JavaScript | 复制代码

```javascript
const target = Object.defineProperties({}, {
  foo: {
    value: 123,
    writable: false,
    configurable: false
  },
});

const handler = {
  get(target, propKey) {
    return 'abc';
  }
};

const proxy = new Proxy(target, handler);

proxy.foo
// TypeError: Invariant check failed
```

10.2.5. set()

`set` 方法用来拦截某个属性的赋值操作，可以接受四个参数，依次为目标对象、属性名、属性值和 `Proxy` 实例本身

假定 `Person` 对象有一个 `age` 属性，该属性应该是一个不大于 200 的整数，那么可以使用 `Proxy` 保证 `age` 的属性值符合要求

JavaScript | 复制代码

```javascript
let validator = {
  set: function(obj, prop, value) {
    if (prop === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError('The age is not an integer');
      }
      if (value > 200) {
        throw new RangeError('The age seems invalid');
      }
    }

    // 对于满足条件的 age 属性以及其他属性，直接保存
    obj[prop] = value;
  }
};

let person = new Proxy({}, validator);

person.age = 100;

person.age // 100
person.age = 'young' // 报错
person.age = 300 // 报错
```

如果目标对象自身的某个属性，不可写且不可配置，那么 `set` 方法将不起作用

JavaScript | 复制代码

```javascript
const obj = {};
Object.defineProperty(obj, 'foo', {
  value: 'bar',
  writable: false,
});

const handler = {
  set: function(obj, prop, value, receiver) {
    obj[prop] = 'baz';
  }
};

const proxy = new Proxy(obj, handler);
proxy.foo = 'baz';
proxy.foo // "bar"
```

注意，严格模式下，`set` 代理如果没有返回 `true` ，就会报错

```JavaScript
'use strict';
const handler = {
  set: function(obj, prop, value, receiver) {
    obj[prop] = receiver;
    // 无论有没有下面这一行，都会报错
    return false;
  }
};
const proxy = new Proxy({}, handler);
proxy.foo = 'bar';
// TypeError: 'set' on proxy: trap returned falsish for property 'foo'
```

10.2.6. deleteProperty()

`deleteProperty` 方法用于拦截 `delete` 操作，如果这个方法抛出错误或者返回 `false` ，当前属性就无法被 `delete` 命令删除

```JavaScript
var handler = {
  deleteProperty (target, key) {
    invariant(key, 'delete');
    Reflect.deleteProperty(target,key)
    return true;
  }
};
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`无法删除私有属性`);
  }
}

var target = { _prop: 'foo' };
var proxy = new Proxy(target, handler);
delete proxy._prop
// Error: 无法删除私有属性
```

注意，目标对象自身的不可配置（configurable）的属性，不能被 `deleteProperty` 方法删除，否则报错

10.2.7. 取消代理

```
Plain Text | 复制代码
1  Proxy.revocable(target, handler);
```

10.3. 使用场景

`Proxy` 其功能非常类似于设计模式中的代理模式，常用功能如下：

- 拦截和监视外部对对象的访问
- 降低函数或类的复杂度
- 在复杂操作前对操作进行校验或对所需资源进行管理

使用 `Proxy` 保障数据类型的准确性

```
JavaScript | 复制代码
1  let numericDataStore = { count: 0, amount: 1234, total: 14 };
2  numericDataStore = new Proxy(numericDataStore, {
3      set(target, key, value, proxy) {
4          if (typeof value !== 'number') {
5              throw Error("属性只能是number类型");
6          }
7          return Reflect.set(target, key, value, proxy);
8      }
9  });
10
11 numericDataStore.count = "foo"
12 // Error: 属性只能是number类型
13
14 numericDataStore.count = 333
15 // 赋值成功
```

声明了一个私有的 `apiKey`，便于 `api` 这个对象内部的方法调用，但不希望从外部也能够访问 `api._apiKey`

JavaScript 复制代码

```
let api = {
    _apiKey: '123abc456def',
    getUsers: function(){ },
    getUser: function(userId){ },
    setUser: function(userId, config){ }
};
const RESTRICTED = ['_apiKey'];
api = new Proxy(api, {
    get(target, key, proxy) {
        if(RESTRICTED.indexOf(key) > -1) {
            throw Error(`${key} 不可访问.`);
        } return Reflect.get(target, key, proxy);
    },
    set(target, key, value, proxy) {
        if(RESTRICTED.indexOf(key) > -1) {
            throw Error(`${key} 不可修改`);
        } return Reflect.get(target, key, value, proxy);
    }
});

console.log(api._apiKey)
api._apiKey = '987654321'
// 上述都抛出错误
```

还能通过使用 `Proxy` 实现观察者模式

观察者模式（Observer mode）指的是函数自动观察数据对象，一旦对象有变化，函数就会自动执行

`observable` 函数返回一个原始对象的 `Proxy` 代理，拦截赋值操作，触发充当观察者的各个函数

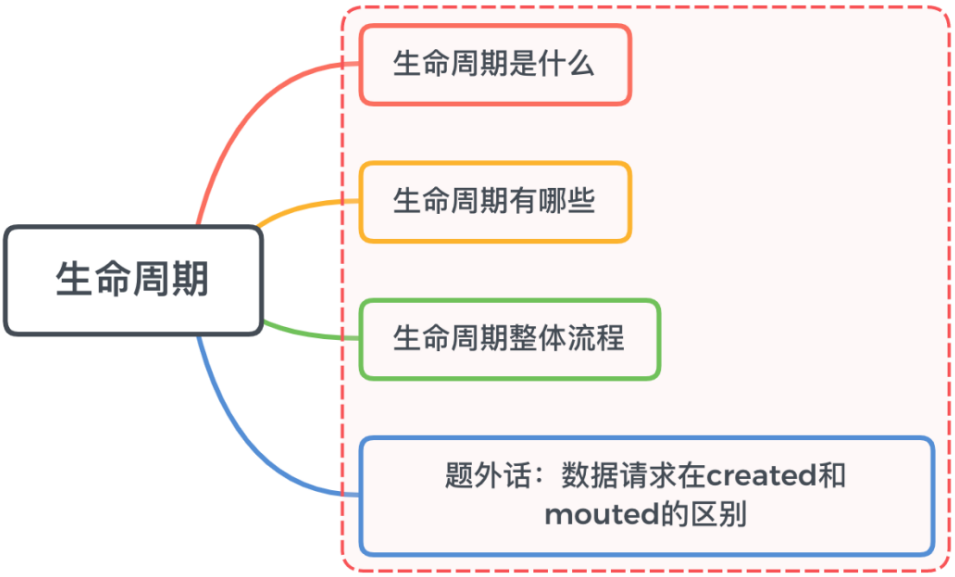
JavaScript 复制代码

```
const queuedObservers = new Set();

const observe = fn => queuedObservers.add(fn);
const observable = obj => new Proxy(obj, {set});

function set(target, key, value, receiver) {
  const result = Reflect.set(target, key, value, receiver);
  queuedObservers.forEach(observer => observer());
  return result;
}
```

观察者函数都放进 `Set` 集合，当修改 `obj` 的值，在会 `set` 函数中拦截，自动执行 `Set` 所有的观察者

Vue2面试真题（29题）

1. 请描述下对vue生命周期的理解



1.1. 生命周期是什么

生命周期 `（Life Cycle）` 的概念应用很广泛，特别是在政治、经济、环境、技术、社会等诸多领域经常出现，其基本涵义可以通俗地理解为“从摇篮到坟墓” `（Cradle-to-Grave）` 的整个过程
在 `Vue` 中实例从创建到销毁的过程就是生命周期，即指从创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、卸载等一系列过程我们可以把组件比喻成工厂里面的一条流水线，每个工人（生命周期）站在各自的岗位，当任务流转到工人身边的时候，工人就开始工作PS：在 `Vue` 生命周期钩子会自动绑定 `this` 上下文到实例中，因此你可以访问数据，对 `property` 和方法进行运算这意味着**你不能使用箭头函数来定义一个生命周期方法**（例如 `created: () => this.fetchTodos()` ）

1.2. 生命周期有哪些

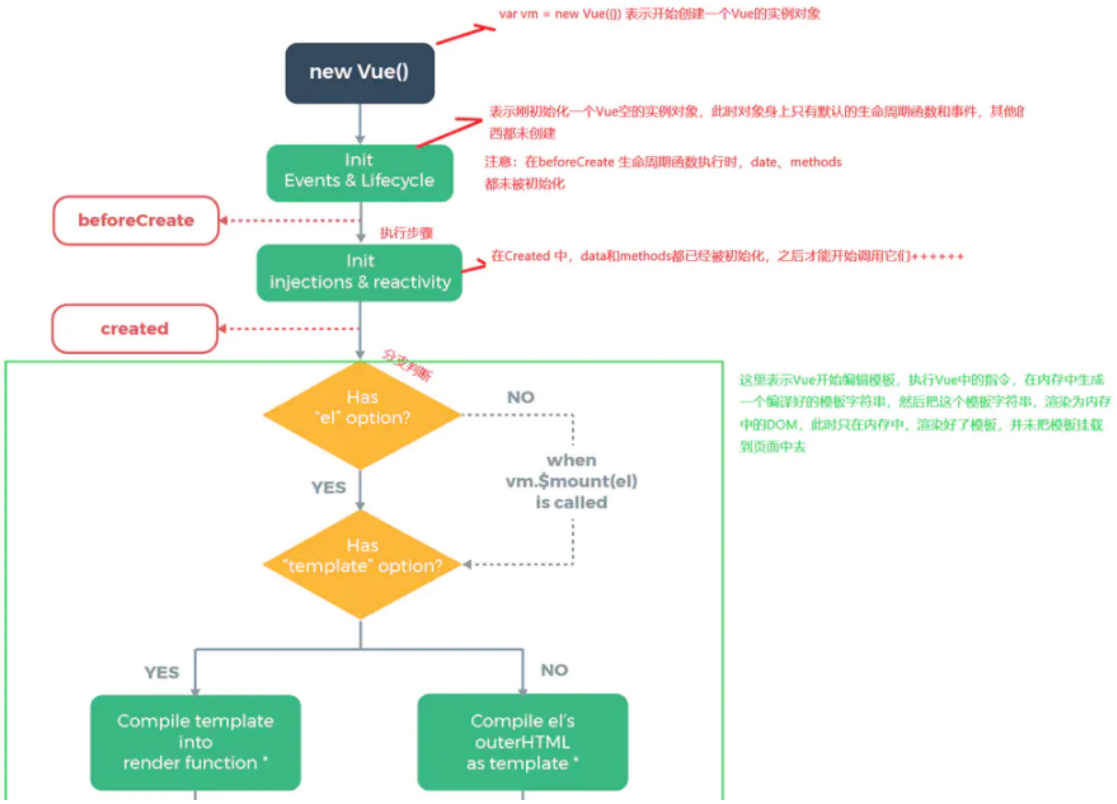
Vue生命周期总共可以分为8个阶段：创建前后，载入前后,更新前后,销毁前销毁后，以及一些特殊场景的生命周期

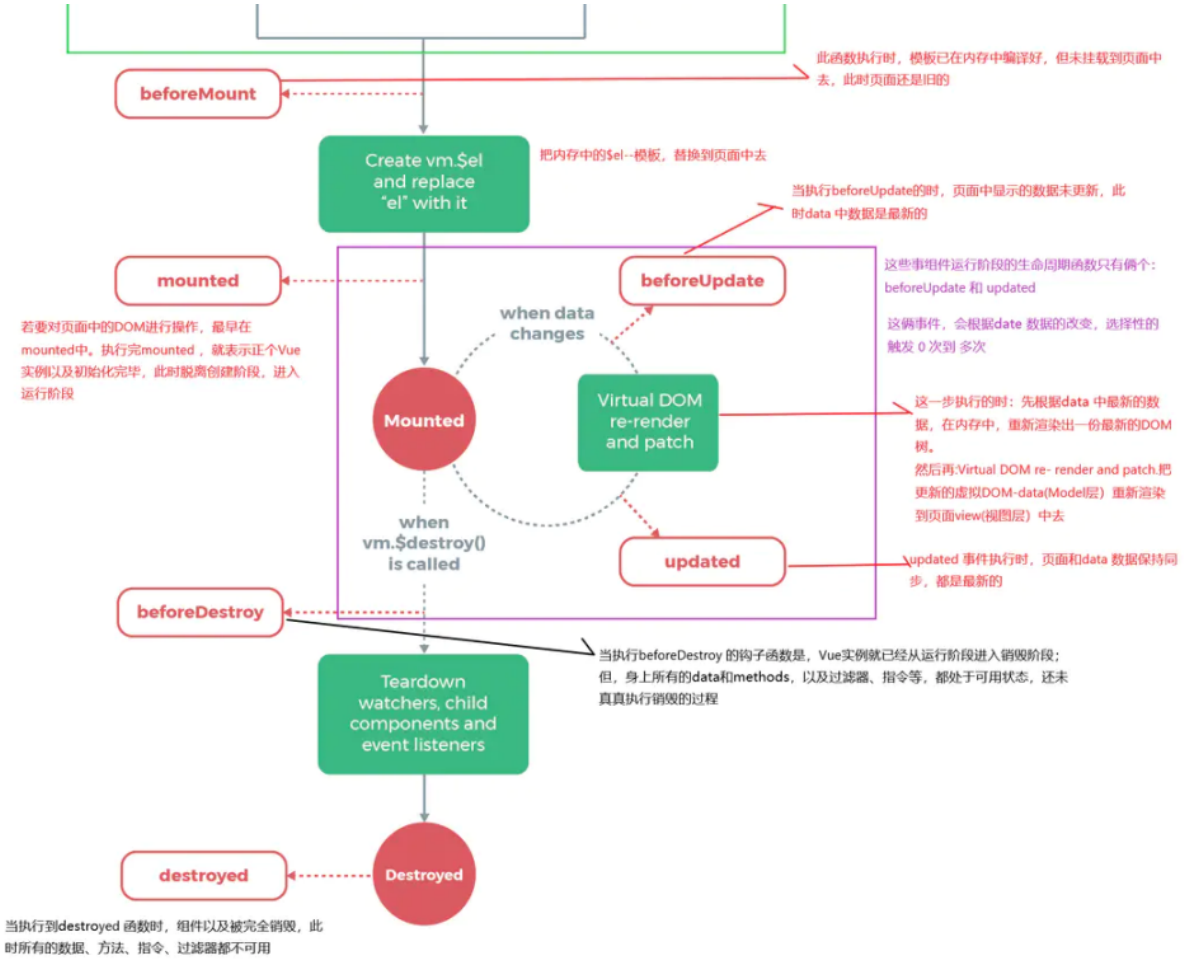
生命周期	描述

beforeCreate	组件实例被创建之初
created	组件实例已经完全创建
beforeMount	组件挂载之前
mounted	组件挂载到实例上去之后
beforeUpdate	组件数据发生变化，更新之前
updated	组件数据更新之后
beforeDestroy	组件实例销毁之前
destroyed	组件实例销毁之后
activated	keep-alive 缓存的组件激活时
deactivated	keep-alive 缓存的组件停用时调用
errorCaptured	捕获一个来自子孙组件的错误时被调用

1.3. 三、生命周期整体流程

`Vue` 生命周期流程图





1.3.1.1. 具体分析

beforeCreate –> created

- 初始化 `vue` 实例，进行数据观测

created

- 完成数据观测，属性与方法的运算， `watch` 、 `event` 事件回调的配置
- 可调用 `methods` 中的方法，访问和修改data数据触发响应式渲染 `dom` ，可通过 `computed` 和 `watch` 完成数据计算
- 此时 `vm.$el` 并没有被创建

created –> beforeMount

- 判断是否存在 `el` 选项，若不存在则停止编译，直到调用 `vm.$mount(el)` 才会继续编译
- 优先级： `render` > `template` > `outerHTML`

- `vm.el` 获取到的是挂载 `DOM` 的

beforeMount

- 在此阶段可获取到 `vm.el`
- 此阶段 `vm.el` 虽已完成DOM初始化，但并未挂载在 `el` 选项上

beforeMount –> mounted

- 此阶段 `vm.el` 完成挂载， `vm.$el` 生成的 `DOM` 替换了 `el` 选项所对应的 `DOM`

mounted

- `vm.el` 已完成 `DOM` 的挂载与渲染，此刻打印 `vm.$el` ，发现之前的挂载点及内容已被替换成新的DOM

beforeUpdate

- 更新的数据必须是被渲染在模板上的（ `el` 、 `template` 、 `render` 之一）
- 此时 `view` 层还未更新
- 若在 `beforeUpdate` 中再次修改数据，不会再次触发更新方法

updated

- 完成 `view` 层的更新
- 若在 `updated` 中再次修改数据，会再次触发更新方法（ `beforeUpdate` 、 `updated` ）

beforeDestroy

- 实例被销毁前调用，此时实例属性与方法仍可访问

destroyed

- 完全销毁一个实例。可清理它与其它实例的连接，解绑它的全部指令及事件监听器
- 并不能清除DOM，仅仅销毁实例

使用场景分析

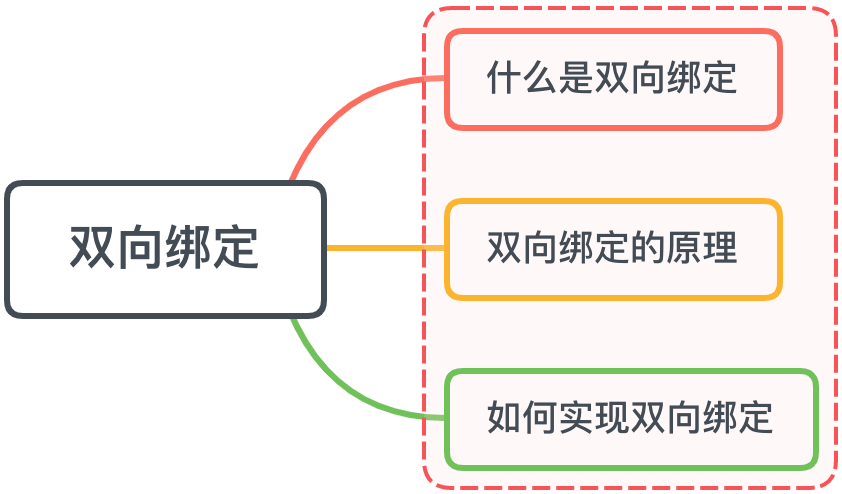
生命周期	描述
beforeCreate	执行时组件实例还未创建，通常用于插件开发中执行一些初始化任务
created	组件初始化完毕，各种数据可以使用，常用于异步数据获取
beforeMount	未执行渲染、更新，dom未创建

mounted	初始化结束，dom已创建，可用于获取访问数据和dom元素
beforeUpdate	更新前，可用于获取更新前各种状态
updated	更新后，所有状态已是最新
beforeDestroy	销毁前，可用于一些定时器或订阅的取消
destroyed	组件已销毁，作用同上

1.4. 数据请求在created和mouted的区别

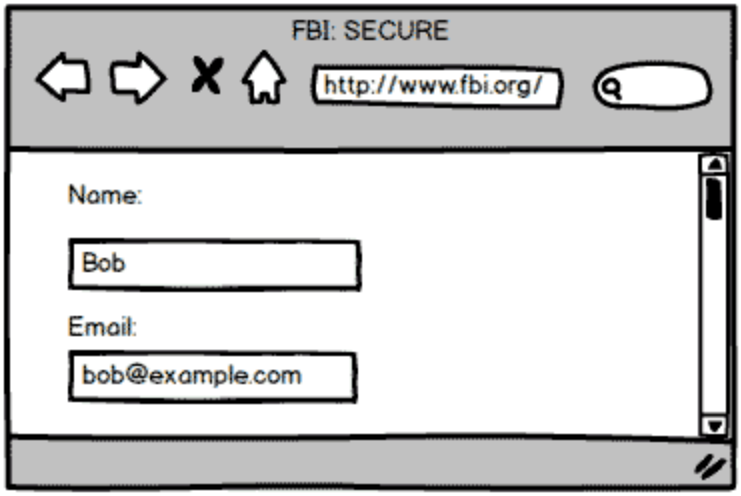
`created` 是在组件实例一旦创建完成的时候立刻调用，这时候页面 `dom` 节点并未生成； `mounted` 是在页面 `dom` 节点渲染完毕之后就立刻执行的。触发时机上 `created` 是比 `mounted` 要更早的，两者的相同点：都能拿到实例对象的属性和方法。
讨论这个问题本质就是触发的时机，放在 `mounted` 中的请求有可能导致页面闪动（因为此时页面 `dom` 结构已经生成），但如果在页面加载前完成请求，则不会出现此情况。建议对页面内容的改动放在 `created` 生命周期当中。

2. 双向数据绑定是什么

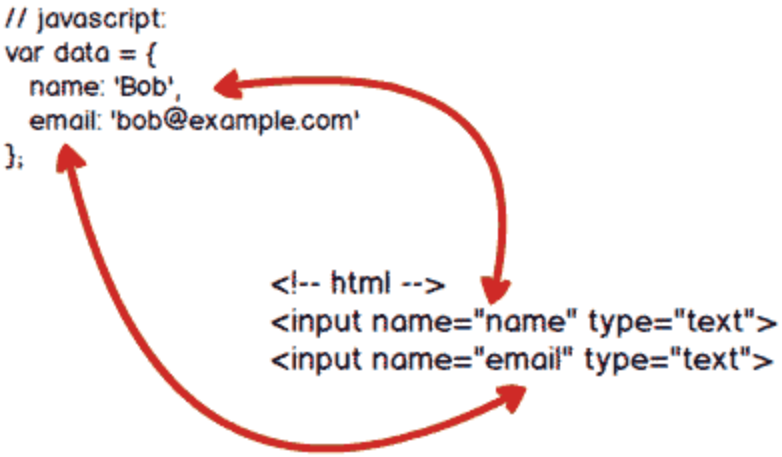


2.1. 什么是双向绑定

我们先从单向绑定切入单向绑定非常简单，就是把 `Model` 绑定到 `View` ，当我们用 `JavaScript` 代码更新 `Model` 时， `View` 就会自动更新双向绑定就很容易联想到了，在单向绑定的基础上，用户更新了 `View` ， `Model` 的数据也自动被更新了，这种情况就是双向绑定举个栗子



当用户填写表单时， `View` 的状态就被更新了，如果此时可以自动更新 `Model` 的状态，那就相当于我们把 `Model` 和 `View` 做了双向绑定关系图如下



2.2. 双向绑定的原理是什么

我们都知道 `Vue` 是数据双向绑定的框架，双向绑定由三个重要部分构成

- 数据层（Model）：应用的数据及业务逻辑
- 视图层（View）：应用的展示效果，各类UI组件
- 业务逻辑层（ViewModel）：框架封装的核心，它负责将数据与视图关联起来

而上面的这个分层的架构方案，可以用一个专业术语进行称呼： `MVVM` 这里的控制层的核心功能便是“数据双向绑定”。自然，我们只需弄懂它是什么，便可以进一步了解数据绑定的原理

2.2.1. 理解ViewModel

它的主要职责就是：

- 数据变化后更新视图
- 视图变化后更新数据

当然，它还有两个主要部分组成

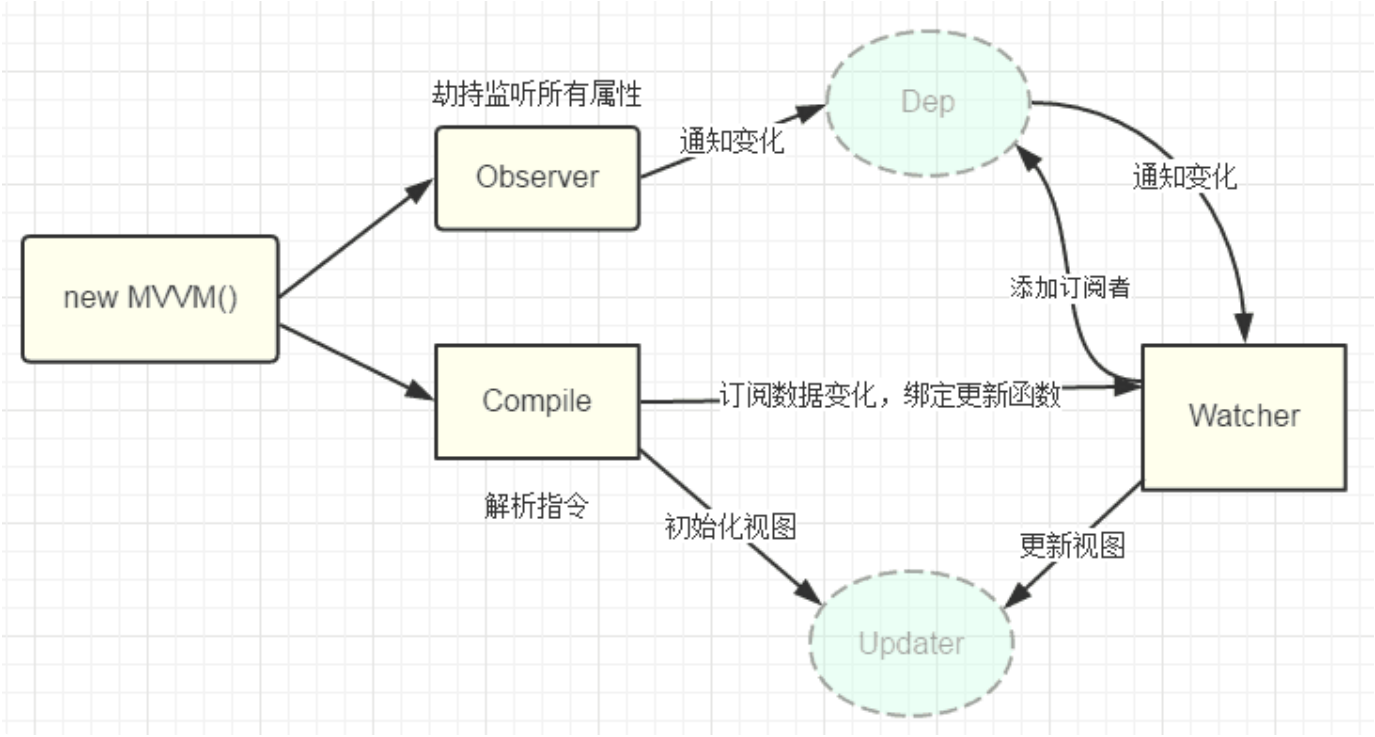
- 监听器（Observer）：对所有数据的属性进行监听
- 解析器（Compiler）：对每个元素节点的指令进行扫描跟解析,根据指令模板替换数据,以及绑定相应的更新函数

2.2.2. 实现双向绑定

我们还是以 `Vue` 为例，先来看看 `Vue` 中的双向绑定流程是什么的

1. `new Vue()` 首先执行初始化，对 `data` 执行响应化处理，这个过程发生 `Observe` 中
2. 同时对模板执行编译，找到其中动态绑定的数据，从 `data` 中获取并初始化视图，这个过程发生在 `Compile` 中
3. 同时定义一个更新函数和 `Watcher` ，将来对应数据变化时 `Watcher` 会调用更新函数
4. 由于 `data` 的某个 `key` 在一个视图中可能出现多次，所以每个 `key` 都需要一个管家 `Dep` 来管理多个 `Watcher`
5. 将来data中数据一旦发生变化，会首先找到对应的 `Dep` ，通知所有 `Watcher` 执行更新函数

流程图如下：



2.2.3. 实现

先来一个构造函数：执行初始化，对 `data` 执行响应化处理

JavaScript | 复制代码

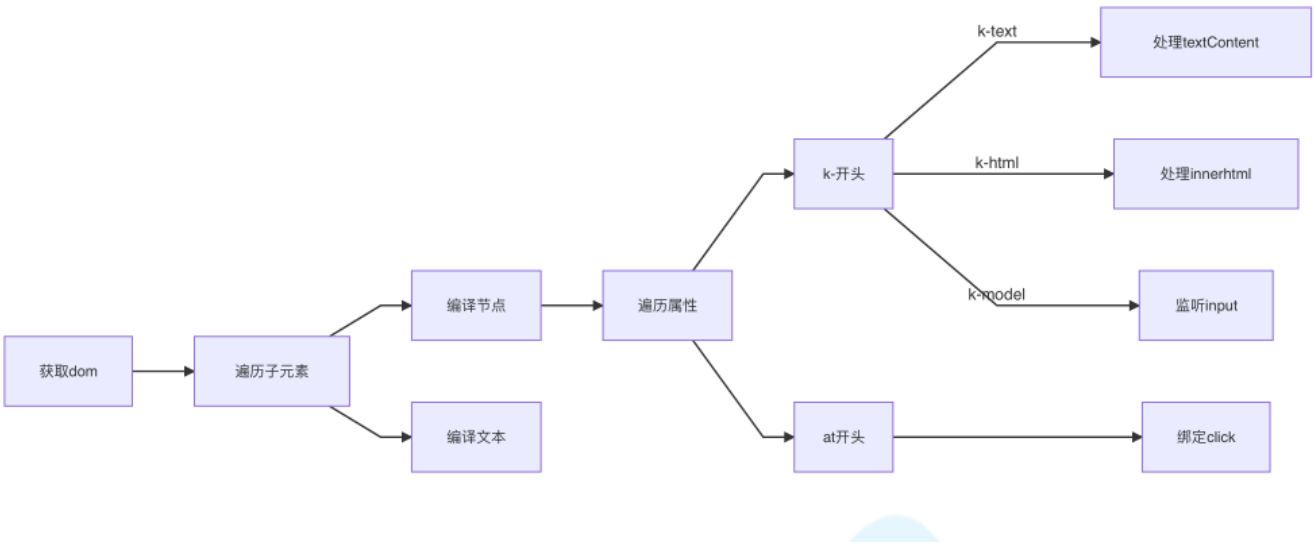
```
class Vue {
  constructor(options) {
    this.$options = options;
    this.$data = options.data;

    // 对data选项做响应式处理
    observe(this.$data);

    // 代理data到vm上
    proxy(this);

    // 执行编译
    new Compile(options.el, this);
  }
}
```

对 `data` 选项执行响应化具体操作

```javascript
function observe(obj) {
  if (typeof obj !== "object" || obj == null) {
    return;
  }
  new Observer(obj);
}

class Observer {
  constructor(value) {
    this.value = value;
    this.walk(value);
  }
  walk(obj) {
    Object.keys(obj).forEach((key) => {
      defineReactive(obj, key, obj[key]);
    });
  }
}
```

2.2.3.1. 编译 `Compile`

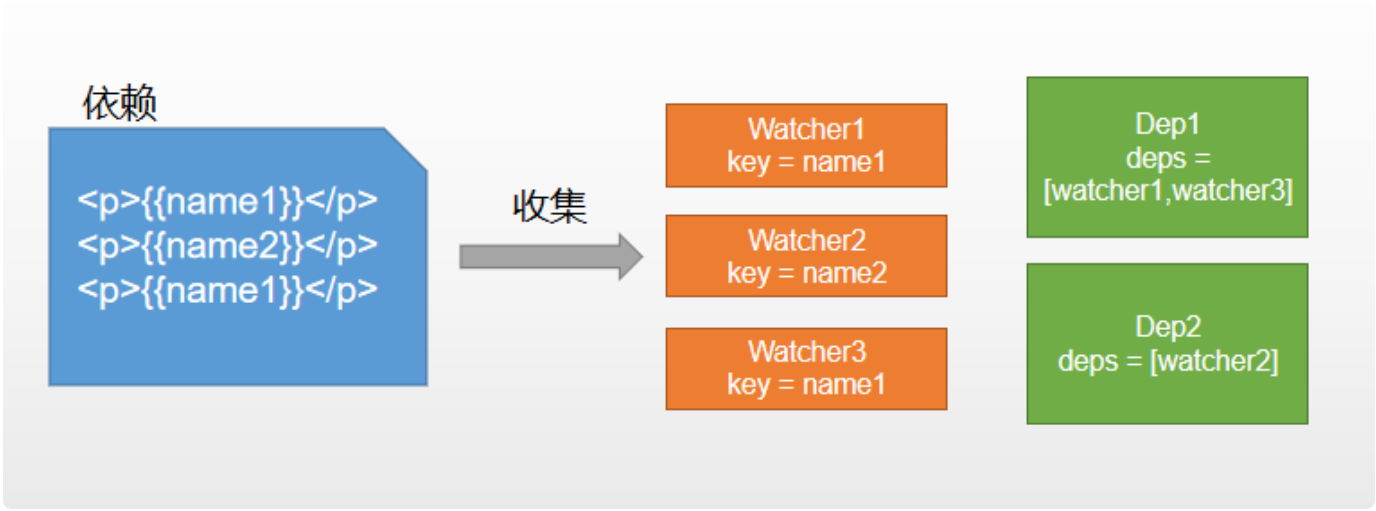
对每个元素节点的指令进行扫描跟解析,根据指令模板替换数据,以及绑定相应的更新函数



```javascript
class Compile {
  constructor(el, vm) {
    this.$vm = vm;
    this.$el = document.querySelector(el);  // 获取dom
    if (this.$el) {
      this.compile(this.$el);
    }
  }
  compile(el) {
    const childNodes = el.childNodes;
    Array.from(childNodes).forEach((node) => { // 遍历子元素
      if (this.isElement(node)) {   // 判断是否为节点
        console.log("编译元素" + node.nodeName);
      } else if (this.isInterpolation(node)) {
        console.log("编译插值文本" + node.textContent);  // 判断是否为插值文本{{}}
      }
      if (node.childNodes && node.childNodes.length > 0) {  // 判断是否有子元素
        this.compile(node);  // 对子元素进行递归遍历
      }
    });
  }
  isElement(node) {
    return node.nodeType == 1;
  }
  isInterpolation(node) {
    return node.nodeType == 3 && /\{\{(.*)\}\}/.test(node.textContent);
  }
}
```

2.2.3.2. 依赖收集

视图中会用到 `data` 中某 `key` ，这称为依赖。同一个 `key` 可能出现多次，每次都需要收集出来用一个 `Watcher` 来维护它们，此过程称为依赖收集多个 `Watcher` 需要一个 `Dep` 来管理，需要更新时由 `Dep` 统一通知



实现思路

1. `defineReactive` 时为每一个 `key` 创建一个 `Dep` 实例
2. 初始化视图时读取某个 `key`，例如 `name1`，创建一个 `watcher1`
3. 由于触发 `name1` 的 `getter` 方法，便将 `watcher1` 添加到 `name1` 对应的Dep中
4. 当 `name1` 更新，`setter` 触发时，便可通过对应 `Dep` 通知其管理所有 `Watcher` 更新

JavaScript | 复制代码

```
// 负责更新视图
class Watcher {
  constructor(vm, key, updater) {
    this.vm = vm
    this.key = key
    this.updaterFn = updater

    // 创建实例时，把当前实例指定到Dep.target静态属性上
    Dep.target = this
    // 读一下key，触发get
    vm[key]
    // 置空
    Dep.target = null
  }

  // 未来执行dom更新函数，由dep调用的
  update() {
    this.updaterFn.call(this.vm, this.vm[this.key])
  }
}
```

声明 `Dep`

```JavaScript
class Dep {
  constructor() {
    this.deps = [];  // 依赖管理
  }
  addDep(dep) {
    this.deps.push(dep);
  }
  notify() {
    this.deps.forEach((dep) => dep.update());
  }
}
```

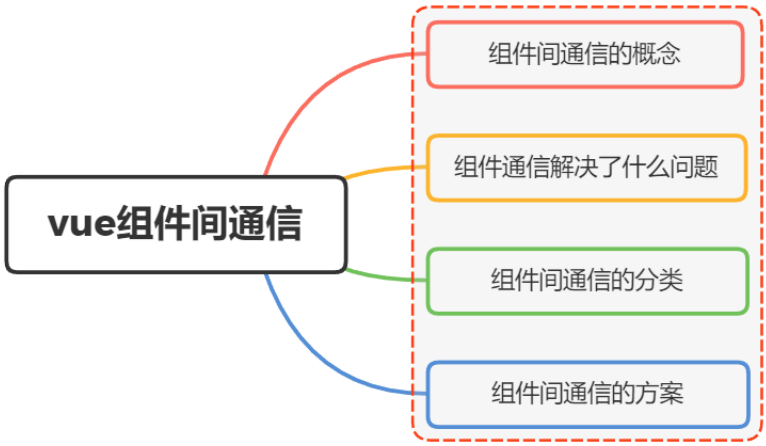
创建 `watcher` 时触发 `getter`

```JavaScript
class Watcher {
  constructor(vm, key, updateFn) {
    Dep.target = this;
    this.vm[this.key];
    Dep.target = null;
  }
}
```

依赖收集，创建 `Dep` 实例

```JavaScript
function defineReactive(obj, key, val) {
  this.observe(val);
  const dep = new Dep();
  Object.defineProperty(obj, key, {
    get() {
      Dep.target && dep.addDep(Dep.target);// Dep.target也就是Watcher实例
      return val;
    },
    set(newVal) {
      if (newVal === val) return;
      dep.notify(); // 通知dep执行更新方法
    },
  });
}
```

3. Vue组件之间的通信方式都有哪些?



3.1. 组件间通信的概念

开始之前，我们把**组件间通信**这个词进行拆分

- 组件
- 通信

都知道组件是 `vue` 最强大的功能之一， `vue` 中每一个 `.vue` 我们都可以视之为一个组件通信指的是发送者通过某种媒体以某种格式来传递信息到收信者以达到某个目的。广义上，任何信息的交通都是通信**组件间通信**即指组件(`.vue`)通过某种方式来传递信息以达到某个目的举个栗子我们在使用 `UI` 框架中的 `table` 组件，可能会往 `table` 组件中传入某些数据，这个本质就形成了组件之间的通信

3.2. 组件间通信解决了什么

在古代，人们通过驿站、飞鸽传书、烽火报警、符号、语言、眼神、触碰等方式进行信息传递，到了今天，随着科技水平的飞速发展，通信基本完全利用有线或无线电完成，相继出现了有线电话、固定电话、无线电话、手机、互联网甚至视频电话等各种通信方式从上面这段话，我们可以看到通信的本质是信息同步，共享回到 `vue` 中，每个组件之间的都有独自的作用域，组件间的数据是无法共享的但实际开发工作中我们常常需要让组件之间共享数据，这也是组件通信的目的要让它们互相之间能进行通讯，这样才能构成一个有机的完整系统

3.3. 组件间通信的分类