

---

# Argument Clinic 的用法

发布 3.9.14

Guido van Rossum  
and the Python development team

九月 11, 2022

Python Software Foundation  
Email: docs@python.org

## Contents

|      |                                      |    |
|------|--------------------------------------|----|
| 1    | Argument Clinic 的设计目标                | 2  |
| 2    | 基本概念和用法                              | 2  |
| 3    | 函数的转换                                | 3  |
| 4    | 进阶                                   | 8  |
| 4.1  | 符号化默认值                               | 8  |
| 4.2  | 对 Argument Clinic 生成的 C 函数和变量进行重命名   | 8  |
| 4.3  | 利用 PyArg_UnpackTuple 对函数进行转换         | 9  |
| 4.4  | 可选参数组                                | 9  |
| 4.5  | 采用真正的 Argument Clinic 转换器，而不是“传统转换器” | 10 |
| 4.6  | Py_buffer                            | 12 |
| 4.7  | 高级转换器                                | 12 |
| 4.8  | 参数的默认值                               | 12 |
| 4.9  | 默认值 NULL                             | 13 |
| 4.10 | 设为默认值的表达式                            | 13 |
| 4.11 | 返回转换器                                | 13 |
| 4.12 | 复制已有的函数                              | 14 |
| 4.13 | 调用 Python 代码                         | 15 |
| 4.14 | 参数 self 转换器的用法                       | 15 |
| 4.15 | 编写自定义转换器                             | 16 |
| 4.16 | 编写自定义的返回转换器                          | 17 |
| 4.17 | METH_O 和 METH_NOARGS                 | 17 |
| 4.18 | tp_new 和 tp_init functions           | 17 |
| 4.19 | 改变和重定向 Clinic 的输出                    | 17 |
| 4.20 | #ifdef 使用技巧                          | 20 |
| 4.21 | 在 Python 文件中使用 Argument Clinic       | 21 |
|      | 索引                                   | 22 |

---

作者 Larry Hastings

## 摘要

Argument Clinic 是 CPython 的一个 C 文件预处理器。旨在自动处理所有与“内置”参数解析有关的代码。本文展示了将 C 函数转换为配合 Argument Clinic 工作的做法，还介绍了一些关于 Argument Clinic 用法的进阶内容。

目前 Argument Clinic 视作仅供 CPython 内部使用。不支持在 CPython 之外的文件中使用，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 语言扩展，欢迎在自己的代码中试用 Argument Clinic。但 Argument Clinic 与新版 CPython 中的版本可能完全不兼容，且会打乱全部代码。

## 1 Argument Clinic 的设计目标

Argument Clinic 的主要目标，是接管 CPython 中的所有参数解析代码。这意味着，如果要把某个函数转换为配合 Argument Clinic 一起工作，则该函数不应再作任何参数解析工作——Argument Clinic 生成的代码应该是个“黑盒”，CPython 会在顶部发起调用，底部则调用自己的代码，`PyObject *args`（也许还有 `PyObject *kwargs`）会神奇地转换成所需的 C 变量和类型。

Argument Clinic 为了能完成主要目标，用起来必须方便。目前，使用 CPython 的参数解析库是一件苦差事，需要在很多地方维护冗余信息。如果使用 Argument Clinic，则不必再重复代码了。

显然，除非 Argument Clinic 解决了自身的问题，且没有产生新的问题，否则没有人会愿意用它。所以，Argument Clinic 最重要的事情就是生成正确的代码。如果能加速代码的运行当然更好，但至少不应引入明显的减速。（最终 Argument Clinic 应该可以实现较大的速度提升——代码生成器可以重写一下，以产生量身定做的参数解析代码，而不是调用通用的 CPython 参数解析库。这会让参数解析达到最佳速度！）

此外，Argument Clinic 必须足够灵活，能够与任何参数解析的方法一起工作。Python 有一些函数具备一些非常奇怪的解析行为；Argument Clinic 的目标是支持所有这些函数。

最后，Argument Clinic 的初衷是为 CPython 内置程序提供内省“签名”。以前如果传入一个内置函数，内省查询函数会抛出异常。有了 Argument Clinic，再不会发生这种问题了！

在与 Argument Clinic 合作时，应该牢记一个理念：给它的信息越多，它做得就会越好。诚然，Argument Clinic 现在还比较简单。但会演变得越来越复杂，应该能够利用给出的全部信息干很多聪明而有趣的事情。

## 2 基本概念和用法

Argument Clinic 与 CPython 一起提供，位于 `Tools/clinic/clinic.py`。若要运行它，请指定一个 C 文件作为参数。

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

Argument Clinic 会扫描 C 文件，精确查找以下代码：

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

一旦找到一条后，就会读取所有内容，直至遇到以下代码：

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

这两行之间的所有内容都是 Argument Clinic 的输入。所有行，包括开始和结束的注释行，统称为 Argument Clinic “块”。

Argument Clinic 在解析某一块时，会生成输出信息。输出信息会紧跟着该块写入 C 文件中，后面还会跟着包含校验和的注释。现在 Argument Clinic 块看起来应如下所示：

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

如果对同一文件第二次运行 **Argument Clinic**，则它会丢弃之前的输出信息，并写入带有新校验行的输出信息。不过如果输入没有变化，则输出也不会后变化。

不应去改动 **Argument Clinic** 块的输出部分。而应去修改输入，直到生成所需的输出信息。（这就是校验和的用途——检测是否有人改动了输出信息，因为在 **Argument Clinic** 下次写入新的输出时，这些改动都会丢失）。

为了清晰起见，下面列出了 **Argument Clinic** 用到的术语：

- 注释的第一行 `/*[clinic input]` 是 起始行。
- 注释 (`[clinic start generated code]*/`) 的最后一行是 结束行。
- 最后一行 (`/*[clinic end generated code: checksum=...]*/`) 是 校验和行。
- 在起始行和结束行之间是 输入数据。
- 在结束行和校验和行之间是 输出数据。
- 从开始行到校验和行的所有文本，都是 块。（**Argument Clinic** 尚未处理成功的块，没有输出或校验和行，但仍视作一个块）。

### 3 函数的转换

要想了解 **Argument Clinic** 是如何工作的，最好的方式就是转换一个函数与之合作。下面介绍需遵循的最基本步骤。请注意，若真的准备在 CPython 中进行检查，则应进行更深入的转换，使用一些本文后续会介绍到的高级概念（比如“返回转换”和“自转换”）。但以下例子将维持简单，以供学习。

就此开始

0. 请确保 CPython 是最新的已签出版本。
1. 找到一个调用 `PyArg_ParseTuple()` 或 `PyArg_ParseTupleAndKeywords()`，且未被转换为采用 **Argument Clinic** 的 Python 内置程序。这里用了 `_pickle.Pickler.dump()`。
2. 如果对 `PyArg_Parse` 函数的调用采用了以下格式：

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

或者多次调用 `PyArg_ParseTuple()`，则应再选一个函数。**Argument Clinic** 确实支持上述这些状况。但这些都是高阶内容——第一次使用就简单一些吧。

此外，如果多次调用 `PyArg_ParseTuple()` 或 `PyArg_ParseTupleAndKeywords()` 且同一参数需支持不同的类型，或者用到 `PyArg_Parse` 以外的函数来解析参数，则可能不适合转换为 **Argument Clinic**。**Argument Clinic** 不支持通用函数或多态参数。

3. 在函数上方添加以下模板，创建块：

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

4. 剪下文档字符串并粘贴到 `[clinic]` 行之间，去除所有的无用字符，使其成为一个正确引用的 C 字符串。最有应该只留下带有左侧缩进的文本，且行宽不大于 80 个字符。（参数 `Clinic` 将保留文档字符串中的缩进。）

如果文档字符串的第一行看起来像是函数的签名，就把这一行去掉吧。（文档串不再需要用到它——将来对内置函数调用 `help()` 时，第一行将根据函数的签名自动建立。）

示例：

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

5. 如果文档字符串中没有“摘要”行，`Argument Clinic` 会报错。所以应确保带有摘要行。“摘要”行应为在文档字符串开头的一个段落，由一个 80 列的单行构成。

（示例的文档字符串只包括一个摘要行，所以示例代码这一步不需改动）。

6. 在文档字符串上方，输入函数的名称，后面是空行。这应是函数的 Python 名称，而且应是句点分隔的完整路径——以模块的名称开始，包含所有子模块名，若函数为类方法则还应包含类名。

示例：

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

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

7. 如果是第一次在此 C 文件中用到 `Argument Clinic` 的模块或类，必须对其进行声明。清晰的 `Argument Clinic` 写法应于 C 文件顶部附近的某个单独块中声明这些，就像 `include` 文件和 `statics` 放在顶部一样。（在这里的示例代码中，将这两个块相邻给出。）

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。

在声明某个类时，还必须指定其 C 语言类型的两个部分：用于指向该类实例的指针的类型声明，和指向该类的 `PyTypeObject` 指针。

示例：

```
/*[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]*/
```

8. 声明函数的所有参数。每个参数都应另起一行。所有的参数行都应对齐函数名和文档字符串进行缩进。

这些参数行的常规形式如下：

```
name_of_parameter: converter
```

如果参数带有默认值，请加在转换器之后：

```
name_of_parameter: converter = default_value
```

`Argument Clinic` 对“缺省值”的支持方式相当复杂；更多信息请参见[关于缺省值的部分](#)。

在参数行下面添加一个空行。

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.

每个参数都要从“PyArg\_Parse()”格式参数中复制其“格式单元”，并以带引号字符串的形式指定其转换器。（“格式单元”是 format 参数的 1-3 个字符的正式名称，用于让参数解析函数知晓该变量的类型及转换方法。关于格式单位的更多信息，请参阅 [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]*/
```

9. 如果函数的格式字符串包含 |，意味着有些参数带有缺省值，这可以忽略。Argument Clinic 根据参数是否有缺省值来推断哪些参数是可选的。

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` 两种格式字符串都没有，所以这里的示例不用改动。)

10. 如果 C 函数调用的是 `PyArg_ParseTuple()` (而不是 `PyArg_ParseTupleAndKeywords()`)，那么其所有参数均是仅限位置参数。

若要在 Argument Clinic 中把所有参数都标记为只认位置，请在最后一个参数后面一行加入一个 /，缩进程度与参数行对齐。

目前这个标记是全体生效；要么所有参数都是只认位置，要么都不是。（以后 Argument Clinic 可能会放宽这一限制。）

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]*/
```

11. 为每个参数都编写一个文档字符串，这很有意义。但这是可选项；可以跳过这一步。

下面介绍如何添加逐参数的文档字符串。逐参数文档字符串的第一行必须比参数定义多缩进一层。第一行的左边距即确定了所有逐参数文档字符串的左边距；所有文档字符串文本都要同等缩进。文本可以用多行编写。

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]*/

```

12. 保存并关闭该文件，然后运行 `Tools/clinic/clinic.py`。运气好的话就万事大吉——程序块现在有了输出信息，并且生成了一个 `.c.h` 文件！在文本编辑器中重新打开该文件，可以看到：

```

/*[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]*/

```

显然，如果 **Argument Clinic** 未产生任何输出，那是因为在输入信息中发现了错误。继续修正错误并重试，直至 **Argument Clinic** 正确地处理好文件。

为了便于阅读，大部分“胶水”代码已写入 `.c.h` 文件中。需在原 `.c` 文件中包含这个文件，通常是在 `clinic` 模块之后：

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

13. 请仔细检查 **Argument Clinic** 生成的参数解析代码，是否与现有代码基本相同。

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

其次，传给 `PyArg_ParseTuple()` 或 `PyArg_ParseTupleAndKeywords()` 的格式字符串应该完全与原有函数中的相同，直到冒号或分号为止。

(**Argument Clinic** 生成的格式串一定是函数名后跟着 `:`。如果现有代码的格式串以 `;` 结尾，这种改动不回影响使用，因此不必担心。)

第三，如果格式串需要指定两个参数（比如长度、编码字符串或指向转换函数的指针），请确保第二个参数在两次调用时完全相同。

第四，在输出部分会有一个预处理器宏，为该内置函数定义合适的静态 `PyMethodDef` 结构：

```

#define __PICKLE_PICKLER_DUMP_METHODDEF \
{"dump", (PyCFunction)__pickle_Pickler_dump, METH_O, __pickle_Pickler_dump__
↪ doc__},

```

此静态结构应与本内置函数现有的静态结构 `PyMethodDef` 完全相同。

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

14. 注意，输出部分的最后一行是“导入”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 所见的参数即为此导入函数的参数；如果实现代码给这些变量采用了不同的命名，请进行修正。

因为稍显怪异，所以还是重申一下。现在的代码应该如下所示：

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

上面是 Argument Clinic 生成的校验值和函数原型。函数应该带有闭合的大括号，里面包含实现代码。

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__()",
                     Py_TYPE(self)->tp_name);
        return NULL;
    }

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

    ...
}
```

15. 还记得用到 PyMethodDef 结构的宏吧？找到函数中已有的 PyMethodDef 结构，并替换为宏的引用。（如果函数是模块级的，可能会在文件的末尾附近；如果函数是个类方法，则可能会在靠近实现代码的下方。）

注意，宏尾部带有一个逗号。所以若用宏替换已有的静态结构 PyMethodDef 时，请勿在结尾添加



逗号了。

示例：

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

16. 编译，然后运行回归测试套件中的有关测试程序。不应引入新的编译警告或错误，且对 Python 也不应有外部可见的变化。

差别只有一个，即 `inspect.signature()` 运行于新的函数上，现在应该新提供一个有效的签名！祝贺你，现在已经用 Argument Clinic 移植了第一个函数。

## 4 进阶

现在 Argument Clinic 的使用经验已具备了一些，该介绍一些高级内容了。

### 4.1 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。
- 以模块名开头的简单符号常量，如 `sys.maxsize`。

如果你感到好奇，这是在 `from_builtin()` (`Lib/inspect.py`) 中实现的。

（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。）

### 4.2 对 Argument Clinic 生成的 C 函数和变量进行重命名

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"`，并用作导入函数的名称。

例如，若对 `pickle.Pickler.dump` 生成的 C 函数进行重命名，应如下所示：

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

```

原函数会被命名为 `pickler_dumper()`，而实现函数现在被命名为 `"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，语法依然是利用 `"as"`：

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

obj: object
file as file_obj: object

```

（下页继续）



```
protocol: object = NULL
*
fix_imports: bool = True
```

这里 Python (签名和 keywords 数组中) 中用的名称是 `file`, 而 C 语言中的变量命名为 `file_obj`。  
`self` 参数也可以进行重命名。

### 4.3 利用 `PyArg_UnpackTuple` 对函数进行转换

若要将函数转换为采用 `PyArg_UnpackTuple()` 解析其参数, 只需写出所有参数, 并将每个参数定义为 `object`。可以指定 `type` 参数, 以便能转换为合适的类型。所有参数都应标记为只认位置 (在最后一个参数后面加上 `/`)。

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

### 4.4 可选参数组

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。

虽然这种函数通常可以转换为采用 `PyArg_ParseTupleAndKeywords()`、可选参数和默认值的方式, 但并不是全都可以做到。这些过时函数中, `PyArg_ParseTupleAndKeywords()` 并不能直接支持某些功能。最明显的例子是内置函数 `range()`, 它的必需参数的左边存在一个可选参数! 另一个例子是 `curses.window.addch()`, 它的两个参数是一组, 必须同时指定。(参数名为 `x` 和 `y`; 如果调用函数时传入了 `x`, 则必须同时传入 `y`; 如果未传入 `x`, 则也不能传入 `y`)。

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。

---

**注解:** 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。

---

若要定义可选组, 可在要分组的参数前面加上 `[`, 在这些参数后加上 `]`, 要在同一行上。举个例子, 下面是 `curses.window.addch` 采用可选组的用法, 前两个参数和最后一个参数可选:

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

[
    x: int
        X-coordinate.
    y: int
        Y-coordinate.
]

ch: object
    Character to add.

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

注释:

- 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。

## 4.5 采用真正的 Argument Clinic 转换器，而不是“传统转换器”

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”只是一种简使用法，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。
- 有些格式符是“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化器。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。

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** 一组 Python 类型（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** 只支持无符号整数。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** 只有 object 转换器才支持。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** 只支持字符串。指定将 Python `str(Unicode)` 转换为 C 语言的 `char *` 时要使用的编码。

**subclass\_of** 只有 object 转换器才支持。要求 Python 值是 Python 类型的子类，用 C 语言表示。

**type** 只有 object 和 self 转换器才支持。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。

请注意，并不是所有参数的组合都能正常生效。通常这些参数是由相应的 PyArg\_ParseTuple 格式化单元实现的，行为是固定的。比如目前不能不指定 bitwise=True 就去调用 unsigned\_short。虽然完全有理由认为这样可行，但这些语义并没有映射到任何现有的格式化单元。所以 Argument Clinic 并不支持。（或者说，至少目前还不支持。）

下表显示了传统转换器映射到实参转换器的情况。左边是传统转换器，右边是要替换它的文本。

|       |   |
|-------|---|
| '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'   | Py_UNICODE  |
| 'u#'  | Py_UNICODE(zeroes=True)   |
| 'w*'  | Py_buffer(accept={rwbuffer})  |
| 'Y'   | PyByteArrayObject   |
| 'y'   | str(accept={bytes})   |
| 'y#'  | str(accept={robuffer}, zeroes=True)   |
| 'y*'  | Py_buffer   |
| 'Z'   | Py_UNICODE(accept={str, NoneType})  |
| 'Z#'  | Py_UNICODE(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。

## 4.6 Py\_buffer

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）。

## 4.7 高级转换器

还记得初次尝试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。

其实诀窍在于，这些格式单元都需要给出参数——要么是转换函数，要么是类型，要么是指定编码的字符串。（但“传统转换器”不支持参数。这就是为什么第一个函数要跳过这些内容）。为格式单元指定的参数于是就成了转换器的参数；参数可以是 `converter``（对于 ``O&`）、`subclass_of``（对于 ``O!`），或者是 `encoding`（对于 `e` 开头的格式单元）。

在使用 `subclass_of` 时，可能还需要用到 `object()` 的另一个自定义参数：`type`，用于设置参数的实际类型。例如，为了确保对象是 `PyUnicode_Type` 的子类，可能想采用转换器 `object(type='PyUnicodeObject *', subclass_of='&PyUnicode_Type')`。

**Argument Clinic** 用起来可能存在问题：丧失了 `e` 开头的格式单位的一些灵活性。在手工编写 `PyArg_Parse` 调用时，理论上可以在运行时决定传给 `PyArg_ParseTuple()` 的编码字符串。但现在这个字符串必须在 **Argument-Clinic** 预处理时进行硬编码。这个限制是故意设置的；以便简化对这种格式单元的支持，并允许以后进行优化。这个限制似乎并不合理；CPython 本身总是为 `e` 开头的格式单位参数传入静态的硬编码字符串。

## 4.8 参数的默认值

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，后续章节中将会介绍。

## 4.9 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`。

## 4.10 设为默认值的表达式

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` 在不同的系统平台可能有不同的值。因此，**Argument Clinic** 不能简单地在本底环境对表达式求值并用 C 语言硬编码。所以默认值将用表达式的方式存储下来，运行的时候在请求函数签名时会被求值。

在对表达式进行求值时，可以使用什么命名空间呢？求值过程运行于内置模块的上下文中。因此，如果模块带有名为 `max_widgets` 的属性，直接引用即可。

```
foo: Py_ssize_t = max_widgets
```

如果表达式不在当前模块中，就会去 `sys.modules` 查找。比如 `sys.maxsize` 就是如此找到的。（因为事先不知道用户会加载哪些模块到解释器中，所以最好只用到 Python 会预加载的模块。）

仅当运行时才对缺省值求值，意味着 **Argument Clinic** 无法计算出正确的 C 缺省值。所以需显式给出。在使用表达式时，必须同时用转换器的“`c_default`”参数指定 C 语言中的等价表达式。

```
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：

- 功能调用
- 行内 if 语句 (`3 if foo else 5`)
- 序列自动解包 (`*[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

## 4.11 返回转换器

**Argument Clinic** 生成的植入函数默认会返回 `PyObject *`。但是通常 C 函数的任务是要对某些 C 类型进行计算，然后将其转换为 `PyObject *` 作为结果。**Argument Clinic** 可以将输入参数由 Python 类型转换为本地 C 类型——为什么不让它将返回值由本地 C 类型转换为 Python 类型呢？

这就是“返回转换器”的用途。它将植入函数修改成返回某种 C 语言类型，然后在生成的（非植入）函数中添加代码，以便将 C 语言值转换为合适的 `PyObject *`。

返回值转换器的语法与参数转换器的类似。返回值转换器的定义方式，类似于函数返回值的注解。返回值转换器的行为与参数转换器基本相同，接受参数，参数只认关键字，如果不修改默认参数则可省略括号。

（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。）

返回值转换器还存在一个复杂的问题：出错信息如何表示？通常函数在执行成功时会返回一个有效（非 NULL）指针，失败则返回 NULL。但如果使用了整数的返回值转换器，所有整数都是有效值。**Argument Clinic** 怎么检测错误呢？解决方案是：返回值转换器会隐含寻找一个代表错误的特殊值。如果返回该特殊值，且设置了出错标记（`PyErr_Occurred()` 返回 `True`），那么生成的代码会传递该错误。否则，会对返回值进行正常编码。

目前 **Argument Clinic** 只支持少数几种返回值转换器。

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

这些转换器都不需要参数。前 3 个转换器如果返回 -1 则表示出错。`DecodeFSDefault` 的返回值类型是 `const char *`；若返回 NULL 指针则表示出错。

(还有一个 `NoneType` 转换器是实验性质的，成功时返回 `Py_None`，失败则返回 NULL，且不会增加 `Py_None` 的引用计数。此转换器是否值得适用，尚不明确)。

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。

## 4.12 复制已有的函数

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，没有什么语法可对函数进行部分克隆或克隆后进行修改。克隆要么全有要么全无。

另外，要克隆的函数必须在当前文件中已有定义。



## 4.13 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:...]*/
```

## 4.14 参数 self 转换器的用法

Argument Clinic 用一个默认的转换器自动添加一个“self”参数。自动将 self 参数的 type 设为声明类型时指定的“指向实例的指针”。不过 Argument Clinic 的转换器可被覆盖，也即自己指定一个转换器。只要将自己的 self 参数作为块的第一个参数即可，并确保其转换器是 self\_converter 的实例或其子类。

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。

如何指定 self 对应的自定义类型呢？如果只有 self 类型相同的一两个函数，可以直接使用 Argument Clinic 现有的 self 转换器，把要用的类型作为 type 参数传入：

```
/*[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]*/
```

如果有很多函数将使用同一类型的 self，则最好创建自己的转换器，继承自 self\_converter 类但要覆盖其 type 成员：

```
/*[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]*/
```



## 4.15 编写自定义转换器

上一节中已有提及……可以编写自己的转换器！转换器就是一个继承自“CConverter”的 Python 类。假如有个参数采用了 O& 格式，对此参数进行解析就会去调用某个“转换器函数”PyArg\_ParseTuple()，也就会用到自定义转换器。

自定义转换器类应命名为 \*something\*\_converter。只要按此规则命名，自定义转换器类就会在 Argument Clinic 中自动注册；转换器的名称就是去除了 \_converter 后缀的类名。（通过元类完成）。

不得由 CConverter.\_\_init\_\_ 派生子类。而应编写一个 converter\_init() 函数。converter\_init() 必须能接受一个 self 参数；所有后续的其他参数必须是只认关键字的参数。传给 Argument Clinic 转换器的所有参数都会传入自定义 converter\_init() 函数。

CConverter 的其他一些成员，可能需要在自定义子类中定义。下面列出了目前的成员：

**type** 变量要采用的 C 语言数据类型。type 应为 int 之类的 Python 字符串，用于指定变量的类型。若为指针类型，则字符串应以 ' \*' 结尾。

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

**py\_default** 用 Python 代码表示的 default，为字符串类型。若无缺省值，则为 None。

**c\_default** 用 C 代码表示的 default，为字符串类型。若无缺省值，则为 None。

**c\_ignored\_default** 在无缺省值时用于初始化 C 变量的缺省值，因为不指定缺省值可能会引发“变量未初始化”的警告。在用到多组可选项的时候很容易发生这种情况，尽管好的代码实际不会用到这个值，但确实会给 impl 传入本值，而 C 编译器则会认为“用到”了未初始化的值。应确保本值为非空字符串。

**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** 布尔值。如果为真，则 Argument Clinic 在将变量传入 PyArg\_ParseTuple 时，会在变量名前加上一个 ``&` `()。

下面是最简单的自定义转换器示例，取自 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 to Argument Clinic named `ssize_t`. Parameters declared as `ssize_t` will be declared as type `Py_ssize_t`, and will be parsed by the 'O&' format unit, which will call the `ssize_t_converter` converter 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。

## 4.16 编写自定义的返回转换器

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。

返回值转换器必须是 `CReturnConverter` 的子类。因为自定义的返回值转换器还没有广泛应用，目前还没有示例。若要编写返回值转换器，请阅读“Tools/clinic/clinic.py”，特别是 `CReturnConverter` 及其所有子类的实现代码。

## 4.17 METH\_O 和 METH\_NOARGS

若要用 `METH_O` 对函数进行转换，请确保对该函数的单个参数使用 `object` 转换器，并将参数标为只认位置的：

```
/*[clinic input]
meth_o_sample

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

若要用 `METH_NOARGS` 对函数进行转换，只需不定义参数即可。

依然可以采用一个 `self` 转换器、一个返回值转换器，并为 `METH_O` 的对象转换器指定一个 `type` 参数。

## 4.18 tp\_new 和 tp\_init functions

`tp_new` 和 `tp_init` 函数也可以转换。只要命名为 `__new__` 或 `__init__` 即可。注意：

- 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。
- 转换这些函数不会生成 `PyMethodDef`、`#define`。
- `__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）。

## 4.19 改变和重定向 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，可能会不方便使用。幸好可以对 `Clinic` 进行配置：可以将输出结果缓存起来以供打印，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：

**field** 在当前上下文中，区块是指 `Clinic` 输出的一个小节。例如，`PyMethodDef` 结构的 `#define` 是一个区块，名为 `methoddef_define`。`Clinic` 在每个函数定义中可以输出 7 个区块。

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

区块均以 "<a>\_<b>" 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），"<b>" 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。）

**目的地 (destination)** 目的地是 Clinic 可以进行输出的地方。内置的输出目的地有 5 种：

**block** 默认目的地：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** 单独的“Clinic 文件”，由 Clinic 自动创建。文件名会是“{basename}.clinic{extension}”，这里的 basename 和 extension 即为对当前文件运行 `os.path.splitext()` 后的结果。（比如：\_pickle.c 的 file 目的地将会是 \_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``PyTypeObject 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。文件目标需要多一个参数，用于生成文件名的模板，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。

## 4.20 #ifdef 使用技巧

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(...)
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

然后，从 PyMethodDef 结构中删除以下 3 行，替换成 Argument Clinic 生成的宏：

```
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.
```

当发生这种问题时，只需打开你的文件，找到由 **Argument Clinic** 添加到你的文件的 `dump buffer` 代码块（它将位于文件末尾），并将其移到使用了那个宏的 `PyMethodDef` 结构体之上。

## 4.21 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:...]*/
```

## 索引

### P

Python 提高建议  
PEP 8, [10](#)