2022前端高频react面试题集锦

Redux 原理及工作流程

(1) 原理 Redux源码主要分为以下几个模块文件

- compose.js 提供从右到左进行函数式编程
- createStore.js 提供作为生成唯一store的函数
- combineReducers.js 提供合并多个reducer的函数，保证store的唯一性
- bindActionCreators.js 可以让开发者在不直接接触dispacth的前提下进行更改state的操作
- applyMiddleware.js 这个方法通过中间件来增强dispatch的功能

javascript 复制代码

```
const actionTypes = {
    ADD: 'ADD',
    CHANGEINFO: 'CHANGEINFO',
}

const initState = {
    info: '初始化',
}

export default function initReducer(state=initState, action) {
    switch(action.type) {
        case actionTypes.CHANGEINFO:
            return {
                ...state,
                info: action.preload.info || '',
            }
        default:
            return { ...state };
    }
}

export default function createStore(reducer, initialState, middleFunc) {

    if (initialState && typeof initialState === 'function') {
        middleFunc = initialState;
        initialState = undefined;
    }

    let currentState = initialState;
```

```
    const listeners = [];

    if (middleFunc && typeof middleFunc === 'function') {
        // 封装dispatch
        return middleFunc(createStore)(reducer, initialState);
    }

    const getState = () => {
        return currentState;
    }

    const dispatch = (action) => {
        currentState = reducer(currentState, action);

        listeners.forEach(listener => {
            listener();
        })
    }

    const subscribe = (listener) => {
        listeners.push(listener);
    }

    return {
        getState,
        dispatch,
        subscribe
    }
}
```

(2) 工作流程

- const store= createStore（fn）生成数据;
- action: {type: Symble('action01), payload:'payload' }定义行为;
- dispatch发起action：store.dispatch(doSomething('action001'));
- reducer：处理action，返回新的state;

通俗点解释：

- 首先，用户（通过View）发出Action，发出方式就用到了dispatch方法
- 然后，Store自动调用Reducer，并且传入两个参数：当前State和收到的Action，Reducer会返回新的State
- State一旦有变化，Store就会调用监听函数，来更新View

以 store 为核心，可以把它看成数据存储中心，但是他要更改数据的时候不能直接修改，数据修改更新的角色由Reducers来担任，store只做存储，中间人，当Reducers的更新完成以后会通过store的订阅来通知react component，组件把新的状态重新获取渲染，组件中也能主动发送action，创建action后这个动作是不会执行的，所以要dispatch这个action，让store通过reducers去做更新React Component 就是react的每个组件。

当调用 `setState` 时，React `render` 是如何工作的?

咱们可以将" `render` "分为两个步骤：

1. 虚拟 DOM 渲染:当 `render` 方法被调用时，它返回一个新的组件的虚拟 DOM 结构。当调用 `setState()` 时， `render` 会被再次调用，因为默认情况下 `shouldComponentUpdate` 总是返回 `true` ，所以默认情况下 React 是没有优化的。

2. 原生 DOM 渲染:React 只会在虚拟DOM中修改真实DOM节点，而且修改的次数非常少——这是很棒的React特性，它优化了真实DOM的变化，使React变得更快。

如何解决 props 层级过深的问题

- 使用Context API：提供一种组件之间的状态共享，而不必通过显式组件树逐层传递props；
- 使用Redux等状态库。

React Hook 的使用限制有哪些?

React Hooks 的限制主要有两条：

- 不要在循环、条件或嵌套函数中调用 Hook；
- 在 React 的函数组件中调用 Hook。

那为什么会有这样的限制呢？Hooks 的设计初衷是为了改进 React 组件的开发模式。在旧有的开发模式下遇到了三个问题。

- 组件之间难以复用状态逻辑。过去常见的解决方案是高阶组件、render props 及状态管理框架。
- 复杂的组件变得难以理解。生命周期函数与业务逻辑耦合太深，导致关联部分难以拆分。
- 人和机器都很容易混淆类。常见的有 this 的问题，但在 React 团队中还有类难以优化的问题，希望在编译优化层面做出一些改进。

这三个问题在一定程度上阻碍了 React 的后续发展，所以为了解决这三个问题，Hooks **基于函数组件**开始设计。然而第三个问题决定了 Hooks 只支持函数组件。

那为什么不要在循环、条件或嵌套函数中调用 Hook 呢？因为 Hooks 的设计是基于数组实现。在调用时按顺序加入数组中，如果使用循环、条件或嵌套函数很有可能导致数组取值错位，执行错误的 Hook。当然，实质上 React 的源码里不是数组，是链表。

这些限制会在编码上造成一定程度的心智负担，新手可能会写错，为了避免这样的情况，可以引入 ESLint 的 Hooks 检查插件进行预防。

为什么列表循环渲染的key最好不要用index

举例说明

```javascript 复制代码
变化前数组的值是[1,2,3,4]，key就是对应的下标：0，1，2，3
变化后数组的值是[4,3,2,1]，key对应的下标也是：0，1，2，3
```

- 那么diff算法在变化前的数组找到key =0的值是1，在变化后数组里找到的key=0的值是4
- 因为子元素不一样就重新删除并更新
- 但是如果加了唯一的key,如下

```javascript 复制代码
变化前数组的值是[1,2,3,4]，key就是对应的下标：id0，id1，id2，id3
变化后数组的值是[4,3,2,1]，key对应的下标也是：id3，id2，id1，id0
```

- 那么diff算法在变化前的数组找到key =id0的值是1，在变化后数组里找到的key=id0的值也是1
- 因为子元素相同，就不删除并更新，只做移动操作，这就提升了性能

Redux 状态管理器和变量挂载到 window 中有什么区别

两者都是存储数据以供后期使用。但是Redux状态更改可回溯——Time travel，数据多了的时候可以很清晰的知道改动在哪里发生，完整的提供了一套状态管理模式。

随着 JavaScript 单页应用开发日趋复杂，JavaScript 需要管理比任何时候都要多的 state （状态）。 这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括 UI状态，如激活的路由，被选中的标签，是否显示加载动效或者分页器等等。

管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个model 的变化，依次地，可能会引起另一个 view 的变化。直至你搞不清楚到底发生了什么。state 在什么时候，由于什么原因，如何变化已然不受控制。 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。如果这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据等等。前端开发者正在经受前所未有的复杂性，难道就这么放弃了吗?当然不是。

这里的复杂性很大程度上来自于：我们总是将两个难以理清的概念混淆在一起：变化和异步。可以称它们为曼妥思和可乐。如果把二者分开，能做的很好，但混到一起，就变得一团糟。一些库如 React 视图在视图层禁止异步和直接操作 DOM来解决这个问题。美中不足的是，React 依旧把处理 state 中数据的问题留给了你。Redux就是为了帮你解决这个问题。

参考 前端进阶面试题详细解答

React setState 调用之后发生了什么？是同步还是异步?

（1）React中setState后发生了什么

在代码中调用setState函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发调和过程(Reconciliation)。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个UI界面。

在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

如果在短时间内频繁setState。React会将state的改变压入栈中，在合适的时机，批量更新state和视图，达到提高性能的效果。

（2）setState 是同步还是异步的

假如所有setState是同步的，意味着每执行一次setState时（有可能一个同步代码中，多次setState），都重新vnode diff + dom修改，这对性能来说是极为不好的。如果是异步，则可以把一个同步代码中的多个setState合并成一次组件更新。所以默认是异步的，但是在一些情况下是同步的。

setState 并不是单纯同步/异步的，它的表现会因调用场景的不同而不同。在源码中，通过 isBatchingUpdates 来判断setState 是先存进 state 队列还是直接更新，如果值为 true 则执行

异步操作，为 false 则直接更新。

- **异步：** 在 React 可以控制的地方，就为 true，比如在 React 生命周期事件和合成事件中，都会走合并操作，延迟更新的策略。
- **同步：** 在 React 无法控制的地方，比如原生事件，具体就是在 addEventListener 、setTimeout、setInterval 等事件中，就只能同步更新。

一般认为，做异步设计是为了性能优化、减少渲染次数：

- `setState` 设计为异步，可以显著的提升性能。如果每次调用 `setState` 都进行一次更新，那么意味着 `render` 函数会被频繁调用，界面重新渲染，这样效率是很低的；最好的办法应该是获取到多个更新，之后进行批量更新；
- 如果同步更新了 `state` ，但是还没有执行 `render` 函数，那么 `state` 和 `props` 不能保持同步。 `state` 和 `props` 不能保持一致性，会在开发中产生很多的问题；

React-Router的实现原理是什么?

客户端路由实现的思想：

- 基于 hash 的路由：通过监听 `hashchange` 事件，感知 hash 的变化
  - 改变 hash 可以直接通过 location.hash=xxx
- 基于 H5 history 路由：
  - 改变 url 可以通过 history.pushState 和 resplaceState 等，会将URL压入堆栈，同时能够应用 `history.go()` 等 API
  - 监听 url 的变化可以通过自定义事件触发实现

react-router 实现的思想：

- 基于 `history` 库来实现上述不同的客户端路由实现思想，并且能够保存历史记录等，磨平浏览器差异，上层无感知
- 通过维护的列表，在每次 URL 发生变化的回收，通过配置的 路由路径，匹配到对应的 Component，并且 render

React.createClass和extends Component的区别有哪些?

React.createClass和extends Component的bai区别主要在于：

（1）语法区别

- createClass本质上是一个工厂函数，extends的方式更加接近最新的ES6规范的class写法。两种方式在语法上的差别主要体现在方法的定义和静态属性的声明上。
- createClass方式的方法定义使用逗号，隔开，因为creatClass本质上是一个函数，传递给它的是一个Object；而class的方式定义方法时务必谨记不要使用逗号隔开，这是ES6 class的语法规范。

(2) propType 和 getDefaultProps

- React.createClass：通过proTypes对象和getDefaultProps()方法来设置和获取props.
- React.Component：通过设置两个属性propTypes和defaultProps

(3) 状态的区别

- React.createClass：通过getInitialState()方法返回一个包含初始值的对象
- React.Component：通过constructor设置初始状态

(4) this区别

- React.createClass：会正确绑定this
- React.Component：由于使用了 ES6，这里会有些微不同，属性并不会自动绑定到 React 类的实例上。

(5) Mixins

- React.createClass：使用 React.createClass 的话，可以在创建组件时添加一个叫做 mixins 的属性，并将可供混合的类的集合以数组的形式赋给 mixins。
- 如果使用 ES6 的方式来创建组件，那么 `React mixins` 的特性将不能被使用了。

React 事件机制

```javascript
<div onClick={this.handleClick.bind(this)}>点我</div>
```


React并不是将click事件绑定到了div的真实DOM上，而是在document处监听了所有的事件，当事件发生并且冒泡到document处的时候，React将事件内容封装并交由真正的处理函数运行。这样的方式不仅仅减少了内存的消耗，还能在组件挂在销毁时统一订阅和移除事件。

除此之外，冒泡到document上的事件也不是原生的浏览器事件，而是由react自己实现的合成事件（SyntheticEvent）。因此如果不想要是事件冒泡的话应该调用event.preventDefault()方法，而不是调用event.stopProppagation()方法。 JSX 上写的事件并没有绑定在对应的真实 DOM 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 `document` 上。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。

另外冒泡到 `document` 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent）。因此我们如果不想要事件冒泡的话，调用 `event.stopPropagation` 是无效的，而应该调用 `event.preventDefault` 。

实现合成事件的目的如下：

- 合成事件首先抹平了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力；
- 对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，当事件需要被使用时，就会从池子中复用对象，事件回调结束后，就会销毁事件对象上的属性，从而便于下次复用事件对象。

在React中组件的props改变时更新组件的有哪些方法?

在一个组件传入的props更新时重新渲染该组件常用的方法是在 `componentWillReceiveProps` 中将新的props更新到组件的state中（这种state被成为派生状态（Derived State）），从而实现重新渲染。React 16.3中还引入了一个新的钩子函数 `getDerivedStateFromProps` 来专门实现这一需求。

（1）componentWillReceiveProps（已废弃）

在react的componentWillReceiveProps(nextProps)生命周期中，可以在子组件的render函数执行前，通过this.props获取旧的属性，通过nextProps获取新的props，对比两次props是否相同，从而更新子组件自己的state。

这样的好处是，可以将数据请求放在这里进行执行，需要传的参数则从componentWillReceiveProps(nextProps)中获取。而不必将所有的请求都放在父组件中。于是该请求只会在该组件渲染时才会发出，从而减轻请求负担。

（2）getDerivedStateFromProps（16.3引入）

这个生命周期函数是为了替代 `componentWillReceiveProps` 存在的，所以在需要使用 `componentWillReceiveProps` 时，就可以考虑使用 `getDerivedStateFromProps` 来进行替代。

两者的参数是不相同的，而 `getDerivedStateFromProps` 是一个静态函数，也就是这个函数不能通过this访问到class的属性，也并不推荐直接访问属性。而是应该通过参数提供的nextProps以及prevState来进行判断，根据新传入的props来映射到state。

需要注意的是，**如果props传入的内容不需要影响到你的state，那么就需要返回一个null**，这个返回值是必须的，所以尽量将其写到函数的末尾：

```javascript 复制代码
static getDerivedStateFromProps(nextProps, prevState) {
    const {type} = nextProps;
    // 当传入的type发生变化的时候，更新state
    if (type !== prevState.type) {
        return {
            type,
        };
    }
    // 否则，对于state不进行任何操作
    return null;
}
```

在React中页面重新加载时怎样保留数据?

这个问题就设计到了**数据持久化，** 主要的实现方式有以下几种：

- **Redux：** 将页面的数据存储在redux中，在重新加载页面时，获取Redux中的数据；
- **data.js：** 使用webpack构建的项目，可以建一个文件，data.js，将数据保存data.js中，跳转页面后获取；
- **sessionStorge：** 在进入选择地址页面之前，componentWillUnMount的时候，将数据存储到sessionStorage中，每次进入页面判断sessionStorage中有没有存储的那个值，有，则读取渲染数据；没有，则说明数据是初始化的状态。返回或进入除了选择地址以外的页面，清掉存储的sessionStorage，保证下次进入是初始化的数据
- **history API：** History API 的 `pushState` 函数可以给历史记录关联一个任意的可序列化 `state` ，所以可以在路由 `push` 的时候将当前页面的一些信息存到 `state` 中，下次返回到这个页面的时候就能从 `state` 里面取出离开前的数据重新渲染。react-router 直接可以支持。这个方法适合一些需要临时存储的场景。

Redux 中异步的请求怎么处理

可以在 componentDidmount 中直接进行请求无须借助redux。但是在一定规模的项目中,上述方法很难进行异步流的管理,通常情况下我们会借助redux的异步中间件进行异步处理。redux异步流中间件其实有很多，当下主流的异步中间件有两种redux-thunk、redux-saga。

（1）使用react-thunk中间件

redux-thunk优点:

- 体积小: redux-thunk的实现方式很简单,只有不到20行代码
- 使用简单: redux-thunk没有引入像redux-saga或者redux-observable额外的范式,上手简单

redux-thunk缺陷:

- 样板代码过多: 与redux本身一样,通常一个请求需要大量的代码,而且很多都是重复性质的
- 耦合严重: 异步操作与redux的action偶合在一起,不方便管理
- 功能孱弱: 有一些实际开发中常用的功能需要自己进行封装

使用步骤:

- 配置中间件，在store的创建中配置

javascript 复制代码

```
import {createStore, applyMiddleware, compose} from 'redux';
import reducer from './reducer';
import thunk from 'redux-thunk'

// 设置调试工具
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ? window.__REDUX_DEVTOOLS_EXTEN
// 设置中间件
const enhancer = composeEnhancers(
  applyMiddleware(thunk)
);

const store = createStore(reducer, enhancer);

export default store;
```

- 添加一个返回函数的actionCreator，将异步请求逻辑放在里面

```javascript 复制代码
/**  发送get请求，并生成相应action，更新store的函数  @param url {string} 请求地址  @param func {functio
// dispatch为自动接收的store.dispatch函数
export const getHttpAction = (url, func) => (dispatch) => {
    axios.get(url).then(function(res){
        const action = func(res.data)
        dispatch(action)
    })
}
```

- 生成action，并发送action

```javascript 复制代码
componentDidMount(){
    var action = getHttpAction('/getData', getInitTodoItemAction)
    // 发送函数类型的action时，该action的函数体会自动执行
    store.dispatch(action)
}
```

（2）使用redux-saga中间件

redux-saga优点:

- 异步解耦: 异步操作被被转移到单独 saga.js 中，不再是掺杂在 action.js 或 component.js 中
- action摆脱thunk function: dispatch 的参数依然是一个纯粹的 action (FSA)，而不是充满 “黑魔法” thunk function
- 异常处理: 受益于 generator function 的 saga 实现，代码异常/请求失败 都可以直接通过 try/catch 语法直接捕获处理
- 功能强大: redux-saga提供了大量的Saga 辅助函数和Effect 创建器供开发者使用,开发者无须封装或者简单封装即可使用
- 灵活: redux-saga可以将多个Saga可以串行/并行组合起来,形成一个非常实用的异步flow
- 易测试，提供了各种case的测试方案，包括mock task，分支覆盖等等

redux-saga缺陷:

- 额外的学习成本: redux-saga不仅在使用难以理解的 generator function,而且有数十个API,学习成本远超redux-thunk,最重要的是你的额外学习成本是只服务于这个库的,与redux-observable不同,redux-observable虽然也有额外学习成本但是背后是rxjs和一整套思想

- 体积庞大: 体积略大,代码近2000行，min版25KB左右
- 功能过剩: 实际上并发控制等功能很难用到,但是我们依然需要引入这些代码
- ts支持不友好: yield无法返回TS类型

redux-saga可以捕获action，然后执行一个函数，那么可以把异步代码放在这个函数中，使用步骤如下：

- 配置中间件

javascript 复制代码

```
import {createStore, applyMiddleware, compose} from 'redux';
import reducer from './reducer';
import createSagaMiddleware from 'redux-saga'
import TodoListSaga from './sagas'

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ? window.__REDUX_DEVTOOLS_EXTEN
const sagaMiddleware = createSagaMiddleware()

const enhancer = composeEnhancers(
  applyMiddleware(sagaMiddleware)
);

const store = createStore(reducer, enhancer);
sagaMiddleware.run(TodoListSaga)

export default store;
```

- 将异步请求放在sagas.js中

javascript 复制代码

```
import {takeEvery, put} from 'redux-saga/effects'
import {initTodoList} from './actionCreator'
import {GET_INIT_ITEM} from './actionTypes'
import axios from 'axios'

function* func(){
    try{
        // 可以获取异步返回数据
        const res = yield axios.get('/getData')
        const action = initTodoList(res.data)
        // 将action发送到reducer
        yield put(action)
    }catch(e){
        console.log('网络请求失败')
    }
}
```

```
function* mySaga(){
    // 自动捕获GET_INIT_ITEM类型的action，并执行func
    yield takeEvery(GET_INIT_ITEM, func)
}

export default mySaga
```

- 发送action

javascript 复制代码

```
componentDidMount(){
  const action = getInitTodoItemAction()
  store.dispatch(action)
}
```

当调用 setState的时候，发生了什么操作？**

当调用 setState时， React做的第一件事是将传递给setState的对象合并到组件的当前状态，这将启动一个称为和解（ reconciliation）的过程。 和解的最终目标是，根据这个新的状态以最有效的方式更新DOM。 为此， React将构建一个新的 React虚拟DOM树（可以将其视为页面DOM元素的对象表示方式）。 一旦有了这个DOM树，为了弄清DOM是如何响应新的状态而改变的， React会将这个新树与上一个虚拟DOM树比较。 这样做， React会知道发生的确切变化，并且通过了解发生的变化后，在绝对必要的情况下进行更新DOM，即可将因操作DOM而占用的空间最小化。

React中setState的第二个参数作用是什么?

`setState` 的第二个参数是一个可选的回调函数。这个回调函数将在组件重新渲染后执行。等价于在 `componentDidUpdate` 生命周期内执行。通常建议使用 `componentDidUpdate` 来代替此方式。在这个回调函数中你可以拿到更新后 `state` 的值：

javascript 复制代码

```
this.setState({
    key1: newState1,
    key2: newState2,
    ...
}, callback) // 第二个参数是 state 更新完成后的回调函数
```

什么是 Props

Props 是 React 中属性的简写。它们是只读组件，必须保持纯，即不可变。它们总是在整个应用中从父组件传递到子组件。子组件永远不能将 prop 送回父组件。这有助于维护单向数据流，通常用于呈现动态生成的数据。

在使用 React Router时，如何获取当前页面的路由或浏览器中地址栏中的地址?

在当前组件的 props中，包含 location属性对象，包含当前页面路由地址信息，在 match中存储当前路由的参数等数据信息。可以直接通过 this .props使用它们。

哪些方法会触发 React 重新渲染？重新渲染 render 会做些什么?

（1）哪些方法会触发 react 重新渲染?

- **setState（）方法被调用**

setState 是 React 中最常用的命令，通常情况下，执行 setState 会触发 render。但是这里有个点值得关注，执行 setState 的时候不一定会重新渲染。当 setState 传入 null 时，并不会触发 render。

javascript 复制代码

```
class App extends React.Component {
  state = {
    a: 1
  };

  render() {
    console.log("render");
    return (
      <React.Fragement>
        <p>{this.state.a}</p>
        <button
          onClick={() => {            this.setState({ a: 1 }); // 这里并没有改变 a 的值          }}
        <button onClick={() => this.setState(null)}>setState null</button>
        <Child />
      </React.Fragement>
    );
  }
}
```

- **父组件重新渲染**

只要父组件重新渲染了，即使传入子组件的 props 未发生变化，那么子组件也会重新渲染，进而触发 render

（2）重新渲染 render 会做些什么?

- 会对新旧 VNode 进行对比，也就是我们所说的Diff算法。
- 对新旧两棵树进行一个深度优先遍历，这样每一个节点都会一个标记，在到深度遍历的时候，每遍历到一和个节点，就把该节点和新的节点树进行对比，如果有差异就放到一个对象里面
- 遍历差异对象，根据差异的类型，根据对应对规则更新VNode

React 的处理 render 的基本思维模式是每次一有变动就会去重新渲染整个应用。在 Virtual DOM 没有出现之前，最简单的方法就是直接调用 innerHTML。Virtual DOM厉害的地方并不是说它比直接操作 DOM 快，而是说不管数据怎么变，都会尽量以最小的代价去更新 DOM。React 将 render 函数返回的虚拟 DOM 树与老的进行比较，从而确定 DOM 要不要更新、怎么更新。当 DOM 树很大时，遍历两棵树进行各种比对还是相当耗性能的，特别是在顶层 setState 一个微小的修改，默认会去遍历整棵树。尽管 React 使用高度优化的 Diff 算法，但是这个过程仍然会损耗性能.

Hooks可以取代 `render props` 和高阶组件吗?

通常， `render props` 和高阶组件仅渲染一个子组件。React团队认为，Hooks 是服务此用例的更简单方法。 这两种模式仍然有一席之地(例如，一个虚拟的 `scroller` 组件可能有一个 `renderItem prop` ，或者一个可视化的容器组件可能有它自己的 DOM 结构)。但在大多数情况下，Hooks 就足够了，可以帮助减少树中的嵌套。

一般可以用哪些值作为key

- 最好使用每一条数据中的唯一标识作为key，比如：手机号，id值，身份证号，学号等

- 也可以用数据的索引值（可能会出现一些问题）