(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 103886095 B
(45)授权公告日 2017.10.24

(21)申请号 201410133527.1

(22)申请日 2014.04.03

(65)同一申请的已公布的文献号
申请公布号 CN 103886095 A

(43)申请公布日 2014.06.25

(73)专利权人 北京深思数盾科技股份有限公司
地址 100193 北京市海淀区西北旺东路10号院东区5号楼5层510

(72)发明人 孙吉平 韩勇

(74)专利代理机构 北京金信知识产权代理有限公司 11225
代理人 黄威 喻嵘

(51)Int.Cl.
G06F 17/30(2006.01)
G06F 9/44(2006.01)

(56)对比文件
CN 101968736 A,2011.02.09,说明书第[0002]-[0287]段.

审查员 李欢

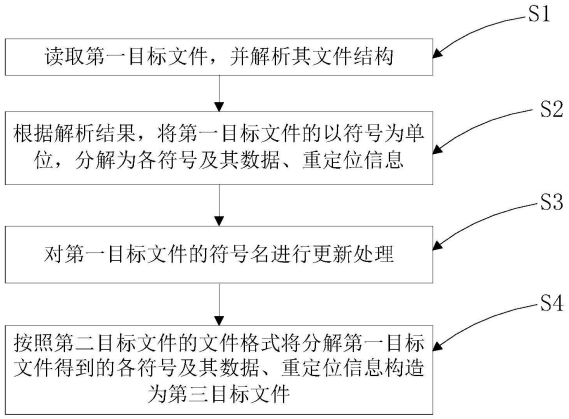
权利要求书2页 说明书18页 附图1页

(54)发明名称

跨平台目标文件复用方法

(57)摘要

本发明公开了一种跨平台目标文件复用方法,包括以下步骤:S1:读取第一目标文件,并解析其文件结构;S2:根据解析结果,将所述第一目标文件以符号为单位,分解为各符号及其数据、重定位信息;S3:对第一目标文件的符号名进行更新处理;S4:按照第二目标文件的文件格式将分解第一目标文件得到的各符号及其数据、重定位信息构造为第三目标文件。本发明的跨平台目标文件复用方法,无需改动编译器,仅对目标文件进行处理,使不同编译器生成的目标文件可以复用。



CN 103886095 B

1.一种跨平台目标文件复用方法,其特征在于,包括以下步骤:

S1:读取第一目标文件,并解析其文件结构;

S2:根据解析结果,将所述第一目标文件以符号为单位,分解为各符号及其数据、重定位信息;

S3:对第一目标文件的符号名进行更新处理;

S4:按照第二目标文件的文件格式将分解第一目标文件得到的各符号及其数据、重定位信息构造为第三目标文件;

其中,所述第一目标文件和第二目标文件分别为通过第一编译器和第二编译器编译生成的格式不同的目标文件。

2.根据权利要求1所述的跨平台目标文件复用方法,其特征在于,第二编译器及其链接器配置为能够复用所述第三目标文件。

3.根据权利要求1所述的跨平台目标文件复用方法,其特征在于,所述符号包括函数、变量、常量。

4.根据权利要求3所述的跨平台目标文件复用方法,其特征在于,所述步骤S3具体包括:

S31:根据所述第一目标文件所属文件类型的符号名命名格式解析第一目标文件的符号名信息,符号名信息包括名字、类型,其中:

对于面向对象语言包括命名空间和类名中的至少一种;

对于函数包括调用约定和参数列表中的至少一种;

对于模板包括模板参数类型;

S32:根据所述符号名信息和所述第二目标文件的符号名命名规则构造新的符号名。

5.根据权利要求4所述的跨平台目标文件复用方法,其特征在于,在对微软的COFF文件符号名信息进行解析的情况下,步骤S31具体包括:

S311:读取第一目标文件符号名的第一个字段,如果该字段不为‘?’,判定第一目标文件符号名为根据C语言命名的符号名;

S312:根据第一目标文件符号名的第二个字段解析函数或者变量名;

S313:根据第一目标文件符号名的第三个字段解析类名/命名空间;

S314:解析第一目标文件符号名的第四个字段表示的是函数还是变量,如果是函数则根据后续字段解析函数调用约定和作用域,然后解析返回值及各参数的类型;如果是变量则根据后续字段解析变量的作用域,然后解析变量类型,再解析变量所在内存空间的访问属性。

6.根据权利要求5所述的跨平台目标文件复用方法,其特征在于,在构造微软的COFF文件符号名的情况下,步骤S32具体包括:

S321:添加C++符号标识‘?’;

S322:添加函数或者变量名;

S323:依次添加类名/命名空间的标识;

S324:如果是函数则添加调用约定标识和作用域标识,然后依次添加返回值和各参数的类型字符串及结束标识,再添加默认的异常规范标识;如果是变量则添加变量作用域字符串,然后添加变量类型字符串,再添加变量所在内存空间的访问属性标识。

7.根据权利要求4所述的跨平台目标文件复用方法，其特征在于，所述步骤S4具体包括：

S41：构造所述第三目标文件的段；

S42：添加数据，构造所述第三目标文件的符号表；

S43：添加所述第三目标文件的重定位；

S44：继续添加与所述第三目标文件格式及运行时的库有关的必要要素信息。

跨平台目标文件复用方法

技术领域

[0001] 本发明涉及计算机领域，尤其涉及一种跨平台目标文件复用方法。

背景技术

[0002] 同一编程语言不同编译器之间甚至不同语言之间的目标文件进行共享/复用，如C语言目标文件有elf格式、coff格式和omf格式。软件保护中，使用自定编译器(一般使用开源编译器以节省时间和成本)对部分源文件进行编译和处理生成加密的代码是一种有效的方法。软件开发需要一整套的开发工具，包括文本编辑器、编译器、汇编器、链接器等，对于软件保护工具开发者而言，开发这样一套工具是不经济也不必要的，用户使用其惯用的开发工具而配以软件保护工具是一种实用的方法。例如Microsoft Visual Studio是一套常用的商用软件开发工具，可以以开源的GCC(GNU Compiler Collection)或clang编译器为基础开发软件保护工具配合VC编译器使用，保护C/C++语言开发的软件。但是，不同编译器的目标文件格式不同，不能兼容。更改结构复杂，源代码冗长的编译器是一种代价较大的方法。

[0003] COFF(Common Object File Format，通用对象文件格式)是一种很流行的对象文件格式。比如，Visual Studio编译器所产生的目标文件(*.obj)就是这种格式。其它的编译器，如GCC、ICL(Intel C/C++Compiler)、VectorC，也使用这种格式的目标文件。不仅仅是C/C++，很多其它语言也使用这种格式的对象文件。

[0004] COFF文件的整体结构如下：

File Header

Optional Header

[0005] Section Header 1

.....

Section Header n

Section Data

Relocation Directives

[0006] Line Numbers

Symbol Table

String Table。

[0007] COFF文件一共有8种数据，自上而下分别为：

[0008] 1.文件头(File Header)

[0009] 2.可选头(Optional Header)

[0010] 3.段落头(Section Header)

[0011] 4.段落数据(Section Data)

[0012] 5.重定位表(Relocation Directives)

[0013] 6.行号表(Line Numbers)

[0014] 7.符号表(Symbol Table)

[0015] 8.字符串表(String Table)。

[0016] 其中,除了段落头可以有多个节(因为可以有多个段落)以外,其它的所有类型的节只能有一个。

[0017] 文件头:COFF文件的头,它用来保存COFF文件的基本信息,如文件标识,各个表的位置等等。

[0018] 可选头:在目标文件中,基本上都没有这个头;但在其它的文件中(如:可执行文件)这个段用来保存在文件头中没有描述到的信息。

[0019] 段落头:用来描述段落信息,每个段落都有一个段落头来描述。段落的数目在文件头中会指出。

[0020] 段落数据:通常是COFF文件中最大的数据段,每个段落真正的数据就保存在这个位置。

[0021] 重定位表:通常只存在于目标文件中,用来描述COFF文件中符号的重定位信息。

[0022] 符号表:用来保存COFF文件中所用到的所有符号的信息,连接多个COFF文件时,这个表帮助我们重定位符号。调试程序时也要用到它。

[0023] 字符串表:用来保存字符串的。符号表是以记录的形式来描述符号信息的,但它只为符号名称留置了8个字符的空间,在现在的程序中,一个符号名动不动就数十个字符,8个字符空间的不够,因此需将这些名称存在字符串表中。而符号表中只记录这些字符串的位置。

[0024] ELF格式简介如下:

[0025] ELF(Executable and Linkable Format,可执行连接格式)是UNIX系统实验室(USL)作为应用程序二进制接口(Application Binary Interface,ABI)而开发和发布的。

[0026] ELF头位于文件的开始,描述了该文件的组织情况。sections保存着object文件的信息,较为常见的section包括指令、数据、符号表、字符串表、重定位信息等等。section头表(section header table)包含了描述文件sections的信息。每个section在这个表中有一个入口;每个入口给出了该section的名字,大小,等等信息。各部分分布如下表所示:

[0027]

Elf header
Program header table
section1
.....
section n
section header table

[0028] 现有的编译器其目标文件格式大多数是公开的,如Microsoft Visual C++的目标文件格式为COFF格式;CLANG、GCC的目标文件格式为ELF格式(也可输出其它格式,如COFF,但有很多不同,不能兼容);C++Builder、Delphi的目标文件格式为OMF格式。虽然上述几种目标文件格式不同,但其要素相似,其基本构成为:节区/段(section/segment)、符号

(symbol)、重定位(relocation/fixup)。

[0029] 不同目标文件格式有着不同的符号名命名方式。一般C语言函数、变量名未加修饰或加“_”前缀(例如:用户在源代码中编辑的函数“Func”,ELF格式中该函数的符号名仍为“Func”,而COFF格式中该函数的符号名为“_Func”),而C++语言函数、变量符号名中都包含了类名、命名空间、类型等信息。

发明内容

[0030] 本发明提供一种跨平台目标文件复用方法,无需改动编译器,仅对目标文件进行处理,使不同编译器生成的目标文件可以复用。

[0031] 为了解决上述技术问题,本发明提供了一种跨平台目标文件复用方法,包括以下步骤:

[0032] S1:读取第一目标文件,并解析其文件结构;

[0033] S2:根据解析结果,将第一目标文件的以符号为单位,分解为各符号及其数据、重定位信息;

[0034] S3:对第一目标文件的符号名进行更新处理;

[0035] S4:按照第二目标文件的文件格式将分解第一目标文件得到的各符号及其数据、重定位信息构造为第三目标文件。

[0036] 作为优选,所述第一目标文件和第二目标文件分别为通过第一编译器和第二编译器编译生成的格式不同的目标文件。

[0037] 作为优选,第二编译器及其链接器配置为能够复用所述第三目标文件。

[0038] 作为优选,所述符号包括函数、变量、常量。

[0039] 作为优选,所述步骤S3具体包括:

[0040] S31:根据所述第一目标文件所属文件类型的符号名命名格式解析第一目标文件的符号名信息,符号名信息包括名字、类型,其中:

[0041] 对于面向对象语言可选地包括命名空间、类名;

[0042] 对于函数可选地包括调用约定、参数列表;

[0043] 对于模板可选地包括模板参数类型;

[0044] S32:根据所述符号名信息和所述第二目标文件的符号名命名规则构造新的符号名。

[0045] 作为优选,在对微软的COFF文件符号名信息进行解析的情况下,步骤S31具体包括:

[0046] S311:读取第一目标文件符号名的第一个字段,如果该字段不为‘?’,判定第一目标文件符号名为根据C语言命名的符号名;

[0047] S312:根据第一目标文件符号名的第二个字段解析函数或者变量名;

[0048] S313:根据第一目标文件符号名的第三个字段解析类名/命名空间;

[0049] S314:解析第一目标文件符号名的第四个字段表示的是函数还是变量,如果是函数则根据后续字段解析函数调用约定和作用域,然后解析返回值及各参数的类型;如果是变量则根据后续字段解析变量的作用域,然后解析变量类型,再解析变量所在内存空间的访问属性。

[0050] 作为优选，在构造微软的COFF文件符号名的情况下，步骤S32具体包括：

[0051] S321：添加C++符号标识‘?’；

[0052] S322：添加函数或者变量名；

[0053] S323：依次添加类名/命名空间的标识；

[0054] S324：如果是函数则添加调用约定标识和作用域标识，然后依次添加返回值和各参数的类型字符串及结束标识，再添加默认的异常规范标识；如果是变量则添加变量作用域字符串，然后添加变量类型字符串，再添加变量所在内存空间的访问属性标识。

[0055] 作为优选，所述步骤S4具体包括：

[0056] S41：构造所述第三目标文件的段；

[0057] S42：添加数据，构造所述第三目标文件的符号表；

[0058] S43：添加所述第三目标文件的重定位；

[0059] S44：继续添加与所述第三目标文件格式及运行时的库有关的必要要素信息。

[0060] 与现有技术相比，本发明的跨平台目标文件复用方法的有益效果在于：通过对异构目标文件格式进行符号和结构转换，构造匹配用户所用开发工具的新的目标文件，无需改动编译器，使不同编译器生成的目标文件可以复用。可以提高不同编译器目标文件格式不同的兼容性、简化操作、节省成本。

附图说明

[0061] 图1为本发明的实施例的跨平台目标文件复用方法的流程示意图。

具体实施方式

[0062] 下面结合附图和具体实施例对本发明的实施例的跨平台目标文件复用方法作进一步详细描述，但不作为对本发明的限定。

[0063] 图1为本发明的实施例的跨平台目标文件复用方法的流程示意图。本发明的实施例的跨平台目标文件复用方法，包括以下步骤：

[0064] S1：读取第一目标文件，并解析其文件结构(段、符号、重定位)；

[0065] S2：根据解析结果，将第一目标文件的以符号为单位，分解为各符号及其数据、重定位信息；

[0066] S3：对第一目标文件的符号名进行更新处理；

[0067] S4：按照第二目标文件的文件格式将分解第一目标文件得到的各符号及其数据、重定位信息构造为第三目标文件。

[0068] 其中，步骤S2中符号包括函数、变量、常量等信息。

[0069] 本发明的方法通过对异构目标文件格式进行符号和结构转换，构造匹配用户所用开发工具的新的目标文件，无需改动编译器，使不同编译器生成的目标文件可以复用。可以提高不同编译器目标文件格式不同的兼容性、简化操作、节省成本。

[0070] 作为本发明的一个改进，第一目标文件和第二目标文件为分别通过编译器编译生成的格式不同的目标文件。作为本实施例的优选方案，采用的具体方式为，第一目标文件为通过第一编译器编译生成的目标文件，而进一步的，第二目标文件为通过第二编译器编译生成的与第一目标文件格式不同的目标文件。第一编译器和第二编译器均为现有的编译

器，如GCC(GNU编译器集合)、Clang编译器等，其区别在于，两个编译器编译生成的目标文件格式不同。

[0071] 作为进一步的改进，第二编译器及其链接器配置为可以复用第三目标文件，以使得第三目标文件与第二目标文件具有相同的文件格式。本实施例中，第三目标文件与第二目标文件具有相同的文件格式，即第二编译器及其链接器可以复用第三目标文件。

[0072] 作为更进一步的改进，步骤S3具体包括：

[0073] S31：根据第一目标文件所属文件类型的符号名命名格式解析第一目标文件的符号名信息，符号名信息包括名字、类型。对面向对象语言(如C++)还可能包括命名空间、类名，对函数，可能包括调用约定、参数列表等，对于模板则可选地包括模板参数类型。

[0074] 具体的，例如对于C++函数“void Calculate::add(int,int)”，在Microsoft Visual C++编译后的目标文件中其符号名为“?add@Calculate@@YAHHH@Z”，其中?表示该符号是C++符号而非C符号；add为函数名；第一个@为命名空间/类名的起始标识；Calculate@表示函数包含在Calculate类/命名空间中；第三个@为命名空间/类名的结束标识；YA表示该符号为cdecl类型的全局函数；HHH表示函数的依次表示返回值和各参数的类型，H表示int类型；第四个@为参数结束标识；最后一个字母Z表示默认的异常规范；总之，该函数除了函数体外的所有信息都包含在符号名中，从而可以解析和重构。作为一种实施方式，在对微软的COFF文件符号名信息进行解析的情况下，以Microsoft Visual C++所用的COFF格式为例，步骤S31具体包括：a.读取符号名第一个字符，如果不为‘?’，表示其为C语言而来的符号；b.解析函数/变量名；c.解析类名/命名空间；d.解析该符号是函数还是变量，如果是函数则跳转到e，否则跳转到g；e.解析函数调用约定和作用域；f.解析返回值及各参数的类型；g.解析变量的作用域；h.解析变量类型；i.解析变量所在内存空间的访问属性。

[0075] S32：根据符号名信息和第二目标文件的符号名命名规则构造新的符号名。

[0076] 作为一种实施方式，在构造微软的COFF文件符号名的情况下，以Microsoft Visual C++所用的COFF格式为例，构造该格式符号名的过程包括：a.添加C++符号标识‘?’；b.添加函数/变量名；c.依次添加类名/命名空间开始标识‘@’、类名/命名空间及分隔符‘@’(如果不在任何类或命名空间中则省略此项)、类名/命名空间结束标识‘@’；d.如果是函数则跳转到e，否则跳转到h；e.添加调用约定标识和作用域标识；f.依次添加返回值和各参数的类型字符串及结束标识‘@’；g.添加默认的异常规范标识‘Z’；h.添加变量作用域字符串；i.添加变量类型字符串；j.添加变量所在内存空间的访问属性标识。

[0077] 作为更进一步的改进，步骤S4具体包括：

[0078] S41：构造第三目标文件的段；

[0079] S42：添加数据，构造第三目标文件的符号表；

[0080] S43：添加第三目标文件的重定位；

[0081] S44：继续添加要素信息，例如其他与目标文件格式及运行时库有关的必要要素信息。

[0082] 为使本发明的目的、技术方案及优点更加清楚明白，以下参照附图并举实施例，对本发明进一步详细说明。

[0083] 实施例1

[0084] 本实施例中，目标文件1为elf格式，目标文件2为coff格式，编程语言为C语言。在

具体实现上，本实施例采用了中间结构对符号、重定位进行描述和转换。

```
//重定位
struct VmReloc
{
int offset;//要重定位的位置
int value;//要定位的位置的值
int type;//重定位类型
std::string symname;//符号名
};
//符号绑定类型(静态或全局)
enum VM_Binding{
VMB_GLOBAL,//全局符号
VMB_STATIC,//静态符号
};
//类型(变量/函数/其它)
enum VM_Type
{
VMT_VAR,//变量
VMT_FUNC,//函数
VMT_OTHER,//其它
};
//在内存中的存储属性，只读、读写、可执行
enum VM_MemAttr
{
VMM_RO,//只读
VMM_RW,//读写
```

[0085]

```
VMM_EXE,//可执行
   VMM_EXTERNAL,//外部定义（没有数据）
};
//符号
class VmSymbol
{
public:
std::string  name;//名字
int indx; //段索引
int offset;//在段内的偏置
int size; //大小
VM_Binding binding;//全局/静态/外部
VM_Type type; //函数/变量
};
//可用来表示段、函数、变量等
class VmObject
:public VmSymbol
{
public:
char* body; //内容
int attr; //内存属性
std::vector<VmSymbol> symbols;//内部符号表
std::vector<VmReloc>  relocs;//重定位表
    };
```

[0086]

[0087] 1.解析目标文件1各个要素，解析结果用各中间结构表示。以如下结构描述ELF文件：

```
struct ElfObjectFile
{
unsigned char* buffer; //存储文件内容
unsigned size; //文件大小
Elf_Ehdr*fhdr; //指向文件头
```

[0088]

[0089]
```
Elf_Shdr*shdr; //指向节区头
unsigned sec_count; //节区数量
char *sec_str; //节区名字区域起始地址
Elf_Sym*symbol; //指向符号表
unsigned sym_count; //符号表数量
char *str; //字符串表起始地址
unsigned str_size; //字符串表字节数
};
```

[0090] 文件载入及基本构成解析是基于ELF文件格式的技术常识，不再赘述。

[0091] 2.分解目标文件1。

[0092] 从ElfObjectFile结构中提取各要素并转换为VmObject表，如下所示：

[0093]
```
class ElfConverter
:public ElfObjectFile
{
public:
// 解析ELF文件，转换为中间格式的VmObjectFile
void ConvertTo(VmObjectList& List);
private:
// 查找指定段的重定位段，可遍历所有重定位段，根据
Elf_Ehdr::st_info判断
int GetRelocSection(int SectionIndex);
};
// 解析ELF文件，转换为中间格式的VmObjectFile
void ElfConverter::ConvertTo(VmObjectList& List)
{
for (unsigned i=0; i<sym_count; ++i)
{
Elf_Sym& sym = symbol[i];
unsigned char type = sym.st_info & 0x0f;
```

```
if (type == STT_SECTION || type == STT_FILE)
{
continue;
}
VmObject* obj = new VmObject;
// 名字
obj->name = str + sym.st_name;
// 偏置
obj->offset = 0;
// 数据
if (sym.st_shndx == SHN_COMMON)
{
obj->body = NULL;
}
else
```

[0094]

```
{
obj->body = (char*)buffer + shdr[sym.st_shndx].sh_offset +
sym.st_value;
}
// 大小
obj->size = sym.st_size;
// 类型，内存属性
if (type == STT_FUNC)
{
obj->type = VMT_FUNC;
obj->attr = VMM_EXE;
}
else
{
obj->type = VMT_VAR;
if (shdr[sym.st_shndx].sh_flags & SHF_WRITE)
```

```
{
obj->attr = VMM_RW;
}
else
{
obj->attr = VMM_RO;
}
}
// 内存属性
if (sym.st_size == 0)
{
obj->attr = VMM_EXTERNAL;
}
// 符号绑定类型
switch (sym.st_info >> 4)
```

[0095]

```
{
case STB_LOCAL: obj->binding = VMB_STATIC;break;
case STB_GLOBAL: obj->binding = VMB_GLOBAL;break;
case STB_WEAK: obj->binding = VMB_WEAK;break;
default: obj->binding = VMB_UNKOWN;
}
// 添加重定位
int RelocIndx = GetRelocSection(sym.st_shndx);
Elf_Shdr& RelocSec = shdr[RelocIndx];
unsigned RelocCount = RelocSec.sh_size/sizeof(Elf32_Rel);
for (unsigned i=0; i<RelocCount; ++i)
{
Elf32_Rel& rel = ((Elf32_Rel*)buffer + RelocSec.sh_offset)[i];
if (rel.r_offset >= sym.st_value && rel.r_offset < sym.st_value
+ sym.st_value)
{
```

```
        VmReloc reloc;
        reloc.offset = rel.r_offset - sym.st_value;
        reloc.value = *(unsigned*)(buffer + rel.r_offset);
        reloc.type = (unsigned char)rel.r_info;
        reloc.symname = str + (rel.r_info >> 8);
[0096]  obj->relocs.push_back(reloc);
       }
       }
       List.push_back(obj);
       }
           }
```

[0097] 3.更新符号名。

[0098] 不同目标文件格式有着不同的符号名命名方式，一般函数、变量符号名中都包含了类名、命名空间、类型等信息，可解析这些信息，并根据新规则重命名符号。可分两个步骤完成：

[0099] 1)根据目标文件1的符号命名格式解析名字、类型等信息。

[0100] 2)根据名字、类型等信息和目标文件2的符号命名规则构造新符号名。由于在具体实现上重定位直接使用了符号名作为重定位目标(而不是符号索引)，所以应同时刷新重定位的目标符号名。

[0101] C语言的命名较为简单，ELF文件中对符号名未加修饰，即为其函数或变量名，COFF文件中，一般为函数或变量名前面加“_”。其实现如下述的RenameELFName所示：

[0102] //解析并重命名从ELF文件中解析出来的VmObject

[0103] //参数:List，从ELF文件中解析出来的所有VmObject列表

[0104] //参数:obj,要重命名的VmObject

[0105] //本函数需将所有List中的重定位的符号名VmRelocation::symname由就名字换为新名字。

[0106]

```
void RenameELFName( const VmObjectList& List, VmObject* obj )
{
::std::string OldName = obj->name;
obj->name = "_" + obj->name;
```

[0107]

```
RefreshVmList(List, OldName.c_str(), obj->name.c_str());
}
// 刷新重定位目标符号名
void RefreshVmList( const VmObjectList& List, const char*
OldName, const char* NewName )
{
for (unsigned i=0; i<List.size(); ++i)
{
VmObject* obj = List[i];
for (unsigned j=0; j<obj->relocs.size(); ++j)
{
VmReloc& reloc = obj->relocs[j];
if (reloc.symname == OldName)
{
reloc.symname = NewName;
}
}
}
}
```

[0108] 4.构造新目标文件。

[0109] 以如下结构表示段、符号重定位和coff文件：

[0110]

```
struct CoffRelocation;
class CCoffSection;
class CoffSymbol;

typedef ::std::vector<CoffRelocation> RelocList;
typedef ::std::vector<CCoffSection> SectionList;
typedef ::std::vector<CoffSymbol> SymbolList;
typedef ::std::vector<char> BinaryData;

// 段
```

```
class CCoffSection
:public IMAGE_SECTION_HEADER
{
public:
RelocListrels;//重定位表
BinaryData  data;//数据
public:
// 在末尾附加数据
unsigned append(const void* _content, int size);
};
// 符号
struct CoffSymbol
{
char     Name[8];
uint32_t Value;
[0111]  uint16_t SectionNumber;
uint16_t Type;
uint8_t  StorageClass;
uint8_t  NumberOfAuxSymbols;
};
// 重定位
struct CoffRelocation {
uint32_t VirtualAddress;
uint32_t SymbolTableIndex;
uint16_t Type;
};

// COFF目标文件，包括从VmObject表构造的方法
class CoffObjectFile
{
public:
```

```
IMAGE_FILE_HEADER fheader;//文件头
IMAGE_OPTIONAL_HEADER oheader;//可选头

SectionList sections;//段
SymbolList  symbols;//符号表
BinaryData  str;//字符串表
int TextSectionIndex;//代码段索引
int DataSectionIndex;//数据段索引
int BssSectionIndex;//未初始化数据段索引
public:
// 根据VmObject列表构建目标文件
void BuildByVmObject(const VmObjectList& List);
// 输出到文件
void Output(const char* name);
private:
```

[0112]

```
// 将VmObject的数据添加到本类，并创建符号
void AddVmSymbol(VmObject& obj);
// 将VmObject中的重定位添加到本类
void AddVmRelocation(const VmObject& obj);
//************************************
// Returns:   int,  符号索引
// Parameter: const char * name, 符号名
// Parameter: unsigned secIndex, 段索引
// Parameter: unsigned offset, 在段内的偏置
// Parameter: int type, 符号类型
// Parameter: int storageClass, 作用域类型
//************************************
int   addSymbolItem(const char* name, unsigned secIndex,
unsigned offset
, int type, int storageClass);
// 创建代码段，返回段索引
```

```
int  CreateTextSection();
// 创建数据段，返回段索引
int  CreateDataSection();
// 创建未初始化数据，返回段索引
int  CreateBssSection();
// 创建drectve段，返回段索引
int  CreateDrectveSection();
// 查找符号，返回符号索引
int  FindSymbol(const char* name);
};
```

主要流程如下述的 CoffObjectFile::BuildByVmObject 所示：

```
// 根据VmObject列表构建目标文件
void  CoffObjectFile::BuildByVmObject(  const  VmObjectList&
List )
```

[0113]

```
{
// 创建段
TextSectionIndex = CreateTextSection();
DataSectionIndex = CreateDataSection();
BssSectionIndex  = CreateBssSection();
// 重命名符号名
for (unsigned i=0; i<List.size(); ++i)
{
VmObject* obj = List[i];
RenameELFName(List, obj);
}
// 添加数据和符号
for (unsigned i=0; i<List.size(); ++i)
{
VmObject* obj = List[i];
AddVmSymbol(*obj);
```

```
}
// 添加重定位
for (unsigned i=0; i<List.size(); ++i)
{
VmObject* obj = List[i];
```

[0114]

```
AddVmRelocation(*obj);
}
// 添加其它符号
VmObject obj = MakeFltVmObject();
AddVmSymbol(obj);
    }
```

[0115] 1)、创建代码段、数据段、未初始化数据段等。具体实现完全是基于目标文件格式的技术常识，不赘述。

[0116] 2)、添加数据，构造符号表。其实现如CoffObjectFile::AddVmSymbol所示：

```
// 将VmObject除重定位的部分添加到本类
void CoffObjectFile::AddVmSymbol( VmObject& obj )
{
int indx = -1;
if (obj.type == VMT_FUNC && obj.size > 0)
{
indx = TextSectionIndex;
}
```

[0117]

```
else if (obj.type == VMT_VAR && obj.body)
{
indx = DataSectionIndex;
}
else if (!obj.body && obj.size > 0)
{
indx = BssSectionIndex;
}
```

[0118]

```
else
{
// 外部定义的符号
}
CCoffSection& sec = sections[indx];
unsigned offset = 0;
if (indx >= 0)
{
sec.append(obj.body, obj.size);
}
int SymbolIndx = addSymbolItem(obj.name.c_str(), indx, offset
, obj.type == VMT_FUNC ? 0x20/*函数*/ : 0/*非函数*/
, obj.binding == VMB_GLOBAL ? 3/*静态*/ : 2/*全局*/);
obj.indx = indx;
obj.offset = SymbolIndx;
    }
```

[0119] 3)、添加重定位，其实现如CoffObjectFile::AddVmRelocation所示。

```
// 添加重定位
void CoffObjectFile::AddVmRelocation( const VmObject& obj )
{
for (unsigned i=0; i<obj.relocs.size(); ++i)
{
const VmReloc& reloc = obj.relocs[i];
```

[0120]

```
int SymbolIndx = FindSymbol(reloc.symname.c_str());
CoffRelocation reloc2;
reloc2.VirtualAddress = reloc.value;
reloc2.SymbolTableIndex = SymbolIndx;
reloc2.Type = (reloc.type == VMR_DIR32 ?
 IMAGE_REL_I386_DIR32 : IMAGE_REL_I386_REL32);
sections[obj.indx].rels.push_back(reloc2);
```

```
    }
```

[0121]

```
        }
```

[0122] 4)、添加其它与目标文件格式及运行时库有关的必要要素。对COFF而言需要添加.drectve段以指定链接选项，加入__fltused符号以指定浮点运算。其实现如MakeFltVmObject函数所示。

```
VmObject MakeFltVmObject()
{
VmObject obj;
obj.name = "__fltused";
obj.type = VMT_VAR;
obj.attr = VMM_EXTERNAL;
```

[0123]

```
obj.binding = VMB_GLOBAL;
obj.body = NULL;
obj.size = 0;
obj.indx = 0;
obj.offset = 0;
return obj;
    }
```

[0124] 本发明的跨平台目标文件复用方法的有益效果在于：通过对异构目标文件格式进行符号和结构转换，构造匹配用户所用开发工具的新的目标文件，无需改动编译器，使不同编译器生成的目标文件可以复用。通过本发明提供的方法，可以解决不同编译器目标文件格式不同，不能兼容的问题，具有可以提高兼容性、简化操作、节省成本的有益效果。

[0125] 以上实施例仅为本发明的示例性实施例，不用于限制本发明，本发明的保护范围由权利要求书限定。本领域技术人员可以在本发明的实质和保护范围内，对本发明做出各种修改或等同替换，这种修改或等同替换也应视为落在本发明的保护范围内。

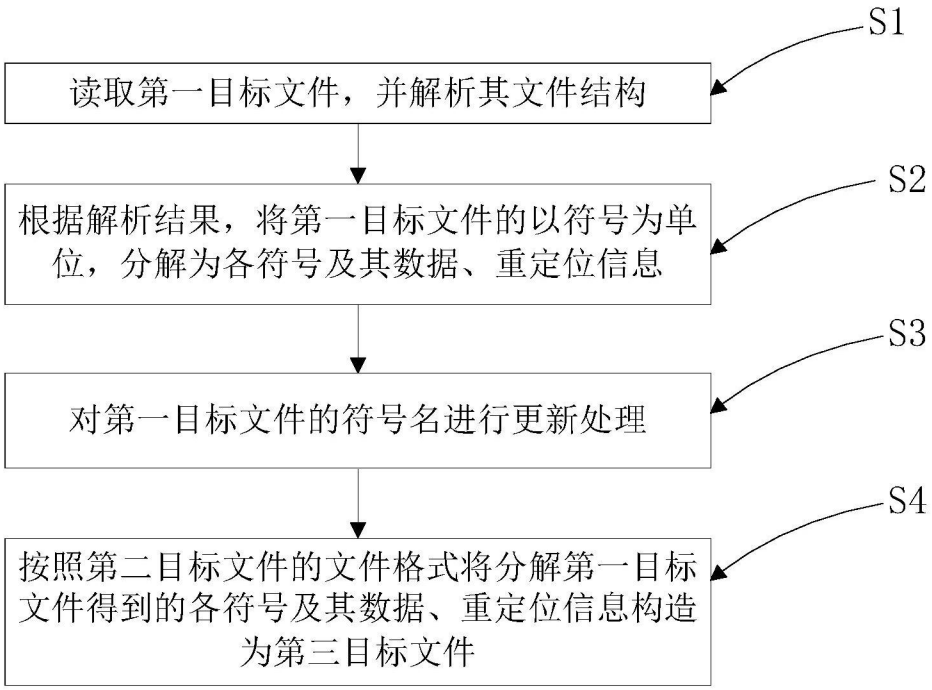

S1
读取第一目标文件，并解析其文件结构
S2
根据解析结果，将第一目标文件的以符号为单位，分解为各符号及其数据、重定位信息
S3
对第一目标文件的符号名进行更新处理
S4
按照第二目标文件的文件格式将分解第一目标文件得到的各符号及其数据、重定位信息构造为第三目标文件

图1