device private

```
struct device {
    struct device_private   *p;     ①
};

/**
 * struct device_private - structure to hold the private to the driver core portions of the device structure.
 *
 * @klist_children - klist containing all children of this device
 * @knode_parent - node in sibling list
 * @knode_driver - node in driver list
 * @knode_bus - node in bus list
 * @deferred_probe - entry in deferred_probe_list which is used to retry the
 *  binding of drivers which were unable to get all the resources needed by
 *  the device; typically because it depends on another driver getting
 *  probed first.
 * @device - pointer back to the struct class that this structure is
 * associated with.
 *
 * Nothing outside of the driver core should ever touch these fields.
 */
struct device_private {
    struct klist klist_children;    ③
    struct klist_node knode_parent; ④
    struct klist_node knode_driver; ⑤
    struct klist_node knode_bus;    ⑥
    struct list_head deferred_probe;⑦
    struct device *device;          ②
};
```

①

p指向该device相关的 `device_private`

②

device指向该 `private` 对应的 `struct device`

③

该device的child device都被链接在klist_children   klist上

in drivers/base/core.c/device_add()

```
    parent = get_device(dev->parent);

......

    if (parent)
        klist_add_tail(&dev->p->knode_parent,
                   &parent->p->klist_children);
```

dev就是当前要add的struct device。把代表当前dev的dev->p->knode_parent添加到parent的管理child device
的parent->p->klist_children klist中。

in drivers/base/core.c/device_for_each_child()

```c
/**
 * device_for_each_child - device child iterator.
 * @parent: parent struct device.
 * @fn: function to be called for each device.
 * @data: data for the callback.
 *
 * Iterate over @parent's child devices, and call @fn for each,
 * passing it @data.
 *
 * We check the return of @fn each time. If it returns anything
 * other than 0, we break out and return that value.
 */
int device_for_each_child(struct device *parent, void *data,
              int (*fn)(struct device *dev, void *data))
{
    struct klist_iter i;
    struct device *child;
    int error = 0;

    if (!parent->p)
        return 0;

    klist_iter_init(&parent->p->klist_children, &i);
    while ((child = next_device(&i)) && !error)
        error = fn(child, data);
    klist_iter_exit(&i);
    return error;
}
EXPORT_SYMBOL_GPL(device_for_each_child);

/**
 * device_find_child - device iterator for locating a particular device.
 * @parent: parent struct device
 * @match: Callback function to check device
 * @data: Data to pass to match function
 *
 * This is similar to the device_for_each_child() function above, but it
 * returns a reference to a device that is 'found' for later use, as
 * determined by the @match callback.
 *
 * The callback should return 0 if the device doesn't match and non-zero
 * if it does.  If the callback returns non-zero and a reference to the
 * current device can be obtained, this function will return to the caller
 * and not iterate over any more devices.
 *
 * NOTE: you will need to drop the reference with put_device() after use.
 */
struct device *device_find_child(struct device *parent, void *data,
                 int (*match)(struct device *dev, void *data))
{
    struct klist_iter i;
    struct device *child;

    if (!parent)
        return NULL;

    klist_iter_init(&parent->p->klist_children, &i);
    while ((child = next_device(&i)))
        if (match(child, data) && get_device(child))
            break;
    klist_iter_exit(&i);
    return child;
}
EXPORT_SYMBOL_GPL(device_find_child);
```

④
把该 `struct device` 挂在parent的klist_children klist上。

⑤
knode_driver把该device挂到struct device_driver的device klist上

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

```
static void driver_bound(struct device *dev)
{
    if (klist_node_attached(&dev->p->knode_driver)) {          (A)
        printk(KERN_WARNING "%s: device %s already bound\n",
            __func__, kobject_name(&dev->kobj));
        return;
    }

    pr_debug("driver: '%s': %s: bound to device '%s'\n", dev->driver->name,
         __func__, dev_name(dev));

    klist_add_tail(&dev->p->knode_driver, &dev->driver->p->klist_devices);   (B)

    /*
     * Make sure the device is no longer in one of the deferred lists and
     * kick off retrying all pending devices
     */
    driver_deferred_probe_del(dev);
    driver_deferred_probe_trigger();

    if (dev->bus)
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,
                         BUS_NOTIFY_BOUND_DRIVER, dev);
}
```

(A)
首先判断该dev是否已经 `bound` 了，即&dev->p->knode_driver node是否是独立的还是已经被链接在某个klist上了?

(B)

&dev->driver->p->klist_devices是该driver管理所有device的klist
把当前device添加到链上。

```c
/**
 * driver_for_each_device - Iterator for devices bound to a driver.
 * @drv: Driver we're iterating.
 * @start: Device to begin with
 * @data: Data to pass to the callback.
 * @fn: Function to call for each device.
 *
 * Iterate over the @drv's list of devices calling @fn for each one.
 */
int driver_for_each_device(struct device_driver *drv, struct device *start,
               void *data, int (*fn)(struct device *, void *))
{
    struct klist_iter i;
    struct device *dev;
    int error = 0;

    if (!drv)
        return -EINVAL;

    klist_iter_init_node(&drv->p->klist_devices, &i,
             start ? &start->p->knode_driver : NULL);
    while ((dev = next_device(&i)) && !error)
        error = fn(dev, data);
    klist_iter_exit(&i);
    return error;
}
EXPORT_SYMBOL_GPL(driver_for_each_device);
```

⑥
每个struct device除了被链接在管理它的driver的device list上，它还属于某种bus的device，所以它也要被链接在bus所管理的device
klist上。

in drives/base/bus.c/bus_add_device()

```c
/**
 * bus_add_device - add device to bus
 * @dev: device being added
 *
 * - Add device's bus attributes.
 * - Create links to device's bus.
 * - Add the device to its bus's list of devices.
 */
int bus_add_device(struct device *dev)
{
    struct bus_type *bus = bus_get(dev->bus);
    int error = 0;

    if (bus) {
        pr_debug("bus: '%s': add device %s\n", bus->name, dev_name(dev));
        error = device_add_attrs(bus, dev);
        if (error)
            goto out_put;
        error = device_add_groups(dev, bus->dev_groups);
        if (error)
            goto out_groups;
        error = sysfs_create_link(&bus->p->devices_kset->kobj,
                        &dev->kobj, dev_name(dev));
        if (error)
            goto out_id;
        error = sysfs_create_link(&dev->kobj,
                &dev->bus->p->subsys.kobj, "subsystem");
        if (error)
            goto out_subsys;
        klist_add_tail(&dev->p->knode_bus, &bus->p->klist_devices); (A)
    }
    return 0;

out_subsys:
    sysfs_remove_link(&bus->p->devices_kset->kobj, dev_name(dev));
out_groups:
    device_remove_groups(dev, bus->dev_groups);
out_id:
    device_remove_attrs(bus, dev);
out_put:
    bus_put(dev->bus);
    return error;
}
```

(A)

&dev->p->knode_bus node

&bus->p->klist_devices

```c
struct bus_type {
    struct subsys_private *p;
};

struct subsys_private {
    struct klist klist_devices;
};
```

⑦
device有时候会延迟probe

deferred_probe node可以位于下面的list中

in drivers/base/dd.c

> static LIST_HEAD(deferred_probe_pending_list);
>
> static LIST_HEAD(deferred_probe_active_list);

---

driver private

```c
struct device_driver {
    struct driver_private *p;        ①
};

struct driver_private {
    struct kobject kobj;
    struct klist klist_devices;      ③
    struct klist_node knode_bus;     ④
    struct module_kobject *mkobj;
    struct device_driver *driver;    ②
};
```

①
指向该driver的 `private`

②
由 `private` 反指向对应的driver

③
一个driver可以管理多个device，用该klist管理

④
bus除了管理着所有属于它的device，还管理着属于该bus的所有driver。

```
struct bus_type {
    struct subsys_private *p;
};

struct subsys_private {
    struct klist klist_drivers;
};
```

in drivers/base/bus.c/bus_add_driver()

> klist_add_tail(&priv->knode_bus, &bus->p->klist_drivers);

---

bus and class private

```c
struct bus_type {
    struct subsys_private *p;       ①
};

struct class {
    struct subsys_private *p;       ②
};
/**
 * struct subsys_private - structure to hold the private to the driver core portions of the bus_type/class structure.
 *
 * @subsys - the struct kset that defines this subsystem
 * @devices_kset - the subsystem's 'devices' directory
 * @interfaces - list of subsystem interfaces associated
 * @mutex - protect the devices, and interfaces lists.
 *
 * @drivers_kset - the list of drivers associated
 * @klist_devices - the klist to iterate over the @devices_kset
 * @klist_drivers - the klist to iterate over the @drivers_kset
 * @bus_notifier - the bus notifier list for anything that cares about things
 *                  on this bus.
 * @bus - pointer back to the struct bus_type that this structure is associated
 *         with.
 *
 * @glue_dirs - "glue" directory to put in-between the parent device to
 *              avoid namespace conflicts
 * @class - pointer back to the struct class that this structure is associated
 *           with.
 *
 * This structure is the one that is the actual kobject allowing struct
 * bus_type/class to be statically allocated safely.  Nothing outside of the
 * driver core should ever touch these fields.
 */
struct subsys_private {
    struct kset subsys;
    struct kset *devices_kset;
    struct list_head interfaces;
    struct mutex mutex;

    struct kset *drivers_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;       ③

    struct kset glue_dirs;
    struct class *class;        ④
};
```

①

②

bus和class共享一个private

③

④

由`priavte`反指向对应的bu和class