由于在task在创建或者task从idle会wakeup的流程中涉及到sched domain和sched group的知识,所以现在提前看下这部分.
根据实际的物理属性,CPU domain分成下面三种:

CPU 分类	Linux内核分类	描述
SMT(多线程)	CONFIG_SHCED_SMT	一个物理核心可以有两个以及之上的执行线程,一个物理核心里面的线程共享相同的CPU资源和L1 cache,task的迁移不会影响cache利用率
MC(多核)	CONFIG_SHCED_MC	每个物理核心有独立的L1 cache,多个物理核心可以组成一个cluster,cluster共享L2 cache
DIE(SoC)	简称为DIE	SoC级别

一 cpu topology的获取

arm64架构的cpu拓扑结构存储在cpu_topology[NR_CPUS]变量当中：

```
struct cpu_topology {
    int thread_id;
    int core_id;
    int cluster_id;
    cpumask_t thread_sibling;
    cpumask_t core_sibling;
};

extern struct cpu_topology cpu_topology[NR_CPUS];
#define topology_physical_package_id(cpu)    (cpu_topology[cpu].cluster_id)
#define topology_core_id(cpu)        (cpu_topology[cpu].core_id)
#define topology_core_cpumask(cpu)   (&cpu_topology[cpu].core_sibling)
#define topology_sibling_cpumask(cpu)    (&cpu_topology[cpu].thread_sibling)
```

cpu_topology[]数组获取的路径如下(都是从dts里面获取的):

```
kernel_init() -> kernel_init_freeable() -> smp_prepare_cpus() ->
init_cpu_topology() -> parse_dt_topology()
--->
static int __init parse_dt_topology(void)
{
    struct device_node *cn, *map;
    int ret = 0;
    int cpu;
    /*从dts里面获取cpu node的跟节点*/
    cn = of_find_node_by_path("/cpus");
    if (!cn) {
        pr_err("No CPU information found in DT\n");
        return 0;
    }

    /*
     * When topology is provided cpu-map is essentially a root
     * cluster with restricted subnodes.
```

```
         */
        map = of_get_child_by_name(cn, "cpu-map");
        if (!map)
            goto out;

        ret = parse_cluster(map, 0);
        if (ret != 0)
            goto out_map;

        /*
         * Check that all cores are in the topology; the SMP code will
         * only mark cores described in the DT as possible.
         */
        for_each_possible_cpu(cpu)
            if (cpu_topology[cpu].cluster_id == -1)
                ret = -EINVAL;

    out_map:
        of_node_put(map);
    out:
        of_node_put(cn);
        return ret;
    }
    /*看下解析cluster怎么做的*/
    static int __init parse_cluster(struct device_node *cluster, int depth)
    {
        char name[10];
        bool leaf = true;
        bool has_cores = false;
        struct device_node *c;
        static int cluster_id __initdata;
        int core_id = 0;
        int i, ret;

        /*
         * First check for child clusters; we currently ignore any
         * information about the nesting of clusters and present the
         * scheduler with a flat list of them.
         */
        i = 0;
        do {  /*多个cluster,迭代遍历*/
            snprintf(name, sizeof(name), "cluster%d", i);
            c = of_get_child_by_name(cluster, name);
            if (c) {
                leaf = false;
                ret = parse_cluster(c, depth + 1);
                of_node_put(c);
                if (ret != 0)
                    return ret;
            }
            i++;
        } while (c);

        /* Now check for cores */
        i = 0;
        do {  /*遍历cluster下面core的节点*/
            snprintf(name, sizeof(name), "core%d", i);
            c = of_get_child_by_name(cluster, name);
            if (c) {
                has_cores = true;

```

```c
                if (depth == 0) {
                    pr_err("%s: cpu-map children should be clusters\n",
                           c->full_name);
                    of_node_put(c);
                    return -EINVAL;
                }
                /*如果是叶子cluster节点，继续遍历core中的cpu节点*/
                if (leaf) {
                    ret = parse_core(c, cluster_id, core_id++);
                } else {
                    pr_err("%s: Non-leaf cluster with core %s\n",
                           cluster->full_name, name);
                    ret = -EINVAL;
                }

                of_node_put(c);
                if (ret != 0)
                    return ret;
            }
            i++;
        } while (c);

        if (leaf && !has_cores)
            pr_warn("%s: empty cluster\n", cluster->full_name);

        if (leaf)
            cluster_id++;

        return 0;
}
static int __init parse_core(struct device_node *core, int cluster_id,
                 int core_id)
{
        char name[10];
        bool leaf = true;
        int i = 0;
        int cpu;
        struct device_node *t;

        do {   /*如果存在thread层级，解析thread和cpu层级*/
            snprintf(name, sizeof(name), "thread%d", i);
            t = of_get_child_by_name(core, name);
            if (t) {
                leaf = false;
                cpu = get_cpu_for_node(t);
                if (cpu >= 0) {
                    cpu_topology[cpu].cluster_id = cluster_id;
                    cpu_topology[cpu].core_id = core_id;
                    cpu_topology[cpu].thread_id = i;
                } else {
                    pr_err("%s: Can't get CPU for thread\n",
                           t->full_name);
                    of_node_put(t);
                    return -EINVAL;
                }
                of_node_put(t);
            }
            i++;
        } while (t);
        /*否则直接解析cpu层级*/
        cpu = get_cpu_for_node(core);
        if (cpu >= 0) {
            if (!leaf) {
                pr_err("%s: Core has both threads and CPU\n",
                       core->full_name);
                return -EINVAL;
```

```
        }
        /*得到了cpu的cluster_id/core_id*/
        cpu_topology[cpu].cluster_id = cluster_id;
        cpu_topology[cpu].core_id = core_id;
    } else if (leaf) {
        pr_err("%s: Can't get CPU for leaf core\n", core->full_name);
        return -EINVAL;
    }

    return 0;
}
static int __init get_cpu_for_node(struct device_node *node)
{
    struct device_node *cpu_node;
    int cpu;
    /*解析cpu层级*/
    cpu_node = of_parse_phandle(node, "cpu", 0);
    if (!cpu_node)
        return -1;

    for_each_possible_cpu(cpu) {
        if (of_get_cpu_node(cpu, NULL) == cpu_node) {
            of_node_put(cpu_node);
            return cpu;
        }
    }

    pr_crit("Unable to find CPU node for %s\n", cpu_node->full_name);

    of_node_put(cpu_node);
    return -1;
}

```

cpu同一层次的关系cpu_topology[cpu].core_sibling/thread_sibling会在update_siblings_masks()中更新

```
kernel_init() -> kernel_init_freeable() -> smp_prepare_cpus()
-> store_cpu_topology() -> update_siblings_masks()
--->
static void update_siblings_masks(unsigned int cpuid)
{
    struct cpu_topology *cpu_topo, *cpuid_topo =
&cpu_topology[cpuid];
    int cpu;

    /* update core and thread sibling masks */
    for_each_possible_cpu(cpu) {
        cpu_topo = &cpu_topology[cpu];

        if (cpuid_topo->cluster_id != cpu_topo->cluster_id)
            continue;

        cpumask_set_cpu(cpuid, &cpu_topo->core_sibling);
        if (cpu != cpuid)
            cpumask_set_cpu(cpu, &cpuid_topo->core_sibling);

        if (cpuid_topo->core_id != cpu_topo->core_id)
            continue;

```

```
            cpumask_set_cpu(cpuid, &cpu_topo->thread_sibling);
            if (cpu != cpuid)
                cpumask_set_cpu(cpu,
&cpuid_topo->thread_sibling);
        }
}
```

以我现在的手机平台为例,cpu相关的dts如下:

```
cpus {
    #address-cells = <2>;
    #size-cells = <0>;

    cpu-map {
        cluster0 {
            core0 {
                cpu = <&CPU0>;
            };
            core1 {
                cpu = <&CPU1>;
            };
            core2 {
                cpu = <&CPU2>;
            };
            core3 {
                cpu = <&CPU3>;
            };
        };
        cluster1 {
            core0 {
                cpu = <&CPU4>;
            };
            core1 {
                cpu = <&CPU5>;
            };
            core2 {
                cpu = <&CPU6>;
            };
            core3 {
                cpu = <&CPU7>;
            };
        };
    };

    CPU0: cpu@0 {
        device_type = "cpu";
        compatible = "arm,cortex-a55","arm,armv8";
        reg = <0x0 0x0>;
        enable-method = "psci";
        cpu-supply = <&fan53555_dcdc>;
        cpufreq-data-v1 = <&cpufreq_clus0>;
        cpu-idle-states = <&CORE_PD>;
        sched-energy-costs = <&CPU_COST_0 &CLUSTER_COST_0>;
    };
    CPU1: cpu@100 {
        device_type = "cpu";
```

```
            compatible = "arm,cortex-a55","arm,armv8";
            reg = <0x0 0x100>;
            enable-method = "psci";
            cpu-supply = <&fan53555_dcdc>;
            cpufreq-data-v1 = <&cpufreq_clus0>;
            cpu-idle-states = <&CORE_PD>;
            sched-energy-costs = <&CPU_COST_0 &CLUSTER_COST_0>;
        };
        CPU2: cpu@200 {
            device_type = "cpu";
            compatible = "arm,cortex-a55","arm,armv8";
            reg = <0x0 0x200>;
            enable-method = "psci";
            cpu-supply = <&fan53555_dcdc>;
            cpufreq-data-v1 = <&cpufreq_clus0>;
            cpu-idle-states = <&CORE_PD>;
            sched-energy-costs = <&CPU_COST_0 &CLUSTER_COST_0>;
        };
        CPU3: cpu@300 {
            device_type = "cpu";
            compatible = "arm,cortex-a55","arm,armv8";
            reg = <0x0 0x300>;
            enable-method = "psci";
            cpu-supply = <&fan53555_dcdc>;
            cpufreq-data-v1 = <&cpufreq_clus0>;
            cpu-idle-states = <&CORE_PD>;
            sched-energy-costs = <&CPU_COST_0 &CLUSTER_COST_0>;
        };
        CPU4: cpu@400 {
            device_type = "cpu";
            compatible = "arm,cortex-a55","arm,armv8";
            reg = <0x0 0x400>;
            enable-method = "psci";
            cpu-supply = <&vddcpu>;
            cpufreq-data-v1 = <&cpufreq_clus1>;
            cpu-idle-states = <&CORE_PD>;
            sched-energy-costs = <&CPU_COST_1 &CLUSTER_COST_1>;
        };
        CPU5: cpu@500 {
            device_type = "cpu";
            compatible = "arm,cortex-a55","arm,armv8";
            reg = <0x0 0x500>;
            enable-method = "psci";
            cpu-supply = <&vddcpu>;
            cpufreq-data-v1 = <&cpufreq_clus1>;
            cpu-idle-states = <&CORE_PD>;
            sched-energy-costs = <&CPU_COST_1 &CLUSTER_COST_1>;
        };
        CPU6: cpu@600 {
            device_type = "cpu";
            compatible = "arm,cortex-a55","arm,armv8";
            reg = <0x0 0x600>;
            enable-method = "psci";
            cpu-supply = <&vddcpu>;
```

```
            cpufreq-data-v1 = <&cpufreq_clus1>;
            cpu-idle-states = <&CORE_PD>;
            sched-energy-costs = <&CPU_COST_1 &CLUSTER_COST_1>;
        };
        CPU7: cpu@700 {
            device_type = "cpu";
            compatible = "arm,cortex-a55","arm,armv8";
            reg = <0x0 0x700>;
            enable-method = "psci";
            cpu-supply = <&vddcpu>;
            cpufreq-data-v1 = <&cpufreq_clus1>;
            cpu-idle-states = <&CORE_PD>;
            sched-energy-costs = <&CPU_COST_1 &CLUSTER_COST_1>;
        };
    };
```

经过parse_dt_topology()、update_siblings_masks()解析后得到cpu_topology[]的值为：

```
cpu 0 cluster_id = 0, core_id = 0, core_sibling = 0xf
cpu 1 cluster_id = 0, core_id = 1, core_sibling = 0xf
cpu 2 cluster_id = 0, core_id = 2, core_sibling = 0xf
cpu 3 cluster_id = 0, core_id = 3, core_sibling = 0xf
cpu 4 cluster_id = 1, core_id = 0, core_sibling = 0xf0
cpu 5 cluster_id = 1, core_id = 1, core_sibling = 0xf0
cpu 6 cluster_id = 1, core_id = 2, core_sibling = 0xf0
cpu 7 cluster_id = 1, core_id = 3, core_sibling = 0xf0
//adb 查看cpu topology信息
meizu1000:/sys/devices/system/cpu/cpu0/topology # ls -l
total 0
-r--r--r-- 1 root root 4096 2018-09-05 16:10 core_id   //0
-r--r--r-- 1 root root 4096 2018-09-05 16:10 core_siblings  //0f
-r--r--r-- 1 root root 4096 2018-09-05 16:10 core_siblings_list//0-3
-r--r--r-- 1 root root 4096 2018-09-05 16:10 physical_package_id  //0
-r--r--r-- 1 root root 4096 2018-09-05 16:10 thread_siblings //01
-r--r--r-- 1 root root 4096 2018-09-05 16:10 thread_siblings_list//0

```

二 调度域,调度组以及调度组capacity的初始化

内核中有预先定义的调度域的结构体struct sched_domain_topology_level:

```
typedef const struct cpumask *(*sched_domain_mask_f)(int cpu);
typedef int (*sched_domain_flags_f)(void);
typedef
const struct sched_group_energy * const(*sched_domain_energy_f)(int cpu);

struct sched_domain_topology_level {
    sched_domain_mask_f mask;
    sched_domain_flags_f sd_flags;
    sched_domain_energy_f energy;
    int             flags;
    int             numa_level;
    struct sd_data      data;
#ifdef CONFIG_SCHED_DEBUG
    char                *name;
#endif
};
```

```c

struct sd_data {
    struct sched_domain **__percpu sd;
    struct sched_group **__percpu sg;
    struct sched_group_capacity **__percpu sgc;
};

/*
 * Topology list, bottom-up.
 */
static struct sched_domain_topology_level default_topology[] = {
#ifdef CONFIG_SCHED_SMT
    { cpu_smt_mask, cpu_smt_flags, SD_INIT_NAME(SMT) },
#endif
#ifdef CONFIG_SCHED_MC
    { cpu_coregroup_mask, cpu_core_flags, SD_INIT_NAME(MC) },
#endif
    { cpu_cpu_mask, SD_INIT_NAME(DIE) },
    { NULL, },
};

static struct sched_domain_topology_level *sched_domain_topology =
    default_topology;
```

我们根据default_topology[]这个结构体数组来解析sched_domain_topology_level(简称SDTL)结构体成员变量的含义

- sched_domain_mask_f mask:函数指针,用于指定某个SDTL层级的cpumask位图
- sched_domain_flags_f sd_flags:函数指针,用于指定某个SDTL层级的标志位
- sched_domain_energy_f energy:函数指针,用于指定某个SDTL层级的energy信息,包括在active和idle状态下的capacity和power信息,具体获取是从dts(kernel/sched/energy.c)获取的([energy是怎么获取的](#))
- struct sd_data:表示SDTL层级关系,包括domain/group/group capacity

从default_topology结构体数组看,DIE是标配,MC和SMT是可选择的.目前ARM不支持SMT超线程技术,目的只有两种topology,MC和DIE.

由于在建立topology的时候,涉及到cpumask的使用,下面几个是比较常用的cpumask:

```c
#ifdef CONFIG_INIT_ALL_POSSIBLE
static DECLARE_BITMAP(cpu_possible_bits, CONFIG_NR_CPUS) __read_mostly
    = CPU_BITS_ALL;
#else
static DECLARE_BITMAP(cpu_possible_bits, CONFIG_NR_CPUS) __read_mostly;
#endif
const struct cpumask *const cpu_possible_mask =
to_cpumask(cpu_possible_bits);
EXPORT_SYMBOL(cpu_possible_mask);

static DECLARE_BITMAP(cpu_online_bits, CONFIG_NR_CPUS) __read_mostly;
const struct cpumask *const cpu_online_mask = to_cpumask(cpu_online_bits);
EXPORT_SYMBOL(cpu_online_mask);

static DECLARE_BITMAP(cpu_present_bits, CONFIG_NR_CPUS) __read_mostly;
const struct cpumask *const cpu_present_mask = to_cpumask(cpu_present_bits);
EXPORT_SYMBOL(cpu_present_mask);
```

```
static DECLARE_BITMAP(cpu_active_bits, CONFIG_NR_CPUS) __read_mostly;
const struct cpumask *const cpu_active_mask = to_cpumask(cpu_active_bits);
EXPORT_SYMBOL(cpu_active_mask);
```

上面的信息通过他们的名字就可以理解,需要注意的一点就是.online mask与active mask区别是:

- cpu_online_mask:表示当前系统online的cpu,包括idle+active cpu
- cpu_active_mask:表示当前系统online并且active的cpu

下面看看cpu topology怎么建立起来并经过初始化建立起sched domain和sched group的关系的.当boot cpu启动丛cpu的时候,cpu topology就开始建立:

start_kernel-->rest_init-->kernel_init-->kernel_init_freeable-->sched_init_smp-->init_sched_domains:

```
/*
 * Set up scheduler domains and groups. Callers must hold the hotplug lock.
 * For now this just excludes isolated cpus, but could be used to
 * exclude other special cases in the future.
 */
static int init_sched_domains(const struct cpumask *cpu_map)
{
    int err;
    /*ARM没有定义*/
    arch_update_cpu_topology();
    ndoms_cur = 1;
    /*分配一个cpumask结构体变量*/
    doms_cur = alloc_sched_domains(ndoms_cur);
    if (!doms_cur)
        doms_cur = &fallback_doms;
    /*doms_curr[0] = cpu_map & (~cpu_isolated_map),cpumask的与非的计算方式
    cpu_isolated_map表示有独立的domain的cpu,当前在建立的cpu topology过程中需要
   剔除cpu_isolated_map cpu*/
    cpumask_andnot(doms_cur[0], cpu_map, cpu_isolated_map);
    /*开始建立cpu topology*/
    err = build_sched_domains(doms_cur[0], NULL);
    /*注册sched domain的系统控制接口*/
    register_sched_domain_sysctl();

    return err;
}
/*
 * arch_update_cpu_topology lets virtualized architectures update the
 * cpu core maps. It is supposed to return 1 if the topology changed
 * or 0 if it stayed the same.
 */
int __weak arch_update_cpu_topology(void)
{
    return 0;
}

cpumask_var_t *alloc_sched_domains(unsigned int ndoms)
{
    int i;
    cpumask_var_t *doms;

    doms = kmalloc(sizeof(*doms) * ndoms, GFP_KERNEL);
    if (!doms)
        return NULL;
```

```
    for (i = 0; i < ndoms; i++) {
        if (!alloc_cpumask_var(&doms[i], GFP_KERNEL)) {
            free_sched_domains(doms, i);
            return NULL;
        }
    }
    return doms;
}

/*上面的cpu_map是active_cpu_mask,那么active cpumask在哪里设置的呢?*/
/* Called by boot processor to activate the rest. */
/*arm_dt_init_cpu_maps()函数会初始化若干个cpumask变量比如possible和present cpumask这个函数会将所有的cpu online起来,同时会等到online cpumask和active cpumask*/
void __init smp_init(void)
{
    unsigned int cpu;

    idle_threads_init();

    /* FIXME: This should be done in userspace --RR */
    for_each_present_cpu(cpu) {
        if (num_online_cpus() >= setup_max_cpus)
            break;
        if (!cpu_online(cpu))
            cpu_up(cpu);
    }

    /* Any cleanup work */
    smp_announce();
    smp_cpus_done(setup_max_cpus);
}

```

接下来解析真正构建cpu topology结构的函数:build_sched_domians

```
/*
 * Build sched domains for a given set of cpus and attach the sched domains
 * to the individual cpus
 */
static int build_sched_domains(const struct cpumask *cpu_map,
                   struct sched_domain_attr *attr)
{
    enum s_alloc alloc_state;
    struct sched_domain *sd;
    struct s_data d;
    int i, ret = -ENOMEM;

    alloc_state = __visit_domain_allocation_hell(&d, cpu_map);
    if (alloc_state != sa_rootdomain)
        goto error;

    /* Set up domains for cpus specified by the cpu_map. */
    for_each_cpu(i, cpu_map) {
        struct sched_domain_topology_level *tl;

        sd = NULL;
        for_each_sd_topology(tl) {

            sd = build_sched_domain(tl, cpu_map, attr, sd, i);
            if (tl == sched_domain_topology)
                *per_cpu_ptr(d.sd, i) = sd;
```

```
            if (tl->flags & SDTL_OVERLAP || sched_feat(FORCE_SD_OVERLAP))
                sd->flags |= SD_OVERLAP;
        }
    }

    /* Build the groups for the domains */
    for_each_cpu(i, cpu_map) {
        for (sd = *per_cpu_ptr(d.sd, i); sd; sd = sd->parent) {
            sd->span_weight = cpumask_weight(sched_domain_span(sd));
            if (sd->flags & SD_OVERLAP) {
                if (build_overlap_sched_groups(sd, i))
                    goto error;
            } else {
                if (build_sched_groups(sd, i))
                    goto error;
            }
        }
    }

    /* Calculate CPU capacity for physical packages and nodes */
    for (i = nr_cpumask_bits-1; i >= 0; i--) {
        struct sched_domain_topology_level *tl = sched_domain_topology;

        if (!cpumask_test_cpu(i, cpu_map))
            continue;

        for (sd = *per_cpu_ptr(d.sd, i); sd; sd = sd->parent, tl++) {
            init_sched_energy(i, sd, tl->energy);
            claim_allocations(i, sd);
            init_sched_groups_capacity(i, sd);
        }
    }

    /* Attach the domains */
    rcu_read_lock();
    for_each_cpu(i, cpu_map) {
        int max_cpu = READ_ONCE(d.rd->max_cap_orig_cpu);
        int min_cpu = READ_ONCE(d.rd->min_cap_orig_cpu);

        if ((max_cpu < 0) || (cpu_rq(i)->cpu_capacity_orig >
            cpu_rq(max_cpu)->cpu_capacity_orig))
            WRITE_ONCE(d.rd->max_cap_orig_cpu, i);

        if ((min_cpu < 0) || (cpu_rq(i)->cpu_capacity_orig <
            cpu_rq(min_cpu)->cpu_capacity_orig))
            WRITE_ONCE(d.rd->min_cap_orig_cpu, i);

        sd = *per_cpu_ptr(d.sd, i);

        cpu_attach_domain(sd, d.rd, i);
    }
    rcu_read_unlock();

    ret = 0;
```

```
error:
    __free_domain_allocs(&d, alloc_state, cpu_map);
    return ret;
}
```

对构建cpu topology函数分如下几个部分来分析:

1.分配空间:在build_sched_domains函数里面调用__visit_domain_allocation_hell函数

```
static enum s_alloc __visit_domain_allocation_hell(struct s_data *d,
                                   const struct cpumask *cpu_map)
{
    memset(d, 0, sizeof(*d));
    /*核心函数:创建调度域等数据结构,详细看下面的解析*/
    if (__sdt_alloc(cpu_map))
        return sa_sd_storage;
    d->sd = alloc_percpu(struct sched_domain *);
    if (!d->sd)
        return sa_sd_storage;
    d->rd = alloc_rootdomain();
    if (!d->rd)
        return sa_sd;
    return sa_rootdomain;
}
static int __sdt_alloc(const struct cpumask *cpu_map)
{
    struct sched_domain_topology_level *tl;
    int j;
/*
static struct sched_domain_topology_level *sched_domain_topology =
       default_topology;
#define for_each_sd_topology(tl)                  \
       for (tl = sched_domain_topology; tl->mask; tl++)
*/  /*对default_topology结构体数据进行遍历*/
    for_each_sd_topology(tl) {
        /*获取每个SDTL层级的sd_data数据结构指针*/
        struct sd_data *sdd = &tl->data;
        /*下面三个alloc_percpu为每一个SDTL层级的实体数据结构sd_data分配空间
         1.sched domain
         2. sched group
         3. sched group capacity*/
        sdd->sd = alloc_percpu(struct sched_domain *);
        if (!sdd->sd)
            return -ENOMEM;

        sdd->sg = alloc_percpu(struct sched_group *);
        if (!sdd->sg)
            return -ENOMEM;

        sdd->sgc = alloc_percpu(struct sched_group_capacity *);
        if (!sdd->sgc)
            return -ENOMEM;
        /*针对每个active_cpu_mask进行遍历,每个cpu上都需要为sd/sg/sgc分配空间,并存放
        在percpu变量中.注意到sg->next指针,有两个不同的用途:
        1.对于MC 层级,不同cluster的每个cpu core sched group会组成一个链表,比如两个
        cluster,每个cluster 四个core,则每个cluster的sched group链表长度都为4,每个
        core的sched domain都有一个sched group.
```

```
        2. DIE层级,不同cluster的sched group会链接起来.一个cluster所有core
         的sched domain都共享一个sched group,后面会以图表说明*/
        for_each_cpu(j, cpu_map) {
            struct sched_domain *sd;
            struct sched_group *sg;
            struct sched_group_capacity *sgc;

            sd = kzalloc_node(sizeof(struct sched_domain) + cpumask_size(),
                    GFP_KERNEL, cpu_to_node(j));
            if (!sd)
                return -ENOMEM;

            *per_cpu_ptr(sdd->sd, j) = sd;

            sg = kzalloc_node(sizeof(struct sched_group) + cpumask_size(),
                    GFP_KERNEL, cpu_to_node(j));
            if (!sg)
                return -ENOMEM;

            sg->next = sg;

            *per_cpu_ptr(sdd->sg, j) = sg;

            sgc = kzalloc_node(sizeof(struct sched_group_capacity) +
cpumask_size(),
                    GFP_KERNEL, cpu_to_node(j));
            if (!sgc)
                return -ENOMEM;

            *per_cpu_ptr(sdd->sgc, j) = sgc;
        }
    }

    return 0;
}static int __sdt_alloc(const struct cpumask *cpu_map)
{
    struct sched_domain_topology_level *tl;
    int j;

    for_each_sd_topology(tl) {
        struct sd_data *sdd = &tl->data;

        sdd->sd = alloc_percpu(struct sched_domain *);
        if (!sdd->sd)
            return -ENOMEM;

        sdd->sg = alloc_percpu(struct sched_group *);
        if (!sdd->sg)
            return -ENOMEM;

        sdd->sgc = alloc_percpu(struct sched_group_capacity *);
        if (!sdd->sgc)
            return -ENOMEM;

```

```c
        for_each_cpu(j, cpu_map) {
            struct sched_domain *sd;
            struct sched_group *sg;
            struct sched_group_capacity *sgc;

            sd = kzalloc_node(sizeof(struct sched_domain) + cpumask_size(),
                    GFP_KERNEL, cpu_to_node(j));
            if (!sd)
                return -ENOMEM;

            *per_cpu_ptr(sdd->sd, j) = sd;

            sg = kzalloc_node(sizeof(struct sched_group) + cpumask_size(),
                    GFP_KERNEL, cpu_to_node(j));
            if (!sg)
                return -ENOMEM;

            sg->next = sg;

            *per_cpu_ptr(sdd->sg, j) = sg;

            sgc = kzalloc_node(sizeof(struct sched_group_capacity) +
cpumask_size(),
                    GFP_KERNEL, cpu_to_node(j));
            if (!sgc)
                return -ENOMEM;

            *per_cpu_ptr(sdd->sgc, j) = sgc;
        }
    }

    return 0;
}
```

说明如下:

- 每个SDTL层级都有一个struct sched_domain_topology_level数据结构来描述,并且内嵌了一个struct sd_data数据结构,包含sched_domain,sched_group,sched_group_capacity的二级指针
- 每个SDTL层级都分配一个percpu变量sched_domain,sched_group,sched_group_capacity数据结构
- 在每个SDTL层级中为每个cpu都分配sched_domain,sched_group,sched_group_capacity数据结构,即每个cpu在没给SDTL层级中都有对应的调度域和调度组.

2. 初始化sched_domain:build_sched_domain函数,__visit_domain_allocation_hell函数已经为每个SDTL层级分配了sched_domain,sched_group/sched_group_capacity空间了,有看空间之后,就开始为cpu_map建立调度域关系了:

```c
build_sched_domains--->build_sched_domain
    /* Set up domains for cpus specified by the cpu_map. */
    /*对每个cpu_map进行遍历,并对每个CPU的SDTL层级进行遍历,相当于每个CPU都有一套SDTL对
应的调度域,为每个CPU都初始化一整套SDTL对应的调度域和调度组,每个
  CPU的每个SDTL都调用build_sched_domain函数来建立调度域和调度组*/
    for_each_cpu(i, cpu_map) {
```

```
            struct sched_domain_topology_level *tl;

            sd = NULL;
            for_each_sd_topology(tl) {
                sd = build_sched_domain(tl, cpu_map, attr, sd, i);
                if (tl == sched_domain_topology)
                    *per_cpu_ptr(d.sd, i) = sd;
                if (tl->flags & SDTL_OVERLAP || sched_feat(FORCE_SD_OVERLAP))
                    sd->flags |= SD_OVERLAP;
            }
        }
  |->
  static struct sched_domain *
  sd_init(struct sched_domain_topology_level *tl,
      struct sched_domain *child, int cpu)
  {   /*获取cpu的sched_domain实体*/
      struct sched_domain *sd = *per_cpu_ptr(tl->data.sd, cpu);
      int sd_weight, sd_flags = 0;

  #ifdef CONFIG_NUMA
      /*
       * Ugly hack to pass state to sd_numa_mask()...
       */
      sched_domains_curr_level = tl->numa_level;
  #endif
      /*通过default_topology[]结构体数据填充对应的参数,涉及到一些cpumask,这个上面已经
      讲过了*/
      sd_weight = cpumask_weight(tl->mask(cpu));

      if (tl->sd_flags)
          sd_flags = (*tl->sd_flags)();
      if (WARN_ONCE(sd_flags & ~TOPOLOGY_SD_FLAGS,
              "wrong sd_flags in topology description\n"))
          sd_flags &= ~TOPOLOGY_SD_FLAGS;
       /*填充sched_domain一些成员变量*/
      *sd = (struct sched_domain){
          .min_interval       = sd_weight,
          .max_interval       = 2*sd_weight,
          .busy_factor        = 32,
          .imbalance_pct      = 125,

          .cache_nice_tries   = 0,
          .busy_idx       = 0,
          .idle_idx       = 0,
          .newidle_idx        = 0,
          .wake_idx       = 0,
          .forkexec_idx       = 0,

          .flags          = 1*SD_LOAD_BALANCE
                      | 1*SD_BALANCE_NEWIDLE
                      | 1*SD_BALANCE_EXEC
                      | 1*SD_BALANCE_FORK
                      | 0*SD_BALANCE_WAKE
                      | 1*SD_WAKE_AFFINE
```

```
                      | 0*SD_SHARE_CPUCAPACITY
                      | 0*SD_SHARE_PKG_RESOURCES
                      | 0*SD_SERIALIZE
                      | 0*SD_PREFER_SIBLING
                      | 0*SD_NUMA
#ifdef CONFIG_INTEL_DWS
                      | 0*SD_INTEL_DWS
#endif
                      | sd_flags
                      ,

        .last_balance       = jiffies,
        .balance_interval   = sd_weight,
        .smt_gain       = 0,
        .max_newidle_lb_cost    = 0,
        .next_decay_max_lb_cost = jiffies,
        .child          = child,
#ifdef CONFIG_INTEL_DWS
        .dws_tf         = 0,
#endif
#ifdef CONFIG_SCHED_DEBUG
        .name           = tl->name,
#endif
    };

    /*
     * Convert topological properties into behaviour.
     */

    if (sd->flags & SD_ASYM_CPUCAPACITY) {
        struct sched_domain *t = sd;

        for_each_lower_domain(t)
            t->flags |= SD_BALANCE_WAKE;
    }

    if (sd->flags & SD_SHARE_CPUCAPACITY) {
        sd->flags |= SD_PREFER_SIBLING;
        sd->imbalance_pct = 110;
        sd->smt_gain = 1178; /* ~15% */

    } else if (sd->flags & SD_SHARE_PKG_RESOURCES) {
        sd->imbalance_pct = 117;
        sd->cache_nice_tries = 1;
        sd->busy_idx = 2;

#ifdef CONFIG_NUMA
    } else if (sd->flags & SD_NUMA) {
        sd->cache_nice_tries = 2;
        sd->busy_idx = 3;
        sd->idle_idx = 2;

        sd->flags |= SD_SERIALIZE;
```

```c
        if (sched_domains_numa_distance[tl->numa_level] > RECLAIM_DISTANCE)
{
            sd->flags &= ~(SD_BALANCE_EXEC |
                           SD_BALANCE_FORK |
                           SD_WAKE_AFFINE);
        }

#endif
    } else {
        sd->flags |= SD_PREFER_SIBLING;
        sd->cache_nice_tries = 1;
        sd->busy_idx = 2;
        sd->idle_idx = 1;
    }

#ifdef CONFIG_INTEL_DWS
    if (sd->flags & SD_INTEL_DWS)
        sd->dws_tf = 120;
    if (sd->flags & SD_INTEL_DWS && sd->flags & SD_SHARE_PKG_RESOURCES)
        sd->dws_tf = 160;

#endif

    sd->private = &tl->data;

    return sd;
}

struct sched_domain *build_sched_domain(struct sched_domain_topology_level
*tl,
        const struct cpumask *cpu_map, struct sched_domain_attr *attr,
        struct sched_domain *child, int cpu)
{   /*上面解释了sd_init函数*/
    struct sched_domain *sd = sd_init(tl, child, cpu);

    cpumask_and(sched_domain_span(sd), cpu_map, tl->mask(cpu));
    if (child) {
        sd->level = child->level + 1;
        sched_domain_level_max = max(sched_domain_level_max, sd->level);
        child->parent = sd;

        if (!cpumask_subset(sched_domain_span(child),
                    sched_domain_span(sd))) {
            pr_err("BUG: arch topology borken\n");
#ifdef CONFIG_SCHED_DEBUG
            pr_err("     the %s domain not a subset of the %s domain\n",
                    child->name, sd->name);
#endif
            /* Fixup, ensure @sd has at least @child cpus. */
            cpumask_or(sched_domain_span(sd),
                   sched_domain_span(sd),
                   sched_domain_span(child));
        }

```

```
    }
    set_domain_attribute(sd, attr);

    return sd;
}
```
说明如下:

- sd_init函数通过default_topology结构体填充sched_domain结构体并填充相关的一些参数
- cpumask_and(sched_domain_span(sd), cpu_map, tl->mask(cpu)); tl->mask(cpu)返回该cpu某个SDTL层级下兄弟cpu的bitmap,cpumask_and则将tl->mask(cpu)的bitmap复制到span数组中.
- 我们知道遍历的顺序是SMT-->MC-->DIE,可以看成父子关系或者上下级关系,struct sched_domain数据结构中有parent和child成员用于描述此关系.
- 后面完成了对调度域的初始化.

3.初始化sched_group.

```
build_sched_domains--->build_sched_groups

/* Build the groups for the domains */
    for_each_cpu(i, cpu_map) {
        /*per_cpu_ptr(d.sd, i)获取每个cpu的最低层级的SDTL,并且通过指针sd->parent遍
历所有层级*/
        for (sd = *per_cpu_ptr(d.sd, i); sd; sd = sd->parent) {
            /*cpumask_and(sched_domain_span(sd), cpu_map, tl->mask(cpu))
     某个SDTL层级cpu兄弟 cpu bitmap,*/
            sd->span_weight = cpumask_weight(sched_domain_span(sd));
            if (sd->flags & SD_OVERLAP) {
                if (build_overlap_sched_groups(sd, i))
                    goto error;
            } else {
                if (build_sched_groups(sd, i))
                    goto error;
            }
        }
    }
-->
/*
 * build_sched_groups will build a circular linked list of the groups
 * covered by the given span, and will set each group's ->cpumask correctly,
 * and ->cpu_capacity to 0.
 *
 * Assumes the sched_domain tree is fully constructed
 * 核心函数
 */
static int
build_sched_groups(struct sched_domain *sd, int cpu)
{
    struct sched_group *first = NULL, *last = NULL;
    struct sd_data *sdd = sd->private;
    const struct cpumask *span = sched_domain_span(sd);
    struct cpumask *covered;
    int i;

```

```
	get_group(cpu, sdd, &sd->groups);
	atomic_inc(&sd->groups->ref);

	if (cpu != cpumask_first(span))
		return 0;

	lockdep_assert_held(&sched_domains_mutex);
	covered = sched_domains_tmpmask;

	cpumask_clear(covered);
	/*span是某个SDTL层级的cpu bitmap,实质某个sched domain包含多少个cpu*/
	/*两个for循环依次设置了该调度域sd中不同CPU对应的调度组的包含关系,这些调度组通过next
指针串联起来*/
	for_each_cpu(i, span) {
		struct sched_group *sg;
		int group, j;

		if (cpumask_test_cpu(i, covered))
			continue;

		group = get_group(i, sdd, &sg);
		cpumask_setall(sched_group_mask(sg));

		for_each_cpu(j, span) {
			if (get_group(j, sdd, NULL) != group)
				continue;

			cpumask_set_cpu(j, covered);
			cpumask_set_cpu(j, sched_group_cpus(sg));
		}

		if (!first)
			first = sg;
		if (last)
			last->next = sg;
		last = sg;
	}
	/*将一个sched domain里面的所有sched group串成一组链表*/
	last->next = first;

	return 0;
}
static int get_group(int cpu, struct sd_data *sdd, struct sched_group **sg)
{
	struct sched_domain *sd = *per_cpu_ptr(sdd->sd, cpu);
	struct sched_domain *child = sd->child;
	/*如果是SDTL=DIE,那么child为true*/
	if (child)
		cpu = cpumask_first(sched_domain_span(child));

	if (sg) {
		*sg = *per_cpu_ptr(sdd->sg, cpu);
		(*sg)->sgc = *per_cpu_ptr(sdd->sgc, cpu);
		atomic_set(&(*sg)->sgc->ref, 1); /* for claim_allocations */
	}

	return cpu;
```

```
}

```

4 初始化sched group capacity:

```
/* Calculate CPU capacity for physical packages and nodes */
for (i = nr_cpumask_bits-1; i >= 0; i--) {
    struct sched_domain_topology_level *tl = sched_domain_topology;

    if (!cpumask_test_cpu(i, cpu_map))
        continue;

    for (sd = *per_cpu_ptr(d.sd, i); sd; sd = sd->parent, tl++) {
        init_sched_energy(i, sd, tl->energy);
        claim_allocations(i, sd);
        init_sched_groups_capacity(i, sd);
    }
}
/*上面的函数目的是填充下面的结构体每个cpu的每个SDTL层级的sched group capacity*/
struct sched_group_capacity {
    atomic_t ref;
    /*
     * CPU capacity of this group, SCHED_LOAD_SCALE being max capacity
     * for a single CPU.
     */
    unsigned long capacity;
    unsigned long max_capacity; /* Max per-cpu capacity in group */
    unsigned long min_capacity; /* Min per-CPU capacity in group */
    unsigned long next_update;
    int imbalance; /* XXX unrelated to capacity but shared group state */
    /*
     * Number of busy cpus in this group.
     */
    atomic_t nr_busy_cpus;

    unsigned long cpumask[0]; /* iteration mask */
};

```

5. 将每个cpu的调度域与每个cpu的struct rq root domain成员关联起来

```
    /* Attach the domains */
    rcu_read_lock();
    for_each_cpu(i, cpu_map) {
        int max_cpu = READ_ONCE(d.rd->max_cap_orig_cpu);
        int min_cpu = READ_ONCE(d.rd->min_cap_orig_cpu);

        if ((max_cpu < 0) || (cpu_rq(i)->cpu_capacity_orig >
            cpu_rq(max_cpu)->cpu_capacity_orig))
            WRITE_ONCE(d.rd->max_cap_orig_cpu, i);

        if ((min_cpu < 0) || (cpu_rq(i)->cpu_capacity_orig <
            cpu_rq(min_cpu)->cpu_capacity_orig))
            WRITE_ONCE(d.rd->min_cap_orig_cpu, i);

        sd = *per_cpu_ptr(d.sd, i);
```

```
        /*将每个cpu的调度域关联到每个cpu的rq的root_domain上*/
        cpu_attach_domain(sd, d.rd, i);
    }
    rcu_read_unlock();

/*
 * Attach the domain 'sd' to 'cpu' as its base domain.
Callers must
 * hold the hotplug lock.
 */
static void
cpu_attach_domain(struct sched_domain *sd, struct root_domain
*rd, int cpu)
{
    struct rq *rq = cpu_rq(cpu);
    struct sched_domain *tmp;

    /* Remove the sched domains which do not contribute to
scheduling. */
    for (tmp = sd; tmp; ) {
        struct sched_domain *parent = tmp->parent;
        if (!parent)
            break;

        if (sd_parent_degenerate(tmp, parent)) {
            tmp->parent = parent->parent;
            if (parent->parent)
                parent->parent->child = tmp;
            /*
             * Transfer SD_PREFER_SIBLING down in case of a
             * degenerate parent; the spans match for this
             * so the property transfers.
             */
            if (parent->flags & SD_PREFER_SIBLING)
                tmp->flags |= SD_PREFER_SIBLING;
            destroy_sched_domain(parent, cpu);
        } else
            tmp = tmp->parent;
    }

    if (sd && sd_degenerate(sd)) {
        tmp = sd;
        sd = sd->parent;
        destroy_sched_domain(tmp, cpu);
        if (sd)
            sd->child = NULL;
    }

    sched_domain_debug(sd, cpu);

    rq_attach_root(rq, rd);
```

```
        tmp = rq->sd;
        rcu_assign_pointer(rq->sd, sd);
        destroy_sched_domains(tmp, cpu);

        update_top_cache_domain(cpu);

    #ifdef CONFIG_INTEL_DWS
        update_dws_domain(sd, cpu);
    #endif
    }
```

通过调度域,调度组的建立过程,可以通过下面的图来直观的说明SDTL不同层级对应关系: