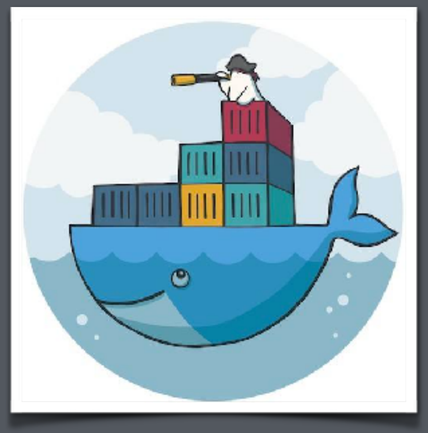


Docker容器监控数据采集的探究与思考

by 王亚普

自我介绍

- 美团点评上海基础架构部
- 2015年加入原大众点评
- 先后参与点评私有云、分布式监控系统Cat、Slb软负载的研发工作
- 开源技术、高性能组件、高效开发工具的狂热爱好者

聊点什么

- 点评私有云设计方案
- 公司组件的生态圈

- 用户使用场景、行为习惯
- 推进过程中一些头疼的事
- 目前的解决方案与不足

- 基于Lxcfs容器物理指标的计算方法
- 扩展与思考

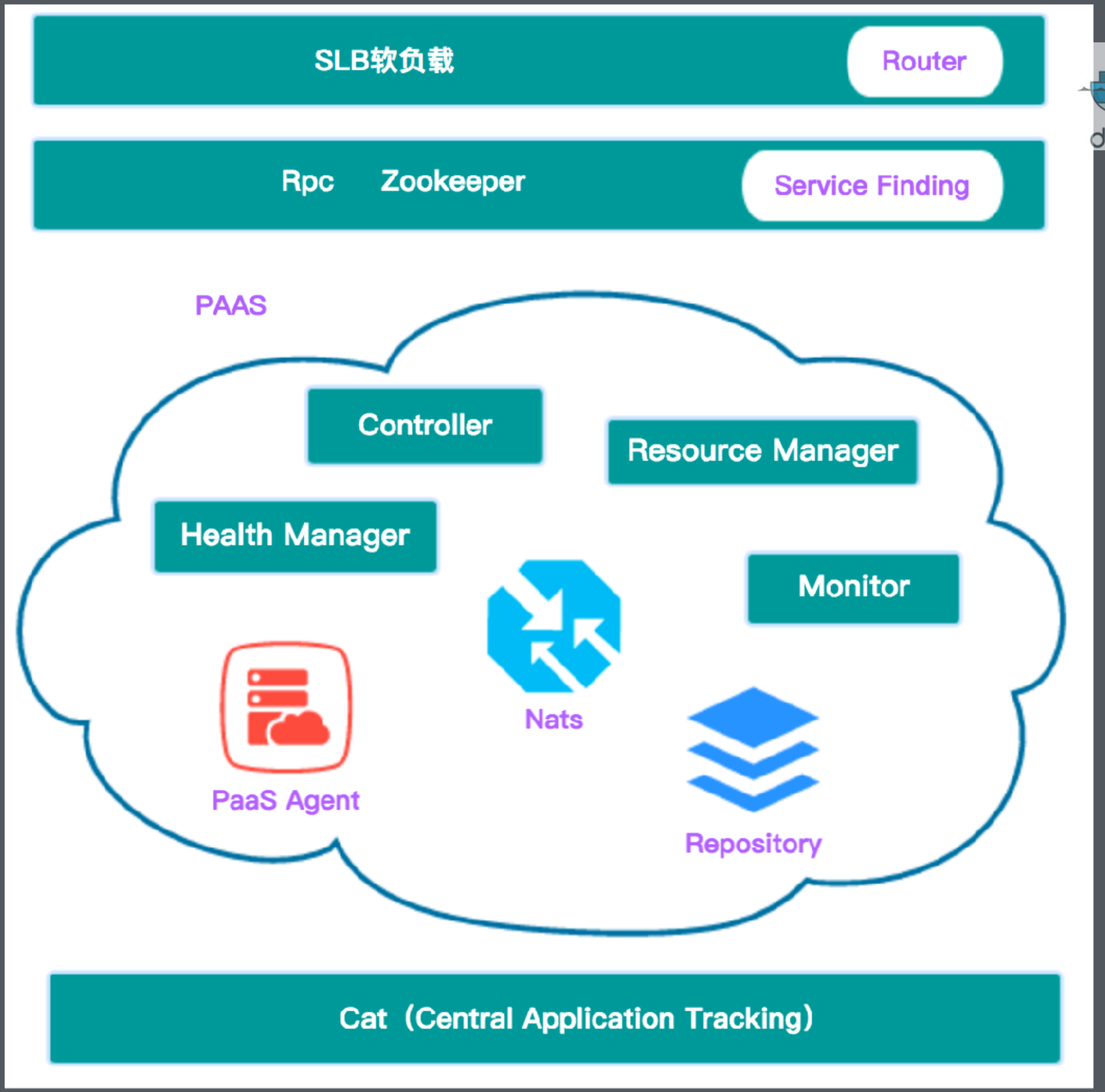
设计目标

- 让用户无感知从Kvm切换到Docker的过程
- 把容器做的更像KVM
- 无缝对接公司的内部组件
- 高密度部署、快速扩容
- 标准化应用运行环境

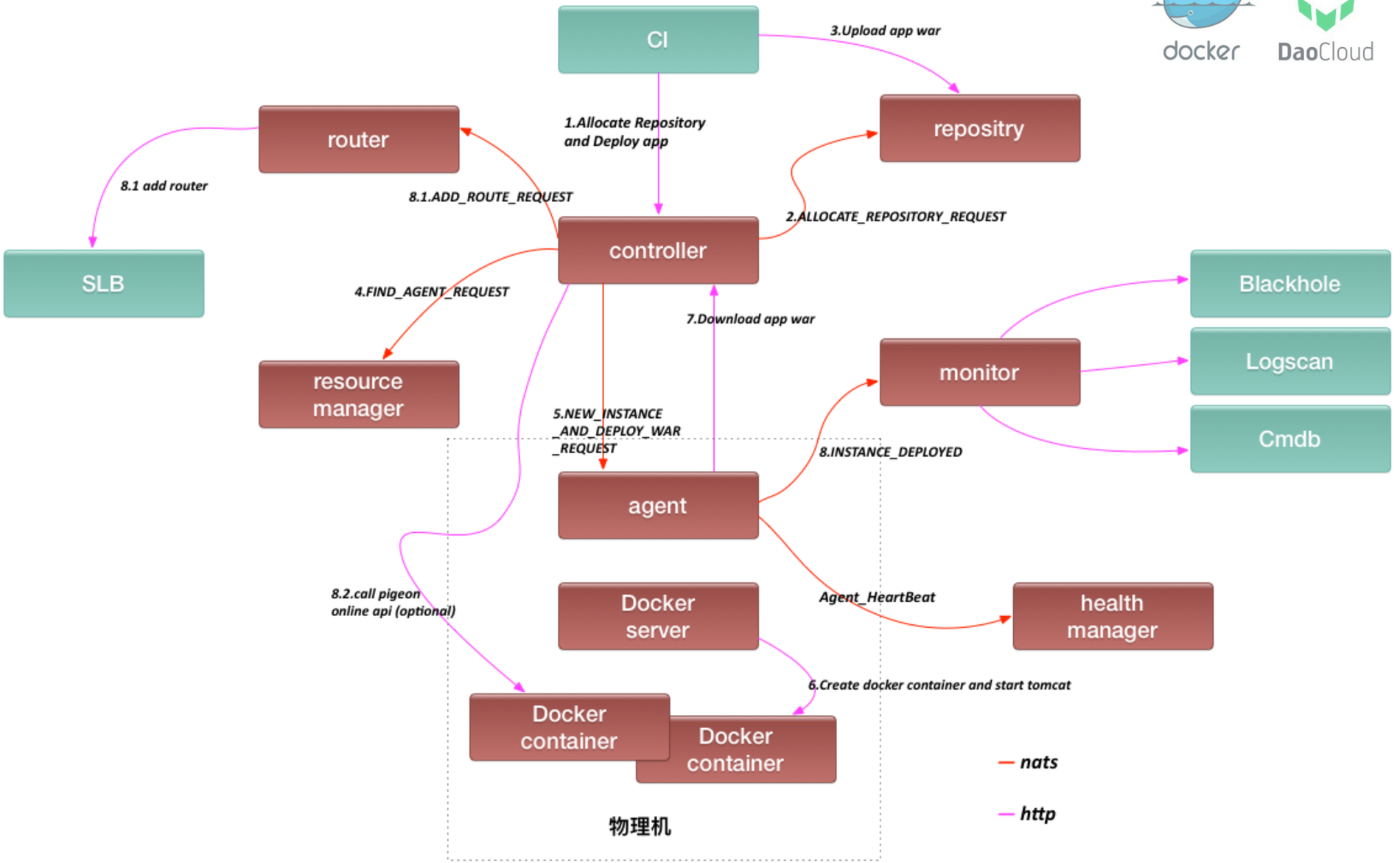
为什么要做的像KVM?

- 考虑到公司组件的对接
- 运维成本适中
- 开发人员无需编写dockerfile，配置各种不同的启动方式

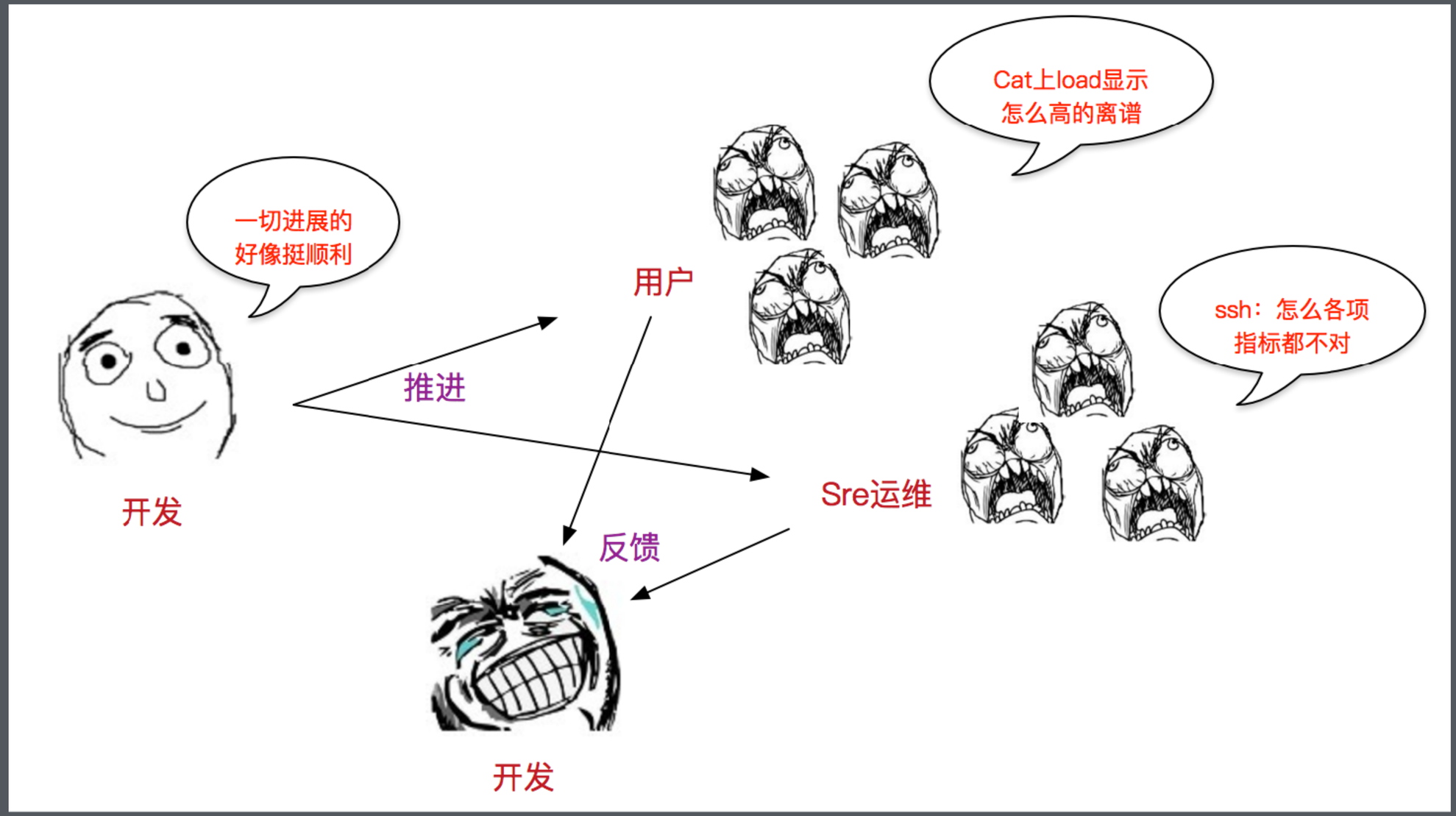
点评私有云生态圈



docker DaoCloud IT大咖说 不止于技术


CI
3.Upload app war
router
1.Allocate Repository
and Deploy app
repositry
8.1 add router
8.1.ADD_ROUTE_REQUEST
2.ALLOCATE_REPOSITORY_REQUEST
controller
SLB
4.FIND_AGENT_REQUEST
7.Download app war
Blackhole
Logscan
Cmdb
monitor
resource
manager
5.NEW_INSTANCE
_AND_DEPLOY_WAR
_REQUEST
8.INSTANCE_DEPLOYED
agent
8.2.call pigeon
online api (optional)
Docker
server
Agent_HeartBeat
health
manager
6.Create docker container and start tomcat
Docker
container
Docker
container
nats
http
物理机

推进落地过程中踩过的坑



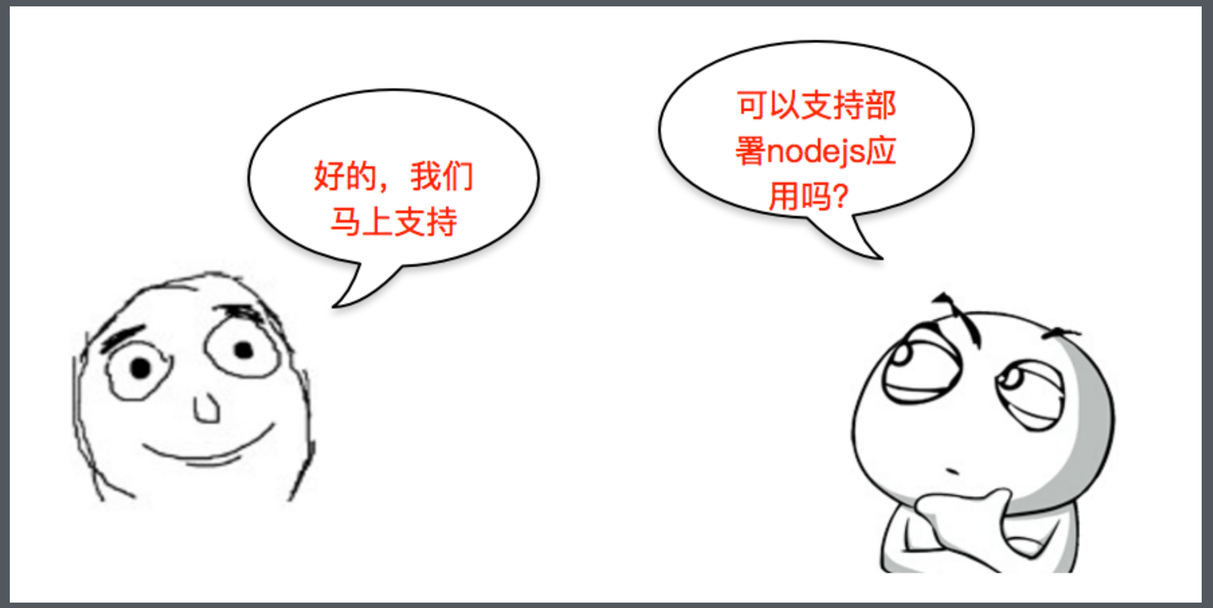
Why? How to Solve?

- 第一反应会是这是docker造成的，但并非完全正确。准确的说，这些问题是由于docker的隔离性不足导致的。

```
static int loadavg_proc_show(struct seq_file *m, void *v)
{
        unsigned long avnrun[3], nr_runnable = 0;
        struct cpumask cpus_allowed;
        int i;

        rcu_read_lock();
        if (task_in_nonroot_cpuacct(current) &&
                in_noninit_pid_ns(current->nsproxy->pid_ns)) {

                get_avenrun_from_tsk(current, avnrun, FIXED_1/200, 0);

                cpumask_copy(&cpus_allowed, cpu_possible_mask);
                if (task_subsys_state(current, cpuset_subsys_id)) {
                        memset(&cpus_allowed, 0, sizeof(cpus_allowed));
                        get_tsk_cpu_allowed(current, &cpus_allowed);
                }

                for_each_cpu_and(i, cpu_possible_mask, &cpus_allowed)
                        nr_runnable += task_ca_running(current, i);

        } else {
                get_avenrun(avnrun, FIXED_1/200, 0);
                nr_runnable = nr_running();
        }
        rcu_read_unlock();
```

大动干戈之后继续推进

- 经过灰度测试后，对现有机器通过打补丁的方式升级、重启。
- 后续新申请的机器都安装最新的内核。



联调后

目前解决方案的弊端

- 为了在容器内准确查看linux的各项指标，就需要大动干戈修改linux code，万一出现bug怎么办？这不是一个可以被广泛接受的方案。
- 如果现在需要升级一个新的linux稳定版，又需要打上修改的补丁，重启物理机，这个工作量真是吃力不讨好。
- 如果再抛出一个未考虑到的问题，是非常令人崩溃的。
- 没有专门维护Linux内核的团队。

梳理问题的共同点

- 依赖于一些系统文件，而容器内的文件内容继承于宿主机。

Info	related file
cpu	/proc/cpuinfo
mem	/proc/meminfo
uptime	/proc/uptime
load	/proc/loadavg
cpu-online	/sys/devices/system/cpu/online

业界调研

- 修改linux系统通用软件包procps，以达到替换原有top、memory等命令。
- 并不允许开发或者运维登陆容器，自己研发一套容器的监控系统。
- 使用lxcfs组件（蘑菇街、腾讯游戏）

What is Lxcfs?

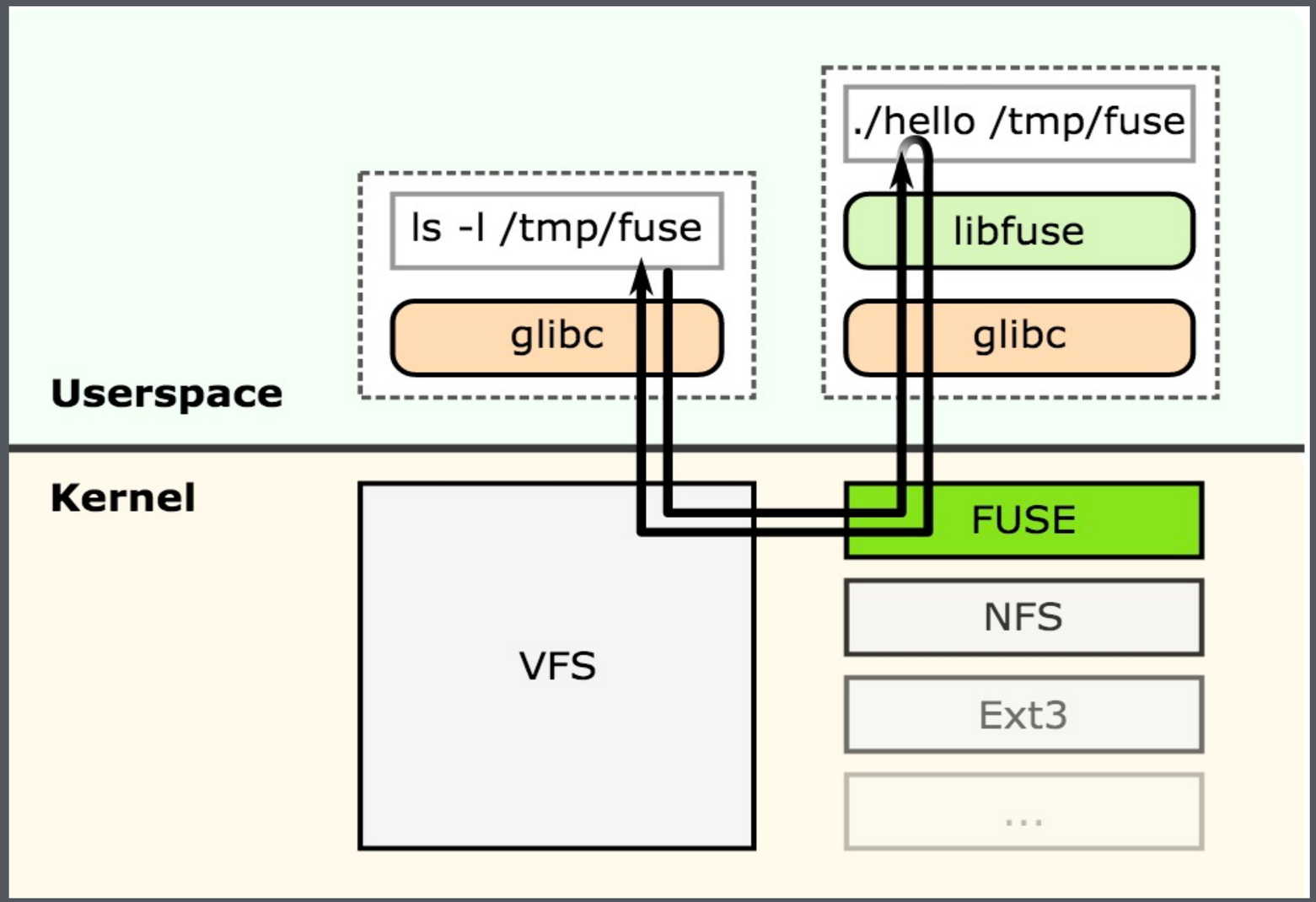
- FUSE filesystem for LXC, offering the following features:
  - ✓ a cgroupfs compatible view for unprivileged containers
  - ✓ a set of cgroup-aware files:
  - ✓ cpuinfo
  - ✓ meminfo
  - ✓ stat
  - ✓ uptime

In other words, it will provide an emulated /proc and /sys/fs/cgroup folder for the containers.

Lxcfs的发展现状

- LXCFS基于C语言开发，代码托管在github上
- 项目发展历程2014.9 – 2017.3，共发布32个版本
- 主要由Ubuntu的工程师提供维护和开发
- Lxcfs对于docker容器监控来说还算是个半成品（没有完成load、cpuonline），需要自行扩展

Lxcfs工作原理




```
struct fuse_operations {

        mkdir;
        open;
        read;
        write;
      opendir;
      readdir;
      gender;
      ……
}
```

Lxcfs工作原理

- LXCFS的启动命令

  sudo mkdir -p /var/lib/lxcfs

  sudo ./lxcfs -s -f -o allow_other /var/lib/lxcfs/

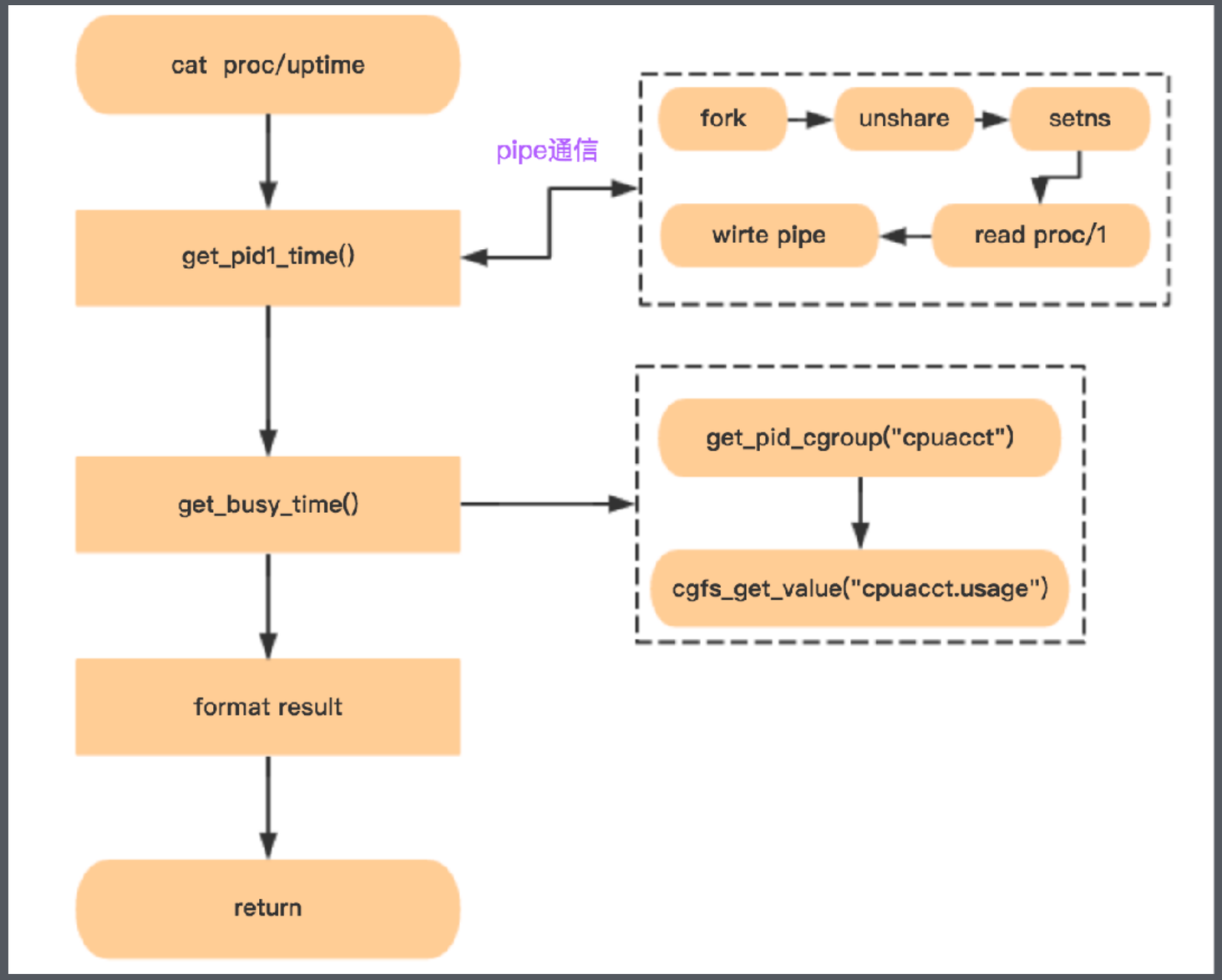
- Docker启动命令

  docker run -it --rm --cpuset=0,1 -v /var/lib/lxcfs/proc/uptime:/proc/uptime:rw -v /var/lib/lxcfs/proc/cpuinfo:/proc/cpuinfo:rw -v /var/lib/lxcfs/proc/stat:/proc/stat -v /var/lib/lxcfs/cgroup/:/cgroup/:rw -v /var/lib/lxcfs/cpu/online:/cpu/online:rw -v /var/lib/lxcfs/proc/loadavg:/proc/loadavg:rw ubuntu:14.04 /bin/bash

Lxcfs_uptime解读

- $ > cat /proc/uptime

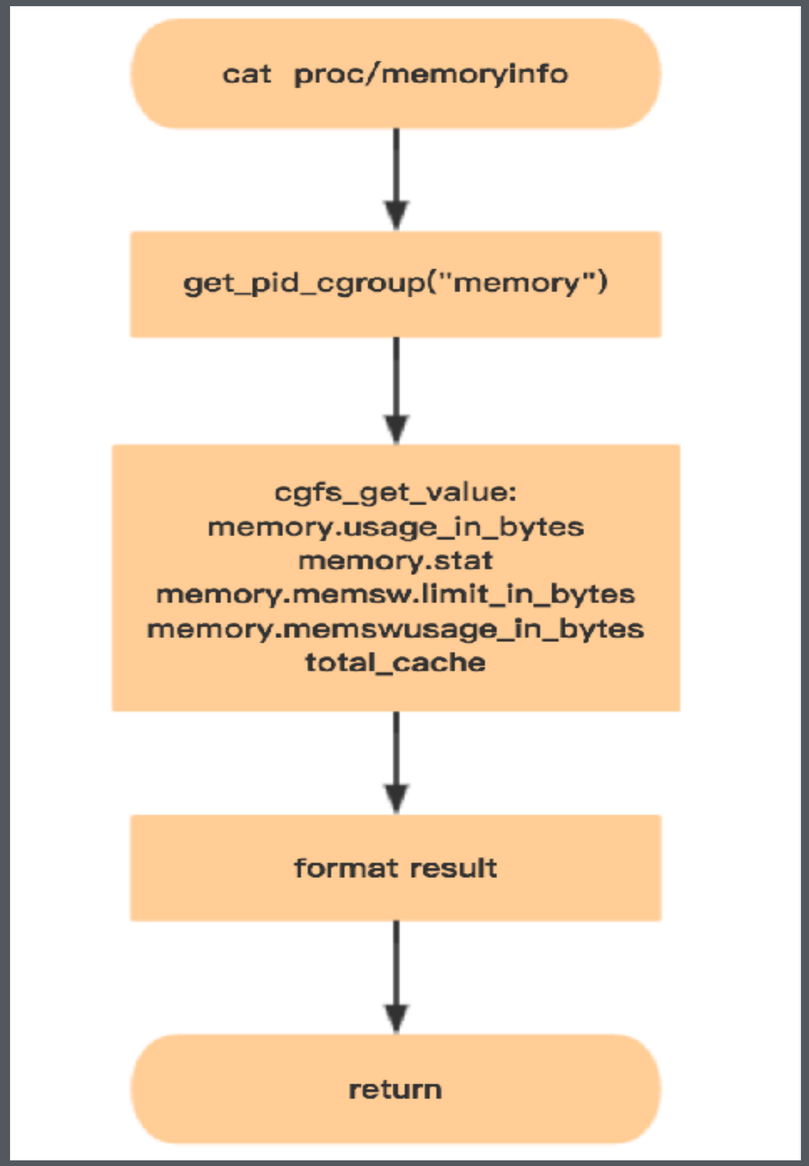
  3387048.81 3310821.00



Lxcfs_memory解读

- $ > cat /proc/meminfo

  MemTotal: 2050028 kB
  MemFree: 1169816 kB
  Cached: 450060 kB
  ……



重新审视这个半成品

- 思想值得借鉴，是个好东西
- 与docker的一些思想是想通的
- 集成难度和稳定性
- 没有cpuonline、loadavg，需要自行扩展

Loadavg的扩展

docker DaoCloud IT大咖说 不止于技术

01
Load的定义
Cpu利用率与load的区别

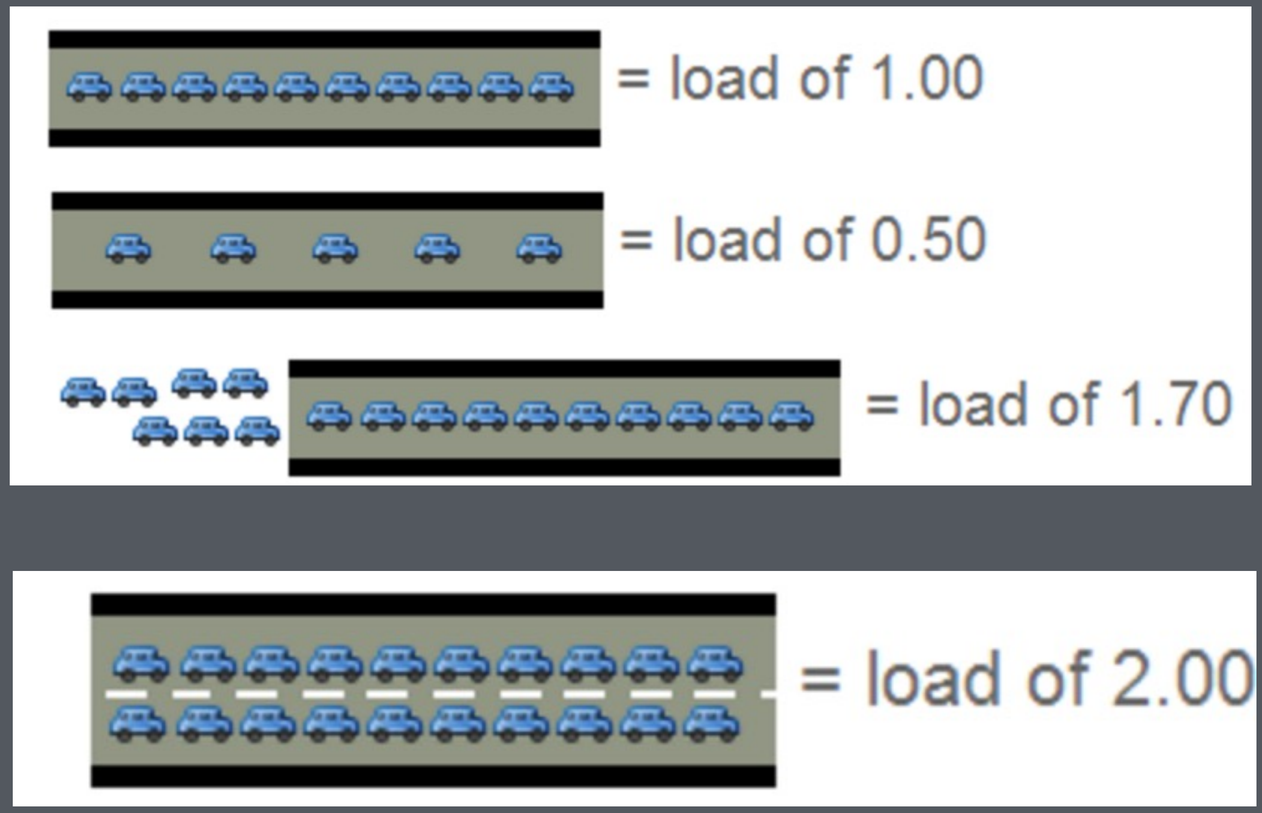
02
Linux计算load的原理

03
Lxcfs扩展load的切入点
实验验证

04
体会与思考

Load的定义与理解

- 系统平均负载被定义：在特定时间间隔内运行队列中的平均进程数。



- CPU利用率 = 1 - 程序占用cpu时间/程序总的运行时间
- 负载均值用来估量CPU利用率的发展趋势，而不是某一时刻的状况
- 负载均值包括所有CPU的需求，而不仅仅是在测量时活跃的

Linux计算load的原理


```
$ > uptime
20:15:01 up  3:07,  6 users,  load average: 0.43, 0.97, 1.05

$ > top
top - 04:00:56 up 37 min,  1 user,  load average: 0.00, 0.01, 0.01
Tasks: 101 total,   2 running,  99 sleeping,   0 stopped,   0 zombie
.....................
```

```
$ > cat /proc/loadavg
0.64 0.81 0.86 3/364 6930
```

每个值的含义依次为：

lavg_1 (0.64) 1-分钟平均负载

lavg_5 (0.81) 5-分钟平均负载

lavg_15(0.86) 15-分钟平均负载

nr_running (3) 在采样时刻，运行队列的任务的数目，与/proc/stat的procs_running表示相同意思

nr_threads (364) 在采样时刻，系统中活跃的任务的个数（不包括运行已经结束的任务）

last_pid(6930) 最大的pid值，包括轻量级进程，即线程。

Linux计算load的原理

- 源码位置

  sched.h

  loadavg.c

```
#define FSHIFT 11 /* nr of bits of precision */
#define FIXED_1 (1<<FSHIFT) /* 1.0 as fixed-point(定点) */
#define LOAD_FREQ (5*HZ) /* 5 sec intervals, 每隔5秒计算一次平均负载值 */
#define CALC_LOAD(load, exp, n) \
    load *= exp; \
    load += n*(FIXED_1 - exp); \
    load >>= FSHIFT;

unsigned long avenrun[3];
EXPORT_SYMBOL(avenrun);

static inline void calc_load(unsigned long ticks) {
    unsigned long active_tasks; /* fixed-point */
    static int count = LOAD_FREQ;
    count -= ticks;
    if (count < 0) {
        count += LOAD_FREQ;
        active_tasks = count_active_tasks();
        CALC_LOAD(avenrun[0], EXP_1, active_tasks);
        CALC_LOAD(avenrun[1], EXP_5, active_tasks);
        CALC_LOAD(avenrun[2], EXP_15, active_tasks);
    }
}
```

如果能计算出来运行队列的长度，一切看来都迎刃而解了

- load推导公式：

  load(t)= load(t-1) * EXP_N / FIXED_1 + nr_active * (1 - EXP_N/FIXED_1)

进程满足什么条件可以位于运行队列之中?

- 它没有在等待I/O操作的结果
- 它没有主动进入等待状态(也就是没有调用'wait')
- 没有被停止(例如：等待终止)

Linux进程几种状态的定义

- R (TASK_RUNNING)，可执行状态。
- S (TASK_INTERRUPTIBLE)，可中断的睡眠状态。
- D (TASK_UNINTERRUPTIBLE)，不可中断的睡眠状态。
- T (TASK_STOPPED or TASK_TRACED)，暂停状态或跟踪状态。
- Z (TASK_DEAD – EXIT_ZOMBIE)，退出状态，进程成为僵尸进程。
- X (TASK_DEAD – EXIT_DEAD)，退出状态，进程即将被销毁。

尝试与探索

- 踩坑记录1：

  /proc文件夹下有所有进程号，通过编译内核模块与linux内核通信，拿到task数据结构，得到该进程下的所有线程以及线程状态。

  结果：有些内核版本并不好使，兼容性差，放弃。

尝试与探索

- 踩坑记录2（无意中的发现）：

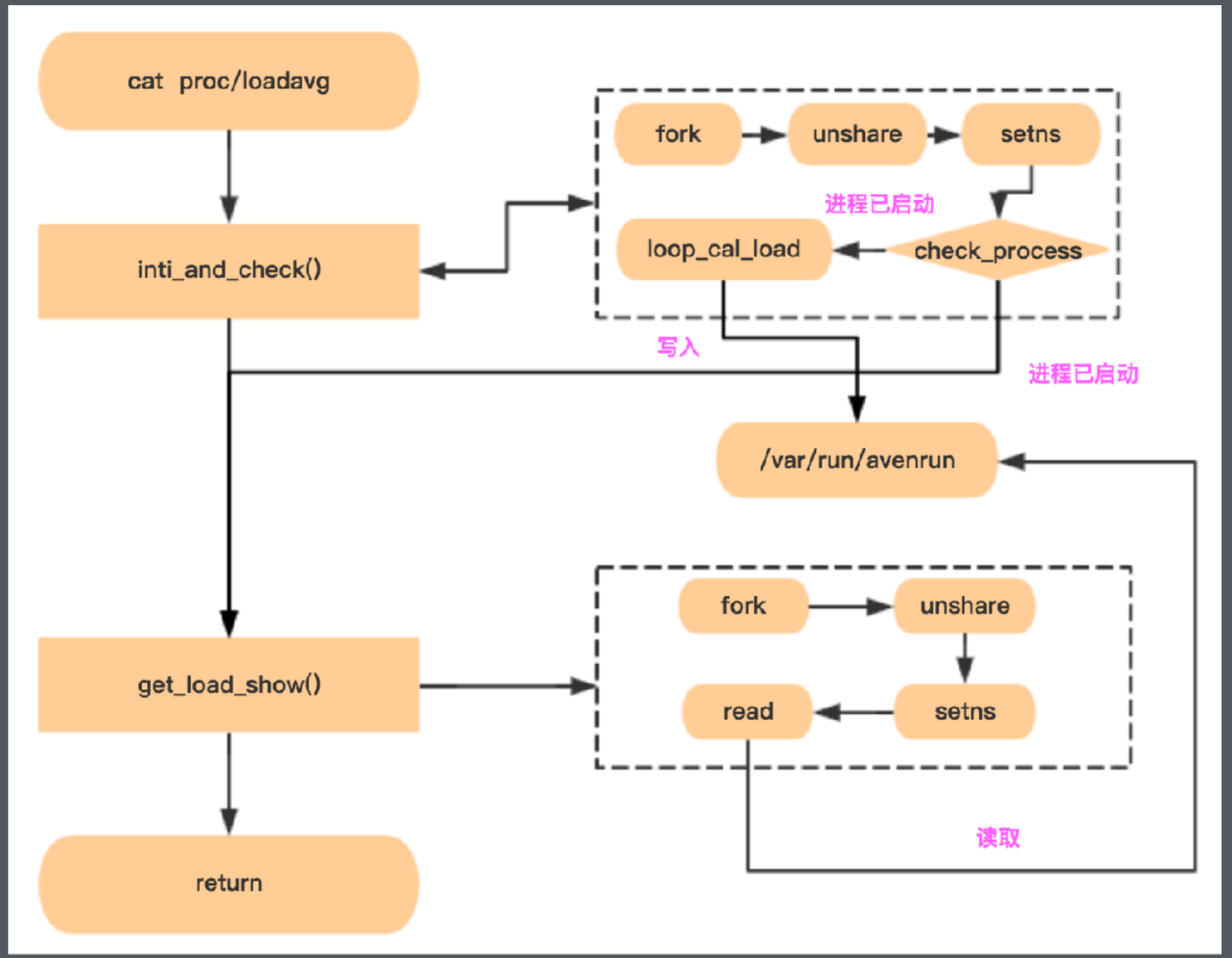
  /proc/{process}/task/下存有所有进程下的所有线程，/proc/{process}/task/{thread}/status存有线程的状态。

  结果：验证可行，lxcfs可以开始扩展了！。

```
root@vagrant-ubuntu-trusty-64:/proc/1019/task# ll
total 0
dr-xr-xr-x 6 syslog syslog 0 Mar  8 08:20 ./
dr-xr-xr-x 9 syslog syslog 0 Mar  8 03:23 ../
dr-xr-xr-x 6 syslog syslog 0 Mar  8 08:21 1019/
dr-xr-xr-x 6 syslog syslog 0 Mar  8 08:21 1020/
dr-xr-xr-x 6 syslog syslog 0 Mar  8 08:21 1021/
dr-xr-xr-x 6 syslog syslog 0 Mar  8 08:21 1022/
```

```
root@vagrant-ubuntu-trusty-64:/proc/1019/task/1022# cat status
Name:   rs:main Q:Reg
State:  S (sleeping)
Tgid:   1019
Ngid:   0
Pid:    1022
PPid:   1
TracerPid:      0
Uid:    101     101     101     101
Gid:    104     104     104     104
FDSize: 64
Groups: 4 104
VmPeak:   257916 kB
VmSize:   257912 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      1596 kB
```

尝试与探索


cat proc/loadavg
inti_and_check()
fork
unshare
setns
进程已启动
loop_cal_load
check_process
写入
进程已启动
/var/run/avenrun
get_load_show()
fork
unshare
read
setns
读取
return

体会与思考

- Lxcfs不仅可以帮助理解docker的思想，而且可以弥补docker容器因隔离性不够完善造成的使用不便。
- cpuonline的扩展相对容易，与memory类似，没想到的是jvm的垃圾回收也受它的影响
- load的扩展思路仅供参考，效率比内核计算要低，且准确度有偏差
- 相比KVM，容器技术在系统层面的隔离性相对不足
- 理解了lxcfs的做法，完全可以使用类似的方法替代它，编程语言也不受限制
- docker容器内置agent通过lxcfs收集监控数据或者从宿主机的角度实时为每个容器计算相应的指标完全可以为容器单独实现一套监控体系

Thanks!