

DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

目录视图

摘要视图

RSS 订阅

个人资料



DroidPhone

访问：1251838次
积分：9517
等级：
排名：第1606名

原创：51篇 转载：0篇
译文：4篇 评论：555条

文章搜索

文章分类

移动开发之Android (11)
Linux内核架构 (15)
Linux设备驱动 (20)
Linux电源管理 (3)
Linux音频子系统 (15)
Linux中断子系统 (5)
Linux时间管理系统 (8)
Linux输入子系统 (4)

文章存档

2014年07月 (1)
2014年04月 (4)
2013年11月 (4)
2013年10月 (3)
2013年07月 (3)

展开

阅读排行

Linux ALSA声卡驱动之- (87756)
Android Audio System 之 (61776)
Linux ALSA声卡驱动之- (52760)
Linux ALSA声卡驱动之- (46894)
Android Audio System 之

【活动】Python创意编程活动开始啦!!! CSDN日报20170425 ——《私活，是对技术达人最好的点赞》【CSDN日报】 | 3.18-4.18 上榜作者排行出炉

Linux时间子系统之六：高精度定时器（HRTIMER）的原理和实现

2012-10-19 23:05 44239人阅读 评论(26) 收藏 举报

分类：Linux内核架构（14） Linux时间管理系统（7）

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?) [+]

上一篇文章，我介绍了传统的**低分辨率定时器**的实现原理。而随着内核的不断演进，大牛们已经对这种低分辨率定时器的精度不再满足，而且，硬件也在不断地发展，系统中的定时器硬件的精度也越来越高，这也给高分辨率定时器的出现创造了条件。内核从2.6.16开始加入了高精度定时器**架构**。在实现方式上，内核的高分辨率定时器的实现代码几乎没有借用低分辨率定时器的**数据结构**和代码，内核文档给出的解释主要有以下几点：

- 低分辨率定时器的代码和jiffies的关系太过紧密，并且默认按32位进行设计，并且它的代码已经经过长时间的优化，目前的使用也是没有任何错误，如果硬要基于它来实现高分辨率定时器，势必会打破原有的时间轮概念，并且会引入一大堆if--#else判断；
- 虽然大部分时间里，时间轮可以实现O(1)时间复杂度，但是当有进位发生时，不可预测的O(N)定时器级联迁移时间，这对于低分辨率定时器来说问题不大，可是它大大地影响了定时器的精度；
- 低分辨率定时器几乎是“超时”而设计的，并为此对它进行了大量的优化，对于这些以“超时”为目的而使用定时器，它们大多数期望在超时到来之前获得正确的结果，然后删除定时器，精确时间并不是它们主要的目的，例如网络通信、设备IO等等。

为此，内核为高精度定时器重新设计了一套软件架构，它可以为我们提供纳秒级的定时精度，以满足对精确时间有迫切需求的应用程序或内核驱动，例如多媒体应用，音频设备的驱动程序等等。以下的讨论用hrtimer(high resolution timer)表示高精度定时器。

```
/******  
声明：本博内容均由http://blog.csdn.NET/droidphone原创，转载请注明出处，谢谢！  
*****
```

1. 如何组织hrtimer？

我们知道，低分辨率定时器使用5个链表数组来组织timer_list结构，形成了著名的时间轮概念，对于高分辨率定时器，我们期望组织它们的数据结构至少具备以下条件：

- 稳定而且快速的查找能力；
- 快速地插入和删除定时器的能力；
- 排序功能；

内核的开发者考察了多种数据结构，例如基数树、哈希表等等，最终他们选择了红黑树（rbtree）来组织hrtimer，红黑树已经以库的形式存在于内核中，并被成功地使用在内存管理子系统和文件系统中，随着系统的运行，hrtimer不停地被创建和销毁，新的hrtimer按顺序被插入到红黑树中，树的最左边的节点就是最快到期的定时器，内核用一个hrtimer结构来表示一个高精度定时器：

```
01. struct hrtimer {  
    [cpp]
```

http://blog.csdn.net/droidphone/article/details/8074892

1/14

Linux时间子系统之六：高精度定时器（HRTIMER）的原理和实现 - DroidPhone的专栏 - 博客频道 - CSDN.NET	(46469)
Linux ALSA声卡驱动之六：ASo ganye88: 活捉一只	(44235)
Linux ALSA声卡驱动之五：移动qq_21836933: 看了很多遍，越看觉得思路越清晰，博主确实厉害，我服	(43530)
Linux ALSA声卡驱动之一：ALS qianxuedegushi: " drivers 放置一些与CPU、BUS架构无关的公用代码"是"有关"还...	(41137)
Android Audio System 之一：AmazingDLC: 楼主你好，我目前在做音频相关的工作，经常会接触到音频方面的代码，但是现在自己一点基础都没有（应届生）...	(37842)
Linux ALSA声卡驱动之六：ASo qq_21836933: 看了很多遍，越看觉得思路越清晰，博主确实厉害，我服	(36875)

评论排行	
Android Audio System 之一：AmazingDLC: 楼主你好，我目前在做音频相关的工作，经常会接触到音频方面的代码，但是现在自己一点基础都没有（应届生）...	(57)
Linux ALSA声卡驱动之六：ASo ganye88: 活捉一只	(42)
Linux ALSA声卡驱动之五：移动qq_21836933: 看了很多遍，越看觉得思路越清晰，博主确实厉害，我服	(35)
Linux时间子系统之六：高精度定时器（HRTIMER）的原理和实现 - DroidPhone的专栏 - 博客频道 - CSDN.NET	(26)
Linux中断（interrupt）之四：牛牛牛 谢谢楼主分享	(24)
Linux ALSA声卡驱动之六：ASo ganye88: 活捉一只	(22)
Android SurfaceFlinger 之六：牛牛牛 谢谢楼主分享	(21)
Linux ALSA声卡驱动之六：ASo ganye88: 活捉一只	(19)
Android Audio System 之一：AmazingDLC: 楼主你好，我目前在做音频相关的工作，经常会接触到音频方面的代码，但是现在自己一点基础都没有（应届生）...	(18)
Linux中断（interrupt）之四：牛牛牛 谢谢楼主分享	(18)

推荐文章	
* 探索通用可编程数据平面	
* 这是一份很有诚意的 Protocol Buffer 语法详解	
* CSDN日报20170420 ——《开发和产品之间的恩怨从何来?》	
* Android图片加载框架最全解析——从源码的角度理解Glide的执行流程	
* 如果两个程序员差不多，选写作能力更好的那个	
* 从构造函数看线程安全	

最新评论	
Linux时间子系统之一：clock so huanguansong: 我同意4楼的说法是正确的。另外我不理解那个10分钟以及idle状态下时间就不准的意思。	
Linux ALSA声卡驱动之五：移动张鸢: 我还提不出问题，怎么办呢！	
Android Audio System 之一：AmazingDLC: 楼主你好，我目前在做音频相关的工作，经常会接触到音频方面的代码，但是现在自己一点基础都没有（应届生）...	
Linux时间子系统之一：clock so Cowincent: 请问咱们的程序流向图使用什么软件画的	
Linux ALSA声卡驱动之七：ASo ganye88: 活捉一只	
Linux ALSA声卡驱动之一：ALS qianxuedegushi: " drivers 放置一些与CPU、BUS架构无关的公用代码"是"有关"还...	
Linux ALSA声卡驱动之一：ALS chenzen1080: 牛牛牛 谢谢楼主分享	
Linux ALSA声卡驱动之六：ASo qq_21836933: 看了很多遍，越看觉得思路越清晰，博主确实厉害，我服	
Linux ALSA声卡驱动之五：移动qq_21836933: 建议楼主可以在网上开课，花钱都愿意听楼主讲课！	
Linux SPI总线和设备驱动架构之	

```
02. struct timerqueue_node node;
03. ktime_t _softexpires;
04. enum hrtimer_restart (*function)(struct hrtimer *);
05. struct hrtimer_clock_base *base;
06. unsigned long state;
07. ....
08. };
```

定时器的到期时间用ktime_t来表示，_softexpires字段记录了时间，定时器一旦到期，function字段指定的回调函数会被调用，该函数的返回值为一个枚举值，它决定了该hrtimer是否需要被重新激活：

```
[cpp]
01. enum hrtimer_restart {
02.     HRTIMER_NORESTART, /* Timer is not restarted */
03.     HRTIMER_RESTART, /* Timer must be restarted */
04. };
```

state字段用于表示hrtimer当前的状态，有以下几种位组合：

```
[cpp]
01. #define HRTIMER_STATE_INACTIVE 0x00 // 定时器未激活
02. #define HRTIMER_STATE_ENQUEUED 0x01 // 定时器已经被排入红黑树中
03. #define HRTIMER_STATE_CALLBACK 0x02 // 定时器的回调函数正在被调用
04. #define HRTIMER_STATE_MIGRATE 0x04 // 定时器正在CPU之间做迁移
```

hrtimer的到期时间可以基于以下几种时间基准系统：

```
[cpp]
01. enum hrtimer_base_type {
02.     HRTIMER_BASE_MONOTONIC, // 单调递增的monotonic时间, 不包含休眠时间
03.     HRTIMER_BASE_REALTIME, // 平常使用的墙上真实时间
04.     HRTIMER_BASE_BOOTTIME, // 单调递增的boottime, 包含休眠时间
05.     HRTIMER_MAX_CLOCK_BASES, // 用于后续数组的定义
06. };
```

和分辨率定时器一样，处于效率和上锁的考虑，每个cpu单独管理属于自己的hrtimer，为此，专门定义了一个结构hrtimer_cpu_base：

```
[cpp]
01. struct hrtimer_cpu_base {
02.     ....
03.     struct hrtimer_clock_base clock_base[HRTIMER_MAX_CLOCK_BASES];
04. };
```

其中，clock_base数组为每种时间基准系统都定义了一个hrtimer_clock_base结构，它的定义如下：

```
[cpp]
01. struct hrtimer_clock_base {
02.     struct hrtimer_cpu_base *cpu_base; // 指向所属cpu的hrtimer_cpu_base结构
03.     ....
04.     struct timerqueue_head active; // 红黑树, 包含了所有使用该时间基准系统的hrtimer
05.     ktime_t resolution; // 时间基准系统的分辨率
06.     ktime_t (*get_time)(void); // 获取该基准系统的时间函数
07.     ktime_t softirq_time; // 当用jiffies
08.     ktime_t offset; //
09. };
```

active字段是一个timerqueue_head结构，它实际上是对rbtree的进一步封装：

```
[cpp]
01. struct timerqueue_node {
02.     struct rb_node node; // 红黑树的节点
03.     ktime_t expires; // 该节点代表hrtimer的到期时间, 与hrtimer结构中的_softexpires稍有不同
04. };
05.
```

gyfkyu: 写的很好，怒赞！！不过假如楼主能写写具体spi从设备驱动的实现就更好了。这样的话你就把控制器驱动、中间...

```
06. struct timerqueue_head {
07.     struct rb_root head;           // 红黑树的根节点
08.     struct timerqueue_node *next; // 该红黑树中最早到期的节点,也就是最左下的节点
09. };
```

timerqueue_head结构在红黑树的基础上，增加了一个next字段，用于保存树中最先到期的定时器节点，实际上就是树的最左下方的节点，有了next字段，当到期事件到来时，系统不必遍历整个红黑树，只要取出next字段对应的节点进行处理即可。timerqueue_node用于表示一个hrtimer节点，它在标准红黑树节点rb_node的基础上增加了expires字段，该字段和hrtimer中的_softwareexpires字段一起，设定了hrtimer的到期时间的一个范围，hrtimer可以在hrtimer_softwareexpires至timerqueue_node.expires之间的任何时刻到期，我们也称timerqueue_node.expires为硬过期时间(hard)，意思很明显：到了此时刻，定时器一定会到期，有了这个范围可以选择，定时器系统可以让范围接近的多个定时器在同一时刻同时到期，这种设计可以降低进程频繁地被hrtimer进行唤醒。经过以上的讨论，我们可以得出以下的图示，它表明了每个cpu上的hrtimer是如何被组织在一起的：

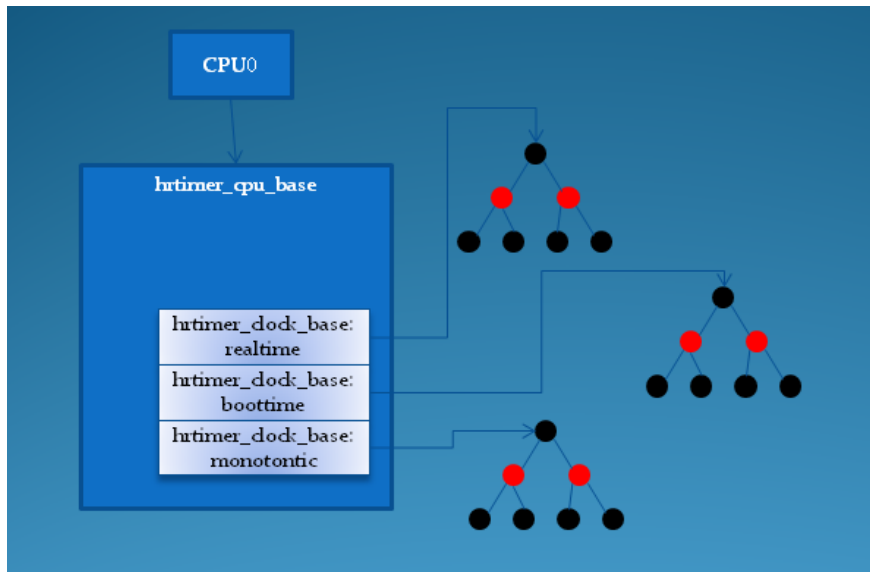


图 1.1 每个cpu的hrtimer组织结构

总结一下：

- 每个cpu有一个hrtimer_cpu_base结构；
- hrtimer_cpu_base结构管理着3种不同的时间基准系统的hrtimer，分别是：实时时间，启动时间和单调时间；
- 每种时间基准系统通过它的active字段（timerqueue_head结构指针），指向它们各自的红黑树；
- 红黑树上，按到期时间进行排序，最早到期的hrtimer位于最左下的节点，并被记录在active.next字段中；
- 3中时间基准的最先到期时间可能不同，所以，它们之中最早到期的时间被记录在hrtimer_cpu_base的expires_next字段中。

2. hrtimer如何运转

hrtimer的实现需要一定的硬件基础，它的实现依赖于我们前几章介绍的timekeeper和clock_event_device，如果你对timekeeper和clock_event_device不了解请参考以下文章：[Linux时间子系统之三：时间的维护者：timekeeper](#)，[Linux时间子系统之四：定时器的引擎：clock_event_device](#)。hrtimer系统需要通过timekeeper获取当前的时间，计算与到期时间的差值，并根据该差值，设定该cpu的tick_device（clock_event_device）的下一次的到期时间，时间一到，在clock_event_device的事件回调函数中处理到期的hrtimer。现在你或许有疑问：前面在介绍clock_event_device时，我们知道，每个cpu有自己的tick_device，通常用于周期性地产生进程调度和时间统计的tick事件，这里又说要用tick_device调度hrtimer系统，通常cpu只有一个tick_device，那他们如何协调工作？这个问题也一度困扰着我，如果再加上NO_HZ配置带来tickless特性，你可能会更晕。这里我们先把这个疑问放下，我将在后面的章节中来讨论这个问题，现在我们只要先知道，一旦开启了hrtimer，tick_device所关联的clock_event_device的事件回调函数会被修改为：hrtimer_interrupt，并且会被设置成工作于CLOCK_EVT_MODE_ONESHOT单触发模式。

2.1 添加一个hrtimer

要添加一个hrtimer，系统提供了一些api供我们使用，首先我们需要定义一个hrtimer结构的实例，然后用hrtimer_init函数对它进行初始化，它的原型如下：

```
[cpp]
01. void hrtimer_init(struct hrtimer *timer, clockid_t which_clock,
02.                  enum hrtimer_mode mode);
```

which_clock可以是CLOCK_REALTIME、CLOCK_MONOTONIC、CLOCK_BOOTTIME中的一种，mode则可以是相对时间HRTIMER_MODE_REL，也可以是绝对时间HRTIMER_MODE_ABS。设定回调函数：

```
[cpp]
01. timer.function = hr_callback;
```

如果定时器无需指定一个到期范围，可以在设定回调函数后直接使用hrtimer_start激活该定时器：

```
[cpp]
01. int hrtimer_start(struct hrtimer *timer, ktime_t tim,
02.                  const enum hrtimer_mode mode);
```

如果需要指定到期范围，则可以使用hrtimer_start_range_ns激活定时器：

```
[cpp]
01. hrtimer_start_range_ns(struct hrtimer *timer, ktime_t tim,
02.                        unsigned long range_ns, const enum hrtimer_mode mode);
```

要取消一个hrtimer，使用hrtimer_cancel：

```
[cpp]
01. int hrtimer_cancel(struct hrtimer *timer);
```

以下两个函数用于推后定时器的到期时间：

```
[cpp]
01. extern u64
02. hrtimer_forward(struct hrtimer *timer, ktime_t now, ktime_t interval);
03.
04. /* Forward a hrtimer so it expires after the hrtimer's current now */
05. static inline u64 hrtimer_forward_now(struct hrtimer *timer,
06.                                       ktime_t interval)
07. {
08.     return hrtimer_forward(timer, timer->base->get_time(), interval);
09. }
```

以下几个函数用于获取定时器的当前状态：

```
[cpp]
01. static inline int hrtimer_active(const struct hrtimer *timer)
02. {
03.     return timer->state != HRTIMER_STATE_INACTIVE;
04. }
05.
06. static inline int hrtimer_is_queued(struct hrtimer *timer)
07. {
08.     return timer->state & HRTIMER_STATE_ENQUEUED;
09. }
10.
11. static inline int hrtimer_callback_running(struct hrtimer *timer)
12. {
13.     return timer->state & HRTIMER_STATE_CALLBACK;
14. }
```

hrtimer_init最终会进入__hrtimer_init函数，该函数的主要目的是初始化hrtimer的base字段，同时初始化作为红黑树的节点的node字段：

```
[cpp]
01. static void __hrtimer_init(struct hrtimer *timer, clockid_t clock_id,
02.                          enum hrtimer_mode mode)
03. {
04.     struct hrtimer_cpu_base *cpu_base;
05.     int base;
06.
07.     memset(timer, 0, sizeof(struct hrtimer));
08.
09.     cpu_base = &__raw_get_cpu_var(hrtimer_bases);
10.
11.     if (clock_id == CLOCK_REALTIME && mode != HRTIMER_MODE_ABS)
12.         clock_id = CLOCK_MONOTONIC;
13.
14.     base = hrtimer_clockid_to_base(clock_id);
15.     timer->base = &cpu_base->clock_base[base];
16.     timerqueue_init(&timer->node);
17.     .....
18. }
```

hrtimer_start和hrtimer_start_range_ns最终会把实际的工作交由__hrtimer_start_range_ns来完成：

```
[cpp]
01. int __hrtimer_start_range_ns(struct hrtimer *timer, ktime_t tim,
02.                             unsigned long delta_ns, const enum hrtimer_mode mode,
03.                             int wakeup)
04. {
05.     .....
06.     /* 取得hrtimer_clock_base指针 */
07.     base = lock_hrtimer_base(timer, &flags);
08.     /* 如果已经在红黑树中, 先移除它: */
09.     ret = remove_hrtimer(timer, base); .....
10.     /* 如果是相对时间, 则需要加上当前时间, 因为内部是使用绝对时间 */
11.     if (mode & HRTIMER_MODE_REL) {
12.         tim = ktime_add_safe(tim, new_base->get_time());
13.         .....
14.     }
15.     /* 设置到期的时间范围 */
16.     hrtimer_set_expires_range_ns(timer, tim, delta_ns);
17.     .....
18.     /* 把hrtime按到期时间排序, 加入到对应时间基准系统的红黑树中 */
19.     /* 如果该定时器的是最早到期的, 将会返回true */
20.     leftmost = enqueue_hrtimer(timer, new_base);
21.     /*
22.      * Only allow reprogramming if the new base is on this CPU.
23.      * (it might still be on another CPU if the timer was pending)
24.      *
25.      * XXX send_remote_softirq() ?
26.      * 定时器比之前的到期时间要早, 所以需要重新对tick_device进行编程, 重新设置的到期时间
27.      */
28.     if (leftmost && new_base->cpu_base == &__get_cpu_var(hrtimer_bases))
29.         hrtimer_enqueue_reprogram(timer, new_base, wakeup);
30.     unlock_hrtimer_base(timer, &flags);
31.     return ret;
32. }
33. <p>
34. </p>
```

2.2 hrtimer的到期处理

高精度定时器系统有3个入口可以对到期定时器进行处理，它们分别是：

- 没有切换到高精度模式时，在每个jiffie的tick事件中中断中进行查询和处理；
- 在HRTIMER_SOFTIRQ软中断中进行查询和处理；
- 切换到高精度模式后，在每个clock_event_device的到期事件中中断中进行查询和处理；

低精度模式 因为系统并不是一开始就会支持高精度模式，而是在系统启动后的某个阶段，等待所有的条件都满足后，才会切换到高精度模式，当系统还没有切换到高精度模式时，所有的高精度定时器运行在低精度模式下，在每

一个jiffie的tick事件中中断中进行到期定时器的查询和处理，显然这时候的精度和低分辨率定时器是一样的（HZ级别）。低精度模式下，每个tick事件中中断中，hrtimer_run_queues函数会被调用，由它完成定时器的到期处理。hrtimer_run_queues首先判断目前高精度模式是否已经启用，如果已经切换到了高精度模式，什么也不做，直接返回：

```
[cpp]
01. void hrtimer_run_queues(void)
02. {
03.
04.     if (hrtimer_hres_active())
05.         return;
```

如果hrtimer_hres_active返回false，说明目前处于低精度模式下，则继续处理，它用一个for循环遍历各个时间基准系统，查询每个hrtimer_clock_base对应红黑树的左下节点，判断它的时间是否到期，如果到期，通过__run_hrtimer函数，对到期定时器进行处理，包括：调用定时器的回调函数、从红黑树中移除该定时器、根据回调函数的返回值决定是否重新启动该定时器等等：

```
[cpp]
01. for (index = 0; index < HRTIMER_MAX_CLOCK_BASES; index++) {
02.     base = &cpu_base->clock_base[index];
03.     if (!timerqueue_getnext(&base->active))
04.         continue;
05.
06.     if (gettime) {
07.         hrtimer_get_softirq_time(cpu_base);
08.         gettime = 0;
09.     }
10.
11.     raw_spin_lock(&cpu_base->lock);
12.
13.     while ((node = timerqueue_getnext(&base->active))) {
14.         struct hrtimer *timer;
15.
16.         timer = container_of(node, struct hrtimer, node);
17.         if (base->softirq_time.tv64 <=
18.             hrtimer_get_expires_tv64(timer))
19.             break;
20.
21.         __run_hrtimer(timer, &base->softirq_time);
22.     }
23.     raw_spin_unlock(&cpu_base->lock);
24. }
```

上面的timerqueue_getnext函数返回红黑树中的左下节点，之所以可以在while循环中使用该函数，是因为__run_hrtimer会在移除旧的左下节点时，新的左下节点会被更新到base->active->next字段中，使得循环可以继续执行，直到没有新的到期定时器为止。

高精度模式 切换到高精度模式后，原来给cpu提供tick事件的tick_device（clock_event_device）会被高精度定时器系统接管，它的中断事件回调函数被设置为hrtimer_interrupt，红黑树中最左下的节点的定时器的到期时间被编程到该clock_event_device中，这样每次clock_event_device的中断意味着至少有一个高精度定时器到期。另外，当timekeeper系统中的时间需要修正，或者clock_event_device的到期事件时间被重新编程时，系统会发出HRTIMER_SOFTIRQ软中断，软中断的处理函数run_hrtimer_softirq最终也会调用hrtimer_interrupt函数对到期定时器进行处理，所以在这里我们只要讨论hrtimer_interrupt函数的实现即可。

hrtimer_interrupt函数的前半部分和低精度模式下的hrtimer_run_queues函数完成相同的事情：它用一个for循环遍历各个时间基准系统，查询每个hrtimer_clock_base对应红黑树的左下节点，判断它的时间是否到期，如果到期，通过__run_hrtimer函数，对到期定时器进行处理，所以我们只讨论后半部分，在处理完所有到期定时器后，下一个到期定时器的到期时间保存在变量expires_next中，接下来的工作就是把这个到期时间编程到tick_device中：

```
[cpp]
01. void hrtimer_interrupt(struct clock_event_device *dev)
02. {
03.     .....
04.     for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++) {
05.         .....
```

```

06.         while ((node = timerqueue_getnext(&base->active))) {
07.             .....
08.             if (basenow.tv64 < hrtimer_get_software_expires_tv64(timer)) {
09.                 ktime_t expires;
10.
11.                 expires = ktime_sub(hrtimer_get_expires(timer),
12.                                     base->offset);
13.                 if (expires.tv64 < expires_next.tv64)
14.                     expires_next = expires;
15.                 break;
16.             }
17.
18.             __run_hrtimer(timer, &basenow);
19.         }
20.     }
21.
22.     /*
23.      * Store the new expiry value so the migration code can verify
24.      * against it.
25.      */
26.     cpu_base->expires_next = expires_next;
27.     raw_spin_unlock(&cpu_base->lock);
28.
29.     /* Reprogramming necessary ? */
30.     if (expires_next.tv64 == KTIME_MAX ||
31.         !tick_program_event(expires_next, 0)) {
32.         cpu_base->hang_detected = 0;
33.         return;
34.     }

```

如果这时的tick_program_event返回了非0值，表示过期时间已经在当前时间的前面，这通常由以下原因造成：

- 系统正在被调试跟踪，导致时间在走，程序不走；
- 定时器的回调函数花了太长的时间；
- 系统运行在虚拟机中，而虚拟机被调度导致停止运行；

为了避免这些情况的发生，接下来系统提供3次机会，重新执行前面的循环，处理到期的定时器：

```

[cpp]
01. raw_spin_lock(&cpu_base->lock);
02. now = hrtimer_update_base(cpu_base);
03. cpu_base->nr_retries++;
04. if (++retries < 3)
05.     goto retry;

```

如果3次循环后还无法完成到期处理，系统不再循环，转为计算本次总循环的时间，然后把tick_device的到期时间强制设置为当前时间加上本次的总循环时间，不过推后的时间被限制在100ms以内：

```

[cpp]
01. delta = ktime_sub(now, entry_time);
02. if (delta.tv64 > cpu_base->max_hang_time.tv64)
03.     cpu_base->max_hang_time = delta;
04. /*
05.  * Limit it to a sensible value as we enforce a longer
06.  * delay. Give the CPU at least 100ms to catch up.
07.  */
08. if (delta.tv64 > 100 * NSEC_PER_MSEC)
09.     expires_next = ktime_add_ns(now, 100 * NSEC_PER_MSEC);
10. else
11.     expires_next = ktime_add(now, delta);
12. tick_program_event(expires_next, 1);
13. printk_once(KERN_WARNING "hrtimer: interrupt took %llu ns\n",
14.             ktime_to_ns(delta));
15. }

```

3. 切换到高精度模式

上面提到，尽管内核配置成支持高精度定时器，但并不是一开始就工作于高精度模式，系统在启动的开始阶段，还是按照传统的模式在运行：tick_device按HZ频率定期地产生tick事件，这时的hrtimer工作在低分辨率模式，到期事件在每个tick事件中中断由hrtimer_run_queues函数处理，同时，在低分辨率定时器（时间轮）的软件中断

TIMER_SOFTIRQ中，hrtimer_run_pending会被调用，系统在这个函数中判断系统的条件是否满足切换到高精度模式，如果条件满足，则会切换至高分辨率模式，另外提一下，NO_HZ模式也是在该函数中判断并切换。

```
[cpp]
01. void hrtimer_run_pending(void)
02. {
03.     if (hrtimer_hres_active())
04.         return;
05.     .....
06.     if (tick_check_oneshot_change(!hrtimer_is_hres_enabled()))
07.         hrtimer_switch_to_hres();
08. }
```

因为不管系统是否工作于高精度模式，每个TIMER_SOFTIRQ期间，该函数都会被调用，所以函数一开始先用hrtimer_hres_active判断目前高精度模式是否已经激活，如果已经激活，则说明之前的调用中已经切换了工作模式，不必再次切换，直接返回。hrtimer_hres_active很简单：

```
[cpp]
01. DEFINE_PER_CPU(struct hrtimer_cpu_base, hrtimer_bases) = {
02.     .....
03. }
04.
05. static inline int hrtimer_hres_active(void)
06. {
07.     return __this_cpu_read(hrtimer_bases.hres_active);
08. }
```

hrtimer_run_pending函数接着通过tick_check_oneshot_change判断系统是否可以切换到高精度模式，

```
[cpp]
01. int tick_check_oneshot_change(int allow_nohz)
02. {
03.     struct tick_sched *ts = &__get_cpu_var(tick_cpu_sched);
04.
05.     if (!test_and_clear_bit(0, &ts->check_clocks))
06.         return 0;
07.
08.     if (ts->nohz_mode != NOHZ_MODE_INACTIVE)
09.         return 0;
10.
11.     if (!timekeeping_valid_for_hres() || !tick_is_oneshot_available())
12.         return 0;
13.
14.     if (!allow_nohz)
15.         return 1;
16.
17.     tick_nohz_switch_to_nohz();
18.     return 0;
19. }
```

函数的一开始先判断check_clock标志的第0位是否被置位，如果没有置位，说明系统中没有注册符合要求的时钟事件设备，函数直接返回，check_clock标志由clocksource和clock_event_device系统的notify系统置位，当系统中有更高精度的clocksource被注册和选择后，或者有更精确的支持CLOCK_EVT_MODE_ONESHOT模式的clock_event_device被注册时，通过它们的notify函数，check_clock标志的第0位会置位。

如果tick_sched结构中的nohz_mode字段不是NOHZ_MODE_INACTIVE，表明系统已经切换到其它模式，直接返回。nohz_mode的取值有3种：

- NOHZ_MODE_INACTIVE // 未启用NO_HZ模式
- NOHZ_MODE_LOWRES // 启用NO_HZ模式，hrtimer工作于低精度模式下
- NOHZ_MODE_HIGHRES // 启用NO_HZ模式，hrtimer工作于高精度模式下

接下来的timekeeping_valid_for_hres判断timekeeper系统是否支持高精度模式，tick_is_oneshot_available判断tick_device是否支持CLOCK_EVT_MODE_ONESHOT模式。如果都满足要求，则继续往下判断。allow_nohz是函数的参数，为true表明可以切换到NOHZ_MODE_LOWRES模式，函数将进入tick_nohz_switch_to_nohz，切换至NOHZ_MODE_LOWRES模式，这里我们传入的allow_nohz是表达式：


```
(!hrtimer_is_hres_enabled())
```

所以当系统不允许高精度模式时，将会在tick_check_oneshot_change函数内，通过tick_nohz_switch_to_nohz切换至NOHZ_MODE_LOWRES 模式，如果系统允许高精度模式，传入的allow_nohz参数为false，tick_check_oneshot_change函数返回1，回到上面的hrtimer_run_pending函数，hrtimer_switch_to_hres函数将会被调用，已完成切换到NOHZ_MODE_HIGHRES高精度模式。好啦，真正的切换函数找到了，我们看一看它如何切换：

首先，它通过hrtimer_cpu_base中的hres_active字段判断该cpu是否已经切换至高精度模式，如果是则直接返回：

```
[cpp]
01. static int hrtimer_switch_to_hres(void)
02. {
03.     int i, cpu = smp_processor_id();
04.     struct hrtimer_cpu_base *base = &per_cpu(hrtimer_bases, cpu);
05.     unsigned long flags;
06.
07.     if (base->hres_active)
08.         return 1;
```

接着，通过tick_init_highres函数接管tick_device关联的clock_event_device：

```
[cpp]
01. local_irq_save(flags);
02.
03. if (tick_init_highres()) {
04.     local_irq_restore(flags);
05.     printk(KERN_WARNING "Could not switch to high resolution "
06.             "mode on CPU %d\n", cpu);
07.     return 0;
08. }
```

tick_init_highres函数把tick_device切换到CLOCK_EVT_FEAT_ONESHOT模式，同时把clock_event_device的回调handler设置为hrtimer_interrupt，这样设置以后，tick_device的中断回调将由hrtimer_interrupt接管，hrtimer_interrupt在上面已经讨论过，它将完成高精度定时器的调度和到期处理。

接着，设置hres_active标志，以表明高精度模式已经切换，然后把3个时间基准系统的resolution字段设为KTIME_HIGH_RES：

```
[cpp]
01. base->hres_active = 1;
02. for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++)
03.     base->clock_base[i].resolution = KTIME_HIGH_RES;
```

最后，因为tick_device被高精度定时器接管，它将不会再提供原有的tick事件机制，所以需要由高精度定时器系统模拟一个tick事件设备，继续为系统提供tick事件能力，这个工作由tick_setup_sched_timer函数完成。因为刚刚完成切换，tick_device的到期时间并没有被正确地设置为下一个到期定时器的时间，这里使用retrigger_next_event函数，传入参数NULL，使得tick_device立刻产生到期中断，hrtimer_interrupt被调用一次，然后下一个到期的定时器的时间会编程到tick_device中，从而完成了到高精度模式的切换：

```
[cpp]
01. tick_setup_sched_timer();
02. /* "Retrigger" the interrupt to get things going */
03. retrigger_next_event(NULL);
04. local_irq_restore(flags);
05. return 1;
```

整个切换过程可以用下图表示：

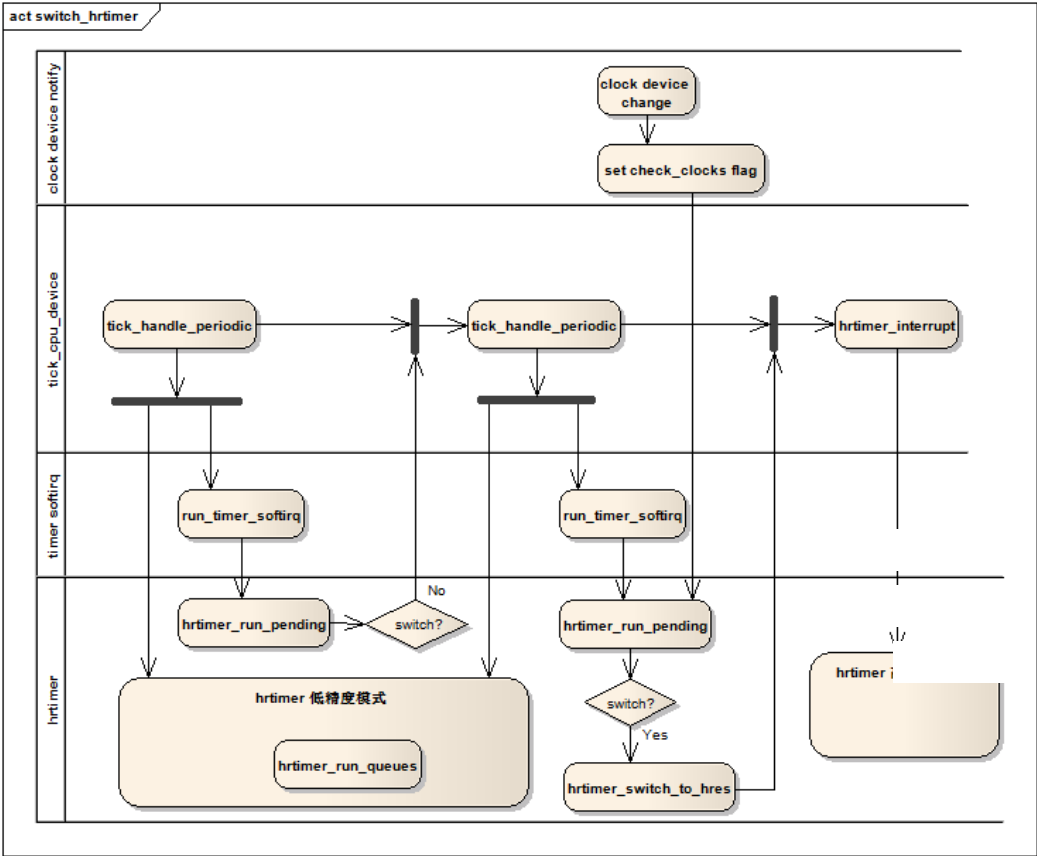


图3.1 低精度模式切换至高精度模式

4. 模拟tick事件

根据上一节的讨论，当系统切换到高精度模式后，tick_device被高精度定时器系统接管，不再定期地产生tick事件，我们知道，到目前的版本为止（V3.4），内核还没有彻底废除jiffies机制，系统还是依赖定期到来的tick事件，供进程调度系统和时间更新等操作，大量存在的低精度定时器也仍然依赖于jiffies的计数，所以，尽管tick_device被接管，高精度定时器系统还是要想办法继续提供定期的tick事件。为了达到这一目的，内核使用了一个取巧的办法：既然高精度模式已经启用，可以定义一个hrtimer，把它的到期时间设定为一个jiffy的时间，当这个hrtimer到期时，在这个hrtimer的到期回调函数中，进行和原来的tick_device同样的操作，然后把该hrtimer的到期时间顺延一个jiffy周期，如此反复循环，完美地模拟了原有tick_device的功能。下面我们看看具体点代码是如何实现的。

在kernel/time/tick-sched.c中，内核定义了一个per_cpu全局变量：tick_cpu_sched，从而为每个cpu提供了一个tick_sched结构，该结构主要用于管理NO_HZ配置下的tickless处理，因为模拟tick事件与tickless有很强的相关性，所以高精度定时器系统也利用了该结构的以下字段，用于完成模拟tick事件的操作：

```
[cpp]
01. struct tick_sched {
02.     struct hrtimer      sched_timer;
03.     unsigned long       check_clocks;
04.     enum tick_nohz_mode nohz_mode;
05.     .....
06. };
```

sched_timer就是要用于模拟tick事件的hrtimer，check_clock上面几节已经讨论过，用于notify系统通知hrtimer系统需要检查是否切换到高精度模式，nohz_mode则用于表示当前的工作模式。

上一节提到，用于切换至高精度模式的函数是hrtimer_switch_to_hres，在它的最后，调用了函数tick_setup_sched_timer，该函数的作用就是设置一个用于模拟tick事件的hrtimer：

```
[cpp]
01. void tick_setup_sched_timer(void)
02. {
03.     struct tick_sched *ts = &__get_cpu_var(tick_cpu_sched);
04.     ktime_t now = ktime_get();
```

```

05.
06.     /*
07.      * Emulate tick processing via per-CPU hrtimers:
08.      */
09.     hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
10.     ts->sched_timer.function = tick_sched_timer;
11.
12.     /* Get the next period (per cpu) */
13.     hrtimer_set_expires(&ts->sched_timer, tick_init_jiffy_update());
14.
15.     for (;;) {
16.         hrtimer_forward(&ts->sched_timer, now, tick_period);
17.         hrtimer_start_expires(&ts->sched_timer,
18.                               HRTIMER_MODE_ABS_PINNED);
19.         /* Check, if the timer was already in the past */
20.         if (hrtimer_active(&ts->sched_timer))
21.             break;
22.         now = ktime_get();
23.     }
24.
25. #ifdef CONFIG_NO_HZ
26.     if (tick_nohz_enabled)
27.         ts->nohz_mode = NOHZ_MODE_HIGHRES;
28. #endif
29. }

```

该函数首先初始化该cpu所属的tick_sched结构中sched_timer字段，把该hrtimer的回调函数设置为tick_sched_timer，然后把它的到期时间设定为下一个jiffy时刻，返回前把工作模式设置为NOHZ_MODE_HIGHRES，表明是利用高精度模式实现NO_HZ。

接着我们关注一下hrtimer的回调函数tick_sched_timer，我们知道，系统中的jiffies计数，时间更新等是全局操作，在smp系统中，只有一个cpu负责该工作，所以在tick_sched_timer的一开始，先判断当前cpu是否负责更新jiffies和时间，如果是，则执行更新操作：

```

[cpp]
01. static enum hrtimer_restart tick_sched_timer(struct hrtimer *timer)
02. {
03.     .....
04.
05. #ifdef CONFIG_NO_HZ
06.     if (unlikely(tick_do_timer_cpu == TICK_DO_TIMER_NONE))
07.         tick_do_timer_cpu = cpu;
08. #endif
09.
10.     /* Check, if the jiffies need an update */
11.     if (tick_do_timer_cpu == cpu)
12.         tick_do_update_jiffies64(now);

```

然后，利用regs指针确保当前是在中断上下文中，然后调用update_process_timer：

```

[cpp]
01. if (regs) {
02.     .....
03.     update_process_times(user_mode(regs));
04.     .....
05. }

```

最后，把hrtimer的到期时间推进一个tick周期，返回HRTIMER_RESTART表明该hrtimer需要再次启动，以便产生下一个tick事件。

```

[cpp]
01. hrtimer_forward(timer, now, tick_period);
02.
03. return HRTIMER_RESTART;
04. }

```

关于update_process_times，如果你感兴趣，回看一下本系列关于clock_event_device的那一章：[Linux时间子系统之四：定时器的引擎：clock_event_device](#)中的第5小节，对比一下模拟tick事件的hrtimer的回调函数

tick_sched_timer和切换前tick_device的回调函数tick_handle_periodic，它们是如此地相像，实际上，它们几乎完成了一样的工作。

顶

1

踩

0

上一篇Linux时间子系统之五：低分辨率定时器的原理和实现

下一篇Linux时间子系统之七：定时器的应用--msleep()，hrtimer_nanosleep()

我的同类文章

Linux内核架构（14）

Linux时间管理系统（7）

Linux动态频率调节系统CPU...2013-07-20阅读13826

Linux时间子系统之八：动态...2012-10-27阅读17748

Linux时间子系统之五：低分...2012-10-13阅读13912

Linux时间子系统之三：时间...2012-09-19阅读22238

Linux时间子系统之一：cloc...2012-09-13阅读23642

Linux动态频率调节系统CPU...2013-07-17阅读140040

Linux时间子系统之七：定时...2012-10-23阅读17462

Linux时间子系统之四：定时...2012-09-28阅读16811

Linux时间子系统之二：表示...2012-09-14阅读20867

Linux中断（interrupt）子系...2012-05-01阅读25573

更多文章

参考知识库



.NET知识库

3774 关注 | 833 收录



Linux知识库

11722 关注 | 3939 收录



算法与数据结构知识库

15817 关注 | 2320 收录



大型网站架构知识库


8578 关注 | 708 收录

猜你在找

- 话说linux内核-uboot和系统移植第14部分
- linux下jiffies定时器和hrtimer高精度定时器
- 嵌入式软件调试技术专题(3)：Linux内核日志与信息打
- Linux时间子系统之七定时器的应用--
- 从零写Bootloader及移植uboot、linux内核、文件系统
- Linux时间子系统之七定时器的应用--
- 嵌入式Linux内核移植
- Linux时间子系统之七定时器的应用--
- 嵌入式Linux系统移植入门
- 彻底实现Linux TCP的Pacing发送逻辑-高精度hrtimer版


查看评论

14楼 [_圆的_](#) 2016-08-25 15:39发表



博主,你文中说到hrtimer的到期时间可以基于以下几种时间基准系统：
HRTIMER_BASE_MONOTONIC,HRTIMER_BASE_REALTIME,HRTIMER_BASE_BOOTTIME,HRTIMER_MAX_CLOCK_BASE,
可是在初始化的时候 which_clock可以是CLOCK_REALTIME、CLOCK_MONOTONIC、CLOCK_BOOTTIME中的一种,并没有
用到上面几种时间基准系统阿,求解!!!

13楼 [Kevin_Smart](#) 2016-06-17 11:37发表



引用“DroidPhone”的评论：
回复Ikingz：单看这一篇可能不好理解，要把这个系列的几篇文章联合起来看...

像楼主说的，原则上，hrtimer是利用一个硬件计数器来实现的，所以精度才可以做到ns级别。硬件的计数器要达到ns级，时钟频率需达到GHZ，这是cpu内部支持还是需要外围器件？达到GHZ的cpu在嵌入式上还是比较少的吧

12楼 [mayorlsx](#) 2015-09-16 16:01发表



请问，如何判断系统所带内核支持高精度定时器（我用的测试机为centos7，内核3.10.0-229.el7.x86_64，处理器 intel(R) Core(TM)i5-4570 CPU 3.2GHZ）？用户空间的应用程序如何使用hrtimer高精度定时器？有没有高精度的例子，急用。我的邮箱:mayorlsx@163.com。多谢！

11楼 [No威_](#) 2015-07-22 17:24发表



厉害

10楼 [dinitial](#) 2015-05-29 17:40发表



楼主，请问你的第二张图用什么工具画的？

9楼 [superbool](#) 2014-10-21 11:19发表



问一下，tick_check_oneshot_change，把hrtimer_is_hres_enabled()传给形参allow_nohz，是不是意味着hrtimer和nohz是互斥的，开了hrtimer就不能开nohz？

如果是这样，那为什么nohz_mode的取值有3种：

- NOHZ_MODE_INACTIVE // 未启用NO_HZ模式
- NOHZ_MODE_LOWRES // 启用NO_HZ模式，hrtimer工作于低精度模式下
- NOHZ_MODE_HIGHRES // 启用NO_HZ模式，hrtimer工作于高精度模式下？？

非常感谢

Re: [puppypyb](#) 2014-11-25 12:07发表



回复superbool：你在文中搜索这一句：“返回前把工作模式设置为NOHZ_MODE_HIGHRES，表明是利用高精度模式实现NO_HZ”若kernel使能了hrtimer和NO_HZ模式，那么进入到hrtimer的同时也会开启NO_HZ模式，也就是一口气切换到NOHZ_MODE_HIGHRES。具体是在tick_setup_sched_timer函数中完成的。

Re: [puppypyb](#) 2014-11-25 16:18发表



回复puppypyb：没说全面，比如还有一种情况是开始没有使能hrtimer，先进入到NOHZ_MODE_LOWRES模式。后面通过setup_hrtimer_hres使能hrtimer，就可以通过调用路径“tick_nohz_handler-->update_process_times-->run_local_timers-->run_timer_softirq-->hrtimer_run_pending-->hrtimer_switch_to_hres”切换到NOHZ_MODE_HIGHRES模式。

8楼 [决战北京城](#) 2014-08-12 15:20发表



引用“DroidPhone”的评论：

回复obanaganastar：嗯，你用低精度当然到不了us级别啦，按照2...

如果和滴答有关系，就不需要使用到hrtimer了吧。如果要精确到us级别是否要使用高精度的模式。我不能完全理解博主的文章，组织不起来。我看网上例子的高精度其实是hrtimer的低精度模式，博主有没有高精度的例子，也比较急用。我的邮箱是:ouhuayu@126.com。谢谢了~

Re: [DroidPhone](#) 2014-08-12 22:21发表



回复决战北京城：不知道你是希望在内核代码下还是在用户空间的应用程序使用高精度定时器？不过你确定你的系统所带的内核已经支持高精度定时器才行。

Re: [决战北京城](#) 2014-08-13 22:00发表



回复DroidPhone：我的用的是4核exynos4412 主频1.5GHZ，理论值达得到的。我写的是驱动，想在内核态实现，我想在板子上加个ds18b20。谢谢lz

Re: [DroidPhone](#) 2014-08-13 23:24发表



回复决战北京城：嗯，内核驱动要使用高精度定时器，那就按照2.1节的内容照做应该可以了。前提是你的内核必须配置了高精度时钟的相关内核配置项。

7楼 [决战北京城](#) 2014-08-12 12:11发表



我试了低精度的发现还是不能达到us级别，误差很大，如果是滴答。我机器是HZ=200

Re: [DroidPhone](#) 2014-08-12 12:37发表



回复决战北京城：嗯，你用低精度当然到不了us级别啦，按照200HZ来算，精度也就1/200秒，相当于5ms。


6楼 [Ikingz](#) 2014-07-31 17:12发表



看不太懂。。。


问下，linux系统中的us甚至ns级别的定时或者延时最终都是通过硬件实现守时的吗，比如usleep及select文式延时

Re: [DroidPhone](#) 2014-08-01 17:51发表




回复lkingz：单看这一篇可能不好理解，要把这个系列的几篇文章联合起来看可能更容易理解。原则上，hrtimer是利用一个硬件计数器来实现的，所以精度才可以做到ns级别。

5楼 [yazhouren](#) 2014-06-24 14:00发表



必须看kernel/Document/timers下面的文档才能明白呀

Re: [DroidPhone](#) 2014-06-26 13:41发表



回复yazhouren：嗯，看Documents／目录下对文档是对理解kernel对各个子系统和驱动是有很大帮助的。

4楼 [_海哲](#) 2014-02-25 13:48发表




xuexile

3楼 [_海哲](#) 2014-02-25 13:48发表



xuexile

2楼 [ryan12345678](#) 2013-11-05 14:22发表



我的高精度定时器最低只能到5毫秒, 请问是内核配置问题吗？

1楼 [apprentice89](#) 2013-09-06 16:58发表



赞！

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

[全部主题](#) [Hadoop](#) [AWS](#) [移动游戏](#) [Java](#) [Android](#) [iOS](#) [Swift](#) [智能硬件](#) [Docker](#) [OpenStack](#)
[VPN](#) [Spark](#) [ERP](#) [IE10](#) [Eclipse](#) [CRM](#) [JavaScript](#) [数据库](#) [Ubuntu](#) [NFC](#) [WAP](#) [jQuery](#)
[BI](#) [HTML5](#) [Spring](#) [Apache](#) [.NET](#) [API](#) [HTML](#) [SDK](#) [IIS](#) [Fedora](#) [XML](#) [LBS](#) [Unity](#)
[Splashtop](#) [UML](#) [components](#) [Windows Mobile](#) [Rails](#) [QEMU](#) [KDE](#) [Cassandra](#) [CloudStack](#)
[FTC](#) [coremail](#) [OPhone](#) [CouchBase](#) [云计算](#) [iOS6](#) [Rackspace](#) [Web App](#) [SpringSide](#) [Maemo](#)
[Compuware](#) [大数据](#) [aptech](#) [Perl](#) [Tornado](#) [Ruby](#) [Hibernate](#) [ThinkPHP](#) [HBase](#) [Pure](#) [Solr](#)
[Angular](#) [Cloud Foundry](#) [Redis](#) [Scala](#) [Django](#) [Bootstrap](#)

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved 