再将 `v-for` 与 `v-if` 置于不同标签

HTML | 复制代码

```html
<div id="app">
    <template v-if="isShow">
        <p v-for="item in items">{{item.title}}</p>
    </template>
</div>
```

再输出下 `render` 函数

JavaScript | 复制代码

```javascript
ƒ anonymous() {
  with(this){return
    _c('div',{attrs:{"id":"app"}},
    [(isShow)?[_v("\n"),
    _l((items),function(item){return _c('p',[_v(_s(item.title))])})]:_e()],
2)}
}
```

这时候我们可以看到， `v-for` 与 `v-if` 作用在不同标签时候，是先进行判断，再进行列表的渲染

我们再在查看下 `vue` 源码

源码位置： `\vue-dev\src\compiler\codegen\index.js`

```JavaScript
export function genElement (el: ASTElement, state: CodegenState): string {
  if (el.parent) {
    el.pre = el.pre || el.parent.pre
  }
  if (el.staticRoot && !el.staticProcessed) {
    return genStatic(el, state)
  } else if (el.once && !el.onceProcessed) {
    return genOnce(el, state)
  } else if (el.for && !el.forProcessed) {
    return genFor(el, state)
  } else if (el.if && !el.ifProcessed) {
    return genIf(el, state)
  } else if (el.tag === 'template' && !el.slotTarget && !state.pre) {
    return genChildren(el, state) || 'void 0'
  } else if (el.tag === 'slot') {
    return genSlot(el, state)
  } else {
    // component or element
    ...
}
```

在进行 `if` 判断的时候， `v-for` 是比 `v-if` 先进行判断

最终结论： `v-for` 优先级比 `v-if` 高

6.3. 注意事项

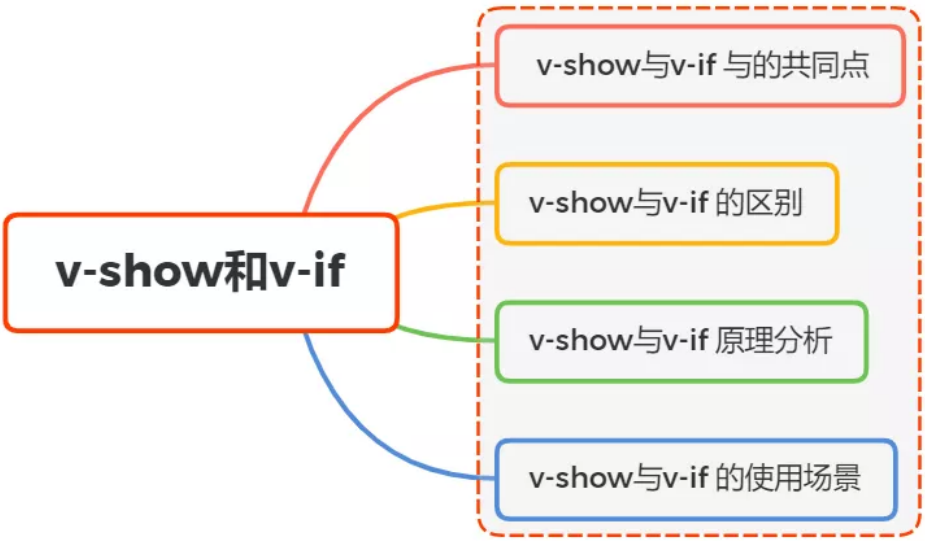
1. 永远不要把 `v-if` 和 `v-for` 同时用在同一个元素上，带来性能方面的浪费（每次渲染都会先循环再进行条件判断）
2. 如果避免出现这种情况，则在外层嵌套 `template` （页面渲染不生成 `dom` 节点），在这一层进行v-if判断，然后在内部进行v-for循环

```JavaScript
<template v-if="isShow">
    <p v-for="item in items">
</template>
```

3. 如果条件出现在循环内部，可通过计算属性 `computed` 提前过滤掉那些不需要显示的项

```javascript
computed: {
    items: function() {
      return this.list.filter(function (item) {
        return item.isShow
      })
    }
}
```

7. v-show和v-if有什么区别？使用场景分别是什么？



7.1. v-show与v-if的共同点

我们都知道在 `vue` 中 `v-show` 与 `v-if` 的作用效果是相同的(不含v-else)，都能控制元素在页面是否显示

在用法上也是相同的

```javascript
<Model v-show="isShow" />
<Model v-if="isShow" />
```

- 当表达式为 `true` 的时候，都会占据页面的位置

- 当表达式都为 `false` 时，都不会占据页面位置

7.2. v-show与v-if的区别

- 控制手段不同
- 编译过程不同
- 编译条件不同

控制手段： `v-show` 隐藏则是为该元素添加 `css--display:none` ， `dom` 元素依旧还在。 `v-if` 显示隐藏是将 `dom` 元素整个添加或删除

编译过程： `v-if` 切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件； `v-show` 只是简单的基于css切换

编译条件： `v-if` 是真正的条件渲染，它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。只有渲染条件为假时，并不做操作，直到为真才渲染

- `v-show` 由 `false` 变为 `true` 的时候不会触发组件的生命周期
- `v-if` 由 `false` 变为 `true` 的时候，触发组件的 `beforeCreate` 、 `create` 、 `beforeMount` 、 `mounted` 钩子，由 `true` 变为 `false` 的时候触发组件的 `beforeDestory` 、 `destoryed` 方法

性能消耗： `v-if` 有更高的切换消耗； `v-show` 有更高的初始渲染消耗；

7.3. v-show与v-if原理分析

具体解析流程这里不展开讲，大致流程如下

- 将模板 `template` 转为 `ast` 结构的 `JS` 对象
- 用 `ast` 得到的 `JS` 对象拼装 `render` 和 `staticRenderFns` 函数
- `render` 和 `staticRenderFns` 函数被调用后生成虚拟 `VNODE` 节点，该节点包含创建 `DOM` 节点所需信息
- `vm.patch` 函数通过虚拟 `DOM` 算法利用 `VNODE` 节点创建真实 `DOM` 节点

7.3.1. v-show原理

不管初始条件是什么，元素总是会被渲染

我们看一下在 `vue` 中是如何实现的

代码很好理解，有 `transition` 就执行 `transition` ，没有就直接设置 `display` 属性

JavaScript | 复制代码

```javascript
// https://github.com/vuejs/vue-next/blob/3cd30c5245da0733f9eb6f29d220f39c46518162/packages/runtime-dom/src/directives/vShow.ts
export const vShow: ObjectDirective<VShowElement> = {
  beforeMount(el, { value }, { transition }) {
    el._vod = el.style.display === 'none' ? '' : el.style.display
    if (transition && value) {
      transition.beforeEnter(el)
    } else {
      setDisplay(el, value)
    }
  },
  mounted(el, { value }, { transition }) {
    if (transition && value) {
      transition.enter(el)
    }
  },
  updated(el, { value, oldValue }, { transition }) {
    // ...
  },
  beforeUnmount(el, { value }) {
    setDisplay(el, value)
  }
}
```

7.3.2. v-if原理

`v-if` 在实现上比 `v-show` 要复杂的多，因为还有 `else` `else-if` 等条件需要处理，这里我们也只摘抄源码中处理 `v-if` 的一小部分

返回一个 `node` 节点， `render` 函数通过表达式的值来决定是否生成 `DOM`

JavaScript | 复制代码

```javascript
// https://github.com/vuejs/vue-next/blob/cdc9f336fd/packages/compiler-core/src/transforms/vIf.ts
export const transformIf = createStructuralDirectiveTransform(
  /^(if|else|else-if)$/,
  (node, dir, context) => {
    return processIf(node, dir, context, (ifNode, branch, isRoot) => {
      // ...
      return () => {
        if (isRoot) {
          ifNode.codegenNode = createCodegenNodeForBranch(
            branch,
            key,
            context
          ) as IfConditionalExpression
        } else {
          // attach this branch's codegen node to the v-if root.
          const parentCondition = getParentCondition(ifNode.codegenNode!)
          parentCondition.alternate = createCodegenNodeForBranch(
            branch,
            key + ifNode.branches.length - 1,
            context
          )
        }
      }
    })
  }
)
```

7.4. v-show与v-if的使用场景

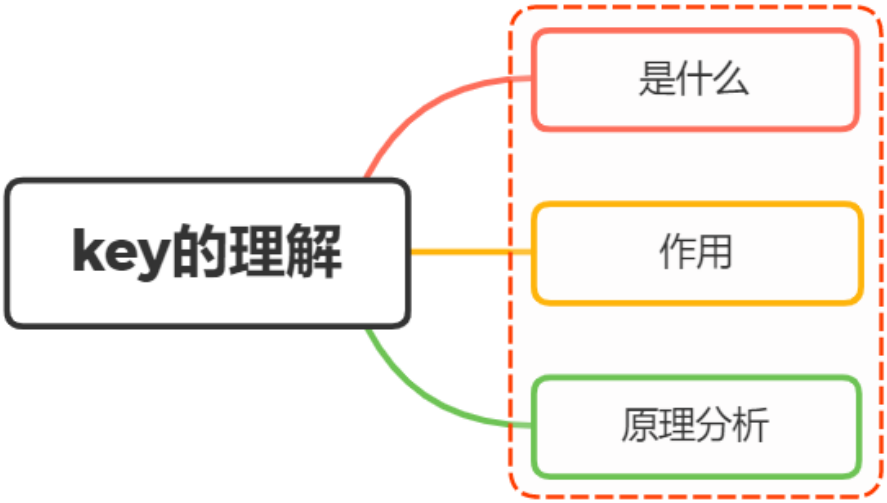
`v-if` 与 `v-show` 都能控制 `dom` 元素在页面的显示

`v-if` 相比 `v-show` 开销更大的（直接操作 `dom` 节点增加与删除）

如果需要非常频繁地切换，则使用 v-show 较好

如果在运行时条件很少改变，则使用 v-if 较好

8. 你知道vue中key的原理吗？说说你对它的理解



8.1. Key是什么

开始之前，我们先还原两个实际工作场景

1. 当我们在使用 `v-for` 时，需要给单元加上 `key`

JavaScript 复制代码

```
<ul>
    <li v-for="item in items" :key="item.id">...</li>
</ul>
```

2. 用 `+new Date()` 生成的时间戳作为 `key` ，手动强制触发重新渲染

JavaScript 复制代码

```
<Comp :key="+new Date()" />
```

那么这背后的逻辑是什么， `key` 的作用又是什么?

一句话来讲

> key是给每一个vnode的唯一id，也是diff的一种优化策略，可以根据key，更准确， 更快的找到对应的vnode节点

8.1.1. 场景背后的逻辑

当我们在使用 `v-for` 时，需要给单元加上 `key`

- 如果不用key，Vue会采用就地复地原则：最小化element的移动，并且会尝试尽最大程度在同适当

的地方对相同类型的element，做patch或者reuse。

- 如果使用了key，Vue会根据keys的顺序记录element，曾经拥有了key的element如果不再出现的话，会被直接remove或者destoryed

用 `+new Date()` 生成的时间戳作为 `key` ，手动强制触发重新渲染

- 当拥有新值的rerender作为key时，拥有了新key的Comp出现了，那么旧key Comp会被移除，新key Comp触发渲染

8.2. 设置key与不设置key区别

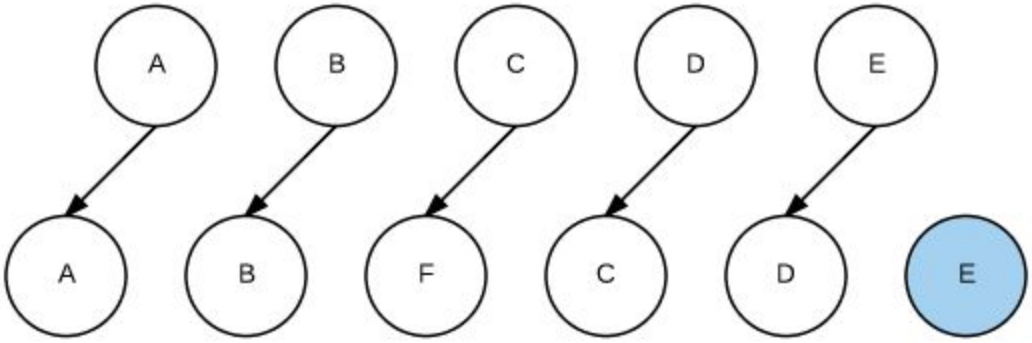
举个例子：

创建一个实例，2秒后往 `items` 数组插入数据

HTML | 复制代码

```html
<body>
  <div id="demo">
    <p v-for="item in items" :key="item">{{item}}</p>
  </div>
  <script src="../../dist/vue.js"></script>
  <script>
    // 创建实例
    const app = new Vue({
      el: '#demo',
      data: { items: ['a', 'b', 'c', 'd', 'e'] },
      mounted () {
        setTimeout(() => {
          this.items.splice(2, 0, 'f')  //
        }, 2000);
      },
    });
  </script>
</body>
```

在不使用 `key` 的情况， `vue` 会进行这样的操作：



分析下整体流程：

- 比较A，A，相同类型的节点，进行 `patch` ，但数据相同，不发生 `dom` 操作
- 比较B，B，相同类型的节点，进行 `patch` ，但数据相同，不发生 `dom` 操作
- 比较C，F，相同类型的节点，进行 `patch` ，数据不同，发生 `dom` 操作
- 比较D，C，相同类型的节点，进行 `patch` ，数据不同，发生 `dom` 操作
- 比较E，D，相同类型的节点，进行 `patch` ，数据不同，发生 `dom` 操作
- 循环结束，将E插入到 `DOM` 中

一共发生了3次更新，1次插入操作

在使用 `key` 的情况： `vue` 会进行这样的操作：

- 比较A，A，相同类型的节点，进行 `patch` ，但数据相同，不发生 `dom` 操作
- 比较B，B，相同类型的节点，进行 `patch` ，但数据相同，不发生 `dom` 操作
- 比较C，F，不相同类型的节点
  - 比较E、E，相同类型的节点，进行 `patch` ，但数据相同，不发生 `dom` 操作
- 比较D、D，相同类型的节点，进行 `patch` ，但数据相同，不发生 `dom` 操作
- 比较C、C，相同类型的节点，进行 `patch` ，但数据相同，不发生 `dom` 操作
- 循环结束，将F插入到C之前

一共发生了0次更新，1次插入操作

通过上面两个小例子，可见设置 `key` 能够大大减少对页面的 `DOM` 操作，提高了 `diff` 效率

8.2.1. 设置key值一定能提高diff效率吗?

其实不然，文档中也明确表示

> 当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在

> 特定索引下显示已被渲染过的每个元素

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出

建议尽可能在使用 `v-for` 时提供 `key`，除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升

8.3. 原理分析

源码位置：core/vdom/patch.js

这里判断是否为同一个 `key`，首先判断的是 `key` 值是否相等如果没有设置 `key`，那么 `key` 为 `undefined`，这时候 `undefined` 是恒等于 `undefined`

JavaScript | 复制代码

```
function sameVnode (a, b) {
    return (
        a.key === b.key && (
            (
                a.tag === b.tag &&
                a.isComment === b.isComment &&
                isDef(a.data) === isDef(b.data) &&
                sameInputType(a, b)
            ) || (
                isTrue(a.isAsyncPlaceholder) &&
                a.asyncFactory === b.asyncFactory &&
                isUndef(b.asyncFactory.error)
            )
        )
    )
}
```

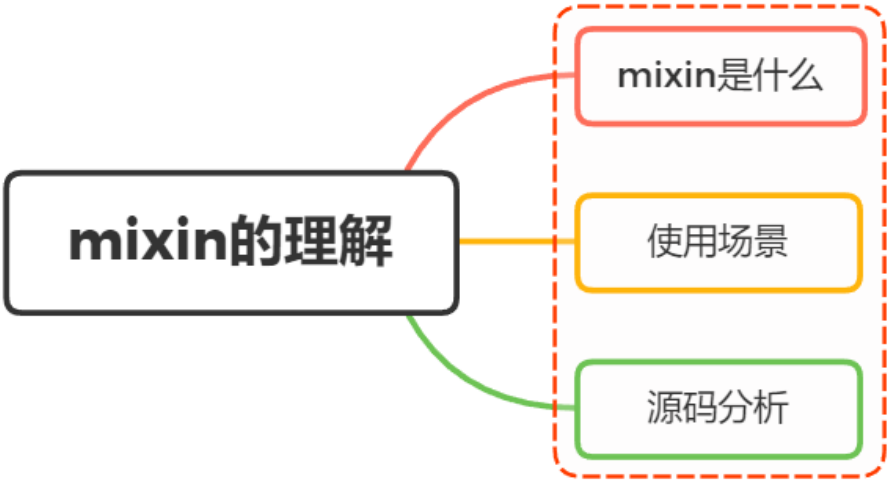
`updateChildren` 方法中会对新旧 `vnode` 进行 `diff`，然后将比对出的结果用来更新真实的 `DOM`

JavaScript | 复制代码

```
function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {
    ...
    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
        if (isUndef(oldStartVnode)) {
            ...
        } else if (isUndef(oldEndVnode)) {
            ...
        } else if (sameVnode(oldStartVnode, newStartVnode)) {
            ...
        } else if (sameVnode(oldEndVnode, newEndVnode)) {
            ...
        } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
            ...
        } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
            ...
        } else {
            if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx)
            idxInOld = isDef(newStartVnode.key)
                ? oldKeyToIdx[newStartVnode.key]
                : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
            if (isUndef(idxInOld)) { // New element
                createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false, newCh, newStartIdx)
            } else {
                vnodeToMove = oldCh[idxInOld]
                if (sameVnode(vnodeToMove, newStartVnode)) {
                    patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue, newCh, newStartIdx)
                    oldCh[idxInOld] = undefined
                    canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm, oldStartVnode.elm)
                } else {
                    // same key but different element. treat as new element
                    createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false, newCh, newStartIdx)
                }
            }
            newStartVnode = newCh[++newStartIdx]
        }
```

```
36        }
37        ...
38    }
```

9. 说说你对vue的mixin的理解，有什么应用场景?



9.1. mixin是什么

`Mixin` 是面向对象程序设计语言中的类，提供了方法的实现。其他类可以访问 `mixin` 类的方法而不必成为其子类

`Mixin` 类通常作为功能模块使用，在需要该功能时“混入”，有利于代码复用又避免了多继承的复杂

9.1.1. Vue中的mixin

先来看一下官方定义

> `mixin` （混入），提供了一种非常灵活的方式，来分发 `Vue` 组件中的可复用功能。

本质其实就是一个 `js` 对象，它可以包含我们组件中任意功能选项，如 `data` 、 `components` 、 `methods` 、 `created` 、 `computed` 等等

我们只要将共用的功能以对象的方式传入 `mixins` 选项中，当组件使用 `mixins` 对象时所有 `mixins` 对象的选项都将被混入该组件本身的选项中来

在 `Vue` 中我们可以**局部混入**跟**全局混入**

9.1.2. 局部混入

定义一个 `mixin` 对象，有组件 `options` 的 `data` 、 `methods` 属性

```javascript
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}
```

组件通过 `mixins` 属性调用 `mixin` 对象

```javascript
Vue.component('componentA',{
  mixins: [myMixin]
})
```

该组件在使用的时候，混合了 `mixin` 里面的方法，在自动执行 `created` 生命钩子，执行 `hello` 方法

9.1.3. 全局混入

通过 `Vue.mixin()` 进行全局的混入

```javascript
Vue.mixin({
  created: function () {
      console.log("全局混入")
   }
})
```

使用全局混入需要特别注意，因为它会影响到每一个组件实例（包括第三方组件）

PS：全局混入常用于插件的编写

9.1.4. 注意事项：

当组件存在与 `mixin` 对象相同的选项的时候，进行递归合并的时候组件的选项会覆盖 `mixin` 的选项

但是如果相同选项为生命周期钩子的时候，会合并成一个数组，先执行 `mixin` 的钩子，再执行组件的钩子

9.2. 使用场景

在日常的开发中，我们经常会遇到在不同的组件中经常会需要用到一些相同或者相似的代码，这些代码的功能相对独立

这时，可以通过 `Vue` 的 `mixin` 功能将相同或者相似的代码提出来

举个例子

定义一个 `modal` 弹窗组件，内部通过 `isShowing` 来控制显示

JavaScript | 复制代码

```
const Modal = {
  template: '#modal',
  data() {
    return {
      isShowing: false
    }
  },
  methods: {
    toggleShow() {
      this.isShowing = !this.isShowing;
    }
  }
}
```

定义一个 `tooltip` 提示框，内部通过 `isShowing` 来控制显示

```javascript
const Tooltip = {
  template: '#tooltip',
  data() {
    return {
      isShowing: false
    }
  },
  methods: {
    toggleShow() {
      this.isShowing = !this.isShowing;
    }
  }
}
```

通过观察上面两个组件，发现两者的逻辑是相同，代码控制显示也是相同的，这时候 `mixin` 就派上用场了

首先抽出共同代码，编写一个 `mixin`

```javascript
const toggle = {
  data() {
    return {
      isShowing: false
    }
  },
  methods: {
    toggleShow() {
      this.isShowing = !this.isShowing;
    }
  }
}
```

两个组件在用上，只需要引入 `mixin`

```javascript
const Modal = {
  template: '#modal',
  mixins: [toggle]
};

const Tooltip = {
  template: '#tooltip',
  mixins: [toggle]
}
```

通过上面小小的例子，让我们知道了 `Mixin` 对于封装一些可复用的功能如此有趣、方便、实用

9.3. 源码分析

首先从 `Vue.mixin` 入手

源码位置：/src/core/global-api/mixin.js

```javascript
export function initMixin (Vue: GlobalAPI) {
  Vue.mixin = function (mixin: Object) {
    this.options = mergeOptions(this.options, mixin)
    return this
  }
}
```

主要是调用 `merOptions` 方法

源码位置：/src/core/util/options.js

JavaScript | 复制代码

```
export function mergeOptions (
  parent: Object,
  child: Object,
  vm?: Component
): Object {

if (child.mixins) { // 判断有没有mixin 也就是mixin里面挂mixin的情况 有的话递归进行合并
    for (let i = 0, l = child.mixins.length; i < l; i++) {
    parent = mergeOptions(parent, child.mixins[i], vm)
    }
}

  const options = {}
  let key
  for (key in parent) {
    mergeField(key) // 先遍历parent的key 调对应的strats[XXX]方法进行合并
  }
  for (key in child) {
    if (!hasOwn(parent, key)) { // 如果parent已经处理过某个key 就不处理了
      mergeField(key) // 处理child中的key 也就parent中没有处理过的key
    }
  }
  function mergeField (key) {
    const strat = strats[key] || defaultStrat
    options[key] = strat(parent[key], child[key], vm, key) // 根据不同类型的options调用strats中不同的方法进行合并
  }
  return options
}
```

从上面的源码，我们得到以下几点：

- 优先递归处理 `mixins`
- 先遍历合并 `parent` 中的 `key`，调用 `mergeField` 方法进行合并，然后保存在变量 `options`
- 再遍历 `child`，合并补上 `parent` 中没有的 `key`，调用 `mergeField` 方法进行合并，保存在变量 `options`
- 通过 `mergeField` 函数进行了合并

下面是关于 `Vue` 的几种类型的合并策略

- 替换型

- 合并型
- 队列型
- 叠加型

9.3.1. 替换型

替换型合并有 `props` 、 `methods` 、 `inject` 、 `computed`

JavaScript | 复制代码

```
strats.props =
strats.methods =
strats.inject =
strats.computed = function (
  parentVal: ?Object,
  childVal: ?Object,
  vm?: Component,
  key: string
): ?Object {
  if (!parentVal) return childVal // 如果parentVal没有值，直接返回childVal
  const ret = Object.create(null) // 创建一个第三方对象 ret
  extend(ret, parentVal) // extend方法实际是把parentVal的属性复制到ret中
  if (childVal) extend(ret, childVal) // 把childVal的属性复制到ret中
  return ret
}
strats.provide = mergeDataOrFn
```

同名的 `props` 、 `methods` 、 `inject` 、 `computed` 会被后来者代替

9.3.2. 合并型

和并型合并有： `data`

JavaScript | 复制代码

```javascript
strats.data = function(parentVal, childVal, vm) {
    return mergeDataOrFn(
        parentVal, childVal, vm
    )
};

function mergeDataOrFn(parentVal, childVal, vm) {
    return function mergedInstanceDataFn() {
        var childData = childVal.call(vm, vm) // 执行data挂的函数得到对象
        var parentData = parentVal.call(vm, vm)
        if (childData) {
            return mergeData(childData, parentData) // 将2个对象进行合并

        } else {
            return parentData // 如果没有childData 直接返回parentData
        }
    }
}

function mergeData(to, from) {
    if (!from) return to
    var key, toVal, fromVal;
    var keys = Object.keys(from);
    for (var i = 0; i < keys.length; i++) {
        key = keys[i];
        toVal = to[key];
        fromVal = from[key];
        // 如果不存在这个属性，就重新设置
        if (!to.hasOwnProperty(key)) {
            set(to, key, fromVal);
        }
        // 存在相同属性，合并对象
        else if (typeof toVal =="object" && typeof fromVal =="object") {
            mergeData(toVal, fromVal);
        }
    }
    return to
}
```

`mergeData` 函数遍历了要合并的 data 的所有属性，然后根据不同情况进行合并：

- 当目标 data 对象不包含当前属性时，调用 `set` 方法进行合并（set方法其实就是一些合并重新赋值的方法）
- 当目标 data 对象包含当前属性并且当前值为纯对象时，递归合并当前对象值，这样做是为了防止对