布局系统

> RCRE基于的CSS布局系统基础上, 做了一些优化性的改进. 让用户可以更轻松的搭建出适合管理系统的布局结构.

RCRE的布局系统是针对目前MIS系统的业务需求做了一些针对性的改进, 它分为2大块, 栅栏系统和容器内相对布局.

栅栏系统

栅栏系统的核心思想是把整个页面划分成网格状, 然后再将页面元素填充进网格中.

在栅栏系统中, 排版的方式非常类似于表格的排列方式.

行容器

现在, 我们可以先在页面中放入一个行容器.

```
{
    "body": [
        {
            "type": "row",
            "showBorder": true,
            "children": []
        }
    ]
}
```

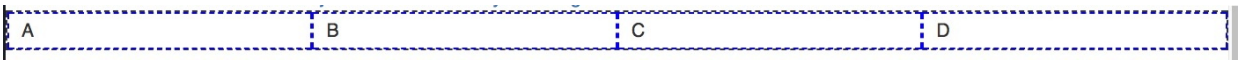
`type`为`row`的一个组件就是一个行容器. 它在页面中, 默认情况下是一个高度为50px的框.

`showBorder`属性是专门设计用于调试布局结构使用的, 设置为`true`会为所有的容器添加一个边框, 这样就能看到每个布局组件的位置.

这样, 这个行容器运行之后就是图示的结果. 接下来, 我们要在行容器内部放入4个不同的文本元素.

```
{
    "body": [
        {
            "type": "row",
            "showBorder": true,
            "children": [
                {
                    "type": "text",
                    "text": "A"
                },
                {
                    "type": "text",
                    "text": "B"
                },
                {
                    "type": "text",
                    "text": "C"
                },
                {
                    "type": "text",
                    "text": "D"
                }
            ]##
        }
    ]
}
```

在默认情况下, `row`组件内的元素会按照成比例的方式进行排列. 如果放入4个元素进去, 那么这4个元素会平分整个`row`组件的宽度. 所以上面的代码运行结果是这样的.



自定义宽度

同样, 如果想要让这4个元素按照不同的宽度比例来排列. 可以使用`gridCount`这个属性.

栅栏布局系统把一行划分为`12`份. 如果`row`组件内部一个组件的`gridCount`设置为6, 则代表这个元素将会占据6份的空间, 也就是会占据1/2的位置.

如果子级元素中, 设置了`gridCount`和没有设置的同时出现的话, 布局系统会优先保证设置的`gridCount`的容器的宽度稳定, 然后剩余的空间留给没有设置的进行瓜分.

下面, 我们再来看个例子:

```
{
    "body": [
        {
            "type": "row",
            "showBorder": true,
            "children": [
                {
                    "type": "text",
                    "text": "A",
                    "gridCount": 5
                },
                {
                    "type": "text",
                    "text": "B",
                    "gridCount": 4
                },
                {
                    "type": "text",
                    "text": "C"
                },
                {
                    "type": "text",
                    "text": "D"
                }
            ]
        }
    ]
}
```

这个例子中, 我们设置`A`的占比是5/12, `B`的宽度占比是4/12. 然后剩下3份空间留给`C`和`D`, 所以`C`和`D`平分这3份空间, 各占1.5/12. 因此最终运行的结果是:

gridCount的竞争

我们再来举一个极端的例子. 如果一个`row`组件内部所有的子级组件的`gridCount`总和大于了12. 那么没有设置`gridCount`的元素会被压缩到能完整展示内容的最小尺寸. 设置了`gridCount`的元素之间会按照各自`gridCount`之间的比例来瓜分剩余的空间. 不过需要提醒的是, 一旦`gridCount`的总和大于的12, 整个一行所有元素的宽度都将会是自适应的,

例如:

```
{
    "body": [
        {
            "type": "row",
            "showBorder": true,
            "children": [
                {
                    "type": "text",
                    "text": "A",
                    "gridCount": 5
                },
                {
                    "type": "text",
                    "text": "B",
                    "gridCount": 10
                },
                {
                    "type": "text",
                    "text": "CCCC"
                },
                {
                    "type": "text",
                    "text": "D"
                }
            ]
        }
    ]
}
```

这个时候, `gridCount`的总和已经大于了12. 所以`C`和`D`会被积压到只能容纳文本的大小. 由于`A`和`B`的`gridCount`的值的比例为1 : 2. 所以剩余的空间就将以1:2的比例进行划分.

运行结果如下:

行容器嵌套

`row`组件一样可以嵌套`row`组件. 作为子级组件的`row`组件, 会以整个当前整个`row`元素的宽度作为基准.

例如:

```
{
    "body": [
        {
            "type": "row",
            "showBorder": true,
            "children": [
                {
                    "type": "text",
                    "text": "A",
                    "gridCount": 5
                },
                {
                    "type": "text",
                    "text": "CCCC"
                },
                {
                    "type": "row",
                    "gridCount": 5,
                    "showBorder": true,
                    "children": [
                        {
                            "type": "text",
                            "text": "helloworld",
                            "gridCount": 5
                        },
                        {
                            "type": "text",
                            "text": "another",
                            "gridCount": 5
                        }
                    ]
                }
            ]
        }
    ]
}
```

这个例子中, 被嵌套的`row`组件只会持有外层`row`组件5/12的宽度, 而内部的2个`text`组件都只会持有占最外层`row`组件 5/12 * 5/12的宽度.

所以, 运行结果如下:

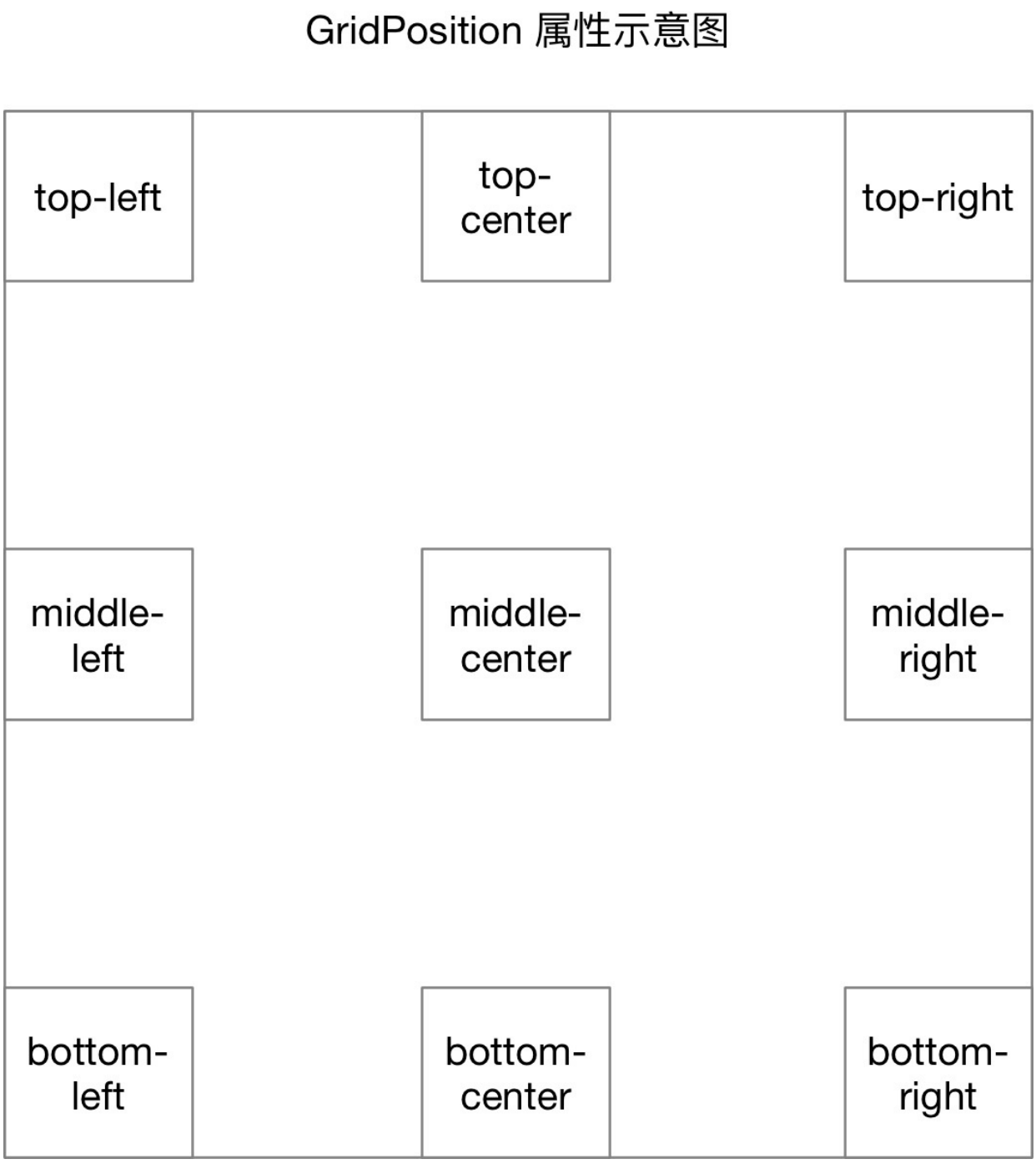
容器内相对布局

当一个元素放置在容器内部, 它相对容器本身也需要一个相对的偏移.

相对布局属性可以通过`gridPosition`属性来进行设置.

相对布局位置

一个容器内部, 元素相对容器的排布一共会有9中不同的情况. 下面的图展示的`gridPosition`的9种不同的值所显示的不同位置.



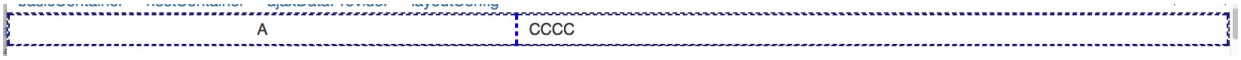
如果没有指定`gridPosition` 的值, 默认值为`middle-left`.

下面我们来看个例子:

```
{
    "body": [
        {
            "type": "row",
            "showBorder": true,
            "children": [
                {
                    "type": "text",
                    "text": "A",
                    "gridCount": 5,
                    "gridPosition": "middle-center"
                },
                {
                    "type": "text",
                    "text": "CCCC"
                }
            ]
        }
    ]
}
```

假如我想让A位于垂直居中的位置, 那么`gridPosition`的值将设置为`middle-center`.

结果如下:



相对偏移

相对容器的9种布局方式虽然能解决大部分问题. 但是总有一些元素需要一些特殊的位置. 这个时候就需要引入相对偏移功能. 相对偏移功能是在元素已经通过`gridPosition`设定好位置之后, 再相对目前的位置进行上下位置偏移.

上下偏移采用`gridTop`属性, 正数将会向下偏移, 负数向上偏移.

左右偏移采用`gridLeft`属性. 正数将会向右偏移, 负数向左偏移.

例如:

```
{
    "body": [
        {
            "type": "row",
            "showBorder": true,
            "children": [
                {
                    "type": "text",
                    "text": "A",
                    "gridCount": 5,
                    "gridPosition": "middle-center",
                    "gridLeft": 10,
                    "gridTop": -20
                },
                {
                    "type": "text",
                    "text": "CCCC"
                }
            ]
        }
    ]
}
```

运行结果如下: