GCC编译工作流程

曾雷杰

2025年3月28日

简单例子（一个Project内部含三个文件）

add.h

```
#ifndef __ADD_H__
#define __ADD_H__

int add(int a, int b);

#define ADD(a, b) ((a) + (b))

#endif
```

add.c

```
#include "add.h"

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

main.c

```
#include "add.h"

int g1 = 10;
int g2;

int main()
{
    int a = 10;
    int c;

    c = add(a, g1);

    c = ADD(c, g2);

    return c;
}
```

如何编译成可执行程序

IDE

make

nmake

qmake

cmake

GCC编译-常见方式

头文件没有直接传入给GCC编译器，它们参与编译了吗？

GCC工作流程-动态链接编译

编译程序:

```
$gcc -o main main.c add.c
$./main
$echo $?
```

查看文件属性:

```
$file main
main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=d94481eab5583bf110ae0fd5291f358eb2eac0d6, not stripped
```

可执行程序依赖:

```
$ldd main
linux-vdso.so.1 =>  (0x00007ffdcbb7e000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3fcf24b000)
/lib64/ld-linux-x86-64.so.2 (0x00007f3fcf615000)
```

编译详细内部:

```
$gcc -o main main.c add.c --verbose
```

简化结果:

```
cc1 main.c -o /tmp/ccjLnMMc.s
as -o /tmp/cc3FZAno.o /tmp/ccjLnMMc.s

cc1 add.c -o /tmp/ccjLnMMc.s
as -o /tmp/ccH4iZ0z.o /tmp/ccjLnMMc.s

collect2 -o main crt1.o crti.o crtbegin.o
 /tmp/cc3FZAno.o /tmp/ccH4iZ0z.o -lgcc -lgcc_s -lc
crtend.o crtn.o
```

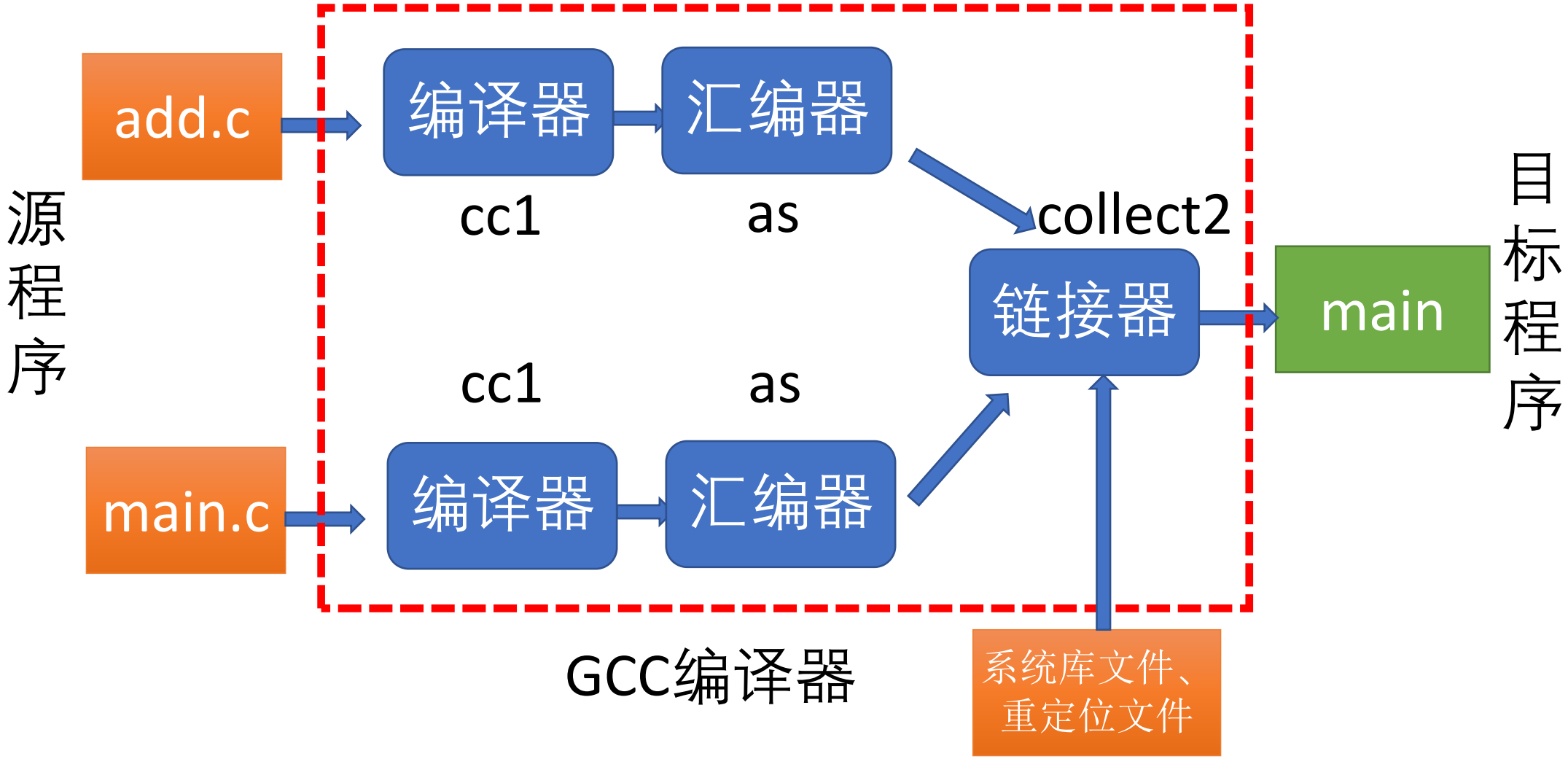
编译器 (Compiler)

汇编器 (Assembler)

链接器 (Linker)

GCC编译-内部详细过程

串行编译



GCC工作流程-静态链接编译

静态链接方式编译程序:
$gcc -o main main.c add.c -static
$./main
$echo $?

查看文件属性:
$file main
main: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), **statically linked**, for GNU/Linux 2.6.32, BuildID[sha1]=a93638bb291986c00fee9f3e6bae1084df17ff3c, not stripped

可执行程序依赖:
$ldd main
not a dynamic executable

静态编译详细:
$gcc -o main main.c add.c --verbose

简化结果:

cc1 main.c -o /tmp/ ccc17E93.s
as -o /tmp/ ccOLyWRN.o /tmp/ccc17E93.s

cc1 add.c -o /tmp/ccc17E93.s
as -o /tmp/ ccHY20Cx.o /tmp/ccc17E93.s

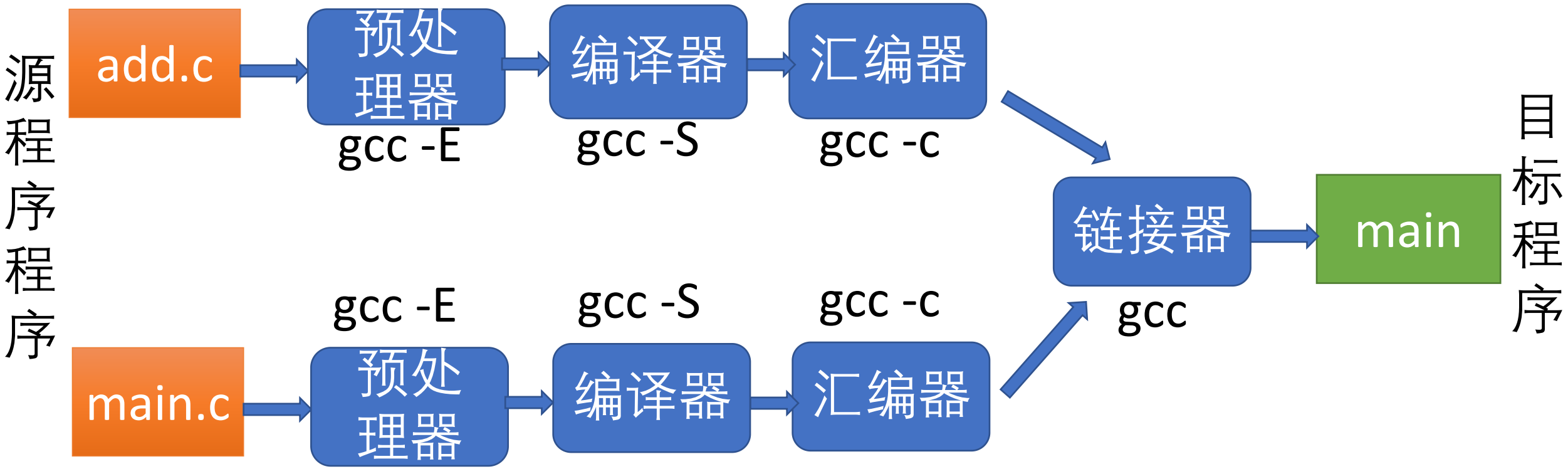
collect2 -o main **-static** crt1.o crti.o crtbeginT.o /tmp/ ccOLyWRN.o /tmp/ ccHY20Cx.o --start-group -lgcc -lgcc_eh -lc --end-group crtend.o crtn.o

无第三方程序依赖

静态链接与动态链接区别

- 可执行程序的格式： dynamically/statically linked
- 在编译的过程中cc1和as的过程都一样，在链接时有区别。默认动态链接，-static指定时静态链接。
- 动态链接的可执行程序依赖第三方动态库，所有依赖的动态库在单独存在，不包含在可执行程序中。程序运行时动态库会动态加载到内存中。
- 静态链接的可执行程序不依赖第三方库，所有依赖的静态库都已经内嵌在可执行程序中，程序文件大。

GCC编译-详细方式



GCC编译过程

GCC工作流程-详细-预处理器(Preprocessor)

- 预处理器，主要是处理宏定义和文件包含等信息，例如#include所包含文件拷贝到C文件中，#define宏展开；预处理宏#ifdef等启用。

- 采用gcc -E选项可对C程序预处理

- 具体命令

  gcc -E -o main.i main.c

  gcc -E -o add.i add.c

请问预处理指令是C语言的合法语句吗？

```
#ifndef __ADD_H__
#define __ADD_H__

int add(int a, int b);

#define ADD(a,b) ((a)+(b))

#endif
```

```
#include "add.h"

int g1 = 10;
int g2;

int main()
{
    int a = 10;
    int c;

    c = add(a, g1);

    c = ADD(c, g2);

    return c;
}
```

```
int add(int a, int b);
# 2 "main.c" 2

int g1 = 10;
int g2;

int main()
{
    int a = 10;
    int c;

    c = add(a, g1);

    c = ((c)+(g2));

    return c;
}
```

能否把add.h中的由X指示的代码删除？

本项目能否编译成可执行程序？为何？

test.h

```c
#ifndef __TEST_H__
#define __TEST_H__

int add(int a, int b)
{
    return a + b;
}

int test01(int a, int b);

#endif
```

test.c

```c
#include "test.h"

int test01(int a, int b)
{
    return add(a, b);
}
```

main.c

```c
#include "test.h"

int main()
{
    int a = 10;
    int c;

    c = add(a, 10);
    c = test01(c, 20);

    return c;
}
```

修改成这样，OK?

test.h

```
#ifndef __TEST_H__
#define __TEST_H__

int add(int a, int b);
int test01(int a, int b);

#endif
```

test.c

```
#include "test.h"

int test01(int a, int b)
{
    return add(a, b);
}

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

main.c

```
#include "test.h"

int main()
{
    int a = 10;
    int c;

    c = add(a, 10);
    c = test01(c, 20);

    return c;
}
```

改成这样，OK?

test.h

```
#ifndef __TEST_H__
#define __TEST_H__

static int add(int a, int b)
{
    return a + b;
}

int test01(int a, int b);

#endif
```

test.c

```
#include "test.h"

int test01(int a, int b)
{
    return add(a, b);
}
```

main.c

```
#include "test.h"

int main()
{
    int a = 10;
    int c;

    c = add(a, 10);
    c = test01(c, 20);

    return c;
}
```

GCC工作流程-详细-编译器(Compiler)

- 主要功能是对源程序进行翻译，目标是汇编语言程序。其翻译主要包含把字符流(文本文件）形成记号（Token）流或符号流、检查是否满足C语言的文法要求、进行语义转换、代码优化、生成目标汇编代码等。

- 采用gcc -S选项可对C程序编译。可追加-O n进行代码优化

- 具体命令

```
gcc -S -o main.s main.c
gcc -S -o add.s add.c
gcc -S -o main.s main.i
gcc -S -o add.s add.i
```

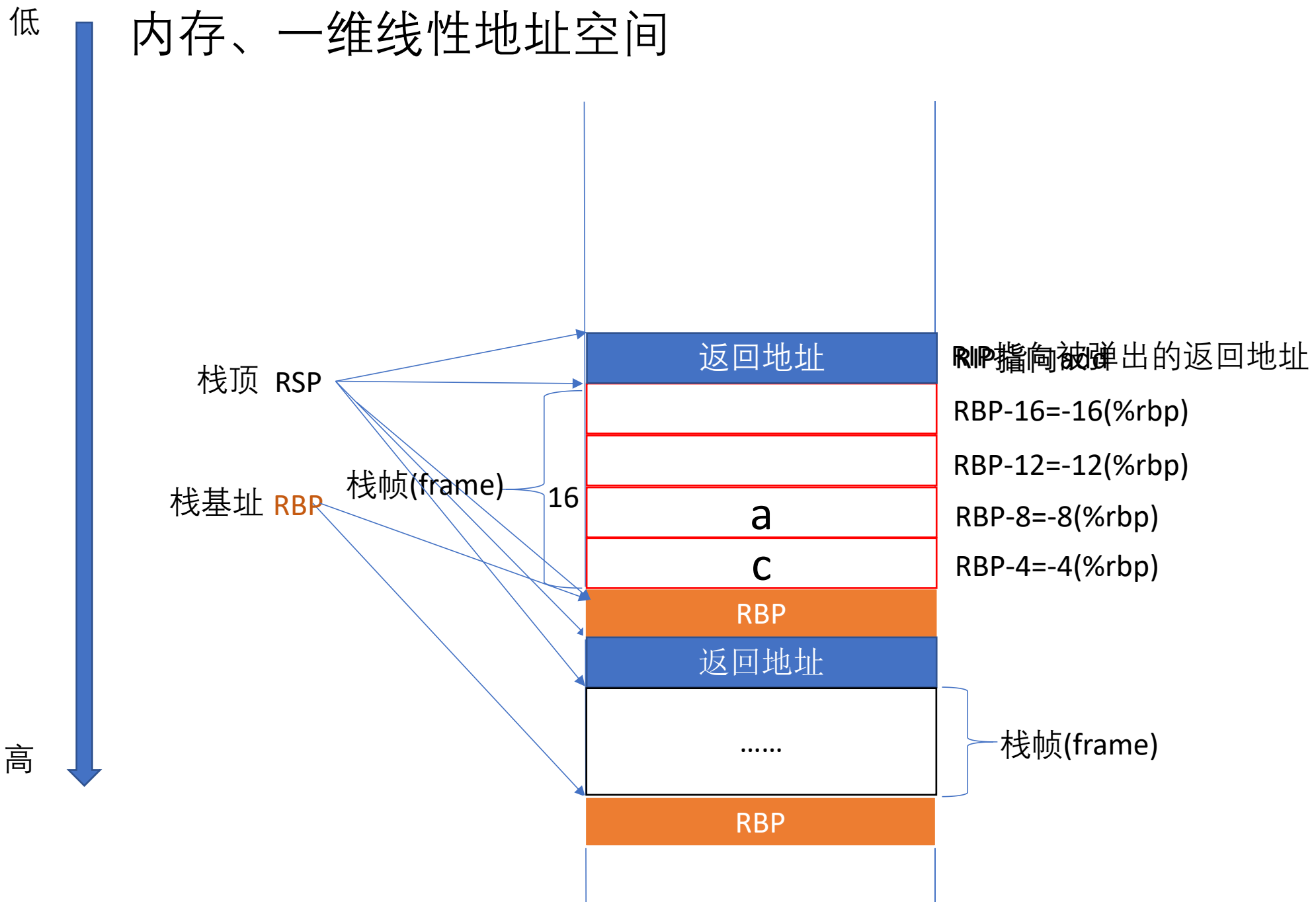
```c
int add(int a, int b);
# 2 "main.c" 2

int g1 = 10;
int g2;

int main()
{
    int a = 10;
    int c;

    c = add(a, g1);

    c = ((c)+(g2));

    return c;
}
```

```asm
    .file"main.c"
    .globl  g1
    .data
    .align 4
    .typeg1, @object
    .sizeg1, 4
g1:
    .long10
    .commg2,4,4
    .text
    .globl  main
    .typemain, @function
main:
    pushq%rbp
    movq %rsp, %rbp
    subq $16, %rsp
    movl $10, -8(%rbp)
    movl g1(%rip), %edx
    movl -8(%rbp), %eax
    movl %edx, %esi
    movl %eax, %edi
    call add
    movl %eax, -4(%rbp)
    movl g2(%rip), %eax
    addl %eax, -4(%rbp)
    movl -4(%rbp), %eax
    leave
    ret
```

.data开始的数据段，包含两个全局变量g1和g2，g1被初始化，g2没有初始化

.text开始的代码段，存储让CPU执行的一系列指令

局部变量保存在什么地方？函数如何调用？




```
push %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $10, -8(%rbp)
movl g1(%rip), %edx
movl -8(%rbp), %eax
movl %edx, %esi
movl %eax, %edi
call add
movl %eax, -4(%rbp)

……

Leave
⇔
movl %rbp, %rsp
pop %rbp
ret
```

其它中间形式的语言

- GCC可生成RTL形式的中间语言

gcc -S -fdump-tree-all main.c

```c
#include "add.h"

int g1 = 10;
int g2;

int main()
{
    int a = 10;
    int c;

    c = add(a, g1);

    c = ADD(c, g2);

    return c;
}
```

CFG

```
main ()
{
  int c;
  int a;
  int D.1941;

  <bb 2> :
  a = 10;
  g1.0_1 = g1;
  c = add (a, g1.0_1);
  g2.1_2 = g2;
  c = c + g2.1_2;
  D.1941 = c;

  <bb 3> :
<L0>:
  return D.1941;

}
```

SSA

```
main ()
{
  int c;
  int a;
  int D.1941;
  int g1.0_1;
  int g2.1_2;
  int _8;

  <bb 2> :
  a_3 = 10;
  g1.0_1 = g1;
  c_6 = add (a_3, g1.0_1);
  g2.1_2 = g2;
  c_7 = c_6 + g2.1_2;
  _8 = c_7;

  <bb 3> :
<L0>:
  return _8;

}
```

GCC工作流程-详细-汇编器(Assembler)

- 主要功能是对汇编源程序进行翻译，目标是可重定位的目标文件。目标文件不能通过文本编辑器查看，不过可以通过objdump命令分析它的内容。

- 采用gcc -c或者as命令可对汇编源程序进行汇编

- 具体命令

  gcc -c -o main.o main.s

  gcc -c -o add.o add.s

main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

objdump -f main.o 文件头信息

重定位

```
main.o:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
```

程序的开始地址
未知,需要重定位

objdump -x main.o Section信息

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000032  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  0000000000000000  0000000000000000  00000074  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000078  2**0
                  ALLOC
  3 .comment      00000036  0000000000000000  0000000000000000  00000078  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000ae  2
                  CONTENTS, READONLY
  5 .eh_frame     00000038  0000000000000000  0000000000000000  000000b0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     O .data  0000000000000004 g1
0000000000000004       O *COM*  0000000000000004 g2
0000000000000000 g     F .text  0000000000000032 main
0000000000000000         *UND*  0000000000000000 add

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000011 R_X86_64_PC32     g1-0x0000000000000004
000000000000001d R_X86_64_PC32     add-0x0000000000000004
0000000000000026 R_X86_64_PC32     g2-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

.text 代码段，CPU指令，只读

.data 数据段，初始化全局变量或者静态变量等

.bss未初始化段，包含未初始化的全局变量或者静态变量等

符号表，主要包含Section符号、函数符号等

重定位记录，这些需要重定位，地址还不是最终的地址

objdump -d main.o CPU指令序列

地址都是0，需要在链接时重定位

```
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 f8 0a 00 00 00    movl   $0xa,-0x8(%rbp)
   f:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 15 <main+0x15>
  15:   8b 45 f8                mov    -0x8(%rbp),%eax
  18:   89 d6                   mov    %edx,%esi
  1a:   89 c7                   mov    %eax,%edi
  1c:   e8 00 00 00 00          callq  21 <main+0x21>
  21:   89 45 fc                mov    %eax,-0x4(%rbp)
  24:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 2a <main+0x2a>
  2a:   01 45 fc                add    %eax,-0x4(%rbp)
  2d:   8b 45 fc                mov    -0x4(%rbp),%eax
  30:   c9                      leaveq
  31:   c3                      retq
```

```
main:
    pushq%rbp
    movq %rsp, %rbp
    subq $16, %rsp
    movl $10, -8(%rbp)
    movl g1(%rip), %edx
    movl -8(%rbp), %eax
    movl %edx, %esi
    movl %eax, %edi
    call add
    movl %eax, -4(%rbp)
    movl g2(%rip), %eax
    addl %eax, -4(%rbp)
    movl -4(%rbp), %eax
    leave
    ret
```

请与汇编程序对比看看有何不同？

GCC工作流程-详细-链接器(Linker)

- 主要功能是把多个可重定位目标文件或者动态库或静态库进行符号地址值确定、多个目标文件合并成可执行程序。

- 采用gcc或者ld命令执行链接，若静态链接则指定-static选项

- 具体命令

  gcc -o main main.o add.o

main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=db918c4b0e6b1b942634d454ba5cb0d2fe6832da, not stripped

objdump -f main 文件头信息

```
main:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004003e0
```

取消了可重定位标志，设置为可执行。
新版编译器可能不存在此标志

程序的开始地址或者入口地址已经设置为有效的地址

objdump -x main Section节选信息

.text 代码段

.rodata 只读段

.data 数据段

.bss未初始化段

Section大小

虚拟地址

逻辑地址

文件中的位置

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
 10 .init         0000001a  0000000000400390  0000000000400390  00000390  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .text         000001b2  00000000004003e0  00000000004003e0  000003e0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .rodata       00000004  00000000004005a0  00000000004005a0  000005a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 24 .data         00000014  0000000000601020  0000000000601020  00001020  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 25 .bss          0000000c  0000000000601034  0000000000601034  00001034  2**2
                  ALLOC
```

g1变量的地址

g2变量的地址

main函数的地址

可执行程序的入口地址

```
SYMBOL TABLE:
00000000004003e0 l    d  .text  0000000000000000              .text
0000000000601030 g     O .data  0000000000000004              g1
00000000004003e0 g     F .text  000000000000002a              _start
0000000000601038 g     O .bss   0000000000000004              g2
00000000004004d6 g     F .text  0000000000000032              main
```

objdump -d main CPU指令序列

```
00000000004004d6 <main>:
  4004d6:       55                      push   %rbp
  4004d7:       48 89 e5                mov    %rsp,%rbp
  4004da:       48 83 ec 10             sub    $0x10,%rsp
  4004de:       c7 45 f8 0a 00 00 00    movl   $0xa,-0x8(%rbp)
  4004e5:       8b 15 45 0b 20 00       mov    0x200b45(%rip),%edx
  4004eb:       8b 45 f8                mov    -0x8(%rbp),%eax
  4004ee:       89 d6                   mov    %edx,%esi
  4004f0:       89 c7                   mov    %eax,%edi
  4004f2:       e8 11 00 00 00          callq  400508 <add>
  4004f7:       89 45 fc                mov    %eax,-0x4(%rbp)
  4004fa:       8b 05 38 0b 20 00       mov    0x200b38(%rip),%eax
  400500:       01 45 fc                add    %eax,-0x4(%rbp)
  400503:       8b 45 fc                mov    -0x4(%rbp),%eax
  400506:       c9                      leaveq
  400507:       c3                      retq

0000000000400508 <add>:
  400508:       55                      push   %rbp
  400509:       48 89 e5                mov    %rsp,%rbp
  40050c:       89 7d fc                mov    %edi,-0x4(%rbp)
  40050f:       89 75 f8                mov    %esi,-0x8(%rbp)
  400512:       8b 55 fc                mov    -0x4(%rbp),%edx
  400515:       8b 45 f8                mov    -0x8(%rbp),%eax
  400518:       01 d0                   add    %edx,%eax
  40051a:       5d                      pop    %rbp
  40051b:       c3                      retq
  40051c:       0f 1f 40 00             nopl   0x0(%rax)
```

```
0000000000601030 g     O .data  0000000000000004              g1
```

```
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
movl   $0xa,-0x8(%rbp)
mov    0x0(%rip),%edx
mov    -0x8(%rbp),%eax
mov    %edx,%esi
mov    %eax,%edi
callq  21 <main+0x21>
mov    %eax,-0x4(%rbp)
mov    0x0(%rip),%eax
add    %eax,-0x4(%rbp)
mov    -0x4(%rbp),%eax
leaveq
retq
```

%

=(

=(

)0b45

请与可重定位程序进行对比看看有何不同?

GCC工作流程-并行+批处理

```
cmake_minimum_required(VERSION 3.25)

project(example01 C)

add_executable(example01 main.c add.c add.h)
```

编写CMakeLists.txt文件，可借助如下的命令实现两个c文件的并行编译和串行链接。请关注--parallel 2选项，会开启两个进程进行编译；若2省略，则采用主机上的全部CPU并行编译。

```
cmake -B build -S . -DCMAKE_BUILD_TYPE=Debug
cmake --build build --parallel 2
```