in drivers/base/dd.c/driver_probe_device()

```
ret = really_probe(dev, drv);
```

```
static int really_probe(struct device *dev, struct device_driver *drv)
{
    int ret = 0;
    int local_trigger_count = atomic_read(&deferred_trigger_count);     ⑧

    atomic_inc(&probe_count);
    pr_debug("bus: '%s': %s: probing driver %s with device %s\n",
         drv->bus->name, __func__, drv->name, dev_name(dev));
    WARN_ON(!list_empty(&dev->devres_head));

    dev->driver = drv;

    /* If using pinctrl, bind pins now before probing */
    ret = pinctrl_bind_pins(dev);       ①
    if (ret)
        goto probe_failed;

    if (driver_sysfs_add(dev)) {
        printk(KERN_ERR "%s: driver_sysfs_add(%s) failed\n",
            __func__, dev_name(dev));
        goto probe_failed;
    }

    if (dev->bus->probe) {              ②
        ret = dev->bus->probe(dev);
        if (ret)
            goto probe_failed;
    } else if (drv->probe) {            ③
        ret = drv->probe(dev);
        if (ret)
            goto probe_failed;
    }

    driver_bound(dev);
    ret = 1;
    pr_debug("bus: '%s': %s: bound device %s to driver %s\n",
         drv->bus->name, __func__, dev_name(dev), drv->name);
    goto done;

probe_failed:
    devres_release_all(dev);            ④
    driver_sysfs_remove(dev);
    dev->driver = NULL;
    dev_set_drvdata(dev, NULL);

    if (ret == -EPROBE_DEFER) {         ⑤
        /* Driver requested deferred probing */
        dev_info(dev, "Driver %s requests probe deferral\n", drv->name);
        driver_deferred_probe_add(dev); ⑥
        /* Did a trigger occur while probing? Need to re-trigger if yes */
        if (local_trigger_count != atomic_read(&deferred_trigger_count))
            driver_deferred_probe_trigger();    ⑦
    } else if (ret != -ENODEV && ret != -ENXIO) {
        /* driver matched but the probe failed */
        printk(KERN_WARNING
               "%s: probe of %s failed with error %d\n",
               drv->name, dev_name(dev), ret);
    } else {
        pr_debug("%s: probe of %s rejects match %d\n",
               drv->name, dev_name(dev), ret);
    }
    /*
     * Ignore errors returned by ->probe so that the next driver can try
     * its luck.
     */
    ret = 0;
done:
    atomic_dec(&probe_count);
    wake_up(&probe_waitqueue);
    return ret;
}
```

device与driver就在这里相结合。device与driver实行的是一夫多妻的婚姻。driver是丈夫,device是妻子。

①

device相关的pin configuration就是在这里起作用的。

②③

如果device所在的bus定义了probe()，则优先由其来probe之；其次则是driver提供的probe()来匹配device。

④

device与driver probe失败的情况下，由managed resource API申请的resource在这里会被释放，无需driver的
probe()显示释放。

⑤

如果返回的是-EPROBE_DEFER，表示该driver的probe()并不认为失败了，可能是由于它所依赖的resource还未初始化好，希望在伺候能再次
probe。

⑥

```
static void driver_deferred_probe_add(struct device *dev)
{
    mutex_lock(&deferred_probe_mutex);
    if (list_empty(&dev->p->deferred_probe)) {
        dev_dbg(dev, "Added to deferred list\n");
        list_add_tail(&dev->p->deferred_probe, &deferred_probe_pending_list);
    }
    mutex_unlock(&deferred_probe_mutex);
}
```

把当前处理的device放入deferred_probe_pending_list　global list中。

⑦
如果从函数进入(⑧)到现在，deferred_trigger_count被其他thread运行的
driver_deferred_probe_trigger()
修改了。
可能改变deferred_trigger_count global count的只有一个函数。

触发work queue，`deferred_wq`。

```c
/**
 * driver_deferred_probe_trigger() - Kick off re-probing deferred devices
 *
 * This functions moves all devices from the pending list to the active
 * list and schedules the deferred probe workqueue to process them.  It
 * should be called anytime a driver is successfully bound to a device.
 *
 * Note, there is a race condition in multi-threaded probe. In the case where
 * more than one device is probing at the same time, it is possible for one
 * probe to complete successfully while another is about to defer. If the second
 * depends on the first, then it will get put on the pending list after the
 * trigger event has already occured and will be stuck there.
 *
 * The atomic 'deferred_trigger_count' is used to determine if a successful
 * trigger has occurred in the midst of probing a driver. If the trigger count
 * changes in the midst of a probe, then deferred processing should be triggered
 * again.
 */
static void driver_deferred_probe_trigger(void)
{
	if (!driver_deferred_probe_enable)
		return;

	/*
	 * A successful probe means that all the devices in the pending list
	 * should be triggered to be reprobed.  Move all the deferred devices
	 * into the active list so they can be retried by the workqueue
	 */
	mutex_lock(&deferred_probe_mutex);
	atomic_inc(&deferred_trigger_count);
	list_splice_tail_init(&deferred_probe_pending_list,
			      &deferred_probe_active_list);
	mutex_unlock(&deferred_probe_mutex);

	/*
	 * Kick the re-probe thread.  It may already be scheduled, but it is
	 * safe to kick it again.
	 */
	queue_work(deferred_wq, &deferred_probe_work);
}
```

触发work queue，`deferred_wq`。

---

deferred_wq执行的是如下function

```c
/*
 * deferred_probe_work_func() - Retry probing devices in the active list.
 */
static void deferred_probe_work_func(struct work_struct *work)
{
    struct device *dev;
    struct device_private *private;
    /*
     * This block processes every device in the deferred 'active' list.
     * Each device is removed from the active list and passed to
     * bus_probe_device() to re-attempt the probe.  The loop continues
     * until every device in the active list is removed and retried.
     *
     * Note: Once the device is removed from the list and the mutex is
     * released, it is possible for the device get freed by another thread
     * and cause a illegal pointer dereference.  This code uses
     * get/put_device() to ensure the device structure cannot disappear
     * from under our feet.
     */
    mutex_lock(&deferred_probe_mutex);
    while (!list_empty(&deferred_probe_active_list)) {
        private = list_first_entry(&deferred_probe_active_list,
                    typeof(*dev->p), deferred_probe);
        dev = private->device;
        list_del_init(&private->deferred_probe);

        get_device(dev);

        /*
         * Drop the mutex while probing each device; the probe path may
         * manipulate the deferred list
         */
        mutex_unlock(&deferred_probe_mutex);

        /*
         * Force the device to the end of the dpm_list since
         * the PM code assumes that the order we add things to
         * the list is a good order for suspend but deferred
         * probe makes that very unsafe.
         */
        device_pm_lock();
        device_pm_move_last(dev);
        device_pm_unlock();

        dev_dbg(dev, "Retrying from deferred list\n");
        bus_probe_device(dev);

        mutex_lock(&deferred_probe_mutex);

        put_device(dev);
    }
    mutex_unlock(&deferred_probe_mutex);
}
```

要求defer probe的device都在deferred_probe_active_list global list上，该function所做的就是重新
让bus上的各个driver来依次probe该device。依然是由really_probe()来完成probe,也即指示时间上延迟了，但
步骤是一样的。bus->probe优先。