```
      sub.update()
    })
  }
}
// 全局属性，通过该属性配置 Watcher
Dep.target = null

function update(value) {
  document.querySelector( 'div').innerText = value
}

class Watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj [key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj [this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}
var data = { name: 'yck' }
observe(data)
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'
```

> 接下来,对 `defineReactive` 函数进行改造

```
function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
```

```
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log( 'get value')
      // 将 Watcher 添加到订阅
      if (Dep.target) {
        dp.addSub(Dep.target)
      }
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log( 'change value')
      val = newVal
      // 执行 watcher 的 update 方法
      dp.notify()
    }
  })
}
```

> 以上实现了一个简易的双向绑定，核心思路就是手动触发一次属性的 `getter` 来实现发布订阅的添加

Proxy 与 Object.defineProperty 对比

> `Object.defineProperty` 虽然已经能够实现双向绑定了，但是他还是有缺陷的。

- 只能对属性进行数据劫持，所以需要深度遍历整个对象 对于数组不能监听到数据的变化
- 虽然 `Vue` 中确实能检测到数组数据的变化，但是其实是使用了 `hack` 的办法， 并且也是有缺陷的。

```js
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// hack 以下几个函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
```

```
]
methodsToPatch.forEach(function (method) {
  // 获得原生函数
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    // 调用原生函数
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // 触发更新
    ob.dep.notify()
    return result
  })
})
```

> 反观 Proxy 就没以上的问题，原生支持监听数组变化， 并且可以直接对整个对象进行拦截，所以 Vue 也将在下个大版本中使用 Proxy 替换 Object.defineProperty

js

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver);
    },
    set(target, property, value, receiver) {
      setBind(value);
      return Reflect.set(target, property, value);
    }
  };
  return new Proxy(obj, handler);
};

let obj = { a: 1 }
```

```
let value
let p = onWatch(obj, (v) => {
  value = v
}, (target, property) => {
  console.log(`Get '${property} ' = ${target[property]}`);
})
p.a = 2 // bind `value` to `2`
p.a // -> Get 'a' = 2
```

2 路由原理

> 前端路由实现起来其实很简单，本质就是监听 URL 的变化，然后匹配路由规则， 显示相应的页面， 并且无须刷新 。 目前单页面使用的路由就只有两种实现方式

- hash 模式
- history 模式

> www.test.com/##/ 就是 Hash URL ， 当 ## 后面的哈希值发生变化时，不会向服务器请求数据， 可以通过 hashchange 事件来监听到 URL 的变化，从而进行跳转页面。

> History 模式是 HTML5 新推出的功能， 比之 Hash URL 更加美观

3 Virtual Dom

为什么需要 Virtual Dom

> 众所周知， 操作 DOM 是很耗费性能的一件事情， 既然如此， 我们可以考虑通过 JS 对象来模拟 DOM 对象， 毕竟操作 JS 对象比操作 DOM 省时的多

```js
// 假设这里模拟一个 ul， 其中包含了 5 个 li
[1, 2, 3, 4, 5]
// 这里替换上面的 li
[1, 2, 5, 4]
```

> 从上述例子中， 我们一眼就可以看出先前的 `ul` 中的第三个 `li` 被移除了，四五替换了位置。

- 如果以上操作对应到 DOM 中，那么就是以下代码

```js
// 删除第三个 li
ul.childNodes [2].remove()
// 将第四个 li 和第五个交换位置
let fromNode = ul.childNodes [4]
let toNode = node.childNodes [3]
let cloneFromNode = fromNode.cloneNode(true)
let cloenToNode = toNode.cloneNode(true)
ul.replaceChild(cloneFromNode, toNode)
ul.replaceChild(cloenToNode, fromNode)
```

> 当然在实际操作中， 我们还需要给每个节点一个标识，作为判断是同一个节点的依据 。所以这也是 Vue 和 React 中官方推荐列表里的节点使用唯一的 `key` 来保证性能。

- 那么既然 DOM 对象可以通过 JS 对象来模拟， 反之也可以通过 JS 对象来渲染出对应的 DOM
- 以下是一个 JS 对象模拟 DOM 对象的简单实现

```js
export default class Element {
  /**
   * @param {String} tag 'div'
   * @param {Object} props { class: 'item' }
   * @param {Array} children [ Element1, 'text']
   * @param {String} key option
   */
  constructor(tag, props, children, key) {
    this.tag = tag
    this.props = props
    if (Array.isArray(children)) {
```

```
      this.children = children
    } else if (isString(children)) {
      this.key = children
      this.children = null
    }
    if (key) this.key = key
  }
  // 渲染
  render() {
    let root = this._createElement(
      this.tag,
      this.props,
      this.children,
      this.key
    )
    document.body.appendChild(root)
    return root
  }
  create() {
    return this._createElement(this.tag, this.props, this.children, this.ke
  }
  // 创建节点
  _createElement(tag, props, child, key) {
    // 通过 tag 创建节点
    let el = document.createElement(tag)
    // 设置节点属性
    for (const key in props) {
      if (props.hasOwnProperty(key)) {
        const value = props[key]
        el.setAttribute(key, value)
      }
    }
    if (key) {
      el.setAttribute( 'key', key)
    }
    // 递归添加子节点
    if (child) {
      child.forEach(element => {
        let child
        if (element instanceof Element) {
          child = this._createElement(
            element.tag,
            element.props,
            element.children,
            element.key
          )
        } else {
```

```
            child = document.createTextNode(element)
          }
          el.appendChild(child)
        })
      }
      return el
    }
  }
```

Virtual Dom 算法简述

- 既然我们已经通过 JS 来模拟实现了 DOM ，那么接下来的难点就在于如何判断旧的对象和新的对象之间的差异。
- DOM 是多叉树的结构， 如果需要完整的对比两颗树的差异，那么需要的时间复杂度会是 O(n ^ 3) ， 这个复杂度肯定是不能接受的 。于是 React 团队优化了算法， 实现了 O(n) 的复杂度来对比差异。
- 实现 O(n) 复杂度的关键就是只对比同层的节点， 而不是跨层对比， 这也是考虑到在实际业务中很少会去跨层的移动 DOM 元素

> 所以判断差异的算法就分为了两步

- 首先从上至下， 从左往右遍历对象，也就是树的深度遍历， 这一步中会给每个节点添加索引，便于最后渲染差异
- 一旦节点有子元素，就去判断子元素是否有不同

Virtual Dom 算法实现

树的递归

- 首先我们来实现树的递归算法，在实现该算法前， 先来考虑下两个节点对比会有几种情况
- 新的节点的 tagName 或者 key 和旧的不同， 这种情况代表需要替换旧的节点， 并且也不再需要遍历新旧节点的子元素了， 因为整个旧节点都被删掉了
- 新的节点的 tagName 和 key （可能都没有）和旧的相同， 开始遍历子树
- 没有新的节点，那么什么都不用做

```js
import { StateEnums, isString, move } from './util'
import Element from './element'

export default function diff(oldDomTree, newDomTree) {
  // 用于记录差异
  let pathchs = {}
```

```
  // 一开始的索引为 0
  dfs(oldDomTree, newDomTree, 0, pathchs)
  return pathchs
}

function dfs(oldNode, newNode, index, patches) {
  // 用于保存子树的更改
  let curPatches = []
  // 需要判断三种情况
  // 1.没有新的节点，那么什么都不用做
  // 2.新的节点的 tagName 和 `key` 和旧的不同，就替换
  // 3.新的节点的 tagName 和 key（可能都没有） 和旧的相同， 开始遍历子树
  if ( !newNode) {
  } else if (newNode.tag === oldNode.tag && newNode.key === oldNode.key) {
    // 判断属性是否变更
    let props = diffProps(oldNode.props, newNode.props)
    if (props.length) curPatches.push({ type: StateEnums.ChangeProps, props
    // 遍历子树
    diffChildren(oldNode.children, newNode.children, index, patches)
  } else {
    // 节点不同，需要替换
    curPatches.push({ type: StateEnums.Replace, node: newNode })
  }

  if (curPatches.length) {
    if (patches [index]) {
      patches [index] = patches [index].concat(curPatches)
    } else {
      patches [index] = curPatches
    }
  }
}
```

判断属性的更改

> 判断属性的更改也分三个步骤

- 遍历旧的属性列表，查看每个属性是否还存在于新的属性列表中
- 遍历新的属性列表， 判断两个列表中都存在的属性的值是否有变化
- 在第二步中同时查看是否有属性不存在与旧的属性列列表中

```
function diffProps(oldProps, newProps) {
  // 判断 Props 分以下三步骤
```

```
  // 先遍历 oldProps 查看是否存在删除的属性
  // 然后遍历 newProps 查看是否有属性值被修改
  // 最后查看是否有属性新增
  let change = []
  for (const key in oldProps) {
    if (oldProps.hasOwnProperty(key) && !newProps[key]) {
      change.push({
        prop: key
      })
    }
  }
  for (const key in newProps) {
    if (newProps.hasOwnProperty(key)) {
      const prop = newProps[key]
      if (oldProps[key] && oldProps[key] !== newProps[key]) {
        change.push({
          prop: key,
          value: newProps[key]
        })
      } else if ( !oldProps[key]) {
        change.push({
          prop: key,
          value: newProps[key]
        })
      }
    }
  }
  return change
}
```

判断列表差异算法实现

> 这个算法是整个 Virtual Dom 中最核心的算法，且让我一一为你道来。 这里的主要步骤其实和判断属性差异是类似的，也是分为三步

- 遍历旧的节点列表，查看每个节点是否还存在于新的节点列表中
- 遍历新的节点列表， 判断是否有新的节点
- 在第二步中同时判断节点是否有移动

> PS：该算法只对有 key 的节点做处理

```js
function listDiff(oldList, newList, index, patches) {
  // 为了遍历方便，先取出两个 list 的所有 keys
  let oldKeys = getKeys(oldList)
  let newKeys = getKeys(newList)
  let changes = []

  // 用于保存变更后的节点数据
  // 使用该数组保存有以下好处
  // 1.可以正确获得被删除节点索引
  // 2.交换节点位置只需要操作一遍 DOM
  // 3.用于 `diffChildren` 函数中的判断， 只需要遍历
  // 两个树中都存在的节点，而对于新增或者删除的节点来说，完全没必要
  // 再去判断一遍
  let list = []
  oldList &&
    oldList.forEach(item => {
      let key = item.key
      if (isString(item)) {
        key = item
      }
      // 寻找新的 children 中是否含有当前节点
      // 没有的话需要删除
      let index = newKeys.indexOf(key)
      if (index === -1) {
        list.push(null)
      } else list.push(key)
    })
  // 遍历变更后的数组
  let length = list.length
  // 因为删除数组元素是会更改索引的
  // 所有从后往前删可以保证索引不变
  for (let i = length - 1; i >= 0; i--) {
    // 判断当前元素是否为空，为空表示需要删除
    if ( !list [i]) {
      list.splice(i, 1)
      changes.push({
        type: StateEnums.Remove,
        index: i
      })
    }
  }
  // 遍历新的 list，判断是否有节点新增或移动
  // 同时也对 `list` 做节点新增和移动节点的操作
  newList &&
    newList.forEach((item, i) => {
      let key = item.key
```

```
        if (isString(item)) {
          key = item
        }
        // 寻找旧的 children 中是否含有当前节点
        let index = list.indexOf(key)
        // 没找到代表新节点，需要插入
        if (index === -1 || key == null) {
          changes.push({
            type: StateEnums.Insert,
            node: item,
            index: i
          })
          list.splice(i, 0, key)
        } else {
          // 找到了，需要判断是否需要移动
          if (index !== i) {
            changes.push({
              type: StateEnums.Move,
              from: index,
              to: i
            })
            move(list, index, i)
          }
        }
      })
    return { changes, list }
}

function getKeys(list) {
    let keys = []
    let text
    list &&
      list.forEach(item => {
        let key
        if (isString(item)) {
          key = [item]
        } else if (item instanceof Element) {
          key = item.key
        }
        keys.push(key)
      })
    return keys
}
```

遍历子元素打标识

> 对于这个函数来说， 主要功能就两个

- 判断两个列表差异
  给节点打上标记
  总体来说，该函数实现的功能很简单

```js
function diffChildren(oldChild, newChild, index, patches) {
  let { changes, list } = listDiff(oldChild, newChild, index, patches)
  if (changes.length) {
    if (patches [index]) {
      patches [index] = patches [index].concat(changes)
    } else {
      patches [index] = changes
    }
  }
  // 记录上一个遍历过的节点
  let last = null
  oldChild &&
    oldChild.forEach((item, i) => {
      let child = item && item.children
      if (child) {
        index =
          last && last.children ? index + last.children.length + 1 : index
        let keyIndex = list.indexOf(item.key)
        let node = newChild [keyIndex]
        // 只遍历新旧中都存在的节点，其他新增或者删除的没必要遍历
        if (node) {
          dfs(item, node, index, patches)
        }
      } else index += 1
      last = item
    })
}
```

渲染差异

> 通过之前的算法， 我们已经可以得出两个树的差异了 。既然知道了差异，就需要局部去更新 DOM 了，下面就让我们来看看 Virtual Dom 算法的最后一步骤

这个函数主要两个功能

- 深度遍历树，将需要做变更操作的取出来
- 局部更新 DOM

```js
let index = 0
export default function patch(node, patchs) {
  let changes = patchs [index]
  let childNodes = node && node.childNodes
  // 这里的深度遍历和 diff 中是一样的
  if ( !childNodes) index += 1
  if (changes && changes.length && patchs [index]) {
    changeDom(node, changes)
  }
  let last = null
  if (childNodes && childNodes.length) {
    childNodes.forEach((item, i) => {
      index =
        last && last.children ? index + last.children.length + 1 : index +
      patch(item, patchs)
      last = item
    })
  }
}

function changeDom(node, changes, noChild) {
  changes &&
    changes.forEach(change => {
      let { type } = change
      switch (type) {
        case StateEnums.ChangeProps:
          let { props } = change
          props.forEach(item => {
            if (item.value) {
              node.setAttribute(item.prop, item.value)
            } else {
              node.removeAttribute(item.prop)
            }
          })
          break
        case StateEnums.Remove:
          node.childNodes [change.index].remove()
          break
        case StateEnums.Insert:
          let dom
          if (isString(change.node)) {
            dom = document.createTextNode(change.node)
          } else if (change.node instanceof Element) {
```

```
            dom = change.node.create()
          }
          node.insertBefore(dom, node.childNodes [change.index])
          break
        case StateEnums.Replace:
          node.parentNode.replaceChild(change.node.create(), node)
          break
        case StateEnums.Move:
          let fromNode = node.childNodes [change.from]
          let toNode = node.childNodes [change.to]
          let cloneFromNode = fromNode.cloneNode(true)
          let cloenToNode = toNode.cloneNode(true)
          node.replaceChild(cloneFromNode, toNode)
          node.replaceChild(cloenToNode, fromNode)
          break
        default:
          break
      }
    })
}
```

Virtual Dom 算法的实现也就是以下三步

- 通过 JS 来模拟创建 DOM 对象
- 判断两个对象的差异
- 渲染差异

```js
let test4 = new Element( 'div', { class: 'my-div' }, [ 'test4'])
let test5 = new Element( 'ul', { class: 'my-div' }, [ 'test5'])

let test1 = new Element( 'div', { class: 'my-div' }, [test4])

let test2 = new Element( 'div', { id: '11' }, [test5, test4])

let root = test1.render()

let pathchs = diff(test1, test2)
console.log(pathchs)

setTimeout(() => {
  console.log( '开始更新')
  patch(root, pathchs)
  console.log( '结束更新')
}, 1000)
```

第四部分： 计算机基础

一、网络

1 UDP

1.1 面向报文

> UDP 是一个面向报文（报文可以理解为一段段的数据）的协议。意思就是 UDP 只是报文的搬运工，不会对报文进行任何拆分和拼接操作

具体来说

- 在发送端，应用层将数据传递给传输层的 UDP 协议， UDP 只会给数据增加一个 UDP 头标识下是 UDP 协议，然后就传递给网络层了
- 在接收端， 网络层将数据传递给传输层， UDP 只去除 IP 报文头就传递给应用层，不会任何拼接操作

1.2 不可靠性

- UDP 是无连接的，也就是说通信不需要建立和断开连接。
- UDP 也是不可靠的。协议收到什么数据就传递什么数据， 并且也不会备份数据，对方能不能收到是不关心的
- UDP 没有拥塞控制，一直会以恒定的速度发送数据。即使网络条件不好，也不会对发送速率进行调整。这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 UDP 而不是 TCP

1.3 高效

- 因为 UDP 没有 TCP 那么复杂， 需要保证数据不丢失且有序到达。所以 UDP 的头部开销小， 只有八字节，相比 TCP 的至少二十字节要少得多，在传输数据报文时是很高效的

头部包含了以下几个数据

- 两个十六位的端口号，分别为源端口（可选字段）和目标端口 整个数据报文的长度

- 整个数据报文的检验和（ IPv4 可选 字段)， 该字段用于发现头部信息和数据中的错误

1.4 传输方式

> UDP 不止支持一对一的传输方式， 同样支持一对多， 多对多， 多对一的方式，也就是说 UDP 提供了单播， 多播，广播的功能

2 TCP

2.1 头部

> TCP 头部比 UDP 头部复杂的多

> 对于 TCP 头部来说， 以下几个字段是很重要的

- Sequence number ， 这个序号保证了 TCP 传输的报文都是有序的，对端可以通过序号顺序的拼接报文
- Acknowledgement Number ， 这个序号表示数据接收端期望接收的下一个字节的编号是多少， 同时也表示上一个序号的数据已经收到
- Window Size ， 窗口大小，表示还能接收多少字节的数据，用于流量控制

标识符

- URG=1 ： 该字段为一表示本数据报的数据部分包含紧急信息， 是一个高优先级数据报文，此时紧急指针有效 。紧急数据一定位于当前数据包数据部分的最前面， 紧急指针标明了紧急数据的尾部。
- ACK=1 ： 该字段为一表示确认号字段有效 。此外， TCP 还规定在连接建立后传送的所有报文段都必须把 ACK 置为一 PSH=1 ： 该字段为一表示接收端应该立即将数据 push 给应用层， 而不是等到缓冲区满后再提交。
- RST=1 ： 该字段为一表示当前 TCP 连接出现严重问题， 可能需要重新建立 TCP 连接，也可以用于拒绝非法的报文段和拒绝连接请求。
- SYN=1 ： 当 SYN=1 ， ACK=0 时，表示当前报文段是一个连接请求报文 。当 SYN=1 ， ACK=1 时，表示当前报文段是一个同意建立连接的应答报文 。
- FIN=1 ： 该字段为一表示此报文段是一个释放连接的请求报文

2.2 状态机

> HTTP 是无连接的，所以作为下层的 TCP 协议也是无连接的， 虽然看似 TCP 将两端连接了起来，但是其实只是两端共同维护了一个状态

- TCP 的状态机是很复杂的， 并且与建立断开连接时的握手息息相关，接下来就来详细描述下两种握手。
- 在这之前需要了解一个重要的性能指标 RTT。该指标表示发送端发送数据到接收到对端数据所需的往返时间

建立连接三次握手

- 在 TCP 协议中， 主动发起请求的一端为客户端，被动连接的一端称为服务端 。不管是客户端还是服务端， TCP 连接建立完后都能发送和接收数据，所以 TCP 也是一个全双工的协议。
- 起初，两端都为 CLOSED 状态 。在通信开始前， 双方都会创建 TCB 。 服务器创建完 TCB 后遍进入 LISTEN 状态， 此时开始等待客户端发送数据

第一次握手

> 客户端向服务端发送连接请求报文段 。该报文段中包含自身的数据通讯初始序号 。请求发送后，客户端便进入 SYN-SENT 状态，x 表示客户端的数据通信初始序号。

第二次握手

> 服务端收到连接请求报文段后， 如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 SYN-RECEIVED 状态。

第三次握手

> 当客户端收到连接同意的应答后， 还要向服务端发送一个确认报文 。客户端发完这个报文段后便进入 ESTABLISHED 状态， 服务端收到这个应答后也进入 ESTABLISHED 状态，此时连接建立成功。

- PS：第三次握手可以包含数据， 通过 TCP 快速打开（ TFO ）技术。其实只要涉及到握手的协议，都可以使用类似 TFO 的方式，客户端和服务端存储相同 cookie ，下次握手时发出 cookie 达到减少 RTT 的目的

你是否有疑惑明明两次握手就可以建立起连接，为什么还需要第三次应答?

- 因为这是为了防止失效的连接请求报文段被服务端接收，从而产生错误

> 可以想象如下场景。客户端发送了一个连接请求 A，但是因为网络原因造成了超时， 这时 TCP 会启动超时重传的机制再次发送一个连接请求 B。此时请求顺利到达服务端， 服务端应答完就建立了请求。如果连接请求 A 在两端关闭后终于抵达了服务端，那么这时服务端会认为客户端又需要建立 TCP 连接，从而应答了该请求并进入 ESTABLISHED 状态。此时客户端其实是 CLOSED 状态，那么就会导致服务端一直等待， 造成资源的浪费

> PS：在建立连接中，任意一端掉线，TCP 都会重发 SYN 包，一般会重试五次，在建立连接中可能会遇到 SYN FLOOD 攻击。遇到这种情况你可以选择调低重试次数或者干脆在不能处理的情况下拒绝请求

断开链接四次握手

> TCP 是全双工的，在断开连接时两端都需要发送 FIN 和 ACK 。

第一次握手

> 若客户端 A 认为数据发送完成，则它需要向服务端 B 发送连接释放请求。

第二次握手

> B 收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包， 并进入 CLOSE_WAIT 状态，表示 A 到 B 的连接已经释放，不接收 A 发的数据了。但是因为 TCP 连接时双向的，所以 B 仍旧可以发送数据给 A。

第三次握手

> B 如果此时还有没发完的数据会继续发送， 完毕后会向 A 发送连接释放请求，然后 B 便进入 LAST-ACK 状态。

> PS：通过延迟确认的技术（通常有时间限制， 否则对方会误认为需要重传），可以将第二次和第三次握手合并，延迟 ACK 包的发送。

第四次握手

- A 收到释放请求后， 向 B 发送确认应答，此时 A 进入 TIME-WAIT 状态 。该状态会持续 2MSL（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃） 时间，若该时间段内没有 B 的重发请求的话，就进入 CLOSED 状态 。当 B 收到确认应答后，也便进入 CLOSED 状态。

> 为什么 A 要进入 TIME-WAIT 状态， 等待 2MSL 时间后才进入 CLOSED 状态?

- 为了保证 B 能收到 A 的确认应答 。若 A 发完确认应答后直接进入 CLOSED 状态， 如果确认应答因为网络问题一直没有到达，那么会造成 B 不能正常关闭

3 HTTP

> `HTTP` 协议是个无状态协议，不会保存状态

3.1 Post 和 Get 的区别

- `Get` 请求能缓存， `Post` 不能
- `Post` 相对 `Get` 安全一点点， 因为 `Get` 请求都包含在 `URL` 里，且会被浏览器保存历史纪录， `Post` 不会，但是在抓包的情况下都是一样的。
- `Post` 可以通过 `request body` 来传输比 `Get` 更多的数据， `Get` 没有这个技术
- `URL` 有长度限制，会影响 `Get` 请求，但是这个长度限制是浏览器规定的，不是 `RFC` 规定的
- `Post` 支持更多的编码类型且不对数据类型限制

3.2 常见状态码

2XX 成功

- `200 OK` ，表示从客户端发来的请求在服务器端被正确处理

- 204 No content ，表示请求成功，但响应报文不含实体的主体部分
- 205 Reset Content ，表示请求成功，但响应报文不含实体的主体部分，但是与 204 响应不同在于要求请求方重置内容
- 206 Partial Content ， 进行范围请求

3XX 重定向

- 301 moved permanently ，永久性重定向，表示资源已被分配了新的 URL
- 302 found ， 临时性重定向，表示资源临时被分配了新的 URL
- 303 see other ，表示资源存在着另一个 URL，应使用 GET 方法丁香获取资源
- 304 not modified ，表示服务器允许访问资源，但因发生请求未满足条件的情况
- 307 temporary redirect ， 临时重定向，和302含义类似，但是期望客户端保持请求方法不变向新的地址发出请求

4XX 客户端错误

- 400 bad request ，请求报文存在语法错误
- 401 unauthorized ，表示发送的请求需要有通过 HTTP 认证的认证信息
- 403 forbidden ，表示对请求资源的访问被服务器拒绝
- 404 not found ，表示在服务器上没有找到请求的资源

5XX 服务器错误

- 500 internal sever error ，表示服务器端在执行请求时发生了错误
- 501 Not Implemented ，表示服务器不支持当前请求所需要的某个功能
- 503 service unavailable ，表明服务器暂时处于超负载或正在停机维护， 无法处理请求

3.3 HTTP 首部

通用字段	作用
Cache-Control	控制缓存的行为
Connection	浏览器想要优先使用的连接类型， 比如 keep-alive
Date	创建报文时间
Pragma	报文指令
Via	代理服务器相关信息
Transfer-Encoding	传输编码方式

通用字段	作用
Upgrade	要求客户端升级协议
Warning	在内容中可能存在错误

请求字段	作用
Accept	能正确接收的媒体类型
Accept-Charset	能正确接收的字符集
Accept-Encoding	能正确接收的编码格式列表
Accept-Language	能正确接收的语言列表
Expect	期待服务端的指定行为
From	请求方邮箱地址
Host	服务器的域名
If-Match	两端资源标记比较
If-Modified-Since	本地资源未修改返回 304（比较时间)
If-None-Match	本地资源未修改返回 304（比较标记)
User-Agent	客户端信息
Max-Forwards	限制可被代理及网关转发的次数
Proxy-Authorization	向代理服务器发送验证信息
Range	请求某个内容的一部分
Referer	表示浏览器所访问的前一个页面
TE	传输编码方式

响应字段	作用
Accept-Ranges	是否支持某些种类的范围
Age	资源在代理缓存中存在的时间
ETag	资源标识