206.2.0

Huawei LiteOS 常见问题解答

文档版本	3.0
发布日期	2022-03-23

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司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的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://www.huawei.com

客户服务邮箱： support@huawei.com

客户服务电话： 4008302118

目 录

1 前言

概述

本文为使用Huawei LiteOS开发的程序员而写，目的是为您在开发过程中遇到的问题提供解决办法和帮助。

说明

本文以Hi3516CV300为例，如未有特殊说明，Hi3516EV100、Hi3516EV200、Hi3516EV300、Hi3518EV200、Hi3518EV300、Hi3559A、Hi3556A与Hi3516CV300完全一致。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
Hi3516C	V300
Hi3516E	V100
Hi3516E	V200
Hi3516E	V300
Hi3518E	V200
Hi3518E	V300

读者对象

本文档（本指南）主要适用于以下工程师：

- 技术支持工程师
- 软件开发工程师

修订记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

日期	修订版本	描述
2018年08月22日	1.0	第1次正式版本发布
2021年11月25日	2.0	206.1.0版本发布
2022年3月23日	3.0	206.2.0版本发布

2 动态加载常见问题

2.1 动态库加载成功，执行库中函数进入异常

- **原因1**：生成库的时候未加编译选项 **-nostdlib**，导致动态库使用了编译器提供的标准库。

  解决方案：原则上动态库使用的标准库不能是系统的标准库或者编译器提供的标准库，必须使用 Huawei LiteOS 系统提供的。需要在编译生成动态库的时候添加 **-nostdlib**，同时不要使用 **-l** 链接标准库或者添加标准库的搜索路径。

- **原因2**：生成动态库的时候未加编译选项 **-fPIC-shared**，这两个编译选项是生成共享动态库必须加的选项。

  解决方案：编译生成动态库的时候添加 **-fPIC-shared**，另外如果生成动态库的过程是由 .c 文件（或者 .cpp 文件）到 .o 文件，再由 .o 文件到 .so 文件，生成 .o 文件的过程也需要加 **-fPIC** 。

- **原因3**：最终生成镜像的时候没有使用 **LITEOS_CFLAGS** 或 **LITEOS_CXXFLAGS** ，因而系统使用了编译器提供的标准库，导致使用的相关函数符号地址错误。

  解决方案：在编译生成镜像的 makefile 文件中添加“include config.mk”（在 Huawei LiteOS系统主目录下），编译选项使用 **LITEOS_CFLAGS** 或 **LITEOS_CXXFLAGS** 。

- **原因4**：没有加编译选项 **-mlong-calls** ，导致长跳转函数调用失败。

  解决方案：编译选项加上 **-mlong-calls**（生成的目标码允许同一个源文件中的函数调用，调用点和被调函数的距离可以超过 256K 之远）。

2.2 动态库加载不成功，出现无法重定位符号的问题

- **原因1**：该加载的库，存在外部依赖库，导致加载过程重定位外部符号失败。

解决方案：LiteOS提供.so及.o两种类型文件的动态加载能力，.so文件加载支持优先加载依赖文件，.o文件加载，需要用户优先加载依赖的库，同时在加载该库前指定依赖库的加载路径。

- **原因2**：该加载的库，未使用 sym.sh 脚本（Huawei LiteOS 系统目录 “tools/scripts/dynload_tools”下）处理，导致库中使用的外部符号（Huawei LiteOS 系统提供）未存在引用，而在编译成系统镜像的时候被优化处理掉了。

  解决方案：将要加载的库和其依赖的库放在一个路径下，使用 sym.sh 脚本处理，脚本入参为文件夹路径。

- **原因3：**LiteOS主目录下无kernel/extended/dynload/src路径，导致该路径下的未定义符号文件los_dynload_gsymbol.c无法生成。

  解决方案：在LiteOS主目录下创建kernel/extended/dynload/src路径，再按前述原因2的解决方案处理。

- **原因4：**编译环境python版本不匹配，导致抽取动态库未定义符号的python脚本执行失败（sym.sh执行过程出错），生成的los_dynload_gsymbol.c文件不包含未定义符号。

  解决方案：在编译环境安装python2.7版本，保证sym.sh执行过程无错误。

- **原因5：**LiteOS主目录下未定义符号的文件kernel/extended/dynload/src/los_dynload_gsymbol.c，未参与编译链接。

  解决方案： 修改Makefile文件，保证los_dynload_gsymbol.c在生成镜像时参与编译和链接。

- **原因6：**生成镜像的Makefile文件中包含dynload_ld.mk失败或者未包含dynload_ld.mk，导致未定义符号未参与链接。

  解决方案：保证生成镜像的Makefile文件中成功包含dynload_ld.mk。

2.3 动态库加载不成功，LOS_SoLoad 返回 NULL

- **原因1**：动态库的加载路径不正确或者不存在。

  解决方案：传入正确的动态库加载路径及库名。

- **原因2**：动态库加载个数已经达到上限。

  解决方案：请确定已经加载库的个数是否超过宏 **N_MAX_USER_MODULES** 的值，对不需要的动态库调用 LOS_ModuleUnload 进行卸载，或者根据可用内存情况适当修改 **N_MAX_USER_MODULES** 宏的值。

- **原因3**：系统内存不足。

  解决方案：请为动态库预留适当的内存空间加载。

3 C++常见问题

3.1 C++源码包含 math.h 导致数学函数重定义

原因：C++库中实现了部分数学函数库，并在头文件中进行了using 引用，导致与Huawei LiteOS系统C库中的定义冲突了。

解决方案：C++源码math.h的包含修改为cmath，即“#include <math.h>”修改为“#include <cmath>”。

3.2 C++源码编译找不到 stdlib.h

原因：系统中 C++ 头文件包含路径不当，导致cstdlib中执行“#include_next <stdlib.h>”时，找不到系统 libc 中的 stdlib.h 。

解决方案：修改 **los_config.mk 中LITEOS_CXXINCLUDE和LITEOS_LIBC_INCLUDE顺序：LITEOS_LIBC_INCLUDE应在LITEOS_CXXINCLUDE之后。**

须知

如果还存在问题，请确定 **LITEOS_CXXINCLUDE** 编译选项在 **LITEOS_LIBC_INCLUDE** 之前（或者不要加 **LITEOS_LIBC_INCLUDE** ），或者确定 **LITEOS_CXXFLAGS** 在编译选项在 **LITEOS_CFLAGS** 之前。

3.3 C++源码编译找不到 bits/c++config.h

原因：同步修改 los_config.mk 时，添加的 C++ 头文件路径缺失。

解决方案：继承LiteOS代码到自己工程时，确保编译环境和LiteOS编译环境一致，包含了C++的路径。用户可以在自己编译框架中包含C++编译器路径，具体请参考LITEOS_COMPILER_CXX_PATH、LITEOS_CXXINCLUDE在compiler_gcc.mk中的定义。

3.4 使用 C++库运行死机

原因：未调用C++初始化的情况下调用C++代码，会出现访问0地址异常，可根据异常信息排查是否为未初始化导致的问题。

解决方案：调用 LOS_CppSystemInit 函数初始化，具体用法请参见 Huawei LiteOS 开发指南。

3.5 C 调用 C++函数，函数符号找不到

原因：被调用的函数未遵循C的调用约定，导致函数符号编译时按C++的命名修饰规则编译，而在调用处按C的命令修饰规则编译，调用时就会找不到定义。

解决方案：请在函数定义和声明前添加如下C的调用约定：

```
#ifdef __cplusplus
#if __cplusplus
extern "C" {
#endif /* __cplusplus */
#endif /* __cplusplus */
/* code */
...
#ifdef __cplusplus
#if __cplusplus
}
#endif /* __cplusplus */
#endif /* __cplusplus */
```

须知

C++调用C函数，也存在类似问题，请在声明前添加C的调用约定。

3.6 cout 打印不出内容

原因：输出相关的功能或者设备未初始化。

解决方案：在使用cout之前需保证下面函数依次被调用：

```
los_vfs_init();
uart_dev_init();
virtual_serial_init(TTY_DEVICE);
system_console_init(SERIAL);
```

3.7 未定义符号__kernel_cmpxchg 和__kernel_dmb

原因：C++ 一些类型使用的时候会触及原子操作，会去调用定义在 libgcc.a 中上述两个固定内存地址的函数，引起异常死机，因此在生成 libgcc.a 时去掉中的这个函数定义。

解决方案：调用 LiteOS 系统适配的上述两个函数，在根目录下的 “compat/linux/src” 创建 atomic-linux.c 文件将下面的代码放入即可。

```
extern int LOS_AtomicCmpXchg32bits(volatile UINT32 *puwAddr, UINT32 uwNewVal, UINT32 uwOldVal);
extern void dmb(void);

/* Kernel helper for compare-and-exchange.  */
int __kernel_cmpxchg(int oldval,int newval,int *ptr)
{
    return LOS_AtomicCmpXchg32bits(ptr,newval,oldval);
}

/* Kernel helper for memory barrier.  */
void __kernel_dmb(void)
{
}
```

4 文件系统常见问题

4.1 如何定位哪些文件导致的文件打开失败的问题?

答：lsfd 命令可以列出当前已经 open 的文件，当出现文件句柄泄露，导致文件句柄不够而 open 失败的时候，可以通过 **lsfd** 查看当前哪些文件没有关闭。

4.2 如何提升 ramfs 写文件性能?

答：在多次向一个ramfs文件write追加数据的情况下，会导致反复重新申请内存和拷贝，性能较低。因为ramfs文件保存在内存中，文件增大的情况下，需要重新分配一块更大的内存来保存文件。可以通过 void los_set_ramfs_unit(off_t size) 接口设置一次增加分配的内存。如：要循环100次，每次向一个文件 write 32 字节，默认情况下每次 write 都会重新一块内存，将原先数据拷贝到新内存，非常耗时，可以通过 los_set_ramfs_unit(32*100) 直接在下次 write 就分配一块32*100大小的内存，后面的 write 就不需要再重新分配内存了。在不知道要写的文件最终大小情况下，也可以通过该接口将一次分配内存增大，减少需要重新分配内存和拷贝的次数。使用完后需要用 los_set_ramfs_unit(0) 设置回默认值。否则会出现写小文件时也分配了一块很大的内存，导致内存消耗过多。

4.3 jffs2 文件系统挂载后无法创建文件，提示 no space

原因：jffs2 文件系统挂载分区大小有限制，不得小于（3*eraseblock + partitionsize/50）。

解决方案：由于 jffs2 文件系统是日志型文件系统，写和删除操作都需要额外的 erase block ； 且 jffs2 包含 gc 回收机制，需要额外的 erase block 空间进行 gc 操作；因此 flash 分区需要预留一部分空间用于 gc 操作和进行写/删除操作，此空间大小由（ 3 * blocksize + partitionsize/50 ）决定，因此添加分区时，分区大小尽量保证比此预留空间多2个 erase block 以上，否则会影响文件系统正常使用。

4.4 jffs2 文件系统 statfs 显示还有剩余空间，但是写入文件失败

原因：

1. jffs2文件系统写入和删除都需要额外的block，参见4.3。
2. jffs2文件系统中的脏数据小于一个block时，gc不会回收这部分脏数据.

statfs显示的是dirty_size + free_size的大小。即使显示还有剩余空间，由于以上两个原因，也可能会写入失败。目前无法查看jffs2文件系统实际剩余的可用空间。

4.5 同样大小的 jffs2 分区，为什么 linux 能写入的数据比 LiteOS 多?

原因：首先，jffs2文件系统是以node的形式将数据存储到flash上。一次写入操作实际是将一个节点的信息和数据一起写到flash上，比如写入10字节，实际存到flash上占用的空间在72字节左右，其中就包含了节点的额外信息。因此jffs2写入小块数据可能导致负压缩率，单次写入大块数据对flash利用率更高。linux中有一个优化机制，通过牺牲内存来提高flash的利用率。在linux中，每次向flash写入数据的同时，还会缓存在内存中。每当写满4096字节后，系统将这些数据聚合起来进行压缩，形成一个节点写入flash，并将之前的节点都标记为脏数据等待gc回收。减少了节点数，节约了flash空间。但是linux中，向flash写入多少数据就需要耗费同等大小的内存，不足4K以4K计。考虑到内存占用和内存碎片化的问题，LiteOS目前暂时不支持该机制。

解决方案：建议单次写入大块的数据，以提高flash的利用率。

4.6 nfs 文件系统在 Windows 环境下 mount 子目录，提示：No such file or directory

原因：Windows环境下nfs server 共享路径属性配置缺少“-alldirs”选项。

解决方案：在NFS Server中配置共享路径时，需包含“-alldirs”选项。

图 4-1 NFS Server 中选择“Edit exports file”

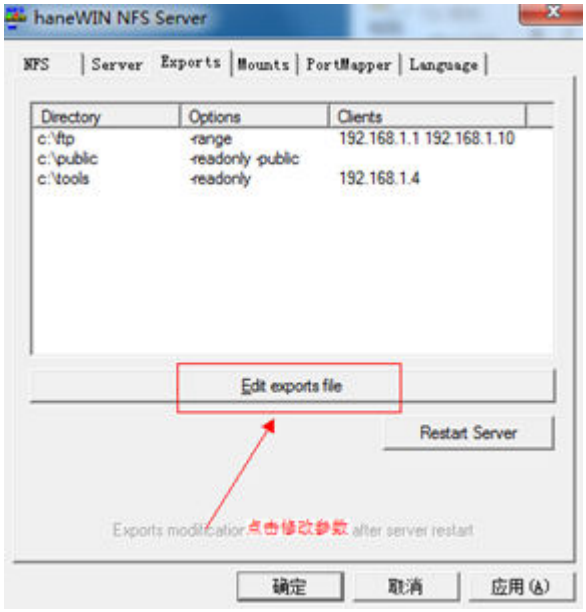
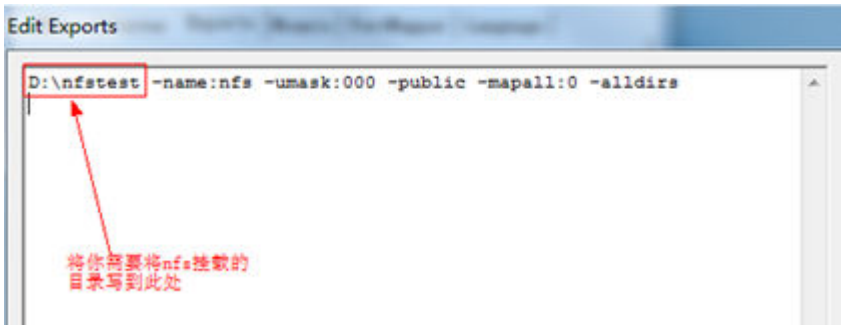


图 4-2 配置共享路径：d:\NFS -name:nfs -umask:000 -public -mapall:0 -alldirs



属性解读：

- **d:\NFS**：指定的window实际挂载的目录。
- **name:nfs**：挂载的名称，即在客户端挂载时所用的名称。
- **-umask:000**：设置用户在NFS文件系统中创建文件的默认权限，为补码形式，默认为022，这里设置为000，以保证最大权限，用户可自行设置。
- **-public**：打开WebNFS访问
- **-mapall:0**：将所有Unix用户id和组id映射为指定用户id和组id。
- **-alldirs**：支持挂载主机文件系统的任何目录。

4.7 写 SD 卡过程中拔卡，重新插卡后发现最后写的文件不存在或文件长度为 0

原因：由于bcache采用LRU（最近最少使用）机制，每次写文件都需要修改目录项，因此目录项所在的cache块一直没有被写回磁盘，重新插卡后查看文件信息时访问目录项无法得到正确结果。

解决方案：

1. 用户可根据自身场景需要，上层主动调用sync()接口，以确保将目录项内容写进磁盘；
2. 可在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，详见《Huawei LiteOS 开发指南和API参考.chm》。

5 网络常见问题

5.1 如何获取系统网关地址

- **Linux平台：**系统的网关地址是指默认路由所对应的下一跳地址，如下图所示，业务代码中一般是通过 system 执行 **route** 命令（或者打开“/proc/net/route”文件），然后分析其输出来确定系统的网关地址。

  图 5-1 Linux 上获取网关地址

```
g00400460@ubuntu:~$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         10.175.100.1    0.0.0.0         UG    0      0        0 enp2s0f0
localnet        *               255.255.252.0   U     0      0        0 enp2s0f0
172.17.0.0      *               255.255.0.0     U     0      0        0 docker0
192.168.122.0   *               255.255.255.0   U     0      0        0 virbr0
```

- **Huawei LiteOS平台：**不支持 route 命令，也不支持 system，proc 下面也没有“ /proc/net/route ”文件，所以无法通过Linux下的方式得到系统的网关地址。LiteOS平台上存在一张默认网卡，其用途等同于 Linux 下的默认路由。默认网卡中的网关地址对应的就是系统的默认网关。

  图 5-2 LiteOS 上获取网关地址

```
Huawei LiteOS # ifconfig
eth0    ip:192.168.1.2 netmask:255.255.255.0 gateway:192.168.1.1
        HWaddr 52:c3:90:3f:93:b6 MTU:1500 Runing Default Link UP
lo      ip:127.0.0.1 netmask:255.0.0.0 gateway:127.0.0.1
        HWaddr 00 MTU:0 Runing Link UP
```

举例如下：

```
#include "lwip/netifapi.h"
#include "lwip/ip_addr.h"

struct in_addr gw;
struct netif *netif = NULL;
netif = netifapi_netif_get_default();
if (NULL == netif) {
    printf("no default netif...\n");
} else {
    gw.s_addr = netif->gw.addr;
    printf("gateway: %s\n", inet_ntoa(gw));
}
```

5.2 如何设置系统网关地址

- **Linux平台：**可以通过系统命令 **route** 来管理系统的路由，业务中可以通过system(“route add default gw 192.168.1.1”) 来设置系统网关，其实route命令设置系统网关，底层使用的是 ioctl 调用，通过 **SIOCADDRT** 命令来设置系统网关。
- **Huawei LiteOS平台：**也支持 **SIOCADDRT** 命令，但是由于路由机制的不同，**SIOCADDRT** 的使用与 Linux 相比，主要有两点略有不同：

  a. Huawei LiteOS 只支持设置系统网关，不支持设置子网路由与主机路由

  b. Huawei LiteOS 平台下设置系统网关时，rt_flags 必须为 **RTF_GATEWAY**，不支持其他的 flag 。

  举例如下：

```
#include "lwip/netifapi.h"
#include "lwip/ip_addr.h"

struct sockaddr_in *saddr;
struct rtentry rt;
int sock;
int ret;

memset(&rt, 0, sizeof(rt));
saddr = (struct sockaddr_in*)&(rt.rt_gateway);
saddr->sin_family = AF_INET;
saddr->sin_addr.s_addr = inet_addr("10.173.210.1");
rt.rt_flags = RTF_GATEWAY;

sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock < 0) {
    perror("socket");
    return;
}
ret = ioctl(sock, SIOCADDRT, &rt);
if (ret < 0) {
    perror("ioctl SIOCADDRT");
} else {
    printf("default gateway changed to 10.173.210.1\n");
}

close(sock);
```

5.3 如何设置与获取系统 DNS Server 地址

- **Linux平台：**用户可以通过修改 “/etc/ resolv.conf” 文件配置系统的 DNS server 。

- **Huawei LiteOS平台：**不支持上述方式，但是提供了 API 接口，可以让用户去配置或者获取系统的 DNS server 。

  举例如下：

```
#include "lwip/netifapi.h"
#include "lwip/ip_addr.h"

  ip_addr_t dns1, dns2;
  ip_addr_t dns1_g, dns2_g;
  struct in_addr d1, d2;
  struct in_addr d1_g, d2_g;
  int ret;

  inet_aton("8.8.8.8", &d1);
  inet_aton("114.114.114.114", &d2);

  dns1.addr = d1.s_addr;
  dns2.addr = d2.s_addr;

  ret = lwip_dns_setserver(0, &dns1);
  if (ret < 0) {
      printf("set dns server1 failed %d\n", ret);
  }

  ret = lwip_dns_setserver(1, &dns2);
  if (ret < 0) {
      printf("set dns server2 failed %d\n", ret);
  }

  lwip_dns_getserver(0, &dns1_g);
  lwip_dns_getserver(1, &dns2_g);

  d1_g.s_addr = dns1_g.addr;
  d2_g.s_addr = dns2_g.addr;
  printf("dns server1 is %s\n", inet_ntoa(d1_g));
  printf("dns server2 is %s\n", inet_ntoa(d2_g));
```

须知

Huawei LiteOS 当前默认支持两个 DNS server ，如果需要配置更多的 DNS server ，需要修改 **DNS_MAX_SERVERS** 宏，然后重编 lwip 库。另外，系统提供了一个“**dns**”shell 命令，用于可以查询或者设置 DNS server 。

5.4 如何使用 Huawei LiteOS 自带的 DHCP client 服务

- **Linux平台：**busybox 中一般都自带 udhcpc，可用于开启 DHCP client 服务。
- **Huawei LiteOS平台：**不存在 udhcpc，因此无法使用 Linux 的方式开启 DHCP client 。但是 Huawei LiteOS 平台的协议栈自带 DHCP client 实现，用户可以调用协议栈提供的对应接口来使用 Huawei LiteOS 自带的 DHCP client 服务。

  举例如下：

```
#include "lwip/netifapi.h"

int ret, i;
struct netif *netif = netifapi_netif_find_by_name("eth0");

/* start dhcp client on eth0 */
ret = netifapi_dhcp_start(netif);
if (ret < 0) {
    printf("dhcp satrt failed %d\n", ret);
} else {
```

```
    for (i = 0; i < 10; i++) {
        ret = netifapi_dhcp_is_bound(netif);
        if (ret < 0) {
            sleep(1);
            continue;
        } else {
            printf("dhcp ok...\n");
            break;
        }
    }

    if (i == 10) {
        printf("dhcp failed after 10s, stop...\n");
        /* stop dhcp client on eth0 */
        netifapi_dhcp_stop(netif);
        /* clear dhcp client resource */
        netifapi_dhcp_cleanup(netif);
    }
}
```

须知

如果用户需要在 DHCP client 报文中携带私有的选项，可以自行移植 DHCP client ，Huawei LiteOS 不强制要求用户必须使用协议栈自带的DHCP client。

5.5 如何使用 Huawei LiteOS 自带的 DHCP server 服务

- **Linux平台：**busybox 中一般都自带 udhcpd ，可用于开启 DHCP server 服务。
- **Huawei LiteOS平台：**不存在 udhcpd ，因此无法使用 Linux 的方式开启 DHCP server 。但是 Huawei LiteOS 平台的协议栈自带 DHCP server 实现，用户可以调用协议栈提供的对应接口来使用 Huawei LiteOS 自带的 DHCP server 服务。

  举例如下：

```
#include "lwip/netifapi.h"

int ret;
struct netif *netif = netifapi_netif_find_by_name("eth0");

/* start dhcp server */
ret = netifapi_dhcps_start(netif, NULL, 0);
if (ret < 0) {
    printf("dhcp server start failed %d\n", ret);
}

sleep(10);
/* stop dhcp server */
netifapi_dhcps_stop(netif);
```

须知

如果用户需要在 DHCP server 报文中携带私有的选项，可以自行移植 DHCP server ，LiteOS 不强制要求用户必须使用协议栈自带的 DHCP server 。

5.6 如何使用 Huawei LiteOS 自带的 SNTP 进行网络校时

- **Linux 平台：**存在 **ntpdate** 系统命令，可以使用 system(“ntpdate 192.168.1.1”）这种方式来进行网络较时。
- **Huawei LiteOS平台：**由于不支持 system 命令，因此不支持这种方式。 Huawei LiteOS 平台下提供一个 API 接口 lwip_sntp_start ，用于网络校时。需要注意的是，此接口不支持并发调用。另外， Huawei LiteOS 也提供了一个“**ntpdate**”shell命令，使用方法详见 “..\osdrv\opensource\liteos\liteos \doc ”目录下的《Huawei LiteOS lwIP 开发指南》。

5.7 Huawei LiteOS 路由机制与 Linux 的差异

- **Linux平台：**系统存在一张路由表，用于 IP 封包的路由。路由表由内核维护，用户在配置 IP 地址时会自动在路由表中增加对应的路由选项，用户也可以使用 **route** 命令或者 **ip route** 命令，人为的增加路由选项。路由表是整个 IP 层的核心功能，封包接收，发送，转发时都需要查询路由表。发送IP封包时，如果存在多个路由选项可以路由此封包，则根据最长前缀匹配算法（Longest Prefix Match）确定最终使用的路由选项。另外，Linux 支持策略路由，多播路由，封包转发等机制。
- **Huawei LiteOS平台：**并不存在路由表，也没有 **route** 命令或者 **ip route** 命令，用户无法人为的增加路由。不过 Huawei LiteOS 支持通过 ioctl **SIOCADDRT** 修改系统网关，相当于 Linux 下修改系统默认路由。 Huawei LiteOS 下所有的网卡会在链接成一条单链表，路由 IP 封包时，从前往后去遍历这条单链表，只要当前网卡的地址与目的地址在同一个子网内，就确定使用当前网卡发送封包，因此路由的选择有 Linux 是有区别的。另外， Huawei LiteOS 不支持策略路由，多播路由，封包转发等机制。

5.8 Huawei LiteOS 支持的 ioctl 命令，套接字属性，套接字类型

一些经常遇到的问题罗列如下：

- 不支持 **SO_LINGER** ， **SO_REUSEPORT** 套接字属性；
- 不支持设置 UDP 或者 RAW 套接字的发送缓存，支持设置 TCP 套接字的发送缓存，但是设置的范围必须为 [2MSS，256K]。
- 不支持设置 TCP 套接字的接收缓存，支持设置 UDP 或者 RAW 套接字的接收缓存。

链路层套接字，不支持 **SOCK_DGRAM** 与 **SOCK_PACKET** 类型，仅支持 **SOCK_RAW** 类型。

6 USB 模块常见问题

6.1 LiteOS USB 驱动支持规格?

答: LiteOS USB 驱动由于受芯片硬件资源限制，支持的规格也不尽相同，详细可以参考《外围设备驱动操作指南（Huawei LiteOS）》。

6.2 LiteOS USB 驱动 USB 转网口支持哪些型号?

答: 目前 USB 转网口支持5种芯片型号，分别为 Ax88178a、Ax88178、Ax88772、Ax88772A、Ax88772B。

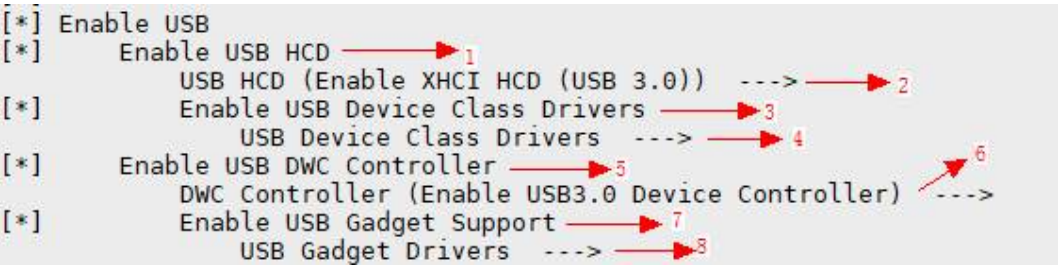
6.3 LiteOS USB 驱动协议是否支持动态切换?

答: USB 驱动支持Device协议之间的动态切换，也支持Host模式和Device模式之间的切换。通过 usb_init 接口入参方式指定需要加载的协议，动态切换时，先调用usb_deinit卸载当前使用的协议，再通过usb_init重新加载待切换的协议。部分协议使用中不支持卸载（如：DFU、UVC、Camera和HID），需要先停用当前业务后，再调用usb_deinit进行卸载。

6.4 LiteOS USB 驱动在 menuconfig 如何配置?

答: 下图是 Huawei LiteOS 中 USB 驱动 menuconfig 配置项截图，可以通过是否选中“Enable USB”打开或关闭 USB 驱动功能。

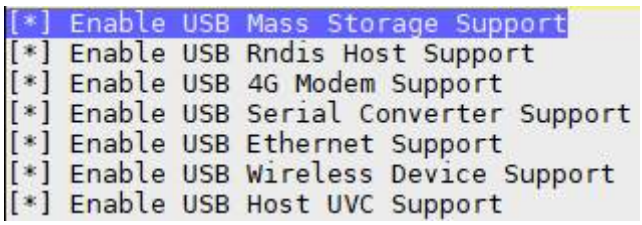
图 6-1 USB 驱动 menuconfig 配置



- **Host 端:**

  如上方“USB 驱动 menuconfig 配置”图中所示，1 表示是否打开 USB 驱动 Host 功能；2 表示 Host 控制器种类，分别有2.0和3.0控制器（可视具体板子的 USB 口类别而定）；3 表示是否开启 Host 模式下的 USB 协议功能；4 就是 Host 模式下的具体 USB 协议功能了，如下图：

  图 6-2 USB 驱动 Host 端配置



  具体有：1、U盘； 2、Host rndis协议； 3、4G Modem； 4、USB转串口； 5、USB转网口； 6、USB-WIFI；7、Host UVC；

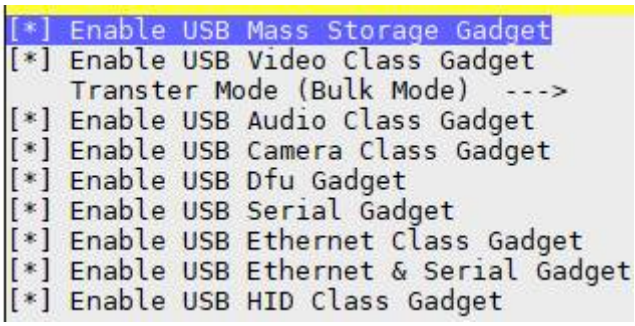
  须知

  Host 模式协议初始化可以调用 usb_init(HOST, 0)，根据业务需要选择协议，也可以全部选择，协议栈会根据插入设备类型自动加载对应驱动，但是依赖的外围组件需要提前初始化。

- **Device端:**

  如上方“USB 驱动 menuconfig 配置”图中所示，5 表示是否打开 USB 驱动 Device 功能；6表示 Device 控制器种类，分别有 Dwc2.0 和3.0（可视具体板子的 USB 口类别而定）；7表示是否开启 Device 模式下的 USB 协议功能；8就是 Device 模式下的具体 USB 协议功能了，如下图：

  图 6-3 USB 驱动 Device 端配置



具体有：1、Device U盘； 2、UVC； 3、UAC； 4、UAV（UVC&UAC）； 5、DFU升级； 6、虚拟串口；7、虚拟网口；8、复合设备（虚拟串口&虚拟网口）；9、HID

须知

Device 模式初始化也是调用 usb_init(DEVICE, xxx)，根据业务诉求开启需要的协议，也可以全部开启。与 Host 模式区别在于需要在初始化时静态指定需要加载的协议，例如xxx表示指定的协议，可以参考 usb_init.h，有对应协议的等效宏；此外相同点就是外围依赖的组件也要事先加载好。

6.5 LiteOS USB YUV 1080P 能够达到多少帧?

答： Huawei LiteOS USB YUV 目前只支持到 YUV 720P，因为 YUV 帧存储非常占用内存，所以综合考虑仅仅支持到 720P（两者内存占用相差5倍多），如果内存上不受限制，改大对应的帧内存，实测 YUV 1080P 在4.2帧左右。

7 编译器常见问题

7.1 ARM 平台下 glibc 提供的 backtrace 函数回溯信息少

原因：ARM 平台 arm-linux-gcc 交叉编译跟 x86 平台的 gcc 默认编译选项存在差异，并且一些优化（比如栈指针优化，尾调用优化）也会导致回溯信息减少。

解决方案：在编译使用backtrace函数的文件或者模块时，添加四个编译选项：

```
-rdynamic -funwind-tables -fno-omit-frame-pointer -fno-optimize-sibling-calls
```

7.2 链接生成的目标文件，shell 命令找不到，文件系统 mount 失败

原因：Makefile链接成目标文件时，缺少链接参数。

解决方案：编译时必须添加编译选项宏 **LITEOS_LDFLAGS**、**LITEOS_TABLES_LDFLAGS**。

8 其他问题

8.1 如何查看当前系统中已创建中断和触发的次数?

答： hwi 命令可以查看当前系统中已创建中断和触发的次数，通过该命令，可以查看创建的中断是否有成功触发。

8.2 系统运行时循环打印 Warning: DO NOT call xxx in software timer callback 及调用栈异常信息

原因： LiteOS 系统中为保证软件定时器任务的定时准确，不允许在软件定时器回调中调用可能造成任务阻塞的接口（例如 LOS_SemPend / LOS_MuxPend ），因此当检测到软件定时器回调中调用此类接口时，系统输出 warning 提示信息及调用栈信息。

解决方案： 建议用户通过调用栈信息找到调用处，对代码进行修改。

8.3 系统不支持 system 函数，怎么处理?

答：LiteOS 系统不支持 system 函数执行相关操作，在使用 system 函数进行功能调用的地方，可以直接调用该功能接口函数实现功能调用。

8.4 用户自己注册 shell 命令导致 shell 任务栈溢出

原因：用户在使用或者测试时，有时会通过增加一个 shell 命令去执行，由于系统配置的 shell 任务栈不是很大，因此如果用户将所有操作都放在 shell 命令中直接执行，很容易造成 shell 任务栈溢出。

解决方案：此种情况下用户可在 shell 命令入口函数中创建任务来执行自己的实际操作。

8.5 用户移植第三方或私有库编译通过，但是运行出错

原因：LiteOS 有自己的 C 库，需要使用自己的 C 库头文件进行编译。

解决方案：系统镜像 bin 文件编译 Makefile 必须 include 根目录下的 config.mk 文件，并使用其中的 **LITEOS_CFLAGS** 或 **LITEOS_CXXFLAGS** 编译选项，示例如下：

```
LITEOSTOPDIR ?= ../..
SAMPLE_OUT = .
include $(LITEOSTOPDIR)/config.mk
RM = -rm -rf
LITEOS_LIBDEPS := --start-group $(LITEOS_LIBDEP) --end-group
SRCS = $(wildcard sample.c)
OBJS = $(patsubst %.c,$(SAMPLE_OUT)/%.o,$(SRCS))
all: $(OBJS)
clean:
@$(RM) *.o sample *.bin *.map *.asm
$(OBJS): $(SAMPLE_OUT)/%.o : %.c
$(CC) $(LITEOS_CFLAGS) -c $< -o $@
$(LD) $(LITEOS_LDFLAGS) -uinit_jffspar_param --gc-sections -Map=$(SAMPLE_OUT)/sample.map -o $
(SAMPLE_OUT)/sample ./$@ $(LITEOS_LIBDEPS) $(LITEOS_TABLES_LDFLAGS)
$(OBJCOPY) -O binary $(SAMPLE_OUT)/sample $(SAMPLE_OUT)/sample.bin
$(OBJDUMP) -d $(SAMPLE_OUT)/sample >$(SAMPLE_OUT)/sample.asm
```

8.6 串口端和 telnet 端同时进行调测，调测代码均需要获取标准输入时，某一端输入无响应

原因：LiteOS属于实时操作系统，没有进程概念。在两个端口同时获取标准输入时，等同于同一进程下的两个线程监听标准输入。由于getchar等获取输入函数具有竞争关系，系统会对标准输入加锁保护，因此串口端和telnet端同时获取标准输入时，系统只会在先请求输入的端口进行响应。

解决方案：避免在串口端和telnet端同时使用getchar，gets，scanf，fgets，vfscanf等标准输入接口获取输入。

8.7 LiteOS 不支持 TLS，使用__thread 编译器内置关键字会触发异常

原因：TLS，即Thread Local Storage。LiteOS系统启动后一直运行在内核态，而__thread关键字一般用在用户态程序中。编译器会在使用处打桩，访问el0或pl0模式下的寄存器。LiteOS未对该寄存器进行相关配置，其内容无效，且在linux内核代码中添加该关键字也会出现运行异常。

解决方案：规避使用TLS的场景。

8.8 用户移植某算法程序到 LiteOS, 经测试在 LiteOS 系统上运行时间比在 Linux 系统上运行时间长

原因：LiteOS编译选项中默认包含-fno-builtin，此选项的作用是对非_builtin_前缀的函数不识别为内置函数，以防止LiteOS系统函数被识别为内置函数而被特殊代码替换。Linux系统上gcc可能会生成特殊代码去更高效的处理内置函数，从而算法运行时间可能减少。

解决方案：若用户根据算法程序，明确要将某函数识别为内置函数（一般是math数学函数），可在算法源文件中通过如下宏定义处理（以funcA为例）：

```
#define funcA（x） __builtin_funcA（x）
```

8.9 cat 等打印命令打印大量数据时，可能存在数据打印不全

原因：当系统开启console时，诸如dprintf等打印函数会通过console输出，console通过ringbuf来进行数据缓冲，系统支持ringbuf的size可配，默认为4K大小，实际由配置项LOSCFG_CONSOLE_RINGBUFF_SIZE决定，可满足用户一般的打印需求。数据打印不全的原因有以下两种情况：

1. 如果使用cat等打印命令，打印大量数据，首先需确保ringbuf可以容纳所打印的数据，否则超出缓冲区大小的数据将会被丢弃，从而造成打印数据不全。
2. console的打印输出是通过任务调度实现，用户业务中打印任务的优先级如果高于console打印任务的优先级，在打印大量数据时，可能会因为console的任务得不到及时调度，而使缓冲区中数据溢出，造成数据打印不全。

解决方案：可以适当增大ringbuf的size，来缓解打印大量数据时缓冲区溢出风险。