河南工業大學

2021 届毕业生

毕 业 设 计 说 明 书

题　　目：计算机联锁培训系统设计

院系名称：电气工程学院　　专业班级：轨道 1702

学生姓名：张睿　　学　　号：201712010311

指导教师：尚庆松　　教师职称：讲师

2021 年 5 月 2 日

摘　要

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum pretium libero non odio tincidunt semper. Vivamus sollicitudin egestas mattis. Sed vitae risus vel ex tincidunt molestie nec vel leo. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Maecenas quis massa tincidunt, faucibus magna non, fringilla sapien. In ullamcorper justo a scelerisque egestas. Ut maximus, elit a rutrum viverra, lectus sapien varius est, vel tempor neque mi et augue. Fusce ornare venenatis nunc nec feugiat. Proin a enim mauris. Mauris dignissim vulputate erat, vitae cursus risus elementum at. Cras luctus pharetra congue. Aliquam id est dictum, finibus ligula sed, tempus arcu.

关键字： 分布式系统；Web 应用；GraphQL；Rust 程序设计语言；计算机联锁

Title Design of Computer-based interlocking edutraining system

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum pretium libero non odio tincidunt semper. Vivamus sollicitudin egestas mattis. Sed vitae risus vel ex tincidunt molestie nec vel leo. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Maecenas quis massa tincidunt, faucibus magna non, fringilla sapien. In ullamcorper justo a scelerisque egestas. Ut maximus, elit a rutrum viverra, lectus sapien varius est, vel tempor neque mi et augue. Fusce ornare venenatis nunc nec feugiat. Proin a enim mauris. Mauris dignissim vulputate erat, vitae cursus risus elementum at. Cras luctus pharetra congue. Aliquam id est dictum, finibus ligula sed, tempus arcu.

Keywords：Distributed systems；Web application；GraphQL；Rust Programming Language；Computer-based interlocking

目 次

1 序论

在计算机联锁系统快速发展的今天，传统的培训模式已经逐渐不能适应我国铁路相关方面的需求，如果使用完全真实计算机联锁系统进行人才培训，不仅价格高昂，而且规模太大，不符合实际要求，另一方面，我国的铁路站数量庞大，不可能每一个站都配有仿真操作培训的联锁设备。因此，针对计算机联锁的仿真系统设计迫在眉睫，该系统不仅有助于解决具体站的计算机联锁的教学问题，而且对人员操作培训、联锁试验培训方面也有很大帮助。因此我们需要开发一种针对标准站的计算机联锁培训系统，为这方面的人才培养提供一个方便快捷的软件平台。

2 需求分析

uroj 旨在设计一款高性能、可扩展、高并发、通用性计算机联锁培训系统。通用性在于我们需要一款不拘束于某个具体车站的计算机联锁培训系统。本系统需要拥有对任意一车站进行联锁培训的能力，可扩展性在于需要让用户可以自己定义车站。用户可以在自定义的车站上进行联锁培训，高性能在于作为一款 web 应用，尽量缩短相应时间、提升硬件利用效率，减少冗余。高并发在于作为一款 web 应用，通过设计保证系统能够同时并行处理尽量多的请求。

3 系统概览

3.1 系统架构

为提高并发，提升应用性能，本案采用分布式系统设计，如图 3.1 所示，

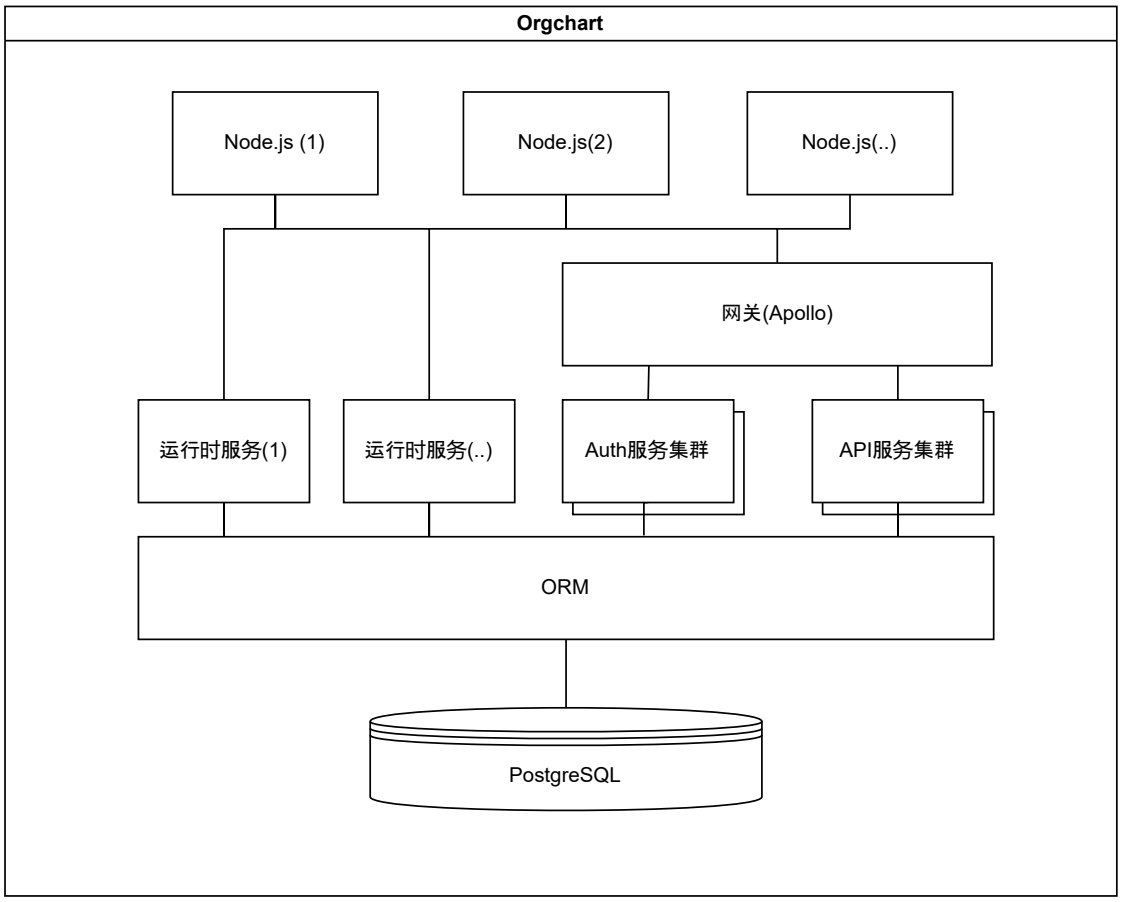

图 3.1: 系统架构组织图

本案符合非典型的 web 应用层次结构，分为表现层，接入层，业务逻辑层，数据访问层，其中数据访问层采用名为 diesel 的 Rust crate 作为 ORM。业务逻辑层分为 Auth、API、Runtime 等数个服务，每个服务都是独立的应用，可以横向扩展组成集群。接入层使用 Apollo 作为 GraphQL 的网关，向外暴露所有的服务接口，还可以进行流量控制，但为了支持运行时 (Runtime) 服务的“热插拔”，运行时服务并不会使用网关。表现层使用 Node.js 作为 Web 的运行时，使用 React 作为 GUI 框架。表现层和接入层、业务逻辑层使用 GraphQL 实现 Schema，使用 HTTP 和 WebSocket 协议通信，并使用 actix-web 作为 web 服务端。

3.2 生命周期与任务调度

一个典型的实例生命周期由以下几部分组成

1. 创建车站
2. 创建实例
3. 初始化实例
4. 访问实例
5. 结束实例

其中，创建车站就需要用到后文提到的车站描述文件，车站被创建后将存入数据库中。创建好车站后，就可以创建这个车站的实例。本案支持预约或称定时开始的实例，在开始之前若用户尝试在 executor 初始化一个实例，就会报错。在 GUI 上，在开始时间之前，不渲染开始按钮，和后端的时间约束形成两层约束。实例创建后同样也会被记录在数据库中，当时间到后用户就可以在创建实例时指定的 executor 上初始化实例 – executor 从数据库中读入 instance，并运行。实例初始化后用户就可以在该实例中进行进路车辆的各种操作。最后实例会被结束。

4 业务层

4.1 API 服务

API 服务主要负责耦合 data 层和 view 层，上承用户的请求，下接数据库，从数据库读写数据并呈递给前端。在本案中，API 服务提供车站、实例、考试、用户和班级四个类型的服务。在 api 服务中，Station 数据是最完备最上游的车站静态数据，其直接来源于用户的输入。Station 数据直接来源于车站描述文件

4.1.1 车站描述文件

车站描述文件用于描述车站，即使用上述属性来定义一个车站，车站描述文件作为用户向本系统的输入，是用户唯一定义车站的方式，因此，为兼顾可读性和文件体积需求，本案采用 yaml 作为车站的描述语言。yaml 是一个可读性高，用来表达资料序列化的格式。Clark Evans 在 2001 年首次发表了这种语言 [1]，另外 Ingy döt Net 与 Oren Ben-Kiki 也是这语言的共同设计者 [2]。目前已经有数种编程语言或脚本语言支持（或者说解析）这种语言。车站描述文件将在 Executor 中被解析成实例，与此相关的细节参见第七章。前文曾道“基本上，一个车站是由数个 Signal 和数个 Node 构成的”，但车站描述文件中除了信号机和节点的定义之外，还有另外两个字段其一是车站的标题，一般为站名，另一为独立按钮，譬如咽喉区设置的列车终端按钮 LZA，这种按钮是不依附于信号机的，因此需要单独定义，包括按钮的 id，位置和其映射的节点。这里给出一个非典型的车站描述文件作为例子：在注释中解释上述内容

```yaml
---
title: 测试站
nodes:
  - node_id: 129
    node_kind: SIDING  #类型SIDING为站线节点
    turnout_id: []  #道岔id组为空，说明该node不处于道岔区段
    track_id: 3G  #其所属轨道区段的id为3G
    left_adj: [97]  #左邻node 97
    right_adj: [44]  #右邻node 44
    conflicted_nodes: []  #没有相抵触的node
    line: [[30.0, 2.0], [200.0, 2.0]]  #线段自(30,2)至(200,2)
    joint: [NORMAL, NORMAL]  #两端绝缘节均为普通绝缘节
signals:
  - id: S
```

```
    side: UPPER  #朝上
    sig_type: HOME_SIGNAL  #进站信号机
    sig_mnt: POST_MOUNTING  #高柱
    protect_node_id: 3  #防护节点3
    toward_node_id: 1  #朝向节点1
    btns: [PASS, GUIDE, TRAIN]  #拥有通过、引导、列车按钮
  - id: S3
    side: UNDER  #朝下
    sgn_kind: STARTING_SIGNAL  #出站信号机
    sgn_mnt: GROUND_MOUNTING  #矮柱
    protect_node_id: 97  #防护节点97
    toward_node_id: 129  #朝向节点129
    btns: [TRAIN, SHUNT]  #拥有列车和调车按钮
    dif_sgn: X3  #差置信号机为X3
independent_btns: [] #没有独立按钮
```

显然上述文件中定义了一个站线节点、一架进站信号机、一架出站兼调车信号机。

4.1.2 车站

车站属性从性质上可以分为图形属性和逻辑属性，图形属性用于表现层初始化 Instance 时正确地渲染出车站底平面图，逻辑属性用于 Runtime 初始化实例时正确地描述车站的拓扑关系和耦合逻辑。

但车站的某个属性并非一定为图形属性或逻辑属性。本案特别地为此做出优化：本案只需要输入可以独自或和其他属性一起提供渲染或联锁逻辑所需信息的车站属性。也就是一个车站由完整描述车站的最小属性集合所描述，而之后业务中所需的所有信息都将由这个集合推导。如此以来用户不必输入非必要的冗余信息，提升了用户体验。基本上，一个车站是由数个 Signal 和数个 Node 构成的，所以，车站属性从组件上可分为 Signal 属性和 Node 属性。表 4.1中为 Node 的属性，表 4.2为 Signal 的属性。

这些信息保存在用户上传的车站描述文件中，并储存在数据库车站表 (station) 的 yaml 行内，其会在运行时服务被反序列化成所需的各种对象。

除了承载着车站属性们的车站描述文件，车站还有一些其他信息保存在数据库的 station 表中。譬如创建时间、创建人、是否为草稿、修改时间等等。详情可见于第六章。

(1) 新建车站

表 4.1: Node 属性

属性	作用	图形属性	逻辑属性
NodeID	唯一确定 Node	✓	✓
NodeKind	类型		✓
TurnoutID*	所属道岔		✓
TrackID	所属轨道电路		✓
LeftAdj*	左邻 Node		✓
RightAdj*	右邻 Node		✓
ConflictedNode*	抵触节点		✓
Line	渲染线段	✓	
Joint	绝缘节类型	✓	

* 表示该属性有数个

表 4.2: Signal 属性

属性	作用	图形属性	逻辑属性
id	唯一确定 Signal	✓	✓
SgnKind	信号类型	✓	✓
SgnMount	安装方式	✓	
Pos†	安装位置	✓	
dir†	左右朝向	✓	✓
side	上下两侧	✓	
ProtectNodeID	防护 Node	✓	✓
TowardNodeID	朝向 Node	✓	✓
Btns*	信号机安按钮	✓	✓
JuxSgn†	并置信号机		✓
DifSgn†	差置信号机		✓

* 表示该属性有数个

† 表示该属性非必须（可省略）

新建车站是用户输入车站信息（上述提到的承载车站属性的车站描述文件和一些其他信息）并将其插入数据库的过程。表现层通过调用 api 服务 Mutation 中的 create_station 来创建新车站。新建车站时需要的输入的项见表 4.3

表 4.3: 创建车站输入

属性	解释
title	标题
description*	注释
draft	是否草稿
yaml	车站描述文件

* 表示该属性非必须（可省略）

(2) 获得车站：

获得车站是通过查询数据库，使用户取得车站信息的过程。当表现层调用 api 服务 Query 中的 station 方法时，该方法会通过用户输入的 station id 在数据库中查找车站从 API 服务中获得的车站返回车站的所有原始信息，需要注意的是，该车站信息是表现层控制面板中用于获得车站列表、查看车站信息使用的。不是在初始化实例时渲染车站平面图用的，渲染车站平面图用的是运行时服务的“查询车站布局”。

4.1.3 实例

(1) 创建实例

Instance 是 Station 的实例，所以 Instance 需要 Station 数据进行初始化。但要想创建 Instance，还需要一些必要的信息，譬如 Instance 类型，Instance 支持两种类型：练习和考试，二者都需要一些描述该类型的详细内容，比如 Instance 需要和用户交互，所以需要在创建实例时指定实例的用户、详细的输入项见表 4.4。

实例用户和实例创建者不一定相同，在大多数练习场景中自然是相同的，但是在考试场景中，考试实例一般是由管理员（教师）创建给普通用户（学生）的。因为本案是分布式架构，因此整个系统不一定只有一台 Executor，因此创建 Instance 时需要指定一个 Executor 以执行该 Instance。另外，用户创建实例是还需要为其指定标题与描述（可选）。

创建实例时可以选择是否指定开始时间，若不指定则缺省为即时开始。一般练习的场景中，实例是即时创建的，但在考试的场景中，教师通常会提前配置好未来的考试。在创建实例时指定实例的开始时间（也必须指定结束的时间）。

表 4.4: 创建实例输入

属性	解释
title	标题
description*	注释
player	用户
station_id	车站 id
executor_id	运行时 id

* 表示该属性非必须（可省略）

当新建实例之后，API 会为实例生成唯一的 UUID（Universally Unique Identifier），UUID 是用于计算机体系中以识别信息数目的一个 128 位标识符，UUID 根据标准方法生成，不依赖中央机构的注册和分配，UUID 具有唯一性，这与其他大多数编号方案不同。重复 UUID 码概率接近零，可以忽略不计。因此 UUID 十分适合用在分布式系统数据表的主键。因为服务集群中即使有多个数据库、多个服务节点也能保证某个实例的主键是世界上唯一的。uroj 使用 PostgreSQL 的 gen_random_uuid 函数生成版本 4 的 UUID。

4.1.4 用户

4.1.5 班级

4.2 Auth 服务

本案设管理员和用户两种用户身份，不同的服务需要响应的权限才能运行

4.2.1 JWT

JSON Web Token(JWT) 是一个开放标准（RFC 7519），用于创建具有可选的签名和/或可选的加密的数据，其载荷持有 JSON。token 使用私人秘密或公共/私人密钥进行签名。服务器可以生成一个包含用户身份和用户 id 的 token，并将其提供给客户端。然后，客户端可以使用该 token 来证明其身份。

本案的 uroj-common crate 中封装了 JWT 相关的函数，其 claim 定义为

```
pub struct Claims {
    pub sub: String,
    pub exp: i64,
    pub role: String,
}
```

其中 sub 是用户 ID，exp 是 token 有效期，role 是用户身份，当用户登入时，生成一个 claim 并将其编码成 token。在 web 端将该 token 存入 cookie 中，在之后的所有请求头中携带 token 进行访问，服务端就可以将 token 解码成 claim，从而得知用户 id 和用户身份，从而判断用户是否有请求该方法的权限。

这个过程可以很形象的理解成当学生或教师进入大学时发放相关证件（学生卡/教职卡），卡片上记录着持卡人的信息和其身份（学生/教师等）在学校内需要验证身份的时候就可以使用证件来验证身份。token 就是一种这样的证件，由服务端签发，由 web 端持有，在服务端需要验证身份时使用的。

4.3 运行时服务

运行时是供实例执行的运行时（runtime）环境，一个运行时中可以执行多个实例，所有的实例会被管理在一个 HashMap 中。实例在初始化时被插入该 HashMap，在结束时移除。

4.3.1 实例 (Instance)

实例是运行时中运行的基本单位，运行时服务的含义就是运行实例的服务。一个实例由某个车站所实例化而来，在运行时中和表现层可以直接与相应的实例进行交互。若拿做菜作类比，车站是“菜谱”、实例是“菜肴”、“上菜”是实例初始化并运行，与实例交互就是“吃菜”一台实例的生命周期如下图所示：

一台实例只有一个逻辑用户，这是显而易见的，现实中一台终端只能同时由一个人操作。但本案还支持管理员控制和状态共享。管理员控制是允许管理员对任意一个运行中的实例进行最高权限的操作，包括普通用户的所有权限还有设置隐患（故障），任意生成列车等不和现实逻辑但有益于提高教学效率的操作。

4.3.2 状态对象

状态对象是状态机的组件，由 Signal（信号机状态对象），Node（结点状态对象）和 Train（车辆状态对象）组成，状态对象中保存着相应车站信号设备的实时状态。

(1) 信号状态

譬如，Signal 中的二元组 filament_status 表征灯丝状态，灯丝状态可取表 4.5。Signal 中的 state 属性表征信号机点灯状态，其可如表 4.6所列。

表 4.5: 灯丝状态定义

属性	含义
Normal	正常
Fused	熔断
None	空

(2) 结点状态

与信号的状态类似，结点状态枚举参见表 4.7，但需要注意的是在程序中还定义了锁闭枚举（Lock），但 Lock 并不参与业务逻辑，真正表征锁闭状态的是 Node 状态

表 4.6: 信号机状态定义

属性	含义
L	绿
U	黄
H	红
B	月白
A	蓝
UU	双黄
LU	绿黄
LL	双绿
US	黄闪
HB	红白
OFF	灭灯

对象中的 is_lock 属性。该 Lock 枚举仅仅用于当 Node 锁闭时序列化成为状态帧向表示层发送。

表 4.7: 轨道区段状态定义

属性	含义	轨道电路状态
Vacant	空闲	调整
Occupied	占用	分路
Unexpected	异常	断轨、分路不良等

结点状态可以由用户输入和车辆运动两种事件决定，相较信号状态更复杂一些，因此用状态转移图来表示，如图 4.1（不考虑非理想状况，如司机冒进信号）：

其中状态变量{a, b, c}分别为：a: 0 为 vacant、1 为 occupied、2 为 unexpected，b: 是否锁闭 c: 是否曾占用。图中未绘出异常状态，但，图 4.1 中的任意一种状态均可发生异常从而使 node 状态变为 unexpected.

(3) 车辆对象

Train 中除了有 id 属性外，还有以下两个属性：

(4) past_node ，其记录着一列车辆所历经的结点。用于解锁区段结点以及判断考题得分。车辆对象可以自动的行进，如同一个完美的司机。

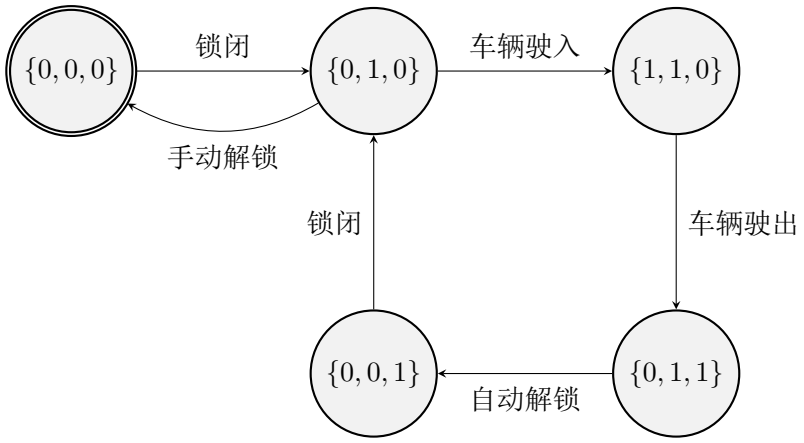

图 4.1: 结点状态转移图

(5) process，不大于 1 的浮点数，表示车辆在当前结点的进程，比如 0.5 表示车辆位于当前节点正中

4.3.3 实例组成

一个实例基本由 fsm、topo、layout 三个独立的部分构成。在实例初始化时会通过 Station 信息同时生成这三个部分。

(1) topo

topo 保存一个实例所有的拓扑关系，包括车站图（即联锁关系，包含 R 关系和 S 关系），并置信号机映射，差置信号机映射，以及独立按钮映射，topo 能表征一个实例的各种组件（信号机、节点和按钮）在联锁逻辑上是如何耦合的。比如 R 关系表示了轨道结点之间是怎么连接的。

通过 topo，运行时可以静态地从车站的拓扑关系上找到一条可能的进路。再通过后续的判断，来确定这条可能的进路是不是进路。

(2) 状态机（FSM）

FSM（finite-state machine）即有限状态机, 保存了一个实例所有的状态对象，包括上述的信号机、节点和车辆，并且管理整个车站的状态。后文将能改变 FSM 状态的因子称为事件 (Event)，而每发生一个事件，都有可能会导致实例向表现层发送一个状态帧。

(3) 布局（Layout）

实例中的布局对象是车站布局的载荷，即在用户请求车站布局时向表示层发送的车站布局信息。布局在实例初始化时会和状态机同时生成，实际上，关于表现层的车站布局信息有两种方案，其一是不在初始化实例时在实例中储存布局信息而在用户请求车站布局时再计算得出。其二是本案采用的，在实例初始化时同步计算布

局信息并保存，当用户请求时直接返回布局信息。这样做的好处是，以空间换时间，若有大量用户同时访问一个实例，或一个用户多次访问一个实例（如刷新页面），表现层请求车站布局用来渲染车站平面图时，多次计算布局信息会造成不必要的时间开销。一个布局由一组结点布局对象（NodeData）、一组信号布局对象（SignalData）和一个标题构成，标题用于渲染车站名。NodeData 和 SignalData 用于渲染结点和信号机。

表 4.8: 结点布局对象结构

属性	作用
NodeID	唯一确定 Node
TrackID	所属轨道电路
LeftP	左端点
RightP	右端点
LeftJoint	左端绝缘节
RightJoint	右端绝缘节

表 4.9: 信号布局对象结构

属性	作用
id	唯一确定 Signal
SgnKind	信号类型
SgnMount	安装方式
Pos	安装位置
dir	左右朝向
side	上下两侧
ProtectNodeID	防护 Node
Btns	信号机按钮

(4) 考试管理器（ExamManager）

考试管理器是可选属性 (Option)，若且唯若实例为考试实例时才会有此属性，管理考试题目、考试进度以及考试分数相关的内容。一场考试包含数个题目，考试管理器在考题被完成、跳过、超时的时候，会向表现层发送状态帧，供 GUI 显示相关

的考试进度，对于每一道考题在数据库中是由题目外键和实例外键构成的联合主键，意即即使对于参加同一场考试的不同用户，其题目是相同的，对于一道相同的题目，可能有会出现在很多名用户的考试管理器中。通过实例 id 和题目 id，可以唯一的确定一个实例中的一道考题。考试管理器也可以凭此将用户的成绩信息上传至数据库中。详细的数据结构可以参见第六章。

4.3.4 获取实例信息

表现层 (用户) 能从实例上取得的信息，有布局信息、考题信息和状态信息三种，布局信息和考题信息是静态的、一次性的。而状态信息是实时的，动态的。布局信息用来正确地绘制车站的布局，考题信息用于向用户下达考试实例的题目，状态信息用来更新车站上信号设备的状态。布局信息透过布局对象 (layout) 来传递，考试信息通过考试管理器发送，状态信息透过状态帧 (GameFrame) 来传递。

uroj 的运行时，无论在哪一层，布局和状态都是无耦合的。这意味着表现层需要单独地查询车站的布局和订阅实例状态的更新。

(1) 查询车站布局 (Query Station Layout)

下面将介绍运行时是如何将上述的布局信息呈递给表现层的，而我们将在第九章看到如何利用表 4.8 和表 4.9 中所列之属性在网页上正确地渲染出车站平面。

运行时提供的车站布局接口是 station_layout 方法，该方法会返回一个请求车站的 layout 的副本。当请求该方法时，需要一个传入一个字符串作为 id(UUID) 参数，用来指明所请求的 layout 是哪个实例的 layout，运行时会在当前运行的实例中寻找用户所输入的 id 所对应的实例，如果没找到则说明输入的 id 不是某个正在运行的实例。如果找到了则把该实例的 layout 克隆并返回。

(2) 查询考题信息 (Query Instance Question)

若且唯若实例为考试实例时（即实例的考试管理器不为空），表现层会查询实例的考题信息，考题信息

(3) 订阅状态更新 (Subscribe Status Update)

为保证表现层的状态能实时的被更新渲染，状态更新应该是长连接的单向流，uroj 采用了 graphql 的 subscription。其能通过 websocket 协议源源不断的向订阅者 (表现层) 发送数据。

状态帧 (GameFrame) 即是状态更新的载荷 (Payload)，即在需要更新表现层车站状态渲染的时候向表现层发送的“新状态”，譬如某个信号机的灯光颜色，某个车辆的新位置，考试题目的完成，等等。但不是所有的事件都会导致产生并发送状态帧，

譬如轨道曾占用是用于解锁逻辑判断使用的，而不需要在视图上有任何表示，所以轨道曾占用就不会产生状态帧。

状态帧目前分为 6 种，UpdateSignal：更新信号，UpdateNode：更新结点，UpdateGlobalStatus：更新全局状态 MoveTrain：车辆移动，UpdateQuestion：更新题目，InstanceFinish：实例结束。其中 UpdateSignal 和 UpdateNode 分别包含 id 和 state 两个属性，表示更新设备的 id 和新状态。UpdateGlobalStatus 则是 UpdateSiganal 和 UpdateNode 的数组。而 MoveTrain 有 id 属性表示被移动车辆的 id，node_id 属性表示车辆所处的结点编号，process 属性，表示车辆在当前节点的位置，为小于 1 的浮点数，指的是车辆相对于结点的进程（即车辆走过了 node_id 结点的百分之几）。这些属性是与 Train 状态对象的属性相同的，另外为了正确的渲染车辆的位置，MoveTrain 状态帧还有一个属性名为 dir，意思是方向。结合 process 就能知道此时车辆是在结点从左到右百分之几还是从右到左百分之几的位置处。UpdateQuestion 只有当实例为考试实例时才会被发送，可以将某个编号的考题更新至：已完成、已超时、已跳过三个状态。另外，和前四种状态帧不同的是，前四种状态帧是由状态机发送的，而更新考题状态帧是由考试管理器发送的。

状态更新的接口是 game_update 方法，该方法会返回一个内容为 GameFrame(状态帧) 的流。另外需要考虑的是当表现层订阅状态更新后，其会不断的获得实例的最新状态变化。但也仅限于状态变化，因为如果没有一些途径让表现层得知订阅状态更新时的初始状态，就不能正确地表现整个车站的所有状态。因此在订阅状态更新时收到的首个状态帧一定是 UpdateGlobalStatus，用来渲染请求状态更新时车站的状态，后续的状态帧都是在这个初始状态之上的状态改变。这便是 UpdateGlobalStatus 状态帧存在的意义，另外，当有需求重置整个实例时，UpdateGlobalStatus 这个状态帧也会被发送。

可能会发现，前文所定义的状态帧中的状态都只包含了新状态，而没有包含旧状态。这和某些事件驱动应用中的“事件”不同。在本案中，表现层对于状态更新的策略是乐观的。意即表现层总是认为：自己接收到的状态帧中包含着最新的状态。因此状态帧中不需要加入旧状态或者时间戳以保证状态变化的连续。

4.3.5 实例初始化与运行

无论用户是想要进行前文提到的请求车站布局 (layout) 亦或者是更新车站状态，最大的一个前提是实例要处于运行状态。这是理所当然的，就像你不能品尝到一碟没有上菜的菜肴一样。你需要先让实例加载并运行在运行时内，才能获取实例的车

站布局或者更新实例状态。

在第五章，我们能够在 api 层中定义一个实例，预约实例运行的时间、配置实例相关的信息，并将这些信息存在数据库的 Instance 表中，那么在实例所指定的运行时 上，就可以运行该实例。要想运行实例。用户需要输入实例 ID（UUID）。而后运行时会通过数据访问层 (uroj-db) 在 Instance 表中查找相应的实例。如果未找到，则说明访问的实例 ID 不合法。如果寻到对应的实例，则会验证实例的相关信息：如果访问实例的用户没有权限（没有 Guest、Player 或 Operator 权限），则返回 forbidden 禁止访问。若用户有权限。那么还需要验证开始时间。因为在表现层，未满足开始时间要求的实例根本不会渲染开始入口（相关按钮），这里的验证看似冗余而没有必要，但其实不然，uroj 的各个模块间的耦合策略是悲观的，意思是，运行时 不应该信任前端（表现层）传来的数据一定不会包含违反开始时间约束的实例访问请求。

当一切验证完成，运行时 便会将 yaml 反序列化成 RawStation 对象，用数据库中查到的实例信息，和 RawStation 对象在运行时中新建实例。在这个过程中，Instance 的 new 方法会将传入的 RawStation 转化为 fsm, topo 和 layout。

下面说明其中某些属性的推导过程，首先给出一个定义和一个推论：

定义 1. 对于一个信号机，我们称其朝向的结点为该信号机的朝向结点，称其背向的结点为该信号机的防护结点

推论 1. 某个信号机的朝向等于其所防护区段的端，相反于列车行进方向和其防护区段相对信号机的位置。

举例说明：

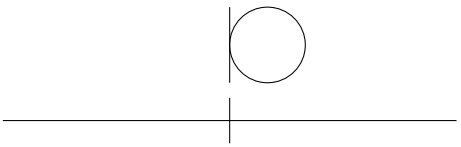

图 4.2: 例子

上述信号机朝左，因此其左边的结点为该信号机的朝向结点，右边的结点为其防护结点，并且，该信号机防护其右侧区段的左端，限制来自右行的调车。有了这个推论便可以由原始数据推导出下列信息。

(1) 信号机布局对象的位置和朝向

应该能注意到表 4.2 中的 Pos 和 dir 属性是可选的。这是因为其中的 pos 和 dir 是缺省值。即信号机的位置和朝向是可以从其他信息中推断出来的，这两个值如果

留空则自动推断，如果不留空则有限使用 pos 和 dir 作为信号机的位置和朝向，那么应该如何推断呢？

一般情况下，信号机位于两个轨道区段的衔接处, 绝缘节的旁边，因此通过表 4.2 定义的 ProtectNodeID 和 TowardNodeID 可以找到信号机所对应的防护结点和朝向结点，那么由推论 1 必然有信号机的朝向为从防护结点到朝向结点的方向，如果防护结点和朝向结点邻接（即在 r 关系中存在防护结点到朝向结点的边）那么在有向图 r 中就能得知信号机的方向。换言之如果防护结点和朝向结点不邻接，则说明违反了定义 1，说明车站描述文件出错。

对于信号机的位置而言是同理的，由推论 1，若信号机朝左则一定位于防护节点的左端点，若其朝右则一定位于防护结点的右端点。而左端点或右端点的坐标是在 RawNode（见表 4.1 ）中定义的。

(2)　结点状态对象的防护信号机

对于 FSM 的结点状态对象来说，需要知道一个 Node 的左端信号机和右端信号机，（这里需要明确一点：Fsm 的 Node 状态对象中的左右端信号机，都应指的是防护本结点的信号机。）可以通过 RawStation 在信号机上定义的防护结点，结合信号机的朝向就可得知：由推论 1 若信号机朝左，则其防护结点的左端信号机是该信号机，若信号机朝右，则其防护结点的右端信号机是该信号机。

4.3.6　结束实例

相比实例的运行，结束实例要简单得多。结束实例分为两种，自动结束和手动结束。对于考试实例而言，可以自动结束或手动结束。对于练习实例则只能手动结束。对于考试实例而言，结束实例时会将考试管理器（ExamManager）的问题成绩同步至数据库 instance_questions 表中。对于练习实例没有什么额外的操作。

因为实例保存在 HashMap 中，因为 rust 语言的所有权和强制 RAII 的特性，所以不需要对实例进行析构。只需要将 Instance 从 HashMap 中删除就可以结束实例。

在结束实例之后，会发送一个实例结束状态帧，以在表现层提示用户实例的结束。

4.3.7　新建进路

在详细说明建立过程之前，需要强调的一点是，新建进路过程的一个重要的性质为原子性：和取消进路的分段过程不同，建立进路时必须保证所有轨道节点要么全部锁闭，要么全部不锁闭（建立失败），不能出现部分节点锁闭部分结点不能锁闭的情况。

若不考虑竞态条件（多线程时），先检测一个可能进路中所有节点是否满足封闭条件，再决定条件不满足而建立失败或者对所有结点统一进行锁闭以及对其进路扩展集进行征用。

若不事前检测进路锁闭条件，则需要在封闭结点到中途遇到无法锁闭之结点时对之前锁闭的所有结点进行回滚（Rollback）。事实上，数据库事务的原子性便是通过这种方案保证的。但本案出于建立进路的性质考量，采用第一种方案。

本案采用了为培训系统改良的新建进路算法，新建进路可以分为几个过程：查找结点、寻径、进路约束检查、进路条件检查、封闭区段、点灯。

(1) 查找结点

本案排选进路的核心算法以始终结点为输入，但实际上用户的输入却是按钮，如此一来程序便需要从用户的输入得知用户真正想要建立的进路是从哪一个结点到哪一个结点的。

前文可知本案所定义的按钮有两种，一为信号机按钮，二为独立按钮。独立按钮自然有其到某个结点的映射。但信号机按钮所映射的实体是信号机，而信号机有两个属性都和结点相关（防护结点和朝向结点）。那么当用户点击信号机按钮时，究竟哪个结点才是用户想要建立进路的起点，哪个结点才是终点呢？

不难发现，对于起点来说，不论建立哪种进路，起点总是始端信号机的防护结点。真正有问题的是终点的判断。不同类型的进路其终点和终端信号机的相对位置不同，就算是同种类型的进路，若终端信号机有并置或差置，位置就又有不同。

枚举所有种类的进路，可以总结出终端结点的规律见表 4.10。

(2) 寻径

在站场中查找路径本案中采用了 petgraph 这个 crate 提供的 A*算法，将起点和终点输入，便可能在有向图 R 中找到一条路径。但这条路径不一定是我们要找的进路，为了使路径满足进路链的定义。还需要保证路径中的所有结点都互相没有 S 关系，即任意两结点在 S 图中都不存在边。若这一点也满足，则称这条路径为“可能的进路”。

(3) 进路约束检查

为什么我要将寻径得到的路径成为“可能的路径”呢，这里使用两个例子来说明，第一个例子来分析图 4.3 情况：

若用户点击 D1，D3，则不应该存在合法的进路。若没有额外的约束，则从结点 1 到结点 5 确实存在一条完全符合进路链定义的进路：$1 \to 3 \to 5$。但显然这是不应该存在的。

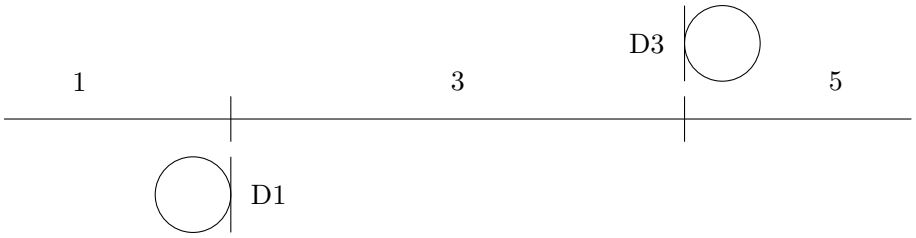

图 4.3: 例子 1

第二个例子：假设想要建立一接车进路，无疑地，用户需要输入的始端信号机是进站信号机，终端信号机是与始端信号机反向的差置发车信号机。若使用同向的发车信号机作为终端信号机按钮输入，则不应该存在进路。但一个问题是，一对差置的接车信号机其朝向结点是相同的，而根据要求，接车进路的终点正是终点信号机的朝向结点。因此若不加限制则会造成若终端按钮点击的是两个差置的接车信号机的任意一个均可以成功找到合法的接车进路。这就是并置和差置信号机引发的问题。

为解决这些问题，才需要引入方向约束，方向约束是本案中保证进路映射唯一性的一种约束。进路映射唯一性的含义是：一个进路输入只能找到唯一的进路并且能查到某条进路的输入有且唯有一个。方向约束的含义是：对于一条可能进路的始/终端方向必须和欲建立进路的列车行进始/终端方向相同，故而方向约束由两个约束构成：始端方向约束和终端方向约束。

显然地，欲建立进路的列车行进始端方向总是始端信号机的朝向的反向（称为信号机的防护方向），这点很好理解，司机进入进路时一定是面朝进路的始端防护信号机的，那么车辆的行进方向便是始端信号机朝向的反向。与终点规律相同欲建立进路列车行进终端方向也需要分类讨论，与终点选择的规律一并总结在表 4.10 中。

在查找进路时我们把站场图中寻到的路径成为可能进路。一条可能进路的始终端方向必须同时满足始端方向约束和终端方向约束，那么这条可能进路才能成为进路。

对于第一个例子，就可以使用始端方向约束解决，对于可能的进路 $1 \to 3 \to 5$, 其方向是向右的。但始端信号机 D1 朝右，其防护的车辆必然向左行驶，则不满足始端方向约束，因此 $1 \to 3 \to 5$ 不是合法的进路。

对于第二个例子，就需要用到终端方向约束来解决，在第二个例子中，起点按钮是列车按钮（进站信号），终点按钮是列车按钮（出站信号）终点是朝向结点，到这里和上述推论一样没有问题。当终端按钮按下的是同向的出站信号机按钮，终端信号朝向就会和车辆行驶方向相反，则违反了终端方向约束，不合法。只有点击差置的反向出站信号机按钮，其信号机朝向才和行驶方向一致，成为合法进路。

表 4.10: 终点和方向

起点按钮	终点按钮	进路类型	终点	终端方向
通过	列车	通过	防护结点	终端信号朝向
通过	列车终端	通过	LZA 映射结点	始端信号朝向
列车 (进站信号)	列车 (出站信号)	接车	朝向结点	终端信号朝向
列车 (出站信号)	列车 (进站信号)	发车	防护结点	终端信号朝向
列车 (出站信号)	列车终端	发车	LZA 映射结点	始端信号朝向
调车	调车	调车	朝向结点	终端信号反向

(4) 进路条件检查

进路条件即允许开放进路的站场状态。即对进路中的结点状态和途径的信号机的状态进行判断。对于信号机而言，应处于防护状态，因为采用了图论算法，所以不需要像传统继电联锁逻辑一样验证敌对信号 []，对于结点而言，需要验证其必须处于空闲状态，不得被封闭 (锁闭)，不得被征用。这里将锁闭和征用区别开来。封闭进路中的点称为锁闭，使用 is_lock 表征，封闭进路链扩展集中的点成为征用，使用征用计数器 used_count 来表征。判断征用这里采用了和引用论文中不同的方法，这么做的好处是降低了算法时空复杂度。每当一个结点被征用，则征用计数自增一，因为被征用的点不能成为进路中的点但可以成为进路扩展集中的点，所以只要征用计数器大于零，则说明该结点被至少一条进路征用，不能建立包含该结点的进路。

(5) 封闭区段

通过对进路的遍历，锁闭进路的点，重置点的曾占用 flag (曾占用 flag 是供进路解锁时三点检查法判断区段是否曾占用后又出清使用的，因此要在建立进路时重置) 并同时将进路的点的扩展集（即和该点有 S 关系的点集）中的点的征用计数器自增。

(6) 点灯过程

点灯可以分情况讨论，当欲建立进路是发车进路时：需要点亮发车信号机的允许信号（不考虑区间上的状态）。

当欲建立进路是接车进路时，需要点亮进站（反向进站）信号机的相应允许信号，但和发车不同之处在于，进站信号机所点亮之信号与接车的终点相关。因此本案采用特殊结点标记来判断接车进路会接车到哪种结点上。

通过进路和进站接车在点灯上的行为相似，通过进路和进站进路的点灯逻辑可以用表 4.11来表示

表 4.11: 通过和进站信号

起点	终点	进站信号机
咽喉	咽喉	L
咽喉	站线	UU
咽喉	18 号道岔以上站线	US
咽喉	正线	U

另外，得益于图论算法的强力驱动，本案还支持建立长调车进路，长调车进路开放信号时，需要开放途经所有朝向和进路方向相背的调车信号机。由推论 1，如果途经的某个结点有防护调车信号机，若进路车辆向左行驶则该信号机位于结点的右端，若向右行驶，则信号机位于结点的左端。因此通过调车行驶方向，就可以取到所有途经迎面的调车信号机。

4.3.8 总取消进路

因为 uroj 创建进路后只改变状态机中的状态，而不对创建的进路进行记录，因此当用户输入一个起点按钮以取消进路时，需要先从实例中找到一条已经建立的进路。从用户所输入的信号机按钮可以得知：用户想取消进路的方向，和用户欲取消进路的起点。这是因为对于起点来说，不论建立哪种进路，起点总是始端信号机的防护结点，始端方向总是始端信号机的反向。因此有以下算法：

i. 初始化一个 vector，压入起点

ii. 找到 R 有向图中从该 vector 顶部结点出发且符合指定进路方向的第一条边

iii. 如果该边的终点锁闭且空闲，则压入该终点至 vector 中，并跳转至 2

iv. 若 3 的条件不满足，则返回 vector

我们称该 vector 为一可能的进路。该算法的作用仅是找到从给定起点向给定方向的最长连续锁闭结点。

那么，显然地，对于上述的可能进路。若点击终端或者中途的某个信号机，依然可以得出其一部分作为可能进路。与创建进路类似，取消进路也要满足原子性，显然不能只取消进路的一部分。所以这种可能进路是不允许成为可取消的进路的。

uroj 采用始端信号机状态约束来解决这一点，如果用户输入的信号机没有开放，其必然不是某个现存进路的始端信号机。

但是对于长进路，途经的调车信号机必然是开放的，看起来没办法使用始端信号机状态来验证了。但没关系，取消进路时会验证接近区段的状态，即接近区段必须空闲。这个约束是联锁逻辑所要求的。在验证此逻辑同时验证接近区段必须未锁闭就能解决问题。因为如果一个区段锁闭则其一定存在于某个进路中。从而一定不是某个进路的接近区段。

那么接近区段该如何得到呢？显然地，有了始端信号机，始端信号机的朝向节点其实就是进路的接近区段。

4.3.9 总人解进路

总人解是当车辆已经进入接近区段时采用的解锁方法，但实际上其适用性是包含了总取消的。因此其逻辑和总取消大体上相同，只是在检查完解锁条件后需要延迟一定时间（本案采用 3 秒）再解锁。

4.3.10 区故解进路

在现实联锁中，区间故障解锁是需要登记并输入口令的，但作为一款仿真模拟应用，显然是不需要的。因此其逻辑和总取消进路也差不多。不过需要表示层弹出对话框要求输入口令，其中因为口令是预设的 (123)，所以只需要在表现层判断即可，以优化后端的逻辑。

4.3.11 调/列车辆

车辆有自动创建和手动创建两种，在考试模式中，当用户创建的进路的起点同时是当前题目所要求进路的起点时，在进路创建成功的同时会自动的在接近区段放置车辆。在练习模式中，车辆不会自动被放置。用户需要手动选择结点以创建车辆。创建车辆分为两步

(1) 生成车辆：车辆将会被生成在指定结点的一半处，具体过程是将生成点插入历经结点中，并且令 process = 0.5。

(2) 启动自动行驶任务：每个车辆都会有一个 tokio 任务，用于自动判断执行列车的行驶，该任务会循环执行，首先先向左方自车辆所在的结点找到下一个进路结点（类似取消进路中所列的寻找进路的算法，不过在第三步直接返回），如果找到的话则尝试移动车辆至该节点，如果没找到则向右方寻找。若左右两方都不能找到进路结点。则车保持不动。若左右两方有一方是进路结点，则将尝试移动车辆至该结点，尝试移动车辆的过程用语言叙述比较复杂，故此绘流程图 4.4,

图中，靶点即目标结点，今点即列车当前所处的结点。当 process = 1 时说明列车已经行至今点的末尾，可以继续进行列车是否能够驶入靶点的判断。若不足 1，则说明尚未行驶到今点的尽头，应继续前进一个单位速度的距离。

确认靶点和今点的位置关系是为了判断物理上列车可以由今点驶入靶点。若根本不邻接便不可能驶入靶点。

确认防护信号的状态是为了遵守列车行驶规则，即只有允许信号开放才能驶过信号机。若信号不开放则也不能驶入。若该结点根本没有防护信号机则说明列车可以直接驶入。

判断完成后，将列车驶入靶点，有如下几步：

i. 将靶点状态对象的状态置为 Occupied

ii. 将今点状态对象的状态置为 Vacant

iii. 将今点状态对象的曾占用标记置为 true

iv. 将靶点压入 past_node 中

v. 将列车 process 置为 0

4.3.12 状态共享

状态共享指的是当多个用户访问一个实例时，他们都能实时的观察到该实例的所有状态。但是逻辑上只能存在一个使用者，所以我们要对实例的操作权限做出一些限制。

对于存取某个实例的用户来说，可以分为三类身份：Guest、Player 和 Operator。其中，对于所有的管理员用户都自动的拥有实例的 Operator 权限。而 Player 用户是每个实例唯一的，也就是该实例的逻辑使用者，但为了管理员能对所有实例实时控制，因此管理员也拥有一个实例的所有权限，即 Operator 是实例的事实使用者。

而对于 Guest，所有使用 token 访问实例的非 Operator 用户都是 Guest，Guest 不能对 instance 做出任何操作，而只能旁观。类似看直播一样的效果。设计 Guest 以及 token 的动机在于，一个实例的 Player 或者 Operator 可以将实例的 token 分享给其他用户，如此一来拿到 token 的用户就可以通过 token 访问到该实例，查看实例的实时状态。这种设计在演示教学上十分有用。比如教师可以使用本系统进行教学演示。将 token 分享给其学生们，学生们可以藉此观看教师的演示，而无需投影或者直

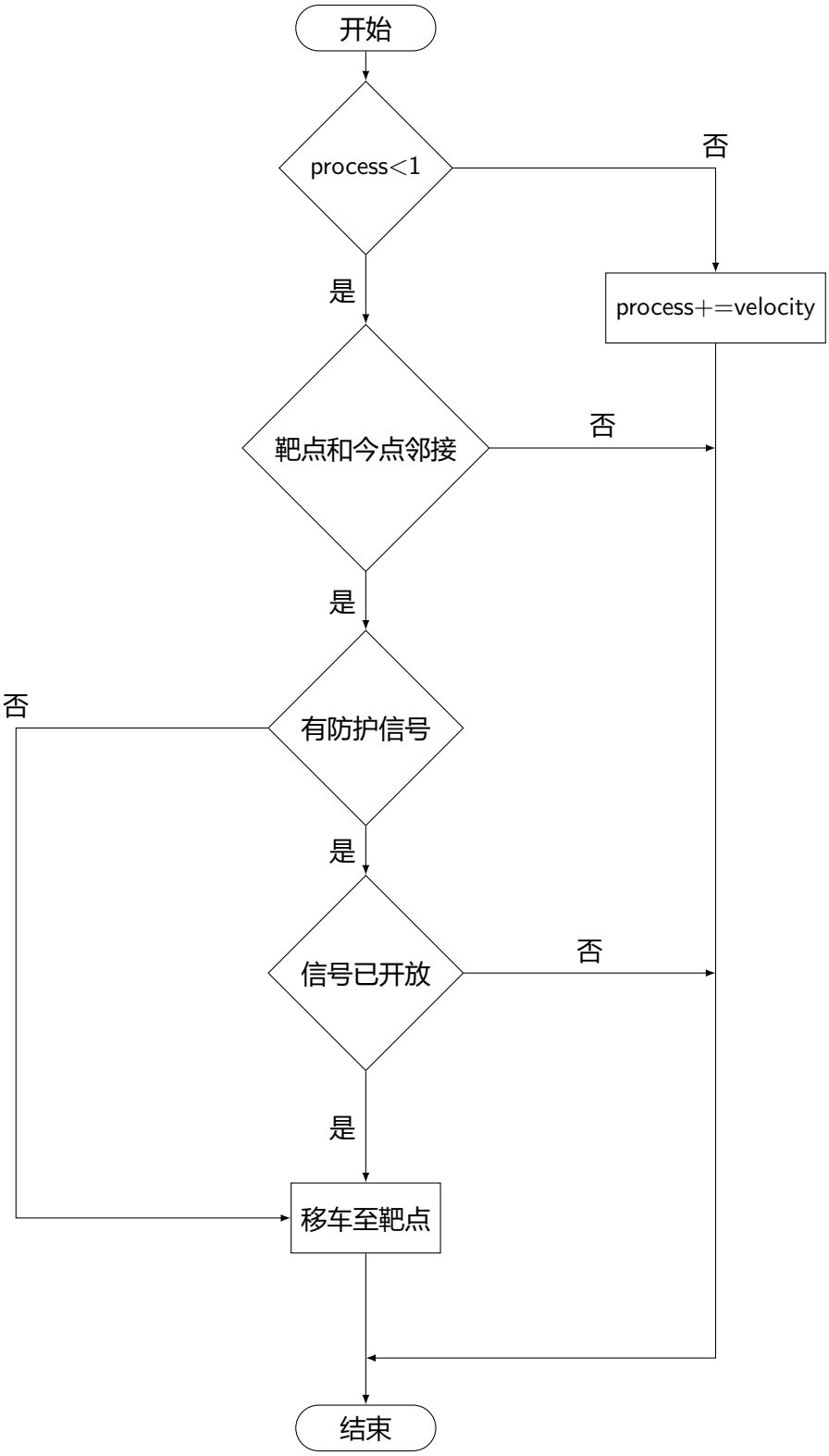
开始
process<1
否
是
process+=velocity
靶点和今点邻接
否
是
否
有防护信号
是
信号已开放
否
是
移车至靶点
结束

图 4.4: 移动车辆流程图

播演示。对于无论何种身份，其区别只是 Mutation 的权限有区别，而订阅车站状态更新都是一样的。

得益于 tokio 提供的 broadcast channel，uroj 可以将一个实例产生状态帧同时发送给不同的订阅者。而这也正是状态共享能够得以实现的基础。

4.4 性能优化

业务层采用了许多技术以提升应用性能

4.4.1 分布式 SOA

面向服务的体系结构（英语：service-oriented architecture）是一种分布式运算的软件设计方法。软件的部分组件（调用者），可以透过网络上的通用协议调用另一个应用软件组件运行、运作，让调用者获得服务。SOA 原则上采用开放标准、与软件资源进行交互并采用表示的标准方式。一项服务应视为一个独立的功能单元，可以远程访问并独立运行与更新。根据架构图，uroj 可以无限制的增加 api 和 runtime 的服务器数量，从而组成集群。如此便可以根据实际负载而提高应用吞吐量和算力。

4.4.2 优化查询

某些 GraphQL 查询需要执行数百个数据库查询，这些查询通常包含重复的数据，可以通过 DataLoader 来修复之。我们需要对查询分组，并且排除重复的查询。Dataloader 就能完成这个工作，facebook 给出了一个请求范围的批处理和缓存解决方案。

```
pub struct UserLoader {
    pub pool: Arc<PgPool>,
}

#[async_trait::async_trait]
impl Loader<String> for UserLoader {
    type Value = User;
    type Error = Error;

    async fn load(&self, keys: &[String]) ->
    Result<HashMap<String, Self::Value>, Self::Error> {
        let conn = self.pool.get().expect("...");
```

```
13          let users = UserData::find_many(keys, &conn)
14                      .expect("Can't get users' details");
15          Ok(
16            users.iter().map(|u|(u.id.clone(), u.into())).collect()
17          )
18      }
19  }
```

以本案中采用的 UserLoader 为例，每次在数据库中查询新 User 时，都会将查询到的 User 放到 Loader 的缓存中（HashMap），若再查询相同的 User，则会先在缓存中查找。若缓存中没有则再去数据库中查询。

4.4.3 异步

大多数计算机程序的执行顺序与它的编写顺序相同。第一行执行，然后是下一行，以此类推。在同步编程中，当程序遇到一个不能立即完成的操作时，它将阻塞，直到该操作完成。例如，建立一个 TCP 连接需要在网络上与一个 peer 进行交换，这可能需要相当长的时间。在这段时间内，线程会被阻塞。

通过异步编程，不能立即完成的操作被暂停到后台。线程不会被阻塞，可以继续运行其他事情。一旦操作完成，任务就会从其之前阻塞处恢复执行。

uroj 作为异步应用开发并采用 tokio crate 作为异步任务的运行时。并且采用了 rust 性能最优的 actix-web 框架作为 web 服务器。

5 表现层

web 端以 deno 为服务器，

5.1 实例绘图

本案采用 Two.js 作为 web 图形库，Two.js 是一个为现代浏览器设计的二维绘图 api。Two.js 与渲染器无关，其使同一个 api 可以在多种情况下进行渲染：webgl、canvas2d 和 svg。经过测试，webgl 的执行效率最高的，但是 webgl 会把矢量图转换成位图，导致图像解析度下降，因此本案中采用 svg 进行渲染，同时 svg 也是 Two.js 的缺省渲染器。

5.1.1 轨道节点绘图

一个轨道结点可以分为三个部分：线段、左端绝缘节、右端绝缘节。本案定义线段宽度为 4，渲染线段从起点到终点。这里举个例子，以展示 uroj 是如何使用 Two.js 进行绘图的。

```
const segment = two.makeLine(x1, y1, x2, y2)
```

上述代码表示的是使用 two.js 绘制一条从 (x_1, y_1) 到 (x_2, y_2) 的线段。

绝缘节图像从样式上可分为三种，包括终端绝缘节、一般绝缘节和侵限绝缘节。普通绝缘节是和结点线段正交的短线段，侵限绝缘节是和结点线段正交的短线段和以短线段为直径的圆，终端绝缘节是与结点线段正交的短线段，和与短线段正交的短线段。渲染绝缘节的关键在于将一定长度的旋转一定的角度使其与线段正交。

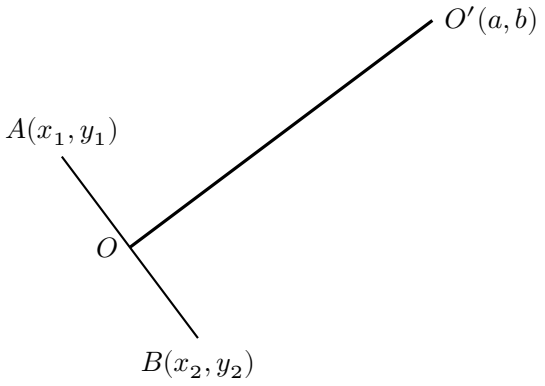

图 5.1: 绝缘节渲染

以普通绝缘节举例，如图 5.1 ，设 O 点为原点，假设 OO' 是轨道结点的线段，AB 是绝缘节。由图可知，想要正确地渲染绝缘节，关键在于求出 A 和 B 的坐标, 显

然地，因为 AB 与 OO' 正交，所以 $\overrightarrow{OA} \cdot \overrightarrow{OO'} = 0$：

$$ax + by = 0$$

而绝缘节的长度是定好的，假设绝缘节长 l 则又有

$$x^2 + y^2 = \frac{l^2}{4}$$

两个方程组联立，方程组正定，可求出：$x = \pm\frac{bl}{2\sqrt{a^2+b^2}}$, $y = \mp\frac{al}{2\sqrt{a^2+b^2}}$。方程组共有正负两组解，正好是绝缘节的两端。

对于侵限绝缘节而言，不过是在普通绝缘节上再绘制一以 l 为圆心的圆，而该圆是不需要旋转的。

5.1.2 信号机绘图

按照物理分类，信号机可以分为进站、出站、调车等信号机，还有高柱或矮柱等安装方式的区别。在二维的信号平面图上，信号机的方向也是需要考虑的。综合以上考量。本案中，为了正确的渲染一个信号机

5.1.3 时钟绘图

5.1.4 独立按钮绘图

5.1.5 功能按钮绘图

5.2 实例状态

5.3 panel 绘图

6 持久与数据层

6.1 数据库

本案选用 ProsgreSQL 作为数据库管理系统，PostgreSQL 是开源的对象-关系数据库数据库管理系统，在类似 BSD 许可与 MIT 许可的 PostgreSQL 许可下发行。本案的 SQL 结构设计如图 6.1：

6.1.1 表结构

(1) 班级 classes

```
CREATE TABLE classes(
    id          serial       primary key,
    class_name  varchar(50)  unique not null
)
```

主键 id 为自增整数，class_name 为班级名称

(2) 用户 users

```
CREATE TABLE users (
    id              varchar(30)     primary key,
    hash_pwd        varchar(60)     not null,
    email           text            not null,
    class_id        int  references classes(id) on delete set null,
    user_role       varchar(20)     not null,
    is_active       boolean         not null default 't',
    joined_at       timestamp       not null default now(),
    last_login_at   timestamp       default now()
)
```

解释：

- 主键为 id，类型为字符串，即用户自定义的用户 id
- hash_pwd 为加密后的用户密码
- email 为用户的电子邮箱地址
- class_id 是表 classes 的外键，表示用户所属的班级
- user_role 表示用户角色

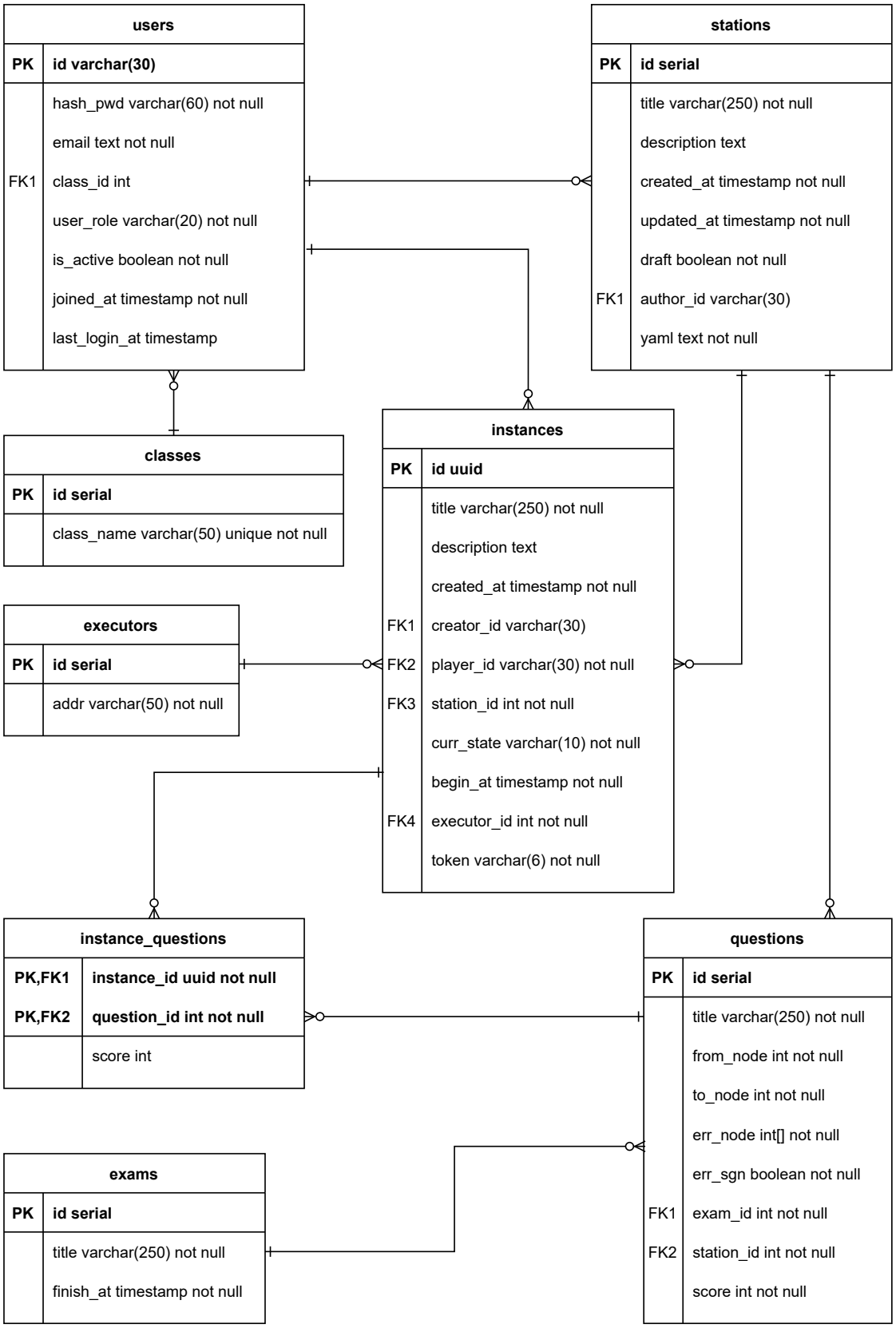

图 6.1: 数据库实体关系图

- is_active 表示账户是否可用 (未被禁用)
- joined_at 和 last_login_at 默认是插入时的时间

(3) 车站 stations

```
CREATE TABLE stations (
    id          serial        primary key,
    title       varchar(250)  not null,
    description text,
    created_at  timestamp     not null default now(),
    updated_at  timestamp     not null default now(),
    draft       boolean       not null default 'f',
    author_id varchar(30) references users(id) on delete set null,
    yaml        text          not null
)
```

解释：

- 主键为 id, 自增整数
- title 为车站的标题
- description 为可空键，表示车站的备注
- draft 表示是否为草稿
- author_id 是表 users 的外键，表示作者
- created_at 和 updated_at 默认是插入时的时间
- yaml 即为车站的描述文件内容

(4) 执行器 executors

```
CREATE TABLE executors (
    id              serial        primary key,
    addr            varchar(50)   not null
)
```

addr 是该 runtime 的地址。

(5) 考试 exams

```
CREATE TABLE exams (
    id           serial        primary key,
    title        varchar(250)  not null,
    finish_at    timestamp     not null
)
```

finish_at 表示结束时间，是考试特有的属性。

(6) 实例 instances

```
CREATE TABLE instances (
    id      uuid    primary key default gen_random_uuid(),
    title   varchar(250)  not null,
    description text,
    created_at  timestamp     not null  default now(),
    creator_id  varchar(30) references users(id) on delete ...,
    player_id   varchar(30)   not null  references users(id),
    station_id  int           not null  references stations(id),
    curr_state  varchar(10)   not null,
    begin_at    timestamp     not null  default now(),
    executor_id int           not null  references executors(id),
    token       varchar(6)    not null
)
```

解释：

- 主键为 id, 类型是 uuid
- title 为实例标题
- curr_state 表示实例当前的状态
- creator_id 是表 users 的外键，表示创建者
- player_id 是表 users 的外键，表示实例的用户
- created_at 默认是插入时的时间
- begin_at 表示实例的开始时间
- executor_id 表示该实例的运行时 id，是 executors 表的外键

- station_id 是表 stations 的外键，表示实例的车站
- token 是游客令牌

(7) 问题 questions

```
CREATE TABLE questions (
    id          serial        primary key,
    title       varchar(250)  not null,
    from_node   int           not null,
    to_node     int           not null,
    err_node    int[]         not null,
    err_sgn     boolean       not null,
    exam_id     int           not null  references exams(id),
    station_id  int           not null  references stations(id),
    score       int           not null
)
```

解释：

- from_node 是本题目进路自何处
- to_node 是本题目进路往何处
- err_node 是本题目的预设故障结点
- err_dgn 是本题目进路信号机是否出错
- exam_id 是表 exams 的外键，表示本题目属于哪个考试
- station_id 是表 stations 的外键，表示本题目属于哪个车站
- score 表示本道题赋分几何

(8) 实例问题（即“考题”）instance_questions

```
CREATE TABLE instance_questions (
    instance_id    uuid   not null references instances(id),
    question_id    int    not null references questions(id),
    score          int,
    PRIMARY KEY (instance_id, question_id)
)
```

本表使用 instance_id 和 question_id 两个外键作为联合主键，本表的一个记录表示某个实例（必然为考试实例）的某个题目得分几何。score 是可空的，因为在新建考试实例的时候就会在本表中新建进路，在完成某道考题的时候更新记录。

6.2 持久层

6.2.1 ORM

本案采用 ORM 以提升开发效率，ORM 是一种程序设计技术，用于将数据库的记录映射到程序语言的对象中，或者将对象映射到某个表中，其封装了 CRUD 的 SQL 语句操作，可以让开发者从表中直接读入一个对象。或者将一个对象插入某个表。效果上说，它其实是创建了一个可在编程语言里使用的“虚拟对象数据库”。

在本案的持久层 uroj-db 中，定义了程序的 DAO 层逻辑，封装了所有项目需要的数据库访问方法。以便供 Api, Auth, Executor 等服务复用。uroj 采用 diesel 作为本案的 ORM 库，将 DAO 层的各种结构体定义和上小结所定义的 SQL 表映射起来的，就是 diesel client 所生成的 schema。

这里简单介绍一下 diesel 的使用步骤。首先需要定义 migration，migration 可以简单理解为创建和删除表的 sql 文件。将上一节的表定义好后。使用 diesel 生成 schema，schema 是 diesel 使用 rust macro 定义的一些字段。之后我们需要定义 DAO 层的 struct，对于一个表一般而言需要两种 struct，一个是读取用一个是插入用，但需要 derive diesel 提供的相应的过程宏，这样就可以将 sql 表和 DAO 层 struct 映射起来，再使用 diesel 提供的方法进行 CRUD 操作。

6.2.2 连接池

连接池（英语：connection pool）是维护的数据库连接的缓存，以便在将来需要对数据库发出请求时可以重用连接。每次需要再打开一个新的数据库连接都是低效的，而且在高流量条件下会导致资源耗尽。可以使用连接池解决这个问题以提高在数据库上执行命令的性能。为每个用户打开和维护数据库连接，尤其是对动态数据库驱动的网站应用程序发出的请求，既昂贵又浪费资源。在连接池中，创建连接之后，将连接放在池中并再次使用，这样就不必创建新的连接。如果所有连接都正在使用，则创建一个新连接并将其添加到池中。连接池还减少了用户必须等待创建与数据库的连接的时间。

uroj 采用 r2d2 作为数据库连接池，r2d2 是 rust 的一个通用连接池。r2d2 对于它所管理的连接类型是不可知的。ManageConnection 特性的实现者提供了数据库特

定的逻辑来创建和检查连接的健康状况。

在 uroj-db crate 中使用如下 create_connection_pool 函数创建一个连接池。在 uroj-api 和 uroj-runtime 使用本函数创建连接池。

```
use diesel::r2d2::{ConnectionManager, Pool};
use diesel::{pg::PgConnection, r2d2::PooledConnection};

pub type PgPool = Pool<ConnectionManager<PgConnection>>;
pub type Conn = PooledConnection<ConnectionManager<PgConnection>>;

pub fn create_connection_pool() -> PgPool {
    let url = env::var("DATABASE_URL").expect("Can't get DB URL");
    let manager = ConnectionManager::<PgConnection>::new(url);
    Pool::builder()
        .build(manager)
        .expect("Failed to create pool")
}
```

7 测试

杀杀杀