

# 任务零：理解目标文件格式

在开始实现链接器之前，我们需要先理解输入和输出的数据格式。你可能在CSAPP第七章中见过ELF（Executable and Linkable Format）格式——这是Linux系统上可执行文件和目标文件的标准格式。ELF格式非常强大，但也相当复杂。它包含了大量的细节和边界情况，这些对于理解链接的核心概念并不是必需的。

为了让你专注于链接器的本质工作，而不是被文件格式的细节淹没，我们设计了FLE（Friendly Linking Executable）格式。这是一个简化的、人类友好的格式，它保留了链接过程中所有重要的信息，但用更直观的方式组织。

## 从一个例子开始

让我们从一个简单的C程序开始理解：

```
// test.c
int message[2] = {1, 2}; // 全局数组

static int helper(int x) { // 静态函数，只在本文件可见
    return x + message[0];
}

int main() { // 程序入口
    return helper(42);
}
```

这个程序包含了几种不同类型的内容：代码（`helper` 和 `main` 函数）、数据（`message` 数组）、符号（函数名和变量名）、以及引用关系（`helper` 使用了 `message`）。编译器需要把这些信息组织成某种格式，以便链接器能够理解。

当你用我们提供的编译工具处理这个文件时：

```
./cc test.c -o test.o
```

会生成一个 `test.fo` 文件。让我们看看它的内容。

## FLE格式文件的结构

FLE格式使用JSON来表示目标文件的结构，这意味着你可以用任何文本编辑器打开它，直接看到里面的内容。这是 `test.fo` 的样子：

```
{
  "type": ".obj", // 这是一个目标文件
  "shdrs": [
    // 节头：描述每个节的元数据
    {
      "name": ".text", // 代码段
      "type": 1, // 节类型
      "flags": 1, // 节属性
      "addr": 0, // 在内存中的地址（目标文件中为0，链接时确定）
      "offset": 0, // 在文件中的位置
      "size": 36 // 节的大小（字节）
    }
  ]
}
```

```

    },
    {
        "name": ".data",    // 数据段
        "type": 1,
        "flags": 1,
        "addr": 0,
        "offset": 36,
        "size": 8
    },
    {
        "name": ".bss",    // 未初始化数据段
        "type": 8,
        "flags": 9,
        "addr": 0,
        "offset": 44,
        "size": 0
    }
],
".text": [
    "📌: helper 20 0",    // 局部符号定义
    "1234: 55 48 89 e5 89 7d fc 8b 15", // 机器码
    "?: .abs32s(message + 0)",    // 需要重定位
    "1234: 8b 45 fc 01 d0 5d c3",    // 更多机器码
    "📍: main 16 20",    // 全局符号定义
    "1234: 55 48 89 e5 bf 2a 00 00 00 e8 de ff ff ff 5d c3"
],
".data": [
    "📍: message 8 0",    // 全局符号定义
    "1234: 01 00 00 00 02 00 00 00" // 数组内容: 1和2
],
".bss": []
}

```

第一眼看到这个格式，你可能会注意到那些表情符号。它们是FLE格式的核心特性，用来标记不同类型的信息，让你一眼就能看出每一行是什么。

## 理解表情符号标记

让我们逐一理解这些标记的含义。

**1234 表示机器码或数据。**这些是实际的字节内容，以十六进制表示。比如 55 48 89 e5 是函数序言的机器码，对应汇编指令 `push %rbp; mov %rsp, %rbp`。在 .data 节中， 01 00 00 00 02 00 00 00 是数组 {1, 2} 的二进制表示（小端序，每个整数占4字节）。

**📍 表示全局符号。**这些是可以被其他文件引用的符号。在上面的例子中， `main` 和 `message` 都是全局符号——其他文件可能会调用 `main` 函数或访问 `message` 数组。符号后面跟着两个数字：第一个是符号的大小（字节），第二个是符号在其所在节中的偏移量。比如 📍: main 16 20 表示 `main` 函数大小为 16 字节，从 .text 节的偏移 20 处开始。

**📌 表示局部符号。**这些只在当前文件内可见。`helper` 函数被声明为 `static`，所以它是一个局部符号。其他文件无法直接引用它，这是封装性的一部分。

**🔗 表示弱符号。**你会在任务四中详细了解到弱符号，现在只需要知道它表示“可被覆盖的符号”——如果有同名的普通全局符号，链接器会选择那个而不是这个。

**? 表示需要重定位的位置。**这是链接器需要特别关注的标记。它表示"这里需要一个地址，但现在还不知道具体是多少"。比如 `? : .abs32s(message + 0)` 的意思是："这里需要 `message` 的绝对地址，以32位有符号整数的形式填充。"重定位类型（这里是 `.abs32s`）告诉链接器如何计算和填充这个地址。

## 节头的作用

你可能注意到文件开头有个 `shdrs` 数组，这是节头（section headers）的列表。每个节头描述一个节的基本信息。

让我们理解节头的各个字段：

`name` 字段标识节的名称。`.text` 是代码段，`.data` 是已初始化的数据段，`.bss` 是未初始化的数据段，`.rodata` 是只读数据段（比如字符串常量）。这些名称是约定俗成的，链接器会根据名称判断如何处理每个节。

`type` 字段表示节的类型。值为1表示这是一个包含实际内容的节（`SHT_PROGBITS`），值为8表示这是一个不占用文件空间的节（`SHT_NOBITS`），典型的就 `.bss` 节。

`size` 字段是节的大小。注意 `.text` 节是36字节，`.data` 节是8字节，而 `.bss` 节是0字节。`.bss` 的大小为0是因为它不存储实际内容——它只是一个"预约"，告诉链接器这里需要分配一块内存，但不需要在文件中存储任何字节。

`offset` 和 `addr` 字段在目标文件中通常不太重要。`offset` 记录节在文件中的位置，主要是为了解析方便。`addr` 在目标文件中为0，因为目标文件还不知道自己会被加载到内存的哪里——这是链接器的工作。

## 从文件到内存：FLEObject结构

FLE格式文件是存储在磁盘上的，人类可读的表示。但程序运行时，我们需要一个在内存中的、方便操作的数据结构。这就是 `FLEObject`。

在你实现链接器的过程中，你主要和 `FLEObject` 打交道。我们的框架会在读取文件时自动将FLE格式转换为 `FLEObject`，在写入文件时自动将 `FLEObject` 转换回FLE格式。这个转换过程被称为序列化和反序列化。

这个结构体定义在 `include/fle.hpp` 中：

```
struct FLEObject {
    std::string name;                // 对象名称（通常是文件名）
    std::string type;                // ".obj" 或 ".exe"
    std::map<std::string, FLESection> sections; // 节名 -> 节内容
    std::vector<Symbol> symbols;     // 全局符号表
    std::vector<ProgramHeader> phdrs; // 程序头（仅可执行文件）
    std::vector<SectionHeader> shdrs; // 节头
    size_t entry = 0;                // 程序入口点（仅可执行文件）
};
```

注意这个结构体和FLE文件格式的对应关系。文件中的 `type` 字段对应 `FLEObject::type`，`shdrs` 数组对应 `FLEObject::shdrs` 向量。但也有一些重要的差异。

最大的差异在于符号和重定位的组织方式。在FLE文件中，符号定义和重定位项是"内联"的——它们直接出现在节内容中，用表情符号标记。但在 `FLEObject` 中，它们被提取到独立的数据结构中。

## 符号表的表示

在内存中，每个符号是一个 `Symbol` 结构体：

```
struct Symbol {
    SymbolType type;    // LOCAL (🔑), GLOBAL (🌐), WEAK (🌀), 或 UNDEF (未定义)
    std::string section; // 符号所在的节名，如 ".text"
    size_t offset;      // 符号在节内的偏移
    size_t size;        // 符号的大小
    std::string name;    // 符号名称
};
```

回到我们的例子。FLE文件中的这一行：

```
"🌐: main 16 20"
```

在 `FLEObject` 的符号表中对应这样一个条目：

```
Symbol {
    .type = SymbolType::GLOBAL, // 🌐 表示全局符号
    .section = ".text",        // 定义在代码段
    .offset = 20,               // 在代码段偏移20处
    .size = 16,                // 大小16字节
    .name = "main"             // 函数名
}
```

这个 `Symbol` 对象本质上是在说：“有一个名为 `main` 的全局符号，它指向从 `.text` 节偏移20字节开始的、长度为16字节的一段内容。”

所有符号——无论是全局的、局部的还是弱符号——都收集在 `FLEObject::symbols` 向量中。这样你在处理符号解析时，可以方便地遍历所有符号，不需要再解析文件格式。

## 重定位表的表示

类似地，重定位信息也被提取到独立的结构中。每个节可能包含需要重定位的位置，这些信息存储在 `FLESection::relocs` 中：

```
struct Relocation {
    RelocationType type; // 重定位类型
    size_t offset;       // 重定位位置在节内的偏移
    std::string symbol;  // 目标符号名
    int64_t addend;      // 附加值
};
```

FLE文件中的这一行：

```
"? : .abs32s(message + 0)"
```

对应这样一个 `Relocation` 对象：

```
Relocation {
    .type = RelocationType::R_X86_64_32S, // .abs32s对应的类型
    .offset = 9,                          // 在.text节偏移9处
    .symbol = "message",                  // 目标符号
    .addend = 0                           // 附加值
}
```

这个重定位项在说："在 `.text` 节偏移9字节的位置，需要填入 `message` 符号的地址，采用32位有符号绝对寻址的方式。"

## 节内容的表示

在FLE文件中，节的内容是一系列带标记的行。在 `FLEObject` 中，每个节的实际字节内容存储在 `FLESection::data` 中，这是一个 `std::vector<uint8_t>` ——简单的字节数组。

这里有一个重要的细节需要注意。在FLE文件中，需要重定位的位置用 `?` 标记，那个位置**没有占位的字节**。但在 `FLESection::data` 中，那些位置会有占位的0字节。

让我们用一个具体例子来说明。FLE文件中 `.text` 节的内容是：

```
"12: 55 48 89 e5 89 7d fc 8b 15",
"? : .abs32s(message + 0)",
"14: 8b 45 fc 01 d0 5d c3"
```

在 `FLESection` 的 `data` 字段中，这会被表示为：

```
std::vector<uint8_t> {
    0x55, 0x48, 0x89, 0xe5, 0x89, 0x7d, 0xfc, 0x8b, 0x15,
    0x00, 0x00, 0x00, 0x00, // 重定位位置的占位符（4个字节）
    0x8b, 0x45, 0xfc, 0x01, 0xd0, 0x5d, 0xc3
}
```

重定位信息则单独存储在 `relocs` 向量中，指明"偏移9处需要重定位"。这样的设计使得节的内容始终是连续的字节数组，而重定位信息单独管理，互不干扰。

## Ready?

理解了FLE格式后，你就掌握了链接器工作的"语言"。在接下来的任务中，你会读取FLE格式的目标文件，操作 `FLEObject` 结构，最终生成新的FLE格式的可执行文件。

目标文件告诉链接器："我有这些代码和数据，我定义了这些符号，我需要这些外部符号。"链接器的工作就是综合所有输入文件的信息，解决依赖关系，安排内存布局，填充地址引用，最终生成一个完整的程序。

### 💡 Tip

如果你在阅读框架代码时对某些C++语法感到困惑，可以参考[C程序员的C++实用指南](#)。这份指南解释了实验中会用到的C++特性，不需要从头学习整个C++语言。