Linux Kernel主要通过三类机制来实现SMP（Symmetric Multiprocessing，对称多核）系统CPU core的电源管理：

- cpu hotplug: 根据应用场景来up/down CPU
- cpuidle framework: 当cpu上没有可执行任务时，就会进入空闲状态
- cpufreq framework: 根据使用场景和系统负荷来调整CPU的电压和频率

cpufreq framework的核心功能，是通过调整CPU core的电压或频率，兼顾系统的性能和功耗。在不需要高性能时，降低电压或频率，以降低功耗；在需要高性能时，提高电压或频率，以提高性能。

cpufreq framework中的几个重要概念：

1. policy（策略）：同一个簇的CPU动态调频的一个集合结构体，包含了当前使用的governor和cpufreq driver
2. governor（调节器）：决定如何计算合适的频率或电压
3. cpufreq driver：来实现真正的调频执行工作（与平台相关）

常用的governor类型

1. Performance：总是将CPU置于最高性能的状态，即硬件所支持的最高频率、电压
2. Powersaving：总是将CPU置于最节能的状态，即硬件所支持的最低频率、电压
3. Ondemand：设置CPU负载的阈值T，当负载低于T时，调节至一个刚好能够满足当前负载需求的最低频/最低压；当负载高于T时，立即提升到最高性能状态
4. Conservative：跟Ondemand策略类似，设置CPU负载的阈值T，当负载低于T时，调节至一个刚好能够满足当前负载需求的最低频/最低压；但当负载高于T时，不是立即设置为最高性能状态，而是逐级升高主频/电压
5. Userspace：将控制接口通过sysfs开放给用户，由用户进行自定义策略
6. Schedutil：这是从Linux-4.7版本开始才引入的策略，其原理是根据调度器所提供的CPU利用率信息进行电压/频率调节，EAS使用schedutil进行调频

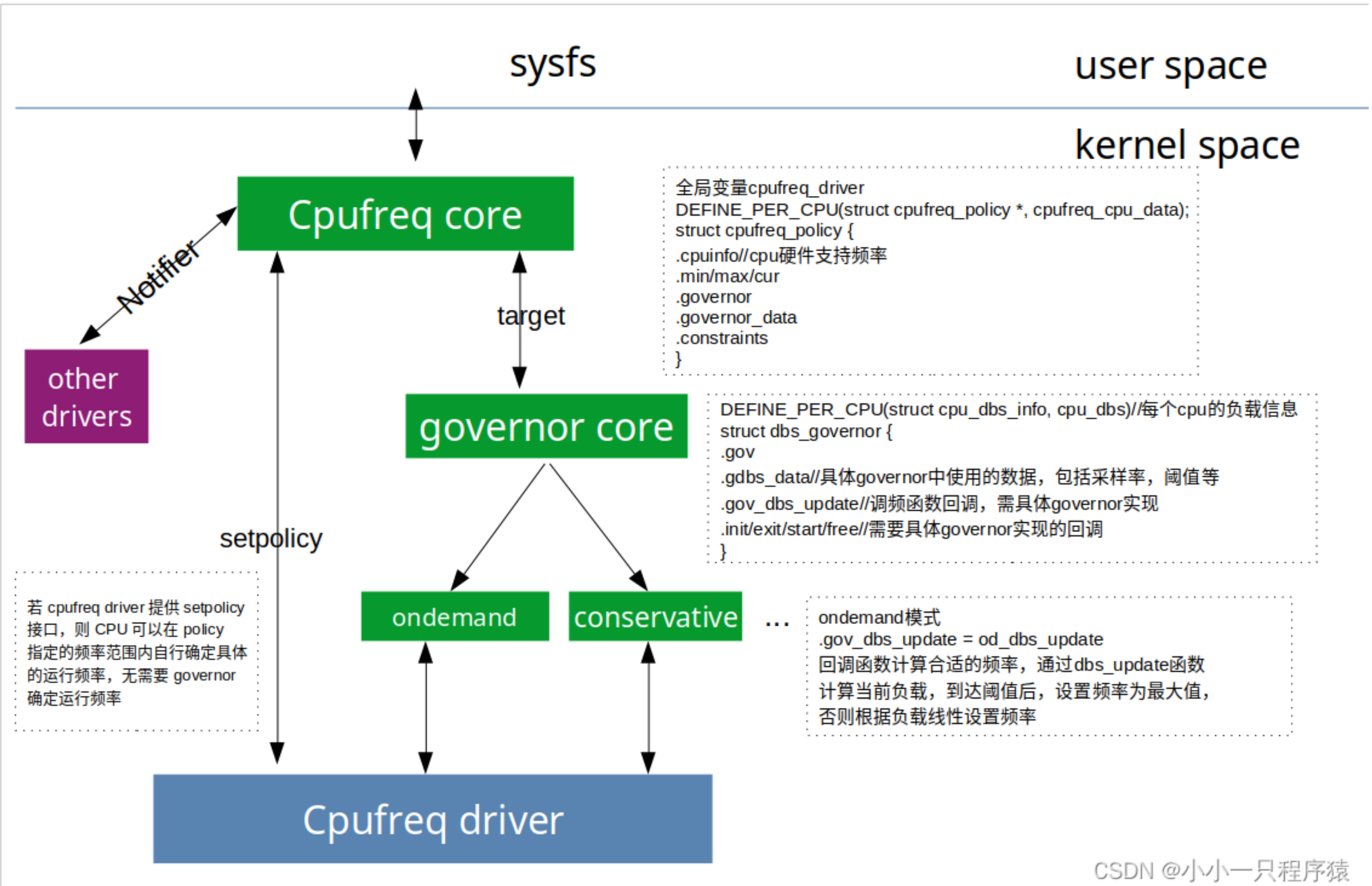
sysfs用户层接口，目录位于 `/sys/devices/system/cpu/cpufreq/policy`

```
root@Ubuntu:/sys/devices/system/cpu/cpufreq/policy0# ls
affected_cpus               related_cpus                   scaling_max_freq
cpuinfo_cur_freq            scaling_available_frequencies  scaling_min_freq
cpuinfo_max_freq            scaling_available_governors    scaling_setspeed
cpuinfo_min_freq            scaling_cur_freq               stats
cpuinfo_transition_latency  scaling_driver
ondemand                    scaling_governor
root@Ubuntu:/sys/devices/system/cpu/cpufreq/policy0# 
```

名称	说明
cpuinfo_max_freq	硬件所支持的最高频率
cpuinfo_min_freq	硬件所支持的最低频率
affected_cpus	该policy影响到哪些cpu（没显示offline状态的cpu）
related_cpus	该policy影响到的所有cpu，包括offline状态的cpu
scaling_max_freq	该policy支持调整的最高频率
scaling_min_freq	该policy支持调整的最低频率

名称	说明
scaling_cur_freq	policy当前设置的频率
scaling_available_governors	当前系统支持的governor
scaling_available_frequencies	支持的调频频率
scaling_driver	当前使用的调频驱动
scaling_governor	当前使用的governor
scaling_setspeed	在userspace模式下才能使用，手动设置频率

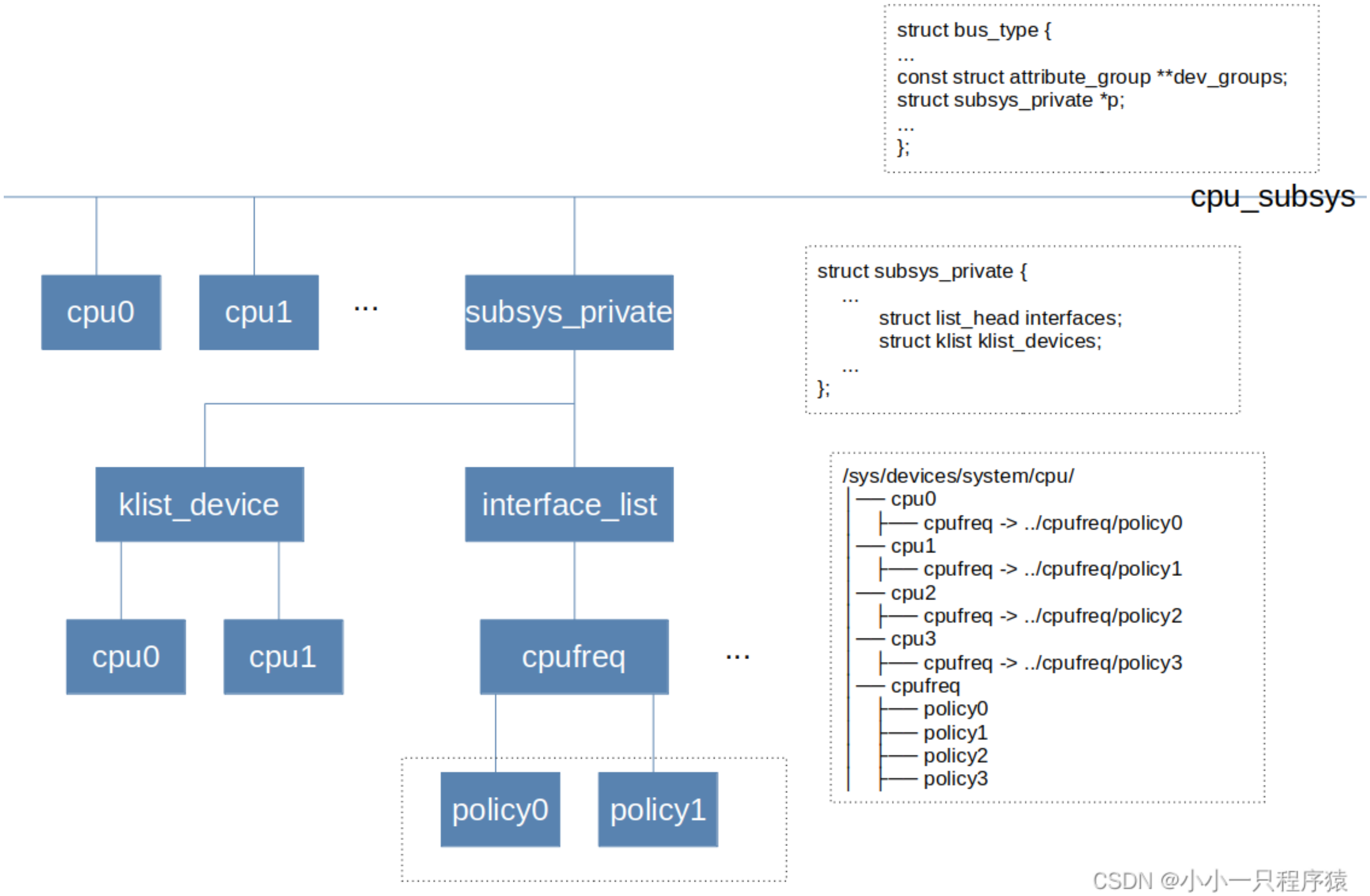
cpufreq软件架构



cpufreq core（可以理解为对policy的操作）：把一些公共的逻辑和接口代码抽象出来

- `cpufreq` 作为所有cpu设备的一个功能，注册到了 `cpu_subsys` 总线上
- 对上以sysfs的形式向用户空间提供统一的接口，以notifier的形式向其他driver提供频率变化的通知
- 对下提供CPU频率和电压控制的驱动框架，方便底层driver的开发，同时提供governor框架，用于实现不同的频率调整机制
- 内部封装各种逻辑，主要围绕 `struct cpufreq_policy` `struct cpufreq_driver` `struct cpufreq_governor` 三个数据结构进行

kernel使用 `struct cpufreq_policy` 用来抽象cpufreq，它从一定程度上代表了一个CPU簇的cpufreq的属性



`cpufreq_policy` 结构体

```c
struct cpufreq_cpuinfo {
	unsigned int		max_freq;			// cpu最大频率
	unsigned int		min_freq;			// cpu最小频率

	/* in 10^(-9) s = nanoseconds */
	unsigned int		transition_latency;	// cpu频率转换时间 单位: ns
};

struct cpufreq_policy {
	/* CPUs sharing clock, require sw coordination */
	cpumask_var_t		cpus;	/* Online CPUs only */
	cpumask_var_t		related_cpus; /* Online + Offline CPUs */
	cpumask_var_t		real_cpus; /* Related and present */

	unsigned int		shared_type; /* ACPI: ANY or ALL affected CPUs
						should set cpufreq */
	unsigned int		cpu;    /* cpu managing this policy, must be online */

	struct clk		*clk;
	struct cpufreq_cpuinfo	cpuinfo;/* see above */

	unsigned int		min;    /* in kHz */
	unsigned int		max;    /* in kHz */
	unsigned int		cur;    /* in kHz, only needed if cpufreq
					 * governors are used */

	struct cpufreq_governor	*governor; /* see below */
	void			*governor_data;
	char			last_governor[CPUFREQ_NAME_LEN]; /* last governor used */

	struct work_struct	update; /* if update_policy() needs to be
					 * called, but you're in IRQ context */

	struct cpufreq_user_policy user_policy;
	struct cpufreq_frequency_table	*freq_table;
	enum cpufreq_table_sorting freq_table_sorted;

	struct list_head	policy_list;
	struct kobject		kobj;
	struct completion	kobj_unregister;

	/*
	 * Preferred average time interval between consecutive invocations of
	 * the driver to set the frequency for this policy.  To be set by the
	 * scaling driver (0, which is the default, means no preference).
	 */
	unsigned int		transition_delay_us;

	/*
	 * Remote DVFS flag (Not added to the driver structure as we don't want
	 * to access another structure from scheduler hotpath).
	 *
	 * Should be set if CPUs can do DVFS on behalf of other CPUs from
	 * different cpufreq policies.
	 */
	bool			dvfs_possible_from_any_cpu;

	 /* Cached frequency lookup from cpufreq_driver_resolve_freq. */
	unsigned int cached_target_freq;
```

```
        int cached_resolved_idx;

        /* cpufreq-stats */
        struct cpufreq_stats    *stats;

        /* For cpufreq driver's internal use */
        void                    *driver_data;

        ...
};
```

`driver/cpufreq/cpufreq.c` 中定义了一个全局的percpu变量

```
static DEFINE_PER_CPU(struct cpufreq_policy *, cpufreq_cpu_data);
```

这里对应E2000 sysfs中3个policy文件夹，两个小核在一个簇中，使用1个policy，另外两个大核分别对应1个policy

```
root@Ubuntu:/sys/devices/system/cpu/cpufreq# tree -L 1
.
├── policy0
├── policy2
└── policy3
```

> per-CPU变量是linux系统一个非常重要的特性，它为系统中的每个处理器都分配了该变量的副本。这样做的好处是，在多处理器系统中，当处理器操作属于它的变量副本时，不需要考虑与其他处理器的竞争的问题，同时该副本还可以充分利用处理器本地的硬件缓冲cache来提供访问速度

cpufreq初始化过程

`cpufreq_driver` 结构体如下：

```c
struct cpufreq_driver {
        char            name[CPUFREQ_NAME_LEN];
        u16                 flags;
        void            *driver_data;

        /* needed by all drivers */
        int             (*init)(struct cpufreq_policy *policy);
        int             (*verify)(struct cpufreq_policy_data *policy);
        int             (*target_index)(struct cpufreq_policy *policy,
                                        unsigned int index);
        unsigned int    (*fast_switch)(struct cpufreq_policy *policy,
                                       unsigned int target_freq);

        /* should be defined, if possible */
        unsigned int    (*get)(unsigned int cpu);

        /* Called to update policy limits on firmware notifications. */
        void            (*update_limits)(unsigned int cpu);

        int             (*online)(struct cpufreq_policy *policy);
        int             (*offline)(struct cpufreq_policy *policy);
        int             (*exit)(struct cpufreq_policy *policy);

        struct freq_attr **attr;
};
```

cpufreq初始化概述

在kconfig中(CPU Power Management -> CPU Frequency scaling)可以对cpufreq进行配置，可以配置支持的governor及系统默认的governor，以及cpufreq调频driver，例如Phytium E2000 5.10内核的配置如下，默认使用schedutil governor，根据调度器所提供的CPU利用率信息进行电压/频率调节，EAS能源感知依赖该governor工作：

```
[*] CPU Frequency scaling
[*]   CPU frequency transition statistics
      Default CPUFreq governor (schedutil)  --->
-*-   'performance' governor
<M>   'powersave' governor
<*>   'userspace' governor for userspace frequency scaling
<*>   'ondemand' cpufreq policy governor
<M>   'conservative' cpufreq governor
-*-   'schedutil' cpufreq policy governor
      *** CPU frequency scaling drivers ***
<*>   Generic DT based cpufreq driver
<M>   CPUFreq driver based on the ACPI CPPC spec
<*>   SCPI based CPUFreq driver
<*>   SCMI based CPUFreq driver
```

cpufreq的初始化从cpufreq_drvier注册开始， `cpufreq_register_driver()` 函数为cpufreq驱动注册的入口，驱动程序通过调用该函数进行初始化，传入相关的 `struct cpufreq_driver` ， `cpufreq_register_driver()` 会调用 `subsys_interface_register()` 最终执行回调函数 `cpufreq_add_dev` ，然后调用 `cpufreq_online()` 走初始化流程

Performance Domain opp（Operating Performance Points）表初始化

OPP表的定义：域中每个设备支持的电压和频率的离散元组的集合称为Operating Performance Points（OPP）,内核设备树opp文档 `Documentation/devicetree/bindings/opp/opp.txt`

假设一个CPU设备支持如下的电压和频率关系：
{300MHz at minimum voltage of 1V}
{800MHz at minimum voltage of 1.2V}
{1GHz at minimum voltage of 1.3V}
用OPP表示就可以用{Hz, uV}方式表示如下:
{300000000, 1000000}
{800000000, 1200000}
{1000000000, 1300000}

这里初始化的就是各个性能域（即不同CPU簇）的OPP表，在E2000平台中是通过SCMI的Performace domain management protocol协议获取PERFORMANCE_DESCRIBE_LEVELS这个参数表，具体的协议实现源码在 `drivers/firmware/arm_scmi/perf.c` 里面， `perf.c` 实现了SCMI的Performance domain managment protocol，scmi cpufreq_drvier也是通过 `perf_ops` 函数集进行调频

```c
// include/linux/scmi_protocol.h
// 抽象描述scmi协议的结构体，相应的ops操作集对应scmi的一个协议
struct scmi_handle {
	struct device *dev;
	struct scmi_revision_info *version;
	const struct scmi_perf_ops *perf_ops;
	const struct scmi_clk_ops *clk_ops;
	const struct scmi_power_ops *power_ops;
	const struct scmi_sensor_ops *sensor_ops;
	const struct scmi_reset_ops *reset_ops;
	const struct scmi_notify_ops *notify_ops;
	/* for protocol internal use */
	void *perf_priv;
	void *clk_priv;
	void *power_priv;
	void *sensor_priv;
	void *reset_priv;
	void *notify_priv;
	void *system_priv;
};

// scmi_handle这个结构体实现的就是scmi整个协议的处理
static const struct scmi_handle *handle;

// include/linux/scmi_protocol.h
/**
 * struct scmi_perf_ops - represents the various operations provided
 *	by SCMI Performance Protocol
 *
 * @limits_set: sets limits on the performance level of a domain
 * @limits_get: gets limits on the performance level of a domain
 * @level_set: sets the performance level of a domain
 * @level_get: gets the performance level of a domain
 * @device_domain_id: gets the scmi domain id for a given device
 * @transition_latency_get: gets the DVFS transition latency for a given device
 * @device_opps_add: adds all the OPPs for a given device
 * @freq_set: sets the frequency for a given device using sustained frequency
 *	to sustained performance level mapping
 * @freq_get: gets the frequency for a given device using sustained frequency
 *	to sustained performance level mapping
 * @est_power_get: gets the estimated power cost for a given performance domain
 *	at a given frequency
 */
struct scmi_perf_ops {
	int (*limits_set)(const struct scmi_handle *handle, u32 domain,
			  u32 max_perf, u32 min_perf);
	int (*limits_get)(const struct scmi_handle *handle, u32 domain,
			  u32 *max_perf, u32 *min_perf);
	int (*level_set)(const struct scmi_handle *handle, u32 domain,
			 u32 level, bool poll);
	int (*level_get)(const struct scmi_handle *handle, u32 domain,
			 u32 *level, bool poll);
	int (*device_domain_id)(struct device *dev);
	int (*transition_latency_get)(const struct scmi_handle *handle,
				      struct device *dev);
	int (*device_opps_add)(const struct scmi_handle *handle,
			       struct device *dev);
	int (*freq_set)(const struct scmi_handle *handle, u32 domain,
			unsigned long rate, bool poll);
```

```c
        int (*freq_get)(const struct scmi_handle *handle, u32 domain,
                        unsigned long *rate, bool poll);
        int (*est_power_get)(const struct scmi_handle *handle, u32 domain,
                             unsigned long *rate, unsigned long *power);
        bool (*fast_switch_possible)(const struct scmi_handle *handle,
                                     struct device *dev);
};

// scmi performance domain management protocol(性能域管理相关协议 0x13), messageid: 0x03
// scmi opp结构体定义
struct scmi_opp {
        u32 perf;               // 性能级别，单位KHz
        u32 power;              // 当前性能级别的功耗
        u32 trans_latency_us;   // 切换延时
};

// scmi performance domain management protocol(性能域管理相关协议 0x13)对应操作函数集
// scmi cpufreq_driver 主要利用这个函数集进行调频相关操作
// 对应Performace domain management protocol各个message_id
static const struct scmi_perf_ops perf_ops = {
        .limits_set = scmi_perf_limits_set,
        .limits_get = scmi_perf_limits_get,
        .level_set = scmi_perf_level_set,
        .level_get = scmi_perf_level_get,
        .device_domain_id = scmi_dev_domain_id,
        .transition_latency_get = scmi_dvfs_transition_latency_get,
        .device_opps_add = scmi_dvfs_device_opps_add,
        .freq_set = scmi_dvfs_freq_set,
        .freq_get = scmi_dvfs_freq_get,
        .est_power_get = scmi_dvfs_est_power_get,
        .fast_switch_possible = scmi_fast_switch_possible,
};

// 在这个宏进行SCMI performance domain management protocol协议的初始化
DEFINE_SCMI_PROTOCOL_REGISTER_UNREGISTER(SCMI_PROTOCOL_PERF, perf)

#define DEFINE_SCMI_PROTOCOL_REGISTER_UNREGISTER(id, name) \
int __init scmi_##name##_register(void) \
{ \
        return scmi_protocol_register((id), &scmi_##name##_protocol_init); \
} \
\
void __exit scmi_##name##_unregister(void) \
{ \
        scmi_protocol_unregister((id)); \
}
// 展开该宏
int __init scmi_perf_register(void)
{
        return scmi_protocol_register(SCMI_PROTOCOL_PER, &scmi_perf_protocol_init);
}

// 初始化过程中调用了scmi_perf_protocol_init();
static int scmi_perf_protocol_init(struct scmi_handle *handle)
{
        int domain;
        u32 version;
        struct scmi_perf_info *pinfo;
```

```
        // 获取当前perf domain management协议版本
        scmi_version_get(handle, SCMI_PROTOCOL_PERF, &version);

        dev_dbg(handle->dev, "Performance Version %d.%d\n",
                PROTOCOL_REV_MAJOR(version), PROTOCOL_REV_MINOR(version));

        pinfo = devm_kzalloc(handle->dev, sizeof(*pinfo), GFP_KERNEL);
        if (!pinfo)
                return -ENOMEM;

        scmi_perf_attributes_get(handle, pinfo);

        pinfo->dom_info = devm_kcalloc(handle->dev, pinfo->num_domains,
                                       sizeof(*pinfo->dom_info), GFP_KERNEL);
        if (!pinfo->dom_info)
                return -ENOMEM;

        // 遍历每个performance_domain，获取performance domain的属性和performance level参数
        for (domain = 0; domain < pinfo->num_domains; domain++) {
                struct perf_dom_info *dom = pinfo->dom_info + domain;

                // 获取performance domain属性
                scmi_perf_domain_attributes_get(handle, domain, dom);
                // 获取performance level参数即opp表
                scmi_perf_describe_levels_get(handle, domain, dom);

                if (dom->perf_fastchannels)
                        scmi_perf_domain_init_fc(handle, domain, &dom->fc_info);
        }

        scmi_register_protocol_events(handle,
                                      SCMI_PROTOCOL_PERF, SCMI_PROTO_QUEUE_SZ,
                                      &perf_event_ops, perf_events,
                                      ARRAY_SIZE(perf_events),
                                      pinfo->num_domains);

        pinfo->version = version;
        handle->perf_ops = &perf_ops;
        handle->perf_priv = pinfo;

        return 0;
}
```

最终获取得到的OPP表如下

FTC664 (capacity: 576)		
perf_value	**power**	**trans_latency**
250MHz	79	1000
500MHz	197	1000
1000MHz	409	1000
2000MHz	839	1000

FTC310 (capacity: 1024)		
perf_value	**power**	**trans_latency**
187.5MHz	1	1000
375MHz	9	1000
750MHz	55	1000
1500MHz	125	1000

cpufreq初始化过程

```
// driver/base/cpu.c
struct bus_type cpu_subsys = {
        .name = "cpu",
        .dev_name = "cpu",
        .match = cpu_subsys_match,
        .online = cpu_subsys_online,
        .offline = cpu_subsys_offline,
};

// driver/cpufreq/cpufreq.c
// 指向当前使用的cpufreq_driver
static struct cpufreq_driver *cpufreq_driver;

// cpufreq subsys接口，用来挂到CPU subsys总线上
static struct subsys_interface cpufreq_interface = {
        .name           = "cpufreq",
        .subsys         = &cpu_subsys,
        .add_dev        = cpufreq_add_dev,
        .remove_dev     = cpufreq_remove_dev,
};

// scmi cpufreq_driver结构体定义
static struct cpufreq_driver scmi_cpufreq_driver = {
        .name   = "scmi",
        .flags  = CPUFREQ_STICKY | CPUFREQ_HAVE_GOVERNOR_PER_POLICY |
                  CPUFREQ_NEED_INITIAL_FREQ_CHECK,
        .verify = cpufreq_generic_frequency_table_verify,
        .attr   = cpufreq_generic_attr,
        .target_index   = scmi_cpufreq_set_target,
        .fast_switch    = scmi_cpufreq_fast_switch,
        .get    = scmi_cpufreq_get_rate,
        .init   = scmi_cpufreq_init,
        .exit   = scmi_cpufreq_exit,
        .ready  = scmi_cpufreq_ready,
};

// 为每个cluster定义一个cpufreq_policy结构体，对每个cluster上的CPU进行调频管理
// 其中又分别进行cpufreq_driver的初始化和governor的初始化
static DEFINE_PER_CPU(struct cpufreq_policy *, cpufreq_cpu_data);

// cpufreq驱动框架初始化过程，整个过程都围绕着policy这个结构体进行，逐步进行初始化
cpufreq_register_driver(&scmi_cpufreq_driver);
        subsys_interface_register(&cpufreq_interface);
                cpufreq_add_dev(dev, sif);
                        cpufreq_online(cpu);
                                // 初步初始化policy
                                policy = cpufreq_policy_alloc(cpu);
                                // 调用cpufreq_drvier init接口，完善policy结构体
                                // 将opp表添加到对应的device，通过dev_pm_opp_add接口
                                // 生成频率表 freq_table
                                cpufreq_driver->init(policy) -> scmi_cpufreq_init(policy)
                                cpufreq_table_validate_and_sort(policy);
                                // 创建/sys/device/system/cpu/cpux目录下的cpufreq符号链接
                                add_cpu_dev_symlink();
                                freq_qos_and_request();
                                blocking_notifier_call_chain();
                                // CPU进行频率调整，使当前运行频率在频率表中
```

```
__cpufreq_driver_target();
// 创建sys节点，/sys/device/system/cpu/cpufreq/policyx目录下的一些可选属性
cpufreq_add_dev_interface(policy);
cpufreq_stats_create_table(policy);
list_add(&policy->polic_list, &cpufreq_policy_list);
// 使用默认governor初始化policy
cpufreq_init_policy();
```

cpufreq_governor的初始化过程

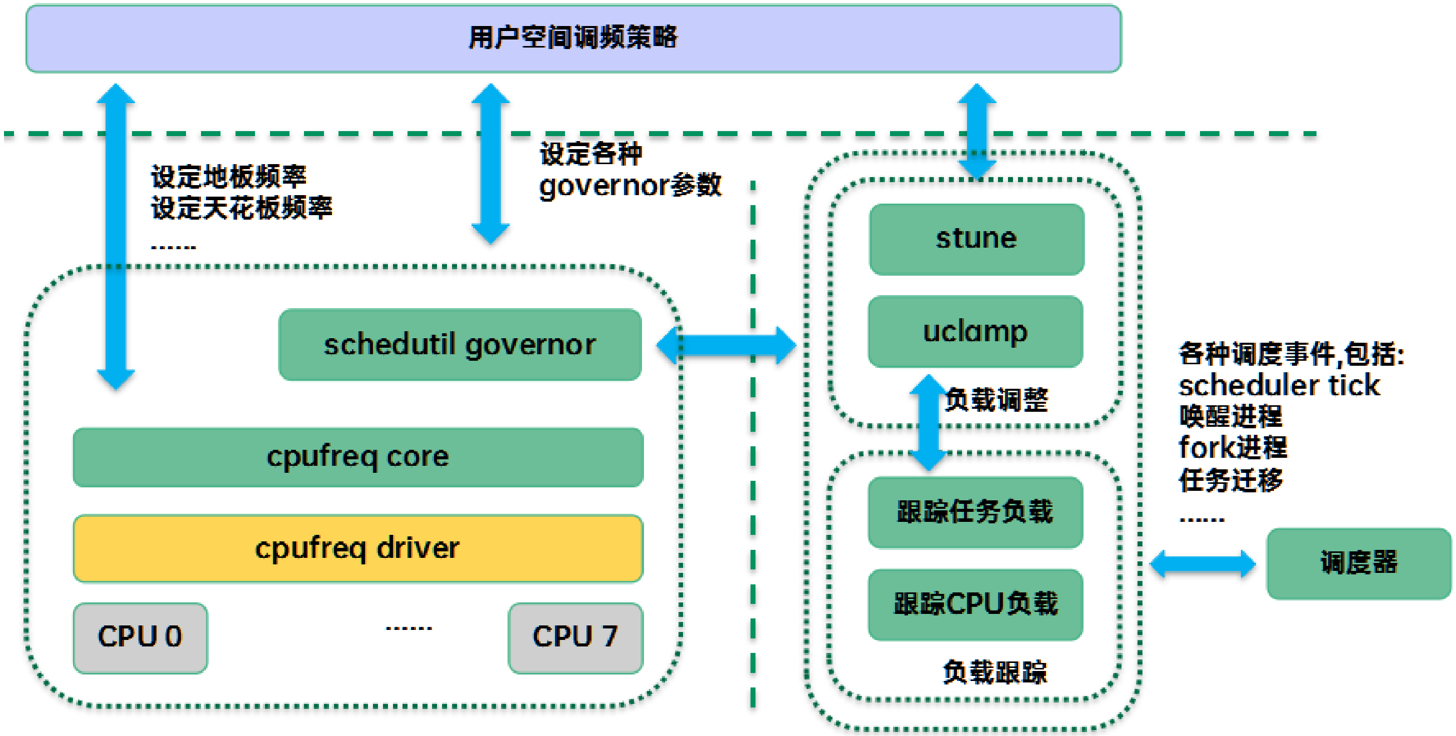
cpufreq governor的初始化过程，在cpufreq_init_policy(policy)中进行，这里以ondemand为例进行分析

```c
// include/linux/cpufreq.h
struct cpufreq_governor {
        char    name[CPUFREQ_NAME_LEN];
        int     (*init)(struct cpufreq_policy *policy);
        void    (*exit)(struct cpufreq_policy *policy);
        int     (*start)(struct cpufreq_policy *policy);
        void    (*stop)(struct cpufreq_policy *policy);
        void    (*limits)(struct cpufreq_policy *policy);
        ssize_t (*show_setspeed)        (struct cpufreq_policy *policy,
                                         char *buf);
        int     (*store_setspeed)       (struct cpufreq_policy *policy,
                                         unsigned int freq);
        struct list_head        governor_list;
        struct module           *owner;
        u8                      flags;
};

/* Common Governor data across policies */
// 抽象出的governor调度器结构体
// drivers/cpufreq/cpufreq_governor.h
struct dbs_governor {
        struct cpufreq_governor gov;
        struct kobj_type kobj_type;
        /*
         * Common data for platforms that don't set
         * CPUFREQ_HAVE_GOVERNOR_PER_POLICY
         */
        struct dbs_data *gdbs_data;

        unsigned int (*gov_dbs_update)(struct cpufreq_policy *policy);
        struct policy_dbs_info *(*alloc)(void);
        void (*free)(struct policy_dbs_info *policy_dbs);
        int (*init)(struct dbs_data *dbs_data);
        void (*exit)(struct dbs_data *dbs_data);
        void (*start)(struct cpufreq_policy *policy);
};

// drivers/cpufreq/cpufreq_governor.h
// governor初始化宏
#define CPUFREQ_DBS_GOVERNOR_INITIALIZER(_name_)                        \
        {                                                               \
                .name = _name_,                                         \
                .flags = CPUFREQ_GOV_DYNAMIC_SWITCHING,                 \
                .owner = THIS_MODULE,                                   \
                .init = cpufreq_dbs_governor_init,                      \
                .exit = cpufreq_dbs_governor_exit,                      \
                .start = cpufreq_dbs_governor_start,                    \
                .stop = cpufreq_dbs_governor_stop,                      \
                .limits = cpufreq_dbs_governor_limits,                  \
        }

// ondemand governor定义
// driver/cpufreq/cpufreq_ondemand.c
static struct dbs_governor od_dbs_gov = {
        .gov = CPUFREQ_DBS_GOVERNOR_INITIALIZER("ondemand"),
        .kobj_type = { .default_attrs = od_attributes },
        .gov_dbs_update = od_dbs_update,
        .alloc = od_alloc,
        .free = od_free,
```

```
        .init = od_init,
        .exit = od_exit,
        .start = od_start,
};

// 初始化governor
// 该函数会在governor模块驱动的入口函数调用
// 只要编译该模块，就会注册到cpufreq framework中
#define CPU_FREQ_GOV_ONDEMAND   (od_dbs_gov.gov)
cpufreq_governor_init(CPU_FREQ_GOV_ONDEMAND);

#define cpufreq_governor_init(__governor)                       \
static int __init __governor##_init(void)                       \
{                                                               \
        return cpufreq_register_governor(&__governor);  \
}                                                               \
core_initcall(__governor##_init)

// 在cpufreq_online()中调用默认governor对policy进行完善，启动当前governor
cpufreq_init_policy(policy);
        gov = get_governor(default_governor);
        // 设置新的governor
        cpufreq_set_policy(policy, gov, pol);
                cpufreq_init_governor(policy);
                        policy->governor->init();  -> cpufreq_dbs_governor_init(policy);
                                gov->init(dbs_data); -> odinit(dbs_data);
                cpufreq_start_governor(policy);
                        policy->governor->start(); -> cpufreq_dbs_governor_start(policy);
                                // 设置governor回调函数， 以ondemand为例
                                gov_set_update_util(policy_dbs, sampling_rate);
                                        cpufreq_add_update_util_hook(cpu, &cdbs->updata_util,
                                                                dbs_update_util_handler);
```

启动governor中比较重要的是设置调频回调函数,该函数是真正调频时计算合适频率的函数

schedutil调节器



sugov作为一种内核调频策略模块，它主要是根据当前CPU的利用率进行调频。因此，sugov会注册一个callback函数（sugov_update_shared/sugov_update_single）到调度器负载跟踪模块，当CPU util发生变化的时候就会调用该callback函数，检查一下当前CPU频率是否和当前的CPU util匹配，如果不匹配，那么就进行升频或者降频。

`sugov_tunables` 结构体，用来描述sugov的可调参数

```
// sugov_tunables结构体
struct sugov_tunables {
        struct gov_attr_set attr_set;
        // 目前sugov只有这一个可调参数，该参数用来限制连续调频的间隔时间
        unsigned int rate_limit_us;
};
```

`sugov_policy` 结构体，sugov为每个cluster构建了该数据结构，记录每个cluster的调频数据信息

```c
// sugov_policy结构体，为每个簇构建了该数据结构，记录每个簇的调频数据信息
struct sugov_policy {
        // 指向cpufreq framework层的policy对象
        struct cpufreq_policy   *policy;
        // sugov的可调参数
        struct sugov_tunables   *tunables;
        struct list_head        tunables_hook;

        raw_spinlock_t          update_lock;    /* For shared policies */
        // 记录上次进行频率调整的时间点
        u64                     last_freq_update_time;
        // 最小调频时间间隔
        s64                     freq_update_delay_ns;
        // 下一个需要调整到的频率值，回调函数主要是计算这个参数
        unsigned int            next_freq;
        // 根据CPU util计算出来的原始频率，在频率表中向上找最接近的频率进行调整
        unsigned int            cached_raw_freq;

        /* The next fields are only needed if fast switch cannot be used: */
        struct                  irq_work irq_work;
        struct                  kthread_work work;
        struct                  mutex work_lock;
        struct                  kthread_worker worker;
        struct task_struct      *thread;
        bool                    work_in_progress;

        bool                    limits_changed;
        bool                    need_freq_update;
};
```

sugov_cpu 结构体，sugov为每个cpu构建了该数据结构，记录per-cpu的调频数据信息

```c
struct sugov_cpu {
        // 保存了cpu util变化后的回调函数
        struct update_util_data update_util;
        // 该sugov_cpu对应的sugov_policy对象
        struct sugov_policy     *sg_policy;
        // 对应的CPU id
        unsigned int            cpu;

        bool                    iowait_boost_pending;
        unsigned int            iowait_boost;
        // 上一次cpu负载变化驱动调频的时间点，util更新的时间点
        u64                     last_update;

        unsigned long           bw_dl;
        // 该CPU的最大算力，即最大utilities，归一化到1024
        unsigned long           max;

        /* The field below is for single-CPU policies only: */
#ifdef CONFIG_NO_HZ_COMMON
        unsigned long           saved_idle_calls;
#endif
};
```

sugov初始化过程和ondemand初始化过程相似，当内核设定默认governor为sugov时，在 `cpufreq_init_governor(policy);` 中会调用 `sugov_init()` 初始化sugov，然后调用 `sugov_start()` 设置调频回调函数，每当CPU利用率发生变化的时候，调度器都会调用 `cpufreq_update_util()` 通知sugov，在 `cpufreq_update_util()` 被调用时，即任务调度后CPU当前的util发生变化，会调用sugov的回调函数进行调频， `sugov_update_shared()` 当一个簇中有多个CPU调用该回调，遍历簇上的CPU找到当前最大util的CPU，然后根据该util映射到频率； `sugov_update_single()` 即一个簇上单个CPU的情况直接根据该CPUte_shared()` util计算频率

调度事件的发生还是非常密集的，特别是在重载的情况下，很多任务可能执行若干个us就切换出去了。如果每次都计算CPU util看看是否需要调整频率，那么本身sugov就给系统带来较重的负荷，因此并非每次调频时机都会真正执行调频检查，sugov设置了一个最小调频间隔，小于这个间隔的调频请求会被过滤掉。

schedutil频率计算过程

```
// sugov_start会遍历该sugov policy（cluster）中的所有cpu
// 调用cpufreq_add_update_util_hook为sugov cpu注册调频回调函数，代码逻辑如下:
static int sugov_start(struct cpufreq_policy *policy)
{
        ...
        for_each_cpu(cpu, policy->cpus) {
                struct sugov_cpu *sg_cpu = &per_cpu(sugov_cpu, cpu);
                // 设置governor 计算回调函数，cpufreq_update_util()被调用时
                // 即任务调度后CPU当前的util发生变化，会调用sugov的回调函数进行调频计算
                cpufreq_add_update_util_hook(cpu, &sg_cpu->update_util,
                                             policy_is_shared(policy) ?
                                                        sugov_update_shared :
                                                        sugov_update_single);
        }
        ...
}

// schedutil频率计算过程
sugov_update_single();
        // 调频最小间隔时间检查，小于设定时间，直接返回
        sugov_should_update_freq();
        util = sugov_get_util(sg_cpuu);
                schedutil_cpu_util(sg_cpu->cpu, util, max, FREQUENCY_UTIL, NULL);
        // 根据当前CPU的util映射到具体的频率上
        next_f = get_next_freq(sg_policy, util, max);
        // 调用cpufreq_driver进行调频
        sugov_deferred_update(sg_policy, time, next_f);
                __cpufreq_driver_target()

// 计算cpu当前的utility
unsigned long schedutil_cpu_util(int cpu, unsigned long util_cfs,
                                 unsigned long max, enum schedutil_type type,
                                 struct task_struct *p)
{
        unsigned long dl_util, util, irq;
        struct rq *rq = cpu_rq(cpu);

        if (!uclamp_is_used() &&
            type == FREQUENCY_UTIL && rt_rq_is_runnable(&rq->rt)) {
                return max;
        }

        // 如果CPU处理了过多的中断服务函数，irq负载已经高过CPU最大算力，直接返回最大算力
        irq = cpu_util_irq(rq);
        if (unlikely(irq >= max))
                return max;

        // 累加了cfs和rt任务的utility
        util = util_cfs + cpu_util_rt(rq);
        if (type == FREQUENCY_UTIL)
                util = uclamp_rq_util_with(rq, util, p);

        dl_util = cpu_util_dl(rq);

        if (util + dl_util >= max)
                return max;
```

```c
	/*
	 * OTOH, for energy computation we need the estimated running time, so
	 * include util_dl and ignore dl_bw.
	 */
	if (type == ENERGY_UTIL)
		util += dl_util;

	/*
	 * There is still idle time; further improve the number by using the
	 * irq metric. Because IRQ/steal time is hidden from the task clock we
	 * need to scale the task numbers:
	 *
	 *              max - irq
	 *   U' = irq + --------- * U
	 *                 max
	 */
	// irq会偷走一部分的cpu算力，从而让其capacity没有那么大。
	// 这里通过scale_irq_capacity对任务的utility进行调整
	util = scale_irq_capacity(util, irq, max);
	util += irq;

	/*
	 * Bandwidth required by DEADLINE must always be granted while, for
	 * FAIR and RT, we use blocked utilization of IDLE CPUs as a mechanism
	 * to gracefully reduce the frequency when no tasks show up for longer
	 * periods of time.
	 *
	 * Ideally we would like to set bw_dl as min/guaranteed freq and util +
	 * bw_dl as requested freq. However, cpufreq is not yet ready for such
	 * an interface. So, we only do the latter for now.
	 */
	if (type == FREQUENCY_UTIL)
		util += cpu_bw_dl(rq);

	return min(max, util);
}

// 根据当前CPU计算的util映射对应频率
static unsigned int get_next_freq(struct sugov_policy *sg_policy,
				  unsigned long util, unsigned long max)
{
	struct cpufreq_policy *policy = sg_policy->policy;
	// 先取得当前CPU的最大频率
	unsigned int freq = arch_scale_freq_invariant() ?
				policy->cpuinfo.max_freq : policy->cur;
	// 计算当前util对应频率，计算公式: freq = (1.25) * freq * util / max
	// 这里冗余了25%的算力余量
	freq = map_util_freq(util, freq, max);

	// 若计算出的freq和上次缓存的一样，则实际调整的next_freq计算后肯定也是一样的，直接返回
	// 上次记录的频率值
	if (freq == sg_policy->cached_raw_freq && !sg_policy->need_freq_update)
		return sg_policy->next_freq;

	sg_policy->cached_raw_freq = freq;

	// 根据当前算的freq，在CPU频率表上查找对应的频率
	freq = cpufreq_driver_resolve_freq(policy, freq);
```

```
        return freq;
}
```

EAS能源感知调度

EAS整体框架

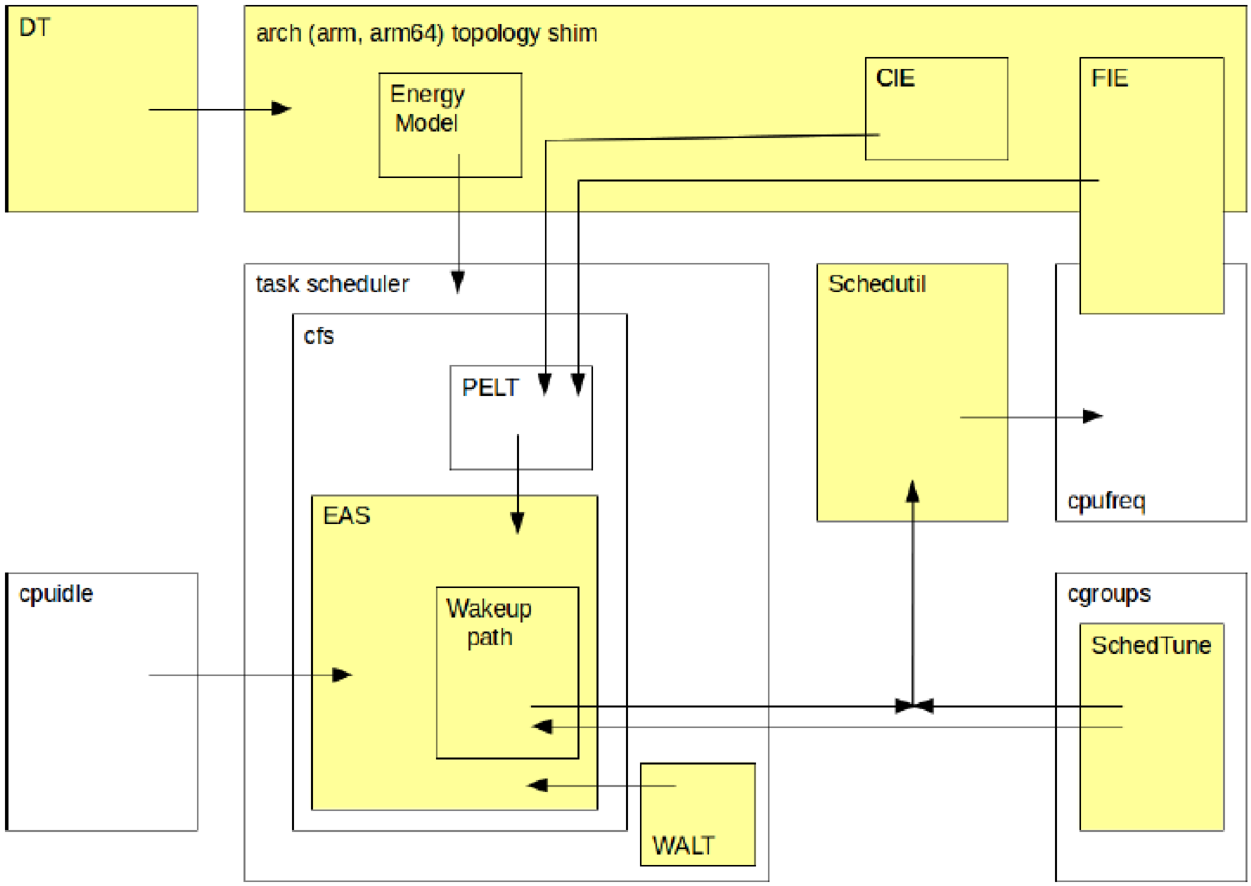


Figure 1 EAS building blocks in relation to Linux task scheduler, cgroups subsystem and related power management subsystems

完全公平调度（Completely Fair Scheduler CFS）实现了面向吞吐量的的任务调度策略，EAS为这个调度器添加了一个基于能耗的调度策略，在优化CPU算力冗余的同时实现了节能，EAS在系统中、低度负载情况下工作，CFS在系统满负载情况下工作。

EAS在CPU调度领域，在为任务选核是起作用，目的是保证性能的情况下尽可能节省功耗，EAS涉及内核的几个子系统（任务调度、能源管理、CPU动态调频），EAS代码主要位于 `kernel/sched/fair.c` ，能源感知的任务调度需要调度器评估各个任务在CPU上运行带来的能耗影响

EAS负载跟踪有两种模式，一种是“每实体负载跟踪（Per_Entity Load Track）”，通常用于负载跟踪，然后该信息用于确定频率以及如何在CPU上委派任务，另一种是“窗口辅助的负载跟踪（Window-Assisted Load Tracking）”，WALT更具有突发性，而PELT试图让频率保持连贯性，负载跟踪器实际上并不影响CPU频率，它只是告诉系统CPU使用率是多少

EAS全局控制开关 `/proc/sys/kernel/sched_energy_aware`

CPU算力归一化过程

当前，Linux无法凭自身算出CPU算力，因此必须要有把这个信息传递给Linux的方式，它是从 `capacity-dmips-mhz` CPU 设备树binding中衍生计算出来的

归一化CPU capacity， `topology_normalize_cpu_scale()` 定义在 `drivers/base/arch_topology()` ，这个capacity在schedutil调度中被 `sugov_get_util()` 函数读取

`topology_normalize_cpu_scale()` 在CPU初始化 `parse_dt_topology()` 中被调用，capacity归一化的前提条件是需要在设备树中CPU节点设置 `capacity-dmips-mhz` 属性，该属性表示不同CPU的计算能力，内核读取该属性设置CPU的 `raw_capacity` 为 `capacity-dmips-mhz` ，参考内核文档 `Documentation/devicetree/bindings/arm/cpu-capacity.txt`

> ARM推荐的测试CPU的性能工具：Dhrystone 2.1以上版本，可以通过单核跑分成绩作为 `capacity-dmips-mhz` 属性的参考，DMIPS： Dhrystone Million Instructions executed Per Second，表示了在Dhrystone这样一种测试方法下的MIPS，Dhrystone是一种整数运算测试程序。MIPS/MHz，就是说每MHz频率能产生多大的MIPS，CPU性能通常由每秒百万指令（Millions of Instructions Per Second，MIPS）表示，设备树里表示为dmips/mhz

CPU算力归一化公式，并不是简单的将capacity-dmips-mhz归一化到capacity，CPU的频率也参与到了计算中

`capacity = (own(capacity-dmips-mhz) * own(max_freq)) / (max(capacity-dmips-mhz) * max(max_freq)) * 1024`

根据测试部测试的E2000QCPU单核性能数据，E2000Q的 `capacity-dmips-mhz` 属性值可以设置为如下，放大1000倍：

dhrystone	Dhrystones/s	N/A	19920318	7530120
	DMIPS/MHz	N/A	5.66	2.85

```
// 小核
cpu_l0: cpu@0 {
        ...
        capacity-dmips-mhz = <2850>;
        ...
};
// 大核
cpu_b0: cpu@0 {
        ...
        capacity-dmips-mhz = <5660>;
        ...
};
```

实际经过CPU算力归一化到1024之后，对应的小核CPU算力为386，大核为1024

EAS代码相关结构体

perf_domain结构表示一个CPU性能域，perf_domain和cpufreq_policy是一一对应的，性能域之间形成链，链表头存放在root_domian中

```c
// kernel/sched/sched.h
// perf_comain 结构表示一个CPU性能域，perf_domain和cpufreq_policy是一一对应的
struct perf_domain {
	struct em_perf_domain *em_pd;
	struct perf_domain *next;
	struct rcu_head rcu;
};

struct root_domain {
	atomic_t		refcount;
	atomic_t		rto_count;
	struct rcu_head		rcu;
	cpumask_var_t		span;
	cpumask_var_t		online;

	// 该root_domain是否处于overload状态
	int			overload;

	// 该root_domain是否处于overutilized状态
	int			overutilized;

	/*
	 * The bit corresponding to a CPU gets set here if such CPU has more
	 * than one runnable -deadline task (as it is below for RT tasks).
	 */
	cpumask_var_t		dlo_mask;
	atomic_t		dlo_count;
	struct dl_bw		dl_bw;
	struct cpudl		cpudl;

#ifdef HAVE_RT_PUSH_IPI
	/*
	 * For IPI pull requests, loop across the rto_mask.
	 */
	struct irq_work		rto_push_work;
	raw_spinlock_t		rto_lock;
	/* These are only updated and read within rto_lock */
	int			rto_loop;
	int			rto_cpu;
	/* These atomics are updated outside of a lock */
	atomic_t		rto_loop_next;
	atomic_t		rto_loop_start;
#endif
	/*
	 * The "RT overload" flag: it gets set if a CPU has more than
	 * one runnable RT task.
	 */
	cpumask_var_t		rto_mask;
	struct cpupri		cpupri;

	// 系统中算力最大的CPU的算力
	unsigned long		max_cpu_capacity;

	/*
	 * NULL-terminated list of performance domains intersecting with the
	 * CPUs of the rd. Protected by RCU.
	 */
	// perf_domain单链表的表头
	struct perf_domain __rcu *pd;
```

```
};

// include/linux/energy_model.h
struct em_perf_state {
        unsigned long frequency;        // CPU频点，单位KHz
        unsigned long power;            // 此频点下的功耗
        unsigned long cost;                     // 此频点下的成本系数，等于 power * max_freq / freq
};

struct em_perf_domain {
        struct em_perf_state *table;    // CPU频点表
        int nr_perf_states;                             // 频点表中元素的个数
        unsigned long cpus[];                   // 此性能域中包括哪些CPU
};
```

E2000Q 5.10内核，perf_domain_debug 打印信息

```
[    2.574534] root_domain 0-3: pd3:{ cpus=3 nr_pstate=4 }
[    2.574540] freq: 250000, power: 79, cost: 632
[    2.579072] freq: 500000, power: 197, cost: 788
[    2.583690] freq: 1000000, power: 409, cost: 818
[    2.588390] freq: 2000000, power: 839, cost: 839
[    2.593094]  pd2:{ cpus=2 nr_pstate=4 }
[    2.593096] freq: 250000, power: 79, cost: 632
[    2.601445] freq: 500000, power: 197, cost: 788
[    2.606054] freq: 1000000, power: 409, cost: 818
[    2.610749] freq: 2000000, power: 839, cost: 839
[    2.615445]  pd0:{ cpus=0-1 nr_pstate=4 }
[    2.615447] freq: 187500, power: 1, cost: 8
[    2.623709] freq: 375000, power: 9, cost: 36
[    2.628058] freq: 750000, power: 55, cost: 110
[    2.632579] freq: 1500000, power: 125, cost: 125
```

root_domain的overload和overutilized说明：

- 对于一个 CPU 而言，其处于 overload 状态则说明其 rq 上有大于等于2个任务，或者虽然只有一个任务，但是是 misfit task
- 对于一个 CPU 而言，其处于 overutilized 状态说明该 cpu 的 utility 超过其 capacity（缺省预留20%的算力，另外，这里的 capacity 是用于cfs任务的算力）
- 对于 root domain，overload 表示至少有一个 cpu 处于 overload 状态。overutilized 表示至少有一个 cpu 处于 overutilized 状态
- overutilized 状态非常重要，它决定了调度器是否启用EAS，只有在系统没有 overutilized 的情况下EAS才会生效。overload和newidle balance的频次控制相关，当系统在overload的情况下，newidle balance才会启动进行均衡。

EAS能量计算方法

CPU在某个performance state(ps)下的计算能力：
ps->cap = ps->freq * scale_cpu / cpu_max_freq （1）

CPU在该频点performace state(ps)下的能量消耗：
cpu_nrg = ps->power * cpu_util / ps->cap （2）

结合(1) (2)可以得出CPU在该ps下的能量消耗
cpu_nrg = ps->power * cpu_max_freq * cpu_util / ps->freq * scale_cpu (3)

其中 ps->power * cpu_max_freq / ps->freq 是一个固定数据存放在频点表的cost成员中

一个pd内的CPU，拥有相同的cost，所以一个pd内所有CPU的能量消耗可以表示为
pd_nrg = ps->cost * sum(cpu_util) / scale_cpu

EAS的调度过程

在任务被重新唤醒或者fork新建时，会通过 `select_task_rq_fair()` 将任务进行balance，达到充分利用CPU的目的。在 `select_task_rq_fair()` ，若任务是被重新唤醒就会调用 `find_energy_efficient_cpu()` 进行选核执行

```c
/*
 * compute_energy(): Estimates the energy that @pd would consume if @p was
 * migrated to @dst_cpu. compute_energy() predicts what will be the utilization
 * landscape of @pd's CPUs after the task migration, and uses the Energy Model
 * to compute what would be the energy if we decided to actually migrate that
 * task.
 */
// 计算任务迁移到dst_cpu后，整个pd，即此cluster的energy
static long
compute_energy(struct task_struct *p, int dst_cpu, struct perf_domain *pd)
{
	struct cpumask *pd_mask = perf_domain_span(pd);
	// 获取该CPU的算力
	unsigned long cpu_cap = arch_scale_cpu_capacity(cpumask_first(pd_mask));
	unsigned long max_util = 0, sum_util = 0;
	int cpu;

	// 对此pd中每个online cpu都执行计算
	for_each_cpu_and(cpu, pd_mask, cpu_online_mask) {
		unsigned long cpu_util, util_cfs = cpu_util_next(cpu, p, dst_cpu);
		struct task_struct *tsk = cpu == dst_cpu ? p : NULL;

		// 返回该CPU下cfs+irq+rt+dl使用掉的CPU算力总和
		sum_util += schedutil_cpu_util(cpu, util_cfs, cpu_cap,
					       ENERGY_UTIL, NULL);

		cpu_util = schedutil_cpu_util(cpu, util_cfs, cpu_cap,
					      FREQUENCY_UTIL, tsk);
		max_util = max(max_util, cpu_util);
	}

	// 计算该pd下所有CPU的功耗和
	return em_cpu_energy(pd->em_pd, max_util, sum_util);
}

// 计算该pd下所有CPU的功耗和
static inline unsigned long em_cpu_energy(struct em_perf_domain *pd,
				unsigned long max_util, unsigned long sum_util)
{
	unsigned long freq, scale_cpu;
	struct em_perf_state *ps;
	int i, cpu;

	/*
	 * In order to predict the performance state, map the utilization of
	 * the most utilized CPU of the performance domain to a requested
	 * frequency, like schedutil.
	 */
	cpu = cpumask_first(to_cpumask(pd->cpus));
	scale_cpu = arch_scale_cpu_capacity(cpu);
	ps = &pd->table[pd->nr_perf_states - 1];
	freq = map_util_freq(max_util, ps->frequency, scale_cpu);

	/*
	 * Find the lowest performance state of the Energy Model above the
	 * requested frequency.
	 */
	for (i = 0; i < pd->nr_perf_states; i++) {
		ps = &pd->table[i];
```

```c
                if (ps->frequency >= freq)
                        break;
        }

        /*
         * The capacity of a CPU in the domain at the performance state (ps)
         * can be computed as:
         *
         *             ps->freq * scale_cpu
         *   ps->cap = --------------------                          (1)
         *                 cpu_max_freq
         *
         * So, ignoring the costs of idle states (which are not available in
         * the EM), the energy consumed by this CPU at that performance state
         * is estimated as:
         *
         *             ps->power * cpu_util
         *   cpu_nrg = --------------------                          (2)
         *                   ps->cap
         *
         * since 'cpu_util / ps->cap' represents its percentage of busy time.
         *
         *   NOTE: Although the result of this computation actually is in
         *         units of power, it can be manipulated as an energy value
         *         over a scheduling period, since it is assumed to be
         *         constant during that interval.
         *
         * By injecting (1) in (2), 'cpu_nrg' can be re-expressed as a product
         * of two terms:
         *
         *             ps->power * cpu_max_freq   cpu_util
         *   cpu_nrg = ------------------------ * ---------          (3)
         *                    ps->freq            scale_cpu
         *
         * The first term is static, and is stored in the em_perf_state struct
         * as 'ps->cost'.
         *
         * Since all CPUs of the domain have the same micro-architecture, they
         * share the same 'ps->cost', and the same CPU capacity. Hence, the
         * total energy of the domain (which is the simple sum of the energy of
         * all of its CPUs) can be factorized as:
         *
         *            ps->cost * \Sum cpu_util
         *   pd_nrg = ------------------------                       (4)
         *                  scale_cpu
         */
        return ps->cost * sum_util / scale_cpu;
}

// 寻找工作能耗最低的CPU
static int find_energy_efficient_cpu(struct task_struct *p, int prev_cpu)
{
        unsigned long prev_delta = ULONG_MAX, best_delta = ULONG_MAX;
        struct root_domain *rd = cpu_rq(smp_processor_id())->rd;
        unsigned long cpu_cap, util, base_energy = 0;
        int cpu, best_energy_cpu = prev_cpu;
        struct sched_domain *sd;
        struct perf_domain *pd;
```

```
rcu_read_lock();
// 从rd取pd的指针
pd = rcu_dereference(rd->pd);
if (!pd || READ_ONCE(rd->overutilized))
        goto fail;

/*
 * Energy-aware wake-up happens on the lowest sched_domain starting
 * from sd_asym_cpucapacity spanning over this_cpu and prev_cpu.
 */
sd = rcu_dereference(*this_cpu_ptr(&sd_asym_cpucapacity));
while (sd && !cpumask_test_cpu(prev_cpu, sched_domain_span(sd)))
        sd = sd->parent;
if (!sd)
        goto fail;

sync_entity_load_avg(&p->se);
if (!task_util_est(p))
        goto unlock;

// 遍历整个pd链表，计算p在不同pd下的能耗
for (; pd; pd = pd->next) {
        unsigned long cur_delta, spare_cap, max_spare_cap = 0;
        unsigned long base_energy_pd;
        int max_spare_cap_cpu = -1;

        /* Compute the 'base' energy of the pd, without @p */
        // 计算不包括p的情况下此pd当前的energy
        base_energy_pd = compute_energy(p, -1, pd);
        // 不包括p的情况下系统的总energy
        base_energy += base_energy_pd;

        // 遍历整个pd中的CPU，计算p放在该CPU上的功耗
        for_each_cpu_and(cpu, perf_domain_span(pd), sched_domain_span(sd)) {
                if (!cpumask_test_cpu(cpu, p->cpus_ptr))
                        continue;
                // 计算p放到此CPU后该CPU总共消耗的算力
                util = cpu_util_next(cpu, p, cpu);
                cpu_cap = capacity_of(cpu);
                spare_cap = cpu_cap;
                // 计算p放到此CPU后剩余的算力
                lsub_positive(&spare_cap, util);

                /*
                 * Skip CPUs that cannot satisfy the capacity request.
                 * IOW, placing the task there would make the CPU
                 * overutilized. Take uclamp into account to see how
                 * much capacity we can get out of the CPU; this is
                 * aligned with schedutil_cpu_util().
                 */
                util = uclamp_rq_util_with(cpu_rq(cpu), util, p);
                // CPU需要保留20%左右的算力，不满足需求后进行下一个CPU的探测
                if (!fits_capacity(util, cpu_cap))
                        continue;

                /* Always use prev_cpu as a candidate. */
                // 若对比的这个CPU就是任务之前运行的CPU
                if (cpu == prev_cpu) {
                        // 计算p放在该cpu后整个pd的能量消耗
```

```
                                prev_delta = compute_energy(p, prev_cpu, pd);
                                // 计算p放在该CPU后整个pd增加的能量消耗
                                prev_delta -= base_energy_pd;
                                // 更新best_delta，取最优能耗
                                best_delta = min(best_delta, prev_delta);
                        }

                        /*
                         * Find the CPU with the maximum spare capacity in
                         * the performance domain
                         */
                        // 记录p放上去后剩余算力最大的CPU和最大的剩余算力
                        if (spare_cap > max_spare_cap) {
                                max_spare_cap = spare_cap;
                                max_spare_cap_cpu = cpu;
                        }
                }

                /* Evaluate the energy impact of using this CPU. */
                // 同一个簇上的CPU取最大余量算力的那个CPU与其他簇的CPU做能量消耗对比
                if (max_spare_cap_cpu >= 0 && max_spare_cap_cpu != prev_cpu) {
                        // 计算p放在算力剩余最大的CPU后整个pd的能量消耗
                        cur_delta = compute_energy(p, max_spare_cap_cpu, pd);
                        // 计算能量消耗增量
                        cur_delta -= base_energy_pd;
                        // 如果当前能量增量优于p放在prev_cpu运行的能量消耗，则取该cpu运行p
                        if (cur_delta < best_delta) {
                                best_delta = cur_delta;
                                best_energy_cpu = max_spare_cap_cpu;
                        }
                }
        }
unlock:
        rcu_read_unlock();

        /*
         * Pick the best CPU if prev_cpu cannot be used, or if it saves at
         * least 6% of the energy used by prev_cpu.
         */
        // 若prev_cpu找不到，就直接返回最优能耗cpu
        if (prev_delta == ULONG_MAX)
                return best_energy_cpu;

        // 若最优能耗比放在prev_cpu上运行的能耗还要低6.25%以上，则取最优能耗cpu
        if ((prev_delta - best_delta) > ((prev_delta + base_energy) >> 4))
                return best_energy_cpu;

        // 否则不做改变，直接使用prev_cpu运行p
        return prev_cpu;

fail:
        rcu_read_unlock();

        return -1;
}
```