

目录

概述	3
使用	3
示例	3
效果	3
mermaid	4
流程图	4
图表方向	4
节点形状	4
默认结点	4
带文字说明的结点	5
带圆角文字说明的结点	5
带文字说明的圆形结点	5
带文字说明的飘带结点	5
带文字说明的菱形结点	6
结点间的连线	6
带箭头的线	6
没有任何修饰的线	6
在线上有描述文字的线	6
有箭头同时带上文字描述的线	7
点状形式的线	7
加粗的线	7
带文字加粗的线	8
子图表	8
结点样式自定义	8
跟 fontawesome 字体的集成	9
序列图	9
参与者	9
别名	10
连线	10
消息文本	10
活动中	11
标注	12

循环	13
条件选择	14
自定义样式	15
类图	15
语法	17
类结构	17
定义一个类	17
定义类的成员	18
返回类型	18
泛型	19
可见性	19
定义类之间的关系	20
基数和多重性在关系上的表示	22
类的注解	23
注释	24
设置类图显示布局方向	24
状态图	25
状态	26
转换	27
起始状态和结束状态	27
组合状态	28
选择	30
分支／汇流	30
标记	31
并行	32
状态图方向	32
注释	33
实体关系图	33
语法	35
实体和关系	35
关系语法	36
鉴别	36
属性	36
其他	37
用户行程图	38

饼图	38
语法	39
示例	40
甘特图	40
疑问	42
相关	42

小书匠语法说明之mermaid

概述

使用

示例

效果

流程图

图表方向

默认结点

带文字说明的结点

带圆角文字说明的结点

带文字说明的圆形结点

带文字说明的飘带结点

带文字说明的菱形结点

节点形状

带箭头的线

没有任何修饰的线

在线上有描述文字的线

有箭头同时带上文字描述的线

点状形式的线

加粗的线

带文字加粗的线

子图表

结点样式自定义

跟 fontawesome 字体的集成

序列图

参与者

别名

连线

消息文本

活动中

标注

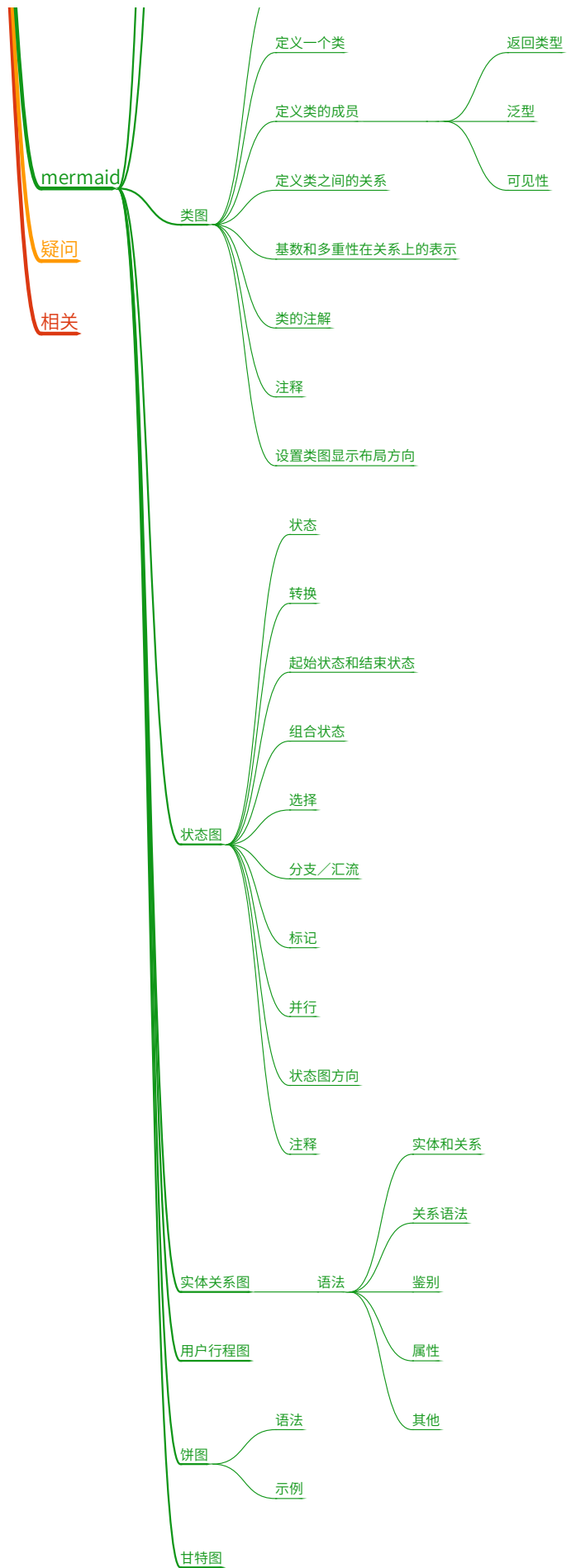
循环

条件选择

自定义样式

语法

类结构



概述

Generation of diagram and flowchart from text in a similar manner as markdown

通过 mermaid 可以实现以纯文本的方式绘制流程图, 序列图, 甘特图等。

小书匠编辑器在 markdown 强大的优势下, 更是无缝集成了 mermaid 的所有功能, 用户不需要任何配置, 只要打开了语法开关, 就可以使用所有 mermaid 图例功能。再加上小书匠的实时预览功能, 可以马上查看到输入的 mermaid 代码生成的效果图。大大提高了用户绘图的速度。

使用

元数据标识: grammar_mermaid

想要使用该语法, 需要在 [设置>扩展语法](#) 里把mermaid选项打开。或者在每篇文章的元数据里通过 `grammar_mermaid` 进行控制。系统默认关闭mermaid语法功能

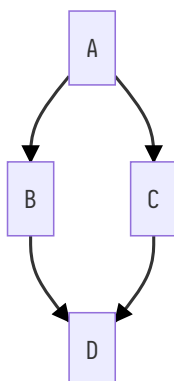
mermaid 提供了流程图, 序列图, 甘特图等多种图形效果

mermaid的语法为代码块语法的执行功能, 语言为 `mermaid`

示例

```
1  ``mermaid!  
2  graph TD;  
3  A-->B;  
4  A-->C;  
5  B-->D;  
6  C-->D;  
7  ```
```

效果



mermaid

流程图

因为 mermaid 支持不同图表, 所以所有的流程图需要以 `graph` 开头

图表方向

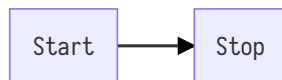
在 `graph` 的后方添加流程图的显示方向

- TB - 从上到下
- BT - 从下到上
- RL - 从右到左
- LR - 从左到右
- TD - 跟 TB 一样, 从上到下

从左到右的图表

4行

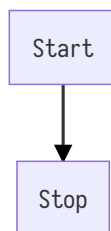
```
1  ```` mermaid!  
2  graph LR  
3      Start --> Stop  
4  ````
```



从上到下的图表

4行

```
1  ```` mermaid!  
2  graph TB  
3      Start --> Stop  
4  ````
```



节点形状

默认结点

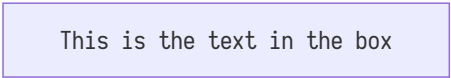
```
1  ```` mermaid!  
2  graph TB  
3      id  
4  ````
```



```
graph LR
    id
```

带文字说明的结点

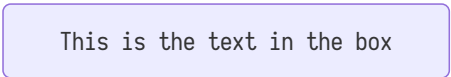
```
1  `` mermaid!
2  graph LR
3      id1[This is the text in the box]
4  ``
```



```
graph LR
    id1[This is the text in the box]
```

带圆角文字说明的结点

```
1  `` mermaid!
2  graph LR
3      id1(This is the text in the box)
4  ``
```



```
graph LR
    id1(This is the text in the box)
```

带文字说明的圆形结点

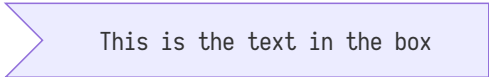
```
1  `` mermaid!
2  graph LR
3      id1((This is the text in the circle))
4  ``
```



```
graph LR
    id1((This is the text in the circle))
```

带文字说明的飘带结点

```
1  `` mermaid!
2  graph LR
3      id1>This is the text in the box]
4  ``
```

```
graph LR; id1[This is the text in the box]
```

带文字说明的菱形结点

```
1  ``` mermaid!
2  graph LR
3      id1{This is the text in the box}
4  ```
```



```
graph LR; id1{This is the text in the box}
```

结点间的连线

结点间可以有不同形式的连接, 比如实线, 虚线, 带箭头的线等

带箭头的线

```
1  ``` mermaid!
2  graph LR
3      A-->B
4  ```
```



没有任何修饰的线

```
1  ``` mermaid!
2  graph LR
3      A --- B
4  ```
```

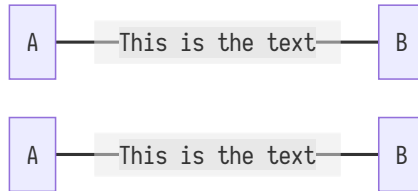


在线上有描述文字的线

```

1  ``` mermaid!
2  graph LR
3      A-- This is the text ---B
4  ```
5  或者
6
7  ``` mermaid!
8  graph LR
9      A---|This is the text|B
10 ```

```



有箭头同时带上文字描述的线

```

1  ``` mermaid!
2  graph LR
3      A-->|text|B
4  ```
5  或者
6  ``` mermaid!
7  graph LR
8      A-- text -->B
9  ```

```



或者

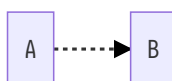


点状形式的线

```

1  ``` mermaid!
2  graph LR;
3      A-.->B;
4  ```

```



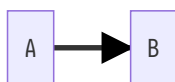
加粗的线

```

1  ``` mermaid!
2  graph LR
3      A ==> B

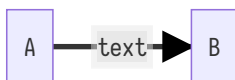
```

4 | ```



带文字加粗的线

```
1  ``` mermaid!
2  graph LR
3    A == text ==> B
4  ```
```

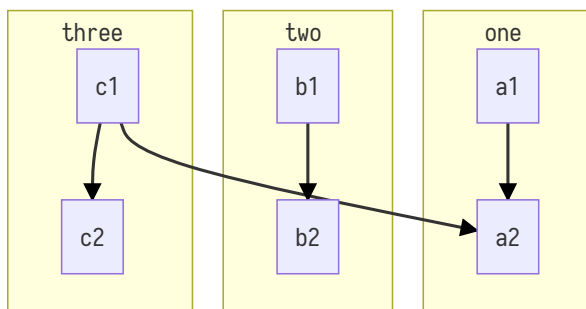


子图表

```
1  subgraph 子图表名称
2    子图表中的描述语句...
3  end
```

示例代码

```
1  ``` mermaid!
2  graph TB
3    c1-->a2
4    subgraph one
5      a1-->a2
6    end
7    subgraph two
8      b1-->b2
9    end
10   subgraph three
11     c1-->c2
12   end
13  ```
```



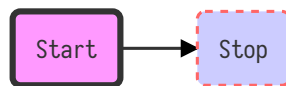
结点样式自定义

```
1  ``` mermaid!
2  graph LR
3    id1(Start)-->id2(Stop)
```

```

4 style id1 fill:#f9f,stroke:#333,stroke-width:4px
5 style id2 fill:#ccf,stroke:#f66,stroke-width:2px,stroke-dasharray: 5, 5
6 ...

```



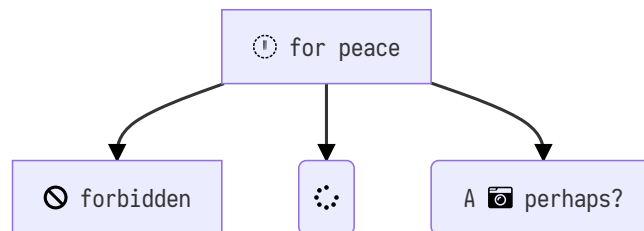
跟 fontawesome 字体的集成

使用 `fa: #图表名称#` 的语法添加 fontawesome

```

1 ``` mermaid!
2 graph TD
3   B["fa:fa-twitter for peace"]
4   B-->C["fa:fa-ban forbidden"]
5   B-->D["fa:fa-spinner"];
6   B-->E("A fa:fa-camera-retro perhaps?");
7 ...

```



序列图

<https://mermaidjs.github.io/sequenceDiagram.html>

因为 mermaid 支持不同图表, 所以所有的序列图需要以 `sequenceDiagram` 开头

```

1 sequenceDiagram
2   [参与者1][连线][参与者2]:消息文本
3 ...

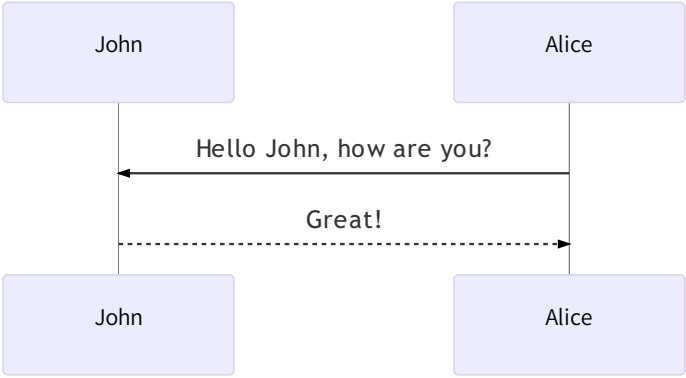
```

参与者

```

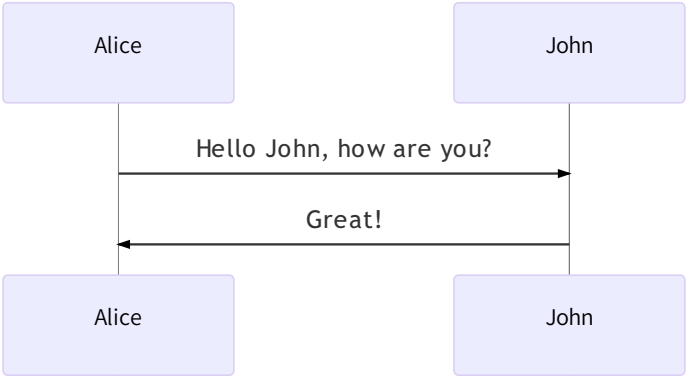
1 ```mermaid!
2 sequenceDiagram
3   participant John
4   participant Alice
5   Alice->>John: Hello John, how are you?
6   John-->>Alice: Great!
7 ...

```



别名

```
1  ``mermaid!
2  sequenceDiagram
3      participant A as Alice
4      participant J as John
5      A->>J: Hello John, how are you?
6      J->>A: Great!
7  ````
```



连线

6种不同类型的连线


Type	Description
->	不带箭头的实线
-->	不带箭头的虚线
->>	带箭头的实线
-->>	带箭头的虚线
-x	末端为叉的实线(表示异步)
--x	末端为叉的虚线(表示异步)


消息文本

消息显示在连线的上方

```
1 | [参与者1][连线][参与者2]:消息文本
```

活动中

在消息线末尾增加  ,则消息接收者进入当前消息的“活动中”状态;

在消息线末尾增加  ,则消息接收者离开当前消息的“活动中”状态。

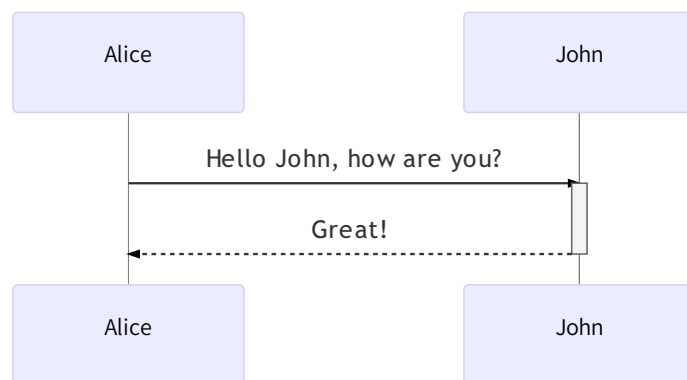
或者使用以下语法直接说明某个参与者进入/离开“活动中”状态:

```
1 | activate 参与者
2 | deactivate 参与者
```

active,deactivate

13行

```
1 | ``` mermaid!
2 | sequenceDiagram
3 |     Alice->>John: Hello John, how are you?
4 |     activate John
5 |     John-->>Alice: Great!
6 |     deactivate John
7 | ```
8 |
9 | ``` mermaid!
10 | sequenceDiagram
11 |     Alice->>+John: Hello John, how are you?
12 |     John-->>-Alice: Great!
13 | ```
```

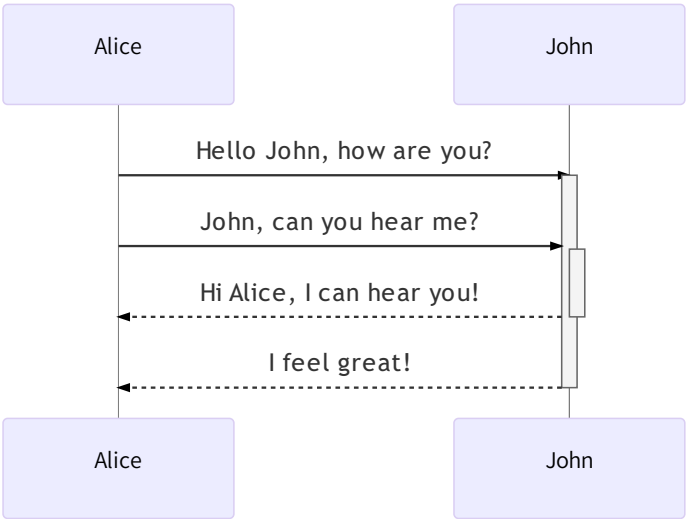


在同一个参与者上面可以叠加多个活动中状态

多个活动叠加

7行

```
1 | ``` mermaid!
2 | sequenceDiagram
3 |     Alice->>+John: Hello John, how are you?
4 |     Alice->>+John: John, can you hear me?
5 |     John-->>-Alice: Hi Alice, I can hear you!
6 |     John-->>-Alice: I feel great!
7 | ```
```



标注

可以对流程图进行标注, 标注位置支持 左侧 , 右侧 和 横跨 三种方式

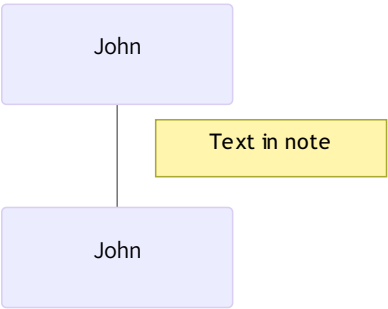
```
1 | Note 位置表述 参与者: 标注文字
```

其中位置表述可以为

表述	含义
right of	右侧
left of	左侧
over	在当中, 可以横跨多个参与者

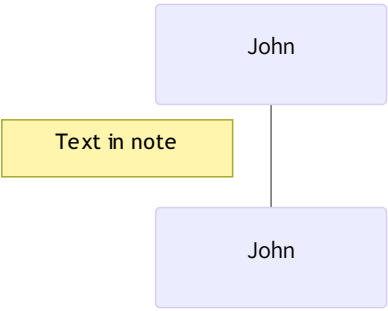
居右标注5行

```
1 | ``` mermaid!  
2 | sequenceDiagram  
3 |   participant John  
4 |   Note right of John: Text in note  
5 | ```
```



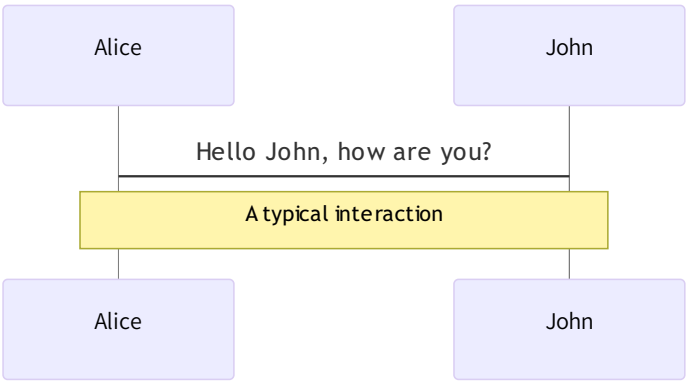
居左标注5行

```
1 | ``` mermaid!  
2 | sequenceDiagram  
3 |   participant John  
4 |   Note left of John: Text in note  
5 | ```
```



横跨标注5行

```
1  `` mermaid!
2  sequenceDiagram
3      Alice->>John: Hello John, how are you?
4      Note over Alice,John: A typical interaction
5  ``
```



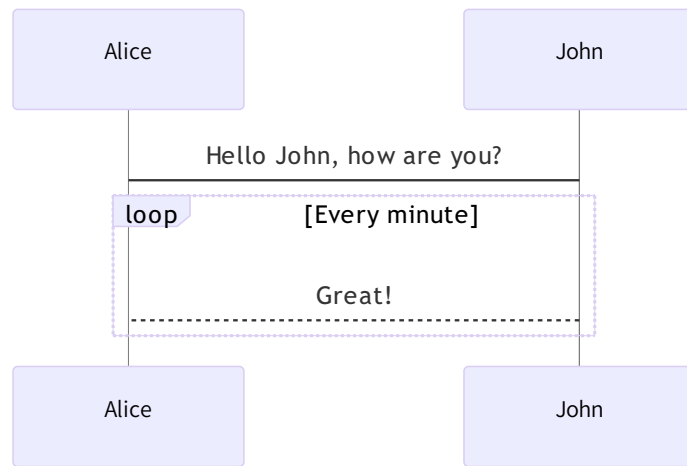
循环

语法如下

```
1  loop 循环的条件
2      循环体描述语句
3  end
```

示例

```
1  `` mermaid!
2  sequenceDiagram
3      Alice->>John: Hello John, how are you?
4      loop Every minute
5          John-->>Alice: Great!
6      end
7  ``
```

条件选择

语法如下

```

1 alt 条件 1 描述
2   分支 1 描述语句
3 else 条件 2 描述 # else 分支可选
4   分支 2 描述语句
5 else ...
6   ...
7 end
  
```

如果遇到可选的情况, 即没有 else 分支的情况, 使用如下语法:

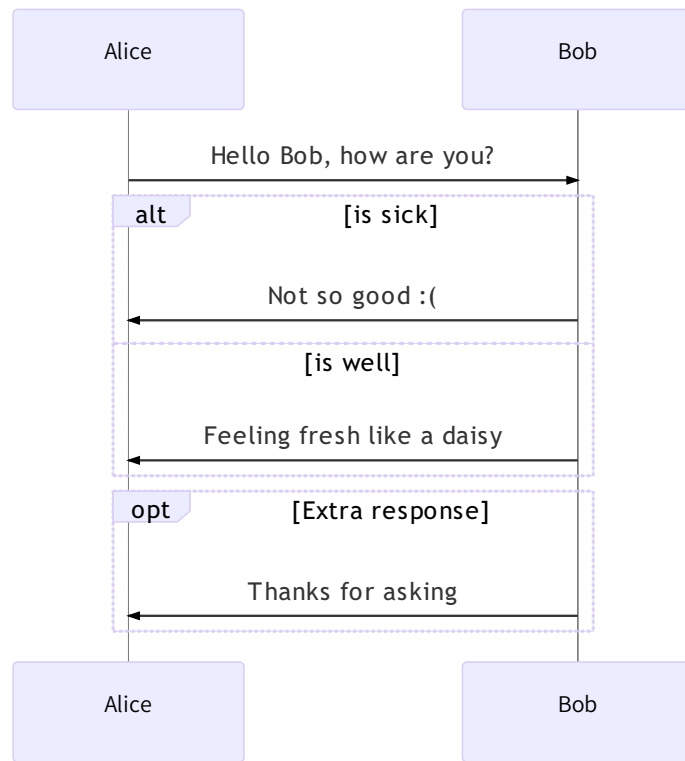
```

1 opt 条件描述
2   分支描述语句
3 end
  
```

示例

```

1 ```` mermaid!
2 sequenceDiagram
3     Alice->>Bob: Hello Bob, how are you?
4     alt is sick
5         Bob->>Alice: Not so good :(
6     else is well
7         Bob->>Alice: Feeling fresh like a daisy
8     end
9     opt Extra response
10        Bob->>Alice: Thanks for asking
11    end
12 ````
  
```



自定义样式

mermaid 在生成序列图时, 会定义很多 class, 配合上小书匠的自定义 css 样式, 用户可以进行更个性化的样式控制

下面列出的是 mermaid 生成的 class

Class	Description
actor	Style for the actor box at the top of the diagram.
text.actor	Styles for text in the actor box at the top of the diagram.
actor-line	The vertical line for an actor.
messageLine0	Styles for the solid message line.
messageLine1	Styles for the dotted message line.
messageText	Defines styles for the text on the message arrows.
labelBox	Defines styles label to left in a loop.
labelText	Styles for the text in label for loops.
loopText	Styles for the text in the loop box.
loopLine	Defines styles for the lines in the loop box.
note	Styles for the note box.
noteText	Styles for the text on in the note boxes.

类图

"In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects."

Wikipedia

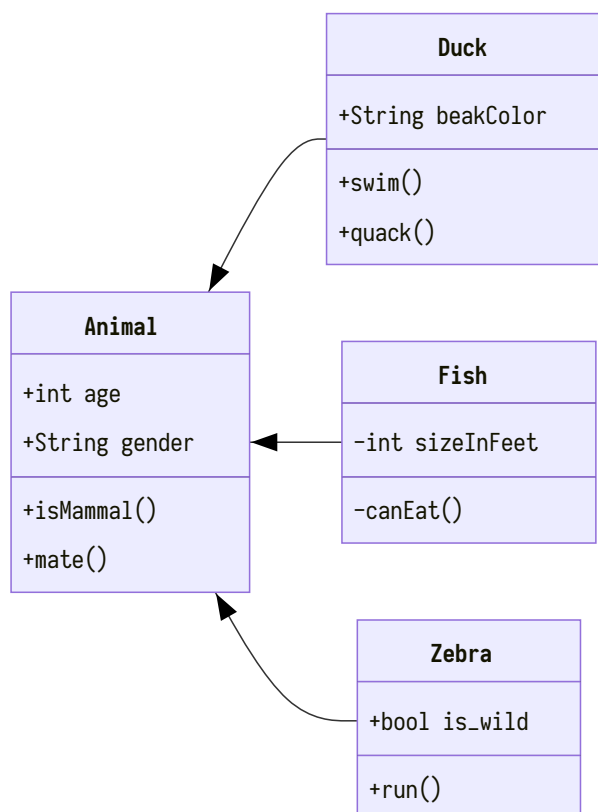
类图用于面向对象对于应用结构概念建模, 也用于把具体的模型翻译成程序代码。类图也可以用于数据建模。

Mermaid 支持类图语法显示

```

1  classDiagram
2      Animal <|-- Duck
3      Animal <|-- Fish
4      Animal <|-- Zebra
5      Animal : +int age
6      Animal : +String gender
7      Animal: +isMammal()
8      Animal: +mate()
9      class Duck{
10         +String beakColor
11         +swim()
12         +quack()
13     }
14     class Fish{
15         -int sizeInFeet
16         -canEat()
17     }
18     class Zebra{
19         +bool is_wild
20         +run()
21     }
22

```



语法

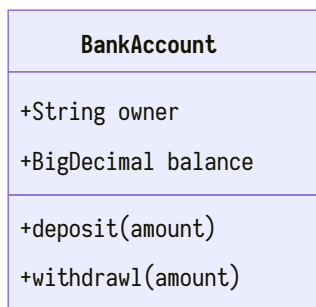
类结构

UML 提供一种机制表示类及成员, 比如属性, 方法和一些其他额外信息。

一个简单的类在图形上主要包含下面三种主要部份:

- 最顶层为类的名称。显示为粗体, 且首字母为大写。同时可以定义一些描述文字。
- 中间层为类的属性。多个属性之间将进行左对齐排列, 属性名称以小写开头。
- 底层部份为类的一些方法函数。多个方法之间将进行左对齐排列, 方法名称也是以小写开头。

```
1 classDiagram
2   class BankAccount
3   BankAccount : +String owner
4   BankAccount : +BigDecimal balance
5   BankAccount : +deposit(amount)
6   BankAccount : +withdrawl(amount)
7
```

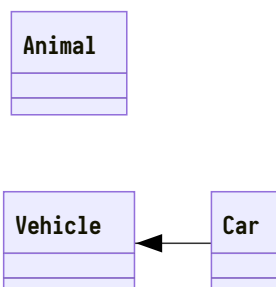


定义一个类

两种方法定义一个类:

- 通过关键字 **class**, 比如 `class Animal`, 显示定义一个类。
- 通过两个类之间的关系 **relationship**, 来隐式定义类。比如 `Vehicle <|-- Car`, 这样就定义了两个类 **Vehicle** 和 **Car**, 以及他们之间的关系。

```
1 classDiagram
2   class Animal
3   Vehicle <|-- Car
```



注: 类的名称只能由字母, 数字和下划线组成。

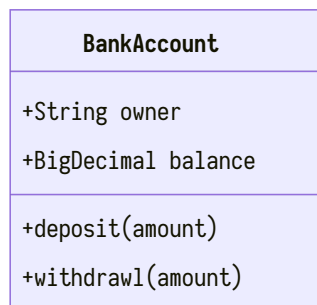
定义类的成员

Mermaid 通过括号 `()` 区分类的属性和方法。带有 `()` 的内容, 将会做为方法进行显示, 其他的则做为类的属性。

Mermaid 提供两种方法定义类的成员

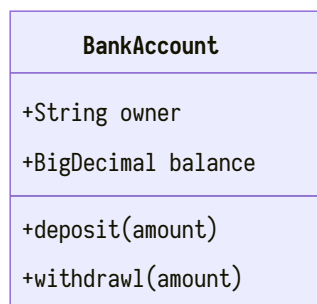
- 类外字义

```
1 class BankAccount
2   BankAccount : +String owner
3   BankAccount : +BigDecimal balance
4   BankAccount : +deposit(amount)
5   BankAccount : +withdrawal(amount)
```



- 类内部定义

```
1 class BankAccount{
2   +String owner
3   +BigDecimal balance
4   +deposit(amount)
5   +withdrawal(amount)
6 }
```



返回类型

通过在类成员的方法最后, 也就是 `)` 添加上一个空格, 然后就可以定义该方法的返回类型。

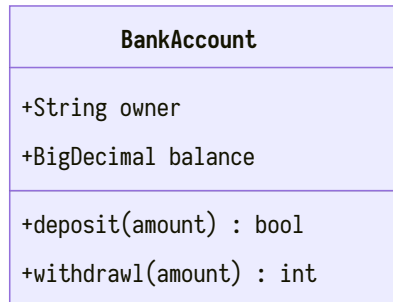
example:

```
1 class BankAccount{
```

```

2   +String owner
3   +BigDecimal balance
4   +deposit(amount) bool
5   +withdrawl(amount) int
6   }

```



泛型

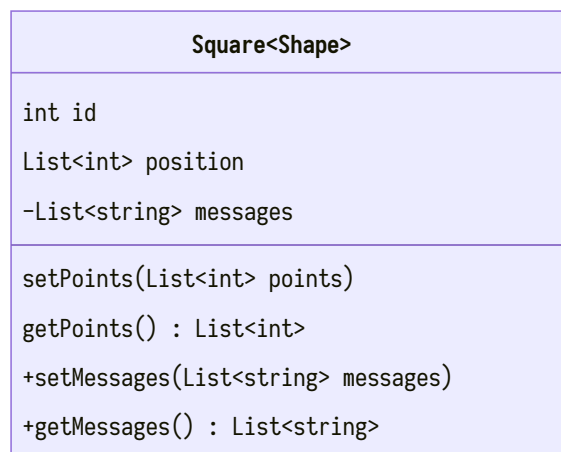
在 Mermaid 里, 对于泛型, 比如 `List<int>` 泛型类型, 只要将 `<>` 用波浪线 `~` 代替, 生成的图形就可以表示为泛型。

注: 目前 Mermaid 还不支持嵌套泛型定义, 比如 `List<List<int>>`

```

1  classDiagram
2  class Square~Shape~{
3      int id
4      List~int~ position
5      setPoints(List~int~ points)
6      getPoints() List~int~
7  }
8
9  Square : -List~string~ messages
10 Square : +setMessages(List~string~ messages)
11 Square : +getMessages() List~string~

```



可见性

类成员的可见性有四种, 在类成员的名称前面添加下面这些符号来进行表示。

- `+` Public
- `-` Private

- `#` Protected
- `~` Package/Internal

类成员方法还可以在其结尾处添加下面两种符号来表示方法的类型, 比如静态和虚函数。

- `*` Abstract e.g.: `someAbstractMethod()*`
- `$` Static e.g.: `someStaticMethod()$`

定义类之间的关系

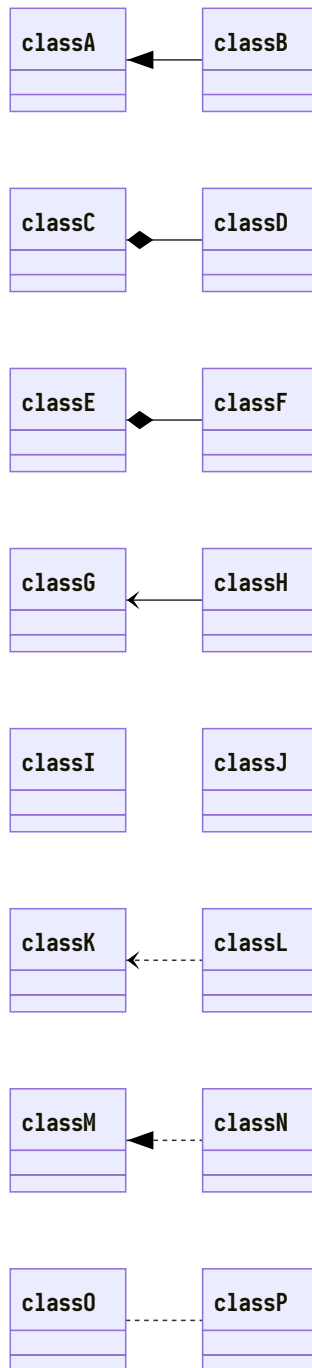
主要语法:

```
1 | [classA][Arrow][ClassB]:LabelText
```

Mermaid 支持的几种类之间关系

语法	描述
<code>< --</code>	继承
<code>*--</code>	组合
<code>o--</code>	聚合
<code>--></code>	关联
<code>--</code>	实线连接
<code>..></code>	依赖
<code>.. ></code>	实现
<code>..</code>	虚线连接

```
1 | classDiagram
2 | classA <|-- classB
3 | classC *-- classD
4 | classE o-- classF
5 | classG <-- classH
6 | classI -- classJ
7 | classK <.. classL
8 | classM <|.. classN
9 | classO .. classP
10 |
```

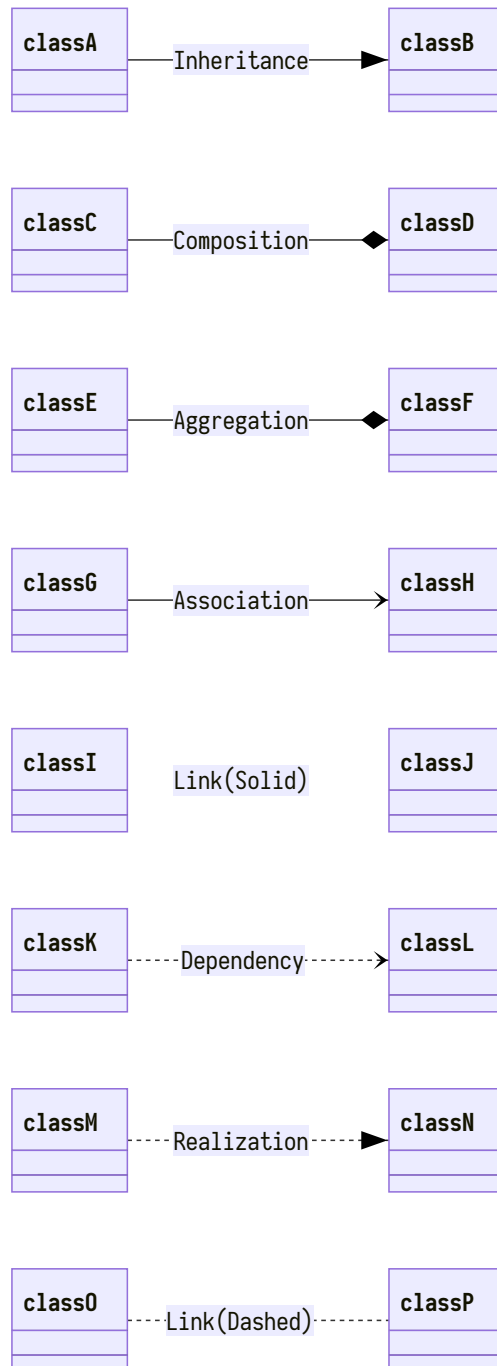


在类关系对就行的结尾, 添加上冒号, 其后的内容就可以做为对这个关系的说明。同时, 如果类之间的关系带有方向性, 只要把语法里的符号反过来, 这个方向也就会进行对换显示了。

```

1 classDiagram
2 classA --> classB : Inheritance
3 classC --* classD : Composition
4 classE --o classF : Aggregation
5 classG --> classH : Association
6 classI -- classJ : Link(Solid)
7 classK ..> classL : Dependency
8 classM ..|> classN : Realization
9 classO .. classP : Link(Dashed)
10

```

基数和多重性在关系上的表示

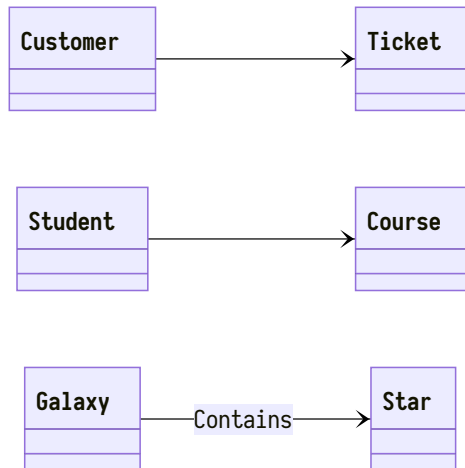
多种基数表示

- `1` Only 1
- `0..1` Zero or One
- `1..*` One or more
- `*` Many
- `n` n {where $n > 1$ }
- `0..n` zero to n {where $n > 1$ }
- `1..n` one to n {where $n > 1$ }

在关系 `arrow` 的前面或者后面可以添加用引用 `"` 包裹的基数内容,用以实现多重性的定义显示效果。实际上,该引号内包裹的内容可以不一定需要上面提到的基数文本,用户自己输入一些描述文字也是可以的。

```
1 [classA] "cardinality1" [Arrow] "cardinality2" [ClassB]:LabelText
```

```
1 classDiagram
2   Customer "1" --> "*" Ticket
3   Student "1" --> "1..*" Course
4   Galaxy --> "many" Star : Contains
```



类的注解

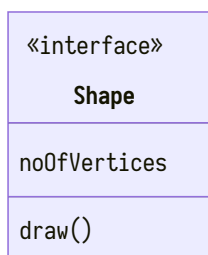
通过特定的文本来注解类的信息

- `<<Interface>>` 表示接口
- `<<abstract>>` 表示抽象类
- `<<Service>>` 服务类
- `<<enumeration>>` 枚举类

将要注解的信息用 `<<` 和 `>>` 包裹实现。系统提供两种方式的类注解

- 单独行注解:

```
1 classDiagram
2   class Shape
3   <<interface>> Shape
```



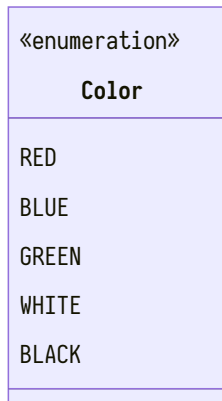
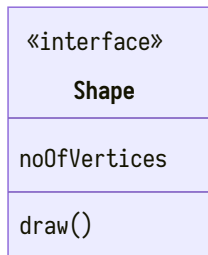
- 在类内部注解

```
1 classDiagram
2   class Shape{
```

```

3      <<interface>>
4      noOfVertices
5      draw()
6  }
7  class Color{
8      <<enumeration>>
9      RED
10     BLUE
11     GREEN
12     WHITE
13     BLACK
14 }
15

```



注释

mermaid注释用%%开头, 系统会忽略注释的文本行内容。

```

1  classDiagram
2  %% This whole line is a comment classDiagram class Shape <<interface>>
3  class Shape{
4      <<interface>>
5      noOfVertices
6      draw()
7  }
8

```

设置类图显示布局方向

类图支持四种布局方向, **LR** , **RL** , **BT** 和 **TB** 。通过指令 **direction** 控制。

```

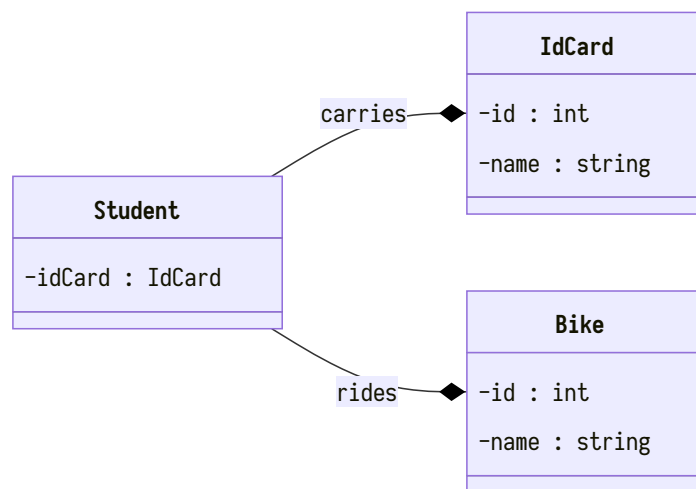
1  classDiagram
2  direction RL
3  class Student {

```

```

4   -idCard : IdCard
5   }
6   class IdCard{
7     -id : int
8     -name : string
9   }
10  class Bike{
11    -id : int
12    -name : string
13  }
14  Student "1" --o "1" IdCard : carries
15  Student "1" --o "1" Bike : rides

```



状态图

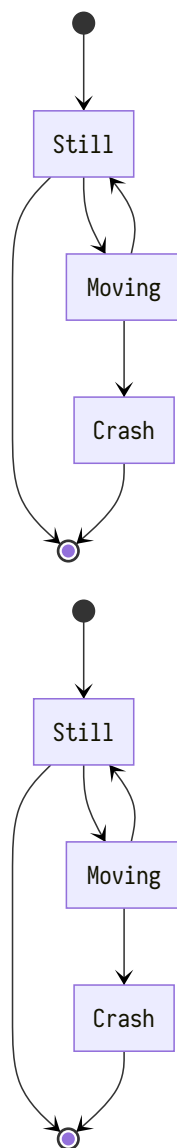
"A state diagram is a type of diagram used in computer science and related fields to describe the behavior of systems. State diagrams require that the system described is composed of a finite number of states; sometimes, this is indeed the case, while at other times this is a reasonable abstraction." Wikipedia

Mermaid 最新版本支持状态图的显示, 语法与 plantUml 类似

```

1  stateDiagram-v2
2    [*] --> Still
3    Still --> [*]
4
5    Still --> Moving
6    Moving --> Still
7    Moving --> Crash
8    Crash --> [*]

```



In state diagrams systems are described in terms of its states and how the systems state can change to another state via a transitions. The example diagram above shows three states **Still**, **Moving** and **Crash**. You start in the state of **Still**. From **Still** you can change the state to **Moving**. In **Moving** you can change the state either back to **Still** or to **Crash**. There is no transition from **Still** to **Crash**.

状态

状态可以用多种方式表示。最简单的方式就是定义状态 id 默认为状态的显示

```

1 | stateDiagram-v2
2 |     s1
  
```

s1

另一种方法使用 `state` 指令定义一个带描述的状态

```

1 | stateDiagram-v2
2 |     state "This is a state description" as s2
  
```

```
graph TD
    s2["This is a state description"]
```

第三种方法就是通过冒号分格开状态 id 和状态描述

```
1 stateDiagram-v2
2     s2 : This is a state description
```

```
graph TD
    s2["This is a state description"]
```

转换

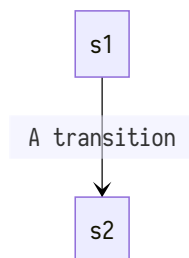
用 `-->` 表示两个状态间的转换, 比如: `s1 --> s2`。代码为:

```
1 stateDiagram-v2
2     s1 --> s2
```



也可以用 `:` 加上描述符组成转换条件, 代码为:

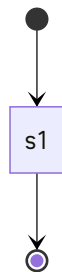
```
1 stateDiagram-v2
2     s1 --> s2: A transition
```



起始状态和结束状态

将 `[*]` 放在转换的开头, 或者结尾, 用来表示起始状态, 或者结束状态。

```
1 stateDiagram-v2
2     [*] --> s1
3     s1 --> [*]
```



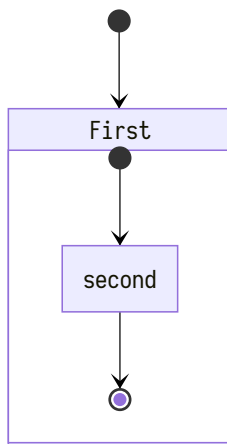
组合状态

组合状态, 也可以说成子状态, 也就是所谓的一个状态里包含多个状态, 他们对于外部的状态, 其实是一个黑盒子, 但在内部里, 却有自己的逻辑关系。

定义一个组合状态, 通过 `state` 指令, 指定哪个状态需要变成组合状态, 然后再通过 `{ }` 符号, 将子状态的状态图包裹在内就可以了。

```

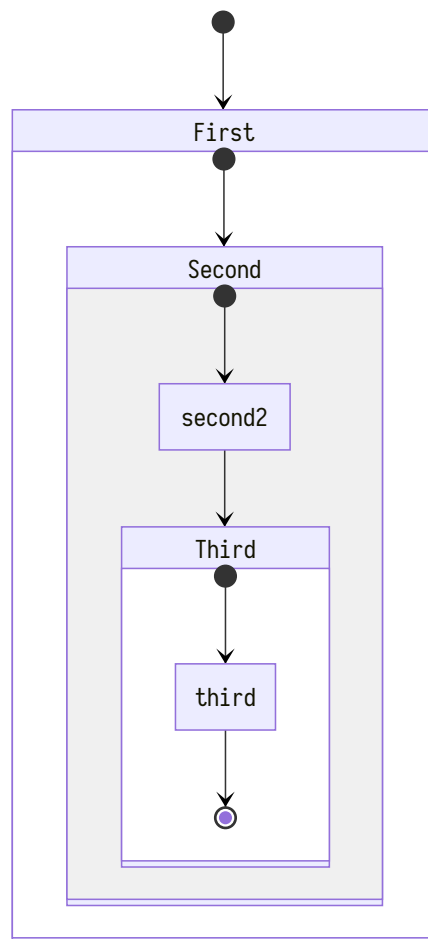
1 stateDiagram-v2
2   [*] --> First
3   state First {
4     [*] --> second
5     second --> [*]
6   }
  
```



Mermaid 支持多层嵌套的组合状态:

```

1 stateDiagram-v2
2   [*] --> First
3
4   state First {
5     [*] --> Second
6
7     state Second {
8       [*] --> second
9       second --> Third
10
11     state Third {
12       [*] --> third
13       third --> [*]
14     }
15   }
16 }
  
```

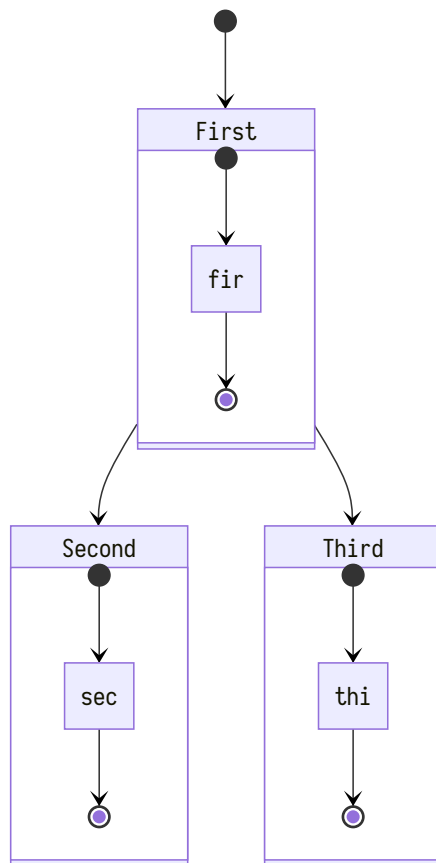


组合状态之间支持状态转换:

```

1 stateDiagram-v2
2   [*] --> First
3   First --> Second
4   First --> Third
5
6   state First {
7       [*] --> fir
8       fir --> [*]
9   }
10  state Second {
11      [*] --> sec
12      sec --> [*]
13  }
14  state Third {
15      [*] --> thi
16      thi --> [*]
17  }

```

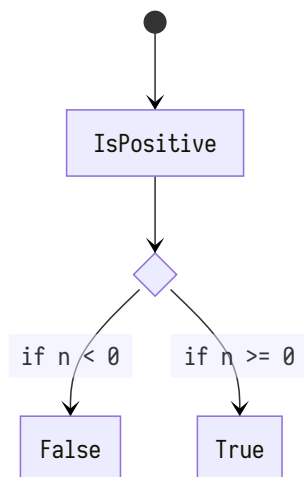
注: 不支持将组合状态内部的状态转换到外部状态或者其他组合状态的内部状态

选择

使用 `<<choice>>` ,让状态进入选择模式。

```

1 stateDiagram-v2
2   state if_state <<choice>>
3   [*] --> IsPositive
4   IsPositive --> if_state
5   if_state --> False: if n < 0
6   if_state --> True : if n >= 0
  
```



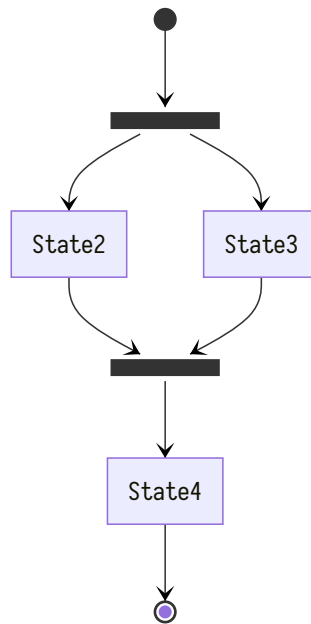
分支／汇流

使用 `<<fork>>` , `<<join>>` ,可以让状态进行分支或者汇流。

```

1 stateDiagram-v2
2 state fork_state <<fork>>
3 [*] --> fork_state
4 fork_state --> State2
5 fork_state --> State3
6
7 state join_state <<join>>
8 State2 --> join_state
9 State3 --> join_state
10 join_state --> State4
11 State4 --> [*]

```



标记

有些状态可以通过名称很容易理解状态的意义,但有些状态无法通过状态名称快速的了解其作用,或者这些状态需要一些额外的说明,这时候,可以通过标记功能实现。

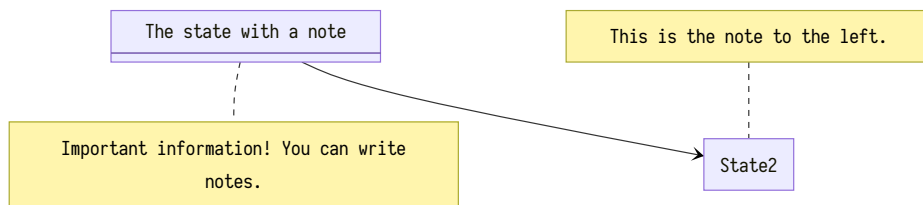
除了指令 `note` 外,还需要使用 `right of` 或者 `left of` 指定标记放置的位置。

多行标记时,需要使用 `end note` 结束标记显示。

```

1 stateDiagram-v2
2 State1: The state with a note
3 note right of State1
4 Important information! You can write
5 notes.
6 end note
7 State1 --> State2
8 note left of State2 : This is the note to the left.

```

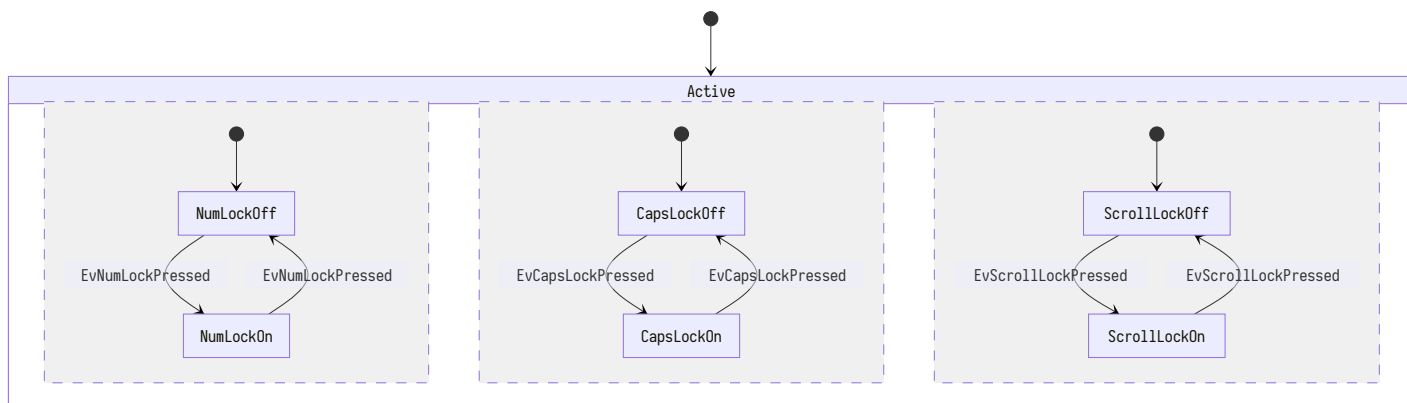


并行

使用符号 `--` 可以用来状多个状态进行并行处理

```

1 stateDiagram-v2
2   [*] --> Active
3
4   state Active {
5     [*] --> NumLockOff
6     NumLockOff --> NumLockOn : EvNumLockPressed
7     NumLockOn --> NumLockOff : EvNumLockPressed
8     --
9     [*] --> CapsLockOff
10    CapsLockOff --> CapsLockOn : EvCapsLockPressed
11    CapsLockOn --> CapsLockOff : EvCapsLockPressed
12    --
13    [*] --> ScrollLockOff
14    ScrollLockOff --> ScrollLockOn : EvScrollLockPressed
15    ScrollLockOn --> ScrollLockOff : EvScrollLockPressed
16  }
  
```



状态图方向

使用 `direction` 指令, 可以改变状态图显示的方向布局, 支持的方向有 `LR`, `RL`, `TB` 和 `BT`

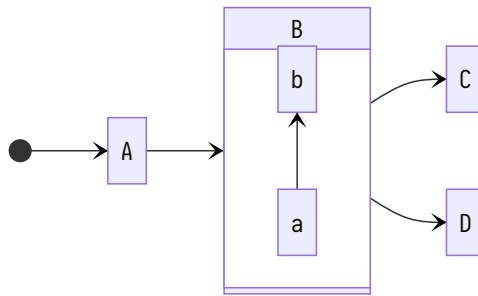
```

1 stateDiagram
2   direction LR
3   [*] --> A
4   A --> B
5   B --> C
6   state B {
7     direction LR
8     a --> b
  
```

```

9   }
10  B --> D

```



注释

使用 `%%` 指令, 可以将当前行的内容进行注释处理, 注释的内容将不会被显示到结果里。

```

1  stateDiagram-v2
2      [*] --> Still
3      Still --> [*]
4      %% this is a comment
5      Still --> Moving
6      Moving --> Still %% another comment
7      Moving --> Crash
8      Crash --> [*]

```

实体关系图

An entity-relationship model (or ER model) describes interrelated things of interest in a specific domain of knowledge. A basic ER model is composed of entity types (which classify the things of interest) and specifies relationships that can exist between entities (instances of those entity types).

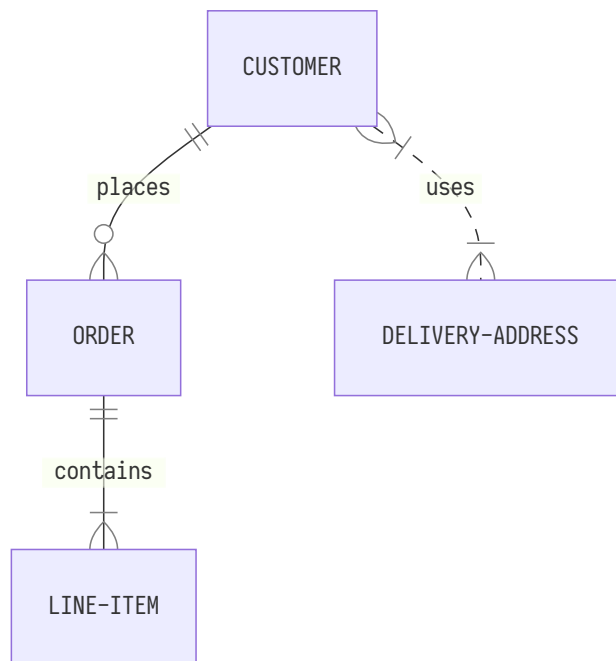
实体关系图里的**实体**, 通常指的是实体类别, 并不是具体的一个实体。比如我们说的**客户**, 就是指具有相似性的所有客户, 而不是指具体的一个客户。实体关系图里的**关系**, 也是对实体这一类别之间的关系描述。

Mermaid 显示实体关系图示例

```

1  erDiagram
2      CUSTOMER ||--o{ ORDER : places
3      ORDER ||--|{ LINE-ITEM : contains
4      CUSTOMER }|..|{ DELIVERY-ADDRESS : uses

```

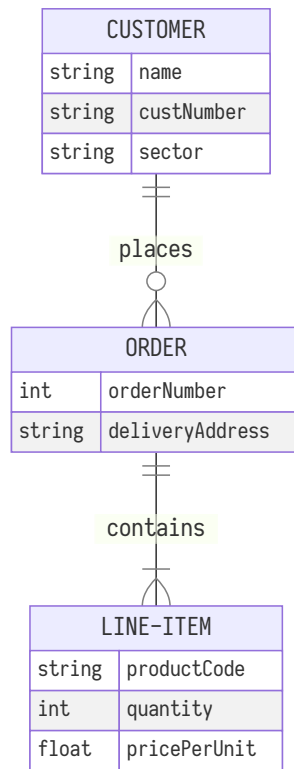


实体名称一般由字母组成

实体关系图可以用在很多范围领域的应用里, 包括从抽象的逻辑模型到具体的数据库关系模型上。

```

1  erDiagram
2    CUSTOMER ||--o{ ORDER : places
3    CUSTOMER {
4      string name
5      string custNumber
6      string sector
7    }
8    ORDER ||--|{ LINE-ITEM : contains
9    ORDER {
10     int orderNumber
11     string deliveryAddress
12   }
13   LINE-ITEM {
14     string productCode
15     int quantity
16     float pricePerUnit
17   }
  
```



语法

实体和关系

Mermaid 使用了和 PlantUML 类似的语法来实现实体关系图的显示。

每一行语句主要由下面几个部分组成：

```
1 | <first-entity> [<relationship> <second-entity> : <relationship-label>]
```

这些字段的含义：

- **first-entity** 第一个实体名称。名称只能包含字母, 数字, 连接符或者下划线。
- **relationship** 实体之间的关系。后面会详细介绍关系的含义。
- **second-entity** 第二个实体名称
- **relationship-label** 从第一个实体的角度描述与第二个实体之间的关系

例如：

```
1 | PROPERTY ||--|{ ROOM : contains
```

该语句可以解释为一个 **PROPERTY** 包含一个或者多个 **ROOM**, 一个 **ROOM** 是一个 **PROPERTY** 的一部份。**contains** 是从 **PROPERTY** 角度出发, 描述 **PROPERTY** 和 **ROOM** 之间为包含关系。

只有 **first-entity** 是必须的, 其他都是可选的。如果只有 **first-entity**, 则表示这个实体与其他实体没有任何关系。如果除了 **first-entity** 部份, 还有其他部份, 则其他部份就必须全部包括在内, 否则就是一个错误的语句格式。

关系语法

relationship 可以分成三个部分

- 第一个实体相对于第二个实体的基数,
- 该关系是否赋予了 "子" 实体身份
- 第二个实体相对于第一个实体的基数

Value (left)	Value (right)	Meaning
o	o	0或1
		1
}o	o{	0或多个(无上限)
}	{	1或多个 (无上限)

鉴别

关系可以分为**identifying鉴别**和**non-identifying非鉴别**两种, 它们分别用实线 `--` 和虚线 `..` 表示。区分它们的方法就是第二个实体是否可以独立存在。更简单的区分方法, 就是类似数据库的多对多, 如果在图上不想显示中间表, 就使用虚线, 想显示中间表, 则使用实现。其他的一般都是实线。

Relationships may be classified as either identifying or non-identifying and these are rendered with either solid or dashed lines respectively. This is relevant when one of the entities in question can not have independent existence without the other. For example a firm that insures people to drive cars might need to store data on **NAMED-DRIVER** s. In modelling this we might start out by observing that a **CAR** can be driven by many **PERSON** instances, and a **PERSON** can drive many **CAR** s - both entities can exist without the other, so this is a non-identifying relationship that we might specify in Mermaid as: **PERSON** }|..|{ **CAR** : "driver" . Note the two dots in the middle of the relationship that will result in a dashed line being drawn between the two entities. But when this many-to-many relationship is resolved into two one-to-many relationships, we observe that a **NAMED-DRIVER** cannot exist without both a **PERSON** and a **CAR** - the relationships become identifying and would be specified using hyphens, which translate to a solid line:

```
1 CAR ||--o{ NAMED-DRIVER : allows
2 PERSON ||--o{ NAMED-DRIVER : is
```

属性

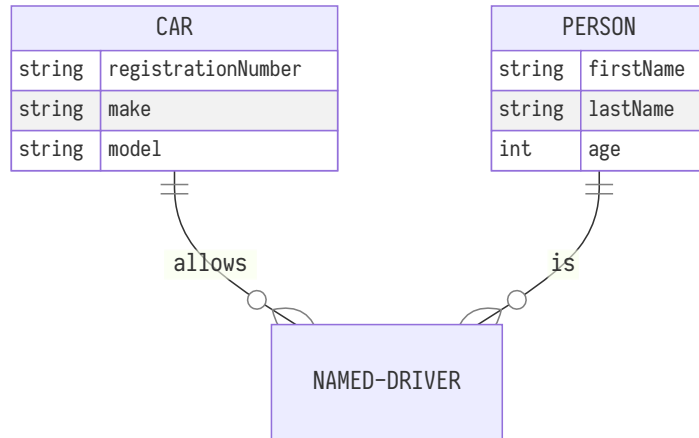
Mermaid 支持对实体的属性定义。在实体后面, 用 `{}` 包裹具体的属性定义即可。

```
1 erDiagram
2   CAR ||--o{ NAMED-DRIVER : allows
3   CAR {
4       string registrationNumber
5       string make
6       string model
```

```

7      }
8      PERSON ||--o{ NAMED-DRIVER : is
9      PERSON {
10         string firstName
11         string lastName
12         int age
13     }

```

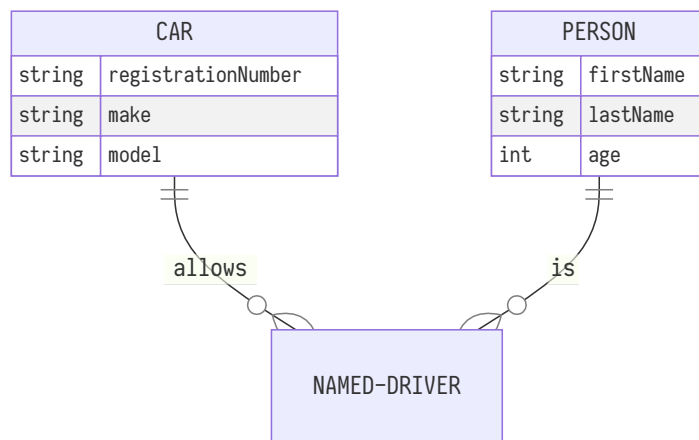


属性将会显示在实体的方框内:

```

1  erDiagram
2      CAR ||--o{ NAMED-DRIVER : allows
3      CAR {
4          string registrationNumber
5          string make
6          string model
7      }
8      PERSON ||--o{ NAMED-DRIVER : is
9      PERSON {
10         string firstName
11         string lastName
12         int age
13     }

```



其他

- 如果想让关系说明进行多词显示, 需要使用双引号包裹这些说明文字

- 如果关系说明为空, 需要使用双引号包裹空字符串

用户行程图

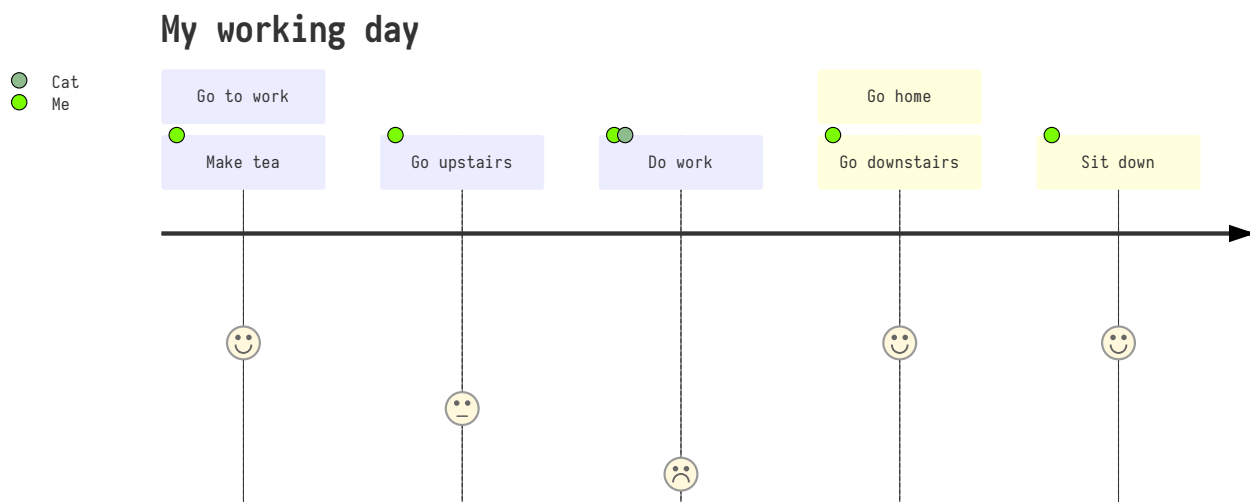
User journeys describe at a high level of detail exactly what steps different users take to complete a specific task within a system, application or website. This technique shows the current (as-is) user workflow, and reveals areas of improvement for the to-be workflow. (Wikipedia)

Mermaid 显示用户行程图示例:

```

1 journey
2   title My working day
3   section Go to work
4     Make tea: 5: Me
5     Go upstairs: 3: Me
6     Do work: 1: Me, Cat
7   section Go home
8     Go downstairs: 5: Me
9     Sit down: 5: Me

```



Each user journey is split into sections, these describe the part of the task the user is trying to complete.

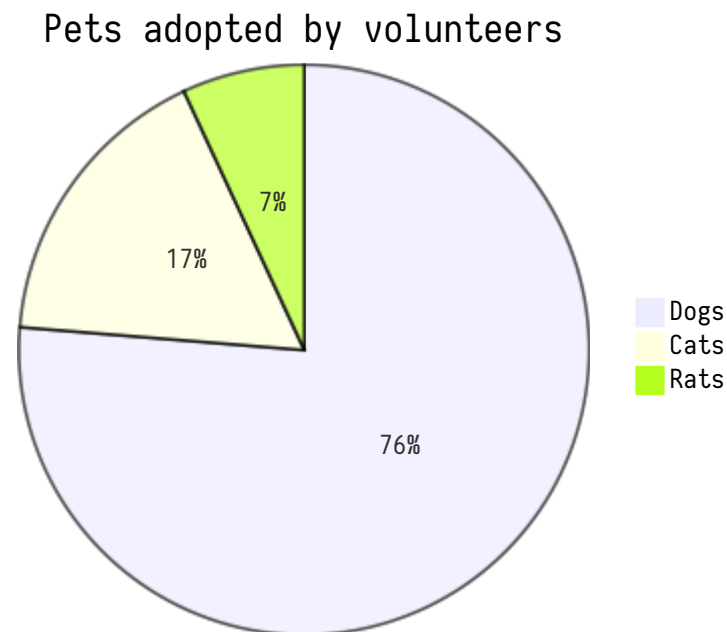
任务语法 **Task name:** <score>: <comma separated list of actors>

饼图

A pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion. In a pie chart, the arc length of each slice (and consequently its central angle and area), is proportional to the quantity it represents. While it is named for its resemblance to a pie which has been sliced, there are variations on the way it can be presented. The earliest known pie chart is generally credited to William Playfair's Statistical Breviary of 1801 -Wikipedia

Mermaid 显示饼图示例.

```
1 pie title Pets adopted by volunteers
2   "Dogs" : 386
3   "Cats" : 85
4   "Rats" : 15
```



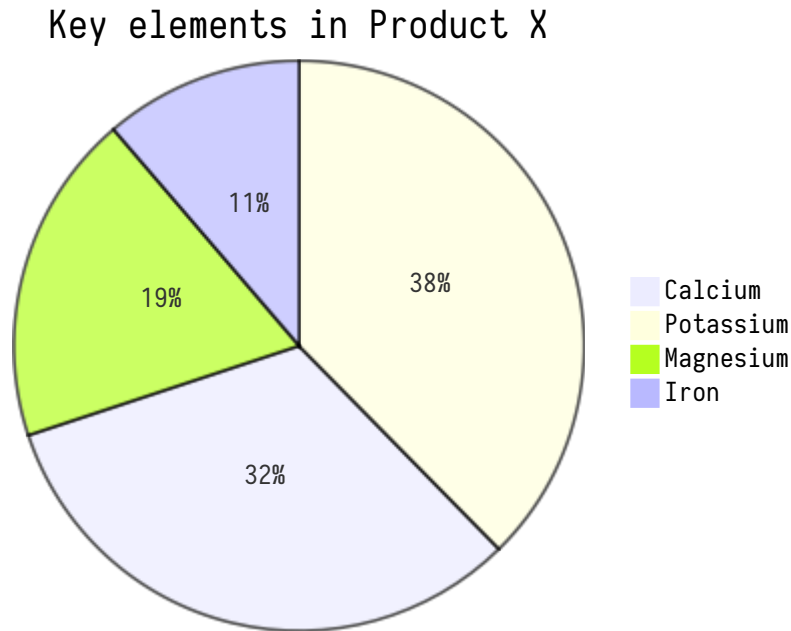
语法

- 以 `pie` 关键字开始
- 然后再 `title` 关键字, 以及具体的标题名称
- 最后输入数据集
 - `label` for a section in the pie diagram within `" "` quotes.
 - Followed by `:` colon as separator
 - Followed by `positive numeric value` (supported upto two decimal places)

```
1 [pie]
2   [title] [titlevalue] (OPTIONAL)
3   "[datakey1]" : [dataValue1]
4   "[datakey2]" : [dataValue2]
5   "[datakey3]" : [dataValue3]
6   .
7
```

示例

```
1 pie
2   title Key elements in Product X
3   "Calcium" : 42.96
4   "Potassium" : 50.05
5   "Magnesium" : 10.01
6   "Iron" : 5
```



甘特图

<https://mermaidjs.github.io/gantt.html>

A Gantt chart is a type of bar chart, first developed by Karol Adamiecki in 1896, and independently by Henry Gantt in the 1910s, that illustrates a project schedule. Gantt charts illustrate the start and finish dates of the terminal elements and summary elements of a project.

甘特图是一类条形图,由Karol Adamiechi在1896年提出,而在1910年Henry Gantt也独立的提出了此种图形表示。通常用在对项目终端元素和总结元素的开始及完成时间进行的描述。

语法

```
1 gantt
2   dateFormat 时间格式
3   title 标题
4
5   section 小节标题
6   任务描述      :状态, 起始节点, 时间段
```

标记	简介
title	标题
dateFormat	日期格式
section	模块
done	已经完成
active	当前正在进行
	后续待处理
crit	关键阶段
日期缺失	默认从上一项完成后

关于日期的格式可以参考：<http://momentjs.com/docs/#/parsing/string-format/>

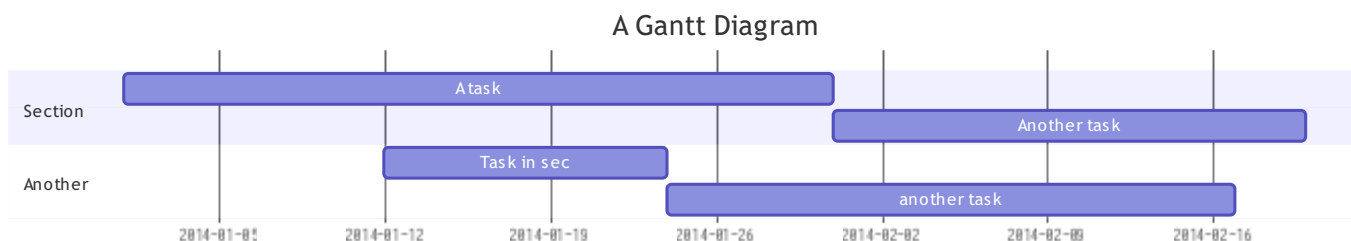
关于 Scale 的格式可以参考：<https://github.com/mbostock/d3/wiki/Time-Formatting>

示例

```

1  ``mermaid!
2  gantt
3      title A Gantt Diagram
4      dateFormat YYYY-MM-DD
5      section Section
6          A task          :a1, 2014-01-01, 30d
7          Another task    :after a1 , 20d
8      section Another
9          Task in sec     :2014-01-12 , 12d
10         another task    : 24d
11  ```

```



一个更加复杂的例子

```

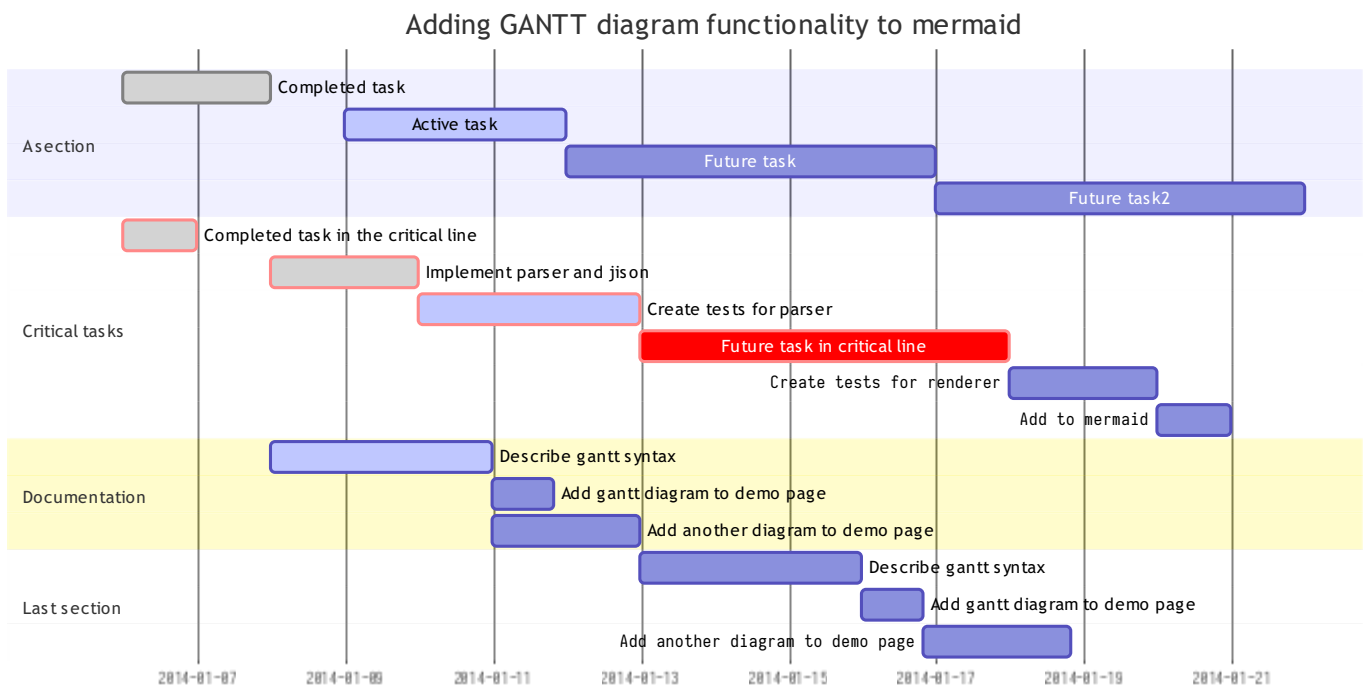
1  ``mermaid!
2  gantt
3      dateFormat YYYY-MM-DD
4      title Adding GANTT diagram functionality to mermaid
5
6      section A section
7          Completed task      :done,    des1, 2014-01-06,2014-01-08
8          Active task         :active,  des2, 2014-01-09, 3d
9          Future task         :         des3, after des2, 5d
10         Future task2        :         des4, after des3, 5d
11
12         section Critical tasks
13         Completed task in the critical line :crit, done, 2014-01-06,24h

```

```

14      Implement parser and jison                :crit, done, after des1, 2d
15      Create tests for parser                  :crit, active, 3d
16      Future task in critical line             :crit, 5d
17      Create tests for renderer                :2d
18      Add to mermaid                           :1d
19
20      section Documentation
21      Describe gantt syntax                    :active, a1, after des1, 3d
22      Add gantt diagram to demo page          :after a1 , 20h
23      Add another diagram to demo page        :doc1, after a1 , 48h
24
25      section Last section
26      Describe gantt syntax                    :after doc1, 3d
27      Add gantt diagram to demo page          :20h
28      Add another diagram to demo page        :48h
29      ...

```



疑问

相关

1. [mermaid 官网](#)