编译与链接详解：ELF 格式目标文件的结构与符号分配

在现代编译系统中，编译生成的目标文件（`.o` 文件）以 ELF（Executable and Linkable Format）格式存储，包含程序的逻辑结构、符号信息及重定位信息。本文将深入剖析 ELF 格式目标文件的组成部分及符号处理机制，详细解释为什么编译阶段符号的虚拟地址未分配，以及链接阶段如何完成符号解析。

---

1. ELF 目标文件的组成

ELF 格式是现代操作系统中广泛采用的二进制文件格式，`.o` 文件作为编译器的中间输出，其结构主要由以下部分组成：

1.1 ELF 文件头（ELF Header）

- **位置**：文件的起始部分。
- **作用**：描述文件的整体信息，包括文件类型、目标架构、段偏移、节偏移等。
- **关键字段**：
  - `Type`：文件类型（`REL` 表示可重定位文件，`EXEC` 表示可执行文件，`DYN` 表示共享库）。
  - `Machine`：目标架构（如 x86 或 x86_64）。
  - `Entry point address`：程序入口点地址（对于目标文件，通常为 `0x0`）。

1.2 `.text` 段（代码段）

- **作用**：存储程序的机器码指令。
- **特性**：只读、可执行。
- **示例内容**：
  - 编译后的函数实现，例如 `main()` 或 `sum()` 的指令。
- **内存分布**：在编译阶段，`.text` 段中的指令地址以相对于段起始位置的偏移表示，尚未分配全局虚拟地址。

1.3 `.data` 段（已初始化数据段）

- **作用**：存储已初始化的全局变量和静态变量。
- **特性**：可读写，包含初始值。
- **示例内容**：
  - 示例代码中的 `int data = 20;` 存储在 `.data` 段中。

1.4 `.bss` 段（未初始化数据段）

- **作用**：存储未初始化的全局变量和静态变量。
- **特性**：只分配内存，不占用目标文件的存储空间。
- **示例内容**：

- 示例代码中的 `extern int gdata;` 会在 `.bss` 段分配内存。

1.5 `.symtab`（符号表）

- **作用**：记录目标文件中的所有符号（函数、变量等）的信息。
- **关键字段**：
  - 符号名称。
  - 符号类型（如函数、变量、常量）。
  - 符号的定义位置（段名称及段内偏移地址）。
  - 符号的绑定（`LOCAL` 表示仅在当前目标文件中可见，`GLOBAL` 表示全局可见）。
  - 符号的状态：
    - `UND`：未定义符号，需要链接器在其他目标文件中解析。
    - 已定义符号的段地址。

1.6 `.rel.text`（重定位表）

- **作用**：记录 `.text` 段中所有需要重定位的符号引用。
- **示例内容**：
  - 如果 `.text` 段中引用了 `gdata` 或调用了 `sum()`，其地址在编译阶段尚未确定，需要记录在重定位表中供链接器解析。

1.7 节头表（Section Table）

- **作用**：记录 ELF 文件中每个段的信息，包括段名称、类型、偏移地址、大小等。
- **关键字段**：
  - 段名称（如 `.text`、`.data`）。
  - 段类型（如 `PROGBITS` 表示存储数据，`NOBITS` 表示仅分配内存）。
  - 段在文件中的偏移地址和长度。

---

2. 编译阶段的符号处理

在编译生成目标文件的过程中，**所有符号的虚拟地址均未分配**，符号地址仅相对于段起始位置。以下是原因：

1. **未分配最终虚拟地址**
   - 目标文件是中间文件，尚未完成程序的整体布局。
   - 符号地址仅以段内偏移的形式表示，最终虚拟地址需由链接器分配。
2. **符号状态的记录**
   - 未定义的符号（如 `gdata` 和 `sum()`）在符号表中标记为 `UND`，需要链接器在其他目标文件中解析。
   - 定义在本文件的符号记录在 `.text` 或 `.data` 等段中。
3. **重定位机制**

- 符号引用记录在重定位表中，链接器通过解析重定位表修正符号地址。

---

3. 符号在目标文件中的作用

3.1 符号的分类

1. **变量符号**
   - 包括已初始化变量（存储于 `.data` 段）和未初始化变量（存储于 `.bss` 段）。
   - 示例：

```
int data = 10;  // 符号 `data` 是已定义变量。
extern int gdata;  // 符号 `gdata` 是未定义变量。
```

2. **函数符号**
   - 函数的名称存储在 `.text` 段。
   - 示例：

```
int sum(int a, int b) {
    return a + b;
}
```

3. **未定义符号**
   - 当前目标文件中声明但未定义的符号。
   - 示例：

```
extern int gdata;  // 未定义符号。
int sum(int, int);  // 未定义符号。
```

---

4. 工具查看 ELF 文件

4.1 查看 ELF 文件头

```
readelf -h main.o
```

4.2 查看段信息

```
readelf -S main.o
```

4.3 查看符号表

```
readelf -s main.o
```

4.4 查看重定位表

```
readelf -r main.o
```

通过这些工具可以详细分析目标文件的结构和符号状态，验证符号是否定义、引用及其重定位情况。

---

5. 示例：符号解析与地址分配

以下代码展示了目标文件中的符号处理机制：

示例代码

```
// main.cpp
extern int gdata;
int sum(int, int);

int main() {
    return sum(10, gdata);
}
```

```
// sum.cpp
int gdata = 10;
int sum(int a, int b) {
    return a + b;
}
```

编译命令

```
g++ -c main.cpp -o main.o
g++ -c sum.cpp -o sum.o
```

查看符号表

```
readelf -s main.o
```

---

6. 链接阶段的符号处理

链接器将多个目标文件合并，并完成以下任务：

1. **符号解析**
    - 将 `main.o` 中的未定义符号 `gdata` 和 `sum` 与 `sum.o` 中的定义匹配。
2. **地址分配**
    - 为所有符号分配全局虚拟地址。
    - 修正重定位表中记录的符号地址。

---

7. 总结

- **ELF 文件结构**：
    - 包含 ELF 文件头、段表、符号表、重定位表等。
    - 提供完整的程序逻辑和符号引用信息。
- **符号地址未分配的原因**：
    - 在编译阶段，符号地址仅以段内偏移表示。
    - 链接阶段才会为符号分配全局虚拟地址。
- **符号表的作用**：
    - 记录符号的定义、引用及其重定位信息。
    - 是链接器解析符号和修正地址的核心数据。