C 语言后端开发技术管理规范

命名规范

- **强制 (a)：** 文件名应全小写，可使用下划线或短横线分隔单词。例如 http_server_logs.c。
- **强制 (a)：** 变量名（包括函数参数）使用蛇形命名（snake_case），即全小写且单词间用下划线分隔。例如 file_path、max_connections。
- **强制 (a)：** 函数名使用驼峰式（CamelCase）命名，首字母大写，每个单词首字母均大写。例如 InitDatabase()、ComputeResult()。
- **强制 (a)：** 类型名（包括 struct、enum、typedef 定义的类型）使用帕斯卡命名（PascalCase），每个单词首字母大写，不使用下划线。例如 HttpRequest、ErrorCode。
- **强制 (a)：** 常量名（const 或 constexpr 定义的只读值）使用小写字母 k 前缀加驼峰式命名。例如 kMaxBufferSize、kTimeoutMs。
- **允许 (c)：** 宏名应全部大写，单词间用下划线分隔。例如 #define MAX_THREADS 8。不过应尽量少用宏。
- **推荐 (b)：** 尽量避免不常见的缩写；如果使用缩写（如接口名或领域术语），应以全大写或首字母大写形式出现（例如用 HtmlParser 而不是 HTMLParser，或 ReadRpc()而不是 ReadRPC()）。
- **强制 (a)：** 命名中不得使用匈牙利命名法（不要给变量名前加类型前缀）。即变量名不能包含类型信息，如不写 iCount、fValue 等前缀。
- **推荐 (b)：** 枚举类型中的枚举值命名应遵循常量命名风格（建议使用小写字母 k 开头加驼峰式命名）。例如 enum ErrorType { kOk, kNotFound, kInvalidParam };。

代码风格

- **强制 (a)：** 缩进仅使用空格，不得使用制表符（Tab）；每级缩进为 **2 个空格**。确保编辑器自动转换制表符为空格。
- **强制 (a)：** 每行代码长度不超过 80 个字符。超过时对齐换行，使代码可读性更高。
- **强制 (a)：** 采用 K&R 大括号风格：左花括号{应放在控制语句或函数签名所在行的末尾，右花括号单独成行，与其对应的起始行保持相同缩进。例如：

```
void Func(int x) {
  if (x > 0) {
    // ...
  }
}
```

- **强制 (a)：** 函数声明/定义格式：函数名与左括号之间**无空格**（Name()，所有参数对齐；右圆括号)与后继左花括号{之间应保留一个空格。例如：

```
int ComputeSum(int a,  // 两级缩进对齐
                   int b) {
// ...
}
```

- **强制 (a)：** 控制语句（如 if、for、while）与左括号之间留一个空格。例如 if (condition)。for (、while (等也同理。
- **推荐 (b)：** 操作符前后应加空格以提高可读性，例如 a + b 而不是 a+b。但一元操作符（如取地址符&、取反!等）和后缀操作符（如自增 i++）紧贴操作数。
- **允许 (c)：** 在头文件里编写的行注释建议使用//；对于多行注释或段落说明可用/* ... */。注释应尽量简洁明了，描述“为什么”而不是“做什么”。

内存管理

- **强制 (a)：** 调用 malloc/calloc/realloc 等动态内存分配函数后必须检查返回值，确保非 NULL，绝不能忽略返回值。例如：

```
char *buf = malloc(n);
if (buf == NULL) {
  // 处理内存分配失败
}
```

- **强制 (a)：** 对每次分配的内存都必须在不再使用时调用 free 释放；释放后应将指针赋为 NULL 以避免悬挂指针。禁止重复释放同一指针。
- **推荐 (b)：** 项目应使用静态分析工具或内存检查工具（如 Valgrind、AddressSanitizer）定期检测内存泄漏、越界和使用后释放等错误。这样可以在开发阶段及时发现和修复内存问题。

并发与多线程编程

- **强制 (a)：** 访问多线程共享资源（全局变量、静态变量或堆上数据）时，必须使用互斥锁、读写锁或其他同步机制进行保护，否则会产生数据竞争。确保每个共享数据的访问都处于锁保护之下。
- **强制 (a)：** 避免在多线程环境中使用函数内部的 static 局部变量或全局变量，因为它们在所有线程间共享，会引发竞态条件。如果需要线程私有数据，应使用线程局部存储（见下）。

- **推荐 (b)：** 在需要锁定多个锁时，应制定统一的锁定顺序并遵循它，以防止发生死锁。例如可以规定按锁地址从小到大顺序加锁。
- **推荐 (b)：** 使用条件变量（condition variable）时，应将 wait 操作放在循环中反复检查条件谓词，即使出现伪唤醒也能重新验证条件。例如：

```
pthread_mutex_lock(&m);
while (!condition) {
    pthread_cond_wait(&cv, &m);
}
// 处理满足条件后的逻辑
pthread_mutex_unlock(&m);
```

- **允许 (c)：** 对于线程私有的数据，可以使用线程局部存储（如 C11 的 _Thread_local 或 GCC 的__thread 关键字）保证每个线程拥有自己的独立副本。

安全编程

- **强制 (a)：** 严禁将用户输入或可控数据直接用作 printf/fprintf 等格式化输出函数的格式字符串。应当使用固定的格式字符串，变量数据作为参数传递。
- **强制 (a)：** 避免使用不带长度限制的字符串函数和 IO 函数（如 gets、strcpy、strcat、sprintf 等），因为它们容易导致缓冲区溢出。应使用带长度限制的安全函数，例如使用 fgets 替代 gets、使用 strncpy 或 strlcpy 替代 strcpy、使用 snprintf 替代 sprintf 等。
- **推荐 (b)：** 确保所有字符串目标缓冲区预留足够大小，并留出 1 个字节空间用于存放结尾的\0 字符。在进行字符串拷贝或拼接前，应检查源字符串长度以免超出目标缓冲区。
- **允许 (c)：** 遇到复杂的输入数据时，应做好输入验证，比如检查数值范围、字符编码和特殊字符等，必要时采用现成的安全库进行处理（如使用 sntprintf 检查整数范围或使用正则表达式库验证格式）。

模块划分与头文件管理

- **强制 (a)：** 所有头文件必须使用包含保护（include guard），防止重复包含。通常格式为：#ifndef PROJ_DIR_FILE_H_ ... #define PROJ_DIR_FILE_H_ ... #endif。宏名可根据项目/目录路径设置，保证唯一。
- **强制 (a)：** 头文件应自给自足（self-contained），即能单独编译通过。头文件内部必须包含其所依赖的其他头文件，保证使用者无需依赖传递包含。

- **推荐 (b)：** 遵循 “Include What You Use” 原则：在源文件或头文件中使用某个符号时，应直接#include 声明该符号的头文件，不要依赖其他头文件的间接包含。
- **推荐 (b)：** 每个模块（功能单元）使用一对.h/.c 文件实现。.c 文件首先#include 对应的.h 头文件，确保接口实现匹配。模块间应通过头文件暴露接口，隐藏内部实现。
- **强制 (a)：** 禁止在头文件中使用非标准的#pragma once；应使用标准的#ifndef/#define/#endif 保护，以保证可移植性。

编译与构建规范

- **强制 (a)：** 编译时应开启严格的警告选项（如 GCC/Clang 的-Wall -Wextra，MSVC 的警告等级 3 或更高），并将所有警告视为错误对待。这样可以提前发现潜在问题。
- **推荐 (b)：** 避免依赖编译器特有的扩展和关键字（如 GCC 的__attribute__、MSVC 的__declspec 等），除非绝对必要。若需使用，应通过宏封装（如定义 ATTRIBUTE_UNUSED）以提高代码可移植性。
- **推荐 (b)：** 建议在构建流程中集成静态分析工具（如 Cppcheck、Clang-Tidy、Coverity 等）和代码质量检查，以在编译前或提交前自动检测常见缺陷。
- **允许 (c)：** 可使用自动化构建系统（Make/CMake 等）管理项目，根据环境提供不同编译选项（调试/发布），但应保证可重现的构建过程，例如不在代码中硬编码版本信息、避免依赖未固定的外部库版本。

日志与错误处理

- **强制 (a)：** 所有系统调用和库函数的返回值（尤其是文件/网络 IO、内存分配、线程操作等）都必须检查。发生错误时，应及时记录日志并进行恰当的错误处理，而不是直接忽略或吞掉错误。
- **强制 (a)：** 使用统一的日志系统（如 syslog、日志库或项目自定义的日志接口）输出错误和运行信息，日志内容应包括时间戳、所属模块、错误描述等上下文信息。不要直接使用 printf 向标准输出打印日志。
- **推荐 (b)：** 函数或模块遇到错误时，优先返回错误码或状态值，让调用者决定如何处理。只有在程序运行已无继续意义时（如严重内部错误），才可以打印错误并退出。库函数内部尽量不要调用 exit。
- **允许 (c)：** 在开发和调试阶段，可使用 assert()检查不应发生的条件（如不变量），以捕获程序错误。生产环境代码中避免将 assert 用于处理可预见的运行时错误，应改为返回错误码或进行异常处理。

性能优化建议

- **强制 (a)：** 尽量使用局部变量（栈上分配）替代全局变量或静态数据，以提高访问效率。全局变量的访问开销更大且影响编译器优化。
- **推荐 (b)：** 对于大量数据复制操作，优先使用 memcpy/memmove 等高效内存函数，避免使用 strcpy/strcat 等逐字节查找空字符的函数。同样，对于数值类型数组的快速初始化，使用 memset 而非手写循环。
- **允许 (c)：** 在性能敏感的计算中，可以根据平台特点选择更快的数据类型。例如在多数架构上，int 运算一般比 char/short 更快；如果无内存瓶颈，使用 int 代替小整型可能提高性能。
- **允许 (c)：** 在进行性能优化前，应使用性能分析工具（如 gprof、perf、VTune 等）定位热点，避免盲目优化。针对关键路径进行优化（算法、内存访问模式等），并在每步优化后重新测试性能，以确保优化有效且无副作用。

参考资料： 本规范参照 Google C/C++ 风格指南、CERT C 安全规范等权威来源编制。其中各项规则已在描述中通过链接形式标明参考出处等。所有示例均为说明原则之用。维护团队应定期复核并更新规范，确保与最新的行业最佳实践保持一致。