1.输入属性

星期四，十一月 22，2018　　11:25 上午

我们的组件就是要设定成一个黑盒模型，如果一个组件要从外部接收一些东西，应该用输入属性声明它需要的东西，至于这些东西从哪里来，组件不需要知道。只需要知道外部提供了需要的东西之后，需要怎么做，一个组件想把外部世界可能感兴趣的东西告诉外部世界，应该通过输出属性来发射事件，至于这些事件发射给谁，组件也不需要知道，那些对组件发射东西感兴趣的应该自己来订阅组件发射的事件。

组件的输入属性是指被input装饰器注解的属性，用来从父组件接收数据。

场景很简单，在父组件里输入我买的股票的名字，通过输入属性，传给子组件，子组件显示出来

步骤1：ng g component order ；
在order子组件里定义输入属性

```
// 定义输入属性，等着父组件给她传值
 @Input()
 stockCode: string;

 // 定义输入属性，等着父组件给她传值
 @Input()
 amount: number;
```

步骤2：order组件的html

```
<p>子组件</p>
<div>买{{amount}}手{{stockCode}}股票</div>
```

步骤3：父组件app.component.ts中定义stock

```
stock: ''; // 输入的股票代码默认是一个空的就可以
```

步骤4：　app.component.html

```
<div>我是父组件</div>
<div>
  <!--[(ngModel)]="stock"双向绑定，输入框会改变stock的值-->
  <input type="text" placeholder="股票代码" [(ngModel)]="stock">
  <!--展示子组件的内容，父组件的stock属性通过输入属性传递给子组件的stockCode-->
  <app-order [stockCode]="stock" [amount]="100"></app-order>
</div>
```

效果：

父组件

1

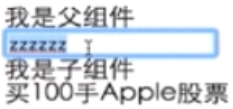
子组件

买100手1股票

尝试改变子组件的stock值

```
constructor() {
  setInterval( handler: () =>
  {
    // 隔三秒改变stockCode的值，
    // 界面会发现父组件没有改变而子组件的stock变为apple
    // 说明属性绑定是单向的，只能父组件影响子组件，
    // 子组件改变不影响父组件的值
    this.stockCode = 'apple';
  }, timeout: 3000)
}
```

效果：3s后



比较输入属性和路由传递参数传递

输入属性：是通过属性来传递数据的，这种传递只能在由父子关系的组件间从父组件给子组件传递数据。(像父组件模板引用另一个形成的父子组件关系才能传递)

路由参数：我们是通过构造函数传递的，在构造函数依赖注入一个ActivatedRoute这样类型的对象进来，通过这个对象的参数快照或参数订阅来获取外面传入的参数。通过路由来往组件里传递数据。

2.输出属性

星期四，十一月 22，2018　　2:14 下午

组件可以使用一个EventEmitter对象来发射自定义的事件，这些事件可以被其他组件处理，EventEmitter是RXJS的subscribe类的一个子类，在响应式编程中，既可以做观察者，又可以做被观察者。EventEmitter对象既可以emit方法发射自定义事件，也可以通过subscribe来订阅EventEmitter发射的事件流。（EventEmitter后的类型同emit类型一样，和父组件声明的一样）

如何使用EventEmitter从组件内部向外部发射事件？

案例：写一个组件，组件链接股票交易所，显示变动价格，为了组件重用，这个组件除了显示价格外，还将最新价格发送到组件之外的其他组件。其他组件可以针对变动的价格执行相应的业务逻辑。（写一个事件，用emit发一个事件，然后父组件接收一个事件。）

步骤1：ng g component priceQuote;
priceQuote.ts

```
export class PriceQuoteComponent implements OnInit {
 /*单独声明两个属性，用来和模板绑定。*/
  stockCode = 'IBM';
  price: number;
  constructor() { }
  ngOnInit() {
  }
}
// 首先定义报价对象，封装股票价格信息。
// 将特定的数据结构，用类或接口来明确的定义是一个良好的习惯。
export class PriceQuote {
  constructor(public stockCode: string, public lastPrice: number) {
  }
```

priceQuote.html

```
<p>
  这里是报价组件
</p>
<div>
  股票代码是{{stockCode}},股票价格是{{price | number:'2.2-2'}}
</div>
```

app.html

```
<app-price-quote></app-price-quote>
```

效果：

这里是报价组件

股票代码是IBM,股票价格是

步骤2：模拟股票价格完成，priceQuote.ts

```
constructor() {
    setInterval(() => {
      // 生成一个变量，类型是priceQuote
      const priceQuote: PriceQuote = new PriceQuote(this.stockCode, 100 * Math.random());
      this.price = priceQuote.lastPrice; /*价格等于最新生成的价格*/
    }, 1000);
```

```
  }
```

效果：

```
这里是报价组件
股票代码是IBM,股票价格是76.48
```

步骤3：priceQuote.ts，把信息发射出去，谁感兴趣谁订阅(步骤4加两句代码)

```
// 信息输出出去，谁感兴趣来订阅它，EventEmitter需要一个泛型。lastPrice,可以发射事件，也可以订阅事件，发射事件，写上output
  @Output()
  lastPrice: EventEmitter<PriceQuote> = new EventEmitter();// 泛型就是你要往外发的数据是什么类型的，
  constructor() {
    setInterval(() => {
      this.lastPrice.emit(priceQuote); /*emit一个值出去，当前最新的报价*/
    }, 1000);
  }
```

步骤4：在父组件中接收然后显示。app.ts

```
priceQuote: PriceQuote = new PriceQuote('', 0);
// 发射的事件是这个类型的，声明这个类型属性，接收发射的报价信息
```

app.html

```
<div>
  这是组件外部，股票代码是{{priceQuote.stockCode}}
  股票价格是{{priceQuote.lastPrice | number:'2.2-2'}}
</div>
```

效果：

```
这里是报价组件
股票代码是IBM,股票价格是68.99
这是组件外部，股票代码是 股票价格是0
```

捕捉 子组件发射的这个事件lastPrice

app.html

```
<!--自定义事件的名字和输出的名字一样@Output('priceChange')需要捕获的那个事件的名字，下面可以改成（priceChange）-->
<app-price-quote （lastPrice)="priceQuoteHandler($event)"></app-price-quote>
```

app.ts

```
priceQuoteHandler(event: PriceQuote) {
// 方法里接收一个event:PriceQuote，事件的类型就是子组件发射emit的类型，父组件通过这个类型就能拿到它。子组件发射的报价就被父组件拿到了。
    this.priceQuote = event;
// this.priceQuote =event;把本地的变量等于 捕获的事件
  }
```

效果：

这里是报价组件

股票代码是IBM,股票价格是76.74
这是组件外部，股票代码是IBM 股票价格是76.74

默认情况下，自定义属性的名字和输出属性的名字是一样的，都是lastPrice，如果要用另外一个名字发射事件，只要指定output装饰器属性就可以了，比如把@Output改成@Output('priceChange')，priceChange就是需要捕获的那个事件的名字，之后再在html把lastPrice改成priceChange就行

priceQuote.ts

```
@Output('priceChange')
  // <PriceQuote>就是一个泛型，泛型就是你要往外发的数据是什么类型的，
lastPrice: EventEmitter<PriceQuote> = new EventEmitter();
```

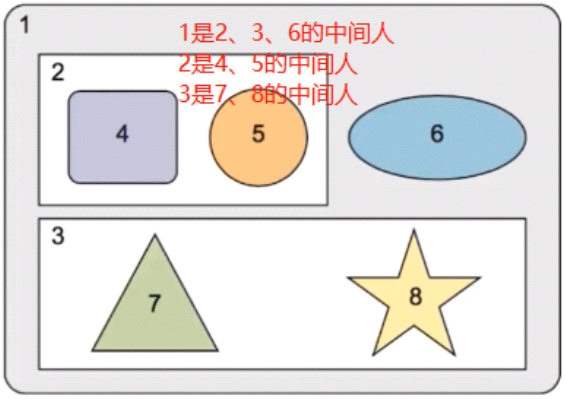
app.html

<app-price-quote(priceChange)="priceQuoteHandler($event)"></app-price-quote>

我们学习了如何使用输出属性来向组件外发射事件，并通过事件携带数据，这个事件只能有父组件模板通过事件绑定的方式处理，如果不存在父子关系，使用中间人模式

3. 中间人模式

星期四，十一月 22，2018　3:08 下午



设计一个组件时，不依赖外部存在的组件，要实现这样的组件，要使用中间人模式。**中间人负责从组件接收数据，并将其传递到另一个组件，当另个组件不是父子组件关系时，需要两个共同的父组件，这个父组件就是中间人模式，中间人模式同时使用了输入属性和输出属性。**

案例：假设交易员，价格达到一个值，购买，报价组件添加一个购买按钮，报价组件并不知道如何下单买股票，报价组件通知父组件，交易员在某个价位购买股票，中间人知道哪个组件能完成下单，将股票代码和价格传给组件。

1. 报价组件html

```
<div>这是报价组件</div>
<div>
  购票代码是{{stockCode}}
  <p></p>
  股票价格是{{price |number: "2.1-3" }}
</div>
<div>
  <input type="button" value="立即购买" (click)="buyStock($event)">
</div>
```

2. 报价组件（把价格发出去，不管谁接收）

```
// 信息输出出去，谁感兴趣来订阅它，EventEmitter需要一个泛型。
// lastPrice,可以发射事件，也可以订阅事件，发射事件，就必须用@output装饰器来装饰
@Output()
  // <PriceQuote>就是一个泛型，泛型就是你要往外发的数据是什么类型的，
lastPrice: EventEmitter<PriceQuote> = new EventEmitter();

 // 点击按钮时，我们需要知道股票的报价，
 // 故需要把点击时的股票价格发出去，这样就需要一个eventEmitter来发射
 buyStock(event){
   // 这样就把购买时的股票价格发出去了
   // 然后在中间人模式中，他的父组件(app.ts)来接收
   this.buy.emit(new PriceQuote(this.stockCode, this.price))
 }
```

3. 父组件的控制器（父组件接收）

```
// 发射的事件是这个类型的，就声明这个类型属性，接收发射的报价信息
priceQuote: PriceQuote = new PriceQuote( stockCode: "",  lastPrice: 0);

  buyHandle(event: PriceQuote) {
    // 把本地的变量赋值为捕获的事件
    this.priceQuote = event;
  }
```

4. 父组件的html（传给下单组件）

```
<app-price-quote (buy)="buyHandle($event)"></app-price-quote>
<!--拿到报价组件传出的购买事件之后，需要把股票信息传递给下单组件，这里就是和中间人做的事情了-->
<app-order [priceQuote]="priceQuote"></app-order>
```

5. 订单组件控制器（接收priceQuote）

```
  //这边接收中间人传过来的股票信息
  @Input()
  priceQuote: PriceQuote;
```

6. 订单组件html

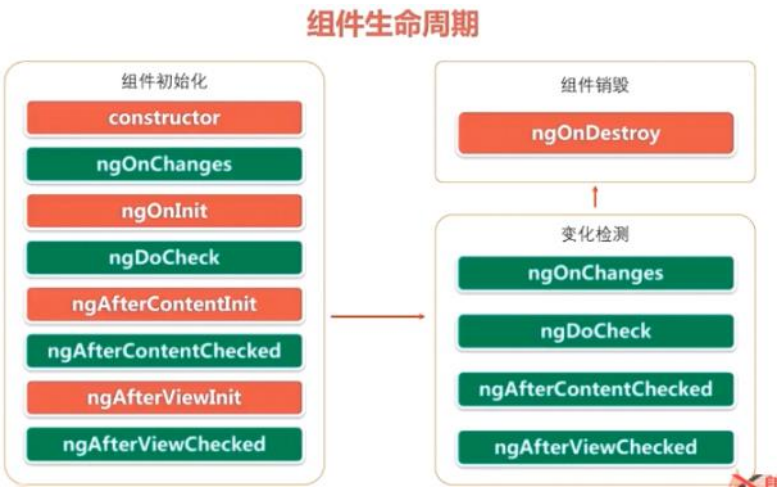
```
<div>
  买100手{{priceQuote.stockCode}}股票
  <p></p>
  买入价格是{{priceQuote.lastPrice}}
</div>
```

如果两个组件没有共同父组件，应该注入一个服务作为中间人，无论何时，组件被创建，组件可订阅服务发生事件流。

在Angular构建之前，需要先明确那些作为中间人那些为父组件，各组件之间的通信是如何来做的，在开始写代码，一定要深入思考应该如何设计，要有大局观

4. 组件生命周期钩子概述

星期四，十一月 22，2018　　4:07 下午



红色的方法只调用一次，绿色的方法可以调用多次，一共9个钩子，变化检测的那是个在组件初始化的时候已经构建好了

```
export class LiftComponent implements OnInit {
  // 每一个钩子都是ng core库里定义的接口，每一个接口都有一个唯一的钩子方法
  // 名字是由接口名字加上ng构成的。OnInit接口方法就是ngOnInit
  // 检查组件类，一旦发现钩子方法被定义，就调用它，会找到ngOnInit，有无接口无所谓
  // 建议在组件中添加接口，利于ide的支持，强类型的检查。

  constructor() { }

  ngOnInit() {
  }
```

例子演示所有钩子的调用顺序

1. 子组件

```
// 将所有的钩子都引入进来，然后一个个看
// 这个例子只是来测试钩子的调用顺序的后面会详细说各个钩子的用法
export class LifeComponent implements OnInit, OnChanges, DoCheck, AfterContentChecked,
  AfterContentInit, AfterViewChecked, AfterViewInit, OnDestroy{

  // 声明一个输入属性目的是为了演示构造函数和NgOnInit的区别
  @Input()
  name: string;

  logIt(msg: string) {
    // 打印当前编号和msg
    // #代表的是标记 ${}显示值 ++后自增
    // 前后的点不是引号而是~对应的那个点·
    console.log(`#${logIndex++} ${msg}`);
  }

  constructor() {
    this.logIt( msg: "name属性在constructor里的值是: " + name)
  }

  // ngOnChanges的changes是一个SimpleChanges对象
  // 这里面包含了所有输入属性变化前的值和变化后的值
  ngOnChanges(changes: SimpleChanges): void {
    //  需要从changes里面把name属性变化以后的值取出来
    let name = changes['name'].currentValue;
    this.logIt( msg: "name属性在constructor里的值是: " + name)
  }
  /* 除了上面那两个构造方式不同之外，剩下的构造方式都一样，只要把当前方法名打出来就可以了*/
```

```
ngOnInit() {
  this.logIt( msg: "ngOnInit")
}

ngAfterContentChecked(): void {
  this.logIt( msg: "ngAfterContentChecked")
}

ngAfterContentInit(): void {
  this.logIt( msg: "ngAfterContentInit")

}

ngAfterViewChecked(): void {
  this.logIt( msg: "ngAfterViewChecked")

}

ngAfterViewInit(): void {
  this.logIt( msg: "ngAfterViewInit")
}

ngDoCheck(): void {
  this.logIt( msg: "ngDoCheck")
}

ngOnDestroy(): void {
}
```

2. 父组件app.html

```
<app-life [name]="title"></app-life>
```

3. 父组件app.ts

```
title = 'Tom';
```

效果：

```
#1 name属性在constructor里的值是：
#2 name属性在ngOnChanges里的值是：tom
#3 ngOnInit
#4 ngDoCheck
#5 ngAfterContentInit
#6 ngAfterContentChecked
#7 ngAfterViewInit
#8 ngAfterViewChecked
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
#9 ngDoCheck
#10 ngAfterContentChecked
#11 ngAfterViewChecked
```

结论：构造函数一定存在，其他方法根据实际实现。如果没实现这些钩子，发生这些事，组件就跳过去了。

ngOnChanges：在ngOnInit之前，当数据绑定输入属性的值发生变化时。

ngOnInit：在第一次ngOnChanges之后。用来初始化 （输入属性name在构造函数是空的，在第一次调用ngOnChanges被初始化，如果组件初始化依赖输入属性值，初始化逻辑写在ngOnInit中）

什么是生命周期钩子

简单点来说生命周期钩子就是Angular中一个组件从被创建当销毁期间的一些有意义的关键时刻.这些关键时刻在Angular中被Angular核心模块 @angular/core 暴露出来,赋予了我们在它们发生时采取行动的能力.

指令和组件的实例有一个生命周期：新建、更新和销毁。

每个接口都有唯一的一个钩子方法，它们的名字是由接口名加上 ng前缀构成的。比如，OnInit接口的钩子方法叫做ngOnInit。

生命周期顺序简写

在Angular通过构造函数创建组件/指令时，它调用这些生命周期钩子方法的顺序是：

ngOnChanges：在ngOnInit之前，当数据绑定输入属性的值发生变化时。

ngOnInit：在第一次ngOnChanges之后。用来初始化 （输入属性name在构造函数是空的，在第一次调用ngOnChanges被初始化，如果组件初始化依赖输入属性值，初始化逻辑写在ngOnInit中）

ngDoCheck：每次Angular变化检测时。

ngAfterContentInit：在外部内容放到组件内之后。

ngAfterContentChecked：在放到组件内的外部内容每次检查之后。

ngAfterViewInit：在初始化组件视图和子视图之后。

ngAfterViewChecked：在妹子组件视图和子视图检查之后。

ngOnDestroy：在Angular销毁组件/指令之前。

除此之外，一些组件还提供了自己的生命周期钩子。例如router有routerOnActivate

5. OnChanges钩子

星期四，十一月 22，2018　　5:03 下午

OnChanges简介：

OnChanges钩子是在父组件初始化或修改子组件的参数时调用，为了理解这个方法为什么会调用，就要理解**可变对象**和**不可变对象**，字符串是不可变的，当字符串创建，值不会改变；即使对象实例属性变化了，字符串也会一直保存在固定的内存里

```
constructor(){
  // greeting 改变是指向的内存地址，
  // 但字符串创建了就不会改变了即使实例对象改变了也会存在内存中
  var greeting = "Hello";
  greeting = "Hello World";

  // user变量本身仍然保持在被创建时的内存地址，
  // 改的是name的属性的内存地址指向
  var user = {name: "Tom"};
  user.name = "Jerry"
}
```

分析：

1. 第二次greeting赋值改变的是内存地址

2. user变量本身仍然保持在被创建时的内存地址，改的是内存地址对象的内容。内存地址没有变。

一. 演示OnChanges方法

1. ng g component chat

2. chat组件声明三个属性，（两个输入类型，普通的message）

```
export class ChatComponent implements OnInit ,OnChanges, DoCheck{

  @Input()
  greeting: string;

  @Input()
  user:{name: string};

  message: string = "初始化消息";
```

3. chat.html子组件模板

```
<div class="child">
  <h3>我是子组件</h3>
   <div>问候语: {{greeting}}</div>
   <div>姓名: {{user.name}}</div>
   <div>消息:  <input [(ngModel)]="message"></div>
</div>
```

child加一个背景颜色样式

```
.child {
  background: yellow;
}
```

4. chat组件实现OnChanges这个方法

```
export class ChatComponent implements OnInit ,OnChanges, DoCheck{

  @Input()
  greeting: string;

  @Input()
  user:{name: string};

  message: string = "初始化消息";

  constructor() { }

  ngOnInit() {
  }
  // JSON.stringify控制台打印以json格式输出
  ngOnChanges(changes: SimpleChanges): void {
    console.log(JSON.stringify(changes, replacer: null, space: 2));
  }
```

JSON.stringify(changes,null,2)转成json格式,使用两个空格缩进.JSON.stringify()方法用于将 JavaScript 值转换为 JSON 字符串。

5.父组件app需要给子组件传刚才两个输入属性，所以在父组件也得声明刚才两个属性。

app.ts

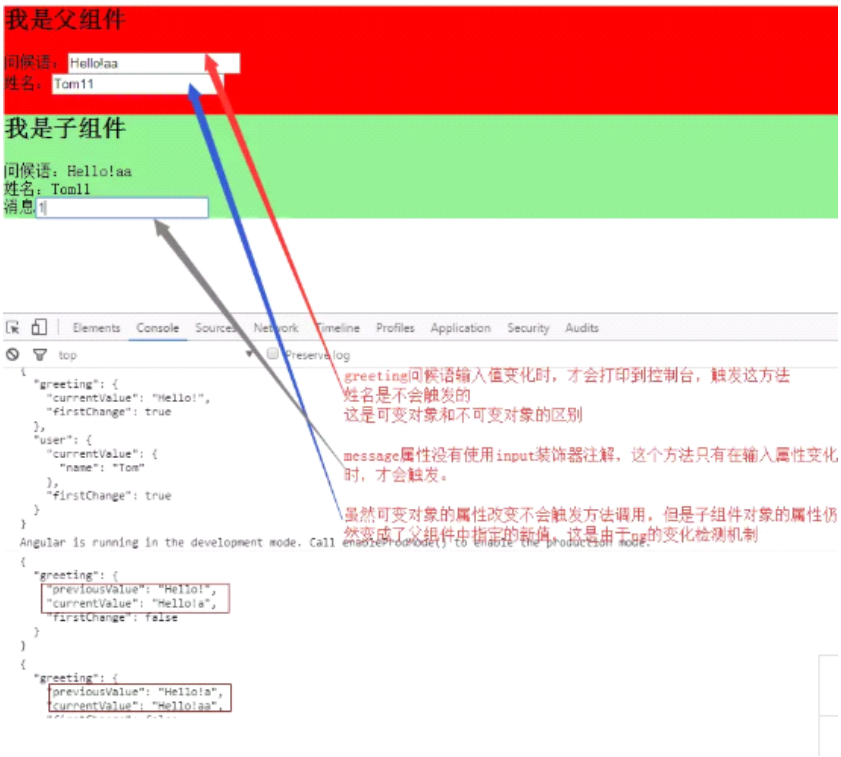
```
greeting:string = "Hello";
user:{name:string} = {name: 'Tom'};
```

app.html

```
<div class="parent">
  <h2>我是父组件</h2>
  <!--父组件随时改变着两个值，这两个值变化时，传进的子组件的值也会变化-->
  <!--传入子组件的值发生变化时，会触发这个ngOnChanges-->
  <div>
    问候语: <input type="text" [(ngModel)]="greeting">
  </div>
  <div>
    姓名: <input type="text" [(ngModel)]="user.name">
  </div>
</div>
<app-chat [greeting]="greeting" [user]="user"></app-chat>
```

效果:

解析:


我是父组件
问候语：Hello!aa
姓名：Tom11
我是子组件
问候语：Hello!aa
姓名：Tom11
消息：
greeting问候语输入值变化时，才会打印到控制台，触发这方法
姓名是不会触发的
这是可变对象和不可变对象的区别
message属性没有使用input装饰器注解，这个方法只有在输入属性变化时，才会触发。
虽然可变对象的属性改变不会触发方法调用，但是子组件对象的属性仍然变成了父组件中指定的新值，这是由于ng的变化检测机制

6.变更检测机制和DoCheck钩子

星期四，十一月 22，2018　　6:40 下午

一.目的：

保证组件属性变化和浏览器的显示同步，浏览器里面发生任何异步变化都会触发“变更检测”，当变更检测运行时，会检测应用中所有的绑定关系。（Anguler中有了zone.js，所以他会根据需要做相应的改变。变更检测只是把属性和页面模板同步，不会改变属性值。每个组件会生成它自己的变更检测器，当检查到变化，zone会根据组件变更检查策略来检查策略，来更新模板）

二.检测策略：

A:Default策略：检查所有组件

B.Onpush策略：阻止检查继续走下去

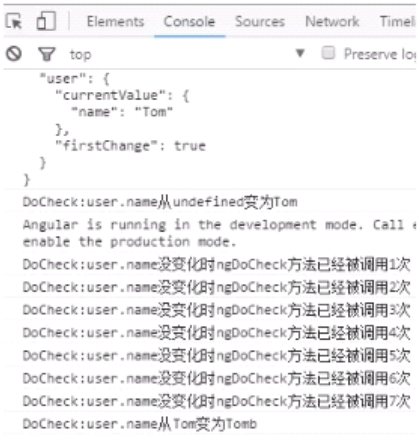
三.DoCheck钩子的调用

演示：

1.子组件

```
  oldUsername: string; // 用来保存user.name改变之前的值
  changeDetected: boolean; // 标记当前的username属性是否发生变化
  changeCount: number; // 标记变更监测机智被调用的次数
  //触发变更监测机制就调用这个钩子
  ngDoCheck(): void {
    if(this.oldUsername !== this.user.name) {
      this.changeDetected = true;
      console.log("DoCheck user.name从" + this.oldUsername + "变为" + this.user.name);
      this.oldUsername = this.user.name
    }
    //有变化
    if(this.changeDetected){
      this.changeCount = 0; // 计数器清空
    }else{
      this.changeCount++;
      console.log("DoCheck user.name 没有变化时ngDoCheck被调用了" + this.changeCount + "次");
    }
    this.changeDetected = false; // 无论变化没变化，最后都更改标志位
  }
}
```

效果：



解析：

每次点点击事件都会触发变更检测机制，DoCheck就会被调用，如果改了tom值，这个变化就会被监测机制捕捉到捕捉到，计数器就清空了。

钩子会检测什么时候发生变化，注意钩子的频繁调用，只要发生变化，都会被调用，大部分调用都没有太大作用，很显然用这个钩子要非常高效，要不然会引起性能问题，有check关键字的钩子，变更检测发生，所有写check关键字钩子都会被调用。所有写有check关键字的钩子用的时候要非常高效(即不要在里面写处理时间过长的代码否则会卡顿)。

7.view钩子少图

星期四，十一月 22，2018　　6:41 下午

知识点1.父组件调用子组件的方法

（1）子组件的控制器

```
greeting(name: string){
  console.log("hello " + name)
}
```

（2）父组件调用（A,B两种方法，选一就可以）

A.父组件控制器调用

```
<!--方法一：结合控制器调用-->
<app-child #child1></app-child>
  // 通过装饰器可以在父组件里引用一个子组件，
  // 从而在父组件的方法里调用子组件的方法
  @ViewChild('child1')
  //  通过child1找到调用的组件，并赋值给child1
  child1: ChildComponent;

  ngOnInit(): void {
    // 调用子组件的方法
    setTimeout( callback: () =>{
      this.child1.greeting( name: "Tom");
    },  ms: 5000);
  }
```

B.父组件模板

```
<!--方法二：纯模板调用-->
<app-child #child2></app-child>
<button (click)="child2.greeting('Helen')">调用child2的greeting方法</button>
```

知识点2.AfterViewInit，AfterViewChecked

（1）在父组件上实现 AfterViewInit，AfterViewChecked接口，这两个钩子是在组件模板所有内容组装完成以后，组件模板已经给用户看了以后，这两个钩子才被调用

```
  ngAfterViewChecked(): void {
    console.log("父组件的视图变更检测完毕")
  }

  ngAfterViewInit(): void {
    console.log("父组件的视图初始化完毕");
  }
```

（2）同样，在子组件中也实现这两个钩子

```
export class ChildComponent implements OnInit, AfterViewChecked, AfterViewInit{

  constructor() { }
  ngAfterViewChecked(): void {
    console.log("子组件的视图变更检测完毕")
  }

  ngAfterViewInit(): void {
    console.log("子组件的视图初始化完毕")
  }
```

（3）父组件中调用子组件方法

```
@ViewChild('child1')
child1: ChildComponent;
ngOnInit(): void {
  // 调用子组件的方法
  setTimeout( callback: () =>{
    this.child1.greeting( name: "Tom");
  }, ms: 5000);
}
```

效果：

```
子组件的视图初始化完毕
子组件的视图变更检测完毕
子组件的视图初始化完毕
子组件的视图变更检测完毕
父组件的视图初始化完毕
父组件的视图变更检测完毕
Angular is running in the develo
to enable the production mode.
子组件的视图变更检测完毕
子组件的视图变更检测完毕
父组件的视图变更检测完毕
hello Tom
子组件的视图变更检测完毕
子组件的视图变更检测完毕
父组件的视图变更检测完毕
hello Tom
```

解析：当两个子组件初始化完毕，父组件初始化完毕才会调用

1. 如果父组件的视图想要完全装好，首先子组件的视图要先装好。

2. 视图初始化完毕，AfterViewInit调用一次，在也不会调用了。初始化方法只会调用一次，每隔五秒调用子组件方法时，可能会触发变更检测机制，会把所有带check关键字的方法都实现一变。

Note：

1. AfterViewInit，AfterViewChecked这两个组件都是在所有视图组装完成以后才开始调用，且AfterViewInit会在AfterViewChecked前面调用
2. 如果组件有子组件，只有所有子组件的视图组装完毕以后，父组件的这两个方法才会被调用
3. 不要在这两个方法里去改变视图中绑定的东西，如果非要改变某个值，要写在一个setTimeout里面，否则会抛出异常。变更检测禁止在视图组装好在更新这个视图的。如下：

```
<!--测试视图组合好后改变视图的某个属性会产生什么影响-->
<p>{{massage}}</p>
```

```
// 测试父模板组装完成后改变视图绑定的东西会有什么影响
massage: string;
ngAfterViewInit(): void {
  console.log("父组件的视图初始化完毕");
  // this.massage = "hello"; // 会报异常
  // 如果要传值，需要用setTimeout方法
  setTimeout( callback: () => {
    this.massage = 'Hello';
  }, ms: 300)
}
```

8. ngContent

星期四，十一月 22，2018　　7:32 下午

知识点一. 投影：

在某些情况下，需要动态改变模板的内容，可以用路由，但路由是一个相对比较麻烦的东西，而我要实现的功能没有那么复杂，没有什么业务逻辑，也不需要重用。这个时候可以用投影。可以用ngContent将父组件中任意片段投影到子组件中

1. 父组件模板

```
<div class="wrapper">
  <h2>我是父组件</h2>
  <div>这个是div定义在父组件中</div>
  <app-child2>
    <div class="header">这是页头，父组件投影到子组件的,title是{{title}}</div>
    <!--这个会投影到子组件的ng-content中。父组件模板中插值表达式只能绑定父组件的属性-->
    <div class="footer">这是页脚，父组件投影到子组件的</div>
  </app-child2>
</div>
<!--Angular还可以用属性绑定的形式很方便的插入一段HTML。-->
<div [innerHTML]="divContent"></div>
```

2. 子组件模板

```
<div class="wrapper">
  <h2>我是子组件</h2>
  <div>这个div定义在子组件中</div>
  <!--ng-content Angular提供的投影标签，将父组件投影到子组件的投影点-->
  <!--select=".header" 在固定区域引入父组件的指定标签-->
  <ng-content select=".header"></ng-content>
  <div>内容区域</div>
  <ng-content select=".footer"></ng-content>
</div>
<!--一个组件可以在其模板中声明多个ng-content标签，假设子组件的部分是由三部分组成
页头，页脚和内容区。页头和页脚由父组件投影进来，内容区自己定义-->
```

3. 父组件控制器

```
title = "app component"
divContent = "<div>慕课网</div>";
```

4. 修改父组件及子组件的wrapper颜色

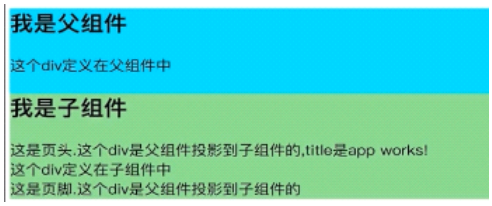
app.css

```
.wrapper {
  background: blueviolet;
}
```

child2.css

```
.wrapper{
  background: cornflowerblue;
}
```

效果图：



解析：

1. innerHTML这种方式只能在浏览器中使用，只能绑定当前组件的内容。ngContent是跨平台的，可以在app应用中使用
2. innerHTML只能插一段html，且只能绑定当前组件的属性，ngContent可以设置多个投影点，但只能绑定父组件中的属性。
3. 动态生成一段HTML，应该优先考虑ngContent这种方式。

9. 最后的钩子

星期五，十一月 23，2018　　9:18 上午

知识点一、ngAfterContentChecked，ngAfterContentInit：

这两个钩子是在被投影的内容组装完调用的。

1. 父组件ts

```
export class AppComponent implements AfterContentInit,
AfterContentChecked, AfterViewInit{

  message:string = 'hello';
  ngAfterContentChecked(): void {
    console.log("父组件投影内容变更检测完毕")
  }
  // ngAfterContentInit与AfterViewInit不同在于，
  // 前者可以在视图初始化之前改变某一属性且不会报错，后者初始化改变属性会报错
  // 前者只是投影初始化完毕
  ngAfterContentInit(): void {
    console.log("父组件投影内容初始化完毕");
    this.message = 'hello world';
  }

  ngAfterViewInit(): void {
    console.log("父组件视图初始化完毕")
  }
}
```

2. 子组件ts

```
export class Child4Component implements OnInit,AfterContentInit,
  AfterContentChecked, OnDestroy{
  ngAfterContentChecked(): void {
    console.log("子组件投影内容变更检测完毕")
  }

  ngAfterContentInit(): void {
    console.log("子组件投影内容初始化完毕")
  }

  constructor() { }

  ngOnInit() {
  }
```

效果图：

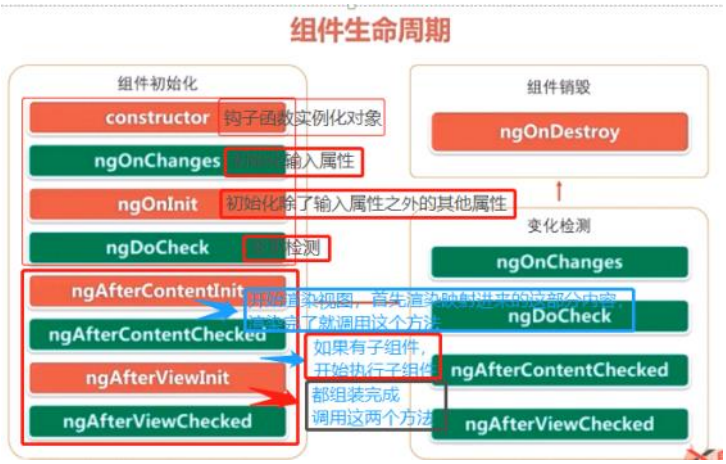
```
父组件投影内容初始化完毕
父组件投影内容变更检测完毕
子组件投影内容初始化完毕
子组件投影内容变更检测完毕
父组件视图内容初始化完毕
Angular is running in the
to enable the production r
父组件投影内容变更检测完毕
子组件投影内容变更检测完毕
```

解析：首先组装的是投影进来的内容，然后是子组件中视图的内容，再是父组件本身的视图内容。

Note：ngAfterContentInit与AfterViewInit不同在于

```
// ngAfterContentInit与AfterViewInit不同在于，
// 前者可以在视图初始化之前改变某一属性且不会报错，后者初始化改变属性会报错
// 前者只是投影初始化完毕
ngAfterContentInit(): void {
  console.log("父组件投影内容初始化完毕");
  this.message = 'hello world';
}
```

可以改变内容，因为只是投影完毕，没有组装完毕



当这个组件初始化过程完毕，组件呈现出来给用户做交互，例如点击、输入等任何一件事，都会触发变更检测机制，一旦触发，当前组件树带有check 关键字的方法都被调用，用来检查当前组件的一些变化，如果变化导致了输入属性改变，那么那个组件的ngOnchanges也会调用，最终组件销毁的时候调用ngOnDestroy。而在路由跳转时，会调用这个组件，将前一个路由组件销毁，并创建后一个路由组件。

知识点二、OnDestroy:

1.定义两个路由：

app-routing

```
const routes: Routes = [
  {path: "", component: Child4Component},
  {path: "child6", component: Child6Component}
];
```

app.html

```
<!--最后一个钩子 ngOnDestroy-->
<a [routerLink]="['/']">child4</a>
<p></p>
<a [routerLink]="['/child6']">child6</a>
<router-outlet></router-outlet>
```

在两个子组件下都创建OnDestory这个函数

child4

```
export class Child4Component implements OnInit,AfterContentInit,
  AfterContentChecked, OnDestroy{
  ngAfterContentChecked(): void {
    console.log("子组件投影内容变更检测完毕")
  }

  ngAfterContentInit(): void {
    console.log("子组件投影内容初始化完毕")
  }

  constructor() { }

  ngOnInit() {
  }

  ngOnDestroy(): void {
    console.log("child4组件被销毁")
  }
}
```

child6:

```
export class Child6Component implements OnInit, OnDestroy {

  ngOnDestroy(): void {
    console.log("child6组件被销毁")
  }
  constructor() { }
  ngOnInit() {
  }
}
```

效果图：

从child4跳转到child6时，child4被销毁

父组件投影内容变更检测完毕
子组件投影内容变更检测完毕
child4组件被销毁
父组件投影内容变更检测完毕

从child6跳转到child4时，child6被销毁

父组件投影内容变更检测完毕
child6组件被销毁
父组件投影内容变更检测完毕
子组件投影内容初始化完毕
子组件投影内容变更检测完毕