个人资料

DroidPhone

访问： 1077200次
积分： 8768
等级： BLOG 6
排名： 第1448名

原创： 51篇 转载： 0篇
译文： 4篇 评论： 537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声卡驱动之一 (74187)
Android Audio System 之 (58758)
Linux ALSA声卡驱动之二 (46387)
Android Audio System 之 (42720)
Linux ALSA声卡驱动之三 (41553)
Linux ALSA声卡驱动之六

ALSA声卡驱动中的DAPM详解之七：dapm事件机制（dapm event）

标签： linux audio driver widget alsa dapm

2013-11-09 01:05 8980人阅读 评论(7) 收藏 举报

分类： Linux音频子系统（14）

目录(?) [+]

前面的六篇文章，我们已经讨论了dapm关于动态电源管理的有关知识，包括widget的创建和初始化，widget之间的连接以及widget的上下电顺序等等。本章我们准备讨论dapm框架中的另一个机制：事件机制。通过dapm事件机制，widget可以对它所关心的dapm事件做出反应，这种机制对于扩充widget的能力非常有用，例如，对于那些位于codec之外的widget，好像喇叭功放、外部的前置放大器等等，由于不是使用codec内部的寄存器进行电源控制，我们就必须利用dapm的事件机制，获得相应的上下电事件，从而可以定制widget自身的电源控制功能。

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

dapm event的种类

dapm目前为我们定义了9种dapm event，他们分别是：

事件类型	说明
SND_SOC_DAPM_PRE_PMU	widget要上电前发出的事件
SND_SOC_DAPM_POST_PMU	widget要上电后发出的事件
SND_SOC_DAPM_PRE_PMD	widget要下电前发出的事件
SND_SOC_DAPM_POST_PMD	widget要下电后发出的事件
SND_SOC_DAPM_PRE_REG	音频路径设置之前发出的事件
SND_SOC_DAPM_POST_REG	音频路径设置之后发出的事件
SND_SOC_DAPM_WILL_PMU	在处理up_list链表之前发出的事件
SND_SOC_DAPM_WILL_PMD	在处理down_list链表之前发出的事件
SND_SOC_DAPM_PRE_POST_PMD	SND_SOC_DAPM_PRE_PMD和SND_SOC_DAPM_POST_PMD的合并

前8种每种占据一个位，所以，我们可以在一个整数中表达多个我们需要关心的dapm事件，只要把它们按位或进行合并即可。

widget的event回调函数

ALSA声卡驱动中的DAPM详解之二：widget-具备路径和电源管理信息的kcontrol中，我们已经介绍过代表widget的snd_soc_widget结构，在这个结构体中，有一个event字段用于保存该widget的事件回调函数，同时，event_flags字段用于保存该widget需要关心的dapm事件种类，只有event_flags字段中相应的事件位被设置了的事件才会发到event回调函数中进行处理。

Linux时间子系统之六：高 (37505)
Android Audio System 之 (36411)
Linux ALSA声卡驱动之七 (36317)
Linux ALSA声卡驱动之四 (36146)
(31714)

评论排行

Android Audio System 之 (56)
Linux ALSA声卡驱动之三 (42)
Linux ALSA声卡驱动之八 (35)
Linux时间子系统之六：高 (25)
Linux中断（interrupt）子 (24)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (19)
Android Audio System 之 (18)
Linux ALSA声卡驱动之七 (17)
Linux中断（interrupt）子 (17)

推荐文章

* 致JavaScript也将征服的物联网世界
* 从苏宁电器到卡巴斯基：难忘的三年硕士时光
* 作为一名基层管理者如何利用情商管理自己和团队（一）
* Android CircleImageView圆形ImageView
* 高质量代码的命名法则

最新评论

Linux输入子系统：输入设备编程
u012839187: 看着没有问题。你要弄清楚的是新的内核有许多函数宏的变动。
Linux时间子系统之一：clock sou
zhqh100: 1.2 read回调函数时钟源本身不会产生中断，要获得时钟源的当前计数，只能通过主动调用它的rea...
Linux中断（interrupt）子系统之
liunix61: 大神！必须顶@！！！
Linux时间子系统之三：时间的维
Kevin_Smart: 学习了
Linux时间子系统之六：高精度定
Kevin_Smart: 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...
Linux中断（interrupt）子系统之
12期-马金兴: 恩，虽然这么多字但是我要好好学习一下
已知二叉树的前序遍历和中序遍
重修月: 很喜欢博主的文章，刚刚用豆约翰博客备份专家备份了您的全部博文。
Linux ALSA声卡驱动之三：PCM
灿哥哥: 学习了
Android Audio System 之三： A
ss0429: 楼主的文章写的很精炼，多谢分享~
Linux中断（interrupt）子系统之
KrisFei: 针对这句话有两个问题想讨论下：1. disable_irq()放在中断上半部有会导致死锁。2. 如果...

我们知道，dapm为我们提供了常用widget的定义辅助宏，使用以下这几种辅助宏定义widget时，默认需要我们提供dapm event回调函数

- SND_SOC_DAPM_MIC
- SND_SOC_DAPM_HP
- SND_SOC_DAPM_SPK
- SND_SOC_DAPM_LINE

这些widget都是位于codec外部的器件，它们无法使用通用的寄存器操作来控制widget的电源状态，所以需要我们提供event回调函数。以下的例子来自dapm的内核文档，外部的喇叭功放通过CORGI_GPIO_APM_ON这个gpio来控制它的电源状态：

```cpp
/* turn speaker amplifier on/off depending on use */
static int corgi_amp_event(struct snd_soc_dapm_widget *w, int event)
{
    gpio_set_value(CORGI_GPIO_APM_ON, SND_SOC_DAPM_EVENT_ON(event));
    return 0;
}

/* corgi machine dapm widgets */
static const struct snd_soc_dapm_widget wm8731_dapm_widgets =
    SND_SOC_DAPM_SPK("Ext Spk", corgi_amp_event);
```

另外，我们也可以通过以下这些带"_E"后缀的辅助宏版本来定义需要dapm事件的widget：

- SND_SOC_DAPM_PGA_E
- SND_SOC_DAPM_OUT_DRV_E
- SND_SOC_DAPM_MIXER_E
- SND_SOC_DAPM_MIXER_NAMED_CTL_E
- SND_SOC_DAPM_SWITCH_E
- SND_SOC_DAPM_MUX_E
- SND_SOC_DAPM_VIRT_MUX_E

触发dapm event

我们已经定义好了带有event回调的widget，那么，在那里触发这些dapm event？答案是：在dapm_power_widgets函数的处理过程中，dapm_power_widgets函数我们已经在ALSA声卡驱动中的DAPM详解之六：精髓所在，牵一发而动全身中做了详细的分析，其中，在所有需要处理电源变化的widget被分别放入up_list和down_list链表后，会相应地发出各种dapm事件：

```cpp
static int dapm_power_widgets(struct snd_soc_card *card, int event)
{
        ......
        list_for_each_entry(w, &down_list, power_list) {
                dapm_seq_check_event(card, w, SND_SOC_DAPM_WILL_PMD);
        }

        list_for_each_entry(w, &up_list, power_list) {
                dapm_seq_check_event(card, w, SND_SOC_DAPM_WILL_PMU);
        }

        /* Power down widgets first; try to avoid amplifying pops. */
        dapm_seq_run(card, &down_list, event, false);

        dapm_widget_update(card);

        /* Now power up. */
        dapm_seq_run(card, &up_list, event, true);
        ......
}
```

可见，在真正地进行上电和下电之前，dapm向down_list链表中的每个widget发出SND_SOC_DAPM_WILL_PMD事件，而向up_list链表中的每个widget发出SND_SOC_DAPM_WILL_PMU事件。在处理上下电的函数dapm_seq_run中，会调用dapm_seq_run_coalesced函数执行真正的寄存器操作，进行widget的电源控制，dapm_seq_run_coalesced也会发出另外几种dapm事件：

```cpp
static void dapm_seq_run_coalesced(struct snd_soc_card *card,
                                   struct list_head *pending)
{
        ......
        list_for_each_entry(w, pending, power_list) {
                ......
                /* Check for events */
                dapm_seq_check_event(card, w, SND_SOC_DAPM_PRE_PMU);
                dapm_seq_check_event(card, w, SND_SOC_DAPM_PRE_PMD);
        }

        if (reg >= 0) {
                ......
                pop_wait(card->pop_time);
                soc_widget_update_bits_locked(w, reg, mask, value);
        }

        list_for_each_entry(w, pending, power_list) {
                dapm_seq_check_event(card, w, SND_SOC_DAPM_POST_PMU);
                dapm_seq_check_event(card, w, SND_SOC_DAPM_POST_PMD);
        }
}
```

另外，负责更新音频路径的dapm_widget_update函数中也会发出dapm事件：

```cpp
static void dapm_widget_update(struct snd_soc_card *card)
{
        struct snd_soc_dapm_update *update = card->update;
        struct snd_soc_dapm_widget_list *wlist;
        struct snd_soc_dapm_widget *w = NULL;
        unsigned int wi;
        int ret;

        if (!update || !dapm_kcontrol_is_powered(update->kcontrol))
                return;

        wlist = dapm_kcontrol_get_wlist(update->kcontrol);

        for (wi = 0; wi < wlist->num_widgets; wi++) {
                w = wlist->widgets[wi];

                if (w->event && (w->event_flags & SND_SOC_DAPM_PRE_REG)) {
                        ret = w->event(w, update->kcontrol, SND_SOC_DAPM_PRE_REG);
                        ......
                }
        }

        ......
        /* 更新kcontrol的值，改变音频路径 */
        ret = soc_widget_update_bits_locked(w, update->reg, update->mask,
                                update->val);
        ......

        for (wi = 0; wi < wlist->num_widgets; wi++) {
                w = wlist->widgets[wi];

                if (w->event && (w->event_flags & SND_SOC_DAPM_POST_REG)) {
                        ret = w->event(w, update->kcontrol, SND_SOC_DAPM_POST_REG);
                        ......
                }
        }
}
```

可见，改变路径的前后，分别发出了SND_SOC_DAPM_PRE_REG事件和SND_SOC_DAPM_POST_REG事件。

dai widget与stream widget

dai widget 在ALSA声卡驱动中的DAPM详解之四：在驱动程序中初始化并注册widget和route一文中，我们已经讨论过dai widget，dai widget又分为cpu dai widget和codec dai widget，它们在machine驱动分别匹配上相应的codec和platform后，由soc_probe_platform和soc_probe_codec这两个函数通过调用dapm的api函数:

- snd_soc_dapm_new_dai_widgets

来创建的，通常会为playback和capture各自创建一个dai widget，他们的类型分别是：

- snd_soc_dapm_dai_in　对应playback dai
- snd_soc_dapm_dai_out　对应capture dai

另外，dai widget的名字是使用stream name来命名的，他通常来自snd_soc_dai_driver中的stream_name字段。dai widget的sname字段也使用同样的名字。

stream widget　stream widget通常是指那些要处理音频流数据的widget，它们包含以下这几种类型：

- snd_soc_dapm_aif_in　用SND_SOC_DAPM_AIF_IN辅助宏定义
- snd_soc_dapm_aif_out　用SND_SOC_DAPM_AIF_OUT辅助宏定义
- snd_soc_dapm_dac　用SND_SOC_DAPM_AIF_DAC辅助宏定义
- snd_soc_dapm_adc　用SND_SOC_DAPM_AIF_ADC辅助宏定义

对于这几种widget，我们除了要指定widget的名字外，还要指定他对应的stream的名字，保存在widget的sname字段中。

连接dai widget和stream widget

默认情况下，驱动不会通过snd_soc_route来主动定义dai widget和stream widget之间的连接关系，实际上，他们之间的连接关系是由ASoc负责的，在声卡的初始化函数中，使用snd_soc_dapm_link_dai_widgets函数来建立他们之间的连接关系：

```cpp
static int snd_soc_instantiate_card(struct snd_soc_card *card)
{
        ......
        /* card bind complete so register a sound card */
        ret = snd_card_create(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1,
                        card->owner, 0, &card->snd_card);
        ......
        if (card->dapm_widgets)
                snd_soc_dapm_new_controls(&card->dapm, card->dapm_widgets,
                                          card->num_dapm_widgets);
        /*  建立dai widget和stream widget之间的连接关系  */
        snd_soc_dapm_link_dai_widgets(card);
        ......
        if (card->controls)
                snd_soc_add_card_controls(card, card->controls, card->num_controls);
        ......
        if (card->dapm_routes)
                snd_soc_dapm_add_routes(&card->dapm, card->dapm_routes,
                                        card->num_dapm_routes);
        ......
        if (card->fully_routed)
                list_for_each_entry(codec, &card->codec_dev_list, card_list)
                        snd_soc_dapm_auto_nc_codec_pins(codec);

        snd_soc_dapm_new_widgets(card);

        ret = snd_card_register(card->snd_card);
        ......
        return 0;
}
```

我们再来分析一下snd_soc_dapm_link_dai_widgets函数，看看它是如何连接这两种widget的，它先是遍历声卡中所有的widget，找出类型为snd_soc_dapm_dai_in和snd_soc_dapm_dai_out的widget，通过widget的priv字段，取出widget对应的snd_soc_dai结构指针：

```cpp
int snd_soc_dapm_link_dai_widgets(struct snd_soc_card *card)
{
        struct snd_soc_dapm_widget *dai_w, *w;
        struct snd_soc_dai *dai;

        /* For each DAI widget... */
        list_for_each_entry(dai_w, &card->widgets, list) {
                switch (dai_w->id) {
                case snd_soc_dapm_dai_in:
                case snd_soc_dapm_dai_out:
                        break;
                default:
                        continue;
```

```
                }

                dai = dai_w->priv;
```

接着，再次从头遍历声卡中所有的widget，找出能与dai widget相连接的stream widget，第一个前提条件是这两个widget必须位于同一个dapm context中：

```cpp
/* ...find all widgets with the same stream and link them */
list_for_each_entry(w, &card->widgets, list) {
        if (w->dapm != dai_w->dapm)
                continue;
```

dai widget不会与dai widget相连，所以跳过它们：

```cpp
switch (w->id) {
case snd_soc_dapm_dai_in:
case snd_soc_dapm_dai_out:
        continue;
default:
        break;
}
```

dai widget的名字没有出现在要连接的widget的stream name中，跳过这个widget：

```cpp
if (!w->sname || !strstr(w->sname, dai_w->name))
        continue;
```

如果widget的stream name包含了dai的stream name，则匹配成功，连接这两个widget：

```cpp
                if (dai->driver->playback.stream_name &&
                    strstr(w->sname,
                           dai->driver->playback.stream_name)) {
                        dev_dbg(dai->dev, "%s -> %s\n",
                                dai->playback_widget->name, w->name);

                        snd_soc_dapm_add_path(w->dapm,
                                dai->playback_widget, w, NULL, NULL);
                }

                if (dai->driver->capture.stream_name &&
                    strstr(w->sname,
                           dai->driver->capture.stream_name)) {
                        dev_dbg(dai->dev, "%s -> %s\n",
                                w->name, dai->capture_widget->name);

                        snd_soc_dapm_add_path(w->dapm, w,
                                dai->capture_widget, NULL, NULL);
                }
        }
}

return 0;
```

由此可见，dai widget和stream widget是通过stream name进行匹配的，所以，我们在定义codec的stream widget时，它们的stream name必须要包含dai的stream name，这样才能让ASoc自动把这两种widget连接在一起，只有把它们连接在一起，ASoc中的播放、录音和停止等事件，才能通过dai widget传递到codec中，使得codec中的widget能根据目前的播放状态，动态地开启或关闭音频路径上所有widget的电源。我们看看wm8993中的例子：

```cpp
SND_SOC_DAPM_AIF_OUT("AIFOUTL", "Capture", 0, SND_SOC_NOPM, 0, 0),
SND_SOC_DAPM_AIF_OUT("AIFOUTR", "Capture", 1, SND_SOC_NOPM, 0, 0),

SND_SOC_DAPM_AIF_IN("AIFINL", "Playback", 0, SND_SOC_NOPM, 0, 0),
SND_SOC_DAPM_AIF_IN("AIFINR", "Playback", 1, SND_SOC_NOPM, 0, 0),
```

分别定义了左右声道两个stream name为Capture和Playback的stream widget。对应的dai driver结构定义如下：

```cpp
static struct snd_soc_dai_driver wm8993_dai = {
        .name = "wm8993-hifi",
        .playback = {
                .stream_name = "Playback",
                .channels_min = 1,
                .channels_max = 2,
                .rates = WM8993_RATES,
                .formats = WM8993_FORMATS,
                .sig_bits = 24,
        },
        .capture = {
                 .stream_name = "Capture",
                 .channels_min = 1,
                 .channels_max = 2,
                 .rates = WM8993_RATES,
                 .formats = WM8993_FORMATS,
                 .sig_bits = 24,
         },
        .ops = &wm8993_ops,
        .symmetric_rates = 1,
};
```

可见，它们的stream name是一样的，声卡初始化阶段会把它们连接在一起。需要注意的是，如果我们定义了snd_soc_dapm_aif_in和snd_soc_dapm_aif_out类型的stream widget，并指定了他们的stream name，在定义DAC或ADC对应的widget时，它们的stream name最好不要也使用相同的名字，否则，dai widget即会连接上AIF，也会连接上DAC/ADC，造成音频路径的混乱：

```cpp
SND_SOC_DAPM_ADC("ADCL", NULL, WM8993_POWER_MANAGEMENT_2, 1, 0),
SND_SOC_DAPM_ADC("ADCR", NULL, WM8993_POWER_MANAGEMENT_2, 0, 0),

SND_SOC_DAPM_DAC("DACL", NULL, WM8993_POWER_MANAGEMENT_3, 1, 0),
SND_SOC_DAPM_DAC("DACR", NULL, WM8993_POWER_MANAGEMENT_3, 0, 0),
```

stream event

把dai widget和stream widget连接在一起，就是为了能把ASoc中的pcm处理部分和dapm进行关联，pcm的处理过程中，会通过发出stream event来通知dapm系统，重新扫描并调整音频路径上各个widget的电源状态，目前dapm提供了以下几种stream event:

```cpp
/* dapm stream operations */
#define SND_SOC_DAPM_STREAM_NOP                 0x0
#define SND_SOC_DAPM_STREAM_START               0x1
#define SND_SOC_DAPM_STREAM_STOP                0x2
#define SND_SOC_DAPM_STREAM_SUSPEND             0x4
#define SND_SOC_DAPM_STREAM_RESUME              0x8
#define SND_SOC_DAPM_STREAM_PAUSE_PUSH  0x10
#define SND_SOC_DAPM_STREAM_PAUSE_RELEASE       0x20
```

比如，在soc_pcm_prepare函数中，会发出SND_SOC_DAPM_STREAM_START事件：

```cpp
snd_soc_dapm_stream_event(rtd, substream->stream,
                SND_SOC_DAPM_STREAM_START);
```

而在soc_pcm_close函数中，会发出SND_SOC_DAPM_STREAM_STOP事件：

```cpp
snd_soc_dapm_stream_event(rtd,
                          SNDRV_PCM_STREAM_PLAYBACK,
                          SND_SOC_DAPM_STREAM_STOP);
```

snd_soc_dapm_stream_event函数最终会使用soc_dapm_stream_event函数来完成具体的工作：

```cpp
static void soc_dapm_stream_event(struct snd_soc_pcm_runtime *rtd, int stream,
        int event)
```

```
{

        struct snd_soc_dapm_widget *w_cpu, *w_codec;
        struct snd_soc_dai *cpu_dai = rtd->cpu_dai;
        struct snd_soc_dai *codec_dai = rtd->codec_dai;

        if (stream == SNDRV_PCM_STREAM_PLAYBACK) {
                w_cpu = cpu_dai->playback_widget;
                w_codec = codec_dai->playback_widget;
        } else {
                w_cpu = cpu_dai->capture_widget;
                w_codec = codec_dai->capture_widget;
        }
```

该函数首先从snd_soc_pcm_runtime结构中取出cpu dai widget和codec dai widget，接下来：

```cpp
if (w_cpu) {

        dapm_mark_dirty(w_cpu, "stream event");

        switch (event) {
        case SND_SOC_DAPM_STREAM_START:
                w_cpu->active = 1;
                break;
        case SND_SOC_DAPM_STREAM_STOP:
                w_cpu->active = 0;
                break;
        case SND_SOC_DAPM_STREAM_SUSPEND:
        case SND_SOC_DAPM_STREAM_RESUME:
        case SND_SOC_DAPM_STREAM_PAUSE_PUSH:
        case SND_SOC_DAPM_STREAM_PAUSE_RELEASE:
                break;
        }
}
```

把cpu dai widget加入到dapm_dirty链表中，根据stream event的类型，把cpu dai widget设定为激活状态或非激活状态，接下来，对codec dai widget做出同样的处理：

```cpp
if (w_codec) {

        dapm_mark_dirty(w_codec, "stream event");

        switch (event) {
        case SND_SOC_DAPM_STREAM_START:
                w_codec->active = 1;
                break;
        case SND_SOC_DAPM_STREAM_STOP:
                w_codec->active = 0;
                break;
        case SND_SOC_DAPM_STREAM_SUSPEND:
        case SND_SOC_DAPM_STREAM_RESUME:
        case SND_SOC_DAPM_STREAM_PAUSE_PUSH:
        case SND_SOC_DAPM_STREAM_PAUSE_RELEASE:
                break;
        }
}
```

最后，它调用了我们熟悉的dapm_power_widgets函数：

```cpp
dapm_power_widgets(rtd->card, event);
```

因为dai widget和codec上的stream widget是相连的，所以，dai widget的激活状态改变，会沿着音频路径传递到路径上的所有widget，等dapm_power_widgets返回后，如果发出的是SND_SOC_DAPM_STREAM_START事件，路径上的所有widget会处于上电状态，保证音频数据流的顺利播放，如果发出的是SND_SOC_DAPM_STREAM_STOP事件，路径上的所有widget会处于下电状态，保证最小的功耗水平。

上一篇 ALSA声卡驱动中的DAPM详解之六：精髓所在，牵一发而动全身
下一篇 Linux SPI总线和设备驱动架构之一：系统概述

我的同类文章

Linux音频子系统（14）

- ALSA声卡驱动中的DAPM详… 2013-11-04 阅读 9950
- ALSA声卡驱动中的DAPM详… 2013-11-04 阅读 12941
- ALSA声卡驱动中的DAPM详… 2013-11-01 阅读 10222
- ALSA声卡驱动中的DAPM详… 2013-10-24 阅读 12568
- ALSA声卡驱动中的DAPM详… 2013-10-23 阅读 10804
- ALSA声卡驱动中的DAPM详… 2013-10-18 阅读 17125
- Linux ALSA声卡驱动之八：… 2012-03-13 阅读 31315
- Linux ALSA声卡驱动之七：… 2012-02-23 阅读 36156
- Linux ALSA声卡驱动之六：… 2012-02-03 阅读 37515
- Linux ALSA声卡驱动之五：… 2012-01-17 阅读 26715

更多文章

猜你在找

嵌入式Linux项目实战：三个大项目(数码相框、摄像头驱
从零写Bootloader及移植uboot、linux内核、文件系统移
从三星官方内核开始移植-uboot与系统移植第17部分
话说linux内核-uboot和系统移植第14部分
内核的启动过程分析-uboot和系统移植第16部分

查看评论

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