程序设计训练之 Rust 编程语言

第一次习题课

陈嘉杰，郑友捷

清华大学计算机科学与技术系

2025 年 8 月 27 日

1

小作业

第一次小作业

- 从标准输入读取：
  - 课上讲的：`std::io::stdin().read_line(&mut line)`
  - 注意 `read_line()` 返回的字符串包括换行符号（LF 0xA `'\n'`）
  - 因此先 `trim()` 再 `parse()`：`line.trim().parse()`
  - OJ 上引入了第三方库 `text_io` 的 `text_io::read!()`
  - 类似 C++ 的 `std::cin`：根据类型来判断要如何解析输入数据
  - 对比 C 的 `scanf`：解析格式化字符串来判断数据格式
- 返回多个参数的函数：
  - C/C++：传指针 `int exgcd(int a, int b, int *x, int *y)`
  - Rust：利用元组 `fn exgcd(a: i64, b: i64) -> (i64, i64, i64)`

第二次小作业

使用迭代器遍历 Vec 的时候，如何修改 Vec 的内容?

```rust
// Task 5
let mut data = vec![100, 200, 300];
// region: Task 5
for i in &data {
    if i == 200 {
        data[0] += 200;
    }
}
// endregion: Task 5

// Expected: "Task 5: [300, 200, 300]"
println!("Task 5: {:?}", data);
```

可变迭代器

第一种场景：需要修改的元素就是正在遍历的元素，此时可以用可变迭代器：

```
for i in &mut data {
    *i += 200;
}
```

此时迭代器一定是有效的，因为 Vec 的元素个数并不会变，不会重新分配内存。

第二种场景：需要修改的元素不是正在遍历的元素：

```
for i in &mut data {
    // cannot borrow `data` as mutable more than once at a time
    data[0] += 200;
}
```

正在遍历数组的时候，想要修改第 0 个元素，怎么办?

解决方法

第一种办法，先记录下来要完成的修改，在循环结束后再修改：

```
let mut count = 0;
for i in &data {
    count += 1;
}
data[0] += 200 * count;
```

第二种办法，用下标访问数组：

```
for i in 0..data.len() {
    data[0] += 200;
}
```

退化为和 C/C++ 一样，无法享受 Rust 提供的迭代器安全性保证。

枚举和模式匹配

- 使用枚举来表示等级

```
enum Grade {
    APlus,
    // ...
    F
}
```

- 使用模式匹配来转换 &str 为 Grade，再转换 Grade 为对应的绩点

```
match self {
    Self::APlus | Self::AMinus => 4.0,
    Self::BMinus => 3.0,
    // ...
}
```

第三次小作业

- 使用 BTreeMap/HashMap 统计出现次数 `*map.entry(key).or_insert(0) += 1`
  - 类比 Python 的 defaultdict，C++ 的 map
  - Wordle 中统计单词出现次数时可以用到
  - 也可以用数组来统计：`let count = [0; N]`
- 迭代器的高级用法
  - 课上讲的：`a.iter().map(|x| x * 2).collect()`，等价于

```
let ret = vec![];
for x in &a {
    ret.push(x * 2);
}
ret
```

  - 迭代器“拉链”：`a.iter().zip(b.iter()).map(|(a, b)| a + b).collect()`
  - 求和：`a.iter().sum()` 或 `a.iter().fold(0, |acc, x| acc + x)`

二叉搜索树

实现一个二叉搜索树：

Rust

```rust
struct Node {
    value: i32,
    left: Option<Box<Node>>,
    right: Option<Box<Node>>,
}

struct Tree {
    root: Option<Box<Node>>,
}
```

C/C++

```cpp
struct Node {
    int value;
    Node *left;
    Node *right;
}

struct Tree {
    Node *root;
}
```

二叉搜索树 - 插入

插入算法的思路：

1. 从根结点开始，按照要插入的值与结点的值的大小关系，进入左子树或右子树
2. 发现空结点，说明找到了要插入的位置，新建一个结点然后更新父结点的左儿子或右儿子

如何用 Rust 实现?

1. 记录当前结点：不希望把结点的所有权移走，所以要用借用；但是最终需要修改结点的内容，还需要是可变借用。
   - 类比 C++ 的 `T *`，既可以修改指向的 `T` 内容，也可以修改指针以指向另一个 `T`
   - Rust 中则是 `let mut current_node: &mut T`
2. 模式匹配：遇到 Option 的时候，需要判断它是 Some 还是 None

二叉搜索树 - 插入

常见错误：

```
// cannot move out of `self.root.0` which is behind a mutable reference
match self.root {
    None => {}
    Some(node) => {
    }
}
```

回忆一下，上课讲的如何在模式匹配时借用：

1. 直接对借用变量进行模式匹配：`match &self.root`
2. 模式匹配时使用 `ref`：`Some(ref node)`

还有一种方法：`Option::as_ref` 把 `&Option<T>` 转换为 `Option<&T>`

二叉搜索树 - 插入

常见错误：

```
match &mut self.root {
    None => {}
    Some(node) => {
        if node.value < value {
            // cannot assign twice to immutable variable `node`
            node = node.left.as_mut().unwrap();
        }
    }
}
```

这里的 node 是一个可变借用，但是 node **本身不可变**，那么 node 这个代表着当前节点的引用不能指向新的结点，即无法向下移动。

解决方法：let mut current = node; 就是可变的可变借用了。

二叉搜索树 - 插入

```rust
let mut current = &mut self.root;
while let Some(node) = current {
    if value < node.value {
        current = &mut node.left;
    } else if value > node.value {
        current = &mut node.right;
    } else {
        return Err("Duplicate value");
    }
}
// Create a new node in `current`
// *current = Some(xxx);
```

二叉搜索树 - 遍历、搜索和 Left 操作

中序遍历已经给出了提示，递归编写，通过传递一个 `&mut Vec<i32>` 参数来写入结果。

查询和插入的过程是类似的，只不过因为查询不需要修改结点内容，所以所有的可变借用都改成不可变借用即可。同时用一个 `Vec<Path>` 记录下查询的路径。

`Right` 操作：舍弃根结点以及左儿子结点，仅保留右儿子结点。核心矛盾在于，如果借用了右儿子，就无法更新根结点。因此可以先用 `Option::take` 转移所有权（原来的变成了 `None`），然后再修改根结点。

2

Wordle 大作业

基础要求

- 颜色判断逻辑：给出的答案一定是不能误导玩家的，也就是说不能把正确答案排除出去
  - 位置和字母都正确的，一定是 G（绿色）
  - 剩下的字母正确但是位置不正确的，先出现的是 Y（黄色），后出现的是 R（红色），根据数量判断
  - 其余字母不正确的是 R（红色）
  - 实现：利用小作业里学到的计算出现次数方法
- 随机打乱：
  - 真随机数是不可预测的，伪随机数是从种子中计算出来的
  - 每次随机一个数，可能还会出现重复
  - 随机打乱再顺序取，保证了随机性和不重复性
  - 不同版本的 rand crate 可能表现不同，需要严格按照指导书说明
- 多关键字排序：
  - 如何实现按次数降序和字典序升序?
  - 第一种方法：自定义比较函数，当次数相等时，按字典序比较
  - 第二种方法：使用稳定排序算法，先按字典序排序，再按次数排序

基础要求

- 命令行参数匹配：
  - 使用 `clap` 解析命令行参数为结构体
  - 可选的参数可以用 `Option`
  - 指定默认值等等
- JSON
  - 类型：字符串，字典，数组，数字
  - 使用 `serde_json` 解析为结构体
  - 类似地，可能不出现的键值也用 `Option`
  - 需要注意的是，不同的库的标注是分别工作的，也就是说 `clap` 的默认值不会作用于 `serde_json` 的默认值
- 合并配置文件和命令行参数：第三方库 `config` 或者 `merge`

提高要求

- 筛选可能的答案：
  - 第一种方法：根据猜测的词和颜色，去掉不满足要求的答案
    - 绿色：去掉那些同位置上字母不对的词
    - 黄色：去掉那些同位置上字母不对的词，还要考虑同字母出现的次数和顺序
    - 红色：去掉那些同位置上字母对的词，还要考虑同字母出现的次数和顺序
    - 这个方法比较复杂，容易遗漏边界情况
  - 第二种方法：根据猜测的词和答案，计算出颜色，和已知颜色比较，去掉颜色不同的
    - 复用已有的颜色计算函数
    - 没有新的边界情况

提高要求

- 信息熵计算：
  - 信息熵：做出这个决策可以带来的信息量的期望
  - 算法思路是，对于每个词，计算后续可能出现的 3^5 种状态下（SSSSS），分别可能出现多少个词语，然后计算信息熵
  - 朴素办法：枚举猜测词，枚举 3^5 种状态，枚举答案，然后判断猜测单词与答案是否对应状态，然后计数
  - 优化：猜测词和答案唯一确定状态，因此只需要枚举猜测词，枚举答案，然后计算出状态，计数
  - 在 Release 模式下，初始情况下的猜测大概需要几秒
  - 按信息熵排序：`f64` 没有实现 `Ord`，是因为 `NaN` 不满足全序关系。因为信息熵不会出现 `NaN`，所以可以自定义比较函数来绕过这个问题。

提高要求

- 交互器设计
  - 目的：为用户提供独立求解任一 Wordle 游戏的功能
  - 示例：
    - 在一个窗口打开官方的 Wordle 游戏
    - 用户输入任一词汇，得到游戏端的反馈之后输入到求解器
    - 求解器给出推荐之后，由用户再次输入到游戏端，依此循环
  - 可以尝试设计为独立的 bin target，复用已有代码，并提供独立运行的逻辑

常见问题与解决方法

1

Q：为什么我和测例用的同一个种子，但生成的随机数不同?

A：不同版本的随机数实现不同，按照文档说明应当采用 0.8.5 版本的 `rand` crate。

2

Q：为什么我在自动测试下会卡死，但是手动输入却没有问题?

A：这种情况一般出现在非法参数的测例，此时我们的输入文件是空的。你应该在检查到参数非法或冲突之后结束程序，而不是继续等待输入。

全局变量

开发程序时，我们时常想要定义一个全局变量用来存储某个信息，但是在 Rust 中直接定义全局变量会出现报错。后续课程上会讲到这个问题，这里提前预告一下，在多线程的场景下，全局变量可能同时被多个线程访问，无法保证其同时只有一个可变借用。

解决方法是使用 `Arc<Mutex<T>>`，其中 `Mutex` 是互斥锁，提供了线程安全的内部可变性，`Arc` 是线程安全的引用计数器。例如要用 `Vec<Job>` 保存所有评测任务的话，可以用 `Arc<Mutex<Vec<Job>>>` 类型来定义一个全局变量，比如：

```
use std::sync::{Arc, Mutex};

struct Job;
pub static JOB_LIST: Arc<Mutex<Vec<Job>>> =
    Arc::new(Mutex::new(Vec::new()));
```

但是这样的代码不能通过编译。

全局变量 + LazyLock

但是，上面的代码不能通过编译，因为 `Arc::new()` 函数无法用于全局变量的初始化。因为 Rust 的 static / const 变量的初始化必须是编译期常量 (const context)，但 `Arc::new()` 函数是在运行时调用的。

因此，我们需要引入 `std::sync::LazyLock` 来实现这个事情，比如：

```rust
use std::sync::{Arc, Mutex, LazyLock};

struct Job;

static JOB_LIST: LazyLock<Arc<Mutex<Vec<Job>>>> =
    LazyLock::new(|| Arc::new(Mutex::new(Vec::new())));
```

这样就可以通过编译了。

JSON 序列化与反序列化

Wordle 大作业和将来的 OJ 大作业都出现了 JSON 的序列化和反序列化。你可以使用 `serde_json` 来实现 JSON 的序列化（Serialize）与反序列化（Deserialize），实现代码中的值与 JSON 值的双向转换：

```rust
#[derive(Serialize, Deserialize)]
struct Config {
    random: Option<bool>,
    difficult: Option<bool>,
    stats: Option<bool>,
    day: Option<usize>,
    seed: Option<u64>,
    // ...
}
```

```json
{
  "random": true,
  "difficult": false,
  "stats": true,
  "day": 5,
  "seed": 20220123,
  // ...
}
```

在 OJ 大作业中，涉及到 JSON 的地方会更多，因此这里介绍一下 `serde_json` 的进阶用法。

枚举与 JSON 的转换

在 OJ 大作业中，为了表示评测任务的状态，会出现枚举：

```
{ "state": "Queueing" }
```

此时，我们可以用枚举类型来表示 `state`：

```
#[derive(Serialize, Deserialize)]
enum State { Queueing, Running, Finished }

#[derive(Serialize, Deserialize)]
struct Job {
    // ...
    state: State,
}
```

枚举与 JSON 的转换

但有时候，并不能直接对应到 Rust 代码，因为出现了空格：

```
{ "result": "Compilation Error" }
```

此时，可以用 `#[serde(rename = "")]`：

```rust
#[derive(Serialize, Deserialize)]
enum JudgeResult {
    Waiting,
    #[serde(rename = "Compilation Error")]
    CompilationError,
}
```

枚举与 JSON 的转换

有时候，也会出现命名方式的不兼容：CamelCase vs snake_case：

```
{ "type": "standard" }
```

此时，可以用 `#[serde(rename_all = "snake_case")]` 批量重命名：

```
#[derive(Serialize, Deserialize)]
#[serde(rename_all = "snake_case")]
enum ProblemType {
    Standard,
    Strict,
    DynamicRanking
}
```

JSON 与 Rust 关键字冲突

还有一种情况，JSON 的键正好对应了 Rust 中的关键字：

```
{ "type": "standard" }
```

```
#[derive(Serialize, Deserialize)]
struct Problem {
    // expected identifier, found keyword `type`
    type: ProblemType,
}
```

此时有两种方法可以解决：

1. 把 `type` 改为 `r#type`，强制定义一个名为 `type` 的成员
2. 把 `type` 改为 `ty`，然后加上标注 `#[serde(rename = "type")]`

错误处理

Rust 提供了 `Result` 和 `Option` 来处理错误和可选值。但在使用时，需要仔细考虑错误和可选值的处理手段。

针对错误的处理方式一般可以分为如下几种：

- 直接拿值：使用 `unwrap/expect` 直接获取值，如果出现错误，则程序会 panic。
- 传播错误：使用 `?` 自动传播错误，如果出现错误，则返回错误，否则继续执行。
- 自定义处理：使用 `match` 等手段获取内部值并分别处理

错误处理——直接拿值

使用 unwrap/expect 直接获取值的方法较为简单，适用于**快速原型/测试阶段**，但要避免用于生产环境中，否则程序可能会崩溃

```rust
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

在 Wordle 大作业和接下来的 OJ 大作业的验收中，助教会测试各种功能的边界情况（如指定的配置文件不存在等）。若程序在这些情况下崩溃，则会酌情扣除一定分数。因此同学们要注意，不要滥用直接拿值。

错误处理——传播错误

使用 ? 自动传播错误，如果出现错误，则返回错误，否则继续执行。它适用于**需要继续执行的场景**，比如读取文件、网络请求等

```rust
fn read_file(path: &str) -> Result<String, std::io::Error> {
    // Return error directly if it failed
    let content = std::fs::read_to_string(path)?;
    // More operations for content...
}
```

? 运算符也可以用于链式传递，如下代码：

```rust
fn read_username_from_file() -> Result<String, std::io::Error> {
    let mut username = String::new();
    File::open("hello.txt")?.read_to_string(&mut username)?;
    // More operations for username...
}
```

错误处理——传播错误

`?` 运算符也可以用于 `Option` 类型的值，如果值为 `None`，则返回 `None`，否则继续执行。

```rust
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

当函数的返回值类型和 `?` 运算符作用的值类型不一致时，需要进行类型转换，否则会导致报错。

```rust
fn read_file(path: &str) -> Result<String, String> {
    // Convert std::io::Error to String
    let content = std::fs::read_to_string(path)
        .map_err(|e| e.to_string())?;
    // More operations for content...
}
```

错误处理——自定义处理

我们可以使用 match 等手段获取内部值并分别进行处理。

```rust
fn read_file(path: &str) -> Result<String, std::io::Error> {
    match std::fs::read_to_string(path) {
        Ok(content) => {
            // More operations for content...
        }
        Err(e) => {
            // Print error message
            eprintln!("Error reading file: {}", e);
            Err(e)
        },
    }
}
```

错误处理

除去上面三种较为典型的方法，还有众多函数可以作用于 `Result` 和 `Option` 类型：

- `if let Ok(v) / if let Err(e)`：简洁版条件匹配。
- `unwrap_or(default)`：失败时返回默认值。
- `unwrap_or_else(f)`：失败时调用一个函数来生成值。
- `unwrap_or_default()`：失败时返回 `T::default()`。
- `unwrap_or_else(|e| ...)`：失败时对错误进行处理。

及时捕获项目中的错误，并进行合适的处理，是保证程序健壮性、易于调试性的重要手段。同学们需要仔细分析好每一个功能可能存在的边界情况，并进行合适的错误处理，避免程序崩溃。

错误处理——定义错误类型

我们以 Linux 这一成熟项目为例，讲述它的错误定义和处理方式

- Linux 和用户程序交互是通过系统调用（system call）这一接口规范实现的
- Linux 在 errno.h 中定义了系统调用在各种情况下的错误码
- 在 Linux 文档中，定义了各个接口的边界情况与对应的错误码（以 read syscall 为例）
- 在处理系统调用时，Linux 会根据文档规定，**先检查各种边界情况**，如果出现错误，则返回对应的错误码，否则再进行正常处理流程，避免程序崩溃
- 用户程序在调用 System Call 之后，若出现了报错，可以根据所得到的错误码以及文档，快速判断出错的原因

3

开发辅助工具

clippy

- Rust 官方提供的静态代码分析工具
- 可以帮助发现代码中的潜在问题和改进建议，让代码更加规范、健壮

```
warning: equality checks against true are unnecessary
--> src/main.rs:3:8
|
3 |     if x == true {
|        ^^^^^^^^^^ help: try simplifying it as shown: `x`
|
= help: for further information visit https://xxx
= note: `#[warn(clippy::bool_comparison)]` on by default
```

clippy 分析

- clippy 的分析与报告都有对应的规则，详见 index.html，可以使用 “
- 在得到报告，需要仔细阅读并分析 clippy 附带的原因和修改建议
  - 不要急着质疑 clippy，大部分情况下它是为了代码的兼容性等方面考虑
  - 但少数时候 clippy 也存在假阳性：按照 clippy 的建议修改后，代码反而会出错
  - 可以通过 `#[allow(clippy::xxx)]` 来手动忽略对某些规则的检查
  - clippy 的报告可能包含错误和警告，建议不要忽视了警告（部分程序员的坏习惯）
- 最好随时发现并解决 clippy 的问题（可以利用 IDE 的自动检查和修复功能），而不是等到写完一个大项目之后再一口气解决

rustfmt

- Rust 官方提供的代码格式化工具，可以帮助你保持代码风格一致
- 每个程序员都有自己的代码风格，但是团队合作时需要保持一致
  - 本次课程统一使用默认的 rustfmt 配置，详见 Configuring Rustfmt
  - 个人开发时，可以参考上述文档，自定义本项目的 rustfmt 配置（比如大括号是否换行）
- 本次课程中，若在大作业 CI 中未能通过 clippy 和 rustfmt 测试，则会酌情扣除代码规范分

更多辅助工具

Rust 生态中存在着许多由官方或第三方维护的工具，可以帮助更加深入的分析、了解程序。但部分工具可能存在学习成本。以下为一些第三方工具，更多辅助工具详见扩展工具。

- cargo doc：官方提供的文档检查、生成工具，可以检查文档注释是否符合规范，并生成文档
- miri：Rust 中高质量的 UB（Undefined Behavior）检查结果
- cargo-udeps：检查项目中未使用的依赖
- cargo-outdated：检查项目中过时的依赖，尽可能使用新的依赖
- semver：检查项目的语义化版本控制（详见 semver 2.0）
- lockbud：检查 deadlock 等并发问题

IDE 辅助开发

使用一个好的 IDE 可以显著提高开发效率。多样且强大的插件可以很好的优化开发体验。本课程推荐如下 IDE:

- VSCode + rust-analyzer: VSCode 本体是一个文本编辑器，但配合 rust-analyzer 等插件可以提供非常强大的代码补全、类型检查、代码跳转等功能
- RustRover: JetBrains 专为 Rust 打造，集成了 rust-analyzer，HTTP Client 等生产力工具。学生可以免费使用
- Cursor/Claude Code 等集成 AI 的 IDE: 强大的 AI 支持，新时代的开发体验
- Zed: 用 Rust 编写的高性能、协作型编辑器，更快、更轻，适合多人协作

IDE 可以提供的功能

- 自动检查与修复功能
- 断点调试功能（可阅读 代码调试——docs9 了解更多细节）
- 代码补全、跳转功能等
- 丰富的插件扩展

vibe coding

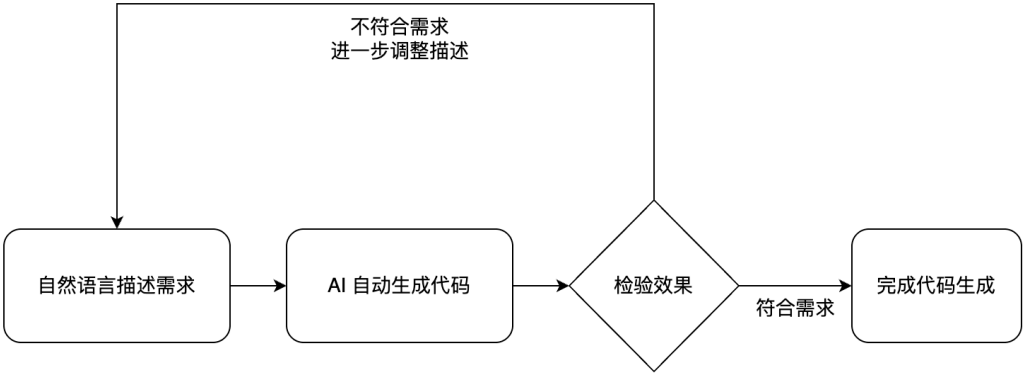
2025 年 2 月，OpenAI 创始人之一 Andrej Karpathy 提出了 vibe coding（氛围编程）的概念，主张使用人工智能（AI）根据自然语言提示生成指定代码，从而加快开发速度，让应用构建变得更加容易。

以下是 Google Cloud 针对 vibe coding 和传统编程的对比介绍[1]：

- 传统编程：专注于**实现细节**，**手动编写**语言所需的特定命令、关键字和标点符号
- vibe coding：专注于**所需的结果**，用**通俗易懂**的语言描述您的目标，AI 则负责处理实际的代码，连代码初学者都可以快速上手

[1] Google Cloud. 什么是氛围编程 (vibe coding)?

vibe coding



vibe coding

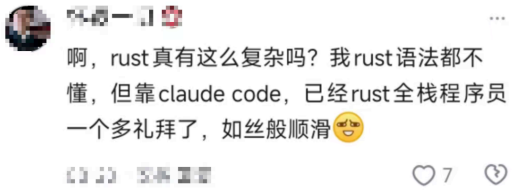
目前较常见的 vibe coding 工具：

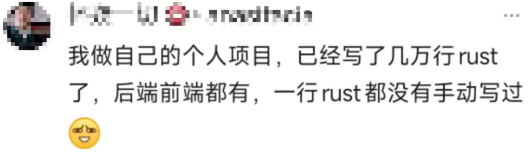
- Cursor：基于 VSCode 开发的集成式 IDE，可以复用绝大部分的 VSCode 插件，在项目上下文理解上更加自然，擅长日常开发中的增量编码、补全等需求
- Claude Code：终端型 AI 编程助手，擅长处理跨文件、复杂描述的任务，开发者需要在命令行和 AI 进行交互，并返回编辑器进行编辑
- GitHub Copilot：VSCode 原生支持的插件，提供智能补全、对话式帮助等功能，相比前两者门槛更低，但功能相对有限
- Google AI Studio：基于浏览器的集成 IDE，提供了一个便捷的环境，让用户可以快速尝试不同的模型，并使用自然语言提示来进行实验

vibe coding

也有很多程序员仍然抵触 vibe coding 的使用，认为它只是降低了编程的门槛，并没有真正提高编程能力。

编程的根本还是程序员自己的基础知识，AI 只是辅助工具，不能完全依赖。它生成的结果不一定是正确的，无论是否正确，都一定要仔细检查。





事实上现在 AI 能够直接解决的问题还停留在比较简单的水平，真正复杂的项目仍然需要程序员去分析、设计、实现。

vibe coding

本课程使用原则：

- 课程作业的主体必须由学生本人独立完成
- AI 仅可用于辅助开发，不可直接生成或抄袭作业的**核心内容**。
- 可以使用 AI 的场景包括：
  - 代码片段补全：在编写代码时，可以用 AI 辅助完成简单的代码补全，如意义相近的分支判断、循环结构等
  - 格式化文档：可被 AI 规范化的文档包括代码注释、项目报告等，AI 可以帮助格式化文档，从而使其更加规范、易读
  - 辅助调试：如将报错信息和对应源码发送给 AI，让 AI 帮助你分析问题所在，从而协助找出问题
  - 询问思路：对于某个具体问题，可以询问 AI 一些关于实现思路的建议，但是不要让 AI 直接生成完整代码

验收时我们可能会对代码内容进行提问，若同学们无法回答，则可能被认为抄袭，从而影响课程成绩。