
Argument Clinic 的用法

发布 3.12.0rc3

Guido van Rossum and the Python development team

十月 01, 2023

Python Software Foundation
Email: docs@python.org

Contents

1	背景	2
1.1	基本概念	2
2	参考	3
2.1	术语	3
2.2	命令行接口	3
2.3	用于扩展 Argument Clinic 的类	4
3	教程	4
4	常用方案指引	9
4.1	How to rename C functions and variables generated by Argument Clinic	9
4.2	如何转换使用 PyArg_UnpackTuple 的函数	10
4.3	如何使用可选分组	10
4.4	如何使用真正的 Argument Clinic 转换器，而不是“旧式转换器”	11
4.5	如何使用 Py_buffer 转换器	13
4.6	如何使用高级转换器	13
4.7	如何给形参赋默认值	14
4.8	如何使用返回转换器	15
4.9	如何克隆现有的函数	16
4.10	如何调用 Python 代码	16
4.11	如何使用“自转换器”	17
4.12	如何使用“定义类”转换器	17
4.13	如何编写自定义转换器	18
4.14	如何编写自定义返回转换器	19
4.15	如何转换 METH_O 和 METH_NOARGS 函数	19
4.16	如何转换 tp_new 和 tp_init 函数	19
4.17	如何修改和重定向 Clinic 的输出	19
4.18	如何使用 #ifdef 技巧	22
4.19	如何在 Python 文件中使用 Argument Clinic	23
	Python 模块索引	24
	索引	25

作者 Larry Hastings

源代码: [Tools/clinic/clinic.py](#)。

摘要

Argument Clinic 是一个针对 CPython C 文件的预处理器。它是根据 [PEP 436](#) 在 Python 3.4 中引入的，其目的是提供自省签名，并为 CPython 内置函数、模块级函数和类方法中的参数解析生成高性能的定制模板代码。本文档分为四个主要部分：

- [背景](#) 谈论有关 Argument Clinic 的基本概念和目标。
- [参考](#) 描述了命令行接口和 Argument Clinic 相关术语。
- [教程](#) 指导你熟悉将现有 C 函数适配至 Argument Clinic 所需的所有步骤。
- [常用方案指引](#) 详细介绍如何处理一些特定的任务。

备注：Argument Clinic 被视为仅供 CPython 内部使用。它不支持在 CPython 以外的文件中使用，也不保证未来版本会向下兼容。换句话说：如果维护的是 CPython 的外部 C 扩展，欢迎在你自己的代码中试用 Argument Clinic。但是 Argument Clinic 当前版本与新版 CPython 附带的版本可能完全不兼容并会破坏你的所有代码。

1 背景

1.1 基本概念

在文件上运行 Argument Clinic 时，无论是通过[命令行接口](#)还是通过 `make clinic`，它都会扫描输入文件来查找[开始行](#)：

```
/*[clinic input]
```

在找到之后，它会读取所有内容直到[end line](#)：

```
[clinic start generated code]*/
```

这两行之间的所有内容都是 Argument Clinic 的[input](#)。当 Argument Clinic 解析 input 时，会生成[output](#)。output 会紧接着输入被重新写入到 C 文件中，后面再加一个[checksum line](#)。所有这些行，包括[start line](#)和[checksum line](#)，被统称为 Argument Clinic 的[block](#)：

```
/*[clinic input]
... clinic input goes here ...
[clinic start generated code]*/
... clinic output goes here ...
/*[clinic end generated code: ...]*/
```

如果您在同一文件上第二次运行 Argument Clinic，Argument Clinic 将丢弃旧的[output](#)并写出新的 output 加一个新的[checksum line](#)。如果[input](#)没有改变，则输出也不会改变。

备注：你绝不应该修改 Argument Clinic 代码块的输出，因为任何修改都会导致 Argument Clinic 今后的运行中丢失；Argument Clinic 会检测到输出校验和不匹配并重新生成正确的输出。如果你对生成的输出结果不满意，你应当修改输入直到它产生你想要的输出。

2 参考

2.1 术语

开始行 `/*[clinic input]` 行。这一行标志着 Argument Clinic 输入的开始。请注意 开始行将开启一个 C 的块注释。

结束行 `[clinic start generated code]*/` 行。结束行标志着 Argument Clinic *input* 的 `_结束_`，但同时也标志着 Argument Clinic *output* 的 `_开始_`，因此文本 `"clinic start start generated code"` 表示结束行关闭了由 开始行所开启的 C 的块注释。

checksum 一个哈希值，用于区分唯一的输入 和输出。

校验行 A line that looks like `/*[clinic end generated code: ...]*/`. The three dots will be replaced by a *checksum* generated from the *input*, and a *checksum* generated from the *output*. The checksum line marks the end of Argument Clinic generated code, and is used by Argument Clinic to determine if it needs to regenerate output.

输入 The text between the *start line* and the *end line*. Note that the start and end lines open and close a C block comment; the *input* is thus a part of that same C block comment.

output The text between the *end line* and the *checksum line*.

block All text from the *start line* to the *checksum line* inclusively.

2.2 命令行接口

The Argument Clinic CLI (Command-Line Interface) is typically used to process a single source file, like this:

```
$ python3 ./Tools/clinic/clinic.py foo.c
```

The CLI supports the following options:

-h, --help

打印 CLI 使用说明。

-f, --force

强制输出重新生成。

-o, --output OUTPUT

将文件输出重定向到 OUTPUT

-v, --verbose

启用详细模式。

--converters

打印所有支持的转换器列表并返回这些转换器。

--make

遍历 `--srcdir` 处理所有相关文件。

--srcdir SRCDIR

在 `--make` 模式下要遍历的目录树。

FILE ...

要处理的文件列表。

2.3 用于扩展 Argument Clinic 的类

class clinic.CConverter

The base class for all converters. See [如何编写自定义转换器](#) for how to subclass this class.

type

The C type to use for this variable. `type` should be a Python string specifying the type, e.g. `'int'`. If this is a pointer type, the type string should end with `'*'`.

default

该参数的缺省值，为 Python 数据类型。若无缺省值，则为 `unspecified`。

py_default

default as it should appear in Python code, as a string. Or `None` if there is no default.

c_default

default as it should appear in C code, as a string. Or `None` if there is no default.

c_ignored_default

在无默认值时用于初始化 C 变量的默认值，不指定默认值可能会导致“变量未初始化”警告。这在使用选项组时很容易发生——尽管编写合理的代码将永远不会真的使用这个值，但该变量确实会被传入 `impl`，而 C 编译器将会报告“使用了”未初始化的值。这个值应当总是一个非空字符串。

converter

C 转换器的名称，字符串类型。

impl_by_reference

布尔值。如果为 `True`，则 Argument Clinic 在将变量传入 `impl` 函数时，会在其名称前加上一个 `&`。

parse_by_reference

一个布尔值。如果为真，则 Argument Clinic 在将其传入 `PyArg_ParseTuple()` 时将在变量名之前添加一个 `&`。

3 教程

The best way to get a sense of how Argument Clinic works is to convert a function to work with it. Here, then, are the bare minimum steps you'd need to follow to convert a function to work with Argument Clinic. Note that for code you plan to check in to CPython, you really should take the conversion farther, using some of the [advanced concepts](#) you'll see later on in the document, like [如何使用返回转换器](#) 和 [如何使用“自转换器”](#). But we'll keep it simple for this walkthrough so you can learn.

First, make sure you're working with a freshly updated checkout of the CPython trunk.

Next, find a Python builtin that calls either `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()`, and hasn't been converted to work with Argument Clinic yet. For this tutorial, we'll be using `_pickle.Pickler.dump`.

If the call to the `PyArg_Parse*()` function uses any of the following format units...:

```
O&
O!
es
es#
et
et#
```

... or if it has multiple calls to `PyArg_ParseTuple()`, you should choose a different function. (See [如何使用高级转换器](#) for those scenarios.)

Also, if the function has multiple calls to `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()` where it supports different types for the same argument, or if the function uses something besides `PyArg_Parse*` functions to parse its arguments, it probably isn't suitable for conversion to Argument Clinic. Argument Clinic doesn't support generic functions or polymorphic parameters.

Next, add the following boilerplate above the function, creating our input block:

```
/*[clinic input]
[clinic start generated code]*/
```

Cut the docstring and paste it in between the `[clinic]` lines, removing all the junk that makes it a properly quoted C string. When you're done you should have just the text, based at the left margin, with no line wider than 80 characters. Argument Clinic will preserve indents inside the docstring.

If the old docstring had a first line that looked like a function signature, throw that line away; The docstring doesn't need it anymore --- when you use `help()` on your builtin in the future, the first line will be built automatically based on the function's signature.

Example docstring summary line:

```
/*[clinic input]
Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

If your docstring doesn't have a "summary" line, Argument Clinic will complain, so let's make sure it has one. The "summary" line should be a paragraph consisting of a single 80-column line at the beginning of the docstring. (See [PEP 257](#) regarding docstring conventions.)

Our example docstring consists solely of a summary line, so the sample code doesn't have to change for this step.

Now, above the docstring, enter the name of the function, followed by a blank line. This should be the Python name of the function, and should be the full dotted path to the function --- it should start with the name of the module, include any sub-modules, and if the function is a method on a class it should include the class name too.

In our example, `_pickle` is the module, `Pickler` is the class, and `dump()` is the method, so the name becomes `_pickle.Pickler.dump()`:

```
/*[clinic input]
_pickle.Pickler.dump

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

If this is the first time that module or class has been used with Argument Clinic in this C file, you must declare the module and/or class. Proper Argument Clinic hygiene prefers declaring these in a separate block somewhere near the top of the C file, in the same way that include files and statics go at the top. In our sample code we'll just show the two blocks next to each other.

类和模块的名称应与暴露给 Python 的相同。请适时检查 `PyModuleDef` 或 `PyTypeObject` 中定义的名称。

When you declare a class, you must also specify two aspects of its type in C: the type declaration you'd use for a pointer to an instance of this class, and a pointer to the `PyTypeObject` for this class:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

Declare each of the parameters to the function. Each parameter should get its own line. All the parameter lines should be indented from the function name and the docstring. The general form of these parameter lines is as follows:

```
name_of_parameter: converter
```

如果参数带有缺省值，请加在转换器之后：

```
name_of_parameter: converter = default_value
```

Argument Clinic's support for "default values" is quite sophisticated; see [如何给形参赋默认值](#) for more information.

Next, add a blank line below the parameters.

What's a "converter"? It establishes both the type of the variable used in C, and the method to convert the Python value into a C value at runtime. For now you're going to use what's called a "legacy converter" --- a convenience syntax intended to make porting old code into Argument Clinic easier.

For each parameter, copy the "format unit" for that parameter from the `PyArg_Parse()` format argument and specify *that* as its converter, as a quoted string. The "format unit" is the formal name for the one-to-three character substring of the *format* parameter that tells the argument parsing function what the type of the variable is and how to convert it. For more on format units please see [arg-parsing](#).

对于像 `z#` 这样的多字符格式单元，要使用 2-3 个字符组成的整个字符串。

示例：

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

If your function has `|` in the format string, meaning some parameters have default values, you can ignore it. Argument Clinic infers which parameters are optional based on whether or not they have default values.

如果函数的格式字符串中包含 `$`，意味着只接受关键字参数，请在第一个关键字参数之前单独给出一行 `*`，缩进与参数行对齐。

`_pickle.Pickler.dump()` has neither, so our sample is unchanged.

Next, if the existing C function calls `PyArg_ParseTuple()` (as opposed to `PyArg_ParseTupleAndKeywords()`), then all its arguments are positional-only.

To mark parameters as positional-only in Argument Clinic, add a `/` on a line by itself after the last positional-only parameter, indented the same as the parameter lines.

示例：

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
/
```

(下页继续)

```
Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

It can be helpful to write a per-parameter docstring for each parameter. Since per-parameter docstrings are optional, you can skip this step if you prefer.

Nevertheless, here's how to add a per-parameter docstring. The first line of the per-parameter docstring must be indented further than the parameter definition. The left margin of this first line establishes the left margin for the whole per-parameter docstring; all the text you write will be outdented by this amount. You can write as much text as you like, across multiple lines if you wish.

示例:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

Save and close the file, then run `Tools/clinic/clinic.py` on it. With luck everything worked---your block now has output, and a `.c.h` file has been generated! Reload the file in your text editor to see the generated code:

```
/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

static PyObject *
_pickle_Pickler_dump(PicklerObject *self, PyObject *obj)
/*[clinic end generated code: output=87ecad1261e02ac7 input=552eb1c0f52260d9]*/
```

Obviously, if Argument Clinic didn't produce any output, it's because it found an error in your input. Keep fixing your errors and retrying until Argument Clinic processes your file without complaint.

For readability, most of the glue code has been generated to a `.c.h` file. You'll need to include that in your original `.c` file, typically right after the clinic module block:

```
#include "clinic/_pickle.c.h"
```

请仔细检查 Argument Clinic 生成的参数解析代码, 是否与原有代码基本相同。

首先, 确保两种代码使用相同的参数解析函数。原有代码必须调用 `PyArg_ParseTuple()` 或 `PyArg_ParseTupleAndKeywords()`; 确保 Argument Clinic 生成的代码调用 完全相同的函数。

Second, the format string passed in to `PyArg_ParseTuple()` or `PyArg_ParseTupleAndKeywords()` should be *exactly* the same as the hand-written one in the existing function, up to the colon or semi-colon.

Argument Clinic always generates its format strings with a `:` followed by the name of the function. If the existing code's format string ends with `;`, to provide usage help, this change is harmless --- don't worry about it.

Third, for parameters whose format units require two arguments, like a length variable, an encoding string, or a pointer to a conversion function, ensure that the second argument is *exactly* the same between the two invocations.

Fourth, inside the output portion of the block, you'll find a preprocessor macro defining the appropriate static PyMethodDef structure for this builtin:

```
#define __PICKLE_PICKLER_DUMP_METHODDEF \
{"dump", (PyCFunction)__pickle_Pickler_dump, METH_O, __pickle_Pickler_dump__doc__},
```

This static structure should be *exactly* the same as the existing static PyMethodDef structure for this builtin.

只要上述这几点存在不一致，请调整 Argument Clinic 函数定义，并重新运行 Tools/clinic/clinic.py，直至完全相同。

注意，输出部分的最后一行是“实现”函数的声明。也就是该内置函数的实现代码所在。删除需要修改的函数的现有原型，但保留开头的大括号。再删除其参数解析代码和输入变量的所有声明。注意现在 Python 所见的参数即为此实现函数的参数；如果实现代码给这些变量采用了不同的命名，请进行修正。

Let's reiterate, just because it's kind of weird. Your code should now look like this:

```
static return_type
your_function_impl(...)
/*[clinic end generated code: input=..., output=...]*/
{
    ...
}
```

Argument Clinic generated the checksum line and the function prototype just above it. You should write the opening and closing curly braces for the function, and the implementation inside.

示例：

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/
/*[clinic end generated code: checksum=da39a3ee5e6b4b0d3255bfef95601890afd80709]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

PyDoc_STRVAR(__pickle_Pickler_dump__doc__,
"Write a pickled representation of obj to the open file.\n"
"\n"
...
static PyObject *
_pickle_Pickler_dump_impl(PicklerObject *self, PyObject *obj)
/*[clinic end generated code: checksum=3bd30745bf206a48f8b576a1da3d90f55a0a4187]*/
{
    /* Check whether the Pickler was initialized correctly (issue3664).
       Developers often forget to call __init__() in their subclasses, which
       would trigger a segfault without this check. */
    if (self->write == NULL) {
        PyErr_Format(PicklingError,
                     "Pickler.__init__() was not called by %s.__init__()",
```

(下页继续)


```

        Py_TYPE(self)->tp_name);
    return NULL;
}

if (_Pickler_ClearBuffer(self) < 0) {
    return NULL;
}

...

```

Remember the macro with the `PyMethodDef` structure for this function? Find the existing `PyMethodDef` structure for this function and replace it with a reference to the macro. If the builtin is at module scope, this will probably be very near the end of the file; if the builtin is a class method, this will probably be below but relatively near to the implementation.

Note that the body of the macro contains a trailing comma; when you replace the existing static `PyMethodDef` structure with the macro, *don't* add a comma to the end.

示例:

```

static struct PyMethodDef Pickler_methods[] = {
    __PICKLE_PICKLER_DUMP_METHODDEF
    __PICKLE_PICKLER_CLEAR_MEMO_METHODDEF
    {NULL, NULL} /* sentinel */
};

```

Argument Clinic 可能生成新的 `_Py_ID` 实例。举例来说:

```
&_Py_ID(new_unique_py_id)
```

If it does, you'll have to run `make regen-global-objects` to regenerate the list of precompiled identifiers at this point.

Finally, compile, then run the relevant portions of the regression-test suite. This change should not introduce any new compile-time warnings or errors, and there should be no externally visible change to Python's behavior, except for one difference: `inspect.signature()` run on your function should now provide a valid signature!

祝贺你, 现在已经用 Argument Clinic 移植了第一个函数。

4 常用方案指引

4.1 How to rename C functions and variables generated by Argument Clinic

Argument Clinic 会自动为其生成的函数命名。如果生成的名称与现有的 C 函数冲突, 这偶尔可能会造成问题, 有一个简单的解决方案: 覆盖 C 函数的名称。只要在函数声明中加入关键字 `"as"`, 然后再加上要使用的函数名。Argument Clinic 将以该函数名为基础作为 (生成的) 函数名, 然后在后面加上 `"_impl"`, 并用作实现函数的名称。

For example, if we wanted to rename the C function names generated for `pickle.Pickler.dump()`, it'd look like this:

```

/*[clinic input]
pickle.Pickler.dump as pickler_dumper
...

```

The base function would now be named `pickler_dumper()`, and the impl function would now be named `pickler_dumper_impl()`.

同样的问题依然会出现：想给某个参数取个 Python 用名，但在 C 语言中可能用不了。Argument Clinic 允许在 Python 和 C 中为同一个参数取不同的名字，依然是利用 "as" 语法：

```
/*[clinic input]
pickle.Pickler.dump

    obj: object
    file as file_obj: object
    protocol: object = NULL
    *
    fix_imports: bool = True
```

Here, the name used in Python (in the signature and the keywords array) would be *file*, but the C variable would be named *file_obj*.

You can use this to rename the *self* parameter too!

4.2 如何转换使用 PyArg_UnpackTuple 的函数

To convert a function parsing its arguments with `PyArg_UnpackTuple()`, simply write out all the arguments, specifying each as an `object`. You may specify the *type* argument to cast the type as appropriate. All arguments should be marked positional-only (add a `/` on a line by itself after the last argument).

目前，所生成的代码将会用到 `PyArg_ParseTuple()`，但很快会做出改动。

4.3 如何使用可选分组

有些过时的函数用到了一种让人头疼的函数解析方式：计算位置参数的数量，据此用 `switch` 语句进行各个不同的 `PyArg_ParseTuple()` 调用。（这些函数不能接受只认关键字的参数。）在没有 `PyArg_ParseTupleAndKeywords()` 之前，这种方式曾被用于模拟可选参数。

While functions using this approach can often be converted to use `PyArg_ParseTupleAndKeywords()`, optional arguments, and default values, it's not always possible. Some of these legacy functions have behaviors `PyArg_ParseTupleAndKeywords()` doesn't directly support. The most obvious example is the builtin function `range()`, which has an optional argument on the *left* side of its required argument! Another example is `curses.window.addch()`, which has a group of two arguments that must always be specified together. (The arguments are called *x* and *y*; if you call the function passing in *x*, you must also pass in *y* —and if you don't pass in *x* you may not pass in *y* either.)

不管怎么说，Argument Clinic 的目标就是在不改变语义的情况下支持所有现有 CPython 内置参数的解析。因此，Argument Clinic 采用所谓的 可选组 方案来支持这种解析方式。可选组是必须一起传入的参数组。他们可以在必需参数的左边或右边，只能用于只认位置的参数。

备注：可选组 仅适用于多次调用 `PyArg_ParseTuple()` 的函数！采用任何其他方式解析参数的函数，应该几乎不采用可选组转换为 Argument Clinic 解析。目前，采用可选组的函数在 Python 中无法获得准确的签名，因为 Python 不能理解这个概念。请尽可能避免使用可选组。

To specify an optional group, add a `[` on a line by itself before the parameters you wish to group together, and a `]` on a line by itself after these parameters. As an example, here's how `curses.window.addch()` uses optional groups to make the first two parameters and the last parameter optional:

```
/*[clinic input]
curses.window.addch

    [
    x: int
        X-coordinate.
```

(下页继续)

```

y: int
    Y-coordinate.
]

ch: object
    Character to add.

[
    attr: long
        Attributes for the character.
]
/
...

```

注：

- 每一个可选组，都会额外传入一个代表分组的参数。参数为 `int` 型，名为 `group_{direction}_{number}`，其中 `{direction}` 取决于此参数组位于必需参数 `right` 还是 `left`，而 `{number}` 是一个递增数字（从 1 开始），表示此参数组与必需参数之间的距离。在调用函数时，若未用到此参数组则此参数将设为零，若用到了参数组则该参数为非零。所谓的用到或未用到，是指在本次调用中形参是否收到了实参。
- 如果不存在必需参数，可选组的行为等同于出现在必需参数的右侧。
- 在模棱两可的情况下，参数解析代码更倾向于参数左侧（在必需参数之前）。
- 可选组只能包含只认位置的参数。
- 可选组 仅限用于过时代码。请勿在新的代码中使用可选组。

4.4 如何使用真正的 Argument Clinic 转换器，而不是“旧式转换器”

为了节省时间，尽量减少要学习的内容，实现第一次适用 Argument Clinic 的移植，上述练习简述的是“传统转换器”的用法。“传统转换器”只是一种简便用法，目的就是更容易地让现有代码移植为适用于 Argument Clinic。说白了，在移植 Python 3.4 的代码时，可以考虑采用。

不过从长远来看，可能希望所有代码块都采用真正的 Argument Clinic 转换器语法。原因如下：

- 合适的转换器可读性更好，意图也更清晰。
- 有些格式单元是“传统转换器”无法支持的，因为这些格式需要带上参数，而传统转换器的语法不支持指定参数。
- 后续可能会有新版的参数解析库，提供超过 `PyArg_ParseTuple()` 支持的功能；而这种灵活性将无法适用于传统转换器转换的参数。

因此，若是不介意多花一点精力，请使用正常的转换器，而不是传统转换器。

简而言之，Argument Clinic（非传统）转换器的语法看起来像是 Python 函数调用。但如果函数没有明确的参数（所有函数都取默认值），则可以省略括号。因此 `bool` 和 `bool()` 是完全相同的转换器。

Argument Clinic 转换器的所有参数都只认关键字。所有 Argument Clinic 转换器均可接受以下参数：

c_default 该参数在 C 语言中的默认值。具体来说，将是在“解析函数”中声明的变量的初始值。用法参见 [the section on default values](#)。定义为字符串。

annotation 参数的注解值。目前尚不支持，因为 **PEP 8** 规定 Python 库不得使用注解。

unused 在 `impl` 函数签名中用 `Py_UNUSED` 来包装函数。

此外，某些转换器还可接受额外的参数。下面列出了这些额外参数及其含义：

accept 一些 Python 类型的集合（可能还有伪类型）；用于限制只接受这些类型的 Python 参数。（并非通用特性；只支持传统转换器列表中给出的类型）。

若要能接受 None, 请在集合中添加 NoneType。

bitwise 仅用于无符号整数。写入形参的将是 Python 实参的原生整数值, 不做任何越界检查, 即便是负值也一样。

converter 仅用于 object 转换器。为某个 C 转换函数指定名称, 用于将对象转换为原生类型。

encoding 仅用于字符串。指定将 Python str(Unicode) 转换为 C 语言的 char * 时应该采用的编码。

subclass_of 仅用于 object 转换器。要求 Python 值是 Python 类型的子类, 用 C 语言表示。

type 仅用于 object 和 self 转换器。指定用于声明变量的 C 类型。默认值是 "PyObject *"。

zeroes 仅用于字符串。如果为 True, 则允许在值中嵌入 NUL 字节 ('\0')。字符串的长度将通过名为 <parameter_name>_length 的参数传入, 跟在字符串参数的后面。

Please note, not every possible combination of arguments will work. Usually these arguments are implemented by specific PyArg_ParseTuple() *format units*, with specific behavior. For example, currently you cannot call unsigned_short without also specifying bitwise=True. Although it's perfectly reasonable to think this would work, these semantics don't map to any existing format unit. So Argument Clinic doesn't support it. (Or, at least, not yet.)

下表列出了传统转换器与真正的 Argument Clinic 转换器之间的映射关系。左边是传统的转换器, 右边是应该换成的文本。

'B'	unsigned_char(bitwise=True)
'b'	unsigned_char
'c'	char
'C'	int(accept={str})
'd'	double
'D'	Py_complex
'es'	str(encoding='name_of_encoding')
'es#'	str(encoding='name_of_encoding', zeroes=True)
'et'	str(encoding='name_of_encoding', accept={bytes, bytearray, str})
'et#'	str(encoding='name_of_encoding', accept={bytes, bytearray, str}, zeroes=True)
'f'	float
'h'	short
'H'	unsigned_short(bitwise=True)
'i'	int
'I'	unsigned_int(bitwise=True)
'k'	unsigned_long(bitwise=True)
'K'	unsigned_long_long(bitwise=True)
'l'	long
'L'	long long
'n'	Py_ssize_t
'O'	object
'O!'	object(subclass_of='&PySomething_Type')
'O&'	object(converter='name_of_c_function')
'p'	bool
'S'	PyBytesObject
's'	str
's#'	str(zeroes=True)
's*'	Py_buffer(accept={buffer, str})
'U'	unicode
'u'	wchar_t
'u#'	wchar_t(zeroes=True)
'w*'	Py_buffer(accept={rwbuffer})

下页继续

表 1 - 续上页

'Y'	PyByteArrayObject
'y'	str(accept={bytes})
'y#'	str(accept={robuffer}, zeroes=True)
'y*'	Py_buffer
'Z'	wchar_t(accept={str, NoneType})
'Z#'	wchar_t(accept={str, NoneType}, zeroes=True)
'z'	str(accept={str, NoneType})
'z#'	str(accept={str, NoneType}, zeroes=True)
'z*'	Py_buffer(accept={buffer, str, NoneType})

举个例子，下面是采用合适的转换器的例子 `pickle.Pickler.dump`:

```
/*[clinic input]
pickle.Pickler.dump

    obj: object
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

真正的转换器有一个优点，就是比传统的转换器更加灵活。例如，`unsigned_int` 转换器（以及所有 `unsigned_` 转换器）可以不设置 `bitwise=True`。他们默认会对数值进行范围检查，而且不会接受负数。用传统转换器就做不到这一点。

Argument Clinic 会列明其全部转换器。每个转换器都会给出可接受的全部参数，以及每个参数的默认值。只要运行 `Tools/clinic/clinic.py --converters` 就能得到完整的列表。

4.5 如何使用 `Py_buffer` 转换器

在使用 `Py_buffer` 转换器（或者 `'s*'`、`'w*'`、`'*y'` 或 `'z*'` 传统转换器）时，不可在所提供的缓冲区上调用 `PyBuffer_Release()`。**Argument Clinic** 生成的代码会自动完成此操作（在解析函数中）。

4.6 如何使用高级转换器

还记得编写第一个函数时跳过的那些格式单元吗，因为他们是高级内容？下面就来介绍这些内容。

The trick is, all those format units take arguments—either conversion functions, or types, or strings specifying an encoding. (But “legacy converters” don’t support arguments. That’s why we skipped them for your first function.) The argument you specified to the format unit is now an argument to the converter; this argument is either *converter* (for `O&`), *subclass_of* (for `O!`), or *encoding* (for all the format units that start with `e`).

When using *subclass_of*, you may also want to use the other custom argument for `object()`: *type*, which lets you set the type actually used for the parameter. For example, if you want to ensure that the object is a subclass of `PyUnicode_Type`, you probably want to use the converter `object(type='PyUnicodeObject *', subclass_of='&PyUnicode_Type')`.

One possible problem with using **Argument Clinic**: it takes away some possible flexibility for the format units starting with `e`. When writing a `PyArg_Parse*` call by hand, you could theoretically decide at runtime what encoding string to pass to that call. But now this string must be hard-coded at **Argument-Clinic**-preprocessing-time. This limitation is deliberate; it made supporting this format unit much easier, and may allow for future optimizations. This restriction doesn’t seem unreasonable; CPython itself always passes in static hard-coded encoding strings for parameters whose format units start with `e`.

4.7 如何给形参赋默认值

参数的默认值可以是多个值中的一个。最简单的可以是字符串、int 或 float 字面量。

```
foo: str = "abc"
bar: int = 123
bat: float = 45.6
```

还可以使用 Python 的任何内置常量。

```
yep: bool = True
nope: bool = False
nada: object = None
```

对默认值 NULL 和简单表达式还提供特别的支持，下面将一一介绍。

默认值 NULL

对于字符串和对象参数而言，可以设为 None，表示没有默认值。但这意味着会将 C 变量初始化为 Py_None。为了方便起见，提供了一个特殊值 NULL，目的就是为了让 Python 认为默认值就是 None，而 C 变量则会初始化为 NULL。

符号化默认值

提供给参数的默认值不能是表达式。目前明确支持以下形式：

- 数值型常数（整数和浮点数）。
- 字符串常量
- True、False 和 None。
- Simple symbolic constants like `sys.maxsize`, which must start with the name of the module

（未来可能需要加以细化，以便可以采用 `CONSTANT - 1` 之类的完整表达式。）

表达式作为默认值

参数的默认值不仅可以是字面量。还可以是一个完整的表达式，可采用数学运算符及对象的属性。但这种支持并没有那么简单，因为存在一些不明显的语义。

请考虑以下例子：

```
foo: Py_ssize_t = sys.maxsize - 1
```

`sys.maxsize` can have different values on different platforms. Therefore Argument Clinic can't simply evaluate that expression locally and hard-code it in C. So it stores the default in such a way that it will get evaluated at runtime, when the user asks for the function's signature.

What namespace is available when the expression is evaluated? It's evaluated in the context of the module the builtin came from. So, if your module has an attribute called `max_widgets`, you may simply use it:

```
foo: Py_ssize_t = max_widgets
```

If the symbol isn't found in the current module, it fails over to looking in `sys.modules`. That's how it can find `sys.maxsize` for example. (Since you don't know in advance what modules the user will load into their interpreter, it's best to restrict yourself to modules that are preloaded by Python itself.)

Evaluating default values only at runtime means Argument Clinic can't compute the correct equivalent C default value. So you need to tell it explicitly. When you use an expression, you must also specify the equivalent expression in C, using the `c_default` parameter to the converter:

```
foo: Py_ssize_t(c_default="PY_SSIZE_T_MAX - 1") = sys.maxsize - 1
```

还有一个问题也比较复杂。**Argument Clinic** 无法事先知道表达式是否有效。解析只能保证看起来是有效值，但无法实际知晓。在用表达式时须十分小心，确保在运行时能得到有效值。

最后一点，由于表达式必须能表示为静态的 C 语言值，所以存在许多限制。以下列出了不得使用的 Python 特性：

- 功能
- 行内 if 语句 (3 if foo else 5)
- 序列类自动解包 (*[1, 2, 3])
- 列表、集合、字典的解析和生成器表达式。
- 元组、列表、集合、字典的字面量

4.8 如何使用返回转换器

在默认情况下，**Argument Clinic** 生成的 `impl` 函数返回 `PyObject *`。但是你的 C 函数往往要计算某个 C 类型，然后最终将其转换为 `PyObject *`。**Argument Clinic** 会负责将你的输入由 Python 类型转换为原生 C 类型——为什么不让它将你的返回值也由原生 C 类型转换为 Python 类型呢？

这就是“返回值转换器”要做的事。它将你的 `impl` 函数修改为返回某种 C 类型，然后在生成的 `type, then adds code to the generated` (非 `impl`) 函数中添加代码来处理该值到相应 `PyObject *` 的转换。

返回值转换器的语法类似于形参转换器。你以为函数自身添加返回标注的形式来指定返回值转换器，即使用 `->` 标注。

例如：

```
/*[clinic input]
add -> int

    a: int
    b: int
/

[clinic start generated code]*/
```

返回值转换器的行为和形参转换器基本一致；它们都接受一些参数，这些参数全部是仅限关键字的，如果你不修改任何默认参数则可以省略括号。

(如果函数同时用到了 `"as"` 和返回值转换器，`"as"` 应位于返回值转换器之前。)

There's one additional complication when using return converters: how do you indicate an error has occurred? Normally, a function returns a valid (non-NULL) pointer for success, and NULL for failure. But if you use an integer return converter, all integers are valid. How can **Argument Clinic** detect an error? Its solution: each return converter implicitly looks for a special value that indicates an error. If you return that value, and an error has been set (`c:func:PyErr_Occurred` returns a true value), then the generated code will propagate the error. Otherwise it will encode the value you return like normal.

目前 **Argument Clinic** 只支持少数几种返回值转换器。

```
bool
double
float
int
long
Py_ssize_t
size_t
unsigned int
unsigned long
```


它们全都不接受形参。它们全都返回 -1 来表示错误。to indicate error.

只要运行 `Tools/clinic/clinic.py --converters`，即可查看 Argument Clinic 支持的所有返回值转换器，包括其参数。

4.9 如何克隆现有的函数

如果已有一些函数比较相似，或许可以采用 Clinic 的“克隆”功能。克隆之后能够复用以下内容：

- 参数，包括：
 - 名称
 - 转换器（带有全部参数）
 - 默认值
 - 参数前的文档字符串
 - 类别（只认位置、位置或关键字、只认关键字）
- 返回值转换器

唯一不从原函数中复制的是文档字符串；这样就能指定一个新的文档串。

下面是函数的克隆方法：

```
/*[clinic input]
module.class.new_function [as c_basename] = module.class.existing_function

Docstring for new_function goes here.
[clinic start generated code]*/
```

（原函数可以位于不同的模块或类中。示例中的 `module.class` 只是为了说明，两个函数都必须使用全路径）。

对不起，没有用于部分克隆某个函数、或克隆某个函数并对其进行修改的语法。克隆是一个只有全部和全无两种选项的操作。

另外，要克隆的函数必须在当前文件中已有定义。

4.10 如何调用 Python 代码

下面的高级内容需要编写 Python 代码，存于 C 文件中，并修改 Argument Clinic 的运行状态。其实很简单：只需定义一个 Python 块。

Python 块的分隔线与 Argument Clinic 函数块不同。如下所示：

```
/*[python input]
# python code goes here
[python start generated code]*/
```

Python 块内的所有代码都会在解析时执行。块内写入 `stdout` 的所有文本都被重定向到块后的“输出”部分。

以下例子包含了 Python 块，用于在 C 代码中添加一个静态整数变量：

```
/*[python input]
print('static int __ignored_unused_variable__ = 0;')
[python start generated code]*/
static int __ignored_unused_variable__ = 0;
/*[python checksum:...]*/*
```


4.11 如何使用“自转换器”

Argument Clinic automatically adds a "self" parameter for you using a default converter. It automatically sets the type of this parameter to the "pointer to an instance" you specified when you declared the type. However, you can override Argument Clinic's converter and specify one yourself. Just add your own *self* parameter as the first parameter in a block, and ensure that its converter is an instance of `self_converter` or a subclass thereof.

这有什么用呢？可用于覆盖 `self` 的类型，或为其给个不同的默认名称。

How do you specify the custom type you want to cast `self` to? If you only have one or two functions with the same type for `self`, you can directly use Argument Clinic's existing `self` converter, passing in the type you want to use as the *type* parameter:

```
/*[clinic input]
_pickle.Pickler.dump

    self: self(type="PicklerObject *")
    obj: object
/

Write a pickled representation of the given object to the open file.
[clinic start generated code]*/
```

On the other hand, if you have a lot of functions that will use the same type for `self`, it's best to create your own converter, subclassing `self_converter` but overwriting the `type` member:

```
/*[python input]
class PicklerObject_converter(self_converter):
    type = "PicklerObject *"
[python start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    self: PicklerObject
    obj: object
/

Write a pickled representation of the given object to the open file.
[clinic start generated code]*/
```

4.12 如何使用“定义类”转换器

Argument Clinic 为访问方法定义所在的类提供了便利。因为 `heap type` 方法需要获取模块级的运行状态，所以就十分有用。`PyType_FromModuleAndSpec()` 会将堆类型与模块关联起来。然后类就可用 `PyType_GetModuleState()` 获取模块状态了，比如利用模块的方法进行获取。

Example from `Modules/zlibmodule.c`. First, `defining_class` is added to the clinic input:

```
/*[clinic input]
zlib.Compress.compress

    cls: defining_class
    data: Py_buffer
        Binary data to be compressed.
/
```

运行 Argument Clinic 工具后，会生成以下函数签名：

```

/*[clinic start generated code]*/
static PyObject *
zlib_Compress_compress_impl(compobject *self, PyTypeObject *cls,
                           Py_buffer *data)
/*[clinic end generated code: output=6731b3f0ff357ca6 input=04d00f65ab01d260]*/

```

现在，以下代码可以用 `PyType_GetModuleState(cls)` 获取模块状态了：

```
zlibstate *state = PyType_GetModuleState(cls);
```

Each method may only have one argument using this converter, and it must appear after `self`, or, if `self` is not used, as the first argument. The argument will be of type `PyTypeObject *`. The argument will not appear in the `__text_signature__`.

The `defining_class` converter is not compatible with `__init__()` and `__new__()` methods, which cannot use the `METH_METHOD` convention.

It is not possible to use `defining_class` with slot methods. In order to fetch the module state from such methods, use `PyType_GetModuleByDef()` to look up the module and then `PyModule_GetState()` to fetch the module state. Example from the `setattro` slot method in [Modules/_threadmodule.c](#):

```

static int
local_setattro(localobject *self, PyObject *name, PyObject *v)
{
    PyObject *module = PyType_GetModuleByDef(Py_TYPE(self), &thread_module);
    thread_module_state *state = get_thread_state(module);
    ...
}

```

参见 [PEP 573](#)。

4.13 如何编写自定义转换器

A converter is a Python class that inherits from [CConverter](#). The main purpose of a custom converter, is for parameters parsed with the `O&` format unit --- parsing such a parameter means calling a `PyArg_ParseTuple()` "converter function".

Your converter class should be named `ConverterName_converter`. By following this convention, your converter class will be automatically registered with Argument Clinic, with its *converter name* being the name of your converter class with the `_converter` suffix stripped off.

Instead of subclassing `CConverter.__init__()`, write a `converter_init()` method. `converter_init()` always accepts a *self* parameter. After *self*, all additional parameters **must** be keyword-only. Any arguments passed to the converter in Argument Clinic will be passed along to your `converter_init()` method. See [CConverter](#) for a list of members you may wish to specify in your subclass.

Here's the simplest example of a custom converter, from [Modules/zlibmodule.c](#):

```

/*[python input]

class ssize_t_converter(CConverter):
    type = 'Py_ssize_t'
    converter = 'ssize_t_converter'

[python start generated code]*/
/*[python end generated code: output=da39a3ee5e6b4b0d input=35521e4e733823c7]*/

```

This block adds a converter named `ssize_t` to Argument Clinic. Parameters declared as `ssize_t` will be declared with type `Py_ssize_t`, and will be parsed by the `'O&'` format unit, which will call the `ssize_t_converter()` converter C function. `ssize_t` variables automatically support default values.

更复杂些的自定义转换器，可以插入自定义 C 代码来进行初始化和清理工作。可以在 CPython 源码中看到自定义转换器的更多例子；只要在 C 文件中搜索字符串 CConverter 即可。

4.14 如何编写自定义返回转换器

自定义的返回值转换器的写法，与自定义的转换器十分类似。因为返回值转换器本身就很简单，编写起来就简单一些。

Return converters must subclass CReturnConverter. There are no examples yet of custom return converters, because they are not widely used yet. If you wish to write your own return converter, please read [Tools/clinic/clinic.py](#), specifically the implementation of CReturnConverter and all its subclasses.

4.15 如何转换 METH_O 和 METH_NOARGS 函数

To convert a function using METH_O, make sure the function's single argument is using the object converter, and mark the arguments as positional-only:

```
/*[clinic input]
meth_o_sample

    argument: object
/
[clinic start generated code]*/
```

To convert a function using METH_NOARGS, just don't specify any arguments.

You can still use a self converter, a return converter, and specify a *type* argument to the object converter for METH_O.

4.16 如何转换 tp_new 和 tp_init 函数

You can convert tp_new and tp_init functions. Just name them `__new__` or `__init__` as appropriate. Notes:

- 为转换 `__new__` 而生成的函数名不会以其默认名称结尾。只会是转换为合法 C 标识符的类名。
- No PyMethodDef #define is generated for these functions.
- `__init__` 函数将返回 `int`，而不是 `PyObject *`。
- 将文档字符串用作类文档字符串。
- 虽然 `__new__` 和 `__init__` 函数必须以 `args` 和 `kwargs` 对象作为参数，但在转换时可按个人喜好定义函数签名。（如果原函数不支持关键字参数，则生成的解析函数在收到关键字参数时会抛出异常）。

4.17 如何修改和重定向 Clinic 的输出

若是让 Clinic 的输出与传统的手写 C 代码交织在一起，可能会不方便阅读。幸好可以对 Clinic 进行配置：可以将输出结果缓存起来以供输出，或将输出结果写入文件中。针对 Clinic 生成的输出结果，还可以为每一行都加上前缀或后缀。

虽然修改 Clinic 的输出提升了可读性，但可能会导致 Clinic 代码使用了未经定义的类型，或者会提前用到 Clinic 生成的代码。通过重新安排声明在代码文件的位置，或将 Clinic 生成的代码移个位置，即可轻松解决上述问题。（这就是 Clinic 默认是全部输出到当前代码块的原因；虽然许多人认为降低了可读性，但这样就根本不用重新编排代码来解决提前引用的问题）。

就从定义一些术语开始吧：

**** 区块 (field) **** A field, in this context, is a subsection of Clinic's output. For example, the `#define` for the `PyMethodDef` structure is a field, called `methoddef_define`. Clinic has seven different fields it can output per function definition:

```
docstring_prototype
docstring_definition
methoddef_define
impl_prototype
parser_prototype
parser_definition
impl_definition
```

区块均以 "`<a>_`" 形式命名, 其中 "`<a>`" 是所代表的语义对象 (解析函数、`impl` 函数、文档字符串或 `methoddef` 结构), "``" 表示该区块的类别。以 "`_prototype`" 结尾的区块名表示这只是个前向声明, 没有实际的函数体或数据; 以 "`_definition`" 结尾的区块名则表示这是实际的函数定义, 包含了函数体和数据。 ("`methoddef`" 比较特殊, 是唯一一个以 "`_define`" 结尾的区块名, 表明这是一个预处理器 `#define`。)

**** 输出目标 (destination) **** 输出目标是 Clinic 可以进行输出的地方。内置的输出目标有 5 种:

block 默认的输出目标: 在 Clinic 当前代码块的输出区域进行输出。

buffer 文本缓冲区, 可将文本保存起来以便后续使用。输出的文本会加入现有文本的末尾。如果 Clinic 处理完文件后缓冲区中还留有文本, 则会报错。

file A separate "clinic file" that will be created automatically by Clinic. The filename chosen for the file is `{basename}.clinic{extension}`, where `basename` and `extension` were assigned the output from `os.path.splitext()` run on the current file. (Example: the file destination for `_pickle.c` would be written to `_pickle.clinic.c`.)

重点: 若要使用 **** "file" **** 作为输出目标, 你 ***** 必须签入 ***** 生成的文件!

two-pass 类似于 `buffer` 的缓冲区。不过 `two-pass` 缓冲区只能转储一次, 将会输出处理过程中发送给它的所有文本, 甚至包括转储点 **** 之后 **** 的 Clinic 块。

suppress 禁止输出文本——抛弃输出。

Clinic 定义了 5 个新的指令, 以便修改输出方式。

第一个新指令是 `dump`:

```
dump <destination>
```

将指定输出目标的当前内容转储到当前块的输出中, 并清空输出目标。仅适用于 `buffer` 和 `two-pass` 目标。

第二个新指令是 `output`。`output` 最简单的格式如下所示:

```
output <field> <destination>
```

这会通知 Clinic 将指定 ****field**** 输出到指定 ****destination**** 中去。`output` 还支持一个特殊的元目标 `everything`, 通知 Clinic 将 **** 所有 **** 区块都输出到该 **** 目标 ****。

`output` 还包含一些函数:

```
output push
output pop
output preset <preset>
```

`output push` 和 `output pop` 能在内部的配置栈中压入和弹出配置, 这样就可以临时修改输出配置, 然后再轻松恢复之前的配置。只需在修改前入栈保存当前配置, 在恢复配置时再弹出即可。

`output preset` 将 Clinic 的输出目标设为内置预设目标之一, 如下所示:

block Clinic 的初始设置。输入块后面紧接着写入所有内容。

关闭 `parser_prototype` 和 `docstring_prototype`，并将其他所有内容写入 `block`。

file 目的是全部输出至“Clinic 文件”中。然后在文件顶部附近 `#include` 该文件。可能需要重新调整代码顺序才能正常运行，通常只要为 `typedef` 和 `PyObject` 定义创建前向声明即可。

关闭 `parser_prototype` 和 `docstring_prototype`，将 `impl_definition` 写入 `block`，其他内容写入 `file`。

默认文件名为 `"{dirname}/clinic/{basename}.h"`。

buffer 将 Clinic 的大部分输出保存起来，在快结束时写入文件。如果 Python 文件存放的是编写模块或内置类型的代码，建议紧挨着模块或内置类型的静态结构之前对缓冲区进行转储；这些结构通常位于结尾附近。如果在文件的中间位置定义了静态 `PyMethodDef` 数组，采用 `buffer` 输出所需的代码编辑工作可能比用 `file` 要多些。

关闭 `parser_prototype`、`impl_prototype` 和 `docstring_prototype`，将 `impl_definition` 写入 `block`，其他输出都写入 `file`。

two-pass 类似于预设的 `buffer` 输出，但会把前向声明写入 `two-pass` 缓冲区，将函数定义写入 `buffer`。这与预设的 `buffer` 类似，但所需的代码编辑工作可能会减少。将 `two-pass` 缓冲区转储到文件的顶部，将 `buffer` 转储到文件末尾，就像预设的 `buffer` 一样。

关闭 `impl_prototype`，将 `impl_definition` 写入 `block`，将 `docstring_prototype`、`methoddef_define` 和 `parser_prototype` 写入 `two-pass`，其他输出都写入 `buffer`。

partial-buffer 与预设的 `buffer` 类似，但会向 `block` 写入更多内容，而只向 `buffer` 写入真正大块的生成代码。这样能完全避免 `buffer` 的提前引用问题，代价是输出到代码块中的内容会稍有增加。在快结束时转储 `buffer`，就像采用预设的 `buffer` 配置一样。

关闭 `impl_prototype`，将 `docstring_definition` 和 `parser_definition` 写入 `buffer`，其他输出都写入 `block`。

第三个新指令是 `destination`：

```
destination <name> <command> [...]
```

向名为 `name` 的目标执行输出。

定义了两个子命令：`new` 和 `clear`。

子命令 `new` 工作方式如下：

```
destination <name> new <type>
```

新建一个目标，名称为 `<name>`，类型为 `<type>`。

输出目标的类型有 5 种：

suppress 忽略文本。

block 将文本写入当前代码块中。这就是 Clinic 原来的做法。

buffer 简单的文本缓冲区，就像上述的内置“buffer”目标。

file 文本文件。文件目标多了一个参数，模板用于生成文件名，类似于：

```
destination <name> new <type> <file_template>
```

模版可以引用 3 个内部字符串，将会用文件名的对应部分替代：

{path} 文件的全路径，包含文件夹和完整的文件名。

{dirname} 文件所在文件夹名。

{basename} 只有文件名，不含文件夹。

{basename_root} 去除了扩展名后的文件名（不含最后一个“.”）。

{basename_extension} 包含最后一个“.”及后面的字符。如果文件名中不含句点，则为空字符串。

如果文件名中不含句点符，**{basename}** 和 **{basename_root}** 是一样的，而 **{basename_extension}** 则为空。“**{basename_root}{basename_extension}**”与“**{basename}**”一定是完全相同的。（英文原文貌似有误）

two-pass two-pass 缓冲区，类似于上述的内置“two-pass”输出目标。

子命令 **clear** 的工作方式如下：

```
destination <name> clear
```

清空输出目标中所有文本。（不知用途何在，但也许做实验时会有用吧。）

第 4 个新指令是 **set**：

```
set line_prefix "string"
set line_suffix "string"
```

set 能设置 *Clinic* 的两个内部变量值。“*line_prefix*”是 *Clinic* 每行输出的前缀字符串；*line_suffix* 是 *Clinic* 每行输出的后缀字符串。

两者都支持两种格式字符串：

{block comment start} 转成字符串 `/*`，是 C 文件的注释起始标记。

{block comment end} 转成字符串 `*/`，是 C 文件的注释结束标记。

最后一个新指令是无需直接使用的 **preserve**。

```
preserve
```

通知 *Clinic* 输出内容应保持原样。这是在转储至 `file` 文件中时，供 *Clinic* 内部使用的；以便 *Clinic* 能利用已有的校验函数，确保文件在被覆盖之前没进行人工修改过。

4.18 如何使用 `#ifdef` 技巧

若要转换的函数并非通用于所有平台，可以采用一个技巧。当前代码可能如下所示：

```
#ifdef HAVE_FUNCTIONNAME
static module_functionname(...)
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

在底部的 `PyMethodDef` 结构中，当前代码如下：

```
#ifdef HAVE_FUNCTIONNAME
{'functionname', ... },
#endif /* HAVE_FUNCTIONNAME */
```

这时应将 `impl` 函数体用 `#ifdef` 包裹起来，如下所示：

```
#ifdef HAVE_FUNCTIONNAME
/*[clinic input]
module.functionname
...
[clinic start generated code]*/
static module_functionname(...)
{
```

(下页继续)

```
...
}
#endif /* HAVE_FUNCTIONNAME */
```

Then, remove those three lines from the PyMethodDef structure, replacing them with the macro Argument Clinic generated:

```
MODULE_FUNCTIONNAME_METHODDEF
```

(在生成的代码中可找到宏的真实名称。或者可以自行求一下值：块的第一行定义的函数名，句点改为下划线，全部大写，并在末尾加上 "_METHODDEF")

如果 HAVE_FUNCTIONNAME 未定义怎么办？那么 MODULE_FUNCTIONNAME_METHODDEF 宏也不会定义。

这正是 Argument Clinic 变聪明的地方。它其实能检测到 #ifdef 屏蔽了 Argument Clinic 块。于是会额外生成一小段代码，如下所示：

```
#ifndef MODULE_FUNCTIONNAME_METHODDEF
#define MODULE_FUNCTIONNAME_METHODDEF
#endif /* !defined(MODULE_FUNCTIONNAME_METHODDEF) */
```

这样宏总是会生效。如果定义了函数，则会转换为正确的结构，包括尾部的逗号。如果函数未定义，就不做什么转换。

不过，这导致了一个棘手的问题：当使用“block”输出预设时 Argument Clinic 应该把额外的代码放到哪里呢？它不能放在输出代码块中，因为它可能会被 #ifdef 停用。（它的作用就是这个！）

在此情况下，Argument Clinic 会将额外的代码的写入目标设为“buffer”。这意味着你可能会收到来自 Argument Clinic 的抱怨：

```
Warning in file "Modules/posixmodule.c" on line 12357:
Destination buffer 'buffer' not empty at end of file, emptying.
```

When this happens, just open your file, find the dump buffer block that Argument Clinic added to your file (it'll be at the very bottom), then move it above the PyMethodDef structure where that macro is used.

4.19 如何在 Python 文件中使用 Argument Clinic

实际上使用 Argument Clinic 来预处理 Python 文件也是可行的。当然使用 Argument Clinic 代码块并没有什么意义，因为其输出对于 Python 解释器来说是没有意义的。但是使用 Argument Clinic 来运行 Python 代码块可以让你将 Python 当作 Python 预处理器来使用！

由于 Python 注释不同于 C 注释，嵌入到 Python 文件的 Argument Clinic 代码块看起来会有一点不同。它们看起来像是这样：

```
#!/[python input]
#print("def foo(): pass")
#[python start generated code]*/
def foo(): pass
#!/[python checksum:...]*/*
```

Python 模块索引

C

`clinic`, 4

索引

非字母

`./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
命令行选项
 `--converters`, 3
 `-f`, 3
 `FILE`, 3
 `--force`, 3
 `-h`, 3
 `--help`, 3
 `--make`, 3
 `-o`, 3
 `--output`, 3
 `--srcdir`, 3
 `-v`, 3
 `--verbose`, 3
结束行, 3
输入, 3

B

`block`, 3

C

`c_default` (*clinic.CConverter* 属性), 4
`c_ignored_default` (*clinic.CConverter* 属性), 4
`CConverter` (*clinic* 中的类), 4
`checksum`, 3
`clinic`
 模块, 4
`converter` (*clinic.CConverter* 属性), 4
`--converters`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3

D

`default` (*clinic.CConverter* 属性), 4

F

`-f`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3
`FILE`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3
`--force`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3

H

`-h`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3
`--help`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3

I

M

`--make`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3

O

`-o`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3
`output`, 3
`--output`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3

P

`parse_by_reference` (*clinic.CConverter* 属性), 4
`py_default` (*clinic.CConverter* 属性), 4
Python 提高建议
 PEP 8, 11
 PEP 257, 5
 PEP 436, 2
 PEP 573, 18

S

`--srcdir`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3

T

`type` (*clinic.CConverter* 属性), 4

V

`-v`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3
`--verbose`
 `./Tools/clinic/clinic.py` `[-h]` `[-f]` `[-o-OUTPUT]` `[-v]` `[-i-include]` `[-r-reference (clinic.CConverter 属性), 4]` `[-m-make]` `[-s-srcdir]`
 命令行选项, 3
开始行, 3

W

`校验行`, 3
模块
 `clinic`, 4