选项式API(Options API)

使用选项式API，可以用包含多个选项的对象来描述组件的逻辑，例如`data`、`methods`和`mounted`。选项所定义的属性都会暴露在函数内部的`this`上，它会指向当前的组件实例

组合式API(Composition API)

通过组合式API，可以使用导入的API函数来描述组件逻辑

创建vue项目

1、npm init vue@lastest

2、根据提示继续完成创建其中项目名称不要含有大写字符

VUE项目目录结构

```
.vscode          --- VSCode工具的配置文件
node_modules     --- VUE项目的运行依赖文件
public           --- 资源文件夹（浏览器图标）
src              --- 源码文件夹
.gitignore       --- git忽略文件
index.html       --- 入口HTML文件
package.json     --- 信息描述文件
README.md        --- 注释文件
vite.config.js   --- VUE配置文件
```

模板语法

文本插值

最基本的数据绑定形式是文本插值，它使用的是”Mustache"语法（即双大括号）

```
<template>
  <h1>学习</h1>
  <h3>{{  msg }}</h3>
  <p>{{ number + 1 }}</p>
  <P>{{ check ? "真" : "假" }}</P>
  <p>{{ myString.split("").reverse().join("") }}</p>
  <p>{{ rawHtml }}</p>
  <p v-html="rawHtml"></p>
</template>


<script>

export default {
  data(){
    return{
      msg : "学习vue!",
      number : 110,
      check : true,
      myString : "天气不错",
```

```
        rawHtml : "<a href='http://www.swu.edu.cn'>西南大学</a>"
      }
    }
}

</script>
```

每个绑定仅支持**单一表达式**，也就是一段能够被求值的javascript代码，一个简答的判断方法是是否可以合法的写在**return**后面

属性绑定

双大括号不能在HTML attributes中使用。想要响应式地绑定一个attribute，应该使用`v-bind`指令

```
<template>

    <div v-bind:id="dynamicId" v-bind:class="msg">测试</div>


</template>

<script>

export default {
    data(){
        return{
            msg : "active",
            dynamicId : "app"
        }
    }
}

</script>
```

结果：

```
<!DOCTYPE html>
<html lang="en">
▶ <head> ... </head>
▼ <body>
  ▼ <div id="app" data-v-app>
      <!-- <h1>学习</h1>
        <h3>{{  msg }}</h3>
        <p>{{ number + 1 }}</p>
        <P>{{ check ? "真" : "假" }}</P>
        <p>{{ myString.split("").reverse().join("") }}</p>
        <p>{{ rawHtml }}</p>
        <p v-html="rawHtml"></p>  > == $0
      <div id="app" class="active">测试</div>
    </div>
    <script type="module" src="/src/main.js?t=1726041097140"></script>
  </body>
</html>
```

`v-bind`指令指示VUE将元素的`id` attribute 与组件的`dynamicId`属性保持一致，如果绑定的值是`null`或者`undefined`，那么该attribute将会从渲染的元素上移除

简写

因为`v-bind`非常常用，所以提供了特定的简写语法

```
<div :id="dymanicId" :class="dymanicClass"></div>
```

布尔型attribute

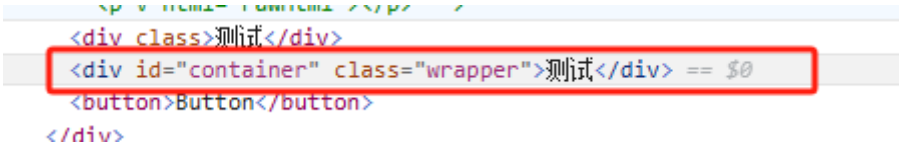
布尔型attribute依据true/false值来决定attribute是否应该存在于该元素上，`disabled`就是最常见的例子之一

```
<template>

    <button :disabled="isButtonDisabled">Button</button>


</template>

<script>

export default {
    data(){
        return{
            isButtonDisabled : true
        }
    }
}

</script>
```

动态绑定多个值

```
<template>
    <div v-bind="objectAttrs">测试</div>
</template>

<script>
export default {
    data(){
        return{
            objectAttrs: {
                id: "container",
                class: "wrapper"
            }
        }
    }
```

```
}

</script>
```



条件渲染

在vue中，提供了条件渲染，这类似于javascript中的条件语句

- `v-if`
- `v-else`
- `v-else-if`
- `v-show`

v-if

`v-if`指令用于条件性地渲染一块内容，这块内容只会在指令的表达式返回真值时才被渲染

```
<template>
    <h3>条件渲染</h3>
    <div v-if="flag">是否能够看见</div>
</template>

<script>
export default {
    data(){
        return {
            flag: false
        }
    }
}

</script>
```

v-else

可以使用`v-else`为`v-if`添加一个else区块

```
<template>
    <h3>条件渲染</h3>
    <div v-if="flag">是否能够看见</div>
    <div v-else>看不见就显示</div>
</template>

<script>
export default {
    data(){
        return {
            flag: false
        }
    }
}
```

```
</script>
```

v-else-if

顾名思义，`v-else-if`提供的是相当于`v-if`的else if区块，它可以连续多次重复使用

```
<template>
    <h3>条件渲染</h3>
    <div v-if="flag">是否能够看见</div>
    <div v-else>看不见就显示</div>
    <div v-if="type == 'A'">A</div>
    <div v-else-if="type == 'B'">B</div>
    <div v-else-if="type == 'C'">C</div>
    <div v-else>D</div>
</template>

<script>
export default {
    data(){
        return {
            flag: false,
            type: "A"
        }
    }
}

</script>
```

v-show

另一个可以用来按条件显示一个元素的指令，其用法和`v-if`用法基本一致

v-if VS v-show

`v-if`是“真实的”按条件渲染，因为它确保了在切换时，条件区块内的事件监听器和子组件都会被销毁与重建

`v-if`也是**惰性**的：如果在初次渲染时条件值为false,则不会做任何事，条件区块只有当条件首次变为true时才被渲染

相比之下，`v-show`简单许多，元素无论初始条件如何，始终会被渲染，只有CSS `display`属性会被切换

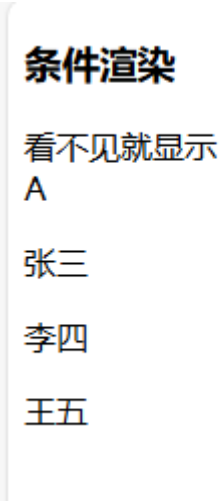
总的来说，`v-if`有更高的切换开销，而`v-show`有更高的初始渲染开销，因此，如果需要频繁切换，则使用`v-show`较好，如果在运行时绑定条件很少改变，则`v-if`会更合适。

列表渲染

可以使用`v-for`指令基于一个数组来渲染一个列表。`v-for`指令的值需要使用`item in tiems`形式的特殊语法，其中`items`是源数据的数组，而`item`是迭代项的别名

```
<template>
   <IfDemo/>
   <ItemDemo/>
</template>


<script setup>

import HelloWorld from './components/HelloWorld.vue'
import IfDemo from './components/if-demo.vue'
import ItemDemo from './components/itemDemo.vue'

</script>
```

```
itemDemo.vue:

<template>
    <p v-for="name in names">{{ name }}</p>

</template>

<script>

export default {
    data() {
        return {
            names: ["张三", "李四", "王五"]
        }
    }
}

</script>
```



复杂数据

大多数情况，渲染的数据源来源于网络请求，也就是`JSON`格式

```
<template>
    <p v-for="name in names">{{ name }}</p>

    <div v-for="result in results">
```

```
        <p>姓名：{{ result.name }}</p>
        <p>年龄：{{ result.age }}</p>
        <img v-bind:src="result.img" :class="class"></img>
    </div>

</template>

<script>

export default {
    data() {
        return {
            names: ["张三", "李四", "王五"],
            results: [
                {
                    "name" : "张三",
                    "age" : 24,
                    "img" :
"https://img2.baidu.com/it/u=1739463261,948605086&fm=253&fmt=auto&app=120&f=JPEG?
w=500&h=512"
                },
                {
                    "name" : "李四",
                    "age" : 25,
                    "img" :
"https://img0.baidu.com/it/u=950345748,3026505306&fm=253&fmt=auto&app=120&f=JPEG?
w=800&h=800"
                },
                {
                    "name" : "王五",
                    "age" : 27,
                    "img" :
"https://img0.baidu.com/it/u=721018011,1708281781&fm=253&fmt=auto&app=120&f=JPEG?
w=500&h=500"
                },
            ],
            class: "myclass"

        }
    }
}

</script>

<style>

.myclass{
    width: 50px;
    height: 50px;
}
</style>
```

姓名：张三

年龄：24

姓名：李四

年龄：25

姓名：王五

年龄：27

`v-for`也支持使用可选的第二个参数表示当前项的位置索引

```
<template>
    <p v-for="(name, index) in names">{{ index}}:{{ name }}</p>

</template>

<script>

export default {
    data() {
        return {
            names: ["张三", "李四", "王五"]
        }
    }
}

</script>
```

也可以使用`of`作为分隔符来替代`in`，这更接近JavaScript的迭代器语法

```
<div v-for="item of items"></div>
```

`v-for`与对象

也可以使用`v-for`来遍历一个对象的所有属性

```
<template>
    <p v-for="(value, key, index) in names">{{ key}}:{{ value }}:{{ index }}</p>
    <!-- 前面的顺序不要修改 -->
</template>
```

```
<script>

export default {
    data() {
        return {
            userInfo: {
                name: "zhangsan",
                age: 24
            }
        }
    }
}

</script>
```

通过key管理状态

vue默认按照"就地更新"的策略来更新通过v-for渲染的元素列表，当数据项的顺序改变时，vue不会随之移动DOM元素的顺序，而是就地更新每个元素，确保它们在原本指定的索引位置上渲染。

为了给vue一个提示，以便它可以跟踪每个节点的标识，从而重用和重新排序现有的元素，你需要为每个元素对应的块提供一个唯一的key attribute

```
<template>
    <p v-for="(name, index) in names" :key="index">{{ name }}</p>
    <!-- 前面的顺序不要修改 -->
</template>

<script>

export default {
    data() {
        return {
            names : ["张三", "李四", "王五"]
        }
    }
}

</script>
```

> key 在这里是通过 v-bind 绑定的特殊attribute
>
> 推荐在任何可行的时候为 v-for 提供一个 key attribute
>
> key 绑定的值期望是一个基础类型的值，例如字符串或number类型

key的来源

不要使用index作为key的值，要确保每一条数据的唯一索引不会发生变化，例如使用user.id作为key

事件处理

我们可以使用`v-on`指令（简写为`@`）来监听DOM事件，并在事件触发时执行对应的JavaScript。用法：`v-on:click="methodName"`或`@click="handler"`

事件处理器的值可以是：

- **内联事件处理器：**事件被触发时执行的内联javascript语句（与`onclick`类似）
- **方法事件处理器：**一个指向组件上定义的方法的属性名或是路径

内联事件处理器

内联事件处理器通常用于简单场景

```
<template>
    <h3>内联事件处理器</h3>
    <div>
        <button v-on:click="count++">add</button>
        <p>{{ count }}</p>
    </div>
</template>

<script>

export default {
    data() {
        return {
            count: 0
        }
    }
}
</script>
```

方法事件处理器

```
<template>
    <h3>内联事件处理器</h3>
    <div>
        <button v-on:click="count++">add</button>
        <button @click="addCount">methodAddCount</button>
        <p>{{ count }}</p>
    </div>
</template>

<script>

export default {
    data() {
        return {
            count: 0
        }
    },
        //所有的方法和函数都写在这里
    methods: {
        addCount() {
            //读取到data里面的数据
            this.count += 1
        }
    }
}


</script>
```

事件参数

事件参数可以获取 `event` 对象和通过事件传递数据

获取`event`对象

```
<template>
    <h3>内联事件处理器</h3>
    <div>
        <button @click="addCount">Add</button>
        <p>{{ count }}</p>
    </div>
</template>

<script>

export default {
    data() {
        return {
            count: 0
        }
    },
    methods: {
        //event对象
        addCount(e) {
            //vue中的event对象，就是原生JS的event对象
            e.target.innerHTML="Add" + this.count
            this.count += 1
        }
    }
}
</script>
```

传递参数

```
<template>
    <h3>内联事件处理器</h3>
    <div>
        <button @click="addCount('hello')">Add</button>
        <p>{{ count }}</p>
    </div>
</template>

<script>

export default {
    data() {
        return {
            count: 0
        }
    },
    methods: {
        addCount(msg) {
            console.log(msg)
            this.count += 1
```

```
        }
    }
}
</script>
```

传递参数的过程中获取`event`

```
<template>
    <h3>内联事件处理器</h3>
    <div>
        <button v-for="name in names" @click="addCount(name, $event)">{{ name }}
</button>
    </div>
</template>

<script>

export default {
    data() {
        return {
            names: ["张三", "李四", "王五"]
        }
    },
    methods: {
        addCount(msg,e) {
            console.log(msg)
            console.log(e)
            this.count += 1
        }
    }
}
</script>
```

事件修饰符

在处理事件时调用`event.preventDefault()`或`event.stopPropagation()`是很常见的，尽管我们可以直接在方法内调用，但如果方法能更专注于数据逻辑而不用去处理DOM时间的细节会更好

为解决这一问题，vue为`v-on`提供了**事件修饰符**，常用有以下几个：

- `.stop`
- `.prevent`
- `.once`
- `.enter`
- ...

具体参考官网文档

阻止默认事件

```
<template>
    <h3>事件修饰符</h3>
    <!-- 法一：使用@click.prevent -->
    <a @click.prevent="clickLink" href="http://www.swu.edu.cn">西南大学</a>
</template>
```

```
<script>

export default {
    data() {
        return {

        }
    },
    methods: {
        clickLink(e) {
            //阻止默认事件
            // 法二：使用原生js的event对象
            //e.preventDefault()
            console.log("点击了链接")
        }
    }
}

</script>
```

阻止事件冒泡

```
<template>
    <h3>事件修饰符</h3>
    <div @click="clickDiv">
        <!-- 法二：使用click.stop事件触发器 -->
        <p @click.stop="clickP">冒泡测试</p>
    </div>
</template>

<script>

export default {
    data() {
        return {

        }
    },
    methods: {
        clickDiv() {
            console.log("DIV")
        },
        clickP(e) {
            // 法一：原生js的event对象
            e.stopPropagation()
            console.log("P");
        }
    }
}

</script>
```

数组变化侦测

变更方法

vue能够侦听响应式数组的变更方法，并在它们被调用时触发相关的更新(UI界面同时更新)，这些变更方法包括：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

```
<template>
    <h3>数组变化侦测</h3>
    <button @click="addName">add name</button>
    <ul>
        <li v-for="(name, index) in names" :key="index">{{ name }}</li>
    </ul>
</template>

<script>

export default {
    data() {
        return {
            names: ["张三", "李四", "王五"]
        }
    },
    methods: {
        addName() {
            this.names.push("麻子")
        }
    }
}

</script>
```

替换一个数组

变更方法，顾名思义，就是会对调用它们的原数组进行变更，相对地，也有一些不可变(immutable)方法，例如 `filter()`,`concat()` 和 `slice()`，这些都不会变更原数组，而总是**返回一个新数组。**当遇到的是非变更方法时，我们需要将旧的数组替换为新的

```
<template>
    <h3>数组变化侦测</h3>
    <button @click="addName">add name</button>
    <ul>
        <li v-for="(name, index) in names" :key="index">{{ name }}</li>
    </ul>
    <button @click="concatHandle">合并数组2到数组1</button>
    <h3>数组1</h3>
    <div>
        <p v-for="(num, index) in nums1" :key="index">{{ num }}</p>
    </div>
```

```
    <h3>数组2</h3>
    <div>
        <p v-for="(num, index) in nums2" :key="index">{{ num }}</p>
    </div>
</template>

<script>

export default {
    data() {
        return {
            names: ["张三", "李四", "王五"],
            nums1: [1, 2, 3, 4, 5],
            nums2: [6, 7, 8, 9, 10]
        }
    },
    methods: {
        addName() {
            //引起UI自动更新
            // this.names.push("麻子")
            //不会引起ui自动更新
            // this.names.concat(["麻子"])
            // console.log(this.names.concat(["麻子"]))
            //重新赋值给原数组后可以引起ui更新
            this.names = this.names.concat(["麻子"])
        },
        concatHandle() {
            for (const num of this.nums2) {
                this.nums1.push(num)
            }
            // this.nums1 = this.nums1.concat(this.nums2)
        }
    }
}

</script>
```

计算属性

模板中的表达式虽然方便，但也只能用来做简单的操作。如果在模板中写太多逻辑，会让模板变得臃肿，难以维护，因此我们推荐使用**计算属性**来描述依赖响应式状态的复杂逻辑

```
<template>
    <h3>{{ person.name }}</h3>
    <p>{{ personContent }}</p>

</template>

<script>

export default {
    data() {
        return {
            person: {
                name: "张三",
                color: ["green", "red", "orange"]
            }
```

```
        }
    },
    //计算属性
    computed: {
        personContent() {
            return this.person.color.length > 0 ? "yse" : "no"
        }
    }

}
</script>
```

计算属性VS函数方法

> **重点区别：**
>
> 计算属性：**计算属性值会基于其响应式依赖被缓存。**一个计算属性仅会在其响应式依赖更新时才重新计算
>
> 方法：方法调用总是会在重渲染发生时再次执行函数

Class绑定

数据绑定的一个常见需求场景是操纵元素的CSS class列表，因为`class`是attribute，我们可以和其他attribute一样使用`v-bind`将它们和动态的字符串绑定，但是，在处理比较复杂的绑定时，通过拼接生成字符串是麻烦且易出错的，因此，vue专门为`class`的`v-bind`用法提供了特殊的功能增强，除了字符串外，表达式的值也可以是对象或数组

```
<template>
    <!-- 法一 -->
    <p :class="{'active' : isActive, 'text-denger': hasError}">class样式绑定1</p>
    <!-- 法二：用对象形式 -->
    <p :class="classObject">class样式绑定2</p>
    <!-- 法三：用数组形式 -->
    <p :class="[arrActive, arrTextDenger]">class样式绑定3</p>
    <!-- 法四：三目运算表达式 -->
    <p :class="[isActive ? 'active text-denger' : '']">class样式绑定4</p>
    <!-- 法五：数组和对象一起使用 -->
    <p :class="[{active : 'isActive'}, arrTextDenger]">class样式绑定5</p>
</template>

<script>

export default {
    data(){
        return {
            //法一
            isActive : true,
            hasError : false,
            //法二
            classObject: {
                'active' : true,
                'text-denger' : false
            },
            //法三
            // arrActive : 'active',
            arrTextDenger : 'text-denger'
```

```
        }
    }
}
</script>

<style>
.active {
    color: green;
    font-size: 40px;
}
.text-denger {
    color: red;
    font-size: 40px;
}

</style>
```

> 提示
>
> 数组和对象嵌套过程中，只能是数组里面嵌套对象，不能反其道而行

Style绑定

数据绑定的一个常见需求场景是操纵元素的CSS style列表，因为`style`是attribute，我们可以和其他attribute一样使用`v-bind`将它们和动态的字符串绑定，但是，在处理比较复杂的绑定时，通过拼接生成字符串是麻烦且易出错的，因此，vue专门为`style`的`v-bind`用法提供了特殊的功能增强，除了字符串外，表达式的值也可以是对象或数组(一般不推荐使用数组，而使用对象避免混淆)

```
<template>
    <p :style="{color: activeColor, fontSize: activeFontSize}">Style绑定1</p>
    <p :style="styleObject">Style绑定2</p>
    <p :style="[styleObject]">Style绑定3</p>
</template>

<script>

export default {
    data() {
        return {
            activeColor : "red",
            activeFontSize : "30px",
            styleObject: {
                color: "red",
                fontSize: "30px"
            }
        }
    }
}

</script>
```

侦听器

我们可以使用`watch`选项在每次响应式属性发生变化时触发一个函数

```
<template>
```

```
    <h3>侦听器</h3>
    <p>{{ message }}</p>
    <button @click="updateHandle">changeValue</button>
</template>

<script>

export default {
    data() {
        return {
            message: "hello"
        }
    },
    methods: {
        updateHandle() {
            this.message = "world"
        }
    },
    //侦听器
    watch: {
        // newValue: 改变之后的数据
        // oldValue: 改变之前的数据
        // 函数名必须与侦听的数据对象保持一致
        message(newValue, oldValue) {
            // 数据发生变化，自动执行的函数
            console.log(oldValue, newValue)
        }
    }
}
</script>
```

表单输入绑定

在前端处理表单时，我们尝尝需要将表单输入框的内容同步给javascript中相应的变量。手动连接值绑定和更改事件监听器可能会很麻烦，`v-model`指令帮我们简化了这一步骤

input输入框

```
<template>
    <h3>表单输入绑定</h3>
    <form>
        <input type="text" v-model="message">
    </form>
    <p>{{ message }}</p>
</template>

<script>
export default {
    data() {
        return {
            message: ""
        }
    }
}

</script>
```

复选框

单一的复选框，绑定布尔类型值

```
<template>
    <h3>表单输入绑定</h3>
    <form>
        <input type="text" v-model="message">
    </form>
    <p>{{ message }}</p>
    <form>
        <input type="checkbox" id="checkbox" v-model="checked">
        <label for="checkbox">{{ checked }}</label>
    </form>
</template>

<script>
export default {
    data() {
        return {
            message: "",
            checked: false
        }
    }
}

</script>
```

修饰符

`v-model`也提供了修饰符：`.lazy`，`.number`，`.trim`

`.lazy`

默认情况下，`v-model`会在每次`input`事件后更新数据，你可以添加`.lazy`修饰符来改为在每次`change`事件后更新数据

```
<template>
    <h3>表单输入绑定</h3>
    <form>
        <input type="text" v-model.lazy="message">
    </form>
    <p>{{ message }}</p>
</template>

<script>
export default {
    data() {
        return {
            message: ""
        }
    }
}

</script>
```

`.number`：只接收数字

`.trim`：去掉接收数据的前后空格

模板引用

虽然vue的声明性渲染模型为你抽象了大部分对DOM的直接操作，但在某些情况下，我们仍然要直接访问底层DOM元素，要实现这一点，我们可以使用特殊的 `ref` attribute

挂载结束后引用都会被暴露在 `this.$refs` 之上

```
<template>
    <h3>模板引用</h3>
    <div ref="container" class="container">{{ content }}</div>
    <input type="text" ref="username">
    <button @click="getElementHandle">获取元素</button>
</template>




<script>

/**
 * 内容改变：{{ 模板语法 }}
 * 属性改变：v-bind:指令
 * 事件:v-on:事件
 *
 * 如果没有特别的需求，不要操作DOM
 */
export default {
    data() {
        return {
            content: "内容"
        }
    },
    methods: {
        getElementHandle() {
            // innerHTML: 原生JS的属性
            this.$refs.container.innerHTML = "hello world"
            console.log(this.$refs.username.value)
        }
    }
}
</script>
```

组件组成

组件最大的优势就是可复用性

当使用构建步骤时，我们一般会将Vue组件定义在一个单独的 `.vue` 文件中，这被叫做单文件组件（简称SFC）

组件组成结构

```
<template>
    <div>承载标签</div>
</template>
<script>
export default{}
</script>

<!-- scoped: 让当前样式只在当前组件中生效 -->
<style scoped>
</style>
```

组件引用

```
<template>
<!-- 第三步：显示组件 -->
 <MyComponent1/>

</template>

<script>

//第一步：引入组件
import MyComponent1 from './MyComponent1.vue';

// 第二步：注入组件,在app.vue中会自动引用
export default {
    components: {
        MyComponent1
    }
}

</script>

<style>
</style>
```

组件嵌套关系

组件允许我们将UI划分为独立的、可重用的部分，并且可以对每个部分进行单独的思考。在实际应用中，组件常常被组织成层层嵌套的树状结构。

这和我们的HTML元素的方式类似，vue实现了自己的组件模型，使我们可以在每个组件内封装自定义内容与逻辑

```
// Header.vue:
<template>
    <h3>Hearder</h3>
</template>

<script>

</script>
```

```
<style scoped>
h3{
    width: 100%;
    height: 100px;
    border: 5px solid #999;
    text-align: center;
    line-height: 100px;
    box-sizing: border-box;
}

</style>

//Main.vue:
<template>
    <div class="main">
        <h3>Main</h3>
        <Article/>
        <Article/>
    </div>
</template>
<script>

import Article from './Article.vue';


export default {
    components: {
        Article
    }
}
</script>
<style scoped>
.main{
    width: 70%;
    height: 400px;
    float: left;
    border: 5px solid #999;
    box-sizing: border-box;
}
</style>

//Aside.vue:
<template>
    <div class="aside">
        <h3>Aside</h3>
        <Item/>
        <Item/>
        <Item/>
    </div>
</template>
<script>
import Item from './Item.vue';

export default {
    components: {
        Item
    }
```

```
}
</script>
<style scoped>
.aside{
    width: 29%;
    height: 400px;
    float: right;
    border: 5px solid #999;
    box-sizing: border-box;
}
</style>

// Article.vue:
<template>
    <div class="article">
        <h3>Article</h3>
    </div>
</template>
<script>
export default {

}
</script>
<style scoped>
h3{
    width: 90%;
    margin: 0 auto;
    text-align: center;
    line-height: 100px;
    box-sizing: border-box;
    margin-top: 50px;
    background: #999;
}
</style>

// Item.vue:
<template>
    <div class="item">
        <h3>Item</h3>
    </div>
</template>
<script>
export default {

}
</script>
<style scoped>
h3{
    width: 90%;
    margin: 0 auto;
    text-align: center;
    line-height: 50px;
    box-sizing: border-box;
    margin-top: 50px;
    background: #999;
}
</style>
```

```
// App.vue:
<template>
  <Header/>
  <Main/>
  <Aside/>
</template>


<script setup>
import Header from './pages/Header.vue';
import Main from './pages/Main.vue';
import Aside from './pages/Aside.vue';
</script>
```

组件注册方式

一个vue组件在使用前需要先被“注册”，这样vue才能在渲染模板时找到其对应的实现。组件注册有两种方式：全局注册和局部注册

全局注册

```
// main.js:

import { createApp } from 'vue'
import App from './App.vue'
import Header from './pages/Header.vue'

const app = createApp(App)

// 必须在create之下，.mount()之前写组件的注册
// 第一个参数是在其他组件中使用的名字，第二个是导入时取的名字
app.component("Header", Header)
app.mount('#app')

// app.vue
<template>
  <Header/>
  <Main/>
  <Aside/>
</template>


<script setup>
// import Header from './pages/Header.vue';
import Main from './pages/Main.vue';
import Aside from './pages/Aside.vue';
</script>
```

局部注册

全局注册虽然方便，但有以下几个问题：

- 全局注册时，没有被使用的组件在生产打包时即使它没有被实际引用，但仍然会出现在打包后的JS文件中
- 全局注册在大型项目的以来关系变得不那么明确，在父组件中使用子组件时，不太容易定位子组件的实现，和使用过多的全局变量一样，这可能会影响应用长期的可维护性。

局部组件的使用之前已经多次使用

组件传递数据: props

静态传递

```
// Parent.vue:
<template>
    <h3>Paremt</h3>
    <Child title="参数传递"/>
</template>

<script>

import Child from './Child.vue';
export default {
    data() {
        return {

        }
    },
    components: {
        Child
    }
}
</script>

// Child.vue:
<template>
    <h3>Child</h3>
    <p>{{ title }}</p>
</template>

<script>
export default {
    data() {
        return {

        }
    },
    props: ["title"]
}
</script>
```

动态传递

```
// Parent.vue:
<template>
    <h3>Paremt</h3>
    <Child :title="message"/>
</template>

<script>

import Child from './Child.vue';
export default {
```

```
    data() {
        return {
            message: "参数再次传递"
        }
    },
    components: {
        Child
    }
}
</script>
```

> 注意：
>
> `props`传递参数，只能从父级传递给子级，不能由子级传递给父级

组件传递多种数据类型

任何类型的值都可以作为props的值被传递，如对象、数字、数组等

```
// Parent.vue
<template>
    <h3>Paremt</h3>
    <Child :title="message" :age="age" :names="names" :user="user"/>
</template>

<script>

import Child from './Child.vue';
export default {
    data() {
        return {
            message: "参数再次传递",
            age: 25,
            names: ["张三", "李四", "王五"],
            user: {
                name: "张三",
                age: 24
            }
        }
    },
    components: {
        Child
    }
}
</script>

// Child.vue:
<template>
    <h3>Child</h3>
    <p>{{ title }}</p>
    <p>{{ age }}</p>
    <ul>
        <li v-for="(name, index) of names" :key="index">{{ name }}</li>
    </ul>
    <p>{{ user.name }}</p>
    <p>{{ user.age }}</p>
</template>
```

```
<script>
export default {
    data() {
        return {

        }
    },
    props: ["title", "age", "names", "user"]
}
</script>
```

组件传递props校验

vue组件可以更细致地声明对传入的props的校验要求

```
// CpmponentA
<template>
    <h3>ComponentA</h3>

    <ComponentB  />
</template>

<script>

import ComponentB from './ComponentB.vue';
export default {
    data() {
        return {
            title: 20
        }
    },
    components: {
        ComponentB
    }
}

</script>

<style>
</style>

// ComponentB
<template>
    <h3>ComponentB</h3>
    <p>{{ title }}</p>
    <p>{{ age }}</p>
    <p v-for="(item, index) of names" :key="index">{{ item }}</p>
</template>

<script>
export default {
    data() {
        return {

        }
    },
```

```
    props: {
        title: {
            // 一种类型
            //type: String
            // 多种类型
            type: [String, Number, Array, Object],
            // 必传选项
            required: true
        },
        age: {
            type: Number,
            // 设置默认值
            default: 0
        },
        // 数字和字符串可以直接default，但是如果是数组和对象，必须通过工厂函数返回默认值
        names: {
            type: Array,
            default() {
                return ["空", "空"]
            }
        }
    }
}

</script>

<style>
</style>
```

> **注意：**props中接收的数据是只读的，不能对其进行修改

组件事件（组件传递数据）

在组件的模板表达式中，可以直接使用`$emit`方法触发自定义事件

触发自定义事件的目的是组件之间传递数据（子组件传递数据到父组件，prop只能由父组件传递数据给子组件）

```
// ComponentEvent
<template>
    <h3>ComponentEvent</h3>
    <ComponentEventChild @sent-event="getMsgHandle"/>
    <p>child获取到的消息：{{ message }}</p>
</template>

<script>

import ComponentEventChild from './ComponentEventChild.vue';
export default {
    data() {
        return {
            message: ""
        }
    },
    components: {
        ComponentEventChild
    },
```

```
    methods: {
        getMsgHandle(data) {
            this.message = data
        }
    }
}
</script>

// ComponentEventChild
<template>
    <h3>ComponentEventChild</h3>
    <button @click="clickHandle">传递数据</button>
</template>

<script>
export default {
    data() {
        return {
            msg: "child消息"
        }

    },
    methods: {
        clickHandle() {
            // 自定义事件
            this.$emit("sent-event", this.msg)
        }
    }
}
</script>
```

> 提示
>
> 组件之间传递数据的方案：
>
> 1、父传子：`props`
>
> 2、子传父：`自定义事件(this.$emit)`

组件事件配合`v-model`使用

如果是用户输入，我们希望在获取数据的同时发送数据配合`v-model`来使用

```
// SearchComponent.vue
<template>
    搜索 : <input type="text" v-model="search">
</template>

<script>
export default {
    data() {
        return {
            search: ""
        }
    },
    // 侦听器
    watch: {
```

```
        // newValue和oldValue的顺序不能够反着写
        search(newValue, oldValue) {
            this.$emit("search-event", newValue)
        }
    }
}
</script>

// Main.vue
<template>
    <h3>Main</h3>

    <SearchComponent @search-event="getSearchHandle"/>
    <p>搜索结果为：{{ message }}</p>
</template>

<script>
import SearchComponent from './SearchComponent.vue';
export default {
    data() {
        return {
            message: ""
        }
    },
    components: {
        SearchComponent
    },
    methods: {
        getSearchHandle(data) {
            this.message = data
        }
    }
}
</script>
```

组件数据传递

之前学习了组件之间的数据传递，`props`和自定义事件两种方式

除了上述的方案，`props`也可以实现子传父（父级先传递一个函数给子级，子级使用函数给父级传递数据）

```
// 父类ComponentA.vue
<template>
    <h3>ComponentA</h3>
    <ComponentB :onEvent="sentFunc" title="标题"/>
    <p>{{ message }}</p>
</template>

<script>
import ComponentB from './ComponentB.vue';
export default {
    data() {
        return {
            message: ""
        }
    },
```

```
    components: {
        ComponentB
    },
    methods: {
        sentFunc(data) {
            this.message = data
        }
    }
}

</script>

// 子类ComponentB.vue
<template>
    <h3>ComponentB</h3>
    <p>{{ title }}</p>
    <p>{{ onEvent("子类参数传递") }}</p>
</template>

<script>
export default {
    data() {
        return {

        }
    },
    props: {
        title: String,
        onEvent: Function
    }
}

</script>
```

透传Attributes

"透传Attribute"指的是传递给一个组件，却没有被该组件声明为props或emits的attribute或者`v-on`事件监听器。最常见的例子就是`class`、`style`和`id`

当一个组件以单个元素为根作渲染时，透传的attribute会自动被添加到根元素上

```
// app.vue
<template>
  <AttrComponent class="MyClass"/>
</template>

import AttrComponent from './components/AttrComponent.vue';

// AttrComponent.vue
<template>
    <!-- 必须是唯一根元素 -->
    <div>
        <h3>透传Attribute</h3>
    </div>
</template>
<script>
export default {
```

```
    // 禁止继承
    //inheritAttrs: false
}
</script>
<style>
.MyClass {
    color: red;
}
</style>
```

插槽slots

在前面了解到组件能够接受任意类型的javascript值作为props，但在某些场景中，我们可能想要为子组件传递一些模板片段，让子组件在它们的组件中渲染这些片段

```
// SlotBase.vue
<template>
    <h3>插槽使用</h3>
    <!-- 可以根据slot的位置来改变slot引用的位置 -->
    <slot></slot>
</template>

<script>

</script>

<style>
</style>

// app.vue
<template>
  <SlotBase>
    <!-- 插槽使用 -->
    <div>
      <h3>插槽标题</h3>
      <p>插槽内容</p>
    </div>
  </SlotBase>
</template>


<script setup>
import SlotBase from './components/slots/SlotBase.vue'

</script>
```

`<slot>`元素是一个插槽出口(slot outlet)，标示了父元素提供的插槽内容(slot content)将在哪里被渲染

渲染作用域

插槽内容可以访问到父组件的数据作用域，因为插槽内容本身是在父组件模板中定义的

默认内容

在外部没有提供任何内容的情况下，可以为插槽指定默认内容

```
<template>
    <h3>Component</h3>
    <slot>插槽默认值</slot>
</template>
```

具名插槽

当一个组件中不只是需要一个插槽时，可以使用具名插槽来将不同的插槽进行区分开来

```
// SlotName.vue
<template>
    <h3>具名插槽使用</h3>
    <slot name="header">默认值</slot>
    <hr/>
    <slot name="body">插槽默认值</slot>
</template>

<script>

</script>

<style>
</style>

// app.vue
<template>
  <SlotName>
    <template #header>
      <h3>插槽标题</h3>
    </template>
    <template v-slot:body>
      <h3>插槽内容</h3>
    </template>
  </SlotName>
</template>


<script setup>
import SlotName from './components/slots/SlotsName.vue'

</script>
```

`v-slot` 有对应的简写 `#`，因此 `<template v-slot:header>` 可以简写为 `<template #header>`，其意思就是“将这部分模板片段传入子组件的header插槽中”

插槽中的数据传递

在某些场景下插槽的内容可能想要同时使用父组件域内和子组件域内的数据，要做到这一点，我们需要一种方法来让子组件在渲染时将一部分数据提供给插槽

可以像对组件传递props那样，向一个插槽的出口上传递attributes

```
// SlotAttr.vue
<template>
    <h3>插槽使用</h3>
    <!-- 使用props的格式来传递 -->
    <slot :msg="childMsg">默认值</slot>
</template>

<script>
export default {
    data() {
        return {
            childMsg: "子类传递数据"
        }
    }
}

</script>

<style>
</style>

// App.vue
<template>
  <!-- 接收到的slotProps是对象数据类型 -->
  <SlotAttr v-slot="slotProps">
    <h3>{{ message }}--{{ slotProps.msg }}</h3>
  </SlotAttr>
</template>


<script>
import SlotAttr from './components/slots/SlotsAttr.vue'

export default {
  data() {
    return {
      message: "父类数据"
    }
  }
}

</script>
```

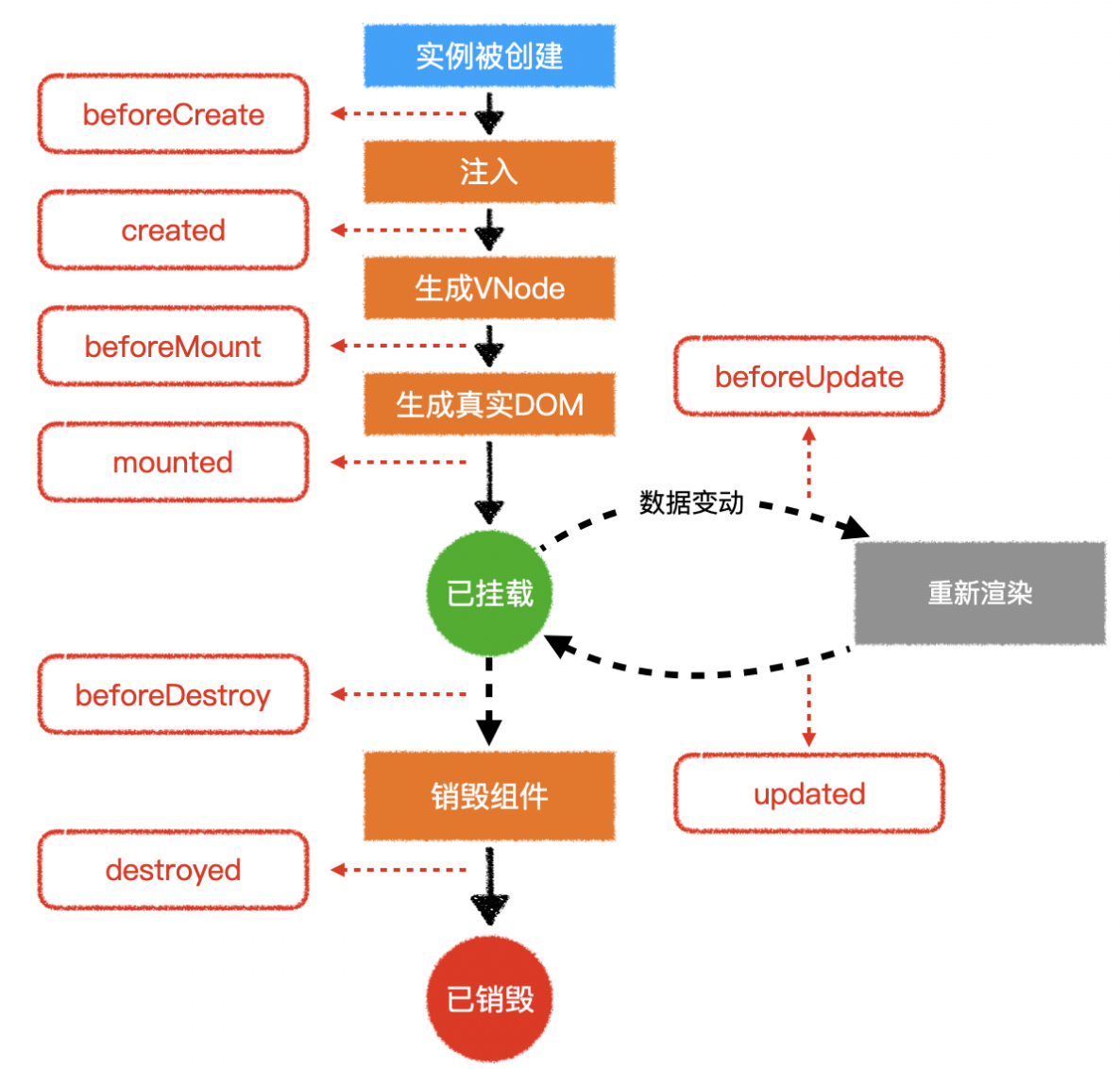
具名插槽接收传递数据

```
// SlotsAttr.vue
<template>
    <h3>插槽使用</h3>
    <!-- 使用props的格式来传递 -->
    <slot :msg="childMsg" name="header">默认值</slot>
    <hr/>
    <slot :job="jobMsg" name="body">body默认值</slot>
</template>

<script>
export default {
```

```
    data() {
        return {
            childMsg: "子类传递数据",
            jobMsg: "程序员"
        }
    }
}

</script>

<style>
</style>

// App.vue
<template>
  <!-- 具名插槽获取数据 -->
  <SlotsAttr>
    <template #header="slotProps">
      <h3>{{ message}} -- {{ slotProps.msg }}</h3>
    </template>

    <template #body="slotProps">
      <p>{{ slotProps.job }}</p>
    </template>
  </SlotsAttr>
</template>
<script>
import SlotsAttr from './components/slots/SlotsAttr.vue';
export default {
  data() {
    return {
      message: "父类数据"
    }
  }
}

</script>
```

组件的生命周期

每个vue组件实例在创建时都需要经历一系列的初始化步骤，比如设置好数据侦听，编译模板，挂载实例到DOM，以及在数据改变时更新DOM。在此过程中，它也会运行被称为生命周期钩子的函数，让开发者有机会在特定阶段运行自己的代码



生命周期函数

- 创建期：beforeCreate created
- 挂载期：beforeMount　mounted
- 更新期：beforeUpdate　updated
- 销毁期：beforeUnmount　unmounted

```
<template>
    <h3>组件生命周期演示</h3>
    <p>{{ msg }}</p>
    <button @click="updateHandle">点击更新</button>
</template>

<script>
export default {
    data() {
        return {
            msg: "原始数据"
        }
    },
    methods: {
        updateHandle() {
            this.msg = "更新后数据"
        }
```

```
    },
    beforeCreate() {
        console.log("组件创建之前");
    },
    created() {
        console.log("组件创建之后");
    },
    beforeMount() {
        console.log("组件挂载之前");
    },
    mounted() {
        console.log("组件挂载之后");
    },
    beforeUpdate() {
        console.log("组件更新之前");
    },
    updated() {
        console.log("组件更新之后");
    },
    beforeUnmount() {
        console.log("组件销毁之前");
    },
    unmounted() {
        console.log("组件销毁之后");
    }
}
</script>
```

生命周期应用

组件的生命周期会随着我们对`vue`的了解越多，也会越来越重要，比如两个常用的应用：

1. 通过`ref`获取元素DOM结构
2. 模拟网络请求渲染数据

```
<template>
    <h3>组件生命周期应用</h3>
    <p ref="name">张三</p>
    <ul>
        <li v-for="user of users" :key="user.id">
            <h4>{{ user.id }}</h4>
            <p>{{ user.name }}</p>
        </li>
    </ul>
</template>

<script>
export default {
    data(I) {
        return {
            users: ""
        }
    },
    beforeMount() {
        // 挂载之前，获取不到DOM
        console.log(this.$refs.name //undefined
    },
```

```
    mounted() {
        //挂载后可以获取到DOM
        console.log(this.$refs.name);
        // 在创建后就可以模拟网络请求，但是建议在挂载后再获取网络数据请求，因为挂载界面要先执行
        this.users = [
            {
                "id": 1001,
                "name": "张三"
            },
            {
                "id": 1002,
                "name": "李四"
            },
            {
                "id": 1003,
                "name": "王五"
            }
        ]

    }
}
</script>
```

动态组件

有些场景会需要在两个组件之间来回切换，如tab界面

```
<template>
    <!-- 通过 :is="" 来设置当前组件 -->
    <component :is="isComponent"></component>
    <button @click="changeComponent">切换组件</button>
</template>

<script>
import ComponentA from './ComponentA.vue';
import ComponentB from './ComponentB.vue';
export default {
    data() {
        return {
            //isComponent是字符串格式
            isComponent: "ComponentA"
        }
    },
    methods: {
        changeComponent() {
            //实现组件之间的切换
            this.isComponent = this.isComponent == "ComponentA" ? "ComponentB" :
"ComponentA"
        }
    },
    components: {
        ComponentA,
        ComponentB
    }
}
</script>
```

组件保持存活

当使用`<component :is="...">`来在多个组件之间做切换时，被切掉的组件会被卸载（切换的时候会重新走一遍组件生命流程），我们通过`<keep-alive>`组件强制被换掉的组件仍然保持"存活"的状态，切换的时候不会再走一遍组件生命周期

```
// ComponentA.vue
<template>
    <h3>ComponentA</h3>
    <p>{{ message }}</p>
    <button @click="updataDataHandle">切换数据</button>
</template>

<script>

export default {
    data() {
        return{
            message: "数据更新之前"
        }
    },
    beforeUnmount() {
        console.log("组件卸载之前");
    },
    unmounted() {
        console.log("组件卸载之后");
    },
    methods: {
        updataDataHandle() {
            this.message = "数据更新之后"
        }
    }
}
</script>

// App.vue
<template>
    <!-- 保持组件存活 -->
    <keep-alive>
        <component :is="isComponent"></component>
    </keep-alive>
    <button @click="changeComponent">切换组件</button>

</template>

<script>
import ComponentA from './ComponentA.vue';
import ComponentB from './ComponentB.vue';
export default {
    data() {
        return {
            isComponent: "ComponentA"
        }
    },
    methods: {
        changeComponent() {
```

```
            this.isComponent = this.isComponent == "ComponentA" ? "ComponentB" :
"ComponentA"
        }
    },
    components: {
        ComponentA,
        ComponentB
    }
}
</script>
```

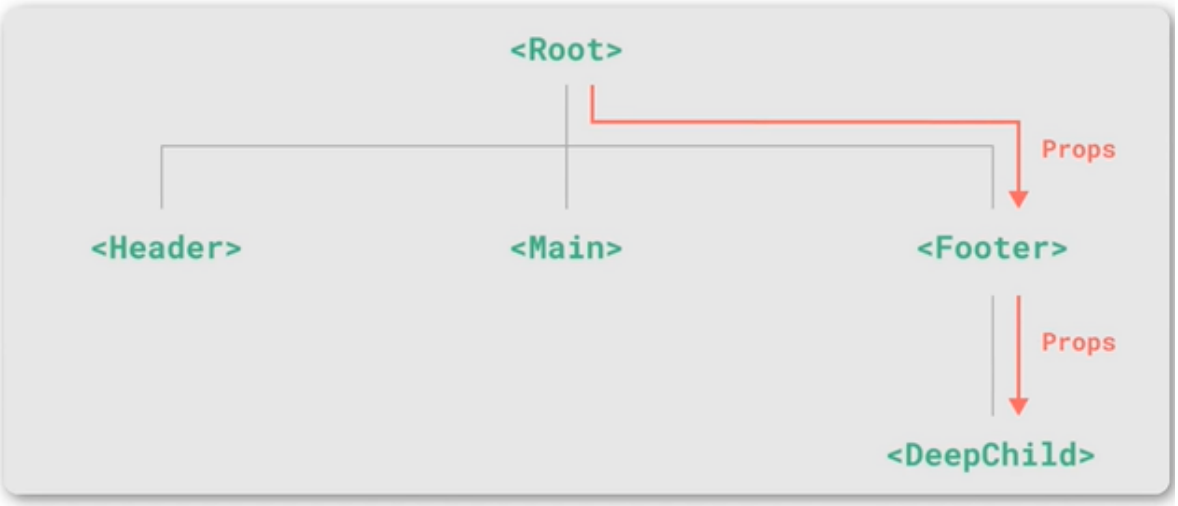
异步组件

在大型项目中，我们可能需要拆分应用为更小的块，并仅在需要时再从服务器加载相关组件。vue提供了`defineAsyncComponent`方法来实现此功能，这在大型项目中十分有用

```
<template>
    <keep-alive>
        <component :is="isComponent"></component>
    </keep-alive>
    <button @click="changeComponent">切换组件</button>

</template>

<script>
import ComponentA from './ComponentA.vue';
// import ComponentB from './ComponentB.vue';
import { defineAsyncComponent } from 'vue';
// 异步加载组件
const ComponentB = defineAsyncComponent(() =>
    import('./ComponentB.vue')
)
export default {
    data() {
        return {
            isComponent: "ComponentA"
        }
    },
    methods: {
        changeComponent() {
            this.isComponent = this.isComponent == "ComponentA" ? "ComponentB" :
"ComponentA"
        }
    },
    components: {
        ComponentA,
        ComponentB
    }
}
</script>
```

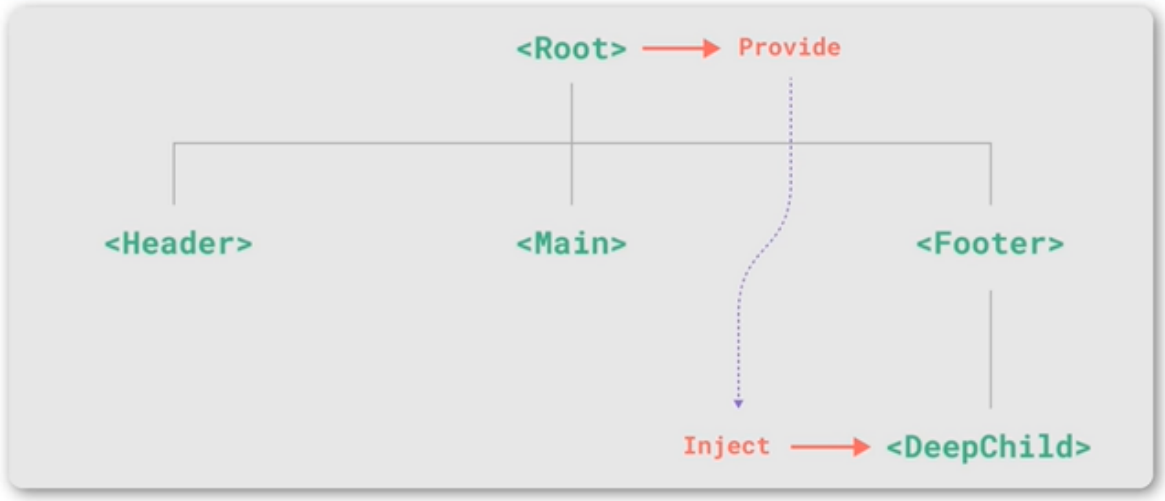
依赖注入

通常情况下，当我们需要从父组件向子组件传递数据时，会使用`props`。想象一下这样的结构：有一些多层级嵌套的组件，形成了一颗巨大的组件树，而某个深层的子组件需要一个较远的祖先组件中的部分数据。在这种情况下，如果仅使用 props 则必须将其沿着组件链逐级传递下去，这会非常麻烦



这一问题被称为“prop逐级透传”

`provide`和`inject`可以帮助我们解决这一问题。一个父组件相对于其所有的后代组件，会作为**依赖提供者**。任何后代的组件树，无论层级有多深，都可以**注入**由父组件提供给整条链路的依赖。



Provide(提供)

要为组件后代提供数据，需要使用到`provide`选项

```
// RootComponent.vue
<template>
    <h3>RootComponent</h3>
    <ParentComponent />
</template>

<script>
import ParentComponent from './ParentComponent.vue';

export default {
    data() {
        return {

        }
    },
    components: {
        ParentComponent
    },
```

```
    provide: {
        message: "根节点的消息"
    }
}
</script>

// ChildComponent.vue
<template>
    <p>ChildComponent</p>
    <P>{{ message }}</P>
</template>

<script>
import { inject } from 'vue';

export default {
    data() {
        return {

        }
    },
    inject: ["message"]
}
</script>
```

也可以读取`data`中的数据

```
// RootComponent.vue
<template>
    <h3>RootComponent</h3>
    <ParentComponent />
</template>

<script>
import ParentComponent from './ParentComponent.vue';

export default {
    data() {
        return {
            message: "根节点的第二个消息"
        }
    },
    components: {
        ParentComponent
    },
    // provide: {
    //     message: "根节点的消息"
    // },
    // 如果要返回data中的数据，需要使用函数式返回
    provide() {
        return {
            message: this.message
        }
    }
}
</script>
```

除了在一个组件中提供依赖，我们还可以在整个应用层面提供依赖，子类以同样的方式获取数据

```
//main.js
//import './assets/main.css'

import { createApp } from 'vue'
import App from './App.vue'

const app = createApp(App)

// 设置全局数据
app.provide("msg", "这是全局数据")

app.mount('#app')
```

inject(注入)

子类直接通过`inject`获取数据

```
<template>
    <p>ChildComponent</p>
    <P>{{ message }}</P>
</template>

<script>
import { inject } from 'vue';

export default {
    data() {
        return {

        }
    },
    inject: ["message"]
}
</script>
```

子类也可以将数据存入data中再取出

```
<template>
    <p>ChildComponent</p>
    <P>{{ fullMessage }}</P>
    <P>{{ msg }}</P>
</template>

<script>
import { inject } from 'vue';

export default {
    data() {
        return {
            fullMessage: this.message
        }
    },
    inject: ["message", "msg"]
}
```

```
</script>
```

> 注意：
> `provide`和`inject`只能由上到下的传递，不能反向传递

Vue应用

应用实例

每个Vue应用都是通过`createApp`函数创建一个新的**应用实例**

```
import { createApp } from 'vue'
const app = createApp({
    //根组件选项
})
```

根组件

我们传入`createApp`的对象实际上是一个组件，每个应用都需要一个"根组件"，其他组件将作为其子组件

```
import { createApp } from 'vue'
// 从一个单文件组件中导入根组件
import App from './App.vue'
// app: vue的实例对象
// 在一个vue项目中，有且只有一个vue的实例对象
const app = createApp(App)
// app就是一个根组件
app.mount('#app')
```

挂载应用

应用实例必须在调用了`.mount()`方法后才会渲染出来，该方法接收一个“容器”参数，可以是一个实际的DOM元素或者是一个CSS选择器字符串

> app.mount('#app') // #app是指项目目录中的index.html中的div标签

>

公共资源

在`src`目录下的`assets`文件夹的作用就是存放公共资源，例如：图片、公共CSS或者字体图标等