父组件在调用子组件的时候，只需要在子组件标签内传递参数，子组件通过 `props` 属性就能接收父组件传递过来的参数

JSX | 复制代码

```jsx
function EmailInput(props) {
  return (
    <label>
      Email: <input value={props.email} />
    </label>
  );
}

const element = <EmailInput email="123124132@163.com" />;
```

### 11.2.2. 子组件向父组件传递

子组件向父组件通信的基本思路是，父组件向子组件传一个函数，然后通过这个函数的回调，拿到子组件传过来的值

父组件对应代码如下：

JSX | 复制代码

```jsx
class Parents extends Component {
  constructor() {
    super();
    this.state = {
      price: 0
    };
  }

  getItemPrice(e) {
    this.setState({
      price: e
    });
  }

  render() {
    return (
      <div>
        <div>price: {this.state.price}</div>
        {/* 向子组件中传入一个函数  */}
        <Child getPrice={this.getItemPrice.bind(this)} />
      </div>
    );
  }
}
```

子组件对应代码如下：

JSX | 复制代码

```jsx
class Child extends Component {
  clickGoods(e) {
    // 在此函数中传入值
    this.props.getPrice(e);
  }

  render() {
    return (
      <div>
        <button onClick={this.clickGoods.bind(this, 100)}>goods1</button>
        <button onClick={this.clickGoods.bind(this, 1000)}>goods2</button>
      </div>
    );
  }
}
```

### 11.2.3. 兄弟组件之间的通信

如果是兄弟组件之间的传递，则父组件作为中间层来实现数据的互通，通过使用父组件传递

JSX 复制代码

```jsx
class Parent extends React.Component {
  constructor(props) {
    super(props)
    this.state = {count: 0}
  }
  setCount = () => {
    this.setState({count: this.state.count + 1})
  }
  render() {
    return (
      <div>
        <SiblingA
          count={this.state.count}
        />
        <SiblingB
          onClick={this.setCount}
        />
      </div>
    );
  }
}
```

### 11.2.4. 父组件向后代组件传递

父组件向后代组件传递数据是一件最普通的事情，就像全局数据一样

使用 `context` 提供了组件之间通讯的一种方式，可以共享数据，其他数据都能读取对应的数据

通过使用 `React.createContext` 创建一个 `context`

JavaScript 复制代码

```javascript
const PriceContext = React.createContext('price')
```

`context` 创建成功后，其下存在 `Provider` 组件用于创建数据源， `Consumer` 组件用于接收数据，使用实例如下：

`Provider` 组件通过 `value` 属性用于给后代组件传递数据：

```jsx
<PriceContext.Provider value={100}>
</PriceContext.Provider>
```

如果想要获取 `Provider` 传递的数据，可以通过 `Consumer` 组件或者或者使用 `contextType` 属性接收，对应分别如下

```jsx
class MyClass extends React.Component {
  static contextType = PriceContext;
  render() {
    let price = this.context;
    /* 基于这个值进行渲染工作 */
  }
}
```

`Consumer` 组件：

```jsx
<PriceContext.Consumer>
    { /*这里是一个函数*/ }
    {
        price => <div>price: {price}</div>
    }
</PriceContext.Consumer>
```

### 11.2.5. 非关系组件传递

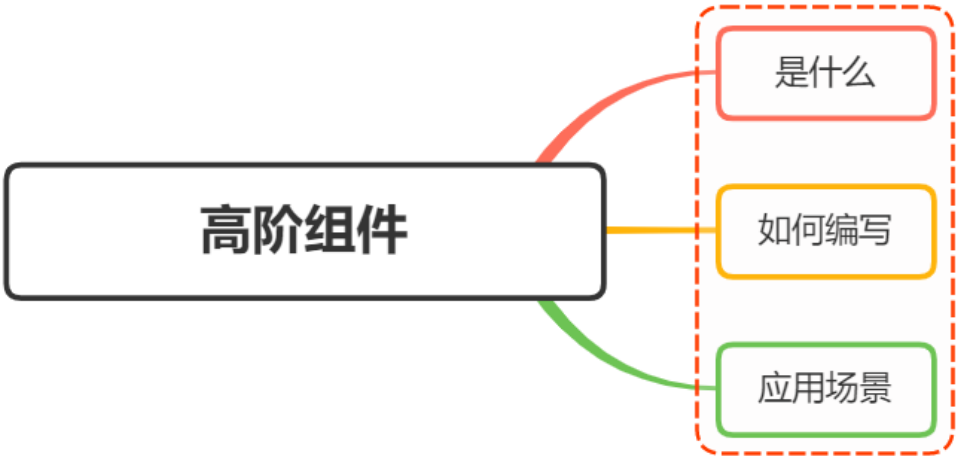
如果组件之间关系类型比较复杂的情况，建议将数据进行一个全局资源管理，从而实现通信，例如 `redux` 。关于 `redux` 的使用后续再详细介绍

## 11.3. 总结

由于 `React` 是单向数据流，主要思想是组件不会改变接收的数据，只会监听数据的变化，当数据发生变化时它们会使用接收到的新值，而不是去修改已有的值

因此，可以看到通信过程中，数据的存储位置都是存放在上级位置中

# 12. 说说对高阶组件的理解？应用场景？




## 12.1. 是什么

高阶函数（Higher-order function），至少满足下列一个条件的函数

- 接受一个或多个函数作为输入
- 输出一个函数

在 `React` 中，高阶组件即接受一个或多个组件作为参数并且返回一个组件，本质也就是一个函数，并不是一个组件

```jsx
const EnhancedComponent = highOrderComponent(WrappedComponent);
```

上述代码中，该函数接受一个组件 `WrappedComponent` 作为参数，返回加工过的新组件 `EnhancedComponent`

高阶组件的这种实现方式，本质上是一个装饰者设计模式

## 12.2. 如何编写

最基本的高阶组件的编写模板如下：

```jsx
import React, { Component } from 'react';

export default (WrappedComponent) => {
  return class EnhancedComponent extends Component {
    // do something
    render() {
      return <WrappedComponent />;
    }
  }
}
```

通过对传入的原始组件 `WrappedComponent` 做一些你想要的操作（比如操作 props，提取 state，给原始组件包裹其他元素等），从而加工出想要的组件 `EnhancedComponent`

把通用的逻辑放在高阶组件中，对组件实现一致的处理，从而实现代码的复用

所以，高阶组件的主要功能是封装并分离组件的通用逻辑，让通用逻辑在组件间更好地被复用

但在使用高阶组件的同时，一般遵循一些约定，如下：

- props 保持一致
- 你不能在函数式（无状态）组件上使用 ref 属性，因为它没有实例
- 不要以任何方式改变原始组件 WrappedComponent
- 透传不相关 props 属性给被包裹的组件 WrappedComponent
- 不要再 render() 方法中使用高阶组件
- 使用 compose 组合高阶组件
- 包装显示名字以便于调试

这里需要注意的是，高阶组件可以传递所有的 `props`，但是不能传递 `ref`

如果向一个高阶组件添加 `refe` 引用，那么 `ref` 指向的是最外层容器组件实例的，而不是被包裹的组件，如果需要传递 `refs` 的话，则使用 `React.forwardRef`，如下：

JSX | 复制代码

```jsx
function withLogging(WrappedComponent) {
    class Enhance extends WrappedComponent {
        componentWillReceiveProps() {
            console.log('Current props', this.props);
            console.log('Next props', nextProps);
        }
        render() {
            const {forwardedRef, ...rest} = this.props;
            // 把 forwardedRef 赋值给 ref
            return <WrappedComponent {...rest} ref={forwardedRef} />;
        }
    };

    // React.forwardRef 方法会传入 props 和 ref 两个参数给其回调函数
    // 所以这边的 ref 是由 React.forwardRef 提供的
    function forwardRef(props, ref) {
        return <Enhance {...props} forwardRef={ref} />
    }

    return React.forwardRef(forwardRef);
}
const EnhancedComponent = withLogging(SomeComponent);
```

## 12.3. 应用场景

通过上面的了解，高阶组件能够提高代码的复用性和灵活性，在实际应用中，常常用于与核心业务无关但又在多个模块使用的功能，如权限控制、日志记录、数据校验、异常处理、统计上报等

举个例子，存在一个组件，需要从缓存中获取数据，然后渲染。一般情况，我们会如下编写：

```jsx
import React, { Component } from 'react'

class MyComponent extends Component {

  componentWillMount() {
      let data = localStorage.getItem('data');
      this.setState({data});
  }

  render() {
    return <div>{this.state.data}</div>
  }
}
```

上述代码当然可以实现该功能，但是如果还有其他组件也有类似功能的时候，每个组件都需要重复写 `componentWillMount` 中的代码，这明显是冗杂的

下面就可以通过高价组件来进行改写，如下：

JSX | 复制代码

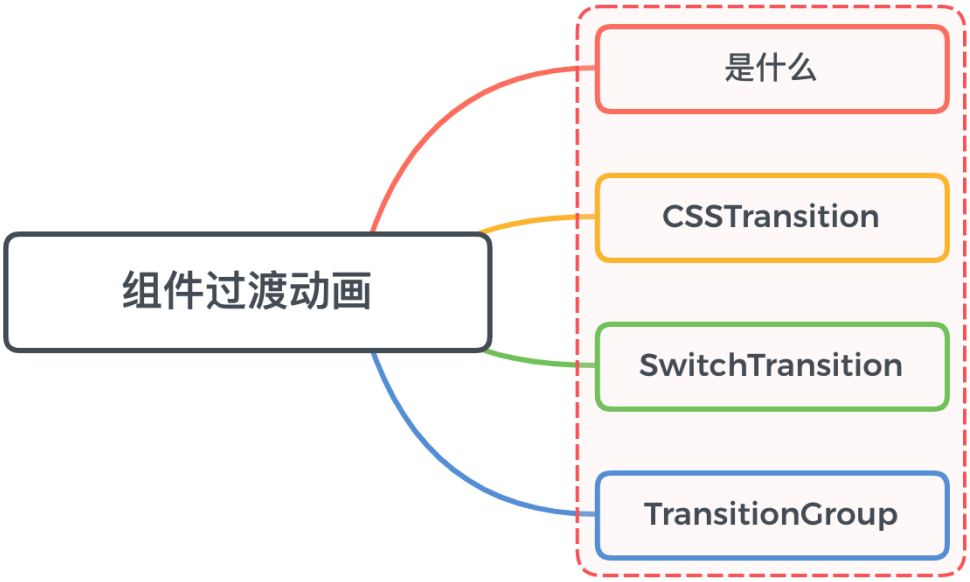
```jsx
import React, { Component } from 'react'

function withPersistentData(WrappedComponent) {
  return class extends Component {
    componentWillMount() {
      let data = localStorage.getItem('data');
        this.setState({data});
    }

    render() {
      // 通过{...this.props} 把传递给当前组件的属性继续传递给被包装的组件WrappedComponent
      return <WrappedComponent data={this.state.data} {...this.props} />
    }
  }
}

class MyComponent2 extends Component {
  render() {
    return <div>{this.props.data}</div>
  }
}

const MyComponentWithPersistentData = withPersistentData(MyComponent2)
```

再比如组件渲染性能监控，如下：

JSX | 复制代码

```jsx
class Home extends React.Component {
    render() {
        return (<h1>Hello World.</h1>);
    }
}
function withTiming(WrappedComponent) {
    return class extends WrappedComponent {
        constructor(props) {
            super(props);
            this.start = 0;
            this.end = 0;
        }
        componentWillMount() {
            super.componentWillMount && super.componentWillMount();
            this.start = Date.now();
        }
        componentDidMount() {
            super.componentDidMount && super.componentDidMount();
            this.end = Date.now();
            console.log(`${WrappedComponent.name} 组件渲染时间为 ${this.end - this.start} ms`);
        }
        render() {
            return super.render();
        }
    };
}

export default withTiming(Home);
```

## 13. 在react中组件间过渡动画如何实现?




## 13.1. 是什么

在日常开发中，页面切换时的转场动画是比较基础的一个场景

当一个组件在显示与消失过程中存在过渡动画，可以很好的增加用户的体验

在 `react` 中实现过渡动画效果会有很多种选择，如 `react-transition-group` ， `react-motion` ， `Animated` ，以及原生的 `CSS` 都能完成切换动画

## 13.2. 如何实现

在 `react` 中， `react-transition-group` 是一种很好的解决方案，其为元素添加 `enter` ， `enter-active` ， `exit` ， `exit-active` 这一系列勾子

可以帮助我们方便的实现组件的入场和离场动画

其主要提供了三个主要的组件：

- CSSTransition：在前端开发中，结合 CSS 来完成过渡动画效果
- SwitchTransition：两个组件显示和隐藏切换时，使用该组件
- TransitionGroup：将多个动画组件包裹在其中，一般用于列表中元素的动画

### 13.2.1. CSSTransition

其实现动画的原理在于，当 `CSSTransition` 的 `in` 属性置为 `true` 时， `CSSTransition` 首先会给其子组件加上 `xxx-enter` 、 `xxx-enter-active` 的 `class` 执行动画

当动画执行结束后，会移除两个 `class` ，并且添加 `-enter-done` 的 `class`

所以可以利用这一点，通过 `css` 的 `transition` 属性，让元素在两个状态之间平滑过渡，从而得到相应的动画效果

当 `in` 属性置为 `false` 时， `CSSTransition` 会给子组件加上 `xxx-exit` 和 `xxx-exit-active` 的 `class` ，然后开始执行动画，当动画结束后，移除两个 `class` ，然后添加 `-enter-done` 的 `class`

如下例子：

JSX | 复制代码

```jsx
export default class App2 extends React.PureComponent {

  state = {show: true};

  onToggle = () => this.setState({show: !this.state.show});

  render() {
    const {show} = this.state;
    return (
      <div className={'container'}>
        <div className={'square-wrapper'}>
          <CSSTransition
            in={show}
            timeout={500}
            classNames={'fade'}
            unmountOnExit={true}
          >
            <div className={'square'} />
          </CSSTransition>
        </div>
        <Button onClick={this.onToggle}>toggle</Button>
      </div>
    );
  }
}
```

对应 `css` 样式如下：

```css
.fade-enter {
  opacity: 0;
  transform: translateX(100%);
}

.fade-enter-active {
  opacity: 1;
  transform: translateX(0);
  transition: all 500ms;
}

.fade-exit {
  opacity: 1;
  transform: translateX(0);
}

.fade-exit-active {
  opacity: 0;
  transform: translateX(-100%);
  transition: all 500ms;
}
```

### 13.2.2. SwitchTransition

`SwitchTransition` 可以完成两个组件之间切换的炫酷动画

比如有一个按钮需要在 `on` 和 `off` 之间切换，我们希望看到 `on` 先从左侧退出， `off` 再从右侧进入

`SwitchTransition` 中主要有一个属性 `mode` ，对应两个值：

- in-out：表示新组件先进入，旧组件再移除；
- out-in：表示就组件先移除，新组建再进入

`SwitchTransition` 组件里面要有 `CSSTransition` ，不能直接包裹你想要切换的组件

里面的 `CSSTransition` 组件不再像以前那样接受 `in` 属性来判断元素是何种状态，取而代之的是 `key` 属性

下面给出一个按钮入场和出场的示例，如下：

JSX | 复制代码

```jsx
import { SwitchTransition, CSSTransition } from "react-transition-group";

export default class SwitchAnimation extends PureComponent {
  constructor(props) {
    super(props);

    this.state = {
      isOn: true
    }
  }

  render() {
    const {isOn} = this.state;

    return (
      <SwitchTransition mode="out-in">
        <CSSTransition classNames="btn"
                       timeout={500}
                       key={isOn ? "on" : "off"}>
          {
          <button onClick={this.btnClick.bind(this)}>
            {isOn ? "on": "off"}
          </button>
        }
        </CSSTransition>
      </SwitchTransition>
    )
  }

  btnClick() {
    this.setState({isOn: !this.state.isOn})
  }
}
```

`css` 文件对应如下：

```css
.btn-enter {
  transform: translate(100%, 0);
  opacity: 0;
}

.btn-enter-active {
  transform: translate(0, 0);
  opacity: 1;
  transition: all 500ms;
}

.btn-exit {
  transform: translate(0, 0);
  opacity: 1;
}

.btn-exit-active {
  transform: translate(-100%, 0);
  opacity: 0;
  transition: all 500ms;
}
```

### 13.2.3. TransitionGroup

当有一组动画的时候，就可将这些 `CSSTransition` 放入到一个 `TransitionGroup` 中来完成动画

同样 `CSSTransition` 里面没有 `in` 属性，用到了 `key` 属性

`TransitionGroup` 在感知 `children` 发生变化的时候，先保存移除的节点，当动画结束后才真正移除

其处理方式如下：

- 插入的节点，先渲染dom，然后再做动画
- 删除的节点，先做动画，然后再删除dom

如下：

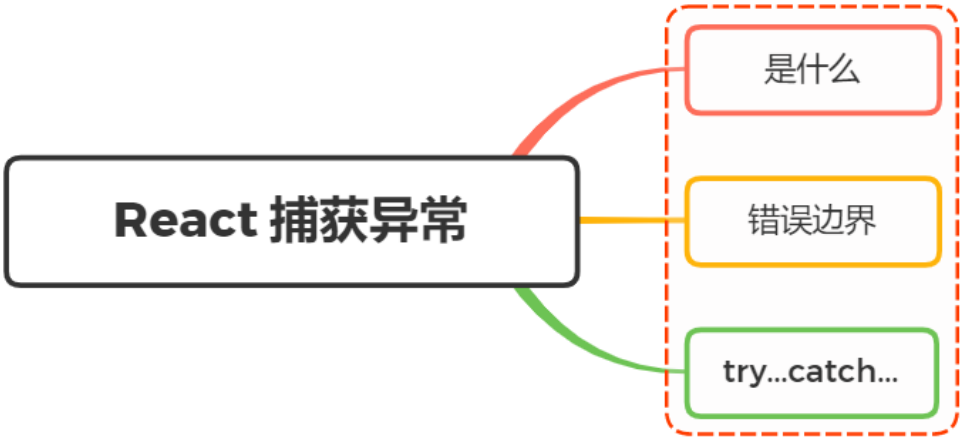
JSX | 复制代码

```jsx
import React, { PureComponent } from 'react'
import { CSSTransition, TransitionGroup } from 'react-transition-group';

export default class GroupAnimation extends PureComponent {
  constructor(props) {
    super(props);

    this.state = {
      friends: []
    }
  }

  render() {
    return (
      <div>
        <TransitionGroup>
          {
            this.state.friends.map((item, index) => {
              return (
                <CSSTransition classNames="friend" timeout={300} key={index}>
                  <div>{item}</div>
                </CSSTransition>
              )
            })
          }
        </TransitionGroup>
        <button onClick={e => this.addFriend()}>+friend</button>
      </div>
    )
  }

  addFriend() {
    this.setState({
      friends: [...this.state.friends, "coderwhy"]
    })
  }
}
```

对应 `css` 如下：

```css
.friend-enter {
    transform: translate(100%, 0);
    opacity: 0;
}

.friend-enter-active {
    transform: translate(0, 0);
    opacity: 1;
    transition: all 500ms;
}

.friend-exit {
    transform: translate(0, 0);
    opacity: 1;
}

.friend-exit-active {
    transform: translate(-100%, 0);
    opacity: 0;
    transition: all 500ms;
}
```

# 14. 说说你在React项目是如何捕获错误的?




## 14.1. 是什么

错误在我们日常编写代码是非常常见的

举个例子，在 `react` 项目中去编写组件内 `JavaScript` 代码错误会导致 `React` 的内部状态被破坏，导致整个应用崩溃，这是不应该出现的现象

作为一个框架， `react` 也有自身对于错误的处理的解决方案

## 14.2. 如何做

为了解决出现的错误导致整个应用崩溃的问题， `react16` 引用了**错误边界**新的概念

错误边界是一种 `React` 组件，这种组件可以捕获发生在其子组件树任何位置的 `JavaScript` 错误，并打印这些错误，同时展示降级 `UI` ，而并不会渲染那些发生崩溃的子组件树

错误边界在渲染期间、生命周期方法和整个组件树的构造函数中捕获错误

形成错误边界组件的两个条件：

- 使用了 static getDerivedStateFromError()
- 使用了 componentDidCatch()

抛出错误后，请使用 `static getDerivedStateFromError()` 渲染备用 UI ，使用 `componentDidCatch()` 打印错误信息，如下：

JSX | 复制代码

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

  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染能够显示降级后的 UI
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // 你同样可以将错误日志上报给服务器
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // 你可以自定义降级后的 UI 并渲染
      return <h1>Something went wrong.</h1>;
    }

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

然后就可以把自身组件的作为错误边界的子组件，如下：

JSX | 复制代码

```jsx
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

下面这些情况无法捕获到异常：

- 事件处理
- 异步代码
- 服务端渲染
- 自身抛出来的错误

在 `react 16` 版本之后，会把渲染期间发生的所有错误打印到控制台

除了错误信息和 JavaScript 栈外，React 16 还提供了组件栈追踪。现在你可以准确地查看发生在组件树内的错误信息：

```
▶React caught an error thrown by BuggyCounter. You should fix this error in your code.    react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
    in BuggyCounter (created by App)
    in ErrorBoundary (created by App)
    in div (created by App)
    in App
```

可以看到在错误信息下方文字中存在一个组件栈，便于我们追踪错误

对于错误边界无法捕获的异常，如事件处理过程中发生问题并不会捕获到，是因为其不会在渲染期间触发，并不会导致渲染时候问题

这种情况可以使用 `js` 的 `try...catch...` 语法，如下：

JSX | 复制代码

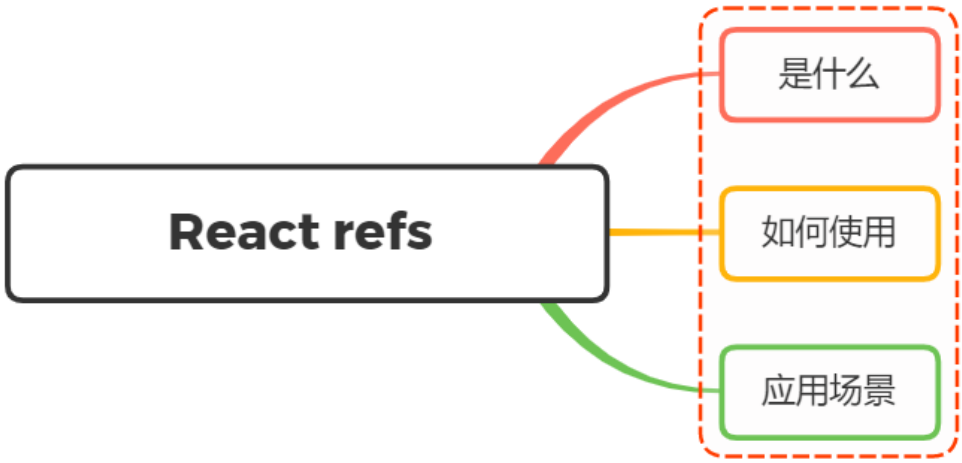
```jsx
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // 执行操作，如有错误则会抛出
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

除此之外还可以通过监听 `onerror` 事件

JavaScript | 复制代码

```javascript
window.addEventListener('error', function(event) { ... })
```

# 15. 说说对React refs 的理解？应用场景？




## 15.1. 是什么

`Refs` 在计算机中称为弹性文件系统（英语：Resilient File System，简称ReFS）

`React` 中的 `Refs` 提供了一种方式，允许我们访问 `DOM` 节点或在 `render` 方法中创建的 `React` 元素

本质为 `ReactDOM.render()` 返回的组件实例，如果是渲染组件则返回的是组件实例，如果渲染 `dom` 则返回的是具体的 `dom` 节点

## 15.2. 如何使用

创建 `ref` 的形式有三种：

- 传入字符串，使用时通过 this.refs.传入的字符串的格式获取对应的元素
- 传入对象，对象是通过 React.createRef() 方式创建出来，使用时获取到创建的对象中存在 current 属性就是对应的元素
- 传入函数，该函数会在 DOM 被挂载时进行回调，这个函数会传入一个 元素对象，可以自己保存，使用时，直接拿到之前保存的元素对象即可
- 传入hook，hook是通过 useRef() 方式创建，使用时通过生成hook对象的 current 属性就是对应的元素

### 15.2.1. 传入字符串

只需要在对应元素或组件中 `ref` 属性

```jsx
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref="myref" />;
  }
}
```

访问当前节点的方式如下：

```javascript
this.refs.myref.innerHTML = "hello";
```

### 15.2.2. 传入对象

`refs` 通过 `React.createRef()` 创建，然后将 `ref` 属性添加到 `React` 元素中，如下：

```jsx
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

当 `ref` 被传递给 `render` 中的元素时，对该节点的引用可以在 `ref` 的 `current` 属性中访问

```javascript
const node = this.myRef.current;
```

### 15.2.3. 传入函数

当 `ref` 传入为一个函数的时候，在渲染过程中，回调函数参数会传入一个元素对象，然后通过实例将对象进行保存

JSX 复制代码

```jsx
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={element => this.myref = element} />;
  }
}
```

获取 `ref` 对象只需要通过先前存储的对象即可

JavaScript 复制代码

```javascript
const node = this.myref
```

### 15.2.4. 传入hook

通过 `useRef` 创建一个 `ref` ，整体使用方式与 `React.createRef` 一致

JSX 复制代码

```jsx
function App(props) {
  const myref = useRef()
  return (
    <>
      <div ref={myref}></div>
    </>
  )
}
```

获取 `ref` 属性也是通过 `hook` 对象的 `current` 属性

JavaScript 复制代码

```javascript
const node = myref.current;
```

上述三种情况都是 `ref` 属性用于原生 `HTML` 元素上，如果 `ref` 设置的组件为一个类组件的时候， `ref` 对象接收到的是组件的挂载实例

注意的是，不能在函数组件上使用 `ref` 属性，因为他们并没有实例

## 15.3. 应用场景

在某些情况下，我们会通过使用 `refs` 来更新组件，但这种方式并不推荐，更多情况我们是通过 `props` 与 `state` 的方式进行去重新渲染子元素

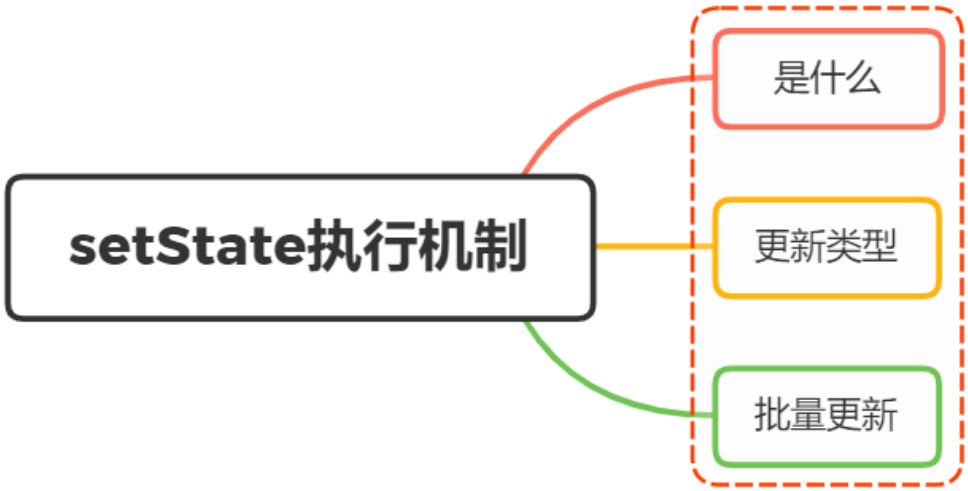
过多使用 `refs` ，会使组件的实例或者是 `DOM` 结构暴露，违反组件封装的原则

例如，避免在 `Dialog` 组件里暴露 `open()` 和 `close()` 方法，最好传递 `isOpen` 属性

但下面的场景使用 `refs` 非常有用：

- 对Dom元素的焦点控制、内容选择、控制
- 对Dom元素的内容设置及媒体播放
- 对Dom元素的操作和对组件实例的操作
- 集成第三方 DOM 库

# 16. 说说 React中的setState执行机制




## 16.1. 是什么

一个组件的显示形态可以由数据状态和外部参数所决定，而数据状态就是 `state`

当需要修改里面的值的状态需要通过调用 `setState` 来改变，从而达到更新组件内部数据的作用

如下例子