架构师如何考虑软件的可持续迭代

讲师自我介绍

段和尘

“毕业于中科院，十二年从事移动端Android研发。

百度→创业→魅族→创业→腾讯→字节。

目前负责头条的Android客户端架构。

又失败了咋办？《在岁月中远行》”

几个小问题

1. 你所了解的架构是什么？

A. 一个完美的设计，能够解决各种软件问题
B. 没有完美的架构，需要不断对实现做重构

2. 你觉得架构师们每天都在干什么？

A. 每天都根据用户的需求，讨论如何做软件设计
B. 每天都在填坑，填不完的坑…

01 架构面临的问题

02 架构常见的手段

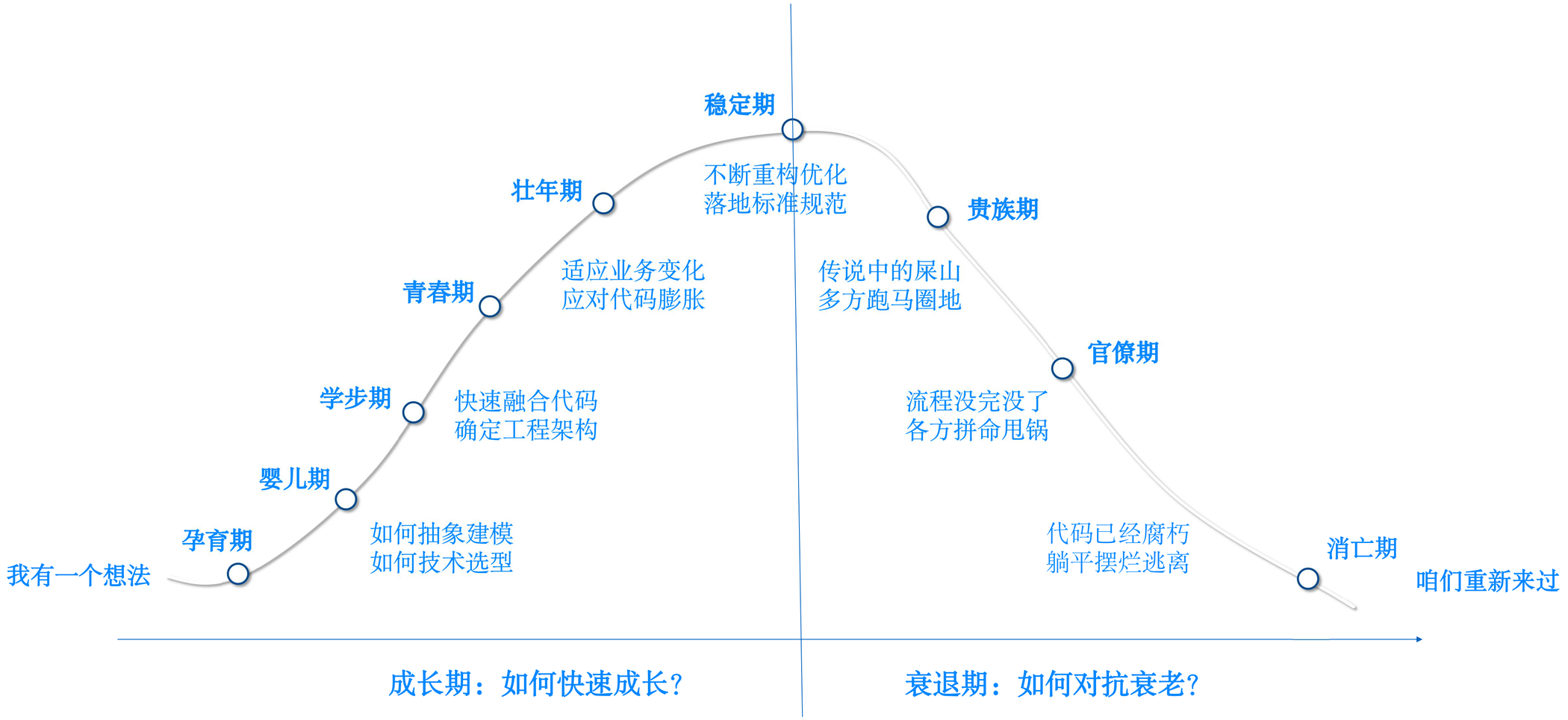
03 架构演进的例子

04 成为优秀架构师

01

架构面临的问题

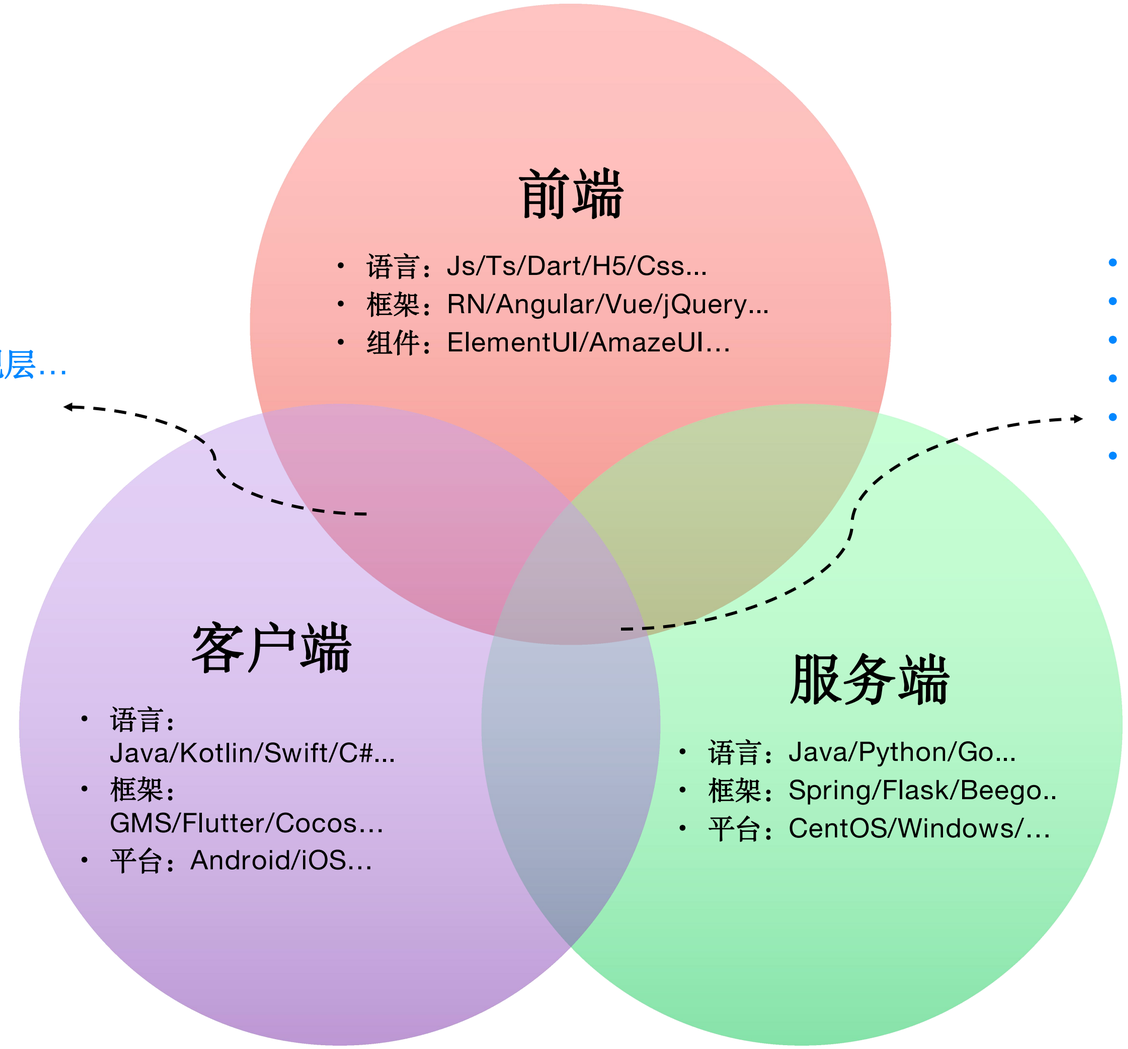
不同产品生命周期的架构问题



不同技术领域的架构问题

交叉问题

- 跨端调用：JsBridge/JNI…
- 多端一致：统一接口层/统一实现层…
- 数据通信：IDL协议/压缩/安全…



公共问题

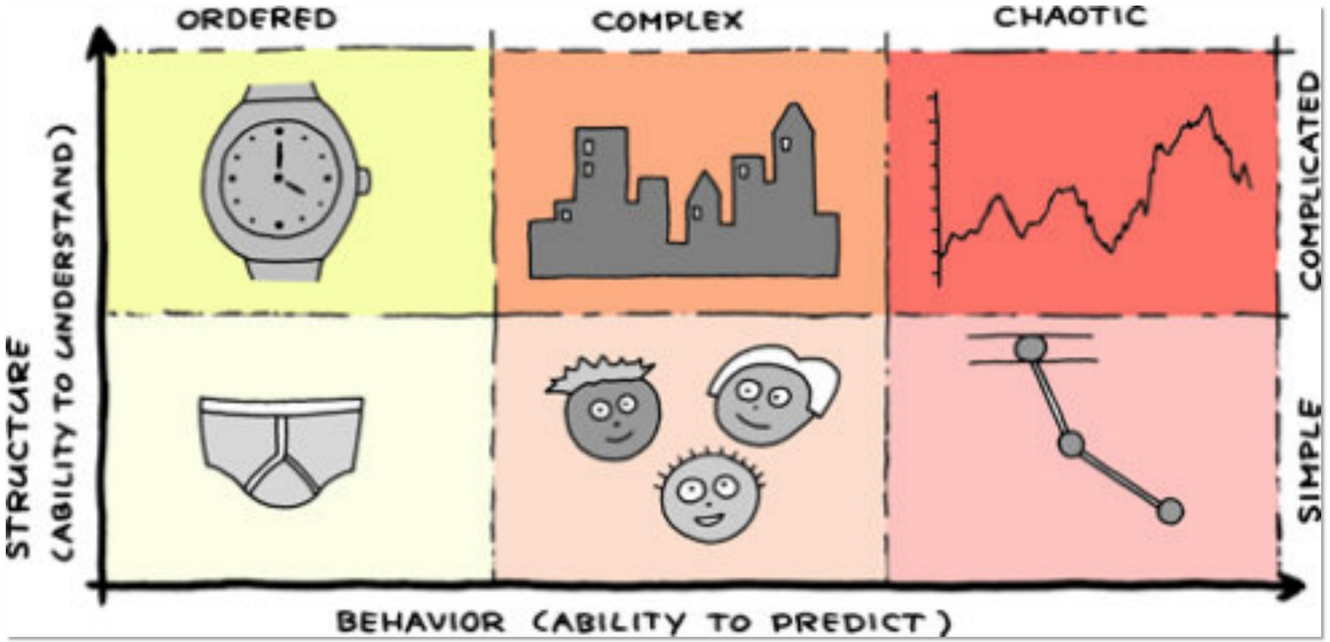
- 编程思想：OOP/AOP/IoC…
- 问题分解：按业务/按技术…
- 领域建模：接口设计/DSL/…
- 服务治理：模块化/容器化…
- 流程机制：敏捷开发/单测手段…
- 架构标准：公约文档/数据监测…

还有一些其他挑战

只要业务继续发展，越来越复杂就是必然趋势。

理解成本变高

- 宏大的规模是不好理解的
- 复杂的结构是不好理解的



预测难度变大

- 业务变化不可预测
- 技术变化不可预测

典型的架构设计 – Android OS

应用框架层

App开发者直接使用的接口层，UI的实现、数据的处理、资源的使用，都是利用这一层的API

系统服务

提供窗口管理、相机、音视频等重要的系统能力，包含各种子系统，内部逻辑十分庞大，往下调用HAL层封装的硬件能力；往上通过Binder暴露可以远程调用的API

Linux内核

CPU、内存、唤醒服务等重要的驱动实现都是基于该层操作系统的核心实现。

Binder IPC

提供跨进程访问的能力，App可以高效的访问由系统进程暴露的能力。App进程与系统进程之间的通信是典型的C/S模型。

硬件抽象层

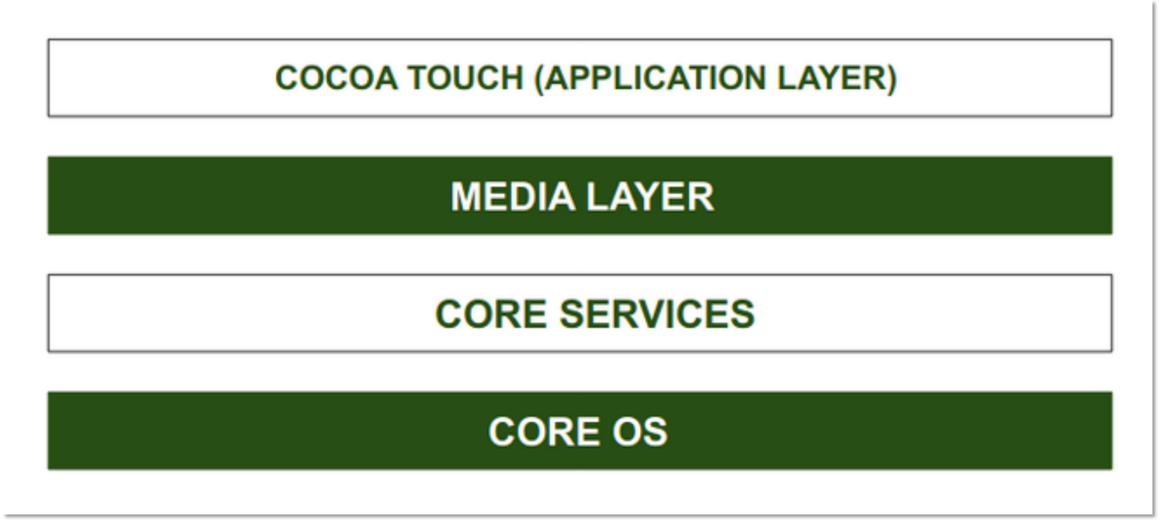
屏蔽底层不同驱动的差异，使得系统服务层可以快速适配到不同的硬件设备

典型的架构设计 – iOS

Android	iOS
Activity	ViewController
Intents	Segues/ViewControllers
Service	“Background Mode”
ContentProvider	CoreData
Layouts	Storyboards and scenes

应用框架层
EventKit、GameKit、MapKit、PushKit

核心服务层
Location、Motion、Health、GPS、Telephony、Foundation…



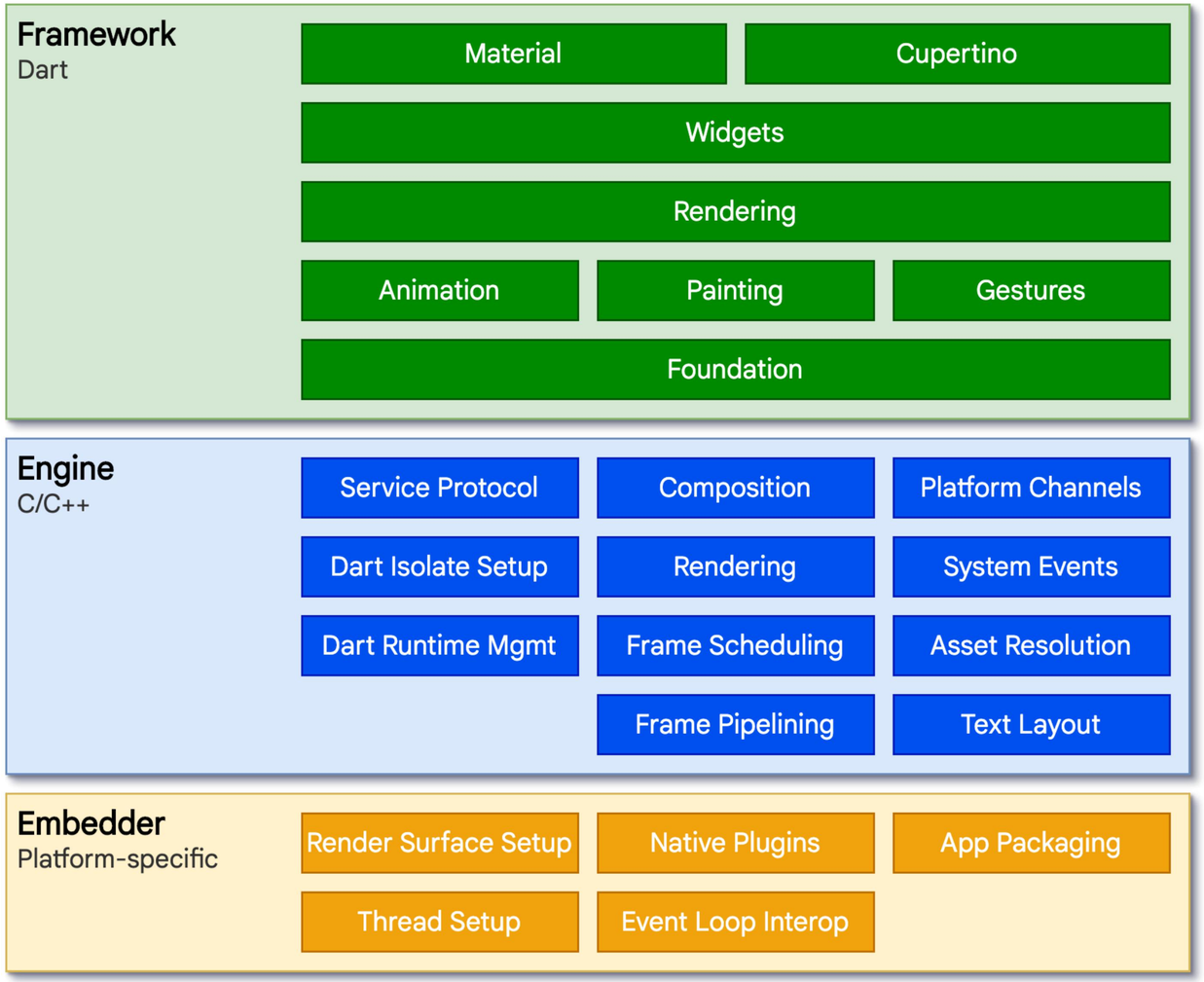
图形图像层
ULKit、Animation、Graphics、Images

内核层
Bluetooth、Security、Accessories…

典型的架构设计 - Flutter

UI框架层

提供不同样式的组件和动画，声明式UI。采用了Dart作为编程语言，能够同时支持JIT和AOT，在开发调试和运行阶段都能有不错的效率提升



引擎层

将上层定义的UI树转换成屏幕像素，提供平台调用接口和Dart虚拟机

嵌入层

Flutter引擎需嵌入不同的平台：Android/iOS/Windows/Linux等

小结

- 架构设计是为了解决特定领域不同发展阶段的业务问题
- 不同领域的架构有明显的技术差异，但也有很多相似性
- 架构不仅面临技术挑战，还要应对组织业务膨胀的熵增
- 移动端需要利用有限的设备资源设计符合小屏幕的架构

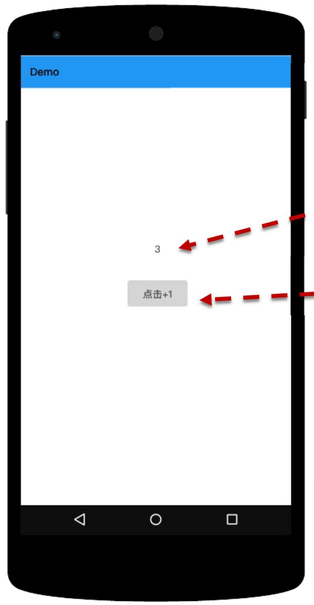
02

常见的架构手段

小的架构手段 – MVC/MVP/MVVM

架构模型	优点	缺点
MVC	• 模块职责划分明确。主要划分层M,V,C三个模块，利于代码的维护	• View和Controller容易膨胀 • View与Model没有完全分离
MVP	• View与Model完全分离，可以修改视图而不影响模型，交互都发生Presenter • Presenter与View的交互是通过接口来进行的，方便单元测试	• 页面逻辑复杂的话，相应的接口也会变多，增加维护成本
MVVM	• ViewModel与View的耦合更彻底，ViewModel只负责处理和提供数据 • View Model里面只包含数据和业务逻辑，没有UI，方便单元测试	• 数据绑定使得程序较难调试，因为数据都是自动更新到UI

小的架构手段 – MVC



调用控制

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical">

    <TextView
        android:id="@+id/tv_show"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="0"/>

    <Button
        android:id="@+id/btn_add"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="点击+1"/>
</LinearLayout>
```

```java
public class ControllerActivity extends Activity {

    private TextView mTextView;
    private Button mButton;
    private NumModel mNumModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_controller);

        mTextView = findViewById(R.id.tv_show);
        mButton = findViewById(R.id.btn_add);
        mNumModel = new NumModel();

        mButton.setOnClickListener(new View.OnClickListener() {//接收来自View的事件
            @Override
            public void onClick(View v) {
                mNumModel.add(ControllerActivity.this);//通知Model处理数据
            }
        });
    }

    public void updateUI(String text) {//更新UI
        mTextView.setText(text);
    }
}
```

```java
public class NumModel {
    private int num = 0;

    public void add(ControllerActivity activity) {
        num = ++num;//更新数据
        activity.updateUI(num + "");//更新UI
    }

}
```

小的架构手段 - MVVM

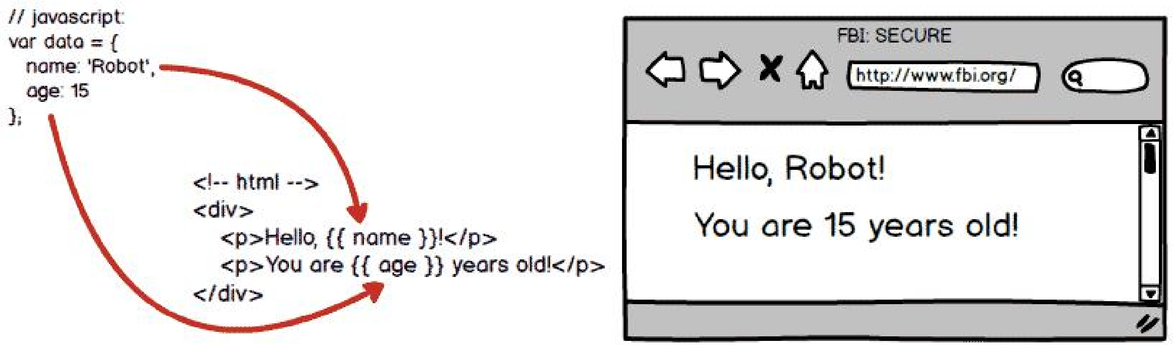
```xml
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">

    <data>
        <variable
            name="numVM"
            type="com.xx.oo.NumViewModel"/>
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{numVM.num}"/>

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="@{numVM.onClickAdd}"
            android:text="点击+1"/>
    </LinearLayout>

</layout>
```

```java
public class VmActivity extends Activity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        NumViewModel numViewModel = new NumViewModel();
        VmActivityBinding binding = DataBindingUtil.setContentView(this, R.layout.vm_
        binding.setNumVM(numViewModel);//View与ViewModel绑定
    }
}
```

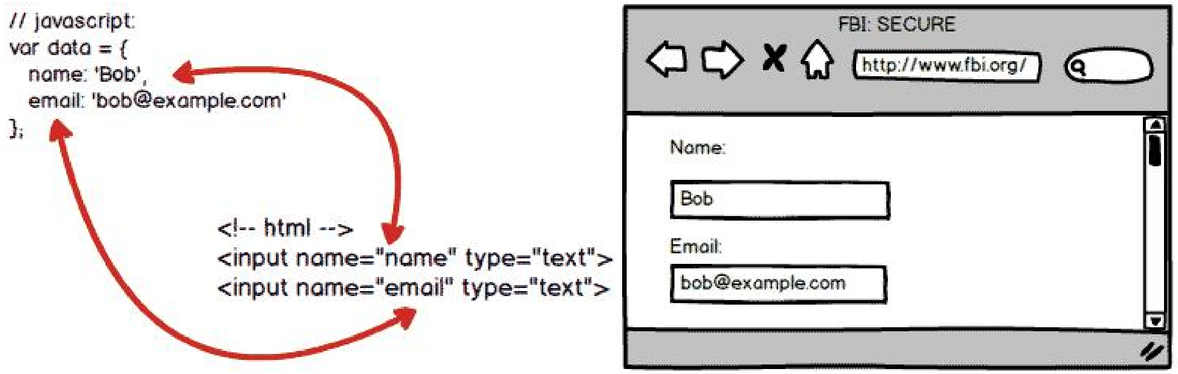
```java
public class NumViewModel extends BaseObservable {
    private String num;

    private final NumModel mNumModel;

    public NumViewModel() {
        mNumModel = new NumModel();
    }

    @Bindable
    public String getNum() {
        return num;
    }

    public void setNum(String num) {
        this.num = num;
        notifyPropertyChanged(BR.num);//更新UI
    }

    public void onClickAdd(View view) {//点击事件处理
        mNumModel.add(new NumModel.ModelCallback() {//相关逻辑处理，这里直接交给Model层
            @Override
            public void onSuccess(int num) {
                setNum(num + "");//成功，更新数据
            }

            @Override
            public void onFailed(String text) {
                setNum(text);//失败，更新数据
            }
        });
    }
}
```

```java
public class NumModel {
    private int num = 0;

    public void add(ModelCallback callback) {
        callback.onSuccess(++num);//通知Presenter结果
    }

    public interface ModelCallback {//数据回调接口
        void onSuccess(int num);

        void onFailed(String text);
    }

}
```

小的架构手段 - Data Binding



单向绑定：数据 → 视图

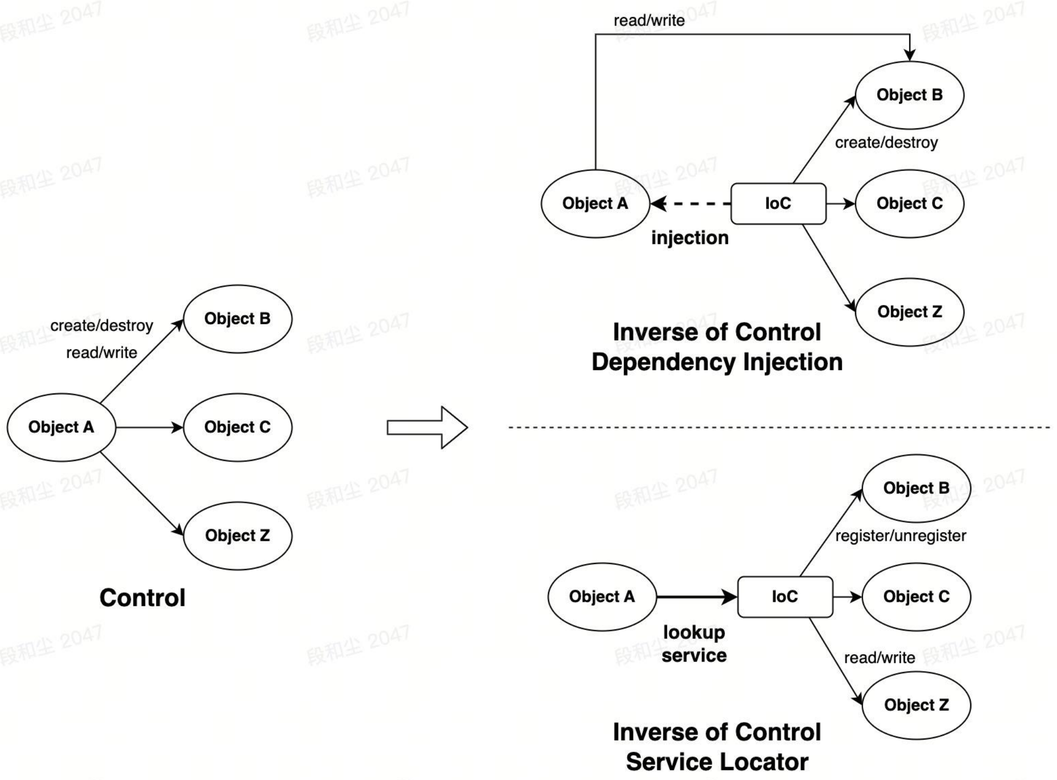


双向绑定：数据 ←→ 视图

Data Binding	单向自动更新	双向自动更新	Android使用特征	典型场景
Static			@{}	界面模板，不同区域填充不同数据
Observable			Observable，@BindAdapter	不同状态显示不同界面，界面局部更新
Two-way			@={}，@InverseBindingAdapter	输入框，选择框

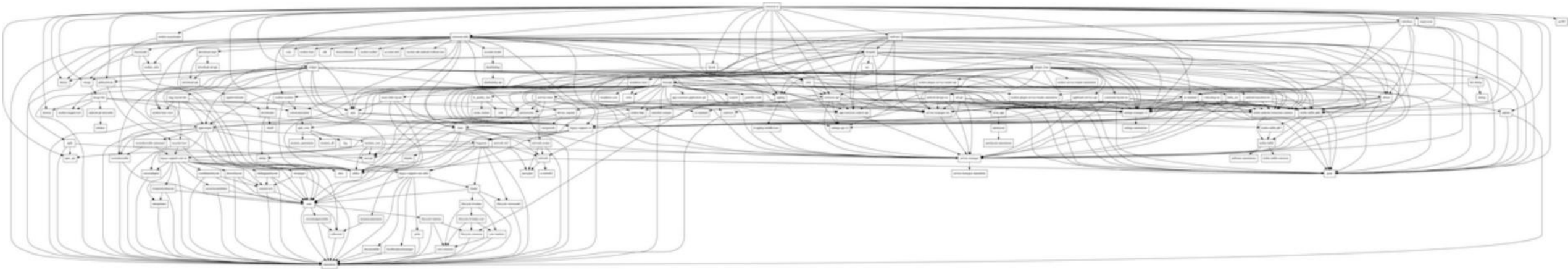
小的架构手段 – IoC

控制反转，是一种**模块化**的编程思想，有不同的实现方式。

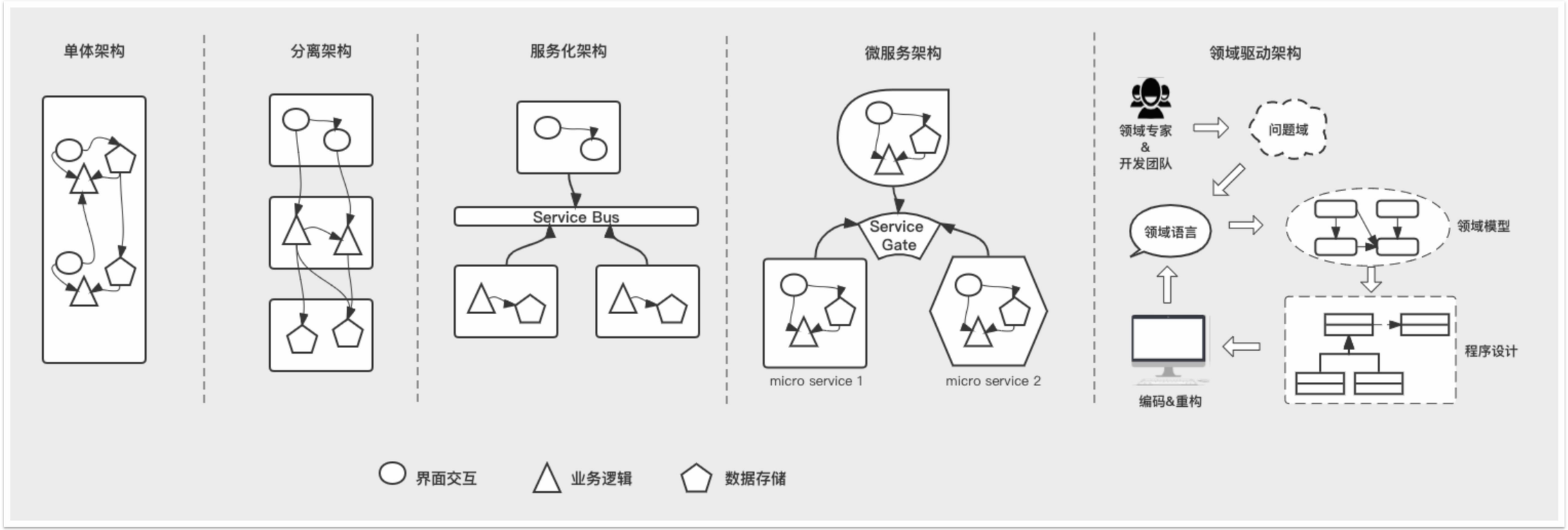


- 依赖注入(Denpendency Injection)
  - Constructor Injection(eg: Spring @Autowired，在构造的时候，将被依赖的对象注入)
  - Setter Injection(eg: Spring 通过setter方法将依赖的对象注入)
  - Parameter Injection(eg: Retorfit @Header，@Query，@Path)
  - Interface/Method injection(eg: Callback)
- 依赖查询(Dependency Lookup)
  - Service Locator(eg: Service Provide Interface)
  - Contextualized Lookup(eg: Android Context)
  - Template Method(eg: ViewDataBinding...)
  - Strategy(eg: Scheduler, Event-Driven)

小的架构手段 – IoC滥用的案例



大的架构手段


单体架构
分离架构
服务化架构
Service Bus
微服务架构
Service Gate
micro service 1
micro service 2
领域驱动架构
领域专家
&
开发团队
问题域
领域语言
领域模型
程序设计
编码&重构
界面交互
业务逻辑
数据存储

小结

- 不同架构手段的共同目标是高内聚低耦合
- 找到适合业务场景的架构而**不是炫技滥用**
- 一个复杂的系统是多种架构模型的组合体

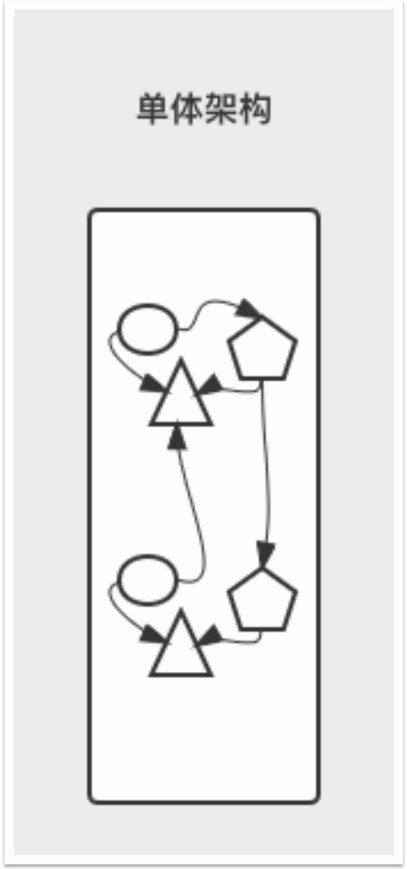
03

架构演进的例子

孕育期

做一个信息流产品，可以无限刷的列表

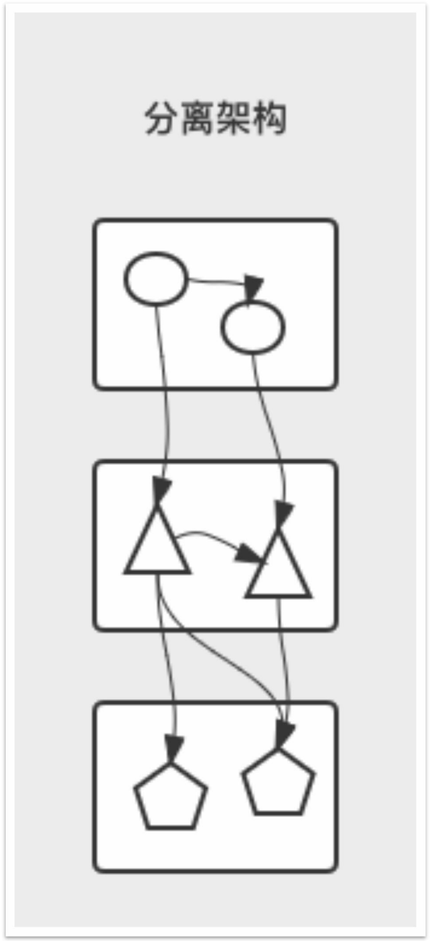
- 首先，需要实现一个列表，支持上下滑动
- 然后，每次滑动，都需要请求服务端数据
- 接着，列表的每一项都需要响应点击操作
- …



婴儿期

产品功能开始变多，需要拆分模块

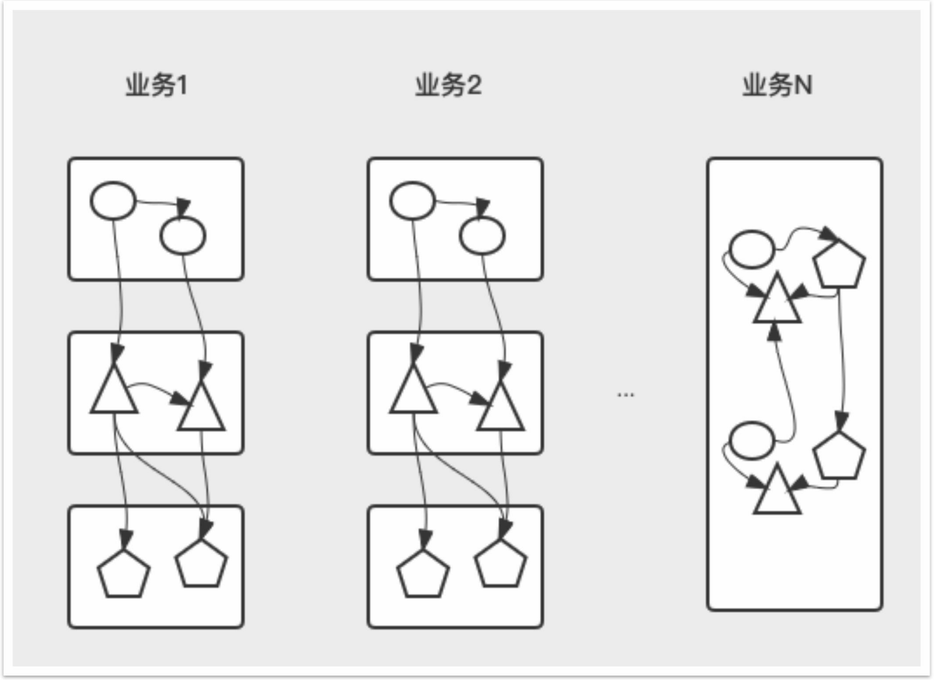
- 首先，需要支持图文内容的组合混排显示
- 接着，需要引入账号体系，用户注册登录
- 然后，用户可以收藏感兴趣的内容并分享
- …
- 继续，场景越来越多，拆分网络和多线程
- …



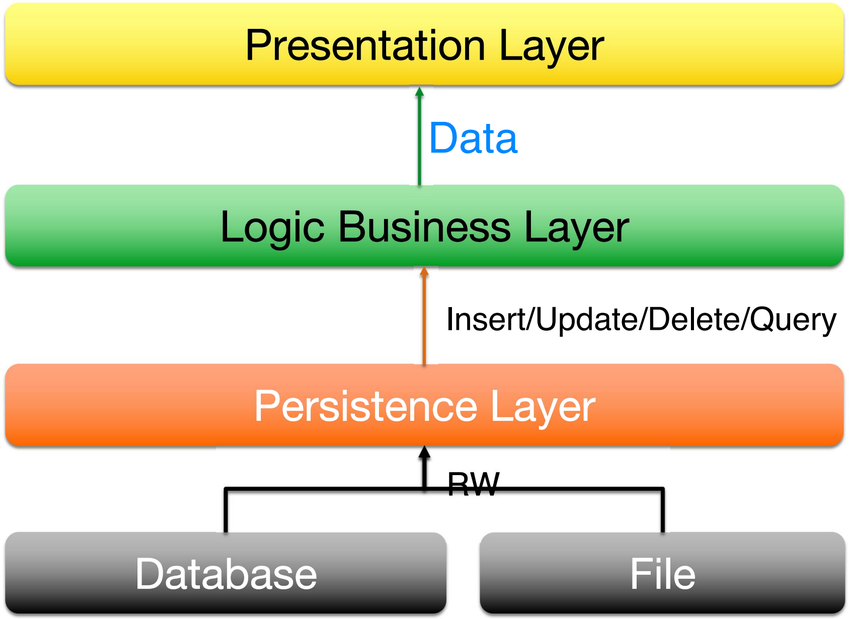
学步期

业务场景变多，需要拆分业务

- 首先，支持用户发布文章，并给予奖励
- 接着，视频这个重要的内容也需要支持
- 然后，不同业务之间越做越大需要拆分
- …
- 继续，拆分出视频业务，架构自成体系
- …
- 继续，拆分出中台业务，供多业务使用
- …



学步期 – 分层架构



表现层：接收用户输入，获取数据，呈现界面

业务层：处理业务数据，数据流转，安全检查

持久层：提供数据的增删改查能力

存储层：按照特定的格式存储数据

优点

- 结构简单清晰，易于理解和管控
- 层级关系适合不同技能人员分工

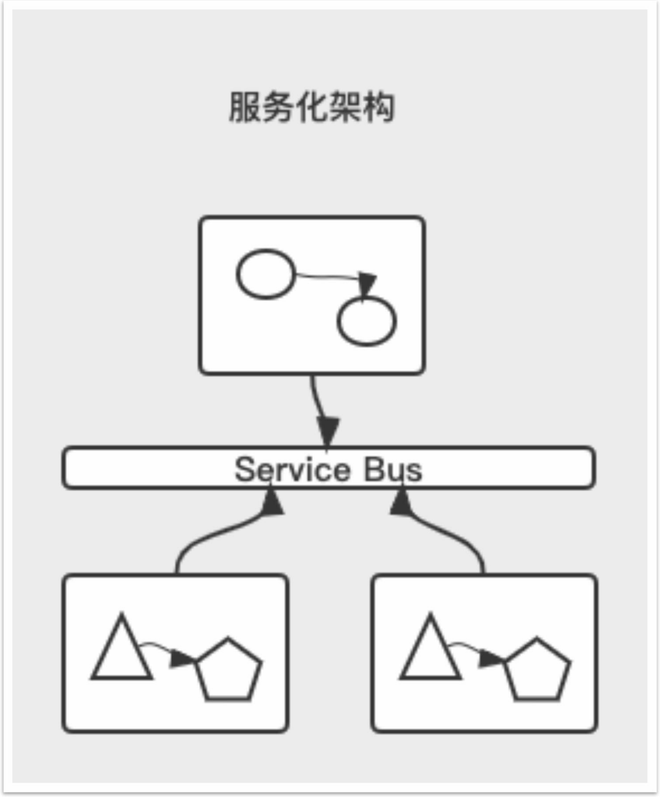
缺点

- 对层级管控要求严格，灵活性低
- 为了解耦容易拆分出很多中间层

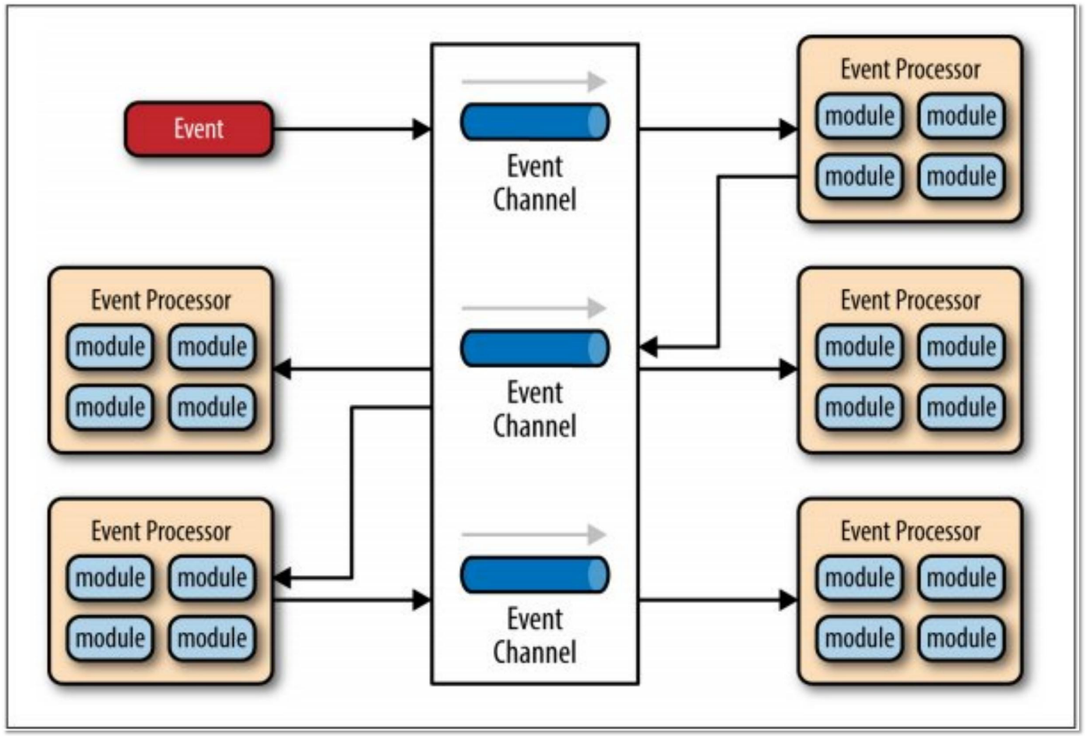
青春期

不同业务和模块混合，需要解耦

- 首先，需要约定模块可以对外提供的能力
- 接着，模块之间需要遵循相同的调用方式
- 然后，旧的模块需要按照相同标准来改造
- …
- 继续，使用方不应该直接依赖于实现方
- …



青春期 – 事件驱动架构



事件Event：一个消息，譬如触摸了屏幕、点击了按钮

事件处理器Processor：事件的实际消费者，对关注的消息进行订阅，消息处理的过程很灵活，可以在当前处理器“吞”掉一个消息，也可以继续将消息交由其他消息队列处理(责任链)

事件队列Event Channel：消息队列，事件将以消息的形式发布到队列中，并设计特定的派发机制来处理队列中的消息

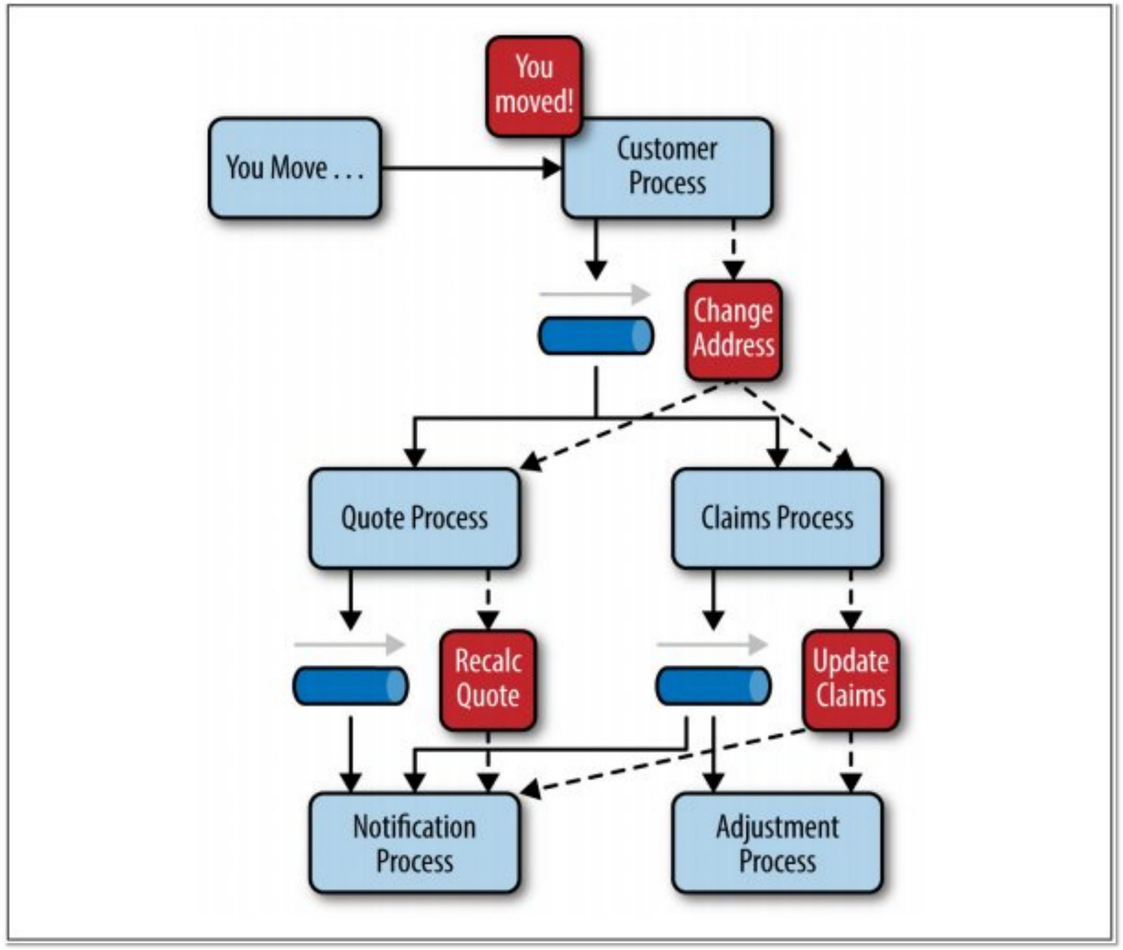
优点

- 适用广泛，容易扩展出不同变种
- 性能较好，支持消息的异步处理

缺点

- 缺少约束，消息处理器容器膨胀
- 处理链路容易变长，理解成本高

青春期 – 事件驱动架构实例



实例

1. **事件发布：**「You Moved」事件被处理器「Customer」接收处理

2. **事件处理：**「Customer」将事件转换为「Change Address」，并发布到消息队列

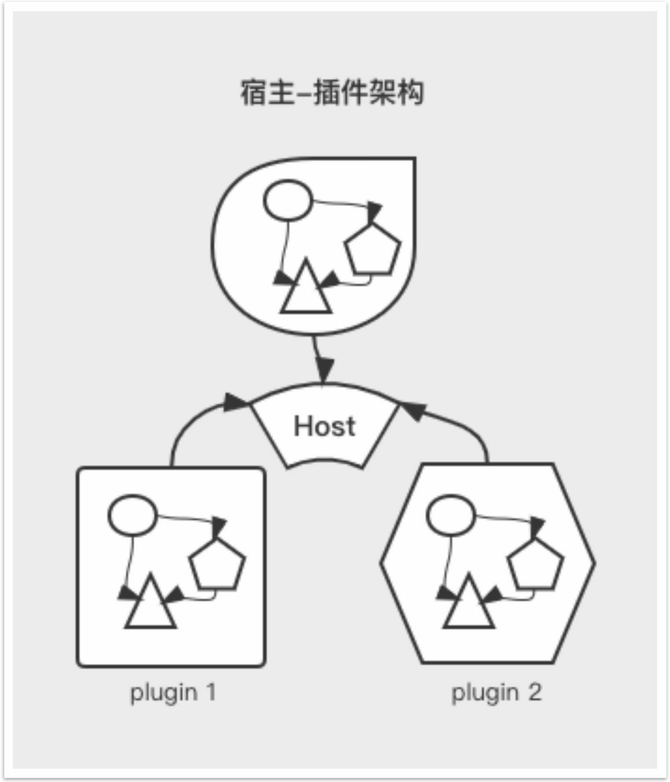
3. **事件处理：**「Quote」和「Claims」两个事件处理器都订阅了「Change Address」消息，并收到了从消息队列派发出的消息

4. **事件处理：**「Quote」将消息转换为「Recalc Quote」并将其派发给「Notification」这个处理器；「Claims」将消息转换为「Update Claims」，并将其派发给「Notification」和「Adjustment」

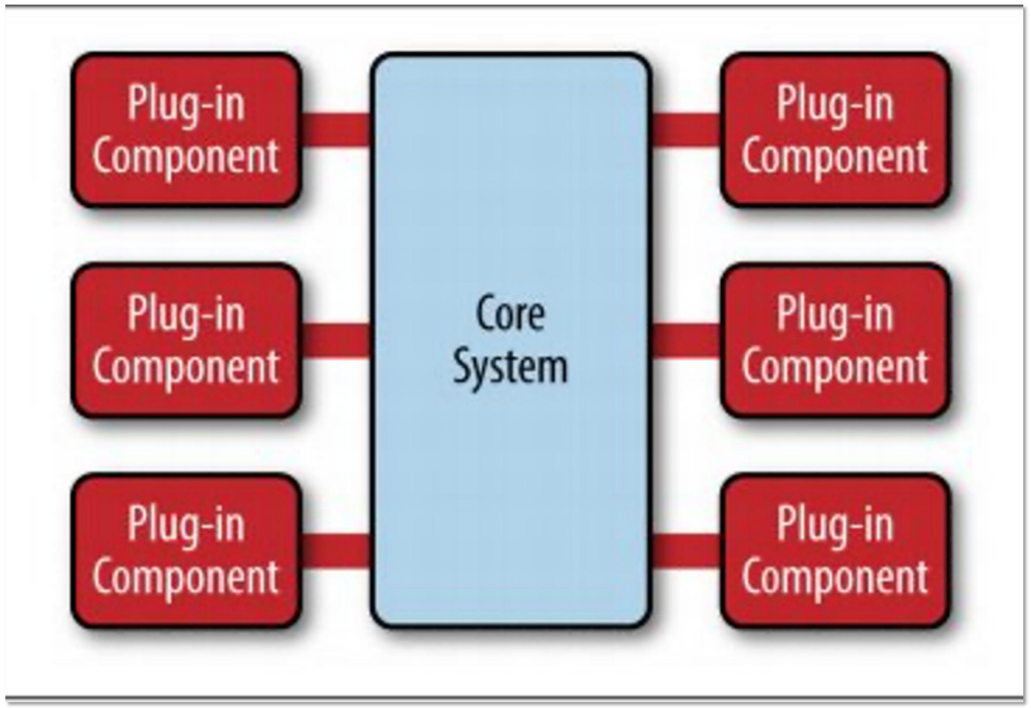
壮年期

业务变得更加内聚，需要灵活插拔

- 首先，扫一扫等能力不是所有场景都需要
- 接着，视频的子能力可以拆解后按需使用
- 然后，越来越多的业务想动态化发布产物
- …
- 继续，动态化发布引入很多问题需要调优
- …



壮年期 – 微内核架构



宿主容器：Core System，一般不包含具体业务逻辑，而是从中抽象出模型，相当于一个运行环境，供不同业务以插件的形式在其中运行

插件：Plug-in Component，具体的业务实现，通过一定的技术手段挂载到宿主容器中，插件一般可以访问宿主的资源

优点

- 高扩展性，需要什么就开发什么
- 插件隔离，能够简化业务耦合度

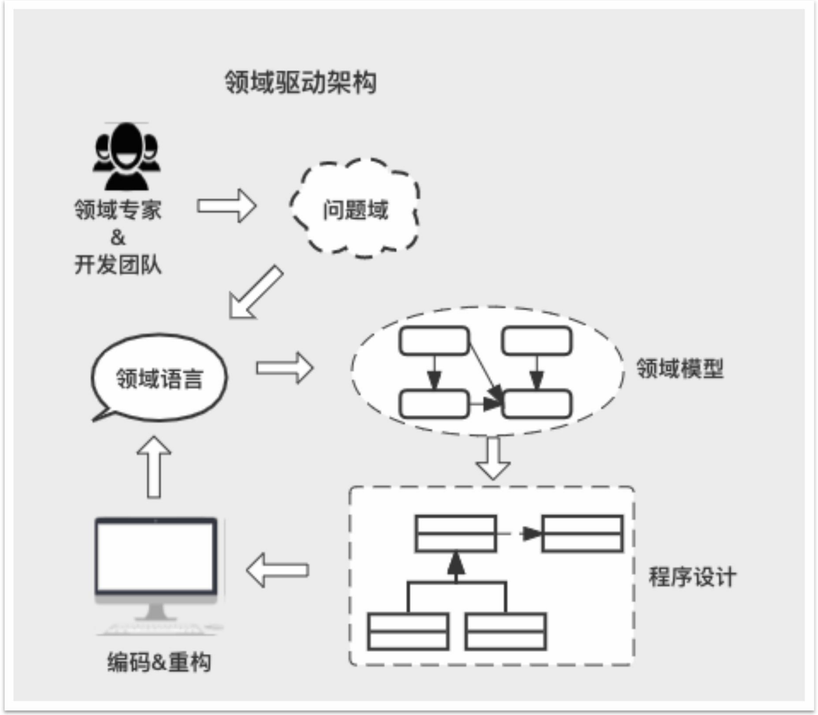
缺点

- 对宿主容器的要求很高，且宿主不易扩展
- 插件的注册和通信机制通常比较复杂

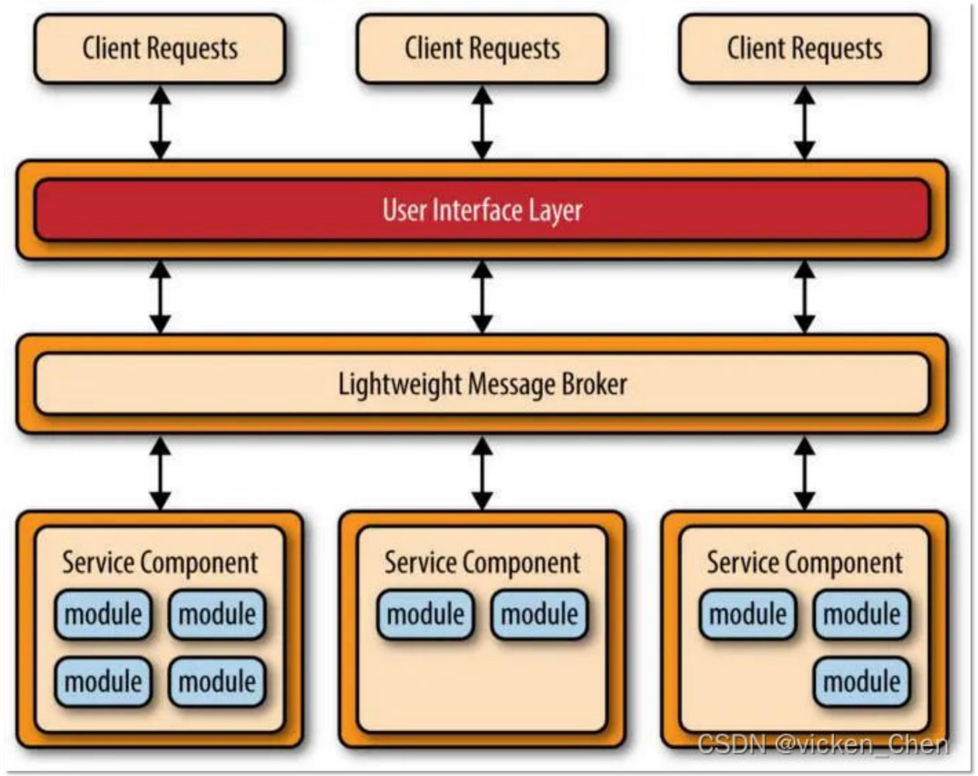
稳定期

用户规模和团队稳定，历史包袱重

- 首先，经过前期快速发展已经积累了大量历史包袱
- 接着，需要深入了解业务才能设计出更合理的架构
- 然后，很多改动牵一发动全局，代码被迫变得更差
- …
- 继续，新旧技术栈共存，版本依赖冲突，冗余代码
- …



稳定期 – 微服务架构



请求接入：服务使用方发起请求，请求以一定的方式(可以直接调用，也可以跨进程调用)发送到服务注册中心，等待请求的处理

服务调度：Broker，将服务请求调度到对应的微服务节点上进行处理

微服务：Service Component，一个高度内聚的模块集合，对外暴露服务接口。每一个微服务都是独立的，分别向服务注册中心注册自身所能提供的服务接口

优点

- 健壮性好，单个服务不影响全局
- 服务隔离，服务之间互不相耦合

缺点

- 容器出现服务数量腹胀难以管控
- 服务发现和通信需要额外的成本

贵族期

期望高

- 良好的顶层设计，从上到下有统一的认知
- 遵循共同的规范，写出让人舒适的代码
- 有没有“一劳永逸”的设计，可保基业长青
- 什么时候能从架构工作中找到成就感

事情难

- 业务历久弥新，历史包袱叠加新的场景
- 随便动动刀子就拔出萝卜带出泥
- 技术栈层出不穷，老朽学不动了啊
- 一方面要保持成熟稳定，一方面要积极探索落地

责任大

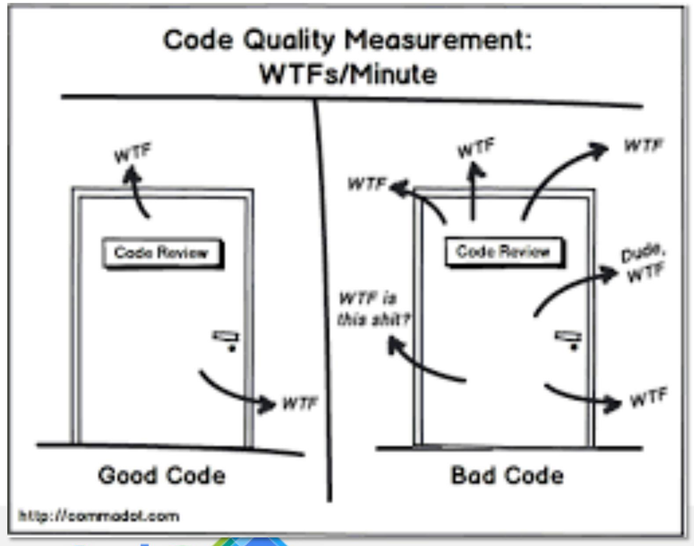
- 设计不合理：这谁写的，看小爷我推到重来
- 使用不规范：这压根就不该这么用
- 逻辑太晦涩：这尼玛谁看得懂
- 编码坑太多：这特么是隐藏技能啊，悄无声息改代码

疗效慢

- 影响复杂度的因子众多，单个因子优化不足以撼动
- 没有一年半载，一波治理搞不下来
- 没有特效药，长期在隐隐作痛中前行
- 往往也只是开了一个“方子”，想要根治得长期健身

贵族期

	产品雏形期	产品发展期	产品瓶颈期
复用	这个功能可以从XX搬运一下	这两个功能类似，合并一下吧	这个业务要打磨一下用到其他业务
重构	这么实现会更优美一点	兄台，建议你看一下Clean Code	推到重来吧
规范	啥？	这要之前按照规范来就好了	打嘴炮的吧，根本消费不了
减负	毫无负担，就是飞	这一坨代码要不先留着	哪哪都是坑



对抗架构衰老，从我做起

“这个组件不迭代了，不用改了吧” → “这个组件的API不合理，其他App很难用，得改”
“这一坨陈年无用代码反正也不影响功能” → “这对其他App就是累赘，必须删掉”
“这个逻辑有点多，改起来容易错，还是新搞一个吧” → “通用逻辑，还是好好重构统一吧”
“这个着急合码，先这样吧，后面再改” → “不行啊，此时不改，祸害好几方”
“小问题来着，就头条有，先观察一下” → “不仅仅影响到局部了，得谨慎分析一下”

官僚期

人们相互甩锅

代码混乱无序

不同形态的架构

架构演进	核心思想	优点	缺点
单体架构	快速组织界面交互、业务逻辑和数据之间的关系	•快速成型 •管理简单	•依赖复杂 •单点失效
分离架构	分而治之。拆分方式多样，譬如：按业务拆分(视频、图文、搜索等)；或者按技术拆分(MVC、MVP、MVVM)	•业务分层更清晰 •组件复用	•强制遵循规范 •业务依旧耦合
服务化架构	设立业务之间的接口契约	•业务解耦更内聚	•服务数量膨胀
微内核架构	配套插件注册和发现机制，业务动态发布	•服务隔离可插拔 •局部失效影响可控	•管理成本高 •调用链路长
领域驱动设计(DDD)	战略设计从业务视角出发，建立业务领域模型 战术设计从技术视角出发，侧重于技术实现	•解决复杂业务问题 •架构随着业务发展	•需成熟的团队 •经验高度抽象

小结

- 架构随业务发展由简单变得复杂是规律
- 没必要最初用复杂架构来解决简单问题
- 需要用规范持续重构来对抗代码的腐朽

04

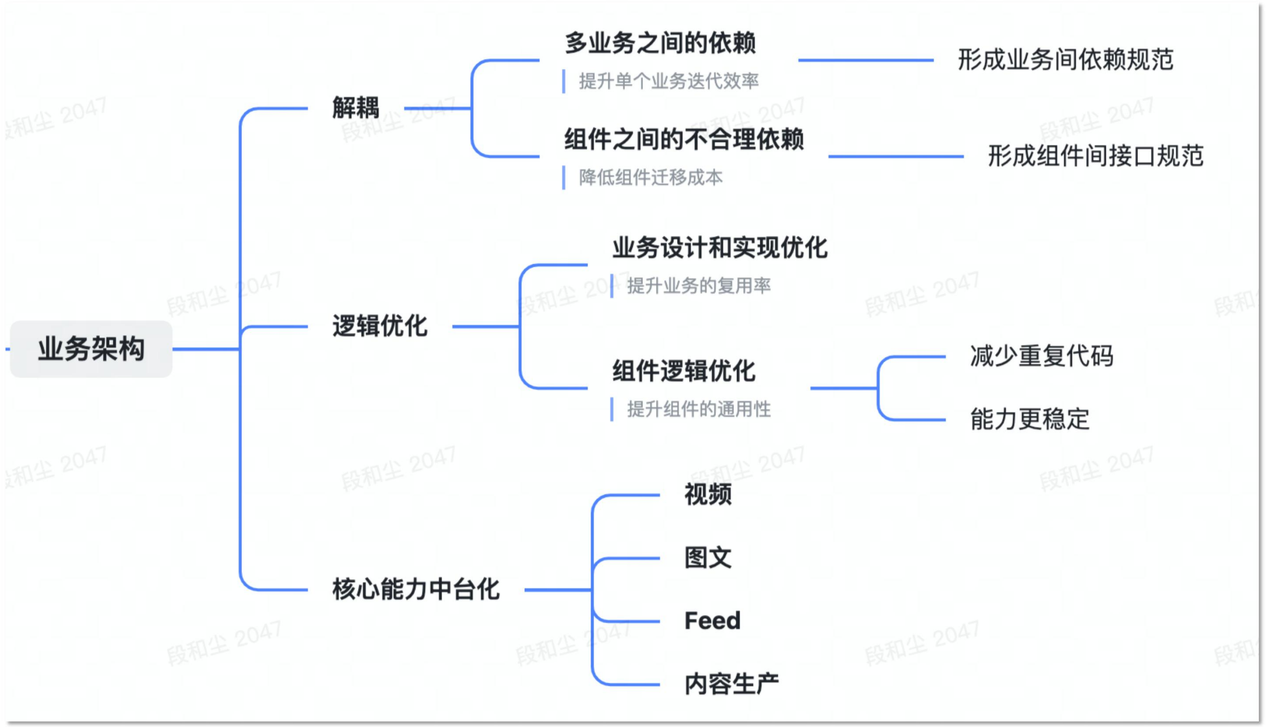
成为优秀架构师

越是前面越难

定义问题 → 确定架构 → 方案落地 → 结果复盘

认清问题 – 分类

架构的问题是盘根错节的，将所有问题放在一起，就有轻重缓急之分，就有类别之分

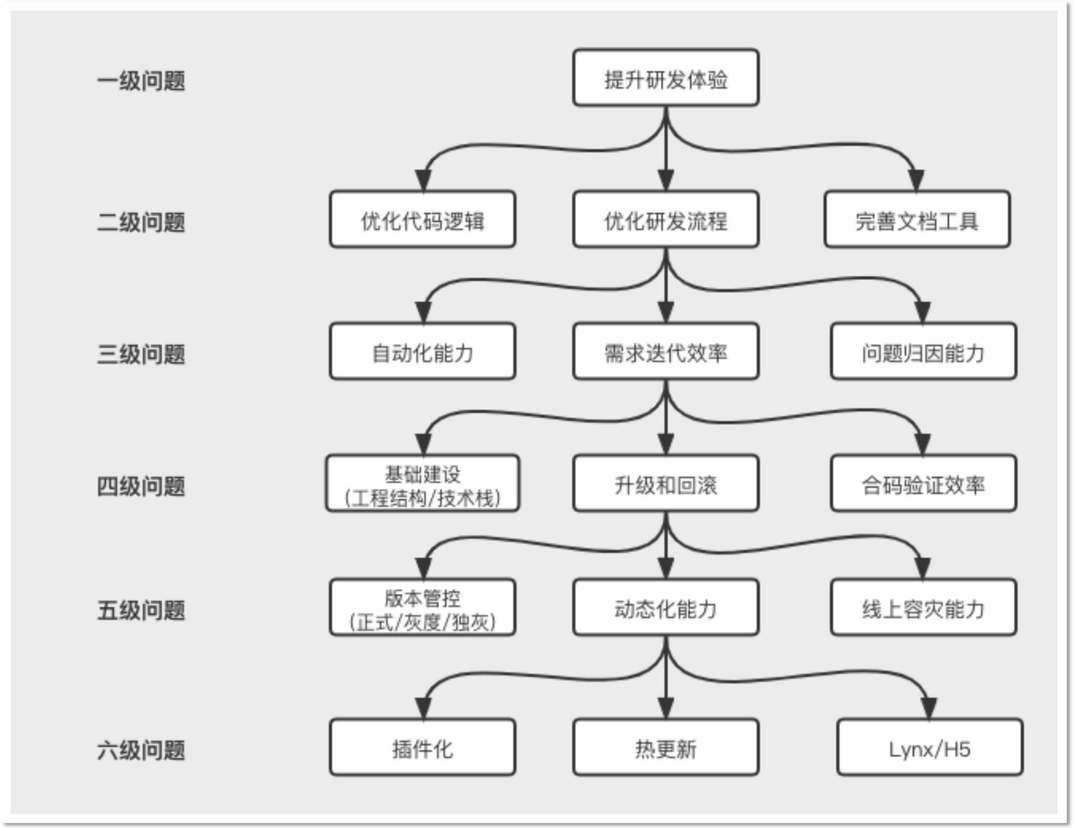


认清问题 – 分类

区分问题的类别，就能在一定的边界内，匹配上
对应的人来解决问题

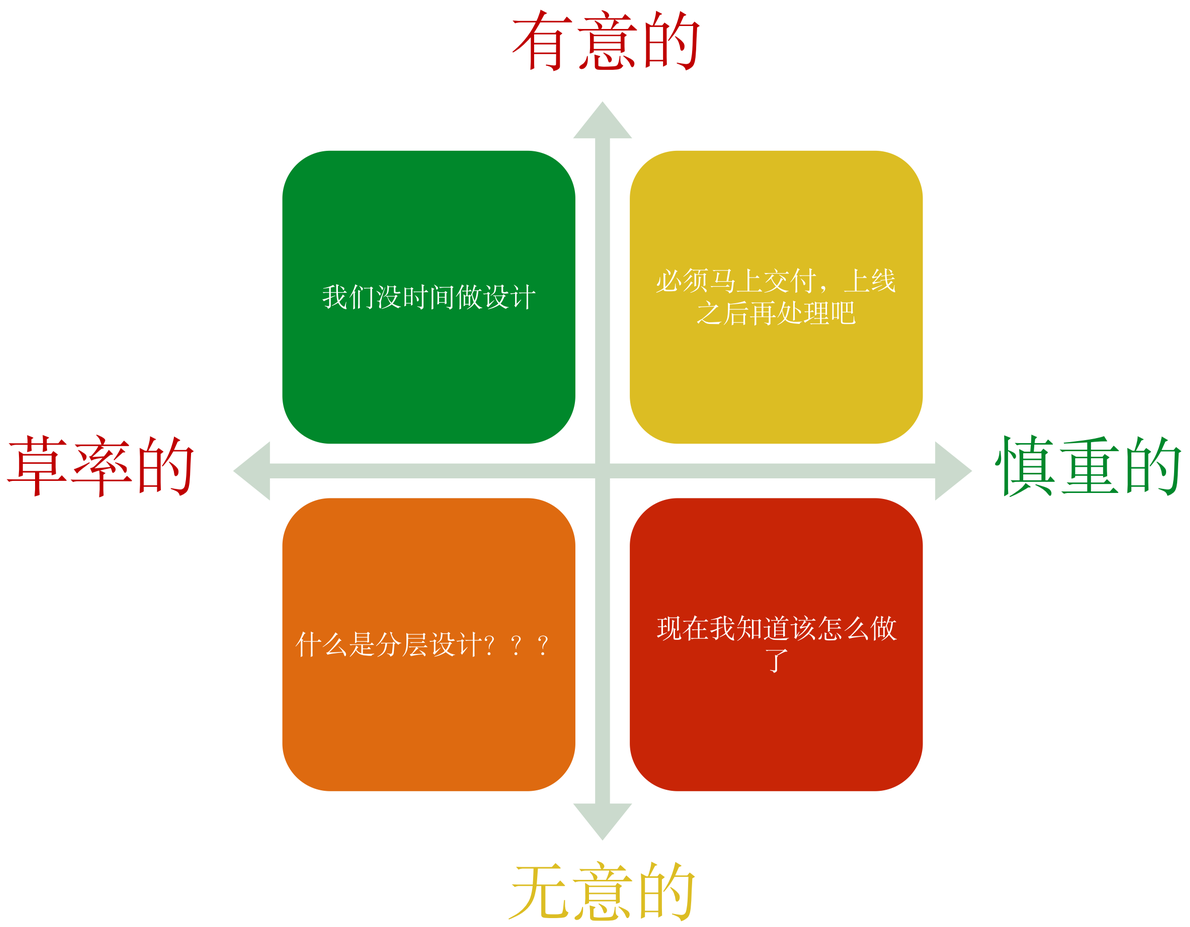
认清问题 – 分级

挑战、问题、手段这些经常混为一谈，哪些是挑战？哪些是问题？那些是手段？
其实这些都是一回事，就是矛盾，只是不同场景下，矛盾所在的层级不同



理性看待 - 技术债的产生和应对

Technical Debt，本应采用最佳方案，但妥协了，从而给未来带来了负担。



公开技术债

一开始就与利益相关方权衡技术债的利弊，明确影响和解决方案
可以将所有的技术债量化公示，短期和长期技术债可以区分对待
大家遵循相同的编码规范

消费技术债

每次迭代确定一定数量的技术债作为消费目标
无需偿还技术债的产品是即将消亡、一次性的原型或者短命的产品
避免使用过时的技术，旧资产和老方法充斥着安全漏洞，难以集成

勤于编码

	目标	描述
技术基建 工具流程易使用 品质优化可复制 质量基建强约束 整体架构高可用	**平台接入工具便捷**	持续集成流水线、埋点上报和分析平台，实验平台等
	本地工具随取随用	各类IDE插件、编译工具链等可直接用于新产品
	通用场景直接复用	诸如磁盘、内存、线程调度优化等，可直接复制迁移
	特殊场景沉淀基建	诸如启动、预加载、分片处理等，需沉淀框架能力，形成通用组件
	线下监测分析准入	代码静态检查和运行时监控、自动化测试、内测众测等通用能力和流程
	线上监控止损归因	各类指标监控报警手段、反馈舆情收集、熔断和回滚机制
	工程结构简洁清晰	架构模型、演进思路、工程要素的组织关系需要做到清晰明确
	技术栈保持先进性	工具、组件、Android版本，跨端、热修等动态化能力需持续更新
多个业务 业务之间低耦合 扇入扇出标准化 业务集合可维护	**服务注册发现机制**	利用中心服务管理模块进行解耦
	公共模块下沉机制	利用具备公共代码的模块进行解耦
	接口使用规范一致	多业务对外暴露的接口遵循相同的命名、参数、语义规范
	业务外部依赖收敛	最小化业务对外部的依赖，同步多业务相同依赖的版本
	业务复用的清单制	业务组合的场景很多(比如大业务拆开分层组合)，需整体维护组合清单
	业务集合的准入制	大而全，但整体可复用程度低的业务集合，不如少而精
单个业务 具备好的扩展性 支持傻瓜式集成 宿主依赖接口化	**外围能力可插拔**	业务内部具备事件路由能力，能够将事件调度到外围
	扩展定制有范式	差异化改造可以参照已有的扩展方式，不需要魔改
	样例文档完备	可以用人工支持频次的降低来衡量样例文档是否健全
	侵入式改造少	减少集成的改造点和代码量
	数据链路闭环	从外部获取数据后，尽量业务内部闭环处理，交付结果给外部
	默认实现兜底	外部依赖尽可能有兜底实现，可让业务独立可调试

想一想，我该如何把这些技术应用在工作实践中？

THANKS