中国科学技术大学

模块化程序设计

徐 伟

E-mail: xuweihf@ustc.edu.cn

课程概述

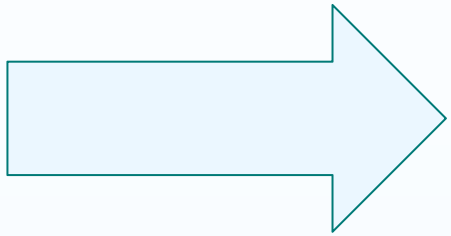
- 模块化编程概念与原则
- 头文件设计与使用规范
- 多文件编译与链接

模块化编程概念与原则

- 模块
  - 硬件模块
    - 计算机的某一种或几种特定功能，做成一个独立的、可以插拔的硬件部件
    - 内存、硬盘、CPU、显卡…
  - 软件模块
    - 将复杂的软件程序，根据功能，拆分成一个个独立的、可以互相协作的**代码块**
    - 每个模块负责一项具体的任务，对外隐藏了内部的实现细节，只留下一些“**接口**”供其他模块使用
    - 函数、类、库…

单文件与多文件开发

- 单文件
  - 代码全在一个文件
  - 快速原型验证
  - 个人项目足够



- 多文件
  - 代码复用
  - 独立测试
  - 迭代演进
  - 避免冲突

核心问题

如何把一个大程序拆成多个小模块
使其易维护、可复用、可测试？

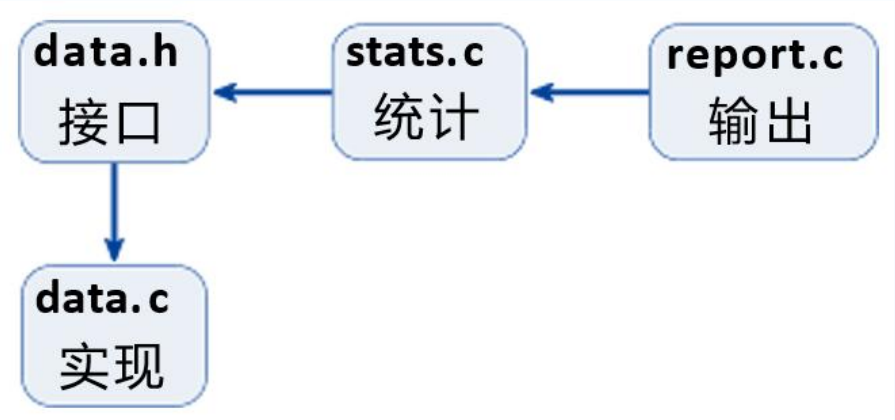
模块化设计三原则

- 高内聚 (High Cohesion)
  - 同一模块只做一类事，功能单一、职责明确
- 低耦合 (Low Coupling)
  - 模块间依赖最少、依赖方向稳定
- 信息隐藏 (Information Hiding)
  - 对外只承诺接口，对内可自由更换实现

模块化设计三原则

高内聚、低耦合、信息隐藏

小例子：成绩统计程序



高内聚：stats.c只做统计，data.c只负责读取

低耦合：stats.c只依赖data.h 接口，不关心文件格式

信息隐藏：data.h只声明struct Data

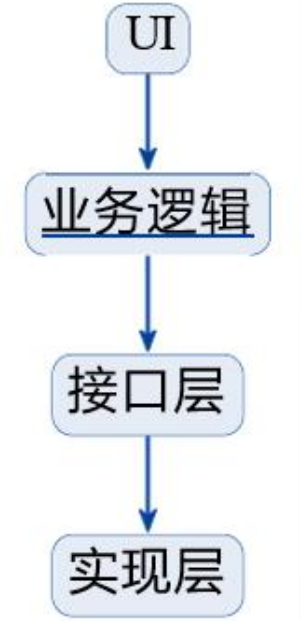
模块边界如何定

- 把经常变化的东西隔离起来
  - 配置、平台差异、算法选择、 I/O
- 把可复用的能力抽成模块
  - 容器（vector/map）、字符串处理、日志、错误处理
- 让依赖方向“朝稳定处走”
  - 业务依赖接口，不依赖具体实现文件
- 经验法则
  - 模块对外只暴露最少必须的信息（类型名/函数/常量）

小技巧
先写.h，再写 .c：逼自己先想清楚“别人怎么用”。

模块化设计

- 不好的模块化设计
  - 改一个结构体字段，很多文件都要改
  - 需要“到处加 include”才能编译
  - 新增功能要改很多旧代码（回归风险高



- 好的模块化设计
  - 改动集中在少数文件
  - 依赖链短、方向清晰
  - 新增功能主要是“加代码”不是“改代码”

写 .h 之前先做接口设计

- 接口五问
  - 这个模块提供什么能力 ？（最小集合）
  - 输入/输出是什么？返回值是否足够表达错误 ？
  - 资源/所有权怎么交接？谁负责 create/destroy ？
  - 哪些内容必须暴露为类型？哪些可以只暴露“句柄”（不透明指针）？
  - 依赖哪些别的模块？能不能只依赖“抽象”（前向声明/回调）？

问题1模块能力

- 这个模块提供什么能力？（最小集合）
  - 含义：明确模块的核心功能，避免“什么都做”
  - 例子： vector 模块只提供：创建、销毁、添加元素、获取元素、查询大小
  - 反例：在 vector 模块里加排序、搜索等功能
    - 这些应该是独立模块

问题2输入输出

- 输入/输出是什么？返回值是否足够表达错误？
  - 含义：函数参数和返回值要能表达所有可能的情况
  - 好例子： int vec_push(Vec* v, int x) 返回0成功， -1失败
  - 坏例子： void vec_push(Vec* v, int x) 无法知道是否成功

问题3资源交接

- 资源/所有权怎么交接？谁负责 create/destroy ？
  - 含义：明确内存由谁分配、谁释放，避免内存泄漏或重复释放
  - 规则：谁创建谁销毁——vec_create() 创建， vec_destroy() 销毁
  - 重要：调用方必须知道何时调用 destroy，否则会内存泄漏

问题4暴露内容

- 哪些内容必须暴露为类型？哪些可以只暴露“句柄”（不透明指针）？
  - 含义：“句柄”就是不透明指针，调用方看不到内部结构
  - 好处：隐藏实现细节，修改内部字段不影响调用方
  - 例子：
    - typedef struct Vec Vec; ——只暴露类型名（句柄）
    - 结构体定义放在 .c 文件里，调用方看不到
- 何时暴露完整类型：调用方需要栈上分配、或访问字段时

问题4暴露内容

- 什么是句柄（Handle）
  - 用于标识和访问某个资源的间接引用
  - 通常是一个不透明的值（比如整数、指针或对象引用），外部代码只能持有这个值，而无法直接看到或操作它所指向的内部数据结构

```
#include <stdio.h>

int main() {
    // 获取文件句柄
    FILE* fileHandle = fopen("example.txt", "r");
    if (fileHandle == NULL) {
        // 错误处理
        return 1;
    }
    …
```

FILE* 就是一个典型的句柄
虽然被实现为指针，但内部结构（如缓冲区位置、文件描述符）对普通程序员是隐藏的
只能通过 fopen、fread、fclose 等函数来操作它。

问题4暴露内容

暴露结构体

```
// vec.h
struct Vec {
        int* data;
        int size;
        int capacity;
};
```

问题

- ►调用方可以直接访问 v->size
- ►修改字段触发大量重编译
- ►难以改变内部实现

不透明指针

```
// vec.h
typedef struct Vec Vec;

size_t vec_size(const Vec* v);
```

好处

- ► 调用方只能通过vec_size()使用
- ► 修改内部不影响 .h
- ► 可以随时优化实现

问题4暴露内容

- 什么时候应该暴露具体类型？
  - 值语义与栈上分配
    - 经常被复制、比较、直接访问成员，暴露具体类型可以避免堆分配和间接访问的开销，性能更高
  - 需要直接操作数据成员
    - 库文件调用，调用方可能需要直接读写成员变量
  - 需要与其他模块交换数据
    - 交换数据是纯数据结构，
  - 编译时依赖无法避免
    - 调用方需要继承类或实现接口

问题4暴露内容

- 什么时候应该使用句柄（不透明指针）？
  - 模块内部实现可能频繁变化
    - 数据库连接池、网络会话、复杂的数据结构（红黑树、哈希表）。使用句柄，可以替换内部算法或存储方式
  - 需要限制调用方对内部状态的直接操作
    - 强制调用方通过提供的API操作对象，可以确保数据完整性
  - 跨语言或跨平台编程
    - 屏蔽不同语言/平台的实现差异

问题4暴露内容

	暴露具体类型	使用句柄
优点	直接访问，无间接开销；适合简单数据结构；方便调试	隐藏实现；降低耦合；增强模块独立性；支持二进制兼容
缺点	任何内部改动都可能导致调用方重新编译/修改；破坏封装	增加一层间接（可能影响性能）；需要额外API管理生命周期；调试需要符号信息
最佳实践	在公开的API边界上，优先使用句柄（或抽象接口）；在模块内部实现中，可以自由使用具体类型。这也是“开闭原则”和“依赖倒置原则”的体现。	

问题5依赖关系

- 依赖哪些别的模块？能不能只依赖“抽象”（前向声明/回调）？
  - 含义：减少模块间的直接依赖，提高灵活性
  - 前向声明：只用指针时，不需要 #include 完整定义
  - 回调：用函数指针代替直接调用，调用方提供具体实现
  - 好处：降低耦合，模块更容易独立编译和测试

前向声明

```
// 在头文件 widget.h 中
struct Widget;  // 前向声明：Widget 是一个结构体类型，但不知道成员

// 使用指针操作，因为指针大小已知
void widget_process(struct Widget* w);
```

```
// 在源文件 widget.c 中真正定义
struct Widget {
    int id;
    char name[32];
    // ... 其他成员
};
```

外部代码只能通过 widget_process 函数操作 struct Widget*，无法直接访问成员，达到了信息隐藏

前向声明

- 为什么需要前向声明
  - 解决相互依赖
    - 当两个模块互相引用时，必须有前向声明才能编译通过
  - 隐藏实现
    - 头文件中，只给出类型的前向声明，定义放源文件，外部无法看到内部细节，只能通过函数操作该类型（对应之前讨论的“句柄”模式）
  - 减少编译依赖
    - 头文件只需要知道某个类型存在（比如作为指针参数），不需要知道其大小或成员
    - 用前向声明代替包含完整的定义，避免不必要的重新编译

前向声明的限制

- 当编译器需要知道结构体大小时，必须有完整的定义

可以使用前向声明

```
// 只用指针
typedef struct Vec Vec;
void process(Vec* v);

struct Data {
    Vec* vec;
};
```

指针大小固定（通常 8 字节），编译器不需要知道 Vec 的完整结构

必须include完整定义

```
// 需要知道大小
void process(Vec v);  // 错误!
struct Data {
    Vec vec;  // 错误!
};

int size = sizeof(Vec);  // 错误!
v->data[0] = 1;  // 错误!
```

编译器需要知道 Vec 的大小和字段布局

回调

```
#include <stdio.h>

// 定义回调函数类型
typedef void (*EventHandler)(int eventCode);

// 一个函数，接受回调作为参数
void simulateEvent(EventHandler callback) {
    printf("模拟事件发生...\n");
    callback(42);  // 调用回调，传入事件码
}

// 实际的事件处理函数
void myHandler(int code) {
    printf("事件处理：收到代码 %d\n", code);
}

int main() {
    simulateEvent(myHandler);  // 传递回调
    return 0;
}
```

回调

- 回调（Callback）
  - 一段可执行的代码（通常是函数或方法）作为参数传递给另一个函数或模块，以便在将来某个时刻被调用

- 为什么需要回调
  - 实现异步处理
  - 提高灵活性
  - 事件驱动编程

前向声明与回调

	前向声明	回调
核心作用	提前告知编译器某个类型/函数存在，允许引用而不需完整定义	将可执行代码作为参数传递，实现反向调用和定制行为
目的	解决依赖、隐藏实现、减少编译耦合	解耦调用者与被调用者，增强灵活性
常见场景	句柄类型声明、相互递归的结构体、接口头文件设计	事件处理、异步操作、算法定制（如排序比较器）

课程概述

- 模块化编程概念与原则
- 头文件设计与使用规范
- 多文件编译与链接

思考题

- 如果把所有结构体细节都放到.h，会发生什么？

思考题

- 如果把所有结构体细节都放到.h，会发生什么？
  - 封装性破坏
  - 编译依赖爆炸，构建变慢
  - 兼容性丧失
  - 安全风险
  - …

为什么修改头文件结构体导致重新编译？

场景：结构体定义在头文件中

```
// vec.h (暴露了结构体细节)
struct Vec {
   int* data;
   int  size;          // 假设我们要修改这个字段
   int  capacity;
   };
```

有 3 个文件包含了这个头文件

```
// a.c b.c c,c都包含了 vec.h
#include "vec.h"
```

当修改结构体字段时 （比如把 int size 改成 size_t size）会如何？

为什么修改头文件结构体导致重新编译？

- 原因分析
  - 预处理机制
    - #include是文本替换，头文件内容被“复制”到每个包含它的 .c 文件中
  - 依赖传播
    - 如果 vec.h 被 a.c, b.c, c.c 包含，修改 vec.h 会导致这 3 个文件都需要重新编译
  - 时间戳检查
    - Make等构建工具会检查文件修改时间，发现 .h 更新后会重新编译所有依赖它的.c

对比：使用不透明指针

改进方案：使用不透明指针（Opaque Pointer）

```
// vec.h（只暴露类型名）
typedef struct Vec Vec;        //前向声明，  不暴露细节
Vec* vec_create(void);
void vec_destroy(Vec* v);
-----------------------------------------------------------------
// vec.c（结构体定义放在这里）
struct Vec {
    int* data;
    size_t size;     // 修改这里不影响 vec.h
    size_t capacity;   };
```

好处：修改 `vec.c` 中的结构体字段， `vec.h` 不变

如何减少重新编译

- 使用前向声明
  - 减少头文件中的include
- 信息隐藏
  - 实现细节放到.c，只在.h中暴露必要的接口
- 合理组织模块
  - 避免大而全的头文件

推荐的头文件组织

```
Project/
    include/                #对外API（安装/发布）
        vec.h
        logger.h
    src  #实现
        vec.c
        logger.c
        internal/           #私有头（仅src内部使用）
            vec_impl.h
    tests/                  #测试/示例（可选）
        test_vec.c
```

规则

- `include/`：只放稳定接口
- `src/internal/`：实现细节可以随时改
- 公共头文件不 include 私有头

收益

- 依赖方向清晰
- 更易信息隐藏
- 编译更快（减少include）

头文件模板

```
// vec.h（公共接口）
#ifndef VEC_H
#define VEC_H

#include <stddef.h>        // 只include自己真的需要的

typedef struct Vec Vec;   // 不透明指针：隐藏实现细节

Vec* vec_create(void);
void  vec_destroy(Vec* v);
int     vec_push(Vec* v, int  x);     // 返回错误码
size_t vec_size(const Vec* v);

#endif   // VEC_H
```

`#ifndef/#define`： 包含保护，防止头文件被重复包含导致重复定义

.h与.c职责边界

.h文件：对外接口

- 类型定义（typedef, struct）
- 宏常量（#define）
- 函数声明
- 必要的注释说明

.c文件：内部实现细节

- 私有函数
- 私有数据
- 算法与优化
- 内部实现

只暴露”用得上的声明”，不暴露”能暴露的一切”

头文件设计原则

- 包含保护
  - #ifndef/#define/#endif 或 #pragma once
- 自洽
- 依赖最小化
- 禁止循环包含
- 避免变量定义
- 避免函数实现
- 模块合理命名
- …

自洽

- 一个头文件是自洽（Self-Contained）的，说明：
  - 它独自被#include时能够编译通过，不依赖其他头文件被提前包含
  - 它显式地包含了直接使用的所有类型、宏、函数的声明

自洽

- 一个头文件是自洽（Self-Contained）的，说明：
  - 它独自被#include时能够编译通过，不依赖其他头文件被提前包含
  - 它显式地包含了直接使用的所有类型、宏、函数的声明

反例：隐式依赖

```
// vec.h
void vec_add(Vector* v, size_t n);  // 使用了 size_t，但没有包含 <stddef.h>
```

若某个.c文件

```
#include "vec.h"                    // 编译错误！ size_t 未定义
```

必须改为

```
#include <stddef.h>
#include "vec.h"   // 正确，但依赖包含顺序
```

“必须先包含 A 才能包含 B”的依赖就是**不自洽**

自洽

- 自洽（Self-Contained）重要性
  - 避免隐式顺序约束
    - 使用者无需记住头文件的包含顺序，降低出错概率。
  - 提高可维护性
    - 修改头文件时，不会因为依赖关系隐藏而破坏现有代码。
  - 简化代码审查和重构
    - 每个头文件都是独立的单元，易于理解和测试

自洽

- 一个头文件是自洽（Self-Contained）的，说明：
  - 它独自被#include时能够编译通过，不依赖其他头文件被提前包含
  - 它显式地包含了直接使用的所有类型、宏、函数的声明

自洽性测试

写一个简单的测试文件 test_vec.c:

```
#include "vec.h"   // 仅包含这一行
```

如果能编译通过，说明头文件是自洽的

自洽的头文件 = 独立、可靠、易用的接口

依赖最小化

- 采用前向声明等方式，减少依赖
  - 只声明类型名，不包含完整定义
  - 减少依赖，只用指针时不需要`#include`

```
// a.h
typedef struct A A;          // 前向声明
typedef struct B B;          // 前向声明

void a_do_something(A* a, B* b);  // 只使用指针，不需要完整定义
```

禁止循环包含

- 循环包含
  - 头文件 A 直接或间接地包含了头文件 B，而头文件 B 又直接或间接地包含了头文件 A

```
// a.h
#include "b.h"

// b.h
#include "a.h"
```

- 循环包含的问题
  - 编译错误
  - 设计耦合
  - 难以维护

为什么头文件保护（#ifndef）不能解决循环包含？

- 头文件保护可以防止同一个头文件被**重复包含**，但无法打破循环依赖的逻辑

```
// a.h
#ifndef A_H
#define A_H
#include "b.h"
typedef struct A { B* b; } A;
#endif

// b.h
#ifndef B_H
#define B_H
#include "a.h"
typedef struct B { A* a; } B;
#endif
```

当编译 **a.c** 包含 **a.h** 时，会发生什么？

如何避免循环包含

- 使用前向声明（Forward Declaration）代替包含
- 将互相依赖的类型放到同一个头文件中
- 检查循环包含的工具
  - 头文件依赖检查软件（如include-graph等）
- 使用#pragma once
  - 不能解决设计上的循环
  - 可以防止同一文件多次包含，配合前向声明使用更简洁

#pragma once工作原理

- 预处理器处理到#pragma once时，它会记录当前文件的唯一标识。当后续再次遇到#include指向同一个文件时，预处理器会检查该文件是否已经在列表中：
  - 如果已经在列表中，则完全跳过该文件的内容
  - 如果不在列表，则正常展开文件内容，并将其加入列表

#pragma once工作原理

- 预处理器处理到#pragma once时，它会记录当前文件的唯一标识。当后续再次遇到#include指向同一个文件时，预处理器会检查该文件是否已经在列表中：
  - 如果已经在列表中，则完全跳过该文件的内容
  - 如果不在列表，则正常展开文件内容，并将其加入列表

- 不依赖宏定义
  - 传统的头文件保护（#ifndef）需要手动定义一个宏，并检查该宏是否存在
  - #pragma once 是编译器直接提供的机制，不需要程序员操心宏名的唯一性

#pragma once工作原理

```
// example.h
#pragma once   // 只出现一次，无需额外宏

struct Example {
    int value;
};
```

#pragma once工作原理

```
// a.h
#pragma once
#include "b.h"
typedef struct A { B* b; } A;
#endif

// b.h
#pragma once
#include "a.h"
typedef struct B { A* a; } B;
#endif
```

当编译 **a.c** 包含 **a.h** 时，会发生什么？

用**#pragma once**能否解决这个问题？

#pragma once工作原理

正确解决办法

```
// a.h
#pragma once
struct B;  // 前置声明，告诉编译器 B 是一个结构体类型
typedef struct A { struct B* b; } A;  // 使用 struct B*（B 尚不完整，但指针允许）
#include "b.h"  // 如果需要使用 B 的完整定义，可以放在这里，但通常可以省略


// b.h
#pragma once
struct A;  // 前置声明
typedef struct B { struct A* a; } B;
…
```

避免变量定义

- 除非static const或extern声明
  - static变量具有内部链接，即只在定义它的编译单元（当前 .c 文件及其直接或间接包含的头文件）内可见
  - extern声明变量而不定义，告诉编译器变量在别处定义

```
// counter.h
extern int g_count;      // 只是声明，不分配内存


// counter.c
int g_count = 0;         // 真正的定义，分配内存
```

避免函数实现

- 当头文件被多个 .c 文件包含时，导致多重定义
  - 链接时，链接器发现多个相同的符号，会报“重复定义”错误
- 编译时间增加
- 代码膨胀
- 破坏封装

避免函数实现

- **场景： util.h 中定义了函数 add，两个 .c 文件都包含了它**

文件结构

```
# util.h
int add(int a, int b) {
    return a + b;
}

# main.c
#include "util.h"
int main()   { return add(1,2); }

# test.c
#include "util.h"
void test ()   { add(3,4); }
```

编译过程

```
# 编译 main.c
gcc -c main.c → main.o
main.o 包含  add 的定义
# 编译 test.c
gcc -c test .c → test .o
test .o 也包含 add 的定义
# 链接
gcc main.o test .o -o app
错误！ 链接器发现 add 被定义了2
次   → multiple definition of 'add '
```

模块合理命名

- 避免不通模块的函数名冲突（C语言没有命名空间）
- 同一模块的所有函数用统一前缀（如 vec_*、 logger_*）

例：

```
vec_create()
vec_push()
vec_size()
```

头文件常见警告和错误

- Error：编译/链接失败，无法生成可执行文件
- Warning：可能有问题，但仍能编译成功

Error （编译/链接期错误）

- multiple definition of ...
  阶段：链接期
  结果：无法生成可执行文件
- undefined reference to ...
  阶段：链接期
  结果：找不到某个符号的具体定义，无法生成可执行文件
- unknown type name 'X'
  阶段：编译期
  结果：从未见过 X 的定义或声明，无法生成.o

Warning（警告）

- unused variable
  阶段：编译期
  结果：可以编译，但提示有未使用的变量
- format specifies type 'int' but the argument has type 'long'
  阶段：编译期
  结果：可以编译，但提示有类型不匹配的变量

头文件常见警告和错误

multiple definition of ...
同名定义出现在多个翻译单元
常见原因：在 .h 写了变量/函数定义

undefined reference to ...
声明存在但实现没参与链接
常见原因：漏编译某个 .c / 库链接顺序不对

include循环
A 包含 B， B 又包含 A
解决方法：用前向声明

头文件常见警告和错误

预处理： #include 是文本替换

► 头文件内容会被“复制粘贴”进每个 .c

► 所以：循环包含、宏污染、头文件变 “胖”都会产生连锁效应

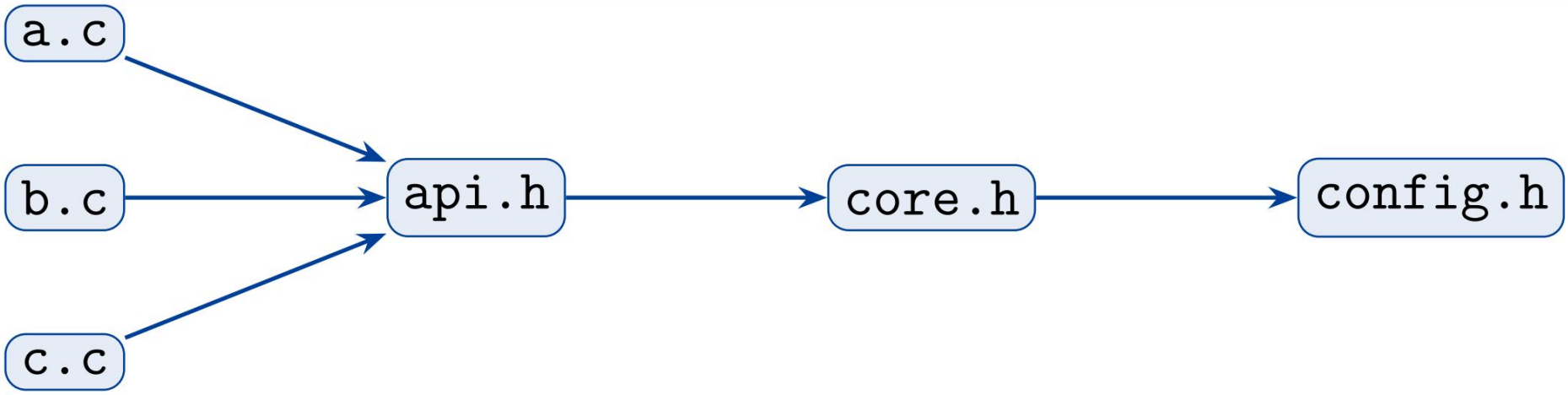
编译：每个 .c 都是独立翻译单元

► 每个 .c 单独编译成 .o

► 如果头文件里放了“定义”，每个 .o 都会各自带一份

► 链接时就会出现 multiple definition

重编译风暴的消除



- #include 是文本替换，会带来传递依赖
- 头文件越 “胖”，重编译范围越大
- 头文件规范（前向声明/最小包含）能显著减少耦合

重编译风暴的消除

- **场景**：修改了 `config.h`

依赖链条

文件包含关系

a.c includes api.h

api.h includes core.h

core.h includes config.h

传递依赖

a.c → api.h → core.h → config.h

解释：

- a.c 包含 api.h
- api.h 包含 core.h
- core h 包含 config h
- 所以 a.c 间接依赖 config.h

重编译过程

修改 config.h 后

1. config.h 时间戳更新
2. core.h 依赖 config.h → 重编译
3. api.h 依赖 core.h → 重编译
4. a.c, b.c, c.c 依赖api.h →全部重编译!

结果：

- 修改 1 个文件（config.h）
- 触发 3 个.c 文件重编译
- 这就是 ” 重编译风暴 ” ！

重编译风暴的消除

- **核心思想： 减少头文件依赖，使用前向声明**

不好的做法

```
// api.h（暴露太多）
#include "core.h" // 完整包含
#include "config.h"
#include "logger.h"
#include "utils.h"
typedef struct Data {
   CoreType core; // 需要完整定义
} Data;
```

问题：

►包含了 4 个头文件

好的做法

```
// api.h（最小包含）
#include <stddef.h> // 只包含必需的

// 前向声明， 不包含头文件
typedef struct CoreType CoreType;
typedef struct Data {
    CoreType* core; // 只用指针
} Data;
```

好处：

►只包含 1 个标准库头文件

课程概述

- 模块化编程概念与原则
- 头文件设计与使用规范
- 多文件编译与链接

什么是makefile

- Small programs → single file

- 项目规模不断增大
  - Many lines of code
  - Multiple components
  - More than one programmer

- 项目规模增大，带来的问题
  - 长文件难于管理
  - 每次修改均需要长时间编译
  - 多开发人员无法同时修改同一文件

什么是makefile

- 解决办法
  - 将工程拆分到多个源文件
- 目标
  - 对模块进行良好切分
  - 缩短编译时间，仅对修改的部分进行重新编译
  - 易于维护项目结构
    - 依赖和创建

什么是makefile

- 办法采用makefile
  - 使用make命令和makefile工具
  - 理顺各个源文件之间纷繁复杂的相互关系

makefile安装

- Centos
  - sudo yum -y install gcc automake autoconf libtool make
  - sudo yum install gcc gcc-c++
- Ubuntu
  - sudo apt-get update
  - sudo apt-get install make

Makefile基础

- 显示规则
  - 目标文件：依赖文件
  - 原则上第一个文件就是最终的文件
  - 缩进必须TAB键（必须的，不可省略）

```
1  test:test.o      #目标文件test，依赖文件test.o
2      gcc test.o -o test
3  test.o:test.s
4      gcc -c test.s -o test.o
5  test.s:test.i
6      gcc -S test.i -o test.s
7  test.i:test.c
8      gcc -E test.c -o test.i
```

假设单文件 test.c 编译

采用递归方式书写

使用make命令自动编译出目标文件test

#表示注释

Makefile基础

- 变量
  - 变量一般定义在文件头部
  - 使用$(变量名)表示变量

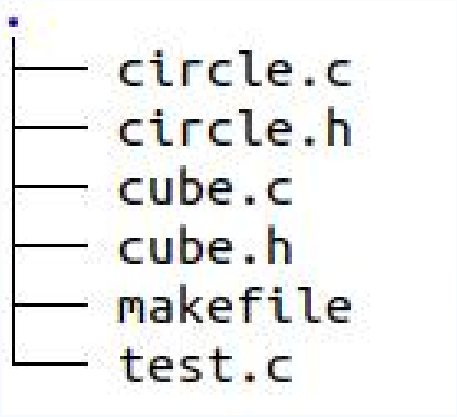
符号	含义
=	赋值
+=	增加赋值
:=	常量赋值

```
1  TAR = test          #赋值
2  CC := gcc           #常量赋值
3
4  $(TAR):test.o       #常量使用
5      $(CC) test.o -o $(TAR)
6  test.o:test.s
7      $(CC) -c test.s -o test.o
8  test.s:test.i
9      $(CC) -S test.i -o test.s
10 test.i:test.c
11     $(CC) -E test.c -o test.i
```

Makefile基础

- 多文件编译

  示例circle.c circle.h cube.c cube.h test.c编译



Makefile基础

- 多文件编译

示例circle.c circle.h cube.c cube.h test.c编译

```c
#include <stdio.h>

int circle();
```

circle.h

```c
#include "circle.h"

int circle()
{
    printf("This is cicle fun\n");
    return 0;
}
```

circle.c

```c
#include <stdio.h>

int cube();
```

cube.h

```c
#include "cube.h"

int cube()
{
    printf("This is cube fun\n");
    return 0;
}
```

cube.c

Makefile基础

- 多文件编译

  示例circle.c circle.h cube.c cube.h test.c编译

test.c

```c
#include "cube.h"
#include "circle.h"

int main()
{
    cube();
    circle();
    printf("This is a test program\n");
    return 0;
}
```

Makefile基础

- 多文件编译

  示例circle.c

  circle.h

  cube.c

  cube.h

  test.c编译

```
1  TAR = test
2  CC := gcc
3
4  $(TAR):circle.o cube.o test.o
5      $(CC) circle.o cube.o test.o -o $(TAR)
6  circle.o:circle.c
7      $(CC) -c circle.c -o test.o
8  cube.o:cube.c
9      $(CC) -c cube.c -o cube.o
10 test.o:test.c
11     $(CC) -c test.c -o test.o
12
13 .PHONY:      #伪目标
14 clean:       #make clean调用
15     rm -rf circle.o cube.o test.o test
```

Makefile基础-通配符

通配符	含义
%	模式字符，用来通配任意个字符
$@	目标文件名。多目标模式规则中，它代表的是触发规则被执行的文件名
$%	当目标文件是一个静态库文件时，代表静态库的一个成员名
$<	第一个依赖的文件名
$?	所有比目标文件更新的依赖文件列表
$^	代表的是所有依赖文件列表
$+	保留了依赖文件中重复出现的文件。主要用在程序链接时库的交叉引用场合

Makefile基础-通配符

```
1  TAR = test
2  CC := gcc
3
4  $(TAR):circle.o cube.o test.o
5      $(CC) circle.o cube.o test.o -o $(TAR)
6  circle.o:circle.c
7      $(CC) -c circle.c -o test.o
8  cube.o:cube.c
9      $(CC) -c cube.c -o cube.o
10 test.o:test.c
11     $(CC) -c test.c -o test.o
12
13 .PHONY:      #伪目标
14 clean:       #make clean调用
15     rm -rf circle.o cube.o test.o test
```

make clean　清除

```
1  TAR = test
2  OBJ = circle.o cube.o test.o
3  CC := gcc
4
5  $(TAR):$(OBJ)
6      $(CC) $^ -o $@
7  %.o:%.c
8      $(CC) -c $^ -o $@
9
10 .PHONY:      #伪目标
11 clean:       #make clean调用
12     rm -rf $(OBJ) $(TAR)
```

通配符 $@　$^

Makefile基础

- 条件判断

If　　ifeq　　ifneq　　ifdef　　ifndf

以endif结尾

复杂条件判断使用elif和else

Example:

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif
```

Makefile工作

- Make命令工作顺序
  - make在当前目录下寻找“Makefile”或“makefile”文件
  - 若找到，查找文件中的第一个目标文件.o
  - 若目标文件不存在，根据依赖关系查找.s文件
  - 若.s文件不存在，根据依赖关系查找.i文件
  - 若.i文件不存在，根据依赖关系查找.c文件
  - 若.c文件一定存在，于是按顺序依次生成.i、.s、.o文件，再去执行

什么是CMake

- 不同平台编译所使用的Makefile标准、格式不同
- CMake编写一种与平台无关的文件来指导整个编译流程，根据目标平台生成所需要的Makefile和工程文件
- CMake具有最小的依赖性，只需要在工作系统上安装编译器即可

Cmake工作的流程

在linux平台下使用CMake生成Makefile并编译的流程如下：

- 编写CMake配置文件CMakeLists.txt
- 执行命令 cmake PATH 或者 ccmake PATH 生成 Makefile（ccmake 和 cmake 的区别在于前者提供了一个交互式的界面）。其中， PATH 是CMakeLists.txt 所在的目录
- 使用 make 命令进行编译

CMake安装

- Windows可以去cmake官下载安装包来进行安装 (https://cmake.org/download/)
- Ubuntu系统上可以使用包管理工具apt（或apt-get）进行安装

  sudo apt install cmake
- 指定版本的cmake，可以手动下载源码编译/二进制安装包

推荐的CMake学习资料

- https://cmake.org/cmake/help/latest/guide/tutorial/index.html
- https://www.zhihu.com/search?type=content&q=cmake
- https://cliutils.gitlab.io/modern-cmake/
- https://modern-cmake-cn.github.io/Modern-CMake-zh_CN/

多文件编译错误排查

1）报错是编译期还是链接期？（undefined reference / multiple definition 属于链接期）

↓

2）缺的符号在哪里定义？（用 rg / nm 查）

↓

3）定义所在的 .c 是否被编译成 .o ？

↓

4）该 .o 或库是否参与链接？顺序是否正确？

↓

5）如果是多重定义：是否在头文件写了定义？是否重复链接同一对 象/库？

多文件编译错误排查

错误信息	常见原因	优先排查
`undefined reference`	漏编译/漏链接；库顺序错误；声明与定义不一致	目标文件是否参与链接； `-L/-l`是否正确； 符号名是否拼写一致
`multiple definition`	在头文件放了定义；同名全局变量/函数重复定义	是否把变量定义写进 `.h`； 是否缺少`extern`； 是否重复链接了同一对象文件

目标文件内容

部分	含义
.text	机器指令（函数代码）
.data	已初始化的全局/静态变量
.bss	未初始化的全局/静态变量（占位，不占文件体积）
符号表	“我提供哪些符号 / 我还缺哪些符号”

链接器做的事

把多个 .o 拼起来： 符号解析 + 地址重定位

多文件编译错误排查

- 现象

undefined reference to ...

nm 列出目标文件（.o 文件、静态库 .a、可执行文件）的符号表，每个符号前有一个字母表示其类型

多文件编译错误排查例子-nm

1.1 编译 vs 链接（为什么会有 undefined reference）

编译（compile）：把每个 .c 单独翻译成目标文件 .o。

这个阶段只需要知道函数“长什么样”（声明），不需要看到函数“怎么实现”（定义）。

链接（link）：把多个 .o（以及库 .a/.so）合并成一个可执行文件。

这个阶段必须把每一个“被引用的外部符号”都找到对应的实现。

如果在 main.c 里调用了 vec_create()，但链接时忘记把实现它的 vec.o（或库）加进去，链接器就会报：

undefined reference to 'vec_create'

多文件编译错误排查例子-nm

1.2 什么是“符号（symbol）”

在目标文件/库中，函数名、全局变量名等都会变成符号记录在符号表里。

链接器工作的核心就是：把引用符号和定义符号配对。

多文件编译错误排查例子-nm

例子

工程结构是：

main.c：调用 vec_* 函数

vec.c：实现 vec_* 函数

vec.h：对外提供声明（接口），并且用了 opaque pointer（隐藏结构体字段）

所以：

main.o 里会出现 U vec_create / U vec_push ...

vec.o 里会出现 T vec_create / T vec_push ...

多文件编译错误排查例子-nm

3.1 正常情况：能编译能运行

make clean

make

./app

```
make clean
make
./app
rm -f *.o app
cc -Wall -Wextra -std=c11 -g -c vec.c
cc -Wall -Wextra -std=c11 -g -c main.c
cc vec.o main.o -o app
0: 0
1: 1
2: 4
3: 9
4: 16
5: 25
6: 36
7: 49
8: 64
9: 81
```

多文件编译错误排查例子-nm

3.2 故意制造错误：漏链接 vec.o → undefined reference

gcc -Wall -Wextra -std=c11 -g -c main.c -o main_demo.o

gcc main_demo.o -o app_broken

```
$ cc -Wall -Wextra -std=c11 -g -c main.c -o main_demo.o

cc main_demo.o -o app_broken
/usr/bin/ld: main_demo.o: in function `main':
/home/ubuntu/2026-friday/ch1/opaque_vec/main.c:7:(.text
+0x1c): undefined reference to `vec_create'
/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec/ma
in.c:8:(.text+0x3f): undefined reference to `vec_push'
/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec/ma
in.c:13:(.text+0x71): undefined reference to `vec_get'
/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec/ma
in.c:11:(.text+0xa4): undefined reference to `vec_size'
/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec/ma
in.c:16:(.text+0xb6): undefined reference to `vec_destr
oy'
collect2: error: ld returned 1 exit status
```

多文件编译错误排查例子-nm

3.3 用 nm 定位：谁引用、谁定义

```
$ nm -C main_demo.o
nm -C vec.o
0000000000000000 T main
                 U printf
                 U __stack_chk_fail
                 U vec_create
                 U vec_destroy
                 U vec_get
                 U vec_push
                 U vec_size
                 U calloc
                 U free
                 U realloc
0000000000000081 T vec_create
00000000000000a6 T vec_destroy
000000000000017d T vec_get
0000000000000000 t vec_grow
00000000000000dd T vec_push
0000000000000159 T vec_size
```

多文件编译错误排查例子-nm

3.4 正确修复：把 vec.o 链接进去

```
$ cc vec.o main_demo.o -o app_ok
./app_ok
0: 0
1: 1
2: 4
3: 9
4: 16
5: 25
6: 36
7: 49
8: 64
9: 81
```

多文件编译错误排查

- 现象

undefined reference to ...

rg（ripgrep）是超快的代码搜索工具，用来在源码目录中查找符号的定义或引用

相当于 “更现代、更好用、更快的 grep -R”，专门为代码仓库搜索优化。

rg 是“为代码仓库量身定制的 grep”。

多文件编译错误排查-rg

rg 在排查 undefined reference 里扮演什么角色

undefined reference to X 是链接阶段错误，本质在于：

某个文件 引用了符号 X（调用了函数/用了全局变量）

但在链接进去的 .o/.a 里 没找到符号 X 的定义

这时 rg 就非常适合做两件事：

找引用点：哪里写了 X(...) 或使用了 X

找定义点：哪里有 X(...) { ... } 或全局变量定义

然后回到链接命令/Makefile：确认“定义所在的文件/库”有没有被链接进去。

多文件编译错误排查-rg

1) 环境确认：rg 已安装

```
h1/opaque_vec$ rg --version
ripgrep 14.1.0

features:-simd-accel,+pcre2
simd(compile):+SSE2,-SSSE3,-AVX2
simd(runtime):+SSE2,+SSSE3,+AVX2

PCRE2 10.42 is available (JIT is available
)
```

多文件编译错误排查-rg

2) 第一步：复现 undefined reference（故意漏链接）

2.1 先正常编译（可选，用来对照）

```
make clean
make
./app
```

2.2 故意漏链接 vec.o

```
gcc -Wall -Wextra -std=c11 -g -c main.c -o main_demo.o
gcc main_demo.o -o app_broken
```

多文件编译错误排查-rg

```
ubuntu@VM11452-compilerclj:~/2026-friday/ch1/opaque_vec
$ cc -Wall -Wextra -std=c11 -g -c main.c -o main_demo.o

cc main_demo.o -o app_broken 2> link_error_rg.txt
sed -n '1,120p' link_error_rg.txt
/usr/bin/ld: main_demo.o: in function `main':
/home/ubuntu/2026-friday/ch1/opaque_vec/main.c:7:(.text
+0x1c): undefined reference to `vec_create'
/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec/ma
in.c:8:(.text+0x3f): undefined reference to `vec_push'
/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec/ma
in.c:13:(.text+0x71): undefined reference to `vec_get'
/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec/ma
in.c:11:(.text+0xa4): undefined reference to `vec_size'
/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec/ma
in.c:16:(.text+0xb6): undefined reference to `vec_destr
oy'
collect2: error: ld returned 1 exit status
```

多文件编译错误排查-rg

3) 第二步：用 nm 看符号表（谁引用/谁定义）

3.1 看调用者：main_demo.o（应该显示 U vec_*）

```
$ nm -C main_demo.o | head -n 40
0000000000000000 T main
                 U printf
                 U __stack_chk_fail
                 U vec_create
                 U vec_destroy
                 U vec_get
                 U vec_push
                 U vec_size
```

多文件编译错误排查-rg

3.2 看实现者：vec.o（应该显示 T/t vec_*）

```
$ nm -C vec.o | head -n 80
                 U calloc
                 U free
                 U realloc
0000000000000081 T vec_create
00000000000000a6 T vec_destroy
000000000000017d T vec_get
0000000000000000 t vec_grow
00000000000000dd T vec_push
0000000000000159 T vec_size
```

多文件编译错误排查-rg

4) 用 rg 在多文件中定位“引用点/声明/定义”

4.1 一次搜索所有报错符号

rg -n "vec_create|vec_push|vec_get|vec_size|vec_destroy" -S .

```
$ rg -n "vec_create|vec_push|vec_get|vec_size|vec_destr
oy" -S .
./demo_nm_rg_full.md
65:/home/ubuntu/2026-friday/ch1/opaque_vec/main.c:7:(.t
ext+0x1c): undefined reference to `vec_create'
66:/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec
/main.c:8:(.text+0x3f): undefined reference to `vec_pus
h'
67:/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec
/main.c:13:(.text+0x71): undefined reference to `vec_ge
t'
68:/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec
/main.c:11:(.text+0xa4): undefined reference to `vec_si
ze'
69:/usr/bin/ld: /home/ubuntu/2026-friday/ch1/opaque_vec
/main.c:16:(.text+0xb6): undefined reference to `vec_de
stroy'
95:                 U vec_create
96:                 U vec_destroy
97:                 U vec_get
98:                 U vec_push
99:                 U vec_size
```

多文件编译错误排查-rg

5) 第四步：修复（正确链接）

gcc vec.o main_demo.o -o app_ok

./app_ok

```
$ cc vec.o main_demo.o -o app_ok
./app_ok
0: 0
1: 1
2: 4
3: 9
4: 16
5: 25
6: 36
7: 49
8: 64
9: 81
```

典型场景示例

- 场景：忘记实现函数

```
// calc.h
int add(int a, int b);
```

```
// main.c
#include "calc.h"
int main() { return add(1,2); }
```

编译 gcc main.c → 报错 undefined reference to 'add'。

用 nm -u main.o 看到 U add。

```
/add_missing_impl_demo$ gcc -c main.c
nm -u main.o
                 U add
```

用 rg 'add\(' 搜索发现只有声明没有定义。添加 calc.c 实现后重新编译即可

```
/add_missing_impl_demo$ rg 'add\('
main.c
2:int main() { return add(1,2); }
```

多文件编译错误排查

- 现象

undefined reference to ...，但确定库是对的

```
# 常见正确顺序： 先对象文件， 后库
gcc main.o math.o -lm -o app

# 错误顺序： 库放在前面不起作用
gcc -lm main.o math.o -o app
```

链接器通常从左到右处理：需要某个符号时，才会从后面的库里“把用到的那部分拉进来”

本章小结

- 理解模块划分原则
- 写出规范的头文件 （包含保护、自洽可编译）
- 用命令行完成多文件编译与链接
- 常见链接错误排查与解决

谢谢！

中国科学技术大学
University of Science and Technology of China