vue3.0

vue2和vue3的区别

1vue2 和**vue3**双向数据绑定原理不同

2.0的响应式基于Object.defineProperty中的set和get方法实现兼容主流浏览器和ie9以上的ie浏览器，能够监听数据对象的变化，但是监听不到对象属性的增删、数组元素和长度的变化，同时会在vue初始化的时候把所有的Observer（观察者）都建立好，才能观察到数据对象属性的变化。

3.0的响应式采用了ES2015的Proxy来代替Object.defineProperty，可以做到监听对象属性的增删和数组元素和长度的修改，同时还实现了惰性的监听（不会在初始化的时候创建所有的Observer，而是会在用到的时候才去监听）但是，虽然主流的浏览器都支持Proxy，ie系列却还是不兼容，所以针对ie11，vue3.0决定做单独的适配，暴露出来的api一样，但是底层实现还是Object.defineProperty

2vue3默认进行懒观察（**lazy observation**）

vue2.0中数据一开始就创建了观察者，数据很大的时候，就会出现问题，vue3中进行了优化只有用于渲染初始化可见部分的数据，才会创建观察者，效率更高。

3Vue3 对**ts**的支持更好

4项目目录结构发生了变化

2.x中 移除了配置文件目录，config 和 build 文件夹
3.x中 相关的配置需要在根目录中手动创建vue.config.js

5更精准的变更通知

2.x 版本中，你使用 Vue.set 来给对象新增一个属性时，这个对象的所有 watcher 都会重新运行；
3.x 版本中，只有依赖那个属性的 watcher 才会重新运行

6.vue2组件属性方式变成**CompositionAPI**（组合式）函数式风格。

vue/cli 5x

在vue3开发中 脚手架vue/cli版本必须最新 cnpm install -g @vue/cli

创建项目同 之前方式 使用 vue create 项目名

注意 在vue/cli 5x中对我们的文件名 有约束必须大驼峰

注意 在项目根目录找到一个vue.config.js 文件，没有就在根目录创建一个，写上下面标注的代码，保存，在重新编译。项目就可以正常运行了。

```
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
  transpileDependencies: true,
  lintOnSave: false//关闭eslint的校验
})
```

Vue3 新特性函数 ---- setup

1.setup函数是处于 生命周期函数 beforeCreate 和 Created 两个钩子函数之间的函数 也就说在 setup函数中是无法 使用 data 和 methods 中的数据和方法的
2、setup函数是 Composition API（组合API）的入口
3、在setup函数中定义的变量和方法最后都是需要 return 出去的 不然无法再模板中使用

注意：**vue3**和**vue2**不同的地方就是不必写 **data**、**methods**、等代码块了所有的东西都可以在 **setup** 中返回

setup 可以返回两种值：

返回对象

1、返回对象，对象中的属性、方法都可以直接在模板中使用--稍后马上学到

返回渲染函数

2、返回渲染函数，可以自定义渲染内容（其实也就是 **render** 函数）注意：渲染函数优先级还高，他会替换掉模板中的其他内容（用的比较少记住就好）

VUE3中h()函数和createVNode()函数的使用

h()函数和createVNode()函数都是创建dom节点，他们的作用是一样的，但是在VUE3中createVNode()函数的功能比h()函数要多且做了性能优化，渲染节点的速度也更快。

语法：

h(标签,{属性},内容)

```
<template>
  <div>

  </div>
</template>

<script>
// 1.要使用先引用
import {h} from "vue"
export default {
  setup() {
    return () => h('div',{class:"demodiv"}, ['Hello,Vue3'])
  }
}
</script>

<style>

</style>
```

h(标签,{属性},[可以继续嵌套h()])

```
<template>
  <div>

  </div>
</template>

<script>
// 1.要使用先引用
import {h} from "vue"
export default {
```

```
  setup() {
    return () => h('div',{class:"demodiv"}, [
      h("h1",{class:"demoh"},["我是标题标签"])
    ])
  }
}
</script>

<style>

</style>
```

createVNode(标签,{属性},[内容])

```
<template>
  <div>

  </div>
</template>

<script>
// 1.要使用先引用
import {createVNode} from "vue"
export default {
  setup() {
    return () => createVNode('div',{class:"demodiv"}, ['Hello,Vue3'])
  }
}
</script>

<style>

</style>
```

createVNode(标签,{属性},[可以继续嵌套createVNode()])

```
<template>
  <div>

  </div>
</template>

<script>
// 1.要使用先引用
import {createVNode} from "vue"
```

```
export default {
    setup() {
     return () => createVNode('div',{class:"demodiv"}, [
       createVNode("h1",{class:"demoh"},["我是标题标签"])
     ])
   }
}
</script>

<style>

</style>
```

setup参数

setup有两个形参

参数1第一个参数props

props是一个对象，包含父组件传递给子组件的所有数据。

父组件

```
<template>
  <div>
    我是父组件
    <Zc title="我是正向传值的数据"/>
  </div>
</template>

<script>
import { defineComponent } from 'vue'
import Zc from "../components/ZiCom.vue"
export default defineComponent({
  components:{
    Zc
  },
  setup() {
    return {

    }
  },
})
</script>

<style>

</style>
```

子组件

```
<template>
  <div>
      我是子组件--{{title}}
  </div>
</template>

<script>
import { defineComponent } from 'vue'

export default defineComponent({
    props:["title"],//子组件设置props
    setup(props) {//setup第一个参数是props
        console.log(props)//如果想在setup中读取props数据必须设置第一个形参

        return {

        }
    },
})
</script>


<style>

</style>
```

参数2第一个参数context

context是setup的第二个参数里面包含了以下三个属性 attrs emit slots

attrs

attrs 获取当前标签上面的所有属性的对象

注意 attrs是接收props没有声明的属性

注意 如果子组件用props已经声明 就不能使用attrs 否则会返回undefind

父组件

```
<template>
  <div>
    我是父组件
    <Zc title="我是正向传值的数据"/>
  </div>
```

```
</template>

<script>
import { defineComponent } from 'vue'
import Zc from "../components/ZiCom.vue"
export default defineComponent({
  components:{
    Zc
  },
  setup() {
    return {

    }
  },
})
</script>

<style>

</style>
```

子组件

```
<template>
  <div>
      我是子组件--{{title}}
  </div>
</template>

<script>
import { defineComponent } from 'vue'

export default defineComponent({
    // props:["title"],//如果使用attrs接收就不能声明props否则接收不到
    setup(props,context) {//setup第2个参数是context
        console.log(context.attrs)

        return {

        }
    },
})
</script>


<style>

</style>
```

emit

emit事件分发 用于子传父 如果子组件的数据想传递给父组件 就是用emit（逆向传值）

子组件

```
<template>
  <div>
      我是子组件
      <button @click="zifun">点我抛出事件</button>
  </div>
</template>

<script>
import { defineComponent } from 'vue'

export default defineComponent({
    setup(props,context) {
    //     子组件通过context的emit方法进行数据的抛出
       let zifun=()=>{
           context.emit("zipao",'我是子组件的数据')
       }

        return {//不要忘了return
            zifun
        }
    },
})
</script>


<style>

</style>
```

父组件

```
<template>
  <div>
    我是父组件
    <!-- 接收子组件的数据 注意不加()不加()-->
    <Zc @zipao="fufun"/>
  </div>
</template>

<script>
import { defineComponent } from 'vue'
import Zc from "../components/ZiCom.vue"
export default defineComponent({
  components:{
```

```
    Zc
  },
  setup() {
    let fufun=(val)=>{
      console.log("子组件的数据",val)
    }
    return {
      fufun
    }
  },
})
</script>

<style>

</style>
```

slots

插槽 带有dom的属性 （用来接收父组件传递的html内容）

子组件

```
<template>
  <div>
      我是子组件
      <!-- 定义槽口 -->
      <slot name="com"></slot>

  </div>
</template>

<script>
import { defineComponent } from 'vue'

export default defineComponent({
    setup(props,context) {

        console.log(context.slots)

        return {
        }
    },
})
</script>


<style>
```

```
</style>
```

父组件

```
<template>
  <div>
    我是父组件

    <Zc>
      <!--
        两个语法不能同时使用
        <template v-slot:com>
           <h1>我是插入的内容</h1>
        </template>
      -->
      <template #com>
        <h1>我是插入的内容</h1>
      </template>

    </Zc>
  </div>
</template>

<script>
import { defineComponent } from 'vue'
import Zc from "../components/ZiCom.vue"
export default defineComponent({
  components:{
    Zc
  },
  setup() {

    return {

    }
  },
})
</script>

<style>

</style>
```

新语法

Setup新方式3.2：

- 不需要return任何东西，所有定义的变量和方法会自动导出，template模板可以直接使用
- import引入的组件也会自动导出，模板里可以直接引用。
- 引入的组件命名需要首字母大写。
- 确切的说是：所有的变量方法和**import**导入都可以在模板中直接使用。

```
<script setup>

</script>
```

ref

ref:的主要作用是定义数据和进行dom操作
ref 它接收一个参数作为值,然后返回一个响应式对象
要是想改变 **ref** 的值 必须改变它里面有个属性 **value**

ref创建基本类型数据

```
<template>
  <div>
      我是一个组件-{{text}}

  </div>
</template>

// <script>
// export default {
//     // 旧方式
//    setup(){
//      let text=ref("你好我是数据")
//      return {
//        text
//      }
//    }
// }
// </script>

<script setup>
  import {ref} from "vue"
  let text=ref("你好我是数据")
```

```
</script>

<style>

</style>
```

要是想改变 **ref** 的值 必须改变它里面有个属性 **value**

```
<template>
  <div>
    我是一个组件-{{ text }}
    <button @click="fun()">点我修改</button>
  </div>
</template>

// <script>
// export default {
//   // 旧方式
//   setup() {
//     let text = ref("你好我是数据");
//     let fun = () => {
//       text.value = "我变了";
//     };

//     return {
//       text,
//       fun,
//     };
//   },
// };
// </script>

<script setup>
import { ref } from "vue";
let text = ref("你好我是数据");
let fun = () => {
  text.value = "我变了";
};
</script>

<style>
</style>
```

获取dom的ref

```
<template>
  <div>
    <!-- 2.绑定 -->
      <h1 ref="demoh">我是一个dom</h1>
```

```
      <button @click="fun()">点我获取</button>
  </div>
</template>


<script setup>
import { ref } from "vue";
// 1.创建
let demoh = ref(null);
let fun = () => {
  // 3使用 这个.value是ref获取数据的时候必要的
  console.log(demoh.value)
};
</script>

<style>
</style>
```

reactive：定义对象、数组

```
<template>
  <div>
      <h1>{{obj.name}}</h1>
      <button @click="fun()">修改</button>
  </div>
</template>


<script setup>
import { reactive } from "vue";

let obj = reactive({
  name:"xixi",
  age:18
});
let fun = () => {
    obj.name="haha"
};
</script>

<style>
</style>
```

ref和reactive区别

使用方式：ref修改数据需要使用这样：数据.value=xxx的形式，而reactive只需要数据.key=值 这样来使用

计算属性computed

计算属性computed是用对数据进行逻辑处理操作，实现数据包装。计算属性通常依赖于当前vue对象中的普通属性

当依赖的依赖的普通属性发生变化的时候，计算属性也会发生变化。

基本使用

```
<template>
  <div>
      <input type="text" v-model="data">
      <h1>默认的：{{data}}</h1>
      <h1>变大写的：{{newdata}}</h1>
  </div>
</template>

<script setup>

  import {ref,computed} from "vue"
  let data=ref("abcdefg");

  // 计算属性
  let newdata=computed(()=>{
    return data.value.toUpperCase()
  })









</script>

<style>

</style>
```

watch监听

监听单个数据

```
<template>
  <div>
    <input type="text" v-model="data" />
    <h1>{{ data }}</h1>
  </div>
</template>

<script setup>
import { ref, watch } from "vue";
let data = ref("abcdefg");
// 第一个参数是你要监听的数据
// 第二个参数是数据改变之后触发的函数
watch(data, (newVal, oldVal) => {
  console.log("newVal", newVal);
  console.log("oldVal", oldVal);
});
</script>

<style>
</style>
```

监听多个数据

将需要监听的数据添加到数组

```
<template>
  <div>
    <input type="text" v-model="data" />
    <h1>{{ data }}</h1>
    <input type="text" v-model="datab" />
    <h1>{{ datab }}</h1>
  </div>
</template>

<script setup>
import { ref, watch } from "vue";
let data = ref("第1个数据");
let datab = ref("第2个数据");
// 第一个参数是你要监听的数据多个数据使用数组来表示
// 第二个参数是数据改变之后触发的函数
watch([data,datab], (newVal, oldVal) => {
  console.log("newVal", newVal);
  console.log("oldVal", oldVal);
});
```

```
</script>

<style>
</style>
```

watch监听对象

在vue2中watch默认是无法监听到对象中的属性 必须手动开启deep深度监听

在vue3中强制开启**deep**深度监听（但是注意如果使用默认的**watch**语法当 监听值为响应式对象时，**oldValue**值将出现异常，此时与**newValue**相同）

观察下面得代码会发现 newval和oldval的值是一样的

```
<template>
  <div>
    <input type="text" v-model="obj.name" />
    <h1>{{ obj.name }}</h1>
  </div>
</template>

<script setup>
// 监听对象
// 在watch监听对象的时候vue3中会开启强制deep深度监听


import { reactive, watch } from "vue";
  let obj=reactive({
    name:"xixi",
    age:18
  })
// 第一个参数是你要监听的数据多个数据使用数组来表示
// 第二个参数是数据改变之后触发的函数
watch(obj, (newVal, oldVal) => {
  console.log("newVal", newVal);
  console.log("oldVal", oldVal);
});
</script>

<style>
</style>
```

监听对象属性正确写法

在监听对象的时候我们可以把第一个参数设置成函数并且返回你要监听的数据

```
<template>
  <div>
```

```
    <input type="text" v-model="obj.name" />
    <h1>{{ obj.name }}</h1>
  </div>
</template>

<script setup>


import { reactive, watch } from "vue";
  let obj=reactive({
    name:"xixi",
    age:18
  })
// 在监听对象的时候我们可以把第一个参数设置成函数并且返回你要监听的数据
watch(()=>{return obj.name}, (newVal, oldVal) => {
  console.log("newVal", newVal);
  console.log("oldVal", oldVal);
});
</script>

<style>
</style>
```

监听多个对象属性

监听多个对象属性的时候使用数组来进行设置

```
<template>
  <div>
    <input type="text" v-model="obj.name" />
    <input type="text" v-model="obj.age" />
    <h1>{{ obj.name }}</h1>
    <h1>{{ obj.age }}</h1>
  </div>
</template>

<script setup>


import { reactive, watch } from "vue";
  let obj=reactive({
    name:"xixi",
    age:18
  })
// 监听多个对象属性的时候使用数组来进行设置
watch([()=>{return obj.name},()=>{return obj.age}], (newVal, oldVal) =>
{
  console.log("newVal", newVal);
  console.log("oldVal", oldVal);
```

```
});
</script>

<style>
</style>
```

手动开启deep深度监听

当监听响应式对象的属性为复杂数据类型时，需要开启**deep**深度监听

当监听**proxy**对象的属性为复杂数据类型时，需要开启**deep**深度监听

当监听**proxy**对象的属性为复杂数据类型时，需要开启**deep**深度监听

比如下面得数据 我们监听obj下的user vue3也没有办法了

```
let obj = reactive({
  user: {
    name: "xixi",
    age: 18,
  },
});
```

```
<template>
  <div>
    <input type="text" v-model="obj.user.name" />
    <input type="text" v-model="obj.user.age" />
    <h1>{{ obj.user.name }}</h1>
    <h1>{{ obj.user.age }}</h1>
  </div>
</template>



<script setup>
import { reactive, watch } from "vue";
let obj = reactive({
  user: {
    name: "xixi",
    age: 18,
  },
});
// 监听多个对象属性的时候使用数组来进行设置
watch(
  () => {return obj.user;},
  (newVal, oldVal) => {
    console.log("newVal", newVal);
    console.log("oldVal", oldVal);
```

```
  },
  // 第三个参数就是设置deep
  {
    deep:true
  }
);
</script>

<style>
</style>
```

但是newval和oldval还是有问题 但是数据修改函数可以触发

生命周期

```
<template>
  <div>

  </div>
</template>

<script setup>
import {
  onBeforeMount,
  onMounted,
  onBeforeUpdate,
  onUpdated,
  onBeforeUnmount,
  onUnmounted,
} from "vue";
// 其他的生命周期
onBeforeMount(() => {
  console.log("App ===> 相当于 vue2.x 中 beforeMount");
});
onMounted(() => {
  console.log("App ===> 相当于 vue2.x 中 mounted");
});

// 注意，onBeforeUpdate 和 onUpdated 里面不要修改值
onBeforeUpdate(() => {
  console.log("App ===> 相当于 vue2.x 中 beforeUpdate");
});

onUpdated(() => {
  console.log("App ===> 相当于 vue2.x 中 updated");
});

onBeforeUnmount(() => {
```

```
  console.log("App ===> 相当于 vue2.x 中 beforeDestroy");
});

onUnmounted(() => {
  console.log("App ===> 相当于 vue2.x 中 destroyed");
});
</script>

<style>
</style>
```

过滤器

vue3取消了**vue2**中的过滤器，但是变相一下，在vue3中可以把数据当成函数的实参传递给一个函数 这样一来就可以使用一个函数来模拟原来的vue2过滤器

比如我们要过滤下 一个数据

```
<template>
  <div>
    <h1>正常展示:{{data}}</h1>
    <!-- 创建一个函数模拟过滤器 -->
    <h1>只显示年:{{setdata(data)}}</h1>

  </div>
</template>

<script setup>
import {ref} from "vue"

let data=ref("2022-10-1")

let setdata=(val)=>{
  return val.substr(0,4)
}
</script>

<style>

</style>
```

插槽--slot

就是给组件设置了一个开口（组件默认是一个完整地独立地个体 外部内容不能插入 如果你想插入数据可以直接使用props）槽口/插槽是吧一个dom元素传入到组件中 传递了dom就可以提高组件的复用性

默认插槽

定义slot接受

```
<template>
  <div>
    <h1>我是子组件</h1>
    <slot></slot>
  </div>
</template>


<script setup>

</script>

<style>
</style>
```

插入的时候需要使用#default来设置

```
<template>
  <div>
    <h1>我是父组件</h1>
    <ZiView>

      <template #default>
        <h1>我是内容</h1>
        <h1>我是内容</h1>
        <h1>我是内容</h1>
      </template>
    </ZiView>
  </div>
</template>

<script setup>
  import ZiView from "../components/ZiView.vue"

</script>

<style>
```

```
</style>
```

具名插槽

使用name起名

```
<template>
  <div>
    <h1>我是子组件</h1>
    <slot name="main"></slot>
  </div>
</template>


<script setup>

</script>

<style>
</style>
```

使用#slot的名字 代替原来2.0的slot属性

```
<template>
  <div>
    <h1>我是父组件</h1>
    <ZiView>

      <!-- 使用#slot的名字  代替原来2.0的slot属性 -->
      <template #main>
        <h1>我是内容</h1>
        <h1>我是内容</h1>
        <h1>我是内容</h1>
      </template>
    </ZiView>
  </div>
</template>

<script setup>
  import ZiView from "../components/ZiView.vue"

</script>

<style>
</style>
```

作用域插槽

关于插槽就是无非就是在子组件中挖个坑，坑里面放什么东西由父组件决定。而作用域插槽就是父组件可以直接拿到子组件的值

定义子组件

```
<template>
  <div>
    <h1>我是子组件</h1>
     <!-- 向父组件 传递数据 以data接受 -->
        <slot :data="item"></slot>
  </div>
</template>


<script setup>
  import {ref} from "vue"
  let item=ref("我是数据")
</script>

<style>
</style>
```

父组件

```
<template>
  <div>
    <h1>我是父组件</h1>
    <ZiView>
      <!-- 使用#slot的名字  代替原来2.0的slot属性 -->
<template #default="{ data }"> {{ data}} </template>
    </ZiView>
  </div>
</template>

<script setup>
  import ZiView from "../components/ZiView.vue"

</script>

<style>
</style>
```

####

组件

全局组件---component

1.在main.js中引用所需要的组件

2.使用component调用

3.在任意组件可以直接使用

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'
// 1.引用
import SunView from "./components/SunView.vue";
// 2.使用component调用
createApp(App).component('SunView',SunView).use(store).use(router).mount
('#app')
```

局部组件

在vue3.x中，引入的组件是需要在components中声明的（和vue2x一样写法），但是在vue3.2中，你可以直接使用。

旧方式

创建子组件

```
<template>
  <div>
      我是一个子组件
  </div>
</template>

<script>
import {defineComponent} from "vue"
export default defineComponent({




})
</script>

<style>
```

```
</style>
```

在父组件中使用子组件

```
<template>
  <div>
    <h1>我是父组件</h1>
    <!-- 3使用 -->
    <Zc/>
  </div>
</template>

 <script>
import { defineComponent } from 'vue'
// 1.引用
import Zc from "../components/ZiCom.vue"

export default defineComponent({
  // 2.调用
  components:{
    Zc
  },


  setup() {

      return {

      }
  },
})
</script>

<style>

</style>
```

新语法

新方式3.2:

- 不需要return任何东西，所有定义的变量和方法会自动导出，template模板可以直接使用
- import引入的组件也会自动导出，模板里可以直接引用。
- 引入的组件命名需要首字母大写。
- 确切的说是：所有的变量方法和**import**导入都可以在模板中直接使用。

```
<template>
```

```
  <div>
    <!--2使用-->
    <ZiView/>
  </div>
</template>

<script setup>
// 1.要使用先引用
import ZiView from "@/components/ZiView.vue"

</script>

<style>

</style>
```

旧方式

```
<template>
  <div>
    <ZiView/>
  </div>
</template>

<script>
// 1.要使用先引用
import ZiView from "@/components/ZiView.vue"
export default {
  // 2.调用
  components:{
    ZiView
  }

}
</script>

<style>

</style>
```

组件传值

正向传值--defineProps

旧语法

子组件接收正向传值的语法同vue2x

```
<script>
import { defineComponent } from 'vue'

export default defineComponent({

    // 旧语法和vue2x一样使用props
    props:["title","num"],

    setup() {

    },
})
</script>
```

新语法

子组件

```
<template>
  <div>
      我是一个子组件--{{text}}
  </div>
</template>

<script setup>
import {defineProps} from "vue"

// 定义接受参数
defineProps({
  text:{
    type:String
  }
})


</script>

<style>

</style>
```

父组件

```
<template>
```

```
  <div>
    <ZiView  text="我是正向传值"/>
  </div>
</template>

<script setup>
// 1.要使用先引用
import ZiView from "@/components/ZiView.vue"

</script>

<style>

</style>
```

逆向传值

逆向传值--defineEmits

在vue2x中逆向传值使用this.$emit()自定义事件来完成

子组件--使用emits抛出自定义事件

```
<template>
  <div>
      我是一个子组件
      <button @click="fun()">点我逆向传值</button>
  </div>
</template>

<script setup>
import {defineEmits} from "vue"
const emits = defineEmits(['xiaoming'])

let fun=()=>{
  emits('xiaoming',"我是子组件数据")
}

</script>

<style>

</style>
```

父组件--接受自定义事件

```
<template>
  <div>
    <ZiView @xiaoming="fufun"/>
  </div>
</template>

<script setup>
// 1.要使用先引用
import ZiView from "@/components/ZiView.vue"
let fufun=(val)=>{
  console.log("父组件",val)
}
</script>

<style>

</style>
```

逆向传值--ref---defineExpose

子组件---千万不要忘了 ref方式传值 需要在子组件使用**defineExpose**（子组件暴露自己的属性）

```
<template>
  <div>
    <h1>我是子组件--{{ obj.name }}</h1>
  </div>
</template>


<script setup>
import { reactive ,defineExpose} from "vue";

let obj = reactive({
  name: "xixi",
  age: 18,
});
// 千万不要忘了 ref方式传值 需要在子组件使用defineExpose（暴露自己的属性） 暴露自己的属性
defineExpose({ obj })
</script>

<style>
</style>
```

父组件

```
<template>
  <div>
```

```
    <!-- 2.绑定到组件之上 -->
    <ZiView ref="com"/>

    <button @click="fun()">点我得到子组件的内容</button>
  </div>
</template>

<script setup>

import ZiView from "@/components/ZiView.vue"
import {ref} from "vue"
// 1.创建ref
let com=ref(null)

let fun=()=>{
  console.log(com.value.obj)
}

</script>

<style>

</style>
```

跨组件provide/inject

provide和inject是Vue中提供的一对API，该API可以实现父组件向子组件传递数据，无论层级有多深，都可以通过这对API实现

祖先组件--使用provide传递

```
<template>
  <div>
   <h1>爷组件</h1>
    <ZiView ref="com"/>


  </div>
</template>

<script setup>

import ZiView from "@/components/ZiView.vue"
import {ref,provide} from "vue"
let yeText=ref("我是爷爷组件的数据")

// 向子组件提供数据
provide('list', yeText.value)
```

```
</script>

<style>

</style>
```

后代组件--inject使用数据

```
<template>
  <div>
      我是孙组件--{{yeText}}
  </div>
</template>

<script setup>

import { inject } from 'vue'
// 接受父组件提供的数据
const yeText = inject('list')

</script>

<style>

</style>
```

跨组件传值--vuex 4x

vuex4 是 vue3的兼容版本，提供了和vuex3 的相同API。因此我们可以在 vue3 中复用之前已存在的 vuex 代码。

state

1.定义状态

```
import { createStore } from 'vuex'

export default createStore({
  state: {
    text:"我是vuex中的数据"//定义状态
  },
  getters: {
  },
  mutations: {
  },
  actions: {
  },
  modules: {
  }
})
```

2.组件中使用

在组件的模板中直接使用，与之前的api保持一致 没有this

```
<template>
  <div>

   <h1>{{$store.state.text}}</h1>
  </div>
</template>

<script setup>

</script>
```

3.在js中使用 通过 **useStore** 把**store** 引入组件内，然后操作 **store** 。

```
<template>
  <div>
    <!-- 在组件的模板中直接使用，与之前的api保持一致 没有this-->
    <h1>{{ $store.state.text }}</h1>
  </div>
</template>

<script setup>
import { onMounted } from "vue";
// 1.引用useStore
import { useStore } from "vuex";
// 2.得到store对象
const store = useStore();
onMounted(() => {
  // 3.使用
  console.log(store.state.text);
});
</script>
```

module

1.模块拆分--同之前

```
export default {
    state: {
        text: "我是vuex中的数据"//定义状态
    },
    getters: {
    },
    mutations: {
    },
    actions: {
    },
}
```

2.在文件中配置

```
import { createStore } from 'vuex'
// 1.引用
import demom from "./demom"
export default createStore({

  modules: {
    demom//调用
  }
})
```

3.使用

$store.state.模块名.数据名

```
<template>
  <div>

    <h1>{{ $store.state.demom.text }}</h1>
  </div>
</template>

<script setup>
import { onMounted } from "vue";
// 1.引用useStore
import { useStore } from "vuex";
// 2.得到store对象
const store = useStore();
onMounted(() => {
  // 3.使用
  console.log(store.state.demom.text);
});
</script>
```

mutations

同之前

actions

同之前

getters

同之前

toRef与toRefs

引子

下面代码就是修改一个对象的值

```
<template>
  <div>
    <h2>传统方式:{{ user.name }}--{{user.age}}</h2>
    <button @click="hello">点我修改</button>
  </div>
</template>

<script setup>
    import { reactive } from "vue";
    let user = reactive({
      name: "xixi",
      age:18
    });

    let hello=()=>{
       user.name = "小白";
       user.age = 19;
      }
</script>
```

但是页面中出现太多的**user. xxx** 这样就显得很难受

在这个时候我们想 可以修改下代码变成如下代码

```
<template>
  <div>
```

```
    <h2>传统方式:{{ name }}--{{age}}</h2>
    <button @click="hello">点我修改</button>
  </div>
</template>

<script setup>
    import { reactive } from "vue";
    let user = reactive({
      name: "xixi",
      age:18
    });
  // 添加如下代码
   // let name=user.name
   // let age=user.age

    //技术好点的同学可能会想到用解构
    let {name,age}=user

    let hello=()=>{
       name = "小白";
       age = 19;
      }
</script>
```

但是发现数据不改变了 因为我们创建的name 和 age的两个变量不是vue的响应式数据 所以没有办法改变

所以我们把**name**和**age**变成响应式

```
<template>
  <div>
    <h2>传统方式:{{ name }}--{{age}}</h2>
    <button @click="hello">点我修改</button>
  </div>
</template>

<script setup>
    import { reactive,ref } from "vue";
    let user = reactive({
      name: "xixi",
      age:18
    });
  // 添加如下代码
    let name=ref(user.name)
    let age=ref(user.age)

    let hello=()=>{
      // 不要忘了ref创建数据需要.value
       name.value = "小白";
```

```
        age.value = 19;
      }
</script>
```

发现数据正常改变了 但是上面这种方式太麻烦了 我们只是为了简化语法 有点丢了西瓜捡了芝麻的感觉

toRef

toRef作用：将对象某一个属性，作为引用返回为响应式ref数据。

所以这个使用toRef就派上了作用

```
<template>
  <div>
    <h2>传统方式:{{ name }}--{{age}}</h2>
    <button @click="hello">点我修改</button>
  </div>
</template>

<script setup>
    import { reactive,toRef } from "vue";
    let user = reactive({
      name: "xixi",
      age:18
    });
  // 添加如下代码
  //参数一为一个响应对象，
  //参数二为参数一这个对象中的某个属性。
    let name=toRef(user,"name")
    let age=toRef(user,"age")

    let hello=()=>{
       //注意要使用.value取值
       name.value = "小白";
       age.value = 19;
      }
</script>
```

toRefs

toRefs 返回对象中所有属性都响应式，相比之下比toRef写法跟简单，但是肯定会牺牲耗性能(因为会把数据中所有内容都进行操作)

```
<template>
  <div>
```

```
    <h2>传统方式:{{ name }}--{{age}}</h2>
    <button @click="hello">点我修改</button>
  </div>
</template>

<script setup>
    import { reactive,toRefs } from "vue";
    let user = reactive({
      name: "xixi",
      age:18
    });
  // 添加如下代码
  //参数一为一个响应对象,
  //参数二为参数一这个对象中的某个属性。
    let {name,age}=toRefs(user)


    let hello=()=>{
       //注意要使用.value取值
       name.value = "小白";
       age.value = 19;
      }
</script>
```

区别

toRefs一次性全部将对象的属性变成ref对象，而toRef单个的

readonly与shallowReadonly

readonly，让一个响应式数据只读（深层次只读）。

```
<template>
  <div>
    <h2>传统方式:{{ name }}--{{age}}--{{sex}}</h2>
    <button @click="hello">点我修改</button>
  </div>
</template>

<script setup>
    import { reactive,toRef,readonly } from "vue";
    let user = reactive({
      name: "xixi",
      age:18,
      love:{
        sex:"女"
      }
```

```
    });

    // 把响应式数据编程只读属性
   user=readonly(user)
  // 添加如下代码
  //参数一为一个响应对象，
  //参数二为参数一这个对象中的某个属性。
    let name=toRef(user,"name")
    let age=toRef(user,"age")
    let sex=toRef(user.love,"sex")

    let hello=()=>{
       //注意要使用.value取值
       name.value = "小白";
       age.value = 19;
       sex.value="女人"
    }
</script>
```

shallowReadonly，让一个响应式数据只读（浅层次只读）。

把上面的代码修改下 会发现多层级数据不能变成只读

```
<template>
  <div>
    <h2>传统方式:{{ name }}--{{age}}--{{sex}}</h2>
    <button @click="hello">点我修改</button>
  </div>
</template>

<script setup>
    import { reactive,toRef,shallowReadonly } from "vue";
    let user = reactive({
      name: "xixi",
      age:18,
      love:{
        sex:"女"
      }
    });

    // 把响应式数据编程只读属性
   user=shallowReadonly(user)
  // 添加如下代码
  //参数一为一个响应对象，
  //参数二为参数一这个对象中的某个属性。
    let name=toRef(user,"name")
    let age=toRef(user,"age")
    let sex=toRef(user.love,"sex")
```

```
    let hello=()=>{
        //注意要使用.value取值
        name.value = "小白";
        age.value = 19;
        sex.value="女人"
    }
</script>
```

shallowReactive和shallowRef对数据进行非深度监听

默认情况下， reactive 都深度监听。 深度监听存在的问题： 如果数据量比较大，非常消耗性能。 有些时候我们并不需要对数据进行深度监听。 这个时候就没有必要使用reactive

shallowReactive

Vue3为我们提供了，shallowReactive进行非深度监听。 shallowReactive只会包装第一层的数据 默认情况它只能够监听数据的第一层。 如果想更改多层的数据， 你必须先更改第一层的数据。 然后在去更改其他层的数据。 这样视图上的数据才会发生变化。

```
<template>
  <div>
    <h2>传统方式:{{ user.name }}--{{user.age}}--{{user.love}}</h2>
    <button @click="hello">点我修改</button>
  </div>
</template>

<script setup>
    import { shallowReactive } from "vue";
    let user = shallowReactive({
      name: "xixi",
      age:18,
      love:{
        sex:"女"
      }
    });



    let hello=()=>{
      //先修改其他层级在修改深层级就会成功
      // user.name="haha"
      // user.love.sex="女人"

      // 直接修改深层级别就不行了
```

```
        user.love.sex="女人"
    }
</script>
```

shallowRef

如果是通过shallowRef创建的数据。 那么Vue监听的是.value 变化。 xxx.value={}

该方法就可以包装对象和基础类型的数据了，不过包装基础属性的数据和普通的`ref`没有区别，这里就只说说包装对象时的用法

简单的来说**shallowRef**只会在**value**本身发生变化时触发监听器

```
<template>
    <div class="about">
        {{ test }}
        <button @click="change">change</button>
    </div>
</template>
<script setup>
import {shallowRef} from "vue"
const test = shallowRef({
    a: "a",
});

function change() {
  //和shallowReactive不同，这里修改了对象的第一层属性不会触发视图更新，shallowRef的第一层数据实际指的是包装的整个数据
    //test.value.a = "new";

    test.value = { b: "b" };
    console.log(test.value);
}
</script>
```

自定义HOOK

HOOK是什么?

本质上HOOK是一个函数 只是把原来**setup**中我们写的组合式**API**进行了封装（就是把原来需要重复的逻辑封装起来方便复用）

类似于**vue2**的**mixins** 方便的把多个组件需要重复使用的逻辑封装起来

例子:

我们需要得到当前鼠标的点击坐标 传统方式

```
<template>
  <div>

    <h1>x:{{num.x}}---y:{{num.y}}</h1>
  </div>
</template>

<script setup>
import { reactive,onMounted } from "vue";

let num=reactive({
  x:0,
  y:0
})

let clickPint=()=>{
  window.addEventListener("click",(event)=>{
    num.x=event.pageX
    num.y=event.pageY
  })
}

// 初始化调用函数
onMounted(()=>{
  clickPint()
})
</script>
```

但是多个组件都想用怎么办?

封装自定义HOOK

1.新建一个hooks的文件夹用来容纳 把原来的逻辑复制过来 但是不要忘了**return**数据

```
import { reactive, onMounted } from "vue";
export default function () {


    let num = reactive({
        x: 0,
        y: 0
    })

    let clickPint = () => {
        window.addEventListener("click", (event) => {
```

```
            num.x = event.pageX
            num.y = event.pageY
        })
    }

    // 初始化调用函数
    onMounted(() => {
        clickPint()
    })

    // 不要忘了return 数据
       // 不要忘了return 数据
          // 不要忘了return 数据
    return num
}
```

2.在想使用的组件中引用调用

```
<template>
  <div>
  <!-- 3.使用 -->
    <h1>x:{{useclick.x}}---y:{{useclick.y}}</h1>
  </div>
</template>

<script setup>
// 1。引用
  import useClick from "../hooks/index.js"
  // 2.调用
  let useclick=useClick()
</script>
```

Vue Router4路由

Vue3支持最新版本由于Vue 3 引入了createApp API，该API更改了将插件添加到Vue实例的方式。 因此，以前版本的Vue Router将与Vue3不兼容。Vue Router 4 引入了createRouter API，该API创建了一个可以在Vue3中安装 router 实例。

路由配置

同传统路由

路由模式

History模式

History选项在Vue Router 4中，这些模式已被抽象到模块中，可以将其导入并分配给新的 `history` 选项。

```
import { createRouter, createWebHistory } from "vue-router";
export default createRouter({
  history: createWebHistory(),//定义history模式url不带#
  routes: [],
});
```

hash模式

```
import { createRouter, createWebHashHistory } from "vue-router";
export default createRouter({
  history: createWebHashHistory(),//定义hash模式
  routes: [],
});
```

路由导航

声明式

同传统路由---router-link

编程式--useRouter()

使用useRouter() 来替代 this.$router

```
<template>
  <div>

    <button @click="fun">dianwo</button>
  </div>
</template>

<script setup>
  import { useRouter, useRoute } from "vue-router";
  // useRouter赋值给一个变量
      let router = useRouter();
      let fun=()=>{
       router.push("/")
      }
</script>
```

动态路由匹配--useRoute()

params

1.路由规则配置接受参数

2.发送数据--同之前

3.接受数据

```
<template>
  <div class="home">


  </div>
</template>

<script lang="ts" setup>
import {onMounted } from 'vue';

 import {useRoute} from "vue-router";
 let route=useRoute()
  onMounted(()=>{
    // 接收params
    console.log(route.params.xiaoming)
  })

</script>
```

query

1.发送

2.接受

```
<template>
  <div class="home">


  </div>
</template>

<script lang="ts" setup>
import {onMounted } from 'vue';

 import {useRoute} from "vue-router";
 let route=useRoute()
```

```
  onMounted(()=>{
    // 接收query
    console.log(route.query.xiaoming)
  })

</script>
```

路由守卫

同之前

vue3新组件

Fragment

在vue2中组件必须有一个根组件包裹多行内容 （但是往往这个根组件仅仅是包裹的作用 但是会在页面生成多余的冗余标签）

在vue3中 组件是可以不需要写根标签的 因为我们如果写了多行标签vue3会创建一个虚拟的Fragment根标签自动帮助我们包裹

优点：较少了不必要的标签渲染 减少过多的内存损耗

Teleport

Teleport 是什么

Teleport 可以把我们组件中的dom内容 移动到当前项目的任意dom节点中

观察下面得代码 会发现 当我们点击显示隐藏的时候 demob这个div会根据内容的显示隐藏变大变小

```
<template>
  <div>
    <div class="demoa">
      <h1>我是demoa</h1>
    </div>
    <div class="demob">
      <h1>我是demob</h1>
```

```
      <button @click="bool=!bool">点我显示隐藏</button>
      <div v-if="bool">
        <p>我是占位的</p>
        <p>我是占位的</p>
        <p>我是占位的</p>
        <p>我是占位的</p>
        <p>我是占位的</p>
        <p>我是占位的</p>
        <p>我是占位的</p>
      </div>
    </div>
  </div>
</template>

<script setup>
import {ref} from "vue"
let bool=ref(true)
</script>
<style scoped>

.demoa{
  background-color: pink;
}

.demob{
  background-color: goldenrod;
}
</style>
```

使用Teleport 其中to属性就是传送到那个dom上

```
<template>
  <div>
    <div class="demoa">
      <h1>我是demoa</h1>
    </div>
    <div class="demob">
      <h1>我是demob</h1>
      <button @click="bool = !bool">点我显示隐藏</button>

      <teleport to="body">
        <div v-if="bool">
          <p>我是占位的</p>
          <p>我是占位的</p>
          <p>我是占位的</p>
```

```
            <p>我是占位的</p>
            <p>我是占位的</p>
            <p>我是占位的</p>
            <p>我是占位的</p>
          </div>
        </teleport>
      </div>
    </div>
  </template>

  <script setup>
  import { ref } from "vue";
  let bool = ref(true);
  </script>
  <style scoped>
  .demoa {
    background-color: pink;
  }

  .demob {
    background-color: goldenrod;
  }
  </style>
```

使用场景： 比如在某个组件中我们需要有一个弹框 弹出在页面中 那么就可以使用teleport 把这个组件内部的dom挂载到组件之外的地方 方便我们设置层级

typescript

TypeScript简介

TypeScript是一种由微软开发的自由和开源的编程语言。它是JavaScript的一个超集，而且本质上**TypeScript**扩展了**JavaScript**的语法解决JavaScript的“痛点”：弱类型和没有命名空间，导致很难模块化。

1. TypeScript是JavaScript的超集。
2. 它对JS进行了扩展，向JS中引入了类型的概念，并添加了许多新的特性。

3. TS代码需要通过编译器编译为JS，然后再交由JS解析器执行。
4. TS完全兼容JS，换言之，任何的JS代码都可以直接当成JS使用。
5. 相较于JS而言，TS拥有了静态类型，更加严格的语法，更强大的功能；TS可以在代码执行前就完成代码的检查，减小了运行时异常的出现的几率；TS代码可以编译为任意版本的JS代码，可有效解决不同JS运行环境的兼容问题；同样的功能，TS的代码量要大于JS，但由于TS的代码结构更加清晰，变量类型更加明确，在后期代码的维护中TS却远远胜于JS。

为什么要用TypeScript

开源

简单
TypeScript 是 JavaScript 的超集，这意味着他支持所有的 JavaScript 语法。

兼容性好
TScript 是 JS的强类型版本。然后在编译期去掉类型和特有语法，生成纯粹的 JavaScript 代码。由于最终在浏览器中运行的仍然是 JS，所以 TScript 并不依赖于浏览器的支持，也并不会带来兼容性问题。任何现有的JS程序可以不加改变的在TScript下工作。

TypeScript 开发环境搭建

1. 下载Node.js
    - 64位：https://nodejs.org/dist/v14.15.1/node-v14.15.1-x64.msi
    - 32位：https://nodejs.org/dist/v14.15.1/node-v14.15.1-x86.msi
2. 安装Node.js
3. 使用npm全局安装typescript
    - 进入命令行
    - 输入：npm i -g typescript
    - tsc空格-v命令用来测试是否安装成功
4. 创建一个ts文件
5. 使用tsc对ts文件进行编译
    - 进入命令行
    - 进入ts文件所在目录
    - 执行命令：tsc xxx.ts

通过配置编译

每次写完ts文件都要输入一次命令是不是很麻烦呢，能不能保存文件时就自动编译运行ts文件呢

- cd到项目下
- 使用 **tsc -init** 会生成tsconfig.json 文件 （tsconfig.json文件与TypeScript编译器(tsc)的配置相对应 是 *TypeScript* 使用 *tsconfig.json* 文件作为其配置文件 用来 指定待编译文件和定义编译选项。）
- 直接使用tsc命令即可编译

为什么要使用这个文件

通常我们可以使用 `tsc` 命令来编译少量 TypeScript 文件

但如果实际开发的项目，很少是只有单个文件，当我们需要编译整个项目时，就可以使用 tsconfig.json 文件，将需要使用到的配置都写进 tsconfig.json 文件，这样就不用每次编译都手动输入配置，另外也方便团队协作开发。

常见配置

```
{
  "compilerOptions": {
    "target": "ES5",                  // 目标语言的版本
    "module": "commonjs",             // 指定生成代码的模板标准
    "noImplicitAny": true,            // 不允许隐式的 any 类型
    "removeComments": true,           // 删除注释
    "preserveConstEnums": true,   // 保留 const 和 enum 声明
    "sourceMap": true,                 // 生成目标文件的sourceMap文件(简单说，Source map就是一个信息文件，里面储存着位置信息。也就是说，转换后的代码的每一个位置，所对应的转换前的位置。有了它，出错的时候，除错工具将直接显示原始代码，而不是转换后的代码。这无疑给开发者带来了很大方便)
    "outDir":"./out/"              //编译输出的文件夹
  },
  "files": [    // 指定待编译文件(files 配置项值是一个数组，用来指定了待编译文件，即入口文件。入口文件依赖其他文件时，不需要将被依赖文件也指定到 files 中，因为编译器会自动将所有的依赖文件归纳为编译对象，即 index.ts 依赖 user.ts 时，不需要在 files 中指定 user.ts ， user.ts 会自动纳入待编译文件。)
    "./src/index.ts"
  ]
}
```

自动编译

vscode 自动编译

需要 监视tsconfig.json文件:

1.点击终端运行任务

2. 选择typescript:

3. 选择监视tsconfig.json文件

命令自动编译

- 编译文件时，使用 -w 指令后，TS编译器会自动监视文件的变化，并在文件发生变化时对文件进行重新编译。
- 示例:

```
tsc xxx.ts -w
```

- 监控全部文件

```
tsc -w
```

基本类型

变量

注意：let变量不能重复声明
注意：const它拥有与 let相同的作用域规则，但是不能对它们重新赋值。
注意:除了下划线 _ 和美元 $ 符号外，不能包含其他特殊字符，包括空格

类型声明

- 类型声明是TS非常重要的一个特点
- 通过类型声明可以指定TS中变量（参数、形参）的类型
- 指定类型后，当为变量赋值时，TS编译器会自动检查值是否符合类型声明，符合则赋值，否则报错
- 简而言之，类型声明给变量设置了类型，使得变量只能存储某种类型的值
- 语法:
    - ```
      let 变量: 类型;

      let 变量: 类型 = 值;

      function fn(参数: 类型, 参数: 类型): 返回值类型{
          ...
      }
      ```
      ```

基础/基元类型

类型	例子	描述
number	1, -33, 2.5	任意数字
string	'hi', "hi", `hi`	任意字符串
boolean	true、false	布尔值true或false

数组类型

```
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];//泛型语法
```

对象类型

{}用来指定对象中可以包含哪些属性

语法：{属性: 属性值, 属性: 属性值...}

```
let obj:{name:string,age:number}={
    name:"xixi",
    age:18
}
```

注意：如果在使用对象的时候 必须给每个属性都要传入对应的值 否则会报错

```
// 没有给age传值就会报错
let obj:{name:string,age:number}={
    name:"xixi"
}
```

可选属性

在属性名后面加?，表示该属性是可选的

```
// age为可选属性 所以不传之也不会报错
let obj:{name:string,age?:number}={
    name:"xixi"
}
```

null与unfined类型

null是 定以不存在的

unfined 是未初始化的值

enum 枚举

TS中新增类型 使用枚举类型可以为一组数值赋予友好的名字。枚举表示的是一个命名元素的集合值

就是给一组数据起一个友好的名字

数字枚举类型和字符串枚举类型;

```
enum user{xiaoming,xiaohong,xiaobai}
console.log(user.xiaohong)//默认情况下数据的值从0开始

// 设置值
enum user{xiaoming,xiaohong=99,xiaobai}
console.log(user.xiaohong)
console.log(user.xiaobai)

// 字符串枚举设置值
enum user{xiaoming,xiaohong="小红",xiaobai="小白"}
console.log(user.xiaohong)
console.log(user.xiaobai)
```

tuple元组

元组，TS新增类型，元组类型用来表示已知元素数量和类型的数组，各元素的类型不必相同，对应位置的类型需要相同（赋值的顺序不能变）

- ```
  let x: [string, number];
  x = ["hello", 10];
  ```

any

在一些情况下，如果我们无法确定变量的类型时（或者无需确认类型时），我们可以将其指定为 any 类型。 TS中对于被标记为 any 类型的变量，是没有进行类型检查而直接通过编译阶段的检查。 在我们的系统中还是应当尽量避免使用 **any** 类型，以尽可能的保证系统健壮性。

- ```
  let d: any = 4;
  d = 'hello';
  d = true;
  ```

void

一种类型，告诉你函数和方法在调用时不返回任何内容。

```
function fun(text:string,num:number=18):void{
    console.log(text+"---"+num)
}
fun("xixi")
```

自动类型判断

TS拥有自动的类型判断机制

当对变量的声明和赋值是同时进行的，TS编译器会自动判断变量的类型

所以如果你的变量的声明和赋值时同时进行的，可以省略掉类型声明

类型别名

类型别名用来给一个类型起个新名字，使用 type 创建类型别名，类型别名常用于联合类型。

在实际应用中，有些类型名字比较长或者难以记忆，重新命名是一个较好的解决方案。

```
// 创建一个String别名
type xiaoming=String;
// 使用别名
let textCon:xiaoming="xixi";

console.log(textCon);//xixi
```

联合类型-Union Type

联合类型表示的值可能是多种不同类型当中的某一个。 联合类型放宽了类型的取值的范围，也就是说值的范围不再限于某个单一的数据类型。同时，它也不是无限制地放宽取值的范围，如果那样的话，完全可以使用 **any** 代替。

```
// 给多个类型创建一个名字
type newType=String|Number;
// 可以在赋值的时候赋值字符串与数字
let demoText:newType="你好我可以创建字符串与number"
```

接口（Interface）

接口是定义对象类型的另外一种方式，在程序设计里面，接口起到一种限制和规范的作用

使用*interface*关键字定义 接口一般首字母大写 有的编程语言中会建议接口的名称加上 *I* 前缀

- 示例（检查对象类型）：
  - ```
    interface IUser{
        name:String,
        showname():void
    }
    ```
- 示例（实现）
- 接口使用：使用：号接口名来进行使用
  注意：定义的变量比接口少了一些属性是不允许的，多一些属性也是不允许的。赋值的时候，变量的形状必须和接口的形状保持一致。
  - ```
    interface IUser{
        name:String,
        showname():void
    }
    let user:IUser={
        name:"xixi",
        showname(){
            console.log(`名字是${this.name}`)
        }
    }
    console.log(user.name)
    user.showname()
    ```

泛型

在开发的时候我们需要考虑我们的代码有非常好的重用性（我们开发的模块 不仅仅能支持当前的数据类型 同时也要支持未来不确定的数据类型）

那么在这个时候就可以使用泛型来创建可以重用的组件功能 这样一来我们的组件就可以根据后续的需要 来接收任意的类型

什么是泛型?

就是把不能明确的类型 变成一个参数（就是一个类型变量--用来存储类型的变量）

泛型通常用字母T来表示（不一定必须是**T** 可以是任何的单词）

```
// 下面这个 函数定义了一个T的泛型 那么这个函数就可以接受传递进来的任意类型
function fun<T>(name:T):T{
    return name
}

console.log(fun<number>(123))//传递number类型
console.log(fun<string>("你好"))//传递string类型
```

面向对象

面向对象是程序中一个非常重要的思想，它被很多同学理解成了一个比较难，比较深奥的问题，其实不然。面向对象很简单，简而言之就是程序之中所有的操作都需要通过对象来完成。

- 举例来说：
    - 操作浏览器要使用window对象
    - 操作网页要使用document对象
    - 操作控制台要使用console对象

一切操作都要通过对象，也就是所谓的面向对象，那么对象到底是什么呢？这就要先说到程序是什么，计算机程序的本质就是对现实事物的抽象，抽象的反义词是具体，比如：照片是对一个具体的人的抽象，汽车模型是对具体汽车的抽象等等。程序也是对事物的抽象，在程序中我们可以表示一个人、一条狗、一把枪、一颗子弹等等所有的事物。一个事物到了程序中就变成了一个对象。

在程序中所有的对象都被分成了两个部分数据和功能，以人为例，人的姓名、性别、年龄、身高、体重等属于数据，人可以说话、走路、吃饭、睡觉这些属于人的功能。数据在对象中被成为属性，而功能就被称为方法。所以简而言之，在程序中一切皆是对象。

类（class）

要想面向对象，操作对象，首先便要拥有对象，那么下一个问题就是如何创建对象。要创建对象，必须要先定义类，所谓的类可以理解为对象的模型，程序中可以根据类创建指定类型的对象，举例来说：可以通过Person类来创建人的对象，通过Dog类创建狗的对象，通过Car类来创建汽车的对象，不同的类可以用来创建不同的对象。

- 定义类：

```
class 类名 {
    属性名: 类型;
    //constructor 方法是类的构造函数，是一个默认方法，通过 new 命令创建对象实例时，自动调用该方法。一个类必须有 constructor 方法，如果没有显式定义，一个默认的 consructor 方法会被默认添加。所以即使你没有添加构造函数，也是会有一个默认的构造函数的。

    constructor(参数: 类型){
        this.属性名 = 参数;
    }

    方法名(){
        ....
    }
}
```

- 示例：

```
class Person{
    name: string;
    age: number;

    constructor(name: string, age: number){
        this.name = name;
        this.age = age;
    }

    sayHello(){
        console.log(`大家好，我是${this.name}`);
    }
}
```

- 使用类：

```
//通过new关键字可以方便的生产一个类的实例对象，这个生产对象的过程叫实例化
const p = new Person('孙悟空', 18);
p.sayHello();
```

面向对象的特点

- 封装
  - 对象实质上就是属性和方法的容器，它的主要作用就是存储属性和方法，这就是所谓的封装

- 默认情况下，对象的属性是可以任意的修改的，为了确保数据的安全性，在TS中可以对属性的权限进行设置
- 只读属性（readonly）：
  - 如果在声明属性时添加一个readonly，则属性便成了只读属性无法修改
- TS中属性具有三种修饰符：
  - public（默认值），公开的，谁都能用 (默认public) 可以在类、子类和对象中修改
  - protected ， 受保护的，仅仅类和类的子类能使用
  - private ， 私有的，仅类自己里头才能使用
- 示例：
  - public
    - ```
      class a{
          public name:String="xixi"
      }
      let demoa=new a()
      // 因为name属性是使用public 谁都可以修改
      demoa.name="haha"
      console.log(demoa.name)
      ```
  - protected
    - ```
      class a{
          protected name:String="xixi"
          public showname(){
              console.log("因为name是使用protected修饰的只能在当前类和类的子类中使用"+this.name)
          }
      }
      let demoa=new a()
      demoa.showname()

      // 在外部是不能使用的
      // console.log(demoa.name)
      ```
  - private

```
class a{
    private  name:String="xixi"
    public showname(){
        console.log("因为name是使用private仅类自己里头才能使用"+this.name)
    }
}
let demoa=new a()
demoa.showname()

// 在外部是不能使用的
// console.log(demoa.name)
```

- 静态属性
  - 静态属性（方法），也称为类属性。使用静态属性无需创建实例，通过类即可直接使用
  - 静态属性（方法）使用static开头
  - 示例:

```
class Tools{
    static PI:Number = 3.1415926;

  static sum(num1: number, num2: number){
        return num1 + num2
  }
}

console.log(Tools.PI);
console.log(Tools.sum(123, 456));
```

- 继承
  - 继承时面向对象中的又一个特性
  - 通过继承可以将其他类中的属性和方法引入到当前类中
    - 示例:

```
class Animal{
    name: string;
    age: number;

    constructor(name: string, age: number){
        this.name = name;
        this.age = age;
    }
}
```

```
class Dog extends Animal{

    bark(){
        console.log(`${this.name}在汪汪叫！`);
    }
}

const dog = new Dog('旺财', 4);
dog.bark();
```

- 通过继承可以在不修改类的情况下完成对类的扩展
- 重写
  - 发生继承时，如果子类中的方法会替换掉父类中的同名方法，这就称为方法的重写
  - 示例：
    - ```
      class Animal{
          name: string;
          age: number;

          constructor(name: string, age: number){
              this.name = name;
              this.age = age;
          }

          run(){
              console.log(`父类中的run方法！`);
          }
      }

      class Dog extends Animal{

          bark(){
              console.log(`${this.name}在汪汪叫！`);
          }

          run(){
              console.log(`子类中的run方法，会重写父类中的run方法！`);
          }
      }

      const dog = new Dog('旺财', 4);
      dog.bark();
      ```
    - 在子类中可以使用super来完成对父类的引用
      ```

super

```
class A {}
class B extends A {
  constructor() {
    super();  // ES6 要求，子类的构造函数必须执行一次 super 函数，否则会报错。在 constructor 中必须调用 super 方法，因为子类没有自己的 this 对象，而是继承父类的 this 对象，然后对其进行加工,而 super 就代表了父类的构造函数。super 虽然代表了父类 A 的构造函数，但是返回的是子类 B 的实例，即 super 内部的 this 指的是 B
    console.log(this)
  }
}
```

- **可选属性**

  可选属性：可选属性的含义是该属性可以不存在 有时候不要完全匹配一个接口，那么可以用可选属性。使用？号

```
interface IUser{
    name?:String,
    showname():void
}
let user:IUser={
    // 因为name是可选属性所以可以不用定义
    showname(){
        console.log(`名字是${this.name}`)
    }
}
console.log(user.name)
user.showname()
```

ts+vue3

项目创建

在创建项目的时候选中typescript即可

基本单文件组件创建

```
<template>
  <div>
    <h1>我是一个组件</h1>
  </div>
</template>


<script lang="ts" setup>
// 在script中指定类型为ts
</script>

<style>

</style>
```