CFFI 文档

发布 *1.12.3*

Armin Rigo, Maciej Fijalkowski, Jairo(翻译者)

2019 年 11 月 18 日

Contents

CFFI(C Foreign Function Interface) 是 Python 的 C 语言外部函数接口。Python 可以与几乎任何 C 语言代码进行交互，基于类似 C 语言的声明，您通常可以从头文件或文档中复制粘贴。

CHAPTER 1

目标

该接口基于 LuaJIT's FFI，并遵循如下一些原则:

- 目标是在不学习第三种编程语言的情况下从 Python 调用 C 语言代码: 现有的替代方案要求用户学习特定领域语言 (Cython，SWIG) 或 API (ctypes)。CFFI 设计目的要求用户只知道 C 和 Python，最大限度地减少需要学习 API 的额外部分。
- 将所有与 Python 相关的逻辑代码保存在 Python 中，这样您就不需要编写很多 C 语言代码 (不同于 CPython 原生 C 扩展).
- 首选方法是在 API (Application Programming Interface) 级别运行: C 语言编译器根据您编写的声明直接链接 C 语言结构。或者，也可以选择 ABI 级别（Application Binary Interface），这种方法通过 ctypes 运行。但是，在非 Windows 平台上，C 语言库通常具有指定的 C API 但不具有 ABI (例如他们可能将"struct" 记录为至少包含这些字段，但可能更多)。
- 尽量完成。目前不支持一些 C99 结构，但所有 C89 结构都应该支持，包括宏 (包括"abuses"，你可以手动包装 在看起来很神奇的 C 语言函数)。
- 尝试同时支持 PyPy 和 CPython，为 IronPython 和 Jython 等其他 Python 实现提供合理的路径。
- 注意，与 Weave **不同**的是，这个项目并不是要在 Python 中嵌入可执行 C 代码。而是从 Python 调用现有的语言库。
- 没有支持 C++。有时，围绕 C++ 代码编写一个 C 包装器然后用 CFFI 调用这个 C API 是合理的。否者，看看其他项目。我会推荐 cppyy，它有一些相似之处 (并且还可以在 CPython 和 PyPy 上高效工作)。

开始阅读 概述.

CHAPTER 2

意见和错误

联系我们的最佳方式是在 `irc.freenode.net` 的 IRC `#pypy` 频道。随意讨论那里或 邮件列表 中的问题。请向 问题跟踪器 报告任何错误。

作为基本规则，当需要解决设计问题时，我们选择” 最像 C” 的解决方案。我们希望这个模块拥有访问 C 语言代码所需的一切，仅此而已。

— 作者，Armin Rigo 和 Maciej Fijalkowski，Jairo(翻译者)

CHAPTER 3

有什么新变化

3.1 v1.12.3

- 修复以 var 大小数组结尾的嵌套结构类型（# 405）。
- 添加对在 `ffi.cdef()` 中整数常量末尾使用 `U` 和 `L` 符的支持（感谢 Guillaume）。
- 更多 3.8 修复。

3.2 v1.12.2

- 添加了临时解决方法以在 CPython 3.8.0a2 上进行编译。

3.3 v1.12.1

- Windows 上的 CPython 3：我们再次默认不再使用 `Py_LIMITED_API` 进行编译，因为这些模块仍然无法与 virtualenv 一起使用。问题是它在 CPython <= 3.4 中不起作用，并且由于技术原因，我们无法根据 Python 的版本自动启用此标志。

  就像之前一样，问题 #350 提到了一个解决方法，如果您仍然需要 `Py_LIMITED_API` 标志，并且您不关心 virtualenv，或者您确定您的模块不会在 CPython <= 3.4 上使用：将 `define_macros=[("Py_LIMITED_API", None)]` 传递给 `ffibuilder.set_source()` 调用。

3.4 v1.12

- 直接支持 pkg-config.
- `ffi.from_buffer()` 接受一个新的可选的第一个参数，该参数给出结果的数组类型。它还需要一个可选的关键字参数 `require_writable` 来拒绝只读 Python 缓冲区。
- `ffi.new()`，`ffi.gc()` 或 `ffi.from_buffer()` 现在可以通过使用 `with` 关键字或通过调用新的 `ffi.release()` 在已知时间释放 cdata 对象。
- Windows，CPython 3.x: cffi 模块再次与 `python3.dll` 链接。这使得它们与 CPython 版本无关，就像它们在其他平台上一样。**它需要 virtualenv 16.0.0**。
- 如果 `p` 本身是另一个 cdata `int[4]`，则接受像 `ffi.new("int[4]", p)` 这样的表达式。
- CPython 2.x: `ffi.dlopen()` 在 Posix 上使用非 ascii 文件名失败
- CPython: 如果一个线程是从 C 启动然后运行 Python 代码 (使用回调或嵌入解决方案)，那么以前版本的 cffi 将包含可能的崩溃和/或内存泄漏。希望这已得到修复 (参见 问题 #362).
- 支持 `ffi.cdef(..., pack=N)`，其中 N 是 2 的幂。在 MSVC 上模拟 `#pragma pack(N)` 的方法。此外，Windows 上的默认值现在为 `pack=8`，就像在 MSVC 上一样。这可能会对角落情况产生影响，尽管我在 CFFI 的背景下无法想到这一点。旧方法 `ffi.cdef(..., packed=True)` 保持不变，相当于 `pack=1` (比如说，像 `int` 这样的字段应该对齐到 1 字节而不是 4 字节)。

3.5 旧版本

注：本文档不对旧版本文档的更新内容进行翻译，如有需要，阅读下面内容或自行翻译

3.5.1 v1.11.5

- Issue #357: fix `ffi.emit_python_code()` which generated a buggy Python file if you are using a `struct` with an anonymous `union` field or vice-versa.
- Windows: `ffi.dlopen()` should now handle unicode filenames.
- ABI mode: implemented `ffi.dlclose()` for the in-line case (it used to be present only in the out-of-line case).
- Fixed a corner case for `setup.py install --record=xx --root=yy` with an out-of-line ABI module. Also fixed Issue #345.
- More hacks on Windows for running CFFI's own `setup.py`.
- Issue #358: in embedding, to protect against (the rare case of) Python initialization from several threads in parallel, we have to use a spin-lock. On CPython 3 it is worse because it might spin-lock for

a long time (execution of `Py_InitializeEx()`). Sadly, recent changes to CPython make that solution needed on CPython 2 too.

- CPython 3 on Windows: we no longer compile with `Py_LIMITED_API` by default because such modules cannot be used with virtualenv. 问题 #350 mentions a workaround if you still want that and are not concerned about virtualenv: pass a `define_macros=[("Py_LIMITED_API", None)]` to the `ffibuilder.set_source()` call.

3.5.2 v1.11.4

- Windows: reverted linking with `python3.dll`, because virtualenv does not make this DLL available to virtual environments for now. See Issue #355. On Windows only, the C extension modules created by cffi follow for now the standard naming scheme `foo.cp36-win32.pyd`, to make it clear that they are regular CPython modules depending on `python36.dll`.

3.5.3 v1.11.3

- Fix on CPython 3.x: reading the attributes `__loader__` or `__spec__` from the cffi-generated lib modules gave a buggy SystemError. (These attributes are always None, and provided only to help compatibility with tools that expect them in all modules.)
- More Windows fixes: workaround for MSVC not supporting large literal strings in C code (from `ffi.embedding_init_code(large_string)`); and an issue with `Py_LIMITED_API` linking with `python35.dll/python36.dll` instead of `python3.dll`.
- Small documentation improvements.

3.5.4 v1.11.2

- Fix Windows issue with managing the thread-state on CPython 3.0 to 3.5

3.5.5 v1.11.1

- Fix tests, remove deprecated C API usage
- Fix (hack) for 3.6.0/3.6.1/3.6.2 giving incompatible binary extensions (cpython issue #29943)
- Fix for 3.7.0a1+

3.5.6 v1.11

- Support the modern standard types `char16_t` and `char32_t`. These work like `wchar_t`: they represent one unicode character, or when used as `charN_t *` or `charN_t[]` they represent a unicode string. The

difference with `wchar_t` is that they have a known, fixed size. They should work at all places that used to work with `wchar_t` (please report an issue if I missed something). Note that with `set_source()`, you need to make sure that these types are actually defined by the C source you provide (if used in `cdef()`).

- Support the C99 types `float _Complex` and `double _Complex`. Note that libffi doesn't support them, which means that in the ABI mode you still cannot call C functions that take complex numbers directly as arguments or return type.
- Fixed a rare race condition when creating multiple `FFI` instances from multiple threads. (Note that you aren't meant to create many `FFI` instances: in inline mode, you should write `ffi = cffi.FFI()` at module level just after `import cffi`; and in out-of-line mode you don't instantiate `FFI` explicitly at all.)
- Windows: using callbacks can be messy because the CFFI internal error messages show up to stderr—but stderr goes nowhere in many applications. This makes it particularly hard to get started with the embedding mode. (Once you get started, you can at least use `@ffi.def_extern(onerror=...)` and send the error logs where it makes sense for your application, or record them in log files, and so on.) So what is new in CFFI is that now, on Windows CFFI will try to open a non-modal MessageBox (in addition to sending raw messages to stderr). The MessageBox is only visible if the process stays alive: typically, console applications that crash close immediately, but that is also the situation where stderr should be visible anyway.
- Progress on support for callbacks in NetBSD.
- Functions returning booleans would in some case still return 0 or 1 instead of False or True. Fixed.
- ffi.gc() now takes an optional third parameter, which gives an estimate of the size (in bytes) of the object. So far, this is only used by PyPy, to make the next GC occur more quickly (issue #320). In the future, this might have an effect on CPython too (provided the CPython issue 31105 is addressed).
- Add a note to the documentation: the ABI mode gives function objects that are *slower* to call than the API mode does. For some reason it is often thought to be faster. It is not!

3.5.7 v1.10.1

(only released inside PyPy 5.8.0)

- Fixed the line numbers reported in case of `cdef()` errors. Also, I just noticed, but pycparser always supported the preprocessor directive `# 42 "foo.h"` to mean "from the next line, we're in file foo.h starting from line 42", which it puts in the error messages.

3.5.8 v1.10

- Issue #295: use calloc() directly instead of PyObject_Malloc()+memset() to handle ffi.new() with a default allocator. Speeds up `ffi.new(large-array)` where most of the time you never touch most of

the array.

- Some OS/X build fixes ("only with Xcode but without CLT").
- Improve a couple of error messages: when getting mismatched versions of cffi and its backend; and when calling functions which cannot be called with libffi because an argument is a struct that is "too complicated" (and not a struct *pointer*, which always works).
- Add support for some unusual compilers (non-msvc, non-gcc, non-icc, non-clang)
- Implemented the remaining cases for `ffi.from_buffer`. Now all buffer/memoryview objects can be passed. The one remaining check is against passing unicode strings in Python 2. (They support the buffer interface, but that gives the raw bytes behind the UTF16/UCS4 storage, which is most of the times not what you expect. In Python 3 this has been fixed and the unicode strings don't support the memoryview interface any more.)
- The C type `_Bool` or `bool` now converts to a Python boolean when reading, instead of the content of the byte as an integer. The potential incompatibility here is what occurs if the byte contains a value different from 0 and 1. Previously, it would just return it; with this change, CFFI raises an exception in this case. But this case means "undefined behavior" in C; if you really have to interface with a library relying on this, don't use `bool` in the CFFI side. Also, it is still valid to use a byte string as initializer for a `bool[]`, but now it must only contain `\x00` or `\x01`. As an aside, `ffi.string()` no longer works on `bool[]` (but it never made much sense, as this function stops at the first zero).
- `ffi.buffer` is now the name of cffi's buffer type, and `ffi.buffer()` works like before but is the constructor of that type.
- `ffi.addressof(lib, "name")` now works also in in-line mode, not only in out-of-line mode. This is useful for taking the address of global variables.
- Issue #255: `cdata` objects of a primitive type (integers, floats, char) are now compared and ordered by value. For example, `<cdata 'int' 42>` compares equal to `42` and `<cdata 'char' b'A'>` compares equal to `b'A'`. Unlike C, `<cdata 'int' -1>` does not compare equal to `ffi.cast("unsigned int", -1)`: it compares smaller, because `-1 < 4294967295`.
- PyPy: `ffi.new()` and `ffi.new_allocator()()` did not record "memory pressure", causing the GC to run too infrequently if you call `ffi.new()` very often and/or with large arrays. Fixed in PyPy 5.7.
- Support in `ffi.cdef()` for numeric expressions with `+` or `-`. Assumes that there is no overflow; it should be fixed first before we add more general support for arbitrary arithmetic on constants.

3.5.9 v1.9

- Structs with variable-sized arrays as their last field: now we track the length of the array after `ffi.new()` is called, just like we always tracked the length of `ffi.new("int[]", 42)`. This lets us detect out-of-range accesses to array items. This also lets us display a better `repr()`, and have the total size returned by `ffi.sizeof()` and `ffi.buffer()`. Previously both functions would return a result based

on the size of the declared structure type, with an assumed empty array. (Thanks andrew for starting this refactoring.)

- Add support in `cdef()/set_source()` for unspecified-length arrays in typedefs: `typedef int foo_t[...];`. It was already supported for global variables or structure fields.
- I turned in v1.8 a warning from `cffi/model.py` into an error: `'enum xxx' has no values explicitly defined: refusing to guess which integer type it is meant to be (unsigned/signed, int/long)`. Now I'm turning it back to a warning again; it seems that guessing that the enum has size `int` is a 99%-safe bet. (But not 100%, so it stays as a warning.)
- Fix leaks in the code handling `FILE *` arguments. In CPython 3 there is a remaining issue that is hard to fix: if you pass a Python file object to a `FILE *` argument, then `os.dup()` is used and the new file descriptor is only closed when the GC reclaims the Python file object—and not at the earlier time when you call `close()`, which only closes the original file descriptor. If this is an issue, you should avoid this automatic convertion of Python file objects: instead, explicitly manipulate file descriptors and call `fdopen()` from C (...via cffi).

3.5.10 v1.8.3

- When passing a `void *` argument to a function with a different pointer type, or vice-versa, the cast occurs automatically, like in C. The same occurs for initialization with `ffi.new()` and a few other places. However, I thought that `char *` had the same property—but I was mistaken. In C you get the usual warning if you try to give a `char *` to a `char **` argument, for example. Sorry about the confusion. This has been fixed in CFFI by giving for now a warning, too. It will turn into an error in a future version.

3.5.11 v1.8.2

- Issue #283: fixed `ffi.new()` on structures/unions with nested anonymous structures/unions, when there is at least one union in the mix. When initialized with a list or a dict, it should now behave more closely like the `{ }` syntax does in GCC.

3.5.12 v1.8.1

- CPython 3.x: experimental: the generated C extension modules now use the "limited API", which means that, as a compiled .so/.dll, it should work directly on any version of CPython >= 3.2. The name produced by distutils is still version-specific. To get the version-independent name, you can rename it manually to `NAME.abi3.so`, or use the very recent setuptools 26.
- Added `ffi.compile(debug=...)`, similar to `python setup.py build --debug` but defaulting to True if we are running a debugging version of Python itself.

3.5.13 v1.8

- Removed the restriction that `ffi.from_buffer()` cannot be used on byte strings. Now you can get a `char *` out of a byte string, which is valid as long as the string object is kept alive. (But don't use it to *modify* the string object! If you need this, use `bytearray` or other official techniques.)
- PyPy 5.4 can now pass a byte string directly to a `char *` argument (in older versions, a copy would be made). This used to be a CPython-only optimization.

3.5.14 v1.7

- `ffi.gc(p, None)` removes the destructor on an object previously created by another call to `ffi.gc()`
- `bool(ffi.cast("primitive type", x))` now returns False if the value is zero (including `-0.0`), and True otherwise. Previously this would only return False for cdata objects of a pointer type when the pointer is NULL.
- bytearrays: `ffi.from_buffer(bytearray-object)` is now supported. (The reason it was not supported was that it was hard to do in PyPy, but it works since PyPy 5.3.) To call a C function with a `char *` argument from a buffer object—now including bytearrays—you write `lib.foo(ffi.from_buffer(x))`. Additionally, this is now supported: `p[0:length] = bytearray-object`. The problem with this was that a iterating over bytearrays gives *numbers* instead of *characters*. (Now it is implemented with just a memcpy, of course, not actually iterating over the characters.)
- C++: compiling the generated C code with C++ was supposed to work, but failed if you make use the `bool` type (because that is rendered as the C `_Bool` type, which doesn't exist in C++).
- `help(lib)` and `help(lib.myfunc)` now give useful information, as well as `dir(p)` where `p` is a struct or pointer-to-struct.

3.5.15 v1.6

- ffi.list_types()
- ffi.unpack()
- extern "Python+C"
- in API mode, `lib.foo.__doc__` contains the C signature now. On CPython you can say `help(lib.foo)`, but for some reason `help(lib)` (or `help(lib.foo)` on PyPy) is still useless; I haven't yet figured out the hacks needed to convince `pydoc` to show more. (You can use `dir(lib)` but it is not most helpful.)
- Yet another attempt at robustness of `ffi.def_extern()` against CPython's interpreter shutdown logic.

3.5.16 v1.5.2

- Fix 1.5.1 for Python 2.6.

3.5.17 v1.5.1

- A few installation-time tweaks (thanks Stefano!)
- Issue #245: Win32: `__stdcall` was never generated for `extern "Python"` functions
- Issue #246: trying to be more robust against CPython's fragile interpreter shutdown logic

3.5.18 v1.5.0

- Support for using CFFI for embedding.

3.5.19 v1.4.2

Nothing changed from v1.4.1.

3.5.20 v1.4.1

- Fix the compilation failure of cffi on CPython 3.5.0. (3.5.1 works; some detail changed that makes some underscore-starting macros disappear from view of extension modules, and I worked around it, thinking it changed in all 3.5 versions—but no: it was only in 3.5.1.)

3.5.21 v1.4.0

- A better way to do callbacks has been added (faster and more portable, and usually cleaner). It is a mechanism for the out-of-line API mode that replaces the dynamic creation of callback objects (i.e. C functions that invoke Python) with the static declaration in `cdef()` of which callbacks are needed. This is more C-like, in that you have to structure your code around the idea that you get a fixed number of function pointers, instead of creating them on-the-fly.
- `ffi.compile()` now takes an optional `verbose` argument. When `True`, distutils prints the calls to the compiler.
- `ffi.compile()` used to fail if given `sources` with a path that includes `".."`. Fixed.
- `ffi.init_once()` added. See docs.
- `dir(lib)` now works on libs returned by `ffi.dlopen()` too.
- Cleaned up and modernized the content of the `demo` subdirectory in the sources (thanks matti!).

- `ffi.new_handle()` is now guaranteed to return unique `void *` values, even if called twice on the same object. Previously, in that case, CPython would return two `cdata` objects with the same `void *` value. This change is useful to add and remove handles from a global dict (or set) without worrying about duplicates. It already used to work like that on PyPy. *This change can break code that used to work on CPython by relying on the object to be kept alive by other means than keeping the result of ffi.new_handle() alive.* (The corresponding warning in the docs of `ffi.new_handle()` has been here since v0.8!)

3.5.22 v1.3.1

- The optional typedefs (`bool`, `FILE` and all Windows types) were not always available from out-of-line FFI objects.
- Opaque enums are phased out from the cdefs: they now give a warning, instead of (possibly wrongly) being assumed equal to `unsigned int`. Please report if you get a reasonable use case for them.
- Some parsing details, notably `volatile` is passed along like `const` and `restrict`. Also, older versions of pycparser mis-parse some pointer-to-pointer types like `char * const *`: the "const" ends up at the wrong place. Added a workaround.

3.5.23 v1.3.0

- Added ffi.memmove().
- Pull request #64: out-of-line API mode: we can now declare floating-point types with `typedef float. .. foo_t;`. This only works if `foo_t` is a float or a double, not `long double`.
- Issue #217: fix possible unaligned pointer manipulation, which crashes on some architectures (64-bit, non-x86).
- Issues #64 and #126: when using `set_source()` or `verify()`, the `const` and `restrict` keywords are copied from the cdef to the generated C code; this fixes warnings by the C compiler. It also fixes corner cases like `typedef const int T; T a;` which would previously not consider `a` as a constant. (The cdata objects themselves are never `const`.)
- Win32: support for `__stdcall`. For callbacks and function pointers; regular C functions still don't need to have their calling convention declared.
- Windows: CPython 2.7 distutils doesn't work with Microsoft's official Visual Studio for Python, and I'm told this is not a bug. For ffi.compile(), we removed a workaround that was inside cffi but which had unwanted side-effects. Try saying `import setuptools` first, which patches distutils...

3.5.24 v1.2.1

Nothing changed from v1.2.0.

3.5.25 v1.2.0

- Out-of-line mode: `int a[][...];` can be used to declare a structure field or global variable which is, simultaneously, of total length unknown to the C compiler (the `a[]` part) and each element is itself an array of N integers, where the value of N *is* known to the C compiler (the `int` and `[...]` parts around it). Similarly, `int a[5][...];` is supported (but probably less useful: remember that in C it means `int (a[5])[...];`).
- PyPy: the `lib.some_function` objects were missing the attributes `__name__`, `__module__` and `__doc__` that are expected e.g. by some decorators-management functions from `functools`.
- Out-of-line API mode: you can now do `from _example.lib import x` to import the name `x` from `_example.lib`, even though the `lib` object is not a standard module object. (Also works in `from _example.lib import *`, but this is even more of a hack and will fail if `lib` happens to declare a name called `__all__`. Note that `*` excludes the global variables; only the functions and constants make sense to import like this.)
- `lib.__dict__` works again and gives you a copy of the dict—assuming that `lib` has got no symbol called precisely `__dict__`. (In general, it is safer to use `dir(lib)`.)
- Out-of-line API mode: global variables are now fetched on demand at every access. It fixes issue #212 (Windows DLL variables), and also allows variables that are defined as dynamic macros (like `errno`) or `__thread` -local variables. (This change might also tighten the C compiler's check on the variables' type.)
- Issue #209: dereferencing NULL pointers now raises RuntimeError instead of segfaulting. Meant as a debugging aid. The check is only for NULL: if you dereference random or dead pointers you might still get segfaults.
- Issue #152: callbacks: added an argument `ffi.callback(..., onerror=...)`. If the main callback function raises an exception and `onerror` is provided, then `onerror(exception, exc_value, traceback)` is called. This is similar to writing a `try: except:` in the main callback function, but in some cases (e.g. a signal) an exception can occur at the very start of the callback function—before it had time to enter the `try: except:` block.
- Issue #115: added `ffi.new_allocator()`, which officializes support for alternative allocators.

3.5.26 v1.1.2

- `ffi.gc()`: fixed a race condition in multithreaded programs introduced in 1.1.1

3.5.27 v1.1.1

- Out-of-line mode: `ffi.string()`, `ffi.buffer()` and `ffi.getwinerror()` didn't accept their arguments as keyword arguments, unlike their in-line mode equivalent. (It worked in PyPy.)

- Out-of-line ABI mode: documented a restriction of `ffi.dlopen()` when compared to the in-line mode.
- `ffi.gc()`: when called several times with equal pointers, it was accidentally registering only the last destructor, or even none at all depending on details. (It was correctly registering all of them only in PyPy, and only with the out-of-line FFIs.)

3.5.28 v1.1.0

- Out-of-line API mode: we can now declare integer types with `typedef int... foo_t;`. The exact size and signedness of `foo_t` is figured out by the compiler.
- Out-of-line API mode: we can now declare multidimensional arrays (as fields or as globals) with `int n[...][...]`. Before, only the outermost dimension would support the `...` syntax.
- Out-of-line ABI mode: we now support any constant declaration, instead of only integers whose value is given in the cdef. Such "new" constants, i.e. either non-integers or without a value given in the cdef, must correspond to actual symbols in the lib. At runtime they are looked up the first time we access them. This is useful if the library defines `extern const sometype somename;`.
- `ffi.addressof(lib, "func_name")` now returns a regular cdata object of type "pointer to function". You can use it on any function from a library in API mode (in ABI mode, all functions are already regular cdata objects). To support this, you need to recompile your cffi modules.
- Issue #198: in API mode, if you declare constants of a `struct` type, what you saw from lib.CONSTANT was corrupted.
- Issue #196: `ffi.set_source("package._ffi", None)` would incorrectly generate the Python source to `package._ffi.py` instead of `package/_ffi.py`. Also fixed: in some cases, if the C file was in `build/foo.c`, the .o file would be put in `build/build/foo.o`.

3.5.29 v1.0.3

- Same as 1.0.2, apart from doc and test fixes on some platforms.

3.5.30 v1.0.2

- Variadic C functions (ending in a "..." argument) were not supported in the out-of-line ABI mode. This was a bug—there was even a (non-working) example doing exactly that!

3.5.31 v1.0.1

- `ffi.set_source()` crashed if passed a `sources=[..]` argument. Fixed by chrippa on pull request #60.

- Issue #193: if we use a struct between the first cdef() where it is declared and another cdef() where its fields are defined, then this definition was ignored.
- Enums were buggy if you used too many "..." in their definition.

3.5.32 v1.0.0

- The main news item is out-of-line module generation:
  - for ABI level, with `ffi.dlopen()`
  - for API level, which used to be with `ffi.verify()`, now deprecated
- (this page will list what is new from all versions from 1.0.0 forward.)

CHAPTER 4

安装和状态

CPython 快速安装 (cffi 随 PyPy 一起发布):

- `pip install cffi`
- 或者通过 Python Package Index 获取源代码。

更多细节:

此代码是在 Linux 上开发的，但应该适用于任何 POSIX 平台以及 Windows 32 位和 64 位。(它偶尔会依赖于 libffi，所以这取决于 libffi 有没有 bug; 在一些更奇特的平台上，情况可能并非如此。)

CFFI 支持 CPython 2.6，2.7，3.x (用 3.2 到 3.4 进行测试); CFFI 并与 PyPy 一起发布 (CFFI 1.0 需要随 PyPy 2.6 一起发布).

CFFI 的核心速度优于 ctypes，如果使用 1.0 之后的功能，则意味着时间更短，或者更高，除非你不这样做。您通常需要围绕原始 CFFI 接口编写的包装 Python 代码会减慢 CPython 的速度，但并非不合理。在 PyPy 上，由于 JIT 编译器，这个包装器代码的影响很小。这使得 CFFI 成为 PyPy 与 C 库接口的推荐方式。

要求:

- CPython 2.6 或 2.7 或 3.x，或 PyPy (PyPy 2.0 是提供给 CFFI 最早的版本; 或 PyPy 2.6 提供 CFFI 1.0).
- 在某些情况下, 您需要能够编译 C 语言的扩展模块。在非 Windows 平台上, 一般方法安装 `python-dev` 包。请参阅适用于您的操作系统的相应文档。
- 在 CPython，非 Windows 平台上，你还需要安装 `libffi-dev` 才能编译 CFFI 本身。
- pycparser >= 2.06: https://github.com/eliben/pycparser (`pip install cffi` 自动跟踪).

- CFFI 自身需要 py.test 来运行测试。

下载和安装:

- https://pypi.python.org/pypi/cffi
- 校验"source" 包 version 1.12.3:
  - MD5: 35ad1f9e1003cac9404c1493eb10d7f5
  - SHA: ccc49cf31bc3f4248f45b9ec83685e4e8090a9fa
  - SHA256: 041c81822e9f84b1d9c401182e174996f0bae9991f33725d059b771744290774
- 或者从 Bitbucket page 页面获取最新版本: `hg clone https://bitbucket.org/cffi/cffi`
- `python setup.py install` 或 `python setup_base.py install` (在 Linux 或 Windows 上应该开箱即用; 请参阅*MacOS X* 或*Windows 64*.)
- 运行测试: `py.test c/ testing/` (如果你还没有安装 cffi，首先你需要 `python setup_base.py build_ext -f -i`)

演示:

- demo 目录包含许多使用 `cffi` 的小型和大型的演示。
- 下面的文档可能在细节上是简略的; 目前测试给出了最终的参考，特别是 testing/cffi1/test_verify1.py 和 testing/cffi0/backend_tests.py.

4.1 特定于平台的说明

`libffi` 的安装和使用非常混乱 - 以至于 CPython 包含自己的副本以避免依赖外部软件包。CFFI 对 Windows 也是如此，但对其他平台则没有 (哪一个应该具有他们自己的加工的 libffi)。感谢 `pkg-config` 当前 Linux 开箱即用。以下是（用户提供的）其他平台的一些说明。

4.1.1 MacOS X

Homebrew (感谢 David Griffin)

1) 安装 homebrew: http://brew.sh
2) 在终端中运行以下命令

```
brew install pkg-config libffi
PKG_CONFIG_PATH=/usr/local/opt/libffi/lib/pkgconfig pip install cffi
```

可选，**on OS/X 10.6** (感谢 Juraj Sukop)

要构建 libffi，您可以使用默认安装路径，但是然后，在 `setup.py` 中你需要改变:

```
include_dirs = []
```

为:

```
include_dirs = ['/usr/local/lib/libffi-3.0.11/include']
```

然后运行 `python setup.py build` 报出"fatal error: error writing to -: Broken pipe"，可以通过运行一下来修复:

```
ARCHFLAGS="-arch i386 -arch x86_64" python setup.py build
```

如 此处 所述。

4.1.2 Windows (常规 32-bit)

Win32 工作并至少在每个正式版本中进行测试。

与 Python 2.7 兼容的推荐 C 编译器是这个: http://www.microsoft.com/en-us/download/details.aspx?id=44266 Python 2.7 上存在 distutils 的已知问题，如中所述 https://bugs.python.org/issue23246，当你想运行 compile() 来使用这个特定编译器套件下载来构建的一个 dll 时，同样的问题也适用。`import setuptools` 可能有帮助，但是 YMMV

适用于 Python 3.4 及更高版本: https://www.visualstudio.com/en-us/downloads/visual-studio-2015-ctp-vs

4.1.3 Windows 64

Win64 接受了非常基本的测试，我们在 cffi 0.7 中应用了一些基本的修复。上面的评论也适用于 Windows 64 上的 Python 2.7。请报告任何其他问题。

请注意，这只是关于在 64 位操作系统上运行 64 位版本的 Python。如果您正在使用 32 位版本 (显然是常见的情况)，你是使用 Win32 那么就我们关心 Win32。

4.1.4 Linux and OS/X: UCS2 与 UCS4

这是关于使用像 `Symbol not found: _PyUnicodeUCS2_AsASCIIString` 这样的消息来获取关于 `_cffi_backend.so` 的 ImportError。只要混合使用 Python 的 "ucs2" 和 "ucs4" 版本，就会在 Python 2 中发生此错误。这意味着您现在正在运行使用 "ucs4" 编译的 Python，但是扩展模块 `_cffi_backend.so` 是由不同的 Python 编译的: 一个正在运行 "ucs2" 。(如果出现相反的问题，则会收到有关 `_PyUnicodeUCS4_AsASCIIString` 的错误。)

如果您正在使用 `pyenv` ，请参阅 https://github.com/yyuu/pyenv/issues/257.

更一般地说，应该始终有效的解决方案是下载 CFFI 的源代码 (而不是预先构建的二进制代码) 并确保使用与使用它相同版本的 Python 构建它。例如，使用 virtualenv:

- `virtualenv ~/venv`
- `cd ~/path/to/sources/of/cffi`
- `~/venv/bin/python setup.py build --force` # forcing a rebuild to make sure
- `~/venv/bin/python setup.py install`

这将使用 virtualenv 中的 Python 在这个 virtualenv 中编译和安装 CFFI。

4.1.5 NetBSD

您需要确保拥有最新版本的 libffi，它修复了一些错误。

CHAPTER 5

概览

Contents

第一部分介绍了一个使用 CFFI 从 Python 编译动态库 (DLL) 中调用 C 函数的简单工作示例。CFFI 非常灵活，涵盖了第二部分中介绍的其他几个用例。第三部分展示了如何将 Python 函数导出到嵌入在 C 或 C++

应用程序中的 Python 解释器。最后两节深入探讨了 CFFI 库。

确保你有 安装 cffi.

5.1 主要使用方式

使用 CFFI 的主要方式是作为一些已经编译的动态库的接口，这是由其他方法提供的。想象一下，您有一个系统安装的动态库名为 `piapprox.dll` (Windows) 或 `libpiapprox.so` (Linux 和其它) 或 `libpiapprox.dylib` (OS X)，导出函数 `float pi_approx(int n);` 在给定迭代次数的情况下计算 pi 的一些近似值。你想从 Python 调用这个函数。请注意，此方法与静态库 `piapprox.lib` (Windows) 或 `libpiapprox.a` 同样适用。

创建文件 `piapprox_build.py`:

```
from cffi import FFI
ffibuilder = FFI()

# cdef() expects a single string declaring the C types, functions and
# globals needed to use the shared object. It must be in valid C syntax.
ffibuilder.cdef("""
    float pi_approx(int n);
""")

# set_source() gives the name of the python extension module to
# produce, and some C source code as a string.  This C code needs
# to make the declarated functions, types and globals available,
# so it is often just the "#include".
ffibuilder.set_source("_pi_cffi",
"""
     #include "pi.h"   // the C header of the library
""",
     libraries=['piapprox'])   # library name, for the linker

if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

执行此脚本。如果一切正常，它应该产生 `_pi_cffi.c`，然后在其上调用编译器。生成的 `_pi_cffi.c` 包含 `set_source()` 中给出的字符串的副本，即这个例子中的 `#include "pi.h"`。之后，它包含上面 `cdef()` 中声明的所有函数，类型和全局变量的胶水代码。

在运行时，您可以像这样使用扩展模块:

```
from _pi_cffi import ffi, lib
print(lib.pi_approx(5000))
```

就这样！在本页的其余部分，我们将介绍一些更高级的示例和其他 CFFI 模式。特别是，*如果您没有已安装的 C 库来调用* ，这是一个完整的示例。

有关 FFI 类的 cdef() 和 set_source() 方法的更多信息，请参阅 准备和分发模块。

当您的示例有效时，手动运行构建脚本的常见替代方法是将其作为 setup.py 的一部分运行。以下是使用 Setuptools 分发的示例:

```
from setuptools import setup

setup(
    ...
    setup_requires=["cffi>=1.0.0"],
    cffi_modules=["piapprox_build:ffibuilder"], # "filename:global"
    install_requires=["cffi>=1.0.0"],
)
```

5.2 其他 CFFI 模式

CFFI 可以用于四种模式之一: "ABI" 与"API" 级别，这两种分别都有在线"in-line" 编译模式或离线"out-of-line" 编译模式。

ABI 模式以二进制级别访问库，而较快的 **API 模式**使用 C 编译器访问它们。我们解释了这个区别，*更多细节如下*。

在 **in-line 模式**下，每次导入 Python 代码时都会设置所有内容。在 **out-of-line 模式**下，你有一个单独的准备步骤 (可能还有 C 编译)，它产生一个模块，你的主程序可以导入该模块。

5.2.1 简单示例 (ABI 级别，in-line)

对于那些使用过 ctypes 的人来说可能看起来很熟悉。

```
>>> from cffi import FFI
>>> ffi = FFI()
>>> ffi.cdef("""
...     int printf(const char *format, ...);   // copy-pasted from the man page
... """)
>>> C = ffi.dlopen(None)                     # loads the entire C namespace
>>> arg = ffi.new("char[]", b"world")        # equivalent to C code: char arg[] = "world
↪";
```

(下页继续)

(续上页)

```
>>> C.printf(b"hi there, %s.\n", arg)        # call printf
hi there, world.
17                                           # this is the return value
>>>
```

请注意 char * 参数需要一个 bytes 对象。如果你有一个 str (或 Python 2 上 unicode) 你需要使用 somestring.encode(myencoding) 显示编码。

Windows 上的 Python 3: ffi.dlopen(None) 不起作用。这个问题很乱，而且无法解决。如果您尝试从系统上存在的特定 DLL 调用函数，则不会发生此问题: 然后你使用 ffi.dlopen("path.dll") 。

此示例不调用任何 *C* 编译器。它在所谓的 *ABI* 模式下工作，这意味着如果你调用某个函数或访问 *cdef()* 中稍微错误声明结构的某些字段，它将崩溃。

如果使用 C 编译器安装模块是一个选项，强烈建议使用 API 模式。(它也更快)

5.2.2 Struct/Array 示例 (minimal，in-line)

```
from cffi import FFI
ffi = FFI()
ffi.cdef("""
    typedef struct {
        unsigned char r, g, b;
    } pixel_t;
""")
image = ffi.new("pixel_t[]", 800*600)

f = open('data', 'rb')      # binary mode -- important
f.readinto(ffi.buffer(image))
f.close()

image[100].r = 255
image[100].g = 192
image[100].b = 128

f = open('data', 'wb')
f.write(ffi.buffer(image))
f.close()
```

这可以用作 struct 和 array 模块的更灵活的替换，并替换 ctypes 。你可以调用 ffi.new("pixel_t[600][800]") 并得到一个二维数组。

此示例不调用任何 *C* 编译器。

这个例子也承认与 out-of-line 等价。它类似于上面的第一个主要使用方式 示例，但将 None 作为第二个参数传递给 ffibuilder.set_source()。接着在主程序中写入 from _simple_example import ffi 然后从行 image = ffi.new("pixel_t[]", 800*600) 开始，与上面的 in-line 示例相同的内容。

5.2.3 API 模式，调用 C 标准库

```
# file "example_build.py"

# Note: we instantiate the same 'cffi.FFI' class as in the previous
# example, but call the result 'ffibuilder' now instead of 'ffi';
# this is to avoid confusion with the other 'ffi' object you get below

from cffi import FFI
ffibuilder = FFI()

ffibuilder.set_source("_example",
   r""" // passed to the real C compiler,
        // contains implementation of things declared in cdef()
        #include <sys/types.h>
        #include <pwd.h>

        // We can also define custom wrappers or other functions
        // here (this is an example only):
        static struct passwd *get_pw_for_root(void) {
            return getpwuid(0);
        }
    """,
    libraries=[])   # or a list of libraries to link with
    # (more arguments like setup.py's Extension class:
    # include_dirs=[..], extra_objects=[..], and so on)

ffibuilder.cdef("""
    // declarations that are shared between Python and C
    struct passwd {
        char *pw_name;
        ...;     // literally dot-dot-dot
    };
    struct passwd *getpwuid(int uid);     // defined in <pwd.h>
    struct passwd *get_pw_for_root(void); // defined in set_source()
""")
```

(下页继续)

(续上页)

```
if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

您需要运行一次 example_build.py 脚本以在文件 _example.c 中生成” 源代码”，并将其编译为常规 C 扩展模块。(CFFI 根据 set_source() 的第二个参数是否为 None 来选择要生成 Python 模块或 C 模块)

这个步骤需要一个 *C* 编译器。它产生一个名为例如 *_example.so* 或 *_example.pyd* 的文件。如果需要，它可以像任何其他扩展模块一样以预编译形式分发。

然后，在您的主程序中，您使用:

```
from _example import ffi, lib

p = lib.getpwuid(0)
assert ffi.string(p.pw_name) == b'root'
p = lib.get_pw_for_root()
assert ffi.string(p.pw_name) == b'root'
```

请注意 struct passwd 与 C 设计确切无关 (它是”API 级别”，而不是”ABI 级别”)。它需要一个 C 编译器才能运行 example_build.py，但它比尝试完全正确地获取 struct passwd 字段的细节要便携得多。同样，在 cdef() 中我们将 getpwuid() 声明为采用 int 参数; 在某些平台上，这可能稍微不正确，但并不重要。

另请注意，在运行时，API 模式比 ABI 模式更快。

要使用 Setuptools 进行分发，将其集成到 setup.py :

```
from setuptools import setup

setup(
    ...
    setup_requires=["cffi>=1.0.0"],
    cffi_modules=["example_build.py:ffibuilder"],
    install_requires=["cffi>=1.0.0"],
)
```

5.2.4 API 模式，调用 C 语言源码而不是编译库

如果要调用某些未预编译的库，但是你有 C 语言源代码，那么最简单的解决方案是创建一个从这个库中的 C 语言源代码编译的扩展模块，和额外的 CFFI 包装器 (用于封装 C 语言源代码的库并构建扩展模块)。例如，从 pi.c 和 pi.h 文件开始:

```
/* filename: pi.c*/
# include <stdlib.h>
# include <math.h>

/* Returns a very crude approximation of Pi
   given a int: a number of iteration */
float pi_approx(int n){

  double i,x,y,sum=0;

  for(i=0;i<n;i++){

    x=rand();
    y=rand();

    if (sqrt(x*x+y*y) < sqrt((double)RAND_MAX*RAND_MAX))
      sum++; }

  return 4*(float)sum/(float)n; }
```

```
/* filename: pi.h*/
float pi_approx(int n);
```

创建一个脚本名为 pi_extension_build.py，构建 C 语言扩展:

```
from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef("float pi_approx(int n);")

ffibuilder.set_source("_pi",  # name of the output C extension
"""
    #include "pi.h"',
""",
    sources=['pi.c'],   # includes pi.c as additional sources
    libraries=['m'])    # on Unix, link with the math library

if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

构建扩展:

```
python pi_extension_build.py
```

注意到，在工作目录下，生成的输出文件: `_pi.c`，`_pi.o` 和编译的 C 语言扩展 (例如，在 Linux 上叫 `_pi.so`)。它可以被 PYthon 调用:

```
from _pi.lib import pi_approx

approx = pi_approx(10)
assert str(pi_approximation).startswith("3.")

approx = pi_approx(10000)
assert str(approx).startswith("3.1")
```

5.2.5 单纯的性能 (API level，out-of-line)

以上部分 的变型，其目标不是调用现有的 C 库，而是编译并调用直接在构建脚本中编写的一些 C 语言函数:

```
# file "example_build.py"

from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef("int foo(int *, int *, int);")

ffibuilder.set_source("_example",
r"""
    static int foo(int *buffer_in, int *buffer_out, int x)
    {
        /* some algorithm that is seriously faster in C than in Python */
    }
""")

if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

```
# file "example.py"

from _example import ffi, lib

buffer_in = ffi.new("int[]", 1000)
```

(下页继续)

(续上页)

```
# initialize buffer_in here...

# easier to do all buffer allocations in Python and pass them to C,
# even for output-only arguments
buffer_out = ffi.new("int[]", 1000)

result = lib.foo(buffer_in, buffer_out, 1000)
```

您需要一个 C 编译器来运行 *example_build.py* 一次。它产生一个文件名为 *_example.so* 或 *_example.pyd*。如果可以，它可以像任何其他扩展模块一样以预编译形式分发。

5.2.6 Out-of-line，ABI 模式

out-of-line ABI 模式是常规 (API) out-of-line 模式和 in-line ABI 模式的混合。它允许您使用 ABI 模式，具有其优点 (不需要 C 编译器) 和问题 (更容易崩溃).

这种混合模式可以大大减少导入时间，因为解析较大 C 头文件很慢。它还允许您在构建期间进行更详细的检查，而不必担心性能 (例如根据系统上检测到的库版本，使用小块声明多次调用 `cdef()`)。

```
# file "simple_example_build.py"

from cffi import FFI

ffibuilder = FFI()
# Note that the actual source is None
ffibuilder.set_source("_simple_example", None)
ffibuilder.cdef("""
    int printf(const char *format, ...);
""")

if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

运行会产生 `_simple_example.py`。您的主程序仅导入此生成的模块，而不再是 `simple_example_build.py`:

```
from _simple_example import ffi

lib = ffi.dlopen(None)      # Unix: open the standard C library
#import ctypes.util         # or, try this on Windows:
#lib = ffi.dlopen(ctypes.util.find_library("c"))
```

(下页继续)

(续上页)

```
lib.printf(b"hi there, number %d\n", ffi.cast("int", 2))
```

注意这个 `ffi.dlopen()`，不像 in-line 模式，不会调用任何额外的魔法来定位库：它必须是路径名 (带或不带目录)，根据 C 的 `dlopen()` 或 `LoadLibrary()` 函数的要求。这意味着 `ffi.dlopen("libfoo.so")` 没问题，但 `ffi.dlopen("foo")` 却不行。在后一种情况下，你可以用 `ffi.dlopen(ctypes.util.find_library("foo"))` 替换它。此外，None 仅在 Unix 上被识别以打开标准 C 库。

出于分发目的，请记住生成了一个新的 `_simple_example.py` 文件。您可以在项目的源文件中静态包含它，或者，使用 Setuptools，您可以在 `setup.py` 中这样编写:

```
from setuptools import setup

setup(
    ...
    setup_requires=["cffi>=1.0.0"],
    cffi_modules=["simple_example_build.py:ffibuilder"],
    install_requires=["cffi>=1.0.0"],
)
```

总之，当您希望声明许多 C 语言结构但不需要与共享对象快速交互时，此模式很有用。例如，它对于解析二进制文件很有用。

5.2.7 In-line，API 模式

"API level + in-line" 模式存在错误，但很久就会弃用。它曾经用 `lib = ffi.verify("C header")` 。具有 `set_source("modname", "C header")` 的 out-of-line 变型是首选的，并且当项目规模增大时避免了许多问题。

5.3 嵌入

版本 1.5 中的新功能。

CFFI 可用于 嵌入: 创建一个标准的动态链接库 (Windows 下 `.dll` ，其他地方 `.so`) 可以在 C 应用程序中使用。

```
import cffi
ffibuilder = cffi.FFI()

ffibuilder.embedding_api("""
    int do_stuff(int, int);
""")
```

(下页继续)

(续上页)

```
ffibuilder.set_source("my_plugin", "")

ffibuilder.embedding_init_code("""
    from my_plugin import ffi

    @ffi.def_extern()
    def do_stuff(x, y):
        print("adding %d and %d" % (x, y))
        return x + y
""")

ffibuilder.compile(target="plugin-1.5.*", verbose=True)
```

这个简单的示例将 `plugin-1.5.dll` 或 `plugin-1.5.so` 创建为具有单个导出函数 `do_stuff()` 的 DLL。您使用要在内部使用的解释器执行上面的脚本一次; 它可以是 CPython 2.x 或 3.x 或 PyPy。然后可以从应用程序" 照常" 使用此 DLL; 应用程序不需要知道它正在与使用 Python 和 CFFI 创建的库进行通信。在运行时，当应用程序调用 `int do_stuff(int, int)` 时，Python 解释器会自动初始化并且被 `def do_stuff(x, y):` 调用。请参阅有关嵌入的文档中的详细信息。

5.4 究竟发生了什么？

CFFI 接口在与 C 语言相同的级别上运行————您使用与 C 语言中相同的语法声明类型和函数定义它们。

这意味着大多数文档或示例都可以直接从手册页中复制。

声明可以包含 **类型，函数，常量**和 **和全局变量**。传递给 `cdef()` 的内容不得包含其他内容; 特别是，`#ifdef` 或 `#include` 指令是不支持的。上面例子中的 cdef 就是这样————他们声明 "在 C 级别中有一个具有此给定签名的函数"，或者 "存在具有此形状的结构类型"。

在 ABI 示例中，`dlopen()` 手动调用加载库。在二进制级别，程序被分成多个命名空间 - 一个全局命名空间 (在某些平台上)，每个库加一个命名空间。因此 `dlopen()` 返回一个 `<FFILibrary>` 对象，并且该对象具有来自该库的所有函数，常量和变量符号作为属性，并且已在 `cdef()` 中声明。如果要加载多个相互依赖的库，则只能调用一次 `cdef()` ，但可以多次调用 `dlopen()` 。

相反，API 模式更像 C 语言程序: C 链接器 (静态或动态) 负责查找使用的任何符号。你将库名 `libraries` 作为 `set_source()` 的关键字参数，但永远不需要说明哪个符号来自哪个库。`set_source()` 的其他常见参数包括 `library_dirs` 和 `include_dirs`; 所有这些参数都传递给标准的 distutils/setuptools.

`ffi.new()` 行分配 C 对象。除非使用可选的第二个参数，否则它们最初用零填充。如果指定，这个参数给出了一个" 初始值"，就像你可以用 C 代码初始化全局变量一样。

实际的 `lib.*()` 函数调用应该是显而易见的: 就像 C 一样。

5.5 ABI 与 API

在二进制级别 ("ABI") 访问 C 库充满了问题，特别是在非 Windows 平台上。

ABI 级别最直接的缺点是调用函数需要通过非常通用的 *libffi* 库，这很慢 (并且在非标准平台上并不总是完美测试)。API 模式改为编译直接调用目标函数的 CPython C 包装器。它可以更快 (并且比 libffi 工作得更好)。

更喜欢 API 模式的根本原因是 *C 库通常用于与 C 编译器一起使用*。你不应该做猜测结构中字段的位置。上面的" 真实示例" 显示了 CFFI 如何使用 C 编译器: 此示例使用 `set_source(..., "C source...")` 而不是 `dlopen()`。使用这种方法时，我们的优点是我们可以在 `cdef()` 中的不同位置使用字面"`...`"，缺少的信息将在 C 编译器的帮助下完成。CFFI 会将其转换为单个 C 源文件，其中包含未经修改的"C 源代码" 部分，后跟一些" 魔术"C 语言代码和从 `cdef()` 派生的声明。编译此 C 语言文件时，生成的 C 扩展模块将包含我们需要的所有信息- 或者 C 编译器将发出警告或错误，例如. 如果我们错误地声明了某些函数的签名。

请注意 `set_source()` 中的"C source" 部分可以包含任意 C 代码。您可以使用它来声明一些用 C 编写的辅助函数。要将这些帮助程序导出到 Python，请将它们的签名放在 `cdef()` 中。(您可以在"C source" 部分中使用 `static` C 关键字，如 `static int myhelper(int x) { return x * 42; }` 因为这些辅助只是在同一个 C 文件中生成的" 魔术"C 代码中引用。)

这可以用于例如将"crazy" 宏包装到更标准的 C 函数中。额外的 C 语言层在其他方面也很有用，喜欢调用期望一些复杂的参数结构的函数，你喜欢用 C 而不是 Python 构建。(另一方面，如果您只需要调用" 类似函数" 的宏，那么您可以直接在 `cdef()` 中声明它们，就好像它们是函数一样。)

生成的 C 语言代码应该在运行它的平台 (或 Python 版本) 上独立相同，因此在简单的情况下，您可以直接分发预生成的 C 语言代码并将其视为常规 C 扩展模块 (这取决于 CPython 上的 `_cffi_backend` 模块。) 上面示例 中的特殊 Setuptools 行是针对更复杂的情况，我们需要重新生成 C 源代码————例如: 因为重新生成此文件的 Python 脚本本身会查看系统以了解它应包含的内容。

CHAPTER 6

使用 ffi/lib 对象

Contents

– *FFI 接口*

把这个页面放在枕边作为参考

6.1 使用指针，结构体和数组

C 语言代码的整数和浮点值映射到 Python 的常规 `int`，`long` 和 `float`。而且，C 语言类型 `char` 对应于 Python 中的单字符字符串。(如果要将其映射到小整数，请使用 `signed char` 或 `unsigned char`。)

同样，C 语言类型 `wchar_t` 对应于单字符 unicode 字符串。请注意，在某些情况下 (一个 narrow 的 Python 构建，具有底层的 4 字节 wchar_t 类型)，单个 wchar_t 字符可能对应于一对代理 (码元，码位)，它们表示为长度为 2 的 unicode 字符串。如果需要将这样的 2-chars unicode 字符串转换为整数，则 `ord(x)` 不起作用; 使用 `int(ffi.cast('wchar_t', x))` 替换。

版本 1.11 中的新功能: 除了 `wchar_t` 之外，C 语言类型 `char16_t` 和 `char32_t` 的工作方式相同，但具有已知的固定大小。在以前的版本中，这可以使用 `uint16_t` 和 `int32_t` 实现，但不能自动转换为 Python unicodes。

指针，结构和数组更复杂: 他们没有明显的 Python 等价物。因此，它们对应类型 `cdata` 的对象，例如，它们被打印为 `<cdata 'struct foo_s *' 0xa3290d8>`。

`ffi.new(ctype, [initializer])`: 此函数构建并返回给定 `ctype` 的新 cdata 对象。ctype 通常是一些描述 C 类型的常量字符串。它必须是指针或数组类型。如果是指针，例如 `"int *"` 或 `struct foo *`，然后它为一个 `int` 或 `struct foo` 分配内存。如果是数组，例如 `int[10]`，然后它为 10 个 `int` 分配内存。在这两种情况下，返回的 cdata 都是 `ctype` 类型。

内存最初用零填充。如下所述，也可以给出初始值。

例:

```
>>> ffi.new("int *")
<cdata 'int *' owning 4 bytes>
>>> ffi.new("int[10]")
<cdata 'int[10]' owning 40 bytes>

>>> ffi.new("char *")          # allocates only one char---not a C string!
<cdata 'char *' owning 1 bytes>
>>> ffi.new("char[]", "foobar")  # this allocates a C string, ending in \0
<cdata 'char[]' owning 7 bytes>
```

与 C 不同，返回的指针对象对分配的内存具有 *所有权*: 当这个确切的对象被垃圾收集时，内存被释放。如果在 C 语言级别，你在其他地方存储了一个指向内存的指针，接着确保你还可以根据需要保持对象存活。(如果您立即将返回的指针强制转换为其他类型的指针，这也适用: 只有原始对象拥有所有权，所以你必须保持它的存活。一旦忽略它，那么转换的指针将指向垃圾! 换句话说，所有权规则附加到包装器 cdata 对象: 它们不是也不能，附加到底层的原始内存中。) 例:

```
global_weakkeydict = weakref.WeakKeyDictionary()

def make_foo():
    s1   = ffi.new("struct foo *")
    fld1 = ffi.new("struct bar *")
    fld2 = ffi.new("struct bar *")
    s1.thefield1 = fld1
    s1.thefield2 = fld2
    # here the 'fld1' and 'fld2' object must not go away,
    # otherwise 's1.thefield1/2' will point to garbage!
    global_weakkeydict[s1] = (fld1, fld2)
    # now 's1' keeps alive 'fld1' and 'fld2'.  When 's1' goes
    # away, then the weak dictionary entry will be removed.
    return s1
```

通常你不需要弱字典：例如，要使用包含指向 `char *` 指针的指针的 `char * *` 参数调用函数，这样就足够了：

```
p = ffi.new("char[]", "hello, world")    # p is a 'char *'
q = ffi.new("char **", p)                # q is a 'char **'
lib.myfunction(q)
# p is alive at least until here, so that's fine
```

然而，这总是错的 (使用释放的内存)：

```
p = ffi.new("char **", ffi.new("char[]", "hello, world"))
# WRONG!  as soon as p is built, the inner ffi.new() gets freed!
```

出于同样的原因，这也是错误的：

```
p = ffi.new("struct my_stuff")
p.foo = ffi.new("char[]", "hello, world")
# WRONG!  as soon as p.foo is set, the ffi.new() gets freed!
```

cdata 对象主要支持与 C 中相同的操作：你可以从指针，数组和结构中读取或写入。取消引用一个指针通常在 C 语言中语法为 `*p`，这不是有效的 Python，所以你必须使用替代语法 `p[0]` (这也是有效的 C)。另外，C 语言中的 `p.x` 和 `p->x` 语法都在 Python 中成为 `p.x`。

我们有 `ffi.NULL` 同 C 语言 `NULL` 在相同位置使用。与后者类似，它实际上被定义为 `ffi.cast("void *", 0)`。例如，读取一个 NULL 指针返回一个 `<cdata 'type *' NULL>`，你可以检查，例如通过与 `ffi.NULL` 比较。

C 中的 `&` 运算符没有一般等价物 (因为它不适合模型，而且这里似乎不需要它)。有 ffi.addressof()，但仅限于某些情况。例如，你不能在 Python 中获取数字的” 地址”; 同样，你不能获取 CFFI 指针的地址。如果你有

这种 C 语言代码:

```
int x, y;
fetch_size(&x, &y);

opaque_t *handle;      // some opaque pointer
init_stuff(&handle);   // initializes the variable 'handle'
more_stuff(handle);    // pass the handle around to more functions
```

那么你需要像这样重写它，用逻辑上指向变量的指针替换 C 中的变量:

```
px = ffi.new("int *")
py = ffi.new("int *")              arr = ffi.new("int[2]")
lib.fetch_size(px, py)    -OR-     lib.fetch_size(arr, arr + 1)
x = px[0]                          x = arr[0]
y = py[0]                          y = arr[1]

p_handle = ffi.new("opaque_t **")
lib.init_stuff(p_handle)   # pass the pointer to the 'handle' pointer
handle = p_handle[0]       # now we can read 'handle' out of 'p_handle'
lib.more_stuff(handle)
```

在 C 中返回指针或数组或结构类型的任何操作都会为您提供一个新的 cdata 对象。与" 原始" 的方式不同，这些新的 cdata 对象没有所有权: 它们仅仅是对现有内存的引用。

作为上述规则的例外，取消引用一个拥有 *struct* 或 *union* 对象的指针会返回一个" 共同拥有" 相同内存的 cdata struct 或 union 对象。因此，在这种情况下，有两个对象可以保持相同的内存存活。这样做是为了你真正想拥有一个 struct 对象但没有任何合适的位置来保存原始指针对象 (由 `ffi.new()` 返回)。

例:

```
# void somefunction(int *);

x = ffi.new("int *")       # allocate one int, and return a pointer to it
x[0] = 42                  # fill it
lib.somefunction(x)        # call the C function
print x[0]                 # read the possibly-changed value
```

`ffi.cast("type", value)` 是 C 转换 (casts) 提供的等价物。他们应该像在 C 中一样工作。另外，这是获取整数或浮点类型的 cdata 对象的唯一方法:

```
>>> x = ffi.cast("int", 42)
>>> x
<cdata 'int' 42>
```

(下页继续)

(续上页)

```
>>> int(x)
42
```

将指针强制转换为 int，将它转换为 intptr_t 或 uintptr_t，由 C 定义为足够大的整数类型 (例如 32 位):

```
>>> int(ffi.cast("intptr_t", pointer_cdata))    # signed
-1340782304
>>> int(ffi.cast("uintptr_t", pointer_cdata))   # unsigned
2954184992L
```

ffi.new() 的可选第二个参数给出的初始值可以是您用作 C 代码的初始值的任何东西, 使用列表或元组而不是使用 C 语法 { .., .., .. }。例:

```
typedef struct { int x, y; } foo_t;

foo_t v = { 1, 2 };            // C syntax
v = ffi.new("foo_t *", [1, 2]) # CFFI equivalent

foo_t v = { .y=1, .x=2 };                // C99 syntax
v = ffi.new("foo_t *", {'y': 1, 'x': 2}) # CFFI equivalent
```

与 C 一样，字符数组也可以从字符串初始化，在这种情况下，隐式附加终止空字符:

```
>>> x = ffi.new("char[]", "hello")
>>> x
<cdata 'char[]' owning 6 bytes>
>>> len(x)        # the actual size of the array
6
>>> x[5]          # the last item in the array
'\x00'
>>> x[0] = 'H'    # change the first item
>>> ffi.string(x) # interpret 'x' as a regular null-terminated string
'Hello'
```

同样，可以从 unicode 字符串初始化 wchar_t 或 char16_t 或 char32_t 的数组，并在 cdata 对象上调用 ffi.string() 返回存储在源数组中的当前 unicode 字符串 (必要时添加代理 (码元，码位))。有关更多详细信息，请参阅 Unicode 字符类型 部分。

请注意，与 Python 列表或元组不同，但与 C 类似，你不能使用负数在最后的 C 数组中索引。

更一般地说，C 语言数组类型的长度可以在 C 类型中未指定，只要它们的长度可以从初始化值得到，就像在 C 中:

```
int array[] = { 1, 2, 3, 4 };           // C syntax
array = ffi.new("int[]", [1, 2, 3, 4])  # CFFI equivalent
```

作为扩展，初始化值也可以只是一个数字，而且给出了长度 (如果你只想要零初始化):

```
int array[1000];                  // C syntax
array = ffi.new("int[1000]")      # CFFI 1st equivalent
array = ffi.new("int[]", 1000)    # CFFI 2nd equivalent
```

如果长度实际上不是常数，这将非常有用，以避免像 `ffi.new("int[%d]" % x)` 这样的事情。实际上，不建议这样做: `ffi` 通常缓存字符串 `"int[]"` 不需要一直重新解析它。

C99 支持可变大小结构，只要初始化值说明数组长度:

```
# typedef struct { int x; int y[]; } foo_t;

p = ffi.new("foo_t *", [5, [6, 7, 8]]) # length 3
p = ffi.new("foo_t *", [5, 3])         # length 3 with 0 in the array
p = ffi.new("foo_t *", {'y': 3})       # length 3 with 0 everywhere
```

最后，请注意，任何用作初始化值的 Python 对象也可以在没有 `ffi.new()` 的情况下直接用于数组项或结构字段的赋值。实际上，`p = ffi.new("T*", initializer)` 是等价于 `p = ffi.new("T*"); p[0] = initializer`。例:

```
# if 'p' is a <cdata 'int[5][5]'>
p[2] = [10, 20]             # writes to p[2][0] and p[2][1]

# if 'p' is a <cdata 'foo_t *'>, and foo_t has fields x, y and z
p[0] = {'x': 10, 'z': 20}   # writes to p.x and p.z; p.y unmodified

# if, on the other hand, foo_t has a field 'char a[5]':
p.a = "abc"                 # writes 'a', 'b', 'c' and '\0'; p.a[4] unmodified
```

在函数调用中，传递参数时，这些规则也可以使用; 请参见函数调用.

6.2 Python 3 支持

支持 Python 3，但要注意的要点是 C 语言类型 `char` 对应 Python 类型 `bytes`，而不是 `str`。在将所有 Python 字符串传递给 CFFI 或从 CFFI 接收它们时，您有责任将所有 Python 字符串编码/解码为字节。

这仅涉及 `char` 类型和派生类型; 在 Python 2 中接受字符串的 API 的其他部分继续接受 Python 3 中的字符串。

6.3 调用类似 main 的一个例子

想象一下，我们有某些类似这个:

```
from cffi import FFI
ffi = FFI()
ffi.cdef("""
   int main_like(int argv, char *argv[]);
""")
lib = ffi.dlopen("some_library.so")
```

现在，一切都很简单，除了，我们如何在这里创建 `char**` 参数? 第一个想法:

```
lib.main_like(2, ["arg0", "arg1"])
```

不起作用，因为初始化值接收两个 Python `str` 对象，期望它是 `<cdata 'char *'>` 对象。您需要显式使用 `ffi.new()` 来创建这些对象:

```
lib.main_like(2, [ffi.new("char[]", "arg0"),
                  ffi.new("char[]", "arg1")])
```

请注意，两个 `<cdata 'char[]'>` 对象在调用期间保持活动状态: 它们仅在列表本身被释放时释放，并且仅在调用返回时释放列表。

如果你想要构建一个你想要重复使用的"argv" 变量，那么需要更加小心:

```
# DOES NOT WORK!
argv = ffi.new("char *[]", [ffi.new("char[]", "arg0"),
                            ffi.new("char[]", "arg1")])
```

在上面的示例中，只要构建"argv"，就会释放内部"arg0" 字符串。您必须确保保留对内部"char[]" 对象的引用，直接或通过保持列表活动，像这样:

```
argv_keepalive = [ffi.new("char[]", "arg0"),
                  ffi.new("char[]", "arg1")]
argv = ffi.new("char *[]", argv_keepalive)
```

6.4 函数调用

调用 C 函数时，传递参数主要遵循与分配给结构字段相同的规则，返回值遵循与读取结构字段相同的规则。例如:

```
# int foo(short a, int b);

n = lib.foo(2, 3)      # returns a normal integer
lib.foo(40000, 3)      # raises OverflowError
```

您可以将 char * 参数传递给普通的 Python 字符串 (但是不要将普通的 Python 字符串传递给带有 char * 参数的函数并且改变它!):

```
# size_t strlen(const char *);

assert lib.strlen("hello") == 5
```

您还可以将 unicode 字符串作为 wchar_t * 或 char16_t * 或 char32_t * 参数传递。请注意 C 语言在使用 type * 或 type[] 的参数声明之间没有区别。例如，int * 完全等价于 int[] (甚至 int[5]; 5 被忽略了)。对于 CFFI，这意味着您始终可以传递参数以转换为 int * 或 int[]。例如:

```
# void do_something_with_array(int *array);

lib.do_something_with_array([1, 2, 3, 4, 5])    # works for int[]
```

请参见 参考: 转换 类似于传递 struct foo_s * 参数的方法————但一般来说，在这种情况下传递 ffi.new('struct foo_s *', initializer) 更清晰。

CFFI 支持传递和返回结构和联合到函数和回调。例:

```
# struct foo_s { int a, b; };
# struct foo_s function_returning_a_struct(void);

myfoo = lib.function_returning_a_struct()
# `myfoo`: <cdata 'struct foo_s' owning 8 bytes>
```

出于性能，通过编写 lib.some_function 获得的非可变 API 级别级函数不是 <cdata> 对象，而是不同类型的对象 (在 CPython 上，<built-in function>)。这意味着您不能将它们直接传递给其他 C 语言函数期望的函数指针参数。只有 ffi.typeof() 才能使用它们。要获取包含常规函数指针的 cdata，请使用 ffi.addressof(lib, "name")。

支持的参数和返回类型有一些（模糊的）限制。这些限制来自 libffi，仅适用于调用 <cdata> 函数指针; 换句话说，如果您使用 API 模式，它们不适用于不可变参数 cdef() 声明的函数。限制是您不能直接作为参数传递或返回类型:

- 联合 (union) (但是联合 指针是不受限制的);
- 一个使用位字段的结构体 (struct) (但这样的结构体 指针是不受限制的);
- 在 cdef() 中用"..." 声明的结构体。.

在 API 模式下，你可以解决这些限制: 例如，如果你需要从 Python 调用这样的函数指针，您可以改为编写一个自定义 C 语言函数，该函数接受函数指针和真实参数，并从 C 语言执行调用。然后在 cdef() 中声明自定义 C 语言函数并从 Python 中使用它。

6.5 可变函数调用

C 语言中的可变参数函数 (以"..." 作为最后一个参数结束) 可以正常声明和调用，但可变部分中传递的所有参数必须是 cdata 对象。这是因为无法猜测，如果你写了这个:

```
lib.printf("hello, %d\n", 42)    # doesn't work!
```

你真的认为 42 作为 C 语言 int 传递，并不是 long 或 long long。float 与 double 会出现同样的问题。所以你必须强制你想要的 C 语言类型是 cdata 对象, 必要时使用 ffi.cast():

```
lib.printf("hello, %d\n", ffi.cast("int", 42))
lib.printf("hello, %ld\n", ffi.cast("long", 42))
lib.printf("hello, %f\n", ffi.cast("double", 42))
```

但理所当然:

```
lib.printf("hello, %s\n", ffi.new("char[]", "world"))
```

请注意，如果您使用的是 dlopen()，cdef() 中的函数声明必须与 C 中的原始声明完全匹配，像往常一样————尤其如此，如果此函数在 C 语言中是可变参数的，那么它的 cdef() 声明也必须是可变的。您不能使用固定参数在 cdef() 中声明它，即使你打算只用这些参数类型调用它。原因是某些体系结构具有不同的调用约定，具体取决于函数签名是否固定。(在 x86-64 上，如果某些参数是 double 类型的，有时可以在 PyPy 的 JIT 生成的代码中看到差异。)

注意函数签名 int foo(); 由 CFFI 解释为等同于 int foo(void);。这与 C 标准不同，其中 int foo(); 真的像 int foo(...); 并且可以使用任何参数调用。(这个特征是 C89 之前的遗留: 在不依赖于特定于编译器的扩展的情况下，在 foo() 的主体中根本无法访问参数。现在几乎所有代码都使用 int foo(); 的真实意思是 int foo(void);。)

6.6 内存压力 (PyPy)

本段仅适用于 PyPy，因为它的垃圾收集器 (GC) 与 CPython 不同。在 C 语言代码中，通常有一对函数，一个执行内存分配或获取其他资源，而另一个让它们再次释放。根据您构建 Python 代码的方式，只有在 GC 决定可以释放特定 (Python) 对象时才会调用释放函数。这种情况尤其明显:

- 如果你使用 __del__() 方法来调用释放函数。
- 如果你使用 ffi.gc() 而不使用 ffi.release()。

- 如果在确定的时间调用释放函数，则不会发生这种情况，例如在常规 `try: finally:` 语句块。然而它确实发生 *在生成器内*————如果生成器没有明确释放但忘记了 `yield` 这一点，则封闭在 `finally` 块中的代码仅在下一个 GC 处调用。

在这些情况下，您可能必须使用内置函数 `__pypy__.add_memory_pressure(n)`。它的参数 `n` 是对要添加的内存压力的估量。例如，如果这对 C 语言函数我们谈论的是 `malloc(n)` 和 `free()` 或类似的函数，则在 `malloc(n)` 之后调用 `__pypy__.add_memory_pressure(n)`。这样做不总是问题的完整答案，但它使下一个 GC 发生得更早，这通常就足够了。

如果内存分配是间接的，则同样适用，例如 C 语言函数分配一些内部数据结构。在这种情况下，使用参数 `n` 调用 `__pypy__.add_memory_pressure(n)` 这是一个粗略估量。知道确切的大小并不重要，并且在调用释放函数后不必再次手动降低内存压力。如果您正在为分配/释放函数编写包装器，你应该在前者中调用 `__pypy__.add_memory_pressure()`，即使用户可以在 `finally:` 块的已知点调用后者。.

如果这个解决方案还不够，或者，如果获取的资源不是内存，而是其他更有限的内容 (如文件描述符)，那么没有比重构代码更好的方法来确保在已知点调用释放函数而不是由 GC 间接调用。

请注意，在 PyPy <= 5.6 中，上面的讨论也适用于 `ffi.new()`. 在更新版本的 PyPy 中，both `ffi.new()` 和 `ffi.new_allocator()()` 都会自动解释它们创建的内存压力。(如果您需要支持较旧和较新的 PyPy，无论如何，尝试调用 `__pypy__.add_memory_pressure()`; 最好扩大估量而不是考虑内存压力。)

6.7 外部"Python" (新式回调)

当语言 C 代码需要一个指向函数的指针，该函数调用您选择的 Python 函数时，以下是在 out-of-line API 模式下执行此操作的方法。关于回调 的下一节描述了 ABI 模式解决方案。

这是 *1.4 版本中的新功能*。如果向后兼容性存在问题，请使用旧式回调。(原始回调调用较慢，并且与 libffi 的回调具有相同的问题; 值得注意的是，请看警告 。本节中描述的新样式根本不使用 libffi 的回调。)

在构建器脚本中，在 cdef 中声明一个以 `extern "Python"` 为前缀的函数:

```
ffibuilder.cdef("""
    extern "Python" int my_callback(int, int);

    void library_function(int(*callback)(int, int));
""")
ffibuilder.set_source("_my_example", r"""
    #include <some_library.h>
""")
```

然后，函数 `my_callback()` 在应用程序代码中的 Python 中实现:

```
from _my_example import ffi, lib
```

(下页继续)

(续上页)

```
@ffi.def_extern()
def my_callback(x, y):
    return 42
```

通过获取 lib.my_callback 获得 <cdata> 指针函数对象。这个 <cdata> 可以传递给 C 代码，然后像回调一样工作: 当 C 代码调用此函数指针时，调用 Python 函数 my_callback。(您需要将 lib.my_callback 传递给 C 语言代码，而不是 my_callback: 后者只是上面的 Python 函数，不能传递给 C 语言。)

CFFI 通过将 my_callback 定义为静态 C 语言函数来实现此功能，写在 set_source() 代码之后。<cdata> 然后指向此功能。这个函数的作用是调用 Python 函数对象，该函数在运行时附加 @ffi.def_extern()。

@ffi.def_extern() 装饰器应该应用于 **全局函数**，一个用于同名的每个 extern "Python" 函数。

为了支持某些极端情况，可以通过再次调用 @ffi.def_extern() 来重新定义附加的 Python 函数，但是不建议这样做! 更好地为此名称附加单个全局 Python 函数，并首先灵活地写出来。这是因为每个 extern "Python" 函数只能变成一个 C 语言函数。调用 @ffi.def_extern() 会再次更改此函数的 C 逻辑以调用新的 Python 函数; 旧的 Python 函数不再可调用。从 lib.my_function 获得的 C 语言函数指针始终是此 C 语言函数的地址，即它保持不变。

6.7.1 外部"Python" 和 void * 参数

如前所述，您不能使用 extern "Python" 来生成可变数量的语言 C 函数指针。然而，在纯 C 语言代码中也无法实现该结果。因此，C 通常使用 void *data 参数定义回调。您可以使用 ffi.new_handle() 和 ffi.from_handle() 通过 void * 参数传递 Python 对象。例如，如果回调的 C 语言类型是:

```
typedef void (*event_cb_t)(event_t *evt, void *userdata);
```

并通过调用此函数来注册事件:

```
void event_cb_register(event_cb_t cb, void *userdata);
```

然后你会在构建脚本中编写这个:

```
ffibuilder.cdef("""
    typedef ... event_t;
    typedef void (*event_cb_t)(event_t *evt, void *userdata);
    void event_cb_register(event_cb_t cb, void *userdata);

    extern "Python" void my_event_callback(event_t *, void *);
""")
ffibuilder.set_source("_demo_cffi", r"""
    #include <the_event_library.h>
""")
```

并在您的主应用程序中注册这样的事件:

```
from _demo_cffi import ffi, lib

class Widget(object):
    def __init__(self):
        userdata = ffi.new_handle(self)
        self._userdata = userdata     # must keep this alive!
        lib.event_cb_register(lib.my_event_callback, userdata)

    def process_event(self, evt):
        print "got event!"

@ffi.def_extern()
def my_event_callback(evt, userdata):
    widget = ffi.from_handle(userdata)
    widget.process_event(evt)
```

其他一些库没有明确的 `void *` 参数，但是允许您将 `void *` 附加到现有结构中。例如，库可能会说 `widget->userdata` 是为应用程序保留的通用字段。如果事件的签名现在是这样的话:

```
typedef void (*event_cb_t)(widget_t *w, event_t *evt);
```

然后你可以使用低级 `widget_t *` 中的 `void *` 字段:

```
from _demo_cffi import ffi, lib

class Widget(object):
    def __init__(self):
        ll_widget = lib.new_widget(500, 500)
        self.ll_widget = ll_widget       # <cdata 'struct widget *'>
        userdata = ffi.new_handle(self)
        self._userdata = userdata        # must still keep this alive!
        ll_widget.userdata = userdata    # this makes a copy of the "void *"
        lib.event_cb_register(ll_widget, lib.my_event_callback)

    def process_event(self, evt):
        print "got event!"

@ffi.def_extern()
def my_event_callback(ll_widget, evt):
    widget = ffi.from_handle(ll_widget.userdata)
```

(下页继续)

(续上页)

```
    widget.process_event(evt)
```

6.7.2 从 C 语言直接访问外部"Python"

如果你想直接从 set_source() 编写的 C 代码访问一些 extern "Python" 函数，你需要在之前写一个声明。默认情况下，它必须是静态的，但请参阅下一段 。) 在 C 代码之后由 CFFI 添加此函数的实际实现——这是必需的，因为声明可能使用由 set_source() 定义的类型 (例如上面的 event_t 来自 #include)，所以在此之前不能生成它。

```
ffibuilder.set_source("_demo_cffi", r"""
    #include <the_event_library.h>

    static void my_event_callback(widget_t *, event_t *);

    /* here you can write C code which uses '&my_event_callback' */
""")
```

这也可以用来编写直接调用 Python 的自定义 C 代码。这是一个例子 (在这种情况下效率很低，但如果 my_algo() 中的逻辑要复杂得多，则可能会有用):

```
ffibuilder.cdef("""
    extern "Python" int f(int);
    int my_algo(int);
""")
ffibuilder.set_source("_example_cffi", r"""
    static int f(int);   /* the forward declaration */

    static int my_algo(int n) {
        int i, sum = 0;
        for (i = 0; i < n; i++)
            sum += f(i);     /* call f() here */
        return sum;
    }
""")
```

6.7.3 外部"Python+C"

使用 extern "Python" 声明的函数在 C 源中生成为 static 函数。但是，在某些情况下，将它们设置为非静态是很方便的，通常当您想要从其他 C 语言源文件直接调用它们时。要做到这一点，你可以说 extern "Python+C" 而不只是 extern "Python"。*版本 1.6 中的新功能。*

如果 cdef 包含	然后 CFFI 生成
`extern "Python" int f(int);`	`static int f(int) { /* code */ }`
`extern "Python+C" int f(int);`	`int f(int) { /* code */ }`

名称 `extern "Python+C"` 来自于我们想要两种意义上的外部函数: 作为一个 `extern "Python"`，并且作为非静态的 C 语言函数。

你不能让 CFFI 生成额外的宏或其他特定于编译器的东西，比如 GCC `__attribute__`。您只能控制该功能是否应该是 `static` 的。但通常，这些属性必须与函数 *头文件*一起写入，如果函数 *实现*不重复它们就没问题:

```
ffibuilder.cdef("""
    extern "Python+C" int f(int);      /* not static */
""")
ffibuilder.set_source("_example_cffi", r"""
    /* the forward declaration, setting a gcc attribute
       (this line could also be in some .h file, to be included
       both here and in the other C files of the project) */
    int f(int) __attribute__((visibility("hidden")));
""")
```

6.7.4 外部"Python": 参考

`extern "Python"` 必须出现在 cdef() 中。就像 C++ `extern "C"` 语法一样，它也可以用于围绕一组函数的大括号:

```
extern "Python" {
    int foo(int);
    int bar(int);
}
```

The `extern "Python"` 函数现在不能是可变参数函数。这可以在将来实施。(这个 演示 展示了如何做到这一点，但它有点冗长。)

每个对应的 Python 回调函数都是使用 `@ffi.def_extern()` 装饰器定义的。编写此函数时要小心: 如果它引发异常，或尝试返回错误类型的对象，那么异常无法传播。而是将异常打印到 stderr，并使 C 语言级回调返回默认值。这可以通过 `error` 和 `onerror` 来控制，如下面所描述的。

`@ffi.def_extern()` 装饰器接受这些可选参数:

- `name`: cdef 中写入的函数的名称。默认情况下，它取自您装饰的 Python 函数的名称。
- `error`: 如果 Python 函数引发异常则返回一个值。默认为 0 或 null。异常仍然打印到 stderr，所以这应该只用作最后的解决方案。

- onerror: 如果你想确保捕获所有异常，使用 @ffi.def_extern(onerror=my_handler)。如果发生异常并指定了 onerror，然后调用 onerror(exception, exc_value, traceback)。这在某些情况下非常有用，在这种情况下，您不能简单地编写 try: except: 在主回调函数中，因为它可能无法捕获信号处理程序引发的异常: 如果在 C 中发生信号，则尽快调用 Python 信号处理程序，这是在进入回调函数之后但在执行 try: 之前。如果信号处理程序抛出，我们还没有进入 try: except:。

  如果调用 onerror 并正常返回，然后假设它自己处理异常并且没有任何内容打印到 stderr。如果 onerror 抛出，则会打印两个 traceback 信息。最后，onerror 本身可以在 C 语言中提供回调的结果值，但不一定: 如果它只是返回 None————或者如果 onerror 本身失败————那么将使用 error 的值，如果有的话。

  注意下面的技巧: 在 onerror 中，您可以按如下方式访问原始回调参数。首先检查 traceback 是否为 None (它是 None 例如如果整个函数成功运行但转换返回值时出错: 这在调用后发生). 如果 traceback 不是 None，traceback.tb_frame 是最外层函数的框架，即直接用 @ffi.def_extern() 装饰的函数的框架。所以你可以通过阅读 traceback.tb_frame.f_locals['argname'] 获得该帧中 argname 的值。

6.8 回调 (旧式)

以下是如何创建一个包含函数指针的新 <cdata> 对象，该函数调用您选择的 Python 函数:

```
>>> @ffi.callback("int(int, int)")
>>> def myfunc(x, y):
...     return x + y
...
>>> myfunc
<cdata 'int(*)(int, int)' calling <function myfunc at 0xf757bbc4>>
```

请注意 "int(*)(int, int)" 是 C 函数指针类型，而 "int(int, int)" 是 C 函数类型。两者都可以指定为 ffi.callback() 结果是相同的。

警告: 为 ABI 模式提供回调或向后兼容。如果你正在使用 out-of-line API 模式，建议使用*extern "Python"* 机制而不是回调: 它提供更快，更清洁的代码。它还避免了旧式回调的几个问题:

- 在不太常见的架构上，更容易在回调上崩溃 (例如在 NetBSD 上);
- 在 PAX and SELinux 等强化系统上，额外的内存保护可能会产生干扰 (例如，在 SELinux 上您需要在 deny_execmem 设置为 off 的情况下运行)。
- On Mac OS X, 您需要为应用程序提供授权 com.apple.security.cs.allow-unsigned-executable-memory。

还要注意尝试针对此问题的 cffi 修复————请参阅 ffi_closure_alloc 分支————但没有合并，因为它使用 fork() 创建潜在的 memory corruption。

> 换一种说法: 是的，允许在程序中写入 + 执行内存是危险的; 这就是为什么存在上述各种”强化”选项的原因。但与此同时，这些选择为另一次攻击打开了大门: 如果程序分支然后尝试调用任何 `ffi.callback()`，然后，这会立即导致崩溃————或者，攻击者只需要很少的工作就可以执行任意代码。对我来说，它听起来比原来的问题更危险，这就是为什么 cffi 没有与他一起使用的原因。
>
> 要在受影响的平台上一劳永逸地解决问题，您需要重构所涉及的代码，以便它不再使用 `ffi.callback()`.

警告: 与 ffi.new() 一样，ffi.callback() 返回一个拥有其 C 语言数据所有权的 cdata。(在这种情况下，必要的 C 语言数据包含用于执行回调的 libffi 数据结构。) 这意味着只要此 cdata 对象处于活动状态，就只能调用回调。如果将函数指针存储到 C 语言代码中，只要可以调用回调，就确保你也保持这个对象的存活。最简单的方法是始终只在模块级使用 `@ffi.callback()`，并尽可能使用 ffi.new_handle() 传递”context” 信息。例:

```
# a good way to use this decorator is once at global level
@ffi.callback("int(int, void *)")
def my_global_callback(x, handle):
    return ffi.from_handle(handle).some_method(x)


class Foo(object):

    def __init__(self):
        handle = ffi.new_handle(self)
        self._handle = handle    # must be kept alive
        lib.register_stuff_with_callback_and_voidp_arg(my_global_callback, handle)

    def some_method(self, x):
        print "method called!"
```

(另请参阅上面关于**‘外部”Python”‘__** 的部分，使用相同的一般风格。)

请注意，不支持可变参数函数类型的回调。解决方法是添加自定义 C 代码。在下面的示例中，回调获取第一个参数，该参数计算传递多少额外的 `int` 参数:

```
# file "example_build.py"

import cffi

ffibuilder = cffi.FFI()
ffibuilder.cdef("""
    int (*python_callback)(int how_many, int *values);
    void *const c_callback;   /* pass this const ptr to C routines */
""")
```

(下页继续)

(续上页)

```
ffibuilder.set_source("_example", r"""
    #include <stdarg.h>
    #include <alloca.h>
    static int (*python_callback)(int how_many, int *values);
    static int c_callback(int how_many, ...) {
        va_list ap;
        /* collect the "..." arguments into the values[] array */
        int i, *values = alloca(how_many * sizeof(int));
        va_start(ap, how_many);
        for (i=0; i<how_many; i++)
            values[i] = va_arg(ap, int);
        va_end(ap);
        return python_callback(how_many, values);
    }
""")
ffibuilder.compile(verbose=True)
```

```
# file "example.py"

from _example import ffi, lib

@ffi.callback("int(int, int *)")
def python_callback(how_many, values):
    print ffi.unpack(values, how_many)
    return 0
lib.python_callback = python_callback
```

弃用: 你也可以使用 `ffi.callback()` 作为装饰器而不是直接作为 `ffi.callback("int(int, int)", myfunc)`。这是不鼓励的: 使用这个样式，我们更有可能在它仍在使用时过早忘记回调对象。

`ffi.callback()` 装饰器也接受可选参数 `error`，并从 CFFI 版本 1.2 接受可选参数 `onerror`。这两个工作方式与*上面描述的”外部 Python”*相同。

6.9 Windows: 调用约定

在 Win32 上，函数可以有两个主要的调用约定: ”cdecl” (默认)，或”stdcall” (也被称为”WINAPI”)。还有其他罕见的调用约定，但这些都不受支持。*版本 1.3 中的新功能。*

当您从 Python 发出调用到 C 时，实现是这样的，它适用于这两个主要调用约定中的任何一个; 你不必指定它。但是，如果您操作”函数指针”类型的变量或声明回调，则调用约定必须正确。这是通过在类型中写入 `__cdecl` 或 `__stdcall` 来完成的，就像在 C 中一样:

```
@ffi.callback("int __stdcall(int, int)")
def AddNumbers(x, y):
    return x + y
```

或:

```
ffibuilder.cdef("""
    struct foo_s {
        int (__stdcall *MyFuncPtr)(int, int);
    };
""")
```

支持 `__cdecl` 但始终是默认值，因此可以省略它。在 `cdef()` 中，您还可以使用 `WINAPI` 作为 `__stdcall` 的等效项。如上所述，它几乎不需要 (但不会受到影响) 在 `cdef()` 中声明一个普通函数时说明 `WINAPI` 或 `__stdcall`。(如果使用 `ffi.addressof()` 显式指向此函数的指针，或者函数是 `extern "Python"`，仍然可以看到差异。)

这些调用约定说明符被接受但在 32 位 Windows 以外的任何平台上都被忽略。

在 1.3 之前的 CFFI 版本中，调用约定说明符无法识别。在 API 模式下，您可以通过使用间接来解决它，就像关于回调 (`"example_build.py"`) 一节中的示例一样。在 ABI 模式下无法使用 stdcall 回调。

6.10 FFI 接口

(FFI 接口的参考已移至 下一页.)

CHAPTER 7

CFFI 参考

Contents

7.1 FFI 接口

这个页面记录了运行时接口的两种类型*"FFI"* 和*"CompiledFFI"*。这两种类型彼此非常相似。如果导入 *out-of-line* 模块，则会获得 *CompiledFFI* 对象。您从显式编写 *cffi.FFI()* 获得 *FFI* 对象。与 *CompiledFFI* 不同，*FFI* 类型还记录了其他方法在 下一页.

7.1.1 ffi.NULL

ffi.NULL: 类型为 `<cdata 'void *'>` 的常量 NULL。

7.1.2 ffi.error

ffi.error: 他在各种情况下引发 Python 异常。(不要将它与 `ffi.errno` 混淆。)

7.1.3 ffi.new()

ffi.new(cdecl, init=None): 根据指定的 C 语言类型分配实例并返回指向它的指针。指定的 C 语言类型必须是指针或数组: `new('X *')` 分配一个 X 并返回一个指向它的指针，而 `new('X[n]')` 分配一个 n 个 X 的数组并返回一个引用它的数组 (它主要像指针一样工作，就像在 C 中一样). 您还可以使用 `new('X[]', n)` 来分配非常数长度为 n 的数组。请参阅其他有效初始化值的 详细文档。

当返回的 `<cdata>` 对象超出范围时，将释放内存。换句话说，返回的 `<cdata>` 对象拥有它指向的类型 `cdecl` 值的所有权。这意味着只要此对象保持活动状态，就可以使用原始数据，但不能长时间使用。将指针复制到其他地方的内存时要小心，例如进入另一个结构。而且，这意味着像 `x = ffi.new(...)[0]` 这样的一行 *总是错误的:* 新分配的对象超出范围即刻，因此立即被释放，`x` 是垃圾变量。

在应用可选的初始化值之前，返回的内存最初被清除 (用零填充)。有关性能，请参阅*ffi.new_allocator()* 以获取分配非零初始化内存的方法。

版本 1.12 中的新功能: 另见 `ffi.release()`。

7.1.4 ffi.cast()

ffi.cast("C type", value): 类似于 C 语言转换 (cast): 返回使用给定值初始化的命名 C 语言类型的实例。该值在任何类型的整数或指针之间转换。

7.1.5 ffi.errno, ffi.getwinerror()

ffi.errno: 从此线程中最近的 C 语言调用收到的 `errno` 的值，并传递给后面的 C 语言调用。(这是一个线程本地读写属性。)

ffi.getwinerror(code=-1): 在 Windows 上，除了 `errno` 之外，我们还在函数调用中保存和恢复 `GetLastError()` 值。此函数将此错误代码作为元组 `(code, message)` 返回，在引发 WindowsError 时添加类似 Python 的可读消息。如果给出参数 `code`，则将该代码格式化为消息而不是使用 `GetLastError()`。(则将该代码格式化为消息而不是使用 `GetLastError()` 函数)

7.1.6 ffi.string(), ffi.unpack()

ffi.string(cdata, [maxlen]): 从'cdata' 返回一个 Python 字符串 (或 unicode 字符串)。

- 如果'cdata' 是一个指针或字符数组或字节，则返回以 null 结尾的字符串。返回的字符串将一直延伸到第一个空字符。'maxlen' 参数限制我们查找空字符的距离。如果'cdata' 是一个数组，那么'maxlen' 默认为它的长度。请参阅下面的 `ffi.unpack()` 以获取继续经过第一个空字符的方法。*Python 3:* 这会返回一个 `bytes`，而不是 `str`.
- 如果'cdata' 是 wchar_t 的指针或数组，则返回遵循相同规则的 unicode 字符串。*版本 1.11 中的新功能:* 可以是 char16_t 或 char32_t。
- 如果'cdata' 是单个字符或字节或 wchar_t 或 charN_t，则将其作为字节字符串或 unicode 字符串返回。(请注意，在某些情况下，单个 wchar_t 或 char32_t 可能需要长度为 2 的 Python unicode 字符串。)
- 如果'cdata' 是枚举，则将枚举数的值作为字符串返回。如果该值超出范围，则只返回字符串整数。

ffi.unpack(cdata, length): 解包给定长度的 C 语言数据数组，返回 Python 字符串/ unicode/list。'cdata' 应该是一个指针; 如果是数组，则首先将其转换为指针类型。*版本 1.6 中的新功能*。

- 如果'cdata' 是指向'char' 的指针，则返回一个字节字符串。它不会在第一个空值处停止。(一个等效的方法是 `ffi.buffer(cdata, length)[:]`。)
- 如果'cdata' 是指向'wchar_t' 的指针，则返回一个 unicode 字符串。('length' 以 wchar_t 的数量来衡量; 它不是以字节为单位的大小。) *版本 1.11 中的新功能:* 也可以是 char16_t 或 char32_t。

- 如果'cdata' 是指向其他任何内容的指针，则返回给定'length' 的列表。(一个较慢的方法是 `[cdata[i] for i in range(length)]`。)

7.1.7 ffi.buffer(), ffi.from_buffer()

ffi.buffer(cdata, [size]): 返回一个缓冲区对象，该对象引用给定'cdata' 指向的原始 C 数据，其长度为'size' 字节。Python 调用" 一个缓冲区"，或者更准确地说是" 支持缓冲区接口的对象"，是一个表示一些原始内存的对象，可以传递给各种内置或扩展函数; 这些内置函数可以读取或写入原始内存直接，无需额外的拷贝。

'cdata' 参数必须是指针或数组。如果未指定，缓冲区的大小是 `cdata` 指向的大小，或者数组的整个大小。

以下是 buffer() 有用的几个示例:

- 使用 `file.write()` 和 `file.readinto()` 与这样的缓冲区 (对于以二进制模式打开的文件)
- 覆盖结构的内容: 如果 `p` 是指向一个 cdata，使用 `ffi.buffer(p)[:] = newcontent`，其中 `newcontent` 是一个字节对象 (`str` in Python 2)。

请记住，就像在 C 语言中一样，您可以使用 `array + index` 来获取指向数组的索引项的指针。(在 C 中你可能更自然地编写 `&array[index]`，但这是等价的。)

返回的对象的类型不是内置 `buffer` ，也不是 `memoryview` 类型，因为这些类型的 API 在 Python 版本中变化太大。相反，除了支持缓冲区接口之外，它还具有以下 Python API (Python 2 的 `buffer` 子集。):

- `buf[:]` or `bytes(buf)`: 将数据复制出缓冲区，返回常规字节字符串 (或 `buf[start:end]` 作为一个部分)
- `buf[:] = newstr`: 将数据复制到缓冲区中 (或 `buf[start:end] = newstr`)
- `len(buf)`，`buf[index]`，`buf[index] = newchar`: 作为一系列字符访问。

`ffi.buffer(cdata)` 返回的缓冲区对象使 `cdata` 对象保持活动状态: 如果它最初是一个拥有的 cdata，那么只要缓冲区处于活动状态，它的拥有内存就不会被释放。

Python 2/3 兼容性说明: 你应该避免使用 `str(buf)`，因为它在 Python 2 和 Python 3 之间产生不一致的结果。(这类似于 `str()` 在常规字节字符串上给出不一致的结果)。请改用 `buf[:]`。

版本 1.10 中的新功能: `ffi.buffer` 此时是返回的缓冲区对象的类型; `ffi.buffer()` 实际上调用了构造函数。

ffi.from_buffer([cdecl,] python_buffer, require_writable=False): 返回一个 cdata 数组 (默认情况下为 `<cdata 'char[]'>`)，指向给定 Python 对象的数据，该对象必须支持缓冲区接口。请注意，`ffi.from_buffer()` 将通用 Python 缓冲区对象转换为 cdata 对象，而 `ffi.buffer()` 执行相反的转换。两个调用实际上都不会复制任何数据。

`ffi.from_buffer()` 用于包含大量原始数据的对象，如字节数组 (bytearrays) 或 `array.array` 或 numpy 数组。它支持旧的 *缓冲区* API (在 Python 2.x 中) 和新的 *memoryview* API。请注意，如果传递只读缓冲区对象，则仍会获得常规 `<cdata 'char[]'>`; 如果原始缓冲区不希望您这样做，那么您有责任不在那里写。*特别是，永远不要修改字节串!*

只要 ffi.from_buffer() 返回的 cdata 对象处于活动状态，原始对象就会保持活动状态 (并且在内存视图的情况下被锁定)。

一个常见的用例是调用一个带有一些的 c 函数，该 char * 指向一个 python 对象的内部缓冲区; 对于这种情况，您可以直接将 ffi.from_buffer(python_buffer) 作为参数传递给调用。

版本 1.10 中的新功能: python_buffer 可以是支持 buffer/memoryview 接口的任何东西 (unicode 字符串除外)。以前，1.7 版本版本支持 bytearray 对象 (小心，如果你调整 bytearray 的大小 <cdata> 对象将指向释放的内存); 版本 1.8 及以上版本支持字节字符串。

版本 1.12 中的新功能: 添加了可选的第 一个参数 cdecl 和关键字参数 require_writable:

- cdecl 默认为 "char[]"，但是可以为结果指定不同的数组类型。像 "int[]" 这样的值将返回一个整数数组而不是字符，其长度将设置为适合缓冲区的整数数。(如果划分不准确，则向下舍入)。像 "int[42]" 或 "int[2][3]" 这样的值将返回一个正好为 42(相应的 2 乘 3) 整数的数组，如果缓冲区太小则会引发 ValueError。指定 "int[]" 和使用旧代码 p1 = ffi.from_buffer(x); p2 = ffi.cast("int *", p1) 间的区别在于，只要 p2 在使用，旧代码就需要保持 p1 活动，因为只有 p1 保持底层 python 对象活动和锁定。(另外，ffi.from_buffer("int[]", x) 提供了更好的数组绑定检查。)
- 如果 require_writable 设置为 True，则如果从 python_buffer 得的缓冲区是只读的 (例如，如果 python_buffer 是字节字符串)。则函数将失败。确切的异常是由对象本身引发的，对于字节这样的东西，它随 Python 版本而变化，所以不要依赖它。(在版本 1.12 之前，使用修改可以实现相同的效果: 调用 ffi.memmove(python_buffer, b"", 0)。如果对象是可写的，这没有效果，但如果它是只读的，则会失败。) 请记住，CFFI 没有实现 C 关键字 const: 即使您将 require_writable 显式设置为 False，您仍然会得到常规的读写 cdata 指针。

版本 1.12 中的新功能: 另见 ffi.release()。

7.1.8 ffi.memmove()

ffi.memmove(dest, src, n): 将 n 个字节从内存区域 src 复制到内存区域 dest。见下面的例子。受 C 函数 memcpy() 和 memmove() 的启发————就像后者一样，这些区域可以重叠。dest 和 src 中的每一个都可以是 cdata 指针，也可以是支持 buffer/memoryview 接口的 python 对象。在 dest 的情况下，buffer/memoryview 必须是可写的。*版本 1.3 中的新功能。*例:

- ffi.memmove(myptr, b"hello", 5) 将 b"hello" 的 5 个字节复制到 myptr 指向的区域。
- ba = bytearray(100); ffi.memmove(ba, myptr, 100) 将 100 个字节从 myptr 复制到 bytearray ba 中。
- ffi.memmove(myptr + 1, myptr, 100) 将 100 个字节从 myptr 的内存移到 myptr + 1 的内存。

在 1.10 之前的版本中，ffi.from_buffer() 对缓冲区的类型有限制，这使得 ffi.memmove() 更加通用。

7.1.9 ffi.typeof(), ffi.sizeof(), ffi.alignof()

ffi.typeof("C type" or cdata object): 返回与解析后的字符串对应的 <ctype> 类型的对象，或者返回 cdata 实例的 C 语言类型。通常，您不需要调用此函数或在代码中显式操作 <ctype> 对象: 任何接受 C 类型的地方都可以接收字符串或预先解析的 ctype 对象 (由于字符串的缓存，因此没有真正的性能差异)。它在编写类型检查时仍然有用，例如:

```
def myfunction(ptr):
    assert ffi.typeof(ptr) is ffi.typeof("foo_t*")
    ...
```

还要注意，从字符串 "foo_t*" 到 <ctype> 对象的映射存储在一些内部字典中。这样可以确保只有一个 <ctype 'foo_t *'> 对象，因此您可以使用 is 运算符来比较它。缺点是字典项目现在是唯一的。将来，我们可能会添加易懂的改造旧未使用的旧条目。同时，请注意，如果使用许多不同长度的字符串 (如 "int[%d]" % length) 来命名类型，则会创建许多不唯一的缓存项。

ffi.sizeof("C type" or cdata object): 以字节为单位返回参数的大小。参数可以是 C 类型，也可以是 cdata 对象，就像 C 语言中等效的 sizeof 算符一样。

对于 array = ffi.new("T[]", n)，然后 ffi.sizeof(array) 返回 n * ffi.sizeof("T"). *版本 1.9 中的新功能:* 类似的规则适用于末尾具有可变大小数组的结构。更准确地说，如果 p 由 ffi.new("struct foo *", ...) 返回，则 ffi.sizeof(p[0]) 此时返回总分配大小。在以前的版本中，它只用于返回 ffi.sizeof(ffi.typeof(p[0])), 这是忽略可变大小部分的结构的大小。(请注意，由于对齐，ffi.sizeof(p[0]) 可能返回小于 ffi.sizeof(ffi.typeof(p[0])) 的值。)

ffi.alignof("C type"): 返回参数的自然对齐大小 (以字节为单位)。对应于 GCC 中的 __alignof__ 运算符。

7.1.10 ffi.offsetof(), ffi.addressof()

ffi.offsetof("C struct or array type", *fields_or_indexes): 返回给定字段结构中的偏移量。对应于 C 语言中的 offsetof()。

在嵌套结构的情况下，您可以给出几个字段名称。在指针或数组类型的情况下，您还可以提供与数组项对应的数值。例如，ffi.offsetof("int[5]", 2) 等于两个整数的大小，也是如此。ffi.offsetof("int *", 2)。

ffi.addressof(cdata, *fields_or_indexes): 相当于 C 语言中的'&' 运算符:

1. ffi.addressof(<cdata 'struct-or-union'>) 返回一个 cdata，它是指向此结构或联合的指针。返回的指针只有是原始的 cdata 对象才有效; 如果它是直接从 ffi.new() 获得的，请确保它保持活动状态。

2. ffi.addressof(<cdata>, field-or-index...) 返回给定结构或数组中的字段或数组项的地址。对于嵌套结构或数组，您可以提供多个字段或索引以递归查看。注意，ffi.addressof(array, index) 也可以表示为 array + index: 在 CFFI 和 C 中都是如此，其中 &array[index] 只是 array + index。

3. `ffi.addressof(<library>, "name")` 从给定的库对象返回指定函数或全局变量的地址。对于函数，它返回一个包含指向函数的指针的常规 cdata 对象。

请注意，案例 1. 不能用于获取原始或指针的地址，而只能用于获取结构或联合。实现起来很困难，因为只有结构和联合在内部存储为数据的间接指针。如果你需要一个可以获取地址的 C 语言 int，首先使用 `ffi.new("int[1]")`; 同样，对于指针，使用 `ffi.new("foo_t *[1]")`。

7.1.11 ffi.CData, ffi.CType

ffi.CData, ffi.CType: 在本文档的其余部分中称为 `<cdata>` 和 `<ctype>` 的对象的 Python 类型。请注意，某些 cdata 对象实际上可能是 `ffi.CData` 的子类，并且与 ctype 类似，因此您应该检查 `if isinstance(x, ffi.CData)`。此外，`<ctype>` 对象具有许多内建属性: `kind` 和 `cname` 总是存在，根据它们的类型，它们也可能有 `item`, `length`, `fields`, `args`, `result`, `ellipsis`, `abi`, `elements` 和 `relements`。

版本 1.10 中的新功能: `ffi.buffer` 现在也是一种类型。

7.1.12 ffi.gc()

ffi.gc(cdata, destructor, size=0): 返回指向相同数据的新 cdata 对象。稍后，当这个新的 cdata 对象被垃圾收集时，将调用 `destructor(old_cdata_object)`。用法示例: `ptr = ffi.gc(lib.custom_malloc(42), lib.custom_free)`. 请注意，`ffi.new()` 返回类似的对象，返回的指针对象具有所有权，这意味着只要这个确切的返回对象被垃圾收集，就会调用析构函数。

版本 1.12 中的新功能: 另见 `ffi.release()`。

ffi.gc(ptr, None, size=0): 删除对常规调用 `ffi.gc` 返回的对象的所有权，并且在垃圾收集时不会调用析构函数。该对象在本地修改，并且该函数返回 `None`。*版本 1.7 中的新功能: ffi.gc(ptr, None)*

请注意，对于有限的资源应该避免使用 `ffi.gc()`，或者 (cffi 低于 1.11) 用于大内存分配。在 PyPy 上尤其如此: 它的 GC 不知道返回的 `ptr` 有多少内存或多少资源。只有在分配了足够的内存时，它才会运行 GC (因此可能比你预期的更晚地运行析构函数)。而且，析构函数在 PyPy 当时的任何线程中被调用，这对于某些 C 语言库来说可能是一个问题。在这些情况下，请考虑使用自定义 `__enter__()` 和 `__exit__()` 方法编写包装类，在已知时间点分配和释放 C 语言数据，并在 `with` 语句中使用它。在 cffi 1.12 中，另见 `ffi.release()`。

版本 1.11 中的新功能: `size` 参数。如果给定，这应该是 `ptr` 保持活动的大小 (以字节为单位) 的估计值。该信息被传递给垃圾收集器，解决了上述问题的一部分。`size` 参数在 PyPy 上最为重要; 在 CPython 上，到目前为止它被忽略了，但是将来它也可以用来更友好地触发循环引用 GC (参见 CPython 问题 31105)。

可以使用负 `size` 调用 `ffi.gc(ptr, None, size=0)`，以撤销估量。但这不是强制性的: 如果大小估计不匹配，则不会有任何不同步。它只会使得下一次 GC 开始或多或少提前开始。

请注意，如果您有多个 `ffi.gc()` 对象，则将以随机顺序调用相应的析构函数。如果您需要特定顺序，参见问题 340 的讨论。

7.1.13 ffi.new_handle(), ffi.from_handle()

ffi.new_handle(python_object): 返回 `void *` 类型的非 NULL cdata, 其中包含对 `python_object` 的不透明引用。您可以将其传递给 C 函数或将其存储到 C 语言结构中。稍后，您可以使用 **ffi.from_handle(p)** 从具有相同 `void *` 指针的值中检索原始 `python_object`。*调用 ffi.from_handle(p) 无效，如果 new_handle() 返回的 cdata 对象未保持活动状态，则可能会崩溃!*

请参阅下面的*典型用法示例*。

(如果你想知道，这个 `void *` 是不是 `PyObject *` 指针。无论如何，这对 PyPy 没有意义。)

`ffi.new_handle()/from_handle()` 函数在 概念上的工作方式如下:

- `new_handle()` 返回包含 Python 对象引用的 cdata 对象; 我们将它们统称为" 句柄"cdata 对象。这些句柄 cdata 对象中的 `void *` 值是随机的但是唯一的。
- `from_handle(p)` 搜索所有实时" 句柄"cdata 对象，以获得与其 `void *` 值具有相同值 `p` 的对象。然后它返回该句柄 cdata 对象引用的 Python 对象。如果没有找到，则会出现" 未定义的行为" (即崩溃)。

" 句柄"cdata 对象使 Python 对象保持活动状态，类似于 `ffi.new()` 返回一个使一块内存保持活动状态的 cdata 对象。如果句柄 cdata 对象本身不再存在，则关联 `void * -> python_object` 将失效，而 `from_handle()` 将崩溃。

版本 1.4 中的新功能: 对 `new_handle(x)` 的两次调用保证返回具有不同 `void *` 值的 cdata 对象，即使使用相同的 `x` 也是如此。这是一个有用的功能，可以避免以下技巧中出现意外重复的问题: 如果你需要保持 "句柄"，直到明确要求释放它，但没有一个自然的 Python 端附加它，那么最简单的是将它 `add()` 到一个全局集合。稍后可以通过 `global_set.discard(p)` 稍后可以通过 `p` 为任何 cdata 对象，其 `void *` 值比较相等。

用法示例: 假设你有一个 C 语言库，你必须调用一个 `lib.process_document()` 函数来调用一些回调。`process_document()` 函数接收指向回调和 `void *` 参数的指针。然后使用等于提供值的 `void *data` 参数调用回调。在这种典型情况下，您可以像这样实现它 (out-of-line API 模式):

```
class MyDocument:
    ...

    def process(self):
        h = ffi.new_handle(self)
        lib.process_document(lib.my_callback,    # the callback
                             h,                  # 'void *data'
                             args...)
        # 'h' stays alive until here, which means that the
        # ffi.from_handle() done in my_callback() during
        # the call to process_document() is safe

    def callback(self, arg1, arg2):
        ...
```

(下页继续)

(续上页)

```
# the actual callback is this one-liner global function:
@ffi.def_extern()
def my_callback(arg1, arg2, data):
    return ffi.from_handle(data).callback(arg1, arg2)
```

7.1.14 ffi.dlopen(), ffi.dlclose()

ffi.dlopen(libpath, [flags]): 打开并将" 句柄" 作为 <lib> 对象返回到动态库。参见 准备和分发模块。

ffi.dlclose(lib): 显式关闭 ffi.dlopen() 返回的 <lib> 对象。

ffi.RTLD_...: 常量: ffi.dlopen() 的标志。

7.1.15 ffi.new_allocator()

ffi.new_allocator(alloc=None, free=None, should_clear_after_alloc=True): 返回一个新的分配器。是一个可调用的，其行为类似于 ffi.new() ，但使用提供的低级 alloc 和 free 函数。*版本 1.2 中的新功能。*

alloc() 是以 size 作为唯一参数调用的。如果返回空值，则引发 MemoryError。稍后，如果 free 不是 None，则将使用 alloc() 的结果作为参数调用它。两者都可以是 Python 函数，也可以直接是 C 语言函数。如果只有 free 是 None，则不调用释放函数。如果 alloc 和 free 都为 None，则使用默认的 alloc/free 组合。(换句话说，调用 ffi.new(*args) 等同于 ffi.new_allocator()(*args)。)

如果 should_clear_after_alloc 设置为 False，则假定 alloc() 返回的内存已被清除 (或者你对内存垃圾没问题); 否则 CFFI 会清除它。例: 为了提高性能，如果使用 ffi.new() 来分配大内存块，使初始内容保持未初始化状态，则可以执行以下操作:

```
# at module level
new_nonzero = ffi.new_allocator(should_clear_after_alloc=False)

# then replace `p = ffi.new("char[]", bigsize)` with:
    p = new_nonzero("char[]", bigsize)
```

注意: 以下是一般性警告，特别适用于 (但不仅限于) PyPy 5.6 或更早版本 (PyPy > 5.6 尝试说明 ffi.new() 或自定义分配器返回的内存; CPython 使用引用计数)。如果您进行了大量的分配，那么就无法保证何时释放内存。如果要确保内存被及时释放 (例如，在分配更多内存之前)，则应同时避免 new() 和 new_allocator()() 。

另一种方法是声明并调用 C 语言 malloc() 和 free() 函数，或者像 mmap() 和 munmap() 这样的变体。然后，您可以精确地控制分配和释放内存的时间。例如，将这两行添加到现有的 ffibuilder.cdef():

```
void *malloc(size_t size);
void free(void *ptr);
```

然后手动调用这两个函数:

```
p = lib.malloc(n * ffi.sizeof("int"))
try:
    my_array = ffi.cast("int *", p)
    ...
finally:
    lib.free(p)
```

在 cffi 版本 1.12 中，您确实可以使用 `ffi.new_allocator()` 但是使用 `with` 语句 (请参阅 `ffi.release()`) 来强制在已知点调用释放函数。以上相当于此代码:

```
my_new = ffi.new_allocator(lib.malloc, lib.free)  # at global level
...
with my_new("int[]", n) as my_array:
    ...
```

7.1.16 ffi.release() and the context manager

ffi.release(cdata): 从 `ffi.new()`, `ffi.gc()`, `ffi.from_buffer()` 或 `ffi.new_allocator()()` 释放 cdata 对象持有的资源。之后不得使用 cdata 对象。cdata 对象的普通 Python 析构函数释放相同的资源，但这允许在已知的时间释放，而不是在将来的某个未指定的点释放。*版本 1.12 中的新功能。*

`ffi.release(cdata)` 相当于 `cdata.__exit__()`，这意味着您可以使用 `with` 语句来确保在块末尾释放 cdata。(在版本 1.12 及以上):

```
with ffi.from_buffer(...) as p:
    do something with p
```

效果更为精确，如下所示:

- 对于从 `ffi.gc(destructor)` 返回的对象，`ffi.release()` 将导致立即调用 `destructor`。
- 在自定义分配器返回的对象上，立即调用自定义自由函数。
- 在 CPython 上, `ffi.from_buffer(buf)` 锁定缓冲区，因此可以使用 `ffi.release()` 在已知时间解锁它。在 PyPy 上，没有锁定 (到目前为止); `ffi.release()` 的效果仅限于删除链接，即使 cdata 对象保持活动状态，也允许对原始缓冲区对象进行垃圾回收。
- 在 CPython 上，这个方法对 `ffi.new()` 返回的对象没有影响 (到目前为止)，因为内存是与 cdata 对象内联分配的，不能独立释放。可能会在将来的 cffi 版本中修复它。

- 在 PyPy 上, `ffi.release()` 立即释放 `ffi.new()` 内存。它很有用，因为否则内存将保持活动状态，直到下一次 GC 发生。如果使用 `ffi.new()` 分配大量内存并且不使用 `ffi.release()` 分配大量内存并且不使用，PyPy (>= 5.7) 会更频繁地运行其 GC 以进行补偿，因此分配的总内存应保持在边界内无论如何; 但是显式调用 `ffi.release()` 应该通过降低 GC 运行的频率来提高性能。

在 `ffi.release(x)` 之后，不要再使用 `x` 指向的任何内容。作为此规则的一个例外，您可以为完全相同的 cdata 对象 x 多次调用 `ffi.release(x)`; 第一个之后的调用被忽略。

7.1.17 ffi.init_once()

ffi.init_once(function, tag): 运行 `function()` 一次。`tag` 应该是标识函数的原始对象，如字符串: `function()` 仅在我们第一次看到 `tag` 时调用。`function()` 的返回值将被当前和所有将来的 `init_once()` 用相同的标记记住并返回。如果从多个线程并行调用 `init_once()` 则所有调用都会阻塞，直到执行 `function()` 为止。如果 `function()` 引发异常，则会传播它，并且不会缓存任何内容 (即如果我们捕获异常并再次尝试 `init_once()`，将再次调用 `function()`。). *版本 1.4 中的新功能。*

例:

```
from _xyz_cffi import ffi, lib

def initlib():
    lib.init_my_library()

def make_new_foo():
    ffi.init_once(initlib, "init")
    return lib.make_foo()
```

如果已经调用了 `function()`，则 `init_once()` 被优化为非常快速地运行。(在 PyPy 上，成本为零————JIT 通常会删除它生成的机器代码中的所有内容。)

注意: `init_once()` 的一个 动机 是嵌入式案例中"subinterpreters" 的 CPython 概念。如果使用的是 out-of-line API 模式，即使存在多个子解释器，也只调用一次 `function()`，并且所有子解释器之间共享其返回值。目标是模仿传统的 cpython C 扩展模块的 init 代码总共只执行一次，即使有子解释器。在上面的示例中，C 函数 `init_my_library()` 总共调用一次，而不是每个子解释器调用一次。因此，避免 `function()` 中的 python 级副作用。(因为它们只会应用于第一个子解释器中运行); 相反，返回一个值，如下例所示:

```
def init_get_max():
    return lib.initialize_once_and_get_some_maximum_number()

def process(i):
    if i > ffi.init_once(init_get_max, "max"):
        raise IndexError("index too large!")
    ...
```

7.1.18 ffi.getctype(), ffi.list_types()

ffi.getctype("C type" or <ctype>, extra=""): 返回给定 C 类型的字符串表示形式。如果非空，则追加" 额外" 字符串 (或插入更复杂的情况下的正确位置); 它可以是要声明的变量的名称，也可以是类型的额外部分，如 like `"*"` 或 `"[5]"`。例如 `ffi.getctype(ffi.typeof(x), "*")` 返回 C 类型" 指向与 x 相同类型的指针" 的字符串表示形式; 并且 `ffi.getctype("char[80]", "a") == "char a[80]"`。

ffi.list_types(): 返回此 FFI 实例已知的用户类型名称。这将返回一个包含三个名称列表的元组: `(typedef_names, names_of_structs, names_of_unions)`。*版本 1.6 中的新功能。*

7.2 转换

本节介绍了在 *写入* C 数据结构 (或将参数传递给函数调用) 以及从 C 语言数据结构中 *读取* (或获取函数调用的结果) 时允许的所有转换。最后一列给出了允许的特定于类型的操作。

C 语言类型	写入	读取	其他操作
整形和枚举 *[5]*	一个整数或 int() 返回的任何东西 (但不是浮点数!)。必须在范围内。	Python int 或 long，具体取决于类型 (版本 1.10: 或者 bool)	int(), bool() *[6]*, <
`char`	一个长度为 1 或类似 <cdata char> 的字符串	长度为 1 的字符串	int(), bool(), <
`wchar_t`, `char16_t`, `char32_t` *[8]*	一个长度为 1 的 unicode (如果是代理 (码元)，则可能是 2 个) 或其他类似的 <cdata>	一个长度为 1 的 unicode (如果是代理 (码元)，则可能是 2)	int(), bool(), <
`float`, `double`	浮点数或 float() 返回的任何东西	一个 Python 浮点数	float(), int(), bool(), <
`long double`	类似带有 `long double` 的 <cdata>，或者 float() 返回的任何东西	一个 <cdata>，以避免失去精度 *[3]*	float(), int(), bool()
`float _Complex`, `double _Complex`	一个复数或任何 complex() 返回的任何东西	一个 Python 复数	complex(), bool() *[7]*
指针	类似兼容类型的 <cdata> (即相同类型或 `void*`，或作为数组 *[1]*	一个 <cdata>	`[]` *[4]*, +, -, bool()
`void *`	类似带有任何指针或数组类型的 <cdata>		
指向结构体或联合的指针	与指针相同		`[]`, +, -, bool(), 和 read/write struct 字段
函数指针	与指针相同		bool(), call *[2]*
数组	列表或元组的元素	一个 <cdata>	len(), iter(), `[]` *[4]*, +, -
`char[]`, `un/signed char[]`, `_Bool[]`	与数组或 Python 字节字符串相同		len(), iter(), `[]`, +, -
`wchar_t[]`, `char16_t[]`, `char32_t[]`	与数组或 Python unicode 字符串相同		len(), iter(), `[]`, +, -
结构体	字段值的列表或元组或字典，或相同类型的 <cdata>	一个 <cdata>	read/write 字段
联合	与 struct 相同，但最多只有一个字段		read/write 字段

[1] `item *` 是函数参数中的 `item[]`：

在函数声明中，根据 C 标准，`item *` 参数与 `item[]` 参数相同 (并且 `ffi.cdef()` 不记录差异)。所以当你调用这样一个函数时，你可以传递一个 C 类型接受的参数，例如将一个 Python 字符

串传递给一个 char * 参数 (因为它适用于 char[] 参数) 或 int * 参数的整数列表 (它适用于 int[] 参数)。请注意, 即使您要传递单个 item, 也需要在长度为 1 的列表中指定它; 例如, struct point_s * 参数可能会传递为 [[x, y]] 或 [{'x': 5, 'y': 10}]。

作为优化, CFFI 假定具有 char * 参数的函数, 您传递 Python 字符串将不会实际修改传入的字符数组, 因此直接传递 Python 字符串对象内的指针。(在 PyPy 上, 这种优化仅在 PyPy 5.4 和 CFFI 1.8 之后才可用。)

[2] C 函数调用在 GIL 释放时完成。

请注意, 我们假设被调用的函数不使用 Python.h 中的 Python API。例如, 我们之后不会检查它们是否设置了 Python 异常。您可以解决它, 但不建议将 CFFI 与 Python.h 混合使用。(如果您这样做, 在 PyPy 和 Windows 等某些平台上, 您可能需要显式链接到 libpypy-c.dll 才能访问 CPython C API 兼容层; 实际上, PyPy 上的 CFFI 生成的模块本身并没有链接到 libpypy-c.dll 。但实际上, 首先不要这样做。)

[3] long double 的支持:

我们在 cdata 对象中保留 long double 值以避免丢失精度。普通 Python 浮点数只包含 double 的精度。如果你真的想将这样的对象转换为常规的 Python float (即 C double), 请调用 float()。如果你需要对这些数字进行算术而没有任何精度损失, 你需要定义和使用一系列 C 函数, 如 long double add(long double a, long double b);。

[4] 切片 x[start:stop]:

只要你明确指定 start 和 stop, 就允许切片 (并且不给任何 step)。它给出了一个"view"cdata 对象, 它是从 start 到 stop 的所有元素 (数据项)。它是数组类型的 cdata (所以例如将它作为参数传递给 C 函数只会将其转换为指向 start 元素的指针). 与索引一样, 负边界意味着真正的负索引, 如在 C 中。至于切片赋值, 它接受任何可迭代的, 包括元素列表或另一个类似数组的 cdata 对象, 但长度必须匹配。(请注意, 此行为与初始化不同: 例如你可以这样 chararray[10:15] = "hello", 但是指定的字符串必须是正确的长度; 没有添加隐式空字符。)

[5] 枚举像 int 一样处理:

与 C 一样, 枚举类型主要是 int 类型 (unsigned 或 signed, int 或 long; 请注意, GCC 的首选是 unsigned)。例如, 读取结构的枚举字段会返回一个整数。要象征性地比较它们的值, 请使用 if x.field == lib.FOO 之类的代码。如果你真的想要将它们的值作为字符串, 请使用 ffi.string(ffi.cast("the_enum_type", x.field))。

[6] 原始 cdata 上的 bool() :

版本 1.7 中的新功能。在以前的版本中, 它只适用于指针; 对于原语, 它总是返回 True。

N 版本 1.10 中的新功能: C 语言类型 _Bool 或 bool 现在转换为 Python 布尔值。如果 C _Bool 碰巧包含不同于 0 和 1 的值, 则会出现异常 (这种情况在 C 中触发未定义的行为; 如果你真的必须与依赖于它的库接口, 不要在 CFFI 端使用 _Bool)。此外, 从字节字符串转换为 _Bool[] 时, 只接受字节 \x00 和 \x01。

[7] libffi 不支持复数:

版本 1.11 中的新功能: CFFI 现在直接支持复数。但请注意，libffi 没有。这意味着 CFFI 无法调用直接作为参数类型或返回 complex 类型的 C 函数，除非它们直接使用 API 模式。

[8] `wchar_t`, `char16_t` 和 `char32_t`

请参阅下面的*Unicode 字符类型*。

7.2.1 支持文件

您可以使用 `FILE *` 参数声明 C 函数，并使用 Python 文件对象调用它们。如果需要，你也可以这样做 `c_f = ffi.cast("FILE *", fileobj)` 然后传递 `c_f`。

但请注意，CFFI 通过尽力而为的方法来做到这一点。如果您需要更好地控制缓冲，刷新和及时关闭 `FILE *`，那么您不应该对 `FILE *` 使用此特殊支持。相反，您可以使用 fdopen() 处理您明确使用的常规 `FILE *` cdata 对象，如下所示:

```
ffi.cdef('''
    FILE *fdopen(int, const char *);   // from the C <stdio.h>
    int fclose(FILE *);
''')

myfile.flush()                    # make sure the file is flushed
newfd = os.dup(myfile.fileno())   # make a copy of the file descriptor
fp = lib.fdopen(newfd, "w")       # make a cdata 'FILE *' around newfd
lib.write_stuff_to_file(fp)       # invoke the external function
lib.fclose(fp)                    # when you're done, close fp (and newfd)
```

无论如何，对 `FILE *` 的特殊支持在 CPython 3.x 和 PyPy 上以类似的方式实现，因为这些 Python 实现的文件本身不是基于 `FILE *`。这样做显式地提供了更多的控制。

7.2.2 Unicode 字符类型

`wchar_t` 类型与底层平台具有相同的签名。例如，在 Linux 上，它是一个带符号的 32 位整数。但是，`char16_t` 和 `char32_t` 类型 (*版本 1.11 中的新功能*) 始终是无符号的。

请注意，CFFI 假定这些类型在本机字节序中包含 UTF-16 或 UTF-32 字符。更确切地说:

- 假设 `char32_t` 包含 UTF-32 或 UCS4，它只是 unicode 码位;
- 假设 `char16_t` 包含 UTF-16，即 UCS2 加代理 (码元);
- 假设 `wchar_t` 包含 UTF-32 或 UTF-16，基于其实际平台定义的大小为 4 或 2 个字节。

这一假设是真是假，C 语言没有说明。理论上，您正在链接的 C 语言库可以使用其中一种具有不同含义的类型。然后，您需要自己处理它，例如，在 `cdef()` 使用 `uint32_t` 而不是 `char32_t` ，并手动构建预期的 `uint32_t` 数组。

Python 本身可以使用 `sys.maxunicode == 65535` 或 `sys.maxunicode == 1114111` (Python >= 3.3 始终是 1114111)。这改变了代理的处理方式 (这是一对 16 位" 字符"，实际上代表一个值大于 65535 的码位)。如果您的 Python 是 `sys.maxunicode == 1114111`，那么它可以存储任意 unicode 码位; 从 Python unicodes 转换为 UTF-16 时会自动插入代理，并在转换回时自动删除。另一方面，如果你的 Python 是 `sys.maxunicode == 65535`，那么它就是另一种方式: 从 Python unicodes 转换为 UTF-32 时会删除代理，并在转换回时添加。换句话说，仅在存在大小不匹配时才进行代理转换。

请注意，未指定 Python 的内部表示。例如，在 CPython >= 3.3，它将使用 1 或 2 或 4 字节数组，具体取决于字符串实际包含的内容。使用 CFFI，当您将 Python 字节字符串传递给期望 `char*` 的 C 函数时，我们直接传递指向现有数据的指针，而无需临时缓冲区; 但是，由于内部表示的变化，使用 unicode 字符串参数和 `wchar_t*` / `char16_t*` / `char32_t*` 类型无法完成相同的操作。因此，为了保持一致性，CFFI 总是为 unicode 字符串分配一个临时缓冲区。

警告: 现在，如果你将 `char16_t` 和 `char32_t` 与 `set_source()` 一起使用，你必须确保自己的类型是由你提供给 `set_source()` 的 C 语言源代码中声明的。如果 `#include` 显式使用它们的库，则会声明它们，例如，使用 C++ 11 时。否则，您需要在 Linux 上使用 `#include <uchar.h>`，或者通常使用类似于 `typedef uint16_t char16_t;` 的方法。这不是由 CFFI 自动完成的，因为 `uchar.h` 不是跨平台的标准，如果类型恰好已经定义，写上面的 `typedef` 会崩溃。

CHAPTER 8

编写和分发模块

Contents

在项目中使用 CFFI 有三种或四种不同的方法。按顺序排列:

- **"in-line", "ABI 模式"**:

```
import cffi

ffi = cffi.FFI()
ffi.cdef("C-like declarations")
lib = ffi.dlopen("libpath")

# use ffi and lib here
```

- **"out-of-line", 但仍然是 "ABI 模式"**，对于组织代码和减少导入时间很有用:

```
# in a separate file "package/foo_build.py"
import cffi

ffibuilder = cffi.FFI()
ffibuilder.set_source("package._foo", None)
ffibuilder.cdef("C-like declarations")

if __name__ == "__main__":
    ffibuilder.compile()
```

运行 `python foo_build.py` 会生成一个文件 `_foo.py`，然后可以在主程序中导入该文件:

```
from package._foo import ffi
lib = ffi.dlopen("libpath")

# use ffi and lib here
```

- **"out-of-line", "API 模式"** 为您提供了在 C 级别而不是二进制级别访问 C 库的最大灵活性和速度:

```
# in a separate file "package/foo_build.py"
import cffi

ffibuilder = cffi.FFI()
ffibuilder.set_source("package._foo", r"""real C code""")   # <=
ffibuilder.cdef("C-like declarations with '...'")

if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

运行 `python foo_build.py` 生成一个文件 `_foo.c` 并调用 C 编译器将其转换为文件 `_foo.so` (或 `_foo.pyd` 或 `_foo.dylib`)。它是一个 C 扩展模块，可以在主程序中导入:

```
from package._foo import ffi, lib
# no ffi.dlopen()

# use ffi and lib here
```

- 最后，在编写 setup.py 时，您可以 (但不必) 使用 CFFI 的 **Distutils 或 Setuptools 集成**。对于 Distutils (仅在 out-of-line API 模式):

```
# setup.py (requires CFFI to be installed first)
from distutils.core import setup

import foo_build   # possibly with sys.path tricks to find it

setup(
    ...,
    ext_modules=[foo_build.ffibuilder.distutils_extension()],
)
```

对于 Setuptools (out-of-line, 但适用于 ABI 或 API 模式; 推荐):

```
# setup.py (with automatic dependency tracking)
from setuptools import setup

setup(
    ...,
    setup_requires=["cffi>=1.0.0"],
    cffi_modules=["package/foo_build.py:ffibuilder"],
    install_requires=["cffi>=1.0.0"],
)
```

再次注意，foo_build.py 示例包含以下行，这意味着仅在导入 package.foo_build 时实际上不编译 ffibuilder，它将由 Setuptools 逻辑独立编译，使用 Setuptools 提供的编译参数:

```
if __name__ == "__main__":    # not when running with setuptools
    ffibuilder.compile(verbose=True)
```

- 请注意，尝试查找项目使用的所有模块的一些捆绑工具 (如 PyInstaller) 将在 out-of-line 模式下避免 _cffi_backend，因为您的程序不包含显式 import cffi 或 import _cffi_backend。您需要显式添加 _cffi_backend (作为 PyInstaller 中的"hidden import"，但通常也可以通过在主程序中添加 import _cffi_backend 来更好地完成它)。

请注意，CFFI 实际上包含两个不同的 FFI 类。页面 使用 ffi/lib 对象 描述了常用功能。这是你从上面的 from package._foo import ffi 中得到的。另一方面，扩展的 FFI 类是从 import cffi; ffi_or_ffibuilder

= cffi.FFI() 得到的; 它具有相同的功能 (用于 in-line 使用)，但也有下面描述的额外方法 (编写 FFI)。注意: 当代码是关于生成 _foo.so 时，我们在 out-of-line 上下文中使用名称 ffibuilder 而不是 ffi; 这是尝试将它与后来 from _foo import ffi 所获得的不同 ffi 对象区分开来。

这种功能分离的原因是使用 CFFI 外联的常规程序根本不需要导入 cffi 纯 Python 包。(在内部，它仍然需要 _cffi_backend，一个 CFFI 附带的 C 扩展模块; 这就是为什么 CFFI 也在 install_requires=.. 上面列出的原因。将来，这可能会拆分为仅安装 _cffi_backend 的不同 PyPI 包。)

请注意，确实存在一些小的差异: 值得注意的是，从 from _foo import ffi 返回一个用 C 语言编写的类型的对象，它不允许你向它添加随机属性 (它也没有 Python 版本的所有下划线前缀内部属性)。类似地，除了对全局变量的写入之外，C 版本返回的 lib 对象是只读的此外，lib.__dict__ 在版本 1.2 之前不起作用，或者如果 lib 恰好声明一个名为 __dict__ 的名称 (使用 dir(lib) 代替)。对于连续版本中添加的 lib.__class__，lib.__all__ 和 lib.__name__ 也是如此。

8.1 ffi/ffibuilder.cdef(): 声明类型和函数

ffi/ffibuilder.cdef(source): 解析给定的 C 源。它在 C 源中注册所有函数，类型，常量和全局变量。这些类型可以在 ffi.new() 和其他函数中立即使用。在您可以访问函数和全局变量之前，您需要为 ffi 提供另一条信息: 它们实际上来自那里 (您可以使用 ffi.dlopen() 或 ffi.set_source() 执行此操作)。

内部解析 C 语言源代码 (使用 pycparser)。此代码不能包含 #include。它通常应该是从手册页中提取的自包含声明。它可以假设存在的唯一事物是标准类型:

- char, short, int, long, long long (both signed 和 unsigned)
- float, double, long double
- intN_t, uintN_t (对于 N=8,16,32,64), intptr_t, uintptr_t, ptrdiff_t, size_t, ssize_t
- wchar_t (如果后端支持). *版本 1.11 中的新功能:* char16_t 和 char32_t.
- _Bool 和 bool (相等)。如果 C 编译器没有直接支持，则使用 unsigned char 的大小声明它。
- FILE. 看这里。
- 如果在 Windows 上运行，则定义所有 常见的 Windows 类型 (DWORD, LPARAM, 等)。例外: TBYTE TCHAR LPCTSTR PCTSTR LPTSTR PTSTR PTBYTE PTCHAR 不会自动定义; 参见*ffi.set_unicode()*。
- stdint.h 中的其他标准整数类型，如 intmax_t，只要它们映射到 1,2,4 或 8 字节的整数即可。不支持更大的整数。

声明还可以在各个地方包含"...": 这些是将由编译器完成的占位符。有关它的更多信息，请参阅让 C 编译器填补空白。

请注意，上面列出的所有标准类型名称仅作为 默认值处理 (除了那些是 C 语言中的关键词)。如果您的 cdef 包含重新定义上述类型之一的显式 typedef，则忽略上述默认值。(这有点难以干净地实现，因此在某些极端情况下它可能会失败，尤其是错误 Multiple type specifiers with a type tag。如果确实如此，请将其报告为错误。)

可以多次调用 ffi.cdef()。请注意，很多时候调用 ffi.cdef() 的速度很慢，主要考虑 in-line 模式.

ffi.cdef() 调用可选地接受一个额外参数: packed 或 pack。如果传递 packed=True，则在此 cdef 中声明的所有结构都是"packed" 的。(如果您需要 packed 和非 packed 结构，请按顺序使用多个 cdef。) 这与 GCC 中的 __attribute__((packed)) 的含义相似。它指定所有结构字段的对齐方式应为一个字节。(请注意，到目前为止，packed 属性对位字段没有影响，这意味着它们可能与 GCC 上的 packed 方式不同。此外，这对使用 "...;" 声明的结构没有影响，稍后会详细介绍让 *C* 编译器填补空白。) *版本 1.12 中的新功能:* 在 ABI 模式中，你也可以传递 pack=n，整数 n 必须是 2 的幂。则任何字段的对齐限制为 n，否则将大于 n。传递 pack=1 相当于传递 packed=True。这是为了模拟 MSVC 编译器中的 #pragma pack(n)。在 Windows 上，默认值为 pack=8 (从 cffi 1.12 开始); 在其他平台上，默认值为 pack=None。

请注意，您可以在 cdef() 中使用类型修饰符 const 和 restrict (但不是 __restrict 或 __restrict__)，但这对运行时获得的 cdata 对象没有影响 (他们永远不会是 const)。效果仅限于知道全局变量是否为常量。此外，*版本 1.3 中的新功能:* 当使用 set_source() 或 verify()，这两个限定符将从 cdef 复制到生成的 C 语言代码中; 这修复了 C 编译器的警告。

如果从具有额外宏的源代码复制粘贴代码，请注意一个技巧 (例如，Windows 文档使用 SAL 注释，如 _In_ 或 _Out_)。必须在给 cdef() 的字符串中删除这些提示，但可以像这样以编程方式完成:

```
ffi.cdef(re.sub(r"\b(_In_|_Inout_|_Out_|_Outptr_)(opt_)?\b", " ",
  """
    DWORD WINAPI GetModuleFileName(
      _In_opt_ HMODULE hModule,
      _Out_    LPTSTR  lpFilename,
      _In_     DWORD   nSize
    );
  """))
```

另请注意，pycparser 是底层 C 解析器，它以下列格式识别类似预处理器的指令: # NUMBER "FILE"。例如，如果你把 # 42 "foo.h" 放在传递给 cdef() 的字符串的中间，之后会出现两行错误，然后会报告一条以 foo.h:43: 开头的错误消息 (给出数字 42 的行是指令后面的行). *版本 1.10.1 中的新功能:* CFFI 自动将行 # 1 "<cdef source string>" 放在您给 cdef() 的字符串之前。

ffi.set_unicode(enabled_flag): Windows: 如果 enabled_flag 为 True, 在 C 中启用 UNICODE 和 _UNICODE 定义，并声明类型 TBYTE TCHAR LPCTSTR PCTSTR LPTSTR PTSTR PTBYTE PTCHAR 是 (指针) wchar_t。如果 enabled_flag 为 False，则声明这些类型为 (指针) 普通 8 位字符。(如果不调用 set_unicode() 则根本不用预先声明这些类型。)

这个方法背后的原因是很多标准函数都有两个版本，比如 MessageBoxA() 和 MessageBoxW()。官方接口是 MessageBox() 其参数类似于 LPTCSTR。根据是否定义 UNICODE，标准头将通用函数名重命名为两个专用版本之一，并声明正确的 (unicode 或 not) 类型。

U 通常，正确的做法是使用 True 调用此方法。请注意 (特别是在 Python 2 上)，之后，您需要将 unicode 字符串作为参数而不是字节字符串传递。

8.2 ffi.dlopen(): 以 ABI 模式加载库

`ffi.dlopen(libpath, [flags])`: 此函数打开一个共享库并返回类似模块的库对象。当您对系统的 ABI 级别访问权限有限制时，可以使用此选项。(依赖于 ABI 详细信息，获取崩溃而不是 C 编译器错误/警告，以及调用 C 函数的更高开销)。如有疑问，请在概述中再次阅读 ABI 与 API 。

您可以使用库对象来调用先前由 `ffi.cdef()` 声明的函数，读取常量以及读取或写入全局变量。请注意，只要使用 `dlopen()` 加载每个函数并使用正确的函数访问函数，就可以使用单个 `cdef()` 来声明多个库中的函数。

`libpath` 是共享库的文件名，它可以包含完整路径 (在这种情况下，它在标准位置搜索，如 `man dlopen` 中所述)，是否包含扩展名。或者，如果 `libpath` 为 None，则返回标准 C 库 (在 Linux 上可以用来访问 glibc 的功能)。请注意，在使用 Python 3 的 Windows 中 `libpath` 不能为 None。

让我再说一遍: 这提供了对库的 ABI 级访问，因此您需要手动声明所有类型，而不是在创建库时。没有检查。不匹配可能导致随机崩溃。另一方面，API 级访问更安全。速度方面，API 级别的访问速度要快得多 (对性能有相反的误解是很常见的)。

请注意，只有函数和全局变量存在于库对象中; 这些类型存在于 `ffi` 例中，与库对象无关。这是由于 C 模型: 您在 C 中声明的类型不依赖于特定库，只要您 `#include` 其标题即可; 但是你不能在程序中调用函数而不在程序中链接它，因为 `dlopen()` 在 C 中动态地执行。

对于可选的 `flags` 参数，参见 `man dlopen` (在 Windows 上被忽略)。它默认为 `ffi.RTLD_NOW`。

此函数返回一个"library" 对象，当它超出范围时会被关闭。确保在需要时保留库对象。(或者，out-of-line FFI 有一个方法 `ffi.dlclose(lib)`。)

注意: 如果无法直接找到库，则来自 in-line ABI 模式的旧版本的 `ffi.dlopen()` 会尝试使用 `ctypes.util.find_library()`。更新的 out-of-line `ffi.dlopen()` 不再自动执行此操作; 它只是将它接收的参数传递给底层的 `dlopen()` 或 `LoadLibrary()` 函数。如果需要，您可以使用 `ctypes.util.find_library()` 或任何其他方式查找库的文件名。这也意味着 `ffi.dlopen(None)` 不再适用于 Windows; 尝试改为 `ffi.dlopen(ctypes.util.find_library('c'))`。

8.3 ffibuilder.set_source(): 编写 out-of-line 模块

ffibuilder.set_source(module_name, c_header_source, [keywords...])**: 编写 ffi 以生成一个名为 `module_name` 的外部模块。

`ffibuilder.set_source()` 本身不会写任何文件，而只是记录其参数以供日后使用。因此可以在 `ffibuilder.cdef()` 之前或之后调用它。

在 **ABI 模式**，你调用 `ffibuilder.set_source(module_name, None)`。参数是要生成的 Python 模块的名称 (或包内带点号名称)。在此模式下，不会调用 C 编译器。

在 **API 模式**， `c_header_source` 参数是一个字符串，将粘贴到生成的.c 文件中。通常，它被指定为 `r""" ...multiple lines of C code... """` (例如，`r` 前缀允许这些行包含文字 `\n`)。这段 C 语言代码通常包

含一些 #include，但也可能包含更多内容，例如自定义” 包装器”C 语言函数的定义。目标是可以像这样生成.c 文件:

```
// C file "module_name.c"
#include <Python.h>

...c_header_source...

...magic code...
```

其中” 魔术代码” 是从 cdef() 自动生成的。例如，如果 cdef() 包含 int foo(int x);，然后魔术代码将包含用整数参数调用函数 foo() 的逻辑，它本身包含在一些 CPython 或 PyPy 特定的代码中。

set_source() 的关键字参数控制 C 编译器的调用方式。它们直接传递给 distutils 或 setuptools，至少包括 sources, include_dirs, define_macros, undef_macros, libraries, library_dirs, extra_objects, extra_compile_args 和 extra_link_args。您通常至少需要 libraries=['foo'] 才能在 Windows 上与 libfoo.so 或 libfoo.so.X.Y, 或 foo.dll 链接。sources 是一组编译和链接在一起的额外.c 文件 (始终生成上面显示的文件 module_name.c 并自动添加为 sources 的第一个参数)。有关其他参数的更多信息 请参阅 distutils 文档。

内部处理的额外关键字参数是 source_extension，默认为 ".c"。生成的文件实际上调用 module_name + source_extension。例如 C++ (但请注意，仍存在一些已知的 C 与 C ++ 兼容性问题):

```
ffibuilder.set_source("mymodule", r'''
extern "C" {
    int somefunc(int somearg) { return real_cpp_func(somearg); }
}
''', source_extension='.cpp')
```

ffibuilder.set_source_pkgconfig(module_name, pkgconfig_libs, c_header_source, [keywords...])**:

版本 1.12 中的新功能。这相当于 set_source()，但它首先使用列表 pkgconfig_libs 中给出的包名称调用系统实用程序 pkg-config。它收集以这种方式获得的信息，并将其添加到明确提供的 **keywords (如果有) 中。这也许不应该在 Windows 上使用。

如果未安装 pkg-config 程序或不知道所请求的库，则调用将失败并显示 cffi.PkgConfigError。如果有必要，您可以捕获此错误并尝试直接调用 set_source()。(理想情况下，如果 ffibuilder 实例没有方法 set_source_pkgconfig()，您也应该这样做，以支持旧版本的 cffi。)

8.4 让 C 编译器填补空白

如果您使用的是 C 编译器 (”API 模式”)，那么:

- 获取或返回整数或浮点数参数的函数可能被误报: 如果是一个函数由 cdef() 声明为接受一个 int，但实际上需要一个 long，然后 C 编译器处理差异。
- 检查其他参数: 如果将 int * 参数传递给期望 long * 的函数，则会收到编译警告或错误。
- 类似地，在 cdef() 中声明的大多数其他事情都被检查，达到目前为止我们实现的最佳效果; mistakes 给出编译警告或错误。

此外，您可以在 cdef() 中的各个位置使用"..." (按照字面意思, 点点点)，以便让 C 编译器填写详细信息。这些地方是:

- 结构声明: 以"...;" 结尾的任何 struct { } 作为最后一个" 字段" 是部分的: 可能缺少字段和/或已将其声明为无序。此声明将由编译器更正。(但请注意，您只能访问您声明的字段，而不能访问其他字段。) 任何不使用"..." 的 struct 声明都被认为是精确的，但这是检查的: 如果不正确，你会收到错误。
- 整数类型: 语法"typedef int... foo_t;" 将类型 foo_t 声明为整数类型，其未指定精确大小和符号。编译器会搞清楚。(请注意，这需要 set_source(); 它不适用于 verify()。) int... 可以用 long... 或 unsigned long long... 或任何其他原始整数类型替换, 但不起作用。该类型将始终映射到 Python 中的 (u)int(8,16,32,64)_t 之中，但在生成的 C 代码中，仅使用 foo_t。
- *版本 1.3 中的新功能:* 浮点类型: "typedef float... foo_t;" (或者等价"typedef double... foo_t;") 将 foo_t 声明为一个 float 或一个 double; 编译器会弄清楚它是什么。请注意，如果实际的 C 类型更大 (long double 在某些平台上)，则编译将失败。问题是 Python"float" 类型不能用于存储额外的精度。(使用不是点点点的语法 typedef long double foo_t; 像往常一样，它返回的值不是 Python 浮点数，而是 cdata "long double" 对象。)
- 未知类型: 语法"typedef ... foo_t;" 将类型 foo_t 声明为不确定。主要用于 API 采用并返回 foo_t * 而无需查看 foo_t。也适用于"typedef ... *foo_p;"，它声明指针类型 foo_p 而不给不确定类型本身命名。请注意，这种不确定的结构没有已知的大小，这会阻止某些操作执行 (大多数情况下像在 C 语言中)。*您不能使用此语法来声明特定类型，如整数类型！它仅声明不确定的类似结构的类型。*在某些情况下，你需要说 foo_t 不是不确定的，而只是一个你不知道任何字段的结构; 然后你会使用"typedef struct { ...; } foo_t;"。
- 数组长度: 当用作结构字段或全局变量时，数组可以具有未指定的长度，如"int n[...];" 中所示。长度由 C 编译器完成。这与"int n[];" 略有不同，因为后者意味着即使对 C 编译器也不知道长度，因此不会尝试完成它。这支持多维数组: "int n[...][...];"。

  版本 1.2 中的新功能: "int m[][...];"，即 ... 可以在最里面的维度中使用，而不是也用在最外面的维度中。在给出的示例中，假定 m 数组的长度不为 C 编译器所知，但是每个项的长度 (如子数组 m[0]) 总是为 C 编译器所知。换句话说，只有最外层的维度可以在 C 和 CFFI 中指定为 []，但任何维度都可以在 CFFI 中以 [...] 的形式给出。
- 枚举: 如果您不知道声明的常量的确切顺序 (或值)，请使用以下语法: "enum foo { A, B, C, ... };" (后面有个"...")。C 编译器将用于计算常量的确切值。另一种语法是"enum foo { A=..., B, C };" 或甚至"enum foo { A=..., B=..., C=... };"。与结构一样，没有"..." 的 enum 被认为是精确的，并且这被检查。

- 整数常量和宏: 你可以在 cdef 中写一行"#define FOO ...", 使用任何宏名 FOO 都使用 ... 作为一个值。如果将宏定义为整数值，则该值将通过库对象的属性提供。通过编写声明 static const int FOO; 可以实现相同的效果。后者更通用，因为它支持除整数类型之外的其他类型 (注意: 然后用 C 语言语法将 const 与变量名一起写入，如 static char *const FOO;)。

目前，不支持自动查找在哪个地方需要的各种整数或浮点类型，除了以下情况：如果这样的类型是显式命名的。对于整数类型，请使用 typedef int... the_type_name; 或其他类型，如 typedef unsigned long... the_type_name;。两者都是等价的，并且由真实的 C 语言类型取代，它必须是整数类型。同样，对于浮点类型，请使 typedef float... the_type_name; 或等效的 typedef double... the_type_name;。请注意，这种方式无法检测到 long double。

在函数参数或返回类型的情况下，当它是一个简单的整数/浮点类型时，你可以简单地误判它。如果你将函数 void f(long) 误认为是 void f(int)，它仍然有效 (但你必须使用适合 int 的参数调用它)。它的工作原理是因为 C 编译器会为我们进行转换。此参数和返回类型的 C 语言级转换仅适用于常规函数，而不适用于函数指针类型; 目前，它也不适用于可变函数。

对于更复杂的类型，您别无选择，只能精确。例如，你不能将 int * 参数误认为是 long *，或者是全局数组 int a[5]; 误认为 long a[5];。CFFI 认为上面列出的所有类型 视为原始类型 (所以 long long a[5]; 和 int64_t a[5] 是不同的声明)。其中的原因在 关于问题的讨论 中有详细说明。

8.5 ffibuilder.compile() 等: 编译 out-of-line 模块

您可以使用以下某个函数来实际生成使用 ffibuilder.set_source() 和 ffibuilder.cdef() 编写的.py 或.c 文件。

请注意，这些函数不会覆盖具有完全相同内容的.py/.c 文件，以保留最后修改时间。在某些情况下，无论如何都需要更新最后修改时间，请在调用函数之前删除该文件。

版本 1.8 中的新功能: emit_c_code() 或 compile() 生成的 C 语言代码包含 #define Py_LIMITED_API。这意味着在 CPython >= 3.2 时，编译此源会生成二进制.so/.dll，它应该适用于任何 CPython >= 3.2 的版本 (而不是仅适用于相同版本的 CPython x.y)。但是，标准的 distutils 包仍会产生一个名为 NAME.cpython-35m-x86_64-linux-gnu.so 的文件。您可以手动将其重命名为 NAME.abi3.so，或使用 setuptools 版本 26 或更高版本。另请注意，使用 Python 的调试版本进行编译实际上并不会定义 Py_LIMITED_API，因为这样做会使 Python.h 不适当。

版本 1.12 中的新功能: Py_LIMITED_API 现在也在 Windows 上定义。如果你使用 virtualenv，你需要它的最新版本: 早于 16.0.0 的版本不用将 python3.dll 复制到虚拟环境中。如果升级 virtualenv 是一个真正的问题，您可以手动编辑 C 代码以删除第一行 # define Py_LIMITED_API。

ffibuilder.compile(tmpdir='.', verbose=False, debug=None): 显式生成.py 或.c 文件，并 (假如是.c) 编译它。输出文件是（或者是）放在 tmpdir 给出的目录中。在这里给出的示例中，我们在构建脚本中使用 if __name__ == "__main__": ffibuilder.compile()，如果直接执行它们，这会使它们重建当前目录中的.py/.c 文件。(注意: 如果在调用 set_source() 时指定了包，则使用 tmpdir 相应子目录。)

版本 1.4 中的新功能: verbose 参数。如果为 True，则打印通常的 distutils 输出，包括调用编译器的命令行。(在将来的版本中，此参数可能默认更改为 True。)

版本 1.8.1 中的新功能: debug 参数。如果设置为 bool，它将控制是否以调试模式编译 C 代码。默认的 None 表示使用本机 Python 的 sys.flags.debug。从版本 1.8.1 开始，如果您运行的是调试模式 Python，则默认情况下会以调试模式编译 C 代码 (请注意，无论如何必须在 Windows 上执行此操作)。

ffibuilder.emit_python_code(filename): 生成给定的.py 文件 (与用于 ABI 模式的 ffibuilder.compile() 相同，具有要写入的显式命名文件)。如果您愿意，可以将此.py 文件包含在您自己的发行版中: 对于任何 Python 版本 (2 或 3) 都是相同的。

ffibuilder.emit_c_code(filename): 生成给定的.c 文件 (用于 API 模式) 而不编译它。如果您有其他方法来编译它，则可以使用，例如如果您想与一些更大的构建系统集成，这些系统将为您编译这个文件。您还可以分发.c 文件: 除非您使用的构建脚本依赖于操作系统或平台，否则.c 文件本身是通用的 (如果在不同的操作系统上生成，使用不同版本的 CPython 或使用 PyPy，它将完全相同; 它是通过生成适当的 #ifdef 来完成的)。

ffibuilder.distutils_extension(tmpdir='build', verbose=True): 用于基于 distutils 的 setup.py 文件。如果需要，在给定的 tmpdir 中调用会创建.c 文件，并返回 distutils.core.Extension 实例。

对于 Setuptools，在 setup.py 中使用 cffi_modules=["path/to/foo_build.py:ffibuilder"] 行代替。这行要求 Setuptools 导入并使用 CFFI 提供的帮助程序，CFFI 执行文件 path/to/foo_build.py (与 execfile() 一样)，并查找名为 ffibuilder 的全局变量。你也可以这样 cffi_modules=["path/to/foo_build.py:maker"]，其中 maker 命名为全局函数; 调用它时没有参数，应该返回一个 FFI 对象。

8.6 ffi/ffibuilder.include(): 合并多个 CFFI 接口

ffi/ffibuilder.include(other_ffi): 包括在另一个 FFI 实例中定义的类型定义 (typedef)，结构体 (struct)，联合 (union)，枚举 (enum) 和常量 (const)。这适用于大型项目，其中一个基于 CFFI 的接口依赖于在不同的基于 CFFI 的接口中声明的某些类型。

*请注意，每个库只应使用一个 ffi 对象; ffi.include() 的预期用法是要与几个相互依赖的库进行交互。*对于一个库，请创建一个 ffi 对象。(如果一个文件太大，你可以从几个 Python 文件中通过相同的 ffi 编写多个 cdef() 调用。)

对于 out-of-line 模块，ffibuilder.include(other_ffibuilder) 行应该出现在构建脚本中，而 other_ffibuilder 参数应该是来自另一个构建脚本的另一个 FFI 实例。当两个构建脚本转换为生成的文件时，比如 _ffi.so 和 _other_ffi.so，然后导入 _ffi.so 在内部导致 _other_ffi.so 被导入。此时，_other_ffi.so 中的实际声明与 _ffi.so 中的实际声明相结合。

ffi.include() 的用法是 C 语言中 #include 的 cdef 级别等价物，其中程序的一部分可能包含在另一部分中为其自身使用而定义的类型和函数。您可以在 ffi 对象 (以及 包含支持的关联 lib 对象) 上看到包含支持声明的类型和常量。在 API 模式下，您还可以直接查看函数和全局变量。在 ABI 模式下，必须通过 other_ffi 上的 dlopen() 方法返回的原始 other_lib 对象访问这些对象。

8.7 ffi.cdef() 限制

cdef() 和一些 C99 应该支持所有的 ANSI C 声明。(这不包括任何 #include 或 #ifdef。) 已知缺少的功能是 C99，或 GCC 或 MSVC 扩展:

- 任何 __attribute__ 或 #pragma pack(n)
- 其他类型: 特殊大小的浮点和定点类型，向量类型等。
- 自版本 1.11 起 cffi 支持 C99 类型 float _Complex 和 double _Complex，但不支持 libffi: 您不能使用复杂参数或返回值调用 C 函数，除非它们是直接 API 模式函数。完全不支持 long double _Complex 类型 (声明并使用它，好像它是一个包含两个 long double 的数组，并在 C 语言中使用 set_source() 编写包装函数).
- __restrict__ 或 __restrict 分别是 GCC 和 MSVC 的扩展。他们不被认可。但是 restrict 是一个 C 关键字并被接受 (和忽略)。

注意像 int field[]; 结构中的结构被解释为可变长度结构。另一方面，像 int field[...]; 这样的声明是数组，其长度将由编译器完成。你可以使用 int field[]; 对于实际上不是可变长度的数组字段; 它也可以工作，但在这种情况下，由于 CFFI 认为它无法向 C 编译器询问数组的长度，因此可以减少安全检查：例如，您可能会通过在构造函数中传递过多的数组项来覆盖以下字段。

版本 1.2 中的新功能: 可以访问线程局部变量 (__thread)，以及定义为动态宏的变量 (#define myvar (*fetchme()))。在 1.2 版之前，您需要编写 getter/setter 函数。

请注意，如果在不使用 const 的情况下在 cdef() 中声明变量，CFFI 会假定它是一个读写变量并生成两段代码，一段用于读取，一段用于写入。如果变量实际上不能用 C 代码写入，由于某种原因，它将无法编译。在这种情况下，您可以将其声明为常量: 例如，你会使用 foo_t *const myglob; 而不是 foo_t *myglob;。另请注意 const foo_t *myglob; 是一个 *变量;* 它包含一个指向常量 foo_t 的变量指针。

8.8 调试 dlopenC 库

在 dlopen() 设置中，一些 C 库实际上很难正确使用。这是因为大多数 C 语言库都是针对它们与另一个程序 *链接*的情况下使用静态链接或动态链接进行测试，但是从 C 语言编写的程序，在启动时，使用链接器的功能而不是 dlopen()。

这有时可能会产生问题。你会在另一个设置中遇到与 CFFI 相同的问题，比如使用 ctypes 甚至是调用 dlopen() 的普通 C 代码。本节包含一些通常有用的环境变量 (在 Linux 上)，可以在调试这些问题时提供帮助。

export LD_TRACE_LOADED_OBJECTS=all

> 提供了很多信息，有时大多取决于设置。输出有关动态链接器的详细调试信息。如果设置为 all，则打印它具有的所有调试信息，如果设置为 help 打印有关可在此环境变量中指定哪些类别的帮助消息

export LD_VERBOSE=1

(glibc 自 2.1 起) 如果设置为非空字符串，则在查询有关程序的信息时输出有关程序的符号版本控制信息 (即，已设置 LD_TRACE_LOADED_OBJECTS，或者已为动态链接程序提供了 --list 或 --verify 选项)。

export LD_WARN=1

(仅限 ELF)(glibc 自 2.1.3 起) 如果设置为非空字符串，则警告未解析的符号。

8.9 ffi.verify(): in-line API 模式

ffi.verify() 支持向后兼容，但已弃用。ffi.verify(c_header_source, tmpdir=.., ext_package=.., modulename=.., flags=.., **kwargs) 从 ffi.cdef() 生成并编译 C 语言文件，如 ffi.set_source() 在 API 模式下，然后立即加载并返回动态库对象。使用一些重要的逻辑来决定是否必须重新编译动态库; 请参阅以下有关控制它的方法。

c_header_source 和额外关键字参数的含义与 ffi.set_source() 中的含义相同。

ffi.verify() 的一个剩余用例将以下修改以明确查找任何类型的大小，以字节为单位，并立即在 Python 中使用它 (例如因为需要编写构建脚本的其余部分):

```
ffi = cffi.FFI()
ffi.cdef("const int mysize;")
lib = ffi.verify("const int mysize = sizeof(THE_TYPE);")
print lib.mysize
```

ffi.verify() 的额外参数:

- tmpdir 控制 C 语言文件的创建和编译位置。除非设置了 CFFI_TMPDIR 环境变量，默认值为 directory_containing_the_py_file/__pycache__ 使用.py 文件的目录名，该文件包含对 ffi.verify() 的实际调用。(这有一点修改，但通常与您的库的.pyc 文件的位置一致。名称 __pycache__ 本身来自 Python 3。)
- ext_package 控制应该从哪个包中查找编译的扩展模块。这仅在分发基于 ffi.verify() 的模块后才有用。
- tag 参数在扩展模块的名称中间插入一个额外的字符串: _cffi_<tag>_<hash>。有用的是提供更多的上下文，例如调试时。
- modulename 参数可用于强制特定模块名称，覆盖名称 _cffi_<tag>_<hash>。小心使用，例如如果要将变量信息传递给 verify() 但仍希望模块名称始终相同 (例如本地文件的绝对路径)。在这种情况下，不计算散列，如果模块名称已经存在，则无需进一步检查即可重复使用。每当您更改源时，请务必使用其他方法清除 tmpdir。
- source_extension 与 ffibuilder.set_source() 中的含义相同。
- 可选的 flags 参数 (在 Windows 上被忽略) 默认为 ffi.RTLD_NOW; 参见 man dlopen。(使用 ffibuilder.set_source()，您将使用 sys.setdlopenflags()。)
- 如果需要列出传递给 C 编译器的本地文件，则可选的 relative_to 参数很有用:

```
ext = ffi.verify(..., sources=['foo.c'], relative_to=__file__)
```

与上行大致相同的:

```
ext = ffi.verify(..., sources=['/path/to/this/file/foo.c'])
```

除了生成的库的默认名称是根据参数 sources 的 CRC 校验和和构建的，以及您给 ffi.verify() 的大多数其他参数，但不是 relative_to。因此，如果您使用第二行，它将在您的项目安装后停止查 找已编译的库，因为 '/path/to/this/file' 突然改变了。第一行没有这个问题。

请注意，在开发期间，每次更改传递给 cdef() 或 verify() 的 C 源时，后者将根据从这些字符串计算的两个 CRC32 哈希值创建新的模块文件名。这会在 __pycache__ 目录中创建越来越多的文件。建议您不时的清理它。一个很好的方法是在测试套件中添加对 cffi.verifier.cleanup_tmpdir() 的调用。或者，您可以手动删除整个 __pycache__ 目录。

另一个缓存目录可以作为 verify() 的 tmpdir 参数，通过环境变量 CFFI_TMPDIR，或者在调用 verify 之前调用 cffi.verifier.set_tmpdir(path)。。

8.10 从 CFFI 0.9 升级到 CFFI 1.0

CFFI 1.0 是向后兼容的，但考虑转向 1.0 中的新 out-of-line 方法仍然是一个好主意。这是步骤。

ABI 模式，如果您的 CFFI 项目使用 ffi.dlopen():

```
import cffi

ffi = cffi.FFI()
ffi.cdef("stuff")
lib = ffi.dlopen("libpath")
```

如果"stuff" 部分足够大以至于导入时间是一个问题，那么按照*out-of-line 但仍然是 ABI 模式* 的描述重写它。可选，另请参见*setuptools 集成* 段落。

API 模式，如果您的 CFFI 项目使用 ffi.verify():

```
import cffi

ffi = cffi.FFI()
ffi.cdef("stuff")
lib = ffi.verify("real C code")
```

然后你应该按照上面的*out-of-line, API 模式* 中的描述重写它。它避免了一些导致 ffi.verify() 随着时间推移增加一些额外参数的问题。然后查看*distutils 或 setuptools* 段落。另外，请记住从 setup.py 中删除 ext_package=".."，这有时需要使用 verify() 但只是与 set_source() 产生混淆。

以下示例应该适用于旧版本 (1.0 之前版本) 和新版本的 CFFI 版本，支持这两个版本在旧版本的 PyPy 上运行非常重要 (CFFI 1.0 在 PyPy < 2.6 中不起作用):

```
# in a separate file "package/foo_build.py"
import cffi

ffi = cffi.FFI()
C_HEADER_SRC = r'''
    #include "somelib.h"
'''
C_KEYWORDS = dict(libraries=['somelib'])

if hasattr(ffi, 'set_source'):
    ffi.set_source("package._foo", C_HEADER_SRC, **C_KEYWORDS)

ffi.cdef('''
    int foo(int);
''')

if __name__ == "__main__":
    ffi.compile()
```

并在主程序中:

```
try:
    from package._foo import ffi, lib
except ImportError:
    from package.foo_build import ffi, C_HEADER_SRC, C_KEYWORDS
    lib = ffi.verify(C_HEADER_SRC, **C_KEYWORDS)
```

(不论好坏，这个最新技巧可以更普遍地用于允许导入" 执行"，即使没有生成 `_foo` 模块。)

编写一个兼容 CFFI 0.9 和 1.0 的 `setup.py` 脚本需要显式检查我们可以拥有的 CFFI 版本，它被硬编码为 PyPy 中的内置模块:

```
if '_cffi_backend' in sys.builtin_module_names:   # PyPy
    import _cffi_backend
    requires_cffi = "cffi==" + _cffi_backend.__version__
else:
    requires_cffi = "cffi>=1.0.0"
```

然后我们使用 `requires_cffi` 变量根据需要为 `setup()` 提供不同的参数，例如:

```
if requires_cffi.startswith("cffi==0."):
    # backward compatibility: we have "cffi==0.*"
    from package.foo_build import ffi
    extra_args = dict(
        ext_modules=[ffi.verifier.get_extension()],
        ext_package="...",    # if needed
    )
else:
    extra_args = dict(
        setup_requires=[requires_cffi],
        cffi_modules=['package/foo_build.py:ffi'],
    )
setup(
    name=...,
    ...,
    install_requires=[requires_cffi],
    **extra_args
)
```

CHAPTER 9

使用 CFFI 进行嵌入

Contents

您可以使用 CFFI 生成 C 代码，该代码将您选择的 API 导出到任何想要与此 C 代码链接的 C 应用程序。您自己定义的此 API 最终将作为 `.so/.dll/.dylib` 库的 API，或者您可以在较大的应用程序中静态链接它。

可能的用例:

- 将用 Python 编写的库直接显露给 C/C++ 程序。
- 使用 Python 为已经编写的现有 C/C++ 程序制作” 插件” 以加载它们。
- 使用 Python 实现更大的 C/C++ 应用程序的一部分 (使用静态链接)。

- 在 Python 中编写一个小的 C/C++ 包装器，隐藏了应用程序实际上是用 Python 编写的事实 (创建自定义命令行界面; 用于分发目的; 或者只是简单地对应用程序进行逆向工程).

总体思路如下:

- 您编写并执行 Python 脚本，该脚本生成带有您选择的 API 的 `.c` 文件 (并可选择将其编译为 `.so/.dll/.dylib`)。该脚本还提供了一些 Python 代码在 `.so` 中" 封装"。
- 在运行时, C 应用程序加载此 `.so/.dll/.dylib` (或与 `.c` 源代码静态链接), 而不必知道它是从 Python 和 CFFI 生成的。
- 第一次调用 C 函数时，Python 被初始化并执行封装的 Python 代码。
- 封装的 Python 代码定义了更多实现 API 的 C 函数的 Python 函数，然后用于所有后续的 C 函数调用。

这种方法的目标之一是完全独立于 CPython C API: 没有 `Py_Initialize()` 和 `PyRun_SimpleString()` 甚至没有 `PyObject`。它在 CPython 和 PyPy 上的工作方式相同。

这完全是 *版本 1.5 中的新功能*。 (PyPy 包含自 5.0 版以来的 CFFI 1.5。)

9.1 用法

有关快速介绍，请参阅 概述页面中的段落 。在本节中，我们将更详细地解释每一步。我们将在这里使用这个稍微扩展的例子:

```
/* file plugin.h */
typedef struct { int x, y; } point_t;
extern int do_stuff(point_t *);
```

```
/* file plugin.h, Windows-friendly version */
typedef struct { int x, y; } point_t;

/* When including this file from ffibuilder.set_source(), the
   following macro is defined to '__declspec(dllexport)'.  When
   including this file directly from your C program, we define
   it to 'extern __declspec(dllimport)' instead.

   With non-MSVC compilers we simply define it to 'extern'.
   (The 'extern' is needed for sharing global variables;
   functions would be fine without it.  The macros always
   include 'extern': you must not repeat it when using the
   macros later.)
*/
```

(下页继续)

(续上页)

```
#ifndef CFFI_DLLEXPORT
#  if defined(_MSC_VER)
#    define CFFI_DLLEXPORT  extern __declspec(dllimport)
#  else
#    define CFFI_DLLEXPORT  extern
#  endif
#endif

CFFI_DLLEXPORT int do_stuff(point_t *);
```

```
# file plugin_build.py
import cffi
ffibuilder = cffi.FFI()

with open('plugin.h') as f:
    # read plugin.h and pass it to embedding_api(), manually
    # removing the '#' directives and the CFFI_DLLEXPORT
    data = ''.join([line for line in f if not line.startswith('#')])
    data = data.replace('CFFI_DLLEXPORT', '')
    ffibuilder.embedding_api(data)

ffibuilder.set_source("my_plugin", r'''
    #include "plugin.h"
''')

ffibuilder.embedding_init_code("""
    from my_plugin import ffi

    @ffi.def_extern()
    def do_stuff(p):
        print("adding %d and %d" % (p.x, p.y))
        return p.x + p.y
""")

ffibuilder.compile(target="plugin-1.5.*", verbose=True)
# or: ffibuilder.emit_c_code("my_plugin.c")
```

运行上面的代码会生成一个 *DLL*, 即, 一个可动态加载的库。它是 Windows 上的扩展名为 `.dll`, Mac OS/X 上的 `.dylib` 或其他平台上的 `.so` 文件。像往常一样, 它是通过生成一些中间 `.c` 代码然后调用常规平台特定的 C 编译器来生成的。有关使用生成的库的 C 语言级别问题的一些指示, 请参见下文。

以下是有关上述方法的一些细节:

- **ffibuilder.embedding_api(source):** 解析给定的 C 源，它声明了您希望由 DLL 导出的函数。它还可以声明类型，常量和全局变量，它们是 DLL 的 C 语言级别 API 的一部分。

  在 source 文件中找到的函数将在 .c 文件中自动定义: 它们将包含在第一次调用 Python 解释器时初始化 Python 解释器的代码，然后是调用附加的 Python 函数的代码 (使用 @ffi.def_extern()，请参阅下一点)。

  另一方面，全局变量不会自动生成。您必须在 ffibuilder.set_source() 中显式地编写它们的定义，作为常规 C 代码 (参见接下来的一点)。

- **ffibuilder.embedding_init_code(python_code):** 这给出了初始化时间 Python 源代码。此代码在 DLL 中被复制 (" 封装")。在运行时，代码在首次初始化 DLL 时执行，就在 Python 本身初始化之后。这个新初始化的 Python 解释器有一个额外的" 内置" 模块，可以神奇地加载而无需访问任何文件，使用类似"from my_plugin import ffi, lib" 的一行。名称 my_plugin 来自 ffibuilder.set_source() 的第一个参数。从 Python 的角度来看，这个模块代表了 "调用者的 C 语言世界"。

  初始化时间 Python 代码可以像往常一样导入其他模块或包。您可能会遇到典型的 Python 问题，例如需要先手动设置 sys.path。

  对于 ffibuilder.embedding_api() 中声明的每个函数，初始化时间 Python 代码或其导入的模块之一应使用装饰器 @ffi.def_extern() 将相应的 Python 函数附加到它。

  如果初始化时间 Python 代码因异常而失败，那么您将获得打印到 stderr 的 traceback 以及更多信息，以帮助您识别错误的 sys.path 等问题。如果某个函数在 C 代码尝试调用它时仍未附加，则还会向 stderr 打印一条错误消息，该函数返回零/null。

  请注意，CFFI 模块从不调用 exit()，但 CPython 本身包含调用 exit() 的代码，例如，如果导入 site 失败。这可能会在将来解决。

- **ffibuilder.set_source(c_module_name, c_code):** 从 Python 的角度设置模块的名称。它还提供了更多的 C 代码，这些代码将包含在生成的 C 代码中。在简单的例子中，它可以是一个空字符串。您可以在其中 #include 其他一些文件，定义全局变量等。宏 CFFI_DLLEXPORT 可用于此 C 语言代码: 它扩展到特定于平台的方式表示" 应该从 DLL 导出以下声明"。例如，您可以将"extern int my_glob; " 放在 ffibuilder.embedding_api() 和"CFFI_DLLEXPORT int my_glob = 42;" 放在 ffibuilder.set_source() 中。

  目前，ffibuilder.embedding_api() 中声明的任何类型也必须存在于 c_code 中。如果此代码在上面的示例中包含类似 #include "plugin.h" 的行，则这是自动的。

- **ffibuilder.compile([target=...] [, verbose=True]):** 制作 C 代码并编译它。默认情况下，它会生成一个名为 c_module_name.dll，c_module_name.dylib 或 c_module_name.so 的文件，但可以使用可选的 target 关键字参数更改默认值。你可以使用带有文字 * target="foo.*" 在 Windows 上请求一个名为 foo.dll 的文件，在 OS/X 上请求 foo.dylib ，在其他地方使用 foo.so。指定备用目标的一个原因是包括 Python 模块名称中通常不允许的字符，例如"plugin-1.5.*"。

  对于更复杂的情况，您可以调用 ffibuilder.emit_c_code("foo.c") 并使用其他方法编译生成的 foo.c 文件。CFFI 的编译逻辑基于标准库 distutils 包，它是为了制作 CPython 扩展模块而开

发和测试的; 它可能并不总是适合制作通用 DLL。此外，如果您不想制作独立的 .so/.dll/.dylib 文件，只需获取 C 代码即可: 这个 C 文件可以作为更大的应用程序的一部分进行编译和静态链接。

9.2 阅读更多

如果您正在阅读有关嵌入的此页面，并且您已经不熟悉 CFFI，请参阅下面的内容:

- 对于 @ffi.def_extern() 函数，整数 C 类型只是作为 Python 整数传递; 简单的指向结构和基本数组的指针都很简单。但是，迟早您需要在 此处 详细了解此主题。
- @ffi.def_extern(): 请参阅 此处的文档, 特别是如果 Python 函数引发异常会发生什么。
- 要创建附加到 C 数据的 Python 对象，一种常见的解决方案是使用 ffi.new_handle()。请参阅 此处 的文档。
- 在嵌入模式中，主要方向是调用 Python 函数的 C 代码。这与 CFFI 的常规扩展模式相反，其中主要方向是调用 C 的 Python 代码。这就是为什么页面 使用 ffi/lib 对象 首先讨论后者，以及为什么"C 代码调用 Python" 的方向通常在该页面中被称为" 回调"。如果您还需要让 Python 代码调用 C 代码，请阅读下面有关嵌入和扩展 的更多信息。
- ffibuilder.embedding_api(source): 遵循与 ffibuilder.cdef() 相同的语法，文档在此。 您也可以使用"..." 语法，但在实践中它可能没有 cdef() 那么有用。另一方面，预计通常需要提供给 ffibuilder.embedding_api() 的 C 语言 source 与您希望提供给 DLL 用户的某些 .h 文件的内容完全相同。这就是上面的例子这样做的原因:

```
with open('foo.h') as f:
    ffibuilder.embedding_api(f.read())
```

请注意，这种方法的缺点是 ffibuilder.embedding_api() 不支持 #ifdef 指令。你可能不得不使用更复杂的表达式:

```
with open('foo.h') as f:
    lines = [line for line in f if not line.startswith('#')]
    ffibuilder.embedding_api(''.join(lines))
```

如上例所示，您也可以使用 ffibuilder.set_source() 中的相同 foo.h:

```
ffibuilder.set_source('module_name', r'''
    #include "foo.h"
''')
```

9.3 疑难解答

- 错误消息

cffi extension module 'c_module_name' has unknown version 0x2701

表示正在运行的 Python 解释器位于早于 1.5 的 CFFI 版本。必须在正在运行的 Python 中安装 CFFI 1.5 或更高版本。

- 在 PyPy 上，错误消息

  debug: pypy_setup_home: directories 'lib-python' and 'lib_pypy' not found in pypy's shared library location or in any parent directory

  表示找到了 `libpypy-c.so` 文件，但未在此位置找到标准库。至少在某些 Linux 发行版中会出现这种情况，因为它们将 `libpypy-c.so` 放在 `/usr/lib/`, 中，而不是我们推荐的方式，这是: 将该文件保存在 `/opt/pypy/bin/` 中，并在 `/usr/lib/` 中添加符号链接。最快的解决方法是手动进行更改。

9.4 关于使用.so 的问题

本段描述的问题不一定是 CFFI 特有的。它假定您已经获得了如上所述的 `.so/.dylib/.dll` 文件，但是您在使用它时遇到了麻烦。(总之: 这是一团糟。这是我自己的经验，通过使用 Google 和查看来自各种平台的报告。请报告本段中的任何不准确之处或更好的方法。)

- CFFI 生成的文件应遵循此命名模式: Linux 上的 `libmy_plugin.so`，Mac 上的 `libmy_plugin.dylib` 或 Windows 上的 `my_plugin.dll` (Windows 上没有 `lib` 前缀)。
- 首先请注意，此文件不包含 Python 解释器，也不包含 Python 的标准库。你仍然需要它在某个地方。有一些方法可以将它压缩为较少数量的文件，但这超出了 CFFI 的范围 (请报告您是否成功使用了其中一些方法，以便我可以在此处添加一些链接).
- 在我们称之为" 主程序" 的地方，`.so` 可以动态使用 (例如通过在主程序中调用 `dlopen()` 或 `LoadLibrary()`)，也可以在编译时使用 (例如通过用 `gcc -lmy_plugin` 编译它)。如果您正在为程序构建插件，则始终使用前一种情况，并且程序本身不需要重新编译。后一种情况是为了使 CFFI 库更紧密地集成在主程序中。
- 在编译时使用的情况下: 你可以在 `-Lsome/path/` 之前添加 gcc 选项 `-lmy_plugin` 来描述 `libmy_plugin.so` 的位置。在某些平台上，特别是 Linux，如果能找到 `libmy_plugin.so` 而不是 `libpython27.so` 或 `libpypy-c.so`，`gcc` 会报错。要修复它，您需要调用 `LD_LIBRARY_PATH=/some/path/to/libpypy gcc`。
- 实际执行主程序时，需要找到 `libmy_plugin.so` 以及 `libpython27.so` 或 `libpypy-c.so`。对于 PyPy，解压缩 PyPy 发行版，并在 `bin` 子目录中获得 `libpypy-c.so` 的完整目录结构，或者在顶级目录中的 Windows `pypy-c.dll` 上获取完整目录结构; 你不能移动这个文件，只是指向它。指向它的一种方法是使用一些环境变量运行主程序: Linux 上的 `LD_LIBRARY_PATH=/some/path/to/libpypy`，OS/X 上的 `DYLD_LIBRARY_PATH=/some/path/to/libpypy`。
- 如果使用内部硬编码的路径编译 `libmy_plugin.so`，则可以避免 `LD_LIBRARY_PATH` 问题。在 Linux 中，这是由 `gcc -Wl,-rpath=/some/path` 完成的。你可以把这个选项放在 `ffibuilder.set_source("my_plugin", ..., extra_link_args=['-Wl,-rpath=/some/path/to/libpypy'])` 中。该路径可以以 `$ORIGIN` 开头，表示"`libmy_plugin.so` 所在的目录"。然后，您可以指定相

对于该位置的路径，例如 `extra_link_args=['-Wl,-rpath=$ORIGIN/../venv/bin']`。使用 `ldd libmy_plugin.so` 查看 `$ORIGIN` 扩展后当前编译的路径。）

在此之后，您不再需要 `LD_LIBRARY_PATH` 来在运行时找到 `libpython27.so` 或 `libpypy-c.so`。从理论上讲，它还应该包括对主要程序的 `gcc` 调用。如果 rpath 以 `$ORIGIN` 开头，我在 Linux 上没有 `LD_LIBRARY_PATH` 就无法很好的使用 `gcc`

- 可以使用相同的 rpath 技巧让主程序在没有 `LD_LIBRARY_PATH` 的情况下首先找到 `libmy_plugin.so`. (如果主程序使用 `dlopen()` 将其作为动态插件加载，则不适用。) 您可以使用 `gcc -Wl,-rpath=/path/to/libmyplugin` 创建主程序，可能使用 `$ORIGIN`。`$ORIGIN` 中的 `$` 会导致各种 shell 问题: 如果使用通用 shell，则需要说明 `gcc -Wl,-rpath=\$ORIGIN`。从 Makefile 中，你需要说明一些类似 `gcc -Wl,-rpath=\$$ORIGIN` 的语句。
- 在某些 Linux 发行版上，特别是 Debian，CPython C 扩展模块的 `.so` 文件可能会被编译而不会说明它们依赖于 `libpythonX.Y.so`。如果嵌入器使用 `dlopen(..., RTLD_LOCAL)` 这使得这样的 Python 系统不适合嵌入。您得到一个 `undefined symbol` 错误。参见 问题 #264。解决方法是首先调用 `dlopen("libpythonX.Y.so", RTLD_LAZY|RTLD_GLOBAL)`，这将强制首先加载 `libpythonX.Y.so`。

9.5 使用多个 CFFI 制作的 DLL

多个 CFFI 制作的 DLL 可以由相同的过程使用。

请注意，进程中所有 CFFI 制作的 DLL 共享一个 Python 解释器。这种效果与通过组装大量不相关的包来构建大型 Python 应用程序所获得的效果相同。其中一些可能是从标准库中修补某些函数的库，例如，其他部分可能出乎意料。

9.6 多线程

基于 Python 的标准全局解释器锁 (Global Interpreter Lock)，多线程应该透明地工作。

如果两个线程在 Python 尚未初始化时都尝试调用 C 函数，则会发生死锁。一个线程继续初始化并阻塞另一个线程。只有在执行初始化时间 Python 代码时才允许另一个线程继续。

如果两个线程调用两个不同的 CFFI 制造的 DLL，Python 初始化本身仍将被序列化，但两段初始化时间的 Python 代码不会。其思想是，事先没有理由让一个 DLL 等待另一个 DLL 的初始化完成。

初始化之后，Python 的标准全局解释器锁启动。最终结果是当一个 CPU 在执行 Python 代码时，没有其他 CPU 可以从同一进程的另一个线程执行更多 Python 代码。每隔一段时间，锁会切换到一个不同的线程，这样就不会出现任何单个线程无限期阻塞。

9.7 测试

出于测试目的，可以在正在运行的 Python 解释器中导入 CFFI 制造的 DLL，而不是像 C 共享库一样加载。

您可能在文件名方面存在一些问题: 例如, 在 Windows 上, Python 期望的文件被称为 `c_module_name.pyd`, 但 CFFI 制造的 DLL 被称为 `target.dll`。基本名称 `target` 是 `ffibuilder.compile()` 中指定的目标, 在 Windows 上, 扩展名为 `.dll` 而不是 `.pyd`。您必须重命名或复制文件, 或者在 POSIX 上使用符号链接。

然后该模块就像常规的 CFFI 扩展模块一样工作。它使用"`from c_module_name import ffi, lib`" 导入, 并在 `lib` 对象上公开所有 C 函数。您可以通过调用这些 C 函数来测试它。DLL 内部封装的初始化时间 Python 代码在第一次完成此类调用时执行。

9.8 嵌入和扩展

嵌入模式与 CFFI 的非嵌入模式不兼容。

您可以在同一构建脚本中同时使用 `ffibuilder.embedding_api()` 和 `ffibuilder.cdef()`。你把前面想要由 DLL 导出的声明放在前面; 你只需要在 C 和 Python 之间共享 C 函数和类型, 而不是从 DLL 中导出。

作为一个例子, 考虑你希望直接用 C 语言编写 DLL 导出的 C 函数的情况, 也许在调用 Python 函数之前处理一些情况。为此, 您不能将函数的签名放在 `ffibuilder.embedding_api()`。(请注意, 如果您使用 `ffibuilder.embedding_api(f.read())` 则需要更多修改。) 您只能在 `ffibuilder.set_source()` 中编写自定义函数定义, 并使用宏 CFFI_DLLEXPORT 作为前缀:

```
CFFI_DLLEXPORT int myfunc(int a, int b)
{
    /* implementation here */
}
```

如果需要, 这个函数可以使用" 回调" 的一般机制调用 Python 函数, 这是因为它是从 C 到 Python 的调用, 尽管在这种情况下它不会调用任何东西:

```
ffibuilder.cdef("""
    extern "Python" int mycb(int);
""")

ffibuilder.set_source("my_plugin", r"""

    static int mycb(int);   /* the callback: forward declaration, to make
                               it accessible from the C code that follows */

    CFFI_DLLEXPORT int myfunc(int a, int b)
    {
        int product = a * b;   /* some custom C code */
        return mycb(product);
    }
""")
```

然后 Python 初始化代码需要包含以下行:

```
@ffi.def_extern()
def mycb(x):
    print "hi, I'm called with x =", x
    return x * 10
```

这个 `@ffi.def_extern` 将一个 Python 函数附加到 C 回调 `mycb()`，在这种情况下，它不会从 DLL 导出。然而，当调用 `mycb()` 时，会发生 Python 的自动初始化，如果它恰好是从 C 调用的第一个函数。更确切地说，调用 `myfunc()` 时不会发生这种情况: 这只是一个 C 函数，没有额外的代码如魔法般地镶嵌在它周围。它只发生在 `myfunc()` 调用 `mycb()` 时。

如上面的解释提示，这就是 `ffibuilder.embedding_api()` 实际实现直接调用 Python 代码的函数调用的方式; 在这里，我们只是明确地分解它，以便在中间添加一些自定义 C 代码。

如果您需要强制从 C 代码中调用 Python，在调用第一个 `@ffi.def_extern()` 之前进行初始化，你可以通过调用没有参数的 C 函数 `cffi_start_python()` 来实现。它返回一个整数 0 或-1，以判断初始化是否成功。目前，无法阻止初始化失败，也无法将 traceback 和更多信息转储到 stderr。