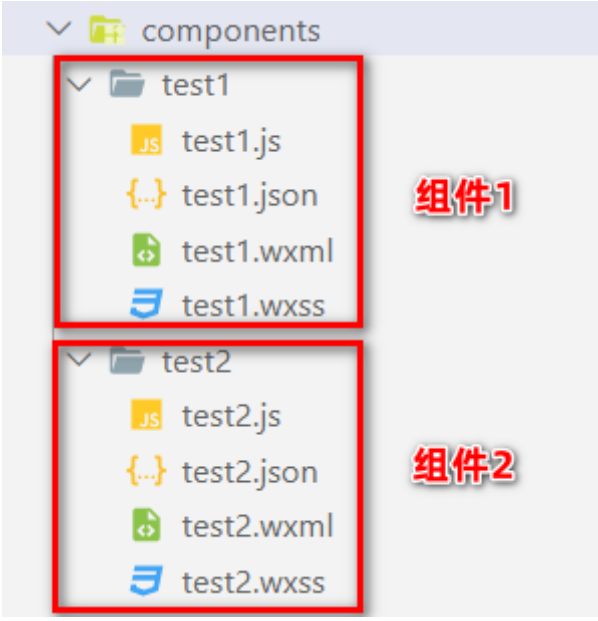
自定义组件

1、组件的创建与引用

1.1、创建组件

① 在项目的根目录中，鼠标右键，创建 components -> test 文件夹
② 在新建的 components -> test 文件夹上，鼠标右键，点击“新建Component”
③ 键入组件的名称之后回车，会自动生成组件对应的 4 个文件，后缀名分别为 .js，.json， .wxml 和 .wxss
注意：为了保证目录结构的清晰，建议把不同的组件，存放到单独目录中，例如：



1.2、引用组件

组件的引用方式分为“局部引用”和“全局引用”，顾名思义：

1. 局部引用：组件只能在当前被引用的页面内使用
2. 全局引用：组件可以在每个小程序页面中使用

1.3、局部引用组件

在页面的 .json 配置文件中引用组件的方式，叫做“局部引用”。示例代码如下：

```
{
    "usingComponents": {
        "my-test1": "/components/test1/test1"
    }
}
```

```
<my-test1></my-test1>
```

1.4、全局引用组件

在 app.json 全局配置文件中引用组件的方式，叫做“全局引用”。示例代码如下：

```
{
    "pages":[],
    "window": {},
    "usingComponents": {
        "my-test2": "/components/test2/test2"
    }
}
```

1.5、全局引用 VS 局部引用

根据组件的使用频率和范围，来选择合适的引用方式：

1. 如果某组件在多个页面中经常被用到，建议进行“全局引用”
2. 如果某组件只在特定的页面中被用到，建议进行“局部引用”

1.6、组件和页面的区别

从表面来看，组件和页面都是由 .js、.json、.wxml 和 .wxss 这四个文件组成的。但是，组件和页面的.js 与
.json 文件有明显的不同：

1. 组件的 .json 文件中需要声明 "component": true 属性
2. 组件的 .js 文件中调用的是 Component() 函数
3. 组件的事件处理函数需要定义到 methods 节点中

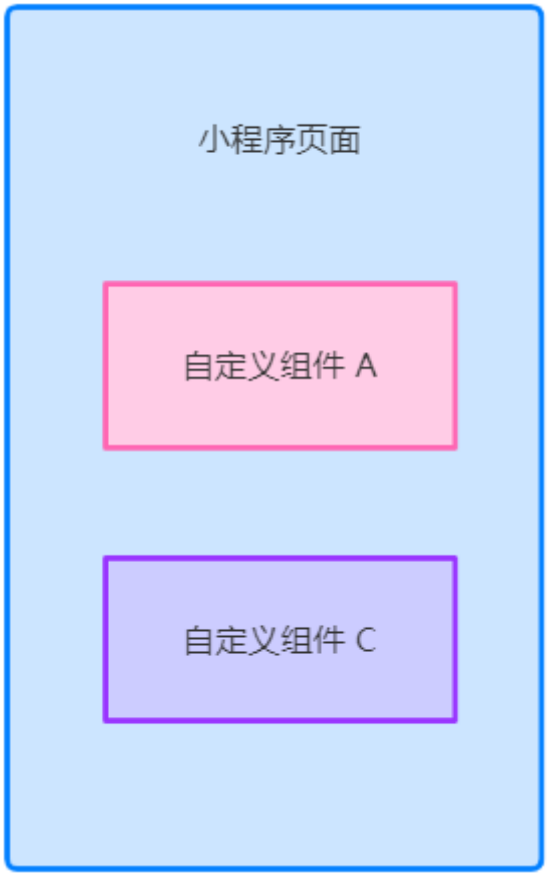
2、 样式

2.1、组件样式隔离

默认情况下，自定义组件的样式只对当前组件生效，不会影响到组件之外的
UI 结构，如图所示：

1. 组件 A 的样式不会影响组件 C 的样式
2. 组件 A 的样式不会影响小程序页面的样式
3. 小程序页面的样式不会影响组件 A 和 C 的样式

好处：
① 防止外界的样式影响组件内部的样式
② 防止组件的样式破坏外界的样式



2.2、组件样式隔离的注意点

1. app.wxss 中的全局样式对组件无效
2. 只有 class 选择器会有样式隔离效果，id 选择器、属性选择器、标签选择器不受样式隔离的影响
   建议：在组件和引用组件的页面中建议使用 class 选择器，不要使用 id、属性、标签选择器！

2.3、修改组件的样式隔离选项

默认情况下，自定义组件的样式隔离特性能够防止组件内外样式互相干扰的问题。但有时，我们希望在外界能
够控制组件内部的样式，此时，可以通过 styleIsolation 修改组件的样式隔离选项，用法如下：

```
Component({
    options: {
        stylesolation: 'isolated'
    }
})
```

```
{
    "styleIsolation": "isolated"
}
```

2.4、styleIsolation 的可选值

可选值	默认值	描述
isolated	是	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
apply-shared	否	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	否	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 apply-shared 或 shared 的自定义组件

3、数据、方法和属性

3.1、data 数据

在小程序组件中，用于组件模板渲染的私有数据，需要定义到 data 节点中，示例如下：

```
Component({
    data: {
        count:0,
    }
})
```

3.2、 methods 方法

在小程序组件中，事件处理函数和自定义方法需要定义到 methods 节点中，示例代码如下：

```
Component({
    methods: {
        addCount() {
            this.setData({count: this.data.count + 1})
            this._showCount()
        },
        _showCount() {
            wx.showToast({
                title: 'count值为：' + this.data.count,
                icon: 'none'
            })
        }
    }
})
```

3.3、 properties 属性

在小程序组件中，properties 是组件的对外属性，用来接收外界传递到组件中的数据，示例代码如下：

```
Component({
    properties: {
        max: {
            type: Number,
            value: 10
        },
        max: Number
    }
})
```

```
<my-test1 max="10"></my-test1>
```

3.4、data 和 properties 的区别

在小程序的组件中，properties 属性和 data 数据的用法相同，它们都是可读可写的，只不过：

1. data 更倾向于存储组件的私有数据
2. properties 更倾向于存储外界传递到组件中的数据

```
Component({
    methods: {
        showInfo() {
            console.log(this.data);
            console.log(this.properties);
            console.log(this.data ===  this.properies)
        }
    }
})
```

3.5、使用 setData 修改 properties 的值

由于 data 数据和 properties 属性在本质上没有任何区别，因此properties 属性的值也可以用于页面渲染，
或使用 setData 为 properties 中的属性重新赋值，示例代码如下：

```
Component({
    properties: {max: Number},
    methods: {
        addCount() {
            this.setData({max: this.properties.max + 1})
        }
    }
})
```

```
<view>max属性值为：</view>
```

4、 数据监听器

4.1、什么是数据监听器

数据监听器用于监听和响应任何属性和数据字段的变化，从而执行特定的操作。它的作用类似于vue 中的
watch 侦听器。在小程序组件中，数据监听器的基本语法格式如下：

```
Component({
    observers: {
        '字段A,字段B':function(字段A新值,字段B新值){}
    }
})
```

4.2、数据监听器的基本用法

```
<view>{{n1}} + {{n2}} = {{sum}}</view>
<button size="mini" bindtap="addN1">n1自增</button>
<button size="mini" bindtap="addN2">n2自增</button>
```

```
Component({
    data: {n1: 0, n2:0, sum: 0},
    methods: {
        addN1() { this.setData({n1: this.data.n1 + 1})},
        addN2() { this.setData({n2: this.data.n2 + 1})},
    },
    observ3ers: {
        'n1,n2':function(n1,n2) {
            this.setData({sum: n1+ n2})
        }
    }
})
```

4.3、监听对象属性的变化

数据监听器支持监听对象中单个或多个属性的变化，示例语法如下：

```
Component({
    observers: {
        '对象.属性A,对象.属性B':function(属性A新值,属性B新值){}
    }
})
```

如果某个对象中需要被监听的属性太多，为了方便，可以使用通配符 ** 来监听对象中所有属性的变化，示例
代码如下：

```
Component({
    observers: {
        '对象.**':function(obj) {
            this.setData({
                fullColor: `${obj.r},${obj.g},${obj.b}`
            })
        }
    }
})
```

5、纯数据字段

5.1、什么是纯数据字段

概念：纯数据字段指的是那些不用于界面渲染的data 字段。
应用场景：例如有些情况下，某些 data 中的字段既不会展示在界面上，也不会传递给其他组件，仅仅在当前
组件内部使用。带有这种特性的 data 字段适合被设置为纯数据字段。
好处：纯数据字段有助于提升页面更新的性能。

5.2、使用规则

在 Component 构造器的 options 节点中，指定 pureDataPattern 为一个正则表达式，字段名符合这个正则
表达式的字段将成为纯数据字段，示例代码如下：

```
Component({
    options: {
        pureDataPattern: /^_/
    },
    data: {
    a: true,
    _b: true
}
})
```

6、组件的生命周期

6.1、组件全部的生命周期函数

小程序组件可用的全部生命周期如下表所示：

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

6.2、 组件主要的生命周期函数

在小程序组件中，最重要的生命周期函数有 3 个，分别是 created、attached、detached。它们各自的特点
如下：
① 组件实例刚被创建好的时候，created 生命周期函数会被触发

1. 此时还不能调用 setData
2. 通常在这个生命周期函数中，只应该用于给组件的this 添加一些自定义的属性字段

② 在组件完全初始化完毕、进入页面节点树后， attached 生命周期函数会被触发

1. 此时， this.data 已被初始化完毕
2. 这个生命周期很有用，绝大多数初始化的工作可以在这个时机进行（例如发请求获取初始数据）

③ 在组件离开页面节点树后， detached 生命周期函数会被触发

1. 退出一个页面时，会触发页面内每个自定义组件的detached 生命周期函数
2. 此时适合做一些清理性质的工作

6.3、lifetimes 节点

在小程序组件中，生命周期函数可以直接定义在 Component 构造器的第一级参数中，可以在lifetimes 字段
内进行声明（这是推荐的方式，其优先级最高）。示例代码如下：

```
Component({
    lifetimes: {
        attached() {},
        detached() {},
    },
    attached() {},
    detached() {},
})
```

7、组件所在页面的生命周期

7.1、什么是组件所在页面的生命周期

有时，自定义组件的行为依赖于页面状态的变化，此时就需要用到组件所在页面的生命周期。
例如：每当触发页面的show 生命周期函数的时候，我们希望能够重新生成一个随机的 RGB 颜色值。
在自定义组件中，组件所在页面的生命周期函数有如下 3 个，分别是：

生命周期函数	参数	描述说明
show	无	组件所在的页面被展示时执行
hide	无	组件所在的页面被隐藏时执行
resize	Oject Size	组件所在的页面尺寸变化时执行

7.2、pageLifetimes 节点

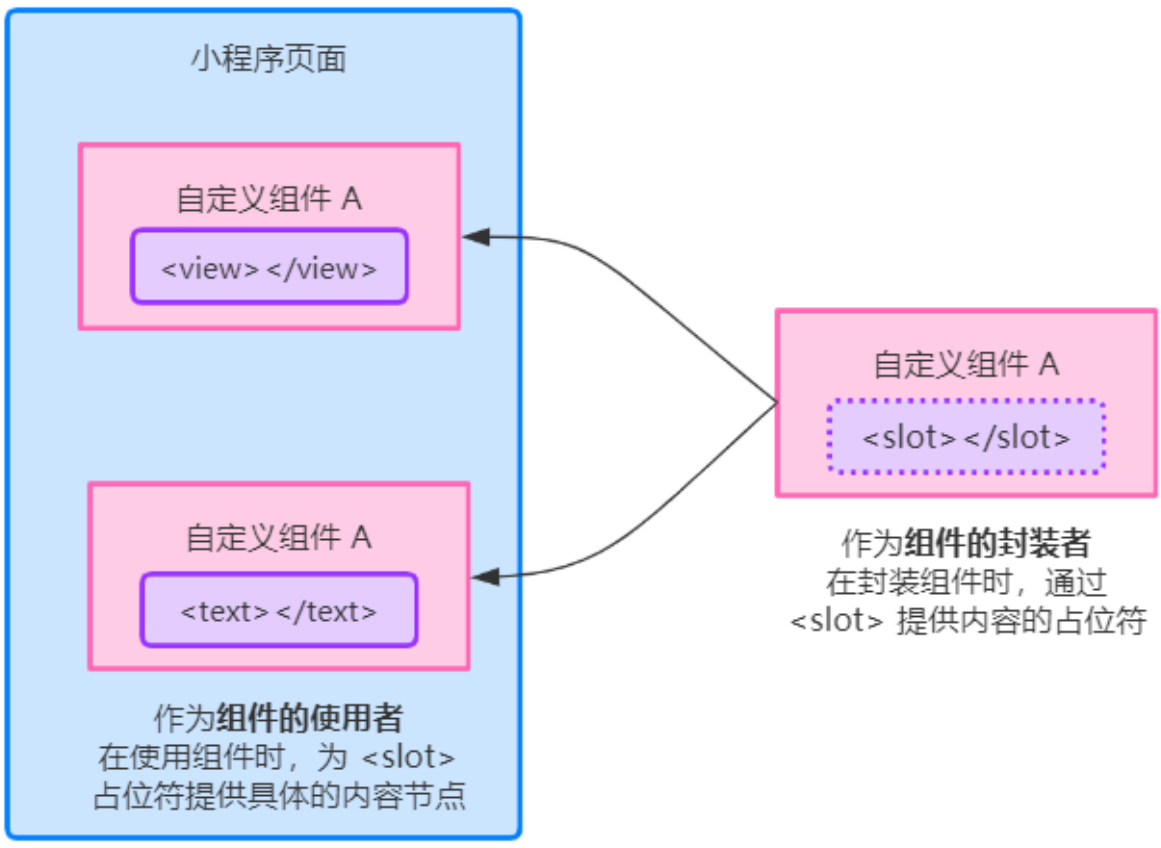
件所在页面的生命周期函数，需要定义在 pageLifetimes 节点中，示例代码如下：

```
Component({
    pageLifetimes : {
        show() {},
        hide() {},
        resize: function(size) {}
    },
})
```

8、插槽

8.1、什么是插槽

在自定义组件的 wxml 结构中，可以提供一个 节点（插槽），用于承载组件使用者提供的 wxml 结构。



8.2、单个插槽

在小程序中，默认每个自定义组件中只允许使用一个 进行占位，这种个数上的限制叫做单个插槽。

```
<view class="wrapper">
    <view>这里是组件内部节点</view>
    <slot></slot>
</view>


<my-test>
    <view>这里是插入组件slot中的内容</view>
</my-test>
```

8.3、启用多个插槽

在小程序的自定义组件中，需要使用多 插槽时，可以在组件的 .js 文件中，通过如下方式进行启用。示例代码如下：

```
Component({
    options: {
        multipleSlots: true
    },
    properties:{},
    mehtods: {}
})
```

8.4、 定义使用多个插槽

可以在组件的 .wxml 中使用多个 标签，以不同的 name 来区分不同的插槽。示例代码如下：

```
<view class="wrapper">
    <slot name="before"></slot>
    <view>这是一段固定的文本内容</view>
    <slot name="after"></slot>
</view>
```

在使用带有多个插槽的自定义组件时，需要用 slot 属性来将节点插入到不同的 中。示例代码如下：

```
<my-test>
    <view slot="before">这里是插入到组件的slot,name=before中内容</view>
    <view slot="after">这里是插入到组件的slot,name=after中内容</view>
</my-test>
```

9、父子组件之间的通信

9.1、父子组件之间通信的 3 种方式

① 属性绑定
用于父组件向子组件的指定属性设置数据，仅能设置 JSON 兼容的数据
② 事件绑定

用于子组件向父组件传递数据，可以传递任意数据
③ 获取组件实例
1. 父组件还可以通过 this.selectComponent() 获取子组件实例对象

2. 这样就可以直接访问子组件的任意数据和方法

9.2、属性绑定

属性绑定用于实现父向子传值，而且只能传递普通类型的数据，无法将方法传递给子组件。父组件的示例代码
如下：

```
data: {
    count: 0
}


<my-test count="count"></my-test>

<view>父组件，count值：{{count}}</view>
```

子组件在 properties 节点中声明对应的属性并使用。示例代码如下：

```
properties: {
    count: Number
}

<text>子组件中， count值为：{{count}}</text>
```

9.3、 事件绑定

事件绑定用于实现子向父传值，可以传递任何类型的数据。使用步骤如下：
① 在父组件的 js 中，定义一个函数，这个函数即将通过自定义事件的形式，传递给子组件
② 在父组件的 wxml 中，通过自定义事件的形式，将步骤1 中定义的函数引用，传递给子组件
③ 在子组件的 js 中，通过调用 this.triggerEvent('自定义事件名称', { /* 参数对象 */ }) ，将数据发送到父组件
④ 在父组件的 js 中，通过 e.detail 获取到子组件传递过来的数据

步骤1：在父组件的 js 中，定义一个函数，这个函数即将通过自定义事件的形式，传递给子组件。

```
syncCount() {
    console.log('syncCount')
}
```

步骤2：在父组件的 wxml 中，通过自定义事件的形式，将步骤 1 中定义的函数引用，传递给子组件。

```
<my-test count={{count}} bind: sync="synCount()"></my-test>
<my-test count={{count}} bindsync="synCount()"></my-test>
```

步骤3：在子组件的 js 中，通过调用 this.triggerEvent('自定义事件名称', { /* 参数对象 */ }) ，将数据发送到父组件。

```
<text>自组件值,count值为：{{count}}</text>
<button type="primary" bindtap="addCount()">+ 1</button>
```

```
methods: {
    addCount() {
        this.setData({
            count: this.properties.count + 1
        })
        this.triggerEvent('sync', {value: this.properties.count})
    }
}
```

步骤4：在父组件的 js 中，通过 e.detail 获取到子组件传递过来的数据。

```
syncCount(e) {
    this.setData({
        coutn: e.detail.value
    })
}
```

9.4、获取组件实例

可在父组件里调用 this.selectComponent("id或class选择器") ，获取子组件的实例对象，从而直接访问子组
件的任意数据和方法。调用时需要传入一个选择器，例如 this.selectComponent(".my-component")。

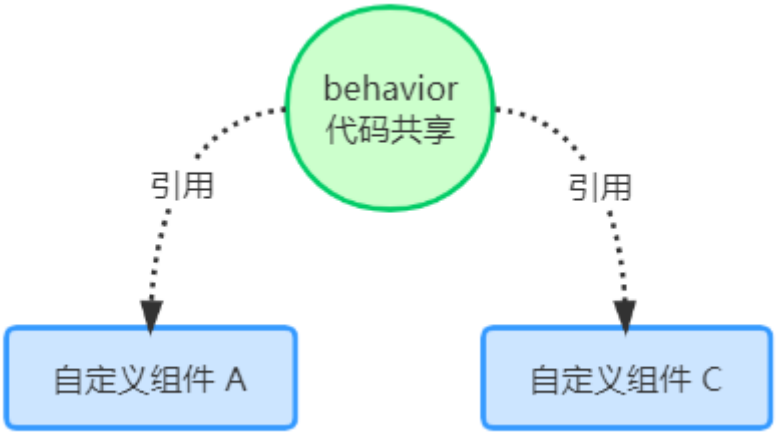
```
<my-test count="{{count}}" bind:sync="syncCount()" class="customA" id="cA"></my-test>
<my-test count="{{count}}" bindtap="getChild()"></my-test>
```

```
getChild() {
    const child = this.selectComponent('.customA')
    child.setData({count: child.properties.count + 1})
    child.addCount()
}
```

10、behaviors

10.1、 什么是 behaviors

behaviors 是小程序中，用于实现组件间代码共享的特性，类似于 Vue.js 中的 “mixins”。



10.2、behaviors 的工作方式

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。
每个组件可以引用多个behavior，behavior 也可以引用其它 behavior。

调用 Behavior(Object object) 方法即可创建一个共享的behavior 实例对象，供所有的组件使用：

10.3、创建 behavior

调用 Behavior(Object object) 方法即可创建一个共享的behavior 实例对象，供所有的组件使用：

```
module.exports= Behavior({
    properties: {},
    data: {username: 'zs'},
    methods: {}
})
```

10.4、导入并使用 behavior

在组件中，使用 require() 方法导入需要的 behavior，挂载后即可访问 behavior 中的数据或方法，示例代码
如下：

```
const myBehavior = require("../../behaviors/my-behavior")

Component({
    behaviors: [myBehavior]
})
```

10.5、 behavior 中所有可用的节点

可用节点	类型	必填	描述
properties	Object Map	否	同组件的属性
data	Object	否	同组件的数据
methods	Object	否	同自定义组件的方法
behaviors	String Array	否	引入其它的 behavior
created	Function	否	生命周期函数
attached	Function	否	生命周期函数
ready	Function	否	生命周期函数
moved	Function	否	生命周期函数
detached	Function	否	生命周期函数

10.6、名字段的覆盖和组合规则*

组件和它引用的 behavior 中可以包含同名的字段，此时可以参考如下 3 种同名时的处理规则：
① 同名的数据字段 (data)
② 同名的属性 (properties) 或方法 (methods)
③ 同名的生命周期函数
关于详细的覆盖和组合规则，大家可以参考微信小程序官方文档给出的说明：
https://developers.weixin.qq.com/miniprogram/dev/framework/custom-component/behaviors.html

11、总结

① 能够创建并引用组件
　全局引用、局部引用、usingComponents
② 能够知道如何修改组件的样式隔离选项
　options -> styleIsolation （ isolated, apply-shared, shared）
③ 能够知道如何定义和使用数据监听器
　observers
④ 能够知道如何定义和使用纯数据字段
　options -> pureDataPattern
⑤ 能够知道实现组件父子通信有哪3种方式
　属性绑定、事件绑定、this.selectComponent(' id或class选择器')
⑥ 能够知道如何定义和使用behaviors
　调用 Behavior() 构造器方法