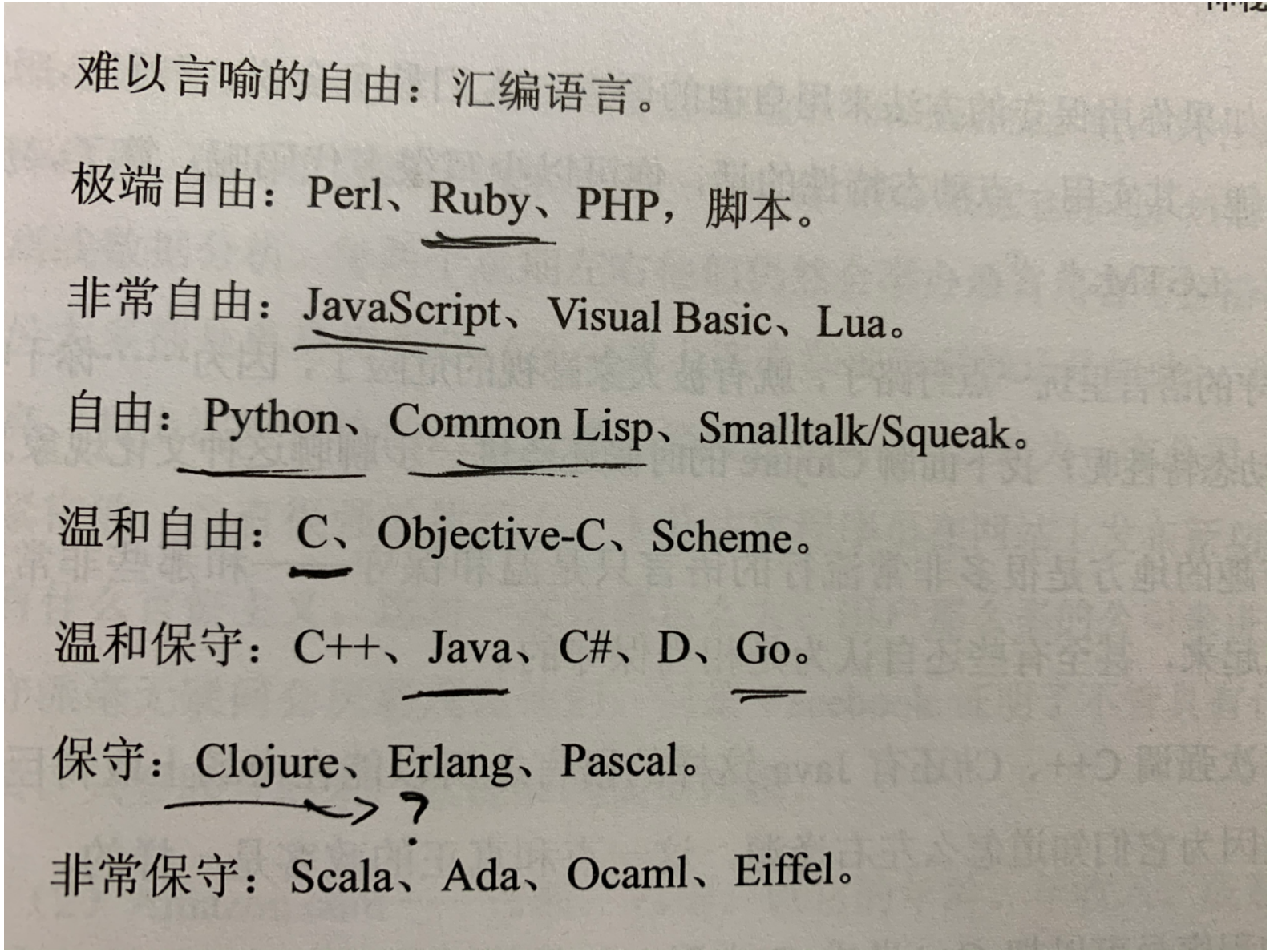
不是很系统, 想到哪讲到哪

“漫谈” 技术选型

@wklken

自由 vs 保守



难以言喻的自由：汇编语言。

极端自由：Perl、Ruby、PHP，脚本。

非常自由：JavaScript、Visual Basic、Lua。

自由：Python、Common Lisp、Smalltalk/Squeak。

温和自由：C、Objective-C、Scheme。

温和保守：C++、Java、C#、D、Go。

保守：Clojure、Erlang、Pascal。

?

非常保守：Scala、Ada、Ocaml、Eiffel。

重构的时候

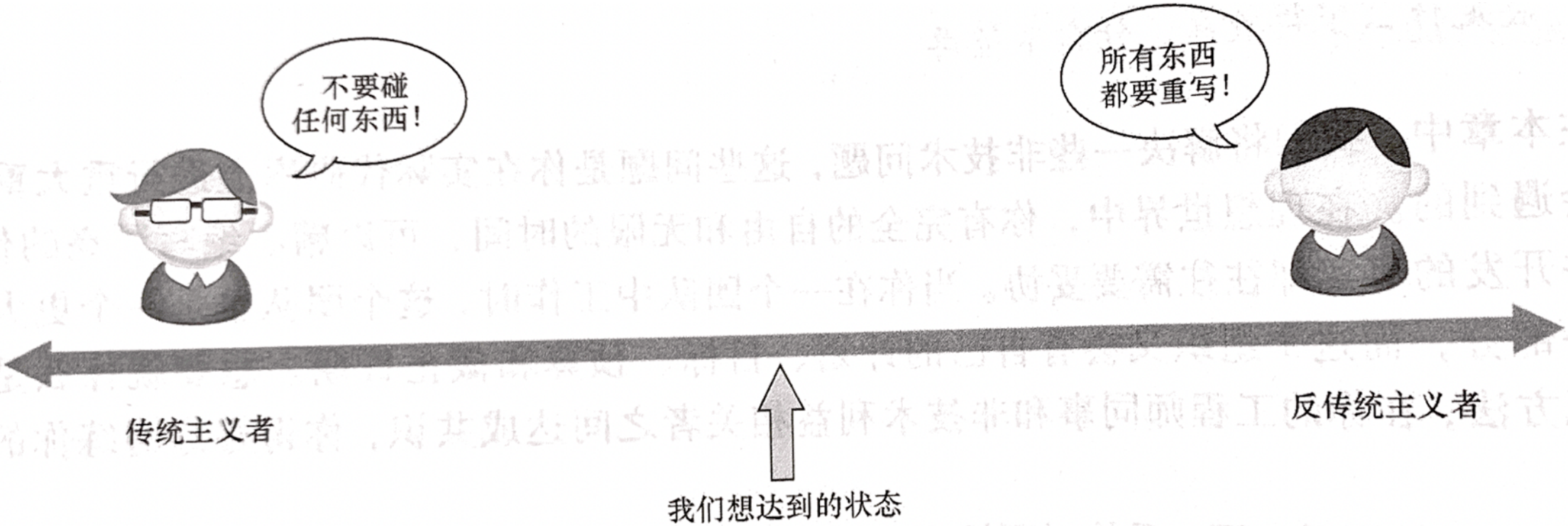


图 3-1　开发人员对遗留代码的态度图谱

你是什么风格?

克制的激进派

架构设计

- 演进式设计: 系统的设计随着系统实现的增长而增长(敏捷)
- 计划式设计: 在项目构建开始之前, 就非常详细地制定各种计划(桥梁施工)
- 最小计划式设计: 20%的计划式, 80%的演进式

不同项目
或者同一个项目的不同阶段,
采取的策略不一样!

架构设计

没有最完美的设计
只有最合适的设计

技术选型

- 四板斧:

需求分析
方案调研
方案对比
方案确定

需求分析

- 需求是什么?
- 约束是什么?
- 一定不能！
- 风险是什么?

需求分析： 风险驱动模型

运用最小的架构技术集合去降低最紧迫的风险，以求事半功倍

- 步骤
  - 识别风险，并排定优先级(从需求出发)
  - 选择并运用一组技术
  - 评估风险降低的程度

需求分析：what & how

1.我不知道要做什么
2.我好像知道要做什么
3.我明确知道不能做什么
4.我明确知道要做什么
5.我明确知道要怎么做

方案调研

你现在已知的: Keywords

1.资料收集
2.明确方向
3.确认细节

方案调研：资料收集

- 搜索: Google & Github
- 书/Github star/日常阅读/笔记库
- 现有已知的项目是否有类似的方案?

What you should do daily? 没吃过猪肉还没见过猪跑?

1. 看各种架构方案
2. 看各种`in action`实践, 例如Redis in Action/RabbitMQ in Action/Kubernetes in Action
3. 看各种网上的文章(碎片化)

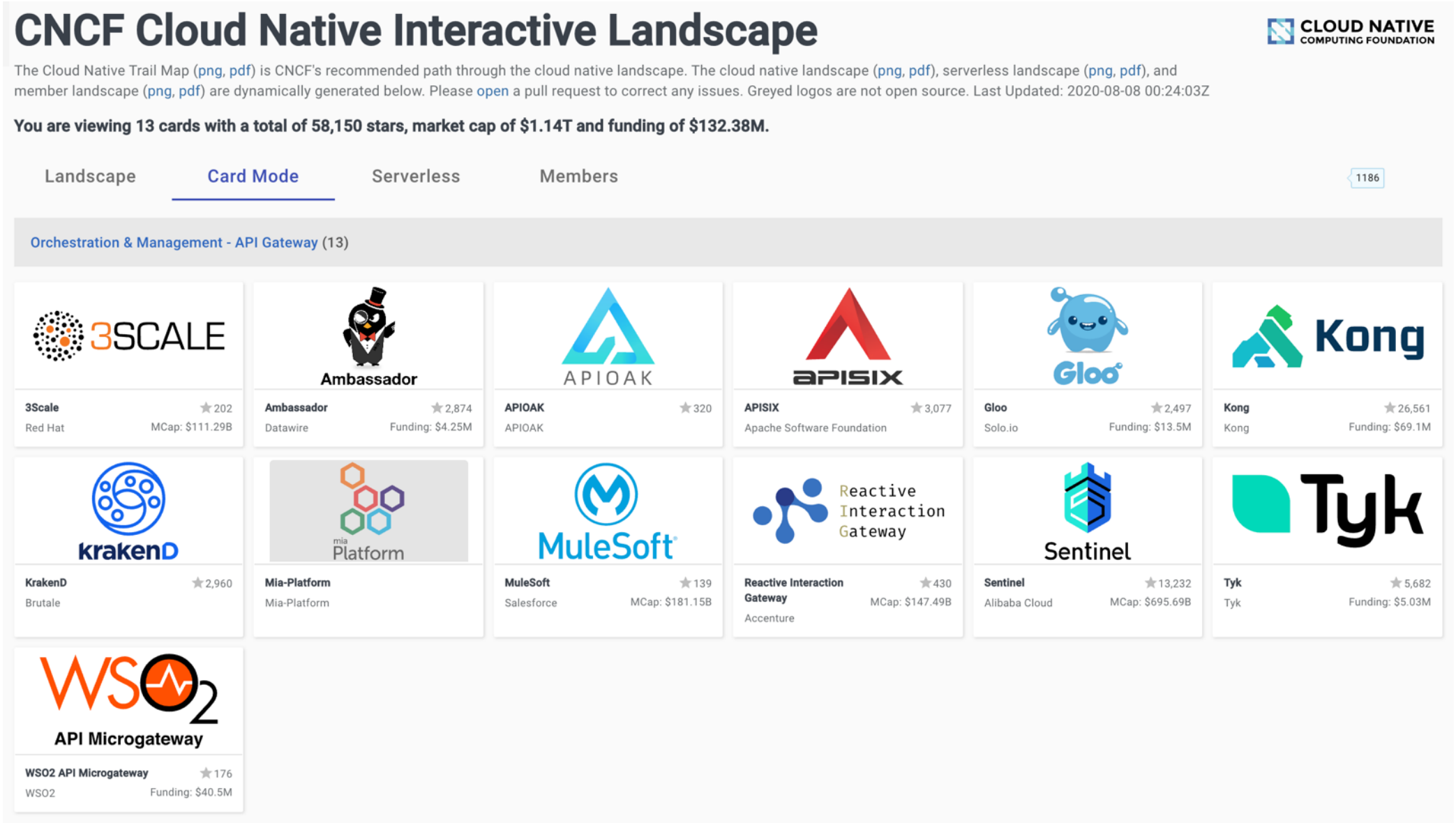
方案调研：方向

- 业界竞品: 主要看文档 - 能力版图/领域知识
- 开源: 看实现
- 类似方案: 看机制

方案调研：APIGateway例子

- 业界: AWS/阿里云/腾讯云/左耳朵耗子MegaEase APIgateway(文档)
- 开源:
  - Openresty: Kong (文档)
  - GO: Tyk / KrakenD / Janus (实现)
  - 国内开发者: **Apache APISIX** https://github.com/apache/apisix
- 类似方案：非网关 traefik / Go版本的翻墙代理 (机制)

方案调研：APIGateway例子



方案调研：调研到什么程度?

你必须完全知道
每一个`关键细节`
是如何实现的!

方案调研：调研到什么程度?

对于权限中心来说, 策略表达式是`核心`
对于APIGateway来说, 如何`代理`并`管理连接`是核心

例子: APIGateway 技术选型的时候, 怎么做代理? 如何同时支持websocket?开源的几个项目, 各自如何实现的?

Janus: httputil.ReverseProxy

- httputil.ReverseProxy was updated in Go 1.12 to support websockets automatically. issues / 对应commit / 源码， httputil 对 websocket 转发处理，参考源码的 handleUpgradeResponse 方法

net/http/httputil

Go 1.15

ReverseProxy now supports not modifying the X-Forwarded-For header when the incoming Request.Header map entry for that field is nil.

When a Switching Protocol (like WebSocket) request handled by ReverseProxy is canceled, the backend connection is now correctly closed.

方案对比

没有最完美的方案
只有最合适的方案

合适 = 现实 * 期望

1. 每个方案的`现实`是什么?
2. 你的`期望`(需求)是什么?

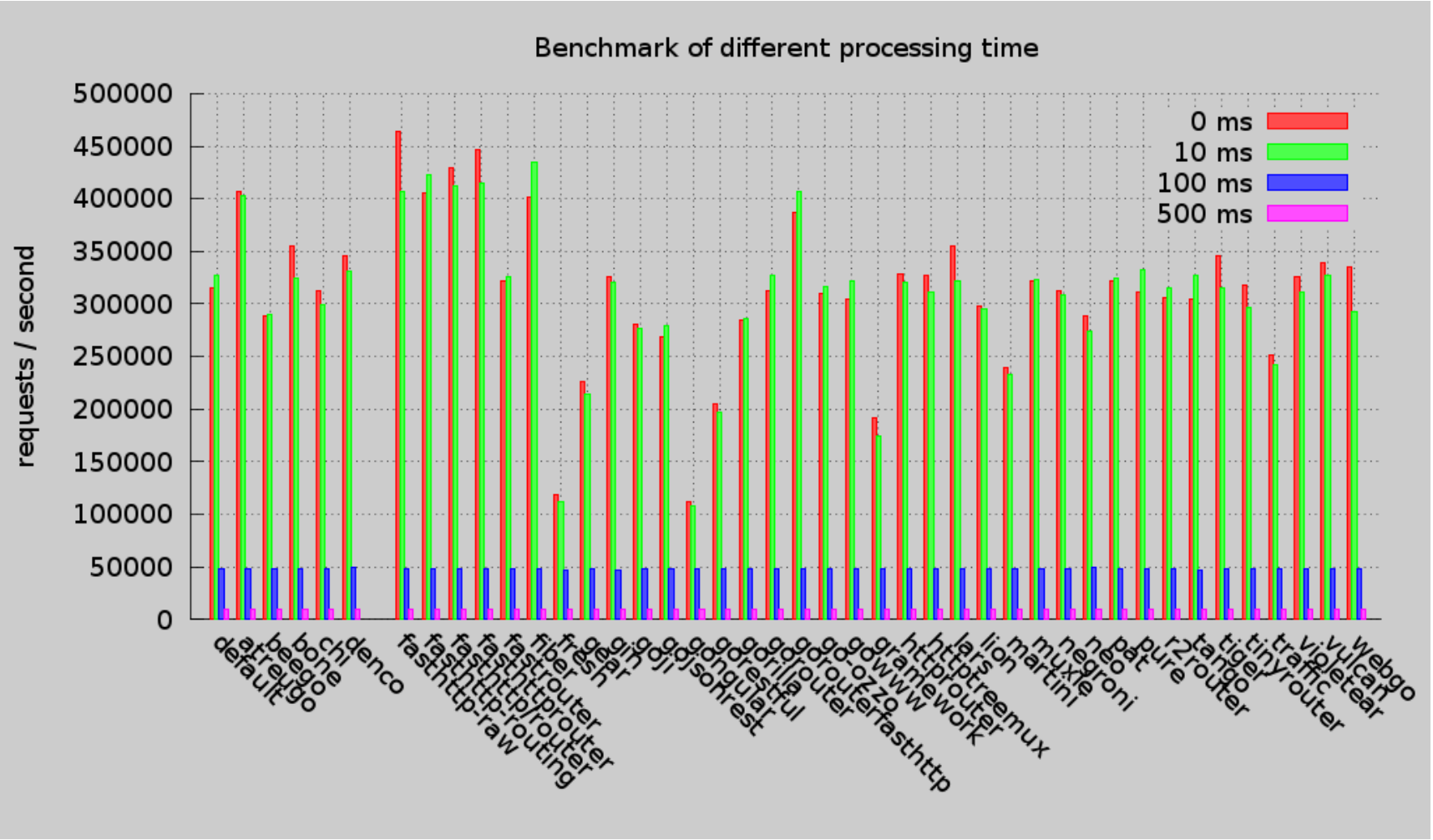
方案对比：合适原则

合适原则: 合适优于业界领先
简单原则: 简单优于复杂
演化原则: 演化优于一步到位

复杂度在哪?

- 务实:
  - 合适但是也不能太落后
  - 简单但是要能满足现在以及未来可见范围内的需求
  - 演化但是底子要好, 必须要赢在起跑线

方案对比：选择开发框架



https://github.com/smallnest/go-web-framework-benchmark

方案对比：Why chi?

Why APIGateway choose chi?

chi is a lightweight, idiomatic and composable router for building Go HTTP services.

Features

- **Lightweight** - cloc'd in ~1000 LOC for the chi router
- **Fast** - yes, see benchmarks
- **100% compatible with net/http** - use any http or middleware pkg in the ecosystem that is also compatible with `net/http`
- **Designed for modular/composable APIs** - middlewares, inline middlewares, route groups and subrouter mounting
- **Context control** - built on new `context` package, providing value chaining, cancelations and timeouts
- **Robust** - in production at Pressly, CloudFlare, Heroku, 99Designs, and many others (see discussion)
- **Doc generation** - `docgen` auto-generates routing documentation from your source to JSON or Markdown
- **No external dependencies** - plain ol' Go stdlib + net/http

方案对比：Why gin?

Why IAM choose gin?

Gin is a web framework written in Go (Golang). It features a martini-like API with performance that is up to 40 times faster thanks to httprouter. If you need performance and good productivity, you will love Gin.

Router

Validation

Middleware & Response

方案对比：稳定依赖原则

稳定依赖原则: 依赖关系必须要指向更稳定的方向. 预期会经常变更的组件都不应该被一个难以修改的组件所依赖, 否则这个多变的组件也将会变得难以被修改.

稳定性指标=出依赖/(出依赖+入依赖)

活跃度如何? 发布频率? 最后一个稳定版之后的issue有哪些关键的尚未解决?

确定方案：技术债务

风险驱动模型 意味着某段时间内资源有限的情况下
只能关注某个核心;
如果做取舍, 就会欠债!

什么时候还! 有没有能力还?

确定方案：可逆还是不可逆

可逆的决策，需要理性看待，主要考虑变更的成本！

不可逆的决策，需要谨慎评估！

开弓没有回头箭，决定了就只能一路趟到黑

确定方案：时刻关注更新

Read the change log!

Feature + Bugfix + 性能

确定方案

难以决策的时候的一种选择

确定方案：推迟决策

- 一个系统中最消耗人力资源的是什么? 系统中存在的耦合–尤其是那些过早做出的, 不成熟的决策所导致的耦合. 那些决策与系统业务需求(用例)无关的, 细节性的决策(框架/数据库/web服务器/工具库/依赖注入等)应该是辅助的, 可以被推迟.
- 一个设计良好的系统架构不应该依赖于这些细节, 而应该尽可能地推迟这些细节性的决策, 并致力于将这种推迟所产生的影响降到最低.

- 未来会遇到的需要提前考虑方案扩展性, 但是不做决策. 例如Redis缓存未来单实例抗不住, 用sharding? Cluster?

确定方案：可扩展性

协议上支持，但是不实现
接口上支持，但是不开放
存储上支持，但是不使用

看得足够远，就不担心需求频繁变更
保留未来的`可能性`

确定方案：每一个选择都需谨慎

无论大小/无关难易

- 为什么我们舍弃了**GORM**而转向sqlx(最后一个版本2018)?
- 为什么我们后端用Go, 前端用Django(而不是GO?)
- 为什么CMDB是微服务框架而我们不是?
- 为什么APIGateway的go-redis ratelimit没有升级到最新?
- ……

小结

技术选型是需要在日常项目中不断`磨练`的一种意识

Q & A

END