管道与Unix哲学

2014-09-25 宾狗 理论 Unix 架构

0x00 前言

最近看到一篇文章关于Unix管道的，讲的非常透彻，所以这次依然做一个简单的**翻译和解读**~原文地址请戳这里

下面正式开始~

管道(Pipelines)是现代软件工程中一个非常有用架构模型，最早使用在Unix系统中，有句话是这么说的

> 如果说Unix是计算机文明中最伟大的发明，那么，Unix下的Pipe管道就是跟随Unix所带来的另一个伟大的发明

管道所要解决的问题，还是软件设计中老生常谈的设计目标——高内聚，低耦合。它以一种“链式模型”来串接不同的程序或者不同的组件，让它们组成一条直线的工作流。这样给定一个完整的输入，经过各个组件的先后协同处理，得到唯一的最终输出。

如果你经常使用Unix的话，一定对管道符号`|`不陌生。那么，我们来看看下面这个例子

```
cat /usr/share/dict/words |     # Read in the system's dictionary.
grep purple |                   # Find words containing 'purple'
awk '{print length($1), $1}' |  # Count the letters in each word
sort -n |                       # Sort lines ("${length} ${word}")
tail -n 1 |                     # Take the last line of the input
cut -d " " -f 2 |               # Take the second part of each line
cowsay -f tux                   # Put the resulting word into Tux's mouth
```

用`bash`运行上面的命令，最终会返回一只可爱的`Linux`小企鹅，并告诉你字典中包含`purple`最长的一个单词，看起来是下面这个样子

```
 _____________
< unimpurpled >
 -------------
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/
```

0x01 执行流程

上面看似简单的命令却执行了一个非常复杂的流程，当我们按下回车键时，下面的步骤依次执行

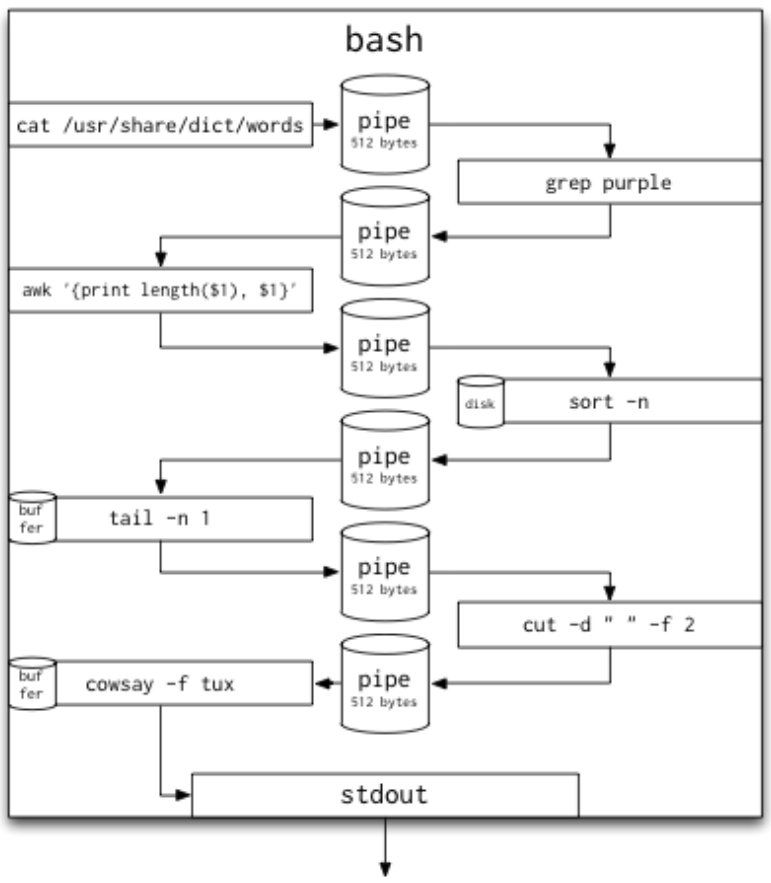
1. `shell`会立即创建7个进程
2. 每个进程的标准输入(`stdin`)和标准输出(`stdout`)将被重定向到`shell`的内部缓存中(在我的机器上每个这样的缓存大小为512字节，你可以用`ulimit -a`命令来查看你自己机器上这个缓存的大小)
3. 源进程`cat`开始从文件中读取内容并输出到`stdout`。这个数据流通过第一个管道进入到第一个缓存中，而且很快就会到达缓存容量的上限。一旦到达这个上限，`cat`就会被它自己的`write(2)`所阻塞。这是管道的优点之一：`cat`的执行过程隐式的受管道数据处理的能力的控制。(有点类似**协程**的概念，这里每一个进程都充当了一个“协程”)
4. 接下来第一个过滤进程`grep`调用`read(2)`从它的`stdin`管道中读取数据。当`grep`进程刚被创建的时候，这个管道是空的(`cat`进程的输出数据还没有传输过来)，所以`grep`进程会被阻塞，直到有新数据到来。这里我们再一次看到了基于数据处理能力隐式执行控制。`grep`从读取的数据的每一行中进行匹配，并将匹配的单词输出到`grep`进程的`stdout`中，同样通过管道传送给下一个进程
5. `awk`进程的执行过程与`grep`类似。没有数据的时候`awk`进程将被阻塞，有新数据到来的时候，`awk`对数据进行处理并输出到`stdout`中
6. `sort`与之前两个进程稍有不同。因为`sort`涉及到排序，必须在完整的数据上执行。`sort`会把每次从`shell`缓存中读取的数据保存在磁盘上的一个缓存空间中。当`sort`的`stdin`关闭时(意味着没有更多新数据了)，`sort`会进行排序操作，并输出到`stdout`

7. `tail` 与 `sort` 类似，也是需要在完整的数据上执行。不过 `tail` 不需要占用太多的缓存空间，因为 `tail` 只关心输入的最后一行数据
8. `cut` 又相当于一个过滤器，与 `grep` 和 `awk` 类似。
9. `cowsay` 同样一直等待输入，计算输入的长度，并用ASCII码字符绘制出说话的 `Linux` 企鹅

利用管道来执行这个任务非常简单，一个没有编程基础的人都可以轻松完成。每个步骤所用到的数据都经过了上一个任务的过滤，当前数据集在每一步都会改变。正如Unix哲学所倡导的

> Do one thing, Do it well

为了更直观的展示这个过程，我们把上述步骤画在一张图上



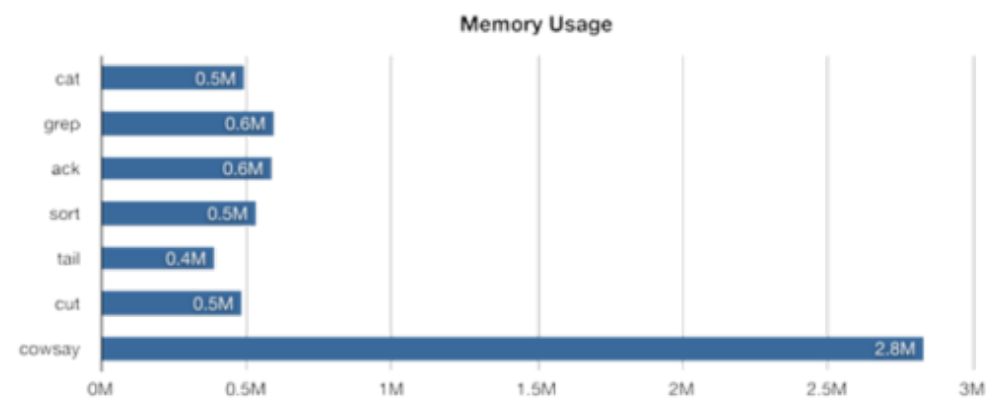
0x02 性能和复杂度

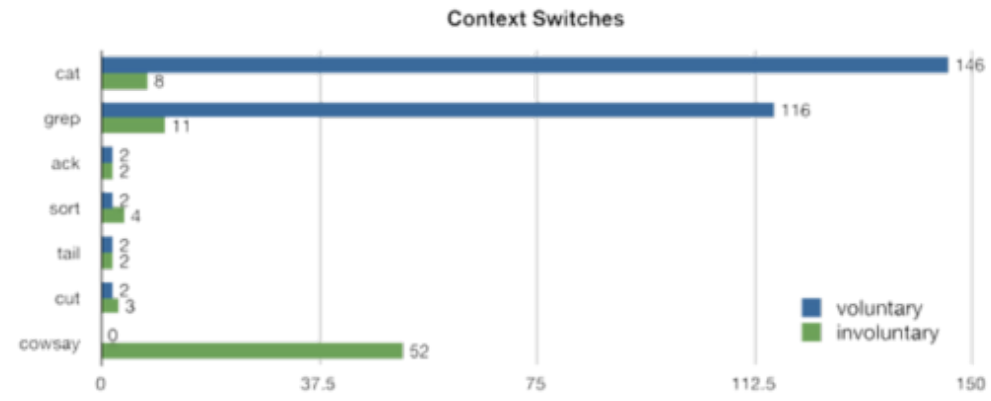
管道的另一个优点就是它天生的高性能。我们对上面的命令稍作修改以观察其中每一个过滤器组件的内存和CPU占用率

```
/usr/bin/time -l cat /usr/share/dict/words 2> cat.time.txt |
/usr/bin/time -l grep purple 2> grep.time.txt |
/usr/bin/time -l awk '{print length($1), $1}' 2> awk.time.txt |
/usr/bin/time -l sort -n 2> sort.time.txt |
/usr/bin/time -l tail -n 1 2> tail.time.txt |
/usr/bin/time -l cut -d " " -f 2 2> cut.time.txt |
/usr/bin/time -l cowsay -f tux 2> cowsay.time.txt
```

(注意：如果你使用的是 `Linux` ，可以使用 `-v` 选项实现同样的效果。 `2> something.time.txt` 会将 `stderr` 重定向到文件中)

在运行完上述命令后，通过查看输出文件里面的相关信息，我们可以画出下面几张图





- 其中内存占用最大的过滤器是`cowsay`，占用了`2,830,336`字节，因为它是用`Perl`实现的(在我的机器上仅启动`Perl`的解释器就要占用`1,126,400`字节的内存)，内存占用最小的是`tail`，只占用了`389,120`字节
- 尽管我们的源文件(`/usr/share/dict/words`)有`2.4MB`，但是大部分的过滤器都没有占用大于源文件1/5的内存，这得益于我们之前所提到的：管道只保存它所能处理的最大数据量，一旦超过这个值，相关进程将被阻塞，直到下一级进程将数据取走并清空缓存。这个特性决定了管道是一个非常节省内存空间的轻量级解决方案，无论处理多大的文件，管道占用的都是一块恒定的内存空间
- 注意到前两个进程`cat`和`grep`有大量自愿上下文切换(voluntary context switches)，这主要是由IO阻塞引起的(毕竟缓存大小只有512字节)。`cat`从磁盘上读取文件时必须将上下文切换到操作系统，然后将读取的内容输出到`stdout`，`grep`从管道读取数据时同样要进行上下文切换。而`ack`、`sort`、`tail`、`cut`并没有太多上下文切换的原因在于它们处理的数据量要小的多，`grep`已经替它们过滤了大量数据，实际交由`ack`处理的数据只有12行，任何一个缓存都可以装下这些数据
- `cowsay`有许多非自愿上下文切换(involuntary context switches)，不过这并不是它所处理的数据太多引起的。前面提到了`cowsay`是用`Perl`实现的，效率与其他相比是较低的，因此这里较多的非自愿上下文切换是进程时间片(time quantum)耗尽引起的

我们只是展示了一个简单的例子，不过设想一下，如果其中某些进程执行的是复杂的计算任务，那么它们将自动在多处理器上并行执行。这么来看的话管道的优势非常明显~

0x03 异常

回顾一下管道的优点，省内存、省CPU时间，基于数据处理能力的隐式执行控制，方便快捷……既然管道有如此多的优点，为什么没有在实际中大规模应用呢？

原因在于异常处理，如果管道的某个中间环节出了问题，那么整个管道就彻底挂掉了。

我们对刚才那个例子稍作修改，加入一个自己用python实现的`fail.py`程序，它把`stdin`的内容直接输出到`stdout`中去，但是它有50%的概率产生异常

```
cat /usr/share/dict/words |     # Read in the system's dictionary.
grep purple |                   # Find words containing 'purple'
awk '{print length($1), $1}' |  # Count the letters in each word
sort -n |                       # Sort lines ("${length} ${word}")
python fail.py |                # Play Russian Roulette with our data!
tail -n 1 |                     # Take the last line of the input
cut -d " " -f 2 |               # Take the second part of each line
cowsay -f tux                   # Put the resulting word into Tux's mouth
```

`fail.py`的源码

```
import sys
import random

while True:
    if random.choice([True, False]):
        sys.exit(1)
    line = sys.stdin.readline()
    if not line:
        break
    sys.stdout.write(line)
    sys.stdout.flush()
```

那么我们来看看异常产生时会发生什么事情。当`fail.py`在读取`stdin`之前就退出时，它的`stdin`和`stdout`管道就关闭了，这相当于把整个管道切成了两半，接下来就会产生一系列连锁反应

1. `sort`进程会接收到一个`SIGPIPE`信号通知它的`stdout`管道被关闭了。此时`sort`可以选择立即处理这个`SIGPIPE`或者重新尝试`write(2)`，但是此时调用`write(2)`只能返回`-1`，因为`stdout`管道已被关闭。所以`sort`无法正常输出，只以选择退出，并关闭它的`stdin`管道，这又会导致`sort`之前的进程依次关闭退出(当然进程并非遇到`SIGPIPE`或者写错误就必须退出，但是这个例子中异常是由进程的输出管道被关闭引起的，所以除了退出没有其他办法)
2. `tail`进程同样收到一个`SIGPIPE`信号通知它的`stdin`管道被关闭，此时`tail`可以选择处理这个`SIGPIPE`信号或者忽略这个信号，但是无论怎样，它下一次调用`read(2)`时将会返回错误，而且无法与正常的流结束区分开来(因为同样都是没有新的数据了)，因此`tail`会认为这是正常的流结束标志，会在现有数据基础上进行操作，并将结果输出到`stdout`
3. `cut`同样不会察觉有异常发生，会在`tail`提供的数据上正常执行

4. `cowsay`会输出在`python`发生异常之前的数据集中含`purple`且最长的单词

结果是什么呢？

```
 _________
< repurple >
 ----------
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/
```

企鹅说的不再是unimpurpled，而变成了repurple，这个结果是错误的！虽然其中一个过滤器发生了异常，我们还是得到结果了，只不过是一个错误的结果，但是更糟糕的事情是当我们查看管道的返回码时，得到了这样的结果

```
bash-3.2$ echo $?
0
```

`bash`显示管道正确执行了，这是由于`bash`把最后一个进程的返回码当作整个管道的状态码返回给我们。如果要查看管道中每一个进程的返回码，要使用到一个很少用到的`$PIPESTATUS`变量

```
bash-3.2$ echo ${PIPESTATUS[*]}
0 0 0 0 1 0 0 0
```

这个数组保存了管道中每一个进程的返回码，只有从这里我们才能发现管道中的一个过滤器发生了异常

这就是传统的Unix管道一个非常大的缺点，如果想在管道处理数据时探测异常需要用到带外数据(out-of-band)信号来检测异常，并将消息发到其他进程(如果你的过滤器有不止一个输入管道这是非常好实现的，但如果你使用的仅仅是Unix管道就比较困难了)

0x04 管道的其他用途

看完了上面的介绍，你可能会有下面的问题

> 管道在现实中如何使用呢？
>
> 能否在我的web app中使用管道呢？

这些对管道来说都不是问题，只要你的任务可以被划分为很小的部分而且处理时可以逐步完成就可以了。下面我们来看几个实例~

音频转码

假设你有许多`.flac`格式的文件——高品质音乐文件，你想把他们放到MP3里面去，但是它不支持`.flac`格式。而且由于某些原因，你的电脑上可用的RAM空间不超10M，这时该怎么办呢？你可以使用管道

```
ls *.flac |
while read song
do
    flac -d "$song" --stdout |
    lame -V2 - "$song".mp3
done
```

这个命令比我们之前用到的更复杂一点，这里用到了内建的`bash`结构——`while`，从输入的每一行中读取文件名(从ls输出的管道中获取)，内层循环中调用了`flac`解码音频文件，然后调用`lame`将其编码为MP3格式。

这个管道的性能如何呢？在一个全部为`.flac`文件，总大小为115MB的文件夹中运行上述命令，只占用了**1.3MB**内存

web app

设想这样一个网页应用，用户提交了一个表单，现在你需要在后端对这个表单进行处理，涉及数据清洗，数据验证，最终保存为一个PDF文件(当然这个例子可能显得有些做作)。这些任务依然可以用管道来完成

```
my_webserver |
line_sanitizer |
verifier |
pdf_renderer
```

当用户将表单提交到`my_webserver`，它会将这些数据转换为一行JSON并输出到`stdout`，假设这个JSON数据是这样的

```
{"name": "Raymond Luxury Yacht", "organization": "Flying Circus"}
```

管道中的下一个进程`line_sanitizer`可以作如下处理

```
import sys
import json

for line in sys.stdin:
    obj = json.loads(line)

    if "Eric Idle" in obj['name']:
        # Ignore forms submitted by Eric Idle.
        continue

    sys.stdout.write(line)
    sys.stdout.flush()
```

下一个进程验证用户填写的organization是否存在

```
import sys
import json
import requests

for line in sys.stdin:
    obj = json.loads(line)
    org = obj['organization']
    resp = requests.get("http://does.it/exist", data=org)

    if resp.response_code == 404:
        continue

    sys.stdout.write(line)
    sys.stdout.flush()
```

最后一个进程把这些信息保存到PDF文件中

```
import sys
import json
import magical_pdf_writer_that_doesnt_exist as writer

for line in sys.stdin:
    obj = json.loads(line)
    writer.write_to_file(obj)
```

不过还有一个问题没有解决，就是我们前面提到的异常处理。比如当某个用户使用了Eric Idle这个用户名或者填写了一个不存在的organization时，我们如何reject这次提交并告知用户错误信息呢？一个非常有Unix特色的方法就是使用**命名管道(named pipe)**来处理所有失败请求

```
mkfifo errors  # create a named pipe for our errors

my_webserver |
```

```
line_sanitizer 2> errors |
verifier 2> errors |
pdf_renderer 2> errors
```

任何进程都可以从我们自定义的命名管道`errors`中读取数据，此外管道中的每个进程都可以把错误信息输出到命名管道`errors`中。我们还可以添加一个reader，当发生异常的时候从命名管道`errors`中读取信息并给我们发一封邮件

```
mkfifo errors               # create a named pipe for our errors
email_on_error < errors &  # add a reader to this pipe

my_webserver |
line_sanitizer 2> errors |
verifier 2> errors |
pdf_renderer 2> errors
```

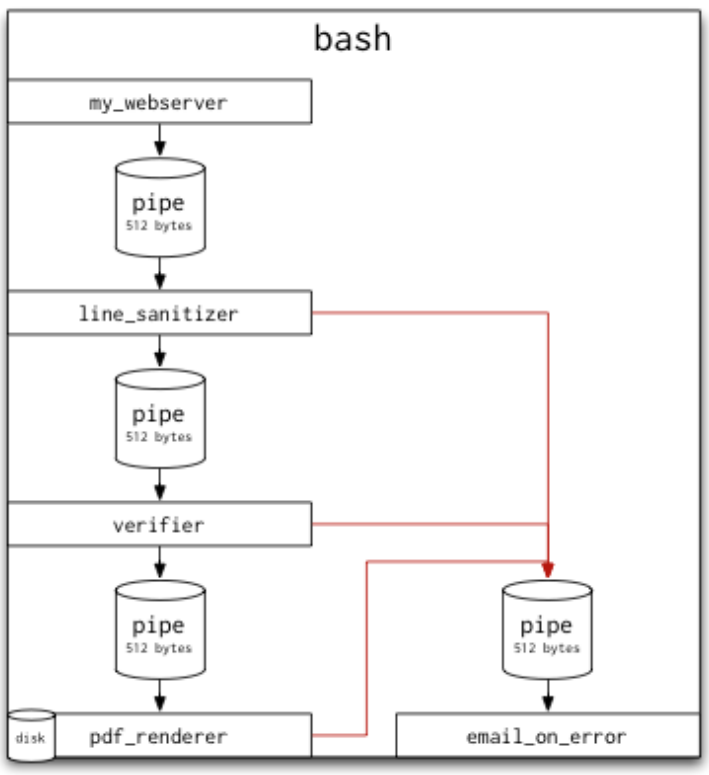
此时，`line_sanitizer`可以这样写

```
import sys
import json

for line in sys.stdin:
    obj = json.loads(line)

    if "Eric Idle" in obj['name']:
        sys.stderr.write(line)
        sys.stderr.flush()
    else:
        sys.stdout.write(line)
        sys.stdout.flush()
```

把这个流程也画成一张图，红色的线代表`stderr`



0x05 分布式管道

Unix管道有很多优点，但也存在诸多不足。并非所有软件都能适用Unix管道模型，但是好在我们有替代方案

现在“工作队列”软件包近几年发展比较迅猛，有些可以支持简单的跨机器之间的FIFO队列。`beanstalk`和`celery`支持在任意进程间创建工作队列。这些都可以简单模拟传统Unix管道模型，而且由于可以在多台机器之间进行，也拥有分布式的优点。但是这些都只适用于异步的任务处理，它们的队列不会阻塞某个进程发送消息，我们前面提到的传统Unix管道模型中隐式的执行控制并没有体现出来，这些解决方案更像是消息系统或者工作队列而并非协程。

因此，我创建了一个基于`Redis`且同步的分布式管道库——`pressure`，`pressure`允许你在不同进程之间建立管道，并加入了管道缓存区的概念，而且支持跨多台机器使用。通过使用`Redis`作为稳定的消息代理，所有进程间通信都可以被`Redis`接管，能保证很好的操作系统和平台无关性。(`Redis`还拥有可靠性和可复用性等诸多优点)

`pressure`是用`python`实现的，目前仍处于起步阶段。为了展示`pressure`的强大之处，我们用`pressure`自带的Unix管道适配器实现本文开始的那个例子。(`put`和`get`是用C实现的两个小程序，充当传统Unix管道和存放

在`Redis`中分布式`pressure`队列之间的桥梁)

```
# Read in the system's dictionary
cat /usr/share/dict/words | ./put test_1 &

# Find words containing 'purple'
./get test_1 | grep purple | ./put test_2 &

# Count the letters in each word
./get test_2 | awk '{print length($1), $1}' | ./put test_3 &

# Sort lines
./get test_3 | sort -n | ./put test_4 &

# Take the last line of the input
./get test_4 | tail -n 1 | ./put test_5 &

# Take the second part of each line
./get test_5 | cut -d " " -f 2 | ./put test_6 &

# Put the resulting word into Tux's mouth
./get test_6 | cowsay -f tux
```

首先要说明的是这是一个非常慢的过程，整个过程结束时通过`Redis`发送的消息共有`235,912`条，这部分的时间开销就有将近4分钟(如果我们让`grep`紧接着`cat`执行，而不是先把数据放到`Redis`中，整个过程可以提速1200倍)，但是不管怎样，最终我们得到了一个正确的结果

```
 _____________
< unimpurpled >
 -------------
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/
```

`pressure`依然保持了传统Unix管道节省空间的特性，我们可以用`Redis`命令行工具查看内存使用情况

```
$ redis-cli info | grep memory
used_memory:3126928
used_memory_human:2.98M
used_memory_rss:2850816
used_memory_peak:3127664
used_memory_peak_human:2.98M
used_memory_lua:31744
```

`pressure`仍在开发阶段，现在还不能大规模部署，希望大家都能来用一用，提出宝贵的建议~

0x06 总结

到这里全文就结束了，下面还是谈谈我个人的感受吧~

1. 以前对管道什么的还真不太了解，主要还是Unix、Linux系统用的太少了，但是读完这篇文章涨姿势了~
2. 管道的精髓不在于某个应用，而是它的哲学思想，”Do one thing, Do it well”，影响了一代又一代的软件架构，今天的云计算中依然可以看到它的身影
3. 原作者开发的`pressure`是一个非常有意义的探索，但能否在实际中大规模应用还有很长的路要走
4. `Redis`是一个非常强大的Key-Value存储系统，现在很多大型系统架构都在用~
