Linux common clock framework(2)_clock provider

作者：wowo 发布于：2014-10-23 23:49 分类：电源管理子系统

1. 前言

本文接上篇文章，从 clock driver 的角度，分析怎么借助 common clock framework 管理系统的时钟资源。换句话说，就是怎么编写一个 clock driver。
由于 kernel 称 clock driver 为 clock provider（相应的，clock 的使用者为 clock consumer），因此本文遵循这个规则，统一以 clock provider 命名。

2. clock 有关的 DTS

我们在“Linux common clock framework(1)_概述”中讲述 clock consumer 怎么使用 clock 时，提到过 clock consumer 怎么在 DTS 中指定所使用的 clock。这里再做进一步说明。

2.1 clock provider 的 DTS

我们知道，DTS（Device Tree Source）是用来描述设备信息的，那系统的 clock 资源，是什么设备呢？换句话，用什么设备表示呢？这决定了 clock provider 的 DTS 怎么写。
通常有两种方式：
方式 1，将系统所有的 clock，抽象为一个虚拟的设备，用一个 DTS node 表示。这个虚拟的设备称作 clock controller，参考如下例子：

```
1: /* arch/arm/boot/dts/exynos4210.dtsi */
2: clock: clock-controller@0x10030000 {
3:         compatible = "samsung,exynos4210-clock";
4:         reg = <0x10030000 0x20000>;
5:         #clock-cells = <1>;
6: };
```

clock，该 clock 设备的名称，clock consumer 可以根据该名称引用 clock；
#clock-cells，该 clock 的 cells，1 表示该 clock 有多个输出，clock consumer 需要通过 ID 值指定所要使用的 clock（很好理解，系统那么多 clock，被抽象为 1 个设备，因而需要额外的 ID 标识）。
方式 2，每一个可输出 clock 的器件，如“Linux common clock framework(1)_概述”所提及的 Oscillator、PLL、Mux 等等，都是一个设备，用一个 DTS node 表示。每一个器件，即是 clock provider，也是 clock consumer（根节点除外，如 OSC），因为它需要接受 clock 输入，经过处理后，输出 clock。参考如下例子（如果能拿到对应的 datasheet 会更容易理解）：

```
1: /* arch/arm/boot/dts/sun4i-a10.dtsi */
2: clocks {
3:     #address-cells = <1>;
4:     #size-cells = <1>;
5:     ranges;
7:     /*
8:      * This is a dummy clock, to be used as placeholder on
9:      * other mux clocks when a specific parent clock is not
10:     * yet implemented. It should be dropped when the driver
11:     * is complete.
12:     */
13:    dummy: dummy {
```

```
14:         #clock-cells = <0>;
15:         compatible = "fixed-clock";
16:         clock-frequency = <0>;
17:     };
18:
19:     osc24M: osc24M@01c20050 {
20:         #clock-cells = <0>;
21:         compatible = "allwinner,sun4i-osc-clk";
22:         reg = <0x01c20050 0x4>;
23:         clock-frequency = <24000000>;
24:     };
25:
26:     osc32k: osc32k {
27:         #clock-cells = <0>;
28:         compatible = "fixed-clock";
29:         clock-frequency = <32768>;
30:     };
31:
32:     pll1: pll1@01c20000 {
33:         #clock-cells = <0>;
34:         compatible = "allwinner,sun4i-pll1-clk";
35:         reg = <0x01c20000 0x4>;
36:         clocks = <&osc24M>;
37:     };
38:
39:     /* dummy is 200M */
40:     cpu: cpu@01c20054 {
41:         #clock-cells = <0>;
42:         compatible = "allwinner,sun4i-cpu-clk";
43:         reg = <0x01c20054 0x4>;
44:         clocks = <&osc32k>, <&osc24M>, <&pll1>, <&dummy>;
45:     };
46:
47:     axi: axi@01c20054 {
48:         #clock-cells = <0>;
49:         compatible = "allwinner,sun4i-axi-clk";
50:         reg = <0x01c20054 0x4>;
51:         clocks = <&cpu>;
52:     };
53:
54:     axi_gates: axi_gates@01c2005c {
55:         #clock-cells = <1>;
56:         compatible = "allwinner,sun4i-axi-gates-clk";
57:         reg = <0x01c2005c 0x4>;
58:         clocks = <&axi>;
59:         clock-output-names = "axi_dram";
60:     };
61:
62:     ahb: ahb@01c20054 {
63:         #clock-cells = <0>;
```

```
64:         compatible = "allwinner,sun4i-ahb-clk";
65:         reg = <0x01c20054 0x4>;
66:         clocks = <&axi>;
67:     };
68:
69:     ahb_gates: ahb_gates@01c20060 {
70:         #clock-cells = <1>;
71:         compatible = "allwinner,sun4i-ahb-gates-clk";
72:         reg = <0x01c20060 0x8>;
73:         clocks = <&ahb>;
74:         clock-output-names = "ahb_usb0", "ahb_ehci0",
75:             "ahb_ohci0", "ahb_ehci1", "ahb_ohci1", "ahb_ss",
76:             "ahb_dma", "ahb_bist", "ahb_mmc0", "ahb_mmc1",
77:             "ahb_mmc2", "ahb_mmc3", "ahb_ms", "ahb_nand",
78:             "ahb_sdram", "ahb_ace",    "ahb_emac", "ahb_ts",
79:             "ahb_spi0", "ahb_spi1", "ahb_spi2", "ahb_spi3",
80:             "ahb_pata", "ahb_sata", "ahb_gps", "ahb_ve",
81:             "ahb_tvd", "ahb_tve0", "ahb_tve1", "ahb_lcd0",
82:             "ahb_lcd1", "ahb_csi0", "ahb_csi1", "ahb_hdmi",
83:             "ahb_de_be0", "ahb_de_be1", "ahb_de_fe0",
84:             "ahb_de_fe1", "ahb_mp", "ahb_mali400";
85:     };
86:
87:     apb0: apb0@01c20054 {
88:         #clock-cells = <0>;
89:         compatible = "allwinner,sun4i-apb0-clk";
90:         reg = <0x01c20054 0x4>;
91:         clocks = <&ahb>;
92:     };
93:
94:     apb0_gates: apb0_gates@01c20068 {
95:         #clock-cells = <1>;
96:         compatible = "allwinner,sun4i-apb0-gates-clk";
97:         reg = <0x01c20068 0x4>;
98:         clocks = <&apb0>;
99:         clock-output-names = "apb0_codec", "apb0_spdif",
100:             "apb0_ac97", "apb0_iis", "apb0_pio", "apb0_ir0",
101:             "apb0_ir1", "apb0_keypad";
102:     };
103:
104:     /* dummy is pll62 */
105:     apb1_mux: apb1_mux@01c20058 {
106:         #clock-cells = <0>;
107:         compatible = "allwinner,sun4i-apb1-mux-clk";
108:         reg = <0x01c20058 0x4>;
109:         clocks = <&osc24M>, <&dummy>, <&osc32k>;
110:     };
111:
112:     apb1: apb1@01c20058 {
113:         #clock-cells = <0>;
```

```
114:        compatible = "allwinner,sun4i-apb1-clk";
115:        reg = <0x01c20058 0x4>;
116:        clocks = <&apb1_mux>;
117:    };
118:
119:    apb1_gates: apb1_gates@01c2006c {
120:        #clock-cells = <1>;
121:        compatible = "allwinner,sun4i-apb1-gates-clk";
122:        reg = <0x01c2006c 0x4>;
123:        clocks = <&apb1>;
124:        clock-output-names = "apb1_i2c0", "apb1_i2c1",
125:            "apb1_i2c2", "apb1_can", "apb1_scr",
126:            "apb1_ps20", "apb1_ps21", "apb1_uart0",
127:            "apb1_uart1", "apb1_uart2", "apb1_uart3",
128:            "apb1_uart4", "apb1_uart5", "apb1_uart6",
129:            "apb1_uart7";
130:    };
131: };
```

osc24M 和 osc32k 是两个 root clock，因此只做 clock provider 功能。它们的 cells 均为 0，因为直接使用名字即可引用。另外，增加了“clock-frequency”自定义关键字，这样在板子使用的 OSC 频率改变时，如变为 12M，不需要重新编译代码，只需更改 DTS 的频率即可（这不正是 Device Tree 的核心思想吗！）。话说回来了，osc24M 的命名不是很好，如果频率改变，名称也得改吧，clock consumer 的引用也得改吧；
pll1 即是 clock provider（cell 为 0，直接用名字引用），也是 clock consumer（clocks 关键字，指定输入 clock 为“osc24M”）；
再看一个复杂一点的，ahb_gates，它是 clock provider（cell 为 1），通过 clock-output-names 关键字，描述所有的输出时钟。同时它也是 clock consumer（由 clocks 关键字可知输入 clock 为“ahb”）。需要注意的是，clock-output-names 关键字只为了方便 clock provider 编程方便（后面会讲），clock consumer 不能使用（或者可理解为不可见）；
也许您会问，这些 DTS 描述，怎么使用？怎么和代码关联起来？先不着急，我们慢慢看。

2.2 clock consumer 的 DTS

在 2.1 中的方法二，我们已经看到 clock consumer 的 DTS 了，因为很多 clock provider 也是 clock consumer。这里再举几个例子，做进一步说明。
例子 1（对应 2.1 中的方式 1，来自同一个 DTS 文件）：

```
1: /* arch/arm/boot/dts/exynos4210.dtsi */
2: mct@10050000 {
3:        compatible = "samsung,exynos4210-mct";
4:        ...
5:        clocks = <&clock 3>, <&clock 344>;
6:        clock-names = "fin_pll", "mct";
7:        ...
8: };
```

clocks，指明该设备的 clock 列表，clk_get 时，会以它为关键字，去 device_node 中搜索，以得到对应的 struct clk 指针；
clocks 需要指明的信息，由 clock provider 的“#clock-cells”规定：为 0 时，只需要提供一个 clock provider name（称作 phandle）；为 1 时，表示 phandle 有多个输出，则需要额外提供一个 ID，指明具体需要使用那个输出。这个例子直接用立即数表示，更好的做法是，将系统所有 clock 的 ID，定义在一个头文件中，而 DTS 可以包含这个头文件，如“clocks = <&clock CLK_SPI0>”；
clock-names，为 clocks 指定的那些 clock 分配一些易于使用的名字，driver 可以直接以名字为参数，get clock 的句柄（具体可参考“Linux common clock framework(1)_概述”中 clk_get 相关的接口描述）。

例子 2，如果 clock provider 的“#clock-cells”为 0，可直接引用该 clock provider 的名字，具体可参考 2.1 中的方式 2。
例子 3，2.1 中方式 2 有一个 clock provider 的名字为 apb0_gates，它的“#clock-cells”为 1，并通过 clock-output-names 指定了所有的输出 clock，那么，clock consumer 怎么引用呢？如下（2 和.1 中的方式 2，来自同一个 DTS 文件）：

```
1: /* arch/arm/boot/dts/sun4i-a10.dtsi */
2: soc@01c20000 {
3:         compatible = "simple-bus";
4:         ...
5:
6:         pio: pinctrl@01c20800 {
7:                 compatible = "allwinner,sun4i-a10-pinctrl";
8:                 reg = <0x01c20800 0x400>;
9:                 clocks = <&apb0_gates 5>;
10:                ...
11:        }
12: }
```

和例子 1 一样，指定 phandle 为“aph0_gates”，ID 为 5。

2.3 DTS 相关的讨论和总结

我们在上面提到了 clock provider 的两种 DTS 定义方式，哪一种好呢？
从规范化、条理性的角度，毫无疑问方式 2 是好的，它真正理解了 Device Tree 的精髓，并细致的执行。且可以利用很多 clock framework 的标准实现（后面会讲）。
而方式 1 的优点是，DTS 容易写，相应的 clock driver 也较为直观，只是注册一个一个 clock provider 即可，没有什么逻辑可言。换句话说，方式 1 比较懒。
后面的 API 描述，蜗蜗会着重从方式 2 的角度，因为这样才能体会到软件设计中的美学。
注 1：上面例子中用到了两个公司的代码，方式 1 是三星的，方式 2 是全志的。说实话，全志的代码写的真漂亮，一个默默无闻的白牌公司，比三星这种国际大公司强多了。从这里，我们可以看到中国科技业的未来，还是很乐观的。

3. clock provider 有关的 API 汇整

clock provider 的 API 位于 include/linux/clk_provider.h。

3.1 struct clk_hw

由“Linux common clock framework(1)_概述”可知，clock framework 使用 struct clk 结构抽象 clock，但该结构对 clock consumer 是透明的（不需要知道它的内部细节）。同样，struct clk 对 clock provider 也是透明的。framework 提供了 struct clk_hw 结构，从 clock provider 的角度，描述 clock，该结构的定义如下：

```
1: struct clk_hw {
2:         struct clk *clk;
3:         const struct clk_init_data *init;
4: };
```

clk，struct clk 指针，由 clock framework 分配并维护，并在需要时提供给 clock consumer 使用；
init，描述该 clock 的静态数据，clock provider 负责把系统中每个 clock 的静态数据准备好，然后交给 clock framework 的核心逻辑，剩下的事情，clock provider 就不用操心了。这个过程，就是 clock driver 的编写过程，简单吧？该静态数据的数据结构如下。

```
1: struct clk_init_data {
2:         const char           *name;
3:         const struct clk_ops   *ops;
4:         const char           **parent_names;
```

```
5:        u8                    num_parents;
6:        unsigned long           flags;
7: };
```

name，该 clock 的名称；
ops，该 clock 相关的操作函数集，具体参考下面的描述；
parent_names，该 clock 所有的 parent clock 的名称。这是一个字符串数组，保存了所有可能的 parent；
num_parents，parent 的个数；
flags，一些 framework 级别的 flags，后面会详细说明。

```
1: struct clk_ops {
2:        int             (*prepare)(struct clk_hw *hw);
3:        void            (*unprepare)(struct clk_hw *hw);
4:        int             (*is_prepared)(struct clk_hw *hw);
5:        void            (*unprepare_unused)(struct clk_hw *hw);
6:        int             (*enable)(struct clk_hw *hw);
7:        void            (*disable)(struct clk_hw *hw);
8:        int             (*is_enabled)(struct clk_hw *hw);
9:        void            (*disable_unused)(struct clk_hw *hw);
10:       unsigned long   (*recalc_rate)(struct clk_hw *hw,
11:                                      unsigned long parent_rate);
12:       long            (*round_rate)(struct clk_hw *hw, unsigned long,
13:                                      unsigned long *);
14:       int             (*set_parent)(struct clk_hw *hw, u8 index);
15:       u8              (*get_parent)(struct clk_hw *hw);
16:       int             (*set_rate)(struct clk_hw *hw, unsigned long,
17:                                   unsigned long);
18:       void            (*init)(struct clk_hw *hw);
19: };
```

这是 clock 的操作函数集， 很多和“Linux common clock framework(1)_概述”中的 clock framework 通用 API 一致（通用 API 会直接调用相应的操作函数）：
is_prepared，判断 clock 是否已经 prepared。可以不提供，clock framework core 会维护一个 prepare 的计数（该计数在 clk_prepare 调用时加一，在 clk_unprepare 时减一），并依据该计数判断是否 prepared；
unprepare_unused，自动 unprepare unused clocks；
is_enabled，和 is_prepared 类似；
disable_unused，自动 disable unused clocks；
注 2：clock framework core 提供一个 clk_disable_unused 接口，在系统初始化的 late_call 中调用，用于关闭 unused clocks，这个接口会调用相应 clock 的.unprepare_unused 和.disable_unused 函数。
recalc_rate，以 parent clock rate 为参数，从新计算并返回 clock rate；
注 3：细心的读者可能会发现，该结构没有提供 get_rate 函数，因为会有一个 rate 变量缓存，另外可以使用 recalc_rate。
round_rate，该接口有点特别，在返回 rounded rate 的同时，会通过一个指针，返回 round 后 parent 的 rate。这和 CLK_SET_RATE_PARENT flag 有关，后面会详细解释；
init，clock 的初始化接口，会在 clock 被 register 到内核时调用。

```
1: /*
2:  * flags used across common struct clk.  these flags should only affect the
3:  * top-level framework.  custom flags for dealing with hardware specifics
4:  * belong in struct clk_foo
5:  */
6: #define CLK_SET_RATE_GATE       BIT(0) /* must be gated across rate change */
7: #define CLK_SET_PARENT_GATE     BIT(1) /* must be gated across re-parent */
8: #define CLK_SET_RATE_PARENT     BIT(2) /* propagate rate change up one level */
9: #define CLK_IGNORE_UNUSED       BIT(3) /* do not gate even if unused */
```

```
#define CLK_IS_ROOT            BIT(4) /* root clk, has no parent */
#define CLK_IS_BASIC           BIT(5) /* Basic clk, can't do a to_clk_foo() */
#define CLK_GET_RATE_NOCACHE   BIT(6) /* do not use the cached clk rate */
```

上面是 framework 级别的 flags，可以使用或的关系，指定多个 flags，解释如下：
CLK_SET_RATE_GATE，表示在改变该 clock 的 rate 时，必须 gated（关闭）；
CLK_SET_PARENT_GATE，表示在改变该 clock 的 parent 时，必须 gated（关闭）；
CLK_SET_RATE_PARENT，表示改变该 clock 的 rate 时，要将该改变传递到上层 parent（下面再详细说明）；
CLK_IGNORE_UNUSED，忽略 disable unused 的调用；
CLK_IS_ROOT，该 clock 为 root clock，没有 parent；
CLK_IS_BASIC，不再使用了；
CLK_GET_RATE_NOCACHE，get rate 时，不要从缓存中拿，而是从新计算。

注 4：round_rate 和 CLK_SET_RATE_PARENT
当 clock consumer 调用 clk_round_rate 获取一个近似的 rate 时，如果该 clock 没有提供.round_rate 函数，有两种方法：
1）在没有设置 CLK_SET_RATE_PARENT 标志时，直接返回该 clock 的 cache rate
2）如果设置了 CLK_SET_RATE_PARENT 标志，则会询问 parent，即调用 clk_round_rate 获取 parent clock 能提供的、最接近该 rate 的值。这是什么意思呢？也就是说，如果 parent clock 可以得到一个近似的 rate 值，那么通过改变 parent clock，就能得到所需的 clock。
在后续的 clk_set_rate 接口中，会再次使用该 flag，如果置位，则会在设置 rate 时，传递到 parent clock，因此 parent clock 的 rate 可能会重设。
讲的很拗口，我觉得我也没说清楚，那么最好的方案就是：在写 clock driver 时，最好不用这个 flag，简单的就是最好的（前提是能满足需求）。

3.2 clock tree 建立相关的 API

3.2.1 clk_register

系统中，每一个 clock 都有一个 struct clk_hw 变量描述，clock provider 需要使用 register 相关的接口，将这些 clock 注册到 kernel，clock framework 的核心代码会把它们转换为 struct clk 变量，并以 tree 的形式组织起来。这些接口的原型如下：

```
/**
 * clk_register - allocate a new clock, register it and return an opaque cookie
 * @dev: device that is registering this clock
 * @hw: link to hardware-specific clock data
 *
 * clk_register is the primary interface for populating the clock tree with new
 * clock nodes.  It returns a pointer to the newly allocated struct clk which
 * cannot be dereferenced by driver code but may be used in conjuction with the
 * rest of the clock API.  In the event of an error clk_register will return an
 * error code; drivers must test for an error code after calling clk_register.
 */
struct clk *clk_register(struct device *dev, struct clk_hw *hw);
struct clk *devm_clk_register(struct device *dev, struct clk_hw *hw);
void clk_unregister(struct clk *clk);
void devm_clk_unregister(struct device *dev, struct clk *clk);
```

这些 API 比较简单（复杂的是怎么填充 struct clk_hw 变量），register 接口接受一个填充好的 struct clk_hw 指针，将它转换为 sruct clk 结构，并根据 parent 的名字，添加到 clock tree 中。
不过，clock framework 所做的远比这周到，它基于 clk_register，又封装了其它接口，使 clock provider 在注册 clock 时，连 struct clk_hw 都不需要关心，而是直接使用类似人类语言的方式，下面继续。

3.2.2 clock 分类及 register

根据 clock 的特点，clock framework 将 clock 分为 fixed rate、gate、devider、mux、fixed factor、composite 六类，每一类 clock 都有相似的功能、相似的控制方式，因而可以使用相同的逻辑 s，统一处理，这充分体现了面向对象的思想。

1）fixed rate clock

这一类 clock 具有固定的频率，不能开关、不能调整频率、不能选择 parent、不需要提供任何的 clk_ops 回调函数，是最简单的一类 clock。

可以直接通过 DTS 配置的方式支持，clock framework core 能直接从 DTS 中解出 clock 信息，并自动注册到 kernel，不需要任何 driver 支持。

clock framework 使用 struct clk_fixed_rate 结构抽象这一类 clock，另外提供了一个接口，可以直接注册 fixed rate clock，如下：

```
1: /**
2:  * struct clk_fixed_rate - fixed-rate clock
3:  * @hw:        handle between common and hardware-specific interfaces
4:  * @fixed_rate: constant frequency of clock
5:  */
6: struct clk_fixed_rate {
7:        struct         clk_hw hw;
8:        unsigned long   fixed_rate;
9:        u8            flags;
10: };
12: extern const struct clk_ops clk_fixed_rate_ops;
13: struct clk *clk_register_fixed_rate(struct device *dev, const char *name,
14:               const char *parent_name, unsigned long flags,
15:               unsigned long fixed_rate);
```

clock provider 一般不需要直接使用 struct clk_fixed_rate 结构，因为 clk_register_fixed_rate 接口是非常方便的；

clk_register_fixed_rate 接口以 clock name、parent name、fixed_rate 为参数，创建一个具有固定频率的 clock，该 clock 的 clk_ops 也是 clock framework 提供的，不需要 provider 关心；

如果使用 DTS 的话，clk_register_fixed_rate 都不需要，直接在 DTS 中配置即可，后面会说明。

2）gate clock

这一类 clock 只可开关（会提供.enable/.disable 回调），可使用下面接口注册：

```
1: struct clk *clk_register_gate(struct device *dev, const char *name,
2:               const char *parent_name, unsigned long flags,
3:               void __iomem *reg, u8 bit_idx,
4:               u8 clk_gate_flags, spinlock_t *lock);
```

需要提供的参数包括：

name，clock 的名称；

parent_name，parent clock 的名称，没有的话可留空；

flags，可参考 3.1 中的说明；

reg，控制该 clock 开关的寄存器地址（虚拟地址）；

bit_idx，控制 clock 开关的 bit 位（是 1 开，还是 0 开，可通过下面 gate 特有的 flag 指定）；

clk_gate_flags，gate clock 特有的 flag，当前只有一种：CLK_GATE_SET_TO_DISABLE，clock 开关控制的方式，如果置位，表示写 1 关闭 clock，反之亦然；

lock，如果 clock 开关时需要互斥，可提供一个 spinlock。

3）divider clock

这一类 clock 可以设置分频值（因而会提供.recalc_rate/.set_rate/.round_rate 回调），可通过下面两个接口注册：

```
1: struct clk *clk_register_divider(struct device *dev, const char *name,
2:               const char *parent_name, unsigned long flags,
3:               void __iomem *reg, u8 shift, u8 width,
4:               u8 clk_divider_flags, spinlock_t *lock);
```

该接口用于注册分频比规则的 clock：

reg，控制 clock 分频比的寄存器；

shift，控制分频比的 bit 在寄存器中的偏移；
width，控制分频比的 bit 位数，默认情况下，实际的 divider 值是寄存器值加 1。如果有其它例外，可使用下面的的 flag 指示；
clk_divider_flags，divider clock 特有的 flag，包括：
CLK_DIVIDER_ONE_BASED，实际的 divider 值就是寄存器值（0 是无效的，除非设置 CLK_DIVIDER_ALLOW_ZERO flag）；
CLK_DIVIDER_POWER_OF_TWO，实际的 divider 值是寄存器值得 2 次方；
CLK_DIVIDER_ALLOW_ZERO，divider 值可以为 0（不改变，视硬件支持而定）。
如有需要其他分频方式，就需要使用另外一个接口，如下：

```
1: struct clk *clk_register_divider_table(struct device *dev, const char *name,
2:              const char *parent_name, unsigned long flags,
3:              void __iomem *reg, u8 shift, u8 width,
4:              u8 clk_divider_flags, const struct clk_div_table *table,
5:              spinlock_t *lock);
```

该接口用于注册分频比不规则的 clock，和上面接口比较，差别在于 divider 值和寄存器值得对应关系由一个 table 决定，该 table 的原型为：

```
struct clk_div_table {
    unsigned int    val;
    unsigned int    div;
};
```

其中 val 表示寄存器值，div 表示分频值，它们的关系也可以通过 clk_divider_flags 改变。
4）mux clock
这一类 clock 可以选择多个 parent，因为会实现.get_parent/.set_parent/.recalc_rate 回调，可通过下面两个接口注册：

```
1: struct clk *clk_register_mux(struct device *dev, const char *name,
2:              const char **parent_names, u8 num_parents, unsigned long flags,
3:              void __iomem *reg, u8 shift, u8 width,
4:              u8 clk_mux_flags, spinlock_t *lock);
```

该接口可注册 mux 控制比较规则的 clock（类似 divider clock）：
parent_names，一个字符串数组，用于描述所有可能的 parent clock；
num_parents，parent clock 的个数；
reg、shift、width，选择 parent 的寄存器、偏移、宽度，默认情况下，寄存器值为 0 时，对应第一个 parent，依此类推。
如有例外，可通过下面的 flags，以及另外一个接口实现；
clk_mux_flags，mux clock 特有的 flag：
CLK_MUX_INDEX_ONE，寄存器值不是从 0 开始，而是从 1 开始；
CLK_MUX_INDEX_BIT，寄存器值为 2 的幂。

```
1: struct clk *clk_register_mux_table(struct device *dev, const char *name,
2:              const char **parent_names, u8 num_parents, unsigned long flags,
3:              void __iomem *reg, u8 shift, u32 mask,
4:              u8 clk_mux_flags, u32 *table, spinlock_t *lock);
```

该接口通过一个 table，注册 mux 控制不规则的 clock，原理和 divider clock 类似，不再详细介绍。
5）fixed factor clock
这一类 clock 具有固定的 factor（即 multiplier 和 divider），clock 的频率是由 parent clock 的频率，乘以 mul，除以 div，多用于一些具有固定分频系数的 clock。由于 parent clock 的频率可以改变，因而 fix factor clock 也可该改变频率，因此也会提供.recalc_rate/.set_rate/.round_rate 等回调。
可通过下面接口注册：

```
1: struct clk *clk_register_fixed_factor(struct device *dev, const char *name,
2:              const char *parent_name, unsigned long flags,
3:              unsigned int mult, unsigned int div);
```

另外，这一类接口和 fixed rate clock 类似，不需要提供 driver，只需要配置 dts 即可。
6）composite clock

顾名思义，就是 mux、divider、gate 等 clock 的组合，可通过下面接口注册：

```
1: struct clk *clk_register_composite(struct device *dev, const char *name,
2:              const char **parent_names, int num_parents,
3:              struct clk_hw *mux_hw, const struct clk_ops *mux_ops,
4:              struct clk_hw *rate_hw, const struct clk_ops *rate_ops,
5:              struct clk_hw *gate_hw, const struct clk_ops *gate_ops,
6:              unsigned long flags);
```

看着有点复杂，但理解了上面 1~5 类 clock，这里就只剩下苦力了，耐心一点，就可以了。

3.2.3 DTS 相关的 API

再回到第 2 章 DTS 相关的介绍，clock driver 使用一个 DTS node 描述一个 clock provider，而 clock consumer 则会使用类似“clocks = <&clock 32>, <&clock 45>;”的形式引用，clock framework 会自行把这些抽象的数字转换成实际的 struct clk 结构，怎么做的呢？肯定离不开 clock provider 的帮助。

3.2.1 和 3.2.2 小节所描述的 regitser 接口，负责把 clocks 抽象为一个一个的 struct clock，与此同时，clock provider 需要把这些 struct clk 结构保存起来，并调用 clock framework 的接口，将这些对应信息告知 framework 的 OF 模块，这样才可以帮助将 clock consumer 的 DTS 描述转换为 struct clk 结构。该接口如下：

```
1: int of_clk_add_provider(struct device_node *np,
2:                  struct clk *(*clk_src_get)(struct of_phandle_args *args,
3:                                  void *data),
4:                  void *data);
```

np，device_node 指针，clock provider 在和自己的 DTS 匹配时获得；

clk_src_get，获取 struct clk 指针的回调函数，由 clock provider 根据实际的逻辑实现，参数说明如下：

args，struct of_phandle_args 类型的指针，由 DTS 在解析参数时传递。例如上面的“clocks = <&clock 32>, <&clock 45>;”，32、45 就是通过这个指针传进来的；

data，保存 struct clk 结构的指针，通常是一个数组，具体由 provider 决定。

data，和回调函数中的 data 意义相同，只是这里由 provider 提供，get 时由 clock framework core 传递给回调函数。

对于常用的 one cell clock provider（第 2 章的例子），clock framework core 提供一个默认的会调用函数，如下：

```
1: struct clk_onecell_data {
2:       struct clk **clks;
3:       unsigned int clk_num;
4: };
5: struct clk *of_clk_src_onecell_get(struct of_phandle_args *clkspec, void *data);
```

其中 data 指针为 struct clk_onecell_data 结构，该结构提供了 clk 指针和 clk_num 的对应，clock provider 在 regitser clocks 时，同时维护一个 clk 和 num 对应的数组，并调用 of_clk_add_provider 接口告知 clock framework core 即可。

4. 使用 clock framework 编写 clock 驱动的步骤

编写 clock driver 的步骤大概如下：

1）分析硬件的 clock tree，按照上面所描述的分类，讲这些 clock 分类。

2）将 clock tree 在 DTS 中描述出来，需要注意以下几 2 点：

a）对于 fixed rate clocks，.compatible 固定填充"fixed-clock"，并提供"clock-frequency"和"clock-output-names"关键字。之后不需要再 driver 中做任何处理，clock framework core 会帮我们搞定一切。

b）同样，对于 fixed factor clock，.compatible 为"fixed-factor-clock"，并提供"clock-div"、"clock-mult"和"clock-output-names"关键字。clock framework core 会帮我们搞定一切。

切记，尽量利用 kernel 已有资源，不要多写一行代码，简洁的就是美的！

3）对于不能由 clock framework core 处理的 clock，需要在 driver 中使用 struct of_device_id 进行匹配，并在初始化时，调用 OF 模块，查找所有的 DTS 匹配项，并执行合适的 regitser 接口，注册 clock。

4）注册 clock 的同时，将返回的 struct clk 指针，保存在一个数组中，并调用 of_clk_add_provider 接口，告知 clock framework core。

5）最后，也是最重要的一点，多看 kernel 源代码，多模仿，多抄几遍，什么都熟悉了！