- ## 什么是Proxy及其作用

  - 概述：

    Proxy可以理解成，在目标对象之前架设一层 "拦截"，当外界对该对象进行操作的时候，都必须经过这层拦截，而Proxy就充当了这种机制，类似于代理的含义，它可以对外界访问对象之前进行过滤和改写该对象。

  - 引子：

    如果对vue2.xx了解或看过源码的人都知道，vue2.xx中使用 Object.defineProperty()方法对该对象通过 递归+遍历的方式来实现对数据的监控的，但这种方式是有缺点的。

    缺点如下：

    - 当我们使用数组的方法或改变数组的下标是不能重新触发 Object.defineProperty中的set()方法的，因此就做不到实时响应了。
    - 必须遍历每个对象的每个属性，才能进行监听，如果对象嵌套很深的话，需要使用递归调用。

    因此vue3.xx中之后就改用Proxy来更好的解决如上面的问题。

  - 基本语法：

```
const obj = new Proxy(target, handler);

/*
** 参数说明如下：
** target：被代理对象。
** handler：是一个对象，声明了代理target的一些操作。
** obj：是被代理完成之后返回的对象。
*/
```

当外界每次对obj进行操作时，就会执行handler对象上的一些方法。handler中常用的对象方法如下：

```
1. get(target, propKey, receiver)
2. set(target, propKey, value, receiver)
3. has(target, propKey)
4. construct(target, args):
5. apply(target, object, args)
```

  - 使用Proxy实现简单的Vue双向绑定

    步骤：

    - 需要实现一个数据监听器 Observer, 能够对所有数据进行监听，如果有数据变动的话，拿到最新的值并通知订阅者Watcher.
    - 需要实现一个指令解析器Compile，它能够对每个元素的指令进行扫描和解析，根

据指令模板替换数据，以及绑定相对应的函数。
    - 需要实现一个Watcher, 它是链接Observer和Compile的桥梁，它能够订阅并收到每个属性变动的通知，然后会执行指令绑定的相对应 的回调函数，从而更新视图。
- 实现Vue双向绑定的js部分

```
class Vue {
          constructor(options) {
            this.$el = document.querySelector(options.el);
            this.$methods = options.methods;
            this._binding = {};
            this._observer(options.data);
            this._compile(this.$el);
          }
          _pushWatcher(watcher) {
            if (!this._binding[watcher.key]) {
              this._binding[watcher.key] = [];
            }
            this._binding[watcher.key].push(watcher);
          }
          /*
           observer的作用是能够对所有的数据进行监听操作，通过使用Proxy对象中的set方法来监听，如有发生变动就会拿到最新值通知订阅者。
          */
          _observer(datas) {
            const me = this;
            const handler = {
              set(target, key, value) {
                const rets = Reflect.set(target, key, value);
                me._binding[key].map(item => {
                  item.update();
                });
                return rets;
              }
            };
            this.$data = new Proxy(datas, handler);
          }
          /*
           指令解析器，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相对应的更新函数
          */
          _compile(root) {
            const nodes =
Array.prototype.slice.call(root.children);
            const data = this.$data;
            nodes.map(node => {
              if (node.children && node.children.length) {
                this._compile(node.children);
              }
```

```
                const $input = node.tagName.toLocaleUpperCase() ===
"INPUT";
                const $textarea = node.tagName.toLocaleUpperCase()
=== "TEXTAREA";
                const $vmodel = node.hasAttribute('v-model');
                // 如果是input框 或 textarea 的话，并且带有 v-model 属性
的
                if (($vmodel && $input) || ($vmodel && $textarea)) {
                  const key = node.getAttribute('v-model');
                  this._pushWatcher(new Watcher(node, 'value', data,
key));
                  node.addEventListener('input', () => {
                    data[key] = node.value;
                  });
                }
                if (node.hasAttribute('v-bind')) {
                  const key = node.getAttribute('v-bind');
                  this._pushWatcher(new Watcher(node, 'innerHTML',
data, key));
                }
                if (node.hasAttribute('@click')) {
                  const methodName = node.getAttribute('@click');
                  const method =
this.$methods[methodName].bind(data);
                  node.addEventListener('click', method);
                }
              });
            }
          }
          /*
            watcher的作用是 链接Observer 和 Compile的桥梁，能够订阅并收到每
个属性变动的通知，
            执行指令绑定的响应的回调函数，从而更新视图。
          */
          class Watcher {
            constructor(node, attr, data, key) {
              this.node = node;
              this.attr = attr;
              this.data = data;
              this.key = key;
            }
            update() {
              this.node[this.attr] = this.data[this.key];
            }
          }
```