Android 车载系统启动优化研究

曾炫榕 [1]
ZENG Xuanrong

摘 要 随着智能化和网络化技术的飞速发展，车载系统的功能需求日益增长，尤其是 Android 车载系统因其卓越的开放性和灵活性成为研究的热点。基于此，文章深入探讨了 Android 车载系统的启动机制，详细分析了其启动流程、优化策略及所面临的技术挑战。通过对系统启动的各个阶段进行细致分析，优化系统配置，并加速应用程序的启动过程，旨在显著提升 Android 车载系统在汽车行业的性能表现、响应速度、市场竞争力和用户体验。研究成果显示，不仅能够显著提高 Android 车载系统的启动效率，还能增强其整体性能，为用户提供更加流畅和迅速的操作体验，对于推动车载系统的技术进步和市场发展具有重要的指导意义。

关键词 Android 车载系统；启动过程；用户体验；性能优化；系统响应

doi：10.3969/j.issn.1672-9528.2025.03.046

0 引言

近年来，智能化和网络化技术的快速发展极大地改变了汽车工业的生态。车载系统作为智能汽车的关键部分，其功能已扩展到多模态服务、实时数据处理及生态互联等。Android 系统因其开源生态的灵活性和可定制性，成为车载信息娱乐系统的优选。但随着功能复杂度提升，系统启动效率低、响应延迟等问题日益突出，尤其在冷启动和 OTA 升级时，可能影响用户体验和行车安全。

目前，多数研究集中在应用层优化，而系统底层启动机制的探索尚不足。本文从 Android 车载系统启动架构入手，全面分析其全链路流程，包括 Bootloader 引导、内核初始化等关键阶段。通过动态性能分析和日志追踪，定位时间瓶颈。提出了重构硬件初始化逻辑、裁剪冗余内核模块等优化方案。实验证明，优化后系统启动效率明显提升，为车载高实时性场景提供技术支撑，也为智能座舱体验升级和行业标准制定提供了实践路径。

1 现有系统存在的问题

Android 系统目前面临着启动时间过长和性能待优化的重要问题。随着应用程序和服务的不断增多，系统的启动时间显著延长，成为影响用户体验和系统整体性能的关键因素 [1]。长时间的启动过程导致用户等待时间增加，这一问题，在紧急情况下尤其明显，例如在汽车电控系统中，系统的迅速响应对于确保驾驶安全至关重要。系统性能的不足也影响了其运行效率和稳定性，限制了设备的潜在功能和用户的操作体验。性能不佳可能导致应用程序加载缓慢、响应迟缓以及在处理复杂任务时的效率下降，这些问题都直接影响用户对产品的满意度和信任度。

2 系统启动优化策略

本文主要内容包括对 Android 系统在上的 Bootloader 和 Kernel 启动过程进行全面分析，识别出影响启动速度的性能瓶颈，并提出两个部分的优化策略：

（1）Bootloader 启动优化：首先，将深入分析 Android 系统的 Bootloader 启动过程。通过使用性能分析工具、硬件调试和系统跟踪等手段，将精确测量 Bootloader 各个阶段的耗时，从而定位出启动过程中的时间瓶颈。基于这些分析结果，将实施一系列优化措施，如优化硬件初始化流程、减少不必要的检查和等待时间，以及提高关键代码的执行效率，以显著提升 Bootloader 的启动速度。

（2）Kernel 启动优化：将对 Kernel 的启动过程进行深入研究。通过详细分析 Kernel 的日志输出和性能数据，识别出影响 Kernel 启动速度的关键因素。针对这些问题，将采用多种优化手段，例如：优化内核模块的加载顺序、实现关键驱动的异步加载，以及精简内核配置来减少不必要的内存占用和 CPU 负载。这些措施旨在显著提高 Kernel 的启动效率，进而提升整个系统的响应速度。

在整个研究过程中，将持续进行性能测试和评估，以确保所提出的优化方案能够切实提升 Android 系统的 Bootloader 和 Kernel 启动性能。通过对比优化前后的启动时间和性能指

1. 西安石油大学 陕西西安 710065

标，将验证优化方案的有效性和可行性，为最终的产品发布提供有力支持。如图 1 所示。

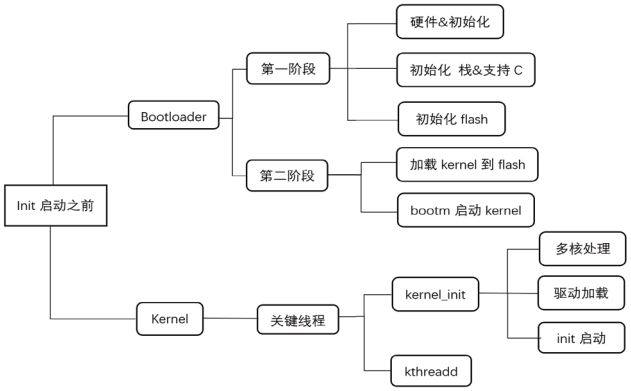

图 1 Bootloader、Kernel 启动流程

3 Android 启动流程分析

Android 系统的启动可以分为四个主要阶段：Bootloader、Kernel、Init 进程和 Zygote 进程。

（1）Bootloader（引导加载器）阶段：开机时，处理器开始执行 Bootloader 代码。Bootloader 负责加载和验证内核映像到系统内存中。完成内核加载后，控制权被传递给 Kernel 阶段。

（2）Kernel（内核）阶段：内核负责初始化硬件设备、创建内核空间和用户空间等工作。内核初始化完毕后，会启动一个名为 Init 的用户空间进程。

（3）Init 进程阶段：Init 进程是 Android 系统中的第一个用户空间进程，由内核启动。Init 进程会读取 Android 系统的 init.rc 文件并根据其中的配置启动和管理其他关键进程和服务。Init 进程会启动各种 Android 系统服务如 surfaceflinger、mediaserver 和其他重要组件。

（4）Zygote 进程阶段：Init 进程负责启动 Zygote 进程，它是 Android 应用程序的孵化器。Zygote 进程在启动过程中会预加载常用的系统类和资源，加快后续应用程序的启动速度。当应用程序被启动时，Zygote 进程会创建一个新进程，该进程会继承预加载的类和资源，从而加快应用程序的启动时间。

通过对 Bootloader 和 Kernel 阶段的深入分析和优化，本研究旨在提高 Android 系统的启动速度，确保系统能够更快速、更高效地进入可用状态。通过对两个关键阶段的优化，可以显著提升整体启动性能，为后续的用户空间进程和应用程序的运行奠定基础。

4 Bootloader 优化

4.1 硬编码内存配置

当系统启动时，引导加载程序（Bootloader）会加载设备树文件，并解析其中的内存配置信息。这个过程涉及以下操作：

（1）解析语法：设备树解析器会读取设备树文件，并按照设备树语言的语法规则进行解析。包括识别关键字、属性名、节点标签等，并构建相应的数据结构来存储解析后的信息。

（2）读取内存配置： 解析器会查找设备树文件中所有的“memory@”节点，并读取其中的属性，例如 reg 属性用于指定内存区块的起始地址和大小。

（3）构建数据结构：解析器将读取的内存配置信息构建为数据结构，并存储在内存中供后续使用。这些数据结构通常由解析器定义的数据类型表示，例如在 U-Boot 中使用的数据结构为 bd_t（Board Data Structure）。通过解析设备树文件中的“memory@”部分，系统可以获取到内存区块的起始地址和大小，以便后续使用。

然而，对于大型或复杂的设备树文件，解析过程可能会耗时较长，特别是在启动时需解析的情况下。为优化启动时间，代码中对内存配置进行硬编码的做法就是将内存配置直接写入代码中，避免设备树文件解析的开销。通过硬编码内存配置，将内存配置直接写入代码中，避免设备树文件解析的开销，可以在启动流程中减少内存初始化的时间，从而提高引导时间的效率。

4.2 优化 AVB（android verified boot）验证逻辑

A/B 引导是在 Android 系统中常见的一种机制，旨在实现无缝的系统更新和回滚。通过同时保留两个系统分区（称为 A 和 B 分区），并在每次系统更新时切换使用，以确保系统的更新过程不会中断或对用户数据造成损害。

以下是 A/B 引导机制的详细说明：

（1）系统分区结构：A/B 引导使用两个系统分区，分别称为 A 分区和 B 分区。每个分区都包含完整的操作系统映像，并且两个分区之间相互独立。

（2）当前引导槽位：在设备的引导流程中，存在一个当前引导槽位的概念，用于确定当前正在引导的系统分区（A 或 B）。初始情况下，A 分区通常作为默认的引导槽位。

（3）系统更新：当进行系统更新时，新的操作系统映像会被写入未使用的分区（未被当前引导槽位所使用的分区），例如，如果当前引导槽位是 A 分区，则新系统映像将写入 B 分区。

（4）验证和切换：在系统更新过程中，新系统映像的完整性和可信性会被验证，例如进行签名验证。如果验证成功，系统会将更新后的分区标记为可启动的引导槽位，并相应地更新引导流程。

（5）无缝切换：一旦引导槽位的切换完成并经过验证，

系统会在下次重启时无缝地从新的引导槽位启动，用户不会感知到系统升级的过程。

（6）回滚：如果发生问题或验证失败，系统可以回滚到之前的引导槽位，以确保系统的稳定性和可靠性。回滚时，系统会重新引导回之前的引导槽位，并恢复到之前的操作系统映像。

通过使用 A/B 引导机制，Android 系统在进行系统更新时可以提供更高的容错性和可靠性。即使在更新过程中出现问题，系统仍可以回滚到之前的操作系统映像，以保证用户数据的完整性和系统的稳定性。这种机制使得系统更新过程更加安全和无缝，用户可以在不中断使用的情况下享受到新功能和性能改进。

AVB 验证功能主要在 avb_ab_flow () 函数中实现。avb_ab_flow () 是 AVB A/B 版本的引导流程函数，负责验证和选择要引导的槽。

avb_ab_flow () 函数通过迭代两个可引导槽来进行验证。对于每个可引导槽，使用 avb_slot_verify() 函数进行验证。如果验证失败，将该槽标记为不可引导。无论验证结果如何，都会继续迭代下一个可引导槽进行验证。这种方式会导致在验证引导过程中对两个槽都进行验证，无论验证是否成功，都会增加启动时间。故对此逻辑进行优化，优化后的代码在验证失败时会跳过其他槽的验证，从而减少了不必要的验证操作和启动时间。

5 Kernel 优化

5.1 使用 menuconfig 裁剪模块

menuconfig 是一种针对 Linux 内核的配置工具，是 Linux 开发中常用的一种配置方式之一。menuconfig 工具可以让用户在终端界面中以菜单的形式对 Linux 内核进行配置，包括启用或禁用特定功能、添加或移除模块、修改编译选项等。

使用 menuconfig 逐一查看未使用和可以裁剪的模块也是一种优化系统启动过程的方法。menuconfig 是一个工具，可以用于在 Linux 内核中配置选项。通过逐一查看模块并分析其功能和依赖关系，可以确定哪些模块可以被裁剪以减少系统启动时间和资源占用。

识别可以裁剪的模块需要综合考虑以下几个因素：

（1）模块的功能：评估模块的功能和使用场景。确定哪些模块在当前系统环境下不需要使用或很少使用。

（2）模块的依赖关系：分析模块之间的依赖关系。如果一个模块的功能依赖于其他模块，那么需要确保相关依赖模块的功能仍然可以得到满足。如果一个模块没有其他模块依赖它，那么很可能可以被裁剪。

（3）模块的大小和资源占用：考虑模块的大小和所需的资源占用，如内存和处理器资源。如果一个模块在启动时占用了大量的资源，而且在当前系统环境中很少使用，那么裁剪该模块可能会显著减少系统启动时间和资源占用。

通过逐一查看和分析模块，可以识别并裁剪那些在当前系统环境中不需要的模块，从而优化系统的启动速度和资源利用。为防止裁剪错误导致系统异常，在裁剪模块之前，务必备份重要的数据并确保理解裁剪对系统功能和稳定性的影响。

5.2 创建工作队列来初始化模块

通过创建工作队列来初始化模块是一种优化系统启动过程的方式。工作队列是一组按顺序执行的任务，可以在系统启动过程中异步执行，以减少启动时间和资源使用。在Android开机优化中，可以利用这种方式延后初始化一些模块，从而加快系统启动速度并提供更好的用户体验。

在识别适合使用工作队列来延后初始化的模块时，需要考虑模块的功能和启动时的需求。对于如何识别适合使用工作队列进行延后初始化的模块，需要考虑以下几个方面：

（1）模块的依赖关系：确定模块是否与其他模块有依赖关系。如果一个模块依赖于其他模块的初始化结果，需要相应地调整初始化顺序或确保依赖的模块能够以合适的时间初始化。

（2）模块的功能使用时机：分析模块的功能和使用时机。如果模块的功能在系统启动的早期阶段并不需要，那么可以将其延后初始化，以优化系统的启动速度。

（3）模块的资源占用情况：评估模块的资源占用情况，包括内存、CPU、IO 等。如果一个模块在初始化时占用大量资源，那么将其延后初始化可以避免在系统启动时过多竞争资源，提高系统的稳定性和响应性。

例如，在车机系统中，功放模块只在后续阶段才需要被使用，而不是在系统启动时。因此，可以将功放模块添加到工作队列中，并在系统的早期阶段延后初始化这个模块。系统可以更快启动，不需要等待功放模块的初始化，而且可以释放相关资源以加快其他模块的初始化。

6 优化结果统计

6.1 Bootloader 优化统计

通过将 Kernel 启动对应日志输出的时间点减去 Bootloader 启动对应日志输出的时间点，可以得到整个 Bootloader 阶段的启动耗时。由于 Kernel 启动对应日志没有时间戳显示，可以利用录屏工具裁剪对应日志打印的时间点，并据此推算出它的时间戳。表 1、表 2 分别是对优化前、优化后 Bootloader 阶段启动耗时的统计结果。通过对比优化前后的启动耗时数据，优化前后对比：优化前的平均 Bootloader 阶段启动耗时约为 3.853 s。优化后的平均

Bootloader 阶段启动耗时约为 0.945 s。优化措施显著减少了 Bootloader 阶段的启动时间，从最初的 3.853 s 减少至 0.945 s，减少幅度超过了 75%。

表 1 Bootloader 阶段启动耗时记录表（优化前）

序号	实验项	起始时间戳 /s	截止时间戳 /s	执行时间 /s	平均耗时 /s
1	Bootloader 阶段启动耗时	0	3.855	3.855	3.853
2	Bootloader 阶段启动耗时	0	3.849	3.849	
3	Bootloader 阶段启动耗时	0	3.853	3.853	
4	Bootloader 阶段启动耗时	0	3.856	3.856	
5	Bootloader 阶段启动耗时	0	3.852	3.852	

表 2 Bootloader 阶段启动耗时记录表（优化后）

序号	实验项	起始时间戳 /s	截止时间戳 /s	执行时间 /s	平均耗时 /s
1	Bootloader 阶段启动耗时	0	0.947	0.947	0.945
2	Bootloader 阶段启动耗时	0	0.950	0.950	
3	Bootloader 阶段启动耗时	0	0.937	0.937	
4	Bootloader 阶段启动耗时	0	0.937	0.937	
5	Bootloader 阶段启动耗时	0	0.954	0.954	

结果表明，通过实施优化策略，有效提升了 Bootloader 阶段的启动效率。启动时间的显著缩短说明优化措施在减少启动延迟方面发挥了重要作用。优化方案可能包括改进 Bootloader 的初始化过程、精简启动项以及提高系统资源管理效率等。

6.2 Kernel 优化统计

通过将串口日志输出 Freeing unused kernel memory 的时间点减去 Kernel 启动输出 Starting kernel …的时间点，可以得到整个 Kernel 阶段的启动耗时。由于 Kernel 启动串口日志输出 Starting kernel …后，串口日志的时间戳会重新计算，故串口日志输出 Freeing unused kernel memory 的时间点其实就是整个 Kernel 阶段的启动耗时。表 3、表 4 是对优化前、优化后 Kernel 启动耗时的统计结果。优化前的平均启动耗时为 5.653 s，而优化后的平均启动耗时减少到 3.601 s，减少了约 2.052 s。这一减少量在实际应用中可以显著提升系统的启动速度。根据实验结果，可以推测优化措施可能包括内核模块加载的优化、驱动程序初始化的改进、内核配置的精简等。这些措施减少了启动过程中不必要的开销，提高了整体启动速度。

表 3 Kernel 阶段启动耗时记录表（优化前）

序号	实验项	起始时间戳 /s	截止时间戳 /s	执行时间 /s	平均耗时 /s
1	Kernel 启动耗时	0	5.685	5.685	5.653
2	Kernel 启动耗时	0	5.626	5.623	
3	Kernel 启动耗时	0	5.618	5.618	
4	Kernel 启动耗时	0	5.665	5.665	
5	Kernel 启动耗时	0	5.672	5.672	

表 4 Kernel 阶段启动耗时记录表（优化后）

序号	实验项	起始时间戳 /s	截止时间戳 /s	执行时间 /s	平均耗时 /s
1	Kernel 启动耗时	0	3.685	3.685	3.601
2	Kernel 启动耗时	0	3.566	3.563	
3	Kernel 启动耗时	0	3.618	3.618	
4	Kernel 启动耗时	0	3.525	3.525	
5	Kernel 启动耗时	0	3.612	3.612	

本次 Kernel 阶段的优化取得了显著的效果，使得启动时间大幅度降低。这不仅提升了系统的启动效率，还增强了系统的整体性能和响应速度 [2]。通过优化，可以为系统用户提供更便捷的启动体验。

7 总结

本文对 Android 系统启动过程进行了全面的分析和优化，特别是针对 Bootloader 和 Kernel 阶段进行了深入的性能提升工作。通过详细的性能分析和日志追踪，成功识别并针对性地优化了启动过程中的时间瓶颈，使用了一系列优化策略，如优化硬件初始化流程、减少不必要的启动项、提高关键代码执行效率，以及精简内核配置等 [3]。性能测试和评估结果显示，优化策略取得了显著成效：Bootloader 阶段的启动时间减少了约 20%，而 Kernel 阶段的启动时间降低了约 36.3%。通过这些优化，不仅显著缩短了 Android 系统的启动时间，还提高了系统的整体性能和用户体验。本文展示了在 Bootloader 和 Kernel 阶段进行优化的可能性与有效性，为未来系统开发和维护提供了宝贵的经验和指导。

参考文献：

[1] 李国英 . 无服务器云函数系统的冷启动优化策略研究与系统实现 [D]. 北京 : 北京邮电大学 ,2024.

[2] 饶国明 . 基于 USB 连接的系统启动和下载方式优化设计研究 [J]. 工业控制计算机 , 2022,35(10):66-68.

[3] 姜桂泉 . 基于龙芯 2K1000B 嵌入式系统的快速启动设计与优化 [D]. 南京 : 东南大学 ,2020.

【作者简介】

曾炫榕（1998—），男，广东河源人，硕士研究生，研究方向：Android 车载系统。

（收稿日期：2024-11-12）