<https://zhuanlan.zhihu.com/p/152220505>

搭建可拓展性（scalable）的软件是一项具有挑战性的任务。当我们在思考前端应用的可拓展性时，我们会想到递增的复杂度，越来越多的业务规则，应用需要加载越来越多的数据以及遍布全球的分布式团队。为了应对上述的各种因素从而保证高质量的交付和避免技术债的产生，健壮及牢固的架构必不可少。虽然 Angular 自身是非常具有技术倾向性的框架，迫使开发者以*恰当*的方式进行开发，但是开发过程中依然容易犯错。在这篇文章里，我会展示极力推荐的基于最佳实践和经过实战检验的具有良好设计的 Angular 应用架构。我们在本文的终极目标是学习如何设计一个长期内的**可持续的开发效率**和**增加新功能的简易程度**的 Angular 应用。为了达成这些目标我们会使用到

1 - 应用层次之间恰当的抽象

2 - 单向数据流

3 - 响应式（reactive）状态管理

4 - 模块设计

5 - 容器（smart）组件模式和展现（dumb）组件模式

译者注:

我听过一种说法说 React 是类库，Angular 是框架，类库和框架区别在于类库是被你所写的代码调用，而框架是调用你所写的代码。然而这个对 React 真的成立吗？React 不能自成框架吗？一定要搭配的 Redux 使用？

smart component 和 dumb component 在 Angular 中也是成立的

***目录***

前端的可拓展性问题

软件架构

高层次抽象层

展现层

抽象层

核心层

单向数据流和响应式状态管理

模块设计

模块目录结构

容器组件模式和展现组件模式

总结

**前端的可扩展性问题**

让我们思考一下在开发现代前端应用中面临扩展性问题。当下前端应用不再“仅仅展现”数据和接受用户的输入。单页面应用（Single Page Applications）为用户提供了丰富交互并使用后端作为数据持久层。这也意味着更多的职责被转移到了软件系统的前端。这导致了我们需要处理的前端逻辑变得越发复杂。不只需求一直在增长，连应用需要加载的数据量也在增加。在这些现实之上，我们还需要考虑到脆弱的的性能问题。最后因为我们的开发团队在增长（或者至少在轮替，有人来有人走） ，让新加入的开发者尽可能快的融入也变得非常重要

解决上述问题的方案之一就是坚固的系统架构。但是这需要代价，代价是从第一天起就要拥抱架构。当系统非常小时，快速交付功能对我们开发者来说非常有快感。在这个阶段，一起都容易理解，所以开发速度非常快。但是除非我们关心架构，否则经过几轮程序员的轮替，开发完奇怪的功能，重构，引入一些新模块之后，开发速度会断崖式下跌。下面的图表展示了在我的开发生涯中通常遇到的情况。这不是任何的科学研究，只代表我对它的印象

## **软件架构**

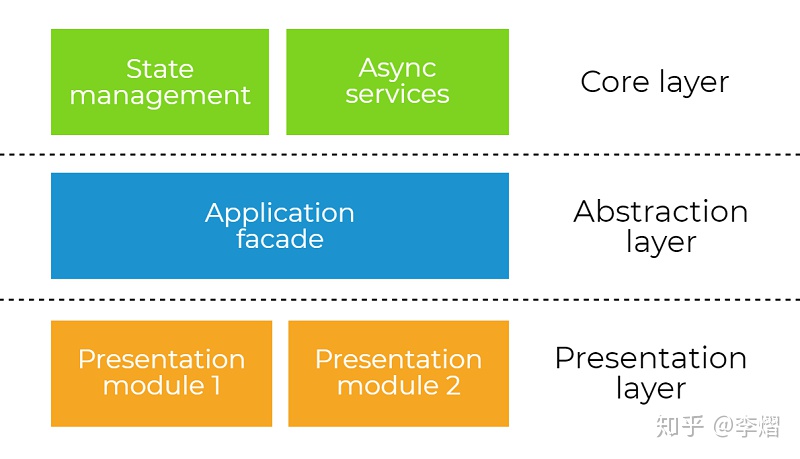
为了讨论架构的最佳实践和模式，我们首先需要回答一个问题，软件架构是什么。[Martin Fowler](https://link.zhihu.com/?target=https://martinfowler.com/" \t "https://zhuanlan.zhihu.com/p/_blank)把架构定义为：“最高层次的将系统拆解为不同的部分（highest-level breakdown of a system into its parts）” 。基于此我会将软件架构描述为关于软件如何将它部分组合在一起并且相互间通信的规则和约束。通常来说，我们在系统开发中的架构决策在系统演进的过程中很难发生更改。这也是为什么非常重要的是从项目一开始就要对这些决策多加留意，尤其当是我们搭建的软件需要在生产环境中运行多年的话。[Robert C. Martin](https://link.zhihu.com/?target=https://en.wikipedia.org/wiki/Robert_C._Martin" \t "https://zhuanlan.zhihu.com/p/_blank) 曾经说过：软件真正的开销在于维护。拥有牢固地基的架构能够帮助减少系统维护的成本

***软件架构是关于如何组织它部分的方式以及相互之间通信的规则和约束***

*译者注：在上面的翻译中我将 parts 翻译成了“部分”。你或许会认为或者翻译为“组件”听上去更合适，但“组件（components）”在不同的编程语言中有更特定的指向。在 React 和 Angular 是我们最熟悉的那个概念，但在 Java 里可以是一个 jar 包。通常来说它是比类大但是又比应用小的一个单位。思考题是，如果技术上允许在 React 应用内存在这么一个单位存在，我们应该按照什么规则组织它？我认为打包时产出的 chunk 或者 bundle 不算，它们是打包优化的产物，而并非是你思考后刻意产生的结果。Martin 的曾经提出的几个问题能够引导你思考：*

## **高层次抽象层**

首先我们将通过抽象层来分解系统。下图描述了这种分解中使用的常见概念。根本原理是将适当的职责放入适当的层中：核心层（Core），抽象层（abstraction）或者是表现（presentation）层。我们独立的观察每一个层并且分析它的职责。这样的系统分解也明确了通信规则。比如表现层只能够通过抽象层与核心层进行交谈。之后我们学习这种约束带来的优势



### **表现层**

让我们从表现层开始拆解分析我们的系统。这里是所有 Angular 组件存在的地方。这一层的唯一职责是呈现（present）和委托（delegate）。换句话说，它展示界面并且把用户的操作通过抽象层委托给核心层。它知道展示什么（what）和做什么（what），但是它不知道用户的交互应该如何（how）被处理

*译者注：这一部分的开头其实给出了一个结论或者说的假设，已经把表现层的职责限定死了。如果我们把它的一部分职责转移给其它的模块会怎么样？可以转移吗？很多时候假设“没有它会怎么样”的思考问题的方式能帮助你更好的理解某些工作原理*

下方代码片段展现了CategoriesComponent组件是如何将用户的交互行为委托给了来自抽象层的SetttingsFacade实例（通过addCategory()和updateCategory()）的，并且将一些状态呈现在模板中（通过isUpdating$）

@Component({

selector: 'categories',

templateUrl: './categories.component.html',

styleUrls: ['./categories.component.scss']

})

export class CategoriesComponent implements OnInit {

@Input()

cashflowCategories$: CashflowCategory[];

newCategory: CashflowCategory = new CashflowCategory();

isUpdating$: Observable<boolean>;

constructor(private settingsFacade: SettingsFacade) {

this.isUpdating$ = settingsFacade.isUpdating$();

}

ngOnInit() {

this.settingsFacade.loadCashflowCategories();

}

addCategory(category: CashflowCategory) {

this.settingsFacade.addCashflowCategory(category);

}

updateCategory(category: CashflowCategory) {

this.settingsFacade.updateCashflowCategory(category);

}

}

### **抽象层**

抽象层除了将表现层与核心层解耦之外同时也拥有自己的职责。这一层为展现层组件提供状态流（streams of state）和接口（interface），以及负责扮演门面（facade）模块。这种门面模块将组件在系统里的可见和可为都沙盒化。我们能简单的通过 Angular 的 provider 类来实现接口。这些类或者可以加以 Facade 后缀，比如SettingsFacade。下方就是门面的一个例子

*译者注：*

*我不确定 facade 应该如何翻译。直接翻译成“外观”似乎听上去有些奇怪，但又找不到更好的翻译方式。望赐教。（更新：感谢评论中指出应该翻译为“门面”）*

*我认为这里的抽象层对应就是 Martin Fowler 在 Enterprise Application Architecture 里的服务层（Service Layer） 。并且“沙盒化”对应的就是服务层里定义的应用边界（application's boundary），即说白了这个服务究竟能干些什么，完全由限定在服务层暴露的接口方法里。*

@Injectable()

export class SettingsFacade {

constructor(private cashflowCategoryApi: CashflowCategoryApi,

private settingsState: SettingsState) { }

isUpdating$(): Observable<boolean> {

return this.settingsState.isUpdating$();

}

getCashflowCategories$(): Observable<CashflowCategory[]> {

// 在这里，我们只是经过这个州，没有任何投影

//可能需要合并两个或多个流并公开给组件

return this.settingsState.getCashflowCategories$();

}

loadCashflowCategories() {

return this.cashflowCategoryApi.getCashflowCategories()

.pipe(tap(

categories =>

this.settingsState.setCashflowCategories(categories)));

}

// optimistic update

// 1. update UI state

// 2. call API

addCashflowCategory(category: CashflowCategory) {

this.settingsState.addCashflowCategory(category);

this.cashflowCategoryApi.createCashflowCategory(category)

.subscribe(

(addedCategoryWithId: CashflowCategory) => {

// 成功回调-我们有服务器生成的id，让我们更新状态

this.settingsState.updateCashflowCategoryId(

category, addedCategoryWithId)

},

(error: any) => {

// 错误回调-我们需要回滚状态更改

this.settingsState.removeCashflowCategory(category);

console.log(error);

}

);

}

// pessimistic update

// 1. call API

// 2. update UI state

updateCashflowCategory(category: CashflowCategory) {

this.settingsState.setUpdating(true);

this.cashflowCategoryApi.updateCashflowCategory(category)

.subscribe(

() => this.settingsState.updateCashflowCategory(category),

(error) => console.log(error),

() => this.settingsState.setUpdating(false)

);

}

}

### **抽象接口**

我们已经知道了这一层的主要职责是：为组件提供状态流和接口。让我们从接口开始讲起。公有方法 loadCashflowCategories() ， addCashflowCategory() 和 updateCashflowCategory()封装了从组件中抽离出了状态管理的细节以及外部的 API 调用。我们不在组件中直接使用 API provider (比如 CashflowCategoryApi) 因为它们存在于核心层中。组件也并不关心状态是如何变化的。表现层不应该关心工作是如何（how）完成的，组件在必要的时候只需要调用（just call）抽象层的方法即可（委托）。查看抽象层的公共方法能够让我们快速了解系统这部分的用户用例概况

但是我们应该记住抽象层不是实现业务逻辑的地方。这里我们只是将表现层和业务逻辑联系（connect）在了一起，并且把联系的方式抽象了出来

*译者注：Angular 采用了 reactive 编程思想，使用了 RxJS 作为基础类库。“流（stream）”是其中一个重要概念，接下来也会涉及其他和 RxJS 相关的函数和概念 ，比如 Observable，BehaviorSubjects*

### **状态**

至于状态，抽象层使得组件独立于状态管理解决方案。带有数据的 Observable 对象赋值给组件模板, 但组件并不关心数据是如何产生以及从哪来的。为了管理状态我们能够使用任何支持 RxJS (像 NgRx) 的状态管理类库，又或者仅仅使用 BehaviorSubject 对我们的状态建模。在上面的例子中我们使用的状态对象内部的实现借助于 BehaviorSubjects （状态对象是我们核心层的一部分）。如果是使用 NgRx 的实现状态管理的化，我们从 store 触发 actions。

拥有这类的抽象给了我们很大的灵活性，并且允许我们在不触碰表现层的情况下更改状态管理方式。甚至可能无缝的迁移到像 Firebase 这样的实时后端，让我们的应用变得实时（real-time）。我个人喜欢一开始使用 BehaviorSubjects 来管理状态。如果之后在开发系统的某个时间点有需要使用其他东西，在这个架构下，重构起来非常容易

*译者注：“状态解决方案”不是必须的，状态管理可以有，架构可以有，但不一定要借助于第三方类库来实现。React 可以有 Redux 架构，但是不一定需要 Redux 框架。仅仅依赖 React 自己的 hook 机制和 provider 就足以实现一套状态管理机制。*

### **同步策略**

现在让我们更进一步的看看抽象层的重要一面。无论我们选择什么样的状态管理解决方案，我们都可以以乐观或者悲观的方式实现界面的更新。想象我们想要在实体的集合中创建一条新记录。集合请求自后端并且在 DOM 中展示；在悲观的实现方式下，我们首先尝试在更新后端状态（比如通过 HTTP 请求），成功之后我们再更新前端状态。另一种乐观的实现方式是以不同的顺序执行。首先我们会在后端一定会更新成功的假设上立即更新前端。然后我们才发请求更新服务端状态。如果成功了，我们不用做任何事情，但是如果失败了，我们需要回滚前端的更改并且告知用户

乐观更新（Optimistic update）首先改变界面状态然后才尝试更新后端状态。这为我们的用户带来更好的体验，不会因为网络延迟看到任何的滞后。如果后端的那个更新失败了，界面更改必须回滚

悲观更新（Pessimistic update）首先更改后端状态并且只有在成功的情况下才更改界面状态。因为网络延迟，通常需要在后端的执行的过程中显示加载进度条

### **缓存**

有时候我们也许会认定来某些自后端的数据并不会成为我们应用状态的一部分。这通常是那些我们不会对它执行任何操作只是把它们（通过抽象层）传递给组件的只读（read-only）数据。在这个场景下，我们可以把数据缓存在门面模块中。实现它最简单的方式是使用 RxJS 的 shareReplay() 操作符，它能够为流的新的订阅者重放（replay）最新的数据。看看下面RecordsFacade使用RecordsApi为组件请求，缓存并且过滤数据的代码片段

@Injectable()

export class RecordsFacade {

private records$: Observable<Record[]>;

constructor(private recordApi: RecordApi) {

this.records$ = this.recordApi

.getRecords()

.pipe(shareReplay(1)); // cache the data

}

getRecords() { return this.records$; }

// 为组件规划缓存的数据

getRecordsFromPeriod(period?: Period): Observable<Record[]> {

return this.records$

.pipe(map(records => records.filter(record => record.inPeriod(period))));

}

searchRecords(search: string): Observable<Record[]> {

return this.recordApi.searchRecords(search);

}

}

总结下来，我们在抽象层能做的事情有：

1. 为组件提供接口方法：
2. 把执行逻辑委托给核心层
3. 决定数据的同步策略（乐观或者悲观）
4. 为组件提供状态流
5. 选取一个或多个界面状态流（如果有必要的话把它们结合在一起）
6. 从外部 API 中缓存数据

正如我们看到的，抽象层在我们的分层架构中扮演了一个非常重要的角色。它清晰的定义了能够帮助我们更好理解和推理系统的职责。依据你的具体例子，你可以给每一个 Angular 模块或者每一个实体创建一个门面模块。举个例子，SettingsModule仅有一个SettingsFacade。但有时为每一个实体创建更细力度的抽象门面会更好，比如UserFacade之于User实体存在。

### **核心层**

最后一层是核心层，这也是应用的核心逻辑实现的地方。所有的数据操作（data manipulation）和与外界的通信（outside world communication）都发生在这里。如果我们使用 NgRx 作为状态管理方案的话，这里就是放置 state，actions 和 reducer 的地方。因为我们的例子使用 BehaviorSubjiect 对状态建模的缘故，我们可以把它封装在一个便携的状态类中。你可以在下面找到来自核心层的 SettingsState 的例子

@Injectable()

export class SettingsState {

private updating$ = new BehaviorSubject<boolean>(false);

private cashflowCategories$ = new BehaviorSubject<CashflowCategory[]>(null);

isUpdating$() { return this.updating$.asObservable(); }

setUpdating(isUpdating: boolean) {

this.updating$.next(isUpdating);

}

getCashflowCategories$() {

return this.cashflowCategories$.asObservable();

}

setCashflowCategories(categories: CashflowCategory[]) {

this.cashflowCategories$.next(categories);

}

addCashflowCategory(category: CashflowCategory) {

const currentValue = this.cashflowCategories$.getValue();

this.cashflowCategories$.next([...currentValue, category]);

}

updateCashflowCategory(updatedCategory: CashflowCategory) {

const categories = this.cashflowCategories$.getValue();

const indexOfUpdated = categories.findIndex(

category => category.id === updatedCategory.id);

categories[indexOfUpdated] = updatedCategory;

this.cashflowCategories$.next([...categories]);

}

updateCashflowCategoryId(categoryToReplace: CashflowCategory,

addedCategoryWithId: CashflowCategory) {

const categories = this.cashflowCategories$.getValue();

const updatedCategoryIndex = categories.findIndex(

category => category === categoryToReplace);

categories[updatedCategoryIndex] = addedCategoryWithId;

this.cashflowCategories$.next([...categories]);

}

removeCashflowCategory(categoryRemove: CashflowCategory) {

const currentValue = this.cashflowCategories$.getValue();

this.cashflowCategories$.next(

currentValue.filter(category => category !== categoryRemove));

}

}

在核心层里，我们以 provider 类的形式实现 HTTP 查询，这种类有 Api或者Service 名称后缀。API 服务只有一个职责——除了与 API 端点通信外别无他用。我会应该避免任何的缓存，逻辑又或者数据操作。一个简单的 API 服务的例子如下：

@Injectable()

export class CashflowCategoryApi {

readonly API = '/api/cashflowCategories';

constructor(private http: HttpClient) {}

getCashflowCategories(): Observable<CashflowCategory[]> {

return this.http.get<CashflowCategory[]>(this.API);

}

createCashflowCategory(category: CashflowCategory): Observable<any> {

return this.http.post(this.API, category);

}

updateCashflowCategory(category: CashflowCategory): Observable<any> {

return this.http.put(`${this.API}/${category.id}`, category);

}

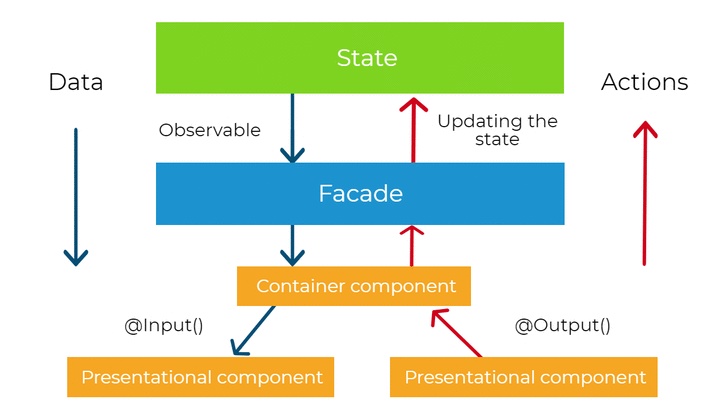
}

在这一层中，我们也会拥有校验，映射或者更多需要操作界面状态的高级用例。

我们已经谈论了前端应用中关于抽象层的话题。每一层都有它恰当的边界和职责。我们也定义了层之间严格的通信规则。随着时间的推移当系统变得越来越复杂时，这些所有都将更好的帮助我们理解和调试它

### **单向数据流和响应式状态管理**

下一个我想介绍的系统中的另一个原则是数据流（data flow）和变化的传播（propagation of change）。Angular 自身在表现层使用单向数据流（通过输入绑定的方式），但是我们也能在应用层面加以相同的限制。与（基于流式的）状态管理一起，它会赋予我们系统一个非常重要的属性——数据一致性（data consistency）。下图呈现了这种单向数据流的大致思路



无论何时只要我们应用中模型的值发生了变化，Angular 的变化监测系统都能够应对变化的传播。它借助对整棵组件树自顶向下（the top to bottom）的输入属性绑定来实现。这意味着孩子组件只能依赖父组件，并且永远不会依赖反转。这也是为什么我们称它为单向数据流。这允许 Angular 只会遍历组件树一次（only once）（因为树的结构中不存在循环）就能够取得一个稳定的状态，也意味着绑定里的值都能都能得到周知

*译者注：*

*1) 这里直接介绍和安利自顶向下的单向数据流机制。为什么不是能自低向上？为什么不能通过事件？*

*2) 如果你有 Angular 1.x （又被称为 AngularJS）的经验的话，AngularJS 的类似的脏检查机制并非如此。AngularJS 中脏检查被称为 Dirty Check，而 Angular 中被称为 Change Detection. Dirty Check 会不停的轮询所有被检视的变量直到没有变化发生。具体可以参考我的上一篇文章*

从前几章我们得知在表现层之上还有核心层，也就是我们实现应用逻辑的地方。那里有 services 和 providers 服务于我们的数据。如果我们把同样的原则也应用于那一层的数据处理上会怎么样？我们把应用数据（状态）放一个一个所有组件“之上”的地方，并且让值借助 Observable 流（Redux 和 NgRx 称之为 store）向下进行传播。状态能够传播到多个组件并且显示在多个地方，但是从不会在展示处发生修改。这些更改只会“来自上方”并且下方的组件只会“反映”系统的当前状态。这给予了我们系统上面提到的最重要的特性——数据一致性（data consistency）——状态对象变成了唯一数据来源（the single source of truth）。实际上说，我们可以多个地方展示同一份数据然后不用担心值会不同

我们的状态对象为核心层的各种服务提供了用于修改状态方法。当有需要改变状态时，只需要调用状态对象上的一个方法（在 NgRx 的例子里时触发一个 action）。接着，变更就会通过流“向下”传播给表现层（或者其他的服务）。这种方式就意味着状态管理是响应式（reactive）的了。不仅如此，因为严格的修改和共享状态的规则，我们可以增加我们系统的可预测性。下方是使用 BehaviorSubject 对状态建模的代码片段

@Injectable()

export class SettingsState {

private updating$ = new BehaviorSubject<boolean>(false);

private cashflowCategories$ = new BehaviorSubject<CashflowCategory[]>(null);

isUpdating$() { return this.updating$.asObservable(); }

setUpdating(isUpdating: boolean) { this.updating$.next(isUpdating); }

getCashflowCategories$() {

return this.cashflowCategories$.asObservable();

}

setCashflowCategories(categories: CashflowCategory[]) {

this.cashflowCategories$.next(categories);

}

addCashflowCategory(category: CashflowCategory) {

const currentValue = this.cashflowCategories$.getValue();

this.cashflowCategories$.next([...currentValue, category]);

}

updateCashflowCategory(updatedCategory: CashflowCategory) {

const categories = this.cashflowCategories$.getValue();

const indexOfUpdated = categories.findIndex(

category => category.id === updatedCategory.id);

categories[indexOfUpdated] = updatedCategory;

this.cashflowCategories$.next([...categories]);

}

updateCashflowCategoryId(categoryToReplace: CashflowCategory,

addedCategoryWithId: CashflowCategory) {

const categories = this.cashflowCategories$.getValue();

const updatedCategoryIndex = categories.findIndex(

category => category === categoryToReplace);

categories[updatedCategoryIndex] = addedCategoryWithId;

this.cashflowCategories$.next([...categories]);

}

removeCashflowCategory(categoryRemove: CashflowCategory) {

const currentValue = this.cashflowCategories$.getValue();

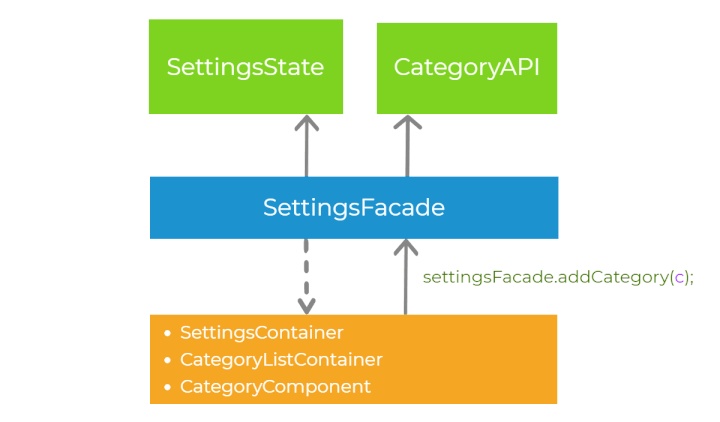
this.cashflowCategories$.next(

currentValue.filter(category => category !== categoryRemove));

}

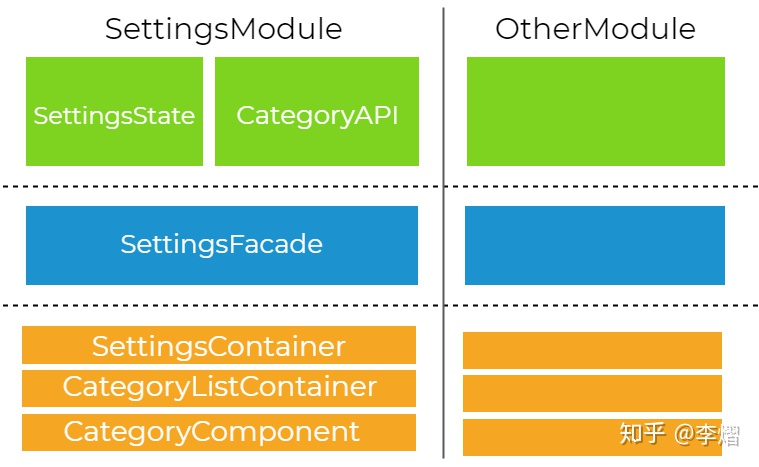
}

让我们基于我们已经学习到的原则复习一遍处理用户交互的步骤。首先让我们想象在表现层发生了一些事件（比如点击按钮）。然后组件通过调用门面模块上的 settingsFacade.addCategory()方法把执行委托给了抽象层。接着门面调用核心层服务里的方法——categoryApi.create()和settingsState.addCategory()。这两个方法调用的顺序取决于我们选择的同步策略。最终，应用状态通过 observable 流传递给表现层。这整个流程清晰明确（well-defined）

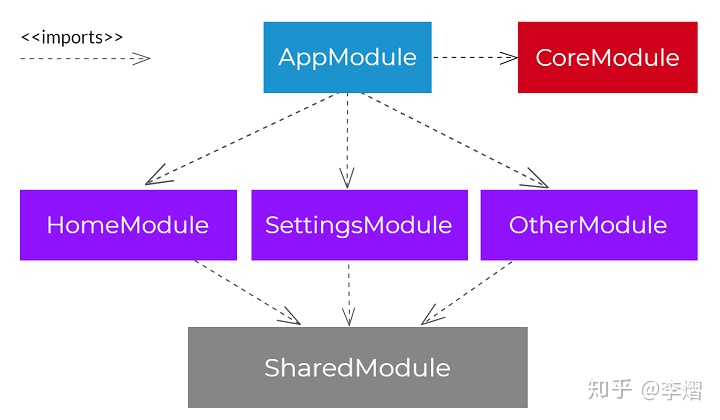


### **模块设计**

我们已经讲解了系统的横向拆分以及它们之间的通信模式。现在我们即将介绍如何纵向划分为功能模块。出发点是将应用划分为代表不同业务功能的特性模块（feature modules）。这也是为了更好的可维护性将系统分解为更小单元中的另一个步骤。每一个特性模块共享横向划分的核心层，抽象层和展现层。非常值得注意的是这些模块可以在浏览器中懒加载（和预先加载）以减少应用的初始加载时间。下方是这种特性模块划分图解：



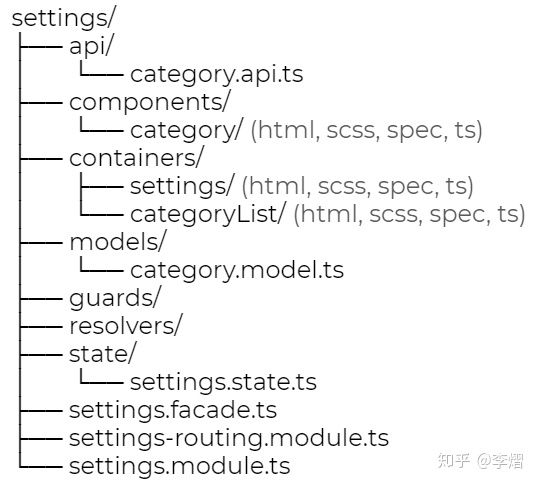
出于技术原因我们的应用里也有两个额外的模块。其中CoreModule用于定义我们的单例服务，单实例组件以及配置项，并且向 AppModule中导出任何的需要的第三方模块。这个模块只会在 AppModule中导入一次。第二个模块是SharedModule，它包含公共的 components /pipes/directives， 也包含公用的 Angular 模块（比如 CommonModule）。SharedModule可以被特性模块导入。下图展示了导入结构



*译者注：或许你阅读到这里的时候已经注意到了，与 React 不同，在 Angular 中有更多类型的角色定义，比如 provider，service，以及上面说的 CommonModule , SharedModule 等等。你觉得这些详细的职责划分是好事还是坏事？它们是有助于开发还是让开发变得更繁琐了？*

### **模块目录结构**

下图呈现了我们是如何将所有的SettingsModule的代码片段放入目录结构中的。我们可以把文件置于名字能显示它们功能的文件夹中



### **容器组件模式和展现组件模式**

我们最后一个介绍的架构模式是关于组件自身的。我们根据它们的职责将组件划分为两类。首先是容器组件（smart components aka containers）。这些组件通常特征有

有其他门面模块和服务的注入

与核心层通信

传递数据给展现组件

响应来自展示组件的事件

属于顶层组件（但也不绝对！）

之前展示的CategoriesComponent就是容器组件。SettingsFacade被注入其中并用于和我们应用中的核心层进行通信

第二个分类就是展示组件（dumb components aka presentational）。它们的唯一职责就是展示界面元素并且把用户交互以事件的形式“向上”委托给容器组件。想象一个类似于<button>Click me</button>的原生 HTML 元素。该元素并没有实现特殊的逻辑。我们可以认为文字 “Click me” 作为组件的输入。它也可以订阅一些事件，比如点击事件。下方就是一个简单的包含一个输入零个输出事件的展示组件的代码片段

@Component({

selector: 'budget-progress',

templateUrl: './budget-progress.component.html',

styleUrls: ['./budget-progress.component.scss'],

changeDetection: ChangeDetectionStrategy.OnPush

})

export class BudgetProgressComponent {

@Input()

budget: Budget;

today: string;

}

### **总结**

我们已经叙述了如何设计 Angular 应用架构的一些思路。这些原则如果应用恰当的话，能够在时间的推移中帮助我们维持一个可持续性的开发速率，并且使得交付新功能更加容易。请不要把它们当作一些严格的规则，而是在需要时可以采纳的建议。

我们已经近距离细观察了抽象层，单向数据流和响应式状态管理，模块设计，容器组件/展示组件设计模式。我希望这些概念能对你的项目有所帮助，并且一以贯之的是，如果你有任何的疑问，我很乐意与你交流

最后，我想致敬写了这篇博客文章的 Brecht Billiet，给了我抽象层和门面模块的灵感。多谢 Brecht ! 也多谢 Tomek Sulkowski 对我分层架构的观点进行了评审

*译者注：如果一定要做一个映射的话，这篇文章里的三层架构应该如何与 React 相关的架构和框架做对应？*

*<https://zhuanlan.zhihu.com/p/47396514>*

*<https://zhuanlan.zhihu.com/p/157890922>*

**序**

这篇文章或许看上去仅和 Angular2 开发者相关，但我相信它也适用于其它的框架。这只是一份关于编写具有可拓展性和可维护性单页面应用的指南。需要指出的非常重要的是，这并不是达成目标的唯一方式，但是对我个人而言它们在不少的场景中都行之有效

**编写可拓展性的单页面应用**

许多开发者在编写大型的具有可拓展性和可维护性的单页面应用时都遇到困难。导致在开发早期就留下了技术债，修复 bug 时举步维艰，编写测试和创建复用代码时也踌躇不前

最大的一个挑战是：在一个拧巴的基础之上拓展现有逻辑和编写新的功能

对于那些能允许你用100种不同方式设计一个应用，没有结构和封装可言，一切东西都紧紧的耦合在一起的旧时框架，单页面应用是全新的概念

大部分时候在项目的开发的初始阶段都希望快速迭代。但是经过一些开发者，几轮功能迭代和重构之后，代码变得越来越难以维护。它开始看上去像意大利面了。虽然目前框架成熟了很多，但重要的是你编写的软件架构也要与时俱进

**2016 年的单页面应用（甚至更早之前）**

为了解释这篇文章谈到的架构，有必要回顾一下2016年的web应用长什么样子。这些概念是你的应用在今天也会用到的。无论使用的是 React，Angular 2 又或者其它的框架。这些已知的原则能够让你的 web 应用变得易于维护

*译者注：这已经算非近代了，如果你们想有兴趣了解2013年左右的单页面是怎么做的，可以去考察一下 Backbone.js 或者 Ember.js 以及 AngularJS。那个年代没有单项数据流，没有组件的概念。只有 MVC*

**原则1：组件**

像 React 和 Angular 2 这样的 SPA 技术让我们开始使用组件。一个组件是 HTML 片段和 JavaScript 和结合。我们不想再使用独立的 view 和 controller。因为它们会出现爆炸式增长并且相互关联导致难以维护

*译者注：你真的明白这最后一句话 “爆炸式增长” 和 “相互关联导致难以维护” 是什么意思吗？为什么会出现这样的情况以及这究竟是怎样一个场景？*

所以最基本的原则是：万物皆是组件（everything should be a component），甚至你的页面和应用也是。一个应用可以长成这个样子：

<application>

<navbar fullname="Brecht Billiet" logout="logMeOut()">

</navbar>

<users-page>

...

<grid data="users">

</grid>

...

</users-page>

</application>

一些在设计组件时非常简单但又重要的提示是：

保持它们尽可能的小

保持它们尽可能的仅与绘制界面相关

*译者注："仅与绘制界面相关" 在原文中就一个词：dumb, 即 dumb component。仅负责绘制界面但不包含业务逻辑和数据存储的组件*

如果你是设计组件的新手，这篇文章或许能帮助你：components demystified

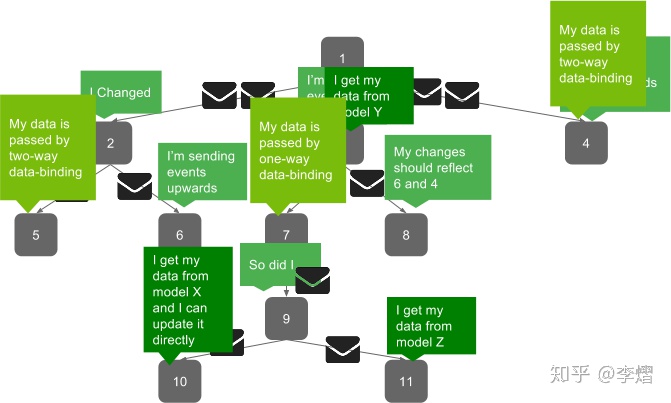
注意：聪明组件（smart component）也被称为结构组件（structural components），容器组件（containers）或者含状态组件（stateful componnets）

**原则2：单向数据流**

在此之前我们通过一种非常没有效率的方式来更新应用的状态：

1. 我们尝试让兄弟组件相互通信
2. 父组件尝试通过 action 通知子组件
3. 我们尝试在不同的组件间发送事件
4. 我们使用单向绑定，双向绑定
5. 我们把数据模型注入的到处都是用以共享状态

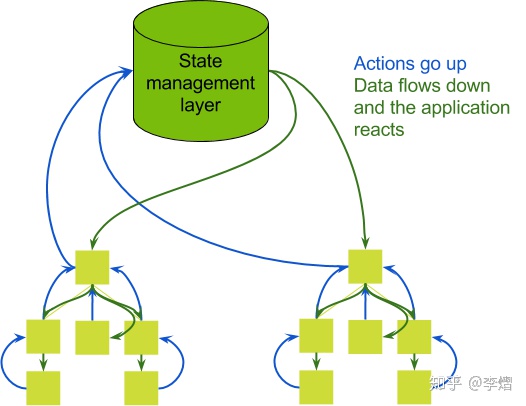
你有尝试过让兄弟组件相互通信吗？有时候这么做看上去理所应当，但是请不要这么做



这是一个糟糕的设计！这样的话几乎无法看清数据的流动方向。这样的代码无论是对于修复 bug 又或者开发功能也非常难以维护。我们想要像 Flux 和 Redux 的单向数据流

*译者注：这个图是一个很好的关于无方向的数据流导致代码失控的例子，但其中有些问题和序号被遮挡了。我猜这应该是来自一个 PPT。但可惜我没能找到原图。*

它基本上的工作原理是这样的：子组件只会通知它们的父组件，父组件（容器组件）向包含状态的 store 发送一个 action，action 会更新整个应用状态。当状态被更新之后，我们会重新计算组件树。结果就是数据朝相同的方向流动（向下）。



这种方式的最大好处是：

1 使得组件之间解耦 、

2 好的可维护性

3切换为实时应用的代价较小，因为软件是响应式的（reactive）

4通过监控 action 我们就知道发生了什么

如果你是单向数据流的新手请访问 introduction to redux 关于 单向数据流（unidirectional dataflow） 的部分

**可拓展架构**

我在设计一些架构时会参考当今（大部分上面都解释过了）乃至十年前的原则。但这对每个 SPA 都是成立的吗？不尽然……每一种软件类型都值得拥有它自己的架构，我只是展示一种对于我而言在大多数场景下都成立的理念。这个架构对于正在受益于依赖注入特性的 Angular 2 开发者而言迟早会有用，但也能适用于其它的框架中

**有意义的抽象**

这个原则部分是基于 Nicholas Zakas 的沙盒原则（Sandbox principle），它已经有一些年头了。对于 Nicholas Zakas 而言，沙盒扮演着包含一个容器组件的几个组件间的调度器（dispatcher）的角色。在这个架构中，没有单向数据流。对我而言一个沙盒是一种将应用逻辑与表现层解耦的方式，但那并不是它唯一的职责。

但是让我们从头开始……在这个具体的场景中我将假设我们使用 Redux，但其实无论你使用什么样的状态管理，了解它背后的原理非常重要。

**规则一：不要让你的组件接触到所有的玩具**

这和你不能让孩子玩耍所有的东西背后的原因一样：“可能会变得一团糟”。容器组件应该遵循一套严格的规则。不能因为我们能做到而把我们想要的每一个服务都注入进去。举个例子：把游戏引擎注入仅权限模块似乎就没有任何意义

下面的例子展示了一个拥有非常多依赖的构造函数。在这个场景里MyComponent可以在应用里做任何它想做的事情。任何它想要的想要做的都被注入其中。这不是好主意

export class MyComponent{

constructor(...,private foo:Foo,

private bar: Bar,

private store: Store<ApplicationState>,

private authService: AuthService,

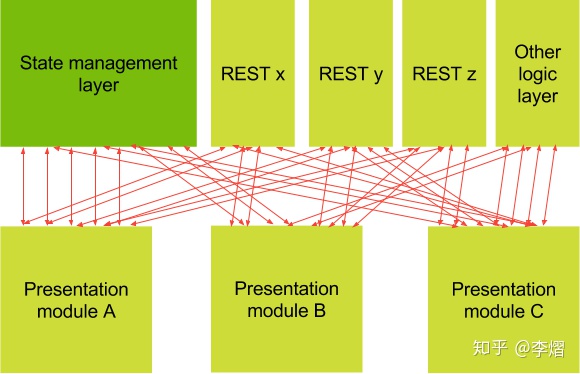
private fooHttpService: FooHttpService,

private barMapper: BarMapper, ...){

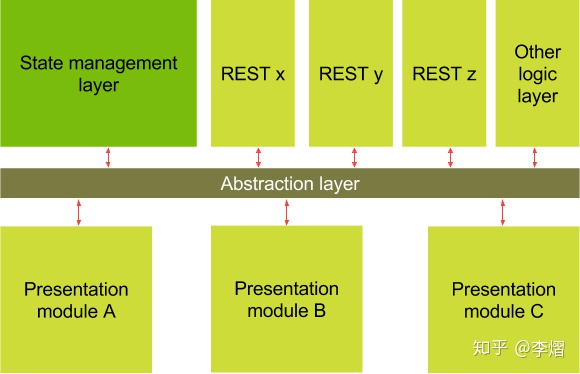
}

}

上面的例子的构造函数里有大多的依赖了。它与太多应用里的其它部分产生了联系。当向复盘这是一种怎样的设计时结果看上去像这个样子：（REST 表示 restful 服务，所以它们只是 HTTP 模块）



这看上去就像意大利面代码，一切都是紧耦合的。一个抽象层能够给予一些帮助。下面的例子中我们看到表现层已经完全解耦，抽象层接管了所有事情



抽象层代码大概像这个样子：

export class MyComponent{

constructor(private abstraction: SomeAbstractionType){

}

doSomething(): void{

this.abstraction.doSomething();

}

*译者注：添加了 abstraction layer 之后从外观上整体上看上去清晰了很多，但有没有可能这种混乱的关系被封装在abstraction layer 中而已？它应该如何在内部保持调理清晰？*

**规则二：组件应该对状态管理层不知情**

展示组件和容器组件都不应该知道你用的是 Redux 还是其它的状态管理层。它们不应该关心状态是如何被管理的。它们只是相信状态被管理的很好。管理也不是它们的职责。表现层的职责是“展现”和“委托”

下面的代码片段是一个糟糕设计下组件和 redux 紧耦合的反面例子（在这个例子中我们使用 ngrx/store ）

export class MyComponent{

// we just care about the state, not where it comes from...

users$ = this.store.select(state => state.users);

foo$ = this.store.select(state => state.foo);

bar$ = this.store.select(state => state.bar);

constructor(...,

private store: Store<ApplicationState>, ...){

}

addUser(user: User): void{

// we don't care about the actiontype, payload or store

this.store.dispatch({type: ADD\_USER, payload: {user}}

}

removeUser(userId: string): void{

// we don't care about the actiontype, payload or store

this.store.dispatch({type: REMOVE\_USER, payload: {userId}}

}

}

这样或许更加清晰，也更松耦合：

export class MyComponent{

users$ = this.facade.users$;

foo$ = this.facade.foo$;

bar$ = this.facade.bar$;

constructor(private facade: ...){

}

addUser(user: User): void{

this.facade.addUser(user);

}

removeUser(userId: string): void{

this.facade.removeUser(userId)

}

}

现在组件真正的聚焦在它自己的职责上了。它不知道用户是如何被添加或者移除的，它不知道谁产生的 users$ 流，foo$流和bar$流。它把这些职责都托付给抽象层。现在，表现层已经完全和剩余的应用部分解耦了。给我们带来以下的优势：

我们终于有了封装

便于测试，我们只需要 mock 抽象层

如果抽象层保证它的接口不发生变化，它们就能并行开发

它响应抽象层，因此实时开发变得容易许多

表现层与应用中其它的部分不再耦合，所以重构变得更加容易

*译者注：注意上面的优势没有一条是和确切的实现功能相关的，满足的都是所谓的“非功能需求”，在我们公司内部更习惯称之为“跨功能需求”。如果你所在的项目不讲究非功能需求，不在乎测试并且每两年就推倒重来的话。架构对你们来说并没有太大意义*

**规则三：HTTP 服务不应该知道状态管理层**

第一眼看上去，似乎把处理 GET 请求放在含有 store 的服务中是顺理成章的事情。但是一个 HTTP 服务的唯一目标是发出 HTTP 请求并且返回这些请求的结果。代码顺理成章的看上去长这样：

export class UserService{

// expose the users$-stream directly in the service

users$ = this.store.select(state => state.users);

constructor(private store: Store<ApplicationState>, private http: Http){

}

fetchUsers(): void{

this.http.get("...")

.map(...)

.subscribe((users) => {

// when successful, put the users in the store

// is this really my responsibility?

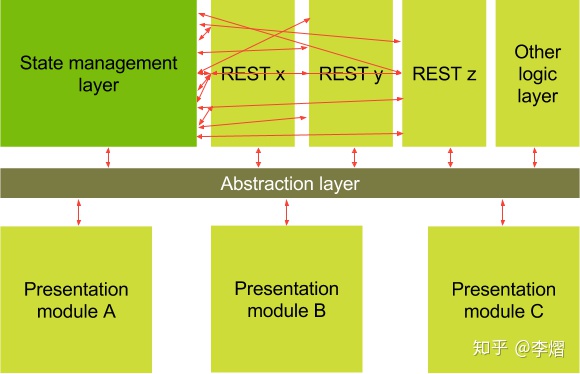
this.store.dispatch({type: SET\_USERS, payload: {users}});

});

}

}

但是现在 http 服务于状态管理系统变得非常的紧耦合



http 服务的唯一职责应是如此

export class UserService{

constructor(private http: Http){

}

// just let the consumer of this service handle the store interaction

// this will just return a stream of users

fetchUsers(): Observable<Array<User>>{

return this.http.get("...").map(...)

}

}

**一个可能的解决方案**

这当然不是唯一的解决方案，但是它对我来说绝大部分时候都管用。每一个展现模块能够访问它自己的沙盒（sandbox），沙盒一个有以下功能服务：

状态流（在这个例子中从 redux 中选取）

（消费沙盒的）展现模块可能执行的方法

但是它不仅仅是一个外观模块:) 它应该有部分的逻辑：

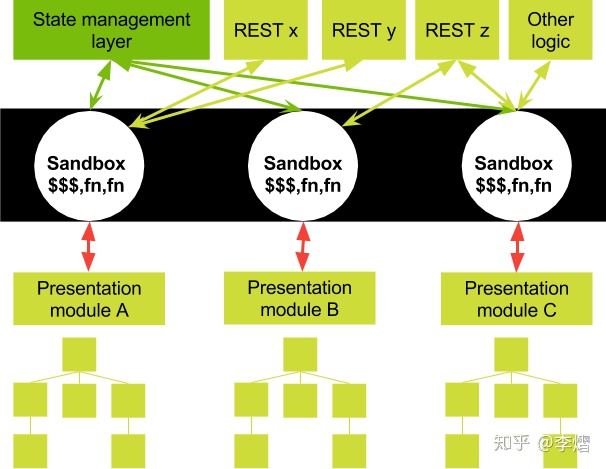
1 从 store 中抓取正确的状态碎片

2 向 store 派遣拥有正确类型和正确负载的操作

3 委托给不同模块里正确的服务

4 处理乐观更新(optimistic updates)

5 如果使用了类似于 firebase 的服务，暴露来自 firebase 的正确数据流



沙盒的好处有

* 将表现层与其它部分解耦
* 抽离状态管理
* 能够看清一个模块究竟能访问什么
* 组件没法为所欲为，更好的封装
* 把乐观更新逻辑放对了地方，因为组件和服务都不关心这个功能。
* 在不用重写服务和组件的情况下能够切换到不同的状态管理下面

译者注：注意这个 sandbox 的几个特质：

* 它只能单向的去调用 state management layer 和 rest 服务，不能反之
* sandbox 之间不能相互调用

sandbox 更像是一个调度员，它来协调组件想操作的一切。它还是应用的边界，它暴露的方法决定了整个应用的行为

## **总结**

这整个架构是其中的一种方式，但并不意味着是唯一方式。

我们相信封装、松耦合以及正确的职责体系是非常重要的。前端领域更新的非常快，也意味着我们想要更加灵活的应对重构和想尝试一些新技术。我希望你喜欢这篇文章。

## **想要学习更多**

请访问 [strongbrew.io](https://link.zhihu.com/?target=http://strongbrew.io/" \t "https://zhuanlan.zhihu.com/p/_blank)，[我](https://link.zhihu.com/?target=http://twitter.com/brechtbilliet" \t "https://zhuanlan.zhihu.com/p/_blank)和[Kwinten Pisman](https://link.zhihu.com/?target=http://twitter.com/kwintenp" \t "https://zhuanlan.zhihu.com/p/_blank)在上面开设一个了“反应式应用工作坊（reactive applications workshop）”来更谈论这个主题更多的细节

## **特别感谢**

我想要感谢所有评审这篇文章和给予价值输入的人。感谢 [Jurgen Van de Moere](https://link.zhihu.com/?target=http://twitter.com/jvandemo" \t "https://zhuanlan.zhihu.com/p/_blank)，[Carmen Popoviciu](https://link.zhihu.com/?target=http://twitter.com/carmenpopoviciu" \t "https://zhuanlan.zhihu.com/p/_blank)，[Manfred Steyer](https://link.zhihu.com/?target=http://twitter.com/manfredsteyer" \t "https://zhuanlan.zhihu.com/p/_blank) 和 [Juri Strumpflohner](https://link.zhihu.com/?target=http://twitter.com/juristr" \t "https://zhuanlan.zhihu.com/p/_blank)！！！

<https://zhuanlan.zhihu.com/p/159478723>

**复杂应用**

一个复杂应该至少拥有以下某些特征

* 组件树中的多个组件展示同一份应用状态
* 拥有多个更新应用状态的来源：
* 多个用户同时交互
* 后端实时推送状态更新给浏览器
* 后台定时任务
* 近距离传感器或者其它设备传感器
* 频繁的更新应用状态
* 大量的组件
* 代码量大的组件，回想一下之前的[大泥球](https://link.zhihu.com/?target=http://www.laputan.org/mud/" \t "https://zhuanlan.zhihu.com/p/_blank)般的 AngularJS controller
* 组件内部的高度复杂循环——高度集中的逻辑分支和异步控制流

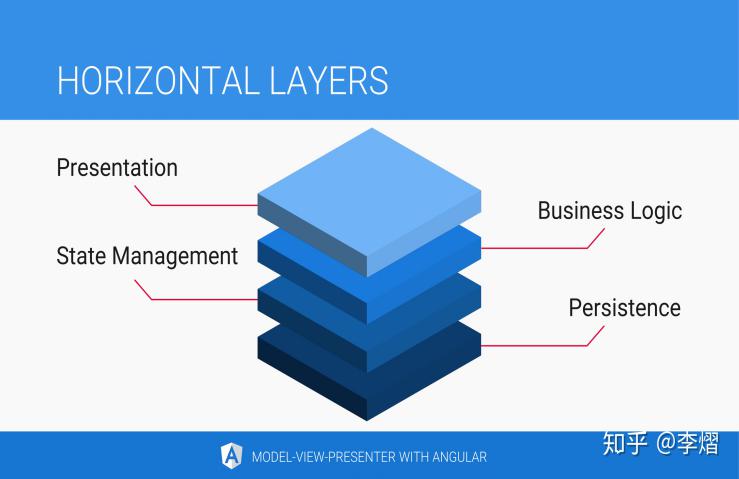
但与此同时，我们希望应用是具有可维护的，可测试的，可拓展的和具有良好性能的

复杂的应用很少拥有所有这些宝贵特征。我们也不能在完成高级功能需求的情况下避免这些所有的特征，但是我们可以通过设计应用来最大化利用它的宝贵特征

*译者注：我理解这些特征，但是我不理解为什么作者把这些特征称之为“宝贵特征（valuable traits）”。在我看来这些复杂的行为正是应用变得难以维护的原因之一。我们的设计应用的目标是要保证它的简单和清晰。我们不应该想法设法的去争取这些特征，而是应该想方设法的避免它们*

## **分离关注点**

我们可以将*分离关注点（separation of concerns）*作为应用的分层基础。我们可以按照系统的关注点组织逻辑，以便独立的关注它们。在所有之上，分离关注点是首要的架构原则。在日常开发中无论何时它都应该改被铭记于心

**