Go 工程配置最佳实践

毛剑

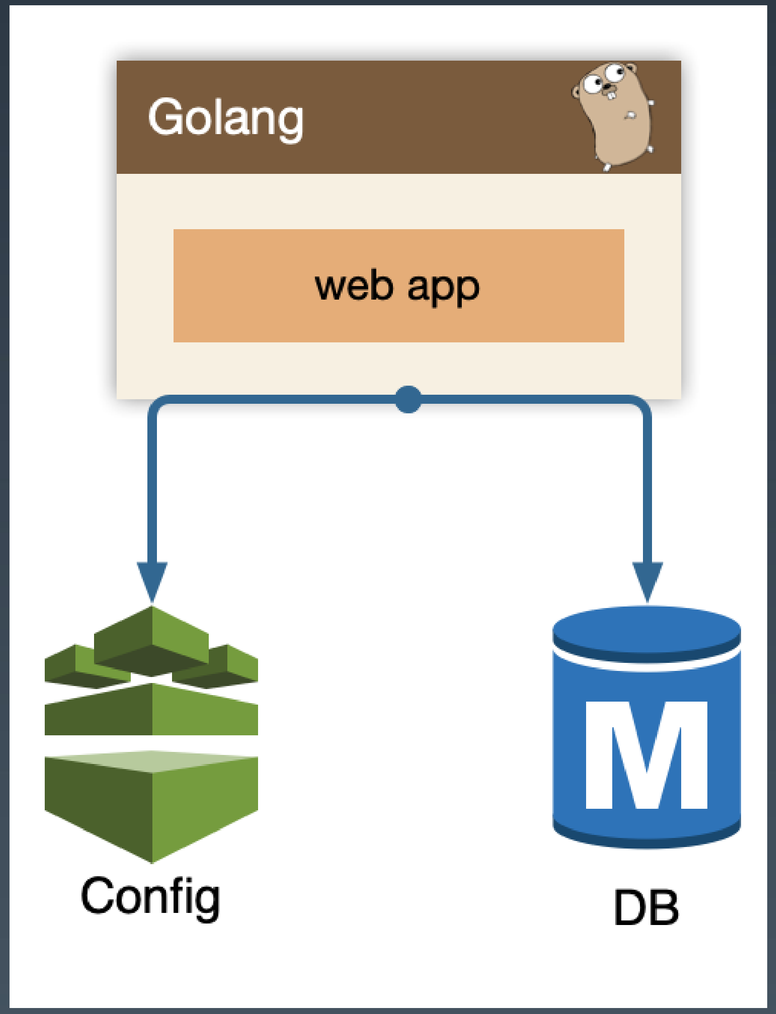
应用构成

当我们需要快速更改系统的行为，但是不更改过程，且不需要昂贵、冗长的重建和重新部署过程时，代码更改就不够用。相反，配置可提供一种低开销的方式更改系统功能。

一个系统具有三个部分组成：

- *Application*
- *Dataset*
- *Configuration*

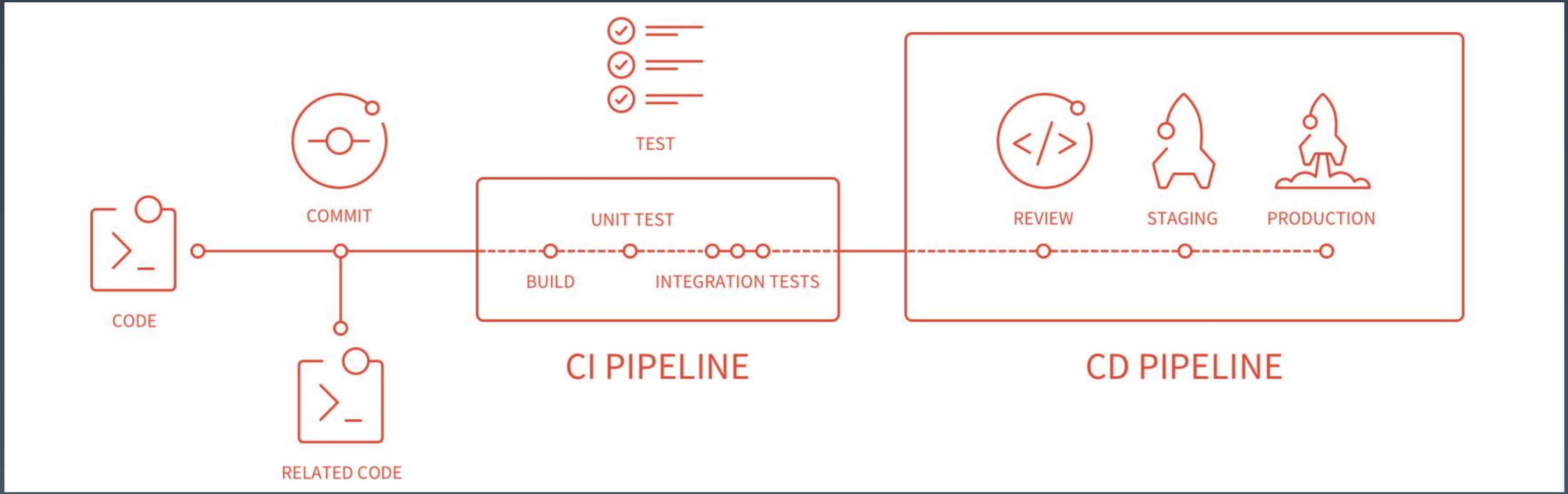
良好的配置界面可实现快速、可靠和可测试的配置更改。如果用户没有直接的方法来更新配置，则很容易出错，同时会经历认知负荷和学习曲线。



配置管理

配置和代码最大的区别在于，代码更改会经过一系列的 CI/CD 流程，每次增量的更改，都要经过代码检查和测试。而更改单个配置选项可能未经测试（甚至无法测试）就交付到生产环境中，导致类似 Facebook BGP 这类超大型连锁事故。

在事故期间，需要承受巨大压力情况下，快速安全地进行配置变更。



配置哲学 - 配置试图

配置哲学，是远离大量的可调参数，转向简单的重点理念。关键任务系统提供大量系统配置，需要对运维人员进行精心的培训和训练。趋于简单化配置的做法，减少了错误的表面积和操作人员的认知负担。

- 以基础设施为中心的视图

  提供尽可能多的配置选项很有用。这样用户可以根据自己的实际需求调整系统的性能表现。配置选项越多，系统可自定义性就越强，也就越能适配业务场景做到最完美。

- 以用户为中心的视图

  以业务服务为主，业务目标是面向 C 端用户人群，那么配置越少越好，专注在业务功能实现上。

但最终都会趋同像以用户中心的视图移动。

Figure 14-1. Control panel in the NASA spacecraft control center, illustrating possibly very complex configuration

配置哲学 - 必填和可选

给定的配置设置可能包含两种类型的问题：必填和可选配置，比如 Redis Client 的监听网络协议和地址。而设置读写超时时间可能是一个可选配置。

为了保持以用户为中心并易于使用的配置，我们要尽可能的减少必填配置，通常做法是把必填变为选填。比如 Redis Server 启动的时候监听本机 IP，以及 6379 端口，比如 Nginx 可以设置多少个 worker 进行工作。那么减少必填配置未必都是静态硬编码的值，也可以动态根据上下文计算，例如 Nginx 读取物理机的 CPU 工作线程设定自己的 workers。

在默认值策略中，必须是安全保守的配置，因为很容易错误传播到各个系统，比如 Jedis 某个版本的默认重试次数是: 86400。

真正需要可选配置的时，演进思路为：hard code -> 可选配置 -> 模板/继承配置或配置套餐。

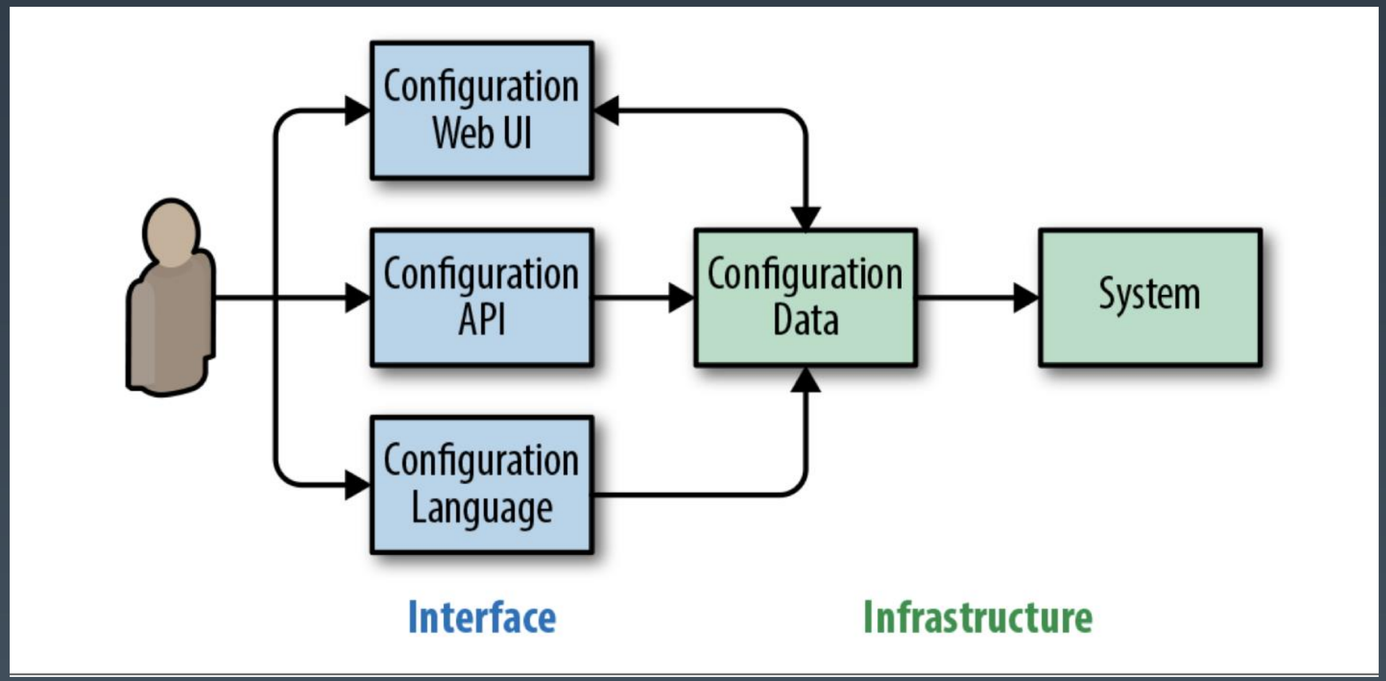
```
// Dial connects to the Redis server at the given network and
// address using the specified options.
func Dial(network, address string, options ...DialOption) (Conn, error)

// NewConn new a redis conn.
func NewConn(c *Config) (cn Conn, err error)
```

配置机制

用什么来存储配置，是不可避免的问题。是 INI，YAML，XML，JSON 还是更加灵活的高级语言来存储配置，比如 Lua，Python 等。按照我们的经验，即便有一些更加灵活的配置高级语言或者 DSL，我们仍然建议将控制和数据分离的思想：系统的基础架构应在静态数据上运行，可以是 JSON，YAML 等，也可以通过更高级别的界面来进行交互，比如使用 API 可以 overlay + automation。

- *静态数据交互，动态控制面。*
- *静态数据基于 DSL，高级语言或者界面生成。*



配置机制

一旦获得静态配置数据，使用这些最终数据格式配置的时候，提取配置的元数据很有用，我推荐使用 protobuf 的核心原因在于，很多配置系统都没有考虑配置本身的 schema 元数据，基于元数据我们可以实现更多高级的控制能力：

- *语义验证，配置是否语法有效，是否符合 validation，比如以外的内存配置导致多一千倍的内存（单位错误），最大化验证配置，在交付应用前中断并降低运维成本。*
- *友好的语法高亮，方便查看。*
- *Linter，标准化要求。*
- *自动化语法格式，类似 gofmt。减少配置 owner 对切换项目时的认知负担，标准格式可以简化自动编辑的过程。*

```
syntax = "proto3";

import "google/protobuf/duration.proto";

package config.redis.v1;

// redis config.
message redis {
  string network = 1;
  string address = 2;
  int32 database = 3;
  string password = 4;
  google.protobuf.Duration read_timeout = 5;
}
```

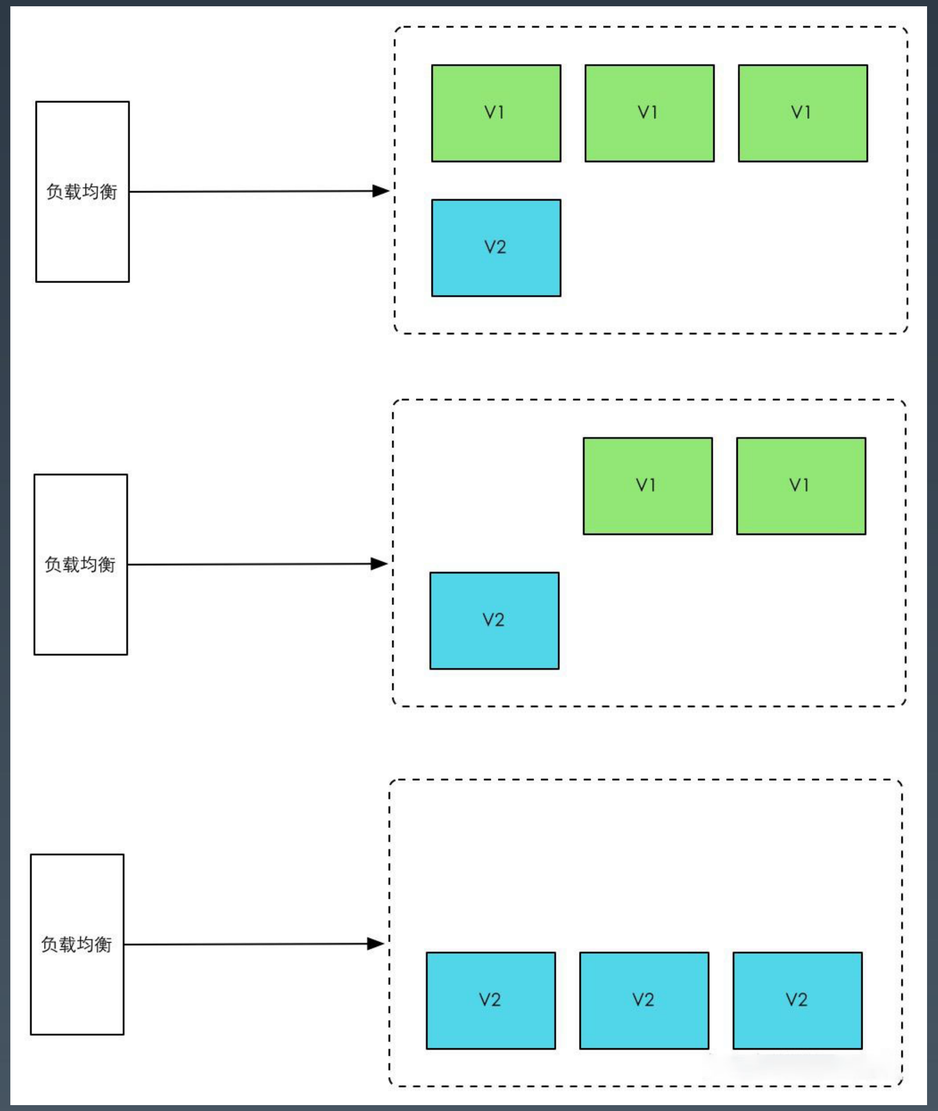
配置机制

- 安全配置更改应用，配置通常不经过单元测试，甚至不是易于测试的，我们希望避免发生可靠性事故。
- 逐步部署的能力，避免了全有或全无的变化，参考 Kubernetes rolling update 策略；
- 如果发现危险，则可以快速回滚更改；

回滚的能力对于减少事件持续事件很重要，与尝试使用临时修复程序对其进行修复相比，回滚有问题的配置可以更快地缓解中断，也避免临时的修复导致二次事故。回滚配置必须是密封的，否则可能很难 reset 到上一次的状态；

- 如果更改导致 operator 失去控制，则自动控制（至少自动停止进度的能力）；

例如修改防火墙，当本机 agent 发现丢失了系统管理员权限，可以自动恢复到上个版本，并且在修改是否需要多次确认危险性操作；



配置分类

- 环境信息（上下文）

  Region、Zone、Cluster、Environment、Color、Discovery、AppID、Host 等之类的环境信息，都是通过在线运行时平台打入到容器或者物理机，供 kit 库读取使用。

- 静态配置

  资源需要初始化的配置信息，比如 HTTP/gRPC Server、Redis、MySQL 等，这类资源在线变更配置的风险非常大，我通常不鼓励 on-the-fly 变更，很可能会导致业务出现不可预期的事故，变更静态配置和发布 Bianry App 没有区别，应该走一次迭代发布的流程。

- 动态配置

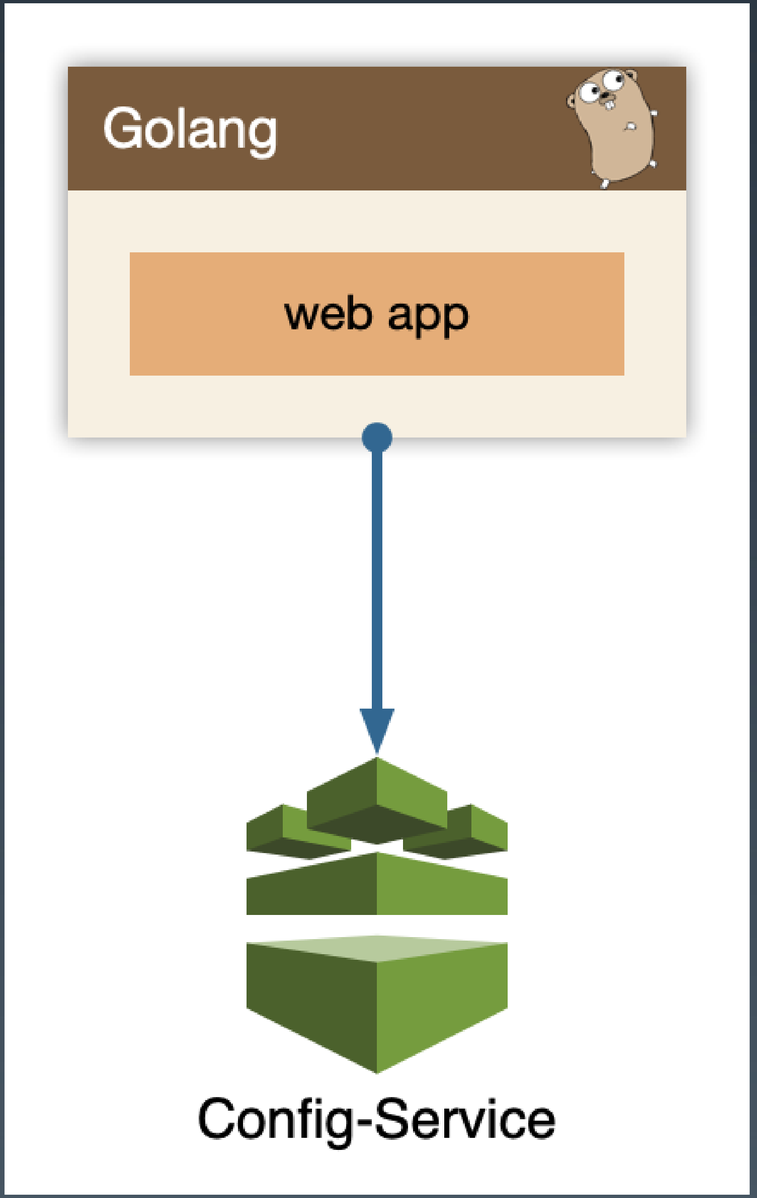
  应用程序可能需要一些在线的开关，来控制业务的一些简单策略，会频繁地调整和使用，我们把这类是基础类型（int, bool）的配置，用于可以动态变更业务流的收归一起，同时可以考虑结合类似 https://pkg.go.dev/expvar 来结合使用。

配置场景 - 业务应用

通用业务应用配置可以直接依赖配置中心，但是比较容易出现的问题是应用 binary 本身和配置存在联动，因此维护这种 mapping 关系变得非常重要，回滚必须是密封的，联动的。

即使我们直接依赖配置中心的动态加载能力，我们仍然建议按照三种分类来构建你的应用程序的配置装载工作。

- *应用读取环境上下文信息，通常例如机房信息等，这类配置是环境变量读取，维护在配置中会导致多机房配置维护的复杂度。*
- *应用依赖静态配置初始化资源和监听端口等，无法热更新加载配置，需要依赖应用发布系统重新发布。*
- *业务的管控面更多是针对业务功能开关的，在配置中心管理中，可以通过不通的 prefix key 区分方便动态推送和加载。*



配置场景 - 自建基础设施

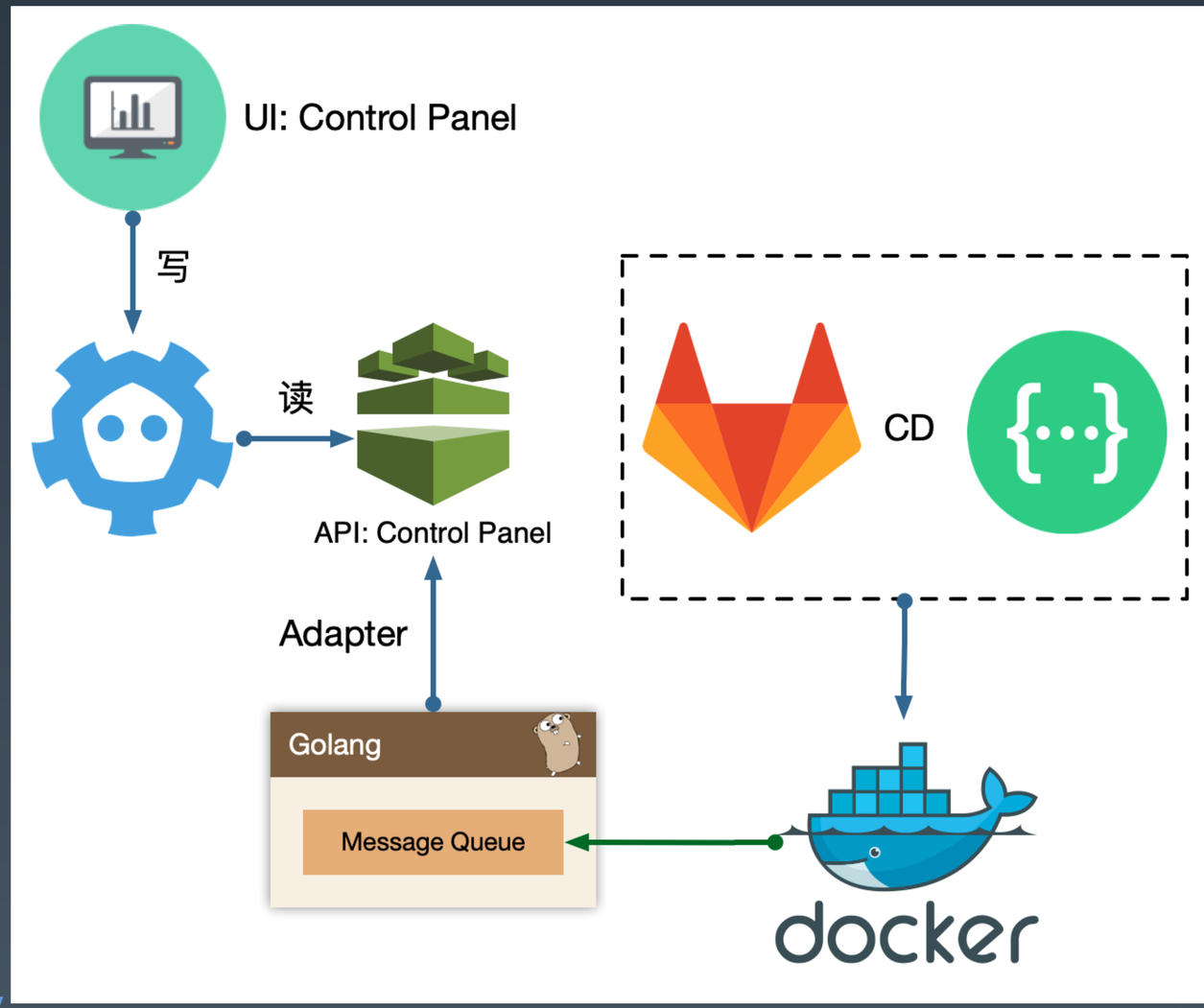
自建的基础设施，要考虑两个点：更好地接入企业内现有的基础设施，未来开源也可以方便外部开发者快速适配。API Control 作为适配层，入口&出口适配：

- *入口适配：提供 Poll + Watch Endpoint 定义，基础设施数据面直接依赖。*
- *出口适配：定义配置装载源码接口（即：language interface），用户自定义实现。*

这样开源生态修改的是出口适配。

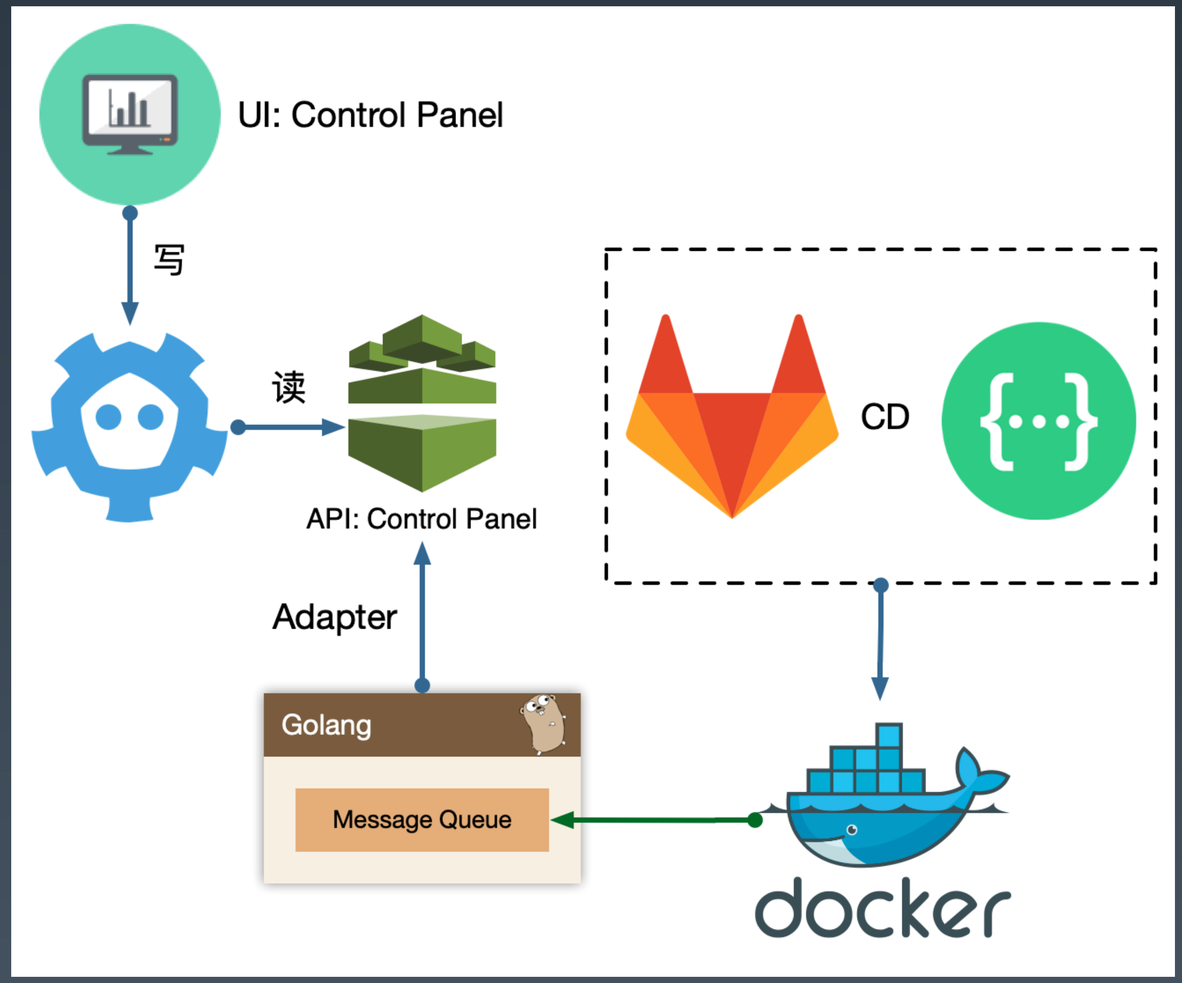
API GW -> API CTRL -> Config-Service <- API UI

- *API Control 底层存储用配置中心，方便查看版本和权限追踪。*
- *UI Control 直接操作配置中心，不直接用配置中心的 UI，这样读写分离，内部系统不太会切换存储。*



配置场景 - 外部基础设施

我们有时候也直接依赖外部基础设施作为企业的核心设施，例如南北流量入口 Nginx，外部生态的基础设施有时候会直接依赖本地配置文件来装载配置，这个时候需要扩展其具备动态能力也至关重要。



References

- 《站点可靠性工作手册》，第14-15章
- AWS CloudFormation
- Envoy & Kubernetes 配置管理
- 其他国内公司的配置管理

Q&A