setState(arg1,arg2) 括号内的arg1可传入两种参数，一种是对象，一种是函数. 而arg2是更改state之后的回调方法,只能传入方法，但可以为空, 无论采用对象式还是函数式setState都可以传入回调函数作为arg2。并且该回调函数会在arg1所指定的状态更新完，组件重新渲染完以后再调用。

P.S 一般情况下，我们可以选用对象式或者函数式来调用setState更新状态。但是在某些情况下，我们必须采用函数式的调用方法。例如根据现状态进行计算然后设置新状态的值。

**对象式的setState用法 （arg1传入的是对象，字典）**

例如：

//第一种

this.setState({ isAuthenticated: true}); 🡪 第一个arg填入对象，不传入回调函数

this.setState({ isAuthenticated: true},()=>{});

this.setState(

{isAuthenticated: true}, 🡪 第一个arg填入对象

() => console.log(this.state.isAuthenticated + '.')); 🡪 第二个arg填入回调函数

**函数式的setState用法 （arg1传入的是方法）**

这个函数会接收到两个参数，第一个是当前的state值，第二个是当前的props，***这个函数应该返回一个对象***，这个对象代表想要对this.state的更改，换句话说，之前你想给this.setState传递什么对象参数（在采用对象式调用setState那样），在这种函数里就返回什么对象，不过，计算这个对象的方法有些改变，不再依赖于this.state，而是依赖于输入参数state。

这个函数格式是固定的，必须第一个参数是state的以前状态，第二个参数是属性对象props，这两个对象setState会自动传递到函数中去

有两种写法：

//第二种 , 有两种写法 区别在于括号

this.setState((preState, props) => {

return {isAuthenticated: props.isAuthen}

});

this.setState((preState, props) => ({

isAuthenticated: !preState.isAuthenticated

}));

仔细看可以发现，前者是return了一个对象，后者直接写了一个对象，但是后者的对象外层比前者的写法多了个括号，使用时需要注意。

如果你希望在setState之前做些什么 可以使用前者的写法：

this.setState((preState, props) => {

console.log(preState)

return {isAuthenticated: true}

}, () => console.log(this.state.isAuthenticated));

//这个写法意思就是，在改变state中的数据之前，打印出当前的state，然后在更新state之后，

//利用回调打印出this.state.isAuthenticated

**SetState的系统调用时机**

调用setState，组件的state并不会立即改变，setState只是把要修改的状态放入一个队列中，React会优化真正的执行时机，并且React会出于性能原因，可能会将多次setState的状态修改合并成一次状态修改。

**采用函数式调用setState方法解决依据现状态计算更新状态的问题（对象式setState与函数式setState区别）**

如果我们要使用当前的State，计算下个State。 则需要使用函数式调用setState.

例如首先我们采用对象式setState：

function incrementMultiple() {

this.setState({count: this.state.count + 1});

this.setState({count: this.state.count + 1});

this.setState({count: this.state.count + 1});

}

直观上来看，当上面的incrementMultiple函数被调用时，组件状态的count值被增加了3次，每次增加1，那最后count被增加了3，但是，实际上的结果只给state增加了1。

原因并不复杂，就是因为调用this.setState时，并没有立即更改this.state，所以this.setState只是在反复设置同一个值而已，上面的code等同下面这样。

function incrementMultiple() {

const currentCount = this.state.count;

this.setState({count: currentCount + 1});

this.setState({count: currentCount + 1});

this.setState({count: currentCount + 1});

}

currentCount就是一个快照结果，重复地给count设置同一个值，不要说重复3次，哪怕重复一万次，得到的结果也只是增加1而已。

那么如果想要结果是3怎么做呢？这时就需要第二种写法：

\_bsetIncrementMultiple = () => {

this.setState(prevState => ({

count: prevState.count + 1

}));

this.setState(prevState => ({

count: prevState.count + 1

}));

this.setState(prevState => ({

count: prevState.count + 1

}));

}

这样，每一次改变count的时候，都是prevState.count + 1，pervState是前一个状态，每次setState之后，前一个状态都会改变，那么这时候，结果就是想要的3了。

所以我们可以看出：

**React对于传入函数的方式和传入对象的方式进行更新 state 的各自具体理念是不一样的。对于传入对象方式的setState调用，React将合成多次调用，并且在合适的时候一步执行完。而对于传入函数的方式调用setState时， React 会按照各个 setState 的调用顺序，将它们依次放入一个特别的队列，然后，在进行状态更新时，则按照队列中的先后顺序依次调用，并将上一个调用结束时产生的 state 传入到下一个调用的函数中，当然，第一个 setState 调用时，传入的 prevState 则是当前的state ，如此，便解决了传入对象式调用 setState 方法所存在的不能依赖上一次的 state 去计算本次 state 的问题。**

**更新Nested嵌套式状态**

如果状态是nested的结构，并且嵌套层数只有一层，则可以使用…或者Object.Assign进行更新。需要注意的是…或者Object.Assign都是浅复制，所以只能处理嵌套层数只有一层的情况。

例如；

this.state = {

someProperty: {

name: “AAA”

flag:true

}

}

当我们要更新someProperty中的flag为false的时候，

this.setState({ someProperty.flag: false }); // 非法，无法识别someProperty.flag

this.setState({someProperty: {flag:false}};//合法，但是name将会丢失

方式1:采用… ES6的扩展符号

this.setState({

someProperty: {

...this.state.someProperty,

flag: false,

},

});

方式2:采用Object.assign方式

this.setState({

someProperty: Object.assign({}, this.state.someProperty, {

flag: false

}),

});