#%s"># %s @</a>' % (hash, name), end = "")
=> 输出带有锚点和链接的有名片段 <107>
=> 简陋的 Markdown 后端脚本 <110>
```

将上述代码片段嵌入片段序列遍历过程，便可得到 Markdown 后端的半成品：

```
@ Markdown 后端的半成品 #
import sys
import yaml

if __name__=="__main__":
    x = None
```

```
    # 解析 YAML 数据 -> x @ <101>
    for e in x:
        if e["type"] == 0:
            # 排版无名片段 @ <105>
        elif e["type"] == 1:
            # 输出带有锚点和链接的有名片段 @ <107>
```

使用 orez 程序抽取上述代码，将其作用于 foo.yaml，结果为以下 Markdown 文本：

```
有名片段 foo:

<pre id="foo">
<span style="color:darkred">@ foo #</span>
我是 foo。
    <span># bar @</span> <a href="#bar-1">[1]</a> <a href="#bar-2">[2]</a>
</pre>

有名片段 bar:

<pre id="bar-1">
<span style="color:darkred">@ bar #</span>
我是 bar。
</pre>

有名片段 bar 的一个同名片段:

<pre id="bar-2">
<span style="color:darkred">@ bar #</span>
    我也是 bar。
</pre>
```

使用 pandoc 将上述 Markdown 文本转化为 HTML 格式，在网页浏览器中的排版结果如图 **2** 所示。在该排版结果中，若点击 `foo` 片段中片段引用之后的 `[1]` 和 `[2]`，可分别跳转两个 `bar` 片段。

8.6 回跳

上一节实现了从片段 `foo` 中对片段 `bar` 的引用跳转到片段 `bar`，那么从 `bar` 可否跳回 `foo` 呢？

答案是能。对于任一被其他有名片段引用的片段，Orez 为其生成的 YAML 数据项中必定含有一个名为 `emissions` 的序列，例如上文中的 foo.yaml 中第 1 个 `bar` 片段，其 `emissions` 的序列为

图 2　有名片段的引用

```
emissions:
  - emission:
      name: |-
        'foo'
      hash: 'foo'
```

基于该信息，便可为该 bar 片段构造跳回片段 foo 的链接，构造过程与上一节为片段引用构造锚点和连接相似。假设有名片段 e，为其构造引用者链接的过程如下：

```
@ 输出片段回跳连接 #
if "emissions" in e:
    for p in e["emissions"]:
        emi = p["emission"]
        emi_name = emi["name"].strip("'")
        emi_hash = emi["hash"]
        print("") # 构造空行
        if "id" in emi:
            emi_link = '=> <a href="#{0}-{1}">{2}</a>'
            print('=> <a href="#%s-%d">%s</a>'
                  % (emi_hash, emi["id"], emi_name), end = "")
        else:
            print('=> <a href="#%s">%s</a>'
                  % (emi_hash, emi_name), end = "")
=> 简陋的 Markdown 后端脚本 <110>
```

将上述代码片段与上一节的示例代码结合，便得到了一份简陋的 Markdown 后端脚本：

```
@ 简陋的 Markdown 后端脚本 #
import sys
import yaml

if __name__=="__main__":
    x = None
    # 解析 YAML 数据 -> x @ <101>
    for e in x:
        if e["type"] == 0:
            # 排版无名片段 @ <105>
        elif e["type"] == 1:
            hash = e["hash"]
            if "id" in e: # 存在同名结点，需要将用 hash 值和序号构造锚点
                print('<pre id="%s-%d">' % (hash, e["id"]))
            else: # 没有同名结点，直接用 hash 值锚点
                print('<pre id="%s">' % (hash))
            # 输出片段名字及其内容 @ <107>
            # 输出片段回跳连接 @ <109>
            print("\n</pre>\n")
```

使用 orez 程序抽取上述代码，将其作用于 foo.yaml，可得以下 Markdown 文本：

```
有名片段 foo:

<pre id="foo">
<span style="color:darkred">@ foo #</span>
我是 foo。
    <span># bar @</span> <a href="#bar-1">[1]</a> <a href="#bar-2">[2]</a>
</pre>

有名片段 bar:

<pre id="bar-1">
<span style="color:darkred">@ bar #</span>
我是 bar。
=> <a href="#foo">foo</a>
</pre>

有名片段 bar 的一个同名片段:

<pre id="bar-2">
<span style="color:darkred">@ bar #</span>
```

```
    我也是 bar。
=> <a href="#foo">foo</a>
</pre>
```

在上述的 Markdown 和 HTML 混合文本中出现了基于 HTML 链接标记的三个有名片段的交叉引用，排版结果如图 **3** 所示。

有名片段 foo:

```
@ foo #
我是 foo。
    # bar @ [1] [2]
```

有名片段 bar:

```
@ bar #
我是 bar。
=> foo
```

有名片段 bar 的一个同名片段:

```
@ bar #
    我也是 bar。
=> foo
```

图 3　交叉引用

8.7 总结

本节以 Python 3 和 PyYAML 库解析 Orez 输出的 YAML 数据为例，对如何为 Orez 的文档排版编写后端程序进行了简要介绍，所实现的示例较为简陋，例如未考虑源码片段渲染，HTML 样式定制等问题。更为完善的 Markdown 后端程序的实现可参考随 orez 源码发布的 orez-md 脚本：

- https://github.com/liyanrui/orez/blob/master/orez-md

不过，本节的示例表明，倘若你熟悉某种排版语言或标记，并尝试理解 Orez 输出的 YAML 数据格式，为该排版语言实现后端程序并非难事。

附录

辅助函数 `am_i_here` 用于在一个字符串区间 `[a, b]` 内判断 `*cursor` 所指位置与 `me` 等长的片段 x 是否与 `me` 相同。`dir` 表示 `*cursor` 的走向，其值为负，表示 `*cursor` 此时指向 x 的尾字节，否则表示 `*cursor` 指向 x 的首字节。若 `am_i_here` 匹配成功，若 `dir` 为负值，将 `*cursor` 修改为指向 x 的首字节，否则将 `*cursor` 修改为指向 x 的尾字节。

```
@ 文本处理辅助函数 #
static bool am_i_here(char *a, char *b, char **cursor, char *me, int dir)
{
        size_t n = strlen(me);
        char *p = *cursor;
        /* 若 cursor 超范围，无需匹配 */
        if (dir < 0 && p < a + n - 1) return false;
        if (dir >= 0 && p > b - n + 1) return false;
        /* 否则 */
        bool result = false;
        char *t = malloc((n + 1) * sizeof(char));
        for (size_t i = 0; i < n; i++) {
                t[i] = (dir < 0) ? *(p + i - n + 1) : *(p + i);
        }
        t[n] = '\0';
        if (strcmp(t, me) == 0) {
                result = true;
                *cursor = (dir < 0) ? (p - n + 1) : (p + n - 1);
        }
        free(t);
        return result;
}
=> 分段过程所需的辅助函数 <17>
```

辅助函数 make_padding 用于构造指定长度的空白字符串，定义如下：

```
@ 文本处理辅助函数 # +
static GString *make_padding(size_t len)
{
        GString *padding = g_string_new(NULL);
        for (size_t i = 0; i < len; i++) {
                g_string_append_c(padding, ' ');
        }
        return padding;
}
=> 分段过程所需的辅助函数 <17>
```