Vue 2.0 真实点餐项目实战

一次点餐的项目的历程让我对vue有了更深的理解，进行了一次重构对整个项目结构也有深刻的体会。

通过项目的结总，讲以前重要的几点：

1. 如何正确的利用子路由，去优化 ajax 请求
2. 如何通过 vux 正确的把控大数据量的流动和把控
3. 用哪些技巧可以进行重点性能优化
4. 如何对项目迭代时的把控
5. 正确的对项目组件划分，层级划分，木偶组件，智能组件
   在项目中如何慢慢的结合项目，为项目定制一套可维护，通用性的组件库
6. `$attrs` `$listeners` ,使用占位符，局部加载等等
7. 如何正确的使用 mixin 去混合代码，实现最大重复利用

> 合理的运用子路由，避免ajax请求

在路由层面，往往对路由层面理解不够深的开发者来说，子路由感觉可能就是一个摆设，通常都是一个页面一个路由，此时就会出现了明显的问题，每当切换到下个路由或者返回到前一个路由，都会重新执行生命周期，大多数数据的来源都是在 `created` 里去请求的。

> 子路由的原理

先说说子路由的原理，子路由原理则是拿到单个.vue文件的实例，通过append节点到父路由设定的dom节点里，当切换路由的时候，则是把父节点插入的子路由的节点再进行

```
]
}
```

从上面一个基本的路由可以看出，DishDetail是Menus的子路由，对于点餐项目，Menus是点餐的主页面，DishDetail是菜单详情页Menus通常进入页面的时候几乎所有的数据向后台请求都是经过这个页面进行操作，而我做的云客官项目，此页面向后台发送了五个数据请求，只能说这是避免不了的请求
DishDetail是菜单详情页，同样也是用户频繁进行的页面，操作率还是很高的，为了避免重复发送ajax请求，设定DishDetail为Menus的子路由。

当进入DishDetail路由的时候，只是把DishDetaili当作dom插入Meuns页面的中，DishDetail会执行自己的生命周期，返回到Menus路由的时候，DishDetail路由只是把自己当Menus路由中移除，同时destory掉自己。

无论前进还是后退，Meuns路由并没有销毁，也没有加载。

```
{
path:'menus',
components:Menus,
},
{
path: 'dishDetail',
components:DishDetail
}
```


DishDetail和Menus是同级路由，通过Menus路由进入到Detail路由的时候，两者都发生了变化，Menus路由则是进行销毁，加载进来的是DishDetail路由，一旦路由进行销毁再重新加载此路的时候，都需要重新执行生命周期，如何不通过子路由来加载DishDetail的话，每次进入详情页再返回之后，都会重新执行Menus的生命周期，执行五次AJAX请求，能避免HTTP请求则避免。

> 智能组件和木偶组件的划分

- components //存放智能组件
  common.vue
- page //存放页面
  - Menus //菜单页面
    header.vue
    body.vue
    footer.vue
  - othersPage //其它页面
    header.vue
    body.vue
    footer.vue

把智能组件存在components文件夹中，当前页面所拆分出的组件，则放在当前页面的文件夹下，这个有便于后续文件庞大了，组件拆分的细了，这样就很容易的快速定位到组件的位置。

> 智能组件和木偶组件传递的要素

智能组件

在第三方组件库中，都是向外暴露event和props两个接口参数，这个智能组件类似，智能组件主要服务于多个页面，所以对向外暴露接口的同时也需要规范性。

props规范性和event规范性，因为对于智能组件来说，永远都不知道在位于那个页面或者可能欠到到第几层使用，要明确的进行props原子化进行传递，数据双向绑定同时也要以接口的形式向外暴露。

木偶组件

木偶组件在页面拆分时，可以明确的知道组件位于第几层，也只会服务指定页面，可以使用 `$parents` 和 `$children` 这种快捷方便的模式进行数据交互和行为交互，不用考虑

3. 大方向单个组件内小规模改动
4. 组件库的暴露的接口不符合业务需求

对于面向C端的产品必然是做出自己的特色，所以自己掌握一套组件库的应用，这个是必然的抉择。

> 如何方便快速稳健的搭建一套基本的组件库。

首先组件库必然的两点：

- 样式组件
- 功能组件

一个基本的组件库样式组件可以让开发布局flex布局grid布局对做项目来说会显的更加轻松，对于基本的功能组件不必要忙着把所有市面上所见的全部造出来，组件库和业务要并行，设计图拿到手之后，看设计图用到的那些通用型组件，把先用到的造出来，不需要的留一边，等项目结束后再造。

> 借鉴组件库，不要往死想

大量第三方组件库都是一个团队，经过大量的测试和用户反馈，兼容性测试进行发布的，无论组件的设计模式和接口暴露的方式，技术都是比较领先的，可以前先大厂商中的组件库中吸收一些组件库的写法和思想，可以先考虑把第三方的组件好的写法拿过来，还有很重要的一点是，第三方的组件库对于单个组件都尽量做到很全面，暴露了很多接口，一开始只需要根据自己的项目要求暴露部分接口。

> 不要一味着造组件库

像一些比较不容易理解的组件或者自己还没有能力去吸收和改造 的情况下，我建议还是暂时使用第三方的，比如说一些swiper这种比较复杂的组件，也是项目通用型组件，在快速完成项目和人手不够的情况 下，建议暂时使用第三方的，不要浪费时间去研究组件

页面的共享的话,用这两个属性非常便捷。

组件与组件之间大胆解耦

有些开发者，特别对vuex没有深入理解和实战的时候,同时对组件与组件多层传递时,不敢大胆的解耦组件,只能进行到父子组件这个层面，而且组件复用率层面上也有所下降

`$attr` 与 `interitAttrs` 之间的关系

> interitAttrs :

默认情况下父作用域的不被认作 props 的特性绑定 (attribute bindings) 将会“回退”且作为普通的 HTML 特性应用在子组件的根元素上。当撰写包裹一个目标元素或另一个组件的组件时，这可能不会总是符合预期行为。通过设置 inheritAttrs 到 false，这些默认行为将会被去掉。而通过 (同样是 2.4 新增的) 实例属性 `$attrs` 可以让这些特性生效，且可以通过 v-bind 显性的绑定到非根元素上。

注意：这个选项不影响 class 和 style 绑定。

what?

官网上并没有给出一点demo,语意上看起来还是比较官方的,理解起来总是有点不太友好,通过一些demo来看看发生了什么。

子组件

```
<template>
   <div>
```

```
<template>
  <div class="hello">
     <demo :first="firstMsg" :second="secondMessage"></demo>
  </div>
</template>

<script>
  import Demo from './Demo.vue'
  export default {
    name: 'hello',
    components: {
      Demo
    },
    data () {
      return {
        firstMsg: 'first props',
        secondMessage: 'second props'
      }
    },
  }
</script>
```

父组件在子组件中进行传递firstMsg和secondMsg两个数据，在子组件中，应该有相对应的Props定义的接收点，如果在props中定义了，你会发现无论是firstMsg和secondMsg都成了子组件的接收来的数据了，可以用来进行数据展示和行为操作。

虽然在父组件中在子组件模版上通过props定义了两个数据，但是子组件中的Props只接收了一个，只接收了firstMsg,并没有接收secondMsg,没有进行接收的此时就会成为子组件根无素的属性节点。

事件代理

当我们用v-for渲染大量的同样的DOM结构时，但是每个上面都加一个点击事件，这个会

```
<script>
  import Demo from './Demo.vue'
  export default {
    name: 'hello',
    components: {
      Demo
    },
    data () {
      return {
        firstMsg: 'first props',
        secondMsg: 'secondMsg'
      }
    },
    methods: {
      ff (e) {
        if(e.target.dataset.second == 'secondMsg') {
            console.log('通过事件委托拿到了自定义属性')
        }
      }
    }
  }
</script>
```


经过改动之后，在父组件中，把向子组件传递的参数名改成了html自定义的data-second属性，同样在子组件中不进行Props接收，就顺其自然的成为了子组件每一个根节点的自定义属性。

通过事件冒泡的原理，然而可以从e.target.dataset.second就能找对应的dom节点进行逻辑操作。

同样，在子组件模版上可以绑定多个自定义属性，在子组件包裹的外层进行一次监听，通过data自定义属性拿到循环出来组件的对应的数据，进行逻辑操作。

interitAttrs = false 发生了什么 ？

对子组件进行一个改动，我们加上inheritAttrs: false，从字面上的翻译的意思，取消继承的属性，然而props里仍然没有接收seconed，发现就算Props里没有接收seconed，在子组件的根元素上并没有绑定任何属性。

```
$attrs
```

包含了父作用域中不被认为 (且不预期为) props 的特性绑定 (class 和 style 除外)。当一个组件没有声明任何 props 时，这里会包含所有父作用域的绑定 (class 和 style 除外)，并且可以通过 v-bind=" `$attrs` " 传入内部组件——在创建更高层次的组件时非常有用。

在前面的例子中，子组件props中并没有接受seconed，设置选项inheritAttrs: false，同样也不会作为根元素的属性节点，整个没有接收的数据都被$attr实例属性给接收，里面包含着所有父组件传入而子组件并没有在Props里显示接收。

为了验证事实，可以在子组件中加上：

```
created () {
        console.log(this.`$attrs`)
    }
```

打印出来则是一个对象：{second: “secondMsg”, third: “thirdMsg”}

注意

想要通 `$attr` 接收，但必须要保证设置选项inheritAttrs: false，不然会默认变成根元素的属性节点。

开头说了，最有用的情况则是在深层次组件运用的时候。创建第三层组件，作为第二层组件的子组件,在子组件引入的孙子组件，在模版上把整个$attr当数作数据传递下去，中间则并不用通过任何方法去手动转换数据。

子组件：

```
<template>
  <div>
      {{second}}{{third}}
  </div>
</template>

<script>
  export default {
     props : [ 'second' , 'third']
  }
</script>
```

孙子组件在props接收子组件中通过 `$attr` 包裹传来的数据，同样是通过父组件传来的数据，只是在子组件用了 `$attrs` 进行了统一接收，再往下传递，最后通过孙子组件进行接收。

依次类推孙子组件仍然不想接收，再传入下级组件，我们仍然需要对孙子组件实力选项进行设置选项inheritAttrs: false，否则仍然会成为孙子组件根元素的属性节点。

从而利用 `$attrs` 来接收props为接收的数据再次向下传递是一件很方便的事件，深层次接收数据我们理解了，那从深层次向层请求改变数据如何实现。意思就是让顶层数据和最底层数据进行一个双向绑定。

`$listeners`

listeners可以认为是监听者。

向下如何容易的传递数据已经了解了,面临的问题是如何向顶层的组件如何改变数据,父子组件可以通过v-model,.sync,v-on等一系列方法，深层及的组件可以通过 `$listeners` 去管理。

`$listeners` 和 `$attrs` 两者表面层都是一个意思， `$attrs` 是向下传递数据，`$listeners` 是向下传递方法，通过手动去调用 `$listeners` 对象里的方法，则原理就

```
<script>
  import Demo from './Demo.vue'
  export default {
    name: 'hello',
    components: {
      Demo
    },
    data () {
      return {
        firstMsg: '父组件',
      }
    },
    methods: {
      changeData (params) {
         this.firstMsg = params
      },
      another () {
        alert(2)
      }
    }
  }
</script>
```

在父组件中引入子组件,在子组件模板上面进行changeData和another两个事件监听，其它这两个监听事件并不打算被触发，而是直接被调用,再简单的理解则是向下传递两个函数。


```
<template>
   <div>
      <p @click="$emit('another')">子组件</p>
      <next-demo  v-on='`$listeners`'></next-demo>
   </div>

</template>
<script>
import NextDemo from './nextDemo.vue'
```

在子组件中，引入孙子组件nextDemo,在子组件中,像 $attrs 一样,可以用 $listeners 去整体接收监听的事件, {changeData: ƒ, another: ƒ} 以一个对象去接收,此时在父组件中在子组件模板上监听的两个事件不但可以被子组件实例属性 $listeners 去整体接收,并且同时可以在子组件进行触发。

同样的在孙子nextDemo组件中，继续向下传递,通过v-on把整个 $listeners 所接收的事件传递到孙子组件中,只是通过 $listeners 把其所有在父组件拿到监听事件一并通过 $listeners 一起传递到孙子组件里。

孙子组件

```
<template>
  <div class="hello">
      <p @click='`$listeners`.changeData("change")'>孙子组件</p>
  </div>
</template>

<script>
export default {
  name: 'demo',
  created () {
      console.log(this.`$listeners`)
  },
}
</script>
```


依然能拿到从子组中传递过来的 $listeners 所有的监听事件,此时并不是通过 ***emit*** 去触发，而是像调用函数一样,emit只是针对于父子组件的双向通信, $listeners 包了一个对象,分别是changeData和another，通过 $listeners .changeData('change')等于直接触发了事件，执行监听后的回调函数,就是通过函数的传递，调用了父组件的函数。

作，毕镜vuex属于数据共享的原则，只要你保证正确的数据单向流动，而vuex在跨路由的情况下就能显示出自己的优势。

> vuex跨路由操作，必然只能在子路由操作

在h5页面中，当使用vuex进行跨路由操作时，会导致当用户刷新页面的时候，vuex同样的也会初始化，同时就会导致报错，那如果是子路由的话，必然所有数据都是通过父路由进行数据操作完毕之后，放入数据共享，然后再由子路去获取，这样就算刷新页面不会导致数据错误。