BaneWfn

Multiwfn 工作流调度程序

https://bane-dysta.top/software/banewfn

version： 1.3

Bane QC Project

目录

本文档默认读者已经了解 Multiwfn 的基本用途，但不假定读者已经接触过 BaneWfn。第一次使用时，建议先阅读“总览”“运行时配置”和“输入文件与 DSL”三章，再结合“示例”动手验证；如果你已经在项目中使用过 BaneWfn，则可以把本手册作为语法、行为和命名规则的系统参考。

1 总览

1.1 定位

BaneWfn 是一个面向 Multiwfn 的模块化工作流程序。它并不替代 Multiwfn 本身，也不额外实现量子化学分析算法，而是把 Multiwfn 中那些固定、重复、容易出错的操作路径抽象为可复用脚本，从而把“手工菜单操作”转化为“可维护、可共享、可批量执行的工作流”。

在设计上，BaneWfn 以文本脚本为核心：输入文件负责描述任务顺序、变量和批处理逻辑，`.conf` 模块配置负责封装常用菜单路径，`%raw` 与 `%command` 则分别承担原始 Multiwfn 输入和后处理命令。借助这一组合方式，用户既可以沉淀稳定流程，也可以在同一份脚本中插入一次性的临时补丁，而不必为了少量例外单独维护另一套流程。

从使用体验来看，BaneWfn 更像是一层“工作流胶水”。它把 Multiwfn、shell / batch 命令以及项目中的命名规范、归档习惯和批量执行需求连接在一起，适合用于实验室内部复用、项目归档和跨平台共享。

1.2 适用场景

BaneWfn 特别适合以下几类工作场景：

- 当同一套 Multiwfn 分析需要在大量 `fchk`、`wfn`、`log`、`cub` 等输入文件上重复执行时，可以把原本的手工流程整理为统一脚本，从而显著减少重复劳动。
- 当分析流程由多个固定菜单操作组成，并且这些操作会在不同项目中反复出现时，可以通过模块配置把流程沉淀为可复用步骤。
- 当工作流中需要穿插 shell / batch 命令，对 Multiwfn 生成的 `cub`、文本、图像或中间文件做重命名、归档和二次处理时，BaneWfn 可以把计算和后处理收拢到同一份脚本中。
- 当一个分析过程需要在多个输入文件之间反复切换，例如先生成 `cub` 或 `fch` 工件，再以这些工件作为后续任务输入时，BaneWfn 可以把这种“分阶段切换输入”的逻辑写成显式、可追踪的脚本步骤。
- 当一个分析过程有非常繁琐的前置输入，且后处理未必能一次完成时（典型场景如绘制填色图），BaneWfn 可以执行完预定繁琐冗长的命令后将控制权交还给用户，让用户直接开始后处理。

1.3 核心能力

BaneWfn 的价值不在于增加新的计算功能，而在于把已有分析能力组织得更稳定、更可复用。具体而言，它提供了以下几项核心能力：

- 它使用统一 DSL 描述模块块、步骤块、原始命令块和命令块，使脚本既能表达结构化流程，也能容纳少量例外操作。
- 它允许通过 `.conf` 模块配置文件把常用 Multiwfn 菜单路径封装为步骤，从而将经验性操作沉淀为可共享模板。

- 它支持 %raw 原始命令块，因此即便某个流程尚未被模块化，也可以直接在脚本中写入原始 Multiwfn 输入序列。
- 它支持 %command 后处理块，可以把重命名、移动文件、调用 Python / VMD / gnuplot 等操作紧接在计算结果之后执行。
- 它支持自定义变量、数组变量、列展开变量和交互式变量，能够把批量分析、参数扫描以及单轮内部的多值展开纳入同一工作流框架。
- 它支持 wfn_rebase=，因此可以在同一脚本中显式切换后续任务使用的输入文件，而不必拆分成多个独立脚本。
- 它支持 inline conf 与 bwpack 打包，有利于单文件分发、归档和跨机器迁移。
- 它支持 Linux 与 Windows；在 Windows 下，还可以选择通过 Git Bash 执行 #!/bin/bash 风格的 %command 块。

2 运行流程

2.1 标准执行流程

理解 BaneWfn 的执行顺序，对编写稳定脚本非常重要。总体上，程序会先完成“配置解析”，再完成“任务展开”，最后进入“逐轮执行”。其标准流程如下：

1. 程序首先查找并读取 banewfn.rc，从中确定 Multiwfn_exec、confpath、默认核心数以及可选的 gitbash_exec。
2. 随后解析输入文件，读取 wfn、core、dryrun、nogui 等头部保留项，以及自定义变量和 wfn_rebase 指令。
3. 输入文件中的任务块会被整理为内部任务序列，包括 module 块、独立 %raw 块和独立 %command 块。
4. 程序会合并命令行变量与文件中的自定义变量；若存在 var=?、var*=? 或 len(var)=? 形式的交互式变量，则会在运行时提示用户输入。
5. 对于脚本中实际引用到的模块，程序会先检查输入文件末尾是否包含对应的 inline conf；若存在，则优先加载内嵌配置，否则从 confpath 读取外部 .conf 文件。
6. 程序会展开波函数文件通配符，并根据数组变量生成变量组合。由此得到的每一轮执行，都对应一个“输入文件 × 变量组合”的具体实例。
7. 在每一轮实例中，程序会先做输入文件侧占位符替换，再根据模块配置展开出最终的 Multiwfn 命令序列。
8. 对于需要调用 Multiwfn 的任务，程序会根据块的收尾方式选择 end（非交互模式）或 wait（交互模式）执行。
9. 当某个任务成功结束且带有 %command 时，程序会继续执行该任务对应的后处理命令。

换句话说，BaneWfn 的核心思路是：先把脚本“解释清楚”，再把它“展开成一组具体任务”，最后按顺序逐一执行。只要理解了这三层关系，脚本中的变量替换、批处理、命令命名和 wfn_rebase 行为就会变得更容易预测。

2.2 循环展开规则

批量执行时，最容易混淆的是“文件循环”和“变量循环”的嵌套顺序。当前版本的规则是：**外层先按波函数文件列表循环，内层再按数组变量组合循环**。也就是说，程序会先固定一个输入文件，再在该文件上依次跑完所有变量组合，然后才切换到下一个文件。

当前版本的循环顺序可概括如下：

1. 先展开 `wfn=` 或 `-w` 指定的文件列表。
2. 再基于数组变量生成变量组合；如果有多个数组变量，则先求笛卡尔积。(注意，使用多组数组变量会使得计算的任务数激增！)
3. 对于每一个“文件 × 变量组合”，都从头到尾执行整份输入文件中的任务序列。

例如，当前目录下存在 `m1.fchk` 和 `m2.fchk`，输入文件为：

```
wfn=*.fchk
state=(1 2)
[hole-ele]
state ${state}
%process
cub
%command
mv hole.cub ${input}_${state}_hole.cub
mv electron.cub ${input}_${state}_ele.cub
end
```

那么执行顺序将是：先对 `m1.fchk` 执行 `state=1` 和 `state=2` 两轮空穴电子分析，再对 `m2.fchk` 执行 `state=1` 和 `state=2` 两轮空穴电子分析。换言之，实际顺序为 `m1/state=1 -> m1/state=2 -> m2/state=1 -> m2/state=2`。

3 安装与运行依赖

3.1 编译安装

3.1.1 构建依赖

构建 BaneWfn 只需要一个正常的 C++17 编译环境。通常来说，你需要准备以下条件：

- 支持 C++17 的编译器；
- CMake 3.10 或更高版本，或者 GNU Make 构建环境；
- Linux 或 Windows 运行环境。

3.1.2 运行时依赖

BaneWfn 本身不实现 Multiwfn 中的分析算法，因此在运行阶段需要依赖外部程序和输入数据。一个可用的运行环境通常至少应满足以下条件：

- 系统中可以访问到 Multiwfn 可执行文件；
- 模块配置可以从 `confpath` 找到，或者已经以内嵌形式打包到输入文件末尾；
- 如果 `%command` 中调用了额外工具，例如 shell、batch、Python、VMD、gnuplot 等，则这些工具也需要在目标环境中可用。

3.1.3 CMake 构建

如果你使用 CMake，推荐的构建方式如下：

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build -j
```

当前 CMake 工程具备以下特性：

- 语言标准为 C++17；
- BUILD_STATIC 用于控制静态链接，默认开启；
- BANEWFN_BUILD_TESTS 用于控制单元测试构建，默认关闭；
- 默认会生成两个可执行文件：banewfn 与 bwpack；
- 二进制输出目录为 build/bin/。

对于大多数使用者来说，CMake 构建适合做跨平台开发、调试和测试，因为它对构建目录、测试开关和编译选项的组织更清晰。

3.1.4 Makefile 构建

如果你更偏好直接使用 Makefile，也可以采用如下命令：

```
make linux
make windows
make both
```

当前 Makefile 额外支持以下目标或变量：

- make install：安装可执行文件和配置文件；
- INSTALL_BINDIR：安装二进制目录，默认 /usr/local/bin；
- INSTALL_CONFDIR：安装配置目录，默认 ~/.bane/wfn；
- CORES：安装时写入 banewfn.rc 的默认核心数。

如果你的使用场景以本机部署为主，Makefile 通常足够直接；如果需要更细粒度地控制构建选项或集成 CI，CMake 会更合适。

3.1.5 测试

当你需要运行单元测试时，可以使用以下命令：

```
cmake -S . -B build -DBANEWFN_BUILD_TESTS=ON
cmake --build build
ctest --test-dir build --output-on-failure
```

3.2 预编译版安装

请前往软件主页按照指引找到最新版下载方式。在一个完整的安装流程中，通常会得到以下几类产物：

- 主程序 banewfn；
- 打包工具 bwpack；
- 配置目录中的示例模块配置文件 *.conf；
- 默认的 banewfn.rc 配置文件。

4 运行时配置

4.1 推荐配置目录布局

为了降低项目维护成本，建议把运行配置和模块配置集中放在同一目录中。一个典型布局如下：

```
~/.bane/wfn/
  banewfn.rc
  aromatic.conf
  bond_order.conf
  ...
```

这种布局的优点在于：主程序配置和模块配置位于同一层级，查找规则清晰，跨项目复用时也更容易复制和备份。如果你所在团队已经形成了统一的 `.conf` 库，那么把它们集中到 `confpath` 指向的目录里，会比散落在各个项目目录中更便于维护。

4.2 `banewfn.rc`

4.2.1 主要结构

尽管文件扩展名是 `rc`，但 `banewfn.rc` 的实际格式更接近简单的 `key=value` 配置文件。它主要负责告诉程序“Multiwfn 在哪里”“模块配置目录在哪里”“默认并行数是多少”，以及在 Windows 下是否需要通过 Git Bash 执行 bash 风格命令块。一个最小可用示例如下：

```
Multiwfn_exec=Multiwfn
confpath=~/.bane/wfn
cores=8
# Windows 可选
# gitbash_exec="C:\Program Files\Git\bin\bash.exe"
```

各字段含义如下：

- `Multiwfn_exec`：必需。用于指定 Multiwfn 可执行文件路径或命令名，并支持 `~`、`$HOME`、`${HOME}` 等主目录写法。
- `confpath`：可选。用于指定模块 `.conf` 所在目录；未设置时默认使用 `~/.bane/wfn`。
- `cores`：可选。用于指定默认核心数；命令行 `-c` 和输入头 `core=` 都可以覆盖它。
- `gitbash_exec`：可选。仅在 Windows 下使用；当 `%command` 首行为 `#!/bin/bash` 时，程序会通过这里指定的 Git Bash 去执行命令块。

从职责划分看，`banewfn.rc` 更像是“运行环境配置”，而不是某个具体工作流的组成部分。也正因为如此，建议把它看作机器级配置，而把项目相关变量放在 `.bw/.inp` 脚本中维护。

4.2.2 查找顺序

`banewfn.rc` 按以下顺序查找：

1. 当前目录：`./banewfn.rc`
2. 可执行文件所在目录：`<exe_dir>/banewfn.rc`
3. 用户目录：`~/.bane/wfn/banewfn.rc`

当主程序找不到该文件时会终止运行。bwpack 在未显式指定 --confdir 时，也会尝试使用同一查找顺序；若仍未找到配置，则回退到 ~/.bane/wfn 作为默认配置目录。

如果你想在某个项目中临时覆盖全局配置，只需要把一个项目专用的 banewfn.rc 放到当前目录即可，而不必修改用户目录下的通用配置。

4.2.3 注释与路径规则

banewfn.rc 与 .conf 共享同一套注释和路径展开规则：

- 引号外的 # 视为注释起始符；
- 单引号或双引号内部的 # 会保留为字面字符，尽管路径里不会有 # 号。
- 在引号外写 \# 可以保留字面 #，能在 banewfn.rc 里用到这个的也是属于神人了。
- ~、$HOME 和 ${HOME} 会在配置读取时展开为用户主目录。

4.3 配置文件

4.3.1 格式

模块配置文件是纯文本 .conf 文件，用于定义某个模块如何进入主菜单、如何执行步骤，以及在非交互模式下如何退出。一个较完整的结构如下：

```
[main]
命令序列...

[process_name]
命令序列...

[process2_name]
命令序列...

[quit]
退出命令序列...
```

各段语义如下：

- [main]：进入模块主逻辑时固定追加的 Multiwfn 命令序列；
- [process_name]：供 %process 中某一步调用的命令序列；
- [quit]：任务以 end 收尾时自动追加的退出序列。

4.3.2 模板参数与占位符

.conf 中的命令本质上都是模板。它们不会在读取配置文件时立刻定值，而是要等到输入文件调用模块或步骤时，才根据传入参数完成替换。因此，理解 .conf 的关键，不只是“某一步会执行哪些命令”，还包括“这些命令预留了哪些参数入口”。

模板侧支持以下占位符写法：

写法	含义
$name / ${name}	使用参数 name 的值。

写法	含义
`${name:-default}`	当参数 name 未设置或为空时，使用默认值 default。
`${name*}`	把列表参数 name 逐项展开为多条命令。
`${len(name)}`	读取列表参数 name 的元素个数。
`${name*:- (a b c)}`	当列表参数 name 不存在时，使用一个临时列表默认值。

其中，`${name*}` 与 `${len(name)}` 读取的是“列表参数”。这个列表既可以直接由 `name*=(a b c)` 传入，也可以由 name1、name2、name3 这一组编号参数拼出来。程序会按编号顺序收集这些值，因此模板作者不必关心列表究竟来自哪种写法，只需要按统一占位符去读取即可。

例如，若某一步收到的参数中存在 `frag*=(2 5 9)`，或者等价地存在 `frag1=2`、`frag2=5`、`frag3=9`，那么：

```
show ${frag*}
count ${len(frag)}
```

会展开为：

```
show 2
show 5
show 9
count 3
```

参数来源的优先级如下：

1. 当前输入文件中传给模块或步骤的参数；
2. 以 `key=value` 形式写在当前 section 的 `-default-` 块中的键值（legacy）；
3. 模板内 `${name:-default}` 写法提供的内联默认值。

这种设计使得 `.conf` 可以保持通用，而具体脚本只需要按需覆盖少数参数。通常建议把“模块作者认为合理的默认值”写进模板默认值，把“项目级差异化参数”留给输入文件传入。

4.3.3 约定与建议

为了保持模块之间的可组合性，建议每个 `[step_name]` 在执行结束后都回到 `[main]` 进入的主逻辑界面。这样做的好处是，多个步骤可以在同一模块块中顺序串联，而不需要每做完一步就重新走完整个 Multiwfn 菜单路径。

此外，建议在 `.conf` 中把“稳定、可复用”的菜单路径写成标准 section，把临时补丁保留给 `%raw`。这样既能保证模块配置本身清晰简洁，又不会为了少量特殊情况把通用模板写得过于臃肿。

5 输入文件与 DSL

5.1 输入文件类型

主程序对文件扩展名本身没有语义分支，所有扩展名都使用同一套 DSL 语法。约定如下：

扩展名	用途
`.inp`	普通工作流输入文件。
`.bw`	可复用脚本文件。
`.bwc`	已打包 inline conf 的单文件脚本。

实际项目中，三者的差别更多体现在交付方式而不是语法本身：`.inp` 多作脚本产生的临时产物，`.bw` 更像可维护的脚本源文件，而 `.bwc` 则适合做归档和单文件分享。

5.2 整体结构

一份输入文件通常由三部分组成：文件头中的保留项与自定义变量、一个或多个任务块，以及可选的 inline conf 尾块。最推荐的写法，是把“环境与变量”集中写在上方，把“任务流程”集中写在中间，把“内嵌配置”统一放到文件末尾，这样最便于长期维护。

示例如下：

```
core=8
state=(1 2 3)

[excit]
%process
    nto state ${state}
%command
#!/bin/bash
mv ex_${state}.fchk ${input}_NTO${state}.fch
mkdir -p ${input}_NTOs
mv ${output} ${input}_NTOs
end

wfn_rebase=${input}_NTO${state}.fch

%process
    orb index h
    orb index l
%command
#!/bin/bash
mkdir -p ${input}_NTOs
mv ${input}_NTO${state}.fch ${input}_NTOs
mv ${output} ${input}_NTOs
mv h.cub ${input}_NTOs/${input}_NTO${state}_oH.cub
mv l.cub ${input}_NTOs/${input}_NTO${state}_oL.cub
echo "vmd -e orb.vmd" > ${input}_NTOs/orb.bat
echo "vcube *.cub" > ${input}_NTOs/orb.vmd
end

%command
cd NTOs
call orb.bat
```

```
end
```

5.3 注释规则

在普通输入上下文中，注释规则与配置文件一致：

- 引号外的 # 及其后内容会被去除；
- 引号内的 # 会保留；
- \# 在引号外会保留为字面 #。

但在 %raw 与 %command 中，情况完全不同。这两个块是**字面块（literal block）**，其内容会按原样保留：

- 空行会被保留；
- 形如 # comment 的行也会被保留；
- %raw 中的 # ... 会直接送入 Multiwfn，而不会被当作输入文件注释。

因此，若你只是想给脚本加说明，请把注释写在块外；若把注释放进 %raw，程序就会把它当作 Multiwfn 输入的一部分。

5.4 输入文件头

输入文件头通常位于第一个任务块之前，用来声明当前脚本的默认输入、默认并行数以及一些执行开关。当前保留项如下：

```
wfn=*.fchk
core=8
dryrun=on
nogui=true
wfn_rebase=next.fchk
```

字段说明如下：

写法	说明
wfn=<path-or-pattern>	当前脚本默认的波函数文件或通配符模式。
core=<N>	当前脚本默认核心数。
dryrun=<bool>	设为真时启用测试运行。
nogui=<bool>	设为真时向 Multiwfn 启动命令追加 -silent。

其中，wfn_rebase=<path> 比较特殊。它不是单纯的文件头配置，而是一个可以出现在块间的流程指令，用于临时切换后续块使用的输入文件。

Note: 与 wfn 不同，wfn_rebase 不支持通配符。

除保留项外，块外的其他 key=value 会被解析为自定义变量。例如：

```
prefix=result
state=(1 2 3)
answer=?
```

相关规则如下：

- 变量名只能包含字母、数字和下划线；
- 如前所述，`wfn`、`core`、`wfn_rebase`、`dryrun`、`nogui` 不能作为自定义变量名；
- 变量值不可留空；若希望运行时询问，请使用 `?`；
- 从可维护性角度出发，建议把自定义变量集中写在文件头部，而不要零散分布在任务之间。

5.5 输入侧变量引用

输入文件中的变量替换，作用于模块参数、`%process` 参数、`%raw`、`%command` 与 `wfn_rebase=`。它总是基于“当前输入文件 × 当前变量组合”这个具体实例来求值：也就是说，先确定这一轮处理的是哪个波函数文件、数组变量当前取到哪个元素，然后才开始替换文本。

输入侧支持以下写法：

写法	含义
`$name` / `${name}`	读取当前实例中的普通变量。
`${name:-default}`	当变量 `name` 未设置或为空时，使用默认值 `default`。
`$input` / `${input}`	当前 Multiwfn 输入文件的基名，去掉路径和扩展名。
`$wfn` / `${wfn}`	当前 Multiwfn 输入文件的完整路径。
`${output}`	预留给 `%command` 在执行前替换为当前块的 `.out` 文件名。

除此之外，带花括号的 `${name}` 在未命中变量时，还会继续尝试读取当前工作目录下同名文件的内容，并去掉首尾空白后作为值。这种写法尤其适合接入由外部流程临时生成的参数文件，例如拟合得到的平面向量或一段待插入的文本。

输入侧的取值优先级如下：

1. 命令行 `-v/--var`
2. 输入文件头部自定义变量
3. 特殊变量 `wfn` 与 `input`
4. `${name}` 形式的同名文件读取

5.5.1 `${output}` 的取值规则

`${output}` 比较特殊。它不会在脚本解析阶段提前定值，而是要等到对应的 Multiwfn 任务真正执行完成后，才会在 `%command` 中得到实际文件名。它代表当前块对应的 Multiwfn 输出文件名，且并不是任何时候都有值。其行为如下：

场景	`${output}` 值
非交互模式且未使用 `--screen`	当前 Multiwfn 输出文件名，例如 `fmo_sample.out`。
`wait` 交互模式	空字符串。
`--screen` 模式	空字符串。
独立 `%command` 块	空字符串。

之所以如此，是因为 `wait` 和 `--screen` 模式本来就不会生成对应的 `.out` 文件。换句话说，`${output}` 不是一个“永远存在的逻辑文件名”，而是一个与实际执行模式严格绑定的运行时占位符。

5.5.2 数组变量

数组变量使用 Bash 风格语法定义，例如：

```
state=(1 2 3 4)
```

对于一个数组变量，程序会把同一脚本重复执行多次，每次取数组中的一个元素；如果存在多个数组变量，则先计算这些数组的笛卡尔积，再对每个组合执行整套任务序列。

需要注意的是，数组变量在每一轮里都会退化为一个普通标量。换句话说，`state=(1 2 3)` 会让脚本跑三轮，而在任意一轮里 `${state}` 都只对应一个当前值，并不会一次性展开成三行命令。

数组元素可以带成对引号，解析后会去除元素级外层引号。当元素本身不含空格时，通常不必额外加引号；只有在你明确需要保留某些特殊字符或 shell 风格写法时，才需要这样做。

5.5.3 列展开变量

很多人第一次见 `${name*}` 时，会把它和 `name=(...)` 这种数组变量混在一起。两者最大的差别在于：

- 数组变量增加的是整份脚本的执行轮数；
- 列展开变量不增加执行轮数，而是在某一轮内部把一条命令展开为多条，或者把整列值作为一个列表继续传给模板。

列展开变量有两种等价写法。你既可以直接写成：

```
frag*=(2 5 9)
```

也可以显式写成：

```
frag1=2
frag2=5
frag3=9
```

程序会把前一种写法物化为后一种编号变量，并可进一步推导出 `len(frag)=3`。反过来，只要存在 `frag1`、`frag2`、`frag3` 这一组变量，`${frag*}` 与 `${len(frag)}` 也仍然可以正常工作。

与之对应，输入侧新增了两种读法：

写法	含义
`${name*}`	读取列表变量 `name` 的全部元素。
`${len(name)}`	读取列表变量 `name` 的元素个数。

例如：

```
frag*=(2 5 9)

%raw
${len(frag)}
show ${frag*}
end
```

会被展开为：

```
3
show 2
show 5
show 9
```

这里最需要强调的一点是：`${name*}` 只有在支持行展开的上下文里才会真正裂成多行。对输入文件而言，这种上下文主要是 `%raw`；对模块配置而言，则是 `.conf` 模板中的命令行。若把 `${name*}` 写在模块参数、`%process` 参数、`%command` 或 `wfn_rebase=` 里，它会先被视为一个单独字符串，例如 `(2 5 9)`，再交给后续模板继续处理。

因此，更准确地说，`${name*}` 不是一个“永远直接展开”的占位符，而是一个“把列表值传入当前上下文”的占位符：在 `%raw` 和 `.conf` 里，它会展开为多条命令；在只接受单值文本的位置，它则会保留为一个列表字面量。

5.5.4 交互式变量

`var=?` 表示运行时询问用户决定其值。例如：

```
state=?
```

程序会在运行时提示：

```
Bane need value for variable 'state' (supports bash array like (a b c), empty
↪  for blank):
```

这里的交互输入既支持单值，也支持数组形式。换言之，即使某个变量在脚本编写阶段还无法确定，也仍然可以在运行时以一次输入的方式生成单值或批量组合。

对于列表变量，则可以进一步写成：

```
frag*=?
len(frag)=?
```

这种组合表示：先在运行时决定列表长度，再逐个询问 `frag1`、`frag2`、`frag3` ……的值。若只写 `frag*=?` 而没有提供 `len(frag)`，程序会持续收集变量值，直到用户使用空输入主动结束为止。

5.5.5 变量替换时机

BaneWfn 中的变量替换，始终遵循“先输入文件，后模块模板”的顺序：

1. 先展开 `wfn=` 指定的文件列表，以及数组变量对应的变量组合；
2. 对于每一个“输入文件 × 变量组合”实例，先替换输入文件侧的变量与占位符；

3. 再把这些已经定值的模块参数交给 `.conf` 模板，继续做模板侧占位符替换；
4. 对于 `%command` 中的 `${output}`，则要等到对应的 Multiwfn 任务运行结束后，才会获得最终文件名。

换句话说，输入文件侧负责回答“这一轮跑谁、用哪些项目变量”，而模块模板侧负责回答“拿到这些参数后，应当生成哪些具体命令”。把这两层分开理解之后，变量、默认值、列展开和 `${output}` 的行为都会变得更容易预测。

5.6 任务块类型

5.6.1 模块块

模块块以 `[module]` 开始，可包含参数行、`%process`、`%raw` 与 `%command`。一个典型例子如下：

```
[fmo]
index h
%process
    orb grid 2
%raw
0
%command
    mv h.cub ${input}_H.cub
end
```

模块块的处理顺序固定为：

1. 先执行模块配置中的 `[main]`；
2. 再按 `%process` 中的顺序执行各步骤；
3. 然后拼接模块内 `%raw`；
4. 若以 `end` 收尾，则自动追加 `[quit]`；
5. 最后在 Multiwfn 成功结束后执行 `%command`。

如果 `[main]` 序列中有变量，可以在模块块开头通过 `key value` 的形式指定参数，每行一个，例如：

```
index h
```

Note: 模块参数和 `%process` 参数使用的是**空格分隔的 `key value` 语法**，而不是 `key=value`。

5.6.2 %process

`%process` 用于声明模块步骤，它依赖于当前已经进入某个模块块。例如：

```
%process
    orb index h grid 2
    orb_num index 15
```

相关规则如下：

- 每行第一个 token 是步骤名，对应 `.conf` 中的 section 名；

- 后续参数按“键值”成对解析；
- 当前版本不支持参数值中包含空格；
- 若一行参数数量为奇数，最后一个未成对 token 会被忽略；
- `%process` 只能用于 module 块内部；若在块外书写，只会产生警告，不会形成任务。

因此，`%process` 的职责应理解为“调用模块中的一个已定义步骤，并在调用时为它补入参数”，而不是直接写原始 Multiwfn 命令。

5.6.3 %raw

`%raw` 表示原始 Multiwfn 输入序列。它既可以写在模块内，也可以单独存在。例如：

```
%raw
12
1
2
q
end
```

`%raw` 的核心特点是“程序不解释内容，只负责原样传递”。它的行为如下：

- 独立 `%raw` 块不依赖任何模块配置；
- 独立 `%raw` 在 end 模式下不会自动补 `[quit]`，若需要优雅地退出，应在 `%raw` 中自行写完整退出序列。
- `%raw` 内每一行都会原样送给 Multiwfn；
- `%raw` 中的空行、`#` 行和占位符替换后的文本都会保留；
- 若写在模块内，`%raw` 序列会追加在 `[main]` 与 `%process` 生成内容之后；

从实践角度看，`%raw` 适合处理两类情况：一类是尚未模块化/不值得模块化、但有确定的原始菜单输入；另一类是在模块化流程中临时插入一两步补丁，而不值得为此改动公共 `.conf` 模板的需求。

5.6.4 %command

`%command` 用于执行 shell 或 batch 后处理命令，也可以写在模块内或独立存在。例如：

```
%command
#!/bin/bash
mv density.cub ${input}_den.cub
end
```

其行为如下：

- 模块内 `%command` 会在对应 Multiwfn 任务成功结束后执行；
- 顶层允许写独立 `%command` 块；这类任务不会调用 Multiwfn，而只执行命令块本身；
- `%command` 内容按原样保留，空行和注释行也会写入最终脚本；
- 在 `--dryrun` 下不会实际执行 `%command`，只会打印替换后的最终命令。

如果你的工作流需要整理输出目录、批量改名、写日志、调用可视化脚本或触发额外分析，那么 `%command` 往往是把“算完之后该做什么”显式写进工作流的最好位置。

5.6.5 `end` 与 `wait`

每个 `%raw`、module 块或命令块都以 end 或 `wait` 收尾。两者决定了整个任务按哪种模式执行。

5.6.5.1 `end`

- 对于 module 块，end 表示按非交互模式执行 Multiwfn，并在成功后执行 `%command`；
- 对于独立 `%raw`，end 表示按非交互模式执行原始 Multiwfn 序列；
- 对于独立 `%command`，end 表示命令块结束。

5.6.5.2 `wait`

- 对于 module 块，`wait` 表示进入交互模式；程序会先喂入预设命令，再把 Multiwfn 会话交还给用户；
- 对于独立 `%raw`，`wait` 表示把 `%raw` 内容作为交互模式的预输入；
- 对于独立 `%command`，`wait` 与 end 行为一致。

如果你希望脚本先帮你走完一部分固定菜单，再由你手动接管剩余操作，那么 `wait` 是最自然的选择；如果你希望整段流程完全无人值守地执行，则应使用 end。

5.6.6 `wfn_rebase=`

`wfn_rebase=` 用于在同一脚本中切换后续任务使用的输入文件。例如：

```
wfn_rebase=hole.cub
wfn_rebase=
```

它的规则如下：

- 只能写在块外；
- 非空值表示切换后续任务输入文件；
- 空值表示恢复到本轮原始输入文件；
- 该值参与输入侧占位符替换；
- 若目标文件在执行时不存在，程序会给出警告，但仍继续执行，最终由 Multiwfn 或后续命令报告错误。

`wfn_rebase` 常用于需要 Multiwfn 预处理的任务，如进行激发态波函数分析分两步，第一步是生成激发态自然轨道，第二步需要用生成的激发态自然轨道计算静电势，则 `wfn_rebase` 可以用来在同一个脚本内切换波函数文件，而不必另开一个脚本。

6 执行行为与文件产物

6.1 文件模式

以 end 收尾的 module 块或独立 `%raw` 块，会使用非交互文件模式执行。程序在这一模式下会依次完成以下操作：

1. 生成临时 Multiwfn 命令文件；
2. 通过重定向方式调用 Multiwfn；

3. 默认把标准输出追加到 `.out` 文件；
4. 若 Multiwfn 成功结束，再执行对应的 `%command`。

从自动化角度看，文件模式最适合用于批量处理、无人值守计算和需要保留完整日志的场景，因为它会把中间命令和输出文件命名为可追踪的产物。

6.2 交互模式

以 `wait` 收尾的 module 块或独立 `%raw` 块，会进入交互模式。在这种模式下，程序会：

1. 通过管道向 Multiwfn 发送预设输入；
2. 保留会话，让用户继续手动交互；
3. 不生成 `.out` 文件；
4. 若存在 `%command`，则在 Multiwfn 成功结束后再执行。

交互模式最常用于绘图时需要手动调整，但前置命令极度繁琐，敲错一次白干很久的场景。善用 `wait`，可以将繁琐重复的步骤全部省略，直接进入绘图细调阶段。

6.3 文件命名规则

6.3.1 Multiwfn 命令文件与输出文件

为了方便定位每一轮任务产物，BaneWfn 会为 Multiwfn 命令文件和输出文件生成可预测的名称：

任务类型	命令文件	输出文件
module 块	`<module>_<wfnBase>.txt`	`<module>_<wfnBase>.out`
独立 `%raw`	`raw_<wfnBase>.txt`	`raw_<wfnBase>.out`

若同名 module 块或 raw 块重复出现，第二个及之后的块会自动追加编号后缀，例如 `_1`、`_2`。这意味着，即便你在同一脚本中多次调用同一模块，程序仍能为每个块生成彼此可区分的中间文件与日志文件。

6.3.2 `%command` 临时脚本

`%command` 在执行时会临时生成脚本文件，并在执行后删除。命名模式如下：

平台	文件名模式
Linux / POSIX	`<module>_commands[_{N}].sh`、`raw_commands[_{N}].sh` 或 `commands[_{N}].sh`
Windows	上述相应 `.bat`；若启用 Git Bash，则依然生成 `.sh`

如果用户在调试 `%command` 时发现命令块没有按预期工作，最直接的办法通常是使用 `--dryrun` 查看替换后的命令内容，而不是试图在执行后寻找临时脚本残留。

6.4 %command 的平台执行规则

6.4.1 Linux / POSIX

在 Linux / POSIX 平台下，程序会生成带 `#!/bin/bash` 头的临时 `.sh` 脚本，赋予执行权限后直接运行。对于一般的文件移动、目录创建和外部工具调用，这种模式足够直接，也与大多数科研脚本工作流兼容。

6.4.2 Windows

在 Windows 下，默认行为是生成 `.bat` 脚本，并通过 `cmd /c` 执行；若 `banewfn.rc` 中配置了 `gitbash_exec`，且 `%command` 第一行是 `#!/bin/bash`，则程序会改为使用 Git Bash 执行该命令块。

因此，同一份脚本既可以写成 Windows 原生命令风格，也可以在明确声明 `#!/bin/bash` 的前提下保留更接近 POSIX 的写法，减少跨平台迁移时的重复维护。

6.5 其他行为说明

下面这些规则虽然不复杂，但在脚本调试时经常会遇到：

- `%raw` 与 `%command` 中的 `#` 行是字面内容，不是输入文件注释；
- 自定义变量与命令行 `-v` 要求使用 `key=value` 且 `value` 非空；
- 命令行 `-w` 与第二个位置参数优先于输入文件头 `wfn=`；仅在命令行未提供时，才回退到文件内 `wfn=`；
- `%command` 只有在前置 Multiwfn 块返回成功时才会自动执行，独立 `%command` 块除外；
- `wait` 模式与 `--screen` 模式通常不会生成 `.out` 文件，因此 `${output}` 为空；
- `wfn_rebase` 目标文件缺失只会产生警告，不会阻止后续任务继续提交给 Multiwfn；
- 通配符展开只处理普通文件，不处理目录。

如果你希望脚本行为可预测、可交付，那么最稳妥的策略永远是：把核心流程写成模块，把例外写进 `%raw`，把结果整理写进 `%command`，同时通过 `--dryrun` 验证生成的命令文件和变量替换结果。

7 inline conf 与单文件打包

7.1 inline conf 语法

inline conf 允许把模块配置直接嵌入输入文件末尾，从而把原本依赖外部 `.conf` 的脚本，打包成一个可单独分发的自包含文件。其语法如下：

```
#>>> BANEWFN_INLINE_CONF_BEGIN fmo
# [main]
# 200
# [orb]
# 3
# ${index:-h}
# [quit]
# 0
```

```
# q
#<<< BANEWFN_INLINE_CONF_END fmo
```

其规则如下：

- 每个块都绑定一个模块名；
- 块内容按“每行前加一个 #”的方式写在文件末尾；
- 读取时，程序会去掉每行最前面的一个 # 以及其后的一个可选空白字符；
- 若输入文件中存在对应模块的 inline conf，则优先使用内嵌文本，而不是 `confpath` 下的外部文件。

在项目交付中，inline conf 的意义非常明确：它让“脚本逻辑”和“模块定义”被打包到同一个文件中，从而大幅减少“别人拿到脚本却跑不起来”的情况。

7.2 bwpack 打包规则

`bwpack` 用于把输入文件中**实际引用到的模块配置**打包到文件末尾，生成自包含的 `.bwc` 脚本。其行为可以概括如下：

- 先解析输入文件，找出其中实际使用到的模块名；
- 再逐个读取对应 `.conf` 文本；
- 去除原文件中已有的 inline conf 尾块；
- 最后在文件末尾追加新的 `#>>> BANEWFN_INLINE_CONF_BEGIN ...` / `#<<< ...` 打包块。

Note：

- 纯 `%raw` / `%command` 脚本不依赖配置文件，无需过 bwpack 包装；
- 输入文件中的 inline conf 应置于文件末尾；重新打包时，旧 inline conf 尾块会被整体替换；
- `bwpack` 只打包**脚本实际引用到的模块**，不会把整个 `confpath` 目录无差别塞进输出文件。

8 命令行工具

8.1 banewfn

8.1.1 基本用法

`banewfn` 是主程序，它负责解析输入文件、加载模块配置、展开批量任务并执行整个工作流。基本调用方式如下：

```
banewfn <input.inp> <molecule.fchk> [options]
banewfn -w <molecule.fchk> <input.inp> [options]
```

8.1.2 主要选项

选项	作用
-h, --help	显示帮助。
-l, --list	列出可用模块；若后接模块名，则显示该模块的 section 与占位符摘要。
-c, --cores <num>	指定 CPU 核数。
-d, --dryrun	干运行。
-e, --extargs <args>	给 Multiwfn 追加额外参数；支持一整串带引号参数。
-s, --screen	输出直接打印到屏幕，而不是写入 .out 文件。
-n, --nogui	给 Multiwfn 追加 -silent。
-w, --wfn <file>	指定波函数文件或通配符模式。
-v, --var <key=value>	设置自定义变量；可重复使用，支持数组写法，也支持 name*= 与 len(name)

8.1.3 波函数文件来源规则

命令行 -w/--wfn 将覆盖输入文件头部 wfn=。若均不存在，在启动后向用户交互式询问。

8.1.4 核心数来源规则

核心数按以下顺序确定：

1. 命令行 -c/--cores
2. 输入文件头部 core=
3. banewfn.rc 中的 cores

对于需要临时提高并行度的场景，直接使用 -c 即可，无需改动脚本本身。

8.1.5 变量来源规则

变量按以下顺序确定：

1. 命令行 -v/--var
2. 输入文件头部自定义变量
3. 交互式变量输入值

在项目协作中，推荐把“稳定默认值”写进脚本，把“每次都可能变动的值”通过命令行传入。这样既能保留脚本的可读性，又能避免为了少量输入差异频繁复制脚本。

8.1.6 -l 列表功能

banewfn -l 的行为如下：

- 读取 banewfn.rc 以确定 confpath；
- 无模块名参数时，列出 confpath 目录下所有 .conf 文件的基名；
- 有模块名参数时，读取对应 .conf 并显示各 section 与识别到的参数名摘要；
- [quit] 不在摘要中显示。

对于维护模块库的人来说，-l 是一个非常实用的“快速自查”入口。它可以帮助你确认某个模块是否存在、有哪些 section、参数大致是什么，而不必每次都手工打开 .conf 文件通读。

8.2 bwpack

bwpack 用于把输入文件中实际引用到的外部模块配置打包到同一文件末尾，生成自包含的 .bwc 脚本。它的重点不是执行工作流，而是整理交付形式。

8.2.1 基本用法

```
bwpack <input.bw> [options]
```

8.2.2 选项

选项	作用
-h, --help	显示帮助。
-o, --output <file>	输出文件名；默认把输入扩展名替换为 .bwc。
-c, --confdir <dir>	显式指定模块配置目录。
--rc <banewfn.rc>	从给定 rc 读取 confpath。
-i, --inplace	原地覆盖输入文件。

8.2.3 bwpack 的行为

- 先解析输入文件，找出其中实际使用到的模块名；
- 逐个读取对应 .conf 文本；
- 去除原文件中已有的 inline conf 尾块；
- 在文件末尾追加新的打包块。

如果你的目标是把脚本发给别人、存档到项目目录，或者减少“缺少 conf 文件”造成的环境问题，那么 bwpack 通常是交付前最后一步非常值得做的整理工作。

9 示例

9.1 最小模块示例

```
wfn=test.fchk

[fmo]
%process
    orb index h
end
```

该脚本会加载 fmo.conf，执行 [main] 后再执行 orb 步骤，并在 end 模式下自动追加 [quit]。如果你第一次验证环境是否可用，建议就从这种最小示例开始，因为它最容易定位是配置问题、模块问题还是 Multiwfn 本体问题。

9.2 模块与后处理联合示例

```
wfn=*.fchk
prefix=result

[grid]
%process
    electron grid 3
    esp grid 1
%command
#!/bin/bash
mkdir -p ESP
mv density.cub ESP/${prefix}_${input}_den.cub
mv totesp.cub ESP/${prefix}_${input}_esp.cub
mv ${output} ESP/
end
```

这个例子展示了“模块执行 + 结果归档”的典型组合方式。其含义如下：

- `wfn=*.fchk` 会批量匹配当前目录下所有 `fchk` 文件；
- `%process` 调用模块中的两个步骤，分别生成电子密度和 ESP 网格；
- `%command` 负责创建目录、重命名文件并把 `.out` 一并归档；
- `${input}` 会在每一轮执行中替换为当前文件的基名；
- `${output}` 会被替换为当前块实际产生的 `.out` 文件名。

9.3 数组变量示例

```
wfn=test.fchk
state=(1 2 3)

[excit]
%process
    nto state ${state}
end
```

该脚本会对 `state=1`、`state=2`、`state=3` 依次重复执行整套任务。通过这一写法，可以把原本需要复制三份脚本的参数扫描，浓缩为一份结构清晰的工作流定义。

9.4 列展开变量示例

```
wfn=test.fchk
frag*=(2 5 9)

%raw
${len(frag)}
show ${frag*}
end
```

这个例子只会执行一轮，因为 `frag*` 不是“让脚本重复运行”的数组变量，而是“在单轮

内部展开多条命令”的列表变量。最终送给 Multiwfn 的内容将是：

```
3
show 2
show 5
show 9
```

如果把同样的 `frag*` 先作为模块参数传入 `.conf` 模板，那么模板侧也可以继续用 `${frag*}` 与 `${len(frag)}` 做同样的展开。这类写法最适合展示列表变量在输入侧与模板侧之间的传递关系。

9.5 `wfn_rebase` 示例

```
wfn=origin.fchk
state=1
[excit]
%process
no state ${state}
end

wfn_rebase=NO_000${state}.mwfn

[grid]
%process
    electron
    esp
%command
#!/bin/bash
    mkdir -p ESP_${state}
    mv density.cub ESP_${state}/${input}_den.cub
    mv totesp.cub ESP_${state}/${input}_esp.cub
    mv NO_000${state}.mwfn ${output} ESP_${state}
    cat << EOF > ESP_${state}/esp.bat
    vmd -e esp.vmd
EOF

    cat << EOF2 > ESP_${state}/esp.vmd
    vcube *_den.cub map *_esp.cub
    set colorlow -20
    set colorhigh 20
    mol scaleminmax 0 1 -20 20
    puts "unit: kcal/mol"
EOF2
end
```

该脚本展示了“先生成中间文件，再切换输入继续分析”的典型写法：

- 第一段以 `origin.fchk` 作为输入，生成后续会用到的 `NO_0001.mwfn`；
- `wfn_rebase=NO_0001.mwfn` 之后，后续 `grid` 块会改为以 `NO_0001.mwfn` 作为 Multiwfn 输入计算静电势。

相比把这些步骤拆成两个脚本手工接力，这种写法可以把输入切换关系直接保留在一份文件里，后期复查时也更容易看懂整条数据流。

9.6 自包含 .bwc 工作流示例

例如 fmo.bw：

```
wfn=*.fchk
[fmo]
%process
    orb index h-1
    orb index h
    orb index l
    orb index l+1
%command
#!/bin/bash
mkdir -p ${input}
mv h-1.cub ${input}/${input}_oH1.cub
mv h.cub ${input}/${input}_oH.cub
mv l.cub ${input}/${input}_oL.cub
mv l+1.cub ${input}/${input}_oL1.cub
echo "vmd -e orb.vmd" > ${input}/orb.bat
echo "vcube *.cub" > ${input}/orb.vmd
rm ${output}
end
```

可以使用如下命令进行打包

```
bwpack fmo.bw
```

执行后会生成 `fmo.bwc`，其中包含该脚本实际使用到的模块配置块。

```
wfn=*.fchk
[fmo]
%process
    orb index h-1
    orb index h
    orb index l
    orb index l+1
%command
#!/bin/bash
mkdir -p ${input}
mv h-1.cub ${input}/${input}_oH1.cub
mv h.cub ${input}/${input}_oH.cub
mv l.cub ${input}/${input}_oL.cub
mv l+1.cub ${input}/${input}_oL1.cub
echo "vmd -e orb.vmd" > ${input}/orb.bat
echo "vcube *.cub" > ${input}/orb.vmd
rm ${output}
end
```

```
# Bundled by bane dysta
# ConfDir: D:\MyProgram\banewfn\conf

#>>> BANEWFN_INLINE_CONF_BEGIN fmo
## bundled module: fmo
# # 主逻辑
# [main]
# 200
#
# [orb]
# 3
# ${index:-h}
# ${grid:-2}
#
# # output: orb0000xx.cub
# [orb_num]
# 3
# ${index:-}
# ${grid:-2}
# 1
#
# # 退出
# [quit]
# 0
# q
#<<< BANEWFN_INLINE_CONF_END fmo
```

对于需要共享给同事、附在项目归档中，或者在另一台机器上尽量减少外部依赖的场景，这种自包含脚本往往更稳妥。