第5章 CHAPTER 5 响应式API

本章将详细介绍响应式 API，响应式核心实现包括 Proxy 对象代理和 effect 副作用函数。读者可以结合第 2 章的响应式参数逻辑和第 3 章的整体实现，更深入地了解响应式 API 的处理逻辑。

观看视频速览本章主要内容。

5.1 reactive 响应式 API

从本节开始，将介绍 Vue3 中的响应式 API，了解响应式 API 的核心实现逻辑和运行流程。其中，reactive 和 ref 是 Vue3 中最核心的两个响应式 API。下面将详细介绍这两个 API 的实现，以帮助了解响应式 API 的原理。

5.1.1 使用方式

在 Vue2 中，响应式数据是通过 Object.defineProperty 实现的；在 Vue3 中，对该部分进行了重构，通过 Proxy 对象代替 Object.defineProperty 方法，解决 Vue2 中无法侦测 Object 对象的添加和删除、数组元素的增加和删除等问题。

Vue3 中响应式的原理和 Vue2 中差别不大，依然是依赖收集和派发更新两个部分。但在 Vue3 中引入了 effect 副作用函数，该函数与第 2 章的 watchEffect 内部函数实现基本一致。首先通过 reactive 处理响应式数据，当组件内部执行完 setup 后，通过 setupRenderEffect()函数调用 effect 副作用函数。该完整步骤可以在 $ runtime-core_render 文件的 mountComponent 函数内查看。下面展开介绍响应式数据的详细内容。

5.1.2 兼容写法

除上述源码声明响应式数据外，还可以使用 Vue2 的写法声明响应式数据。Vue3 兼容 Vue2 写法，因此可以将参数声明放到 data 方法内，返回响应式数据，统一对属性进行响应式处理。使用方式如下：

```
data() {
  return {
    count: 1
  }
}
```

该处理逻辑在 $ runtime-core_component.ts 文件内实现，在进行 setup 处理时同步处理 data 方法适配 Vue2 写法，涉及代码如下：

```
// support for 2.x options
if (__FEATURE_OPTIONS_API__ && !(__COMPAT__ && skipOptions)) {
  setCurrentInstance(instance)
  pauseTracking()
  // 处理 data 数据的情况
  applyOptions(instance, Component)
  resetTracking()
  unsetCurrentInstance()
}
```

继续查看响应式参数的内部实现。根据调用逻辑,将通过 applyOptions 函数对 data 返回的对象进行处理,涉及代码如下:

```
if (dataOptions) {
  if (__DEV__ && !isFunction(dataOptions)) { ... }
  const data = dataOptions.call(publicThis, publicThis)
  if (__DEV__ && isPromise(data)) { ... }
  if (!isObject(data)) {
    __DEV__ && warn(`data() should return an object.`)
  } else {
    instance.data = reactive(data)
    if (__DEV__) { ... }
  }
}
```

5.1.3 reactive()函数

此处调用 reactive()函数对 data 数据进行处理,通过 reactive()结合 effect 副作用函数实现响应式功能,代码位于 $reactivity_reactive 文件内,涉及代码如下:

```
// $ reactivity_reactive
export function reactive(target: object) {
  if (target && (target as Target)[ReactiveFlags.IS_READONLY]) {
    return target
  }
  return createReactiveObject(
    target,
    false,
    mutableHandlers,
    mutableCollectionHandlers,
    reactiveMap
  )
}
```

5.1.4 createReactiveObject()函数

上述代码通过一个高阶函数来返回另外一个创建响应式对象的函数 createReactiveObject()。通过高阶函数的方式保存参数,再通过参数形态导出更高阶的 reactive、shallowReadonly 等函数。

注:*在无类型 lambda 演算中,所有函数都是高阶的;在有类型 lambda 演算(大多数函数式编程语言都从中演化而来)中,高阶函数就是参数为函数或返回值为函数的函数。在函数式编程中,返回另一个函数的高阶函数被称为柯里化函数。*

createReactiveObject()函数涉及代码如下:

```
function createReactiveObject(
  target: Target,
```

```
    isReadonly: boolean,
    baseHandlers: ProxyHandler<any>,
    collectionHandlers: ProxyHandler<any>,
    proxyMap: WeakMap<Target, any>
  ) {
    if (!isObject(target)) {
      // 传入的不是对象,直接返回,不做处理
      // 在开发环境中提出警告
      if (__DEV__) {
        console.warn(`value cannot be made reactive: ${String(target)}`);
      }
      return target;
    }
    // 如果 target 已经是一个 reactive 对象,则直接返回
    // 异常:在响应式对象上调用 readonly()
    if (
      target[ReactiveFlags.RAW] &&
      !(isReadonly && target[ReactiveFlags.IS_REACTIVE])
    ) {
      return target;
    }
    // 存在代理,直接返回
    const existingProxy = proxyMap.get(target)
    if (existingProxy) {
      return existingProxy
    }
    // 是否为可代理数据类型
    const targetType = getTargetType(target)
    if (targetType === TargetType.INVALID) {
      return target
    }
    // 生成代理对象
    const proxy = new Proxy(
      target,
      targetType === TargetType.COLLECTION ? collectionHandlers : baseHandlers
    )
    // 在原对象上缓存代理后的对象
    proxyMap.set(target, proxy)
    return proxy
  }
```

target 可代理对象包括：没有 skip 标识的源对象，没有冻结的对象和 Object、Array、Map、Set、WeakMap、WeakSet 类型的对象。createReactiveObject()的主要逻辑可简化成如下几个步骤：

(1) 过滤不可代理的情况；

(2) 查询可使用缓存的情况；

(3) 生成缓存代理对象。

下面结合代码对以上步骤进行分析。

(1) 过滤不可代理的情况：通过 reactive 代理 readonly 对象。

```
const obj = { key: 1 }
const n1 = readonly(obj)
const n2 = reactive(n1)
console.log('直接返回 obj 对象')
```

(2) 查询可使用缓存的情况：通过 reactive 多次代理同一个对象。

```
const obj = { key: 1 }
const n1 = reactive(obj)
const n2 = reactive(obj)
console.log('返回第一次代理 obj 对象')
```

(3) 生成缓存代理对象：通过 reactive 代理 reactive 响应式对象。

```
const obj = { key: 1 }
const n1 = reactive(obj)
const n2 = reactive(n1)
console.log('直接返回 obj 对象')
```

5.1.5 mutableHandlers()函数

createReactiveObject()函数内部没有复杂逻辑，需关注的还是 ProxyHandler 部分，涉及代码如下：

```
export const mutableHandlers: ProxyHandler<object> = {
  get,
  set,
  deleteProperty,
  has,
  ownKeys
}
```

5.1.6 createGetter()函数

由上述代码可知，Vue3 对 5 种方法做了代理，分别是 get、set、has、deleteProperty 和 ownKeys。对用户来说接触最多的是 get 和 set 方法。

get 通过 createGetter()函数实现，涉及代码如下：

```
function createGetter(isReadonly = false, shallow = false) {
  return function get(target: Target, key: string | symbol, receiver: object) {
    // 1. reactive 标识位处理
    if (key === ReactiveFlags.IS_REACTIVE) {
      return !isReadonly;
    } else if (key === ReactiveFlags.IS_READONLY) {
      return isReadonly;
    } else if (
      key === ReactiveFlags.RAW &&
      receiver ===
        (isReadonly
          ? shallow
            ? shallowReadonlyMap
            : readonlyMap
          : shallow
          ? shallowReactiveMap
          : reactiveMap
        ).get(target)
    ) {
      return target;
    }
    // 2. 处理数组方法 key,若有缓存,直接返回
    const targetIsArray = isArray(target);
    if (!isReadonly && targetIsArray && hasOwn(arrayInstrumentations, key)) {
      return Reflect.get(arrayInstrumentations, key, receiver)
```

```
    }
    // 3. 缓存取值
    const res = Reflect.get(target, key, receiver);
    // 4. 过滤无须 track 的 key
    if (
      // 访问 builtInSymbols 内的 symbol key 或者访问原型和 ref 内部属性
      isSymbol(key) ? builtInSymbols.has(key) : isNonTrackableKeys(key)
    ) {
      return res;
    }
    // 5. 依赖收集
    if (!isReadonly) {
      // 非 readonly 收集一次类型为 get 的依赖
      track(target, TrackOpTypes.GET, key);
    }
    // 6. 处理取出的值
    if (shallow) {
      // 浅代理则直接返回通过 key 取到的值
      return res;
    }
    if (isRef(res)) {
      // 如果通过 key 去除的是 ref,则自动解开,仅针对对象
      const shouldUnwrap = !targetIsArray || !isIntegerKey(key)
      return shouldUnwrap ? res.value : res
    }
    if (isObject(res)) {
      // 如果通过 key 取出是对象,且 shallow 为 false,则进行递归代理
      return isReadonly ? readonly(res) : reactive(res);
    }
    return res;
  };
}
```

上述方法传入两个参数,并通过高阶函数的方式保存两个参数的值。函数体内首先针对3个 reactive 标识 key 进行判断,根据 key 类型不同返回不同值。判断流程如图 5.1 所示。

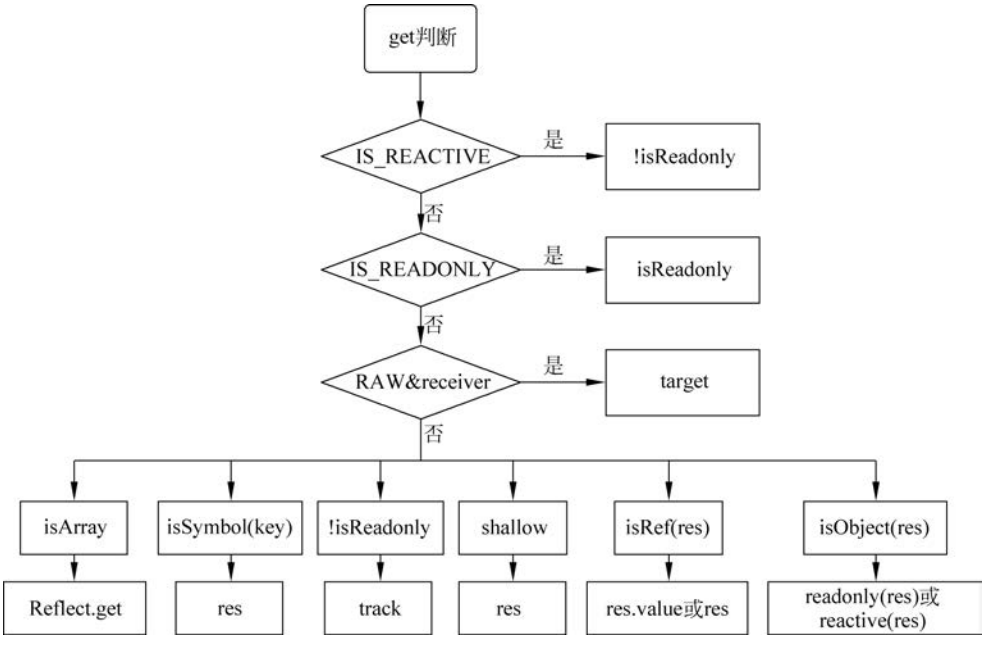

图 5.1 判断流程

由图 5.1 可知，整个函数执行共分以下几种情况：

(1) 判断 key 是否为 3 种特定类型的值。若是，则根据 key 直接返回对应值，结束当前函数的执行。

(2) 若 target 是数组，则调用 arrayInstrumentations 函数针对数组进行处理。

(3) 使用 Reflect.get 对 target 进行取值和依赖收集。

(4) 取值前过滤不需要依赖收集的 key，直接返回对应值。

(5) 针对非 readonly 的情况进行一次 get 的依赖收集。

注：Reflect 是一个内置的对象，用于获取目标对象的行为，提供拦截 JavaScript 操作的方法。这些方法与 proxy handlers 的方法相同。Reflect 不是一个函数对象，因此它是不可构造的。

对于当前 key 取出的值仍需要按基本类型、ref 类型和对象类型来处理。

(1) 基本类型，直接返回。

(2) ref 类型，原数据为对象时解构 ref 类型后返回，原数据为数组时直接返回。

(3) 对象类型，若是 shallow 浅代理则直接返回，若不是则返回递归代理的代理对象。

在上述步骤(2)中，针对数组处理的 arrayInstrumentations()函数内部实现了对数组的 includes、indexOf 和 lastIndexOf 方法重写，加入依赖收集。代码实现如下：

```
const arrayInstrumentations = /* #__PURE__ */ createArrayInstrumentations()
function createArrayInstrumentations() {
  const instrumentations: Record<string, Function> = {}
  ;(['includes', 'indexOf', 'lastIndexOf'] as const).forEach(key => {
    instrumentations[key] = function (this: unknown[], ...args: unknown[]) {
      const arr = toRaw(this) as any
      for (let i = 0, l = this.length; i < l; i++) {
        // 针对数组元素进行依赖收集
        track(arr, TrackOpTypes.GET, i + '')
      }
      const res = arr[key](...args)
      if (res === -1 || res === false) {
        return arr[key](...args.map(toRaw))
      } else {
        return res
      }
    }
  })
  ;(['push', 'pop', 'shift', 'unshift', 'splice'] as const).forEach(key => {
    instrumentations[key] = function (this: unknown[], ...args: unknown[]) {
      // 上述方法操作时，会改变数组长度，此处暂停依赖收集，避免无限递归
      pauseTracking()
      const res = (toRaw(this) as any)[key].apply(this, args)
      resetTracking()
      return res
    }
  })
  return instrumentations
}
```

在完成对应解析后会进行依赖收集，依赖收集通过 track 实现，下面介绍依赖收集函数。该函数位于 $ reactivity_effect.ts 内，涉及代码如下：

```
export function track(target: object, type: TrackOpTypes, key: unknown) {
  // 依赖收集进行的前置条件：
  // 1. 全局收集标识开启
```

```
    // 2. 存在激活的副作用函数
    if (!isTracking()) {
      return
    }
    // 创建依赖收集 map target → deps → effect
    let depsMap = targetMap.get(target);
    if (!depsMap) {
      targetMap.set(target, (depsMap = new Map()));
    }
    let dep = depsMap.get(key);
    if (!dep) {
      depsMap.set(key, (dep = new Set()));
    }
    trackEffects(dep, eventInfo)
  }
  export function trackEffects(
      dep: Dep,
      debuggerEventExtraInfo?: DebuggerEventExtraInfo
    ) {
      let shouldTrack = false
      // 设置收集深度
      if (effectTrackDepth <= maxMarkerBits) {
        if (!newTracked(dep)) {
          dep.n |= trackOpBit // set newly tracked
          shouldTrack = !wasTracked(dep)
        }
      } else {
        // 全量清理
        shouldTrack = !dep.has(activeEffect!)
      }

      if (shouldTrack) {
        // 依赖收集副作用
        dep.add(activeEffect!)
        // 副作用保存依赖
        activeEffect!.deps.push(dep)
        // 依赖收集的钩子函数
        if (__DEV__ && activeEffect!.onTrack) {
          activeEffect!.onTrack(
            Object.assign(
              {
                effect: activeEffect!
              },
              debuggerEventExtraInfo
            )
          )
        }
      }
    }
```

track 的目标很简单，建立当前 key 与当前激活 effect 的依赖关系，源码中使用了一个较为复杂的方式来保存这种依赖关系，依赖关系如图 5.2 所示。

完成后整个数据结构如图 5.3 所示。

通过 target→key→dep 的数据结构，完整地存储了对象、键值和副作用的关系，并且通过 set 实例来对 effect 副作用函数去重。

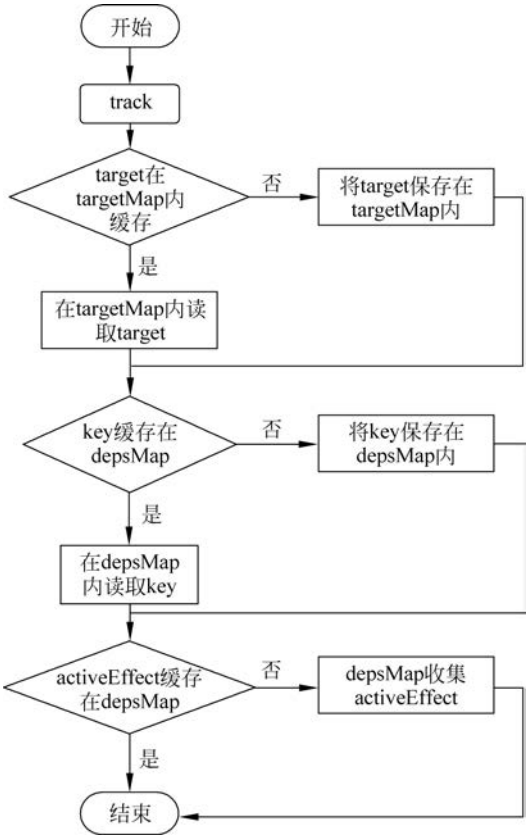

图 5.2　依赖关系

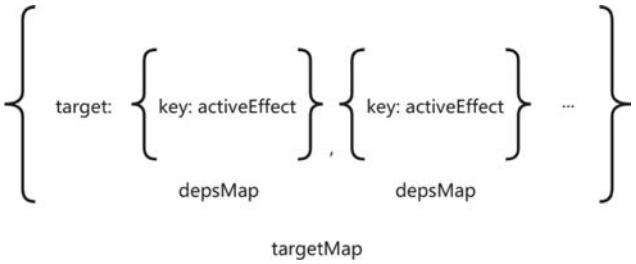

图 5.3　数据结构

理清依赖关系的数据结构后，track 函数基本介绍完成。

5.1.7　createSetter()函数

依赖收集主要通过 get 触发，派发更新主要通过 set 触发。set 方法在 $reactivity_reactive 文件内，通过 createSetter()函数实现派发更新，具体代码如下：

```
function createSetter(shallow = false) {
  return function set(
    target: object,
    key: string | symbol,
    value: unknown,
    receiver: object
  ): boolean {
    // 取旧值
```

```
    const oldValue = (target as any)[key];
    if (!shallow && !isReadonly(value)) {
      // 在深度代理情况下,需要手动处理属性值为 ref 的情况,将 trigger 交给 ref 来触发
      value = toRaw(value);
      oldValue = toRaw(oldValue)
      if (!isArray(target) && isRef(oldValue) && !isRef(value)) {
        oldValue.value = value;
        return true;
      }
    } else {
      // 在浅代理模式下,行为与普通对象一致
    }
    // 判断是新增或者修改
    const hadKey =
      isArray(target) && isIntegerKey(key)
        ? Number(key) < target.length
        : hasOwn(target, key)
    // 设置新值
    const result = Reflect.set(target, key, value, receiver);
    // 如果修改了通过原型查找得到的属性,无须 trigger
    if (target === toRaw(receiver)) {
      if (!hadKey) {
        // 新增
        trigger(target, TriggerOpTypes.ADD, key, value);
      } else if (hasChanged(value, oldValue)) {
        // 修改时,需要去除未改变的情况;
        // 数组增加元素时,会使 length 改变,但在达到 length 修改的 set 时;
        // 数组已经添加元素成功,取到的 oldValue 会与 value 相等,直接过滤掉此次不必要的
        // trigger.
        trigger(target, TriggerOpTypes.SET, key, value, oldValue);
      }
    }
    return result;
  };
}
```

set 函数的主要目标是修改值并正确地派发更新。首先在非 shallow 的情况下,需要手动处理属性值为 ref 的情况,将 trigger()交给 ref 来触发。

赋值完成后的 target 并没有变化,说明当前设置的 key 来自原型链,无须触发 trigger()。

如果确定是对 target 上 key 的修改,仍需要进行 add 和 set 的区分。因为存在一种场景如下:

```
let arr = reactive([1, 2, 3]);
arr.push(4);
```

当向响应式数组添加元素时会触发数组 length 的修改,在 length 属性修改前,数组元素已经添加成功,通过(target as any)[key]取到的值会与 value 相等。在这种情况下无须执行 trigger()函数,使用 hasChanged()函数来进行过滤。

通过 trigger()函数触发更新,具体代码实现如下:

```
export const ITERATE_KEY = Symbol(__DEV__ ? 'iterate' : '')
export const MAP_KEY_ITERATE_KEY = Symbol(__DEV__ ? 'Map key iterate' : '')
const targetMap = new WeakMap<any, KeyToDepMap>()
export function trigger(
  target: object,
  type: TriggerOpTypes,
```

```
  key?: unknown,
  newValue?: unknown,
  oldValue?: unknown,
  oldTarget?: Map<unknown, unknown> | Set<unknown>
) {
  const depsMap = targetMap.get(target)
  if (!depsMap) {
    // 未被依赖收集
    return
  }
  let deps: (Dep | undefined)[] = []
  if (type === TriggerOpTypes.CLEAR) {
    // 保存 target 的所有 effect 到 deps
    deps = [...depsMap.values()]
  } else if (key === 'length' && isArray(target)) {
    // 数组长度变化
    depsMap.forEach((dep, key) => {
      if (key === 'length' || key >= (newValue as number)) {
        deps.push(dep)
      }
    })
  } else {
    // schedule runs for SET | ADD | DELETE
    // 除去以上两种特殊情况,若 key 还存在,则直接添加所有有依赖的副作用函数
    if (key !== void 0) {
      deps.push(depsMap.get(key))
    }
    // 根据类型操作待缓存数据
    switch (type) {
      case TriggerOpTypes.ADD:
        if (!isArray(target)) {
          deps.push(depsMap.get(ITERATE_KEY))
          if (isMap(target)) {
            deps.push(depsMap.get(MAP_KEY_ITERATE_KEY))
          }
        } else if (isIntegerKey(key)) {
          // 添加新索引到数组,长度改变
          deps.push(depsMap.get('length'))
        }
        break
      case TriggerOpTypes.DELETE:
        if (!isArray(target)) {
          deps.push(depsMap.get(ITERATE_KEY))
          if (isMap(target)) {
            deps.push(depsMap.get(MAP_KEY_ITERATE_KEY))
          }
        }
        break
      case TriggerOpTypes.SET:
        if (isMap(target)) {
          deps.push(depsMap.get(ITERATE_KEY))
        }
        break
    }
  }
  const eventInfo = __DEV__
    ? { target, type, key, newValue, oldValue, oldTarget }
```

```
      : undefined
    // 如果 deps 的长度为 1,则代表为空
    if (deps.length === 1) {
      if (deps[0]) {
        if (__DEV__) {
          triggerEffects(deps[0], eventInfo)
        } else {
          triggerEffects(deps[0])
        }
      }
    } else {
      // 新增数组变量 effects,保存所有的 effect 副作用函数
      const effects: ReactiveEffect[] = []
      for (const dep of deps) {
        if (dep) {
          effects.push(...dep)
        }
      }
      // 调用 triggerEffects()执行 effect 副作用函数
      if (__DEV__) {
        triggerEffects(createDep(effects), eventInfo)
      } else {
        triggerEffects(createDep(effects))
      }
    }
  }
  export function triggerEffects(
    dep: Dep | ReactiveEffect[],
    debuggerEventExtraInfo?: DebuggerEventExtraInfo
  ) {
    // 若是数组则直接遍历,若不是则解构并保存为数组
    for (const effect of isArray(dep) ? dep : [...dep]) {
      // 如果 effect 不是激活状态或者允许递归,则触发 effect 副作用函数执行
      if (effect !== activeEffect || effect.allowRecurse) {
        // 开发环境下运行 trigger 钩子函数
        if (__DEV__ && effect.onTrigger) {
          effect.onTrigger(extend({ effect }, debuggerEventExtraInfo))
        }
        // 如果存在调度,则使用调度来执行 effect 副作用函数
        if (effect.scheduler) {
          effect.scheduler()
        } else {
          effect.run()
        }
      }
    }
  }
```

上述代码对 trigger()函数的核心步骤标注,主要有如下步骤:

(1) 依据不同类型保存副作用函数列表;

(2) 过滤当前激活副作用函数,添加其他副作用函数;

(3) 遍历所有被添加副作用函数,派发更新。

targetMap 对象存储依赖和副作用函数之间的关系。trigger()函数会对 targetMap 对象进行遍历,调用需要被执行的副作用函数。

5.1.8 ref 解析

前面已经讲过 ref 的出现是为了包装基本类型以实现代理，API 形态也有所展现，通过.value 来进行访问和修改。ref 函数的实现代码如下：

```
export function ref(value?: unknown) {
  return createRef(value, false);
}
function createRef(rawValue: unknown, shallow = false) {
  // 如果已经是 ref,则直接返回
  if (isRef(rawValue)) {
    return rawValue;
  }
  return new RefImpl(rawValue, shallow)
}
class RefImpl<T> {
  private _value: T
  private _rawValue: T
  public dep?: Dep = undefined
  // ref 标识
  public readonly __v_isRef = true
  constructor(value: T, public readonly _shallow: boolean) {
    // 如果是浅代理,则使用 toRaw 获取原始值
    this._rawValue = _shallow ? value : toRaw(value)
    this._value = _shallow ? value : toReactive(value)
  }
  get value() {
    // 依赖收集
    trackRefValue(this)
    return this._value
  }
  set value(newVal) {
    // 产生了变化,则修改
    newVal = this._shallow ? newVal : toRaw(newVal)
    if (hasChanged(newVal, this._rawValue)) {
      this._rawValue = newVal
      this._value = this._shallow ? newVal : toReactive(newVal)
      // 派发更新
      triggerRefValue(this, newVal)
    }
  }
}
// 如果是对象,则使用 reactive 代理对象
export const toReactive = <T extends unknown>(value: T): T =>
  isObject(value) ? reactive(value) : value
```

ref 也是以一个高阶函数的形式来创建的，因为 ref 也存在 shallowRef；通过上述源码也可以看到 ref 返回的是一个对象，因此在读取和写入时，需要显式地指定.value，用于触发 get 和 set 拦截。在完成对 reactive 方法解读的情况下，再查看 ref 代码，会发现 ref 内部使用了一个对象来包装传入的值，若传入的是对象，则直接使用 reactive 代理，ref 只负责处理来自.value 的访问和修改。

5.1.9 总结

响应式对象调用 reactive 函数对传入的对象进行判断，决定是否调用 createReactiveObject()

函数,该函数通过 Proxy 对象实现对对象的拦截。使用代理函数实现对传入对象的代理监听,保证在对象值变化时能及时得到通知。createReactiveObject()函数内部通过 get 方法收集依赖和 set 方法派发更新。在使用 get 进行依赖收集时,巧妙地使用 Reflect 对象进行 get 拦截,再触发整个依赖的收集。在使用 set 进行派发更新时,对读取收集到的 effect 副作用函数进行遍历,巧妙地通过 run 方法,来控制 effect 副作用函数的执行时机。

整个拦截处理完成后,再根据情况执行属性值读取或修改操作。根据传入的参数可知,此处主要针对对象进行处理,关于数组的处理方法将在 5.1.6 节详细介绍。整个响应式参数涉及流程如图 5.4 所示。

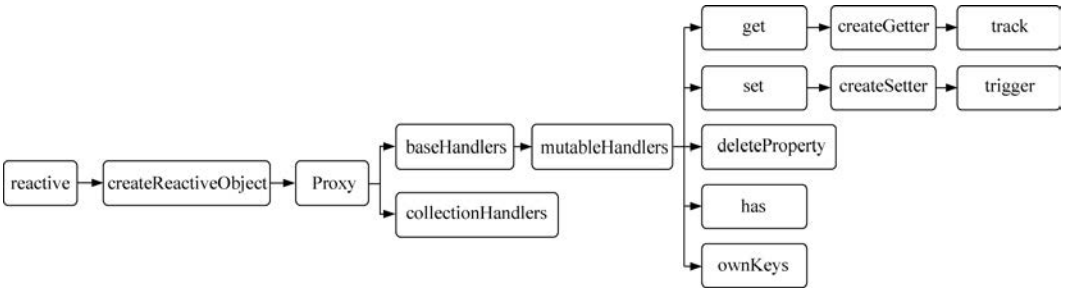

图 5.4 响应式参数

上面介绍了 track 依赖收集和 trigger 派发更新,串联 track、trigger 和 effect 副作用函数的执行步骤,完成响应式对象的处理。5.2 节将介绍 effect 副作用函数。

5.2 effect 副作用函数

当响应式数据变化时,effect 副作用函数重新触发执行。接下来对 effect 副作用函数进行分析,其核心流程与第 2 章基本类似。

5.2.1 实现

effect 副作用函数内部的主要作用是针对组件的首次渲染和更新,派发处理逻辑,默认会在首次渲染和触发响应式数据变化阶段执行。在组件首次挂载时,触发 effect 副作用函数;在组件更新时,重新执行 effect 副作用函数。该函数位于 $ runtime-core_render 文件内,当组件更新时,触发 componentUpdateFn()函数的执行。具体代码实现如下:

```
// $ runtime - core_render
const effect = (instance.effect = new ReactiveEffect(
  componentUpdateFn,
  () => queueJob(instance.update),
  instance.scope // 限制在组件范围内进行跟踪
))
const update = (instance.update = effect.run.bind(effect) as SchedulerJob)
update.id = instance.uid
update()
```

componentUpdateFn()函数内有 mount(挂载)和 update(更新)两个流程。该函数在 update 属性上,首次挂载的时候,调用 update 方法执行一次 effect 副作用函数。

5.2.2 mount(挂载)

mount(挂载)流程,主要执行 patch 挂载,并且标识当前组件为已挂载状态,实现代码

如下：

```
// $ runtime - core_render
if (!instance.isMounted) {
  let vnodeHook: VNodeHook | null | undefined
  const { el, props } = initialVNode
  const { bm, m, parent } = instance
  const isAsyncWrapperVNode = isAsyncWrapper(initialVNode)
  // 服务器端渲染
  if (el && hydrateNode) {
    const hydrateSubTree = () => {
      instance.subTree = renderComponentRoot(instance)
      hydrateNode!(
        el as Node,
        instance.subTree,
        instance,
        parentSuspense,
        null
      )
    }
    if (isAsyncWrapperVNode) {
      ;(initialVNode.type as ComponentOptions).__asyncLoader!().then(
        () => !instance.isUnmounted && hydrateSubTree()
      )
    } else {
      hydrateSubTree()
    }
  } else {
    const subTree = (instance.subTree = renderComponentRoot(instance))
    patch(
      null,
      subTree,
      container,
      anchor,
      instance,
      parentSuspense,
      isSVG
    )
    initialVNode.el = subTree.el
  }
  // onVnodeMounted
  if (
    !isAsyncWrapperVNode &&
    (vnodeHook = props && props.onVnodeMounted)
  ) {
    const scopedInitialVNode = initialVNode
    queuePostRenderEffect(
      () => invokeVNodeHook(vnodeHook!, parent, scopedInitialVNode),
      parentSuspense
    )
  }
  if (initialVNode.shapeFlag & ShapeFlags.COMPONENT_SHOULD_KEEP_ALIVE) {
    instance.a && queuePostRenderEffect(instance.a, parentSuspense)
  }
  instance.isMounted = true
  initialVNode = container = anchor = null as any
}
```

5.2.3 update(更新)

update(更新)流程同样执行 patch 渲染,因为是更新操作,涉及 diff 新旧 VNode 的变化,需要提前处理 props、slot 等内容,所以对更新逻辑进行分开处理。具体代码实现如下:

```
else {
  let { next, bu, u, parent, vnode } = instance
  let originNext = next
  let vnodeHook: VNodeHook | null | undefined
  toggleRecurse(instance, false)
  if (next) {
    next.el = vnode.el
    updateComponentPreRender(instance, next, optimized)
  } else {
    next = vnode
  }
  if (
    __COMPAT__ &&
    isCompatEnabled(DeprecationTypes.INSTANCE_EVENT_HOOKS, instance)
  ) {
    instance.emit('hook:beforeUpdate')
  }
  toggleRecurse(instance, true)
  const nextTree = renderComponentRoot(instance)
  const prevTree = instance.subTree
  instance.subTree = nextTree
  patch(
    prevTree,
    nextTree,
    //如果是 teleport,那么父组件可能会改变
    hostParentNode(prevTree.el!)!,
    // 如果是 fragment,那么 anchor 可能会改变
    getNextHostNode(prevTree),
    instance,
    parentSuspense,
    isSVG
  )
  next.el = nextTree.el
  if (originNext === null) {
    updateHOCHostEl(instance, nextTree.el)
  }
  if (
    __COMPAT__ &&
    isCompatEnabled(DeprecationTypes.INSTANCE_EVENT_HOOKS, instance)
  ) {
    queuePostRenderEffect(
      () => instance.emit('hook:updated'),
      parentSuspense
    )
  }
}
```

5.2.4 创建 effect 副作用函数

effect 副作用函数本身对应的是第 2 章的 watchEffect()函数在 Vue3 中源码的实现。接

下来简单查看该函数的内部逻辑，以便于理解其作用。具体代码实现如下：

```
export function effect<T = any>(
  fn: () => T,
  options?: ReactiveEffectOptions
): ReactiveEffectRunner {
  // 获取副作用函数
  if ((fn as ReactiveEffectRunner).effect) {
    fn = (fn as ReactiveEffectRunner).effect.fn
  }
  const _effect = new ReactiveEffect(fn)
  if (options) {
    extend(_effect, options)
    if (options.scope) recordEffectScope(_effect, options.scope)
  }
  // lazy 属性值控制是否立即执行
  if (!options || !options.lazy) {
    _effect.run()
  }
  const runner = _effect.run.bind(_effect) as ReactiveEffectRunner
  runner.effect = _effect
  return runner
}
```

上述代码对 fn 本身进行处理后，创建了一个 effect 副作用函数，再判断是否需要懒加载，判断 effect 副作用函数是否立即执行。

5.2.5 ReactiveEffect()函数

effect 副作用函数调用了 ReactiveEffect()构造函数，该函数接收 3 个参数：执行函数、异步队列任务和可选参数 scope。ReactiveEffect()构造函数的主要实现逻辑为：创建一个 effect 副作用函数；设置内部属性使需要执行的 effect 副作用函数在缓存队列快速被找到。捕获传入的 fn 函数执行的错误，避免 fn 函数错误导致 effect 执行崩溃。具体代码实现如下：

```
export class ReactiveEffect<T = any> {
    active = true
    deps: Dep[] = []
    // 创建后可以再次计算
    computed?: boolean
    allowRecurse?: boolean
    onStop?: () => void
    // dev only
    onTrack?: (event: DebuggerEvent) => void
    // dev only
    onTrigger?: (event: DebuggerEvent) => void
    constructor(
      public fn: () => T,
      public scheduler: EffectScheduler | null = null,
      scope?: EffectScope | null
    ) {
      recordEffectScope(this, scope)
    }
    run() {
      if (!this.active) {
        // 非激活状态处理
        return this.fn()
```

```
        }
        // 确保副作用栈中没有当前副作用函数
        if (!effectStack.includes(this)) {
          try {
            // 设置当前副作用函数为激活副作用
            effectStack.push((activeEffect = this))
            // 开启收集
            enableTracking()
            // 递归次数加 1
            trackOpBit = 1 << ++effectTrackDepth
            // 判断是否达到最大递归调用次数
            if (effectTrackDepth <= maxMarkerBits) {
              initDepMarkers(this)
            } else {
              // 清理当前副作用函数依赖
              cleanupEffect(this)
            }
            return this.fn()
          } finally {
            // 如果小于最大调用次数,则完成
            if (effectTrackDepth <= maxMarkerBits) {
              finalizeDepMarkers(this)
            }
            // 完成本次 effect 副作用函数执行后,标识位减少 1
            trackOpBit = 1 << --effectTrackDepth
            // 重置收集栈,删除已执行的副作用函数,重置当前激活的副作用函数
            resetTracking()
            effectStack.pop()
            const n = effectStack.length
            activeEffect = n > 0 ? effectStack[n - 1] : undefined
          }
        }
      }
      stop() {
        if (this.active) {
          cleanupEffect(this)
          if (this.onStop) {
            this.onStop()
          }
          this.active = false
        }
      }
    }
```

effect 副作用函数负责预处理 fn 函数,确保 fn 函数是一个非 effect 副作用函数。函数内部 lazy 选项和 computed 强相关,配置成 true 会使得 effect 副作用函数默认不立即执行。

由 ReactiveEffect 函数的内部实现逻辑可知,effect 副作用函数仅是一个包裹函数,在它的内部执行 fn 函数和处理 effectStack 相关内容。

5.2.6 处理激活状态

effect.active 表示副作用函数的激活状态,默认为 true。在停止一个副作用函数后会将其置成 false。非激活状态的 effect 副作用函数被调用时,如果不存在异步调度,则直接执行 fn 函数;如果存在异步调度,则直接返回 null。

5.2.7 清除操作

effect.deps 是一个数组，存储的是该 effect 副作用函数依赖的每个属性的 depsSet 副作用函数表，track 阶段建立的依赖存储在表中。每个响应式对象触发依赖收集的 key 都会对应 depsSet 表中的一个副作用函数。

effect 的 deps 存储是的当前 effect 依赖属性的副作用 depsSet 表，这是一个双向指针的处理方式，不仅在依赖收集的时候，会将副作用函数与 target→key→depsSet 关联起来，同时也会保持 depsSet 的引用存储在 effect.deps 上。当超出最大递归次数，或调用 effect 副作用函数的 stop 方法时，将调用 cleanupEffect()函数清理 effect 副作用函数中保存的依赖。

cleanupEffect()函数的代码实现如下：

```
function cleanupEffect(effect: ReactiveEffect) {
  const { deps } = effect
  if (deps.length) {
    for (let i = 0; i < deps.length; i++) {
      deps[i].delete(effect)
    }
    deps.length = 0
  }
}
```

5.2.8 执行 fn

fn 的执行采用 try...finally 来捕获错误，即使 fn 执行出错，还是能保证 effectStack、activeEffect 的正确维护。

在正式执行 fn 之前，会有一个压栈的操作，由于 effect 的执行会存在嵌套的情况，比如，组件渲染函数的执行遇到了子组件就会跳到子组件的渲染函数中，函数的嵌套调用使用到栈结构，而栈的先进后出性质能很好地保证 activeEffect 的正确回退。fn 执行完成后，就进行出栈操作，跳回到上一个 effect 中，至此整个逻辑完整地进行了串联。

5.2.9 总结

整个响应式逻辑已介绍完成，包括 effect()函数与 track 和 trigger 之间的关系，串联响应式数据的处理流程。可以结合第 2 章的逻辑一起理解整个流程。虽然 Vue3 内的核心逻辑代码有所增加，对不同值类型、边界情况等进行了处理，但核心逻辑实现与第 2 章完全一致。下面通过关系图(见图 5.5)再对整个 effect 副作用函数的执行进行梳理。以便读者理解 effect 副作用函数的作用以及响应式数据的 get 和 set 内部作用。

图 5.5 createReactiveEffect()函数流程

5.3 节将会介绍使用 effect、track 和 trigger 实现的方法。

5.3 watch 监听

前面介绍了响应式数据原理后,本节将介绍响应式核心 API 的使用。watch 函数就是对响应式数据的封装实现。watch 函数在 Vue2.x 已存在,在 Vue3 中进行了增强,并且扩展出 watchEffect()函数。

5.3.1 watch 函数

watch 函数可以在 $reactivity_apiWatch.ts 文件内查看。在该文件内可以看到多个 watch 函数导出,该方法为 TypeScript 语法内的函数重载。具体代码实现如下:

```
export function watch<T = any, Immediate extends Readonly<boolean> = false>(
  source: T | WatchSource<T>,
  cb: any,
  options?: WatchOptions<Immediate>
): WatchStopHandle {
  return doWatch(source as any, cb, options)
}
```

上述 watch 函数接收 3 个参数:WatchSource、cb 和 options。WatchSource 是需要通过 watch 监听的函数,cb 是需要执行的回调函数,options 是对应的配置属性。通过高阶函数方法调用 doWatch()函数。下面根据代码逻辑查看 doWatch()函数的实现。该函数实现内容较多,主要包括以下逻辑:

(1) 初始化 getter 和相关变量;

(2) 判断传入数据类型,根据数据类型执行不同的数据处理,得到 getter()函数;

(3) 处理 SSR 情况;

(4) 创建调度工作;

(5) 初始化调度函数;

(6) 创建 effect 副作用函数;

(7) 返回 stop 函数。

上面对 doWatch()函数所做的工作进行了简单的介绍,下面详细介绍 doWatch 函数内部的执行情况。

5.3.2 初始化

根据传入的参数对 getter()函数执行不同的初始化。watch 支持的传入参数类型为 ref、reactive object、getter/effect function 和包含上述类型的数组。具体逻辑如下:

如果数据源为 ref,则直接获取 value 值并返回,并且判断是否为浅代理,标识是否强制触发。

```
if (isRef(source)) {
  getter = () => source.value
  forceTrigger = !!source._shallow
}
```

如果数据源为 reactive 类型,则直接返回该对象,并强制设置为深度监听。

```
else if (isReactive(source)) {
   getter = () => source
   deep = true
}
```

如果数据源为数组类型，则标识为多元数据，并根据是否有 reactive 类型数据标识强制触发。完成标识后遍历数组的每一项，若为 ref 类型则读取 value 值；若为 reactive 类型则递归数据；若为函数，则执行对应函数，并通过 callWithErrorHandling()函数捕获对应错误。

```
else if (isArray(source)) {
  isMultiSource = true
  forceTrigger = source.some(isReactive)
  getter = () => source.map(s => {
    if (isRef(s)) {
      return s.value
    } else if (isReactive(s)) {
      return traverse(s)
    } else if (isFunction(s)) {
      return callWithErrorHandling(s, instance, ErrorCodes.WATCH_GETTER)
    } else {
      __DEV__ && warnInvalidSource(s)
    }
  })
}
```

如果数据源(source)为函数类型，且有回调函数，则执行对应的 source 函数，并通过 callWithErrorHandling()函数捕获对应的错误；若没有回调函数，则判断是否有上下文，若有上下文则判断是否为激活状态，清理上一次执行的回调函数，并执行该 source 函数。具体代码实现如下：

```
else if (isFunction(source)) {
  if (cb) {
    // getter with cb
    getter = () =>
      callWithErrorHandling(source, instance, ErrorCodes.WATCH_GETTER)
  } else {
    // no cb -> simple effect
    getter = () => {
      if (instance && instance.isUnmounted) {
        return
      }
      if (cleanup) {
        cleanup()
      }
      return callWithAsyncErrorHandling(
        source,
        instance,
        ErrorCodes.WATCH_CALLBACK,
        [onInvalidate]
      )
    }
  }
}
```

完成 getter()函数后，处理 Vue2.x 的 watch 监听类型，判断是否为兼容模式。在有回调函数且不是深度遍历的情况下，执行 getter()函数，并对函数返回值进行处理，如果是数组则深度遍历，如果不是则直接返回。具体代码实现如下：

```
if (__COMPAT__ && cb && !deep) {
  const baseGetter = getter
  getter = () => {
```

```
    const val = baseGetter()
    if (
      isArray(val) &&
      checkCompatEnabled(DeprecationTypes.WATCH_ARRAY, instance)
    ) {
      traverse(val)
    }
    return val
  }
}
```

在有回调函数和深度遍历情况下,直接通过 traverse()函数执行深度遍历。具体代码实现如下:

```
if (cb && deep) {
  const baseGetter = getter
  getter = () => traverse(baseGetter())
}
```

上述判断完成后,初始化 wacth 的停止监听函数 onInvalidate()。onInvalidate()函数用于 watch 监听停止后,执行用户传递的回调函数,并且将该方法保存在 cleanup 对象内,以便后续执行。watch 函数监听也会保存在 cleanup 对象内,在元素进行修改时,会先调用 cleanup 进行清理。

```
let cleanup: () => void
let onInvalidate: InvalidateCbRegistrator = (fn: () => void) => {
  cleanup = runner.options.onStop = () => {
    callWithErrorHandling(fn, instance, ErrorCodes.WATCH_CLEANUP)
  }
}
```

判断是否为 SSR 环境,若是则立刻执行 getter()函数,注销监听,结束该函数的执行。

```
if (__SSR__ && isInSSRComponentSetup) {
  onInvalidate = NOOP
  if (!cb) {
    getter()
  } else if (immediate) {
    callWithAsyncErrorHandling(cb, instance, ErrorCodes.WATCH_CALLBACK, [
      getter(),
      isMultiSource ? [] : undefined,
      onInvalidate
    ])
  }
  return NOOP
}
```

接下来初始化异步队列需要执行的 job 回调函数,该函数将判断是否有用户传入的 cb 回调函数。若没有则直接执行 effect.run()方法,并结束该函数的执行;若有 cb 回调函数,则同样执行 effect.run()方法,将执行结果保存为新值(newValue)后开始处理旧值,判断是否深度监听、强制触发、多元数据、新值有变化,如果以上情况中有一种为是,则直接清理上一次执行队列,并执行 cb 回调函数,返回新值、旧值和 watch 监听停止回调函数 onInvalidate。具体代码实现如下:

```
let oldValue = isMultiSource ? [] : INITIAL_WATCHER_VALUE
const job: SchedulerJob = () => {
```

```
  if (!effect.active) {
    return
  }
  if (cb) {
    // watch(source, cb)
    const newValue = effect.run()
    if (
      deep ||
      forceTrigger ||
      (isMultiSource
        ? (newValue as any[]).some((v, i) =>
            hasChanged(v, (oldValue as any[])[i])
          )
        : hasChanged(newValue, oldValue)) ||
      (__COMPAT__ &&
        isArray(newValue) &&
        isCompatEnabled(DeprecationTypes.WATCH_ARRAY, instance))
    ) {
      // 再次运行 cb 前进行清理
      if (cleanup) {
        cleanup()
      }
      callWithAsyncErrorHandling(cb, instance, ErrorCodes.WATCH_CALLBACK, [
        newValue,
        // 旧值在第一次变化时若未定义,则忽略
        oldValue === INITIAL_WATCHER_VALUE ? undefined : oldValue,
        onInvalidate
      ])
      oldValue = newValue
    }
  } else {
    // watchEffect
    effect.run()
  }
}
```

5.3.3 scheduler 异步队列

异步执行队列声明后,开始创建 scheduler 异步队列,并根据 flush 的值决定是同步执行、异步执行,还是在组件挂载前执行一次。

```
job.allowRecurse = !!cb
scheduler: EffectScheduler
if (flush === 'sync') {
  scheduler = job as any
} else if (flush === 'post') {
  scheduler = () => queuePostRenderEffect(job, instance && instance.suspense)
} else {
  // default: 'pre'
  scheduler = () => {
    if (!instance || instance.isMounted) {
      queuePreFlushCb(job)
    } else {
      job()
    }
  }
}
```

完成该异步队列声明后,使用 ReactiveEffect()构造函数实例化出 effect 副作用函数,以便后续执行。传入 getter 函数和 scheduler()函数(副作用执行函数),代码实现如下:

```
const effect = new ReactiveEffect(getter, scheduler)
```

若有 cb 回调函数,且 immediate 参数值为 true,则调用 job()函数,若 immediate 参数值为 false,则执行 runner()函数并将函数返回值赋值给 oldValue;若无 cb 回调函数,且 flush 为 post,则使用 queuePostRenderEffect()函数异步执行 effect.run()方法。其他情况直接执行 job()方法。

```
if (cb) {
    if (immediate) {
      job()
    } else {
      oldValue = effect.run()
    }
  } else if (flush === 'post') {
    queuePostRenderEffect(effect.run.bind(effect), instance && instance.suspense)
  } else {
    job()
  }
```

调用 queuePostRenderEffect()函数判断上下文是否为 suspense 异步组件,若为 suspense 异步组件,则收集对应的 effect;若不是 suspense 异步组件,则直接调用异步队列处理。具体代码实现如下:

```
//$ runtime - core_render.ts
export const queuePostRenderEffect = __FEATURE_SUSPENSE__
? queueEffectWithSuspense
: queuePostFlushCb
```

上述方法根据是否为 suspense 异步组件调用 queueEffectWithSuspense()或调用 queuePostFlushCb()函数。

queueEffectWithSuspense()内部判断是否异步组件,若是,则在异步组件内收集 effect 副作用函数;若不是,则执行 queuePostFlushCb()函数。具体代码实现如下:

```
export function queueEffectWithSuspense(
  fn: Function | Function[],
  suspense: SuspenseBoundary | null
): void {
  if (suspense && suspense.pendingBranch) {
    if (isArray(fn)) {
      suspense.effects.push(...fn)
    } else {
      suspense.effects.push(fn)
    }
  } else {
    queuePostFlushCb(fn)
  }
}
```

完成执行后,返回停止监听函数。以上便是 watch 函数的具体内部实现。根据对代码的拆解可以看出,watch 函数内部其实也是调用的 effect 副作用函数收集对应依赖,处理数据监听,并在数据改变后,执行对应的 effect 副作用函数。

5.3.4 watchEffect()函数

完成 watch 函数的实现后,再来分析 watchEffect()函数,可以发现,和 watch 函数相比,

二者本质上只是传入 dowatch()函数的参数不同，在介绍 dowatch()函数内部实现时，已经对 watchEffect()函数进行了适配，根据实现传入参数不同，通过递归等方法注册 getter 函数，实现依赖收集。具体代码实现如下：

```
function watchEffect(
  effect: WatchEffect,
  options?: WatchOptionsBase
): WatchStopHandle {
  return doWatch(effect, null, options)
}
```

5.3.5 总结

watch 函数依赖响应式逻辑实现，内部根据传入参数调用 getter 函数进行依赖收集，并实时监听数据。在数据变化后及时派发更新，通知 effect 执行。通过对本节的学习，可更好地了解 watch 函数本身的特性以及 watch 和 watchEffect()函数的差异。

5.4 computed 函数

Vue3 中的 computed 函数与 Vue2 中的用法不同，Vue2 中在 data 内直接写 computed 方法即可，在 Vue3 中该方法已经被函数化。在使用时需要提供一个 get 和 set 方法，分别用于接收和传递响应式数据。

computed 函数的代码实现如下：

```
// reload 1
export function computed<T>(
  getter: ComputedGetter<T>,
  debugOptions?: DebuggerOptions
): ComputedRef<T>
// reload 2
export function computed<T>(
  options: WritableComputedOptions<T>,
  debugOptions?: DebuggerOptions
): WritableComputedRef<T>
// computed
export function computed<T>(
  getterOrOptions: ComputedGetter<T> | WritableComputedOptions<T>,
  debugOptions?: DebuggerOptions
) {
  let getter: ComputedGetter<T>
  let setter: ComputedSetter<T>
  const onlyGetter = isFunction(getterOrOptions)
  if (onlyGetter) {
    getter = getterOrOptions
    setter = __DEV__
      ? () => {
          console.warn('Write operation failed: computed value is readonly')
        }
      : NOOP
  } else {
    getter = getterOrOptions.get
    setter = getterOrOptions.set
  }
```

```
  const cRef = new ComputedRefImpl(getter, setter, onlyGetter || !setter)
  if (__DEV__ && debugOptions) {
    cRef.effect.onTrack = debugOptions.onTrack
    cRef.effect.onTrigger = debugOptions.onTrigger
  }
  return cRef as any
}
```

此处通过函数重载对多种传参进行处理,直接查看 computed()函数的实现。首先声明 getter 和 setter,判断 getterOrOptions()是否为函数,若为函数则将该函数赋值给 getter 变量保存,setter 变量设置为空,表示只读,不可修改;若不是则将传递的参数 getterOrOptions 中的 get 和 set 属性通过 getter 和 setter 变量保存,最后返回 ComputedRefImpl 实例。具体代码实现如下:

```
class ComputedRefImpl<T> {
  public dep?: Dep = undefined
  // 缓存当前值
  private _value!: T
  // 是否需要重新求值
  private _dirty = true
  public readonly effect: ReactiveEffect<T>
  public readonly __v_isRef = true;
  public readonly [ReactiveFlags.IS_READONLY]: boolean
  constructor(
    getter: ComputedGetter<T>,
    private readonly _setter: ComputedSetter<T>,
    isReadonly: boolean
  ) {
  // 创建 getter 副作用函数
    this.effect = new ReactiveEffect(getter, () => {
      if (!this._dirty) {
        this._dirty = true
        triggerRefValue(this)
      }
    })
    this[ReactiveFlags.IS_READONLY] = isReadonly
  }
// 执行 get
  get value() {
    const self = toRaw(this)
    trackRefValue(self)
    if (self._dirty) {
      self._dirty = false
      self._value = self.effect.run()!
    }
    return self._value
  }
// 执行 set
  set value(newValue: T) {
    this._setter(newValue)
  }
}
```

由上述代码可以看出整个 ComputedRefImpl()的执行情况。

5.4.1 创建 getter 副作用函数

在 computed 内部创建 ReactiveEffect 实例并传入 getter 副作用函数,其中第二个参数是

一个通过匿名函数传入 ReactiveEffect()内的定时调度函数。该定时调度函数是否执行,将依赖_dirty 属性值的状态。若为 true,则不执行内部逻辑;若为 false,则调用 triggerRefValue()触发 ref 值。

ReactiveEffect()内部主要涉及收集、触发、运行和停止的方法。在内部通过数组维护一个临时栈,若当前执行函数未在临时栈内,则执行入栈操作,在执行完成后通过 pop 执行出栈操作。

此处将会根据栈内数据的情况判断,如果超过 30 个栈,则直接清理;若少于 30 个栈则处理,并且在完成后调用 finally()函数,该方法内执行栈的删除和数据的重置工作,以便下一个对象使用。具体代码实现如下:

```
//$ reactivity_effect.ts
run() {
    if (!this.active) {
      return this.fn()
    }
    if (!effectStack.includes(this)) {
      try {
        effectStack.push((activeEffect = this))
        enableTracking()
        trackOpBit = 1 << ++effectTrackDepth
        if (effectTrackDepth <= maxMarkerBits) {
          initDepMarkers(this)
        } else {
          cleanupEffect(this)
        }
        return this.fn()
      } finally {
        if (effectTrackDepth <= maxMarkerBits) {
          finalizeDepMarkers(this)
        }
        trackOpBit = 1 << --effectTrackDepth
        resetTracking()
        effectStack.pop()
        const n = effectStack.length
        activeEffect = n > 0 ? effectStack[n - 1] : undefined
      }
    }
  }
```

在后面调用 run 方法时,将执行上述代码,调用副作用函数,触发数据的修改。run 方法的调用位置将会在 5.4.2 节中介绍。该步骤完成后,对 ReactiveEffect()构造函数内部有初步的认识。在 ComputedRefImpl 实例内创建了一个 getter 副作用函数,并且赋予了对应的方法,可在合适的时机执行该方法。完成 getter 副作用函数的创建后,应继续创建 ref。

5.4.2 创建 cRef

cRef 的创建主要关注 get 的实现即可。get 返回的是 computed 函数作用域下的 value。在 get 方法内部可以看到,get 内部主要将当前上下文的数据进行响应式处理,再调用 trackRefValue()函数,执行依赖收集,并且通过_dirty 属性值判断是否需要重新求值,当_dirty 为 true 时会执行 effect 副作用函数的 run 方法求新值,如果不需要则直接返回 value。

此处需要注意,_dirty 参数值为 true 时,将把_dirty 参数改为 false,并调用当前上下文的

run 方法。run 方法在初始化时创建,调度执行函数会根据!this._dirty 的值决定是否执行 triggerRefValue()函数进行重新求值。

关于触发新值更新的问题,需回到 5.4 节介绍的 run 方法的声明位置,在副作用函数的调度函数选项中传入如下匿名函数:

```
() => {
  if (!this._dirty) {
    this._dirty = true
    triggerRefValue(this)
  }
…
}
```

当第一次访问 computedRef.value 时会执行 run 方法。经过 5.4 节的解析可知,effect 返回的是一个包裹 getter 的副作用函数,执行 run 方法就会触发 getter 内部访问的响应式变量的依赖收集。当 getter 依赖的响应式数据发生变化时会触发 trigger,重新执行 run 方法。若传递了调度函数选项,则 run 方法以 scheduler 的形式调用。因此在 getter 依赖的响应式数据产生变化时会触发 run 方法执行更新。

继续查看 scheduler 的内部实现,发现其中并没有直接通过 getter 求值,而是在_dirty 为 false 时触发 computedRef 的 trigger,这意味着此时依赖于 computedRef 的副作用函数会重新执行,而在这个副作用函数中一定会对 computedRef 产生 get 访问,此时回到 get 函数内部会发现_drity 的值被设置为 true,执行 run 方法进行真实的求值。需要求新值的时刻发生在传入 computed 函数内的响应式数据发生改变的时候。

5.4.3 总结

computed 函数同样依赖响应式系统,该函数通过 getter 和 setter 方法实现,将当前上下文的数据进行响应式处理,再执行依赖收集和派发更新。当依赖的数据发生变化时会重新计算需返回的新值。

5.5 拓展方法

5.5.1 customRef()函数

前面介绍 reactive 和 ref 时,对数据响应式原理进行了分析和理解。在了解基本原理的基础上,本节继续介绍与 ref 类似的 customRef()函数。该函数可以更加灵活地实现响应式数据操作,并且可根据情况自行决定依赖收集和派发更新。具体代码实现如下:

```
customRef((track, trigger) => {
  return {
    get() {
      track();
      return value;
    },
    set(newValue) {
      value = newValue;
      trigger();
    },
  };
});
```

在上述代码中，customRef()将 track 和 trigger 函数以参数形式传入 get 和 set 方法，并可以在 get 和 set 方法内手动执行依赖收集和派发更新。下面详细介绍该函数的实现方式。

该函数的实现位于 $ reactivity_ref 文件内，函数实现如下：

```
export function customRef<T>(factory: CustomRefFactory<T>): Ref<T> {
return new CustomRefImpl(factory) as any
}
```

上述代码返回一个接收 factory 参数的 CustomRefImpl 实例。下面给出该实例的实现逻辑：

```
class CustomRefImpl<T> {
  private readonly _get: ReturnType<CustomRefFactory<T>>['get']
  private readonly _set: ReturnType<CustomRefFactory<T>>['set']
  public readonly __v_isRef = true
  constructor(factory: CustomRefFactory<T>) {
    const { get, set } = factory(
      () => trackRefValue(this),
      () => triggerRefValue(this),
    )
    this._get = get
    this._set = set
  }
  get value() {
    return this._get()
  }
  set value(newVal) {
    this._set(newVal)
  }
}
```

CustomRefImpl()构造函数内通过 factory 参数将 track 和 trigger 进行暴露，并且将用户自定义的 get 和 set 保存在私有变量_get 和_set 中。调用 get 和 set 方法时，执行_get 和_set 变量对应的方法，以达到重写 get 和 set 的目的，最终返回包装的 ref 对象。

经过简单的改造，可以更加灵活地通过 customRef()函数控制 track 和 trigger 的时机。

5.5.2 readonly()函数

围绕响应式数据的封装，本节将会继续介绍 readonly()函数的实现，该函数返回只读的响应式数据。使用 readonly()函数创建的内容将不支持修改。下面将具体介绍该函数的实现原理。

readonly()函数定义在 $ reactivity_reactive 文件内。具体代码实现如下：

```
export function readonly<T extends object>(
  target: T
): DeepReadonly<UnwrapNestedRefs<T>> {
  return createReactiveObject(
    target,
    true,
    readonlyHandlers,
    readonlyCollectionHandlers,
    readonlyMap
  )
}
```

由上述代码可知,readonly()函数内部返回 createReactiveObject()函数,该函数在 5.1.4 节已进行过介绍。该函数需传入 5 个参数,分别为 target、isReadonly、baseHandlers、collectionHandlers 和 readonlyMap。在该函数内,如果 isReadonly 状态为 true,则使用 readonlyMap 对象保存数据,并且通过 Proxy 对象对传入的代理函数进行代理,此处逻辑与 reactive 基本相同。拦截函数 readonlyHandlers()位于 $ reactivity_baseHandlrs 文件内,具体代码实现如下:

```
export const readonlyHandlers: ProxyHandler<object> = {
  get: readonlyGet,
  set(target, key) {
    if (__DEV__) {
      console.warn(
        `Set operation on key "${String(key)}" failed: target is readonly.`,
        target
      )
    }
    return true
  },
  deleteProperty(target, key) {
    if (__DEV__) {
      console.warn(
        `Delete operation on key "${String(key)}" failed: target is readonly.`,
        target
      )
    }
    return true
  }
}
```

由上述代码可知,对于只读对象,调用 get 方法将执行 readonlyGet()函数,调用 set 和 deleteProperty()函数将会在开发环境抛出警告,提示不能修改或删除。readonlyGet()函数内部调用 createGetter()函数时将传入 true 参数。该函数内部主要在只读情况下对数据类型进行判断,然后返回对应值,此处不展开讲解。readonly()函数和 reactive()函数最大的区别在于,readonly()函数不进行依赖收集,如果不是浅代理则递归返回数据;若为浅代理,则直接返回。

5.5.3 shallow()函数

shallow()函数表示浅代理,若遇到该状态将不会对数据递归处理,涉及的浅代理函数有 shallowRef()、shallowReactive()和 shallowReadonly()。通过浅代理声明的对象将仅代理第一层基础数据。

shallowRef()函数定义在 $ reactitity_ref 文件内,通过高阶函数形式返回 createRef()函数。具体代码实现如下:

```
export function shallowRef<T extends object>(
  value: T
): T extends Ref ? T : Ref<T>
export function shallowRef<T>(value: T): Ref<T>
export function shallowRef<T = any>(): Ref<T | undefined>
export function shallowRef(value?: unknown) {
  return createRef(value, true)
}
```

shallowRef()函数实现也涉及函数重载,根据传入参数不同调用不同的方法。该函数内

部对数据进行简单过滤后，通过 new 返回新实例，该实例的构造函数为 RefImpl()。具体代码实现如下：

```
class RefImpl<T> {
  private _value: T
  private _rawValue: T
  public dep?: Dep = undefined
  public readonly __v_isRef = true
  constructor(value: T, public readonly _shallow: boolean) {
    this._rawValue = _shallow ? value : toRaw(value)
    this._value = _shallow ? value : toReactive(value)
  }
  get value() {
    trackRefValue(this)
    return this._value
  }
  set value(newVal) {
    newVal = this._shallow ? newVal : toRaw(newVal)
    if (hasChanged(newVal, this._rawValue)) {
      this._rawValue = newVal
      this._value = this._shallow ? newVal : toReactive(newVal)
      triggerRefValue(this, newVal)
    }
  }
}
```

RefImpl()构造函数内部实现 get 和 set 方法，在 constructor 初始化阶段，若处于浅代理，在 get 时直接返回 value 值并做依赖收集。在 set 时直接返回传入的 value 值，派发更新，并且不对数据做深层次处理。

5.5.4 shallowReactive()函数

shallowReactive()函数的实现与 readonly()函数在同一位置，位于 $ reactivity_reactive 文件内。该函数同样返回 createReactiveObject()函数，并直接查看代理对象 shallowReactiveHandlers。具体代码实现如下：

```
// $ reactivity_bbaseHandlers.ts
export const shallowReactiveHandlers = /* #__PURE__ */ extend(
  {},
  mutableHandlers,
  {
    get: shallowGet,
    set: shallowSet
  }
)
```

通过 extend 方法合并 mutableHandlers、get 和 set 对象，mutableHandlers 对象的代码实现如下：

```
export const mutableHandlers: ProxyHandler<object> = {
  get,
  set,
  deleteProperty,
  has,
  ownKeys
}
```

mutableHandlers 对象代理 set、deleteProperty、has 和 ownKeys 对象，并且使用 shallowGet()和 shallowSet()函数重写 set 和 get，上述两个函数的定义内容如下：

```
const shallowGet = /* #__PURE__ */ createGetter(false, true)
const shallowSet = /* #__PURE__ */ createSetter(true)
```

5.5.5 shallowReadonly()函数

shallowReadonly()函数同样定义在 $reactivity.ts 文件内，该函数实现逻辑基本与 shallowReactive()函数类似。具体代码实现如下：

```
export function shallowReadonly<T extends object>(
  target: T
): Readonly<{ [K in keyof T]: UnwrapNestedRefs<T[K]> }> {
  return createReactiveObject(
    target,
    true,
    shallowReadonlyHandlers,
    readonlyCollectionHandlers,
    shallowReadonlyMap
  )
}
```

同样，返回 createReactiveObject()函数，并且传入代理对象 shallowReadonlyHandlers，该对象的代码实现如下：

```
export const shallowReadonlyHandlers = /* #__PURE__ */ extend(
  {},
  readonlyHandlers,
  {
    get: shallowReadonlyGet
  }
)
```

因为是浅代理并且是 readonly 状态，所以只需要代理 get 对象。

5.5.6 总结

通过对 shallow()函数、readonly()函数、customRef()函数的介绍可知，与 reactive 相关的函数实现均定义在 $reactitity_reactive 文件内，与 ref 相关的函数实现均定义在 $reactitity_ref 文件内。由上面的分析可知，与 ref 相关的内容均是调用 createRef()函数，与 reactive 相关的内容均是调用 createReactiveObject()函数，这两个函数内部实现了对 readonly 状态、shallow 状态的处理，在创建对应的方法时，可通过策略模式和高阶函数的方式减少重复代码。