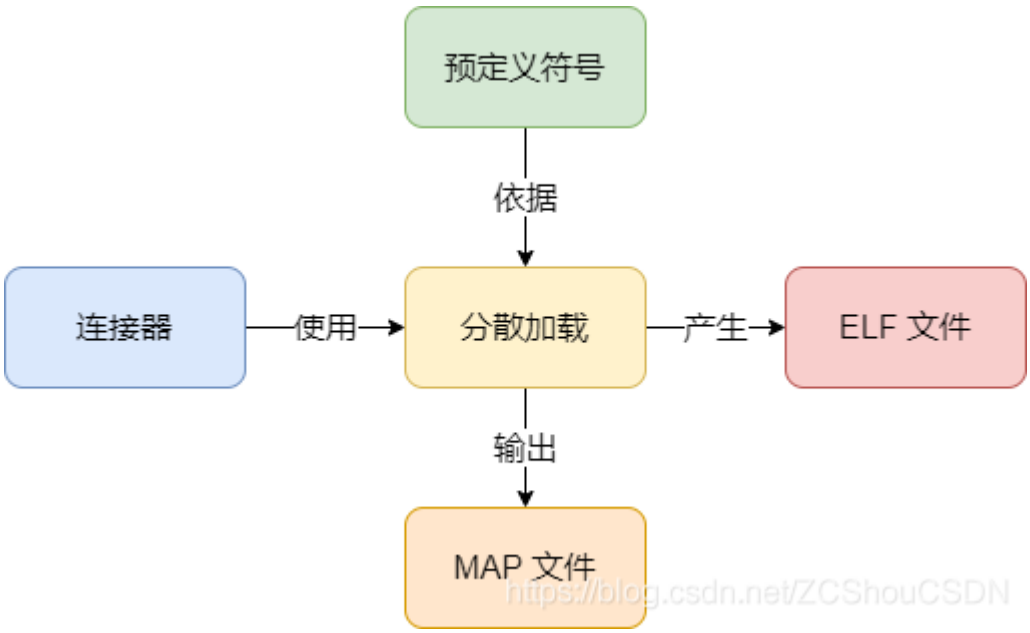
ARM 之十三 armlink（Keil） 分散加载机制详解 及 分散加载文件的编写 - zxddesk

cnblogs.com/zxdplay/p/17764095.html

分散加载是在连接阶段指定连接器如何生成镜像文件的。所以在看这篇文章之前

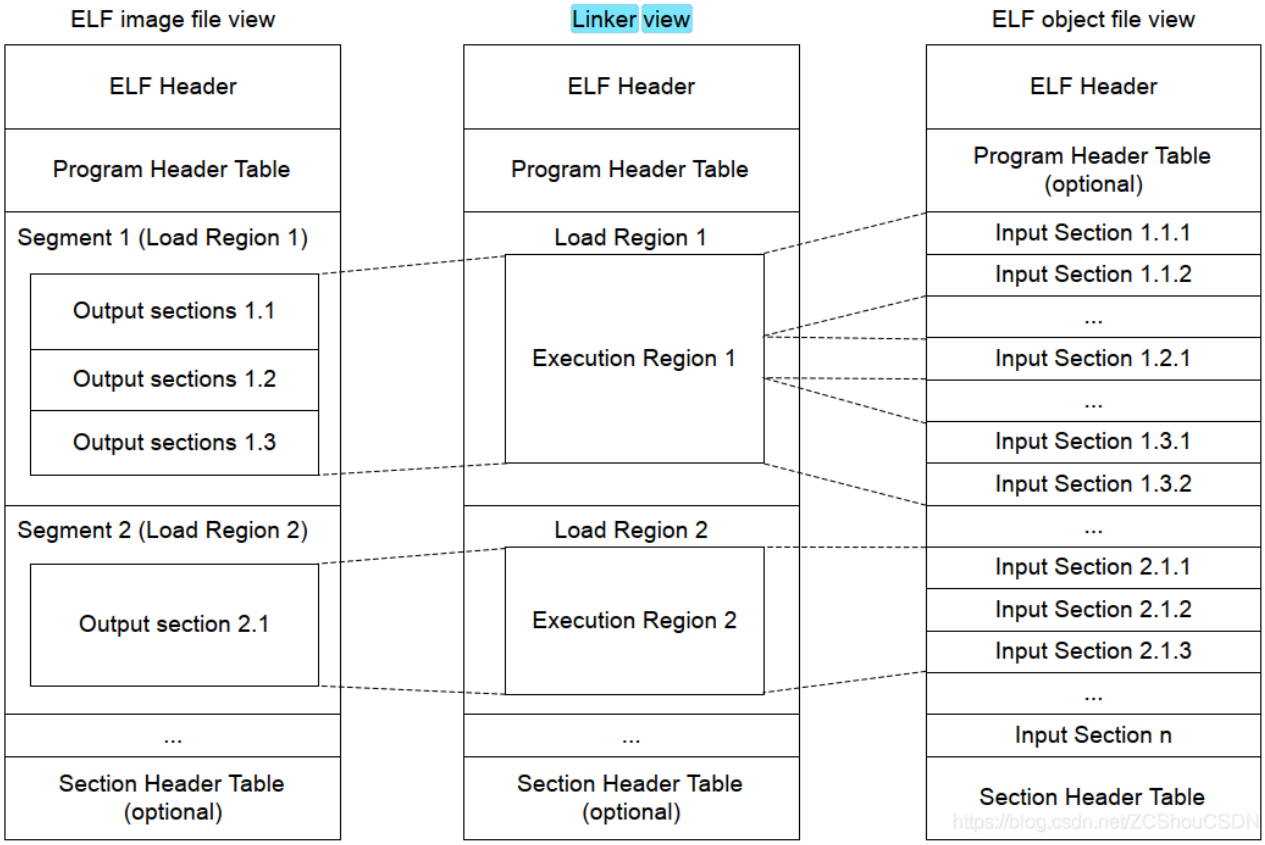
1. 需要对 ARM ELF 文件有一定的了解。了解什么是域（Region）、节（Section，也称为节区）、段（Segment）、镜像（Image）、镜像文件（Image File）等概念。
2. 需要对编译、链接过程有一定的了解。如下图：



3. 更详细的内容，请参考 ARM 连接器手册

ARM ELF 镜像的结构

ARM ELF 镜像由节（Section），域（Region）和段（Segment）组成，并且每个链接阶段都有一个不同的镜像视图。下面我们简单来说明一下，以方便理解分散加载。更详细的介绍，参见博文 ARM 之一 ELF 文件、镜像（Image）文件、可执行文件、对象文件详解。



- **ELF object file view (linker input)：** ELF 对象文件视图由输入节组成。ELF 对象文件可以是：
  - 一种可重定位的文件，它保存适合与其他目标文件链接以创建可执行或共享目标文件的代码和数据。
  - 包含代码和数据的共享对象文件
- **Linker view：** 链接器有两个用于程序地址空间的视图（加载视图和执行视图），当存在重叠的、位置无关的和可重定位的程序片段（代码或数据）时，这两个视图变得不同。
  - 程序片段的加载地址是链接器希望外部代理（例如程序加载器，动态链接器或调试器）从 ELF 文件复制的片段所在的地址。 这个地址可能不是片段执行的地址。
  - 程序片段的执行地址是每当它参与程序的执行时，链接器期望片段驻留的目标地址。如果一个片段是位置无关的或可重定位的，它的执行地址可以在执行过程中变化。
- **ELF image file view (linker output)：** ELF 镜像文件视图由程序段和输出节组成：
  - 一个加载域对应一个程序段
  - 一个执行域包含一个或多个以下输出节：
    - RO section.
    - RW section.
    - XO section.
    - ZI section.

一个或多个执行域构成一个加载域。每个域可以具有不同的加载和执行地址。默认情况下，链接器将以特定顺序将输入节放置到执行域中。默认的排序规则如下（如果需要，我们也可以指定其他排列顺序）：

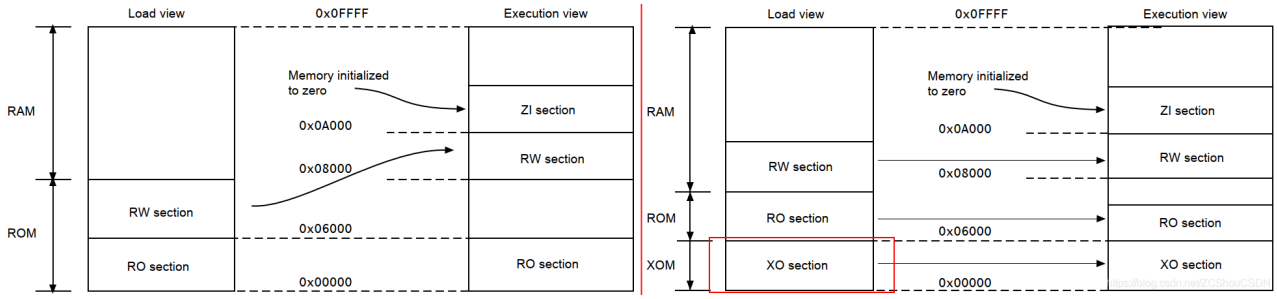
1. 按照属性的如下顺序进行排序：
    - 只读的代码（Read-only code）
    - 只读的数据（Read-only data）
    - 可读写代码（Read-write code）
    - 可读写数据（Read-write data）
    - ZI 数据（Zero-initialized data）
2. 如果具有相同的属性，则通过输入节的名字排序。名字是按照字符的 ASCII 码的顺序排序的（区分大小写的）。
3. 如果具有相同的属性和节名字，默认情况下，按照 `armlink` 处理节的顺序来排序。 可以使用 `FIRST` 或 `LAST` 执行域属性来改变排序。

每个输入节的基地址由链接器定义的排序顺序决定，并在包含它的输出节中正确对齐。链接器会为执行域中的每个属性（如果有）生成一个输出节：

- 如果执行域包含一个或多个 XO 节，则生成一个仅执行（XO）节。
- 如果执行域包含只读代码或数据，则生成一个 RO 节。
- 如果执行域包含可读写代码或数据，则生成一个 RW 节。
- 如果执行域包含零初始化数据，则生成一个 ZI 节

镜像的加载域和执行域

镜像的各个域在加载时是被放置在系统存储器中的。 在执行镜像之前，可能必须将一些域移动到它们的执行地址，并创建 ZI 输出节。例如，初始化的 RW 数据可能必须从 ROM 中的加载地址复制到 RAM 中的执行地址。为了可以灵活的处理这种情况，ARM 定义了如下两个视图：



- **Load view：** 根据镜像在加载到内存中时所位于的地址（镜像执行开始前的位置），描述每个镜像域和节。
- **Execution view：** 根据镜像执行过程中所位于的地址，描述每个镜像域和节。

下面是对这两种视图的一个对比：

Load	Description	Execution	Description
加载地址	在包含分节或域的镜像开始执行之前，要加载到内存中的节或者域的地址。节或者非跟域的加载地址和他们执行地址可以不同	执行地址	当包含某个节或域的镜像被执行时，该节或域所在的地址
加载域	加载域描述在加载地址空间中连续内存块的布局	执行域	执行域描述在执行地址空间中的连续内存块的布局

Image entry points

镜像中的入口点就是镜像中的一个位置（地址），该位置（地址）会被加载到 PC 寄存器。 它是程序执行开始的位置。 虽然镜像中可以有多个入口点，但在链接时只能指定一个入口点。并非每个 ELF 文件都必须有入口点。 不允许在单个 ELF 文件中存在多个入口点。

对于嵌入式 Cortex-M 核的程序，程序的执行是从复位向量所在的位置（地址）开始执行。复位向量会被加载到 PC 寄存器中，且复位向量的位置（地址）并不固定。 通常，复位向量指向 CMSIS Reset_Handler 函数。

有两种不同类型的入口点：

- **初始化入口点（Initial entry point）：** 镜像的初始入口点是存储在 ELF 头文件中的单个值。 对于那些需要由操作系统或引导加载程序加载到 RAM 中的程序，加载程序通过将控制转移到镜像中的初始入口点来启动镜像执行。一个镜像只能有一个初始化入口点。初始入口点可以是 ENTRY 指令设置的入口点之一，但不是必需的。
- **ENTRY 指令指定的入口点：** ENTRY 指令可以为镜像从多个可能的入口点中选择其中一个。每个镜像只能有一个入口点。您可以在汇编程序文件中使用 `ENTRY` 指令在对象中创建入口点。 在嵌入式系统中，该指令的典型用途是标记进入处理器异常向量（例如 RESET，IRQ 和 FIQ）的代码。该指令使用 `ENTRY` 关键字标记输出代码部分，该关键字指示链接器在执行未使用的部分消除时不删除该部分。对于 C/C++ 程序，C 库中的 `__main` 就是入口点。

如果加载程序要使用嵌入式的镜像，则它必须在标头中指定一个初始入口点。 使用 `--entry` 命令行选项选择入口点。

映射符号

映射符号由编译器和汇编器生成，以识别文字池边界处的代码和数据之间的内联转换，以及 ARM 代码和 Thumb 代码之间的内联转换。例如 ARM/Thumb 交互操作胶合代码。映射符号有如下这些：

- $a：一系列 ARM 指令的开始

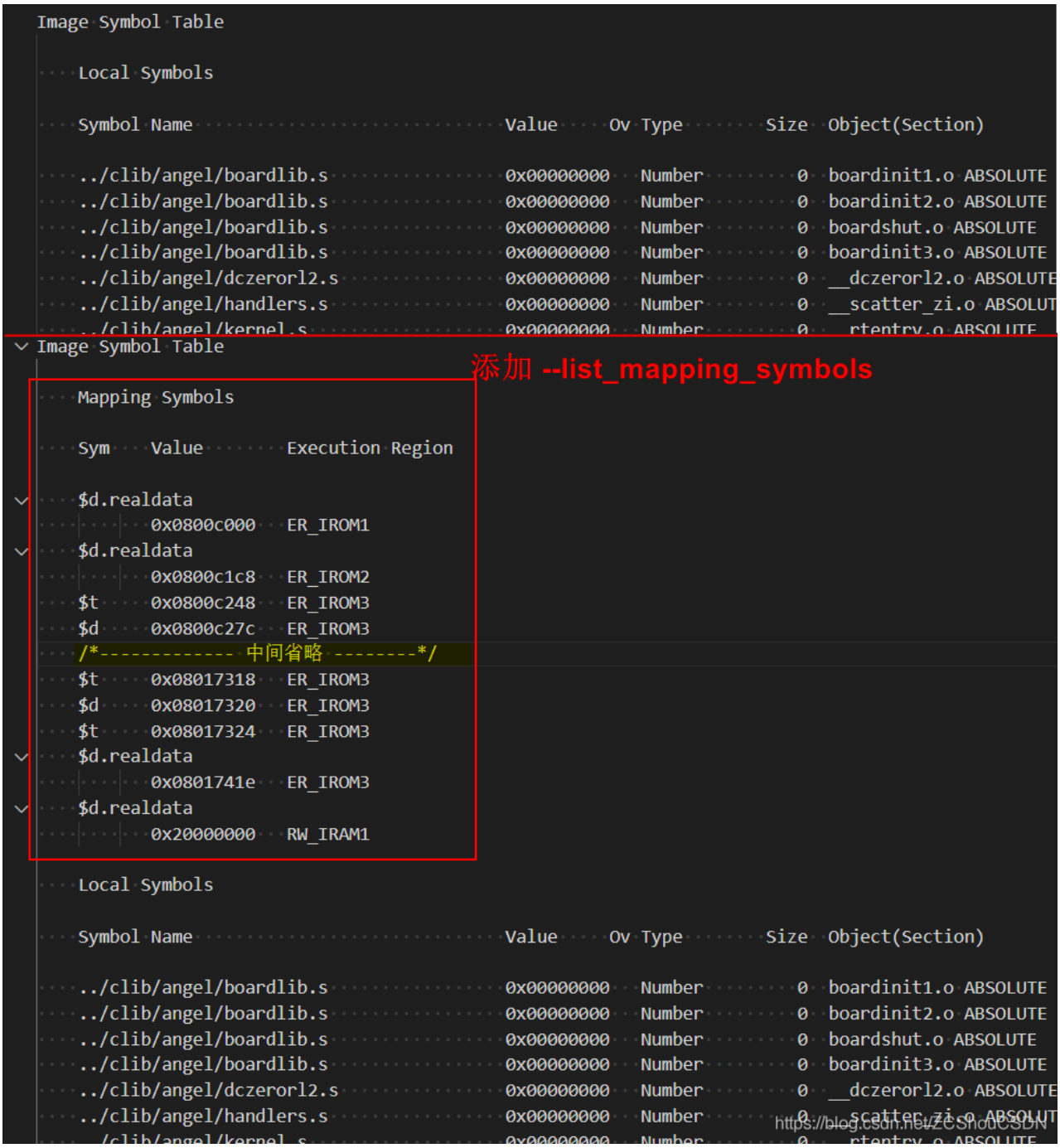
- $t：一系列 Thumb 指令的开始
- $t.x：一系列 ThumbEE 指令的开始
- $d：一系列数据项的开始，如文字池

> 1. **文字池** 是代码段中存放常量数据的区域。因为没有一条指令可以生成一个 4 字节的常量，因此编译器将这些常量放到文字池中，然后生成从文字池加载这些常量的代码。
> 2. ARM/Thumb交互（ARM/Thumb interworking）是指对汇编语言和 C/C++ 语言的 ARM 和 Thumb 代码进行连接的方法，它进行两种状态（ARM 和 Thumb）间的切换。
> 3. 胶合代码（Veneer）：在进行 ARM/Thumb 交互时，有时需使用额外的代码，这些代码被称为 胶合代码（Veneer）。
> 4. AAPCS 定义了 ARM 和 Thumb 过程调用的标准。

此外，`armlink` 还会生成 `$d.realdata` 映射符号，以告诉 `fromelf` 该数据是来自非可执行节区。因此，`fromelf -z` 输出的代码和数据大小与 `armlink --info sizes` 的输出相同。例如：

```
Code (inc. data)   RO Data
   x          y          z
```

在以上的示例中，y 标记为 $d，RO Data 标记为 $d.realdata。如果启用了`armlink` 的 `--list_mapping_symbols` 参数，则会在 map 文件中有体现，如下图：



请注意：

1. 以字符 $v 开头的符号是与 VFP 相关的映射符号，在使用 VFP 构建目标时可能会输出。 避免在源代码中使用以 $v 开头的符号。
2. 使用 `fromelf --elf --strip=localsymbols` 命令修改可执行镜像会从镜像中删除所有映射符号。
3. 注意：`--list_mapping_symbols` 和 `--no_list_mapping_symbols` 属于 `--symbols` 输出的额外信息。例如：上面的 Keil 配置中我们只使用了 `--symbols`。

链接器预定义符号

当链接器创建镜像文件时，它会创建一些 ARM 预定义的与域或者节相关的符号。这些符号就代表了链接器创建创建镜像的依据。下面我们就重点来了解一下这些符号。

链接器定义了一些 ARM 保留的符号，我们可以在需要时访问这些符号。***这些符号是包含 `$$` 字符序列的符号以及所有其他包含 `$$` 字符序列的外部名称。*** 您可以导入这些符号地址，并将它们作为汇编语言程序的可重定位地址使用，或者将它们作为 C 或 C++ 源代码中的 extern 符号来引用。

- 如果使用 `--strict` 编译器命令行选项，则编译器不接受包含 $ 的符号名称。 要重新启用支持，请在编译器命令行中包含 `--dollar` 选项。
- ***链接器定义的符号只有在代码引用它们时才会生成。***
- 如果存在仅执行（XO）节，则链接器定义的符号受以下约束：
  - 不能对没有 XO 节的域或者空域定义 XO 连接器定义符号
  - 不能对仅包含 RO 节的域定义 XO 连接器定义符号
  - 对于仅包含 XO 节的域，不能定义 RO 连接器定义符号

引入到 C/C++

可以通过 **值引用** 或 **地址引用** 这两种方式将链接器定义的符号导入到的 C 或 C++ 源代码中来供我们使用：

- 值引用：`extern unsigned int symbol_name;`
- 地址引用：`extern void *symbol_name;`

注意，如果将符号声明为 int 类型的值引用，则必须使用寻址操作符（&）来获得正确的值，如下例所示：

```
// Importing a linker-defined symbol
extern unsigned int Image$$ZI$$Limit;
config.heap_base = (unsigned int) &Image$$ZI$$Limit;

//Importing symbols that define a ZI output section
extern unsigned int Image$$ZI$$Length;
extern char Image$$ZI$$Base[];
memset(Image$$ZI$$Base, 0, (unsigned int)&Image$$Length);
```

引入到 汇编

可以使用指令 `IMPORT` 将连接器定义的符号引入到 ARM 汇编文件中来供我们使用：

```
IMPORT |Image$$ZI$$Limit|
…
zi_limit DCD |Image$$ZI$$Limit|

LDR r1, zi_limit
```

域相关的符号

链接器为镜像文件中的每个域生成不同类型的与域相关的符号，我们可以根据需要访问这些符号。域相关的符号主要有以下两种：

- `Image$$` 或者 `Load$$` 开头的符号，用于各执行域
- `Load$$LR$$` 开头的符号，用于各加载域

如果未使用分散加载文件，则会以默认的 region 名称来生成域相关的符号。链接器默认的域名称如下：

- **ER_XO**： 用于仅执行属性的执行域（如果存在）。
- **ER_RO**： 用于只读执行域。
- **ER_RW**： 用于可读写执行域。
- **ER_ZI**： 用于零初始化的执行域。

  可以将这些名称插入 `Image$$` 和 `Load$$` 中以获取所需的地址，例如：
  `Load$$ER_RO$$Base` 就是只读域的基地址。

使用分散加载时，连接器将使用分散加载文件中的名称来生成各种域相关的符号。分散加载文件可以实现以下功能：

- 命名镜像中的所有执行域，并提供他们的加载和执行地址。
- 定义堆栈和堆。 链接器还会生成特殊的栈和堆符号。

> 1. 镜像的 ZI 输出节不是静态创建的，而是在运行时自动动态创建的。 因此，ZI 输出节没有加载地址符号。
> 2. 符号 `Load$$region_name` 仅适用于执行域。`Load$$LR$$load_region_name` 符号仅适用于加载域。

执行域符号 `Image$$`

链接器为镜像中存在的每个执行域生成符号 `Image$$`。下表列出了链接器为镜像中存在的每个执行域生成的符号。 初始化 C 库后，所有符号都指向执行地址。

Symbol	Description
`Image$$region_name$$Base`	域的执行地址
`Image$$region_name$$Length`	执行域长度（以字节为单位），不包括 ZI 的长度。
`Image$$region_name$$Limit`	超出执行域中非 ZI 部分末尾的字节的地址
`Image$$region_name$$RO$$Base`	域中的输出节 RO 的执行地址
`Image$$region_name$$RO$$Length`	Length of the RO output section in bytes.

Symbol	Description
`Image$$region_name$$RO$$Limit`	Address of the byte beyond the end of the RO output section in the execution region.
`Image$$region_name$$RW$$Base`	Execution address of the RW output section in this region.
`Image$$region_name$$RW$$Length`	Length of the RW output section in bytes.
`Image$$region_name$$RW$$Limit`	Address of the byte beyond the end of the RW output section in the execution region.
`Image$$region_name$$XO$$Base`	Execution address of the XO output section in this region.
`Image$$region_name$$XO$$Length`	Length of the XO output section in bytes.
`Image$$region_name$$XO$$Limit`	Address of the byte beyond the end of the XO output section in the execution region.
`Image$$region_name$$ZI$$Base`	Execution address of the ZI output section in this region.
`Image$$region_name$$ZI$$Length`	Length of the ZI output section in bytes.
`Image$$region_name$$ZI$$Limit`	Address of the byte beyond the end of the ZI output section in the execution region.

执行域符号 `Load$$`

链接器为镜像中存在的每个执行域生成符号 `Load$$`。下表列出了链接器为镜像中存在的每个 `Load$$` 执行域生成的符号。 初始化 C 库后，所有符号都指向加载地址。

Symbol	Description
`Load$$region_name$$Base`	Load address of the region.
`Load$$region_name$$Length`	Region length in bytes.
`Load$$region_name$$Limit`	Address of the byte beyond the end of the execution region.
`Load$$region_name$$RO$$Base`	Address of the RO output section in this execution region.
`Load$$region_name$$RO$$Length`	Length of the RO output section in bytes.

Symbol	Description
`Load$$region_name$$RO$$Limit`	Address of the byte beyond the end of the RO output section in the execution region.
`Load$$region_name$$RW$$Base`	Address of the RW output section in this execution region.
`Load$$region_name$$RW$$Length`	Length of the RW output section in bytes.
`Load$$region_name$$RW$$Limit`	Address of the byte beyond the end of the RW output section in the execution region.
`Load$$region_name$$XO$$Base`	Address of the XO output section in this execution region.
`Load$$region_name$$XO$$Length`	Length of the XO output section in bytes.
`Load$$region_name$$XO$$Limit`	Address of the byte beyond the end of the XO output section in the execution region.
`Load$$region_name$$ZI$$Base`	Load address of the ZI output section in this execution region.
`Load$$region_name$$ZI$$Length`	Load length of the ZI output section in bytes. The Load Length of ZI is zero unless region_name has the ZEROPAD scatter-loading keyword set. If ZEROPAD is set then: Load Length = `Image$$region_name$$ZI$$Length`
`Load$$region_name$$ZI$$Limit`	Load address of the byte beyond the end of the ZI output section in the execution region.

初始化 C 库之前，此表中的所有符号均指加载地址。请注意以下事项：

- 这些符号是绝对的，因为相对于节的符号只能有执行地址。
- 这些符号考虑了 RW 压缩。
- 从 RW 压缩执行域引用的链接器定义的符号必须是在应用 RW 压缩之前可解析的符号。
- 如果链接器检测到从 RW 压缩域到依赖于 RW 压缩的链接器定义符号的重定位，则链接器将禁用当前域的压缩。
- Limit 和 Length 值影响写入文件的任何零初始化数据。 使用 ZEROPAD 分散加载关键字时，零初始化数据将写入文件。

加载域符号 `Load$$LR$$`

链接器为镜像中存在的每个加载区生成符号 `Load$$LR$$`。一个 `Load$$LR$$` 加载域可以包含许多执行域，因此没有单独的 `$$RO` 和 `$$RW` 部分。下表显示了链接器为镜像中存在的每个 `Load$$LR$$` 加载域生成的符号。

Symbol	Description
`Load$$LR$$load_region_name$$Base`	Address of the load region.
`Load$$LR$$load_region_name$$Length`	Length of the load region.
`Load$$LR$$load_region_name$$Limit`	Address of the byte beyond the end of the load region.

节相关的符号

与节相关的符号是链接器在创建没有使用分散加载文件的镜像时生成的符号。链接器会为输出和输入节生成不同类型的与节相关的符号：

- **镜像符号（Image symbols）**（如果不使用分散加载来创建简单的镜像文件）。 简单的镜像文件具有多达四个输出节（XO，RO，RW 和 ZI），用于生成相应的执行域。
- **输入节符号（Input section symbols）** 镜像中存在的每个输入节的输入节符号（Input section symbols）

链接器首先按属性 RO，RW 或 ZI 对执行域内的节进行排序，然后按名称排序。 例如，所有 .text 节都放在一个连续的块中。 具有相同属性和名称的连续块部分称为合并节。

> 1. ARM 建议优先使用与域相关的符号，而不是与节相关的符号。

镜像符号

当您不使用分散加载文件来创建简单镜像时，镜像符号将由链接器生成。我们常用的 Keil 会默认生成分散加载文件的，所以基本没有不使用分散加载文件的情况。下表显示了镜像符号：

Symbol	Section type	Description
`Image$$RO$$Base`	Output	Address of the start of the RO output section.
`Image$$RO$$Limit`	Output	Address of the first byte beyond the end of the RO output section.
`Image$$RW$$Base`	Output	Address of the start of the RW output section.

Symbol	Section type	Description
`Image$$RW$$Limit`	Output	Address of the byte beyond the end of the ZI output section. (The choice of the end of the ZI region rather than the end of the RW region is to maintain compatibility with legacy code.)
`Image$$ZI$$Base`	Output	Address of the start of the ZI output section.
`Image$$ZI$$Limit`	Output	Address of the byte beyond the end of the ZI output section.

> 如果存在 XO 节，那么还包含符号 `Image$$XO$$Base` 和 `Image$$XO$$Limit`

如果使用了分散加载文件，则上面这些镜像符号都将称为未定义的。 如果在代码中访问这些符号中的任何一个，则必须将它们视为**弱引用**。`__user_setup_stackheap()` 的标准实现中就使用 `Image$$ZI$$Limit` 中的值，因此，如果您使用的是分散加载文件，则必须手动设置堆栈和堆。 方法主要有以下两种：

- 在分散文件中使用下列方法之一
  - 定义名为 `ARM_LIB_STACK` 和 `ARM_LIB_HEAP` 的单独的栈和单独的堆域。
  - 定义包含堆栈和堆的组合域，名为 `ARM_LIB_STACKHEAP`。
- 通过重新实现 `__user_setup_stackheap()` 来设置堆和堆栈边界。（在我们的项目中的 .s 启动文件中，是这种方法）

后文的 堆栈 章节，我会详细介绍这两种方法。

输入节符号

链接器为镜像中存在的每个输入节生成输入节符号。下表显示了链接器定义的输入节符号：

Symbol	Section type	Description
`SectionName$$Base`	Input	Address of the start of the consolidated section called SectionName.
`SectionName$$Length`	Input	Length of the consolidated section called SectionName (in bytes).
`SectionName$$Limit`	Input	Address of the byte beyond the end of the consolidated section called SectionName.

如果在的代码引用输入节符号，则表示希望将镜像中具有相同名称的所有输入节都连续放置在镜像内存映射中。如果分散加载文件不连续地放置输入节，则链接器会发出错误。 这是因为在非连续存储器上将导致 Base 符号和 Limit 符号是不明确的。

分散加载机制

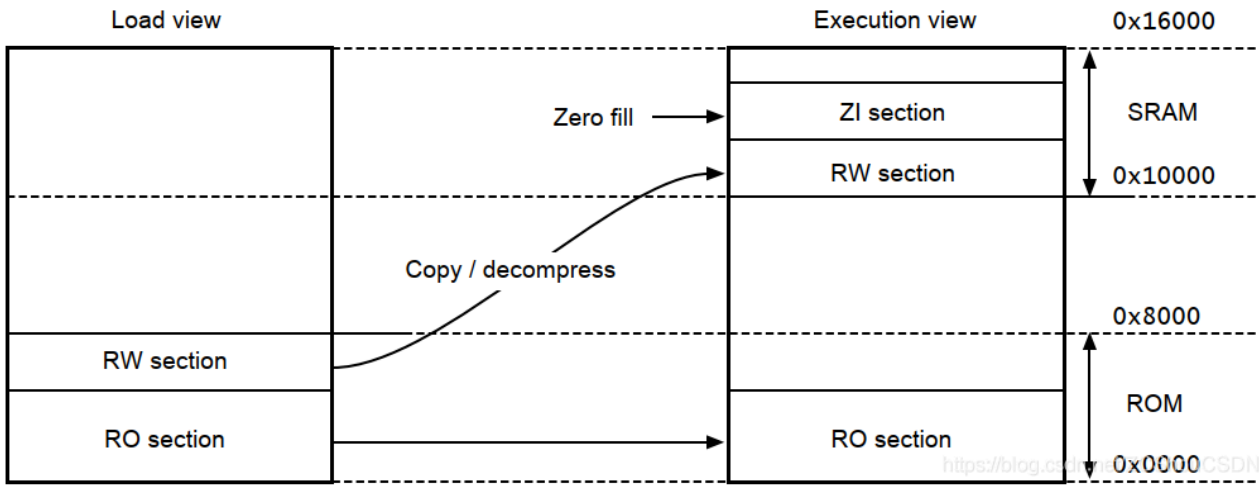
镜像的内存映射由域和输出节组成。内存映射中的每个域可以具有不同的加载和执行地址。分散加载机制是 ARM 连接器 `armlink` 定义的一种特性。通过该机制，我们可以让连接器完全按照我们自己的描述来组织镜像文件的内存映射，以适应复杂的嵌入式环境。

> 1. 所谓的分散加载是指，在加载和执行时，多个内存域分散在内存映射中。

分散加载机制规定需要把我们的要求描述在一个文本文件中，这个文件被称为 **分散加载文件**。 `armlink` 通过参数 `--scatter "分散加载文件名"` 来引用我们的分散加载文件。

> 注意，中文中我们通常称为 **分散加载文件**。但是在 ARM 官方文档中只有 （分散文件）Scatter File

对于简单的内存映射，可以使用以下与内存映射相关的连接器命令行（`--partial`、`--ro_base`、`--rw_base`、`--ropi`、`--rosplit`、`--split`、`--reloc`、`--startup`、`--xo_base`、`--zi_base`）参数来放置代码和数据。例如，如下的内存映射，我们可以使用连接器参数 `armlink --ro_base 0x0 --rw_base 0x10000` 来实现。

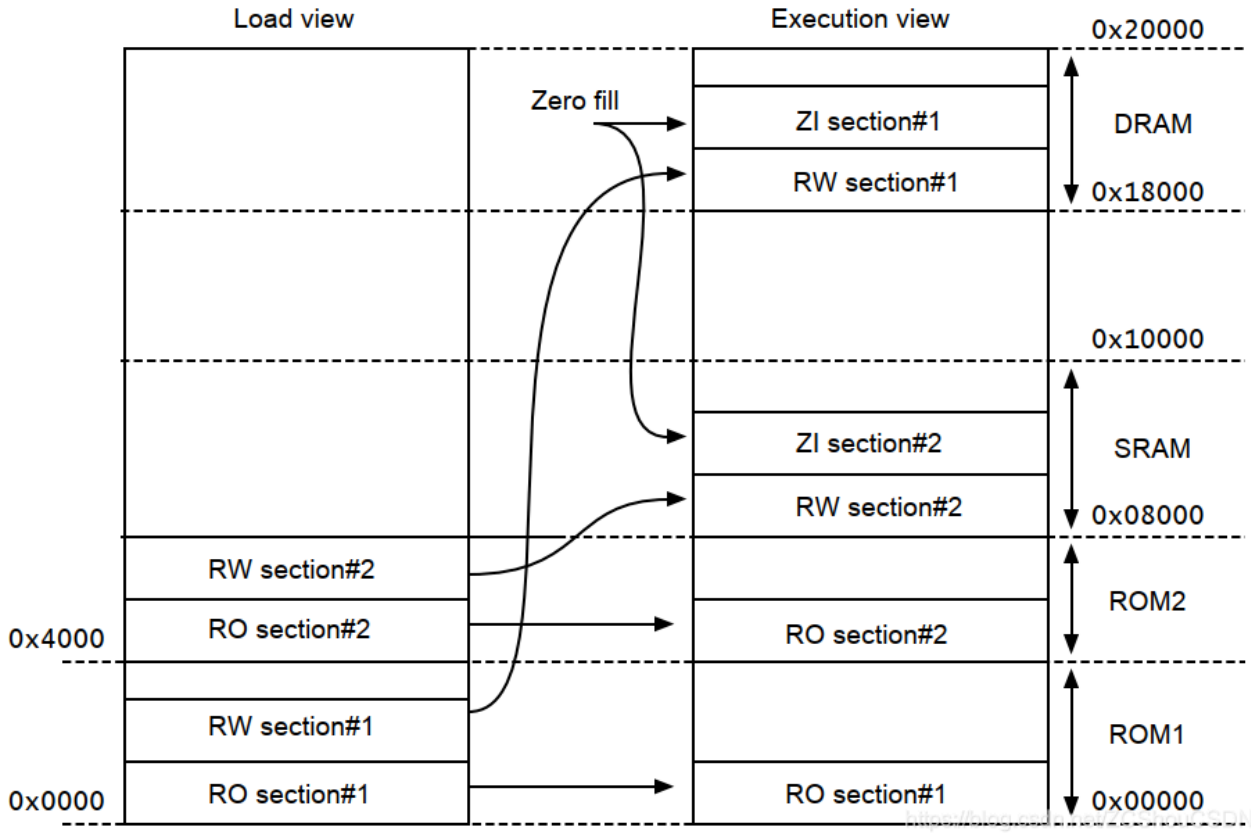


当然，也可以使用分散加载来实现，下面的示例显示了相应的分散加载描述，该描述将对象文件中的段加载到内存中：

```
LOAD_ROM 0x0000 0x8000       ; Name of load region (LOAD_ROM),
                             ; Start address for load region (0x0000),
                             ; Maximum size of load region (0x8000)
{
    EXEC_ROM 0x0000 0x8000   ; Name of first exec region (EXEC_ROM),
                             ; Start address for exec region (0x0000),
                             ; Maximum size of first exec region (0x8000)
    {
        * (+RO)              ; Place all code and RO data into
                             ; this exec region
    }
    SRAM 0x10000 0x6000      ; Name of second exec region (SRAM),
                             ; Start address of second exec region (0x10000),
                             ; Maximum size of second exec region (0x6000)
    {
        * (+RW, +ZI)         ; Place all RW and ZI data into
                             ; this exec region
    }
}
```

但需要注意，参数 `--scatter "分散加载文件名"` 是不能与上面这些一起使用的！

分散加载通常只用于具有复杂内存映射的镜像中。对于具有复杂内存映射的镜像，不能仅使用链接器命令行选项指定内存映射。



如上图所示的内存映射，下面的示例显示了对应的分散加载描述，该描述将来自 `program1.o` 和 `program2.o` 文件的段加载到内存中：

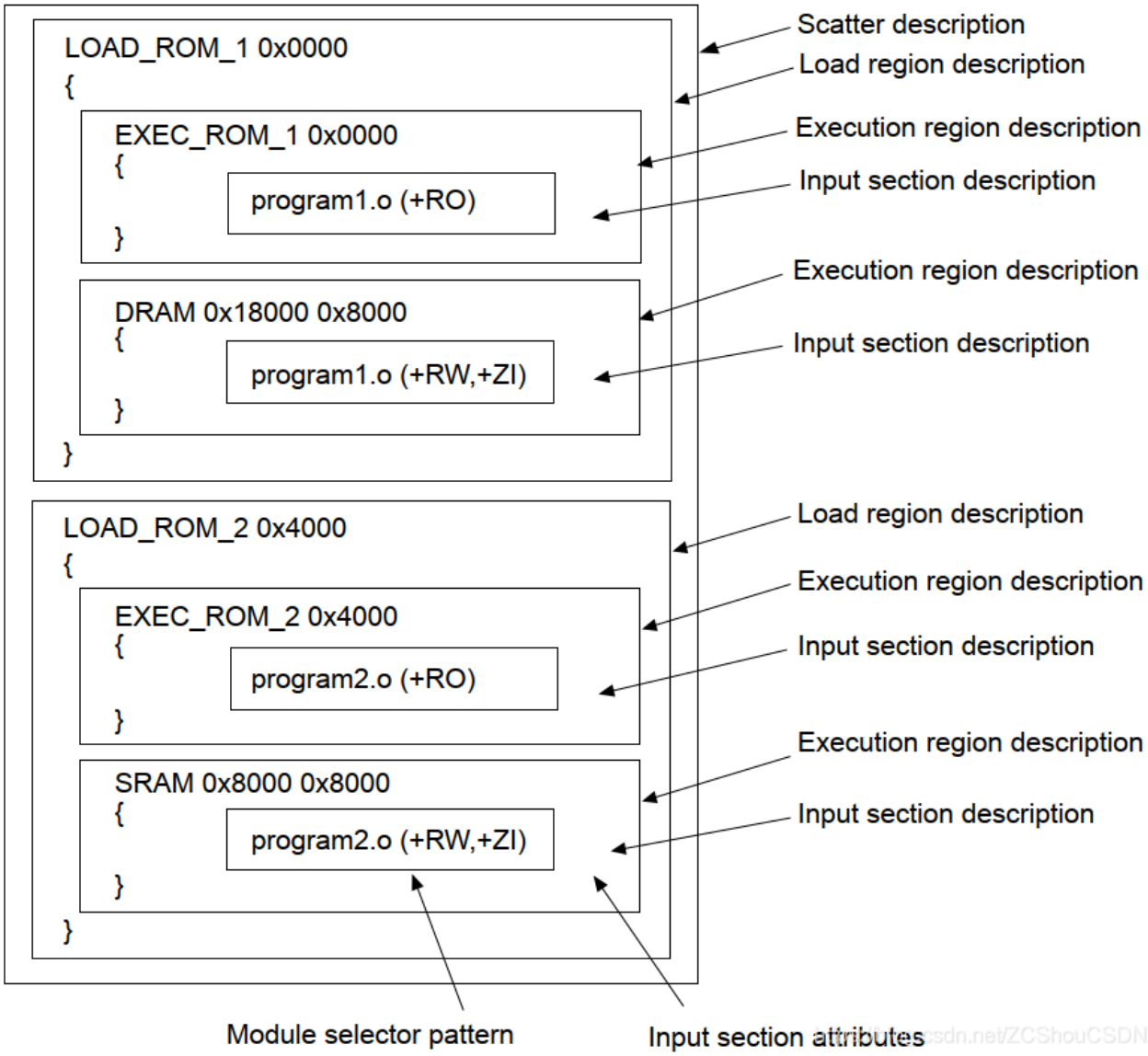
```
LOAD_ROM_1 0x0000               ; Start address for first load region (0x0000)
{
    EXEC_ROM_1 0x0000           ; Start address for first exec region (0x0000)
    {
        program1.o (+RO)        ; Place all code and RO data from
                                ; program1.o into this exec region
    }
    DRAM 0x18000 0x8000         ; Start address for this exec region (0x18000),
                                ; Maximum size of this exec region (0x8000)
    {
        program1.o (+RW, +ZI)   ; Place all RW and ZI data from
                                ; program1.o into this exec region
    }
}
LOAD_ROM_2 0x4000               ; Start address for second load region (0x4000)
{
    EXEC_ROM_2 0x4000
    {
        program2.o (+RO)        ; Place all code and RO data from
                                ; program2.o into this exec region
    }
    SRAM 0x8000 0x8000
    {
        program2.o (+RW, +ZI)   ; Place all RW and ZI data from
                                ; program2.o into this exec region
    }
}
```

在嵌入式系统，分散加载机制是非常必要的，因为这些系统通常会有 ROM、RAM 和内存映射外设。比如在下面几种情况就充分体现了分散加载文件的优势：

1. **复杂内存映射：** 如果必须将代码和数据放在多个不同的内存域中，则需要使用详细指令指定将哪些数据放在哪个内存空间中。
2. **不同类型的内存：** 许多系统都包含多种不同的物理内存设备，如闪存、 ROM、SDRAM 和快速 SRAM。分散加载描述可以将代码和数据与最适合的内存类型相匹配。例如，可以将中断代码放在快速 SRAM 中以缩短中断等待时间，而将不经常使用的配置信息放在较慢的闪存中。
3. **内存映射外设：** 分散加载描述可以将数据段放置在内存映射中的精确地址，这样内存映射的外设就可以被访问。
4. **位于固定位置的函数：** 可以将函数放在内存中的固定位置，即使已修改并重新编译周围的应用程序。
5. **使用符号标识堆和栈：** 链接应用程序时，可以为堆和栈位置定义一些符号。

分散加载文件

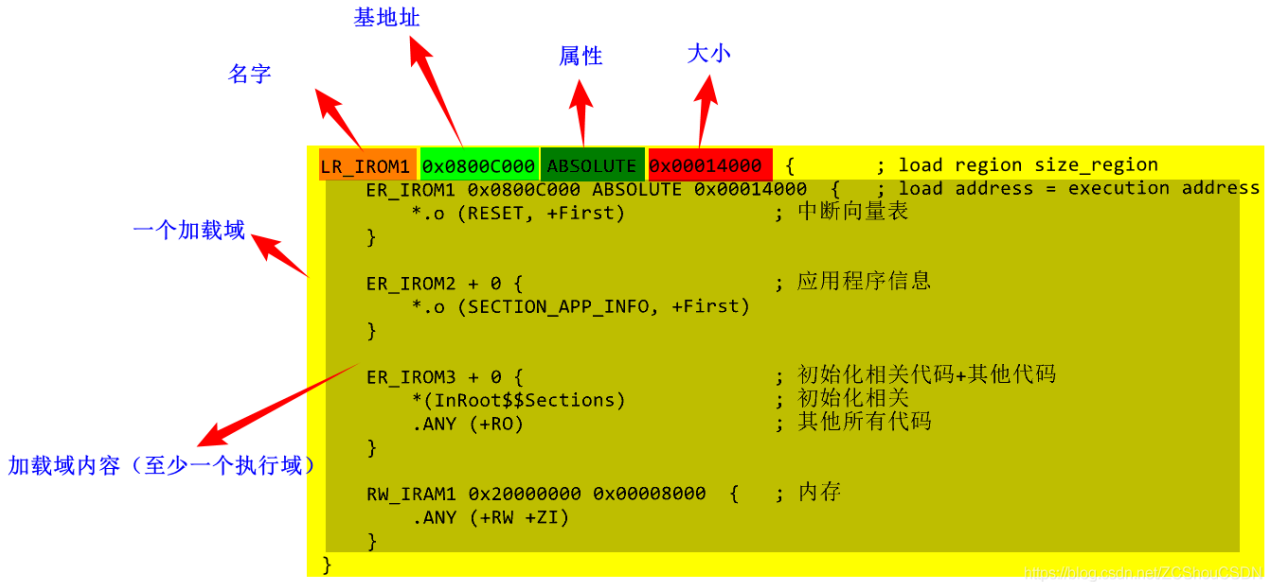
分散加载文件是对分散加载机制的内容描述，也被称为分散描述文件。***分散加载文件包含一个或多个加载域。 每个加载域可以包含一个或多个执行域。*** 一个典型的分散加载文件如下图所示：



分散加载文件的语法使用标准的 Backus-Naur Form (BNF) 表示法。关于 BNF 可以去 ARM 官网了解。

加载域

加载域指出了其中的执行域要放置的存储区域。一个加载域通常由如下部分组成：



- 一个名称(由链接器用来识别不同的加载域)。
- 一个基地址（加载视图中代码和数据的起始地址）。
- 一个或者多个属性，它们指定加载域的属性。
- 可选的大小限制
- 一个或多个执行域。

以 Backus-Naur Form（BNF）表示的加载域的语法为：

```
load_region_description ::=
  load_region_name  (base_address  | ("+" offset)) [attribute_list] [max_size]
  "{"
        execution_region_description+
  "}"
```

- **load_region_name：** 加载域的名字。可以使用引号括起来。如果程序中使用了任何域相关的连接器定义符号，则这个名字区分大小写。

- **base_address：** 指定当前加载域中的对象在被连接时的地址。 base_address 必须满足加载域的对齐约束。与 **+offset** 二选一使用。

- **+offset：** offset 为具体的阿拉伯数字。表示当前加载域比前一个加载域的末尾偏移的字节数。例如，`+0`。偏移量的值必须能被 4 整除。如果用在第一个加载域中，则 +offset 表示从零开始偏移字节。如果使用 +offset，则加载域可能会继承前一个加载域的某些属性。与 **base_address** 二选一使用。

- **attribute_list：** 指出当前加载域的属性列表。加载域可以继承先前加载域的属性。主要有以下这些（多个使用时以空格间隔开）：
  - `ABSOLUTE`： 指定将内容放置在链接后不会更改的 **base_address** 所表示的固定地址上。***这是默认值***，除非使用了 `PI` 或 `RELOC`。
  - `ALIGN alignment`： 指定对齐约束。alignment 为阿拉伯数字，最小为 4，必须是 2 的整数次幂。例如，`ALIGN 4`。 **base_address** 的值必须符合该对齐约束。如果使用了 **+offset**，连接器将计算并使用对齐后的地址。
  - `NOCOMPRESS`： RW 数据压缩默认情况下处于启用状态。 使用 NOCOMPRESS 关键字可以指定在最终镜像中不得压缩加载域的内容。
  - `OVERLAY`： 使用 `OVERLAY` 关键字可以在同一地址具有多个加载域。 ARM 工具不提供覆盖机制。 要在同一地址使用多个加载域，必须提供自己的叠加层管理器。***该属性不能被继承。***
  - `PI`： 表示当前域与位置无关。 内容不依赖于任何固定地址，并且在链接后无需任何额外处理即可移动。

    > 如果镜像中包含仅执行节（XO），则不支持此属性。

  - `PROTECTED`： 该关键字将阻止以下情况：
    - Overlapping of load regions.
    - Veneer sharing.
    - String sharing with the --merge option.
  - `RELOC`： 指出当前域是可重定位的。 内容取决于固定地址。 输出重定位信息，以使内容可以通过另一个工具移动到另一个位置。

    > 如果镜像中包含仅执行节（XO），则不支持此属性。

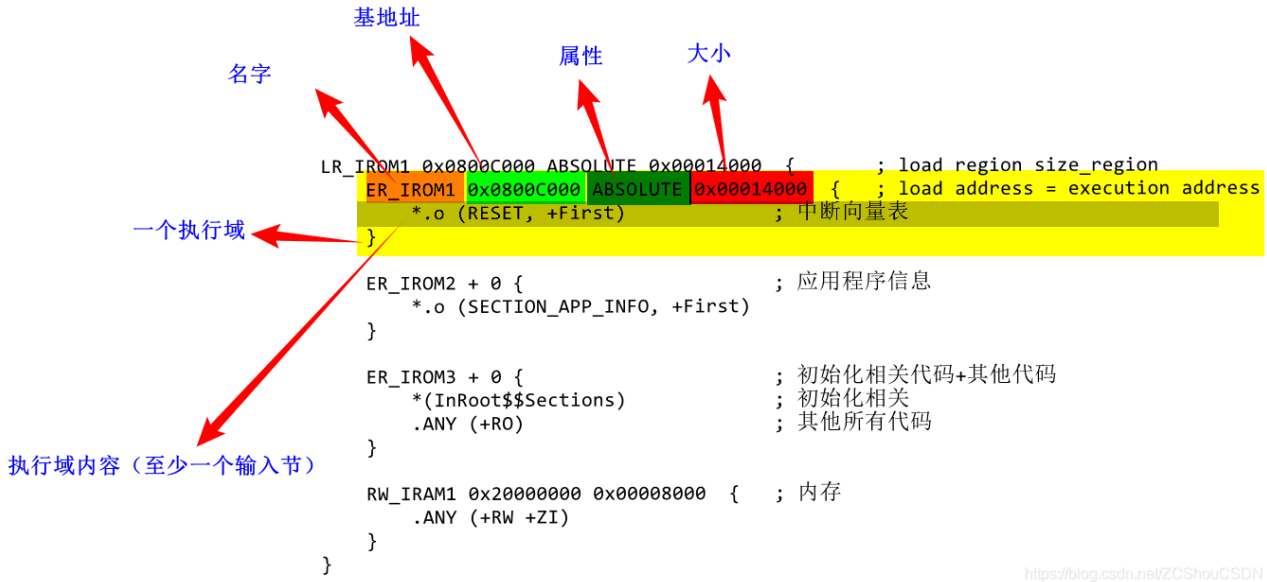
  关于属性的继承，参考下面的示例：

```
 LR1 0x8000 PI    ; 默认为 ABSOLUTE
{
    …
}
LR2 +0             ; LR2 从 LR1 继承 PI 属性
{
    …
}
LR3 0x1000         ; LR3 不继承 LR2 的任何属性，因为它没有相对基地址，恢复默认为
ABSOLUTE
{
    …
}
LR4 +0             ; LR4 继承 LR3 的  ABSOLUTE 属性
{
    …
}
LR5 +0 RELOC       ; LR5 不继承 LR4 的任何属性，因为它显式设置了 RELOC
{
    …
}
LR6 +0 OVERLAY     ; LR6 不继承 LR5 的任何属性，因为它显式设置了 OVERLAY
{
    …
}
LR7 +0             ; LR7 无法继承 OVERLAY，恢复默认为 ABSOLUTE
{
    …
}
```

- **max_size：** 指定加载域的最大大小。这是任何解压缩或零初始化发生之前的加载域的大小。如果指定了 max_size 值，则当该加载域中的内容超过 max_size 时，`armlink` 会生成错误。不指定 max_size 值，默认为 0xFFFFFFFF。
- **execution_region_description：** 加载域的内容。一个或者多个执行域。通常由一对 { } 包裹起来。

执行域

执行域指定镜像中的某些部分在运行时应该放置的存储区域。一个执行域通常由以下部分组成：



- 一个名称(由链接器用来识别不同的执行域)。
- 一个基址(绝对的或相对的)。
- 一个或者多个属性
- 可选的大小限制
- 一个或多个输入节（放置在此执行域中的模块）。

以 Backus-Naur Form（BNF）表示的执行域描述的语法为：

```
execution_region_description ::=
  exec_region_name (base_address | "+" offset) [attribute_list] [max_size | length]
  "{"
       input_section_description*
  "}"
```

- **exec_region_name：** 执行域的名字。可以使用引号括起来。如果程序中使用了任何域相关的连接器定义符号，则这个名字区分大小写。
- **base_address：** 指定当前执行域中的对象在连接时的地址。 base_address 必须字对齐。与 **+offset** 二选一使用。

  > 在执行域上使用ALIGN会使加载地址和执行地址都对齐。

- **+offset：** offset 为具体的阿拉伯数字。表示当前执行域比前一个执行域的末尾偏移的字节数。例如，`+0`。偏移量的值必须能被 4 整除。如果用在加载域中的第一个执行域中，则 +offset 表示从其所在的加载域基地址的偏移字节。如果使用 +offset，则执行域可能会继承前一个执行域或者其所在的加载域的某些属性。与 **base_address** 二选一使用。

- **attribute_list：** 指出当前执行域的属性列表。执行域可以继承先前执行域的属性（多个使用时以空格间隔开）。主要有以下这些：
  - `ABSOLUTE`：指定将内容放置在链接后不会更改的 **base_address** 所表示的固定地址上。
  - `ALIGN alignment`：指定对齐约束。alignment 为阿拉伯数字，最小为 4，必须是 2 的整数次幂。例如，`ALIGN 4`。 **base_address** 的值必须符合该对齐约束。如果使用了 **+offset**，连接器将计算并使用对齐后的地址。

    > 执行域上的 ALIGN 属性将导致加载地址和执行地址都对齐。

  - `ALIGNALL value`：增加执行域中各节的对齐方式。value 的值必须是 2 的正幂，并且必须大于或等于 4。
  - `ANY_SIZE max_size`：指定 `armlink` 可以用未分配的节填充的执行域内的最大大小。max_size 必须小于或等于域的大小。
  - `EMPTY [-]length`：在执行域中保留给定大小的空内存块，通常由堆或堆栈使用。 带有 `EMPTY` 属性的域中不能放置任何节。
  - `FILL value`：创建包含 value 值的链接器生成的区域（例如，`FILL 0xFFFFFFFF`）。FILL 属性可以替换以下组合：`EMPTY ZEROPAD PADVALUE`。
  - `FIXED`：固定地址。 链接器会尝试使执行地址等于加载地址。 这使得该域成为根区域。 如果不可能，则链接器会产生错误。
  - `NOCOMPRESS`：RW 数据压缩默认情况下处于启用状态。 使用 `NOCOMPRESS` 关键字，可以指定执行域中的 RW 数据不得在最终镜像中压缩。
  - `OVERLAY`：用于地址范围重叠的节。 如果连续的执行域具有相同的 **+offset**，那么它们将被赋予相同的基地址。
  - `PADVALUE value`：定义用于填充的值。例如，`EXEC 0x10000 PADVALUE 0xFFFFFFFF EMPTY ZEROPAD 0x2000` 表示创建一个大小为 0x2000 且使用 0xFFFFFFFF 填充的域。
  - `PI`：该域仅包含与位置无关的节。 内容不依赖于任何固定地址，并且在链接后无需任何额外处理即可移动。

    > 如果镜像中包含仅执行节（XO），则不支持此属性。

  - `SORTTYPE algorithm`：指定执行域的排序算法，例如 `ER1 +0 SORTTYPE CallTree`。该属性的优先级高于通过连接器参数 `--sort 算法` 的方式。
  - `UNINIT`：用于创建包含未初始化数据或内存映射 I/O 的执行域。

- ZEROPAD：零初始化的段作为零填充块写入 ELF 文件。只有根执行域可以使用 ZEROPAD 属性进行零初始化。在非根执行域中使用 ZEROPAD 属性会生成警告，并忽略该属性。

关于属性的继承，参考下面的示例：

```
LR1 0x8000 PI
{
    ER1 +0          ; ER1 从 LR1 继承 PI
    {
        …
    }
    ER2 +0          ; ER2 从 ER1 继承 PI
    {
        …
    }
    ER3 0x10000     ; ER3不继承，因为它没有相对基地址，将恢复使用默认值 ABSOLUTE
    {
        …
    }
    ER4 +0          ; ER4 从 ER3 继承 ABSOLUTE
    {
        …
    }
    ER5 +0 PI       ; ER5 不继承，它显式设置 PI
    {
        …
    }
    ER6 +0 OVERLAY ; ER6 不继承，它显式设置 OVERLAY
    {
        …
    }
    ER7 +0         ; ER7 无法继承 OVERLAY，恢复默认值 ABSOLUTE
    {
        …
    }
}
```

max_size：

对于具有

EMPTY

或

FILL

属性的执行域，将

max_size

表示域的长度。 否则，max_size 表示当前执行域的最大大小。

- **[–]length：** 只能与 EMPTY 一起使用，以表示在内存中是向下生长的。 如果长度为负值，则将 base_address 视为该域的结束地址。

- **input_section_description：** 执行域的内容，一个或者多个输入节。通常由一对 { } 包裹起来。

根执行域

根执行域是具有相同的加载地址和执行地址的执行域。镜像的初始入口点必须在根执行域中。 如果初始入口点不在根执行域中，则链接失败，链接器给出一个错误提示。我们可以有以下两种方式指定根执行域：

强制指定执行域的基地址 = 加载域的基地址。例如：

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        * (+RO)        ; all RO sections (must include section with
                       ; initial entry point)
    }
    …                  ; rest of scatter-loading description
}
```

为加载域中的第一个执行域指定 +0 偏移量。例如：

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO +0               ; +offset
    {
        * (+RO)        ; all RO sections (must include section with
                       ; initial entry point)
    }
    …                  ; rest of scatter-loading description
}
```

如果为加载域中的所有后续执行域指定了 0(+0) 的偏移量，那么所有不在包含 ZI 的执行域后面的执行域也都是根执行域。

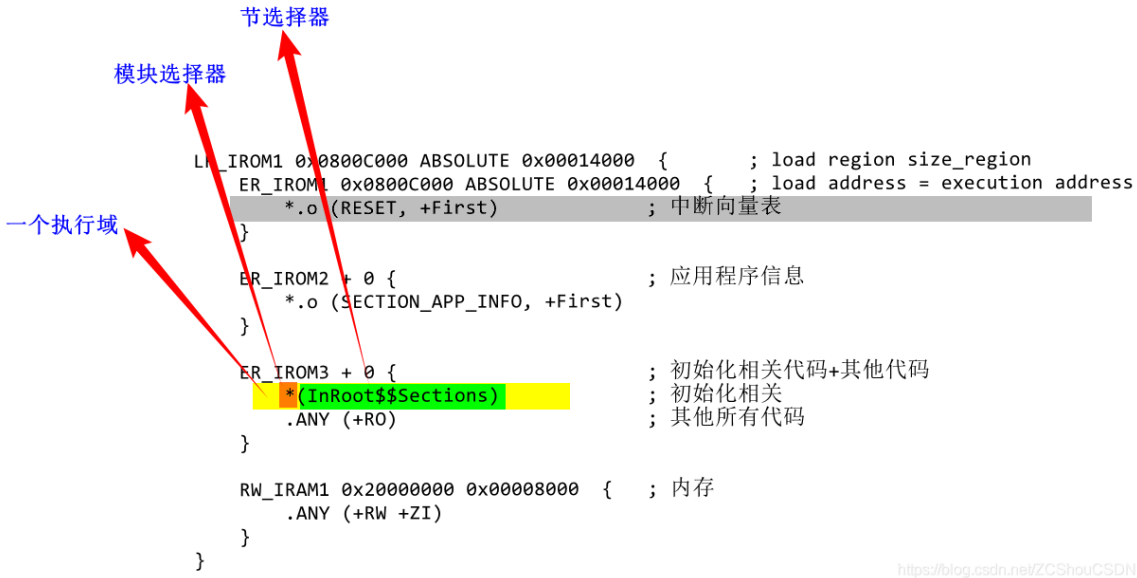
对执行域使用 FIXED 属性，以创建在固定地址加载和执行的根执行域。例如：

```
LR_1 0x040000              ; load region starts at 0x40000
{                          ; start of execution region descriptions
    ER_RO 0x040000         ; load address = execution address
    {
        * (+RO)            ; RO sections other than those in init.o
    }
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)        ; all RO sections from init.o
    }
    …                      ; rest of scatter-loading description
}
```

可以用它来把一个函数或一个数据块，比如一个常量表或一个校验和，放在 ROM 中的一个固定地址上，这样就可以很容易地通过指针访问它。

输入节

输入节描述指定将哪些输入节放到执行域中。输入节描述通过以下方式标识输入节：



- 模块名称（对象文件名，库成员名称或库文件名）。 模块名称可以使用通配符。
- 输入节名称或输入节属性，例如 READ 或 CODE。 可以使用通配符作为输入节的名称。
- 符号名

以 Backus-Naur Form（BNF）表示的输入节描述的语法为：

```
input_section_description ::=
  module_select_pattern
        [ "(" input_section_selector ( "," input_section_selector )* ")" ]

input_section_selector ::=
        ("+" input_section_attr | input_section_pattern | input_symbol_pattern |
section_properties
```

- **module_select_pattern：** 模块匹配器。用来指出输入节所在的模块。可以使用通配符。通配符 * 匹配零个或多个字符；通配符 ? 匹配任何单个字符。匹配不区分大小写。分散文件中不能有两个* 选择器。
  - * 匹配任何模块或库
  - *.o 匹配任何对象模块
  - math.o 与 math.o 模块匹配
  - *armlib* 匹配 ARM 提供的所有C库
  - "file 1.o" 与 file 1.o 相匹配。注意有空格时需要双引号。
  - *math.lib 匹配以math.lib结尾的任何库路径。例如，C:\apps\lib\math\satmath.lib

- **input_section_selector：** 输入节选择器。主要有以下几种：
  - **input_section_attr：** 与输入节相匹配的属性选择器。每个 **input_section_attr** 后前面都会有个 `+` 。由一对小括号包裹，多个属性以逗号间隔开。选择器不区分大小写。主要有以下这些：
    - RO-CODE
    - RO-DATA
    - RO 用来同时选择RO-CODE和RO-DATA
    - RW-DATA
    - RW-CODE
    - RW 用来同时选择 RW-CODE 和 RW-DATA
    - XO
    - ZI
    - ENTRY, that is, a section containing an ENTRY point

    可以识别以下同义词：
    - CODE for RO-CODE
    - CONST for RO-DATA
    - TEXT for RO
    - DATA for RW
    - BSS for ZI

    可以识别以下伪属性：
    - FIRST
    - LAST
  - **input_section_pattern：** 输入节选择器，用来指出选择的输入节。不区分大小写。可以使用通配符。通配符 `*` 匹配 0个或多个字符；通配符 `?` 匹配任何单个字符。可以使用引号括起来。
  - **input_symbol_pattern：** 可以通过输入节定义的全局符号的名称来选择输入节。这使我们可以从部分链接的对象中选择具有相同名称的各个节。
  - **section_properties：** 可以是 `+FIRST`，`+LAST` 和 `OVERALIGN value`。其中，OVERALIGN 中的 value 必须为 2 的正幂，并且必须大于或等于 4

堆栈

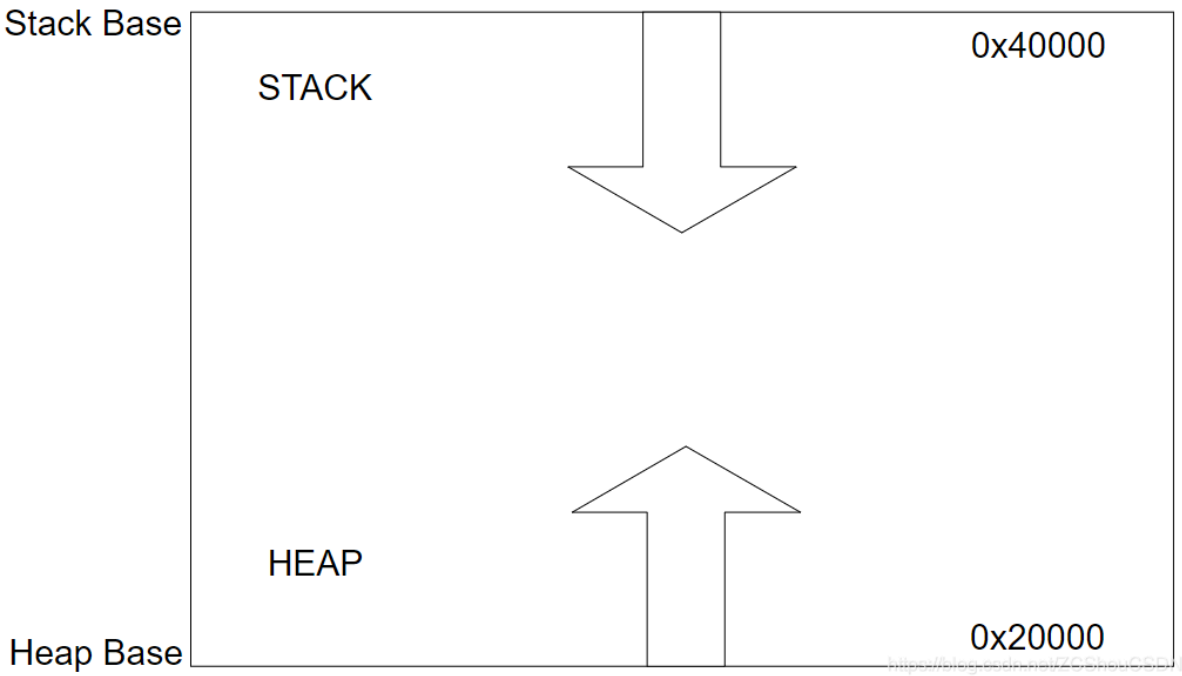
前面我们说过，如果使用了分散加载文件，那么有些连接器定义的符号就变成了未定义。这些符号就包含堆栈的定义函数所使用的符号，因此，使用的是分散加载文件时，就必须要手动设置栈和堆。方法主要有以下两种：

- 在分散文件中使用下列方法之一
  - 定义名为 `ARM_LIB_STACK` 和 `ARM_LIB_HEAP` 的单独的栈和单独的堆域。
  - 定义包含堆栈和堆的组合域，名为 `ARM_LIB_STACKHEAP`。
- 通过重新实现 `__user_setup_stackheap()` 来设置堆和堆栈边界。（在我们的项目中的 .s 启动文件中，是这种方法）

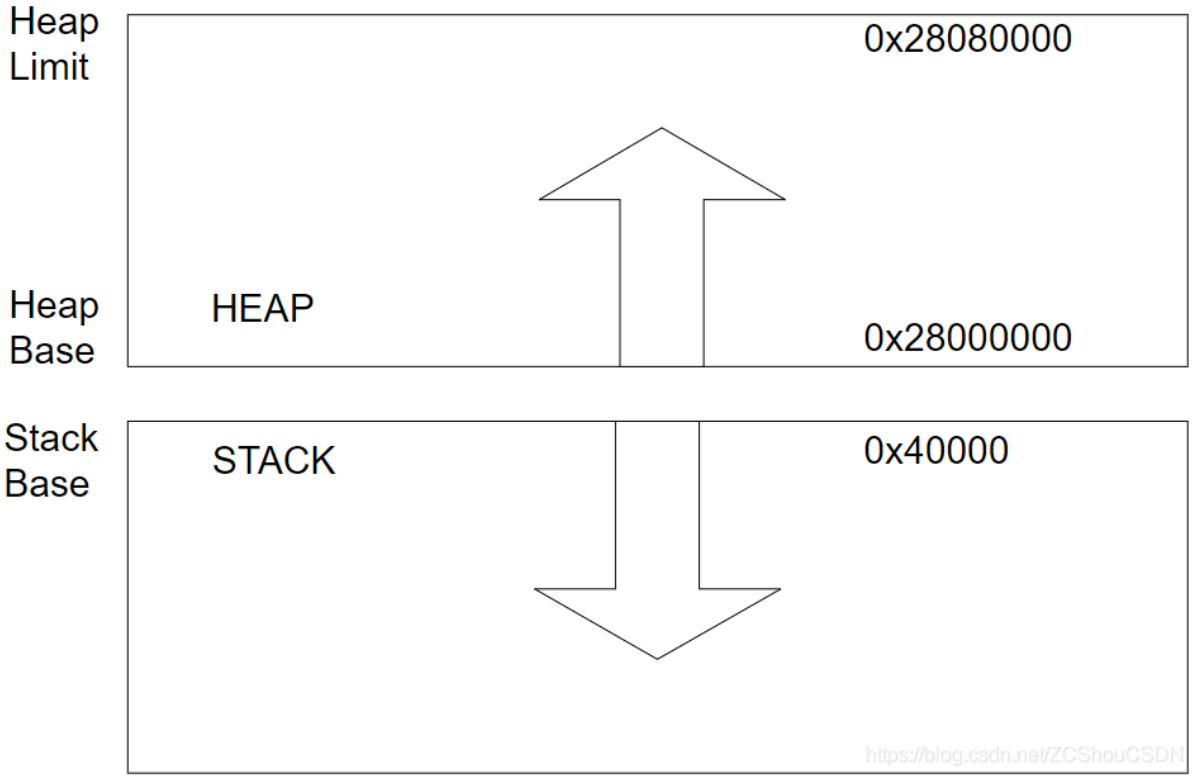
内存模型

在手动设置栈和堆之前，我们必须先了解一下 ARM 的内存模型。ARM 编译套件提供了两种内存模型：

- **One-Region Model：** 应用程序的栈和堆在相同的内存域中彼此接近，在此运行时内存模型中，当分配了新的堆空间时（例如，在调用`malloc()`时），将根据栈指针的值检查堆。



- **Two-Region Model：** 应用程序的栈和堆放置在单独的内存域中。要使用 Two-Region Model，必须导入函数 `__use_two_region_memory`。在这个运行时内存模型中，当分配新的堆空间时，将根据堆限制检查堆。



注意，在这两种运行时内存模型中，栈的增长都是未检查的。

在 Keil 中，一般都是用的 Two-Region Model。无论哪一种方式，都需要我们自己来定义。在 Keil 项目中，通常会在启动文件(.s) 中指定内存模型。

在分散加载文件中定义

ARM C 库提供了 `__user_setup_stackheap()` 函数的多种实现，该函数会使用在分散加载文件中定义的与堆和栈相关的符号来生成堆栈。

如果要选择 Two-Region Model，则需要在分散文件中定义两个特殊的执行域，分别为 `ARM_LIB_HEAP` 和 `ARM_LIB_STACK`。两个域都有 `EMPTY` 属性。这样，ARM C 库将不再使用默认的 `__user_setup_stackheap()` ，而是使用如下符号定义栈：

- `Image$$ARM_LIB_STACK$$Base`
- `Image$$ARM_LIB_STACK$$ZI$$Limit`
- `Image$$ARM_LIB_HEAP$$Base`
- `Image$$ARM_LIB_HEAP$$ZI$$Limit`

只能指定一个`ARM_LIB_STACK` 域和 一个`ARM_LIB_HEAP` 域，并且必须分配一个大小，例如，在自己的分散加载文件中如下来定义栈和堆：

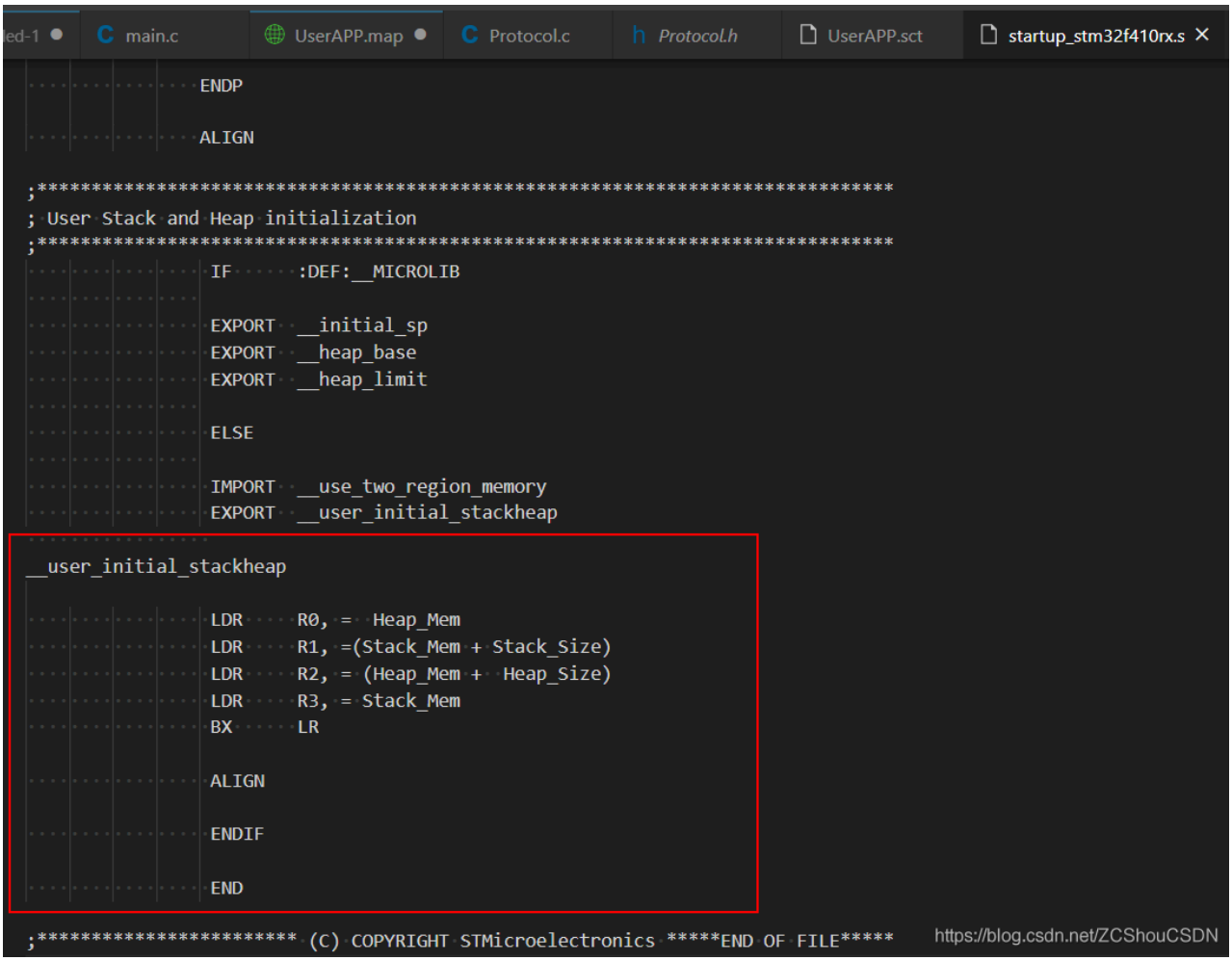
```
LOAD_FLASH …
{
    …
    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
    …
}
```

如果要选择 One-Region Model，则必须在分散加载文件中定义一个名为 `ARM_LIB_STACKHEAP` 的执行域，当前域也同样是 `EMPTY` 属性。这样，`__user_setup_stackheap()` 将使用以下符号：`Image$$ARM_LIB_STACKHEAP$$Base`、`Image$$ARM_LIB_STACKHEAP$$ZI$$Limit`。例如，在自己的分散加载文件中如下来定义栈和堆：

```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACKHEAP 0x20000 EMPTY 0x20000  ; Heap and stack growing towards
    { }                                      ; each other in the same region
    ...
}
```

实现函数 `__user_setup_stackheap()`

如果我们在自己的代码中重新实现 `__user_setup_stackheap()`，则此方法将覆盖所有库中相同的实现。在我们的项目中，一般会在启动文件（例如 STM32 中的 `startup_stm32f410rx.s`）实现 `__user_setup_stackheap()` 函数，如下图：



注意，如果要选择 Two-Region Model，则必须要引入函数 `__use_two_region_memory`。

参考

1. http://www.vlsiip.com/c/embedded_c/sctf.html
2. ARM 连接器手册
3. https://www.keil.com/support/man/docs/armlink/armlink_pge1362075656353.htm