

# 小程序的组件化开发

王红元 coderwhy

# 目录

## content



**1** 小程序组件化思想

**2** 自定义组件的过程

**3** 组件样式实现细节

**4** 组件使用过程通信

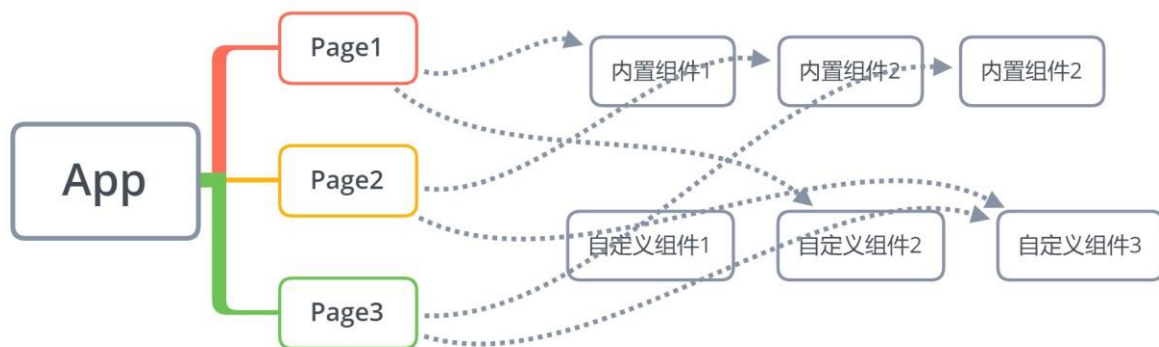
**5** 组件插槽定义使用

**6** Component构造器

# 小程序组件化开发

## ■ 小程序在刚刚推出时是不支持组件化的，也是为人诟病的一个点：

- 但是从v1.6.3开始，小程序开始支持自定义组件开发，也让我们更加方便的在程序中使用组件化。



## ■ 组件化思想的应用：

- 有了组件化的思想，我们在之后的开发中就要充分的利用它。
- 尽可能的将页面拆分成一个个小的、可复用的组件。
- 这样让我们的代码更加方便组织和管理，并且扩展性也更强。

## ■ 所以，组件是目前小程序开发中，非常重要的一个篇章，要认真学习（不过学习Vue的过程中我们已经强调过很多次了）。

# 创建一个组件

## ■ 类似于页面，自定义组件由 json wxml wxss js 4个文件组成。

- 按照我的个人习惯, 我们会先在根目录下创建一个文件夹;
- components, 里面存放我们之后自定义的公共组件;
- 常见一个自定义组件 my-cpn: 包含对应的四个文件;

## ■ 自定义组件的步骤:

- 1.首先需要在 json 文件中进行自定义组件声明 (将component 字段设为 true 可这一组文件设为自定义组件) :
- 2.在wxml中编写属于我们组件自己的模板
- 3.在wxss中编写属于我们组件自己的相关样式
- 4.在js文件中, 可以定义数据或组件内部的相关逻辑(后续我们再使用)

```
my-cpn.json ×
components > my-cpn > my-cpn.js
1 {
2   "component": true,
3   "usingComponents": {}
4 }
```

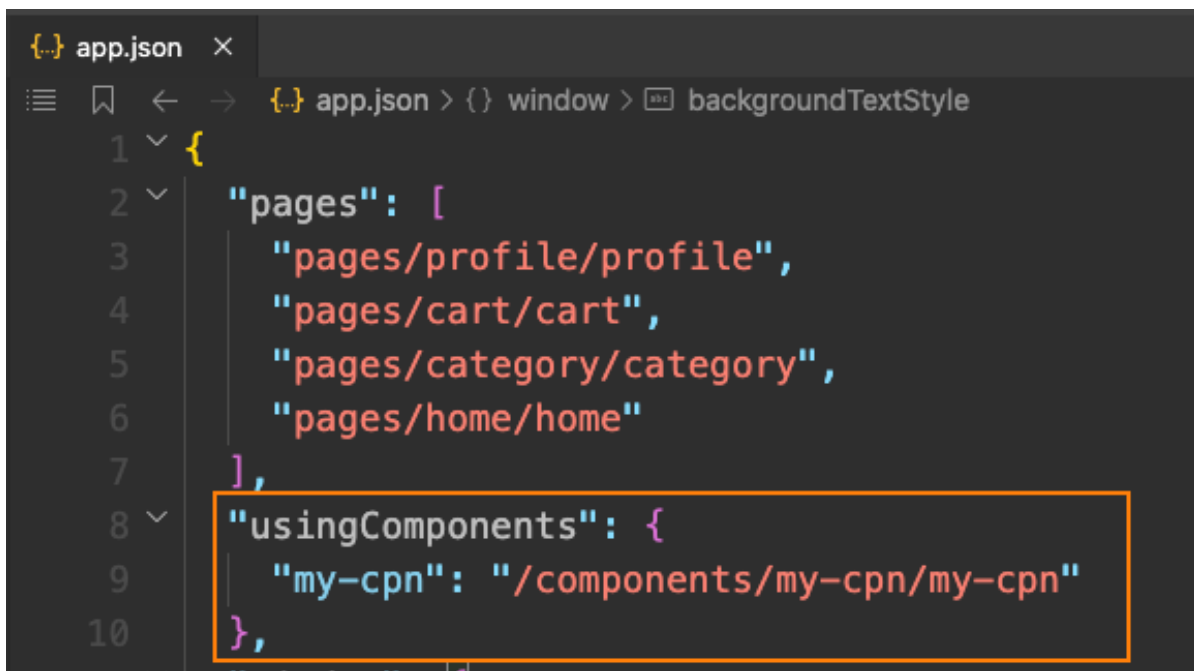
```
my-cpn.wxml ×
components > my-cpn > my-cpn.wxml > ...
<!--components/my-cpn/my-cpn.wxml-->
<view class="my-cpn">
  <view class="title">我是组件标题</view>
  <view class="content">我是组件内容，哈哈</view>
</view>
```

```
my-cpn.json  my-cpn.wxml  my-cpn.js ×
components > my-cpn > my-cpn.js > ...
1 // components/my-cpn/my-cpn.js
2 Component({
3   properties: {
4
5   },
6   data: {
7
8   },
9   methods: {
10
11   }
12 })
```

# 使用自定义组件和细节注意事项

## ■ 一些需要注意的细节:

- 自定义组件也是可以引用自定义组件的，引用方法类似于页面引用自定义组件的方式（使用usingComponents 字段）。
- 自定义组件和页面所在项目根目录名 不能以 “wx-” 为前缀，否则会报错。
- 如果在app.json的usingComponents声明某个组件，那么所有页面和组件可以直接使用该组件。



```
{
  "pages": [
    "pages/profile/profile",
    "pages/cart/cart",
    "pages/category/category",
    "pages/home/home"
  ],
  "usingComponents": {
    "my-cpn": "/components/my-cpn/my-cpn"
  }
}
```

# 组件的样式细节

## ■ 课题一：组件内的样式 对 外部样式 的影响

- 结论一：组件内的class样式，只对组件wxml内的节点生效，对于引用组件的Page页面不生效。
- 结论二：组件内不能使用id选择器、属性选择器、标签选择器

## ■ 课题二：外部样式 对 组件内样式 的影响

- 结论一：外部使用class的样式，只对外部wxml的class生效，对组件内是不生效的
- 结论二：外部使用了id选择器、属性选择器不会对组件内产生影响
- 结论三：外部使用了标签选择器，会对组件内产生影响

## ■ 课题三：如何让class可以相互影响

- 在Component对象中，可以传入一个options属性，其中options属性中有一个styleIsolation（隔离）属性。
- styleIsolation有三个取值：
  - ✓ - **isolated** 表示启用样式隔离，在自定义组件内外，使用 class 指定的样式将不会相互影响（默认取值）；
  - ✓ - **apply-shared** 表示页面 wxss 样式将影响到自定义组件，但自定义组件 wxss 中指定的样式不会影响页面；
  - ✓ - **shared** 表示页面 wxss 样式将影响到自定义组件，自定义组件 wxss 中指定的样式也会影响页面和其他设置 了

# 组件的通信

- 很多情况下，组件内展示的内容（数据、样式、标签），并不是在组件内写死的，而且可以由使用者来决定。



# 向组件传递数据 - properties

## ■ 给组件传递数据：

- 大部分情况下，组件只负责布局和样式，内容是由使用组件的对象决定的；
- 所以，我们经常需要从外部传递数据给我们的组件，让我们的组件来进行展示；

## ■ 如何传递呢？

- 使用properties属性；

## ■ 支持的类型：

- String、Number、Boolean
- Object、Array、null（不限制类型）

## ■ 默认值：

- 可以通过value设置默认值；

```
properties: {  
  title: {  
    type: String,  
    value: "默认标题"  
  },  
  content: {  
    type: String,  
    value: "默认内容"  
  }  
},
```



# 向组件传递样式 - externalClasses

## ■ 给组件传递样式：

□ 有时候，我们不希望将样式在组件内固定不变，而是外部可以决定样式。

## ■ 这个时候，我们可以使用externalClasses属性：

- 1.在Component对象中，定义externalClasses属性
- 2.在组件内的wxml中使用externalClasses属性中的class
- 3.在页面中传入对应的class，并且给这个class设置样式

```
<view class="my-cpn" id="test">
  <view class="title">我是组件标题: {{ title }}</view>
  <view class="content info">我是组件内容，哈哈: {{ content }}</view>
</view>
```

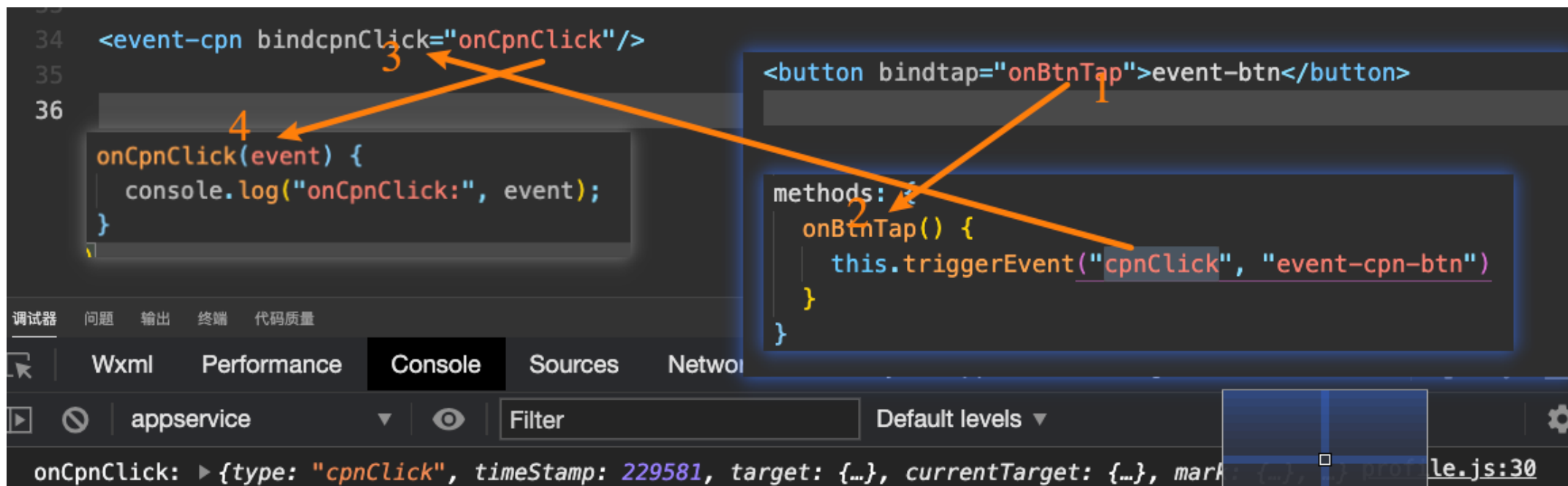
```
<my-cpn info="info"/>
```

```
Component({
  options: {
    styleIsolation: "isolated"
  },
  externalClasses: ["info"],
```

```
.info {
  background-color: orange;
}
```

# 组件向外传递事件 – 自定义事件

- 有时候是自定义组件内部发生了事件，需要告知使用者，这个时候可以使用自定义事件：



# 自定义组件练习

流行

新款

精选

```
<view class="tab-control">
  <block wx:for="{{titles}}" wx:key="*this">
    <view
      class="item {{ index === currentIndex ? 'active': '' }}"
      bindtap="onItemTap"
      mark-index="{{index}}"
    >
      <text class="title">{{ item }}</text>
    </view>
  </block>
</view>
```

```
methods: {
  onItemTap(event) {
    const index = event.mark.index
    this.setData({ currentIndex: index })

    this.triggerEvent("tabchange", index)
  }
}
```

```
<tab-control titles="{{[['电脑', '手机', '平板']]]}" bindtabchange="onTabChange"/>
```

# 页面直接调用组件方法

■ 可在父组件里调用 `this.selectComponent`，获取子组件的实例对象。

□ 调用时需要传入一个匹配选择器 `selector`，如：`this.selectComponent(".my-component")`。

```
<tab-control class="tabs" titles="{['电脑', '手机', '平板']}" bindtabchange="onTabChange"/>
<button bindtap="onChangeBtnTap">修改tab</button>
```

```
onChangeBtnTap() {
  const tabControl = this.selectComponent(".tabs")
  tabControl.innerTest()
}
```

# 什么是插槽？

## ■ slot翻译为插槽：

- 在生活中很多地方都有插槽，电脑的USB插槽，插板当中的电源插槽。
- 插槽的目的是让我们原来的设备具备更多的扩展性。
- 比如电脑的USB我们可以插入U盘、硬盘、手机、音响、键盘、鼠标等等。

## ■ 组件的插槽：

- 组件的插槽也是为了让我们封装的组件更加具有扩展性。
- 让使用者可以决定组件内部的一些内容到底展示什么。

## ■ 栗子：移动网站中的导航栏。

- 移动开发中，几乎每个页面都有导航栏。
- 导航栏我们必然会封装成一个插件，比如nav-bar组件。
- 一旦有了这个组件，我们就可以在多个页面中复用了。

## ■ 但是，每个页面的导航是一样的吗？类似右图



# 单个插槽的使用

## ■ 除了内容和样式可能由外界决定之外，也可能外界想决定显示的方式

- 比如我们有一个组件定义了头部和尾部，但是中间的内容可能是一段文字，也可能是一张图片，或者是一个进度条。
- 在不确定外界想插入什么其他组件的前提下，我们可以在组件内预留插槽：

```
<view>
  <view>header</view>
  <view class="content">
    <slot></slot>
  </view>
  <view>footer</view>
</view>
```

```
<!-- 1.插入文本 -->
<my-slot>
  <text>我是文本</text>
</my-slot>
<!-- 2.插件按钮 -->
<my-slot>
  <button>哈哈</button>
</my-slot>
<!-- 3.插入图片 -->
<my-slot>
  <image src="/assets/nhlt.jpg"></image>
</my-slot>
```

# 多个插槽的使用

- 有时候为了让组件更加灵活, 我们需要定义多个插槽:

```
<view class="slot">
  <view class="left">
    <slot name="left"></slot>
  </view>
  <view class="center">
    <slot name="center"></slot>
  </view>
  <view class="right">
    <slot name="right"></slot>
  </view>
</view>
```

```
Component({
  options: {
    multipleSlots: true
  }
})
```

```
<m-slot>
  <text slot="left">左边</text>
  <view slot="center">标题</view>
  <text slot="right">右边</text>
</m-slot>
```

# behaviors

■ behaviors 是用于组件间代码共享的特性，类似于一些编程语言中的 “mixins”。

- 每个 behavior 可以包含一组属性、数据、生命周期函数和方法；
- 组件引用它时，它的属性、数据和方法会被合并到组件中，生命周期函数也会在对应时机被调用；
- 每个组件可以引用多个 behavior，behavior 也可以引用其它 behavior；

```
const counter = Behavior({
  data: {
    counter: 0
  },
  lifetimes: {
    created() {
      console.log("counter behavior created");
    }
  },
  methods: {
    increment() {
      this.setData({
        counter: this.data.counter + 1
      })
    }
  }
})
```

```
import { counter } from "../../behaviors/counter"

Component({
  behaviors: [counter],
  options: {
    multipleSlots: true
  }
})
```





# 组件的生命周期

- 组件的生命周期，指的是组件自身的一些函数，这些函数在特殊的时间点或遇到一些特殊的框架事件时被自动触发。
  - 其中，最重要的生命周期是 created attached detached ，包含一个组件实例生命流程的最主要时间点。
- 自小程序基础库版本 2.2.3 起，组件的的生命周期也可以在 lifetimes 字段内进行声明（这是推荐的方式，其优先级最高）。

生命周期	参数	描述	最低版本
created	无	在组件实例刚刚被创建时执行	1.6.3
attached	无	在组件实例进入页面节点树时执行	1.6.3
ready	无	在组件在视图层布局完成后执行	1.6.3
moved	无	在组件实例被移动到节点树另一个位置时执行	1.6.3
detached	无	在组件实例被从页面节点树移除时执行	1.6.3
error	Object Error	每当组件方法抛出错误时执行	2.4.1

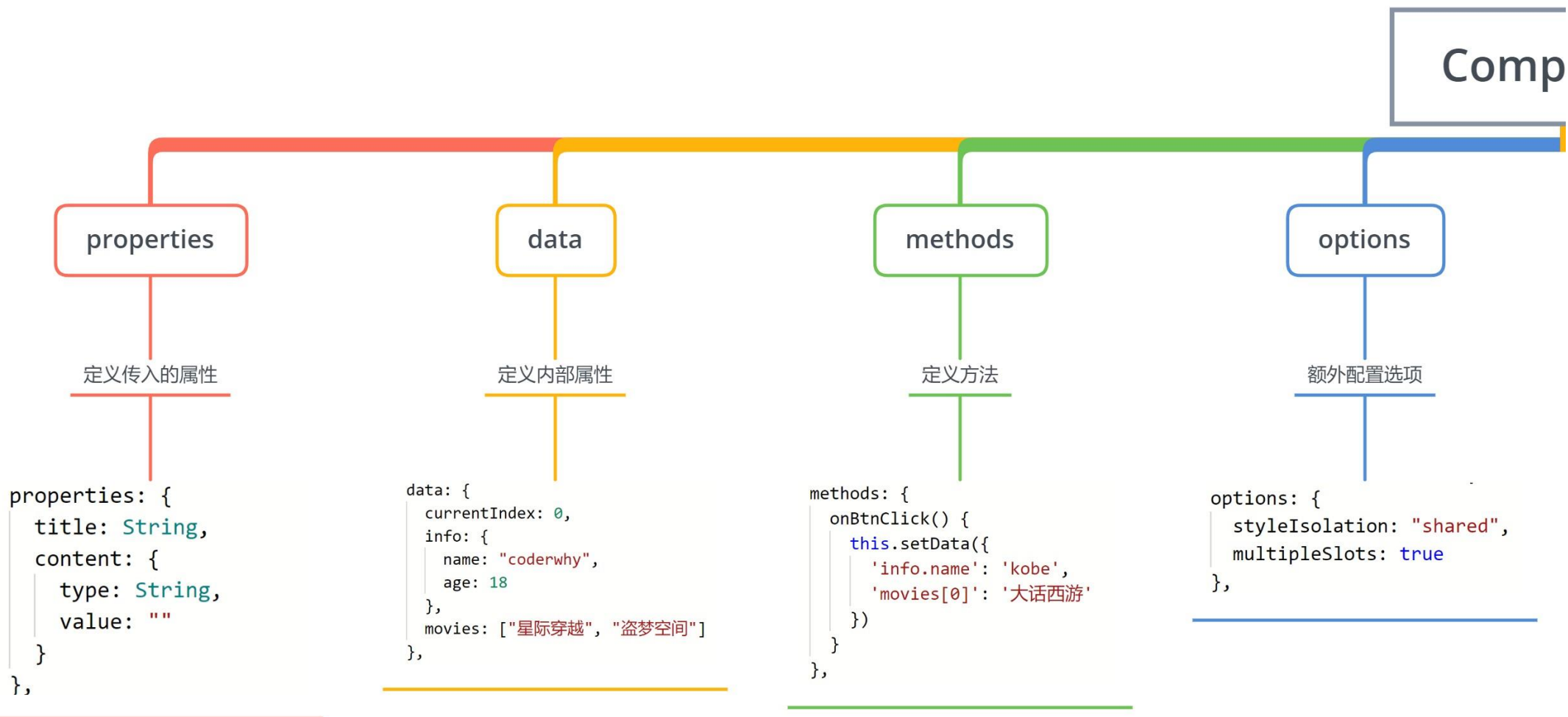


# 组件所在页面的生命周期

- 还有一些特殊的生命周期，它们并非与组件有很强的关联，但有时组件需要获知，以便组件内部处理。
  - 样的生命周期称为“组件所在页面的生命周期”，在 pageLifetimes 定义段中定义。
- 其中可用的生命周期包括：

生命周期	参数	描述	最低版本
show	无	组件所在的页面被展示时执行	2.2.3
hide	无	组件所在的页面被隐藏时执行	2.2.3
resize	Object Size	组件所在的页面尺寸变化时执行	2.4.0

# Component构造器



# Component构造器

Component

externalClasses

引用外部样式

```
externalClasses: ['title'],
```

observers

属性和数据监听

```
observers: {  
  title: function(newVal, oldVal) {  
    console.log(newVal, oldVal)  
  },  
  "info.name": function(newVal) {  
    console.log(newVal)  
  },  
  "movies[0]": function(newVal) {  
    console.log(newVal)  
  }  
},
```

pageLifetimes

页面生命周期

```
pageLifetimes: {  
  show() {  
    console.log('页面显示出来')  
  },  
  hide() {  
    console.log('页面隐藏起来')  
  },  
  resize() {  
    console.log('尺寸发生改变')  
  }  
},
```

lifetimes

组件生命周期

```
lifetimes: {  
  created() {  
    console.log('组件被创建')  
  },  
  attached() {  
    console.log('组件被添加到页面中')  
  },  
  ready() {  
    console.log('组件被渲染出来')  
  },  
  moved() {  
    console.log('组件被移动到节点树另一个位置')  
  },  
  detached() {  
    console.log('组件被移除掉')  
  }  
}
```