



# **Tina Linux 启动优化 开发指南**

**版本号: 1.2**  
**发布日期: 2021.04.20**

## 版本历史

版本号	日期	制/修订人	内容描述
0.1	2019.01.29	AWA1046	初版
1.0	2019.02.14	AWA0916	增加 Tina 启动优化配置
1.1	2019.06.04	AWA1046	完善说明，更新配置文件路径
1.2	2021.04.20	AWA0916	新增硬件平台支持，完善配置文件路径



## 目 录

<b>1 概述</b>	<b>1</b>
1.1 编写目的	1
1.2 适用范围	1
1.3 相关人员	1
<b>2 启动速度优化简介</b>	<b>2</b>
2.1 启动流程	2
2.2 测量方法	3
2.2.1 printk time	3
2.2.2 initcall_debug	3
2.2.3 bootgraph	3
2.2.4 bootchart	4
2.2.5 gpio + 示波器	4
2.2.6 grabserial	4
2.3 优化方法	5
2.3.1 boot0 启动优化	5
2.3.2 uboot 启动优化	5
2.3.2.1 完全去掉 uboot	5
2.3.2.2 避免 burnkey 的影响	6
2.3.2.3 提高 CPU 以及 flash 读取频率	6
2.3.2.4 关闭串口输出	6
2.3.2.5 修改 kernel 加载位置	7
2.3.2.6 修改 kernel 加载大小	7
2.3.2.7 关闭 kernel 校验	8
2.3.2.8 uboot 重定位	8
2.3.2.9 裁剪 uboot	8
2.3.2.10 开启 logo 及音乐	8
2.3.3 kernel 启动优化	8
2.3.3.1 kernel 压缩方式	9
2.3.3.2 加载位置	9
2.3.3.3 内核裁剪	9
2.3.3.4 预设置 lpj 数值	9
2.3.3.5 initcall 优化	10
2.3.3.6 内核 initcall module 并行	10
2.3.3.7 减少 pty/tty 个数	10
2.3.3.8 内核 module	10
2.3.3.9 Deferred Initcalls	11
2.3.4 rootfs 启动优化	11
2.3.4.1 initramfs	11
2.3.4.2 rootfs 类型以及压缩	11
2.3.4.3 rootfs 裁剪	12

2.3.4.4 指定文件系统类型 . . . . .	12
2.3.4.5 静态创建 dev 节点 . . . . .	12
2.3.4.6 rootfs 拆分 . . . . .	12
2.3.5 主应用程序启动优化 . . . . .	12
<b>3 Tina 启动速度优化</b>	<b>13</b>
3.1 开启 Tina 启动速度优化 . . . . .	13
3.2 实验结果 . . . . .	14
<b>4 参考资料</b>	<b>15</b>



# 1 概述

---

## 1.1 编写目的

介绍 TinaLinux 下启动速度优化使用方法。

## 1.2 适用范围

硬件平台：全志 R/V/F/MR 系列芯片。

软件平台：Tina V3.5 及其后续版本。

## 1.3 相关人员

适用于 TinaLinux 平台的客户及相关技术人员。

## 2 启动速度优化简介

启动速度是嵌入式产品一个重要的性能指标，更快的启动速度会让客户有更好的使用体验，在某些方面还会节省能耗，因为可以直接关机而不需要休眠。

启动速度优化可提升产品的竞争力。对于某些系统来说，启动速度是硬性要求。

### 2.1 启动流程

TinaLinux 系统当前的启动流程如下：

```
brom --> boot0 --> (monitor/secure os) --> uboot --> rootfs --> app
```

brom 固化在 IC 内部，芯片出厂后就无法更改。

后续将从 boot0 开始分阶段介绍启动优化的方法。

对于某些方案，会存在 monitor 或 secure os，这两者耗时很短，本文略过。

下文涉及到一些配置文件，提前在此说明。

env 配置文件路径：

```
tina/device/config/chips/<chip>/configs/<board>/env.cfg #优先级高  
tina/device/config/chips/<chip>/configs/<board>/linux/env-<kernel-version>.cfg #优先级中  
tina/device/config/chips/<chip>/configs/default/env.cfg #优先级低  
tina/target/allwinner/<board>/configs/env-<kernel-version>.cfg #旧SDK
```

sys\_config.fex 路径：

```
tina/device/config/chips/<chip>/configs/<board>/sys_config.fex #新SDK  
tina/target/allwinner/<board>/configs/sys_config.fex #旧SDK
```

uboot-board.dts 路径：

```
tina/device/config/chips/<chip>/configs/<board>/uboot-board.dts
```

#### 警告

如果存在 uboot-board.dts，uboot 会使用 uboot-board.dts 中配置；如果不存在 uboot-board.dts，uboot 会使用 sys\_config.fex 中的配置。

## 2.2 测量方法

### 2.2.1 printk time

打开 kernel 配置，使能如下选项：

```
kernel hacking --->
[*] Show timing information on printks
```

将会在内核的 log 前加入时间戳。

**注：此方法主要用来测量内核启动过程中各个阶段的耗时。**

### 2.2.2 initcall\_debug

修改 env 文件，在 kernel 的 cmdline 中加入参数，

```
# 增加initcall_debug变量
initcall_debug=1

# 将initcall_debug=${initcall_debug} 加入 setargs_xxx 中, 如 setargs_nand, setargs_mmc,
  setargs_nor, setatgs_nand_ubi 等,
setargs_nand=setenv bootargs console=${console} earlyprintk=${earlyprintk} root=${nand_root
} initcall_debug=${initcall_debug} init=${init}
```

开启之后，启动中会打印每个 initcall 函数调用及其耗时。

**注：此方法主要用来测量内核 initcall 的耗时。**

一般需同时配置上内核符号表，即 kallsyms 选项，以打印函数名。

### 2.2.3 bootgraph

在内核源码中自带了一个工具 (scripts/bootgraph.pl) 可用于分析启动时间。

- kernel 编译时需要包含 CONFIG\_PRINTK\_TIME 选项。
- 在 kernel cmdline 加上 "initcall\_debug=1"。
- 在系统启动完毕后执行 "dmesg | perl \$(Kernel\_DIR)/scripts/bootgraph.pl > output.svg"。
- 使用 SVG 浏览器（比如 Inkscape, Gimp, Firefox 等）来查看输出文件 output.svg。

**注：此方法主要用来测量内核启动过程中各个阶段的耗时。**

## 2.2.4 bootchart

bootchart 是一个用于 linux 启动过程性能分析的开源软件工具，在系统启动过程自动收集 CPU 占用率、进程等信息，并以图形方式显示分析结果，可用作指导优化系统启动过程。

- 修改 kernel cmdline。修改 env 配置文件 (路径见上文说明)，将其中的 init 修改为 "init=/sbin/bootchartd"。
- 收集信息。bootchartd 会从 /proc/stat, /proc/diskstat, /proc/[pid]/stat 中采集信息，经过处理后保存为 bootchart.tgz 文件。
- 转换图片。在 PC 上通过 pybootchartgui.py 工具将 bootchart.tgz 转换为 bootchart.png，方便分析。

**注：此方法主要用来测量挂载文件系统到主应用程序启动过程中的耗时。**

## 2.2.5 gpio + 示波器

在适当的地方加入操作 gpio 的代码，通过示波器抓取波形得到各阶段耗时。

**注：此方法可用来测量整个启动中各阶段的耗时。**

## 2.2.6 grabserial

Grabserial 是 Tim Bird 用 python 写的一个抓取串口的工具，这个工具能够为收到的每一行信息添加上时间戳。可从如下路径下载使用：<https://github.com/tbird20d/grabserial>

介绍文档：<http://elinux.org/Grabserial>

常见的用法：

```
sudo grabserial -v -S -d /dev/ttyUSB0 -e 30 -t
```

如果要在某个字符串重置时间戳，可以使用 -m 参数：

```
sudo grabserial -v -S -d /dev/ttyUSB0 -e 30 -t -m "Starting kernel"
```

- -v 显示参数等信息。
- -s 跳过对串口的检查。
- -d 指定串口，如上述为指定 /dev/ttyUSB0 为操作的串口。
- -e 参数指定时间，如上述命令表示抓取 30s 的串口记录。
- -t 表示加上时间戳。
- -m 匹配到指定字符串就重置时间戳的时间，也就是从 0 开始。



更多配置可以使用 -h 参数查看帮助。

**注：此方法可用来测量整个启动中各阶段的耗时。**

## 2.3 优化方法

**注：本节提供一些优化方法以供参考，并非所有都在 Tina 上集成，主要原因有：**

- 优化没有止境。需要根据目标来选择优化方法，综合考虑优化效果与优化难度。
- 优化需要具有针对性。由于各方案 CPU 个数及频率、flash 类型及大小、kernel/rootfs 压缩类型与尺寸、所需功能、主应用等的不同，需要针对性的进行优化。

### 2.3.1 boot0 启动优化

boot0 运行在 SRAM，主要功能是对 DRAM 进行初始化，并将 uboot 加载至 DRAM。

对于安全方案来说，boot0 还会对 uboot、monitor、secure-os 等进行签名校验。

boot0 可优化的地方不多，可以做的是：

- 关闭串口输出。
- 减少检测按键和检测串口的等待时间。
- 加载 uboot 的时候，不要先加载后搬运，直接加载到 uboot 的运行地址。

对于 spinor 的方案，还可以直接从 boot0 启动，只需要在 boot0 中加载好 kernel 和 dtb，不需要经过 uboot，然后直接跳转到 kernel 运行，可节省一定的时间。如果采用 boot0 启动 OS，则 boot0 读取数据量较大，其 flash 驱动也需要进行优化，如提高时钟，开启双线/四线/DMA/Cache 等。

### 2.3.2 uboot 启动优化

uboot 主要功能是引导内核、量产升级、电源管理、开机音乐/logo、fastboot 刷机。

#### 2.3.2.1 完全去掉 uboot

uboot 的包含很多重要功能，通常会保留。某些情况可以去掉，直接从 boot0 加载内核并启动，可节省一些时间。

### 2.3.2.2 避免 burnkey 的影响

对于启用了 burnkey 支持，且还没使用 DragonSN 工具将 key 烧录进去的板子，每次启动到 uboot 都会尝试跟 PC 端工具交互产生如下 log，带来延时。

```
[1.334]usb burn from boot
...
[1.400]usb prepare ok
usb sof ok
[1.662]usb probe ok
[1.664]usb setup ok
...
[4.698]do_burn_from_boot usb : have no handshake
```

如果产品不需要 burnkey，可将 uboot-board.dts 或 sys\_config.fex 中的 [target] 下 burn\_key 设置为 0。

或者使用 DragonSN 工具，烧录一次 key，并设置烧录标志，以使后续启动可跳过检测。

### 2.3.2.3 提高 CPU 以及 flash 读取频率

可设置 uboot-board.dts 或 sys\_config.fex 中的 [target] 下 boot\_clock 来修改 uboot 运行时 CPU 频率（注：不能超过 SPEC 最大频率）。

对于 spinor/spinand，使用较高的时钟频率（一般是 100M），使用四线模式或者双线模式（看硬件是否支持），提高加载速度。

### 2.3.2.4 关闭串口输出

可将 uboot-board.dts 或 sys\_config.fex 中的 [platform] 下 debug\_mode 设置为 0 来关闭 uboot 的串口输出。

可将 sys\_config.fex 中的 [platform] 下 debug\_mode 设置为 0 来关闭 boot0 串口输出。

配置此项后，如果还有少量输出，有两个可能的原因：

第一是这些输出是在获取 debug\_mode 流程之前产生。

第二是因为源码中直接使用了 puts 而没有使用 printf。

对于这两者情况，需要修改源码来完全关闭串口输出。

### 2.3.2.5 修改 kernel 加载位置

如果 uboot 将内核加载到 DRAM 的地址与内核中 load address 不匹配，就需要将内核移动到正确位置，这样会浪费一定的时间。因此，可以直接修改 uboot 加载内核为正确的地址。

具体是修改 env 文件 (路径见上文) 的 boot\_normal 与 boot\_recovery 变量。

需要根据不同的内核镜像格式来设置不同的值。

假设 kernel 的 load address 为 0x40008000。

- 如果使用的是 uImage，也就是在 kernel 的镜像前加了 64 字节，所以 uboot 应该将 kernel 加载到  $0x40008000 - 0x40 = 0x40007fc0$ 。

```
#uImage/raw
boot_normal=sunxi_flash read 40007fc0 ${boot_partition};bootm 40007fc0
boot_recovery=sunxi_flash read 40007fc0 recovery;bootm 40007fc0
```

- 如果使用的是 boot.img，即 android 的 kernel 格式，其头部大小为 0x800，所以 uboot 应该将 kernel 加载到  $0x40008000 - 0x800 = 40007800$ 。

```
#boot.img/raw
boot_normal=sunxi_flash read 40007800 ${boot_partition};bootm 40007800
boot_recovery=sunxi_flash read 40007800 recovery;bootm 40007800
```

如果 uboot 加载 kernel 地址与 load address 不匹配，uboot 过程中串口输出可能会有：

```
Loading Kernel Image ... OK
```

如果是匹配的，uboot 过程中串口输出可能会有：

```
XIP Kernel Image ... OK
```

### 2.3.2.6 修改 kernel 加载大小

最新代码会根据 uImage/boot.img 的头部信息，只读取必要的大小，可忽略此优化项。

对于旧代码，uboot 在加载内核的时候，有些情况会直接将整个分区读取出来。

就是说假如内核只有 2M，而分区分了 4M 的话，uboot 就会读取 4M。这种情况下，可以将分区大小设置得刚好容纳下内核，这样可避免 uboot 在加载内核的时候浪费时间。

可修改 sys\_partition\*.fex 中 boot 分区的大小。

uboot 具体读出多少，通常会有 log 信息，可同真正内核镜像的 size 进行比较。

### 2.3.2.7 关闭 kernel 校验

uboot 加载了内核以后，默认会对内核进行校验，可以在串口输出中看到：

```
Verifying Checksum ... OK
```

如果不想校验可以去掉，目前的情况是可以减少几十毫秒（不同平台，不同内核大小，时间不同）的启动时间。

具体修改 env 配置文件（路径见上文），新增一行"verify=no"。

### 2.3.2.8 uboot 重定位

目前的启动过程中，uboot 在执行过程中会进行一次重定位，可以在串口中打印出这个值，然后修改 uboot 的加载地址使得 boot0 将 uboot 加载进 DRAM 的时候就直接加载到这个地址。

修改文件 tina/lichee/brandy\*/u-boot\*/include/configs/sun\*iw\*p\*.h 中的

```
#define CONFIG_SYS_TEXT_BASE (0x40900000)
```

但这个方法有个弊端，如果后续修改了 uboot 的代码，则可能需要重新设置。

目前这个操作耗时很少（某平台测得十几毫秒），不必要的话不建议做这个修改。

### 2.3.2.9 裁剪 uboot

即使流程没有简化，uboot 体积的减小也可减少加载 uboot 的时间。

依据具体情况，可对 uboot 不需要的功能的模块进行裁剪，避免了启动中执行不必要的流程，可减少 uboot 加载时间。

### 2.3.2.10 开启 logo 及音乐

可尝试在 uboot 中开启开机 logo/音乐，尽快播出第一帧/声，提升用户体验。

此操作会延缓到达 OS/APP 的时间，但如果产品定义/用户体验是以第一帧/声为准的话，则有较大价值。

## 2.3.3 kernel 启动优化

通常来说，内核启动耗时较多，需要更深入的优化。

### 2.3.3.1 kernel 压缩方式

比较不同压缩方式的启动时间和 flash 占用情况，选择一种符合实际情况的。

此处给出某次测试结果供参考。实际优化的时候，需要重新测试，根据实际情况选择。

压缩方式	内核大小/M	加载时间/s	解压时间/s	总时间/s
LZO	2.4	0.38	0.23	0.61
GZIP	1.9	0.35	0.44	0.79
XZ	1.5	0.25	2.17	2.42

### 2.3.3.2 加载位置

内核镜像可以由 kernel 自解压，也有 uboot 进行解压的情况。

对于 kernel 自解压的情况，如果压缩过的 kernel 与解压后的 kernel 地址冲突，则会先把自己复制到安全的地方，然后再解压，防止自我覆盖。这就需要耗费复制的时间。

比如对于运行地址为 0x80008000 的内核来说，bootloader 可以将其加载到 0x81008000，当然其他位置也可以。

### 2.3.3.3 内核裁剪

裁剪内核，带来的加速是两个方面的。一是体积变小，加载解压耗时减少；二是内核启动时初始化内容变少。

裁剪要根据产品的实际情况来，将不需要的功能及模块都去掉。

具体是执行"make kernel\_menuconfig"，关闭不需要的选项。可参考《TinaLinux\_系统裁剪开发指南.pdf》。

### 2.3.3.4 预设置 lpj 数值

LPJ 也就是 loops\_per\_jiffy，每次启动都会计算一次，但如果没有做修改的话，这个值每次启动算出来都是一样的，可以直接提供数值跳过计算。

如下 log 所示，有 skipped，lpj 由 timer 计算得来，不需要再校准 calibrate 了。

```
[ 0.019918] Calibrating delay loop (skipped), value calculated using timer frequency..  
48.00 BogoMIPS (lpj=240000)
```

如果没有 skipped，则可以在 cmdline 中添加 lpj=XXX 进行预设。

### 2.3.3.5 initcall 优化

在 cmdline 中设置 `initcall_debug=1`，即可打印跟踪所有内核初始化过程中调用 `initcall` 的顺序以及耗时。

具体修改 `env` 配置文件（路径见上文），新增一行 `initcall_debug=1`，并在 `setargs_*` 后加入 `initcall_debug=${initcall_debug}`，如下所示。

```
setargs_nand=setenv bootargs console=${console} console=tty0 root=${nand_root} init=${init}  
loglevel=${loglevel} partitions=${partitions} initcall_debug=${initcall_debug}
```

加入后，内核启动时就会有类似如下的打印，对于耗时较多的 `initcall`，可进行深入优化。

```
[ 0.021772] initcall sunxi_pinctrl_init+0x0/0x44 returned 0 after 9765 usecs  
[ 0.067694] initcall param_sysfs_init+0x0/0x198 returned 0 after 29296 usecs  
[ 0.070240] initcall genhd_device_init+0x0/0x88 returned 0 after 9765 usecs  
[ 0.080405] initcall init_scsi+0x0/0x90 returned 0 after 9765 usecs  
[ 0.090384] initcall mmc_init+0x0/0x84 returned 0 after 9765 usecs
```

### 2.3.3.6 内核 initcall module 并行

内核 `initcall` 有很多级别，其中启动中最耗时的就是各 `module` 的 `initcall`，针对多核方案，可以考虑将 `module initcall` 并行执行来节省时间。

目前内核 `do_initcalls` 是一个一个按照顺序来执行，可以修改成新建内核线程来执行。

注：当前 Tina 还未加入该优化。

### 2.3.3.7 减少 pty/tty 个数

加入 `initcall` 打印之后，部分平台发现 `pty/tty init` 耗时很多，可减少个数来缩短 `init` 时间。

```
initcall pty_init+0x0/0x3c4 returned 0 after 239627 usecs  
initcall chr_dev_init+0x0/0xdc returned 0 after 36581 usecs
```

### 2.3.3.8 内核 module

需要考虑启动速度的界定，对于内核 `module` 的优化主要有两点：

- 对于必须要加载的模块，直接编译进内核
- 对于不急需的功能，可以编译成模块。

比如某个应用，会开启主界面联网，启动速度以出现主界面为准，那么可以考虑将 disp 编入内核，wifi 编译成模块，后续需要时再动态加载。

### 2.3.3.9 Deferred Initcalls

介绍页面及 patch: [http://elinux.org/Deferred\\_Initcalls](http://elinux.org/Deferred_Initcalls)

打上这个 patch 之后，可以标记一些 initcall 为 Deferred\_Initcall。这些被标记的初始化函数，在系统启动的时候不会被调用

进入文件系统后，在合适的时间，比如启动主应用之后，再通过文件系统接口，启动这些推迟了的调用，彻底完成初始化。

## 2.3.4 rootfs 启动优化

rootfs 启动优化主要是优化 rootfs 的挂载到 init 进程执行。

### 2.3.4.1 initramfs

initramfs 是一个内存文件系统，会占用较多 DRAM。

部分产品可能会用到 initramfs 来过渡到 rootfs，其优化思路大体与 rootfs 类似。可参考本节后续的方案。

### 2.3.4.2 rootfs 类型以及压缩

存储介质、文件系统类型，压缩方式对 rootfs 挂载有很大影响。

此处给出某次测试结果供参考。实际优化的时候，需要重新测试，根据实际情况选择。

类型	压缩	介质	总时间/s
squashfs	gzip	emmc	0.12
squashfs	xz	emmc	0.27
squashfs	xz	nand	0.26
ext4	-	emmc	0.12

### 2.3.4.3 rootfs 裁剪

文件系统越小，加载速度越快。裁剪的主要思路是：删换压，即删除没有用到的，用小的换大的，选择合适的压缩方式。

### 2.3.4.4 指定文件系统类型

内核在挂载 rootfs 时，会有一个 try 文件系统类型的过程。可以在 cmdline 直接指定，节省时间。

具体是在 cmdline 中添加"`rootfstype=<type>`"，其中 type 为文件系统类型，如 ext4、squashfs 等。

### 2.3.4.5 静态创建 dev 节点

对于 dev 下面的节点，事先根据实际情况创建好，而不是在系统启动后动态生成，理论上也可以节省一定的时间。

### 2.3.4.6 rootfs 拆分

可以将 rootfs 拆分成两个部分，一个小的文件系统先挂载执行，大的文件系统根据需要动态挂载。

## 2.3.5 主应用程序启动优化

主应用程序主要是由客户开发，因此主导优化的还是客户，这里提一些优化措施：

- 提升运行顺序。将应用程序放在 init 很前面执行。
- 动态/静态链接。
- 编译选项。
- 暂时不使用的库采用 dlopen 方式。
- 应用程序拆分。



## 3 Tina 启动速度优化

Tina 中启动优化主要依靠宏 CONFIG\_BOOT\_TIME\_OPTIMIZATION 来完成，该宏会进行如下工作：

- 调整 Linux 内核镜像的压缩方式，调整 rootfs 的压缩方式。具体如何调整需要依据具体方案进行预先设定。
- 使 boot0、uboot、kernel 的打印不会输出到控制台。具体是在 scripts/pack\_img.sh 脚本中完成。
- uboot 加载内核时不进行校验。具体是在 scripts/pack\_img.sh 脚本中完成。
- 设置内核命令行参数 rootfstype，某些情况下会加快根文件系统的加载。具体是在 scripts/pack\_img.sh 脚本中完成。
- 部分方案会调整 kernel 镜像的加载地址。具体是在 scripts/pack\_img.sh 脚本中完成。

**注：**通过该宏预计可达到 70% 左右的优化效果，如还需优化，可参考第二章的内容。

### 3.1 开启 Tina 启动速度优化

在 tina 根目录下执行 make menuconfig 使能 CONFIG\_BOOT\_TIME\_OPTIMIZATION，具体如下所示

```
Tina Configuration
└─> Target Images
    └─>[*] Boot Time Optimization
```

```
Target Images
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in
[ ] excluded <M> module < > module capable

^(-)
*** Root filesystem images ***
[ ] ext4 ----
[ ] jffs2
[*] squashfs --->
*** Image Options ***
(3) Boot (SD Card) filesystem partition size (in MB)
[ ] For storage less than 32M, enable this when using ota
(7) Boot-Recovery initramfs filesystem partition size (in MB)
*** Kernel Image Compression Mode setting ***
[*] kernel compression mode setting --->
*** Downsize root filesystem ***
[*] downsize the root filesystem or initramfs
*** Optimize kernel size ***
[ ] downsize the kernel size
*** Boot Time Optimization ***
[*] Boot Time Optimization
```

图 3-1: Tina menuconfig

注：如果看不到该选项，使用？键搜索，会发现此项有一些依赖选项，使能依赖选项即可看到 Boot Time Optimization

## 3.2 实验结果

在某 norflash 方案上开启 CONFIG\_BOOT\_TIME\_OPTIMIZATION 后，启动速度提升效果如下：

- Linux 内核镜像压缩方式从 GZIP 换成 LZO，优化 > 0.2s。
- rootfs 从 squashfs XZ 压缩换成 squashfs GZIP 压缩，优化 > 0.15s。
- 屏蔽 boot0、uboot、kernel 启动阶段控制台打印，优化 > 2s。
- 取消内核加载时的校验，优化 0.3~0.4s。

注：对于不同的方案，由于 CPU 运算速度、存储器类型、内核压缩及尺寸、根文件系统类型及尺寸、主应用等的不同，优化结果会有一定差异，请以实际优化结果为准。

## 4 参考资料

---

[1] [https://elinux.org/Boot\\_Time](https://elinux.org/Boot_Time)

[2] [https://docs.blackfin.uclinux.org/doku.php?id=fast\\_boot\\_example](https://docs.blackfin.uclinux.org/doku.php?id=fast_boot_example)

[3] <https://github.com/tbird20d/grabserial>

[4] <http://www.bootchart.org>

[5] A Framework for Optimization of the Boot Time on Embedded Linux Environment with Raspberry Pi Platform






## 著作权声明

版权所有 © 2021 珠海全志科技股份有限公司。保留一切权利。

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

## 商标声明

、 **全志科技** （不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

## 免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。