RT-Thread

简介

裸机开发与操作系统的区别

在嵌入式系统的开发中，开发者通常面临两种选择：**裸机开发**和**基于操作系统的开发**。这两种开发模式的最大区别在于对硬件的管理、任务调度方式以及系统资源的利用效率。以下是这两者的关键区别：

裸机开发

裸机开发（Bare-metal programming）指的是**直接操作硬件**，程序在没有操作系统的环境中运行。通常，裸机程序会从系统启动后直接运行，开发者需要手动管理所有硬件资源和软件流程。q

- **单一任务运行**
  裸机开发中的程序通常是线性执行的，即从主函数开始，按照代码的先后顺序执行，缺少复杂的任务管理。没有多任务调度的机制，意味着程序一次只能执行一个任务，其他功能通常是通过**中断**实现的。
- **手动管理资源**
  在裸机开发中，开发者需要自行管理CPU、内存、外设等硬件资源。例如，计时器、中断、串口等外设都需要手动配置和控制。对于复杂系统，这会导致代码可读性和可维护性下降。
- **缺乏抽象**
  裸机开发没有提供硬件抽象层，开发者需要深入理解硬件的工作原理，并直接通过寄存器操作硬件设备。这种方式虽然灵活，但对于复杂系统开发来说，难以扩展和维护。

操作系统

基于操作系统（Operating System, OS）的开发则不同，操作系统为应用程序提供了**资源管理**、**任务调度**等功能，使得开发者无需关心底层硬件的细节，专注于应用逻辑的编写。

- **多任务调度**
  操作系统的核心是提供**多任务并发**能力。通过任务调度器（Scheduler），操作系统可以管理多个线程或任务的执行时间，使得多个任务能够同时“运行”（实际上是快速切换），提高了系统的响应性和效率。
- **自动资源管理**
  操作系统为开发者提供了**内存管理**、**设备管理**等抽象层，开发者不再需要直接操作硬件。操作系统负责分配和回收系统资源，如内存、CPU时间片等。这种管理方式大大降低了开发复杂度，并提升了系统的稳定性。

- **硬件抽象层（HAL）**
  操作系统通常提供硬件抽象层，使得应用程序与硬件解耦。开发者可以编写与硬件无关的代码，操作系统通过驱动程序控制底层硬件，增加了系统的可移植性。
- **实时性支持**
  对于嵌入式实时系统，操作系统（尤其是实时操作系统，如RT-Thread）提供了严格的**实时性保证**，可以确保关键任务在特定时间内完成。这对需要精确时序的任务，如电机控制、通信协议栈等场景尤为重要。

什么是RT-Thread

RT-Thread 是一款开源、轻量级的**实时操作系统（RTOS）**，专为资源受限的嵌入式设备设计。它提供了多线程管理、实时调度、设备驱动框架、组件与中间件支持，能够帮助开发者在嵌入式平台上快速开发出功能强大且实时性要求高的应用。

- **开源与可扩展性**
  RT-Thread采用**Apache 2.0许可证**，是一个开源项目，开发者可以自由使用、修改、分发。其模块化设计使得开发者可以根据项目需求裁剪功能模块，灵活配置系统，确保操作系统在资源受限的嵌入式设备上高效运行。
- **多平台支持**
  RT-Thread支持**ARM Cortex-M**、**RISC-V**、**MIPS**等多种嵌入式硬件架构，并且可以移植到不同的开发板上，如STM32、NXP、ESP32等。
- **实时性保障**
  作为一款RTOS，RT-Thread支持**硬实时调度**，能够确保系统在规定时间内响应外部事件，满足对时间要求严格的任务需求。这使其非常适用于自动控制、工业控制、物联网等场景。
- **丰富的组件与中间件**
  RT-Thread拥有丰富的组件库，如网络通信协议栈（如LWIP）、文件系统（FATFS、SPIFFS）、图形用户界面（GUI）等，能够轻松实现复杂嵌入式应用功能。
- **工具与开发生态**
  RT-Thread提供了专用的开发工具，如**RT-Thread Studio**，支持GCC、Keil、IAR等主流编译工具链，简化了开发与调试流程。活跃的社区也为开发者提供了丰富的学习资源和技术支持。

以下是**裸机开发**与**基于操作系统开发（以RT-Thread为例）**的对比列表：

对比项	裸机开发	基于操作系统开发（RT-Thread）
任务管理	单一任务，依次顺序执行。通过中断实现基本的异步任务处理。	支持多任务调度，操作系统自动管理线程并行，支持优先级调度。
资源管理	开发者手动管理所有资源（内存、CPU、外设等）。	操作系统自动分配和回收系统资源，开发者无需关心底层细节。

对比项	裸机开发	基于操作系统开发（RT-Thread）
中断处理	依赖中断服务程序(ISR)处理异步任务，可能导致系统复杂性增加。	支持中断服务程序，并提供线程与中断协作机制，简化复杂任务管理。
实时性	实时性由开发者手动控制，依赖硬件定时器和中断，复杂度高。	操作系统内置实时调度算法，支持硬实时应用，保证任务的时效性。
硬件抽象	没有硬件抽象，直接操作硬件寄存器。	提供硬件抽象层（HAL），开发者通过API调用，不需关心底层实现。
多线程支持	不支持多线程并行，需要通过中断或状态机手动模拟多任务。	原生支持多线程，提供线程管理API，简化并行任务的实现。
内存管理	需要手动管理内存，容易发生内存泄漏或内存不足问题。	提供动态内存管理和静态内存管理机制，自动处理内存分配与释放。
开发难度	适合简单项目，复杂系统开发难度大，需要手动处理大量细节。	开发者只需专注应用层开发，操作系统负责底层管理，开发更简单高效。
可移植性	与硬件紧密耦合，移植到不同平台的工作量大。	提供硬件抽象层和标准接口，代码更容易移植到不同硬件平台。
扩展性	功能有限，增加新功能时代码结构复杂，维护难度较大。	支持丰富的中间件和组件（网络、文件系统等），扩展性强，易于维护。
代码复杂度	随系统复杂度提升，代码维护性和可读性下降。	操作系统结构清晰，模块化设计，代码更具可读性，维护简单。
典型应用场景	简单嵌入式项目，如单一传感器控制、LED闪烁等。	复杂嵌入式系统，如多任务并发、物联网设备、工业控制等。
开发工具链	通常使用简单的编译工具链和调试工具。	提供专门的IDE（如RT-Thread Studio）和丰富的调试工具支持。

以下是 **RT-Thread** 和 **FreeRTOS** 的详细对比：

对比项	RT-Thread	FreeRTOS
开源协议	Apache 2.0	MIT License
内核架构	基于对象的模块化设计，支持丰富的组件和扩展功能。	简单的微内核架构，主要提供基本的任务调度和内存管理功能。
实时性	支持硬实时操作，提供灵活的实时调度算法，可以满足高实时性需求。	提供基本的实时调度，适合软实时应用，适当优化也可以用于某些硬实时应用场景。
硬件支持	支持广泛的硬件平台，如ARM Cortex-M、RISC-V、MIPS、ESP32等，并且可以通过硬件抽象层（HAL）进行扩展。	同样支持广泛的硬件平台，主要是ARM Cortex-M，但扩展性较弱，需要手动进行平台移植。

对比项	RT-Thread	FreeRTOS
线程管理	支持多线程调度、优先级调度、时间片轮转等机制，线程间支持消息队列、信号量、互斥锁等同步机制。	提供基本的任务管理，支持优先级调度和协作式多任务处理，但高级线程管理功能需要额外配置。
内存管理	支持动态内存和静态内存管理，包括小内存池和内存块分配机制，优化内存使用。	支持动态内存分配，也提供静态分配方式，但内存管理机制较为简单，需要开发者仔细管理内存分配和释放。
组件和中间件	提供丰富的组件库，如网络协议栈（LWIP、TCP/IP）、文件系统（FATFS、SPIFFS）、GUI（Persimmon）、多媒体等。	基本内核之外没有内置的中间件组件，第三方支持丰富，但通常需要额外集成和配置。
文件系统支持	支持FATFS、SPIFFS、ELM FAT、YAFFS等多种文件系统，且可以灵活选择和配置。	不直接支持文件系统，需要开发者自行集成外部文件系统，如FATFS等。
网络协议栈	内置轻量级网络协议栈（LWIP），也支持以太网、Wi-Fi、LoRa等协议，方便物联网应用。	支持通过第三方集成轻量级的TCP/IP协议栈（如LWIP），需要自行配置与集成，缺乏原生支持。
图形用户界面（GUI）	提供Persimmon GUI等图形界面中间件，支持嵌入式显示开发，适合开发带显示的嵌入式设备。	FreeRTOS本身不提供GUI支持，需要通过第三方GUI库集成，如LVGL等。
开发工具	提供RT-Thread Studio，官方IDE，集成了GCC工具链，支持可视化配置、调试与监控，开发体验好。	FreeRTOS不提供官方IDE，但支持多种IDE，如Keil、IAR、Eclipse等，开发环境灵活但需要手动配置。
社区与支持	活跃的社区，尤其在中国有庞大的用户群体和支持体系。官方提供详细的文档、教程和技术支持。	全球用户广泛，社区活跃，资源丰富，尤其在嵌入式开发领域有大量应用案例和支持文档。
扩展性	支持多种功能扩展，模块化设计，支持系统裁剪，适合资源受限的嵌入式系统和复杂应用。	FreeRTOS核心轻量，但功能单一，扩展性不如RT-Thread，复杂功能需要开发者自己集成第三方库。
实时调度支持	提供多种调度策略，包括抢占式调度、时间片轮转调度，支持软硬实时应用。	基本支持抢占式调度和协作式调度，适合软实时任务，硬实时场景支持较弱。
应用场景	适用于广泛的嵌入式系统，从物联网设备到工业控制、车载设备、消费电子等。	主要应用于小型嵌入式设备和物联网应用，适合轻量级、资源受限的系统。

为什么选择RT-Thread

1. **模块化设计与丰富的组件支持**
   RT-Thread提供了丰富的组件库和中间件，包括网络协议栈、文件系统、图形界面等，适合开发复杂的嵌入式应用。而FreeRTOS的核心虽然轻量，但很多功能需要通过第三方库集成，增加了复杂度和维护成本。
2. **良好的开发工具与生态支持**
   RT-Thread提供了官方的IDE（RT-Thread Studio），集成了可视化配置和调试功能，简化了开发和调试过程。而FreeRTOS虽然兼容多种IDE，但没有官方的统一开发工具，配置和调试需要更多手动工作。
3. **更强的实时性和线程管理功能**
   RT-Thread提供了丰富的线程管理机制和调度策略，能够满足软硬实时的需求。而FreeRTOS在硬实时任务调度上较弱，适合软实时或低实时性要求的应用。
4. **更灵活的内存管理**
   RT-Thread在内存管理上提供了多种优化机制，支持动态和静态内存管理，适用于不同资源限制的系统，而FreeRTOS的内存管理较为简单，开发者需要手动管理。
5. **广泛的应用场景**
   RT-Thread支持从资源受限的嵌入式设备到复杂的物联网设备、工业控制等多个领域，适用范围广泛，功能全面。而FreeRTOS主要面向轻量级的嵌入式应用，复杂系统的扩展能力相对较弱。

因此，对于需要丰富功能扩展、高实时性、强大工具支持的嵌入式项目，RT-Thread是一个更灵活、全面的选择，尤其适用于物联网、工业控制等复杂应用场景。

> ### 1. 裸机开发（Bare-metal Development）
>
> 裸机开发的特点：
>
> - **无操作系统**：系统中没有调度器、内核等操作系统组件，程序控制完全由开发者负责。
> - **简单高效**：裸机开发系统简单，代码更贴近硬件，开销低，实时性好，适合对响应时间要求极高的场景。
> - **硬件资源占用少**：由于没有操作系统的开销，裸机开发适合资源受限的硬件，如低端微控制器。
> - **开发复杂度高**：开发者需要自己处理所有任务调度、资源共享等，代码结构可能比较复杂。

适合裸机开发的场景：

- **资源非常有限**的设备：如简单的 8-bit/16-bit 微控制器，Flash 和 RAM 容量很小，通常不到几十 KB，不能承受操作系统的额外开销。
- **简单单任务系统**：系统只执行一个任务，如简单的传感器数据采集、信号灯控制、键盘扫描等。
- **实时性非常高**的场景：如一些需要确定性和微秒级精度的系统，操作系统的中断延迟和调度开销可能无法满足要求，裸机开发能提供更快的响应。
- **开发周期要求短**且应用简单：开发需求非常明确，系统设计简单且无需太多任务管理，如简单的机器人控制、家电控制器等。

裸机开发的典型例子：

- **单片机项目**：如使用 AVR、PIC、STM8、STM32F0 系列等开发的简单传感器控制、信号采集、工业设备控制等。
- **超小型设备**：如超低功耗传感器模块或手持设备中控制 LED 显示的简单逻辑。

2. 操作系统开发（OS-based Development）

操作系统开发的特点：

- **多任务并发**：操作系统支持多任务调度，可以同时运行多个任务，操作系统负责分配 CPU 资源，简化复杂应用的实现。
- **简化任务管理**：开发者无需手动管理任务切换、资源同步，操作系统提供线程、任务、队列、信号量、互斥量等机制，降低了管理复杂度。
- **实时性依赖操作系统**：实时操作系统（RTOS）可以提供确定性的任务调度和中断响应，但仍有一定开销，实时性能略低于裸机开发。
- **增加系统开销**：操作系统需要一定的内存和计算资源来运行，系统本身增加了复杂度。

适合操作系统开发的场景：

- **复杂多任务系统**：系统需要同时运行多个任务，如数据采集、网络通信、界面刷新等，任务管理复杂，操作系统的多任务调度和资源管理机制能够简化开发。
- **需要实时性但不极端**的应用：如工业控制、机器人、物联网设备，RTOS 提供的实时调度机制能满足大部分实时性要求，同时简化了多任务编程。
- **网络、文件系统或驱动管理**：如果系统需要处理网络协议、文件系统、外部存储设备等复杂功能，操作系统提供的网络栈、文件系统和驱动管理可以大大减轻开发者的负担。
- **硬件资源充足**：如果系统拥有足够的 RAM 和 Flash（如 100KB 以上的 RAM），操作系统的开销可以接受，且系统功能复杂，操作系统能提供更高效的开发和维护。

- **长时间运行且维护复杂**的系统：操作系统可以提供系统监控、任务重启等机制，保证长时间稳定运行，如嵌入式服务器、智能家居设备、医疗设备等。

操作系统开发的典型例子：

- **物联网设备**：如带有传感器、Wi-Fi 模块的智能设备，使用 RTOS 进行网络管理、任务调度等。
- **工业控制系统**：多个任务需要协调执行，如控制、数据采集和分析等，使用 RTOS 提供的多任务调度和实时响应能力。
- **机器人控制系统**：多任务并发执行，如传感器处理、路径规划、运动控制，操作系统的任务调度简化了系统的管理。

3. 如何选择：裸机开发 vs 操作系统开发

在选择裸机开发或操作系统开发时，主要考虑以下几个因素：

1. 系统复杂度

- **简单系统**：如果系统只是执行一个单一的任务，或是功能非常简单，那么选择裸机开发可以减少不必要的系统开销和复杂度。
- **复杂系统**：如果系统需要执行多个任务，如处理外部事件、数据采集、界面显示和通信等，并且需要更好的任务管理和简化调度，操作系统是更好的选择。

2. 实时性要求

- **极端实时性**：如果系统要求精确的中断响应和微秒级的控制（如电机控制、信号采集），裸机开发可以提供更好的实时性。
- **一般实时性**：如果系统的实时性要求相对宽松，但需要执行多个任务，RTOS 提供的实时调度机制能满足大多数需求。

3. 硬件资源

- **资源有限的硬件**：如果系统资源（RAM 和 Flash）非常有限，操作系统的开销可能过大，此时选择裸机开发更合适。
- **资源充足的硬件**：如果系统有足够的资源运行操作系统，且项目功能复杂，操作系统可以大幅简化开发。

4. 开发周期与维护

- **开发周期短、单一任务**：对于功能明确、实现简单的项目，裸机开发可能更快捷，因为不需要学习和配置操作系统。
- **长期维护、复杂任务**：对于复杂项目，操作系统提供的任务调度、资源管理和错误处理机制能减少后续维护的复杂性，并且更容易扩展和调试。

> **总结**
>
> - **裸机开发**适合资源有限、单一功能、极端实时性要求的系统，优点是响应时间短，资源占用低，缺点是开发难度较高，尤其是多任务管理时。
> - **操作系统开发**适合功能复杂、多任务并发、资源充足的系统，优点是简化了多任务管理和硬件抽象，便于扩展和维护，缺点是增加了系统开销，并且对实时性略有影响。

内核基础

本章将简化介绍 RT-Thread 内核的基础知识，包括内核的组成、启动流程及配置方法。这些内容帮助初学者快速理解 RT-Thread 内核的工作原理。

什么是 RT-Thread 内核

RT-Thread 内核是操作系统的“心脏”，负责管理系统中的各个线程（相当于我们日常生活中的“任务”），并确保它们可以有条不紊地完成。内核还负责处理线程之间的通信、时钟管理、中断响应和内存管理等复杂的系统功能。

可以把内核想象成一位高效的“经理”，它协调公司里的各个部门（线程），确保每个部门按时完成任务，并合理分配资源。

线程调度

在 RT-Thread 中，“线程”是最小的调度单位。线程就像你一天中的不同任务，比如早上起床、上班、吃饭，每个任务（线程）都有其优先级。RT-Thread 使用的是基于优先级的全抢占式调度算法，意思是优先处理重要的任务，而不是所有任务按顺序进行。比如，如果正在吃饭时，手机响了（中断），系统会暂时停止吃饭的任务，优先处理电话（更高优先级的任务）。

RT-Thread 支持多达 256 个优先级（当然可以通过配置改成 32 或 8 个），0 是最高优先级，而最不重要的任务，比如空闲任务，就会分配到最低优先级。

此外，同一个优先级下的任务使用“时间片轮转”的方式。简单来说，系统给每个任务分配相同的时间片，大家轮流执行，不会有人独占系统资源。

时钟管理

RT-Thread 的时钟管理基于“时钟节拍”（tick），类似于日常生活中的秒表，每个节拍是系统中最小的时间单位。系统提供两类定时器：

1. **单次触发定时器**：只触发一次，然后停止。比如你设定闹钟，只响一次。
2. **周期触发定时器**：不断触发，直到你手动停止它。就像洗衣机的定时模式，洗完一遍后可以自动重复。

你还可以根据定时任务的要求选择硬定时器或软定时器，来决定任务的处理方式是否足够实时。

线程间同步

在多线程系统中，多个线程之间需要“协调”，就像你和同事们一起合作工作。RT-Thread 提供了信号量、互斥量和事件集来帮助线程间进行同步。

- **信号量**（Semaphore）：就像车站发车信号，只有收到信号的线程才能开始执行任务。
- **互斥量**（Mutex）：确保多个线程不会同时访问同一资源，避免混乱。比如，打印机只能同时被一个人使用，互斥量相当于一把钥匙，谁拿到钥匙谁就可以用打印机。
- **事件集**：允许线程在多个事件触发时进行操作。就像你等着多个快递到达，可以选择全部到齐再处理，或者一个到达就处理一个。

线程间通信

线程之间需要交换信息，RT-Thread 提供了**邮箱**和**消息队列**来实现。邮箱就像是一个固定容量的收件箱，而消息队列是一个可伸缩的队列，消息可以有不同的长度。邮箱更高效，但消息队列更加灵活。

内存管理

RT-Thread 提供了两种内存管理方式：静态内存和动态内存。

- **静态内存**：提前分配好一块固定大小的内存区域。就像你提前买好一间固定大小的仓库，存放东西很方便，但空间是固定的。
- **动态内存**：根据需要申请和释放内存。像是随时租用仓库，用完就可以退租，但每次申请可能花费一些时间。

有时候，系统中的内存是分散的，这时可以通过 **memheap** 来把多个内存块“拼接”起来，像拼图一样组合成一个整体。

I/O 设备管理

RT-Thread 统一管理各类外设设备（比如 PIN、I2C、SPI、USB、UART），你可以通过设备名称方便地访问硬件设备。这样就像给每个设备都贴上了标签，使用起来非常方便。

RT-Thread 启动流程

RT-Thread 的启动流程相当于系统开机的步骤。系统首先执行启动文件的代码，完成基本的硬件初始化。然后调用 `rtthread_startup()` 函数，执行系统的初始化工作，最后进入 `main()` 函数，正式开始执行用户代码。

你可以把这个过程想象成开车的步骤：启动引擎（初始化硬件），检查车况（初始化系统），最后开始驾驶（执行用户程序）。

线程管理

我们可以把多线程系统想象成一个繁忙的工厂，里面有很多工人（线程），每个工人负责特定的任务。这个工厂里有一个非常聪明的调度员（调度器），它负责分配工作，确保最重要的任务优先完成。工人们在完成各自的工作时，会用工厂的资源（CPU），但不是每个人都能一直占用资源，有时候得让出资源，轮到其他工人来使用。

线程的概念

在 RT-Thread 中，**线程**就是这些工人，它是最小的工作单元。每个线程负责一个任务，例如读取传感器、更新显示屏等。你可以将一个大的任务拆分成多个线程，各自负责不同的部分，这样整个任务能更快地完成。RT-Thread 会根据线程的优先级来决定哪个任务最先完成。

线程的组成部分

每个线程都有几个重要的组成部分，就像工人有自己的工具包：

1. **线程控制块**：它类似工人的身份证，记录了工人的名字、任务、优先级和工作状态等信息。
2. **线程栈**：它是工人用来保存临时数据的地方，当线程执行任务时，临时数据都保存在栈中。
3. **入口函数**：线程的任务内容，告诉线程具体要做什么。比如说，一个线程的任务是读取传感器数据，另一个线程的任务是把数据传到屏幕上显示。

线程的状态

线程有五种状态，就像工人有不同的工作状态一样：

1. **初始状态**：线程刚刚创建，还没开始工作。
2. **就绪状态**：线程准备好了，随时可以开始工作，但目前还没有拿到资源。
3. **运行状态**：线程正在执行任务，正在使用 CPU。
4. **挂起状态**：线程暂时停止工作，可能是因为在等资源，比如等数据到来或者等待定时器。
5. **关闭状态**：线程的任务完成了，线程结束，不再使用。

线程优先级

在工厂里，工人有高低之分，紧急任务由高优先级的工人来处理。RT-Thread 支持 256 个优先级（0-255），数字越小，优先级越高。例如，负责读取传感器数据的线程可能优先级高，而负责输出调试信息的线程优先级低。

时间片

如果多个工人的优先级相同，那么工厂会给每个工人分配固定的工作时间，这就叫做“时间片”。每个线程轮流执行一段时间，这样大家都有机会工作。例如线程 A 和线程 B 优先级相同，但 A 的时间片是 10，B 的时间片是 5，那么 A 会工作 10 个时间单位，B 只工作 5 个时间单位，接着再轮到 A。

线程的创建和启动

创建一个线程就像招聘一个工人，你需要指定工人的名字、任务（入口函数）和工具（栈）。以下是如何创建和启动一个线程的例子：

```c
/* 创建线程示例 */
static rt_thread_t tid1 = RT_NULL;

static void thread1_entry(void* parameter)
{
    int count = 0;
    while (1)
    {
        rt_kprintf("Thread 1 is running, count: %d\n", count);
        count++;
        rt_thread_mdelay(1000);  // 休息 1 秒
    }
}

int main(void)
{
    /* 创建线程 1 */
    tid1 = rt_thread_create("thread1",
                            thread1_entry, RT_NULL,
                            1024, 20, 10);

    /* 启动线程 */
    if (tid1 != RT_NULL)
    {
        rt_thread_startup(tid1);
    }
    return 0;
}
```

在这个例子中，创建了一个名为 `thread1` 的线程，它每秒输出一次计数值，并且执行 5 次后结束。你可以把这个线程看作是一个工人，负责执行简单的打印任务。

线程调度

线程的调度就像工厂的调度员一样，负责决定哪个工人先工作。调度员会选择优先级最高的工人工作，直到他完成任务或时间片用完，然后切换到下一个工人。

空闲线程

空闲线程是优先级最低的线程，就像工厂里没什么紧急任务时，调度员让一些工人去做杂活（比如清理现场）。在 RT-Thread 中，空闲线程负责回收已完成任务的线程资源。

`rt_thread_create` 函数原型

```
rt_thread_t rt_thread_create(const char *name,
                             void (*entry)(void *parameter),
                             void *parameter,
                             rt_uint32_t stack_size,
                             rt_uint8_t priority,
                             rt_uint32_t tick);
```

比喻：餐厅里的厨师

想象你在一家餐厅管理厨师，每个厨师负责做不同的菜品。现在你需要招募和管理多个厨师（线程），让他们在厨房（系统中）各自独立地完成任务。每个厨师在上岗时都需要指定名字、工作内容（做什么菜）、工具和材料（参数）、负责任务的规模（所需的工作台空间）、优先级（做菜的顺序）以及工作时间分配（时间片）。这就像是在一个操作系统中管理线程的创建和运行。

参数详细说明及通俗比喻

1. `name` （厨师名字）
   - **解释**：每个厨师（线程）都有一个独特的名字，可以用来标识和管理该厨师。这就像给每个厨师发一张名牌，方便在厨房中知道谁在做什么。
   - **通俗例子**：名字 `"Chef1"` 、 `"Chef2"` 、 `"Chef3"` 等。
   - **代码示例**： `"led_ctrl"` ，表示该线程用于控制 LED 灯。
2. `entry` （厨师的工作内容）
   - **解释**：表示每个厨师在厨房中具体要做什么（线程的主任务）。每个厨师都有自己擅长的菜品和任务，所以你要告诉厨师他们的工作内容。这就是线程的入口函数（每个线程的具体执行逻辑）。

- **通俗例子**：比如，`Chef1` 的工作内容是炒菜、`Chef2` 的工作内容是烤肉。
- **代码示例**：定义一个线程入口函数，如 `void led_thread_entry(void *parameter)`，表示这个线程要做的任务是“闪烁 LED 灯”。

3. `parameter`（厨师的配料）

- **解释**：给厨师的配料或工作材料（传递给线程的参数）。有些厨师需要不同的配料（不同类型的参数）来完成任务，比如炒菜时给盐或酱油等。
- **通俗例子**：`Chef1` 需要辣椒、`Chef2` 需要酱油，而 `Chef3` 只需要盐和蒜。这些就是不同的参数。
- **代码示例**：`RT_NULL` 表示不传递任何配料（参数）。例如：`led_thread_entry` 中不需要额外的参数时，就可以设为 `RT_NULL`。

4. `stack_size`（厨师的工作台大小）

- **解释**：每个厨师需要不同大小的工作台来放材料和工具（线程栈大小）。如果工作台太小，厨师无法放下所有工具和材料，工作会受到限制甚至失败。
- **通俗例子**：做大菜的厨师需要更大的工作台，而只需要切点水果的厨师可能只需要一个小桌子就够了。
- **确定方法**：如果线程任务简单（比如控制 LED），可以设置较小的栈空间（如 512 字节）；如果线程需要处理复杂数据（如图像处理），则需要较大的栈空间（如 1024 字节）。
- **代码示例**：`512` 表示该线程分配了 512 字节的栈空间，用于存储局部变量和函数调用等数据。

5. `priority`（厨师的做菜顺序）

- **解释**：指定每个厨师（线程）的做菜顺序（优先级）。在餐厅中，主厨（优先级高的线程）要优先准备高级菜品，普通厨师（优先级低的线程）则可以稍后再做。
- **通俗例子**：`Chef1` 优先级最高，专门负责烤牛排；`Chef2` 做普通菜，优先级稍低；`Chef3` 负责清理厨房，优先级最低。
- **确定方法**：优先级范围从 0（最高）到 `RT_THREAD_PRIORITY_MAX-1`（最低）。紧急任务（如数据采集）可以设为 5，普通任务（如 LED 控制）可以设为 25。
- **代码示例**：`25` 表示该线程优先级较低，用于非紧急的周期性任务。

6. `tick`（厨师每次做菜的时间片）

- **解释**：表示同一优先级的多个厨师轮流使用厨房的时间长度。时间片越长，厨师可以连续做菜的时间越多（一次性执行时间越长）。
- **通俗例子**：`Chef1` 每次可以在厨房待 10 分钟，`Chef2` 每次可以待 5 分钟。
- **确定方法**：对于需要频繁切换的线程（如信号采集），可以设置较小的时间片；对于稳定执行的线程（如数据处理），可设定较大的时间片。

- **代码示例**： `5` 表示线程每次获得 5 个时钟周期的执行时间。

代码实例与解析

下面是一个用于 LED 控制的线程创建实例，并逐一讲解参数如何应用。

```
#include <rtthread.h>   // 引入 RT-Thread 头文件

#define LED_STACK_SIZE    512           // 栈大小
#define LED_PRIORITY      25            // 线程优先级
#define LED_TICK          5             // 时间片大小

// 1. 定义 LED 控制线程的入口函数
void led_thread_entry(void *parameter)
{
    rt_kprintf("LED Thread Running...\n");  // 输出线程运行提示
    while (1)
    {
        /* 控制 LED 的点亮与熄灭 */
        rt_pin_write(LED_PIN, PIN_HIGH); // 点亮 LED
        rt_thread_mdelay(500);           // 延时 500 ms
        rt_pin_write(LED_PIN, PIN_LOW);  // 熄灭 LED
        rt_thread_mdelay(500);           // 再延时 500 ms
    }
}

// 2. 在主函数中创建并启动线程
int main(void)
{
    // 使用 rt_thread_create 创建 LED 控制线程
    rt_thread_t led_thread = rt_thread_create("led_ctrl",     // 线程名，便
于调试时查看
                                             led_thread_entry, // 线程入
口函数
                                             RT_NULL,          // 线程入
口参数，不需要参数时设为 RT_NULL
                                             LED_STACK_SIZE,   // 栈大小
512 字节
                                             LED_PRIORITY,     // 优先级
25
                                             LED_TICK);        // 时间片
大小 5

    // 如果线程创建成功，则启动该线程
    if (led_thread != RT_NULL)
```

```
        {
            rt_thread_startup(led_thread);    // 启动线程
        }
        return 0;
    }
```

在 RT-Thread 操作系统中， `rt_thread_mdelay` 和普通的 `delay` 函数（如常见的 `delay_ms` ）有本质上的区别，主要体现在系统调度、实时性和功耗管理等方面。下面我将从工作原理、使用场景、优缺点等多个方面进行详细解释，并以实际案例帮助理解。

1. 工作原理上的区别

- `rt_thread_mdelay` ：
  - `rt_thread_mdelay` 是 RT-Thread 中提供的一个基于系统节拍（tick）实现的延时函数。它是线程级别的延时函数，会让调用该函数的线程进入**休眠状态**，并释放 CPU 资源给其他线程继续运行。在延时结束后，系统会根据线程调度规则恢复该线程的运行。
  - 工作机制：
    1. 当 `rt_thread_mdelay` 被调用时，当前线程会进入 `RT_THREAD_SUSPEND` 状态（挂起状态）。
    2. 调度器开始运行其他优先级相同或更高的线程。
    3. 延时时间结束后，线程从 `挂起状态` 转变为 `就绪状态` ，等待系统再次调度该线程。
- **普通 `delay` （如 `delay_ms` ）**：
  - 普通的 `delay` 函数一般是基于**忙等待（busy-waiting）**机制实现的，即通过不断循环来耗费时间，从而达到延时的效果。普通延时函数在延时期间不会释放 CPU 资源，因此 CPU 处于**忙碌状态**，无法执行其他任务。
  - 工作机制：
    1. CPU 执行一个空循环来持续占用时间。
    2. 在整个延时过程中，CPU 资源不会被释放。
    3. 只有当延时结束时，才能继续执行下一个任务。

2. 使用场景上的区别

- **`rt_thread_mdelay` 适用场景**：
  - 适用于需要在多线程环境中释放 CPU 资源的延时操作。
  - 当需要在延时过程中让出 CPU 资源给其他线程使用时，如：多线程间的通信、任务调度等场景。
  - 适合实时操作系统环境，在多任务环境下能有效管理 CPU 资源。
- **普通 `delay` 适用场景**：
  - 适用于单线程、裸机环境或不涉及多任务调度的简单延时需求。
  - 当延时过程中不希望其他任务打断（如一些简单的硬件初始化延时）时。
  - 适用于对 CPU 资源利用率和功耗管理要求不高的场合。

3. 优缺点对比

区别点	`rt_thread_mdelay`	普通 `delay`
是否释放 CPU	是，延时过程中释放 CPU 资源	否，延时期间 CPU 一直被占用
系统调度	支持多线程调度和任务切换	无法调度其他任务（除非使用中断机制）
功耗管理	可以进入低功耗模式，节约能量	无法进入低功耗模式，CPU 一直忙等待
实时性	适合实时操作系统，延时更精确	仅适合简单场合，实时性较差
复杂度	需要在多线程环境中使用	简单、易用

4. 通俗比喻

假设你在一个咖啡店排队点单：

- **普通 `delay` 就像你在柜台前排队时干等着，什么都不做**。在排队的这段时间内，你不能去做其他事情（比如不能去看看菜单或者去买其他东西），只能原地等待。这段时间对于你来说是“浪费的”时间。
- **`rt_thread_mdelay` 就像你在排队时先去做别的事情，比如去取咖啡杯或看看店里的装饰**。你没有白白等待，而是去做了别的事情。等到时间到了，你再回来继续排队。这样就不会浪费时间，系统（CPU）可以去处理其他任务（线程）。

5. 代码实例对比

以下是两个代码实例，分别展示 `rt_thread_mdelay` 和普通 `delay` 在多线程中的区别。

1. 使用 `rt_thread_mdelay` 的例子

```c
#include <rtthread.h>

// 线程入口函数，用于控制 LED 的闪烁
void led_blink_thread(void *parameter)
{
    while (1)
    {
        rt_kprintf("LED ON\n");    // 模拟 LED 点亮
        rt_thread_mdelay(1000);    // 延时 1000 ms，期间 CPU 资源释放
        rt_kprintf("LED OFF\n");   // 模拟 LED 熄灭
        rt_thread_mdelay(1000);    // 再延时 1000 ms，期间 CPU 资源释放
    }
}

int main(void)
{
    rt_thread_t led_thread = rt_thread_create("led_blink",     // 线程名
                                              led_blink_thread, // 入口函数
                                              RT_NULL,          // 无参数
                                              1024,             // 栈大小
                                              20,               // 优先级
                                              10);              // 时间片
    if (led_thread != RT_NULL)
        rt_thread_startup(led_thread);  // 启动线程

    return 0;
}
```

在上面的例子中：

- `rt_thread_mdelay(1000);` 会让 LED 控制线程进入挂起状态，并释放 CPU 资源给其他线程。
- 系统调度器可以切换到其他优先级相同或更高的线程来执行其他任务，CPU 资源得到了有效利用。

2. 使用普通 `delay` 的例子（基于忙等待）

```c
#include <rtthread.h>

void led_blink_thread(void *parameter)
{
    while (1)
    {
        rt_kprintf("LED ON\n");    // 模拟 LED 点亮
        delay_ms(1000);            // 延时 1000 ms，期间 CPU 被占用
        rt_kprintf("LED OFF\n");   // 模拟 LED 熄灭
        delay_ms(1000);            // 再延时 1000 ms，期间 CPU 被占用
    }
}

int main(void)
{
    rt_thread_t led_thread = rt_thread_create("led_blink",      // 线程名
                                              led_blink_thread, // 入口函
数
                                              RT_NULL,          // 无参数
                                              1024,             // 栈大小
                                              20,               // 优先级
                                              10);              // 时间片
    if (led_thread != RT_NULL)
        rt_thread_startup(led_thread);  // 启动线程

    return 0;
}
```

在这个例子中：

- `delay_ms(1000);` 使用了忙等待机制。该线程在延时期间会持续占用 CPU 资源，导致 CPU 无法调度其他线程执行。
- 整个延时过程没有释放 CPU，因此多线程系统中的 CPU 利用效率较低。

6. 总结

- `rt_thread_mdelay` 更适合在多线程实时操作系统中使用，因为它能够释放 CPU 资源，方便调度其他任务，提高 CPU 利用率和系统响应速度。
- 普通 `delay` 仅适用于简单的单任务系统，虽然实现容易，但在复杂系统中可能引起 CPU 资源浪费和实时性下降。

示例：创建和切换线程

我们可以创建多个线程，让它们在同一个系统中轮流执行。以下是一个简单的示例，创建两个线程，它们会交替打印各自的计数值：

```
#include <rtthread.h>

#define THREAD_PRIORITY 25
#define THREAD_STACK_SIZE 1024
#define THREAD_TIMESLICE 5

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    int count = 0;
    while (count < 10)
    {
        rt_kprintf("Thread 1 is running, count: %d\n", count);
        count++;
        rt_thread_mdelay(1000);  // 每秒打印一次
    }
}

/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    int count = 0;
    while (count < 10)
    {
        rt_kprintf("Thread 2 is running, count: %d\n", count);
        count++;
        rt_thread_mdelay(1000);  // 每秒打印一次
    }
}

int main(void)
{
    /* 创建线程 1 */
    tid1 = rt_thread_create("thread1",
                            thread1_entry, RT_NULL,
                            THREAD_STACK_SIZE, THREAD_PRIORITY,
THREAD_TIMESLICE);
```

```
    /* 创建线程 2 */
    tid2 = rt_thread_create("thread2",
                            thread2_entry, RT_NULL,
                            THREAD_STACK_SIZE, THREAD_PRIORITY,
THREAD_TIMESLICE);

    /* 启动线程 */
    if (tid1 != RT_NULL) rt_thread_startup(tid1);
    if (tid2 != RT_NULL) rt_thread_startup(tid2);

    return 0;
}
```

在这个例子中，两个线程 `thread1` 和 `thread2` 轮流执行，各自输出计数。由于它们的优先级和时间片相同，调度器会公平地让它们轮流工作。

时钟管理

你可以将时钟管理比作家中的闹钟，它定时发出声音提醒我们该起床了。操作系统也有类似的“闹钟”，它会在一定的时间间隔内“响一次”，提醒系统完成某些任务。在 RT-Thread 中，**时钟节拍 (OS Tick)** 就像闹钟的每一次响声，它是系统处理时间相关事件（如延时、任务切换）的基础。

什么是时钟节拍?

操作系统中的**时钟节拍**就像闹钟每响一次，代表时间过去了一小段。时钟节拍的长短可以调整，通常在 1 毫秒到 100 毫秒之间。在 RT-Thread 中，时钟节拍的时间间隔由 `RT_TICK_PER_SECOND` 来决定。例如，如果 `RT_TICK_PER_SECOND` 定义为 100，那每 1 秒系统会响 100 次时钟节拍。

时钟节拍的产生

时钟节拍依赖于硬件中的定时器。你可以把定时器看作是一个不停计时的装置，它每隔一段时间就会提醒系统“滴答”一次。每当定时器中断发生时，RT-Thread 会调用 `rt_tick_increase()` 函数，系统时间增加 1 个节拍。就像家中的闹钟响一次，时间就过去了一分钟一样。

举个简单的例子，STM32 处理器中时钟节拍的实现可以用 `SysTick_Handler()` 函数来处理，每次定时器中断触发时都会执行：

```
void SysTick_Handler(void)
{
    rt_interrupt_enter();  // 进入中断
    rt_tick_increase();    // 增加系统时钟
    rt_interrupt_leave();  // 退出中断
}
```

当 `rt_tick_increase()` 被调用时，它让系统的全局变量 `rt_tick` 自加1，表示系统已经经历了一个节拍。你可以把 `rt_tick` 理解为“时钟总计数器”，它记录了系统从启动以来经历的总时间。

如何获取当前时钟节拍?

就像我们时不时看一下闹钟，知道现在几点了，我们可以通过 `rt_tick_get()` 来获得当前的时钟节拍数。

```
rt_tick_t rt_tick_get(void);
```

返回值是当前的时钟节拍数，你可以用它来记录某个任务执行了多长时间。比如你可以在任务开始时获取一次 `rt_tick` 值，在任务结束时再获取一次，二者相减就是任务花费的时间。

定时器管理

你可以将定时器想象成你家中的定时器或计时器。比如，你设定一个10分钟的定时器，当时间到了之后，它就响起并提醒你，该去检查烤箱里的食物了。RT-Thread 中的**定时器**就是这样的功能，它可以在指定的时间后触发一个事件（比如执行一个函数）。

硬件定时器 vs 软件定时器

1. **硬件定时器**：就像厨房里的倒计时器，它是由硬件直接提供的。硬件定时器精度非常高，可以达到纳秒级，适合一些对时间要求非常严格的任务，比如控制电机的旋转时间。
2. **软件定时器**：可以理解为手机上的计时 App。它依赖于操作系统的时钟节拍来工作，它的时间间隔精度由 OS Tick 决定。例如，如果 OS Tick 是 10ms，那么软件定时器只能精确到 10ms。虽然精度不如硬件定时器高，但它更灵活、容易使用。

RT-Thread 定时器的分类

RT-Thread 提供两种类型的定时器：

- **单次触发定时器**：就像倒计时一次的闹钟，响一次后就停止。例如，设定一个 5 秒的倒计时，时间到了闹钟响一次就结束了。

- **周期触发定时器**：就像那些每天早上 7 点重复响起的闹钟。每隔一段时间，它会周期性地触发事件，直到你手动关闭它。

根据定时器回调函数执行的上下文环境，定时器还可以分为两种模式：

- **硬件模式 (HARD_TIMER)**：定时器的回调函数在中断上下文中执行，就像闹钟响起时你立刻跳起来一样。它反应非常快，但要求处理时间短。
- **软件模式 (SOFT_TIMER)**：回调函数在系统的 `timer` 线程中执行，相当于有人提醒你“闹钟响了”，然后你再去执行一些动作。这个模式可以做更复杂的操作，因为它不是在中断中执行。

如何创建和使用定时器?

就像你给家里的闹钟设定时间一样，RT-Thread 提供了创建定时器的函数：

```
rt_timer_t rt_timer_create(const char* name,
                           void (*timeout)(void* parameter),
                           void* parameter,
                           rt_tick_t time,
                           rt_uint8_t flag);
```

- `name` 是定时器的名字，相当于你给闹钟取了个名字。
- `timeout` 是当定时器时间到了之后要执行的函数，就像闹钟响了你要做的事情。
- `time` 是定时时间长度，单位是 OS Tick（相当于设定闹钟的倒计时）。
- `flag` 表示定时器类型，比如单次触发还是周期触发。

使用示例

以下是创建两个定时器的例子，一个是单次定时器，另一个是周期性定时器：

```
#include <rtthread.h>

static rt_timer_t timer1;  // 定义定时器 1
static rt_timer_t timer2;  // 定义定时器 2

/* 定时器 1 的超时回调函数 */
static void timeout1(void *parameter)
{
    rt_kprintf("Periodic timer timeout\n");
}

/* 定时器 2 的超时回调函数 */
static void timeout2(void *parameter)
```

```
{
    rt_kprintf("One-shot timer timeout\n");
}


int main(void)
{
    /* 创建周期性定时器 */
    timer1 = rt_timer_create("timer1", timeout1,
                             RT_NULL, 10,
                             RT_TIMER_FLAG_PERIODIC);

    /* 启动定时器 1 */
    if (timer1 != RT_NULL) rt_timer_start(timer1);

    /* 创建单次定时器 */
    timer2 = rt_timer_create("timer2", timeout2,
                             RT_NULL, 30,
                             RT_TIMER_FLAG_ONE_SHOT);

    /* 启动定时器 2 */
    if (timer2 != RT_NULL) rt_timer_start(timer2);

    return 0;
}
```

在这个例子中，**周期性定时器** `timer1` 每 10 个 OS Tick 会触发一次超时函数并输出 "周期性定时器超时"；而 **单次定时器** `timer2` 在 30 个 OS Tick 后会触发超时函数，并输出 "单次定时器超时"。

控制定时器

你还可以通过 `rt_timer_control()` 来控制定时器，比如改变定时器的时间、设定为单次或周期触发：

```
rt_err_t rt_timer_control(rt_timer_t timer, rt_uint8_t cmd, void* arg);
```

- `RT_TIMER_CTRL_SET_TIME`：改变定时器的超时时间。
- `RT_TIMER_CTRL_SET_ONESHOT`：设置定时器为单次定时器。
- `RT_TIMER_CTRL_SET_PERIODIC`：设置定时器为周期性定时器。

高精度延时

有时候你需要的延时比 OS Tick 更加精确，例如在某些场景下你可能需要 1 毫秒甚至更短的延时。这时可以通过硬件定时器或者芯片的功能来实现。在一些 ARM Cortex-M 系统中，可以通过 `SysTick` 实现微秒级的延时：

```
void rt_hw_us_delay(rt_uint32_t us)
{
    rt_uint32_t ticks = us * (SysTick->LOAD / 1000000);
    rt_uint32_t start = SysTick->VAL;
    while ((SysTick->VAL - start) < ticks);
}
```

这个函数可以实现微秒级的精确延时，适合那些需要高精度时间控制的场合。

线程间同步

在多线程实时系统中，一项工作的完成往往需要多个线程协调工作。如果多个线程同时访问共享资源，可能会导致数据错乱。为了保证数据一致性，线程需要通过同步机制来协作。以下介绍几种常见的同步机制：信号量、互斥量和事件集。

信号量（Semaphore）

信号量可以理解为电影院的座位数（资源数量）。当所有座位都满了，新的观众（线程）就需要等前面的观众离开（释放资源）才能进入。当有空位时，观众可以进入。信号量通过控制资源的数量来管理多个线程对资源的访问。

信号量的工作机制

信号量的核心思想是控制资源的访问数量。信号量的值代表了可以访问的资源数量：

- **获取信号量**：当资源可用时，线程可以获取信号量，信号量的值减1。如果信号量的值为0，线程就会等待，直到资源释放。
- **释放信号量**：使用完资源后，线程释放信号量，信号量的值加1，允许其他线程访问资源。

信号量应用示例

```
#include <rtthread.h>

/* 创建信号量 */
rt_sem_t sem = RT_NULL;

/* 线程1：等待进入电影院 */
```

```c
void thread_entry1(void *parameter)
{
    rt_kprintf("线程1: 等待进入电影院...\n");
    rt_sem_take(sem, RT_WAITING_FOREVER); // 尝试获取信号量
    rt_kprintf("线程1: 进入电影院! \n");
}

/* 线程2: 离开电影院，释放信号量 */
void thread_entry2(void *parameter)
{
    rt_kprintf("线程2: 离开电影院，释放一个座位\n");
    rt_sem_release(sem); // 释放信号量
}

int main(void)
{
    /* 初始化信号量，初始值为1，表示电影院有1个座位 */
    sem = rt_sem_create("sem", 1, RT_IPC_FLAG_PRIO);

    /* 创建两个线程 */
    rt_thread_t tid1 = rt_thread_create("t1", thread_entry1, RT_NULL, 1024, 25,
10);
    rt_thread_t tid2 = rt_thread_create("t2", thread_entry2, RT_NULL, 1024, 25,
10);

    /* 启动线程 */
    rt_thread_startup(tid1);
    rt_thread_startup(tid2);

    return 0;
}
```

解释:

- `thread_entry1` 线程等待进入电影院，它需要等到有空座位（信号量大于0）。
- `thread_entry2` 线程模拟一个观众离开电影院，释放一个座位，其他等待的线程可以进入。

> 在 RT-Thread 中，信号量的创建 API 提供了一种机制来进行线程之间的同步和互斥。信号量 API 主要用于创建、获取（P 操作）、释放（V 操作）、删除等操作。通过前面提供的三个历程（任务同步、任务互斥、事件触发）中的信号量使用，我们可以总结 RT-Thread 中信号量创建的相关 API。

1. rt_sem_create

用于动态创建信号量。

```
rt_sem_t rt_sem_create(const char *name, rt_uint32_t value, rt_uint8_t
flag);
```

参数说明:

- `name` ： 信号量的名字（可以为 `RT_NULL` 表示匿名信号量）。
- `value` ： 信号量的初始值，表示信号量当前持有的资源数量（例如初始值为 0 表示等待事件，为 1 表示互斥量，或者更大值表示资源计数）。
- `flag` ： IPC 对象的属性标志，常用值:
  - `RT_IPC_FLAG_PRIO` ： 优先级等待方式，等待线程按照优先级顺序排列。
  - `RT_IPC_FLAG_FIFO` ： 先入先出等待方式，等待线程按照进入顺序排列。

返回值:

- 成功：返回信号量的句柄 `rt_sem_t` 。
- 失败：返回 `RT_NULL` 。

示例:

```
/* 创建一个初始值为 0 的信号量，用于同步场景 */
rt_sem_t sync_sem = rt_sem_create("sync_sem", 0, RT_IPC_FLAG_PRIO);
```

2. rt_sem_init

用于静态信号量的初始化，一般用于静态分配内存的场景。

```
rt_err_t rt_sem_init(rt_sem_t sem, const char *name, rt_uint32_t value,
rt_uint8_t flag);
```

参数说明:

- `sem` ： 信号量对象的指针（需要预先定义好该信号量结构体）。
- `name` ： 信号量的名字。
- `value` ： 信号量的初始值。

- `flag` ：IPC 对象的属性标志（同 `rt_sem_create` ）。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回相应的错误代码。

示例：

```
/* 静态信号量定义 */
static struct rt_semaphore static_sem;

/* 静态信号量初始化，初始值为1，用于互斥操作 */
rt_sem_init(&static_sem, "static_sem", 1, RT_IPC_FLAG_PRIO);
```

3. rt_sem_take

用于获取信号量（P 操作），线程会尝试获取信号量，如果信号量的计数值为 0，线程会进入等待状态，直到信号量被释放或超时。

```
rt_err_t rt_sem_take(rt_sem_t sem, rt_int32_t time);
```

参数说明：

- `sem` ：信号量的句柄。
- `time` ：超时时间（单位为系统 tick），表示最大等待时间。如果设置为 `RT_WAITING_FOREVER` ，线程会一直等待。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回 `-RT_ETIMEOUT` （超时）或其他错误代码。

示例：

```
/* 线程1等待信号量，直到收到信号 */
rt_sem_take(sync_sem, RT_WAITING_FOREVER);
```

4. rt_sem_release

用于释放信号量（V 操作），增加信号量的计数值，并唤醒等待该信号量的线程。

```
rt_err_t rt_sem_release(rt_sem_t sem);
```

参数说明：

- `sem` ：信号量的句柄。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回相应的错误代码。

示例：

```
/* 线程2完成任务后释放信号量，通知线程1 */
rt_sem_release(sync_sem);
```

5. rt_sem_delete

用于删除动态信号量（仅限于 `rt_sem_create` 创建的动态信号量）。

```
rt_err_t rt_sem_delete(rt_sem_t sem);
```

参数说明：

- `sem` ：信号量的句柄。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回相应的错误代码。

示例：

```
/* 删除动态创建的信号量 */
rt_sem_delete(sync_sem);
```

在 RT-Thread 中，信号量（semaphore）是一种常见的同步机制，主要用于任务之间的同步与互斥。它可以通过信号量的计数机制来协调多个线程的运行，确保资源的安全访问。信号量的开发场景通常有以下几种：

1. 任务同步

多个任务之间需要在某些时间点进行同步时，可以使用信号量。信号量可以用来实现线程间的等待和通知机制。例如，当一个任务完成某项工作后，发送一个信号量，通知等待该信号量的任务继续执行。

应用场景示例：

- 一个任务处理传感器数据，处理完成后通知另一个任务进行数据发送。
- 一个任务等待外部事件（如外部中断）发生，中断发生后释放信号量，任务获取信号量后继续执行。

2. 任务互斥

当多个任务需要访问共享资源时，可以使用信号量来实现互斥访问。互斥信号量（Mutex）可以保证同一时刻只有一个任务能够访问共享资源，避免出现竞争条件和数据不一致的情况。

应用场景示例：

- 多个任务同时访问同一个硬件外设（如串口、I2C 总线）时，通过信号量保证同一时刻只有一个任务访问外设，防止访问冲突。
- 多个任务需要访问同一块内存数据时，使用信号量确保数据访问的独占性。

3. 事件触发

某些外部事件发生后，需要通知任务执行相应的处理逻辑。这种场景下，外部中断或事件处理函数可以通过释放信号量来通知任务进行处理。任务会等待信号量，当信号量被释放时，任务被唤醒并开始执行。

应用场景示例：

- 一个任务等待按键事件，当按键被按下时，中断处理程序释放信号量，任务获取信号量并执行按键响应逻辑。
- 网络模块等待数据接收事件，当网络数据到达时，触发信号量，通知任务进行数据处理。

4. 限量资源管理

信号量可以用来管理有限的资源，信号量的计数值可以表示可用资源的数量。当一个任务获取到资源时，信号量的计数值减少；当任务释放资源时，信号量的计数值增加。如果信号量的计数值为 0，则任务必须等待资源可用。

应用场景示例：

- 一个系统有多个任务需要访问一个有限的资源池（如线程池、内存池），信号量可以表示可用资源的数量，确保任务只能在资源可用时进行访问。
- 限制并发访问的数量，比如同一时刻只能有固定数量的任务访问某个服务。

好的，以下是任务同步场景下的示例代码，使用中文注释和调试输出：

1. 任务同步示例

假设有两个线程 `thread1` 和 `thread2`， `thread1` 需要等待 `thread2` 完成某些操作后再继续执行，使用信号量来实现同步。

```
#include <rtthread.h>

#define THREAD_STACK_SIZE 512
#define THREAD_PRIORITY 10
#define THREAD_TIMESLICE 5

static rt_thread_t thread1_handle = RT_NULL;
static rt_thread_t thread2_handle = RT_NULL;
static rt_sem_t sync_sem = RT_NULL;

/* 线程1入口函数 */
static void thread1_entry(void *parameter)
{
    rt_kprintf("线程1：等待线程2发送信号...\n");

    /* 等待信号量，超时时间为永远等待 */
    rt_sem_take(sync_sem, RT_WAITING_FOREVER);

    rt_kprintf("线程1：收到线程2的信号，继续执行...\n");
}
```

```c
/* 线程2入口函数 */
static void thread2_entry(void *parameter)
{
    rt_kprintf("线程2: 执行一些操作...\n");

    /* 模拟耗时操作 */
    rt_thread_mdelay(2000);

    rt_kprintf("线程2: 操作完成，发送信号给线程1...\n");

    /* 释放信号量，通知线程1 */
    rt_sem_release(sync_sem);
}

int main(void)
{
    /* 创建一个信号量，初始值为0 */
    sync_sem = rt_sem_create("sync_sem", 0, RT_IPC_FLAG_PRIO);

    if (sync_sem == RT_NULL)
    {
        rt_kprintf("信号量创建失败! \n");
        return -1;
    }

    /* 创建线程1 */
    thread1_handle = rt_thread_create("thread1",
                                      thread1_entry,
                                      RT_NULL,
                                      THREAD_STACK_SIZE,
                                      THREAD_PRIORITY,
                                      THREAD_TIMESLICE);
    if (thread1_handle != RT_NULL)
        rt_thread_startup(thread1_handle);

    /* 创建线程2 */
    thread2_handle = rt_thread_create("thread2",
                                      thread2_entry,
                                      RT_NULL,
                                      THREAD_STACK_SIZE,
                                      THREAD_PRIORITY,
                                      THREAD_TIMESLICE);
    if (thread2_handle != RT_NULL)
        rt_thread_startup(thread2_handle);
```

```
    return 0;
}
```

2. 任务互斥示例

当两个线程需要互斥访问共享资源时，可以使用信号量来确保同一时刻只有一个线程能够访问资源。

```
#include <rtthread.h>

#define THREAD_STACK_SIZE 512
#define THREAD_PRIORITY 10
#define THREAD_TIMESLICE 5

static rt_thread_t thread1_handle = RT_NULL;
static rt_thread_t thread2_handle = RT_NULL;
static rt_sem_t mutex_sem = RT_NULL;

/* 模拟共享资源 */
static int shared_resource = 0;

/* 线程1入口函数 */
static void thread1_entry(void *parameter)
{
    while (1)
    {
        /* 获取互斥信号量 */
        rt_sem_take(mutex_sem, RT_WAITING_FOREVER);

        rt_kprintf("线程1: 访问共享资源，当前值为 %d\n", shared_resource);
        shared_resource++;
        rt_kprintf("线程1: 修改共享资源，值变为 %d\n", shared_resource);

        /* 模拟资源处理耗时 */
        rt_thread_mdelay(1000);

        /* 释放互斥信号量 */
        rt_sem_release(mutex_sem);

        /* 让出CPU给其他线程 */
        rt_thread_mdelay(500);
    }
}

/* 线程2入口函数 */
```

```c
static void thread2_entry(void *parameter)
{
    while (1)
    {
        /* 获取互斥信号量 */
        rt_sem_take(mutex_sem, RT_WAITING_FOREVER);

        rt_kprintf("线程2: 访问共享资源，当前值为 %d\n", shared_resource);
        shared_resource++;
        rt_kprintf("线程2: 修改共享资源，值变为 %d\n", shared_resource);

        /* 模拟资源处理耗时 */
        rt_thread_mdelay(1000);

        /* 释放互斥信号量 */
        rt_sem_release(mutex_sem);

        /* 让出CPU给其他线程 */
        rt_thread_mdelay(500);
    }
}

int main(void)
{
    /* 创建一个互斥信号量，初始值为1 */
    mutex_sem = rt_sem_create("mutex_sem", 1, RT_IPC_FLAG_PRIO);

    if (mutex_sem == RT_NULL)
    {
        rt_kprintf("互斥信号量创建失败! \n");
        return -1;
    }

    /* 创建线程1 */
    thread1_handle = rt_thread_create("thread1",
                                      thread1_entry,
                                      RT_NULL,
                                      THREAD_STACK_SIZE,
                                      THREAD_PRIORITY,
                                      THREAD_TIMESLICE);
    if (thread1_handle != RT_NULL)
        rt_thread_startup(thread1_handle);

    /* 创建线程2 */
    thread2_handle = rt_thread_create("thread2",
                                      thread2_entry,
```

```
                                        RT_NULL,
                                        THREAD_STACK_SIZE,
                                        THREAD_PRIORITY,
                                        THREAD_TIMESLICE);
    if (thread2_handle != RT_NULL)
        rt_thread_startup(thread2_handle);

    return 0;
}
```

3. 事件触发示例

假设有一个中断或外部事件会触发，线程在等待事件发生时获取信号量。

```
#include <rtthread.h>

#define THREAD_STACK_SIZE 512
#define THREAD_PRIORITY 10
#define THREAD_TIMESLICE 5

static rt_thread_t thread_handle = RT_NULL;
static rt_sem_t event_sem = RT_NULL;

/* 模拟外部事件发生，例如按键中断 */
void external_event_handler(void)
{
    rt_kprintf("外部事件发生！发送信号量...\n");

    /* 释放信号量，通知等待的线程 */
    rt_sem_release(event_sem);
}

/* 线程入口函数 */
static void thread_entry(void *parameter)
{
    while (1)
    {
        rt_kprintf("线程：等待外部事件...\n");

        /* 等待外部事件的信号量 */
        rt_sem_take(event_sem, RT_WAITING_FOREVER);

        rt_kprintf("线程：收到外部事件信号，处理事件...\n");

        /* 模拟事件处理过程 */
```

```
            rt_thread_mdelay(1000);
        }
    }

    int main(void)
    {
        /* 创建一个信号量，初始值为0 */
        event_sem = rt_sem_create("event_sem", 0, RT_IPC_FLAG_PRIO);

        if (event_sem == RT_NULL)
        {
            rt_kprintf("信号量创建失败! \n");
            return -1;
        }

        /* 创建线程 */
        thread_handle = rt_thread_create("thread",
                                         thread_entry,
                                         RT_NULL,
                                         THREAD_STACK_SIZE,
                                         THREAD_PRIORITY,
                                         THREAD_TIMESLICE);
        if (thread_handle != RT_NULL)
            rt_thread_startup(thread_handle);

        /* 模拟外部事件 */
        rt_thread_mdelay(2000);
        external_event_handler();

        return 0;
    }
```

4. 限量资源管理示例

假设有多个线程同时请求访问有限的资源池（如网络连接或内存块），使用信号量来控制访问的数量。

```
    #include <rtthread.h>

    #define THREAD_STACK_SIZE 512
    #define THREAD_PRIORITY 10
    #define THREAD_TIMESLICE 5

    static rt_thread_t thread1_handle = RT_NULL;
    static rt_thread_t thread2_handle = RT_NULL;
```

```c
static rt_sem_t resource_sem = RT_NULL;

/* 线程1入口函数 */
static void thread1_entry(void *parameter)
{
    while (1)
    {
        /* 尝试获取资源 */
        if (rt_sem_take(resource_sem, RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("线程1: 成功获取资源! \n");

            /* 模拟使用资源 */
            rt_thread_mdelay(2000);

            rt_kprintf("线程1: 释放资源! \n");
            rt_sem_release(resource_sem);
        }

        /* 让出CPU给其他线程 */
        rt_thread_mdelay(500);
    }
}

/* 线程2入口函数 */
static void thread2_entry(void *parameter)
{
    while (1)
    {
        /* 尝试获取资源 */
        if (rt_sem_take(resource_sem, RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("线程2: 成功获取资源! \n");

            /* 模拟使用资源 */
            rt_thread_mdelay(2000);

            rt_kprintf("线程2: 释放资源! \n");
            rt_sem_release(resource_sem);
        }

        /* 让出CPU给其他线程 */
        rt_thread_mdelay(500);
    }
}
```

```c
int main(void)
{
    /* 创建一个信号量，初始值为1，表示有一个可用资源 */
    resource_sem = rt_sem_create("resource_sem", 1, RT_IPC_FLAG_PRIO);

    if (resource_sem == RT_NULL)
    {
        rt_kprintf("信号量创建失败! \n");
        return -1;
    }

    /* 创建线程1 */
    thread1_handle = rt_thread_create("thread1",
                                      thread1_entry

,
                                      RT_NULL,
                                      THREAD_STACK_SIZE,
                                      THREAD_PRIORITY,
                                      THREAD_TIMESLICE);
    if (thread1_handle != RT_NULL)
        rt_thread_startup(thread1_handle);

    /* 创建线程2 */
    thread2_handle = rt_thread_create("thread2",
                                      thread2_entry,
                                      RT_NULL,
                                      THREAD_STACK_SIZE,
                                      THREAD_PRIORITY,
                                      THREAD_TIMESLICE);
    if (thread2_handle != RT_NULL)
        rt_thread_startup(thread2_handle);

    return 0;
}
```

这些示例展示了信号量在不同场景下的应用，分别对应任务同步、互斥、事件触发和资源管理。在实际开发中，可以根据具体需求调整这些代码。

互斥量（Mutex）

互斥量类似于家里的洗手间，任何时候只能有一个人使用洗手间（共享资源）。如果有人正在使用，其他人必须等待，直到当前使用者释放洗手间。互斥量可以保证多个线程对共享资源的互斥访问。

互斥量的工作机制

互斥量确保同一时刻只有一个线程能够访问共享资源：

- **获取互斥量**：当一个线程获取到互斥量时，其他线程不能访问该资源，直到互斥量被释放。
- **释放互斥量**：当线程释放互斥量后，其他等待的线程才能访问资源。

互斥量应用示例

```
#include <rtthread.h>

/* 创建互斥量 */
rt_mutex_t mutex = RT_NULL;

/* 线程1：想使用洗手间 */
void thread_entry1(void *parameter)
{
    rt_kprintf("线程1：想使用洗手间...\n");
    rt_mutex_take(mutex, RT_WAITING_FOREVER); // 获取互斥量
    rt_kprintf("线程1：正在使用洗手间\n");
    rt_thread_mdelay(1000); // 模拟使用时间
    rt_mutex_release(mutex); // 释放互斥量
    rt_kprintf("线程1：使用完毕，退出洗手间\n");
}

/* 线程2：想使用洗手间 */
void thread_entry2(void *parameter)
{
    rt_kprintf("线程2：想使用洗手间...\n");
    rt_mutex_take(mutex, RT_WAITING_FOREVER); // 获取互斥量
    rt_kprintf("线程2：正在使用洗手间\n");
    rt_thread_mdelay(1000); // 模拟使用时间
    rt_mutex_release(mutex); // 释放互斥量
    rt_kprintf("线程2：使用完毕，退出洗手间\n");
}

int main(void)
{
    /* 初始化互斥量 */
    mutex = rt_mutex_create("mutex", RT_IPC_FLAG_PRIO);

    /* 创建两个线程 */
    rt_thread_t tid1 = rt_thread_create("t1", thread_entry1, RT_NULL, 1024, 25,
10);
    rt_thread_t tid2 = rt_thread_create("t2", thread_entry2, RT_NULL, 1024, 25,
10);
```

```
    /* 启动线程 */
    rt_thread_startup(tid1);
    rt_thread_startup(tid2);

    return 0;
}
```

解释：

- `thread_entry1` 和 `thread_entry2` 模拟两个线程想使用同一个洗手间。互斥量保证了在同一时刻，只有一个线程能够使用洗手间。

1. rt_mutex_create

用于动态创建一个互斥量。

```
rt_mutex_t rt_mutex_create(const char *name, rt_uint8_t flag);
```

参数说明：

- `name` ：互斥量的名字（可以为 `RT_NULL` 表示匿名互斥量）。
- `flag` ：IPC 对象的属性标志，通常使用 `RT_IPC_FLAG_PRIO` 表示优先级等待。

返回值：

- 成功：返回互斥量的句柄 `rt_mutex_t` 。
- 失败：返回 `RT_NULL` 。

示例：

```
rt_mutex_t mutex = rt_mutex_create("mutex", RT_IPC_FLAG_PRIO);
```

2. rt_mutex_init

用于静态初始化互斥量，适用于需要预先分配互斥量结构体的场景。

```
rt_err_t rt_mutex_init(rt_mutex_t mutex, const char *name, rt_uint8_t
flag);
```

参数说明：

- `mutex` ： 互斥量对象的指针（需预定义）。
- `name` ： 互斥量的名字。
- `flag` ： IPC 对象的属性标志，通常为 `RT_IPC_FLAG_PRIO` 。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回相应的错误代码。

示例：

```
/* 静态互斥量定义 */
static struct rt_mutex static_mutex;

/* 初始化互斥量 */
rt_mutex_init(&static_mutex, "static_mutex", RT_IPC_FLAG_PRIO);
```

3. rt_mutex_take

用于获取互斥量，线程通过调用此函数进入临界区。如果互斥量已经被其他线程获取，当前线程会进入等待状态，直到获取成功或超时。

```
rt_err_t rt_mutex_take(rt_mutex_t mutex, rt_int32_t time);
```

参数说明：

- `mutex` ： 互斥量的句柄。
- `time` ： 超时时间（单位为系统 tick），可以指定等待时长或 `RT_WAITING_FOREVER` 表示永远等待。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回相应的错误代码（如超时返回 `-RT_ETIMEOUT` ）。

示例：

```
rt_mutex_take(mutex, RT_WAITING_FOREVER); // 永久等待获取互斥量
```

4. rt_mutex_release

用于释放互斥量，使其他等待该互斥量的线程能够继续执行。

```
rt_err_t rt_mutex_release(rt_mutex_t mutex);
```

参数说明：

- `mutex`：互斥量的句柄。

返回值：

- 成功：返回 `RT_EOK`。
- 失败：返回相应的错误代码。

示例：

```
rt_mutex_release(mutex); // 释放互斥量
```

5. rt_mutex_detach

用于删除或卸载静态互斥量（互斥量对象由 `rt_mutex_init` 初始化的情况下）。

```
rt_err_t rt_mutex_detach(rt_mutex_t mutex);
```

参数说明：

- `mutex`：互斥量的句柄。

返回值：

- 成功：返回 `RT_EOK`。
- 失败：返回相应的错误代码。

示例：

```
rt_mutex_detach(&static_mutex); // 卸载静态互斥量
```

6. rt_mutex_delete

用于删除动态互斥量（互斥量对象由 `rt_mutex_create` 创建的情况下）。

```
rt_err_t rt_mutex_delete(rt_mutex_t mutex);
```

参数说明：

- `mutex` ：互斥量的句柄。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回相应的错误代码。

示例：

```
rt_mutex_delete(mutex); // 删除动态互斥量
```

事件集（Event）

事件集可以看作公交车站的场景。线程可以等待多个事件发生，比如等待公交车或等待同伴到达。如果满足某个或某几个条件，线程将被唤醒继续执行。

事件集的工作机制

事件集用于线程间的同步，可以让线程等待一个或多个事件的触发：

- **逻辑与（AND）**：线程等待多个事件同时发生才被唤醒。
- **逻辑或（OR）**：线程只需等待其中一个事件发生即可被唤醒。

事件集应用示例

```
#include <rtthread.h>
```

```c
/* 创建事件对象 */
rt_event_t event = RT_NULL;

/* 线程1：等待事件 */
void thread_entry1(void *parameter)
{
    rt_uint32_t received;
    rt_kprintf("线程1: 等待公交车3或5到来...\n");
    rt_event_recv(event, (1 << 3 | 1 << 5), RT_EVENT_FLAG_OR |
RT_EVENT_FLAG_CLEAR, RT_WAITING_FOREVER, &received);
    rt_kprintf("线程1: 公交车 %d 到站，出发！\n", received);
}

/* 线程2：发送事件（公交车到站） */
void thread_entry2(void *parameter)
{
    rt_thread_mdelay(500); // 模拟公交车到站
    rt_kprintf("线程2: 公交车3到站\n");
    rt_event_send(event, (1 << 3)); // 发送公交车3事件
}

int main(void)
{
    /* 初始化事件对象 */
    event = rt_event_create("event", RT_IPC_FLAG_PRIO);

    /* 创建两个线程 */
    rt_thread_t tid1 = rt_thread_create("t1", thread_entry1, RT_NULL, 1024, 25,
10);
    rt_thread_t tid2 = rt_thread_create("t2", thread_entry2, RT_NULL, 1024, 25,
10);

    /* 启动线程 */
    rt_thread_startup(tid1);
    rt_thread_startup(tid2);

    return 0;
}
```

解释：

- `thread_entry1` 线程等待公交车3或公交车5的到来，公交车到站时线程被唤醒。
- `thread_entry2` 线程模拟公交车3到站，发送事件，唤醒等待的线程。

场景：运动员等待两个裁判发令信号

假设有两个裁判，裁判1负责发出“准备”信号，裁判2负责发出“开始”信号。运动员需要等到两个信号都收到后才能起跑。

代码实现：

```
#include <rtthread.h>

/* 创建事件对象 */
rt_event_t event = RT_NULL;

/* 运动员线程：等待两个裁判信号 */
void athlete_entry(void *parameter)
{
rt_uint32_t received;
rt_kprintf("运动员: 等待裁判的准备信号和开始信号...\n");

 /* 等待两个事件：准备信号(位1)和开始信号(位2) */
rt_event_recv(event, (1 << 1 | 1 << 2), RT_EVENT_FLAG_AND |
RT_EVENT_FLAG_CLEAR, RT_WAITING_FOREVER, &received);

 /* 两个事件都收到，运动员起跑 */
rt_kprintf("运动员: 收到准备信号和开始信号，准备起跑! \n");
}

/* 裁判1线程：发送准备信号 */
void referee1_entry(void *parameter)
{
rt_thread_mdelay(500); // 模拟准备时间
rt_kprintf("裁判1: 发送准备信号\n");
 rt_event_send(event, (1 << 1)); // 发送准备信号事件（位1）
}

/* 裁判2线程：发送开始信号 */
void referee2_entry(void *parameter)
{
rt_thread_mdelay(1000); // 模拟准备时间
rt_kprintf("裁判2: 发送开始信号\n");
 rt_event_send(event, (1 << 2)); // 发送开始信号事件（位2）
}

int main(void)
```

```
{
/* 初始化事件对象 */
event = rt_event_create("event", RT_IPC_FLAG_PRIO);

 /* 创建运动员线程 */
rt_thread_t athlete_tid = rt_thread_create("athlete", athlete_entry,
RT_NULL, 1024, 25, 10);

 /* 创建裁判1线程 */
rt_thread_t referee1_tid = rt_thread_create("referee1", referee1_entry,
RT_NULL, 1024, 25, 10);

 /* 创建裁判2线程 */
rt_thread_t referee2_tid = rt_thread_create("referee2", referee2_entry,
RT_NULL, 1024, 25, 10);

 /* 启动线程 */
rt_thread_startup(athlete_tid);
 rt_thread_startup(referee1_tid);
 rt_thread_startup(referee2_tid);

 return 0;
}
```

代码说明：

1. **事件对象创建**：
   - 使用 `rt_event_create` 创建一个事件对象 `event` ，用于接收裁判1和裁判2的信号。
2. **运动员线程**：

- 运动员（ `athlete_entry` ）启动后，会等待两个裁判的信号（准备信号和开始信号）。
- `rt_event_recv` 的标志为 `RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR` ，表示运动员需要同时收到这两个事件才能继续执行，并且在接收后自动清除这两个事件。

3. **裁判1线程**：

- 裁判1（ `referee1_entry` ）在 500 毫秒后发送“准备”信号（位 1）。

4. **裁判2线程**：

- 裁判2（ `referee2_entry` ）在1000毫秒后发送“开始”信号（位2）。

1. rt_event_create

用于动态创建一个事件对象。

```
rt_event_t rt_event_create(const char *name, rt_uint8_t flag);
```

参数说明：

- `name` ：事件对象的名字（可以为 `RT_NULL` 表示匿名事件对象）。
- `flag` ：IPC 对象的属性标志，一般使用 `RT_IPC_FLAG_PRIO` 表示优先级等待，或 `RT_IPC_FLAG_FIFO` 表示先进先出等待。

返回值：

- 成功：返回事件对象的句柄 `rt_event_t` 。
- 失败：返回 `RT_NULL` 。

示例：

```
rt_event_t event = rt_event_create("event", RT_IPC_FLAG_PRIO);
```

2. rt_event_init

用于静态初始化事件对象。一般用于系统或内存已静态分配的场景。

```
rt_err_t rt_event_init(rt_event_t event, const char *name, rt_uint8_t
flag);
```

参数说明：

- `event` ：事件对象指针（需要预先定义事件对象结构体）。
- `name` ：事件对象名字。
- `flag` ：IPC 对象的属性标志，和 `rt_event_create` 一样，通常使用 `RT_IPC_FLAG_PRIO` 或 `RT_IPC_FLAG_FIFO` 。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回相应的错误代码。

示例：

```
/* 静态事件对象定义 */
static struct rt_event static_event;

/* 初始化静态事件对象 */
rt_event_init(&static_event, "static_event", RT_IPC_FLAG_PRIO);
```

3. rt_event_delete

用于删除动态创建的事件对象（ `rt_event_create` 创建的事件对象）。

```
rt_err_t rt_event_delete(rt_event_t event);
```

参数说明：

- `event` ： 事件对象的句柄。

返回值：

- 成功：返回 `RT_EOK` 。
- 失败：返回相应的错误代码。

示例：

```
rt_event_delete(event); // 删除动态创建的事件对象
```

4. rt_event_detach

用于卸载静态初始化的事件对象（ `rt_event_init` 初始化的事件对象）。

```
rt_err_t rt_event_detach(rt_event_t event);
```

参数说明：

- `event`：事件对象的句柄。

返回值：

- 成功：返回 `RT_EOK`。
- 失败：返回相应的错误代码。

示例：

```
rt_event_detach(&static_event); // 卸载静态初始化的事件对象
```

5. rt_event_send

用于发送（触发）一个事件，设置指定的事件标志位。线程会根据其等待的事件标志来判断是否继续执行。

```
rt_err_t rt_event_send(rt_event_t event, rt_uint32_t set);
```

参数说明：

- `event`：事件对象的句柄。
- `set`：事件标志，设置哪几位事件标志被触发（可以用 `1 << n` 来表示第 n 位的事件）。

返回值：

- 成功：返回 `RT_EOK`。
- 失败：返回相应的错误代码。

示例：

```
rt_event_send(event, (1 << 3)); // 触发位3的事件
```

6. rt_event_recv

用于接收（等待）事件，线程可以等待多个事件标志。当等待的事件标志满足条件后，线程被唤醒继续执行。

```
rt_err_t rt_event_recv(rt_event_t event, rt_uint32_t set, rt_uint8_t
option, rt_int32_t timeout, rt_uint32_t *recved);
```

参数说明：

- `event`：事件对象的句柄。
- `set`：需要等待的事件标志（例如 `1 << 3 | 1 << 5` 表示同时等待第3位和第5位的事件）。
- `option`：等待选项，常见选项包括：
  - `RT_EVENT_FLAG_AND`：所有指定的事件标志都满足时才唤醒线程。
  - `RT_EVENT_FLAG_OR`：只要有一个指定的事件标志满足就唤醒线程。
  - `RT_EVENT_FLAG_CLEAR`：收到事件标志后清除这些事件标志。
- `timeout`：超时时间（单位为系统 tick），可以为 `RT_WAITING_FOREVER` 表示永远等待。
- `recved`：输出参数，返回实际接收到的事件标志。

返回值：

- 成功：返回 `RT_EOK`。
- 失败：返回 `-RT_ETIMEOUT` 表示超时，或其他错误代码。

示例：

```
rt_uint32_t recved;
rt_event_recv(event, (1 << 3 | 1 << 5), RT_EVENT_FLAG_OR |
RT_EVENT_FLAG_CLEAR, RT_WAITING_FOREVER, &recved);
```

总结：

- **rt_event_create**：用于动态创建事件对象。
- **rt_event_init**：用于静态初始化事件对象。
- **rt_event_delete**：用于删除动态事件对象。
- **rt_event_detach**：用于卸载静态初始化的事件对象。
- **rt_event_send**：发送事件，触发指定的事件标志位。
- **rt_event_recv**：接收事件，线程根据等待条件阻塞，直到指定事件发生。

线程间通信

在多线程编程中，线程间通信是非常常见的需求。RT-Thread 提供了多种工具来帮助我们在不同线程之间传递信息。

> 全局变量的确可以用于线程间通信，但在多线程环境下使用全局变量来传递信息存在许多潜在问题和局限性，因此操作系统提供了更安全、高效的通信机制，如邮箱、消息队列和信号。这些机制能够避免全局变量带来的不良影响，并在实际应用中提供了更好的性能和灵活性。
>
> 以下是使用全局变量的问题和局限性，以及为什么使用邮箱、消息队列、信号等工具更好：
>
> ## 1. 数据竞争与同步问题
>
> 多线程环境下，如果多个线程同时读写一个全局变量，而没有使用锁或同步机制，会导致数据竞争（Race Condition），即多个线程同时操作全局变量，导致结果不确定或数据错误。
>
> 举个例子：
>
> 假设有两个线程 `thread1` 和 `thread2` 都要对同一个全局变量 `counter` 进行加1操作：
>
> ```
> int counter = 0;
>
> void thread1_entry(void *parameter)
> {
>  counter += 1;
> }
>
> void thread2_entry(void *parameter)
> {
>  counter += 1;
> }
> ```
>
> 两个线程可能会在不同的时间修改 `counter` ，但如果它们同时修改这个变量，可能会出现以下问题：
>
> - `counter` 在两个线程同时读取时都是 0；
> - 然后两个线程都将 `counter` 加 1；
> - 最终， `counter` 变成了 1，而不是期望的 2。

邮箱/消息队列的优势：
邮箱和消息队列都是线程安全的通信机制，能够保证消息传递的原子性和数据一致性。在使用这些工具时，操作系统会自动处理线程间的同步问题，不会发生类似的数据竞争。

2. 数据丢失问题

全局变量是共享的，如果一个线程写入了某个全局变量的值，而另一个线程还没有及时读取该值，写入的值可能会被覆盖，导致数据丢失。

举个例子：

如果 `thread1` 先将 `global_value` 设置为10，然后 `thread2` 将其改为20，而 `thread3` 需要读取这个值，那么 `thread3` 可能读取到的是最新的20，而忽略了中间的10。也就是说，如果通信信息被新值覆盖了，旧值将永远丢失。

邮箱/消息队列的优势：
邮箱和消息队列可以缓冲多个消息，并且消息不会被覆盖。当多个线程发送消息时，这些消息会被保存在队列中，直到目标线程逐一处理所有消息。这确保了数据的完整性和顺序性。

3. 同步复杂性

使用全局变量时，开发者需要手动管理线程间的同步机制，比如使用信号量、互斥锁等工具，来确保多个线程同时操作全局变量时不会产生冲突。然而，手动管理同步机制既复杂又容易出错，特别是在涉及多个线程的场景下。

举个例子：

如果你需要在多个线程中通过全局变量传递消息，那么你可能需要这样做：

```
int message;
rt_mutex_t lock;

void thread1_entry(void *parameter)
{
    rt_mutex_take(lock, RT_WAITING_FOREVER); // 获取锁
    message = 1; // 写入数据
    rt_mutex_release(lock); // 释放锁
}

void thread2_entry(void *parameter)
{
    rt_mutex_take(lock, RT_WAITING_FOREVER); // 获取锁
    int msg = message; // 读取数据
```

```
        rt_mutex_release(lock); // 释放锁
    }
```

在上面的例子中，你需要手动管理锁的获取和释放，避免线程竞争。如果忘记释放锁，可能会导致死锁等问题。随着线程和全局变量的数量增多，管理复杂度也会成倍增加。

邮箱/消息队列的优势：
邮箱和消息队列的同步机制由操作系统自动处理。你只需发送和接收消息，不需要关心底层的锁和同步问题，简化了代码编写的复杂性。

4. 扩展性差

当系统中有多个线程需要通信时，使用全局变量的扩展性很差。每个通信操作都需要为全局变量设计新的同步机制，这会导致系统变得难以维护。如果你需要同时传递多个消息，还需要为每个全局变量手动设计新的逻辑。

邮箱/消息队列的优势：
邮箱和消息队列具有天然的扩展性。多个线程可以通过邮箱或消息队列进行通信，每个线程只需处理自己的消息即可，无需设计复杂的共享内存机制。消息队列还可以处理不同大小的消息，因此适用于传递多种类型的信息。

5. 信号机制的异步性

有时线程之间需要进行异步通信，比如当某个线程需要紧急通知其他线程执行某些操作时。全局变量无法很好地实现这种异步通知机制，必须通过轮询（Polling）等低效的方式进行检查。

信号的优势：
信号是一种异步通信机制，允许线程在不等待的情况下接受通知。当某个线程发送信号给另一个线程时，接收线程可以立即停止当前任务并处理信号。这使得信号非常适合用来实现异步事件处理。

总结

虽然全局变量在一些简单场景下可以实现线程间的通信，但它容易引发数据竞争、数据丢失、同步复杂等问题。邮箱、消息队列和信号等工具不仅解决了这些问题，还提供了更加安全、高效和灵活的线程间通信方式。

使用邮箱、消息队列、信号等机制，你可以：

1. **避免数据竞争**，保证通信数据的准确性；
2. **不丢失数据**，保证所有发送的数据都能被处理；
3. **简化同步处理**，无需手动管理复杂的锁；
4. **提升扩展性**，轻松处理多线程、多种消息类型；

> 5. **实现异步通信**，应对实时性要求高的场景。
>
> 这些机制使得你的系统更加可靠、可维护，并且易于扩展。

邮箱

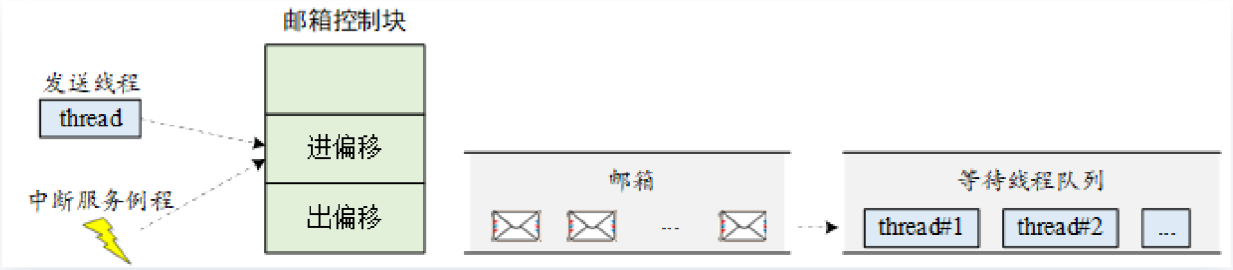
邮箱在操作系统中类似于一个专门用来传递“邮件”的信箱。线程 1 可以将一封邮件（即 4 字节的消息）放入邮箱，线程 2 则可以从邮箱中接收到这封邮件。这种机制适合多个线程相互之间传递简单的数据或信息。

比喻解释

假设有两个朋友，一个负责收集信件（线程 1），一个负责查看信件（线程 2）。收集信件的朋友将信息放入邮箱中，查看信件的朋友会定期打开邮箱查收邮件。这样，他们就可以通过“邮箱”进行通信，而不需要直接对话。

邮箱的工作机制

- **发送邮件**：线程将一条 4 字节的数据（邮件）发送到邮箱。如果邮箱满了，发送方可以选择等待，直到邮箱中有空位。
- **接收邮件**：接收方线程从邮箱中读取邮件。如果邮箱为空，接收方可以选择等待，直到邮箱中有新邮件。



邮箱示例代码

```c
#include <rtthread.h>

/* 创建邮箱 */
rt_mailbox_t mb;

/* 线程1：发送邮件 */
void thread_entry1(void *parameter)
{
    char msg = 'A'; // 发送字母'A'作为邮件
    rt_kprintf("线程1：发送邮件...\n");
    rt_mb_send(mb, (rt_uint32_t)msg); // 发送邮件
```

```c
}

/* 线程2：接收邮件 */
void thread_entry2(void *parameter)
{
    char msg;
    rt_kprintf("线程2：等待接收邮件...\n");
    rt_mb_recv(mb, (rt_uint32_t*)&msg, RT_WAITING_FOREVER); // 接收邮件
    rt_kprintf("线程2：收到邮件：%c\n", msg); // 打印收到的邮件内容
}

int main(void)
{
    /* 创建一个容量为4的邮箱 */
    mb = rt_mb_create("mb", 4, RT_IPC_FLAG_PRIO);

    /* 创建两个线程 */
    rt_thread_t tid1 = rt_thread_create("t1", thread_entry1, RT_NULL, 1024, 10,
10);
    rt_thread_t tid2 = rt_thread_create("t2", thread_entry2, RT_NULL, 1024, 10,
10);

    /* 启动线程 */
    rt_thread_startup(tid1);
    rt_thread_startup(tid2);

    return 0;
}


#include <rtthread.h>

/* 定义传感器数据结构体 */
struct sensor_data
{
    int temperature;    // 温度（整型）
    int humidity;       // 湿度（整型）
    int acceleration;   // 加速度（整型）

};

/* 创建邮箱 */
rt_mailbox_t mb;

/* 全局结构体变量，确保生命周期有效 */
static struct sensor_data global_data;
```

```c
/* 线程1：发送传感器数据 */
void thread_entry1(void *parameter)
{

    /* 模拟整型传感器数据 */
    global_data.temperature = 25;     // 模拟温度数据，整型
    global_data.humidity = 60;        // 模拟湿度数据，整型
    global_data.acceleration = 10;    // 模拟加速度数据，整型

    rt_kprintf("线程1：获取传感器数据...\n");
    rt_thread_delay(2000);     // 模拟数据获取延迟

    /* 发送传感器数据 */
    rt_kprintf("线程1：发送传感器数据...\n");
    rt_mb_send(mb, (rt_uint32_t)&global_data); // 发送全局结构体数据的地址
}

/* 线程2：接收传感器数据 */
void thread_entry2(void *parameter)
{
    struct sensor_data *received_data;

    rt_kprintf("线程2：等待接收传感器数据...\n");
    /* 接收传感器数据 */
    rt_mb_recv(mb, (rt_uint32_t*)&received_data, RT_WAITING_FOREVER);

    /* 打印接收到的传感器数据 */
    rt_kprintf("线程2：收到传感器数据：\n");
    rt_kprintf("温度：%d°C\n", received_data->temperature);
    rt_kprintf("湿度：%d%%\n", received_data->humidity);
    rt_kprintf("加速度：%d m/s^2\n", received_data->acceleration);
}

int main(void)
{
    /* 创建一个容量为4的邮箱 */
    mb = rt_mb_create("mb", 4, RT_IPC_FLAG_PRIO);

    /* 创建两个线程 */
    rt_thread_t tid1 = rt_thread_create("t1", thread_entry1, RT_NULL, 1024, 10,
10);
    rt_thread_t tid2 = rt_thread_create("t2", thread_entry2, RT_NULL, 1024, 10,
10);

    /* 启动线程 */
```

```
    rt_thread_startup(tid1);
    rt_thread_startup(tid2);

    return 0;
}
```

解释:

- `thread_entry1` 发送一封邮件（字符 'A'）。
- `thread_entry2` 从邮箱中接收邮件，并打印出来

1. rt_mb_init

- **功能**: 初始化一个邮箱（邮件队列）对象。
- **原型**:

```
rt_err_t rt_mb_init(rt_mailbox_t mb, const char *name, void
*msgpool, rt_size_t size, rt_uint8_t flag);
```

- **参数**:
  - `mb` ：邮箱控制块。
  - `name` ：邮箱的名称。
  - `msgpool` ：消息池（存储消息的地方）。
  - `size` ：消息池的大小，单位为消息个数。
  - `flag` ：线程等待邮箱操作时的方式。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

2. **rt_mb_detach**

- **功能**: 销毁已初始化的邮箱对象。
- **原型**:

```
rt_err_t rt_mb_detach(rt_mailbox_t mb);
```

- **参数**:
  - `mb` : 邮箱控制块。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

3. **rt_mb_create**

- **功能**: 动态创建一个邮箱。
- **原型**:

```
rt_mailbox_t rt_mb_create(const char *name, rt_size_t size,
rt_uint8_t flag);
```

- **参数**:
  - `name` : 邮箱的名称。
  - `size` : 邮箱的大小（存储消息的个数）。
  - `flag` : 邮箱操作的标志。
- **返回值**: 返回动态创建的邮箱对象，失败时返回 `RT_NULL` 。

4. **rt_mb_delete**

- **功能**: 删除动态创建的邮箱。
- **原型**:

```
rt_err_t rt_mb_delete(rt_mailbox_t mb);
```

- **参数**:
  - `mb` : 邮箱控制块。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

5. **rt_mb_send**

- **功能**: 向邮箱发送一条消息。
- **原型**:

```
rt_err_t rt_mb_send(rt_mailbox_t mb, rt_ubase_t value);
```

- **参数**:
  - `mb` : 邮箱控制块。
  - `value` : 发送的消息内容（消息一般是地址或整型数据）。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

6. **rt_mb_send_wait**

- **功能**: 向邮箱发送一条消息（带超时等待）。
- **原型**:

```
rt_err_t rt_mb_send_wait(rt_mailbox_t mb, rt_ubase_t value,
rt_int32_t timeout);
```

- **参数**:
  - `mb` : 邮箱控制块。
  - `value` : 发送的消息内容。
  - `timeout` : 超时时间（以 tick 为单位）， `RT_WAITING_FOREVER` 表示永久等待。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

7. **rt_mb_recv**

- **功能**: 接收一条消息，如果没有消息则等待。
- **原型**:

```
rt_err_t rt_mb_recv(rt_mailbox_t mb, rt_ubase_t *value, rt_int32_t
timeout);
```

- **参数**:
  - `mb` : 邮箱控制块。
  - `value` : 接收的消息内容指针。

    - `timeout`：超时时间，`RT_WAITING_FOREVER` 表示永久等待。
  - **返回值**：`RT_EOK` 表示成功，其他错误码表示失败。

8. **rt_mb_urgent_send**
   - **功能**：发送一条紧急消息（紧急消息会插入到消息队列的前端，而不是末端）。
   - **原型**：

     ```
     rt_err_t rt_mb_urgent_send(rt_mailbox_t mb, rt_ubase_t value);
     ```

   - **参数**：
     - `mb`：邮箱控制块。
     - `value`：要发送的消息内容。
   - **返回值**：`RT_EOK` 表示成功，其他错误码表示失败。

1. 动态创建（Dynamic Creation）

动态创建意味着在程序运行时，通过 API 动态地申请内存和创建对象。这种方式的特点是灵活，适合在运行时根据需求创建对象。

特点：

- **内存分配**：动态创建的对象使用**堆**中的内存，这意味着内存是在运行时通过动态分配函数（如 `malloc` ）获取的。
- **函数调用**：使用 `rt_mb_create` 来创建邮箱对象。
  - 例如，`rt_mb_create` 会在堆上分配内存，然后初始化该对象。
- **销毁方式**：动态创建的对象需要在使用完之后通过相应的销毁函数（如 `rt_mb_delete` ）来释放。
- **灵活性**：动态创建适合于在程序运行期间根据实际需求动态创建和销毁对象，尤其适合那些数量或大小在编译期无法确定的场景。
- **例子**：

```
rt_mailbox_t mb = rt_mb_create("mb", 10, RT_IPC_FLAG_FIFO); // 动态创建
// 使用完成后需要删除
rt_mb_delete(mb);
```

优点：

- 可以根据运行时的实际需求创建资源，不需要在编译时决定资源的大小或数量。
- 更加灵活，适用于复杂和资源需求变化的应用。

缺点：

- 由于需要动态分配内存，可能会带来额外的时间开销和内存碎片问题。
- 必须显式删除/释放，否则可能导致内存泄漏。

2. 静态初始化（Static Initialization）

静态初始化则是在编译时或系统启动时就已经分配好内存，并初始化对象。这种方式使用在编译时已知大小和数量的对象。

特点：

- **内存分配**: 静态初始化的对象使用**栈**或**静态内存区域**，也就是说内存分配是在编译时或系统启动时完成的，不依赖于动态内存管理。
- **函数调用**: 使用 `rt_mb_init` 来初始化邮箱对象。
  - 例如， `rt_mb_init` 初始化时需要一个预分配好的消息池和控制块，内存是由用户提前准备的。
- **销毁方式**: 静态初始化的对象不需要删除，只需在系统退出或不再使用时通过 `rt_mb_detach` 函数进行资源释放或复位。
- **确定性**: 静态初始化适用于那些对象大小和数量在编译期可以确定的场景，因为内存已经在编译时分配好。
- **例子**:

```
static struct rt_mailbox mb;
static rt_uint32_t mb_pool[10];
rt_mb_init(&mb, "mb", mb_pool, sizeof(mb_pool) / 4,
RT_IPC_FLAG_FIFO); // 静态初始化
// 使用完成后可以 detach
rt_mb_detach(&mb);
```

优点：

- 没有动态内存分配的开销，初始化更快，适合对性能要求较高的场景。
- 内存分配在编译时完成，使用上更安全，不存在动态分配失败的风险。
- 无需担心内存泄漏问题，因为所有的内存都是预分配好的。

缺点：

- 缺乏灵活性，必须在编译时确定对象的大小和数量，无法在运行时调整。
- 需要提前分配内存资源，如果对象很大或者数量很多，可能会浪费内存。

3. 总结

特性	动态创建	静态初始化
内存分配	在堆中动态分配	在栈或静态内存中分配
内存管理	需要显式删除和释放	不需要显式删除，系统管理
使用场景	运行时灵活创建，适用于动态需求	编译时确定大小，适用于固定需求
性能开销	存在内存分配和释放的开销	无动态分配的性能开销
安全性	可能导致内存泄漏或碎片	更安全，内存使用确定
API	`rt_mb_create` / `rt_mb_delete`	`rt_mb_init` / `rt_mb_detach`

什么时候选择动态创建 vs 静态初始化?

- **动态创建**: 如果你的应用需要根据运行时的条件动态地创建和销毁对象，例如数量或大小不固定，或者资源使用具有不确定性，那么动态创建更合适。
- **静态初始化**: 如果你已经知道在应用中需要的资源数量和大小，并且希望更高效和确定的资源管理，那么静态初始化是更好的选择。

两者可以根据实际需求选择，并且在同一个项目中也可以混合使用。

消息队列

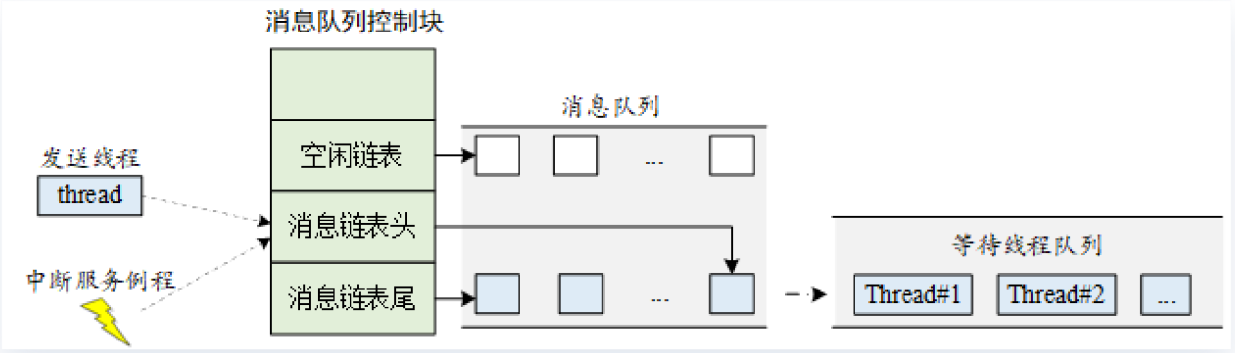
消息队列类似于邮箱，但它能处理更大、更复杂的消息，可以包含多个字节的数据，而不仅仅是 4 字节的简单邮件。多个线程可以向消息队列发送消息，接收线程可以从中读取并处理这些消息。

比喻解释

假设你和朋友在共享一个快递柜（消息队列），你们可以向柜子里放入不同大小的包裹（消息），快递柜负责保管这些包裹。另一个朋友则可以从快递柜中取出包裹并处理，无论是大包裹还是小包裹都可以轻松处理。

消息队列的工作机制

- **发送消息**：线程可以将任意大小的消息发送到消息队列。如果消息队列满了，线程会等待，直到有空间可以存储新消息。
- **接收消息**：线程可以从消息队列中接收消息。如果消息队列为空，线程可以选择等待新的消息到来。



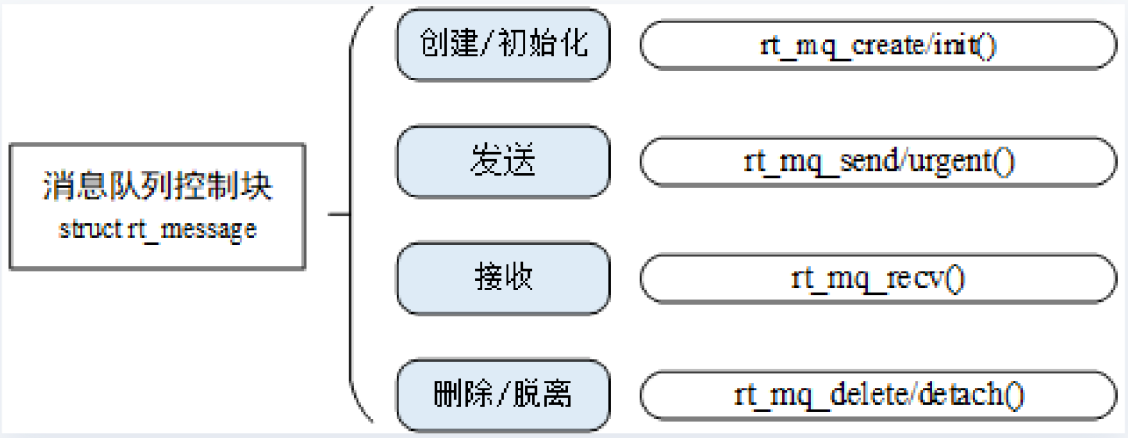
消息队列示例代码

```c
#include <rtthread.h>

/* 定义传感器数据结构体 */
struct sensor_data
{
    int temperature;    // 温度（整型）
    int humidity;       // 湿度（整型）
    int acceleration;   // 加速度（整型）
};

/* 创建消息队列 */
rt_mq_t mq;

/* 线程1：发送传感器数据 */
void thread_entry1(void *parameter)
{
    /* 模拟整型传感器数据 */
    struct sensor_data data;
    data.temperature = 25;     // 模拟温度数据，整型
    data.humidity = 60;        // 模拟湿度数据，整型
    data.acceleration = 10;    // 模拟加速度数据，整型

    rt_kprintf("线程1：获取传感器数据...\n");
    rt_thread_delay(2000);     // 模拟数据获取延迟

    /* 发送传感器数据 */
```

```
    rt_kprintf("线程1: 发送传感器数据...\n");
    rt_mq_send(mq, &data, sizeof(data)); // 发送整个结构体数据
}

/* 线程2: 接收传感器数据 */
void thread_entry2(void *parameter)
{
    struct sensor_data received_data;

    rt_kprintf("线程2: 等待接收传感器数据...\n");
    /* 接收传感器数据 */
    rt_mq_recv(mq, &received_data, sizeof(received_data), RT_WAITING_FOREVER);

    /* 打印接收到的传感器数据 */
    rt_kprintf("线程2: 收到传感器数据: \n");
    rt_kprintf("温度: %d°C\n", received_data.temperature);
    rt_kprintf("湿度: %d%%\n", received_data.humidity);
    rt_kprintf("加速度: %d m/s^2\n", received_data.acceleration);
}

int main(void)
{
    /* 创建一个容量为4，消息大小为结构体大小的消息队列 */
    mq = rt_mq_create("mq", sizeof(struct sensor_data), 4, RT_IPC_FLAG_PRIO);

    /* 创建两个线程 */
    rt_thread_t tid1 = rt_thread_create("t1", thread_entry1, RT_NULL, 1024, 10,
10);
    rt_thread_t tid2 = rt_thread_create("t2", thread_entry2, RT_NULL, 1024, 10,
10);

    /* 启动线程 */
    rt_thread_startup(tid1);
    rt_thread_startup(tid2);

    return 0;
}
```

解释:

- `thread_entry1` 发送一条消息到消息队列。
- `thread_entry2` 从消息队列接收并打印消息。



1. rt_mq_init

- **功能**: 初始化一个静态消息队列。
- **原型**:

```
rt_err_t rt_mq_init(rt_mq_t mq, const char *name, void *msgpool,
rt_size_t msg_size, rt_size_t pool_size, rt_uint8_t flag);
```

- **参数**:
  - `mq` : 消息队列控制块。
  - `name` : 消息队列的名称。
  - `msgpool` : 消息存储池。
  - `msg_size` : 每个消息的大小。
  - `pool_size` : 消息池的大小（总消息个数）。
  - `flag` : 消息队列的操作方式。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

2. rt_mq_detach

- **功能**: 销毁一个静态初始化的消息队列。
- **原型**:

```
rt_err_t rt_mq_detach(rt_mq_t mq);
```

- **参数**:

    - `mq`：消息队列控制块。
- **返回值**：`RT_EOK` 表示成功，其他错误码表示失败。

3. rt_mq_create

- **功能**：动态创建一个消息队列。
- **原型**：

```
rt_mq_t rt_mq_create(const char *name, rt_size_t msg_size, rt_size_t
max_msgs, rt_uint8_t flag);
```

- **参数**：
    - `name`：消息队列名称。
    - `msg_size`：每条消息的大小。
    - `max_msgs`：消息队列可以容纳的最大消息数。
    - `flag`：消息队列操作的标志。
- **返回值**：成功时返回消息队列控制块指针，失败时返回 `RT_NULL` 。

4. rt_mq_delete

- **功能**：删除动态创建的消息队列。
- **原型**：

```
rt_err_t rt_mq_delete(rt_mq_t mq);
```

- **参数**：
    - `mq`：消息队列控制块。
- **返回值**：`RT_EOK` 表示成功，其他错误码表示失败。

5. rt_mq_send

- **功能**：发送一条消息到消息队列。
- **原型**：

```
rt_err_t rt_mq_send(rt_mq_t mq, const void *buffer, rt_size_t size);
```

- **参数**:
  - `mq` : 消息队列控制块。
  - `buffer` : 待发送消息的内容。
  - `size` : 消息的大小。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

6. rt_mq_send_wait

- **功能**: 发送一条消息到消息队列，带超时等待。
- **原型**:

```
rt_err_t rt_mq_send_wait(rt_mq_t mq, const void *buffer, rt_size_t
size, rt_int32_t timeout);
```

- **参数**:
  - `mq` : 消息队列控制块。
  - `buffer` : 待发送消息的内容。
  - `size` : 消息的大小。
  - `timeout` : 超时时间，单位是系统 tick， `RT_WAITING_FOREVER` 表示永久等待。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

7. rt_mq_urgent_send

- **功能**: 发送一条紧急消息到消息队列（紧急消息插入到队列前端）。
- **原型**:

```
rt_err_t rt_mq_urgent_send(rt_mq_t mq, const void *buffer, rt_size_t
size);
```

- **参数**:
  - `mq` : 消息队列控制块。
  - `buffer` : 待发送的紧急消息内容。
  - `size` : 消息的大小。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

8. rt_mq_recv

- **功能**: 接收一条消息，如果没有消息则阻塞等待。
- **原型**:

```
rt_err_t rt_mq_recv(rt_mq_t mq, void *buffer, rt_size_t size,
rt_int32_t timeout);
```

- **参数**:
  - `mq` : 消息队列控制块。
  - `buffer` : 存放接收到的消息内容的缓冲区。
  - `size` : 消息的大小。
  - `timeout` : 超时时间，单位为系统 tick， `RT_WAITING_FOREVER` 表示永久等待。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

9. rt_mq_control

- **功能**: 控制消息队列的某些功能，例如复位消息队列。
- **原型**:

```
rt_err_t rt_mq_control(rt_mq_t mq, int cmd, void *arg);
```

- **参数**:
  - `mq` : 消息队列控制块。
  - `cmd` : 控制命令（如 `RT_IPC_CMD_RESET` ，表示复位消息队列）。
  - `arg` : 相关的命令参数。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。

信号

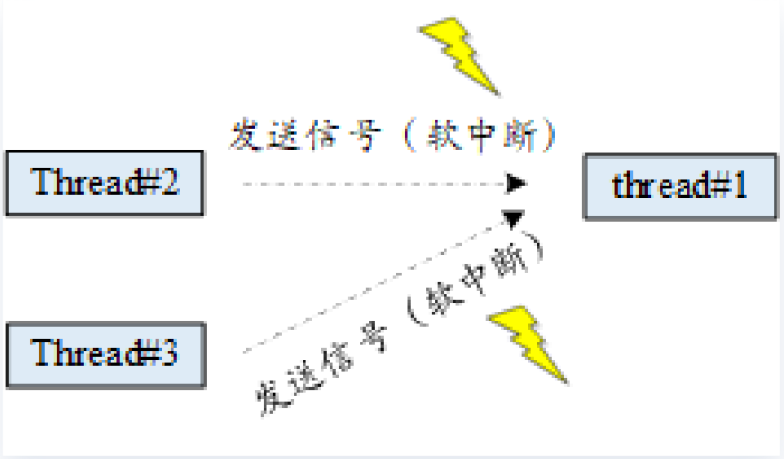
信号可以理解为一种“通知机制”，类似于紧急情况下发出的警报。线程可以通过信号告知另一个线程某个事件已经发生，常用于线程之间的异步通信和通知。

比喻解释

信号就像一个紧急电话。如果你想让你的朋友注意到某件重要的事情，你可以打电话通知他。朋友接到电话（信号）后，可以立即采取行动或处理相关事宜。

信号的工作机制

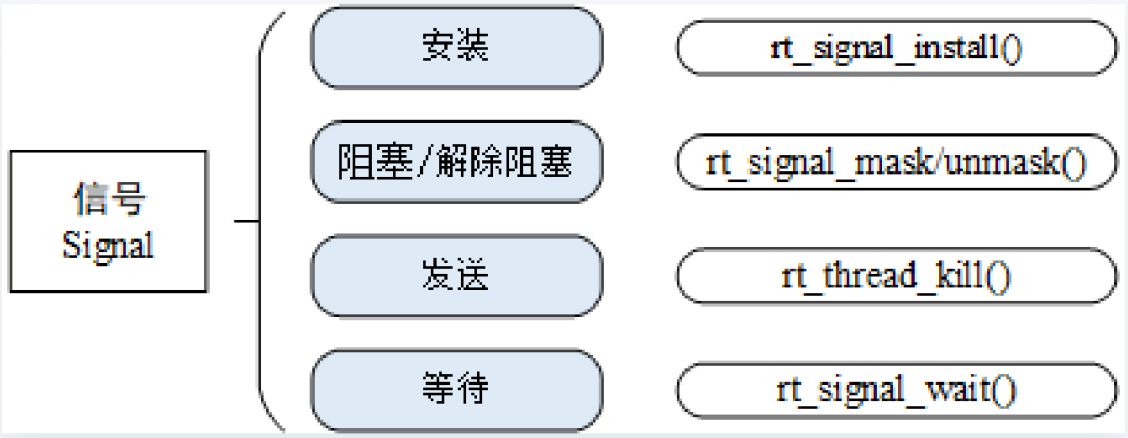
- **发送信号**：线程可以向另一个线程发送信号。信号本质上是一个异步通知，可以让目标线程立即中断当前工作，转而处理信号事件。
- **处理信号**：目标线程需要先安装信号处理函数，收到信号时会执行相应的处理操作。



信号示例代码

```
#include <rtthread.h>

#define THREAD_PRIORITY         25
#define THREAD_STACK_SIZE       512
#define THREAD_TIMESLICE        5

static rt_thread_t tid1 = RT_NULL;

/* 线程 1 的信号处理函数 */
void thread1_signal_handler(int sig)
{
    rt_kprintf("thread1 received signal %d\n", sig);
}

/* 线程 1 的入口函数 */
static void thread1_entry(void *parameter)
{
    int cnt = 0;

    /* 安装信号 */
```

```
    rt_signal_install(SIGUSR1, thread1_signal_handler);
    rt_signal_unmask(SIGUSR1);

    /* 运行 10 次 */
    while (cnt < 10)
    {
        /* 线程 1 采用低优先级运行，一直打印计数值 */
        rt_kprintf("thread1 count : %d\n", cnt);

        cnt++;
        rt_thread_mdelay(100);
    }
}

/* 信号示例的初始化 */
int signal_sample(void)
{
    /* 创建线程 1 */
    tid1 = rt_thread_create("thread1",
                            thread1_entry, RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);

    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    rt_thread_mdelay(300);

    /* 发送信号 SIGUSR1 给线程 1 */
    rt_thread_kill(tid1, SIGUSR1);

    return 0;
}

int main(void)
{
    signal_sample();
}
```



1. rt_thread_kill

- **功能**: 发送信号给指定线程。
- **原型**:

```
int rt_thread_kill(rt_thread_t tid, int sig);
```

- **参数**:
  - `tid` : 目标线程的控制块指针。
  - `sig` : 要发送的信号，信号编号通常是 `SIGUSR1` 、 `SIGUSR2` 等。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。
- **使用**:
  在 `thread2_entry` 中，使用 `rt_thread_kill` 向线程1发送 `SIGUSR1` 和 `SIGUSR2` 信号。

```
rt_thread_kill(tid1, SIGUSR1);  // 向线程1发送SIGUSR1信号
```

2. rt_signal_wait

- **功能**: 等待指定的信号集，直到信号到达或者超时。
- **原型**:

```
int rt_signal_wait(const rt_sigset_t *set, rt_siginfo_t *si,
rt_int32_t timeout);
```

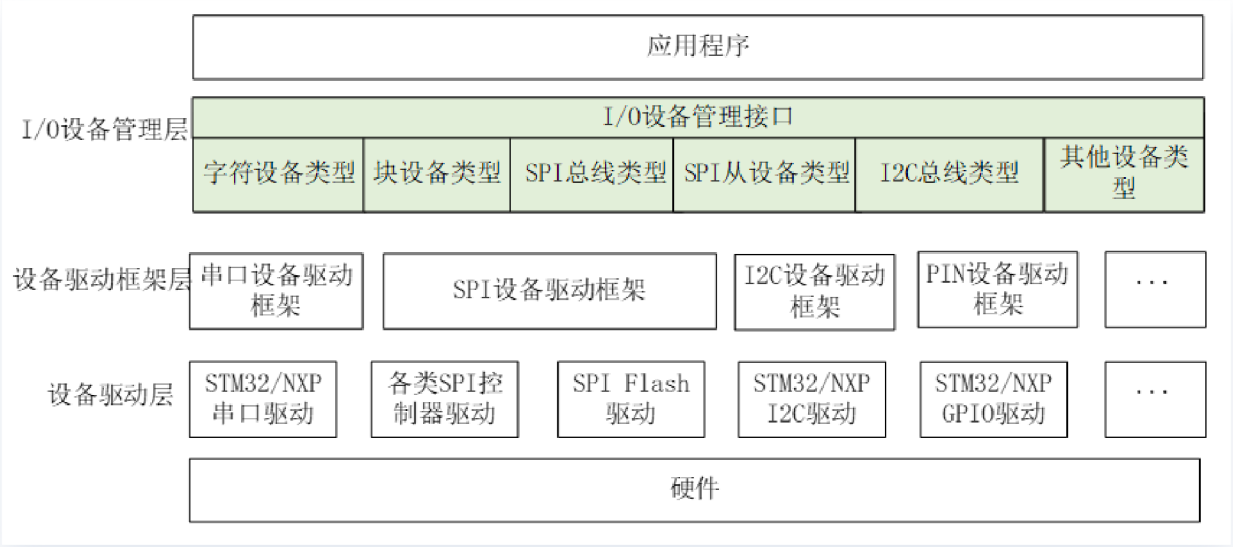
- **参数**:
  - `set`：需要等待的信号集，使用 `sig_mask` 创建信号掩码。
  - `si`：用于存储接收到的信号信息的结构体指针。
  - `timeout`：等待的超时时间，单位为系统 tick， `RT_WAITING_FOREVER` 表示永久等待。
- **返回值**: `RT_EOK` 表示成功，其他错误码表示失败。
- **使用**:
  在 `thread1_entry` 中，使用 `rt_signal_wait` 等待 `SIGUSR1` 和 `SIGUSR2` 信号。

```
rt_signal_wait(&set, &si, RT_WAITING_FOREVER);  // 等待信号
```

I/O 设备模型

嵌入式系统中经常使用各种 I/O（输入/输出）设备，比如你手机上的显示屏、连接的耳机，或者工业设备中的串口通信模块等。RT-Thread 的 I/O 设备模型帮助我们管理这些设备，确保它们可以高效、灵活地使用。

I/O 设备模型框架



RT-Thread 提供了一种三层的 I/O 设备模型框架，用来处理硬件和应用程序之间的通信。你可以把它想象成一座桥梁，连接了底层的硬件设备（比如串口、传感器）和上层的应用程序。三层架构如下：

1. **I/O 设备管理层**：这是最上层，就像一名调度员，负责管理不同的设备驱动，确保应用程序可以通过统一的方式与设备互动，不用关心硬件的具体细节。
2. **设备驱动框架层**：这是中间层，对相同类型的设备进行了抽象和统一管理。你可以把它理解为各种家电（比如冰箱、电视）的通用接口，即使它们来自不同的品牌，但使用方式是一样的。
3. **设备驱动层**：这是最底层，直接与硬件打交道，负责实际控制设备。这一层根据具体的硬件实现功能，比如控制串口的读写操作。

通过这个模型，设备的硬件细节被很好地隐藏起来，应用程序可以专注于自身的功能，而不需要操心硬件的变化。

设备注册与查找

就像你新买了一个家电，必须要先安装和注册（告诉系统有这个设备），才能开始使用。在 RT-Thread 中，我们通过 `rt_device_register()` 函数来注册设备。

注册后，应用程序可以通过 `rt_device_find()` 函数来查找设备，就像你在家里找到了电冰箱，知道它的位置，接下来就可以使用了。

常见 I/O 设备类型

RT-Thread 支持多种 I/O 设备类型，比如：

- **字符设备**：像键盘、串口这种每次传输一个字符的设备。
- **块设备**：像 SD 卡或者硬盘这种每次传输一块数据的设备。

举个例子，字符设备就像你一个字母一个字母地输入，而块设备就像你一次搬运一箱货物，速度更快但处理起来稍微复杂一些。

设备操作方法

每个设备都有一套标准操作方法，就像你操作家电时的开关、调节按钮一样。设备的常见操作包括：

- **初始化**（`init`）：启动设备，让它准备好工作。比如打开冰箱电源，让它开始运行。
- **打开**（`open`）：设备开始真正工作，比如让冰箱开始制冷。
- **关闭**（`close`）：停止设备工作。
- **读写数据**（`read/write`）：与设备进行数据交互，比如读取传感器的数据或者向串口发送信息。

访问 I/O 设备

设备注册好后，应用程序就可以通过这些操作方法访问设备。常见的步骤如下：

1. **查找设备**：通过 `rt_device_find()` 查找到需要使用的设备。
2. **初始化设备**：通过 `rt_device_init()` 初始化设备，确保设备已准备好。
3. **打开设备**：通过 `rt_device_open()` 开启设备。
4. **读写数据**：使用 `rt_device_read()` 和 `rt_device_write()` 来读取和写入数据。
5. **关闭设备**：操作结束后，通过 `rt_device_close()` 关闭设备。

设备管理示例

假设你有一个看门狗设备（类似一个系统安全守护程序，确保系统不会挂掉）。首先，我们通过 `rt_device_find()` 找到看门狗设备，然后初始化并设置它的超时时间。当系统空闲时，还可以调用回调函数，确保看门狗一直保持活跃。具体代码如下：

```
#include <rtthread.h>
#include <rtdevice.h>

#define IWDG_DEVICE_NAME    "wdt"    // 看门狗设备名称

static rt_device_t wdg_dev;  // 看门狗设备句柄

static void idle_hook(void)
{
    // 在空闲线程的回调函数里喂狗，避免超时
    rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_KEEPALIVE, NULL);
    rt_kprintf("feed the dog!\n");
}

int main(void)
{
    rt_err_t res = RT_EOK;
    rt_uint32_t timeout = 10;  // 看门狗超时时间

    // 查找看门狗设备
    wdg_dev = rt_device_find(IWDG_DEVICE_NAME);
    if (!wdg_dev)
    {
        rt_kprintf("find %s failed!\n", IWDG_DEVICE_NAME);
        return RT_ERROR;
    }

    // 初始化设备
```

```
    res = rt_device_init(wdg_dev);
    if (res != RT_EOK)
    {
        rt_kprintf("initialize %s failed!\n", IWDG_DEVICE_NAME);
        return res;
    }

    // 设置看门狗超时时间
    res = rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_SET_TIMEOUT, &timeout);
    if (res != RT_EOK)
    {
        rt_kprintf("set %s timeout failed!\n", IWDG_DEVICE_NAME);
        return res;
    }

    // 设置空闲线程的回调函数
    rt_thread_idle_sethook(idle_hook);

    return res;
}
```

引脚设备（PIN）

引脚的基本分类

芯片上的引脚通常可以分为以下几类：

- **电源引脚**：为芯片供电。
- **时钟引脚**：提供芯片的时钟信号。
- **控制引脚**：用于芯片的状态控制。
- **I/O引脚**：用于数据的输入输出。

其中，I/O 引脚又分为两类：

1. **通用输入输出引脚（GPIO）**：可以自由设置为输入或输出。
2. **复用功能引脚**：用于特定的通信功能，比如 SPI、I2C、UART 等。

I/O 引脚的主要特点是：

- 可以编程控制中断，支持 5 种中断触发模式，如上升沿、下降沿等。
- 输入输出模式可以灵活配置，比如上拉、下拉、开漏、推挽等。

RT-Thread 访问引脚设备

RT-Thread 提供了一组简单的接口，帮助你访问和控制引脚。主要接口包括：

- **获取引脚编号**：`rt_pin_get()`
- **设置引脚模式**：`rt_pin_mode()`
- **设置引脚电平**：`rt_pin_write()`
- **读取引脚电平**：`rt_pin_read()`
- **绑定中断回调函数**：`rt_pin_attach_irq()`
- **启用中断**：`rt_pin_irq_enable()`

如何获取引脚编号

RT-Thread 提供了一个独立的引脚编号系统，可以通过以下几种方式获取：

1. **通过 API 获取**：使用 `rt_pin_get("PF.9")` 获取 PF9 引脚的编号。
2. **通过宏定义获取**：使用 `GET_PIN(port, pin)` 宏定义获取编号，比如 `GET_PIN(F, 9)` 获取 PF9 的编号。
3. **查看驱动文件**：可以查看驱动文件 `drv_gpio.c` 来确认引脚编号。

如何设置引脚模式

你可以通过 `rt_pin_mode()` 函数来设置引脚的工作模式，比如输入、输出、上拉输入等。

示例：

```
rt_pin_mode(35, PIN_MODE_OUTPUT); // 设置引脚为输出模式
```

如何控制引脚电平

你可以使用 `rt_pin_write()` 来设置引脚输出的电平，`PIN_LOW` 表示低电平，`PIN_HIGH` 表示高电平。

示例：

```
rt_pin_write(35, PIN_HIGH); // 设置引脚输出高电平
```

如何读取引脚电平

你可以使用 `rt_pin_read()` 函数来读取引脚的电平状态，返回值为 `PIN_LOW` 或 `PIN_HIGH`。

示例：

```
int status = rt_pin_read(35); // 读取引脚电平
```

如何绑定引脚中断

你可以通过 `rt_pin_attach_irq()` 来为某个引脚绑定中断回调函数，当发生中断时，系统会调用该函数。

示例：

```
rt_pin_attach_irq(55, PIN_IRQ_MODE_FALLING, beep_on, RT_NULL); // 绑定中断
```

如何启用引脚中断

绑定好中断回调函数后，需要使用 `rt_pin_irq_enable()` 来启用中断。

示例：

```
rt_pin_irq_enable(55, PIN_IRQ_ENABLE); // 启用中断
```

> RT-Thread 的 PIN 管理框架是一套硬件抽象层（HAL）设计，用于统一管理和控制不同硬件平台的 GPIO 引脚。通过这个框架，RT-Thread 将底层的硬件差异屏蔽，提供统一的接口给上层应用使用。其核心思想是通过设备驱动模型，将引脚的功能抽象为设备，按照标准的驱动模型接口操作，增强了系统的可移植性和扩展性。
>
> ### 1. PIN 管理的抽象层设计
>
> PIN 设备的管理涉及对不同 MCU 的 GPIO 引脚进行配置和操作。由于不同芯片的 GPIO 控制寄存器和操作方式不尽相同，RT-Thread 通过**硬件抽象层（HAL）**实现了对这些差异的屏蔽。
>
> 核心抽象是通过定义一个 `rt_device_pin` 结构体，来代表每个芯片的 GPIO 设备。这种设计使得每个 MCU 的引脚控制都可以通过标准的接口进行操作，而不用关心底层硬件的差异。

2. rt_device_pin 结构体

在 RT-Thread 的 PIN 框架中，每个引脚（GPIO）都是作为一个设备来处理的，GPIO 的驱动也就作为一个标准的 RT-Thread 设备驱动存在。下面是 `rt_device_pin` 结构体的主要部分，它是 PIN 设备的核心数据结构：

```
struct rt_device_pin
{
 struct rt_device parent;              // 继承 rt_device 设备结构
 const struct rt_pin_ops *ops;         // 引脚操作接口
};
```

- `parent`：继承了 `rt_device` 结构体，这使得 PIN 设备符合 RT-Thread 的设备模型，具有标准设备的生命周期管理（打开、关闭等操作），符合设备驱动的抽象。
- `ops`：定义了操作引脚的接口函数指针，这是一套操作 GPIO 引脚的方法集合，类似于 Linux 设备模型中的文件操作集，定义了所有 PIN 设备的标准操作。

3. rt_pin_ops 操作接口

`rt_pin_ops` 是一组操作 GPIO 引脚的函数指针集合，不同的硬件平台实现不同的 GPIO 控制逻辑时，只需实现这些接口即可。它主要包含如下操作：

```
struct rt_pin_ops
{
    void (*pin_mode)(struct rt_device *device, rt_base_t pin, rt_base_t
mode);     // 设置引脚模式
    void (*pin_write)(struct rt_device *device, rt_base_t pin, rt_base_t
value);  // 设置引脚电平
    int (*pin_read)(struct rt_device *device, rt_base_t pin);
          // 读取引脚电平
    rt_err_t (*pin_attach_irq)(struct rt_device *device, rt_int32_t pin,
rt_uint32_t mode,
                               void (*hdr)(void *args), void *args);
         // 绑定引脚中断回调函数
    rt_err_t (*pin_detach_irq)(struct rt_device *device, rt_int32_t
pin);          // 解除引脚中断绑定
    rt_err_t (*pin_irq_enable)(struct rt_device *device, rt_base_t pin,
rt_uint32_t enabled); // 使能或禁用中断
};
```

这些接口函数允许 PIN 设备驱动支持通用的 GPIO 引脚功能，比如配置输入输出模式、读写引脚电平、绑定中断等。

- `pin_mode`：用于设置 GPIO 引脚的工作模式，比如输入、输出、上拉、下拉等。
- `pin_write`：用于控制 GPIO 输出高电平还是低电平。
- `pin_read`：用于读取 GPIO 的电平状态。
- `pin_attach_irq`：绑定一个中断回调函数，在引脚的某个中断条件下触发。
- `pin_detach_irq`：用于解除绑定的中断函数。
- `pin_irq_enable`：使能或禁用引脚中断。

每个硬件平台的 GPIO 驱动只需实现这些接口，将底层的硬件控制逻辑与上层的操作解耦。

4. 设备初始化

每个芯片平台的 GPIO 驱动程序（如 `drv_gpio.c`）都会在系统初始化阶段注册为一个标准设备。以 STM32 平台为例，它的 GPIO 驱动初始化时，通常会调用类似 `rt_device_pin_register()` 的函数，将 GPIO 驱动注册到 RT-Thread 的设备管理系统中。

设备注册的过程如下：

```
rt_err_t rt_device_pin_register(const char *name, const struct
rt_pin_ops *ops, void *user_data)
{
    struct rt_device_pin *pin;

    pin = rt_malloc(sizeof(struct rt_device_pin));
    if (pin == RT_NULL)
        return -RT_ENOMEM;

    pin->ops = ops;  // 绑定硬件操作集

    // 注册设备
    rt_device_register(&pin->parent, name, RT_DEVICE_FLAG_RDWR);

    return RT_EOK;
}
```

这个函数做了两件事：

1. 分配了 `rt_device_pin` 结构体实例，并将硬件操作接口 `ops` 绑定到该设备上。
2. 将设备注册到 RT-Thread 的设备管理框架中，这样上层应用可以通过设备名称进行访问。

5. 引脚操作流程

上层 API，如 `rt_pin_mode()` 、 `rt_pin_write()` 等接口，都是对底层 PIN 驱动的封装。它们通过查找设备，最终调用的是底层 `rt_pin_ops` 中的具体操作函数。

例如， `rt_pin_mode()` 的实现如下：

```
void rt_pin_mode(rt_base_t pin, rt_base_t mode)
{
    if (pin_dev.ops->pin_mode)
    {
        pin_dev.ops->pin_mode(RT_NULL, pin, mode);  // 调用底层驱动实现的
pin_mode 函数
    }
}
```

`rt_pin_mode()` 通过设备对象 `pin_dev` 调用了具体的 `pin_mode()` 函数，这个函数是由具体硬件平台实现的。比如 STM32 平台的 `pin_mode()` 会操作 STM32 的 GPIO 控制寄存器来设置引脚的工作模式。

类似的， `rt_pin_write()` 和 `rt_pin_read()` 也会调用 `pin_dev.ops->pin_write()` 和 `pin_dev.ops->pin_read()` 。

6. 中断管理

RT-Thread 的引脚中断功能也是通过 `rt_pin_attach_irq()` 和 `rt_pin_irq_enable()` 等接口实现的。这些接口负责为特定的引脚绑定中断回调函数，并在中断发生时调用。

```
rt_err_t rt_pin_attach_irq(rt_int32_t pin, rt_uint32_t mode, void (*hdr)
(void *args), void *args)
{
    if (pin_dev.ops->pin_attach_irq)
    {
        return pin_dev.ops->pin_attach_irq(RT_NULL, pin, mode, hdr,
args);
    }
    return -RT_ENOSYS;
}
```

该函数会调用具体硬件的 `pin_attach_irq` 操作，将中断回调函数 `hdr` 绑定到指定的引脚 `pin` 上。同时， `rt_pin_irq_enable()` 用于启用或禁用引脚中断。

7. 硬件平台的 GPIO 驱动实现

以 STM32 平台为例， `drv_gpio.c` 实现了对 STM32 GPIO 的具体操作。这些操作通过寄存器直接访问 GPIO 外设来控制引脚的模式、读写电平、中断绑定等。

以 `stm32_pin_mode()` 函数为例，它根据传入的 `mode` 参数配置 STM32 的 GPIO 控制寄存器：

```
static void stm32_pin_mode(rt_device_t dev, rt_base_t pin, rt_base_t
mode)
{
    // 解析 pin 对应的 GPIO 端口和引脚号
    GPIO_InitTypeDef GPIO_InitStruct;

    GPIO_InitStruct.Pin = stm32_pin_get_pin(pin);
    GPIO_InitStruct.Pull = GPIO_NOPULL;

    switch (mode)
    {
        case PIN_MODE_OUTPUT:
            GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
            break;
        case PIN_MODE_INPUT:
            GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
            break;
        // 其他模式配置省略
    }

    HAL_GPIO_Init(stm32_pin_get_port(pin), &GPIO_InitStruct);  // 初始化
GPIO
}
```

这里的 `stm32_pin_mode()` 操作 STM32 的 HAL 库函数 `HAL_GPIO_Init()` ，配置具体的 GPIO 模式。

8. 框架总结

RT-Thread 的 PIN 管理框架通过设备模型抽象了对 GPIO 引脚的控制。其设计思路主要体现在以下几点：

- **统一接口**：通过 `rt_pin_ops` 结构体，将所有的引脚操作接口标准化，不同硬件平台只需实现这些接口即可。
- **设备模型**：每个 GPIO 引脚被抽象为一个设备，符合 RT-Thread 设备管理框架，具备良好的可移植性和扩展性。

> - **硬件无关性**：上层应用只需调用统一的接口，不需要关心底层硬件平台的差异，增强了代码的通用性。

UART 设备

UART 简介

UART（Universal Asynchronous Receiver/Transmitter，通用异步收发传输器）是一种常见的异步串口通信协议，主要用于点对点数据传输。UART 工作时，将每个字符的数据一位一位地传输，实现简单、双向的串行通信。UART 需要的主要参数有：**波特率**、**起始位**、**数据位**、**停止位**、**奇偶校验位**。这些参数在通信双方必须匹配，才能实现正常的数据传输。

RT-Thread 串口设备操作接口

RT-Thread 提供了一组统一的接口，供应用程序访问 UART 硬件设备。常用的 UART 相关接口如下：

函数	描述
`rt_device_find()`	查找设备
`rt_device_open()`	打开设备
`rt_device_read()`	读取数据
`rt_device_write()`	写入数据
`rt_device_control()`	控制设备
`rt_device_set_rx_indicate()`	设置接收回调函数
`rt_device_set_tx_complete()`	设置发送完成回调函数
`rt_device_close()`	关闭设备

查找串口设备

应用程序可以通过设备名称获取 UART 设备句柄：

```
rt_device_t rt_device_find(const char* name);
```

- 参数 `name` 是设备名称，返回设备句柄或 `RT_NULL`（没有找到设备）。
- 示例：

```
#define SAMPLE_UART_NAME "uart2"
rt_device_t serial = rt_device_find(SAMPLE_UART_NAME);
```

打开串口设备

使用 `rt_device_open()` 函数打开设备：

```
rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags);
```

- 参数 `dev` 是设备句柄，`oflags` 是打开的设备模式标志。常见的 `oflags` 包括：
  - `RT_DEVICE_FLAG_INT_RX`：中断接收模式
  - `RT_DEVICE_FLAG_DMA_RX`：DMA 接收模式
  - `RT_DEVICE_FLAG_INT_TX`：中断发送模式
  - `RT_DEVICE_FLAG_DMA_TX`：DMA 发送模式

例如，以下代码以中断接收模式打开 UART 设备：

```
rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);
```

控制串口设备

通过 `rt_device_control()` 函数可以控制串口的配置：

```
rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg);
```

- 参数 `cmd` 可取值 `RT_DEVICE_CTRL_CONFIG`，用于配置 UART。
- `arg` 是控制的参数结构体 `serial_configure`，其成员包括波特率、数据位、校验位、缓冲区大小等。

UART 配置示例：

```
struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT;
config.baud_rate = BAUD_RATE_9600;
config.data_bits = DATA_BITS_8;
config.stop_bits = STOP_BITS_1;
config.bufsz = 128;
rt_device_control(serial, RT_DEVICE_CTRL_CONFIG, &config);
```

发送数据

调用 `rt_device_write()` 函数可以将数据写入到 UART 设备：

```
rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer,
rt_size_t size);
```

- 参数 `buffer` 是要发送的数据， `size` 是数据的大小。下面是一个发送字符串的例子：

```
char str[] = "hello RT-Thread!\r\n";
rt_device_write(serial, 0, str, sizeof(str) - 1);
```

接收数据

通过 `rt_device_read()` 函数读取 UART 接收到的数据：

```
rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t
size);
```

- `buffer` 是存放接收到数据的缓冲区， `size` 是要读取的数据大小。示例：

```
char buffer[64];
rt_device_read(serial, 0, buffer, sizeof(buffer));
```

设置接收回调函数

可以通过 `rt_device_set_rx_indicate()` 函数设置接收回调，当 UART 接收到数据时，会调用回调函数：

```
rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)
(rt_device_t dev, rt_size_t size));
```

- 参数 `rx_ind` 是接收数据的回调函数。

示例：设置回调函数，接收到数据后发送信号量唤醒线程：

```c
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    rt_sem_release(&rx_sem);  // 发送信号量
    return RT_EOK;
}
rt_device_set_rx_indicate(serial, uart_input);
```

关闭串口设备

当完成串口操作后，调用 `rt_device_close()` 函数关闭设备：

```c
rt_err_t rt_device_close(rt_device_t dev);
```

1. RT-Thread 串口框架总体设计

RT-Thread 的串口框架是基于设备模型的，它将底层的硬件控制逻辑抽象为统一的设备接口，并通过硬件抽象层（HAL）库与具体硬件进行对接。串口框架包括以下主要部分：

- **HAL 层（底层硬件抽象）**：直接操作硬件的驱动层，控制硬件串口的初始化、发送、接收等。
- **驱动层（串口驱动）**：对接 HAL 层，提供 RT-Thread 系统标准的串口操作接口（`rt_serial_device`）。
- **设备管理层**：RT-Thread 设备模型，用于统一管理设备，支持设备的注册、打开、关闭、读写等操作。
- **应用层**：用户应用程序通过 `rt_device_*` API 访问串口设备，无需关心底层的硬件实现。

2. HAL 库与 RT-Thread 串口驱动对接

RT-Thread 支持多种硬件平台，每个平台可能有不同的硬件抽象层（HAL）库。在 STM32 平台上，HAL 库是由 ST 提供的用于控制硬件外设的抽象库。在 RT-Thread 的串口驱动中，HAL 库负责底层的 UART 初始化、发送、接收和中断管理等操作。

2.1 HAL 库的串口初始化

串口的初始化通常在 HAL 库中由 `HAL_UART_Init()` 完成，该函数负责配置 UART 外设的寄存器。RT-Thread 的串口框架通过调用 HAL 库的初始化函数来完成硬件的配置。

示例（STM32 平台）：

```
static rt_err_t stm32_configure(struct rt_serial_device *serial, struct
serial_configure *cfg)
{
    UART_HandleTypeDef *huart = (UART_HandleTypeDef *)serial-
>parent.user_data;

    // 配置 HAL 库的 UART 初始化结构体
    huart->Init.BaudRate = cfg->baud_rate;
    huart->Init.WordLength = (cfg->data_bits == DATA_BITS_8) ?
UART_WORDLENGTH_8B : UART_WORDLENGTH_9B;
    huart->Init.StopBits = (cfg->stop_bits == STOP_BITS_1) ?
UART_STOPBITS_1 : UART_STOPBITS_2;
    huart->Init.Parity = (cfg->parity == PARITY_NONE) ? UART_PARITY_NONE
:
                         (cfg->parity == PARITY_ODD) ? UART_PARITY_ODD :
UART_PARITY_EVEN;
    huart->Init.Mode = UART_MODE_TX_RX;
    huart->Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart->Init.OverSampling = UART_OVERSAMPLING_16;

    // 调用 HAL 库的初始化函数，配置硬件
    if (HAL_UART_Init(huart) != HAL_OK)
    {
        return -RT_ERROR;
    }

    return RT_EOK;
}
```

在这个函数中：

- `serial->parent.user_data` 是用户传递的硬件相关数据，通常是 HAL 库的 UART 控制结构体（`UART_HandleTypeDef`）。
- 配置 UART 的参数（如波特率、数据位、停止位等）是通过 HAL 库中的 `UART_InitTypeDef` 结构体完成的。
- 调用 `HAL_UART_Init()` 完成底层硬件的初始化。

2.2 HAL 库的发送与接收

HAL 库提供了 UART 发送和接收函数，RT-Thread 的串口驱动通过这些 HAL 接口来完成串口的数据传输。

- **发送数据**：通过 HAL 库的 `HAL_UART_Transmit()` 函数完成。
- **接收数据**：通过 HAL 库的 `HAL_UART_Receive()` 函数完成。

在 RT-Thread 驱动中，这些函数被封装在串口操作接口中，例如 `putc` 和 `getc` 函数。示例（STM32 平台）：

```
// 发送一个字符
static int stm32_putc(struct rt_serial_device *serial, char c)
{
    UART_HandleTypeDef *huart = (UART_HandleTypeDef *)serial->parent.user_data;

    // 调用 HAL 库的发送函数
    if (HAL_UART_Transmit(huart, (uint8_t *)&c, 1, 1000) == HAL_OK)
    {
        return 1;
    }

    return -1;
}

// 接收一个字符
static int stm32_getc(struct rt_serial_device *serial)
{
    UART_HandleTypeDef *huart = (UART_HandleTypeDef *)serial->parent.user_data;
    uint8_t ch = 0;

    // 调用 HAL 库的接收函数
    if (HAL_UART_Receive(huart, &ch, 1, 0) == HAL_OK)
    {
        return ch;
    }

    return -1;
}
```

在 `putc` 函数中，字符通过 `HAL_UART_Transmit()` 发送到硬件串口。在 `getc` 函数中，使用 `HAL_UART_Receive()` 从硬件串口读取一个字符。

3. RT-Thread 串口驱动实现

RT-Thread 串口驱动层将 HAL 库的操作接口（如 `HAL_UART_Transmit()` 、 `HAL_UART_Receive()` ）封装到串口操作结构体 `struct rt_uart_ops` 中，并提供标准的设备接口供上层应用调用。

3.1 串口操作接口 struct rt_uart_ops

struct rt_uart_ops 是 RT-Thread 中串口驱动的核心，它定义了串口设备的基本操作，如配置、发送、接收、DMA 传输等。

```
struct rt_uart_ops
{
    rt_err_t (*configure)(struct rt_serial_device *serial, struct
serial_configure *cfg);  // 配置串口
    rt_err_t (*control)(struct rt_serial_device *serial, int cmd, void
*arg);              // 控制串口设备
    int (*putc)(struct rt_serial_device *serial, char c);
                    // 发送一个字符
    int (*getc)(struct rt_serial_device *serial);
                    // 接收一个字符
    rt_size_t (*dma_transmit)(struct rt_serial_device *serial,
rt_uint8_t *buf, rt_size_t size, int direction); // DMA 传输
};
```

- configure：配置串口设备（如波特率、数据位等），通常调用 HAL 的 HAL_UART_Init() 。
- putc：发送一个字符，通常通过 HAL 库的 HAL_UART_Transmit() 实现。
- getc：接收一个字符，通常通过 HAL 库的 HAL_UART_Receive() 实现。
- dma_transmit：通过 DMA 进行数据传输。

3.2 串口设备的注册

串口设备通过 rt_hw_serial_register() 函数注册到 RT-Thread 设备管理系统中。该函数将串口设备与操作接口绑定，并注册为 RT-Thread 的字符设备。

```
rt_err_t rt_hw_serial_register(struct rt_serial_device *serial,
                               const char *name,
                               rt_uint32_t flag,
                               void *data)
{
    struct rt_device *device;

    RT_ASSERT(serial != RT_NULL);
    RT_ASSERT(name != RT_NULL);

    device = &(serial->parent);  // 获取设备父类
    device->type = RT_Device_Class_Char;  // 设置设备类型为字符设备
```

```
    // 初始化设备操作接口
    device->rx_indicate = RT_NULL;
    device->tx_complete = RT_NULL;
    device->init    = RT_NULL;
    device->open    = rt_serial_open;
    device->close   = rt_serial_close;
    device->read    = rt_serial_read;
    device->write   = rt_serial_write;
    device->control = rt_serial_control;
    device->user_data = data;  // 存储串口硬件的用户数据（如 HAL 的
`UART_HandleTypeDef`）

    // 将设备注册到 RT-Thread 设备管理系统中
    return rt_device_register(device, name, flag);
}
```

- `serial`：串口设备的结构体（`struct rt_serial_device`），包含了串口设备的操作接口、配置信息等。
- `name`：设备的名称（如 `"uart1"`）。
- `flag`：设备的标志，指定使用中断、DMA 等模式。
- `data`：指向具体硬件的用户数据（如 HAL 库的 `UART_HandleTypeDef`），在操作时会传递给 HAL 函数。

4. 上层应用访问串口

上层应用通过 RT-Thread 提供的 `rt_device_*` API 访问串口设备。这些 API 封装了对底层硬件的操作，使得应用开发者无需关心底层硬件细节。

- **打开串口设备**：`rt_device_open()`
- **配置串口设备**：`rt_device_control()`
- **发送数据**：`rt_device_write()`
- **接收数据**：`rt_device_read()`
- **关闭串口设备**：`rt_device_close()`

示例：

```
#include <rtthread.h>
#include <rtdevice.h>

#define UART_NAME "uart2"   // 串口名称
```

```c
int main(void)
{
    rt_device_t serial;
    char str[] = "Hello RT-Thread!\n";
    char buffer[64];
    int length;

    // 查找并打开串口设备
    serial = rt_device_find(UART_NAME);
    rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);  // 打开串口，启用中断接收

    // 发送数据
    rt_device_write(serial, 0, str, sizeof(str));

    // 接收数据
    length = rt_device_read(serial, 0, buffer, sizeof(buffer));
    if (length > 0)
    {
        rt_kprintf("Received data: %s\n", buffer);
    }

    // 关闭串口设备
    rt_device_close(serial);

    return 0;
}
```

总结

RT-Thread 的串口框架通过与底层 HAL 库对接，提供了统一的串口操作接口。框架将 HAL 库的硬件操作（如初始化、发送、接收）封装到 `rt_uart_ops` 接口中，并通过设备模型管理串口设备。上层应用可以通过标准的 `rt_device_*` API 访问串口设备，完成数据收发和配置操作。

这一设计确保了 RT-Thread 串口框架具有良好的可移植性和扩展性，使得不同硬件平台可以通过简单的驱动层对接，实现统一的串口操作。

串口设备使用示例

中断接收及轮询发送

```
#include <rtthread.h>

#define SAMPLE_UART_NAME "uart1"

/* 信号量，用于接收数据通知 */
static struct rt_semaphore rx_sem;
static rt_device_t serial;

/* 接收回调函数 */
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    rt_sem_release(&rx_sem);  // 发送信号量
    return RT_EOK;
}

/* 处理串口接收到的数据 */
static void serial_thread_entry(void *parameter)
{
    char ch;
    while (1)
    {
        rt_sem_take(&rx_sem, RT_WAITING_FOREVER);  // 等待信号量
        while (rt_device_read(serial, -1, &ch, 1) == 1)
        {
            rt_device_write(serial, 0, &ch, 1);  // 回显接收到的数据
        }
    }
}

static int uart_sample(void)
{
    /* 查找并打开串口设备 */
    serial = rt_device_find(SAMPLE_UART_NAME);
    rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);

    /* 初始化信号量和接收回调 */
    rt_sem_init(&rx_sem, "rx_sem", 0, RT_IPC_FLAG_FIFO);
    rt_device_set_rx_indicate(serial, uart_input);

    /* 创建线程处理接收到的数据 */
```

```
    rt_thread_t thread = rt_thread_create("serial", serial_thread_entry,
RT_NULL, 1024, 25, 10);
    rt_thread_startup(thread);

    return 0;
}

INIT_APP_EXPORT(uart_sample);
```

`rt_sem_t` 和 `struct rt_semaphore` 都用于表示**信号量**，但它们的用法有所不同，主要在于它们的定义方式和应用场景。下面详细解释它们的区别：

1. rt_sem_t

- **定义**：`rt_sem_t` 是 RT-Thread 中用于表示**信号量指针**的类型，实际定义如下：

  ```
  typedef struct rt_semaphore* rt_sem_t;
  ```

  `rt_sem_t` 是指向 `struct rt_semaphore` 的指针。它用于在运行时通过动态方式创建和管理信号量。
- **用途**：`rt_sem_t` 主要用于动态分配的信号量。使用时，信号量的内存通过调用 `rt_sem_create()` 函数动态分配，使用完后需要调用 `rt_sem_delete()` 来释放内存。
- **动态信号量的生命周期**：
  - 动态创建的信号量在运行时分配内存，可以在需要时创建和销毁信号量，更加灵活，适合于那些**临时需要**的信号量。
- **示例**：

  ```
  rt_sem_t sem;

  /* 动态创建信号量，初始值为 0，FIFO 模式 */
  sem = rt_sem_create("sem", 0, RT_IPC_FLAG_FIFO);

  if (sem != RT_NULL)
  {
      /* 使用信号量 */
      rt_sem_take(sem, RT_WAITING_FOREVER);
      rt_sem_release(sem);

      /* 使用完后删除信号量 */
      rt_sem_delete(sem);
  ```

```
}
```

2. struct rt_semaphore

- **定义**：`struct rt_semaphore` 是 RT-Thread 中的一个结构体，表示 **信号量控制块**。它用于保存信号量的相关信息，如计数器、等待队列等。
- **用途**：`struct rt_semaphore` 通常用于静态定义信号量，即编译时分配内存，信号量的控制块在程序编译时就已经存在。在这种情况下，通常使用 `rt_sem_init()` 函数初始化信号量，而不需要手动释放。
- **静态信号量的生命周期**：
  - 静态信号量的内存分配是在编译时确定的，因此它的生命周期贯穿整个程序运行过程，适合那些 **长时间需要** 的信号量，通常无需销毁。
- **示例**：

```c
static struct rt_semaphore sem;  // 静态定义信号量

/* 初始化信号量，初始值为 0，FIFO 模式 */
rt_sem_init(&sem, "sem", 0, RT_IPC_FLAG_FIFO);

/* 使用信号量 */
rt_sem_take(&sem, RT_WAITING_FOREVER);
rt_sem_release(&sem);

/* 静态信号量无需删除，系统退出时会自动释放 */
```

区别总结

区别项	`rt_sem_t`（动态分配）	`struct rt_semaphore`（静态分配）
类型	指向 `struct rt_semaphore` 的指针类型	实际的 `struct rt_semaphore` 结构体
内存分配	动态分配，通过 `rt_sem_create()` 动态创建	静态分配，编译时分配内存，通过 `rt_sem_init()` 初始化
内存管理	需要在使用完后通过 `rt_sem_delete()` 释放内存	不需要手动释放，信号量的生命周期与程序一致
使用场景	适合需要动态创建和销毁信号量的场景	适合长期使用、生命周期贯穿整个程序的场景

区别项	`rt_sem_t`（动态分配）	`struct rt_semaphore`（静态分配）
灵活性	更加灵活，可以在需要时创建，使用完毕后销毁	不灵活，信号量始终存在，适用于稳定的、长期的资源管理
示例	`rt_sem_t sem = rt_sem_create(...);`	`static struct rt_semaphore sem;`

选择哪一个?

- 使用 `rt_sem_t`（动态信号量）：
  - 当需要 **灵活管理信号量的生命周期**，根据程序运行时的需求动态创建和销毁信号量时，使用 `rt_sem_t`。
  - 例如：当某个模块在运行时才需要某个信号量，或者当信号量在程序的某个阶段不再需要时，可以使用动态信号量。
- 使用 `struct rt_semaphore`（静态信号量）：
  - 当信号量的 **生命周期贯穿整个程序** 或者在程序启动时就确定需要信号量时，使用 `static struct rt_semaphore` 静态信号量。
  - 例如：用于全局资源同步的信号量，或者串口通信等场景中，信号量的生命周期可能与整个系统的生命周期一致。

结论

- `rt_sem_t`：指针类型，用于动态分配和销毁信号量，适合灵活、短时使用场景。
- `struct rt_semaphore`：静态定义的控制块，用于编译时分配信号量，适合固定且长期存在的信号量使用场景。

根据您的需求选择合适的信号量类型可以提高代码的灵活性和效率。

`INIT_APP_EXPORT` 是 RT-Thread 中的一个 **宏定义**，用于将指定的初始化函数注册到系统的初始化序列中，并指定该初始化函数的 **执行阶段**。它帮助开发者将自定义的初始化函数自动集成到 RT-Thread 启动时的初始化流程中。

宏定义的作用

`INIT_APP_EXPORT` 的作用是将一个函数标记为 **应用程序初始化函数**，并将其添加到 **应用程序初始化阶段** 的初始化表中。这个宏确保函数在操作系统启动的相应阶段被调用，从而实现模块的自动初始化。

宏定义的实际代码

`INIT_APP_EXPORT` 的实现通常如下：

```
#define INIT_APP_EXPORT(fn) INIT_EXPORT(fn, "3")
```

其中：

- `fn` 是要导出的初始化函数的名称。
- `INIT_EXPORT(fn, "3")` 是另一个宏，它用于将 `fn` 函数注册到系统的某个初始化阶段（这里的 `"3"` 代表初始化的优先级阶段为 3，应用程序初始化阶段）。

INIT_EXPORT 宏

`INIT_EXPORT` 宏是 RT-Thread 系统中用于将某个函数注册到 **初始化序列** 中的机制，它接受两个参数：要导出的函数和初始化阶段的优先级。例如， `INIT_EXPORT(fn, "3")` 会将 `fn` 函数注册到 **优先级为 3** 的初始化阶段。

不同的初始化阶段代表了不同的系统初始化阶段，每个阶段的初始化顺序如下：

- **0**：硬件相关初始化（ `INIT_BOARD_EXPORT` ）。
- **1**：设备驱动初始化阶段（ `INIT_DEVICE_EXPORT` ）。
- **2**：系统初始化阶段（ `INIT_COMPONENT_EXPORT` ）。
- **3**：应用程序初始化阶段（ `INIT_APP_EXPORT` ）。

使用场景

`INIT_APP_EXPORT` 主要用于那些需要在系统启动时自动调用的 **应用程序级初始化函数**，比如应用程序组件的初始化，驱动程序的注册等。这让开发者可以方便地控制初始化顺序，而不需要在 `main()` 函数中显式调用这些初始化函数。

使用示例

假设我们有一个需要在系统启动时自动执行的函数 `my_app_init` ，这个函数可以通过 `INIT_APP_EXPORT` 宏导出：

```
void my_app_init(void)
{
    // 应用程序的初始化逻辑
    rt_kprintf("Application initialization\n");
}

// 使用 INIT_APP_EXPORT 导出函数
INIT_APP_EXPORT(my_app_init);
```

在这个例子中， `my_app_init()` 函数将在系统启动时的应用程序初始化阶段被自动调用。这个函数的调用顺序将位于系统启动流程的较后阶段（即优先级为 3 的阶段），系统的其他组件（如硬件和设备驱动程序）在此阶段之前已经被初始化。

扩展：RT-Thread 初始化阶段

在 RT-Thread 中，初始化阶段按照以下几个宏进行划分：

1. `INIT_BOARD_EXPORT` ：硬件相关初始化函数，比如板级初始化。
2. `INIT_DEVICE_EXPORT` ：设备相关的初始化函数，比如注册设备驱动程序。
3. `INIT_COMPONENT_EXPORT` ：系统级组件的初始化，比如文件系统、网络协议栈等。
4. `INIT_APP_EXPORT` ：应用程序相关的初始化函数。

DMA 接收及轮询发送

```
#include <rtthread.h>

#define SAMPLE_UART_NAME "uart1"

static rt_device_t serial;
static struct rt_messagequeue rx_mq;

/* 串口接收消息结构 */
struct rx_msg
{
    rt_device_t dev;     // 设备句柄
    rt_size_t size;      // 接收到的数据大小
};


/* 接收回调函数 */
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)
```

```
{
    struct rx_msg msg;
    msg.dev = dev;
    msg.size = size;
    rt_mq_send(&rx_mq, &msg, sizeof(msg));
    return RT_EOK;
}

/* 串口线程处理接收到的数据 */
static void serial_thread_entry(void *parameter)
{
    struct rx_msg msg;
    char buffer[RT_SERIAL_RB_BUFSZ + 1];
    while (1)
    {
        rt_mq_recv(&rx_mq, &msg, sizeof(msg), RT_WAITING_FOREVER);
        rt_device_read(msg.dev, 0, buffer, msg.size);
        rt_device_write(serial, 0, buffer, msg.size);
    }
}

static int uart_dma_sample(void)
{
    static char msg_pool[256];

    /* 查找并打开串口设备 */
    serial = rt_device_find(SAMPLE_UART_NAME);
    rt_device_open(serial, RT_DEVICE_FLAG_DMA_RX);

    /* 初始化消息队列和接收回调 */
    rt_mq_init(&rx_mq, "rx_mq", msg_pool, sizeof(struct rx_msg),
sizeof(msg_pool), RT_IPC_FLAG_FIFO);
    rt_device_set_rx_indicate(serial, uart_input);

    /* 创建线程处理接收到的数据 */
    rt_thread_t thread = rt_thread_create("serial", serial_thread_entry,
RT_NULL, 1024, 25, 10);
    rt_thread_startup(thread);

    return 0;
}

INIT_APP_EXPORT(uart_dma_sample);
```

在 RT-Thread 中，`rt_mq_t` 是一个指向 `struct rt_messagequeue` 的指针类型，通常用于在动态创建消息队列时使用。而 `static struct rt_messagequeue` 是定义一个静态的消息队列控制块变量。这两种方式的主要区别在于 **内存管理** 和 **使用场景**，它们各有适用的场合。

区别

1. `rt_mq_t` （指针类型，动态创建）

- `rt_mq_t` 是指向 `struct rt_messagequeue` 的指针类型，通常用于 **动态分配内存** 的情况，即消息队列控制块和消息存储区都是通过 `rt_mq_create()` 动态创建。
- 当使用 `rt_mq_t` 时，你需要手动创建和销毁消息队列，适合那些 **需要灵活管理内存** 或 **消息队列的生命周期较短** 的情况。

使用示例：

```
rt_mq_t rx_mq;

/* 动态创建消息队列 */
rx_mq = rt_mq_create("rx_mq",              // 队列名称
                     sizeof(struct rx_msg), // 每条消息的大小
                     10,                   // 队列最大消息数
                     RT_IPC_FLAG_FIFO);    // FIFO模式

if (rx_mq == RT_NULL)
{
    rt_kprintf("Create message queue failed!\n");
}

/* 使用完后销毁消息队列 */
rt_mq_delete(rx_mq);
```

2. `static struct rt_messagequeue` （静态分配）

- `struct rt_messagequeue` 是一个实际的结构体变量，而不是指针。它在 **编译时静态分配内存**，并且其生命周期从程序启动到结束，适用于那些 **不需要动态内存管理**，或者 **消息队列的生命周期贯穿整个程序** 的情况。
- 使用静态消息队列时，通常使用 `rt_mq_init()` 进行初始化，不需要手动销毁，因为它的内存在编译时分配，不会被动态释放。

使用示例：

```c
static struct rt_messagequeue rx_mq;   // 静态消息队列控制块
static char msg_pool[256];             // 消息存储缓冲区

/* 初始化消息队列 */
rt_mq_init(&rx_mq,                     // 消息队列控制块
           "rx_mq",                    // 队列名称
           msg_pool,                   // 存储消息的缓冲区
           sizeof(struct rx_msg),      // 每条消息的大小
           sizeof(msg_pool),           // 缓冲区总大小
           RT_IPC_FLAG_FIFO);          // FIFO 模式

/* 使用完后无需手动销毁静态队列 */
```

什么时候使用 rt_mq_t 和 struct rt_messagequeue

- 使用 rt_mq_t （动态创建）：
  - 当你需要灵活管理消息队列的生命周期（例如：根据运行时的条件动态创建和销毁）。
  - 当你的系统中某些资源可能在某些时刻不再需要，因此可以通过动态释放内存来节省资源。
  - 比如设备热插拔场景下，设备插入时创建消息队列，设备拔出时删除消息队列。
- 使用 static struct rt_messagequeue （静态分配）：
  - 当你明确知道消息队列的生命周期贯穿整个程序的运行，无需动态管理其创建与销毁。
  - 当你希望避免动态内存分配带来的额外开销，或者系统不允许使用动态内存分配。
  - 比如一个系统中长期存在的、常用的模块（如串口收发、日志系统等）使用消息队列来管理数据。

环形缓冲区

1. Ringbuffer 简介

ringbuffer（环形缓冲区）是一种 **高效的 FIFO 数据结构**，可以实现无缝的循环读写操作。其主要特点是：

- 数据可以不断地填充到缓冲区中，不用担心缓冲区的边界。
- 通过读写指针和镜像值来区分缓冲区的状态（空、满）。

在嵌入式系统中，ringbuffer 常用于串口通信、数据流缓存等场景。

2. API 总结

RT-Thread 提供了一整套操作 ringbuffer 的 API，涵盖了创建、初始化、读写、销毁等操作。以下是 API 的分类和使用方法总结：

2.1 创建和销毁 ringbuffer

- **动态创建 ringbuffer**

```
struct rt_ringbuffer* rt_ringbuffer_create(rt_uint16_t length);
```

创建指定大小的动态缓冲区。适用于需要在运行时根据需求分配缓冲区的场景。

- **动态销毁 ringbuffer**

```
void rt_ringbuffer_destroy(struct rt_ringbuffer *rb);
```

释放动态分配的缓冲区。使用完毕后应销毁，避免内存泄漏。

2.2 初始化和复位 ringbuffer

- **静态初始化 ringbuffer**

```
void rt_ringbuffer_init(struct rt_ringbuffer *rb, rt_uint8_t *pool,
rt_int16_t size);
```

静态初始化，适合系统不支持动态内存分配，或希望使用静态内存的场景。需要预先分配缓冲区，并传入缓冲区指针和大小。

- **重置 ringbuffer**

```
void rt_ringbuffer_reset(struct rt_ringbuffer *rb);
```

将 ringbuffer 的状态重置为初始状态，清空其中的数据。

2.3 向 ringbuffer 写入数据

- **写入单个字节**

```
rt_size_t rt_ringbuffer_putchar(struct rt_ringbuffer *rb, const rt_uint8_t
ch);
```

向缓冲区写入一个字节，若缓冲区已满，写入失败。

- **强制写入单个字节（覆盖旧数据）**

```
rt_size_t rt_ringbuffer_putchar_force(struct rt_ringbuffer *rb, const
rt_uint8_t ch);
```

即使缓冲区已满，也会覆盖旧数据进行写入。

- **写入数据块**

```
rt_size_t rt_ringbuffer_put(struct rt_ringbuffer *rb, const rt_uint8_t *ptr,
rt_uint16_t length);
```

写入指定长度的数据块，若缓冲区已满，多余数据将被丢弃。

- **强制写入数据块（覆盖旧数据）**

```
rt_size_t rt_ringbuffer_put_force(struct rt_ringbuffer *rb, const rt_uint8_t
*ptr, rt_uint16_t length);
```

覆盖旧数据，保证所有数据都写入缓冲区。

2.4 从 ringbuffer 读取数据

- **读取单个字节**

```
rt_size_t rt_ringbuffer_getchar(struct rt_ringbuffer *rb, rt_uint8_t *ch);
```

从缓冲区读取一个字节，若缓冲区为空，读取失败。

- **读取数据块**

```
rt_size_t rt_ringbuffer_get(struct rt_ringbuffer *rb, rt_uint8_t *ptr,
rt_uint16_t length);
```

读取指定长度的数据块，若缓冲区内数据不足，则读取可用的数据。

- **查看数据但不取出数据（peek 操作）**

```
rt_size_t rt_ringbuffer_peek(struct rt_ringbuffer *rb, rt_uint8_t **ptr);
```

获取缓冲区内第一个可读数据的地址，但不取出数据。该操作用于查看数据但不改变缓冲区状态。

2.5 获取 ringbuffer 状态

- **获取缓冲区内存储的数据长度**

```
rt_size_t rt_ringbuffer_data_len(struct rt_ringbuffer *rb);
```

获取当前缓冲区内存储的数据长度。

- **获取 ringbuffer 大小**

```
rt_inline rt_uint16_t rt_ringbuffer_get_size(struct rt_ringbuffer *rb);
```

获取缓冲区的总大小。

3. Ringbuffer 开发方法

3.1 选择适当的内存管理方式

- **动态内存**：如果系统支持动态内存分配，建议使用 `rt_ringbuffer_create()` 动态创建缓冲区，避免静态分配导致的内存浪费。
- **静态内存**：如果内存较为紧张或者系统不支持动态分配，可以使用 `rt_ringbuffer_init()` 静态分配缓冲区。需要在代码中显式地分配一个静态数组，并将其传入 `rt_ringbuffer_init()` 。

3.2 多线程应用中的互斥问题

RT-Thread 的 ringbuffer 不具备内置的线程同步机制，因此在多线程（或多任务）环境中使用 ringbuffer 时，务必通过 **互斥锁** 或 **禁用中断** 的方式保护读写操作，避免数据竞争问题。

3.3 Peek 操作的注意事项

在使用 `rt_ringbuffer_peek()` 函数时，获取到的仅是缓冲区中第一个可读数据的地址。该接口适合用于访问单个字节，如果需要访问多个字节的数据，需结合其他读取方式，避免访问越界。

3.4 缓冲区满与数据覆盖

在某些情况下，环形缓冲区可能会出现满的情况。此时可以选择：

- **丢弃数据**：通过 `rt_ringbuffer_putchar()` 或 `rt_ringbuffer_put()` ，满时不写入数据。

- **覆盖旧数据**：通过 `rt_ringbuffer_putchar_force()` 或 `rt_ringbuffer_put_force()`，覆盖旧数据以保证最新的数据可以写入缓冲区。

3.5 数据存储和处理

在设计数据存储时，需要根据系统的内存和性能要求，合理选择 ringbuffer 的大小。在数据处理场景中，通常结合中断或定时器，进行定期的数据读取、处理和清空操作，保证数据流的顺畅传输。

4. 典型应用场景

- **串口通信**：在嵌入式系统中，串口通信往往是异步的，ringbuffer 适合作为串口接收数据的缓冲区，接收中断可将数据写入 ringbuffer，而主程序可定时读取缓冲区中的数据进行处理。
- **音频数据缓存**：对于音频处理应用，ringbuffer 可以作为音频流的数据缓冲区，实时接收音频输入并送给解码器。
- **数据采集**：在传感器数据采集的应用中，ringbuffer 可用于临时缓存高频采集的数据，避免数据丢失。

```
#include <uart_app.h>

#include <rtthread.h>
#include <rtdevice.h>
#include <string.h>
#include <ipc/ringbuffer.h>  // 包含 RT-Thread 的 ringbuffer 头文件
#include <stdio.h>           // 标准输入输出

#define BUFFER_SIZE 256                // 环形缓冲区大小
#define UART_DEVICE_NAME "uart1"       // 串口设备名称
#define DEBUG_MESSAGE "Uart Application Initialized.\r\n"
#define UART_TIMEOUT_TICKS 50          // 超时的系统tick数，具体值根据需求调整

/* 环形缓冲区存储池 */
static uint8_t rb_pool[BUFFER_SIZE];

/* 环形缓冲区对象 */
static struct rt_ringbuffer uart_rb;

/* 读取数据的缓冲区 */
static uint8_t uart_read_buffer[BUFFER_SIZE];

/* 用于同步的信号量 */
static struct rt_semaphore rx_sem;
```

```c
/* UART设备指针（用于重定向） */
static rt_device_t uart_dev_redirect = RT_NULL;

struct rt_device *uart_dev;

/* 记录上一次接收数据的tick */
static rt_tick_t last_rx_tick = 0;

int u1_printf(const char *format, ...)
{
 char buffer[512];  // 缓冲区大小，根据需求调整
 va_list args;
 va_start(args, format);

 // 使用 vsnprintf 格式化字符串到 buffer 中
 vsnprintf(buffer, sizeof(buffer), format, args);

 va_end(args);

 // 通过 rt_device_write 发送格式化后的字符串到串口1
 rt_device_write(uart_dev_redirect, 0, buffer, strlen(buffer));

 return strlen(buffer);
}

/* UART接收回调函数 */
static rt_err_t uart_rx_ind(rt_device_t dev, rt_size_t size)
{
 rt_size_t read_size;
 rt_uint8_t temp_buffer[BUFFER_SIZE];

 /* 复位计时变量 */
 last_rx_tick = rt_tick_get();

 /* 从串口设备读取数据 */
 read_size = rt_device_read(uart_dev, 0, temp_buffer, BUFFER_SIZE);

 if (read_size > 0)
 {
     /* 将读取的数据写入应用环形缓冲区 */
     size = rt_ringbuffer_put(&uart_rb, temp_buffer, read_size);
     if (size < read_size)
     {
         /* 如果环形缓冲区空间不足，丢弃多余的数据 */
         u1_printf("Ringbuffer full! Data lost.\n");
```

```
        }
    }
    return RT_EOK;
}

/* 数据处理线程入口函数 */
static void uart_thread_entry(void *parameter)
{
    rt_size_t size;
    rt_tick_t current_tick;

    while (1)
    {
        /* 获取当前的tick值 */
        current_tick = rt_tick_get();

        /* 判断是否已经超时 */
        if (current_tick - last_rx_tick ≥ UART_TIMEOUT_TICKS)
        {
            /* 如果超时了，认为数据已经接收完成，可以处理数据 */
            if (rt_ringbuffer_data_len(&uart_rb) > 0)
            {
                size = rt_ringbuffer_get(&uart_rb, uart_read_buffer,
sizeof(uart_read_buffer) - 1);
                if (size > 0)
                {
                    /* 发送调试信息 */
                    u1_printf("ringbuffer data: %s\r\n", uart_read_buffer);

                    /* 清空读取缓冲区 */
                    memset(uart_read_buffer, 0, sizeof(uart_read_buffer));
                }
            }
        }
        rt_thread_delay(50);
    }
}

/* 测试线程入口函数 */
static void test_thread_entry(void *parameter)
{
    while (1)
    {
        /* 打印测试信息 */
        u1_printf("system running\r\n");
```

```
        /* 延迟1秒（1000毫秒） */
        rt_thread_mdelay(1000);
    }
}

/* UART应用初始化函数 */
int uart_app_init(void)
{
    rt_err_t result;
    struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT;
    rt_thread_t thread,test_thread;

    /* 初始化信号量 */
    rt_sem_init(&rx_sem, "rx_sem", 0, RT_IPC_FLAG_FIFO);

    /* 初始化环形缓冲区 */
    rt_ringbuffer_init(&uart_rb, rb_pool, sizeof(rb_pool));

    /* 查找串口设备 */
    uart_dev = rt_device_find(UART_DEVICE_NAME);
    if (!uart_dev)
    {
        u1_printf("Find %s failed!\n", UART_DEVICE_NAME);
        return RT_ERROR;
    }

    /* 配置串口参数（可根据需要修改） */
    config.baud_rate = BAUD_RATE_115200; /* 波特率 */
    config.data_bits = DATA_BITS_8;      /* 数据位 */
    config.stop_bits = STOP_BITS_1;      /* 停止位 */
    config.parity    = PARITY_NONE;      /* 无奇偶校验 */
    config.bufsz     = BUFFER_SIZE;      /* 缓冲区大小 */

    /* 控制串口设备，设置配置参数 */
    result = rt_device_control(uart_dev, RT_DEVICE_CTRL_CONFIG, &config);
    if (result != RT_EOK)
    {
        u1_printf("Configure %s failed!\n", UART_DEVICE_NAME);
        return RT_ERROR;
    }

    /* 打开串口设备，以中断接收模式打开 */
    result = rt_device_open(uart_dev, RT_DEVICE_FLAG_INT_RX |
RT_DEVICE_FLAG_INT_TX);
    if (result != RT_EOK)
    {
```

```
        u1_printf("Open %s failed!\n", UART_DEVICE_NAME);
        return RT_ERROR;
    }

    /* 设置接收回调函数 */
    rt_device_set_rx_indicate(uart_dev, uart_rx_ind);

    /* 重定向输出到UART */
    uart_dev_redirect = uart_dev;

    /* 发送初始化消息 */
    u1_printf("%s", DEBUG_MESSAGE);

    /* 创建数据处理线程 */
    thread = rt_thread_create("uart_thread",
                              uart_thread_entry,
                              (void *)uart_dev,
                              4096,
                              RT_THREAD_PRIORITY_MAX / 2,
                              10);
    if (thread != RT_NULL)
    {
        rt_thread_startup(thread);
    }
    else
    {
        u1_printf("Create uart_thread failed!\n");
        return RT_ERROR;
    }

    /* 创建测试线程 */
    test_thread = rt_thread_create("test_thread",
                                   test_thread_entry,
                                   RT_NULL,
                                   2048,
                                   RT_THREAD_PRIORITY_MAX / 3,
                                   10);
    if (test_thread != RT_NULL)
    {
        rt_thread_startup(test_thread);
    }
    else
    {
        u1_printf("Create test_thread failed!\n");
        return RT_ERROR;
    }
```

```
    return RT_EOK;
}

/* 导出初始化函数到自动初始化列表中 */
INIT_APP_EXPORT(uart_app_init);
```

AT 组件

AT组件是RT-Thread系统中为使用AT命令控制设备提供的一种通信模块。它包含了客户端（AT Client）和服务器端（AT Server）两部分，实现了完整的命令发送、响应接收、数据解析等流程。通过AT组件，嵌入式设备可以控制外部模块，例如WiFi或蜂窝模块，实现网络连接和数据通信。

什么是AT命令?

AT命令是一种标准化的指令集，最早用于控制调制解调器，后来被引入到GSM等通信模块控制中。AT命令通常通过串口传输，设备可以向外部模块发送命令或接收数据。目前，许多嵌入式开发项目中仍使用AT命令来控制通信模块。

AT组件功能介绍

AT组件主要包含 `AT Server` 和 `AT Client` 功能：

- **AT Server**：负责接收和解析来自客户端的命令，并返回对应的响应。常用于设置或调试模式。
- **AT Client**：负责向服务器发送命令，并接收和解析响应，便于设备与外部模块通信。

AT Server主要特点

- **基础命令支持**：提供通用的基础命令（如ATE、ATZ等）；
- **命令兼容性**：支持大小写无关，提高兼容性；
- **调试模式**：提供CLI命令行模式，便于调试设备。

AT Client主要特点

- **数据解析**：支持自定义响应数据解析，便于获取信息；
- **URC数据处理**：完善的URC（非请求信息）处理机制；
- **多客户端支持**：可以同时连接多个外部设备。

AT Server 配置和初始化

AT Server配置

在RT-Thread系统中启用AT Server功能，需要在 `rtconfig.h` 中进行如下配置：

宏定义	描述
`RT_USING_AT`	开启AT组件
`AT_USING_SERVER`	启用AT Server功能
`AT_SERVER_DEVICE`	定义AT Server使用的串口名称（如 `uart3` ）

AT Server初始化

启用AT Server后，需调用 `at_server_init()` 完成初始化，系统会创建 `at_server` 线程以处理命令接收和数据解析。

```
int at_server_init(void);
```

自定义AT命令

AT Server支持基础命令如ATE、ATZ等，开发者可根据需求添加新的命令。自定义命令格式如下：

```
AT_CMD_EXPORT("AT+TEST", =<value1>[,<value2>], NULL, at_test_query, NULL,
at_test_exec);
```

- `AT+TEST` ： 命令名称。
- `=<value1>[,<value2>]` ： 参数格式， `<value1>` 必选， `<value2>` 可选。
- `NULL` ： 命令测试功能（未定义可用 `NULL` ）。
- `at_test_query` ： 查询功能函数。
- `at_test_exec` ： 执行功能函数。

示例代码：

```
static at_result_t at_test_exec(void)
{
    at_server_printfln("AT test command executed!");
    return 0;
}
AT_CMD_EXPORT("AT+TEST", =<value1>[,<value2>], NULL, NULL, NULL, at_test_exec);
```

AT Client 配置和初始化

在 `rtconfig.h` 中启用AT Client功能：

```
#define RT_USING_AT
#define AT_USING_CLIENT
#define AT_CLIENT_NUM_MAX 1
```

AT Client初始化

调用 `at_client_init()` 进行初始化，完成数据接收与URC数据处理等功能的准备。

```
int at_client_init(const char *dev_name, rt_size_t recv_bufsz);
```

AT Client 数据收发和解析

AT Client主要负责命令的发送、响应数据接收和解析。关键结构体为 `at_response` ：

```
struct at_response
{
    char *buf;           // 接收数据的缓冲区
    rt_size_t buf_size;    // 缓冲区大小
    rt_size_t line_num;    // 接收行数限制
    rt_int32_t timeout;    // 响应超时时间
};
```

常用API接口

- **发送命令并接收响应**

```
rt_err_t at_exec_cmd(at_response_t resp, const char *cmd_expr, ...);
```

- **解析指定行数据**

```
int at_resp_parse_line_args(at_response_t resp, rt_size_t resp_line, const char
*resp_expr, ...);
```

- **自定义响应结构体创建与删除**

```
at_response_t at_create_resp(rt_size_t buf_size, rt_size_t line_num, rt_int32_t
timeout);
void at_delete_resp(at_response_t resp);
```

数据解析示例

```
/* 解析串口设置 */
at_resp_parse_line_args(resp, 1,"%*[^=]=%d,%d,%d,%d,%d", &baudrate, &databits,
&stopbits, &parity, &control);
```

AT Client URC（非请求信息）处理

URC是指AT Server主动向AT Client发送的数据。URC数据处理通过结构体实现：

```
struct at_urc
{
    const char *cmd_prefix;
    const char *cmd_suffix;
    void (*func)(struct at_client *client, const char *data, rt_size_t size);
};
```

配置URC数据列表

在AT Client初始化中，通过 `at_set_urc_table()` 函数配置URC数据处理列表。示例：

```
static struct at_urc urc_table[] = {
    {"WIFI CONNECTED", "\r\n", urc_conn_func},
    {"+RECV", ":", urc_recv_func},
};

at_set_urc_table(urc_table, sizeof(urc_table) / sizeof(urc_table[0]));
```

多客户端支持

AT组件允许同时连接多个外部模块，多客户端模式提供独立的API，如：

- **单客户端模式**：默认使用第一个初始化的客户端。
- **多客户端模式**：支持动态获取和指定客户端对象。

使用多客户端示例

```
/* 初始化两个客户端 */
at_client_init("uart2", 512);
at_client_init("uart3", 512);

/* 获取并使用客户端对象 */
at_client_t client = at_client_get("uart3");
at_obj_exec_cmd(client, resp, "AT+CIFSR");
```

```
#include <at_app.h>

#include <at.h>

#define AT_CLIENT_UART_NAME "uart2"   // 使用串口2
#define AT_RESP_BUFF_SIZE   2048      // 响应缓冲区大小
#define WIFI_SSID           "rtthread"        // WiFi名称
#define WIFI_PASSWORD       "12345678"    // WiFi密码

int scan_wifi_esp8266(void)
{
 at_client_t client = RT_NULL;
 at_response_t resp = RT_NULL;
 rt_err_t result;

 // 初始化串口2上的AT Client
 result = at_client_init(AT_CLIENT_UART_NAME, AT_RESP_BUFF_SIZE);
 if (result != RT_EOK)
 {
     rt_kprintf("AT client init failed!\n");
     return -1;
 }

 // 获取AT Client对象
 client = at_client_get(AT_CLIENT_UART_NAME);
 if (client == RT_NULL)
```

```
    {
        rt_kprintf("Get AT client failed!\n");
        return -1;
    }

    // 创建一个响应对象，用于接收ESP8266返回的数据
    resp = at_create_resp(AT_RESP_BUFF_SIZE, 0,
rt_tick_from_millisecond(10000));
    if (resp == RT_NULL)
    {
        rt_kprintf("No memory for response structure!\n");
        return -1;
    }

    // 发送AT指令“AT+CWLAP”，扫描周围WiFi网络
    if (at_obj_exec_cmd(client, resp, "AT+CWLAP") != RT_EOK)
    {
        rt_kprintf("Failed to execute command AT+CWLAP\n");
        at_delete_resp(resp);
        return -1;
    }

    // 按行获取并打印WiFi信息
    for (int i = 1; i <= 15; i++)
    {
        const char *line = at_resp_get_line(resp, i);
        if (line != RT_NULL)
        {
            rt_kprintf("WiFi Network: %s\n", line);
        }
        else
        {
            // 如果行为空，表示已经没有更多WiFi数据
            break;
        }
    }

    // 发送AT指令“AT+CWJAP="SSID","PASSWORD"”，连接到WiFi网络
    if (at_obj_exec_cmd(client, resp, "AT+CWJAP=\"%s\",\"%s\"", WIFI_SSID,
WIFI_PASSWORD) != RT_EOK)
    {
        rt_kprintf("Failed to execute command AT+CWJAP\n");
        at_delete_resp(resp);
        return -1;
    }
```

```
    // 检查是否连接成功
    const char *line = at_resp_get_line(resp, 1);
    if (line != RT_NULL)
    {
        rt_kprintf("ESP8266 Response: %s\n", line);
    }

    // 删除响应对象
    at_delete_resp(resp);

    return 0;
}
```

ESP8266 AT 设备库

ESP8266 AT 设备库为 RT-Thread 提供了与 ESP8266 WiFi 模块进行通信的接口，通过 AT 指令实现 WiFi 连接、网络配置和 Socket 操作。该库支持客户端和服务器模式，包含网络接口操作、设备初始化、控制和 Socket 管理等功能模块。

主要模块概览

1. **设备初始化与配置**（`at_device_esp8266.c`）
2. **网络接口操作**（`at_device_esp8266.c`）
3. **Socket 管理**（`at_socket_esp8266.c`）
4. **头文件定义**（`at_device_esp8266.h`）

设备初始化与配置

设备初始化模块提供了 ESP8266 的网络设备注册、初始化以及 WiFi 配置操作。设备通过串口发送 AT 指令完成 WiFi 连接和网络配置：

- **初始化函数**：`esp8266_init()` 初始化设备，包括设置工作模式、关闭回显、配置 WiFi 模式等。
- **连接 WiFi**：通过 `AT+CWJAP="SSID","PASSWORD"` 指令连接到指定 WiFi。
- **设备重置**：使用 `AT+RST` 指令复位设备。
- **版本号获取**：`esp8266_at_version_to_hex` 和 `esp8266_get_at_version` 转换并获取 AT 固件版本

网络接口操作

此模块封装了网络接口的操作接口，设置 IP 地址、DNS 服务器和 DHCP 状态。主要功能：

- **IP 地址配置**：`esp8266_netdev_set_addr_info` 通过 AT 指令 `AT+CIPSTA` 设置 IP、网关和子网掩码。
- **DNS 配置**：`esp8266_netdev_set_dns_server` 设置主备 DNS 服务器。
- **DHCP 配置**：`esp8266_netdev_set_dhcp` 控制 DHCP 开启和关闭

Socket 管理

该模块通过 AT 指令实现 TCP/UDP 连接、发送、接收等 Socket 操作。支持客户端和服务器模式，主要功能包括：

- **连接 Socket**：`esp8266_socket_connect` 使用 `AT+CIPSTART` 指令创建 TCP/UDP 连接。
- **数据发送**：`esp8266_socket_send` 通过 `AT+CIPSEND` 发送数据，并支持自动分包，确保符合模块的最大数据包限制。
- **Socket 关闭**：`esp8266_socket_close` 通过 `AT+CIPCLOSE` 关闭连接。
- **域名解析**：`esp8266_domain_resolve` 使用 `AT+CIPDOMAIN` 获取域名对应的 IP 地址

> **Socket** 是一种在网络上进行数据通信的"插座"或"接口"。它的作用就像一条数据传输的通道，允许两台计算机或设备之间发送和接收数据。我们可以把它想象成电话系统中的一个电话插座，两台设备通过"插入"Socket来连接，之后就可以互相传递信息了。
>
> 1. **连接双方**：Socket在一台设备上打开一个"接口"，另一台设备可以通过这个接口与它通信。这个连接可以是本地的，也可以是远程的，比如互联网上的计算机之间的通信。
> 2. **端口和IP**：每个Socket都有一个地址（IP地址）和端口号，IP地址相当于设备的地址，端口号就像某种特定的"房间号"，让数据知道发送到设备的哪个应用程序。
> 3. **双向通信**：通过Socket，设备可以接收和发送数据，这种双向交流就像电话双方都可以说话和听到对方一样。
>
> 比如，我们的计算机和一台远程服务器想进行数据交换：
>
> - **客户端**（你的计算机）通过Socket连接到服务器的IP和端口。
> - **服务器**接受客户端的连接请求，两者之间建立起了连接。
> - **数据传输**开始，客户端和服务器通过Socket接口相互传输数据。

事件和回调处理

库中定义了多个 URC（Unsolicited Result Code）回调，用于处理 ESP8266 主动推送的信息，如连接成功、连接断开、数据接收等：

- **WiFi 连接事件**：通过 URC 检测 `WIFI CONNECTED` 和 `WIFI DISCONNECT` 事件。
- **Socket 事件**：如 `SEND OK`、`RECV`、`CLOSED` 等，分别表示发送完成、接收数据和连接关闭。
- **自定义回调**：可在 `esp8266_socket_set_event_cb` 中为不同事件（如连接、关闭、接收等）注册自定义回调函数。

配置与初始化

库的配置项及初始化步骤：

1. **在 `rtconfig.h` 中开启配置**：

```
#define AT_DEVICE_USING_ESP8266
#define AT_USING_SOCKET
```

2. **注册设备类**：调用 `esp8266_device_class_register`，初始化设备并添加到网络接口列表中。
3. **设置 WiFi 信息**：调用 `esp8266_wifi_info_set` 函数，配置 SSID 和密码并连接

> 使用ESP8266库相较于直接使用RT-Thread的AT库来手动编写AT指令，有几个显著的优势。这些优势主要体现在**开发便捷性**、**可靠性**和**功能扩展**等方面，简化了操作，提高了代码的可读性和可维护性。以下是具体的优势：
>
> ### 1. 封装复杂的AT指令流程
>
> ESP8266库将常用的AT指令和响应解析过程进行了封装，简化了操作。例如：
>
> - **WiFi连接**：只需调用 `esp8266_wifi_info_set` 接口，库会自动处理AT指令的发送、响应接收、错误检查和重试等步骤，而不用手动编写多个AT指令（如 `AT+CWJAP`、`AT+CIPSTA` 等）和解析响应的逻辑。
> - **Socket通信**：库封装了Socket的创建、发送、接收和关闭等操作，比如 `esp8266_socket_connect`、`esp8266_socket_send` 等函数可以直接调用，减少了管理Socket状态的复杂性。

2. 内置事件和错误处理机制

ESP8266库集成了URC（Unsolicited Result Code）机制，自动处理ESP8266主动推送的事件。例如：

- **WiFi连接/断开**：ESP8266的库注册了URC回调函数，自动处理 `WIFI CONNECTED` 和 `WIFI DISCONNECT` 事件，无需开发者手动检测。
- **Socket事件处理**：库内置Socket事件回调机制，如接收到数据时会自动触发 `urc_recv_func` 回调。这样，应用层代码无需关注底层的AT响应，只需处理应用逻辑。
- **错误重试**：对于Socket连接失败、数据发送失败等情况，库内置了自动重试机制，避免了开发者手动处理错误。

3. 提高开发效率，代码更简洁易读

ESP8266库对常用的操作提供了高层API，开发者只需调用接口即可完成WiFi连接、网络配置、Socket通信等操作。例如：

- 直接调用 `esp8266_netdev_set_dns_server` 即可设置DNS服务器，无需手动拼接 `AT+CIPDNS` 命令。
- `esp8266_socket_send` 封装了分包发送和事件等待，使得代码更简洁、易于阅读。

4. 统一的Socket接口支持网络通信

ESP8266库对接了RT-Thread的Socket接口，使其可以通过标准Socket API访问网络。这意味着：

- **接口一致性**：开发者可以使用类似BSD Socket的接口进行通信，如 `connect` 、 `send` 、 `recv` 等操作，降低了学习成本。
- **可扩展性**：应用层代码可以复用在其他支持Socket的网络模块上，而不需要针对不同的模块重新编写AT指令。

5. 支持多客户端和服务器模式

ESP8266库支持多个Socket客户端以及服务器模式，可以方便地在多个设备之间建立连接。

- **多Socket支持**：通过配置 `AT_DEVICE_ESP8266_SOCKETS_NUM` ，ESP8266库可以同时管理多个Socket连接，使得同一个模块可以同时连接多个服务器或客户端。
- **服务器模式**：库中支持的服务器模式允许ESP8266作为服务端接收连接请求，适合本地控制等场景，无需开发者手动管理连接和监听的细节。

6. 提供更好的兼容性和维护性

ESP8266库对不同AT指令版本的兼容进行了处理，使得库可以在不同版本的ESP8266固件上工作。

- **版本兼容性**：库内置了AT指令版本的检测机制，如通过 `esp8266_at_version_to_hex` 来判断固件版本，确保在不同固件版本下调用合适的命令和处理响应。
- **维护性**：由于ESP8266库的代码已标准化，具有更高的可读性和一致性，RT-Thread开发团队会维护和更新库，以支持新的功能和AT指令版本，用户可以直接升级库来获得改进。

API接口列表

1. 设备初始化与控制

- `int esp8266_init(struct at_device *device);`
  - 初始化ESP8266设备，包括配置AT客户端、注册URC处理程序、设置WiFi工作模式等。
- `int esp8266_reset(struct at_device *device);`
  - 复位ESP8266设备，通过发送 `AT+RST` 命令复位模块并重新初始化网络。
- `int esp8266_wifi_info_set(struct at_device *device, struct at_device_ssid_pwd *info);`
  - 设置WiFi网络的SSID和密码，并尝试连接到指定网络。
- `unsigned int esp8266_get_at_version(void);`
  - 获取ESP8266 AT指令的版本信息，返回值为十六进制格式的版本号。

---

2. 网络配置与状态查询

- `int esp8266_netdev_set_addr_info(struct netdev *netdev, ip_addr_t *ip_addr, ip_addr_t *netmask, ip_addr_t *gw);`
  - 配置IP地址、网关和子网掩码，使用 `AT+CIPSTA` 命令。
- `int esp8266_netdev_set_dns_server(struct netdev *netdev, uint8_t dns_num, ip_addr_t *dns_server);`
  - 设置DNS服务器的IP地址，可配置主DNS和备选DNS。
- `int esp8266_netdev_set_dhcp(struct netdev *netdev, rt_bool_t is_enabled);`
  - 启用或禁用DHCP功能，使用 `AT+CWDHCP` 命令。

- `int esp8266_netdev_ping(struct netdev *netdev, const char *host, size_t data_len, uint32_t timeout, struct netdev_ping_resp *ping_resp);`
  - 发送Ping请求到指定主机，并返回Ping响应信息，包括IP地址和响应时间。
- `void esp8266_netdev_netstat(struct netdev *netdev);`
  - 查询并打印当前网络连接状态，使用 `AT+CIPSTATUS` 命令获取连接信息。

---

3. Socket操作（网络通信）

- `int esp8266_socket_connect(struct at_socket *socket, char *ip, int32_t port, enum at_socket_type type, rt_bool_t is_client);`
  - 建立TCP或UDP连接，作为客户端连接到指定的IP和端口。
- `int esp8266_socket_close(struct at_socket *socket);`
  - 关闭指定的Socket连接，通过 `AT+CIPCLOSE` 指令断开连接。
- `int esp8266_socket_send(struct at_socket *socket, const char *buff, size_t bfsz, enum at_socket_type type);`
  - 发送数据到指定的Socket连接，支持TCP和UDP，使用 `AT+CIPSEND` 命令。
- `int esp8266_domain_resolve(const char *name, char ip[16]);`
  - 域名解析，将域名转换为IP地址，使用 `AT+CIPDOMAIN` 命令。
- `void esp8266_socket_set_event_cb(at_socket_evt_t event, at_evt_cb_t cb);`
  - 为Socket事件注册回调函数，支持接收、连接、关闭等事件通知。

---

4. 其他辅助功能

- `unsigned int esp8266_at_version_to_hex(const char *str);`
  - 将ESP8266 AT版本号字符串转换为十六进制数，便于进行版本比较和显示。

> ## 1. 什么是Socket?
>
> 在这里，**Socket**就是ESP8266模块和外部网络（如服务器、手机或其他设备）之间建立的通信通道。它让ESP8266可以通过网络传输数据，实现与其他设备的交互。Socket可以支持两种类型的通信：
>
> - **TCP**：一种可靠的连接方式，适合长时间的通信，确保数据按顺序到达，比如用于聊天、文件传输。

- **UDP**：一种不可靠的、轻量的方式，适合快速、短暂的通信，不保证数据一定到达，比如用于发送传感器数据。

---

2. 客户端和服务端的概念

在网络通信中，**客户端**和**服务端**指的是两种不同的角色。以Socket连接为例：

- **客户端（Client）**：主动发起连接的一方，通常是想获取数据或发送请求的设备。
- **服务端（Server）**：被动等待连接请求的一方，通常提供数据或服务。

在ESP8266应用场景中，ESP8266可以充当**客户端**或**服务端**，取决于你的应用需求。

---

3. ESP8266作为客户端的应用场景

当ESP8266作为客户端时，它会主动连接远程的服务器，发送或接收数据。例如：

- **物联网数据上传**：ESP8266连接到一个云服务器，并定期上传传感器数据。ESP8266主动发起连接请求，所以它是客户端，而云服务器是服务端。
- **访问网络服务**：比如通过Socket连接到一个网站的API，ESP8266作为客户端发送请求，获取数据。

工作流程：

1. ESP8266发起Socket连接请求（如 `AT+CIPSTART="TCP","服务器IP",端口` ）。
2. 连接成功后，ESP8266可以通过Socket发送数据到服务器（如传感器数据）。
3. 服务器接收到数据并返回响应，ESP8266读取响应内容。

---

4. ESP8266作为服务端的应用场景

当ESP8266作为服务端时，它会在本地监听一个端口，等待其他设备（如手机、计算机）连接到它。例如：

- **本地控制**：用手机App控制ESP8266上的设备，比如通过Socket连接发送开关指令。
- **数据查询**：外部设备连接到ESP8266并请求数据，ESP8266将采集到的传感器数据返回。

工作流程：

1. ESP8266设置为服务端，并在指定端口上监听连接请求（如 `AT+CIPSERVER=1,端口` ）。
2. 当其他设备（如手机）发起连接请求时，ESP8266接受连接并建立Socket通信。

3. 外部设备发送指令或请求，ESP8266收到后进行处理，并返回结果。

总结

- **Socket** 是通信通道，让ESP8266与外部网络设备建立连接并传输数据。
- **客户端** 是主动发起连接的一方，通常用于从服务器获取数据或上传数据。
- **服务端** 是被动等待连接的一方，通常用于接收来自客户端的请求并返回

1. 注册ESP8266设备类

- `esp8266_device_class_register` 会创建一个 `at_device_class` 结构体实例，该结构体包含 ESP8266 的设备类信息，并初始化设备操作和Socket操作的回调指针。
- 该函数调用 `at_device_class_register` 将此类注册到 RT-Thread 的 AT 设备框架中，这一步完成后，ESP8266设备类已成功注册，RT-Thread的AT框架可以识别和管理ESP8266设备。

2. 配置设备操作函数

- `esp8266_device_class_register` 中设置的 `esp8266_device_ops` 是一组操作接口，包括初始化（init）、反初始化（deinit）和控制（control）等，具体包括：
  - `esp8266_init` ：初始化ESP8266设备。
  - `esp8266_deinit` ：反初始化设备，关闭网络连接。
  - `esp8266_control` ：提供对设备的控制操作，例如复位、WiFi配置、重连等。
- 当需要执行具体的设备操作时，这些操作接口会被调用，从而确保设备按需初始化、控制或关闭。

3. 配置Socket操作函数

如果使用了Socket功能（通过 `#define AT_USING_SOCKET` ），会注册ESP8266设备的Socket操作函数：

- **Socket操作函数**：通过 `esp8266_socket_class_register` 注册，具体包括：
  - **连接/关闭**： `esp8266_socket_connect` 和 `esp8266_socket_close` 分别用于建立和关闭TCP/UDP连接。

- **数据传输**： `esp8266_socket_send` 用于发送数据包， `esp8266_domain_resolve` 用于域名解析。
- **事件回调**：通过 `esp8266_socket_set_event_cb` 注册Socket事件回调，处理连接、关闭、接收等事件。

4. AT客户端和URC处理注册

- **AT客户端初始化**：在ESP8266初始化时，会调用 `at_client_init` 创建AT客户端，并将客户端与指定的串口（如uart2）关联，供AT命令通信使用。
- **URC（Unsolicited Result Code）表注册**：通过 `at_obj_set_urc_table` 注册URC回调表，配置ESP8266的URC事件处理。例如：
  - `WIFI CONNECTED` 和 `WIFI DISCONNECT` 事件分别触发WiFi连接和断开的处理逻辑。
  - Socket相关的URC事件（如 `+IPD` 表示数据接收、 `SEND OK` 表示发送成功）会调用相应回调，进行后续处理。

5. 设备注册到网络接口

- **设备添加到网络接口**：ESP8266通过 `esp8266_netdev_add` 添加到RT-Thread的网络接口中，使其成为一个可识别的网络设备。
- **网络接口设置**：ESP8266设备会自动与RT-Thread的网络栈对接，实现IP地址设置、DNS配置、Ping检测等操作。

6. 执行初始化线程

- 在 `esp8266_net_init` 中，创建并启动设备初始化线程 `esp8266_init_thread_entry` 。此线程执行实际的AT命令初始化流程，包括：
  - 复位设备
  - 关闭AT指令回显
  - 配置工作模式
  - 连接到指定WiFi
  - 查询和配置模块的网络参数
-

传感器框架

1. 传感器框架的设计目标

传感器框架的设计目的是为了简化开发人员对各种传感器的操作，使上层应用可以通过统一的接口来访问不同厂商的传感器，提高代码的复用性。框架提供了标准化的接口和数据格式，便于开发者使用和扩展。

2. 传感器设备的接口

传感器框架为传感器设备提供了标准的设备接口，如打开、关闭、读取和控制等操作，这些接口符合 RT-Thread I/O 设备管理接口规范：

函数	描述
`rt_device_find()`	根据传感器设备名称查找设备并获取句柄
`rt_device_open()`	打开传感器设备
`rt_device_read()`	读取传感器数据
`rt_device_control()`	控制传感器设备
`rt_device_set_rx_indicate()`	设置接收回调函数
`rt_device_close()`	关闭传感器设备

这些接口实现了传感器的基本操作，能够适用于不同类型的传感器。

3. 工作模式

RT-Thread 传感器框架支持多种工作模式，以适应不同的应用需求：

- **轮询模式**：应用程序周期性地访问传感器设备，获取传感器数据。此模式适用于对数据采集频率要求不高的场景。
- **中断模式**：传感器设备在数据准备好后，通过中断通知上层应用。此模式适用于数据变化较频繁的场景。
- **FIFO 模式**：传感器支持数据存储在硬件的 FIFO 中，系统一次性读取多个数据，这样可以降低系统的采集频率，从而节省 CPU 资源。此模式特别适用于低功耗应用。

在编程时，可以通过设备打开标志（`oflags`）来选择传感器的工作模式。示例如下：

```
rt_device_open(dev, RT_DEVICE_FLAG_RDONLY);  // 以轮询模式打开设备
rt_device_open(dev, RT_DEVICE_FLAG_INT_RX);  // 以中断模式打开设备
rt_device_open(dev, RT_DEVICE_FLAG_FIFO_RX); // 以 FIFO 模式打开设备
```

4. 电源管理模式

传感器框架支持不同的电源管理模式，使传感器可以在高性能和低功耗之间切换：

- **掉电模式**：关闭传感器电源，以节省能量。
- **普通模式**：传感器以常规功率运行。
- **低功耗模式**：适合低功率场景，传感器在此模式下会降低数据更新频率。
- **高性能模式**：传感器以最高功率运行，以达到最快的数据更新速度。

可以通过 `rt_device_control()` 设置电源模式，例如：

```
rt_device_control(dev, RT_SENSOR_CTRL_SET_POWER, (void *)RT_SEN_POWER_HIGH);
```

5. 数据读取

传感器框架允许通过 `rt_device_read()` 函数获取传感器数据。读取的数据会以 `rt_sensor_data` 结构体返回，包含具体的数值和时间戳，方便后续处理。

数据读取示例

以下是读取传感器数据的示例代码：

```
struct rt_sensor_data data;
if (rt_device_read(dev, 0, &data, 1) == 1)
{
    rt_kprintf("Temperature: %d.%d°C\n", data.data.temp / 10, data.data.temp %
10);
}
```

6. 控制接口

框架提供了丰富的控制命令，通过 `rt_device_control()` 接口可以配置传感器的工作参数，如数据输出速率、测量范围等。常见的控制命令如下：

命令	功能
`RT_SENSOR_CTRL_GET_ID`	获取设备 ID
`RT_SENSOR_CTRL_GET_INFO`	获取设备信息
`RT_SENSOR_CTRL_SET_RANGE`	设置测量范围
`RT_SENSOR_CTRL_SET_ODR`	设置数据输出速率
`RT_SENSOR_CTRL_SET_POWER`	设置电源模式

命令	功能
`RT_SENSOR_CTRL_SELF_TEST`	设备自检

7. 回调函数设置

在中断模式下，传感器数据接收完成后，框架允许通过设置回调函数来通知上层应用。例如，当传感器数据到达时，回调函数可以发出信号量来通知处理线程：

```
rt_device_set_rx_indicate(dev, sensor_input); // 设置接收回调函数
```

在回调函数中，可以通过信号量或事件来实现数据的进一步处理。

8. 传感器数据结构

传感器框架的数据结构定义在 `rt_sensor_data` 中，包括测量数据、时间戳等信息，便于统一管理。具体如下：

```
struct rt_sensor_data
{
    union
    {
        struct { rt_int32_t x, y, z; } acce; /* 加速度 */
        struct { rt_int32_t x, y, z; } gyro; /* 陀螺仪 */
        struct { rt_int32_t x, y, z; } mag;  /* 磁力计 */
        rt_int32_t temp;                     /* 温度 */
        rt_int32_t humi;                     /* 湿度 */
        rt_int32_t baro;                     /* 气压 */
        rt_int32_t step;                     /* 步数 */
    } data;
    rt_uint32_t timestamp;                  /* 时间戳 */
};
```

9. 传感器框架的典型使用流程

1. **查找传感器**：调用 `rt_device_find()` 获取设备句柄。
2. **打开传感器**：根据需求选择合适的工作模式，调用 `rt_device_open()` 。
3. **读取数据**：通过 `rt_device_read()` 获取传感器数据。
4. **配置传感器**：根据应用需求，调用 `rt_device_control()` 设置传感器参数。
5. **关闭传感器**：完成操作后，调用 `rt_device_close()` 关闭设备。

1. RT-Thread 传感器框架的核心接口

RT-Thread 传感器框架提供了一套标准化接口，所有传感器设备都要通过这些接口完成设备注册、初始化、数据采集、控制等功能。主要接口包括：

- **设备注册和初始化**：用于将传感器设备注册到系统，并进行初始化配置。
- **数据读取**：用于从传感器读取数据。
- **设备控制**：通过命令控制字对设备进行各种配置，如采样频率、电源模式等。

DHT11 传感器驱动库实现了这些接口，使 DHT11 能够在 RT-Thread 传感器框架中被统一管理和调用。

2. 传感器数据结构定义

在 RT-Thread 传感器框架中，传感器数据使用 `rt_sensor_data` 结构体来存储。它包含传感器类型、数据值和时间戳等字段。

```
struct rt_sensor_data
{
    union
    {
        struct { rt_int32_t x, y, z; } acce; // 加速度数据
        struct { rt_int32_t x, y, z; } gyro; // 陀螺仪数据
        struct { rt_int32_t x, y, z; } mag;  // 磁力计数据
        rt_int32_t temp;                     // 温度
        rt_int32_t humi;                     // 湿度
        rt_int32_t baro;                     // 气压
        rt_int32_t step;                     // 计步
    } data;
    rt_uint32_t timestamp;                  // 时间戳
};
```

DHT11 驱动将读取的温度和湿度数据填充到该结构体中，然后统一返回给上层应用。

3. DHT11 驱动如何与传感器框架对接

DHT11 传感器驱动库包含了以下几个关键的接口实现，用于对接 RT-Thread 传感器框架：

a. 设备初始化

在 RT-Thread 中，传感器设备需要注册到系统，才能被上层应用调用。DHT11 通过 `rt_hw_dht11_init()` 函数完成初始化和注册工作：

- **设备信息配置**：在 `rt_hw_dht11_init()` 中，首先初始化 `rt_sensor_device` 结构体，将 DHT11 的相关信息填充到该结构体中：

```
sensor->info.type       = RT_SENSOR_CLASS_TEMP;     // 设备类型：温度传感器
sensor->info.vendor     = RT_SENSOR_VENDOR_DALLAS;  // 厂商：Dallas
sensor->info.model      = "DHT11";                  // 型号：DHT11
sensor->info.unit       = RT_SENSOR_UNIT_DCELSIUS;  // 数据单位：摄氏度
sensor->info.intf_type  = RT_SENSOR_INTF_ONEWIRE;   // 接口类型：单总线
sensor->info.range_max  = SENSOR_TEMP_RANGE_MAX;    // 温度范围上限
sensor->info.range_min  = SENSOR_TEMP_RANGE_MIN;    // 温度范围下限
sensor->info.period_min = 100;                      // 最小采样周期
```

- **设备接口实现**：将数据采集和控制操作函数注册到 `sensor->ops` 中，这些操作包括数据读取（`dht11_fetch_data`）和控制（`dht11_control`）【7tsource】。
- **注册设备**：调用 `rt_hw_sensor_register()` 将 DHT11 设备注册到系统中。注册时指定了设备名称和设备标志（`RT_DEVICE_FLAG_RDONLY`），确保 DHT11 传感器设备能够被 RT-Thread 统一管理。

```
rt_hw_sensor_register(sensor, name, RT_DEVICE_FLAG_RDONLY, RT_NULL);
```

b. 数据读取接口：`dht11_fetch_data`

`dht11_fetch_data()` 是 DHT11 驱动中用于数据采集的核心函数。此函数对接了 RT-Thread 传感器框架的数据读取接口 `rt_device_read()`，当上层应用调用 `rt_device_read()` 时，会触发 `dht11_fetch_data()`。

具体流程如下：

1. **轮询获取数据**：DHT11 使用轮询模式采集数据，`dht11_fetch_data` 调用 `dht11_polling_get_data()` 函数从硬件读取温湿度数据。
2. **数据填充**：读取的数据填充到 `rt_sensor_data` 结构体的 `temp` 和 `humi` 字段中。DHT11 的温度和湿度数据使用 32 位整数表示，其中低 16 位是温度，高 16 位是湿度【7tsource】。
3. **返回数据**：将数据结构体返回给框架，框架将其提供给上层应用。

示例代码：

```
static rt_size_t dht11_fetch_data(struct rt_sensor_device *sensor, void
*buf, rt_size_t len)
{
    // 轮询模式读取 DHT11 数据
    if (sensor->config.mode == RT_SENSOR_MODE_POLLING)
    {
        return dht11_polling_get_data(sensor, buf);
    }
    return 0;
}
```

c. 控制接口： dht11_control

在 RT-Thread 传感器框架中，控制接口允许应用对传感器进行各种配置（如输出频率、测量范围等）。DHT11 驱动实现了一个基本的控制接口 `dht11_control` ，目前未做复杂的控制操作，但可扩展以支持更多的命令。

当上层调用 `rt_device_control()` 函数时，会触发 `dht11_control()` ，此接口对接了框架的控制功能，支持执行电源模式、数据速率、量程等配置命令。

```
static rt_err_t dht11_control(struct rt_sensor_device *sensor, int cmd,
void *args)
{
    return RT_EOK; // 简化返回值处理，未来可扩展控制命令
}
```

4. 使用 DHT11 的数据读取示例

在成功注册和初始化后，应用可以按照传感器框架的标准流程访问 DHT11 传感器。以下是一个典型的数据读取过程：

1. **查找设备**：通过 `rt_device_find()` 查找传感器设备句柄。

   ```
   dev = rt_device_find("dht11");
   ```

2. **打开设备**：调用 `rt_device_open()` 打开设备，并选择只读模式（ `RT_DEVICE_FLAG_RDONLY` ）。

   ```
   rt_device_open(dev, RT_DEVICE_FLAG_RDONLY);
   ```

3. **读取数据**：调用 `rt_device_read()` 从设备中读取传感器数据。`rt_device_read()` 内部会调用 `dht11_fetch_data()` 函数，采集到的温湿度数据将填充到 `rt_sensor_data` 结构体中【9tsource】。

```
struct rt_sensor_data data;
rt_device_read(dev, 0, &data, 1);
```

4. **解析数据**：温湿度数据被编码为 32 位整数，需要分离出温度和湿度部分。

```
uint8_t temp = (data.data.temp & 0xFFFF);        // 提取温度
uint8_t humi = (data.data.temp & 0xFFFF0000) >> 16; // 提取湿度
```

5. **关闭设备**：操作完成后，调用 `rt_device_close()` 关闭设备。

```
rt_device_close(dev);
```

5. DHT11 示例代码总览

以下代码展示了一个完整的读取过程，展示了如何利用 RT-Thread 传感器框架的接口来访问 DHT11 传感器：

```
/*
 * Copyright (c) 2006-2018, RT-Thread Development Team
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Change Logs:
 * Date           Author         Notes
 * 2019-08-01     LuoGong         the first version.
 * 2019-08-15     MurphyZhao      add lock and modify code style
 */

#include <rtthread.h>
#include <rtdevice.h>
#include "sensor.h"
#include "sensor_dallas_dht11.h"
#include "bsp_system.h"

/* Modify this pin according to the actual wiring situation */
#define DHT11_DATA_PIN    GET_PIN(A, 7)
```

```c
static void read_temp_entry(void *parameter)
{
    rt_device_t dev = RT_NULL;
    struct rt_sensor_data sensor_data;
    rt_size_t res;
    rt_uint8_t get_data_freq = 1; /* 设置数据读取频率为 1Hz */

    /* 查找已注册的 DHT11 设备 */
    dev = rt_device_find("temp_micu");
    if (dev == RT_NULL)
    {
        rt_kprintf("Device not found!\n");
        return;
    }

    /* 以读写模式打开设备 */
    if (rt_device_open(dev, RT_DEVICE_FLAG_RDWR) != RT_EOK)
    {
        rt_kprintf("Open device failed!\n");
        return;
    }

    /* 设置 DHT11 的数据输出速率为 1Hz */
    rt_device_control(dev, RT_SENSOR_CTRL_SET_ODR, (void *)
(&get_data_freq));

    /* 进入循环，读取并输出传感器数据 */
    while (1)
    {
        /* 从传感器读取温湿度数据 */
        res = rt_device_read(dev, 0, &sensor_data, 1);

        if (res != 1)
        {
            rt_kprintf("Read data failed! result is %d\n", res);
            rt_device_close(dev);
            return;
        }
        else
        {
            if (sensor_data.data.temp >= 0)
            {
                uint8_t temp = (sensor_data.data.temp & 0xffff);
  // 提取温度数据
                uint8_t humi = (sensor_data.data.temp & 0xffff0000) >>
16; // 提取湿度数据
```

```
                rt_kprintf("Temperature: %d, Humidity: %d\n", temp,
humi);
            }
        }

        rt_thread_delay(1000); // 延迟 1 秒
    }
}

/* 传感器示例启动函数 */
static int dht11_read_temp_sample(void)
{
    rt_thread_t dht11_thread;

    /* 创建温湿度数据读取线程 */
    dht11_thread = rt_thread_create("dht_tem",
                                    read_temp_entry,
                                    RT_NULL,
                                    1024,
                                    RT_THREAD_PRIORITY_MAX / 2,
                                    20);
    if (dht11_thread != RT_NULL)
    {
        rt_thread_startup(dht11_thread);
    }

    return RT_EOK;
}
INIT_APP_EXPORT(dht11_read_temp_sample);

/* DHT11 硬件初始化和设备注册函数 */
static int rt_hw_dht11_port(void)
{
    struct rt_sensor_config cfg;

    cfg.intf.user_data = (void *)DHT11_DATA_PIN;
    rt_hw_dht11_init("dht", &cfg);

    struct rt_object_information *info;
    struct rt_list_node *node;
    struct rt_device *dev;

    /* 获取设备对象的信息 */
    info = rt_object_get_information(RT_Object_Class_Device);

    if (info == RT_NULL)
```

```
    {
        rt_kprintf("No device information available.\n");
        return 0;
    }

    /* 遍历设备链表 */
    for (node = info->object_list.next; node != &(info->object_list);
node = node->next)
    {
        dev = rt_list_entry(node, struct rt_device, parent.list);
        rt_kprintf("Device Name: %s\n", dev->parent.name);
    }

    return RT_EOK;
}
INIT_COMPONENT_EXPORT(rt_hw_dht11_port);



/*
 * Copyright (c) 2006-2021, RT-Thread Development Team
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Change Logs:
 * Date           Author       Notes
 * 2024-10-18     Xifeng       the first version
 */
#include "sensor_app.h"
#include "sensor_dallas_dht11.h"

// 定义DHT11引脚
#define DHT11_PIN GET_PIN(A, 7)


/**
 * @brief 温湿度处理函数
 *
 * 该函数用于定期读取温湿度数据
 */
void temperature_humidity_proc(void *parameter)
{
    int32_t temperature_humidity = 0;
    while (1)
    {
        // 从 DHT11 读取温湿度数据，返回值包含温度和湿度
```

```
        temperature_humidity = dht11_get_temperature(DHT11_PIN);

        // 从返回值中提取温度和湿度数据
        uint8_t humidity = (temperature_humidity >> 16) & 0xFF;
        uint8_t temperature = temperature_humidity & 0xFF;

        rt_kprintf("Temp: %d Humi: %d%\n", temperature, humidity);

        rt_thread_mdelay(1000);
    }
}

/**
 * @brief 温湿度传感器初始化
 *
 * 初始化 DHT11 传感器
 */
void temperature_humidity_init(void)
{
    // 初始化 DHT11 传感器
    if (dht11_init(DHT11_PIN) == CONNECT_SUCCESS)
    {
        rt_kprintf("DHT11 init Ok\n");

        // 创建并启动温湿度数据处理线程
        rt_thread_t temp_humidity_thread =
rt_thread_create("temp_humidity_proc",

 temperature_humidity_proc,  // 线程入口函数
                                                        RT_NULL,
                // 线程形参
                                                        1024,
                 // 线程栈大小
                                                        10,
                 // 线程优先级
                                                        10);
                // 时间片大小

        if (temp_humidity_thread != RT_NULL)
        {
            rt_thread_startup(temp_humidity_thread);  // 启动线程
        }
    }
    else
    {
        rt_kprintf("DHT11 failed\n");
```

```
        }
    }
```

MQTT协议

什么是MQTT协议

MQTT（Message Queuing Telemetry Transport）是一种发布/订阅模式的消息传输协议，最初由IBM设计，专为需要低带宽、不稳定网络和低功耗的场景设计，比如物联网（IoT）环境。它的核心是通过简洁的消息格式和高效的通信方式，使设备之间的数据传输更加稳定和快速。

MQTT的工作原理

MQTT协议基于“发布/订阅”模式进行通信，这意味着设备（客户端）之间的通信是通过中介（代理/Broker）来实现的。它的主要工作流程包括：设备连接到代理、发布者将消息发布到主题、代理将消息转发给订阅了该主题的订阅者。每一步我们都详细解释并举例说明。

1. 设备连接到代理（Broker）

在MQTT协议中，所有设备（客户端）都必须先与代理建立连接，才能进行消息的发布或订阅。这个连接是一个TCP/IP连接，并且通常是长连接，以保持实时的数据传输。每个客户端连接时都会有一个唯一的客户端ID，用来标识自己。

举例：假设有一套智能家居系统，其中包括多个设备，如温度传感器、灯光控制器、手机App等。每个设备在启动后，都会连接到MQTT代理，并使用唯一的客户端ID注册自己。比如，温度传感器的客户端ID可以是 `sensor_01`，灯光控制器的ID是 `light_controller`，手机App的ID是 `user_app`。

2. 发布者（Publisher）向主题（Topic）发布消息

发布者是指将数据发送到代理的设备或应用。发布者在发送数据时，必须指定一个“主题”（Topic），这个主题可以理解为消息的分类标签。主题的结构是分层的，用“/”来区分层级，使主题可以按层级进行分类。

举例：在智能家居系统中，温度传感器每隔一段时间会将采集到的温度数据发布到 `/home/livingroom/temperature` 主题。这意味着温度数据会通过代理传递到 `/home/livingroom/temperature` 这个主题下。发布消息的内容可以是温度值，例如 `23°C`。

3. 订阅者（Subscriber）订阅主题并接收消息

订阅者是那些对某个主题感兴趣并希望接收该主题下的消息的设备或应用。客户端可以通过代理订阅一个或多个主题，一旦订阅成功，当有新的消息发布到这些主题时，代理会自动将消息转发给所有的订阅者。

举例：在智能家居系统中，用户的手机App可能订阅了 `/home/livingroom/temperature` 主题，以便接收客厅的实时温度数据。当温度传感器发布新数据时，代理会将最新的温度消息推送给手机App，用户可以在手机上看到最新的温度值。

4. 代理（Broker）管理消息的转发

代理在整个MQTT协议中扮演了“消息路由器”的角色。它负责接收发布者的消息，根据消息的主题找到所有订阅了该主题的客户端，并将消息转发给它们。代理的存在使得发布者和订阅者之间无需直接通信，降低了系统的耦合性，提高了灵活性。

举例：假设除了用户的手机App外，智能家居系统中的空调控制器也订阅了 `/home/livingroom/temperature` 主题。当温度传感器发布一条新消息到该主题（例如温度升高到 `27°C`），代理会将这条消息同时发送给手机App和空调控制器。空调控制器收到消息后，可以自动启动降温功能，而用户则在手机上看到当前的温度。

5. 使用通配符订阅多个主题

MQTT支持两种通配符，以便客户端一次性订阅多个主题：

- 单层通配符 `+`：匹配某一层中的所有主题。
- 多层通配符 `#`：匹配指定层级开始的所有子主题。

举例：

- 如果手机App想要订阅家中所有房间的温度数据，可以使用通配符 `+/temperature`。这样一来，无论是 `/home/livingroom/temperature` 还是 `/home/bedroom/temperature` 的消息，手机App都会接收到。
- 如果手机App想要订阅家中的所有传感器数据，可以使用 `#` 通配符，即 `/home/#`，这样家中所有房间的所有数据都会被接收到。

6. 消息传输质量（QoS）

MQTT协议支持不同的消息传输质量（QoS）级别，以保证消息传输的可靠性。用户可以根据实际需要选择以下三种QoS级别：

- **QoS 0**（至多一次）：消息最多会发送一次，不保证传输成功。适用于数据不重要、偶尔丢失也没关系的场景。
- **QoS 1**（至少一次）：消息至少会发送一次，确保送达，但可能会重复。适用于需要确保消息到达的场景。
- **QoS 2**（仅一次）：确保消息仅发送一次，不会重复。适用于对数据重复非常敏感的场景。

举例：

- 温度传感器可能使用QoS 0来发送温度数据，因为即便偶尔丢失一两条数据，也不会影响整体监控。
- 如果灯光控制器需要接收“开灯”或“关灯”指令，可以使用QoS 1，确保消息到达但允许重复。

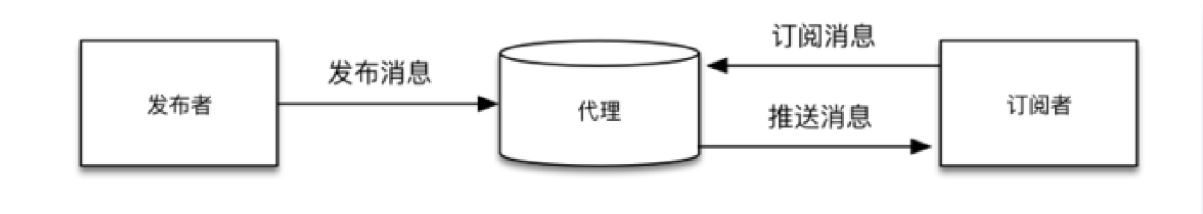
- 如果存在重要的状态更新（如报警状态），可以使用QoS 2，确保消息仅发送一次，避免重复触发报警。

7. 保留消息和遗嘱消息

- **保留消息**：代理可以保留某主题的最新消息。当新订阅者加入时，代理会自动将最新的保留消息发送给该订阅者。
- **遗嘱消息**：在客户端连接时，可以向代理设置一条“遗嘱消息”。如果该客户端意外断开连接，代理会自动向指定的主题发布这条消息，通知其他订阅者。

举例：

- 在智能家居系统中，灯光状态可以设置成保留消息。如果用户的手机App刚连接到代理并订阅了灯光主题 `/home/livingroom/light`，代理会立即将最新的灯光状态发送给手机App，即使没有新的状态更新发布。
- 如果温度传感器发生故障，意外断开连接，可以预先设置一条遗嘱消息，如“温度传感器失联”。代理会在该主题上发布这条消息，通知用户或相关设备。



MQTT的关键特性

1. 轻量级和低带宽

MQTT的设计非常轻量，它的消息头部开销很小，仅仅几个字节。这使得MQTT特别适合在网络不稳定或者带宽有限的环境中使用，如蜂窝网络或卫星连接。

2. 发布/订阅模型

MQTT采用发布/订阅模型而不是请求/响应模型，使得客户端之间可以松耦合。发布者和订阅者无需直接相互通信，只需要和代理通信，代理会负责消息的转发和分发。这种架构特别适合大规模的设备网络，如智能家居和工业物联网。

3. 消息传输质量（QoS）级别

MQTT提供三种质量级别来保证消息传输的可靠性，用户可以根据具体需求选择合适的QoS级别：

- **QoS 0**（至多一次，最多发送一次）：消息会被尽力发送一次，但如果网络异常不会重试。适合对数据丢失不敏感的场景。

- **QoS 1**（至少一次，确保送达）：消息至少会发送一次，代理会确认收到消息。如果未确认，发布者会重试。这保证消息会到达，但可能会重复。
- **QoS 2**（仅一次）：确保消息仅传输一次，避免重复传输。适合对数据重复敏感的场景。

4. 持久会话和“遗嘱”机制

- **持久会话**：在断开连接后，如果客户端设置了持久会话，代理会保留该客户端的订阅信息，重新连接时无需重新订阅。
- **遗嘱消息**：发布者在连接时可以向代理指定一条“遗嘱消息”，如果发布者意外断开连接，代理会向其他订阅者发送这条消息。这在一些重要的监控场景中非常有用。

5. 保留消息

代理可以保存主题的最新消息，当新订阅者加入时会立即收到最新的保留消息。这对于状态更新或控制指令非常有帮助，比如让新加入的设备快速同步到当前系统状态。

MQTT在物联网中的应用

MQTT因其高效的消息传输和轻量特性，非常适合用于物联网（IoT）设备之间的数据通信。典型应用场景包括：

1. **智能家居**：智能家居设备，如灯光、空调、门锁等，可以通过MQTT进行控制。用户的手机应用程序可以订阅设备的状态主题，控制家中的智能设备。
2. **工业监控**：工厂中的传感器和设备可以通过MQTT实时发送温度、湿度、振动等数据到中央服务器，用于监控和数据分析。
3. **远程医疗**：佩戴式设备（如心率监测器）可以使用MQTT将患者的生理数据实时传输给医生，帮助医生远程监控患者健康。
4. **车辆跟踪**：运输车辆可以通过MQTT协议将位置、速度等信息传送到监控平台，便于跟踪和管理车队。

MQTT的优势和不足

优势：

- **低带宽需求**：占用网络资源少，适合不稳定或低带宽网络。
- **灵活的发布/订阅机制**：实现了设备之间的解耦，设备可以独立操作。
- **多级QoS支持**：可根据实际应用场景选择消息传输的可靠性。
- **轻量和易于实现**：简洁的协议设计，让它在资源受限的设备上也能轻松实现。

不足：

- **安全性**：MQTT协议本身没有强制的安全机制。为了确保安全，需要结合TLS加密或使用鉴权机制来保护数据。

- **消息持久化和历史数据**：MQTT适合实时数据传输，但不适合需要保存历史数据的场景。
- **代理负载**：随着设备数量增加，代理的负载也会增加，对代理的处理能力提出更高的要求。

Paho MQTT库

1. 订阅列表

Paho MQTT 采用订阅列表的形式管理多个主题的订阅，订阅列表保存在 `MQTTClient` 结构体实例中，并在 MQTT 启动前进行配置。以下是配置订阅列表的示例代码：

```
// 初始化 MQTT 客户端
MQTTClient client;

// 配置订阅列表和回调函数
client.messageHandlers[0].topicFilter = MQTT_SUBTOPIC;
client.messageHandlers[0].callback = mqtt_sub_callback;
client.messageHandlers[0].qos = QOS1;
```

可以在 `menuconfig` 的 `Max pahomqtt subscribe topic handlers` 选项中配置订阅列表的最大数量。

2. 回调函数

Paho MQTT 使用回调机制来提供 MQTT 客户端的工作状态和相关事件处理。回调函数需要在 `MQTTClient` 结构体实例中注册。

回调名称	描述
connect_callback	MQTT 连接成功的回调
online_callback	MQTT 客户端成功上线的回调
offline_callback	MQTT 客户端掉线的回调
defaultMessageHandler	默认的订阅消息接收回调
messageHandlers[x].callback	订阅列表中指定主题的回调

用户可以使用 `defaultMessageHandler` 处理默认的订阅消息接收，也可以为 `messageHandlers` 数组中的每个主题提供独立的回调函数。

3. MQTT URI

Paho MQTT 支持解析不同类型的 URI，包括域名地址、IPv4 和 IPv6 地址，支持 `tcp://` 和 `ssl://` 协议。用户只需按照格式填写 URI，即可与代理建立连接。

- URI 示例：

```
域名类型：
tcp://iot.eclipse.org:1883

IPv4 类型：
tcp://192.168.10.1:1883
ssl://192.168.10.1:1884

IPv6 类型：
tcp://[fe80::20c:29ff:fe9a:a07e]:1883
ssl://[fe80::20c:29ff:fe9a:a07e]:1884
```

4. 主要 API

paho_mqtt_start

```
int paho_mqtt_start(MQTTClient *client);
```

参数	描述
client	MQTT 客户端实例对象
return	0 表示成功；其他表示失败

该函数用于启动 MQTT 客户端，根据配置项订阅相应的主题。

paho_mqtt_stop

```
int paho_mqtt_stop(MQTTClient *client);
```

参数	描述
client	MQTT 客户端实例对象
return	0 表示成功；其他表示失败

该函数用于关闭 MQTT 客户端，并释放客户端对象占用的资源。

paho_mqtt_subscribe

```
int paho_mqtt_subscribe(MQTTClient *client, enum QoS qos, const char *topic,
subscribe_cb callback);
```

参数	描述
client	MQTT 客户端实例对象
qos	订阅的 QoS 级别，目前支持 QoS1
topic	需要订阅的主题
callback	订阅主题的消息接收回调函数
return	0 表示成功；其他表示失败

该函数用于客户端订阅新的主题，并注册数据接收的回调函数。

paho_mqtt_unsubscribe

```
int paho_mqtt_unsubscribe(MQTTClient *client, const char *topic);
```

参数	描述
client	MQTT 客户端实例对象
topic	需要取消订阅的主题
return	0 表示成功；其他表示失败

该函数用于客户端取消指定主题的订阅。

paho_mqtt_publish

```
int paho_mqtt_publish(MQTTClient *client, enum QoS qos, const char *topic,
const char *msg_str);
```

参数	描述
client	MQTT 客户端实例对象
qos	发送的 QoS 级别，目前支持 QoS1
topic	数据发送的主题
msg_str	需要发送的数据字符串

参数	描述
return	0 表示成功；其他表示失败

该函数用于客户端向指定主题发送数据。

paho_mqtt_control

```
int paho_mqtt_control(MQTTClient *client, int cmd, void *arg);
```

参数	描述
client	MQTT 客户端实例对象
cmd	控制命令参数类型
arg	控制命令参数值
return	0 表示成功；其他表示失败

该函数用于控制客户端的部分参数设置。支持的命令参数包括：

参数名称	描述
MQTT_CTRL_SET_CONN_TIMEO	设置客户端连接超时时间
MQTT_CTRL_SET_RECONN_INTERVAL	设置客户端掉线重连的间隔时间
MQTT_CTRL_SET_KEEPALIVE_INTERVAL	设置发送 ping 的间隔时间
MQTT_CTRL_PUBLISH_BLOCK	设置发布数据时的阻塞或非阻塞模式

cJSON库

1. cJSON 数据结构

cJSON 的核心数据结构为 `cJSON`，每个 `cJSON` 实例表示 JSON 中的一个数据节点。此节点可以是对象、数组、字符串、数字、布尔值或 null。

```
typedef struct cJSON {
    struct cJSON *next;         // 指向链表中的下一个节点
    struct cJSON *prev;         // 指向链表中的上一个节点
    struct cJSON *child;        // 指向子节点（用于对象和数组）
    int type;                   // 节点类型（如cJSON_Object, cJSON_Array）
    char *valuestring;          // 字符串值
    int valueint;               // 整数值
    double valuedouble;         // 浮点值
    char *string;               // 键名（仅对象节点有此字段）
} cJSON;
```

cJSON 支持的数据类型如下：

- **cJSON_False**：布尔值 false
- **cJSON_True**：布尔值 true
- **cJSON_NULL**：空值 null
- **cJSON_Number**：数字类型
- **cJSON_String**：字符串
- **cJSON_Array**：数组
- **cJSON_Object**：对象

2. 初始化与内存管理

cJSON 使用内存分配函数 `malloc` 和 `free` 管理内存，开发者也可以通过 `cJSON_InitHooks` 自定义内存管理函数。在 RT-Thread 环境下，cJSON 默认使用 `rt_malloc`、`rt_free` 和 `rt_realloc` 进行内存管理。

初始化函数

```
void cJSON_InitHooks(cJSON_Hooks *hooks);
```

通过 `cJSON_Hooks` 结构体传入自定义的内存管理函数：

```
typedef struct cJSON_Hooks
{
    void *(CJSON_CDECL *malloc_fn)(size_t size);    // 分配内存
    void (CJSON_CDECL *free_fn)(void *ptr);         // 释放内存
} cJSON_Hooks;
```

使用方法示例：

```
cJSON_Hooks hooks;
hooks.malloc_fn = rt_malloc;
hooks.free_fn = rt_free;
cJSON_InitHooks(&hooks);
```

3. cJSON API

解析 JSON

1. 基本解析函数

`cJSON_Parse` 用于将 JSON 格式的字符串解析为 `cJSON` 对象：

```
cJSON *cJSON_Parse(const char *value);
```

2. 带长度的解析

`cJSON_ParseWithLength` 允许设置字符串长度：

```
cJSON *cJSON_ParseWithLength(const char *value, size_t buffer_length);
```

3. 错误定位

使用 `cJSON_GetErrorPtr` 获取解析错误的具体位置。

```
const char *error = cJSON_GetErrorPtr();
```

生成 JSON

1. 格式化输出

使用 `cJSON_Print` 将 cJSON 对象转换为格式化的 JSON 字符串：

```
char *cJSON_Print(const cJSON *item);
```

2. 非格式化输出

`cJSON_PrintUnformatted` 输出不带缩进的紧凑 JSON 字符串：

```
char *cJSON_PrintUnformatted(const cJSON *item);
```

3. 自定义缓冲区输出

可以使用 `cJSON_PrintPreallocated` 将 JSON 写入已有的缓冲区中：

```
cJSON_bool cJSON_PrintPreallocated(cJSON *item, char *buffer, int length,
cJSON_bool format);
```

修改 JSON 数据

1. 创建 JSON 节点

cJSON 提供了一系列函数来创建不同类型的 JSON 节点：

```
cJSON *cJSON_CreateString(const char *string);
cJSON *cJSON_CreateNumber(double num);
cJSON *cJSON_CreateObject(void);
cJSON *cJSON_CreateArray(void);
```

2. 添加和删除节点

- `cJSON_AddItemToObject`：将节点添加到对象。
- `cJSON_AddItemToArray`：将节点添加到数组。
- `cJSON_Delete`：删除整个 cJSON 对象。

数据访问与获取

1. 获取对象字段

使用 `cJSON_GetObjectItem` 根据键名获取对象字段：

```
cJSON *cJSON_GetObjectItem(const cJSON *object, const char *string);
```

2. 访问数组元素

使用 `cJSON_GetArrayItem` 通过索引访问数组元素：

```
cJSON *cJSON_GetArrayItem(const cJSON *array, int index);
```

3. 检查类型

cJSON 提供了 `cJSON_IsNumber`、`cJSON_IsString` 等类型检查函数【35†source】。

4. 高级功能

JSON Pointer

cJSON 支持 JSON Pointer (RFC 6901)，用于通过路径快速访问 JSON 数据。

1. **JSON Pointer 获取**

   使用 `cJSONUtils_GetPointer` 通过路径获取 JSON 数据：

   ```
   cJSON *cJSONUtils_GetPointer(cJSON *object, const char *pointer);
   ```

2. **大小写敏感获取**

   `cJSONUtils_GetPointerCaseSensitive` 支持大小写敏感的路径访问：

   ```
   cJSON *cJSONUtils_GetPointerCaseSensitive(cJSON *object, const char
   *pointer);
   ```

JSON Patch

cJSON 实现了 JSON Patch (RFC 6902)，用于修改 JSON 数据的差异应用。

1. **生成 Patch**

   `cJSONUtils_GeneratePatches` 根据两个 JSON 对象生成 Patch：

   ```
   cJSON *cJSONUtils_GeneratePatches(cJSON *from, cJSON *to);
   ```

2. **应用 Patch**

   `cJSONUtils_ApplyPatches` 将 Patch 应用于 JSON 对象：

   ```
   int cJSONUtils_ApplyPatches(cJSON *object, const cJSON *patches);
   ```

JSON Merge Patch

cJSON 支持 JSON Merge Patch (RFC 7396) 功能，用于合并 JSON 数据。

1. **生成 Merge Patch**

   使用 `cJSONUtils_GenerateMergePatch` 根据两个对象生成 Merge Patch：

   ```
   cJSON *cJSONUtils_GenerateMergePatch(cJSON *from, cJSON *to);
   ```

2. 应用 Merge Patch

使用 `cJSONUtils_MergePatch` 将 Merge Patch 应用于目标 JSON 对象：

```
cJSON *cJSONUtils_MergePatch(cJSON *target, const cJSON *patch);
```

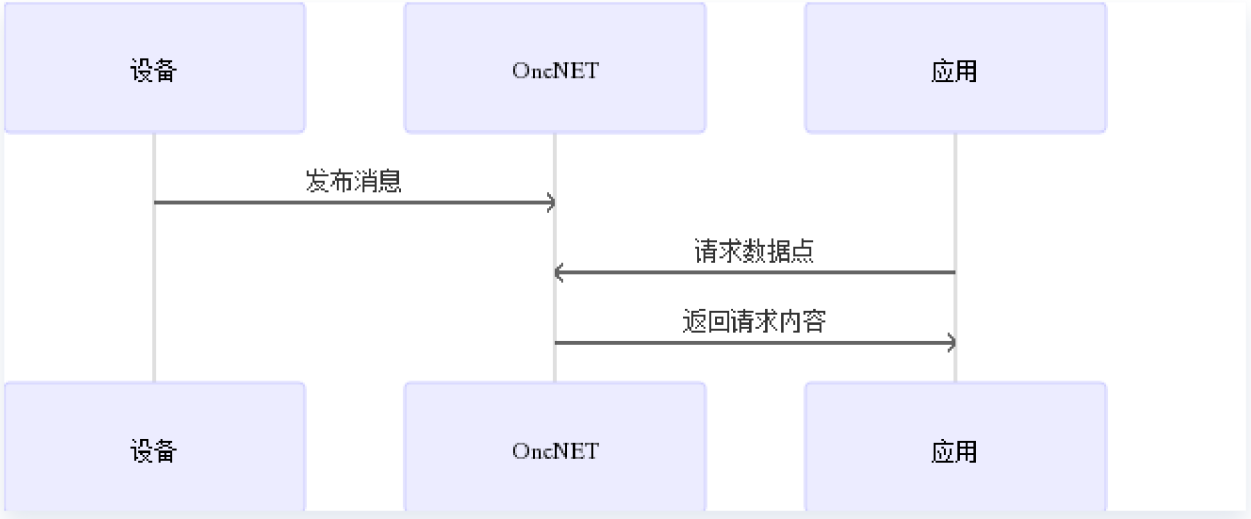
OnetNet库

工作原理

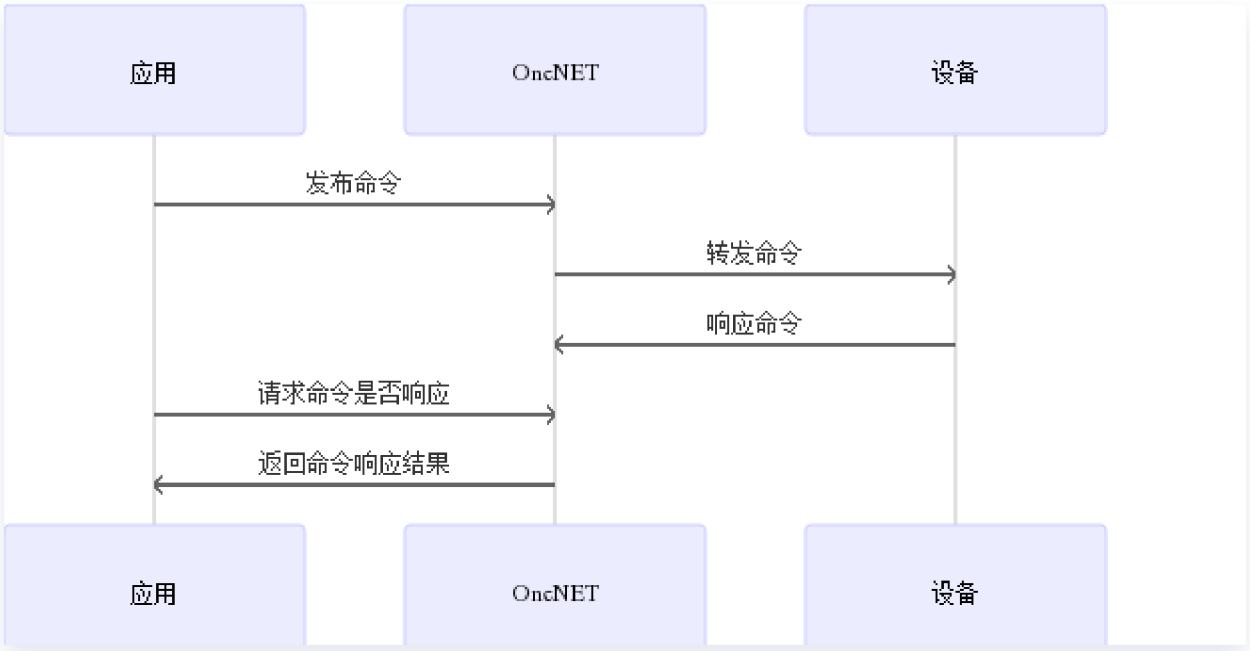
OneNET 软件包数据的上传和命令的接收是基于 MQTT 实现的，OneNET 的初始化其实就是 MQTT 客户端的初始化，初始化完成后，MQTT 客户端会自动连接 OneNET 平台。数据的上传其实就是往特定的 topic 发布消息。当服务器有命令或者响应需要下发时，会将消息推送给设备。

获取数据流、数据点，发布命令则是基于 HTTP Client 实现的，通过 POST 或 GET 将相应的请求发送给 OneNET 平台，OneNET 将对应的数据返回，这样，我们就能在网页上或者手机 APP 上看到设备上传的数据了。

下图是应用显示设备上传数据的流程图



下图是应用下发命令给设备的流程图



1. 主要依赖库

1.1 cJSON

cJSON 是一个轻量级的 JSON 解析和生成库，适合嵌入式系统使用。OneNET 库使用 cJSON 来构建符合 OneNET 平台要求的 JSON 数据，用于发送设备数据、命令响应和二进制文件的元数据等。主要用到的 `cJSON` API 包括：

- `cJSON_CreateObject`：创建一个 JSON 对象。
- `cJSON_AddStringToObject` 和 `cJSON_AddNumberToObject`：将字符串或数字加入到 JSON 对象中。
- `cJSON_PrintUnformatted`：将 JSON 对象转换为不带缩进的字符串形式，用于发布消息。
- `cJSON_Delete`：释放 JSON 对象。

1.2 paho-mqtt

paho-mqtt 是一个开源的 MQTT 客户端库，实现了发布/订阅模式的数据传输。OneNET 库通过 `paho-mqtt` 的 API 来实现设备与 OneNET 云端的连接、数据上传以及命令的接收与响应。主要用到的 `paho-mqtt` API 包括：

- `paho_mqtt_start`：启动 MQTT 客户端，与 MQTT 代理建立连接。
- `MQTTPublish`：将消息发布到指定的主题。
- `MQTTSubscribe`：订阅主题并指定消息到达的回调处理函数。

- 回调函数：连接成功、掉线、消息到达等回调函数，用于管理 MQTT 客户端的状态。

---

2. 库结构概览

OneNET 库包括以下文件：

- `onenet.h` ： 主头文件，定义了库的主要宏、数据结构和 API 接口。
- `onenet_mqtt.c` ： 核心实现文件，负责 MQTT 连接管理、数据上传和回调处理。
- `onenet_port.c` ： 设备端口实现文件，处理接收到的数据和设备信息管理。

---

3. 核心实现

3.1 MQTT 客户端初始化

OneNET 库的初始化通过 `onenet_mqtt_init` 函数完成，此函数负责设置 MQTT 客户端的基本参数，包括连接 OneNET 平台所需的 URI、客户端 ID、用户名和密码等信息，并启动 MQTT 客户端。关键步骤如下：

1. **设置 MQTT 参数**：初始化 `MQTTClient` 结构体中的参数，包括：
   - `uri` ： 连接 OneNET 平台的服务器 URI（如 `tcp://183.230.40.96:1883` ）。
   - `clientID` ： 设备 ID。
   - `username` 和 `password` ： 使用 `cJSON` 提供的设备 ID 和 API 密钥作为认证。
2. **启动 MQTT 客户端**：调用 `paho_mqtt_start` 函数启动 MQTT 客户端，并设置以下回调函数：
   - `connect_callback` ： 连接成功时调用的回调函数。
   - `online_callback` ： 设备上线时调用。
   - `offline_callback` ： 设备掉线时调用。
   - `defaultMessageHandler` ： 默认的消息接收回调。

```
int onenet_mqtt_init(void)
{
    mq_client.uri = onenet_info.server_uri;
    mq_client.condata.clientID.cstring = onenet_info.device_id;
    mq_client.condata.username.cstring = onenet_info.pro_id;
    mq_client.condata.password.cstring = onenet_info.auth_info;
    mq_client.defaultMessageHandler = mqtt_callback;
    paho_mqtt_start(&mq_client);
}
```

3.2 发布数据

OneNET 库支持将设备数据发布到 OneNET 云端，支持的类型包括数字、字符串和二进制数据。每种数据类型通过 `cJSON` 构建 JSON 格式的数据，再利用 `paho-mqtt` 将数据发布到指定主题。

发布数字数据

`onenet_mqtt_upload_digit` 用于发布数字数据，流程如下：

1. **构建 JSON 数据**：调用 `onenet_mqtt_get_digit_data` 构建 JSON 格式的数据。主要操作包括：
   - 使用 `cJSON_CreateObject` 创建根对象 `root` 。
   - 使用 `cJSON_AddNumberToObject` 将数字添加到指定数据流中。

```
static rt_err_t onenet_mqtt_get_digit_data(const char *ds_name, const
double digit, char **out_buff, size_t *length)
{
    cJSON *root = cJSON_CreateObject();
    cJSON *params = cJSON_CreateObject();
    cJSON_AddItemToObject(root, "params", params);
    cJSON_AddNumberToObject(params, ds_name, digit);
    *out_buff = cJSON_PrintUnformatted(root);
}
```

2. **发布数据**：使用 `onenet_mqtt_publish` 将数据发送到 `ONENET_TOPIC_DP` 主题。

```
rt_err_t onenet_mqtt_upload_digit(const char *ds_name, const double
digit)
{
    onenet_mqtt_publish(ONENET_TOPIC_DP, (uint8_t *)send_buffer,
length);
}
```

发布字符串数据

`onenet_mqtt_upload_string` 用于发布字符串数据，与发布数字数据流程相似，只是使用 `cJSON_AddStringToObject` 添加字符串数据。

```
static rt_err_t onenet_mqtt_get_string_data(const char *ds_name, const
char *str, char **out_buff, size_t *length)
{
    cJSON *root = cJSON_CreateObject();
    cJSON *params = cJSON_CreateObject();
    cJSON_AddStringToObject(params, ds_name, str);
    *out_buff = cJSON_PrintUnformatted(root);
}
```

发布二进制数据

`onenet_mqtt_upload_bin` 用于上传二进制数据。通过 `onenet_mqtt_get_bin_data` 函数构建 JSON 元数据，再将二进制文件附加到 JSON 数据中上传。

```
rt_err_t onenet_mqtt_upload_bin(const char *ds_name, uint8_t *bin,
size_t len)
{
    uint8_t *send_buffer = RT_NULL;
    onenet_mqtt_get_bin_data(ds_name, bin, len, &send_buffer, &length);
    onenet_mqtt_publish(ONENET_TOPIC_DP, send_buffer, length);
}
```

3.3 订阅和命令响应

OneNET 库通过 `paho-mqtt` 实现订阅机制，以便设备能够接收来自云端的命令，并提供了 `onenet_set_cmd_rsp_cb` API 用于设置命令响应回调。

- **回调函数注册**：在 `mqtt_callback` 中设置当命令到达时如何响应。命令数据通过 `msg_data->message->payload` 提取，并交由用户设置的 `cmd_rsp_cb` 回调函数处理。

```
static void mqtt_callback(MQTTClient *c, MessageData *msg_data)
{
    if (onenet_mqtt.cmd_rsp_cb)
    {
        onenet_mqtt.cmd_rsp_cb((uint8_t *) msg_data->message->payload,
msg_data->message->payloadlen, &response_buf, &res_len);
        onenet_mqtt_publish(topicname, response_buf, res_len);
    }
}
```

- **设置命令响应回调**：用户可以调用 `onenet_set_cmd_rsp_cb` 设置自定义的命令响应回调函数。

```
void onenet_set_cmd_rsp_cb(void (*cmd_rsp_cb)(uint8_t *recv_data, size_t
recv_size, uint8_t **resp_data, size_t *resp_size))
{
    onenet_mqtt.cmd_rsp_cb = cmd_rsp_cb;
}
```

3.4 文件上传支持

如果启用了文件系统支持（`RT_USING_DFS`），OneNET 库允许从指定路径上传二进制文件，流程如下：

1. **读取文件**：使用 `stat` 和 `read` 函数读取文件内容。
2. **构建 JSON 数据**：使用 `onenet_mqtt_get_bin_data` 构建文件的元数据 JSON。
3. **发布数据**：调用 `onenet_mqtt_publish` 将文件上传到 OneNET 云端。

```
rt_err_t onenet_mqtt_upload_bin_by_path(const char *ds_name, const char
*bin_path)
{
    int fd = open(bin_path, O_RDONLY);
    read(fd, bin_array, file_stat.st_size);
    onenet_mqtt_get_bin_data(ds_name, bin_array, bin_size, &send_buffer,
&length);
    onenet_mqtt_publish(ONENET_TOPIC_DP, send_buffer, length);
}
```

OneNET 初始化

```
int onenet_mqtt_init(void);
```

OneNET 初始化函数，在使用 OneNET 功能前调用。

参数	描述
无	无
返回	--
0	成功
-1	获得设备信息失败
-2	MQTT 客户端初始化失败

设置命令响应函数

```
void onenet_set_cmd_rsp_cb(void(*cmd_rsp_cb)(uint8_t *recv_data, size_t
recv_size, uint8_t **resp_data, size_t *resp_size));
```

设置命令响应回调函数。

参数	描述
recv_data	接收到的数据
recv_size	数据的长度
resp_data	响应数据
resp_size	响应数据的长度
返回	--
无	无

MQTT上传数据到指定主题

```
rt_err_t onenet_mqtt_publish(const char *topic, const uint8_t *msg, size_t
len);
```

利用 MQTT 向指定主题发送消息。

参数	描述
topic	主题
msg	要上传的数据
len	数据长度
返回	--
0	上传成功
-1	上传失败

MQTT上传字符串到 OneNET

```
rt_err_t onenet_mqtt_upload_string(const char *ds_name, const char *str);
```

利用 MQTT 向 OneNET 平台发送字符串数据。

参数	描述
ds_name	数据流名称
str	要上传的字符串
返回	--
0	上传成功
-5	内存不足

MQTT上传数字到 OneNET

```
rt_err_t onenet_mqtt_upload_digit(const char *ds_name, const double digit);
```

利用 MQTT 向 OneNET 平台发送数字数据。

参数	描述
ds_name	数据流名称
digit	要上传的数字
返回	--
0	上传成功
-5	内存不足

MQTT上传二进制文件到 OneNET

```
rt_err_t onenet_mqtt_upload_bin(const char *ds_name, const uint8_t *bin, size_t
len);
```

利用 MQTT 向 OneNET 平台发送二进制文件。会动态申请内存来保存二进制文件，使用前请确保有足够的内存。

参数	描述
ds_name	数据流名称
bin	二进制文件
len	二进制文件大小
返回	--
0	上传成功
-1	上传失败

MQTT通过路径上传二进制文件到 OneNET

```
rt_err_t onenet_mqtt_upload_bin_by_path(const char *ds_name, const char
*bin_path);
```

利用 MQTT 向 OneNET 平台发送二进制文件。

参数	描述
ds_name	数据流名称
bin_path	二进制文件路径
返回	--
0	上传成功
-1	上传失败

HTTP上传字符串到 OneNET

```
rt_err_t onenet_http_upload_string(const char *ds_name, const char *str);
```

利用 HTTP 向 OneNET 平台发送字符串数据，不推荐使用，推荐使用 MQTT 上传。

参数	描述
ds_name	数据流名称
str	要上传的字符串
返回	--
0	上传成功
-5	内存不足

HTTP上传数字到 OneNET

```
rt_err_t onenet_http_upload_digit(const char *ds_name, const double digit);
```

利用 HTTP 向 OneNET 平台发送数字数据，不推荐使用，推荐使用 MQTT 上传。

参数	描述
ds_name	数据流名称
digit	要上传的数字
返回	--
0	上传成功
-5	内存不足

获取数据流信息

```
rt_err_t onenet_http_get_datastream(const char *ds_name, struct
rt_onenet_ds_info *datastream);
```

从 OneNET 平台获取指定数据流信息，并将信息保存在 `datastream` 结构体中。

参数	描述
ds_name	数据流名称
datastream	保存数据流信息的结构体
返回	--
0	成功
-1	获取响应失败
-5	内存不足

获取最后N个数据点信息

```
cJSON *onenet_get_dp_by_limit(char *ds_name, size_t limit);
```

从 OneNET 平台获取指定数据流的 n 个数据点信息。

参数	描述
ds_name	数据流名称
limit	要获取的数据点个数
返回	--
cJSON	数据点信息
RT_NULL	失败

获取指定时间内的数据点信息

```
cJSON *onenet_get_dp_by_start_end(char *ds_name, uint32_t start, uint32_t end,
size_t limit);
```

从 OneNET 平台获取指定数据流在指定时间段内的 n 个数据点信息。时间参数需为 Unix 时间戳。

参数	描述
ds_name	数据流名称
start	开始查询的时间
end	结束查询的时间
limit	要获取的数据点个数
返回	--
cJSON	数据点信息
RT_NULL	失败

注册设备

```
rt_err_t onenet_http_register_device(const char *dev_name, const char
*auth_info);
```

向 OneNET 平台注册设备，并返回设备 ID 和 API Key。设备 ID 和 API Key 会调用 `onenet_port_save_device_info` 由用户处理。

参数	描述
dev_name	设备名字
auth_info	鉴权信息
返回	--
0	注册成功
-5	内存不足

保存设备信息

```
rt_err_t onenet_port_save_device_info(char *dev_id, char *api_key);
```

保存注册后返回的设备信息，需要用户实现。

参数	描述
dev_id	设备 ID
api_key	设备 API Key
返回	--
0	成功
-1	失败

获取设备注册信息

```
rt_err_t onenet_port_get_register_info(char *dev_name, char *auth_info);
```

获取注册设备所需的信息，需要用户实现。

参数	描述
dev_name	存放设备名字的指针
auth_info	存放鉴权信息的指针
返回	--
0	成功

参数	描述
-1	失败

获取设备信息

```
rt_err_t onenet_port_get_device_info(char *dev_id, char *api_key, char
*auth_info);
```

获取设备信息用于登录 OneNET 平台，需要用户实现。

参数	描述
dev_id	存放设备 ID 的指针
api_key	存放设备 API Key 的指针
auth_info	存放鉴权信息的指针
返回	--
0	成功
-1	失败

设备是否注册

```
rt_bool_t onenet_port_is_registed(void);
```

判断设备是否已注册，需要用户实现。

参数	描述
无	无
返回	--
RT_TRUE	已注册
RT_FALSE	未注册