CoDia 课达编程

如何写好前端

近期目标

CODIA完善

- 代码规范和最佳实践
- 产品自我体验
- 交叉审查

如何写好前端

目录

编程思想

- 函数式思想
- 分层抽象和组件分离

编程思想

函数式思想

- 过程式 vs 函数式

```
const arr = [1, 2, 3, 4, 5]

const r = []
for (let i = 0; i < arr.length; i++) {
  if (arr[i] % 2 === 1) {
    r.push(arr[i] * arr[i])
  }
}
```

定义构造过程

```
const arr = [1, 2, 3, 4, 5]

const r = arr
  .filter((x) => x % 2 === 1)
  .map((x) => x * x)
```

定义映射关系

编程思想

函数式思想

- 现代前端框架：维护**数据**到**界面**的映射
- 响应变化，按需更新

编程思想

函数式思想

- Vue 中的“过程式”和“函数式”

```
const errors = ref([])

watch(pack, (data) => {
  for (let i = 0; i < data.nodes.length; i++)
    errors.value.push(func(data.nodes[i]))
})
```

手动监视变量变化、构造结果

```
const errors = computed(() => (
  pack.value.nodes.map(func)
))
```

维护映射，vue自动更新变量

编程思想

函数式思想

- 处理服务器结果

```
const { result } = useQuery(
  // ...
)

const state = ref('')
watch(result, (data) => {
  if (data) state.value = data.name
})
```

```
const { result } = useQuery(
  // ...
)

const state = useResult(
  result,
  '',
  (data) => data.name,
)
```

编程思想

函数式思想

- 补充学习：js数组的函数式API
  - map
  - reduce
  - filter
  - find
  - reverse
  - slice
  - every, some
  - forEach

编程思想

分层抽象和组件分离

- Vue组件
  - props：输入
  - emits：输出

编程思想

分层抽象和组件分离

- 输入输出设计：明确层次
- CODIA前端分层：界面层和业务层
  - 界面层：负责内容呈现、界面交互，不关心业务操作
  - 业务层：使用界面层组件实现业务逻辑，不关心界面细节

编程思想

分层抽象和组件分离

- 例子：题目选择对话框
  - 界面层组件
  - 目的：选择题目，呈现错误和加载状态
  - 不关心：何时打开、何时关闭、初始列表从何处获取、最终列表用于何处（在业务层处理）
  - 输入：初始已选列表、错误和加载状态
  - 输出：点保存时 emit 选择的题目列表

编程思想

分层抽象和组件分离

- 例子：题包内容选项卡
  - 业务层组件
  - 目的：编辑题包的题目列表
  - 不关心：如何让用户点选题目
  - 点击【选择题目】打开题目选择对话框（输入初始列表）
  - 接收到保存事件（包含用户选择的列表）时调用mutation
  - 错误和加载状态传入对话框由对话框负责呈现

代码规范

- Lint
- 提交规范
- 重复代码问题

代码规范

Lint

- 处理所有警告
- 之后会打开提交时 lint 检查，要求必须通过 lint 才能提交

代码规范

提交规范

- 多提交：做完一件很小的事就提交一次
- 实现一项完整的功能改进即提交mr，不要等整个任务做完
- 同步进展：git fetch && git rebase origin/master

代码规范

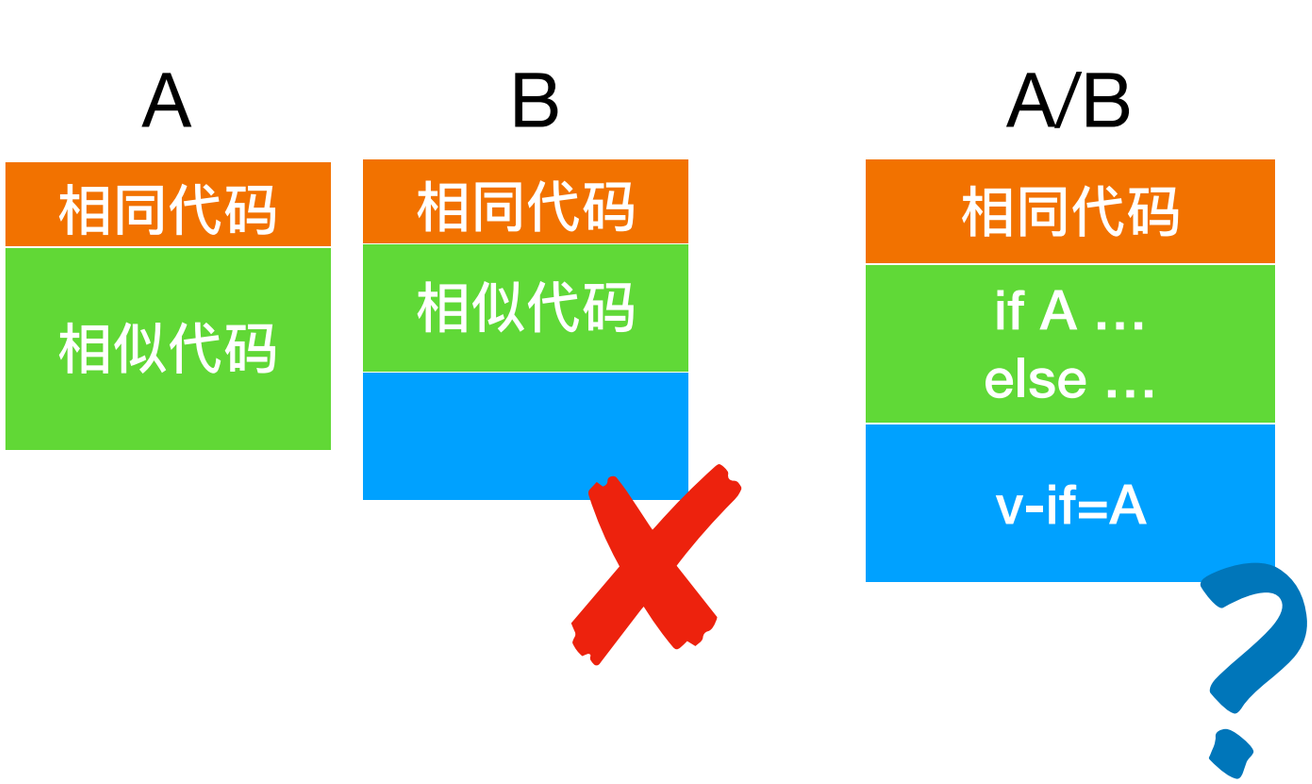
重复代码问题

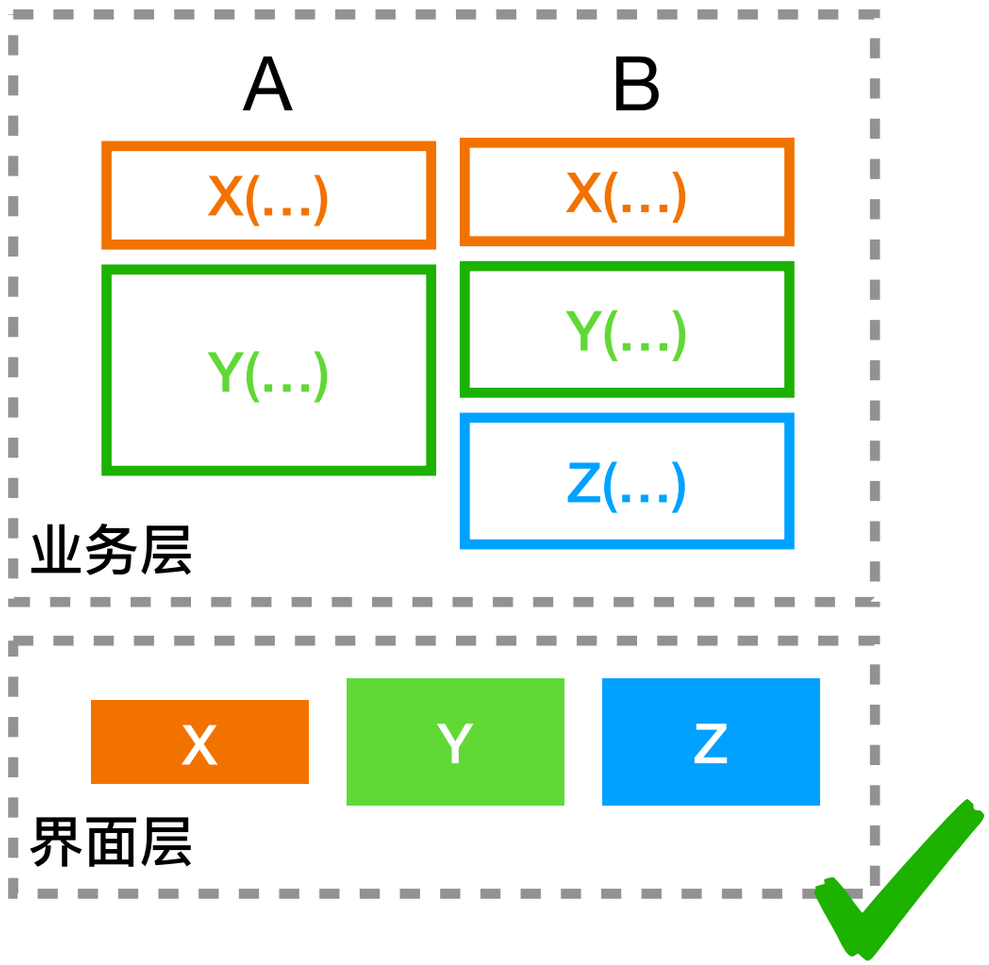
- 重复代码的危害
  - 代码冗长繁复，降低可读性
  - 一处逻辑修改 = 多处代码修改，复杂且容易遗漏
  - 复制粘贴时容易遗漏细节，导致bug

代码规范

重复代码问题

- 解决：分离组件和分层抽象





代码规范

重复代码问题

- 使用工具


```
// 编写查询
const { result, loading } = useQuery(
  gql`query queryPack($pid: ID!) {
    node(id: $pid) { ... on ExercisePack { ... } }
  }`,
  { pid },
)
// 选择结果
const pack = useResult(result, undefined, (data) => data.node)

// 结果解析
// 当前用户的题包状态，如：是否正在进行，截止时间，是否全部完成，等
// 得到每项均为 ComputedRef
const { ongoing, due, finished } = usePack(pack)
```

最佳实践

- 代码组织
- 查询编写
- 加载状态处理
- 移动端优先设计

最佳实践

代码组织

- 逻辑分段
- 通过空行和注释体现分段
- 逻辑分段 -> 将逻辑抽取为工具函数

Composition API

最佳实践

查询编写

- 对象查询
  - **所有对象都需要查询 id**
  - 仅使用 node 接口根据 id 查对象
  - 使用 useResult 选取对象
  - 使用数据工具

https://codia-docs-155-doc.env.bdaa.pro/guide/frontend/cookbook/data.html#数据查询

最佳实践

查询编写

- 列表查询
  - 使用 usePagination
  - 结果处理：
    - 使用 computed + use functions
      https://codia-docs-155-doc.env.bdaa.pro/guide/frontend/cookbook/data.html#使用数据处理工具
    - 交由呈现/处理单个对象的组件处理

https://codia-docs-155-doc.env.bdaa.pro/guide/frontend/cookbook/data.html#分页

最佳实践

查询编写

- fetchPolicy 和 errorPolicy
  - fetchPolicy
    - 不需要更新、或可在mutation中通过更新cache更新：保持默认
    - 通过mutation难以更新：cache-and-network
    - 可先使用 cache-and-network，后续完善相关 mutation 后改回默认
- errorPolicy
  - 涉及权限问题可能无法访问部分字段时使用 ignore
  - 其余情况保持默认（即不加此选项）
  - 未被 ignore 的错误需要呈现/处理

最佳实践

加载状态处理

- 为什么需要处理加载状态?
  - 增加用户对等待的容忍度，避免“死机感”
  - 减少界面布局跳动
- 最佳实践：
  - 存在空内容导致区域缩小时，放置 placeholder
  - 翻页时上页内容已自然构成占位，可仅通过按钮变灰体现加载中
  - 对话框保存等情况，可使用按钮的加载状态
  - 避免过多部分闪烁

最佳实践

移动端优先设计

- 移动端优先：
  - 中心思想：首先为移动端设计界面
  - 移动端设计很多时候可直接在更大屏幕上使用，即使需要改动也相对比较容易
  - 将平面设计变为线性设计，能够降低设计难度和成本
  - 排除干扰，则需要更多考虑内容的**结构**、**主次**、**内在关联**等
  - 在界面上通过间距体现结构，通过字体大小和粗细体现主次