丰羽计划

—— 研发前端编码基本功训练赋能

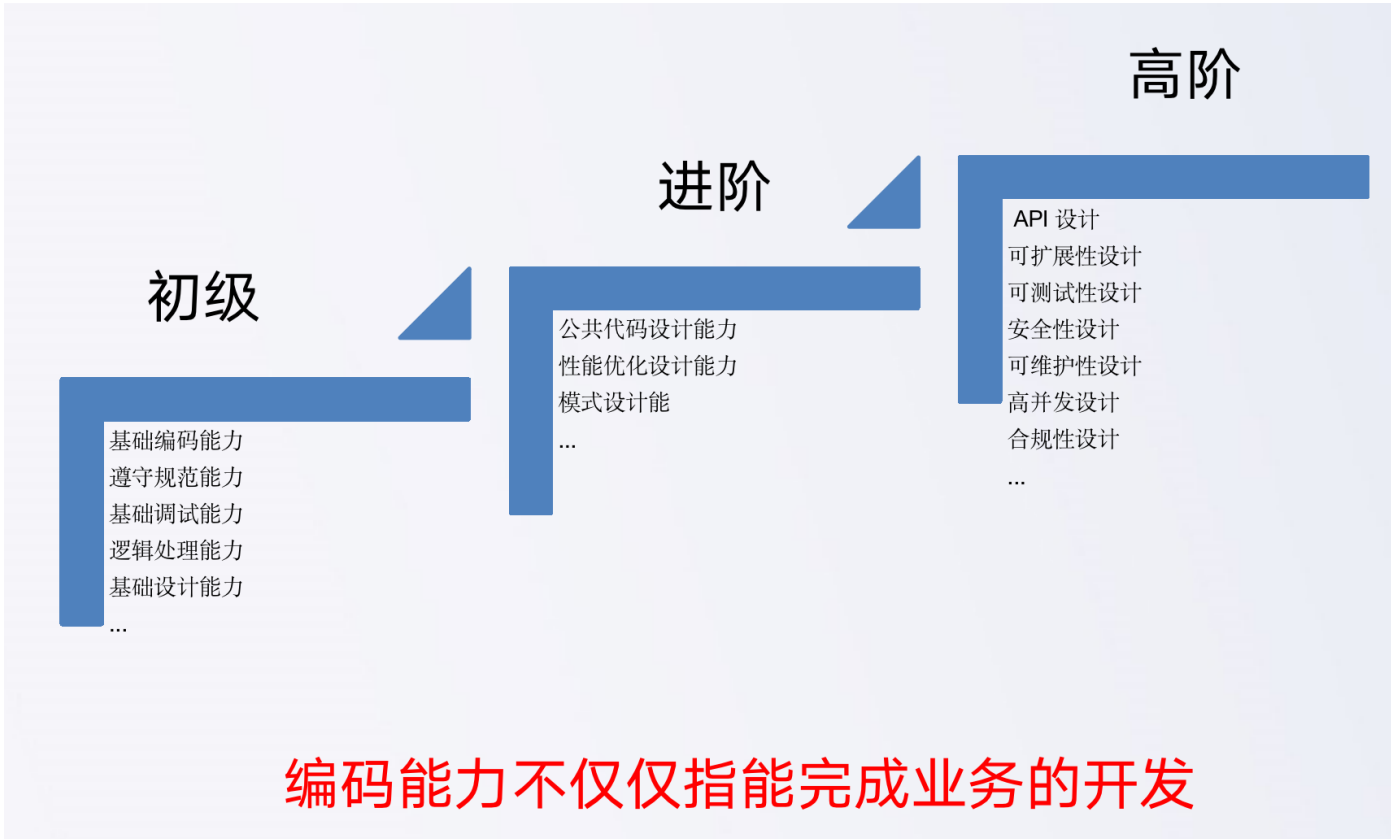
讲师：陈锐

编码是水磨工夫，多琢磨，多练习！

训练的目的

- 通过3天的集训，帮助学员掌握基础编码技巧，提升编码基本功。
- 掌握逻辑解耦、重构、拆分、封装等编码技巧，让你的代码更具备可测试、可调试、可维护、可扩展、可复用。
- 掌握单元测试进而让你的代码更健壮。

什么是编码能力


初级
基础编码能力
遵守规范能力
基础调试能力
逻辑处理能力
基础设计能力
...
进阶
公共代码设计能力
性能优化设计能力
模式设计能
...
高阶
API 设计
可扩展性设计
可测试性设计
安全性设计
可维护性设计
高并发设计
合规性设计
...
编码能力不仅仅指能完成业务的开发

训前回顾

实现一个 mock 数据的库

编写一个 mock 数据的 typescript 库，可以生成各种类型的随机数据，包括但不限于数字、字符串、日期、布尔值、数组、对象等。

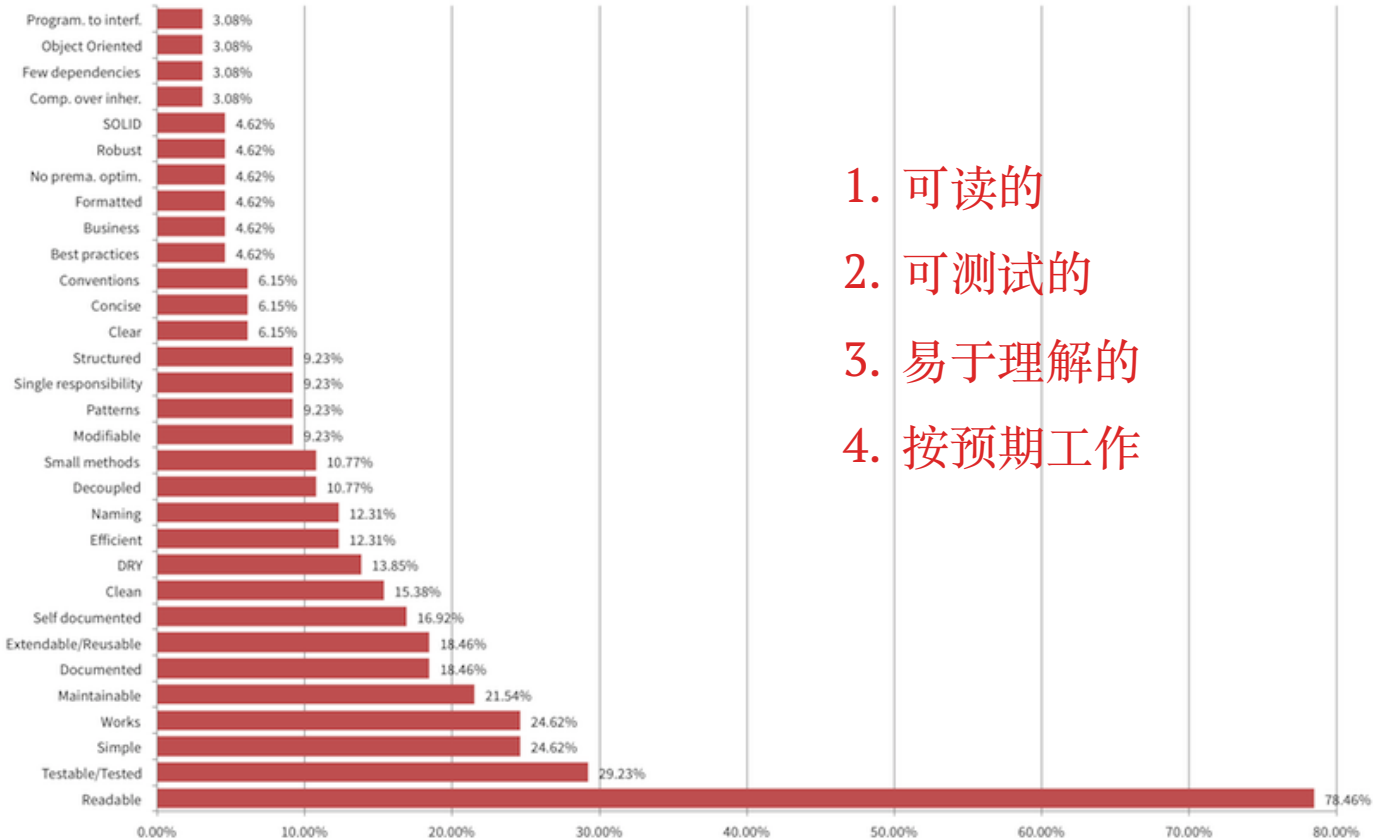
功能需求：

- 支持生成随机字符串、数字、布尔值、日期、时间等基本数据类型。
- 支持生成对象、数组等复杂数据类型。
- 支持自定义数据类型，例如生成身份证号码、手机号码等特定格式的数据。
- 支持生成符合特定规则的数据，例如生成指定范围内的数字、指定长度的字符串、指定格式的日期等。
- 支持根据数据模板生成数据，例如根据一个 JSON 模板生成符合该模板的数据。
- 支持生成大量数据，例如生成 1000 条数据。

典型问题

- 基础功能覆盖不全
- 缺少场景分析，扩展性不强
- API 设计不合理
- 模块分层不合理，代码耦合严重，职责不清
- 缺少异常处理
- 单测不会写，缺少测试
- checklist，编码习惯问题

什么是好代码



1. 可读的
2. 可测试的
3. 易于理解的
4. 按预期工作

什么是好代码

- 可用性：代码要有用，能解决问题

> 不能解决问题，就不会有人要，那就没有收益。所以代码一定要有用，能解决问题。有用的同时尽量不要有bug给使用者带来麻烦（即健壮性），否则有些人嫌麻烦就不会要，收益会下降。

- 可理解性：代码要容易理解

> 代码如果不容易理解，开发者就需要花更多的时间去理解代码，而且可能因为理解不够而犯错，导致需要付出额外的成本解决问题。

- 可调试、可测试性：代码要容易调试、测试

> 代码容易调试，测试，就是说能以更小的成本完成可用的代码，解决问题。

- 可移植、可扩展、可修改：代码要适应更多场景、易于推广，更容易修改以适应变化，和容易扩展以增加新的能力

> 代码适应多场景，也就是说代码可以放到别的场景里，解决更多相同问题。同一段代码，其收益就增加了。代码更容易修改和扩展，也就是说代码可以复制到别的场景，解决更多相似问题，收益也增加了。

- 高效：代码应当高效且不失可理解性
- 适度：代码达成以上要求可获得的收益大于需要付出的代价

什么是好代码

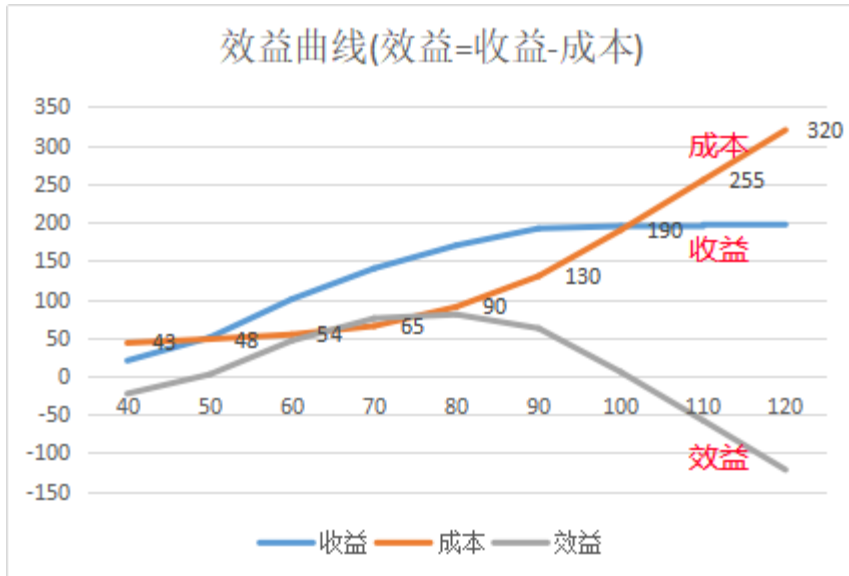
一段代码是好是坏，跟代码所处的场景上下文有关。比如，如果在某个场景里，要做到可修改扩展很难，但收效甚小（比如该场景没有扩展的必要），那把代码写得可修改可扩展就不值得，这样的代码就不是好代码。我们把这种情况叫做“过度设计”。好比写文章也讲究遣词造句，但不应该因为遣词造句影响语义表达，否则会被认为是花架子。

这是很重要的一条判断代码好坏的原则，按照这条原则，我们就无需讨论“代码自文档化”好还是“写注释”好，哪个效益高选哪个。一般情况下，这两者获得的收益差不多，这意味着哪个简单选哪个。

也无需讨论是“高效”好，还是“不要过早优化”好。优化代码要付出成本，如果付出成本能带来明显好处，那就值得优化，否则就按“不要过早优化”去做。有些代码很少运行到，优化之后提升的性能用户根本无感，那就没必要花力气去优化了。

我们无需讨论上述特性（可理解、可调试、可测试、可扩展等）到底应该做到什么程度，在可能产生的收益和需要付出的代价之间，取得一个平衡即可。

什么是好代码


效益曲线(效益=收益-成本)
350
300
250
200
150
100
50
0
-50
-100
-150
40
50
60
70
80
90
100
110
120
43
48
54
65
90
130
190
255
320
成本
收益
效益
收益
成本
效益

模块拆分与封装

需求发散与收敛

发散阶段

知识讲解：使用思维导图工具，以 金字塔原理 作为分析原则。

这些原则可以总结为三个关键点：需求层层分解，上层概括下层，同层独立穷尽。

- 分析需求按树形结构不断分解，发散，直到这棵节点树中每一个节点的内容都已经一目了然没有疑惑。
- 每个节点的下级节点都是对该节点内容的一个细化，不应包含无关内容。
- 同一层节点之间不要有重叠部分，也不要有缺漏，不要过早涉及下层细节。

按这三条原则进行需求分解，能最大程度保证需求点分析全面，没有遗漏。

一般来说，需求发散时从这5个维度进行发散：功能、性能、场景、API、设计约束（如可调试、可测试、可扩展、可复用等）。

需求发散与收敛

收敛阶段

从重要性，复杂度，影响面三个维度对上述发散出来的需求点进行分析，作为后续工作精力分配，后续跟进的依据。

- 筛选和整合：将发散阶段中得到的需求进行筛选和整合，将相似或重复的需求合并，消除冗余和重复的内容。
- 优先级排序：根据项目的目标和约束条件，对需求进行优先级排序。这可以通过与利益相关者讨论、评估业务价值和实现难度等方式来确定。
- 划定边界：明确定义需求的边界和范围，确保需求的清晰和一致性。
- 确定可行性：对需求进行可行性评估，考虑技术、资源和时间等方面的限制条件，确定哪些需求是可实现的，哪些需要进一步调整或放置。

通过以上步骤，你可以在需求分析的收敛阶段将发散得到的需求进行整合、筛选和明确，确保最终的需求列表具有可行性、一致性和清晰性。这有助于后续的系统设计和开发过程。

模块拆分与封装

模块化设计

【特点】

- 强调模块对外交付的接口，将复杂的技术/流程封装为易于使用的接口。
- 屏蔽了模块内部的巨大复杂度，使得每个人只需关注自己负责的一小块工作内容。

【外部呈现】呈现不佳，会导致模块不好用，别人不想用

- 模块职责单一
- 接口满足 SOLID 原则

【内部结构】结构不佳，会导致模块问题多，别人不敢用

- 模块做好分层，分治
- 层与层之间做好解耦

模块拆分与封装

封装

【定义】

- “抽象”是指提炼出不变的逻辑的过程
- “封装”是指将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体（即类、模块）。封装的目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是要通过外部接口，以特定的访问权限来使用类的成员，核心在于数据隐藏

【核心原则】

- 将不需要对外提供的内容都隐藏起来
- 把内部特性都隐藏，提供公共方法、公共配置对其访问

注意，封装数据主要原因是保护隐私，将数据隐藏起来不是目的。隐藏起来然后对外提供操作该数据的接口，然后我们可以在接口附加上对该数据操作的限制，以此完成对数据属性操作的严格控制，也是对模块自身的保护，让外部操作完全可控

API 设计

> SOLID - 提升接口的可复用、可扩展性

- 职责单一原则 & 接口隔离：接口只做一件事情，而不在于事情大小；一个功能实现只允许出现一种解决方案
- 开闭原则：接口针对扩展打开，针对修改封闭，要求在不修改现有代码的基础上扩展功能
- 里氏替换原则：对接口行为约束，需要保证接口的不同实现之间可以相互替换
- 依赖倒置原则：模块之间不要有依赖，一个模块不直接依赖另一个模块的细节，而是共同依赖定义明确的接口
- 迪米特法则：只调用公共接口，不允许私自依赖内部实现

职责单一原则（Single Responsibility Principle）

一个类或模块应该有且只有一个引起它变化的原因。换句话说，一个类或模块只应该负责一项特定的职责或功能。

单一职责是软件工程中一条著名的原则，然而知易行难，一是我们对于具体业务逻辑中「职责」的划分可能存在难度，二是部分同学仍没有养成贯彻此原则的习惯。

小到函数级别的 API，大到整个包，保持单一核心的职责都是很重要的一件事。

```
// fail
component.fetchDataAndRender(url, template);

// good
const data = component.fetchData(url);
component.render(data, template);
```

如上，将混杂在一个大坨函数中的两件独立事情拆分出去，保证函数（function）级别的职责单一。

在文件（file）层面同样如此，一个文件只编写一个类，保证文件的职责单一

最后，视具体的业务关联度而决定，是否将一簇文件做成一个包（package），或是拆成多个。

开闭原则 （Open-Closed Principle）

接口针对扩展打开，针对修改封闭，要求在不修改现有代码的基础上扩展功能

```
const renderFunctions = {
  item: () => {
    console.log('render in item-style.');
  },
  shop: () => {
    console.log('render in shop-style.');
  },
  other: () => {
    console.log('render in other styles, maybe banner or sth.');
  },
};

// 根据 type 调用对应的渲染函数
function renderFeed(type) {
  const renderFunction = renderFunctions[type || 'other'];
  renderFunction();
}

// 使用示例
renderFeed('item'); // 输出: 'render in item-style.'
renderFeed('shop'); // 输出: 'render in shop-style.'
renderFeed('other'); // 输出: 'render in other styles, maybe banner or sth.'
```

依赖倒置原则 （Dependency Inversion Principle）

模块之间不要有依赖，一个模块不直接依赖另一个模块的细节，而是共同依赖定义明确的接口

假设我们正在设计一个日志记录工具库，它可以将日志信息发送到不同的目标，例如控制台、文件或网络接口。为了体现依赖倒置原则，我们可以创建一个抽象的日志目标接口，并让具体的目标实现该接口。

```
// 抽象的日志目标接口
class LoggerTarget {
  log(message) {
    throw new Error('log() method must be implemented');
  }
}
// 控制台日志目标
class ConsoleLoggerTarget extends LoggerTarget {
  log(message) {
    console.log(`Console Logger: ${message}`);
  }
}
// 文件日志目标
class FileLoggerTarget extends LoggerTarget {
  log(message) {
    // 写入日志到文件的逻辑
    console.log(`File Logger: ${message}`);
  }
}
```

```
// 日志记录器
class Logger {
  constructor(target) {
    this.target = target;
  }

  log(message) {
    this.target.log(message);
  }
}
// 使用示例
const consoleLogger = new Logger(new ConsoleLoggerTarget())
consoleLogger.log('This is a console log message.');

const fileLogger = new Logger(new FileLoggerTarget());
fileLogger.log('This is a file log message.');
```

API 设计

> 如何做？ - 以用户视角、以用户视角、以用户视角

- 梳理需求——场景梳理
- 设计接口——功能说明、接口原型、接口约束、使用说明
- 编写 DEMO ——通过 API 接口实现场景 DEMO
- 检视改进接口——修改、简化、增加以及可扩展性需求场景完善，最后一步再做简化，减少 API
- 实现 API 接口

API 设计 - 系统性

不管是大到发布至业界，或小到在公司内跨部门使用，一组 API 一旦公开，整体上就是一个产品，而调用方就是用户。

所谓牵一发而动全身，一个小细节可能影响整个产品的面貌，一个小改动也可能引发整个产品崩坏。因此，我们一定要站在全局的层面，系统性地把握整个体系内 API 的设计，体现大局观。

设计扩展机制

毫无疑问，在保证向下兼容的同时，API 需要有一个对应的扩展机制以可持续发展

一方面便于开发者自身增加功能，另一方面用户也能参与进来共建生态。

- express, koa - 中间件
- webpack, rollup, vite - 插件系统

收敛 API 集

确保API集合在一致的抽象级别上，并适当合并API以减小整个集合的信息量，这是设计一个高效且易于使用的API体系的重要原则之一。

假设我们正在设计一个图形处理的API集合，其中包含一些用于变换和编辑图形的功能。在这个API集合中，我们可能会有以下几个单独的API：

- translateShape(shape, dx, dy): 将给定的图形对象沿着x和y方向平移指定的距离。
- scaleShape(shape, factor): 将给定的图形对象按照指定的因子进行缩放。
- rotateShape(shape, angle): 将给定的图形对象按照指定的角度进行旋转。

这些API虽然提供了变换和编辑图形的基本功能，但在使用时需要关注多个单独的API，每个API都有自己的参数和用法。为了提高一致性和简洁性，我们可以将它们收敛成更高级别的API。

我们可以设计一个更通用的`transformShape(shape, transformations)`方法，其中`transformations`参数是一个包含平移、缩放和旋转等操作的对象。通过将多个变换操作合并到一个API中，用户可以更方便地对图形进行复杂的变换操作，而不必关注每个操作的具体实现。

其他方面

遵循一致的 API 风格

控制 API 的抽象级别

对外暴露的接口要克制

版本控制

今天的练习

关键产出：

- 通过需求的分散与收敛，以及场景分析，输出 **思维导图**
- 根据上面的分析结果，输出 **API设计** 和 **DEMO示例**

其他产出：

- 通过模块拆分与封装提升模块的可扩展性、可复用性
- 完成功能并补上单元测试

补充说明：

- API 设计尽可能完善，但是不是说你设计了多少 API 就需要全部实现。