一、强制（a）

1. 文件编码

所有源文件（.c）和头文件（.h）必须使用 UTF-8 无 BOM 编码。

2. 文件组织与命名

每个模块必须由对应的 `.h`（接口声明）和 `.c`（实现）文件组成，两者文件名一致，例如 `foo.h`、`foo.c`。

头文件名、源文件名全部小写，单词间用下划线分隔（snake_case）。

3. 头文件防卫

每个头文件必须使用宏防卫（include guard）。

```
#ifndef MODULE_FOO_H
#define MODULE_FOO_H
/ … /
#endif   // MODULE_FOO_H
```

防卫宏名必须全部大写，单词间用下划线，前缀为项目或模块名称，确保全局唯一。

4. 宏定义

所有的宏（`#define`）必须以大写字母及下划线命名，例如 `MAX_BUFFER_SIZE`。

严禁无参宏和函数型宏直接定义常量或简单函数，必须优先使用 `static const` 或 `static inline`。

若确实需要函数型宏，务必使用 `do { … } while (0)` 结构封装。

5. 缩进与代码风格

缩进统一使用 4 个空格，禁止使用制表符（Tab）。

行宽不得超过 100 字符。必要时可适当换行，但每行长度不得超过 100。

关键字（`if/else/for/while/switch`）后总是跟一个空格，函数名和左括号之间不留空格（如 `foo(arg)`，而非 `foo (arg)`）。

花括号采用 K\&R 风格：

```
if (cond) {
    / … /
} else {
    / … /
}
for (int i = 0; i < n; i++) {
    / … /
}
```

6. 类型与固定宽度整型

所有涉及跨平台或与外部协议交互的整型，必须使用 `stdint.h` 中的固定宽度类型（如 `int32_t`、`uint64_t` 等）。

对于普通循环索引，可使用 `int`；对于表示大小、偏移等，应使用 `size_t`。

7. 函数声明与定义

所有函数必须在头文件中声明，并在 `.c` 文件中实现。函数原型需对齐形式一致，包括参数类型和命名。

每个函数的参数、返回值类型、用途都要在头文件注释中简要描述（见“文档注释”）。

禁止在函数内部重复声明外部函数（即禁止在 `.c` 中隐式定义未在对应头文件声明的函数）。

8. 变量命名

变量名必须使用 snake_case，且含义清晰；局部变量通常以小写字母开头，例如 `buffer_size`、`ret_code`。

全局变量绝对禁止（除非项目确有需求且经过评审）；若必须使用，必须在头文件中明确标注 `extern`，且在实现文件中加上 `static` 或前缀模块名（如 `module_foo_global_flag`），但一般强烈禁止。

9. 常量与只读数据

全局常量必须使用 `static const` 的方式定义并命名为大写带下划线，例如：

```
static const int MAX_CONNECTIONS = 1024;
```

在头文件暴露的常量可以用宏或 `enum` 定义，但应保证含义明确，且避免命名冲突。

10. 指针与内存管理

所有 `malloc`/`calloc`/`realloc` 后的返回值必须检查是否为 `NULL`，如失败则做适当处理。

每个 `malloc`、`fopen`、`fread` 等获取资源的函数，必须在对应位置释放资源（`free`/`fclose`），不得出现内存/资源泄漏。

对内存分配失败的处理要统一，例如：

```
void ptr = malloc(nbytes);
if (ptr == NULL) {
    / 打日志 + errno + 返回错误码 或者 退出 /
}
```

禁止出现未初始化使用的指针或变量。编译期间启用 `-Wall -Wextra -Werror`，并通过静态分析工具检测。

11. 返回值与错误处理

任何可能失败的函数都必须返回错误码或状态（如返回 `int`，`0` 表示成功，非 0 表示错误）。调用方必须对返回值进行检查，不得直接忽略。

对于系统调用或标准库函数（如 `open`、`read`、`write` 等），必须检查其返回值并处理 `errno`。

若函数遇到无法恢复的致命错误，可在文档中注明可能 `exit()`，但应尽量保证可由上层捕获并清理资源，再统一退出。

12. 头文件包含顺序

在 `.c` 文件中，头文件的包含顺序必须为：

1. 对应自己的模块头文件
2. C 标准库头文件（如 `<stdio.h>`、`<stdlib.h>` 等）
3. 第三方库头文件（如 `<glib.h>`、`<openssl/…>` 等）
4. 本项目内其他模块头文件

每个分组之间保留一行空行。以保证依赖正确且命名冲突可被及时发现。

13. 模块内部可见性

不得将不希望被外部直接调用的函数或变量声明在头文件中。

对于仅在本 `.c` 文件内使用的函数或变量，必须添加 `static` 关键字限制其作用域。

14. 注释规范

文件头部必须包含模块名称、简要功能描述、作者、创建日期及修订记录（可选）。

函数头部注释采用 Doxygen 风格（或类 Doxygen），包含函数功能、参数说明、返回值说明、可能的错误码。示例如下：

```
/
   @brief 计算两个整数的和
```

```
    @param a [in]  第一个加数
    @param b [in]  第二个加数
    @return  将要返回  a + b  的结果
  /
 int add(int a, int b);
```

代码块中若需解释算法或复杂逻辑，使用 `/* … */` 多行注释；若为一行注释，使用 `//`。

代码注释与实际代码不能出现严重不一致的情况，发现必须及时更新。

15. 无符号类型与边界检查

所有与长度、大小、数组索引相关的变量，必须使用无符号类型或 `size_t`。使用有符号整型时，需对负值做充分检查，避免越界或溢出。

数组访问、指针偏移前必须检查下标边界，避免出现缓冲区溢出。

16. 代码编译与警告

工程编译时必须启用至少如下编译选项：

```
-Wall -Wextra -Wconversion -Wshadow -Werror
```

使代码在警告级别上严格，通过所有警告即算通过。

针对不同编译器（GCC、Clang、MSVC），需要单独配置对应的严格警告选项，并在持续集成中保证无警告编译通过。

17. 依赖管理与构建系统

构建工具统一使用 CMake（≥3.10），并强制使用 Out-of-Source Build（即在项目根目录外新建 `build/` 目录进行编译）。

CMakeLists.txt 中禁止写死绝对路径，各模块依赖通过 `find_package` 或 `add_subdirectory` 机制引入。

构建目录中产生的可执行文件、库文件、临时中间文件，禁止提交到版本控制系统。

18. 安全与代码审计

禁止使用不安全的函数，例如 `gets()`、`strcpy()`、`strcat()` 等，必须使用 `fgets()`、`strncpy()`、`strncat()` 或者 `snprintf()` 等安全接口。

对外部输入（网络、文件、命令行参数等）必须进行严格校验，防止缓冲区溢出或格式化字符串漏洞（禁止 `printf(user_input)` 等类似用法）。

敏感操作（如内存释放、文件删除、权限修改等）必须加注释并经过安全组审计。

二、推荐（b）

19. 命名空间与前缀

如果项目较大或有多个子模块，建议给每个模块统一添加前缀，例如 `net_`、`db_`、`ui_`，以避免命名冲突。

20. 常量组织

对于一组相关常量，可使用 `enum` 或 `enum class`（若在 C++ 项目中）。在纯 C 项目中，尽量用 `enum` 代替一堆 `#define`，例如：

```
typedef enum {
    STATUS_OK = 0,
    STATUS_ERROR = -1,
    ...
} status_code_t;
```

21. 函数长度与职责单一

最好保证单个函数的行数不超过 200 行，且功能应保持单一。若函数过长或职责过多，应拆分为更小的子函数。

22. 循环与条件表达式简洁化

循环条件和分支表达式尽量保持简洁，避免多层嵌套。多层嵌套时可优先考虑提前返回（early return）或拆分函数。

23. 避免深度嵌套

如果条件嵌套超过三层，建议重构为多个子函数或采用 `goto` 跳转到统一错误处理标签（仅限于错误处理）。

24. 静态分析

建议在持续集成流水线中引入静态分析工具（如 Cppcheck、clang-tidy、Coverity 等），并定期检查。

25. 代码格式化工具

推荐使用统一的代码格式化工具（如 clang-format）并在项目根目录下维护 `.clang-format`，以保证团队提交的代码风格一致。

26. 内存对齐

对数据结构进行合理对齐，必要时使用 `__attribute__((packed))`（GCC）或 `#pragma pack`。但除非确有必要，否则不应随意打包，以免产生性能问题。

27. 线程安全与重入性

对于可能在多线程环境下调用的函数，需确保线程安全或在文档中注明非线程安全。

避免使用全局或静态可变变量；若必须使用，必须用互斥锁（`pthread_mutex_t`）等机制保护。

28. 日志与调试

推荐在项目中引入专门的日志库（如 `log4c`、`zlog` 等），并统一定义日志级别（DEBUG、INFO、WARN、ERROR）。

所有模块在关键节点（如初始化失败、关键数据读写、异常捕获）处要打印日志，并包含模块名、函数名、行号、错误码等关键信息。

29. 单元测试与覆盖率

推荐使用 C 语言单元测试框架（如 Unity、Check、CMocka 等）编写单元测试，并达到至少 80% 代码覆盖率。

测试代码与生产代码分离，存放在 `tests/` 目录下，构建系统需支持一键触发测试。

30. 持续集成（CI）

建议引用 CI 平台（如 Jenkins、GitLab CI、GitHub Actions 等），设置至少编译检查、单元测试和静态分析三个流水线环节。

31. 文档与 README

每个模块根目录下须有 `README.md`，包含模块功能简介、依赖说明、编译/安装示例、常见问题及联系方式。

项目根目录下 `README.md` 应概述项目整体架构、目录结构、快速上手指南。

32. 接口版本控制

如果头文件导出的 API 可能发生变更，建议在头文件或文档中写明版本号 / 兼容性策略。

对外暴露函数可结合宏定义控制版本，例如：

```
#define MODULE_FOO_API_VERSION 2
```

33. 动态库与静态库

如果项目需要生成动态库（`.so`）或静态库（`.a`），推荐写独立的 CMake 脚本生成 `libfoo.a`、`libfoo.so`，并在发行版中同时放置头文件和对应库文件。

34. 编译选项与宏控制

如需支持多种编译模式（如 `DEBUG`、`RELEASE`），建议使用 CMake 的 `CMAKE_BUILD_TYPE`，并在代码中通过 `#ifdef DEBUG` 控制调试信息的打印。

35. 资源文件与目录结构

项目目录结构应按功能分为 `src/`（源码）、`include/`（公开头文件）、`lib/`（第三方库）、`tests/`（测试）、`docs/`（文档）、`build/`（编译输出）。

36. 外部依赖管理

推荐通过 `find_package`、`ExternalProject_Add` 或 `FetchContent` 管理第三方依赖，避免将第三方源码直接拷贝到项目内。

37. 接口函数可重入

如果模块需提供给外部使用，且可能被多次调用，建议设计为可重入函数，避免在函数内使用静态内存或全局状态。

38. 使用 `restrict` 关键字

针对 C99 编译器，可在函数指针参数上加 `restrict`，以便编译器进行更多优化；如：

```
void copy_array(int restrict dst, const int restrict src, size_t n);
```

39. 单向依赖

在模块间调用时，应尽量保持单向依赖（高层模块调用低层模块接口，低层绝不包含高层头文件），避免循环依赖。

40. 性能剖析

对于性能敏感的模块，建议使用 `gprof`、`perf`、`valgrind` 等工具进行剖析，并在代码评审时提供性能报告。

三、允许（c）

41. 宏中嵌入调试打印

允许在开发阶段使用类似：

```
#ifdef DEBUG
#define DEBUG_PRINT(fmt, ...) printf("[DEBUG] %s:%d: " fmt, __FILE__, __LINE__, ##__VA_ARGS__)
#else
#define DEBUG_PRINT(fmt, ...) ((void)0)
#endif
```

以方便调试，但正式发布时须定义 `NDEBUG` 或去除 `DEBUG_PRINT`。

42. 使用三元运算符

允许在简单条件赋值场景下使用 `?:`，但不鼓励过度嵌套，保证可读性。

43. 可变参数函数

允许定义可变参数函数（如日志接口），但应小心处理参数列表，避免格式化字符串漏洞。

44. 函数指针与回调

允许在模块间传递函数指针实现回调机制，但应在注释中标明回调函数签名及约束。

45. 灵活使用 `goto` 进行错误处理

允许在函数内部使用 `goto` 跳转到统一的错误处理和资源释放位置，但禁止用于正常业务逻辑的流程控制。