链滴

vue 思考什么是依赖收集，为什么要依赖收集？

作者：gmw-zjw
原文链接：https://ld246.com/article/1571125135432
来源网站：链滴

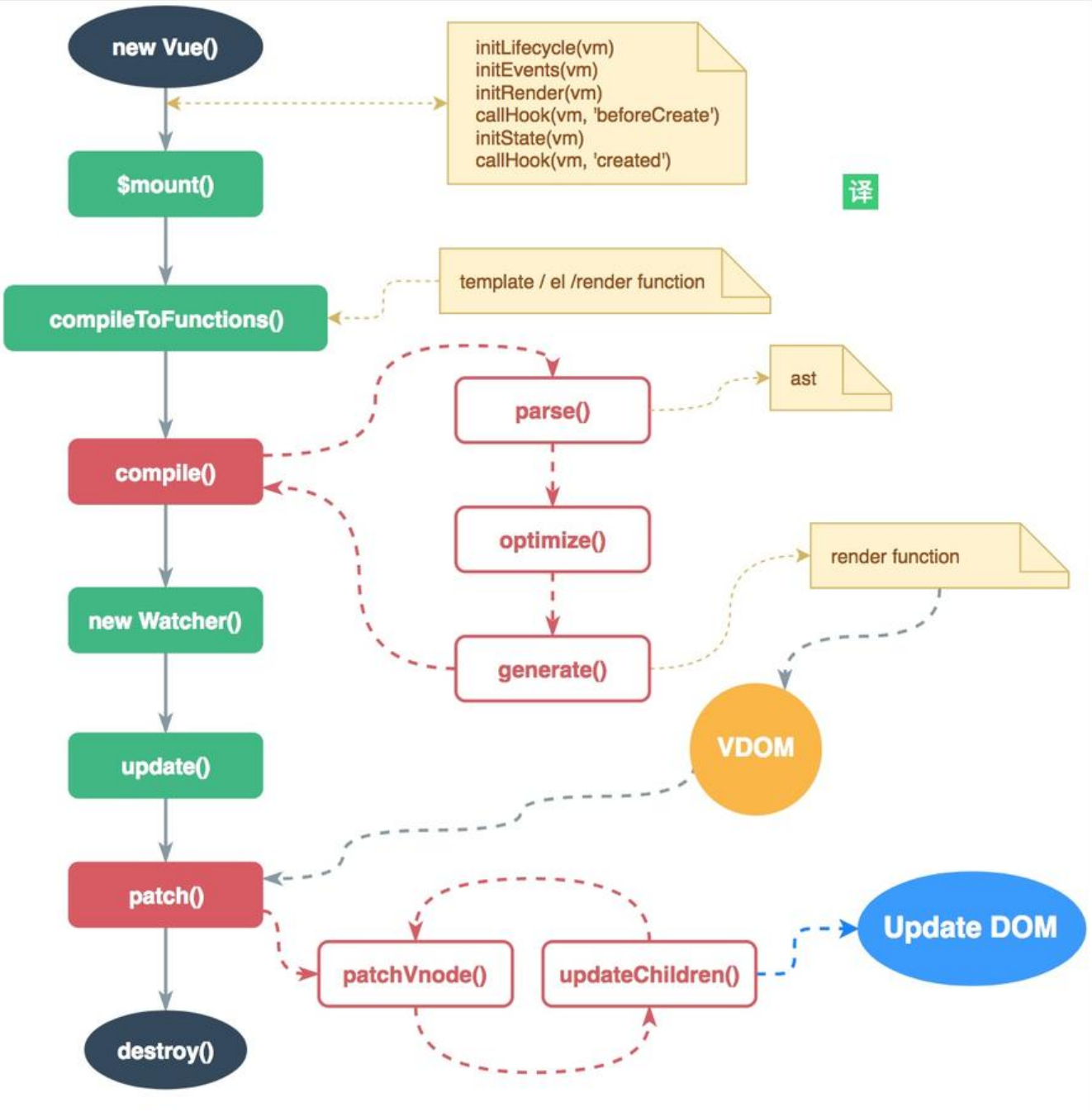
为什么要依赖收集，有如何进行依赖收集，举一个简单的□hestnut:

```
<template>
    <div>
    <span>test1: {{test1}}</span>
    <span>test2: {{test2}}</span>
    </div>
</template>

<script>
export default function() {
    name: 'demo',
    data() {
        return {
        test1: 'test1',
        test2: 'test2',
        test3: 'test3'
    }
    }
}
</script>
```

执行这行代码，test3显然没有使用，但是执行了，这样不好 影响性能，但是为什么会这样呢?

首先列一张vue执行流程图:



Vue 2.0 模板渲染过程

Dep

所谓的dep就是创建一个类，然后通过闭包的形式把data的值保存起来，在使用data中声明的值时会用getter，进而调用，不会对没有使用的加载，从而加快执行效率。

Dep源码：

```
export default class Dep() {
  static target: ?Watcher;
  id: number;
  subs: Array<Watcher>;
```

原文链接：vue 思考什么是依赖收集，为什么要依赖收集？

```
  constructor () {
    this.id = uid++
    this.subs = []
  }
    ...
  // 添加一个可观察的对象
   addSub(sub: Watcher) {
    this.subs.push(sub)
   }

  // 移除一个可观察对象
  remove() {/* 代码略 */}

   //依赖收集
   depend() {
      if (Dep.target) {
         Dep.target.addDep(this)
      }
   }

 // 通知所有订阅者
 notifiy() {
    const subs = this.subs.slice()
        if (process.env.NODE_ENV !== 'production' && !config.async) {
          // subs aren't sorted in scheduler if not running async
          // we need to sort them now to make sure they fire in correct
          // order
          subs.sort((a, b) => a.id - b.id)
        }
        for (let i = 0, l = subs.length; i < l; i++) {
          subs[i].update()
        }
     }
}

// 取消依赖收集
Dep.target = null
```

Watcher对象

```
export default class Weatcher() {
    // 省略非关键代码

// 获取getter的值并且重新进行依赖收集
  get () {
    /*将自身watcher观察者实例设置给Dep.target，用以依赖收集。*/
    pushTarget(this)
    let value
    const vm = this.vm
    try {
      value = this.getter.call(vm, vm)
    } catch (e) {
      /*
      执行了getter操作，看似执行了渲染操作，其实是执行了依赖收集。
```

```
      在将Dep.target设置为自身观察者实例以后，执行getter操作。
      譬如说现在的的data中可能有a、b、c三个数据，getter渲染需要依赖a跟c，
      那么在执行getter的时候就会触发a跟c两个数据的getter函数，
      在getter函数中即可判断Dep.target是否存在然后完成依赖收集，
      将该观察者对象放入闭包中的Dep的subs中去。
      */
      if (this.user) {
        handleError(e, vm, `getter for watcher "${this.expression}"`)
      } else {
        throw e
      }
    } finally {
      // "touch" every property so they are all tracked as
      // dependencies for deep watching
      /*如果存在deep，则触发每个深层对象的依赖，追踪其变化*/
      if (this.deep) {
        // 递归每一个数组或数组，触发他们的getter,使得对象或数组的每一个成员都被依赖收集，形
一个深的(deep)依赖关系
        traverse(value)
      }
      // 将观察者从实例target栈中取出，并设置给Dep.target
      popTarget()
      this.cleanupDeps()
    }
    return value
  }

  // 添加一个依赖收集关系到Deps集合中
   addDep (dep: Dep) {
     const id = dep.id
      if (!this.newDepIds.has(id)) {
       this.newDepIds.add(id)
       this.newDeps.push(dep)
       if (!this.depIds.has(id)) {
         dep.addSub(this)
      }
    }
  }

// 代码略...
}
```

进行依赖收集：

```
/**
 * Define a reactive property on an Object.
 */
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: ?Function,
```

```
  shallow?: boolean
) {
  // 在闭包中定义一个dep对象
  const dep = new Dep()

  const property = Object.getOwnPropertyDescriptor(obj, key)
  if (property && property.configurable === false) {
    return
  }

  // cater for pre-defined getter/setters
  /*
   * 如果之前该对象已经预设了getter以及setter函数则将其取出来，新定义的getter/setter中会将其
行，
   * 保证不会覆盖之前已经定义的getter/setter。
   */
  const getter = property && property.get
  const setter = property && property.set
  if ((!getter || setter) && arguments.length === 2) {
    val = obj[key]
  }

  /*对象的子对象递归进行observe并返回子节点的Observer对象*/
  let childOb = !shallow && observe(val)
  // 生成一个对象的新属性，将对象的子对象递归进行observe
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      // 如果原有对象上有getter方法 则执行
      const value = getter ? getter.call(obj) : val
      if (Dep.target) {
        // 进行依赖收集
        dep.depend()
        if (childOb) {
          /*子对象进行依赖收集，其实就是将同一个watcher观察者实例放进了两个depend中，
           * 一个是正在本身闭包中的depend，另一个是子元素的depend
          */
          childOb.dep.depend()
          if (Array.isArray(value)) {
            /*是数组则需要对每一个成员都进行依赖收集，如果数组的成员还是数组，则递归。*/
            dependArray(value)
          }
        }
      }
      return value
    },
    set: function reactiveSetter (newVal) {
      // 通过 getter 方法获取当前值，与新值进行比较，一致则不需要执行下面的操作
      const value = getter ? getter.call(obj) : val
      /* eslint-disable no-self-compare */
      if (newVal === value || (newVal !== newVal && value !== value)) {
        return
      }
```

```
      /* eslint-enable no-self-compare */
      if (process.env.NODE_ENV !== 'production' && customSetter) {
        customSetter()
      }
      // #7981: for accessor properties without setter
      if (getter && !setter) return
      if (setter) {
        // 如果原本对象setter拥有则直接执行setter
        setter.call(obj, newVal)
      } else {
        val = newVal
      }
      // 新的值重新进行observe，保证数据相应
      childOb = !shallow && observe(newVal)
      // dep对象通知所有观察者
      dep.notify()
    }
  })
}
```

将观察者Watcher实例赋值给全局的Dep.target，然后触发render操作只有被Dep.target标记过的才进行依赖收集。有Dep.target的对象会将Watcher的实例push到subs中，在对象被修改触发setter操的时候dep会调用subs中的Watcher实例的update方法进行渲染。