第3章 典型物联网操作系统

CHAPTER 3

操作系统是物联网时代的战略制高点，ARM、Google、Microsoft、华为、阿里巴巴等国内外著名的IT企业纷纷推出物联网操作系统，整个产业呈现出群雄逐鹿的壮观景象。传统的嵌入式系统公司也不甘示弱，纷纷通过开源和并购策略推出面向物联网软件平台，比如Intel、风河、芯科和Micrium。在一轮新的产业浪潮中，国内创业公司也走在风口浪尖上，他们纷纷推出自己的物联网操作系统，比如庆科、Ruff和RT-Thread。

物联网操作系统是架设在物联网设备上的，物联网设备种类众多，结构迥异，但是从计算资源和存储能力上大体可以分为两类：面向运算、存储能力较强的嵌入式设备，例如，工业网关、采集器等；面向对功耗、存储、计算资源有苛刻限制的终端设备，例如，单片机、芯片、模组。因此物联网操作系统从系统体积和功能应用上大致也可以分为两类：一类以Android Things，Windows 10 IoT等为典型代表，运行内存较大，运算能力较强；另一类是轻量级物联网操作系统，典型代表是华为LiteOS，华为官方宣称LiteOS内核可精简至10KB左右。

由于Linux在嵌入式领域的强势地位，有相当数量的物联网操作系统将Linux内核作为操作系统的基础内核裁剪，优化设计后进行使用，比如Android Things、Ostro等。也有在实时嵌入式系统中广泛应用的实时操作系统在丰富组件和服务的支持下重新在物联网领域内得到广大爱好者喜爱和使用的例子，如RT-Thread。RT-Thread具有庞大的、活跃度较高的社区开发群体。

本章介绍目前的主流物联网操作系统。

3.1 RT-Thread

RT-Thread(Real Time-Thread)是一个集实时操作系统(RTOS)内核、中间件组件和开发者社区于一体的技术平台，RT-Thread也是一个组件完整丰富、高度可伸缩、简易开发、超低功耗、高安全性的物联网操作系统。RT-Thread具备一个IoT OS平台所需的所有关键组件，例如，图形用户界面GUI、网络协议栈、安全传输、低功耗组件等。经过十多年的累积、发展，RT-Thread已经拥有一个国内最大的嵌入式开源社区，同时被广泛应用于能源、车载、医疗、消费电子等多个行业，累积装机量超过两千万台，成为国人自主开发、国内最成熟稳定和装机量最大的开源RTOS。

RT-Thread拥有良好的软件生态，支持市面上所有主流的编译工具如GCC、Keil、IAR

等,工具链完善、友好,支持各类标准接口,如 POSIX、CMSIS、C++应用环境、JavaScript 执行环境等,方便开发者移植各类应用程序。商用支持所有主流微控制器架构,如 ARM Cortex-M/R/A、MIPS、x86、Xtensa、C-Sky、RISC-V,几乎支持市场上所有主流的 MCU 和 WiFi 芯片。RT-Thread 主要采用 C 语言编写,浅显易懂,方便移植。它把面向对象的设计方法应用到实时系统设计中,使得代码风格优雅、架构清晰、系统模块化并且可裁剪性非常好。针对资源受限的 MCU 系统,可通过方便易用的工具,裁剪出仅需要 3KB Flash、1.2KB RAM 内存资源的 NANO 版本(NANO 是 RT-Thread 官方于 2017 年 7 月发布的一个极简版内核);而对于资源丰富的物联网设备,RT-Thread 又能使用在线的软件包管理工具,配合系统配置工具实现直观快速的模块化裁剪,无缝地导入丰富的软件功能包,实现类似 Android 的图形界面及触摸滑动效果、智能语音交互效果等复杂功能。

RT-Thread 系统完全开源,3.1.0 及以前的版本遵循 GPL V2 +开源许可协议。从 3.1.0 以后的版本遵循 Apache License 2.0 开源许可协议,实时操作系统内核及所有开源组件可以免费在商业产品中使用,不需要公布应用程序源码,没有潜在商业风险。

3.1.1 RT-Thread 的架构

RT-Thread 与其他很多 RTOS 之间的主要区别之一是,它不仅是一个实时内核,还具备丰富的中间层组件,它更是一个物联网操作系统。RT-Thread 的架构如图 3-1 所示。

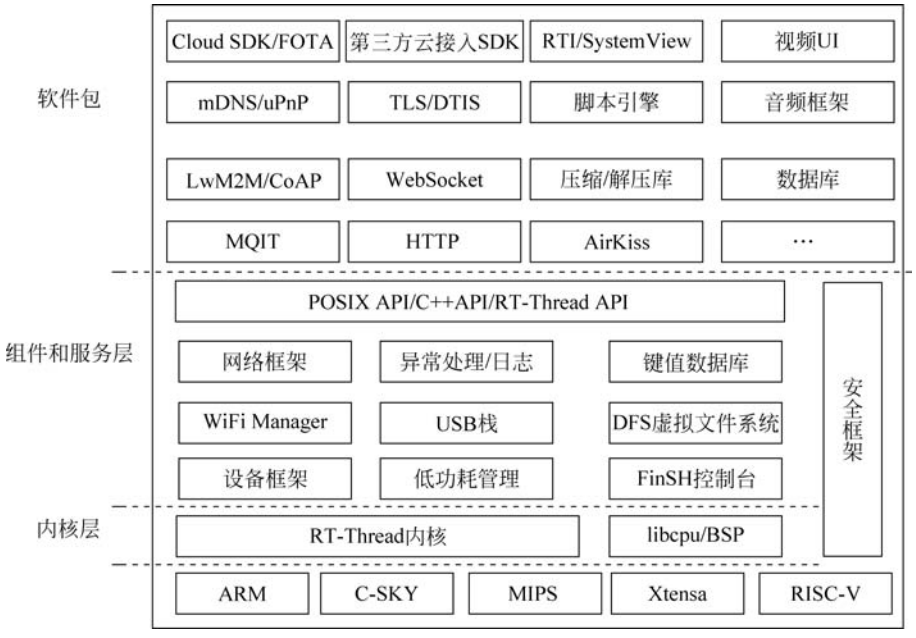

图 3-1 RT-Thread 的架构

具体来说,RT-Thread 包括以下部分。

(1) 内核层:RT-Thread 内核,是 RT-Thread 的核心部分,包括了内核系统中对象的实现,例如,多线程及其调度、信号量、邮箱、消息队列、内存管理、定时器等;与硬件密切相关的 libcpu/BSP(芯片移植相关文件/板级支持包),由外设驱动和 CPU 移植构成。从图 3-1 中可以发现,RT-Thread 支持主流嵌入式架构(如 ARM、MIPS、RISC-V 等)。

（2）组件与服务层：组件是基于 RT-Thread 内核之上的上层软件，例如，虚拟文件系统、FinSH 命令行界面、网络框架、设备框架等。组件与服务层采用模块化设计，做到组件内部高内聚，组件之间低耦合。

（3）RT-Thread 软件包：该软件包运行于 RT-Thread 物联网操作系统平台上，面向不同应用领域的通用软件组件，由描述信息、源代码或库文件组成。RT-Thread 提供了开放的软件包平台，这里存放了官方提供或开发者提供的软件包，该平台为开发者提供了众多可重用软件包的选择，这也是 RT-Thread 生态的重要组成部分。软件包生态对于一个操作系统的选择至关重要，因为这些软件包具有很强的可重用性，模块化程度很高，极大地方便应用开发者在最短时间内，打造出自己想要的系统。RT-Thread 已经支持的软件包数量已经达到 60 多种，下面给出一些例子。

- 物联网相关的软件包：Paho MQTT、WebClient、Mongoose、WebTerminal 等。
- 脚本语言相关的软件包：目前支持 JerryScript、MicroPython。
- 多媒体相关的软件包：Openmv、Mupdf。
- 工具类软件包：CmBacktrace、EasyFlash、EasyLogger、SystemView。
- 系统相关的软件包：RTGUI、Persimmon UI、Lwext4、Partition、SQLite 等。
- 外设库与驱动类软件包：RealTek RTL8710BN SDK。

3.1.2　RT-Thread 内核

操作系统的灵魂是内核，内核是操作系统最基础也是最重要的部分，内核为系统其他部分提供系统服务。图 3-2 为 RT-Thread 内核组成图，从图 3-2 中可以发现，内核处于硬件层之上，内核部分包括内核库、实时内核实现。

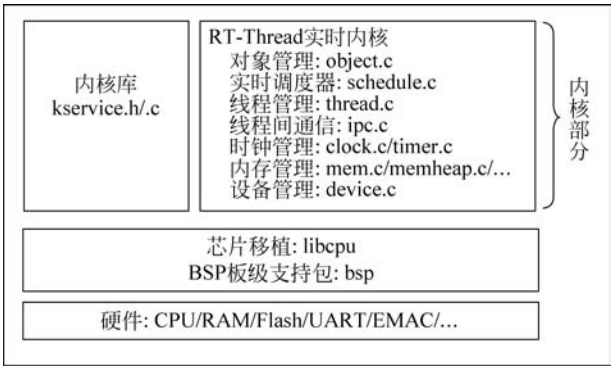

图 3-2　RT-Thread 内核组成

内核的正常工作离不开底层硬件和硬件抽象层的支持。RT-Thread 支持多种 CPU、存储器等设备。硬件抽象层最常见的表现形式是板级支持包 bsp。有关于板级支持包的知识已经在 1.2.1 节中予以说明。

内核库(kernel library)是为了保证内核能够独立运行的一套小型的类似 C 库的函数实现子集。根据编译器的不同，自带 C 库的情况也会有些不同，当使用 GNU GCC 编译器时，会携带更多的标准 C 库实现。RT-Thread Kernel Service Library 仅提供内核用到的一小部分 C 库函数实现，为了避免与标准 C 库重名，在这些函数前都会添加上 rt_前缀。

实时内核(real-time kernel)的实现包括对象管理、线程管理及调度器、线程间通信管理、时钟管理、内存管理和设备管理等。最小的实时内核仅需 3KB ROM、1.2KB RAM 空间。

1) 线程调度

线程是 RT-Thread 操作系统中最小的调度单位,线程调度算法是基于优先级的全抢占式多线程调度算法,即在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码是不可抢占的之外,系统的其他部分都是可以抢占的,包括线程调度器自身。实时内核支持 256 个线程优先级(也可通过配置文件更改为最大支持 32 个或 8 个线程优先级,针对 STM32 默认配置是 32 个线程优先级),0 优先级代表最高优先级,最低优先级留给空闲线程使用;同时它也支持创建多个具有相同优先级的线程,相同优先级的线程间采用时间片轮转调度算法进行调度,使每个线程运行相应时间;另外,调度器在寻找那些处于就绪状态的具有最高优先级的线程时,所经历的时间是恒定的,系统也不限制线程数量的多少,线程数目只与硬件平台的具体内存相关。

2) 内存管理

RT-Thread 支持静态内存池管理及动态内存堆管理。当静态内存池具有可用内存时,系统对内存块分配的时间将是恒定的;当静态内存池为空时,系统将申请内存块的线程挂起或阻塞掉(即线程等待一段时间后仍未获得内存块就放弃申请并返回,或者立刻返回。等待的时间取决于申请内存块时设置的等待时间参数),当其他线程释放内存块到内存池时,如果有挂起的待分配内存块的线程存在,则系统会将这个线程唤醒。动态内存堆管理模块在系统资源不同的情况下,分别提供了面向小内存系统的内存管理算法及面向大内存系统的 SLAB 内存管理算法。

RT-Thread 还有一种动态内存堆管理叫作 memheap(见 2.2.4 节中的介绍),适用于系统含有多个地址可不连续的内存堆。使用 memheap 可以将多个内存堆“粘贴”在一起,让用户操作起来像是在操作一个内存堆。

3) 时钟管理

RT-Thread 的时钟管理以时钟节拍为基础,时钟节拍是 RT-Thread 操作系统中最小的时钟单位。RT-Thread 的定时器提供两类定时器机制:第一类是单次触发定时器,这类定时器在启动后只会触发一次定时器事件,然后定时器自动停止;第二类是周期触发定时器,这类定时器会周期性地触发定时器事件,直到用户手动停止定时器,否则将永远持续执行下去。

另外,根据超时函数执行时所处的上下文环境,RT-Thread 的定时器可以设置为 HARD_TIMER 模式或者 SOFT_TIMER 模式。

通常使用定时器定时回调函数(即超时函数),完成定时服务。用户根据自己对定时处理的实时性要求选择合适类型的定时器。

4) 线程间同步

RT-Thread 实时内核采用信号量、互斥量与事件集实现线程间同步。线程通过对信号量、互斥量的获取与释放进行同步;互斥量采用优先级继承的方式解决了实时系统常见的优先级翻转问题。线程同步机制支持线程按优先级等待或按先进先出方式获取信号量或互斥量。线程通过对事件的发送与接收进行同步;事件集支持多事件的“或触发”和“与触

发”，适合于线程等待多个事件的情况。

5）线程间通信

RT-Thread 支持邮箱和消息队列等通信机制。邮箱中一封邮件的长度固定为 4B 大小；消息队列能够接收不固定长度的消息，并把消息缓存在自己的内存空间中。邮箱效率较消息队列更为高效。邮箱和消息队列的发送动作可安全用于中断服务例程中。通信机制支持线程按优先级等待或按先进先出方式获取。

6）I/O 设备管理

RT-Thread 实时内核将 PIN、I^2C、SPI、USB、UART 等作为外设，统一通过设备注册完成。实现了按名称访问的设备管理子系统，可按照统一的 API 界面访问硬件设备。在设备驱动接口上，根据嵌入式系统的特点，对不同的设备可以挂载相应的事件。当设备事件触发时，由驱动程序通知上层的应用程序。

下面介绍 RT-Thread 中的几个重要概念。

1. RT-Thread 内核对象模型

RT-Thread 在开发中借鉴了 Linux 内核对象的思想，并且根据自身特性和需求发展了 RT-Thread 内核对象。在 Linux 2.6 内核中，引入了一种称为“内核对象”(kernel object)的设备管理机制，该机制是基于一种底层数据结构，通过这个数据结构，可以使所有设备在底层都具有一个公共接口，以便于设备或驱动程序的管理和组织。kobject 在 Linux 2.6 内核中是由 struct kobject 表示的。通过这个数据结构使所有设备在底层都具有统一的接口，kobject 提供基本的对象管理，是构成 Linux 2.6 设备模型的核心结构。

同理，RT-Thread 内核也采用了面向对象的设计思想进行设计，系统级的基础设施都是一种内核对象，例如，线程、信号量、互斥量、定时器等。内核对象分为两类：静态内核对象和动态内核对象，静态内核对象在系统启动后在程序中初始化；动态内核对象则是从内存堆中创建的，而后手动初始化。

以下代码是一个关于静态线程和动态线程的例子。

```
static struct rt_thread thread1;                    /* 线程 1 的对象和运行时用到的栈 */
static rt_uint8_t thread1_stack[512];
void thread1_entry(void* parameter)                 /* 线程 1 入口 */
{
    int i;
    while (1)
    {
        for (i = 0; i < 10; i ++)
        {
            rt_kprintf("%d\n", i);
            rt_thread_mdelay(100);                  /* 延时 100ms */
        }
    }
}
void thread2_entry(void* parameter)                 /* 线程 2 入口 */
{
    int count = 0;
    while (1)
```

```
    {
        rt_kprintf("Thread2 count: %d\n", ++count);
        rt_thread_mdelay(70);                    /* 延时 70ms */
    }
}
int thread_sample_init()                         /* 线程例程初始化 */
{
        rt_thread_t thread2_ptr;
        rt_err_t result;
                                                 /* 初始化线程 1,线程的入口是 thread1_entry,参
                                                    数是 RT_NULL,线程栈是 thread1_stack,优先级
                                                    是 188,时间片是 8 个 OS Tick */
        result = rt_thread_init(&thread1,
                                "thread1",
                                thread1_entry, RT_NULL,
                                &thread1_stack[0], sizeof(thread1_stack),
                                188, 8);
        if (result == RT_EOK) rt_thread_startup(&thread1);
        thread2_ptr = rt_thread_create("thread2",
                               thread2_entry, RT_NULL,
                               512, 250, 25);    /* 创建线程 2, 线程的入口是 thread2_entry,
                                                    参数是 RT_NULL,栈空间是 512,优先级是
                                                    250,时间片是 25 个 OS Tick */
        if (thread2_ptr != RT_NULL) rt_thread_startup(thread2_ptr);
        return 0;
}
```

在这个例子中,thread1 是一个静态线程对象,而 thread2 是一个动态线程对象。thread1 对象所占据的内存空间(包括线程控制块 thread1 与栈空间 thread1_stack)都是编译时决定的,因为代码中都不存在初始值,所以统一放在未初始化数据段中。thread2 运行中用到的空间都是动态分配的,包括线程控制块(thread2_ptr 指向的内容)和栈空间。

静态对象会占用 RAM 空间,不依赖于内存堆管理器,内存分配时间确定。动态对象则依赖于内存堆管理器,运行时申请 RAM 空间,当对象被删除后,占用的 RAM 空间被释放。这两种方式各有利弊,可以根据实际环境需求选择具体使用方式。

2. 内核对象管理架构

RT-Thread 采用内核对象管理系统来访问和管理所有内核对象,内核对象包含了内核中的绝大部分设施,这些内核对象可以是静态分配的静态对象,也可以是从系统内存堆中分配的动态对象。

通过这种内核对象的设计方式,RT-Thread 做到了不依赖于具体的内存分配方式,系统的灵活性得到极大的提高。

RT-Thread 内核对象包括线程、信号量、互斥量、事件、邮箱、消息队列、定时器、内存池和设备驱动等。对象容器中包含了每类内核对象的信息,包括对象类型、大小等。对象容器给每类内核对象分配了一个链表,所有的内核对象都被链接到该链表上,如 RT-Thread 的内核对象容器及链表如图 3-3 所示。

内核对象容器包含了众多的对象信息,其数据结构如下所示。

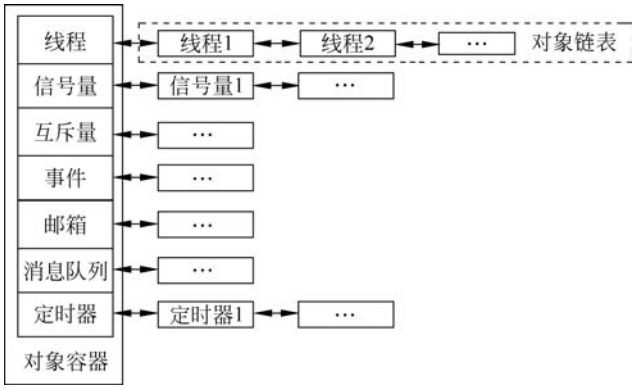

图 3-3　RT-Thread 的内核对象容器及链表

```
struct rt_object_information
{
    enum rt_object_class_type type;          /* 对象类型 */
    rt_list_t object_list;                   /* 对象链表 */
    rt_size_t object_size;                   /* 对象大小 */
};
```

一类对象由一个 rt_object_information 结构体管理，每一个这类对象的具体实例都通过链表的形式挂载在 object_list 上。而这一类对象的内存块尺寸由 object_size 标识出来（每一类对象的具体实例所占有的内存块大小都是相同的）。

图 3-4 显示了 RT-Thread 中各类内核对象的派生和继承关系。rt_object 是各类内核对象的基类，它提供基本的对象管理，是构成 RT-Thread 内核对象模型的核心结构。rt_object 也叫内核对象控制块，它的数据结构如下所示。

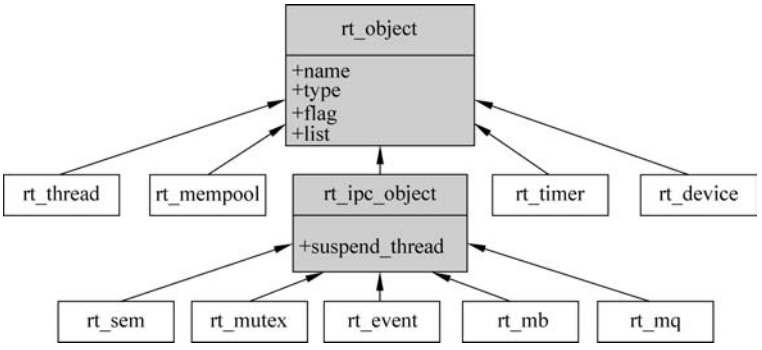

图 3-4　内核对象的派生和继承

```
struct rt_object
{
    char name[RT_NAME_MAX];                  /* 内核对象名称 */
    rt_uint8_t type;                         /* 内核对象类型 */
    rt_uint8_t flag;                         /* 内核对象参数 */
    rt_list_t list;                          /* 内核对象管理链表 */
};
```

对于每一种具体内核对象和对象控制块,除了基本结构外,还有自己的扩展属性(私有属性),例如,对于线程控制块 rt_thread,在基类对象基础上进行扩展,增加了线程状态、优先级等属性。这些属性在基类对象的操作中不会用到,只有在与具体线程相关的操作中才会使用。因此从面向对象的观点,可以认为每一种具体对象是抽象对象的派生,继承了基本对象的属性并在此基础上扩展了与自己相关的属性。

在对象管理模块中,定义了通用的数据结构,用来保存各种对象的共同属性,各种具体对象只需要在此基础上加上自己的某些特别的属性,就可以清楚地表示自己的特征。

这种设计方法的优点有:

(1) 提高了系统的可重用性和扩展性,增加新的对象类别很方便,只需要继承通用对象的属性再加少量扩展。

(2) 提供统一的对象操作方式,简化了各种具体对象的操作,提高了系统的可靠性。

图 3-4 中由对象控制块 rt_object 派生出来的有线程对象、内存池对象、定时器对象、设备对象和 IPC 对象(Inter-Process Communication,简称 IPC,即进程间通信。在 RT-Thread 实时操作系统中,IPC 对象的作用是进行线程间同步与通信);由 IPC 对象派生出信号量、互斥量、事件、邮箱与消息队列、信号等对象。

目前内核对象支持的类型如下。

```
enum rt_object_class_type
{
    RT_Object_Class_Thread = 0,              /* 对象为线程类型 */
#ifdef RT_USING_SEMAPHORE
    RT_Object_Class_Semaphore,               /* 对象为信号量类型 */
#endif
#ifdef RT_USING_MUTEX
    RT_Object_Class_Mutex,                   /* 对象为互斥量类型 */
#endif
#ifdef RT_USING_EVENT
    RT_Object_Class_Event,                   /* 对象为事件类型 */
#endif
#ifdef RT_USING_MAILBOX
    RT_Object_Class_MailBox,                 /* 对象为邮箱类型 */
#endif
#ifdef RT_USING_MESSAGEQUEUE
    RT_Object_Class_MessageQueue,            /* 对象为消息队列类型 */
#endif
#ifdef RT_USING_MEMPOOL
    RT_Object_Class_MemPool,                 /* 对象为内存池类型 */
#endif
#ifdef RT_USING_DEVICE
    RT_Object_Class_Device,                  /* 对象为设备类型 */
#endif
    RT_Object_Class_Timer,                   /* 对象为定时器类型 */
#ifdef RT_USING_MODULE
    RT_Object_Class_Module,                  /* 对象为模块 */
#endif
    RT_Object_Class_Unknown,                 /* 对象类型未知 */
    RT_Object_Class_Static = 0x80            /* 对象为静态对象 */
};
```

在使用一个未初始化的静态对象前，必须先对其初始化。初始化对象使用以下接口：

```
void rt_object_init(struct  rt_object *   object ,
                          enum rt_object_class_type  type,
                          const char *  name)
```

当调用这个函数进行对象初始化时，系统会把这个对象放置到对象容器中管理，即初始化对象的一些参数，然后把这个对象节点插入对象容器的对象链表中，对该函数的输入参数的描述如表 3-1 所示。

表 3-1　初始化函数参数

参数	描　　述
object	需要初始化的对象指针，它必须指向具体的对象内存块，而不能是空指针或野指针
type	对象的类型，必须是 rt_object_class_type 枚举类型中列出的除 RT_Object_Class_Static 以外的类型（对于静态对象，或使用 rt_object_init 接口进行初始化的对象，系统会把它标识成 RT_Object_Class_Static 类型）
name	对象的名字。每个对象可以设置一个名字，这个名字的最大长度由 RT_NAME_MAX 指定，并且系统不关心它是不是以'\0'作为终结符

从内核对象管理器中脱离一个对象。脱离对象使用以下接口。

```
voidrt_object_detach(rt_object_t object);
```

调用该接口，可使得一个静态内核对象从内核对象容器中脱离出来，即从内核对象容器链表上删除相应的对象节点。对象脱离后，对象占用的内存并不会被释放。

上述描述的都是对象初始化、脱离的接口，都是在已经有面向对象内存块的情况下，而动态的对象则可以在需要时申请，不需要时释放出内存空间给其他应用使用。申请分配新的对象可以使用以下接口。

```
rt_object_t rt_object_allocate(enum  rt_object_class_type type ,
                                     const  char *  name)
```

在调用以上接口时，系统首先需要根据对象类型获取对象信息（特别是对象类型的大小信息，以便系统能够分配正确大小的内存数据块），而后从内存堆中分配对象所对应的内存空间，然后再对该对象进行必要的初始化，最后将其插入它所在的对象容器链表中。对该函数的输入参数的描述如表 3-2 所示。

表 3-2　申请分配对象函数参数

参数	描　　述
type	分配对象的类型，只能是 rt_object_class_type 中除 RT_Object_Class_Static 以外的类型。并且经过这个接口分配出来的对象类型是动态的，而不是静态的
name	对象的名字。每个对象可以设置一个名字，这个名字的最大长度由 RT_NAME_MAX 指定，并且系统不关心它是不是以'\0'作为终结符

续表

参数	描　　述
返回	—
分配成功的对象句柄	分配成功
RT_NULL	分配失败

对于一个动态对象，当不再使用时，可以调用如下接口删除对象，并释放相应的系统资源。

```
void rt_object_delete(rt_object_t object);
```

当调用以上接口时，首先从对象容器链表中脱离对象，然后释放对象所占用的内存。参数 object 描述了对象的句柄。

3. RT-Thread 内核配置示例

RT-Thread 的一个重要特性是高度可裁剪性，支持对内核进行精细调整，对组件进行灵活拆卸。

配置主要通过修改工程目录下的 rtconfig.h 文件进行，用户可以通过打开/关闭该文件中的宏定义对代码进行条件编译，最终达到系统配置和裁剪的目的。

1) RT-Thread 内核部分

```
#define RT_NAME_MAX 8                  /* 表示内核对象的名称的最大长度,若代码中对象名称的
                                          最大长度大于宏定义的长度,多余的部分将被截掉 */
#define RT_ALIGN_SIZE 4                /* 字节对齐时设定对齐的字节个数.常使用 ALIGN(RT_
                                          ALIGN_SIZE) 进行字节对齐 */
#define RT_THREAD_PRIORITY_MAX 32      /* 定义系统线程优先级数; 通常用 RT_THREAD_PRIORITY_
                                          MAX-1 定义空闲线程的优先级 */
#define RT_TICK_PER_SECOND 100         /* 定义时钟节拍,为 100 时表示 100 个 tick 每秒,一个
                                          tick 为 10ms */
#define RT_USING_OVERFLOW_CHECK        /* 检查栈是否溢出,未定义则关闭 */
#define RT_DEBUG                       /* 定义该宏开启 debug 模式,未定义则关闭 */
#define RT_DEBUG_INIT 0                /* 开启 debug 模式时: 该宏定义为 0 时表示关闭打印组件
                                          初始化信息,定义为 1 时表示启用 */
#define RT_DEBUG_THREAD 0              /* 开启 debug 模式时: 该宏定义为 0 时表示关闭打印线程
                                          切换信息,定义为 1 时表示启用 */
#define RT_USING_HOOK                  /* 定义该宏表示开启钩子函数的使用,未定义则关闭 */
#define IDLE_THREAD_STACK_SIZE 256     /* 定义了空闲线程的栈大小 */
```

2) 线程间同步与通信部分

该部分会使用到的对象有信号量、互斥量、事件、邮箱、消息队列、信号等。

```
#define RT_USING_SEMAPHORE             /* 定义该宏可开启信号量的使用,未定义则关闭 */
#define RT_USING_MUTEX                 /* 定义该宏可开启互斥量的使用,未定义则关闭 */
#define RT_USING_EVENT                 /* 定义该宏可开启事件集的使用,未定义则关闭 */
#define RT_USING_MAILBOX               /* 定义该宏可开启邮箱的使用,未定义则关闭 */
```

```
#define RT_USING_MESSAGEQUEUE          /* 定义该宏可开启消息队列的使用,未定义则关闭 */
#define RT_USING_SIGNALS               /* 定义该宏可开启信号的使用,未定义则关闭 */
```

3）内存管理部分

```
#define RT_USING_MEMPOOL             /* 开启静态内存池的使用 */
#define RT_USING_MEMHEAP             /* 定义该宏可开启两个或两个以上内存堆拼接的使用,未
                                        定义则关闭 */
#define RT_USING_SMALL_MEM           /* 开启小内存管理算法 */
#define RT_USING_SLAB                /* 关闭 SLAB 内存管理算法 */
#define RT_USING_HEAP                /* 开启堆的使用 */
```

4）内核设备对象

```
#define RT_USING_DEVICE                  /* 表示开启了系统设备的使用 */
#define RT_USING_CONSOLE                 /* 定义该宏可开启系统控制台设备的使用,未定义则
                                            关闭 */
#define RT_CONSOLEBUF_SIZE 128           /* 定义控制台设备的缓冲区大小 */
#define RT_CONSOLE_DEVICE_NAME "uart1"   /* 控制台设备的名称 */
```

5）自动初始化方式

```
#define RT_USING_COMPONENTS_INIT         /* 定义该宏开启自动初始化机制,未定义则关闭 */
#define RT_USING_USER_MAIN               /* 定义该宏开启设置应用入口为 main 函数 */
#define RT_MAIN_THREAD_STACK_SIZE 2048   /* 定义 main 线程的栈大小 */
```

6）FinSH

```
#define RT_USING_FINSH               /* 定义该宏可开启系统 FinSH 调试工具的使用,未定义
                                        则关闭 */
#define FINSH_THREAD_NAME "tshell"   /* 开启系统 FinSH 时: 将该线程名称定义为 tshell */
#define FINSH_USING_HISTORY          /* 开启系统 FinSH 时: 使用历史命令 */
#define FINSH_HISTORY_LINES 5        /* 开启系统 FinSH 时: 对历史命令行数的定义 */
#define FINSH_USING_SYMTAB           /* 开启系统 FinSH 时: 定义该宏开启使用 Tab 键,未定
                                        义则关闭 */
#define FINSH_THREAD_PRIORITY 20     /* 开启系统 FinSH 时: 定义该线程的优先级 */
#define FINSH_THREAD_STACK_SIZE 4096 /* 开启系统 FinSH 时: 定义该线程的栈大小 */
#define FINSH_CMD_SIZE 80            /* 开启系统 FinSH 时: 定义命令字符长度 */
#define FINSH_USING_MSH              /* 开启系统 FinSH 时: 定义该宏开启 MSH 功能 */
#define FINSH_USING_MSH_DEFAULT      /* 开启系统 FinSH 时: 开启 MSH 功能时,定义该宏默认
                                        使用 MSH 功能 */
#define FINSH_USING_MSH_ONLY         /* 开启系统 FinSH 时: 定义该宏,仅使用 MSH 功能 */
```

7）关于 MCU

```
#define STM32F103ZE              /* 定义该工程使用的 MCU 为 STM32F103ZE; 系统通过对芯片
                                    类型的定义,定义芯片的引脚 */
#define RT_HSE_VALUE 8000000     /* 定义时钟源频率 */
#define RT_USING_UART1           /* 定义该宏开启 UART1 的使用 */
```

在实际应用中,系统配置文件 rtconfig.h 是由配置工具自动生成的,无须手动更改。

3.1.3 线程管理

在 RT-Thread 中,线程是实现任务的载体,它是 RT-Thread 中最基本的调度单位,线程描述了一个任务执行的运行环境,也描述了这个任务所处的优先等级。RT-Thread 线程管理的主要功能是对线程进行管理和调度,系统中总共存在两类线程,分别是系统线程和用户线程。系统线程是由 RT-Thread 内核创建的线程,用户线程是由应用程序创建的线程,这两类线程都会从内核对象容器中分配线程对象。当线程被删除时,也会被从对象容器中删除,如图 3-5 所示。每个线程都有重要的属性,如线程控制块、线程栈、入口函数等。

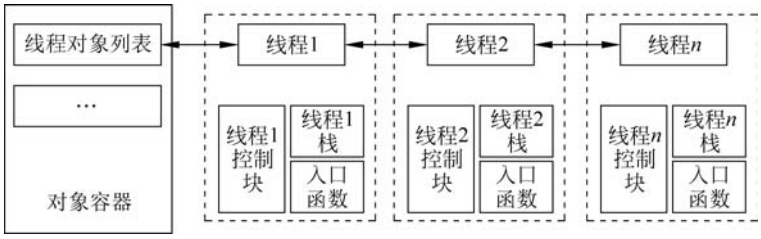

图 3-5 线程管理

RT-Thread 的线程调度器是抢占式的,其主要工作就是从就绪线程列表中查找最高优先级线程,保证最高优先级的线程能够被运行,最高优先级的任务一旦就绪,总能得到 CPU 的使用权。如果一个运行着的线程使一个比它优先级高的线程满足了运行条件,那么当前线程的 CPU 使用权就被剥夺了,或者说被让出了,高优先级的线程立刻得到了 CPU 的使用权。

如果是中断服务程序使一个高优先级的线程满足运行条件,那么中断完成时,被中断的线程挂起,优先级高的线程开始运行。RT-Thread 最大支持 256 个线程优先级(0～255),数值越小的优先级越高,0 为最高优先级。在一些资源比较紧张的系统中,可以根据实际情况选择只支持 8 个或 32 个优先级的系统配置;对于 ARM Cortex-M 系列,普遍采用 32 个优先级。最低优先级默认分配给空闲线程使用,用户一般不使用。

当调度器调度线程切换时,先将当前线程上下文保存起来,当再切回到这个线程时,线程调度器将该线程的上下文信息恢复。

在 RT-Thread 中,线程控制块由结构体 struct rt_thread 表示,线程控制块是操作系统用于管理线程的一个数据结构,它会存放线程的一些信息,例如,优先级、线程名称、线程状态等,也包含线程与线程之间连接用的链表结构、线程等待事件集合等,详细定义如下。

```
struct rt_thread /* 线程控制块 */
{
    /* rt 对象 */
    char name[RT_NAME_MAX];                    /* 线程名称 */
    rt_uint8_t type;                           /* 对象类型 */
    rt_uint8_t flags;                          /* 标志位 */
    rt_list_t list;                            /* 对象列表 */
    rt_list_t tlist;                           /* 线程列表 */
```

```
    /* 栈指针与入口指针 */
    void      * sp;                                  /* 栈指针 */
    void      * entry;                               /* 入口函数指针 */
    void      * parameter;                           /* 参数 */
    void      * stack_addr;                          /* 栈地址指针 */
    rt_uint32_t stack_size;                          /* 栈大小 */
    /* 错误代码 */
    rt_err_t error;                                  /* 线程错误代码 */
    rt_uint8_t stat;                                 /* 线程状态 */
    /* 优先级 */
    rt_uint8_t current_priority;                     /* 当前优先级 */
    rt_uint8_t init_priority;                        /* 初始优先级 */
    rt_uint32_t number_mask;
    ......
    rt_ubase_t init_tick;                            /* 线程初始化计数值 */
    rt_ubase_t remaining_tick;                       /* 线程剩余计数值 */
    struct rt_timer thread_timer;                    /* 内置线程定时器 */
    void ( * cleanup)(struct rt_thread * tid);       /* 线程退出清除函数 */
    rt_uint32_t user_data;                           /* 用户数据 */
};
```

其中，init_priority 是线程创建时指定的线程优先级，在线程运行过程中是不会被改变的（除非用户执行线程控制函数进行手动调整线程优先级）。cleanup 会在线程退出时，被空闲线程回调一次以执行用户设置的清理现场等工作。最后一个成员 user_data 可由用户挂载一些数据信息到线程控制块中，以提供类似线程私有数据的实现。

在线程运行的过程中，同一时间内只允许一个线程在处理器中运行，从运行的过程上划分，线程有多种不同的运行状态，如初始状态、挂起状态、就绪状态等。在 RT-Thread 中，线程包含 5 种状态，操作系统会自动根据它运行的情况来动态调整它的状态。RT-Thread 中线程的 5 种状态，如表 3-3 所示。

表 3-3　线程的 5 种状态

状态	描　　述
初始状态	当线程刚开始创建还没开始运行时就处于初始状态；在初始状态下，线程不参与调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_INIT
就绪状态	在就绪状态下，线程按照优先级排队，等待被执行；一旦当前线程运行完毕让出处理器，操作系统会马上寻找最高优先级的就绪状态线程运行。此状态在 RT-Thread 中的宏定义为 RT_THREAD_READY
运行状态	线程当前正在运行。在单核系统中，只有 rt_thread_self()函数返回的线程处于运行状态；在多核系统中，可能就不止这一个线程处于运行状态。此状态在 RT-Thread 中的宏定义为 RT_THREAD_RUNNING
挂起状态	也称阻塞状态。它可能因为资源不可用而挂起等待，或线程主动延时一段时间而挂起。在挂起状态下，线程不参与调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_SUSPEND
关闭状态	当线程运行结束时将处于关闭状态。关闭状态的线程不参与线程的调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_CLOSE

RT-Thread 提供一系列的操作系统调用接口,使得线程的状态在这 5 种状态之间切换。几种状态间的转换关系如图 3-6 所示。

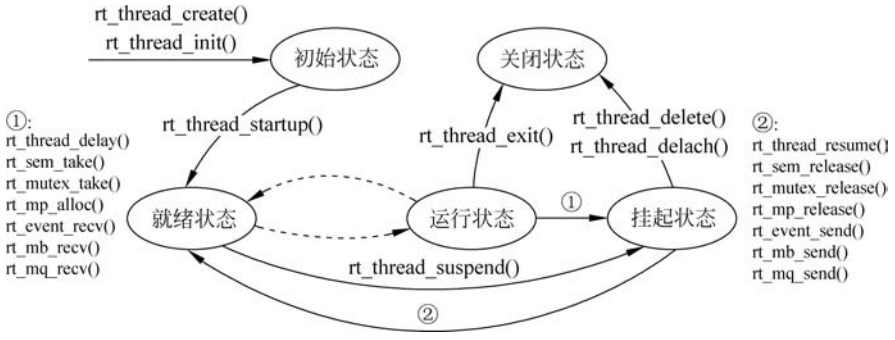

图 3-6 状态的转换

线程通过调用函数 rt_thread_create/init()进入初始状态(RT_THREAD_INIT);初始状态的线程通过调用函数 rt_thread_startup()进入就绪状态(RT_THREAD_READY);就绪状态的线程被调度器调度后进入运行状态(RT_THREAD_RUNNING);当处于运行状态的线程调用 rt_thread_delay()、rt_sem_take()、rt_mutex_take()、rt_mb_recv()等函数或者获取不到资源时,将进入挂起状态(RT_THREAD_SUSPEND);处于挂起状态的线程,如果等待超时依然未能获得资源或由于其他线程释放了资源,那么它将返回就绪状态。挂起状态的线程,如果调用 rt_thread_delete/detach()函数,将更改为关闭状态(RT_THREAD_CLOSE);而运行状态的线程,如果运行结束,就会在线程的最后部分执行 rt_thread_exit()函数,将状态更改为关闭状态。

值得注意的是,在 RT-Thread 中,实际上线程并不存在运行状态,就绪状态和运行状态是等同的。

3.1.4 内存管理

RT-Thread 操作系统在内存管理上,根据上层应用及系统资源的不同,有针对性地提供了不同的内存分配管理算法。总体上可分为两类:内存堆管理与内存池管理,而内存堆管理又根据具体内存设备分为 3 种情况:

- 第 1 种是针对小内存块的分配管理(小内存管理算法);
- 第 2 种是针对大内存块的分配管理(slab 管理算法);
- 第 3 种是针对多内存堆的分配情况(memheap 管理算法)。

小内存管理算法主要针对系统资源比较少,一般用于小于 2MB 内存空间的系统;而 slab 内存管理算法则主要是在系统资源比较丰富时,提供了一种近似多内存池管理算法的快速算法。memheap 方法适用于系统存在多个内存堆的情况,它可以将多个内存“粘贴”在一起,形成一个大的内存堆,这样用户使用起来会非常方便。

这几类内存堆管理算法在系统运行时只能选择其中之一或者完全不使用内存堆管理器,他们提供给应用程序的 API 接口完全相同。

正如在 2.2.4 节中介绍的,小内存管理算法是一个简单的内存分配算法。初始时,它是一块大的内存。当需要分配内存块时,将从这个大的内存块上分割出相匹配的内存块,然后

把分割出来的空闲内存块还回给堆管理系统中。每个内存块都包含一个管理用的数据头，通过这个头把使用块与空闲块用双向链表的方式链接起来。

RT-Thread 的 slab 分配器是在 DragonFly BSD 创始人 Matthew Dillon 实现的 slab 分配器基础上，针对嵌入式系统优化的内存分配算法。最原始的 slab 算法是 Jeff Bonwick 为 Solaris 操作系统引入的一种高效内核内存分配算法。

RT-Thread 的 slab 分配器实现主要是去掉了其中的对象构造及析构过程，只保留了纯粹的缓冲型的内存池算法。slab 分配器会根据对象的大小分成多个 zone(区)，也可以看成每类对象有一个内存池。

内存分配器主要有两种操作：

1) 内存分配

假设分配一个 32B 的内存，slab 内存分配器会先按照 32B 的值，从 zone array 链表表头数组中找到相应的 zone 链表。如果这个链表是空的，则向页分配器分配一个新的 zone，然后从 zone 中返回第一个空闲内存块。如果链表非空，则这个 zone 链表中的第一个 zone 节点必然有空闲块存在(否则它就不应该放在这个链表中)，那么就取相应的空闲块。如果分配完成后，zone 中所有空闲内存块都使用完毕，那么分配器需要把这个 zone 节点从链表中删除。

2) 内存释放

分配器需要找到内存块所在的 zone 节点，然后把内存块链接到 zone 的空闲内存块链表中。如果此时 zone 的空闲链表指示出 zone 的所有内存块都已经释放，即 zone 是完全空闲的，那么当 zone 链表中全空闲 zone 达到一定数目后，系统就会把这个全空闲的 zone 释放到页面分配器中去。

使用 memheap 内存管理可以简化系统存在多个内存堆时的使用：当系统中存在多个内存堆的时候，用户只需要在系统初始化时将多个所需的 memheap 初始化，并开启 memheap 功能就可以很方便地把多个 memheap(地址可不连续)黏合起来用于系统的 heap 分配。

3.1.5　组件与服务

RT-Thread 的组件和服务层建立在基础内核层之上，包含了众多功能强大的组件和服务。该层主要包括各种网络框架、设备框架、异常处理/日志服务、键值数据库、DFS 虚拟文件系统、USB 栈、FinSH 控制台、低功耗管理服务、WiFi Manager 及部分安全框架。

本节介绍几种重要的 RT-Thread 组件与服务。

1. I/O 设备模型框架

RT-Thread 提供了一套简单的 I/O 设备模型框架，如图 3-7 所示，它位于硬件和应用程序之间，共分成 3 层，从上到下分别是 I/O 设备管理层、设备驱动框架层、设备驱动层。

设备模型框架更类似一套驱动框架，涉及 UART、I^2C、SPI、SDIO、USB 从设备/主设备、EMAC、NAND 闪存设备等。它会把这些设备驱动中的共性抽象/抽取出来，而驱动工程师只需要按照固定的模式实现少量的底层硬件操作及板级配置。通过这样的方式，让一个硬件外设更容易地对接到 RT-Thread 系统中，并获得 RT-Thread 平台上的完整软件栈功能。

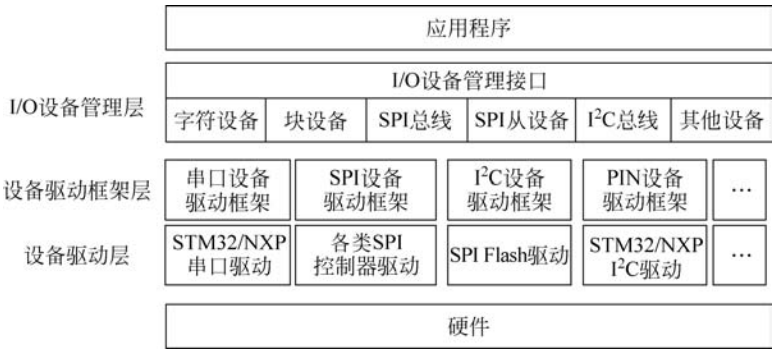

图 3-7 I/O 设备模型框架

应用程序通过 I/O 设备管理接口获得正确的设备驱动,然后通过这个设备驱动与底层 I/O 硬件设备进行数据(或控制)交互。

I/O 设备管理层实现了对设备驱动程序的封装。应用程序通过 I/O 设备层提供的标准接口访问底层设备,设备驱动程序的升级、更替不会对上层应用产生影响。这种方式使得设备的硬件操作相关的代码能够独立于应用程序而存在,双方只需关注各自的功能实现,从而降低了代码的耦合性、复杂性,提高了系统的可靠性。

设备驱动框架层是对同类硬件设备驱动的抽象,将不同厂家的同类硬件设备驱动中相同的部分抽取出来,将不同部分留出接口,由驱动程序实现。

设备驱动层是一组驱使硬件设备工作的程序,实现访问硬件设备的功能。它负责创建和注册 I/O 设备,对于操作逻辑简单的设备,可以不经过设备驱动框架层,直接将设备注册到 I/O 设备管理器中,使用序列图如图 3-8 所示,主要有以下两点:

设备驱动根据设备模型定义,创建出具备硬件访问能力的设备实例,将该设备通过 rt_device_register()接口注册到 I/O 设备管理器中。

应用程序通过 rt_device_find() 接口查找到设备,然后使用 I/O 设备管理接口访问硬件。

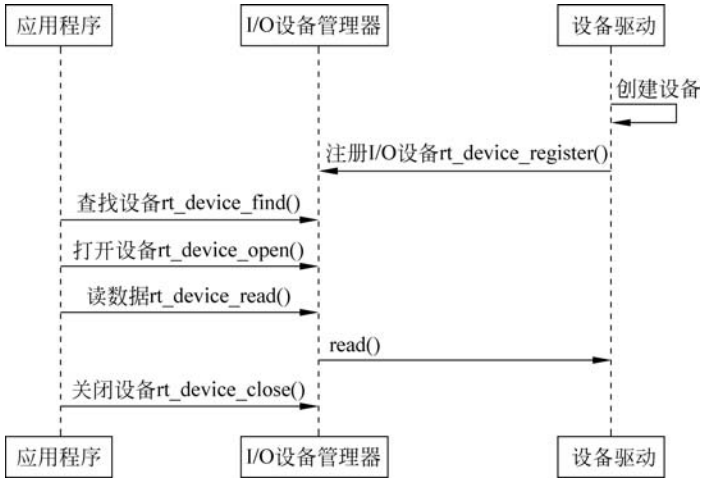

图 3-8 设备驱动层的使用序列

RT-Thread 的设备模型是建立在内核对象模型基础之上的，设备被认为是一类对象，被纳入对象管理器的范畴。每个设备对象都是由基对象派生而来的，每个具体设备都可以继承其父类对象的属性，并派生出其私有属性，图 3-9 是设备对象的继承和派生关系示意图。

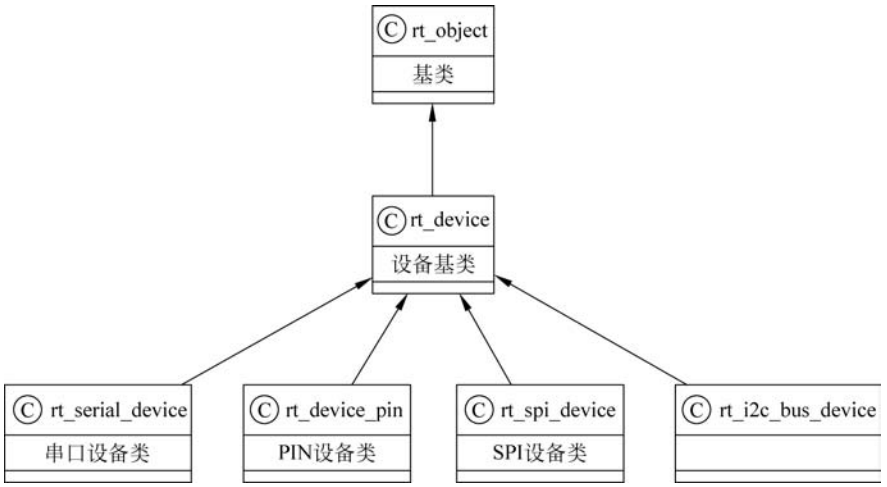

图 3-9　设备对象的继承和派生

设备对象的具体定义如下所示。

```
struct rt_device
{
     struct rt_object       parent;             /* 内核对象基类 */
     enum rt_device_class_type type;            /* 设备类型 */
     rt_uint16_t            flag;               /* 设备参数 */
     rt_uint16_t            open_flag;          /* 设备打开标志 */
     rt_uint8_t             ref_count;          /* 设备被引用次数 */
     rt_uint8_t             device_id;          /* 设备 ID,0 - 255 */
     rt_err_t ( * rx_indicate)(rt_device_t dev, rt_size_t size);
     rt_err_t ( * tx_complete)(rt_device_t dev, void * buffer);
     const struct rt_device_ops * ops;          /* 设备操作方法 */
                                                /* 设备的私有数据 */
     void * user_data;
};
typedef struct rt_device * rt_device_t;
```

RT-Thread 支持多种 I/O 设备类型，主要设备类型如下所示。

```
RT_Device_Class_Char                            /* 字符设备 */
RT_Device_Class_Block                           /* 块设备 */
RT_Device_Class_NetIf                           /* 网络接口设备 */
RT_Device_Class_MTD                             /* 内存设备 */
RT_Device_Class_RTC                             /* RTC 设备 */
RT_Device_Class_Sound                           /* 声音设备 */
RT_Device_Class_Graphic                         /* 图形设备 */
RT_Device_Class_I2CBUS                          /* I2C 总线设备 */
```

```
RT_Device_Class_USBDevice                    /* USB device 设备 */
RT_Device_Class_USBHost                      /* USB host 设备 */
RT_Device_Class_SPIBUS                       /* SPI 总线设备 */
RT_Device_Class_SPIDevice                    /* SPI 设备 */
RT_Device_Class_SDIO                         /* SDIO 设备 */
RT_Device_Class_Miscellaneous                /* 杂类设备 */
```

其中,字符设备、块设备是常用的设备类型,它们的分类依据是设备数据与系统之间的传输处理方式。字符模式设备允许非结构的数据传输,即通常数据传输采用串行的形式,每次一个字节。字符设备通常是一些简单设备,如串口、按键。

块设备每次传输一个数据块,例如,每次传输 512B 数据。这个数据块是硬件强制性的,数据块可能使用某类数据接口或某些强制性的传输协议,否则就可能发生错误。

2. FinSH

FinSH 是 RT-Thread 的命令行组件,提供一套供用户在命令行调用的操作接口,主要用于调试或查看系统信息。它可以使用串口/以太网/USB 等与 PC 进行通信。

用户在控制终端输入命令,控制终端通过串口、USB、网络等方式将命令传给设备中的 FinSH,FinSH 会读取设备输入命令,解析并自动扫描内部函数表,寻找对应函数名,执行函数后输出回应,回应通过原路返回,将结果显示在控制终端上。

当使用串口连接设备与控制终端时,FinSH 命令的执行流程,如图 3-10 所示。

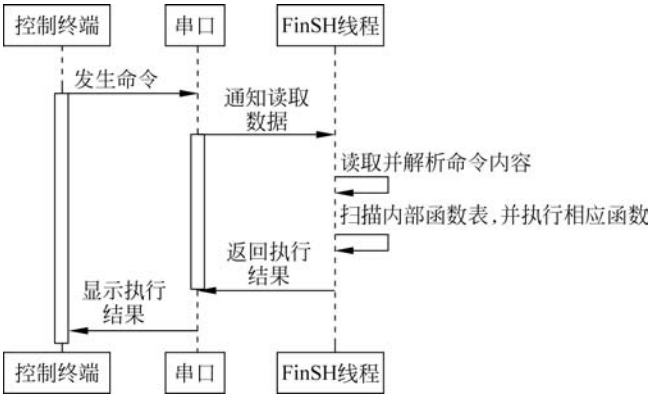

图 3-10 FinSH 命令的执行流程

RT-Thread 具有丰富的组件与服务,由于篇幅限制,此处不再赘述。有兴趣的读者可以在 RT-Thread 官网查阅相关文档获得更多信息。

3. 虚拟文件系统

DFS 是 RT-Thread 提供的虚拟文件系统组件,全称为 Device File System,即设备虚拟文件系统,文件系统的名称使用类似 UNIX 文件、目录的风格。

RT-Thread DFS 组件的主要功能特点有:

- 为应用程序提供统一的 POSIX 文件和目录操作接口,如 read、write、poll/select 等。
- 支持多种类型的文件系统,如 FatFS、RomFS、DevFS 等,并提供普通文件、设备文件、网络文件描述符的管理。

• 支持多种类型的存储设备，如 SD Card、SPI Flash、Nand Flash 等。

DFS 的层次架构如图 3-11 所示，主要分为 POSIX 接口层、虚拟文件系统层和设备抽象层。

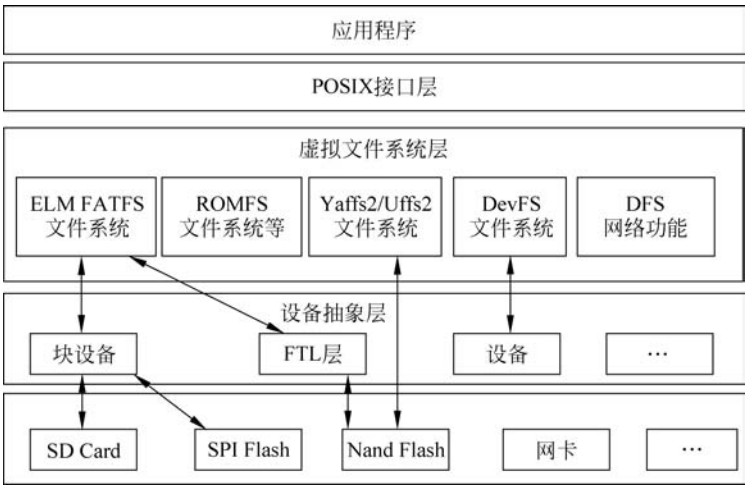

图 3-11　设备虚拟文件系统的层次

1）POSIX 接口层

POSIX 表示可移植操作系统接口（Portable Operating System Interface of UNIX，POSIX），POSIX 标准定义了操作系统应该为应用程序提供的接口标准，是 IEEE 为要在各种 UNIX 操作系统上运行的软件而定义的一系列 API 标准的总称。

POSIX 标准旨在获得源代码级别的软件可移植性。换句话说，为一个 POSIX 兼容的操作系统编写的程序，应该可以在任何其他 POSIX 操作系统（即使是来自另一个厂商）上编译执行。RT-Thread 支持 POSIX 标准接口，因此可以很方便地将 Linux/UNIX 的程序移植到 RT-Thread 操作系统上。

在类 UNIX 系统中，普通文件、设备文件、网络文件描述符是同一种文件描述符。而在 RT-Thread 操作系统中，使用 DFS 来实现这种统一性。有了这种文件描述符的统一性，就可以使用 poll/select 接口来对这几种描述符进行统一轮询，为实现程序功能带来方便。

使用 poll/select 接口可以以阻塞方式同时探测一组支持非阻塞的 I/O 设备是否有事件发生（如可读、可写、有高优先级的错误输出、出现错误等），直至某一个设备触发了事件或者超过了指定的等待时间。这种机制可以帮助调用者寻找当前就绪的设备，降低编程的复杂度。

2）虚拟文件系统层

用户可以将具体的文件系统注册到 DFS 中，如 FatFS、RomFS、DevFS 等，下面介绍几种常用的文件系统类型。

FatFS 是专为小型嵌入式设备开发的一个兼容 Microsoft FAT 格式的文件系统，采用 ANSI C 编写，具有良好的硬件无关性以及可移植性，是 RT-Thread 中最常用的文件系统类型。

传统型的 RomFS 文件系统是一种简单的、紧凑的、只读的文件系统，不支持动态擦写

保存,按顺序存放数据,因而支持应用程序以 XIP(eXecute In Place,片内运行)方式运行,在系统运行时,节省 RAM 空间。

除上述文件系统之外,JFFS2、DevFS 和 NFS 网络文件系统也有较为广泛的应用。

JFFS2 文件系统是一种日志闪存文件系统,基于 MTD 驱动层,特点是:可读写的、支持数据压缩的、基于哈希表的日志型文件系统,并提供了崩溃/掉电安全保护,提供写平衡支持等。

DevFS 即设备文件系统,在 RT-Thread 操作系统中开启该功能后,可以将系统中的设备在/dev 目录下虚拟成文件,使得设备可以按照文件的操作方式使用 read、write 等接口进行操作。

NFS 网络文件系统(Network File System)是一项在不同机器、不同操作系统之间通过网络共享文件的技术。在操作系统的开发调试阶段,可以利用该技术在主机上建立基于 NFS 的根文件系统,挂载到嵌入式设备上,可以很方便地修改根文件系统的内容。

UFFS 是 Ultra-low-cost Flash File System(超低功耗的闪存文件系统)的简称。它是专为嵌入式设备等小内存环境中使用 Nand Flash 的开源文件系统。与嵌入式中常使用的 Yaffs 文件系统相比具有资源占用少、启动速度快、免费等优势。

3) 设备抽象层

设备抽象层将物理设备如 SD Card、SPI Flash、Nand Flash,抽象成符合文件系统能够访问的设备,例如,FAT 文件系统要求存储设备必须是块设备类型。

不同文件系统类型是独立于存储设备驱动而实现的,因此把底层存储设备的驱动接口和文件系统对接起来之后,才可以正确地使用文件系统功能。

3.1.6 软件包

RT-Thread 不仅是一个嵌入式实时操作系统,也是一个优秀的物联网操作系统。这是由于 RT-Thread 具有功能丰富的软件包。随着 RT-Thread 3.0 中的包管理器开启,越来越多的软件组件将以软件包方式出现在 RT-Thread 平台中。RT-Thread 的软件包支持众多的软件模块,本节介绍几个代表性的软件包。

1. ENV

Env 是 RT-Thread 推出的开发辅助工具,针对基于 RT-Thread 操作系统的项目工程,提供编译构建环境、图形化系统配置及软件包管理功能。其内置的 menuconfig 提供了简单易用的配置剪裁工具,可对内核、组件和软件包进行自由裁剪,使系统以搭积木的方式进行构建。

ENV 具有菜单图形化配置界面,该界面交互性好,操作逻辑性强;具有使用灵活,能够自动处理依赖文件和具备彻底开关功能的特点。ENV 能够自动生成 rtconfig.h,无须手动修改,并且可以使用 scons 工具生成工程,提供编译环境,操作简单。ENV 提供多种软件包,模块化软件包耦合关联少,可维护性好,ENV 软件包可在线下载,软件包持续集成,可靠性高。

2. MQTT 软件包

MQTT(Message Queuing Telemetry Transport,消息队列遥测传输协议),是一种基于发布/订阅(publish/subscribe)模式的“轻量级”通信协议,该协议构建于 TCP/IP 协议上,

由 IBM 在 1999 年发布。MQTT 的最大优点在于，可以以极少的代码和有限的带宽，为连接远程设备提供实时可靠的消息服务。作为一种低开销、低带宽占用的即时通信协议，MQTT 在物联网、小型设备、移动应用等方面有较广泛的应用。

MQTT 是一个基于客户端-服务器的消息发布/订阅传输协议。MQTT 协议是轻量、简单、开放和易于实现的，这些特点使它适用范围非常广泛。在很多情况下，包括受限的环境中，如，机器与机器(M2M)通信和物联网(IoT)。其在卫星链路通信传感器、偶尔拨号的医疗设备、智能家居及一些小型化设备中已广泛使用。

MQTT 协议运行在 TCP/IP 或其他网络协议之上，它将建立客户端到服务器的连接，提供两者之间的一个有序的、无损的、基于字节流的双向传输。

3. 阿里巴巴云 IoT 软件包

Ali-iotkit 是 RT-Thread 移植的用于连接阿里巴巴云 IoT 平台的软件包。基础 SDK 是阿里巴巴提供的 iotkit-embedded C-SDK。

该物联网套件能够使嵌入式设备快速接入(设备端 SDK)以及能够桥接到阿里巴巴云其他产品，对设备数据进行存储/计算。

Iotkit SDK 为了方便设备上云封装了丰富的连接协议，如 MQTT、CoAP、HTTP、TLS，并且对硬件平台进行了抽象，使其不受具体的硬件平台限制而更加灵活。在代码架构方面，Iotkit SDK 分为 3 层，如图 3-12 所示。

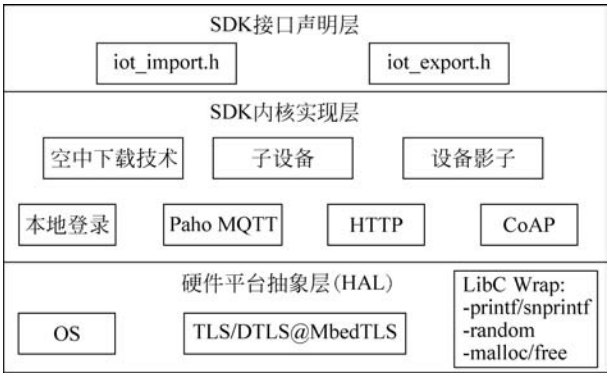

图 3-12　Iotkit SDK 的层次

1) 最底层

最底层称为硬件平台抽象层，也简称 HAL 层(Hardware Abstract Layer)，该层包括对不同的嵌入式目标板的抽象，操作系统对 SDK 的支撑函数(包括网络收发、TLS/DTLS 通道建立和读写，内存申请是否和互斥量加锁解锁等)。

2) 中间层

中间层称为 SDK 内核实现层(IoT SDK Core Implements)，该层是物联网平台 C-SDK 的核心实现部分，它基于 HAL 层接口完成了 MQTT/CoAP 通道等的功能封装，包括 MQTT 的连接建立、报文收发、CoAP 的连接建立、报文收发、OTA 的固件状态查询和 OTA 的固件下载等。中间层的封装，使得用户无须关心内部实现逻辑，可以不经修改地应用。

3) 最上层

最上层称为 SDK 接口声明层(IoT SDK Interface Layer)，该层为应用提供 API，用户

使用该层的 API 完成具体的业务逻辑。

下面以阿里巴巴云 IoT 产品中的 SDK 为例说明软件包的文件构成。

iotkit-embedded 软件包是阿里巴巴物联网平台 C-SDK 源码,包含了连接阿里巴巴云 IoT 所必需的软件包。iotkit-embedded 软件包目录结构如下所示。

```
+-- LICENSE                : 软件许可证, Apache 2.0 版本软件许可证
+-- make.settings          : 功能裁剪配置, 如 MQTT|CoAP, 或裁剪如 OTA|Shadow
+-- README.md              : 快速开始导引
+-- sample                 : 例程目录, 演示通信模块和服务模块的使用
| +-- mqtt                 : 演示如何使用通信模块 MQTT 的 API
| +-- coap                 : 演示如何使用通信模块 CoAP 的 API
| +-- device-shadow        : 演示如何使用服务模块设备影子的 API
| +-- http                 : 演示如何使用通信模块 HTTP 的 API
| +-- ota                  : 演示如何使用服务模块 OTA 的 API
+-- src
    +-- sdk-impl           : SDK 的接口层, 提供总体的头文件,和一些 API 的接口封装
    +-- sdk-tests          : SDK 的单元测试
    +-- mqtt               : 通信模块, 实现以 MQTT 协议接入
    +-- coap               : 通信模块, 实现以 CoAP 协议接入
    +-- http               : 通信模块, 实现以 HTTP 协议接入
    +-- ota                : 服务模块, 实现基于 MQTT|CoAP + HTTP + TLS 的固件下载通道
    +-- shadow             : 服务模块, 实现设备影子
    +-- platform           : 硬件平台抽象层, 需要移植适配
    +-- import             : 外部输入目录, 存放芯片/模组厂商提供的头文件/二进制库
    +-- configs            : 硬件平台编译配置, 如交叉编译工具链设置, 功能模块裁剪等
    +-- scripts            : 编译过程将要外部引用的脚本, 用户不必关注
    +-- packages           : SDK 引用的外部软件模块, 用户不必关注
    +-- log                : 基础模块, 实现运行日志
    +-- system             : 基础模块, 保存全局信息, 如 TLS 根证书,设备标识 ID 等
    +-- tls                : 基础模块, 实现 TLS/DTLS, 来自裁剪过的开源软件 Mbed TLS
    +-- utils              : 基础模块, 实现工具函数, 如 SHA1 摘要计算、NTP 对时等
```

3.2 ARM Mbed OS

ARM Mbed 不仅指一个嵌入式系统,也是一个面向 ARM 处理器的设备开发平台,ARM 提供高效、安全、快速开发下一个物联网产品所需的操作系统、工具和云服务,它具体包括 Mbed OS、Mbed HDK、Mbed Web Complier 工具和物联网平台等,内容涉及从硬件软件、从物端到云端等,功能包括安全、通信传输、设备管理等方面,为实现物联网从原型、开发到生产的快速便捷。具体内容如下。

(1) Mbed OS: 2014 年,ARM 宣布推出用于物联网低功耗设备的嵌入式操作系统 Mbed OS。ARM Mbed OS 是一个免费的操作系统,专为基于 ARM Cortex-M 处理器的设备而设计。该系统将物联网的所有基本组件(包括安全性、通信和设备管理)集成到一整套软件中,以协助开发低功耗、产品级的物联网设备和并优化生产流程。物联网操作系统 Mbed OS 为其运行的微控制器提供了一个抽象层,因此开发人员可以专注于编写调用一系列匹配硬件的 C/C++应用程序。

(2) HDK: Mbed 硬件开发套件(Mbed HDK)是一系列硬件设计资源,以协助开发受

益于 Mbed 生态系统的定制硬件，如 DAPLink。使用基于 Mbed HDK 的开发板是开始使用 Mbed 平台的最有效方式。

(3) SDK：Mbed SDK 旨在提供直观、简洁，但功能强大，足以构建复杂项目的硬件抽象。它基于低级 ARM CMSIS API 构建，目的是屏蔽不同 MCU 厂商微处理之间的差异，对于用户来说，只需使用抽象层与接口打交道即可，不需要担心底层硬件层驱动，减少了开发难度，SDK 可以使基于 Mbed 开发的应用很方便地转移到其他 ARM 微处理器上。

(4) Web Complier：Mbed 为在线编译，Mbed 编译器提供了一个轻量级的在线 C/C++IDE，通过它代替传统的离线编译器，使整个开发在浏览器中完成，使用它以在 Mbed 微控制器上运行而不必进行任何安装或设置操作，省去了用户搭建开发环境的麻烦，用户只要上网就可以开发。

(5) ARM Mbed 物联网设备平台(ARM Mbed IoT Device Platform)：ARM 于 2018 年 8 月发布的 ARM Pelion 平台，为产业首个专为混合环境提供 IoT 连接、设备和资料管理的平台。该平台能够基于 ARM 微控制器以最短的时间创建支持商用与互操作的物联网设备。ARM Mbed 物联网设备平台提供了所有关键组件，通过 ARM 的 Mbed 操作系统、Mbed 设备服务器以及 Mbed 社区生态系统创建安全、高效的物联网应用。

ARM Mbed 物联网设备平台能够提供用于开发物联网设备的通用操作系统基础，以解决嵌入式设计的碎片化问题。该平台支持所有重要的连接性与设备管理开放标准，以实现面向未来的设计。该平台使安全可升级的边缘设备支持新增处理能力与功能。ARM Mbed 物联网设备平台通过自动电源管理解决复杂的能耗问题。该平台提供基于云的开发工具套装，帮助用户以前所未有的速度开发自身产品。

ARM Mbed 物联网设备平台为构建物联网应用提供了理想的构件。Now Reference Apps 是针对 Mbed 及其源代码打造的完整图形 Web 应用程序，为系统集成商、原始设备制造商和 Web 开发人员提供了一套无与伦比的解决方案，以快速部署基于 Mbed 设备服务器的物联网服务。

此外，Mbed 还有丰富的功能组件提供需要的功能。总之，Mbed 提供了物联网整套服务，是一个方便快捷的物联网应用平台。

与其他操作系统一样，Mbed OS 在硬件和硬件抽象层上运行，下面简要介绍 ARM 相关处理器和硬件抽象层 CMSIS。

3.2.1　硬件及硬件抽象层

1. Cortex 内核

最初的 ARM 处理器型号都是用数字命名的，直到最后一个系列 ARM11，在此之后，所有的处理器都改用 Cortex 命名，目前 Cortex 处理器分为 A、R、M 三大类，为不同市场需求服务：A 系列面向尖端的基于虚拟内存的操作系统和用户应用；R 系列针对实时系统；M 系列对微控制器。

ARM 对处理器明确分为三大系列。

(1) Application Processors(应用处理器)：适用于具有高计算要求、运行丰富操作系统以及提供交互媒体和图形体验的应用领域，比如手机、数码家电、路由器、上网本等，具有高效低耗的特点，目前 Cortex-A 正朝着提供完整 Internet 体验的方向发展。

(2) Real-time Processors(实时处理器)：适用于实时应用领域，比如汽车应用中的安全气囊、制动系统、企业应用中的打印机、网络设备等。多数实时处理器不支持 MMU，不过通常具有 MPU、Cache 和其他针对工业应用设计的存储器功能。Cortex-R 运行时的时钟频率非常高，一般为 200MHz～1GHz，同时。虽然实时处理器不能运行完整版本的 Linux 和 Windows 操作系统，但是支持大量的实时操作系统(RTOS)，响应延迟非常低。

(3) Microcontroller Processors(微控制器处理器)：Cortex-M 通常面积比较小、能效比较高，这一类的处理器被设计用来满足嵌入式应用的需要。相较于其他处理器，它能以更低的时钟频率工作，一般小于 200MHz，拥有基于架构的睡眠模式支持。并且新的 Cortex-M 处理器家族设计得极其容易使用。所以，ARM Cortex-M 处理器的主要应用领域是在单片机和嵌入式应用市场，也是较常用的一款处理器。

综上所述，ARM 处理器系列分类如表 3-4 所示。

表 3-4 Cortex 处理器分类

比较项目	Cortex-A	Cortex-R	Cortex-M
设计特点	高时钟频率，长流水线，高性能，对媒体处理支持(NEON 指令集扩展)	高时钟频率，较长流水线，高确定性(中断延迟低)	通常较短的流水线，超低功耗
系统特征	内存管理单元(MMU)，高速缓存，ARM TrustZone 安全扩展	内存保护单元(MPU)，高速缓存，紧耦合内存(TCM)	内存保护单元(MPU)，嵌套向量中断控制器(NVIC)，唤醒中断控制器(WIC)，最新 ARM TrustZone 安全扩展
目标市场	移动计算，智能手机，高效能服务器，高端微处理器	工业微控制器，汽车电子，硬盘控制器，基带	微控制器，深度嵌入系统(例如，传感器、MEMS、混合信号 IC、IoT)

Cortex-M 处理器家族更多地集中在低性能端，但是这些处理器相比于许多微控制器使用的传统处理器性能仍然很强大。例如，Cortex-M4 和 Cortex-M7 处理器应用在许多高性能的微控制器产品中，它们可以以高达 400MHz 的时钟频率工作。

当然，性能不是选择处理器的唯一指标。在许多应用中，低功耗和成本是关键的选择指标。因此，Cortex-M 处理器家族包含各种产品来满足不同的需求，如表 3-5 所示。

表 3-5 Cortex-M 处理器分类

处理器	描　　述
Cortex-M0	面向成本低、超低功耗的微控制器和深度嵌入应用的非常小的处理器(最少 12K 个门电路)
Cortex-M0＋	针对小嵌入式系统的最高能效的处理器，与 Cortex-M0 处理器接近的尺寸大小和编程模式，但是具有扩展功能，如单周期 I/O 接口和向量表重定位功能
Cortex-M1	针对 FPGA 设计优化的小处理器，利用 FPGA 上的存储器块实现了紧耦合内存(TCM)。和 Cortex-M0 有相同的指令集
Cortex-M3	针对低功耗微控制器设计的处理器，面积小但是性能强劲，支持快速处理复杂任务的丰富指令集。具有硬件除法器和乘加指令(MAC)。并且，Cortex-M3 支持全面的调试和跟踪功能，使软件开发者可以快速开发应用

续表

处理器	描　　述
Cortex-M4	不但具有 Cortex-M3 的所有功能，并且扩展了面向数字信号处理(DSP)的指令集，比如单指令多数据指令(SMID)和更快的周期 MAC 操作。此外，它还有一个可选的支持 IEEE 754 浮点标准的单精度浮点运算单元
Cortex-M7	针对高端微处理器和数据处理密集的应用开发的高性能处理器，具备 Cortex-M4 支持的所有指令功能，扩展支持双精度浮点运算，并且具备扩展的存储器功能，例如，Cache 和紧耦合存储器(TCM)
Cortex-M23	面向超低功耗、低成本应用设计的小处理器，和 Cortex-M0 相似，但是支持各种增强的指令集和系统层面的功能特性。Cortex-M23 还支持 TrustZone 安全扩展
Cortex-M33	主流的处理器设计，与之前的 Cortex-M3 和 Cortex-M4 处理器类似，但系统设计更灵活，能耗比更高，性能更高。Cortex-M33 还支持 TrustZone 安全扩展

2. CMSIS

CMSIS 的英文全称为 Cortex Microcontroller Software Interface Standard，专为 Cortex-M 而设计，是独立于供应商的硬件抽象层。

通过 CMSIS 中提供的处理器和外设软件间的标准接口，使得以简单的接口调用就可以实现程序开发，而不用了解过多的处理器硬件原理，极大地降低了开发难度，简化了软件开发流程，缩短了开发中的学习过程。

嵌入式行业认为软件的创建是主要的成本因素。而 CMSIS 的产生，在极大程度上缩减了这一成本，尤其是在设备迁移和创建新项目时，这一成本的降低更为明显，使用 CMSIS 编写的程序在迁移设备时几乎不用做过多的改动，只需要适配到新的接口。

CMSIS 可以分为多个软件层次，分别由 ARM 公司、芯片供应商提供。其中，ARM 提供了下列部分，可用于多种编译器。

内核设备访问层：包含了用来访问内核的寄存器设备的名称定义、地址定义和助手函数。同时也为 RTOS(实时操作系统)定义了独立于微控制器的接口，该接口包括调试通道定义。

中间设备访问层：为软件提供了访问外设的通用方法。芯片供应商应当修改中间设备访问层，以适应中间设备组件用到的微控制器上的外设。

微控制器外设访问层：提供片上所有外设的定义。

外设的访问函数(可选)：为外设提供额外的助手函数。CMSIS 为 Cortex-Mx 微控制器系统定义了：

- 访问外设寄存器的通用方法和定义异常向量的通用方法。
- 内核设备的寄存器名称和内核异常向量的名称。
- 独立于微控制器的 RTOS 接口，带调试通道。
- 中间设备组件接口(TCP/IP 协议栈、闪存文件系统)。

图 3-13 显示了 CMSIS 的组成情况。

CMSIS 包含以下组件：

- CMSIS-CORE——提供与 Cortex-M0、Cortex-M3、Cortex-M4、SC000 和 SC300 处理器与外围寄存器之间的接口。

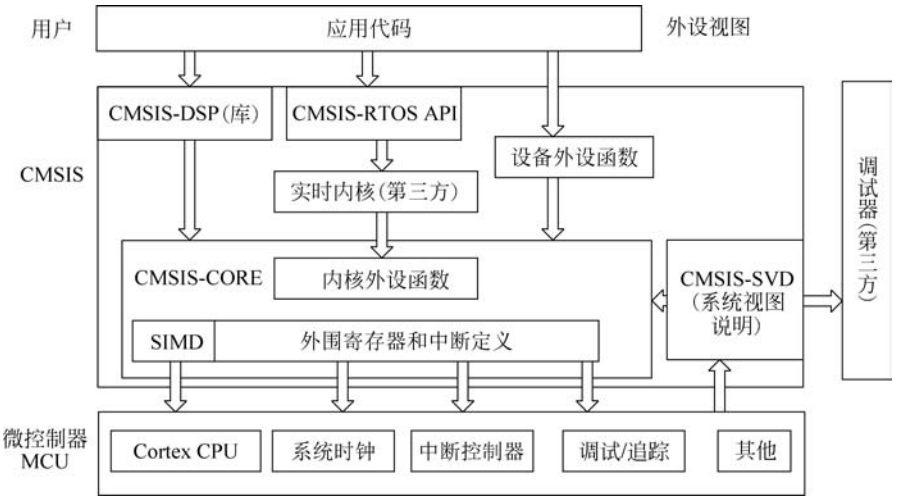

图 3-13 CMSIS 的组成

- CMSIS-DSP——包含以定点(分数 q7、q15、q31)和单精度浮点(32 位)实现的 60 多种函数的 DSP 库。
- CMSIS-RTOS API——用于线程控制、资源和时间管理的实时操作系统的标准化编程接口。
- CMSIS-SVD——包含完整微控制器系统(包括外围设备)的程序员视图的系统视图描述 XML 文件。

如图 3-14 所示,CMSIS-RTOS 在用户的应用代码和第三方的 RTOS Kernel 直接架起一道桥梁,一个设计在不同的 RTOS 之间移植,或者在不同 Cortex MCU 之间直接移植的时候,如果两个 RTOS 都实现了 CMSIS-RTOS,那么用户的应用程序代码完全可以不做修改。

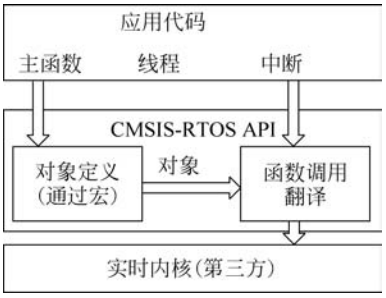

图 3-14 CMSIS-RTOS 的作用

3.2.2 Mbed OS 功能框架及优势

1. Mbed OS 功能框架

Mbed OS 功能框架如图 3-15 所示,Mbed OS 所提供的具有物联网需求的功能和协议包括 6LoWPAN、Web 传输受限制的应用协议(CoAP)、超文本传输协议(HTTP)、用于机器对机器(M2M)连接的消息队列遥测传输(MQTT)、密码协议传输层安全协议(TLS)、数

据包传输层安全性协议(DTLS)、开放移动联盟轻量级 M2M(OMA LwM2M)标准等。

图 3-15　Mbed OS 功能框架图

Mbed OS 操作系统可提供核心操作系统、稳健的安全基础、基于标准的通信功能以及针对传感器、I/O 设备和确认的驱动程序,能够加快从初始创意到部署产品的进程。Mbed OS 操作系统是模块化的可配置软件堆栈,有助于用户轻松针对目标开发设备对其进行自定义,以及通过排除不必要的软件组件降低内存要求。

Mbed OS 操作系统在微控制器上属于 Mbed IoT 设备平台的客户端部分,专为与 Mbed 设备连接器、Mbed 设备服务器和 Mbed 客户端配合使用而设计。总的来说,这一平台为用户提供全面的 IoT 解决方案。

Mbed OS 操作系统提供了 C++应用框架及组件架构,用于创建设备应用,从而减少了大量通常与 MCU 代码开发相关的底层工作。

2. Mbed OS 的优势

首先,Mbed OS 作为嵌入式操作系统的优势在于:

(1) 首先,相较于过去的开发工具和操作系统,Mbed 提供了一个相对更加系统和更加全面的智能硬件开发环境。Mbed 不但把当前智能硬件可能会涉及的外设(红外、电机、蜂鸣器、陀螺仪等)基本都进行了标准化处理,还提供了这些外设的原理、关键知识、示例代码等,这对于当前很多不太熟悉智能硬件的人来说,帮助都是十分巨大的。

(2) 将很多与硬件相关的程序使用中间件进行封装,这使得操作硬件不必再特意关心底层驱动,开发者只需要调用好接口即可,极大地降低了嵌入式开发的难度。

(3) 高度抽象,将硬件和软件完全分离开。Mbed OS 屏蔽了不同 MCU 厂商提供的微处理之间的差异(通过 Cortex-M-CMSIS 框架),换句话说,基于 Mbed 的用户可以轻松地替换来自不同制造商的 ARM 微处理器,而不用修改工程代码,但是,这里还是仅限于支持 Mbed 的处理器。

其次,也是更重要的是 Mbed OS 是专为 IoT 设备而特别构建。默认情况下,Mbed 操作系统是事件驱动的单线程架构,而非多线程(实时操作系统)环境。这确保了它可以扩展

到尺寸最小、成本最低且功耗最低的 IoT 设备。该操作系统包含事件驱动的、可向用户和系统事件提供服务的简单调度程序。

微控制器和 IoT 设备的硬件功能和要求有所不同。Mbed 操作系统包含低级硬件抽象层(HAL)以及适用于常见硬件外设(例如 SPI 和 I^2C 端口、GPIO 针和计时器)的高级抽象驱动程序。此外,Mbed OS 操作系统中的硬件抽象层和驱动程序还为电源管理提供了深层集成支持,加上调度程序中的能效认知功能,可帮助 mbed 操作系统应对高要求应用程序(在这些应用程序中,能效对操作至关重要)。

1) 安全

ARM Mbed IoT 设备平台在多个层级解决了安全问题:

- Mbed 设备安全架构的基础是 Mbed OS,它在微控制器中创建和强制实施独立的安全域。通过使用分隔系统的敏感部分,Mbed OS 解决了安全难题,不仅保护了启动流程和调试会话,确保了固件更新的安全安装,还阻止了恶意或错误代码升级权限和泄露秘密。目前,Mbed OS 需要使用带内存保护单元(MPU)的 Cortex-M3 或 Cortex-M4。
- Mbed OS 还会利用 Mbed TLS 提供最先进的通信安全功能。首先也是最重要的一点是,Mbed TLS(及相关的 Mbed OS)支持传输层安全(TLS) 协议。TLS 以及相关的 Datagram TLS(DTLS 数据包传输层安全) 协议是标准协议,用于保护 Internet 通信安全,已被证实能够防止窃听、篡改和伪造消息。此外,Mbed TLS 还包括一系列常用加密原语、证书处理及其他加密功能的参考质量软件实施。
- ARM 近年来收购了 Sansa Security,这使得在 Mbed OS 的后续版本中可以加入成熟的、功能丰富的轻量级生命周期安全功能。

2) 连接性和联网

Mbed OS 操作系统中支持的核心连接性技术包括以太网、WiFi、IPv6、6LoWPAN、线程和 BLE(Bluetooth Low Energy)。

ARM 主动帮助标准机构开发适用于 IoT 的协议和标准并确保现有行业标准在 IoT 环境中运作良好。

3) 可管理性

现场管理设备的能力是实现大规模部署的关键,同时也是 Mbed IoT 设备平台的核心部分,支持如下协议:

- OMA 轻量级 M2M(LwM2M)——用于监控和管理嵌入式设备的常用协议。它在 Mbed 操作系统、Mbed 客户端和 Mbed 设备服务器中为其提供支持。
- 受限应用程序协议(CoAP)——专为解决使用高效数据共享机制和 RESTful 通信跨受限网络进行通信的难题而设计。Mbed OS 和 Mbed 客户端为其提供支持,用户可以在其他嵌入式操作系统和 Linux 中实施这些协议。

3.3 Android Things/Brillo

2016 年 12 月,Google 公司发布了 Developer Preview 版的 Android Things,该平台为利用 Android 这一世界上最受支持的操作系统的强大功能构建物联网产品铺平了道路。但

它并不是一个全新的操作系统，而是通过同样由 Google 开发的物联网操作系统 Brillo 改进优化的一个操作系统。

尽管 Brillo 的核心是 Android 系统，但是它的开发和部署明显不同于常规 Android 开发。Brillo 把 C++作为主要开发环境，Android Things 面向所有 Java 开发者，不管开发者有没有移动开发经验。

Android Things 主要在 Android 的核心框架中扩展了一些支持物联的 API。开发者可以利用这些 API 直接与自定义的硬件打交道，Android Things 同时简化了单个程序的应用，开机可以自动运行用户程序。

由图 3-16 可见，Android Things 与 Android 一样，仍然使用 Linux 内核作为其操作系统内核。这样 Linux 在物联网领域应用的一些弊端，就被完整地继承到了 Brillo 中。比如，Linux 内核对运行内存的要求较高，同时 Linux 还需要 CPU 硬件支持 MMU(内存管理单元)功能等。这样就间接导致 Android Things 的运行内存要求较高，同时要求 CPU 支持 MMU 功能。这样大量的低端 CPU 或 MCU，比如 STM32 系列，就无法运行 Android Things，因为这些 CPU 的片上内存一般不超过 1MB，同时一般不提供 MMU 功能。由于这些原因，大大限制了 Android Things 的应用范围(Google 公司建议内存都在至少 32 MB 以上，实际都在百兆以上)。这一点和它的主要竞争对手 Windows 10 IoT 不相上下。

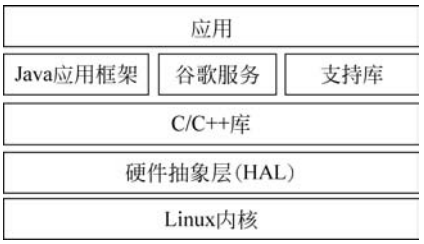

图 3-16　Android Things 的层次

实际上 Android Things 现在支持 4 款开发板：Intel Edison 开发板、Intel Joule 开发板、NXP Pico i.MX6UL 开发板和 Raspberry Pi 3 开发板。这 4 款开发板兼顾了 ARM 和 x86 架构，并且也兼顾了 32 位和 64 位的系统。所有的开发板都支持 WiFi 和蓝牙。

在 Linux 内核之上，Android Things 保留了 Android 操作系统中的一个硬件访问层(Hardware Access Layer，HAL)。这个层次的主要功能是对底层的硬件进行统一的抽象提供给应用程序访问接口。从功能上说，这一层软件并无明显的价值，但是其简化了对硬件的操作，给程序开发带来了较大的便利。按照一般的软件分层规则，这一层软件应该还是属于操作系统内核的一部分，因为它并没有提供额外的附加功能，在代码量上与内核相比也非常少，在某些情况下甚至可以忽略掉。因此，在展示上，应该与操作系统内核放在一起。但是 Google 为了区分这一层软件是来源于 Android 系统，而不是 Linux，才把它单独列出来了。

再往上就是支撑操作系统运行的一些辅助功能组件。主要有谷歌服务(Google Services)、Java API 框架和支持库。这些服务数量众多，功能强大，比如 Weave 框架、在线更新(OTA Updates)、安全相关的一些组件和机制以及在线数据分析和性能测量等。Android Things 整合了物联网设备通信平台 Weave，Weave SDK 将嵌入设备中进行本地和远程通信。Weave Server 是用来处理设备注册、命令传送、状态存储以及与 Google 助手等

Google 服务整合的云服务。在线更新机制可以使运行 Android Things 操作系统的物联网设备在运行过程中就可以更新软件,而不用中断运行。这个特性是非常有价值的。由于 Android Things 是一个复杂的系统,其版本更迭和补丁发布非常频繁,如果不提供在线更新功能,每发布一个新的版本和补丁都需要现场更新物联网设备,显然这是不可操作的。因此,Google 设计了这个特性来支撑在线实时软件更新功能。只要与 Android Things 的后台服务器连接上,Android Things 会自动检查更新,并安排更新,而不会影响设备的正常运行。安全机制主要提供了设备认证、数据加密等功能。在线性能统计和分析功能可以帮助用户实时查看和分析设备状态、性能、消息数量等数据,为设备维护人员提供一个基础的管理平台。开发者可以根据需要,选择启用或关闭这些外围辅助功能。

3.4 Contiki

Contiki 首先是一个适用于有内存的嵌入式系统的开源的、高度可移植的、支持网络的多任务操作系统,包括一个多任务核心、TCP/IP 堆栈、程序集以及低能耗的无线通信堆栈。Contiki 采用 C 语言开发的非常小型的嵌入式操作系统,运行只需要几千字节的内存。

Contiki 拥有出色的 TCP/IP 网络支持,包括 IPv4 和 IPv6,以及 6LoWPAN 报文压缩、RPL 路由、CoAP 应用层等。其中 6LoWPAN 已经成为 IETF 规范,也被 ZigBee SEP 2.0 标准以及 ISA 100.11a 标准所采纳。Contiki 已经成为无线传感器网络和物联网感知层低功耗无线组网协议研发和实验的主要平台。

Contiki 由瑞典计算机科学学院的 Adam Dunkels 和他的团队开发,已经应用在许多项目中,由于具有良好的网络通信能力,Contiki 被视为一款优秀的开源物联网操作系统,主要应用于无线传感器网络和物联网。Contiki 使用 C 语言开发,支持多任务、支持网络、高度可移植和可裁剪。

Contiki 支持 IPv4/IPv6 通信,提供了 uIPv6 协议栈、IPv4 协议栈(uIP),支持 TCP/UDP,还提供了线程、定时器、文件系统等功能。Contiki 具有低功率无线通信功能,提供了完整的网络协议栈和低功率无线通信机制。Contiki 的低功率无线通信协议栈是 Rime,实现了多种传感器网络协议,包括数据采集、最大努力网络洪泛、多跳批量数据传输及数据传播。

Contiki 提供 Web 浏览器访问传感的交互方式,通过 Web 浏览器查看、显示、存储及设置传感器数据。这种方式大大降低了传感器节点的操作方式。在功耗方面,Contiki 严格控制各传感器节点上的功耗,延长了传感器网络生命周期。同时 Contiki 还提供基于 Flash 的文件系统,方便在传感器节点上存储数据。在应用开发方面,Contiki 提供多任务编程模式,开发人员将各应用作为独立的任务,通过创建任务方式将各应用隔离。

3.4.1 架构分析

Contiki 系统的核心模块包括网络(net)、文件系统(cfs)、外部设备(dev)、链接库(lib)等,还包含时钟、I/O、ELF 装载器、网络驱动等的抽象;处理器支持 arm、avr、msp430 等,同时支持开发人员自己扩展处理器支持;硬件平台包括 stm32、mx231cc、micaz、sky、win32 等,支持开发人员扩展硬件平台支持;同时还自带 FTP、hell、WebServer 等应用程序;还提

供诸如 HelloWorld 等应用编程安全案例，方便开发人员编写应用程序。

Contiki 是一个事件驱动型系统，任务负责处理相应的事件而创建，中断与任务、任务与任务间都是通过事件交互。图 3-17 是 Contiki 的运行架构图，系统运行过程中包含 3 类主体：中断、任务及事件。其中，中断和任务是运行实体，而事件是中断与任务及任务与任务之间的交互对象。事件由中断或任务发起，并且由事件接收者任务负责处理。系统运行时先调度任务，当任务空时，再调度事件，根据事件找到对应的接收者任务，调度该任务去处理事件。

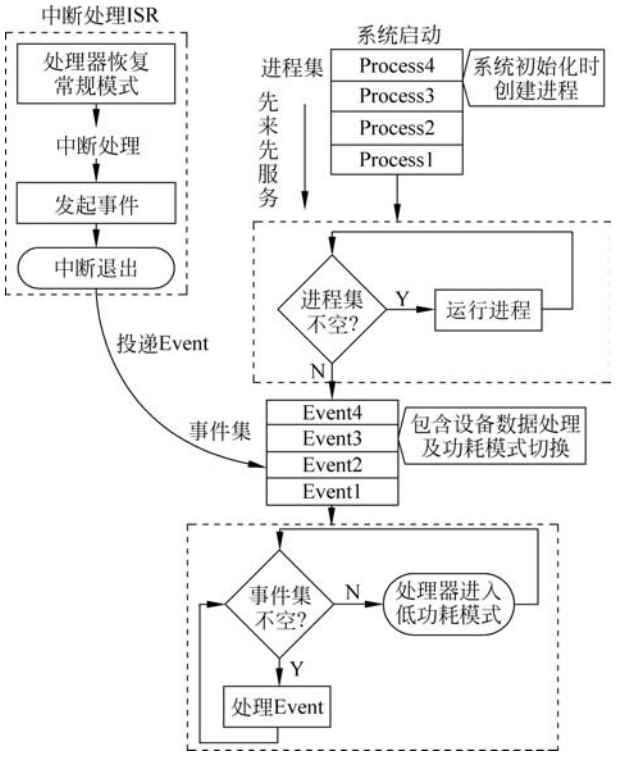

图 3-17　Contiki 运行架构图

3.4.2　任务管理

Contiki 中的任务是 process。一个 process 结构包含名字、入口地址、状态、是否需要轮询标识及上下文栈。系统内核通过单向链表管理所有任务，任务控制块中的 next 指针指向下一个任务；name 描述任务名称；thread 指向任务入口地址；pt 是任务的控制标签；state 标识任务状态中 needspoll 标识任务是否需要轮询，当需要轮询时，处理该任务。

任务调度方式是先来先服务，没有优先级，任务间不可强占，只有中断能够强占任务的执行。系统内的 process 都是挂载在一条链上，指针 next 指向下一个 process，在任务调度时，摘取该链链头上的 process 运行。任务的名字由 name 记录，指示该任务名称。任务的入口地址由 thread 记录，入口函数有 3 个参数，分别是任务游标、事件、数据。这里的参数事件指定了该任务需要处理的事件。

```
struct process {
struct process * next;
#if PROCESS_CONF_NO_PROCESS_NAMES
#define PROCESS_NAME_STRING( process) " "
#else
const char * name;
#define PROCESS _ NAME _ STRING ( process) ( process) ->
name
#endif
PT_THREAD( ( * thread) ( struct pt * ,process_event_t,
process_data_t) ) ;
struct pt pt;
unsigned char state,needspoll;
};
```

系统中的任务调度服务由 process_run 接口负责。该接口根据系统中是否有任务需要处理循环处理每个任务。如果系统中有任务,则循环遍历任务链表上的任务,逐个处理,直到任务链表为空。

这种类型的任务属于轻型任务,结构简单,占用资源少,调度简单,不需要优先级。任务一次性执行,不会被其他任务抢占,没有时间片轮转,只有在调用系统中的阻塞服务时才会被阻塞。这种轻型任务设计约束在设计任务处理过程中,尽量将任务处理流程精简化,以免任务运行时间过长影响其他任务的执行。

```
Int process_run( void)
{
if( poll_requested) {
do_poll( ) ;
}
do_event( ) ;
return nevents + poll_requested;
};
```

3.4.3 事件机制

中断与任务及任务与任务间是通过事件进行交互的。Contiki 系统的事件分为同步事件和异步事件。同步事件意味着事件发送者任务在发送完成事件后,立即执行接收者任务。事件发送者任务通过调用接口 process_post_synch 主动切换到接收者任务。

异步事件是通过事件队列维护当前的一些事件。事件发送者在发送事件时将事件添加到事件队列,该事件标识了接收者。系统在任务调度结束后,会查询事件队列上的事件,如果有事件,则弹出事件,解析事件接收者,再执行接收者任务,由接收者任务完成事件处理。

Contiki 中定义的事件有 9 种,如表 3-6 所示。事件接收者只能是任务,而发送者可以是中断,也可以是任务,还可以是广播任务。任务发送时,指定了接收者。发送同步事件时,发送者执行完成事件发送后,调用同步服务接口切换到接收者执行任务,同步事件发起者只能是任务。异步事件由发起者将事件插入到系统事件队列中。

表 3-6　Contiki 事件类型表

序号	事件标识	事件描述
1	EVENT_NONE	不是具体事件
2	EVENT_INIT	用于进程初始化
3	EVENT_POLL	事件的轮询标记
4	EVENT_EXIT	进程结束，释放或解除相关资源
5	EVENT_CONTINUE	等待释放处理器事件
6	EVENT_MSG	任务间传递消息
7	EVENT_EXITED	通知其他进程，某个进程结束
8	EVENT_TIMER	事件时钟失效事件
9	EVENT_COM	串口通信事件

系统通过事件队列维护所有的异步事件，在没有任务可调度的情况下，系统再调度事件队列中的事件，解析事件接收者，并调用接收者处理事件。由于系统资源受限，事件队列大小在系统配置时固定，只能维护固定数量的异步事件。如果系统异步事件过多，没能及时处理，则事件会被忽略。

综合来看，Contiki 作为一款典型的物联网操作系统，具备物联网应用需求的基本功能，能够作为占用资源极少，功耗极低，并且简化物联网应用的开发模式。

3.5　Windows 10 IoT Core

Windows 10 IoT Core 是面向各种智能设备的 Windows 10 版本系列，涵盖了从小的行业网关到大的更复杂的设备（如销售点终端和 ATM）的各种应用。结合最新的 Microsoft 开发工具和 Azure IoT 服务，合作伙伴可以收集、存储和处理数据，从而打造可行的商业智能和有效的业务结果。在构建基于 Windows 10 IoT Core 的解决方案后，合作伙伴将在利用一系列 Microsoft 技术提供端到端的解决方案时发现更多机会。

由于 Windows 10 IoT Core 是全新产品，它在用户群和经验丰富的开发者方面显然落后于其他许多物联网操作系统。但这款操作系统大有潜力，如果开发者希望在内部开发应用程序，更是如此。最终，那些习惯使用 Visual Studio 和 Azure 物联网服务，针对 Windows 从事开发工作的人会被整套 Windows 10 IoT Core 方案吸引过去。

Microsoft 更强调在 Windows 10 上提出的 Windows One 策略，即希望一个 Windows 适应所有的设备和屏幕，并为用户及开发人员提供一致的体验。这种方式使该系统具有强大的功能，但是势必导致其体量过大。目前 Windows10 IoT Core 提供两个版本，分别针对有显示屏和无显示屏两种场景[也叫有头和无头模式（headed or headless mode）]。无头模式需要 256MB 内存和 2GB 存储，有头模式需要 512MB 内存和 2GB 存储。

Windows 10 for IoT 相关文档可以登录 https://docs.microsoft.com/zh-cn/windows/iot-core/windows-iot-core 页面获得更多信息和资料。

3.6　Ostro

Ostro 是由 Intel 主导创建的一个开源物联网操作系统项目，它的目的是开发一个针对物联网应用的专门操作系统，这个操作系统的名字是 Ostro。它是基于 Linux 内核进行裁

剪,并针对物联网领域的智能设备进行定制,专门应用于物联网的操作系统。

Ostro 可被安装在 USB 存储器或者 SD 卡上,可以直接启动物联网硬件设备。当然,物联网应用开发者也可以根据自己的需要,对 Ostro 进行二次裁剪,自定义一个符合自身应用场景的全新内核。这个特征完全符合物联网操作系统的要求。

Ostro 支持 Intel 的 Quark 和 Intel Atom 处理器,支持采用 Node. JS、Python、Java 和 C/C++ 等语言进行应用程序开发。程序员可通过 RestFUL API 对设备状态进行查询。Ostro 支持符合 OCF 标准的设备发现机制和符合 OCF 标准的 JavaScript API。Ostro 具有较高的安全等级,包含可信启动、应用程序内存隔离、权限管理、OS 镜像完整性验证等安全机制。Ostro 具有丰富的通信技术支持,包括 Bluetooth/BLE、WiFi、6LowPAN 以及 CAN 总线等。最后,Ostro 支持 VirtualBox 虚拟机。

图 3-18 示意了 Ostro 物联网操作系统的整体架构。

图 3-18 Ostro 物联网操作系统的整体架构

在硬件层之上的 Linux 内核,Ostro 的内核就是通用的 Linux 内核,它包括了最基本的驱动程序支持、硬件适配支持、网络支持、文件系统以及设备管理机制等。为了适应物联网的应用,Ostro 对 Linux 内核做了一些微调,使得内核可以支持更多的传感器(Sensor),能够支持更多的连接类型,比如蓝牙、WiFi、ZigBee 等。但是由于 Linux 内核本身的复杂性和不可分割性,使得 Ostro 物联网操作系统很难满足物联网操作系统所应该具备的高度伸缩性要求。

Linux 内核是 Ostro 基本库。Ostro 基本库包括随 Linux 内核一起发行的最基本运行库,比如最常用的 C 运行库等。当然,Ostro 可以根据需要,动态地扩展基本库的范围。

Ostro 服务在基本库层的上一层次。Ostro 服务主要是指系统级的一些进程或线程,这些进程或线程负责管理网络连接,加载必要的支撑服务,以及提供进程间通信(IPC)支持等。在 Ostro 操作系统中,保留了大部分 Linux 操作系统所支持的 systemd、D-Bus 等。除此之外,在线软件更新也是 Ostro 提供的基本服务之一。这是专门为物联网应用提供的一个基本服务,可以快速完成物联网设备的软件更新,而且只需要最小的软件下载量,只需要重新启动必要的物联网设备,而不需要重新启动所有的物联网设备。在线软件更新是确保物联网可管理、可维护的核心机制,通过物联网操作系统与后端云平台的协同,使得物联网设备的软件始终保持在最新和最安全的状态。

物联网协同框架是 Ostro 操作系统非常重要的组成部分。Ostro 内置了对 IoTivity 的支持。2.4.5 节中已经介绍过,IoTivity 是一个开源的软件框架,用于无缝地支持设备到设备的互联,以及人与设备的简便互联。其主要是为了满足物联网开发的需要,构建物联网的生态系统,使得设备和设备之间可以安全可靠地连接。IoTivity 通过提供一系列框架和服务加速设备的互联应用开发。该项目由 Open Interconnect Consortium(OIC) 组织赞助,相当于是 OIC 标准的一个参考实现。

编程接口是 Ostro 提供给应用程序开发者使用的,用于开发各种各样的物联网应用程序。就目前情况来说,Ostro 提供了多种多样的编程接口供程序员根据自己的喜好和特定

应用场景调用。主要有：

- Java 和 Python 编程接口，物联网应用程序开发者可以采用 Python 和 Java 语言，开发特定的应用程序。Ostro 提供了常用的支持类库。
- Node. JS 编程接口。Ostro 提供了 Node. JS 的运行期支持，以及特定的一些 JavaScript API(以 Node. JS 模块方式提供)。这些 Java Script API 涵盖了相对广泛的物联网应用场景，比如包含了开放连接基金会(OCF)定义的 API 接口。这样就非常便于物联网应用程序开发者直接使用这些 API，调用 IoTivity 等协同框架的功能。
- Soletta 编程接口。Soletta 是一个开源的物联网应用程序开发框架，它提供了一些常用的物联网应用开发库，便于程序员方便快速地开发物联网应用程序。Soletta 是一种编程框架，可以采用传统的 C 语言进行应用程序开发，也可以采用一种叫作“基于流的编程语言”(Flow-based Programming)来进行物联网应用的开发。

总之，Ostra 提供了相对丰富的编程框架，供应用开发者选择。

IoT 应用程序位于整个系统的顶层。这个层次包含了所有使用 Ostro 编程接口所开发的物联网应用程序。当前的 Ostro 版本并没有开发任何特定的应用程序实例，仅提供了如何开发应用程序的指导以及一些简单的代码片段。随着 Ostro 的发展，会有针对特定典型场景的物联网应用程序(比如智慧家庭应用程序)，被纳入这个层次中发布。

官方网站地址为 https://ostroproject. org/。

3.7　AliOS Things

阿里巴巴云 IoT 物联网操作系统(又名 AliOS Things)是阿里巴巴云面向物联网领域的、高可伸缩物联网操作系统。AliOS Things 致力于搭建云端一体化 IoT 基础设施，具备极致性能、极简开发、云端一体、丰富组件、安全防护等关键能力。AliOS Things 支持多种多样的设备连接到阿里巴巴云 Link 平台，可广泛应用于智能家居、智慧城市、工业、新出行等领域。AliOS Things 开源代码遵循 Apache 2. 0 license 开源协议。

AliOS Things 架构适用于分层架构和组件化架构。一般来说，从底部到顶部，AliOS Things 包括板级支持包、硬件抽象层、内核(包括 Rhino 实时操作系统内核、Yloop、VFS、KV 存储)、协议栈[包括 TCP/IP 协议栈(LwIP)、uMesh 网络协议栈]、安全组件、中间件(包括常见的物联网组件和阿里巴巴增值服务中间件)和示例应用(包含各种示例代码)。

AliOS Things 的内核是 Rhino 实时操作系统内核，其特点主要有：

(1) 体积小。为大多数的内核对象提供静态和动态分配。为小内存块设计的内存分配器。大部分的内核特性都是可以裁剪的。通过. h 文件进行配置和裁剪。K_config. h 文件一般位于开发板目录结构下。

(2) 功耗低。提供了 CPU 的 tickless idle 模式来帮助系统节约电能和延长时间。类似于 FREERTOS 的 tickless idle 模式。

(3) 实时性。Rhino 提供了两个调度策略：基于优先级的抢占式调度和 round-robin 循环调度策略。对于这两个调度策略而言，具有最高优先级的任务都是被优先处理的。

(4) 调试方便。Rhino 可以支持 stack 溢出、内存泄漏、内存损坏的检测。

下面介绍 AliOS Things 的基本管理功能。

1. 任务管理

现代操作系统都建立在任务的基础上,任务是内核 Rhino 中代码的一个基本执行环境,有的操作系统也称之为线程(thread)。

多任务的运行环境提供了一个基本机制让上层应用软件来控制/反馈真实的/离散的外部世界,从宏观上可以看作单个 CPU 执行单元上同时执行多个任务;从微观上看,CPU 快速地进行切换来执行任务。Rhino 实时操作系统支持多任务机制。

每个任务都具有上下文(context)。上下文是指当任务被调度执行的时候此任务能看见的 CPU 资源和系统资源,当发生任务切换的时候,任务的上下文被保存在任务控制块(ktask_t)中,这些上下文包括当前任务的 CPU 指令地址(PC 指针)、当前任务的栈空间、当前任务的 CPU 寄存器状态等。

任务管理功能的相关源码位于/kernel/rhino/core/目录中。

头文件内容如下:

```
#include <aos/aos.h>
#include <aos/kernel.h>
#include "k_task.h"
```

表 3-7 显示了内核 Rhino 的任务管理 API 情况。

表 3-7 任务管理 API 列表

API 名称	说 明
aos_task_new()	动态创建一个任务,任务句柄不返回,创建完成后自动运行
aos_task_new_ext()	动态创建一个任务,传入任务句柄,并指定优先级,创建完成后自动运行
aos_task_exit()	任务自动退出
aos_task_delete()	任务删除
aos_task_name()	返回任务名
aos_task_key_create()	返回任务私有数据区域的空闲块索引(目前用于 yloop,2.1 版本后不用于 yloop)
aos_task_key_delete()	删除任务私有数据区域的空闲块索引(目前用于 yloop,2.1 版本后不用于 yloop)
aos_task_setspecific()	设置当前任务私有数据区域的某索引空闲块内容(目前用于 yloop,2.1 版本后不用于 yloop)
aos_task_getspecific()	获取当前任务私有数据区域的某索引数据块内容(目前用于 yloop,2.1 版本后不用于 yloop)
aos_msleep()	任务挂起若干毫秒

2. 内存管理

内存管理是指软件运行时对系统内存资源进行分配和使用的技术。其最主要的目的是如何高效、快速地分配,并且在适当的时候释放和回收内存资源。表 3-8 列举了内核 Rhino 的内存管理常用 API。

表 3-8　内存管理 API 列表

API 名称	说　明
aos_malloc()	从系统堆中分配内存给用户
aos_zalloc()	从系统堆中分配内存区域给用户，并且将分配的内存初始化为 0
aos_calloc()	从系统堆中分配连续的块内存区域给用户，并且将分配的内存初始化为 0
aos_realloc()	重新调整之前调用 aos_malloc(aos_calloc、aos_zalloc)所分配的内存块的大小
aos_free()	释放分配的内存

3. 定时器管理

AliOS Things 提供基本的软件定时器功能，包括定时器的创建、删除、运行，以及单次和周期定时器。定时器管理涉及 tick。

tick 一般是作为任务延迟调度的内部机制，其接口主要是系统内部使用。对于使用操作系统的应用软件，也需要定时触发相关功能的接口，包括单次定时器和周期定时器。

从用户层面来讲，不关注底层 CPU 的定时机制以及 tick 的调度。用户希望的定时器接口是，可以创建和使能一个软件接口定时器，时间到了之后，用户的钩子函数能被执行。而对于操作系统的定时器本身来讲，也需要屏蔽底层定时模块的差异。因此，在软件层次上，对于定时器硬件相关的操作由 tick 模块完成，定时器(timer)模块基于 tick 作为最基本的时间调度单元，即最小时间周期，来推动自己时间轴的运行。表 3-9 列举了内核 Rhino 的定时器管理模块的常用 API。

表 3-9　定时器管理 API 列表

API 名称	说　明
aos_timer_new()	动态创建软件定时器
aos_timer_start()	软件定时器启动
aos_timer_stop()	软件定时器停止
aos_timer_change()	改变软件定时器的周期
aos_timer_free()	删除软件定时器

4. 信号量(semphore)

该处理方式可以避免软件在访问共享资源的读写时发生相互影响甚至冲突。

对于多任务，甚至多核的操作系统，需要访问共同的系统资源。共享资源包括软件资源和硬件资源，软件共享资源主要在于共享内存，包括共享变量、共享队列等，硬件共享资源包括一些硬件设备的访问，例如，输入/输出设备、打印机等。

为了避免软件访问共享资源的读写发生的相互影响甚至冲突，一般在保护共享资源时，有下列几种处理方式。

- 开关中断：一般用于单核内多任务之间的互斥，其途径在于关闭任务的调度切换，从而达到单任务访问共享资源的目的。缺点是会影响实际的中断调度效率。
- 信号量：多任务可以通过获取信号量来获取访问共享资源的“门禁”，可以配置信号量数目，让多个任务同时获取“门禁”，当信号量无法获取时，相关任务会按照优先级排序等待信号量释放，并让出 CPU 资源。信号量的缺点是存在高低任务优先级反转的问题。

- 互斥量：任务也是通过获取 mutex 来获取访问共享资源的门禁，但是只有一个任务能获取到该互斥量。互斥量通过动态调整任务的优先级来解决高低优先级反转的问题。表 3-10 列举了内核 Rhino 的信号量常用 API。

表 3-10 信号量 API 列表

API 名称	说 明
aos_sem_new()	动态创建信号量
aos_sem_free()	删除信号量
aos_sem_signal()	释放一个 sem 信号量，并唤醒一个高优先级阻塞任务
aos_sem_signal_all()	释放一个 sem 信号量，并唤醒所有阻塞任务
aos_sem_wait()	信号量获得
aos_sem_is_valid()	判断信号量是否有效

5. 工作队列

在内核中，用户只需要创建一次工作队列(workqueue)即可构建多个挂载不同处理函数的工作队列。

在一个操作系统中，如果需要进行一项工作，往往需要创建一个任务来加入内核的调度队列。一个任务对应一个处理函数，如果要进行不同的事务处理，则需要创建多个不同的任务。任务作为 CPU 调度的基础单元，任务数量越大，则调度成本越高。工作队列机制简化了基本的任务创建和处理机制，一个工作队列对应一个实体任务处理，工作队列下面可以挂载多个工作实体。

当在某些实时性要求较高的任务中，需要进行较繁重的钩子处理时，可以将其处理函数挂载在工作队列中，其执行过程将位于工作队列的上下文，而不会占用原有任务的处理资源。工作队列还提供了工作的延时处理机制，用户可以选择立即执行或是延时处理。

由此可见，在需要创建大量实时性要求不高的任务时，可以使用工作队列来统一调度；或者将任务中实时性要求不高的部分处理延后到工作队列中处理。如果需要设置延后处理，则需要使用工作机制。另外该机制不支持周期工作的处理。

工作队列功能的相关源码位于/kernel/rhino/目录中(v2.1.0 之前位于/kernel/rhino/core/目录中)。

头文件内容如下：

```
#include <aos/aos.h>
#include <aos/kernel.h>
#include "k_workqueue.h"
```

表 3-11 列举了内核 Rhino 的常用工作队列 API。

表 3-11 工作队列 API 列表

API 名称	说 明
aos_workqueue_create()	创建一个工作队列，内部会创建一个任务关联工作队列
aos_work_init()	初始化一个工作，暂不执行
aos_work_destroy()	删除一个工作

续表

API 名称	说　　明
aos_work_run()	运行一个工作，使其在某工作队列内调度执行
aos_work_sched()	运行一个工作，使其在默认工作队列 g_workqueue_default 内调度执行
aos_work_cancel()	取消一个工作，使其从所在的工作队列中删除

6. 内核配置文件 k_config.h

每一个 AliOS Things 支持的单板，都配套有一个 k_config.h，用于设定本单板环境下特定的内核 Rhino 配置。

AliOS Things 的内核（Rhino）可以通过宏进行功能配置。完整的配置宏可以在 k_default_config.h 文件中看到，里面的宏可分为两类——开关类与数值设置类。开关类负责打开或关闭一个内核模块，数值设置用于设定一些参数。k_default_config.h 文件位于 Rhino 内核代码中（文件路径：./kernel/rhino/include/k_default_config.h），其对每个可配置宏都进行了默认值的设置。

k_config.h 文件中的宏配置值通常与 k_default_config.h 值不同。k_config.h 位于 \board*\目录下，其中星号（*）为具体的单板名称。

AliOS Things 内部组件都是通过“#include "k_api.h""来使用这些配置宏的，k_api.h 中固定包含顺序如下。

```
#include "k_config.h"
#include "k_default_config.h"
```

因此，单板特定的宏设置优先于默认设置，以信号量功能开关为例，可以在以下代码中看出最终 RHINO_CONFIG_SEM 生效值为 1，即信号量功能打开。

表 3-12 列举了内核 Rhino 的常用配置选项。

表 3-12　常用配置选项说明

配置项名称	功能描述
RHINO_CONFIG_SEM	信号量模块的开关，0 表示关闭，1 表示打开。主要对应 k_sem.h 中的功能
RHINO_CONFIG_TASK_SEM	任务信号量模块的开关，0 表示关闭，1 表示打开。主要对应 k_task_sem.h 中的功能。对比信号量用于同步或互斥场景，任务信号量只用于同步，提供更高效方便的方式
RHINO_CONFIG_QUEUE	队列模块的开关，0 表示关闭，1 表示打开。主要对应 k_queue.h 中的功能
RHINO_CONFIG_BUF_QUEUE	缓存队列模块的开关，0 表示关闭，1 表示打开。主要对应 k_buf_queue.h 中的功能
RHINO_CONFIG_PWRMGMT	功耗管理模块的开关，0 表示关闭，1 表示打开。用于开启低功耗功能（该功能需要厂商 BSP 配合 OS 一同完成）
RHINO_CONFIG_TIMER	timer 模块的开关，0 表示关闭，1 表示打开。主要对应 k_timer.h 中的功能

续表

配置项名称	功能描述
RHINO_CONFIG_TIMER_TASK_PRI	timer 模块打开时,定时器超时回调都在内核创建的定时器任务上下文中执行。该任务优先级通过上述宏配置。timer 任务优先级与用户的回调实际工作有关,通常优先级设定的较高
RHINO_CONFIG_TIMER_TASK_STACK_SIZE	timer 模块打开时,定时器超时回调都在内核创建的定时器任务上下文中执行。该任务栈大小通过上述宏配置,单位是 4B(例如,宏值默认设定为 256,表示任务栈实际为 1024B 大小)。 timer 任务栈大小与用户的回调实际工作有关,初始可以设定大一点,运行时通过 cli 的 tasklist 命令查看,若 timer 任务(timer_task)栈最小空闲值较大,则可以减小该宏以节省内存
RHINO_CONFIG_SCHED_RR	任务 round robin 调度方式开关,0 表示关闭,1 表示打开。Rhino 为实时调度内核,即高优先级任务会持续优先于低优先级任务执行。而对于相同优先级的任务,则有两种调度策略。 (1) RR,即相同优先级任务分享时间片,每个任务执行到一定时间后自动让出 CPU,供下一个同优先级任务执行。 (2) FIFO,即相同优先级任务先进入 ready 状态的先执行,只有本任务发生阻塞(例如,sleep 或者等待信号量等)后,才能轮到相同优先级下一个任务执行。 RHINO_CONFIG_SCHED_RR 为 0 和 1 分别对应 RR 与 FIFO 方式
RHINO_CONFIG_TIME_SLICE_DEFAULT	当任务 round robin 调度方式打开时(即 RHINO_CONFIG_SCHED_RR 为 1),每个任务的时间片可在创建时指定,若创建时填写 0 则为该宏的默认值。单位为毫秒
RHINO_CONFIG_TICKS_PER_SECOND	配置每秒系统的 tick 数。例如,100 表示每 10ms 到来一个系统 tick,1000 表示每 1ms 都有一个 tick。系统 tick 是内核计时的基础单位。超时时间(如 sleep、sem_take 等)与定时器控制,内部都以 tick 为计数基础。因此该宏值越高,表示计时精度越高,但系统处理 tick 中断本身的消耗也就越大
RHINO_CONFIG_SYSTEM_STATS	内核系统统计开关,0 表示关闭,1 表示打开。打开后完成以下统计。统计全系统的最长关中断时间、最长关任务调度时间与任务切换次数
RHINO_CONFIG_MM_TLF	堆管理算法开关,0 表示关闭,1 表示打开。打开后 Rhino 接管 malloc、free 等 C 库的内存管理,并提供 k_mm.h 中的功能
RHINO_CONFIG_MM_BLK	小内存块优化算法开关,0 表示关闭,1 表示打开。 RHINO_CONFIG_MM_TLF 打开后,Rhino 使用 TLF 算法管理内存,该算法更加健壮但会消耗一定的内存。针对小内存块(例如,小于 32B),可以通过 RHINO_CONFIG_MM_BLK 宏开启 BLK 算法优化,提高内存利用率
RHINO_CONFIG_MM_TLF_BLK_SIZE	小内存块优化空间大小,单位为字节。 RHINO_CONFIG_MM_BLK 打开后,需要配置 RHINO_CONFIG_MM_TLF_BLK_SIZE 来决定堆中多少内存划分给 BLK 算法
RHINO_CONFIG_MM_DEBUG	缓存队列模块的开关,0 表示关闭,1 表示打开。主要对应 k_mm_debug.h 中的功能。打开后,当用户申请内存不足,或者 rhino 检测到内存越界时,都会有详细的输出信息。CLI 中也有 dumpsys mm_info 命令可以查看详细内容

3.8　μT/OS

大连悠龙软件科技有限公司从2008年开始借鉴Google在Android上的成功商业模式，以μT-Kernel规范为基础，2009年年底研发出世界上第一个支持Cortex-M3和μT-Kernel规范的实时操作系统内核，后来逐渐加上Linux上的成熟轻量级开源中间件，推出了中国人自己的物联网开源实时操作系统——μTenux，在μTenux中遵循μT-Kernel规范的内核被命名为μT/OS。μTenux支持Cortex-M0/M3/M4、ARMV4T、ARMV5E等多种32位内核微控制器，在2010年和2011年陆续成为ATMEL和ARM公司全球操作系统战略合作伙伴。

近期μT/OS v3.0已经启动，支持ST全系列Nucleo开发板，支持STM32 Cube库，支持动态下载程序，增加安全API。

Github地址为https://github.com/TenuxOS。

3.9　MiCO

MiCO IoT OS由上海庆科联合阿里巴巴智能云于2014年7月发布，是国内首款真正意义上的物联网操作系统。MiCO(MCU based Internet Connectivity Operating System)是一个基于微控制器的互联网接入操作系统，是一个开发物联网设备的软件平台。

MiCO内含一个面向IoT设备的实时操作系统内核，特别适合运行在资源受限的微控制设备上。MiCO包含了底层芯片驱动、无线网络协议、射频控制技术、应用框架，此外，MiCO还包含了网络通信协议栈、安全算法和协议、硬件抽象层、编程工具等开发IoT必不可少的软件功能包。它提供MCU平台的抽象化，使得基于MiCO的应用程序开发不需要关心MCU具体件功能的实现，通过MiCO中提供的各种编程组件快速构建IoT设备软件。简单地说，它是基于MCU的全实时物联网操作系统，是面向智能硬件设计、运行在微控制器上的高度可移植的操作系统和中间件开发平台，已被广泛应用于智能家电、照明、医疗、安防、娱乐等物联网应用市场。

开发者可以在各种微控制器平台上基于MiCO来设计接入互联网的创新智能产品，实现人物互联。

3.10　Ruff

Ruff是一个支持JavaScript开发应用的物联网操作系统，为软件开发者提供开放、高效、敏捷的物联网应用开发平台，让IoT应用开发更简单。

Ruff对硬件进行了抽象，使用了基于事件驱动、异步I/O的模型，使硬件开发变得轻量而且高效。除了使用JavaScript作为开发语言，它还拥有自己的软件仓库，从模块到驱动一应俱全，提高了软件兼容性，降低了硬件开发门槛。

整个Ruff开发体系包括Ruff OS、Ruff SDK、Ruff软件仓库、Ruff Kit开发套件。只要用户有软件开发经验，就可以用Ruff开发硬件应用。

3.11 Zephyr

Linux基金会宣布了一个微内核项目——Zephyr,由 Intel 主导,风河提供技术。Zephyr 微内核将被用于开发针对物联网设备的实时操作系统。Zephyr 项目得到了 Intel、NXP 半导体、Synopsys 和 UbiquiOS 等公司的支持,Intel 子公司 Wind River 向 Zephyr 项目捐赠了它的 Rocket RTOS 内核。Zephyr 微内核能运行在只有 10KB RAM 的 32 位微控制器上,相比之下基于 Linux 的微控制器项目 uClinux 需要 200KB RAM。

官方网站 https://www.zephyrproject.org/。

3.12 TinyOS

TinyOS 是 UC Berkeley(加州大学伯克利分校)开发的开放源代码操作系统,专为嵌入式无线传感网络设计,操作系统基于构件(component-based)的架构使得快速的更新成为可能,而这又缩短了受传感网络存储器限制的代码长度。TinyOS 是一个具备较高专业性,专门为低功耗无线设备设计的操作系统,主要应用于传感器网络、普适计算、个人局域网、智能家居和智能测量等领域。

TinyOS 作为一个专业性非常强的操作系统,主要存在如下几个特点。

1. 拥有专属的编程语言

TinyOS 应用程序都是用 NesC 编写,其中 NesC 是标准 C 的扩展,在语法上和标准 C 没有区别,它的应用背景是传感器网络这样的嵌入式系统,这类系统的特点是内存有限,且存在任务和中断两类操作,它的编译器一般都是放在 TinyOS 的源码工具路径下。

2. 开放源代码

所有源码都免费公开,可以访问官方网站 www.tinyos.net 去下载相应的源代码,由全世界的 TinyOS 的爱好者共同维护,目前最新的版本是 2.1.1。

3. 基于组件的软件工程建构

inyOS 提供一系列可重用的组件,一个应用程序可以通过连接配置文件(AWiringSpecification)将各种组件连接起来,以完成它所需要的功能。

4. 通过任务和事件来管理并发进程

Tasks:一般用在对于时间要求不是很高的应用中,且任务之间是平等的,即在执行时是按先后顺序执行的,一般为了减少任务的运行时间,要求每一个任务都很短小,能够使系统的负担较轻;支持网络协议的替换。

事件:一般用在对于时间的要求很严格的应用中,而且优于任务执行,它可以被一个操作的完成或是来自外部环境的事件触发,在 TinyOS 中一般由硬件中断处理来驱动事件。

5. 支持网络协议组件的替换

除了默认协议之外,还提供其他协议供用户替换,并且支持客户自定义协议,这对于通信协议分析以及通信协议的研究工作非常有帮助。

6. 代码短小精悍

TinyOS 的程序采用的是模块化设计,所以它的程序核心往往都很小。一般来说,核心

代码和数据为400B左右；能够突破传感器存储资源少的限制，这能够让TinyOS很有效地运行在无线传感器网络上并去执行相应的管理工作等。

TinyOS的特性决定了其在传感器网络中的广泛应用，使其在物联网中占据了举足轻重的地位。

相对于主流操作系统的庞大体积来说，TinyOS显得十分迷你，只需要几千字节的内存空间和几十千字节的编码空间就可以运行，而且功耗较低，特别适合传感器这种受内存、功耗限制的设备。

TinyOS在构建无线传感器网络时，通过一个基本控制台控制各个传感器子节点，聚集和处理各子节点采集到的信息。TinyOS在控制台发出管理信息，然后由各个节点通过无线网络互相传递，最后达到协同一致的目的。

更多内容请参考http://tinyos.stanford.edu/tinyos-wiki/index.php/Main_Page。

3.13　小结

本章介绍了目前主流的物联网操作系统，表3-13对典型的物联网操作系统及特性做了简要总结。

表3-13　典型物联网操作系统及其特性

操作系统	特性简述
Contiki	支持平台较多，能在多平台（如嵌入式设备和传感器等）上运行，较容易开发
Android Things	使用Weave的通信协议，实现设备与云端相连，并且与Google助手等服务交互
ARM Mbed	ARM处理器专用，采用事件驱动的单线程架构，可用于尺寸小、低功耗的物联网设备
Lite OS	华为公司开发的轻量级的物联网操作系统，具备零配置、自组网、跨平台的能力
Ruff	JavaScript编程，跨平台
Ostro	基于Linux操作系统进行裁剪，丰富的通信技术支持
RT-Thread	开源，组件完整丰富、高度可伸缩、简易开发、低功耗

从2014年ARM Mbed OS发布开始计算，当前市场已经有几十种开源的IoT OS，还有一些商业IoT OS，更准确地说，是支持IoT应用的商业嵌入式操作系统。在一个新的物联网项目启动的时候，开发者通过芯片公司生态系统能很方便地接触到1或2种支持IoT OS的开发板，比如STM32 Discovery kit IoT node。从当前主流的物联网操作系统技术来看，IoT OS更趋向是一种集成技术，将已经成熟的操作系统、通信和云计算技术集成到从传感器到云的物联网场景中。IoT OS不只是提供CPU资源管理和应用编程接口（API）的传统意义的操作系统，IoT OS也无法只布置设备端，它需要端云联动。IoT OS一直由产业界在推动其发展，产业界也在寻找可以解决物联网开发过于烦琐、开发团队顾此失彼而延误开发周期的问题。

习题

1. 简述 RT-Thread 的系统结构。
2. RT-Thread 系统的内存管理有何特点？
3. 简述 Mbed OS 的功能框架。
4. CMSIS 包含哪些组件？
5. 简述 Contiki 的系统架构。
6. 简述 Contiki 的任务管理机制。
7. 尝试下载本章介绍的某个物联网操作系统内核镜像并安装测试。