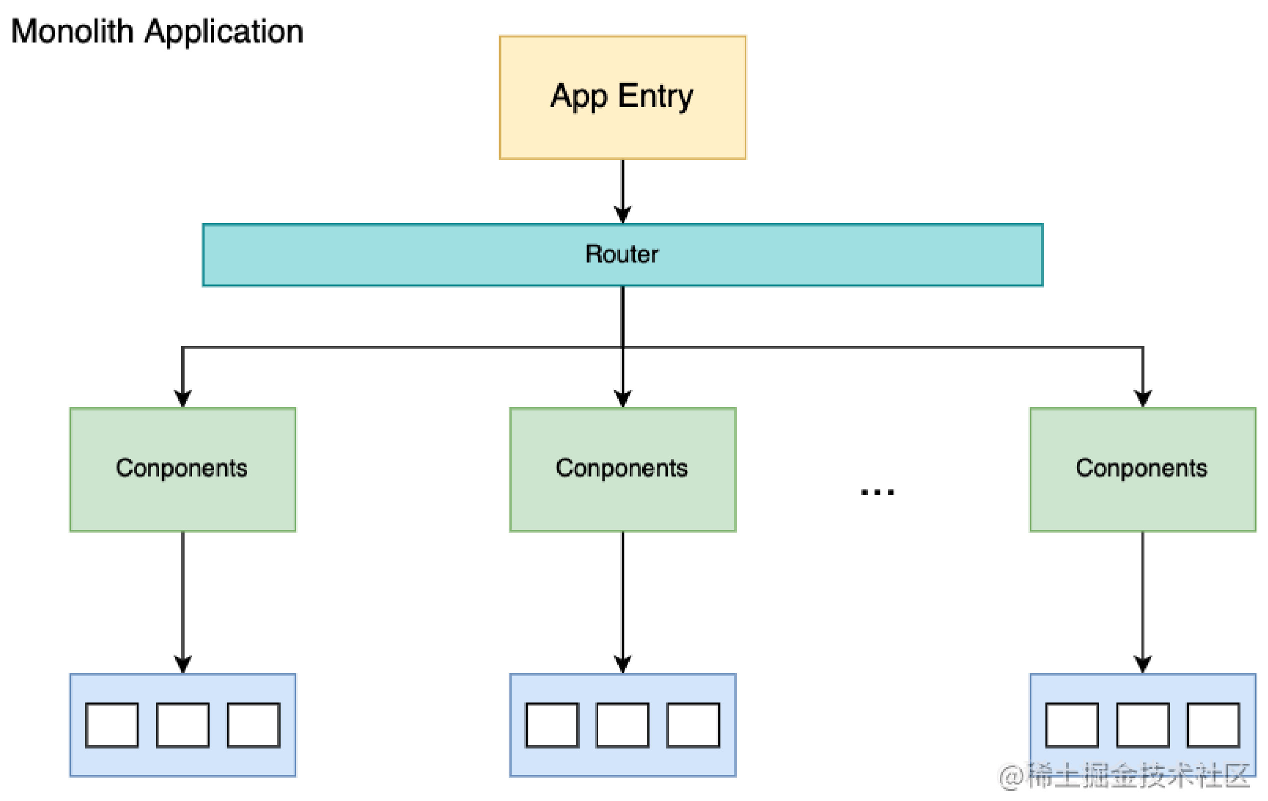
Why Micro Frontend

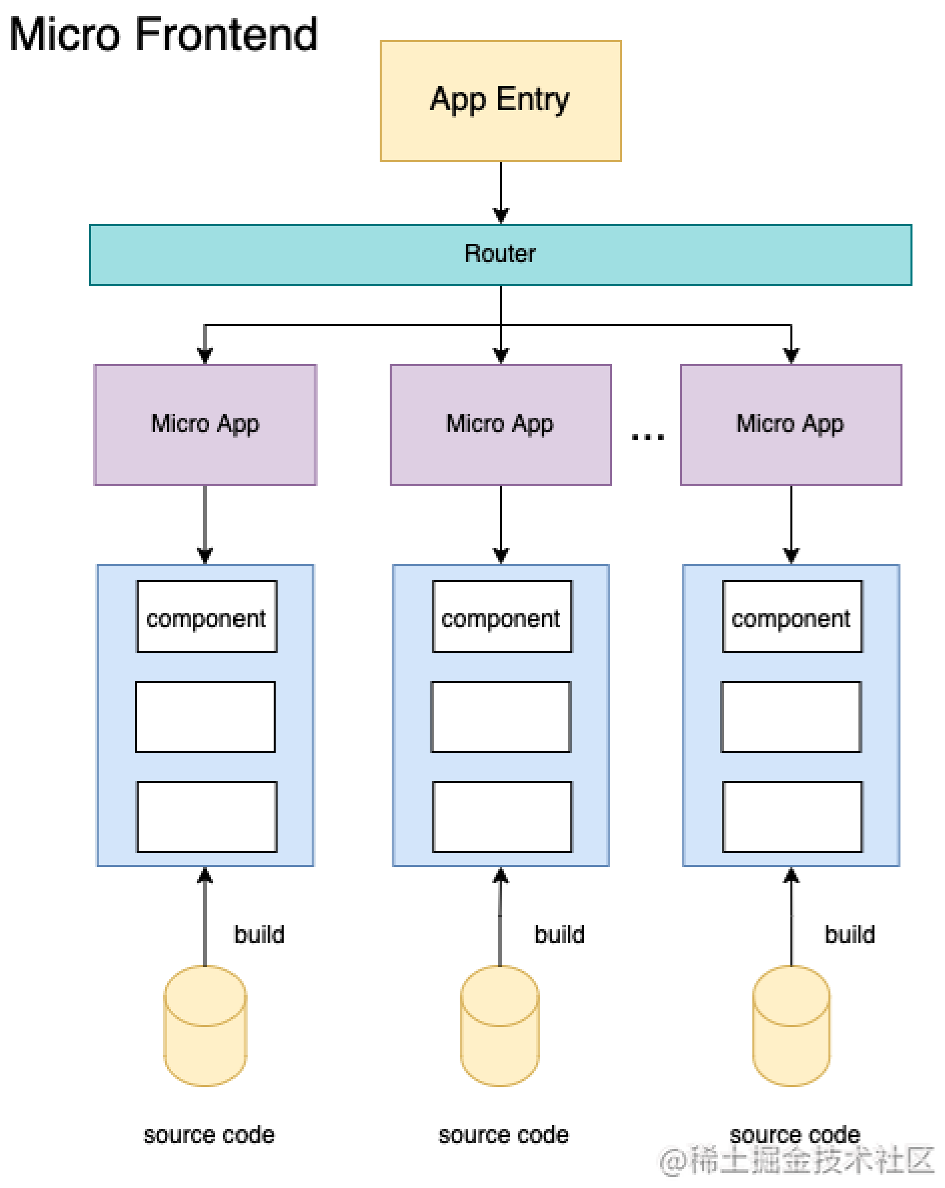
- 巨石应用(Monolith Application)的产生



  - 带来的问题(Disadvantages of Monolith)
    1. Difficult to deploy and maintain
    2. Obstacle to frequent deployments
    3. Dependency between unrelated features
    4. Makes it diffcult to try out new technologies/framework
- 微前端(Micro Frontend)
  - 概念(concept)

    > 微前端是一种类似于微服务的架构，是一种由独立交付的多个前端应用组成整体的架构风格，将前端应用分解成一些更小、更简单的能够独立开发、测试、部署的应用，而在用户看来仍然是内聚的单个产品。

  - 架构

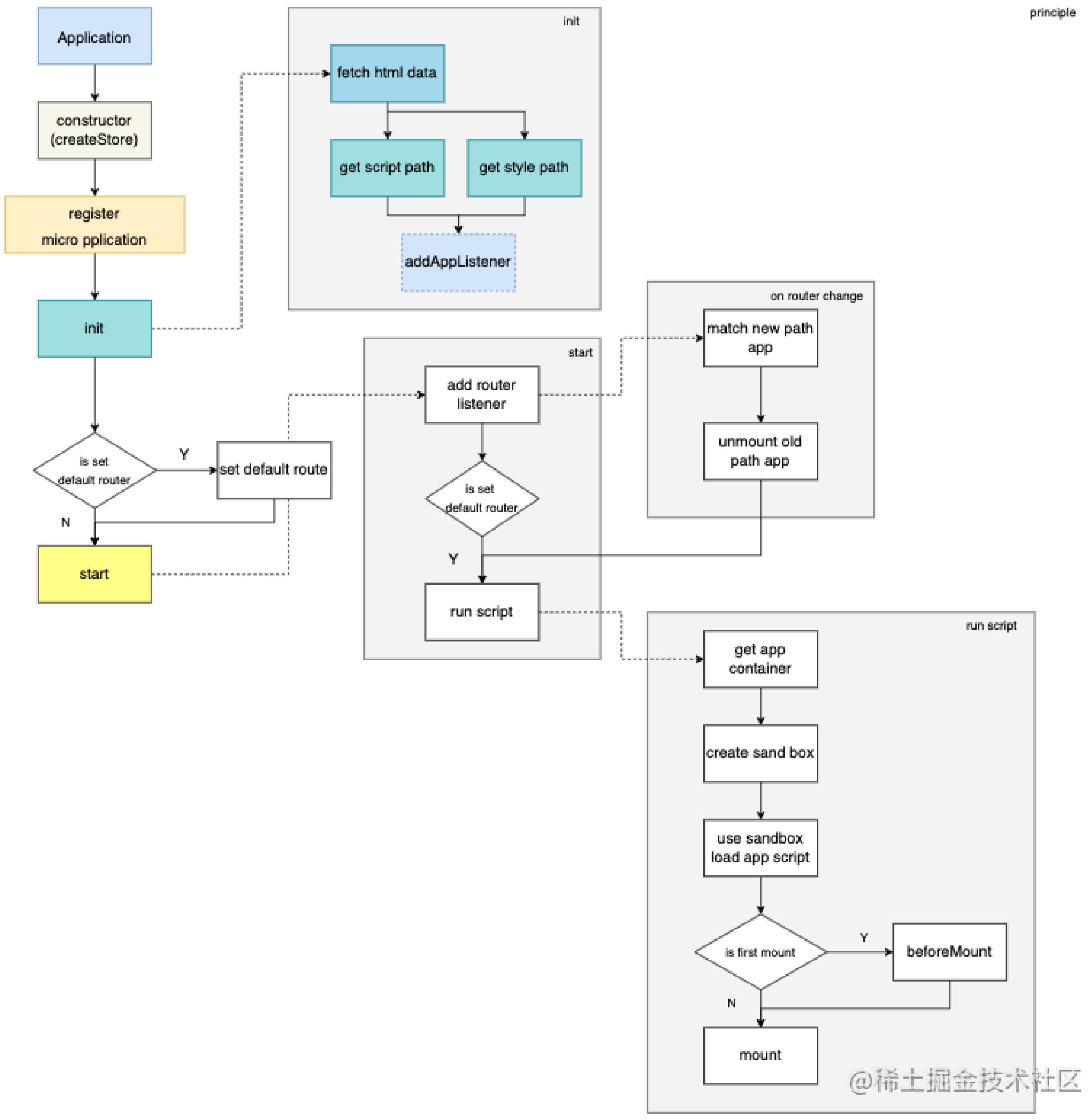


- 优缺点
  - Advantage
    - 应用自治。只需要遵循统一的接口规范或者框架，以便于系统集成到一起，相互之间是不存在依赖关系的。
    - 单一职责。每个前端应用可以只关注于自己所需要完成的功能。
    - 技术栈无关。可以使用 原生 JS 的同时，又可以使用 React 和 Vue。
  - Disadvantage
    - 维护问题
    - 架构复杂
- 应用场景(scenes)
  - Middleground/Background

微前端实现方式与原理分析(principle)

- 微前端实现方式
  - iframe

        - why not iframe[1]
            - 性能问题，iframe 每次进入子应用时都会重新渲染，资源也会重新加载
            - 全局上下文完全隔离，内存变量不共享。cookie 不共享，需要搭建特定的 channel 进行通信
            - DOM 结构不共享，无法使用全局弹窗
            - url 不同步，刷新页面，无法使用前进/后退
    - Web Components
        - 子应用打包成 npm 模块。主应用通过 npm 安装，import 的方式引入
    - nginx + Components
        - 通过 nginx 路由显示不同的子应用
    - 基于系统基座实现
        - 主应用搭建一个基座，在基座中渲染子应用
        - qiankun，singleSPA
- 微前端应用加载流程图

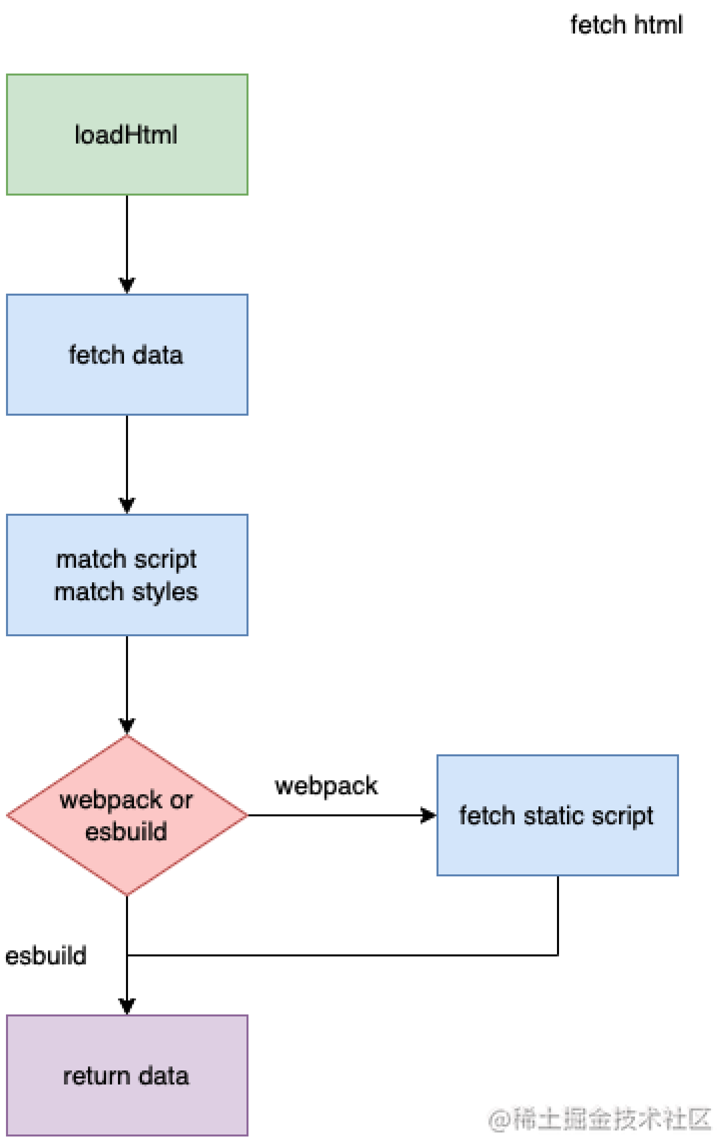


- 需要实现的功能
    1. 加载 HTML

2. 加载 JS 文件
3. 定义生命周期
4. 使用沙箱隔绝执行作用域
5. 创建应用间通信 channel
6. 路由监听
7. 样式隔离

实现流程

1. 获取应用

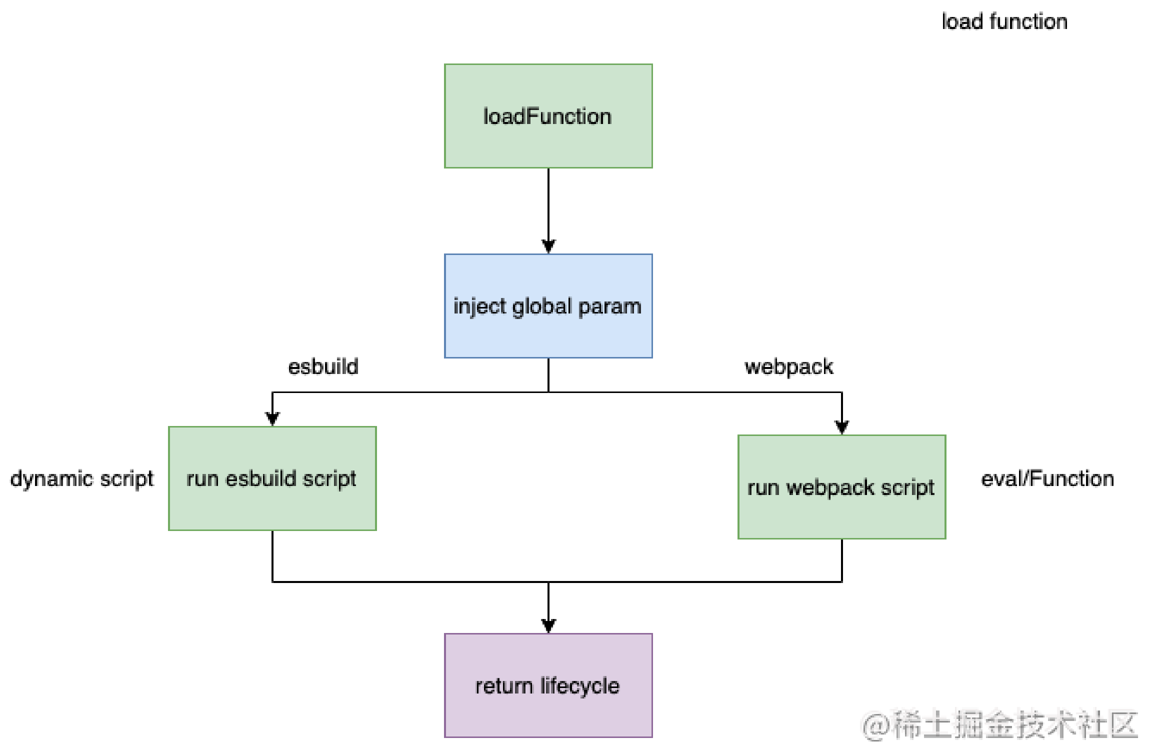


   - 获取 HTML 文件
     - loadHtml

```
async function loadHtml(
  entry: string,
  type: LoadScriptType
): Promise<LoadHtmlResult> {
  const data = await fetch(entry, {
    method: 'GET',
  });
```

```
    let text = await data.text();
    const scriptArr = text
      .match(scriptReg)
      ?.filter((val) => val)
      .map((val) => (isHttp.test(val) ? val : `${entry}${val}`));
    const styleArr = text
      .match(styleReg)
      ?.filter((val) => val)
      .map((val) => (isHttp.test(val) ? val : `${entry}${val}`));
    text = text.replace(/(<script.*><\/script>)/g, '');
    console.log(scriptArr);

    const scriptText: string[] = [];
    if (type === 'webpack' && scriptArr) {
      for (const item of scriptArr) {
        let scriptFetch = await fetch(item, { method: 'GET' });
        scriptText.push(await scriptFetch.text());
      }
    }

    return {
      entry,
      html: text,
      scriptSrc: type === 'webpack' ? scriptText : scriptArr || [],
      styleSrc: styleArr || [],
    };
  }
```

- 运行 JS(runScript)，定义生命周期



  - 生命周期
    - beforeMount

- mount
- unmount

```
/** 生命周期函数 */
export type LoadFunctionResult = {
  beforeMount: () => void;
  mount: (props: LoadFunctionMountParam) => void;
  unmount: (props: UnloadFunctionParam) => void;
};
export type LoadScriptType = 'esbuild' | 'webpack';

/** 注入环境变量 */
export function injectEnvironmentStr(context: ProxyParam) {
  context[PRODUCT_BY_MICRO_FRONTEND] = true;
  context.__vite_plugin_react_preamble_installed__ = true;
  return true;
}

/** 使用import加载script */
export async function loadScriptByImport(scripts: string[]) {
  injectEnvironmentStr(window);
  let scriptStr = `
      return Promise.all([`;
  scripts.forEach((val) => {
    scriptStr += `import("${val}"),`;
  });
  scriptStr = scriptStr.substring(0, scriptStr.length - 1);
  scriptStr += `]);
  `;
  return await new Function(scriptStr)();
}

/** 执行js字符串 */
export async function loadScriptByString(
  scripts: string[],
  context: ProxyParam
) {
  const scriptArr: Promise<Record<string, any>>[] = [];
  injectEnvironmentStr(context);
  scripts.forEach(async (val) => {
    scriptArr.push(
      await new Function(`
          return (window => {
            ${val}
            return window.middleVue;
          })(this)
    `).call(context)
    );
  });
  return scriptArr;
}
```

```
/** 加载JS文件 */
export async function loadFunction<T extends LoadFunctionResult>(
  context: Window,
  scripts: string[] = [],
  type: LoadScriptType = 'esbuild'
): Promise<T> {
  let result = {};
  if (type === 'esbuild') {
    result = await loadScriptByImport(scripts);
  } else {
    result = await loadScriptByString(scripts, context);
  }

  let obj: LoadFunctionResult = {
    beforeMount: () => {},
    mount: () => {},
    unmount: () => {},
  };
  (<Record<string, any>[]>result).forEach((val) => {
    Object.assign(obj, val);
  });
  return <T>obj;
}
```

2. 微应用加载类(MicroFront)

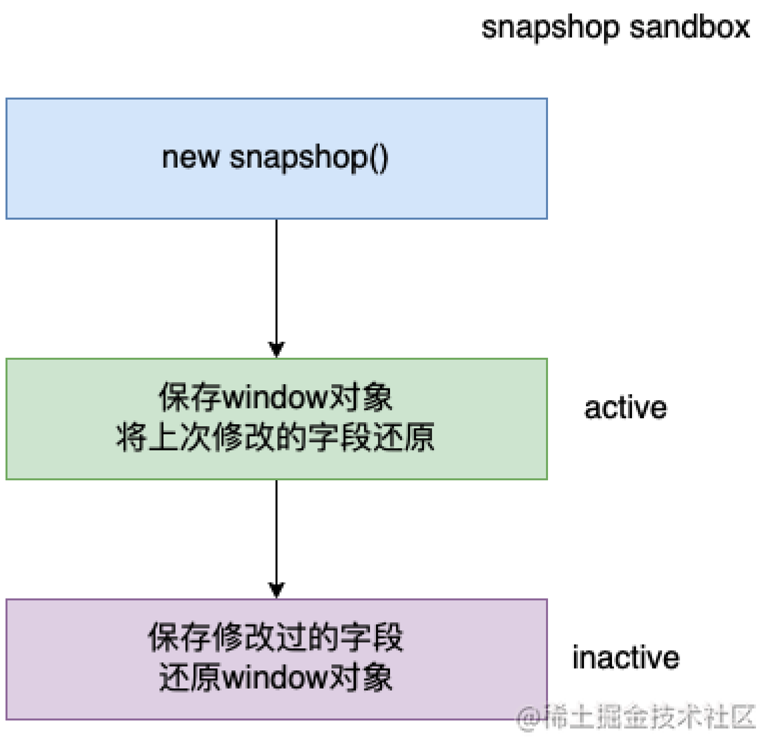
```
interface MicroFrountendMethod {
  init: () => void;
  setCurrentRoute: (routeName: string) => void;
  start: () => void;
}

export default class MicroFrountend implements MicroFrountendMethod {
  /** 微应用列表 */
  private servers: RegisterData[];
  /** 请求后的应用列表 */
  private serverLoadData: Record<string, LoadHtmlResult>;
  /** 当前路由 */
  public currentRoute: string;
  /** 当前开启的微应用容器 */
  public currentActiveApp: string[];
  /** 全局store */
  public store: Record<string, any>;

  constructor(servers: RegisterData[]) {
    this.servers = servers;
    this.serverLoadData = {};
    this.currentRoute = '';
    this.currentActiveApp = [];
    this.store = createStore();
  }
  /** 初始化 */
  public async init() {
    for (let item of this.servers) {
      const serverData = await loadHtml(item.entry, item.type);
      addNewListener(item.appName);
      this.serverLoadData[item.appName] = serverData;
    }

    return true;
  }

  /** 设置路由 */
  public setCurrentRoute(routeName: string) {
    const appIndex = this.servers.findIndex(
      (val) => val.activeRoute === routeName
    );
    if (appIndex === -1) return false;
    const appName = this.servers[appIndex].appName;
    const isInclude =
Object.keys(this.serverLoadData).includes(appName);
    if (!isInclude) {
      return false;
    }

    this.currentRoute = routeName;
    return true;
  }

  /** 开启加载微前端应用 */
  public async start() {
    const currentRoute = this.currentRoute ||
window.location.pathname;
    const appList = this.servers.filter(
      (val) => val.activeRoute === currentRoute
    );
    for (let val of appList) {
      const appName = val.appName;
      const htmlData = this.serverLoadData[appName];
      const scriptResult = await runScript(val, htmlData, this.store);
      this.serverLoadData[appName].lifeCycle = scriptResult.lifeCycle;
      this.serverLoadData[appName].sandbox = scriptResult.sandBox;
    }
  }
```

3. JS 沙箱

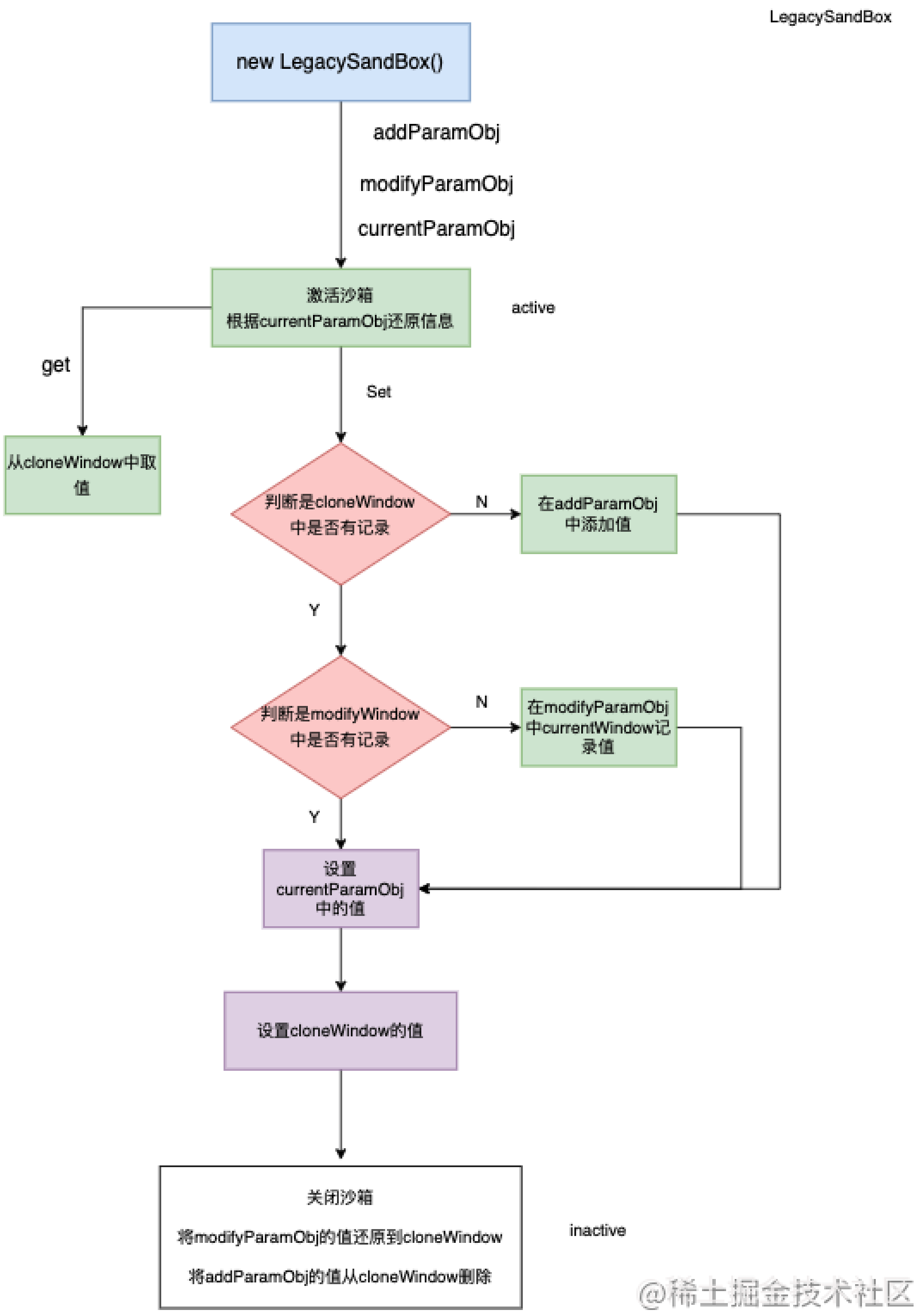
   - 沙箱种类
     iframe

     ```
     <iframe></iframe>
     ```

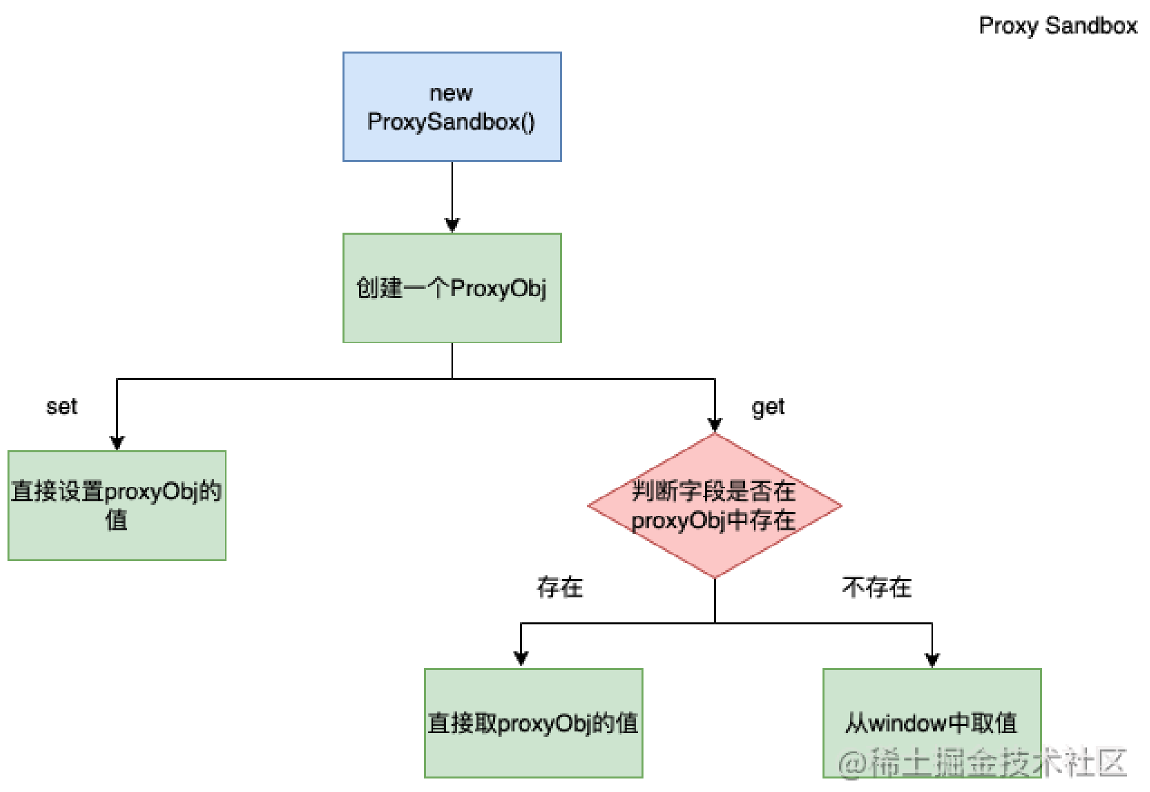
SnapshopSandbox(快照沙箱)



LegacySandbox（单例沙箱）


LegacySandBox
new LegacySandBox()
addParamObj
modifyParamObj
currentParamObj
激活沙箱
根据currentParamObj还原信息
active
get
Set
从cloneWindow中取值
判断是cloneWindow中是否有记录
N
在addParamObj中添加值
Y
判断是modifyWindow中是否有记录
N
在modifyParamObj中currentWindow记录值
Y
设置currentParamObj中的值
设置cloneWindow的值
关闭沙箱
将modifyParamObj的值还原到cloneWindow
将addParamObj的值从cloneWindow删除
inactive

ProxySandbox（多例沙箱）



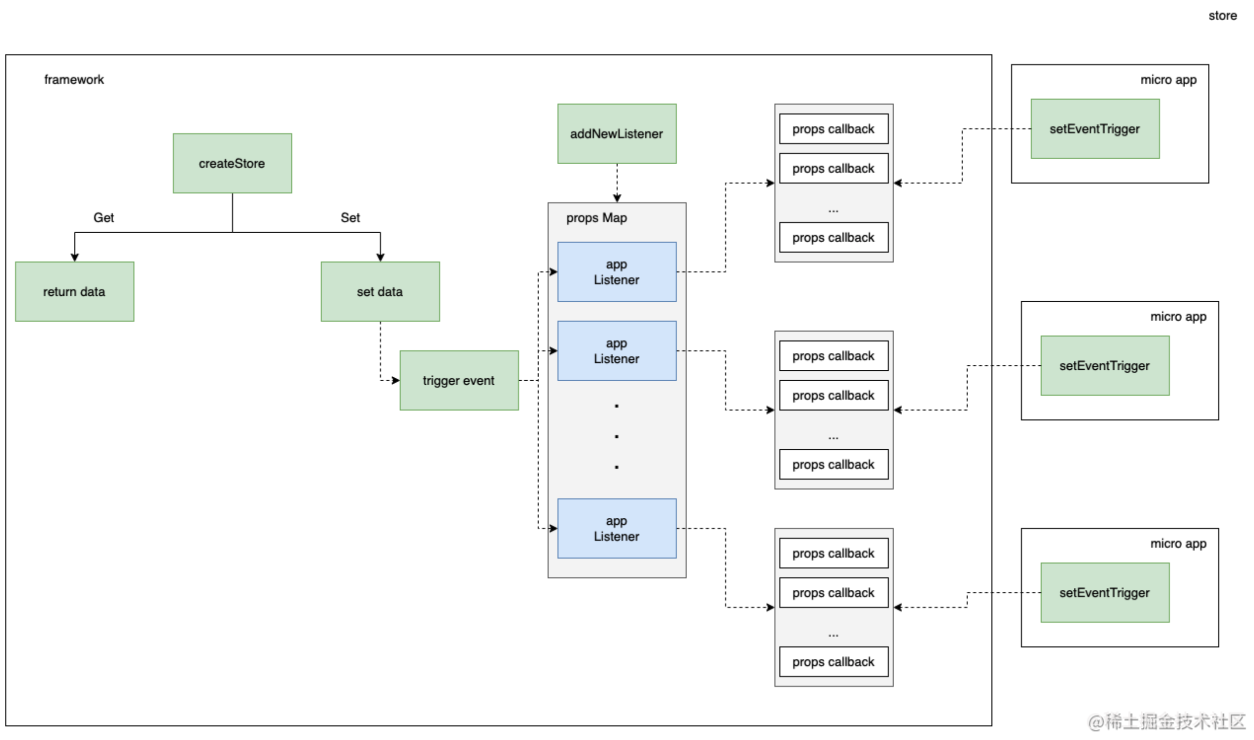
- 沙箱实现（多例沙箱）

```
interface SandBoxImplement {
  active: () => void;
  inActive: () => void;
}

type ProxyParam = Record<string, any> & Window;

/** 沙箱操作 */
class SandBox implements SandBoxImplement {
  public proxy: ProxyParam;
  private isSandboxActive: boolean;
  public name: string;

  /** 激活沙箱 */
  active() {
    this.isSandboxActive = true;
  }

  /** 关闭沙箱 */
  inActive() {
    this.isSandboxActive = false;
  }

  constructor(appName: string, context: Window & Record<string, any>)
{
    this.name = appName;
    this.isSandboxActive = false;
    const fateWindow = {};
    this.proxy = new Proxy(<ProxyParam>fateWindow, {
      set: (target, key, value) => {
        if (this.isSandboxActive) {
          target[<string>key] = value;
        }
        return true;
      },
      get: (target, key) => {
        if (target[<string>key]) {
          return target[<string>key];
        } else if (Object.keys(context).includes(<string>key)) {
          return context[<string>key];
        }

        return undefined;
      },
    });
  }
}

export default SandBox;
```

- 在沙箱中运行微应用

```
/** 注入环境变量 */
function injectEnvironmentStr(context: ProxyParam) {
  context[PRODUCT_BY_MICRO_FRONTEND] = true;
  context.__vite_plugin_react_preamble_installed__ = true;
  return true;
}

/** 执行js字符串 */
async function loadScriptByString(scripts: string[], context:
ProxyParam) {
  const scriptArr: Promise<Record<string, any>>[] = [];
  injectEnvironmentStr(context);
  scripts.forEach(async (val) => {
    scriptArr.push(
      await new Function(`
          return (window => {
            ${val}
            return window.middleVue;
          })(this)
      `).call(context)
    );
  });
  return scriptArr;
}
```

4. 全局通信状态管理



- 创建全局状态

```
/** 创建全局store */
export function createStore() {
  const globalStore = new Proxy(<Record<string, any>>{}, {
    get(target, key: string) {
      return target[key];
    },
    set(target, key: string, value) {
      const oldVal = target[key];
      target[key] = value;

      // 触发监听事件
      triggerEvent({ key, value, oldValue: oldVal });
      return true;
    },
  });
  return globalStore;
}
```

- 新增监听器

```
export type triggerEventParam<T> = {
  key: string;
  value: T;
  oldValue: T;
};
/** 监听对象 */
```

```
const listener: Map<
  string,
  Record<string, (data: triggerEventParam<any>) => void>
> = new Map();
/** 新增store监听器 */
export function addNewListener(appName: string) {
  if (listener.has(appName)) return;
  listener.set(appName, {});
}
```

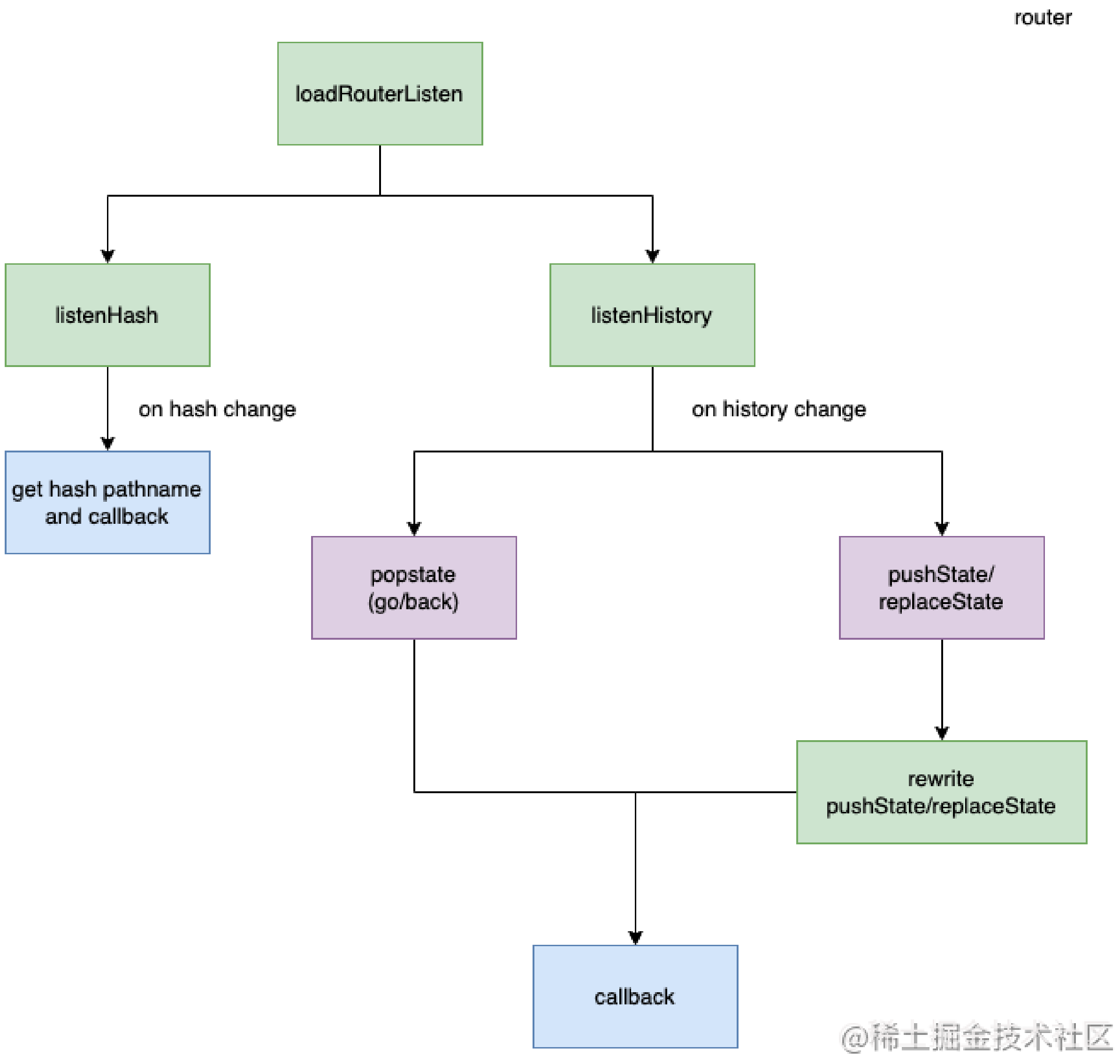
- 订阅事件

```
/** 设置监听事件 */
export function setEventTrigger<T extends any>(
  appName: string,
  key: string,
  callback: (data: triggerEventParam<T>) => void
) {
  if (listener.has(appName)) {
    const obj = listener.get(appName);
    if (obj) {
      obj[key] = callback;
    }
  }
}
```

- 触发事件

```
/** 改变字段值触发事件 */
export function triggerEvent<T extends any>(data:
triggerEventParam<T>) {
  listener.forEach((val) => {
    if (val[data.key] && typeof val[data.key] === 'function') {
      val[data.key](data);
    }
  });
}
```

5. 路由监听



- 监听 hash 路由变化回调

```
/** 监听hash路由变化 */
function listenHash(callback: listenCallback) {
  window.addEventListener('hashchange', (ev) => {
    callback(getHashPathName(ev.oldURL), getHashPathName(ev.newURL),
{});
  });
}

function getHashPathName(url: string) {
  const pathArr = url.split('#');
  return pathArr[1] ? `/${pathArr[1]}` : '/';
}
```

- 监听 history 路由变化回调

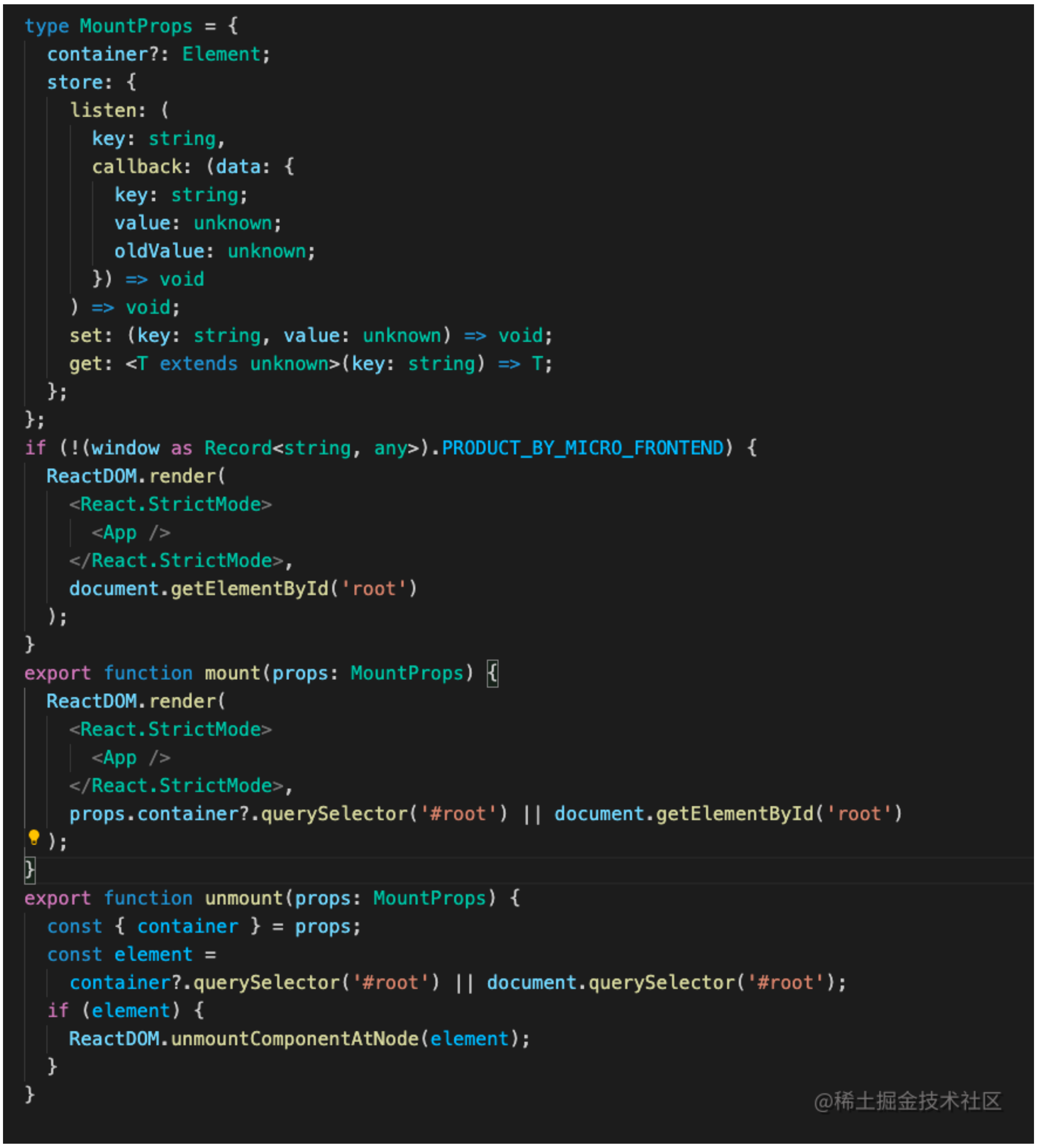
```
export type listenCallback = (
  oldPathName: string,
  pathName: string,
  param: any
) => void;
/** 监听history路由变化 */
function listenHistory(callback: listenCallback, currentRoute: string)
{
  window.history.pushState = historyControlRewrite('pushState',
callback);
  window.history.replaceState = historyControlRewrite(
    'replaceState',
    callback
  );
  window.addEventListener('popstate', (ev) => {
    callback(currentRoute, window.location.pathname, ev.state);
  });
}

// 重写pushState方法
const historyControlRewrite = function (
  name: 'pushState' | 'replaceState',
  callback: listenCallback
) {
  const method = history[name];
  return function (data: any, unused: string, url: string) {
    const oldPathName = window.location.pathname;
    if (oldPathName === url) return;
    method.apply(history, [data, unused, url]);
    callback(oldPathName, url || '', data);
  };
};
```

6. 样式隔离

    - 样式隔离解决方案
        1. 在微前端框架中获取 style 样式并添加唯一前缀
        2. 微应用中约定通过 postcss 处理
    - 利用 postcss 处理样式隔离

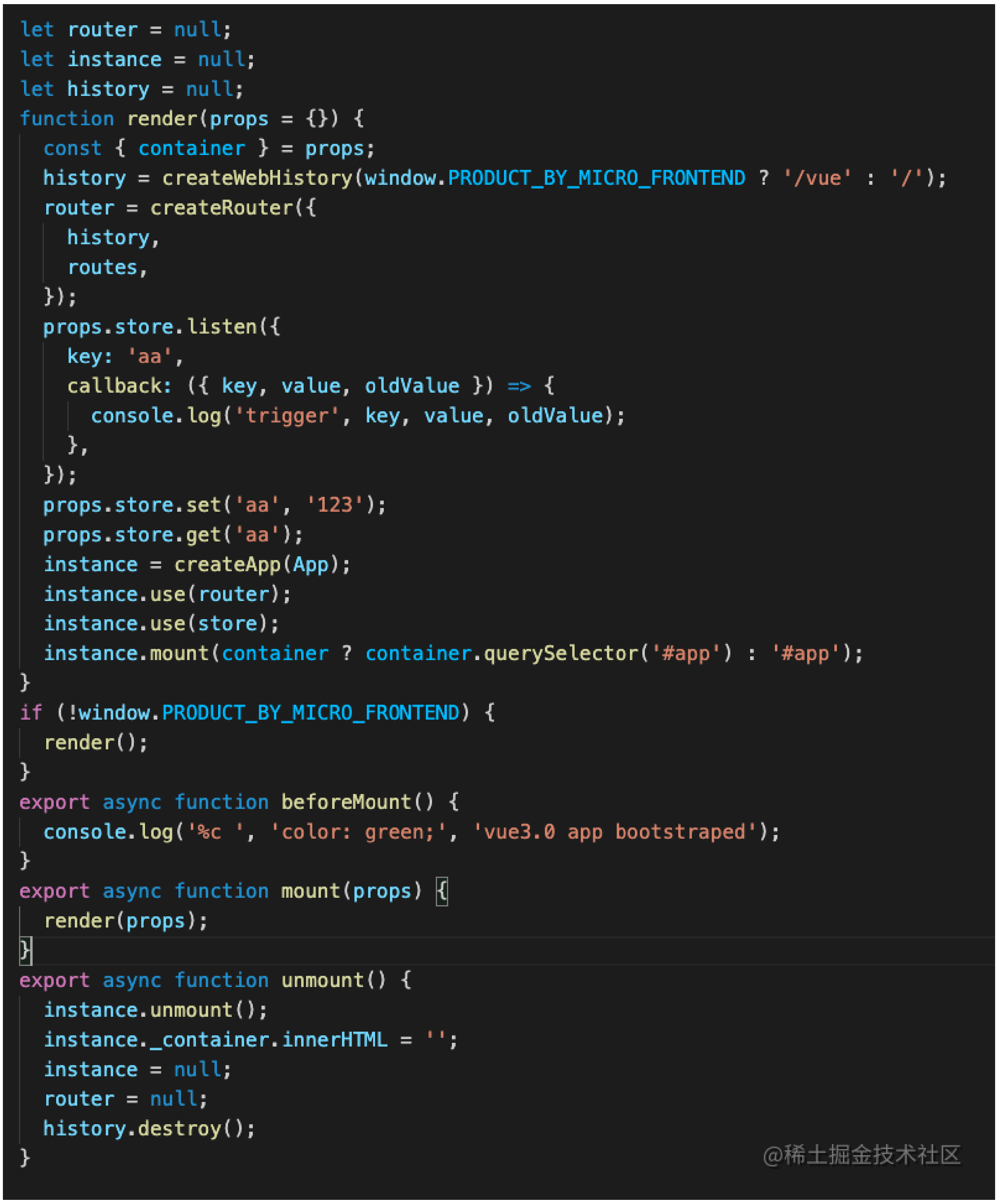
7. 改造旧应用

- React 改造



```
type MountProps = {
  container?: Element;
  store: {
    listen: (
      key: string,
      callback: (data: {
        key: string;
        value: unknown;
        oldValue: unknown;
      }) => void
    ) => void;
    set: (key: string, value: unknown) => void;
    get: <T extends unknown>(key: string) => T;
  };
};
if (!(window as Record<string, any>).PRODUCT_BY_MICRO_FRONTEND) {
  ReactDOM.render(
    <React.StrictMode>
      <App />
    </React.StrictMode>,
    document.getElementById('root')
  );
}
export function mount(props: MountProps) {
  ReactDOM.render(
    <React.StrictMode>
      <App />
    </React.StrictMode>,
    props.container?.querySelector('#root') || document.getElementById('root')
  );
}
export function unmount(props: MountProps) {
  const { container } = props;
  const element =
    container?.querySelector('#root') || document.querySelector('#root');
  if (element) {
    ReactDOM.unmountComponentAtNode(element);
  }
}
```

- Vue 改造



```
let router = null;
let instance = null;
let history = null;
function render(props = {}) {
  const { container } = props;
  history = createWebHistory(window.PRODUCT_BY_MICRO_FRONTEND ? '/vue' : '/');
  router = createRouter({
    history,
    routes,
  });
  props.store.listen({
    key: 'aa',
    callback: ({ key, value, oldValue }) => {
      console.log('trigger', key, value, oldValue);
    },
  });
  props.store.set('aa', '123');
  props.store.get('aa');
  instance = createApp(App);
  instance.use(router);
  instance.use(store);
  instance.mount(container ? container.querySelector('#app') : '#app');
}
if (!window.PRODUCT_BY_MICRO_FRONTEND) {
  render();
}
export async function beforeMount() {
  console.log('%c ', 'color: green;', 'vue3.0 app bootstraped');
}
export async function mount(props) {
  render(props);
}
export async function unmount() {
  instance.unmount();
  instance._container.innerHTML = '';
  instance = null;
  router = null;
  history.destroy();
}
```

小结

参考

1. Why Not Iframe
2. 字节跳动是如何落地微前端的
3. qiankun
4. singleSpa