# Vue项目目录和关键文件



# 组合式 API

Options API 的特点是在对应的属性中编写对应的功能模块

比如：在 methods 中定义方法，在 computed 中定义计算属性

缺点是：

实现某一个功能时，该功能对应的代码逻辑会被 拆分 到各个属性中

然而，当我们的组件变得更大、更复杂的时候，我们需要实现的功能也就越多。每一个功能对应的代码逻辑都被拆的很分散，代码碎片化这是不方便我们的去理解和维护这个组件的因此，在 Vue 组件中我们可以在 setup 中使用组合式 API，来解决上述问题。

注意：在 setup 中，我们应该避免使用 this。

## setup 组件选项

使用 setup 函数时，它将接受两个参数：

* props
* context

Props

对于 props 的相关内容，跟之前讲述的是一样的

// MyBook.vue

export default {

props: {

title: String

},

setup(props) {

console.log(props.title)

}

}

# **Context**

里面包含三个属性：

* attrs
* slots
* emit



结合模板使用



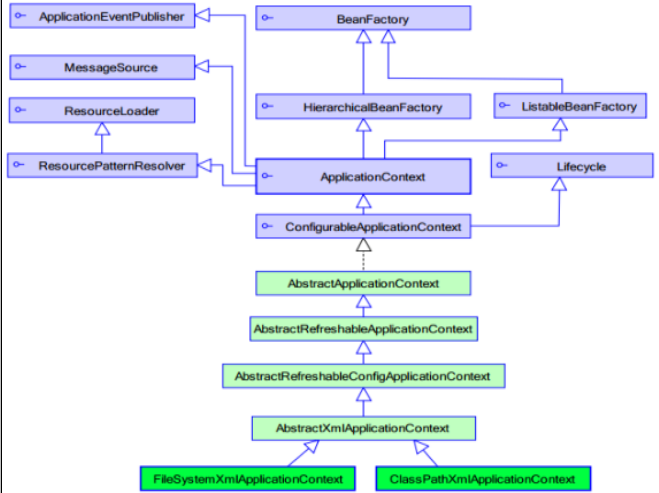
## ApplicationContext的继承体系

ApplicationContext是Spring框架的核心容器，用于管理Bean对象和它们之间的依赖关系。在ApplicationContext的继承体系中，它有三个主要的实现类：ClassPathXmlApplicationContext、FileSystemXmlApplicationContext和XmlWebApplicationContext。

其中，ClassPathXmlApplicationContext和FileSystemXmlApplicationContext都是基于XML配置文件的实现，用于依据类路径和文件系统路径获取XML配置文件中定义的Bean信息。而XmlWebApplicationContext则是基于Web应用程序的实现，可以自动搜索Web应用程序的WEB-INF路径下的XML配置文件，并将其中的Bean信息加载到容器中。

在这三个实现类之上，还可以有许多衍生类和扩展类，例如AnnotationConfigApplicationContext、GenericApplicationContext和SpringApplicationBuilder等，用于支持不同类型的Bean注册和依赖注入方式，并提供更便捷的配置和管理方法。除此之外，ApplicationContext还有多种实现方式，例如基于Java配置类，基于注解，基于Groovy DSL等。不同的实现方式可以满足不同开发场景的需求，同时它们都遵循同样的继承体系，提供了统一的API和管理方法。

applicationContext：接口类型，代表应用上下文，可以通过其实例获得 Spring 容器中的 Bean 对象



**4.2 ApplicationContext的实现类**

1）ClassPathXmlApplicationContext

它是从类的根路径下加载配置文件 推荐使用这种

2）FileSystemXmlApplicationContext

它是从磁盘路径上加载配置文件，配置文件可以在磁盘的任意位置。

3）AnnotationConfigApplicationContext

当使用注解配置容器对象时，需要使用此类来创建 spring 容器。它用来读取注解。

**4.3 getBean()方法使用**

**public** Object getBean(String name) **throws** BeansException {

assertBeanFactoryActive();

**return** getBeanFactory().getBean(name);

}

**public** <T> T getBean(Class<T> requiredType) **throws** BeansException {

assertBeanFactoryActive();

**return** getBeanFactory().getBean(requiredType);

}

其中，当参数的数据类型是字符串时，表示根据Bean的id从容器中获得Bean实例，返回是Object，需要强转。

当参数的数据类型是Class类型时，表示根据类型从容器中匹配Bean实例，当容器中相同类型的Bean有多个时，

则此方法会报错。

ApplicationContext applicationContext = **new**

ClassPathXmlApplicationContext(**"applicationContext.xml"**);

UserService userService1 = (UserService)

applicationContext.getBean(**"userService"**);

UserService userService2 = applicationContext.getBean(UserService.**class**);

# SPA 单页应用程序

单页面应用应用（即Single-page App，以下简称SPA），就是只有一张Web页面的应用。单页应用程序 (SPA) 是加载单个HTML 页面并在用户与应用程序交互时动态更新该页面的Web应用程序。浏览器一开始会加载必需的HTML、CSS和JavaScript，所有的操作都在这张页面上完成，都由JavaScript来控制。因此，对单页应用来说模块化的开发和设计显得相当重要。单页Web应用（single page web application，SPA），就是只有一张Web页面的应用。单页应用程序 (SPA) 是加载单个HTML 页面并在用户与应用程序交互时动态更新该页面的Web应用程序。浏览器一开始会加载必需的HTML、CSS和JavaScript，所有的操作都在这张页面上完成，都由JavaScript来控制。因此，对单页应用来说模块化的开发和设计显得相当重要。

以前，浏览器会收到来自服务器的HTML。当用户访问另一个URL地址时，需要全页刷新，服务器也会发送全新HTML。这就是所谓的服务器端渲染。

但是，在现代SPA中，客户端渲染已经取代了服务器端渲染。浏览器会先从服务器上加载出最初图像，和包括框架、库和应用代码在内的脚本，以及整个应用所需的样式表。当用户访问其他页面时，页面将不会进行整体刷新，而是通过HTML5 History API对页面地址进行更新。以JSON形式呈现出来的新页面所需的新数据，将会通过向服务器发出的AJAX请求，由浏览器检索进行检索。接着，SPA会通过原先在页面中下载好的JavaScript，来对页面数据进行动态更新。ssl证书申请

SPA的特点

* 速度：更好的用户体验，让用户在web app感受native app的速度和流畅，
* MVC：经典MVC开发模式，前后端各负其责。
* ajax：重前端，业务逻辑全部在本地操作，数据都需要通过AJAX同步、提交。

SPA的开发流程

用循环的视角审视Web应用开发框定一个一致的SPA图形用户界面（GUI）和模型将SPA的原则带回服务器端聚集于对合适的应用进行早期SPA开发

SPA的优点

1.应用针对用户操作给出的反应更加灵敏，不会由于页面整体刷新而出现闪退；

2.向服务器发送的HTTP请求减少，无需在每一页进行重复下载；

3.用户和服务器划分明确，无需修改服务器代码就可以轻松为新用户创建不同平台。另外，只要不违背API规则，还可以分别对用户和服务器的技术堆栈进行修改。

SPA的缺点

1.最开始的加载任务较重，包括框架和应用代码等；

2.需要对服务器进行额外的配置操作，让它将所有请求汇集到同一个进入点；

3.SPA依靠JavaScript来呈现内容，但并不是所有搜索引擎都能够在爬虫过程中执行JavaScript。这一点，无疑会对应用的搜索引擎优化带来负面影响。

# 响应式数据

Vue 的响应式，是指当数据改变后，Vue 会通知到使用该数据的代码；例如，视图渲染中使用了数据，数据改变后，视图也会自动更新。

其核心机制是 观察者模式，我们把依赖数据的观察者称为 watcher；

数据可以有多个观察者，怎么记录这种依赖关系呢？

Vue 通过在data 和 watcher 间创建一个 dep 对象，来记录这种依赖关系；

dep 的结构很简单，除了唯一标识属性id，另一个属性就是用于记录所有观察者的 subs：

id - number

subs - [Watcher]

1.语法：

Object.definProperty(obj,prop,descriptor)

2.参数：

* obj是要在其上定义属性的对象；
* prop要定义的或要修改的属性的名称；
* descriptor将被定义或修改的属性描述符；

3.数据描述：

* configurable：是否可以删除属性，默认false；
* enumberable：此属性是否可以被枚举，默认false；
* value：该属性对应的值，默认undefined；
* writable：属性的值是否可以被重写，默认false；

4.访问器描述：

* getter：是一种获得属性值的方法；
* setter：是一种设置属性值的方法；

## 数据变化的过程

侦测数据的变化 (数据劫持 / 数据代理)

收集视图依赖了哪些数据 (依赖收集)

数据变化时，自动“通知”需要更新的视图部分，并进行更新(发布订阅模式)

追踪数据

在 new Vue() 后， Vue 会调用 \_init 函数进行初始化，也就是init 过程，在 这个过程Data通过Observer转换成了getter/setter的形式，来对数据追踪变化，当被设置的对象被读取的时候会执行getter函数，而在当被赋值的时候会执行setter函数。也就是说，当创建Vue实例时，Vue会将这个实例的data对象转换成getter/setter形式，这对用户是不可见的。

所以就出现了无法进入响应式系统的数据，稍后介绍。

收集依赖

每个组件实例都对应一个 watcher 实例，它会在组件渲染的过程中把“接触”过的数据property记录为依赖。之后当依赖项的 setter触发时，会通知watcher，从而使它关联的组件重新渲染。

当外界通过Watcher读取数据时，会触发getter从而将Watcher添加到依赖中。

订阅者  
主要作用是用来存放 Watcher观察者对象

class Dep {

constructor () {

/\* 用来存放Watcher对象的数组 \*/

this.subs = [];

}

/\* 在subs中添加一个Watcher对象 \*/

addSub (sub) {

this.subs.push(sub);

}

/\* 通知所有Watcher对象更新视图 \*/

notify () {

this.subs.forEach((sub) => {

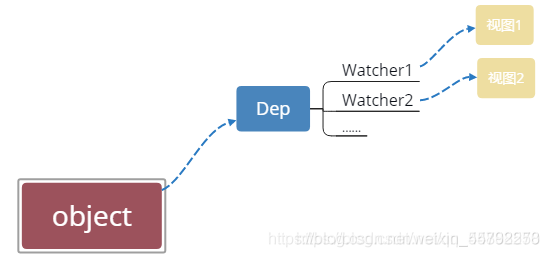
sub.update();

})

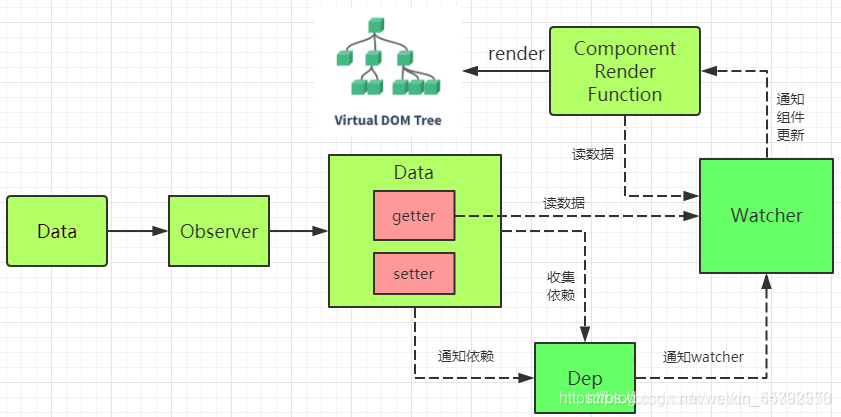
}

}

在修改对象的值的时候，会触发对应的setter， setter通知之前依赖收集得到的 Dep 中的每一个Watcher，告诉它们自己的值改变了，需要重新渲染视图。这时候这些Watcher就会开始调用update来更新视图。



****图解全过程****



响应式注意项

对于对象来说

- 添加或者删除属性，那么这个属性是无法进入响应式系统的

- 覆盖对象是可以进入响应式系统的

- 解决方法可以用Vue.set()或是他的别名$set()

const app = new Vue({

el: '#app',

data: {}

})

app.msg = 'hello' //不是响应式

app.$set(app, 'msg', 'hello') //响应式

对于数组来说

- 改变数组的长度

- 根据索引值直接赋值

- vue响应式原理理解：

const app = new Vue({

el: '#app',

data: {

arr: [1, 2, 3, 4]

}

})

app.arr[1] = 3333 //不是响应式

app.$set(app.arr, 1, 3333) //是响应式

app.arr.length = 3 //不是响应式

app.arr.splice(3) //是响应式

声明响应式: property 由于 Vue 不允许动态添加根级响应式 property，所以你必须在初始化实例前声明所有根级响应式property，哪怕只是一个空值

Vue 在更新 DOM 时是异步执行的，为了在数据变化之后等待 Vue 完成更新DOM，可以在数据变化之后立即使用Vue.nextTick(callback)。这样回调函数将在 DOM 更新完成后被调用。

Vue.nextTick(callback)或实例方法app.$nextTick(callback)

$nextTick()返回一个Promise对象，所以你可以使用新的 ES2017async/await 语法完成

this.title = ''

this.$nextTick(() => {

alert(this.title)

})

this.title = 123

//alert(123)

声响应式数据的理解：

1、把一个普通的js对象传给Vue实例的data选项对象

　　2、Vue将遍历此对象的所有的属性，并使用Object.defineProperty把这些属性全部转换为getter/setter

3、Vue内部会对数据进行劫持操作，进而追踪依赖，在属性被访问和修改时通知变化。

# vue websocket封装调用

Vue基于WebSocket的封装调用主要可以通过Vue的插件机制和mixin实现。下面简单介绍一种使用Vue插件机制封装WebSocket的方式：

## 1、新建socket.js

import proto from "@/proto/proto";

let PuppyWebSocketMessage = proto.lookupType("Puppy.WebSocket.Message");

// websocket实例

let wsObj = null

// ws连接地址

let wsUrl = null

// let userId = null;

// 是否执行重连 true/不执行 ； false/执行

let lockReconnect = false

// 重连定时器

let wsCreateHandler = null

// 连接成功，执行回调函数

let messageCallback = null

// 连接失败，执行回调函数

let errorCallback = null

// 发送给后台的数据

let sendDatas = {}

/\*\*

\* 发起websocket请求函数

\* @param {string} url ws连接地址

\* @param {Object} agentData 传给后台的参数

\* @param {function} successCallback 接收到ws数据，对数据进行处理的回调函数

\* @param {function} errCallback ws连接错误的回调函数

\*/

export const connectWebsocket = (url, agentData, successCallback, errCallback) => {

wsUrl = url

createWebSoket()

messageCallback = successCallback

errorCallback = errCallback

sendDatas = agentData

}

// 手动关闭websocket （这里手动关闭会执行onclose事件）

export const closeWebsocket = () => {

if (wsObj) {

writeToScreen('手动关闭websocket')

wsObj.close() // 关闭websocket

// wsObj.onclose() // 关闭websocket(如果上面的关闭不生效就加上这一条)

// 关闭重连

lockReconnect = true

wsCreateHandler && clearTimeout(wsCreateHandler)

// 关闭心跳检查

heartCheck.stop()

}

}

// 创建ws函数

const createWebSoket = () => {

if (typeof (WebSocket) === 'undefined') {

writeToScreen('您的浏览器不支持WebSocket，无法获取数据')

return false

}

// const host = window.location.host;

// userId = GetQueryString("userId");

// wsUrl = "ws://" + host + "/websoket" + userId;

try {

wsObj = new WebSocket(wsUrl)

initWsEventHandle()

} catch (e) {

writeToScreen('连接异常，开始重连')

reconnect()

}

}

const initWsEventHandle = () => {

try {

// 连接成功

wsObj.binaryType = "arraybuffer";

wsObj.onopen = (event) => {

onWsOpen(event)

// heartCheck.start()

}

// 监听服务器端返回的信息

wsObj.onmessage = (event) => {

onWsMessage(event)

// heartCheck.start()

}

wsObj.onclose = (event) => {

writeToScreen('onclose执行关闭事件')

onWsClose(event)

}

wsObj.onerror = (event) => {

writeToScreen('onerror执行error事件，开始重连')

onWsError(event)

reconnect()

}

} catch (err) {

writeToScreen('绑定事件没有成功，开始重连')

reconnect()

}

}

const onWsOpen = (event) => {

writeToScreen('CONNECT')

// // 客户端与服务器端通信

// wsObj.send('我发送消息给服务端');

// 添加状态判断，当为OPEN时，发送消息

if (wsObj.readyState === wsObj.OPEN) { // wsObj.OPEN = 1

// 发给后端的数据需要字符串化

console.log('发送标识', sendDatas)

wsObj.send(sendDatas)

}

if (wsObj.readyState === wsObj.CLOSED) { // wsObj.CLOSED = 3

writeToScreen('wsObj.readyState=3, ws连接异常，开始重连')

reconnect()

errorCallback()

}

}

const onWsMessage = (event) => {

const jsonStr = event.data

var buffer = new Uint8Array(jsonStr, 0, jsonStr.length);

let message = PuppyWebSocketMessage.decode(buffer);

writeToScreen('onWsMessage接收到服务器的数据: ', jsonStr)

messageCallback(message)

}

const onWsClose = (event) => {

writeToScreen('DISCONNECT')

// e.code === 1000 表示正常关闭。 无论为何目的而创建, 该链接都已成功完成任务。

// e.code !== 1000 表示非正常关闭。

console.log('onclose event: ', event)

if (event && event.code !== 1000) {

writeToScreen('非正常关闭')

errorCallback()

// 如果不是手动关闭，这里的重连会执行；如果调用了手动关闭函数，这里重连不会执行

reconnect()

}

}

const onWsError = (event) => {

writeToScreen('onWsError: ', event.data)

errorCallback()

}

const writeToScreen = (massage) => {

console.log(massage)

}

// 重连函数

const reconnect = () => {

if (lockReconnect) {

return

}

writeToScreen('3秒后重连')

lockReconnect = true

// 没连接上会一直重连，设置延迟避免请求过多

wsCreateHandler && clearTimeout(wsCreateHandler)

wsCreateHandler = setTimeout(() => {

writeToScreen('重连...' + wsUrl)

createWebSoket()

lockReconnect = false

writeToScreen('重连完成')

}, 3000)

}

// 从浏览器地址中获取对应参数

// eslint-disable-next-line no-unused-vars

const GetQueryString = (name) => {

let reg = new RegExp('(^|&)' + name + '=([^&]\*)(&|$)', 'i')

// 获取url中 ? 符后的字符串并正则匹配

let r = window.location.search.substr(1).match(reg)

let context = ''

r && (context = r[2])

reg = null

r = null

return context

}

// 心跳检查（看看websocket是否还在正常连接中）

const heartCheck = {

timeout: 60000,

timeoutObj: null,

serverTimeoutObj: null,

// 重启

reset() {

clearTimeout(this.timeoutObj)

clearTimeout(this.serverTimeoutObj)

this.start()

},

// 停止

stop() {

clearTimeout(this.timeoutObj)

clearTimeout(this.serverTimeoutObj)

},

// 开启定时器

start() {

this.timeoutObj && clearTimeout(this.timeoutObj)

this.serverTimeoutObj && clearTimeout(this.serverTimeoutObj)

// 15s之内如果没有收到后台的消息，则认为是连接断开了，需要重连

this.timeoutObj = setTimeout(() => {

writeToScreen('心跳检查，发送ping到后台')

try {

const datas = { ping: true }

wsObj.send(JSON.stringify(datas))

} catch (err) {

writeToScreen('发送ping异常')

}

console.log('内嵌定时器this.serverTimeoutObj: ', this.serverTimeoutObj)

// 内嵌定时器

this.serverTimeoutObj = setTimeout(() => {

writeToScreen('没有收到后台的数据，重新连接')

reconnect()

}, 100)

}, this.timeout)

}

}

## 2、使用

import { connectWebsocket } from "../api/websocket";

created(){

connectWebsocket(

URL,

// 发送

"HeartBeat",

// 成功拿到后台返回的数据的回调函数

(data) => {

console.log("成功的回调函数, 接收到的data数据： ", data);

this.Socket(data);

},

// websocket连接失败的回调函数

() => {

console.log("失败的回调函数");

}

);

},

methods:{

Socket(e) {

console.log(e);

}

通过上述方式，可以方便地封装和调用WebSocket相关方法，并将其作为Vue应用程序的一部分进行管理和使用。

# HTML5

HTML5是超文本标记语言（Hypertext Markup Language）的第五个版本，是Web标准的下一代。它的设计旨在支持Web应用程序的开发，包括多媒体和图形内容，以及在不同设备之间流畅交互的能力。与之前的HTML版本相比，HTML5引入了一些新的元素、属性、API和技术以满足Web开发的需求。

一些HTML5的新特性：

新的语义化元素（semantic elements），例如header，nav，section，article，footer，用于更好地描述Web页面的内容和结构，使网页更加易于阅读、可维护、可访问。

视频和音频支持（video和audio元素），直接在Web页面中嵌入视频和音频。

画布（canvas元素），一个可以使用JavaScript进行绘图的区域。

SVG（Scalable Vector Graphics）支持，可用于创建高质量的矢量图形。

地理位置定位服务（geolocation API），可用于获取用户当前位置。

离线应用程序（offline application）支持，可使Web应用程序在离线情况下也可访问数据和功能。

总之，HTML5是Web应用程序开发的重要进步，提供了更丰富、更强大的功能，使得Web应用程序的开发和交互体验更加优秀。

HTML的基本标签

(1)HTML标签

整个网页是从<html>这里开始的，然后到</html>结束。

(2)head标签

head标签代表页面的“头”，定义一些特殊内容，这些内容往往都是“不可见内容”（在浏览器不可见）。

表1 <head>内部标签

|  |  |
| --- | --- |
| <head>内部标签 | 说明 |
| <title> | 定义网页的标题 |
| <meta> | 定义网页的基本信息（供搜索引擎） |
| <style> | 定义CSS样式 |
| <link> | 链接外部CSS文件或脚本文件 |
| <script> | 定义脚本语言 |
| <base> | 定义页面所有链接的基础定位（用得很少） |

<head>的内部标签也非常重要，在前期大家只需要感性认知就可以。

(3)body标签

body标签代表页面的“身”，定义网页展示内容，这些内容往往都是“可见内容”（在浏览器可见）。

后续课程讲解的标签都是在<body>标签内部的各种标签。

3、段落与文字

(一)、段落标签

(1)、段落与文字标签

表1 段落与文字标签

|  |  |  |
| --- | --- | --- |
| 标签 | 语义 | 说明 |
| <h1>~<h6> | header | 标题 |
| <p> | paragraph | 段落 |
| <br> | break | 换行 |
| <hr> | horizontal rule | 水平线 |
| <div> | division | 分割（块元素） |
| <span> | span | 区域（行内元素） |

(2)、文本格式化标签

表2 文本格式化标签

|  |  |  |
| --- | --- | --- |
| 标签 | 语义 | 说明 |
| <strong> | strong（加强） | 加粗 |
| <em> | emphasized（强调） | 斜体 |
| <cite> | cite（引用） | 斜体 |
| <sup> | superscripted（上标） | 上标 |
| <sub> | subscripted（下标） | 下标 |

(二)、网页特殊符号

网页特殊符号只需要记忆一个就行了，就是HTML空格“ ”，其他的特殊符号我们不需要记忆，当我们需要的时候再回来查找一下特殊符号表就OK了。

(三)、自闭合标签

HTML标签分为2种，一种是“一般标签”，另外一种是“自闭合标签”。一般标签有开始符号和结束符号，自闭合标签只有开始符号没有结束符号。

一般标签可以在开始符号和结束符号之间插入其他标签或文字。

自闭合标签由于没有结束符号，不能插入其他标签或文字，只能定义自身的属性。

(1)、一般标签

举例：<body></body>

(2)、自闭合标签

举例：<br/>、<hr/>

(四)、块元素和行内元素

(1)、HTML元素根据浏览器表现形式分为两类：①块元素；②行内元素；

(2)、块元素特点：

（1）独占一行，排斥其他元素跟其位于同一行，包括块元素和行内元素；

（2）块元素内部可以容纳其他块元素或行元素；

常见块元素有：h1~h6、p、hr、div等。

(3)、行内元素特点：

（1）可以与其他行内元素位于同一行；

（2）行内内部可以容纳其他行内元素，但不可以容纳块元素，不然会出现无法预知的效果；

常见行内元素有：strong、em、span等。

HTML 5

HTML指的是HTML 4.01, XHTML是HTML的过渡版本，XHTML是XML风格的HTML 4.01。而HTML 5指的是下一代的HTML,也就是HTML 4.01的升级版。

不过HTML 5已经不再是单纯意义上的标签了，它已经远远超越了标签的范畴。HTML 5 除了新增部分标签之外，还增加了一组技术，包括canvas、SVG、WebSocket.本地存储等。这些新增的技术都是使用JavaScript来操作。也就是说，HTML 5使得HTML从一门“标记语言” 转变为一门“编程语言”。

对于HTML 5中的新技术，在此不做详细介绍。单纯从新增的标签上来看，HTML 5有 以下几个特点。

1、文档类型说明

基于HTML 5设计准则中的“化繁为简”原则，页面的文档类型<!DOCTYPE>被极大地 简化了。

XHTML文档声明如下：

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:// www.w3.org/TR/xhtmll/DTD/xhtmll-transitional.dtdn>

HTML5文档声明如下：

<!DOCTYPE html>

2、标签不再区分大小写

<div>绿叶学习网</DIV>

上面这种写法也是完全符合HTML5规范的。但是在实际开发中，建议所有标签以及属 性都采用小写方式。

3、允许属性值不加引号

<div id=wrapper style=co1or: red> 绿叶学习网 </div>

上面这种写法也是完全符合HTML5规范的。但是在实际开发中，建议标签所有属性值 都加引号，单引号或双引号都可以。

4、允许部分属性的属性值省略

在HTML5中，部分具有特殊性属性的属性值是可以省略的。例如，下面代码是完全符 合HTML 5规范的：

<input type=ntext" readonly/>

<input type="checkbox" checked/>

上面两句代码等价于：

<input type="textM readonly="readonly"/>

<input type="checkbox" checked="checked"/>

在HTML 5中，可以省略属性值的属性如表所示。

表1 -1 HTML5中可以省略属性值的属性

|  |  |
| --- | --- |
| 省略形式 | 等价于 |
| Checked | checked=nchecked" |
| Readonly | readonly="readonly" |
| Defer | defer="defer" |
| Ismap | ismap="ismap" |
| Nohref | nohref="nohref" |
| Noshade | noshade="noshade" |
| Nowrap | nowrap="nowrap" |
| Selected | selected="selected" |
| Disabled | ciisabled="disableci" |
| Multiple | multiple="multiple" |
| Noresize | noresize="disabled" |

一句话概括 HTML、XHTML 和 HTML 5 就是：HTML 指的是 HTML 4.01, XHTML 是HTML的过渡版，HTML5是HTML的升级版。