# 子应用目录

|  |
| --- |
| ├── build/ # webpack 编译任务配置文件: 开发环境与生产环境  │ └── ...  ├── config/  │ ├── index.js # 项目核心配置  │ └── ...  ├ ── node\_module/ #项目中安装的依赖模块  ── src/  │ ├── main.js # 程序入口文件  │ ├── App.vue # 程序入口vue组件  │ ├── components/ # 组件  │ │ └── ...  │ └── assets/ # 资源文件夹，一般放一些静态资源文件  │ └── ...  ├── static/ # 纯静态资源 (直接拷贝到dist/static/里面)  ├── test/  │ └── unit/ # 单元测试  │ │ ├── specs/ # 测试规范  │ │ ├── index.js # 测试入口文件  │ │ └── karma.conf.js # 测试运行配置文件  │ └── e2e/ # 端到端测试  │ │ ├── specs/ # 测试规范  │ │ ├── custom-assertions/ # 端到端测试自定义断言  │ │ ├── runner.js # 运行测试的脚本  │ │ └── nightwatch.conf.js # 运行测试的配置文件  ├── .babelrc # babel 配置文件  ├── .editorconfig # 编辑配置文件  ├── .gitignore # 用来过滤一些版本控制的文件，比如node\_modules文件夹  ├── index.html # index.html 入口模板文件  └── package.json # 项目文件，记载着一些命令和依赖还有简要的项目描述信息  └── README.md #介绍自己这个项目的，可参照github上star多的项目。  build/ |

# 通信

在开始介绍 qiankun 的应用通信之前，我们需要先了解微前端架构如何划分子应用。

在微前端架构中，我们应该按业务划分出对应的子应用，而不是通过功能模块划分子应用。这么做的原因有两个：

1. 在微前端架构中，子应用并不是一个模块，而是一个独立的应用，我们将子应用按业务划分可以拥有更好的可维护性和解耦性。
2. 子应用应该具备独立运行的能力，应用间频繁的通信会增加应用的复杂度和耦合度。

综上所述，我们应该从业务的角度出发划分各个子应用，尽可能减少应用间的通信，从而简化整个应用，使得我们的微前端架构可以更加灵活可控。

我们本次教程将介绍两种通信方式，

1. 第一种是 qiankun 官方提供的通信方式 - Actions 通信，适合业务划分清晰，比较简单的微前端应用，一般来说使用第一种方案就可以满足大部分的应用场景需求。
2. 第二种是基于 localStorage、sessionStorage实现的通信方式 - Storage 通信，适合需要跟踪通信状态，类似用登录信息、token等，子应用具备独立运行能力。
3. 第三种是基于props传值，用于主应用给子应用传值。适用于主子应用共享组件、公共方法调用等。

## Actions 通信

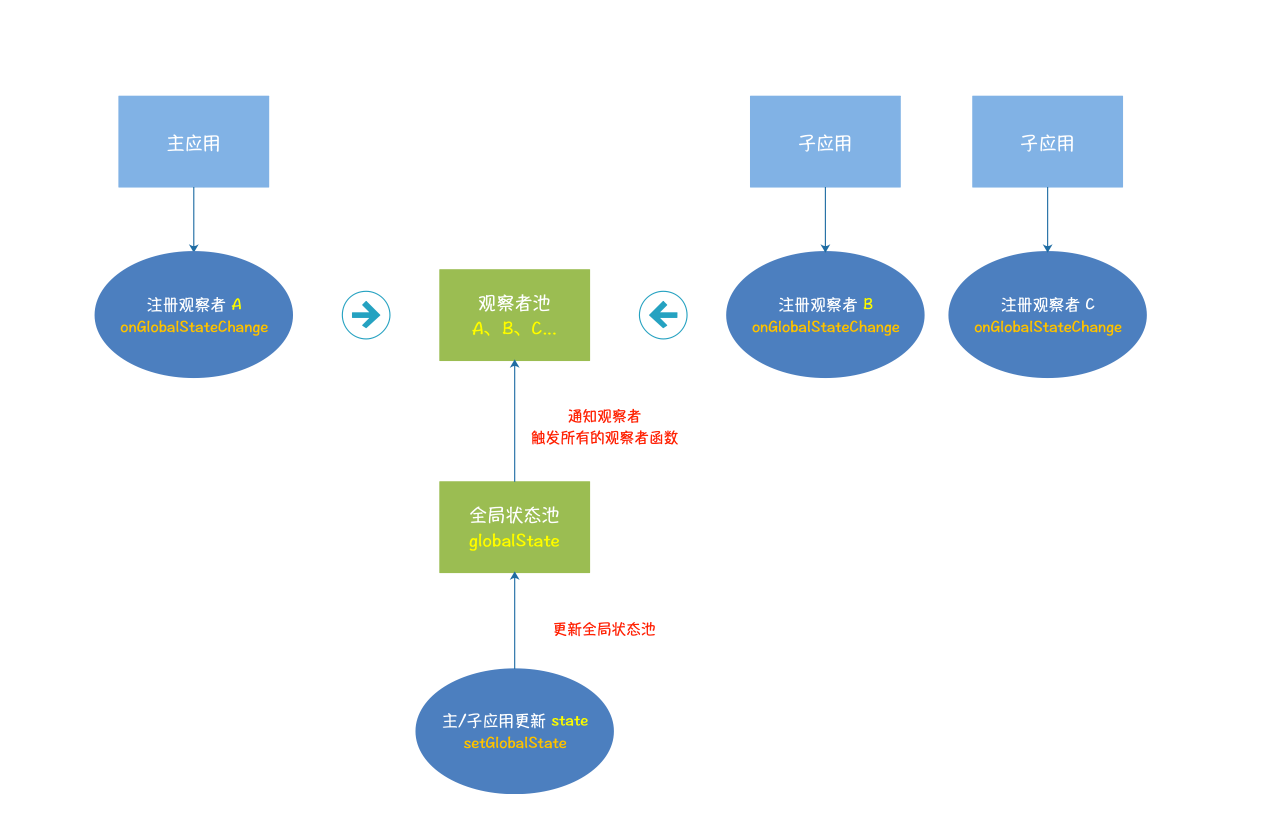
我们先介绍官方提供的应用间通信方式 - Actions 通信，这种通信方式比较适合业务划分清晰，应用间通信较少的微前端应用场景。

### 通信原理

qiankun 内部提供了 initGlobalState 方法用于注册 MicroAppStateActions 实例用于通信，该实例有三个方法，分别是：

* setGlobalState：设置 globalState - 设置新的值时，内部将执行 浅检查，如果检查到 globalState 发生改变则触发通知，通知到所有的 观察者 函数。
* onGlobalStateChange：注册 观察者 函数 - 响应 globalState 变化，在 globalState 发生改变时触发该 观察者 函数。
* offGlobalStateChange：取消 观察者 函数 - 该实例不再响应 globalState 变化。

我们来画一张图来帮助大家理解（见下图）



我们从上图可以看出，我们可以先注册 观察者 到观察者池中，然后通过修改 globalState 可以触发所有的 观察者 函数，从而达到组件间通信的效果。

### Api

<https://qiankun.umijs.org/zh/api#initglobalstatestate>

理解：这里的全局状态和vuex中的state十分类似，不同的是全局状态可以在主应用和微应用中共享

### 简单实验

#### 主应用

新建src->actions.js

import { initGlobalState } from "qiankun";

import store from "./store";

const initialState = {

//这里写初始化数据

};

// 初始化 state

const actions = initGlobalState(initialState);

actions.onGlobalStateChange((state, prev) => {//监听公共状态的变化

console.log("主应用: 变更前");

console.log(prev);

console.log("主应用: 变更后");

console.log(state);

store.commit('setProject',state);//这里我把公共状态存到主应用的vuex里了

});

export default actions;

在组件中使用actions

tab.vue

<template>

<div>

<button @click="sendMes1">点击向子应用发送消息1</button>

<button @click="sendMes2">点击向子应用发送消息2</button>

<p>当前显示的项目: {{project}} </p>

</div>

</template>

jindei

<script>

import actions from './actions'//记得导入actions实例

export default {

data() {

return {

mes1: { project\_id: '项目1' },

mes2: { project\_id: '项目2' },

}

},

computed:{

project:function(){

return this.$store.state.project\_id

}

},

methods: {

sendMes1() {

actions.setGlobalState(this.mes1);//通过setGlobalState改变全局状态

},

sendMes2() {

actions.setGlobalState(this.mes2);

}

},

}

</script>

#### 微应用

新建src->actions.js

function emptyAction() { //设置一个actions实例

// 提示当前使用的是空 Action

console.warn("Current execute action is empty!");

}

class Actions {

// 默认值为空 Action

actions = {

onGlobalStateChange: emptyAction,

setGlobalState: emptyAction,

};

/\*\*

\* 设置 actions

\*/

setActions(actions) {

this.actions = actions;

}

/\*\*

\* 映射

\*/

onGlobalStateChange(...args) {

return this.actions.onGlobalStateChange(...args);

}

/\*\*

\* 映射

\*/

setGlobalState(...args) {

return this.actions.setGlobalState(...args);

}

}

const actions = new Actions();

export default actions;

main.js

在mounted的生命周期里注入actions实例

export async function mount(props) {

actions.setActions(props); //注入actions实例

render(props);

}

在vue 组件中使用

1.vue

<template>

<div>

<div>这是子应用</div>

<p>接收到的消息： {{mes}}</p>

<button @click= "butClick">点击向父应用发送消息</button>

</div>

</template>

<script>

import actions from '../actions'//导入实例

export default {

data() {

return {

mes: '',

}

},

mounted() {

actions.onGlobalStateChange((state) => { //监听全局状态

this.mes = state

}, true);

},

methods:{

butClick(){

actions.setGlobalState({ project\_id: '项目99'})//改变全局状态

}

}

}

</script>

## Storage通信

### 通信原理

storage 命名空间,具体见portal/src/utils/storageNameSpace.js

* **主应用中**

存值 sesstionStorage.setItem('a', 'b', true)

// 会自动在key上添加 `主应用:a` 作为key

取值 sesstionStorage.getItem('a', true)

清除 值 sessionStorage.removeItem('a', true)

清空

sessionStorage.clear(true) // 清除所有sessionStorage 包括子应用的

sessionStorage.clear('self') // 清除主应用的 sessionStorage

* **子应用中**

存值 sessionStorage.setItem('a', 'b')

// 会自动在key上添加 '应用名:a' 作为key

取值 sessionStorage.getItem('a')

清除 值 sessionStorage.removeItem('a')

清空 sessionStorage.clear()

// 只清空当前应用的 sessionStorage获取其他子应用的 sessionStorage

sessionStorage.getItem(子应用名: + key)

localStorage 与sessionStorage一样

## props通信

在上述所建微前端应用中，父子间的通信是极其普遍且无法绕过的需求，而 qiankun 在这方面当然有所考虑。

### 父应用

在主应用 portal/src/singleSpa.js中注册子应用时，将定义好的msg通过props参数传递给子应用。

|  |
| --- |
| // 定义传入子应用的数据  const msg = {  data: {  auth: false  },  fns: {  portal\_alert(txt) {  // txt 子应用传递的值  alert('父应用的方法:' + txt)  },  portal\_logout() {  parentThat.$store.dispatch('fedLogOut').then(() => {  const fullPath = parentThat.$route.fullPath  const path = fullPath ? `/portal-login?redirect=${fullPath}` : '/portal-login'  parentThat.$router.push({ path })  ***location***.reload() // 为了重新实例化vue-router对象 避免bug  })  }  }  // 注册子应用 registerMicroApps([  {  name: 'app1',  entry: '//localhost:7771',  ***render***,  activeRule: genActiveRule('/app1'),  props: msg, // 将定义好的数据传递给子应用  }, ]); |

### 子应用

在 子应用的main.js的bootstrap 函数里将接收到的 props 参数内的函数挂在 vue 原型上方便使用，你也可以在其他导出的生命周期函数内得到 props 并按照你的设想去处理。

|  |
| --- |
| export async function bootstrap (props = {}) {  // console.log('sub-app1 加载中')  // 父应用传递的值 挂载vue原型上  Vue.prototype.parentData = {...props.data}  // 父应用传递的 方法 挂载原型上  Vue.prototype.parentFns = props.fns } |