关于Linux线程的线程栈以及TLS

linux thread struct linux内核 user descriptor

说明：

a.本文描述Linux NPTL的线程栈简要实现以及线程本地存储理，实验环境中Linux内核版本为2.6.32，glibc版本是2.12.1，Linux发行版为ubuntu，硬件平台为x86的32位系统。

b.对于Linux NPTL线程，有很多话题。本文挑选了原则上是每线程私有的地址空间来讨论，分别是线程栈和TLS。原则山私有并不是真的私有，因为大家都知道线程的特点就是共享地址空间，原则私有空间就是一般而言通过正常手段其它线程不会触及这些空间的数据。

一.线程栈

虽然Linux将线程和进程不加区分的统一到了task_struct，但是对待其地址空间的stack还是有些区别的。对于Linux进程或者说主线程，其stack是在fork的时候生成的，实际上就是复制了父亲的stack空间地址，然后写时拷贝(cow)以及动态增长，这可从sys_fork调用do_fork的参数中看出来：

```
int sys_fork(struct pt_regs *regs)
{
    return do_fork(SIGCHLD, regs->sp, regs, 0, NULL, NULL);
}
```

何谓动态增长呢？可以看到子进程初始的size为0，然后由于复制了父亲的sp以及稍后在dup_mm中复制的所有vma，因此子进程stack的flags仍然包含：

```
#define VM_STACK_FLAGS    (VM_GROWSDOWN | VM_STACK_DEFAULT_FLAGS | VM_ACCOUNT)
```

这就说针对带有这个flags的vma(stack也在一个vma中！)可以动态增加其大小了，这可从do_page_fault中看到：

```
if (likely(vma->vm_start <= address))
    goto good_area;
if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {
    bad_area(regs, error_code, address);
    return;
}
```

很清晰。

然而对于主线程生成的子线程而言，其stack将不再是这样的了，而是事先固定下来的，使用mmap系统调用，它不带有VM_STACK_FLAGS 标记(估计以后的内核会支持！)。这个可以从glibc的nptl/allocatestack.c中的allocate_stack函数中看到：

```
mem = mmap (NULL, size, prot,
        MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
```

此调用中的size参数的获取很是复杂，你可以手工传入stack的大小，也可以使用默认的，一般而言就是默认的。这些都不重要，重要的是，这种stack不能动态增长，一旦用尽就没了，这是和生成进程的fork不同的地方。在glibc中通过mmap得到了stack之后，底层将调用sys_clone系统调用：

```
int sys_clone(struct pt_regs *regs)
{
    unsigned long clone_flags;
    unsigned long newsp;
    int __user *parent_tidptr, *child_tidptr;

    clone_flags = regs->bx;
    //获取了mmap得到的线程的stack指针
    newsp = regs->cx;
    parent_tidptr = (int __user *)regs->dx;
    child_tidptr = (int __user *)regs->di;
    if (!newsp)
        newsp = regs->sp;
    return do_fork(clone_flags, newsp, regs, 0, parent_tidptr, child_tidptr);
}
```

因此，对于子线程的stack，它其实是在进程的地址空间中map出来的一块内存区域，原则上是线程私有的，但是同一个进程的所有线程生成的时候浅拷贝生成者的task_struct的很多字段，其中包括所有的vma，如果愿意，其它线程也还是可以访问到的，于是一定要注意。

二.线程本地存储-TLS

Linux的glibc使用GS寄存器来访问TLS，也就是说，GS寄存器指示的段指向本线程的TEB(Windows的术语)，也就是TLS，这么做有个好处，那就

是可以高效的访问TLS里面存储的信息而不用一次次的调用系统调用，当然使用系统调用的方式也是可以的。之所以可以这么做，是因为Intel对各个寄存器的作用的规范规定的比较松散，因此你可以拿GS，FS等段寄存器来做几乎任何事，当然也就可以做TLS直接访问了，最终glibc在线程启动的时候首先将GS寄存器指向GDT的第6个段，完全使用段机制来支持针对TLS的寻址访问，后续的访问TLS信息就和访问用户态的信息一样高效了。

在线程启动的时候，可以通过sys_set_thread_area来设置该线程的TLS信息，所有的信息都得glibc来提供：

```
asmlinkage int sys_set_thread_area(struct user_desc __user *u_info)
{
    int ret = do_set_thread_area(current, -1, u_info, 1);
    asmlinkage_protect(1, ret, u_info);
    return ret;
}
int do_set_thread_area(struct task_struct *p, int idx,
               struct user_desc __user *u_info,
               int can_allocate)
{
    struct user_desc info;

    if (copy_from_user(&info, u_info, sizeof(info)))
        return -EFAULT;

    if (idx == -1)
        idx = info.entry_number;

    /*
     * index -1 means the kernel should try to find and
     * allocate an empty descriptor:
     */
    if (idx == -1 && can_allocate) {
        idx = get_free_idx();
        if (idx < 0)
            return idx;
        if (put_user(idx, &u_info->entry_number))
            return -EFAULT;
    }

    if (idx < GDT_ENTRY_TLS_MIN || idx > GDT_ENTRY_TLS_MAX)
        return -EINVAL;

    set_tls_desc(p, idx, &info, 1);

    return 0;
}
```

fill_ldt设置GDT中第6个段描述符的基址和段限以及DPL等信息，这些信息都是从sys_set_thread_area系统调用的u_info参数中得来的。本质上，最终GDT的第6个段中描述的信息其实就是一块内存，这块内存用于存储TLS节，这块内存其实也是使用brk，mmap之类调用在主线程的堆空间申请的，只是后来调用sys_set_thread_area将其设置成了本线程的私有空间罢了，主线程或者其它线程如果愿意，也是可以通过其它手段访问到这块空间的。

明白了大致原理之后，我们来看一下一切是如何关联起来的。首先看一下Linux内核关于GDT的段定义，如下图所示：

```
#ifdef CONFIG_X86_32
/*
 * The layout of the per-CPU GDT under Linux:
 *
 *  0 - null
 *  1 - reserved
 *  2 - reserved
 *  3 - reserved
 *
 *  4 - unused                  <====== new cacheline
 *  5 - unused
 *
 * ------- start of TLS (Thread-Local Storage) segments:
 *
 *  6 - TLS segment #1          [ glibc's TLS segment ]
 *  7 - TLS segment #2          [ Wine's %fs Win32 segment ]
 *  8 - TLS segment #3
 *  9 - reserved
 * 10 - reserved
 * 11 - reserved
 *
 * ------- start of kernel segments:
 *
 * 12 - kernel code segment          <====== new cacheline
 * 13 - kernel data segment
 * 14 - default user CS
 * 15 - default user DS
 * 16 - TSS
 * 17 - LDT
 * 18 - PNPBIOS support (16->32 gate)
```

glibc使用这个段来表示TLS，glibc自己设置段寄存器GS

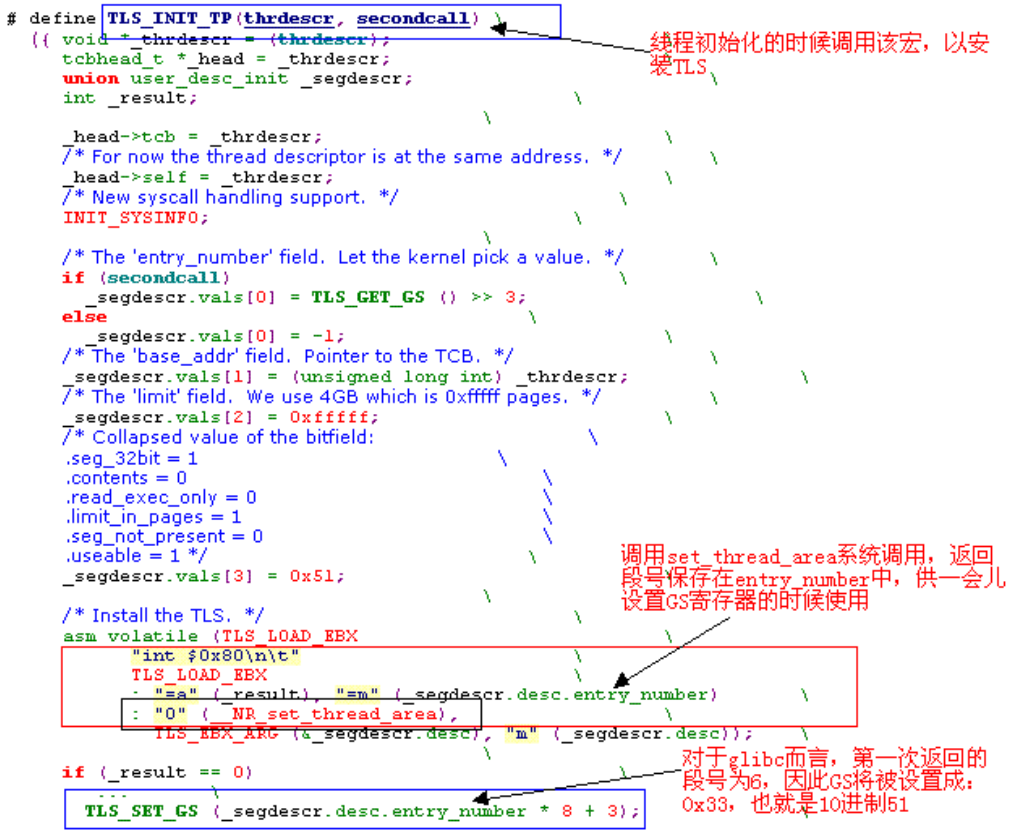
我们发现是第六个段用于记录TLS数据，我了证实一下，写一个最简单的程序，用gdb看一下GS寄存器的值，到此我们已经知道GS寄存器表示的段描述子指向的段记录TLS数据，如下图所示：

```
root@ubuntu:/test# cat test_gs.c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char **argv)
{
        int a = 10;
        printf("show %d\n", a);
}
root@ubuntu:/test# gdb test_gs
GNU gdb (GDB) 7.2-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /test/test_gs...done.
(gdb) b main
Breakpoint 1 at 0x80483f5: file test_gs.c, line 5.
(gdb) r
Starting program: /test/test_gs

Breakpoint 1, main (argc=<value optimized out>, argv=<value optimized out>) at test_gs.c:5
5               int a = 10;
(gdb) info reg
eax            0xbffff6b4       -1073744204
ecx            0xbffff610       -1073744368
edx            0x1      1
ebx            0x288ff4 2658292
esp            0xbffff5d0       0xbffff5d0
ebp            0xbffff5f8       0xbffff5f8
esi            0x0      0
edi            0x0      0
eip            0x80483f5        0x80483f5 <main+17>
eflags         0x282    [ SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

可以看到红色圈住的部分，GS的值是0x33，这个0x33如何解释呢？见下图分解：

这就证实了确实是GS指向的段来表示TLS数据了，在glibc中，初始化的时候会将GS寄存器指向第六个段：



既然如此，我们是不是可以直接通过GS寄存器来访问TLS数据呢？答案当然是肯定的，glibc其实就是这么做的，无非经过封装，使用更加方便了。但是如果想明白其所以然，还是自己折腾一下比较妥当，我的环境是ubuntu glibc-2.12.1，值得注意的是，每一个glibc的版本的TLS header都可能不一样，一定要对照自己调试的那个版本的源码来看，否则一定会发疯的。我将上面的那个test_gs.c修改了一下，成为下面的代码：

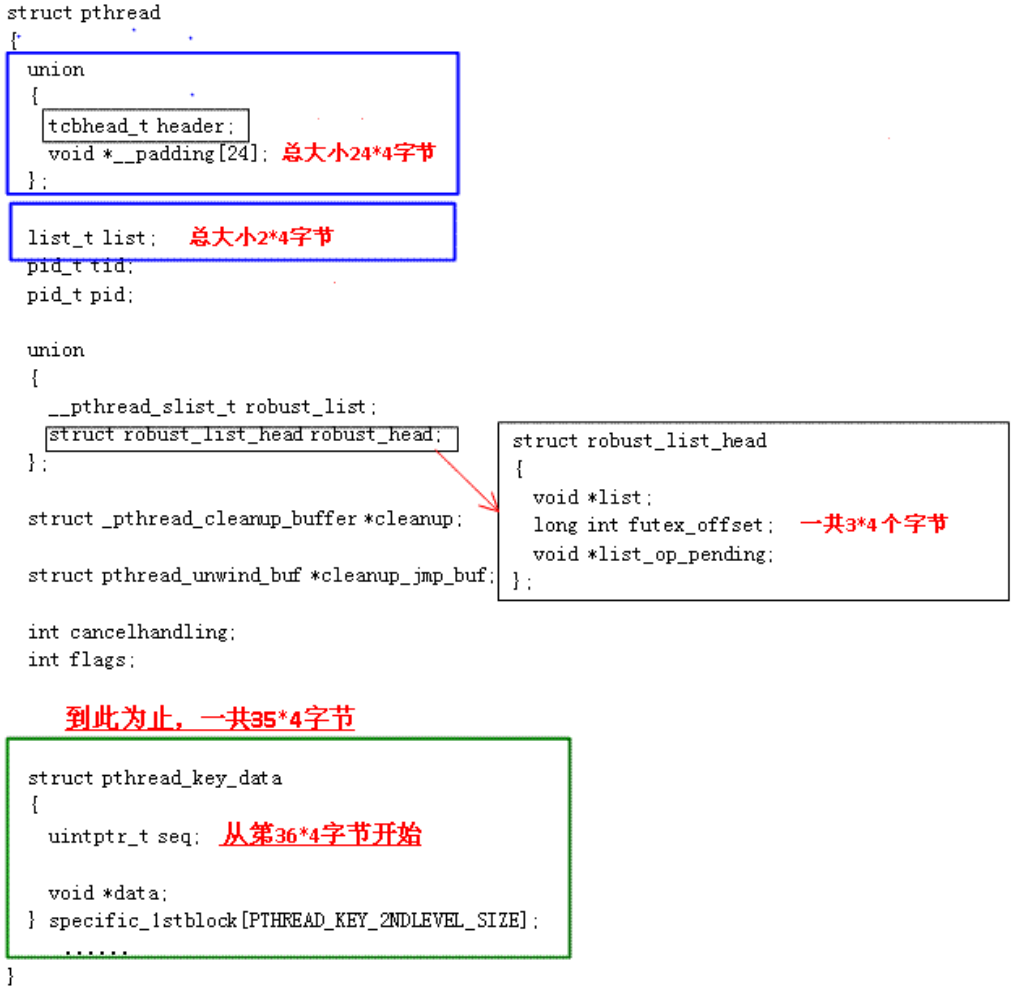
```
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <pthread.h>
int main(int argc, char **argv)
{
    int a=10, b = 0;  //b保存GS寄存器表示的段的地址
    //设置三个TLS变量，其中前两个使用堆内存，最后一个不使用
        static pthread_key_t thread_key1;
        static pthread_key_t thread_key2;
        static pthread_key_t thread_key3;
        char *addr1 = (char *)malloc(5);
        char *addr2 = (char *)malloc(5);
        memset(addr1, 0, 5);
        memset(addr2, 0, 5);
        strcpy(addr1, "aaaa");
```

```
        strcpy(addr2, "bbbb");
        pthread_key_create (&thread_key1, NULL);
        pthread_key_create (&thread_key2, NULL);
        pthread_key_create (&thread_key3, NULL);
        pthread_setspecific (thread_key1, addr1);
        pthread_setspecific (thread_key2, addr2);
        pthread_setspecific (thread_key3, "1111111111");
    //得到GS指示的段，也就是TLS的地址，这个需要用内嵌汇编来做
        asm volatile("movl %%gs:0, %0;"
                :"=r"(b)  /* output */
        );

        printf("ok\n");
}
```

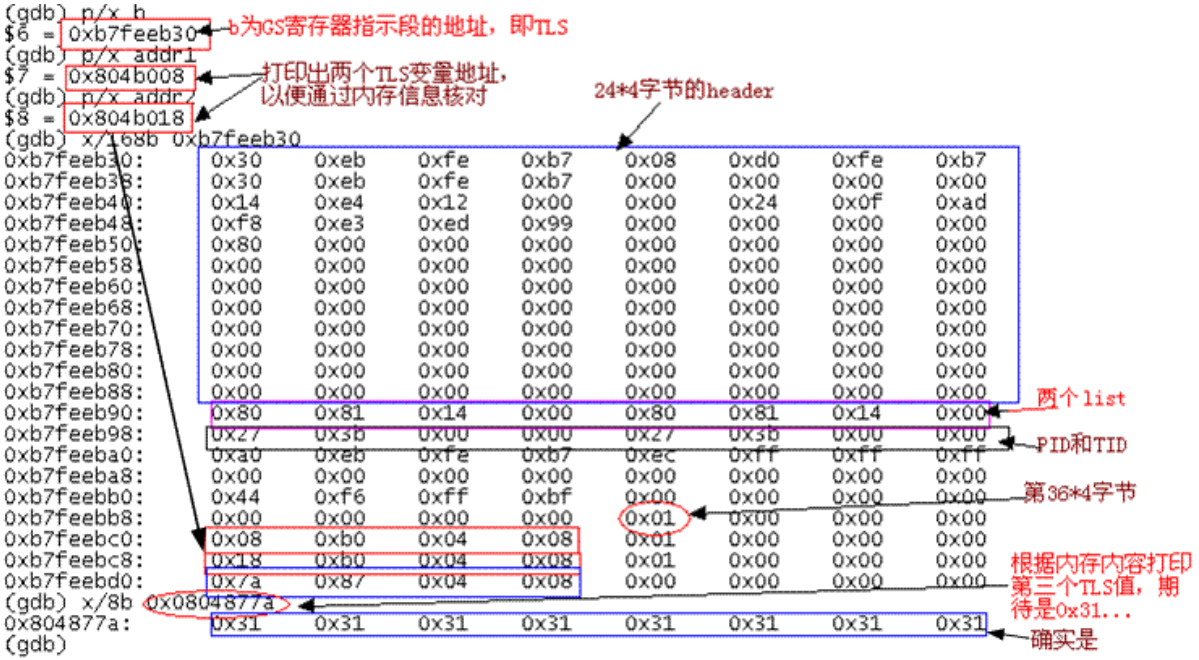
这个代码的含义在于，我可以通过GS寄存器访问到TLS变量，为了方便，我就没有写代码，而是通过gdb来证实，其实通过写代码取出TLS变量和通过gdb查看内存的方式效果是一样的，个人认为通过调试的方法对于理解还更好些。

当调试的时候，在取出GS之后，我们得到了TLS的地址，然后根据该版本的TLS结构体分析哪里存储的是TLS变量，然后查看TLS地址附近的内存，证实那里确实存着一个TLS变量，这可以通过比较地址得出结论。当然在实际操作之前，我们首先看一下glibc-2.12.1版本的TLS数据结构，如下图所示：



注意，由于我们并无意深度hack TLS，因此仅仅知道在何处能取到变量即可，因此我们只需要知道一些字段的大小就可以了，暂且不必理解其含义与设计思想。

我们发现，应该是从第35*4个字节开始就是TLS变量的区域了，是不是这样呢？我们来看一下调试结果，注意我们要把断点设置在asm之后，这样才能打出b的值，当然你也可以调整上述代码，把asm内嵌汇编放在代码最前面也是可以的。gdb命令就不多说了，都是些简单的，如下展示出结果：



结果很明了了。最终还有一个小问题，那就是关于线程切换的问题。

对于Windows而言，线程的TEB几乎是固定的，而对于Linux，它同样也是这样子，只需要得到GS寄存器，就能得到当前线程的TCB，换句话说，GS始终是不变化的，始终是0x33，始终指向GDT的第6个段，变化的是GDT的第6个段的内容，每当进程或者线程切换的时候，第6个段的内容都需要重新加载，载入将要运行线程的TLS info中的信息，这是在切换时switch_to宏中完成的：

```
load_TLS(next, cpu);
```

每个task_struct都有thread_struct，而该线程TLS的元数据信息就保存在thread_struct结构体的tls_array数组中：

```
static inline void native_load_tls(struct thread_struct *t, unsigned int cpu)
{
    unsigned int i;
    struct desc_struct *gdt = get_cpu_gdt_table(cpu);

    for (i = 0; i < GDT_ENTRY_TLS_ENTRIES; i++)
        gdt[GDT_ENTRY_TLS_MIN + i] = t->tls_array[i];
}
```

注意：关于TLS另外需要说的

除了我们使用pthread的API在运行时创建的TLS变量之外，还有一部分TLS称为静态TLS变量，这些TLS元素是在编译期间预先生成的，常见的

有：

1.自定义 _thread修饰符修饰的变量；

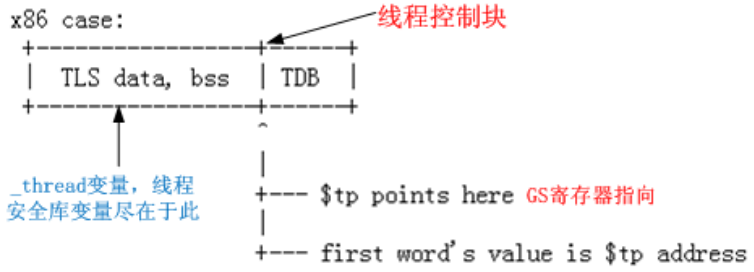
2.一些库级别预定义的变量，比如errno

那么这些变量存储在哪里呢？设计者很明智的将其放在了动态TLS临接的空间内，就是GS寄存器指示的地址下面，其实要是我设计也会这么设计的，你也一样。这样设计的好处在于可以很方便对不管是动态TLS变量还是静态TLS变量的访问，并且对于动态TLS的管理也很方便。

这些数据处于“initialized data section”，然而在链接或者线程初始化的时候被动态重定向到了静态TLS空间内，在我的实验环境中，如果我定义了一个变量：

_thread int test = 123;

那么调试显示的结果，它处于GS寄存器指示tls段地址的紧接着下方4个字节的偏移处，而errno处于_thread变量下方14*4字节的位置。具体这些空间到底怎么安排的，可以看glibc的dl-reloc.c，dl-tls.c等文件，然而本人认为这没有什么意义，由于这涉及到很多关于编译，链接，重定向，ELF等知识，如果不想深度优先的迷失在这里面的化，理解原理也就够了，本人真的是没有时间再写了，回到家就要看孩子，购物，做家务....。最后给出一幅图，重定向后总的示意图如下：



- 上一篇 ip_queue和libcrypto实现另一种VPN
- 下一篇 IEEE802.11数据帧在Linux上的抓取

查看评论

williamturner 7楼 2013-09-26 12:10发表
好屌的样子，不明觉厉。。。。。

manuscola 6楼 2013-04-28 10:23发表
这个TLS相关的问题，我想了好久也没想明白，看了Ulrich Drepper的TLS的doc 也没能明白。这篇写的真好。
敢问博主在什么公司？ RH ？ALI？

nono Re: 2013-04-28 16:48发表
回复manuscola：在下一介草民，从不敢奢望能进入RH之类的大庙啊，呵呵，最主要是人家不要我啊

hbhhww 5楼 2012-07-27 17:14发表

这么牛！犀利啊

next_unless 4楼 2012-07-05 20:11发表

膜拜！完全看不懂了，敢问楼主是redhat之类的公司就职吗？

vip_code99 3楼 2012-07-04 21:03发表

膜拜一下。

wjb_yd 2楼 2012-07-04 10:41发表

真心看不懂，不过还是支持一下楼主，希望有朝一日能达到楼主的技术水平。

l200698278 1楼 2012-07-03 18:10发表

顶！！！！！！

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