React相关知识点介绍

Hook介绍

React自带的hook

useRef

useRef存放的值，不会受到render影响，修改该值也不会造成render，所有闭包环境内获取ref上的值永远能拿到最新的

用法:

引用dom

```
const App = () => {
    const domRef= useRef();
    return <div ref={domRef} />
}
```

存放一些不需要被render影响的值，比如计时器的引用

```
const App = () => {
    const [count, setCount] = useState(0)
    const timer = useRef();

    const startCount = () => {
        timer.current = setInterval(()=>{
            setCount(c=>c+1);
        }, 1000)
    }

    const clearTimer = ()=>{
        clearInterval(timer.current);
        timer.current = null;
    }

    return <div>{count}</div>
}
```

ref上的值变化时也无法被useEffect监听到，所以以下代码在ref变更值时不会触发

```
useEffect(()=>{
    console.log(countRef.current);
}, [countRef.current])

useEffect(()=>{
    console.log(countRef.current);
}, [countRef])
```

useCallback

用于react的多次render过程中保证只会生成一次该函数(在依赖项不变的情况下)，减少不必要的渲染

使用方式:

useCallback(函数体, [依赖项])

依赖项：仅当依赖项变化时，才会重新生成一遍该函数

```
const handleClick = useCallback(() => {

}, [dep1, dep2...])
```

弊端:

1. 传入的函数体如果引用了外部的变量，会导致获取不到最新的值，需要把变量放入依赖项里面，如果引用的变量很多，会导致放置的依赖项也跟着变长([dep1, dep2....])
2. 在某些场景下，比如监听输入框搜索，该搜索函数调用时一定需要获取到最新的输入框中的值，此时就需要把输入框的值放入依赖项中，也就导致了用户每敲一个字符，该函数就更新一次，用了useCallback和不用没什么区别

解决:

不使用useCallback

而是使用useRef和useCallback组合，自定义一个hook，在ahook里面的名字叫useMemoizedFn

react官方针对这些弊端，计划出一个useEvent或者useEffectEvent 的hook来代替useCallback，该计划已在react rfc提案中，目前未出世

https://github.com/reactjs/rfcs/blob/useevent/text/0000-useevent.md

目前可以直接使用useMemoizedFn代替，效果类似(下面会提到)

useMemo

用于缓存某种计算结果，只有依赖项更新时才会自动计算一次，不会每次render都重新计算

用法：

```
const calcValue = useMemo(() => {
  // 在这里进行某些耗性能的计算
  // ...
  // ...
  // 最后return 一个计算的结果
  return result

}, [dep1, dep2...])
```

useEffect

可以监听某些state或者props的变化执行回调，也可以模拟一些生命周期

模拟didmount：

添加空依赖即可，模拟组件加载完成后执行的回调

```
useEffect(() => {

}, [])
```

> 在react18 版本的create-react-app脚手架创建的项目中，会自动给根节点添加 `<React.StrictMode></React.StrictMode>` ，会导致在开发环境下，即便加了空依赖也会执行两次

模拟unmount：

```
useEffect(() => {
    return () => {
        // 这里是模拟的生命周期
    }
}, [])
```

return的函数触发时机:

每次依赖项变化的时候，就会执行一次return的函数，可以在这里解除某些副作用，比如EventListener

```
useEffect(() => {
    window.addEventListener("xxxx", handleXXX)
    return () => {
        window.removeEventListener("xxxx", handleXXX)
    }
}, [dep])
```

useState

React 函数组件 的 state

用法:

```
const [value, setValue] = useState("")
```

也可以如果初始值需要计算得到，也可以传个函数初始化

只能是同步的函数，不能异步

```
const [value, setValue] = useState(() => {
    const num1 = 1 + 2;
    const num2 = 2 + 3;
    return num1 + num2
});
```

setValue的两种传参

```
//1. 直接setState
setValue("这是输入")

//2. 通过传入回调方式执行
setValue((prevState) => {
    return "这是输入"
})
```

为什么要通过传入回调方式去setState

react setState并不是同步的，它会将多个setState合并成一次进行批处理，减少不必要的重复渲染

当多个相同的setState一起顺序执行时，会被覆盖

> 🐵 React 18之前的版本，某些场景下(非合成事件)，比如setTimeout, Promise.then，原生dom事件中setState会是同步，18版本之后使用createRoot创建的组件，自动全部批处理(伪异步)

```
const [count, setCount] = useState(0);
const handleClickA = () => setValue(count+1);

// ----------------------------------
handleClickA(); // setValue(1)
handleClickA(); // setValue(1)
handleClickA(); // setValue(1)
console.log(count, 'count');
```

正常情况下期望应该是value逐渐变成1,2,3，但是最后结果实际上是1，多次setState被合成了一次

解决：

使用回调方式，相当于setState进行了排队，prevCount就是队列中上一个count进行setState的计算后的结果，然后合成一次

```
const [count, setCount] = useState(0);
const handleClickA = () => setCount(prevCount => prevCount + 1);

handleClickA(); // setValue(1)
handleClickA(); // setValue(2)
handleClickA(); // setValue(3)
console.log(count, 'count');
```

class组件中的setState第二个参数回调在函数组件内的实现方式(部分模拟实现)

class组件中的setState可以传入第二个回调，用于setState完成后自动触发回调

```
this.setState({a: 1}, ()=>{
    // 回调内容
```

```
})
```

在函数组件中对应的实现方式

```
const handleSearch = () => {
    request(url, {count})
}

useEffect(() => {
 // 在这里执行回调
    handleSearch();
}, [count])
// 中间混杂着其他代码
// 中间混杂着其他代码
// 中间混杂着其他代码
const handleXXX = () => {
    setCount(1)
}

```

弊端：

1. 需要用一个useEffect去监听state变动，然后执行对应的回调
2. 代码逻辑分离开了，写到一半需要跳到另一个地方继续写逻辑，并且后续逻辑可能就是一行代码去执行某个函数

优化：

自定义hook对该逻辑进行封装, 使用useRef暂存函数，用useEffect监听后执行useRef上暂存的函数

```
import { useRef, useCallback, useState, useEffect } from 'react';

function useStateCallback<T>(
  initialState: T
): [T, (state: T, cb?: (state: T) => void) => void] {
  const [state, setState] = useState(initialState);
  const cbRef = useRef<((state: T) => void) | undefined>(undefined);

  const setStateCallback = useCallback(
    (mergedState: T, cb?: (state: T) => void) => {
      cbRef.current = cb;
      setState(mergedState);
    },
    []
```

```
  );

  useEffect(() => {
    if (cbRef.current && typeof cbRef.current === "function") {
      cbRef.current(state);
      cbRef.current = undefined;
    }
  }, [state]);

  return [state, setStateCallback];
}

export default useStateCallback;
```

使用方式

```
const [count, setCount] = useStateCallback(0);

// 正常当state使用
setCount(1);

// 需要setState完成后触发回调时
setCount(1, (newCount) => {
    handleLog();
})

// 如果该函数里面需要获取最新的count，必须使用useMemoizedFn
const handleLog = useMemoizedFn(() => {
    console.log(count)
})
```

这个handleLog函数里面如果去尝试获取最新的count，因闭包问题无法获取到最新的

需要对handleLog进行useMemoizedFn分装

useContext

Context的hook版本，用于跨多个层级组件传递数据，无需一层一层传递

用法：

```
export const ThemeContext = React.createContext({}) // 这里设置了一个 {} 作为默认初始值
```

然后在要传递的节点中使用刚创建的Context的Provider发射数据，通过value字段传递

> 🏆 React 19版本不需要Provider，直接 `<ThemeContext> </ThemeContext>` 即可

```
import { ThemeContext } from './themeContext';

export default function MyApp() {
 const [theme, setTheme] = useState({a: 1, b:2})
  return (
    <ThemeContext.Provider value={theme}>
      // 结构如下，一层套一层的组件
      // 简单示意下结构
        <Child1>
            |
            |-><Child2>
                  |
                  |-> <Child3 />

        </Child1>

    </ThemeContext.Provider>
  )
}
```

在子节点中(可以是跨多个层级的子节点)，跨多个组件层级直接取到传递过来的theme，类似redux的connect

使用useContext接收数据

```
import { ThemeContext } from './themeContext';
export default function Child3() {
  const theme = useContext(ThemeContext);
  return (
      <div>
          <div>{theme.a}</div>
          <div>{theme.b}</div>
      </div>
  )
}
```

优点:

1. 使得数据流传递更方便，无需一层一层传递
2. 不需要额外增加redux之类的状态管理库

弊端:

因context穿透传递效果，会导致React.memo失效, 比如有个组件用memo优化过了，即便是设置了false

```
const Child = React.memo(()=>{
    const {a, b} = useContext(ThemeContext);
    console.log(a, 'child render')
     return <div>{a}</div>
}, () => false)
```

当context的b改变时，还是会引发render，这就是context的穿透效果，如何避免呢

使用useMemo优化下，把整个render的内容放在useMemo里

```
  const Child = ()=>{
    return useMemo(() => {
        console.log(a, "a render");
        return (
          <div>
            <div>a: {a}</div>
          </div>
        );
      }, [a]);
  }

```

此时会发现，child组件的内容分只会因为a的变化而重新render

但是一般不会这样去用，这样子写会让代码越来越复杂

一般情况下业务组件没有大量复杂计算的逻辑，所以直接无视即可

useId

react18版本特有的hook，用于生成唯一id, 类似uuid，但是生成的id格式比较特别，有冒号包围

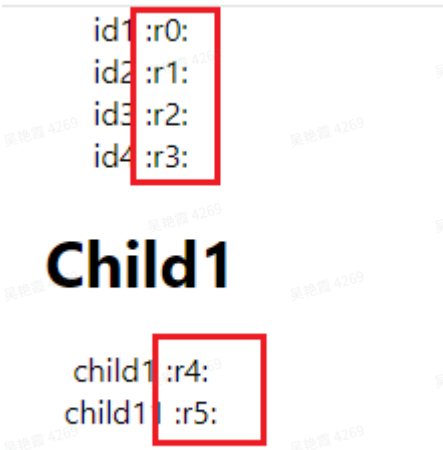
使用:

比较简单

```
import { useId } from 'react';
function App() {
  const id1 = useId();
  const id2 = useId();
  const id3 = useId();
  const id4 = useId();

  return (
    <div>
      <div id={id1}>id1 {id1}</div>
      <div>id2 {id2}</div>
      <div>id3 {id3}</div>
      <div>id4 {id4}</div>
      <Child1 />
    </div>
  );
}
```

可以看到页面生成的效果



特点:

1. 生成的一个在渲染周期内全局唯一的id，在多次渲染中保持不变
2. 主要用于解决服务端渲染(SSR)和客户端生成的id不一致的问题

> ssr脱水(dehydrate)时，会提前在服务器上生成id，但是同样的，客户端重新生成虚拟dom树并往真实dom注入事件(注水hydrate)时，又会重新生成一个id，导致id冲突，所以需要用useId保证双方的id统一
>
> 注：id冲突会导致注水失败

弊端：

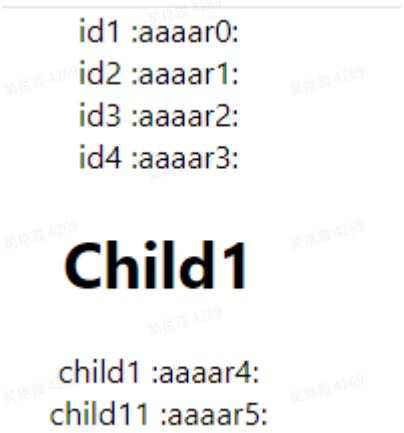
1. 需要react18版本
2. css选择器不支持这种带冒号的字符串

> 如何统一id的前缀

可以这么在createRoot中加identifierPrefix字段，这样生成的id都带前缀

```
const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement,
  { identifierPrefix: 'aaaa' }
);
root.render(
    <App />
);
```

App组件部分还是上面的代码，我们刷新下就可以看到



现在useId生成的id，都带有aaaa前缀了

useImperativeHandle

在以前的react class组件中，我们想要调用子组件的一些方法，我们可以使用ref获取到子组件的实例，再调用对应的方法 ref.current.xxx();

在函数式组件中如何调用呢，函数组件不会new一个实例，也没有this，所以需要配合一个forwardRef

```
  export const ChildFuncC = forwardRef((props, ref) => {
      const [test, setTest] = useState(0);

      useImperativeHandle(ref, () => {
        return {
          getState: () => {
            console.log(test, "test");
          }
        };
      }, [deps]);

      return <div>ChildFuncC</div>;
});


const App = ()=>{
    const childRef = useRef();
    const handleGetState = () => {
        childRef.current.getState();
    }
    return <ChildFuncC ref={childRef} />
}
```

也就是说，useImperativeHandle 是用于把子组件的一些方法暴露给父组件，只不过函数组件里面稍微麻烦一点，需要用forwardRef + useImperativeHandle 去传递

常用的第三方hook

useAntdTable

当我们写antd的table时，会经常手动管理form参数和表格的分页,loading等，但是这些逻辑完全都是可以抽出来的，useAntdTable就是用来做这些的

使用:

```
import { Form, Table } from 'antd';
import { useAntdTable } from 'ahooks';
const Test = () => {
  const [form] = Form.useForm();
  const { tableProps, search } = useAntdTable(getBatteryTypeListAsync, {
    form,
    defaultPageSize: 30,
```

```
  });
  return (
    <div>
      <Form form={form}>
        ......
        <Form.Item>xxxx</Form.Item>
        ......
        <Button onClick={search.submit}>查询</Button>
        <Button onClick={search.reset}>重置</Button>
      </Form>

      <Table {...tableProps} />
    </div>
  );
};
export default Test;
```

1. 需要有个form的ref，这样他才能自动搜集表单的值
2. 返回的tableProps，里面包含了分页的内容，也就是说，我们不需要手动管理分页了，只需要把tableProps解压到Table组件上，里面的分页会被自动接管
3. 查询或者重置，直接调search.submit或者search.reset即可

需要在service上改造下

service的参数固定了，current和pageSize是翻后的页码和一页的大小，formData是传入formRef搜集到的表单值，在这里发起request前可以做一些转换

最后return一个list和total的字段即可(字段名称固定)

```
export const getBatteryTypeListAsync = async (
  { current, pageSize }
  formData
) => {
  const res = await request.get(interfaceUrl.getBatteryTypeUrl, {
    params: {
      pageIndex: current,
      pageSize,
      ...formData,
    },
  });
  return {
    list: res.data?.list,
    total: res.data?.total,
  };
```

```
16  };
```

参数固定也就带来了一些弊端，比如想要表单之外的参数带进来，需要额外套一层函数

useCountDown

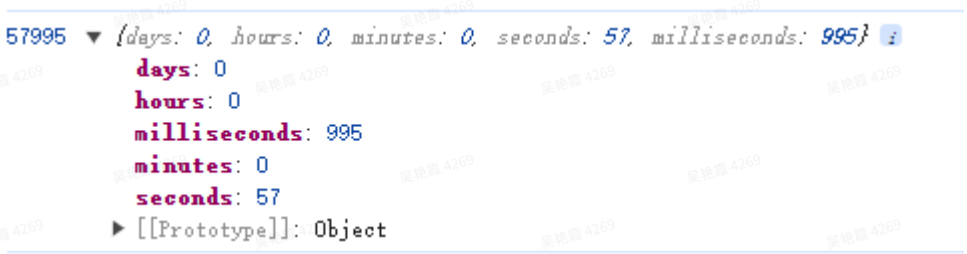
倒计时hook，无需手动写定时器

使用场景：手机验证码倒计时，订单待支付倒计时，协议阅读底部倒计时

在ahook里的使用:

```
1   import React from 'react';
2   import { useCountDown } from 'ahooks';
3
4   export default () => {
5     const [countdown, formattedRes] = useCountDown({
6       leftTime: 60 * 1000,
7     });
8     const { days, hours, minutes, seconds, milliseconds } = formattedRes;
9     console.log(countdown, formattedRes);
10
11    return (
12      <p>
13        There are {days} days {hours} hours {minutes} minutes {seconds} seconds
    {milliseconds}
14      </p>
15    );
16  };
```

There are 0 days 0 hours 0 minutes 57 seconds 880countdown 57880



可以看到log的地方时间戳和格式化后的日期都自动出来了

但是会发现时间可能还不是想要的，期望的倒计时格式可能在某些场景下是 HH:mm:ss 字符串格式的，还得手动格式化下，并且没有暂停和继续功能

在换电支付宝小程序里手写了一个useCountDown，实现了暂停，继续，格式化时间等功能

TODO:

callback更名成onTick

zeroCallback更名成onEnd

组件卸载时清除掉正在进行中的定时器

useUpdateEffect

用法与useEffect完全一致，但首次不执行

useLoadMore

滚动加载更多，用于移动端的分页，移动端一般是不停的往上滑，数据上则需要不停的请求下一页并拼接

以下为个人封装后的用法:

```
1    const { list, onScrollToLower, reset } = useLoadMore<Income>
    (getIncomeHistoryAsync, {
2       params: {}
3       transformRes: (data) => data.list,
4    });
```

初始化时需要手动调用reset重置第一页并查询

TODO:

可选入参initList更名成??

useMemoizedFn

优化部分函数性能，通过useCallback实现性能优化，通过useRef赋值在render过程中更新闭包变量

```
1   import { useRef, useCallback } from 'react';
2
3   function useMemoizedFn(callback) {
4     const callbackRef = useRef(null);
```

```
    callbackRef.current = callback;
    const fn = useCallback((...args) => {
      if (callbackRef.current) {
        (callbackRef.current as any).apply(null, args);
      }
    }, []);

    return fn;
}

export default useMemoizedFn;
```

return的fn使用了空依赖的useCallback，保证了render期间的稳定性，同时在render的过程中，把最新的callback赋值给ref.current，这样可以连带着callback的上下文一起更新

还有一个场景也能通过useMemoizedFn解决

```
const handleClick = () => {
    // 尝试去获取最新的state
    console.log(state)
}

useEffect(() => {
    window.addEventListener("xxx", handleClick);
    return ()=>{
        window.removeEventListener("xxx", handleClick);
    }
}, [])
```

当你在handleClick函数里尝试去获取state时，发现获取不到最新的state

因为闭包环境导致和这个eventListener里注册的handleClick是第一次render时候的handleClick

所以获取到的其实是第一次render时候的handleClick的上下文环境的state

解决：

```
const handleClick = useMemoizedFn(() => {
    console.log(state)
})

useEffect(() => {
    window.addEventListener("xxx", handleClick);
```

```
    return () => {
        window.removeEventListener("xxx", handleClick);
    }
}, [])
```

useDebounceFn

使一个函数变成防抖函数，在触发时，会延迟一段时间再触发(一般是500毫秒)，在这段时间内如果又重复触发时，延迟时间刷新

使用场景：

根据用户输入的值，在不点击搜索按钮的情况下去调接口搜索

```

const handleSearch = useDebounceFn(()=>{
    // 搜索
}, 500)

<Input
    onChange={(v)=>{
        setValue(v);
        handleSearch();
    }}
/>
```

自定义hook的写法

命名

功能类型hook:

use开头+功能名称

useTouch, useGesture, useLoadMore, useStateCallback, useLocation，useForceUpdate, useRouter等

业务类型hook：

use开头+通用业务名称或者数据名称

useMarkerList, useBattery，useBatteryList，useAddressList，useFaqPhone等

React API+组件 介绍

cloneElement

复制一个react元素，并且可以进行魔改

```
React.cloneElement(
  element, // 要克隆的React元素，封装组件，对象
  [props], // 可选，新元素要修改的props
  [...children] // 可选，新元素要替换的children，不光是content还有样式和element,type
)
```

一般是用于复制children，后面有demo会提到

memo

优化组件渲染，避免props没有变化时多余的rerender

第一个参数是组件，第二个参数是自定义比较函数，通过返回boolean来判断是否需要本次更新

```
// React.memo(组件，比较函数)
const Child = () => <div></div>

React.memo(Child, (prevProps, nextProps)=>{
    if(prevProps.a !== nextProps.a){
        return true;
    }
    return false;
})
```

一般不会特地去使用这个，只有项目有强性能优化指标时才会去考虑

lazy

懒加载一个react组件，配合Suspense使用，防止一次性加载文件过多，影响首页速度

```jsx
const MarkdownPreview = React.lazy(() => import('./MarkdownPreview.js'));

<Suspense fallback={<Loading />}>
  <MarkdownPreview />
</Suspense>
```

在懒加载完成前，会显示loading

ErrorBoundary

在react渲染过程中，如果在render时抛出error，会导致react从root节点开始卸载整棵树，通常的表现就是页面白屏，只剩一个root节点，如果不想要白屏，就需要用ErrorBoundary包裹

目前还没有办法将错误边界编写为函数式组件

```jsx
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // 这里做一些埋点，监控上报等

  }

  render() {
    if (this.state.hasError) {
      return <自定义的渲染错误时的UI />
    }

    return this.props.children;
  }
}
```

通常情况下会给重要的一些组件套上一层ErrorBoundary，使得代码报错时，不会整个页面白屏，影响主业务流程，同时在componentDidCatch中还能自定义信息去上报监控平台，搭配第三方或者自建的监控平台，实时通知，可以以最快的速度知道用户报错的原因

React开发常用技巧

使用...rest透传props

使用场景: 当props较多的情况下，将一部分props传给子组件

```
const Parent = ({className, style, ...rest}) =>{
    return <Child {...rest} />
}
```

如上 ...rest 表示除了 className, style之外的所有props的聚合

使用ReactDOM.createPortal 将真实节点挂载在外部

常用于调整一些无法改变组件样式，比如弹窗某些时候需要挂载到body节点上，如果放在某个组件内部可能会有样式冲突等情况

```
import ReactDOM from 'react-dom';

// 1. 直接渲染某组件
const Test = () => {
    return ReactDOM.createPortal(<TestModal />, document.body)
}

// 2. 或者作为共用容器
const ModalWrapper = ({children}) => {
    return createPortal(<>{children}</>, document.body)
}
// 使用时
const Modal = () => {
    return (
        <ModalWrapper>
            <div>aaaaa</div>
        </ModalWrapper>
    )
}
```

封装选择器时不需要关心触发器样式

业务场景：移动端经销商选择器，点击右上角link标签时弹出picker选择层

此时页面其他地方不需要经销商列表数据，只关心选择后的经销商的id，可以考虑使用cloneElement去复制children，并往children上注入一个叫onClick的props

```
const CompanyPicker: React.FC<IProps> = ({ children, onChange }) => {
  const [companyList, setCompanyList] = useState<Company[]>([] as Company[]);

  useDidShow(async () => {
      // 调接口获取companyList
  });

  const handleChange = (e) => {
     // 一些操作
      // .....
       onChange(e)
  };

  if (companyList.length === 0) {
    return React.cloneElement(children, { onClick: () => toast.error('当前城市无经销商') });
  }

  return React.cloneElement(children, { onClick: () => {
    // 拉起picker弹层的操作
  }});
};

export default CompanyPicker;

-----------------------------------------------------------------

<CompanyPicker onChange={handleSelect}>
    <View>选择经销商</View>
```

```
29    </CompanyPicker>
```

优势：

1. 不需要关心触发的UI，只需要对应的UI层能支持onClick即可，不论UI层是一个文字链接，图片，按钮，都可以直接使用该选择器
2. 使用方不需要关心经销商列表的数据来源，只需要关注onChange里的参数即可

缺点：

1. 如果页面其他地方需要这份经销商数据，不太方便提取
2. 如果一个页面有多个一模一样的经销商选择器，会请求多遍，数据会保存多份

解决：

数据来源使用公共hook， 例如useCompanyList，只需多传一个list即可

可以实现：

1. 一个页面共用一份companyList
2. 同时存在多个相同的选择器，只发一次请求

小程序使用伪单例避免弹出多个相同的授权弹窗

场景：

换电支付宝小程序首页初次进入时需要定位授权，由于代码/组件拆分又或者多人分工合作，每个组件都可能会第一时间去拉起定位授权弹窗，导致页面第一次进入时会弹出多个相同的定位授权弹窗

使用伪单例下的方法

```
import getUserLocation from './util';

class UserLocation {
    pendingLocation = null;
    getLocation(){
        if(this.pendingLocation){
            return this.pendingLocation;
        }
        this.pendingLocation = getUserLocation().finally(()=>{this.pendingLocation = null})
        return this.pendingLocation;
    }
}
const userLocation = new UserLocation();

export default userLocation;
```

使用:

```
const location = await userLocation.getLocation();
```

Antd Form 表单组件封装

在日常开发中，可能会有这样的场景，一个页面上面是搜索面板，下面是表格，大概长这样子

对于上半部分的搜索面板 我们可能会使用antd 的Form

```
<Form form={form}>
    <Form.Item name='batteryNo' label='电池编号'>
        <Input />
    </Form.Item>
</Form>
```

这样子可以使用form.validateFields自动校验表单，搜集表单的值

但是有时候会发现，一些控件的值是动态获取的，比如获取经销商列表，作为下拉框的选项

```
useEffect(()=>{
  // 调接口去获取companys
}, [])


<Form form={form}>
    <Form.Item name='companyId' label='经销商'>
        <Select>
            {
                companys.map((company)=>{
                    return <Option value={company.id} key...>{company.name}
</Option>
                })
            }
        </Select>
    </Form.Item>
</Form>
```

如果好几个页面都用到了这个经销商选择器，就会写重复的代码，可以手动抽离一个经销商选择器

```
const CompanySelector = (props) => {
  const [companyList, setCompanyList] = useState<Company[]>([]);

  useEffect(() => {
    // 调接口获取companyList

  }, []);
  return (
    <Select {...props}>
      {companyList.map((item) => {
        return (
          <Option value={item.code} key={item.code}>
            {item.name}
          </Option>
```

```
          );
        })}
      </Select>
    );
  }
```

> 让我们封装的自定义组件，也能被antd的Form自动搜集到值

事实上antd的Form.Item 会自动往children上注入onChange和value方法，我们只要把props的onChange和value给到Select即可，这里直接解压props到Select上也有同样效果

也就是说，直接这样子用，antd的form也能自动获取到用户选择后的id

```
<Form.Item name='companyId' label='经销商'>
    <CompanySelector />
</Form.Item>
```

设计思路：

对于经销商选择器的使用者来说，本质上并不关心列表里面的内容，他们只关心用户选择后的经销商id，因为这个id是要拿去调接口获取表格的数据。所以只需要暴露最后的id即可

存在的问题：

同上，可以抽出hook解决