构建single-spa应用

Building single-spa applications

A single-spa registered application is everything that a normal SPA is, except that it doesn't have an HTML page. In a single-spa world, your SPA contains many registered applications, where each has its own framework. Registered applications have their own client-side routing and their own frameworks/libraries. They render their own html and have full freedom to do whatever they want, whenever they are mounted. The concept of being mounted refers to whether a registered application is putting content on the DOM or not. What determines if a registered application is mounted is its [activity function](https://single-spa.js.org/docs/configuration#activity-function). Whenever a registered application is not mounted, it should remain completely dormant until mounted.

single-spa 应用与普通的单页面是一样的，只不过它没有HTML页面。在一个single-spa中，你的SPA包含许多被注册的应用，而各个应用可以使用不同的框架。被注册的这些应用维护自己的客户端路由，使用自己需要的框架或者类库。应用只要通过挂载，便可渲染自己的html页面，并自由实现功能。“挂载”(mounted)的概念指的是被注册的应用内容是否已展示在DOM上。我们可通过应用的[激活函数](https://zh-hans.single-spa.js.org/docs/configuration#activity-function)来判断其是否已被挂载。应用在未挂载之前，会一直保持休眠状态。

##### 创建并注册一个应用程序

[#](https://single-spa.js.org/docs/building-applications#creating-a-registered-application)Creating a registered application

To create a registered application, first [register the application with single-spa](https://single-spa.js.org/docs/configuration#registering-applications). Once registered, the registered application must correctly implement all of the following lifecycle functions inside of its main entry point.

要添加一个应用，首先需要[注册该应用](https://zh-hans.single-spa.js.org/docs/configuration#registering-applications)。一旦应用被注册后，必须在其入口文件(entry point)实现下面提到的各个生命周期函数。

##### 注册应用的生命周期

[#](https://single-spa.js.org/docs/building-applications#registered-application-lifecycle)Registered application lifecycle

During the course of a single-spa page, registered applications are loaded, bootstrapped (initialized), mounted, unmounted, and unloaded. single-spa provides hooks into each phase via  `lifecycles`.

A lifecycle function is a function or array of functions that single-spa will call on a registered application. single-spa calls these by finding specific named exports from the registered application's main file.

Notes:

1. Implementing `bootstrap`, `mount, and `unmount` is required. But implementing `unload` is optional.

2、Each lifecycle function must either return a `promise` or be an  `async function`

3、If an array of functions is exported (instead of just one function), the functions will be called one-after-the-other, waiting for the resolution of one function's promise before calling the next.

4、If single-spa is [not started](https://single-spa.js.org/docs/api#start), applications will be loaded, but will not be bootstrapped, mounted or unmounted.

Note

Framework-specific helper libraries exist in the [single-spa ecosystem](https://single-spa.js.org/docs/ecosystem) to implement these required lifecycle methods. This documentation is helpful for understanding what those helpers are doing, or for implementing your own.

在一个 single-spa 页面，注册的应用会经过下载(loaded)、初始化(initialized)、被挂载(mounted)、卸载(unmounted)和unloaded（被移除）等过程。single-spa会通过“生命周期”为这些过程提供钩子函数。

生命周期函数是 single-spa 在注册的应用上调用的一系列函数，single-spa 会在各应用的主文件中，查找对应的函数名并进行调用。

注:

1、`bootstrap`, `mount, and `unmount` 的实现是必须的，但`unload`则是可选的

2、生命周期函数必须有返回值，可以是Promise或者 async函数

3、如果导出的是函数数组而不是单个函数，这些函数会被依次调用，对于promise函数，会等到resolve之后再调用下一个函数

4、如果 single-spa [未启动](https://zh-hans.single-spa.js.org/docs/api#start)，各个应用会被下载，但不会被初始化、挂载或卸载。

说明：

在[single-spa 生态系统](https://zh-hans.single-spa.js.org/docs/ecosystem)中有各个主流框架对于生命周期函数的实现，这些文档有助于理解这些helper执行的操作，也有助于你自己实现生命周期函数。

##### 生命周期参数

[#](https://single-spa.js.org/docs/building-applications#lifecyle-props)Lifecyle props

Lifecycle functions are called with a `props` argument, which is an object with some guaranteed information and additional custom information.

function bootstrap(props) {

const {

name, // The name of the application

singleSpa, // The singleSpa instance

mountParcel, // Function for manually mounting

customProps // Additional custom information

} = props; // Props are given to every lifecycle

return Promise.resolve();

}

生命周期函数使用"props" 传参，这个对象包含single-spa相关信息和其他的自定义属性

function bootstrap(props) {

const {

name, // 应用名称

singleSpa, // singleSpa实例

mountParcel, // 手动挂载的函数

customProps // 自定义属性

} = props; // Props 会传给每个生命周期函数

return Promise.resolve();

}

###### 内置参数

[#](https://single-spa.js.org/docs/building-applications#built-in-props)Built-in props

Each lifecycle function is guranteed to be called with the following props:

1、`name` : The string name that was registered to single-spa.

1. singleSpa : A reference to the singleSpa instance, itself. This is intended to allow applications and helper libraries to call singleSpa APIs without having to import it. This is useful in situations where there are multiple webpack configs that are not set up to ensure that only one instance of singleSpa is loaded.
2. mountParcel: The [mountParcel function](https://single-spa.js.org/docs/parcels-api.html#mountparcel).

每个生命周期函数的入参都会保证有如下参数：

1、 name : 注册到 single-spa 的应用名称

2、singleSpa: 对singleSpa 实例的引用, 方便各应用和类库调用singleSpa提供的API时不再导入它。 可以解决有多个webpack配置文件构建时无法保证只引用一个singleSpa实例的问题。

3、mountParcel: [mountParcel 函数](https://zh-hans.single-spa.js.org/docs/parcels-api.html#mountparcel).

###### 自定义参数

[#](https://single-spa.js.org/docs/building-applications#custom-props)Custom props

In addition to the built-in props that are provided by single-spa, you may optionally specify custom props to be passed to an application by providing a fourth argument to

`registerApplication`. These customProps will be passed into each lifecycle method.

//root.application.js

singleSpa.registerApplication(

'app1',

() => {},

() => {},

{ authToken: "d83jD63UdZ6RS6f70D0" }

);

//app1.js

export function mount(props) {

console.log(props.customProps.authToken);

// do something with the common authToken in app1

return reactLifecycles.mount(props);

}

Some use cases could be to:

* share a common access token with all child apps
* pass down some initialization information, like the rendering target
* pass a reference to a common event bus so each app may talk to each other

Note that when no customProps are provided during registration, `props.customProps` defaults to an empty object.

除single-spa提供的内置参数外，还可以指定自定义参数，在调用各个生命周期函数时传入。指定方法是在调用`registerApplication`时，传入第4个参数.

//root.application.js

singleSpa.registerApplication(

'app1',

() => {},

() => {},

{ authToken: "d83jD63UdZ6RS6f70D0" }

);

//app1.js

export function mount(props) {

console.log(props.customProps.authToken);

// 可以在 app1 中获取到authToken参数

return reactLifecycles.mount(props);

}

可能使用到的场景：

* 各个应用共享一个公共的 access token
* 下发初始化信息，如渲染目标
* 传递对事件总线（event bus）的引用，方便各应用之间进行通信

注意如果没有提供自定义参数，则 `props.customProps` 默认会返回一个空对象。

##### 生命周期帮助类

[#](https://single-spa.js.org/docs/building-applications#lifecycle-helpers)Lifecycle helpers

Some helper libraries that implement lifecycle functions for ease of use are available for many popular frameworks/libraries. Learn more on the [Ecosystem page](https://single-spa.js.org/docs/ecosystem).

有一些帮助类库会对针对主流框架的生命周期函数进行实现以方便使用。具体可参见[生态页面](https://single-spa.js.org/docs/ecosystem)。

###### 加载

[#](https://single-spa.js.org/docs/building-applications#load)Load

When registered applications are being lazily loaded, this refers to when the code for a registered application is fetched from the server and executed. This will happen once the registered application's [activity function](https://single-spa.js.org/docs/configuration#activity-function) returns a truthy value for the first time. It is best practice to do as little as possible / nothing at all during `load`, but instead to wait until the bootstrap lifecycle function to do anything. If you need to do something during `load` , simply put the code into a registered application's main entry point, but not inside of an exported function. For example:

注册的应用会被懒加载，这指的是该应用的代码会从服务器端下载并执行。注册的应用在[激活函数](https://zh-hans.single-spa.js.org/docs/configuration#activity-function) 第一次返回真值(truthy value)时，下载动作会发生。在下载过程中，建议尽可能执行少的操作，可以在 `bootstrap`生命周期之后再执行各项操作。若确实有在下载时需要执行的操作，可将代码放入子应用入口文件中，但要放在各导出函数的外部。例如：

console.log("The registered application has been loaded!");

export async function bootstrap(props) {...}

export async function mount(props) {...}

export async function unmount(props) {...}

###### 初始化

[#](https://single-spa.js.org/docs/building-applications#bootstrap)Bootstrap

This lifecycle function will be called once, right before the registered application is mounted for the first time.

这个生命周期函数会在应用第一次挂载前执行一次。

export function bootstrap(props) {

return Promise

.resolve()

.then(() => {

// One-time initialization code goes here

console.log('bootstrapped!')

});

}

###### 挂载

[#](https://single-spa.js.org/docs/building-applications#mount)Mount

This lifecycle function will be called whenever the registered application is not mounted, but its [activity function](https://single-spa.js.org/docs/configuration#activity-function) returns a truthy value. When called, this function should look at the URL to determine the active route and then create DOM elements, DOM event listeners, etc. to render content to the user. Any subsequent routing events (such as `hashchange` and `popstate`) will not trigger more calls to `mount`, but instead should be handled by the application itself.

每当应用的[激活函数](https://zh-hans.single-spa.js.org/docs/configuration#activity-function)返回真值，但该应用处于未挂载状态时，挂载的生命周期函数就会被调用。调用时，函数会根据URL来确定当前被激活的路由，创建DOM元素、监听DOM事件等以向用户呈现渲染的内容。任何子路由的改变（如`hashchange`或popstate等）不会再次触发`mount`，需要各应用自行处理。

export function mount(props) {

return Promise

.resolve()

.then(() => {

// Do framework UI rendering here

console.log('mounted!')

});

}

###### 卸载

[#](https://single-spa.js.org/docs/building-applications#unmount)Unmount

This lifecycle function will be called whenever the registered application is mounted, but its [activity function](https://single-spa.js.org/docs/configuration#activity-function) returns a falsy value. When called, this function should clean up all DOM elements, DOM event listeners, leaked memory, globals, observable subscriptions, etc. that were created at any point when the registered application was mounted.

每当应用的[激活函数](https://zh-hans.single-spa.js.org/docs/configuration#activity-function)返回假值，但该应用已挂载时，卸载的生命周期函数就会被调用。卸载函数被调用时，会清理在挂载应用时被创建的DOM元素、事件监听、内存、全局变量和消息订阅等。

export function unmount(props) {

return Promise

.resolve()

.then(() => {

// Do framework UI unrendering here

console.log('unmounted!');

});

}

###### 移除

[#](https://single-spa.js.org/docs/building-applications#unload)Unload

The `unload` lifecycle is an optionally implemented lifecycle function. It will be called whenever an application should be `unloaded`. This will not ever happen unless someone calls the `unloadApplication` API. If a registered application does not implement the unload lifecycle, then it assumed that unloading the app is a no-op.

The purpose of the `unload` lifecycle is to perform logic right before a single-spa application is unloaded. Once the application is unloaded, the application status will be NOT\_LOADED and the application will be re-bootstrapped.

The motivation for `unload` was to implement the hot-loading of entire registered applications, but it is useful in other scenarios as well when you want to re-bootstrap applications, but perform some logic before applications are re-bootstrapped.

“移除”生命周期函数的实现是可选的，它只有在[unloadApplication](https://zh-hans.single-spa.js.org/docs/api#unloadapplication)被调用时才会触发。如果一个已注册的应用没有实现这个生命周期函数，则假设这个应用无需被移除。

移除的目的是各应用在移除之前执行部分逻辑，一旦应用被移除，它的状态将会变成NOT\_LOADED，下次激活时会被重新初始化。

移除函数的设计动机是对所有注册的应用实现“热加载”，不过在其他场景中也非常有用，比如想要重新初始化一个应用，且在重新初始化之前执行一些逻辑操作时。

export function unload(props) {

return Promise

.resolve()

.then(() => {

// Hot-reloading implementation goes here

console.log('unloaded!');

});

}

##### 超时

[#](https://single-spa.js.org/docs/building-applications#timeouts)Timeouts

By default, registered applications obey the [global timeout configuration](https://single-spa.js.org/docs/api#setbootstrapmaxtime), but can override that behavior for their specific application. This is done by exporting a `timeouts` object from the main entry point of the registered application. Example:

默认情况下，所有注册的应用遵循[全局超时配置](https://zh-hans.single-spa.js.org/docs/api#setbootstrapmaxtime)，但对于每个应用，也可以通过在主入口文件导出一个 `timeouts`对象来重新定义超时时间。如：

//app-1.main-entry.js

export function bootstrap(props) {...}

export function mount(props) {...}

export function unmount(props) {...}

export const timeouts = {

bootstrap: {

millis: 5000,

dieOnTimeout: true,

warningMillis: 2500,

},

mount: {

millis: 5000,

dieOnTimeout: false,

warningMillis: 2500,

},

unmount: {

millis: 5000,

dieOnTimeout: true,

warningMillis: 2500,

},

unload: {

millis: 5000,

dieOnTimeout: true,

warningMillis: 2500,

},

};

Note that `millis` refers to the number of milliseconds for the final console warning, and `warningMillis` refers to the number of milliseconds at which a warning will be printed to the console (on an interval) leading up to the final console warning.

注意 `millis`指的是最终控制台输出警告的毫秒数，`warningMillis`指的是将警告打印到控制台(间隔)的毫秒数。

##### 切换应用时的过渡效果

[#](https://single-spa.js.org/docs/building-applications#transitioning-between-applications)Transitioning between applications

If you find yourself wanting to add transitions as applications are mounted and unmounted, then you'll probably want to tie into the `bootstrap`, `mount`, and `unmoun`lifecycle methods. This [single-spa transitions](https://github.com/frehner/singlespa-transitions) repo is a small proof-of-concept of how you can tie into these lifecycle methods to add transitions as your apps mount and unmount.

Transitions for pages within a mounted application can be handled entirely by the application itself. For example, using [react-transition-group](https://github.com/reactjs/react-transition-group) for React-based projects.

如果你想为应用在挂载和卸载时加一些过渡效果(动画效果等)，则需要将其和`bootstrap`, `mount` 和 `unmount`等生命周期函数关联。这个[single-spa 过渡](https://github.com/frehner/singlespa-transitions)仓库是个小demo，展示了生命周期之间切换时如何过渡。

对于已经挂载的应用，各个页面之间的过渡效果可由应用本身自行处理，如基于React创建的项目可使用using [react-transition-group](https://github.com/reactjs/react-transition-group)实现过渡效果。