Watch实现原理

watch 的核心就是观测一个响应式数据，当数据变化时通知并执行回调 （那也就是说它本身就是一个 effect）

1.监测响应式对象

```
watch(state, (oldValue, newValue) => {
  // 监测一个响应式值的变化
  console.log(oldValue, newValue);
});
```

```
function traverse(value, seen = new Set()) {
  if (!isObject(value)) {
    return value;
  }
  if (seen.has(value)) {
    return value;
  }
  seen.add(value);
  for (const k in value) {
    // 递归访问属性用于依赖收集
    traverse(value[k], seen);
  }
  return value;
}
export function isReactive(value) {
  return !!(value && value[ReactiveFlags.IS_REACTIVE]);
}
export function watch(source, cb) {
  let getter;
  if (isReactive(source)) {
    // 如果是响应式对象
    getter = () => traverse(source); // 包装成effect对应的fn, 函数内部进行遍历达到依赖收集的目的
  }
  let oldValue;
  const job = () => {
    const newValue = effect.run(); // 值变化时再次运行effect函数,获取新值
    cb(newValue, oldValue);
    oldValue = newValue;
```

```
  };
  const effect = new ReactiveEffect(getter, job); // 创建effect
  oldValue = effect.run(); // 运行保存老值
}
```

2.监测函数

```
export function watch(source, cb) {
  let getter;
  if (isReactive(source)) {
    // 如果是响应式对象
    getter = () => traverse(source);
  } else if (isFunction(source)) {
    getter = source; // 如果是函数则让函数作为fn即可
  }
  // ...
}
```

3.immediate实现

```
export function watch(source,cb,{immediate} = {} as any){
    const effect = new ReactiveEffect(getter,job) // 创建effect
    if(immediate){ // 需要立即执行，则立刻执行任务
        job();
    }
    oldValue = effect.run();
}
```

4.watch 中 cleanup 实现

> *连续触发 watch 时需要清理之前的 watch 操作*

```
const state = reactive({ flag: true, name: 'jw', age: 30 });
let timer = 3000
function getData(newVal) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(newVal);
    }, timer -= 1000);
  });
}
watch(state, async (newValue, oldValue, onCleanup) => {
  let clear = false;
  onCleanup(() => {
```

```
    clear = true;
  });
  let r = await getData(newValue.name);
  if (!clear) {
    document.body.innerHTML = r;
  }
  // 监测一个响应式值的变化
}, { immediate: true });
state.age = 31;
state.age = 32;
```

```
let cleanup;
let onCleanup = (fn) => {
  cleanup = fn;
};
const job = () => {
  const newValue = effect.run();
  if (cleanup) cleanup(); // 下次watch执行前调用上次注册的回调
  cb(newValue, oldValue, onCleanup); // 传入onCleanup函数
  oldValue = newValue;
};
```

5.watchEffect

> *我们可以使用响应性属性编写一个方法，**每当它们的任何值更新时**，我们的方法就会重新运行。**watchEffect在初始化时也会立即运行***

```
const state = reactive({ flag: true, name: 'jw', age: 30 });
watchEffect(() => app.innerHTML = state.name);
setTimeout(() => {
  state.name = 'Mr Jiang'
}, 1000)
```

```
export function watch(source, cb, options) {
  return doWatch(source, cb, options);
}
export function watchEffect(effect, options) {
  return doWatch(effect, null, options);
}
```

```
const job = () => {
  if (cb) {
    const newValue = effect.run(); // 值变化时再次运行effect函数,获取新值
    if (cleanup) cleanup(); // 下次watch执行前调用上次注册的回调
    cb(newValue, oldValue, onCleanup);
    oldValue = newValue;
  } else {
    effect.run(); // 重新执行effect即可
  }
};
```