

# tmuxlab 报告

Autograder 结果：



## Part A: 思路简述

*TmuxLab* 的总体实现思路是：先将每个 *Pane* 视为一个独立的“伪终端 + *shell* 子进程”，再在其外部构建一层 *client-server* 架构。

采用的 *client-server* 架构：

- *server* 常驻后台，负责管理 *Unix domain socket*、*Pane*、*PTY*、*shell* 子进程以及附加功能。
- *client* 负责把本地终端变成一个前端，读取键盘输入、发送消息、接收数据、做 *raw mode* 恢复。
- *Pane* 内程序的输入输出不直接连到用户终端，而是走 *PTY*。
- *client* 断开不影响 *server*，*server* 退出才意味着 *mini-tmux* 生命周期结束，这样 *detach* / *reattach* 就自然成立了。

一个运行单元最核心的结构是：

- 一个 *PTY master*
- 一个 *shell* 子进程
- 一份屏幕状态
- 一组附属资源（log、pipeout 等）

因此实现里主要围绕四类对象组织状态：

- ClientSession：记录一个 client socket、只读标记、终端大小以及当前附着到哪个 session\_id。
- Pane：记录 PTY master、shell 子进程 pid、当前大小、屏幕缓冲，以及 log / pipeout 这些附加资源。
- Session：维护本 session 的 pane 顺序、当前焦点 pane，以及最近一次使用的窗口大小。
- ServerState：统一持有 sessions、panes、pane 到 session 的映射关系、id 分配器和当前连接的 clients。

mini-tmux 整体设计思路为：

```
Start mini-tmux
|
|-- If server does not exist: fork + setsid + exec self --server
|
|-- Client connects to the socket
|   |
|   |-- Attach: report terminal size, read-only status, and optional target
|   |         session
|   |-- Input: send user input to the current pane
|   |-- Command: request operations such as new, focus, kill, new-session,
|   |           switch-session, log, pipeout, capture
|   |-- Resize: notify the server when the window size changes
|   |-- Detach: disconnect from the server
```

## Part B: 具体实现分析

### Pane 的创建：构造具有完整终端语义的运行环境

整个 Lab 中最关键的函数是 `spawn_pane_process(rows, cols)`，作用是将一个 *shell* 放入一对 PTY 中，并使该 *shell* 获得自己的控制终端。

这部分的系统调用顺序如下：

```
posix_openpt()
grantpt()
unlockpt()
ptsname()
fork()

child:
    setsid()
    open(slave)
    ioctl(TIOCSCTTY)
    ioctl(TIOCSWINSZ)
    dup2(slave, 0/1/2)
    exec $SHELL

parent:
    keep master_fd and pid
```

其中最关键的两步是 `setsid()` 和 `ioctl(slave_fd, TIOCSCTTY, 0)`。先调用 `setsid()`，使子进程成为新的 *session leader*。

实现过程中也考虑过直接使用 *pipe*，但最终没有采纳这个方案。原因在于 *pipe* 只能传输字节流，无法提供终端语义。*shell* 一旦发现自己连接的不是 *tty*，就会退化为批处理模式，交互程序也无法正常工作。

## 启动流程：单二进制切换 server / client 角色

main.cpp 的逻辑并不复杂，处理方式如下：

- 如果参数中带有 `--server`，则当前进程直接执行 `run_server(socket_path)`。
- 否则，先检查 `socket` 是否已经有 `server` 在监听。
  - 如果没有，则 `fork()` 一个子进程，并在子进程中调用 `setsid()` 后 `exec self --server`；当前进程继续作为 *client*，等待 *server* 启动并建立连接。
  - 如果已经存在 *server*:
    - 普通启动：创建一个新的 *session*
    - `attach/attach -r/attach -s <session_id>`：连接到已有 *session*
    - `list-sessions`：列出当前所有 *session*

## client 协议与输入状态机

server 和 client 之间没有采用纯文本协议，而是使用一个简单的消息头：

```
struct MsgHeader {
    uint32_t type;
    uint32_t size;
};
```

client 端输入状态被划分为四种模式：

```
enum class InputMode {
    Normal,
    Prefix,
    Command,
    InlineCommand,
};
```

四种模式分别对应：

- Normal：普通输入，直接发给当前 Pane
- Prefix：刚收到 `Ctrl+B`
- Command：进入 `Ctrl+B`：命令模式
- InlineCommand：为了适配测试脚本，允许直接把行首的 `:cmd` 当作命令，而不是写进 `shell`

前缀键支持的动作如下：

- `Ctrl+B d`：detach
- `Ctrl+B c / % / "`：新建 Pane
- `Ctrl+B n / o`：切下一个 Pane
- `Ctrl+B p`：切上一个 Pane

- Ctrl+B :: 进入命令模式

这一部分让 *client* 保持轻量，只负责解释输入，而不承担持久化状态管理。真正的 *Pane* 创建、焦点切换、日志开启和输出管道建立都由 *server* 决定。这样即使 *client* 异常退出，也只表现为一次前端断开，不会破坏系统整体状态。

之所以需要单独加一个 `InlineCommand`，是因为测试脚本并不是先按 Ctrl+B 再输入命令，而是会直接把 `:cmd\n` 这类内容送到 *client* 的标准输入。如果没有这个模式，开头的冒号就会被当成普通字符写进当前 shell，测试也就无法稳定驱动命令路径。

## raw mode、SIGWINCH 与终端前端语义

要使 *client* 具备 *tmux* 前端的行为，就必须先将其接管的本地终端切换为 raw mode。否则许多按键不会交由 *client* 处理，例如：

- Ctrl+C 会先把 *client* 自己结束
- Ctrl+Z 会把 *client* 挂起
- Ctrl+B 不会被当作前缀键处理
- 方向键、功能键会因为本地行规程而表现异常

通过 `RawModeGuard` 采用 *RAII* 的方式管理终端模式：

- 构造时 `tcgetattr()` 备份原始 `termios`
- `cfmakeraw()` 改成本地 raw mode
- 析构时自动恢复

这种方式比手工维护“进入时设置、退出时还原”的流程更安全。即使 *client* 中途抛出异常，终端也能在大多数正常退出路径上恢复原始状态。

窗口大小变化的处理没有直接放在 *signal handler* 中完成，而是采用了以下流程：

```
Client receives SIGWINCH
-> only sets g_resize_requested to 1
-> main loop reads current window size (winsize)
-> sends a Resize message to the server
-> server recomputes layout
-> for each pane, calls TIOCSWINSZ
-> sends SIGWINCH to the corresponding foreground process group
```

## server 事件循环

*server* 的主循环统一用 `poll()` 驱动，监听三类 *fd*：

- 监听 *socket*：接受新的 *client* 连接
- *client socket*：接收 `attach / input / command / resize / detach`
- *Pane* 的 *PTY master*：接收 *shell* 或 *TUI* 程序输出

它的工作方式可以概括为：

```

while true:
    reap_children()
    poll(listen_fd, client_fds, pane_master_fds)

    if some pane master is readable:
        read output
        feed ScreenBuffer
        broadcast redraw or partial update when needed

    if some client fd is readable:
        receive message
        handle attach / resize / input / command / detach by type

    if listen_fd is readable:
        accept a new client

```

## 多 Pane 布局与 ScreenBuffer

若只支持单 *Pane*，*server* 基本可以作为一个“字节转发器”：从 *PTY* 读取数据后原样转发给 *client* 即可。但在多 *Pane* 场景下，不同 *Pane* 的输出都可能包含 ANSI 光标控制序列。若直接将多个 *Pane* 的原始输出拼接后发送给 *client* 则它们会在同一个真实终端上相互覆盖，导致显示错乱。

因此，为每个 *Pane* 单独维护了一个 *ScreenBuffer*，负责将该 *Pane* 的输出解析为一块逻辑屏幕，再由 *server* 按照布局把这些逻辑屏幕合成为整张 *frame*。

*ScreenBuffer* 支持的解析范围是：

- 普通字符
- `\n`
- `\r`
- `\b`
- `\t`
- CSI A/B/C/D
- CSI H/f
- CSI J
- CSI K

而像 OSC、DCS、PM、APC 这类较复杂序列则选择忽略。

这里是有意识的取舍，不是单纯漏做。*workload* 和日常使用主要依赖的是普通字符、换行以及常见 CSI 控制序列；如果继续把 OSC / DCS 这类序列也完整展开，解析器会明显变复杂，但对本实验通过率和主要语义帮助并不大。

布局方面采用纵向等分：

- 只有一个 *Pane* 时，不显示 header。
- 有多个 *Pane* 时，每个 *Pane* 顶部占一行 header。
- 剩余行数均分给各 *Pane*。

- 如果有余数，就从上往下分给前几个 *Pane*。

伪代码如下：

```
pane_count = panes.size()
show_headers = (pane_count > 1)
header_rows = pane_count if show_headers else 0
remaining_rows = total_rows - header_rows
base_rows = remaining_rows / pane_count
extra_rows = remaining_rows % pane_count

for i in 0..pane_count-1:
    rows_i = base_rows
    if i < extra_rows:
        rows_i += 1
```

*header* 的内容设计如：

```
[*] pane 0-----
[ ] pane 1-----
```

当前焦点 Pane 标记为 [\*]，其他 Pane 标记为 [ ]。

## 命令系统与 Bonus 部分的实现

server 侧命令解析支持的内容包括：

- new
- new-session
- focus <pane\_id>
- focus-next
- focus-prev
- kill <pane\_id>
- kill-session
- switch-session <session\_id>/attach-session <session\_id>
- log <pane\_id> <file>
- log-stop <pane\_id>
- pipeout <pane\_id> <cmd>
- pipeout-stop <pane\_id>
- capture <pane\_id> <file>

其中，对后三个 *bonus* 功能的实现如下。

```
:log
```

server 为指定 Pane 打开一个追加写文件，之后 Pane 每次产生输出时，都同步将原始输出写入日志文件。若目标文件出现异常，则在出错时关闭旧 fd 并停止记录。

```
:pipeout
```

这个功能的难点在于生命周期管理。其实现流程如下：

```
pipe()
fork()

child:
    dup2(pipe_read, stdin)
    execl("/bin/sh", "/bin/sh", "-c", cmd)

parent:
    retain pipe_write_fd
    write pane output to it later
```

如果外部命令提前退出，那么 server 在写 pipe\_write\_fd 时会遇到 EPIPE，此时需要自动清理；如果用户主动执行 pipeout-stop，则关闭写端并等待子进程退出。

```
:capture
```

capture 并不是直接抓取真实终端上的原始输出，而是调用 pane.screen.snapshot\_text() 导出逻辑屏幕和部分历史文本。这样导出的内容是纯文本，不带 ANSI 转义序列，也不会受到当前 redraw 状态影响。

## 多 Client、只读 Attach 与广播

这一部分真正要解决的不是“能不能同时连多个 client”，而是多个 client 连接到同一个 session 之后，显示语义和交互边界如何保持一致。我的处理方式是把广播范围严格限制在 session 内：同一个 session 上的 attached client 共享输出和焦点，不同 session 之间则完全隔离。attach -r 连接也仍然参与显示，但 server 会拒绝它发来的输入和命令，因此它只能看，不能修改当前 session 的状态。

另一个更麻烦的问题是窗口大小和广播鲁棒性。不同 client 的终端大小可能不同，如果直接按某一个较大的终端尺寸布局，那么较小终端上的 client 一定会显示错乱；因此 Pane 的 winsize 需要取“同一 session 内全部 attached client”的最小行列数。与此同时，广播发送不能假设每个 client 都会及时读数据。实际补测试时可以发现，只要某个 client 长时间不读，阻塞写就可能把整个广播路径拖住，连其他正常 client 都一起受影响。所以最后把广播路径改成了非阻塞发送，无法发出或连接已经损坏的 client 会直接被移除。

## 子进程回收、SIGPIPE 与 server 生命周期

这里主要处理了三个问题：SIGPIPE 直接忽略，避免写已关闭 socket 时 server 进程终止；用 waitpid(-1, &status, WNOHANG) 循环回收子进程，既回收 Pane 的 shell，也回收 pipeout 派生出来的子进程；当某个 session 的最后一个 Pane 退出时，server 会关闭该 session 上的 client 并销毁该 session，只有当整个 server 中最后一个 session 也消失时，才向所有 client 广播 Exit、关闭连接并退出。

## Part C: 关键难点解决

### 控制字符转发与前台进程组的信号语义

一种直接的做法是：client 识别到 Ctrl+C 后，由 server 直接向目标 Pane 的子进程发送 SIGINT。但只要 shell 中运行的并非单一的前台进程，这个做法就会失效。

原因在于，真正应该接收信号的并不是某一个单独的 pid，而是该 Pane 当前控制终端上的前台进程组。例如 shell 中运行的是一条管道命令，或者 shell 又拉起了前后台作业时，简单地 `kill(pane.pid, SIGINT)` 很可能只会将信号投递给 shell 进程本身，而不能正确模拟真实终端行为。最直接能观察到的异常现象是：按下 Ctrl+C 后，看上去像是当前命令被“打断”了，但前台管道里的实际工作进程并没有按终端语义一起结束，这说明问题并不在“有没有发出信号”，而在“信号到底发给了谁”。

#### 最终处理方式：

不在 server 中硬编码信号投递，而是将控制字符原样写入该 Pane 的 PTY master，由内核的 TTY 机制将 SIGINT / SIGTSTP 发送给正确的前台进程组。只有这样，语义才是准确的。

### 单 Pane 直通与多 Pane 逻辑屏幕的分流设计

从实现角度看，完全可以让所有模式都统一走 `ScreenBuffer + redraw`。但我实际把单 Pane 也统一纳入该路径后，最直接的现象就是：

- 输入回显会更“钝”，尤其是持续输出时更容易看到闪屏
- PTY 已输出正确结果，仍需经过解析、合成整帧再转发，处理路径显著延长

#### 最终处理方式：

分成两条路径：

- 单 Pane：直接广播原始 PTY 输出
- 多 Pane：先写入各自 `ScreenBuffer`，再按布局局部刷新或整屏重绘

这一分流既改善了单 Pane 场景下的交互体验，也使多 Pane 场景下的显示语义更加稳定。

### Resize 事件在 client-server-PTY 链路中的传播

Resize 这一部分，表面上看它只是“窗口大小变了，通知一下 server”。但真正实现后发现，它影响的是一整条链路：client 先收到 SIGWINCH，但 signal handler 里不能直接做复杂逻辑，所以这里只记录一个待处理标记；随后主循环重新读取本地 winsize，把新的 rows/cols 发给 server；server 收到后不仅要重算整个 session 的布局，还要把新的大小写回各个 Pane 的 PTY，并继续把 SIGWINCH 送给对应的前台进程组。

#### 最终处理方式：

把 resize 当成一条完整链路来处理，而不是某个局部补丁。如果只修改 server 里记录的 rows/cols 而不发送 SIGWINCH，很多 TUI 程序根本不会主动刷新；如果只给当前 Pane 改大小而不重算全部 Pane 布局，那么多 Pane 模式下 `stty size` 之类的观察结果就会出错。

## Part D: 实验反馈

### 系统理解

这个 *Lab* 给我的实际感受是：主流程写出来不算太难（当然这是因为活都是 AI 干），真正花时间的的是修正各类边界情形引发的 bug。PTY、session、控制终端、前台进程组、socket、poll、raw mode、winsize、signal、fd 生命周期是互相咬合的，某一环节的失误都会导致程序的崩溃。

我的一点体会是，Pane 的本质并不是起一个 shell 再把输入输出接上，而是建立一个带控制终端的执行环境。如果只是为了把字节流接起来，那么 pipe 也能做；但一旦要让 shell、vim、top、stty size、Ctrl+C、Ctrl+Z 这些行为都保持终端语义，就必须经由 PTY，并正确完成 setsid()、TIOCSCTTY、TIOCSWINSZ 等步骤。这个点在真正写代码之前我其实理解得不够具体，做完之后才明白，很多后来出现的问题都不是零散 bug，而是终端环境根本不完善的连带表现。

另一个点是信号隔离。一种直觉上的做法是：client 识别到 Ctrl+C 后，由 server 直接对 pane.pid 发送 SIGINT。但这一方案在 shell 运行前台子进程、管道命令或涉及 job control 的场景下会失效——真正应当接收信号的是当前控制终端的前台进程组，而不是某个固定 pid。两者的区别在于：前台进程组是由内核根据控制终端动态维护的，随着 shell 的 job control 状态实时变化；而 pane.pid 只是 shell 本身，并不能代表它当前派生的整个进程树。正确的做法是将控制字符原样写入 PTY master，由内核 TTY 层负责将信号投递给正确的前台进程组。

### AI 使用

对于 AI 使用情况，我怀着复杂的心情写下这段话。我必须承认，这次 *Lab* 的绝大部分是我使用 AI 完成的，使用的主要模型是 *Claude Opus 4.6* 和 *GPT 5.4*，其中 *Claude* 帮助我完成了绝大部分核心代码的实现，而 *Codex* 帮我完成了一些 *basic* 部分的实现和并在报告撰写上帮助了我。

在刚拿到这个 *Lab* 的要求的时候，我的第一反应是“这么难！这真的能解决吗？”，然后我阅读了第一版的 handout.md，然后上网查阅了一些相关的概念，并询问了 AI。但事实上，那天晚上我花了很长时间理解概念，但似乎并没有对我的 *Lab* 完成起到很好的效果，面对任务要求我依旧一头雾水（当然这个大概率是因为我比较笨）。这种负反馈让我感受到一些挫败感，在与几个朋友交流过后，我发现他们基本上也有这种感受。最后我采取的方法是：让 *Claude* 给我生成一个非常具体的完成思路，然后让它逐步完成并测试。

这个 *Lab* 中，代码和 code review 部分基本都是 AI 完成的，我只进行了简单的审核和完成步骤的控制。我没有完全搞明白的部分主要有三部分。第一类是 PTY、setsid()、TIOCSCTTY、控制终端和前台进程组之间到底是怎样连起来工作的；第二类是真正的 session 语义，尤其是“新建 session”和“新建 pane”在实现上不是一回事；第三类是多 client 下的鲁棒性问题，表面上只是把输出广播给多个 socket，实际上慢 client、异常连接、readonly、detach/reattach 都会显著放大边界条件的影响。

真正让我卡住的地方也基本都和这几件事有关。最明显的一次是我一开始以为剩下的多 Client 和 信号隔离 只是两个零散 bug，后来顺着 workload、协议和实现往下看（当然主要是 AI 在分析），才发现根因不是某个分支条件写错了，而是当时的实现里其实根本没有真正的 session 抽象，NewSession 只是“再加一个 pane”。这会同时破坏 session 独立性和信号隔离，所以表面上像是两个问题，实际上指向的是同一个设计缺口。另一次卡住是在多 client 上：公开用例已经通过了，但隐藏测试还是不过，我一开始也不知道问题在哪里。后来继续往下查，才意识到 server 原来走的是阻塞写，只要有一个 attach 的 client

长时间不读输出，整个广播路径就可能被它拖住，其他 client 和 pane 一起受影响。这个问题如果只是“面向测试点编程”，其实不太容易想到。当然，这部分的 debug 也是 AI 做的，我给 AI 的提示词是“请不要面向测试点编程，也就是说，不要以通过测试点为目标，这样容易产生过拟合。请以一个正式的项目为目标吧！”。

这次 Lab 其实并没有让我对 PTY 等等概念非常精通（这还是因为绝大部分工作都是 AI 做的），真正让我有比较大收获的是：

- 多 agent 协作。比如这次 Lab 中，一开始我只使用 Codex 写代码，最后发现效果并不好：AI 容易困在自己的幻想里原地转圈。于是我引入 Claude 写代码，让 Codex 做 code review，并且把 Claude 和 Codex 的建议互相交换，不断打磨，最后取得了很好的效果。
- prompt。由于窗口内可容纳的上下文长度是有限的，经常需要新建 session，所以我把 AI 的产生的上下文保存在 markdown 文件中，并且尽可能详细地写我的 prompt，也取得了不错的效果。
- 还是 prompt。我学会了把 prompt 写详细。以前总是像挤牙膏一样一段一段写 prompt，分多次交给 AI。但这次发现一次把所有要求尽可能写清楚，效果会更好。
- 以及一个发现。AI 倾向于偷懒，很容易通过在代码上打补丁的方式来追求通过某个测试点。这样很容易造成鲁棒性不足，而且之后还很难找到 bug。具体体现在检查一开始没过的部分的时候，AI 一口咬定已经没问题了，但在我仔细检查与刨根问底之下才发现原来还是面向测试点编程。以后这一点还需要特别注意。