

从内核开发流程到邮件协作

用 CRIEW 更轻松上手 Linux kernel 邮件 workflow

订阅 → 同步 → 阅读 → 应用 patch → 回信

这份分享的目标

先讲 Linux kernel 的邮件式开发流程，再讲 CRIEW 如何把这条流程变成新人也能持续使用的终端 workflow

一句话定位

CRIEW 不是另一个 GUI 邮件客户端，而是面向 kernel patch 邮件协作的

这场交流会会怎么进行

1 先讲传统流程

先用 Linux kernel 的邮件式开发流程建立共同上下文，看 mailing list、patch series 和 reroll 为什么会这样运作。

2 再拆 patch 与 thread

接着看一封 PATCH 文件长什么样，以及 review reply、v2 / v3 在邮件 thread 里怎么串起来。

3 然后看 CRIEW demo

再切到 CRIEW，看看它如何把 thread 阅读、patch apply 和 reply 组织进同一个 TUI 工作流。

4 最后讲现状与参与

最后会讲项目目前到了什么阶段、适合谁现在试，以及会后怎么安装、反馈和参与贡献。

节奏：这份内容按 30 到 60 分钟准备；中间适合穿插 live demo，最后留出讨论和问答。

为什么内核开发仍然以邮件为中心

Linux kernel 的协作单位
不是 PR,
而是 patch series + thread

一封 patch 邮件只是开始；真正的协作发生在整个线程里。

公开讨论

改动发到邮件列表，review 对所有人可见。

线程上下文

后续意见通过 reply 累积在同一 thread。

多轮修订

v2 / v3 / RESEND 是常态，不是例外。

合入路径

maintainer 从邮件线程收集 trailer，再带入自己的 tree。

网页上的 mailing list 是什么样子

```
linux-arch.vger.kernel.org archive mirror
search help / color / mirror / Atom feed

[PATCH v3 0/7] futex: Use runtime constants for futex_hash computation
2026-04-10 9:37 UTC (11+ messages)
~ [PATCH v3 1/7] x86/runtime-const: Introduce runtime_const_mask_32()
~ [PATCH v3 2/7] arm64/runtime-const: Use aarch64_insn_patch_text_nosync() for patching
~ [PATCH v3 3/7] arm64/runtime-const: Introduce runtime_const_mask_32()
~ [PATCH v3 4/7] riscv/runtime-const: "
~ [PATCH v3 5/7] s390/runtime-const: "
~ [PATCH v3 6/7] asm-generic/runtime-const: Add dummy runtime_const_mask_32()
~ [PATCH v3 7/7] futex: Use runtime constants for __futex_hash() hot path

Avoid reading /sys/kernel/mm/transparent_hugepage/?
2026-04-10 8:37 UTC (3+ messages)

[PATCH v14 0/4] arm64: Use generic TIF bits for common thread flags
2026-04-10 3:39 UTC (7+ messages)
~ [PATCH v14 2/4] asm-generic: Move TIF_SINGLESTEP to generic TIF bits
~ [PATCH v14 3/4] arm64: Use generic TIF bits for common thread flags
```

示例入口:

<https://subspace.kernel.org/vger.kernel.org.html>

<https://lore.kernel.org/linux-arch/>

一条 patch 从想法到合入，大致会经历什么

阶段	关键动作	典型关注点
1 选基线	找对 subsystem 与 maintainer tree	你改的东西归谁维护
2 拆 commits	一补丁一件事，写清 commit message	patch 粒度能否被 review
3 形成 series	cover letter + [PATCH vN M/N]	线程是否可持续迭代
4 本地自检	checkpatch / 编译 / 功能测试	不把低级问题扔给 reviewer
5 选收件人	get_maintainer.pl 或 b4 prep -auto-to-cc	To/Cc 列表是否对
6 发邮件	git send-email 或 b4 send	保住邮件线程关系
7 Reroll	读 review，回复并修成 v2 / v3	changelog 和回复质量

核心点：关键不是一次 push，而是围绕 thread 的持续讨论与修订。

传统 Linux patch 流程，命令层面大概长这样

```
git clone
https://git.kernel.org/.../linux.git
cd linux
git checkout -b fix-arch-bug
$EDITOR arch/...
make -j$(nproc)
git add arch/...
git commit -s
git format-patch -1
```

第一段链路还只是本地开发：改代码、编译、自测、整理出 patch 文件。

```
./scripts/checkpatch.pl 0001-*.patch
./scripts/get_maintainer.pl -f
arch/...
git send-email 0001-*.patch
# 去 lore 里继续看 thread / 收 review
git commit --amend -s
git format-patch -v2 -1
git send-email 0001-v2-*.patch
```

难点：单条命令都不复杂，但收件人、thread、版本号 and 上下文关系都要自己维护。

一个 PATCH 文件实际长什么样

邮件头 + 提交说明

```
From abcdef1234567890 Mon Sep 17 00:00:00 2001
From: Alice Dev <a@example.com>
Subject: [PATCH v2 1/1] arch: fix boot order

Fix the fallback order in early boot path.

Signed-off-by: Alice Dev <a@example.com>
---
arch/foo/setup.c | 4 +++-
```

diff 正文

```
diff --git a/arch/foo/setup.c b/arch/foo/setup.c
index 123456789abc..987654321def 100644
--- a/arch/foo/setup.c
+++ b/arch/foo/setup.c
@@ -42,7 +42,9 @@ static int foo_init(void)
-     return old_path();
+     if (needs_fix())
+         return new_path();
+     return old_path();
```

观察点：这不是只有 diff 的补丁片段，而是一封可被邮件列表分发、review、回复、reroll 的纯文本邮件。

thread 里，PATCH / 回复 / v2 是怎么串起来的

```
[PATCH v1 0/2] cover letter
|- [PATCH v1 1/2] foo: add prepare step
|  `-- Re: please split error path
`-- [PATCH v1 2/2] foo: wire caller
    `-- Re: can this fail earlier?
```

```
[PATCH v2 0/2] cover letter
|- [PATCH v2 1/2] foo: add prepare step
|  `-- Re: looks good to me
`-- [PATCH v2 2/2] foo: wire caller
    `-- Re: applied, thanks
```

读 thread 时要注意

- ▶ 每一封 patch、每一条 review reply，都是独立邮件。
- ▶ reviewer 通常挂在具体 patch 下回复，而不是只回 cover letter。
- ▶ v2 / v3 是新一轮 series，不会覆盖旧邮件。
- ▶ 真正要看的，是最新版本和它下面的回复上下文。

为什么新人会晕： 同一个主题会在邮件列表里展开成一棵树，而不是一个线性的 PR 页面。

新人真正卡住的，往往不是写代码

技术动作

- ▶ 正确拆 patch，保持每封邮件主题清晰
- ▶ cover letter、changelog、Signed-off-by
- ▶ 回复时保住 In-Reply-To 与 References
- ▶ 在线程里判断哪一版才是当前上下文

工具碎片

- ▶ lore、收件箱、本地 tree 来回切换
- ▶ b4、git send-email、邮件客户端各管一段
- ▶ 手工维护抄送名单和 Message-ID 容易出错
- ▶ review 结论、patch 状态、代码现场彼此脱节

结论：真正的门槛常常是上下文分裂，而不是命令本身。

Git 和 b4 各自负责什么

Git 是什么

- ▶ 分布式版本控制系统，管理 commit、branch、diff 和历史。
- ▶ 在内核流程里负责本地开发与生成 patch；常见命令有 commit、format-patch、am。

b4 是什么

- ▶ 面向 patch 邮件工作流的辅助工具，围绕 lore、thread 和 trailer 工作。
- ▶ 在内核流程里负责取回 series、整理收件人、导出或应用 patch；常见命令有 prep、am、send。

比较点	Git	b4
核心对象	commit、diff、patch 内容	thread、series 元数据、trailer
分工边界	管代码版本与补丁内容	管邮件上下文与投递 / apply 自动化
相互关系	整个流程的基础设施	建立在 Git 之上的邮件工具

一句话：Git 管代码版本，b4 管邮件上下文；CRIEW 把这条链路收进 TUI。

CRIEW 在这个流程里的定位

Subscribe

订阅 lore 列表
或 My Inbox

Sync

把邮件拉到本地
缓存与索引

Review

thread 树、正文、patch
在 TUI 内阅读

Apply

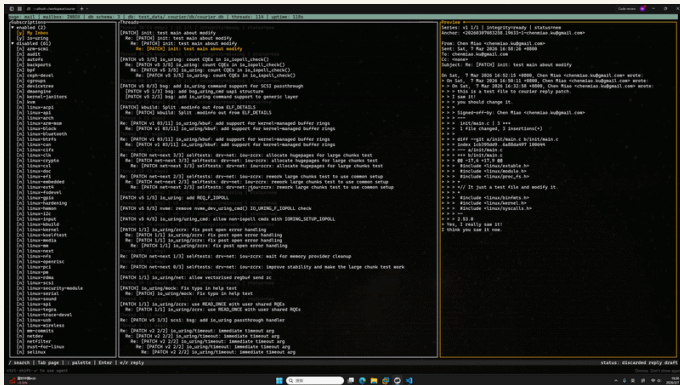
调用 b4 导出 / 应用到 kernel tree

Reply

生成标准 reply 头
并进入发送预览

项目定位：它不是替代 Git，而是把邮件协作链路收束进一个终端工作台。这和仓库文档里的定位一致：terminal-first、local-first、b4-centered。

一个真实 thread 在 CRIEW 里怎么读



读图方式：左侧是 mailbox / 订阅，中间是 thread 树，右侧是当前邮件预览；这让 patch、reply 和上下文重新回到同一屏里。

它替你省掉哪些机械劳动

统一入口

`criew doctor / sync / tui` 把入口压缩成几条命令。

双源同步

既能跟 `lore` 订阅，也能接 `IMAP INBOX`。

线程建模

按 `Message-ID / References` 重建讨论上下文。

Patch 语义

识别 `[PATCH vN M/N]`，自动聚合成 `series`。

b4 集成

`b4 am / shazam` 的 `apply` 结果可以被追踪。

回信面板

预填 `Re:`、`To/Cc`、引用正文，并先看发送预览。

把机械动作变成默认动作，才是降低上手门槛的关键。

对比传统做法，CRIEW 改善了什么

传统链路

- ▶ 浏览器看 lore，终端里单独 apply
- ▶ 邮件客户端与代码现场天然脱节
- ▶ 回信头与抄送名单靠手工维护
- ▶ thread 状态、patch 状态分散在多个工具里

CRIEW 链路

- ▶ 订阅、thread、正文、tree 视图在同一 TUI
- ▶ patch apply 与邮件上下文尽量同屏
- ▶ reply 头与引用结构自动构造
- ▶ 更适合刚上手邮件协作的新同学

总结：它改善的是流程摩擦与上下文切换成本，而不是试图改写 kernel 社区规则。

CRIEW 现在到了什么阶段

已经具备

- ▶ 当前公开版本是 **v0.0.2**。
- ▶ **doctor / sync / tui** 这条主链路已经可以演示和上手。
- ▶ 已经把 **lore / IMAP 同步、thread 阅读、patch apply、reply** 预览串成一套终端流程。
- ▶ **patch** 入口以 **b4** 为中心，回信路径可走 **git send-email**。

当前边界

- ▶ 它面向 **terminal-first** 用户，不会变成网页式 PR 平台。
- ▶ 真实 **IMAP**、**apply**、发送链路仍需要本地环境和账号配置。
- ▶ 它替你省工具摩擦，但不替你做 **maintainer** 选择、测试和工程判断。
- ▶ 现在最适合愿意给真实 **workflow** 反馈的早期用户和贡献者。

交流会价值： 现在正是适合社区交流的阶段，核心链路已经能看见，但产品形态还在和真实使用场景一起打磨。

CRIEW 让流程更顺，但不替代内核社区基本功

工具负责把链路接起来；工程判断、测试质量和沟通质量仍然来自开发者本人。

仍然要做的事

选对 maintainer 和 mailing list；跑构建、测试和 scripts/checkpatch.pl。

写作要求

写清 commit message、cover letter 和 changelog。

交流要求

正确回复 review，并按节奏做 v2 / v3 reroll。

CRIEW 的价值

让新人把注意力更早放在 patch 质量与交流内容上，而不是被工具碎片卡住。

会后自己上手的最短路径

```
cargo install criew  
criew doctor  
criew sync --mailbox  
io-uring  
criew tui
```

第一轮建议只读公开列表，不必一上来就把真实发信链路全部接上。

1. 先同步一个公开列表，建立对 thread 的直觉。
2. 在 TUI 里顺着 series 阅读 cover letter、patch、回复。
3. 需要验证时，再把 series apply 到本地 kernel tree。
4. 熟悉 reply 结构之后，再尝试写 review。
5. 最后才接入 IMAP 与真实发送，把风险前移到阅读阶段。

建议顺序：先练阅读和 apply，再练发信。

会后怎么试，怎么参与

最快的会后动作

- ▶ 从仓库 README 和 wiki 开始，把 `doctor / sync / tui` 这条链路跑通。
- ▶ 先拿一个公开列表做阅读和 `thread` 导航练习。
- ▶ 如果现场演示触发了兴趣，优先复现你自己关心的真实 `thread`。

最有价值的反馈

- ▶ 带具体 `mailing list thread`、具体卡点和具体命令环境来反馈。
- ▶ 对 `sync`、`apply`、`reply` 三条链路的失败样本尤其有价值。
- ▶ 除了代码，也欢迎文档、交互、术语和新人上手路径方面的建议。

入口：CRIEW GitHub repository CRIEW wiki

参考资料

- ▶ CRIEW README
- ▶ Linux kernel process overview
- ▶ Submitting patches
- ▶ Patch submission checklist
- ▶ b4 contributor overview
- ▶ b4 am / shazam
- ▶ git send-email

这份 Beamer 稿现在更适合 30 到 60 分钟社区交流会；建议现场穿插 live demo，并预留问答时间。