DMA使用分析

作者：滑国青

历史：2023.08.25 v0.1 写作中，未完成

# 前言

# Linux DMA基本知识

## Linux 之 DMA

原文链接：https://blog.csdn.net/wencun0310/article/details/129272783

DMA（Direct Memory Access，直接内存访问）是一种计算机系统中常用的数据传输方式，它可以让设备在不占用CPU时间的情况下，直接访问内存，实现高速数据传输。在数据传输量大、速度要求高的场景中，DMA可以大大提高系统性能。

DMA传输过程中，通常需要使用一个特殊的硬件设备——DMA控制器。DMA控制器是一种专门的外设，可以控制数据传输的方向和流程，通过中断或者I/O端口与CPU通信。DMA控制器可以实现从内存到设备、从设备到内存、设备之间等多种传输方式。

1. DMA 控制器驱动程序

DMA 控制器驱动程序是实现 DMA 传输的核心。它需要控制 DMA 控制器的寄存器，设置 DMA 传输参数，开启 DMA 传输等操作。DMA 控制器驱动程序需要根据硬件平台的不同进行适当的修改和调整。

在 Linux 系统中，DMA 控制器驱动程序需要实现 dmaengine 驱动框架，该框架是一个用于支持多种类型的 DMA 控制器和设备的框架。 dmaengine 驱动框架提供了 DMA API 和 DMA 设备 API。DMA API 用于管理 DMA 传输，DMA 设备 API 用于管理 DMA 控制器和设备的注册、卸载等操作。

DMA 控制器驱动程序还需要注册中断处理函数，以便在 DMA 传输完成时能够及时响应中断请求。中断处理函数需要根据 DMA 控制器的具体实现进行相应的修改和调整。

2. 设备驱动程序

设备驱动程序需要使用 DMA 控制器来进行数据传输。在 Linux 系统中，设备驱动程序通常需要使用 dma\_alloc\_coherent() 函数来为 DMA 分配内存，并使用 dma\_map\_single() 或 dma\_map\_sg() 函数将内存映射到 DMA 控制器中。dma\_alloc\_coherent() 函数可以保证所分配的内存是连续的，适合 DMA 传输。

在数据传输完成后，设备驱动程序需要使用 dma\_unmap\_single() 或 dma\_unmap\_sg() 函数来解除内存映射，释放 DMA 控制器和内存资源。

3. DMA 引擎框架

DMA 引擎框架是一个可扩展的框架，用于管理各种类型的 DMA 控制器和设备。它包含了 DMA API，用于实现 DMA 传输，以及 DMA 设备 API，用于管理 DMA 控制器和设备的注册、卸载等操作。DMA 引擎框架可以方便地与其他驱动程序集成，提高了 DMA 的可用性和灵活性。

DMA 引擎框架中的 DMA API 是通用的，适用于各种类型的 DMA 控制器和设备。它包括了 DMA 预处理、DMA 描述符管理、DMA 操作执行等函数。DMA 设备 API 是专门针对某种类型的 DMA 控制器和设备实现的，可以通过 DMA 引擎框架进行管理。

4. DMA 缓存管理

DMA 缓存管理是为了解决 DMA 传输中的缓存一致性问题。由于 CPU 与 DMA 控制器之间使用的是不同的总线，因此存在缓存数据不一致的情况。

DMA 缓存管理机制通过 dma\_map\_single() 和 dma\_map\_sg() 函数将内存映射到 DMA 控制器中，以确保缓存数据的一致性。在 DMA 传输完成后，使用 dma\_unmap\_single() 和 dma\_unmap\_sg() 函数将内存解除映射，以便 CPU 可以访问缓存数据。

除了使用 dma\_map\_single() 和 dma\_map\_sg() 函数，还可以使用 dma\_sync\_single\_for\_device() 和 dma\_sync\_single\_for\_cpu() 函数来实现缓存一致性。这两个函数分别用于将 DMA 写入的数据同步到内存中和将内存中的数据同步到 DMA 读取的缓存中，以确保数据一致性。

5. DMA 内存分配

DMA 内存分配是指在 DMA 传输过程中为数据分配内存。由于 DMA 控制器需要直接访问内存，因此必须分配与硬件平台相适应的内存。在 Linux 系统中，可以使用 dma\_alloc\_coherent() 函数分配 DMA 内存。dma\_alloc\_coherent() 函数分配连续的物理内存，确保 DMA 控制器能够访问这些内存。

6. DMA 原语

DMA 原语是用于实现 DMA 操作的基本函数。在 Linux 系统中，DMA 原语包括 DMA memcpy、DMA memset 和 DMA xor 等。DMA memcpy 用于在内存和 DMA 控制器之间传输数据，DMA memset 用于将内存或 DMA 控制器中的数据初始化为指定的值，DMA xor 用于计算校验和等。

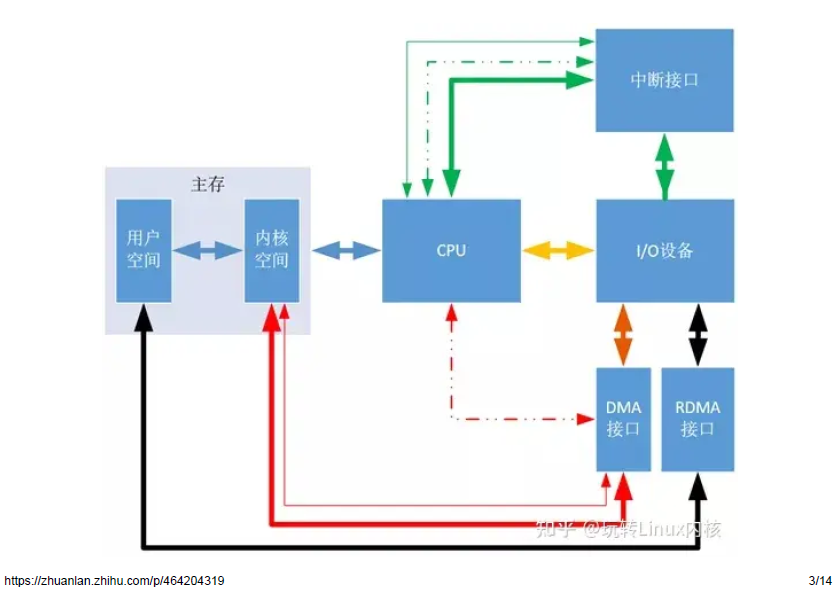
7. DMA 测试工具

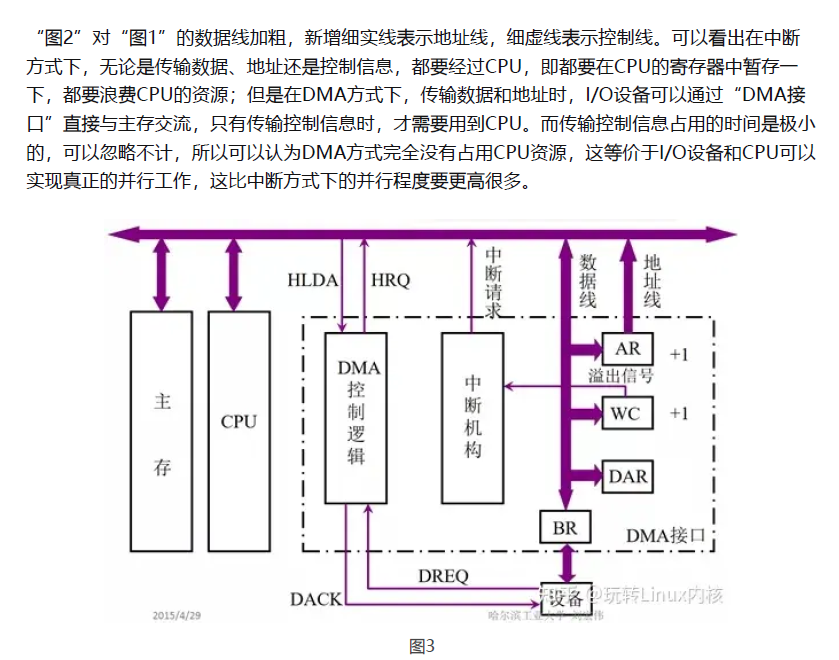
在进行 DMA 相关的开发和调试时，需要使用 DMA 测试工具来验证 DMA 的正确性和性能。Linux 内核提供了多个 DMA 测试工具，包括 dma-test、dmaengine-test 和 dmatest 等。这些工具可以测试 DMA 传输的正确性、性能和稳定性，以及验证 DMA 控制器驱动程序和设备驱动程序的正确性。

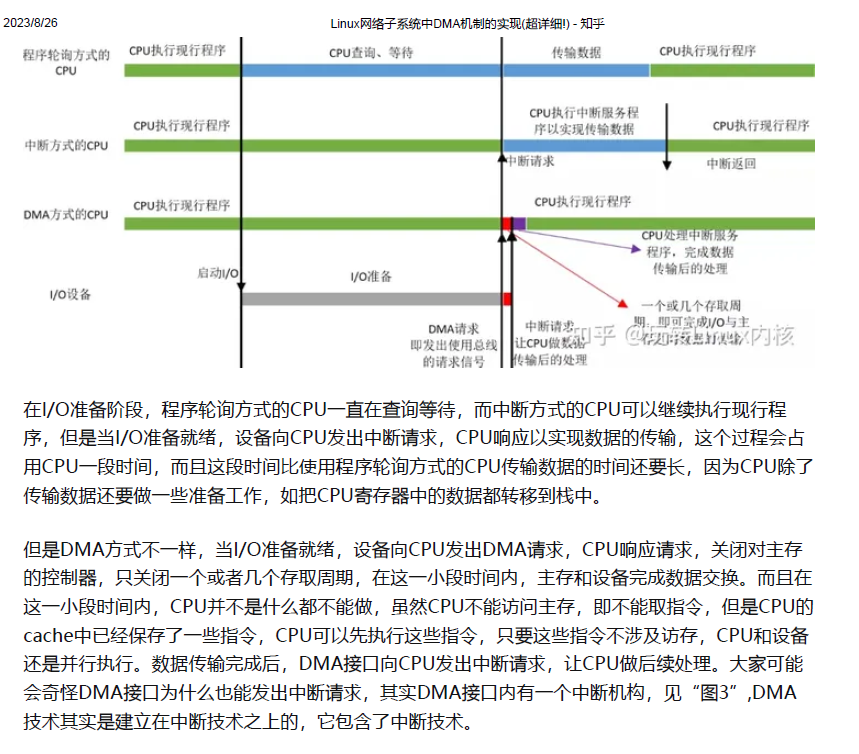
总之，Linux 系统中的 DMA 实现涉及多个方面，包括 DMA 控制器驱动程序、设备驱动程序、DMA 引擎框架、DMA 缓存管理、DMA 内存分配、DMA 原语和 DMA 测试工具等。在开发 DMA 相关的应用程序时，需要对这些方面有一定的了解，并根据硬件平台的不同进行适当的调整和修改。

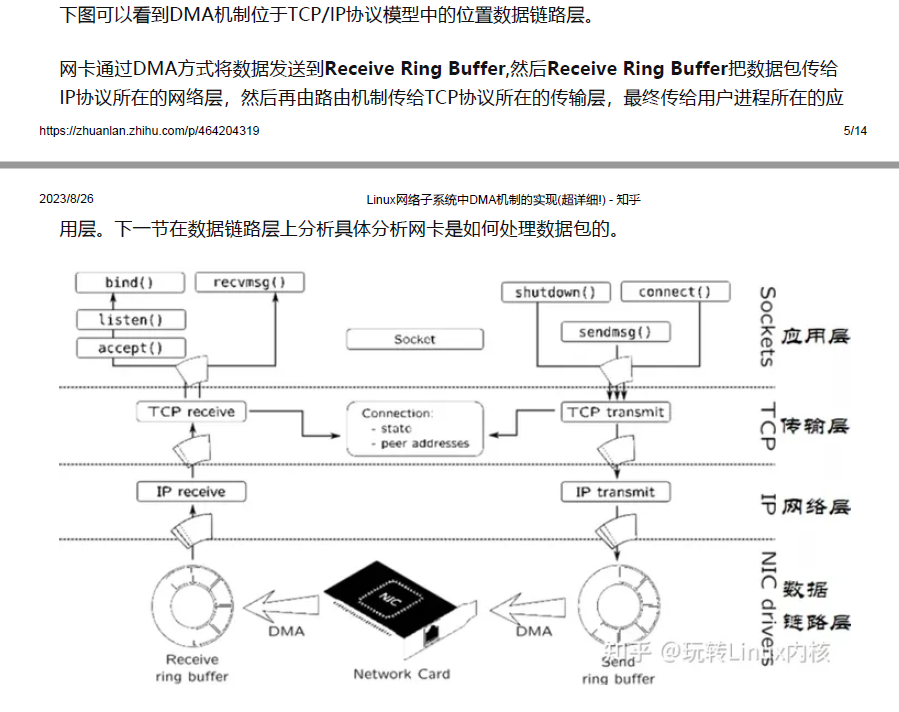
## Linux网络子系统中DMA机制的实现

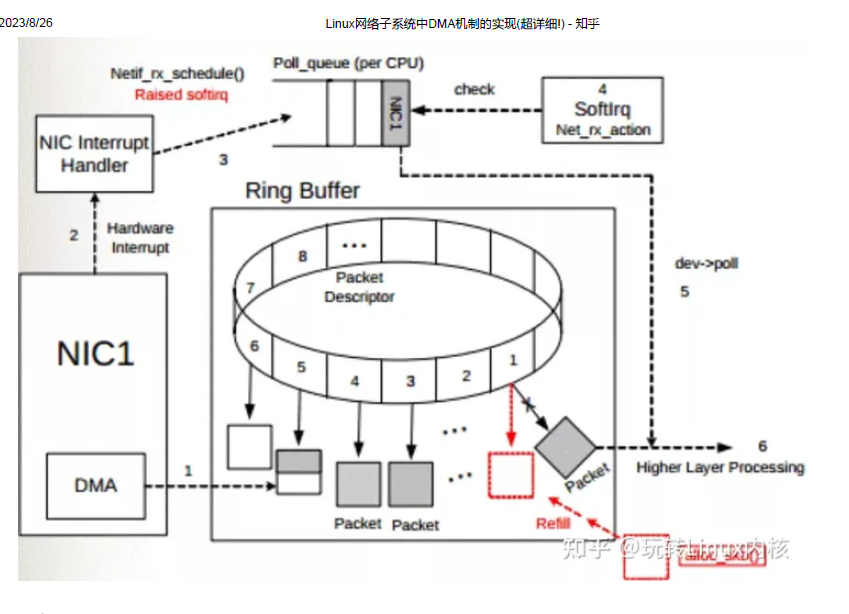
<https://zhuanlan.zhihu.com/p/464204319>











## 简介Linux DMA 功能及接口分析

<https://www.21ic.com/article/874931.html>

## Linux 下的DMA浅析

<https://blog.csdn.net/zqixiao_09/article/details/51089088>







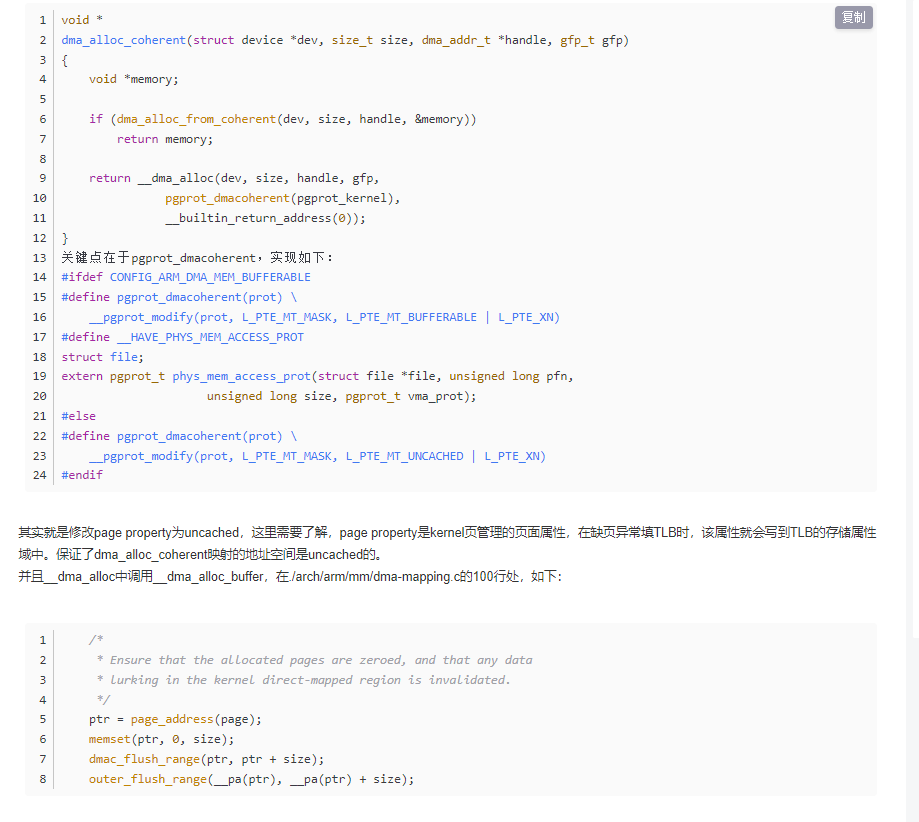


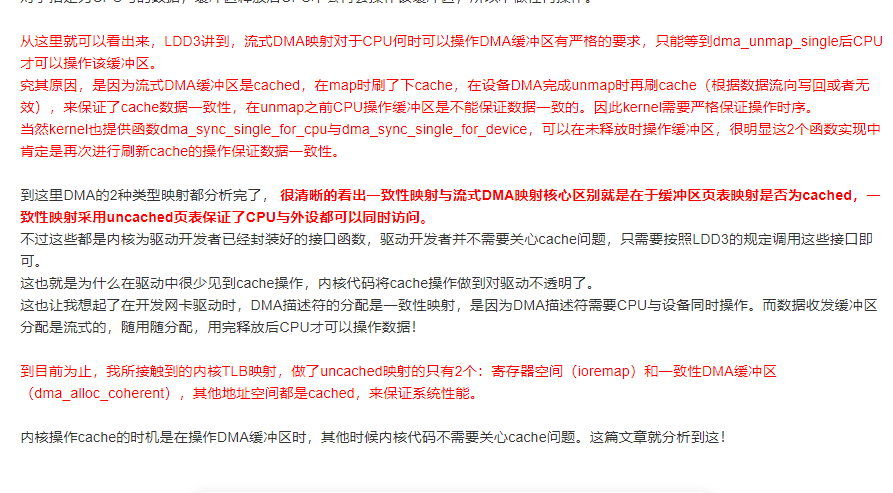


## kernel如何保证cache数据一致性

<https://blog.csdn.net/skyflying2012/article/details/48023447>

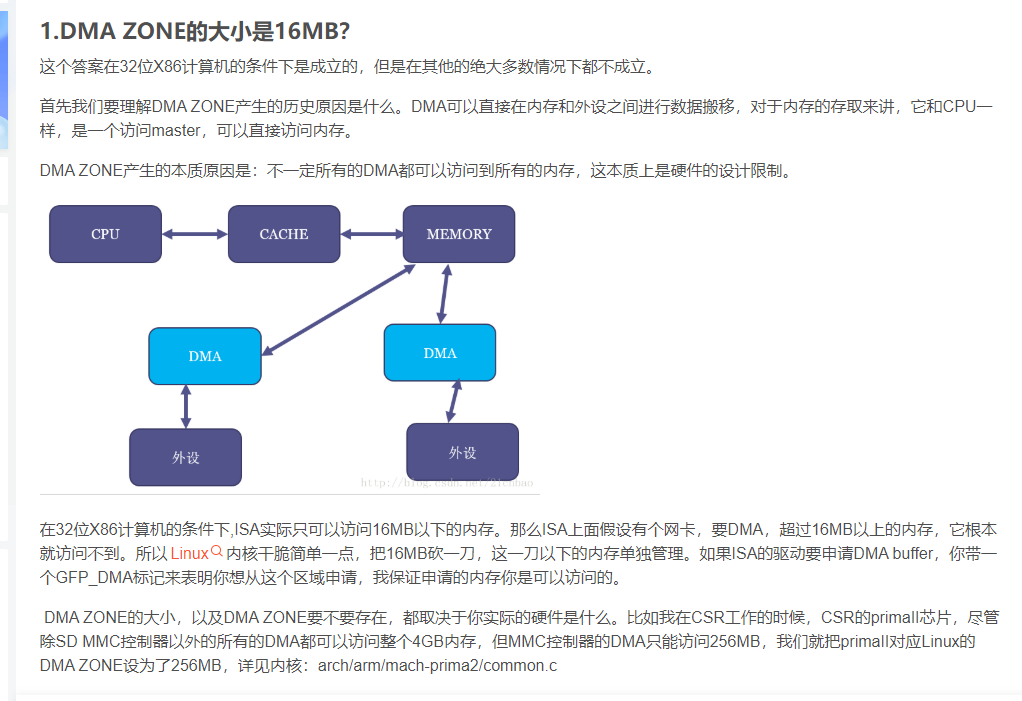




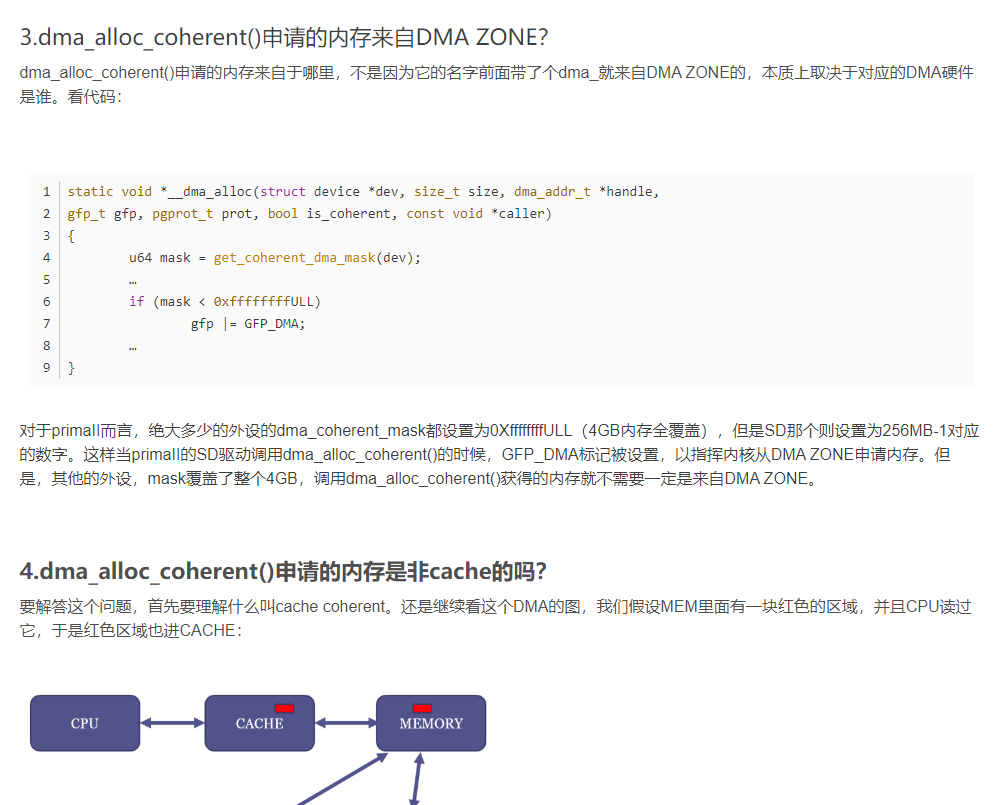


## 关于linux内存管理中DMA ZONE和dma\_alloc\_coherent若干误解的澄清

<https://blog.csdn.net/21cnbao/article/details/79133658>











# Xilinx关于DMA Linux的资料

## AXI DMA test driver on Linux

<https://support.xilinx.com/s/question/0D52E00006iHlLgSAK/axi-dma-test-driver-on-linux?language=en_US>

## Using AXI DMA in petalinux

<https://support.xilinx.com/s/question/0D52E00006iHvGhSAK/using-axi-dma-in-petalinux?language=en_US>

# DMA测试程序

## dma-test

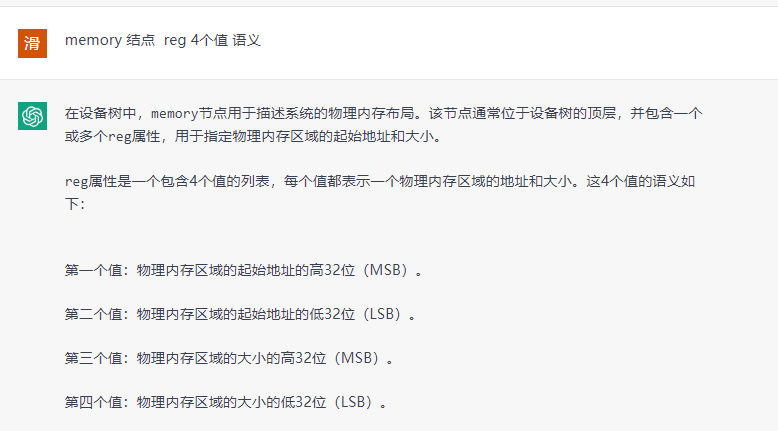
## dmaengine-test

## dmatest

# 关于取消预留内存cache的方法

## 设备树一些知识

### 设备树中memory结点的reg语义



### 对于 xlnx,zynqmp reserved-memory结点的 address-cells size-cells 如何填

对于xlnx,zynqmp平台，reserved-memory节点的address-cells和size-cells属性应该设置为2。这是因为xlnx,zynqmp平台使用两个单元来表示地址和大小。

以下是一个示例的reserved-memory节点的配置：

复制代码

reserved-memory {

#address-cells = <2>;

#size-cells = <2>;

ranges;

ramoops\_reserved: ramoops\_reserved@<start\_address> {

compatible = "ramoops";

reg = <0x0 0x<start\_address> 0x0 0x<size>>;

record-size = <0x<record\_size>>;

console-size = <0x<console\_size>>;

ftrace-size = <0x<ftrace\_size>>;

};

};

在上述示例中，#address-cells和#size-cells属性都设置为2，以表示地址和大小使用两个单元。

请注意，<start\_address>和<size>的值应该根据实际情况进行替换，并且其他属性（如record-size、console-size和ftrace-size）也需要根据需求进行设置。

完成设备树的配置后，重新生成设备树并重新构建PetaLinux项目，以使配置生效。

### 关于xlnx,zynqmp的dma配置



# 内核编程相关

## platform\_get\_resource

设备树 如何定义

在设备树中，可以使用platform\_get\_resource函数来获取设备的资源信息。为了正确定义设备的资源，您需要在设备树中为设备节点添加相应的属性。

以下是一个示例，展示了如何在设备树中定义设备的资源：

c复制代码

my\_device {

compatible = "my\_device";

reg = <0x10000000 0x1000>; // 设备的寄存器地址和大小

interrupts = <0 29 4>; // 设备的中断号

};

在上述示例中，my\_device节点表示一个设备，它具有以下资源信息：

* compatible属性指定设备的兼容性字符串，用于匹配设备驱动程序。
* reg属性指定设备的寄存器地址和大小。第一个值表示寄存器的起始地址，第二个值表示寄存器的大小。
* interrupts属性指定设备的中断号。第一个值表示中断控制器的编号，第二个值表示中断号，第三个值表示中断触发类型。

在设备驱动程序中，您可以使用platform\_get\_resource函数来获取设备的资源信息。以下是一个示例：

c复制代码

struct resource \*res;

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

if (!res) {

dev\_err(&pdev->dev, "Failed to get memory resource\n");

return -ENODEV;

}

// 使用获取到的资源信息进行操作

unsigned long start = res->start;

unsigned long size = resource\_size(res);

在上述示例中，platform\_get\_resource函数用于获取设备的内存资源信息。第一个参数是指向设备结构体的指针，第二个参数是资源类型（例如IORESOURCE\_MEM表示内存资源），第三个参数是资源的索引（如果设备有多个资源，可以使用不同的索引来获取不同的资源）。函数返回一个指向资源结构体的指针，您可以使用该指针获取资源的起始地址和大小。

请注意，上述示例仅展示了如何在设备树中定义设备的资源和如何在设备驱动程序中获取资源信息。实际的设备树和设备驱动程序的定义可能会根据具体的设备和需求而有所不同。

## ioremap\_nocache

ioremap\_nocache()是Linux内核提供的一个函数，用于将物理地址映射到内核虚拟地址空间，并且指定映射的内存区域不使用缓存。

在某些情况下，需要直接访问设备的物理内存，而不经过缓存。这可能是因为设备的内存区域是与其他设备共享的，或者需要绕过缓存以确保对设备的访问是实时的。在这种情况下，可以使用ioremap\_nocache()函数来创建一个不使用缓存的内核虚拟地址，以便直接访问设备的物理内存。

使用ioremap\_nocache()函数的步骤如下：

1. 引入头文件：#include <asm/io.h>
2. 调用ioremap\_nocache()函数，传入物理地址和映射的大小作为参数。函数原型如下：

c复制代码

void \_\_iomem \*ioremap\_nocache(resource\_size\_t phys\_addr, unsigned long size);

* + phys\_addr：要映射的物理地址。
  + size：映射的大小。

1. 函数返回一个void \_\_iomem类型的指针，表示映射后的内核虚拟地址。
2. 使用返回的内核虚拟地址进行直接访问设备的物理内存。

需要注意的是，使用ioremap\_nocache()函数映射的内存区域是不使用缓存的，因此对该内存区域的读写操作会直接访问设备的物理内存。这意味着需要谨慎处理对该内存区域的访问，以避免对设备的操作产生意外的影响。同时，由于绕过了缓存，对该内存区域的访问可能会导致性能上的损失，因此需要权衡使用的需求。

## 应用程序使用mmap从共享内存读数据时，如何作到nocache

在应用程序使用mmap()从共享内存读取数据时，如果需要确保读取的数据是不经过缓存的，可以通过以下方式实现：

1. 使用MAP\_UNCACHED标志：在调用mmap()函数时，可以使用MAP\_UNCACHED标志来指定映射的内存区域不使用缓存。这样可以确保对共享内存的读取操作是直接从物理内存中进行的，绕过了缓存。

c复制代码

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

* + flags参数：可以使用MAP\_UNCACHED标志来指定映射的内存区域不使用缓存。

1. 使用madvise()函数：在调用mmap()函数后，可以使用madvise()函数来设置内存区域的建议行为。可以使用MADV\_DONTNEED标志来指定内存区域不需要被缓存。

c复制代码

int madvise(void \*addr, size\_t length, int advice);

* + advice参数：可以使用MADV\_DONTNEED标志来指定内存区域不需要被缓存。

需要注意的是，使用以上方式可以尽量避免缓存的影响，但并不能完全保证数据不经过缓存。因为缓存的行为是由硬件和操作系统控制的，具体的实现可能会有一定的差异。如果需要确保数据不经过缓存，可能需要考虑使用其他的机制，如使用特殊的DMA操作或直接访问物理内存等。

综上所述，通过使用MAP\_UNCACHED标志或madvise()函数，可以在应用程序使用mmap()从共享内存读取数据时尽量避免缓存的影响。但需要注意，具体的实现可能会受到硬件和操作系统的限制。



## dma\_mmap\_coherent()

dma\_mmap\_coherent is a function in the Linux kernel that is used to map a coherent DMA buffer into the user space.

Coherent DMA buffers are special memory regions that are used for direct memory access (DMA) operations. They are typically used by devices such as network cards or graphics cards to transfer data directly to or from the main memory without involving the CPU.

The dma\_mmap\_coherent function takes a DMA buffer and maps it into the user space, allowing user applications to directly access the buffer. This function ensures that the buffer is mapped in a way that is coherent with the CPU's cache, ensuring that any changes made by the CPU are immediately visible to the device performing the DMA operation.

The function returns a pointer to the mapped buffer in the user space, which can then be used by the application to read from or write to the buffer. The application can also use standard memory operations, such as memcpy, to copy data to or from the buffer.

Overall, dma\_mmap\_coherent provides a convenient way for user applications to access coherent DMA buffers, enabling efficient data transfer between devices and the CPU.

## [Linux kernel中的list怎么使用](https://www.cnblogs.com/hwy89289709/p/6754300.html)

需要使用链表的时候, 就想到直接使用Linux kernel 里面的 list 了. 于是找到一片文章, 相当于翻译这篇文章. 学习怎么用的笔记

原文地址: http://isis.poly.edu/kulesh/stuff/src/klist/

原文日期: This is a working copy, last updated on April 5th, 2005. Feel free to email your comments.

我看完之后,总结这些函数 的用处,以后方便的立刻使用, 像知道为什么, 就往下看哦

 struct list\_head : 链表结构体的指针的数据结构

LIST\_HEAD\_INIT: 用于初始化这个头部编成独立结点

INIT\_LIST\_HEAD(ptr) : 将这个指针初始化为孤立的结点

LIST\_HEAD(name) : 初始化一个变量为name, 并且初始化

list\_add(struct list\_head \*new, struct list\_head \*head): 将新元素, 放到了链表的头端

 void list\_add\_tail(struct list\_head \*new, struct list\_head \*head): 添加一个到尾部

list\_del(struct list\_head \*entry): 删除这个结点, 没有切断联系

list\_del\_init(struct list\_head \*entry) : 删除这个元素并且初始化

list\_move(struct list\_head \*list, struct list\_head \*head): 将第一个链表的头删, 然后连接到第二个链表.

list\_move\_tail(struct list\_head \*list, struct list\_head \*head): 将第一个链表移动到, 到第二个链表后面

list\_empty(struct list\_head \*head): 判断这个链表是否为空

list\_splice(struct list\_head \*list, struct list\_head \*head): 将第一个链表接到第二个链表的开头.

list\_splice\_init(struct list\_head \*list, struct list\_head \*head): 在上面的功能之上, 再将第一个头结点初始化

list\_entry(ptr, type, member): 获得这个指针的所再的项的开头

list\_for\_each(pos, head): 从前到后遍历链表

list\_for\_each\_prev(pos, head): 从后到前遍历链表

list\_for\_each\_safe(pos, n, head): 删除当前结点不会造成断链

list\_for\_each\_entry(pos, head, member): 这个可以自己少写一句list\_entry

 list\_for\_each\_entry\_safe(pos, n, head, member): 遍历可以安全删除

# 结束