> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

前面我们提到过 Vue.js 的核心思想之一是组件化，页面可以由一个个组件构建而成，组件是一种抽象的概念，它是对页面的部分布局和逻辑的封装。

为了让组件支持各种丰富的功能，Vue.js 设计了 Props 特性，它允许组件的使用者在外部传递 Props，然后组件内部就可以根据这些 Props 去实现各种各样的功能。

为了让你更直观地理解，我们来举个例子，假设有这样一个 BlogPost 组件，它是这样定义的：

```
<div>

<h1>{{title}}</h1>

<p>author: {{author}}</p>

</div>

<script>

export default {

props: {

title: String,

author: String

    }

  }

</script>
```

然后我们在父组件使用这个 BlogPost 组件的时候，可以给它传递一些 Props 数据：

```
<blog-post title="Vue3 publish" author="yyx"></blog-post>
```

从最终结果来看，BlogPost 组件会渲染传递的 title 和 author 数据。

我们平时写组件，会经常和 Props 打交道，但你知道 Vue.js 内部是如何初始化以及更新 Props 的呢？Vue.js 3.0 在 props 的 API 设计上和 Vue.js 2.x 保持一致，那它们的底层实现层面有没有不一样的地方呢？带着这些疑问，让我们来一起探索 Props 的相关实现原理吧。

Props 的初始化

首先，我们来了解 Props 的初始化过程。之前在介绍 Setup 组件初始化的章节，我们介绍了在执行 setupComponent 函数的时候，会初始化 Props：

```
function setupComponent (instance, isSSR = false) {

const { props, children, shapeFlag } = instance.vnode

const isStateful = shapeFlag & 4

  initProps(instance, props, isStateful, isSSR)

  initSlots(instance, children)

const setupResult = isStateful

    ? setupStatefulComponent(instance, isSSR)

    : undefined

return setupResult

}
```

所以 Props 初始化，就是通过 initProps 方法来完成的，我们来看一下它的实现：

```
function initProps(instance, rawProps, isStateful, isSSR = false) {

const props = {}

const attrs = {}

  def(attrs, InternalObjectKey, 1)

  setFullProps(instance, rawProps, props, attrs)
```

```
if ((process.env.NODE_ENV !== 'production')) {

    validateProps(props, instance.type)

  }

if (isStateful) {

    instance.props = isSSR ? props : shallowReactive(props)

  }

else {

if (!instance.type.props) {

      instance.props = attrs

    }

else {

      instance.props = props

    }

  }

  instance.attrs = attrs

}
```

这里，初始化 Props 主要做了以下几件事情：**设置 props 的值，验证 props 是否合法，把 props 变成响应式，以及添加到实例 instance.props 上。**

注意，这里我们只分析有状态组件的 Props 初始化过程，所以就默认 isStateful 的值是 true。所谓有状态组件，就是你平时通过对象的方式定义的组件。

接下来，我们来看设置 Props 的流程。

设置 Props

我们看一下 setFullProps 的实现：

```
function setFullProps(instance, rawProps, props, attrs) {

const [options, needCastKeys] = normalizePropsOptions(instance.type)

if (rawProps) {

for (const key in rawProps) {

const value = rawProps[key]

if (isReservedProp(key)) {

continue

      }

let camelKey

if (options && hasOwn(options, (camelKey = camelize(key)))) {

        props[camelKey] = value

      }

else if (!isEmitListener(instance.type, key)) {

        attrs[key] = value

      }

    }

  }

if (needCastKeys) {

const rawCurrentProps = toRaw(props)
```

```
for (let i = 0; i < needCastKeys.length; i++) {

const key = needCastKeys[i]

      props[key] = resolvePropValue(options, rawCurrentProps, key,
rawCurrentProps[key])

    }

  }

}
```

我们先注意函数的几个参数的含义：instance 表示组件实例；rawProps 表示原始的 props 值，也就是创建 vnode 过程中传入的 props 数据；props 用于存储解析后的 props 数据；attrs 用于存储解析后的普通属性数据。

设置 Props 的过程也分成几个步骤：标准化 props 的配置，遍历 props 数据求值，以及对需要转换的 props 求值。

接下来，我们来看标准化 props 配置的过程，先看一下 normalizePropsOptions 函数的实现：

```
function normalizePropsOptions(comp) {

if (comp.__props) {

return comp.__props

  }

const raw = comp.props

const normalized = {}

const needCastKeys = []

  let hasExtends = false

if (!shared.isFunction(comp)) {
```

```
const extendProps = (raw) => {

const [props, keys] = normalizePropsOptions(raw)

      shared.extend(normalized, props)

if (keys)

        needCastKeys.push(...keys)

    }

if (comp.extends) {

      hasExtends = true

extendProps(comp.extends)

    }

if (comp.mixins) {

      hasExtends = true

      comp.mixins.forEach(extendProps)

    }

  }

if (!raw && !hasExtends) {

return (comp.__props = shared.EMPTY_ARR)

  }

if (shared.isArray(raw)) {

for (let i = 0; i < raw.length; i++) {
```

```
if (!shared.isString(raw[i])) {
        warn(`props must be strings when using array syntax.`, raw[i])
      }
const normalizedKey = shared.camelize(raw[i])
if (validatePropName(normalizedKey)) {
        normalized[normalizedKey] = shared.EMPTY_OBJ
      }
    }
  }
else if (raw) {
if (!shared.isObject(raw)) {
      warn(`invalid props options`, raw)
    }
for (const key in raw) {
const normalizedKey = shared.camelize(key)
if (validatePropName(normalizedKey)) {
const opt = raw[key]
const prop = (normalized[normalizedKey] =
          shared.isArray(opt) || shared.isFunction(opt) ? { type: opt } : opt)
```

```
if (prop) {

const booleanIndex = getTypeIndex(Boolean, prop.type)

const stringIndex = getTypeIndex(String, prop.type)

          prop[0 ] = booleanIndex > -1

          prop[1 ] =

            stringIndex < 0 || booleanIndex < stringIndex

if (booleanIndex > -1 || shared.hasOwn(prop, 'default')) {

            needCastKeys.push(normalizedKey)

          }

        }

      }

    }

  }

const normalizedEntry = [normalized, needCastKeys]

  comp.__props = normalizedEntry

return normalizedEntry

}
```

normalizePropsOptions 主要目的是标准化 props 的配置，这里需要注意，你要区分 props 的配置和 props 的数据。所谓 props 的配置，就是你在定义组件时编写的 props 配置，它用来描述一个组件的 props 是什么样的；而 props 的数据，是父组件在调用子组件的时候，给子组件传递的数据。

所以这个函数首先会处理 mixins 和 extends 这两个特殊的属性，因为它们的作用都是扩展组件的定义，所以需要对它们定义中的 props 递归执行 normalizePropsOptions。

接着，函数会处理数组形式的 props 定义，例如：

```
export default {

  props: ['name', 'nick-name']

}
```

如果 props 被定义成数组形式，那么数组的每个元素必须是一个字符串，然后把字符串都变成驼峰形式作为 key，并为 normalized 的 key 对应的每一个值创建一个空对象。针对上述示例，最终标准化的 props 的定义是这样的：

```
export default {

  props: {

    name: {},

    nickName: {}

  }

}
```

如果 props 定义是一个对象形式，接着就是标准化它的每一个 prop 的定义，把数组或者函数形式的 prop 标准化成对象形式，例如：

```
export default {

  title: String,

  author: [String, Boolean]

}
```

注意，上述代码中的 String 和 Boolean 都是内置的构造器函数。经过标准化的 props 的定义：

```
export default {
```

```
  props: {
    title: {
      type: String
    },
    author: {
      type: [String, Boolean]
    }
  }
}
```

接下来，就是判断一些 prop 是否需要转换，其中，含有布尔类型的 prop 和有默认值的 prop 需要转换，这些 prop 的 key 保存在 needCastKeys 中。注意，这里会给 prop 添加两个特殊的 key，prop[0] 和 prop[1] 赋值，它们的作用后续我们会说。

最后，返回标准化结果 normalizedEntry，它包含标准化后的 props 定义 normalized，以及需要转换的 props key needCastKeys，并且用 comp.__props 缓存这个标准化结果，如果对同一个组件重复执行 normalizePropsOptions，直接返回这个标准化结果即可。

标准化 props 配置的目的无非就是支持用户各种的 props 配置写法，标准化统一的对象格式为了后续统一处理。

我们回到 setFullProps 函数，接下来分析遍历 props 数据求值的流程。

```
function setFullProps(instance, rawProps, props, attrs) {
if (rawProps) {
for (const key in rawProps) {
const value = rawProps[key]
if (isReservedProp(key)) {
continue
```

```
      }

let camelKey

if (options && hasOwn(options, (camelKey = camelize(key)))) {

        props[camelKey] = value

      }

else if (!isEmitListener(instance.type, key)) {

        attrs[key] = value

      }

    }

  }

}
```

该过程主要就是遍历 rawProps，拿到每一个 key。由于我们在标准化 props 配置过程中已经把 props 定义的 key 转成了驼峰形式，所以也需要把 rawProps 的 key 转成驼峰形式，然后对比看 prop 是否在配置中定义。

如果 rawProps 中的 prop 在配置中定义了，那么把它的值赋值到 props 对象中，如果不是，那么判断这个 key 是否为非事件派发相关，如果是那么则把它的值赋值到 attrs 对象中。另外，在遍历的过程中，遇到 key、ref 这种 key，则直接跳过。

接下来我们来看 setFullProps 的最后一个流程：对需要转换的 props 求值。

```
function setFullProps(instance, rawProps, props, attrs) {

if (needCastKeys) {

const rawCurrentProps = toRaw(props)

for (let i = 0; i < needCastKeys.length; i++) {
```

```
const key = needCastKeys[i]

      props[key] = resolvePropValue(options, rawCurrentProps, key,
rawCurrentProps[key])

    }

  }

}
```

在 normalizePropsOptions 的时候，我们拿到了需要转换的 props 的 key，接下来就是遍历 needCastKeys，依次执行 resolvePropValue 方法来求值。我们来看一下它的实现：

```
function resolvePropValue(options, props, key, value) {

const opt = options[key]

if (opt != null) {

const hasDefault = hasOwn(opt, 'default')

if (hasDefault && value === undefined) {

const defaultValue = opt.default

      value =

        opt.type !== Function && isFunction(defaultValue)

          ? defaultValue()

          : defaultValue

    }

if (opt[0 ]) {

if (!hasOwn(props, key) && !hasDefault) {
```

```
        value = false

      }

else if (opt[1 ] &&

        (value === '' || value === hyphenate(key))) {

        value = true

      }

    }

  }

return value

}
```

resolvePropValue 主要就是针对两种情况的转换，第一种是默认值的情况，即我们在 prop 配置中定义了默认值，并且父组件没有传递数据的情况，这里 prop 对应的值就取默认值。

第二种是布尔类型的值，前面我们在 normalizePropsOptions 的时候已经给 prop 的定义添加了两个特殊的 key，所以 opt[0] 为 true 表示这是一个含有 Boolean 类型的 prop，然后判断是否有传对应的值，如果不是且没有默认值的话，就直接转成 false，举个例子：

```
export default {

  props: {

    author: Boolean

  }

}
```

如果父组件调用子组件的时候没有给 author 这个 prop 传值，那么它转换后的值就是 false。

接着看 opt[1] 为 true，并且 props 传值是空字符串或者是 key 字符串的情况，命中这个逻辑表示这是一个含有 Boolean 和 String 类型的 prop，且 Boolean 在 String 前面，例如：

```
export default {

  props: {

    author: [Boolean, String]

  }

}
```

这种时候如果传递的 prop 值是空字符串，或者是 author 字符串，则 prop 的值会被转换成 true。

至此，props 的转换求值结束，整个 setFullProps 函数逻辑也结束了，回顾它的整个流程，我们可以发现**它的主要目的就是对 props 求值，然后把求得的值赋值给 props 对象和 attrs 对象中。**

验证 Props

接下来我们再回到 initProps 函数，分析第二个流程：验证 props 是否合法。

```
function initProps(instance, rawProps, isStateful, isSSR = false) {

const props = {}

if ((process.env.NODE_ENV !== 'production')) {

    validateProps(props, instance.type)

  }

}
```

验证过程是在非生产环境下执行的，我们来看一下 validateProps 的实现：

```
function validateProps(props, comp) {

const rawValues = toRaw(props)

const options = normalizePropsOptions(comp)[0]

for (const key in options) {
```

```
    let opt = options[key]

if (opt == null)

continue

validateProp(key, rawValues[key], opt, !hasOwn(rawValues, key))

  }

}

function validateProp(name, value, prop, isAbsent) {

const { type, required, validator } = prop

if (required && isAbsent) {

    warn('Missing required prop: "' + name + '"')

return

  }

if (value == null && !prop.required) {

return

  }

if (type != null && type !== true) {

    let isValid = false

const types = isArray(type) ? type : [type]

const expectedTypes = []

for (let i = 0; i < types.length && !isValid; i++) {
```

```
const { valid, expectedType } = assertType(value, types[i])

      expectedTypes.push(expectedType || '')

      isValid = valid

    }

if (!isValid) {

      warn(getInvalidTypeMessage(name, value, expectedTypes))

return

    }

  }

if (validator && !validator(value)) {

    warn('Invalid prop: custom validator check failed for prop "' + name + '".')

  }

}
```

顾名思义，validateProps 就是用来检测前面求得的 props 值是否合法，它就是对标准化后的 Props 配置对象进行遍历，拿到每一个配置 opt，然后执行 validateProp 验证。

对于单个 Prop 的配置，我们除了配置它的类型 type，还可以配置 required 表明它的必要性，以及 validator 自定义校验器，举个例子：

```
export default {

  props: {

    value: {

      type: Number,
```

```
      required: true,
      validator(val) {
return val >= 0
      }
    }
  }
}
```

因此 validateProp 首先验证 required 的情况，一旦 prop 配置了 required 为 true，那么必须给它传值，否则会报警告。

接着是验证 prop 值的类型，由于 prop 定义的 type 可以是多个类型的数组，那么只要 prop 的值匹配其中一种类型，就是合法的，否则会报警告。

最后是验证如果配了自定义校验器 validator，那么 prop 的值必须满足自定义校验器的规则，否则会报警告。

相信这些警告你在平时的开发工作中或多或少遇到过，了解了 prop 的验证原理，今后再遇到这些警告，你就能知其然并知其所以然了。

响应式处理

我们再回到 initProps 方法，来看最后一个流程：把 props 变成响应式，添加到实例 instance.props 上。

```
function initProps(instance, rawProps, isStateful, isSSR = false) {
if (isStateful) {
    instance.props = isSSR ? props : shallowReactive(props)
  }
else {
if (!instance.type.props) {
```

```
      instance.props = attrs

    }

else {

      instance.props = props

    }

  }

  instance.attrs = attrs

}
```

在前两个流程，我们通过 setFullProps 求值赋值给 props 变量，并对 props 做了检测，接下来，就是把 props 变成响应式，并且赋值到组件的实例上。

至此，Props 的初始化就完成了，相信你可能会有一些疑问，为什么 instance.props 要变成响应式，以及为什么用 shallowReactive API 呢？在接下来的 Props 更新流程的分析中，我来解答这两个问题。

Props 的更新

所谓 Props 的更新主要是指 Props 数据的更新，它最直接的反应是会触发组件的重新渲染，我们可以通过一个简单的示例分析这个过程。例如我们有这样一个子组件 HelloWorld，它是这样定义的：

```
<template>

<div>

<p>{{ msg }}</p>

</div>

</template>

<script>

export default {

props: {
```

```
msg: String

    }

  }

</script>
```

这里，HelloWorld 组件接受一个 msg prop，然后在模板中渲染这个 msg。

然后我们在 App 父组件中引入这个子组件，它的定义如下：

```
<template>

<hello-world :msg="msg"></hello-world>

<button @click="toggleMsg">Toggle Msg</button>

</template>

<script>

import HelloWorld from './components/HelloWorld'

export default {

components: { HelloWorld },

    data() {

return {

msg: 'Hello world'

      }

    },
```

```
methods: {

      toggleMsg() {

this.msg = this.msg === 'Hello world' ? 'Hello Vue' : 'Hello world'

      }

    }

  }

</script>
```

我们给 HelloWorld 子组件传递的 prop 值是 App 组件中定义的 msg 变量，它的初始值是 Hello world，在子组件的模板中会显示出来。

接着当我们点击按钮修改 msg 的值的时候，就会触发父组件的重新渲染，因为我们在模板中引用了这个 msg 变量。我们会发现这时 HelloWorld 子组件显示的字符串变成了 Hello Vue，那么子组件是如何被触发重新渲染的呢?

在组件更新的章节我们说过，组件的重新渲染会触发 patch 过程，然后遍历子节点递归 patch，那么遇到组件节点，会执行 updateComponent 方法：

```
const updateComponent = (n1, n2, parentComponent, optimized) => {

const instance = (n2.component = n1.component)

if (shouldUpdateComponent(n1, n2, parentComponent, optimized)) {

    instance.next = n2

    invalidateJob(instance.update)

    instance.update()

  }

else {

    n2.component = n1.component
```

```
    n2.el = n1.el

  }

}
```

在这个过程中，会执行 shouldUpdateComponent 方法判断是否需要更新子组件，内部会对比 props，由于我们的 prop 数据 msg 由 Hello world 变成了 Hello Vue，值不一样所以 shouldUpdateComponent 会返回 true，这样就把新的子组件 vnode 赋值给 instance.next，然后执行 instance.update 触发子组件的重新渲染。

所以这就是触发子组件重新渲染的原因，但是子组件重新渲染了，子组件实例的 instance.props 的数据需要更新才行，不然还是渲染之前的数据，那么是如何更新 instance.props 的呢，我们接着往下看。

执行 instance.update 函数，实际上是执行 componentEffect 组件副作用渲染函数：

```
const setupRenderEffect = (instance, initialVNode, container, anchor,
parentSuspense, isSVG, optimized) => {

  instance.update = effect(function componentEffect() {

if (!instance.isMounted) {

    }

else {

      let { next, vnode } = instance

if (next) {

        updateComponentPreRender(instance, next, optimized)

      }

else {

        next = vnode

      }
```

```
const nextTree = renderComponentRoot(instance)

const prevTree = instance.subTree

      instance.subTree = nextTree

      patch(prevTree, nextTree,

        hostParentNode(prevTree.el),

        getNextHostNode(prevTree),

        instance,

        parentSuspense,

        isSVG)

      next.el = nextTree.el

    }

  }, prodEffectOptions)

}
```

在更新组件的时候，会判断是否有 instance.next, 它代表新的组件 vnode，根据前面的逻辑 next 不为空，所以会执行 updateComponentPreRender 更新组件 vnode 节点信息，我们来看一下它的实现：

```
const updateComponentPreRender = (instance, nextVNode, optimized) => {

  nextVNode.component = instance

const prevProps = instance.vnode.props

  instance.vnode = nextVNode

  instance.next = null
```

```
updateProps(instance, nextVNode.props, prevProps, optimized)

updateSlots(instance, nextVNode.children)

}
```

其中，会执行 updateProps 更新 props 数据，我们来看它的实现：

```
function updateProps(instance, rawProps, rawPrevProps, optimized) {

const { props, attrs, vnode: { patchFlag } } = instance

const rawCurrentProps = toRaw(props)

const [options] = normalizePropsOptions(instance.type)

if ((optimized || patchFlag > 0) && !(patchFlag & 16 )) {

if (patchFlag & 8 ) {

const propsToUpdate = instance.vnode.dynamicProps

for (let i = 0; i < propsToUpdate.length; i++) {

const key = propsToUpdate[i]

const value = rawProps[key]

if (options) {

if (hasOwn(attrs, key)) {

            attrs[key] = value

          }

else {
```

```
const camelizedKey = camelize(key)

            props[camelizedKey] = resolvePropValue(options, rawCurrentProps,
camelizedKey, value)

          }

        }

else {

          attrs[key] = value

        }

      }

    }

  }

else {

    setFullProps(instance, rawProps, props, attrs)

let kebabKey

for (const key in rawCurrentProps) {

if (!rawProps ||

        (!hasOwn(rawProps, key) &&

          ((kebabKey = hyphenate(key)) === key || !hasOwn(rawProps, kebabKey))))
{

if (options) {

if (rawPrevProps &&
```

```
            (rawPrevProps[key] !== undefined ||

              rawPrevProps[kebabKey] !== undefined)) {

            props[key] = resolvePropValue(options, rawProps || EMPTY_OBJ, key,
undefined)

          }

        }

else {

          delete props[key]

        }

      }

    }

  }

if ((process.env.NODE_ENV !== 'production') && rawProps) {

    validateProps(props, instance.type)

  }

}
```

updateProps 主要的目标就是把父组件渲染时求得的 props 新值，更新到子组件实例的 instance.props 中。

在编译阶段，我们除了捕获一些动态 vnode，也捕获了动态的 props，所以我们可以只去比对动态的 props 数据更新。

当然，如果不满足优化的条件，我们也可以通过 setFullProps 去全量比对更新 props，并且，由于新的 props 可能是动态的，因此会把那些不在新 props 中但存在于旧 props 中的值设置为 undefined。

好了，至此我们搞明白了子组件实例的 props 值是如何更新的，那么我们现在来思考一下前面的一个问题，为什么 instance.props 需要变成响应式呢？其实这是一种需求，因为我们也希望在子组件中可以监听 props 值的变化做一些事情，举个例子：

```
import { ref, h, defineComponent, watchEffect } from 'vue'

const count = ref(0)

let dummy

const Parent = {

  render: () => h(Child, { count: count.value })

}

const Child = defineComponent({

  props: { count: Number },

  setup(props) {

    watchEffect(() => {

      dummy = props.count

    })

return () => h('div', props.count)

  }

})

count.value++
```

这里，我们定义了父组件 Parent 和子组件 Child，子组件 Child 中定义了 prop count，除了在渲染模板中引用了 count，我们在 setup 函数中通过了 watchEffect 注册了一个回调函数，内部依赖了 props.count，当修改 count.value 的时候，我们希望这个回调函数也能执行，所以这个 prop 的值需要是响应式的，由于 setup 函数的第一个参数是 props 变量，其实就是组件实例 instance.props，所以

也就是要求 instance.props 是响应式的。

我们再来看为什么用 shallowReactive API 呢？shallow 的字面意思是浅的，从实现上来说，就是不会递归执行 reactive，只劫持最外一层对象。

shallowReactive 和普通的 reactive 函数的主要区别是处理器函数不同，我们来回顾 getter 的处理器函数：

```
function createGetter(isReadonly = false, shallow = false) {

return function get(target, key, receiver) {

if (key === "__v_isReactive" ) {

return !isReadonly;

    }

else if (key === "__v_isReadonly" ) {

return isReadonly;

    }

else if (key === "__v_raw"  &&

      receiver ===

      (isReadonly

        ? target["__v_readonly" ]

        : target["__v_reactive" ])) {

return target;

    }

const targetIsArray = isArray(target);

if (targetIsArray && hasOwn(arrayInstrumentations, key)) {
```

```
return Reflect.get(arrayInstrumentations, key, receiver);

    }

const res = Reflect.get(target, key, receiver);

if (isSymbol(key)

      ? builtInSymbols.has(key)

      : key === `__proto__` || key === `__v_isRef`) {

return res;

    }

if (!isReadonly) {

      track(target, "get" , key);

    }

if (shallow) {

return res;

    }

if (isRef(res)) {

return targetIsArray ? res : res.value;

    }

if (isObject(res)) {

return isReadonly ? readonly(res) : reactive(res);
```

```
    }

return res;

  };

}
```

shallowReactive 创建的 getter 函数，shallow 变量为 true，那么就不会执行后续的递归 reactive 逻辑。也就是说，shallowReactive 只把对象 target 的最外一层属性的访问和修改处理成响应式。

之所以可以这么做，是因为 props 在更新的过程中，只会修改最外层属性，所以用 shallowReactive 就足够了。

总结

好的，到这里我们这一节的学习也要结束啦，通过这节课的学习，你应该要了解 Props 是如何被初始化的，如何被校验的，你需要区分开 Props 配置和 Props 传值这两个概念；你还应该了解 Props 是如何更新的以及实例上的 props 为什么要定义成响应式的。

最后，给你留一道思考题目，我们把前面的示例稍加修改，HelloWorld 子组件如下：

```
<template>

<div>

<p>{{ msg }}</p>

<p>{{ info.name }}</p>

<p>{{ info.age }}</p>

</div>

</template>

<script>

export default {

props: {

msg: String,
```

```
info: Object

    }

  }

</script>
```

我们添加了 info prop，然后在模板中渲染了 info 的子属性数据，然后我们再修改一下父组件：

```
<template>

<hello-world :msg="msg" :info="info"></hello-world>

<button @click="addAge">Add age</button>

<button @click="toggleMsg">Toggle Msg</button>

</template>

<script>

import HelloWorld from './components/HelloWorld'

export default {

components: { HelloWorld },

    data() {

return {

info: {

name: 'Tom',

age: 18
```

```
        },
msg: 'Hello world'
      }
    },
methods: {
      addAge() {
this.info.age++
      },
      toggleMsg() {
this.msg = this.msg === 'Hello world' ? 'Hello Vue' : 'Hello world'
      }
    }
  }
</script>
```

我们在 data 中添加了 info 变量，然后当我们点击 Add age 按钮去修改 this.info.age 的时候，触发了子组件 props 的变化了吗？子组件为什么会重新渲染呢？欢迎你在留言区与我分享。

> 本节课的相关代码在源代码中的位置如下：
> packages/runtime-core/src/componentProps.ts
> packages/reactivity/src/reactive.ts
> packages/reactivity/src/baseHandlers.ts