

系统设计

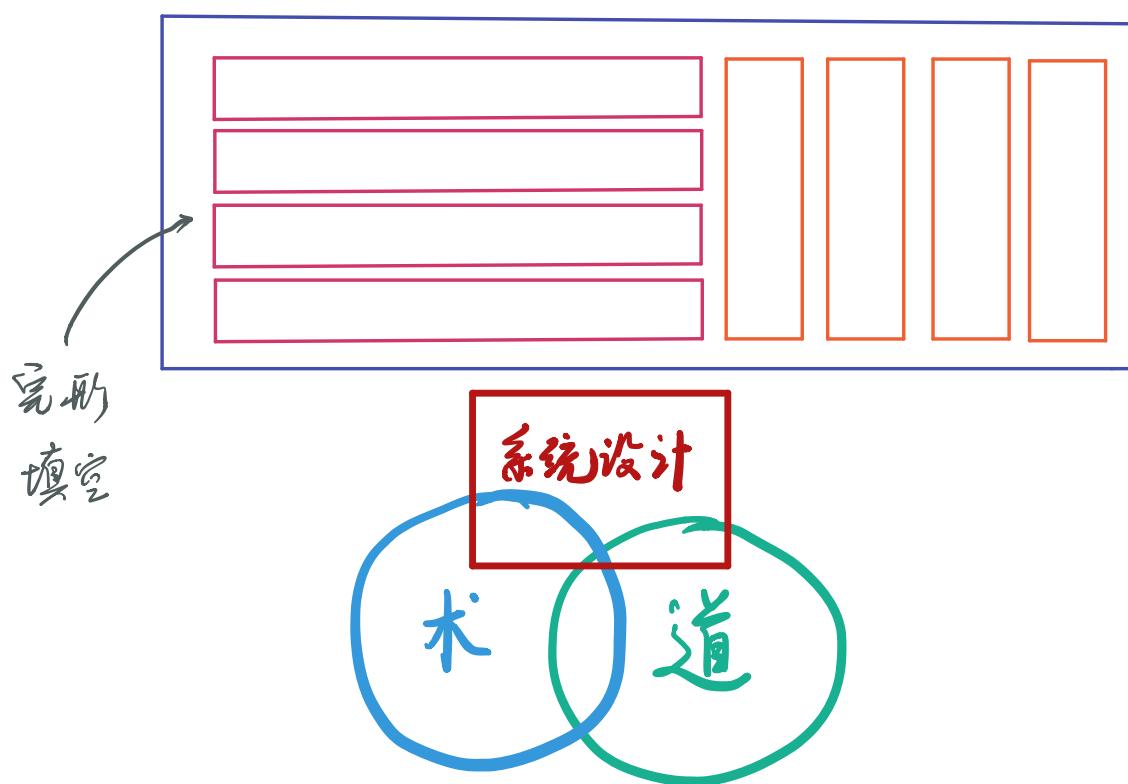
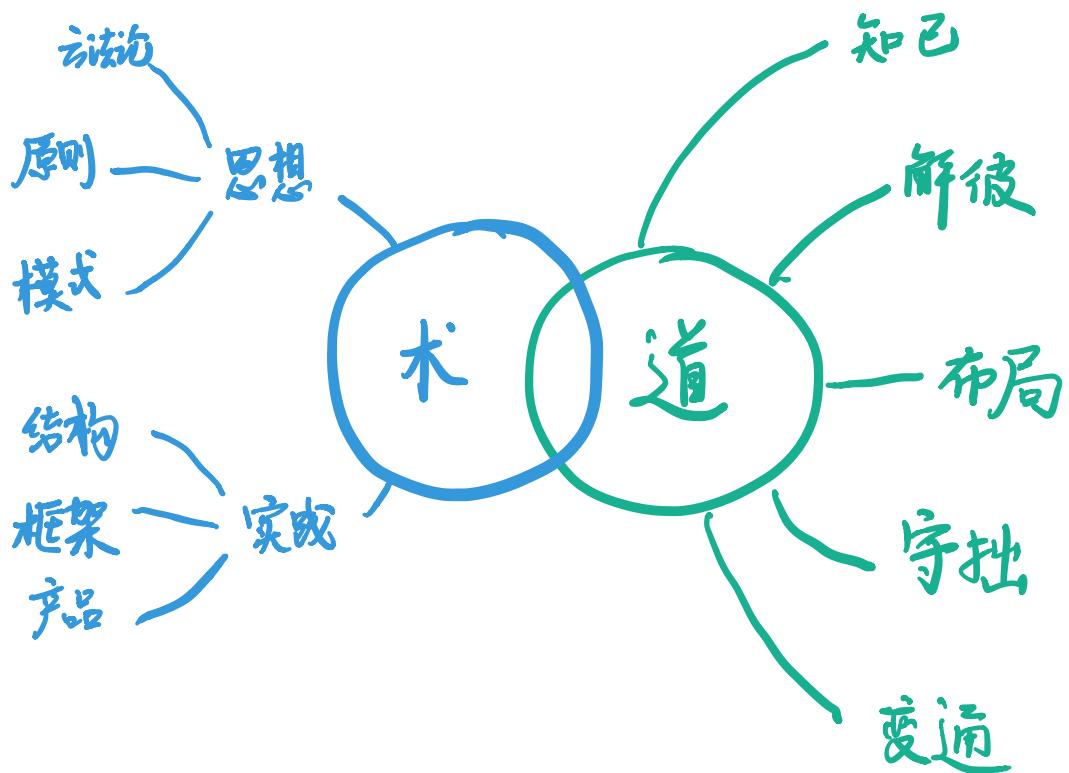
陈天

关于我

2

- 十五年职场生涯：networking/security/distributed systems/internet
- 一本书，一次半创业，两个孩子，三百篇文章，五万粉丝
- 跑步爱好者，全马最佳成绩 4'40"
- 目前在旧金山创业公司 Tubi TV 任 VP Engineering



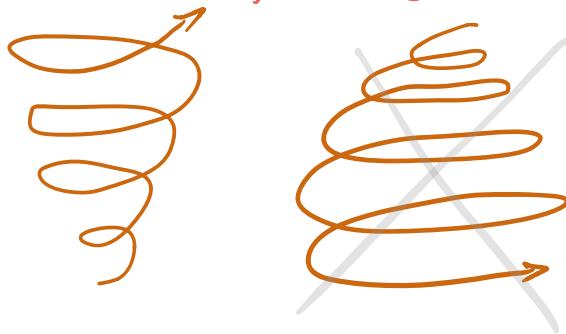


知已
道
而彼布
局
守拙
達通

知己

- 胜: { 知可以战, 不可以战
识众寡之用
以虚待不虚
以逸待劳
- 知人者智
自知者明

- 信: belief is self-fulfilling prophecy.



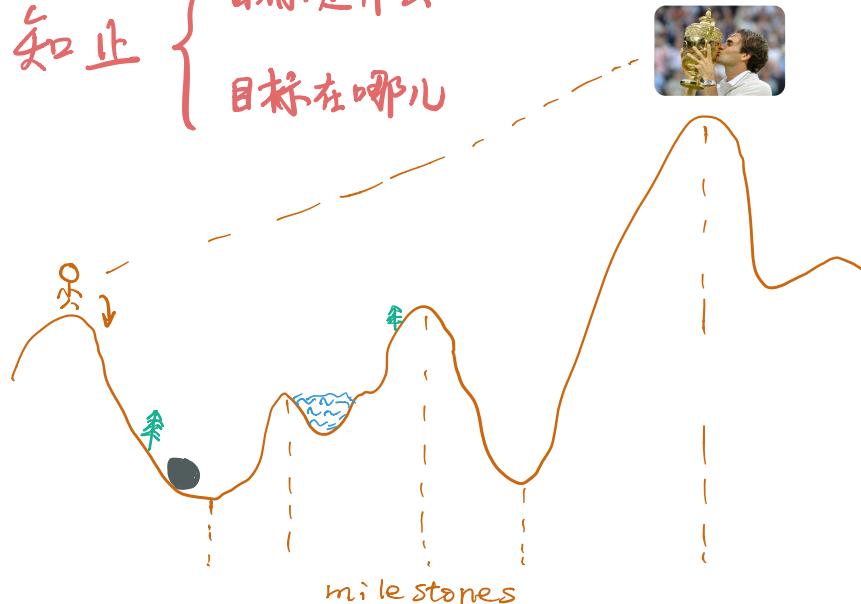
解彼

- Know thy enemy!

需求分析

{
要求
约束条件
场景
变与不变
优先级
日程}

- 知止 {
目标是什么
目标在哪儿}



• 谋定而后动.

hammock-driven development



想清楚 { 需求真的清晰了么?
需求背后的 idea 是什么?
可以从需求中寻找出什么公共的组件?
Spec 该长什么样?

Ex: 设计一个类似于 twitter 的系统 (bullshit!)

WTF ₁	WTF ₂ - pub/sub
API persistent layer pub-sub system notification ad system ⋮ UI?	SLA* user base hot/cold % Limitation*

* SLA (hypo):

- service response time
 - 99th-percentile: 10s
- availability: 99.9% (?)

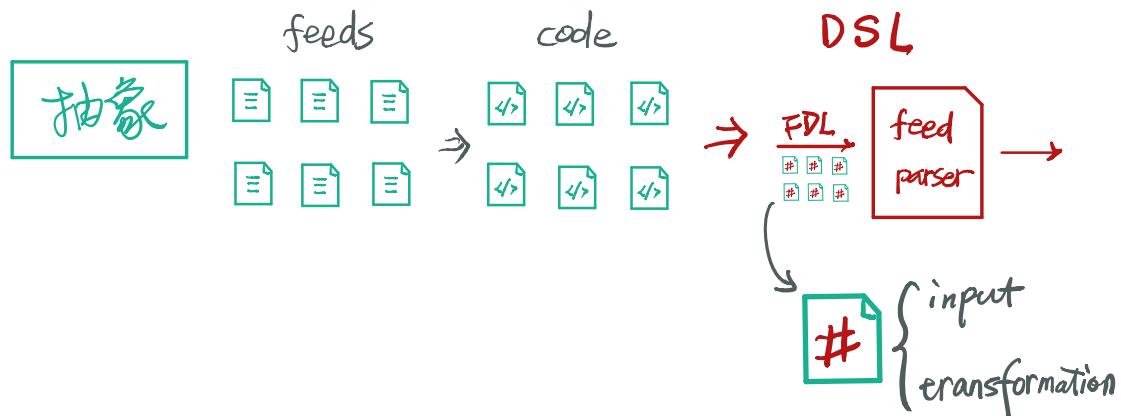
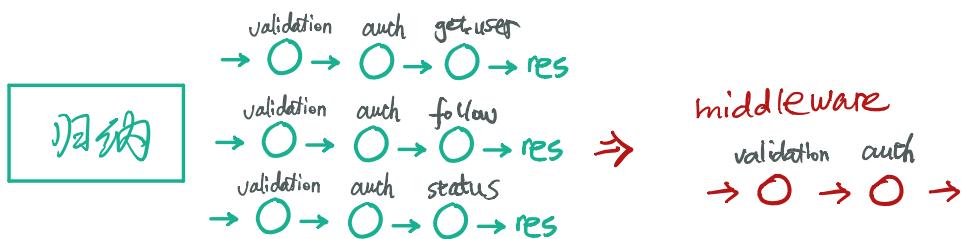
* limitation (hypo):

- 40 words
- 500 follow limit

constraints MATTER !

怎么想

类比 $g(x) \rightarrow f(x)$
 归纳 $f(1), f(2) \dots \rightarrow f(x)$
 抽象 $f(x) = f_1(x) \cdot f_2(x) \cdot \dots \cdot f_n(x)$



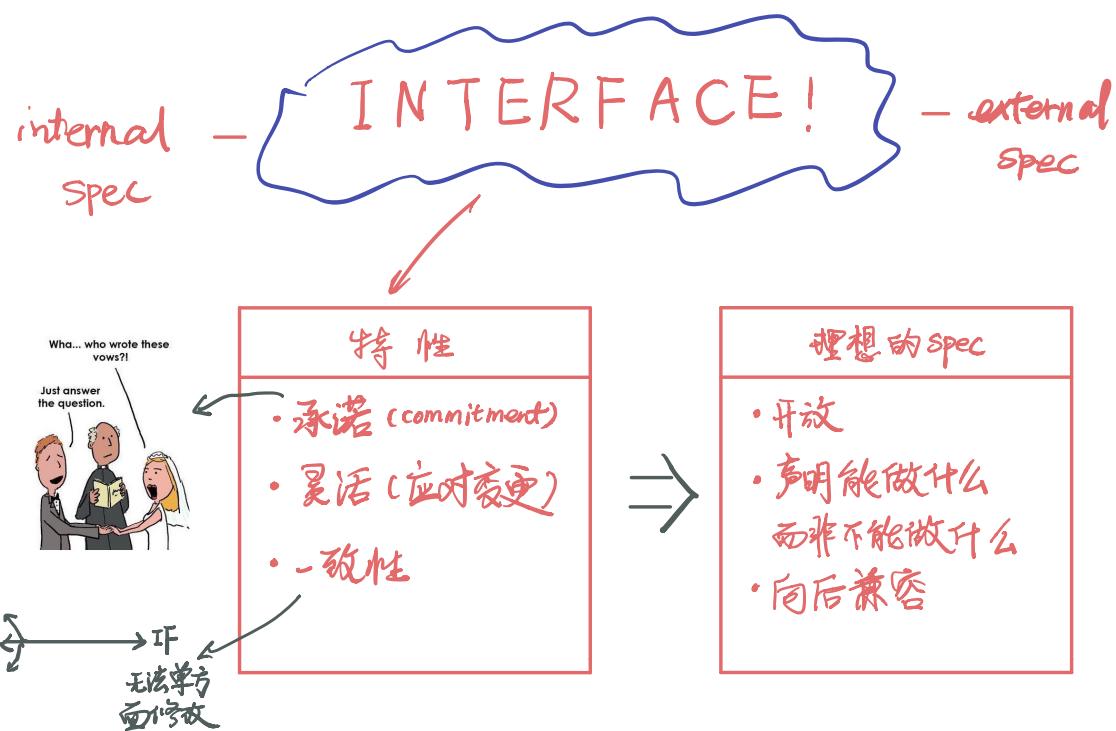
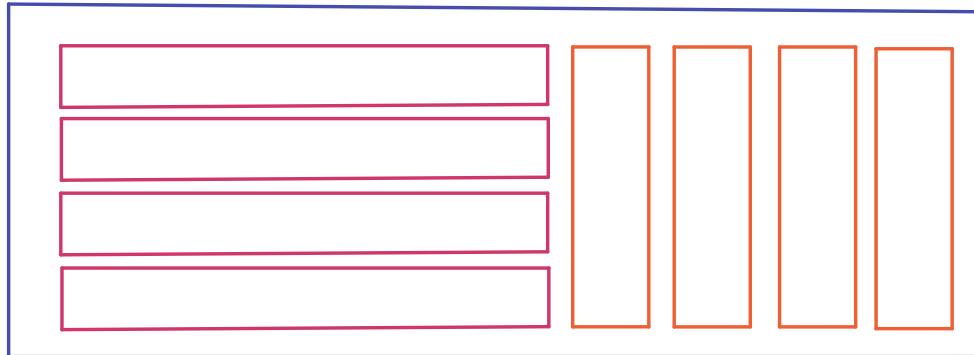
任何问题(需求)都可抽象成一个 DSL

布局

博弈之道，贵乎谨严。高者在腹，下者在边，中者占角。

—棋经·合战篇

what's important here?

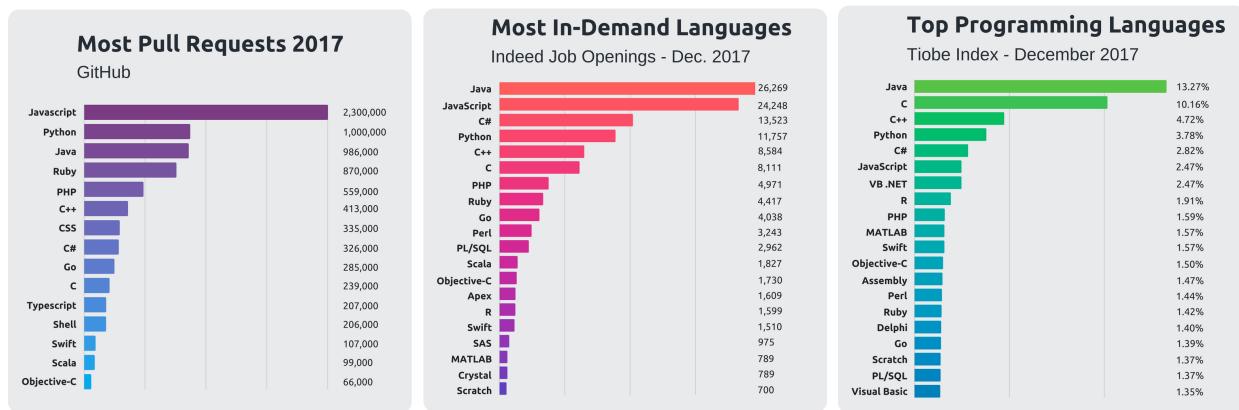


守拙

好于愚拙，不学巧伪。

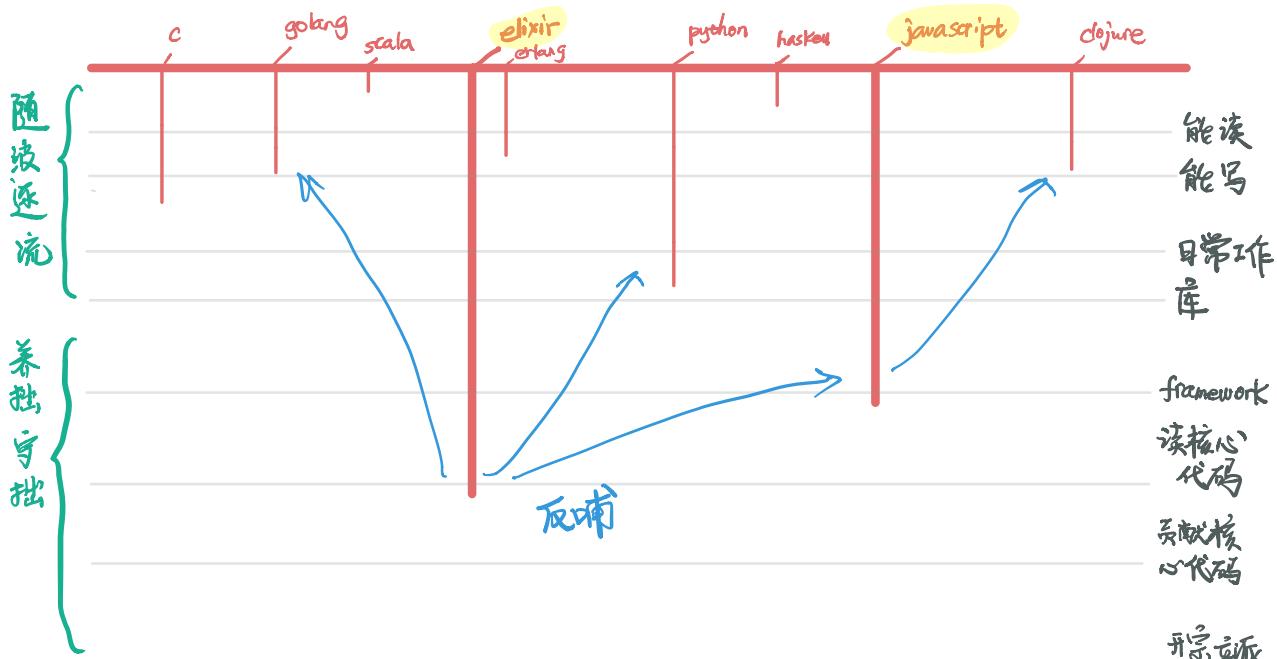
故君子与其练达，不若朴鲁；与其曲谨，不若疏狂。

- 菜根谭



数据库, MQ, frameworks 都是如此 - 百家争鸣，各领风骚。

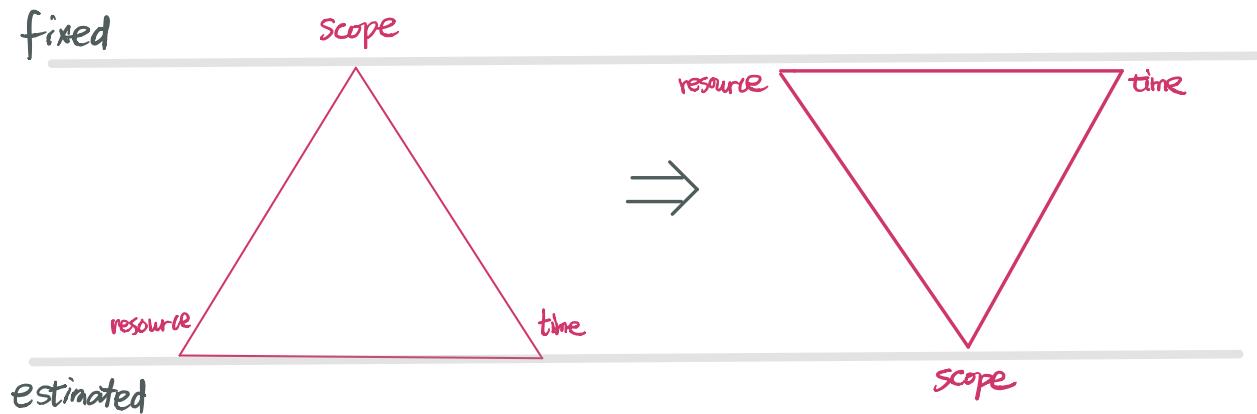
Ex: 编程语言进阶



变通

富则变，变则通，通则久。是以“自天祐之，吉无不利。”

— 家经



变量 { 功能特性
优先级
约束条件
复杂度 < 时间
空间 } don't get me wrong !

Ex: 任务分发

层层优化 ↓

ver 1: $O(N)$

ver 2: $O(\log N)$

ver 3: $\approx O(1)$

ADAPTERS
适者生存？

Begin with end in mind. — 高效能人士习惯

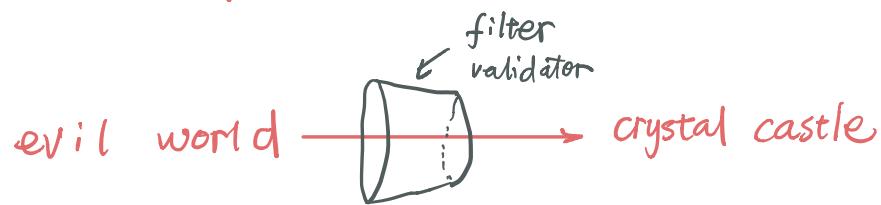
思想、方法论、原则、模式

实践、构架、产品、结构、框架

术

方法论

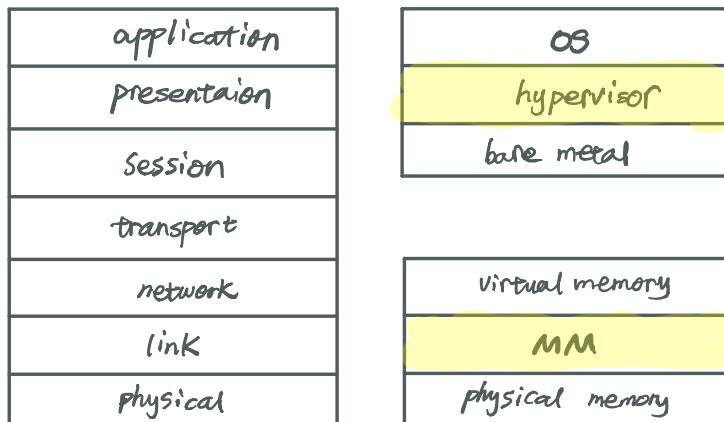
- Isolation : 隔离



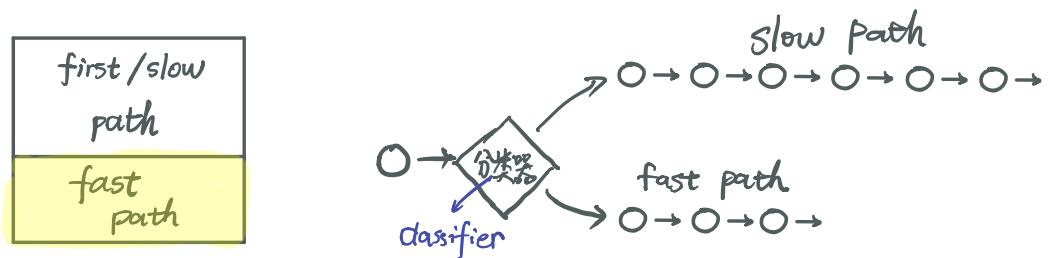
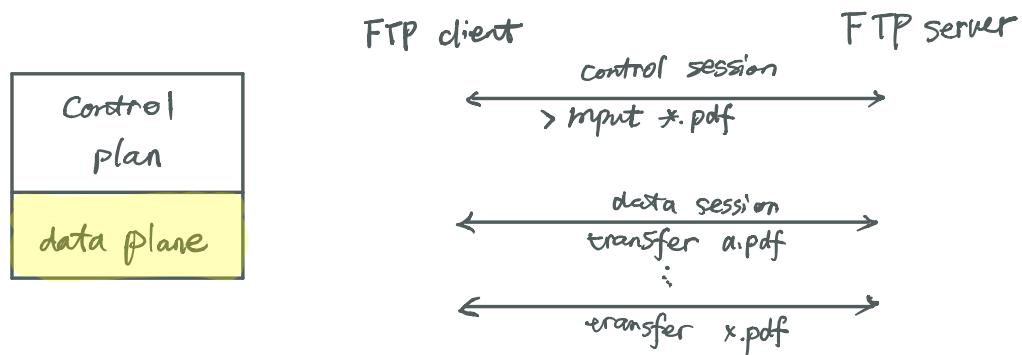
side effect $\cancel{\longrightarrow}$ pure function

constant changing $\cancel{\longrightarrow}$ unchanged

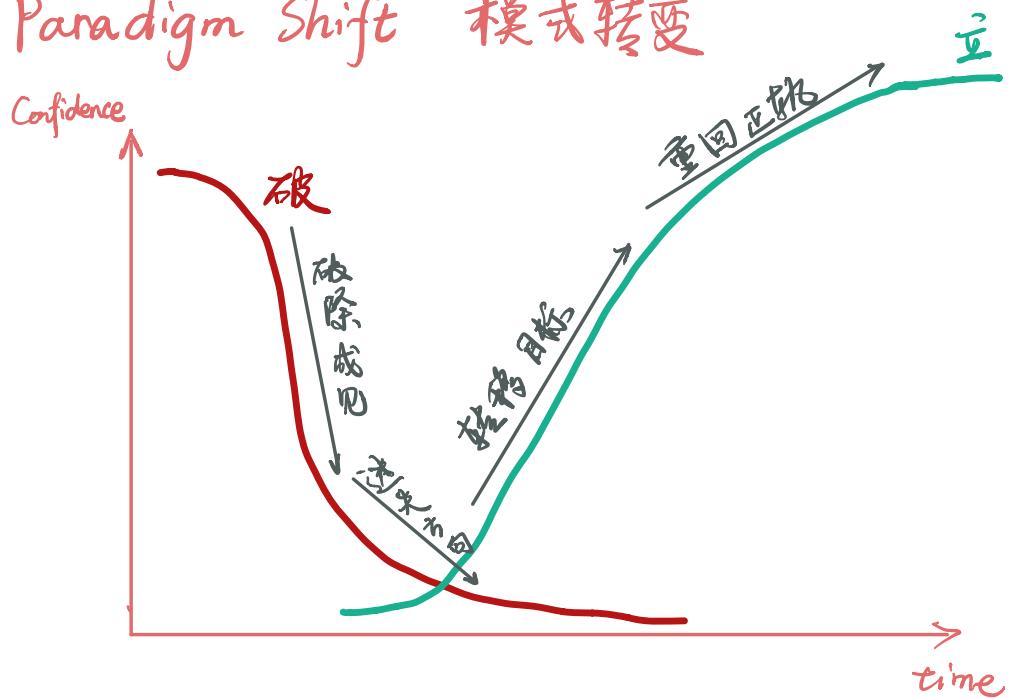
- Layering : 分层



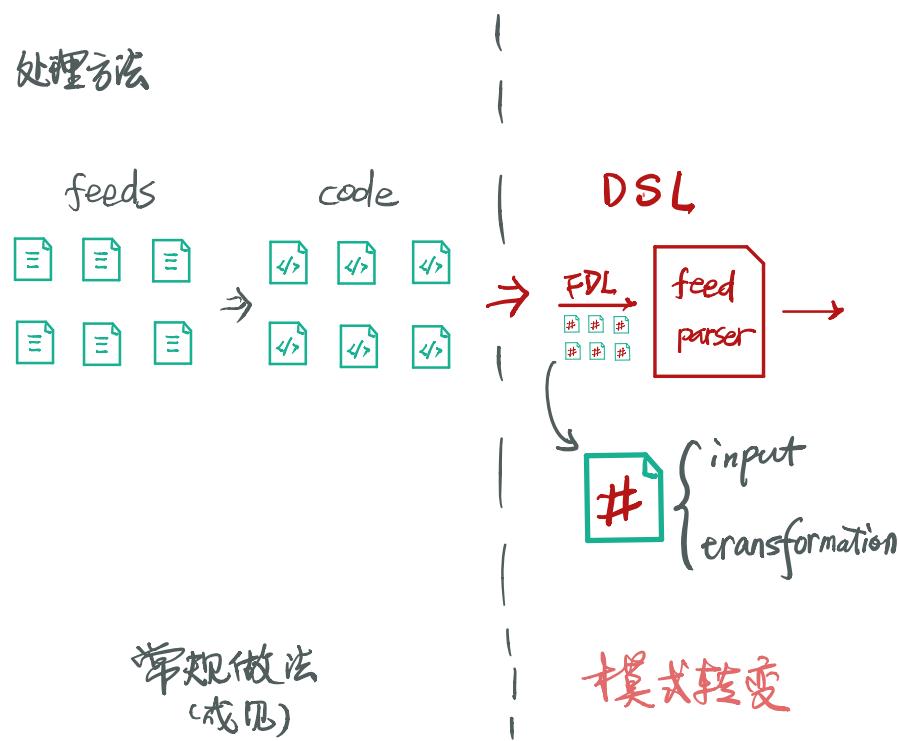
Every problem in computer science could be solved by another layer of indirection.



• Paradigm Shift 模式转变



Ex1: 处理方法



Ex2: 时序



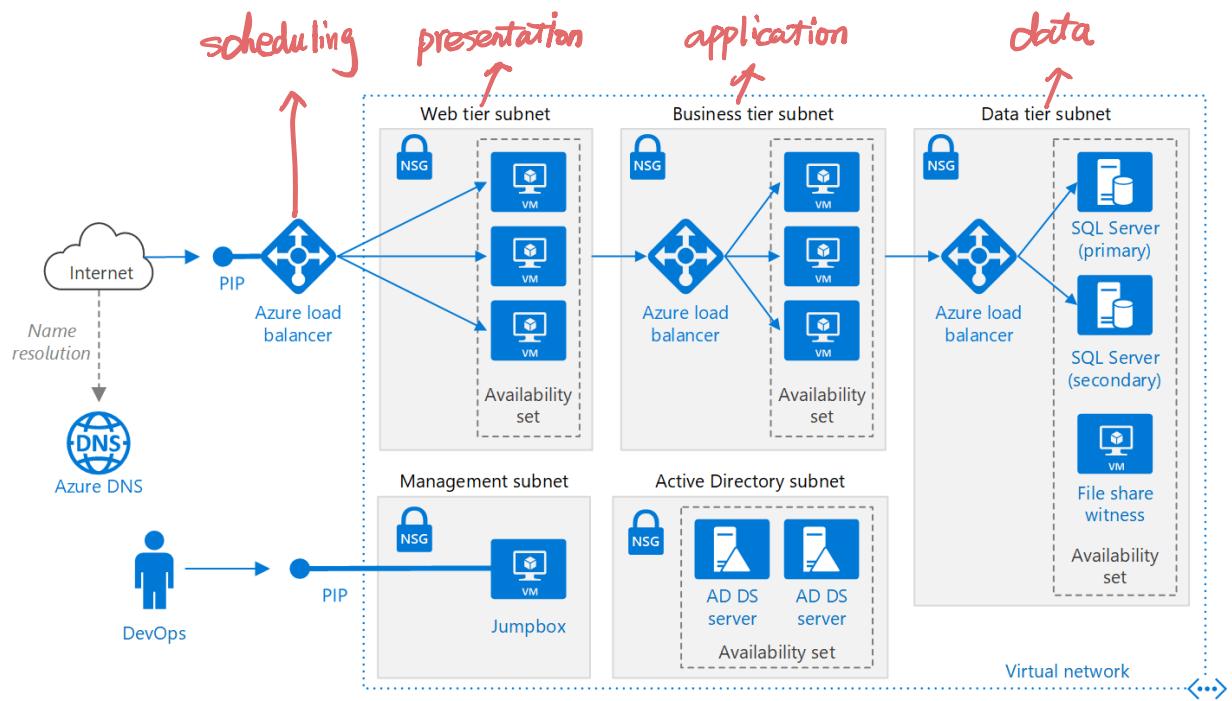
摘自程序员人生 (Programmer Life): 抽象的能力

原则

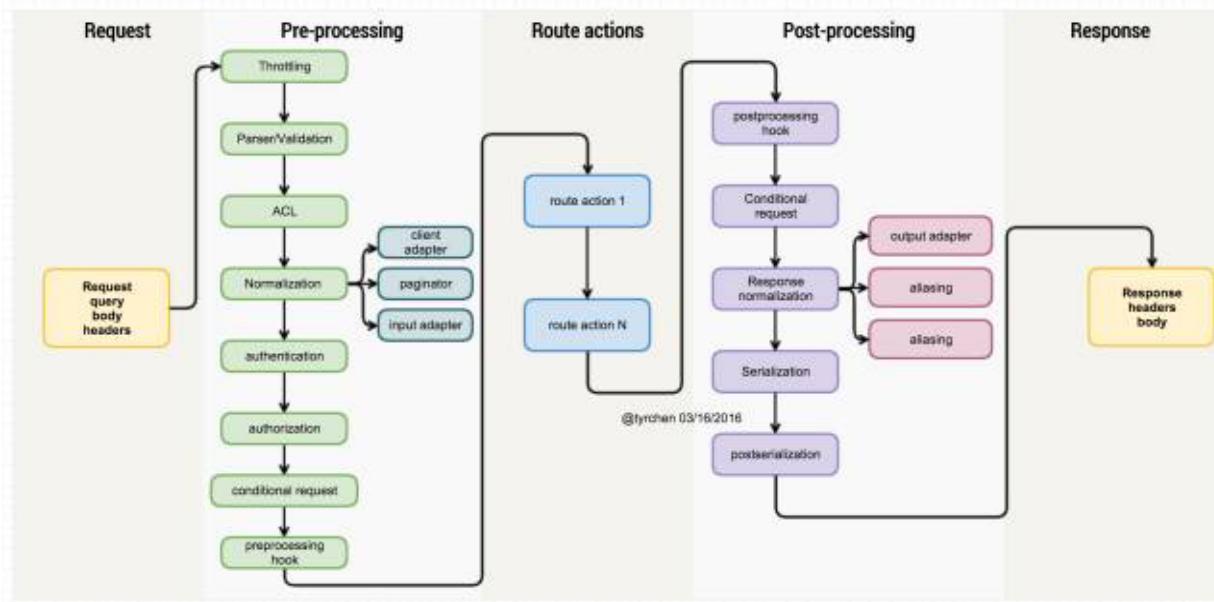
- OCP : Open-Close Principle
- DRY: Don't repeat yourself
- SoC: Separation of Concerns
- IoC: Inversion of Concerns
- CoC: Configuration over Convention

模式

- N-Tier

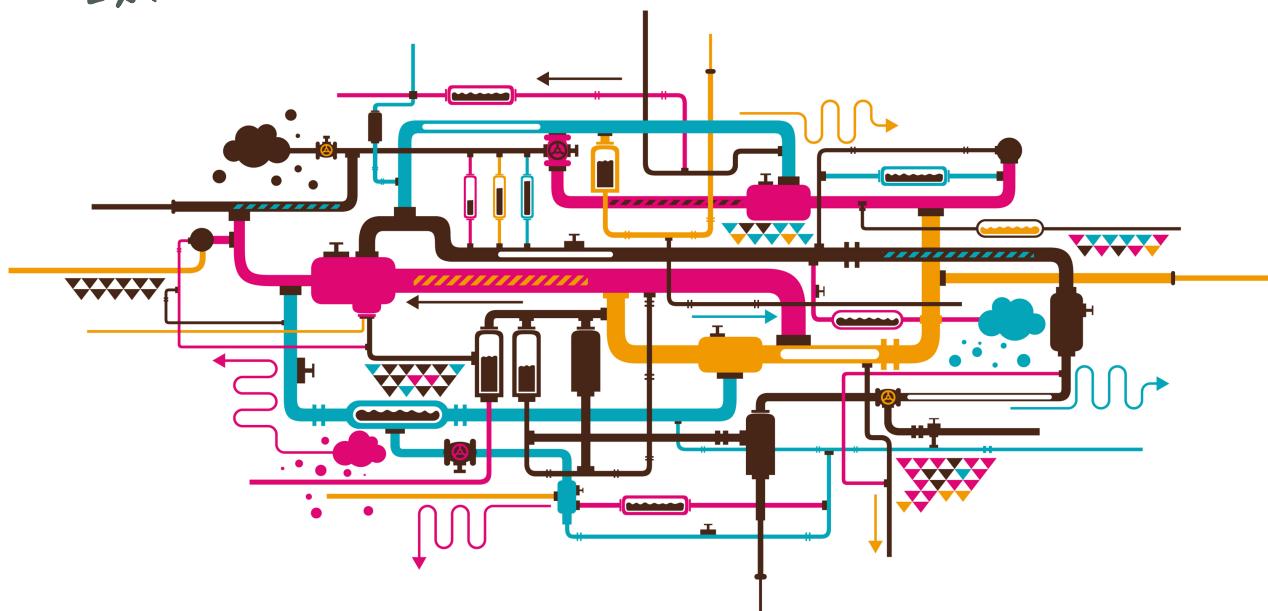


• Pipeline

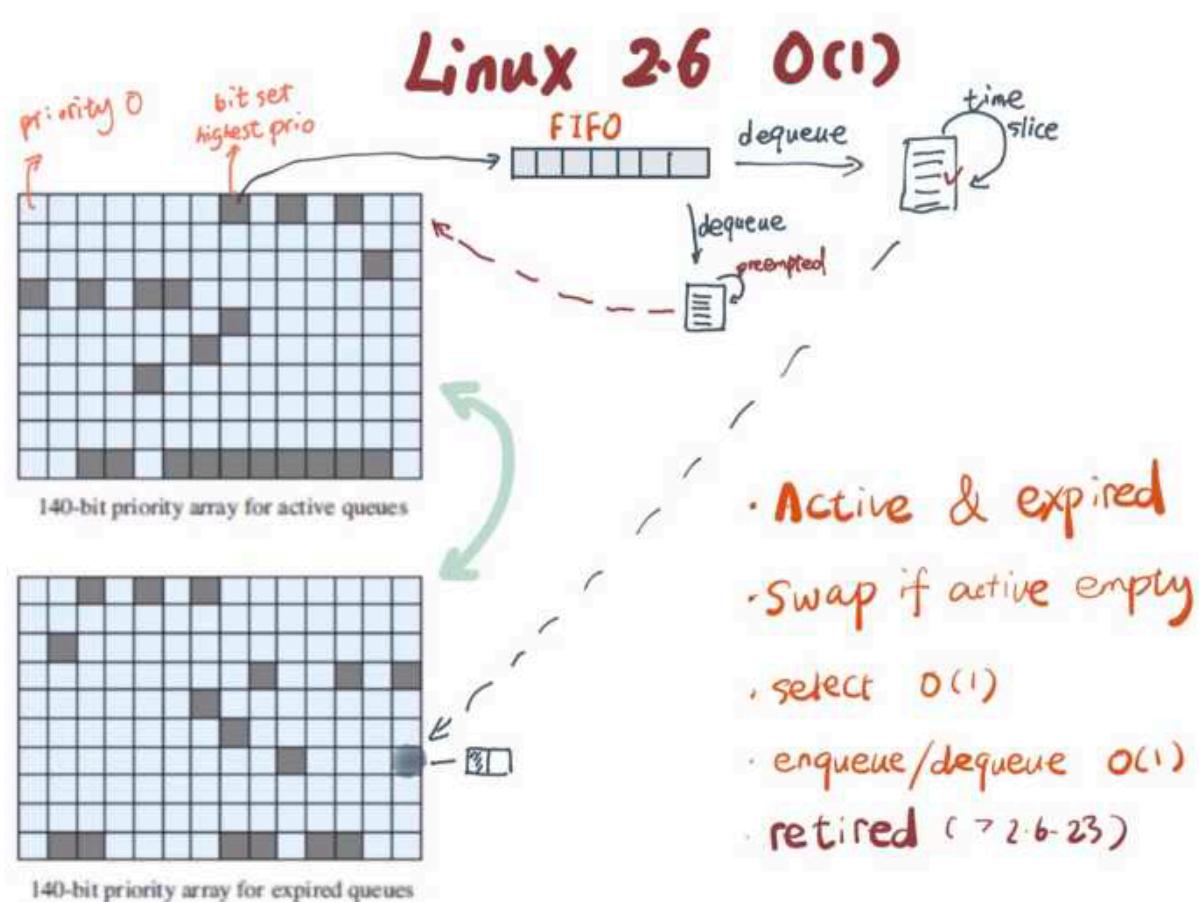


摘自程序人生：再谈 API 的撰写 - 架构

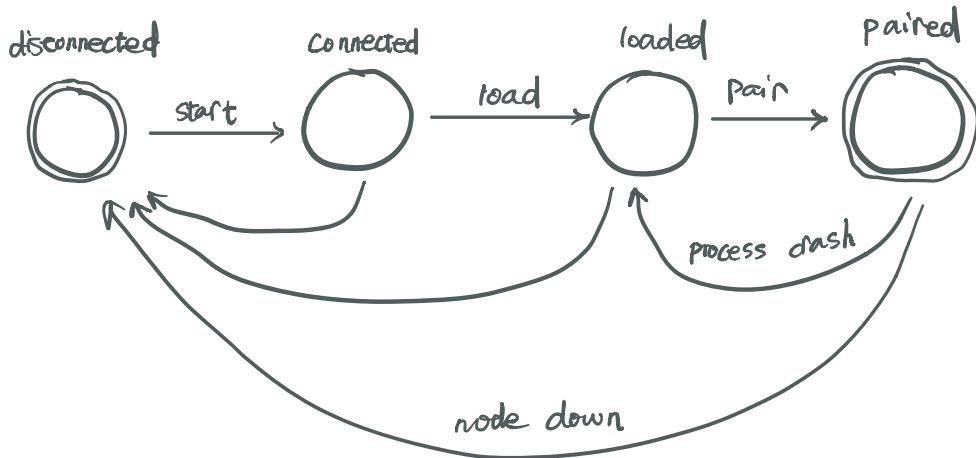
Ex:



• Queue / scheduling



• FSM



谈谈状态机

(原创) 2017-07-27 陈天 程序人生

题记：上周做 BBL 里讲了我们 Tubi TV 内部做 DSL 的一些简单实践，大家反馈不错。有同事建议我给大家先补补 FSM，之后再进阶 CFG，可能会更顺畅些。想想也是。于是我自个花了一两个小时，重温了一些课件。马上要回过了，做 BBL 是三周后的事情了，就没先忙写 slides，写了篇文章。本欲留作他用，考虑再三觉得不合适，干脆在公众号上发出来。这篇文章有些干，看看能有多少阅读（我估计也就 3000+），会掉多少粉。

在谈论一般意义的状态机时，我们先看看有限状态机，Finite State Machine，简称 FSM。

在计算理论（Theory of computation）中，FSM 是一切的基础，也是能力最为有限的机器。在其能力之上是 CFL（Context Free Language），然后是 Turing Machine。

FSM 解决一个输入序列，经过 FSM，最终停留在什么状态这样一个问题。对于一个字符串是否以 \0 结尾（C 语言的字符串结构），FSM 可以给出答案。

CFL 是一切编程语言的基础。你写的一段 python 代码是否语法正确，CFL 能够给出答案。

Turing Machine 就是我们日常用各种算法写代码解决各种问题的基础。不较真地说，JVM 就是一个 Turing Machine。

再往上，就是未知的世界——Turing Machine 也解决不了的问题。

如果你工作多年，已经把 FSM 的知识还给了老师，不打紧，程序君帮你简单复习一下。

一个 FSM 首先有一系列的状态（state）。根据输入的不同，FSM 从一个状态切换到另一个
状态 在这个状态由 右—此状态且特殊的状态 —— 接着状态（next state） 加里输入



微信扫一扫
关注该公众号

“Fundamentals, fundamentals, fundamentals. You've got to get the fundamentals down because otherwise the fancy stuff isn't going to work.”

• CFG

The diagram illustrates the process of parsing Nginx configuration files. On the left, a snippet of Nginx config (server block) is shown. A red arrow labeled "语法分析" points from this code to a middle section containing a parser grammar (Yacc/BNF). Another red arrow labeled "语法分析" points from the grammar to a snippet of lex/bison code on the right, which defines tokens like SERVER, LISTEN, NAME, etc., and handles various Nginx directives.

```
1 server {
2   listen 80;
3   server_name auth;
4   access_log /var/log/nginx/auth.access.log;
5   error_log /var/log/nginx/auth.error.log;
6   location / {
7     proxy_pass http://localhost:3000;
8     include /etc/nginx/proxy_params;
9   }
10 }
```

```
5 %%
6 "server"      {return SERVER;}
7 "listen"       {return LISTEN;}
8 "server_name" {return NAME;}
9 "access_log"  {return ALOG;}
10 "error_log"   {return ELOG;}
11 "location"    {return LOCATION;}
12 "proxy_pass"  {return PROXY_PASS;}
13 "include"     {return INCLUDE;}
14
15 "{"           {return OP;}
16 ")"          {return CP;}
17 ";"           {return TERMINATOR;}
18 [ \s\t\r\n]    /* ignore */
19 [A-Za-z0-9.\.:_-]+ {printf("%s\n", yytext); return VALUE;}
20 %%
```

```
13 %%
14 server: SERVER OP exp_list CP
15 ;
16 exp_list:
17 | subexp exp_list
18 | exp TERMINATOR exp_list
19 ;
20 subexp: LOCATION path OP exp_list CP
21 ;
22 exp: keyword value
23 ;
24 path: VALUE
25 ;
26 keyword: LISTEN
27 | NAME {printf("\n%s: \"%s\"\n");}
28 | ALOG {printf("\n%s: \"%s\"\n");} /* access_log */
29 | ELOG {printf("\n%s: \"%s\"\n");} /* error_log */
30 | PROXY_PASS {printf("\n%s: \"%s\"\n");} /* proxy_pass */
31 | INCLUDE {printf("\n%s - need to read file and put it here: \"%s\"\n");} /* include */
32 value: VALUE
33 ;
34 %%
```

如何愉快地写个小parser

陈天 · 2年前

(一)

在前几日的文章『软件随想录』里，我随意写了一句：「现在似乎已经不是lex/yacc或bison/flex的时代了。我亲眼看见一个同事在费力地用perl一行行解析某个系统的数据文件，却压根没想到写个BNF。BNF对他来说，不是一种选择。」

很多同学不解，问我：lex/yacc不是写编译器[1]的么？我又不发明新的语言，它们对我有什么用？

从这个问题里，我们可以见到国内本科教育荼毒之深。象牙塔里的讲编译原理的老师们，估计用lex/yacc也就是写过个毫无用处的toy language，然后把自己的二知半解传递给了他们的学生，学生们学得半通不通，兴趣索然，考完试之后便把死记硬背的内容如数奉还给了老师。

别笑，我还真就是这么过来的。我用lex/yacc干的唯一一件事，就是TMD设计一个语言。

这世间的语言如此之多，实在容不下我等庸人再设计一门蹩脚的，捉急的，没有颜值，没有性能的语言。况且2000年左右的时候还没有LLVM这种神器，也没有github这样的冥想盆去「偷」别人的思想，设计出来的蹩脚语言只能到语法分析这一步就停下来，没有任何实际用处。

后来lex/yacc进化成flex/bison，在工作中我也无意中翻看了一本orelley叫『Flex & Bison』的书，这书的副标题赫然写着：text processing tools。

书的内容还是挺教条的，和实际的工作 [2] 不太一样，可text processing tools这个说法戳中了我：是啊，词法分析 - lexical parsing (lex/flex) . 语义分析 - grammar

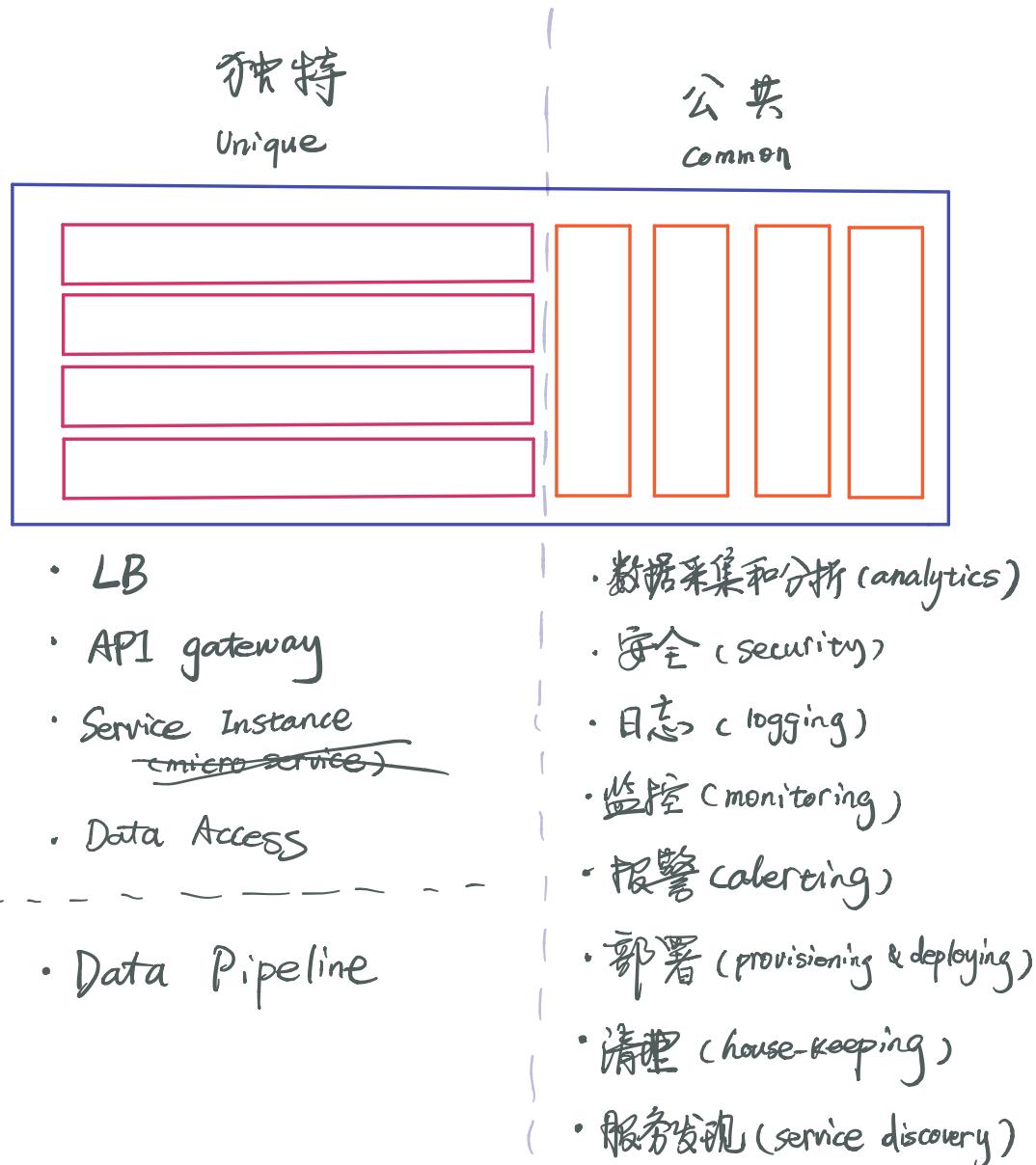
App 内打开

思想、方法论、原则、模式

实践、构架、产品、结构、框架

术

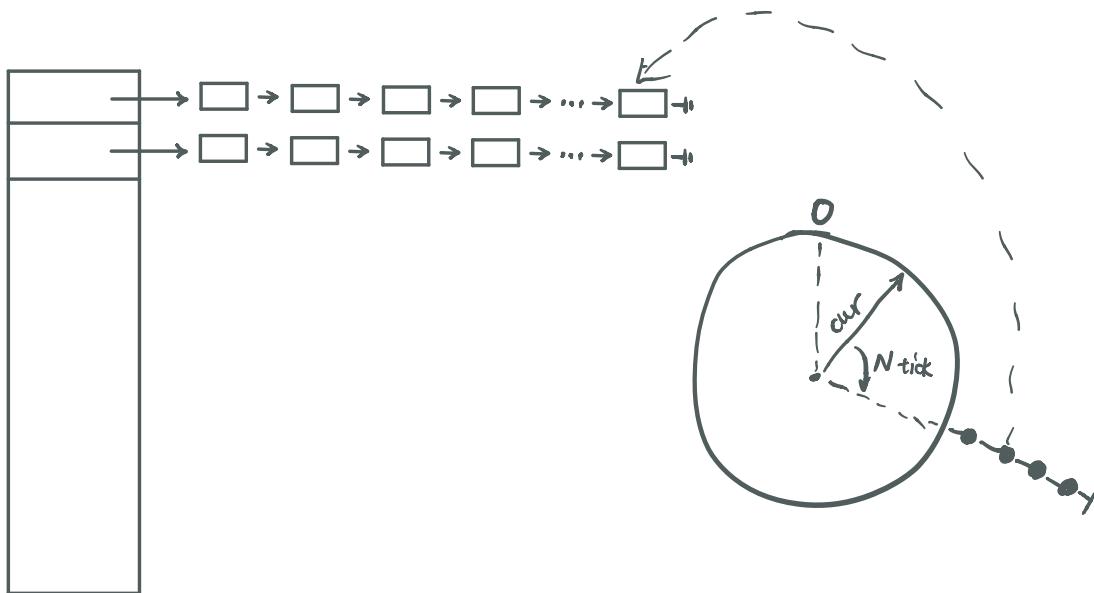
结构



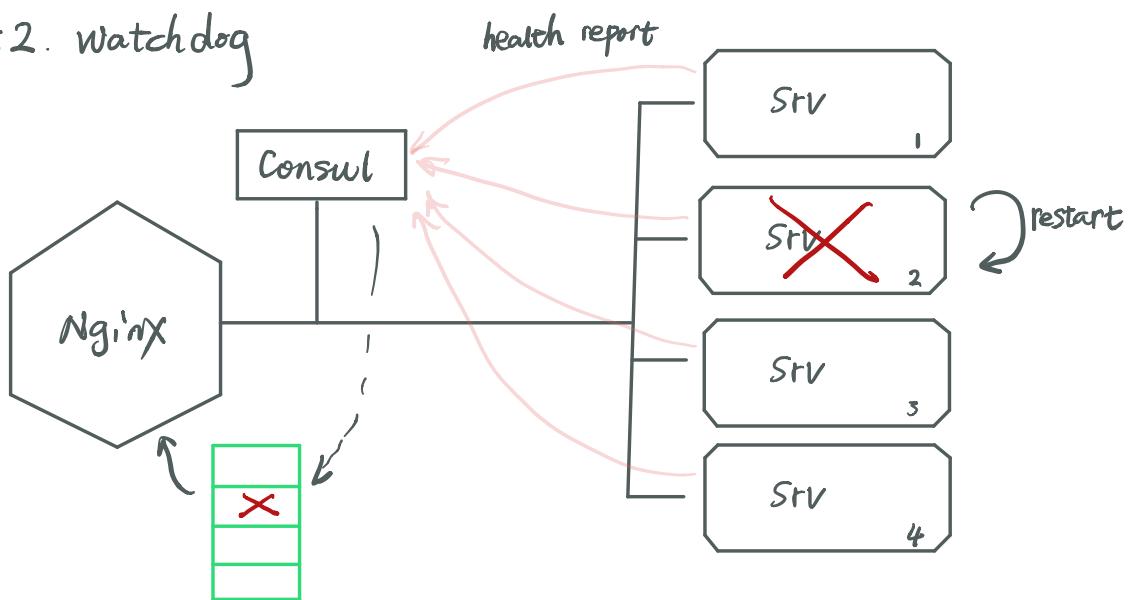
- Sweeper (ager, watchdog, ...)

系统中做 house-keeping 的部分。

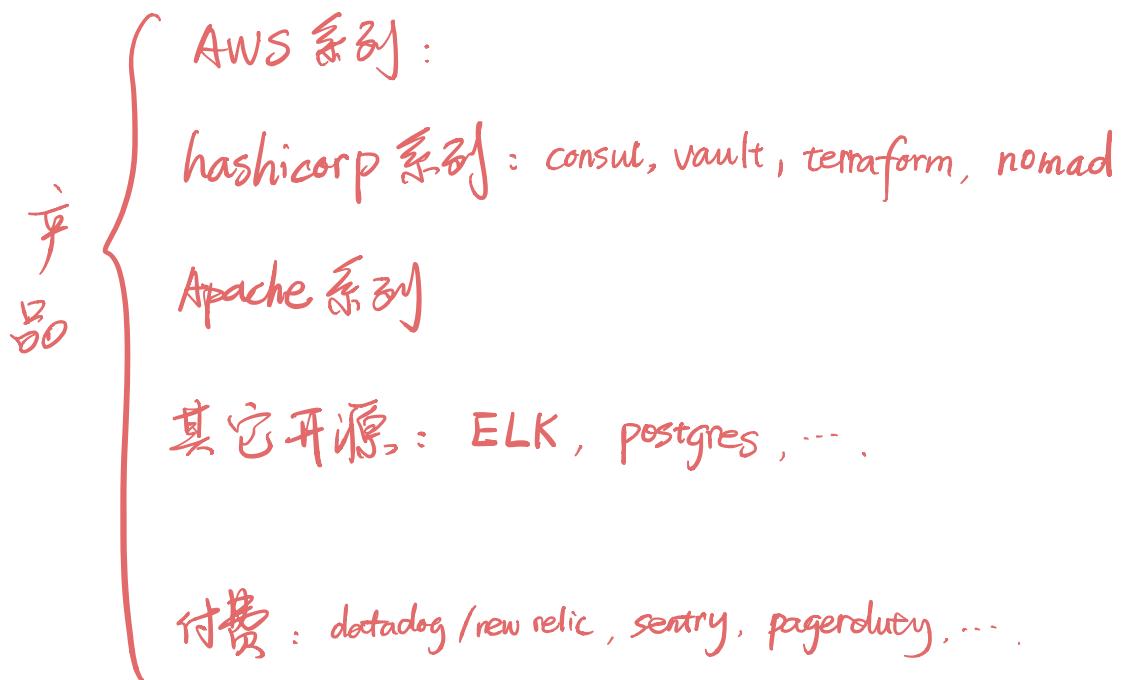
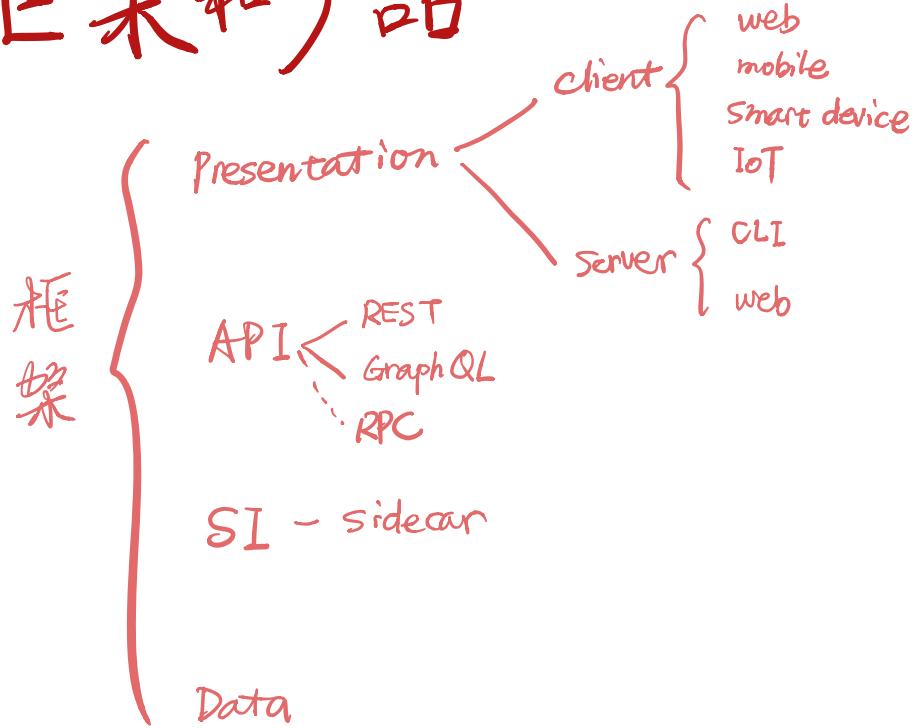
Ex 1: ager ring



Ex 2. watch dog



框架和产品



举几个栗子

谢谢大家



程序人生公众号