从零开始为 RISC-V 构建一个 Linux 系统

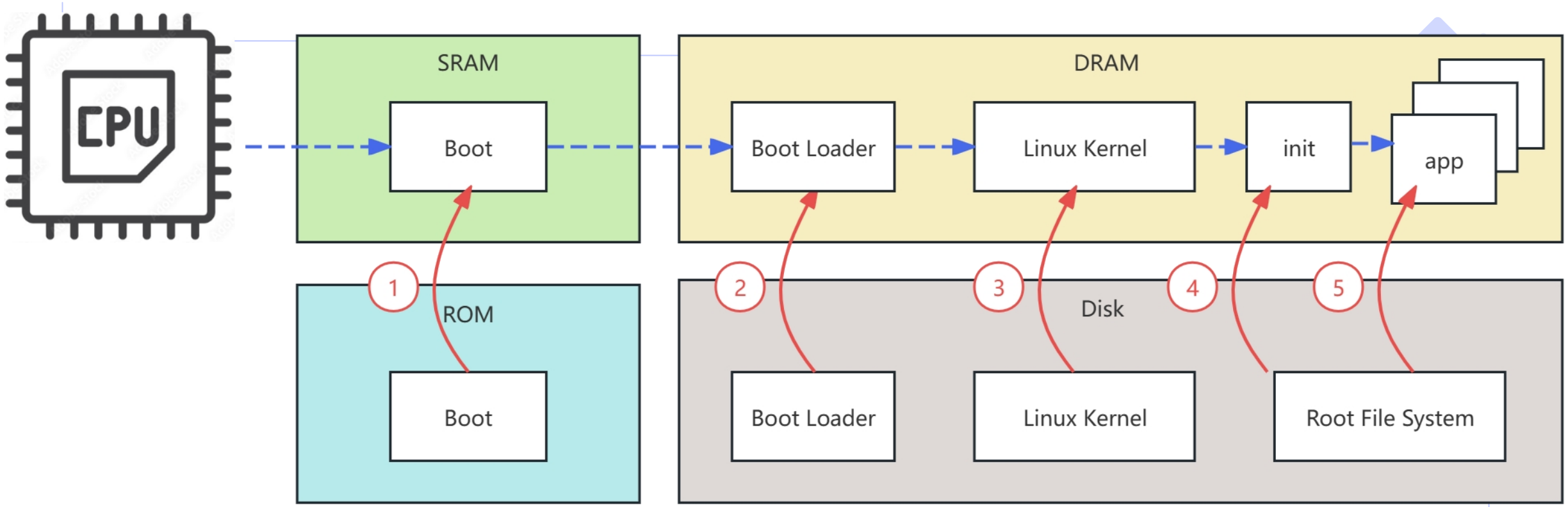
第 6 章 制作 OpenSBI

汪辰

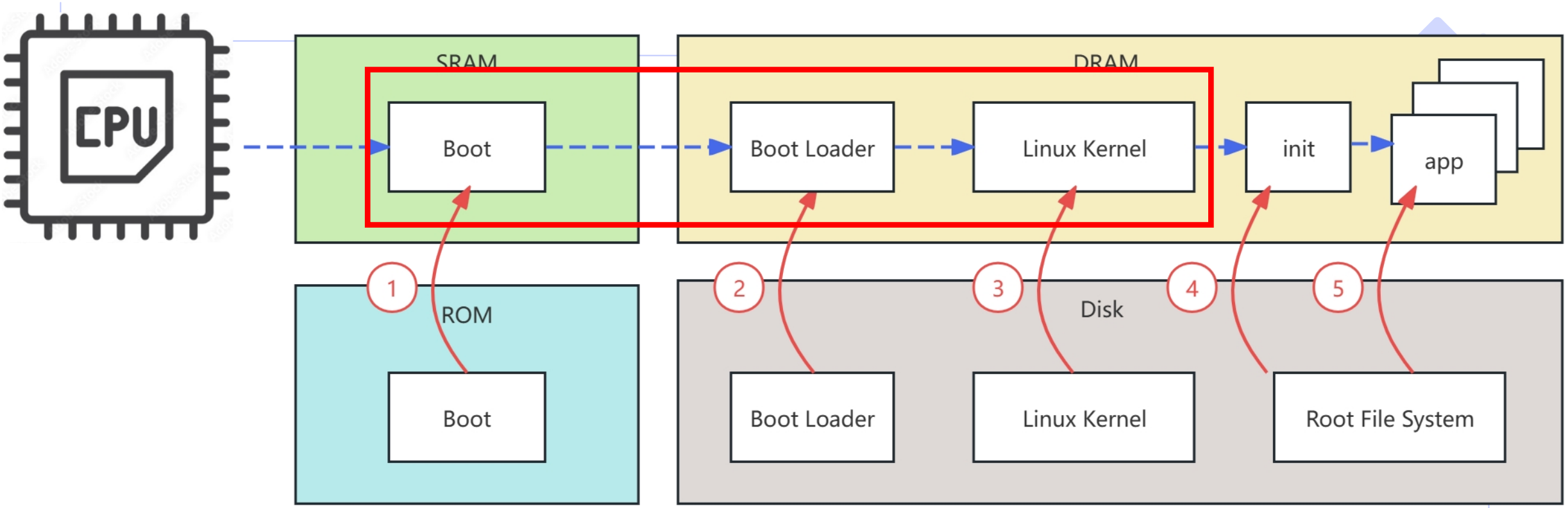
目录

01 Boot 过程和 OpenSBI

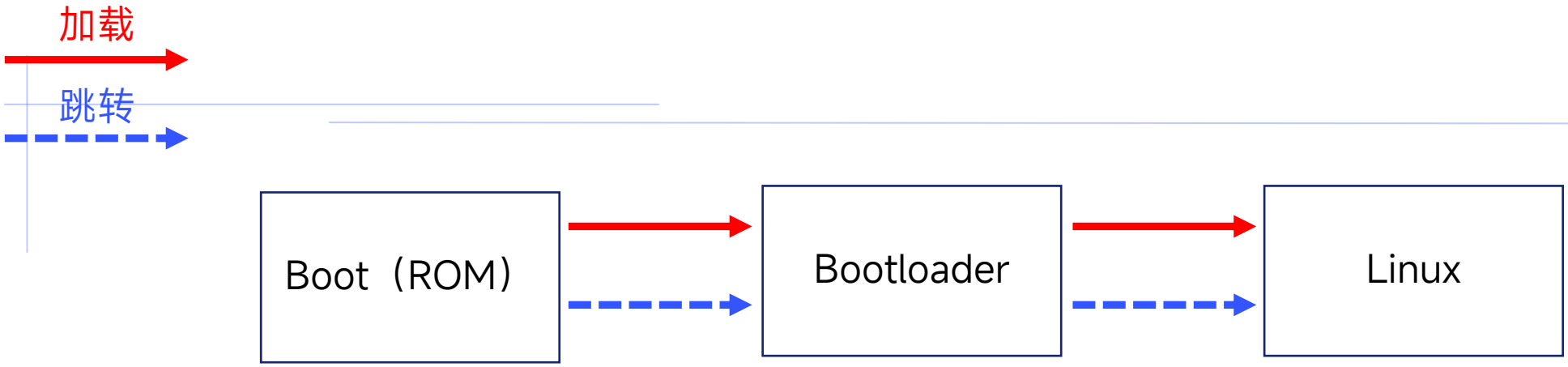
Linux 系统的启动过程例子



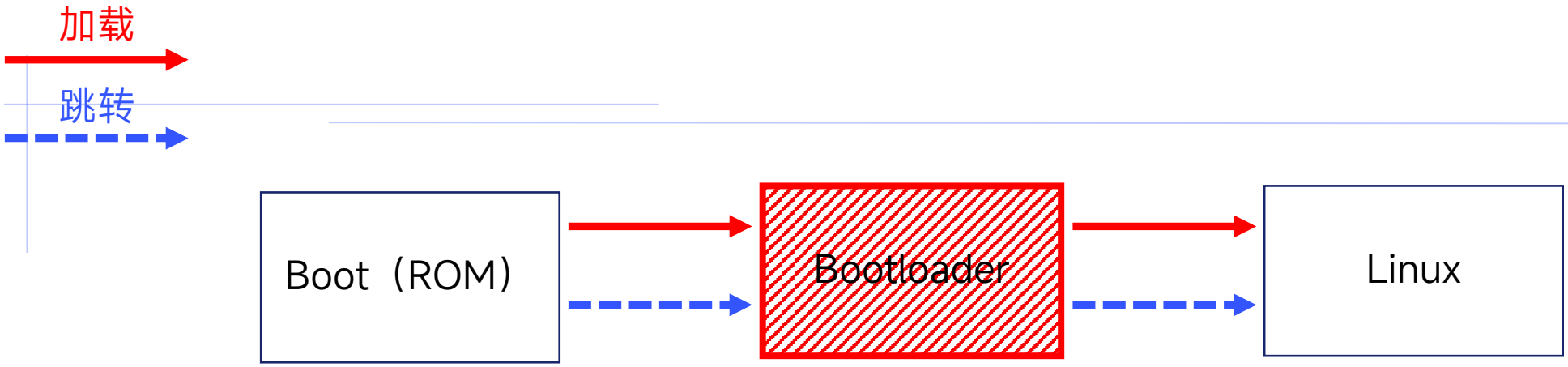
Linux 系统的启动过程例子



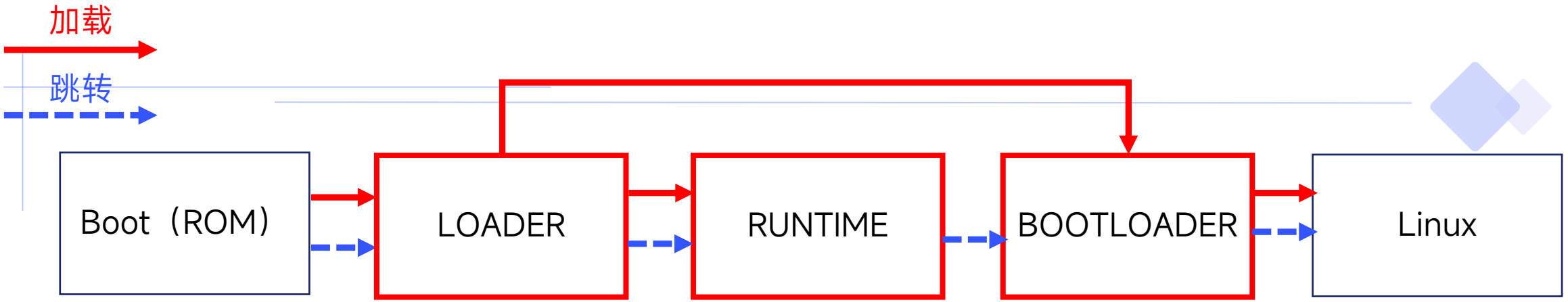
Linux 系统的启动过程例子



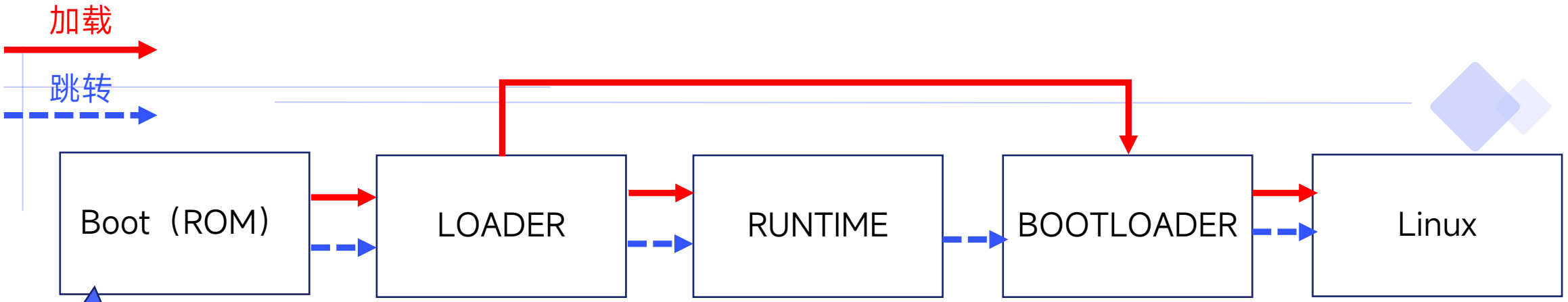
Linux 系统的启动过程例子



Linux 系统的启动过程（细化）

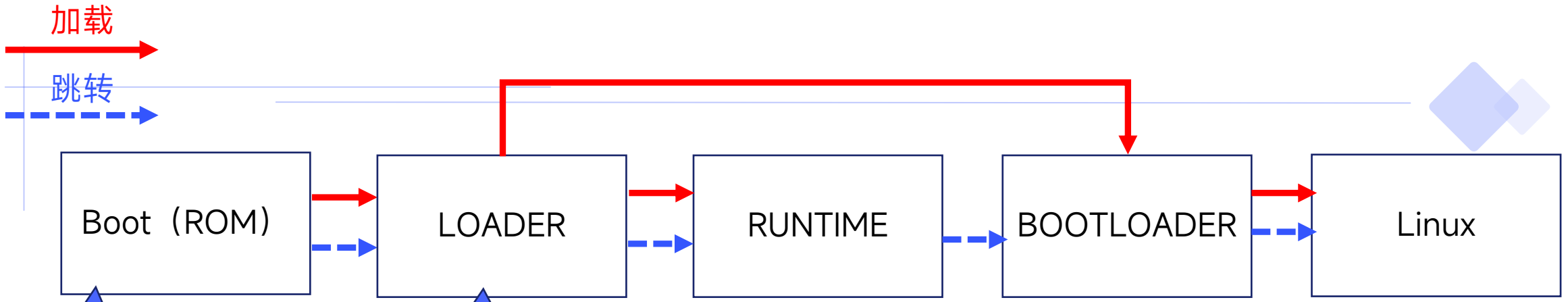


Linux 系统的启动过程（细化）



- 片上 ROM 启动
- 使用片上的 SRAM
- 芯片基本初始化

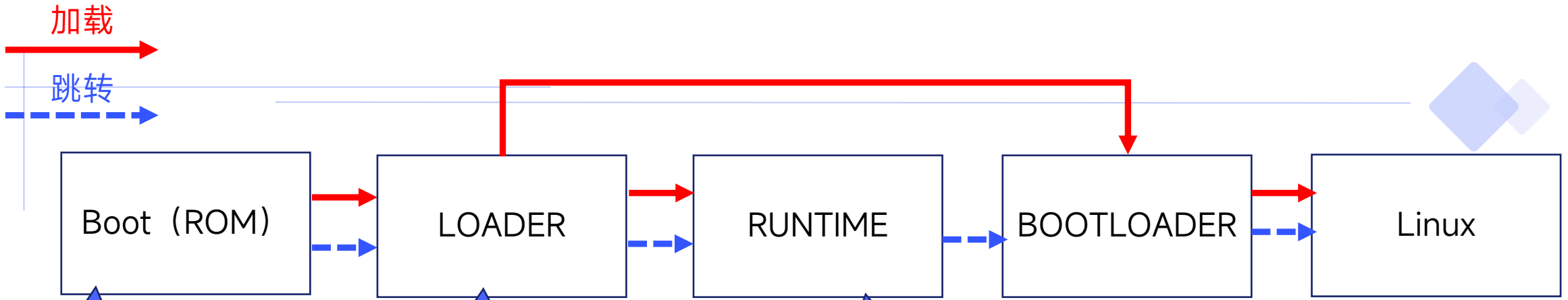
Linux 系统的启动过程（细化）



- 片上 ROM 启动
- 使用片上的 SRAM
- 芯片基本初始化

- 片上 SRAM 或者 DRAM 运行
- 负责加载 RUNTIME 和 BOOTLOADER
- 例子：U-boot SPL

Linux 系统的启动过程（细化）

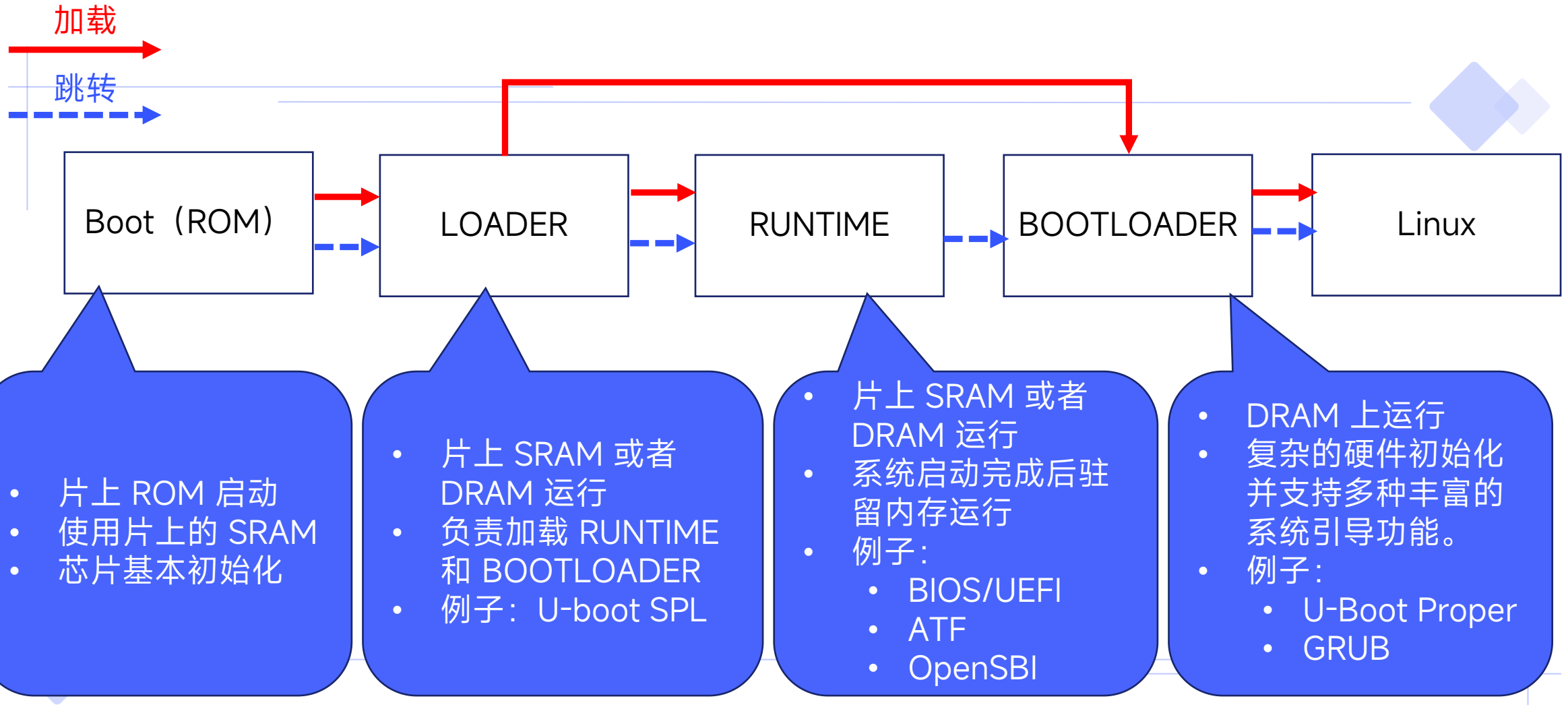


- 片上 ROM 启动
- 使用片上的 SRAM
- 芯片基本初始化

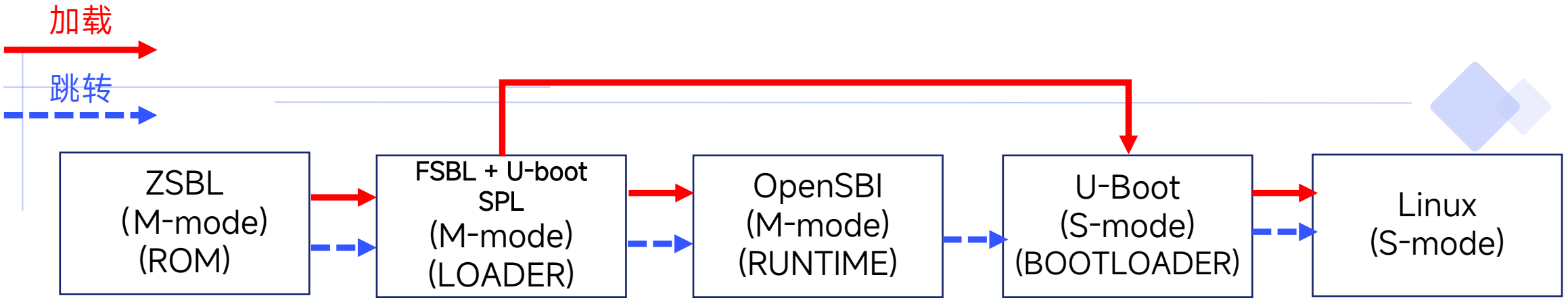
- 片上 SRAM 或者 DRAM 运行
- 负责加载 RUNTIME 和 BOOTLOADER
- 例子：U-boot SPL

- 片上 SRAM 或者 DRAM 运行
- 系统启动完成后驻留内存运行
- 例子：
  - BIOS/UEFI
  - ATF
  - OpenSBI

Linux 系统的启动过程（细化）

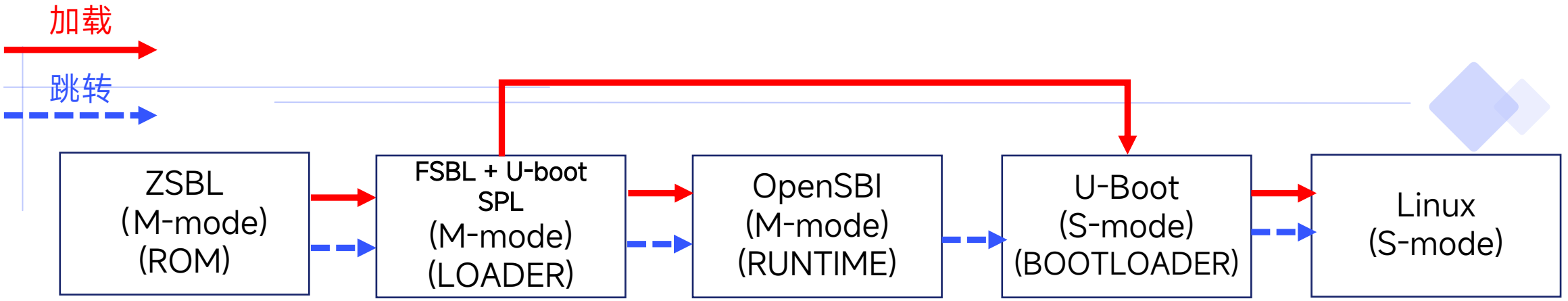


RISC-V Linux 系统的启动过程

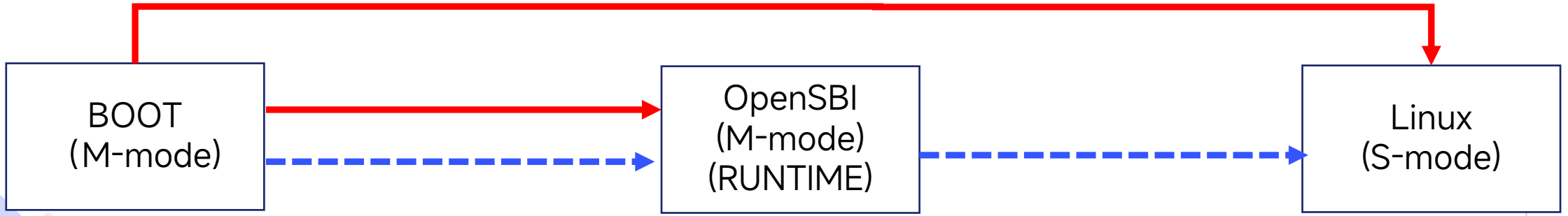


以 HiFive Unleashed 为例

RISC-V Linux 系统的启动过程



以 HiFive Unleashed 为例



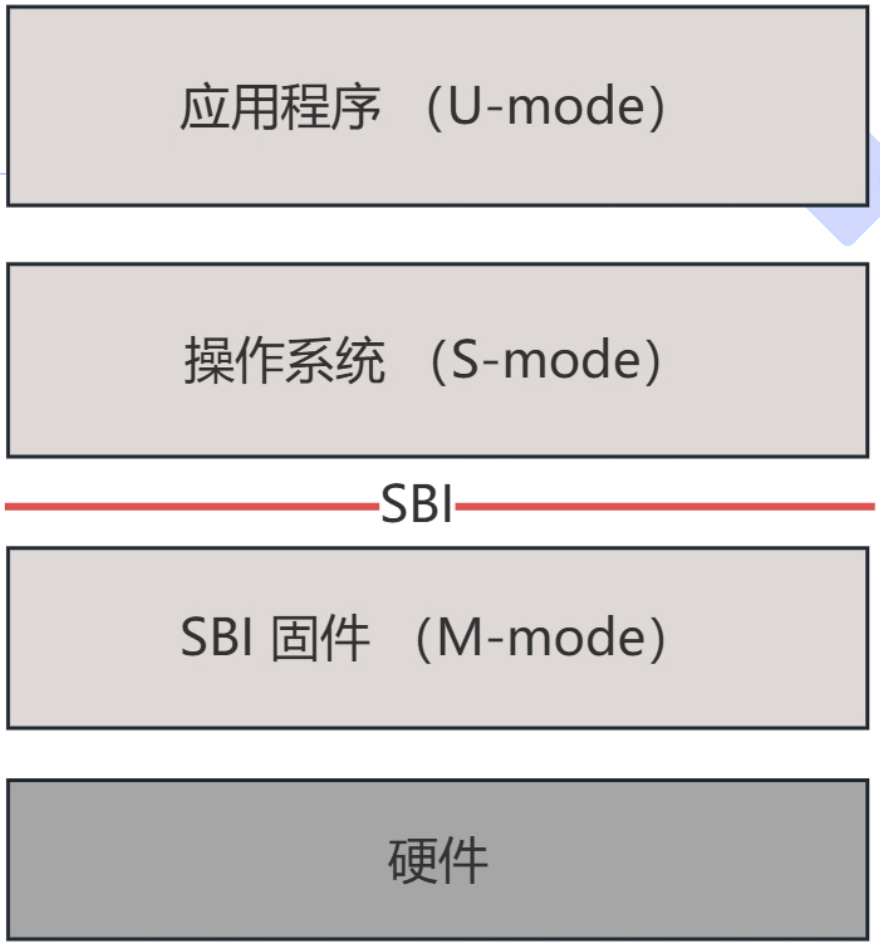
以 QEMU virt riscv64 为例：qemu-system-riscv64 …… -bios fw_jump.bin -kernel Image

02

OpenSBI 介绍

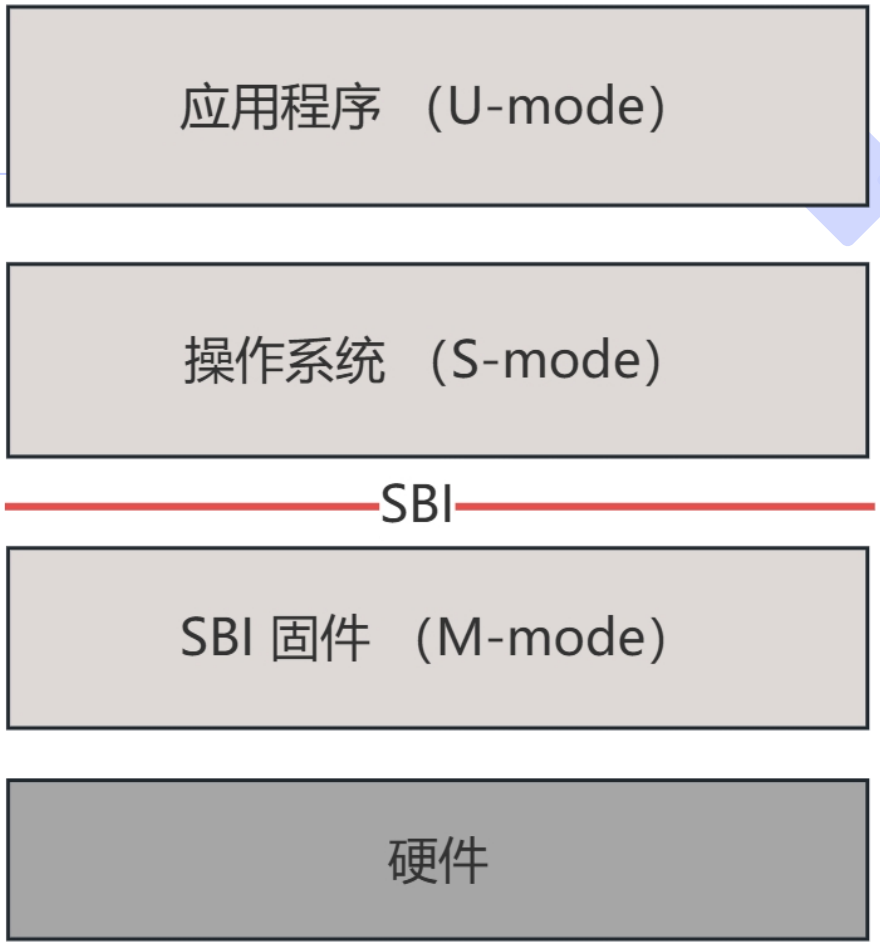
什么是 SBI

- SBI 全称是 **Supervisor Binary Interface**，即 **监管者二进制接口**
- SBI 是一个 RISC-V 软件栈中特有的 **接口标准**。它定义了一组功能调用（如设置定时器、发送核间中断等），规定了内核如何与下层固件进行交互。
- SBI 固件也称为 **SEE（Supervisor Execution Environment）**，特指在 RISC-V 系统中，为了支持一个完整的操作系统内核（运行在 Supervisor Mode，即 S 模式）所需要提供的运行在 Machine 模式的软件环境。



SBI 的作用

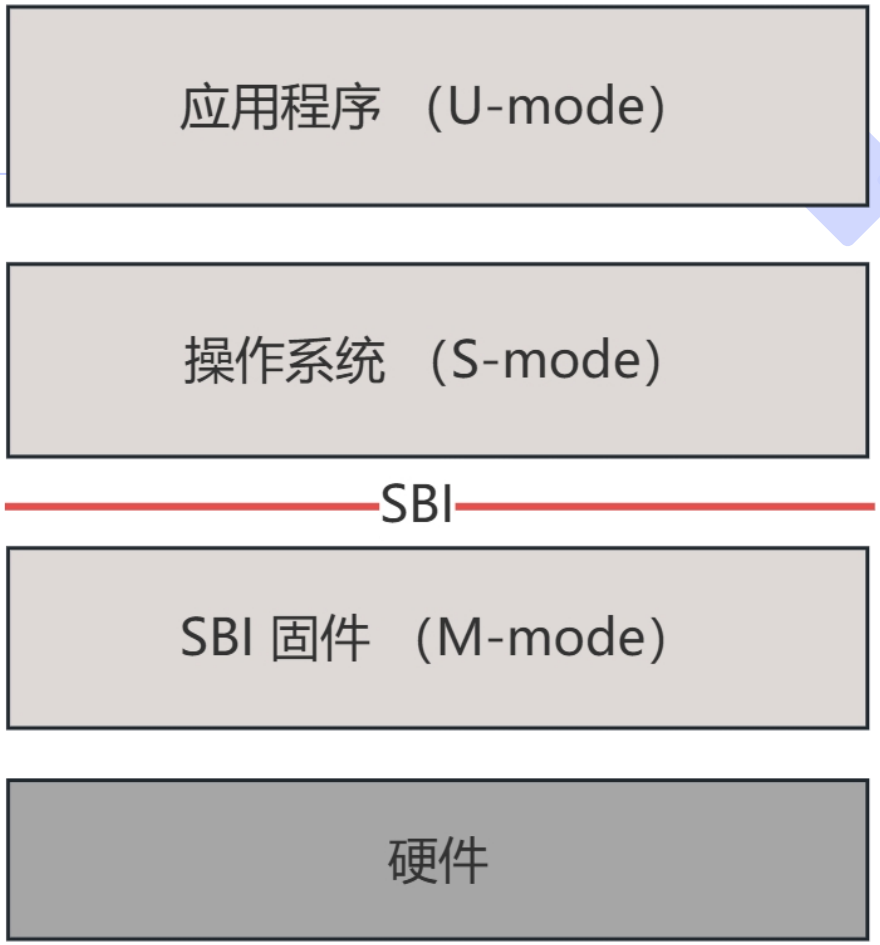
- **标准化 S-mode 软件的移植**：通过定义清晰的 SBI 规范，方便 S-mode 的软件（譬如操作系统）在不同 RISC-V 芯片 上的移植。
- **封装硬件**：硬件平台的差异性封装在 M 模式的固件中。运行在 S-mode 的软件通过标准的 SBI 调用与硬件交互，从而与具体硬件解耦。
- **提供关键服务**：SEE 通过其 SBI 固件为内核提供了启动多核、处理中断、设置定时器、电源管理等关键服务，这些都是现代操作系统运行所不可或缺的。同时，这些敏感操作必须在最高权限级别执行。通过 SBI，内核将这类请求“委托”给可信的 M 模式固件去执行，而不是直接操作硬件。这有助于隔离错误，提高系统的稳定性和安全性。



SBI 调用的执行流程

当运行在 S-mode 的软件（譬如 Linux 内核）需要访问硬件特权功能（如发送核间中断、管理时钟、关机）时，它不能直接执行 M-mode 指令，而是必须通过主动触发 Trap 到 M-mode。这个过程如下：

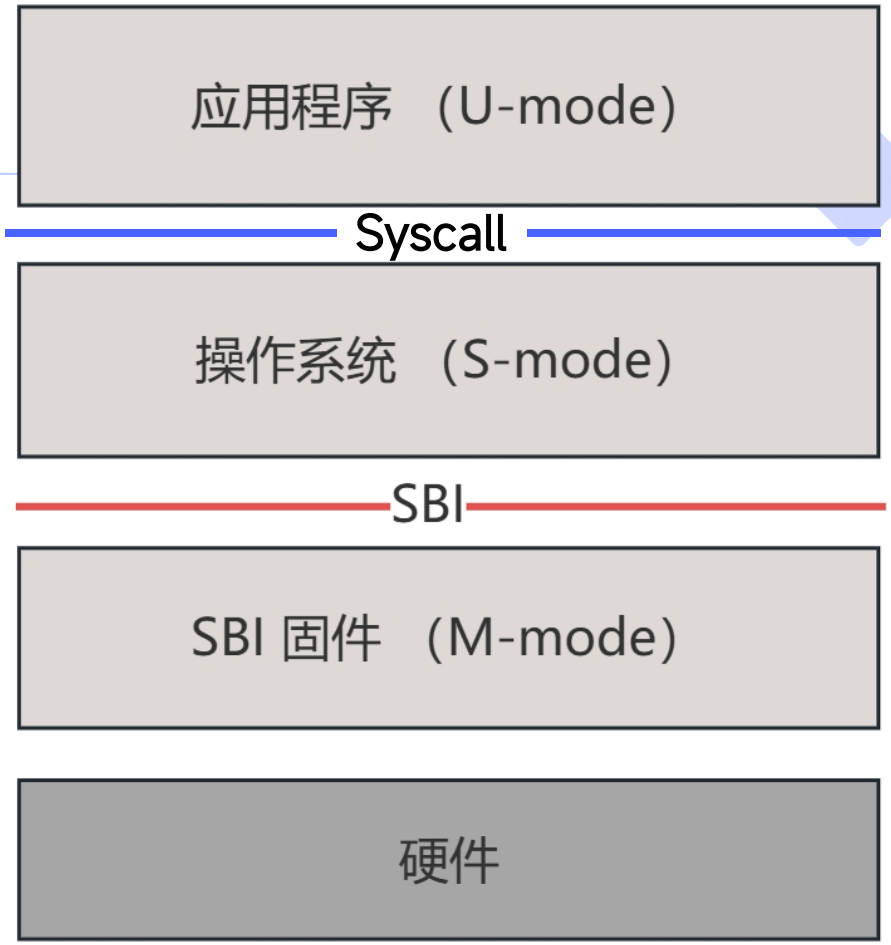
- 操作系统调用 ecall 指令
- CPU 自动从 S-mode 切换回 M-mode。
- M-mode 的异常处理程序（由 SBI 在启动时设置，地址在 mtvec 寄存器）接管执行。
- 异常处理程序（即 SBI 的运行时服务）根据 ecall 传递的参数（如 a7 寄存器中的 SBI 功能号）执行相应的 M-mode 特权操作。
- 操作完成后，SBI 通过 mret 指令将 CPU 特权级切换回之前的 S-mode，并返回到上层操作系统中 ecall 指令之后的位置继续执行。



SBI 调用的执行流程

当运行在 S-mode 的软件（譬如 Linux 内核）需要访问硬件特权功能（如发送核间中断、管理时钟、关机）时，它不能直接执行 M-mode 指令，而是必须通过主动触发Trap 到 M-mode。这个过程如下：

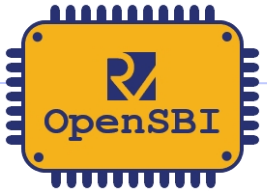
- 操作系统调用 ecall 指令
- CPU 自动从 S-mode 切换回 M-mode。
- M-mode 的异常处理程序（由 SBI 在启动时设置，地址在 mtvec 寄存器）接管执行。
- 异常处理程序（即 SBI 的运行时服务）根据 ecall 传递的参数（如 a7 寄存器中的 SBI 功能号）执行相应的 M-mode 特权操作。
- 操作完成后，SBI 通过 mret 指令将 CPU 特权级切换回之前的 S-mode，并返回到上层操作系统中 ecall 指令之后的位置继续执行。



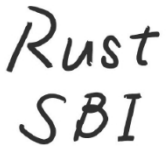
常见的 SBI 实现

- **OpenSBI**：由 RISC-V 官方社区维护的、开源的参考实现。它被广泛用于 QEMU 模拟器和许多硬件产品上。

  https://github.com/riscv-software-src/opensbi



- **RustSBI**：一个用 Rust 语言编写的 SBI 实现。https://rustsbi.com/



- **Xvisor**：一个更轻量级的 Hypervisor 实现，提供 SBI 实现。

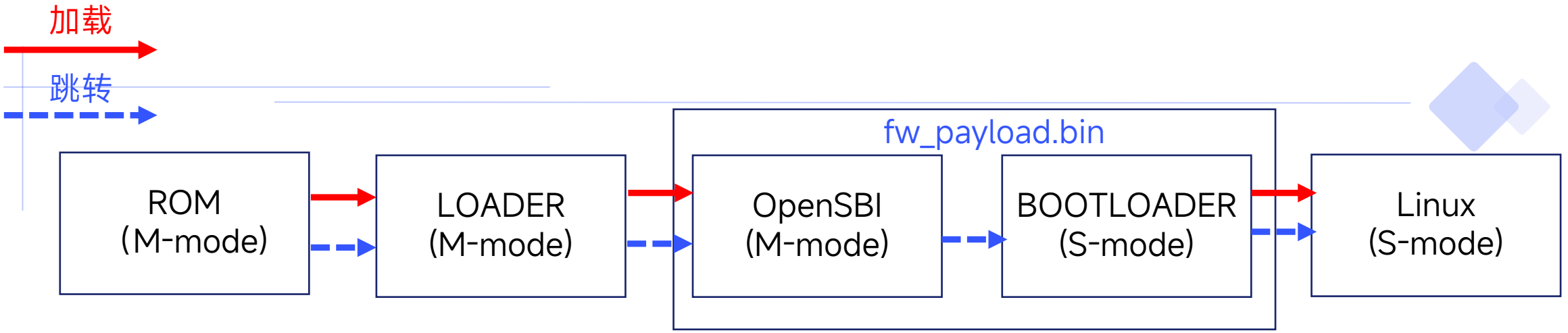
  https://xhypervisor.org/

- **定制版本**：芯片厂商也会提供其自有的、针对特定硬件优化过的 SBI 固件。

03

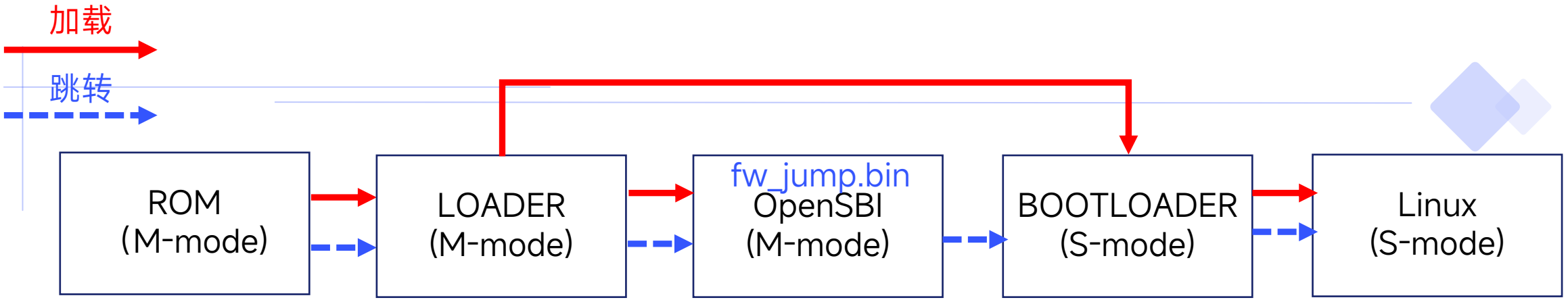
制作 OpenSBI

OpenSBI 固件类型 - FW_PAYLOAD



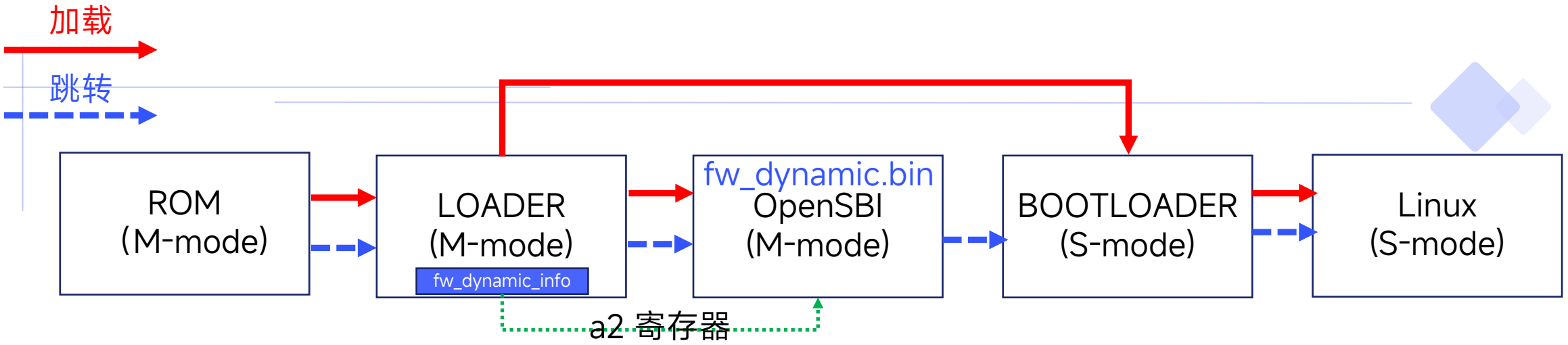
- fw_payload.bin 将下一阶段（如 BOOTLOADER 或 Linux）作为 PAYLOAD 和 RUNTIME（OpenSBI）打包在一起。
- OpenSBI 和 PAYLOAD 编译为一个整体，OpenSBI 执行完后直接跳转到 PAYLOAD 的编译地址处开始执行。
- 当 OpenSBI 或者 PAYLOAD 改变，必须重新创建 fw_payload.bin

OpenSBI 固件类型 - FW_JUMP



- fw_jump.bin 和下一阶段（如 BOOTLOADER 或 Linux）是独立的 image。由 LOADER 负责分别加载到各自的位置。
- 编译生成 fw_jump.bin 时通过参数 FW_JUMP_ADDR 指定 OpenSBI 下一阶段需要跳转的地址，OpenSBI 执行完后根据这个地址再跳转到下一个阶段执行。
- 对于 OpenSBI 来说方便加载预编译的下一阶段 image。

OpenSBI 固件类型 - FW_DYNAMIC



- fw_dynamic.bin 和下一阶段（如 BOOTLOADER 或 Linux）是独立的 image。LOADER 负责将它们分别加载到各自的位置。
- LOADER 中定义结构体 fw_dynamic_info 存放 OpenSBI 的下一个阶段启动信息，并通过 a2 寄存器将结构体地址传递给 OpenSBI 以获取该信息。
- 引导过程最灵活，便于切换不同的 BOOTLOADER 或者 Linux 内核。

OpenSBI 固件类型

特性	FW_PAYLOAD	FW_JUMP	FW_DYNAMIC
工作原理	将下一阶段（如 OS 内核或 Bootloader）**直接嵌入** 到 OpenSBI固件中	已知下一阶段的**固定入口地址**，直接跳转	通过 **动态信息结构体** 从上一阶段获取下一阶段的地址和参数
启动流程	前一阶段 → OpenSBI → 内嵌的 Payload	前一阶段 → OpenSBI → 指定地址的下一阶段	前一阶段（传递参数）→ OpenSBI → 参数指定的下一阶段
适用场景	方便一体化部署与量产	启动流程固定且明确	需要灵活切换不同内核或引导程序；方便开发与调试
灵活性	低（Payload在编译时必有得有）	中（需在编译时确定跳转地址）	高（下一阶段地址可在运行时确定）
构建复杂性	中（需指定 Payload 路径）	低（只需指定跳转地址）	低（固件本身无需特殊配置）

OpenSBI 固件类型

特性	FW_PAYLOAD	FW_JUMP	FW_DYNAMIC
工作原理	将下一阶段（如 OS 内核或 Bootloader）**直接嵌入** 到 OpenSBI固件中	已知下一阶段的**固定入口地址**，直接跳转	通过 **动态信息结构体** 从上一阶段获取下一阶段的地址和参数
启动流程	前一阶段 → OpenSBI → 内嵌的 Payload	前一阶段 → OpenSBI → 指定地址的下一阶段	前一阶段（传递参数）→ OpenSBI → 参数指定的下一阶段
适用场景	前一阶段无法加载多个镜像；或无法传递 FDT	前一阶段能分别加载 OpenSBI 和下一阶段程序	需要灵活切换不同内核或引导程序；通用场景
灵活性	低（Payload在编译时必有得有）	中（需在编译时确定跳转地址）	高（下一阶段地址可在运行时确定）
构建复杂性	中（需指定 Payload 路径）	低（只需指定跳转地址）	低（固件本身无需特殊配置）

OpenSBI 固件的制作

配置（无需配置，采用默认方式）

构建

```
eval "${TARGET_MAKE_ENV} CROSS_COMPILE=${CROSS_COMPILE}  PLATFORM=generic
/usr/bin/make -j${MAXNUM_CPUS} -C ${PKGBUILD_DIR}"
```

安装 （install-image）

```
/usr/bin/install -m 0644 -D
${PKGBUILD_DIR}/build/platform/generic/firmware/fw_jump.bin
${IMAGES_DIR}/fw_jump.bin
```

OpenSBI 快速演示

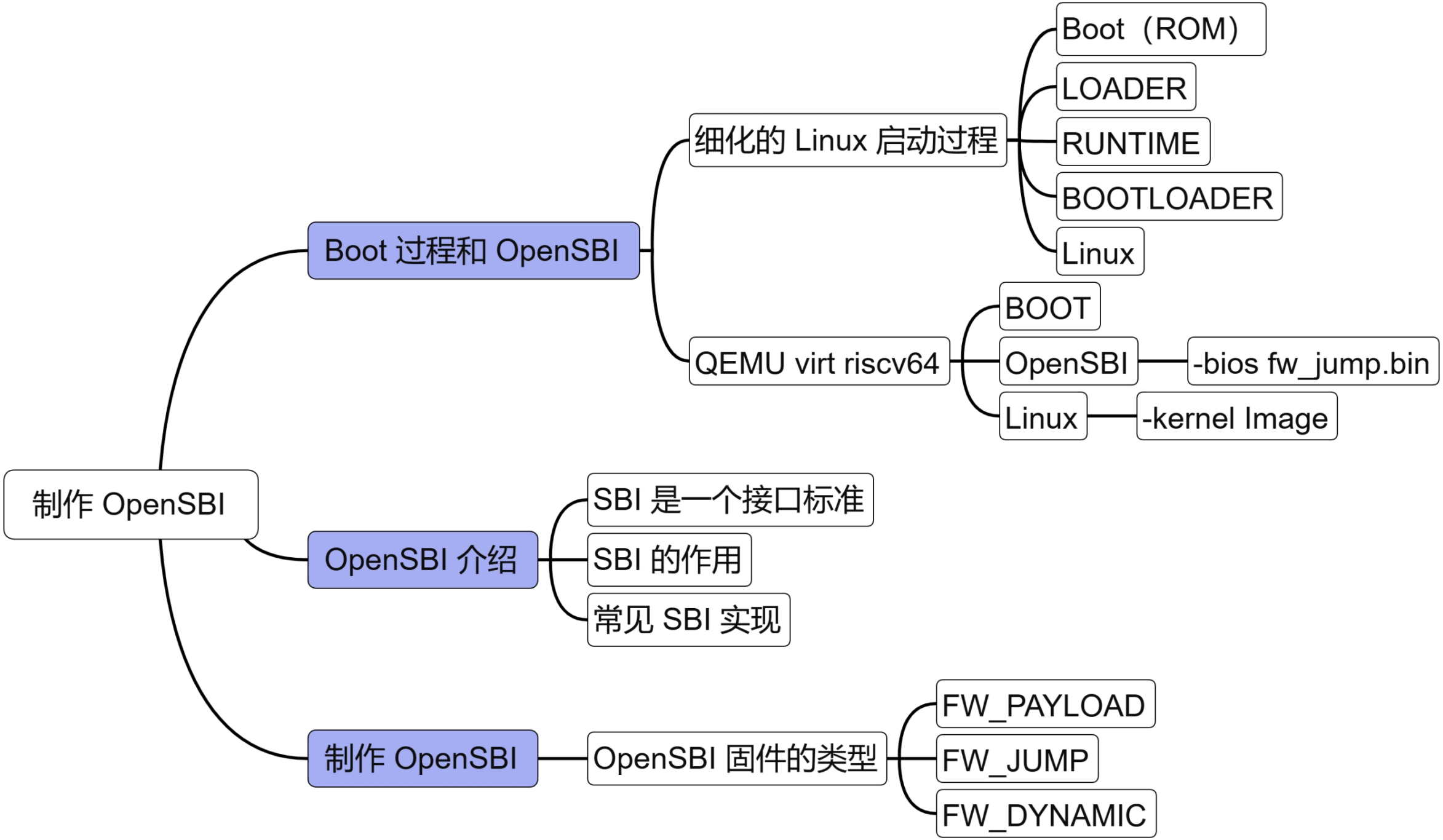
课程仓库代码
package/opensbi/hello.s
package/opensbi/hello.sh

等效 C 代码：

```
void puts(char *str)
{
    while(*str) {
        putc(*str);
        str++;
     }
}

void main()
{
    static const char hello_string[] = “Hello RISC-V world!”;
    puts(hello_string);
}
```

本章总结


制作 OpenSBI
Boot 过程和 OpenSBI
细化的 Linux 启动过程
Boot（ROM）
LOADER
RUNTIME
BOOTLOADER
Linux
QEMU virt riscv64
BOOT
OpenSBI
-bios fw_jump.bin
Linux
-kernel Image
OpenSBI 介绍
SBI 是一个接口标准
SBI 的作用
常见 SBI 实现
制作 OpenSBI
OpenSBI 固件的类型
FW_PAYLOAD
FW_JUMP
FW_DYNAMIC

谢谢