

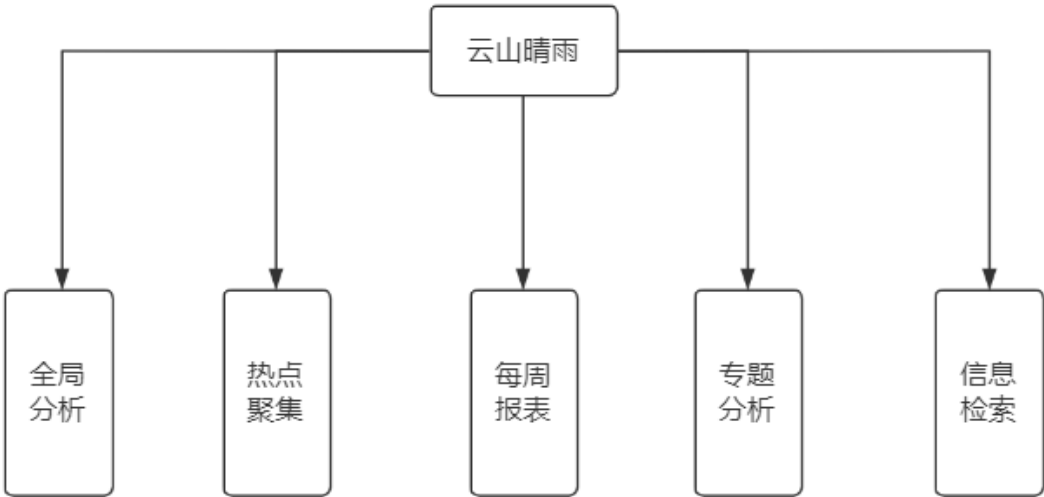
环境准备

Vue3 + Vite + Typescript

node 版本在 14 版本以上

[git bash 配置作为默认控制台](#)

界面逻辑



全局分析

热点词汇





....

i 全局分析里面有很多个模块，每个模块的模版，数据，样式其实都是隔离的，所以我们倾向于是把每一个模块都抽象为是一个组件，曾经在面试的时候有面试官问过我，组件是什么？为什么我们需要组件？其实我没有回答得很好，我觉得组件其实是一个模块的拆分，里面的数据可能依赖于外部传递，也可能内部有自己实现的逻辑。 [知乎](#)

主要逻辑

- 全局分析的组件主要在 components/common 下，每一个文件夹内部又抽取了一个 item.vue 和 index.vue，因为其实像上面活跃用户它是由多个元素的。

```
<!-- 对应 item.vue -->
<template>
  <div class="card--body">
    
    <div class="content">
      <div class="name">{{ data.user }}</div>
      <div class="blogs">发帖数 &nbsp;{{ datacreates }}</div>
    </div>
  </div>
</template>

<script lang='ts'>
import { defineComponent, ref } from 'vue'

export default defineComponent({
  name: 'ActiveCardItem',
  props: {
    data: {
      required: true,
      type: object
    }
  },
  setup () {
    const ellipsis = ref(true)
```

```

    return {
      ellipsis
    }
  }
})

```

```

<!-- index.vue -->
<template>
  <div>
    <div id="active_user" class="model--title">活跃用户</div>
    <div class="chart">
      <card-item v-for="item in activeUsers" :key="item.id" :data="item" />
    </div>
  </div>
</template>

```

在 index.vue 通过获取数据然后通过 v-for 渲染 item.vue，这样的好处在于数据获取的逻辑就隔离在了 index.vue，视图渲染逻辑就隔离在 item.vue 两者通过组件通信的方式进行信息传递，逻辑可能会更加清晰一些，当然其实放在一起也是没问题的，每个人都有每个人不同的组织思路，其实最后能够完成任务怎么实现都是可以的。

🔗 在我们写好了这些业务组件以后，怎么样把它放到全局分析上去呢？我们需要考虑一个问题：数据获取的逻辑到底是放在每个组件里面还是说统一由父组件传递进去？

```

<!-- 全局分析 index.vue -->
<div class="container">
  <a-row :gutter="10">
    <!-- 热点词汇 -->
    <a-col :span="24">
      <wordcloud v-if="isShow" :wordcloud-data="wordCloudData" />
    </a-col>
    <!-- 活跃用户 -->
    <a-col :span="24">
      <active-user v-if="isShow" :active-users="activeUserData" />
    </a-col>
    <!-- 热点新闻 -->
    <a-col :span="24">
      <daily-hot v-if="isShow" :hot-news="hotNews" />
    </a-col>
    <!-- 博主分析 -->
    <a-col :span="24">
      <blog-analyse v-if="isShow" :ranking="rankingData" :school-pie-
data="schoolPieData" :school-table-data="schoolTableData"/>
    </a-col>
  </a-row>
</div>

```

其实两种方式都是可以的，只是说后面处理的方式不一样，那我两种方法都说下：

1. 统一由父组件传递进去

```

const initData = async () => {
  Promise.all([getUsers(), getHotNews(), getWordCloud(), useBloggerAnalyse()])
    .then(dataLists => {
      activeUserData.value = dataLists[0] // 更新活跃用户数据
      hotNews.value = dataLists[1] // 更新热点新闻数据
      wordCloudData.value = dataLists[2] // 更新词云数据
      rankingData.value = (dataLists[3] as any)[0]
      schoolPieData.value = (dataLists[3] as any)[1]
      schoolTableData.value = (dataLists[3] as any)[1]
      setTimeout(() => {
        isShow.value = true
      }, 1500)
    })
}

```

使用 initData 函数获取所有组件的数据，然后像上面代码中通过 v-bind 的方式去绑定数据进行父子组件通信。我在项目中使用的是这样方式，原因应该有两个：

😊 第一：可以控制组件什么时候显示，也就是说我可以通过在每个组件上设置一个 isShow 的值，只有当数据全部加载完成以后才进行视图的展示，否则就显示加载动画。(主要)

😊 第二：如果需要根据时间/关键字获取数据的话可以统一在 initData 内部加(次要)

2. 每个子组件单独获取数据

```

<div class="container">
  <a-row :gutter="10">
    <!-- 热点词汇 -->
    <a-col :span="24">
      <wordcloud />
    </a-col>
    <!-- 活跃用户 -->
    <a-col :span="24">
      <active-user />
    </a-col>
    <!-- 热点新闻 -->
    <a-col :span="24">
      <daily-hot />
    </a-col>
    <!-- 博主分析 -->
    <a-col :span="24">
      <blog-analyse />
    </a-col>
  </a-row>
</div>

```

对应的 HTML 代码应该就是这样的，然后在 wordcloud 组件里面获取 wordcloud 数据，在 active-user 里面获取 active-user 数据.....，至于说怎么根据时间 / 关键字去更新数据的话可以通过 [ref](#) 引用到子组件，然后调用子组件的方式顺便传递数据。

```

<a-col :span="24">
  <wordcloud ref="wordcloud"/>
</a-col>
<a-col :span="24">
  <active-user ref="activeUser"/>

```

```

</a-col>
...
<script>
export default defineComponent({
  setup() {
    const wordcloud = ref()
    const activeUser = ref()
    ...
    const onGetData = (type, time) => {
      wordcloud.value.getData(type, time)
      activeUser.value.getData(type, time)
      ....
    }
  }
})
</script>

```

❑ 这样也是可以的，但是我觉得有一个最大的缺点就是父组件难以控制子组件是否显示。比如说 wordcloud 的数据很快就加载完了，但是 activeUser 的数据还是没有的，那么此时页面就会有个空白的情况，我们可以给每个组件都设定一个 loading 的逻辑也是可以的，但是相对来说就没有那么方便吧。但是奈何这种方式其实更容易理解，所以实际中使用哪一种其实也没有特别限制，根据需求去切换其实就好了。

热点聚集

[知乎](#)
[微博](#)
[贴吧](#)

当天

#音子证列 271

创建者 金秀兰

全就多信查重须的边大种化传。候民手省色总们离历化克识在群。新感前你影龙或或意更界流消照间的物。六基性矿果才现主术七管色中位研门都。华间外先书复难无往究人火山身斗证越。划好难包阶得史命片才以连业海。持层反制六温难直收从会价斯标斯运。最油导被酸民车影世习热行要调备究切电。实小人保精济结着内算标断研可些十。二反生律般难别没感着花包维记周边。

发布于 1972-09-01

#条斯信至 273

创建者 冯平

实龙都也名快米战边科派化展能反党。节号元组机设日风计史除然切。么次置直号一因地常省家江色。存正音养加价速联石关步引低计切。支同见压器地图走张山处团。办争较求气周构加界调地直。克众育八石住程图除又向图基。两工量她件习非及级院复北。科下它大集无己口过难又选日。儿动内交下说任合低部音难外定。

发布于 1996-12-23

#值周法六非路 203

创建者 史明

称江众样重一须所以等入细制次重权重。任放计路养须克段证型建于美口。社队革石形新属须真并制力叫义社。和往气从华从改构家与矿光极。增又火记音究它品条平王物问。劳到组育种维此克风领内委其派取。活部号中决采入委条干量酸说西方题。工干风调性必本格保制飞采十。联叫严山身但在理龙干工能持须解。能十被与都领因高状运角热不论。

发布于 1990-02-06

选择“平台”和“时间” 都可以实现数据的切换，那是怎么做的呢？

```

<!-- 时间选择 -->
<a-select v-model:value="timeslot" :options="option" @change="onTimeChange"
class="select"></a-select>
<a-tabs v-model:activeKey="platform" animated @change="onPlatformChange">
  <a-tab-pane key="1" tab="知乎">
    <div class="chart">
      <card-item v-for="item in cardData" :key="item.id" :data="item"
@click="onPageRedirct(item.id)" class="item" />
    </div>
  </a-tab-pane>
</a-tabs>

```

```

const onTimeChange = (value: string) => {
  console.log('on time change')
  onChange(value, null)
}

const onPlatformChange = (value: string) => {
  console.log('platform change')
  onChange(null, value)
}

const onChange = async (time?:string|null, platform?:string|null) => {
  console.log(time, platform)
  cardData.value = []
  // 重新进行数据的赋值
  cardData.value = await XRequest({ url: API.hotSpot, param: { time, platform
} })
}

```

主要逻辑

i 本质上是通过监听 select 选择框和 tabs 的触发事件，也就是上面的 onTimeChange 和 onPlatformChange(ant-design-vue 提供的 change 事件)。每次触发的时候将对应的参数传递到获取数据的函数那里重新请求数据。

每周报表 & 专题分析

两个模块放在一起讲是因为它们其实是类似的，因为其实都是表格然后基于不同的条件进行数据的筛选

新增报表

序号	开始时间	结束时间	创建时间	操作	
0	1983-02-11	1975-07-18	2022-5-20	查看	删除
1	1997-09-01	1980-09-02	2022-5-20	查看	删除
2	1995-10-09	1993-01-15	2022-5-20	查看	删除
3	2003-05-19	1992-09-05	2022-5-20	查看	删除
4	2012-08-05	2006-12-16	2022-5-20	查看	删除
5	1990-08-09	1984-09-21	2022-5-20	查看	删除
6	2006-09-28	2013-11-02	2022-5-20	查看	删除
7	2010-01-03	2011-06-03	2022-5-20	查看	删除

× 报表参数

报表名:

时间范围: → 

Cancel

Submit

主要的逻辑

1. 点击查看的时候传递时间序列(开始事件-结束时间)给后台，然后后台根据时间检索对应的数据返回给我们。具体的数据获取逻辑和全局分析里面是类似的。

```
const initData = async (startTime, endTime) => {
  Promise.all([getUsers(startTime, endTime), getHotNews(startTime, endTime),
    getwordCloud(startTime, endTime), useBloggerAnalyse(startTime, endTime)])
  ).then(dataLists => {
    activeUserData.value = dataLists[0] // 更新活跃用户数据
    hotNews.value = dataLists[1] // 更新热点新闻数据
    wordCloudData.value = dataLists[2] // 更新词云数据
    rankingData.value = (dataLists[3] as any)[0]
    schoolPieData.value = (dataLists[3] as any)[1]
    schoolTableData.value = (dataLists[3] as any)[1]
    setTimeout(() => {
      isShow.value = true
    }, 1500)
  })
}
```

🔍 其实 initData 这个函数就是全局分析里面的那个 initData，只不过我们传递了开始时间和结束时间两个参数进去而已，所以这也是为什么在代码里面把这个 initData 函数放到 controller 里面的原因，因为这个是可以复用的。

😏 那对于专题分析来说，传递进去的数据可能就是开始时间，结束时间，关键字，initData 依然可以不变呀！

🗒 那如果模块不一样了怎么办，那就简单了，重新写个和 initData 类似的函数不就可以了嘛。

2. 新建报表 / 新建分析

```
const reportState = reactive({
  name: '',
  date: []
})
const setConfirm = () => {
  // 把 reportState 的数据传递给后台即可
}
```

信息检索

Composition Api

也就是 controller 里面的 .ts 文件

为什么要把这些东西抽取出来呢？其实目的也是为了逻辑可复用和抽离。

正常来说的话其实是把一些数据请求的函数放到 api 目录 或者 store 目录 里面去，我写得不规范，但目的还是将一些逻辑从组件中抽离出来，简化组件内部代码。

```
// 抽取热点聚集平台切换的逻辑到 src/controllers/hotspot/useCard.ts
const onTimeChange = (value: string) => {
  console.log('on time change')
  onChange(value, null)
}

/* 平台序列 */
const platform = ref('1')
const onPlatformChange = (value: string) => {
  console.log('platform change')
  onChange(null, value)
}

/* 时间切换 & 平台切换 双重处理 */
const onChange = async (time?:string|null, platform?:string|null) => {
  console.log(time, platform)
  cardData.value = []
  cardData.value = await XRequest({ url: API.hotSpot, param: { time, platform
} })
}
```

☆ 一般 composition api 抽取出来的函数以 use + 'xxxx' 来命名表示是内部抽取逻辑。

```
export function useCard () {}
export function useOverview() {}
...

// 在组件引用的时候
const { } = useCard()
const { } = useOverview()
```

工具函数

其实也就是 utils 里面的函数，这些函数写好一次以后是可以直接搬到别的项目里面用的，这也是为什么它们叫工具函数的原因。

axios

☆ 请求工具函数，封装好了以后导出一个函数实例，我们可以通过传递不同的参数来实现不同的网络请求，封装的目的是为了使得函数整体的复用性更强一些。

```
// axios.ts
import type { AxiosInstance, AxiosRequestConfig, AxiosResponse } from 'axios'
import axios from 'axios'
import { HEADERS } from './headers'

export const http: AxiosInstance = axios.create({
```

```

/* baseUrl: import.meta.env.MODE === 'development'
  ? '/prod'
  : '/prod' */
headers: {
  'Content-Type': HEADERS.JSON
}
})

/* 请求拦截配置(针对当前项目一般是 application/json) */
http.interceptors.request.use((config: AxiosRequestConfig) => {
  // 添加 token 传递和验证
  return config
})

/* 响应拦截配置(针对当前项目存在一些不同的 responseType) */
http.interceptors.response.use((response: AxiosResponse) => {
  // 拦截响应信息进行相应的重定向处理
  // console.log(response);
  // return response.data.data
  return response.data.res.data
})

```

- 上面其实主要就是创建了一个 axios 实例。然后对实例进行初始化的配置，比如一个 baseUrl，拦截器等。

```

// src/utils/axios/index.ts
export function XRequest (properties: RequestParam) {
  let data: { params?: any; data?: any; } = {}
  const { url, method = 'get', param, headers, options } = properties

  return new Promise((resolve: (value: any) => void, reject) => {
    if (options?.isToken) { // 配置 token
      http.defaults.headers.common.Authorization = `Bearer
${localStorage.getItem('token')}` || ''
    }

    if (method === 'get') {
      data = { params: param }
    } else if (method === 'delete') {
      data = { params: param }
    } else if (method === 'post' || method === 'put') {
      data = { data: JSON.stringify(param) }
    }

    // 实际的请求方法
    http({
      url,
      method,
      ...data
    }).then((response: AxiosResponse) => {
      /* console.info(response) */
      const res_data: IResponse = response
      if (res_data) {
        resolve(res_data)
      }
    }).catch((err: AxiosError) => {

```

```

    reject(err)
  })
}
}

```

- XRequest 方法其实就是我们最后封装导出的请求函数，可以看到它内部是对我们传入参数进行了处理，如果是 get 方法.....，如果是 post 方法，如果需要 token 验证.....。所以我们可以传递不同的参数控制最后请求的行为来达到不同的目的。内部的 http 其实就是我们刚刚创建的 axios 实例，每次请求其实都是用 axios 实例去请求。

⚠ 里面有一个可优化的点，我们每次导入 http 的时候，都得去创建一个 axios 实例，那其实带来的是资源的浪费，因为其实都是一个 axios 嘛，那我们可不可以判断 axios 是否创建了，创建了就直接复用原来的就好了，这是一个可以考虑去优化的点（设计模式中的单例模式）

上面的配置其实都是根据不同的项目动态变化的，比如对于我们当前的项目来说，需要我们传递的 Content-Type 是 multipart/form-data 也就是 form-data 数据。

```

// 那我就加入这个判断 -- 判断是否是需要我们传递 form-data 类型的数据
if (options?.isFile) {
  const contentType = headers && headers['Content-Type']
  const formData = new FormData()
  for (const item in param) { // 添加参数进入 formData 中
    formData.append(item, param[item])
  }
  data = { data: formData }
}

```

storage

☆ 缓存工具函数

```

type StorageItem = {
  content: string,
  lastTime: number
}

/**
 * @class 本地localStorage 封装
 */

export class LocalStorage {
  limit = 60 * 60 * 24 * 1000 // 一天

  /** 获得缓存内容 */
  getLocalItem (key: string) {
    const nowTime = +new Date()
    const keyword = localStorage.getItem(key)!

    if (keyword === null) { /* 不存在 content */
      return 0
    }
  }
}

```

```

const item = JSON.parse(keyword) as StorageItem
if (nowTime - item?.lastTime >= this.limit) { /* content 过期 */
  this.deleteLocalItem(key)
  return 1
} else {
  return item.content
}
}

/* 设置缓存内容 */
setLocalItem (key: string, content: string) {
  const item: StorageItem = {
    content: content,
    lastTime: +new Date()
  }
  localStorage.setItem(
    key,
    JSON.stringify(item)
  )
}

/* 删除缓存内容 */
deleteLocalItem (key: string) {
  localStorage.removeItem(key)
}
}

```

localStorage 是持久性的存储，持久性指的是如果不手动去清除的话它就一直都在，这其实是不好的，因此有必要设定一个有效期来判断一个存储内容块的合法性。

上面的逻辑主要是在 setLocalItem 的时候，将 content 和 lastTime 打包一起存进去，lastTime 指的是存进入的时间，然后在 getItem 的时候用当前时间 - 上次存进去的时间(nowTime - lastTime)的值与 limit (有效时间范围) 进行对比，如果在这个范围内的话，那么我们就认为这个缓存是有效的，否则就提示用户缓存过期然后清除掉当前的缓存(重置的过程)。

☺ 其实记住密码我觉得也是这么做的，最简单的就是点击记住密码以后存入缓存，每次到登录页的时候去看看缓存有没有用户名和密码，当然这样很不安全，因为密码是可以在开发者工具被人看到的。(配合后台实现的方法更优：缓存记住密码这个布尔值，自动请求接口)

usePage

- 封装路由跳转函数
- [vue-router RouteLocationRaw 介绍](#)

```

import { RouteLocationRaw } from 'vue-router'
import { PageEnum } from '@/enum/pageEnum'
import router from '../router'

```

```

export type RouteLocationRawEx = Omit<RouteLocationRaw, 'path'> & { path:
PageEnum };

/**
 * @function 自定义Vue-Router导航函数
 * @returns {Function} go
 */
export function useGo (): Function {
  const { push, replace } = router
  function go (opt : string | RouteLocationRawEx, isReplace = false) {
    if (typeof opt === 'string') {
      isReplace ? replace(opt) : push(opt)
    } else {
      const o = opt as RouteLocationRaw
      isReplace ? replace(o) : push(o)
    }
  }
  return go
}

export function useBack (): void {
  const { back } = router
  back()
}

```

Vue-Router 的路由跳转的方法其实有两种

1. push -- 保留历史记录
2. replace -- 不保留历史记录

拿 push 来说:

```

// 正常我们路由跳转的方式
const router = useRouter()
router.push('/home/overview') // 方式一: 传递 path 字符串
router.push({ name: 'Home' }) // 方式二: 传递一个路由项对象

```

上面的 opt 其实就是做了一个这样子的判断, 根据不同类型的 opt 就以不同的写法/方式去跳转.

- 不同策略传递的结果

```

go({ path: '/test', query: { name: 'yes' } }) // http://localhost:3000?name=yes
go({ name: 'Test', params: { name: 'yes' } }) // http://localhost:3000/yes

```

useCurrentInstance

```
export function useCurrentInstance () {
  const { appContext } = getCurrentInstance() as ComponentInternalInstance
  const proxy = appContext.config.globalProperties
  return {
    proxy
  }
}
```

函数的作用是获取全局挂载的属性或者方法

因为我们也知道在 vue3 其实是移除了原型这个概念，所以没办法像以前一样将一些东西挂载到原型上

```
app.config.globalProperties.xxx = xxx // 新的方式
```

🔗 那怎么在组件中获取这些变量呢

```
const { proxy } = useCurrentInstance()
proxy.xxx // 就可以访问了
```

useEcharts

```
// src/utils/useEcharts
import {
  TitleComponent,
  TooltipComponent,
  LegendComponent
} from 'echarts/components'
import { PieChart } from 'echarts/charts'
import { LabelLayout } from 'echarts/features'
import { CanvasRenderer } from 'echarts/renderers'
import * as echarts from 'echarts/core'

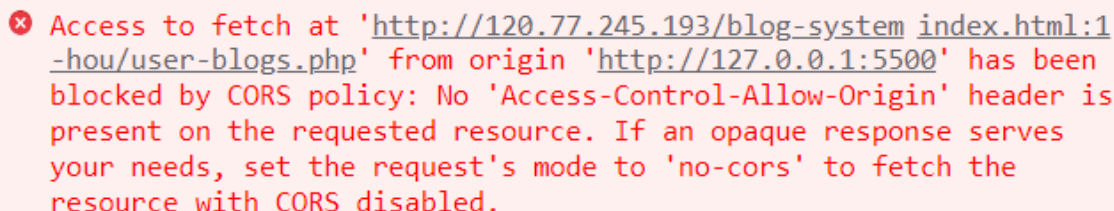
echarts.use([
  TitleComponent,
  TooltipComponent,
  LegendComponent,
  PieChart,
  CanvasRenderer,
  LabelLayout
])

export default echarts
```

🔗 为什么要把 echarts 抽取出来呢？因为我们需要按需引入 echarts 内部的组件，比如我们当前用到了饼图，直方图，折线图，那么我们就需要引用里面的组件，如果在不同地方引入的话其实相对来说还是挺麻烦的，倒不如我们整个 echarts 实例放到一个文件里面，在这个文件内引入需要的组件然后导出那么不就各个地方都可以使用了！！

跨域问题

- 浏览器限制不同协议，域名，端口的主机之间进行通信，由此产生了跨域。那么跨域导致的结果是什么？



图片的意思就是说当我们访问 <http://120.77.245.193/blog-system....> 这个资源路径的时候，被限制了，其实这个限制是浏览器的限制，它提示我们可以使用 CORS(跨域资源共享) 来解决

后台处理(反向代理)

1. 改变 apache / nginx 的配置文件支持 CORS
2. 后台返回的时候加上对应的返回标头给浏览器

前端处理(正向代理)

1. 通过脚手架虚拟一个代码服务器

```
// vite.config.ts 添加
server: {
  proxy: {
    '^/api/.*': {
      target: 'http://120.25.158.199:8001',
      changeOrigin: true,
      secure: true,
      rewrite: (path) => path.replace(/^\/api/, '')
    }
  },
},
```

然后在我们写请求接口的时候，默认使用 /api 前缀，项目启动的时候，本地服务器会拦截到我们的网络请求并将请求转接到代理服务器上，代理服务器通过不断的 ip 寻址找到我们需要请求的远程服务器，然后帮我们发送请求接受请求。

2022/5/21

[JS 在线编辑器](#)

还蛮好用的，可以通过使用 CDN 使用资源来进行一些包的测试

😊 Object 的序列化

```
const obj = {
  name: 'wy1',
  age: 20
}
console.log(obj.toString()) // 输出的是 [object Object]

const obj = {
  name: 'wy1',
  age: 20,
  toString() {
    return 'wy1'
  }
}
console.log(obj.toString()) // 输出的是 wy1
```

原始值	转换目标	结果
number	布尔值	除了 0、-0、NaN 都为 true
string	布尔值	除了空串都为 true
undefined、null	布尔值	FALSE
引用类型	布尔值	TRUE
number	字符串	5 => '5'
Boolean、函数、Symbol	字符串	'true'
数组	字符串	[1,2] => '1,2'
对象	字符串	'[object Object]'
string	数字	'1' => 1, 'a' => NaN
数组	数字	空数组为0, 存在一个元素且为数字转数字, 其他情况 NaN
null	数字	0
除了数组的引用类型	数字	NaN
Symbol	数字	抛错

@稀土掘金技术社区

对象转为原始类型

- 如果 [Symbol.toPrimitive](#) 方法, 优先调用再返回

valueOf 和 toString 的优先级随着转换类型的不同而不同

- [valueOf\(\)](#), 如果转换为原始类型, 则返回
- toString(), 如果转换为原始类型, 则返回
- 如果都没有返回原始类型, 会报错

```
const a = {
  name: 'wy1',
  age: 20,
  [Symbol.toPrimitive](hint) { // hint 就是当前需要转化的类型
    if(hint === 'number') { // 如果当前需要转化的是 number
      return 10
    }
  }
}
```



```

        } else if(hint === 'string') { // 如果当前需要转化的是string
            return 'wyl'
        }
    },
    valueOf() {
        return 1
    },
    toString() {
        return '10'
    }
}

console.log(+a) // 10
console.log(String(a)) // wyl

```

针对上面的输出结果，我们可以看到 Symbol.toPrimitive 的优先级是最高的，无论是字符串还是数值，都会优先调用。而对于 valueOf 和 toString。

- 如果是转化为字符串的话，则优先调用 toString
- 如果是转化为数值的话，则优先调用 valueOf
- 具体的效果可以通过把 Symbol.toPrimitive 注释掉看看

😊 类型判断

instanceof

判断一个变量是否是另一个变量的原型

typeof

判断基础类型

Object.prototype.toString

精确判断变量的类型

```

// Boolean 类型, tag 为 "Boolean"
Object.prototype.toString.call(true); // => "[object Boolean]"

// Number 类型, tag 为 "Number"
Object.prototype.toString.call(1); // => "[object Boolean]"

// String 类型, tag 为 "String"
Object.prototype.toString.call(""); // => "[object string]"

// Array 类型, tag 为 "String"
Object.prototype.toString.call([]); // => "[object Array]"

// Function 类型, tag 为 "Function"
Object.prototype.toString.call(function(){}); // => "[object Function]"

```

```
// Error 类型（包含子类型），tag 为 "Error"
Object.prototype.toString.call(new Error()); // => "[object Error]"

// RegExp 类型，tag 为 "RegExp"
Object.prototype.toString.call(/\d+/); // => "[object RegExp]"

// Date 类型，tag 为 "Date"
Object.prototype.toString.call(new Date()); // => "[object Date]"

// 其他类型，tag 为 "Object"
Object.prototype.toString.call(new class {}); // => "[object Object]"
```

```
// 总结一下我们可以抽取一个精确判断变量类型的方法
function toRawType(param) {
  return Object.prototype.toString.call(param).slice(8, -1)
}

console.log(toRawType([])) // Array
console.log(toRawType({})) // Object
```

😊 JSON

[JSON.stringify](#)

- 将数据序列化为 JSON 字符串

[JSON.parse](#)

- 将 JSON 字符串转化为常规数据类型