Sunday • 更新于 2022-10-19

上一节 07：源码阅读：... 09：框架实现：... 下一节

08：框架实现：深入 scheduler 调度系统实现机制

经过了 `computed` 的代码和 `watch` 的代码之后，其实我们可以发现，在这两块代码中都包含了同样的一个概念那就是：**调度器** `scheduler` 。完整的来说，我们应该叫它：**调度系统**

整个调度系统其实包含两部分实现：

1. `lazy`： 懒执行
2. `scheduler`： 调度器

懒执行

懒执行相对比较简单，我们来看 `packages/reactivity/src/effect.ts` 中第 `183 - 185` 行的代码：

代码块

```
if (!options || !options.lazy) {
  _effect.run()
}
```

这段代码比较简单，其实就是如果存在 `options.lazy` 则 **不立即** 执行 `run` 函数。

我们可以直接对这段代码进行实现：

代码块

```
export interface ReactiveEffectOptions {
    lazy?: boolean
    scheduler?: EffectScheduler
}

/**
 * effect 函数
 * @param fn 执行方法
 * @returns 以 ReactiveEffect 实例为 this 的执行函数
 */
export function effect<T = any>(fn: () => T, options?: ReactiveEffectOptions) {
    // 生成 ReactiveEffect 实例
    const _effect = new ReactiveEffect(fn)
  // !options.lazy 时
    if (!options || !options.lazy) {
        // 执行 run 函数
        _effect.run()
    }
}
```

那么此时，我们就可以新建一个测试案例来测试下 `lazy` ，创建 `packages/vue/examples/reactivity/lazy.html` ：

代码块

```
<script>
  const { reactive, effect } = Vue

  const obj = reactive({
    count: 1
```

```
  // 调用 effect 方法
  effect(() => {
    console.log(obj.count);
  }, {
    lazy: true
  })

  obj.count = 2

  console.log('代码结束');

</script>
```

当不存在 `lazy` 时，打印结果为：

代码块

```
1
2
代码结束
```

当 `lazy` 为 `true` 时，因为不在触发 `run`，所以不会进行依赖收集，打印结果为：

代码块

```
代码结束
```

scheduler：调度器

调度器比懒执行要稍微复杂一些，整体的作用分成两块：

1. 控制执行顺序
2. 控制执行规则

控制执行顺序

我们先来看一个 `vue 3` 官网的例子，创建测试案例 `packages/vue/examples/imooc/scheduler.htm` ：

代码块

```
<script>
  const { reactive, effect } = Vue

  const obj = reactive({
    count: 1
  })

  // 调用 effect 方法
  effect(() => {
    console.log(obj.count);
  })

  obj.count = 2

  console.log('代码结束');

</script>
```

当前代码执行之后的打印顺序为：

代码块

```
1
2
代码结束
```

那么现在我们期望 **在不改变测试案例代码顺序的前提下** 修改一下代码的执行顺序，使其变为：

代码块

```
1
代码结束
2
```

那么想要达到这样的目的我们应该怎么做呢?

修改一下当前测试案例的代码:

代码块

```
// 调用 effect 方法
effect(() => {
  console.log(obj.count);
}, {
  scheduler() {
    setTimeout(() => {
      console.log(obj.count);
    })
  }
})
```

我们给 `effect` 传递了第二个参数 `options` , `options` 是一个对象，内部包含一个 `scheduler` 的选项，此时再次执行代码，得到 **期望** 的打印结果。

那么为什么会这样呢?

我们来回忆一下我们的代码，我们知道，目前在我们的代码中，执行 `scheduler` 的地方只有一个，那就是在 `packages/reactivity/src/effect.ts` 中:

代码块

```
/**
 * 触发指定的依赖
 */
export function triggerEffect(effect: ReactiveEffect) {
    if (effect.scheduler) {
        effect.scheduler()
    } else {
        effect.run()
    }
}
```

当 `effect` 存在 `scheduler` 时，我们会执行该调度器，而不是直接执行 `run` ，所以我们就可以利用 这个特性，在 `scheduler` 函数中执行我们期望的代码逻辑。

接下来，我们也可以为我们的 `effect` 增加 `scheduler` ，以此来实现这个功能:

1. 在 `packages/reactivity/src/effect.ts` 中:

   代码块

   ```
   export function effect<T = any>(fn: () => T, options?: ReactiveEffectOptions) {
       // 生成 ReactiveEffect 实例
       const _effect = new ReactiveEffect(fn)

   +   // 存在 options，则合并配置对象
   +   if (options) {
   +       extend(_effect, options)
   +   }

       if (!options || !options.lazy) {
           // 执行 run 函数
           _effect.run()
       }
   }
   ```

2. 在 `packages/shared/src/index.ts` 中，增加 `extend` 函数:

代码块

```
/**
 * Object.assign
 */
export const extend = Object.assign
```

3. 创建测试案例 `packages/vue/examples/reactivity/scheduler.html`：

代码块

```
<script>
  const { reactive, effect } = Vue

  const obj = reactive({
    count: 1
  })

  // 调用 effect 方法
  effect(() => {
    console.log(obj.count);
  }, {
    scheduler() {
      setTimeout(() => {
        console.log(obj.count);
      })
    }
  })

  obj.count = 2

  console.log('代码结束');

</script>
```

最终，得到期望的执行顺序。

控制执行规则

说完了执行顺序，那么对于执行规则而言，指的又是什么意思呢?

同样我们来看下面的例子 `packages/vue/examples/imooc/scheduler-2.html` ：

代码块

```
<script>
  const { reactive, effect } = Vue

  const obj = reactive({
    count: 1
  })

  // 调用 effect 方法
  effect(() => {
    console.log(obj.count)
  })

  obj.count = 2
  obj.count = 3

</script>
```

运行当前测试实例，得出打印结果：

代码块

```
1
2
3
```

但是我们知道，对于当前代码而言，最终的执行结果是必然为 3 的，那么我们可以不可以 **跳过** 中间的

那么想要达到这个目的，我们可以按照以下的流程去做：

1. 在 `packages/runtime-core/src/index.ts` 中，为 `./scheduler` 新增一个导出方法：

```
代码块
1     export {
2       nextTick,
3     +  queuePreFlushCb
4     } from './scheduler'
```

2. 在测试实例中，使用 `queuePreFlushCb` 配合 `scheduler`：

```
代码块
1       // 调用 effect 方法
2       effect(() => {
3         console.log(obj.count)
4       }, {
5     +     scheduler() {
6     +       queuePreFlushCb(() => { console.log(obj.count) })
7     +     }
8       })
```

3. 得到打印结果为：

```
代码块
1     1
2     3 // 打印两次
```

那么为什么会这样呢？ `queuePreFlushCb` 又做了什么？

在 **第七小节：watch 的源码阅读** 中，我们知道在 `packages/runtime-core/src/apiWatch.ts` 中 第 348 行：

```
代码块
1     scheduler = () => queuePreFlushCb(job)
```

通过 `queuePreFlushCb` 方法，构建了 `scheduler` 调度器。而根据源码我们知道 `queuePreFlushCb` 方法，最终会触发（这里不再详细讲解源码执行流程，忘记的同学可以看一下 **第七小节：watch 的源码阅读**）：

```
代码块
1     resolvedPromise.then(flushJobs)
```

那么根据以上逻辑，我们也可以实现对应的代码：

1. 创建 `packages/runtime-core/src/scheduler.ts` ：

```
代码块
1   // 对应 promise 的 pending 状态
2   let isFlushPending = false
3
4   /**
5    * promise.resolve()
6    */
7   const resolvedPromise = Promise.resolve() as Promise<any>
8   /**
9    * 当前的执行任务
10   */
11  let currentFlushPromise: Promise<void> | null = null
12
13  /**
14   * 待执行的任务队列
15   */
16  const pendingPreFlushCbs: Function[] = []
```

```
 * 队列预处理函数
 */
export function queuePreFlushCb(cb: Function) {
    queueCb(cb, pendingPreFlushCbs)
}

/**
 * 队列处理函数
 */
function queueCb(cb: Function, pendingQueue: Function[]) {
    // 将所有的回调函数，放入队列中
    pendingQueue.push(cb)
    queueFlush()
}

/**
 * 依次处理队列中执行函数
 */
function queueFlush() {
    if (!isFlushPending) {
        isFlushPending = true
        currentFlushPromise = resolvedPromise.then(flushJobs)
    }
}

/**
 * 处理队列
 */
function flushJobs() {
    isFlushPending = false
    flushPreFlushCbs()
}

/**
 * 依次处理队列中的任务
 */
export function flushPreFlushCbs() {
    if (pendingPreFlushCbs.length) {
        let activePreFlushCbs = [...new Set(pendingPreFlushCbs)]
        pendingPreFlushCbs.length = 0
        for (let i = 0; i < activePreFlushCbs.length; i++) {
            activePreFlushCbs[i]()
        }
    }
}
```

2. 创建 `packages/runtime-core/src/index.ts` ，导出 `queuePreFlushCb` 函数：

代码块

```
export { queuePreFlushCb } from './scheduler'
```

3. 在 `packages/vue/src/index.ts` 中，新增导出函数：

代码块

```
export { queuePreFlushCb } from '@vue/runtime-core'
```

4. 创建测试案例 `packages/vue/examples/reactivity/scheduler-2.html` ：

代码块

```
<script>
  const { reactive, effect, queuePreFlushCb } = Vue

  const obj = reactive({
    count: 1
  })

  // 调用 effect 方法
  effect(() => {
```

```
11      }, {
12        scheduler() {
13          queuePreFlushCb(() => { console.log(obj.count) })
14        }
15      })
16
17      obj.count = 2
18      obj.count = 3
19
20    </script>
```

5. 执行代码可得打印结果为：

代码块

```
1    1
2    3 // 打印两次
```

那么至此，我们就完成了调度器中两个比较重要的概念。

总结

懒执行相对比较简单，所以我们的总结主要针对调度器来说明。

调度器是一个相对比较复杂的概念，但是它本身并不具备控制 **执行顺序** 和 **执行规则** 的能力。

想要完成这两个能力，我们需要借助一些其他的东西来实现，这整个的一套系统，我们把它叫做 **调度系统**

那么到目前，我们调度系统的代码就已经实现完成了，这个代码可以在我们将来实现 `watch` 的时候直接使用。

07：源码阅读：响应性的数据监听器 watch... 上一节 下一节 09：框架实现：初步实现 watch 数据监听器

我要提出意见反馈