软件建模

用户指南

文档版本 01

发布日期 2025-02-28

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 UML 建模

1.1 UML 概述

UML是Unified Modeling Language缩写，译为统一建模语言，是一种面向对象的可视化建模语言。UML规范定义了两种主要的UML图，分别为结构图和行为图。

结构图

结构图显示了系统及其部件在不同抽象和实现级别上的静态结构以及它们如何相互关联。结构图中的元素表示系统的有意义的概念，并且可以包括抽象的，现实的和实现的概念。包括：类图、对象图、包图、组件图、部署图、组合结构图。

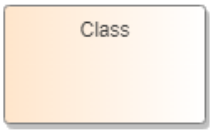
行为图

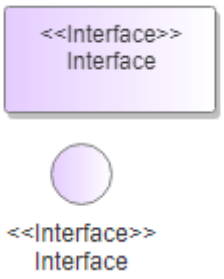
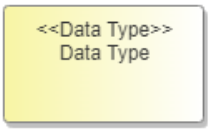
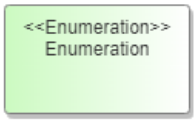
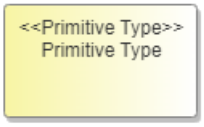
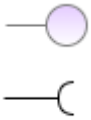
行为图显示了系统中对象的动态行为，可以将其描述为系统随时间的一系列更改。包括：用例图、活动图、状态机图、顺序图、通信图、时间图、交互概述图。

1.2 类图

类图展示了系统的逻辑结构，类和接口的关系。

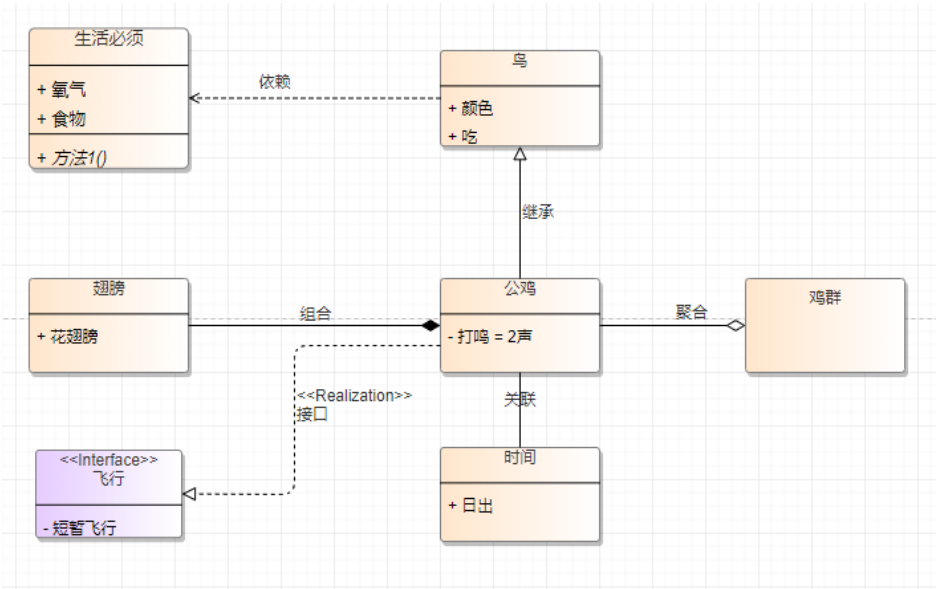
表 1-1 类图元素介绍

元素名	图标	含义
Class		是对象的集合，展示了对象的结构以及与系统的交互行为。

元素名	图标	含义
Interface		接口，可以是单个接口，也可以是抽象的一组接口的组合。 圆形接口与矩形接口意义相同，仅形状不同。
Data Type		数据类型包括原始预 定义的类型和用户自定义的类型。原始类型有：数字、字符串、乘方。用户定义的类型 是枚举类型。程序语言中用于实现的匿名数据类型可以用语言类型定义。
Enumeration		枚举是一种数据结构，它的实例构成了有名字的字面值。通常，同时声明枚举名和其字面值的名字。
Primitive Type		简单类型就是一个事先定义好了的基本数据类型，比如整数或者字符串。
Signal		对象之间异步通讯的声明。信号可以带有表示为属性的参数。
Association Node		关联节点。
Part		表示类或接口的运行时实例。
port		端口。
Interface		Required Interface和Provided Interface之间可以建立Dependency，表明一个组件需要的接口是由另外一个组件提供的。

元素名	图标	含义
Generalization		泛化，表示类与类、接口与接口之间的继承关系，由子一方指向父对象一方。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
Realization		实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Usage		使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。
Instantiate		实例化，声明用一个类的方法创建了另一个类的实例。
Constraint		是一个语义条件或者限制的表达式。UML 预定义了某些约束，其他可以由建模者自行定义。
Anchor		锚点。
Containment		内嵌，表示嵌在内部的类。
Abstraction		抽象是确认一件事物本质特征的行为，这种行为将这个事物与其他所有事物区分开来。 抽象依赖关系表示成从客户元素指向提供者元素的箭头。
Information flow		信息流表示任何图中两个元素之间的信息项（信息项元素或分类器）的流。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

类（Class Diagram）是对象的集合，展示了对象的结构以及与系统的交互行为。类主要有属性（Attribute）和方法（Operation）构成属性代表对象的状态，如果属性被保存到数据库，称为持久化，方法代表对象的操作行为，类具有继承关系，可以继承于父类，也可以与其他的Class进行交互。



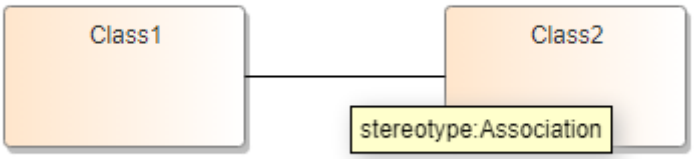
添加属性和方法

类元素添加属性和方法，选中元素右键“属性&方法”，属性&方法的编辑方式参考**如何添加元素属性和方法**。

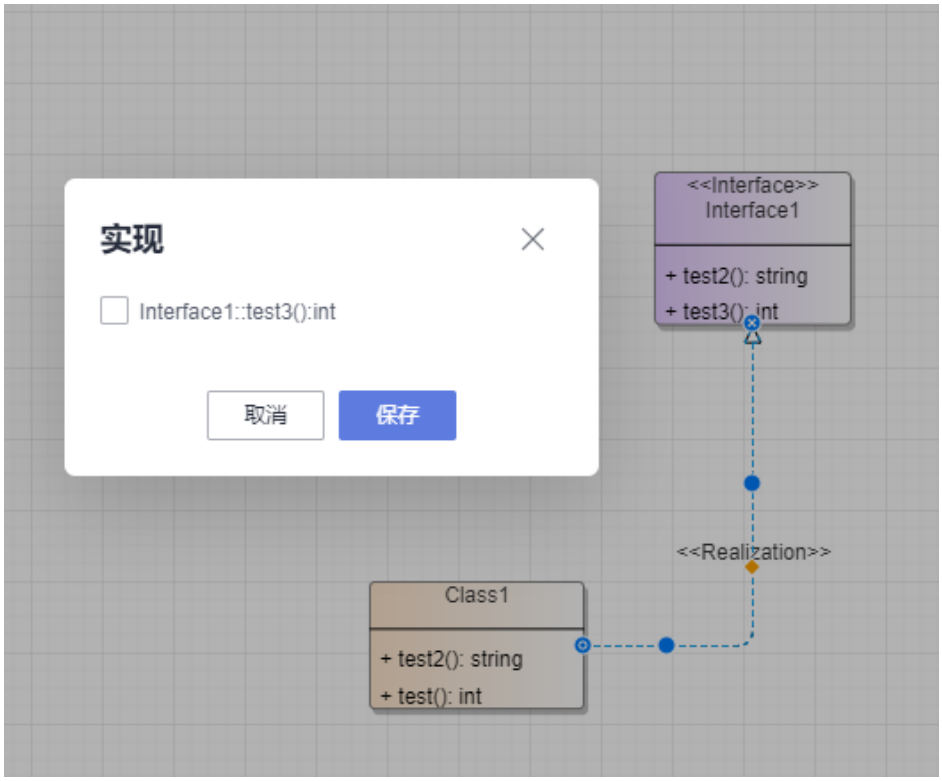
类连线的几种关系

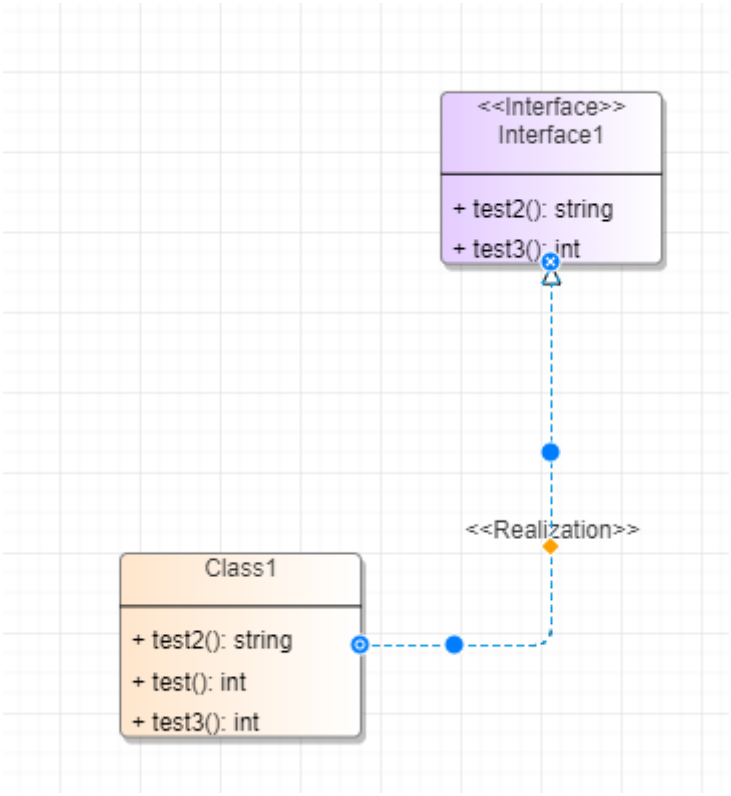
类连线的几种关系表示：关联关系、继承关系、实现关系、依赖关系、聚合关系、组合关系。

- 关联关系：使用Association连线表示类之间的双向关联关系。



- 实现关系：Class类元素通过Realization关系连线快速继承Interface的operation。只支持Class指向Interface，当Interface的operation名称等发生改变，再次Realization连线，会生成新的Operation记录，如下图所示：



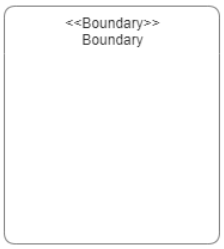
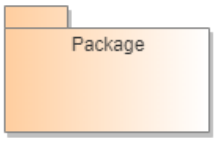
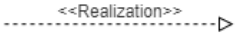
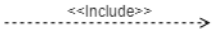


1.3 用例图

用例图用于编写测试用例，将角色与用例联系起来。

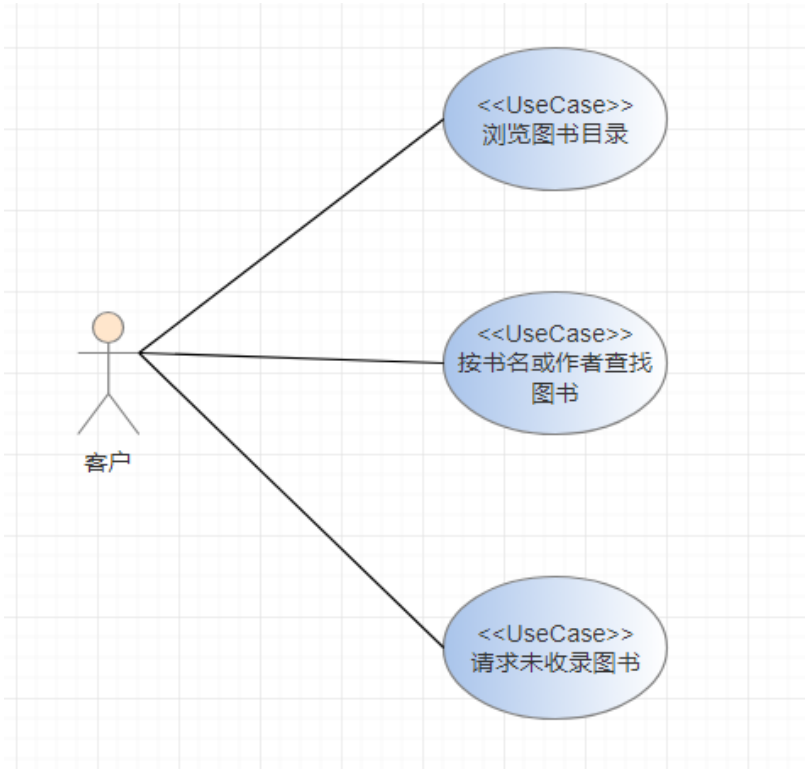
表 1-2 用例图元素介绍

元素名	图标	含义
Use Case		用例，代表的是一个完整的功能。
Test Case		测试用例，是原型的用例元素。通过将元素属性和功能应用于由另一个元素或更确切地说是元素集表示的功能的测试，您可以使用它来扩展“测试”窗口的功能。
Actor		角色，是与系统交互的人或事物。

元素名	图标	含义
Boundary		边界，可以放入元素，形成一个模块。
Collaboration		是对对象和链总体安排的一个描述，这些对象和链在上下文中通过互操作完成一个行为，例如一个用例或者操作。
Collaboration Use		使用协作用于在复合结构图中将协作定义的模式应用于特定情况。
Package		包。对元素进行分组，并为分组的元素提供名称空间。一个程序包可能包含其他程序包，从而提供程序包的分层组织。
Generalization		泛化，表示类与类、接口与接口之间的继承关系，由子一方指向父对象一方。
Realization		实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Constraint		是一个语义条件或者限制的表达式。UML 预定义了某些约束，其他可以由建模者自行定义。
Anchor		锚点。
Containment		内嵌，表示嵌在内部的类。
Include		基用例与包含用例之间的关系。说明如何将包含用例中定义的行为插入基用例定义的行为中。基用例可以看到包含用例，并依赖于包含用例的执行结果。但是二者不能访问对方的属性。

元素名	图标	含义
Extend	<<Extend>>	是指扩展用例与基用例之间的关系。特别是如何将扩展用例定义的行为插入基用例定义的行为序列。
Use	<<Use>>	使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。

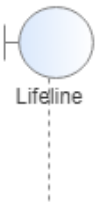
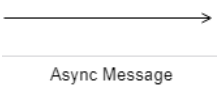
用例图示例，如下图所示：



1.4 顺序图

1.4.1 元素介绍

顺序图是显示对象之间交互的图，这些对象是按时间顺序排列的。顺序图中显示的是参与交互的对象及其对象之间消息交互的顺序。

元素分类	元素名	图标	含义	参考示例
生命线	Lifeline		生命线，在顺序图中表示为从对象图标向下延伸的一条虚线，表示对象存在的时间。	创建生命线
	Boundary Lifeline		边界生命线，表示一个系统的边界，或者系统中的一个软件元素。与用户交互的接口界面，数据库网关，或者菜单，就是边界。	
	Entity Lifeline		实体生命线，系统数据。在顾客服务应用中，顾客实体将管理所有与顾客相关的数据。	
	Control Lifeline		控制生命线，表示一个控制实体或管理者。 它组织和调度在边界（boundary）和实体（entities）间的交互，并作为两者之间的中介者。	
	Actor Lifeline		使用者生命线，使用者是系统的一个用户，意味着一个人，一台机器，一个系统。	
消息线	Message		消息线，消息的发送者把控制传递给消息的接收者，然后停止活动，等待消息的接收者放弃或者返回控制。用来表示同步的意义。	绘制消息线和激活块
	Async Message		异步消息线，消息发送者通过消息把信号传递给消息的接收者，然后继续自己的活动，不等待接收者返回消息或者控制。异步消息的接收者和发送者是并发工作的。	

元素分类	元素名	图标	含义	参考示例
	Reply Message		返回消息线，返回消息表示从过程调用返回。	
	Self Message		自关联消息线，表示方法的自身调用或者一个对象内的一个方法调用另外一个方法。	
	Create Message		创建对象消息线，这个消息指向对象以后，对象的位置就不会出现在顶部，而是创建消息所在的位置。	
	Delete Message		销毁对象消息线，这个消息指向对象以后，对象生命线底部出现一个终止符，表示该对象不再接收新的消息。	
激活块	Activation		在顺序图中，activation位于对象的生命线上。当对象接收到消息并开始执行相关活动时，该对象的生命线上会出现这个矩形框，表示该对象处于激活状态。当活动完成后，矩形框消失，表示对象回到非激活状态。	
组合片段	fragment		组合片段，反映了一个或多个片段的交互（称为交互操作数）由交互运算符控制，其相应的布尔条件被称为互动约束。它将显示为一个透明的窗口并以水平虚线分隔。	绘制组合片段
	loop		循环，片段重复一定次数，可以在临界中指示片段重复的条件。	
	alt		选择，用来指明在两个或更多的消息序列之间的互斥的选择，相当于if…else…。	
-	Note		文本框。	-

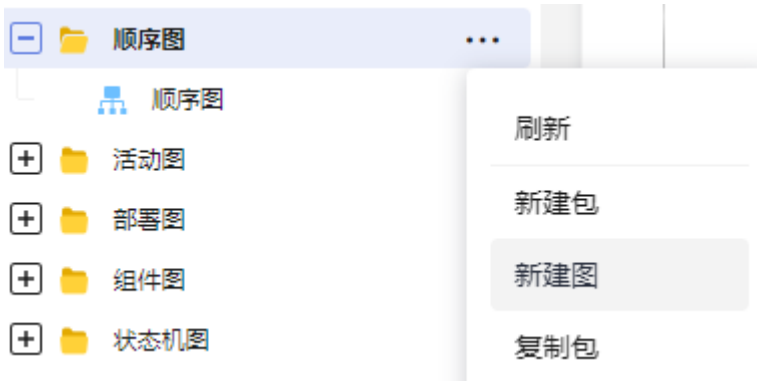
元素分类	元素名	图标	含义	参考示例
-	Endpoint		结束点，流程结束、异常退出的地方用“结束”表示。	-
-	Diagram Gate		表示图的门口，用法是可以链接到另外一张图。	使用Diagram Gate
-	State/Continuation		状态常量是生命线的约束，运行时始终为true。 延续虽与状态常量有同样的标注，但是被用于复合片段，并可以延伸跨越多条生命线。	-

1.4.2 创建顺序图

绘制顺序图时，必须保证图的类型为顺序图，否则可能导致无法绘制对应消息线。

已创建的模型图修改图类型具体请参考**如何查看和修改模型图类型**。

步骤1 选中工程树包节点，单击“更多操作 > 新建图”。



步骤2 弹出新建图弹窗选择“UML > 顺序图”，填写顺序图基本信息。

----结束

1.4.3 创建生命线

生命线介绍

生命线的类型/构造型变化不会影响消息线的发送/接收逻辑。通常情况下，生命线是同高同位置的，表示从发送第一条消息开始，这些生命线所代表的对象在系统中是存在的，直至最后一条消息结束，这些生命线对象也不会被销毁。

由创建对象消息线（Create Message）创建的生命线，它是系统运行过程中动态生成的对象，其生命线位置要低于其余生命线。

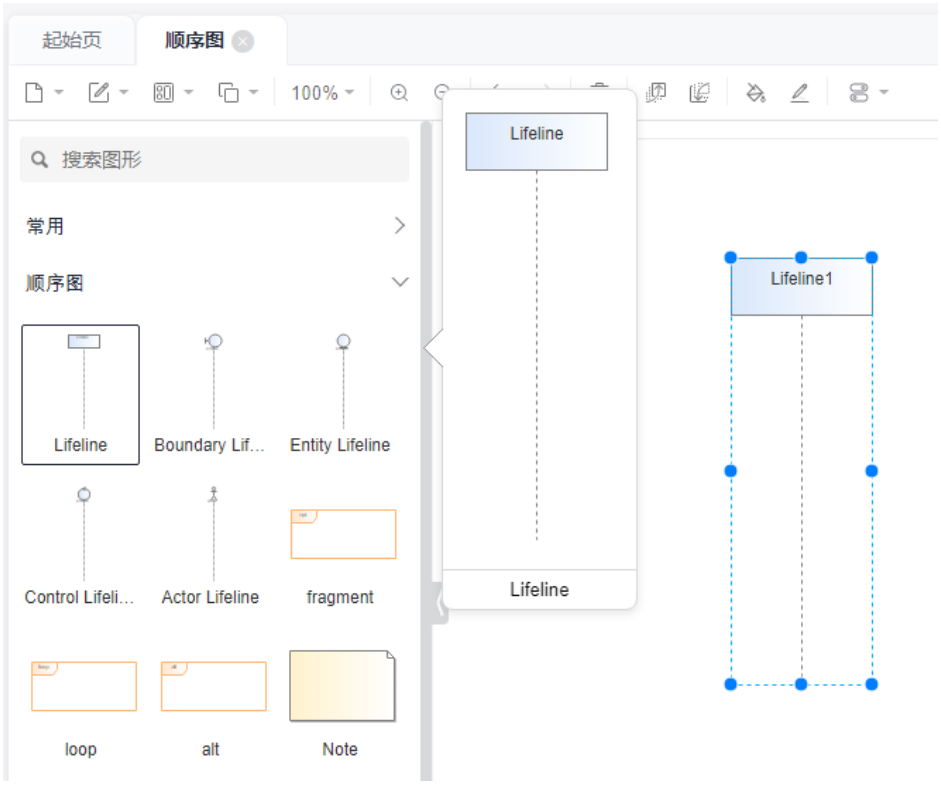
同样，由销毁对象消息线（Delete Message）指向的生命线会在该条消息线指向之后被销毁，其生命线高度短于其余生命线。

介绍参考**元素介绍**中生命线分类。

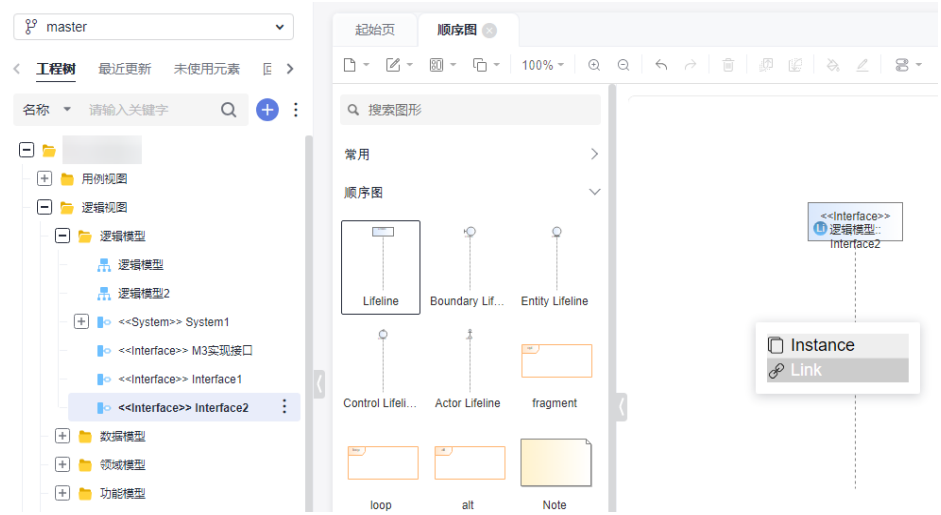
创建生命线

在创建生命线前，需要先考虑哪种类型的生命线能更好地表示所要表达的对象。您可以通过以下三种方式创建生命线。

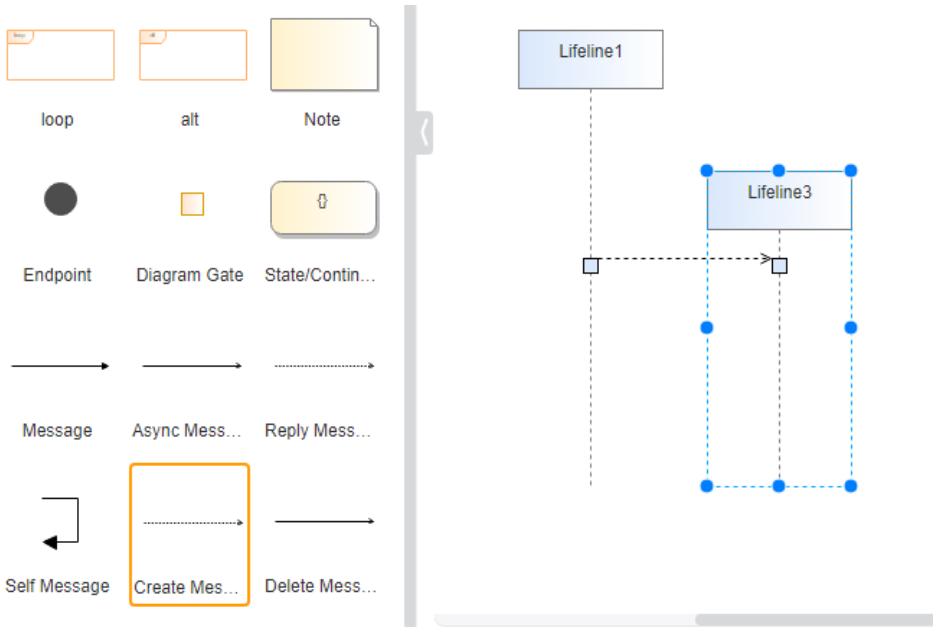
- 工具画板中选择一个生命线拖拽至画布中。



- 从工程树中其它模型视图中的元素拖拽至画布中生成对应生命线，您可以使用Link方式引用。



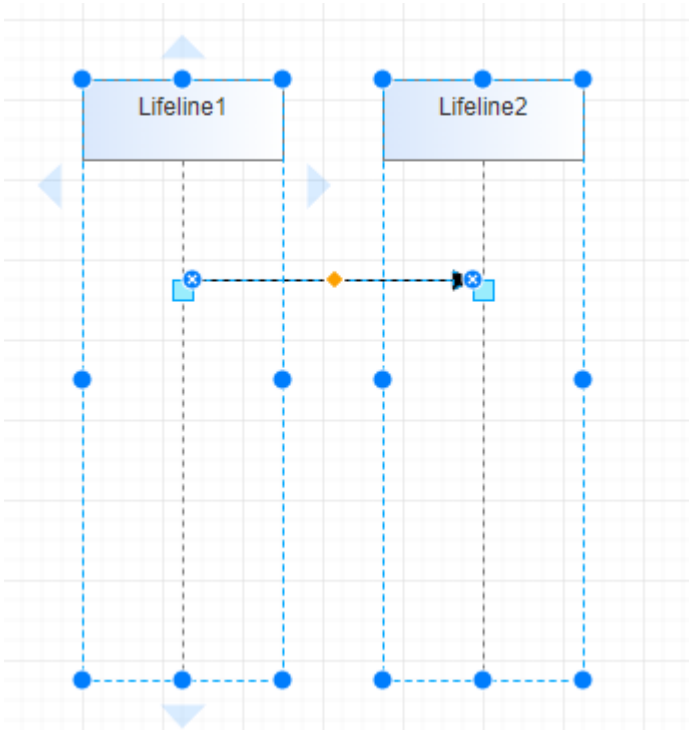
- 生命线也支持由创建消息线（Create Message）触发创建，其位置通常低于别的生命线。



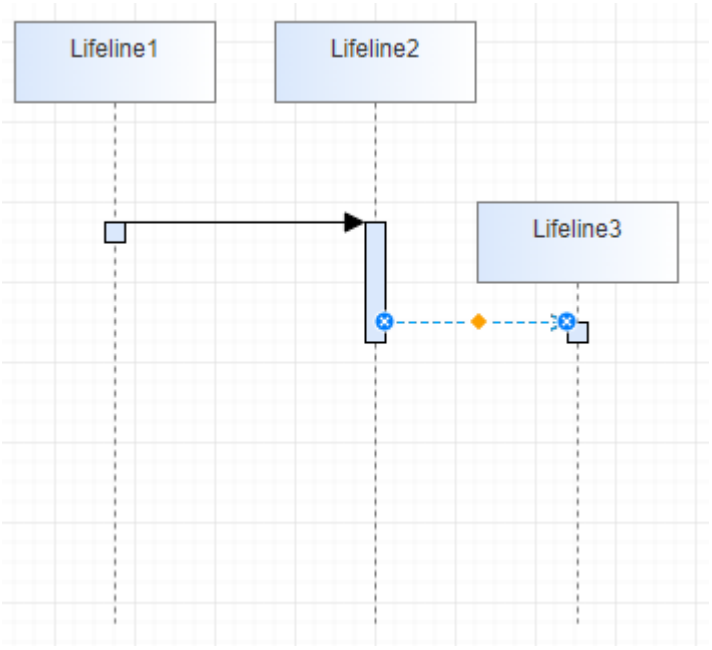
移动生命线

通常情况下，模型图中的生命线具备同样的位置和高度（动态创建和销毁的生命线除外）。

- 单个生命线不支持上下垂直移动，只支持左右水平移动。
- 全选生命线后，可以在垂直方向上移动，调整元素在画布中的位置。
- 调整任一生命线的高度，其余生命线会同步调整，保证所有生命线底部对齐。



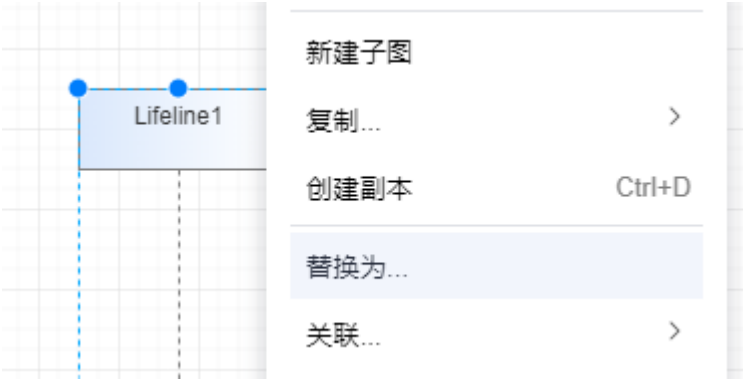
如果生命线是通过Create Message创建的，则可以操作该Create Message对目标生命线进行上下移动，但时间上不能早于之前最早生命线。



替换生命线

模型中的生命线可以变更成其余类型和构造型，也可以快速替换为其他生命线，避免纠错时删除重绘的麻烦。

选中生命线，右键单击“替换为...”。

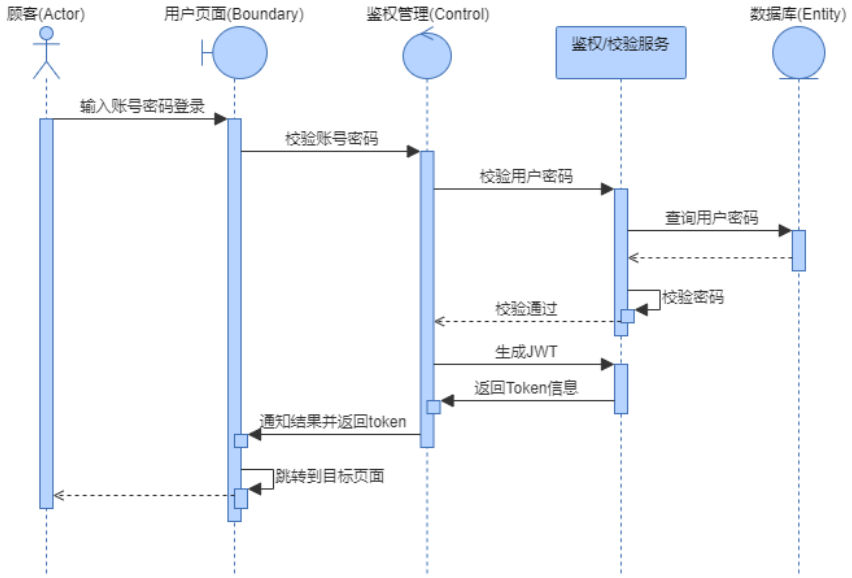


- 类型&构造型：修改生命线的类型和构造型。

- 元素替换：替换成工程内的其它元素。

模型示例

生命线在登录鉴权场景下的使用。



Actor Lifeline表示人或其余参与系统（机器），代表的是一种执行/参与的角色。

Boundary Lifeline表示目标系统中的边界对象，可用于表示MVC模式中的View（视图），该示例中它表示的是运行于浏览器上的用户界面或者其余客户端页面。

Control Lifeline表示目标系统中的管理/调度/控制的对象，常用于表示MVC模式、控制模式中的Controller（控制器），该示例中它负责协调首次登录的密码校验、JWT的生成以及和用户界面的交互。

Entity Lifeline表示目标系统中的数据对象，常用于表示MVC模式中的Model（数据模型）、数据库或者其余数据存储实体，该示例中表示的是存储了用户账号密码的数据库（或数据库表）。

实际上，在用顺序图建模目标系统的时候，这些特殊生命线并不是一定被使用的，它们的存在不会影响顺序图本身的行为逻辑，只在图形展示上具备更好的示意效果，用户完全可以用默认生命线代替它们。大部分行为建模场景中，使用最广泛的是默认生命线，比如对某个子系统的各组件交互逻辑建模，此时使用默认生命线表示各组件对象即可。

1.4.4 绘制消息线和激活块

约束与限制

根据建模规范限制，建模的连线不能独立存在于图中，两端必须是连接在元素上的，因此不允许消息线直接拖拽至画布上。且绘制顺序图时，必须保证图的类型为顺序图，否则可能导致无法绘制对应消息线。参考常见问题**如何查看和修改模型图类型**。

自消息线分为同步自消息线和异步自消息线，工具当前仅支持同步自消息线。

消息线介绍

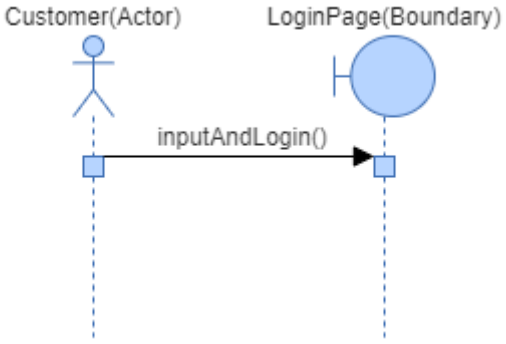
顺序图用消息线描绘元素之间的工作流或者活动。软件模型中，消息线可以用来代表源端或目标端元素的操作或者属性。您可以根据需求**绘制消息线**，绘制消息线后，也可以**提升或降低消息线层级**。

介绍参考**元素介绍**中消息线和激活块分类。

建模规范

- 同步消息线（Message）/异步消息线（Async Message）

  这两类消息线都是指令/任务型消息，一般用来表达发送方向接收方传递指令或调用任务，这些任务通常是有执行时间的。目标生命线在执行任务的这段时间处于激活状态，会在生命线上生成对应的激活块。

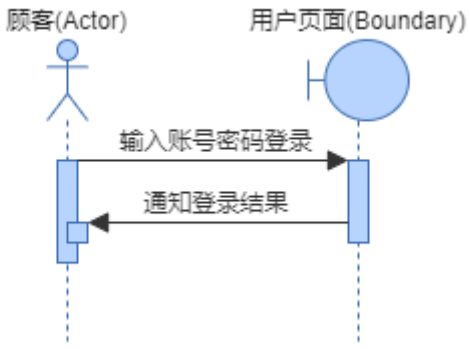


对应的代码模型。

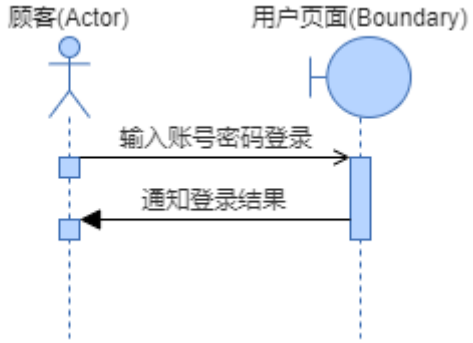
```
class LoginPage {
    inputAndLogin() {
        // 输入用户名密码，并执行登录逻辑
    }
}

class Customer {
    buy() {
        // 从顾客发送消息给用户界面，进行登录操作
        loginPage.inputAndLogin();
    }
}
```

同步消息线与异步消息线的区别是相对于消息发送方而言的，同步消息线的发送方会等待指令/任务执行完成，在指令/任务执行过程中它也是处于激活状态，所以发送方的生命线上会有对应激活块。而异步消息线的发送方不会等待指令/任务完成，在消息线发送之后它就处于非激活状态了。

“输入账号密码登录”是同步消息，“顾客”生命线在发送完之后处于等待状态，直到“通知登录结果”消息传来，它都是处于激活状态。

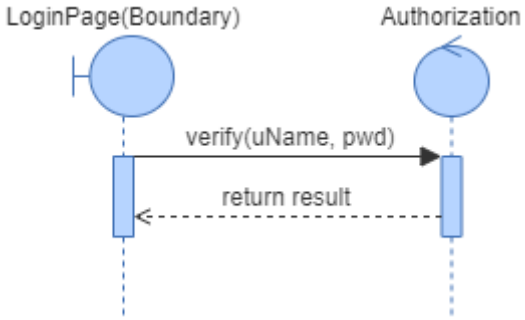


“输入账号密码登录”是异步消息，“顾客”生命线在发送完之后不再等待，在“通知登录结果”传来的这段时间内，它都是处于非激活状态。



- 返回消息线（Reply Message）

  返回消息线是针对同步/异步消息线的结果响应，它本身不是指令/任务型的，即它不会触发目标生命线的任务调用或者指令执行。



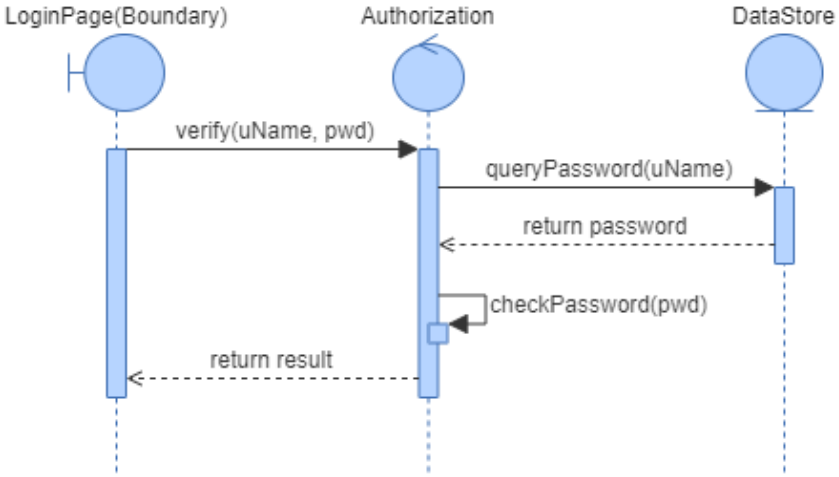
  对应到代码模型，返回消息线一般表示的是方法/函数中的renturn语句，或者是接口请求中的数据返回。

```
class LoginPage {
    inputAndLogin() {
        authorization.verify(this.getUsername(), this.getPassword());
    }
}

class Authorization {
    verify(uName, pwd) {
        const result = this.check(uName, pwd);
        // 返回校验结果就是针对verify消息的响应，对应的是reply message
        return result;
    }
}
```

  从上面模型图和代码中可以看到，同步消息线是对verify方法的调用，属于指令/任务型的消息，而返回消息线是verify方法中的结果返回，它不直接调用或者触发LoginPage中方法的执行。

- 自消息线（Self Message）

  自消息线是目标生命线发向自己的指令/任务型消息，它会在原有激活块上生成二级或更高层级激活块。对应到软件模型，自消息线一般表示对象调用自身的方法/函数。



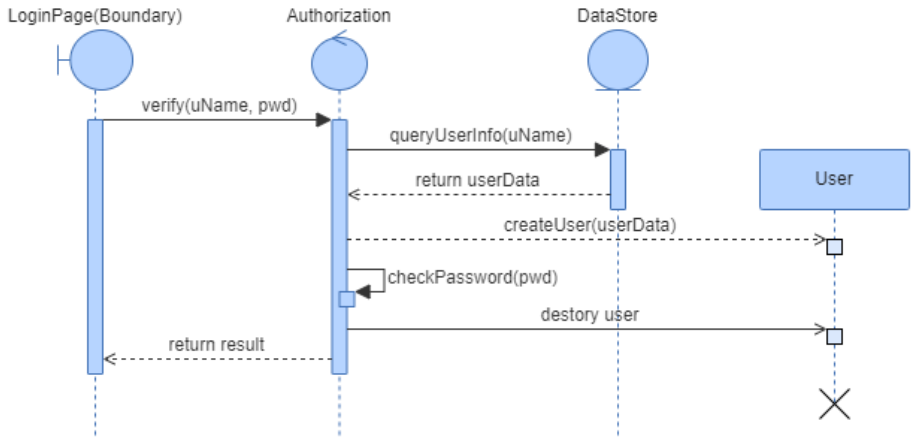
对应的代码模型。

```
class Authorization {
    verify(uName, pwd) {
        this.password = DataStore.queryPassword(uName);
        // 调用对象自身的checkPassword方法在顺序图中表现为自消息线
        const result = this.checkPassword(pwd);
        return result;
    }

    checkPassword(pwd) {
        // 校验用户输入的pwd和this.password，并返回校验结果。
    }
}
```

- 创建对象消息线（Create Message）/ 销毁对象消息线（Delete Message）

  顾名思义，创建对象消息线会动态创建目标对象，该目标对象可能是一个临时对象，也可能是一个动态创建的永久对象。对应软件模型，当一个对象不是在系统运行初期就存在，而是在运行过程中由其余对象创建出来的，则该对象的创建过程适合用创建对象消息线来表示。

  销毁对象消息线会动态销毁目标对象，目标对象可能是由创建对象消息线临时生成的，也可能是系统运行初期就存在的对象。对应软件模型，当要进行资源回收时，通常需要销毁目标对象，此时可以用销毁对象消息线来表示资源回收过程。



对应的代码模型。

```
class Authorization {
    verify(uName, pwd) {
        const userData = DataStore.queryUserInfo(uName);
        // 动态创建User对象，用来进行密码校验
        this.user = new User(userData);

        const result = this.checkPassword(pwd);

        // 销毁不再使用的User对象，回收资源
        this.user = null;

        return result;
    }

    checkPassword(pwd) {
        return this.user.checkPwd(pwd);
    }
}
```

- 激活块（Activation）

  激活块表示的是目标对象（生命线）处于激活状态的时间段，它是由消息线触发生成的。顺序图中一共有六类消息线，其中有五类消息线（Message、Async Message、Self Message、Create Message、Delete Message）是指令/任务型消息，可以在目标生命线上生成激活块（创建对象消息线对应的指令是创建目标对象），而返回消息线（Reply Message）一般是对其余消息线的结果响应。

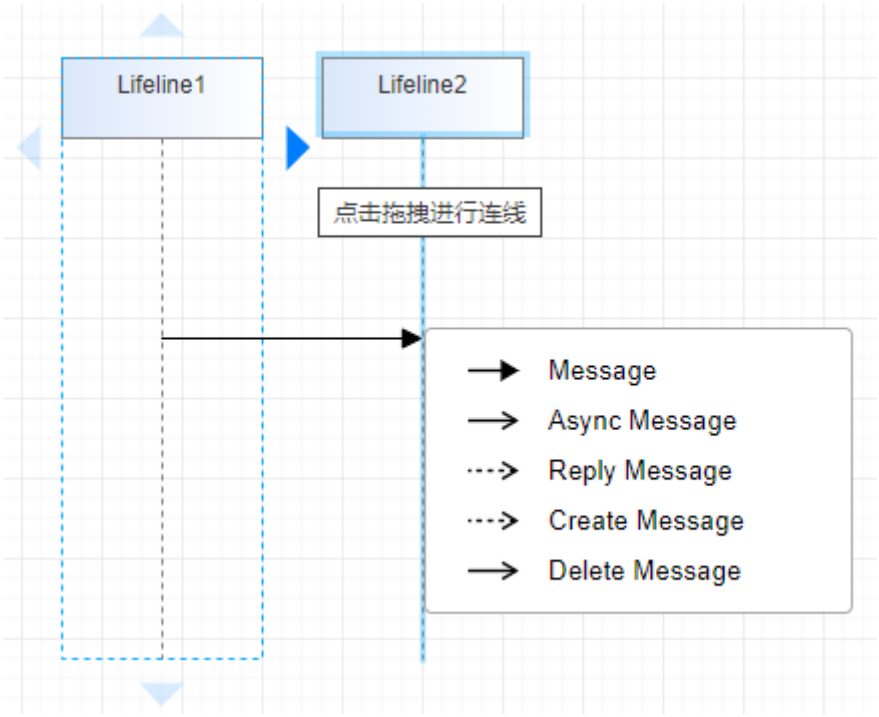
  激活块上可以挂载二级甚至更多层级的激活块，表示的是对应层级消息的子任务激活状态。

建模步骤

您可以通过如下方式绘制生命线：

步骤1 绘制消息线。

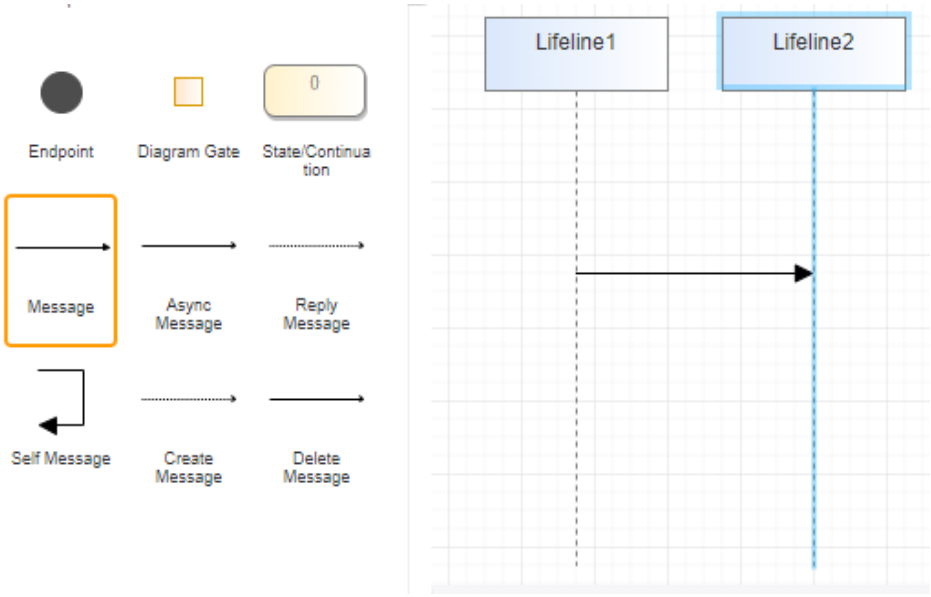
- 选择生命线，从元素的辅助连线三角标单击拖拽至另一个生命线上，选择消息线类型。



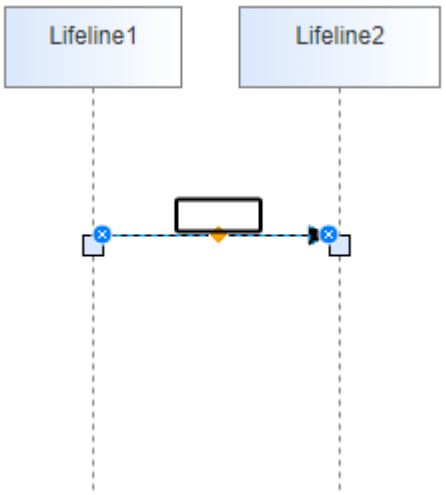
- 在元素面板上单击或者双击选择要使用的消息线，然后从生命线虚线任意位置左键拖拽生成消息线到目标生命线上。

  单击：元素面板消息线外框处于选中状态，可以使用该消息线一次，在图中绘制消息线后会自动取消单击选择状态。

  双击：元素面板消息线外框及背景处于选中状态，可以多次使用该消息线，使用完后需单击取消该连线的选中状态。



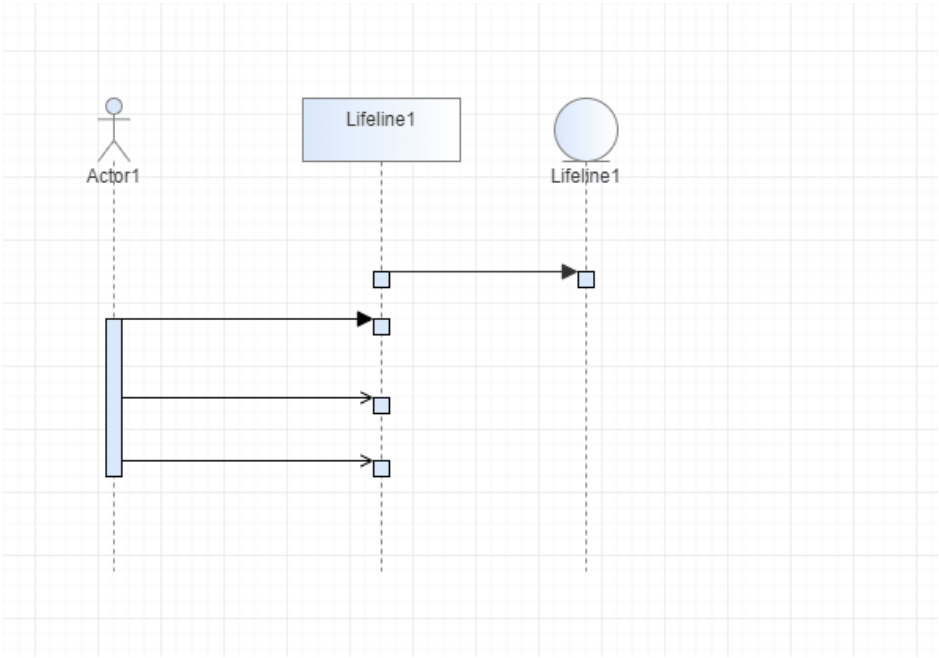
步骤2 编辑消息线。选中连线按F2键/回车键/右键单击消息线选择“编辑”，可以编辑消息线名称。



步骤3 移动消息线。消息线有两种移动方式：平移和跨线移动。

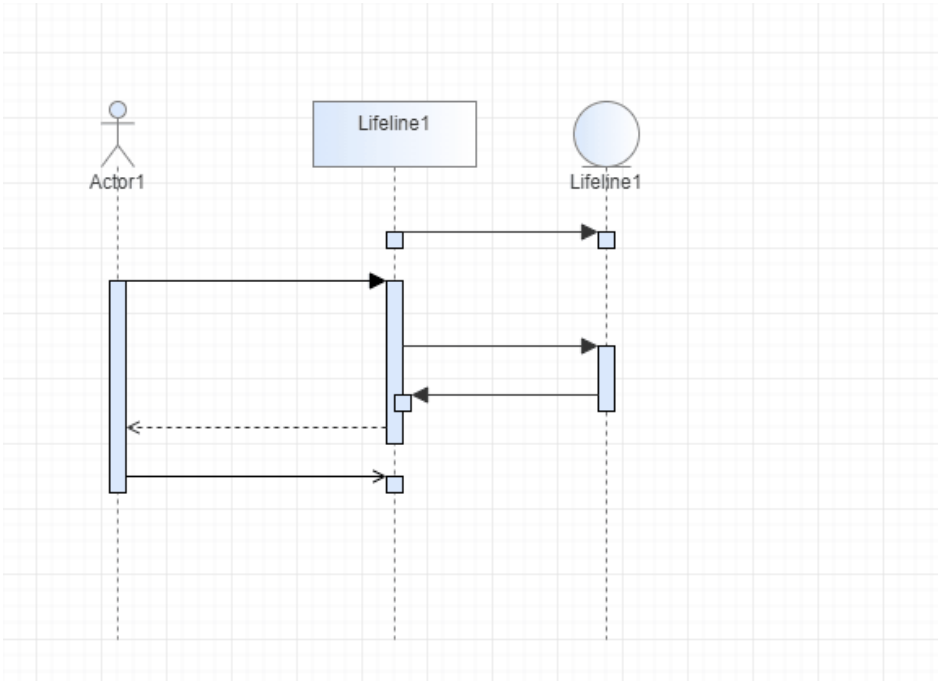
- 平移是指对选定消息线在垂直方向上移动，该消息线相对于其余消息线的位置不变，但该消息线及其下方其余消息线均会同步移动，平移的消息线不能越过该消息线上方的其余消息线，平移操作可能会触发生命线高度变化。平移的目的一般是调整消息线之间的间距，或者为插入新消息线留空间。

  选中消息线，鼠标左键按住消息线可以在垂直方向上平移。



- 跨线移动是指调整选定消息线在生命线上的位置，该消息线相对于其余消息线的位置会变化，但不会影响其余消息线的位置（其余消息线不移动）。跨移的消息线可以移动到生命线上任意位置，由于消息线的顺序发生变化，一般会导致激活块的状态发生改变。跨线移动的目的一般是为了纠错，某个消息线原先的位置画错了，通过跨线移动对其进行调整。

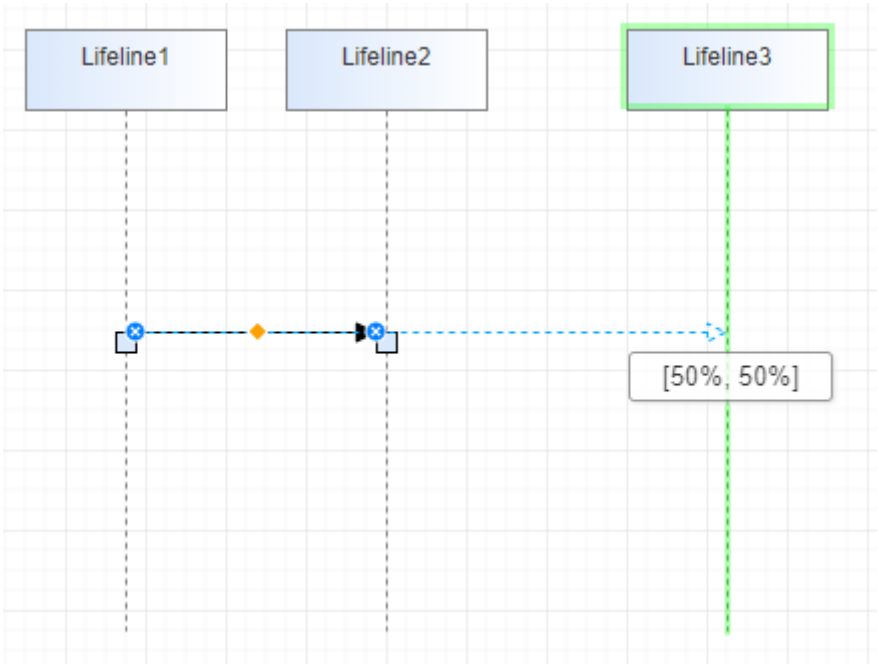
  选中消息线，按住Ctrl按键，再利用鼠标左键按住消息线在垂直方向上移动。



步骤4 变更消息线类型/连接端。对于已连接的消息线，工具提供了变更消息线类型以及变更连接端的能力。

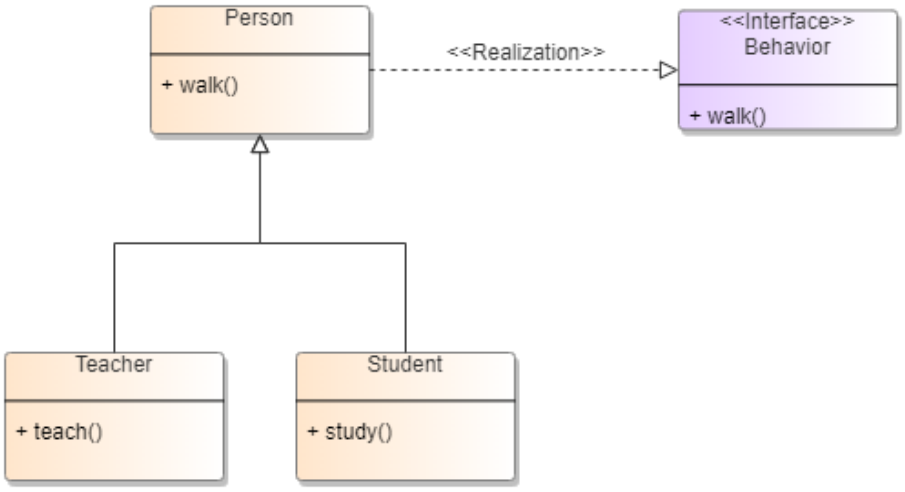
- 右键单击消息线，选择“类型&构造型”，修改消息线类型。

- 左键选中消息线，拖动消息线起点或者终点向其余生命线拖去，可以将连接点变更到新生命线上。

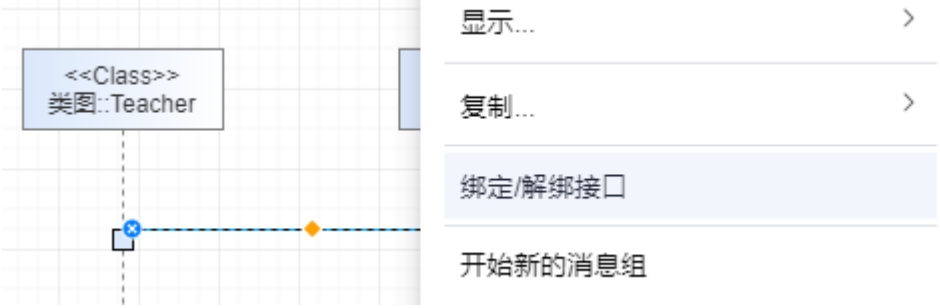


步骤5 绑定方法/接口。如果生命线是定义方法或者实现接口的元素，则对于指向该生命线的消息线，可以绑定目标元素上定义的方法或者接口。

在类图中的Student元素定义方法Study

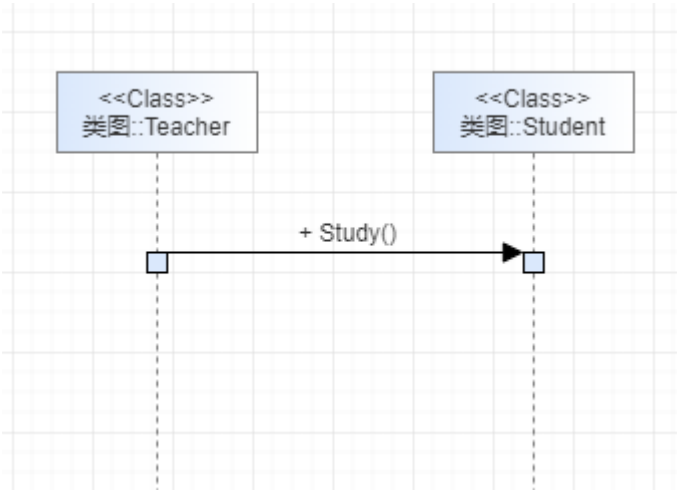


选中顺序图消息线，右键 > 绑定/解绑接口。

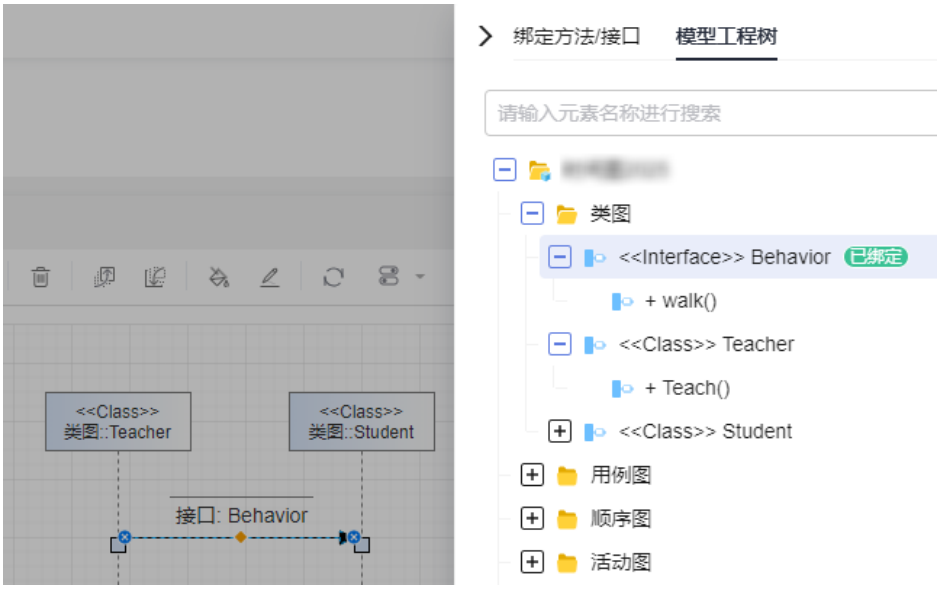


选择Student元素定义的方法Study。

单击绑定，方法同步更新到消息线上。



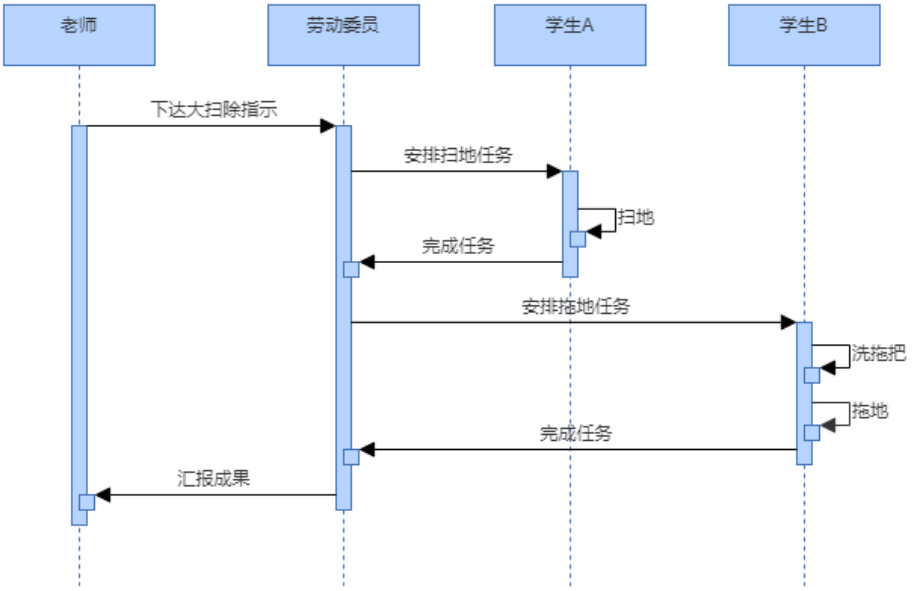
同样可以绑定接口元素。



----结束

模型示例

- 以老师安排学生大扫除为例解释下顺序图中激活块的概念（映射到计算机系统/软件模型，激活块代表的是对应部件或者软件对象处于激活状态的时间段）。



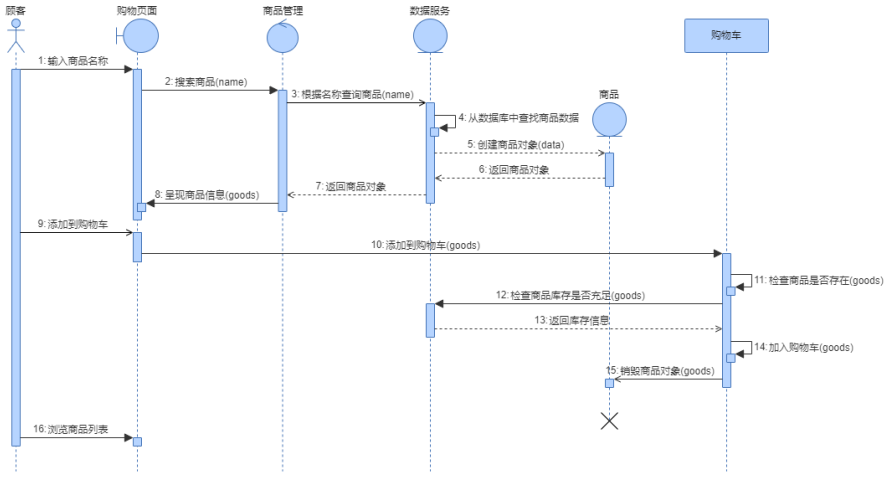
如上所示，老师安排劳动委员组织学生对教室进行大扫除活动，老师先给劳动委员下达了大扫除的任务指示，并要求在大扫除结束后向他汇报成果。假设老师在大扫除期间没有其余事情可做，下达任务后一直等待任务完成，那么在老师这条生命线上会有一个贯穿任务始终的激活块，表示老师在这个时间段内处于活跃状态（全程关注大扫除这个任务）；而劳动委员负责大扫除任务的子任务安排（任务调度），他的活跃状态与老师几乎一致（安排任务、验收结果以及汇报成果）。

学生A和学生B在这个过程中负责不同的子任务，由于子任务之间存在先后顺序（必须先扫地再拖地），所以在图中呈现的是上下顺序。对于学生A来讲，他在接到任务、扫地及任务结束验收期间是处于活跃状态的，而此时学生B还未被安排任务，此时的他是未激活的（所代表的生命线在对应时间段内没有激活块）。同理，对于学生B来讲，从他接到任务、拖地到任务结束验收期间他处于激活状态，而此时学生A由于完成了子任务且没有其余任务，他就是处于未激活状态。

附载在激活块上的小方块，实际也是激活块，它表示的是子任务。比如学生B在接到拖地任务后，就执行了洗拖把和拖地这两个子任务，两个子激活块代表的是学生B在这两个子任务上的活跃时间段。

综上，激活块代表的是目标对象由于执行任务（进入激活状态）到结束任务（恢复未激活状态）的时间段，它的高度并不完全等同于任务时间长短，只是用来表示任务执行顺序以及目标对象激活状态。

- 以用户在线购物为例，结合代码示例解释消息线的使用场景，以及其与对象/方法之间的对应关系。



上述示例对应下面的代码，可以结合代码中函数调用关系理解示例中消息线的连接方式：

```
class Customer {
    browse() {
        // 对应消息线“1: 输入商品名称”
        shoppingPage.inputAndSearch();
        // 对应消息线“9: 添加到购物车”
        shoppingPage.addToShoppingCart();
    }
}

class ShoppingPage {
    constructor() {
        this.goods = null;
        this.goodsManager = new GoodsManager(this);
        this.shoppingCart = new ShoppingCart();
    }

    inputAndSearch() {
        const goodsName = this.getInputValue();
        // 对应消息线“2: 搜索商品名称(name)”
        this.goodsManager.queryGoodsByNameAndDisplay(goodsName);
    }

    displayGoods(goods) {
        this.goods = goods;
        // 在界面上呈现商品详情信息
    }

    async addToShoppingCart(goods) {
        // 对应消息线“10: 添加到购物车(goods)”
        this.shoppingCart.add(goods);
    }
}

class GoodsManager {
    constructor (shoppingPage) {
        this.dataService = new DataService();
        this.shoppingPage = shoppingPage;
    }

    async queryGoodsByNameAndDisplay(goodsName) {
        // 对应消息线“3: 根据名称查询商品(name)”,
        // 通过数据服务查询是异步操作，需要用异步消息线表示。
        const goods = await this.dataService.queryByName(goodsName);
        // 对应消息线“8: 呈现商品信息(goods)”
        this.shoppingPage.displayGoods(goods);
    }
}
```

```
48
49  class DataService {
50      async queryByName(goodsName) {
51          // 对应消息线“4: 从数据库中查找商品数据”
52          const goodsData = await this.queryFromDB(goodsName);
53          // 对应消息线“5: 创建商品对象(data)”和“6: 返回商品对象”
54          const goods = new Goods(goodsData);
55          // 对应消息线“7: 返回商品对象”
56          return goods;
57      }
58
59      async queryFromDB() {
60          // 从数据库中查找商品数据
61      }
62
63      async checkInventory() {
64          // 检测对应商品的库存是否为空
65      }
66  }
67
    Complexity is 4 Everything is cool!
68  class ShoppingCart {
        Complexity is 4 Everything is cool!
69      async add(goods) {
70          // 对应消息线“11: 检查商品是否存在(goods)”
71          if (this.isAdded(goods)) {
72              return;
73          }
74
75          // 对应消息线“12: 检查商品库存是否充足(goods) 和 13: 返回库存信息”
76          const isEmpty = await this.dataService.checkInventory(goods);
77
78          if (!isEmpty) {
79              // 对应消息线“14: 加入购物车(goods)”
80              this.addToList(goods.toData());
81          }
82
83          // 对应消息线“15: 销毁商品对象(goods)”
84          goods.destroy();
85      }
86
87      isAdded() {
88          // 检测购物车里是否有该商品
89      }
90
91      addToList(goods) {
92          // 将商品加入购物车列表
93      }
94  }
```

1. 消息线3对应GoodsManager中的queryByName调用，由于调用DataService中的queryByName是异步操作，所以消息线3是异步消息线。
2. 消息线5基于商品数据动态创建了商品对象，商品对象不是在系统运行初期就存在的，所以消息线5用Create Message来表示对象的动态创建过程。
3. 消息线7是数据服务对商品管理查询信息的返回，对应DataService中queryByName函数的return语句，它不会主动触发商品管理中函数的执行，所以消息线7是返回消息线。
4. 消息线14将商品对象的元数据加入到购物车中，商品对象不再被使用，会在add函数结束后被自动回收。如果在回收之前需要对其进行显式销毁，可以用消息线15的销毁对象消息线来表示。

5. 消息线9是将商品对象加入到购物车中，由于涉及异步查询库存信息，所以消息线9是异步消息线。用户执行该操作后可以继续执行消息线16的商品浏览，消息线16并不需要等待消息线9及其后续所有子任务执行完成，它和消息线9是非阻塞的并发关系。但由于顺序图是平面图，所以在视觉呈现上，消息线9和消息线16还是上下关系。

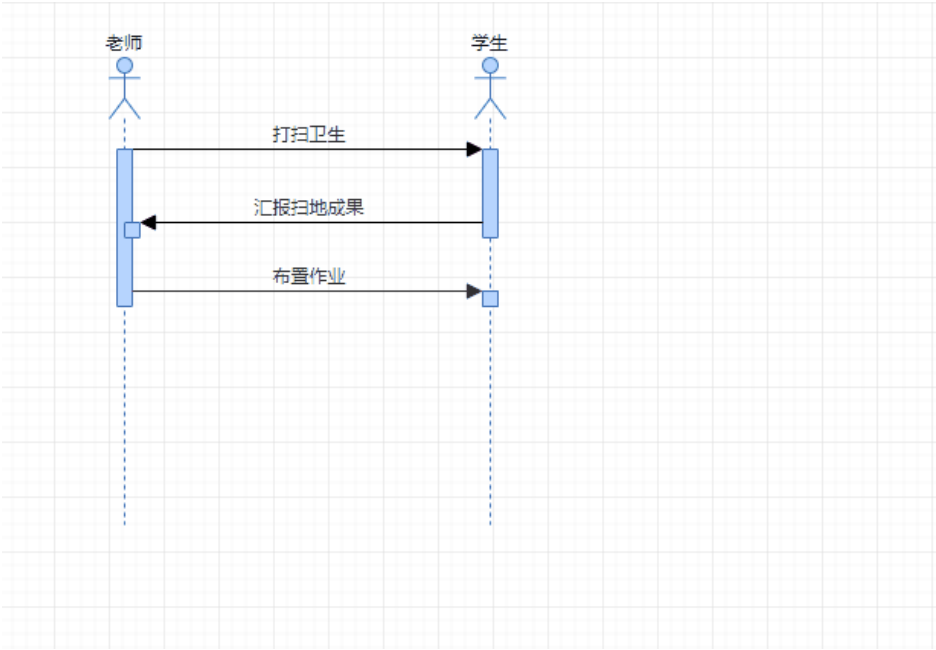
1.4.5 自定义激活块

通常情况下，激活块的生成和断连状态是由指向它的消息线及上下文控制的，基于同步/异步/返回消息线的连线规则，自动计算出对应激活块的长度以及和上下激活块的连接状态。

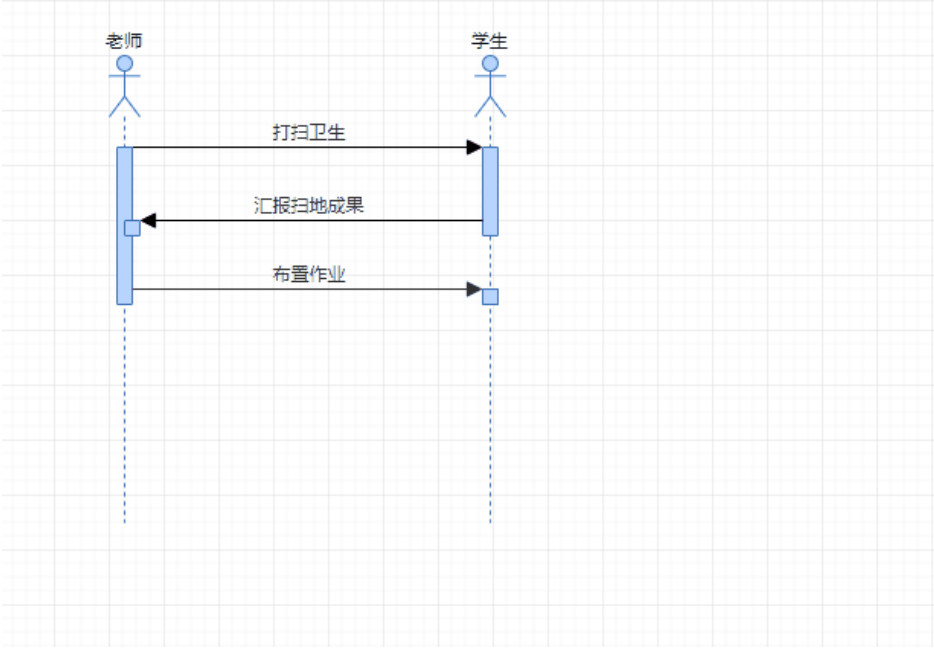
但是，某些建模场景下，现有的计算规则可能无法满足用户对建模逻辑的描述，用户需要在当前计算规则上，对顺序图激活块的状态施加进一步的控制。因此工具提供了六种控制激活块状态的自定义方式。

开始新的消息组

顾名思义，将一条消息线设置为“开始新的消息组”，意味着该消息线及其下方的其余消息线是一段新的执行周期。它通常用来将连在一起的两段逻辑拆分开，该设置会作用于消息线的源端对象上。如下图所示，“布置作业” 任务是老师在“打扫卫生”任务结束后的后续动作，此处通过“设置为新的消息组”，人为将它定义为两个独立的任务。

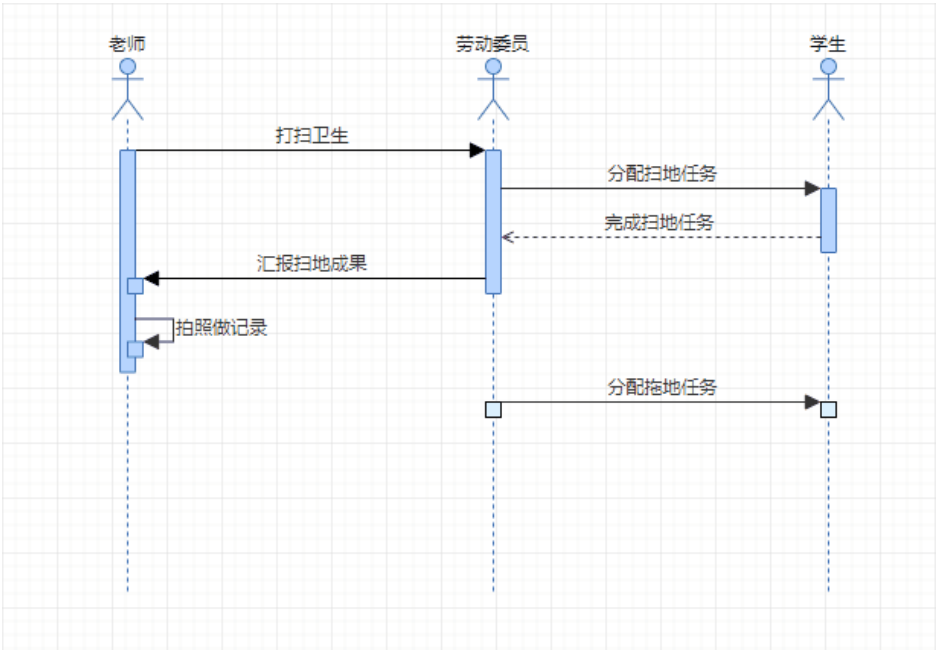


开启新的消息组后，如果对应激活块没有拆分为独立逻辑段，则可能是该激活块上方的消息线设置了“向下延长源端激活块”，导致上方的激活块向下延伸。



向上延长源端激活块

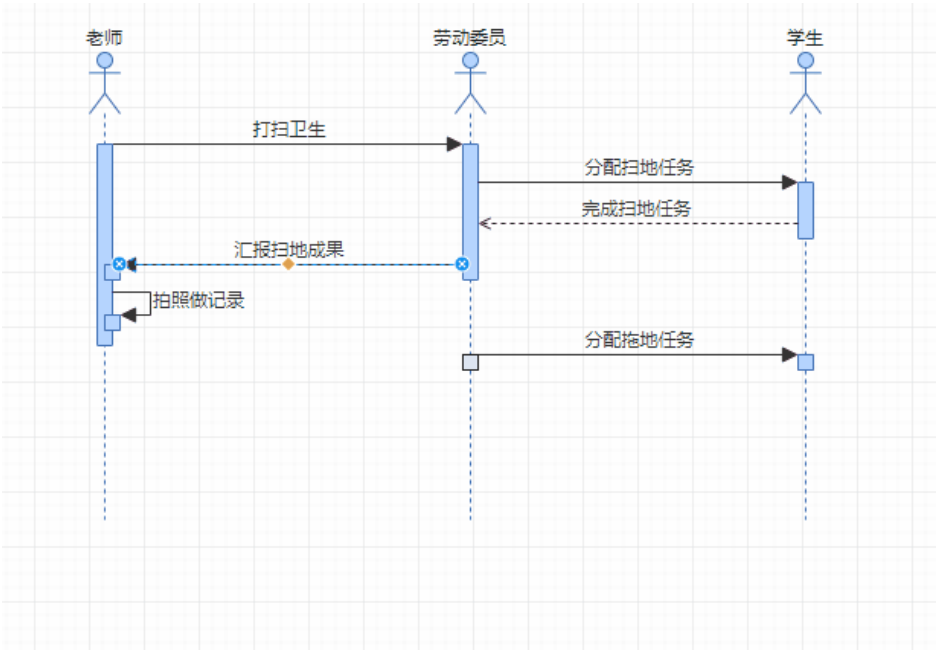
当一条消息线的源端激活块不与它上方的其余激活块相连，通常意味着该消息线代表的是一个独立任务。若要人为将该独立任务和上方任务相连，需要对其设置“向上延长源端激活块”，手动将该任务的执行周期与上方的任务连接到一起。如下图所示，“分配拖地任务”与上面的“分配扫地任务”是两个独立的任务，通过设置“向上延长源端激活块”将其合并成一个执行任务。



向下延长源端激活块

与“向上延长源端激活块”场景类似，当一条消息线的源端激活块不与其下方的其余激活块相连，也可以通过将该消息线设置为“向下延长源端激活块”，将该消息线任

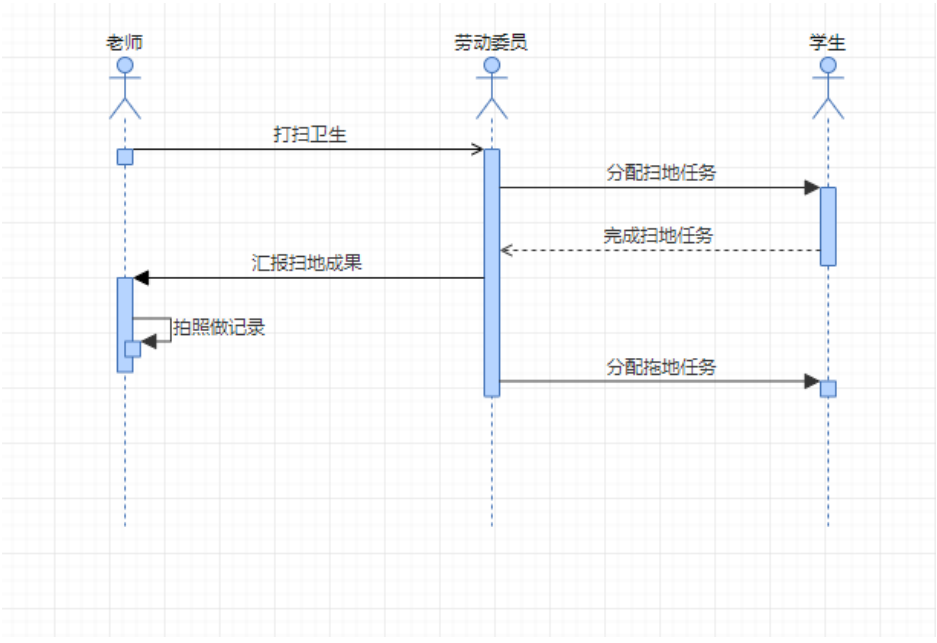
务与下方任务合并为一个执行任务。如下图所示，“汇报扫地成果”与下方的“分配扫地任务”是两个独立的任务，通过对“汇报扫地成果”消息线设置“向下延长源端激活块”将其合并为一个执行任务。



结束源端激活块

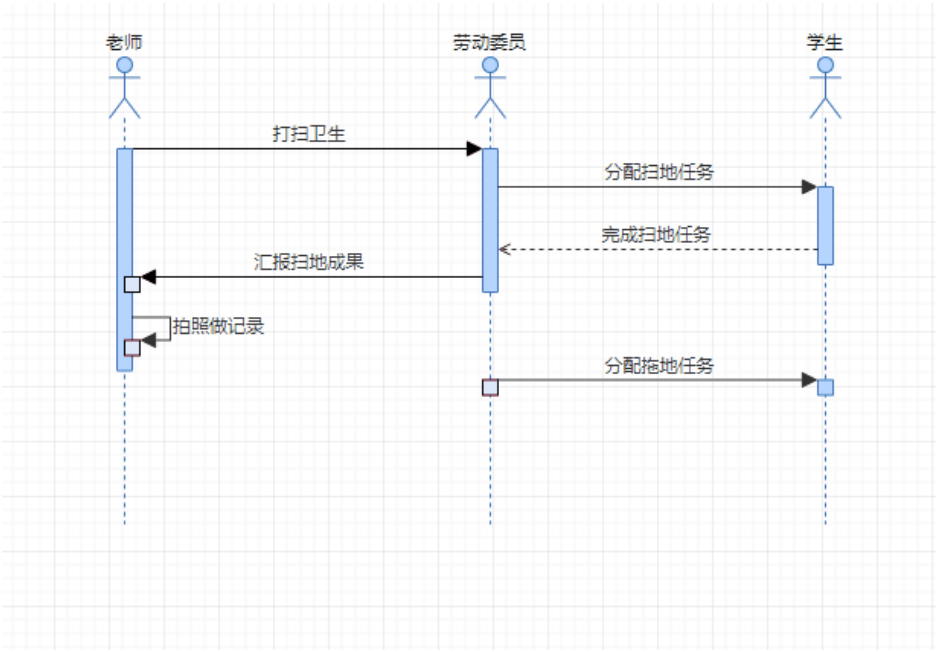
当用户想要表示某一任务在消息指令发出之后，该任务将处于非激活状态，可以通过显式设置某消息线为“结束源端激活块”来实现。通常这样的场景应该用异步消息线来实现，异步消息发出指令/任务后，该消息线的源端对象即处于非激活状态。

假设老师发出“打扫卫生”任务后就去做别的事情，则他的这个任务是个异步任务，通常用“异步消息线”来表示该指令/任务。



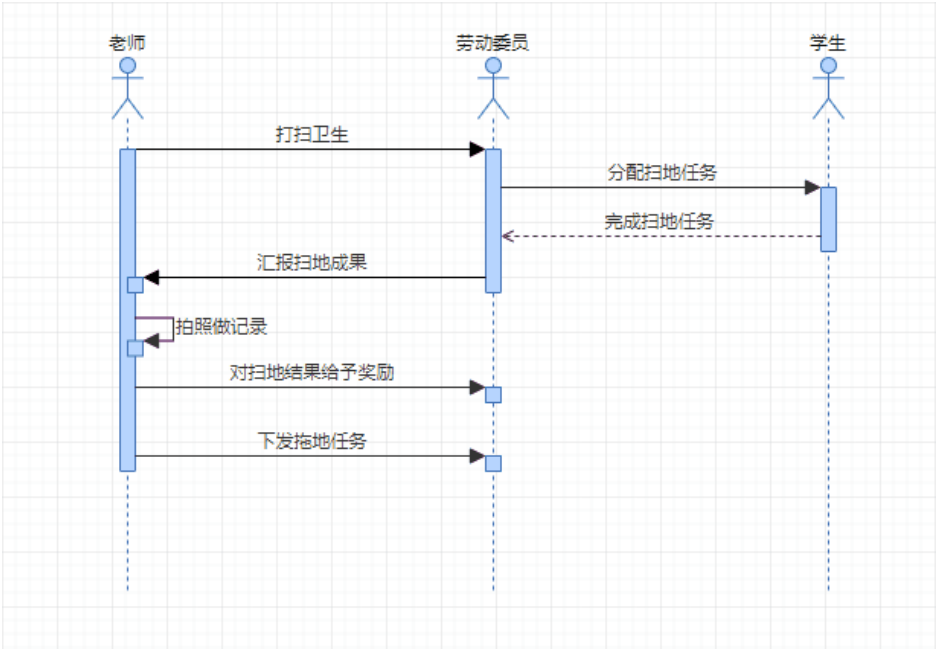
而在下图中，如果用“同步消息线”来表示该指令/任务，则需要将“打扫任务”这个消息线设置为“结束源端激活块”，则此时它的源端激活块状态和左图保持一致，代

表的是“打扫卫生”和“汇报扫地成果”这两个任务的执行期间，老师处于非激活状态。



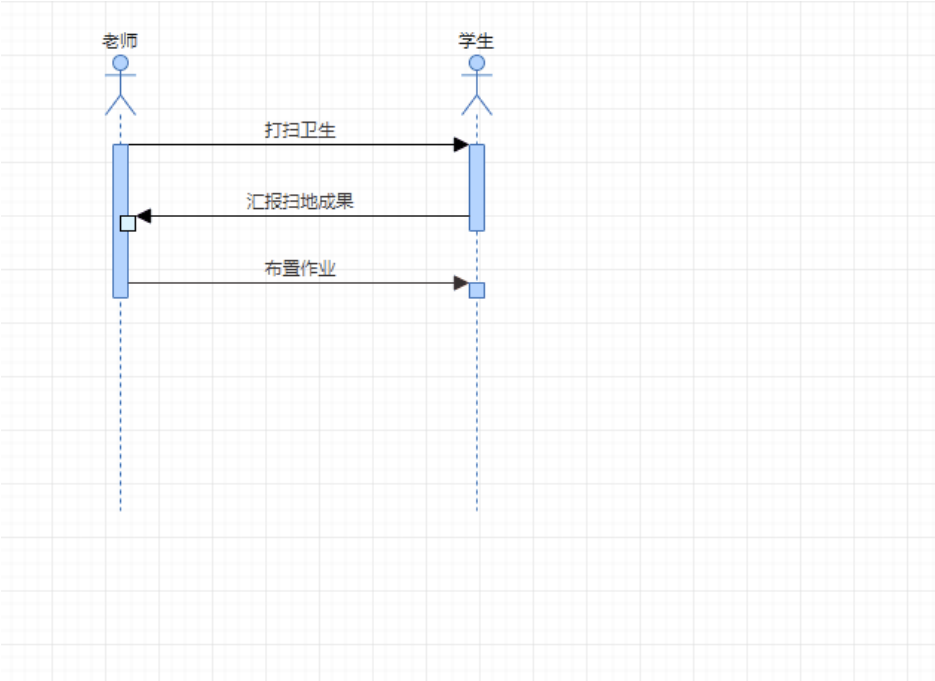
结束目标端激活块

该设置项一般用来断开由“向上延长源端激活块”或“向下延长源端激活块”导致的激活块相连，“延长源端激活块”如果没有显式通过该设置结束，会一直向下延长对应激活块。如下图所示，“对扫地结果给予奖励”任务和“汇报扫地成果”是有关联性的，此时通过对“汇报扫地成果”设置“向下延长源端激活块”可以将其和“对扫地结果给予奖励”的激活块连接到一起。但“下发拖地任务”和上面的扫地任务是并列的独立任务，由于“向下延长源端激活块”的设置影响，它的激活块和上面的连接在一起了，此时就需要对“对扫地结果给予奖励”消息线设置“结束目标端激活块”，取消“延长源端激活块”对“下发拖地任务”的影响。



强制断开目标端

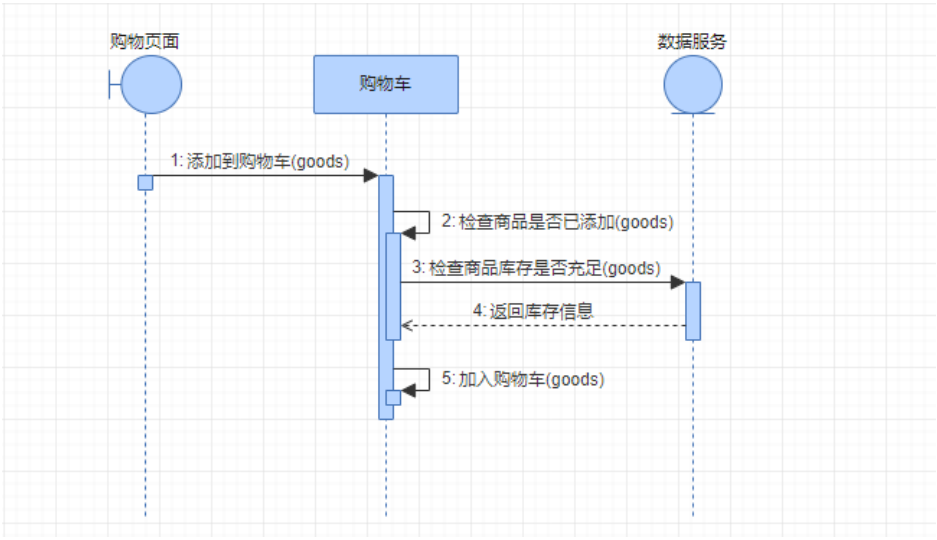
该设置项会强制断开消息线目标端的激活块，使其在目标端与上面的激活块断开。如下图所示，对“汇报扫地成果”任务设置“强制断开目标端”，则该消息线在目标端的激活块会直接断开。



1.4.6 设置消息线层级

顺序图中的消息线是具备层级信息的，通常是该消息线源端激活块在生命线上的层级。如果一条消息线的源端激活块为生命线的直接子激活块，则该消息线的层级为最低的根层级；如果消息线的源端激活块为某个激活块的子激活块，则该消息线的层级就是子激活块对应的层级。对应到软件模型，激活块代表对象中函数的执行周期，层级信息则反映函数的调用关系。

如下图所示，消息线1、消息线2、消息线5都是根层级的消息线，它们的源端为各自生命线下的根级激活块，代表的是这些指令/任务的发送方（调用方）为这些根级激活块对应的函数。而消息线3是子任务（消息线2）中发出的指令，代表消息线3对应的指令/任务是由二级激活块对应的函数调用的。



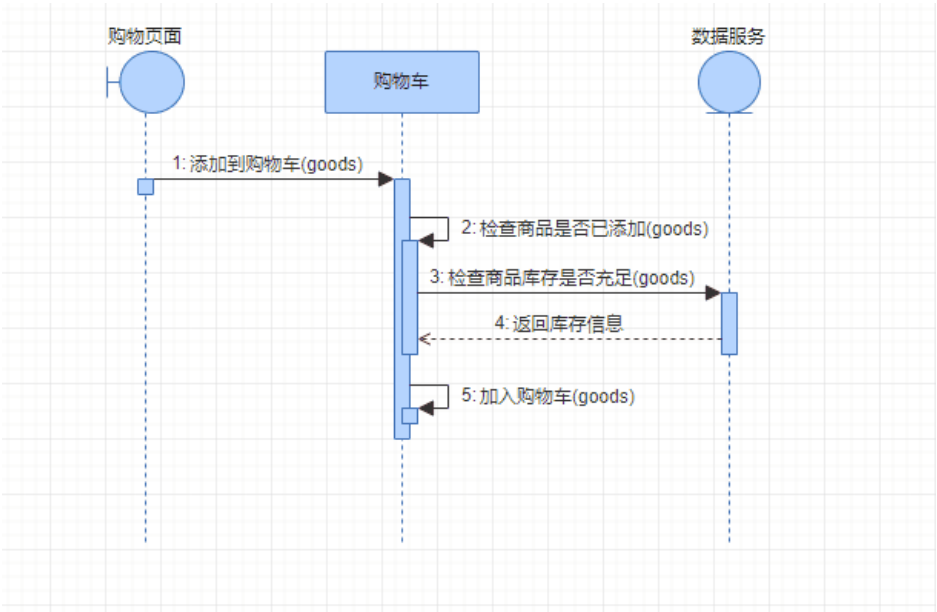
具体示例代码。

```
Complexity is 4 Everything is cool!
class ShoppingCart {
    list = [];

    // 该函数执行周期对应“购物车”生命线上的根级激活块，由“购物页面”对象中的函数触发调用关系
    Complexity is 3 Everything is cool!
    async add(goods) {
        if (!this.checkIfAdded(goods)) {
            return;
        }

        // 对应消息线“5：加入购物车”
        this.addToList();
    }

    // 该函数执行周期对应“购物车”生命线上的二级激活块，由“add”函数（根级激活块）触发调用关系
    Complexity is 4 Everything is cool!
    checkIfAdded(goods) {
        // step1：检测购物车里是否有该物品
        if (this.list.includes(goods)) {
            return false;
        }

        // step2：调用数据服务检测商品库存信息，对应消息线“3：检查商品库存是否充足”
        return this.dataService.checkInventory(goods);
    }
}
```

建模步骤

步骤1 提升消息线层级。

如果消息线源端激活块的上方有子激活块，则可以对该消息线做提升层级操作，将该消息线的发送方设置为上方子激活块。

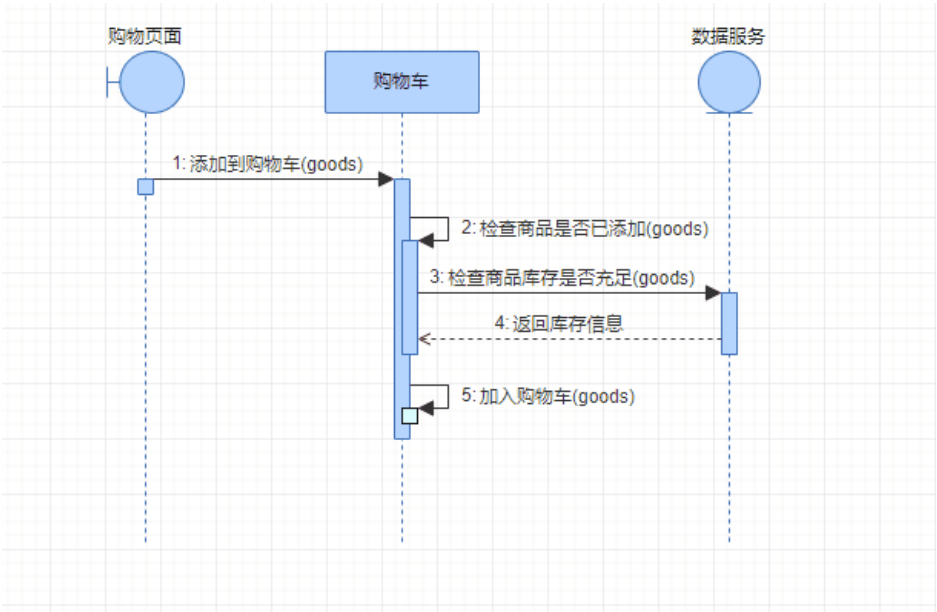


映射到软件代码实现，消息线“5：加入购物车”对应的函数调用方，由消息线“1：添加购物车”变成了消息线“2：检查商品是否已添加”。

```
Complexity is 4 Everything is cool!
class ShoppingCart {
    list = [];

    // 该函数执行周期对应“购物车”生命线上的根级激活块，由“购物页面”对象中的函数触发调用关系
    async add(goods) {
        this.checkIfAdded(goods)
    }

    // 该函数执行周期对应“购物车”生命线上的二级激活块，由“add”函数（根级激活块）触发调用关系
    Complexity is 4 Everything is cool!
    checkIfAdded(goods) {
        // step1: 检测购物车里是否有该物品
        if (this.list.includes(goods)) {
            return false;
        }

        // step2: 调用数据服务检测商品库存信息，对应消息线“3: 检查商品库存是否充足”
        const hasInventory = this.dataService.checkInventory(goods);

        if (hasInventory) {
            // 对应消息线“5: 加入购物车”
            this.addToList();
        }
    }
}
```

步骤2 降低消息线层级。

如果消息线源端激活块不是生命线上的根级激活块，则可以对该消息线做降低层级操作，将该消息线的发送方设置为父激活块。



映射到软件代码实现，消息线“3：检查商品库存是否充足”对应的函数调用方，由消息线“2：检查商品是否已添加”变成了消息线“1：添加到购物车”。

```
Complexity is 4 Everything is cool!
class ShoppingCart {
    list = [];

    // 该函数执行周期对应“购物车”生命线上的根级激活块，由“购物页面”对象中的函数触发调用关系
    Complexity is 4 Everything is cool!
    async add(goods) {
        if (!this.checkIfAdded(goods)) {
            return;
        }

        // 对应消息线“3：检查商品库存是否充足”，调用方由“checkIfAdded”函数 改为 “add”函数
        const hasInventory = this.dataService.checkInventory(goods);

        // 对应消息线“5：加入购物车”
        if (hasInventory) {
            this.addToList();
        }
    }

    // 该函数执行周期对应“购物车”生命线上的二级激活块，由“add”函数（根级激活块）触发调用关系
    checkIfAdded(goods) {
        return this.list.includes(goods);
    }
}
```

----结束

1.4.7 绘制组合片段

Fragment用来对顺序图中的消息发送/接收施加控制，用以将复杂的交互场景分解为更小、更易于管理的部分。每一个Fragment都会有对应的操作符类型，不同的操作符对应着不同的逻辑控制，Fragment中一共有12种操作符类型，可参考下方的操作符介绍说明。

说明

服务将使用频率高的loop、alt元素放到工具箱中直接使用，与通过Fragment设置成loop、alt元素意义相同无区别。

操作符介绍

片段类型	片段名称	说明
alt	抉择	具备alt操作符的Fragment通常表示一组行为选择，且最多只有一个行为被选择。被选择的行为必须有一个明确或隐含的值为true的卫语句表达式，如果没有显式卫语句表达式则表明其是一个隐含的值为true的表达式。 对应到软件模型，alt通常用来表示if...else if...或者switch语句的逻辑执行。
opt	选择	具备opt操作符的Fragment通常表示一个唯一行为是否被选择，它等同于具备一个内容行为和一个空内容行为的alt操作符。 对应到软件模型，alt通常用来表示if 或者 if...else语句的逻辑执行。
break	中断	具备break操作符的Fragment通常表示封闭交互内的一个中断行为，带有卫语句表达式在值为true时，该中断行为会被选择，而封闭交互内的其余交互都不会执行。与opt操作符相比，它多了一个中断后续逻辑执行的能力。该Fragment在绘制时应该包含封闭交互逻辑内的所有生命线。 对应到软件模型，break通常用来表示if...break语句的逻辑执行。
loop	循环	片段重复一定次数，可以在临界中指示片段重复的条件。
par	并行	并行处理。 片段中的事件可以交错。
critical	关键	用在par或seq片段中。指示此片段中的消息不能与其他消息交错。
seq	弱顺序	有两个或更多操作数片段。 涉及同一生命线的消息必须以片段的顺序发生。 如果消息涉及的生命线不同，来自不同片段的消息可能会并行交错。
strict	强顺序	有两个或更多操作数片段。 这些片段必须按给定顺序发生。
consider	考虑	指定此片段描述的消息列表。 其他消息可发生在运行的系统中，但对此描述来说意义不大。
ignore	忽略	此片段未描述的消息列表。 这些消息可发生在运行的系统中，但对此描述来说意义不大。
assert	断言	操作数片段指定唯一有效的序列。 通常用在 consider 或 ignore 片段中。
neg	否定	此片段中显示的序列不会发生。 通常用在 consider 或 ignore 片段中。

建模步骤

步骤1 选择片段元素的操作符类型。

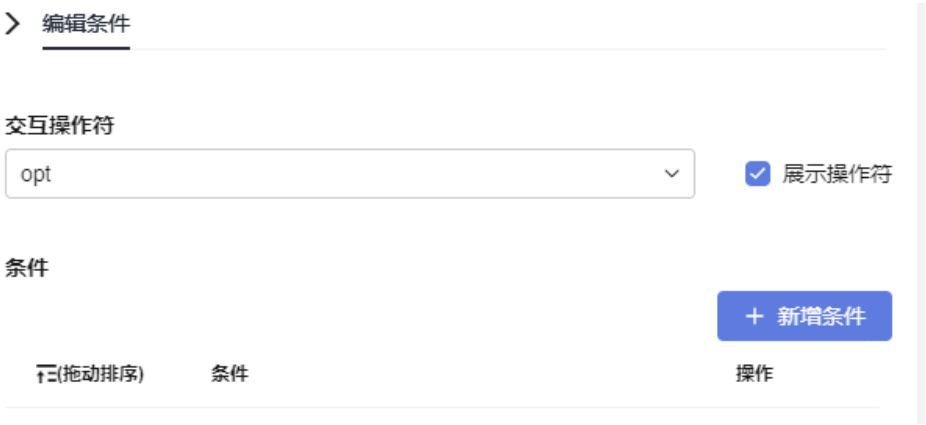
从元素面板拖拽Fragment元素到画布中，会先要求用户选择对应的操作符类型，默认操作符类型是opt。各个操作符对应的含义可参考上方的操作符说明及下方的模型示例。

除了通用Fragment之外，元素面板还额外提供了使用频率较高的loop和alt操作符片段，可以快速拖入使用。

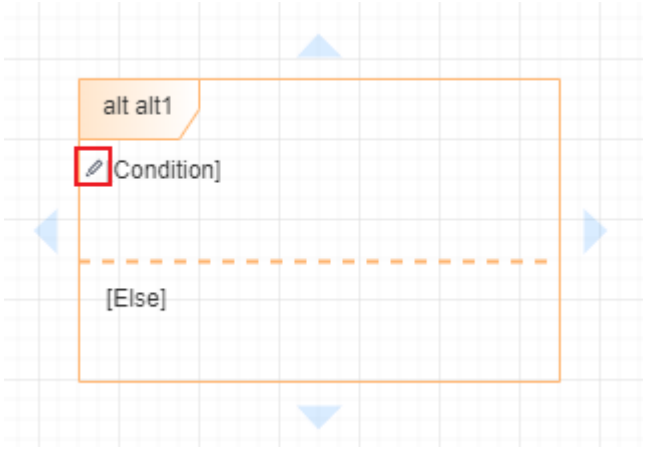
步骤2 打开编辑条件面板。

Fragment元素通常会包含有一定量的条件集，可以在元素上通过单击“右键 > 编辑条件”打开条件编辑面板。

- 通过交互操作符下拉框设置操作符类型，支持设置操作符的显示和隐藏。
- 通过新增条件按钮/删除按钮/单击条件名称等可以新增/删除/修改条件。
- 通过条件列表首列的拖动图标调整各条件顺序。

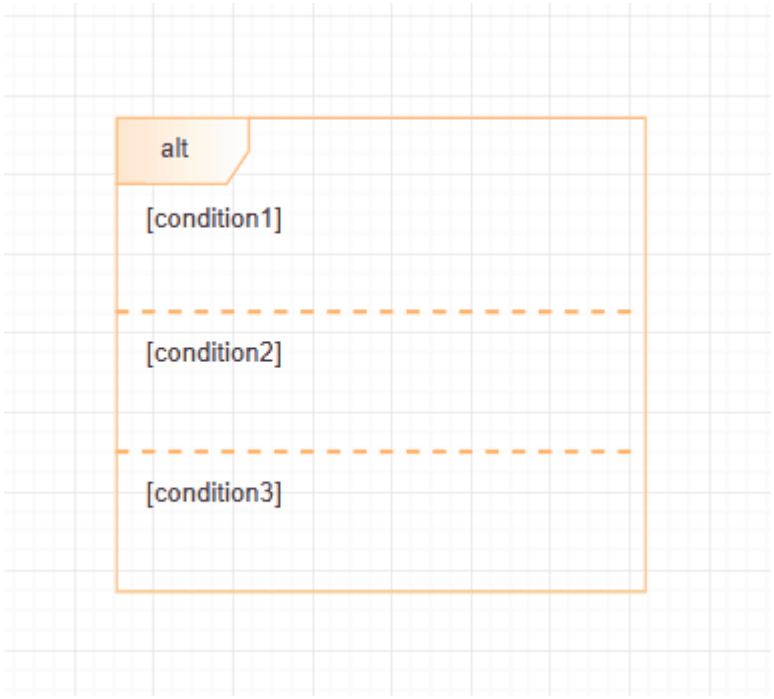


除了右键菜单，也可以通过鼠标悬浮到Fragment元素的条件名称上，单击编辑图标打开条件编辑面板。



步骤3 调整条件区域高度

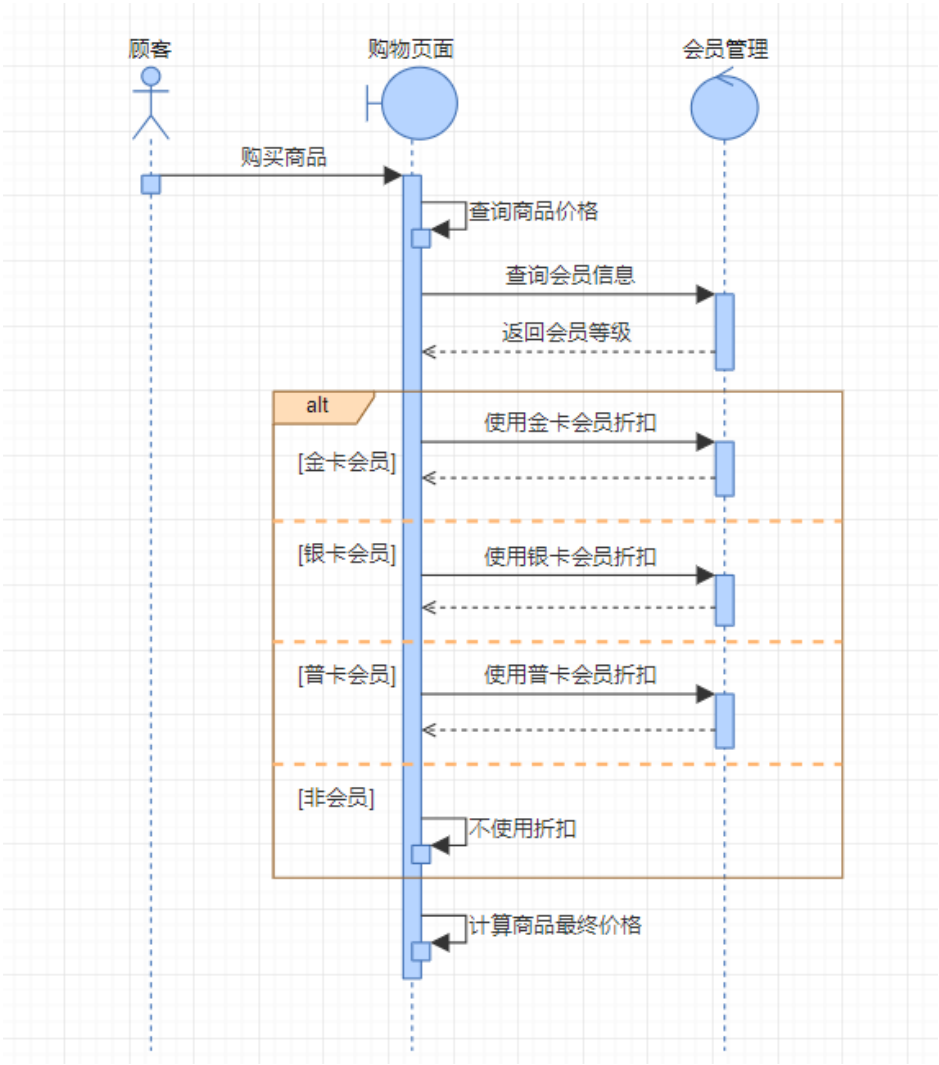
当Fragment元素有多个条件时，在元素内会有多个条件区域，以虚线进行分隔。可以通过拖拽虚线来调整条件区域的高度。



----结束

模型示例

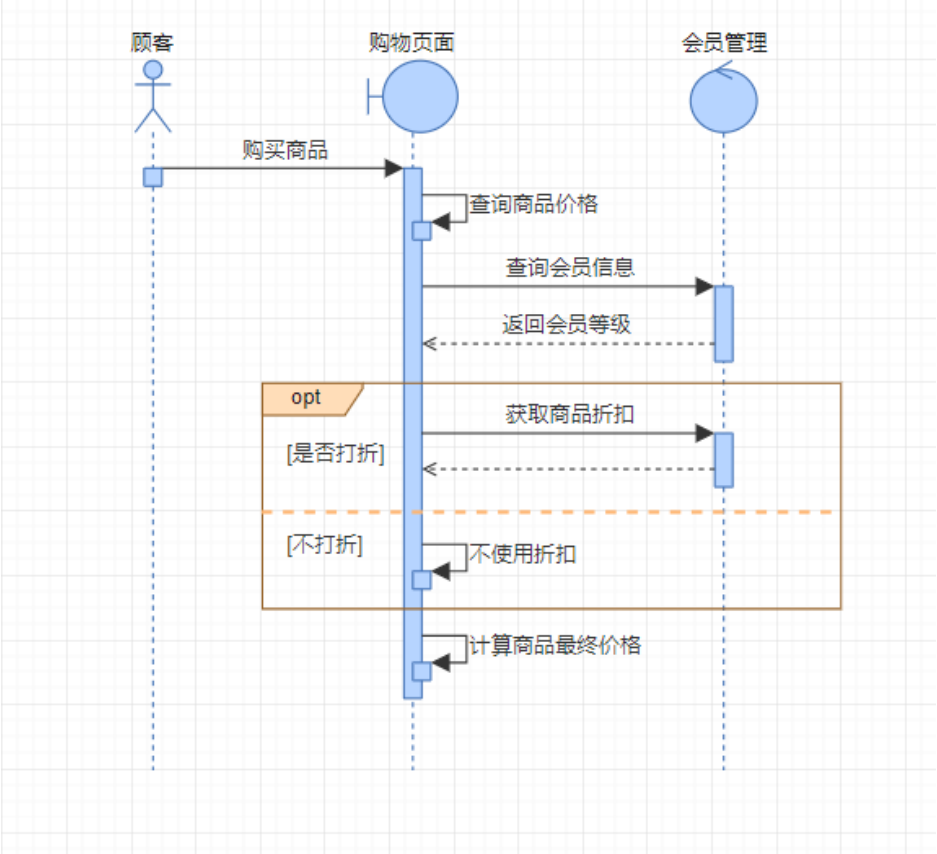
- alt操作符：表示if...else if...或者switch语句的逻辑执行。



对应的代码模型。

```
Complexity is 7 It's time to do something...
class ShoppingPage {
    memberManager = new MemberManager();

    Complexity is 7 It's time to do something...
    buy(goods) {
        let price = this.queryPrice(goods);
        const membership = this.memberManager.queryMembership();
        let discount = 1;

        if (membership.cardType === CARD_TYPE.GOLD_CARD) {
            discount = this.memberManager.queryGoldCardDiscount(membership);
        } else if (membership.cardType === CARD_TYPE.SILVER_CARD) {
            discount = this.memberManager.querySilverCardDiscount(membership);
        } else if (membership.cardType === CARD_TYPE.ORDINARY_CARD) {
            discount = this.memberManager.queryOrdinaryCardDiscount(membership);
        } else {
            discount = 1;
        }

        price = this.getDiscountPrice(price, discount);

        //... 继续执行后续逻辑
    }

    queryPrice(goods) {
        // 执行查询商品价格逻辑
        return 100;
    }

    getDiscountPrice(price, discount) {
        return price * discount;
    }
}

```

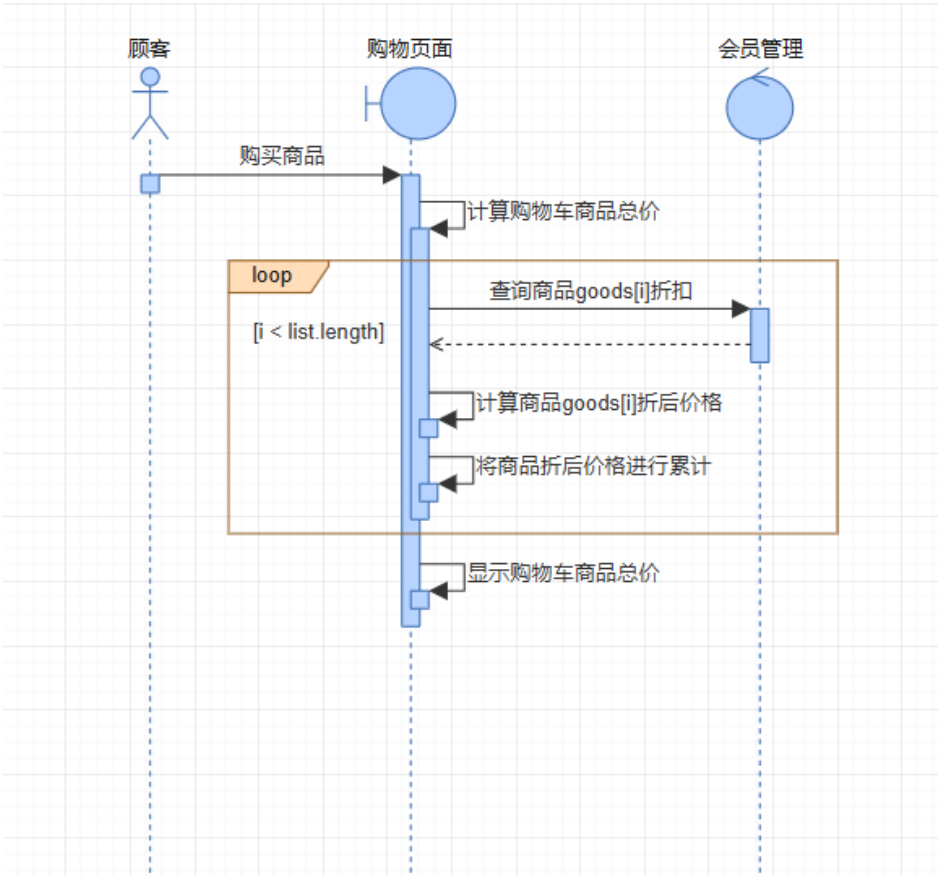
- opt操作符：表示if 或者 if...else语句的逻辑执行。



对应的代码模型。

```
Complexity is 3 Everything is cool!
class ShoppingPage {
    memberManager = new MemberManager();

    Complexity is 3 Everything is cool!
    buy(goods) {
        let price = this.queryPrice(goods);
        let discount = 1;

        if (this.memberManager.hasDiscount(goods)) {
            discount = this.memberManager.queryDiscount(goods);
        } else {
            discount = 1;
        }

        price = this.getDiscountPrice(price, discount);

        //... 继续执行后续逻辑
    }

    queryPrice(goods) {
        // 执行查询商品价格逻辑
        return 100;
    }

    getDiscountPrice(price, discount) {
        return price * discount;
    }
}
```

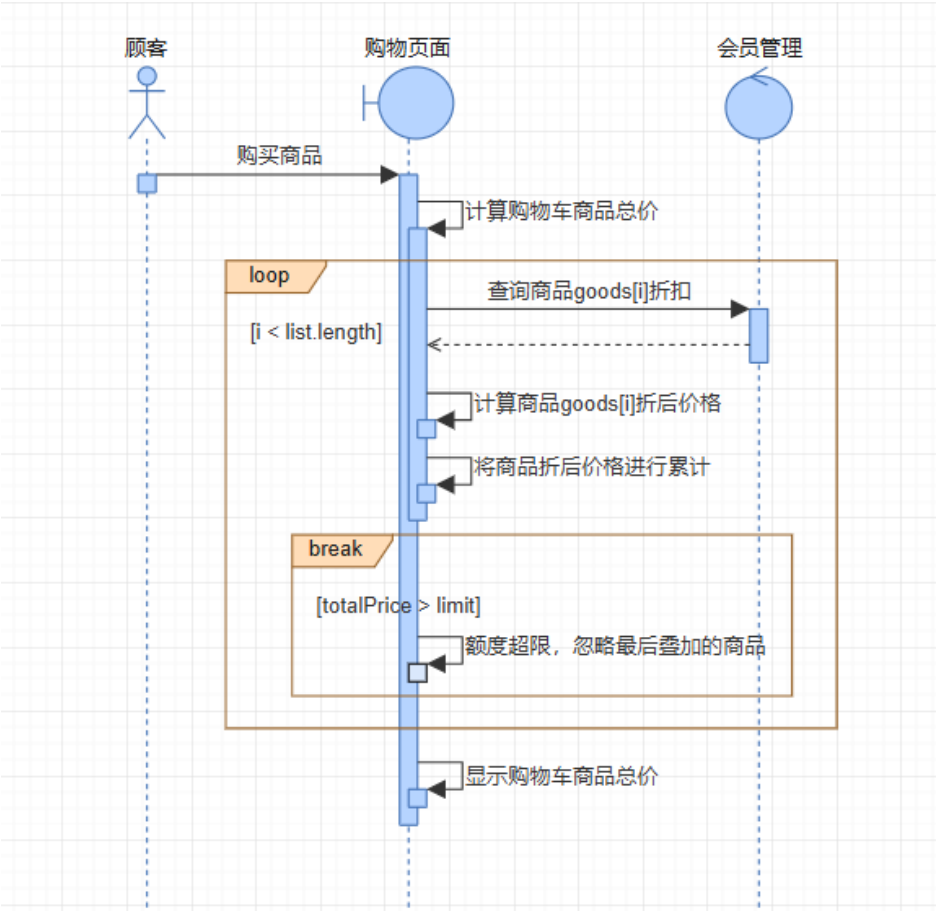
- loop操作符：表示for或者while等循环逻辑的执行。



对应的代码模型。

```
class ShoppingPage {
    memberManager = new MemberManager();
    goodsList = [];
    totalPrice = 0;

    buy() {
        this.calcTotalPrice();
        this.displayTotalPrice();
    }

    calcTotalPrice() {
        // 对购物车中的商品进行遍历，查询每一个商品的折后价格计入总价
        for (let i = 0; i < this.goodsList.length; i++) {
            const goods = this.goodsList[i];
            const discount = this.memberManager.queryDiscount(goods[i]);
            const price = this.getDiscountPrice(goods, discount);

            this.totalPrice += price;
        }
    }

    displayTotalPrice() {
        // 将计算后的totalprice呈现到页面上。
    }

    getDiscountPrice(goods, discount) {
        return goods.price * discount;
    }
}
```

- break操作符：表示对for或者while等循环逻辑的执行中断，通常与loop操作符结合使用。



对应的代码模型。

```
class ShoppingPage {
    memberManager = new MemberManager();
    goodsList = [];
    totalPrice = 0;
    limit = 1000;

    buy() {
        this.calcTotalPrice();
        this.displayTotalPrice();
    }

    Complexity is 4 Everything is cool!
    calcTotalPrice() {
        for (let i = 0; i < this.goodsList.length; i++) {
            const discount = this.memberManager.queryDiscount(this.goodsList[i]);
            const price = this.getDiscountPrice(this.goodsList[i], discount);
            this.totalPrice += price;

            if (this.totalPrice > this.limit) {
                this.totalPrice -= price;
                // 当商品总价超出限额时，停止对后续商品累计价格，并剔除最后一次累计。
                break;
            }
        }
    }

    displayTotalPrice() { }

    getDiscountPrice(goods, discount) { return goods.price * discount; }
}
```

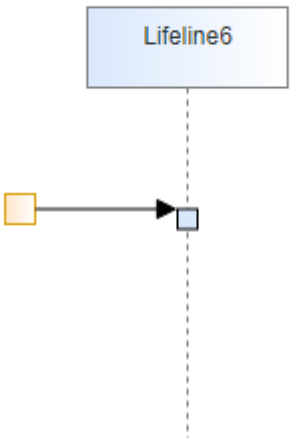
1.4.8 使用 Diagram Gate

图表门是一种简单的图形方式，用于指示可以将消息传输到交互片段和从交互片段传出的点。可能需要一个片段来接收或传递消息。在内部，有序消息反映了这一要求，并在片段帧的边界上指示了门。任何与此内部消息“同步”的外部消息必须适当地对应。可以出现在交互图（顺序，时序，通信或交互概述），交互事件和组合片段（以指定表达式）上。

Diagram Gate并不是激活块（Activation），顺序图在绘制连线时会自动生成激活块，激活块的使用请参考**自定义激活块**。

使用示例

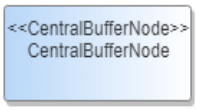
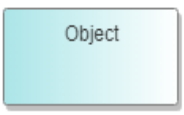
拖动工具箱中的Diagram Gate图标到顺序图中，然后根据需要输入名称即可。



1.5 活动图

活动图对用户和系统遵循流程的行为进行建模，它们是流程图或工作流的一种，但是它们使用的形状略有不同，元素介绍如下表所示：

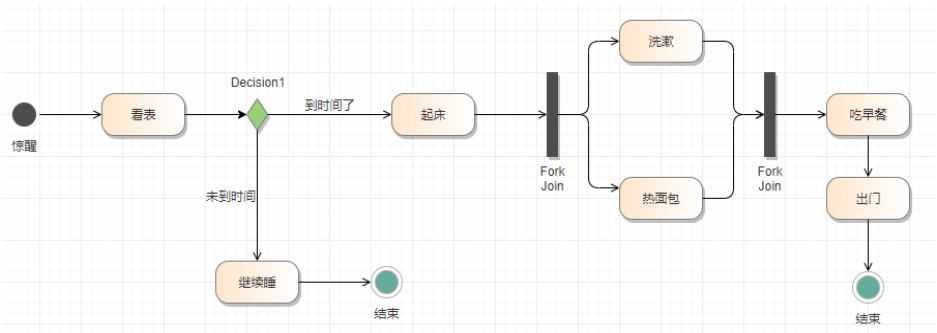
表 1-3 活动图元素介绍

元素名	图标	含义
Action		动作是可执行的原子计算，它导致模型状态的改变和返回值。
Activity		活动是状态机内正在进行的非原子执行。
StructuredActivity		结构化活动是一个活动节点，可以将下级节点作为独立的活动组。
CentralBufferNode		中央缓冲区节点是一个对象节点，用于管理活动图中表示的来自多个源和目标的流。
Datastore		数据存储区定义了永久存储的数据。
ExceptionHandlerNode		异常处理程序元素定义发生异常时要执行的一组操作。
Object		封装了状态和行为的具有良好定义界面和身份的离散实体，即对象实例。
Decision		是状态机中的一个元素，在它当中一个独立的触发可能导致多个可能结果，每个结果有它自己的监护条件。
Merge		状态机中的一个位置，两个或多个可选的控制路径在此汇合或"无分支"。

元素名	图标	含义
Send		即发送者对象生成一个信号实例并把它传送到接收者对象以传送信息。
Receive		接收就是处理从发送者传送过来的消息实例。
Partition		分区元素用于逻辑组织活动的元素。
Partition		分区元素用于逻辑组织活动的元素。
Initial		用来指明其默认起始位置的伪状态。
Final		组成状态中的一个特殊状态，当它处于活动时，说明组成状态已经执行完成。
Flow Final		Flow Final元素描述了系统的退出，与Activity Final相反，后者代表Activity的完成。
Synch		一个特殊的状态，它可以实现在一个状态机里的两个并发区域之间的控制同步。

元素名	图标	含义
Fork Join		Fork，复杂转换中，一个源状态可以转入多个目标状态，使活动状态的数目增加。 Join，状态机活动图或顺序图中的一个位置，在此处有两个或以上并列线程或状态归结为一个线程或状态。
Fork Join		Fork，复杂转换中，一个源状态可以转入多个目标状态，使活动状态的数目增加。 Join，状态机活动图或顺序图中的一个位置，在此处有两个或以上并列线程或状态归结为一个线程或状态。
Region		并发区域。
Control Flow		（控制流）在交互中，控制的后继轨迹之间的关系。
Object Flow		（对象流）各种控制流表示了对象间的关系、对象和产生它（作输出）或使用它（作输入）的操作或转换间的关系。
Constraint		是一个语义条件或者限制的表达式。UML 预定义了某些约束，其他可以由建模者自行定义。
Exception Handler		异常处理。捕获异常根据异常类型查找到对应的异常处理方法，然后执行对应的方法。
Interrupt Flow		中断流是用于定义异常处理程序和可中断活动区域的连接器的两个UML概念的连接。中断流是活动边缘的一种。它通常用于活动图中，以对活动过渡进行建模。
Anchor		锚点。
Containment		内嵌，表示嵌在内部的类。

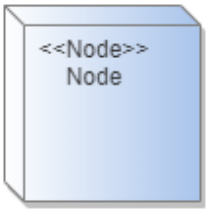
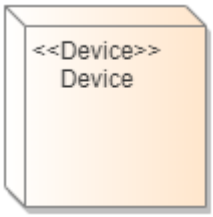
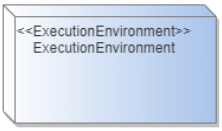
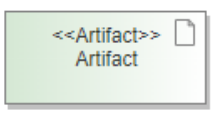
活动图示例如下所示：

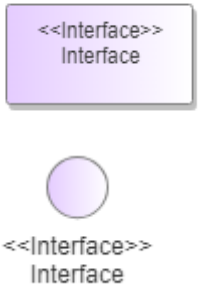
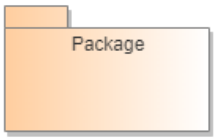
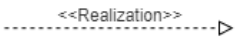
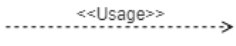
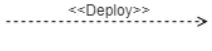


1.6 部署图

部署图用于大型和复杂系统的另一张专门图，其中软件部署在多个系统上，元素介绍如下表所示：

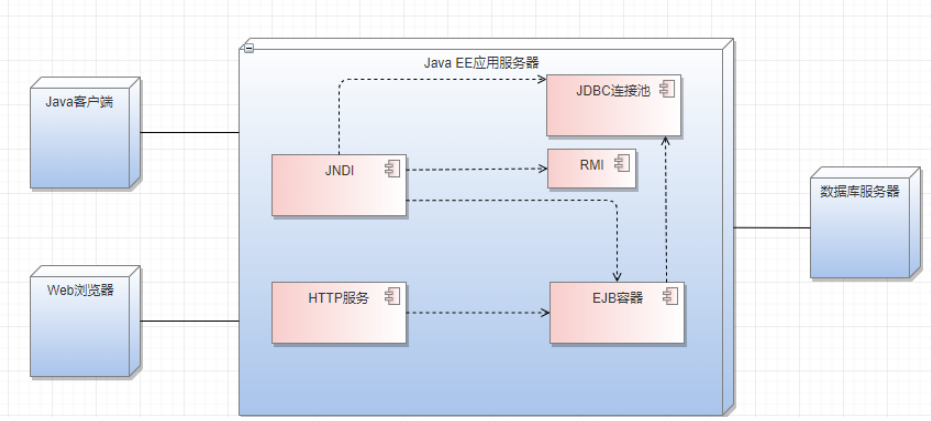
表 1-4 部署图元素介绍

元素名	图标	含义
Node		部署节点。
Device		设备节点。
DeploymentSpecification		部署规格。
ExecutionEnvironment		执行环境。
Artifact		制品是被软件开发过程所利用或通过软件开发过程所生产的一段信息，如外部文档或工作产物。 制品可以是一个模型、描述或软件。

元素名	图标	含义
Component		组件，可独立加载、部署和运行的二进制代码，采用轻量级通讯机制、松耦合高内聚的软件架构构建单元，部署时不能跨节点类型部署（计算机百科全书：组件是软件系统中具有相对独立功能、接口由契约指定、和语境有明显依赖关系、可独立部署、可组装的软件实体）。
Interface		接口，可以是单个接口，也可以是抽象的一组接口的组合。 圆形接口与矩形接口意义相同，仅形状不同。
Package		包。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
Realization		实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Usage		使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。
Generalization		通用化，是一种继承关系，一个类（通用元素）的所有信息（属性或操作）能被另一个类（具体元素）继承，不仅可以有属于类自己的信息，而且还拥有被继承类的信息。
Manifest		Repo和对应的逻辑设计对象使用"Manifest”连接 表示由此代码仓的代码实现此设计对象的功能。
Deployment		描述现实世界环境运行系统的配置的开发步骤。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

部署图一般用于：

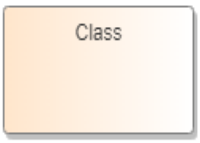
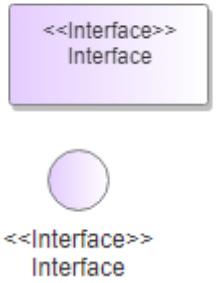
1. 嵌入式系统建模（硬件之间的交互）。
2. 客户端/服务器系统建模（用户界面与数据的分离）。
3. 分布式系统建模（多级服务器）。

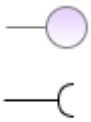
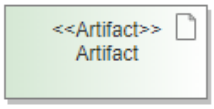
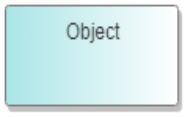
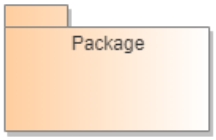
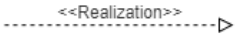
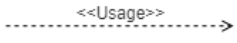


1.7 组件图

组件图显示了复杂软件系统中的各个组件如何相互关联以及如何使用接口进行通信。它们不用于更简单或更直接的系统，元素介绍如下表所示：

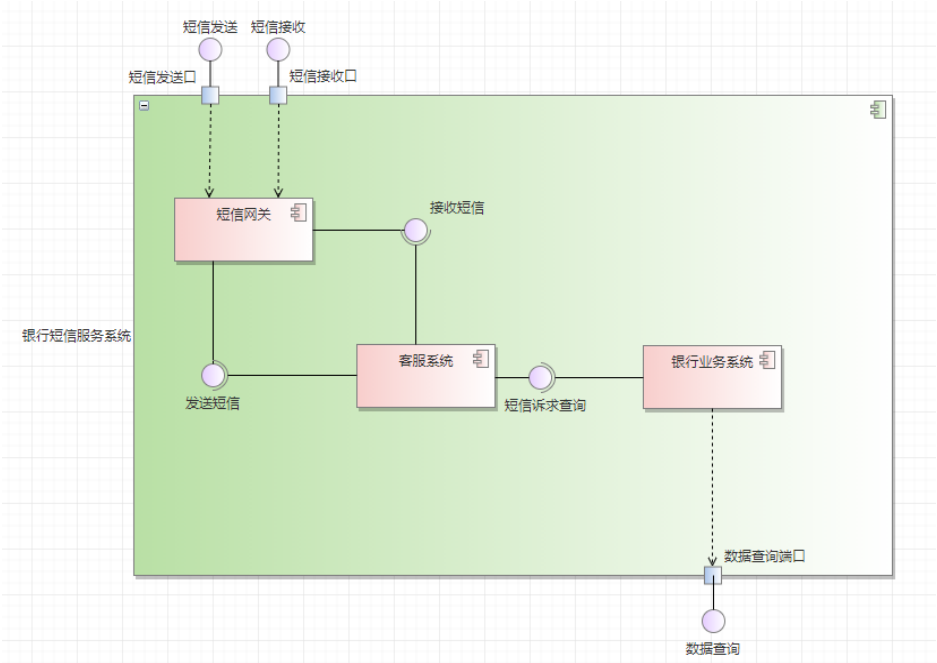
表 1-5 组件图元素介绍

元素名	图标	含义
Class		是对象的集合，展示了对象的结构以及与系统的交互行为。
Interface		接口，可以是单个接口，也可以是抽象的一组接口的组合。 圆形接口与矩形接口意义相同，仅形状不同。
Component		组件，可独立加载、部署和运行的二进制代码，采用轻量级通讯机制、松耦合高内聚的软件架构构建单元，部署时不能跨节点类型部署（计算机百科全书：组件是软件系统中具有相对独立功能、接口由契约指定、和语境有明显依赖关系、可独立部署、可组装的软件实体）。

元素名	图标	含义
Interface		Required Interface和Provided Interface之间可以建立Dependency，表明一个组件需要的接口是由另外一个组件提供的。
port		端口
PackagingComponent		包装组件。
Artifact		制品。
Document Artifact		文档制品。
Object		对象。
Package		包。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Realization		实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Usage		使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。
Constraint		是一个语义条件或者限制的表达式。UML 预定义了某些约束，其他可以由建模者自行定义。

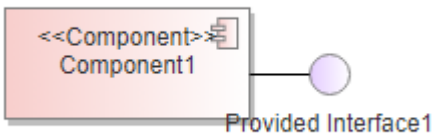
元素名	图标	含义
Anchor		锚点。
Containment		内嵌，表示嵌在内部的类。
Generalization		泛化，表示类与类、接口与接口之间的继承关系，由子一方指向父对象一方。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

组件图示例如下所示：

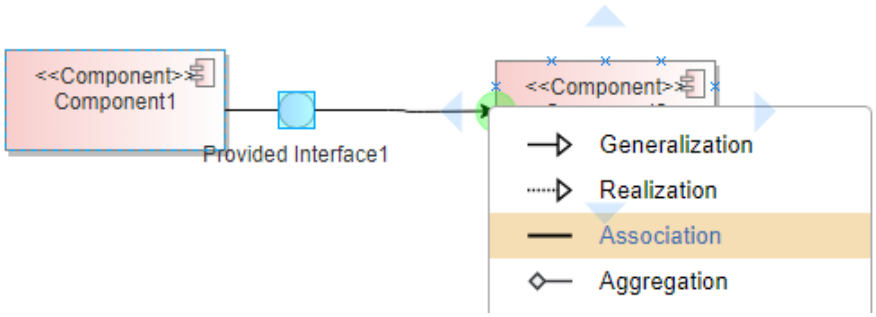


在画暴露接口与请求接口时，可以通过Association关联连线将两种接口合并。

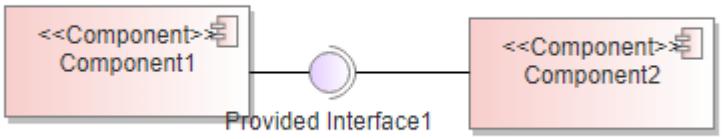
步骤1 在左侧工具箱中中选择“Provided Interface”，将其拖拽至需要连接的图形上。



步骤2 拖拽完成松开左键，在弹出的连线选择列表中选择“Association”关联连线。



步骤3 松开鼠标后即可形成接口合并。



----结束

1.8 状态机图

状态机图元素介绍如下表所示：

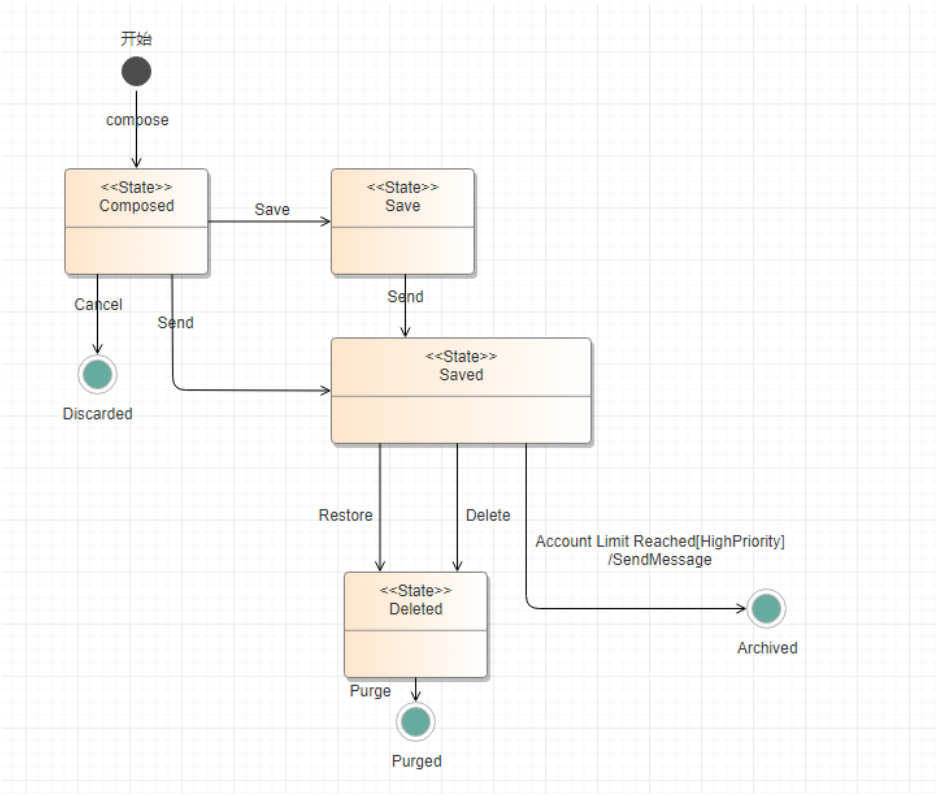
表 1-6 状态机图元素介绍

元素名	图标	含义
State	<<State>> State	对象的生命中的满足一定条件，执行一定操作，或者等待某事件的条件或者情况。
StateMachine	<<StateMachine>> StateMachine	状态机是展示状态与状态转换的图。通常一个状态机依附于一个类，并且描述一个类的实例对接收到的事件所发生的反应。
Fork Join	Fork Join	Fork，复杂转换中，一个源状态可以转入多个目标状态，使活动状态的数目增加。 Join，状态机活动图或顺序图中的一个位置，在此处有两个或以上并列线程或状态归结为一个线程或状态。

元素名	图标	含义
Fork Join		Fork，复杂转换中，一个源状态可以转入多个目标状态，使活动状态的数目增加。 Join，状态机活动图或顺序图中的一个位置，在此处有两个或以上并列线程或状态归结为一个线程或状态。
Initial		用来指明其默认起始位置的伪状态。
Junction		结合状态，作为一个综合转换一部分的伪状态，它在转换执行中不打断运行至完成步骤。
Deep History		历史状态可以记忆浅历史和深历史。深历史状态记忆组成状态中更深的嵌套层次的状态。要记忆深状态，转换必须直接从深状态中转出。
Shallow History		浅历史状态保存并激活与历史状态在同一个嵌套层次上的状态。
EntryPoint		进入某一状态时执行的动作
ExitPoint		离开某一状态时执行的动作。
Final		组成状态中的一个特殊状态，当它处于活动时，说明组成状态已经执行完成。

元素名	图标	含义
Flow Final		Flow Final元素描述了系统的退出，与Activity Final相反，后者代表Activity的完成。
Synch		一个特殊的状态，它可以实现在一个状态机里的两个并发区域之间的控制同步。
Choice		选择，代表多个路径选择。
Terminate		终止。
Transition		转换用实线箭头表示，从一个状态（源状态）到另一个状态（目标状态），用一条转换线标注。
Object flow		各种控制流表示了对象间的关系、对象和产生它（作输出）或使用它（作输入）的操作或转换间的关系。

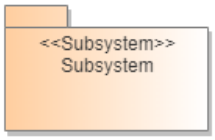
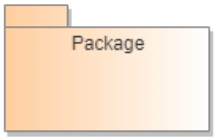
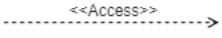
状态机图示例，如下图所示：



1.9 包图

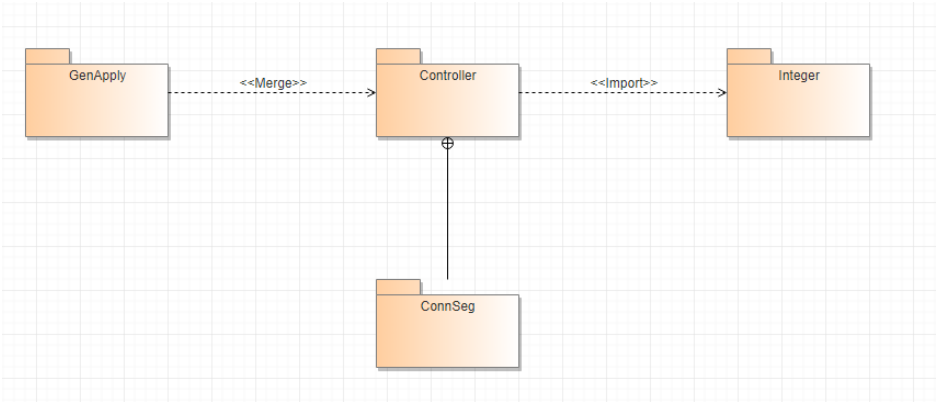
包图元素介绍如下表所示：

表 1-7 包图元素介绍

元素名	图标	含义
Subsystem		作为且有规范、实现和身份的单元的包。
Package		包。
Access		访问依赖关系用一个从客户包指向提供者包的虚箭头表示。
Merge		合并连接器，定义了源包元素与目标包同名元素之间的泛化关系。源包元素的定义被扩展来包含目标包元素定义。当源包元素与目标包内没有同名元素时，目标包元素的定义不受影响。

元素名	图标	含义
Import	<<Import>>	用虚线箭头从得到访问权限的包指向提供者所在的包。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。 依赖关系用两个模型元素之间的虚线箭头表示。箭尾处的模型元素（客户）依赖于箭头处的模型元素（服务者）。

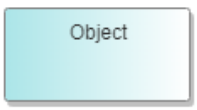
包图示例，如下图所示：



1.10 对象图

对象图元素介绍如下表所示：

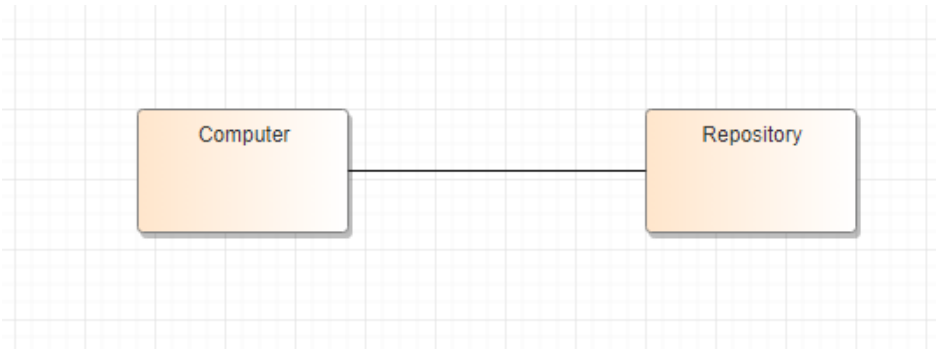
表 1-8 对象图元素介绍

元素名	图标	含义
Actor		角色，是与系统交互的人或事物。
Object		封装了状态和行为的具有良好定义界面和身份的离散实体；即对象实例。

元素名	图标	含义
Collaboration		是对对象和链总体安排的一个描述，这些对象和链在上下文中通过互操作完成一个行为，例如一个用例或者操作。
CollaborationUse		使用协作用于在复合结构图中将协作定义的模式应用于特定情况。
Association Node		关联节点。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

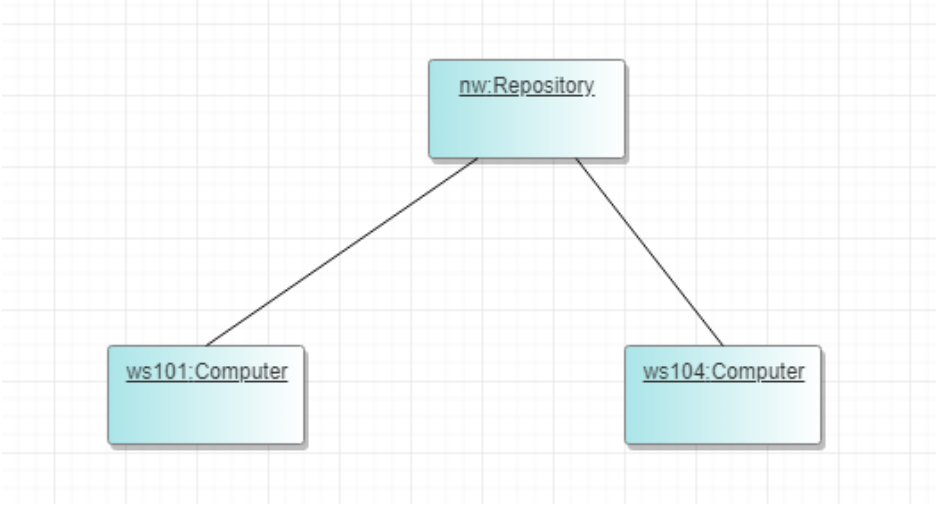
对象图示例，如下图所示：

示例首先显示简单的类图上连接的两个类元素。



上面的类被实例化作为下面对象图中的对象。在此模型中有的两个实例计算机，这可以证明在实践中用类对象作为考虑类之间的关系和相互作用是有用的。

在对象图中添加的Object对象元素，通过右键菜单“元素设置 > 设置源元素”，可以设置对象元素的基于上面的类图中的类元素实例化出来的对象。

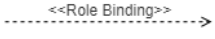
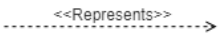


1.11 组合结构图

组合结构图元素介绍如下表所示：

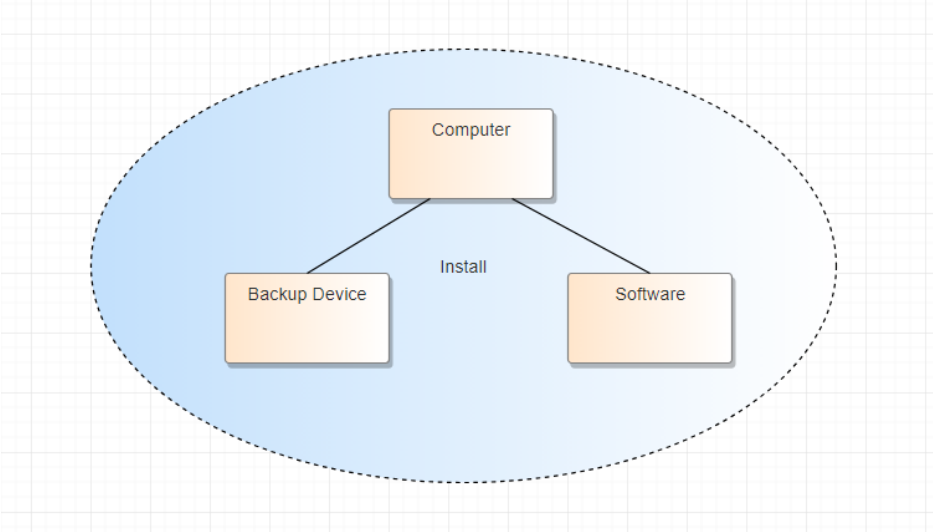
表 1-9 组合结构图元素介绍

元素名	图标	含义
Class	Class	是对象的集合，展示了对象的结构以及与系统的交互行为。
Interface	<<Interface>> Interface <<Interface>> Interface	接口，可以是单个接口，也可以是抽象的一组接口的组合。 圆形接口与矩形接口意义相同，仅形状不同。
Property	Property	特性就是表示传递有关模型元素信息的值的一般性术语。属性具有语义效果，在 UML中一部分属性已经事先定义好了，其他的特性是用户定义的。
Port	Port	端口定义了分类器与其环境之间的交互。

元素名	图标	含义
Collaboration		协作定义了一组协作角色及其连接器。
CollaborationUse		使用协作用于在复合结构图中将协作定义的模式应用于特定情况。
Provided Interface Required Interface		Required Interface和Provided Interface之间可以建立Dependency，表明一个组件需要的接口是由另外一个组件提供的。
Connector		连接器通常在“组合结构”图中说明零件之间的通信链接以实现结构的目的。
Delegate		委托连接器在组件图上定义了组件外部端口和接口的内部组件。
Role Binding		角色绑定是协作使用的内部角色和实现特定情况所需的各个部分之间的映射，通常在复合结构图中。
Ocurrence		在组合结构图中，发生关系表示协作表示分类器。
Represents		表示连接器指示在分类器（通常在“组合结构”图中）中使用了协作。

组合结构图示例，如下图所示：

此图显示了协作在组合结构图中显示执行安装的关系的图

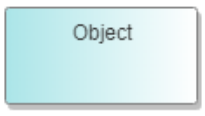
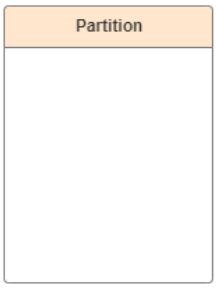


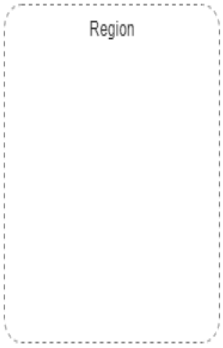
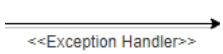
1.12 交互概述图

交互概述图元素介绍如下表所示：

表 1-10 交互概述图元素介绍

元素名	图标	含义
Action		动作是可执行的原子计算，它导致模型状态的改变和返回值。
Activity		活动是状态机内正在进行的非原子执行。
StructuredActivity		结构化活动是一个活动节点，可以将下级节点作为独立的活动组。
CentralBufferNode		中央缓冲区节点是一个对象节点，用于管理活动图中表示的来自多个源和目标的流。
Datastore		数据存储区定义了永久存储的数据。

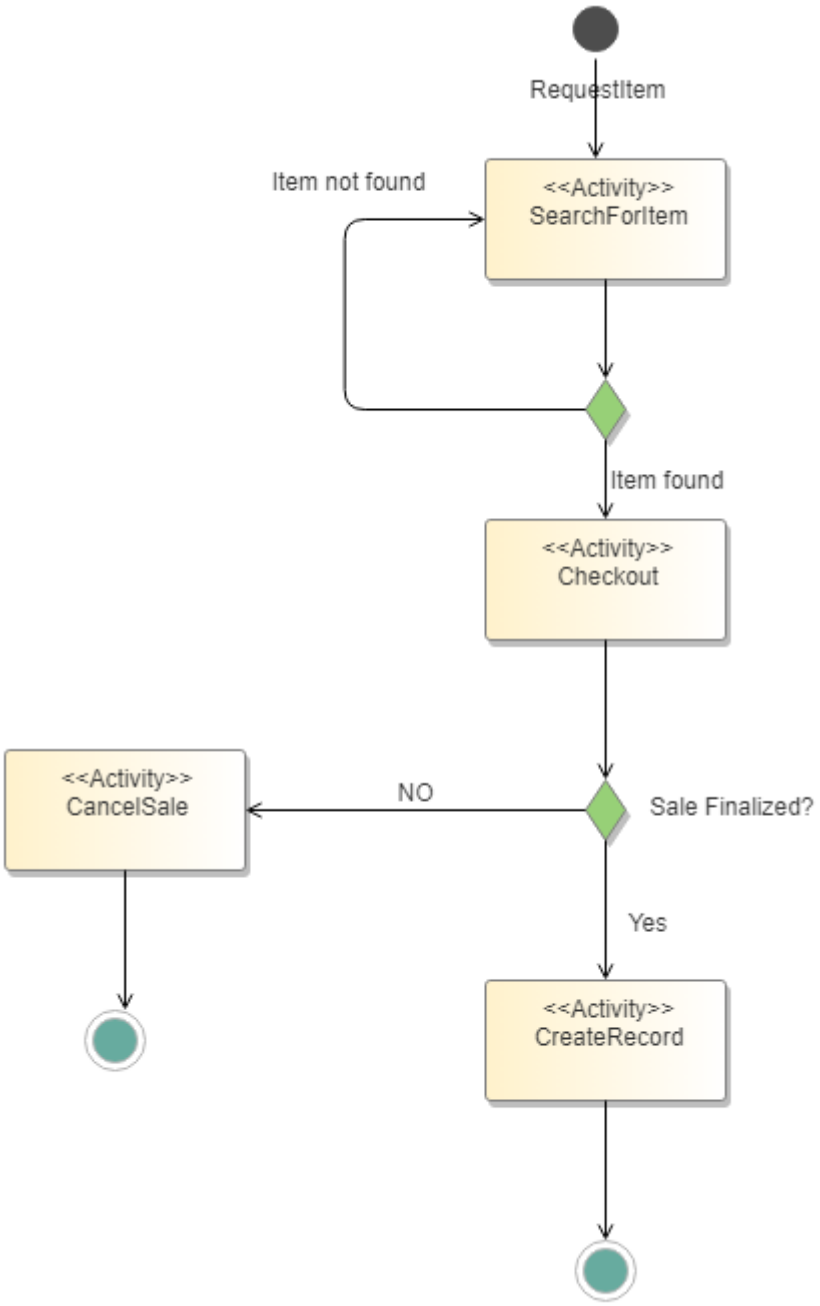
元素名	图标	含义
Object		封装了状态和行为的具有良好定义界面和身份的离散实体；即对象实例。
Decision		是状态机中的一个元素，在它当中一个独立的触发可能导致多个可能结果，每个结果有它自己的监护条件。
Merge		状态机中的一个位置，两个或多个可选的控制路径在此汇合或"无分支"。
Send		发送者对象生成一个信号实例并把它传送到接收者对象以传送信息。
Receive		接收就是处理从发送者传送过来的消息实例。
Partition		分区元素用于逻辑组织元素。
Partition		分区元素用于逻辑组织元素。
Initial		用来指明其默认起始位置的伪状态。
Final		组成状态中的一个特殊状态，当它处于活动时，说明组成状态已经执行完成。

元素名	图标	含义
Flow Final		Flow Final元素描述了系统的退出，与Activity Final相反，后者代表Activity的完成。
Synch		一个特殊的状态，它可以实现在一个状态机里的两个并发区域之间的控制同步。
Fork Join		Fork，复杂转换中，一个源状态可以转入多个目标状态，使活动状态的数目增加。 Join，状态机活动图或顺序图中的一个位置，在此处有两个或以上并列线程或状态归结为一个线程或状态。
Fork Join		Fork，复杂转换中，一个源状态可以转入多个目标状态，使活动状态的数目增加。 Join，状态机活动图或顺序图中的一个位置，在此处有两个或以上并列线程或状态归结为一个线程或状态。
Region		并发区域。
Exception Handler		异常处理。捕获异常根据异常类型查找到对应的异常处理方法，然后执行对应的方法。
Control Flow		（控制流）在交互中，控制的后继轨迹之间的关系。
Object Flow		（对象流）各种控制流表示了对象间的关系、对象和产生它（作输出）或使用它（作输入）的操作或转换间的关系。

元素名	图标	含义
Interrupt Flow		中断流是用于定义异常处理程序和可中断活动区域的连接器的两个UML概念的连接。

交互概述图示例，如下图所示：

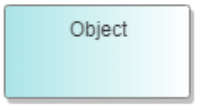
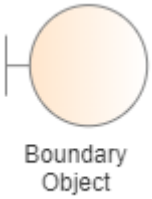
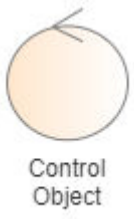
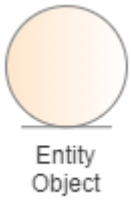
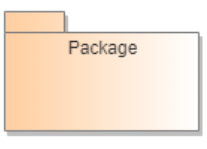
此图描绘了一个示例销售过程，在示例中显示了一个交互销售过程，各活动对象都可以有独立的子交互概述图。



1.13 通信图

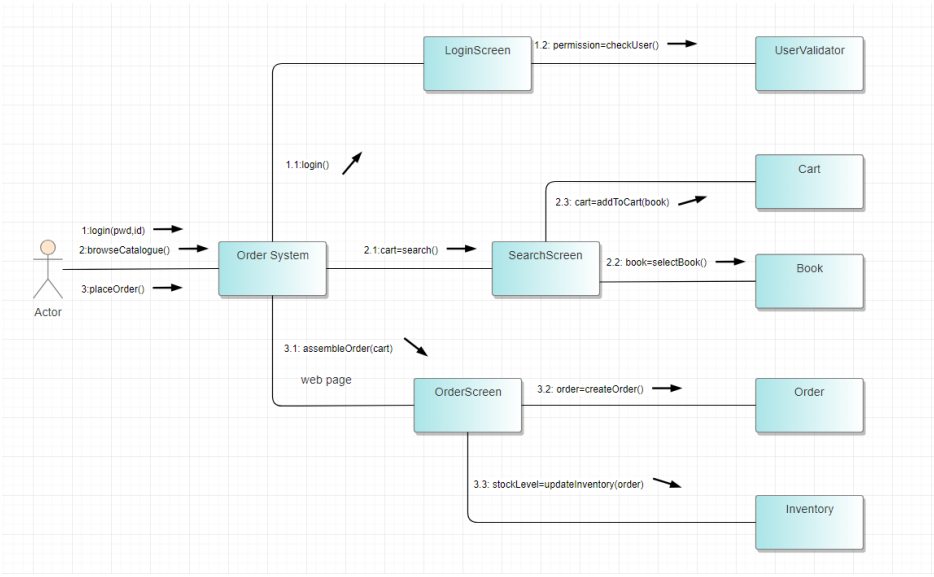
通信图元素介绍如下表所示：

表 1-11 通信图元素介绍

元素名	图标	含义
Actor		角色，是与系统交互的人或事物。
Object		封装了状态和行为的具有良好定义界面和身份的离散实体；即对象实例。
Boundary Object		边界对象。
Control Object		控制对象。
Entity Object		实体对象。
Package		包。对元素进行分组，并为分组的元素提供名称空间。一个程序包可能包含其他程序包，从而提供程序包的分层组织。
Nesting		嵌套，即一个类的嵌套到另一个类。

元素名	图标	含义
Realization	<<Realization>>	实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

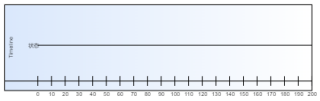
通信图示例，如下图所示：



1.14 时间图

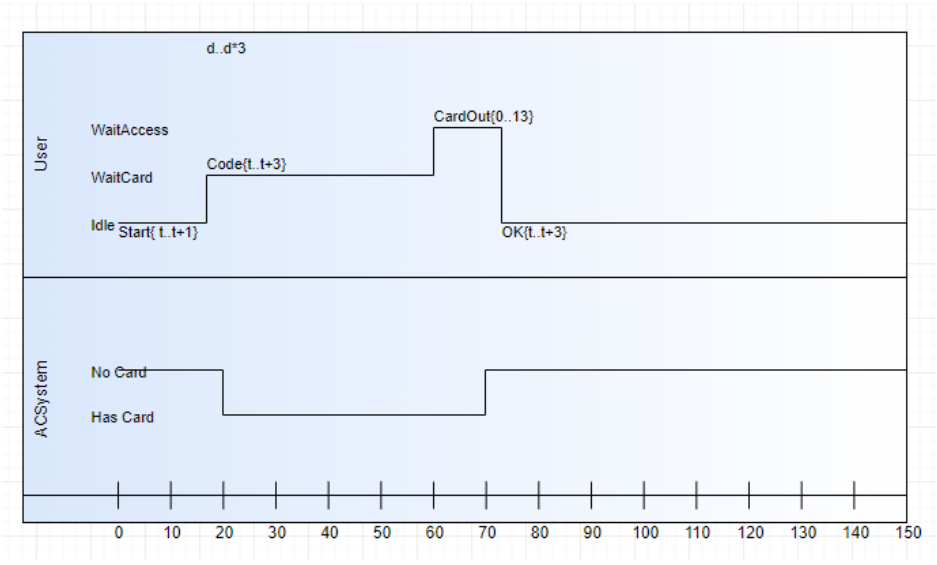
时间图（Timing Diagram）用于详细描述系统中对象的状态随时间变化的情况。

元素介绍

元素名	图标	含义
Timeline		时间生命线，代表一个对象，x轴表示时间，y轴表示离散状态。 多个时间生命线可上下叠加，共用最下层的时间轴。 不支持Link和Instance方式引用。

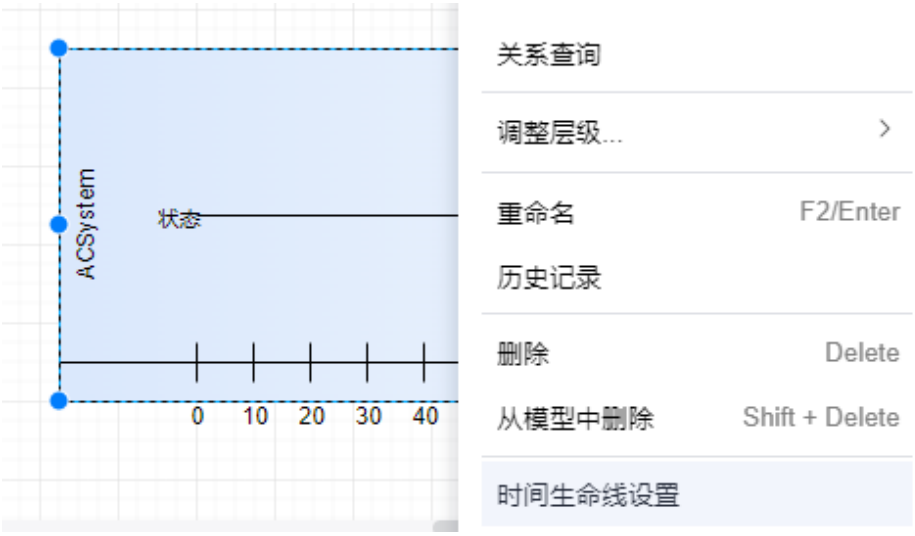
建模步骤

以门禁系统为例展示时间生命线的状态过程。

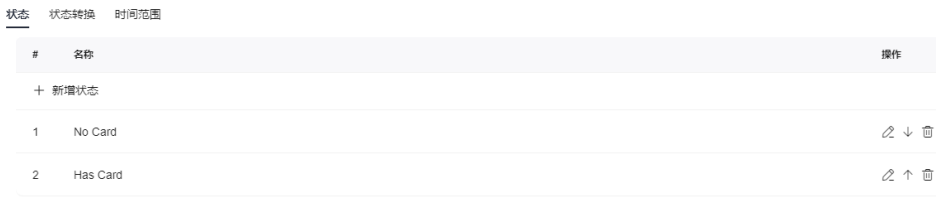


步骤1 从左侧工具箱拖动Timeline元素至画布中并命名，如：ACSystem，选中ACSystem右键“时间生命线设置”支持修改生命线的状态、状态转换和时间范围。

- 状态：对象状态变化场景，支持通过右侧上下移动箭头调整状态顺序，状态变化线对应改变，上限个数100。
- 状态转换：配置状态转换节点，表示生命线转换到什么状态，上限个数100。
- 时间范围：只有最下层时间线支持配置，包括起始时间、结束时间、刻度间距。新建的时间轴默认单位为秒，起始时间默认0，结束时间默认为200。



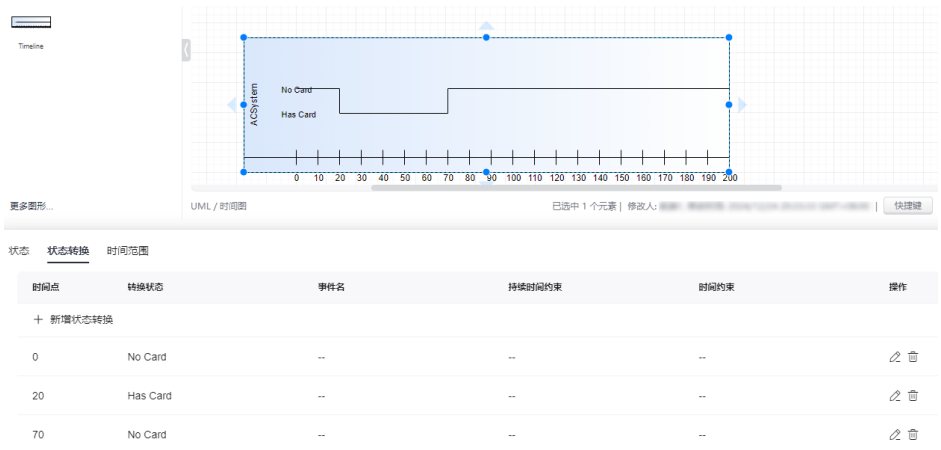
步骤2 修改默认状态名称为No Card并单击“新建状态”添加名称为Has Card的状态。



步骤3 切换到状态转换页签，初始状态No Card默认从时间点0开始，不可修改。

- 时间点：必填。表示目标状态的转换时间，这个时间只能在时间轴表示的时间范围内。
- 转换状态：必填。表示转换到什么状态，从状态列表中选取，严格按照时间点从小到大排序，上下不允许相同。
- 事件名：非必填，触发转换的事件。
- 持续时间约束：非必填。描述状态持续的时间长度。如[>5ms]表示事件的持续时间必须大于5毫秒。
- 时间约束：非必填。描述事件发生的具体时间。如[> 10ms after EventB]表示事件必须在事件B发生后的至少10毫秒后发生。

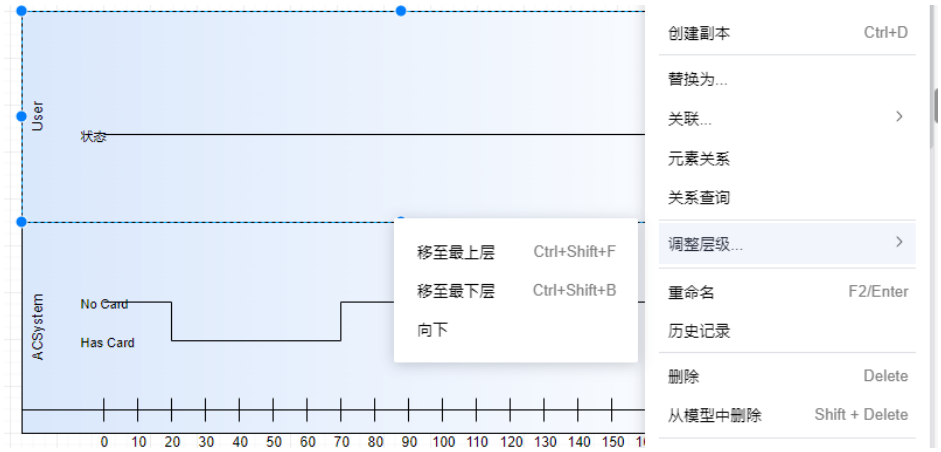
单击新增状态转换按图中从上到下的顺序新增两个转换状态，此时图中时间线在对应的时间点发生转折，表示整个对象由No Card转换到Has Card再转换到No Card的状态过程。



步骤4 同样方式建立名为User的时间生命线。

拖入多个时间生命线根据拖入位置自动叠放到已存在的时间生命线上，可通过“右键菜单 > 调整层级 > 向上/向下”手动调整顺序。

拖动单个时间生命线整体跟随移动。

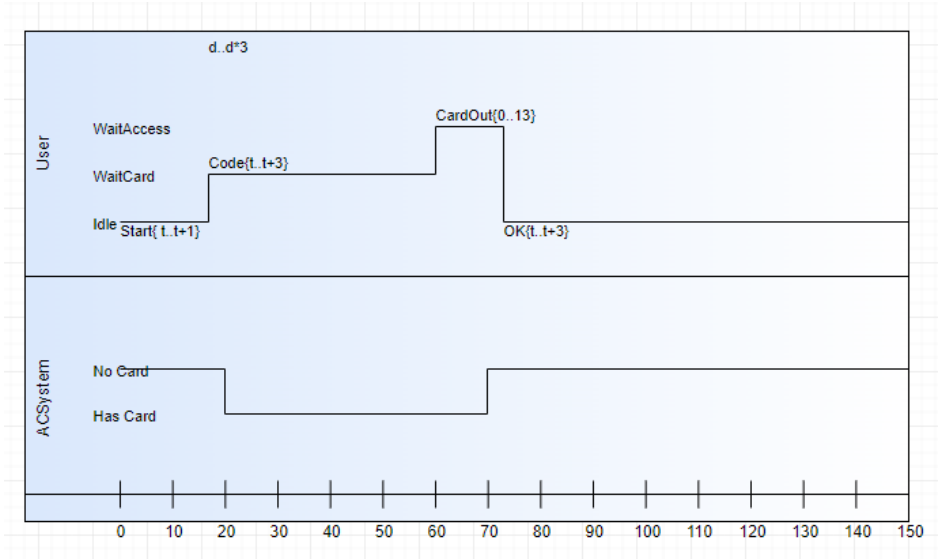


步骤5 在User生命线建立按顺序建立WaitAccess、WaitCard、Idle三种状态，并按下图配置状态切换。

状态 状态转换

时间点	转换状态	事件名	持续时间约束	时间约束	操作
＋ 新增状态转换					
0	Idle	Start	--	t..t+1	
17	WaitCard	Code	d..d*3	t..t+3	
60	WaitAccess	CardOut	--	0..13	
73	Idle	OK	--	t..t+3	

此时图中时间生命线的状态根据条件在不同的时间节点进行切换。



----结束

2 4+1 视图建模

2.1 4+1 视图概述

4+1视图是一组相关联模型的集合，从不同的视角，反映不同利益干系人的关注点。通过逻辑、开发、部署、运行4个典型视角描述系统的各个切面，以用例串接和验证各切面设计。

在架构设计说明书模板中的4+1架构视图模型结构如下图所示：

图 2-1 4+1 架构视图模型结构图

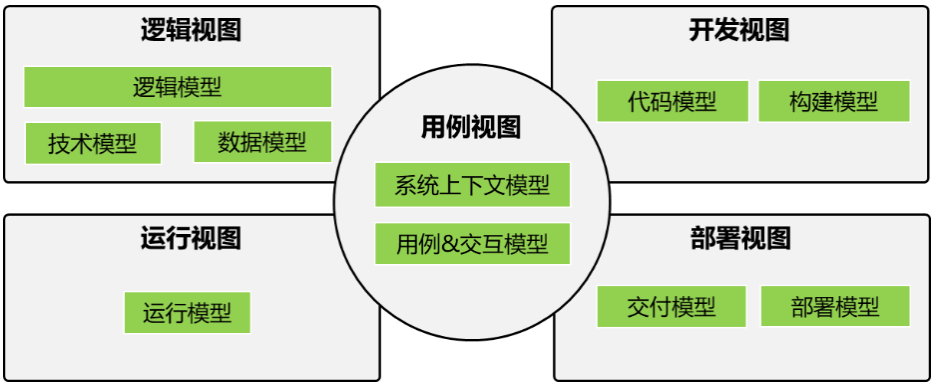


表 2-1 视图类型与描述

视图类型	描述
逻辑视图	逻辑视图面向系统逻辑分析和设计，是描述系统逻辑结构的视图，主要解决系统分析和设计的问题，它描述系统的业务上下文、系统的逻辑分解，以及分解出的逻辑元素间的关系。
开发视图	开发视图面向系统开发及软件管理，是描述系统代码结构，构建结构的视图，主要解决系统技术实现和开发的问题，它依托逻辑视图，描述代码、构建结构。
运行视图	运行视图面向系统运行，是描述系统启动过程、运行期交互的视图，主要解决系统运行期交互，描述各可执行交付件在运行期的交互关系。

视图类型	描述
部署视图	部署视图面向系统部署，是描述系统的交付、安装、部署的视图，主要解决系统安装部署的问题，描述系统的交付、安装、部署关系。
用例视图	用例视图以用例作为驱动元素，驱动和验证其他四个视图的设计，用例视图不增加设计元素，仅增加用例作为输入，因此作为“+1”视图。

2.2 用例视图

2.2.1 用例视图概述

用例视图以用例作为驱动元素，驱动和验证其他四个视图的设计，用例视图不增加设计元素，仅增加用例作为输入，因此作为+1视图。

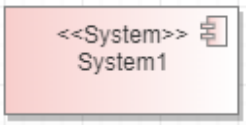
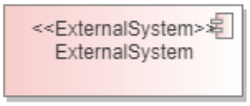
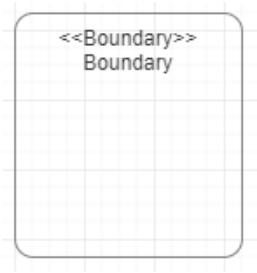
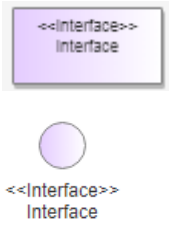
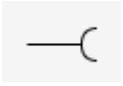
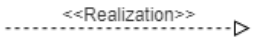
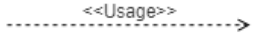
模型类别	描述
上下文模型（必选）	上下文模型描述系统和外部环境（包括人、系统及外部实体）之间的关系，依赖和交互。通过上下文模型可以显示定义系统的范围、职责、边界。
用例模型（必选）	用例模型描述系统的关键用例和交互场景，用于描述系统与外界的交互关系。其中关键用例部分主要描述系统基本的业务用例模型，以及增量版本中影响架构的用例模型；而交互场景描述系统与外部实体之间复杂的交互关系图，采用UML顺序图进行描述绘制，帮助描述隐含的需求和约束，以及系统的验证。

2.2.2 上下文模型

上下文模型描述系统和外部环境（包括人、系统及外部实体）之间的关系，依赖和交互。通过上下文模型可以显示定义系统的范围、职责、边界。元素介绍如下表所示：

表 2-2 上下文模型元素介绍

元素名	图标	含义
Actor		角色，是与系统交互的人或事物。

元素名	图标	含义
System		系统。 广义上，系统是指提供给市场，被客户注意、获取、使用或者消费，并能满足客户某种需求的载体，包括各种有形的物品、无形的电子产品、服务及观念。 狭义上，系统指能独立满足客户某种需求、并符合客户的理解及业界划分习惯的实体。
ExternalSystem		外部系统、设备或者其它系统。
Boundary		边界，可以放入元素，形成一个模块。
Interface		接口，可以是单个接口，也可以是抽象的一组接口的组合。
Provided Interface		提供的接口。 Required Interface和Provided Interface一般是配套使用，一方提供接口，另一方使用，使用Association连线连接两边后，会自动合并。
Required Interface		使用的接口。 Required Interface和Provided Interface一般是配套使用，一方提供接口，另一方使用，使用Association连线连接两边后，会自动合并。
Realization		实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Usage		使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。

元素名	图标	含义
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

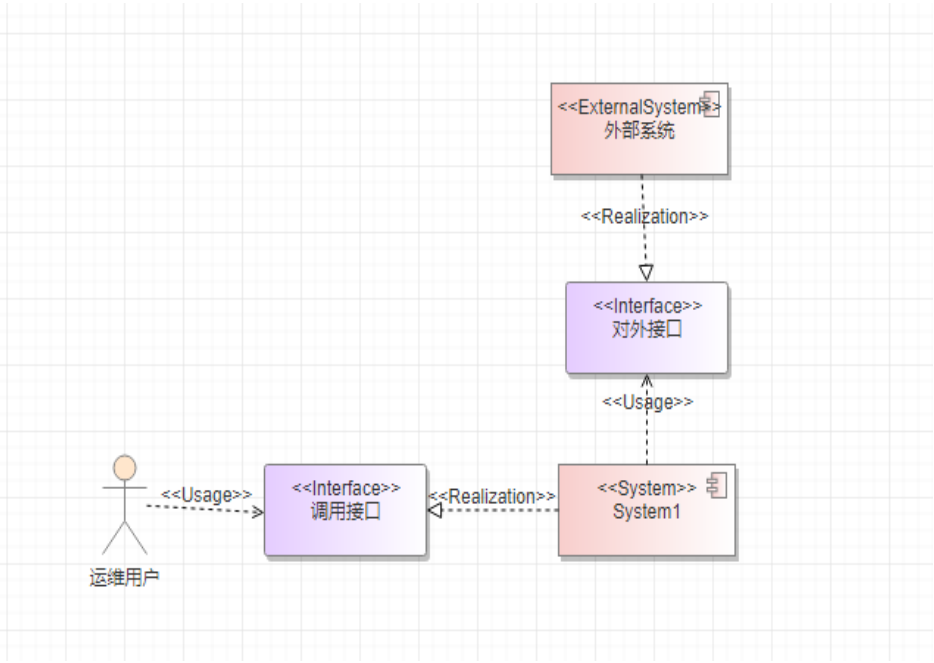
建模步骤

步骤1 创建上下文模型。

您可以使用初始化创建的上下文模型或者创建新的上下文模型，在目录节点右键“新增图”，如果一个系统的交互的外部角色过多时，不适合在一张上下文模型图中建模时，用户可根据外部角色的分类或者产品的应用场景创建不同的上下文模型。

步骤2 建立系统与外部角色的关系。

在上下文模型中描述系统与外部角色的关系通过接口体现，不直接使用连线表示；在上下文模型中需要定义外部角色、交互接口、外部系统、系统，其中系统如果在逻辑模型中已经定义过，则在上下文模型中不能再重复定义，从逻辑模型中引用至上下文模型中即可。



----结束

2.2.3 用例模型

用例模型描述系统的关键用例和交互场景，用于描述系统与外界的交互关系。其中关键用例部分主要描述系统基本的业务用例模型，以及增量版本中影响架构的用例模型；而交互场景描述系统与外部实体之间复杂的交互关系图，采用UML顺序图进行描述绘制，帮助描述隐含的需求和约束，以及系统的验证。元素介绍如下表所示：

表 2-3 用例模型元素介绍

元素名	图标	含义
UserCase		用例，代表的是一个完整的功能。
Actor		角色，是与系统交互的人或事物。
Boundary		边界，可以放入元素，形成一个模块。

元素名	图标	含义
Realization	<<Realization>>	实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Use	<<Use>>	使用关系，指示一个元素需要另一个元素执行一些交互。在用例图中，表示建模参与者如何使用系统功能。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。
Generalization		泛化，是一种继承关系，一个类（通用元素）的所有信息（属性或操作）能被另一个类（具体元素）继承，继承某个类的类中不仅可以有属于自己的信息，而且还拥有了被继承类中的信息。
Include	<<Include>>	包含，包含关系描述的是一个用例需要某种功能，而该功能被另外一个用例定义， 那么在用例的执行过程中，就可以调用已经定义好的用例。
Extend	<<Extend>>	扩展，用例之间的关系， 是指用例功能的延伸，相当于为基础用例提供一个附加功能。

前提条件

用例模型中的Actor需要在上下文模型中定义，再引用至用例模型中，不能在用例模型上重新定义Actor。

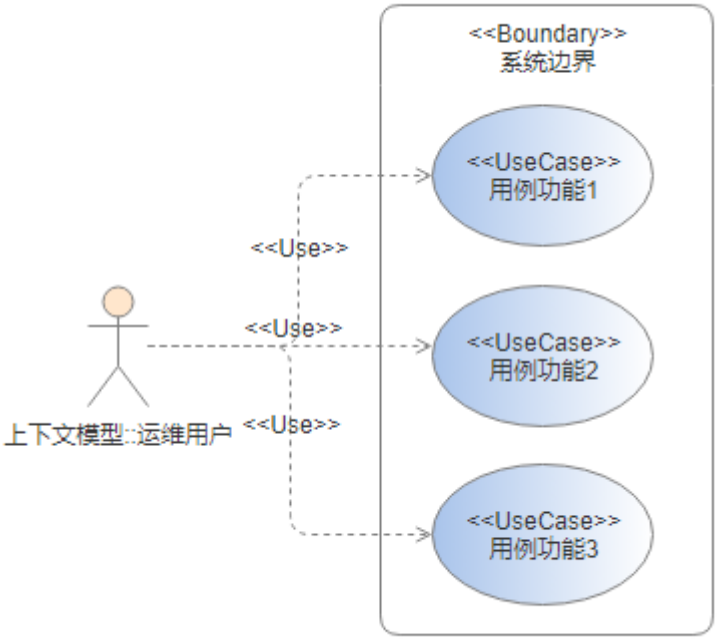
建模步骤

步骤1 创建用例模型。

您可以使用工程初始化建好的用例模型或者在其它目录节点右键菜单中“新增图”，创建新的用例模型，如果用例场景较多，可以创建多个用例模型。

步骤2 画用例模型。

用例模型包含系统基本业务的用例模型、以及增量版本中影响架构的用例模型，从上下文模型中将要用到的Actor角色插入到用例模型图中，再从工具箱中拖入要定义的Use Case元素，和系统边界元素，再建立关系，Actor与用例用的是Use连线关系。



----结束

2.3 逻辑视图

2.3.1 逻辑视图概述

逻辑视图面向系统逻辑分析和设计，描述系统逻辑结构的视图，主要解决系统分析和设计的问题，它描述系统的业务上下文、系统的逻辑分解，以及分解出的逻辑元素间的关系。

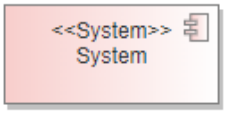
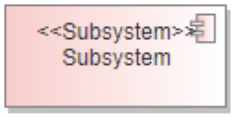
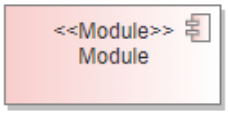
模型类别	描述
逻辑模型（必选）	逻辑模型描述系统的逻辑功能模块分解，将系统分解为相应的逻辑功能元素，并描述各逻辑功能元素之间的关系。
数据模型（强数据场景必选）	数据模型定义系统的关键数据设计，包括关键数据结构设计、数据流，以及数据所有权等。
领域模型（可选）	领域模型描述业务域的概念及其关系，是立足于业务域的分析模型，它通过业务问题域的分析和建模，抽象出领域概念，建立统一的业务语言，从而指导后续的架构设计工作。
功能模型（可选）	功能模型描述按功能分解出特性、功能组、功能元素，以及它们之间的依赖关系。

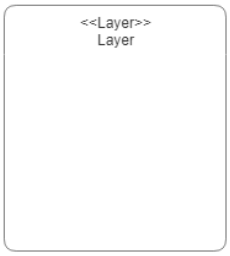
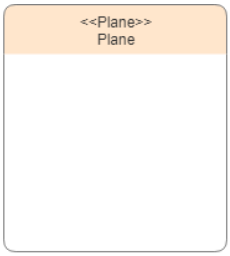
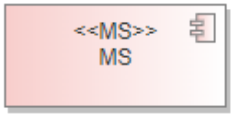
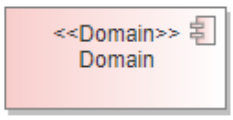
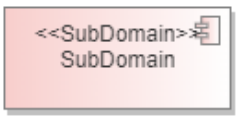
模型类别	描述
技术模型（必选）	技术模型定义系统采用的关键技术部件和技术栈，包括整体框架技术，公共机制，基础设施，公共服务/组件，以及各逻辑功能元素的技术方案等。

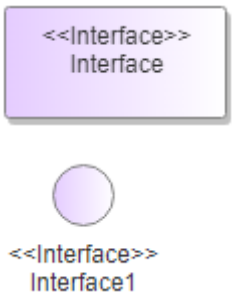
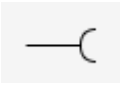
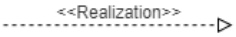
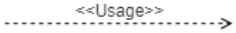
2.3.2 逻辑模型

逻辑模型描述系统的逻辑功能模块分解，将系统分解为相应的逻辑功能元素，并描述各逻辑功能元素之间的关系。元素介绍如下表所示：

表 2-4 逻辑模型元素介绍

元素名	图标	含义
System		广义上，系统是指提供给市场，被客户注意、获取、使用或者消费，并能满足客户某种需求的载体，包括各种有形的物品、无形的电子产品、服务及观念。 狭义上，系统指能独立满足客户某种需求、并符合客户的理解及业界划分习惯的实体。
SubSystem		子系统是一个独立的能够满足特定功能的组合，通过一个或多个它所实现的接口来提供行为。 1. 完全封装自己的内容，通过接口提供行为。 2. 可由组件/模块或更小的子系统组成。 3. 是平台或更大的子系统的组成部分，与其他子系统相对独立。 4. 必须是交付件（能够支持异步开发和外包）。
Module		（IEEE 610.12-1990）系统中一个逻辑上可分离的部分。系统设计中模块特指系统设计阶段输出的系统最小分解部件，系统设计阶段将模块当作黑盒、不涉及模块的内部结构，但要明确给出模块的功能、模块之间的接口。
Service		服务，是指具备明确的业务特征，由一个或多个关联紧密的微服务组成，可直接面向客户/用户进行打包、发布、部署、运维的软件单元。用户可以从业务特征、安装部署、监控运维的角度感知到服务的存在。规模上介于Subsystem与FM（Function Module功能模块）之间的逻辑架构模型元素。Service的功能更加内聚，对外依赖少，接口稳定。

元素名	图标	含义
Component		组件，可独立加载、部署和运行的二进制代码，采用轻量级通讯机制、松耦合高内聚的软件架构构建单元，部署时不能跨节点类型部署（计算机百科全书：组件是软件系统中具有相对独立功能、接口由契约指定、和语境有明显依赖关系、可独立部署、可组装的软件实体）。
SDK		Software Development Kit，软件开发工具包。
Layer		层，辅助图形，不属于架构元素，一般是分层设计，例如网络层、应用层，常见的7层网络模型。
Plane		面，同Layer属于辅助图形，不属于架构元素，在嵌入式系统里常见用户面、管理面等。
MS		MicroService，微服务，是指可独立设计开发部署测试、粒度较小、采用轻量级通讯机制、松耦合高内聚的软件单元。一般来说，用户感知不到微服务的存在。
Domain		域，用于在架构表达、开发管理、对外介绍的过程中，表达系统的层次关系或内部分组，一般由多个服务组成，可以是一级（域）或多级（域/子域，或者域/1级子域/2级子域…）。域和子域不对应实际的设计开发实体，可以根据需要灵活调整。域这个概念，来自于云化产品。以前逻辑架构的实体，这个实体一般指的是子系统，但这个架构实体，带有比较严重嵌入式情节。为此云化产品，习惯用域来表示逻辑架构的实体。
SubDomain		子域，用于在架构表达、开发管理、对外介绍的过程中，表达系统的层次关系或内部分组，域和子域交互使用。

元素名	图标	含义
Interface		接口，可以是单个接口，也可以是抽象的一组接口的组合。 **圆形接口与矩形接口意义相同，仅形状不同。**
Provided Interface		暴露接口。提供接口动作，和Required Interface之间建立Association，表明一个组件提供另外一个组件需要的接口。
Required Interface		请求接口。和Provided Interface之间建立Association，表明一个组件需要的接口是由另外一个组件提供的。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
Realization		实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Usage		使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

建模步骤

参考建议步骤是按逐层分解的方式画图设计，示例步骤分解顺序为：System > Susbsystem > Component > Module，其它的分解顺序结构可参考该方式调整即可，如果产品线有统一的分解规范要求，以产品线要求规范步骤为准。

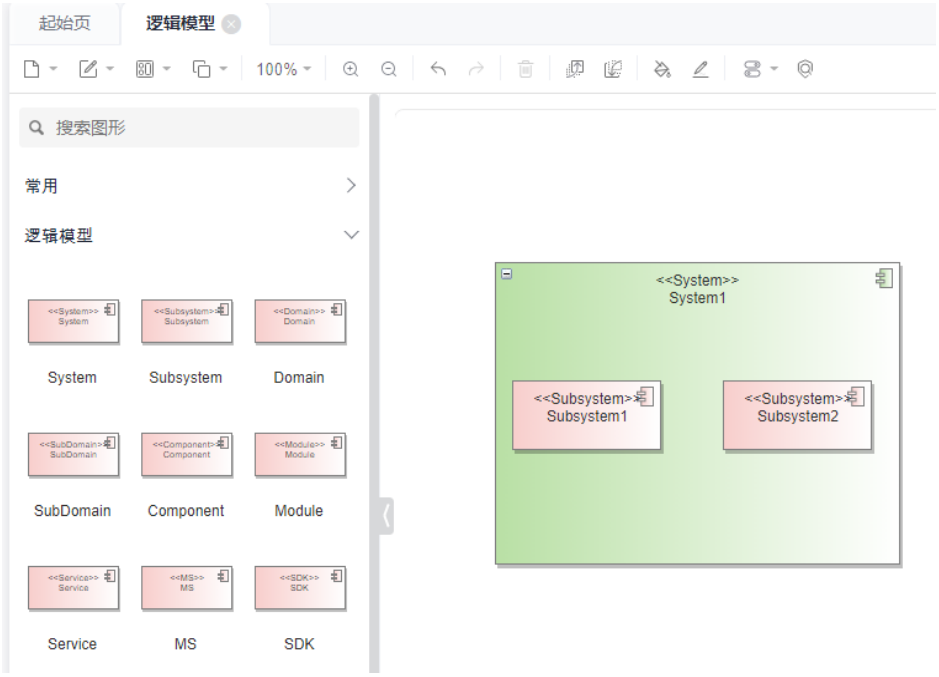
步骤1 创建0层逻辑模型图。

1. 工程初始化创建时会在“逻辑视图 > 逻辑模型”包目录下默认创建一个逻辑模型图，可当作0层逻辑模型，如果是非初始化结构建目录 ，则选择要创建图的包节点 ，单击包后的菜单，选择“新建图”。

2. 图类型选择“4+1视图 > 逻辑视图 > 逻辑模型”，输入图名称，单击保存即可。

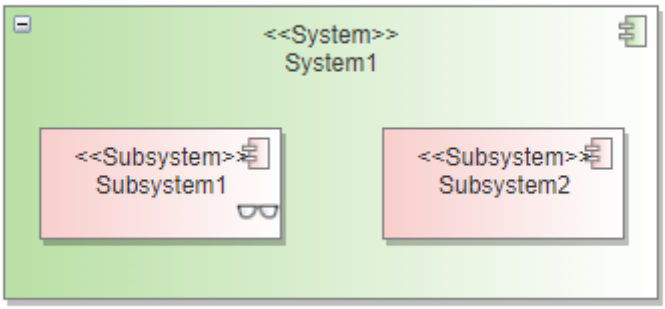
步骤2 创建0层模型逻辑元素。

在0层模型图创建完后，从工具箱中拖入System、Subsystem元素到0层逻辑模型图中。

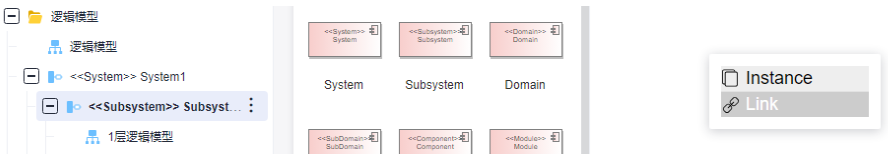


步骤3 创建1层逻辑模型和逻辑元素。

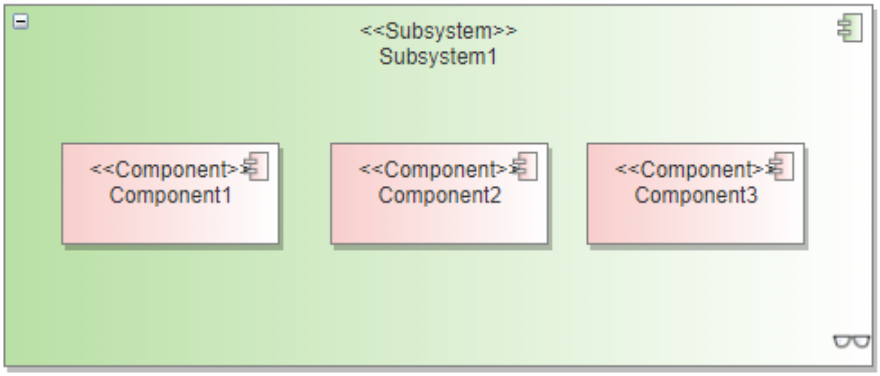
1. 在Subsystem元素下创建子图，子图即为1层逻辑模型，从工程树上将Subsystem元素拖入到1层逻辑模型图中，选择link方式，然后在该Subsystem元素下添加Component元素，建立逻辑关系。
2. 在0层模型上选中Subsystem元素右键“新增图”或者在工程树上的Subsystem元素节点右键“新增图”，在新增图界面图类型仍为逻辑模型。
3. 元素创建完子图后，在所有图中元素图形右下角有一个眼镜图标，双击该图标可快速打开这个元素的子图，如下图所示：



4. 将Subsystem1以Link的形式拖入1层逻辑模型，如下图所示：

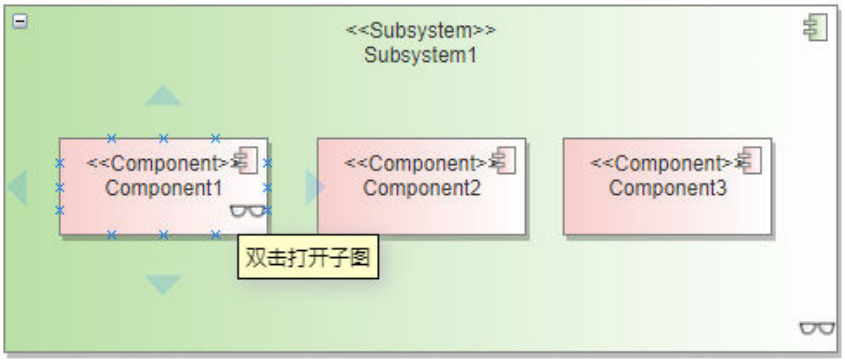


5. 新增的Component组件元素再从工具箱中拖入到图中的Subsystem元素内部，构成包含的父子关系，如下图所示：

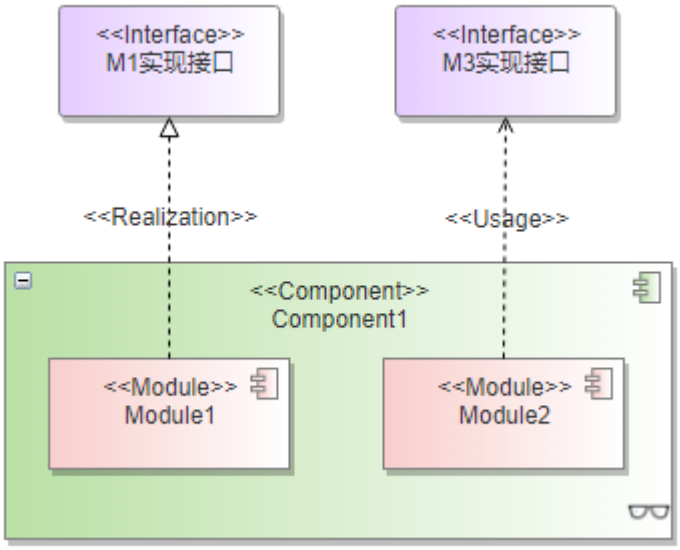


步骤4 创建2层逻辑模型。

1. 参考1层模型创建方式，2层逻辑模型是基于Component1创建子图，如下图所示。



2. 在Component1的子图中引用该父Component1元素到图中，再到图中的从工具箱中创建Module元素到组件下，构成包含的父子关系，在2层模型中会对Module定义对外接口，和使用的接口。
3. Module1暴露实现的接口，提供给外模块调用，连线关系使用Realization。
4. Module2使用其他模块实现提供的接口，以引用的方式拖入图中，连线关系为Usage，如下图所示：

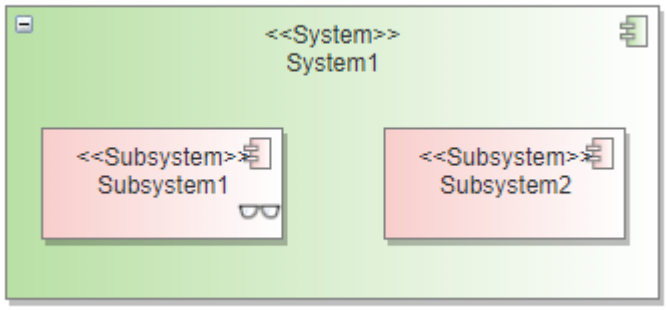


----结束

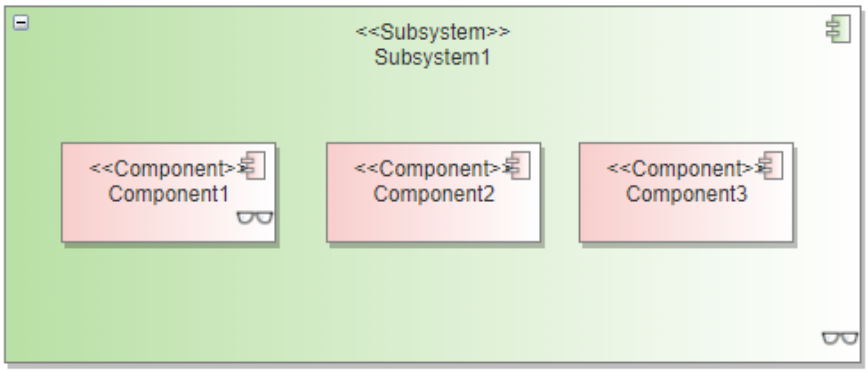
建模示例

最终实现的一个由0层到1层、2层依次展开各层级结构的一个分解逻辑模型图。

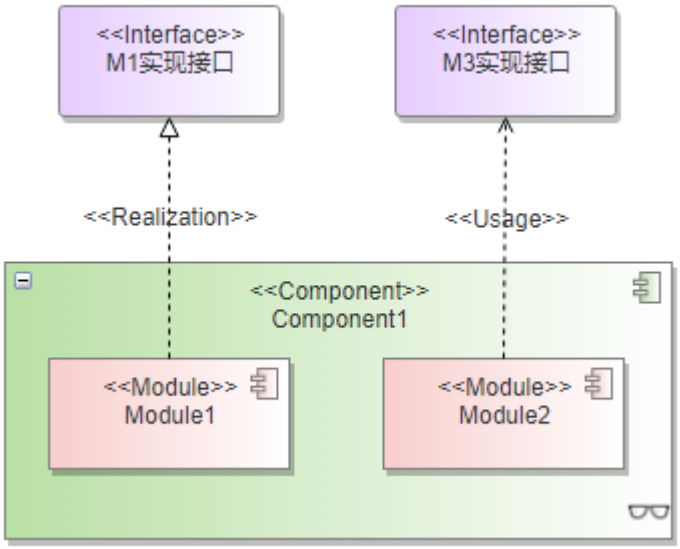
0层逻辑模型。



1层逻辑模型。



2层逻辑模型。

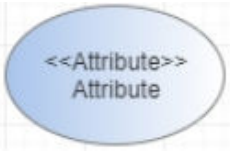
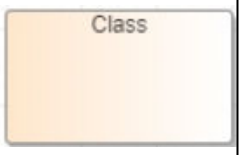
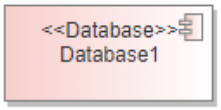
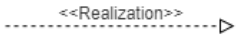
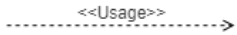


2.3.3 数据模型

数据模型定义系统的关键数据设计，包括关键数据结构设计、数据流，以及数据所有权等。元素介绍如下表所示：

表 2-5 数据模型元素介绍

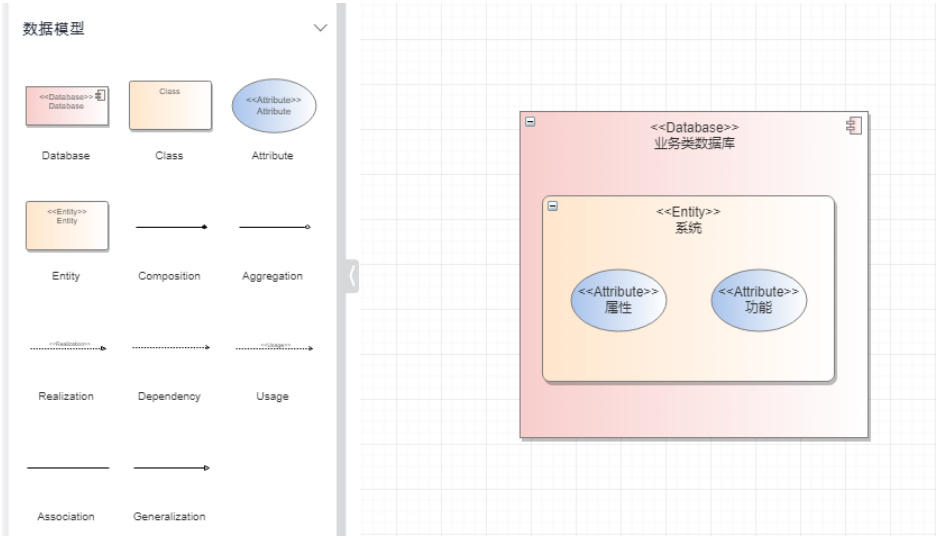
元素名	图标	含义
Entity		实体，该实体建立了一种和数据库表的映射关系。

元素名	图标	含义
Attribute		属性。
Class		类，是对象的集合，展示了对象的结构以及与系统的交互行为。
Database		数据库。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
Realization		实现，是一种类与接口的关系，表示类是接口所有特征和行为的实现。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Usage		使用，是一种使用的关系，表明一个模块在运行的时候，需要使用另外一个模块。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。
Generalization		泛化，是一种继承关系，一个类（通用元素）的所有信息（属性或操作）能被另一个类（具体元素）继承，继承某个类的类中不仅可以有属于自己的信息，而且还拥有了被继承类中的信息。

建模示例

从工具箱中拖入Database、Entity、Attribute元素到数据模型图中，如下图所示：

在数据模型中构建数据对象、实体对象、及实体对象包含的属性对象的结构关系，如果数据模型图对象过多，图形比较复杂时，可以参考逻辑模型中的分层结构，创建多个数据模型，分解来画各实体对象间的关系。



2.3.4 领域模型

领域模型描述业务域的概念及其关系，是立足于业务域的分析模型，它通过业务问题域的分析和建模，抽象出领域概念，建立统一的业务语言，从而指导后续的架构设计工作。元素介绍如下表所示：

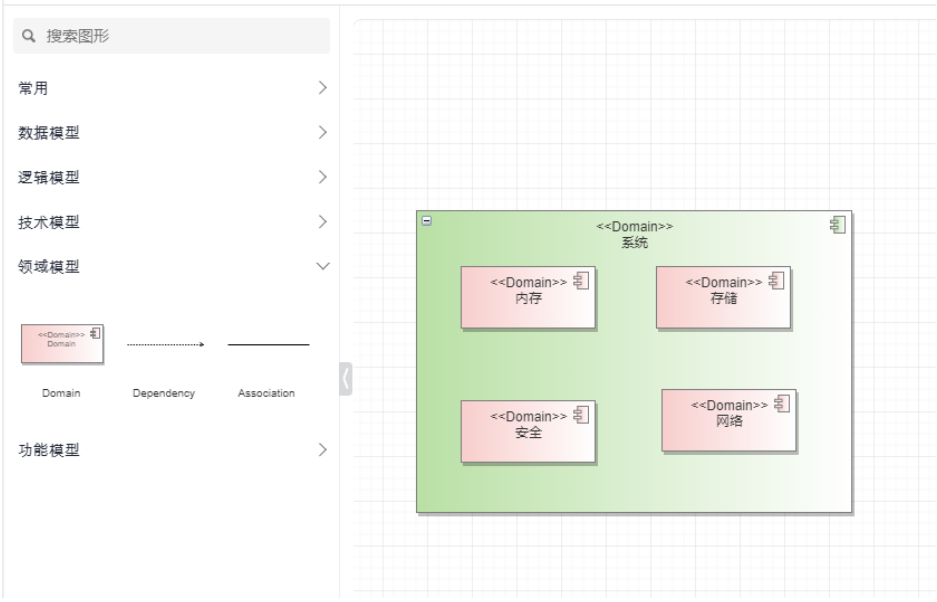
表 2-6 领域模型元素介绍

元素名	图标	含义
Domain		域，用于在架构表达、开发管理、对外介绍的过程中，表达系统的层次关系或内部分组，一般由多个服务组成，可以是一级（域）或多级（域/子域，或者域/1级子域/2级子域…）。 域和子域不对应实际的设计开发实体，可以根据需要灵活调整。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

建模示例

从工具箱中拖入Domain元素到领域模型图中，从系统业务划分上抽象出内存、存储、安全、网络域概念，建立如下图所示模型结构：

在领域模型中以业务域视角进行建模分析，创建业务域对象之间结构关系，如果当领域模型设计图形比较复杂时，可以参考逻辑模型中的分层结构，创建多个领域模型，分解来画各业务域之间的关系



2.3.5 功能模型

功能模型描述按功能分解出特性、功能组、功能元素，以及它们之间的依赖关系。元素介绍如下表所示：

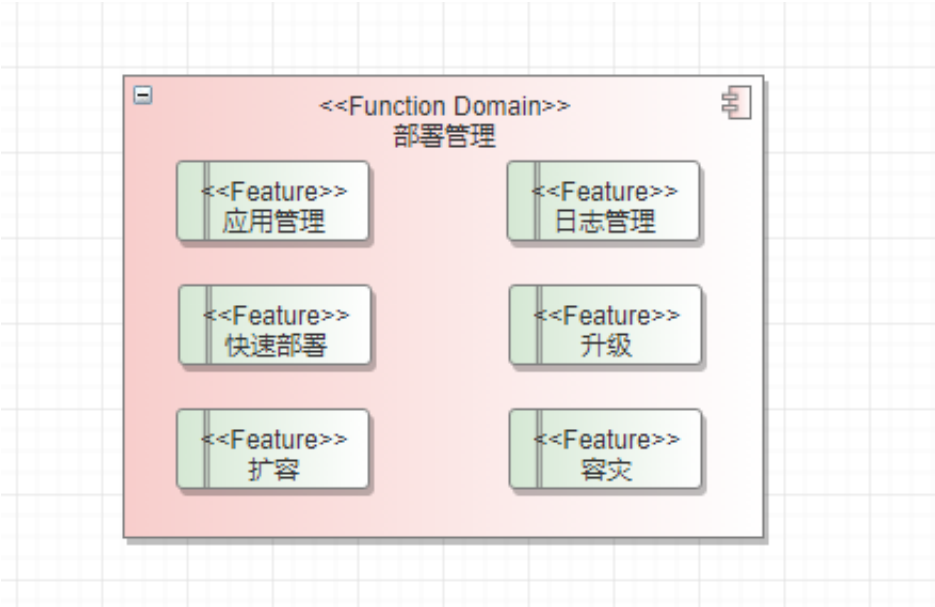
表 2-7 功能模型元素介绍

元素名	图标	含义
Function	<<Function>> Function	功能。
Feature	<<Feature>> Feature	特性。
Function Domain	<<Function Domain>> Function Domain	功能域。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。

建模示例

从工具箱中拖入功能域和特性元素到功能模型图中，以一个应用部署功能为例建立如下图所示模型结构：

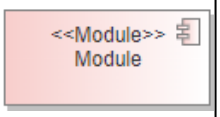
如果当功能模型设计图形比较复杂时，可以参考逻辑模型中的分层结构，创建多个功能模型，分解来画各功能域和特性之间的结构关系。

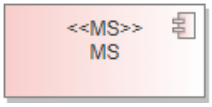
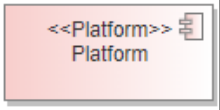
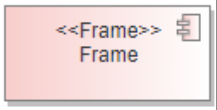
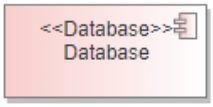
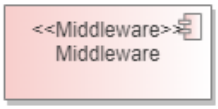
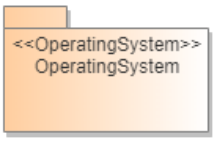
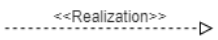
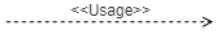


2.3.6 技术模型

技术模型定义系统采用的关键技术部件和技术栈，包括整体框架技术，公共机制，基础设施，公共服务/组件，以及各逻辑功能元素的技术方案等。元素介绍如下表所示：

表 2-8 技术模型元素介绍

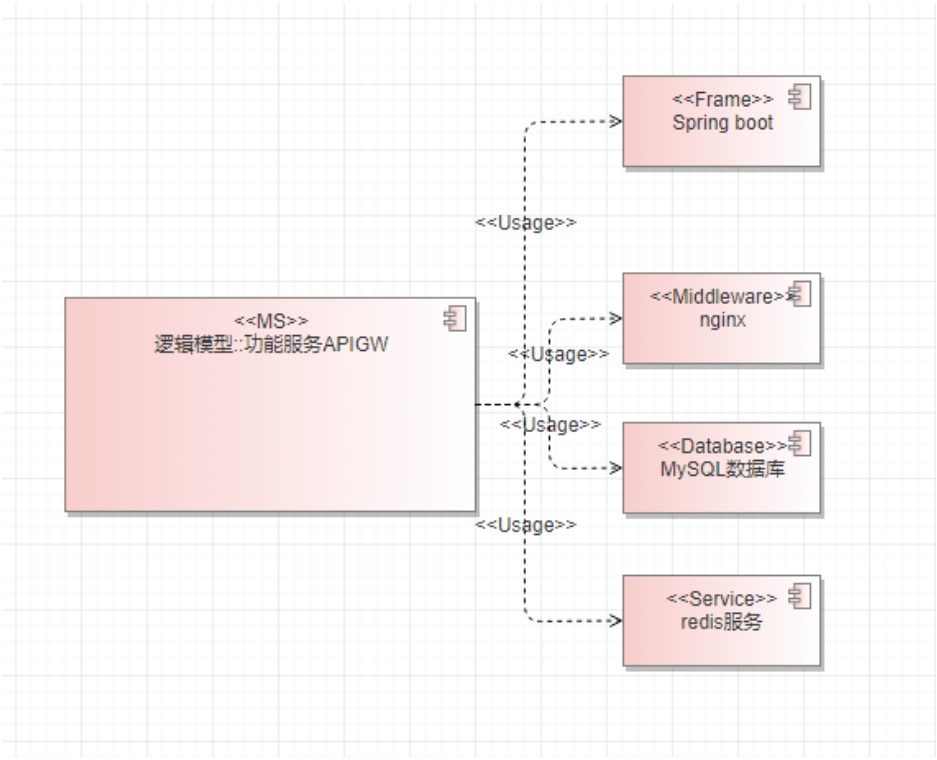
元素名	图标	含义
Module		（IEEE 610.12-1990）系统中一个逻辑上可分离的部分。系统设计中模块特指系统设计阶段输出的系统最小分解部件，系统设计阶段将模块当作黑盒，不涉及模块的内部结构，但要明确给出模块的功能、模块之间的接口。
Service		服务，是指具备明确的业务特征，由一个或多个关联紧密的微服务组成，可直接面向客户/用户进行打包、发布、部署、运维的软件单元。用户从业务特征安装部署、监控运维的角度感知到服务的存在。规模上介于Subsystem与FM之间的逻辑架构模型元素。Service的功能更加内聚，对外依赖少，接口稳定。

元素名	图标	含义
Component		组件，可独立加载、部署和运行的进制代码，采用轻量级通讯机制、松耦合高内聚的软件架构构建单元，部署时不能跨节点类型部署（计算机百科全书：组件是软件系统中具有相对独立功能、接口由契约指定、 和语境有明显依赖关系、可独立部署、 可组装的软件实体）。
MS		是指可独立设计开发部署测试、粒度较小采用轻量级通讯机制、松耦合高内聚的软件单元。一般来说，用户感知不到微服务的存在。
Platform		表示逻辑对象引用的平台，包括名称（Name）、描述（Description） 、架构负责人（Design Owner）、标准名称（artifactName）、版本号（artifactVersion）、平台类型（cpuType）、下载地址（repo）等。
Frame		框架，包含自研或开源框架。
DataBase		数据库。
Middleware		中间件。
OperatingSystem		操作系统。
Realization		实现，是一种类与接口的关系表示类是接口所有特征和行为的实现。
Usage		使用，是一种使用的关系，表明一个模块在运行的时候，需要使用另外一个模块。

建模示例

从工具箱中拖入框架、服务、数据库、组件等等技术元素对象到技术模型图中，从工程树上引用逻辑模型中定义的涉及关键技术的逻辑对象“功能服务APIGW”，建立如下图所示技术部件和技术栈逻辑对象关联的模型结构：

如果当技术模型设计图形比较复杂时，可以参考逻辑模型中的分层结构，创建多个技术模型，分解来画各功能和特性之间的结构关系。



2.4 开发视图

2.4.1 开发视图概述

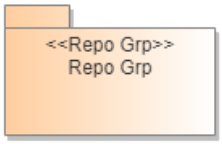
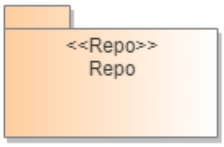
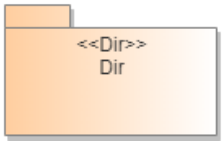
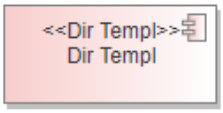
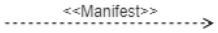
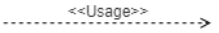
开发视图面向系统开发及软件管理，描述系统代码结构，构建结构的视图，主要解决系统技术实现和开发的问题，它依托逻辑视图，描述代码、构建结构。

模型类别	描述
代码模型（必选）	代码模型定义代码结构以及代码元素逻辑模型中逻辑元素的对应关系，建立逻辑元素到代码仓或者代码目录的映射关系，以实现软件源代码的显示管理。
构建模型（必选）	构建模型定义软件编译构建结构及工具链，构建模型建立代码到运行期文件的映射和追溯关系。

2.4.2 代码模型

代码模型定义代码结构以及代码元素逻辑模型中逻辑元素的对应关系，建立逻辑元素到代码仓或者代码目录的映射关系，以实现软件源代码的显示管理。元素介绍如下表所示：

表 2-9 代码模型元素介绍

元素名	图标	含义
Repo Grp		代码仓组是代码模型分组辅助元素，不对应具体的代码仓，仅表示一个集合。 一个设计对象对应多个代码仓的情况，建议使用Repo Grp标识出来，供构建模型整体引用。
Repo		表示一个代码仓。
Dir		表示一个代码目录。 Dir不单独出现，建立在某个代码仓或者上级目录之下。
File		表示代码仓中的文件，名称中包含文件名+文件类型后缀。
Dir Templ		目录模板。
Manifest		Repo和对应的逻辑设计对象使用Manifest连接。 表示由此代码仓的代码实现此设计对象的功能。连线方向由代码元素指向逻辑元素。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。菱形箭头为整体所在一边。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。菱形箭头为整体所在一边。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Usage		使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。
Build From		构建关系，表示当前构建结果从某一代码目录或者代码文件构建而来，仅适于构建元素与代码元素之间的关系，连线方向由构建元素指向代码元素。

前提条件

因为代码模型主要是描述创建出来的代码元素与逻辑元素的Manifest连线关系，所以在代码模型设计前必须要先完成逻辑模型的设计。

建模步骤

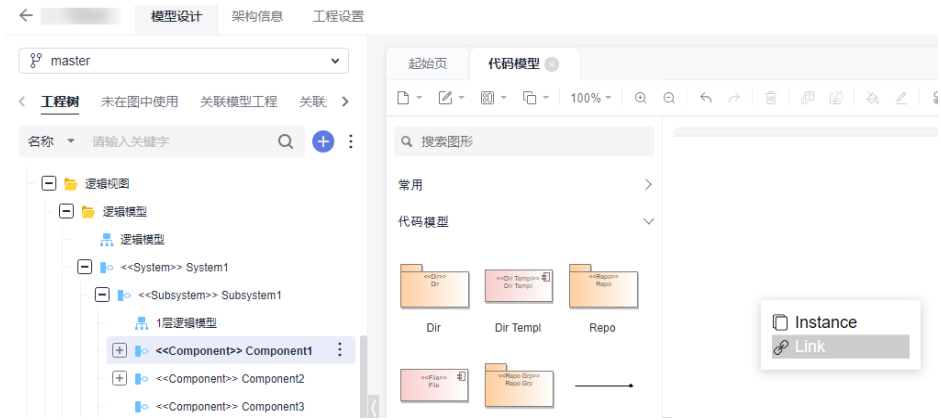
步骤1 创建代码模型图。

创建新的代码模型图或者在已有的代码模型图中进行画图设计，如果设计内容过多，可根据实际情况将内容进行拆分，创建多个代码模型图，在对应的代码模型图中去建立关系。

步骤2 引用逻辑元素到代码模型。

在代码模型中不能创建新的逻辑元素，必须要从逻辑模型中引用到代码模型中，引用逻辑元素的操作方式有两种

方式一：直接从工程树上将逻辑元素节点拖入到打开的代码模型图中，选择Link方式。

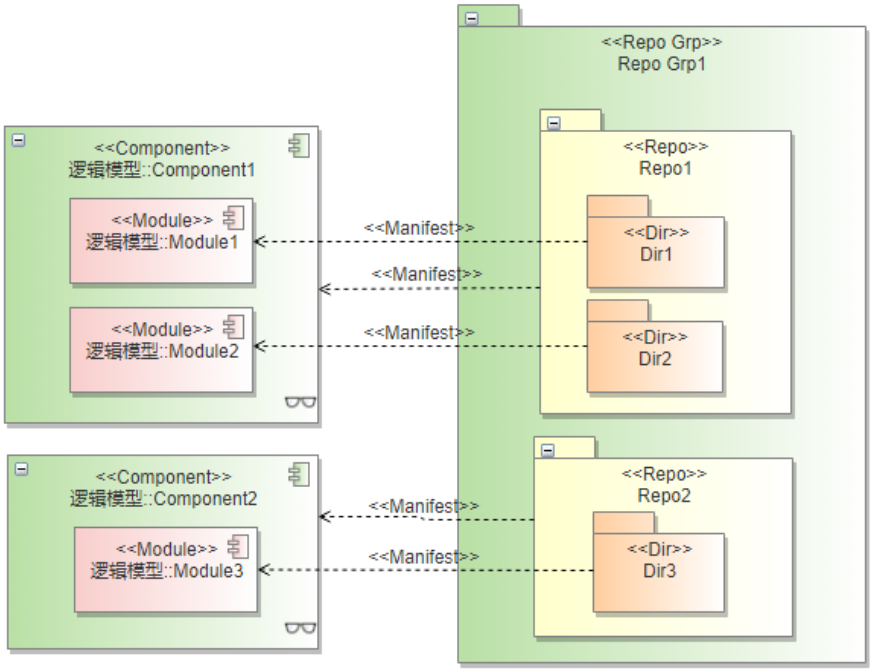


方式二：在逻辑模型图中按Ctrl键**多选**或者**框选多个**逻辑元素Ctrl+C复制，然后再到代码模型中Ctrl+V粘贴，粘贴方式选择引用方式，可以保留嵌套组合的结构样式。

上述两种方式都可以将逻辑元素引用到代码模型图中，在代码模型中只需要引用需要建立映射关系的逻辑元素即可。

步骤3 创建代码元素并与逻辑元素建立Manifest连线关系。

在步骤2中将逻辑元素引用到代码模型中后，再从工具箱中拖入代码仓元素，如果存在一个代码仓组下的多个代码仓元素，可以选代码仓组元素，将多个代码仓元素包含起来，如果具体模块对应的是代码仓中某一目录中的代码，则需要在对应的代码仓元素中创建Dir目录元素，再建立对应逻辑元素与代码元素的Manifest关系。

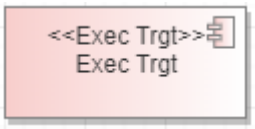
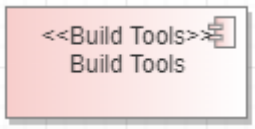
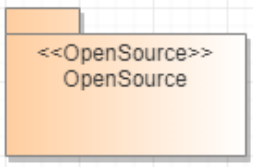
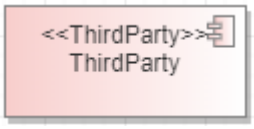
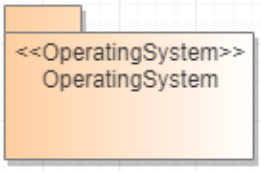
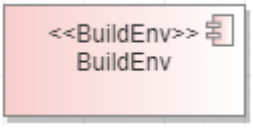


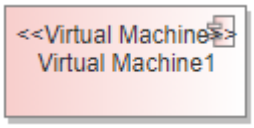
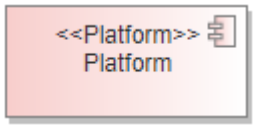
----结束

2.4.3 构建模型

构建模型定义软件编译构建结构及工具链，构建模型建立代码到运行期文件的映射和追溯关系。元素介绍如下表所示：

表 2-10 构建模型元素介绍

元素名	图标	含义
Exec Trgt		表示逻辑对象构建的二进制结果（.so/.bin/rpm等）。
Compile Trgt		表示逻辑对象构建的二进制编译结果（.o/obj/.a等），专指提供二进制编译结果给其他对象使用的场景，其他场景不用体现Compile Trgt。
Build Tools		表示逻辑对象构建时所需的构建工具。 【建议】Build Tools嵌套可以作为分组来区分不同类型的工具集合，作为构建时引用。
OpenSource		表示逻辑对象构建时所需的开源软件代码。
ThirdParty		表示构建中所使用的第三方元素。
OperatingSystem		表示逻辑对象构建时所需的操作系统。
BuildEnv		表示构建使用的构建环境信息。
Container		表示逻辑对象构建时所需的容器。

元素名	图标	含义
Virtual Machine		表示逻辑对象构建时所需的虚拟机。
Platform		表示逻辑对象引用的平台。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Usage		使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。
Deployed To		部署关系是一种依赖关系，在部署图中，指一个工件被部署到一个节点或可执行目标上。
Build From		构建关系，表示当前构建结果从某一代码目录或者代码文件构建而来，仅适于构建元素与代码元素之间的关系，连线方向由构建元素指向代码元素。

前提条件

因为构建模型主要是描述创建出来的构建元素与代码元素的Build From构建关系，所以在画构建模型设计前必须要先完成代码模型的设计。

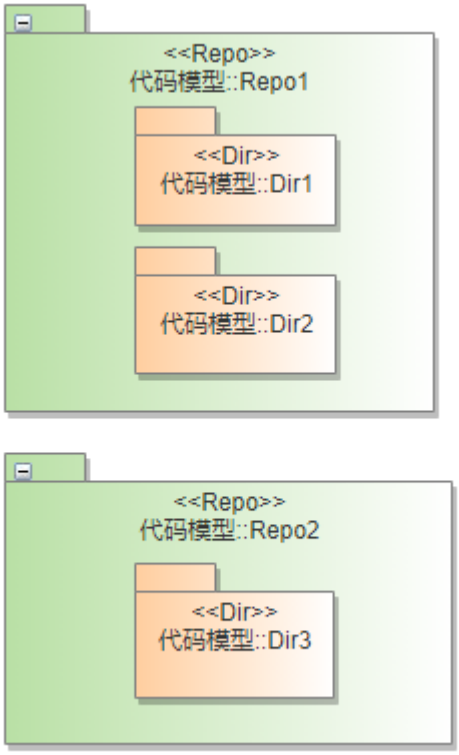
建模步骤

步骤1 创建构建模型。

创建新的构建模型图或者在已有的构建模型图中进行画图设计，如果设计内容过多，可根据实际情况将内容进行拆分，创建多个构建模型图，在对应的构建模型图中去建立关系。

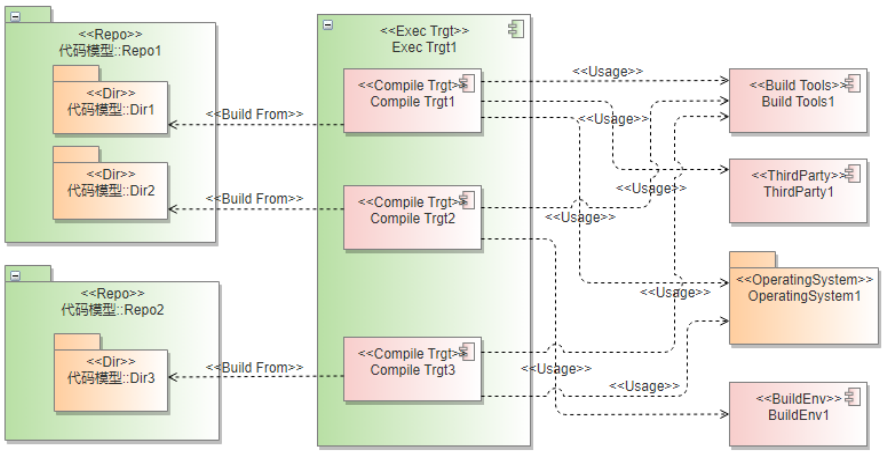
步骤2 引用代码元素到构建模型。

将代码元素引用到构建模型中跟代码模型中的步骤2一样，有两种方式，从工程树上将代码元素拖入到构建模型图中选link方式引用；另一种从代码模型图中多选复制元素，以引用方式粘贴到构建模型图中。



步骤3 建立代码元素与构建元素的Build From构建关系。

在步骤2中将代码元素引用到构建模型图后，再从工具箱中构建模型图形库中拖入构建元素，创建与代码元素需要建立关系的构建元素，并建立构建元素与代码元素的Build From关系，同时需要创建一些构建过程中构建元素使用到的构建工具和依赖的构建环境、平台等信息，并建其中的连线关系。



----结束

2.5 部署视图

2.5.1 部署视图概述

部署视图面向系统部署，描述系统的交付、安装、部署的视图，主要解决系统安装部署的问题，描述系统的交付、安装、部署关系。

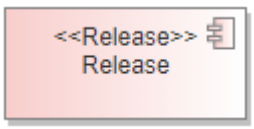
表 2-11 部署视图

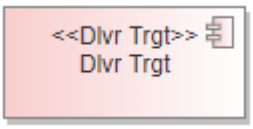
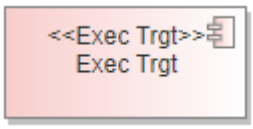
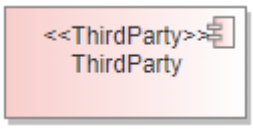
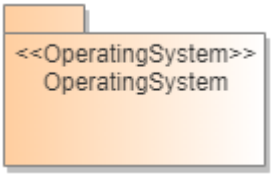
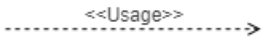
模型类别	描述
交付模型（必选）	交付模型定义的是从构建结果和外部软件一起打包成最终交付给客户的Release Offering的模型设计过程。
部署模型（必选）	部署模型定义产品的部署关系，它依托于构建模型或交付模型，描述每个构建文件或者交付件以及相应的软件部署实体的部署依赖关系和部署约束。

2.5.2 交付模型

交付模型定义的是从构建结果和外部软件一起打包成最终交付给客户的Release Offering的模型设计过程。元素介绍如下表所示：

表 2-12 交付模型元素介绍

元素名	图标	含义
Release		指产品最终发布的release版本，按照公司发布版本命名规定release中自带版本号。

元素名	图标	含义
Dlvr Trgt		指通过Exec Trgt、Exec Trgt+DlvrTrgt、Dlvr Trgt+外部软件打包后的package.Dlvr Trgt 一般是tar/gz包。
Exec Trgt		来源于构建视图中的构建结果，一般场景下不在交付模型图中创建该该元素，都是从构建模型中引用到交付模型使用。
ThirdParty		需要作为软件一起打包交付给客户的第三方件。
OperationSystem		需要作为软件一起打包交付给客户的操作系统。
Package Tools		打包工具，在打包过程使用到的工具都可以用该元素表示，以名称作区分。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
Dependency		依赖，是一种使用的关系，即一个类的实现需要另一个类的协助。
Usage		使用，是一种使用的关系。表明一个模块在运行的时候，需要使用另外一个模块。

前提条件

因为交付模型主要是描述构建模型中的结构元素打包成交付文件的过程，所以必须先完成构建模型的设计才能进行交付模型。

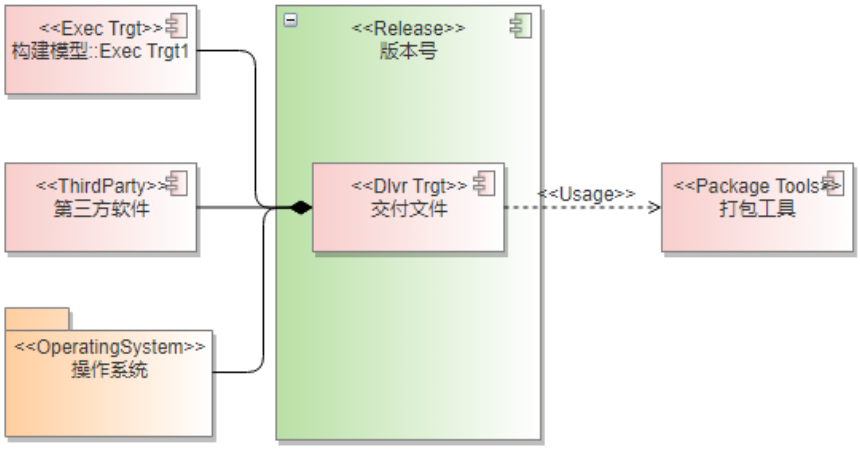
建模步骤

步骤1 创建交付模型。

创建新的交付模型图或者在已有的交付模型图中进行画图设计，如果设计内容过多，可根据实际情况将内容进行拆分，创建多个交付模型图，在对应的交付模型图中去建立关系。

步骤2 建立构建元素与交付元素的组合关系。

将构建模型中生成的构建元素引用到交付模型图中，并创建打包所需要的第三方软件或者操作系统，如果涉及打包工具，也可以在图中描述。



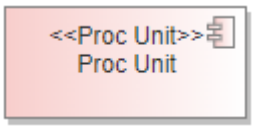
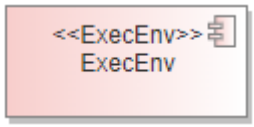
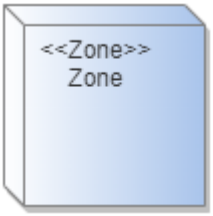
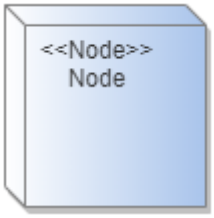
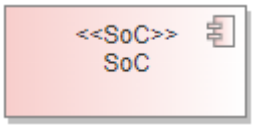
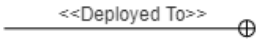
----结束

2.5.3 部署模型

部署模型定义产品的部署关系，它依托于构建模型或交付模型，描述每个构建文件或者交付件以及相应的软件部署实体的部署依赖关系和部署约束。元素介绍如下表所示：

表 2-13 部署模型元素介绍

元素名	图标	含义
FRU		现场可更换单元（Field-Replaceable Unit）。

元素名	图标	含义
Proc Unit		处理单元（Process Unit）。
Process		进程。
ExecEnv		执行环境（Execution Environment），可以是VM、docker等。
Zone		部署区域。
Node		部署节点。
SoC		片上系统（System-On-a-Chip）。
Deployed To		部署关系是一种依赖关系，在部署图中，指一个工件被部署到一个节点或可执行目标上。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。
CommunicationPath		通信路径。定义两个部署目标能够交换信号和消息的通信路径。

前提条件

部署模型描述产品打包交付件的部署场景，所以画部署模型需要完成前面的构建模型或交付模型。

因为有些特殊产品没有交付打包过程，只有构建过程，在部署时使用的是构建过程生成文件来部署到部署模型中，描述部署的场景。

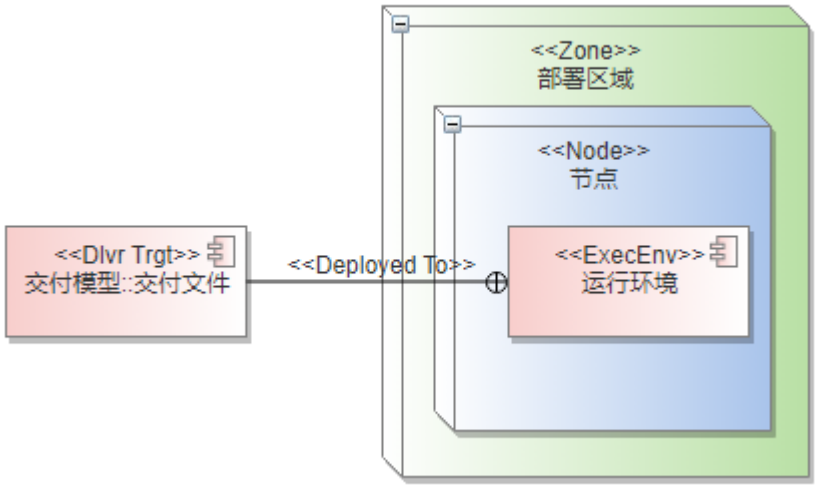
建模步骤

步骤1 创建部署模型。

创建新的部署模型图或者在已有的部署模型图中进行画图设计，如果部署模型场景较多，可根据实际情况将内容进行拆分，按实际部署场景创建多个部署模型图。

步骤2 建立交付元素与部署元素的部署关系。

从工具箱拖入部署元素创建到部署模型图中，描述部署场景，再将交付模型中定义的打包交付件引用到图中，并与部署元素建立部署关系。



----结束

2.6 运行视图

2.6.1 运行视图概述

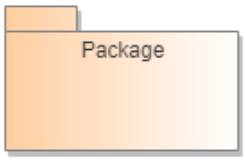
运行视图面向系统运行，描述系统启动过程、运行期交互的视图，主要解决系统运行期交互，描述各可执行交付件在运行期的交互关系。

表 2-14 运行视图

模型类别	描述
运行模型（可选）	运行模型描述系统运行期间的关系，从进程的维度描述系统运行时的交互过程和关键数据流。
运行模型-顺序图（必选）	运行模型-顺序图模型是从逻辑模型中的架构对象维度描述系统运行时的交互过程以及关键的数据流。
运行模型-活动图（可选）	运行模型-活动图展示了从起点到终点的工作流程，详细说明了在活动的进展中存在的许多决策路径。

2.6.2 运行模型

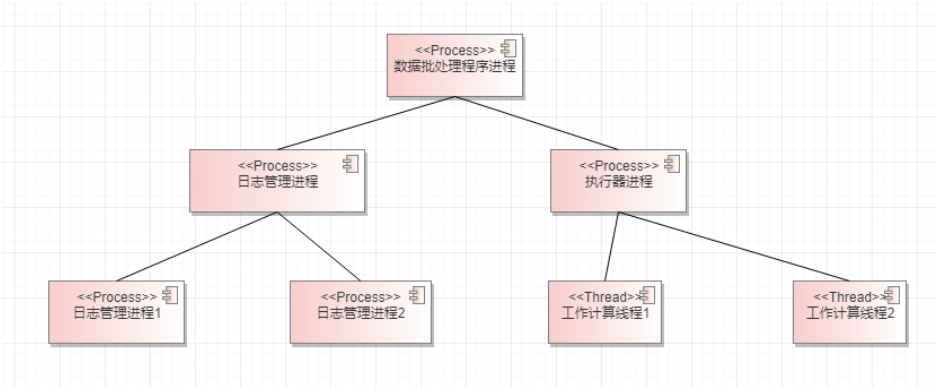
表 2-15 运行模型元素介绍

元素名	图标	含义
Process		进程，加截的组件、服务/微服务列表[1..*]。
Thread		线程，加载的组件、服务/微服务列表[1..*]。
Package		进程组，包含进程列表。
Mutex		锁/临界区，锁类型（自旋锁、排它锁、分布式锁、共享锁等）。
Composition		组合，是整体与部分的关系，但部分不能离开整体而单独存在。
Aggregation		聚合，是整体与部分的关系，且部分可以离开整体而单独存在。

元素名	图标	含义
Association		关联，是一种拥有的关系，它使一个类知道另一个类的属性和方法。

建模示例

运行模型不需要引用其它模型中的元素，根据实际业务流程在图中创建对应的进程和线程元素，并建立它们之间的交互关系。如下图所示描述一个数据批量处理交互过程。



2.6.3 运行模型（顺序图）

运行模型-顺序图中的元素都来自于上下文模型中的用户角色、外部系统或者逻辑模型中定义的逻辑元素，不需要在顺序图中创建新元素，只需要使用到UML顺序图中的消息连线。元素介绍如下表所示：

表 2-16 运行模型（顺序图）元素介绍

元素名	图标	含义
Message		同步消息连线,消息的发送者把控制传递给消息的接收者，然后停止活动，等待消息的接收者放弃或者返回控制。
Async Message		异步消息连线，消息发送者通过消息把信号传递给消息的接收者，然后继续自己的活动，不等待接收者返回消息或者控制。异步消息的接收者和发送者是并发工作的。
Reply Message		返回消息连线，返回消息表示从过程调用返回，一般与Message配套使用。
Self Message		自消息连线，表示方法的自身调用或者一个对象内的一个方法调用另外一个方法。

元素名	图标	含义
Create Message		创建对象消息连线，这个消息指向对象以后，对象的位置就不会出现在顶部，而是创建消息所在的位置。
Delete Message		终止对象消息连线，这个消息线指向对象以后，对象生命线下方出现终止符，表示对象不再接收消息调用。

前提条件

因为运行模型-顺序图中的元素都是来源于逻辑模型或上下文模型中的元素，所以需要先完成上下文模型和逻辑模型中的设计。

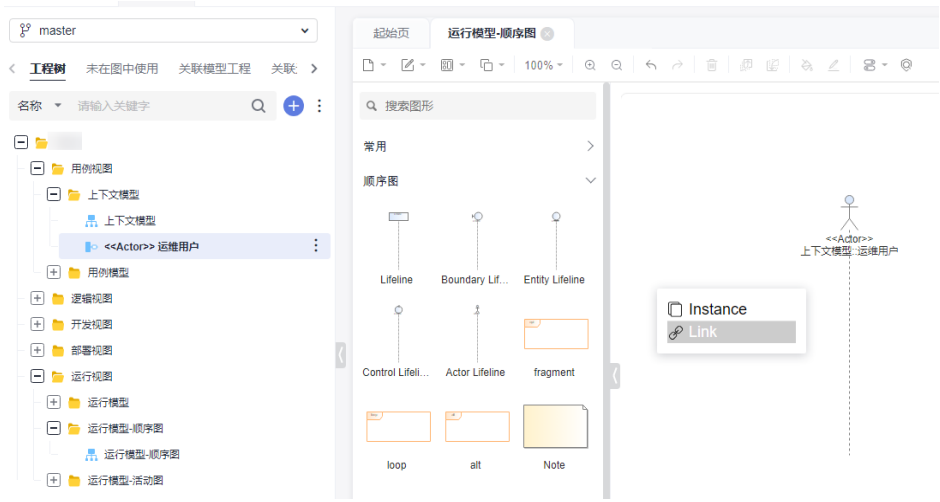
建模步骤

步骤1 创建运行模型-顺序图。

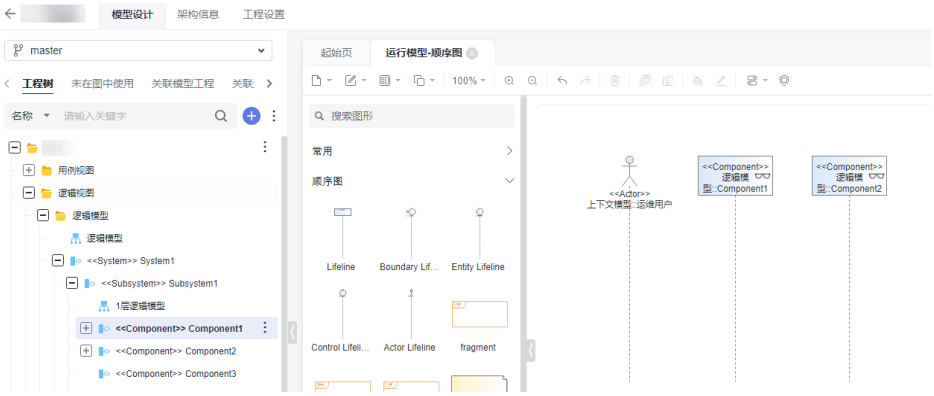
在目录或者元素节点右键菜单，选择“新增图”，在对应的目录或者元素节点下面创建“运行模型 > 顺序图”，如下图所示：

步骤2 引用角色和逻辑对象，描述消息交互过程。

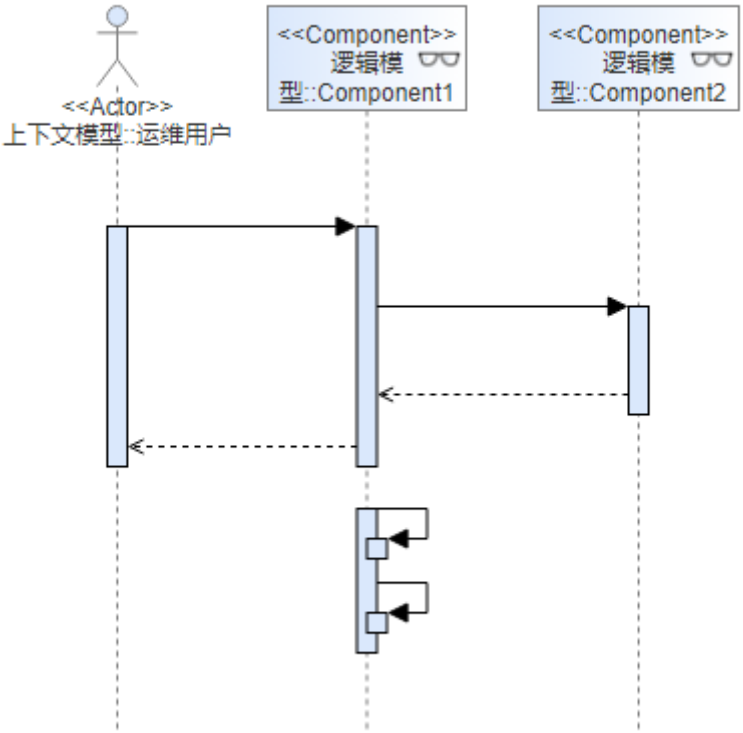
1. 将工程树中上下文模型中定义的用户角色、外部系统元素以引用方式拖入到运行模型-顺序图中，会自动变成生命线样式。如下图所示：



2. 再从工程树上将逻辑模型中定义的涉及交互场景的逻辑元素引用到“运行模型-顺序图”中，如下图所示：



3. 当将需要引入的逻辑元素拖入到图中后，再去绘制交互消息的关系连线，顺序图消息连线画法可参考**绘制消息线**，如下图所示：



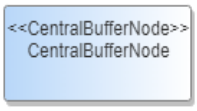
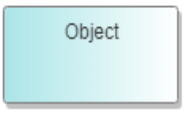
----结束

2.6.4 运行模型（活动图）

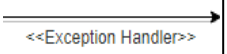
“运行模型-活动图”展示了从起点到终点的工作流程，详细说明了在活动的进展中存在的许多决策路径。

活动图对用户和系统遵循流程的行为进行建模，它们是流程图或工作流的一种，但是它们使用的形状略有不同，元素介绍如下表所示：

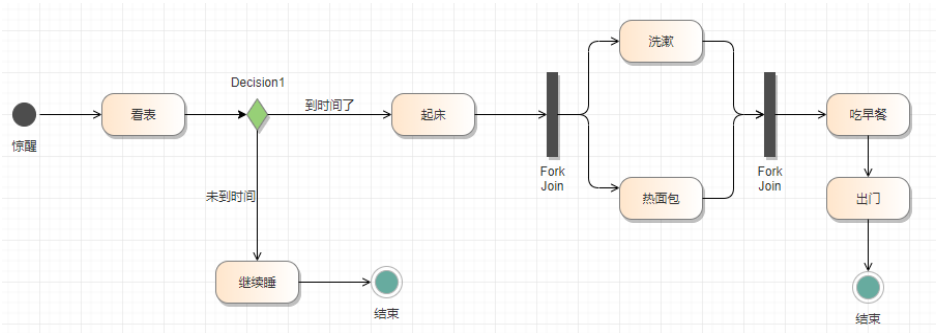
表 2-17 活动图元素介绍

元素名	图标	含义
Action		动作是可执行的原子计算，它导致模型状态的改变和返回值。
Activity		活动是状态机内正在进行的非原子执行。
StructuredActivity		结构化活动是一个活动节点，可以将下级节点作为独立的活动组。
CentralBufferNode		中央缓冲区节点是一个对象节点，用于管理活动图中表示的来自多个源和目标的流。
Datastore		数据存储区定义了永久存储的数据。
ExceptionHandlerNode		异常处理程序元素定义发生异常时要执行的一组操作。
Object		封装了状态和行为的具有良好定义界面和身份的离散实体，即对象实例。
Decision		是状态机中的一个元素，在它当中一个独立的触发可能导致多个可能结果，每个结果有它自己的监护条件。
Merge		状态机中的一个位置，两个或多个可选的控制路径在此汇合或"无分支"。
Send		即发送者对象生成一个信号实例并把它传送到接收者对象以传送信息。

元素名	图标	含义
Receive		接收就是处理从发送者传送过来的消息实例。
Partition		分区元素用于逻辑组织活动的元素。
Partition		分区元素用于逻辑组织活动的元素。
Initial		用来指明其默认起始位置的伪状态。
Final		组成状态中的一个特殊状态，当它处于活动时，说明组成状态已经执行完成。
Flow Final		Flow Final元素描述了系统的退出，与Activity Final相反，后者代表Activity的完成。
Synch		一个特殊的状态，它可以实现在一个状态机里的两个并发区域之间的控制同步。

元素名	图标	含义
Fork Join		Fork，复杂转换中，一个源状态可以转入多个目标状态，使活动状态的数目增加。 Join，状态机活动图或顺序图中的一个位置，在此处有两个或以上并列线程或状态归结为一个线程或状态。
Fork Join		（Fork）复杂转换中，一个源状态可以转入多个目标状态，使活动状态的数目增加。 （Join）状态机活动图或顺序图中的一个位置，在此处有两个或以上并列线程或状态归结为一个线程或状态。
Region		并发区域。
Control Flow		（控制流）在交互中，控制的后继轨迹之间的关系。
Object Flow		（对象流）各种控制流表示了对象间的关系、对象和产生它（作输出）或使用它（作输入）的操作或转换间的关系。
Constraint		是一个语义条件或者限制的表达式。UML 预定义了某些约束，其他可以由建模者自行定义。
Exception Handler		异常处理。捕获异常根据异常类型查找到对应的异常处理方法，然后执行对应的方法。
Interrupt Flow		中断流是用于定义异常处理程序和可中断活动区域的连接器的两个UML概念的连接。中断流是活动边缘的一种。它通常用于活动图中，以对活动过渡进行建模。
Anchor		锚点。
Containment		内嵌，表示嵌在内部的类。

活动图示例如下所示：



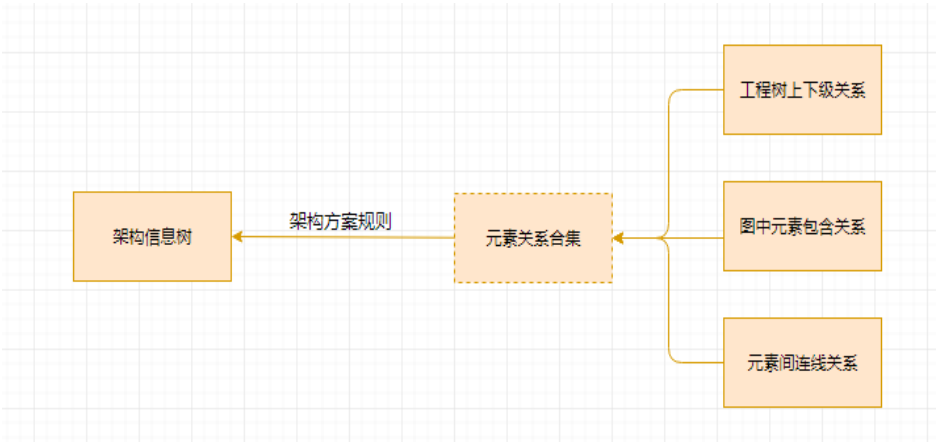
2.7 架构信息

2.7.1 架构信息树

架构信息树页签中可展示基于各种模型的架构方案规则，根据该类模型的元素关系集合生成一棵全量关系结构树。

架构方案规则的配置只有工程的管理员级别角色（带有配置资源操作权限的角色）才可配置，一般是产品的架构师来配置，制定产品的各模型中元素关系约束规则，例如逻辑模型的默认架构方案规则，方案中约束System下级节点只能是Subsystem、Domain、Service、MS这四类元素，如果在实际画图中出现了这四种以外的逻辑元素，则在最后的逻辑架构信息树中不会挂出。

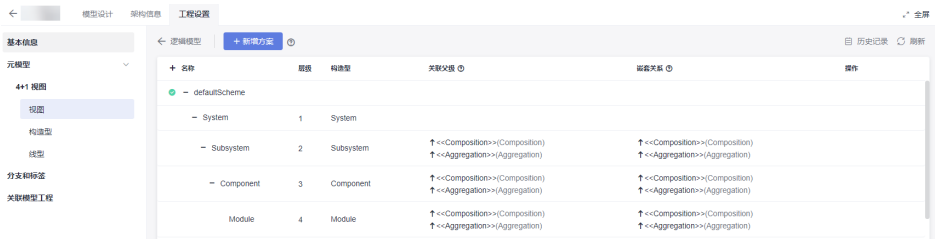
每种模型只能启用一种默认方案，但可以配置多种架构方案规则，在查询架构信息树时可以选择不同的方案，按不同的方案生成不同的信息树，**同一工程不同分支共用的是同一套架构方案。**



有些必选模型是带有默认方案，默认方案不可修改，可供参考，也可以导出默认方案，修改方案名称后，再导入生成新的方案，基于该方案来修改为自己想要的结构方案。

以逻辑模型的默认方案其中一条System > Subsystem > Component > Module结构为例展示构树过程。

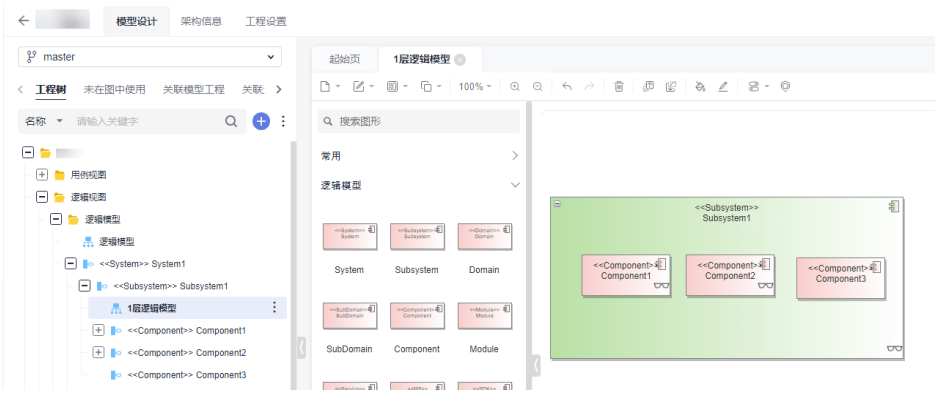
架构方案规则如下：



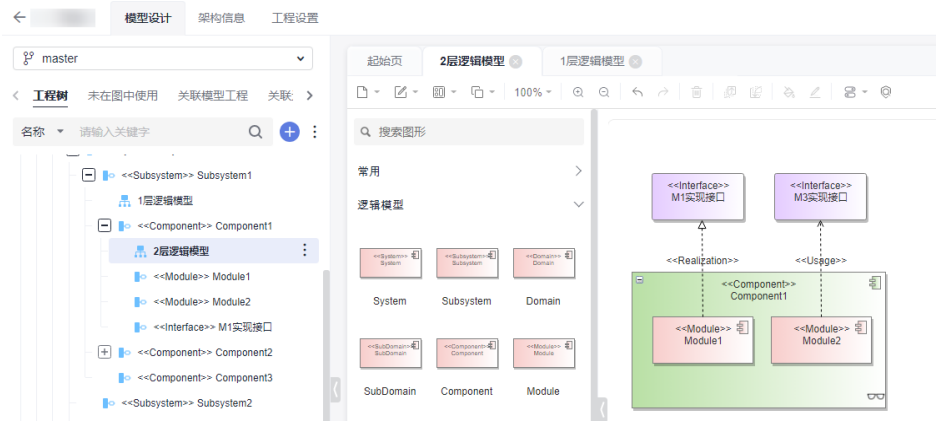
工程中的关系集合：

工程树与图中关系一致时，只取一种有效关系参与构树，下图是在0层模型中System与Subsystem关系。

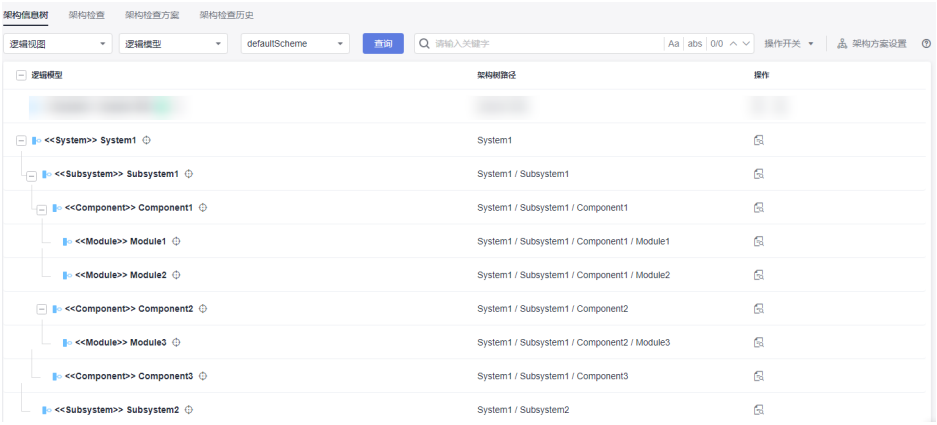
在一层逻辑模型中是Subsystem与Component的关系，图中包含关系，工程树中上下级关系，取其中一种有效关系参与构树。



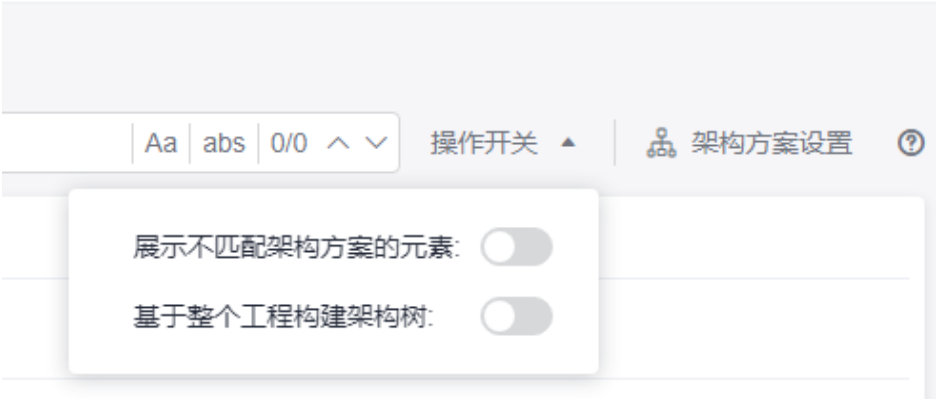
在2层逻辑模型中，是Component与Module的关系，图中包含关系，工程树中上下级关系，取其中一种有效关系参与构树，但是在2层模型中还在Module下定义了实现接口，而方案中该路径下的Module节点下没有配置Interface，故在正常构树时，Interface是不会出现在最后的架构信息树中。



最后单击“查询”按钮，实时生成的逻辑架构信息树如下：

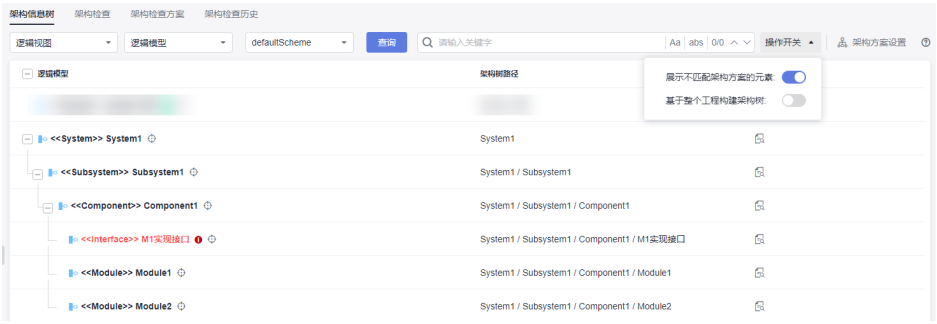


架构信息树中还有两个操作开关配置，一个是“展示不匹配架构方案的元素”，另一个是“基于整个工程构建架构树”



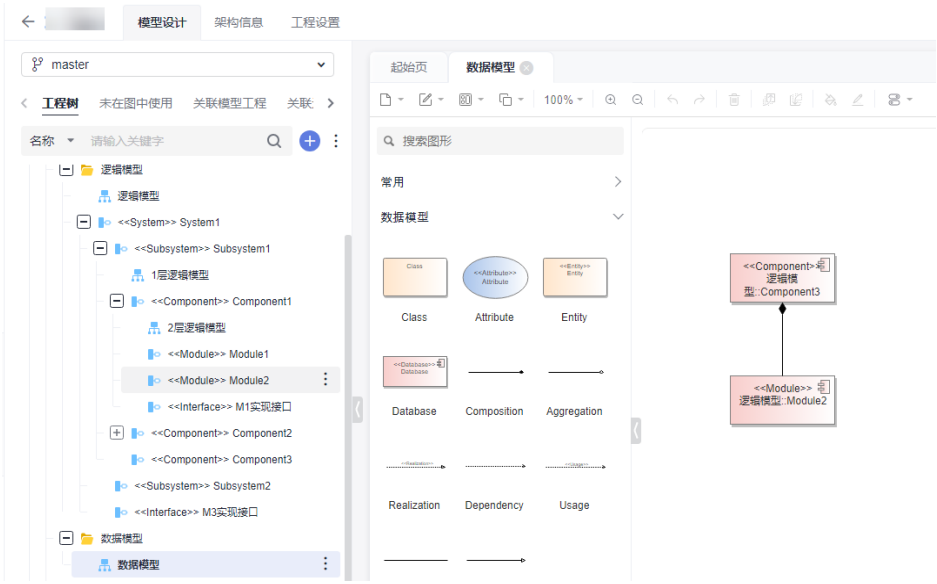
展示不匹配架构方案的元素：在架构信息树中展示出不符合架构方案规则的元素节点，并以红色文本加叹号的方式在架构信息树中标记出。

如下图所示，在方案中，component下面是未配置interface元素，但是在工程树上组件下面挂了接口元素，导致该组件与接口存在了不符合架构方案规则 的上下级关系，当开启了“展示不匹配架构方案的元素”时，该接口就会以红色状态显示在架构信息树中。

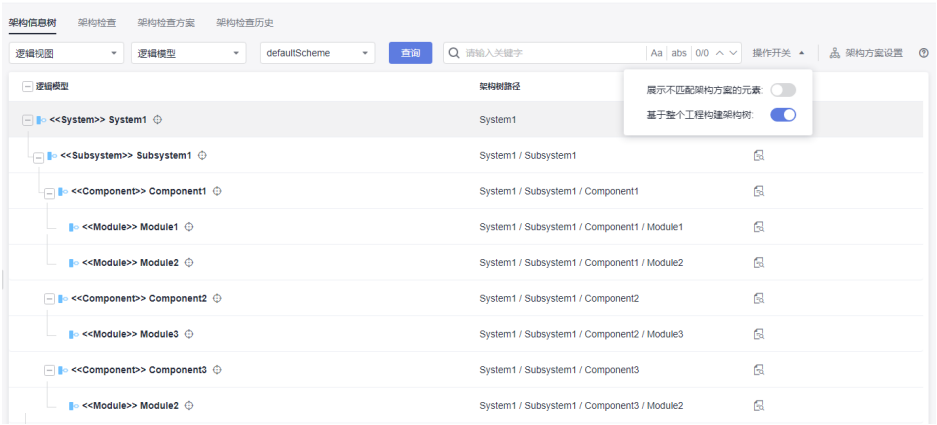


基于整个工程构建架构树：不局限于当前模型类型图中的关系参与构树，会将元素在所有模型图中的连线关系、包含关系算到关系集合中，按架构方案规则生成架构信息树。

如下图所示，在数据模型中对Component3与Module2建立符合架构规则的组合关系。



启用“基于整个工程构建架构树”开关后，上述非逻辑模型图中的组合关系参与逻辑模型的构树。



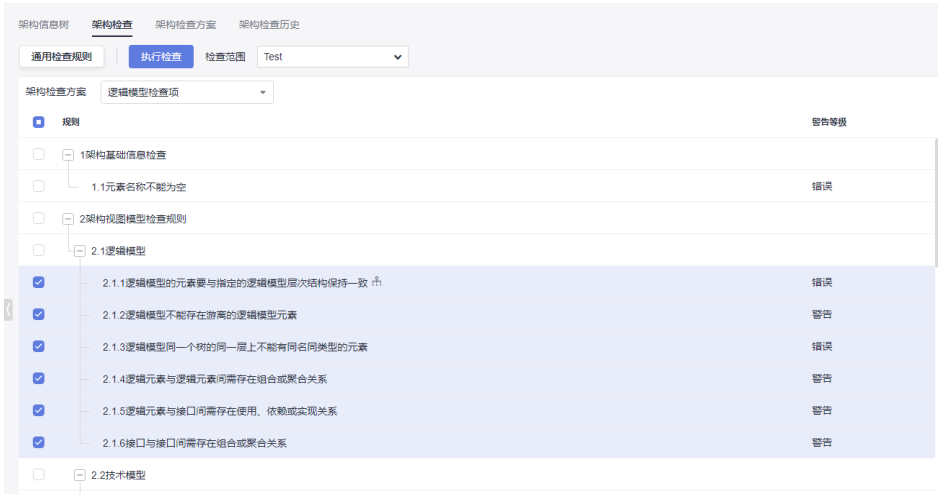
2.7.2 架构检查方案

架构检查方案功能是基于架构检查中的规则项，设置一个检查规则集合，可将该检查集合设置为架构检查中默认启用的检查规则集，该检查会生成检查任务到架构检查历史中。

单击“新增架构检查方案”，输入方案名称，规则集中的检查项来源基于通用检查规则，选择要配置到规则集中的方案。

新建名称为逻辑模型检查项，勾选对应检查规则。

“架构检查 > 通用检查规则”即可看到新增规则。



2.7.3 架构检查历史

在架构检查历史中可以查看过往检查的历史记录，默认打开是当前用户的检查记录。在“我的检查”下拉选项中，可以切换显示出所有人的检查结果，也支持按时间过滤查询检查的记录。

在检查历史记录下“查看”可以打开检查规则结果通过情况。

在是否通过详情下“查看”具体元素定位‘修复指导和关系查询。

2.8 架构检查

2.8.1 通用检查规则

2.8.1.1 架构基础信息检查

1.1 元素名称不能为空

详细描述

建模设计的元素名称不能为空，如果存在名称为空的元素，在检查结果中都会列出。

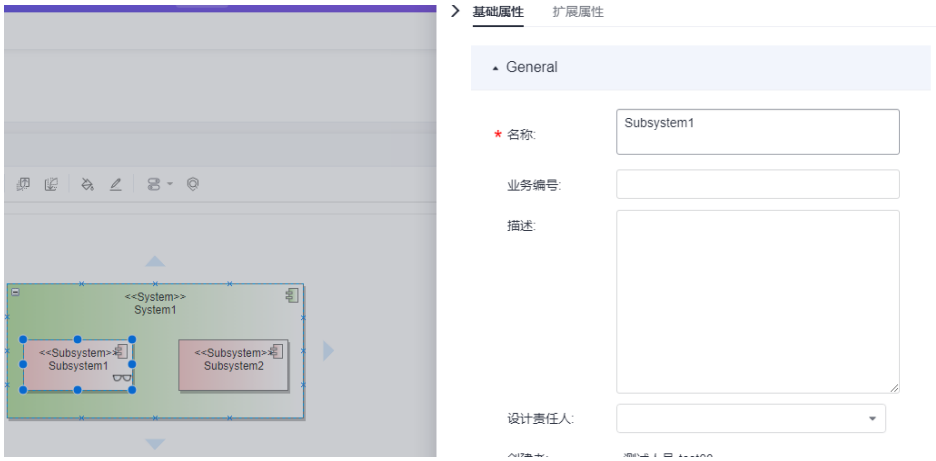
检查范围

在图上创建的元素在工程树中出现对应的节点，即为建模元素，都在被检查范围内。

如何检查

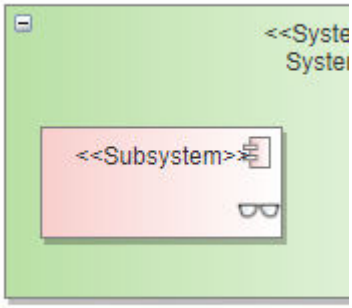
查询模型工程内所有建模元素，检查出名称为空元素。

正确示例



错误示例

Subsystem名称为空。



2.8.1.2 架构视图模型检查规则

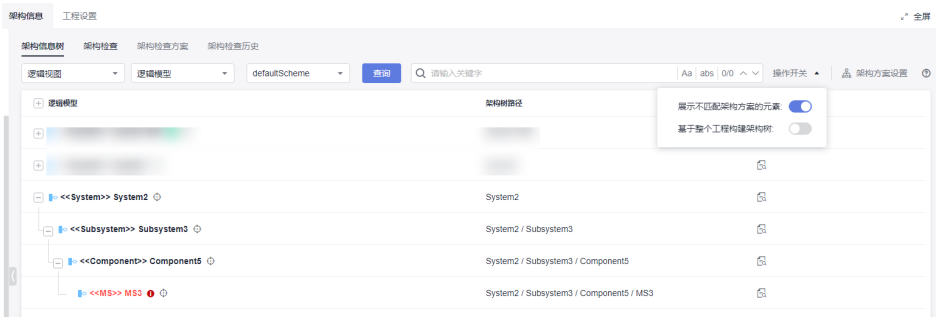
2.8.1.2.1 逻辑模型

2.1.1 逻辑模型的元素要与指定的逻辑模型层次结构保持一致

详细描述

在逻辑模型中创建逻辑元素，逻辑元素在架构树中与上下级元素的关系层级结构要与逻辑模型架构方案配置定义的层次结构一致，即该逻辑元素与上层父级元素、下层子级元素的父子关系（也称上下层级关系）、以及它们之间的连线关系和方向指向，都要与层级规则中定义的保持一致。

该规则项检查出的是架构信息树中按模型图构树后显示红色叹号的告警元素。



检查范围

当前模型工程所有逻辑模型图中的逻辑元素。

逻辑元素定义：“工程设置 > 元素构造型”下，绑定到4+1视图：逻辑模型的基础构造型与自定义构造型元素以及逻辑模型架构方案配置的构造型。

包括：在逻辑模型图上创建出来的逻辑元素，引用到逻辑模型中的逻辑元素（包含关联空间中的引用的逻辑元素）。

如何检查

查询基于模型图（只有逻辑模型图内的逻辑元素参与构树）构出的逻辑模型架构树，找出与架构方案不匹配（标红）的元素。

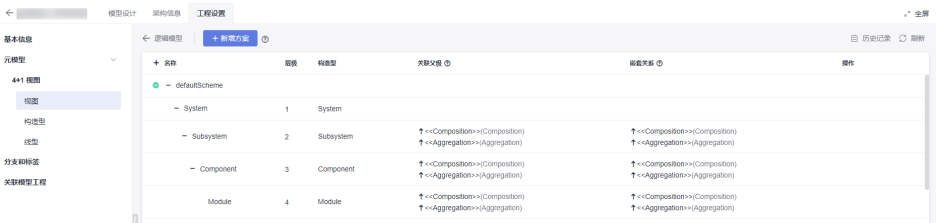
正确示例

架构层级规则示例：

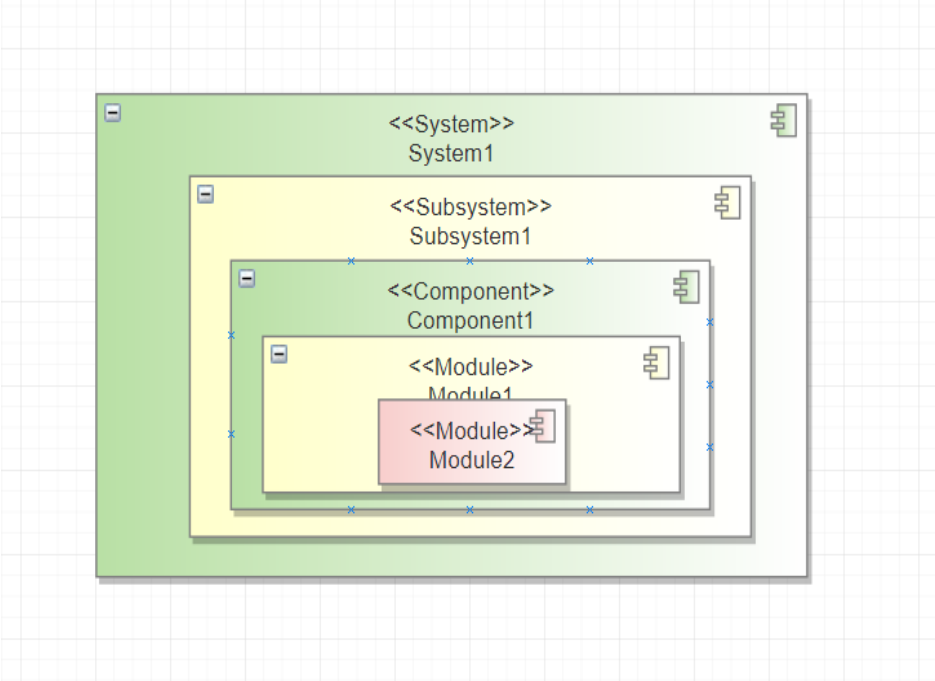
关联父级：配置的是当前层级元素与上一层级的元素之间的连线类型和父子关系指向。

嵌套：是否支持当前类型的元素与同类型元素建立关系。

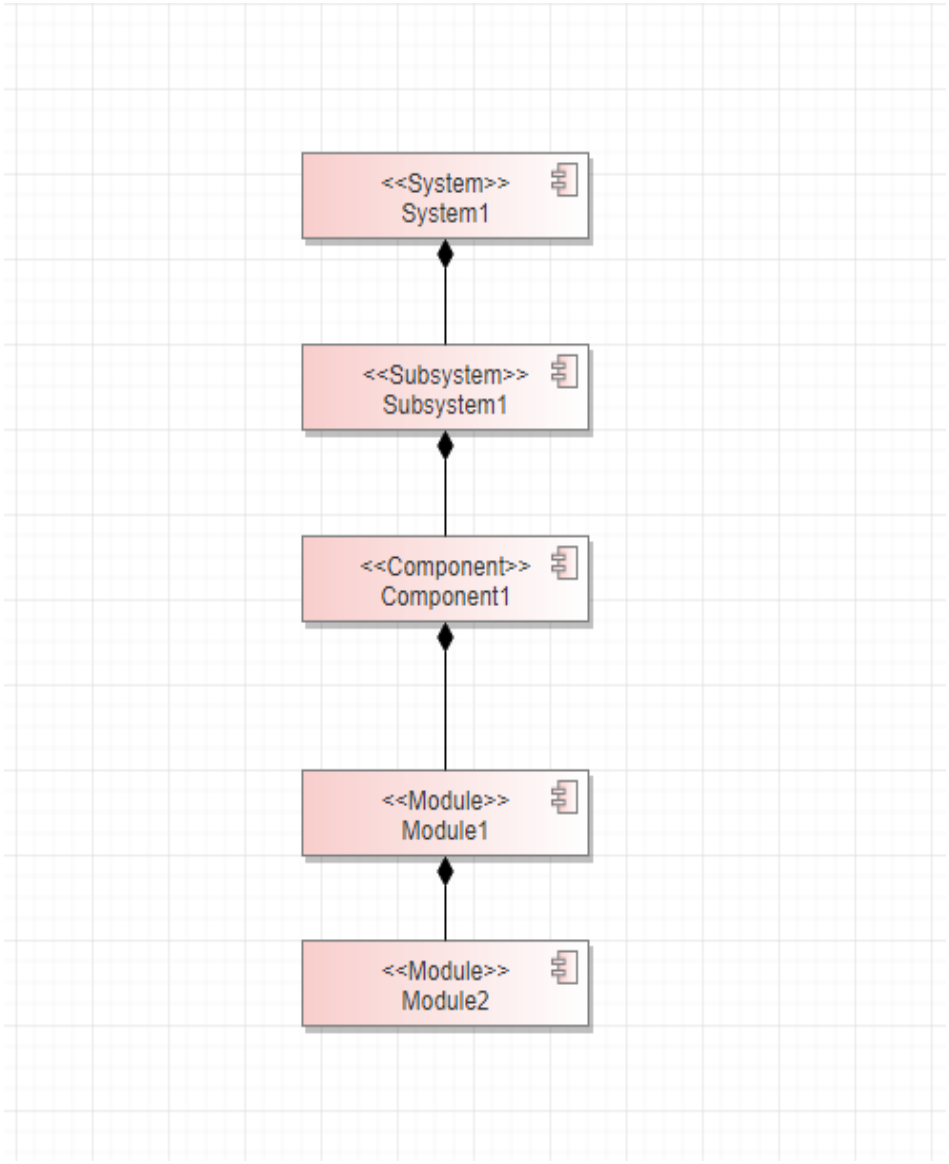
嵌套关系：当前类型的元素与同类型元素建立连线关系类型，指向关系默认为子指向父（即被指向的一方为子）。



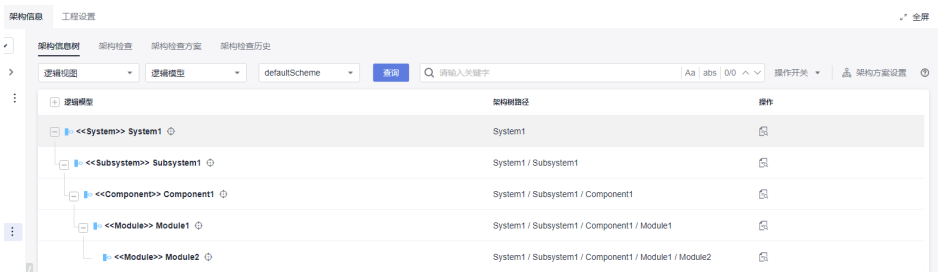
符合架构层级方案的画法示例1--包含的父子关系：



符合架构层级方案的画法示例2–连线的父子关系：



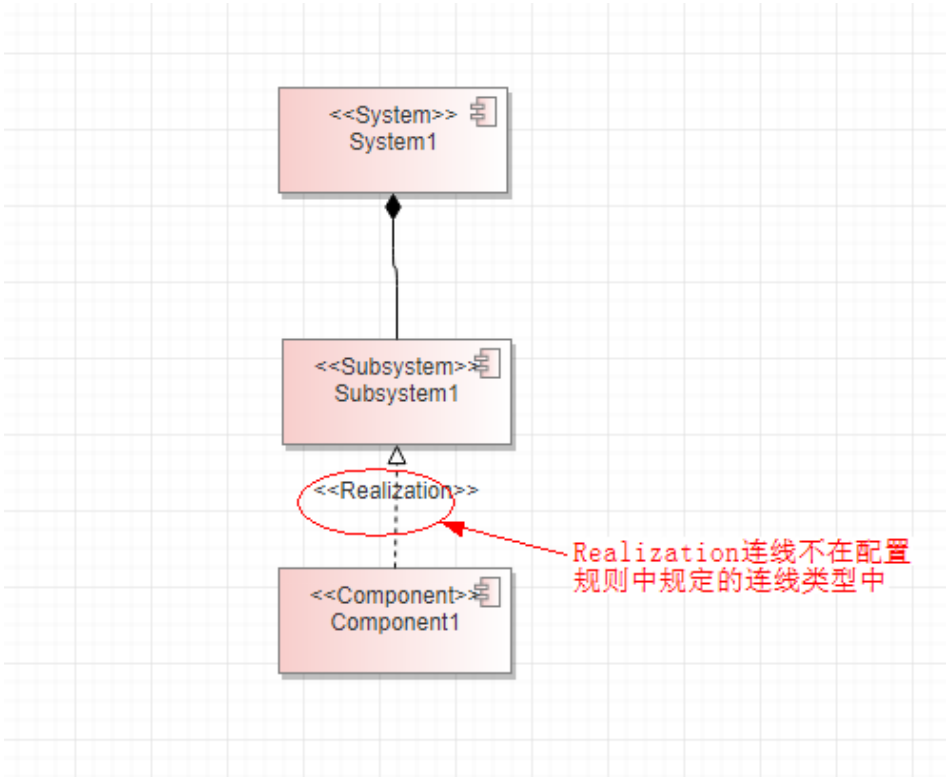
架构信息树展示结果：



当架构树上没有标红元素，就没有2.1.1的检查错误结果。

错误示例

连线类型不对。



架构信息树中报红。

架构检查结果。

2.1.2 逻辑模型不能存在游离的逻辑模型元素

详细描述

逻辑模型元素不能独立存在于逻辑架构树之外，必须要与架构树上的逻辑元素建立关联关系。

检查范围

当前模型工程所有逻辑模型图中的逻辑元素。

（逻辑元素定义：工程设置 > 构造型下，绑定到4+1视图：逻辑模型的基础构造型与自定义构造型元素以及逻辑模型架构方案配置的构造型）。

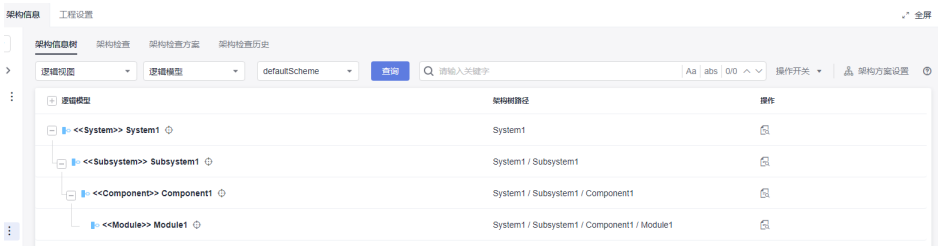
包括：在逻辑模型图上创建出来的逻辑元素，引用到逻辑模型中的逻辑元素（包含关联空间中的引用的逻辑元素）；

如何检查

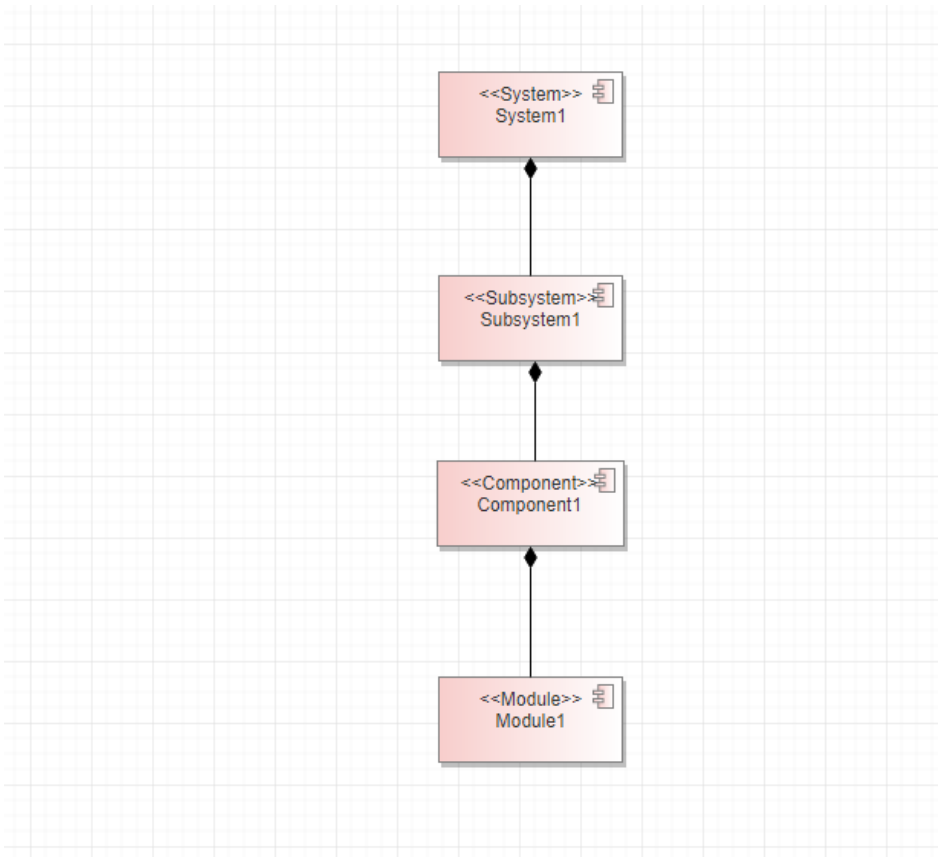
查询基于模型图（只有逻辑模型图内的逻辑元素参与构树）并展示不匹配元素构出的逻辑模型架构树，找出所有逻辑元素中不在架构树中的逻辑元素。

正确示例

按逻辑架构方案构建的架构信息树：

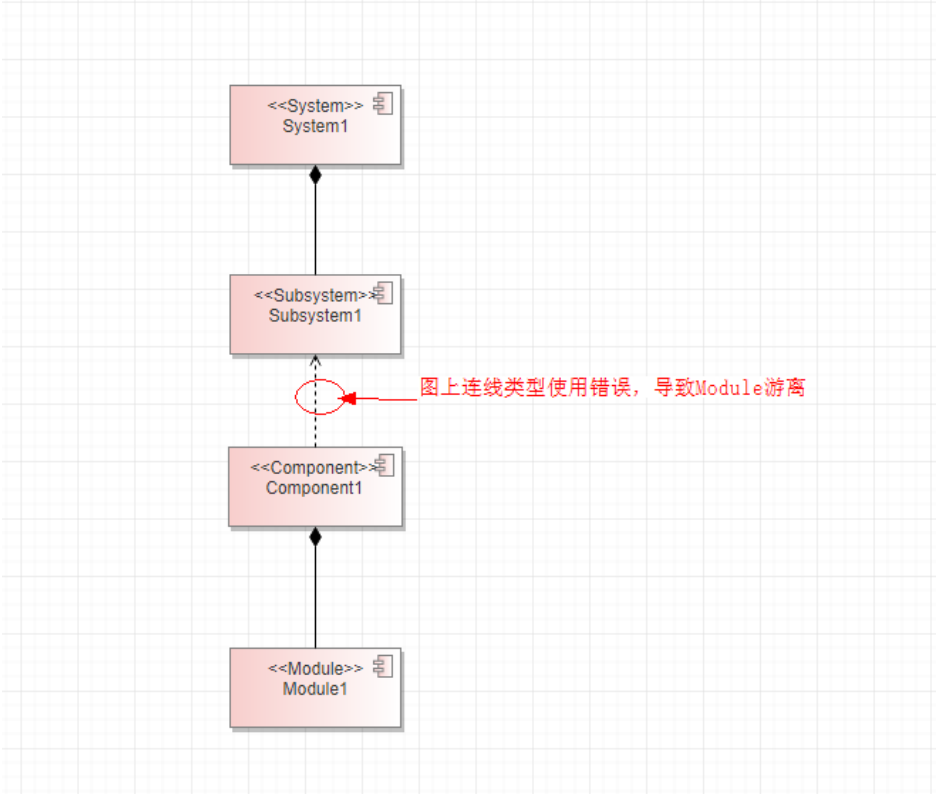


模型图示例：



错误示例

场景一：Component1与父级元素连线错误，导致子元素Module1变成游离的元素。



检查结果：

2.1.3 逻辑模型同一个树的同一层上不能有同名同类型的元素

详细描述

在逻辑架构信息树上，同一个父元素节点下面，不能存在类型相同，名称也相同的元素。

检查范围

当前模型工程所有逻辑模型图中的逻辑元素

（逻辑元素定义：工程设置 > 构造型下，绑定到4+1视图：逻辑模型的基础构造型与自定义构造型元素以及逻辑模型架构方案配置的构造型）。

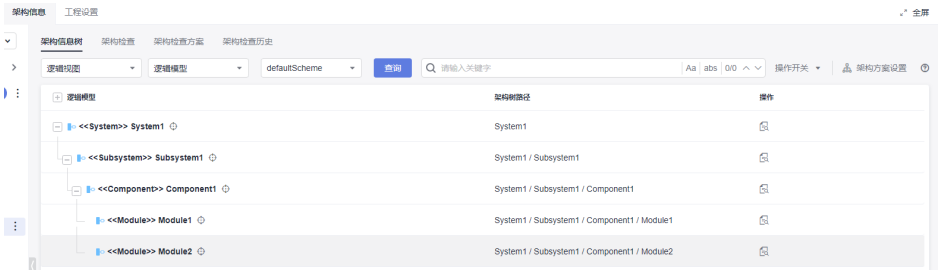
包括：

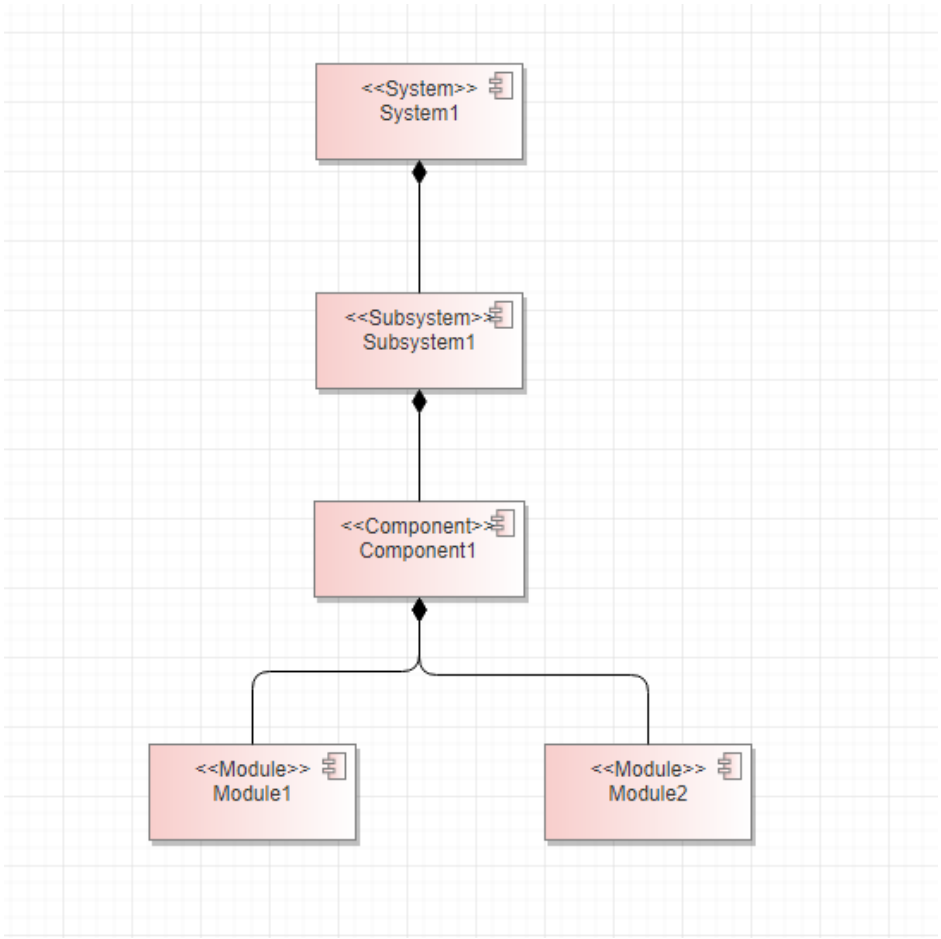
1. 在逻辑模型图上创建出来的逻辑元素；
2. 引用到逻辑模型中的逻辑元素（包含关联空间中的引用的逻辑元素）；

如何检查

查询基于模型图（只有逻辑模型图内的逻辑元素参与构树）构出的逻辑模型架构树，找出同一节点下同名同类型的逻辑元素。

正确示例

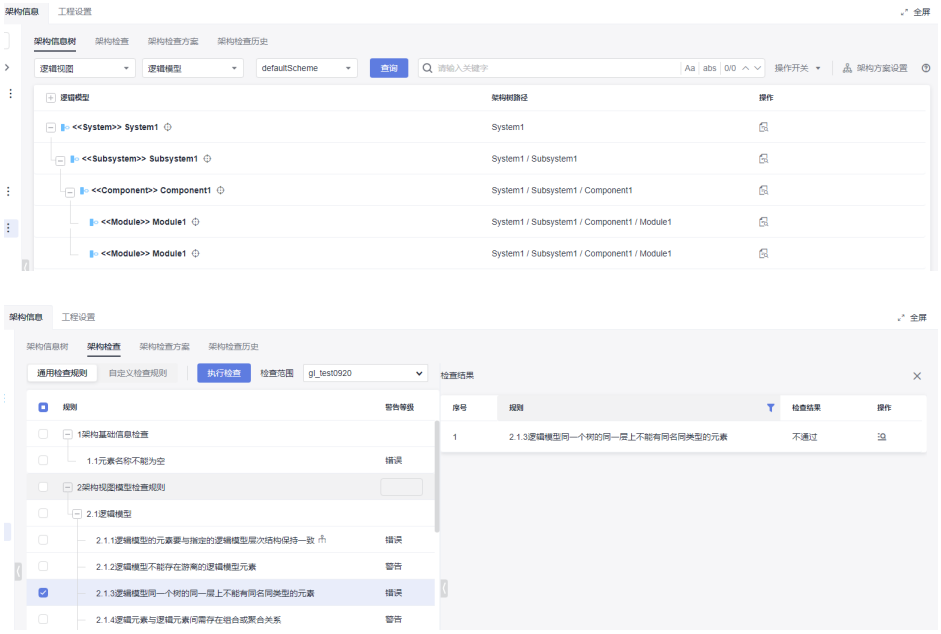




错误示例

场景一：同父元素下面存在同类型且同名称的元素。

按逻辑规则构建的架构信息树，树上不会显示异常：



2.1.4 逻辑元素与逻辑元素间需存在组合或聚合关系

详细描述

逻辑元素与逻辑元素之间如果存在连线关系，必须为组合或者聚合关系。

检查范围

当前模型工程所有逻辑模型图中的逻辑元素

（逻辑元素定义：工程设置 > 构造型下，绑定到4+1视图：逻辑模型的基础构造型与自定义构造型元素以及逻辑模型架构方案配置的构造型）。

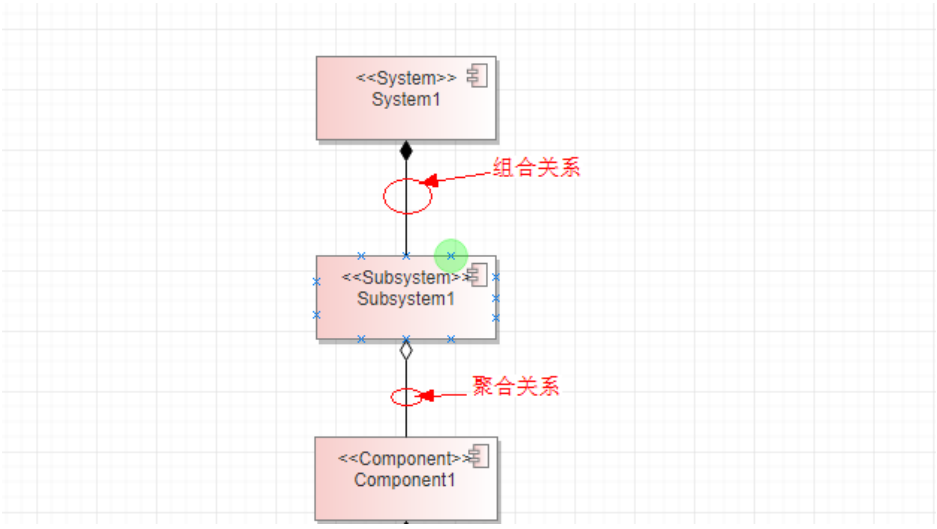
1. 在逻辑模型图上创建出来的逻辑元素之间的连线关系；
2. 引用到逻辑模型中的逻辑元素（包含关联空间中的引用的逻辑元素）之间的连线关系；
3. 不包含接口元素；

如何检查

找出逻辑模型图里的在检查范围内的逻辑元素间存在连线关系但连线关系中没有组合或聚合关系的元素（即使层级规则方案中配置了除组合聚合之外的指定的连线类型，也会检查出来）。

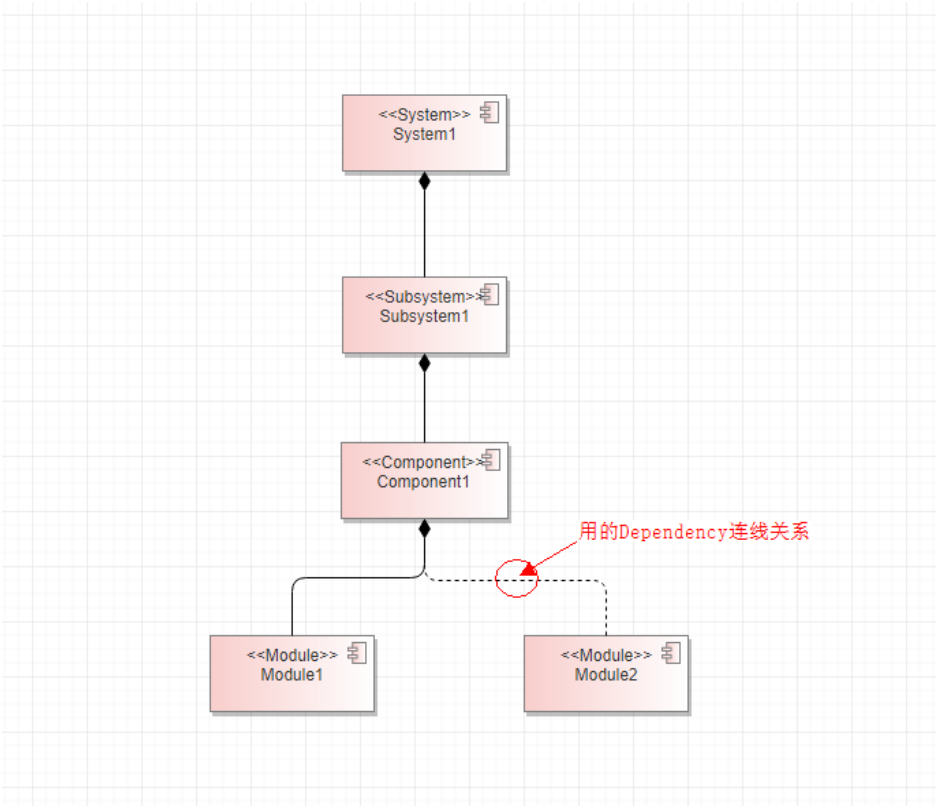
正确示例

组合或聚合关系。



错误示例

场景一：使用非组合或者聚合连线关系。



即使方案中配置了组件与模块之间有使用连线关系，但是该规则依然会检查出来。

2.1.5 逻辑元素与接口间需存在使用、依赖或实现关系

详细描述

逻辑元素与接口之间如果存在连线关系，必须为使用、实现、依赖关系。

检查范围

当前模型工程中的所有符合定义规则的逻辑元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：逻辑模型的基础构造型与自定义构造型元素才认定为逻辑元素）。

1. 在逻辑模型图上创建出来的逻辑元素与接口连线之间的连线关系；
2. 引用到逻辑模型中的逻辑元素（包含关联空间中的引用的逻辑元素）与接口之间的连线关系；
3. 接口只指实体接口，即Interface元素，暴露接口Provided Interface和请求接口Required Interface 非实体接口，不在检查目标中；

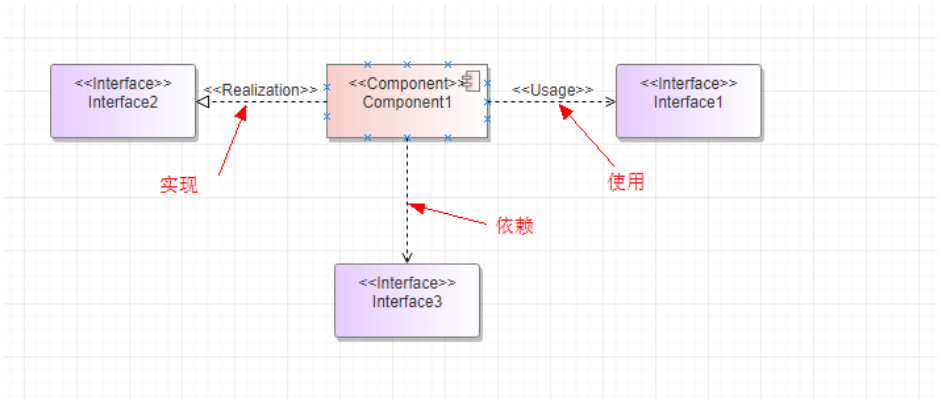
如何检查

找出逻辑模型图里的在检查范围内的逻辑元素与实体接口间存在连线关系但连线关系不是使用、实现、依赖关系的逻辑元素和接口

（即使层级规则方案中配置了除使用、实现、依赖之外的指定的连线类型，也会检查出来）。

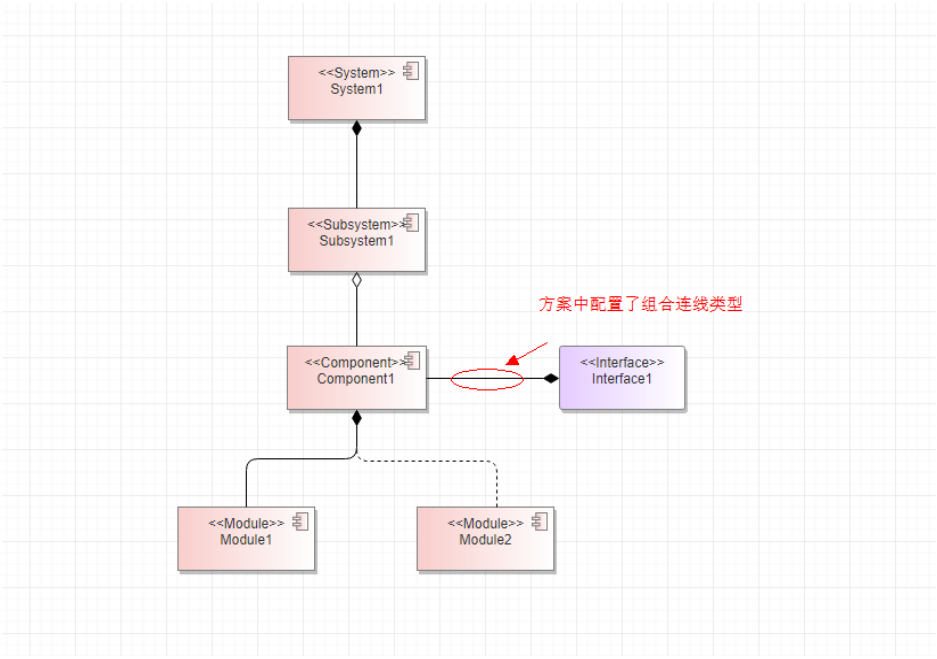
正确示例

使用、实现、依赖关系：



错误示例

场景一：使用非实现、使用、依赖之外的连线关系（即使关系已经配置到方案中）。



即使方案中配置了组件与模块之间有组合连线关系，但是该规则依然会检查出来。

2.1.6 接口与接口间需存在组合或聚合关系

详细描述

接口与接口之间如果存在连线关系，必须为组合或者聚合关系。

检查范围

当前模型工程中的所有符合定义规则的接口（定义规则：工程设置 > 构造型下，绑定到4+1视图：逻辑模型的接口Interface元素）。

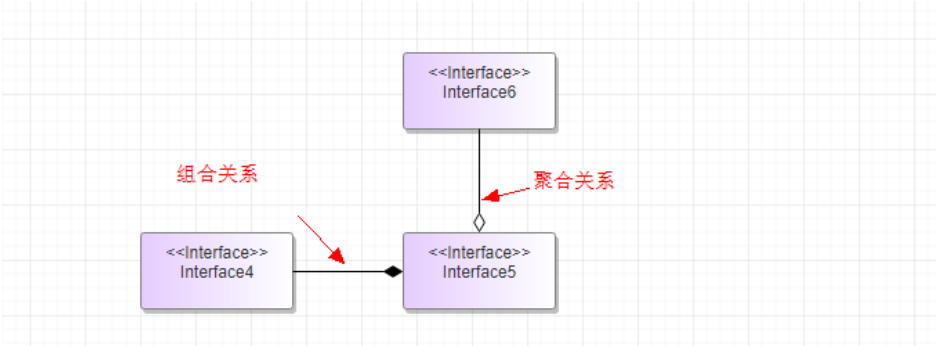
1. 在逻辑模型图上创建出来的接口元素之间的连线关系。
2. 引用到逻辑模型中的接口（包含关联空间中的引用的逻辑元素）之间的连线关系。

如何检查

找出逻辑模型图里的接口间存在连线关系但连线关系中没有组合或聚合关系的接口元素（即使层级规则方案中配置了除组合聚合之外的指定的连线类型，也会检查出来）。

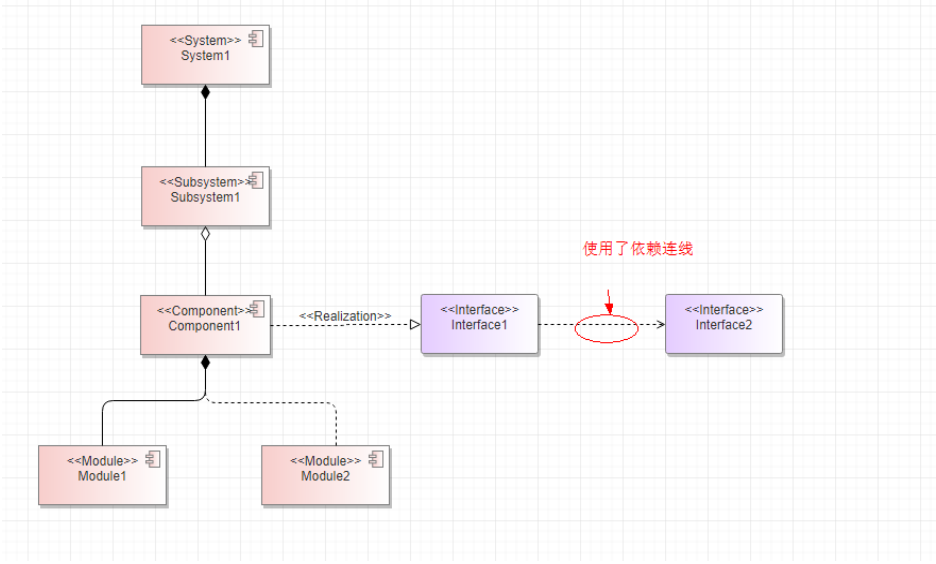
正确示例

组合或聚合关系：



错误示例

场景一：使用非组合或者聚合连线关系。



即使方案中配置了接口与接口之间有使用连线关系，但是该规则依然会检查出来。

2.8.1.2.2 技术模型

2.2.1 技术模型的元素要与指定的技术模型层次结构保持一致

详细描述

在技术模型中创建技术元素，技术元素在架构树中与上下级元素的关系层级结构要与技术模型架构方案配置定义的层次结构一致，即该技术元素与上层父级元素、下层子级元素的父子关系（也称上下层级关系）、以及它们之间的连线关系和方向指向，都要与层级规则中定义的保持一致。

检查范围

当前模型工程中的所有符合定义规则的技术元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：技术模型的基础构造型与自定义构造型元素才认定为技术元素）。

1. 在技术模型图上创建出来的技术元素；
2. 引用到技术模型中的技术元素（包含关联空间中的引用的技术元素）。

如何检查

查询基于模型图（只有技术模型图内的技术元素参与构树）构出的技术模型架构树，找出与架构方案不匹配（标红）的元素。

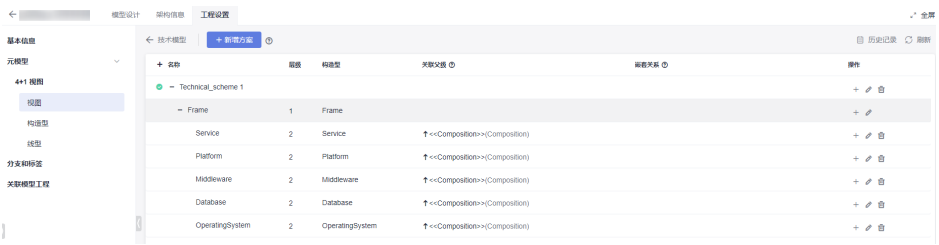
正确示例

架构层级规则示例：

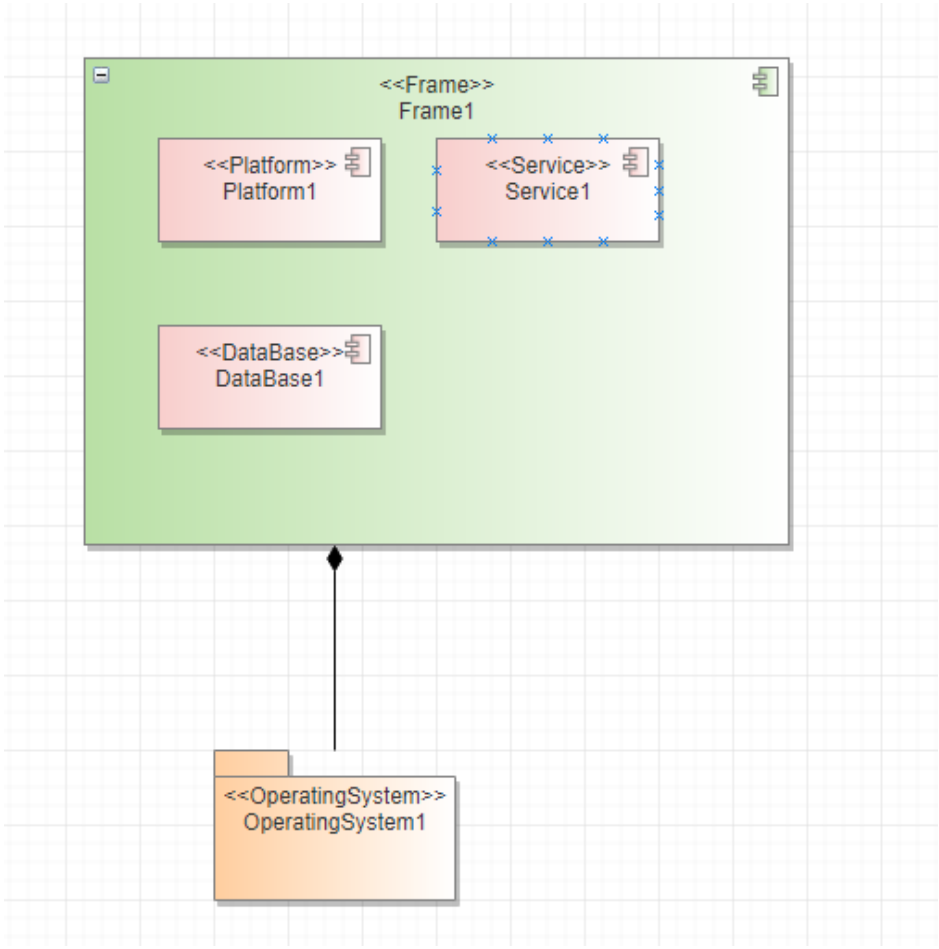
关联父级：配置的是当前层级元素与上一层级的元素之间的连线类型和父子关系指向。

嵌套：是否支持当前类型的元素与同类型元素建立关系。

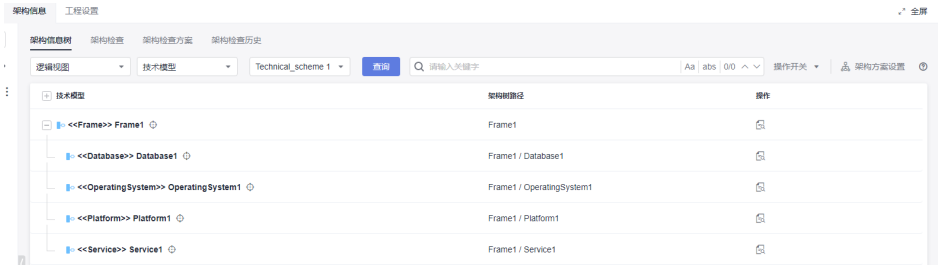
嵌套关系：当前类型的元素与同类型元素建立连线关系类型，指向关系默认为父指向子（即被指向的一方为子）。



图中画法示例1–包含的父子关系和连线关系：



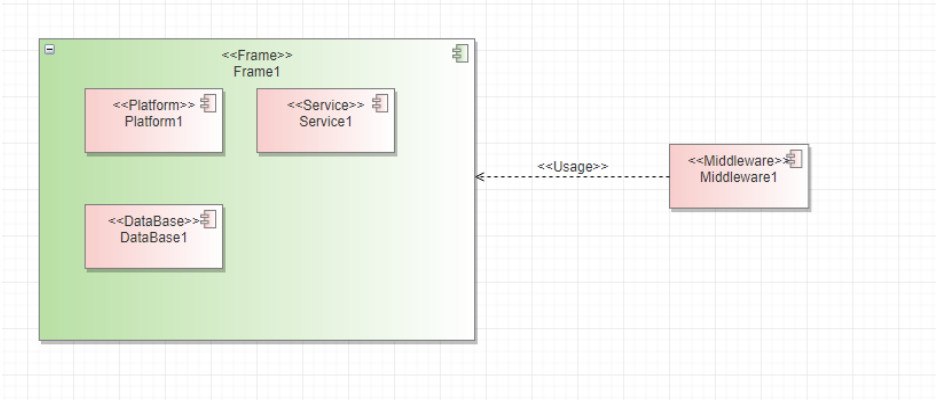
架构信息树展示结果：



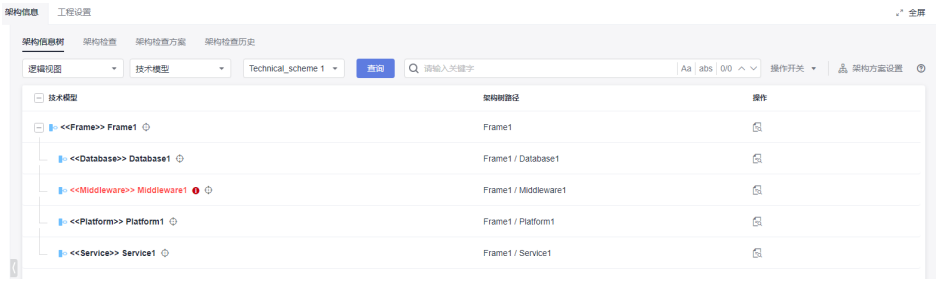
当架构树上没有标红元素，就没有2.2.1的检查错误结果。

错误示例

错误示例-连线类型不对：



架构信息树中报红：



架构检查结果：

2.2.2 技术模型不能存在游离的元素

详细描述

技术模型元素不能独立存在于技术架构树之外，必须要与架构树上的技术元素建立关联关系。

检查范围

当前模型工程中的所有符合定义规则的技术元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：技术模型的基础构造型与自定义构造型元素才认定为技术元素）。

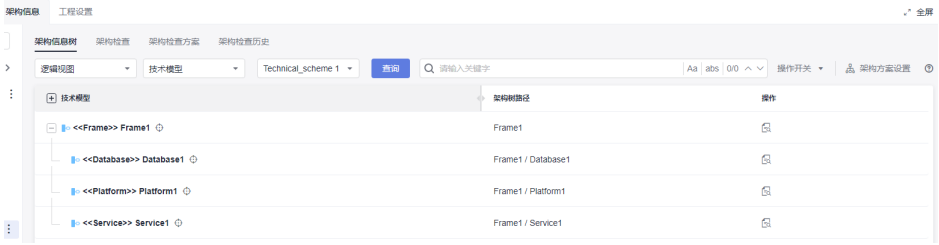
1. 在技术模型图上创建出来的技术元素；
2. 引用到技术模型中的技术元素（包含关联空间中的引用的技术元素）；

如何检查

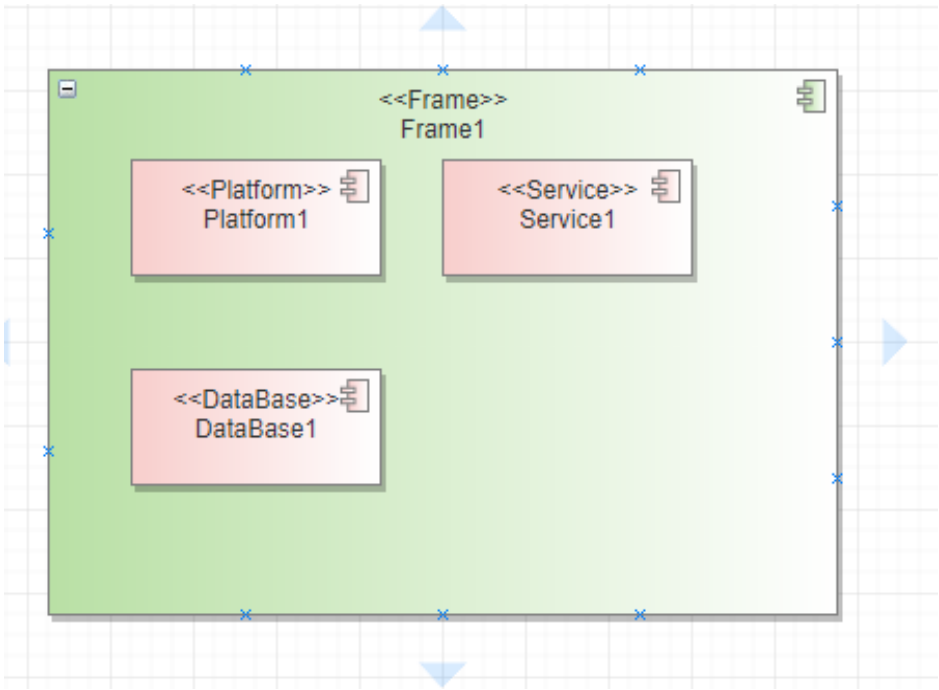
查询基于模型图（只有技术模型图内的技术元素参与构树）并展示不匹配元素构出的技术模型架构树，找出所有技术元素中不在架构树中的技术元素。

正确示例

按逻辑规则构建的架构信息树：

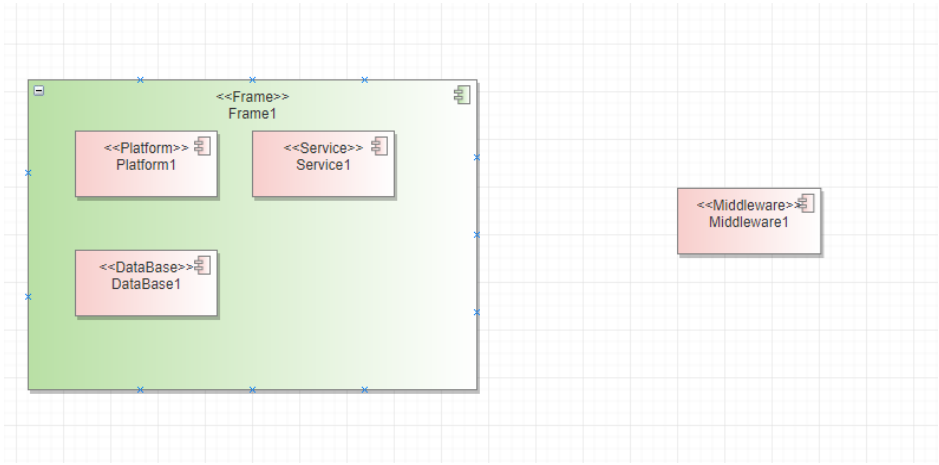


模型图示例：



错误示例

场景一：独立存在在技术模型图上的技术模型元素。



检查结果：

2.2.3 技术模型同一个树的同一层上不能有同名同类型的元素

详细描述

在同一棵技术架构信息树上，在同一个父元素节点下面，不能存在扩展类型相同，并且名称也相同的元素。

检查范围

当前模型工程中的所有符合定义规则的技术元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：技术模型的基础构造型与自定义构造型元素才认定为技术元素）。

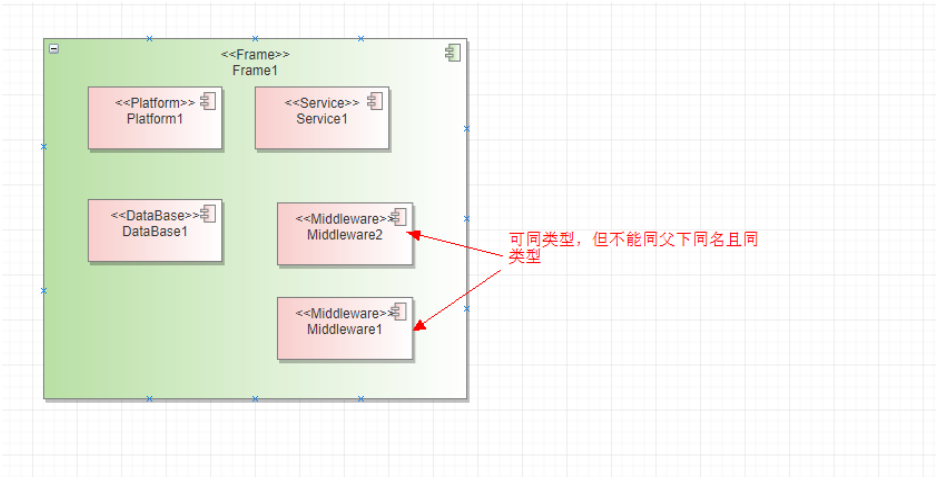
1. 在技术模型图上创建出来的技术元素；
2. 引用到技术模型中的技术元素（包含关联空间中的引用的技术元素）；

如何检查

查询基于模型图（只有技术模型图内的技术元素参与构树）构出的技术模型架构树，找出同一节点下同名同类型的技术元素。

正确示例

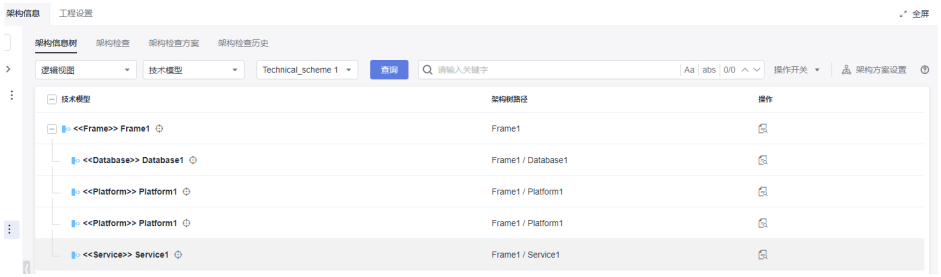
同一父节点下可以同类型或者同名，但是不能同类型且又同名的。



错误示例

场景一：同父元素下面存在同类型且同名称的元素。

按逻辑规则构建的架构信息树，树上不会显示异常：



2.2.4 技术元素与逻辑元素需存在技术元素实现逻辑元素或者逻辑元素使用或依赖技术元素关系

详细描述

技术模型元素与逻辑元素之间如果存在连线关系，必须为使用或者实现类型的连线关系，不能存在其它连线类型的关系。

检查范围

当前模型工程中的所有符合定义规则的技术元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：技术模型的基础构造型与自定义构造型元素才认定为技术元素）。

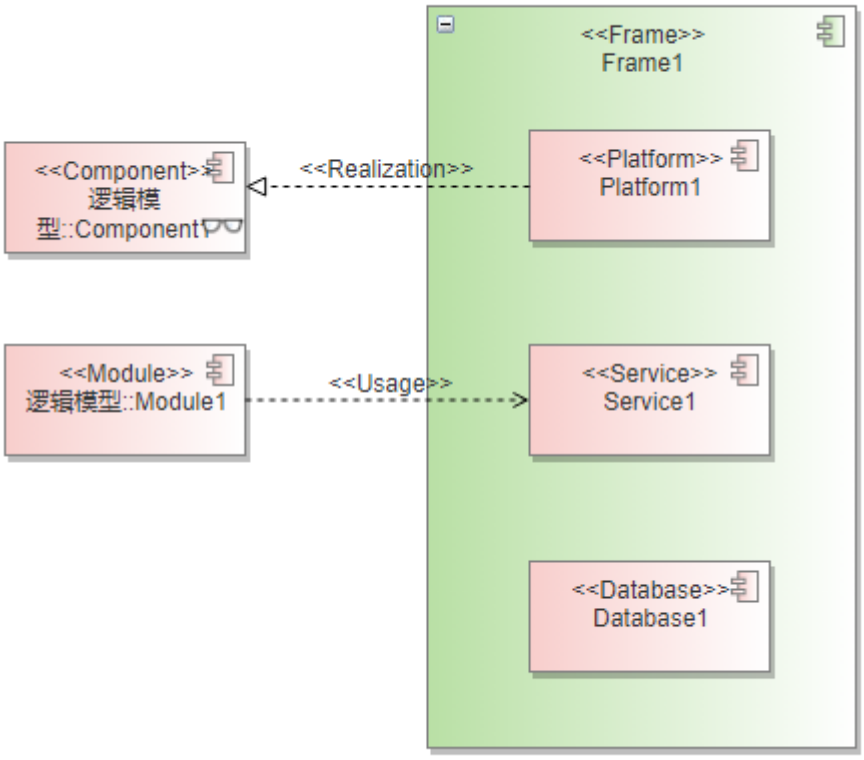
1. 在技术模型图上创建出来的技术元素；
2. 引用到技术模型中的技术元素（包含关联空间中的引用的技术元素）；
3. 从逻辑模型中引用（Link）到技术模型中的逻辑元素（逻辑元素的定义参考2.1.1 逻辑模型检查范围定义）。

如何检查

找出技术元素与逻辑元素中存在连线关系但不是 技术元素实现逻辑元素 或者 逻辑元素使用或依赖技术元素关系的元素。

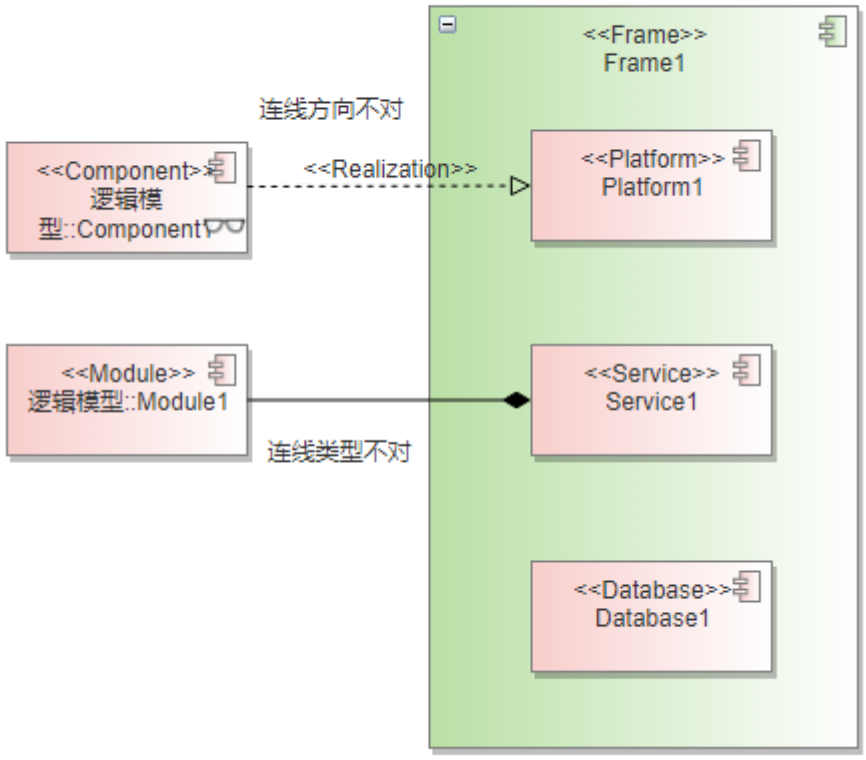
正确示例

模型图示例：正常为逻辑元素指向技术元素，用Usage连线；技术元素指向逻辑元素，用Realization连线。



错误示例

场景一：技术元素与逻辑元素连线类型不对或者连线类型正确但是指向不对。



检查结果：

连线方向不对和连线类型不对的两端的元素都会被检查出来。

2.8.1.2.3 代码模型

2.3.1 代码模型的元素要与指定的代码模型层次结构保持一致

详细描述

在代码模型中创建代码元素，代码元素在架构树中与上下级元素的关系层级结构要与代码模型架构方案配置定义的层次结构一致，即该代码元素与上层父级元素、下层子级元素的父子关系（也称上下层级关系）、以及它们之间的连线关系和方向指向，都要与层级规则中定义的保持一致。

检查范围

当前模型工程中的所有符合定义规则的代码模型元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：代码模型的基础构造型与自定义构造型元素才认定为代码模型元素）。

1. 在代码模型图上创建出来的代码模型元素；
2. 引用到代码模型中的代码元素（包含关联空间中的引用的代码元素）；

如何检查

查询基于代码模型图构出的代码模型架构树，找出与架构方案不匹配（标红）的元素。

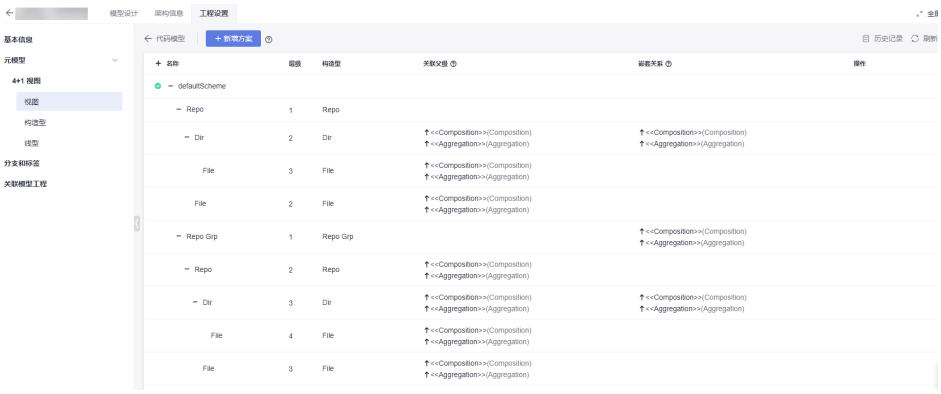
正确示例

架构层级规则示例：

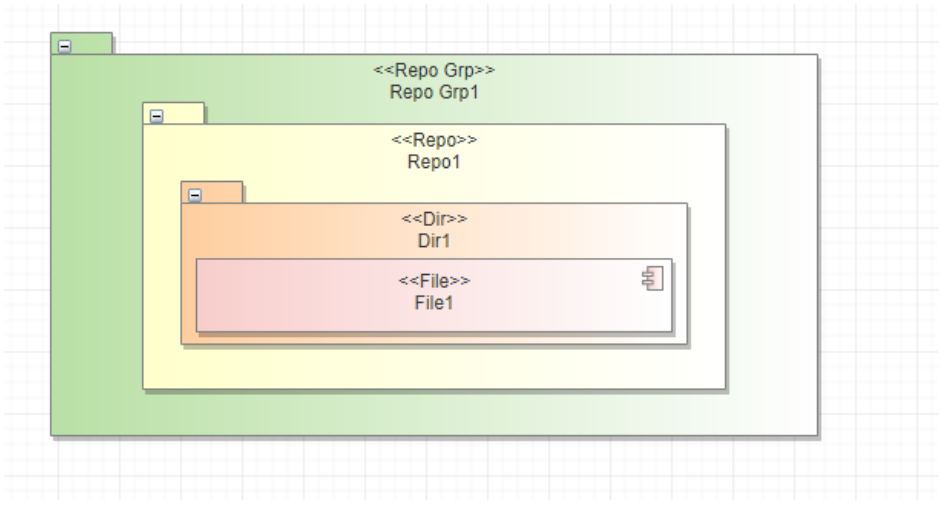
关联父级：配置的是当前层级元素与上一层级的元素之间的连线类型和父子关系指向。

嵌套：是否支持当前类型的元素与同类型元素建立关系。

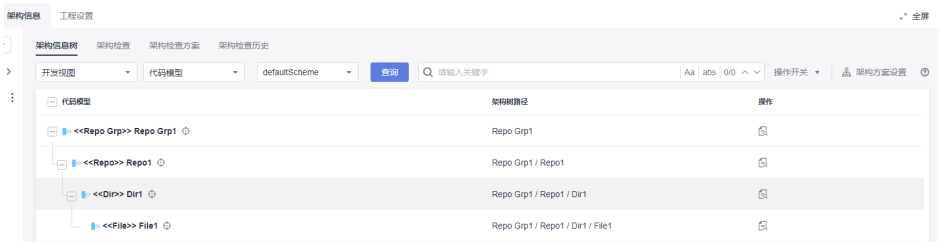
嵌套关系：当前类型的元素与同类型元素建立连线关系类型，指向关系默认为父指向子（即被指向的一方为子）。



图中画法示例1--包含的父子关系：



架构信息树展示结果：

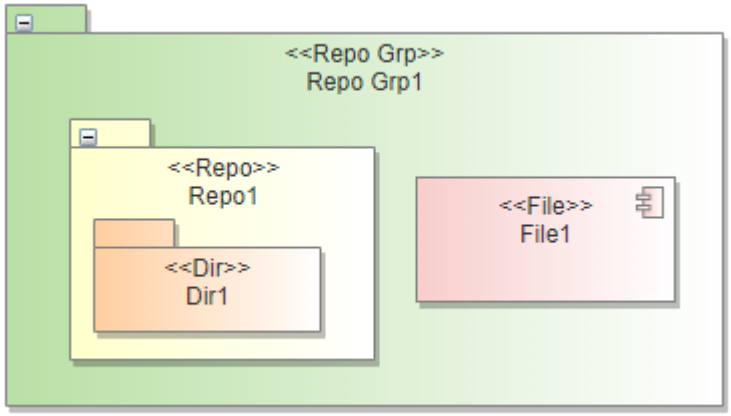


当架构树上没有标红元素，就没有2.3.1的检查错误结果。

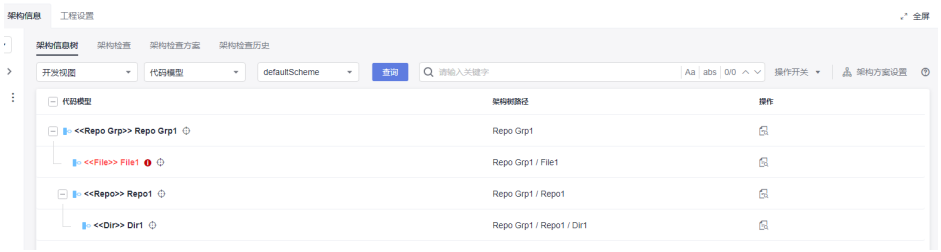
错误示例

场景一：方案中没有配置子节点，但是在画图设计中绘制了子节点。

File画在Repo Grp下是子节点，但方案中Repo Grp没有配置File为子节点。



架构信息树中报红：



架构检查结果：

2.3.2 代码模型不能存在游离的代码模型元素

详细描述

代码模型元素不能独立存在于代码架构树之外，必须要与架构树上的代码元素建立关联关系。

检查范围

当前模型工程中的所有符合定义规则的代码模型元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：代码模型的基础构造型与自定义构造型元素才认定为代码模型元素）。

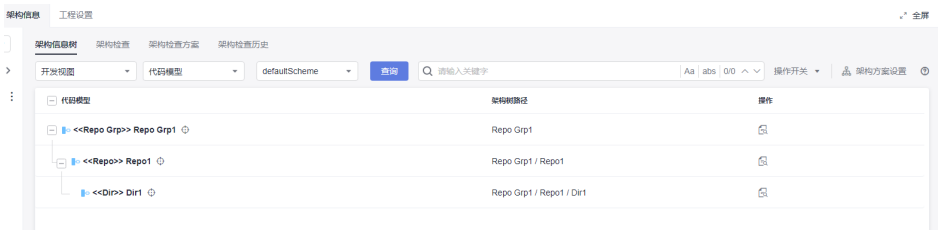
1. 在代码模型图上创建出来的代码模型元素；
2. 引用到代码模型中的代码元素（包含关联空间中的引用的代码元素）；

如何检查

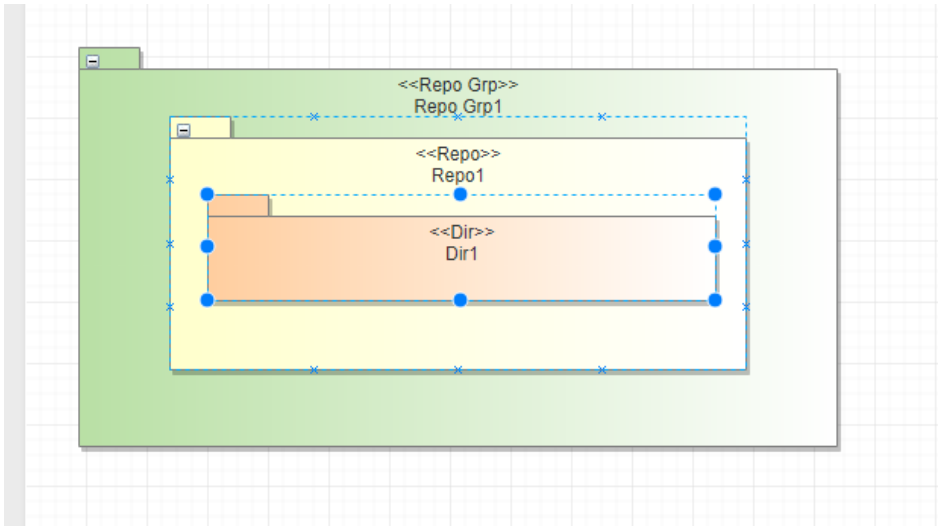
查询基于模型图（只有代码模型图内的代码元素参与构树）并展示不匹配元素构出的代码模型架构树，找出所有代码元素中不在架构树中的代码元素。

正确示例

按代码架构方案构建的架构信息树：

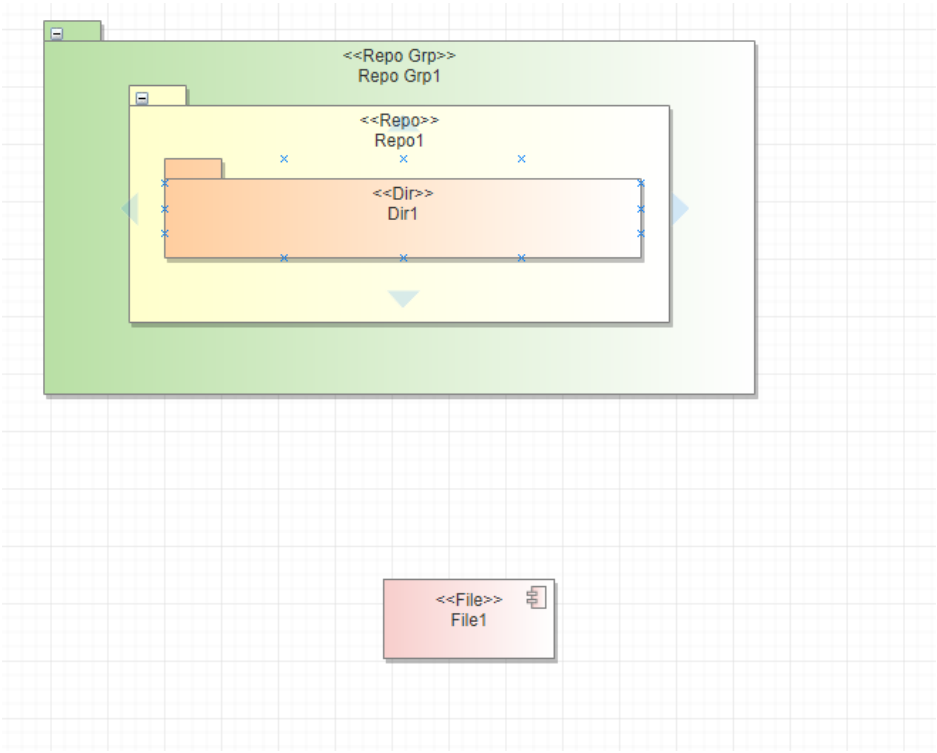


模型图示例：

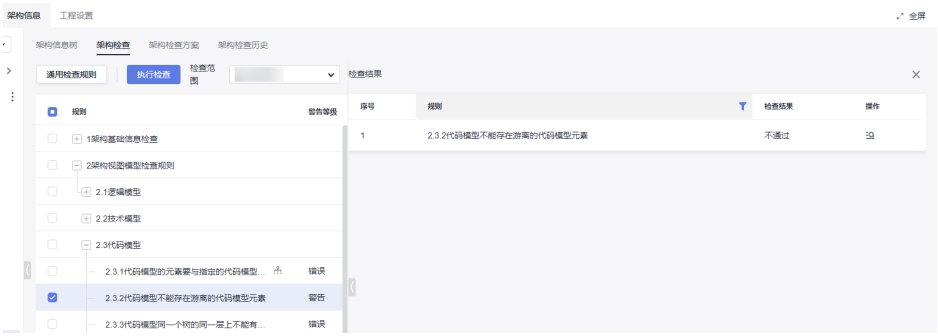


错误示例

场景一：存在没有连线且没有与包含关系的独立元素。



检查结果：



2.3.3 代码模型同一个树的同一层上不能有同名同类型的元素

详细描述

在同一棵代码架构信息树上，在同一个父元素节点下面，不能存在类型相同，并且名称也相同的元素。

检查范围

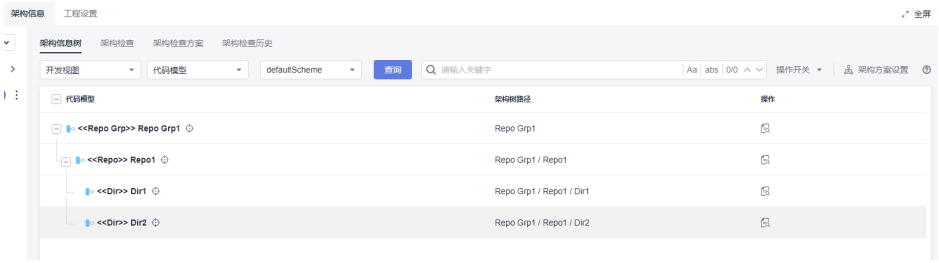
当前模型工程中的所有符合定义规则的代码模型元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：代码模型的基础构造型与自定义构造型元素才认定为代码模型元素）。

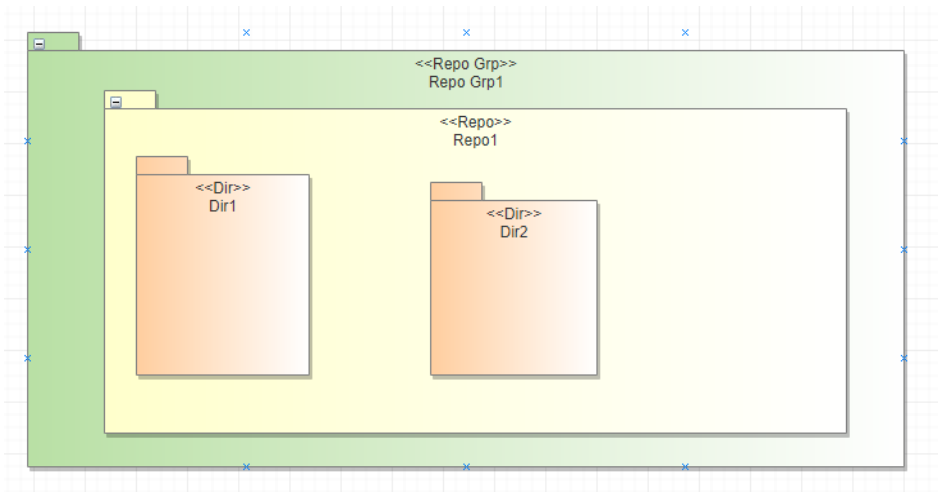
1. 在代码模型图上创建出来的代码模型元素；
2. 引用到代码模型中的代码元素（包含关联空间中的引用的代码元素）；

如何检查

查询基于代码模型图（只有代码模型图内的代码元素参与构树）构出的代码模型架构树，找出同一节点下同名同类型的代码元素。

正确示例

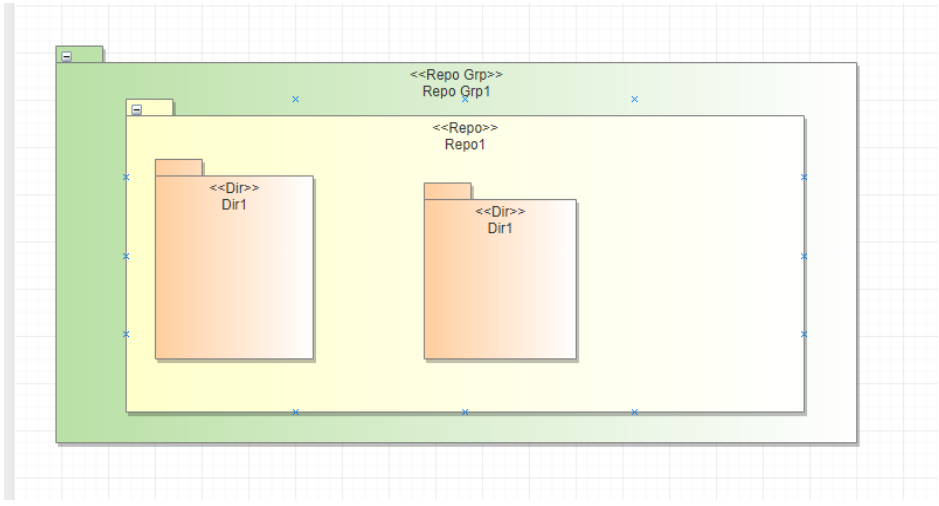


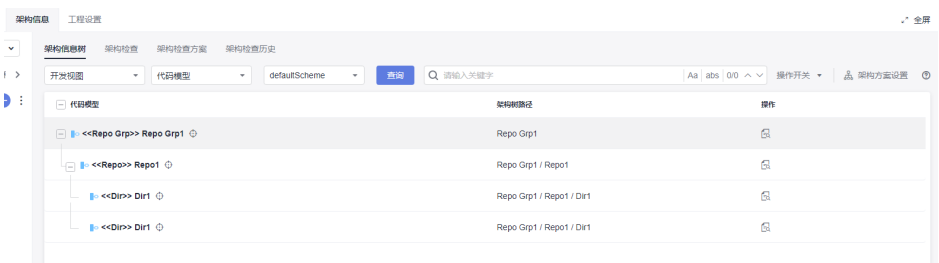


错误示例

场景一：同父元素下面存在同类型且同名称的元素。

按逻辑规则构建的架构信息树，树上不会显示异常。





检查结果：

2.3.4 代码元素与逻辑元素只能是 manifest 关系，且代码元素只能对应一个逻辑元素

详细描述

代码元素与逻辑元素之间的连线类型只能使用manifest连线，且指向方向由代码元素指向逻辑元素；一个代码元素只能连到一个逻辑元素上，而逻辑元素可以连线多个代码元素，即由多个代码元素指向构成。

检查范围

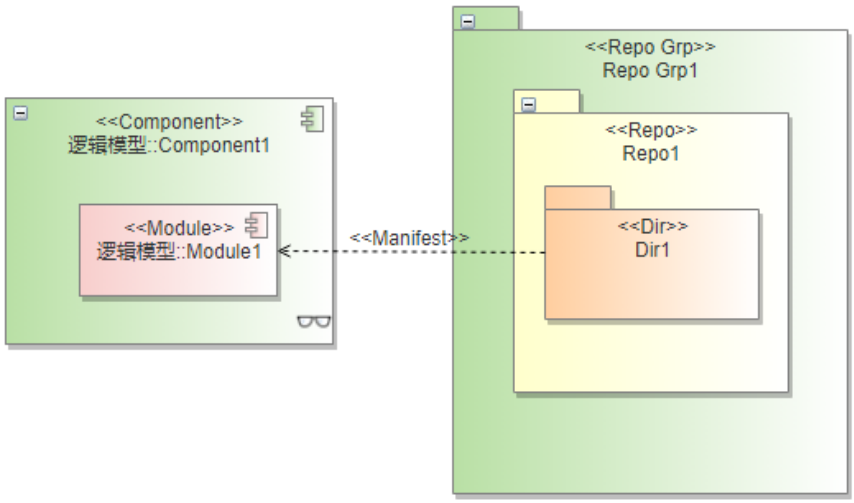
当前模型工程中的所有符合定义规则的代码模型元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：代码模型的基础构造型与自定义构造型元素才认定为代码模型元素）。

1. 在代码模型图上创建出来的代码模型元素；
2. 引用到代码模型中的代码元素（包含关联空间中的引用的代码元素）；

如何检查

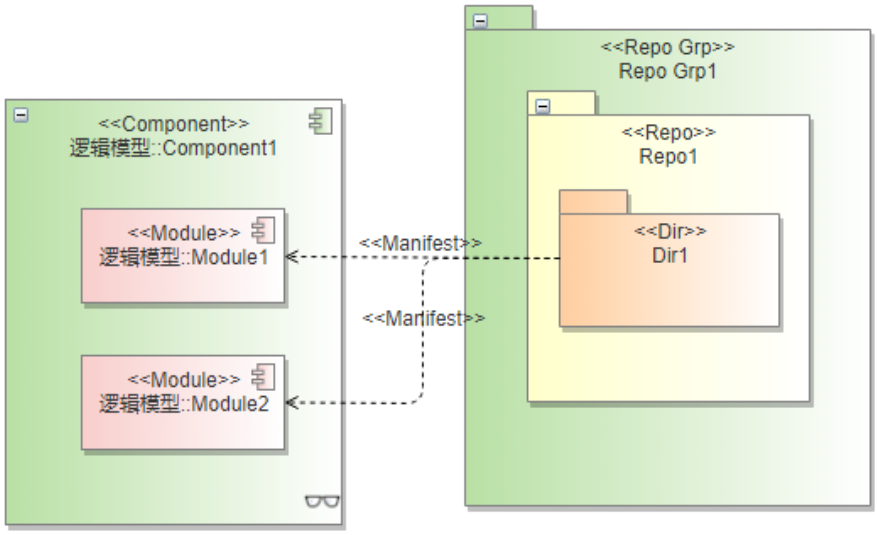
检查所有的代码模型图中的代码元素与逻辑元素之间的连线关系是否为manifest连线关系，并检查代码元素是否只与一个逻辑元素有manifest关系，如果有2个及以上的逻辑元素则不合规则，会列到检查结果中。

正确示例



错误示例

场景一：一个代码元素对应到两个及两个以上的逻辑元素（一对多）。



架构规则检查结果，列出不符合检查项的代码元素：

2.3.5 逻辑元素至少与一个代码元素存在 manifest 关系

详细描述

逻辑模型中的逻辑元素或从逻辑模型引用到代码模型中的逻辑元素至少要与一个代码元素中间有manifest连线关系。

当前规则支持配置检查类型后，已包含3.1.1的检查项，建议使用2.3.6检查项即可，3.1.1可不再重复检查。

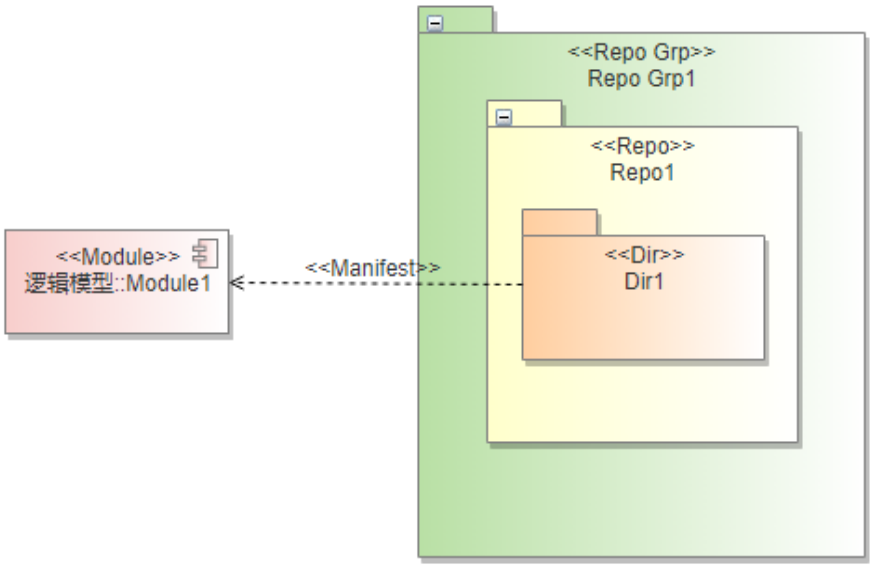
检查范围

1. 在逻辑模型图上创建出来的逻辑模型元素；
2. 引用到代码模型中的逻辑元素；
3. 排除Interface、Provided Interface、Required Interface元素。

如何检查

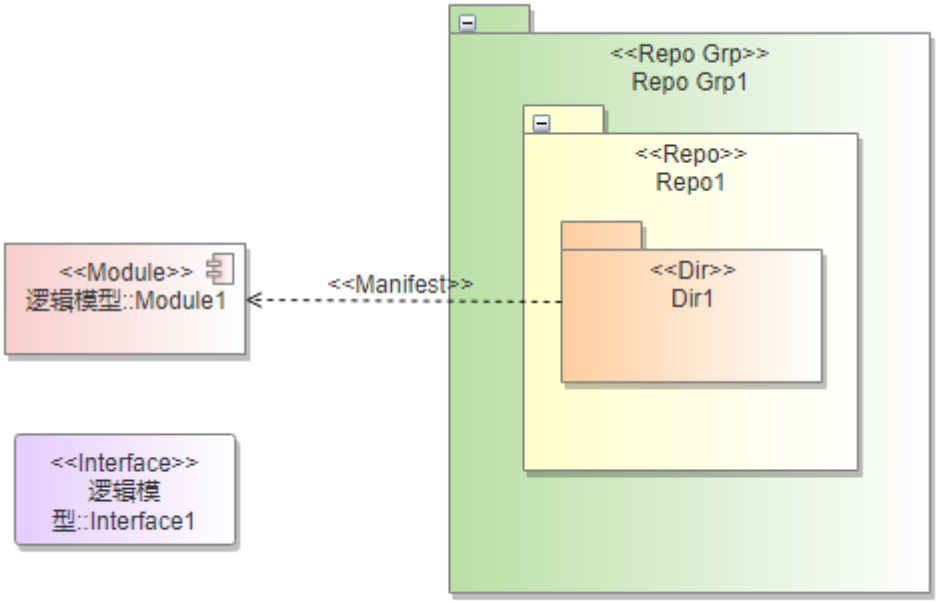
检查规则配置中勾选要检查的元素类型，服务、微服务、组件、模块是默认强制勾选的检查类型，检查这类元素在代码模型图中是否与代码元素存在manifest连线关系，由代码元素指向逻辑元素，不存在对应的代码元素则不符合规则 ，将该类逻辑元素列出到检查结果中。

正确示例



错误示例

引用过来的逻辑元素Interface没有对应任何代码元素。



2.8.1.2.4 构建模型

2.4.1 构建模型的元素要与指定的构建模型层次结构保持一致

详细描述

在构建模型中创建构建元素，构建元素在架构树中与上下级元素的关系层级结构要与构建模型架构方案配置定义的层次结构一致，即该构建元素与上层父级元素、下层子级元素的父子关系（也称上下层级关系）、以及它们之间的连线关系和方向指向，都要与层级规则中定义的保持一致。

检查范围

当前模型工程中的所有符合定义规则的构建元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：构建模型的基础构造型与自定义构造型元素才认定为构建元素）。

1. 在构建模型图上创建出来的构建元素；
2. 引用到构建模型中的构建元素（包含关联空间中的引用的构建元素）；

如何检查

查询基于模型图构出的构建模型架构树，找出与架构方案不匹配（标红）的元素。

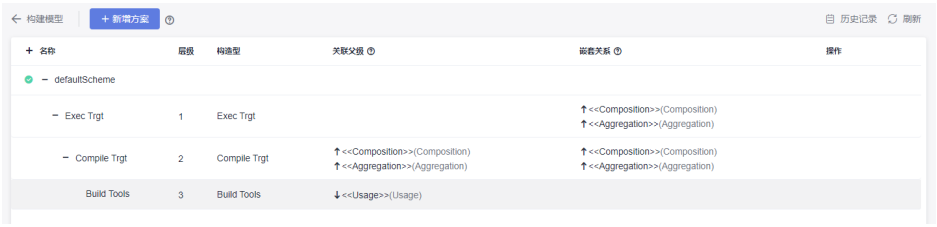
正确示例

架构层级规则示例：

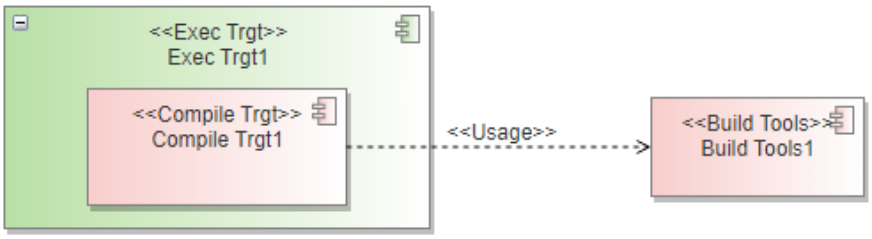
关联父级：配置的是当前层级元素与上一层级的元素之间的连线类型和父子关系指向。

嵌套：是否支持当前类型的元素与同类型元素建立关系。

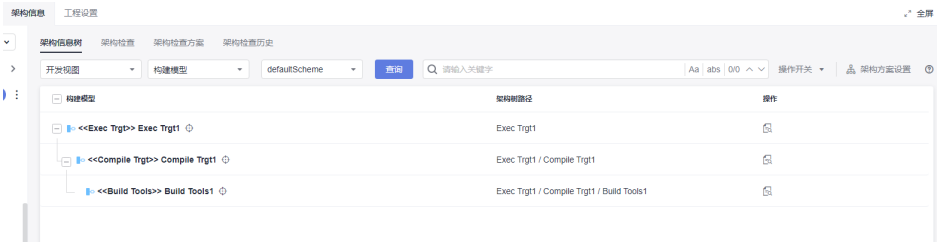
嵌套关系：当前类型的元素与同类型元素建立连线关系类型，指向关系默认为父指向子（即被指向的一方为子）。



图中画法示例1–包含的父子关系和连线关系：

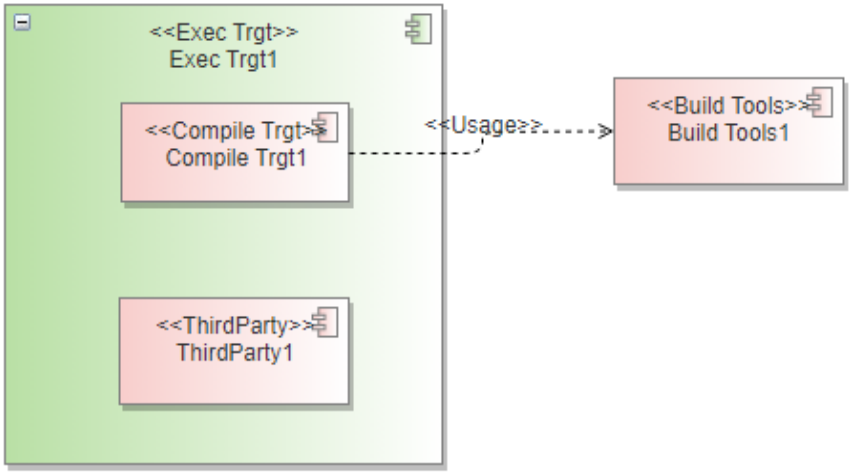


架构信息树展示结果：

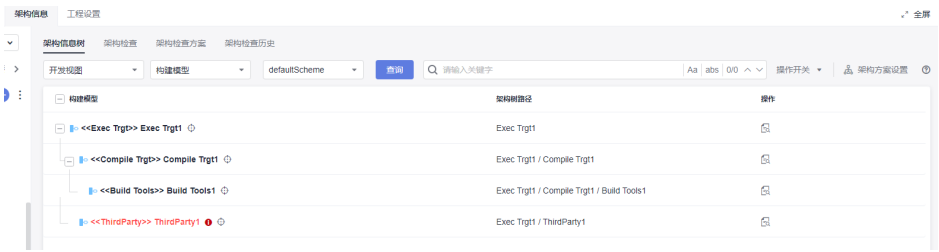


错误示例

错误示例场景1：方案中未配置元素关系，但是绘图中新增了关系。



架构信息树中报红：



架构检查结果：

2.4.2 构建模型不能存在游离的构建模型元素

详细描述

构建模型元素不能独立存在于构建模型架构树之外，必须要与架构树上的构建元素建立关联关系。

检查范围

当前模型工程中的所有符合定义规则的构建元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：构建模型的基础构造型与自定义构造型元素才认定为构建元素）。

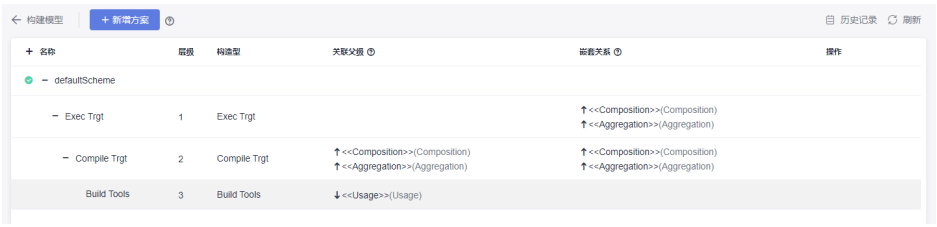
1. 在构建模型图上创建出来的构建元素；
2. 引用到构建模型中的构建元素（包含关联空间中的引用的构建元素）；

如何检查

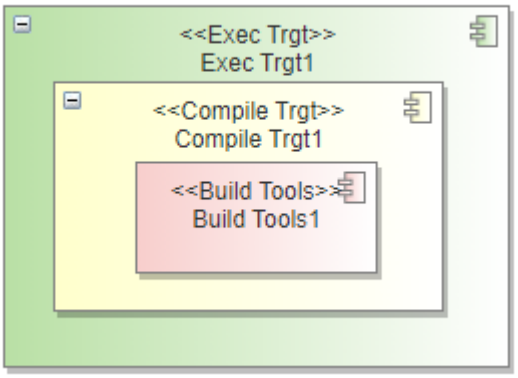
查询基于模型图（只有构建模型图内的构建元素参与构树）并展示不匹配元素构出的构建模型架构树，找出所有构建元素中不在架构树中的构建元素。

正确示例

按构建规则构建的架构信息树：

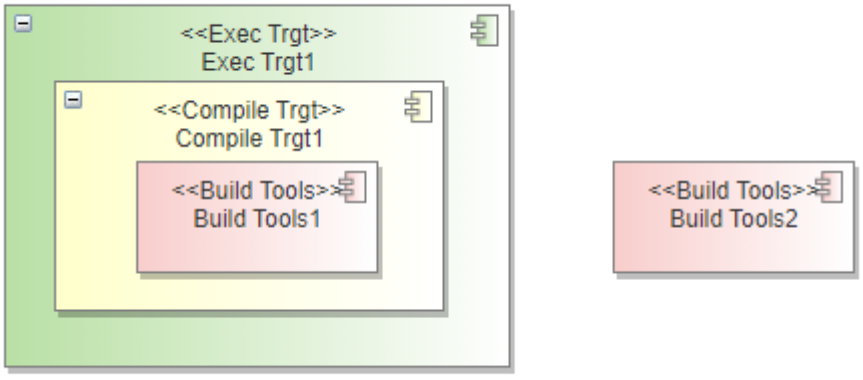


模型图示例：



错误示例

场景一：独立存在的构建元素



检查结果：

2.4.3 构建模型同一个树的同一层上不能有同名同类型的元素

详细描述

在同一棵构建架构信息树上，在同一个父元素节点下面，不能存在类型相同，并且名称也相同的构建元素；

检查范围

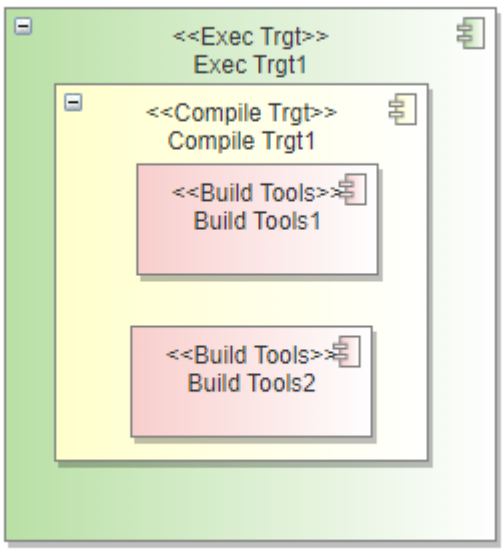
当前模型工程中的所有符合定义规则的构建元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：构建模型的基础构造型与自定义构造型元素才认定为构建元素）。

1. 在构建模型图上创建出来的构建元素；
2. 引用到构建模型中的构建元素（包含关联空间中的引用的构建元素）；

如何检查

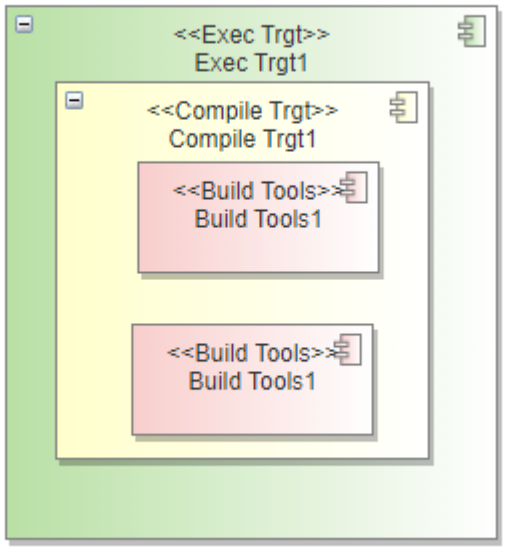
查询基于构建模型图（只有构建模型图内的构建元素参与构树）构出的构建模型架构树，找出同一节点下同名同类型的构建元素。

正确示例

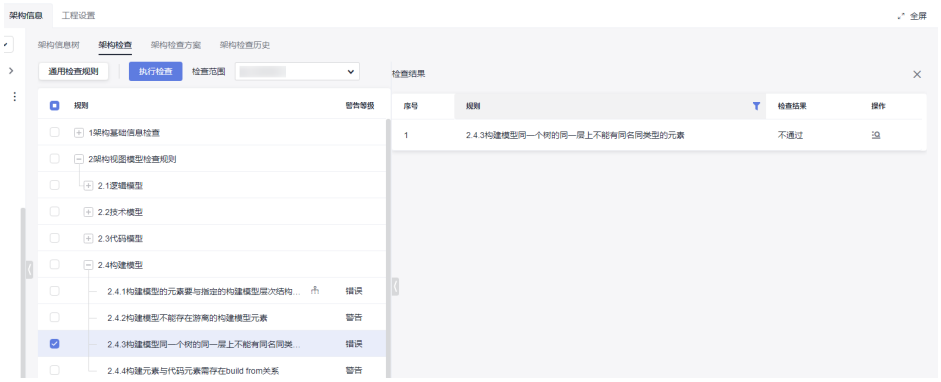


错误示例

错误示例场景1：同父节点下面存在类型相同，名称相同的构建元素。



检查结果：



2.4.4 构建元素与代码元素需存在 build from 关系

详细描述

在构建模型图上，构建元素与从代码模型中引用过来的代码元素如果存在连线关系，必须为build from的构建关系，且方向指向必须从构建元素指向代码元素。

检查范围

当前模型工程中的所有符合定义规则的构建元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：构建模型的基础构造型与自定义构造型元素才认定为构建元素）。

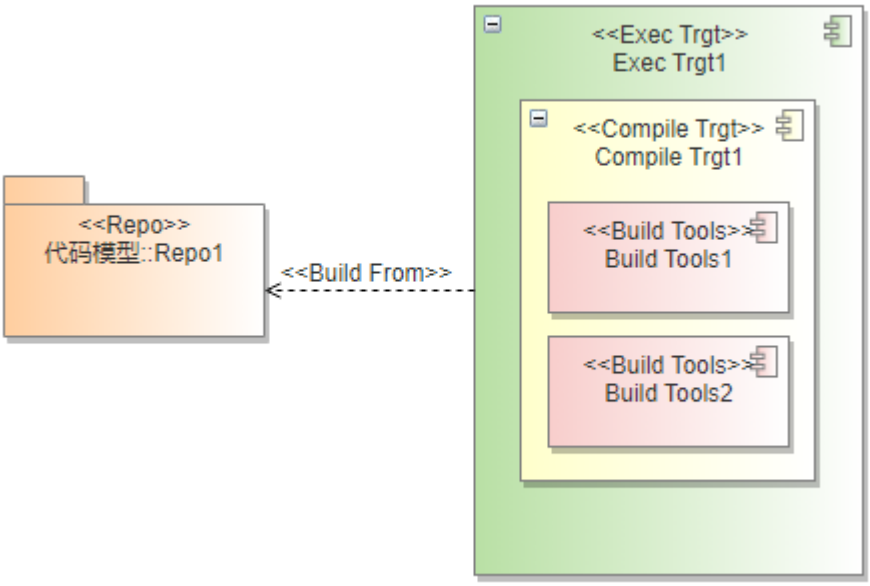
1. 在构建模型图上创建出来的构建元素；
2. 引用到构建模型中的构建元素（包含关联空间中的引用的构建元素）；
3. 引用到构建模型中的代码元素（代码元素的定义参考代码模型检查章节）；

如何检查

检查构建模型中构建元素与引用过来的代码元素之间的连线关系类型是否为build from，如果不存在，则列出这类不符合规则的构建元素。

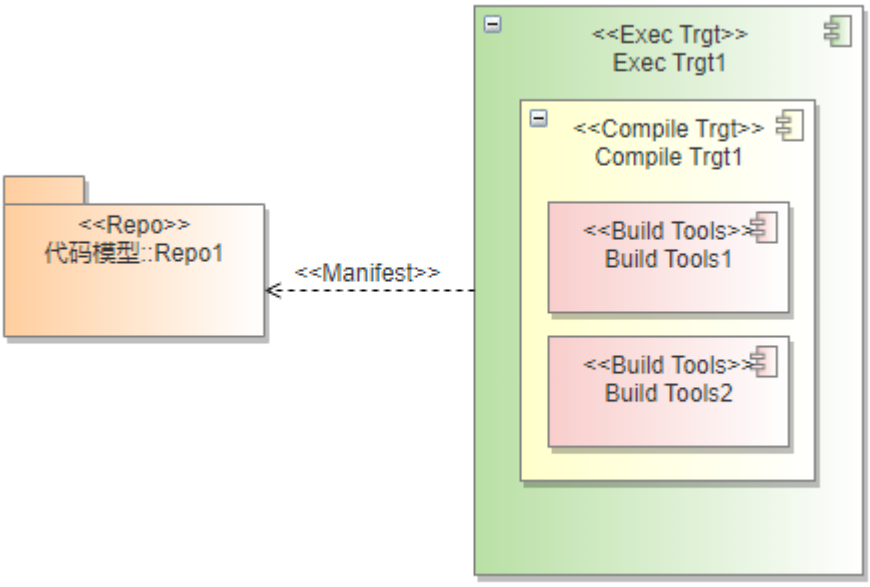
正确示例

构建元素与引用过来的代码元素存在连线关系，关系为Build From类型，且方向由构建元素指向代码元素。



错误示例

错误示例场景1：构建元素与代码元素之间的连线关系类型不对。



检测结果：

2.8.1.2.5 交付模型

2.5.1 交付模型的元素要与指定的交付模型层次结构保持一致

详细描述

在交付模型中创建交付元素，交付元素在架构树中与上下级元素的关系层级结构要与交付模型架构方案配置定义的层次结构一致，即该交付元素与上层父级元素、下层子级元素的父子关系（也称上下层级关系）、以及它们之间的连线关系和方向指向，都要与层级规则中定义的保持一致。

该规则项检查出的是架构信息树中按模型图构树后显示红色叹号的告警元素。

检查范围

当前模型工程中的所有符合定义规则的构建元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：交付模型的基础构造型与自定义构造型元素才认定为交付元素）。

1. 在交付模型图上创建出来的交付元素；
2. 引用到交付模型中的交付元素（包含关联空间中的引用的交付元素）；

如何检查

查询基于模型图构出的交付模型架构树，找出与架构方案不匹配（标红）的元素

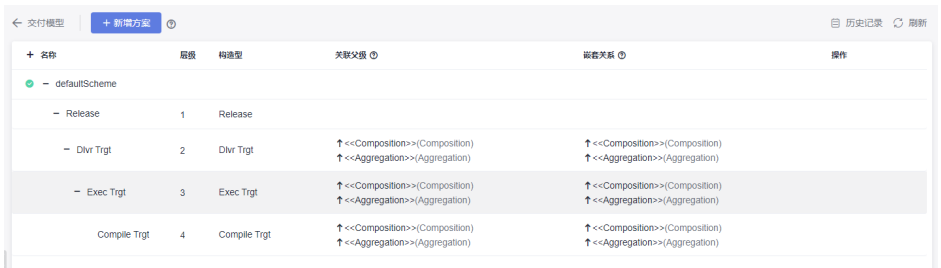
正确示例

架构层级规则示例：

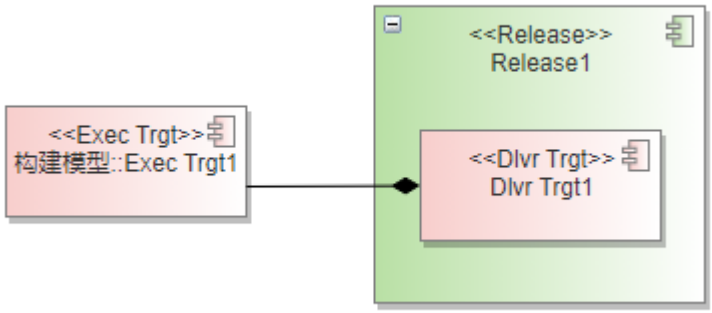
关联父级：配置的是当前层级元素与上一层级的元素之间的连线类型和父子关系指向。

嵌套：是否支持当前类型的元素与同类型元素建立关系。

嵌套关系：当前类型的元素与同类型元素建立连线关系类型，指向关系默认为父指向子（即被指向的一方为子）。



图中画法示例–构建元素连向交付元素，使用组合关系：

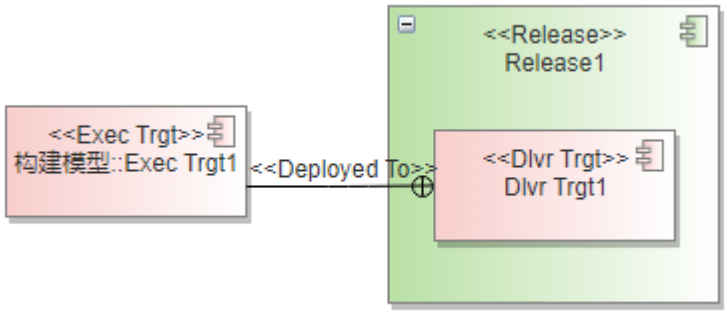


架构信息树展示结果：

当架构树上没有标红元素，就没有2.5.1的检查错误结果。

错误示例

错误示例场景1：连线类型与配置方案中规定的不一致。



架构信息树中报红：

架构检查结果：

2.5.2 交付模型不能存在游离的交付模型元素

详细描述

交付模型元素不能独立存在于交付架构树之外，必须要与架构树上的交付元素建立关联关系。

检查范围

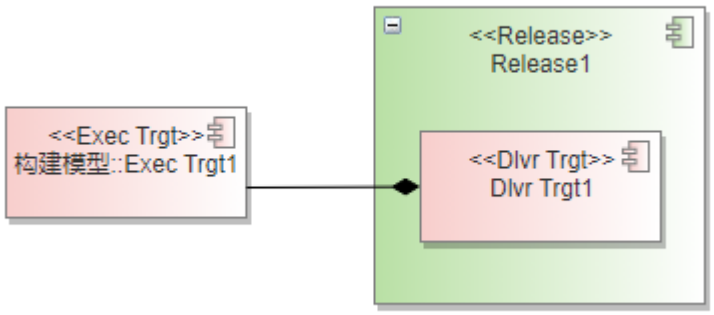
当前模型工程中的所有符合定义规则的构建元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：交付模型的基础构造型与自定义构造型元素才认定为交付元素）。

1. 在交付模型图上创建出来的交付元素；
2. 引用到交付模型中的交付元素（包含关联空间中的引用的交付元素）；

如何检查

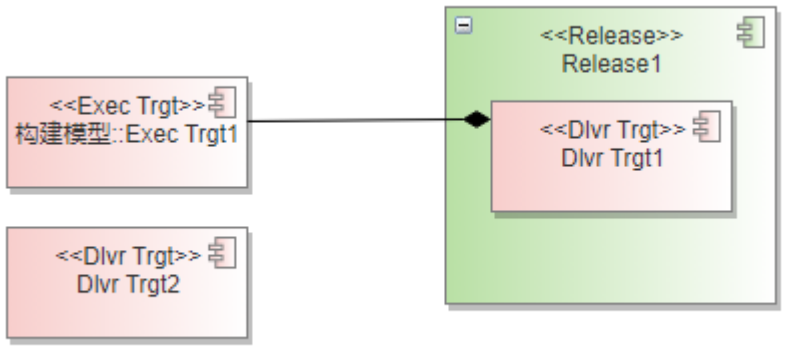
查询基于模型图（只有交付模型图内的交付元素参与构树）并展示不匹配元素构出的交付模型架构树，找出所有交付元素中不在架构树中的交付元素。

正确示例

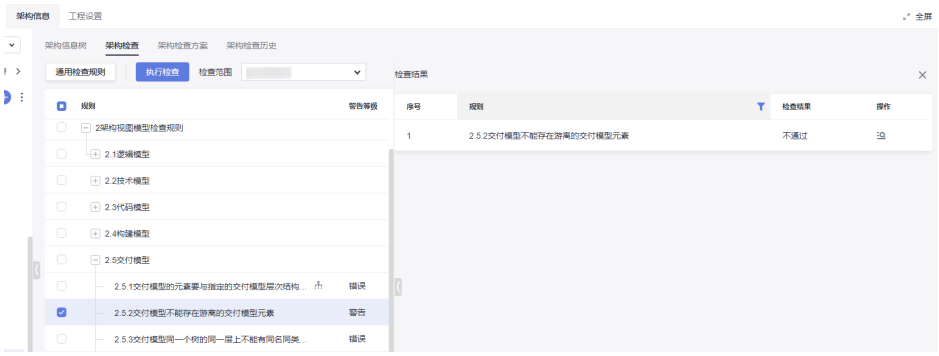


错误示例

错误示例场景1：独立的交付元素Dlvr Trgt存在图上。



架构检查结果：



2.5.3 交付模型同一个树的同一层上不能有同名同类型的元素

详细描述

在同一棵交付架构信息树上，在同一个父元素节点下面，不能存在类型相同，并且名称也相同的交付元素。

检查范围

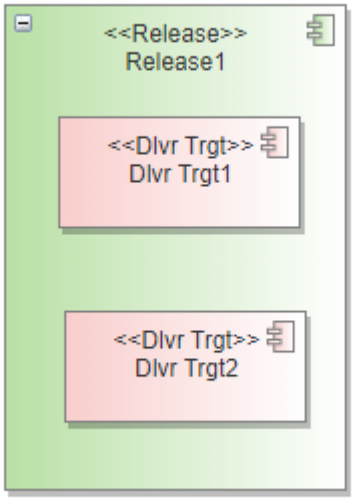
当前模型工程中的所有符合定义规则的交付元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：交付模型的基础构造型与自定义构造型元素才认定为交付元素）。

1. 在交付模型图上创建出来的交付元素；
2. 引用到交付模型中的交付元素（包含关联空间中的引用的交付元素）；

如何检查

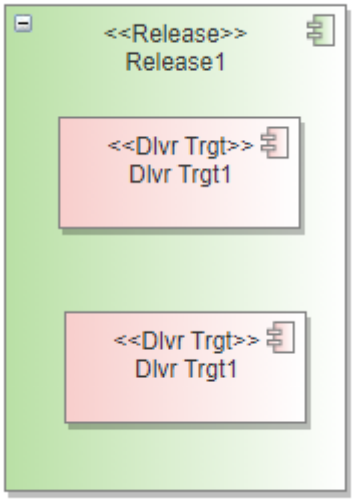
查询基于交付模型图（只有交付模型图内的交付元素参与构树）构出的交付模型架构树，找出同一节点下同名同类型的交付元素。

正确示例

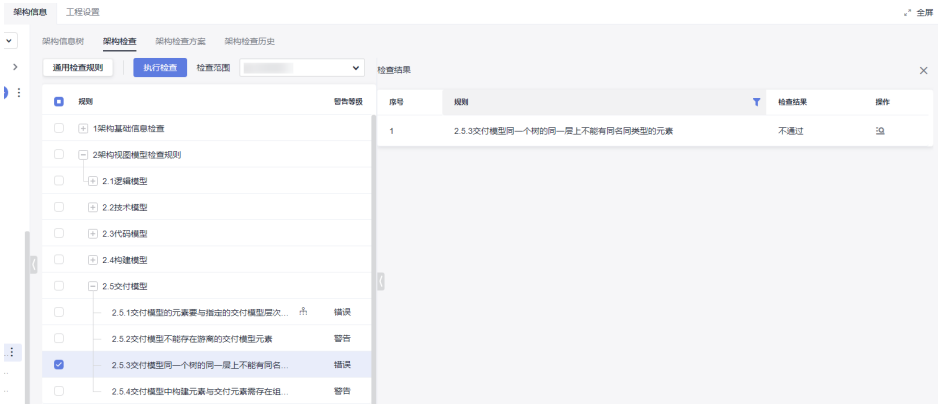


错误示例

错误示例场景1：同父节点下面存在类型相同，名称相同的构建元素。



检测结果：



2.5.4 交付模型中构建元素与交付元素需存在组合或聚合关系

详细描述

在交付模型图上，交付元素与引用过来的构建元素如果存在连线关系，必须为组合或者聚合的关系，且方向指向必须从构建元素指向交付元素。

检查范围

当前模型工程中的所有符合定义规则的构建元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：交付模型的基础构造型与自定义构造型元素才认定为交付元素）。

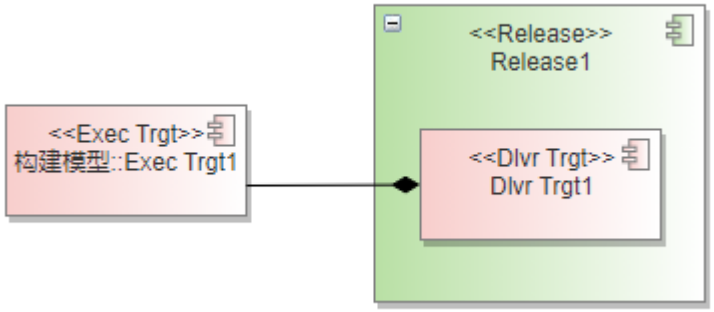
1. 在交付模型图上创建出来的交付元素；
2. 引用到交付模型中的交付元素（包含关联空间中的引用的交付元素）；
3. 引用到交付模型中的构建元素；（构建元素的定义参考代码模型检查章节）；

如何检查

检查交付模型中的引用来的构建元素与交付元素之间的连线关系类型是否为组合Compostion或者聚合Aggregation关系类型，如果为其它类型连线关系，则将该类型元素列出到检查结果中。

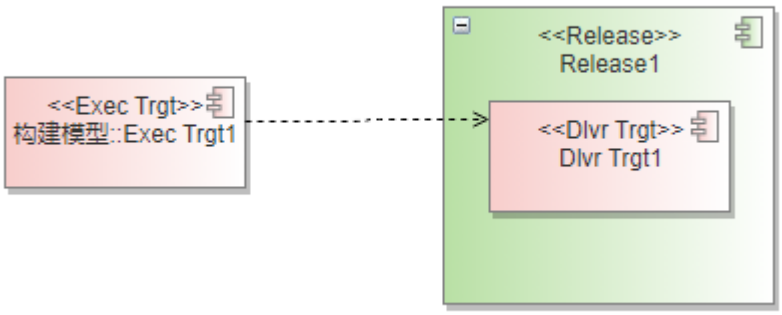
正确示例

交付元素与引用过来的构建元素存在连线关系，关系为组合Composition连线类型，且方向由构建元素指向交付元素。

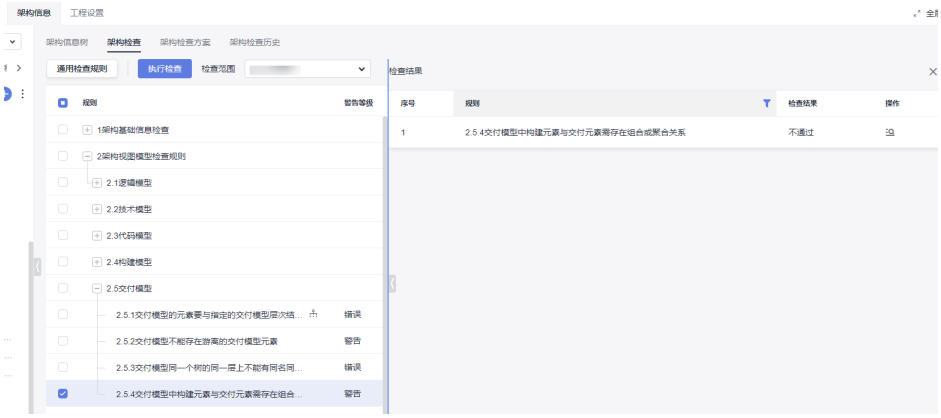


错误示例

错误示例场景1：构建元素与交付元素之间的连线关系类型不对。



检测结果：



2.8.1.2.6 部署模型

2.6.1 检查部署模型的元素是否与架构层次结构是否一致

详细描述

在部署模型中创建部署元素，部署元素在架构树中与上下级元素的关系层级结构要与部署模型架构方案配置定义的层次结构一致，即该部署元素与上层父级元素、下层子级元素的父子关系（也称上下层级关系）、以及它们之间的连线关系和方向指向，都要与层级规则中定义的保持一致。

该规则项检查出的是架构信息树中按模型图构树后显示红色叹号的告警元素。

检查范围

当前模型工程中的所有符合定义规则的部署元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：部署模型的基础构造型与自定义构造型元素才认定为部署元素）。

1. 在部署模型图上创建出来的部署元素；
2. 引用到部署模型中的部署元素（包含关联空间中的引用的部署元素）；

如何检查

查询基于模型图构出的部署模型架构树，找出与架构方案不匹配（标红）的元素。

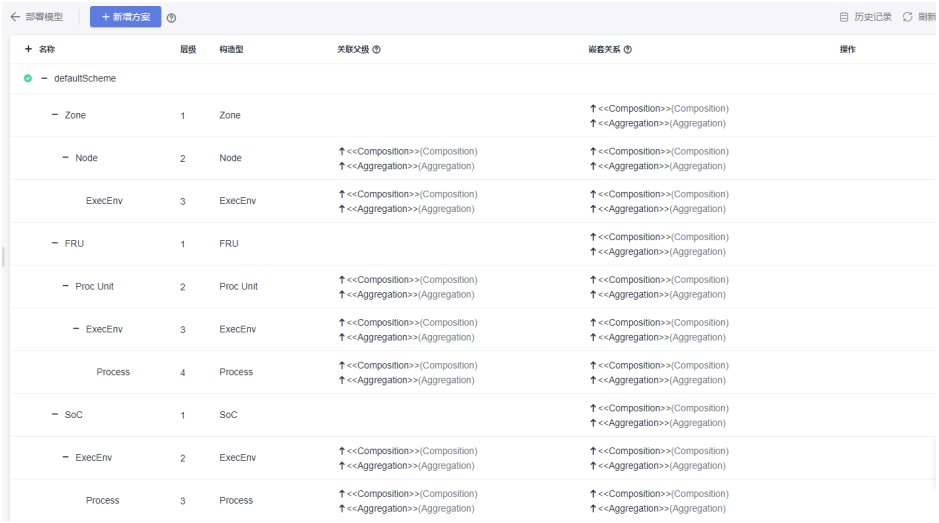
正确示例

架构层级规则示例：

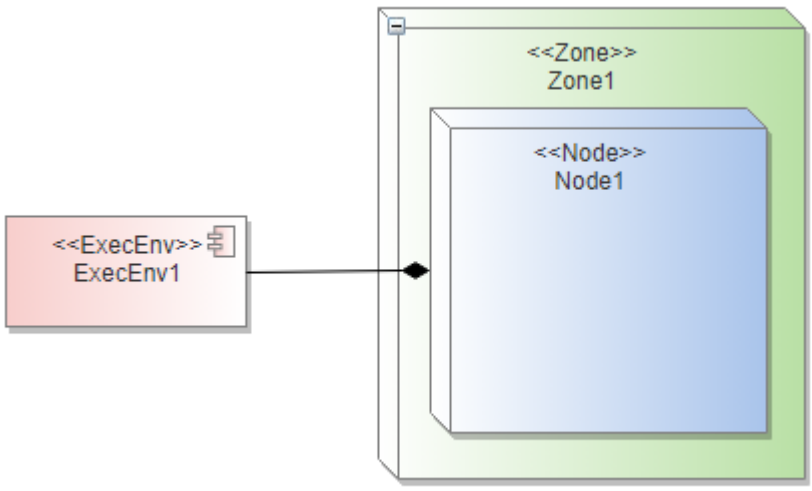
关联父级：配置的是当前层级元素与上一层级的元素之间的连线类型和父子关系指向。

嵌套：是否支持当前类型的元素与同类型元素建立关系

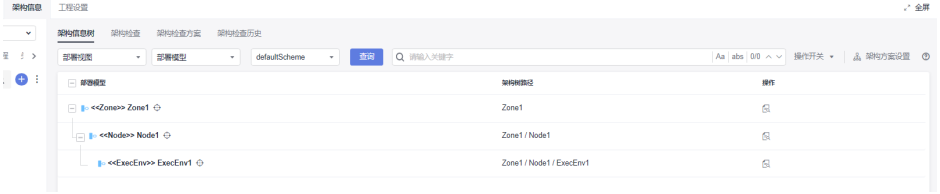
嵌套关系：当前类型的元素与同类型元素建立连线关系类型，指向关系默认为父指向子（即被指向的一方为子）。



包含的父子关系：



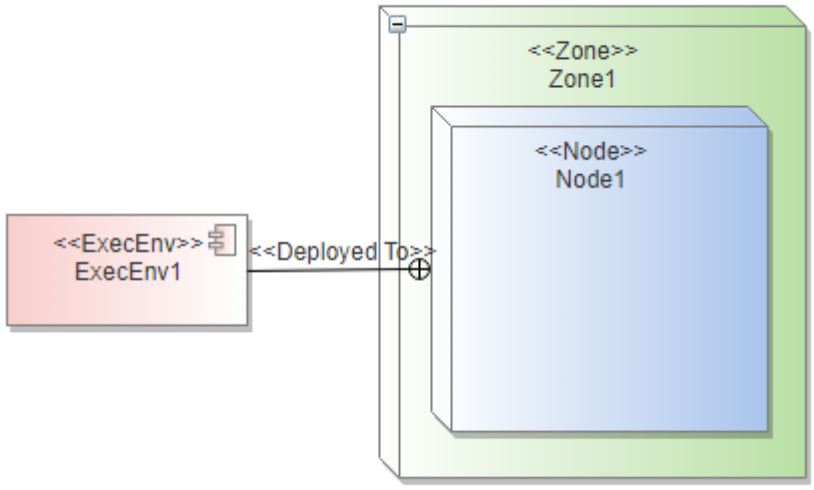
架构信息树展示结果：



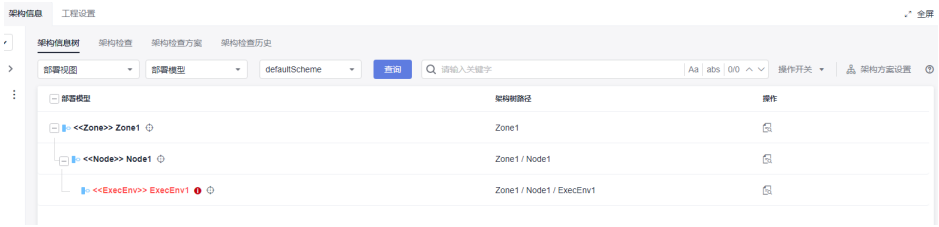
当架构树上没有标红元素，就没有2.6.1的检查错误结果。

错误示例

错误示例场景1：连线类型使用不对。



架构信息树中报红：



架构检查结果：

2.6.2 部署模型不能存在游离的部署模型元素

详细描述

部署模型元素不能独立存在于整个部署架构树之外，必须要与任何一个在架构树上的部署元素建立关系。

检查范围

当前模型工程中的所有符合定义规则的部署元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：部署模型的基础构造型与自定义构造型元素才认定为部署元素）。

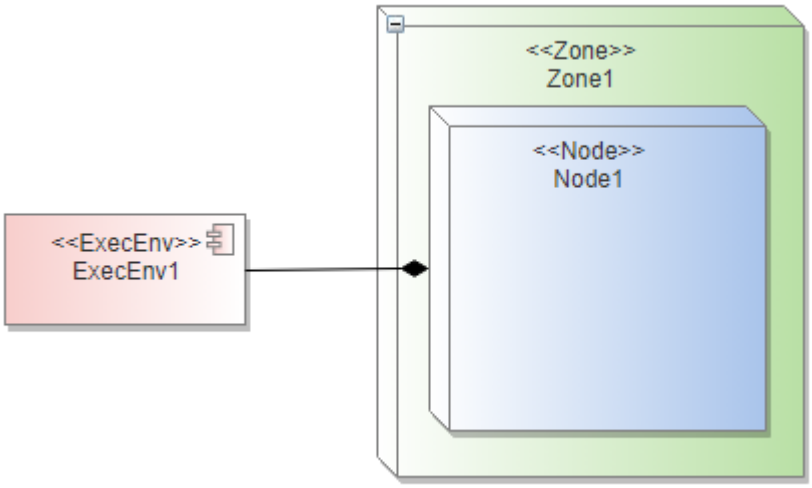
1. 在部署模型图上创建出来的部署元素；
2. 引用到部署模型中的部署元素（包含关联空间中的引用的部署元素）；

如何检查

查询部署模型图内元素类型为架构方案配置构造型的所有元素，查询基于模型图构出的部署模型架构树。

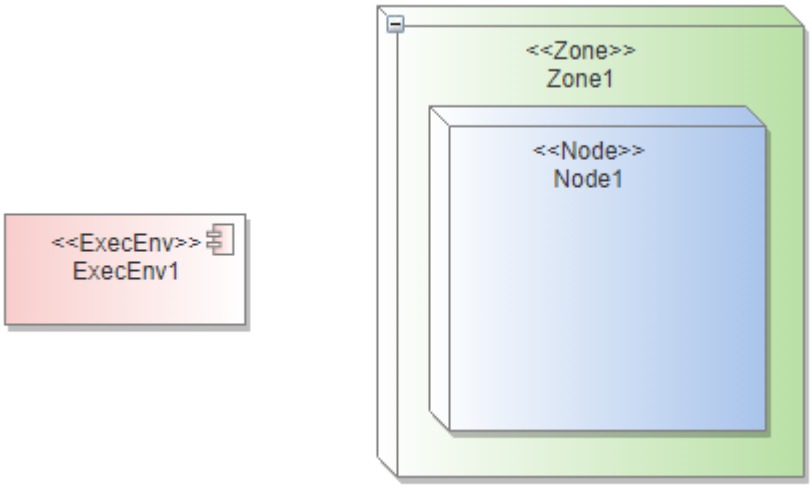
正确示例

每个部署元素都有连线关系和上下级关系（包含关系）。

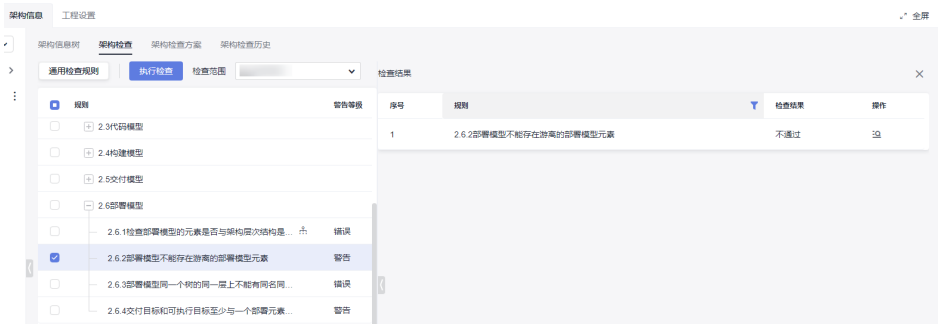


错误示例

错误示例场景1：存在没有建立任务关系的部署元素在图上。



架构检查结果：



2.6.3 部署模型同一个树的同一层上不能有同名同类型的元素

详细描述

在同一棵部署架构信息树上，在同一个父元素节点下面，不能存在类型相同，并且名称也相同的元素。

检查范围

当前模型工程中的所有符合定义规则的部署元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：部署模型的基础构造型与自定义构造型元素才认定为部署元素）。

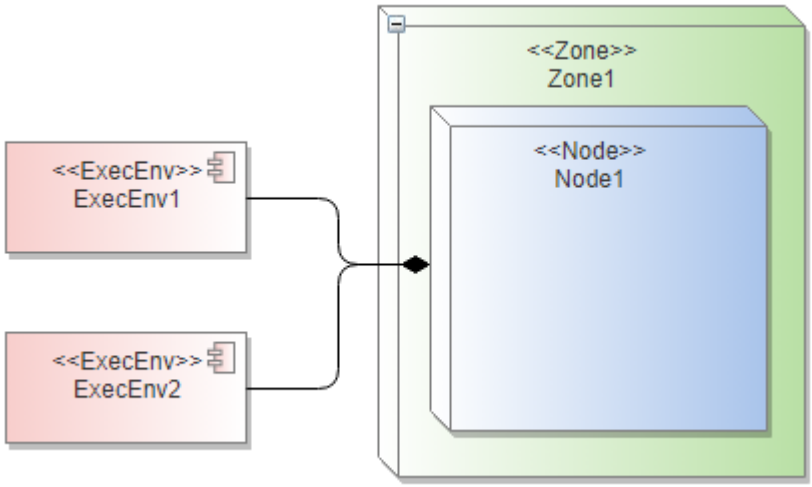
1. 在部署模型图上创建出来的部署元素；
2. 引用到部署模型中的部署元素（包含关联空间中的引用的部署元素）；

如何检查

查询基于模型图构出的部署模型架构树，找出架构树上同一层同名同类型的元素。

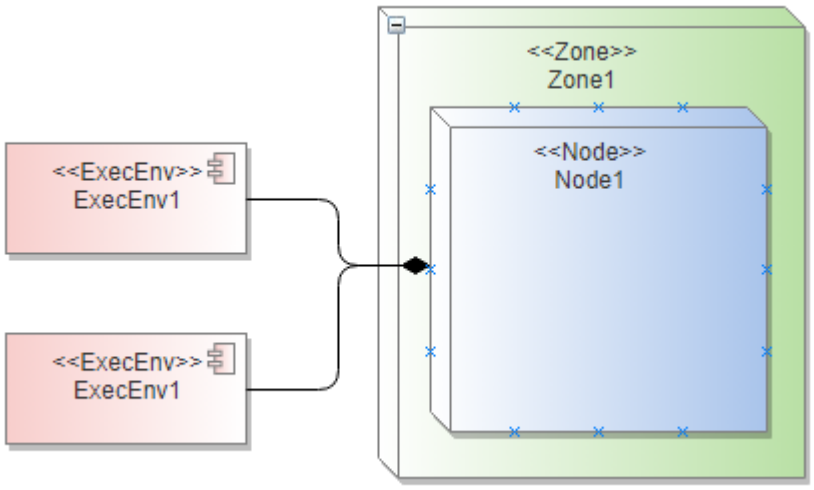
正确示例

同Node节点下面的不能有同类型同名元素。



错误示例

错误示例场景1：存在同名同类型的部署元素。



架构规则检查结果：

2.6.4 交付目标和可执行目标至少与一个部署元素存在部署关系

详细描述

交付目标和可执行目标（Dlvr Trgt、Exec Trgt）至少与一个部署元素有部署关系或者包含关系，部署关系指部署连线关系，即Deployed To连线关系；包含关系，即部署元素在图上包含交付目标和可执行目标（Dlvr Trgt、Exec Trgt）元素。

检查范围

当前模型工程中的所有符合定义规则的部署元素（定义规则：工程设置 > 构造型下，绑定到4+1视图：部署模型的基础构造型与自定义构造型元素才认定为部署元素）。

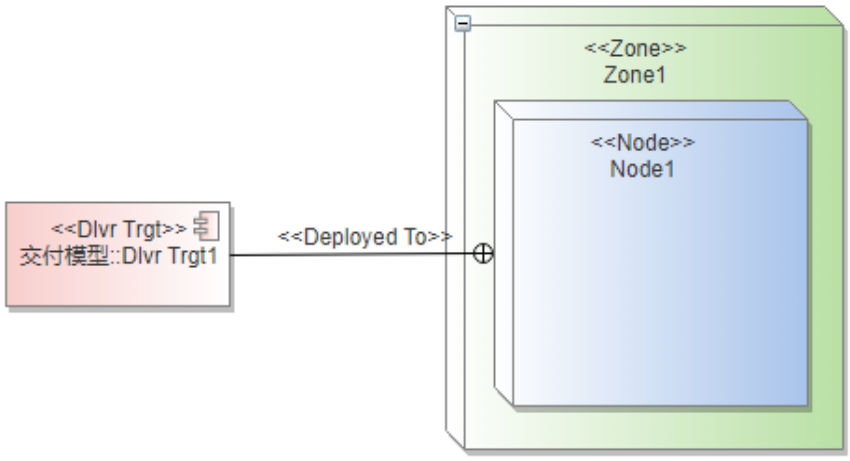
1. 在部署模型图上创建出来的部署元素；
2. 引用到部署模型中的部署元素（包含关联空间中的引用的部署元素）；
3. 交付目标和执行目标，即Dlvr Trgt和Exec Trgt元素；

如何检查

查询部署模型图中生成的部署元素，查询部署模型图中的交付目标和可执行目标（Dlvr Trgt、Exec Trgt），找出交付目标和可执行目标中既无Deployed To部署元素连线关系也没有父子关系（部署元素为父，交付目标和可执行目标为子）的元素。

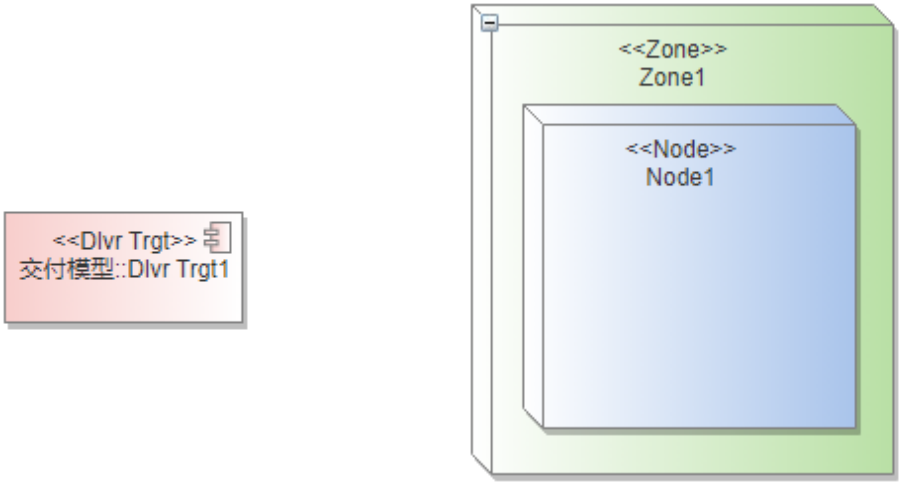
正确示例

交付目标与部署元素存在部署连线关系，且由交付元素指向部署元素。

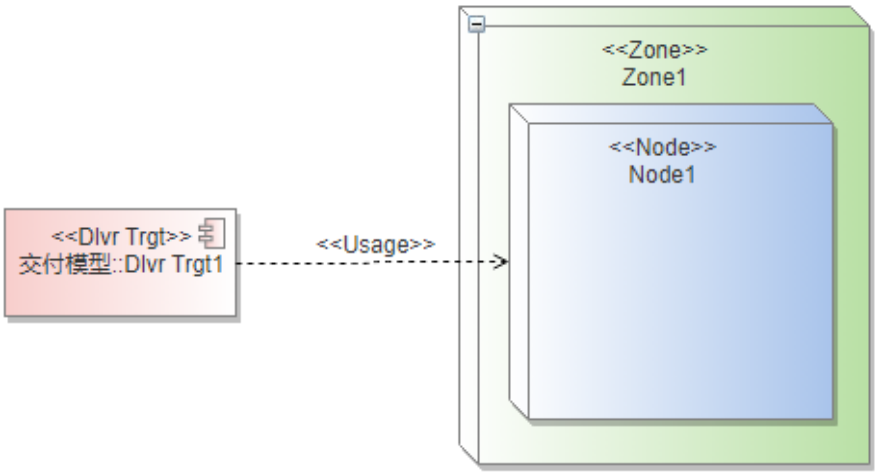


错误示例

错误示例场景1：交付元素与部署元素没有连线关系也没有包含关系。



错误示例场景2：交付元素与部署元素连线关系类型不正确。



2.8.1.2.7 上下文模型

2.7.1 上下文模型中只能有一个 System

详细描述

在上下文模型中只能存在一个类型为System的元素；其它的三方交互的对象用ExternalSystem或者Actor元素表示。

检查范围

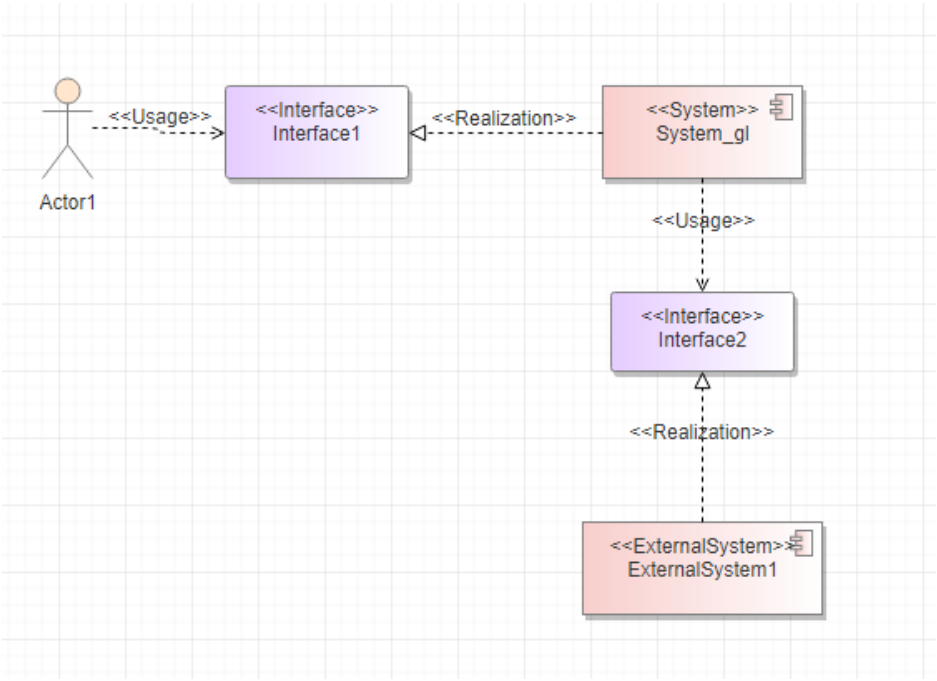
当前模型工程中的所有符合定义规则的System元素，工程设置 > 构造型下，绑定到4+1视图 : 上下文模型的System元素。

1. 在上下文模型图上创建出来的System元素；
2. 引用到上下文模型中的System元素（包含关联空间中的引用的system元素）；

如何检查

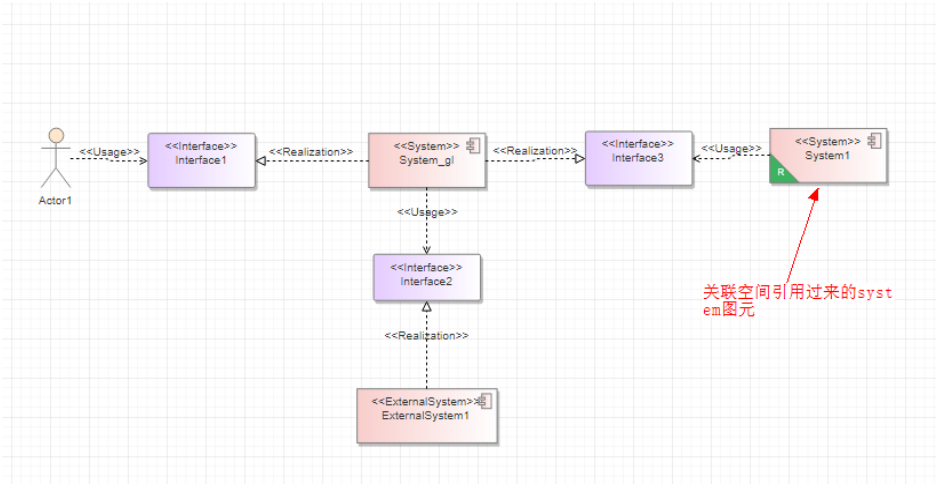
查询上下文模型图中的所有元素，从中找出类型为System的元素，如果存在多个System元素，则全部列出到检查结果中，不符合规则。

正确示例



错误示例

错误示例场景1：关联空间引用system元素到上下文。



架构规则检查结果：

2.7.2ExternalSystem 和 Actor 只能存在下面两种关系中的一种：ExternalSystem 和 Actor 使用或依赖 System 提供的接口；ExternalSystem 和 Actor 提供了接口给 System 使用或依赖

详细描述

ExternalSystem和Actor元素与System之间只能通过接口交互，不能直接使用连线关系表达交互，只能由ExternalSystem和Actor实现（Realization连线）接口，并由System使用（usage连线）该接口；或者由System实现（Realization连线）接口，由ExternalSystem和Actor使用（usage连线）该接口；其中使用关系可以由依赖Dependency连线关系代替。

检查范围

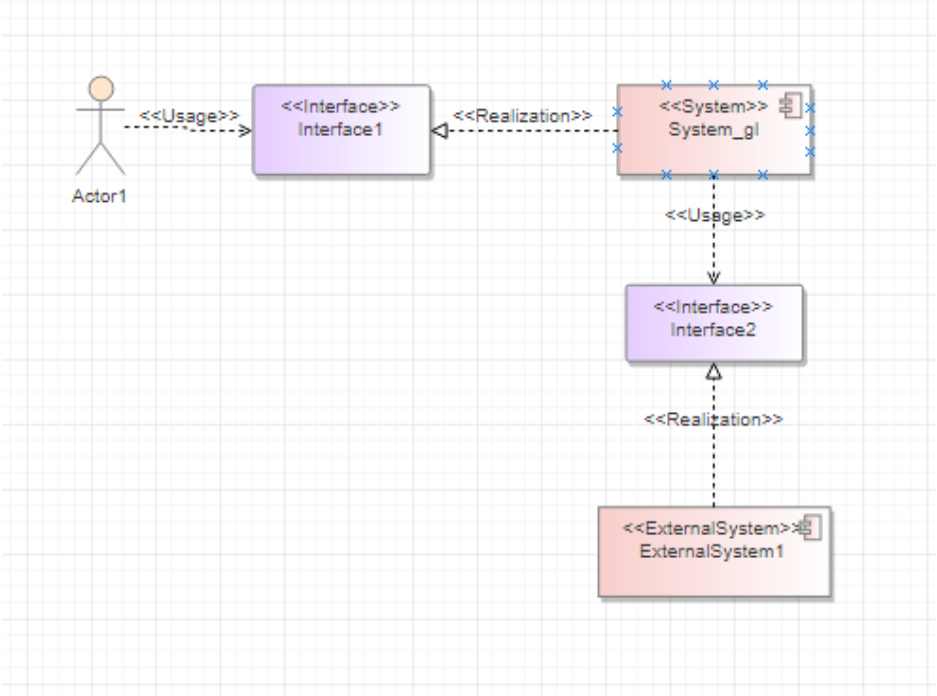
当前模型工程中的所有符合定义规则的System、ExternalSystem和Actor元素，工程设置 > 构造型下，绑定到4+1视图：上下文模型的System、ExternalSystem和Actor元素。

1. 在上下文模型图上创建出来的System、ExternalSystem和Actor元素；
2. 引用到上下文模型中的System、ExternalSystem和Actor元素（包含关联空间中的引用的System、ExternalSystem和Actor元素）；

如何检查

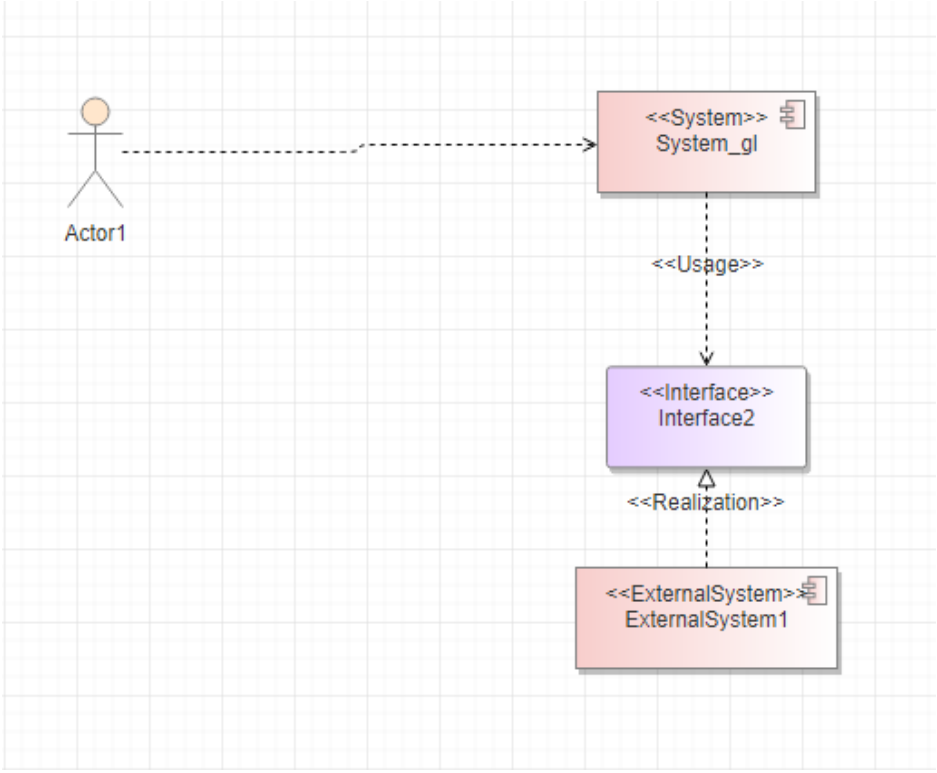
检查上下文模型中的ExternalSystem和Actor元素与System之间是否通过接口相关联，与接口之间存在使用，实现或者依赖关系连线，如果存在其它类型的连线也不符合规则，会列出不符合规则 的ExternalSystem和Actor元素在检查结果列表中。

正确示例



错误示例

错误示例场景1：没有通过接口交互，Actor与system直接用连线表示交互关系。



2.8.1.2.8 运行模型

2.8.1 运行模型、运行模型-顺序图、运行模型-活动图中不能产生新的逻辑元素

详细描述

在运行模型中不能创建新的逻辑元素，只能从逻辑模型中引用或者实例化到运行模型中来进行设计。

检查范围

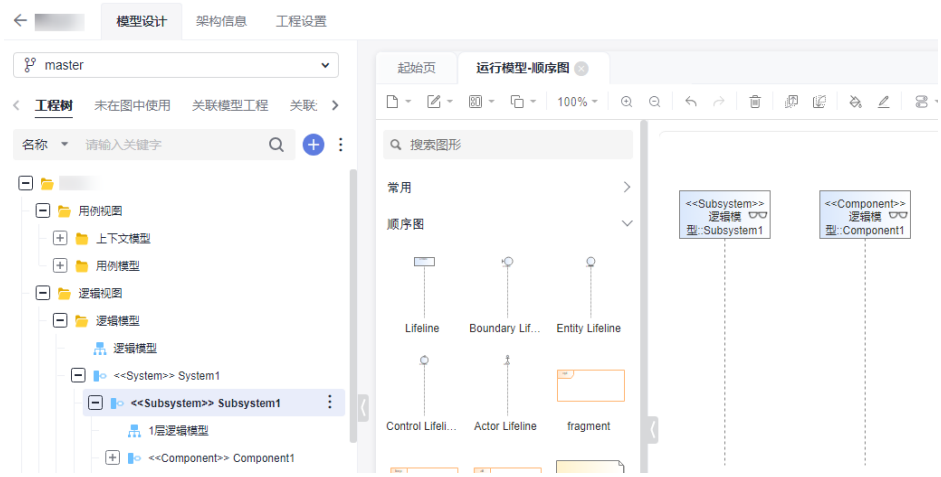
当前模型工程中的所有模型图类型为运行模型图上的逻辑元素，逻辑元素的定义参考逻辑模型检查章节。

如何检查

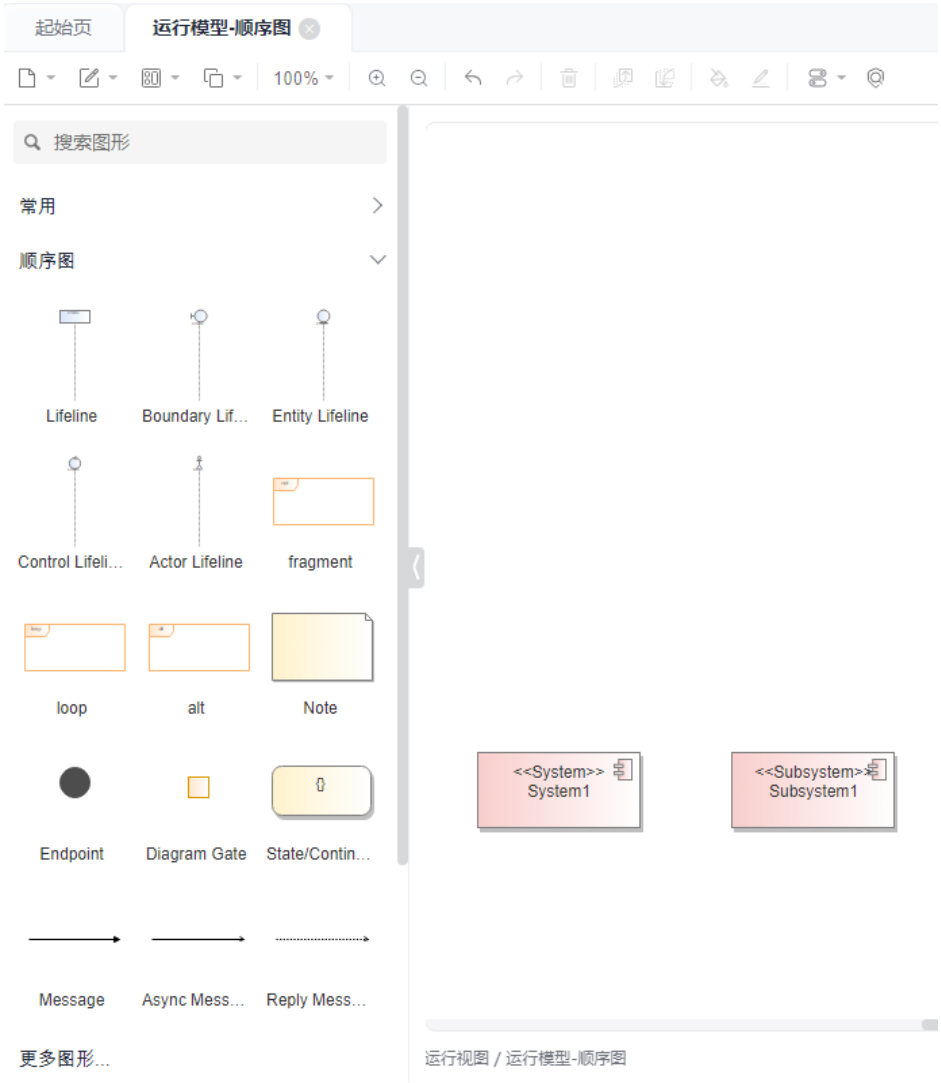
查询所有运行模型图中的元素，找出在运行模型图中创建生成出来的逻辑元素。

正确示例

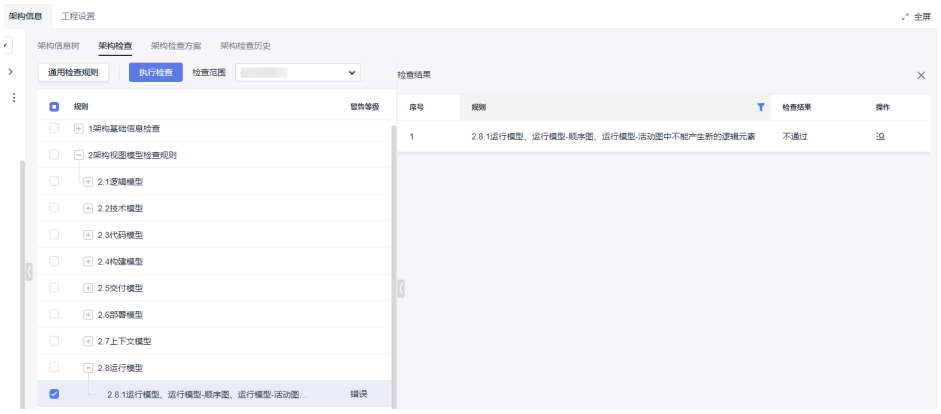
引用自逻辑模型的中定义的逻辑元素。



错误示例



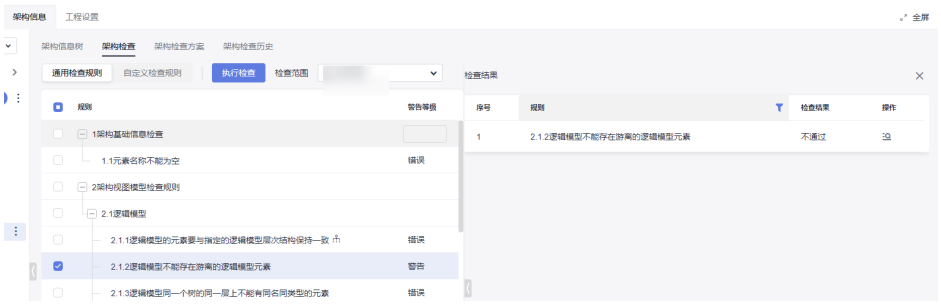
检测结果：



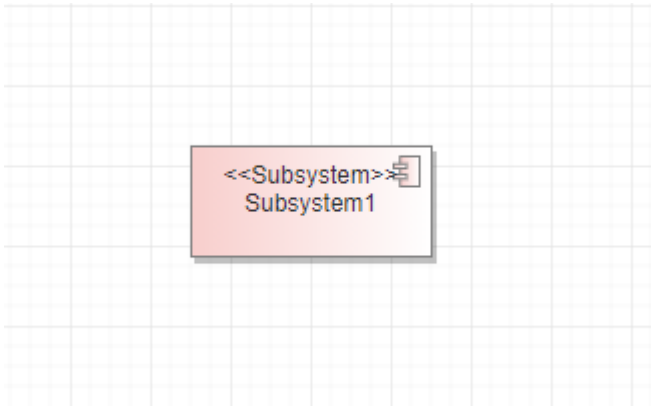
2.8.2 4+1 视图规范一致性检查错误修复指导

- XX模型不能存在游离的逻辑模型元素

以逻辑模型为例:



游离原因：元素没有在逻辑模型架构信息树中出现。

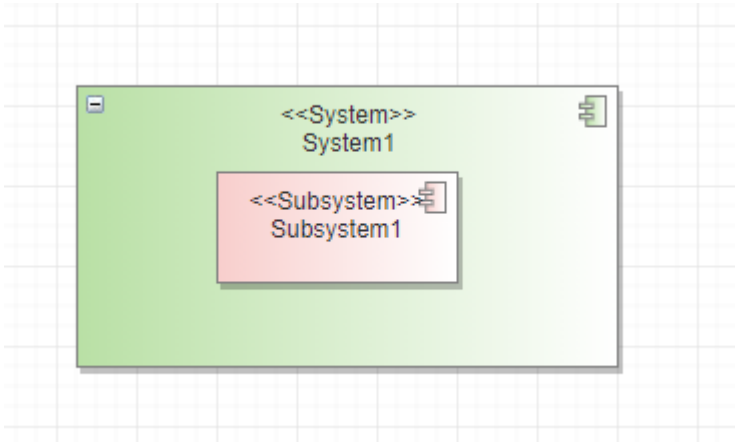


1. 查看逻辑模型架构方案设置。

2. 找到游离元素构造型相关的架构配置信息。

Subsystem需要与System有Composition/Aggregation关系或父子关系。

3. 在模型图中构建架构关系。

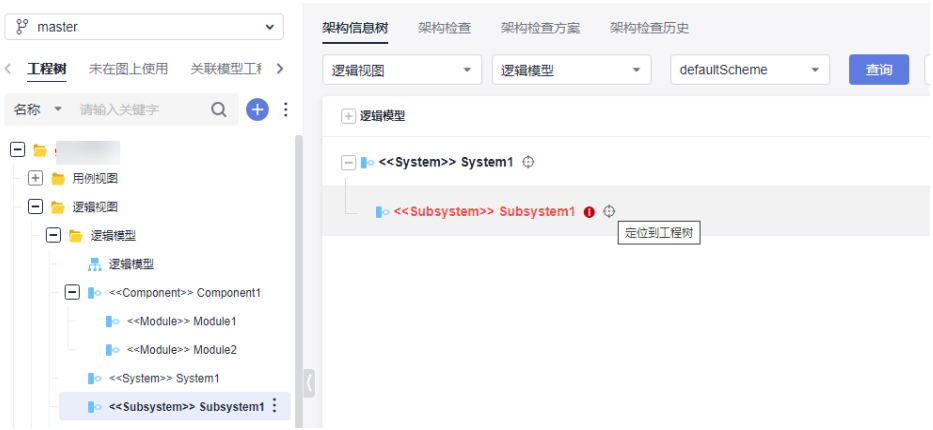


- XX模型的元素要与指定的XX模型层次结构保持一致

以逻辑模型为例:

1. 查询逻辑模型架构信息树，右侧操作开关把展示不匹配架构方案的元素打开。

2. 架构信息树构出后 根据错误元素名称查询定位到其所在架构树节点。



3. 查询错误元素与其他元素关系。

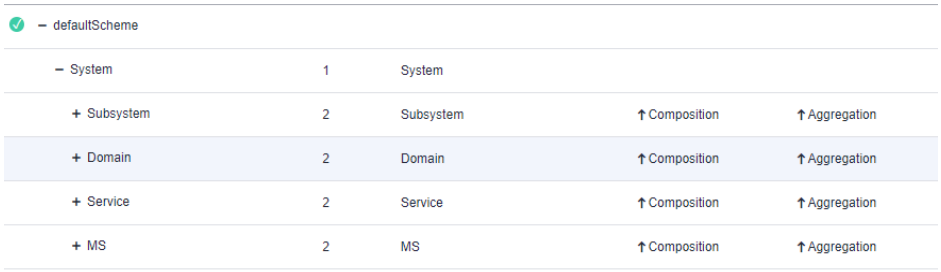
对比架构方案设置。

Subsystem1报错是因为与System1（架构信息树上的父节点）存在错误架构关系，对比发现实际模型图中使用的是Dependency连线 而架构配置方案要求Composition/Aggregation。

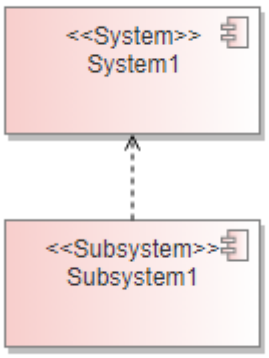
4. 在模型图中修改连线类型为Composition/Aggregation。

常见错误场景：

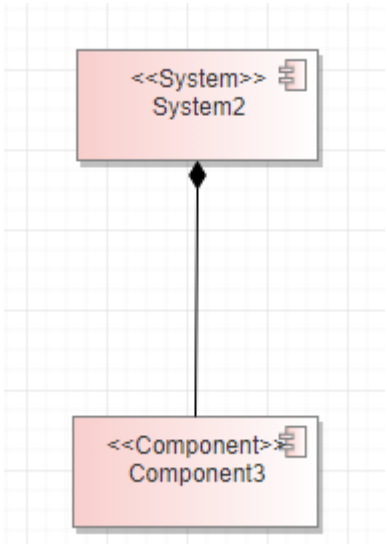
- 连线类型不符合架构配置方案。
  规则

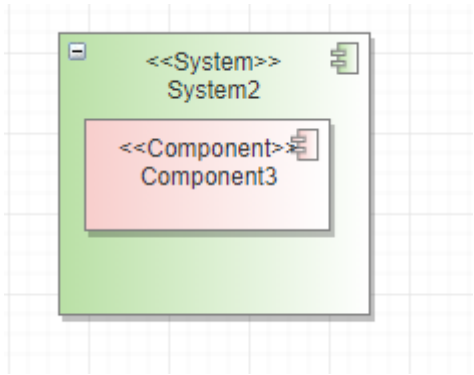


实际



- 子元素构造型不符合架构配置方案。





System下层子元素按架构配置方案只能是Subsystem、Domain、Service、MS 图中是Component。