开发经验分享

前端开发与Git协作

任泽华 2022.08.12

目录

一、基于VUE的前端开发

1. 命名、语法、注释
2. 全局和局部
3. 函数的定义与使用
4. VUE组件传值方式
5. 自己的一些开发经验

二、Git协同操作注意事项

1. 分支规范
2. 开发流程
3. Commit 提交规范
4. 使用技巧
5. 常用命令

一、基于VUE前端开发

1. 命名、语法、注释

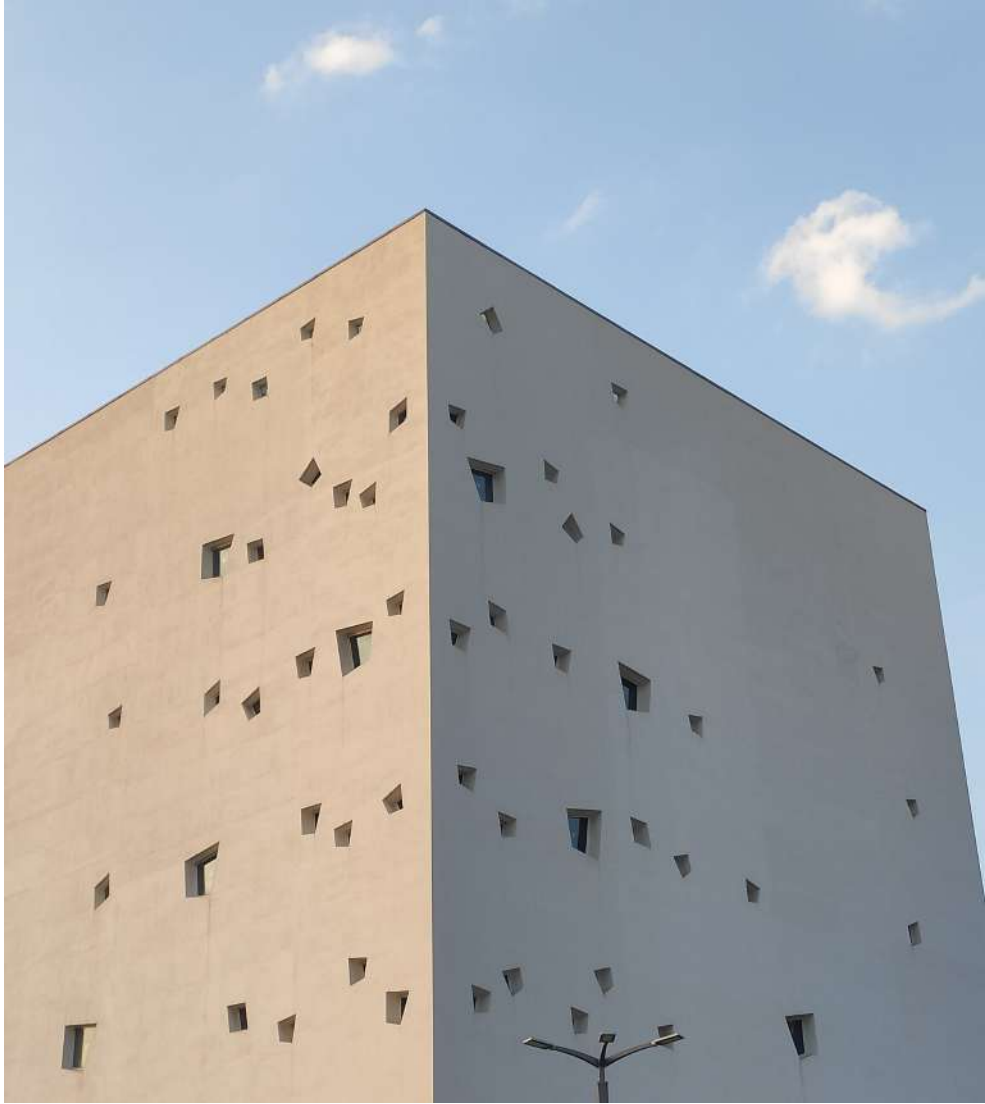
1.1 JS变量、函数命名规范

1.2 CSS命名规范

1.3 JavaScript语法规范

1.4 CSS和HTML语法规范

1.5 代码和注释规范



1.1 JS变量、函数命名规范

> 存在的问题：驼峰式、下划线式混用，少数变量命名随意

```
modifyPatternFormRules = {
    threat_level:[{ required: true, message: "请选择风险等级", trigger: "blur"}],
    handle_info:[{ required: true, message: "请输入标定信息", trigger: "blur"}]
},
addPatternDialogVisible = false, // 模式添加弹窗是否可见

window.onresize = function(){
    a.resize();
    b.resize();
    c.resize();
    d.resize();
}
```

每个人对变量的命名规范都不太一样，建议在一个项目中尽量保持相同，虽不影响阅读和功能，但是整体看起来稍显凌乱。这里介绍一下我自己常用的命名规范和搜集资料时找到的好的规范以作参考：

1.1 JS变量、函数命名规范

> 存在的问题：驼峰式、下划线式混用，少数变量命名随意

```
modifyPatternFormRules = {
    threat_level:[{ required: true, message: "请选择风险等级", trigger: "blur"}],
    handle_info:[{ required: true, message: "请输入标定信息", trigger: "blur"}]
},
addPatternDialogVisible = false, // 模式添加弹窗是否可见

window.onresize = function(){
    a.resize();
    b.resize();
    c.resize();
    d.resize();
}
```

每个人对变量的命名规范都不太一样，建议在一个项目中尽量保持相同，虽不影响阅读和功能，但是整体看起来稍显凌乱。这里介绍一下我自己常用的命名规范和搜集资料时找到的好的规范以作参考：

1.1 JS变量、函数命名规范

1. **驼峰式命名法**：第一个单词首字母小写,其余每一个有意义的单词首字母大写

```
var studentInfo;
let patternInfo // Infomation--good
let cardSty // Style--bad
```

2. **常量全部大写**

```
const NAME
```

3. 方法/函数前加入**操作动词**

   - get 获取/set 设置/add 增加/remove 删除/create 创建/destory 移除；
   - start 启动/stop 停止/open 打开/close 关闭/read 读取/write 写入；
   - load 载入/save 保存/create 创建/destroy 销毁；
   - begin 开始/end 结束/backup 备份/restore 恢复/detach 脱离；
   - import 导入/export 导出/split 分割/merge 合并/inject 注入/extract 提取；

4. **根据类型加前缀**

   - fn：表示函数。例如：fnGetName，fnSetAge；
   - dom：表示Dom对象，例如：domForm；

5. 临时变量不影响阅读时可简写

   - 作用域不大临时变量可以简写，比如：str，num，bol，obj，fun，arr；
   - 循环变量可以简写，比如：i，j，k等。

1.2 CSS命名规范

- 大部分同JS命名规范，有一点需特别注意：**嵌套的元素一定要在前面写上父元素**，这样不仅是为了在继承父元素减少代码量的同时便于阅读，最重要的目的是限制CSS样式的作用域，否则就会出现严重的样式混乱。

```
#header{
    padding: 10px 0 0 0;
    .profilepic{
        display: block;
        position: relative;
        z-index: 100;
    }
    .header-menu{
        height: auto;
        margin: 10px;
        ul{
            text-align: center;
            cursor: default;
        }
        li{
            display: inline-block;
            margin: 3px;
        }
    }
}
```

1、语法区别：

id对应css是用样式选择符“#”（井号）。

class对应css是用样式选择符“.”(英文半角输入句号百)。

2、使用次数区别：

id属性，只能被一个元素调用。同页面只可调用一次。

class类标记，可以用于被多个元素调用，在同一个页面可以调用无数次。

> ID就像一个人的身份证，用于识别这个DIV的，Class就像人身上穿的衣服，用于定义这个DIV的样式。一般一个网页不设二个或二个以上同ID的div，但Class可以多个DIV用同一个Class。

1.2 CSS命名规范

- 大部分同JS命名规范，有一点需特别注意：**嵌套的元素一定要在前面写上父元素**，这样不仅是为了在继承父元素减少代码量的同时便于阅读，最重要的目的是限制CSS样式的作用域，否则就会出现严重的样式混乱。

```
#header{
    padding: 10px 0 0 0;
    .profilepic{
        display: block;
        position: relative;
        z-index: 100;
    }
    .header-menu{
        height: auto;
        margin: 10px;
        ul{
            text-align: center;
            cursor: default;
        }
        li{
            display: inline-block;
            margin: 3px;
        }
    }
}
```

1、语法区别：

id对应css是用样式选择符“#”（井号）。

class对应css是用样式选择符“.”(英文半角输入句号百)。

2、使用次数区别：

id属性，只能被一个元素调用。同页面只可调用一次。

class类标记，可以用于被多个元素调用，在同一个页面可以调用无数次。

> ID就像一个人的身份证，用于识别这个DIV的，Class就像人身上穿的衣服，用于定义这个DIV的样式。一般一个网页不设二个或二个以上同ID的div，但Class可以多个DIV用同一个Class。

1.2 CSS命名规范

- 大部分同JS命名规范，有一点需特别注意：**嵌套的元素一定要在前面写上父元素**，这样不仅是为了在继承父元素减少代码量的同时便于阅读，最重要的目的是限制CSS样式的作用域，否则就会出现严重的样式混乱。

```
#header{
    padding: 10px 0 0 0;
    .profilepic{
        display: block;
        position: relative;
        z-index: 100;
    }
    .header-menu{
        height: auto;
        margin: 10px;
        ul{
            text-align: center;
            cursor: default;
        }
        li{
            display: inline-block;
            margin: 3px;
        }
    }
}
```

1、语法区别：

id对应css是用样式选择符“#”（井号）。

class对应css是用样式选择符“.”(英文半角输入句号百)。

2、使用次数区别：

id属性，只能被一个元素调用。同页面只可调用一次。

class类标记，可以用于被多个元素调用，在同一个页面可以调用无数次。

> ID就像一个人的身份证，用于识别这个DIV的，Class就像人身上穿的衣服，用于定义这个DIV的样式。一般一个网页不设二个或二个以上同ID的div，但Class可以多个DIV用同一个Class。

1.3 JavaScript语法规范

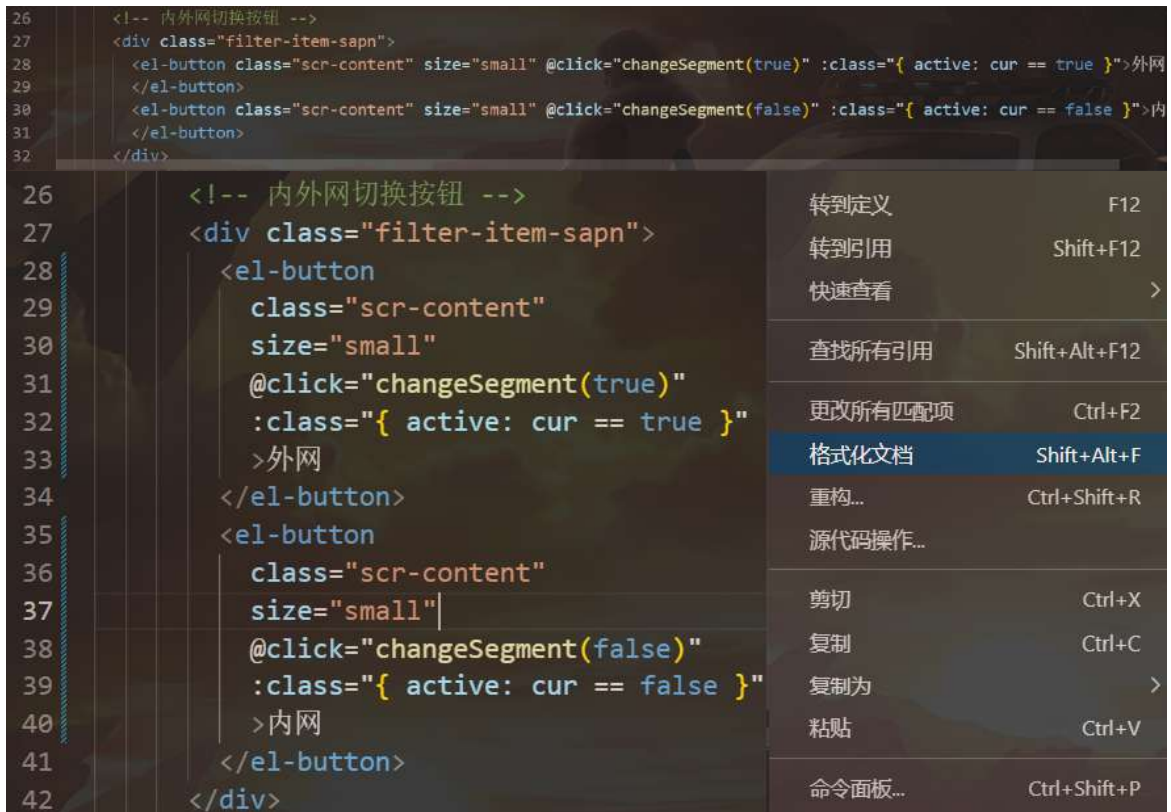
1. JS**严格区分大小写**；
2. JS中每一条语句结尾的分号可加可不加，**建议还是加上分号**，因为在编译的时候可能会出现一些冲突报错；
3. JS中会**忽略多个空格和换行。**
4. **注意缩进**，便于后期调试也便于他人阅读。
5. 只出现一次的dom元素引用就不要建一个变量来存了，会占用没必要的内存。

1.4 CSS和HTML语法规范

- 虽然在HTML5当中，标签可以不用闭合，但是为了规范，也是为了避免一些错误，还是**建议标签闭合。**
- 在HTML中**正确嵌套所有元素**，必须按照打开元素的顺序进行关闭。
- 样式太多时还是不要写在HTML里了，建一个CSS表来存放样式。
- 不要在CSS样式的0后面写px，不要增加代码的负担。
- **单引号和双引号不要混用**，建议全部改为双引号。

1.5 代码和注释规范

- 不要太长以至于超过页面范围，需要横向进度条，**善用回车换行**增加代码可读性。



1.5 代码和注释规范

- **善用注释**，对于某些常量变量方法的定义要说清楚它们的作用，尤其是每个组件中的data全局变量。
- 公共组件和各栏目的维护者最好需要在**文件头部加上注释**说明：

```
/**
*文件用途说明
*作者姓名
*联系方式
*制作日期
**/
```

- **大的模块**在前面使用此方法进行注释：

```
//========
//代码用途
//========
```

- 调试时注释掉的代码块如果确定没用要**及时删除。**

一、基于VUE前端开发

2. 全局和局部

2.1 CSS样式

2.2 系统全局设置

2.3 内存管理

2.4 this和_this

2.1 CSS样式

- 为了减轻代码量，能复用的一定要**复用**，放到全局样式库里面作为全局样式，**根据需要做对应修改**即可。

- **切不可对同名样式在不同的页面分别定义不同的属性**，因为CSS是层叠样式表，通过加载只要样式不冲突，新打开一个页面后是会保留之前的样式的，这就会导致某些组件在打开另一个页面回来后面目全非了。

```
829  .filter-block {
830    /* background-color: #fff; */
831    z-index: 1000;
832    width: 95%;
833    height: 15%;
834    min-height: 80px;
835    display:flex;
836    justify-content:space-between;
837    flex-wrap: wrap;
838    align-items:center;
839    margin-left: 2.5%;
840    margin-right: 2.5%;
841    /* border-radius:20px; */ /* 圆角设置 */
842    background: #fff;
843  }
```

```
333  .filter-block {
334    z-index: 1000;
335    width: 95%;
336    height: 80px;
337    display: flex;
338    justify-content: space-between;
339    align-items: center;
340    margin-left: 2.5%;
341    margin-right: 2.5%;
342    background: #fff;
343  }
344  .filter-item-sapn{
345    margin: 10px;
346  }
347  .prompt-text{
```

2.1 CSS样式

- 可以**通过使用`scoped`属性来避免影响全局样式**，这也是常见处理方式，但也需要注意一些问题：

1. scoped定义的组件不会影响全局样式和孙子组件，却**会影响其根子组件（直接子组件）样式。**这样设计是为了让父组件可以从布局的角度出发，调整其子组件根元素的样式。开发时需要注意。

2. 如果你希望 scoped 样式中的一个选择器能够作用得“更深”，例如影响所有子组件，你可以使用`>>>`操作符，有些像 Sass/less 之类的预处理器无法正确解析 >>>,这种情况下你可以使用`/deep/ `操作符取而代之。

```
<style scoped>
.a >>> .b { /* ... */ }
</style>

<style lang="less" scoped>
.a /deep/ .b  { /* ... */ }
</style>
```

2.2 系统全局设置

- **全局量与方法：**

为便于集中修改与管理，增强复用，我们把一些全局配置内容存为**系统全局量和全局方法。**

1. 系统全局量存放在`src/api`文件夹下，包含了配色表、模拟时间、时间相关方法、服务器IP等。
2. 在组件源文件中使用`import`来引入。
3. 全局变量是可以通函数来修改的，如当前模拟时间。

- **路由配置：**

在`src/router/index.js`里面配置即可，需要指出的是，本template已经有了路由配置，我们直接用即可，但是在了解其原理也是很重要的。

对我个人而言，当时配置路由时踩的最多坑应该就是**路由嵌套**（多级路由）。

1. 子级路由path不能加/，否则会找不到组件。

2. 路由入口要写完整路径，否则会找不到组件。如从“时间态势分析”跳转到“整体态势复查”，必须使用`/temporal/menu1`。

2.2 系统全局设置

在多个页面**引入第三方库**（如Echarts、D3、jQuery等）时，可以将其作为全局库引入。在main.js中配置：

```
5  import axios from 'axios' // 通信框架
6  import echarts from 'echarts' // 绘制统计图框架
7  import ElementUI from 'element-ui'
36 Vue.use(ElementUI)
37
38 // 引入axios框架与后端通信
39 // 设置axios为form-data
40 axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded'
41 axios.defaults.headers.get['Content-Type'] = 'application/x-www-form-urlencoded'
42 axios.defaults.transformRequest = [function(data) {
43   let ret = ''
44   for (let it in data) {
45     ret += encodeURIComponent(it) + '=' + encodeURIComponent(data[it]) + '&'
46   }
47   return ret
48 }]
49 Vue.prototype.$http = axios
50
51 // 全局引入echarts，使用时打'$echarts'即可
52 Vue.prototype.$echarts = echarts
```

2.3 内存管理

> `内存泄漏`并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。文章1 文章2

JavaScript常见内存泄漏：

1. **意外的全局变量**

```
// bar为我们不需要的全局变量
function fn() {
  a = 'global variable'
}
fn()
// foo 调用自己，this 指向了全局对象（window）
function fn() {
  this.a = 'global variable'
}
fn()
```

- 使用严格模式，在 JavaScript 文件头部或者函数的顶部加上 use strict。

- 变量声明的时候**使用var或let**：

区别：

1.使用var声明的变量，其作用域为该语句所在的函数内，且存在变量提升现象；

2.使用let声明的变量，其作用域为该语句所在的代码块内，不存在变量提升；

3.let不允许在相同作用域内，重复声明同一个变量。

- 建议**最好使用let**进行声明，可以很好地避免因变量提升造成的内存泄漏。

2.3 内存管理

2. **闭包**（通过自我调用将函数内部的变量转化为全局变量）

```
function fn () {
  var a = "I'm a";
  return function () {
    console.log(a);
  };
}
```

- 结束后及时清理：`a=null;`。
- 将事件处理函数定义在外部，解除闭包。

3. **没有清理的 DOM 元素引用**

```
const refA = document.getElementById('refA');
document.body.removeChild(refA); // dom删除了
console.log(refA, 'refA'); // 但是还存在引用能console出整个div 没有
refA = null;
console.log(refA, 'refA'); // 解除引用
```

- 手动删除，`refA = null;`

定时器中有 dom 的引用，即使 dom 删除了，但是定时器还在，所以内存中还是有这个 dom。（下一条）

4. **被遗忘的定时器或者监听回调**

```
// 定时器
var serverData = loadData()
setInterval(function () {
  var renderer = document.getElementById('renderer')
  if (renderer) {
    renderer.innerHTML = JSON.stringify(serverData)
  }
}, 5000)

// 观察者模式
var btn = document.getElementById('btn')
function onClick(element) {
  element.innerHTMl = "I'm innerHTML"
}
btn.addEventListener('click', onClick)
```

- 手动删除定时器和 dom。
- removeEventListener 移除事件监听

2.3 内存管理

vue 中容易出现内存泄露的几种情况：

1. **窗口全局变量**造成的内存泄露

```
export default {
  mounted() {
    window.test = {
      // 此处在全局window对象中引用了本页面的dom对象
      name: 'home',
      node: document.getElementById('home'),
    }
  },
   // 解决方法
  destroyed () {
      window.test = null // 页面卸载的时候解除引用
  }
}
```

声明的全局变量在切换页面的时候没有清空

- 在页面卸载的时候顺便处理掉该引用：

2. **监听**在 window/body 等**事件没有解绑**

```
export default {
    mounted () {
      window.addEventListener('resize', this.func) // window
    }
    // 解决方法
    beforeDestroy () {
      window.removeEventListener('resize', this.func)
    }
}
```

- 在页面销毁前解除引用，释放内存

3. 绑在 EventBus 的事件没有解绑（我们没怎么用过，可以自己查阅）

2.3 内存管理

4. **Echarts大量占用——重要！！！D3同理**

每一个图例在没有数据的时候它会创建一个定时器去渲染气泡，页面切换后，echarts 图例是销毁了，但是这个 echarts 的实例还在内存当中，同时它的气泡渲染定时器还在运行。这就导致 Echarts 占用 CPU 高，导致浏览器卡顿，当数据量比较大时甚至浏览器崩溃。

```
beforeDestroy () {
  this.chart.clear()
}
```

5. **v-if 指令**产生的内存泄露

v-if 绑定到 false 的值，但是实际上 dom 元素在隐藏的时候没有被真实的释放掉。

- 及时清除不需要的dom元素，而不是隐藏起来。

2.4 this和_this

- _this是变量名，this代表父函数，如果子函数还用this，指向就变成子函数了，_this是用来存储指向的。

```
848   netDraw(nodesData, linksData, svg, netID) {
849     // 绘制网络
850     svg.selectAll("*").remove(); // 清除上一张画布内容
851     let _this = this; // 用于在子函数中调用this的方法
852     svg
853       .attr("viewBox", width, height) // 用于后续拖动
854       .on("click", function (d) {
855         if (!_this.isEdgeClicked && !_this.isNodeClicked) {
856           _this.clear();
857           _this.subIsNode = false;
858           _this.subIsEdge = false;
859         }
860       });
```

```
281     // 定时请求新数据，一秒钟查询一次当前时间，符合条件就请求刷新
282     let _this = this;
283     this.dynamicCharts = setInterval(function () {
284       let fresh = false; // 是否到达刷新时间
285       switch (_this.$refs.Submenu.timeInterval) {
286         case 1:
287           if (GLOBAL.time.slice(17) === "00") {
288             fresh = true;
289           }
290           break;
```

一、基于VUE前端开发

3. 函数的定义与使用

3.1 函数声明

3.2 函数表达式

3.3 匿名函数（自调用函数）

3.4 构造函数（面向对象）

3.5 箭头函数

javaScript 函数通过 function 关键词进行定义，其后是函数名和括号 ()。文章

3.1 函数声明

必须有名字，会函数提升，在预解析阶段就已经创建，声明前后都可以调用

```
//定义函数名
function fn(){
     console.log(123);
}
fn()
```

3.2 函数表达式

一种变量赋值，函表达式可以没有名字（匿名函数）,没有函数提升。

```
//将函数赋值给一个变量，可以是匿名函数
var fn = function(){
    hi: function(){  }, //方法
};
fn() //调用
fn.hi() //调用方法
```

3.3 匿名-自调用函数

函数表达式可以 "自调用"。

通过添加括号，来说明它是一个函数表达式：

如果需要执行匿名函数，在匿名函数后面加上一个括号即可立即执行！

```
(function (){
    //此时会输出a
    console.log("a");
})()
//以上函数实际上是一个 匿名自我调用的函数（没有函数名）。
```

3.4 构造函数（面向对象）

通过 new 函数名 来实例化对象的函数叫构造函数。

```
function Person(name,age){
    this.name = name
    this.age = age
}
let p1 = new Person('张三','18')
```

3.5 箭头函数

箭头函数表达式的语法比函数表达式更简洁，并且没有自己的this，arguments，super或new.target。箭头函数表达式更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。

```
(参数1, 参数2, …, 参数N) => { 函数声明 }
(参数1, 参数2, …, 参数N) => 表达式(单一)
// 相当于：(参数1, 参数2, …, 参数N) =>{ return 表达式; }

() => {函数声明} // 没有参数的函数应该写成一对圆括号：

const fn = (a, b) => { //多个参数
    let result = a + b;
    console.log(result);//3
}
fn(1, 2)

var fn2 = c => { //只有一个参数
    console.log(c); //davina
}
fn2(‘davina‘);

let fn3 = () => { //没有参数
    console.log(‘123‘);
}
fn3(); //123
```

一、基于VUE前端开发

4. VUE组件之间传值方式

4.1 props,emit

4.2 $parent,$children

4.3 $ref

4.4 provide/inject

4.5 EventBus 事件总线 （任意两个组件通讯）

4.6 其他

4.1 props,emit

父传子的实现方式就是通过props属性，子组件通过props属性接收从父组件传过来的值，而父组件传值的时候使用 v-bind 将子组件中预留的变量名绑定为data里面的数据即可。

子组件通过$emit(事件名，参数)向外弹出一个自定义事件，在父组件中的属性监听事件，可以获得子组件中传出来的值。

```
//  父组件
<hello-world msg="hello world!" @confirm="handleConfirm"><hello-world>
//  子组件
 props: {
    msg: {
      type: String,
      default: ''
    }
  },
  methods: {
    handleEmitParent(){
        this.$emit('confirm', list)
    }
  }
```

4.2 $parent,$children

通过$parent,$children 来访问组件实例，进而去获取或者 改变父子组件的值。 （仅限于父子组件之间，不推荐使用，因为不利于维护，一旦组件层次发生了变化，就需要更改其中的层次关系）

```
675     // 清除卡片
676     deleteCard() {
677       this.drawCard(this.defaltType, null);
678       this.$parent.clear();
679     },
```

4.3 $ref

通过引用的方式获取子节点，常用于父组件中调用子组件的方法或者获取子组件的属性。

```
310       this.initDraw(nodesData, linksData); // 重画网络
311       this.$refs.Details.drawCard("告警", null); // 清空卡片信息
312       this.disStatus = ""; // 刷新后显示模式恢复默认
313       this.compFilter = "";
314       this.markOpen = false;
```

4.4 provide/inject

依赖注入，常见于插件或者组件库里。

多个组件嵌套时，顶层组件provide提供变量,后代组件都可以通过inject来注入变量。

缺陷：传递的数据不是响应式的，inject接收到数据后，provide中的数据改变，但是后代组件中的数据不会改变。所以 建议传一些常量或者方法。

```
//  顶层组件
export default {
  provide(){
    return {
      msg: 'hello world!'
    }
  }
}

//  后代组件
export default {
  inject: ['msg']
}
```

4.5 EventBus 事件总线 （任意两个组件通讯）

用 $emit去监听，用$on去触发，注意需要$off来取消监听，否则可能会造成内存泄漏。

```
// 方法一
// 抽离成一个单独的 js 文件 Bus.js ，然后在需要的地方引入
// Bus.js
import Vue from "vue"
export default new Vue()

// 方法二 直接挂载到全局
// main.js
import Vue from "vue"
Vue.prototype.$bus = new Vue()

// 方法三 注入到 Vue 根对象上
// main.js
import Vue from "vue"
new Vue({
    el:"#app",
    data:{
        Bus: new Vue()
    }
})
```

```
// 在需要向外部发送自定义事件的组件内
<template>
    <button @click="handlerClick">按钮</button>
</template>
import Bus from "./Bus.js"
export default{
    methods:{
        handlerClick(){
            // 自定义事件名 sendMsg
            Bus.$emit("sendMsg", "这是要向外部发送的数据")
        }
    }
}

// 在需要接收外部事件的组件内
import Bus from "./Bus.js"
export default{
    mounted(){
        // 监听事件的触发
        Bus.$on("sendMsg", data => {
            console.log("这是接收到的数据：", data)
        })
    },
    beforeDestroy(){
        // 取消监听
        Bus.$off("sendMsg")
    }
}
```

4.6 其他

$attrs、$listener:

适用于多级组件嵌套，但是不做中间处理的情况。比如祖先组件向孙子组件传递数据。

$attrs 可以获取父组件传进来，但是没有用props接收的属性。可以通过v-bind="$attrs"传入内部组件。

Vuex 状态管理器:

集中式存储管理所有组件的状态。

可以解决 多个视图依赖同一个状态 或者是 来自不同视图的行为需要变更同一个状态 的问题。

localStorage/sessionStorage:

持久化存储。

一、基于VUE前端开发

5. 自己的一些开发经验

5.1 页面布局

5.2 功能拆分与复用

5.3 前后端交互

5.1 页面布局

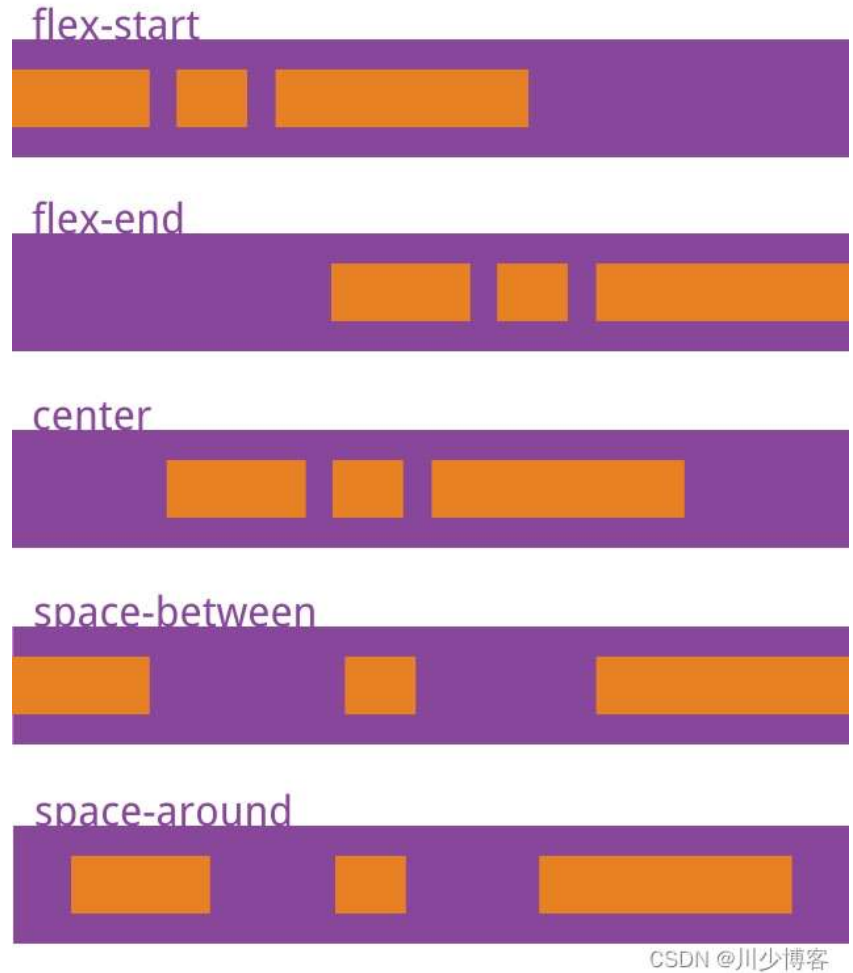
方式推荐：

1. **首推flex布局法**

   - flex是Flexible Box的缩写，意为弹性布局，用来为盒模型提供最大的灵活性。
   - float浮动布局需要动态监控窗口，自己安排左右对齐，适应性差。
   - 个人比较喜欢使用space-between或space-around。

2. **margin和padding**

margin是指从自身边框到另一个容器边框之间的距离，即容器外距离；而padding是盒子边框与盒子内部元素的距离，即容器内距离。




实时处理 / 告警模式分析 / 告警模式管理
拓扑类型: 所有类型
div.filter-item-span 300 × 40 Flex 项目
中心节点数: 单中心
中心IP: *
告警类型: *
边IP: *
风险等级: 未标记
外网
内网
搜索
实时处理 / 子图分析 / 子图模式管理
时间选择: 2021-09-03 14:00:00 至 2021-09-04 14:00:00
拓扑类型: 所有类型
中心节点数: 所有类型
中心IP: *
模式ID: *
风险等级: 所有类型
外网
内网
查询
实时处理 / 告警明细查询
时间选择: 2021-09-01 00:00:00 至 2021-09-01 08:00:00
源IP: *
目的IP: *
源端口: *
目的端口: *
告警类型: *
外网
内网
查询
刷新间隔/秒: 120
确定
当前时间: 2021-09-04 14:41:55
确定
上次刷新时间: 2021-09-04 14:41:55
窗口时间/小时: 1
确定
参考
刷新倒计时/秒: 118
外网
内网
实时查询:

5.1 页面布局

存在的问题

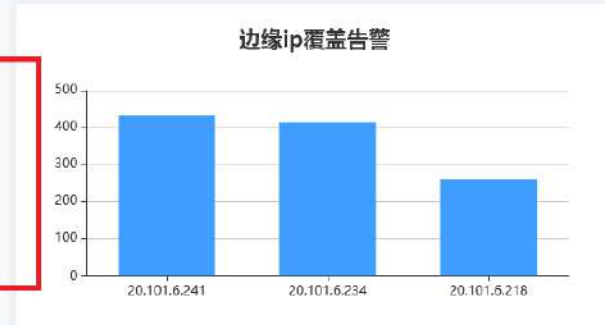
1. **上下margin太宽**，占空间较大。
2. **左右margin太窄**，显得不好看。
3. dom元素过多时要**合理分配上下div容器**，避免出现拐弯回来的情况。
4. **表格**的每一栏按照预期的内容多少排布宽度，**不要默认等宽**，尽量不要让表项超出出现进度条，出现的话要**把包含操作按钮的列固定。**
5. 背景**不要留下大块灰底色**，在子元素中将background设置为white或#FFF。
6. 表格和背景之间再加一层白色的padding，否则很丑。

	18	汇聚型	单中心	196.187.197.156	敏感文件探测;信息泄露;应用漏洞攻击	195.186.196.161;195.186.196.162;20....	未标记	未标定	未标定	查看关联子图	分析	修改
	19	汇聚型	单中心	57.92.181.77	Web应用攻击	202.169.108.246;202.169.13.219;202....	未标记	未标定	未标定	查看关联子图	分析	修改
	20	汇聚型	单中心	71.184.174.31	信息泄露	196.187.197.250;196.187.197.251;196...	未标记	未标定	未标定	查看关联子图	分析	修改

共 812 条 20条/页 < 1 2 3 4 5 6 ••• 41 > 前往 1 页

编号	源IP	告警类型	次数	开始时间	结束时间	目的IP分布特点	历史发生情况	来源特点	发	操作
0	202.169.186.178	MISC攻击	6	2021-09-04 14:01:18	2021-09-04 13:21:18	分散的3个目的IP		单一告警来源	此	处理
4	202.169.186.10	信息泄露	22	2021-09-04 13:52:35	2021-09-04 13:50:56	分散的14个目的IP		单一告警来源	此	处理
6	20.101.114.231	MISC攻击	16	2021-09-04 14:03:00	2021-09-04 13:58:43	分散的4个目的IP		单一告警来源	此	处理
7	202.169.94.127	MISC攻击	6	2021-09-04 14:03:01	2021-09-04 13:23:01	分散的4个目的IP		单一告警来源	此	处理
10	202.169.108.245	MISC攻击	6	2021-09-04 13:54:03	2021-09-04 13:24:03	分散的3个目的IP		单一告警来源	此	处理

参考



告警类型	边IP	覆盖子图数	覆盖日志数
信息泄露	20.101.6.241	72	432
信息泄露	20.101.6.234	72	413
信息泄露	20.101.6.218	72	259

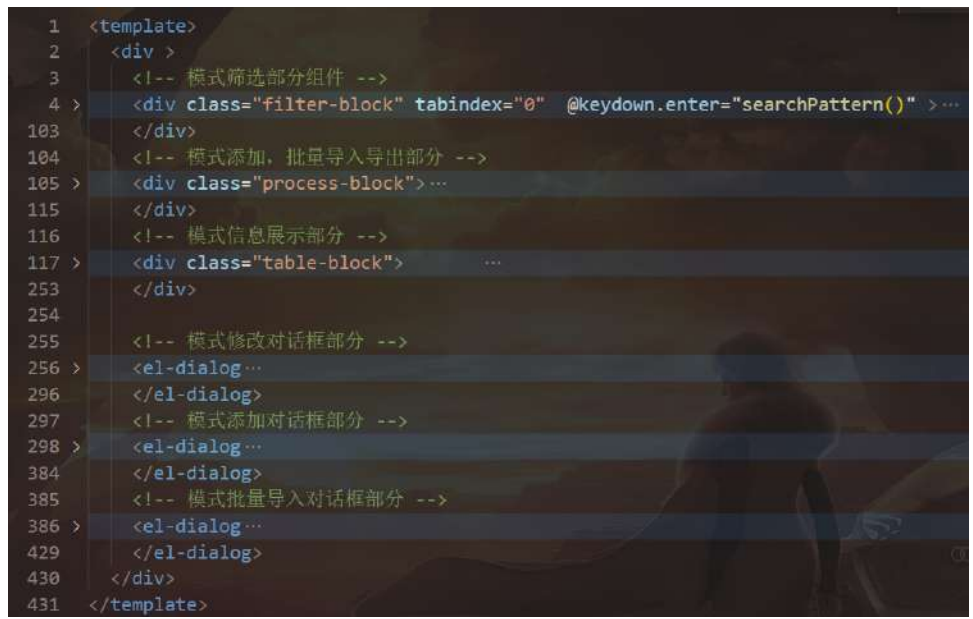
数据导出

时间	告警总数	告警类型数	源IP数	源端口数	目的IP数	目的端口数
2021-09-03 14:10:00	253	16	82	145	41	118
2021-09-03 14:20:00	282	20	82	115	57	128
2021-09-03 14:30:00	277	16	95	145	62	136
2021-09-03 14:40:00	227	19	74	100	55	125

参考

5.2 功能拆分与复用

- 将组件按照功能进行拆分，保留组件之间必要的通信即可，不要放到同一个vue源文件内，否则在代码维护和故障排查时都会很麻烦。一个组件内部耦合度巨大会带来内存溢出等问题。
- 功能性的JavaScript函数能复用尽量复用，不要图省事直接复制一份，这样会带来很多冗余代码。
- 同一份数据不要在多个组件中存放备份，节省内存资源。



5.3 前后端交互

- 每次**请求数据不要太多**，量大时先让后端返回名称列表，再根据列表动态请求。
- **能让后端完成的功能不要在前端完成**，减轻web浏览器负担。
- 请求内容不多时，可以使用`get`；但是请求内容复杂、包含私密信息（如子图信息）的时候，使用`post`。


JSON 原始数据 头
保存 复制 全部折叠 全部展开 过滤 JSON
filename list:
0: "2021-09-01 00:00:00to2021-09-01 01:00:00-out12"
1: "2021-09-01 00:00:00to2021-09-01 01:00:00-out14"
2: "2021-09-01 00:00:00to2021-09-01 01:00:00-out15"
3: "2021-09-01 00:00:00to2021-09-01 01:00:00-out16"
4: "2021-09-01 00:00:00to2021-09-01 01:00:00-out18"
5: "2021-09-01 00:00:00to2021-09-01 01:00:00-out3"
6: "2021-09-01 00:00:00to2021-09-01 01:00:00-out7"
7: "2021-09-01 00:00:00to2021-09-01 01:00:00-out9"
8: "2021-09-01 01:00:00to2021-09-01 02:00:00-out14"
9: "2021-09-01 01:00:00to2021-09-01 02:00:00-out15"
10: "2021-09-01 01:00:00to2021-09-01 02:00:00-out16"
11: "2021-09-01 02:00:00to2021-09-01 03:00:00-out10"
12: "2021-09-01 02:00:00to2021-09-01 03:00:00-out13"
13: "2021-09-01 02:00:00to2021-09-01 03:00:00-out14"
14: "2021-09-01 02:00:00to2021-09-01 03:00:00-out15"
15: "2021-09-01 03:00:00to2021-09-01 04:00:00-out12"
16: "2021-09-01 03:00:00to2021-09-01 04:00:00-out15"
17: "2021-09-01 03:00:00to2021-09-01 04:00:00-out16"
18: "2021-09-01 03:00:00to2021-09-01 04:00:00-out17"
19: "2021-09-01 03:00:00to2021-09-01 04:00:00-out18"
20: "2021-09-01 03:00:00to2021-09-01 04:00:00-out4"
21: "2021-09-01 03:00:00to2021-09-01 04:00:00-out7"
22: "2021-09-01 04:00:00to2021-09-01 05:00:00-out12"
23: "2021-09-01 04:00:00to2021-09-01 05:00:00-out13"
24: "2021-09-01 04:00:00to2021-09-01 05:00:00-out14"
25: "2021-09-01 04:00:00to2021-09-01 05:00:00-out5"
JSON 原始数据 头
保存 复制 全部折叠 全部展开 (慢) 过滤 JSON
2021-09-01 00:00:00to2021-09-01 01:00:00-out12: {…}
2021-09-01 00:00:00to2021-09-01 01:00:00-out14: {…}
2021-09-01 00:00:00to2021-09-01 01:00:00-out15: {…}
2021-09-01 00:00:00to2021-09-01 01:00:00-out16: {…}
2021-09-01 00:00:00to2021-09-01 01:00:00-out18: {…}
2021-09-01 00:00:00to2021-09-01 01:00:00-out3: {…}
2021-09-01 00:00:00to2021-09-01 01:00:00-out7:
state: "successfully searched"
gtries: {…}
pattern info: {…}
handle info: {…}
pattern_match: {…}
subnet_info: {…}
category_distribution: {…}
side_ip_distribution: {…}
2021-09-01 00:00:00to2021-09-01 01:00:00-out9:
state: "successfully searched"
gtries: {…}
pattern info: {…}
handle info: {…}
pattern_match: {}
subnet_info: {…}
category_distribution: {…}
side_ip_distribution:
49.102.142.5: 5
231.220.112.118: 4
49.109.232.115: 2

二、Git协同注意事项

1. 分支规范
2. 开发流程

2.1 一般开发流程

2.2 生产环境Bug修复流程

3. Commit 提交规范

3.1 提交的日志格式

3.2 更新、合并规范

4. 使用技巧
5. 常用命令

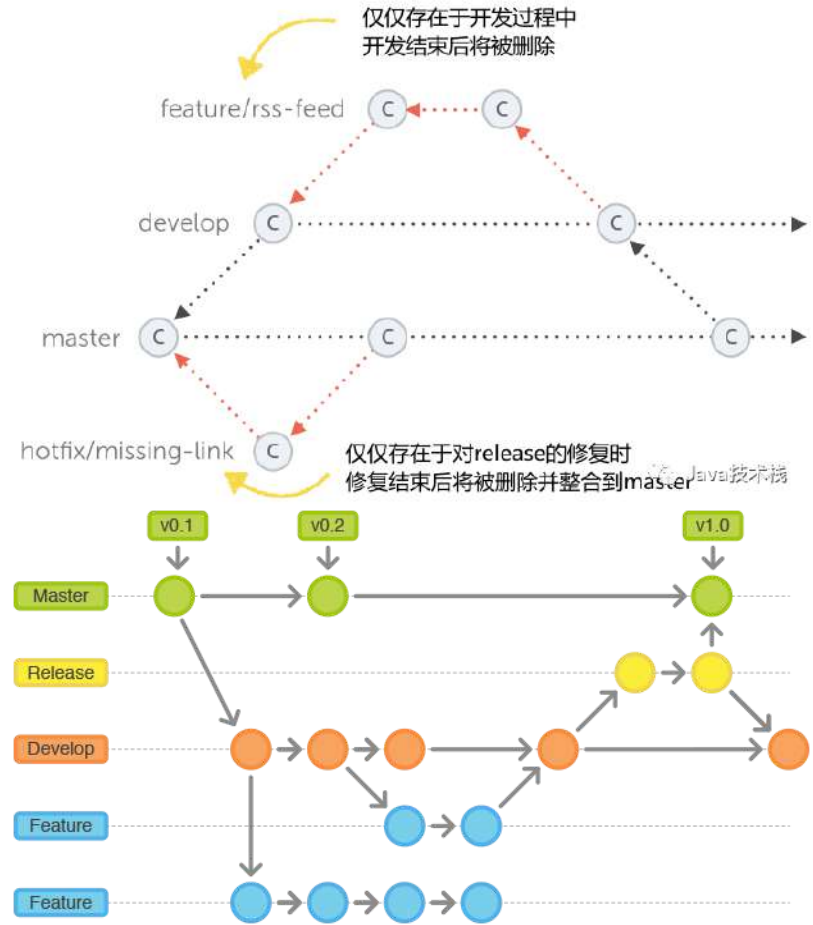
1. 分支规范

1. **常设分支**：永久不删除

- 主干分支 master/main
- 开发分支 develop/dev

2. **临时分支**：用完立即删除

- 功能分支 feature-*
- 测试环境的稳定分支 release
- 测试阶段修复Bug bugfix-*
- 线上出现的紧急Bug修复 hotfix-*



2.1 一般开发流程

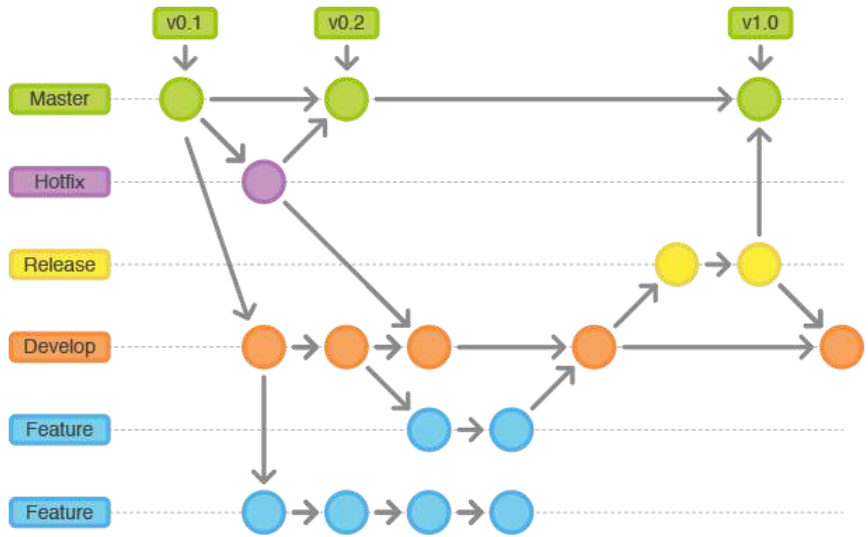
1. 从 develop 分支**切出多个**命名为 feature-* 分支**开发新功能。**
2. 开发者完成开发，**提交**分支到远程仓库。
3. 开发者**发起merge请求**(可在gitlab页面“New merge request”)，将新分支请求merge到 develop 分支，并提醒code reviewer进行review。
4. code reviewer对代码review之后，若无问题，则**接受merge请求**，新分支merge到 develop 分支，同时可删除新建分支；若有问题，则不能进行merge，可close该请求，同时通知开发者在新分支上进行相应调整。调整完后提交代码重复review流程。
5. 测试时，直接从当前 **develop 分支merge到 release 分支**，重新构建测试环境完成转测。
6. 测试完成后，从**release分支merge到 master 分支**，基于 master 分支构建生产环境完成上线。并对 master 分支打tag，tag名可为v1.0.0_2019032115(即版本号_上线时间)

2.2 生产环境Bug修复流程：

非紧急Bug或优化： 非关键业务流程问题，仅影响用户使用体验，或出现频率较小等，为非紧急Bug，可规划到后续版本进行修复，同开发流程。

紧急Bug： 严重影响用户使用的为紧急Bug，需立即进行修复。如关键业务流程存在问题，影响用户正常的业务行为

- 从 master 分支切出一个bug修复分支，完成之后需要同时merge到 master 分支与 develop 分支
- 如果需要测试介入验证，则可先merge到 release 分支，验证通过后再merge到 master 分支上线



3.1 提交的日志格式

每次git提交日志格式为：**「类型:描述」**

- **「类型」**

  用于说明 commit 的类别，只允许使用下面7个标识。

- - feat：新功能
  - fix：修补bug
  - docs：修改文档
  - style：格式化代码结构，没有逻辑上的代码修改
  - refactor：重构，即不是新增功能，也不是修改bug的代码变动，比如重命名变量
  - test：增加测试代码，单元测试一类的，没有生产代码的变更
  - chore：构建过程或辅助工具的变动（不会影响代码运行）

- **「描述」**

是本次commit的描述，说明白本次提交都干了些啥

3.2 更新、合并规范

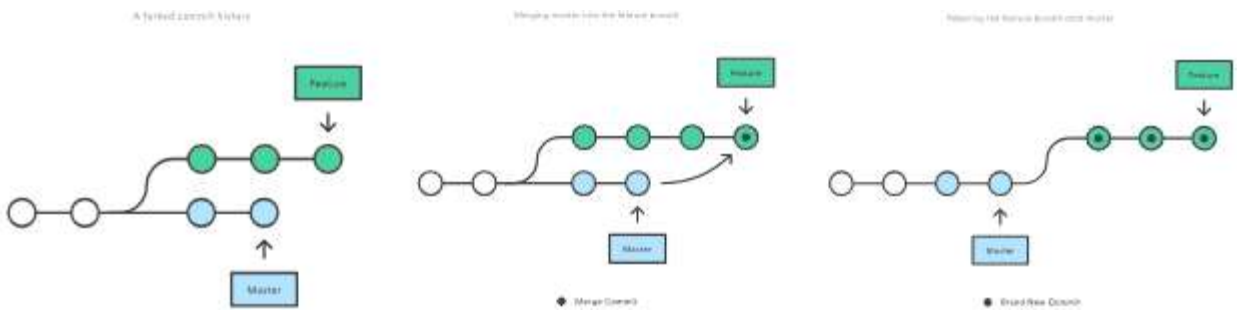
原则：

① 下游分支更新上游分支代码用`rebase`；

② 上游分支合并下游分支代码用`merge`；

③ 更新本分支代码用`--rebase` (如果本分支有多人共同使用开发的时候)；

这样可以消除自动产生的无用`merge`记录，有利于后续查看开发记录。



> 下游分支在更新上游分支代码的时候，如果使用`merge`,会产生一条无用的合并记录，比较影响查看历史，使用`rebase`则不会。

4. 使用技巧

1. 使用命令行代替图形化界面。
2. 提交应该尽可能的表述提交修改内容。
   - 区分 subject 和 body 内容，使用空行隔开
   - subject 一般不超过 50 个字符
   - body 每一行的长度控制在 72 个字符
   - subject 结尾不需要使用句号或者点号结尾
   - body 用来详细解释此次提交具体做了什么
3. 使用 .gitignore 文件来排除无用文件。
4. 不要直接在主干分支上面进行开发。
5. 使用 release 分支和 tag 标记进行版本管理

5. 常用命令

```
//初始化仓库
git init
git init ~/git-server --bare  //初始化一个本地的远程服务器
//对状态的跟踪
git status

//添加文件内容到暂存区（同时文件被跟踪）
git add
//添加所有文件
git add .

//从暂存区提交 -m：注释
git -commit -m 'first commit'
// 从工作区提交
git commit -a -m 'full commit'

git branch <branchName> //创建一个分支
git branch -d <branchName> //删除一个分支
git branch -v //显示所有分支信息

git checkout <branchName> //通过移动HEAD检出版本，可用于切换分支
git checkout -b <branchName> //创件一个分支并切换
git checkout <reference> //将其移动到一个引用
git checkout - //恢复到上一个分支
```

5. 常用命令(附)

```
//git reset   将当前分支回退到历史某个版本
git reset --mixed <commit> //(默认)
git reset --soft<commit>
git reset --hard <commit>
git reflog
// 保存目前的工作目录和暂存区状态，并返回到干净的工作空间
git stash save "push to stash area" // 通过save 后面传入信息标识 放到stash区
git stash list //查看收藏的记录
git stash apply stash@{0} //将保存的内容重新恢复到工作目录
git stash drop stash@{0} //将对应的stash记录删除
git stash pop //= git stash apply + git stash drop

merge fast-forward //默认  不会显示 feature，只保留单条分支记录。git直接把HEAD指针指向合并分支的头，完成合并。属于“快进方式”，不过这种情况如
git merge --no-ff //指的是强行关闭fast-forward方式。可以保存之前的分支历史。能够更好的查看 merge历史，以及branch 状态
git rebase // 修剪提交历史基线，俗称“变基”
git rebase --onto master 5755487

git push // 将本地历史推送到远程
git remote add  origin ~/git-server //添加一个远程仓库的别名
git remote -v //查看远程仓库信息
git fetch  // 获取远程仓库的提交记录
git pull // git pull = git fetch + git merge
git clone //克隆一个远程仓库作为本地仓库
```

谢谢大家！

——Ren Zehua 2022.08.12