

细谈ABI (Application Binary interface)

spider集控团队 2020-11-12 8,268

关注

ABI (Application Binary interface)

ABI (Application Binary interface)：应用程序二进制接口，描述了应用程序和操作系统之间，一个应用和它的库之间，或应用的组成部分之间的底层接口。

API与ABI的区别

ABI 从名字上看是二进制接口，而二进制文件再 **linux** 以 **ELF** 文件类型表示，**windows** 以 **PE-COFF** 文件类型表示。

ELF文件类型	说明	实例
可重定位文件 (Relocatable File)	包含了代码和数据	如Linux的.o、.a windows的.obj、.lib
可执行文件 (Executable File)	包含了可以直接执行的程序	如/bin/lsh 文件 windows的.exe
共享目标文件 (Shared Object File)	包含了代码和数据	Linux 的.so windows的dll

二进制文件的生成是通过编译器或者链接器，那么 **API** 和 **ABI** 都是谁需要去遵循这个规则呢，如下面的代码假设它将会被编译成一个 **myso** 动态库，你可以将它当成一个 **API**。

c++ 复制代码

```
1 int Add(int a, int b) {  
2     return a + b;  
}
```

```
3 }
```

下面是你的应用程序, 我们称它为 `main`

```
1 int main(void) {  
2     int c = Add(3, 2);  
3     return 0;  
4 }
```

c++ 复制代码

- API: 库的使用者可能需要去遵循这个接口规范, `Add` 函数的参数个数以及参数类型等等。
- ABI: `main` 使用到了 `Add` 这个 API, 这个 API 包含再一个 `myso` 动态库里面, 现在设计到一个符号寻找机制, 即编译器需要去 `myso` 动态库里面寻找 `Add` 这个符号, 那符号的命名规则不一致会导致什么结果? 如 `gcc1.0` 版本的符号命名规则是再函数前面加一个 `_`, 即最后 `Add` 符号名称 `_Add`, `gcc2.0` 版本的符号命名规则是再函数后面加一个 `_`, 即最后 `Add` 符号名称 `Add_`。思考一个问题, `myso` 是利用 `gcc1.0` 版本编译, `main` 使用 `gcc2.0` 版本编译, 会出现什么问题? 编译器会提示你 `Add_` 符号未定义, 这里说的符号导出规则也就是属于 ABI 兼容问题。

结论: 影响你 API 不兼容的可能是你使用的 API 新增了参数。影响 ABI 不兼容的可能仅仅就是编译器版本不同, 一个是源码层面, 一个是编译器层面 (或者说二进制层面, 即编译器生成的二进制), 当然编译器仅仅只是导致 ABI 不兼容的一个方面。

影响ABI兼容性的因素

硬件 - 如处理器

思考一个非常简单的问题, 最近 `Apple` 发布了最新款 `Mac` 笔记本, 号称可以直接使用 `iPhone` 和 `ipad` 的应用, 怎么做到的? 这个就是一个二进制兼容问题, `Apple` 再最新的 `Mac` 笔记本上放弃了之前一直使用的 `intel` 芯片, 从而采用自研的 `M1` 芯片, 这个 `M1` 的自研芯片架构就是 `ARM` 架构和苹果 `A` 系列芯片架构一样, 从而才有可能实现二进制级别的兼容。

二进制里面包含了指令和数据, 而 `CPU` 有一个核心作用就是处理指令, 不同架构的 `CPU` 指令集都不同, 从而产生的二进制也会不同, 例如你在代码中调用了 `print` 函数, 最终

在 X86 生成的二进制文件的一条指令是 `call 0x1234`，但是在 ARM 处理器下它可能没有 `call` 指令，它的跳转指令可能是 `jar`。

操作系统

为什么不同系统不能兼容同一个已编译的可执行二进制文件（假设在统一架构）？

一、二进制文件类型

一个可执行的二进制文件包含的不仅仅是机制指令，还包括各种数据、程序运行资源。

如上面提到的二进制文件类型：

- windows - PE-COFF
- linux - ELF
- macos - MACH-O

它们的二进制文件格式各不相同，导致 windows 无法解析 linux 下的 ELF 文件格式，从而无法完成可执行文件在执行之前的一系列初始化操作，如 ELF 文件头中就包含了：

成员	readelf输出结果与含义
e_ident	Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 Class: ELF64 Data: 2's complement, little endian Version: 1(current) OS/ABI: UNIX-System V ABIVersion: 0
e_type	ELF文件类型
e_machine	ELF文件的CPU平台属性，如X86-64
e_version	ELF版本号，一般是0x1
e_entry	入口点地址，规定了ELF程序的入口虚拟地址，操作系统在加载完该程序后从这个地址开始执行进程的指令
e_phoff	程序头起点
e_shoff	段表在文件中的偏移
e_word	ELF标志位
e_ehsize	ELF文件头本身的大小
e_phentsize	程序头的大小
e_phnum	程序头数量
e_shentsize	段表描述符的大小
e_shnum	段表描述符的数量
e_shstrndx	段表字符串表所在的段在段表中的下标

ELF格式解析不正确，也就不能正常得到上面的数据，程序入口地址都不知道在哪里，程序从何处开始执行？

二、程序库不同 (API)

文件操作、输入输出、内存申请释放、任务调度等都需要用到特定操作系统的特定库。

编译器

如 C++ 函数签名：函数签名的目的就是让编译器能够根据对应的签名规则生成一个符号，编译器根据这个符号来识别和处理函数，函数签名包含了一个函数的信息，其中包括

- 函数名
- 参数类型
- 参数个数
- 类名
- 名称空间

```
1 int Function(int i);
```

C++ 复制代码

上面的代码在 gcc 和 vc 编译器生成之后的符号：

- gcc : _Z8Functioni
- vc++ : ?Function@@YAHH@Z

你会发现gcc和vc++的函数签名规则都不一样，那gcc编译的库vc++能够找到它的符号吗，答案肯定是不行的，就算是相同版本的gcc也一样可能出现二进制不兼容，如gcc4.9版本C++ string,list符号命名和gcc5.1之后的符号命名都是不同的gcc5.1上会增加__cxx11，所以一样会产生在gcc4.9编译的库，再gcc5.1上使用不了（符号未定义，如果使用了string, list）

语言层面 - C++

- 内置类型的大小以及对齐方式（如大端、小端）。
- struct、union、数组等的存储方式和内存分布。


- 函数调用方式，比如参数入栈顺序、返回值如何保持等。
- 堆栈的分布方式，比如参数和局部变量在堆栈里的位置，参数传递方法等。
- 继承类体系的内存分布，如基类、虚基类再继承类中的位置等。
- 指向成员函数的指针的内存分布，如何传递this指针
- 如何调用虚函数， `vtable` 的内容和分布形式， `vtable` 指针在 `object` 中的位置等。
- `template` 如何实例化
- 外部符号的修饰
- 全局对象的构造和析构
- 异常的产生和捕获机制
- `RTTI` 如何实现

等等

总结

想要保持二进制兼容相较于 `API` 兼容来说要难上许多，并且影响 `ABI` 兼容的因素也非常多从硬件到操作系统再到编译器，编程语言等，并且非常难以统一如 `gcc` 和 `vc++`，`windows` 和 `linux`，本文也只是浅析了 `ABI` 兼容问题，如果你是一个公共组件的开发者，相信你还需要更加深入连接 `ABI` 兼容问题，如阅读 `LSB (Linux Standard Base)`。

C++ binary compatibility between Visual Studio 2015, 2017, and 2019

11/18/2019 • 2 minutes to read •  +4

The Microsoft C++ (MSVC) compiler toolsets in Visual Studio 2013 and earlier don't guarantee binary compatibility across versions. You can't link object files, static libraries, dynamic libraries, and executables built by different versions. The ABIs, object formats, and runtime libraries are incompatible.

We've changed this behavior in Visual Studio 2015, 2017, and 2019. The runtime libraries and apps compiled by any of these versions of the compiler are binary-compatible. It's reflected in the C++ toolset major number, which is 14 for all three versions. (The toolset version is v140 for Visual Studio 2015, v141 for 2017, and v142 for 2019). Say you have third-party libraries built by Visual Studio 2015. You can still use them in an application built by Visual Studio 2017 or 2019. There's no need to recompile with a matching toolset. The latest version of the Microsoft Visual C++ Redistributable package (the Redistributable) works for all of them.

There are three important restrictions on binary compatibility:

- You can mix binaries built by different versions of the toolset. However, you must use a toolset at least as recent as the most recent binary to link your app. Here's an example: you can link an app compiled using the 2017 toolset to a static library compiled using 2019, if they're linked using the 2019 toolset.
- The Redistributable your app uses has a similar binary-compatibility restriction. When you mix binaries built by different supported versions of the toolset, the Redistributable version must be at least as new as the latest toolset used by any app component.
- Static libraries or object files compiled using the `/GL` (Whole program optimization) compiler switch *aren't* binary-compatible across versions. All object files and libraries compiled using `/GL` must use exactly the same toolset for the compile and the final link.

@稀土掘金技术社区

参考资料

[C++ binary compatibility between Visual Studio 2015, 2017, and 2019](#)

[Mach-O文件格式](#)

[为什么不同系统不能兼容同一个已编译的可执行二进制文件?](#)

[Application binary interface](#)

[What is an application binary interface \(ABI\)?](#)

[Linux Standard Base Common Definitions](#)

[Itanium C++ ABI](#)

标签: 操作系统