```
$ hexo n vue-skill lifecycle
```

前言

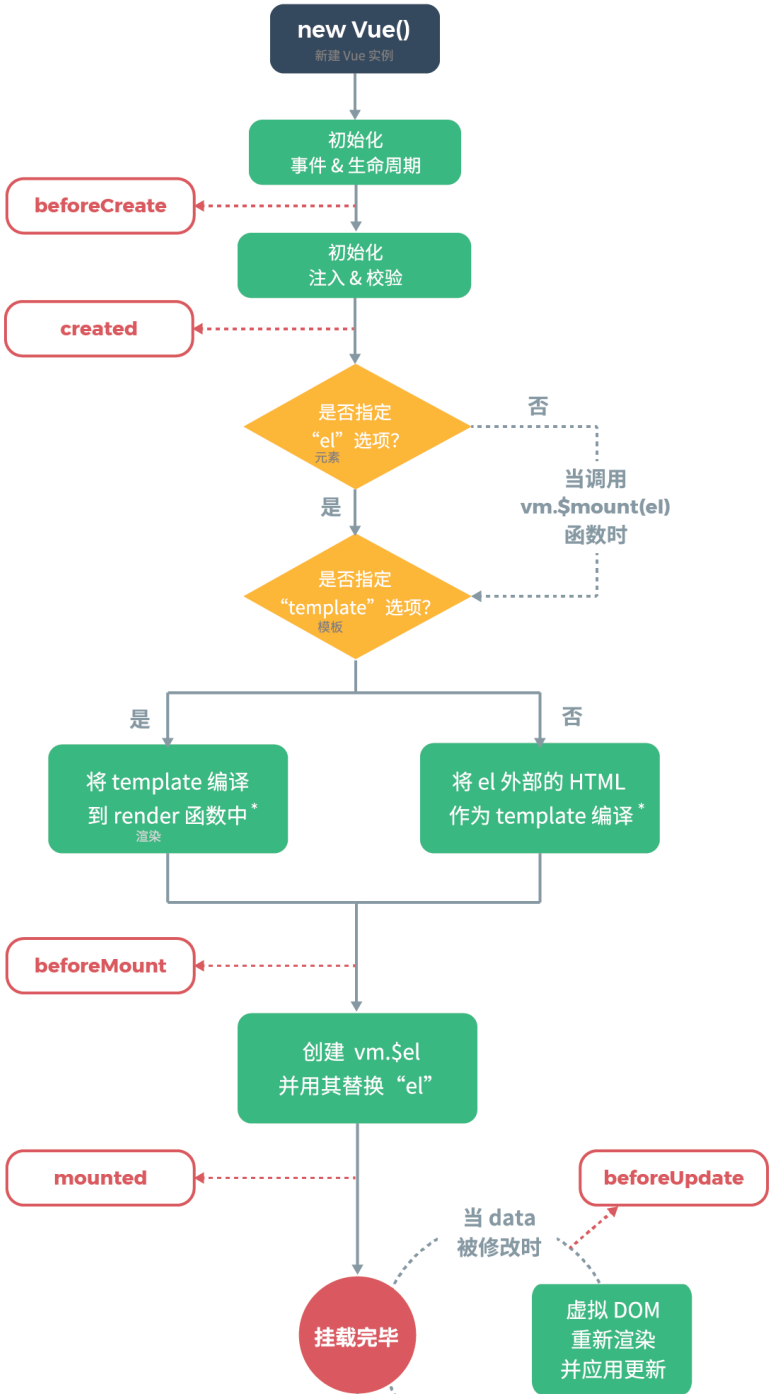
每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给了用户在不同阶段添加自己的代码的机会。

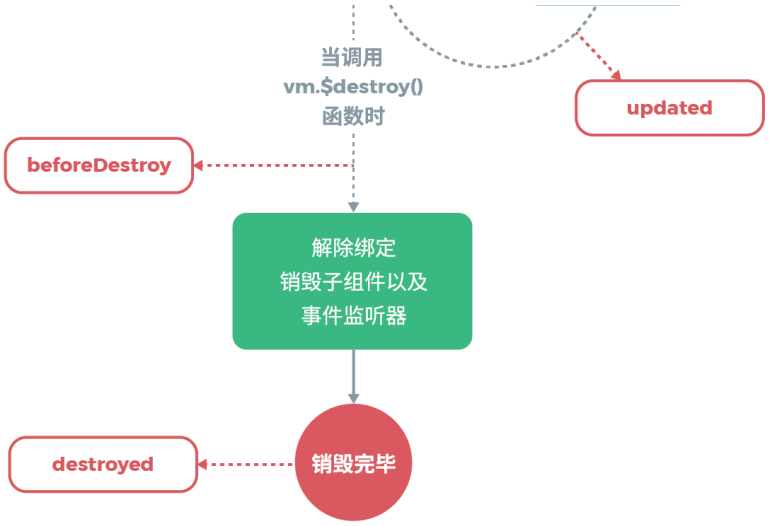
其实对生命周期而言，我们要搞懂的是。

1. 什么阶段初始化数据
2. 什么阶段初始化事件
3. 什么阶段渲染DOM
4. 什么阶段挂载数据

<!-- more -->

生命周期图示





* 如果使用构造生成文件（例如构造单文件组件），
模板编译将提前执行

生命周期钩子函数可以分成6个类型，除了一个最少用的子孙组件错误钩子函数。
每个类型都有 "beforeXX" "XXed"，总共有11个生命周期钩子函数。

序	类型	钩子函数名 - 1	钩子函数名 - 2
1	创建	beforeCreate	created
2	挂载	beforeMount	mounted
3	更新	beforeUpdate	updated
4	销毁	beforeDestroy	destroyed
5	激活	activated	deactivated
6	错误	errorCaptured	\

生命周期钩子官方api 传送门
别看有11个钩子函数，看似一时间难以掌握。
其实也不是很需要全部掌握，常用的就那么几个。
这几个钩子函数会一一介绍，也会先大家演示一遍完整的生命周期。
且实际开发中我们更在意的是，这些钩子函数对组件实例数据/事件的影响。

完整的生命周期

这一章基本是在翻译生命周期图示的内容。
不过很多开发者都对完整的生命周期流程一知半解。
虽然提供源码，还是建议每个人按照自己的理解写一下实例。
新建lifecycle目录，定义lifecycle.vue，导入process.vue

```
<!-- lifecycle.vue -->
<template>
  <lifecycle-process></lifecycle-process>
</template>

<script>
import LifecycleProcess from './process'
```

```
export default {
  components: {
        LifecycleProcess
    }
}
</script>
```

我们在process.vue体现完整的生命周期。

虽然官网的示例都是new Vue() 初始化vue实例。

单文件组件(*.vue)使用export default也同样是初始化vue实例。

这里有几个概念：

1. 数据观测 (data observer) : prop， data， computed
2. 事件机制 (event / watcher)： methods 函数， watch侦听器

我们只简单搞清楚每个阶段发生了什么事情。其他还没有开始做的事情不想提及。

毕竟未开始也未完成，默认就是还没初始化嘛，有什么好说的呢？

create

本小节标题是create，是指vue实例的create阶段。

不是生命周期钩子函数 beforeCreate / created。

我们不打算从生命周期的钩子函数作为切入点。

只要搞清楚了vue实例xx阶段做了什么事情，

那beforeXX / XXed 的区别自然知晓。

我们也根据官方api的资料来表述，实例阶段做了什么事情。

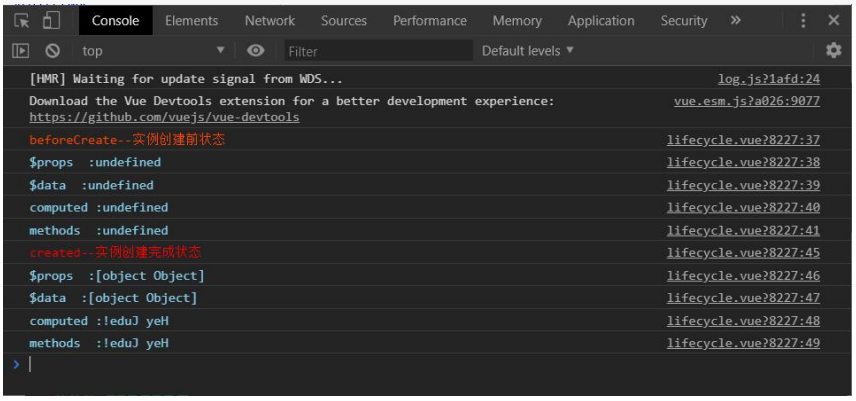
实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调。

那我们应该定义 prop， data， computed methods watch，

然后使用beforeCreate， created前后对比一下。

```
// process.vue

export default {
  // prop， data， computed methods watch
  // 自行定义，这里不浪费篇幅
  props: {},
  data () {
    return {
      msg: 'Hey Jude!'
    }
  },
  methods: {},
  watch: {},
    beforeCreate () {
        console.log("%c%s", "color:orangeRed", 'beforeCreate--实例创建前状态')
        console.log("%c%s", "color:skyblue", "$props    :" + this.$props)
        console.log("%c%s", "color:skyblue", "$data    :" + this.$data)
        console.log("%c%s", "color:skyblue", "computed  :" + this.reverseMsg)
        console.log("%c%s", "color:skyblue", "methods    :" + this.reversedMsg)
        // this.msg = 'msg1'
    },
    created () {
        console.log("%c%s", "color:red", 'created--实例创建完成状态')
        console.log("%c%s", "color:skyblue", "$props    :" + this.$props)
        console.log("%c%s", "color:skyblue", "$data    :" + this.$data)
        console.log("%c%s", "color:skyblue", "computed  :" + this.reverseMsg)
```

```
        console.log("%c%s", "color:skyblue", "methods    :" + this.reversedMsg())
        // this.msg = 'msg2'
    }
}
```



prop， data， computed， methods， watch。

除了watch比较特殊，其他都得到了验证效果。

要验证也很简单，取消 beforeCreate， created 对 this.msg赋值的注释。

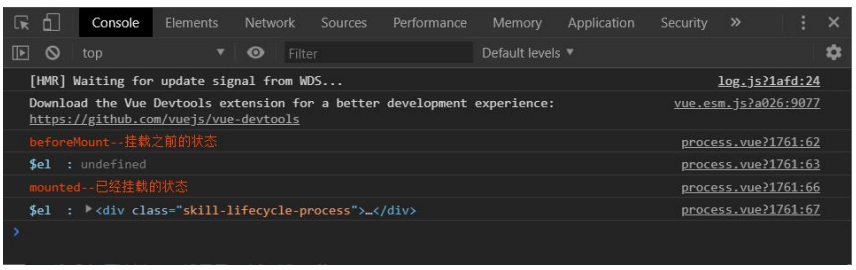
watch msg 看看控制台会打印msg1还是msg2，或者两者皆可。

聪明的你肯定知道控制台只打印msg2的，所以我就不取消注释了。

mount

el 被新创建的 vm.el 替换。 如果根实例挂载到了一个文档内的元素上，当mounted被调用时vm.el也在文档内。

```
<template>
  <div class="skill-lifecycle-process">
    {{ msg }}
  </div>
</template>
export default {
  beforeMount () {
    console.log("%c%s", "color:orangeRed", 'beforeMount--挂载之前的状态')
    console.log("%c%s", "color:skyblue", "$el  :",this.$el)
  },
  mounted () {
    console.log("%c%s", "color:orangeRed", 'mounted--已经挂载的状态')
    console.log("%c%s", "color:skyblue", "$el  :", this.$el)
  }
}
```



mount阶段，由于vue支持多种方式挂载DOM。

而vue实例在created之后，beforeMounted之前这一阶段，

对挂载DOM的方式有判断机制，这里的流程稍微复杂也比较重要。

多种挂载DOM的方式。

- el / $mout
- template

- render

这里打算分别使用n个组件对着这几种挂载方式。

你可以选择暂时跳过，先走完整个周期流程再回来。

create mount是每个组件都必须经历的生命周期，但接下来的生命周期就比较有选择性了。

下一实例阶段 update

这里会按照判断机制的顺序介绍不同的挂载方式。

el / $mount

首先会判断有无el选项声明实例要挂载的DOM。

el选项：提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标。

如果在实例化时存在这个选项，实例将立即进入编译过程，否则，需要显式调用 vm.$mount() 手动开启编译。

el选项需要使用显示使用new创建的实例才生效。

为了方便，这里新建了skill-lifecycle-el.html放在publi（静态资源目录）下。

~~本来想用俄罗斯套娃的方式在vue组件套一个new Vue()，结果行不通。~~

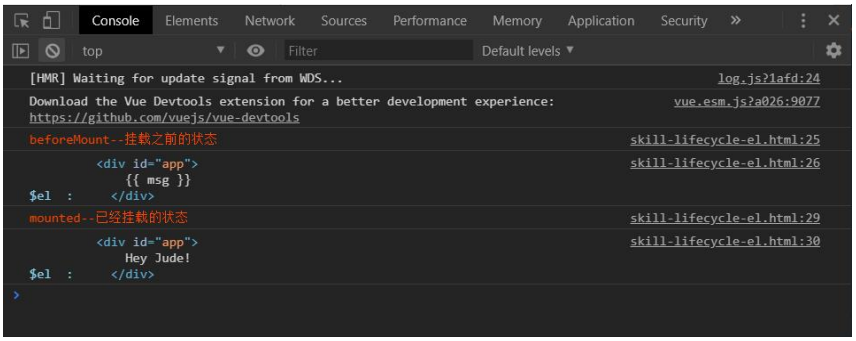
```
<!-- skill-lifecycle-el.html -->
<body>
    <div id="app">
        <p>{{ msg }}</p>
        <p v-text="msg"></p>
    </div>

    <script src="https://cdn.bootcss.com/vue/2.6.10/vue.min.js"></script>

    <script>
        window.onload = function() {
            var vm = new Vue({
                el: '#app',
                props: {},
                data() {
                    return {
                        msg: 'Hey Jude!'
                    }
                },
                computed: {},
                methods: {},
                watch: {},
                beforeMount() {
                    console.log("%c%s", "color:orangeRed", 'beforeMount--挂载之前的状态')
                    console.log("%c%s", "color:skyblue", "$el    :",this.$el)
                    console.log("%c%s", "color:skyblue", "el    :" + this.$el.innerHTML)
                    // debugger
                },
                mounted() {
                    console.log("%c%s", "color:orangeRed", 'mounted--已经挂载的状态')
                    console.log("%c%s", "color:skyblue", "$el    :", this.$el)
                    console.log("%c%s", "color:skyblue", "el    :" + this.$el.innerHTML)
                }
            })
        // vm.$mount('#app')
        }
    </script>
```

```
</bdoy>
```

el 还有 vm.$mount 必须要有一个，不然vue的声明周期就停止，beforeMount不触发。

vm.$mount 手动地挂载一个未挂载的实例。

两种挂载方式的效果是一样的。



值得注意的是，beforeMount真实的DOM确实是会渲染双花括号还有指定的，mounted之后会被替换成真正的数据。

template

判断完el选项，接下来会判断有无template选项

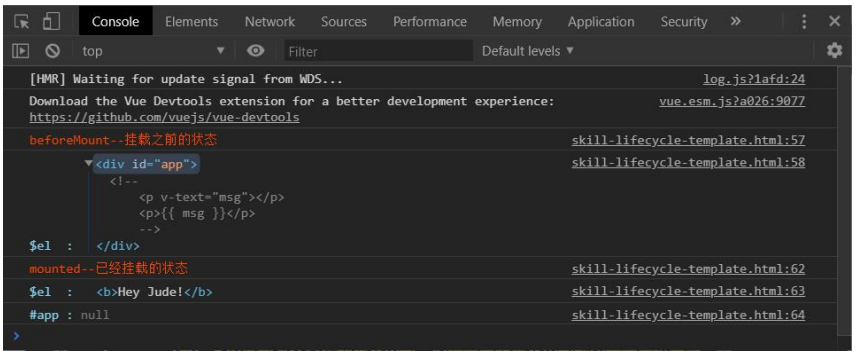
一个字符串模板作为 Vue 实例的标识使用。模板将会替换挂载的元素。

如此说来，作用跟el选项差不多，都是挂载元素的。

那我们声明template选项，写上html tag string，然后把#app DOM里面的内容注释掉。（DOM保留）

```
<!-- skill-lifecycle-template.html -->

<!-- 这个就叫 "#app" DOM -->
<div id="app">
    <!--
    <p v-text="msg"></p>
    <p>{{ msg }}</p>
    -->
</div>

<script>
    new Vue({
        el: '#app',
        template: '<b> {{ msg }}</b>', // 这个就叫 template 选项
        beforeMount () {
            console.log("%c%s", "color:orangeRed", 'beforeMount--挂载之前的状态')
            console.log("%c%s", "color:skyblue", "$el :",this.$el)
            // debugger
        },
        mounted () {
            console.log("%c%s", "color:orangeRed", 'mounted--已经挂载的状态')
            console.log("%c%s", "color:skyblue", "$el :", this.$el)
            console.log("%c%s", "color:skyblue", "#app :", document.querySelector('#app'))
        }
    })
</script>
```

我们在挂载后找一下#app还在不在。



从这图，我们可以知道：

- vm.$el在beforeMount反应的是el选项的#app DOM（此时#app DOM还是模板状态）
- 很明显，template选项把el选择的#app给替换掉了，故**template选项的优先级比el选项/vm.$mount()高**。

el选项：比较温和，只是霸占人家的屋子自己住在里面。

template选项：直接端掉人家的老窝，自己筑新巢。

render

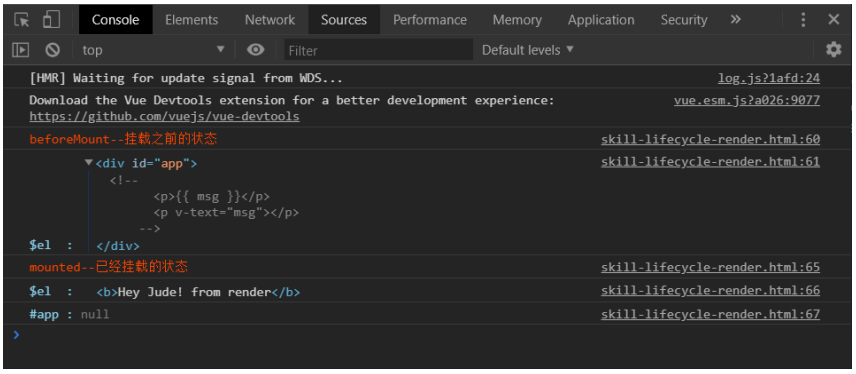
render选项是一个渲染函数，返回虚拟节点（virtual node），又名VNode。

render函数的用法稍微复杂，又牵扯到虚拟DOM、JSX等技术点，之后会另写一篇详细讲解。

假设我们现在并不明白render的用法，只知道它会返回虚拟节点，就够了。

```
<!-- skill-lifecycle-template.html -->
<script>
    new Vue({
        el: '#app',
        template: '<b> {{ msg }}</b>', // 这个就叫 template 选项
        render: function (createElement, context) {
            return createElement('b', this.msg + ' from render')
        }, // render函数
    })
</script>
```

可以看到这里template选项我们不注释，就算我们把注释掉template选项， 输出结果也还是一样。



可以粗暴理解为：render是template的升级版，template字符串模板，render返回的是由函数创建生成的VNode。

所以通过判断机制的流程，我们也很清楚了这几种方式挂载DOM的区别。

1. 判断有无挂载DOM：el选项或者 vm.mount()， 无则停止。
2. 判断有无template选项，有则替换掉挂载DOM元素。
3. 判断有无render函数，有则替换掉挂载DOM元素/template选项。

这几种挂载方式是有优先级的，不过因为按照顺序分析，也不用特意去记，后面的会覆盖前面的。

vue不同构建版本的区别（编译器、运行时）

vuejs有不同的构建版本，他们按照两个维度来分类，模块化及完整性。

模块化容易理解，这取决于使用环境的模块化机制决定。

完整性的话，引用官网资料。

- 完整版：同时包含编译器和运行时的版本。
- 编译器：用来将模板字符串编译成为 JavaScript 渲染函数的代码。
- 运行时：用来创建 Vue 实例、渲染并处理虚拟 DOM 等的代码。基本上就是除去编译器的其它一切。

什么时候必须使用完整版(编译器+运行时)？

template 选项、挂载DOM（el选项/vm.$mount），需要依赖编译器编译，这时必须使用完整版。

当使用 vue-loader 或 vueify 的时候，*.vue 文件内部的模板会在构建时预编译成 JavaScript。你在最终打好的包里实际上是不需要编译器的，所以只用运行时版本即可。

可以看看三个html文件的源码引用的vue版本。

- skill-lifecycle-el.html
- skill-lifecycle-template.html
- skill-lifecycle-render.html

---

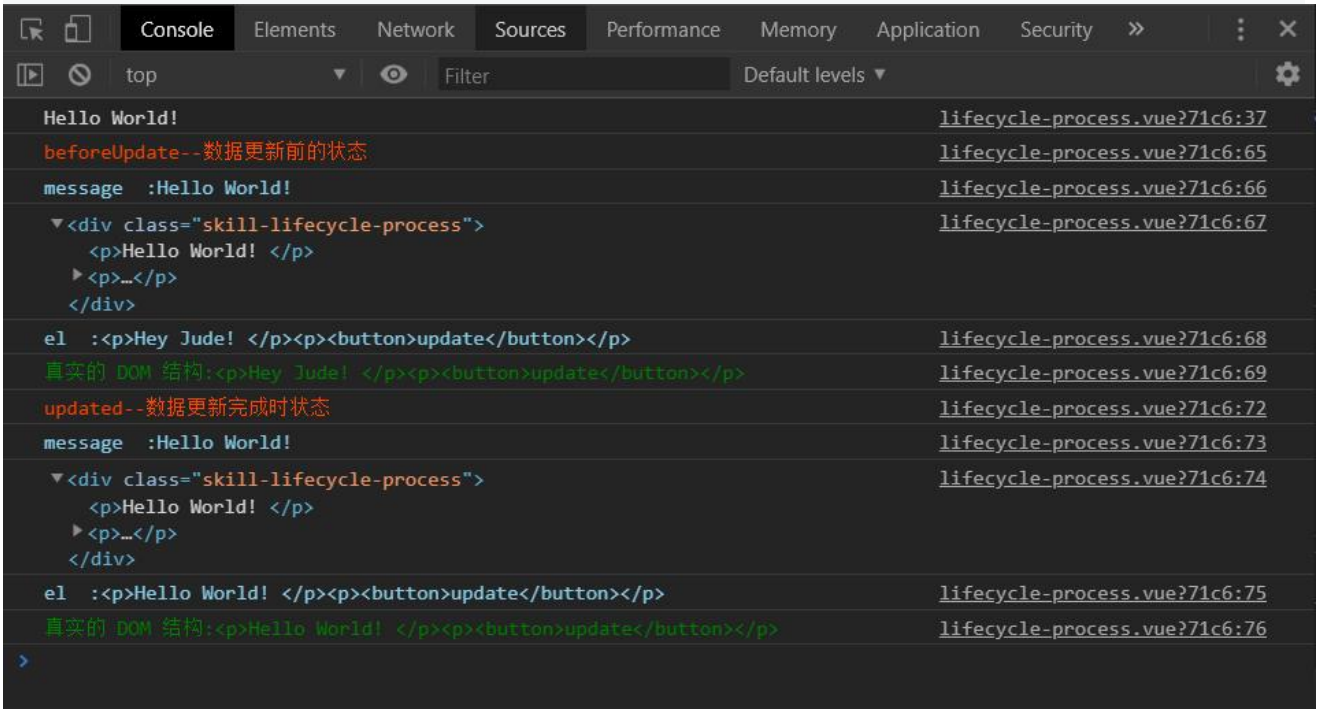
update

数据更改导致的虚拟 DOM 重新渲染和打补丁。

实例data属性更新将会触发update阶段，数据的值改变，才会触发，并不是每次赋值都会触发。

```
<template>
  <div class="skill-lifecycle-process">
    <p>{{ msg }} </p>
    <p><button @click="handleClick">update</button></p>
  </div>
</template>
export default {
  data () {
    return {
      msg: 'Hey Jude!'
    }
  },
  methods: {
    handleClick () {
      this.msg = 'Hello World!'
    }
  },
  watch: {
    msg () {
      console.log(this.msg)
    }
  },
  beforeUpdate () {
    console.log("%c%s", "color:orangeRed", 'beforeUpdate--数据更新前的状态')
    console.log("%c%s", "color:skyblue", "el  :" + this.$el.innerHTML)
    console.log(this.$el)
    console.log("%c%s", "color:skyblue", "message  :" + this.msg)
    console.log("%c%s", "color:green", "真实的 DOM 结构:" + document.querySelector('.skill-lifecycle-
process').innerHTML)
  },
  updated () {
    console.log("%c%s", "color:orangeRed", 'updated--数据更新完成时状态')
    console.log("%c%s", "color:skyblue", "el  :" + this.$el.innerHTML)
    console.log(this.$el);
    console.log("%c%s", "color:skyblue", "message  :" + this.msg)
    console.log("%c%s", "color:green", "真实的 DOM 结构:" + document.querySelector('.skill-lifecycle-
```

```
process').innerHTML)
  }
}
```



不难看出，vue的响应式机制是先改变实例数据。

此时新的实例数据并还没有挂载到DOM，只是存在于虚拟DOM(el)；

再通过虚拟DOM重新渲染DOM元素。

如果这个更新的数据有侦听器，侦听器会在update阶段前触发。

destroy

对应 Vue 实例的所有指令都被解绑，所有的事件监听器被移除，所有的子实例也都被销毁。

销毁指的是销毁vue的响应式系统，事件还有子实例。

都是针对vue层面的，并非销毁DOM之意。

调用vm.$destroy()触发 传送门

调用这个实例方法后，DOM并没有什么变化。

vue实例也还是存在的，只是vue的响应式被销毁。

DOM与vue切断了联系。

active

被 keep-alive 缓存的组件激活/停用时调用

这里需要在lifecycle.vue引用process.vue的地方包裹一层keepa-alive

```
<!-- lifecycle.vue -->
<p><button  @click="handleClick">toggle  show</button></p>
<keep-alive>
  <lifecycle-process  v-if="isShow"></lifecycle-process>
</keep-alive>
```

v-if指令切换组件挂载/移除触发；

v-show指令切换组件显示/隐藏不触发。

```
// lifecycle-process.vue
export default {
  // ...
  activated  ()  {
        console.log('activated')
```

```
    },
    deactivated () {
        console.log('deactivated')
    }
}
```

有意思的是，页面初始化的时候，activated会在mounted之后触发。

单纯的切换组件的挂载/移除状态，activated / deactivated 会触发；

组件不会重新实例化走一遍生命周期，尽管这里用是的v-if。

而当我们destroy组件，之后的每一次切换挂载/移除，组件都会重新实例化，我们只是第一次destroy而已。

errorCapture*

> 当捕获一个来自子孙组件的错误时被调用。此钩子会收到三个参数：错误对象、发生错误的组件实例以及一个包含错误来源信息的字符串。此钩子可以返回 false 以阻止该错误继续向上传播。

这个钩子函数是用来捕获错误的，而且只应用于子孙组件，实际开发中并不常用。 传送门

那么整个周期流程已经介绍完毕了，同样的提供了process.vue源码。也可以选择重新回头深入了解mount机制了。

常用生命周期函数

11个钩子函数就这样介绍完了，常用的钩子函数并不多。

created

此时数据/事件可用，可以在此动态创建数据或者定义自定义事件。

```
export default {
    created () {
        this.$data.staticString = 'static' // 定义变量
        this.$on('on-created', () => { // 定义自定义事件
            console.log(this.$data.staticString)
        })
    }
}
```

注意：created创建的变量，更新不会被vue所监听。 在此处定义变量数据，是为了提升性能，如果这个变量更新与view层无关的话。

mounted

DOM渲染完毕，可以执行页面的初始化操作（移除遮罩），获取DOM（如果有必要的话）。

```
export default {
    mounted () {
        this.init()
        console.log(this.$refs.controlPanel.$options.name)
    }
}
```

vue 并不推荐直接操作DOM，不过还是提供了$ref作为应急解决方案。

useful.vue写的比较简单。

结语

然而， 本篇的内容仅仅讨论的是vue组件的生命周期相关钩子函数。

路由守卫，自定义指令，多个组件引用的钩子函数这些并未提及，推荐几篇文章。

看完相信能收获得更多。

- vue 生命周期深入 针对多个组件引用情况（父子、兄弟组件）等情况生命周期的执行顺序
- vue生命周期探究（一） 包括组件、路由、自定义指令等共计28个的生命周期
- vue生命周期探究（二） 路由导航守卫的钩子函数执行顺序