# HarmonyOS 内核文档

**V1.0**

更多鸿蒙相关技术文章和资料，请关注 **HarmonyOS 社区**

鸿蒙学堂 hmxt.org 整理

2020 年 9 月 10 日

# 目 录

# 1 HarmonyOS 轻内核基础功能

## 1.1 进程

### 1.1.1 基本概念

从系统的角度看，进程是资源管理单元。进程可以使用或等待 CPU、使用内存空间等系统资源，并独立于其它进程运行。

HarmonyOS 内核的进程模块可以给用户提供多个进程，实现了进程之间的切换和通信，帮助用户管理业务程序流程。这样用户可以将更多的精力投入到业务功能的实现中。

HarmonyOS 内核中的进程采用抢占式调度机制，支持时间片轮转调度方式和 FIFO 调度机制。

HarmonyOS 内核的进程一共有 32 个优先级(0-31)，用户进程可配置的优先级有 22 个(10-31)，最高优先级为 10，最低优先级为 31。

高优先级的进程可抢占低优先级进程，低优先级进程必须在高优先级进程阻塞或结束后才能得到调度。

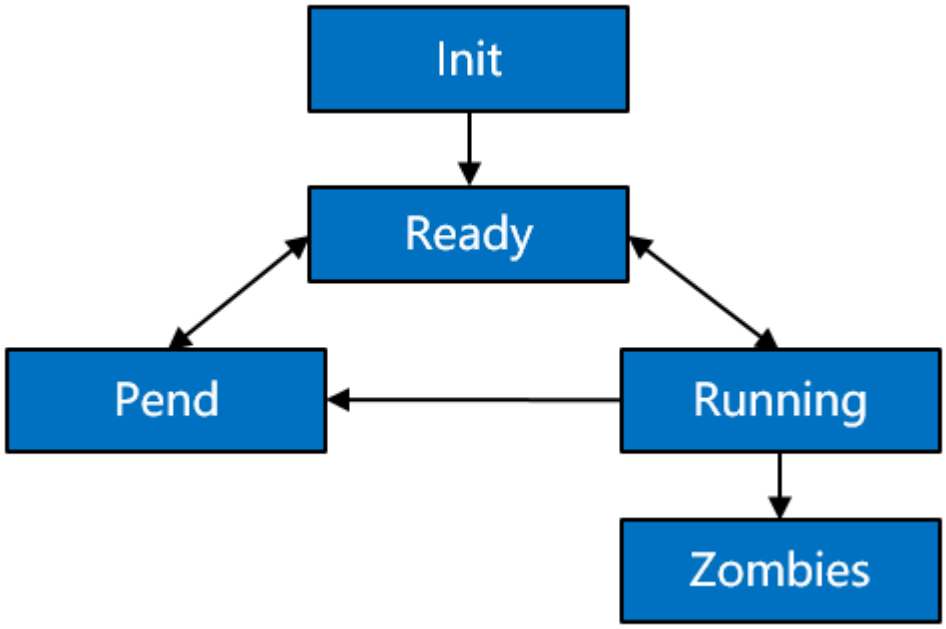
每一个用户态进程均拥有自己独立的进程空间，相互之间不可见，实现进程间隔离。

用户态根进程 Init 由内核态创建，其它用户态进程均由 Init 进程 fork 而来。

**进程状态说明：**

- 初始化（Init）：该进程正在被创建。
- 就绪（Ready）：该进程在就绪列表中，等待 CPU 调度。
- 运行（Running）：该进程正在运行。
- 阻塞（Pend）：该进程被阻塞挂起。本进程内所有的线程均被阻塞时，进程被阻塞挂起。
- 僵尸态（Zombies）：该进程运行结束，等待父进程回收其控制块资源。

**图 1** 进程状态迁移示意图




**进程状态迁移说明：**

- Init→Ready：

  进程创建或 fork 时，拿到该进程控制块后进入 Init 状态，处于进程初始化阶段，当进程初始化完成将进程插入调度队列，此时进程进入就绪状态。

- Ready→Running：

进程创建后进入就绪态，发生进程切换时，就绪列表中最高优先级的进程被执行，从而进入运行态。若此时该进程中已无其它线程处于就绪态，则该进程从就绪列表删除，只处于运行态；若此时该进程中还有其它线程处于就绪态，则该进程依旧在就绪队列，此时进程的就绪态和运行态共存。

- Running→Pend：

  进程内所有的线程均处于阻塞态时，进程在最后一个线程转为阻塞态时，同步进入阻塞态，然后发生进程切换。

- Pend→Ready / Pend→Running：

  阻塞进程内的任意线程恢复就绪态时，进程被加入到就绪队列，同步转为就绪态，若此时发生进程切换，则进程状态由就绪态转为运行态。

- Ready→Pend：

  进程内的最后一个就绪态线程处于阻塞态时，进程从就绪列表中删除，进程由就绪态转为阻塞态。

- Running→Ready：

  进程由运行态转为就绪态的情况有以下两种：

1. 有更高优先级的进程创建或者恢复后，会发生进程调度，此刻就绪列表中最高优先级进程变为运行态，那么原先运行的进程由运行态变为就绪态。
2. 若进程的调度策略为 SCHED_RR，且存在同一优先级的另一个进程处于就绪态，则该进程的时间片消耗光之后，该进程由运行态转为就绪态，另一个同优先级的进程由就绪态转为运行态。

- Running→Zombies：

当进程的主线程或所有线程运行结束后，进程由运行态转为僵尸态，等待父进程回收资源。

## 1.1.2 使用场景

进程创建后，用户只能操作自己进程空间的资源，无法操作其它进程的资源（共享资源除外）。 用户态允许进程挂起，恢复，延时等操作，同时也可以设置用户态进程调度优先级和调度策略，获取进程调度优先级和调度策略。进程结束的时候，进程会主动释放持有的进程资源，但持有的进程 pid 资源需要父进程通过 wait/waitpid 或父进程退出时回收。

## 1.1.3 功能

HarmonyOS 内核系统中的进程管理模块为用户提供下面几种功能：

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| 进程 | fork | 创建一个新进程。 | - |
| | exit | 终止进程。 | - |
| | atexit | 注册正常进程终止的回调函数。 | - |

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| | abort | 中止进程执行。 | - |
| | getpid | 获取进程 ID。 | - |
| | getppid | 获取父进程 ID。 | - |
| | getpgrp | 获取调用进程的进程组 ID。 | - |
| | getpgid | 获取进程的进程组 ID。 | - |
| | setpgrp | 设置调用进程的进程组 ID。 | - |
| | setpgid | 设置进程的进程组 ID。 | - |
| | kill | 给进程发送信号。 | 仅支持 1-30 号信号的发送。 |

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| | | | 信号的默认行为不支持 STOP 及 CONTINUE，无 COREDUMP 功能。<br>不能屏蔽 SIGSTOP、SIGKILL、SIGCONT。<br>异步信号，发送信号给某进程后，直到该进程被调度后才会执行信号回调（为安全起见，杀死进程的动作是进程自己执行的，内核不能通过信号强制杀死对方）。<br>进程消亡会发送 SIGCHLD 给父进程，发送动作无法取消。<br>无法通过信号唤醒正在睡眠的进程。 |

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| | wait | 等待任意子进程结束并回收子进程资源。 | status 的值可以由以下宏定义解析：<br>WIFEXITED(status):如果子进程正常结束，它就返回真；否则返回假。<br>WEXITSTATUS(status):如果 WIFEXITED(status)为真，则可以用该宏取得子进程 exit()返回的退出码。<br>WTERMSIG(status) 仅支持以下情况：子进程触发异常结束后通过 WTERMSIG 获取的进程退出编号始终为 SIGUSR2。 |

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| | | | 不支持的操作：WIFSTOPPED、WSTOPSIG、WCOREDUMP 、WIFCONTINUED。 |
| | waitpid | 等待子进程结束并回收子进程资源。 | options：不支持WUNTRACED，WCONTINUED；<br>status 的值可以由以下宏定义解析：<br>WIFEXITED(status):如果子进程正常结束，它就返回真；否则返回假。<br>WEXITSTATUS(status):如果WIFEXITED(status)为真，则可以用该宏取得 |

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| | | | 子进程 exit()返回的退出码。<br>WTERMSIG(status)仅支持以下情况：子进程触发异常结束后通过 WTERMSIG 获取的进程退出编号始终为 SIGUSR2。<br>不支持：WIFSTOPPED 、WSTOPSIG、WCOREDUMP 、WIFCONTINUED。 |
| 调度 | getpriority | 获取指定 ID 的静态优先级。 | 不支持：PRIO_PGRP、PRIO_USER。<br>无动态优先级概念，用于设置静态优先级。 |
| | setpriority | 设置指定 ID 的静态优先级。 | |

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| | sched_rr_get_interval | 获取执行时间限制。 | - |
| | sched_yield | 系统调用运行进程主动让出执行权。 | - |
| | sched_get_priority_max | 获取进程静态优先级取值范围的最大值。 | 调度策略只支持：SCHED_FIFO 、SCHED_RR。 |
| | sched_get_priority_min | 获取进程静态优先级取值范围的最小值。 | |
| | sched_getscheduler | 获取调度策略。 | |
| | sched_setscheduler | 设置调度策略。 | |
| | sched_getparam | 获取调度参数。 | - |
| | sched_setparam | 设置调度参数。 | - |

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| exec | execl | 执行指定的 elf 格式的用户程序文件。 | - |
| | execle | 执行指定的 elf 格式的用户程序文件。 | - |
| | execlp | 执行指定的 elf 格式的用户程序文件。 | - |
| | execv | 执行指定的 elf 格式的用户程序文件。 | - |
| | execve | 执行指定的 elf 格式的用户程序文件。 | - |

| 功能分类 | 接口名 | 描述 | 备注 |
| --- | --- | --- | --- |
| | execvp | 执行指定的 elf 格式的用户程序文件。 | - |

**表 1** 进程管理模块功能

# 1.2 线程

### 1.2.1.1 基本概念

从系统的角度看，线程是竞争系统资源的最小运行单元。线程可以使用或等待CPU、使用内存空间等系统资源，并独立于其它线程运行。

HarmonyOS 内核每个进程内的线程独立运行、独立调度，当前进程内线程的调度不受其它进程内线程的影响。

HarmonyOS 内核中的线程采用抢占式调度机制，同时支持时间片轮转调度和FIFO 调度方式。

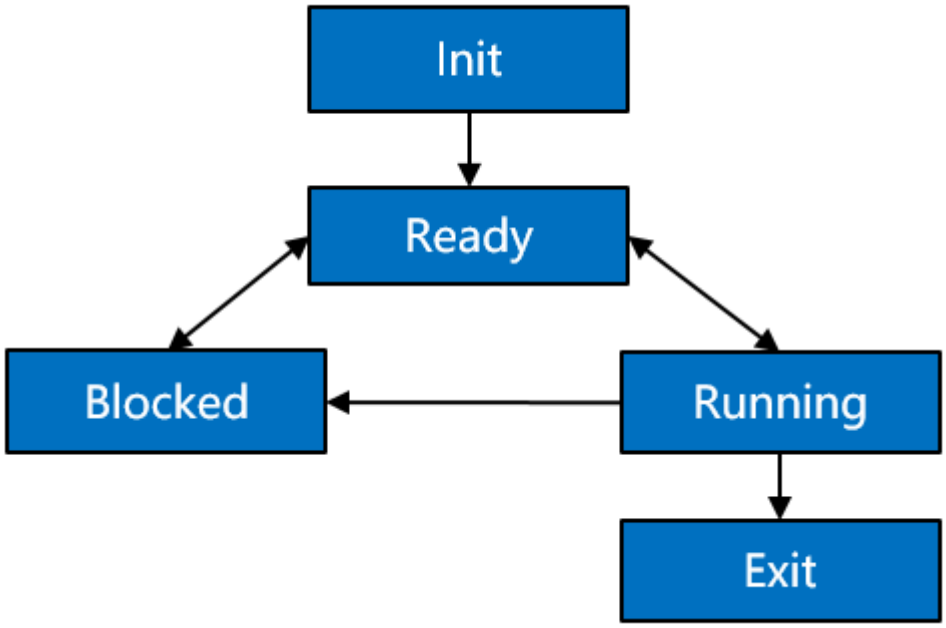
HarmonyOS 内核的线程一共有 32 个优先级(0-31)，最高优先级为 0，最低优先级为 31。

当前进程内高优先级的线程可抢占当前进程内低优先级线程，当前进程内低优先级线程必须在当前进程内高优先级线程阻塞或结束后才能得到调度。

**线程状态说明：**

- 初始化（Init）：该线程正在被创建。
- 就绪（Ready）：该线程在就绪列表中，等待 CPU 调度。
- 运行（Running）：该线程正在运行。
- 阻塞（Blocked）：该线程被阻塞挂起。Blocked 状态包括：pend(因为锁、事件、信号量等阻塞)、suspend（主动 pend）、delay(延时阻塞)、pendtime(因为锁、事件、信号量时间等超时等待)。
- 退出（Exit）：该线程运行结束，等待父线程回收其控制块资源。

图 1 线程状态迁移示意图




**线程状态迁移说明：**

- Init→Ready：

  线程创建拿到控制块后为 Init 状态，处于线程初始化阶段，当线程初始化完成将线程插入调度队列，此时线程进入就绪状态。

- Ready→Running：

  线程创建后进入就绪态，发生线程切换时，就绪列表中最高优先级的线程被执行，从而进入运行态，但此刻该线程会从就绪列表中删除。

- Running→Blocked：

  正在运行的线程发生阻塞（挂起、延时、读信号量等）时，该线程会从就绪列表中删除，线程状态由运行态变成阻塞态，然后发生线程切换，运行就绪列表中剩余最高优先级线程。

- Blocked→Ready / Blocked→Running：

  阻塞的线程被恢复后（线程恢复、延时时间超时、读信号量超时或读到信号量等），此时被恢复的线程会被加入就绪列表，从而由阻塞态变成就绪态；此时如果被恢复线程的优先级高于正在运行线程的优先级，则会发生线程切换，将该线程由就绪态变成运行态。

- Ready→Blocked：

  线程也有可能在就绪态时被阻塞（挂起），此时线程状态会由就绪态转变为阻塞态，该线程从就绪列表中删除，不会参与线程调度，直到该线程被恢复。

- Running→Ready：

  有更高优先级线程创建或者恢复后，会发生线程调度，此刻就绪列表中最高优先级线程变为运行态，那么原先运行的线程由运行态变为就绪态，并加入就绪列表中。

- Running→Exit：

运行中的线程运行结束，线程状态由运行态变为退出态。若未设置分离属性（PTHREAD_CREATE_DETACHED）的线程，运行结束后对外呈现的是 Exit 状态，即退出态。

- Blocked→Exit：

  阻塞的线程调用删除接口，线程状态由阻塞态变为退出态。

## 1.2.2 使用场景

线程创建后，用户态可以执行线程调度、挂起、恢复、延时等操作，同时也可以设置线程优先级和调度策略，获取线程优先级和调度策略。

## 1.2.3 功能

HarmonyOS 内核系统中的线程管理模块，线程间通信为用户提供下面几种功能：

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| pthread.h | pthread_attr_destroy | 销毁线程属性对象。 | - |
| pthread.h | pthread_attr_getinheritsched | 获取线程属性对象的调度属性。 | - |
| pthread.h | pthread_attr_getschedparam | 获取线程属性对象的调度参数属性。 | - |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| pthread.h | pthread_attr_getschedpolicy | 获取线程属性对象的调度策略属性。 | HarmonyOS：支持SCHED_FIFO、SCHED_RR调度策略。 |
| pthread.h | pthread_attr_getstacksize | 获取线程属性对象的堆栈大小。 | - |
| pthread.h | pthread_attr_init | 初始化线程属性对象。 | - |
| pthread.h | pthread_attr_setdetachstate | 设置线程属性对象的分离状态。 | - |
| pthread.h | pthread_attr_setinheritsched | 设置线程属性对象的继承调度属性。 | - |
| pthread.h | pthread_attr_setschedparam | 设置线程属性对象的调度参数属性。 | HarmonyOS：设置线程优先级的参数值越小，线程在系统中的 |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| | | | 优先级越高；设置参数值越大，优先级越低。<br>注意：需要将 pthread_attr_t 线程属性的 inheritsched 字段设置为 PTHREAD_EXPLICIT_SCHED，否则设置的线程调度优先级将不会生效，系统默认设置为 PTHREAD_INHERIT_SCHED。 |
| pthread.h | pthread_attr_setschedpolicy | 设置线程属性对象的调度策略属性。 | HarmonyOS：支持 SCHED_FIFO |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| | | | 、SCHED_RR 调度策略。 |
| pthread.h | pthread_attr_setstacksize | 设置线程属性对象的堆栈大小。 | - |
| pthread.h | pthread_getattr_np | 获取已创建线程的属性。 | - |
| pthread.h | pthread_cancel | 向线程发送取消请求。 | - |
| pthread.h | pthread_testcancel | 请求交付任何未决的取请求。 | - |
| pthread.h | pthread_setcanceltype | 设置线程可取消类型。 | - |
| pthread.h | pthread_setcancelstate | 设置线程可取消状态。 | - |
| pthread.h | pthread_create | 创建一个新的线程。 | - |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| pthread.h | pthread_detach | 分离一个线程。 | - |
| pthread.h | pthread_equal | 比较两个线程 ID 是否相等。 | - |
| pthread.h | pthread_exit | 终止正在调用的线程。 | - |
| pthread.h | pthread_getschedparam | 获取线程的调度策略和参数。 | HarmonyOS：支持 SCHED_FIFO 、SCHED_RR 调度策略。 |
| pthread.h | pthread_join | 等待指定的线程结束。 | - |
| pthread.h | pthread_self | 获取当前线程的 ID。 | - |
| pthread.h | pthread_setschedprio | 设置线程的调度静态优先级。 | - |
| pthread.h | pthread_kill | 向线程发送信号。 | - |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| pthread.h | pthread_once | 使函数调用只能执行一次。 | - |
| pthread.h | pthread_atfork | 注册 fork 的处理程序。 | - |
| pthread.h | pthread_cleanup_pop | 删除位于清理处理程序堆栈顶部的例程。 | - |
| pthread.h | pthread_cleanup_push | 将例程推送到清理处理程序堆栈的顶部。 | - |
| pthread.h | pthread_barrier_destroy | 销毁屏障对象（高级实时线程） | - |
| pthread.h | pthread_barrier_init | 初始化屏障对象（高级实时线程） | - |
| pthread.h | pthread_barrier_wait | 屏障同步（高级实时线程） | - |
| pthread.h | pthread_barrierattr_destroy | 销毁屏障属性对象。 | - |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| pthread.h | pthread_barrierattr_init | 初始化屏障属性对象。 | - |
| pthread.h | pthread_mutex_destroy | 销毁互斥锁。 | - |
| pthread.h | pthread_mutex_init | 初始化互斥锁。 | - |
| pthread.h | pthread_mutex_lock | 互斥锁加锁操作。 | - |
| pthread.h | pthread_mutex_trylock | 互斥锁尝试加锁操作。 | - |
| pthread.h | pthread_mutex_unlock | 互斥锁解锁操作。 | - |
| pthread.h | pthread_mutexattr_destroy | 销毁互斥锁属性对象。 | - |
| pthread.h | pthread_mutexattr_gettype | 获取互斥锁类型属性。 | - |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| pthread.h | pthread_mutexattr_init | 初始化互斥锁属性对象。 | - |
| pthread.h | pthread_mutexattr_settype | 设置互斥锁类型属性。 | - |
| pthread.h | pthread_mutex_timedlock | 使用超时锁定互斥锁。 | - |
| pthread.h | pthread_rwlock_destroy | 销毁读写锁。 | - |
| pthread.h | pthread_rwlock_init | 初始化读写锁。 | - |
| pthread.h | pthread_rwlock_rdlock | 获取读写锁读锁操作。 | - |
| pthread.h | pthread_rwlock_timedrdlock | 使用超时锁定读写锁读锁。 | - |
| pthread.h | pthread_rwlock_timedwrlock | 使用超时锁定读写锁写锁。 | - |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| pthread.h | pthread_rwlock_tryrdlock | 尝试获取读写锁读锁操作。 | - |
| pthread.h | pthread_rwlock_trywrlock | 尝试获取读写锁写锁操作。 | - |
| pthread.h | pthread_rwlock_unlock | 读写锁解锁操作。 | - |
| pthread.h | pthread_rwlock_wrlock | 获取读写锁写锁操作。 | - |
| pthread.h | pthread_rwlockattr_destroy | 销毁读写锁属性对象。 | - |
| pthread.h | pthread_rwlockattr_init | 初始化读写锁属性对象。 | - |
| pthread.h | pthread_cond_broadcast | 解除若干已被等待条件阻塞的线程。 | - |
| pthread.h | pthread_cond_destroy | 销毁条件变量。 | - |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| pthread.h | pthread_cond_init | 初始化条件变量。 | - |
| pthread.h | pthread_cond_signal | 解除被阻塞的线程。 | - |
| pthread.h | pthread_cond_timedwait | 定时等待条件。 | - |
| pthread.h | pthread_cond_wait | 等待条件。 | - |
| semaphore.h | sem_destroy | 销毁指定的无名信号量。 | - |
| semaphore.h | sem_getvalue | 获得指定信号量计数值。 | - |
| semaphore.h | sem_init | 创建并初始化一个无名信号量。 | - |
| semaphore.h | sem_post | 增加信号量计数。 | - |

| 头文件 | 名称 | 说明 | 备注 |
| --- | --- | --- | --- |
| semaphore.h | sem_timedwait | 获取信号量，且有超时返回功能。 | - |
| semaphore.h | sem_trywait | 尝试获取信号量。 | - |
| semaphore.h | sem_wait | 获取信号量。 | - |

**表 1** 线程管理模块功能

# 1.3 内存

## 1.3.1 基本概念

内存管理是开发过程中必须要关注的重要过程，它包括内存的分配、使用和回收。

良好的内存管理对于提高软件性能和可靠性有着十分重要的意义。

## 1.3.2 使用场景

针对用户态开发，HarmonyOS 内存提供了一套内存系统调用接口，支持内存的申请释放、重映射、内存属性的设置等，还有 C 库的标准内存操作函数。

## 1.3.3 功能

| 头文件 | 接口 | 功能 |
| --- | --- | --- |
| strings.h | int bcmp(const void *s1, const void *s2, size_t n) | 比较字节序列。 |
| strings.h | void bcopy(const void *src, void *dest, size_t n) | 拷贝字节序列。 |
| strings.h | void bzero(void *s, size_t n) | 写入零值字节。 |
| string.h | void *memccpy(void *dest, const void *src, int c, size_t n) | 拷贝 src 所指的内存内容前 n 个字节到 dest 所指的地址上。复制时检查参数 c 是否出现，若是则返回 dest 中值为 c 的下一个字节地址。 |
| string.h | void *memchr(const void *s, int c, size_t n) | 在 s 所指内存的前 n 个字节中查找 c。 |

| 头文件 | 接口 | 功能 |
| --- | --- | --- |
| string.h | int memcmp(const void *s1, const void *s2, size_t n) | 内存比较。 |
| string.h | void *memcpy(void *dest, const void *src, size_t n) | 内存拷贝。 |
| string.h | void *memmem(const void *haystack, size_t haystacklen, const void *needle, size_t needlelen) | 找到一个子串。 |
| string.h | void *memmove(void *dest, const void *src, size_t n) | 内存移动。 |
| string.h | void *mempcpy(void *dest, const void *src, size_t n) | 拷贝内存区域。 |
| string.h | void *memset(void *s, int c, size_t n) | 内存初始化。 |
| stdlib.h | void *malloc(size_t size) | 申请内存。 |

| 头文件 | 接口 | 功能 |
| --- | --- | --- |
| stdlib.h | void *calloc(size_t nmemb, size_t size) | 申请内存并清零。 |
| stdlib.h | void *realloc(void *ptr, size_t size) | 重分配内存。 |
| stdlib.h/malloc. | void *valloc(size_t size) | 分配以页对齐的内存。 |
| stdlib.h | void free(void *ptr) | 释放内存。 |
| malloc.h | size_t malloc_usable_size(void *ptr) | 获取从堆分配的内存块的大小。 |
| unistd.h | int getpagesize(void) | 获取页面大小。 |
| unistd.h | void *sbrk(intptr_t increment) | 更改数据段大小。 |

**表 1** 标准 C 库相关接口

差异接口详细说明：

- **mmap**

  **函数原型：**

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

**函数功能：**申请虚拟内存。

**参数说明：**

| 参数 | 描述 |
| --- | --- |
| addr | 用来请求使用某个特定的虚拟内存地址。如果取 NULL，结果地址就将自动分配（这是推荐的做法），否则会降低程序的可移植性，因为不同系统的可用地址范围不一样。 |
| length | 内存段的大小。 |
| prot | 用于设置内存段的访问权限，有如下权限：<br>PROT_READ：允许读该内存段。<br>PROT_WRITE：允许写该内存段。<br>PROT_EXEC：允许执行该内存段。<br>PROT_NONE：不能访问。 |
| flags | 控制程序对内存段的改变所造成的影响，有如下属性：<br>MAP_PRIVATE：内存段私有，对它的修改值仅对本进程有效。<br>MAP_SHARED：把对该内存段的修改保存到磁盘文件中。 |

| 参数 | 描述 |
| --- | --- |
| fd | 打开的文件描述符。 |
| offset | 用以改变经共享内存段访问的文件中数据的起始偏移值。 |

**说明**

mmap 与 Linux 实现差异详见与 Linux 标准库的差异章节。

**返回值：**

- 成功返回：虚拟内存地址，这地址是页对齐。
- 失败返回：(void *)-1。

- **munmap 接口**

  **函数原型：**

  int munmap(void *addr, size_t length);

  **函数功能：**释放虚拟内存。

  **参数说明：**

| 参数 | 描述 |
| --- | --- |
| addr | 虚拟内存起始位置。 |
| length | 内存段的大小。 |

  **返回值：**

- 成功返回 0。

- 失败返回-1。

- **mprotect 接口**

  **函数原型：**

  int mprotect(void *addr, size_t length, int prot);

  **函数功能：**修改内存段的访问权限。

  **参数说明：**

| 参数 | 描述 |
| --- | --- |
| addr | 内存段起始地址，必须页对齐；访问权限异常，内核将直接抛异常，kill 该进程，而不会产生 SIGSEGV 信号给当前进程。 |
| length | 内存段的大小。 |
| prot | 内存段的访问权限，有如下定义：<br>PROT_READ：允许读该内存段。<br>PROT_WRITE：允许写该内存段。<br>PROT_EXEC：允许执行该内存段。<br>PROT_NONE：不能访问。 |

  **返回值：**

- 成功返回 0。

- 失败返回-1。

- **mremap 接口**

**函数原型：**

void *mremap(void *old_address, size_t old_size, size_t new_size, int flags, void new_address);

**函数功能：**重新映射虚拟内存地址。

**参数说明：**

| 参数 | 描述 |
| --- | --- |
| old_address | 需要扩大（或缩小）的内存段的原始地址。注意 old_address 必须是页对齐。 |
| old_size | 内存段的原始大小。 |
| new_size | 新内存段的大小。 |
| flags | 如果没有足够的空间在当前位置展开映射，则返回失败<br>MREMAP_MAYMOVE：允许内核将映射重定位到新的虚拟地址。<br>MREMAP_FIXED：mremap()接受第五个参数，void *new_address，该参数指定映射地址必须页对齐；在 new_address 和 new_size 指定的地址范围内的所有先前映射都被解除映射。如果指定了 MREMAP_FIXED，还必须指定 MREMAP_MAYMOVE。 |

**返回值：**

- 成功返回：重新映射后的虚拟内存地址。

- 失败返回：((void *)-1)。

# 1.4 网络

## 1.4.1 基本概念

网络模块实现了 TCP/IP 协议栈基本功能，提供标准的 POSIX socket 接口。

**说明**

当前系统使用 **lwIP** 提供网络能力。

## 1.4.2 使用场景

针对用户态开发，HarmonyOS 内核提供了一套网络功能系统调用接口，支持 socket 的创建关闭、数据收发、网络属性的设置等，通过 C 库提供标准的 POSIX socket 函数供开发者使用。

## 1.4.3 功能

| 头文件 | 接口 | 功能 |
| --- | --- | --- |
| sys/socket.h | int accept(int socket, struct sockaddr *address, socklen_t *address_len) | 接受连接。 |
| sys/socket.h | int bind(int s, const struct sockaddr *name, socklen_t namelen) | socket 与 IP 地址绑定。 |

| 头文件 | 接口 | 功能 |
| --- | --- | --- |
| sys/socket.h | int shutdown(int socket, int how) | 关闭连接。 |
| sys/socket.h | int getpeername(int s, struct sockaddr *name, socklen_t *namelen) | 获取对端地址。 |
| sys/socket.h | int getsockname(int s, struct sockaddr *name, socklen_t *namelen) | 获取本地地址。 |
| sys/socket.h | int getsockopt(int s, struct sockaddr *name, socklen_t *namelen) | 获取 socket 属性信息。 |
| sys/socket.h | int setsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen) | 配置 socket 属性。 |
| unistd.h | int close(int s) | 关闭 socket。 |
| sys/socket.h | int connect(int s, const struct sockaddr *name, socklen_t namelen) | 连接到指定的目的 IP。 |
| sys/socket.h | int listen(int sockfd, int backlog) | listen 连接本 socket 的请求。 |

| 头文件 | 接口 | 功能 |
| --- | --- | --- |
| sys/socket.h | ssize_t recv(int socket, void *buffer, size_t length, int flags) | 接收 socket 上收到的数据。 |
| sys/socket.h | ssize_t recvmsg(int s, struct msghdr *message, int flags) | 接收 socket 上收到的数据，可使用更丰富的参数。 |
| sys/socket.h | ssize_t recvfrom(int socket, void *buffer, size_t length, int flags, struct sockaddr *address, socklen_t *address_len) | 接收 socket 上收到的数据，可同时获得数据来源 IP 地址。 |
| sys/socket.h | ssize_t send(int s, const void *dataptr, size_t size, int flags) | 通过 socket 发送数据。 |
| sys/socket.h | ssize_t sendmsg(int s, const struct msghdr *message, int flags) | 通过 socket 发送数据，可使用更丰富的参数。 |
| sys/socket.h | ssize_t sendto(int s, const void *dataptr, size_t size, int flags, const struct sockaddr *to, socklen_t tolen) | 通过 socket 发送数据，可指定发送的目的 IP 地址。 |

| 头文件 | 接口 | 功能 |
| --- | --- | --- |
| sys/socket.h | int socket(int domain, int type, int protocol) | 创建 socket。 |
| sys/select.h | int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout) | 多路复用。 |
| sys/ioctl.h | int ioctl(int s, int request, ...) | socket 属性获取、设置。 |
| arpa/inet.h | const char *inet_ntop(int af, const void *src, char *dst, socklen_t size) | 网络地址格式转换：将二进制格式 IP 地址转换为字符串格式。 |
| arpa/inet.h | int inet_pton(int af, const char *src, void *dst) | 网络地址格式转换：将字符串格式 IP 地址转换为二进制格式。 |

**表 1** 标准 C 库相关接口

与标准接口差异详细说明：

- **sendmsg**

  **函数原型：**

  ssize_t sendmsg(int s, const struct msghdr *message, int flags)

  **函数功能：**发送消息。

  **参数说明：**

| 参数 | 描述 |
| --- | --- |
| s | 套接字。 |
| message | 待发送的消息，不支持发送 ancillary 消息。 |
| flags | 用于指定发送消息时行为特性，有如下行为特性：<br>MSG_MORE：允许将多次发送的消息进行拼包发送。<br>MSG_DONTWAIT：非阻塞操作。 |

  **返回值：**

  - 成功返回：已发送的消息长度（字节数）。
  - 失败返回：-1，并设置 errno。

- **recvmsg**

  **函数原型：**

  ssize_t recvmsg(int s, struct msghdr *message, int flags)

  **函数功能：**接收消息。

**参数说明：**

| 参数 | 描述 |
| --- | --- |
| s | 套接字。 |
| message | 存放接收的消息，不支持接收 ancillary 消息。 |
| flags | 用于指定接收消息时行为特性，有如下行为特性：<br>MSG_PEEK：允许预读消息而不取走。<br>MSG_DONTWAIT：非阻塞操作。 |

**返回值：**

- 成功返回：已接收的消息长度（字节数）。
- 失败返回：-1，并设置 errno。

- **ioctl**

  **函数原型：**

  int ioctl(int s, int request, ...)

  **函数功能：**获取或设置 socket 属性。

  **参数说明：**

| 参数 | 描述 |
| --- | --- |
| s | 套接字 |

| 参数 | 描述 |
| --- | --- |
| request | 对 socket 属性要进行的操作，当前支持如下操作：<br>FIONREAD：获取 socket 当前可读取的数据大小（字节数）。<br>FIONBIO：设置 socket 是否非阻塞。 |

**返回值：**

- 成功返回：0。
- 失败返回：-1，并设置 errno。

# 2 HarmonyOS 轻内核文件系统

## 2.1 VFS

### 2.1.1 概述

#### 2.1.1.1 基本概念

VFS 是 Virtual File System（虚拟文件系统）的缩写，它不是一个实际的文件系统，而是一个异构文件系统之上的软件粘合层，为用户提供统一的类 Unix 文件操作接口。

由于不同类型的文件系统接口不统一，若系统中有多个文件系统类型，访问不同的文件系统就需要使用不同的非标准接口。而通过在系统中添加 VFS 层，提

供统一的抽象接口，屏蔽了底层异构类型的文件系统的差异，使得访问文件系统的系统调用不用关心底层的存储介质和文件系统类型，提高开发效率。VFS 和各个具体文件系统的关系如下：

**图 1** VFS 和各个文件系统的关系

HarmonyOS 内核中，VFS 框架是通过在内存中的树结构来实现的，树的每个结点都是一个 inode 结构体。设备注册和文件系统挂载后会根据路径在树中生成相应的结点。VFS 最主要是两个功能：

- 查找节点。
- 统一调用（标准）。

#### 2.1.1.2 运作机制

通过 VFS 层，可以使用标准的 Unix 文件操作函数（如 open、read、write 等）来实现对不同介质上不同文件系统的访问。

VFS 框架内存中的 inode 树结点有三种类型：

- 虚拟结点：作为 VFS 框架的虚拟文件，保持树的连续性，如/usr、/usr/bin。
- 设备结点：/dev 目录下，对应一个设备，如/dev/mmc0blk0。
- 挂载点：挂载具体文件系统，如/vs/sd、/mnt。

inode 的关键在于 u 和 i_private 字段，一个是函数方法结构体的指针，一个是数据指针。

**图 2** 文件系统树形结构

## 2.1.2 注意事项

- VFS 下的所有文件系统，创建的目录名和文件名最多只可以有 255 个字节，能支持的全路径长度最长为 259 字节，超过这个路径长度的文件和目录无法创建。
- 目前仅有 jffs2 文件系统支持完整的权限控制。
- inode_find()函数调用后会使查找到的 inode 节点连接数+1，调用完成后需要调用 inode_release()使连接数-1，所以一般 inode_find()要和 inode_release()配套使用。
- 设备分为字符设备和块设备，为了块设备上的文件系统系统数据安全，需挂载相应文件系统后通过文件系统接口操作数据。
- los_vfs_init()只能调用一次，多次调用将会造成文件系统异常。
- 目前 HarmonyOS 内核所有的文件系统中的文件名和目录名中只可以出现“-” 与“_”两种特殊字符，使用其他特殊字符可能造成的后果不可预知，请谨慎为之。
- HarmonyOS 内核支持 open()+O_DIRECTORY 的方法获取目录数据信息。
- 挂载点必须为空目录，不能重复挂载至同一挂载点或挂载至其他挂载点下的目录或文件，错误挂载可能损坏设备及系统。

- open 打开一个文件时，参数 O_RDWR、O_WRONLY、O_RDONLY 互斥，只能出现一个，若出现 2 个或以上作为 open 的参数，文件读写操作会被拒绝，并返回 EACCESS 错误码，禁止使用。
- HarmonyOS 内核文件系统在 umount 操作之前，需确保所有目录及文件全部关闭，否则 umount 会失败。如果强制 umount，可能导致包括但不限于文件系统损坏、设备损坏等问题。
- SD 卡移除前，需确保所有目录及文件全部关闭，并进行 umount 操作。如果强制拔卡，可能导致包括但不限于 SD 数据丢失、SD 卡损坏等问题。

## 2.1.3 开发指导

**开发流程**

推荐驱动开发人员使用 VFS 框架来注册/卸载设备，应用层使用 open()、read()操作设备（字符设备）文件来调用驱动。

1. 系统调用了 los_vfs_init()后，会将"/"作为 root_inode。
2. 调用 register_driver()、register_blockdriver()接口生成设备结点，调用 mount()挂载的目标路径为挂载点。
3. 生成结点的同时进行结构体信息的填充，然后根据结点名插入到树中合适的位置，保持有序。
4. 在调用时根据路径在树中进行查找，匹配到相应的设备或挂载点。
5. 通过查找到的结点指针可调用相应的函数。

**文件描述符**

本系统中，进程的文件描述符最多有 256 个（File 和 Socket 描述符合并统计），系统文件描述符共 640 个，系统文件描述符规格：

- File 描述符，普通文件描述符，系统总规格为 512。
- Socket 描述符，系统总规格为 128。

**VFS 支持的操作**

open, close, read, write, seek, ioctl, fcntl, mmap, sync, dup, dup2, truncate, opendir, closedir, readdir, readdir, rewinddir, mount, umount, statfs, unlink, remove, mkdir, rmdir, rename, stat, utime, seek64, fallocate, fallocate64, truncate64, chmod, chown。

**说明**

- 当前只提供修改 jffs2 文件以及 vfs 设备节点属性的接口，各个系统对只读等属性有各自的处理方式。
- 在 HarmonyOS 内核中属性并不冲突（可以任意修改）。
- 在 HarmonyOS 内核中只读属性文件/目录不允许被删除。
- 在 HarmonyOS 内核中只读属性文件/目录允许 rename。
- 只读文件不允许以 O_CREAT、O_TRUNC，以及有含有写的权限的方式打开。
- 在 HarmonyOS 内核中设置的系统文件加上隐藏属性，在 Windows 中只能通过命令行找到（在显示，不显示隐藏文件的属性情况下都不能看到）。

## 2.1.4 编程实例

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <dirent.h>
```

```
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>

int main()
{
    int ret;
    char *dirname = "/test";
    char *pathname0 = "/test/test0";
    char *pathname1 = "/test/test1";
    char *pathname2 = "/test/test2";
    struct dirent **namelist;
    int num;

    ret = mkdir(dirname, 0777);
    if ((ret < 0) && (errno != EEXIST)) {
        goto EXIT;
    }

    ret = mkdir(pathname0, 0777);
    if ((ret < 0) && (errno != EEXIST)) {
        goto EXIT0;
    }

    ret = mkdir(pathname1, 0777);
    if ((ret < 0) && (errno != EEXIST)) {
        goto EXIT1;
    }

    ret = mkdir(pathname2, 0777);
    if ((ret < 0) && (errno != EEXIST)) {
        goto EXIT2;
    }

    num = scandir(dirname, &namelist, NULL, alphasort);
    if (num < 0) {
        perror("scandir");
    } else {
```

```
42.             while (num--) {
43.                 printf("%s\n", namelist[num]->d_name);
44.                 free(namelist[num]);
45.             }
46.             free(namelist);
47.         }
48.
49.         printf("fs_demo exit.\n");
50.         return 0;
51.
52.  EXIT2:
53.         remove(pathname2);
54.  EXIT1:
55.         remove(pathname1);
56.  EXIT0:
57.         remove(pathname0);
58.  EXIT:
59.         remove(dirname);
60.         return 0;
61.  }
```

### 2.1.5 结果验证

```
1.  OHOS # test2
2.  test1
3.  test0
4.  fs_demo exit.
```

## 2.2 NFS

### 2.2.1 概述

NFS 是 Network File System（网络文件系统）的缩写。它最大的功能是可以通过网络，让不同的机器、不同的操作系统彼此分享其他用户的文件。因此，

用户可以简单地将它看做是一个文件系统服务，在一定程度上相当于 Windows 环境下的共享文件夹。

NFS 客户端用户，能够将网络远程的 NFS 服务端分享的目录挂载到本地端的机器中，运行程序和共享文件，但不占用当前的系统资源，所以，在本地端的机器看起来，远程服务端的目录就好像是自己的一个磁盘一样。

### 2.2.2 注意事项

- 当前 NFS 文件不支持权限控制，请在创建 NFS 目录和文件时使用 777 权限。
- 当前 NFS 文件不支读阻塞和写阻塞。
- 当前 NFS 文件不支持信号功能。
- 当前 NFS 文件系统 mount 路径长度（不包含 IP 的长度）不超过 255 个字符，超过时返回 ENAMETOOLONG 错误。
- 当前 NFS 文件支持的操作有：open, close, read, write, seek, dup, dup2, sync, opendir, closedir, readdir, readdir_r, rewinddir, scandir, statfs, remove, unlink, mkdir, rmdir, rename, stat, stat64, seek64, mmap, mount, umount。
- 当前 NFS 支持 TCP 和 UDP 两种传输层协议，默认使用 TCP。
- open 打开一个文件，参数有 O_TRUNC 时，必须同时拥有写的权限，才会将文件中的内容清空。
- 在文件未关闭的情况下，rename()函数重命名 A 为 B 之后，不会改变文件 fd。

- NFS 功能目前处于 beta 测试阶段，可能存在功能不稳定的情况，建议您不要用于正式商用产品当中。

## 2.2.3 开发指导

1. **搭建 NFS 服务器。**

   这里以 Ubuntu 操作系统为例，说明服务器端设置步骤。

   a. 安装 NFS 服务器软件。

   设置好 Ubuntu 系统的下载源，保证网络连接好的情况下执行：

```
1. sudo apt-get install nfs-kernel-server
```

   b. 创建用于挂载的目录并设置完全权限

```
1. mkdir /home/sqbin/nfs
2. sudo chmod 777 /home/sqbin/nfs
```

   c. 设置和启动 NFS server。

   修改 NFS 配置文件/etc/exports，添加如下一行：

```
1. /home/sqbin/nfs *(rw,no_root_squash,async)
```

   其中/home/sqbin/nfs 是 NFS 共享的根目录。

   执行以下命令启动 NFS server：

```
2. sudo /etc/init.d/nfs-kernel-server start
```

   执行以下命令重启 NFS server：

```
3. sudo /etc/init.d/nfs-kernel-server restart
```

2. **设置单板为 NFS 客户端。**

   本指导中的 NFS 客户端指运行 HarmonyOS 内核的设备。

   . 硬件连接设置。

HarmonyOS 内核设备连接到 NFS 服务器的网络。设置两者 IP，使其处于同一网段。比如，设置 NFS 服务器的 IP 为 10.67.212.178/24，设 HarmonyOS 内核设备 IP 为 10.67.212.3/24。

HarmonyOS 内核设备上的 IP 信息可通过 ifconfig 命令查看。

a. 启动网络，确保单板到 NFS 服务器之间的网络通畅。

启动以太网或者其他类型网络，使用 ping 命令检查到服务器的网络是否通畅。

```
OHOS # ping 10.67.212.178
[0]Reply from 10.67.212.178: time=1ms TTL=63
[1]Reply from 10.67.212.178: time=0ms TTL=63
[2]Reply from 10.67.212.178: time=1ms TTL=63
[3]Reply from 10.67.212.178: time=1ms TTL=63
--- 10.67.212.178 ping statistics ---
4 packets transmitted, 4 received, 0 loss
```

客户端 NFS 初始化，运行命令：

```
OHOS # mkdir /nfs
OHOS # mount 10.67.212.178:/home/sqbin/nfs /nfs nfs 1011 1000
```

将从串口得到如下回应信息，表明初始化 NFS 客户端成功。

```
OHOS # mount 10.67.212.178:/home/sqbin/nfs /nfs nfs 1011 1000
Mount nfs on 10.67.212.178:/home/sqbin/nfs, uid:1011, gid:1000
Mount nfs finished.
```

该命令将服务器 10.67.212.178 上的/home/sqbin/nfs 目录 mount 在 HarmonyOS 内核设备上的/nfs 上。

**说明**

本例默认 nfs server 已经配置可用，即示例中服务器 10.67.212.178 上的 /home/sqbin/nfs 已配置可访问。

mount 命令的格式为：

```
1. mount <SERVER_IP:SERVER_PATH> <CLIENT_PATH> nfs
```

其中“SERVER_IP”表示服务器的 IP 地址；“SERVER_PATH”表示服务器端 NFS 共享目录路径；“CLIENT_PATH”表示设备上的 NFS 路径。

如果不想有 NFS 访问权限限制，请在 Linux 命令行将 NFS 根目录权限设置成 777：

```
1. chmod -R 777 /home/sqbin/nfs
```

至此，NFS 客户端设置完毕。NFS 文件系统已成功挂载。

2. **利用 NFS 共享文件。**

在 NFS 服务器下新建目录 dir，并保存。在 HarmonyOS 内核下运行 ls 命令：

```
1. OHOS # ls /nfs
```

则可从串口得到如下回应：

```
2. OHOS # ls /nfs
3. Directory /nfs:
4. drwxr-xr-x 0            u:0       g:0       dir
```

可见，刚刚在 NFS 服务器上新建的 dir 目录已同步到客户端(HarmonyOS 内核系统)的/nfs 目录，两者保持同步。

同样地，在客户端(HarmonyOS 内核系统)上创建文件和目录，在 NFS 服务器上也可以访问，读者可自行体验。

**平台差异性。**

目前，NFS 客户端仅支持 NFS v3 部分规范要求，因此对于规范支持不全的服务器，无法完全兼容。在开发测试过程中，建议使用 Linux 的 NFS server，因为其对 NFS 支持很完善。

# 2.3 RAMFS

## 2.3.1 概述

RAMFS 是一个可动态调整大小的基于 RAM 的文件系统。RAMFS 没有后备存储源。向 RAMFS 中进行的文件写操作也会分配目录项和页缓存，但是数据并不写回到任何其他存储介质上，掉电后数据丢失。

RAMFS 文件系统把所有的文件都放在 RAM 中，所以读/写操作发生在 RAM 中，可以用 RAMFS 来存储一些临时性或经常要修改的数据，例如/tmp 和/var 目录，这样既避免了对存储器的读写损耗，也提高了数据读写速度。

HarmonyOS 内核的 RAMFS 是一个简单的文件系统，它是基于 RAM 的动态文件系统的一种缓冲机制。

HarmonyOS 内核的 RAMFS 基于虚拟文件系统层（VFS），不能格式化。

## 2.3.2 注意事项

- RAMFS 文件系统的读写指针没有分离，所以以 O_APPEND（追加写）方式打开文件后，读指针也在文件尾，读文件前需要用户手动置位。
- RAMFS 只能挂载一次，一次挂载成功后，后面不能继续挂载到其他目录。
- RAMFS 文件数量受信号量资源限制，不能超过 LOSCFG_BASE_IPC_SEM_LIMIT。
- open 打开一个文件，参数有 O_TRUNC 时，会将文件中的内容清空。

- RAMFS 文件系统支持的操作有：open, close, read, write, seek, opendir, closedir, readdir, readdir_r, rewinddir, sync, statfs, remove, unlink, mkdir, rmdir, rename, stat, stat64, seek64, mmap, mount, umount。
- RAMFS 属于调测功能，默认配置为关闭，正式产品中禁止使用该功能。

### 2.3.3 开发指导

1. RAMFS 文件系统的初始化。

```
void ram_fs_init(void) {
        int swRet=0;
        swRet = mount(NULL, RAMFS_DIR, "ramfs", 0, NULL);
        if (swRet) {
            dprintf("mount ramfs err %d\n", swRet);
            return;
        }
        dprintf("Mount ramfs finished.\n");
 }
```

调用初始化函数，随后在 HarmonyOS 内核系统启动时可以看到如下显示，表示 RAMFS 文件系统已初始化成功：

```
Mount ramfs finished
```

2. RAMFS 文件系统的挂载。

```
OHOS # mount 0 /ramfs ramfs
```

将从串口得到如下回应信息，表明挂载成功。

```
OHOS # mount 0 /ramfs ramfs
mount ok
```

3. RAMFS 文件系统的卸载。

```
OHOS # umount /ramfs
```

将从串口得到如下回应信息，表明卸载成功。

```
2. OHOS # umount /ramfs
3. umount ok
```

# 2.4 FAT

## 2.4.1 概述

FAT 文件系统是 File Allocation Table（文件配置表）的简称，FAT 文件系统有 FAT12、FAT16、FAT32。FAT 文件系统将硬盘分为 MBR 区、DBR 区、FAT 区、DIR 区、DATA 区等 5 个区域。

FAT 文件系统支持多种介质，特别在可移动存储介质（U 盘、SD 卡、移动硬盘等）上广泛使用。可以使嵌入式设备和 Windows、Linux 等桌面系统保持很好的兼容性，方便用户管理操作文件。

HarmonyOS 内核的 FAT 文件系统具有代码量和资源占用小、可裁切、支持多种物理介质等特性，并且与 Windows、Linux 等系统保持兼容，支持多设备、多分区识别等功能。

HarmonyOS 内核支持硬盘多分区，可以在主分区以及逻辑分区上创建 FAT 文件系统。同时 HarmonyOS 内核也可以识别出硬盘上其他类型的文件系统。

## 2.4.2 注意事项

- 最多支持同时打开的 fatfs 文件（文件夹）数为 512。
- 以可写方式打开一个文件后，未 close 前再次打开会失败。多次打开同一文件，必须全部使用只读方式。长时间打开一个文件，没有 close 时数据会丢失，必须 close 才能保存。

- FAT 文件系统中，单个文件不能大于 4G。
- 当有两个 SD 卡插槽时，卡 0 和卡 1 不固定，先插上的为卡 0，后插上的为卡 1。
- 当多分区功能打开，存在多分区的情况下，卡 0 注册的设备节点 /dev/mmcblk0(主设备)和/dev/mmcblk0p0(次设备)是同一个设备，禁止对主设备进行操作。
- FAT 文件系统的读写指针没有分离，所以以 O_APPEND（追加写）方式打开文件后，读指针也在文件尾，读文件前需要用户手动置位。
- FAT 文件系统的 stat 及 lstat 函数获取出来的文件时间只是文件的修改时间。暂不支持创建时间和最后访问时间。微软 FAT 协议不支持 1980 年以前的时间。
- open 打开一个文件，参数有 O_TRUNC 时，会将文件中的内容清空。
- FAT 文件系统支持的操作有：open, close, read, write, seek, sync, opendir, closedir, readdir, rewinddir, readdir_r, statfs, remove, unlink, mkdir, rmdir, rename, stat, stat64, seek64, fallocate, fallocate64, truncate, truncate64,mount, umount。
- 为避免 SD 卡使用异常和内存泄漏，SD 卡使用过程中拔卡，用户必须先关闭正处于打开状态的文件和目录，之后 umount 挂载节点。
- 在 format 操作之前，若 fat 文件系统已挂载，需确保所有目录及文件全部关闭，否则 format 会失败。
- FAT 支持只读属性挂载：

- 当 mount 函数的入参为 MS_RDONLY 时，FAT 将开启只读属性，所有的带有写入的接口，如 write、mkdir、unlink，以及通过非 O_RDONLY 属性打开的文件，将均被拒绝，并传出 EACCESS 错误码（format 接口除外）。
- 当 mount 函数的入参为 MS_NOSYNC 时，FAT 不会主动将 cache 的内容写回存储器件。FAT 的如下接口（open、close、 unlink、rename、mkdir、rmdir、truncate）不会自动进行 sync 操作，速度可以提升，但是需要上层主动调用 sync 来进行数据同步，否则下电可能会数据丢失。

- FAT 文件系统有定时刷 cache 功能。在 menuconfig 中开启 LOSCFG_FS_FAT_CACHE_SYNC_THREAD 选项，打开后系统会创建一个任务刷 cache，默认每隔 5 秒检查 cache 中脏数据块比例，超过 80%时进行 sync 操作，将 cache 中的脏数据全部写回磁盘。任务优先级、刷新时间间隔以及脏数据块比例的阈值可分别通过接口 LOS_SetSyncThreadPrio、LOS_SetSyncThreadInterval 和 LOS_SetDirtyRatioThreshold 设置。
- 当前 cache 的默认大小为 16 个块，每个块 256 个扇区。

## 2.4.3 开发指导

**设备识别**

- 在 ffconf.h 文件中配置 FF_MULTI_PARTITION 为 1，可使用多分区功能。
- 在 ffconf.h 文件中配置 FF_VOLUMES 大于 2 时，可使用多设备功能。

多设备、多分区功能开启后，系统对于插上的 sd 卡自动识别，自动注册设备节点如上图所示。mmcblk0 和 mmcblk1 为卡 0 和卡 1，是独立的主设备，

mmcblk0p0、mmcblk0p1 为卡 0 的两个分区，可作为分区设备使用。在有分区设备存在的情况下，禁止使用主设备。

可以使用 partinfo 命令查看所识别的分区信息。

```
OHOS # partinfo /dev/mmcblk0p0
part info :
disk id              : 0
part_id in system: 0
part no in disk   : 0
part no in mbr     : 1
part filesystem   : 0C
part dev name      : mmcblk0p0
part sec start     : 8192
part sec count     : 31108096
```

**FAT 文件系统的挂载**

运行命令：

```
OHOS # mount /dev/mmcblk0p0 /vs/sd vfat
```

将从串口得到如下回应信息，表明挂载成功。

```
OHOS # mount /dev/mmcblk0p0 /vs/sd vfat
mount ok
```

**FAT 文件系统的卸载**

运行命令：

```
OHOS # umount /vs/sd
```

将从串口得到如下回应信息，表明卸载成功。

```
OHOS # umount /vs/sd
umount ok
```

# 2.5 JFFS2

## 2.5.1 概述

JFFS2 是 Journalling Flash File System Version 2（日志文件系统）的缩写，是 MTD 设备上实现的日志型文件系统。JFFS2 主要应用于 NOR FLASH，其特点是：可读写、支持数据压缩、提供了崩溃/掉电安全保护、提供“写平衡”支持等。

闪存与磁盘介质有许多差异，因此直接将磁盘文件系统运行在闪存上存在性能和安全性上的不足。为解决这一问题，需要实现一个特别针对闪存的文件系统，JFFS2 就是这样一种文件系统。

HarmonyOS 内核的 JFFS2 主要应用于对 NOR Flash 闪存的文件管理，并且支持多分区。

## 2.5.2 注意事项

- 目前 JFFS2 文件系统用于 NOR Flash，最终调用 NOR Flash 驱动接口，因此使用 JFFS2 文件系统之前要确保硬件上有 NOR Flash，且驱动初始化成功（spinor_init()返回 0）。
- 系统会自动对起始地址和分区大小根据 block 大小进行对齐操作。有效的分区号为 0~19。
- 目前支持 mkfs.jffs2 工具，用户可根据自己实际情况修改参数值，其他用法用户可自行搜索查看。
- open 打开一个文件，参数有 O_TRUNC 时，会将文件中的内容清空。

- 目前 JFFS2 文件系统支持的操作有：open, close, read, write, seek, opendir, closedir, readdir, readdir_r, rewinddir, statfs, sync, remove, unlink, mkdir, rmdir, rename, stat, stat64, seek64, mmap, mount, umount, chmod, chown。
- JFFS2 支持以只读属性挂载，当 mount 函数的入参 mountflags 为 MS_RDONLY 时，JFFS 将开启只读属性，所有的带有写入的接口，如 write、mkdir、unlink，以及通过非 O_RDONLY 属性打开的文件，将均被拒绝，并传出 EACCESS 错误码。

## 2.5.3 开发指导

### 添加 JFFS2 分区

调用 add_mtd_partition 函数添加 JFFS2 分区，该函数会自动为设备节点命名，对于 JFFS2，其命名规则是“/dev/spinorblk”加上分区号。

add_mtd_partition 函数有四个参数，第一个参数表示介质，有“nand”和“spinor”两种，JFFS2 分区在“spinor”上使用，而“nand”是提供给 YAFFS2 使用的。

第二个参数表示起始地址，第三个参数表示分区大小，这两个参数都以 16 进制的形式传入。

最后一个参数表示分区号，有效值为 0~19。

```
1.  if (uwRet = add_mtd_partition("spinor", 0x100000, 0x800000, 0) != 0) {
2.      dprintf("add jffs2 partition failed, return %d\n", uwRet);
```

```
} else {
        dprintf("Mount jffs2 on spinor.\n");
        uwRet = mount("/dev/spinorblk0", "/jffs0", "jffs2", 0, NULL);
        if (uwRet) {
            dprintf("mount jffs2 err %d\n", uwRet);
            dprintf("Mount jffs2 on nor finished.\n");
        }
}

if (uwRet = add_mtd_partition("spinor", 0x900000, 0x200000, 1) != 0) {
        dprintf("add jffs2 partition failed, return %d\n", uwRet);
}
```

成功后，在 Shell 中可以使用 partition spinor 命令查看 spinor flash 分区信息。

```
OHOS # partition spinor
spinor partition num:0, dev name:/dev/spinorblk0, mountpt:/jffs0,
startaddr:0x0100000,length:0x0800000
spinor partition num:1, dev name:/dev/spinorblk1, mountpt:(null),
startaddr:0x0900000,length:0x0200000
```

**挂载 JFFS2**

调用 mount()函数实现设备节点和挂载点的挂载。

该函数有五个参数，第一个参数表示设备节点，这个参数需要和 add_mtd_partition()函数对应起来，第二个参数表示挂载点。第三个参数表示文件系统类型。

最后两个参数表示挂载标志和数据，默认为 0 和 NULL；这一操作也可以在 Shell 中使用 mount 命令实现，最后两个参数不需要用户给出。

运行命令：

```
OHOS # mount /dev/spinorblk1 /jffs1 jffs2
```

将从串口得到如下回应信息，表明挂载成功。

```
OHOS # mount /dev/spinorblk1 /jffs1 jffs2
mount OK
```

挂载成功后，用户就能对 norflash 进行读写操作。

**卸载 JFFS2**

调用 umount()函数卸载分区，只需要正确给出挂载点即可。

运行命令：

```
OHOS # umount /jffs1
```

将从串口得到如下回应信息，表明卸载成功。

```
OHOS # umount /jffs1
umount ok
```

**删除 JFFS2 分区**

调用 delete_mtd_partition 删除已经卸载的分区。

该函数有两个参数，第一个参数是分区号，第二个参数为介质类型，该函数与 add_mtd_partition()函数对应。

```
uwRet = delete_mtd_partition(1,"spinor");
if(uwRet != 0) {
    printf("delete jffs2 error\n");
} else {
    printf("delete jffs2 ok\n");
}
```

```
7.
8.  OHOS # partition spinor
9.  spinor partition num:0, dev name:/dev/spinorblk0, mountpt:/jffs0,
    startaddr:0x0100000,length:0x0800000
```

### 制作 JFFS2 文件系统镜像

使用 mkfs.jffs2 工具，制作镜像默认命令见下。页大小默认为 4KiB，eraseblock 大小默认 64KiB，镜像大小适应源目录并以 0xFF 填充为 eraseblock 大小的整数倍。若实际参数与下面不同时，修改相应参数。

```
1.  ./mkfs.jffs2 -d rootfs/ -o rootfs.jffs2
```

| 指令 | 含义 |
| --- | --- |
| -s | 页大小，不指定默认为 4KiB |
| -e | eraseblock 大小，不指定默认为 64KiB |
| -p | 镜像大小，不指定默认适应源目录并以 0xFF 填充为 eraseblock 大小的整数倍 |
| -d | 要制作成文件系统镜像的源目录 |
| -o | 要制成的镜像名称 |

**表 1** 指令含义表

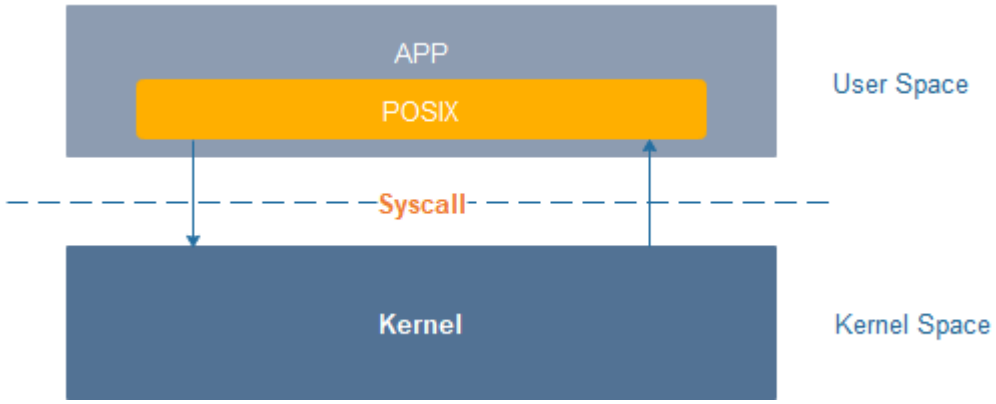
# 3 标准库

## 3.1 标准库

HarmonyOS 内核使用 **musl libc** 库，支持标准 POSIX 接口，开发者可基于 POSIX 标准接口开发内核之上的组件及应用。

### 3.1.1 框架流程

图 1 POSIX 接口框架




**musl libc** 库支持 POSIX 标准，涉及的系统调用相关接口由 HarmonyOS 内核适配支持 ，以满足接口对外描述的功能要求。

### 3.1.2 开发指导

标准库支持接口的详细情况请参考 C 库的 API 文档，其中也涵盖了与 POSIX 标准之间的差异说明。开发者可根据已经提供的接口，开发组件及应用等。

### 3.1.3 操作实例

在本示例中，主线程创建了 THREAD_NUM 个子线程，每个子线程启动后等待被主线程唤醒，主线程成功唤醒所有子线程后，子线程继续执行直至生命周期结束，同时主线程通过 pthread_join 方法等待所有线程执行结束。

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

#ifdef __cplusplus
#if __cplusplus
extern "C" {
#endif /* __cplusplus */
#endif /* __cplusplus */

#define THREAD_NUM 3
int g_startNum = 0; /* 启动的线程数 */
int g_wakenNum = 0; /* 唤醒的线程数 */

struct testdata {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} g_td;

/* *
 * 子线程入口函数
 */
static void *ChildThreadFunc(void *arg)
{
    int rc;
    pthread_t self = pthread_self();

    /* 获取 mutex 锁 */
    rc = pthread_mutex_lock(&g_td.mutex);
    if (rc != 0) {
```

```
31.             printf("ERROR:take mutex lock failed, error code is %d!\n", rc);
32.             goto EXIT;
33.         }
34.
35.         /* g_startNum 计数加一，用于统计已经获得 mutex 锁的子线程个数 */
36.         g_startNum++;
37.
38.         /* 等待 cond 条件变量 */
39.         rc = pthread_cond_wait(&g_td.cond, &g_td.mutex);
40.         if (rc != 0) {
41.             printf("ERROR: pthread condition wait failed, error code is %d!\n", rc);
42.             (void)pthread_mutex_unlock(&g_td.mutex);
43.             goto EXIT;
44.         }
45.
46.         /* 尝试获取 mutex 锁 */
47.         rc = pthread_mutex_trylock(&g_td.mutex);
48.         if (rc == 0) {
49.             /* 失败则退出 */
50.             printf("ERROR: mutex trylock failed, error code is %d!\n", rc);
51.             goto EXIT;
52.         }
53.
54.         /* g_wakenNum 计数加一，用于统计已经被 cond 条件变量唤醒的子线程个数 */
55.         g_wakenNum++;
56.
57.         /* 释放 mutex 锁 */
58.         rc = pthread_mutex_unlock(&g_td.mutex);
59.         if (rc != 0) {
60.             printf("ERROR: mutex release failed, error code is %d!\n", rc);
61.             goto EXIT;
62.         }
63. EXIT:
64.         return NULL;
65. }
66.
67. static int testcase(void)
68. {
```

```
69.     int i, rc;
70.     pthread_t thread[THREAD_NUM];
71.
72.     /* 初始化 mutex 锁 */
73.     rc = pthread_mutex_init(&g_td.mutex, NULL);
74.     if (rc != 0) {
75.         printf("ERROR: mutex init failed, error code is %d!\n", rc);
76.         goto ERROROUT;
77.     }
78.
79.     /* 初始化 cond 条件变量 */
80.     rc = pthread_cond_init(&g_td.cond, NULL);
81.     if (rc != 0) {
82.         printf("ERROR: pthread condition init failed, error code is %d!\n", rc);
83.         goto ERROROUT;
84.     }
85.
86.     /* 批量创建 THREAD_NUM 个子线程 */
87.     for (i = 0; i < THREAD_NUM; i++) {
88.         rc = pthread_create(&thread[i], NULL, ChildThreadFunc, NULL);
89.         if (rc != 0) {
90.             printf("ERROR: pthread create failed, error code is %d!\n", rc);
91.             goto ERROROUT;
92.         }
93.     }
94.
95.     /* 等待所有子线程都完成 mutex 锁的获取 */
96.     while (g_startNum < THREAD_NUM) {
97.         usleep(100);
98.     }
99.
100.    /* 获取 mutex 锁，确保所有子线程都阻塞在 pthread_cond_wait 上 */
101.    rc = pthread_mutex_lock(&g_td.mutex);
102.    if (rc != 0) {
103.        printf("ERROR: mutex lock failed, error code is %d\n", rc);
104.        goto ERROROUT;
105.    }
106.
```

```
107.    /* 释放 mutex 锁 */
108.    rc = pthread_mutex_unlock(&g_td.mutex);
109.    if (rc != 0) {
110.        printf("ERROR: mutex unlock failed, error code is %d!\n", rc);
111.        goto ERROROUT;
112.    }
113.
114.    for (int j = 0; j < THREAD_NUM; j++) {
115.        /* 在 cond 条件变量上广播信号 */
116.        rc = pthread_cond_signal(&g_td.cond);
117.        if (rc != 0) {
118.            printf("ERROR: pthread condition failed, error code is %d!\n", rc);
119.            goto ERROROUT;
120.        }
121.    }
122.
123.    sleep(1);
124.
125.    /* 检查是否所有子线程都已被唤醒 */
126.    if (g_wakenNum != THREAD_NUM) {
127.        printf("ERROR: not all threads awaken, only %d thread(s) awaken!\n", g_wakenNum);
128.        goto ERROROUT;
129.    }
130.
131.    /* join 所有子线程，即等待其结束 */
132.    for (i = 0; i < THREAD_NUM; i++) {
133.        rc = pthread_join(thread[i], NULL);
134.        if (rc != 0) {
135.            printf("ERROR: pthread join failed, error code is %d!\n", rc);
136.            goto ERROROUT;
137.        }
138.    }
139.
140.    /* 销毁 cond 条件变量 */
141.    rc = pthread_cond_destroy(&g_td.cond);
142.    if (rc != 0) {
143.        printf("ERROR: pthread condition destroy failed, error code is %d!\n", rc);
144.        goto ERROROUT;
```

```
    }
    return 0;
ERROROUT:
    return -1;
}

/*
 * 示例代码主函数
 */
int main(int argc, char *argv[])
{
    int rc;

    /* 启动测试函数 */
    rc = testcase();
    if (rc != 0) {
        printf("ERROR: testcase failed!\n");
    }

    return 0;
}
#ifdef __cplusplus
#if __cplusplus
}
#endif /* __cplusplus */
#endif /* __cplusplus */
```

## 3.2 与 Linux 标准库的差异

本章节描述了 HarmonyOS 内核承载的标准库与 Linux 标准库之间存在的关键差异。更多差异详见 C 库 API 文档说明。

## 3.2.1 进程

1. HarmonyOS 用户态进程优先级只支持静态优先级且用户态可配置的优先级范围为 10(最高优先级)-31(最低优先级)。
2. HarmonyOS 用户态线程优先级只支持静态优先级且用户态可配置的优先级范围为 0(最高优先级)-31(最低优先级)。
3. HarmonyOS 调度策略支持 SCHED_RR 和 SCHED_FIFO。
4. sched_yield()为进程主动放弃 CPU；thrd_yield()为线程主动放弃 CPU。

## 3.2.2 内存

### 3.2.2.1 与 Linux mmap 的差异

mmap 接口原型为：void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset)。

其中，参数 fd 的生命周期实现与 Linux glibc 存在差异。具体体现在，glibc 在成功调用 mmap 进行映射后，可以立即释放 fd 句柄。在 HarmonyOS 内核中，不允许用户在映射成功后立即关闭相关 fd，只允许在取消映射 munmap 后再进行 fd 的 close 操作。如果用户不进行 fd 的 close 操作，操作系统将在进程退出时对该 fd 进行回收。

### 3.2.2.2 代码举例

Linux 目前支持的情况如下：

```
int main(int argc, char *argv[])
{
    int fd;
    void *addr = NULL;
    ...
    fd = open(argv[1], O_RDONLY);
    if (fd == -1){
        perror("open");
        exit(EXIT_FAILURE);
    }
    addr = mmap(NULL, length, PROT_READ, MAP_PRIVATE, fd, offset);
    if (addr == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    close(fd); /* close immediately, HarmonyOS do not support this way */
    ...
    exit(EXIT_SUCCESS);
}
```

HarmonyOS 支持的情况如下：

```
int main(int argc, char *argv[])
{
    int fd;
    void *addr = NULL;
    ...
    fd = open(argv[1], O_RDONLY);
    if (fd == -1){
        perror("open");
        exit(EXIT_FAILURE);
    }
    addr = mmap(NULL, length, PROT_READ, MAP_PRIVATE, fd, offset);
    if (addr == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    ...
    munmap(addr, length);
```

```
18.     close(fd); /* close after munmap */
19.     exit(EXIT_SUCCESS);
20. }
```

## 3.2.3 文件系统

**系统目录**：用户无法对其进行修改，或是设备挂载。包含/dev，/proc，/app，/bin，/data，/etc，/lib，/system，/usr 目录。

**用户目录**：用户可以在该目录下进行文件创建、读写，但**不能进行设备挂载。**用户目录指/storage 目录。

除**系统目录**与**用户目录**之外，用户可以自行创建文件夹进行设备的挂载。但是要注意，已挂载的文件夹及其子文件夹不允许重复或者嵌套挂载，非空文件夹不允许挂载。

### 3.2.3.1 信号

- 信号默认行为不支持 STOP、CONTINUE、COREDUMP 功能。
- 无法通过信号唤醒正在睡眠状态（举例：进程调用 sleep 函数进入睡眠）的进程。原因：信号机制无唤醒功能，当且仅当进程被 CPU 调度运行时才能处理信号内容。
- 进程退出后会发送 SIGCHLD 给父进程，发送动作无法取消。
- 信号仅支持 1-30 号信号，接收方收到多次同一信号，仅执行一次回调函数。

### 3.2.3.2 Time

HarmonyOS 当前时间精度以 tick 计算，系统默认 10ms/tick。sleep、timeout 系列函数时间误差<=20ms。

# 4 调测