Linux common clock framework(3)_实现逻辑分析

作者：wowo 发布于：2014-11-24 22:31 分类：电源管理子系统

1. 前言

前面两篇 clock framework 的分析文章，分别从 clock consumer 和 clock provider 的角度，介绍了 Linux kernel 怎么管理系统的 clock 资源，以及 device driver 怎么使用 clock 资源。本文将深入到 clock framework 的内部，分析相关的实现逻辑。
注：本文使用的 kernel 版本为 linux-3.10.29。虽然最新版本的 kernel 增加了一些内容，但主要逻辑没有改变，就不紧跟 kernel 的步伐了。

2. struct clk 结构

到目前为止，我们还没有仔细介绍过 struct clk 这个代表了一个 clock 的数据结构呢。对 consumer 和 provider 来说，可以不关心，但对内部实现逻辑来说，就不得不提了：

```
1: /* include/linux/clk-private.h */
2: struct clk {
3:         const char            *name;
4:         const struct clk_ops    *ops;
5:         struct clk_hw          *hw;
6:         struct clk            *parent;
7:         const char            **parent_names;
8:         struct clk            **parents;
9:         u8                    num_parents;
10:        unsigned long          rate;
11:        unsigned long          new_rate;
12:        unsigned long          flags;
13:        unsigned int           enable_count;
14:        unsigned int           prepare_count;
15:        struct hlist_head      children;
16:        struct hlist_node      child_node;
17:        unsigned int           notifier_count;
18: #ifdef CONFIG_COMMON_CLK_DEBUG
19:        struct dentry          *dentry;
20: #endif
21: };
```

name, ops, hw, parents_name, num_parents, flags, 可参考“Linux common clock framework(2)_clock provider”中的相关描述；
parent，保存了该 clock 当前的 parent clock 的 struct clk 指针；
parents，一个指针数组，保存了所有可能的 parent clock 的 struct clk 指针；
rate，当前的 clock rate；
new_rate,新设置的 clock rate，之所要保存在这里，是因为 set rate 过程中有一些中间计算，后面再详解；
enable_count, prepare_count，该 clock 被 enable 和 prepare 的次数，用于确保 enable/disable 以及 prepare/unprepare 的成对调用；
children，该 clock 的 children clocks（孩儿们），以链表的形式组织；
child_node，一个 list node，自己作为 child 时，挂到 parent 的 children list 时使用；
notifier_count，记录注册到 notifier 的个数。

3. clock regitser/unregister

在“Linux common clock framework(2)_clock provider”中已经讲过，clock provider 需要将系统的 clock 以 tree 的形式组织起来，分门别类，并在系统初始化时，通过 provider 的初始化接口，或者 clock framework core 的 DTS 接口，将所有的 clock 注册到 kernel。
clock 的注册，统一由 clk_regitser 接口实现，但基于该接口，kernel 也提供了其它更为便利注册接口，下面将会一一描述。

3.1 clk_regitser

clk_register 是所有 register 接口的共同实现，负责将 clock 注册到 kernel，并返回代表该 clock 的 struct clk 指针。分析该接口之前，我们先看一下下面的内容：

```
1: 1 F   f    clk_register     .\arch\arm\mach-at91\clock.c
2:             int __init clk_register(struct clk *clk)
3: 2 F   v    clk_register     .\arch\arm\mach-davinci\clock.c
4:             EXPORT_SYMBOL(clk_register);
5: 3 F   f    clk_register     .\arch\arm\mach-davinci\clock.c
6:             int clk_register(struct clk *clk)
7: 4 F   v    clk_register     .\arch\arm\mach-omap1\clock.c
8:             EXPORT_SYMBOL(clk_register);
9: 5 F   f    clk_register     .\arch\arm\mach-omap1\clock.c
10:            int clk_register(struct clk *clk)
11: 6 F   v    clk_register     .\arch\c6x\platforms\pll.c
12:            EXPORT_SYMBOL(clk_register);
13: 7 F   f    clk_register     .\arch\c6x\platforms\pll.c
14:            int clk_register(struct clk *clk)
15: 8 F   v    clk_register     .\arch\unicore32\kernel\clock.c
16:            EXPORT_SYMBOL(clk_register);
17: 9 F   f    clk_register     .\arch\unicore32\kernel\clock.c
18:            int clk_register(struct clk *clk)
19: 0 F   v    clk_register     .\drivers\clk\clk.c
20:            EXPORT_SYMBOL_GPL(clk_register);
21: 1 F   f    clk_register     .\drivers\clk\clk.c
22:            struct clk *clk_register(struct device *dev, struct clk_hw *hw)
23: 2 F   v    clk_register     .\drivers\sh\clk\core.c
24:            EXPORT_SYMBOL_GPL(clk_register);
25: 3 F   f    clk_register     .\drivers\sh\clk\core.c
26:            int clk_register(struct clk *clk)
```

上面是 kernel 中 clk_register 接口可能的实现位置，由此可以看出，clk_register 在“include/linux/clk-provider.h”中声明，却可能在不同的 C 文件中实现。其它 clock API 也类似。这说明了什么？
这恰恰呼应了“Linux common clock framework”中“common”一词。
在旧的 kernel 中，clock framework 只是规定了一系列的 API 声明，具体 API 的实现，由各个 machine 代码完成。这就导致每个 machine 目录下，都有一个类似 clock.c 的文件，以比较相似的逻辑，实现 clock provider 的功能。显然，这里面有很多冗余代码。
后来，kernel 将这些公共代码，以 clock provider 的形式（上面 drivers/clk/clk.c 文件）抽象出来，就成了我们所说的 common clock framework。
后面所有的描述，都会以 common clock framework 的核心代码为基础，其它的，就不再涉及了。
下面是 clk_register 的实现：

```
1: /**
```

```
 2: * clk_register - allocate a new clock, register it and return an opaque cookie
 3: * @dev: device that is registering this clock
 4: * @hw: link to hardware-specific clock data
 5: *
 6: * clk_register is the primary interface for populating the clock tree with new
 7: * clock nodes.  It returns a pointer to the newly allocated struct clk which
 8: * cannot be dereferenced by driver code but may be used in conjuction with the
 9: * rest of the clock API.  In the event of an error clk_register will return an
10: * error code; drivers must test for an error code after calling clk_register.
11: */
12: struct clk *clk_register(struct device *dev, struct clk_hw *hw)
13: {
14:     int i, ret;
15:     struct clk *clk;
16:
17:     clk = kzalloc(sizeof(*clk), GFP_KERNEL);
18:     if (!clk) {
19:         pr_err("%s: could not allocate clk\n", __func__);
20:         ret = -ENOMEM;
21:         goto fail_out;
22:     }
23:
24:     clk->name = kstrdup(hw->init->name, GFP_KERNEL);
25:     if (!clk->name) {
26:         pr_err("%s: could not allocate clk->name\n", __func__);
27:         ret = -ENOMEM;
28:         goto fail_name;
29:     }
30:     clk->ops = hw->init->ops;
31:     if (dev && dev->driver)
32:         clk->owner = dev->driver->owner;
33:     clk->hw = hw;
34:     clk->flags = hw->init->flags;
35:     clk->num_parents = hw->init->num_parents;
36:     hw->clk = clk;
37:
38:     /* allocate local copy in case parent_names is __initdata */
39:     clk->parent_names = kcalloc(clk->num_parents, sizeof(char *),
40:                     GFP_KERNEL);
41:
42:     if (!clk->parent_names) {
43:         pr_err("%s: could not allocate clk->parent_names\n", __func__);
44:         ret = -ENOMEM;
45:         goto fail_parent_names;
46:     }
47:
48:
49:     /* copy each string name in case parent_names is __initdata */
50:     for (i = 0; i < clk->num_parents; i++) {
51:         clk->parent_names[i] = kstrdup(hw->init->parent_names[i],
```

```
52:                     GFP_KERNEL);
53:         if (!clk->parent_names[i]) {
54:             pr_err("%s: could not copy parent_names\n", __func__);
55:             ret = -ENOMEM;
56:             goto fail_parent_names_copy;
57:         }
58:     }
59:
60:     ret = __clk_init(dev, clk);
61:     if (!ret)
62:         return clk;
63:
64: fail_parent_names_copy:
65:     while (--i >= 0)
66:         kfree(clk->parent_names[i]);
67:     kfree(clk->parent_names);
68: fail_parent_names:
69:     kfree(clk->name);
70: fail_name:
71:     kfree(clk);
72: fail_out:
73:     return ERR_PTR(ret);
74: }
75: EXPORT_SYMBOL_GPL(clk_register);
```

该接口接受一个 struct clk_hw 指针，该指针包含了将要注册的 clock 的信息（具体可参考“Linux common clock framework(2)_clock provider”），在内部分配一个 struct clk 变量后，将 clock 信息保存在变量中，并返回给调用者。实现逻辑如下：
分配 struct clk 空间；
根据 struct clk_hw 指针提供的信息，初始化 clk 的 name、ops、hw、flags、num_parents、parents_names 等变量；
调用内部接口__clk_init，执行后续的初始化操作。这个接口包含了 clk_regitser 的主要逻辑，具体如下。

```
1: /**
2:  * __clk_init - initialize the data structures in a struct clk
3:  * @dev:    device initializing this clk, placeholder for now
4:  * @clk:    clk being initialized
5:  *
6:  * Initializes the lists in struct clk, queries the hardware for the
7:  * parent and rate and sets them both.
8:  */
9: int __clk_init(struct device *dev, struct clk *clk)
10: {
11:     int i, ret = 0;
12:     struct clk *orphan;
13:     struct hlist_node *tmp2;
14:
15:     if (!clk)
16:         return -EINVAL;
17:
18:     clk_prepare_lock();
19:
20:     /* check to see if a clock with this name is already registered */
```

```
21:     if (__clk_lookup(clk->name)) {
22:         pr_debug("%s: clk %s already initialized\n",
23:                 __func__, clk->name);
24:         ret = -EEXIST;
25:         goto out;
26:     }
27:
28:     /* check that clk_ops are sane.  See Documentation/clk.txt */
29:     if (clk->ops->set_rate &&
30:             !(clk->ops->round_rate && clk->ops->recalc_rate)) {
31:         pr_warning("%s: %s must implement .round_rate & .recalc_rate\n",
32:                 __func__, clk->name);
33:         ret = -EINVAL;
34:         goto out;
35:     }
36:
37:     if (clk->ops->set_parent && !clk->ops->get_parent) {
38:         pr_warning("%s: %s must implement .get_parent & .set_parent\n",
39:                 __func__, clk->name);
40:         ret = -EINVAL;
41:         goto out;
42:     }
43:
44:     /* throw a WARN if any entries in parent_names are NULL */
45:     for (i = 0; i < clk->num_parents; i++)
46:         WARN(!clk->parent_names[i],
47:                 "%s: invalid NULL in %s's .parent_names\n",
48:                 __func__, clk->name);
49:
50:     /*
51:      * Allocate an array of struct clk *'s to avoid unnecessary string
52:      * look-ups of clk's possible parents.  This can fail for clocks passed
53:      * in to clk_init during early boot; thus any access to clk->parents[]
54:      * must always check for a NULL pointer and try to populate it if
55:      * necessary.
56:      *
57:      * If clk->parents is not NULL we skip this entire block.  This allows
58:      * for clock drivers to statically initialize clk->parents.
59:      */
60:     if (clk->num_parents > 1 && !clk->parents) {
61:         clk->parents = kzalloc((sizeof(struct clk*) * clk->num_parents),
62:                 GFP_KERNEL);
63:         /*
64:          * __clk_lookup returns NULL for parents that have not been
65:          * clk_init'd; thus any access to clk->parents[] must check
66:          * for a NULL pointer.  We can always perform lazy lookups for
67:          * missing parents later on.
68:          */
69:         if (clk->parents)
70:             for (i = 0; i < clk->num_parents; i++)
```

```
71:             clk->parents[i] =
72:                 __clk_lookup(clk->parent_names[i]);
73:     }
74:
75:     clk->parent = __clk_init_parent(clk);
76:
77:     /*
78:      * Populate clk->parent if parent has already been __clk_init'd.  If
79:      * parent has not yet been __clk_init'd then place clk in the orphan
80:      * list.  If clk has set the CLK_IS_ROOT flag then place it in the root
81:      * clk list.
82:      *
83:      * Every time a new clk is clk_init'd then we walk the list of orphan
84:      * clocks and re-parent any that are children of the clock currently
85:      * being clk_init'd.
86:      */
87:     if (clk->parent)
88:         hlist_add_head(&clk->child_node,
89:                 &clk->parent->children);
90:     else if (clk->flags & CLK_IS_ROOT)
91:         hlist_add_head(&clk->child_node, &clk_root_list);
92:     else
93:         hlist_add_head(&clk->child_node, &clk_orphan_list);
94:
95:     /*
96:      * Set clk's rate.  The preferred method is to use .recalc_rate.  For
97:      * simple clocks and lazy developers the default fallback is to use the
98:      * parent's rate.  If a clock doesn't have a parent (or is orphaned)
99:      * then rate is set to zero.
100:     */
101:    if (clk->ops->recalc_rate)
102:        clk->rate = clk->ops->recalc_rate(clk->hw,
103:                __clk_get_rate(clk->parent));
104:    else if (clk->parent)
105:        clk->rate = clk->parent->rate;
106:    else
107:        clk->rate = 0;
108:
109:    /*
110:     * walk the list of orphan clocks and reparent any that are children of
111:     * this clock
112:     */
113:    hlist_for_each_entry_safe(orphan, tmp2, &clk_orphan_list, child_node) {
114:        if (orphan->ops->get_parent) {
115:            i = orphan->ops->get_parent(orphan->hw);
116:            if (!strcmp(clk->name, orphan->parent_names[i]))
117:                __clk_reparent(orphan, clk);
118:            continue;
119:        }
120:
```

```
121:         for (i = 0; i < orphan->num_parents; i++)
122:             if (!strcmp(clk->name, orphan->parent_names[i])) {
123:                 __clk_reparent(orphan, clk);
124:                 break;
125:             }
126:      }
127:
128:     /*
129:      * optional platform-specific magic
130:      *
131:      * The .init callback is not used by any of the basic clock types, but
132:      * exists for weird hardware that must perform initialization magic.
133:      * Please consider other ways of solving initialization problems before
134:      * using this callback, as it's use is discouraged.
135:      */
136:     if (clk->ops->init)
137:         clk->ops->init(clk->hw);
138:
139:     clk_debug_register(clk);
140:
141: out:
142:     clk_prepare_unlock();
143:
144:     return ret;
145: }
```

__clk_init 接口的实现相当繁琐，做的事情包括：

20~26 行，以 clock name 为参数，调用__clk_lookup 接口，查找是否已有相同 name 的 clock 注册，如果有，则返回错误。由此可以看出，clock framework 以 name 唯一识别一个 clock，因此不能有同名的 clock 存在；

28~42 行，检查 clk ops 的完整性，例如：如果提供了 set_rate 接口，就必须提供 round_rate 和 recalc_rate 接口；如果提供了 set_parent，就必须提供 get_parent。这些逻辑背后的含义，会在后面相应的地方详细描述；

50~73 行，分配一个 struct clk *类型的数组，缓存该 clock 的 parents clock。具体方法是根据 parents_name，查找相应的 struct clk 指针；

75 行，获取当前的 parent clock，并将其保存在 parent 指针中。具体可参考下面“说明 2”；

77~93 行，根据该 clock 的特性，将它添加到 clk_root_list、clk_orphan_list 或者 parent->children 三个链表中的一个，具体请参考下面“说明 1”；

95~107 行，计算 clock 的初始 rate，具体请参考下面“说明 3”；

109~126 行，尝试 reparent 当前所有的孤儿（orphan）clock，具体请参考下面“说明 4”；

128~137 行，如果 clock ops 提供了 init 接口，执行之（由注释可知，kernel 不建议提供 init 接口）。

上面的 clock init 流程，有下面 4 点补充说明：

说明 1：clock 的管理和查询

clock framework 有 2 条全局的链表：clk_root_list 和 clk_orphan_list。所有设置了 CLK_IS_ROOT 属性的 clock 都会挂在 clk_root_list 中。其它 clock，如果有 valid 的 parent ，则会挂到 parent 的“children”链表中，如果没有 valid 的 parent，则会挂到 clk_orphan_list 中。

查询时（__clk_lookup 接口做的事情），依次搜索：clk_root_list-->root_clk-->children-->child's children，clk_orphan_list-->orphan_clk-->children-->child's children，即可。

说明 2：当前 parent clock 的选择（__clk_init_parent）

对于没有 parent，或者只有 1 个 parent 的 clock 来说，比较简单，设置为 NULL，或者根据 parent name 获得 parent 的 struct clk 指针接。

对于有多个 parent 的 clock，就必须提供.get_parent ops，该 ops 要根据当前硬件的配置情况，例如寄存器值，返回当前所有使用的 parent 的 index（即第几个 parent）。然后根据 index，取出对应 parent clock 的 struct clk 指针，作为当前的 parent。

说明 3：clock 的初始 rate 计算

对于提供.recalc_rate ops 的 clock 来说，优先使用该 ops 获取初始的 rate。如果没有提供，退而求其次，直接使用 parent clock 的 rate。最后，如果该 clock 没有 parent，则初始的 rate 只能选择为 0。

.recalc_rate ops 的功能，是以 parent clock 的 rate 为输入参数，根据当前硬件的配置情况，如寄存器值，计算获得自身的 rate 值。

说明 4：orphan clocks 的 reparent

有些情况下，child clock 会先于 parent clock 注册，此时该 child 就会成为 orphan clock，被收养在 clk_orphan_list 中。而每当新的 clock 注册时，kernel 都会检查这个 clock 是否是某个 orphan 的 parent，如果是，就把这个 orphan 从 clk_orphan_list 中移除，放到新注册的 clock 的怀抱。这就是 reparent 的功能，它的处理逻辑是：

a） 遍历 orphan list，如果 orphan 提供了.get_parent ops，则通过该 ops 得到当前 parent 的 index，并从 parent_names 中取出该 parent 的 name，然后和新注册的 clock name 比较，如果相同，呵呵，找到 parent 了，执行__clk_reparent，进行后续的操作。

b） 如果没有提供.get_parent ops，只能遍历自己的 parent_names，检查是否有和新注册 clock 匹配的，如果有，执行__clk_reparent，进行后续的操作。

c） __clk_reparent 会把这个 orphan 从 clk_orphan_list 中移除，并挂到新注册的 clock 上。然后调用__clk_recalc_rates，重新计算自己以及自己所有 children 的 rate。计算过程和上面的 clock rate 设置类似。

3.2 clk_unregister/devm_clk_register/devm_clk_unregister

clock 的 regitser 和 init，几乎占了 clock framework 大部分的实现逻辑。clk_unregister 是 regitser 接口的反操作，不过当前没有实现（不需要）。而 devm_clk_register/devm_clk_unregister 则是 clk_register/clk_unregister 的 device resource manager 版本。

3.3 fixed rate clock 的注册

“Linux common clock framework(2)_clock provider”中已经对 fixed rate clock 有过详细的介绍，这种类型的 clock 有两种注册方式，通过 API 注册和通过 DTS 注册，具体的实现位于“drivers/clk/clk-fixed-rate.c”中，介绍如下。

1）通过 API 注册

```
1: struct clk *clk_register_fixed_rate(struct device *dev, const char *name,
2:                 const char *parent_name, unsigned long flags,
3:                 unsigned long fixed_rate)
4: {
5:         struct clk_fixed_rate *fixed;
6:         struct clk *clk;
7:         struct clk_init_data init;
8:
9:         /* allocate fixed-rate clock */
10:        fixed = kzalloc(sizeof(struct clk_fixed_rate), GFP_KERNEL);
11:        if (!fixed) {
12:                pr_err("%s: could not allocate fixed clk\n", __func__);
13:                return ERR_PTR(-ENOMEM);
14:        }
15:
16:        init.name = name;
17:        init.ops = &clk_fixed_rate_ops;
```

```
18:         init.flags = flags | CLK_IS_BASIC;
19:         init.parent_names = (parent_name ? &parent_name: NULL);
20:         init.num_parents = (parent_name ? 1 : 0);
21:
22:         /* struct clk_fixed_rate assignments */
23:         fixed->fixed_rate = fixed_rate;
24:         fixed->hw.init = &init;
25:
26:         /* register the clock */
27:         clk = clk_register(dev, &fixed->hw);
28:
29:         if (IS_ERR(clk))
30:                 kfree(fixed);
31:
32:         return clk;
33: }
```

clk_register_fixed_rate API 用于注册 fixed rate clock，它接收传入的 name、parent_name、flags、fixed_rate 等参数，并转换为 struct clk_hw 结构，最终调用 clk_register 接口，注册 clock。大致的逻辑如下：

16~20 行，定义一个 struct clk_init_data 类型的变量（init），并根据传入的参数以及 fixed rate clock 的特性，初始化该变量；

22~30 行，分配一个私有的数据结构（struct clk_fixed_rate），并将 init 的指针保存在其中，最后调用 clk_regitser 注册该 clock。

说明 1：struct clk_init_data 类型的变量

struct clk_init_data 类型的变量（init），是一个局部变量，传递给 clk_regitser 使用时，用的是它的指针，说明了什么？说明该变量不会再后面使用了。再回忆一下 clk_regitser 的实现逻辑，会把所有的信息 copy 一遍，这里就好理解了。后面其它类型的 clock 注册时，道理相同。

说明 2：fixed rate clock 的实现思路

私有数据结构的定义如下：

```
1: struct clk_fixed_rate {
2:         struct          clk_hw hw;
3:         unsigned long   fixed_rate;
4:         u8              flags;
5: };
```

包含一个 struct clk_hw 变量，用于 clk_regitser。另外两个变量，则为该类型 clock 特有的属性。私有数据结构变量（fixed）是通过 kzalloc 分配的，说明后续还需要使用。那怎么用呢？

由 clk_regitser 的实现可知，fixed rate clock 注册时 hw);>，把 fixed 指针中 hw 变量的地址保存在了 struct clk 指针中了。因此，在任何时候，通过 struct clk 指针（clock 的代表），就可以找到所对应 clock 的 struct clk_hw 指针，从而可以找到相应的私有变量（fixed）的指针以及其中的私有数据。

基于此，fixed rate ops 的实现就顺利成章了：

```
1: #define to_clk_fixed_rate(_hw) container_of(_hw, struct clk_fixed_rate, hw)
2:
3: static unsigned long clk_fixed_rate_recalc_rate(struct clk_hw *hw,
4:                 unsigned long parent_rate)
5: {
6:         return to_clk_fixed_rate(hw)->fixed_rate;
7: }
8:
9: const struct clk_ops clk_fixed_rate_ops = {
10:         .recalc_rate = clk_fixed_rate_recalc_rate,
11: };
```

```
12: EXPORT_SYMBOL_GPL(clk_fixed_rate_ops);
```

2）通过 DTS 注册

fixed rate clock 是非常简单的一种 clock，因而可以直接通过 DTS 的形式注册，clock framework 负责解析 DTS，并调用 API 注册 clock，如下：

```
1: #ifdef CONFIG_OF
2: /**
3:  * of_fixed_clk_setup() - Setup function for simple fixed rate clock
4:  */
5: void of_fixed_clk_setup(struct device_node *node)
6: {
7:        struct clk *clk;
8:        const char *clk_name = node->name;
9:        u32 rate;
10:
11:       if (of_property_read_u32(node, "clock-frequency", &rate))
12:             return;
13:
14:       of_property_read_string(node, "clock-output-names", &clk_name);
15:
16:       clk = clk_register_fixed_rate(NULL, clk_name, NULL, CLK_IS_ROOT, rate);
17:       if (!IS_ERR(clk))
18:             of_clk_add_provider(node, of_clk_src_simple_get, clk);
19: }
20: EXPORT_SYMBOL_GPL(of_fixed_clk_setup);
21: CLK_OF_DECLARE(fixed_clk, "fixed-clock", of_fixed_clk_setup);
22: #endif
```

首先看一下 CLK_OF_DECLARE 宏，它的定义位于“include/linux/clk-provider.h”中，负责在指定的 section 中（以 __clk_of_table 开始的位置），定义 struct of_device_id 类型的变量，并由 of_clk_init 接口解析、匹配，如果匹配成功，则执行相应的回调函数（这里为 of_fixed_clk_setup）；

初始化的时候，device tree 负责读取 DTS，并和这些变量的名字（这里为"fixed-clock"）匹配，如果匹配成功，则执行相应的回调函数（这里为 of_fixed_clk_setup）；

of_fixed_clk_setup 会解析两个 DTS 字段"clock-frequency"和"clock-output-names"，然后调用 clk_register_fixed_rate，注册 clock。注意，注册时的 flags 为 CLK_IS_ROOT，说明目前只支持 ROOT 类型的 clock 通过 DTS 注册；

最后，调用 of_clk_add_provider 接口，将该 clock 添加到 provider list 中，方便后续的查找使用。该接口会在后面再详细介绍。

of_clk_init 负责从 DTS 中扫描并初始化 clock provider，如下：

```
1: /* drivers/clk/clk.c */
2: /**
3:  * of_clk_init() - Scan and init clock providers from the DT
4:  * @matches: array of compatible values and init functions for providers.
5:  *
6:  * This function scans the device tree for matching clock providers and
7:  * calls their initialization functions
8:  */
9: void __init of_clk_init(const struct of_device_id *matches)
10: {
11:       struct device_node *np;
12:
13:       if (!matches)
```

```
14:             matches = __clk_of_table;
15:
16:     for_each_matching_node(np, matches) {
17:             const struct of_device_id *match = of_match_node(matches, np);
18:             of_clk_init_cb_t clk_init_cb = match->data;
19:             clk_init_cb(np);
20:     }
21: }
```

该接口有一个输入参数，用于指定需要扫描的 OF id，如果留空，则会扫描__clk_of_table，就是通过 CLK_OF_DECLARE 宏指定的 fixed rate 等类型的 clock。
在最新的 kernel 中，会在初始化代码（time_init）中以 NULL 为参数调用一次 of_clk_init，以便自动匹配并初始化 DTS 中的描述的类似 fixed rate 的 clock。
注 2：这里使用大量篇幅描述一个简单的 fixed rate clock 的注册方式，主要目的是给大家介绍一种通用的实现方式，或者说通用思路。后面其它类型的 clock，包括我们自定义的类型，实现方法都是一样的。这里就不罗嗦了，大家看代码就可以了。

3.4 gate、devider、mux、fixed factor、composite 以及自定义类型 clock 的注册

和 fixed rate 类似，不再一一说明。

4. 通用 API 的实现

4.1 clock get

clock get 是通过 clock 名称获取 struct clk 指针的过程，由 clk_get、devm_clk_get、clk_get_sys、of_clk_get、of_clk_get_by_name、of_clk_get_from_provider 等接口负责实现，这里以 clk_get 为例，分析其实现过程（位于 drivers/clk/clkdev.c 中）。
1）clk_get

```
1: struct clk *clk_get(struct device *dev, const char *con_id)
2: {
3:     const char *dev_id = dev ? dev_name(dev) : NULL;
4:     struct clk *clk;
5:
6:     if (dev) {
7:             clk = of_clk_get_by_name(dev->of_node, con_id);
8:             if (!IS_ERR(clk) && __clk_get(clk))
9:                     return clk;
10:    }
11:
12:    return clk_get_sys(dev_id, con_id);
13: }
```

如果提供了 struct device 指针，则调用 of_clk_get_by_name 接口，通过 device tree 接口获取 clock 指针。否则，如果没有提供设备指针，或者通过 device tree 不能正确获取 clock，则进一步调用 clk_get_sys。
这两个接口的定义如下。
2) of_clk_get_by_name
我们在“Linux common clock framework(2)_clock provider”中已经提过，clock consumer 会在本设备的 DTS 中，以 clocks、clock-names 为关键字，定义所需的 clock。系统启动后，device tree 会简单的解析，以 struct device_node 指针的形式，保存在本设备的 of_node 变量中。

而 of_clk_get_by_name，就是通过扫描所有“clock-names”中的值，和传入的 name 比较，如果相同，获得它的 index（即“clock-names”中的第几个），调用 of_clk_get，取得 clock 指针。

```
1: struct clk *of_clk_get_by_name(struct device_node *np, const char *name)
2: {
3:         struct clk *clk = ERR_PTR(-ENOENT);
4:
5:         /* Walk up the tree of devices looking for a clock that matches */
6:         while (np) {
7:                 int index = 0;
8:
9:                 /*
10:                 * For named clocks, first look up the name in the
11:                 * "clock-names" property.  If it cannot be found, then
12:                 * index will be an error code, and of_clk_get() will fail.
13:                 */
14:                if (name)
15:                        index = of_property_match_string(np, "clock-names", name);
16:                clk = of_clk_get(np, index);
17:                if (!IS_ERR(clk))
18:                        break;
19:                else if (name && index >= 0) {
20:                        pr_err("ERROR: could not get clock %s:%s(%i)\n",
21:                                np->full_name, name ? name : "", index);
22:                        return clk;
23:                }
24:
25:                /*
26:                 * No matching clock found on this node.  If the parent node
27:                 * has a "clock-ranges" property, then we can try one of its
28:                 * clocks.
29:                 */
30:                np = np->parent;
31:                if (np && !of_get_property(np, "clock-ranges", NULL))
32:                        break;
33:        }
34:
35:        return clk;
36: }
```

6~33 行，是一个 while 循环，用于扫描所有的 device_node；
14~15 行，只要 name 不为空，管它三七二十一，直接以 name 为参数，去和“clock-names”匹配，获得一个 index；
16~18 行，以返回的 index 为参数，调用 of_clk_get。这个 index 可能是 invalid，不过无所谓，最糟糕就是不能获得 clock 指针。如果成功获取，则退出，或者继续；
19~22 行，一个警告，如果 name 和 index 均合法，但是不能获得指针，则视为异常状况；
25~32 行，尝试”clock-ranges“熟悉，比较冷门，不介绍它。

再看一下 of_clk_get 接口。

```
1: struct clk *of_clk_get(struct device_node *np, int index)
2: {
3:         struct of_phandle_args clkspec;
4:         struct clk *clk;
5:         int rc;
```

```
6:
7:        if (index < 0)
8:              return ERR_PTR(-EINVAL);
9:
10:       rc = of_parse_phandle_with_args(np, "clocks", "#clock-cells", index,
11:                                       &clkspec);
12:       if (rc)
13:             return ERR_PTR(rc);
14:
15:       clk = of_clk_get_from_provider(&clkspec);
16:       of_node_put(clkspec.np);
17:       return clk;
18: }
```

10~13 行，通过 of_parse_phandle_with_args 接口，将 index 转换为 struct of_phandle_args 类型的参数句柄；
15 行，调用 of_clk_get_from_provider，获取 clock 指针。
of_clk_get_from_provider 的实现位于 drivers/clk/clk.c，通过便利 of_clk_providers 链表，并调用每一个 provider 的 get 回调函数，获取 clock 指针。如下：

```
1: struct clk *of_clk_get_from_provider(struct of_phandle_args *clkspec)
2: {
3:       struct of_clk_provider *provider;
4:       struct clk *clk = ERR_PTR(-ENOENT);
5:
6:       /* Check if we have such a provider in our array */
7:       mutex_lock(&of_clk_lock);
8:       list_for_each_entry(provider, &of_clk_providers, link) {
9:             if (provider->node == clkspec->np)
10:                  clk = provider->get(clkspec, provider->data);
11:            if (!IS_ERR(clk))
12:                  break;
13:      }
14:      mutex_unlock(&of_clk_lock);
15:
16:      return clk;
17: }
```

注 3：分析到这里之后，consumer 侧的获取流程已经很清晰，再结合“Linux common clock framework(2)_clock provider”中所介绍的 of_clk_add_provider 接口，整个流程都融汇贯通了。篇幅所限，有关 of_clk_add_provider 接口的实现，本文就不再分析了，感兴趣的读者可以自行阅读 kernel 代码。

3）clk_get_sys

clk_get_sys 接口是在调用者没有提供 struct device 指针或者通过 of_clk_get_xxx 获取 clock 失败时，获取 clock 指针的另一种手段。基于 kernel 大力推行 device tree 的现状，蜗蜗不建议使用这种过时的手段，就不分析了。

4.2 clk_prepare/clk_unprepare

prepare 和 unprepare 的的代码位于 drivers/clk/clk.c 中，分别由内部接口__clk_prepare 和__clk_unprepare 实现具体动作，如下：

```
1: int __clk_prepare(struct clk *clk)
2: {
3:       int ret = 0;
```

```
4:
5:        if (!clk)
6:                return 0;
7:
8:        if (clk->prepare_count == 0) {
9:                ret = __clk_prepare(clk->parent);
10:               if (ret)
11:                       return ret;
12:
13:               if (clk->ops->prepare) {
14:                       ret = clk->ops->prepare(clk->hw);
15:                       if (ret) {
16:                               __clk_unprepare(clk->parent);
17:                               return ret;
18:                       }
19:               }
20:       }
21:
22:       clk->prepare_count++;
23:
24:       return 0;
25: }
```

prepare 会维护一个 prepare_count，用于记录 prepare 的次数。且在 prepare_count 为零时：
递归 prepare 自己的 parent（有的话）；
调用 clk ops 中的 prepare 回调函数（有的话）。
unprepare 类似，不再分析。

4.3 clk_enable/clk_disable

enable/disable 和 prepare/unprepare 的实现逻辑基本一致，需要注意的是，enable/disable 时如果 prepare_count 为 0，则会报错并返回。

4.4 clock rate 有关的实现

clock rate 有关的实现包括 get、set 和 round 三类，让我们依次说明。
1）clk_get_rate 负责获取某个 clock 的当前 rate，代码如下：

```
1: /**
2:  * clk_get_rate - return the rate of clk
3:  * @clk: the clk whose rate is being returned
4:  *
5:  * Simply returns the cached rate of the clk, unless CLK_GET_RATE_NOCACHE flag
6:  * is set, which means a recalc_rate will be issued.
7:  * If clk is NULL then returns 0.
8:  */
9: unsigned long clk_get_rate(struct clk *clk)
10: {
11:       unsigned long rate;
12:
```

```
13:         clk_prepare_lock();
14:
15:         if (clk && (clk->flags & CLK_GET_RATE_NOCACHE))
16:                 __clk_recalc_rates(clk, 0);
17:
18:         rate = __clk_get_rate(clk);
19:         clk_prepare_unlock();
20:
21:         return rate;
22: }
23: EXPORT_SYMBOL_GPL(clk_get_rate);
```

a）如果该 clock 设置了 CLK_GET_RATE_NOCACHE 标志，获取 rate 前需要先调用__clk_recalc_rates 接口，根据当前硬件的实际情况，重新计算 rate。

__clk_recalc_rates 的逻辑是：如果提供了 recalc_rate ops，以 parent clock 的 rate 为参数，调用该 ops，否则，直接获取 parent 的 clock 值；然后，递归 recalc 所有 child clock。

b）调用__clk_get_rate 返回实际的 rate 值。

2）clk_round_rate，返回该 clock 支持的，和输入 rate 最接近的 rate 值（不做任何改动），实际是由内部函数 __clk_round_rate 实现，代码如下：

```
1: unsigned long __clk_round_rate(struct clk *clk, unsigned long rate)
2: {
3:         unsigned long parent_rate = 0;
4:
5:         if (!clk)
6:                 return 0;
7:
8:         if (!clk->ops->round_rate) {
9:                 if (clk->flags & CLK_SET_RATE_PARENT)
10:                        return __clk_round_rate(clk->parent, rate);
11:                else
12:                        return clk->rate;
13:        }
14:
15:        if (clk->parent)
16:                parent_rate = clk->parent->rate;
17:
18:        return clk->ops->round_rate(clk->hw, rate, &parent_rate);
19: }
```

a）18 行，如果该 clock 提供了 round_rate ops，直接调用该 ops。

需要说明的是，round_rate ops 接受两个参数，一个是需要 round 的 rate，另一个时 parent rate（以指针的形式提供）。它的意义是，对有些 clock 来说，如果需要得到一个比较接近的值，需要同时 round parent clock，因此会在该指针中返回 round 后的 parent clock。

b）9~10 行，如果 clock 没有提供 round_rate ops，且设置了 CLK_SET_RATE_PARENT 标志，则递归 round parent clock，背后的思考是，直接使用 parent clock 所能提供的最接近的 rate。

c）11~12，最后一种情况，直接返回原值，意味着无法 round。

3）clk_set_rate

set rate 的逻辑比较复杂，代码如下：

```
1: int clk_set_rate(struct clk *clk, unsigned long rate)
2: {
3:         struct clk *top, *fail_clk;
4:         int ret = 0;
```

```
5:
6:      /* prevent racing with updates to the clock topology */
7:      clk_prepare_lock();
9:      /* bail early if nothing to do */
10:     if (rate == clk->rate)
11:             goto out;
12:
13:     if ((clk->flags & CLK_SET_RATE_GATE) && clk->prepare_count) {
14:             ret = -EBUSY;
15:             goto out;
16:     }
18:     /* calculate new rates and get the topmost changed clock */
19:     top = clk_calc_new_rates(clk, rate);
20:     if (!top) {
21:             ret = -EINVAL;
22:             goto out;
23:     }
25:     /* notify that we are about to change rates */
26:     fail_clk = clk_propagate_rate_change(top, PRE_RATE_CHANGE);
27:     if (fail_clk) {
28:             pr_warn("%s: failed to set %s rate\n", __func__,
29:                             fail_clk->name);
30:             clk_propagate_rate_change(top, ABORT_RATE_CHANGE);
31:             ret = -EBUSY;
32:             goto out;
33:     }
35:     /* change the rates */
36:     clk_change_rate(top);
38: out:
39:     clk_prepare_unlock();
41:     return ret;
42: }
```

a）9~16，进行一些合法性判断。
b）19~23 行，调用 clk_calc_new_rates 接口，将需要设置的 rate 缓存在 new_rate 字段。
同时，获取设置该 rate 的话，需要修改到的最顶层的 clock。背后的逻辑是：如果该 clock 的 rate 改变，有可能需要通过改动 parent clock 的 rate 来实现，依次递归。
c）25~23，发送 rate 将要改变的通知。如果有 clock 不能接受改动，即 set rate 失败，再发送 rate 更改停止的通知。
d）调用 clk_change_rate，从最 top 的 clock 开始，依次设置新的 rate。
注 4：clock rate set 有 2 种场景，一是只需修改自身的配置，即可达到 rate set 的目的。第二种是需要同时修改 parent 的 rate（可向上递归）才能达成目的。看似简单的逻辑，里面却包含非常复杂的系统设计的知识。大家在使用 clock framework，知道有这回事即可，并尽可能的不要使用第二种场景，以保持系统的简洁性。

4.5 clock parent 有关的实现

parent 操作包括 get parent 和 set parent 两类。
get parent 的逻辑非常简单，直接从 clk->parent 指针中获取即可。
set parent 稍微复杂，需要执行 reparent 和 recalc_rates 动作，具体不再描述了。