如何架构设计健壮的在线系统

【 PHP&Golang 语言环境版 】

黑夜路人 Black

2022/7

讲师介绍

黑夜路人Black

CSDN博客技术专家
好未来智能内容技术总监
好未来集团服务端研究员
好未来技术委员会服务端通道主席

目录

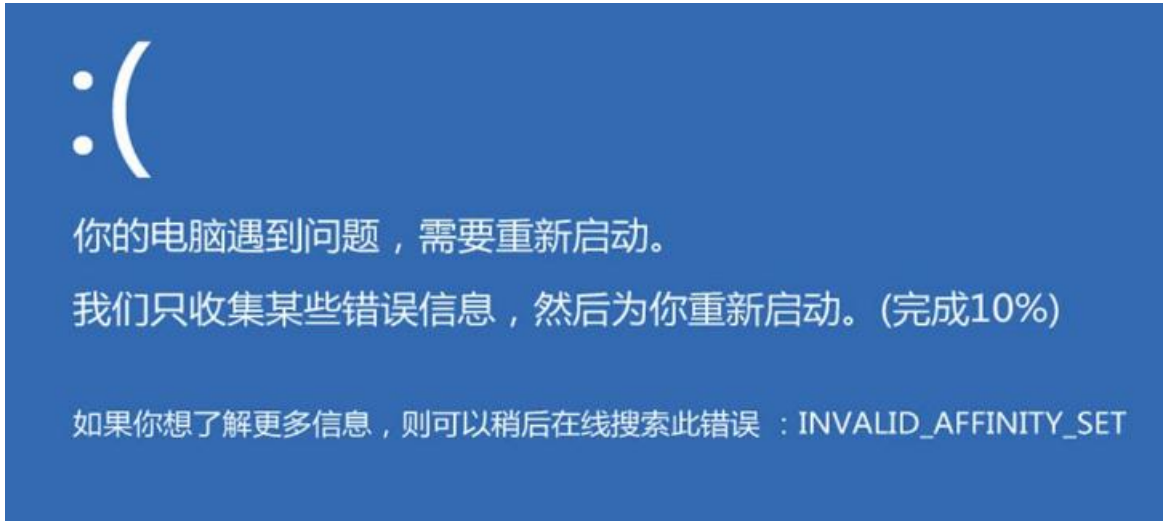
一、为什么要设计健壮系统

为什么要设计健壮系统？

我们系统里经常会遇到哪些问题？

1. 为啥测试好好的，上到线上代码一堆bug，一上线就崩溃或者一堆问题？
2. 为什么感觉自己系统做的性能很好，上线后流量一上来就雪崩了？
3. 为啥自己的系统上线以后出问题不知道问题在哪儿，完全无法跟踪？

...



为什么要设计健壮系统？

构建健壮系统包含哪些关键因素？

核心要素

1. 良好的软件系统架构设计
2. 编程最佳实践和通用原则
3. 个人专业软素质

架构核心：

- 可维护性
- 可扩展性
- 健壮性（容灾）

要素拆解

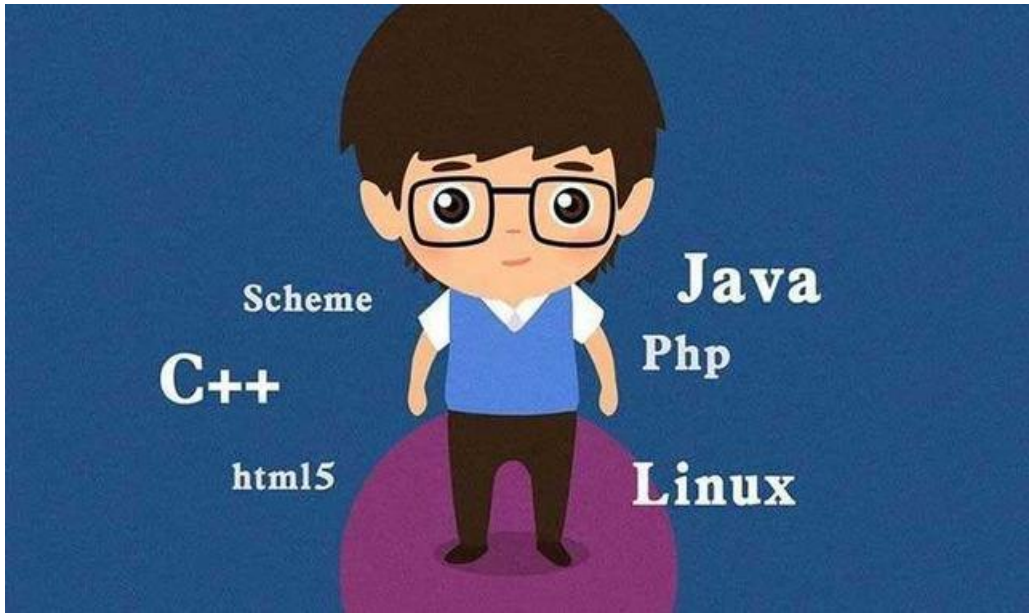
- **系统架构：** 网络拓扑是什么、用什么存储、用什么缓存、整个数据流向是如何、那个核心服务采用那个开源软件支撑、用什么编程语言来构建整个软件连接各个服务、整个系统如何分层？
- **系统设计：** 采用什么编程语言、编程语言使用什么编程框架或中间件、使用什么设计模式来构建代码、中间代码如何分层？
- **编程实现：** 编程语言用什么框架、有什么规范、编程语言需要注意哪些细节、有哪些技巧、那些编程原则？

说明：主要面向PHP为主开发语言的业务，Golang/Java等可以作为参考

二、系统架构关键原则

用架构师的视角来思考问题

- 程序员视角更多考虑的是我如何快速完成这个项目，架构师视角是不仅是我完成整个项目，更多需要思考整个架构。
- 架构是否清晰、是否可维护、是否可扩展、可靠性稳定性如何，整个技术框架和各种体系选型是否方便容易开发维护；整个思考维度和视角是完全不一样的。
- 程序员视角是执行层面具体编码的视角，架构师视角是设计师的视角，会更宏观，想的更远。
- 比如编程语言选型 PHP vs Golang、Java vs Golang、C++ vs Rust等等

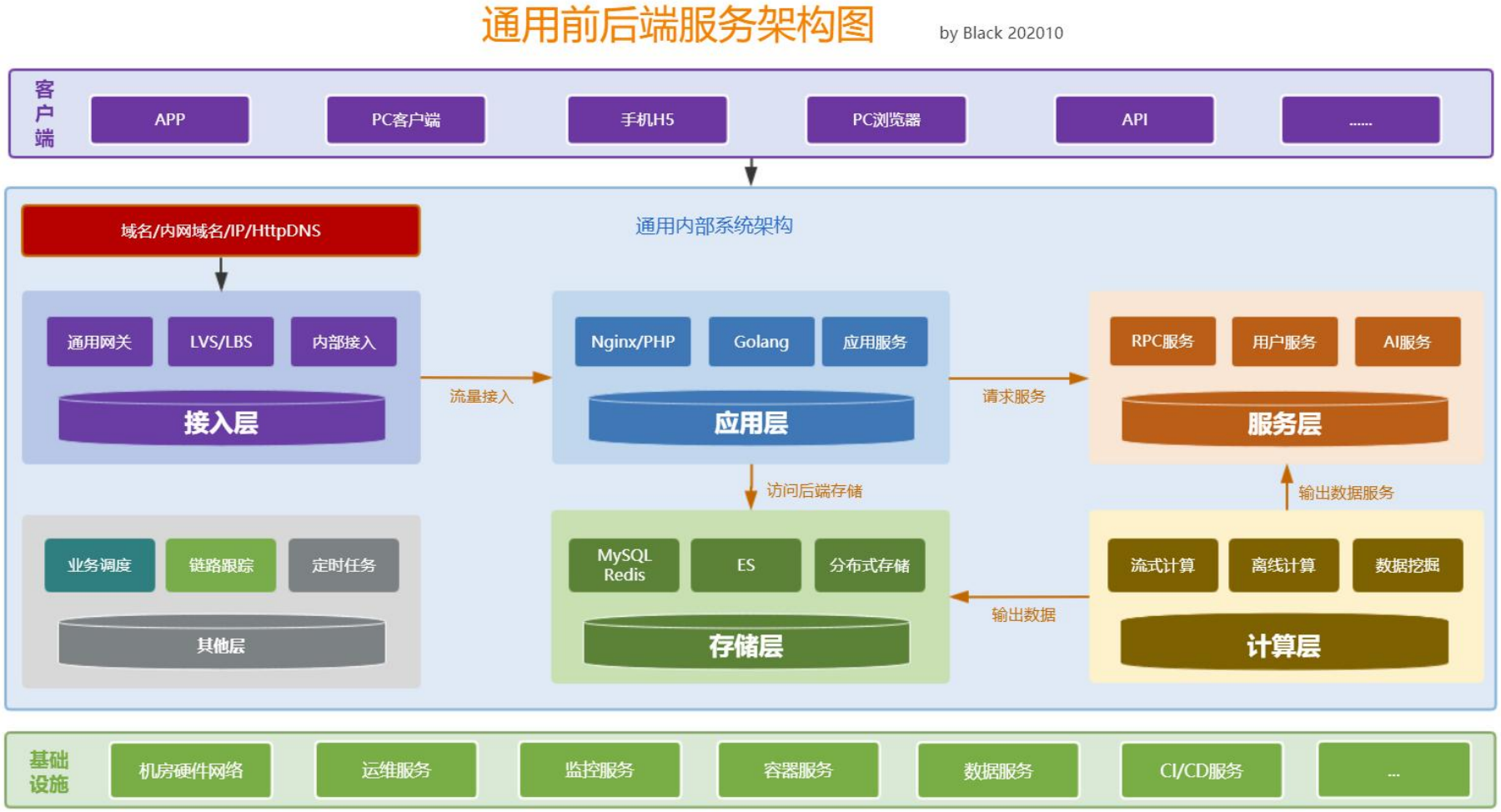


系统架构遵循哪些原则？

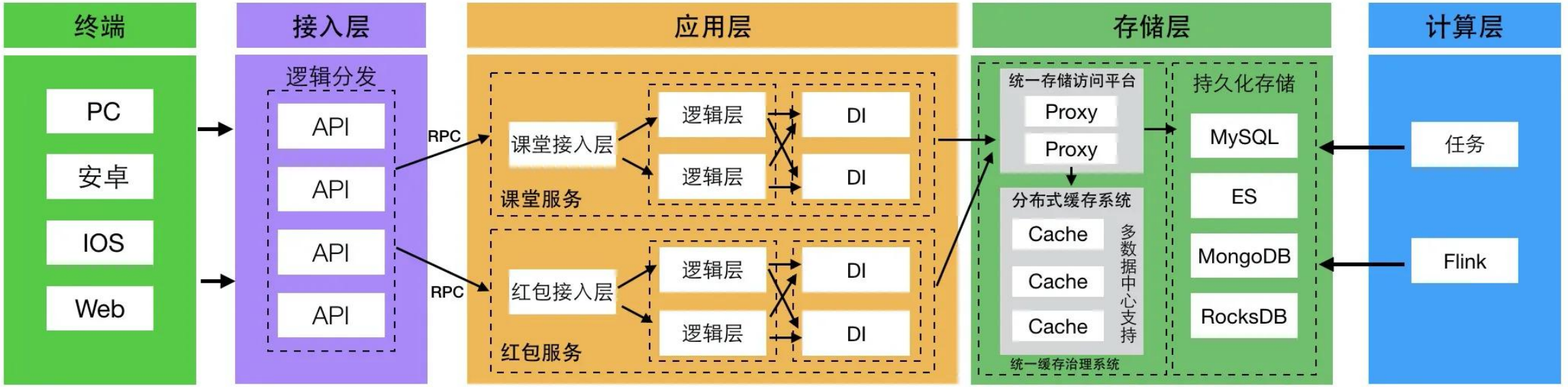
服务架构链路要清晰

整个服务架构包含：

- 接入层（网关、负载均衡）
- 应用层（PHP程序等）
- 服务层（微服务接口等）
- 存储层（DB、缓存、检索ES等）
- 计算层（不一定包含，一般会以服务层或存储层出现）



服务架构链路要清晰



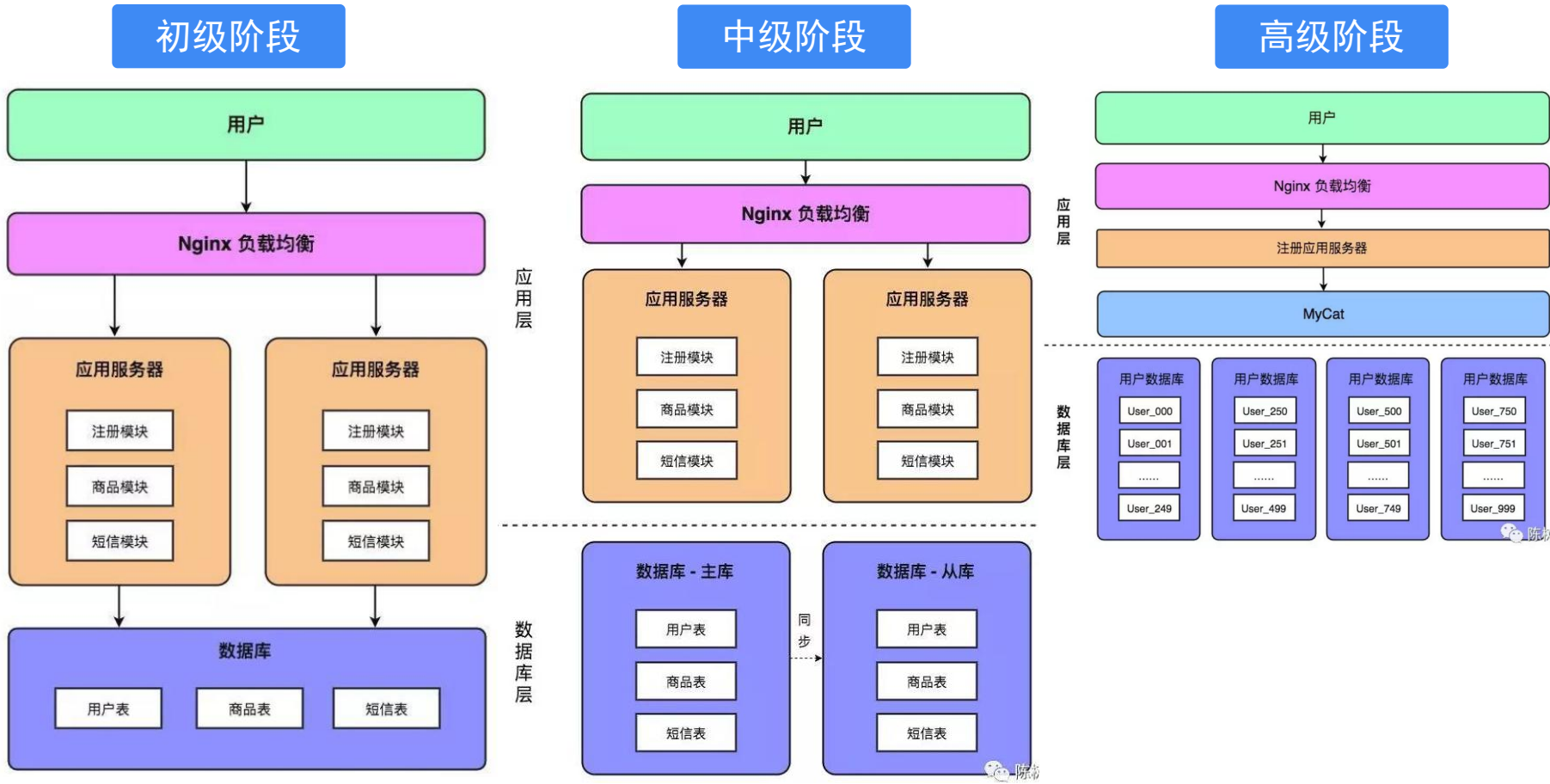
服务架构分层任务：

- 接入层：需要完成逻辑分发
- 应用层：需要解决业务逻辑运算和本地数据共享
- 存储层：存储层，需要考虑缓存和持久化存储问题
- 计算层：需要考虑一些实时或者离线计算场景和定时任务等

系统架构遵循哪些原则？

每个服务都需要考虑去单点

- DB考虑master/slave架构，保持数据不丢，访问不断，数据量大以后是否做分布式存储等等，redis等同样；
- 后端微服务层同样必须多台服务（通过ServiceMesh等调度，或者是etcd/zookeeper等服务发现等方式）；



延伸阅读：https://mp.weixin.qq.com/s/D9IDF5iy73LI7rogLdEr4w

系统架构遵循哪些原则？

每个服务都必须考虑最优的技术选型（最佳实践）

- PHP框架选择，语言选择都是稳定成熟可靠（高性能API选Swoole、Golang等，业务系统考虑Laravel、Symfony、Yii等主流框架）；
- 微服务框架，远程访问接口协议（TCP /UDP /QUIC /HTTP2 /HTTP3等），信息内容格式可靠（json /protobuf /yaml /toml等）
- 选择的扩展稳定可靠（PHP各个可靠扩展，具备久经考验，超时、日志记录等基础特性）；

延伸阅读

存储中间件

- ShardingSphere
- Cetus
- Altas
- MyCat
- Twemproxy
- Codis

接入层

- OpenResty
- Tengine
- HAproxy
- APISIX
- Kong
- Orange

PHP框架

- Swoole
- Hyperf
- Fend
- Symfony
- Laravel
- Yii
- Yaf

PHP扩展

- 消息打包msgpack
- gRPC协议
- Protobuf协议
- RabbitMQ客户端
- Kafka客户端

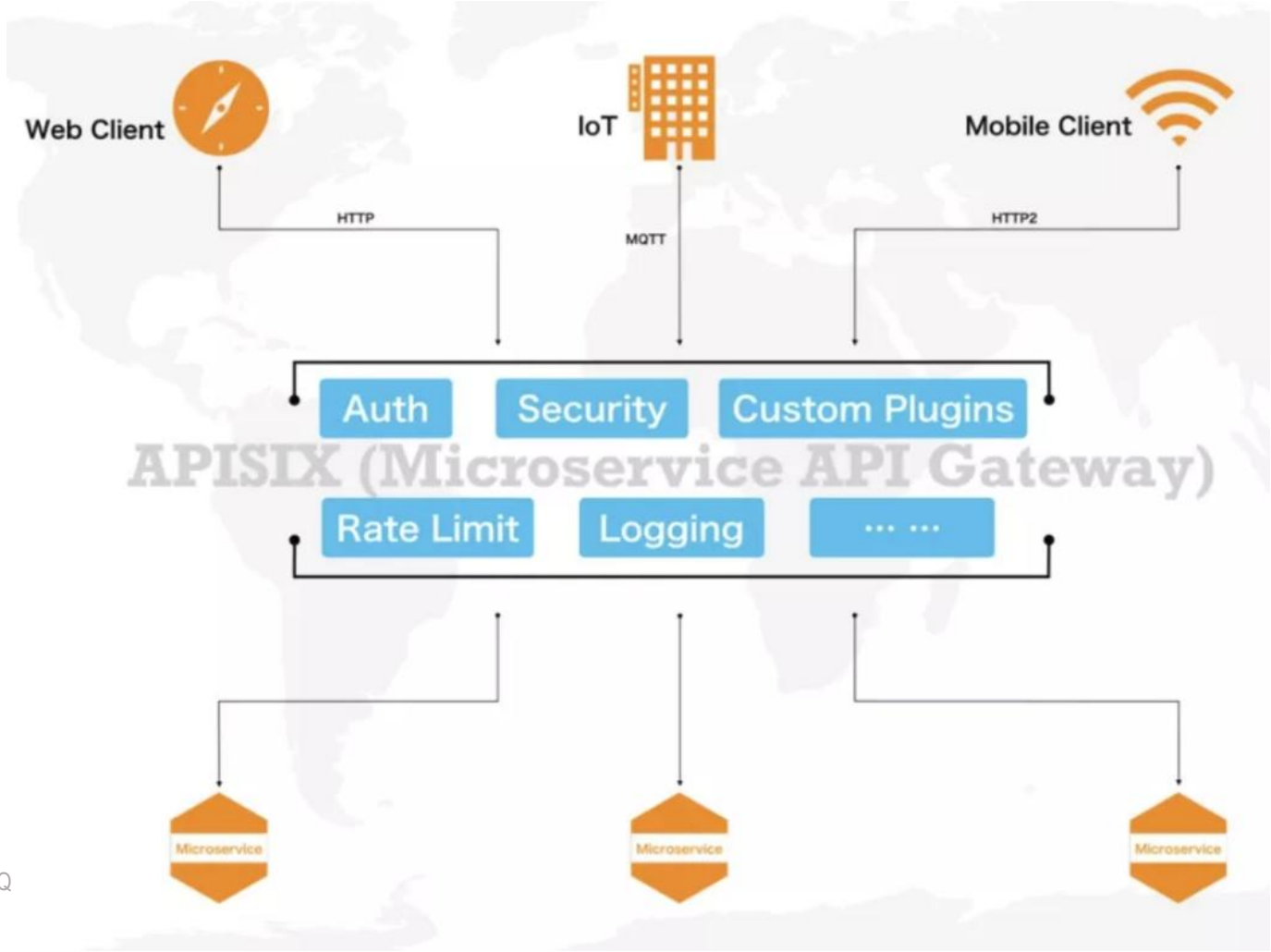
PHP扩展

- 日志扩展SeasLog
- 图片处理imagick
- 用户缓存Yac
- 用户缓存Apcu
- 配置文件解析Yaconf
- Yaml格式解析
- Toml格式解析
- 字典树Trie
- 中文简繁转换Opencc
- 加解密库mcrypt
- 任务分配gearman
- IP地址geoip
- 性能跟踪xhprof
- cURL(QUIC/HTTP2)

系统架构遵循哪些原则？

服务必须考虑熔断降级方案

- 在大流量下，如何保证最核心服务的运转（比如在线课堂中是老师讲课直播重要，还是弹幕或者点赞重要），需要把服务分层，切分一级核心服务、二级重要服务、三级可熔断服务等等区分
- 还需要有对应的预案；（防火/防火演练等）；
- 需要对应的系统支持，比如API网关的选择使用。（OpenResty/Kong/APISIX等，单核2W/qps，4核6W/qps）

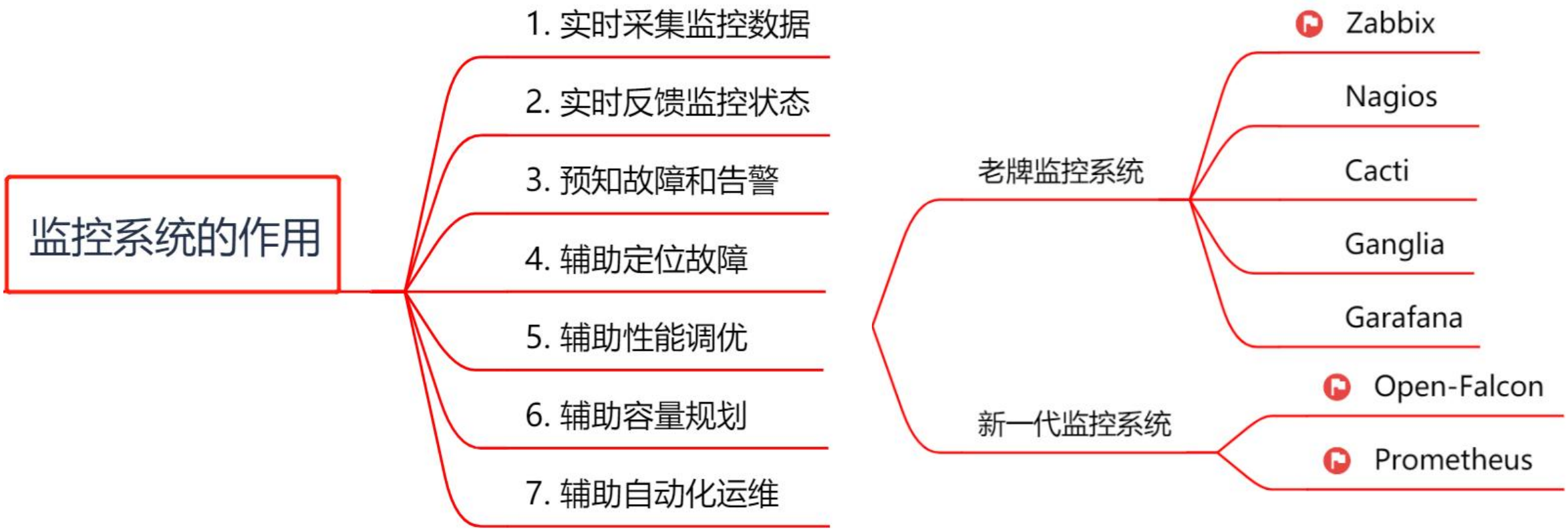


延伸阅读：

https://mp.weixin.qq.com/s/b5SZII2r_MUKJT3qgg-yBw

https://mp.weixin.qq.com/s/FuHsYOTzqBsXIjFJOqQfXQ

运维部署回滚监控等系统需要快速高效



延伸阅读：

https://mp.weixin.qq.com/s/eKc8qoqNCgqrnont2nYNgA

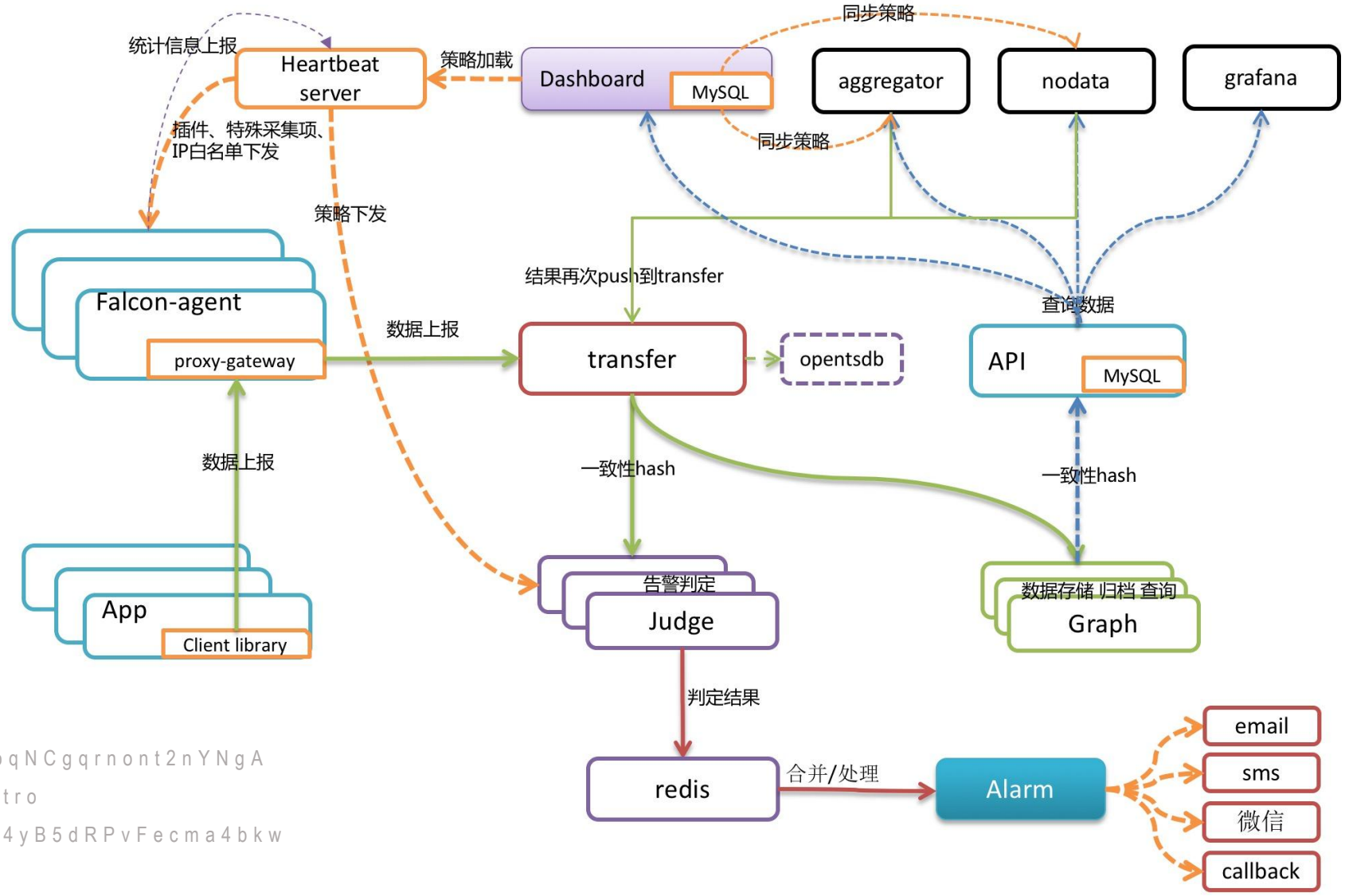
https://book.open-falcon.org/zh/intro

https://mp.weixin.qq.com/s/Wo_em4yB5dRPvFecma4bkw

系统架构遵循哪些原则？

运维部署回滚监控等系统需要快速高效

- 整个代码层次结构，编译上线整个流程，如何是保证高效率可靠的；上线方便，回滚也方便，或者回滚到任何一个版本必须可靠；
- 日志监控、系统报警等等



延伸阅读：

https://mp.weixin.qq.com/s/eKc8qoqNCgqrnont2nYNgA

https://book.open-falcon.org/zh/intro

https://mp.weixin.qq.com/s/Wo_em4yB5dRPvFecma4bkw

编写的接口和前后端联合调试要方便快捷

- 接口可靠性测试必须充分，并且善于使用好的工具（比如 Filder /Postman /SoapUI等）
- 并且对应接口文档清晰，最好是能够通过一些工具生成好API文档（APIJson /Swagger/Eolinker），大家按照对应约定格式协议进行程序开发联调等；

延伸阅读

前端的请求	传统方式	APIJSON

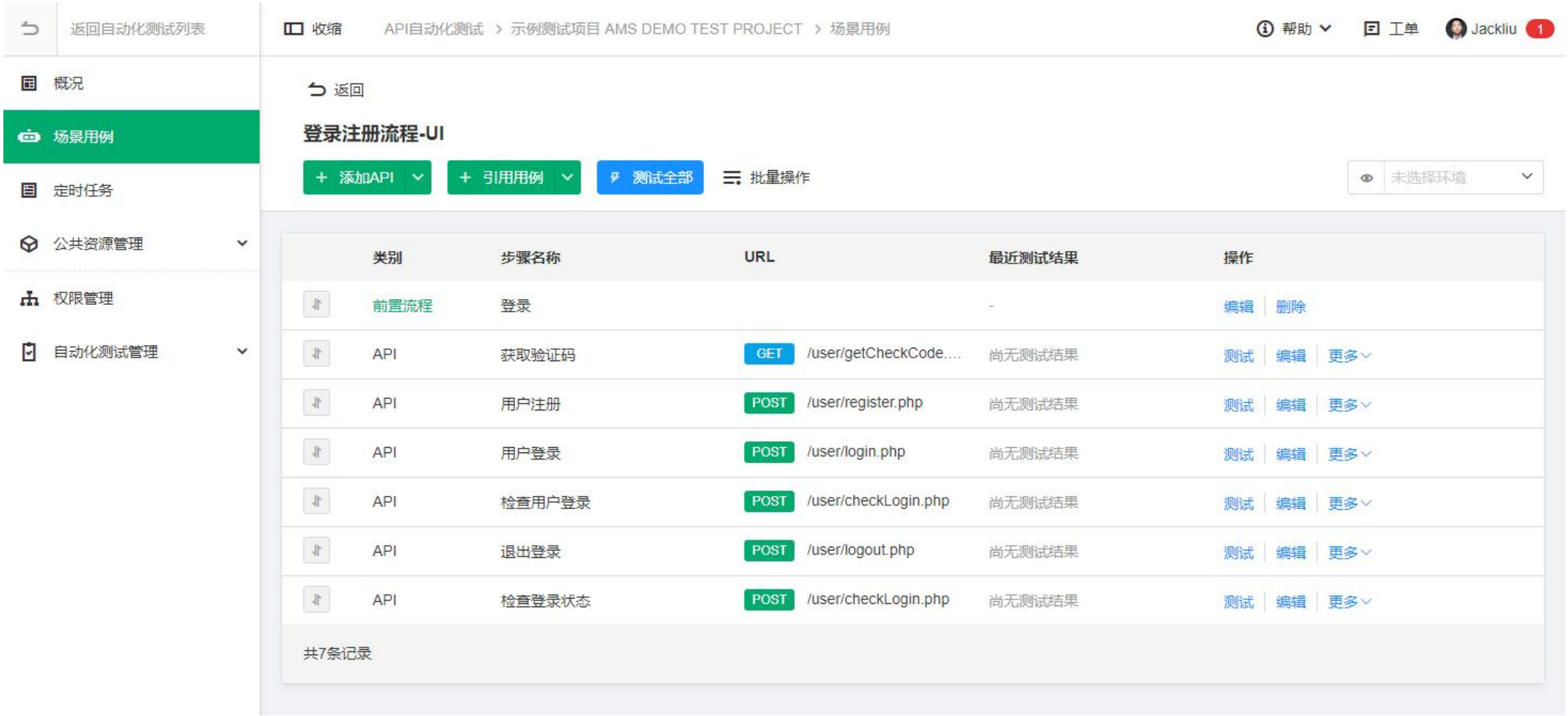
User	base_url/get/user?id=38710	base_url/get/
		{
		"User":{
		"id":38710
		}
		}

Moment和对应的User	分两次请求
	Moment:
	base_url/get/moment?userId=38710
	
	User:
	base_url/get/user?id=38710
		base_url/get/
		{
		"Moment":{
		"userId":38710
		},
		"User":{
		"id":38710
		}
		}

User列表	base_url/get/user/list?
	page=0&count=3&sex=0
		base_url/get/
		{
		"User[]":{
		"page":0,
		"count":3,
		"User":{
		"sex":0
		}
		}
		}

系统架构遵循哪些原则？

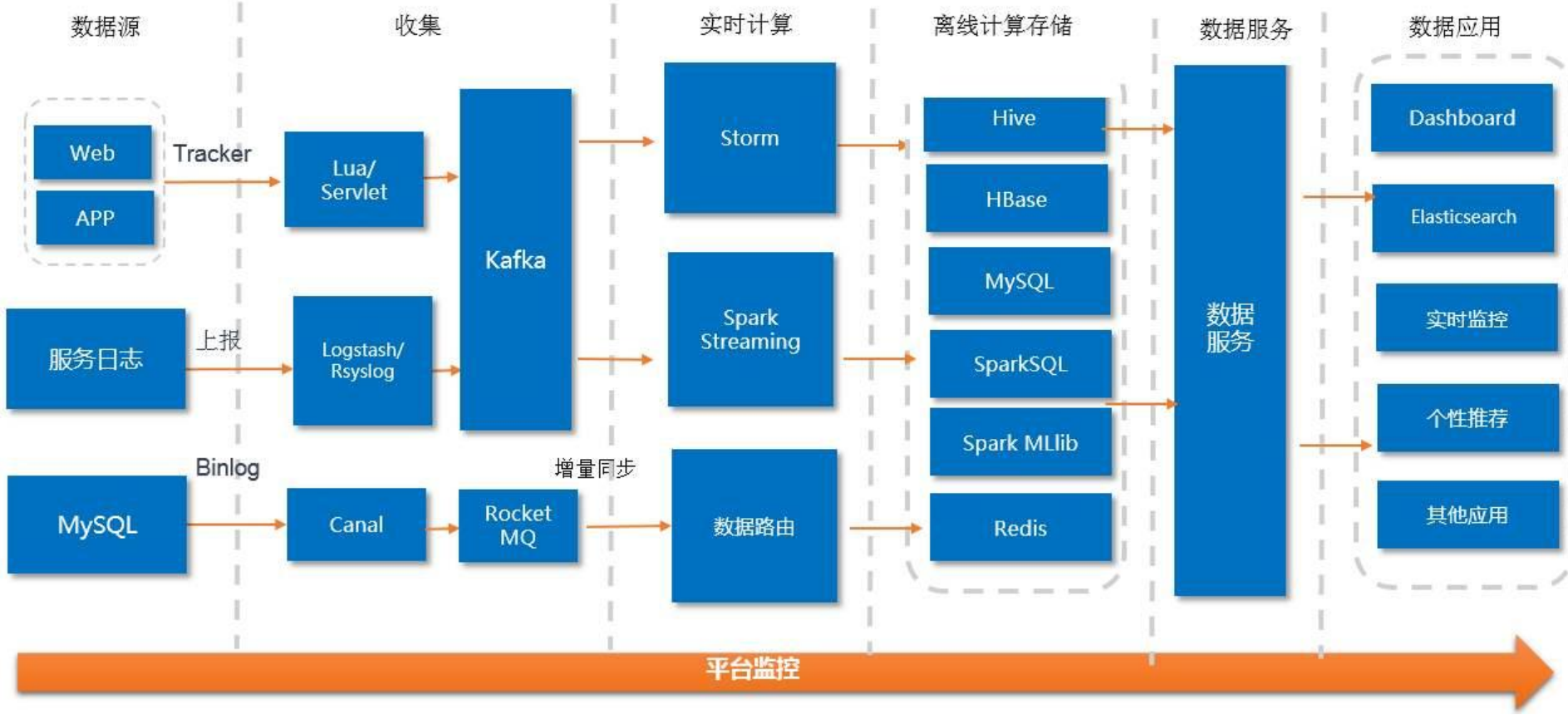
编写的接口和前后端联合调试要方便快捷



系统架构遵循哪些原则？

让整个系统完全可监控可追踪

对应的trace系统（包含trace_id），把从 接入层、应用层、服务层、存储层等都能够串联起来，每个环节出现的问题都可追踪，快速定位问题找到bug或者服务瓶颈短板；也能够了解整个系统运行情况和细节。（比如一些日志采集系统 OpenResty/Filebeat/Flume/LogStash/ELK）



延伸阅读

为什么要设计健壮系统？

让你的应用微服务化、无状态化、容器化

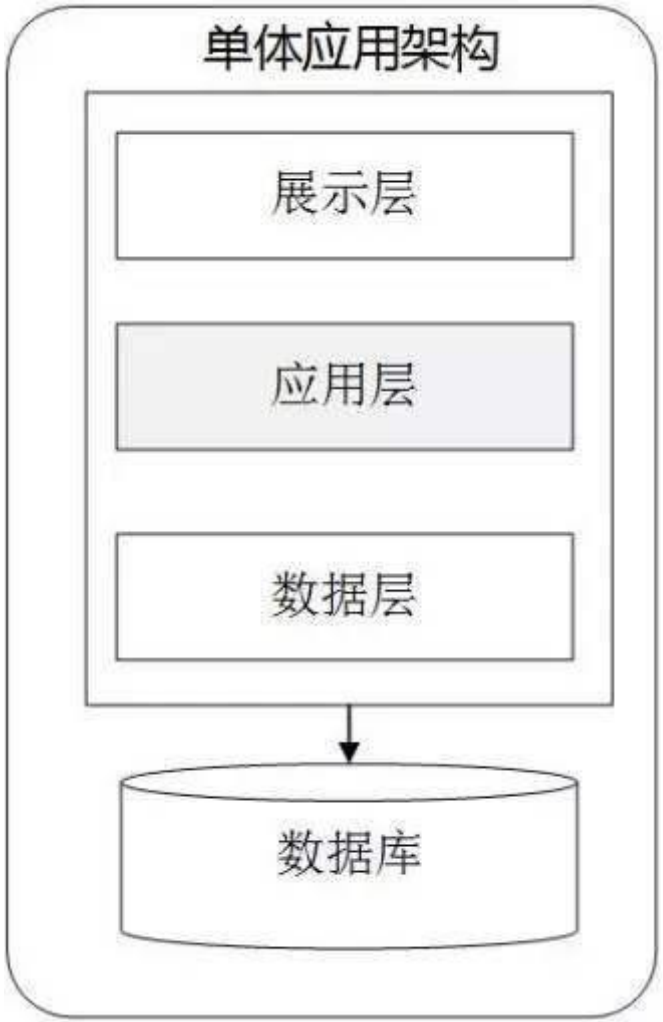
- 让你的应用可以做到：**去中心化、原子化、语言无依赖、独立自治、快速组合、自动部署、快速扩容**，采用微服务+容器化来解决。
- 在面对大并发量请求情况下，在寻求系统资源的状态利用场景，大部分考虑的都是横向扩展，简单说就加机器解决。在docker和k8s的新的容器化时代，横向扩展最好的方法是快速扩充新的应用运行容器；阻碍我们横向扩展的最大的阻碍就是“有状态”，有状态就是有很多应用会存储私有的东西在应用运行的内存、磁盘中，而不是采用通用的分布式缓存、分布式存储解决方案，这会导致我们的应用在容器化的情况下无法快速扩容，所以我们的应用需要“去有状态化”，让我们的应用全部“无状态化”。

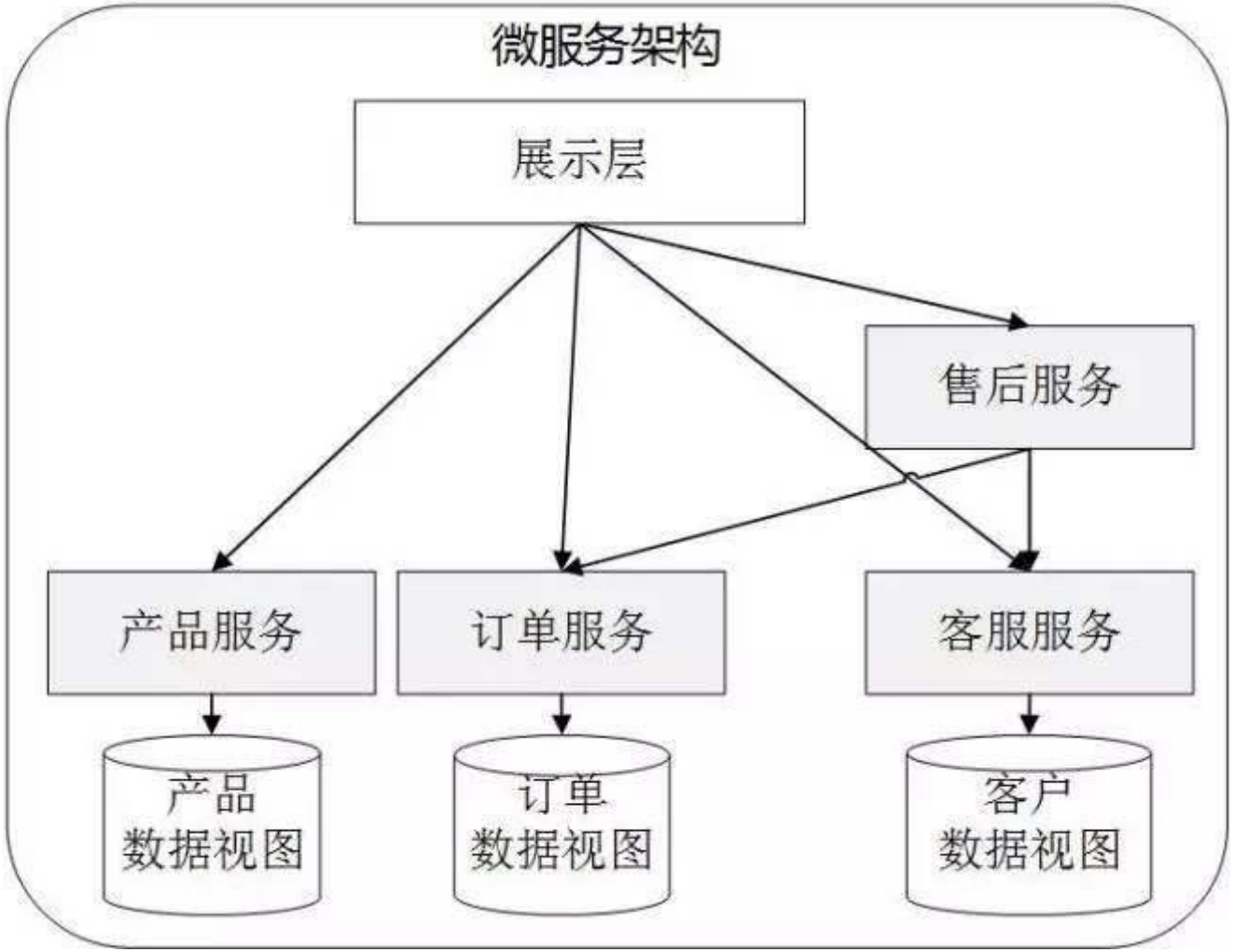
- 微服务化的逻辑是让的每个服务可以独立运行，比如说用户中心系统对外提供的不是代码级别的API，而是基于RESTfu或者gRPC协议的远程一个服务多个接口，这个服务或接口核心用来解决把整个用户中心服务变成独立服务，谁都可以调用，并且这个服务本身不会对内外部有太多的耦合和依赖，对外暴露的也只是一个个独立的远程接口而已。
- 尽量把我们的关键服务可以抽象成为一个个微服务，虽然微服务会增加网络调用的成本，但是每个服务之间的互相依赖性等等都降低了，并且通过容器技术，可以把单给微服务快速的横向扩展部署增加性能，虽然不是银弹，也是一个非常好的解决方案。
- 基于容器的微服务化以后，整个业务系统从开发、测试、发布、线上运维、扩展 等多个方面都比较简单了，可以完全依赖于各种自动半自动化工具完成，整个人工干预参加的成分大幅减少。

为什么要设计健壮系统？

让你的应用微服务化、无状态化、容器化

从单体应用到微服务

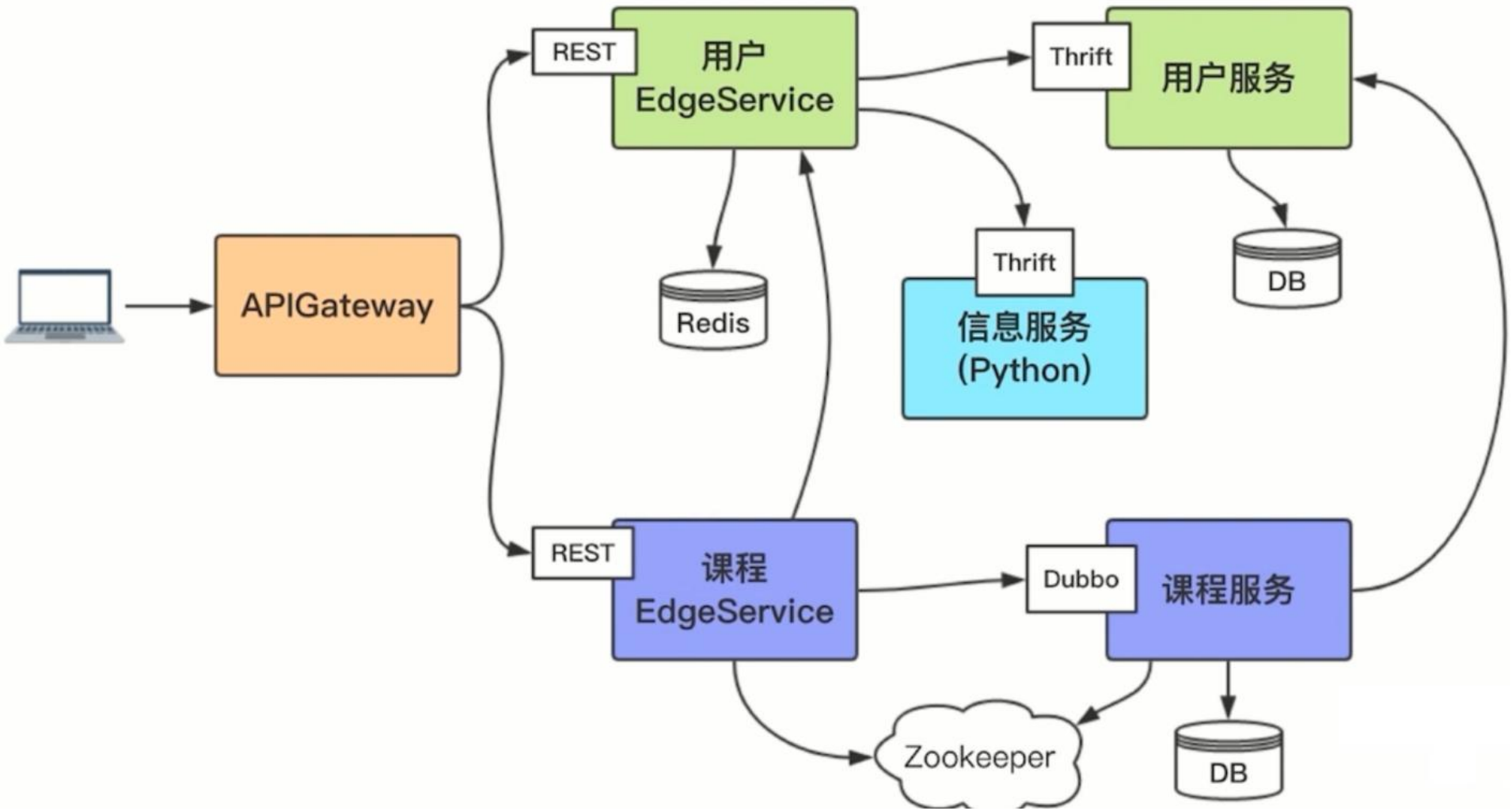




为什么要设计健壮系统？

让你的应用微服务化、无状态化、容器化

应用架构变化



为什么要设计健壮系统？

让你的应用微服务化、无状态化、容器化

容器工作原理

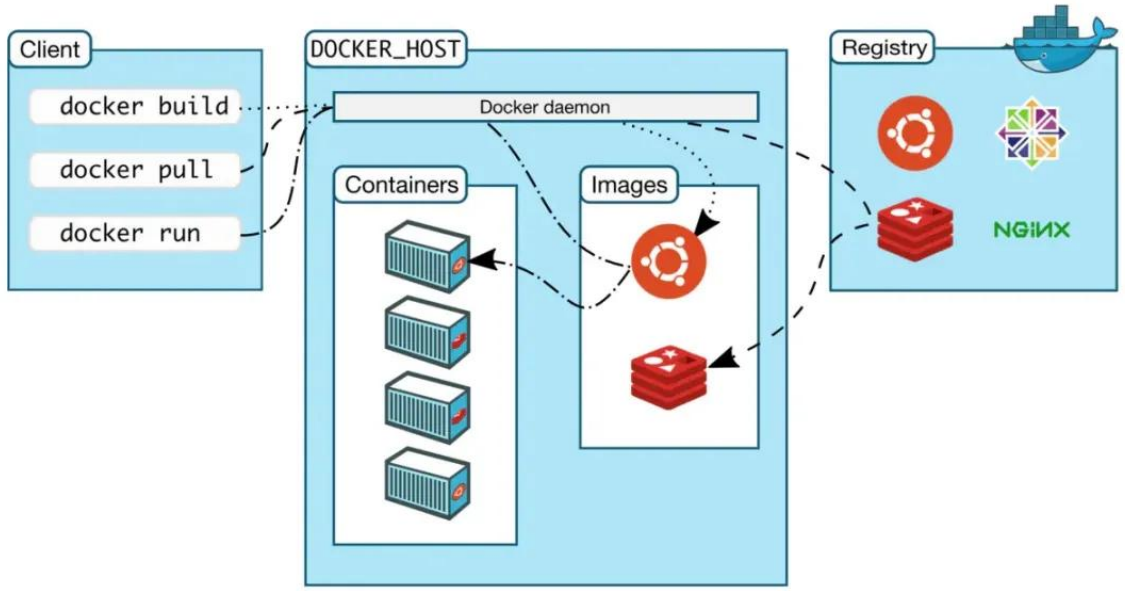
```c
/*
  简单容器底层机制实现模拟演示
  说明：主要利用Linux的Namespace机制来实现，linux系统中unshare命令效果类似
  Docker 调用机制是：  Docker -> libcontainer(like lxc) -> cgroup -> namespace
  Code by Black 2020.10.10
*/
#include <sys/types.h>
#include <sys/wait.h>
#include <linux/sched.h>
#include <sched.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#define STACK_SIZE (1024 * 1024)

static char container_stack[STACK_SIZE];
char* const container_args[] = { "/bin/bash",  NULL };

//容器进行运行的程序主函数
int container_main(void *args)
{
    printf("容器进程开始. \n");
    sethostname("black-container", 16);
    //替换当前进程ps指令读取proc环节
    system("mount -t proc proc /proc");
    execv(container_args[0], container_args);
}

int main(int args, char *argv[])
{
    printf("======Linux 容器功能简单实现 ======\n");
    printf("======code by Black 2020.10 ======\n\n");
    printf("正常主进程开始\n");
    // clone 容器进程: hostname/消息通信/进程id 都独立 (CLONE_NEWUSER未实现)
    int container_pid = clone(container_main, container_stack+STACK_SIZE,
        SIGCHLD | CLONE_NEWUTS | CLONE_NEWIPC | CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET, NULL);
    // 等待容器进程结束
    waitpid(container_pid, NULL, 0);
    //恢复 /proc 下的内容
    system("mount -t proc proc /proc");
    printf("主要进程结束\n");
    return 0;
}
```

```
[root@VM-0-10-centos code]# gcc -o container container.c && ./container
======Linux 容器功能简单实现 ======
======code by Black 2020.10 ======

正常主进程开始
容器进程开始.
[root@black-container code]# hostname
black-container
[root@black-container code]# ip addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
[root@black-container code]# ps ef
  PID TTY      STAT   TIME COMMAND
    1 pts/1    S      0:00 /bin/bash XDG_SESSION_ID=364 HOSTNAME=VM-0-10-centos TERM=linux SHELL=/bi
   33 pts/1    R+     0:00 ps ef XDG_SESSION_ID=364 HOSTNAME=VM-0-10-centos SHELL=/bin/bash TERM=lin
[root@black-container code]# exit
exit
主要进程结束
[root@VM-0-10-centos code]# hostname
VM-0-10-centos
[root@VM-0-10-centos code]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 52:54:00:0b:f6:a2 brd ff:ff:ff:ff:ff:ff
    inet 172.21.0.10/20 brd 172.21.15.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:fe0b:f6a2/64 scope link
       valid_lft forever preferred_lft forever
[root@VM-0-10-centos code]# ps ef
  PID TTY      STAT   TIME COMMAND
29525 pts/1    Ss     0:00 -bash USER=root LOGNAME=root HOME=/root PATH=/usr/local/sbin:/usr/local/b
 3544 pts/1    R+     0:00  \_ ps ef XDG_SESSION_ID=364 HOSTNAME=VM-0-10-centos TERM=linux SHELL=/bi
28880 pts/0    Ss+    0:00 -bash USER=root LOGNAME=root HOME=/root PATH=/usr/local/sbin:/usr/local/b
```

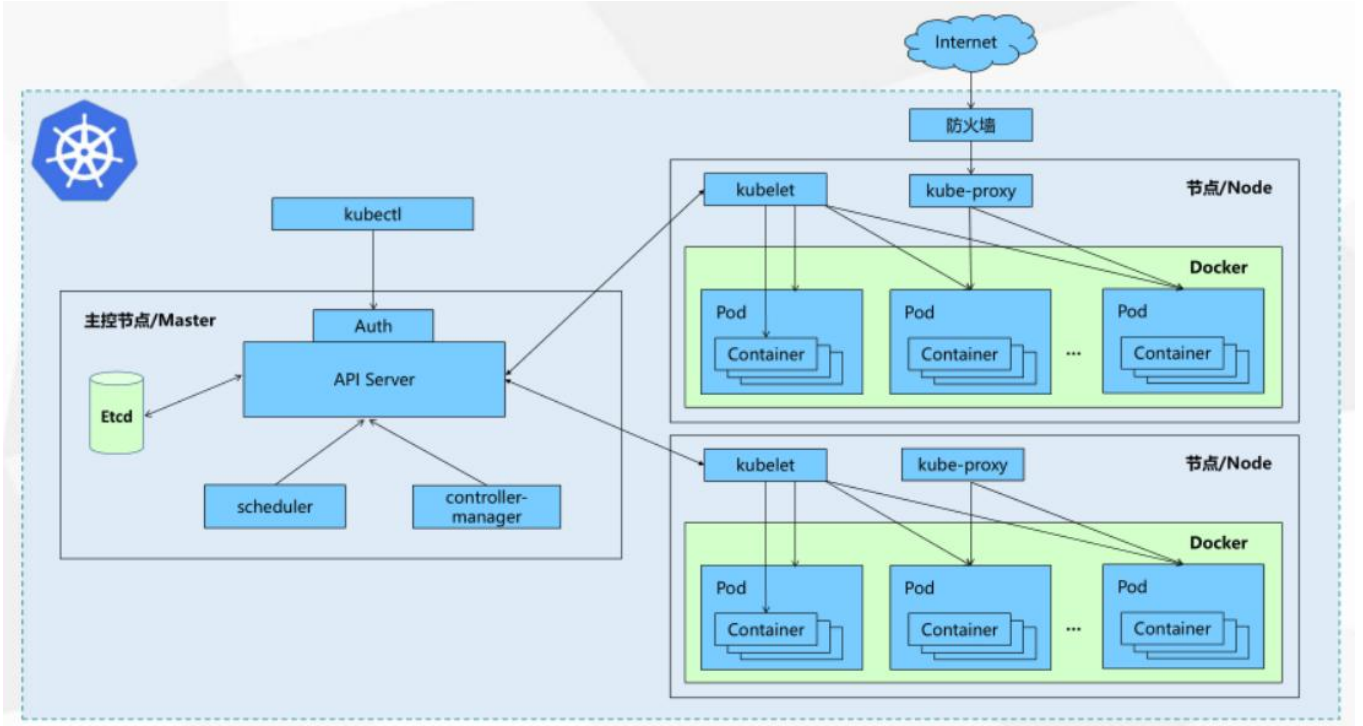
为什么要设计健壮系统？

让你的应用微服务化、无状态化、容器化

Docker架构



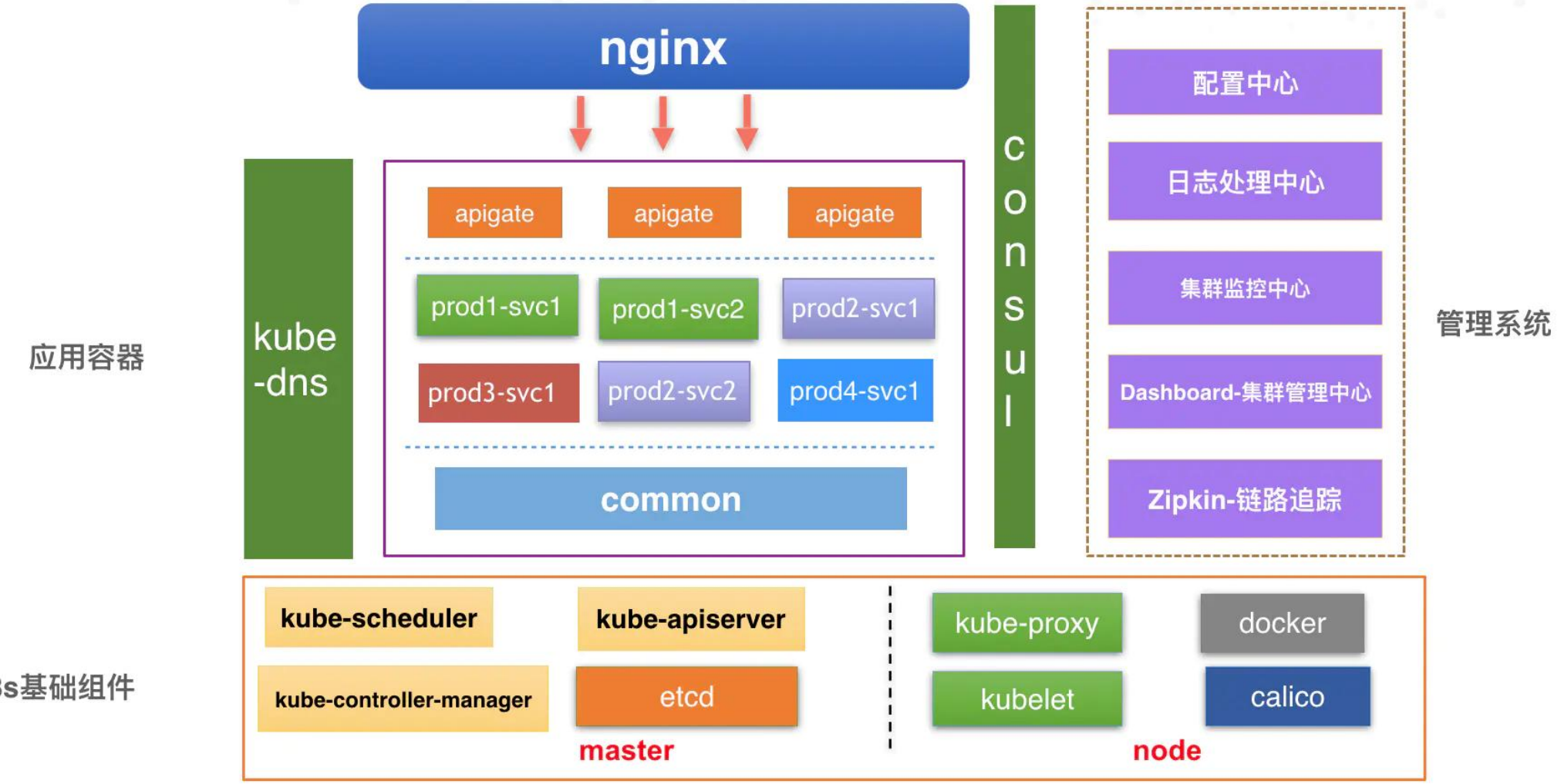
Kubernetes 架构



为什么要设计健壮系统？

让你的应用微服务化、无状态化、容器化

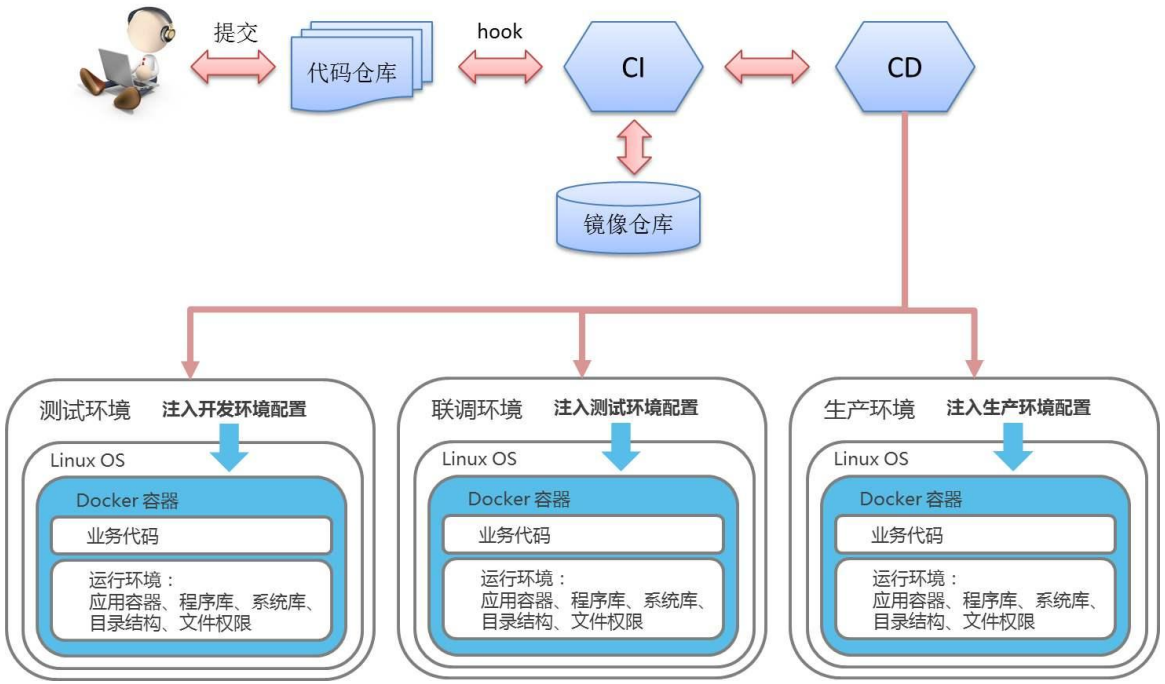
微服务+容器化架构

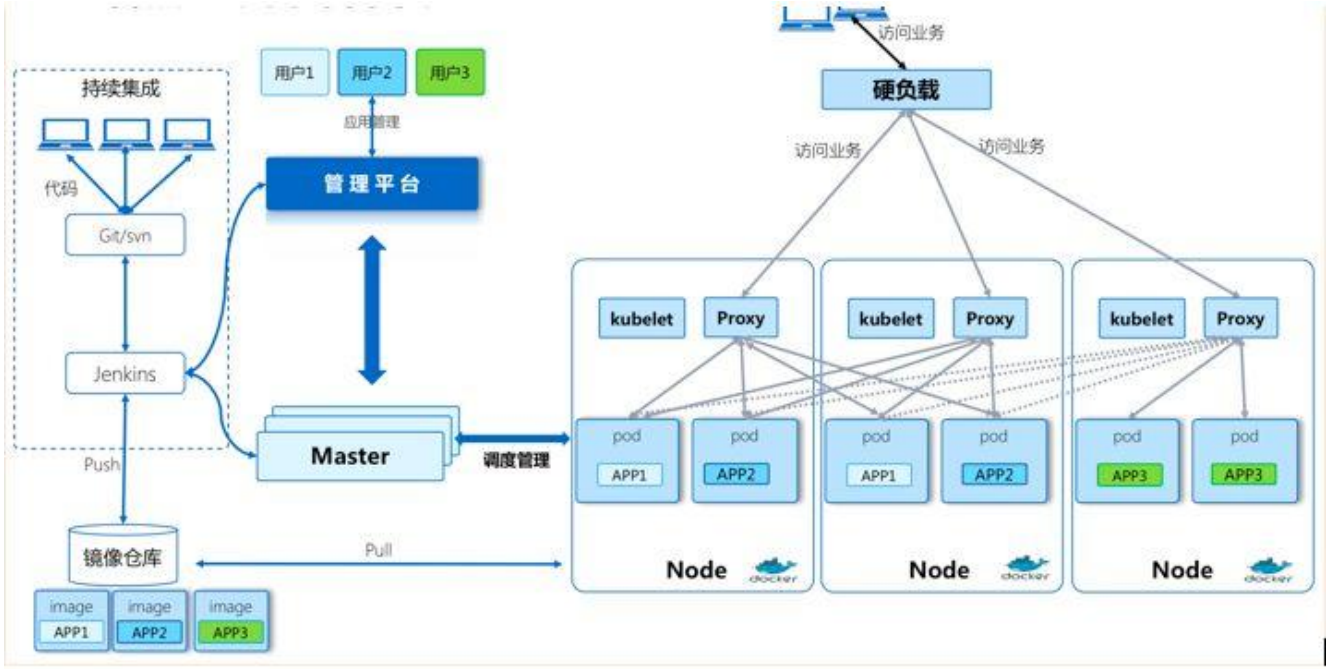


为什么要设计健壮系统？

让你的应用微服务化、无状态化、容器化

微服务+容器化
测试运维流





为什么要设计健壮系统？

系统每个细节都是需要可量化的，不能是模糊不明确的

核心思路

比如单个服务的QPS能力（预计流量需要多少服务器)、并发连接数（系统设置、系统承载连接）、单个进程内存占用、线程数、网络之间访问延迟时间（服务器之间延迟、机房到机房的延迟、客户端到服务器的延迟）、各种硬件性能参数（磁盘IO、服务器网卡吞吐量等等）。

QPS计算公式

原理：

- 每天80%的访问集中在20%的时间里，这20%时间叫做峰值时间：公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)
- 机器：峰值时间每秒QPS / 单台机器的QPS = 需要的机器
- QPS = 总请求数 / (进程总数 * 请求时间)

场景：

- 每天300w PV 的在单台机器上，这台机器需要多少QPS：(3000000 * 0.8) / (86400 * 0.2) = 139 (QPS)
- 如果一台机器的QPS是58，需要几台机器来支持：服务器数量 = ceil(每天总PV / 单台服务器每天总PV ， 139 / 58 = 3)

为什么要设计健壮系统？

选择合适的编程语言

考虑思路

1. 选择行业比较主流，容易维护和招聘的相关岗位的编程语言
2. 选择方便维护适合自己业务应用场景的编程语言
3. 选择自己目前团队能够Cover住或者通过一定学习能够Cover住的语言
4. 语言体系尽量一致，不要碎片化，否则维护会成为灾难

选择建议

API接口开发：

- 建议选择方便高性能、容易部署维护的语言，比如Golang、PHP、Java等语言，首选Golang，性能高，部署方便（一个执行文件搞定），Java系比较笨重要想清楚

内部系统开发

- 建议选择方便快速上线维护的语言，比如 PHP/Nodejs/Python 都是不错的选择；

终端开发语言

- 如果是桌面开发C++/C#都是不错的选择，Android选择 Java/Kotlin，iOS开发选择 OC/Swift，如果需要跨端开发 CPP+Qt，或者是采用 Flutter、RN、Electron或者浏览器嵌入版都是不错选择

运维工具或数据分析

- 这类开发建议简洁明快外部依赖少的语言，比如Python、Golang都是不错的选择

架构设计核心思想

在架构设计中，没有最好的架构，

只有适合业务形态的架构，

只有持续优化进步的架构。

三、程序实现关注哪些原则

程序实现关注哪些原则？

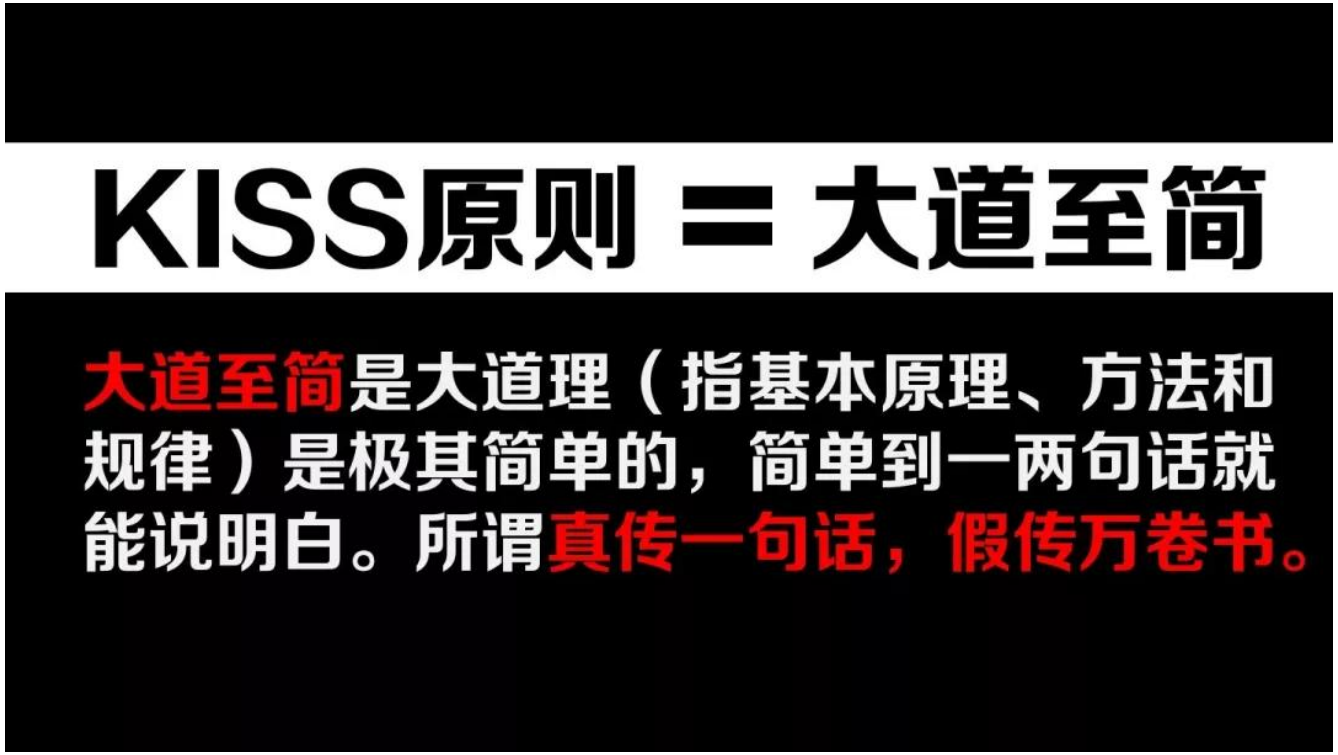
理解业务需求 & 遵循 K.I.S.S 原则

充分理解你的业务需求

- 保证理解业务后，整个程序设计符合需求或者未来几个月可以扩展，既不做过度设计，也不做各种临时硬编码。

代码核心原则：KISS（Keep it simple,stupid）

- 来自于Unix编程艺术，你的东西必须足够简单足够愚蠢，好处非常多，比如容易读懂，容易维护交接，出问题容易追查等等。
- 长期来看复杂的东西都是没有生命力的。（x86 vs ARM / 微型服务器 vs 大型机 / 北欧简约风 vs 欧洲皇室风 / 现在服装 vs 汉服）



程序实现关注哪些原则？

遵守编码规范，代码设计通用灵活

- 学会通过用函数和类进行封装（高内聚、低耦合）、如何定义函数，缩进方式，返回参数定义，注释如何定义、减少硬编码（通过配置、数据库存储变量来解决）。
- 最好让你的代码帮助你生成代码，重复代码自动化

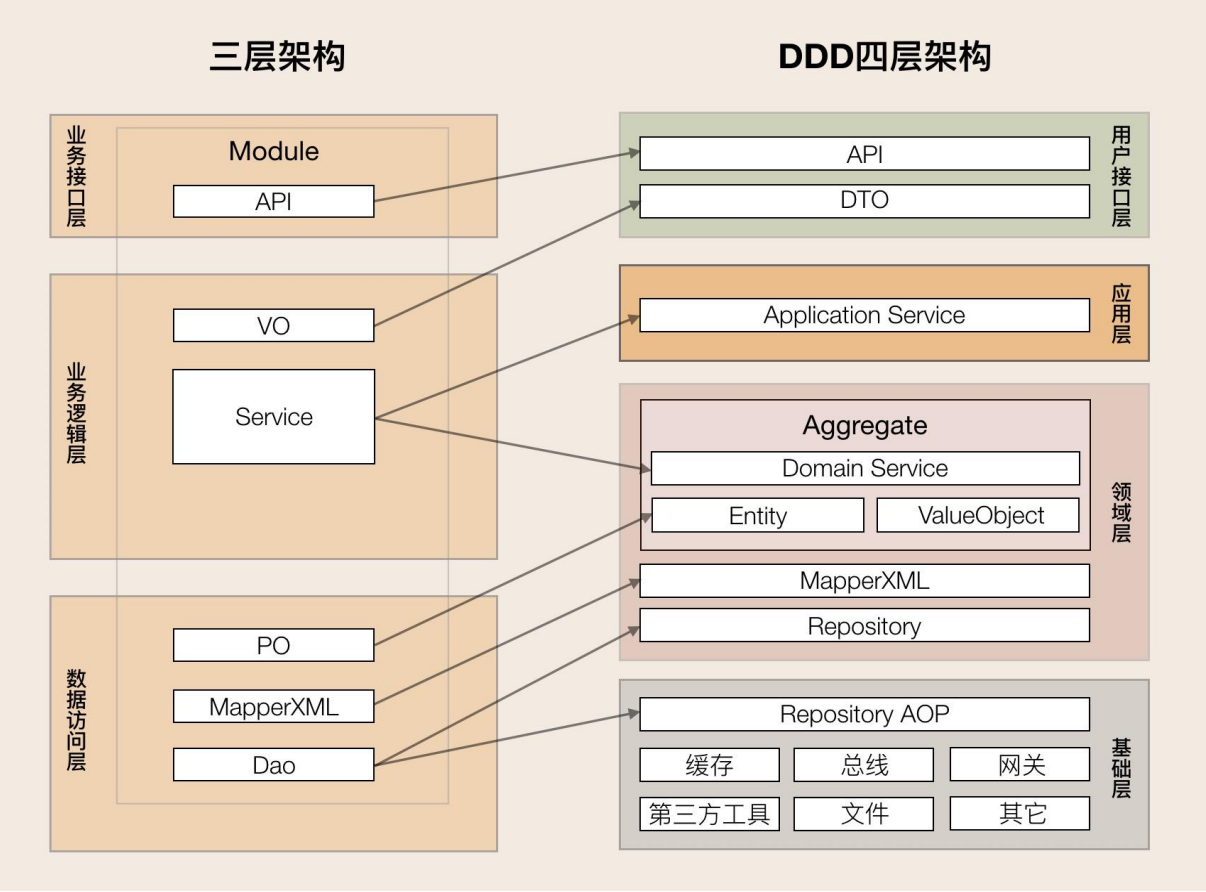
```php
/**
 * 给一个或者多个字段进行自增操作
 *
 * @example  1. $model->incr("cnt", "user_id=9527")   2. $model
 * @param  $field 需要增的字段，如果是传递的一个字段名字字符串，就
 * @param $where [ field1 = 1111111 ]  OR  field1 = xxxx
 * @return
 */
public function incr($field, $where) {
    $db = $this->getDbConn();

    $field_map = $this->getFieldMap();

    $sql = 'UPDATE ' . $this->getOpTableName() . ' SET ';   //

    //简单的一个字段的，默认是 +1
    if ( is_string($field) ) {
        //如果字段不存在直接报错
        if ( !isset($field_map[$field]) ) {
            return false;
        }
        $sql .= " $field = $field + 1 ";
    }
    //数组情况直接拼接 field1=field1+1,field2=field2+1 格式
    else if (is_array($field)) {
        $sql .= '  ';
        foreach ($where as $_fkey => $_fval) {
            //如果字段不存在
            if ( !isset($field_map[$_fkey]) ) {
                return false;
            }
            $sql .= " $_fkey = $_fkey + ".intval($_fval).",";
        }
    }
    else {
        return false;
    }
```

```php
public $_arrFieldMap = array(
    //主键，用作商户流水
    "payment_id" => array(
        "pdo_type"            => \PDO::PARAM_INT,
        "auto_increment"      => 1,
        "data_type"           => "int",
        "key_type"            => "PRI",
        "field_type"          => "int(11)",
        "field_comment"       => "主键，用作商户流水"
    ),
    //订单ID
    "fk_order_id" => array(
        "pdo_type"            => \PDO::PARAM_INT,
        "auto_increment"      => 0,
        "data_type"           => "int",
        "key_type"            => "MUL",
        "field_type"          => "int(11)",
        "field_comment"       => "订单ID"
    ),
    //付款用户ID
    "fk_user_id" => array(
        "pdo_type"            => \PDO::PARAM_INT,
        "auto_increment"      => 0,
        "data_type"           => "int",
        "key_type"            => "",
        "field_type"          => "int(11)",
        "field_comment"       => "付款用户ID"
    ),
    //alipay - 支付宝手机支付，alipay_wap - 支付宝手机网页支付，alipay_qr
    "pay_type" => array(
        "pdo_type"            => \PDO::PARAM_STR,
        "auto_increment"      => 0,
        "data_type"           => "varchar",
        "key_type"            => "",
        "field_type"          => "varchar(16)",
        "field_comment"       => "alipay - 支付宝手机支付，alipay_wap -
    ),
```

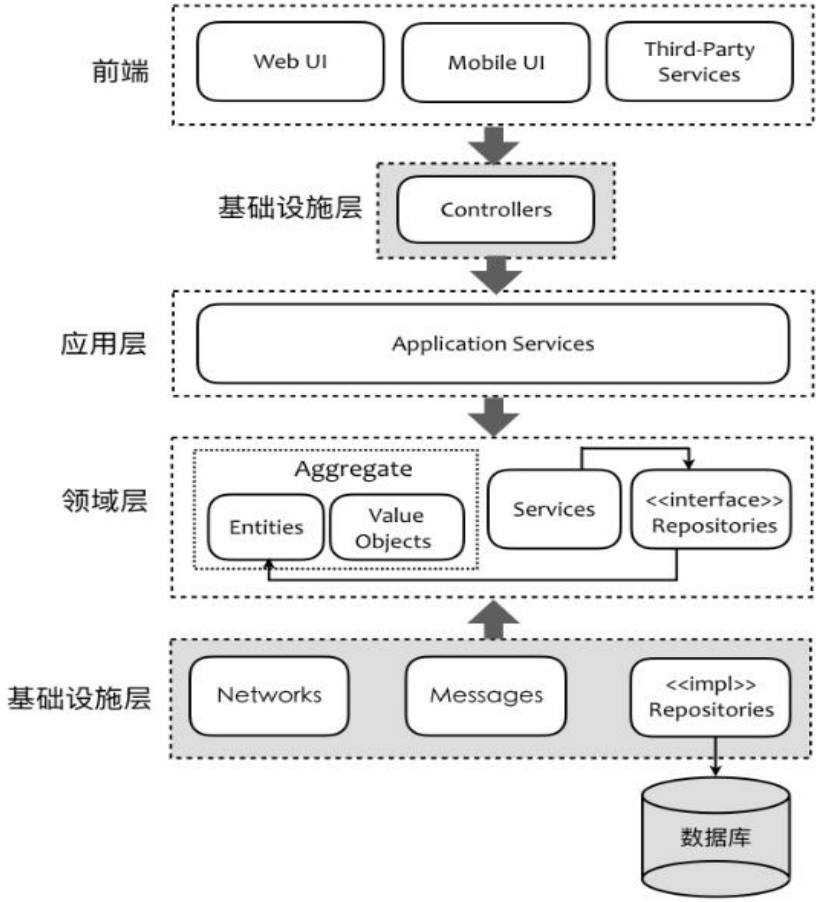
程序实现关注哪些原则？

用合适的设计模式构建代码和架构

善于各种业务抽取模型方法，比如领域驱动设计（DDD）方法，或者更多适合目前场景的设计模式，比如MVC模式、事件消息驱动模式 等等符合业务场景的设计方法



领域驱动设计DDD模式

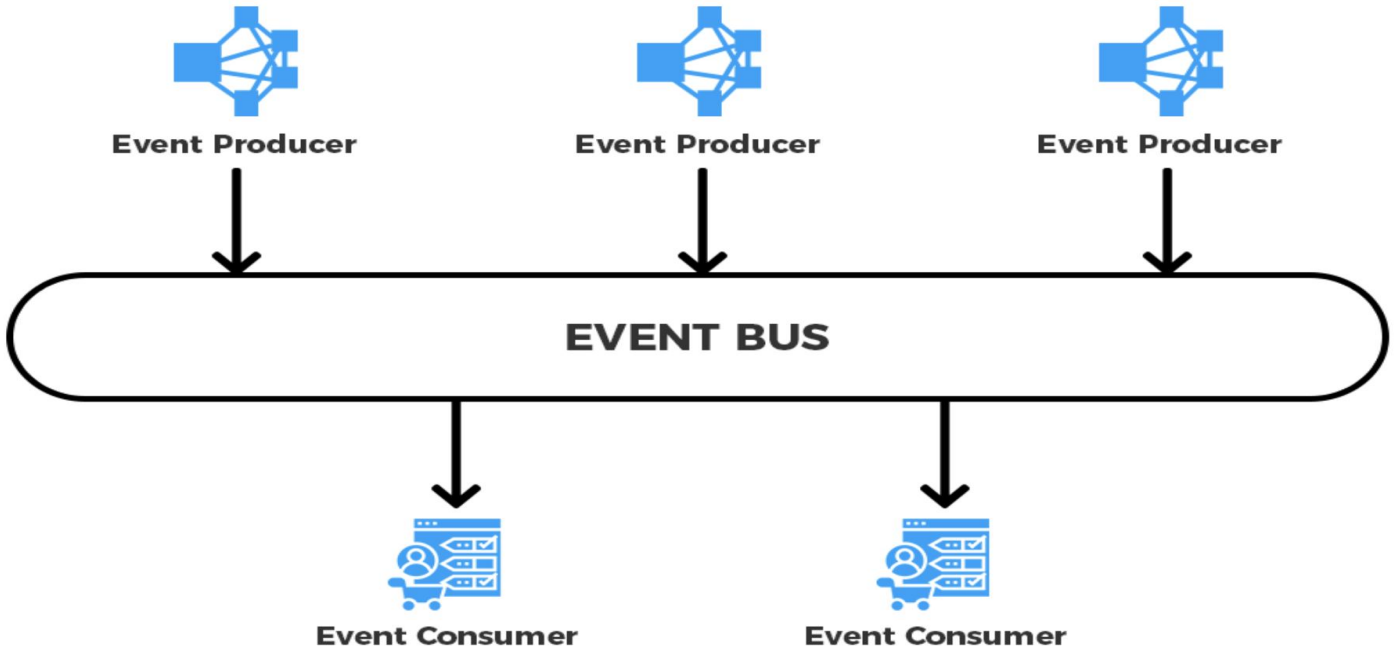


程序实现关注哪些原则？

用合适的设计模式构建代码和架构

善于各种业务抽取模型方法，比如领域驱动设计（DDD）方法，或者更多适合目前场景的设计模式，比如MVC模式、事件消息驱动模式 等等符合业务场景的设计方法

基于消息事件驱动设计

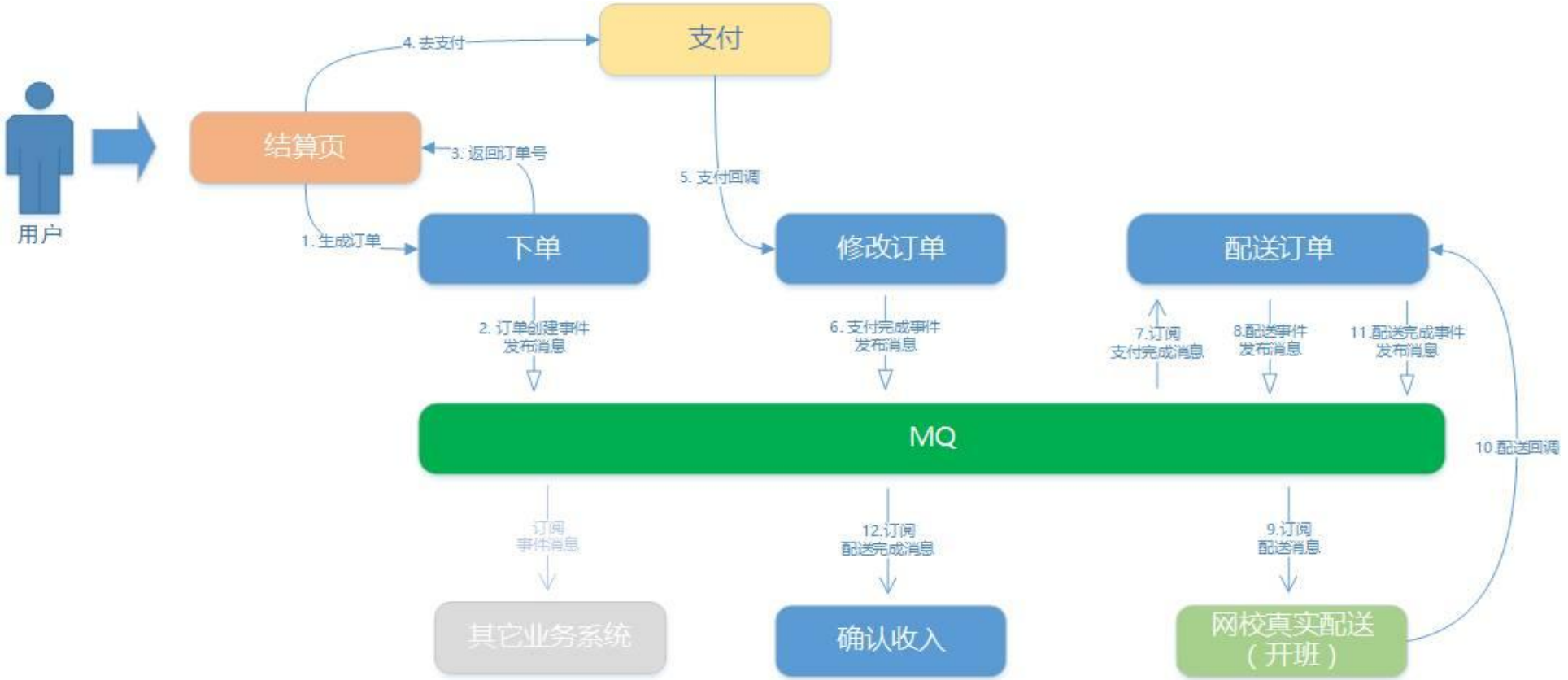


程序实现关注哪些原则？

用合适的设计模式构建代码和架构

善于各种业务抽取模型方法，比如领域驱动设计（DDD）方法，或者更多适合目前场景的设计模式，比如MVC模式、事件消息驱动模式 等等符合业务场景的设计方法

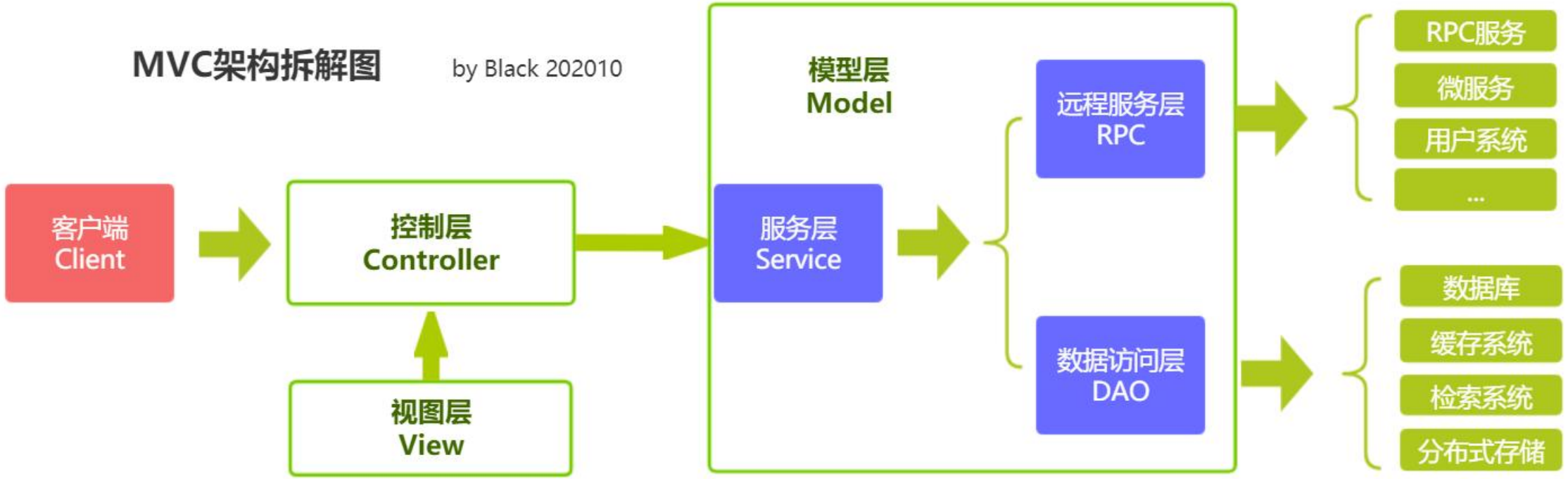
基于消息事件驱动设计



程序实现关注哪些原则？

设计模式和代码结构需要清晰

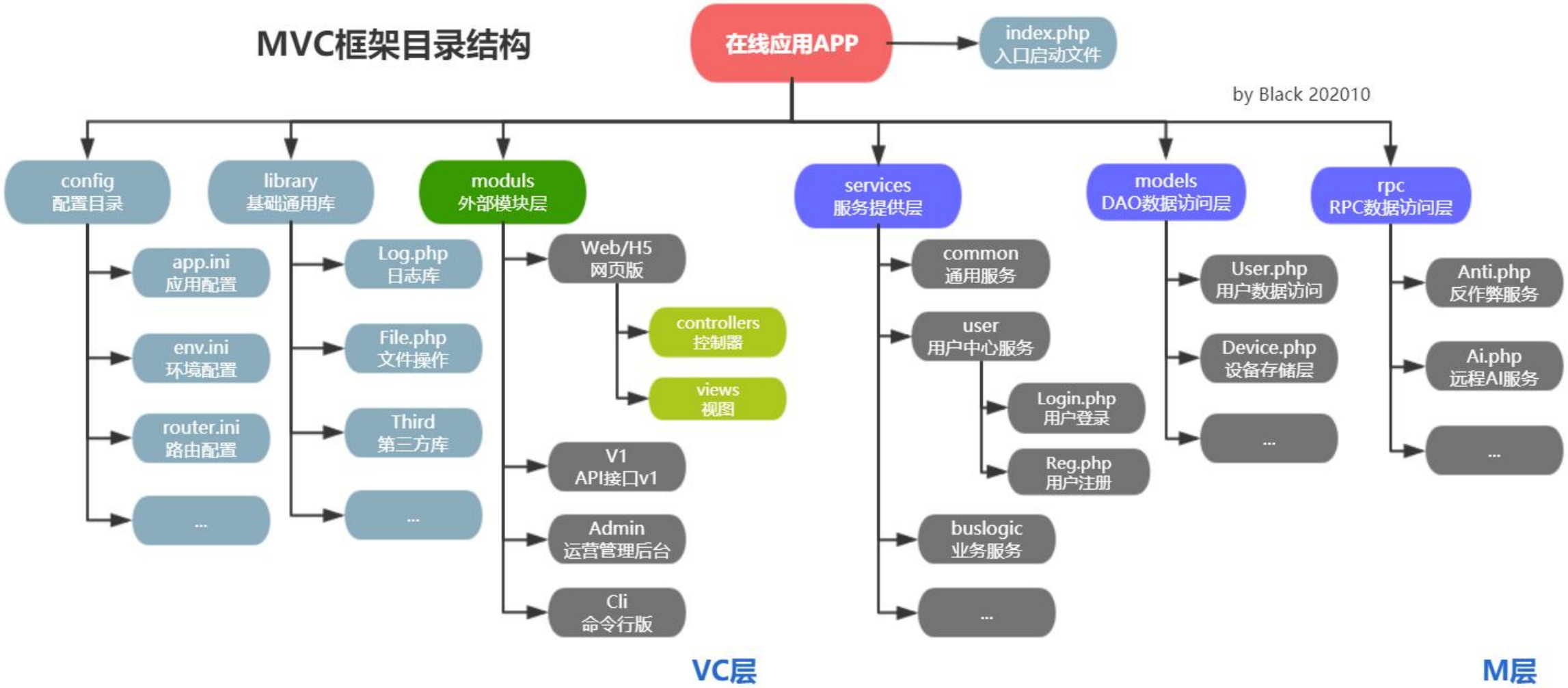
比如我们常规使用的MVC设计模式，为了就是把各个层次代码区分开。（M干好数据读取或者接口访问的事儿，C干好变量收发基本教研，V干好模板渲染或者api接口输出；M可以拆分成为：DAO数据访问层和Service某服务提供层）；



程序实现关注哪些原则？

设计模式和代码结构需要清晰

参考一个常见的MVC框架目录架构（基于yaf框架）：



程序实现关注哪些原则？

程序中一定要写日志，代码日志要记录清晰

程序里多写日志，包括Info、Debug、Waring、Trace等等，调用统一的日志库，不要害怕多写日志）

```php
<?php
//程序里关键的日志都要记录（Debug/Notice/Warning/Error 等信息可以打印，warning/error 信息是一定要打印的
SeasLog::debug('TRACE_ID:{traceId}; this is a {userName} debug',array('{traceId}'=>9527, '{userName}' => 'Black'));
SeasLog::notice('TRACE_ID:{traceId}; this is a notice msg', array('{traceId}'=>9527));
SeasLog::warning('TRACE_ID:{traceId}; this is a warning', array('{traceId}'=>9527));
SeasLog::error('TRACE_ID:{traceId}; a error log', array('{traceId}'=>9527));
```

```
2018-08-18 19:24:49 | DEBUG | 4184 | 5b780201a82ef | 1534591489.689 | test begin!
2018-08-18 19:24:49 | ERROR | 4184 | 5b780201a82ef | 1534591489.689 | php version = 7.1.20,Thread Safety
= disabled,REMOTE_ADDR = 124.202.180.26
2018-08-18 19:24:49 | CRITICAL | 4184 | 5b780201a82ef | 1534591489.689 | Notice - type:8 - file:/var/www/
html/index.php - line:13 - msg:Undefined variable: name
2018-08-18 19:24:49 | CRITICAL | 4184 | 5b780201a82ef | 1534591489.689 | Warning - type:2 - file:/var/www
/html/index.php - line:14 - msg:Wrong parameter count for intval()
2018-08-18 19:24:49 | CRITICAL | 4184 | 5b780201a82ef | 1534591489.689 | Exception - type:0 - file:/var/w
ww/html/index.php - line:19 - msg:please enter the params
2018-08-18 19:24:49 | DEBUG | 4184 | 5b780201a82ef | 1534591489.689 | test over!
```

程序实现关注哪些原则？

稳健性编程小技巧（个人最佳小实践）

1. 代码里尽量不要使用else（超级推荐，Unix编程艺术书籍推荐方法）
2. 所有的循环必须有结束条件或约定，并且不会不可控
3. 不要申请超级大的变量或内存造成资源浪费
4. 无论静态还是动态语言，内存或对象使用完以后尽量及时释放
5. 输入数据务必要校验，用户输入数据必须不可信。
6. 尽量不要使用异步回调的方式（容易混乱，对js和nodejs的鄙视，对协程机制的尊敬）

使用else

```
1  //获取一个整形的值 （使用else，共12行）
2  function getIntValue($val) {
3      if ( $val != "" ) {
4      $ret = (int)$val;
5          if ($ret != 0 ) {
6              return $ret;
7          } else {
8              return false;
9          }
10     } else {
11         return false;
12     }
13 }
```

不用else

```
1  //获取一个整形的值（不使用else，共10行）
2  function getIntValue($val) {
3      if ( $val == "" ) {
4          return false;
5      }
6      $ret = (int)$val;
7      if ($ret == 0 ) {
8          return false;
9      }
10      return $ret;
11 }
```

所有内外部访问都必须有超时机制：保证不连锁反应雪崩

超时是保证我们业务不会连带雪崩的很关键的地方，比如我们在访问后端资源或外部服务一定要经常使用超时操作。

超时细化下来，一般会包括很多类型：连接超时、读超时、写超时、通用超时 等等区分；一般超时粒度大部分都是秒为单位，对于时间敏感业务都是毫秒为单位，建议以毫秒(ms)为单位的超时更可靠，但是很多服务没有提供这类超时操作接口。

cURL超时

```php
function http_call($url)
    $ch = curl_init($url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    //注意，毫秒超时一定要设置这个
    curl_setopt($ch, CURLOPT_NOSIGNAL, 1);
    //超时毫秒，cURL 7.16.2中被加入。从PHP 5.2.3起可使用
    curl_setopt($ch, CURLOPT_TIMEOUT_MS, 200);
    $data = curl_exec($ch);
    $curl_errno = curl_errno($ch);
    $curl_error = curl_error($ch);
    curl_close($ch);
}
http_call('http://example.com')
```

程序实现关注哪些原则？

所有内外部访问都必须有超时机制：保证不连锁反应雪崩

Swoole 4.x 超时 - MySQL

```php
//Swoole 里通用超时设置（针对TCP协议情况，包含通用超时、连接超时、读写超时）
Co::set([
    'socket_timeout' => 5,
    'socket_connect_timeout' => 1,
    'socket_read_timeout' => 1,
    'socket_write_timeout' => 1,
]);

//Swoole 4.x协程方式访问MySQL
Co\run(function () {
    $swoole_mysql = new Swoole\Coroutine\MySQL();
    $swoole_mysql->connect([
        'host'     => '127.0.0.1',
        'port'     => 3306,
        'user'     => 'user',
        'password' => 'pass',
        'database' => 'test',
        'timeout'     => '1',
    ]);
    $res = $swoole_mysql->query('select sleep(1)');
    var_dump($res);
});
```

Swoole 4.x 超时 - Redis

```php
//Swoole 4.x协程方式访问Redis
Co\run(function () {
    $redis = new Swoole\Coroutine\Redis();
    $redis->setOptions(
        'connect_timeout'    => '1',
        'timeout'    => '1',
    );
    $redis->connect('127.0.0.1', 6379);
    $val = $redis->get('key');
});
```

程序实现关注哪些原则？

所有内外部访问都必须有超时机制：保证不连锁反应雪崩

MySQLi超时

- 访问MySQL超时处理（非Swoole情况），调用mysqli扩展方式：mysql 默认在扩展层面没有把很多超时操作暴露给前台，所以需要用一些隐藏方式。
- 目前大部分主流框架都没有提供mysql超时配置，包括Laravel、Symfony、Yii 等框架都没有提供，因为都是基于底层mysqli或pdo等扩展。

```php
<?php
//自己定义读写超时常量
if (!defined('MYSQL_OPT_READ_TIMEOUT')) define('MYSQL_OPT_READ_TIMEOUT',  11);
if (!defined('MYSQL_OPT_WRITE_TIMEOUT')) define('MYSQL_OPT_WRITE_TIMEOUT', 12)

//设置超时
$mysqli = mysqli_init();
$mysqli->options(MYSQL_OPT_READ_TIMEOUT, 3); //读超时，没办法超过3秒
$mysqli->options(MYSQL_OPT_WRITE_TIMEOUT, 1); //写超时，最小可设置为1秒

//连接数据库
$mysqli->real_connect("localhost", "root", "root", "test");
//执行查询 sleep 1秒不超时
printf("Host information: %s/n", $mysqli->host_info);

//执行查询 sleep 1秒不超时
printf("Host information: %s/n", $mysqli->host_info);
if (!($res=$mysqli->query('select sleep(1)'))) {
    echo "query1 error: ". $mysqli->error ."/n";
} else {
    echo "Query1: query success/n";
}
```

程序实现关注哪些原则？

所有内外部访问都必须有超时机制：保证不连锁反应雪崩

流超时

```php
//fopen & file_get_contents 访问HTTP超时控制
//设置超时和压入上下文环境
$timeout = array(
    'http' => array(
        'timeout' => 5 //设置一个超时时间，单位为秒
    )
);
$ctx = stream_context_create($timeout);
//fopen
if ($fp = fopen("http://example.com/", "r", false, $ctx)) {
  while( $c = fread($fp, 8192)) {
    echo $c;
  }
  fclose($fp);
$text = file_get_contents("http://example.com/", 0, $ctx);
echo $text;
```

延伸阅读：

https://blog.csdn.net/heiyeshuwu/article/details/7841366

```php
// ## fsockopen访问HTTP ##
$timeout = 5; //超时5秒
$fp = fsockopen("example.com", 80, $errno, $errstr, $timeout);
if ($fp) {
        fwrite($fp, "GET / HTTP/1.0\r\n");
        fwrite($fp, "Host: example.com\r\n");
        fwrite($fp, "Connection: Close\r\n\r\n");
        stream_set_blocking($fp, true);   //重要，设置为非阻塞模式
        stream_set_timeout($fp,$timeout);   //设置超时
        $info = stream_get_meta_data($fp);
        while ((!feof($fp)) && (!$info['timed_out'])) {
                $data .= fgets($fp, 4096);
                $info = stream_get_meta_data($fp);
                ob_flush;
                flush();
        }
        if ($info['timed_out']) {
                echo "Connection Timed Out!";
        }
        else {
                echo $data;
        }
}
```

程序实现关注哪些原则？

访问外部资源要谨慎，不要引起后端服务雪崩

- 访问外部关键资源，比如存储、DB、Cache等一定要谨慎，避免大规模并发访问，减少雪崩
- 要善于合并请求（Merge-Request），在同时多个进程/协程访问后端资源的同一数据的时候，尽量一个时间内只有一个请求，避免变成DDoS
- 要学会使用批量获取（Multi-Get），单个请求可以批量获取数据，减少请求太多导致夯住
- 对于用户频繁操作能够善于使用内存临时缓存提升访问性能，减少后端频繁访问

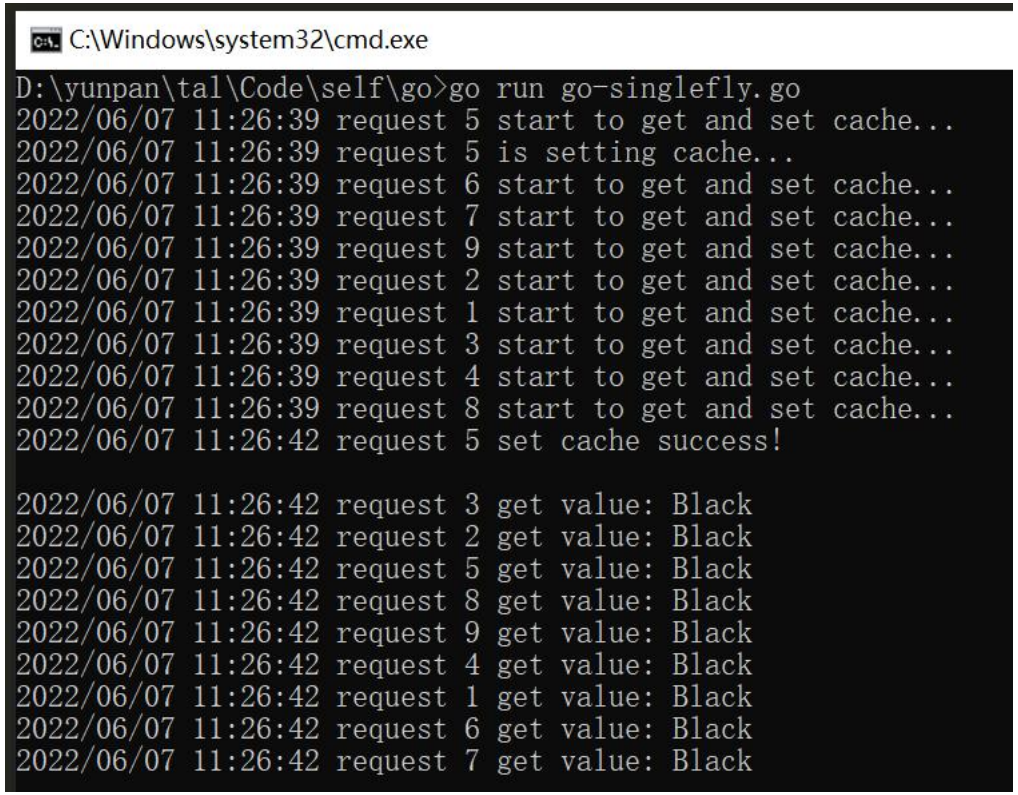
```go
package main
import (
    "log"
    "time"
    "golang.org/x/sync/singleflight"
)

//使用 Go SingleFlight 合并请求
func main() {
    var singleSetCache singleflight.Group

    // 模拟读写缓存场景函数
    getAndSetCache:=func (requestID int,cacheKey string) (string, error) {
        log.Printf("request %v start to get and set cache...",requestID)

        //do的入参key，可以直接使用缓存的key，这样同一个缓存，只有一个协程会去读DB
        value,_, _ := singleSetCache.Do(cacheKey, func() (ret interface{}, err error) {
            log.Printf("request %v is setting cache...",requestID)
            time.Sleep(3*time.Second)
            log.Printf("request %v set cache success!\n\n",requestID)
            return "Black",nil
        })
        return value.(string),nil
    }

    //模拟多个协程同时请求
    cacheKey:="cacheKeyName"
    for i:=1;i<10;i++{
        go func(requestID int) {
            value,_:=getAndSetCache(requestID,cacheKey)
            log.Printf("request %v get value: %v",requestID,value)
        }(i)
    }
    time.Sleep(20*time.Second)

}
```

程序实现关注哪些原则？

访问外部资源要谨慎，不要引起后端服务雪崩

关键技术策略：

- 在PHP中可以使用某个远程Cache中的锁来合并请求，比如多个PHP进程访问本地共享内存锁或远程某个锁来保证同一时间只能一个请求发往后端
- 在Golang中可以使用 sync/WaitGroup或内置的SingleFlight来保证合并请求访问后端只有一个请求，减少后端雪崩概率



程序实现关注哪些原则？

成熟稳定的SQL语言使用习惯和技巧

- 所有SQL语句都必须有约束条件
- SQL查询里抽取字段中明确需要抽取的字段
- 熟知各种SQL操作的最佳实践技巧，包括不限于：常用字段建立索引（单表不超过6个）、尽量减少OR、尽量减少连表查询、尽量不要SQL语句中使用函数（datetime之类）、使用exists代替in、使用explain观察SQL运行情况等等。

SQL好习惯

a. 所有SQL语句都必须有约束条件

常规习惯：

```
1 SELECT uid,uname,email,gender FROM user WHERE uid = '9527'
```

好习惯，增加对应的WHERE条件和LIMIT限制

```
1 SELECT uid,uname,email,gender FROM user WHERE uid = '9527' LIMIT 1
```

b. SQL查询里抽取字段中明确需要抽取的字段

常规习惯：

```
1 SELECT * FROM user WHERE uid = '9527'
```

良好习惯，需要什么字段提取什么字段：

```
1 SELECT uid,uname,email,gender FROM user WHERE uid = '9527' WHERE 1 LIMIT 1
```

主要受约束的是一些mysql的配置相关，包括：max_allowed_packet 之类的会超过限制或者是把网卡带宽打满；

延伸阅读：

https://blog.csdn.net/jie_liang/article/details/77340905

程序实现关注哪些原则？

开发中时刻要记得代码安全

- 我们系统在完成业务开发的基础上，还需要考虑代码安全问题，大部分时候安全和方便中间会存在冲突，但是因为一个不安全的系统，对业务的伤害是巨大的，轻则被非法用户“薅羊毛”，重则服务器被攻陷，整个数据遭到泄露或者遭受恶意损失。

- 我们常见遇到的安全问题包括：SQL注入、XSS、CSRF、URL跳转漏洞、文件上传下载漏洞等等，很多在我们只是关注业务实现不关注安全的时候问题都会出现。

延伸阅读：

https://mp.weixin.qq.com/s/UOQa6yj-7bxUpb_1Xr4tYQ

https://wenku.baidu.com/view/42534969fab069dc502201d5

https://mp.weixin.qq.com/s/RrBVaWve6StQbksSMFEm4w

安全解决方案

安全问题	常见解决方法
SQL注入	输入参数校验：intval、is_numeric、htmlspecialchars、trim 数据入库格式化：PDO::prepare、mysqli_real_escape_string
XSS	过滤危险的HTML/JS等输入参数和显示内容，在js代码中对HTML做相应编码，多使用正则或者 htmlspecialchars、htmlspecialchars_decode 等处理函数；
CSRF	Client和Server交互采用token校验操作，敏感操作判断来源IP等
URL跳转	跳转目标URL必须进行校验，或者采用URL白名单机制
文件上传下载	限制文件上传大小(upload_max_filesize / post_max_size配置) 采用可靠上传的组件（Flash/JS组件）；检查上传文件类型（扩展名+内容头），控制服务器端目录和文件访问权限
配置安全	PHP环境配置做好安全配置，屏蔽敏感函数：eval / exec / system / get_included_files ； 敏感配置关闭：register_globals / allow_url_fopen / allow_url_include/ safe_mode / magic_quotes 等； 资源管控配置： max_execution_time / max_input_time /memory_limit / open_basedir / upload_tmp_dir 等
数据库安全	数据库访问不能用root账户，不同库不同的访问用户，读写账户分离； 敏感库表需要特殊处理（比如用户库表密码库表等加盐存储）；
服务器安全	DDoS攻击防范（流量清洗、黑白名单）、服务安全运行权限（非root运行php进程）、服务器之间访问白名单机制

程序实现关注哪些原则？

调优后端各服务的性能配置

- 开发语言只是一个粘合剂，整个过程是把前端用户操作进行逻辑处理，粘合后端存储和各种RPC服务的数据进行展现输出。除了你的PHP服务或代码、SQL执行效率高，同样也需要考虑前后端各个服务的性能是非常优化高性能的。
- 我把一些关键的Linux系统到各个各个服务一些关键性能相关选项简单罗列一下。

推荐服务配置选项

服务类型	核心性能影响配置
Linux系统	1. 并发文件描述符 永久修改： /etc/security/limits.conf * soft nofile 1000000 * hard nofile 1000000 Session临时修改：ulimit -SHn 1000000 3. 进程数量限制 永久修改：/etc/security/limits.d/20-nproc.conf * soft nproc 4096 root soft nproc unlimited 4. 文件句柄数量 临时修改：echo 1000000 > /proc/sys/fs/file-max 永久修改：echo "fs.file-max = 1000000" >>/etc/sysctl.conf 5. 网络TCP选项，关注 *somaxconn/*backlog/*mem*系列/*time*系列等等 6. 关闭SWAP交换分区（服务器卡死元凶）： echo "vm.swappiness = 0">> /etc/sysctl.conf
Nginx/OpenResty	Nginx Worker性能： worker_processes 4; worker_cpu_affinity 01 10 01 10; worker_rlimit_nofile 10240; worker_connections 10240; Nginx网络性能： use epoll; sendfile on; tcp_nopush on; tcp_nodelay on; keepalive_timeout 30; proxy_connect_timeout 10; Nginx缓存配置： fastcgi_buffer_size 64k; client_max_body_size 300m; client_header_buffer_size 4k; open_file_cache max=65535 inactive=60s; open_file_cache_valid 80s; proxy_buffer_size 256k; proxy_buffers 4 256k; proxy_cache*

PHP/FPM	listen.backlog = -1 #backlog数，-1表示无限制
	rlimit_files = 1024 #设置文件打开描述符的rlimit限制
	rlimit_core = unlimited #生成core dump文件限制，受限于linux系统
	pm.max_children = 256 #子进程最大数
	pm.max_requests = 1000 #设置每个子进程重生之前服务的请求数
	request_terminate_timeout = 0 #设置单个请求的超时中止时间
	request_slowlog_timeout = 10s #当一个请求该设置的超时时间后

MySQL/MariaDB	MySQL服务选项：
	wait_timeout=1800
	max_connections=3000
	max_user_connections=800
	thread_cache_size=64
	skip-name-resolve = 1
	open_tables=512
	max_allowed_packet = 64M
	
	MySQL性能选项：
	innodb_page_size = 8K #脏页大小
	innodb_buffer_pool_size = 10G #建议设置为内存80%
	innodb_log_buffer_size = 20M #日志缓存大小
	innodb_flush_log_at_trx_commit = 0 #事务日志提交方式，设置为0比较合适
	innodb_lock_wait_timeout = 30 #锁获取超时等待时间
	innodb_io_capacity = 2000 #刷脏页的频次默认200，高一些会让io曲线稳

Redis	maxmemory 5000mb #最大内存占用 maxmemory-policy allkeys-lru #达到内存占用后淘汰策略，存在热点数据，淘汰不咋访问的 maxclients 1000 #客户端并发连接数 timeout 150 #客户端超时时间 tcp-keepalive 150 #向客户端发送tcp_ack探测存活 rdbcompression no #磁盘镜像压缩，开启占用cpu rdbchecksum no #存储快照后crc64算法校验，增加10%cpu占用 vm-enabled no #不做数据交换

延伸阅读

善用常用服务的系统监测工具

系统检测常用工具

- 为了快速的监控我们的线上服务情况，需要能够熟练使用常用的性能监控的各种工具和指标。

延伸阅读

服务类型	常用工具或指令
Linux	top、vmstat、iostat、netstat、sar、nmon、dstat、iftop、free、df/du、tcpdump
PHP	Xdebug、xhprof、Fiery、第三方APM工具
MySQL	MySQL主要指令： show processlist show global variables like '%xxx%' show master status; show slave status; show status like '%xx%' 细节查询： 查看连接数：SHOW STATUS LIKE 'Thread_%'; 查看执行事务：SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX; 查看锁定事务：SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS; 每秒查询QPS：SHOW GLOBAL STATUS LIKE 'Questions'; #QPS = Questions / Uptime 每秒事务TPS：SHOW GLOBAL STATUS LIKE 'Com_%'; #TPS = (Com_commit + Com_rollback) / Uptime InnoDB Buffer命中率：show status like 'innodb_buffer_pool_read%'; #innodb_buffer_read_hits = (1 - innodb_buffer_pool_reads / innodb_buffer_pool_read_requests) * 100%
Redis	查看redis服务的各项状态：info / info stats / info CPU / info Keyspace 实时监控redis所有命令：monitor 查看redis慢日志：slowlog get 128 性能测试监控：redis-benchmark -h localhost -p 6379 -c 100 -n 100000

为什么要设计健壮系统？

健壮代码有哪些通用原则：10原则

1、 模块性原则：写简单的，通过干净的接口可被连接的部件。（比如类、函数，高内聚低耦合）

2、 清楚原则：清楚要比小聪明好。（代码中注释需要清晰明确，最好有历史迭代，不要耍小聪明，或者用一些奇怪的实现算法并且没注释）

3、 合并原则：设计能被其它程序连接的程序。（提供好输入输出参数，或者设计好的openapi，尽量让程序可以复用）

4、 简单原则：设计要简单；只有当你需要的时候，增加复杂性。（每个函数类要简单明确，不要太冗长）

5、 健壮性原则：健壮性是透明和简单的追随者。（透明+简单了，健壮就来了）

6、沉默补救原则：当一个程序没有异常的时候就只是记录，减少干扰或者啥都不说；当你必须失败的时候，尽可能快的吵闹地失败。（失败了一定要明确清晰的提示形式，包括错误代码，错误原因的信息，不要悄无声息）

7、经济原则：程序员的时间是宝贵的；优先机器时间节约它。（能够用内存+CPU搞定，不要过多纠结在算法上）

8、产生原则：避免手工堆砌；当你可能的时候，编写可以写程序的程序。（用程序来帮你实现重复的事儿）

9、优化原则：在雕琢之前先有原型；在你优化它之前，先让他可以运行。（先完成，再完美）

10、可扩展原则：为将来做设计，因为它可能比你认为来的要快。（尽量考虑你这个代码2，3，5年后才会重构，为未来负责，不要让后人骂你）

四、个人综合软素质

个人软素质

优秀程序员软素质

细心　认真　谨慎　精进　专业

1、细心：写完代码都需要重新Review一遍，变量名是否正确，变量是否初始化，每个SQL语句是否性能高超或者不会导致超时死锁；

2、认真：每个函数是否都自己校验过输入输出是满足预期的；条件允许，是否核心函数都具备单元测试代码；

3、谨慎：不要相信任何外部输入的数据，包括数据库、文件、缓存、用户HTTP提交各种变量，都需要严格校验和过滤。

4、精进：不要惧怕别人说你代码烂，必须能够持续被人吐槽下优化，在在我革新下优化；要学习别人优秀的代码设计思想和代码风格，持续进步。

5.专业：专业是一种工作态度，也是一种人生态度；代码要专业、架构要专业、变量名要专业、文档要专业、跟别人发工作消息邮件要专业、开会要专业、沟通要专业，等等，专业要伴随自己一生

个人软素质 优秀程序员软素质

持续改进

十四年前 -> 十一年前

```php
<?php
//------------------------------------------------------------------------
//      MySQLi Master/Slave数据库读写操作类
//
// 开发作者: heiyeluren
// 版本历史:
//          2006-09-20  基本单数据库操作功能，25 个接口
//          2007-07-30  支持单Master/多Slave数据库操作，29个接口
//          2008-09-07  修正了上一版本的部分Bug
//          2009-11-17  在Master/Slave类的基础上增加了强化单主机操作，
//                      增加了部分简洁操作接口和调试接口，优化了部分代码，
//                      本版本共42个接口
//                      2009-11-26      增加了部分调试和性能监控接口
//                      2009-12-13  整合到TMPHP项目中
//                      2009-12-19  修改mysqli_free_result判断bug
//                      2009-12-20  整个库移植为底层采用Mysqli API，增加接口，同时修改为PHP5对象方式
//
// 功能描述: 自动支持Master/Slave 读/写 分离操作，支持多Slave主机
//
//------------------------------------------------------------------------
```

参考地址：
https://github.com/heiyeluren/tmphp/blob/master/tmphp/lib/TM_DB_Mysqli.class.php
https://github.com/heiyeluren/HeiyelurenPHPFramework/blob/master/src/bases/Socket.class.php

追求专业

十三年前

```php
<?php
/*****************************************
 *  描述：Socket操作基础类
 *  作者：heiyeluren
 *  创建：2007-04-04 15:51
 *  修改：2007-04-09 19:48
 *****************************************/


//错误代码提示
define("__SOCKET_ERR_NO", -1);
define("__SOCKET_ERR_ADDR_INVALID", -2);
define("__SOCKET_ERR_PORT_INVALID", -3);
define("__SOCKET_ERR_NOT_CONN", -4);
define("__SOCKET_ERR_SET_BUFF_FAILED", -5);
define("__SOCKET_ERR_WRITE_FAILED", -6);
define("__SOCKET_ERR_READ_FAILED", -7);
define("__SOCKET_ERR_DATA_ERR", -8);


//包含基础异常处理类和检查类
include_once("Exception.class.php");
include_once("VerifyUtil.class.php");

/**
 * Socket操作基础类
 *
 * 包含基本的客户端Socket操作方法
 */
class Socket extends ExceptionClass
{
        /**
         * Socket数据流指针
         */
        var $fp = null;
```

愿你们在未来成为

最厉害的程序员

最优秀的架构师

CONTACT 联系我

个人微信

技术公众号

有问题可以联系我共同交流

TAL 好未来

Thanks

感谢聆听