Membership and Analysis Union Package Workflow detail code reference

Note

1，需要稍微了解一下react redux(简易流程图介绍），js 异步操作，闭包等基础知识；

2，项目代码 :national-hub， union，页面例子：registry

3，为了更好的解释说明，每一步讲解前面都会放对应github上的代码链接，请分屏打开同时看，更方便理解

4，如只想了解reportSignUp被cancel的情况请直接点击大纲跳 “用户注册”

1，初始化

1.1，store初始化 code link

Store，就是保存数据的地方，可以看成一个容器，而整个页面应用只能有一个store，用来管理页面上那些公有状态。

通过 createStore这个redux提供的方法来生成store；

```
export default createStore(reducer, initialState, enhancers)
```

第一个参数 reducer在此处被声明

```
const reducer = combineReducers({
 membership: membershipReducer,
 conversations: conversationsReducer,
 membershipModal: membershipModalReducer
})
```

当*store*接受到*action*(改变*store*的*state*的通知）后，重新给出一个新的*state*，而这种*state*的计算过程就叫*reducer*。

combineReducers是redux提供的，用于reducer的拆分，也是为了代码的可读性和可扩展性，每个子reducer负责不同的业务逻辑，membership是处理用户登陆状态的，conversations暂时也不知道是什么业务场景的，membershipModal就是用户登陆注册表单的一些业务场景；

第二个参数initialState，初始化store的state，会覆盖reducer函数的默认初始值

```
const initialState = {}
```

第三个参数 enhancers 就是中间件

中间件其实就是一个函数，对*store.dispatch*(用于发送*action*的）方法进行了改造，在发出*action*和和执行*reducer*这两步之间，添加了其他功能（这里不过多介绍，可以自行了解中间件）

页面enhancers 初始化如下：

```
const enhancers = composeEnhancers(
 applyMiddleware(
   membershipMiddleware(),
   identifyMemberMiddleware(),
   conversationsMiddleware(),
   thunk
 )
)
```

看代码会发现这里有个判断

```
const composeEnhancers =
g.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose
```

这一步是判断了项目是否配置了Redux Dev Tools (用于查看redux状态变化的开发工具），其会占用createStore的第二个参数，为了两个同时可用加的判断（可自行了解）；

applyMiddleware是redux的原生方法，用于将所有中间件组成一个数组，依次执行，

membershipMiddleware：用于整体判断用户是否注册或登陆成功及其对应的操作，页面默认的注册成功重定向操作就是在这进行；

indetifMemberMiddleware: 用于验证用户会话是否连接成功，如果成功则发出i-call；

conversationsMiddleware: 应该也是用于会话验证及对应操作，具体业务暂时不知；

thunk：store.dispatch方法默认情况下，参数只能是对象（action)，而不能是函数；而thunk就是改造了该方法，使其能接受函数作为参数，来实现异步操作，也是非常重要的中间件；

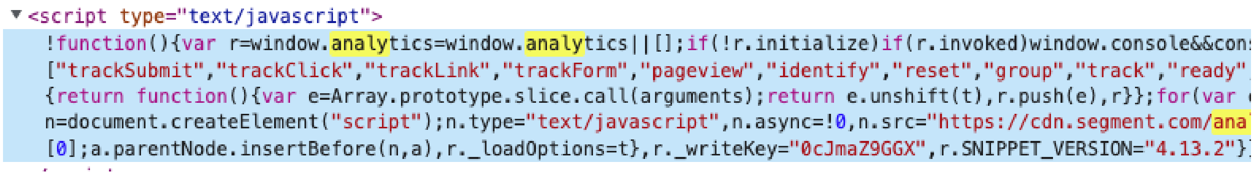
1.2，analytics初始化

在next构建运行中，先生成初始化对应的analytics代码片段
https://github.com/tkww/national-hub/blob/main/next.config.js

```
createDefaultSegmentSnippet: createSegmentSnippet({
  writeKey: segmentDefaultKey,
  minify: true
}),
createRegistrySegmentSnippet: createSegmentSnippet({
  writeKey: segmentRegistryKey,
  minify: true
}),
```

然后插入到页面head标签里
https://github.com/tkww/national-hub/blob/main/pages/_document.js

生成这样一段代码，其把analytics对象挂载到window上



```
▼<script type="text/javascript">
    !function(){var r=window.analytics=window.analytics||[];if(!r.initialize)if(r.invoked)window.console&&con
    ["trackSubmit","trackClick","trackLink","trackForm","pageview","identify","reset","group","track","ready"
    {return function(){var e=Array.prototype.slice.call(arguments);return e.unshift(t),r.push(e),r}};for(var
    n=document.createElement("script");n.type="text/javascript",n.async=!0,n.src="https://cdn.segment.com/ana
    [0];a.parentNode.insertBefore(n,a),r._loadOptions=t},r._writeKey="0cJmaZ9GGX",r.SNIPPET_VERSION="4.13.2"}
```

P-call发出

https://github.com/tkww/national-hub/blob/main/src/apps/registry/initializer/init-analytics.js

```
if(window.analytics) {
  window.analytics.page({
    resolution: getResolutionTrack()
  })
}
```

1.3，membershipModal

把该页面store通过redux提供的connect方法，和membershipModalRedux连接到一起，这是一个高阶组件，即传入一个组件，返回一个新组件

```
export default connect(mapStateToProps, mapDispatchToProps,
mergeProps)(MembershipModalWrapper)
```

第一个参数 mapStateToProps，接受store的state，然后作为props传给组件，每当store的state发生变化，该函数都会自动执行，从而触发ui组件的重新渲染，比如点击打开membershipModal操作

如下是其在项目里的声明代码块：

```
const mapStateToProps = (state) => {
 const { membershipModal = null } = state
 if(!membershipModal) return null

 if(membershipModal) {
   const { openModalType, redirect, signUpView } = membershipModal

   return { openModalType, redirect, signUpView }
 }
}
```

第二个参数mapDispatchToProps, 定义了ui组件怎样发出action传给store

```
const mapDispatchToProps = {
 onClickSignUp: openSignUp,
 onClickLogIn: openLogIn,
 onClose: close,
 onLogInSuccess: close,
 onSignUpSuccess: close
}
```

第三个参数mergeProps，会将mapStateToProps与mapDispatchToProps执行结果结合组件本身props, 传入到该回调函数内，然后作为props传给membershipModalRedux，也是在该函数内对onSignUpSuccess等回调函数进行拓展，来进行重定向等操作

```
const mergeProps = (stateProps, dispatchProps, ownProps) => ({
 ...ownProps,
 ...stateProps,
 ...dispatchProps,
 onClickSignUp: (event) => {
   dispatchProps.onClickSignUp(event, stateProps.redirect,
stateProps.signUpView)
 },
 onClickLogIn: event => dispatchProps.onClickLogIn(event,
stateProps.redirect),
 onLogInSuccess: () => {
   if(stateProps.redirect && stateProps.redirect.login) {
     window.location = stateProps.redirect.login
   }

   dispatchProps.onClose()
 },
 onSignUpSuccess: () => {
   if(stateProps.redirect && stateProps.redirect.signUp) {
     setTimeout(() => {
       window.location = stateProps.redirect.signUp
     }, 500)
   }

   dispatchProps.onClose()
 },
 onClose: () => {
   dispatchProps.onClose()
 }
```

```
})
```

上面的onSignUpSuccess就会被作为props传入membershipModalRedux

```
<MembershipModal
  {...props}
  openModal={openModalType}
  renderMembershipForm={formProps => (
    <MembershipFormRedux
      onSignUpSuccess={onSignUpSuccess}
      onLogInSuccess={onLogInSuccess}
      {...formProps}
    />
  )}
/>
```

2，membershipModal code 逻辑解析

当我们点页面上signup按钮时，会发生以下步骤：

1，触发openSignUp这一个dispatch，发送一个action（type为 membership-modal/OPEN_SIGN_UP），告诉store 准备打开 membershipModal的通知
https://github.com/tkww/national-hub/blob/main/src/share/Layout/components/MembershipModal/store.js

```
const OPEN_SIGN_UP = 'membership-modal/OPEN_SIGN_UP'
export const openSignUp = (event, redirect, signUpView) => ({
 type: OPEN_SIGN_UP,
 redirect,
 signUpView
})
```

2, store接受到对应的action，然后触发对应的reducer进行计算，返回了一个新的state（openModalType为‘SIGN_UP’）；该reducer，在前面初始化时已经传入，即membershipReducer
https://github.com/tkww/national-hub/blob/main/src/share/Layout/components/MembershipModal/store.js

```
export default (state = initialState, action) => {
 switch(action.type) {
   case OPEN_SIGN_UP:
     return { openModalType: 'SIGN_UP', redirect: action.redirect,
signUpView: action.signUpView }
   case OPEN_LOG_IN:
     return { openModalType: 'LOG_IN', redirect: action.redirect }
   case CLOSE:
     return { openModalType: null, signUpView: null }
   default:
     return state
 }
}
```

3，membershipModal 组件的mapStateToProps 监听到store的state发生变化，即触发组件重新渲染

https://github.com/tkww/national-hub/blob/main/src/share/Layout/components/MembershipModal/index.js

```
const mapStateToProps = (state) => {
 const { membershipModal = null } = state

 if(!membershipModal) return null

 if(membershipModal) {
   const { openModalType, redirect, signUpView } = membershipModal

   return { openModalType, redirect, signUpView }
 }
}
```

4，union的membershipModalRedux组件拿到openModalType为‘SIGN_UP'的新props，传给membershipModal, 然后把注册表单在页面上显示出来

https://github.com/tkww/union/blob/main/packages/%40xo-union/tk-component-membership-modal-redux/src/containers/MembershipModalReduxWrapper/index.jsx

```
 <MembershipModal
   {...props}
   openModal={openModalType}
```

```
  renderMembershipForm={formProps => (
    <MembershipFormRedux
      onSignUpSuccess={onSignUpSuccess}
      onLogInSuccess={onLogInSuccess}
      {...formProps}
    />
  )}
/>
```

https://github.com/tkww/union/blob/main/packages/%40xo-union/tk-component-membership-modal/src/components/MembershipModal/index.jsx

```
    <CanSpam>
      {({ ready }) => {
        if (!formType || !ready) {
          return null;
        }

        return (
          <Modal size="sm" onClose={onClose}>
            {renderMembershipForm(
              {
                formType,
                initialFocusRef: emailInputRef,
                ...rest,
              },
              ref,
            )}
            {children}
          </Modal>
        );
      }}
    </CanSpam>
```

以上就是membershipModal结合redux打开注册表单的一个过程；

3，用户注册

用户输入邮箱密码，点击注册后，会发生如下逻辑步骤：

3.1，触发membershipFormRedux的onSignUp方法

```
const onSignUp = React.useCallback(
  ({ valid, data }) => {
    if (valid) {
      dispatch(
        signUp(
          {
            ...data,
            ...additionalMembershipData,
          },
          onSignUpSuccess,
        ),
      );
    }
  },
  [onSignUpSuccess, additionalMembershipData],
);
```

可以看到dispatch里面是一个函数作为参数，这里就是使用thunk改造后的store.dispatch, 里面的函数就是一个中间件，该中间件接受了两个参数，一个是用户注册信息对象，一个就是前面初始化部分提到的传入的onSignUpSuccess函数

3.2, 中间件函数执行

Store-membership-redux

```
export const signUp = membershipActionFactory(
 {
  progress: SIGN_UP_START,
  success: SIGN_UP_SUCCESS,
  error: SIGN_UP_ERROR,
 },
 (m, params) => m.signUp(params),
);
```

可以看到是调用了一个工厂模式的函数 membershipActionFactory,

传入了一个对象和一个回调函数

```
const membershipActionFactory =
 (types, membershipCallback) =>
 (...params) => {
   const [passThroughParams, onSuccess, onError] =
normalizeParams(params);

   return (
     dispatch,
     getState,
     { membershipService = defaultMembershipService } = {},
   ) => {
     dispatch({ type: types.progress });

     return membershipCallback(membershipService, passThroughParams)
       .then((payload) => {
         dispatch({ type: types.success, payload });
         return payload;
       })
       .then(onSuccess)
       .catch((error) => {
         dispatch({ type: types.error, error });
         onError(error);
       });
   };
 };
```

这里的函数嵌套得比较多，建议多看几遍方便理解，也是比较关键的步骤，其实现原理其实就是利用中间件进行异步操作发送action

这里首先一开始先发出一个action（type为SIGN_UP_START）告诉store，要开始注册操作, 此时这里是同步

```
dispatch({ type: types.progress });
```

然后往下执行了一个异步操作

```
 return membershipCallback(membershipService, passThroughParams)
       .then((payload) => {
         dispatch({ type: types.success, payload });
         return payload;
       })
       .then(onSuccess)
       .catch((error) => {
         dispatch({ type: types.error, error });
         onError(error);
       });
```

这里的membsershipCallback就是前面传入的回调函数

```
(m, params) => m.signUp(params),
```

该回调函数接受了两个参数，一个membershipService, 即为sdk-membership 的实例对象（a-call, i-call, t-call即在该步骤触发），passThroughParams为用户输入的信息，该步骤后面部分会再细分析

当执行完前面的异步操作之后，就会发出一个action（type为SIGN_UP_SUCCESS）
然后再执行前面传进来的onSignUpSuccess回调函数，当抛出异常时，就会执行catch里面捕获异常函数。

3.3，解析membershipService（此步骤为重点）
Github-code

前面的的membsershipCallback执行的方法就是下面这一行
(m, params) => m.signUp(params),
转换一下其实就是membershipService.signUp(passThroughParams)

也就是如下代码：

```
signUp(params) {
  return this.#client
    .createMember(params)
    .then(this.#storeSession)
    .then((response) =>
      this.#analytics.reportSignUp(response).then(() => response),
    );
}
```

可以看出也是一个执行异步操作的函数

a, 首先第一步this.#client.createMember(params) 创建用户信息

#client声明如下：

```
constructor({
  persistenceOptions,
  clientOptions,
  analyticsOptions,
  client = createClient(clientOptions),
  persistence = new MembershipBrowserPersistence(persistenceOptions),
  analytics = new MembershipAnalytics(analyticsOptions),
} = {}) {
  this.#client = client;
  this.#persistence = persistence;
  this.#analytics = analytics;
}
```

其实就是这一步 client = createClient(clientOptions),也就是执行下面的方法

```
export default function createIsomorphicClient(params = {}) {
 return new MembershipClient({
   requestBuilder: superagent,
   ...params,
 });
}
```

New 了一个MembershipClient的实例对象，这里的superagent是nodejs用ajax api的第三方模块，params就是前面传入的clientOptions, 也是由前面passThroughParams里传的，但是目前暂未看到有传对应的值，估计也是用于其他方面拓展获取客户端信息的；

this.#client.createMember(params) 执行了MembershipClient 里的createMember方法

```
createMember(params) {
  this.payloadValidator(params);

  return this.requestBuilder
    .post(`${this.host}/members`)
    .query({ apikey: this.apiKey })
    .set('Accept-Version', this.version)
    .send({ members: params })
    .withCredentials()
    .then((response) => new MemberResponseWrapper(response))
    .catch(rethrowWrappedError);
}
```

第一步里的方法是用于判断是否传入了wedding 相关的属性，是的话则抛出异常（gitcode), 这里我们并没有传；

```
this.payloadValidator(params);
```

第二步就是调用superagent发送一个post请求去创建用户信息，也是异步操作，这里的withCredentials()是确保可以发送cookies, 用于跨域的

```
this.requestBuilder
```

```
    .post(`${this.host}/members`)
    .query({ apikey: this.apiKey })
    .set('Accept-Version', this.version)
    .send({ members: params })
    .withCredentials()
```

当前面post请求结束后，调用了MemberResponseWrapper，其把返回的response进行封装，用于分离数据业务模块（get xx)

```
.then((response) => new MemberResponseWrapper(response))
    .catch(rethrowWrappedError);
```

b, 设置cookie

```
.then(this.#storeSession)
```

```
#storeSession = (response) => {
  this.#persistence.onLogIn(response.session);

  return response;
};
```

调用了MembershipBrowserPersistence的实例对象，传入了前面response.session,即：

```
get session() {
  return this.response.body.linked.sessions[0];
}
```

通过打log得出传入的参数如下

```
{
        created_at: "2022-05-30T19:03:23.067Z"
        id: "047891e7-de5b-43a3-bc79-d2db538bce68"
        ticket: ""
        token: "xxxx"
}
```

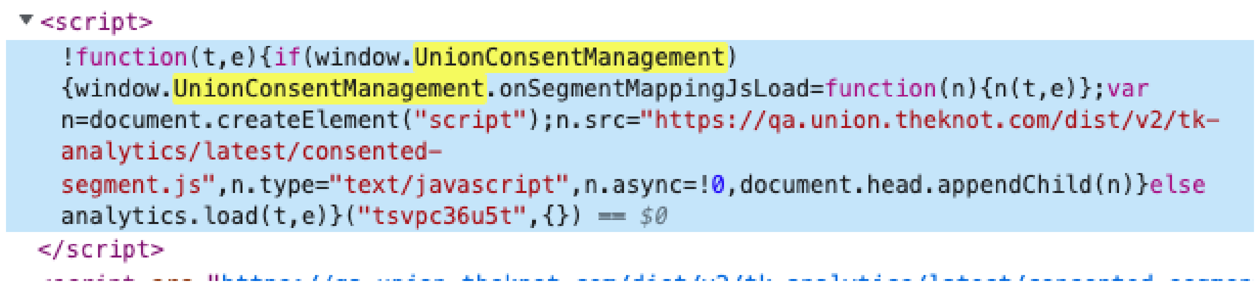
gitcode

```
onLogIn({
  token,
  tokenExpirationDate = daysInTheFuture(this.sessionTokenExpiration),
}) {
  if (globalThis.UnionConsentManagement) {
    globalThis.UnionConsentManagement.onConsentedToNecessary(() => {
      this.storage.setItem(this.sessionTokenCookie, token, {
        expires: tokenExpirationDate,
      });
    });
  } else {
    this.storage.setItem(this.sessionTokenCookie, token, {
      expires: tokenExpirationDate,
    });
  }
}
```

token为前面传入的token， tokenExpirationDate为token过期时间,如果传入的cookieOptions里的useExternalCookieConfig为true的话就是7天，为false或者没传则为30天

判断了window上是否有UnionConsentManagement，有则在其回调里设置了cookie，这里使用了cookie-storage这个第三方包，有兴趣可以了解一下

下面为UnionConsentManagement在页面上插入的代码：



```
▼<script>
   !function(t,e){if(window.UnionConsentManagement)
   {window.UnionConsentManagement.onSegmentMappingJsLoad=function(n){n(t,e)};var
   n=document.createElement("script");n.src="https://qa.union.theknot.com/dist/v2/tk-
   analytics/latest/consented-
   segment.js",n.type="text/javascript",n.async=!0,document.head.appendChild(n)}else
   analytics.load(t,e)}("tsvpc36u5t",{}) == $0
 </script>
```

c, 调用analytic发送event tracking

在这一步进行a-call, i-call，t-call的发送

```
    .then((response) =>
      this.#analytics.reportSignUp(response).then(() => response),
    );
```

也是先new了MembershipAnalytics的实例对象[git code]

```
analytics = new MembershipAnalytics(analyticsOptions),
```

在该类里，有两个services: segment, mixpanel

```
static defaultServices = ({ analytics = analyticsWrapper } = {}) => ({
  segment: new SegmentAnalytics({ analytics }),
  mixpanel,
});
```

reportSignUp调用了私有方法maybeWaitForAllServicesTo 传入了一个callback

```
reportSignUp(data) {
  return this.#maybeWaitForAllServicesTo((service) =>
    callIfDefined(service, 'reportSignUp', data),
  );
}
```

这里的callback也是调用了下面的方法，其实也就是执行segment或mixpanel这两个service的内部方法，比如上面就是执行了segment.reportSignUp或mixpanel.reportSignUp(因mixpanel没有reportSignUp的内部方法，故其在这里没什么作用）

```
const callIfDefined = (service, f, data) => {
 if (typeof service[f] === 'function') {
  return service[f](data);
 }

 return null;
};
```

再回到私有方法maybeWaitForAllServicesTo

```
#maybeWaitForAllServicesTo = (callback) => {
  const promises = this.#getServicesList().map(([name, service]) =>
    (callback(service) || Promise.resolve()).catch((error) => {
      console.warn(`${name} failed to report.`);
      console.error(error);
    }),
  );

  if (this.#wait) {
    return Promise.all(promises);
  }

  return Promise.resolve();
};
```

声明了一个promises的数组对象，该数组就是遍历上面两个service并调用其内部方法进行回调（这一步的callback(service) 是一定会执行的）

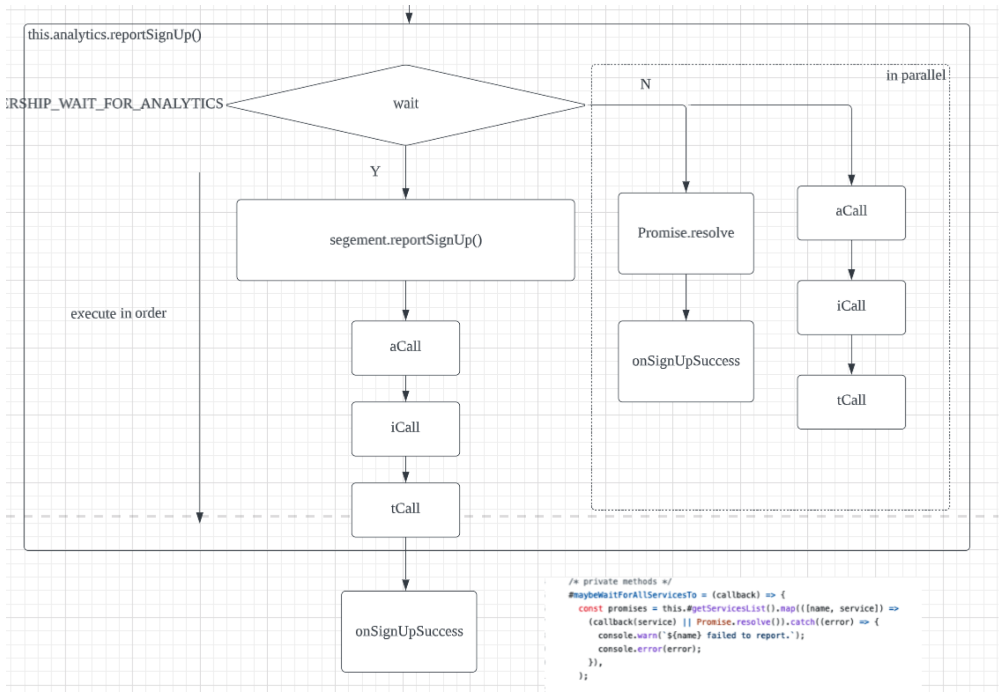
往下执行了判断this.#wait是否为true，这里的this.#wait就是由环境变量UNION_MEMBERSHIP_WAIT_FOR_ANALYTICS决定

```
const defaultWaitConfig =
 process.env.UNION_MEMBERSHIP_WAIT_FOR_ANALYTICS === 'true';
```

```
constructor({
  analytics,
  wait = defaultWaitConfig,
  services = MembershipAnalytics.defaultServices({ analytics }),
} = {}) {
  this.#wait = wait;
  this.#services = services;
}
```

this.#wait为true的话，则执行Promise.all(promises), 就会等前面的promises（segment.reportSignUp, 也是在里面进行a-call， i-call，t-call的发送）全部执行完；

如果为false的话，就不等前面的promises执行完就执行promise.resolve() 即异步操作外面的then(), 但由于前面的promises声明里已经调用了callback(service), 故segment.reportSignUp()也会执行，但是不会等它执行完毕，所以才会出现，a-call发出去了，但是i-call, t-call被cancel的情况，就是因为还没等发送完毕已经执行了跳转。如下流程图：



```
/* private methods */
#maybeWaitForAllServicesTo = (callback) => {
  const promises = this.#getServicesList().map(([name, service]) =>
    (callback(service) || Promise.resolve()).catch((error) => {
      console.warn(`${name} failed to report.`);
      console.error(error);
    }),
  );
```

为了验证该环境变量是否起作用，在本地union代码里打上几个log：

1，在maybeWaitForAllServicesTo 的promises声明里打log，用于验证其遍历执行services(segment, mixpenal)

```
console.log('maybeWaitForAllServicesTo call', name);
```

2, 在segment.reportSignUp里打4个log，用于判断reportSignUp的执行和event的发送

```
    console.log('start segment.reportSignUp');

    return new _Promise(function (resolve, reject) {
      _setTimeout(reject, _this.timeout, new Error('Segment timed out.'));

      console.log('start a-call');
      _this.analytics.alias(memberProperties.userId, memberProperties,
function () {
        var identifyData = _objectSpread(_objectSpread({},
memberProperties), {}, {
          unreadMessages: 0
        });
        console.log('start i-call');
        _this.analytics.identify(memberProperties.userId, identifyData,
function () {
          console.log('start t-call');
          _this.analytics.track('New Registration', trackEventData, resolve);
        });
      });
    });
```

3，在memberService.signUp里打log，用于判断页面结束reportSignUp

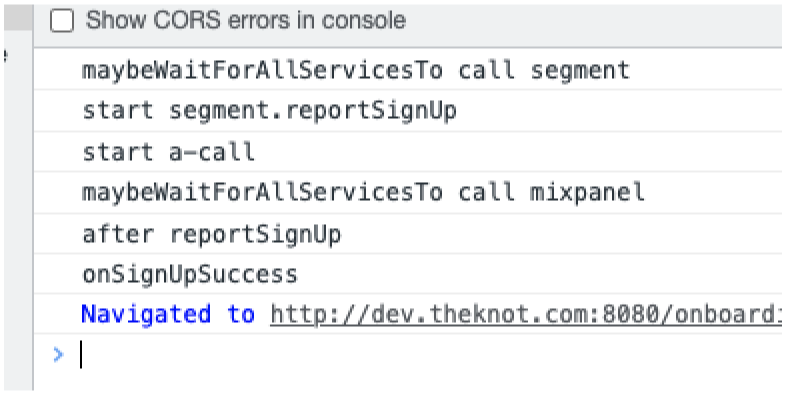
```
    return _classPrivateFieldGet(this,
_client).createMember(params).then(_classPrivateFieldGet(this,
_storeSession)).then(function (response) {
      return _classPrivateFieldGet(_this2,
_analytics).reportSignUp(response).then(function () {
        console.log('after reportSignUp');
        return response;
      });
    });
```

4, 在onSignUpSuccess里打log, 用于判断执行signUpsucess方法

```
if (action.type === SIGN_UP_SUCCESS) {
  console.log('onSignUpSuccess');
  window.location.assign(newMemberLocation);
  return;
}
```

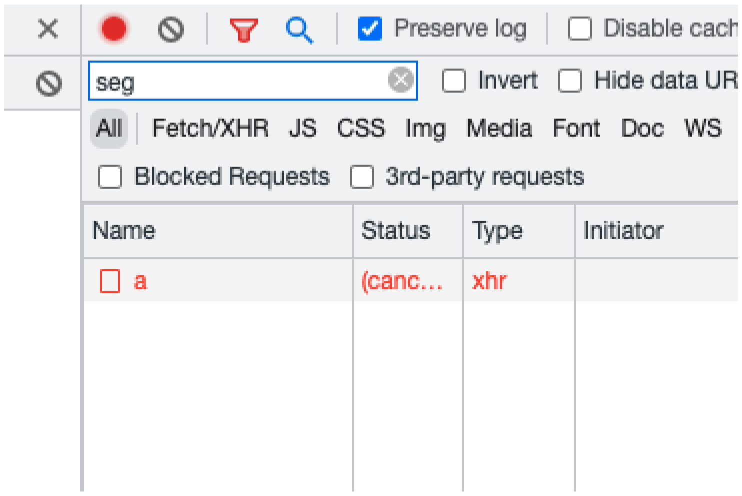
第一个验证：未设置UNION_MEMBERSHIP_WAIT_FOR_ANALYTICS

在页面上注册，得以下log和network结果：



Show CORS errors in console

```
maybeWaitForAllServicesTo call segment
start segment.reportSignUp
start a-call
maybeWaitForAllServicesTo call mixpanel
after reportSignUp
onSignUpSuccess
Navigated to http://dev.theknot.com:8080/onboard:
```

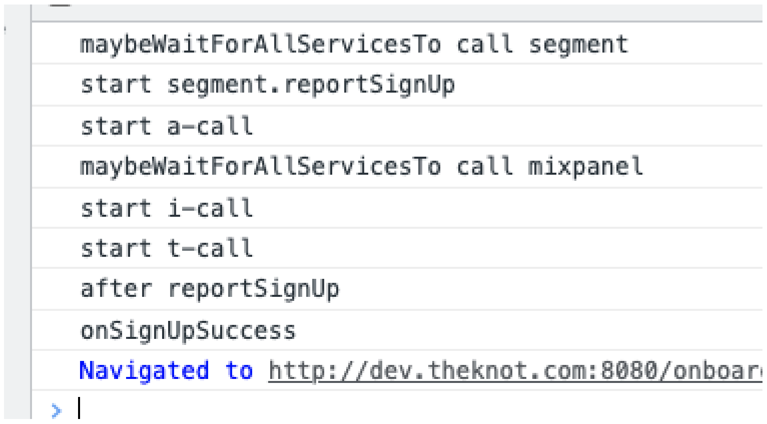


Preserve log / Disable cach / seg / Invert / Hide data UR / All Fetch/XHR JS CSS Img Media Font Doc WS / Blocked Requests / 3rd-party requests

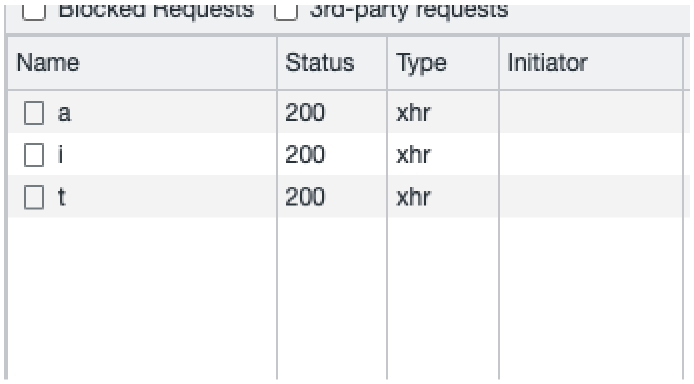
Name	Status	Type	Initiator
a	(canc…	xhr	

由上面log可得，在maybeWaitForAllServicesTo 方法内部，遍历了两个services（segment, mixpenal)，在第一个callback(service)执行的时候，即segment.reportSignUp正常执行时，a-call也开始执行，但是并没有等其执行完毕，就执行下一步了，即后面的onSignUpSuccess，页面也重定向了，故a-call被cancel;

第二个验证，设置UNION_MEMBERSHIP_WAIT_FOR_ANALYTICS 为true 结果如下：



```
maybeWaitForAllServicesTo call segment
start segment.reportSignUp
start a-call
maybeWaitForAllServicesTo call mixpanel
start i-call
start t-call
after reportSignUp
onSignUpSuccess
Navigated to http://dev.theknot.com:8080/onboar
>
```



Name	Status	Type	Initiator
a	200	xhr	
i	200	xhr	
t	200	xhr	

由上结果可得，onSignUpSucess会等前面reportSignUp执行完毕，才执行，故其能正常发送event

d, segment.reportSignUp

segment.reportSignUp如下：

```
reportSignUp({ analyticsProperties }) {
  const { memberProperties, accountCreateProperties } =
analyticsProperties;

  const trackEventData = {
    platform: 'web',
    ...accountCreateProperties,
  };

  return new Promise((resolve, reject) => {
    setTimeout(reject, this.timeout, new Error('Segment timed out.'));
    this.analytics.alias(memberProperties.userId, memberProperties, () => {
      const identifyData = { ...memberProperties, unreadMessages: 0 };

      this.analytics.identify(memberProperties.userId, identifyData, () => {
        this.analytics.track('New Registration', trackEventData, resolve);
      });
    });
  });
}
```

首先设置了segment超时3000ms执行抛出timeout异常；

```
  setTimeout(reject, this.timeout, new Error('Segment timed out.'));
```

当reportSignUp(指发出a-call，i-call，t-call）执行时间超过3000ms, 就会出现onSignUpSuccess的操作先执行，所以才会出现即使设置了环境变量UNION_MEMBERSHIP_WAIT_FOR_ANALYTICS为true，发event仍cancel的情况。所以我们在onSignUpSuccess的重定向里加上定时器，其实也是为了拖延重定向的操作，让发event操作先执行完毕。

接着执行了reportSignUp操作，按顺序发送a-call，i-call，t-call，如果前面哪一步call执行出问题，后面的call就会被cancel掉

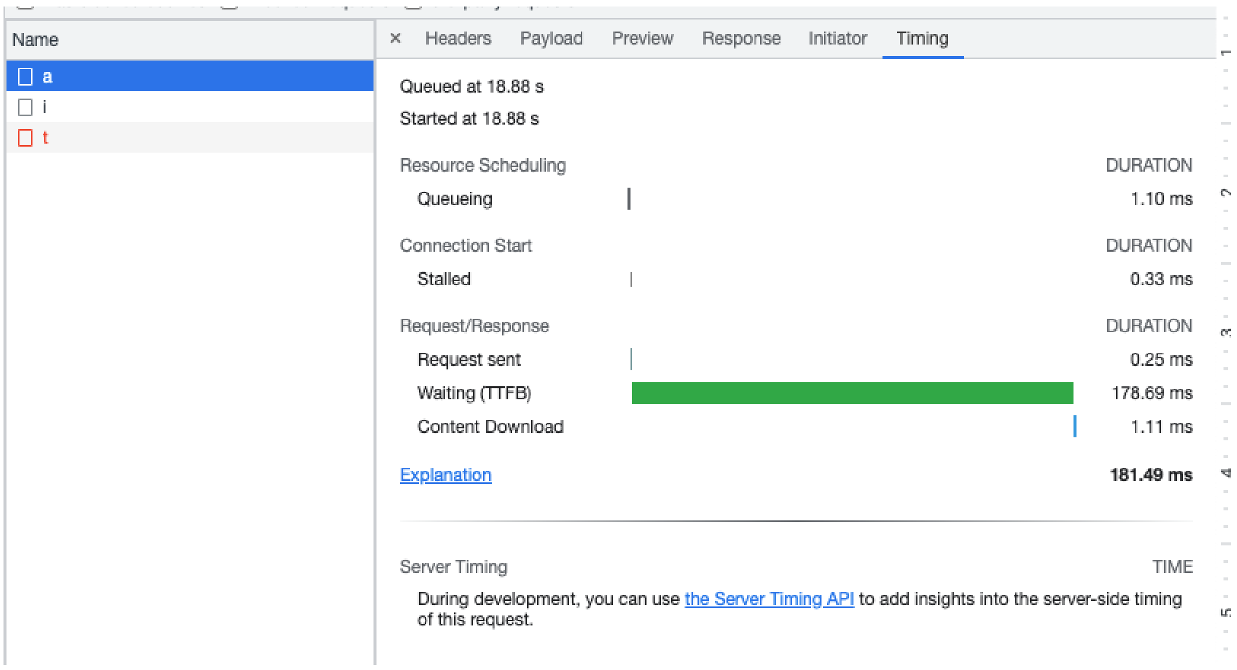
```
 this.analytics.alias(memberProperties.userId, memberProperties, () => {
     const identifyData = { ...memberProperties, unreadMessages: 0 };

     this.analytics.identify(memberProperties.userId, identifyData, () => {
       this.analytics.track('New Registration', trackEventData, resolve);
     });
   });
```

为了验证该结论，修改本地union代码设置timeout为600ms(因为在本地测试过程中，reportSignUp的操作耗时大多低于1000ms）

```
return {
  segment: new SegmentAnalytics({
    analytics: analytics,
    timeout: 600
  }),
  mixpanel: mixpanel
};
```

然后在页面上进行signUp操作，查看network可得：



Name

a
i
t

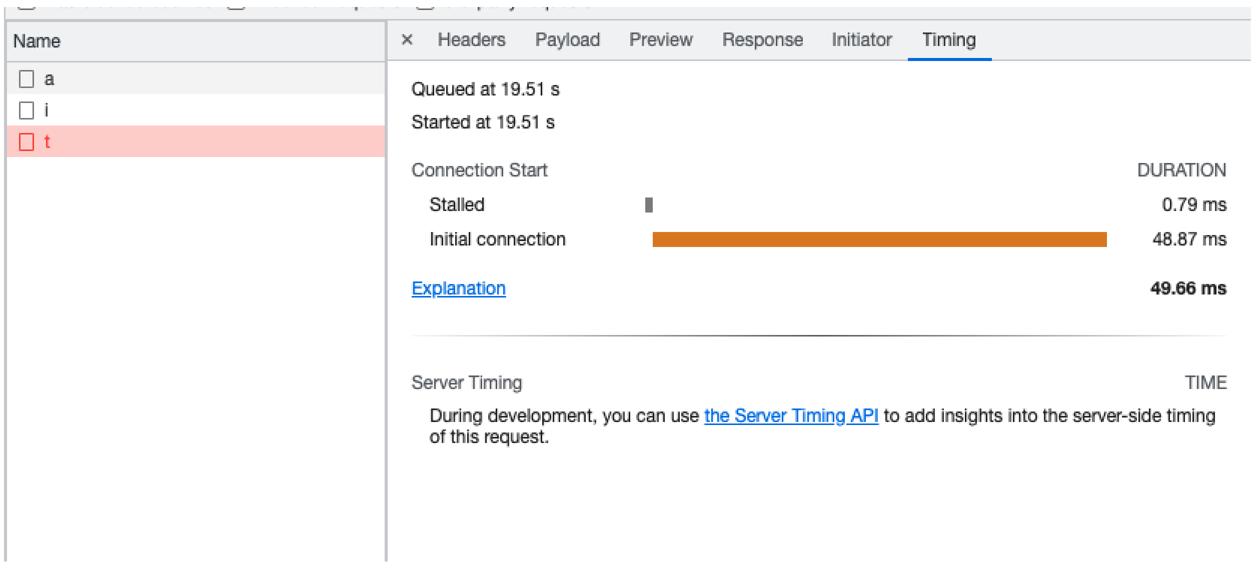
× Headers Payload Preview Response Initiator Timing

Queued at 18.88 s
Started at 18.88 s

	DURATION
Resource Scheduling	
Queueing	1.10 ms
Connection Start	DURATION
Stalled	0.33 ms
Request/Response	DURATION
Request sent	0.25 ms
Waiting (TTFB)	178.69 ms
Content Download	1.11 ms
Explanation	**181.49 ms**

Server Timing TIME

During development, you can use the Server Timing API to add insights into the server-side timing of this request.



Name

a
i
t

× Headers Payload Preview Response Initiator Timing

Queued at 19.51 s
Started at 19.51 s

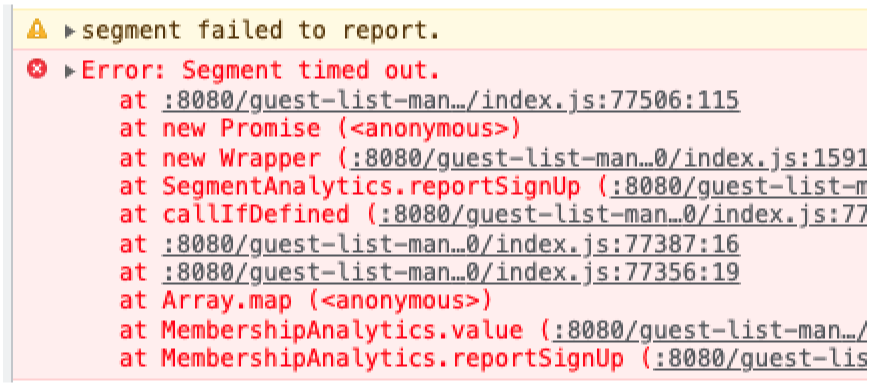
Connection Start	DURATION
Stalled	0.79 ms
Initial connection	48.87 ms
Explanation	**49.66 ms**

Server Timing TIME

During development, you can use the Server Timing API to add insights into the server-side timing of this request.

由上面两图发现，a-call开始于18.88s，t-call开始于19.51s，中间间隔了630ms，是大于我们设置的600ms timeout时间的，而此时页面已经开始跳转了，故t-call被cancel掉了

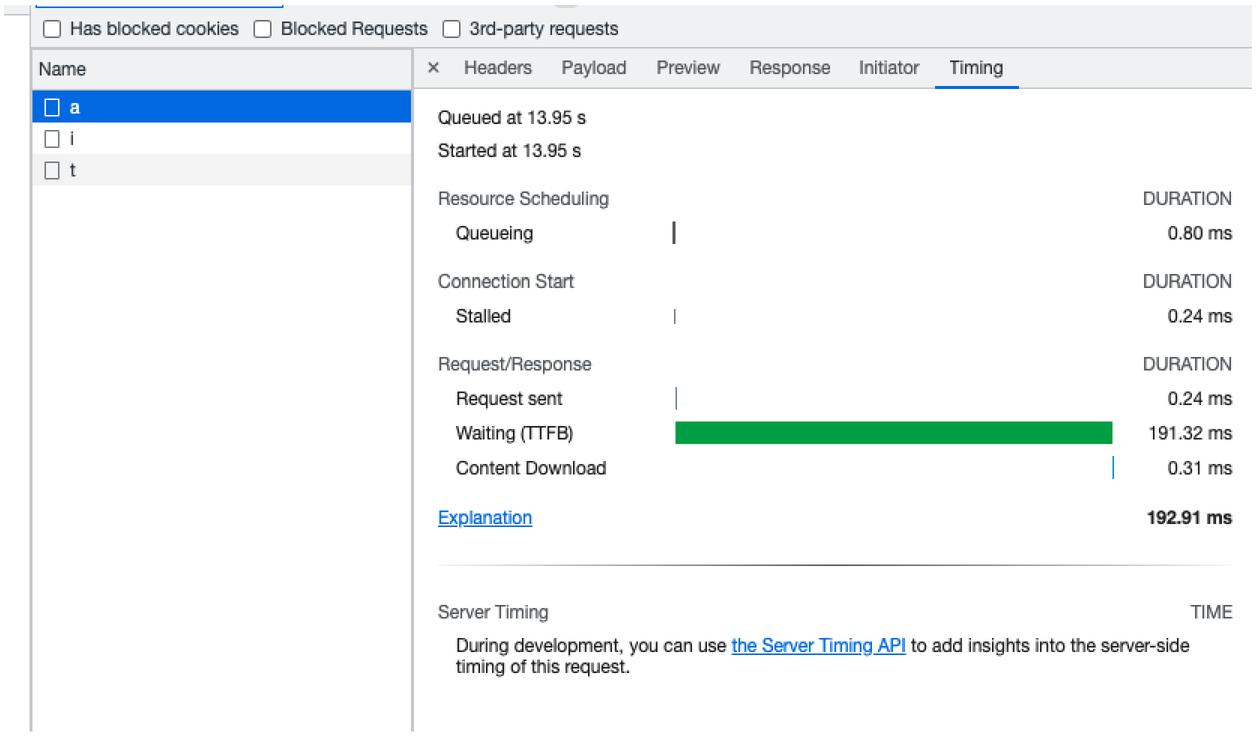
并且查看错误提示可以看到如下警告及错误：



```
▸segment failed to report.
▸Error: Segment timed out.
    at :8080/guest-list-man…/index.js:77506:115
    at new Promise (<anonymous>)
    at new Wrapper (:8080/guest-list-man…0/index.js:1591
    at SegmentAnalytics.reportSignUp (:8080/guest-list-m
    at callIfDefined (:8080/guest-list-man…0/index.js:77
    at :8080/guest-list-man…0/index.js:77387:16
    at :8080/guest-list-man…0/index.js:77356:19
    at Array.map (<anonymous>)
    at MembershipAnalytics.value (:8080/guest-list-man…/
    at MembershipAnalytics.reportSignUp (:8080/guest-lis
```

其即为超时抛出的异常

再进一步验证，把timeout设置为1000ms

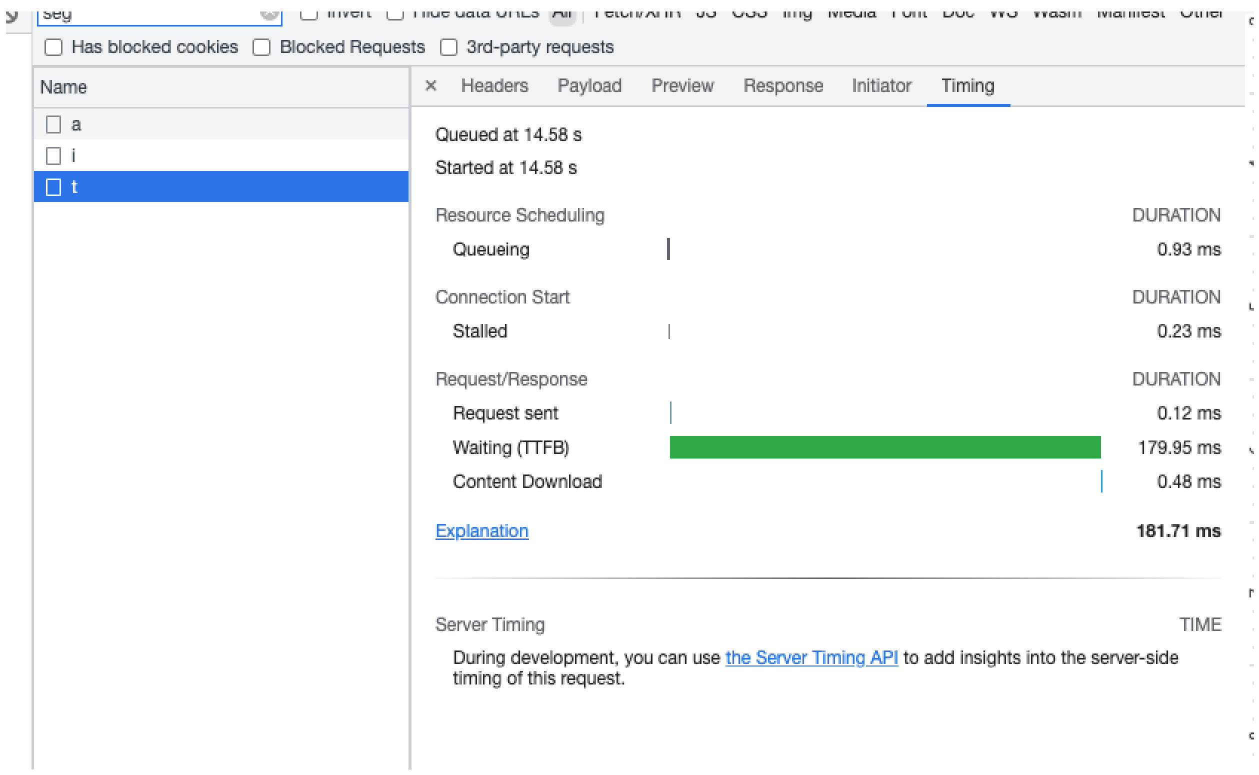


Queued at 13.95 s
Started at 13.95 s

Resource Scheduling	DURATION
Queueing	0.80 ms
Connection Start	DURATION
Stalled	0.24 ms
Request/Response	DURATION
Request sent	0.24 ms
Waiting (TTFB)	191.32 ms
Content Download	0.31 ms
Explanation	**192.91 ms**

Server Timing TIME

During development, you can use the Server Timing API to add insights into the server-side timing of this request.


Has blocked cookies Blocked Requests 3rd-party requests

Name
a
i
t

Headers Payload Preview Response Initiator Timing

Queued at 14.58 s
Started at 14.58 s

Resource Scheduling	DURATION
Queueing	0.93 ms
Connection Start	**DURATION**
Stalled	0.23 ms
Request/Response	**DURATION**
Request sent	0.12 ms
Waiting (TTFB)	179.95 ms
Content Download	0.48 ms
Explanation	181.71 ms

Server Timing TIME

During development, you can use the Server Timing API to add insights into the server-side timing of this request.

由上面两图发现，a-call开始于13.95s，而t-call开始于14.58s，并耗时了181.71ms，可得reportSignUp耗时大约811ms，小于我们设置的timeout时间，故其正常发送event

e，总结

发**event**会被**cancel**的两种情况：

1，环境变量UNION_MEMBERSHIP_WAIT_FOR_ANALYTICS为false或者未定义

2，reportSignUp的耗时超过3000ms（小概率发生）

解决方案：

1，设置UNION_MEMBERSHIP_WAIT_FOR_ANALYTICS为true

2，加定时器setTimeout在onSignUpSuccess的重定向操作里，以等待reportSignUp执行完毕；

3，增加reportSignUp的timeout时间，但是由于union代码里并没有给出这样一个参数，所以该方案需要向union提需求；