0、schedutil governor相关的结构体说明

```
struct sugov_policy {
    struct cpufreq_policy *policy;  /*cpu freq的policy*/

    struct sugov_tunables *tunables;  /*tunable结构体，根据用户需求改变*/
    struct list_head tunables_hook;/*tunable结构体元素链表*/

    raw_spinlock_t update_lock;  /* For shared policies */
    /*下面四个时间参数，第一个是上次频率变化的时间，后面三个是频率变化的颗粒度*/
    u64 last_freq_update_time;
    s64 min_rate_limit_ns;
    s64 up_rate_delay_ns;
    s64 down_rate_delay_ns;
    /*选择的next freq，cached freq是保存在cache的频率*/
    unsigned int next_freq;
    unsigned int cached_raw_freq;
    /*slack定时器，针对idle cpu的*/
    struct timer_list slack_timer;
    /* The next fields are only needed if fast switch cannot be used. */
    /*下面四个work相关最后调用的路径一样的*/
    struct irq_work irq_work;
    struct kthread_work work;
    struct mutex work_lock;
    struct kthread_worker worker;
    /*governor thread*/
    struct task_struct *thread;
    /*是否在频率调节过程中，频率调节完毕清标志位*/
    bool work_in_progress;
    /*频率限制改变会置这个标志位，并在频率update的时候，清这个标志位*/
      bool need_freq_update;
};
 /*每个cpu都存在一个这样的结构体，如果频率是shared的，则调节人一个cpu的频率会同时影响
   其他cpu，一般policy都是一样的。
*/
struct sugov_cpu {
    struct update_util_data update_util;
   /*每个cpu都是同一个sugov_policy，也是同一个cpufreq_policy*/
    struct sugov_policy *sg_policy;
    unsigned int cpu;  /*关联的cpu id*/
    /*是否处于iowait状态，iowait_boost频率及其boost最高频率*/
    bool iowait_boost_pending;
    unsigned int iowait_boost;
    unsigned int iowait_boost_max;
    u64 last_update;  /*cpu util，max最后update时间*/

    /* The fields below are only needed when sharing a policy. */
    unsigned long util;
    unsigned long max;
    unsigned int flags;

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

```
};
/*tunable使用，即用户空间可调的*/
struct sugov_tunables {
    struct gov_attr_set attr_set;/*sys接口属性*/
    unsigned int up_rate_limit_us;  /*频率上升的时间间隔限制*/
    unsigned int down_rate_limit_us;/*频率下降的时间间隔限制*/
    unsigned int timer_slack_val_us;  /*cpuidle期间，启动timer修改
      idlecpuidle的频率*/
    int freq_margin;  /*频率余量，可以修改，区分big/little core*/
};

```

一、schedutil governor如何调节cpu频率

```
static inline void cfs_rq_util_change(struct cfs_rq *cfs_rq)
{
    struct rq *rq = rq_of(cfs_rq);

    if (&rq->cfs == cfs_rq) {
        /*
         * There are a few boundary cases this might miss but it should
         * get called often enough that that should (hopefully) not be
         * a real problem -- added to that it only calls on the local
         * CPU, so if we enqueue remotely we'll miss an update, but
         * the next tick/schedule should update.
         *
         * It will not get called when we go idle, because the idle
         * thread is a different class (!fair), nor will the utilization
         * number include things like RT tasks.
         *
         * As is, the util number is not freq-invariant (we'd have to
         * implement arch_scale_freq_capacity() for that).
         *
         * See cpu_util().
         */
        cpufreq_update_util(rq, 0);
    }
}
```

继续：

```
#ifdef CONFIG_CPU_FREQ
DECLARE_PER_CPU(struct update_util_data *, cpufreq_update_util_data);

/**
 * cpufreq_update_util - Take a note about CPU utilization changes.
 * @rq: Runqueue to carry out the update for.
 * @flags: Update reason flags.
 *
 * This function is called by the scheduler on the CPU whose utilization is
 * being updated.
 *
 * It can only be called from RCU-sched read-side critical sections.
 *
 * The way cpufreq is currently arranged requires it to evaluate the CPU
 * performance state (frequency/voltage) on a regular basis to prevent it
from
```

```
 * being stuck in a completely inadequate performance level for too long.
 * That is not guaranteed to happen if the updates are only triggered from CFS,
 * though, because they may not be coming in if RT or deadline tasks are active
 * all the time (or there are RT and DL tasks only).
 *
 * As a workaround for that issue, this function is called by the RT and DL
 * sched classes to trigger extra cpufreq updates to prevent it from stalling,
 * but that really is a band-aid.  Going forward it should be replaced with
 * solutions targeted more specifically at RT and DL tasks.
 */
static inline void cpufreq_update_util(struct rq *rq, unsigned int flags)
{
        struct update_util_data *data;

    data = rcu_dereference_sched(*per_cpu_ptr(&cpufreq_update_util_data,
                          cpu_of(rq)));
    if (data)
        data->func(data, rq_clock(rq), flags);
}
#else
static inline void cpufreq_update_util(struct rq *rq, unsigned int flags) {}
#endif /* CONFIG_CPU_FREQ */
```

关键点是struct update_util_data这个结构体，仅仅是一个callback函数：

```
#ifdef CONFIG_CPU_FREQ
struct update_util_data {
    void (*func)(struct update_util_data *data, u64 time, unsigned int flags);
};

void cpufreq_add_update_util_hook(int cpu, struct update_util_data *data,
                       void (*func)(struct update_util_data *data, u64 time,
                                    unsigned int flags));
void cpufreq_remove_update_util_hook(int cpu);
#endif /* CONFIG_CPU_FREQ */
```

接下来看下这个结构体与函数cpufreq_add_update_util_hook的关系是什么：

```
DEFINE_PER_CPU(struct update_util_data *, cpufreq_update_util_data);

/**
 * cpufreq_add_update_util_hook - Populate the CPU's update_util_data pointer.
 * @cpu: The CPU to set the pointer for.
 * @data: New pointer value.
 * @func: Callback function to set for the CPU.
 *
 * Set and publish the update_util_data pointer for the given CPU.
 *
 * The update_util_data pointer of @cpu is set to @data and the callback
 * function pointer in the target struct update_util_data is set to @func.
 * That function will be called by cpufreq_update_util() from RCU-sched
 * read-side critical sections, so it must not sleep.  @data will always be
```

```
 * passed to it as the first argument which allows the function to get to
the
 * target update_util_data structure and its container.
 *
 * The update_util_data pointer of @cpu must be NULL when this function is
 * called or it will WARN() and return with no effect.
 */
void cpufreq_add_update_util_hook(int cpu, struct update_util_data *data,
            void (*func)(struct update_util_data *data, u64 time,
                     unsigned int flags))
{
    if (WARN_ON(!data || !func))
        return;

    if (WARN_ON(per_cpu(cpufreq_update_util_data, cpu)))
        return;

    data->func = func;
    rcu_assign_pointer(per_cpu(cpufreq_update_util_data, cpu), data);
}
```

可以看到结构体update_util_data的callback函数指向了函数cpufreq_add_update_util_hook钩子函数的形参：

```
void (*func)(struct update_util_data *data, u64 time,
                     unsigned int flags)
```

那么这个函数在哪里赋值呢？

我们看到在kernel/sched/cpufreq_schedutil.c文件，就是最新的cpu调节频率的governor，不在是原先的interactive或者ondemand governor了。

作为频率调节的governor编写流程与其他governor类型，先注册名字为schedutil governor：

```
#ifndef CONFIG_CPU_FREQ_DEFAULT_GOV_SCHEDUTIL
static
#endif
struct cpufreq_governor cpufreq_gov_schedutil = {
    .name = "schedutil",
    .governor = cpufreq_schedutil_cb,
    .owner = THIS_MODULE,
};

static int __init sugov_register(void)
{
    return cpufreq_register_governor(&cpufreq_gov_schedutil);
}
fs_initcall(sugov_register);
```

之后之后，governor开始走governor的callback函数cpufreq_schedutil_cb，

```
static int cpufreq_schedutil_cb(struct cpufreq_policy *policy,
                unsigned int event)
{
    switch(event) {
    case CPUFREQ_GOV_POLICY_INIT:
        return sugov_init(policy);
    case CPUFREQ_GOV_POLICY_EXIT:
        return sugov_exit(policy);
    case CPUFREQ_GOV_START:
        return sugov_start(policy);
```

```
        case CPUFREQ_GOV_STOP:
            return sugov_stop(policy);
        case CPUFREQ_GOV_LIMITS:
            return sugov_limits(policy);
        default:
            BUG();
        }
}
```

开始执行init，然后执行start，根据event类型来执行。系统刚刚起来执行init和start动作，init是一些参数的初始化，而start才是真正的governor开启work了。

```
static int sugov_start(struct cpufreq_policy *policy)
{
    struct sugov_policy *sg_policy = policy->governor_data;
    unsigned int cpu;

    sg_policy->up_rate_delay_ns =
        sg_policy->tunables->up_rate_limit_us * NSEC_PER_USEC;
    sg_policy->down_rate_delay_ns =
        sg_policy->tunables->down_rate_limit_us * NSEC_PER_USEC;
    update_min_rate_limit_us(sg_policy);
    sg_policy->last_freq_update_time = 0;
    sg_policy->next_freq = UINT_MAX;
    sg_policy->work_in_progress = false;
    sg_policy->need_freq_update = false;
    sg_policy->cached_raw_freq = UINT_MAX;

    for_each_cpu(cpu, policy->cpus) {
        struct sugov_cpu *sg_cpu = &per_cpu(sugov_cpu, cpu);

        memset(sg_cpu, 0, sizeof(*sg_cpu));
        sg_cpu->cpu = cpu;
        sg_cpu->sg_policy = sg_policy;
        sg_cpu->flags = SCHED_CPUFREQ_DL;
        sg_cpu->iowait_boost_max = policy->cpuinfo.max_freq;
                /*OK，真正的struct update_util_data的元素的callback函数现真身了。
*/
        cpufreq_add_update_util_hook(cpu, &sg_cpu->update_util,
                         policy_is_shared(policy) ?
                            sugov_update_shared :
                            sugov_update_single);
    }
    return 0;
}
/*这个函数肯定返回true*/
static inline bool policy_is_shared(struct cpufreq_policy *policy)
{
    return cpumask_weight(policy->cpus) > 1;
}
```

二、sugov_upodate_shared函数怎么计算得到next_freq

可以看到这个函数的实现code如下：

```
static void sugov_update_shared(struct update_util_data *hook, u64 time,
                unsigned int flags)
```

```
{
    struct sugov_cpu *sg_cpu = container_of(hook, struct sugov_cpu, update_util);
    struct sugov_policy *sg_policy = sg_cpu->sg_policy;
    unsigned long util, max;
    unsigned int next_f;

    sugov_get_util(&util, &max, time, sg_cpu->cpu);

    raw_spin_lock(&sg_policy->update_lock);

    sg_cpu->util = util;
    sg_cpu->max = max;
    sg_cpu->flags = flags;

    sugov_set_iowait_boost(sg_cpu, time, flags);
    sg_cpu->last_update = time;

    if (sugov_should_update_freq(sg_policy, time)) {
        if (flags & SCHED_CPUFREQ_DL)
            next_f = sg_policy->policy->cpuinfo.max_freq;
        else
            next_f = sugov_next_freq_shared(sg_cpu, time);

        sugov_update_commit(sg_policy, time, next_f);
    }

    raw_spin_unlock(&sg_policy->update_lock);
}
```

分别来讲解各个重要的函数

2.1 sugov_get_util(&util, &max, time, sg_cpu->cpu)怎么获取util/max的数值的。

函数实现如下：

```
static void sugov_get_util(unsigned long *util, unsigned long *max, u64 time, int cpu)
{
    struct rq *rq = cpu_rq(cpu);
    unsigned long max_cap, rt;
    s64 delta;
    /*不同cluster max_cap不同，我们平台上，cluster0:782，cluster1:1024*/
    max_cap = arch_scale_cpu_capacity(NULL, cpu);

    sched_avg_update(rq);
    delta = time - rq->age_stamp;
    if (unlikely(delta < 0))
        delta = 0;
    rt = div64_u64(rq->rt_avg, sched_avg_period() + delta);
    rt = (rt * max_cap) >> SCHED_CAPACITY_SHIFT;

    *util = boosted_cpu_util(cpu);
    if (likely(use_pelt()))
        *util = *util + rt;

    *util = min(*util, max_cap);
    *max = max_cap;
```

```
}
```

它里面涉及的函数如下：

- sched_avg_update(rq)，是一个update sched avg负载使用的：

```
const_debug unsigned int sysctl_sched_time_avg = MSEC_PER_SEC;
static inline u64 sched_avg_period(void)
{
    return (u64)sysctl_sched_time_avg * NSEC_PER_MSEC / 2;
}
void sched_avg_update(struct rq *rq)
{       /*500ms一次update sched avg*/
    s64 period = sched_avg_period();
        /*age_stamp是当前cpu rq的启动时间，有两个目的：
        * 1. 衰减rt负载，即每个period，衰减一半，也叫老化周期
        * 2. 将age_stamp的启动窗口累加到接近rq_clock的窗口，目的是每次仅仅计算
        * 本period内的load
         */
    while ((s64)(rq_clock(rq) - rq->age_stamp) > period) {
        /*
         * Inline assembly required to prevent the compiler
         * optimising this loop into a divmod call.
         * See __iter_div_u64_rem() for another example of this.
         */
        asm("" : "+rm" (rq->age_stamp));
        rq->age_stamp += period;
        rq->rt_avg /= 2;
    }
}
```

下面这段代码的意思是，计算一个周期内的rt负载并归一化为capacity数值：

```
delta = time - rq->age_stamp;
if (unlikely(delta < 0))
    delta = 0;
rt = div64_u64(rq->rt_avg, sched_avg_period() + delta);
rt = (rt * max_cap) >> SCHED_CAPACITY_SHIFT;
```

- boosted_cpu_util(cpu)怎么得到util的，对于函数schedtune_cpu_margin的实现以后在仔细check，本文不讲解。

```
unsigned long
boosted_cpu_util(int cpu)
{
    unsigned long util = cpu_util_freq(cpu);
    /*仔细check怎么计算的*/
    long margin = schedtune_cpu_margin(util, cpu);

    trace_sched_boost_cpu(cpu, util, margin);

    return util + margin;
}

static inline unsigned long cpu_util_freq(int cpu)
{
    unsigned long util = cpu_rq(cpu)->cfs.avg.util_avg;
/*各个cluster的max_capacity*/
    unsigned long capacity = capacity_orig_of(cpu);
/*按照walt 在各个窗口累加的runnable time/walt_ravg_window归一化
```

```
 *load作为cpu的util数值
* util范围在0~capacity之间。 util从walt获取。
*/
#ifdef CONFIG_SCHED_WALT
    if (!walt_disabled && sysctl_sched_use_walt_cpu_util)
        util = div64_u64(cpu_rq(cpu)->cumulative_runnable_avg,
                 walt_ravg_window >> SCHED_LOAD_SHIFT);
#endif
    return (util >= capacity) ? capacity : util;
}
```

- 最后得到util和max数值。由于使用WALT来计算cpu util，所以util = util(普通进程) + rt(实时进程)。最后util = min(util,max_cap),max=max_cap；计算完毕。max就是各个cluster的每个core的capacity，是一个固定数值，可能在thermal起作用的情况下会变小，这个需要仔细check下。

2.2 sugov_set_iowait_boost(sg_cpu, time, flags)怎么设置iowait_boost数值。

- 继续执行sugov_update_shared函数，更新sugov_cpu结构体元素；
- 根据flags数值：如果flags为2，则是iowait boost情况，并且有一个iowait_boost_pending标志位判断当前是否已经是iowait状态。如果已经是则直接return，否则根据iowait_boost是否有数值来设定iowait_boost的频率数值。
- 如果flags为其他数值，并且iowait_boost存在数值，如果计算load的间隔超过一个tickless时间，则判断是idle状态，将iowait_boost和pending标志位清零。等待下次计算周期在查看iowait状态。
- flags为0，是没有iowait的普通进程。

```
#define SCHED_CPUFREQ_RT        (1U << 0)  /*sched_class rt*/
#define SCHED_CPUFREQ_DL        (1U << 1)  /*sched_class */
#define SCHED_CPUFREQ_IOWAIT    (1U << 2)  /*sched_class fair &&
task->in_iowait!=0*/
```

2.3 sugov_should_update_freq(sg_policy, time)是否需要进行频率update，判定若干个标志位

- dvfs_possible_from_any_cpu，即每个cpu可以单独调节电压并传递给其他cpu一起调节，默认为true
- fast_switch_enabled，快速频率切换是否enable，默认false
- work_in_progress：是否正在调节频率，调节频率之前置为true，调节频率之后置为false，默认false
- need_freq_update，默认false，只有在governor limit阶段置为true。
- 最后判定rq_clock-last_freq_update_time的数值与min_rate_limit_ns比较得出是否需要update frequency。也就是频率调节的最小间隔，小于此间隔不予调节。我们系统是0.5ms

```
static void update_min_rate_limit_us(struct sugov_policy *sg_policy)
{
    mutex_lock(&min_rate_lock);
        /*min(500,1000),unit:us,也就是
           min(up_rate_limit_us,down_rate_limit_us)*/
    sg_policy->min_rate_limit_ns = min(sg_policy->up_rate_delay_ns,
                   sg_policy->down_rate_delay_ns);
    mutex_unlock(&min_rate_lock);
}
```

如果2.3函数返回true，则执行2.4/2.5，否则直接返回，不做频率调整。

2.4 flags不同，如何选择next_f，即下一个cpu frequency

- flags==SCHED_CPUFREQ_DL，next_f = cpuinfo.max_freq
- 其他flags走下面的，对所有cpu，根据sugov_cpu的util，max，iowait_boost，iowait_boost_max数值选择所有cpu里面的max*util最大的一对。每个cpu都有一个util，max，iowait_boost,iowait_boost_max=cpuinfo.max_freq，具体怎么计算的看下code一目了然。比较简单。在函数sugov_next_freq_shared里面实现的。

2.4.1 在函数sugov_next_freq_shared里面会遍历所有的cpu，遍历规则如下：

- 在sugov_update_shared函数一开始，我们就获取了当前cpu的util和max；
- 每次遍历一个cpu，比较(j_util *max > j_max *util),则util=j_util,max=j_max，目的挑选最大的。max一般都是固定数值，还是选择cpu最大的util作为调节频率的依据，有点像ondemand governor，采集cpuloading，也是选择比较各个cpuloading最大的作为调节频率的依据。
- 这是cpu 的util和max的选择，还需要根据iowait_boost和iowait_boost_max来确认最终选择的util和max的数值。iowait boost与正常的util是两个独立的分支，需要互相参考挑选最大数值作为最后的调节频率的依据。

3.4.2 最后会根据util,max选择next_f，具体实现在get_next_freq(sg_policy, util, max)

```
static unsigned int get_next_freq(struct sugov_policy *sg_policy,
                                  unsigned long util, unsigned long max)
{
    struct cpufreq_policy *policy = sg_policy->policy;
    /*freq为max_freq*/
    unsigned int freq = arch_scale_freq_invariant() ?
                  policy->cpuinfo.max_freq : policy->cur;
    /*freq_margin是一开始就设定好的，区分big/little core,根据min_cap_cpu_mask*/
    int freq_margin = sg_policy->tunables->freq_margin;
    /*对最小cluster的util进行调整，变大util数值，capa_margin=1138*/
    if (cpumask_test_cpu(policy->cpu, &min_cap_cpu_mask))
        util = util * capacity_margin / SCHED_CAPACITY_SCALE;
    /*根据设定的margin来决定next freq*/
    if (freq_margin > -100 && freq_margin < 100) {
        freq_margin = (freq * freq_margin) / 100;
        freq = ((int)freq + freq_margin) * util / max;
    } else
        freq = (freq + (freq >> 2)) * util / max;  /*1.25 freq*/

    if (freq == sg_policy->cached_raw_freq && sg_policy->next_freq !=
UINT_MAX)
        return sg_policy->next_freq;
    sg_policy->cached_raw_freq = freq;
    return cpufreq_driver_resolve_freq(policy, freq);  /*选择target_freq*/
}
```

cached_raw_freq是保存的上次频率值，如果一致的话就直接调整，不用再次选择target_freq

3.5 sugov_update_commit(sg_policy, time, next_f)触发变频需求

- sugov_up_down_rate_limit这个函数用来作为频率调整的判断依据，比如是否符合升频的时间限制，降频的时间限制。
- 根据选择的next freq数值来修订slack_timer是否执行
- 如果选择的next freq==sg_policy->next_freq频率不做调整
- 更新sg_policy->next_freq=next_freq，sg_policy->last_freq_update_time=time

- 最后设置work_in_process标志位为true，同时执行worker里面函数，执行sugov_irq_work---->sugov_work---> __cpufreq_driver_target(sg_policy->policy, sg_policy->next_freq,CPUFREQ_RELATION_L);基本上频率调节结束了。

二、kernel在什么时候触发governor去做频率的调整

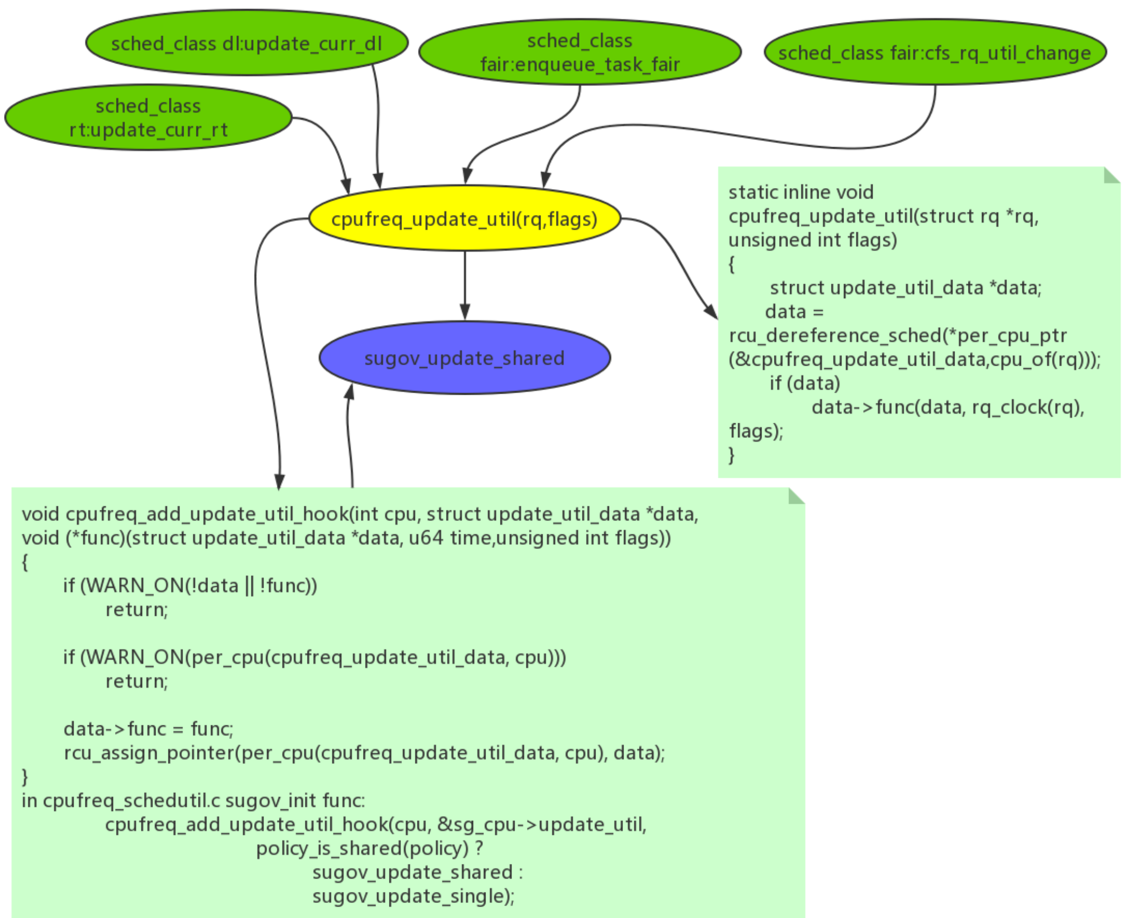
以前我们知道interactive/ondemand governor都自带timer去主动收集cpu loading来做决策是否需要频率的调整，但是从schedutil governor看，并没有看到什么时候主动去计算负载，然后做频率的调整。

从第一章，看到，集中点都在这个函数上：cpufreq_update_util，下面是系统调用的地方

```
• kernel/sched/fair.c:3163:       cpufreq_update_util(rq, 0);
• kernel/sched/fair.c:4847:       cpufreq_update_util(rq,
  SCHED_CPUFREQ_IOWAIT);
• kernel/sched/rt.c:1007: cpufreq_update_util(rq, SCHED_CPUFREQ_RT);
• kernel/sched/deadline.c:759:    cpufreq_update_util(rq, SCHED_CPUFREQ_DL);
```

可以看到flags参数分类三类sched_class，RT(flags=1)，DL(flags=2)，FAIR(iowait(flags=4) or not iowait(flags=0))

目的是在什么实际调用cpufreq_update_util函数：



对于sched class怎么去调用，从何处去调用，后面在研究。所以遗留的问题如下：

1. cpu margin怎么计算的-------done
2. cpufreq_update_util从何处被调用的，这个涉及到进程调度思想，调度算法后续在啃。长期目标