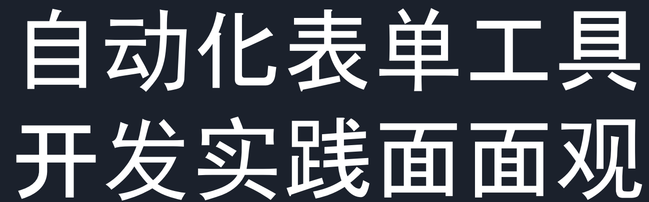


用 chrome API + mock.js + Fiber 解放你的表单填充与页面操作生产力

©hijiangtao

目录

1. 表单场景综述
2. 技术调研
3. 识别表单
4. 构造数据与表单填充
5. 功能集成与插件开发
6. Q&A

©hijiangtao

表单场景综述

©hijiangtao

©hijiangtao

```
<Form
  disabled
  autoComplete='off'
  ref={formRef}
  {...formItemLayout}
  size={size}
  initialValues={{
    slider: 20,
    'a.b[0].c': ['b'],
  }}
  onValuesChange={onValuesChange}
  scrollToFirstError
>
  <FormItem
    label='Username'
    field='name'
    rules={[{ required: true, message: 'username is required' }]}
  >
    <Input placeholder='please enter...' />
  </FormItem>
  <FormItem label='Age' field='age' rules={[{ type: 'number', required
    <InputNumber placeholder='please enter' />
  </FormItem>
  <FormItem
    label='Province'
    field='province'
    rules={[
      {
        type: 'array',
        required: true,
      },
```

* Username: please enter...
* Age: please enter
* Province: please select
* Auto-complete: please enter
* Post: please select
* Multiple Choice: b
* Score
* Date: 请选择日期

Switch

Radio: A B C D

Slide

Upload: +

I have read the employee manual

研发眼中的表单 vs 使用者眼中的表单

举几个例子

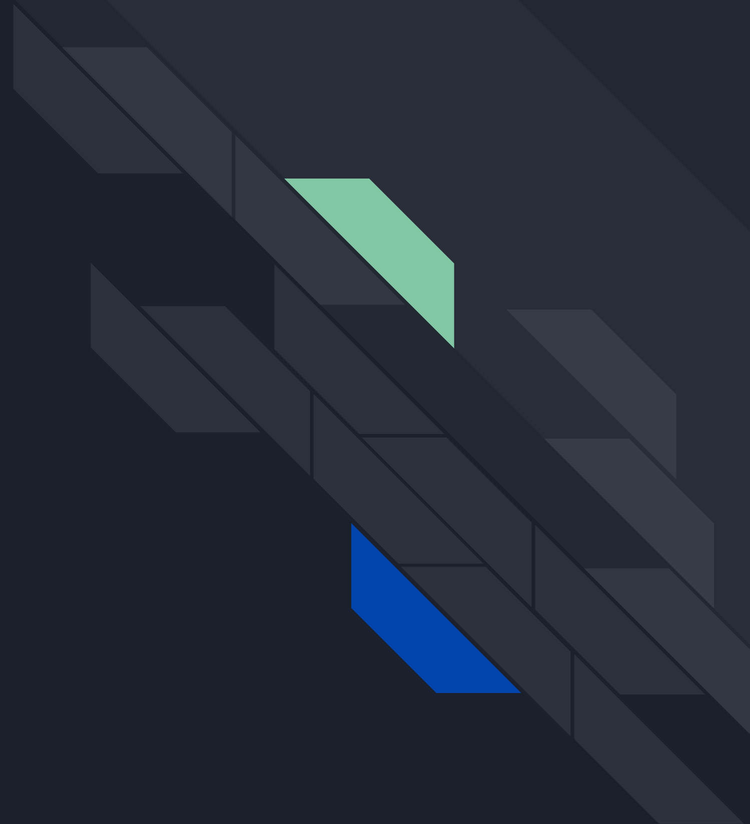
1. 对接完产品需求，终于把字段与各类逻辑组装进表单，想自测看看交互效果
2. 功能提测后，为了测试新功能，需要先填一遍复杂表单，造数据
3. 每次填写表单，由于某个字段是存储主键，每次填写不能完全一致
4. 登陆态过期太频繁，一过期就要重新输入用户名密码再点击按钮登陆一遍

场景汇总

1. 研发自测
2. 测试数据构造
3. 个性化填写
4. 日常登陆操作

©hijiangtao

技术调研

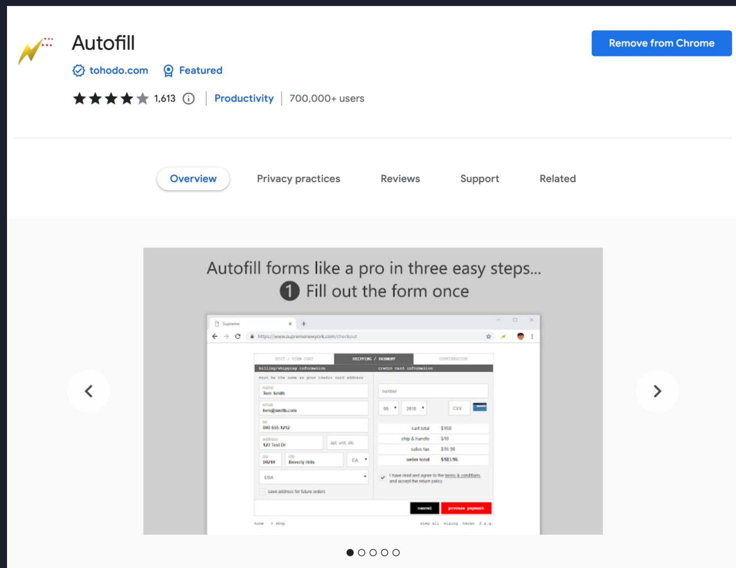


©hijiangtao

技术调研

业务开发无感知，无侵入式方案

1. 不依赖 UI 框架，通用性强
2. 识别与填充成功率低，支持控件有限
3. 随机 mock 数据能力有限
4. 不支持页面上表单与事件连续操作



©hijiangtao

技术调研

侵入式改造

1. 基于具体表单结构定制，通用性弱
2. 完美填充，支持全部表单控件
3. 不支持页面上表单与事件连续操作
4. 填充数据变更时需重新打包，成本高昂

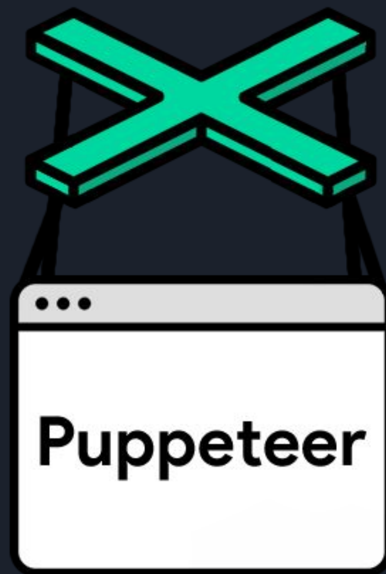
```
79  function App() {
80    const formRef = useRef();
81
82    useEffect(() => {
83      formRef.current.setFieldsValue({
84        name: 'Mock Name',
85        rate: 5,
86      });
87    }, []);
88
89    const onValuesChange = (changeValue, values) => {
90      console.log('onValuesChange: ', changeValue, values);
91    };
92
93    return (
94      <div style={{ maxWidth: 650 }}>
95        <Form
96          ref={formRef}
97          {...formItemLayout}
98          initialValues={{
99            slider: 20,
100           'a.b[0].c': ['b'],
101         }}
102         onValuesChange={onValuesChange}
103         scrollToFirstError
```

©hijiangtao

技术调研

自动化测试方案

1. 通用性强
2. 基于 puppeteer 或者 e2e 框架，准确填充
3. 上手成本高，研发成本相对较高

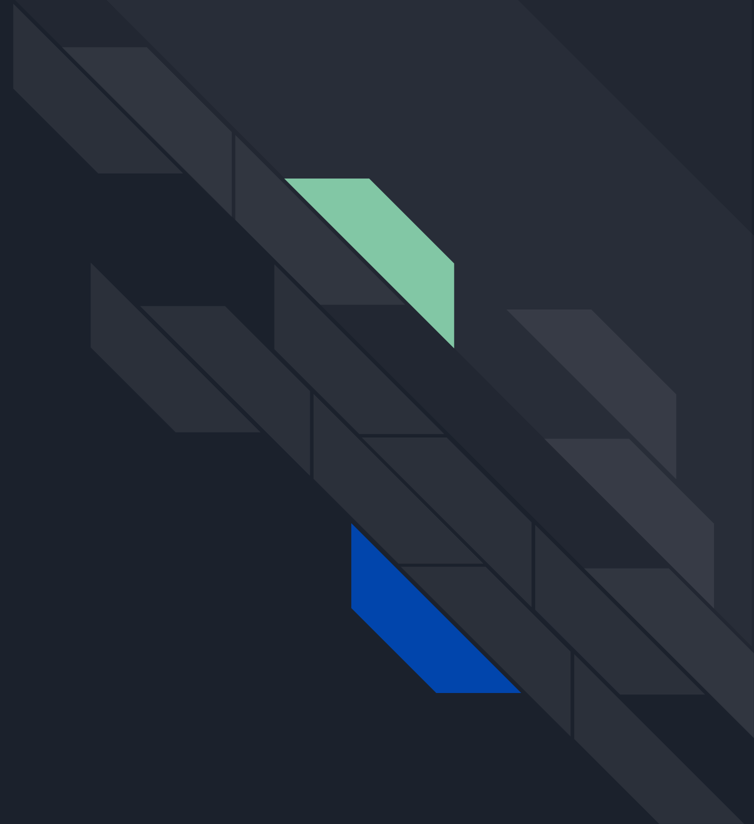


©hijiangtao

「为什么不能把无侵入式改造与高识别率结合？」

取其精华，去其糟粕

识别表单



©hijiangtao

Virtual DOM 是对真实 DOM 的模拟，也是一棵树，通过 Diffing 算法和老树对比，得到差值，再同步给视图要修改哪些部分。

Fiber 是对 React 核心算法的重构，Fiber 对象是一个用于保存「组件状态」、「组件对应的 DOM 的信息」、以及「工作任务 (work)」的数据结构，Fiber node 是 Fiber 对象的实例。

©hijiangtao

Fiber 树的组织与遍历

数组实现

```
{
    "name": "A",
    "children": [
        { "name": "B" },
        {
            "name": "C",
            "children": [
                { "name": "E" }
            ]
        },
        { "name": "D" }
    ]
}
```

链表实现

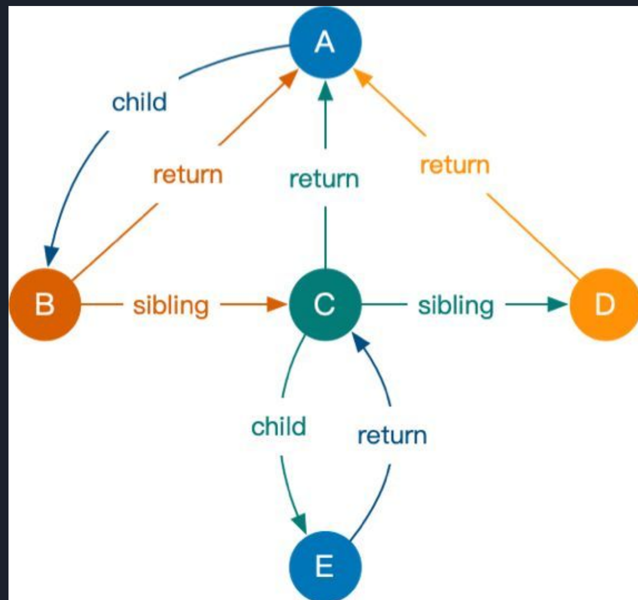
```
A = { child: B }
B = { return: A, sibling: C }
C = { return: A, sibling: D, child: E }
D = { return: A }
E = { return: C }
```

©hijiangtao

Fiber 树的组织与遍历

链表的好处

- 调整节点位置很灵活，只要改改指针
- 方便进行各种方式的遍历
- 可以随时从某一个节点出发还原整棵树



截图取自《如何理解 React Fiber 架构？ - 几木的回答》
https://www.zhihu.com/question/49496872/answer/2517859568

©hijiangtao

Fiber 树的组织与遍历

遍历流程

```
// The fiber we're working on
let workInProgress: Fiber | null = null;

function workLoopSync() {
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}

function performUnitOfWork(unitOfWork: Fiber): void {
  next = beginWork(current, unitOfWork, renderLanes);

  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next === null) {
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}
```

关于 performUnitOfWork 的更完整代 码详见
https://github.com/facebook/react/blob/c1d414d75851aee7f25f69c1b6fda6a14198ba24/packages/react-reconciler/src/ReactFiberWorkLoop.new.js#L2051-L2077

Fiber 树的组织与遍历

遍历中的递归

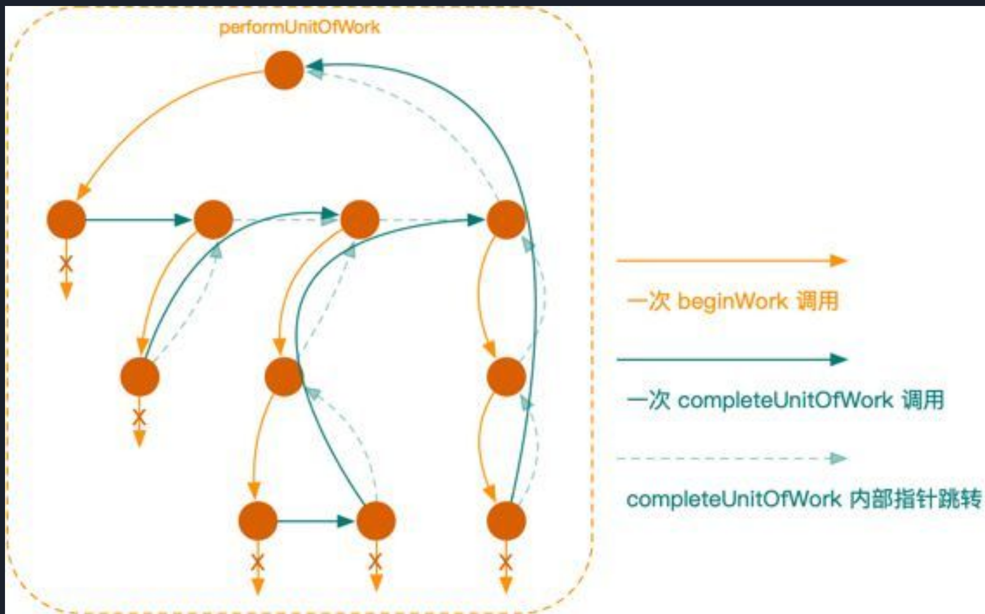
```
function completeUnitOfWork(unitOfWork: Fiber): void {
  let completedWork = unitOfWork;
  do {
    const returnFiber = completedWork.return;

    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      // If there is more work to do in this returnFiber, do that next.
      workInProgress = siblingFiber;
      return;
    }
    // Otherwise, return to the parent
    completedWork = returnFiber;
    // Update the next thing we're working on in case something throws.
    workInProgress = completedWork;
  } while (completedWork !== null);

  // We've reached the root.
  if (workInProgressRootExitStatus === RootInProgress) {
    workInProgressRootExitStatus = RootCompleted;
  }
}
```

关于 completeUnitOfWork 的更完整代 码详见 https://github.com/facebook/react/blob/c1d414d75851aee7f25f69c1b6fda6a14198ba24/packages/react-reconciler/src/ReactFiberWorkLoop.new.js#L2173-L2271

©hijiangtao

Fiber 树的组织与遍历

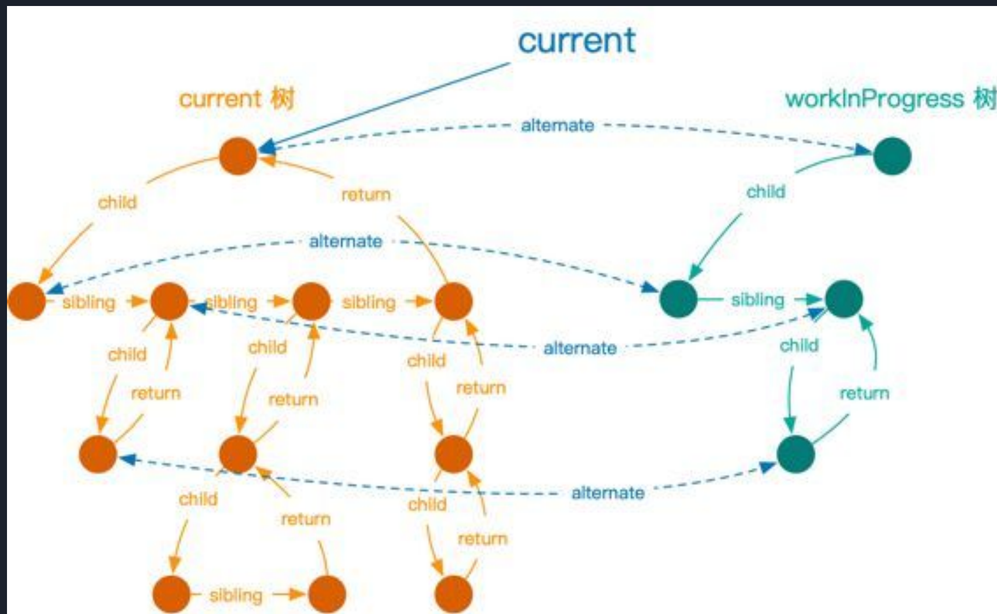
遍历示意图



截图取自《如何理解 React Fiber 架构？ - 几木的回答》
https://www.zhihu.com/question/49496872/answer/2517859568

Fiber 树的构建与 Diffing

新树/旧树的构建与连接结构

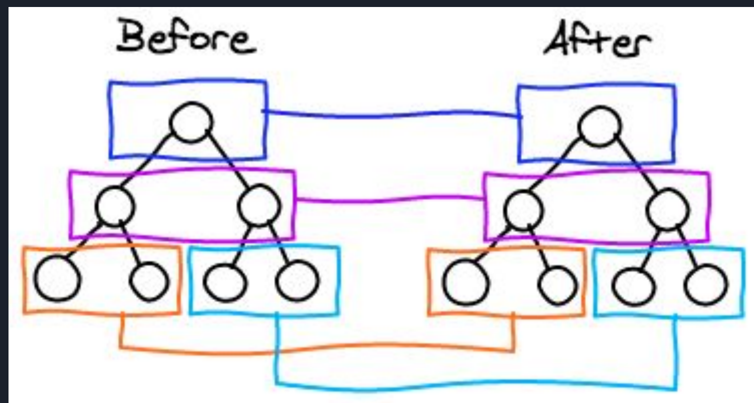


截图取自《如何理解 React Fiber 架构？ - 几木的回答》
https://www.zhihu.com/question/49496872/answer/2517859568

©hijiangtao

Fiber 树的构建与 Diffing

找到两棵任意的树之间的最小的差异是一个复杂度为 O(n3) 的问题，React Diff 算法通过一些假设，最终达到了接近 O(n) 的复杂度。



截图取自《React Diff 算法核心》
https://xyy94813.gitbook.io/x-note/fe/react/react-diff-algorithm

1. 假设一：不同类型的两个元素将产生不同的树，遇此情况时 React 会拆卸原有节点并且建立新的节点（触发重建流程）。
2. 假设二：默认情况下，在 DOM 节点的子节点上递归时，React 只会同时遍历两个子节点列表，并在存在差异时生成一个更新操作。
3. 假设三：用户给每个子节点提供一个 key，标记它们“是同一个”，在有 key 的情况下能保证二者都复用仅做移动，但无 key 就会造成两个不必要的卸载重建。

关于 React Diff 算法思想及其假 设内容的描述 节选自
https://xyy94813.gitbook.io/x-note/fe/react/react-diff-algorithm

©hijiangtao

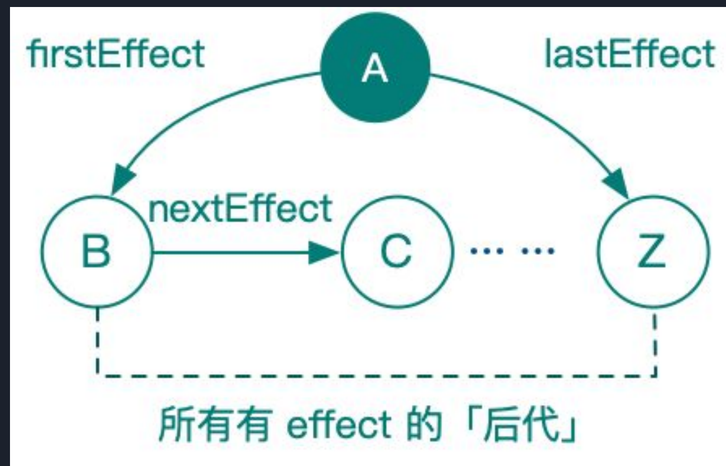
Fiber 树的构建以及 Diffing 都是同时进行的，不是说构建完 Fiber 树之后再开始 Diffing 寻找差距。

同样的，两棵树 Diffing 的过程中，就已经决定了哪些旧节点需要复用、删除、移动，哪些新节点需要创建。

©hijiangtao

副作用与收集过程

1. 副作用是向上收集的，每次在 completeUnitOfWork 中循环经过一个节点时，会同时合并后代节点的 effectList 以及自己的 effectList；
2. 副作用同样采用链表的方式存储，并通过 fisrtEffect —> nextEffect —> lastEffect 的关系串联起来，但此链表与 Fiber 树的链表结构没有关系；



利用 Fiber 识别表单

获取 Fiber 实例，拿到目标属性与方法，解析表单结构

1. 从 DOM 中找到目标 form 元素
2. 获取有效的 Fiber 实例
3. 读取目标属性值，解析表单结构

```
/**
 * 获取 Fiber 实例
 * @param dom
 * @param traverseUp
 */
function getFiberInstance(dom: HTMLElement, traverseUp = 0)
{ if (!dom) {
    return null
  }

  const key = Object.keys(dom).find((key) => {
    return (
      key.startsWith("__reactFiber$") || // react 17+
      key.startsWith("__reactInternalInstance$")
    ) // react <17
  })
  const domFiber = dom[key]
  if (domFiber == null) return null

  // react <16
  if (domFiber._currentElement) {
    let compFiber = domFiber._currentElement._owner
    for (let i = 0; i < traverseUp; i++) {
      compFiber = compFiber._currentElement._owner
    }
    return compFiber._instance
  }

  // react 16+
  const getCompFiber = (fiber) => {
    let parentFiber = fiber.return
    while (typeof parentFiber.type == "string") {
      parentFiber = parentFiber.return
    }

    return parentFiber
  }
  let compFiber = getCompFiber(domFiber)
  for (let i = 0; i < traverseUp; i++) {
    compFiber = getCompFiber(compFiber)
  }

  return compFiber
}
```

©hijiangtao

构造数据与表单填充

©hijiangtao

1. 固定取值：填入，每次填入相同值即可，无需特殊处理
2. 规则取值：每次填入的字段都需要取一个符合相同规则但取值不同的数值
3. 指定集合：针对单选、多选等场景，需要从指定选项中随机选取一个填入
4. 时间取值：在有效的时间范围内随机生成一个时间串
5. 布尔取值：checkbox、radio 等组件实际取值为 true/false 二选一
6. ……

©hijiangtao

表单中都存在哪些数据

mock.js

`Random.last()`

Data Template
```
// Random.last()
Random.last()
Mock.mock('@last')
Mock.mock('@last()')
```
=>

Result
```
// Random.last()
"Johnson"
"Rodriguez"
"Wilson"
```

`Random.name( middle? )`

Data Template
```
// Random.name()
Random.name()
Mock.mock('@name')
Mock.mock('@name()')

// Random.name( middle )
Random.name(true)
Mock.mock('@name(true)')
```
=>

Result
```
// Random.name()
"Michael Anderson"
"Gary Lopez"
"Ronald Moore"

// Random.name( middle )
"Larry Eric White"
"Scott Ronald Lewis"
```

`Random.cfirst()`

Data Template
```
// Random.cfirst()
Random.cfirst()
Mock.mock('@cfirst')
Mock.mock('@cfirst()')
```
=>

Result
```
// Random.cfirst()
"邵"
"赵"
"夏"
```

`Random.clast()`

Data Template
```
// Random.clast()
Random.clast()
Mock.mock('@clast')
Mock.mock('@clast()')
```
=>

Result
```
// Random.clast()
"勇"
"磊"
"超"
```

`Random.cname()`

Data Template
```
// Random.cname()
Random.cname()
Mock.mock('@cname')
Mock.mock('@cname()')
```
=>

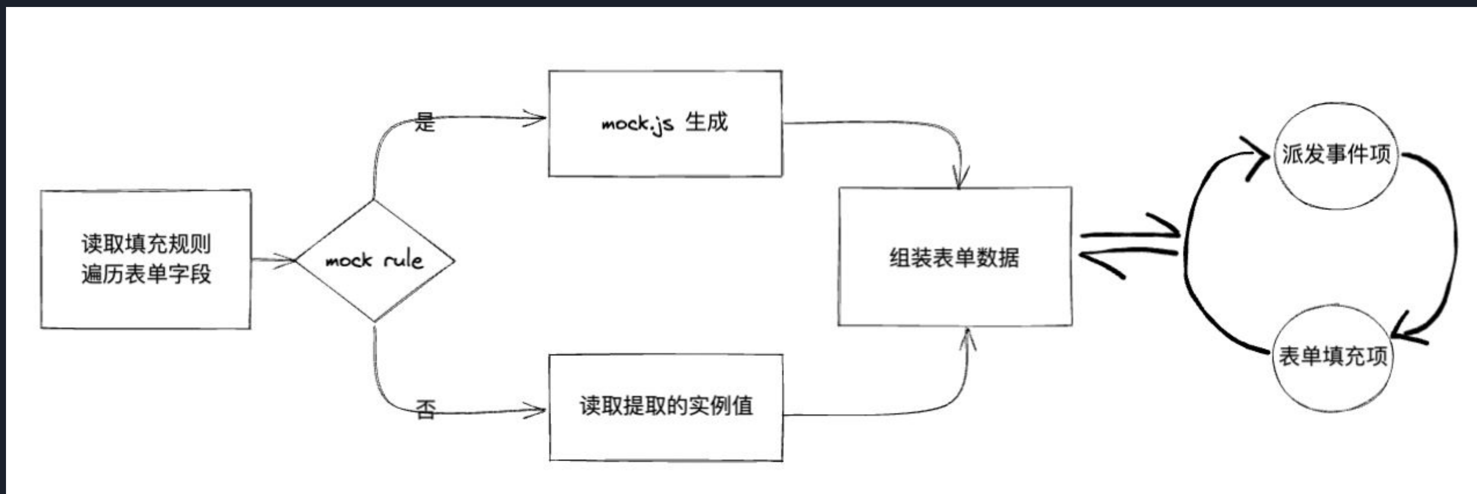
Result
```
// Random.cname()
"龚军"
"顾刚"
"赵娜"
```

©hijiangtao

1. 多步操作：表单存在字段之间的联动，无法一次完成赋值
2. 填充与事件组合：表单填充完下一步可能就是点击事件
3. ……

©hijiangtao

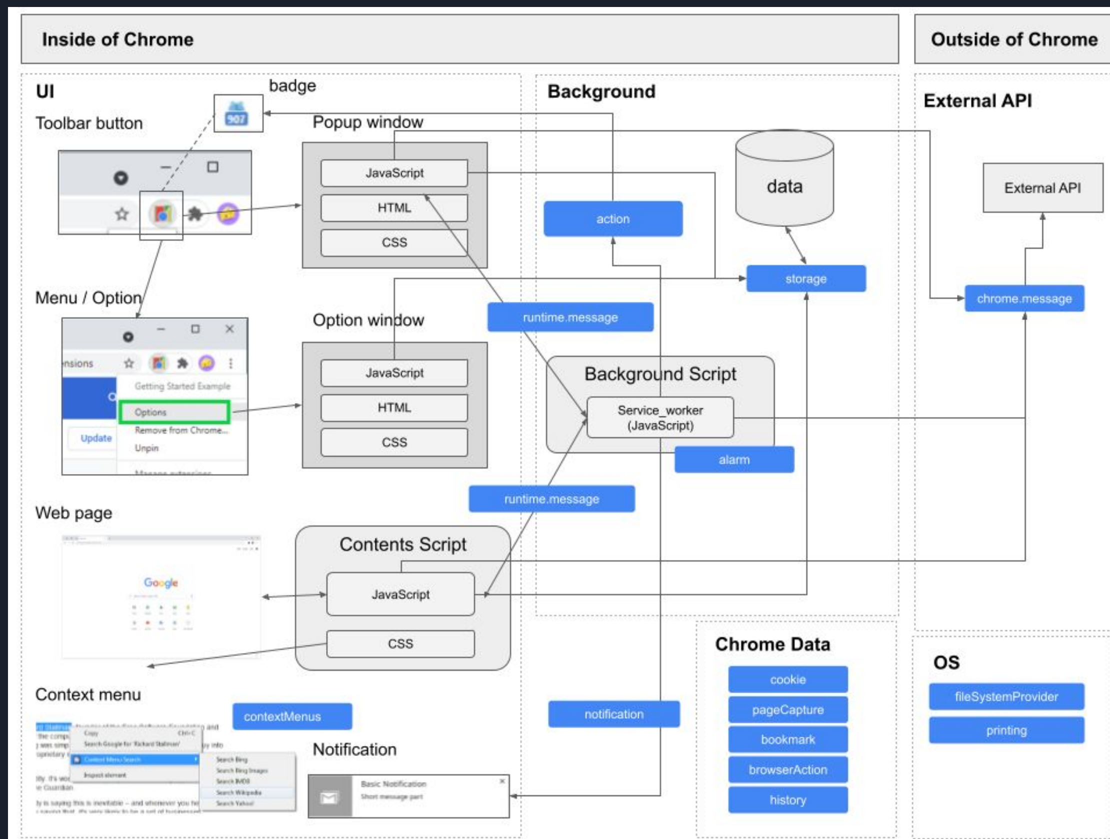
数据填充工作流程



功能集成与插件开发

©hijiangtao

插件结构



©hijiangtao

1. 代码构建打包
2. TypeScript + React 支持
3. 插件声明 manifest.json 生成
4. 跨 script 持久化存储
5. 不同 script 间 (content script / background / popup) 通信

插件通信

1. 通用的通信机制，通过 postMessage 广播消息
2. 非广播传递，MessageChannel 传递消息
3. 通过 chrome API 进行消息通信
    a. chrome.devtools. inspectedWindow.eval
    b. chrome.tabs.sendMessage
    c. chrome.tabs.connect
    d. chrome.extension.getBackgroundPage

关于跨线程通信相关的技 术方案，我在《Service Worker 实践指南》一文中有 详细介绍，感兴趣的同学可以 查阅 https://hijiangtao.github.io/2021/04/13/Service-Worker-Practical-Notes/

©hijiangtao

插件安全

插件权限:chrome 插件在运行中涉及到的权限调用需要在 manifest permissions 中声明

script 对 chrome API 权限:content scripts 中只允许如下几类 chrome.***.api 调用

- chrome.extension(getURL , inIncognitoContext , lastError , onRequest , sendRequest)
- chrome.i18n
- chrome.runtime(connect , getManifest , getURL , id , onConnect , onMessage , sendMessage)
- Chrome.storage

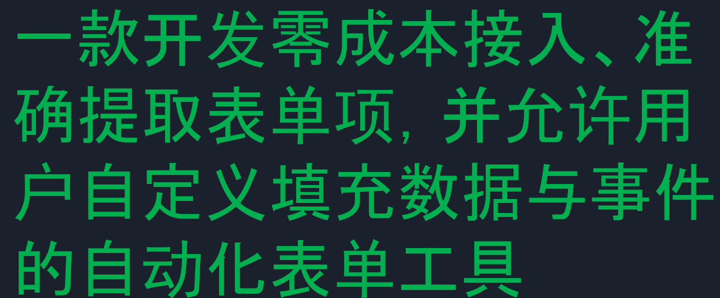
共享 DOM 权限:content scripts UI 部分不支持针对 DOM 的 Expando 属性的共享

©hijiangtao

功能集成

1. 支持表单自定义 mock 规则生成填充数据
2. 不同规则适配不同页面地址
3. 配置数据的导入与导出
4. 表单识别后的规则提取与拆分
5. 支持自定义事件组合
6. ……

©hijiangtao



一款开发零成本接入、准确提取表单项，并允许用户自定义填充数据与事件的自动化表单工具

产品定位

Q&A

自动化表单工具
开发实践面面观

©hijiangtao