丰羽计划

—— 研发前端编码基本功训练赋能

讲师：陈锐

第一天回顾

模块拆分与封装

- 不知道怎么拆封模块？模块拆封不合理？
  - 模块分层上层无需关注下层具体实现
  - 模块职责确定，满足职责单一性
- 场景分析和功能设计如何去做？
  - 一定是先有使用场景，然后才会有相应的功能
  - 可以根据已有功能，推衍出其他使用场景

APi 设计

- 怎么设计 API ?
  - 提升复用能力和扩展能力
  - 降低使用复杂度
  - 屏蔽内部细节，但提供足够的灵活性
  - 防止用户误用，破坏设计者的约束

逻辑解耦与代码健壮

逻辑解耦与代码健壮

解耦定义

将纠缠在一起的若干逻辑分开，让它们不再相互影响，或者影响减小，改善设计可测试性、可维护性、可扩展性

目标

通过机制策略分离提升代码可测试性、可调试性以及可维护性

解耦

> 模块之间，函数之间，经常会出现各种的“藕断丝连”。在定义函数时，需要利用解耦技巧去优化函数的定义，保证每个函数都能做到“高内聚低耦合”。

解耦可以分为三个步骤：

- 识别易变流程和不变流程；
- 分离“易变”与“不变”流程；
- 分离出“制程”阶段，利用“流程描述”定义生产流程，提高生产流程的灵活性；

识别易变与不变

所谓"易变"流程，就是在软件生命周期中，需求/实现容易发生变化，常常需要程序员修改的那部分流程。

所谓"不变"流程，就是在软件生命周期中，一般不会发生变化的流程。

易变：依赖效果不确定的技术方案、依赖不确定的需求，需要不断优化、改变的部分

不变：与具体业务无关

识别易变与不变

这个案例是一个虚构的小工具 batch。batch 的功能是取得某个目录下的文件列表，根据文件的类型和不同命令选项，对每一个文件进行不同处理。

```
function batch(dir, action) {
  fs.readdirSync(dir).forEach((name) => {
    const filePath = path.join(dir, name);
    const extName = path.extname(filePath).slice(1);
    switch (action) {
      case 'show':
        if (['md', 'txt', 'log'].includes(extName)) {
          exec(`cat ${filePath}`);
        }
        break;
      case 'exec':
        if (['exe', 'bat', 'sh'].includes(extName)) {
          exec(filePath);
        }
        break;
      default:
        throw new Error(`${action} is not supported`);
    }
  });
}
```

分离易变与不变

将“易变”和“不变”流程分别实现为独立模块，函数

```
function showCmd(filePath, fileType) {
  const supportedFileType = ['md', 'txt', 'log'];
  if (supportedFileType.includes(fileType)) {
    exec(`cat ${filePath}`);
  }
}

function execCmd(filePath, fileType) {
  const supportedFileType = ['exe', 'bat', 'sh'];
  if (supportedFileType.includes(fileType)) {
    exec(filePath);
  }
}

function getFileType(filePath) {
  return path.extname(filePath).slice(1);
}
```

```
function batch(dir, action) {
  fs.readdirSync(dir).forEach((name) => {
    const filePath = path.join(dir, name);
    const fileType = getFileType(filePath);
    switch (action) {
      case 'show':
        showCmd(filePath, fileType);
        break;
      case 'exec':
        execCmd(filePath, fileType);
        break;
      default:
        throw new Error(`${action} is not supported`);
    }
  });
}
```

制程与生产

定义：

进一步提升“不变”流程的灵活性，令其可以适应不同场景，不同需求，可以将整个代码流程划分为两个阶段，借用自动化生产线的概念就是“制程”阶段和“生产”阶段

制程阶段：确定生产阶段的各种参数等

生产阶段：根据上述参数加工

目的：

让“易变”步骤由变量定义，做到可运行时改变

在原有流程中分化出一个独立的“制程”阶段，并让“制程”阶段可配置化，流程中的“易变”步骤就可以通过配置进行定义，从而拥有了应对变化的能力

制程与生产

```
// 制程阶段
const actionMap = {
  txt: { show: showCmd },
  log: { show: showCmd },
  sh: { exec: execCmd },
  exe: { exec: execCmd },
  // ...
};

function showCmd(filePath) {
  exec(`cat ${filePath}`);
}

function execCmd(filePath) {
  exec(filePath);
}

function getFileType(filePath) {
  return path.extname(filePath).slice(1);
}
```

```
// 生产阶段
function batch(dir, action) {
  fs.readdirSync(dir).forEach((name) => {
    const filePath = path.join(dir, name);
    const fileType = getFileType(filePath);
    if (actionMap[fileType]?.[action]) {
      actionMap[fileType][action](filePath);
    } else {
      throw new Error(`${action} is not supported`);
    }
  });
}
```

layout: fact

异常处理

契约式编程

指在函数调用者和实现者之间制订约定，规定调用者和实现者各自必须遵守的一些约束，以简化函数之间协作的异常处理逻辑

关注的是对输入参数和输出结果的合法性进行约束和检查。

```
function divide(a, b) {
  // 契约式编程：检查输入参数的合法性
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new Error('参数必须是数字');
  }

  // 执行除法操作
  return a / b;
}

console.log(divide(10, '2')); // 抛出异常：参数必须是数字
```

进攻式编程

它强调在代码中预测并处理错误，而不是简单地依赖外部代码或调用者来保证输入的正确性。

进攻式编程的目标是在可能出现错误的地方主动采取措施，以确保代码的正常执行，并提供适当的错误处理和错误消息。

```
function getUserById(id) {
  // 进攻式编程：主动处理可能出现的错误情况
  if (!id) {
    console.error('未提供用户ID');
    return null;
  }

  // 模拟从数据库中获取用户数据
  const user = { id: id, name: 'John Doe' };

  return user;
}

const user = getUserById(); // 进攻式编程：未提供用户ID
console.log(user); // 输出：null
```

防御式编程

防御式编程关注的是通过确保代码能够适应不正常或意外的情况，防止错误传播和程序崩溃。

它强调在代码中进行边界检查、错误处理和异常处理，以应对潜在的问题和异常情况。

防御式编程的目标是通过加入适当的错误处理机制，保护代码免受非预期输入或外部条件的影响，以提高代码的可靠性和容错性。

```
function fetchData(url) {
  // 防御式编程：确保处理可能出现的错误情况
  if (typeof url !== 'string' || url.trim() === '') {
    console.error('无效的URL');
    return Promise.reject(new Error('无效的URL'));
  }

  // 执行网络请求并返回 Promise 对象
  return fetch(url)
    .then((response) => response.json())
    .catch((error) => {
      console.error('发生了一个错误：', error);
      throw error;
    });
}
```

```
// 防御式编程：无效的URL
fetchData(123)
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.log('请求失败：', error.message);
  });
```

日志处理

【目标】

能再现程序执行结果

【日志五元组】

“时间、模块、对象、事件、结果”

【日志级别】

- 实时调试日志（trace）——重要数据状态被修改前后、关键函数调用，帮助研发了解系统内部工作状态细节，用于定位系统故障（五元组 + 关键数据）
- 调试日志（debug）——异常信息，帮助研发了解系统关键活动，用于排除故障
- 信息日志（info）——告知用户系统发生过的重要事件，帮助用户了解系统工作状态
- 告警日志（warn）——系统可能处于不正常状态，需要告知用户介入（五元组 + 建议）
- 错误日志（error）—— 需要管理员的干预，不干预会出现一段时间内的功能不正常（五元组 + 建议）

日志处理

```
// 引入日志库
import { Logger } from 'your-logging-library';

// 创建日志实例
const logger = new Logger('WebSocket');

class WebSocketClient {
  constructor(url) {
    this.url = url;
    this.websocket = null;
  }

  send(message) {
    logger.debug('发送WebSocket消息:', message);
    this.websocket.send(message);
  }

  close() {
    logger.info('关闭WebSocket连接');
    this.websocket.close();
  }
}
```

```
  connect() {
    logger.info('连接到WebSocket:', this.url);
    this.websocket = new WebSocket(this.url);

    this.websocket.onopen = () => {
      logger.info('WebSocket连接已打开');
    };

    this.websocket.onmessage = (event) => {
      logger.debug('收到WebSocket消息:', event.data);
      // 处理收到的消息
    };

    this.websocket.onclose = (event) => {
      logger.warn('WebSocket连接已关闭:', event.code, event.
    };

    this.websocket.onerror = (error) => {
      logger.error('WebSocket错误:', error);
    };
  }
```

日志处理

在上述例子中，我们使用了一个名为 Logger 的日志库，并创建了一个名为 WebSocket 的日志实例。它可以根据需要记录不同级别的日志。

- `logger.info`：用于记录重要操作的信息，如连接到 WebSocket 服务器。
- `logger.debug`：用于记录调试信息，如收到的 WebSocket 消息。
- `logger.warn`：用于记录警告信息，如 WebSocket 连接关闭。
- `logger.error`：用于记录错误信息，如 WebSocket 错误。

关于封装

封装

我们无法预知代码的改动，但可以编写方便后续维护的代码，如何从维护者的角度衡量”易于维护“的代码呢？

在过去很长一段时间内，我都认为：只要改动的地方少，代码就“易于维护”。

- 减少变量的重复，通过配置文件管理全局变量
- 减少代码的重复，封装函数、封装模块
- 减少逻辑的重复，封装组件

减少改动最好的办法就是将统一的逻辑封装起来，封装的核心概念是将系统中经常变化的部分和稳定的部分隔离

按照设想，封装的作用

- 将相同功能逻辑的代码块从物理结构上限制在一起，方便查找，这样后续的改动修改的代码就会变少
- “每一个优雅的接口后面都有一个肮脏的实现”，维护者不需要关心肮脏的实现，也能写出好代码
- 封装能够减少全局变量和自由变量的使用，更容易测试

强行封装

现在需要封装一个商品组件，我们有两种思路

- 一种是根据支持传入某些查询条件，组件先查询商品，再进行展示，相当于组件需要负责查询和展示
- 另一种是只接收一个商品参数，由调用方自己查询并传入商品，相当于组件只负责展示

为了代码复用，很多人大概率会使用第一种方式，把看起来比较通用的逻辑都给封装起来。而在某些需要直接传入商品的场景下，就得再暴露一个参数，同时判断有这个参数就不再请求查询接口了。

封装并不是从物理位置把代码拆分到不同的函数、类或文件中，而应该是从概念上，定义良好的输入和输出。

对于要封装的代码，我们需要考虑变化的来源，先找到变化，这样才能确认哪些是可以封装在一起的。

破坏封装

封装的本意是将业务中变化的地方进行隔离，将不变的地方给封装起来

这就提供给我们一种可以快速修改代码的能力，只需要修改某一处，就会影响所有依赖，看起来对于添加通用功能很诱人，在快速迭代期间，我们往往受不了这种诱惑。

在这些行为下，我们很可能就会破坏封装原本的意义，将一些其他奇奇怪怪的功能引进来，最后的结果就是封装的逻辑不再通用。

破坏封装

目前有一个纯 UI 组件，它接收一个特定的数据结构 config，然后展示出来就行了。目前有 10 个页面在使用

```
<MyComponent :config="configData" />
```

现在多了一个需求：点击这个组件的时候需要上报数据，我们面临两个选择

- 在每个依赖于该组件的页面注册点击事件，处理上报逻辑
- 改 10 个页面太麻烦了，幸好将他封装成通用组件了，在组件内处理数据上报就行了

破坏封装

假设我们选择了第二种做法，很显然，这次需求太简单了，评估一天的工时，花半个小时搞完，剩下的时间就可以摸鱼了

> 改动:我们在 UI 组件里面添加了数据上报的功能

```
<MyComponent :config="configData" @click="handleComponentClick" />
```

这样这个组件就包含了两个功能：UI 展示和埋点上报；

当那些需要该组件进行展示 UI 时，就静默地带上了数据上报的功能，而这个功能可能并不是使用者希望的。

也许可以再添加一个 prop，比如 needReport 之类的，用来控制使用者是否需要日志上报，显然这不是一个很好的做法。

现在打破了组件的通用性，这个 UI 组件已经变得不再通用了，使用者需要知道这个组件有哪些功能，需要传入哪些参数来控制对应功能。

破坏封装

> 没有什么问题是不能加一层中间件解决的，如果有，那么就再加一层

像这种需要动态添加非组件逻辑相关功能的时候，也许可以使用装饰器来实现，封装一个待日志上报和 UI 展示的高阶组件怎么样？

另一种破坏封装

```
async function getXaasData() {
  const pluginsData = await fetchRemoteConfig();
  const plugins = pluginsData?.data?.plugins || [];

  const list = plugins.map((item) => {
    // ...
  });

  return {
    unControlList: getUnControlList(list),
    controlList: getControlList(list),
  };
}
```

```
async function getXaasData() {
  const pluginsData = await fetchRemoteConfig();
  const plugins = pluginsData?.data?.plugins || [];

  setPluginsInfo(plugins);

  const list = plugins.map((item) => {
    // ...
  });

  return {
    unControlList: getUnControlList(list),
    controlList: getControlList(list),
  };
}
```

另一种破坏封装

```
getValue() {
  const data = this.invokeSelect('value');
  return data.value || '';
},
```

```
/**
 * @param {boolean} strict
 * strict 参数主要是 schema 校验场景下获取真实的值
 * 如果兜底返回空字符串，对 schema 来说对应的字段就是有值的，非空校
 */
getValue(strict) {
  const data = this.invokeSelect('value');
  if (strict) {
    return data.value;
  }
  return data.value || '';
},
```

另一种破坏封装

在使用框架时，如果某个功能实现起来比较麻烦，我们想到的是如何实现这个功能，而不是如何修改底层框架来满足我们的需求。

可同样的情况到了我们项目里面，为什么就想着要去随便去修改已经封装的代码呢?比如随手加个参数，加个 if 判断之类的?

封装的代码是我们自己写的，不像其他框架或库里面的代码有天然隔离（比如项目放在 node_modules 里面），从结构化编程带来的惯性思维可能会导致我们下意识的去修改这些代码，就会导致封装更容易被破坏。

今天的练习

- 完善训前代码，掌握模块拆分与封装的技巧，提升模块的可扩展性、可复用性
- 完善异常处理、日志，提升模块的健壮性
- 补上单元测试
- 优化代码”请重构以下代码.ts“