一、常用 composition API

1.setup

1. vue3.0 中的一个新配置项、值为一个函数
2. setup 是所有 composition Api 表演的舞台
3. 组件中所用到的数据、方法等等 均要配置在 setup 中
4. setup 函数两种返回值：
    - 若返回一个对象、则对象中的属性与方法可以在模板中直接使用
    - 若返回一个渲染函数 则可以自定义渲染内容
5. 注意点：
    - vue2.x 中可以访问到 setup 中的属性和方法
    - 但在 setup 中不能访问到 vue2 中的配置，如果有重名 setup 优先
    - setup 不能是一个 async 函数 因为返回值不再是 return 的对象、而是 promise

setup 参数以及注意点

- setup 的执行时机
    - 在 beforeCreate 之前执行一次，this 是 undefined
- setup 的参数
    - props：值为对象，包含组件外部传递过来的，且组件内部声明接受了的属性
    - context ： 上下文对象
        - attrs：值为对象，包含组件外部传递过来，但没有在 props 中声明的属性，相当于 `this.$attrs`
        - slots: 收到的插槽内容，父级更换为 `v-slot:xxx` 指令、相当于 `this.$slots`
        - emit: 分发自定义事件的函数、相当于 `this.$emit`

ref 函数

- 作用：定义一个响应式数据
- 语法：

```
const xxx = ref(initValue);
```

- 创建一个包含响应式数据的引用对象（referebce 对象 简称 ref 对象）
- JS 中操作数据需要通过.value 属性

- 模板中读取数据不需要 value 属性

reactive 函数

- 作用：定义一个引用类型的响应式数据，基本类型用 ref
- 语法：

```
const xxx = reactive(Array || Object);
```

- reactive 定义的响应式数据是深层次的
- 内部基于 ES6 的 proxy 实现、通过代理对象操作源对象内部数据进行操作

toRef 函数

- 作用：创建一个 ref 对象，其 value 值指向另一个对象的某个属性
- 语法： `const name = toRef(person,'name')`
- 应用：要将响应式对象中的某个属性单独提供给外部使用（重新赋值给其他变量）
- 扩展： `toRef` 与 `torefs` 功能一致，但是可以创建多个 ref 对象，语法 `toRefs(person)`

示例：

```
const person1 = reactive({
  name: '张三',
  age: 15
});
const name = person1.name; //单独提供出去，会丢失响应式
```

正确使用

```
const person1 = reactive({
  name: '张三',
  age: 15
});
const name = toRef(person1, 'name'); // 导出单个ref对象
const { age } = toRefs(person1); // 批量导出多个ref对象，可以解构
```

Vue 响应式原理

vue2.x 响应原理

- 实现原理：
  - 对象类型通过 object.defineProperty()方法堆属性进行读取修改拦截
  - 数组类型：通过重写更新数组的一系列方法来实现拦截（对数组方更新法进行重新包装）

```
//源数据
let preson = { name: '张三', age: 18 };
let p = {};
Object.defineProperty(p, 'name', {
  get() {
    console.log('读取了p属性name');
    return preson.name;
  },
  set(value) {
    console.log('修改了p属性name');
    preson.name = value;
    document.getElementById('preson').innerHTML = preson.name;
  }
});
```

  - 存在问题：
    - 新增属性、删除属性，界面不会更改（可通过手动调用 vue.set()、vue.detelt()进行更新）
    - 直接通过下标修改数组，界面不会更新（手动 splice()也可以更新）

vue3.x 响应原理

- 实现原理：
  - 通过 Proxy 代理拦截对象中属性的操作
  - 通过 Reflect 反射对代理对象的属性进行操作

```
//源数据
let preson = { name: '张三', age: 18 };
const p = new Proxy(preson, {
  // 拦截读取属性值
  get(target, prop) {
    return Reflect.get(target, prop);
  },
  // 拦截设置属性值或添加新属性
  set(target, prop, value) {
    console.log(target, prop, value);
    return Reflect.set(target, prop, value);
  },
  // 拦截删除属性
  deleteProperty(target, prop) {
    return Reflect.deleteProperty(target, prop);
  }
});
```

computed and watch

computed 函数

```
import { reactive, computed } from 'vue';
const person = reactive({
  firstName: '',
  lastName: ''
});
// 计算属性简写、传入函数返回计算后的值（只能读取不能修改）
person.fullName = computed(() => person.firstName + '-' + person.lastName);
// 计算属性完整写法、传入对象修改get set方法（可读可写）
person.fullName = computed({
  get() {
    return person.firstName + '-' + person.lastName;
  },
  set(value) {
    const nameArr = value.split('-');
    person.firstName = nameArr[0];
    person.lastName = nameArr[1];
  }
});
```

watch 函数

- 与 Vue2 中的 watch 功能一致
- 两个小坑:
  - 监视 `reactive` 定义的响应式数据时 `oldValue` 无法获取，强制开启 `deep` 深度监视
  - 监视 `reactive` 定义的响应式数据中某个对象属性时， `deep` 有效

```
// 1、监听ref定义的多个响应式数据,immediate属性代表初始化就执行一次
watch([age, name],(newValue, oldValue) =>
    console.log(newValue, oldValue);
  { immediate: true }
);

// 2、监听 reactive 定义的响应式数据，oldValue 无法获取（deep 深度监视失败）
watch(person,(newValue, oldValue) =>
  console.log(newValue,oldValue);
  { deep: true }
);

// 3、监听 reactive 定义的响应式数据中的某个属性(使用箭头函数)
watch(() => person.type,(newValue, oldValue) =>
  console.log(newValue, oldValue)
);

// 4、监听 reactive 定义的响应式数据中的某些属性(使用数组嵌套箭头函数)
watch([() => person.type, () => person.salary], (newValue, oldValue) =>
  console.log(newValue, oldValue)
);
// 5、特殊情况 监听 reactive 内的对象属性要开启 deep 深度检测
watch(() => person.job,(newValue, oldValue) =>
  console.log(newValue, oldValue),
  { deep: true }
);
```

watchEffect 函数

- watch 的套路就是：既要指明监视的属性，也指明监视的回调
- watchEffect 套路：不用指明监视哪属性，监视中的回调用到哪个，就监视哪个
- watchEffect 与 computed 的区别：
  - computed 计算出来的值 很注重返回值，必须要写
  - 而 watchEffect 更注重过程（函数体），无需写返回值

```
const age = ref(0);
const name = ref('张三');
const person = reactive({
  type: '前端开发',
  salary: 10,
  job: {
    salary: 20
  }
});

// watchEffect函数，函数内部用到哪个就监听哪个
watchEffect(() => {
  const x1 = age;
  console.log(person.job.salary);
  console.log('执行了' + x1.value);
});
```

Vue3 生命周期

钩子函数	作用
beforeCreate()	实例在内存中被创建出来，还没有初始化好 data 和 methods 属性。
create()	实例已经在内存中创建，已经初始化好 data 和 method， 此时还没有开始编译模板。
beforeMount()	已经完成了模板的编译，还没有挂载到页面中。
mounted()	将编译好的模板挂载到页面指定的容器中显示。
beforeUpdate()	状态更新之前执行函数，此时 data 中的状态值是最新的， 但是界面上显示的数据还是旧的，因为还没有开始重新渲染 DOM 节点。
updated()	:此时 data 中的状态值和界面上显示的数据都已经完成了跟新， 界面已经被重新渲染好了
beforeDestroy()	实例被销毁之前。
destroyed()	实例销毁后调用，Vue 实例指示的所有东西都会解绑， 所有的事件监听器都会被移除,所有的子实例也都会被销毁。 组件已经被完全销毁板。

- 在实际开发项目中这些钩子函数如何使用

beforeCreate : 可以在这函数中初始化加载动画
created ：做一些数据初始化，实现函数自执行
mounted： 调用后台接口进行网络请求，拿回数据，配合路由钩子做一些事情
destoryed ：当前组件已被删除，清空相关内容
mounted中做网络请求和重新赋值，在destoryed中清空页面数据。

vue3 与 vue2 的对比

```
Vue2--------------vue3
beforeCreate  -> setup()
created       -> setup()
beforeMount   -> onBeforeMount
mounted       -> onMounted
beforeUpdate  -> onBeforeUpdate
updated       -> onUpdated
beforeDestroy -> onBeforeUnmount
destroyed     -> onUnmounted
activated     -> onActivated
deactivated   -> onDeactivated
errorCaptured -> onErrorCaptured
```

Fragment

- 在 tamplate 中无需一个根组件包裹
- 实际上内部会将多个标签包含在一个 Fragment 虚拟元素中
- 好处: 减少标签层级, 减小内存占用

```
<template>
  <img alt="Vue logo" src="./assets/logo.png" />
  <HelloWorld msg="Welcome to Your Vue.js + TypeScript App" />
</template>
```

provide/inject

- 提供和注入 主要用于跨层级组件(祖孙)间通信
- 在多层嵌套组件中使用，不需要将数据一层一层往下传递
- 可以县实现跨层级组件通信

父组件

```
import { provide } from 'vue';
provide('info', helloClg); //传递的属性名，属性值
```

子孙组件

```
import { inject } from 'vue';
const info = inject('info);//父组件传递的属性名
```

Teleport 传送组件