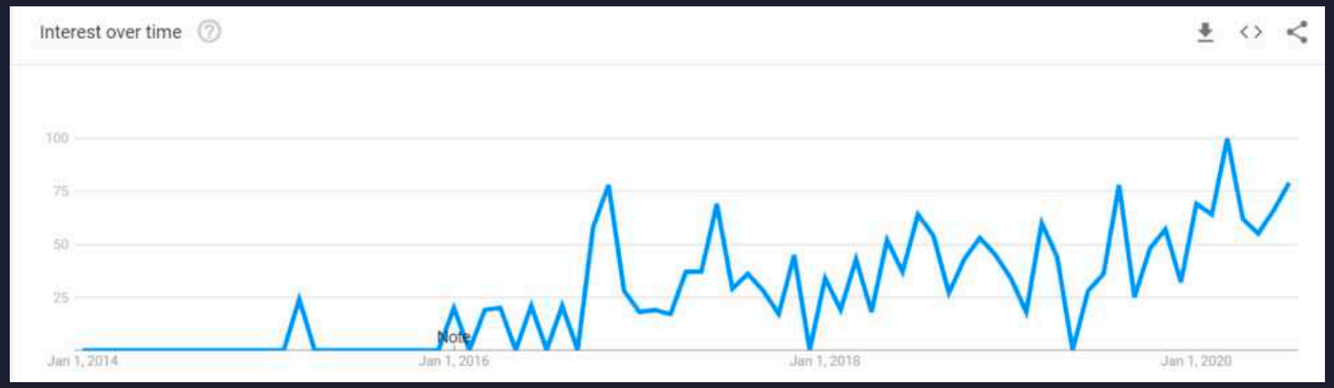
QCon+ 案例研习社

中后台低代码的思考和实践

网易云音乐前端专家 / 葛星

目录

为什么需要低代码平台- 业界的趋势



Google的搜索趋势

Gartner评估超过18家 LowCode 平台中
总规模超过8000亿美金，但中国市场刚刚起步

为什么需要低代码平台?

生产力低下与业务高速发展的矛盾

40+的工作台
随着业务的发展正不断的扩大

数据应用&公共技术

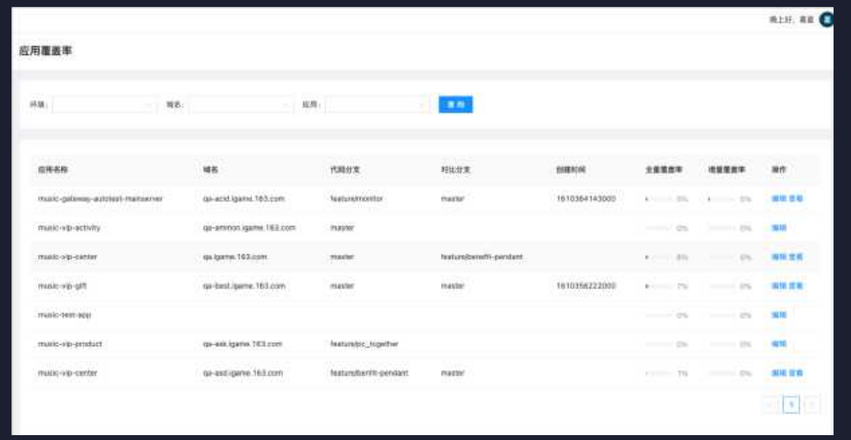
- 激励管理
- 增长引擎（Bamboo)
- 。。。
- 洞察系统（Insight)
- 版权评估
- 用户画像（Sniper）
- 推歌系统（Sniper）

业务规模化

需求多样化

- 声波
- 云音乐
- 音街
- Look直播

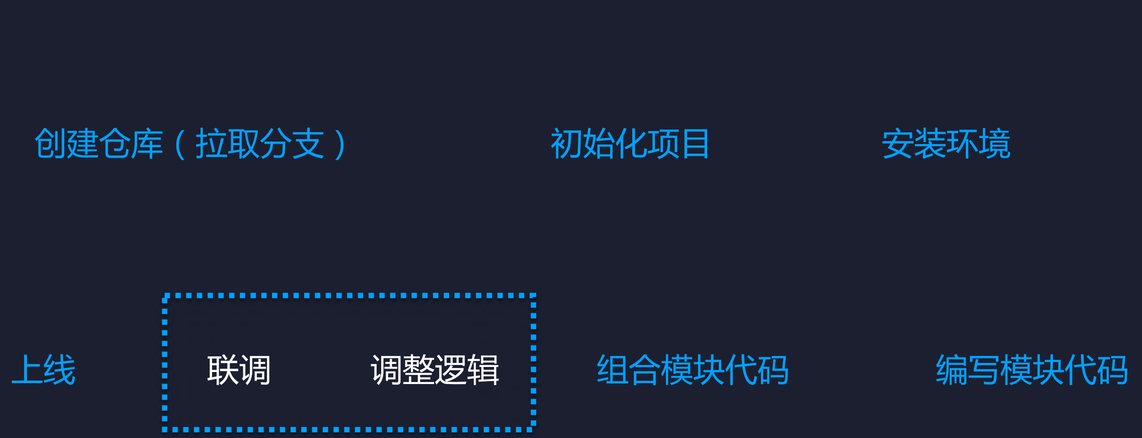
场景模式化



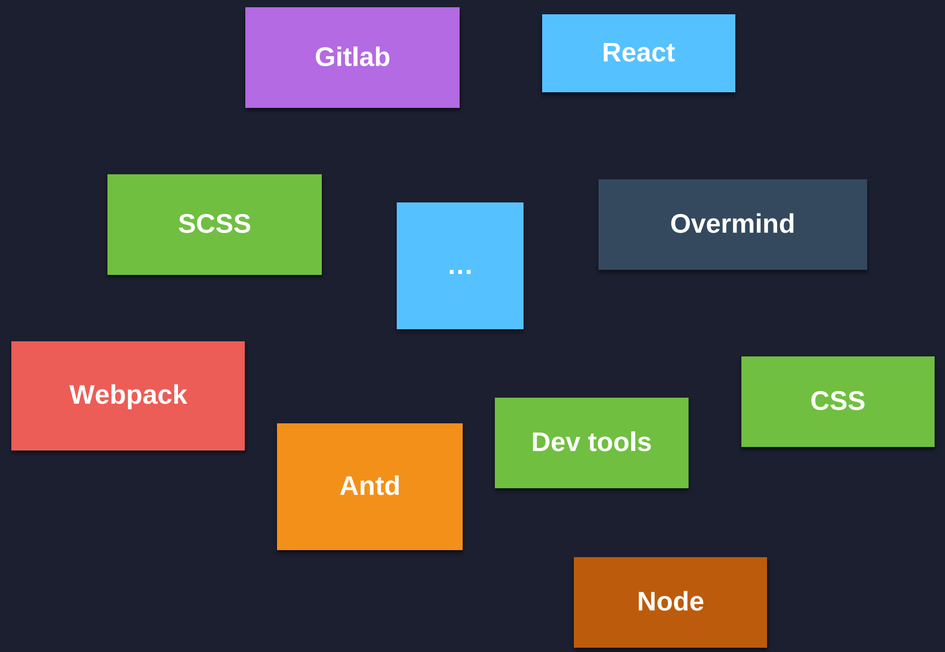
大量场景模式化，
搜索列表页，详情，数据看板等 Web侧65%左右，
中长尾页面较多

为什么需要低代码平台 - 核心问题域

- 如何加速软件的研发过程？ ⟶ 模型驱动
- 如何降低软件的研发门槛？ ⟶ UI可视化编程



24小时的研发过程



低代码平台的核心 - UI可视化编程

?

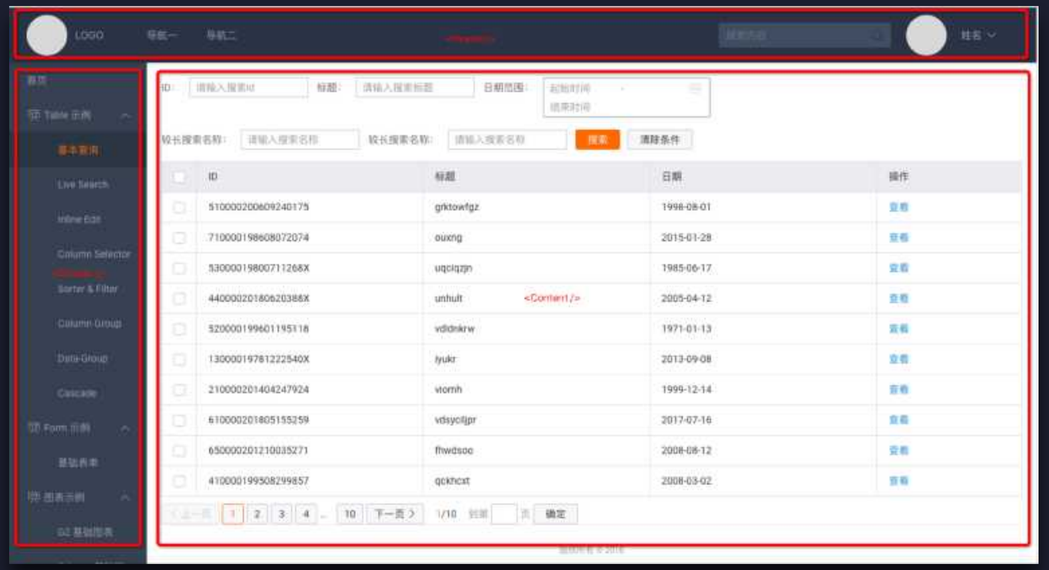
物料 → 可视化设计器 → UI

编辑时 点选，拖拽，配置能力

预览时 所见即所得的能力

UI可视化编程的模型就是将物料编排出产物，通过产物渲染UI的过程

低代码平台的核心 - 传统UI可视化的本质



<Header></Header>

<Sidebar></Sidebar>

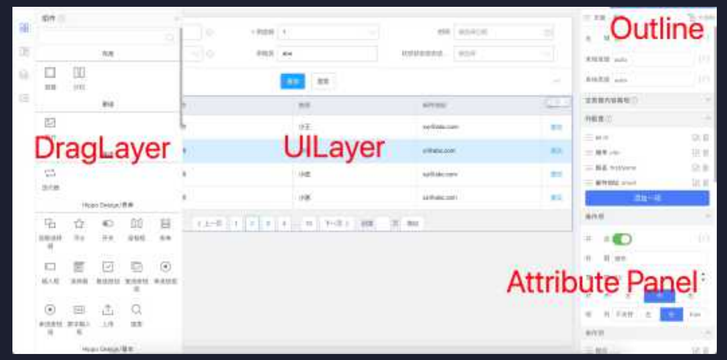
<Content></Content>

?

```
[{
    type: 'Header',
    attributes: {},
    children: []
}, {
    type: 'Sidebar',
    attributes: {},
    children: []
}, {
    type: 'Content',
    attributes: {},
    children: []
}]
```

产物就是**JSON数据**

低代码平台的核心 - 传统UI可视化的整体流程



Drag Layer

拖拽

拖拽

UI Layer

UI

Outline

生成

Mock data

渲染

JSON

写入

收集

收集

Visual Engine

收集

Attribute Panel

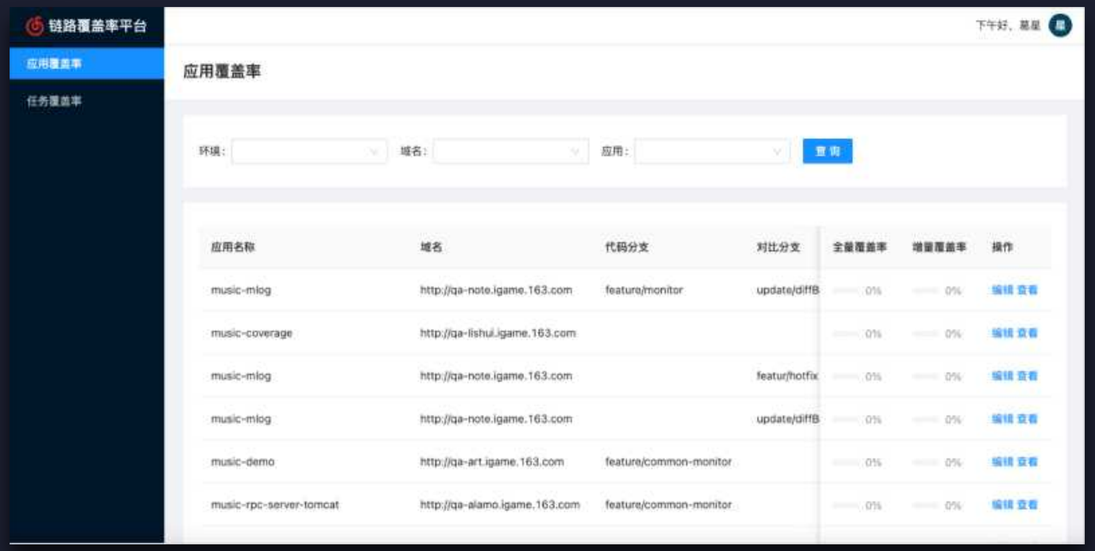
低代码平台的核心 - UI可视化编程

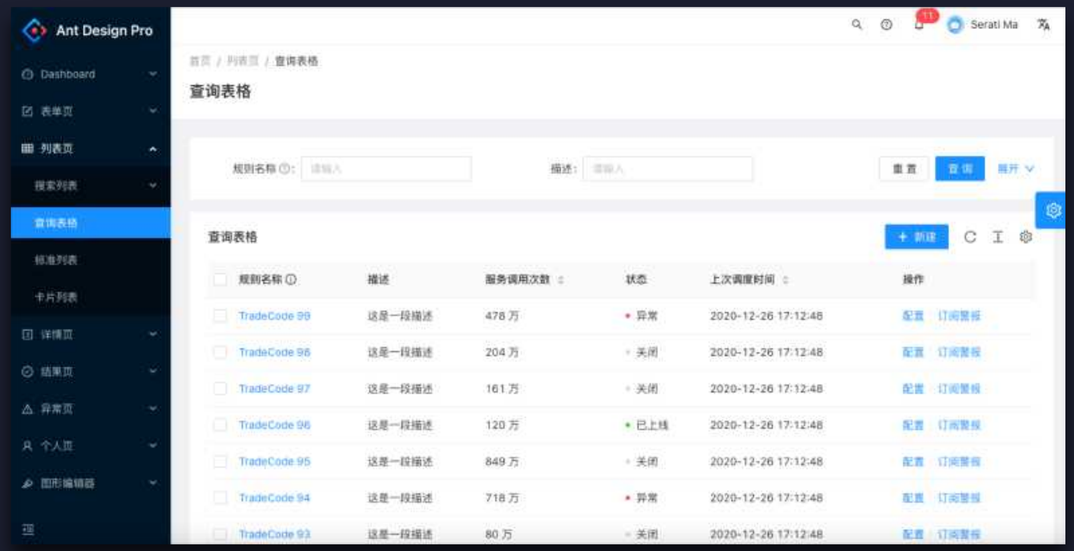
中后台系统的可视化搭建和C端搭建的不同点

必然涉及到不同程度的逻辑编写

低代码平台的核心 - UI可视化编程

想象下页面越来越复杂的情况





一条路走到黑，没有回头路
稍微复杂的场景下效率不如写代码

低代码平台的核心 - UI可视化编程

传统UI可视化编程之殇

可维护性

- 不断膨胀的JSON协议规范
- 看不到边界的JSON描述

```
"attributes": {
  "prop1": 1234, // 简单 json 数据
  "prop2": [{  // 简单 json 数据
    "label": "选项1",
    "value": 1
  }, {
    "label": "选项2",
    "value": 2
  }],
  "prop3": [{
    "name": "myName",
    "rule": {
      "type": "JSExpression",
      "value": "/\w+/i"
    }
  }],
  "valueBind": { // 变量绑定
    "type": "JSExpression",
    "value": "this.state.user.name"
  },
  "onClick": { // 动作绑定
    "type": "JSExpression",
    "value": "function(e) { console.log(e.target.innerText) }",
  },
  "onClick2": { // 动作绑定2
    "type": "JSExpression",
    "value": "this.submit",
```

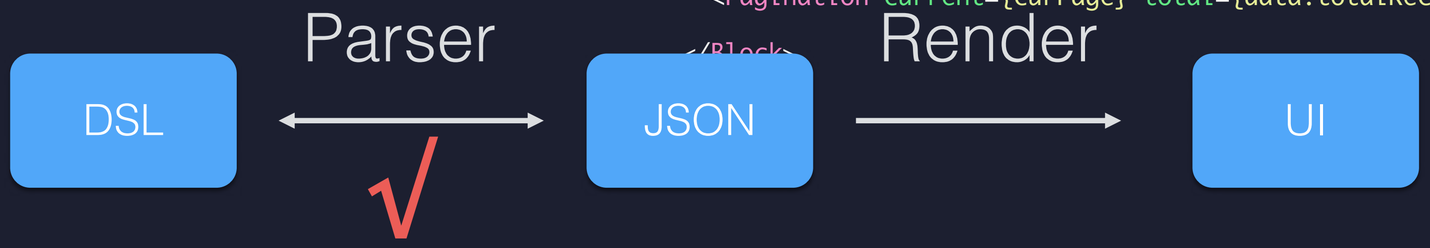
低代码平台的核心 - UI可视化编程

可维护性的解法

DSL - Domain Specific Language

- 可读性
- 代码属性
- 受限 - 双向工程（**Round-trip engineering**）

```
<Panel title="采购合同列表">
  <InlineView marginY="12">
    <Action confirm="确定要删除选中的数据吗？" data={{ids: selectedRowIds}}>
      <Button>
        批量删除
      </Button>
    </Action>
  </InlineView>
  <Table dataSource={data.data} primaryKey="purOrderId" hasBorder={false} isZebra={true}>
    <Table.Column title="订单id" dataIndex="purOrderId"/>
    <Table.Column title="备注" dataIndex="purRemark"/>
    <Table.Column title="状态" dataIndex="statusDesc"/>
    <Table.Column title="操作" dataName="record" cell={(value, index, record) => {
    return lastColumn()({ record: record });
  }} width="150"/>
  </Table>
  <Block marginTop="12">
    <Pagination current={curPage} total={data.totalRecord} pageSize="5" />
  </Block>
```



低代码平台的核心 - UI可视化编程

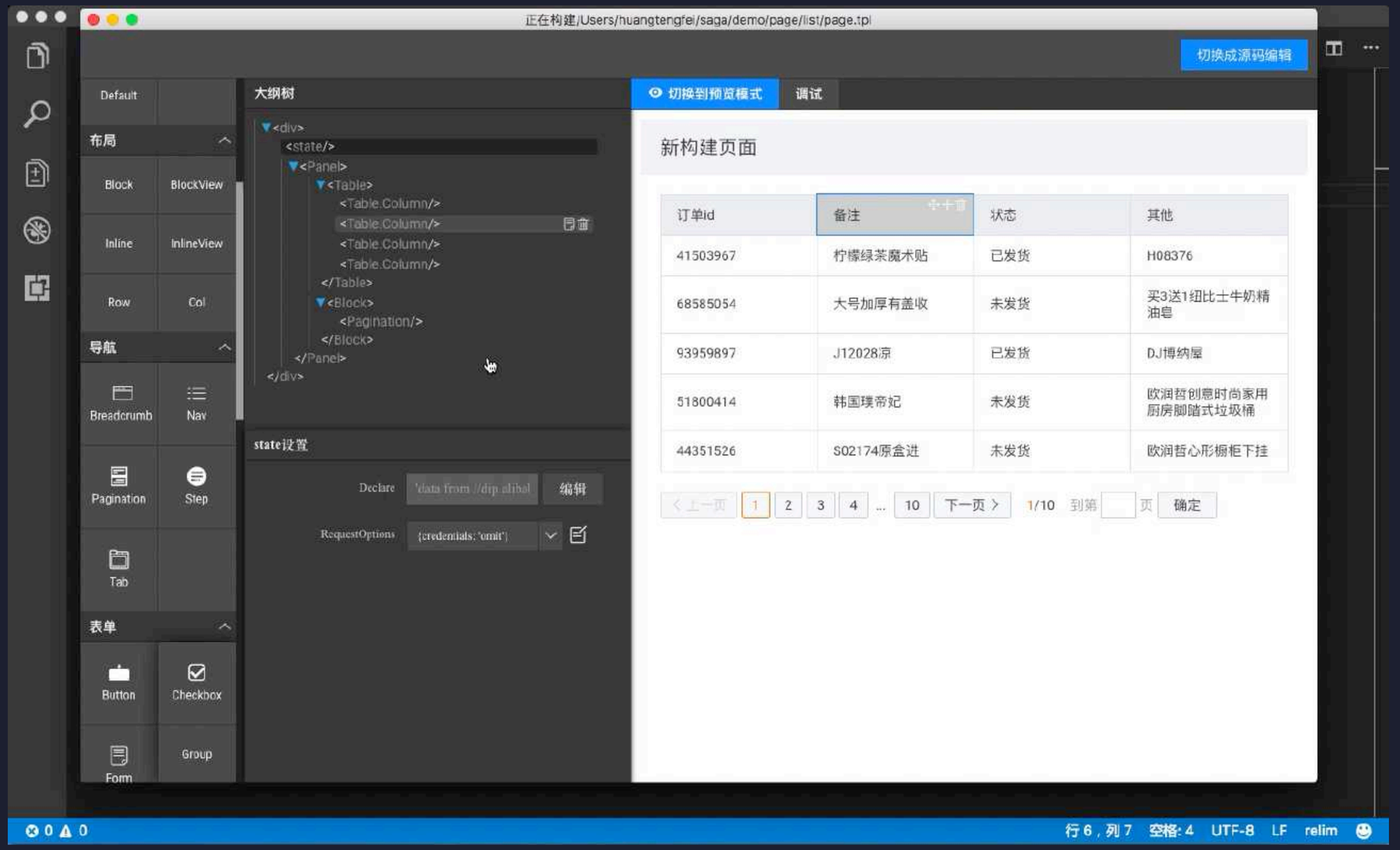
可维护性的解法

DSL - Domain Specific Language

几乎所有的**DSL**都可以通过可视化方式来构建

如果要实现可视化的双向互转，必须基于特定的**DSL**来实现

低代码平台的核心 - UI可视化编程



理想是美好的，现实是骨感的

低代码平台的核心 - UI可视化编程

DSL是否是完美的？

- 学习成本
- 用户心智 - 人群的差异性

你们快来用啊！　　你们自己在用吗？

我只想用React！

推广起来成本很高

低代码平台的核心 - UI可视化编程

我们需要重新思考UI可视化编程的中间产物

JSON

HTMLParser

```
{
  type: "div",
  attributes: {}
  children: []
}
```

VS

AST

<Table />

Babel

```
  "end": 25,
  "loc": {},
  ...
},
"closingElement": {
  "type": "JSXClosingElement",
  "start": 88,
  "end": 96,
  "loc": {
    "start": {
      "line": 3,
      "column": 10
    },
    "end": {
      "line": 3,
      "column": 18
    }
  },
  "name": {
    "type": "JSXIdentifier",
    "start": 90,
    "end": 95,
    "loc": {
      "start": {
        "line": 3,
        "column": 12
      },
      "end": {
```

低代码平台的核心 - UI可视化编程

所以我们只要将原先的可视化对**JSON**的映射

转换成对**AST**的映射就能实现**React**的中后台可视化研发了？

低代码平台的核心 - UI可视化编程

基于ESTree Spec AST的UI可视化编程

难点一

如何关联**DOM** 和 **AST**?

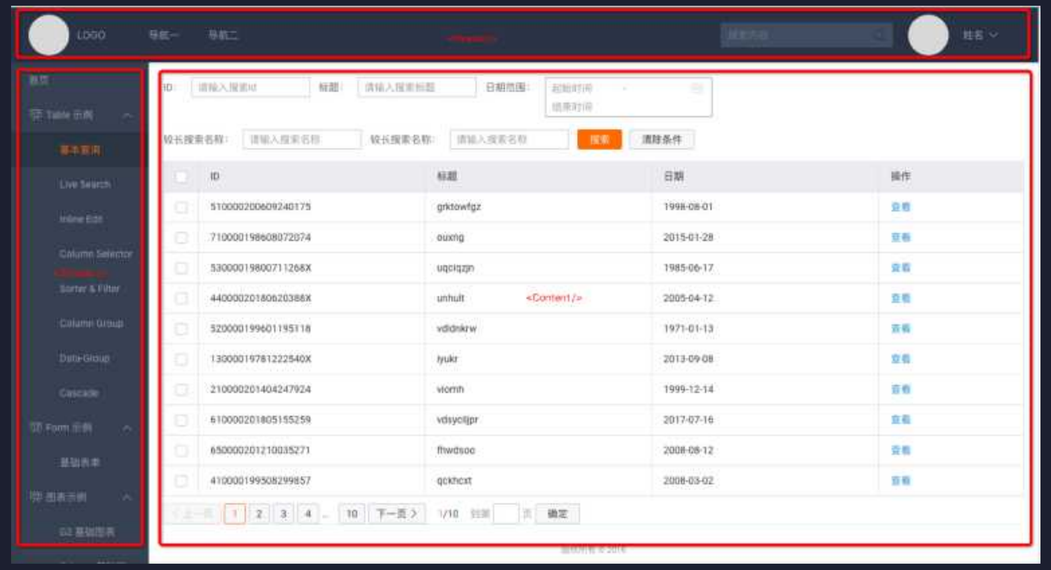
```
traverse(ast, {
  JSXElement(path /* , data */) {
    const { node } = path;
    const { openingElement } = node;

    // 插入 data-hippo data-id
    let { uuid } = node;
    const { attributes } = openingElement;
    const componentName = getNodeName(openingElement);

    if (!uuid) {
      uuid = `U_${componentName}_${Math.random()
        .toFixed(5)
        .replace('0.', '')}`;
      node.uuid = uuid;
      fillTime(attributes, SLOT.id, uuid);
      fillTime(attributes, SLOT.hippo, '');
    }
  }
});
```

低代码平台的核心 - UI可视化编程

基于ESTree Spec AST的UI可视化编程

难点二

源码与可视化不对等的问题？



<Header/>
<Sidebar/>
<Content/>

```
constructor(props) {
    super(props);
    const selectedKey = menuMap[props.location.pathname];
    this.state = {
        selectedKey: selectedKey,
        openMenuKey: getOpenMenu(sideMenus, selectedKey)
    };
}

componentDidMount() {
    this.setSelectedMenu(this.props);
}

componentWillReceiveProps(nextProps) {
    this.setSelectedMenu(nextProps);
}

goto(isLink, link) {
    if (isLink) {
        hashHistory.push(link);
    }
}

setSelectedMenu(props) {
    const selectedKey = menuMap[props.location.pathname];
    const openMenuKey = getOpenMenu(sideMenus, selectedKey);
    this.setState({
        selectedKey: selectedKey,
        openMenuKey: openMenuKey
    });

}

handleClick(openKeys) {
    this.setState({
        openMenuKey: openKeys
    });
}
render() {
    const { selectedKey, openMenuKey } = this.state;
    const headMenuItems = headMenus.map(m => {
```

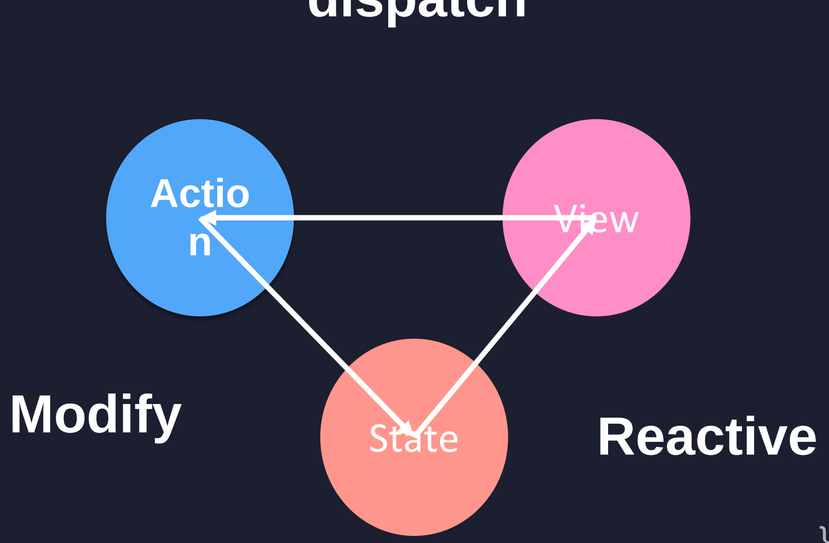
低代码平台的核心 - UI可视化编程

基于ESTree Spec AST的UI可视化编程

难点二

源码与可视化不对等的问题？

UI与逻辑分离




```
export default ({
state: {
  count: 0,
  list: [],
},
actions: {
  increment(state) {
    const { count } = state;
    count++;
  },
  async fetch(state) {
    const ret = await request('/list');
    this.transaction(() => {
      state.data = ret.data;
      state.total = ret.total;
    });
  },
},
});
```

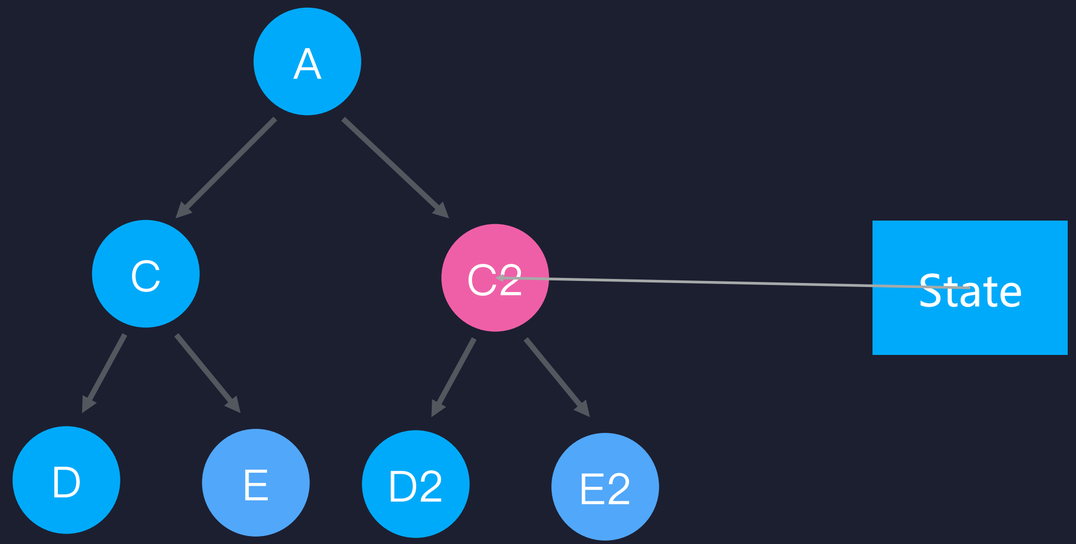
```
@connect(state => state);
class App extends React.Component {
  fetch = () => {
    this.props.dispatch('fetch');
  }
  render() {
    return <span onClick={this.fetch}>
              {this.props.state.count}
              </span>
  }
}
```

SA架构额外带来的好处— 高性能

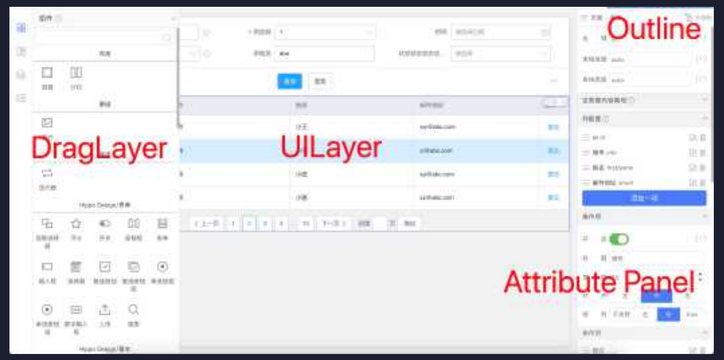
1. 依赖追踪保证只更新对有依赖数据变更的更新
2. 一个事务周期内保证多次状态变更只会有一次更新
3. 子组件无状态更新的时候不渲染

```
export default {
  state: {
    count: 0,
    list: [],
  },
  actions: {
    increment(state) {
      const { count } = state;
      count++;
    },
    async fetch(state) {
      const ret = await request('/list');
      this.transaction(() => {
        state.data = ret.data;
        state.total = ret.total;
      });
    }
```

```
@connect(state => state);
class App extends React.Component {
  render() {
    return <span>{this.props.count}</span>
  }
}
```



低代码平台的核心 - AST UI可视化的整体流程



Drag Layer

拖拽

拖拽

UI Layer

UI

Outline

生成

Mock data

渲染

React

转换

解析

AST

写入

收集

收集

Visual Engine

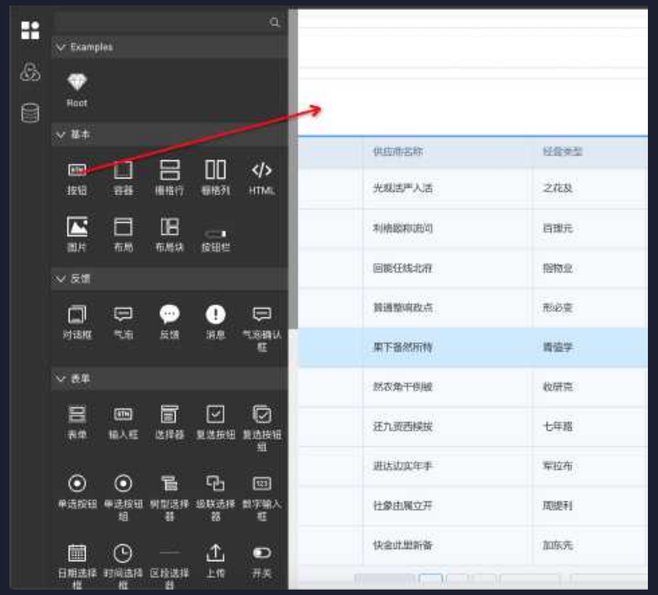
收集

Attribute Panel

低代码平台的核心 - UI可视化编程

基于AST的UI可视化编程本质

把人的意图变成机器可以识别的代码



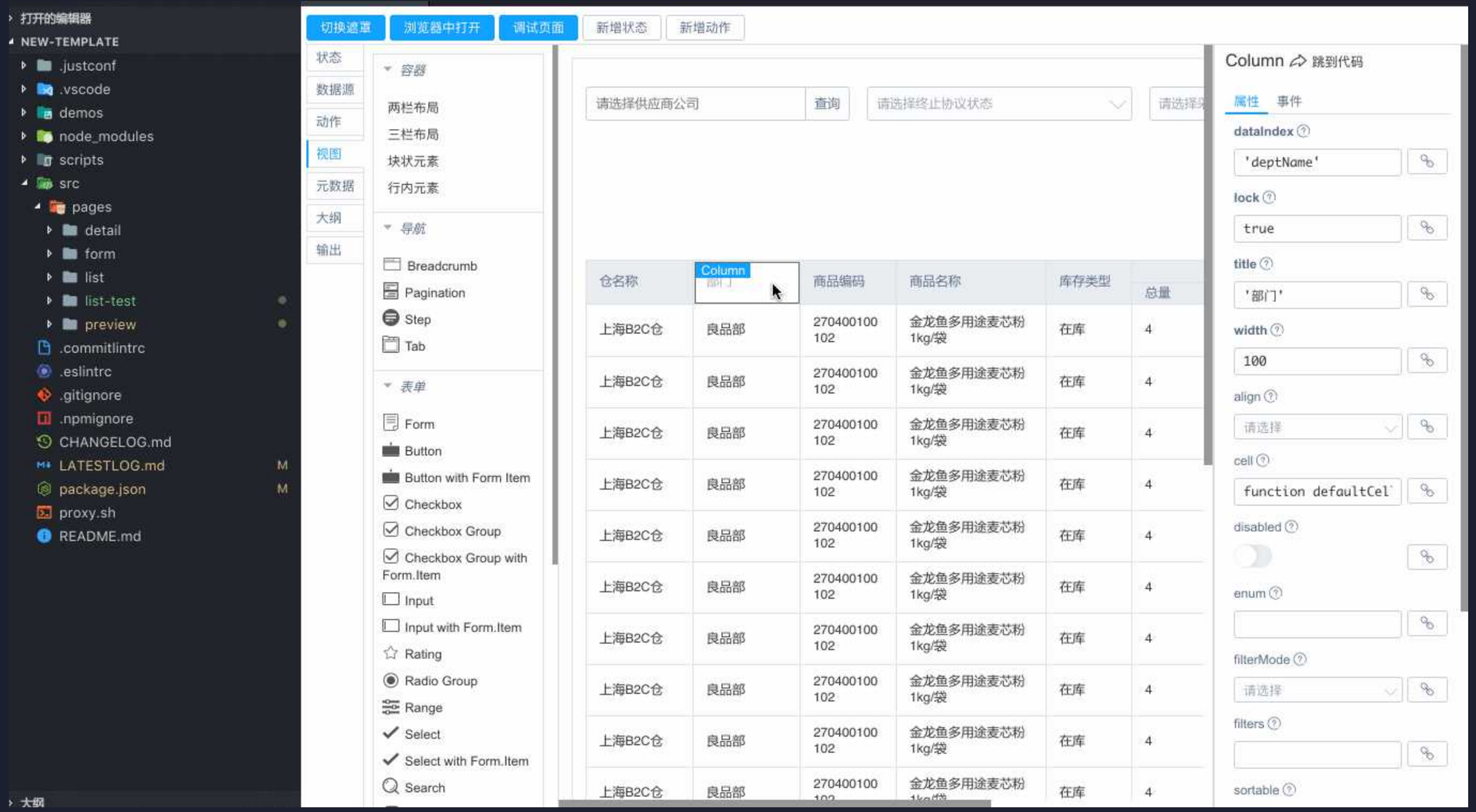
1. 判断Button这个变量是否在View里面导入过
2. 导入过，则返回
3. 没有导入，则导入Button
4. 导入Button，判断包含Button的大包是否存在
5. 有，使用之前的大包
6. 没有，重新写入
7. 最后写入Button的React Element

```
import { Button } from ‘antd';

<Button></Button>
```

```
import { Button, Table } from ‘antd';
```

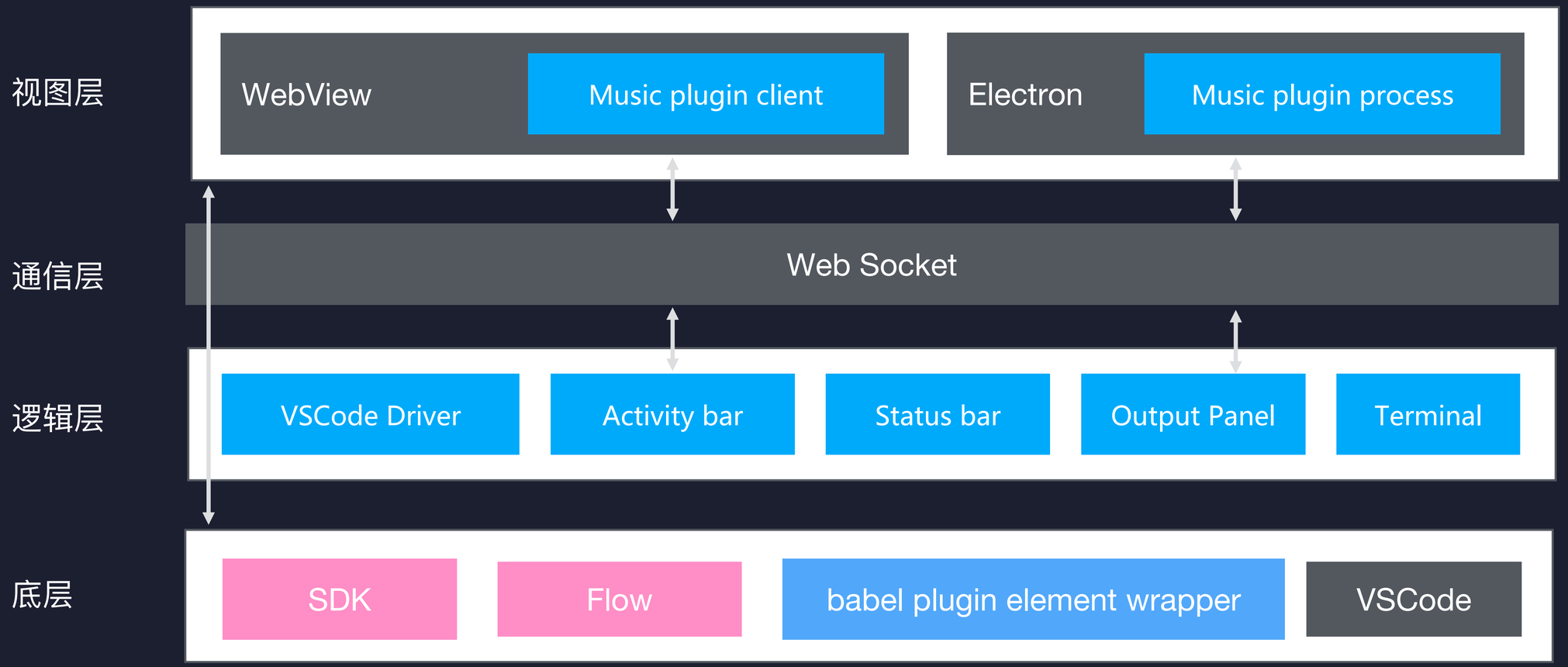
低代码平台的核心 - UI可视化编程

基于AST的UI可视化编程 - 实现 LowCode 和 ProCode 的双向互通



低代码平台的核心 - UI可视化编程

VSCode线下工具架构图



外部依赖　共建　自研

低代码平台的核心 - UI可视化编程

性能是线下方案的阿克琉斯之踵

- webpack 构建性能
- websocket 通信耗时
- 磁盘IO耗时
- webpack HMR 通信耗时

无论如何优化总是很难达到100ms以内

低代码平台的核心 - UI可视化编程

性能是线下方案的阿克琉斯之踵

把流程搬到线上去

webpack能在浏览器端运行吗？

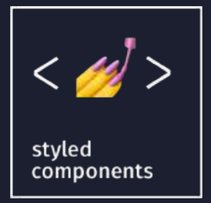
答案是肯定的

低代码平台的核心 - UI可视化编程

UI可视化编译过程线上化

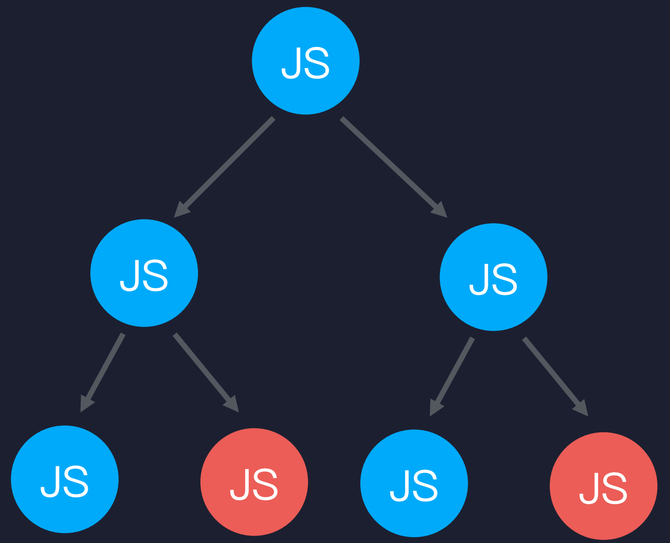
webpack 的本质

modular compile + modular bundle + modular JIT



Sea.js

sea.js 模块构建器

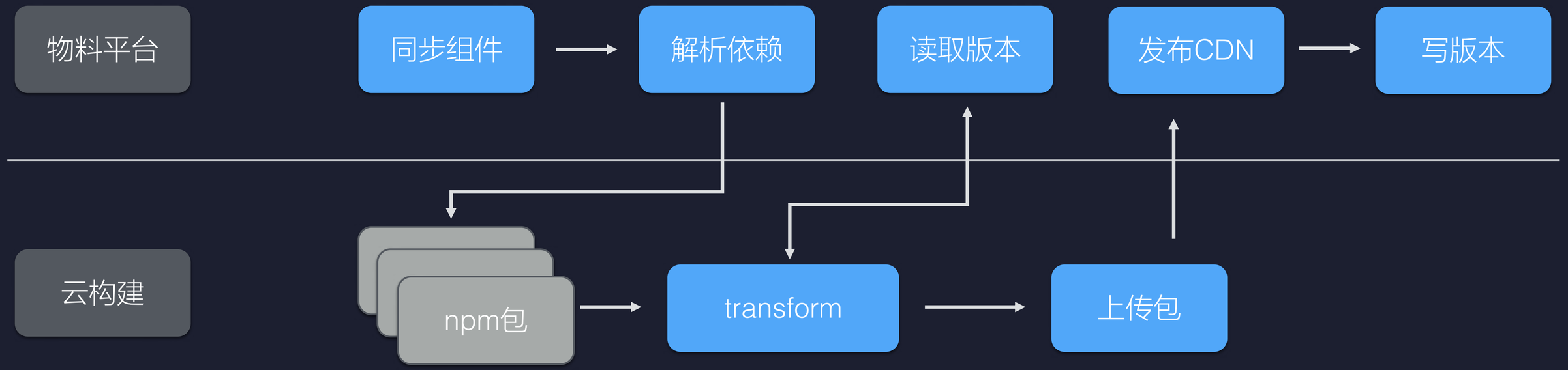


低代码平台的核心 - UI可视化编程

UI可视化编译过程线上化的额外挑战

挑战一：npm生态的问题解决

```
define("@alife/legion/1.12.2/lib/basement/Logger", ["../core/base"], function(require, exports, module) {
   // your code here
})
```



低代码平台的核心 - UI可视化编程

UI可视化编译过程线上化的额外挑战

挑战二：**Seajs下的HMR**

```
seajs.refresh = (url, entry,  callback) => {
  delete seajs.cache[entry];
  delete seajs.data.fetchedList[entry];

  const cache = seajs.cache;
  Object.keys(cache).forEach(key => {
    const module = cache[key];
    const deps = module.dependencies.map(dep => {
      return seajs.resolve(dep, key);
    });
    if (deps.indexOf(item) !== -1) {
      delete seajs.cache[key];
      delete seajs.data.fetchedList[key];
    }
  });

  delete seajs.cache[item];
  delete seajs.data.fetchedList[item];
});

seajs.use([entry], () => {
```

低代码平台的核心 - UI可视化编程

UI可视化编译过程线上化的额外挑战

挑战三：React diff的问题

相同的组件类型才会diff，由于代码每次都进行了HMR
所以每次都是新的组件，页面会重新Render

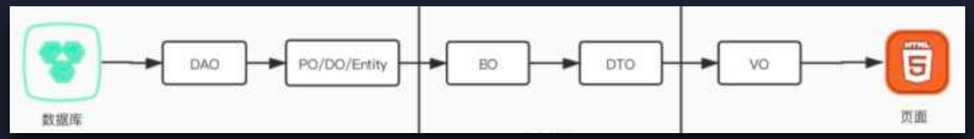
```
ProxyRegister('/src/example/view.jsx#FormItem', FormItem);
ProxyRegister('/src/example/view.jsx#FormSubmit', FormSubmit);
ProxyRegister('/src/example/view.jsx#FormReset', FormReset);
ProxyRegister('/src/example/view.jsx#FormFieldSet', FormFieldSet);
ProxyRegister('/src/example/view.jsx#Dialog', Dialog);
ProxyRegister('/src/example/view.jsx#SearchForm', SearchForm);
ProxyRegister('/src/example/view.jsx#BasicNormalList', BasicNormalList);
```

```
import React from 'react';
import { createProxy } from 'react-proxy';

const PROXIES = {};
const UNIQUE_ID = '__uniqueId';

export default function proxyRegister(id, type) {
  if (typeof type === 'function' && !type[UNIQUE_ID] && type.prototype) {
    Object.defineProperty(type, UNIQUE_ID, {
      value: id
    });
    let proxy = PROXIES[id];
    if (proxy) {
      proxy.update(type);
    } else {
      PROXIES[id] = createProxy(type);
    }
  }
}

const oldCreateElement = React.createElement;

React.createElement = function createElement(type, ...args) {
  if (type && type[UNIQUE_ID]) {
    type = PROXIES[type[UNIQUE_ID]].get();
```

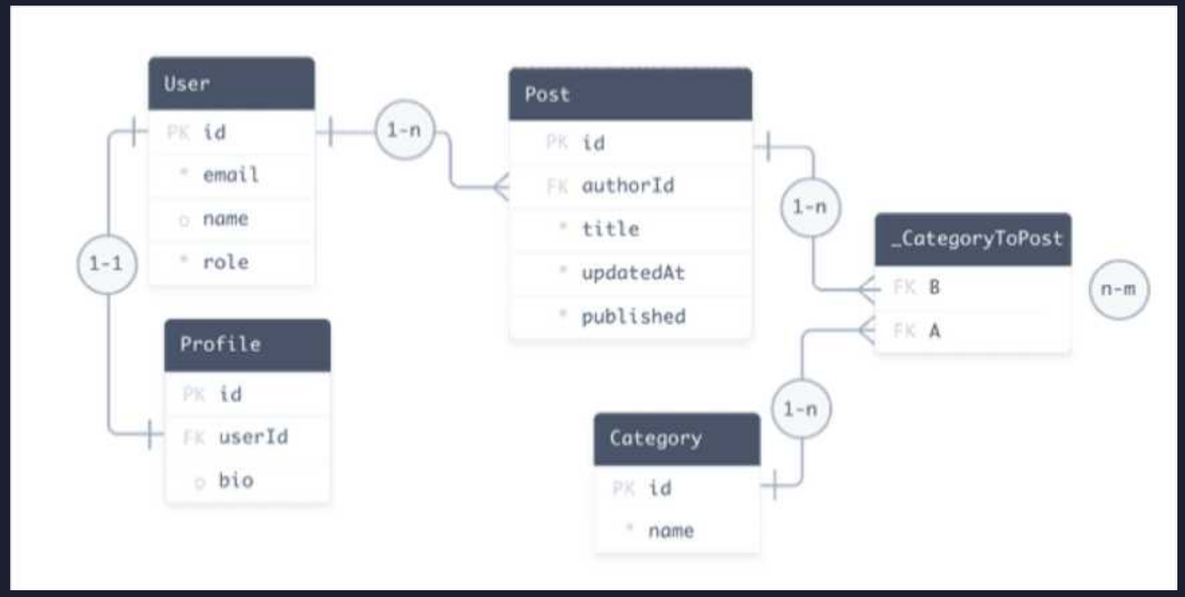
低代码平台的核心 - 模型驱动架构

UI可视化只专注在前端侧解决问题，如何加速整个中后台应用的研发速度呢？

它是一种基于UML以及其他工业标准的框架，
支持软件设计和模型的可视化、存储和交换

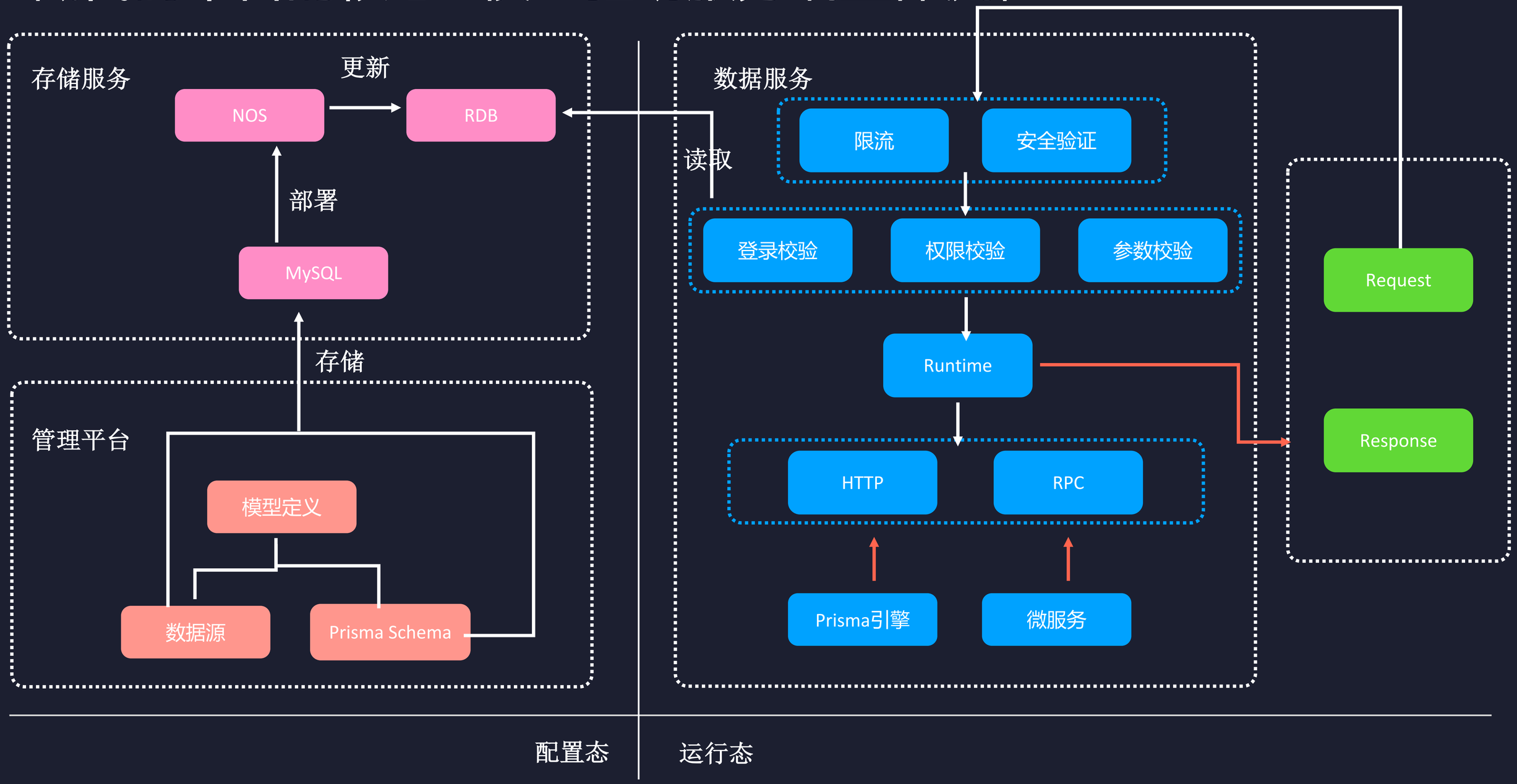


实际上模型并不等于实体，但是在这里简化了其概念



实体关系图

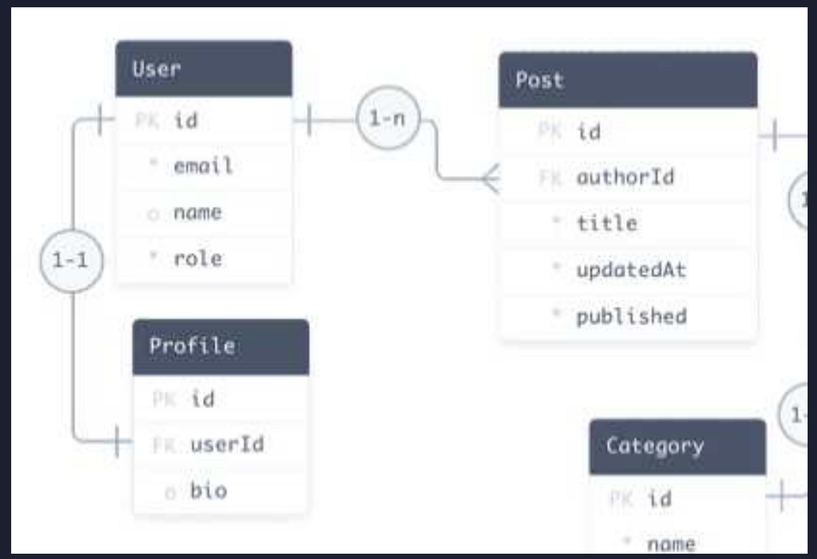
低代码平台的核心 - 模型驱动服务端整体流程



低代码平台的核心 - 模型驱动 - 数据建模

普通多租户

物理模型

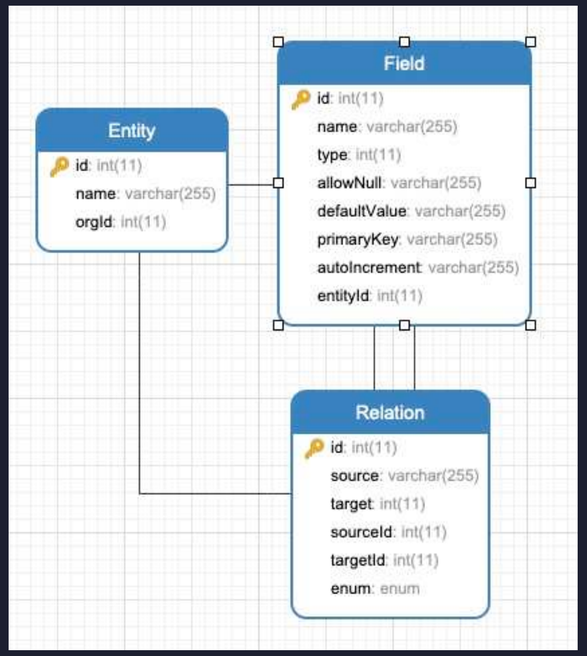


普通多租户的问题

1. 用户具有执行**DDL**权限，模型爆炸导致的冲突，模型升级障碍
2. 由于都是物理模型，没法做有效的隔离，平台升级是个问题

元数据多租户

为租户分配一个标志位，比如**orgId**



元数据驱动的优势

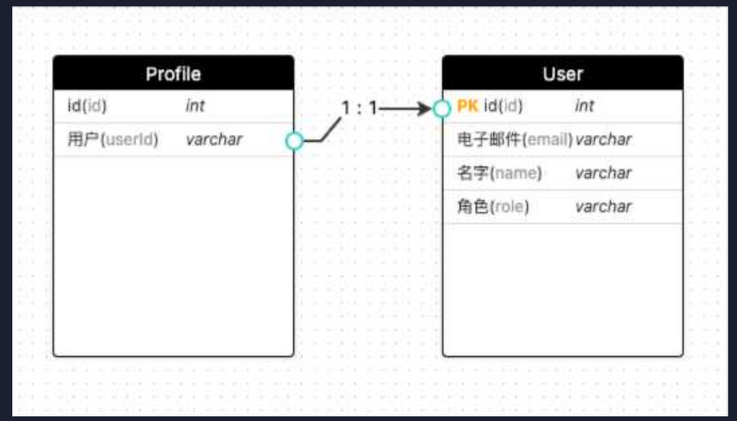
用户客制化变得容易，用户定义的模型和平台模型
通过**orgId**有效的隔离

低代码平台的核心 - 模型驱动 - 代码生成

RTE（双向工程）
语言无关性

模型关系 ⟷ **Single Source of truth**

Server Code

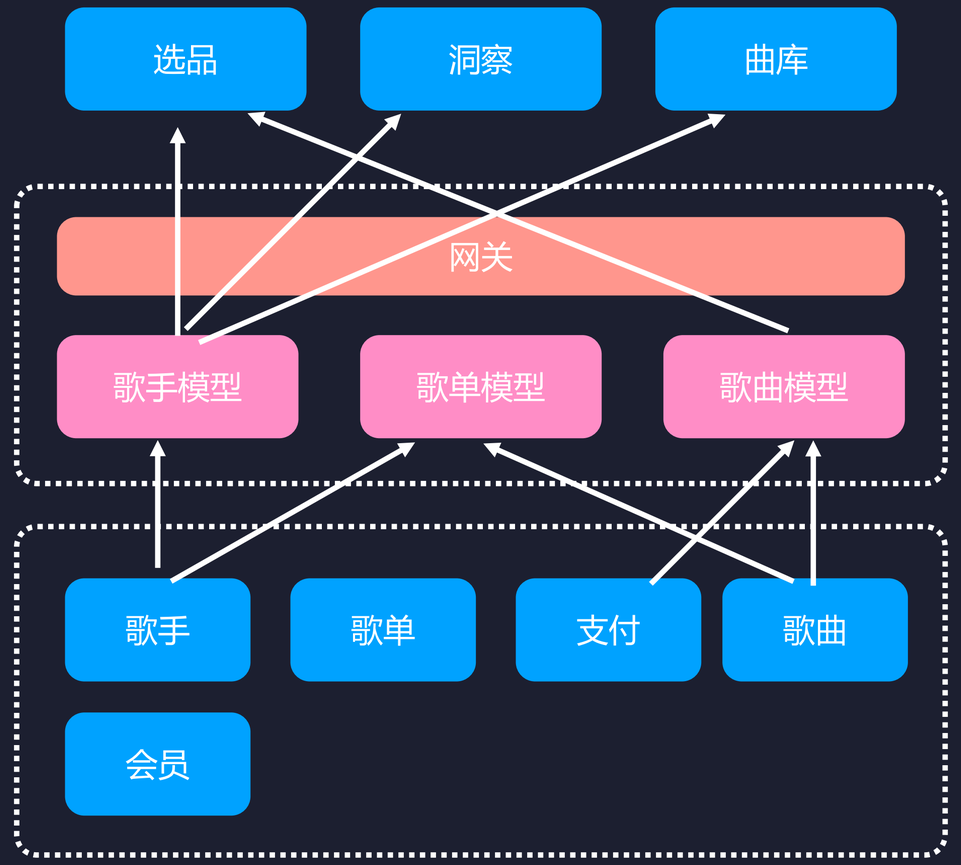



```prisma
model User {
  id      Int      @id @default(autoincrement())
  email   String   @unique
  name    String?
  role    Role     @default(USER)
  posts   Post[]
  profile Profile?
}

model Profile {
  id      Int     @id @default(autoincrement())
  bio     String
  user    User    @relation(fields: [userId], references: [id])
  userId  Int
}
```

```
  return await prisma.user.findUnique({
    where: {
      id
    }
  })
}
find(options) {
  return await prisma.user.findMany(options)
}
create(data) {
  return await prisma.user.create({
    data,
  })
}
remove(id) {
  return await prisma.user.delete({
    where: id
  })
}
update(data) {
  return await prisma.user.update({
    data,
    where: {
```

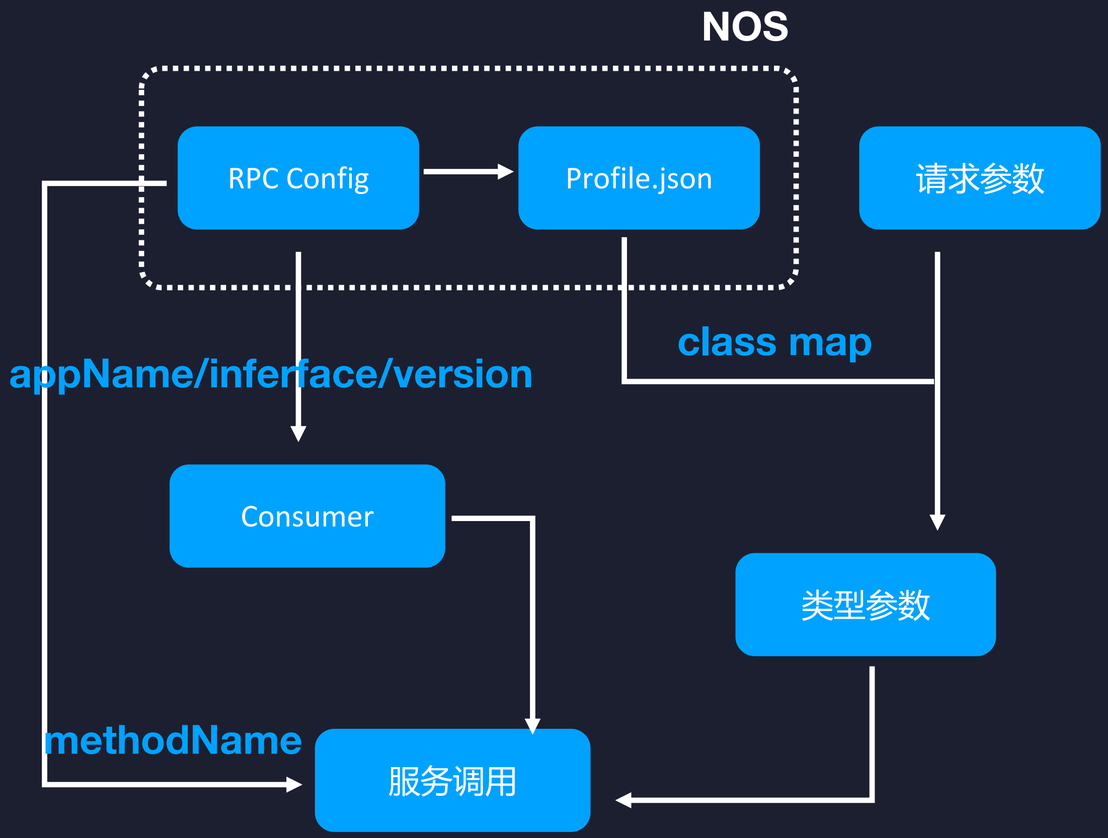
低代码平台的核心 - 模型驱动 - RPC泛化调用

大多数情况下模型并不等于数据库的实体，更多是**DTO**的聚合体

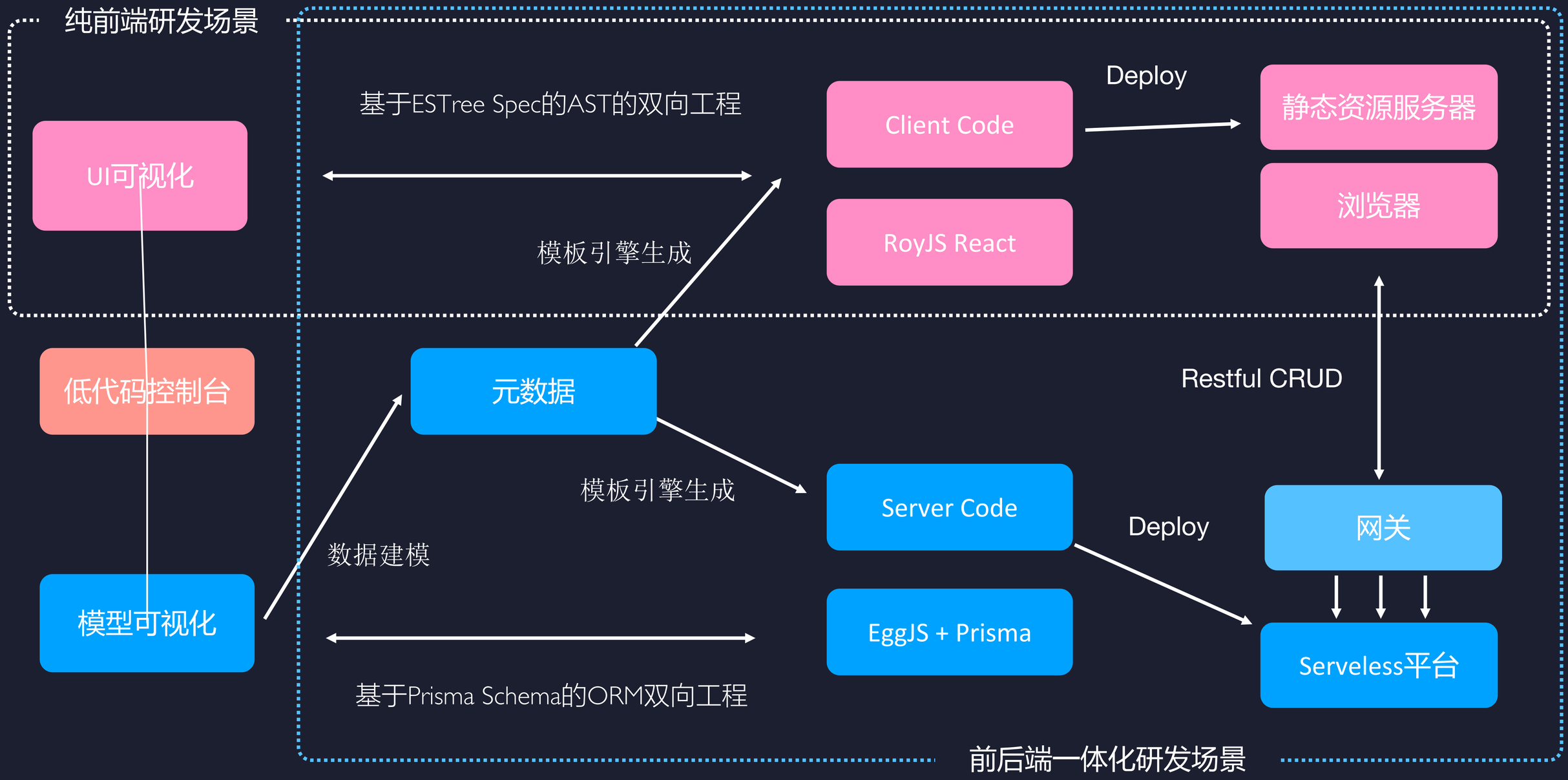


模型

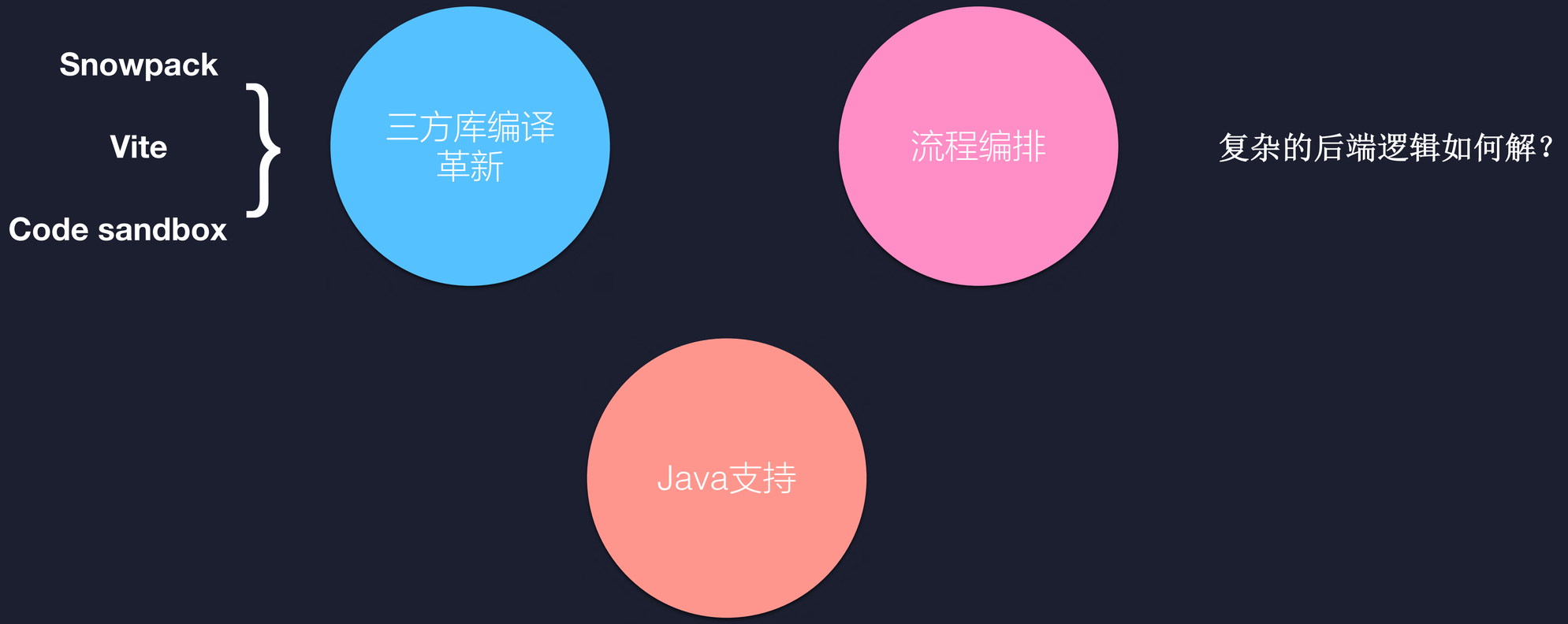
RPC数据源



低代码平台 - Put All Together



云音乐低代码平台后续规划



最后一句话

一定程度上决定技术产品的应用范围的主要是运营

THANKS

QCon+ 案例研习社